CONTENTS

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course File

BCS304 DATA STRUCTURES AND APPLICATIONS

III Sem B 2023-24

Faculty In-charge
BINDU K P
Assistant Professor
Dept. of Computer science and Engineering
K S School of Engineering & Management, Bangalore

K. S. SCHOOL OF ENGINEERING AND MANAGEMENT

VISION

To impart quality education in engineering and management to meet technological, business and societal needs through holistic education and research.

MISSION

K.S. School of Engineering and Management shall,

- Establish state-of-art infrastructure to facilitate effective dissemination of technical and Managerial knowledge.
- Provide comprehensive educational experience through a combination of curricular and Experiential learning, strengthened by industry-institute-interaction.
- Pursue socially relevant research and disseminate knowledge.
- Inculcate leadership skills and foster entrepreneurial spirit among students.

Department of Computer Science and Engineering

VISION

To produce quality Computer Science professional, possessing excellent technical knowledge, skills, personality through education and research.

MISSION

Department of Computer Science and Engineering shall,

- Provide good infrastructure and facilitate learning to become competent engineers who meet global challenges.
- Encourages industry institute interaction to give an edge to the students.
- Facilitates experimental learning through interdisciplinary projects.
- Strengthen soft skill to address global challenges.

DATA STRUCTURES AND APPLICATIONS		Semester	3
Course Code	BCS304	CIE Marks	50
Teaching Hours/Week (L: T:P: S)	3:0:0:0	SEE Marks	50
Total Hours of Pedagogy	40	Total Marks	100
Credits	03	Exam Hours	3
Examination type (SEE)	**Theory**		

Course objectives:

CLO 1. To explain fundamentals of data structures and their applications.
CLO 2. To illustrate representation of Different data structures such as Stack, Queues, Linked Lists, Trees and Graphs.
CLO 3. To Design and Develop Solutions to problems using Linear Data Structures
CLO 4. To discuss applications of Nonlinear Data Structures in problem solving.
CLO 5. To introduce advanced Data structure concepts such as Hashing and Optimal Binary Search Trees

Teaching-Learning Process (General Instructions)

Teachers can use following strategies to accelerate the attainment of the various course outcomes.

1. Chalk and Talk with Black Board
2. ICT based Teaching
3. Demonstration based Teaching

Module-1 **8Hours**

INTRODUCTION TO DATA STRUCTURES: Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations
Review of pointers and dynamic Memory Allocation,
ARRAYS and STRUCTURES: Arrays, Dynamic Allocated Arrays, Structures and Unions, Polynomials, Sparse Matrices, representation of Multidimensional Arrays, Strings
STACKS: Stacks, Stacks Using Dynamic Arrays, Evaluation and conversion of Expressions
Text Book: Chapter-1:1.2 Chapter-2: 2.1 to 2.7 Chapter-3: 3.1,3.2,3.6
Reference Book 1: 1.1 to 1.4

Module-2 **8Hours**

QUEUES: Queues, Circular Queues, Using Dynamic Arrays, Multiple Stacks and queues.
LINKED LISTS : Singly Linked, Lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials
Text Book: Chapter-3: 3.3, 3.4, 3.7 Chapter-4: 4.1 to 4.4

Module-3 **8Hours**

LINKED LISTS : Additional List Operations, Sparse Matrices, Doubly Linked List.
TREES: Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees.
Text Book: Chapter-4: 4.5,4.7,4.8 Chapter-5: 5.1 to 5.3, 5.5

Module-4 **8Hours**

TREES(Cont..): Binary Search trees, Selection Trees, Forests, Representation of Disjoint sets, Counting Binary Trees,
GRAPHS: The Graph Abstract Data Types, Elementary Graph Operations
Text Book: Chapter-5: 5.7 to 5.11 Chapter-6: 6.1, 6.2

Module-5 **8Hours**

HASHING: Introduction, Static Hashing, Dynamic Hashing
PRIORITY QUEUES: Single and double ended Priority Queues, Leftist Trees
INTRODUCTION TO EFFICIENT BINARY SEARCH TREES: Optimal Binary Search Trees
Text Book: Chapter 8: 8.1 to 8.3 Chapter 9: 9.1, 9.2 Chapter 10: 10.1

Course outcome (Course Skill Set)
At the end of the course the student will be able to:
CO 1. Explain different data structures and their applications.
CO 2. Apply Arrays, Stacks and Queue data structures to solve the given problems.
CO 3. Use the concept of linked list in problem solving.
CO 4. Develop solutions using trees and graphs to model the real-world problem.
CO 5. Explain the advanced Data Structures concepts such as Hashing Techniques and Optimal Binary Search Trees.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

Continuous Internal Evaluation:

- For the Assignment component of the CIE, there are 25 marks and for the Internal Assessment Test component, there are 25 marks.
- The first test will be administered after 40-50% of the syllabus has been covered, and the second test will be administered after 85-90% of the syllabus has been covered
- Any two assignment methods mentioned in the 22OB2.4, if an assignment is project-based then only one assignment for the course shall be planned. The teacher should not conduct two assignments at the end of the semester if two assignments are planned.
- For the course, CIE marks will be based on a scaled-down sum of two tests and other methods of assessment.

Internal Assessment Test question paper is designed to attain the different levels of Bloom's taxonomy as per the outcome defined for the course.

Semester-End Examination:
Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (**duration 03 hours**).

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored shall be proportionally reduced to 50 marks

Suggested Learning Resources:

Textbook:

1. Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed, Fundamentals of Data Structures in C, 2nd Ed, Universities Press, 2014

Reference Books:

1. Seymour Lipschutz, Data Structures Schaum's Outlines, Revised 1st Ed, McGraw Hill, 2014.
2. Gilberg & Forouzan, Data Structures: A Pseudo-code approach with C, 2nd Ed, Cengage Learning,2014.
3. Reema Thareja, Data Structures using C, 3rd Ed, Oxford press, 2012.
4. Jean-Paul Tremblay & Paul G. Sorenson, An Introduction to Data Structures with Applications, 2nd Ed, McGraw Hill, 2013
5. A M Tenenbaum, Data Structures using C, PHI, 1989
6. Robert Kruse, Data Structures and Program Design in C, 2nd Ed, PHI, 1996.

Web links and Video Lectures (e-Resources):

- http://elearning.vtu.ac.in/econtent/courses/video/CSE/06CS35.html
- https://nptel.ac.in/courses/106/105/106105171/
- http://www.nptelvideos.in/2012/11/data-structures-and-algorithms.html
- https://www.youtube.com/watch?v=3Xo6P_V-qns&t=201s
- https://ds2-iiith.vlabs.ac.in/exp/selection-sort/index.html
- https://nptel.ac.in/courses/106/102/106102064/
- https://ds1-iiith.vlabs.ac.in/exp/stacks-queues/index.html
- https://ds1-iiith.vlabs.ac.in/exp/linked-list/basics/overview.html
- https://ds1-iiith.vlabs.ac.in/List%20of%20experiments.html
- https://ds1-iiith.vlabs.ac.in/exp/tree-traversal/index.html
- https://ds1-iiith.vlabs.ac.in/exp/tree-traversal/depth-first-traversal/dft-practice.html
- https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_013501595428077568125 59/overview

Activity Based Learning (Suggested Activities in Class)/ Practical Based learning

- Role Play
- Flipped classroom
- Assessment Methods for 25 Marks (opt two Learning Activities)
  - Case Study
  - Programming Assignment
  - Gate Based Aptitude Test
  - MOOC Assignment for selected Module

K. S. SCHOOL OF ENGINEERING AND MANAGEMENT
BENGALURU-560109
TENTATIVE CALENDAR OF EVENTS: III ODD SEMESTER (2023-2024)
SESSION: NOV 2023 TO FEB 2024

Week No.	Month	Day						Days	Activities
		Mon	Tue	Wed	Thu	Fri	Sat		
1	NOV			**15***	16	17	**18DH**	3	**15*-Commencement of III sem**
2	NOV	20	21	22	23	24	25	6	25- Wednesday Time Table
3	NOV/DEC	27	28	29	**30H**	1	**2DH**	4	30- Kanakadasa Jayanti
4	DEC	4	5	6	7	**8**	9	6	9- Tuesday Time Table
5	DEC	11	12	13	14	15	**16DH**	5	
6	DEC	18	19	**20**	21	22	23	6	23- Monday Time Table
7	DEC	**25 H**	**26T1**	**27T1**	**28T1**	29	30 TA	5	25- Christmas 30 - Monday Time Table
8	JAN	1	2	3	4	5	**6DH**	5	
9	JAN	8 BV	9 ASD	**10 *FFB1**	11	12	13	6	10 - First Faculty Feed Back 13- Tuesday Time Table
10	JAN	**15 H**	16	17	18	19	**20DH**	4	15- Makara Sankranti
11	JAN	22	23	24	25	**26 H**	**27T2**	5	26 - Republic Day 27- Monday Time Table
12	JAN / FEB	**29T2**	**30T2**	31	**1**	2	**3DH**	5	
13	FEB	5	6	7	8	9	10	6	~~Wednesday~~ Monday
14	FEB	12	13	14	15	**16*FFB T3**	**17DH**	5	16 - Second Faculty Feed Back
15	FEB	**19 BV T3**	**20*ASD T3**					2	**20* - Last working Day**
Total No of Working Days : 73									

Total Number of working days (Excluding holidays and Tests)=63

H	Holiday
BV	Blue Book Verification
T1,T2	Tests 1,2
ASD	Attendance & Sessional Disnlay
DH	Declared Holiday
LT	Lab Test
TA	Test attendance

Monday	13
Tuesday	14
Wednesday	13
Thursday	11
Friday	12
Total	**63**

SIGNATURE OF PRINCIPAL
Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Management
Bengaluru - 560 109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU-560109

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SESSION: 2023-2024(ODD SEMESTER)

(w. e. f : 15/11/2023)

INDIVIDUAL TIME TABLE

Class: III A & B **Faculty Name: Mrs. Bindu K P**

DAY	8.40-9.35	9.35-10.30	10.30 -10.45	10.45 -11.40	11.40-12.35	12.35-1.20	1.20 -2.10	2.10-3.00	3.00-3.50
MONDAY	IOT (B)	DSC Lab Batch - B1				LUNCH BREAK			
TUESDAY		DSC (III B)	TEA BREAK		IOT (B)		DSC Lab Batch - B2		
WEDNESDAY	DSC (III B)			DSC (III B)	IOT (B)		DSC Lab Batch - A1		
THURSDAY	DSC (III B)						DVP	DVP Lab Batch - A1	
FRIDAY		IOT (B)					DSC Lab Batch - A2		
SATURDAY	AS PER CALENDAR OF EVENTS								

CODE	SUBJECT	Hours /Week	
BCS304	Data Structures and Application	4	Mrs. Bindu K P
BCSL305	Data Structures Laboratory	4	
BCS358D	Data Visualization with Python Laboratory	4	
BETCK105H	Introduction to Internet of Things	4	
18CSP77	Project Work Phase -1	1.5	

Time table Coordinator

Head of the Department

Department of Computer Science Engg
K.S School of Engineering & Management
Bangalore-560109

Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Managem

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU-560109
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SESSION: 2023-2024 (ODD SEMESTER)
CLASS TIME TABLE
(w.e.f. 25/10/2023)

Class: III CSE 'B' | Lecture Hall: LH- 405 | Class Teacher: Mrs. Bindu K P

DAY	8.40-9.35	9.35-10.30	10.30-10.45	10.45 -11.40	11.40-12.35	12.35-1.20	1.20 -2.10	2.10-3.00	3.00-3.50
MONDAY	DDCO (BCS302)	DSC Lab Batch - B1 DDCO Lab Batch - B2				LUNCH BREAK	DVP (BCS358D)	DVP Lab Batch - B1 OOPS with Java Lab Batch - B2	
TUESDAY	OS (BCS303)	DSC (BCS304)	TEA BREAK	MAT- III (BCS301)	DDCO (BCS302)		DSC Lab Batch - B2 DDCO Lab Batch - B1		
WEDNESDAY	DSC (BCS304)	OS (BCS303)		DSC (BCS304)	MAT- III (BCS301)		OOPS (BCS306A)	OS Lab Batch - B1 SCR-B2	
THURSDAY	DSC (BCS304)	OS (BCS303)		DVP Lab Batch - B2 OOPS with Java Lab Batch - B1			OOPS (BCS306A)	MAT- III (BCS301)	DDCO (BCS302)
FRIDAY	DDCO (BCS302)	OOPS (BCS306A)		OS (BCS303)	MAT- III (BCS301)		OOPS (BCS306A)	OS Lab Batch - B2 SCR-B1	
SATURDAY	AS PER CALENDAR OF EVENTS								

CODE	SUBJECT	HOURS /WEEK	STAFF
BCS301	Mathematics for Computer Science	4	Dr. Lakshmi B
BCS302	Digital Design & Computer Organization	4	Ms. Nethravathi K G
BCS302	Digital Design & Computer Organization Laboratory	4	Ms. Nethravathi K G & Mrs. Jayashubha J
BCS303	Operating Systems	4	Mrs. Chhaya S Dule
BCS303	Operating Systems Laboratory	2	Mrs. Chhaya S Dule
BCS304	Data Structures and Application	4	Mrs. Bindu K P
BCSL305	Data Structures Laboratory	3	Mrs. Bindu K P & Mrs. Kavitha K S
BSCK307	Social Connect and Responsibility	2	Mrs. Sougandhika Narayan
BCS306A	OOPS with Java	4	Mrs. Anujna M
BCS306A	OOPS with Java Laboratory	2	Mrs. Anujna M & Mrs. Chhaya S Dule
BCS358D	Data Visualization with Python Laboratory	3	Mrs. Jayashubha J & Mrs. Bindu K P

Time-table Coordinator

Head of the Department
Department of Computer Science & Engineering
K.S School of Engineering & Management
Bangalore-560109

Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Management
Bengaluru - 560 109

K. S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU -560 019

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (ODD SEMESTER)

III Semester - B Student List

Sl. No.	USN	Name of the Student
1	1KG22CS064	MAHALAKSHMI P S
2	1KG22CS065	MANSI M
3	1KG22CS066	MARINENI HANSIKA
4	1KG22CS067	MARPURI SOWMYA
5	1KG22CS068	MOHAMMAD SAMI SHAKIL AHMAD SAYYAD
6	1KG22CS069	MOHITH REDDY K
7	1KG22CS070	MONIKA K
8	1KG22CS071	MUMMANEDI MEGHANA
9	1KG22CS072	MUTHULURI DHEEKSHITH
10	1KG22CS073	NISHANTH R
11	1KG22CS074	NISWANA N SWAMY
12	1KG22CS075	NITHISH KUMAR V
13	1KG22CS076	NIVEDA B
14	1KG22CS077	P HARSHITHA
15	1KG22CS078	PADMASHREE M M
16	1KG22CS079	PAPANI DEVISH CHOWDARY
17	1KG22CS080	PAVAN KUMAR
18	1KG22CS081	PAVAN T L
19	1KG22CS082	PAVAN U
20	1KG22CS083	PEDDINTI MOHAMMAED
21	1KG22CS084	POOJA S
22	1KG22CS085	POOJITHA S
23	1KG22CS086	PRAGNA P S
24	1KG22CS087	PRAJWALKOUSHIK C
25	1KG22CS088	PRANAV RAMESH
26	1KG22CS089	PRAPUL U
27	1KG22CS090	PRIYA R K
28	1KG22CS091	PUNITH B
29	1KG22CS092	R PRUDVI GANESH
30	1KG22CS093	RAGHU KISTHANNAVAR
31	1KG22CS094	RAKESH V
32	1KG22CS095	RAKSHITHA N
33	1KG22CS096	RAKSHITHA S
34	1KG22CS097	RAKSHITHA S
35	1KG22CS098	RAMITHA K
36	1KG22CS099	RAMYA P
37	1KG22CS100	RANI
38	1KG22CS101	RAYAN NADEEM
39	1KG22CS102	SADHVIKA GODAVARTHI
40	1KG22CS103	SAKESH P
41	1KG22CS104	SANJANA B
42	1KG22CS105	SANJAY M D
43	1KG22CS106	SANJAY S

44	1KG22CS107	SANTOSH KUMAR NAGUR
45	1KG22CS108	SARAN R
46	1KG22CS109	SHASHANK
47	1KG22CS110	SHASHANK D URS
48	1KG22CS111	SINDHUSHREE K
49	1KG22CS112	SOWJANYA K S
50	1KG22CS113	SUHAS S
51	1KG22CS114	T KAVYA
52	1KG22CS115	TAANISH M
53	1KG22CS116	TARUN R
54	1KG22CS117	TEJASWINI R M
55	1KG22CS118	THIRUVIDULA ABHISHEK
56	1KG22CS119	TIRUCHANURU VENKATA PRANEETH
57	1KG22CS120	TOLUCHURU HARITHA
58	1KG22CS121	UDAY KIRAN
59	1KG22CS122	VANDANA BASAVARAJ PATIL
60	1KG22CS123	VINAYAK C
61	1KG22CS124	VISMAYA N
62	1KG22CS125	YASHWANTH R
63	1KG21CS020	BHOOMIKA P DESAI
64	Lateral Students	VENKATACHAL S
65		RAJESH P C
66		Suhas Madhusudan Shandilya

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU- 560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (ODD SEMESTER)

LESSON PLAN

NAME OF THE STAFF : **Mrs. Bindu K P**

SUBJECT CODE/NAME : BCS304 / Data Structures and Applications

SEMESTER/ SEC/YEAR : III / B / II (CSE)

Sl. No	Topic to be covered	Mode of Delivery	Teaching Aid	No. of Periods	Cumulative No. of Periods	Proposed Date	Engaged Date
	MODULE 1						
1	**INTRODUCTION TO DATA STRUCTURES:** Data Structures, Classifications (Primitive& Non-Primitive), Data structure Operations	L+D	LCD	1	1	15/11/23	15/11/23
2	**Review of** pointers and dynamic Memory Allocation	L+D	LCD	1	2	15/11/23	15/11/23
3	**ARRAYS and STRUCTURES:** Arrays, Dynamic Allocated Arrays	L+D	LCD	1	3	16/11/23	16/11/23
4	Structures and Unions	L+D	LCD	1	4	21/11/23	21/11/23
5	Polynomials, Sparse Matrices	L+D	LCD	1	5	22/11/23	22/11/23
6	Representation of Multidimensional Arrays, Strings	L+D	LCD	1	6	22/11/23	22/11/23
7	**STACKS:** Stacks, Stacks Using Dynamic Arrays	L+D	LCD	1	7	23/11/23	23/11/23
8	Evaluation and conversion of Expressions	L+D	LCD	1	8	25/11/23	25/11/23
9	Tutorial	L+D	LCD	3	3	25/11/23 28/11/23 29/11/23	26/11/23 to 29/11/23

MODULE 2							
10	**QUEUES:** Queues	L+D	LCD	1	9	29/11/23	5/12/23
11	Circular Queues, Using Dynamic Arrays	L+D	LCD	1	10	5/12/23	5/12/23
12	Multiple Stacks and queues	L+D	LCD	1	11	6/12/23	6/12/23
13	**LINKED LISTS :** Singly Linked	L+D	LCD	1	12	6/12/23	6/12/23
14	Lists and Chains	L+D	LCD	1	13	7/12/23	18/12/23
15	Representing Chains in C	L+D	LCD	1	14	9/12/23	19/12/23
16	Linked Stacks and Queues	L+D	LCD	1	15	12/12/23	19/12/23
17	Polynomials	L+D	LCD	1	16	13/12/23	20/12/23
18	Tutorial	L+D	LCD	3	3	13/12/23 14/12/23 19/12/23	20/12/23 to 22/12/23
MODULE 3							
19	**LINKED LISTS :** Additional List Operations	L+D	LCD	1	17	20/12/23	28/12/23
20	Sparse Matrices	L+D	LCD	1	18	20/12/23	30/12/23
21	Doubly Linked List	L+D	LCD	1	19	21/12/23	9/1/24
22	Doubly Linked List	L+D	LCD	1	20	2/1/24	10/1/24
23	**TREES:** Introduction	L+D	LCD	1	21	3/1/24	10/1/24
24	Binary Trees	L+D	LCD	1	22	3/1/24	11/1/24
25	Binary Tree Traversals	L+D	LCD	1	23	4/1/24	13/1/24
26	Threaded Binary Trees	L+ D	LCD	1	24	9/1/24	16/1/24
27	Tutorial	L+D	LCD	2	2	10/1/24 10/1/24	17/1/24

	MODULE 4						
28	**TREES(Cont..):** Binary Search trees	L+D	LCD	1	25	11/1/24	18/1/24
29	Selection Trees	L+D	LCD	1	26	13/1/24	22/1/24
30	Forests	L+D	LCD	1	27	16/1/24	24/1/24
31	Representation of Disjoint sets	L+D	LCD	1	28	17/1/24	24/1/24
32	Counting Binary Trees	L+D	LCD	1	29	17/1/24	25/1/24
33	**GRAPHS:** The Graph Abstract Data Types	L+D	LCD	1	30	18/1/24	30/1/24
34	Elementary Graph Operations	L+D	LCD	1	31	23/1/24	31/1/24.
35	Elementary Graph Operations	L+D	LCD	1	32	24/1/24	6/2/24
36	Tutorial	L+D	LCD	2	2	24/1/24 25/1/24	13/2/24
	MODULE 5						
37	**HASHING:** Introduction	L+D	LCD	1	33	31/1/24	20/2/24
38	Static Hashing	L+D	LCD	1	34	31/1/24	20/2/24
39	Dynamic Hashing	L+D	LCD	1	35	1/2/24	21/2/24
40	**PRIORITY QUEUES:** Single Priority Queue	L+D	LCD	1	36	6/2/24	21/2/24
41	Double ended Priority Queues	L+D	LCD	1	37	7/2/24	21/2/24
42	Leftist Trees	L+D	LCD	1	38	7/2/24	22/2/24
43	**INTRODUCTION TO EFFICIENT BINARY SEARCH TREES:** Optimal Binary Search Trees	L+D	LCD	1	39	8/2/24	22/2/24
44	Optimal Binary Search Trees	L+D	LCD	1	40	13/2/24	27/2/24
45	Revision	L+D	LCD	3	3	14/2/24 14/2/24 15/2/24	27/2/24 28/2/24 29/2/24

	Week	Remarks
Assignment 1	4th Week – 14/12/23	Mode of Assignment – Written Assignment
Assignment 2	9th Week- 18 /1 /24	

Total No. of Lecture Hours = 40
Total No. of Tutorial Hours =13

Course in charge

Head of the Department

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

IQAC Coordinator

Principal

Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Manage
Bengaluru - 560 109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU-560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (ODD SEMESTER)

DATA STRUCTURES AND APPLICATIONS (BCS304)

Question bank-1

Module-1

1. **Define** data structures. With a neat diagram, **explain** the different classifications of Data structures.
2. **Define** structure. **How** it is represented in C language. Give an example program using structures.
3. **Differentiate** structure and union.
4. **What** are self-referential structures? Give one example.
5. **Define** nested structure. **Give** 2 ways of declaring nested structure with example program.
6. **Give** Abstract Data Type(ADT) for arrays. How array can be declared and initialized?
7. **Define** Array. **Explain** with program, the operations performed on array.
8. Assume each student in a class of 25 students is given 4 test . Assume the students are numbered from 1 to 25 and the test scores are assigned in the 25*4 matrix called score. Assume base of score=200 ,W=4 and the programming language uses row major order and column major order to store this 2D array then **find** the address of 3rd test of 12th student (i.e)score[13,3] in both row major order and column major order .
9. **Develop** C programs to perform each of the following (i) linear search (ii) binary search (iii)bubble sort taking a array of 'n' integers.
10. **Develop** an algorithm for Bubble sort and **Apply** for the following data 40,3,20,35,50,31,5,6
11. Which are the 4 inbuilt functions to perform dynamic memory allocation. **Discuss** the importance of Dynamic memory allocation. Write a C program to create an array dynamically.
12. **Develop** a C function to create 2d array dynamically. Use MALLOC macro.
13. **Illustrate** with an example how sparse matrix is efficiently stored in triple format. Write its C representation.
14. **Define** String. Write a C program to perform pattern matching.
15. **Develop** a function to perform polynomial addition.
16. **Find** the table and corresponding graph for the second pattern matching algorithm where the Pattern is ababab

17. **Define** Stack. List the operations performed on Stack.
18. **Develop** a c program to demonstrate various stack operations, including cases for overflow and underflow of STACK.
19. **Develop** an algorithm to convert infix expression to postfix expression.
20. Convert the following expression to postfix using stack (a+b)*((b^c)*f)/g
21. **Define** Recursion. **Illustrate** with an example how stack is used in recursion.
22. **Define** Recursion and **Evaluate** A(1,3) using Ackermann's function.
23. **Develop** an algorithm for Evaluation of postfix expression.
24. **Develop** a C function to **evaluate** postfix expression.
25. **Develop** a Recursive C program for each of the following:
    a. Tower of Hanoi
    b. Computing GCD of two numbers
    c. Fibonacci series
    d. To compute factorial of N

Module-2

1. **Define** Queue. Implement the operations of the queue using arrays. **Apply** the same on job scheduling.
2. **Show** how queues are represented using arrays?
3. **Explain** queues operations using dynamic arrays.
4. Give the disadvantage of the ordinary queue and how it is solved in a circular queue. **Explain** with a suitable example of how you would implement a circular queue using a dynamically allocated array.
5. **What** is a circular queue? **Explain** how it is different from the linear queue.
6. **Discuss** the following:
    1) Double Ended Queue
    2) Priority Queue

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU-560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (ODD SEMESTER)

DATA STRUCTURES AND APPLICATIONS (BCS304)

Question bank-2

Module-2

Queues

1. Define Queue. Implement the operations of the queue using arrays. Apply the same on job scheduling.
2. Show how queues are represented using arrays?
3. Explain queues operations using dynamic arrays.
4. Give the disadvantage of the ordinary queue and how it is solved in a circular queue. Explain with a suitable example of how you would implement a circular queue using a dynamically allocated array.
5. What is a circular queue? Explain how it is different from the linear queue.

Linked List

1. Describe doubly linked list with advantages and disadvantages. Write a C function to delete a node from a doubly linked list. Ptr is the pointer which points to the node to be deleted.
2. Define linked list. Write the representation of linked lists in memory.
3. How the nodes are represented using C?
4. Explain linked list operation with examples.
5. Write a note on a header linked list.
6. Briefly explain linked stack and queue.
7. Apply a linked list to represent two polynomials and write a function to add the polynomials using the linked list.

Module-3

Linked List

1. What is doubly linked list. Write the declaration of doubly linked list in C.
2. With the C program explain how the elements are inserted and deleted from a doubly linked list
3. List out any two applications of the linked list and any two advantages of doubly linked list over the singly linked list.
4. Write a short note on circular lists. Write a function to insert a node at the front and rear end in a circular linked list. Write down the sequence of steps to be followed.
5. Write the following functions for singly linked list: i) Reverse the list ii)Concatenate two lists.
6. What is a linked list? Explain the different types of linked lists with a diagram. Write C program to implement the insert and delete operation on a queue using a linked list.
7. Explain the sparse matrix using Linked list. Write a node structure for linked representation and apply it for the following Matrix

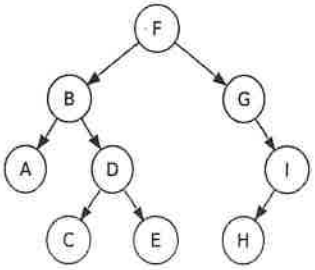
$$A= \begin{pmatrix} 10 & 25 & 0 & 0 & 0 \\ 0 & 23 & 0 & 45 & 0 \\ 0 & 0 & 0 & 0 & 32 \\ 42 & 0 & 0 & 31 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 30 & 0 & 0 \end{pmatrix}$$

Trees

1. **Define** the following and **illustrate** with suitable examples

   i) Binary tree ii) Full binary tree iii) Almost complete binary tree

   iv) Strict binary tree v) Skewed binary tree
2. **Develop** an algorithm for binary tree traversal techniques and **apply** the same for the following tree



3. **Demonstrate** the Array and Linked representation of binary tree with suitable examples.
4. **Explain** insertion in to a Threaded binary tree with neat diagram and also **develop** a C function to do the inorder traversal of a threaded binary tree.
5. **Illustrate** Threaded binary tree and their representation with suitable neat diagram.

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

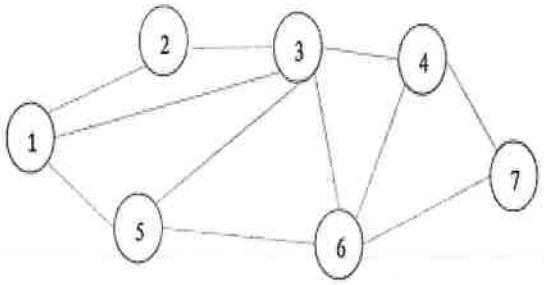
SESSION: 2023-2024 (ODD SEMESTER)

DATA STRUCTURES AND APPLICATIONS (BCS304)

Question Bank-3

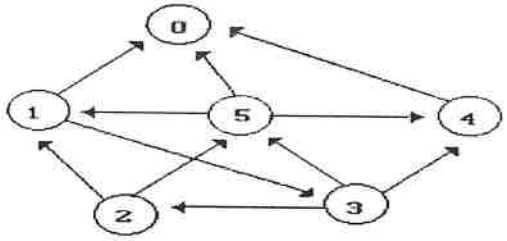
Module-4&5

1. **Construct** a binary search tree for the given data100, 85, 45, 55, 110, 20, 70, 65. **Develop** a C function to inserting an element into Binary search tree.
2. **Develop** a C function for DFS and **apply** it for the following graph



3. **Define** Selection Tree. **Illustrate** Winner Tree and Loser Tree with suitable example.
4. **Define** Binary search tree. **Construct** a binary search tree by using the following preorder and in-order traversals:
   Preorder : ABCDEFGHI
   Inorder : BCAEDGHFI
5. **Define** graph. **Illustrate** the following with suitable examples
   a. Adjacency matrix
   b. Adjacency List
   c. Adjacency Multilist
6. **Explain** Union and Find operations. **Develop** Union function using Weighting rule **illustrate** with suitable example.
7. **Define** Hashing? **Illustrate** different Hashing functions with an example.
8. **Define** Leftist Tree. **Demonstrate** Height biased leftist tree and Weight biased leftist tree.
9. **Define** collision? **List** different methods to resolve collision? **Demonstrate** linear probing with an example.
10. **Define** Hashing? **Illustrate** Dynamic Hashing with an example.

11. **Define** Binary Search Tree. **Develop** a C function to perform searching in a Binary search Tree.
12. **Explain** Selection Tree. Construct Winner Tree and Loser Tree for the following 16, 9, 20, 6, 50, 11, 90, 18
13. For the given graph **show** the adjacency matrix and adjacency list representation



14. **Develop** a C routine to print the reachable nodes of a graph from the source node using Breadth First Search and also **illustrate** with a suitable graph.
15. **Illustrate** with suitable example
    a) Graph b) Connected components
    c) Spanning Tree d) Biconnected components
16. **Define** collision. **List** different overflow handling techniques? **Demonstrate** chaining with an example
17. **Explain** motivation for dynamic hashing. **Illustrate** dynamic hashing using directories and directoryless dynamic hashing.
18. **Develop** a search function for linear probing. **Apply** division method and solve collision (if any) by using linear probing. Given key elements are: 72, 27, 36, 24, 63, 81, 92, 101 size of the hash table is 10.

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU - 560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CO-PO Mapping

Course: Data Structures and Applications				
Type: Professional Core Course			Course Code: BCS304	
No of Hours				
Theory (Lecture Class)	Tutorials	Practical/Field Work/Allied Activities	Total/Week	Total hours of Pedagogy
3	0	0	3	40

Marks			
CIE	SEE	Total	Credits
50	50	100	3

Aim/Objectives of the Course

1. To explain fundamentals of data structures and their applications
2. To illustrate representation of data structures: Stack, Queues, Linked Lists, Trees and Graphs.
3. To design and develop solutions to problems using linear Data Structures.
4. To discuss applications of Nonlinear Data Structures in problem solving
5. To introduce advanced Data structure concepts such as Hashing and Optimal Binary Search Trees.

Course Learning Outcomes

After completing the course, the students will be able to

CO1	**Apply** the basic data structures concepts such as arrays, structures, unions, pointers, strings and dynamic memory allocation function to solve simple problems. **Make use of** stacks to evaluate mathematical expression.	**Applying (K3)**
CO2	**Apply** the concept of queues and linked list in problem solving.	**Applying (K3)**
CO3	**Utilize** linked list for implementation of list operations, doubly linked list and sparse matrix, and **apply** tree traversal method, threaded binary tree.	**Applying (K3)**
CO4	**Make use of** binary search tree, selection trees and forests and graph to solve real world problems.	**Applying (K3)**
CO5	**Analyze** advanced Data Structures concepts such as Hashing Techniques and OptimalBinary Search Trees.	**Applying (K3)**

Syllabus Content

MODULE 1 : INTRODUCTION TO DATA STRUCTURES: Data Structures, Classifications (Primitive & Non-Primitive), Data structure Operations **Review of** pointers and dynamic Memory Allocation, **ARRAYS and STRUCTURES:** Arrays, Dynamic Allocated Arrays, Structures and Unions, Polynomials, Sparse Matrices, representation of Multidimensional	**CO1** 8 hrs PO1-1 PO2-3 PO3-3

Arrays, Strings. **STACKS:** Stacks, Stacks Using Dynamic Arrays, Evaluation and conversion of Expressions. **LO:** At the end of this session the student will be able to 1. Understand the basic data structures concepts. 2. Analyze the stack operations, dynamic memory allocation and Structures. 3. Understand the sparse matrix and evaluation and conversion of expressions.	PO4-3 PO6-1 PO12 -1 PSO1-3 PSO2-1
MODULE 2 : QUEUES: Queues, Circular Queues, Using Dynamic Arrays, Multiple Stacks and queues. **LINKED LISTS :** Singly Linked, Lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials **LO:** At the end of this session the student will be able to 1. Analyze Queue operations using dynamic arrays. 2. Understand the concepts of linked list and chains. 3. Solve simple problems on linked list such as polynomials.	**CO2** 8 hrs. PO1-1 PO2-3 PO3-3 PO4-3 PO6-1 PO12-1 PSO1-3 PSO2-1
MODULE 3: LINKED LISTS : Additional List Operations, Sparse Matrices, Doubly Linked List. **TREES:** Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees. **LO:** At the end of this session the student will be able to 1. Understand the concepts of doubly linked list and Trees terminologies. 2. Solve binary tree traversals. 3. Solve simple problems on linked list such as sparse matrix.	**CO3** 8 hrs PO1-1 PO2-3 PO3-3 PO4-3 PO6-1 PO12-1 PSO1-3 PSO2-1
MODULE 4: TREES(Cont..): Binary Search trees, Selection Trees, Forests, Representation of Disjointsets, Counting Binary Trees, **GRAPHS:** The Graph Abstract Data Types, Elementary Graph Operations **LO:** At the end of this session the student will be able to 1. Understand the Binary Search trees, Forests and counting Binary trees. 2. Understand the graph terminologies. 3. Analyze elementary Graph operations	**CO4** 8 hrs PO1-1 PO2-3 PO3-3 PO4-3 PO6-1 PO12-1 PSO1-3 PSO2-1
MODULE 5: HASHING: Introduction, Static Hashing, Dynamic Hashing **PRIORITY QUEUES:** Single and double ended Priority Queues, Leftist Trees **INTRODUCTION TO EFFICIENT BINARY SEARCH TREES:** Optimal Binary Search Trees. **LO:** At the end of this session the student will be able to 1. Understand hashing technique.	**CO5** 8hrs PO1-1 PO2-3 PO3-3 PO4-3 PO6-1

2. Analyze Single and double ended priority queues, leftist trees. 3. Understand Optimal Binary search Trees.	PO12-1 PSO1-3 PSO2-1

Text Books

1. Ellis Horowitz and Sartaj Sahni, **Fundamentals of Data Structures in C**, 2nd Ed, Universities Press, 2014.

Reference Books (specify minimum two foreign authors text books)

1. Seymour Lipschutz, **Data Structures Schaum's Outlines**, Revised 1st Ed, McGraw Hill, 2014.
2. Gilberg & Forouzan, Data Structures: **A Pseudo-code approach with C**, 2nd Ed, Cengage Learning,2014.
3. Reema Thareja, **Data Structures using C**, 3rd Ed, Oxford press, 2012.
4. Jean-Paul Tremblay & Paul G. Sorenson, **An Introduction to Data Structures with Applications**, 2nd Ed, McGraw Hill, 2013
5. A M Tenenbaum, **Data Structures using C**, PHI, 1989
6. Robert Kruse, **Data Structures and Program Design in C**, 2nd Ed, PHI, 1996.

Useful Websites

- http://elearning.vtu.ac.in/econtent/courses/video/CSE/06CS35.html
- https://nptel.ac.in/courses/106/105/106105171/
- http://www.nptelvideos.in/2012/11/data-structures-and-algorithms.html
- https://www.youtube.com/watch?v=3Xo6P_V-qns&t=201s
- https://ds2-iiith.vlabs.ac.in/exp/selection-sort/index.html
- https://nptel.ac.in/courses/106/102/106102064/
- https://ds1-iiith.vlabs.ac.in/exp/stacks-queues/index.html
- https://ds1-iiith.vlabs.ac.in/exp/linked-list/basics/overview.html
- https://ds1-iiith.vlabs.ac.in/List%20of%20experiments.html
- https://ds1-iiith.vlabs.ac.in/exp/tree-traversal/index.html
- https://ds1-iiith.vlabs.ac.in/exp/tree-traversal/depth-first-traversal/dft-practice.html
- https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_013501595428077568125 59/overview

Teaching and Learning Methods

1. Lecture class: 40 hrs

Assessment Details (both CIE and SEE):

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

Continuous Internal Evaluation:

- For the Assignment component of the CIE, there are 25 marks and for the Internal Assessment Test component, there are 25 marks.
- The first test will be administered after 40-50% of the syllabus has been covered, and the second test will be administered after 85-90% of the syllabus has been covered
- Any two assignment methods mentioned in the 22OB2.4, if an assignment is project-based then only one assignment for the course shall be planned. The teacher should not conduct two assignments at the end of the semester if two assignments are planned.
- For the course, CIE marks will be based on a scaled-down sum of two tests and other methods of assessment.

Internal Assessment Test question paper is designed to attain the different levels of Bloom's taxonomy as per the outcome defined for the course.

Semester-End Examination:

Theory SEE will be conducted by the University as per the scheduled timetable, with common question papers for the course (duration 03 hours).
1. The question paper will have ten questions. Each question is set for 20 marks. 2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), should have a mix of topics under that module.
3. The students have to answer 5 full questions, selecting one full question from each module. Marks scored shall be proportionally reduced to 50 marks.

CO to PO Mapping

PO1: Science and engineering Knowledge **PO2:** Problem Analysis **PO3:** Design & Development **PO4:**Investigations of Complex Problems **PO5:** Modern Tool Usage **PO6:** Engineer & Society	**PO7:**Environment and Society **PO8:**Ethics **PO9:**Individual & Team Work **PO10:** Communication **PO11:**Project Mgmt. & Finance **PO12:** Lifelong Learning

PSO1: Understand fundamental and advanced concepts in the core areas of Computer Science and Engineering to analyze, design and implement the solutions for the real world problems.

PSO2: Utilize modern technological innovations efficiently in various applications to work towards the betterment of society and solve engineering problems.

CO	PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO 1	PSO 2
BCS304	K-level														
CO1	K3	1	3	3	3	-	1	-	-	-	-	-	1	3	1
CO2	K3	1	3	3	3	-	1	-	-	-	-	-	1	3	1
CO3	K3	1	3	3	3	-	1	-	-	-	-	-	1	3	1
CO4	K3	1	3	3	3	-	1	-	-	-	-	-	1	3	1
CO5	K3	1	3	3	3	-	1	-	-	-	-	-	1	2	1

Course In charge

HOD-CSE

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

IQAC Coordinator

Principal

Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Management
Bengaluru - 560 109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (ODD SEMESTER)

FIRST ASSIGNMENT

Degree	:	**B.E**	**Semester**	:	**III A&B**
Branch	:	**CSE**	**Course Code**	:	**BCS304**
Course Title	:	**Data Structures and Applications**	**Max Marks**	:	**25**
Date	:	**14/12/2023**	**Submission Date**	:	**22/12/2023**

Q No.	Questions	Marks	K-Level	CO mapping
1	**a) Define** Data structures. Give its classification and **explain** in brief. **What** are the basic operations that can be performed on Data structure? **b) Explain** the dynamic memory allocation functions in detail with example. **Differentiate** between malloc() and calloc() functions. **c) Define** linear array? **Develop** a C program for the following array operations i) Inserting an element at the given valid position ii) Deleting an element at a given valid position iii) Display of array elements iv) Exit Support the program with functions for each of the above operations. **d) Define** pointers? **How** to declare and initialize pointer? **Justify** how pointers can be dangerous.	5	**Applying K3**	**CO1**
2	**a) Develop** the algorithm of first Pattern matching Algorithm and Knuth Morris Pratt Pattern Matching Algorithm and **Apply** both on the following data T=abcaabaaabcaaabbc P1=aaabb and P2=aaa **b) Define** Stack. **Discuss** how to represent stack using dynamic arrays. **Develop** a c program to demonstrate various stack operations, including cases for overflow and underflow of STACK. **c) Develop** an algorithm for Evaluation of Postfix expression and **evaluate** the following expressions i)23^522^*+126/- ii)12+3-21+3^+	5	**Applying K3**	**CO1**

	d) Develop an algorithm to convert from Infix to Postfix expression. **Apply** the same for the following expressions i) A+(B*C-(D/E^F)*G)*H ii) a/b-c+d*e+a*c			
3	**a) Define** two ways to represent polynomial in C and **show** the structural representation for the given 2 polynomials, $A(x)=4x^{15}+3x^4+5$ and $B(x)=2x^{1000}+10$. **Develop** a C function to add 2 polynomials. **b) Explain** ADT of the polynomial. **c) Explain** ADT of sparse matrix. **d) Develop** a C function for Fast transpose of Sparse Matrix. **Identify** the triplet form of Sparse matrix and **find** the transpose of the given Matrix a) $\begin{pmatrix} 10 & 0 & 0 & 25 & 0 \\ 0 & 23 & 0 & 0 & 45 \\ 0 & 0 & 0 & 0 & 32 \\ 42 & 0 & 0 & 31 & 0 \\ 0 & 0 & 32 & 0 & 0 \end{pmatrix}$ b) $\begin{pmatrix} 0 & 10 & 0 & 20 & 0 \\ 30 & 0 & 0 & 0 & 40 \\ 0 & 50 & 0 & 0 & 0 \\ 0 & 0 & 60 & 0 & 0 \end{pmatrix}$	**5**	**Applying K3**	**CO1**
4	**a) Define** queue. **Explain** the operations performed on queue. **Discuss** dequeue. **b) Discuss** the disadvantage of the ordinary queue and how it is solved using a circular queue? **Develop** insertion and deletion functions for circular queue.	**5**	**Applying K3**	**CO2**
5	**a) Define** priority queue. **Explain** in detail One-Way list representation of a Priority Queue with an example. **b) Explain** with a suitable example of **how** you would implement a circular queue using a dynamically allocated array.	**5**	**Applying K3**	**CO2**

Course In charge

HOD

HOD
Department of Computer Science Engineering
K.S Sch... Ingeer... ...gement

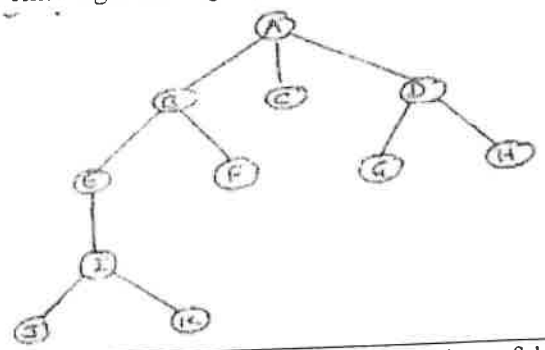
K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

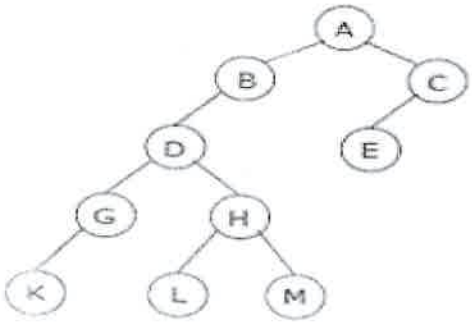
SESSION: 2023-2024 (ODD SEMESTER)

SECOND ASSIGNMENT

Degree	:	B.E	Semester	:	III A&B
Branch	:	CSE	Course Code	:	BCS304
Course Title	:	Data Structures and Applications	Max Marks	:	25
Date	:	22/01/2024	Last Date for submission	:	27/01/2024

Q No.	Questions	Marks	K-Level	CO mapping
1	a) **Define** Recursion **Evaluate** A(1,3) using Ackerman's function. **Develop** a C function for Tower of Hanoi. b) **Define** linked list. **Illustrate** the representation of linked lists in memory. c) **Differentiate** between array and Linked List. d) **Write a note on** a header linked list e) **Explain** linked stack and queue.	5	Applying K3	CO2
2	a) **Construct** the node structure to create a linked of integers and write C functions to perform the following: i. Create a three-node list with data 10,20 and 30. ii. Insert a node with data value 15 in between the nodes having data values 10 and 20. iii. Delete the node which is followed by a node whose data value is 20. iv. Display the resulting singly linked list. b) **Develop** a function for addition of two polynomials using linked List and Consider the given 2 polynomials, a= $3x^2 + 2x+1$ and b=$5x^2-x+2$. Represent the polynomials using Linked list. c) **Define** a linked list? **Explain** the different types of linked lists with a diagram. d) **Develop** C program to implement the insert and delete operation of stack and queue using a single linked list.	5	Applying K3	CO2

3	**a) Develop** the following functions for singly linked list(chains): i. Reverse/Invert the list ii. Concatenate two lists. iii. Search an element in the list **b) Explain** the advantage of Doubly linked list over Singly Linked list. **Develop** a C function for the following operations on DLL i. Insert at front ii. Insert at last, iii. Delete at front iv. Delete at end v. Display and count number of nodes **c) Develop** a C function for the following operations on circular linked list i. Insert at front ii. Insert at rear iii. Finding the length of a circular list **d) Describe** Doubly linked list with advantages and disadvantages. **Develop** a C function to delete a node from a Circular Doubly linked list with the header node.	5	**Applying K3**	**CO3**
4	**a) Explain** the concept of sparse matrix using Linked list. Write a node structure for linked representation and apply it for the following matrix $A = \begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$ **b) Define** the following and **illustrate** with suitable examples i) Binary tree ii) Full binary tree iii) Almost complete binary tree iv) Strict binary tree v) Skewed binary tree **c) Define** Tree. Represent the below given tree using i. Linked list representation with parenthetical notation ii. Left-child right-sibling representation 	5	**Applying K3**	**CO3**
5	**a) Describe** the Array and Linked representation of binary tree with suitable examples. **b) Define** Binary tree with an example. Develop a C recursive routine to	5	**Applying K3**	**CO3**

	traverse the given tree using in-order, pre-order, post-order and level order  c) **Explain** Threaded binary tree and their representation with neat diagram and also **develop** a C function to do the inorder traversal of a threaded binary tree.			

Course In charge

HOD

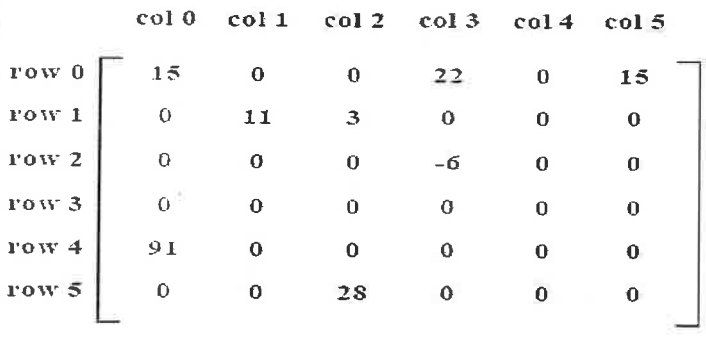
HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SESSION: 2023-2024 (ODD SEMESTER)
I SESSIONAL TEST QUESTION PAPER
SET-A

USN | | | | | | | | | | |

Degree	:	B.E	Semester	:	III
Branch	:	CSE	Course Code	:	BCS304
Course Title	:	Data Structures and Applications	Date	:	05/ 01 / 2024
Duration	:	90 Minutes	Max Marks	:	25

Note: Answer ONE full question from each part.

Q No.	Question	Marks	K-Level	CO mapping
	PART-A			
1(a)	**Define** Data structures. **Explain** its classifications. **List** the basic operations can be performed on data structure.	5	Understanding (K2)	CO1
(b)	**Convert** the given infix expression, ((a/(b-c+d))*(e-a)*c) to postfix expression. **Evaluate** the obtained postfix expression for the given data a=6, b=3, c=1, d=2, e=4.	5	Applying (K3)	CO1
(c)	**Define** the 2 ways to represent polynomial in C and **Show** the structural representation for the given two polynomials, $A(x)=4x^{15}+3x^4+5$ and $B(x)=2x^{1000}+10$. **Develop** a function to add two polynomials.	5	Applying (K3)	CO1
	OR			
2(a)	**Differentiate** between malloc() and calloc() functions.	5	Understanding (K2)	CO1
(b)	**Develop** an algorithm/function for Knuth-Morris-Pratt Pattern Matching Algorithm and **apply** on the following data T=abcaabaaabcaaabbc P1=aaabb	5	Applying (K3)	CO1
(c)	**Develop** a function for Fast transpose of Sparse Matrix. **Identify** the triplet form of Sparse matrix and **identify** the transpose of the given Matrix  	5	Applying (K3)	CO1

	PART-B			
3(a)	**Define** queue. **Explain** the operations performed on queue.	**5**	**Understanding (K2)**	**CO2**
(b)	**Define** priority queue. **Demonstrate** in detail One-Way list representation of a Priority Queue with an example.	**5**	**Applying (K3)**	**CO2**
	OR			
4(a)	**Explain** circular queue using dynamically allocated arrays	**5**	**Understanding (K2)**	**CO2**
(b)	**Discuss** the disadvantage of the ordinary queue. **Demonstrate** implementation of circular queue using arrays.	**5**	**Applying (K3)**	**CO2**

Course Incharge **HOD** **IQAC- Coordinator** **Principal**

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

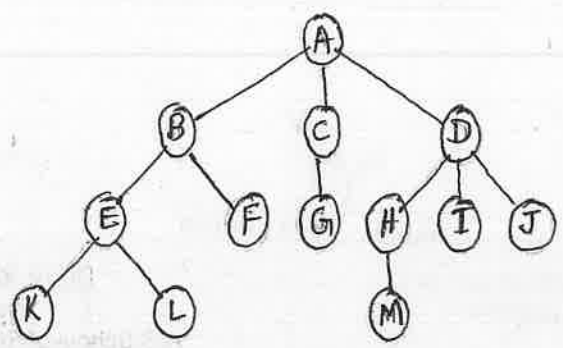
Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Management
Bengaluru - 560 109

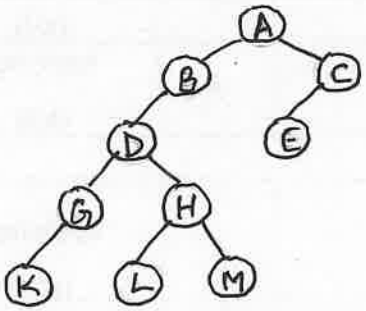
K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SESSION: 2023-2024 (ODD SEMESTER)
II SESSIONAL TEST QUESTION PAPER
SET-B

USN | | | | | | | | | | |

Degree	:	B.E	Semester	:	III A & B
Branch	:	CSE	Course Code	:	BCS304
Course Title	:	Data Structures and Applications	Date	:	10/ 02 / 2024
Duration	:	90 Minutes	Max Marks	:	25

Note: Answer ONE full question from each part.

Q No.	Question	Marks	K-Level	CO mapping
	PART-A			
1(a)	**Define** linked list. **Classify** the differences between array and Linked List.	5	Applying (K3)	CO2
(b)	**Develop** a C function for addition of two polynomials using linked List and Consider the given 2 polynomials, $a=5x^6+6x^4+2x^3$ and $b=8x^6+3x^2+4x+5$. Represent the polynomials using Linked list.	5	Applying (K3)	CO2
	OR			
2(a)	**Evaluate** A(1,2) using Ackerman's function and also **develop** a C recursive function for the same.	5	Applying (K3)	CO2
(b)	**Explain** linked stack and queue. **Develop** C function to perform operations on linked stack.	5	Applying (K3)	CO2
	PART-B			
3(a)	**Develop** the following functions for singly linked list(chains): i. Reverse/Invert the list ii. Concatenate two lists.	5	Applying (K2)	CO3
(b)	**Define** Tree. Represent the below given tree using i. Linked list representation with parenthetical notation ii. Left-child right-sibling representation 	5	Applying (K3)	CO3

(c)	**Explain** the advantage of Doubly linked list over Singly Linked list. **Develop** a C function for the following operations on DLL i. Insert at front ii. Delete at end iii. Display and count number of nodes			
	OR			
4(a)	**Explain** the concept of sparse matrix using Linked list. **Give** a node structure for linked representation and **apply** it for the following matrix 0 0 9 0 0 1 0 0 5 0 9 0 8 0 0 6	5	**Applying (K3)**	**CO3**
(b)	**Define** Binary tree with an example. **Develop** a C recursive routine to traverse the given tree using in-order, pre-order, post-order and level order 	5	**Applying (K3)**	**CO3**
(c)	**Explain** insertion in to a Threaded binary tree with neat diagram and also **develop** a C function to do the inorder traversal of a threaded binary tree.		**Applying (K3)**	**CO3**

Course Incharge

HOD

Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

IQAC- Coordinator

Principal

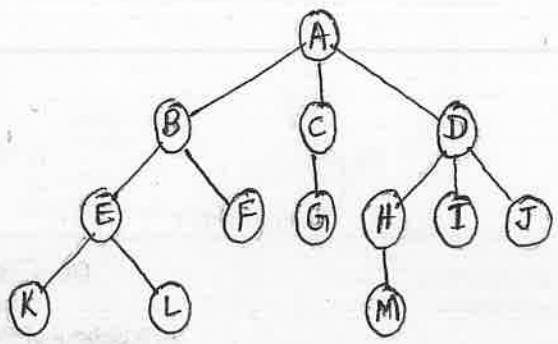
Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Manage
Bengaluru - 560 109

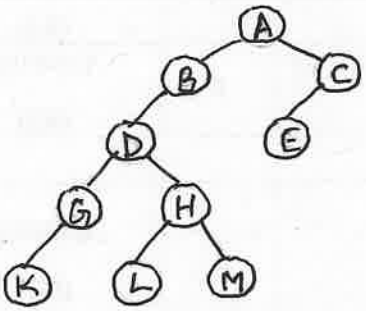
K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SESSION: 2023-2024 (ODD SEMESTER)
II SESSIONAL TEST QUESTION PAPER
SET-B

USN | | | | | | | | | | |

Degree	:	B.E	Semester	: III A & B
Branch	:	CSE	Course Code	: BCS304
Course Title	:	Data Structures and Applications	Date	: 10/ 02 / 2024
Duration	:	90 Minutes	Max Marks	: 25

Note: Answer ONE full question from each part.

Q No.	Question	Marks	K-Level	CO mapping
	PART-A			
1(a)	**Define** linked list. **Classify** the differences between array and Linked List.	5	Applying (K3)	CO2
(b)	**Develop** a C function for addition of two polynomials using linked List and Consider the given 2 polynomials, $a=5x^6+6x^4+2x^3$ and $b=8x^6+3x^2+4x+5$. Represent the polynomials using Linked list.	5	Applying (K3)	CO2
	OR			
2(a)	**Evaluate** A(1,2) using Ackerman's function and also **develop** a C recursive function for the same.	5	Applying (K3)	CO2
(b)	**Explain** linked stack and queue. **Develop** C function to perform operations on linked stack.	5	Applying (K3)	CO2
	PART-B			
3(a)	**Develop** the following functions for singly linked list(chains): i. Reverse/Invert the list ii. Concatenate two lists.	5	Applying (K2)	CO3
(b)	**Define** Tree. Represent the below given tree using i. Linked list representation with parenthetical notation ii. Left-child right-sibling representation 	5	Applying (K3)	CO3

(c)	**Explain** the advantage of Doubly linked list over Singly Linked list. **Develop** a C function for the following operations on DLL i. Insert at front ii. Delete at end iii. Display and count number of nodes			
	OR			
4(a)	**Explain** the concept of sparse matrix using Linked list. **Give** a node structure for linked representation and **apply** it for the following matrix 0 0 9 0 0 1 0 0 5 0 9 0 8 0 0 6	5	**Applying (K3)**	**CO3**
(b)	**Define** Binary tree with an example. **Develop** a C recursive routine to traverse the given tree using in-order, pre-order, post-order and level order 	5	**Applying (K3)**	**CO3**
(c)	**Explain** insertion in to a Threaded binary tree with neat diagram and also **develop** a C function to do the inorder traversal of a threaded binary tree.		**Applying (K3)**	**CO3**

Course Incharge

HOD

Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

IQAC- Coordinator

Principal

Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Manage
Bengaluru - 560 109

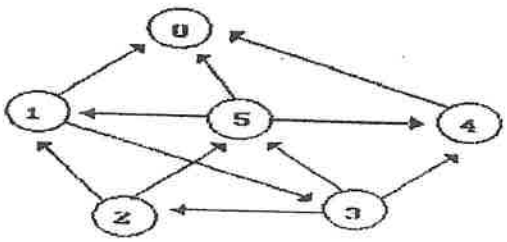
K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SESSION: 2023-2024 (ODD SEMESTER)
III SESSIONAL TEST QUESTION PAPER
SET-B

USN										

Degree : B.E
Branch : CSE
Course Title : Data Structures and Applications
Duration : 90 Minutes

Semester : III
Course Code : BCS304
Date : 6/2/2024
Max Marks : 25

Note: Answer ONE full question from each part.

Q No.	Question	Marks	K-Level	CO mapping
	PART-A			
1(a)	**Define** Binary Search Tree. **Develop** a C function to perform searching in a Binary search Tree.	5	Applying K3	CO4
(b)	**Explain** Selection Tree. Construct Winner Tree and Loser Tree for the following 16, 9, 20, 6, 50, 11, 90, 18	5	Applying K3	CO4
(c)	**Develop** a C function for Weighting rule for Union(i,j) **illustrate** with suitable example.	5	Applying K3	CO4
	OR			
2(a)	For the given graph **show** the adjacency matrix and adjacency list representation 	5	Applying K3	CO4
(b)	**Develop** a C routine to print the reachable nodes of a graph from the source node using Breadth First Search and also **illustrate** with a suitable graph.	5	Applying K3	CO4
(c)	**Illustrate** with suitable example a) Graph b) Connected components c) Spanning Tree d) Biconnected components	5	Applying K3	CO4
	PART-B			
3(a)	**Define** collision. **List** different overflow handling techniques? **Demonstrate** chaining with an example.	5	Applying K3	CO5
(b)	**Define** Hashing. **Illustrate** different Hash functions with a suitable example for each.	5	Applying K3	CO5

	OR			
4(a)	**Explain** motivation for dynamic hashing. **Illustrate** dynamic hashing using directories and directoryless dynamic hashing.	**5**	**Applying K3**	**CO5**
(b)	**Develop** a search function for linear probing. **Apply** division method and solve collision (if any) by using linear probing. Given key elements are: 72, 27, 36, 24, 63, 81, 92, 101 size of the hash table is 10.	**5**	**Applying K3**	**CO5**

(5)

Course In-charge

HOD

Department of Computer Science Engineering
...ool of Engineering & Management
Bangalore-560109

IQAC- Coordinator

Principal

Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Manager
Bengaluru - 560 109

CBCS SCHEME

USN | 1 | K | 9 | 2 | 2 | | | | | |

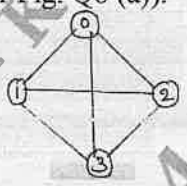
BCS304

Third Semester B.E./B.Tech. Degree Examination, Dec.2023/Jan.2024
Data Structures and Applications

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Define Data Structures. Explain with neat block schematic different type of data structures with examples. What are the primitive operations that can be performed?	10	L2	CO1
	b.	Differentiate between structures and unions shown examples for both.	5	L1	CO1
	c.	What do you mean by pattern matching? Outline knuth, Morris, Pratt pattern matching algorithm.	5	L2	CO1
		OR			
Q.2	a.	Define stack. Give the implementation of Push ( ), POP ( ) and display ( ) functions by considering its empty and full conditions.	7	L2	CO1
	b.	Write an algorithm to evaluate a postfix expression and apply the same for the given postfix expression 6, 2, /, 3, -, 4, 2, *, +	7	L3	CO1
	c.	Write the Postfix form of the following using stack : (i) A*(B*C+D*E) + F (ii) (a + (b*c) / (d-e))	6	L3	CO1
		Module – 2			
Q.3	a.	What are the disadvantages of ordinary queue? Discuss the implementation of circular queue.	8	L2	CO2
	b.	Write a note on multiple stacks and priority queue.	6	L2	CO2
	c.	Define Queue. Discuss how to represent queue using dynamic arrays.	6	L2	CO2
		OR			
Q.4	a.	What is a linked list? Explain the different types of linked lists with neat diagram.	4	L2	CO2
	b.	Give the structure definition for singly linked list (SLL). Write a C function to, (i) Insert on element at the end of SLL. (ii) Delete a node at the beginning of SLL.	8	L3	CO2
	c.	Write a C-function to add two polynomials show the linked list representation of below two polynomials $p(x) = 3x^{14} + 2x^{8} + 1$ $q(x) = 8x^{14} - 3x^{10} + 10x^{6}$	8	L3	CO2
		Module – 3			
Q.5	a.	Write a C-function for the following operations on Doubly Linked List (DLL): (i) addition of a node. (ii) concatenation of two DLL.	8	L3	CO3
	b.	Write C functions for the following operations on circular linked list : (i) Inserting at the front of a list. (ii) Finding the length of a circular list.	8	L3	CO3

	c.	For the given sparse matrix, give the diagrammatic linked representation. $A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$.	4	L3	CO3
		OR			
Q.6	a.	Discuss how binary tree are represented using, (i) Array (ii) Linked list	6	L2	CO3
	b.	Discuss inorder, preorder, postorder and level order traversal with suitable recursive function for each.	8	L2	CO3
	c.	Define Threaded Binary Tree. Discuss In-Threaded binary Tree.	6	L2	CO3
		Module – 4			
Q.7	a.	Write a function to perform the following operations on Binary Search Tree (BST) : (i) Inserting an element into BST. (ii) Recursive search of a BST.	8	L3	CO4
	b.	Discuss selection Trees with an example.	8	L2	CO4
	c.	Explain Transforming a first into a binary tree with an example.	4	L2	CO4
		OR			
Q.8	a.	Define graph. Show the adjacency matrix and adjacency list representation of the graph given below (Refer Fig. Q8 (a)).  Fig. Q8 (a)	6	L3	CO4
	b.	Define the following Terminologies with examples, (i) Digraph (ii) Weighted graph (iii) Self loop (iv) Parallel edges	8	L1	CO4
	c.	Explain in detail elementary graph operations.	6	L2	CO4
		Module – 5			
Q.9	a.	What is collision? What are the methods to resolve collision? Explain linear probing with an example.	7	L2	CO5
	b.	Explain in detail, about static and dynamic hashing.	6	L2	CO5
	c.	Discuss Leftist Trees with an example.	7	L2	CO5
		OR			
Q.10	a.	Explain different types of HASH function with example.	6	L2	CO5
	b.	Discuss AVL tree with an example. Write a function for insertion into an AVL Tree.	6	L3	CO5
	c.	Define Red-black Tree, Splay tree. Discuss the method to insert an element into Red-Black tree.	8	L2	CO5

* * * * *

CBCS SCHEME

USN | 1 | K | G | 2 | 1 | A | D | 0 | 0 | 9 | 21CS32

Third Semester B.E. Degree Examination, Dec.2023/Jan.2024
Data Structures and Applications

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. What is data structure? Explain in detail classification of data structures with example. **(10 Marks)**

b. Write an algorithm for inserting and deleting an element at a given location in an array and implement the same in 'C' language. **(10 Marks)**

OR

2 a. Explain the nested structures with an example of a 'C' program. **(07 Marks)**

b. What are self-referential structures? **(03 Marks)**

c. Explain 'C' library functions for memory allocation/deallocation functions with example. **(10 Marks)**

Module-2

3 a. What is stack? Explain basic operations of stack with algorithm. **(05 Marks)**

b. Write 'C' program to implement stack using array. **(05 Marks)**

c. Write an algorithm to convert an infix notation to post fix notation to post fix notation and apply the algorithm for the following infix expression to convert it into post fix.
$A-(B/C+(D\%E*F)/G)*H$. **(10 Marks)**

OR

4 a. What is queue? Explain basic operations of queue with algorithm. **(06 Marks)**

b. Write 'C' program to implement linear queue using array. **(07 Marks)**

c. Explain different types of queues with example. **(07 Marks)**

Module-3

5 a. What are linked lists? Explain with algorithm inserting a new node in a linked list for the following cases:
Case 1 : The new node is inserted at the beginning.
Case 2 : The new node after a given node. **(10 Marks)**

b. What are circular linked lists? Explain with algorithm deleting a node from a circular linked list for the following cases:
Case 1 : The first node
Case 2 : The last node. **(10 Marks)**

OR

6 a. Represent polynomial using linked list and explain addition of two polynomial with algorithm. **(10 Marks)**

b. Write a 'C' program to implement stack using linked list. **(10 Marks)**

Module-4

7 a. What are binary trees? Explain the linked representation of binary tree. **(08 Marks)**

b. What is binary search tree? Construct the binary tree for the following expression: $exp = ((a+b)-(c*d))\%((e \wedge f)/(g-h))$. **(07 Marks)**

c. Write applications of trees. **(05 Marks)**

OR

8 a. Explain pre-order and in-order traversal with example and also write algorithm. **(10 Marks)**

b. Explain inserting and deleting a new node in a binary search tree with algorithm. **(10 Marks)**

Module-5

9 a. What are AVL trees? Explain operations on AVL trees with example. **(10 Marks)**

b. What are red-black trees? Explain operations on red-black trees with example. **(10 Marks)**

OR

10 a. Explain the graph representation using adjacency matrix. **(05 Marks)**

b. Explain the two standard graph traversal algorithms in detail with example. **(10 Marks)**

c. Explain different hash functions with example. **(05 Marks)**

* * * * *

CBCS SCHEME

USN | | | | | | | | | | 21CS32

Third Semester B.E. Degree Examination, Jan./Feb. 2023
Data Structures and Applications

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. What is linear array? Discuss the representation of linear array in memory. (06 Marks)

b. Differentiate between static and dynamic memory allocations. Discuss four dynamic memory allocation functions. (06 Marks)

c. Write a menu driven program in C for the following array operations:
(i) Inserting an element (ELEM) at a given valid position.
(ii) Deleting an element at a given valid position.
(iii) Display of array elements.
(iv) Exit
Support the program with functions for each of the above operations. (08 Marks)

OR

2 a. Give Abstract Data Type (ADT) for arrays. How array can be declared and initialized? (06 Marks)

b. With suitable example, discuss self-referential structures. (06 Marks)

c. Define Sparse matrix. How to represent a Sparse matrix? Write an algorithm/function to transpose a given Sparse matrix. (08 Marks)

Module-2

3 a. Define Stack. Discuss how to represent stack using dynamic arrays. (06 Marks)

b. Write a menu driven C program for the following operations on STACK of integers:
(i) Push an element on to stack
(ii) Pop an element from the stack
(iii) Display the contest of stack
(iv) Exit
Show the overflow and underflow conditions. (06 Marks)

c. What are the disadvantages of ordinary queue? Discuss the implementation of circular queue using arrays. (08 Marks)

OR

4 a. What is Recursion? Write recursive function to solve Towers of Hanoi problem. (06 Marks)

b. Discuss the following:
(i) Double Ended Queue
(ii) Priority Queue (06 Marks)

c. Write an algorithm to convert infix expression to postfix expression. Show the content of stack to convert the following infix expression:
A * (B + D)/E – F * (G + H/K) (08 Marks)

Module-3

5 a. Write a C function to concatenate two singly linked list. **(06 Marks)**

b. Give the structure definition for singly linked list. Write a C function to:
(i) Insert an element at the end
(ii) Delete a node at the beginning **(08 Marks)**

c. Discuss how to read a polynomial consisting of 'n' terms implemented using linked list. **(06 Marks)**

OR

6 a. Write a function to delete a node whose information field is specified in singly linked list. **(06 Marks)**

b. What is circular doubly linked list? Write a C function to perform the following operations on circular doubly linked list:
(i) Insert a node at the beginning
(ii) Delete a node from the leas **(08 Marks)**

c. Discuss how to implement stacks and queues using linked list. **(06 Marks)**

Module-4

7 a. Define binary tree. List and discuss any two properties of binary tree. **(06 Marks)**

b. Write a function to perform the following operations on Binary Search Tree (BST):
(i) Deletion from a BST
(ii) Inserting an element into a BST **(08 Marks)**

c. Define Threaded Binary Tree. Discuss In-threaded binary tree. **(06 Marks)**

OR

8 a. Discuss how binary tree are represented using (i) Array (ii) Linked list **(06 Marks)**

b. Discuss inorder, preorder, postorder and level order traversal with suitable recursive function for each. **(08 Marks)**

c. Write a C function to evaluate an expression using expression tree. **(06 Marks)**

Module-5

9 a. Design a C program for the following operation on Graph (G) of cities:
(i) Create a graph of N cities using adjacency matrix
(ii) Print all the nodes reachable from a given starting node in a digraph using BFS/DFS method **(10 Marks)**

b. Discuss AVL tree with an example. Write a function for insertion into an AVL tree. **(10 Marks)**

OR

10 a. Define hashing. What are the two criteria, a good hash function should satisfy? Discuss open addressing and chaining method with an example. **(10 Marks)**

b. Define Red-Black tree, Splay tree and B tree. Discuss the method to insert an element into Red-Black tree. **(10 Marks)**

* * * * *

CBCS SCHEME

USN | | | | | | | | | | 21CS32

Third Semester B.E. Degree Examination, June/July 2023
Data Structures and Applications

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. Define data structures. Classify data structures in various features. **(06 Marks)**

b. Write algorithms to insert a data element into array and delete an element from the array. **(07 Marks)**

c. Explain various memory allocation and de-allocation function supported in C. **(07 Marks)**

OR

2 a. Explain user defined structures with respect to C. Give structure definition and declaration for STUDENT data with the following information: USN and Name. Also give self referential structure. **(04 Marks)**

b. Show array representation of two polynomials. Write a C function to add two polynomials A(x) and B(x) term by term to produce D(x) where D(x) = A(x) + B(x). $A(x) = 2x^{10} + x + 3$, $B(x) = x^5 + 10x^3 + 3x^2 + 12$. **(08 Marks)**

c. Obtain triplet representation for the given sparse matrix. Write fast transpose algorithm to obtain transpose of sparse matrix.

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & 15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

(08 Marks)

Module-2

3 a. How recursion uses stack during its execution. Give algorithm to simulate Tower of Hanoi. Trace the algorithm for a total of 3 disc which are placed in source pole. **(06 Marks)**

b. Write C routines to implement operations on stack. Also incorporate useful routines to check the stack status for full and empty. Also include global declarations. **(07 Marks)**

c. Write algorithm to convert infix expression to prefix form. Apply the algorithm to obtain equivalent prefix form. Infix expression : $6 * 2 \wedge 2 \wedge 3/(9 - 3)$ **(07 Marks)**

OR

4 a. Design circular queue using dynamically allocated arrays. Give steps to relocate elements in dynamic array for proper insertion and deletion. **(04 Marks)**

b. With the help of algorithm, evaluate the postfix expression '6223^^*3' using stack. **(08 Marks)**

c. What is the advantage circular queue over ordinary queue? Give ADT to perform various operations on circular queue. Also give ADTs to check for empty and full. **(08 Marks)**

Module-3

5 a. Give structure representation in C to create a single linked list. Give C routine to implement following operations on SLL:
(i) Create SLL of integer data
(ii) Insert a node at rear end
(iii) Delete a node from front end
(iv) Display all nodes neatly
(v) Search for a suitable data in SLL and display appropriate message. **(12 Marks)**

b. What is the advantage of doubly linked list? Give suitable steps to insert a node between A and B (consider A is NULL, B is NULL and A & B are not NULL) in SLL. **(08 Marks)**

OR

6 a. Write the node representation of the linked representation of a polynomial. Also give algorithm to perform addition on two polynomials. **(10 Marks)**

b. Differentiate between SLL, DLL, circular linked list and header linked list. Give algorithm to insert a node circular linked list and traverse the list. **(10 Marks)**

Module-4

7 a. Define tree. For the given tree, explain terminologies and write the answer:
(i) Degree (ii) Non terminal (iii) Sibling
(iv) Ancestor (v) Level (vi) Height

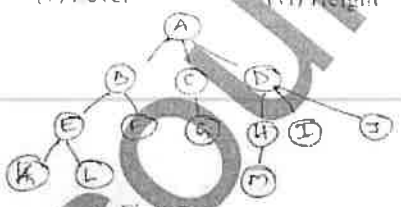


Fig.Q7(a) **(06 Marks)**

b. Give C routine to create BST for the data 12, 0, –90, 5, 3, 10, 0, 8, 18. Give 3 traversals of BST constructed from above data. **(07 Marks)**

c. Given in order sequence DHBEAFICG and post order sequence HDEBFIGCA, construct binary tree and give pre-order traversal. **(07 Marks)**

OR

8 a. Give array and linked list representation for the binary tree.

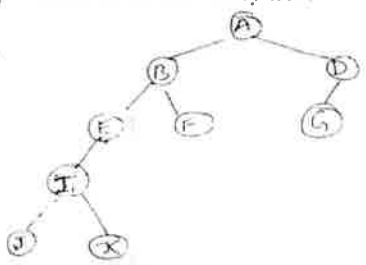


Fig.Q8 **(06 Marks)**

b. Write iterative and recursive search function to search a key in BST. **(08 Marks)**

c. Draw a binary tree for the following expression 3 + 4 * (7 – 6) / 4 – 3. Traverse the tree and obtain pre-order and post order expression. **(06 Marks)**

Module-5

9 a. For the given graph show adjacency matrix and adjacency list representation

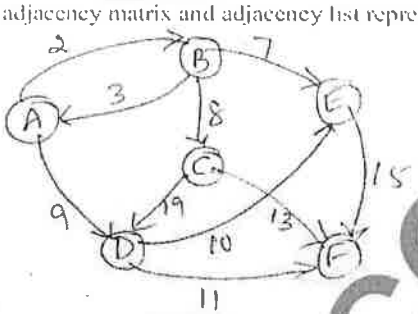


Fig.Q9(a)

(06 Marks)

b. Write BSF and DFS algorithm for graph traversal. (10 Marks)

c. Write a note on AVL tree. (04 Marks)

OR

10 a. What is hashing? Explain different hashing function with suitable numerical example (08 Marks)

b. What is collision? Explain the method to resolve collision with suitable algorithm of liner probing. Insert keys 72, 27, 36, 24, 63, 81, 92, 101 into table [size = 10]. (08 Marks)

c. Write a note on B-tree. (04 Marks)

* * * * *

CBCS SCHEME

USN | | | | | | | | | | |

18CS32

Third Semester B.E. Degree Examination, July/August 2022
Data Structures and Applications

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. Define data structures. Explain the classification of data structures with examples. **(06 Marks)**

b. Explain the dynamic memory allocation functions supported by 'C' with syntax and examples. **(06 Marks)**

c. Consider the pattern P = ababab. Construct the table and the corresponding labeled directed graph used in the fast or second pattern matching algorithm. Trace it for the input text T = abaababababba. **(08 Marks)**

OR

2 a. Differentiate between structures and unions. Show examples for both. **(06 Marks)**

b. Explain any four string handling functions supported by 'C' with syntax and examples. **(06 Marks)**

c. Explain the representation of linear arrays in memory. Also, consider the linear arrays AAA (5:50) and BBB(-5:10).

i) Find the number of elements in each array.

ii) Suppose Base (AAA) = 300, Base (BBB) = 500 and 4 words per memory cell for AAA, 2 words per memory cell for BBB, find the address of AAA[15], AAA[55], BBB[8] and BBB[0]. **(08 Marks)**

Module-2

3 a. Define a stack. Explain the different operations that can be performed on stacks with suitable 'C' functions and examples. **(07 Marks)**

b. Convert the following infix expression into postfix expression using stack.
A + (B * C – (D / E ^ F) * G) * H. **(05 Marks)**

c. Develop a C recursive program for tower of Hanoi problem. Trace it for 3 disks with schematic call tree diagram. **(08 Marks)**

OR

4 a. Develop C functions to implement insertion, deletion and display operations of a circular queue. **(07 Marks)**

b. Write an algorithm to evaluate a postfix expression. Trace the algorithm for the following expression showing the stack contents 6 5 1 – 4 * 2 3 ^ / +. **(06 Marks)**

c. Define Ackermann function recursively and evaluate A(3, 0). Also, develop C code for the same. **(07 Marks)**

Module-3

5 a. Write the differences between arrays and linked lists. **(04 Marks)**

b. Develop C functions to implement the following in a singly linked list:
i) Delete a node from the front ii) Concatenate two linked lists. **(08 Marks)**

c. Develop a C function to add two polynomials using singly linked list. **(08 Marks)**

OR

6 a. Show the diagrammatic linked representation for the following sparse matrix:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

(04 Marks)

b. Develop C functions to implement the following in a doubly linked list:
   i) Insert a node at the front
   ii) Delete a node from the end. (08 Marks)

c. Develop C functions to implement the various operations of queues using linked list. (08 Marks)

Module-4

7 a. With suitable examples, define the following:
   i) Degree of a node
   ii) Level of a binary tree
   iii) Complete binary tree
   iv) Full binary tree. (06 Marks)

b. Construct binary search tree for the given set of values 14, 15, 4, 9, 7, 18, 3, 5, 16, 20. Also, perform inorder, preorder and postorder traversals of the obtained tree. (06 Marks)

c. Explain threaded binary trees and their representation with a neat diagram. Also, develop a C function to do the inorder traversal of a threaded binary tree. (08 Marks)

OR

8 a. Explain the array and inked representation of binary trees with suitable examples. (06 Marks)

b. A binary tree has 9 nodes. The inorder and preorder traversals yield the following sequences of nodes:
   Inorder: E A C K F H D B G
   Preorder: F A E K C D H G B
   Draw the binary tree. Also, perform the post order traversal of the obtained tree. (06 Marks)

c. Develop C functions to implement the following:
   i) Search a key value in a binary search tree
   ii) Copying a binary tree (08 Marks)

Module-5

9 a. Define a graph. For the graph shown in Fig.Q.9(a), show the adjacency matrix and adjacency list representations. (06 Marks)

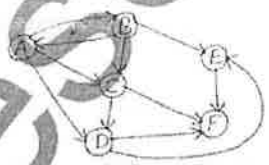


Fig.Q.9(a)

b. Suppose an array contains 8 elements as follows: 77, 33, 44, 11, 88, 22, 66, 55. Sort the array using insertion sort algorithm. (06 Marks)

c. What is hashing? Explain the following hash functions with proper examples:
   i) Division ii) Midsquare iii) Folding. (08 Marks)

OR

10 a. Briefly explain Breadth-First Search (BFS) and Depth-First Search (DFS) traversal of a graph. Also, show the BFS and DFS traversals for the following graph in Fig.Q.10(a).

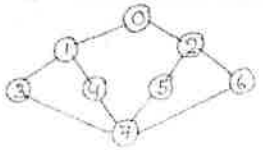


Fig.Q.10(a)

(06 Marks)

b. Suppose 9 cards are punched as follows: 348, 143, 361, 423, 538, 128, 321, 543, 366. Apply radix sort to sort them in 3 phases. (06 Marks)

c. What is Collision? Explain the collision resolution techniques with proper examples. (08 Marks)

* * * * *

MODULE-1
INTRODUCTION

TOPICS:

Module 1: Introduction: Data Structures, Classifications (Primitive & Non Primitive), Data structure Operations, Review of Arrays, Structures, Self-Referential Structures, and Unions. Pointers and Dynamic Memory Allocation Functions. Representation of Linear Arrays in Memory, Dynamically allocated arrays.
Array Operations: Traversing, inserting, deleting, searching, and sorting. Multidimensional Arrays, Polynomials and Sparse Matrices.
Strings: Basic Terminology, Storing, Operations and Pattern Matching algorithms. Programming Examples.

1. INTRODUCTION

Basic Terminology of Data Organization:

- **Data:** The term 'DATA' simply refers to a value or a set of values. These values may present anything about something, like it may be roll no of a student, marks, name of an employee, address of person etc.
- **Data item:** A data item refers to a single unit of value.
    - ✓ **For eg**. roll no of a student, marks, name of an employee, address of person etc. are data items.
    - ✓ Data items that can be divided into sub items are called **group items** (Eg. Address, date, name),
    - ✓ Data items which cannot be divided in to sub items are called **elementary items** (Eg. Roll no, marks, city, pin code etc.).
- **Entity -** with similar attributes (e.g all employees of an organization) form an entity set.
- **Information:** Data with given attribute or processed data.
- **Field** is a single elementary unit of information representing an attribute of an entity.
- **Record** is the collection of field values of a given entity.
- **File** is the collection of records of the entities in a given entity set.
- Each record in a file may contain many field items but the value in a certain field may uniquely determine the record in the file. Such a field K is called a **primary key**, and the values K_1, K2, K3… in such a field are called **keys or key values.**
- Records can be classified as **fixed-length records or variable-length records.** In **fixed length records**, all the records contain the same data items with the same amount of space assigned to each data item .In **variable length records**, file records may contain different lengths.

EXAMPLE:

Attributes:	Name	Age	Sex	Roll Number	Branch
Values:	A	17	M	109cs0132	CSE
	B	18	M	109ee1234	EEE

Here, it is an example of student details where STUDENT is the given entity. Then name, age, sex, roll number, branch are attributes and their values are properties (A, 17, M, 109cs0132, CSE). Collection of student details is student entity set and Roll number is the primary key which uniquely indicates each student details.

DATA STRUCTURE

- **"Data Structure** is a way of collecting and organizing data in such a way that the operations on these data can be performed in an effective way".
- **"Data structure** can be defined as logical or mathematical model of a particular organization of data."
- **"Data structure** is a representation of logical relationship existing between individual elements of data". In other words, a**data structure** defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.
- **Data Structures** is about rendering data elements in terms of some relationship, for better organization and storage in computer.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

For example, data can be player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type. This data can be recorded as a Player record. Now, it is possible to collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33.

In simple language, "**Data Structures** are structures programmed to store ordered data, so that various operations can be performed on it easily. It's an arrangement of data in a computer's memory. Algorithms manipulate the data in these structures in order to accomplish some task**".**

1.1 CLASSIFICATION OF DATA STRUCTURES

The logical or mathematical model of a particular organization of data is called a **Data Structure**.

Data structures can be classified as

- Primitive data structure
- Non-Primitive data structure

Anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures** and some complex Data Structures, which are used to store large and connected data. They are called **Non-primitive DataStructure**s.

Examples:

- Array
- Stack
- Queue
- Linked List
- Tree
- Graph

All these data structures allow us to perform different operations on data. The selection of these data structures is based on which type of operation is required.

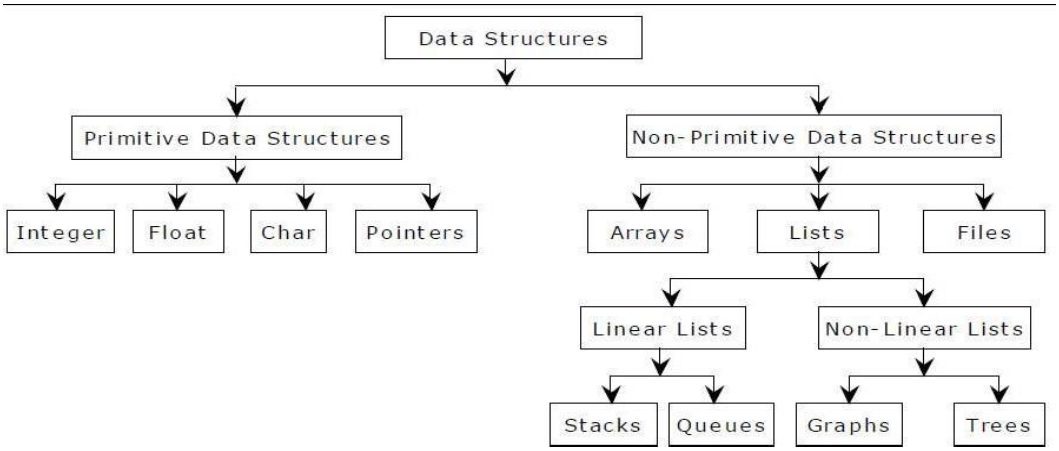


Figure 1: Classification of data structure

Figure 1 gives the complete classification of the data structure.

Definitions:

Primitive Data Structure: "A primitive data structure used to represent the standard data types of any one of the c omputer languages". Variables, pointers, etc. are examples of primitivedata structures. Simple data structure can be constructed with the help of primitive data structure.

Non-Primitive Data structure: "Non-Primitive data structure can be constructed with the help of any one of the primitive data structure and it is having a specific functionality. It can be designed by user". Arrays, Structure, Union, linked list, Stacks, Queue, trees, graphs etc are example for non primitive data structures.

Non primitive data structures can be further classified as

1) Linear data structure
2) Non-linear data structure

Linear Data Structures:

"Linear data structures can be constructed as a continuous arrangement of data elements in the memory. In linear data structure the elements are stored in sequential order. In the linear Data Structures the relationship of adjacency is maintained between the Data elements .It can be represented by using array data type or linked list". Each element has one successor and one predecessor.

The linear data structures are:

- **Array:** Array is a collection of data of same data type stored in consecutive memory location and is referred by common name
- **Stack:** A stack is a Last-In-First-Out (LIFO) linear data structure in which insertion and deletion takes place at only one end called the top of the stack.
- **Queue:** A Queue is a First in First-Out (FIFO) linear data structure in which insertions takes place one end called the rear and the deletions takes place at one end called the Front.
- **Linked list:** Linked list is a collection of data of same data type but the data items need not be stored in consecutive memory locations.

Non-Linear Data Structures:

"**Non-linear data structure** can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the Data items. Elements are stored based on the hierarchical relationship among the data." Each node doesn't have exactly one predecessor and one successor. It may contain more than 1 predecessor or successor.

The following are some of the Non-Linear data structure

- **Trees:** Trees are used to represent data that has some hierarchical relationship among the data elements.(as shown in figure 2)

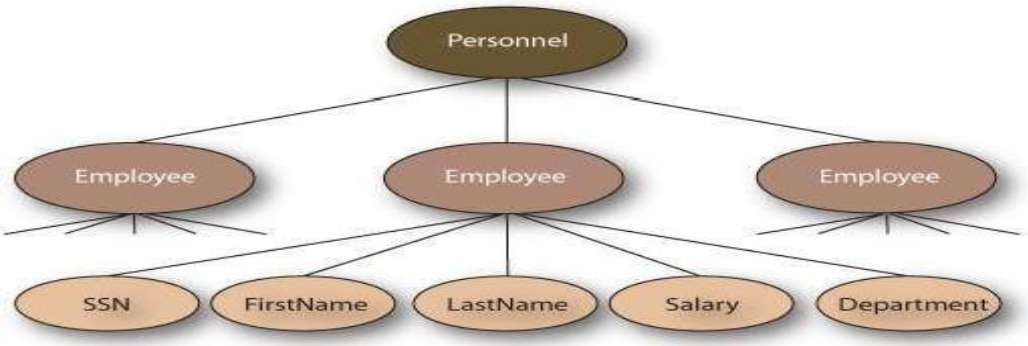


Figure 2: Trees

- **Graph:** Graph is used to represent data that has relationship between pair of elements not necessarily hierarchical in nature. For example electrical and communication networks, airline routes, flow chart, graphs for planning projects.(as shown in figure 3)

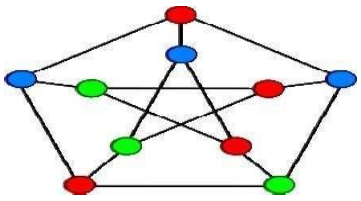


Figure 3: Graph

1.2 DATA STRUCTURE OPERATIONS

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion

- Deletion
- Sorting
- Merging

(1) Traversing: Accessing each record exactly once so that certain items in the record may be processed.

(2) Searching: Finding the location of a particular record with a given key value, or finding the location of all records which satisfy one or more conditions.

(3) Inserting: Adding a new record to the structure.

(4) Deleting: Removing the record from the structure.

(5) Sorting: Managing the data or record in some logical order (Ascending or descending order).

(6) Merging: Combining the record in two different sorted files into a single sorted file.

Operations on linear data structures

1. Add an element
2. Delete an element
3. Traverse all the elements
4. Sort the list of elements
5. Search for a data element

Apply one or more functionality to create different types of data structures.

For example *Stack, Queue, and Linked Lists.*

Operations applied on non-linear data structures

The following list of operations applied on non-linear data structures.

1. Add elements.
2. Delete elements
3. Display the elements
4. Sort the list of elements
5. Search for A data element by applying one or more functionalities and different ways of joining randomly distributed data items to create different types of data structures.

For example Tree, Graphs and Files.

1.3 REVIEW OF STRUCTURES, UNIONS AND POINTERS

STRUCTURE DEFINITION

"Structure is a collection of data items of same or dissimilar data type. Each data item is identified by its name and type. (Or) A **Structure** is a user defined data type that can store related information together. (Or) A **structure** is a collection of different data items / heterogeneous data items

under a single name.**"** The variable within a structure are of different data types and each has a name that is used to select it from the structure.

C arrays allow you to define type of variables that can hold several data items of the same kind but **structure** is another user defined data type available in C programming, which allows you to combine data items of different kinds.

Structures are used to represent a record, Suppose, track of books are kept in a library are recorded then it is required to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

STRUCTURE DECLARATION

It is declared using a keyword struct followed by the name of the structure. The members of the structure are declared within the structure.

Example:

```
struct struct-name
{
        data_type1 member_name1;
        data_type2 member_name2;
            ..          ..
        data_typen member_namen;
}structurevariablename;
```

STRUCTURE INITIALIZATION

Assigning values to the data members of the structure is called **initializing of structure.**

Syntax:

```
struct struct_name
{
         data _type member_name1;
         data _type member_name2;
 } structure variable={constant1,constant2};
```

Accessing the Members of a structure :-

A structure member variable is generally accessed using a **'.' dot operator**.

Syntax: **structurevariable.member_name;**

The dot operator is used to select a particular member of the structure. To assign value to the individual Data members of the structure variable stud, it is written as,

stud.roll=01;

stud.name="Rahul";

To input values for data members of the structure variable stud, can be written as,

scanf("%d",&stud.roll);

scanf(''%s",stud.name);

To print the values of structure variable stud, can be written as:

printf("%d",stud.roll);

printf("%s",stud.name);

Example for structure:

```
struct employee
{
        char name[10];
        int age;
        float salary;
}person;
```

Here **struct** is a keyword.

This example creates a variable whose name is person and that has three fields

- A name that is a character array
- An integer value Age
- A float value salary

The . (Dot operator) is used as the structure member operator to select a particular member of the structure.

Example:
```
strcpy(person.name,"james");
    person.age=20;
person.salary=40000;
```

Program to create a struct person , initializes its data member and print their values

```
#include<stdio.h>
#include<conio.h>
void main()
{
        struct person{
        char name[10];
        int age;
        float salary;
        };
        struct person p1;
        clrscr();
        strcpy(p1.name,"james");
        p1.age=10;
        p1.salary=35000;
        printf("\n name=%s age=%d salary=%f",p1.name,p1.age,p1.salary);
        getch();
}
```

ARRAY OF STRUCTUREs

An array of **structure can also be declared**. Each element of the array representing a **structure** variable.

Example : struct employee emp[5];

The above code define an array **emp** of size 5 elements. Each element of array **emp** is of type **employee**

```
#include<stdio.h>
#include<conio.h>
struct employee
{
        char ename[10];
        int sal;
};
struct employee emp[5];
int i,j;
void ask()
{
        for(i=0;i<3;i++)
        {
        printf("\nEnter %dst employee record\n",i+1);
        printf("\nEmployee name\t");
        scanf("%s",emp[i].ename);
        printf("\nEnter employee salary\t");
        scanf("%d",&emp[i].sal);
        }
        printf("\nDisplaying Employee record\n");
        for(i=0;i<3;i++)
        {
        printf("\nEmployee name is %s",emp[i].ename);
```

```
        printf("\nSlary is %d",emp[i].sal);
        }
}
void main()
{
        clrscr();
        ask();
        getch();
}
```

TYPE DEFINITIONS AND STRUCTURES

The structure definition associated with keyword typedef is called **Type-Defined Structure.**

Syntax 1: typedef struct

```
        {
                data_type member 1;
                data_type member 2;
                ……. …….
                data_type member n;
        }Type_name;
```

Where,

- **typedef** is the keyword used at the beginning of the definition and by using typedef user defined data type can be obtained.
- **struct** is the keyword which tells structure is defined to the complier • The members are declare with their data_type
- **Type_name** is not a variable, it is user defined data_type.

Syntax 2: struct struct_name

```
        {
                data_type member 1;
                data_type member 2;
                ……. …….
                data_type member n;
        };
```

typedef struct struct_name Type_name;

Example: It is possible to create our own data types (user defined)by using typedef statement as below

```
typedef struct
{
    char name[10];
        int age;
} humanbeing;
```

Here **humanbeing** is the name of the type defined by structure definition and we may follow this definition with declarations of variables such as

humanbeing person1, person2;

This statement declares the variable **person1** and **person2** are of type **humanbeing.**

- Structures cannot be directly checked for equality or nor equality. i.e. directly using **person1==person2 is not allowed**. If each individual data member is checked for equality then the entire structure can be checked for equality.

Function to Check equity of two structures

```
#define FALSE 0
#define TRUE 1
int humansEqual(humanBeing person1, humanBeing person2))
{
        if(strcmp(person1.name, person2.name))
                return FALSE;
        if((person1.age != person2.age)
                return FALSE;
        if((person1.salary != person2. salary)
                return FALSE;
        return TRUE;
}
void main()
{
        if(humansEqual(person1, person2))
                printf("the two human beings are the same\n");
        else
                printf("the two human beings are not the same");
}
```

Program to check equality to structure variables.

```
#include<stdio.h>
#include<conio.h>
#define FALSE 0
#define TRUE 1
typedef struct
{
        char name[10];
        int age;
        float salary;
}humanbeing;
int humansEqual(humanbeing p1,humanbeing p2)
{
        if(strcmp(p1.name,p2.name))
                return FALSE;
        if(p1.age!=p2.age)
                return FALSE;
        if(p1.salary!=p2.salary)
                return FALSE;
        else
                return TRUE;
}
void main()
{
        humanbeing p1,p2;
        clrscr();
        strcpy(p1.name,"hiiiii");
        p1.age=12
        ;p1.salary=12000;
        strcpy(p2.name,"hi");
        p2.age=12;
        p2.salary=12000;
        if(humansEqual(p1,p2))
                printf("\n persons are same ");
        else
                printf("\n persons are not same");
        getch();
}
```

POINTERS TO STRUCTURES

Pointer to a structure is a variable that holds the address of a structure. The syntax to declare pointer to a structure can be given as:

*strcut struct_name *ptr;*

eg: struct stud *ptr_stud;

To assign address of stud to the pointer using address operator(&) we would write ***ptr_stud=&stud;*** To access the members of the structure (->) operator is used.

for example ***ptr_stud->name=Raj;***

Example program (pass the address of structure as an argument)

```
#include<stdio.h>
Struct point
{
    int x;
    int y;
};
void print(struct point *ptr)
{
    Printf("%d%d\n", ptr->x, ptr->y);
}
int main()
{
    Struct point p1 = {23, 45};
    Struct point p1 = {23, 45};
    print(&p1);
    print(&p2);
    return 0;
}
```

Output: 23 45
56 90

NESTED STRUCTURES

It is possible to embed a structure within a structure. **"The structure that contains another structure variable as its members is called a nested structure or a structure within a structure is called nested structure."**

- The structure should be declared separately and then be grouped into high level structure. The data members of the nested structures can be accessed using **(.) Dot operator**.
- Syntax to follow while accessing the data members of inner structure in nested structure with dot operator is

  outer most structure variable. inner most structure variable. inner data member;

- Syntax to followed while accessing the data members of outer structure in nested structure with dot operator is

  outer most structure variable .outer data member ;

Type1: declaring structure within a structure

Example :

```
struct student
{
     char name[20];
     int marks;
     float per;
     struct dob
    {
        int day,month,year;
     }date;
}s1;
void main()
{
       printf("\n enter the student details-name marks percentage and date of birth");
       scanf ("%s%d%f%d%d%d",s1.name,&s1.marks,&s1.per,&s1.date.day,&s1.date.month,
                                                                    &s1.date.year);
       printf("the given student details is as follows");
     printf("\n name =%s marks=%d percentage=%d dob=%d/%d/%d", s1.name, s1.marks,s1.per,
              s1.date.day,s1.date.month,s1. date.year);
     getch();
}
```

Type 2: declaring a structure variable of one within another structure

Example :

```
struct dob
{
        int day,month,year;

 };
struct student
{
     char name[20];
     int marks;
     float per;
    struct dob date;
}s1;

void main()

{
```

```
        printf("\n enter the student details-name marks percentage and date of birth");
        scanf ("%s%d%f%d%d%d",s1.name,&s1.marks,&s1.per,&s1.date.day,&s1. date.month,
                &s1.date.year);
        printf("the given student details is as follows");
      printf("\n name =%s marks=%d percentage=%d dob=%d/%d/%d", s1.name, s1.marks,s1.per,
            s1.date.day,s1.date.month,s1.date.year);
      getch();
}
```

Nested structures with typedef

Example :

```
typedef struct
{
        int month;
        int day;
        int year;
}date;

typedef struct
{
        char name[10];
        int age;
        salary;
        date dob;
}humanbeing;
humanbeing person1,person2;
```

If humanbeing person1 and person2 declares person1 and person2 variables of type humanbeing.

Then consider a person born on feb 11, 1944, can have the values for the date struct set as:

person1.dob.month=2;

person1.dob.day=11;

person1.dob.year=1944;

Similarly for considering person2, his dob is 3rd December 1956 then

person2.dob.month=12;

person2.dob.day=3;

person2.dob.year=1956;

Program for illustration of nested structures with typedef

```
#include<stdio.h>
#include<conio.h>
void main()
{
        typedef struct
        {
                int month;
                int day;
                int year;
        }date;
        typedef struct
        {
                char name[10];
         int age;
         salary;
        date dob;
    }humanbeing;
     humanbeing person1;
    strcpy(person1.name,"james");
    person1.age=10;
    person1.salary=35000;
    Person1.dob.month=2;
    Person1.dob.day=11;
    Person1.dob.year=1944;
    printf("\n details of the person");
    printf("\n name=%s age=%d salary=%f dob=%d-%d-%d",person1.name,
    person1.age,person1.salary,Person1.dob.day,Person1.dob.month,Person1.dob.year);
    getch();
}
```

UNIONS

Union is a collection of variables of different data types. Union information can only be stored in one field at any one time.

Definition: "**A union is a special data type available in C that enables you to store different data types in the same memory location.** "

union can define many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

Declaring Union:

```
union union-name
{
        data_type1 member_name1;
```

```
        data_type2 member_name2;
            ..              ..
        data_typen member_namen;
}union variablename;
```

Example:

```
union data
{
        char a;
        int x;
        float f;
}mydata;
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition.

At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 4 bytes of memory space because this is the maximum space which can be occupied by float data.

Accessing a Member of a Union

```
#include <stdio.h>
#include <string.h>
union Data
{
        int i;
        float f;
        char str[20];
};
int main( )
{
        union Data data;
        data.i = 10;
        data.f = 220.5;
        strcpy( data.str, "C Programming");
        printf( "data.i : %d\n", data.i);
        printf( "data.f : %f\n", data.f);
        printf( "data.str : %s\n", data.str); return 0;
}
```

Dot operator can be used to access a member of the union. The member access operator is coded as a period between the union variable name and the union member that we wish to access.

Program to illustrate union with in a structure.

```
#include<stdio.h>
#include<conio.h>
typedef struct
{
        enum tagfield {female,male} sex;
        union
       {
                int child;
                int beard;
        }u;
}sextype;
typedef struct
{
        int m,d,y;
}date;
typedef struct
{
        char name[10];
        int age;
        float salary;
        date dob;
        sextype sexinfo;
}humanbeing;
void main()
{
        humanbeing p1;
        p1.dob.m=2;
        p1.dob.d=11;
        p1.dob.y=1994;
        p1.sexinfo.sex=female;
        p1.sexinfo.u.child=4;
        printf("year=%d/%d/%d",p1.dob.m,p1.dob.d,p1.dob.y);
        printf("\n sex=%d b=%d",p1.sexinfo.sex,p1.sexinfo.u. child);
        getch();
}
```

1.4 SELF REFERENTIAL STRUCTURES

"Self –referential structures are those structures that contain a reference to data of its same type as that of structure. i.e one or more of the components of the structure will be a pointer to itself."

Example 1

```
struct node
{
        int val;
        struct node*next;
}list;
```

Example 2:

```
typedef struct
{
        char data;
        struct list * link;
}list;
```

In this example each instance of the structure list have two components data and link.
Data is a single character, while link is a pointer to a list structure.
The value of link is either the address in memory of an instance of list or NULL pointer.

Program to illustrate self-referential structures

```
#include  <stdio.h>
#include <conio.h>
typedef struct
{
        char data;
        struct list *link;
}list;
void main()
{
        list l1,l2,l3;
        l1.data='a';
        l2.data='b';
        l3.data='c';
        l1.link=l2.link=l3.link =NULL;
        l1.link=&l1;
        l2.link=&l2;
        printf("\n data values of l1=%d,l2=%d,l3=%d",l1.data,l2.data,l3.data);
        printf("\n link of l1,l2,l3=%d %d %d",l1.link,l2.link,l3.link);
        getch();
}
```

Difference between structure and union

1.The keyword **struct** is used to define a Structure	1. The keyword union is used to define a union.
2. When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members. The smaller members may end with unused slack bytes.	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
3. Each member within a structure is assigned unique storage area of location.	3. Memory allocated is shared by individual members of union.
4. The address of each member will be in ascending order. This indicates that memory for each member will start at different offset values.	4. The address is same for all the members of a union. This indicates that every member begins at the same offset value.
5 Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. Individual member can be accessed at a time	6. Only one member can be accessed at a time.
7. Several members of a structure can initialize at once. Ex: struct Book { int isbn; float price; char title[20]; }book; Total memory reserved will be Sizeof(int)+sizeof(float)+(20*sizeof(char))	7. Only the first member of a union can be initialized. Ex: union Book { int isbn; float price; char title[20]; }book; Total memory reserved will be Max(Sizeof(int)+sizeof(float)+(20*sizeof(char))

1.5 POINTERS

"Pointers are variables that hold address of another variable of same data type".

Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language.

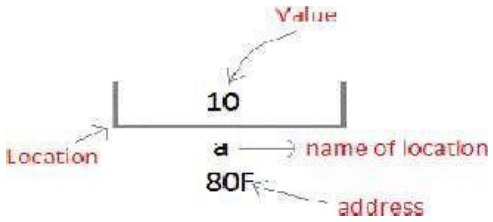
Benefit of using pointers

- Pointers are more efficient in handling Array and Structure.
- Pointer allows references to function and thereby helps in passing of function as arguments to other function.
- It reduces length and the program execution time.
- It allows C to support dynamic memory management.

Concept of Pointer

Whenever a **variable** is declared, system will allocate a location to that variable in the memory, to hold value. This location will have its own address number. Let us assume that system has allocated memory location 80F for a variable **a**.

Example : int a = 10 ;



The value 10 can be accessed by either using the variable name **a** or the address 80F.Since the memory addresses are simply numbers they can be assigned to some other variable. The variable that holds memory address are called **pointer variables**. A **pointer** variable is therefore nothing but a variable that contains an address, which is a location of another variable. Value of **pointer variable** will be stored in another memory location.

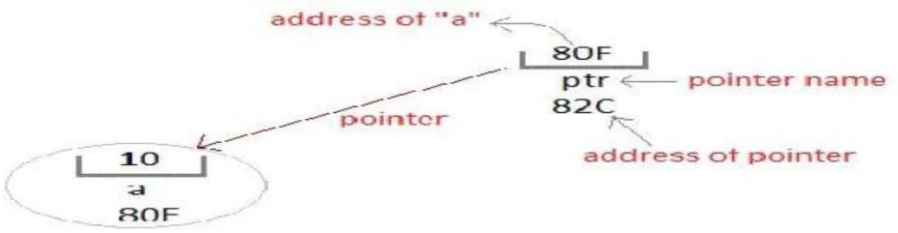


Figure 5: Pointer variable

Declaring a pointer variable

General syntax of pointer declaration is, ***data-type *pointer_name;***

Data type of pointer must be same as the variable, which the pointer is pointing. **void** type pointer works with all data types, but isn't used oftenly.

Initialization of Pointer variable

Pointer Initialization is the process of assigning address of a variable to **pointer** variable. Pointer variable contains address of variable of same data type. In C language **address operator** & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10 ;
int *ptr ; //pointer declaration
```

ptr = &a ; *//pointer initialization*
or,

int *ptr = &a ; *//initialization and declaration together*

Pointer variable always points to same type of data.

```
float a;
float *ptr;
ptr = &a;
```

Dereferencing of Pointer

Once a pointer has been assigned the address of a variable. To access the value of variable, pointer is dereferenced, using the **indirection operator** *.

```
void main()
{
        int a,*p;
        a = 10;
        p = &a;
        printf("%d",*p); //this will print the value of a.
        printf("%d",*&a); //this will also print the value of a.
        printf("%u",&a); //this will print the address of a.
        printf("%u",p); //this will also print the address of a.
        printf("%u",&p); //this will also print the address of p.
}
```

Pointer and Arrays

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address which gives location of the first element is also allocated by the compiler.

Suppose we declare an array **arr**,

int arr[5]={ 1, 2, 3, 4, 5 };

Assuming that the base address of **arr** is 1000 and each integer requires two byte, the five element will be stored as follows in figure 6

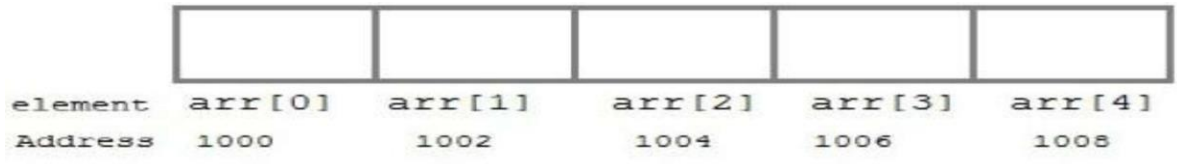


Figure 6: Array Representation

Here variable **arr** will give the base address, which is a constant pointer pointing to the element, **arr[0]**. Therefore **arr** is containing the address of **arr[0]** i.e 1000.

We can declare a pointer of type int to point to the array **arr**.

```
int *p;
p = arr;
```

or p = &arr[0]; //both the statements are equivalent.

Now we can access every element of array **arr** using **p++** to move from one element to another. **NOTE :** You cannot decrement a pointer once incremented. p-- won't work.

Pointer to 1-D Array

As studied above, we can use a pointer to point to an Array, and then we can use that pointer to access the array.

Lets have an example,

```
int i;
int a[5] = {1, 2, 3, 4, 5};
int *p = a; // same as int*p = &a[0]
for (i=0; i<5; i++)
{
printf("%d", *p);
p++;
}
```

In the above program, the pointer ***p** will print all the values stored in the array one by one. We can also use the Base address (**a** in above case) to act as pointer and print all the values.

Replacing the printf("%d",*p); statement of above example ,with below mentioned statements. Let's see what will be the result

- printf("%d",a[i]); prints the array, by incrementing index
- printf("%d",i[a]); this will also print elements of array
- printf("%d",a+i); this will also print address of all elements of array
- printf("%d",*(a+i)); this will also print values of all elements of array
- printf("%d",*a); this will print value of a[0] only
- a++; compile time error, we cannot change base address of the array.

Pointers to multidimensional array

A multidimensional array is of form, a[i][j] . Lets see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In a[i][j] , **a** will give the base address of this array, even a+0+0 will also give the base address, that is the address of **a[0][0]** element. Here is the generalized form for using pointer with multidimensional arrays.

***(*(ptr + i) + j)** *is same as* **a[i][j]**

Example program for pointers 2D-array

```
#include<stdio.h>
int main()
```

```
{
    int a[][3] = {1,2,3,4,5,6};
int (*ptr)[3] = a;  //passing the address of 1st 1-D array to pointer
printf("%d%d",  (*ptr)[1], (*ptr)[2]);
++ptr; //points to 2nd 1-D array
printf("%d%d",  (*ptr)[1], (*ptr)[2]);
return 0;
}
```

Output: 2 3 5 6

2. ARRAYS

Array is a container which can hold fix number of items and these items should be of same type. Most of the data structures make use of array to implement their algorithms. Following are important terms to understand the concepts of Array.

An *array* is a data structure that is a collection of variables of one type that are accessed through a common name. Each element of an array is given a number by which we can access that element which is called an index.To refer to a particular location or element in the array we specify the name to the array and position number of particular element in the array.

Element − each item stored in an array is called an element.

Index − each location of an element in an array has a numerical index which is used to identify the element.

2.1 LINEAR ARRAY REPRESENTATION

Arrays can be declared in various ways in different languages. For illustration, let's take

C array declaration

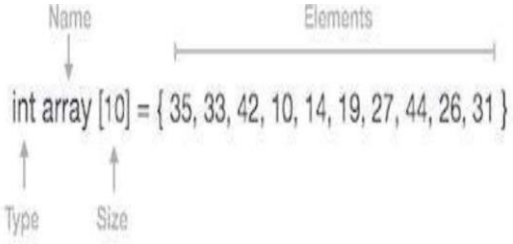


Figure 8a: Array with 10 elements

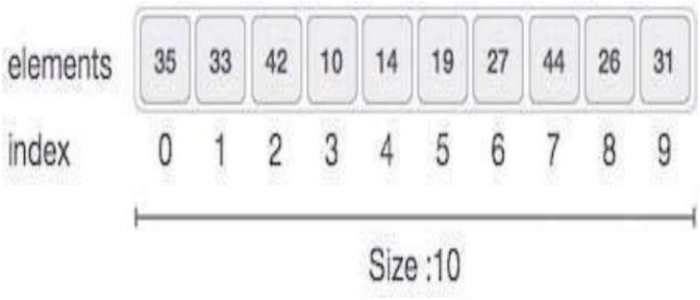


Figure 8b: Array index

As per above shown illustration, following are the important points to be considered.(As shown in figure 8a and figure 8b)

- Index starts with 0.
- Array length is 10 which mean it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch element at index 6 as 27.

ADT ARRAY

Objects: A set of pairs<index,value > where for each of index there is a value from the set item. Index is a finite ordered set of one or more dimensions , for example {0……,n-1} for one dimension, { (0 ,0),(0,1),(0.2),(1,0),(1,2),(1,1),(2,1),(2,2),(2,0))} for two dimensions etc.

Functions:

For all A C Array, I C index, x C item, j, size C integer

1. Array create (j, list) = return an array of j dimensions where list is a j-tuple whose ith element is the size of ith dimension. Items are undefined.

2. Item retrieve (A, i) = if (I C index) return the item associated with index value i in array A else return error.

3. array store (A, i, x) = if (i in index) return an array that is identical to array A expect the new pair <i,x> has been inserted else return error.

end Array

One Dimensional Array

Declaration:

Before using the array in the program it must be declared

Syntax: ***data_type array_name[size];***

Where , data_type represents the type of elements present in the array. array_name represents the name of the array. Size represents the number of elements that can be stored in the array.

Example: ***int age[100]; float sal[15]; char grade[[20];***

Here age is an integer type array, which can store 100 elements of integer type. The array sal is floating type array of size 15, can hold float values. Grade is a character type array which holds 20 characters.

Initialization:

Initialize arrays at the time of declaration.

Syntax: ***data_type array_name[size]={value1, value2,……..valueN};***

Where ,value1, value2, valueN are the constant values known as initializers, which are assigned to the array elements one after another.

Example: ***int marks[5]={10,2,0,23,4};***

The values of the array elements after this initialization are:

marks[0]=10, marks[1]=2, marks[2]=0, marks[3]=23, marks[4]=4;

NOTE:

1. **address of an data element in the array can be calculated as:**

A[K]=BA(A)+W(K-LOWERBOUND);

Where, A is an array, K is the index of the element for which address has to be calculated ,BA is the base address of the array A,and W is the size of one element in memory

2. **calculating the length of an array**

Length = Upperbound-Lowerbound+1

Where , upperbound is index of the last element and lowerbound is index of the first element in the array

Processing: For processing arrays we mostly use for loop. The total no. of passes is equal to the no. of elements present in the array and in each pass one element is processed.

Example: This program reads and displays 3 elements of integer type.

```
#include<stdio.h>
main()
{
        int a[3],i;
        for(i=0;i<=2;i++) //Reading the array values
        {
                printf("enter the elements"); scanf("%d",&a[i]);
        }
        for(i=0;i<=2;i++) //display the array values
        {
                printf("%d",a[i]); printf("\n");
        }
}
```

Example: C Program to Increment every Element of the Array by one & Print Incremented Array.

```
#include <stdio.h> void main()
{
        int i;
        int array[4] = {10, 20, 30, 40};
        for (i = 0; i < 4; i++)
        arr[i]++;
        for (i = 0; i < 4; i++)
        printf("%d\t", array[i]);
}
```

2.2 OPERATIONS ON ARRAYS

Following are the basic operations supported by an array.

- **Traversal –** processing each element in the list.
- **Insertion –** adding a new element at given index to the list.
- **Deletion –** removing an element at givenindex from the list.
- **Search** − finding the location of the element with a given value or the record with a given key
- **Sorting:** arranging the elements in some type of order.
- **Merging:** Combing two lists into a single list.

1. Traversing linear arrays

Let A be the array in the memory of the computer. If the operation requied is to print the contents of each element of A OR Count the number of elements of A. Then this is accomplished by traversing A.

Traversing is **accessing and processing each element in the array A exactly once.**

Algorithm: (Traversing a Linear Array)

Hear LA is a linear array with the lower bound LB and upper bound UB. This algorithm traverses LA applying an operation PROCESS to each element of LA **using while loop.**

1. [Initialize Counter] set K:= LB
2. Repeat step 3 and 4 while K ≤ UB
3. [Visit element] Apply PROCESS to LA [K]
4. [Increase counter] Set K:= K + 1

[End of step 2 loop]

5. Exit

2. Inserting

- Let A be a collection of data elements stored in the memory of the computer. Inserting refers to the operation of adding another element to the collection A.
- Inserting an element at the "end" of the linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element.
- Inserting an element in the middle of the array, then on average, half of the elements must be moved downwards to new locations to accommodate the new element and keep the order of the other elements.

Algorithm:

INSERT(LA,N,K,ITEM)

HERE LA is a linear array with N elements and K is a positive integer such that K<=N. this algorithm inserts an element ITEM into Kth position in LA.

1.[Initalize the counter]. Set J=N.

2.repeat steps 3 and 4 while J>=K

3.[move jth element downward] set LA[J+1]=LA[J].

4.[Decrease the counter] set J=J-1

[end of step 2 loop]

5.[insert element] set LA[K]A=ITEM

6.[reset N] set N=N+1

7.Exit

3. Deleting

Algorithm:

DELETE(LA,N,K,ITEM)

HERE LA is a linear array with N elements and K is a positive integer such that K<=N. this algorithm deletes an element ITEM into Kth position in LA.

1. Set ITEM=LA[K]

2. repeat for J=K to N-1

3.[move j+1st element upward] set LA[J]=LA[J+1].

4.[end of step 2 loop]

5.[reset N] set N=N-1

6.Exit

4. Sorting

Algorithm:

BUBBLE SORT(DATA,N)

Here DATA is an array with N elements . THIS Algorithm sorts the elemts in DATA.

1. repeat steps 2 and 3 for k=1 to N-1
2. [Initalize the pass pointer PTR]. Set PTR=1.
3. Repeat while PTR<=N-K :[Executes pass]

(a) if DATA[PTR]>DATA[PTR+1] then

Interchange DATA[PTR] and DATA[PTR+1]

[end of IF structure]

(b) set PTR=PTR+1

[End of inner loop]

[end of step1 outer loop]

4.Exit

5. Searching

Algorithm:

- **LINEAR SEARCH(DATA,N,ITEM,LOC)**

Here DATA is a linear array with N elements and ITEM is a given item of information. this algorithm finds the location LOC of item in DATA or set LOC=0 if the search is unsuccessful

1.[Insert ITEM at the end of DATA]. Set DATA[N+1]=ITEM.

2.[initialize the counter] set LOC=0,FOUND=0

3.[Search for ITEM]

Repeat for j=0 to N-1

If(ITEM=DATA[J]) then SET FOUND=1 and break

[end of loop]

4.[Succesful?] if FOUND=1Then SUCCESSFUL

else UNSUCCESSFUL

6. Exit

- **BINARY SEARCH(DATA,LB,UB,ITEM,LOC)**

Here DATA is a sorted array with lower bound LB and upper bound UB and ITEM is a given item of information. The variables BEG, END and MID denote respectively the beginning, end and middle locations of a segment of elements of data. This algorithm finds the location LOC of item in DATA or sets LOC=NULL

1.[Initialize segment variables].

Set BEG=LB,END=UB and MID=INT((BEG+END)/2);

2. repeat steps 3 and 4 while BEG<=END and DATA[MID]!=ITEM

3. if ITEM<DATA[MID] then

Set END=MID-1

ELSE

Set BEG=MID+1

[end of loop]

4. set MID=INT(BEG+END)/2

5. if DATA[MID]=ITEM Then

set LOC=MID

else

set LOC=NULL

6. Exit

TWO DIMENSIONAL ARRAYS

Arrays that we have considered up to now are one dimensional array, a single line of elements. Often data come naturally in the form of a table, e.g. spreadsheet, which need a two-dimensional array.

Declaration: The syntax is same as for 1-D array but here 2 subscripts are used.

Syntax: **data_type array_name[rowsize][columnsize];**

Where, Rowsize specifies the no.of rows Columnsize specifies the no.of columns.

Example: ***int a[4][5];***

This is a 2-D array of 4 rows and 5 columns. Here the first element of the array is a[0][0] and last element of the array is a[3][4] and total no.of elements is 4*5=20.

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
Row 2	a[2][1]	a[2][2]	a[2][2]	a[2][3]	a[2][4]
Row 3	a[3][0]	a[3][1]	a[3][2]	3a[3][3]	a[3][4]

Initialization:

2-D arrays can be initialized in a way similar to 1-D arrays.

Example: ***int m[4][3]={1,2,3,4,5,6,7,8,9,10,11,12};***

Example: ***int m[][3]={ {1,10}, {2,20,200}, {3}, {4,40,400} };***

2D ARRAY REPRESENTATION USING COLUMN MAJOR ORDER AND ROW MAJOR ORDER

1. In case of **Column Major Order:**

The formula is:

LOC (A [J, K]) = Base (A) + w [M (K-1) + (J-1)]

Here

LOC (A [J, K])	:	is the location of the element in the Jth row and Kth column.
Base (A)	:	is the base address of the array A.
w	:	is the number of bytes required to store single element of the array A.
M	:	is the total number of rows in the array.
J	:	is the row number of the element.
K	:	is the column number of the element.

E.g.

A 3 x 4 integer array A is as below:

Subscript	Elements	Address
(1,1)	10	1000
(2,1)	20	1002
(3,1)	50	1004
(1,2)	60	1006
(2,2)	90	1008
(3,2)	40	1010
(1,3)	30	1012
(2,3)	80	1014
(3,3)	75	1016
(1,4)	55	1018
(2,4)	65	1020
(3,4)	79	1022

Suppose we have to find the location of A [3, 2]. The required values are:

Base (A)	:	1000
w	:	2 (because an integer takes 2 bytes in memory)
M	:	3
J	:	3
K	:	2

Now put these values in the given formula as below:

LOC (A [3, 2]) = 1000 + 2 [3 (2-1) + (3-1)]

= 1000 + 2 [3 (1) + 2]

= 1000 + 2 [3 + 2]

= 1000 + 2 [5]

= 1000 + 10 = 1010

2. In case of **Row Major Order**:

The formula is:

LOC (A [J, K]) = Base (A) + w [N (J-1) + (K-1)]

Here

LOC (A [J, K])	:	is the location of the element in the Jth row and Kth column.
Base (A)	:	is the base address of the array A.
w	:	is the number of bytes required to store single element of the array A.
N	:	is the total number of columns in the array.
J	:	is the row number of the element.
K	:	is the column number of the element.

E.g.

A 3 x 4 integer array A is as below:

Subscript	Elements	Address
(1,1)	10	1000
(1,2)	60	1002
(1,3)	30	1004
(1,4)	55	1006
(2,1)	20	1008
(2,2)	90	1010
(2,3)	80	1012
(2,4)	65	1014
(3,1)	50	1016
(3,2)	40	1018
(3,3)	75	1020
(3,4)	79	1022

Suppose we have to find the location of A [3, 2]. The required values are:

Base (A)	:	1000
w	:	2 (because an integer takes 2 bytes in memory)
N	:	4
J	:	3
K	:	2

Now put these values in the given formula as below:

LOC (A [3, 2]) = 1000 + 2 [4 (3-1) + (2-1)]

= 1000 + 2 [4 (2) + 1]

= 1000 + 2 [8 + 1]
= 1000 + 2 [9]

= 1000 + 18

= 1018

Example 1:

Write a C program to find sum of two matrices

```
#include <stdio.h>
#include<conio.h>
```

```
void main()
{
        float a[2][2], b[2][2], c[2][2];
        int i,j;
        clrscr();
        printf("Enter the elements of 1st matrix\n");
        /* Reading two dimensional Array with the help of two for loop. If there is an array of 'n'
        dimension, 'n' numbers of loops are needed for inserting data to array.*/
        for(i=0;i<2;I++)
        for(j=0;j<2;j++)
        {
                scanf("%f",&a[i][j]);
        }
        printf("Enter the elements of 2nd matrix\n");
        for(i=0;i<2;i++)
        for(j=0;j<2;j++)
        {
                scanf("%f",&b[i][j]);
        }
        /* accessing corresponding elements of two arrays. */ for(i=0;i<2;i++)
        for(j=0;j<2;j++)
        {
                c[i][j]=a[i][j]+b[i][j]; /* Sum of corresponding elements of two arrays. */
        }
        /* To display matrix sum in order. */
        printf("\nSum Of Matrix:");
        for(i=0;i<2;i++)
        {
                for(j=0;j<2;j++)
                printf("%f\t", c[i][j]);
                printf("\n");
        }
        getch();
}
```

Example 2: Program for multiplication of two matrices

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int i,j,k;
        int row1,col1,row2,col2,row3,col3;
        int mat1[5][5], mat2[5][5], mat3[5][5];
        clrscr();
```

```
printf("\n enter the number of rows in the first matrix:");
scanf("%d", &row1);
printf("\n enter the number of columns in the first matrix:");
scanf("%d", &col1);
printf("\n enter the number of rows in the second matrix:");
scanf("%d", &row2);
printf("\n enter the number of columns in the second matrix:");
scanf("%d", &col2);
if(col1 != row2)
{
        printf("\n The number of columns in the first matrix must be equal to the number of rows
in the second matrix ");
        getch(); exit();
}
row3= row1; col3= col2;
printf("\n Enter the elements of the first matrix");
for(i=0;i<row1;i++)
{
        for(j=0;j<col1;j++)
        scanf("%d",&mat1[i][j]);
}
printf("\n Enter the elements of the second matrix");
for(i=0;i<row2;i++)
{
        for(j=0;j<col2;j++)
        scanf("%d",&mat2[i][j]);
}
for(i=0;i<row3;i++)
{
        for(j=0;j<col3;j++)
        {
        mat3[i][j]=0;
        for(k=0;k<col3;k++)
                mat3[i][j] +=mat1[i][k]*mat2[k][j];
        }
}
printf("\n The elements of the product matrix are"):
for(i=0;i<row3;i++)
{
        printf("\n");
        for(j=0;j<col3;j++)
        printf("\t %d", mat3[i][j]);
}
```

```
    return 0;
    }
```

Output:

```
Enter the number of rows in the first matrix: 2
Enter the number of columns in the first matrix: 2
Enter the number of rows in the second matrix: 2
Enter the number of columns in the second matrix: 2
Enter the elements of the first matrix
1 2 3 4
Enter the elements of the second matrix
5 6 7 8
The elements of the product matrix are
19 22
43 50
```

Example 3: Program to find transpose of a matrix.

```
#include <stdio.h>
int main()
{
    int a[10][10], trans[10][10], r, c, i, j;
    printf("Enter rows and column of matrix: ");
    scanf("%d %d", &r, &c);
    printf("\nEnter elements of matrix:\n");
    for(i=0; i<r; i++)
    for(j=0; j<c; j++)
    {
        printf("Enter elements a%d%d: ",i+1,j+1);
        scanf("%d", &a[i][j]);
    }
    /* Displaying the matrix a[][] */ printf("\n Entered Matrix: \n");
    for(i=0; i<r; i++)
    for(j=0; j<c; j++)
    {
        printf("%d ",a[i][j]);
        if(j==c-1)
        printf("\n\n");
    }
    /* Finding transpose of matrix a[][] and storing it in array trans[][]. */
    for(i=0; i<r;i++)
    for(j=0; j<c; j++)
    {
        trans[j][i]=a[i][j];
    }
```

```
        /* Displaying the array trans[][]. */ printf("\nTranspose of Matrix:\n");
        for(i=0; i<c;i++)
        for(j=0; j<r;j++)
        {
                printf("%d ",trans[i][j]);
                if(j==r-1)
                printf("\n\n");
        }
        return 0;
}
```

4. DYNAMIC MEMORY ALLOCATION

The memory allocation which is used till now was static memory allocation. So the memory that could be used by the program was fixed. So it is not possible to allocate or de allocate memory during the execution of the program. It is not possible to predict how much memory will be needed by the program at run time.

For example assume an array with size 20 elements is declared, which is fixed. So if at run time values to be stored in array are less than 20 then wastage of memory occur or our program may fail if more than 20 values are to be stored in to that array. To solve the above problems and allocate memory during runtime dynamic memory allocation is used.

The process of allocating memory during the runtime is called as DMA (dynamic memory allocation) and memory gets allotted in heap area of program stack.

Example : a new area of memory is allocated using malloc().On success, malloc() returns a pointer to **the first byte of allocated memory**. The returned pointer is of type void, which can be type cast to appropriate type of pointer. The memory allocated by malloc() contains **garbage value** . when the requested memory **is not available**, the pointer **NULL is returned**. when allocated memory is no longer required , it is freed by using another function free().

```
        void main()
        {
                int i,*pi;
            float f,*pf;
            pi=(int*)malloc(sizeof(int));
            pf=(float*)malloc(sizeof(float));
            *pi=1024;
            *pf=3.14;
        printf("an interger=%d,an float number=%f",*pi,*pf);
            free(pi);
```

```
    free(pf);
    getch();
}
```

The following functions are used in C for dynamic memory allocation and are defined in <stdlib.h>

1. malloc(size): Memory Allocation

This function is used to allocate memory dynamically. malloc() allocate a **single large block of contiguous memory** according to the size specified. The argument size specifies the number of bytes to be allocated.

On **success**, malloc() returns a pointer to the address of first byte of allocated memory. The returned pointer is of type void, which can be type cast to appropriate type of pointer. The memory allocated by malloc() contains garbage value .

If there is **insufficient** memory to make the allocation, the returned value is **NULL**.

Declaration: ***void *malloc(size_t size);***

(datatpe*) ptr = (datatpe*)malloc(sizeof(datatype));

Where,

ptr is a pointer variable of data_type

size is the number of bytes

Ex: int *ptr;

ptr = (int *) malloc(100*sizeof(int));

2. calloc(n, size): Contiguous Allocation

This function is used to allocate multiple blocks of contiguous memory.

It takes 2 arguments. The **first** argument specifies the number of blocks and the **second** one specifies the size of each block. The memory allocated by calloc() is initialized to **zero**.

On **success**, calloc() returns a pointer to the address of first byte of allocated memory. The returned pointer is of type void, which can be type cast to appropriate type of pointer

If there is **insufficient memory** to make the allocation, the returned value is **NULL.**

Declaration: ***void *calloc(size_t n, size_t size);***

(datatype*) ptr=(datatype*)calloc(n,sizeof(datatype));

Where,

ptr is a pointer variable of type int

n is the number of block to be allocated

size is the number of bytes in each block

Ex: int *x =(int*) calloc (10, sizeof(int));

The above example is used to define a one-dimensional array of integers. The capacity of this array is n=10 and x [0: n-1] (x [0, 9]) are initially 0

Macro CALLOC

```
#define CALLOC (p, n, s)
if ( ! ((p) = calloc (n, s)))\
{ fprintf(stderr, "Insuffiient memory");
 exit(EXIT_FAILURE);}
```

3. realloc( ): realloc(ptr, size)

The function realloc() is used to change the size of the memory block. It alters the size of the memory block without losing the old data. This function takes two arguments, first is a pointer to the block of memory that was previously allocated by malloc() or calloc() and second one is the new size for that block.

- Before using the realloc( ) function, the memory should have been allocated using malloc( ) or calloc( ) functions.
- The function relloc( ) resizes memory previously allocated by either mallor or calloc, which means, the size of the memory changes by extending or deleting the allocated memory.
- If the existing allocated memory need to extend, the pointer value will not change.
- If the existing allocated memory cannot be extended, the function allocates a new block and copies the contents of existing memory block into new memory block and then deletes the old memory block.
- When realloc() is able to do the resizing, it returns a pointer to the address of first byte of the new block and when it is unable to do the resizing, the old block is unchanged and the function returns the value NULL

Declaration: ***void *realloc(void *ptr ,size_t newsize);***

(datatype*)ptr=(datatype*)realloc(ptr, newsize)

Example: (int *)ptr = (int *) malloc(sizeof(int));

ptr = (int *) realloc(ptr, 2*sizeof(int));

Macro REALLOC

```
#define REALLOC(p,S)\
if (!((p) = realloc(p,s)))
{ \ fprintf(stderr, "Insufficient memory");
exit(EXIT_FAILURE);\ }\
```

4. free( )

Dynamically allocated memory with either malloc( ) or calloc ( ) does not return on its own. The programmer must use free( ) explicitly to release space.

This function is used to release the memory space allocated dynamically. The memory released by free() is made available to the heap again and can be used for some other purpose. We should not try to free any memory location that was not allocated by malloc(), calloc() or realloc().

Syntax: **free(ptr);**

ptr = NULL;

This statement cause the space in memory pointer by ptr to be deallocated

The following program illustrates Dynamic memory allocation.

```
void main()
{
        int *p,n,i;
        printf("Enter the number of integers to be entered");
        scanf("%d",&n);
        p=(int *)malloc(n*sizeof(int)); /* This is same as "(int *)calloc(n,sizeof(int))"*/
        /* If we write "(int *)malloc(sizeof(int))" then only 2 byte of memory will be allocated
        dynamically*/
        if(p==NULL)
        {
                printf("Memory is not available"); exit(1);
        }
        for(i=0;i<n;i++)
        {
                printf("Enter an integer");
                 scanf("%d",p+i);
        }
        for(i=0;i<n;i++)
         printf("%d\t",*(p+i));
}
```

2.4 DYNAMICALLY ALLOCATED ARRAYS

Array can be dynamically allotted using malloc(), calloc() or realloc() functions, similarly the allotted memory can be freed after the use of array using free() function.

One dimensional array

When large programs are written, it is difficult to determine how large array to use. So, solution to this problem is to defer this decision to runtime and allocate the required array size. The **advantage** of dynamic array is that the memory for the array of any desired size can be allotted. There is no need to declare a fixed size array.

During the use of dynamically allocated arrays the following changes are required in first few lines of main function of program

```
int i ,n,*list;
printf("\n enter the n value to generate");
scanf("%d",&n);
if(n<1)
{
        fprintf(stderr,"improper value of \n");
        exit(EXIT_FAILURE);
}
MALLOC(list,n*sizeof(int));
```

This above main function fails only when n<1 or when sufficient memory is not allotted.

Example program for illustration of use of dynamic array:-

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
        int i,n;
        int *ptr;
        printf("\n enter the number of elements\n");
        scanf("%d",&n);
        ptr=(int*)malloc(sizeof(int)*n);
        if(ptr==NULL)
        {
                printf("\n insufficient memory");
                return;
        }
        printf("\n enter the n elements");
        for(i=0;<n;i++)
        scanf("%d",(ptr+i));
        printf("the given array elements are\n");
        for(i=0;i<n;i++)
        printf("%d",*(ptr+i));
        getch();
}
```

Two dimensional array

A two dimensional array is represented as a one dimensional array in which each element is itself a one dimensional array.(As shown in figure 9)

int x[3][5];

Here, actually a one dimensional array x is created whose length is 3 and each element of x is a one dimensional array whose length is 5.

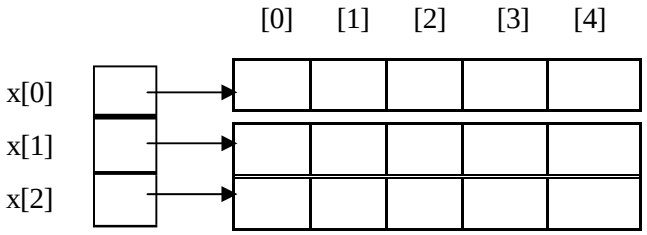


Figure 9: Two dimensional array

C finds the element x[i][j] by first accessing the point in x[i].this pointer gives the address in memory of the zeroth element of row I of the array. Then by adding j*sizeof(int) to this pointer , the address of the [j]th elemnt of row i is determined.

Example program for allocating memory dynamically for 2D array

```
 #include<stdio.h>
#include<conio.h>
#include<alloc.h>
void main()
{
        int **a;
        int p, q;
        int **make2darray();
        printf("\n enter the no of rows");
        scanf("%d",&p);
        printf("\n enter the no of cols");
        scanf("%d",&q);
        a=make2darray(p,q);
        printf("successful memory creation address is",a);
        getch();

}

int **make2darray(int rows ,int cols)
{
        int **x, i;
        x=malloc(rows*sizeof(*x));
        for(i=0;i<rows;i++)
                x[i]=malloc(cols*sizeof(**x));
        return x;
}
```

2.4 APPLICATIONS OF ARRAYS

The two major applications of the arrays are polynomial and sparse matrix

1. POLYNOMIALS

Polynomial is a sum of terms where each term has a form $_ax^e$, where x is the variable, a is the coefficient and e is the exponent.

Examples :

$A(x) = 3x^{20}+2x^5+4$

$B(x) = x^4+10x^3+3x^2+1$

The largest exponent of a polynomial is called its **degree.**

In the above example, degree of first polynomial is 20 and for the second polynomial it is 4.

Note: Coefficients that are zero are not displayed, the term with exponent zero does not show the variable i.e. $4x^2+5x+1$ where 1 is a term having exponent zero so variable is not displayed.

Operations on polynomials

- Addition
- Subtraction
- Multiplication
- But division is not allowed.

Polynomial addition is defined as :

Assume $\mathbf{A(x)=\sum a_i x^i}$ and $\mathbf{B(x)=\sum b_j x^i}$ then $\mathbf{A(x)+ B(x)=\sum (a_i+b_j)x^i}$

Polynomial multiplication is defined as $\mathbf{A(x).B(x)=\sum A(x)=\sum a_i x^i.(\sum b_j x^i))}$

Polynomial representation

In c, typedef is used to create the polynomial as below:-

```
#define MAX_DEGREE 101
typedef struct
{
        int degree;
        float coef[MAX_DEGREE];
}polynomial;
polynomial a;
```

Conisder a is of type polynomial and n>MAX_DEGREE then polynomial $\mathbf{A(x)=\sum a_i x^i}$ for i=0 to n would be represented as

a.degree=n;

a.coef[i]=a_{n-i},0<=i<=n.

Useful polynomial representation

To preserve space, an alternative polynomial representation is given below which uses only one global array, terms to store all polynomials

```
#define MAX_TERMS 100
typedef struct
{
    float coeff;
    int expon;
}polynomial;
polynomial terms[MAX_TERMS];
int avail=0;
```

ADT polynomial

Structure *Polynomial* is
objects: $p(x)=a_1x^e + . . a_nx^e$; a set of ordered pairs of <*ei,ai*> where *ai* in *Coefficients* and *ei* in *Exponents*, *ei* are integers >= 0
functions:
for all *poly, poly1, poly2* ∈ *Polynomial*, *coef* ∈*Coefficients*, *expon* ∈ *Exponents*
Polynomial Zero( ) ::= **return** the polynomial, *p(x)* = 0
Boolean IsZero(*poly*) ::= **if** (*poly*)
return *FALSE*
else
return *TRUE*
Coefficient Coef(*poly, expon*) ::= **if** (*expon* ∈ *poly*)
return its coefficient
else
return Zero
Exponent Lead_Exp(*poly*) ::= **return** the largest exponent in *poly*
Polynomial Attach(*poly,coef, expon*) ::= **if** (*expon* ∈ *poly*)
return error
else
return the polynomial poly with the term <*coef, expon*> inserted
Polynomial Remove(*poly, expon*)::= **if** (*expon* ∈ *poly*)
return the polynomial *poly* with the term whose exponent is *expon deleted*
else
return error
Polynomial SingleMult(*poly, coef, expon*) ::= **return** the polynomial *poly* • *coef* • *xexpon*
Polynomial Add(*poly1, poly2*) ::= **return** the polynomial *poly*1 +*poly*2
Polynomial Mult(*poly1, poly2*)::= **return** the polynomial *poly*1 • *poly*2
End *Polynomia*

Polynomial addition

Function which adds two polynomials A and B giving D, represented as D=A+B. using previous polynomial representation the following figure 10 shows how two polynomials are represented

Ex: **A(x)=2x^{1000}+1**

B(x) =x^4+10x^3+3x^2+1

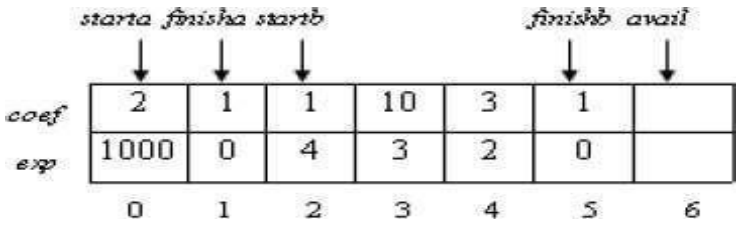


Figure 10: Array representation of polynomial

The index of the first term of A and B is given by startA and startB respectively,while finishA and finishB give the index of the last term of A and B. the index of the next free location in the array is given by avail.

Example **startA=0,finishA=1,startB=2,finish=5 and avail=6.**

This representation ha no limitation on the number of terms in a polynomial but total number of non-zero terms should not exceed MAX_TERMS.

Poly is used to refer a polynomial and is translated poly into a <start,finish>pair. Any polynomial A that has n nonzero terms has startA and finishA such that finishA=startA+n-1.

Function To Add Two Polynomials

```
void padd(int startA,int finishA,int startB, int finish,int *startD,int *FINISHd)
{
        float coefficient;
        *startd=avail;
        while(startA<=finiSHs && startb<=finish)
        switch(COMPARE(terms[startA].expon,terms[startB].expon))
        {
                case -1: /* a.expon<b.expon)*/
                                attach(terms[startB].coef,terms[startB].expon);
                                startB++;
                                break;
                case 0:/* equal exponents*/
                        coefficient=terms[startA].coef+terms[startB].coef;
                        if(coefficient)
                                attach(coefficient , terms[startA].expon);
                        startA++;
                        startB++;
                        break;
                case 1: /*a.expon>b.expon*/
                        attach(terms[startA].coef,terms[startA].expon);
                        startA++;
```

```
        }
        for(;startA<=finishA;startA++)
            attach(terms[startA].coef,terms[startA].expon);
        for(;startB<=finishB;startB++)
            attach(terms[startB].coef,terms[startB].expon);
        *finishD=avail-1;
}
```

Function To Add a new term

```
 void attach(float coefficient ,int exponent)
{
        if(avail>=MAX_TERMS)
        {
                printf(stderr,"too many terms in the polynomial \n");
                exit(0);
        }
        terms[avail].coef=coefficient;
        terms[avail++].expon=exponent;
}
```

2. SPARSE MATRICES

Matrix contains m rows and n columns of elements. it has m rows and n columns. In general mxn, is used to designate a matrix with m rows and n columns. The total no of elements in such matrix is m*n. If m equals n then matrix is square.

"Sparse matrix: A matrix containing more number of zero entries, such matrices is called as sparse matrix."

In figure11, since this matrix contains many zeros it is called as sparse matrix. Here 8 of 36 elements are only having non-zero values so it is called as sparse matrix. When sparse matrix is represented as two dimensional array hence space is wasted so need another form of representation to save memory where only non-zero values are stored.

	col 0	col 1	col 2	col 3	col 4	col 5
row 0	15	0	0	22	0	15
row 1	0	11	3	0	0	0
row 2	0	0	0	-6	0	0
row 3	0	0	0	0	0	0
row 4	91	0	0	0	0	0
row 5	0	0	28	0	0	0

Figure 11: sparse matrix

ADT sparse matrix

Structure *Sparse_Matrix* is
objects: a set of triples, <*row, column, value*>, where *row* and *column* are integers and form a unique combination, and*value* comes from the set *item.*
functions:
for all *a, b* ∈ *Sparse_Matrix*, *x* ∈ *item, i, j, max_col, max_row* ∈ *index*
Sparse_Marix Create(*max_row, max_col*) ::=
return a *Sparse_matrix* that can hold up to *max_items = max _row* x *max_col* and whose maximum row size is *max_row* and whose maximum column size is *max_col.*
Sparse_Matrix Transpose(*a*) ::=
return the matrix produced by interchanging the row and column value of every triple.
Sparse_Matrix Add(*a, b*) ::=
if the dimensions of a and b are the same
return the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.
else
return error
Sparse_Matrix Multiply(*a, b*) ::=
if number of columns in a equals number of rows in b
return the matrix *d* produced by multiplying a by *b* according to the formula:*d* [*i*] [*j*]=Σ(a[i][k]•b[k][j]) where *d (i, j)* is the *(i,j)*th element
else
return error.
End *Sparse_Matrix*

Sparse Matrix representation

A array of triples <row,col,value> is used to represent sparse matrix. In triples the row indices are in ascending order as well as column indices are also in ascending order.(as shown in figure 12). First row in triplet representation gives the total number of rows, total number of columns and total number of non-zero values in sparse matrix.

Create operation of sparse matrix is written as

```
#define MAX_TERMS 101
typdef struct
{
        int col;
        int rows;
        int value;
}term A[MAX_TERMS];
```

The above Sparse matrix is represented as triples.

sparse matrix as triples

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Figure 12: **Triple representation**

Function: Triplet of a sparse matrix.

```
create_triplet_matrix(term a[],term b[]){
    int  k=1;
    for(int i=0; i<rows; i++){
            for(int j=0; j<col; j++){
                    if(a[i][j]!=0){
                            b[k].row = i;
                            b[k].col = j;
                            b[k].value =a[i][j] ;
                            k++ }
            }
    }
    B[0].row = rows;
    B[0].col = col;
    B[0].value = k-1;
}
```

Transposing a Matrix

Transpose of a given matrix is obtained rows and columns. each element a[i][j].In the original matrix becomes element b[j][i] in the transpose matrix.
The following algorithm is given by

```
algorithm BAD_TRANS
for each row i
    take element <i,j,value>;
    store it as element <j,i,value> of the
      transpose;
end;
```

- problem: data movement

```
algorithm TRANS
for all elements in column j
    place element <i,j,value> in
    element <j,i,value>
end;
```

- problem: unnecessary loop for each column

The algorithm indicates that find all the elements in column 0 and store them in row 0 of the transpose matrix, find all the elements in column1 and store them in row1 etc. Since the original matrix ordered the rows, the columns within each row of incorporated in transpose. The first array a is the original array which the second array b holds the transpose.(As shown in figure13)

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

Figure 13: **Transpose of matrix**

Function: Transpose a sparse matrix.

```
void transpose(term a[],term b[])
{
        int n,i,j,currentb;
        n=a[0].value;
        b[0].row=a[0].col;
        b[0].col=a[0].row;
        b[0].value=n;
        if(n>0)
        {
                currentb=1;
                for(i=0;i<a[0].col;i++)
                        for(j=0;j<=n;j++)
                                if(a[j].col==i)
                                        {
                                                b[currentb].row=a[j].col;
                                                b[currentb].col=a[j].row;
                                                b[currentb].value=a[j].value;
                                                currentb++;
                                        }
        }
}
```

Fast Transpose of a Sparse Matrix

- This algorithm, fast-transpose proceeds by first determining the number of elements in each column of the orig
- This gives us the number of elements in each row of the transpose matrix. From this information, we can determ matrix.
- We now can move the elements in the original matrix one by one into their correct position in the transpose ma

```
void fast_transpose(term a[], term b[])
{
/* the transpose of a is placed in b */
   int row_terms[MAX_COL], starting_pos[MAX_COL];
   int i,j, num_cols = a[0].col, num_terms = a[0].value;
   b[0].row = num_cols;  b[0].col = a[0].row;
   b[0].value = num_terms;
   if (num_terms > 0) { /* nonzero matrix */
      for (i = 0; i < num_cols; i++)
         row_terms[i] = 0;
      for (i = 1; i <= num_terms; i++)
         row_terms[a[i].col]++;
      starting_pos[0] = 1;
      for (i = 1; i < num_cols; i++)
         starting_pos[i] =
                     starting_pos[i-1] + row_terms[i-1];
      for (i = 1; i <= num_terms; i++) {
         j = starting_pos[a[i].col]++;
         b[j].row = a[i].col;   b[j].col = a[i].row;
         b[j].value = a[i].value;
      }
   }
}
```

3. STRINGS

Strings are one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**. The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows −

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C

Note: Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array.

Let us try to print the above mentioned string :−

```
#include <stdio.h>
int main ()
{
        char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
        printf("Greeting message: %s\n", greeting );
        return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Greeting message: Hello
```

ADT STRING

Objects: a finite set of zero or more characters

Functions:

For all s, t C string,i,j,m C non-negative intergers

String Null(m) = return a string whose maximum length is m characters , but is initally set to NULL we write NULL as " ".

Integer Compare(s,t) = if s equals t return 0

else if s precedes t return -1

else retun +1

Boolean IsNull(s) = if(Compare(s,NULL)) return FALSE

else return TRUE

Integer Length(s) = if(Compare(s,NULL)) return the number of characters in s

else return 0

String Concat(s,t) = if(Compare(t,NULL)) return a string whose elements are those of s followed by those of t

else return 0

String Substr(s,i,j) = if((j>0) && (i+j-1) < Length (s)) return the string containing the characters of s at positions i, i+1, ..., i+j-1

else return NULL

3.2 STRING OPERATIONS

C supports a wide range of built in functions that manipulate null- terminated strings :–

strcpy(s1, s2):- Copies string s2 into string s1.

strcat(s1, s2):- Concatenates string s2 onto the end of string s1.

strlen(s1):- Returns the length of string s1.

strcmp(s1, s2):- Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1

Function	Description
char *strcat (char *dest, char *src)	Concatenate dest and src strings; return result in dest
char *strncat (char *dest, char *src, int n)	Concatenate dest and n characters from src; return result in dest
int strcmp (char *str1, char *str2)	Compare 2 strings; return < 0 if str1<str2; 0 if str1 = str2; >0 if str1 > str2
int strncmp (char *str1, char *str2, int n)	Compare first n characters; return < 0 if str1<str2; 0 if str1 = str2; >0 if str1 > str2
char *strcpy (char *dest, char *src)	Copy src into dest; return dest
char *strncpy (char *dest, char *src, int n)	Copy n characters from src into dest; return dest
size_t strlen (char *s)	Return the length of 's'
char *strchr(char *s, int c)	Return pointer to the first occurrence of c in s; return NULL if not present
char *strrchr(char *s, int c)	Return pointer to the last occurrence of c in s; return NULL if not present
char *strtok(char *s, char *delimiters)	Return a token from s;token is surrounded by delimiters
char *strstr(char *s, char *pat)	Return pointer to start of pat in s
size_t strspn (char *s, char *spanset)	Scan s for characters in spanset; return length of span
size_t strcspn (char *s, char *spanset)	Scan s for characters not in spanset; return length of span
char *strpbrk (char *s, char *spanset)	Scan s for characters in spanset; return pointer to first occurrence of a character from spanset

```
#include  <stdio.h>
#include <string.h>
int main ()
 {
        char str1[12] = "Hello";
        char str2[12] = "World";
        char str3[12];
        int len ;
        /* copy str1 into str3 */
        strcpy(str3, str1);
        printf("strcpy( str3, str1) : %s\n", str3 );
        strcat( str1, str2);
        printf("strcat( str1, str2): %s\n", str1 );
        /* total lenghth of str1 after concatenation */
        len = strlen(str1);
        printf("strlen(str1) : %d\n", len );
        return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

STRING INSERTION FUNCTION

```
void strnins(char *s,char *t,int  i)
{
        Char string[MAX_SIZE],*temp = string;
        if( i < 0 && i > strlen(s))
{ fprintf(stderr," position is out of bounds\n");
exit(EXIT_FAILURE);
        }
if(! strlen(s))
strcpy(s,t);
else if if( strlen(t))
```

```
{
strncpy(temp, s, i);
strcat(temp, t);
strcat(temp, (s+i));
        strcpy(s,temp);
        }
}
```

3.3 PATTERN MATCHING ALGORITHMS

C programming code to check if a given string is present in another string, For example the string "programming" is present in "c programming". If the string is present then it's location (i.e. at which position it is present) is printed. We create a function match which receives two character pointers and return the position if matching occurs otherwise returns -1. naive string search algorithm is implemented in this c program

Brute Force Pseudo-Code

Here's the pseudo-code

```
do
if (text letter == pattern letter)
compare next letter of pattern to next
letter of text
else
move pattern down text by one letter
while (entire pattern found or end of text)
```

1. **PATTERN MATCHING BY CHECKING END INDICES FIRST**

```
  int nfind(char *string, char *pat)
{
        int i , j , start = 0;
        int lasts = strlen(string)-1;
        int lastp = strlen(pat)-1;
        int endmatch=lenp;
        for( i = 0; endmatch<=lasts ; endmatch++, start++)
        {
                if( string[endmatch] == pat[lastp])
                for( j = 0 , i = start; j< lastp && string[i] == pat[j]; i++, j++)
                        ;
                if(j == lastp)
                return start;
        } return -1;
}
```

2. **KNUTH MORRIS PRATT STRING MATCHING ALGORITHMS**

```
int pmatch(char *string,char *pat)
```

```
    {
        int i = 0,j = 0;
        int lens = strlen(string);
        int lenp = strlen(pat);
        while( i < lens && j < lenp)
        {
            if( string[i] == pat[j])
            {
                i++,j++;
            }
                else if(j == 0) j++;
                else j = failure[j-1]+1;
        }
        return ( if(j == lenp) ? (i-lenp):-1);
}
```

Differences between malloc() and calloc()

S.No.	malloc()	calloc()
1.	malloc() function creates a single block of memory of a specific size.	calloc() function assigns multiple blocks of memory to a single variable.
2.	The number of arguments in malloc() is 1.	The number of arguments in calloc() is 2.
3.	malloc() is faster.	calloc() is slower.
4.	malloc() has high time efficiency. Because no initialization takes place.	calloc() has low time efficiency. Because of zero filling.
5.	The memory block allocated by malloc() has a garbage value.	The memory block allocated by calloc() is initialized by zero.
6.	malloc() indicates memory allocation.	calloc() indicates contiguous allocation.
7.	*Syntax:* *ptr=(datatpe*)malloc(sizeof(datatype));*	*Syntax:* *ptr=(datatpe*)calloc(n, sizeof(datatype));*

Pointers Can Be Dangerous

Because pointers provide access to a memory location. Data and executable code exist in memory together, misuses of pointers can lead to both bizarre effects and very subtle errors.

Potential Problems with Pointers

- *uninitialized pointers,*
- *memory leakage* and
- *dangling pointers.*

Uninitialized pointers (wild ponters)

- Uninitialized pointer pose a significant thread.
  - the value stored in an uninitialized pointer could be randomly pointing anywhere in memory.
  - Storing a value using an uninitialized pointer has the potential to overwrite anything in your program, including your program itself
- Never write a declaration like **int *p;**

  Always give your pointers an initial value or Null if you can't make it point to a real data value. So, best practice is to pointers to NULL like

  Int *p=NULL;

Memory Leakage

A *memory leak* occurs when all pointers to a value allocated on the heap has been lost.

Over time, memory leaks can cause programs to slow down and, eventually, crash.

Worse, a leaky program may come to take up so much of a systems memory that it interferes with the operation of other programs on the same system.

Ex: Programmer creates a memory in heap and forget to delete it. This unused un-accessible memory results in memory leakage.

```
void main(){

int  *p;

p = (int *) malloc(sizeof(int));

Return;}
```

Dangling Pointers

Dangling pointers refer to a pointer which was pointing at an object that has been deleted.

The pointer still has the address of the object even though the memory for that object has been removed.

Ex: int main(){

```
int *ptr = (int *) malloc(sizeof(int));
 …….
……..
free(ptr);  // ptr is still pointing to the deallocated  memory  ie non-existing memory
return 0;}
```

PROGRAMMING EXAMPLES

1) Write a C program to sort N numbers in ascending order using Bubble sort and print both the given and the sorted array

```
#include <stdio.h>
#define MAXSIZE 10
void main()
{
   int array[MAXSIZE];
   int i, j, num, temp;

   printf("Enter the value of num \n");
   scanf("%d", &num);
   printf("Enter the elements one by one \n");
   for (i = 0; i < num; i++)
   {
      scanf("%d", &array[i]);
   }
   printf("Input array is \n");
   for (i = 0; i < num; i++)
   {
      printf("%d\n", array[i]);
   }
   /* Bubble sorting begins */
   for (i = 0; i < num; i++)
   {
      for (j = 0; j < (num - i - 1); j++)
      {
         if (array[j] > array[j + 1])
         {
            temp = array[j];
            array[j] = array[j + 1];
            array[j + 1] = temp;
         }
      }
   }
   printf("Sorted array is...\n");
   for (i = 0; i < num; i++)
   {
      printf("%d\n", array[i]);
   }
}
```

2) Write a C program to input N numbers and store them in an array. Do a linear search for a given key and report success or failure.

```
#include <stdio.h>
 void main()
{
   int array[10];
   int i, num, keynum, found = 0;
   clrscr();
   printf("Enter the value of num \n");
```

```
    scanf("%d", &num);
    printf("Enter the elements one by one \n");
    for (i = 0; i < num; i++)
    {
        scanf("%d", &array[i]);
    }
    printf("Input array is \n");
    for (i = 0; i < num; i++)
    {
            printf("%d\n", array[i]);
    }
    printf("Enter the element to be searched \n");
    scanf("%d", &keynum);
    /* Linear search begins */
    for (i = 0; i < num ; i++)
    {
        if (keynum == array[i] )
        {
            found = 1;
            break;
        }
    }
    if (found == 1)
        printf("Element is present in the array\n");
    else
        printf("Element is not present in the array\n");
}
```

3) Write a C program to input N numbers and store them in an array. Do a binary search for a given key and report success or failure.

```
#include<stdio.h>
main()
 {
 int a[20],i,j,d,t,x,l=0,low,mid,high;
 printf("\nEnter the number of elements\n");
 scanf("%d",&d);
 printf("Enter the numbers\n");
 for(i=0;i<d;i++)
 scanf("%d",&a[i]);
 for(i=0;i<d;i++)
  {
  for(j=i+1;j<d;j++)
   {
   if(a[i]>a[j])
    {
    t=a[i];
    a[i]=a[j];
    a[j]=t;
    }
   }
  }
```

```
printf("\nThe sorted list :");
for(i=0;i<d;i++)
printf("%d ",a[i]);
printf("\nEnter the number to be searched\n");
scanf("%d",&x);
low=0;
high=d-1;
while(low<=high)
 {
 mid=(low+high)/2;
 if(x<a[mid])
 high=mid-1;
 else if(x>a[mid])
 low=mid+1;
 else
  {
  if(x==a[mid])
   {
   l++;
   printf("The item %d is found at location %d\n",x,mid+1);
   exit(0);
   }
  }
 }
if(l==0)
printf("Item not found\n");
}
```

4. Program to illustrate union inside structure

```
#include <stdio.h>
struct student
{
        union
        {
                char name[20];
                int roll_no;
        };
        int marks;
};
int main()
{
        struct student stud;
        char choice;
        printf("\n you can enter the name or roll number of the student");
        printf("\n do you want to enter the name?(Y or N:");
        gets(choice);
        if(choice == 'Y' ||choice == 'y')
        {
             printf(" n enter the name:");
             gets(stud.name);
        }
```

```
                    \
else
{

        printf("\n enter the roll number:");
}       scanf("%d",&stud.roll_no);

printf("\n enter the marks:");
scanf("%d",&stud.marks);
if(choice == 'Y' || choice == 'y')
        printf("\n Name: %s",stud.name);
else
        printf("\n Roll number: %d", stud.roll_no);
printf("\n Marks: %d", stud.marks);
return 0;
}
```

QUEUES

DEFINITION

- "A queue is an **ordered list** in which insertions (additions, pushes) and deletions (removals and pops) take place at different ends."
- The end at which new elements are added is called the **rear**, and that from which old elements are deleted is called the **front**.

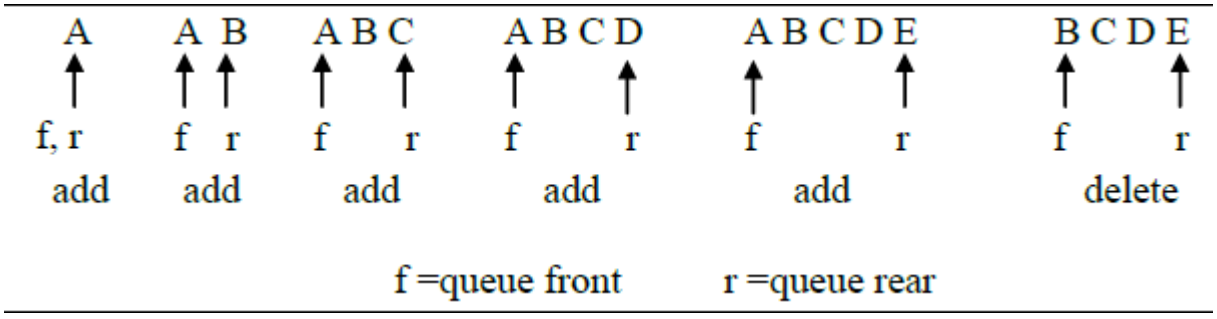
If the elements are inserted A, B, C, D and E in this order, then A is the first element deleted from the queue. Since the first element inserted into a queue is the first element removed, queues are also known as **First-In-First-Out (FIFO) lists.**

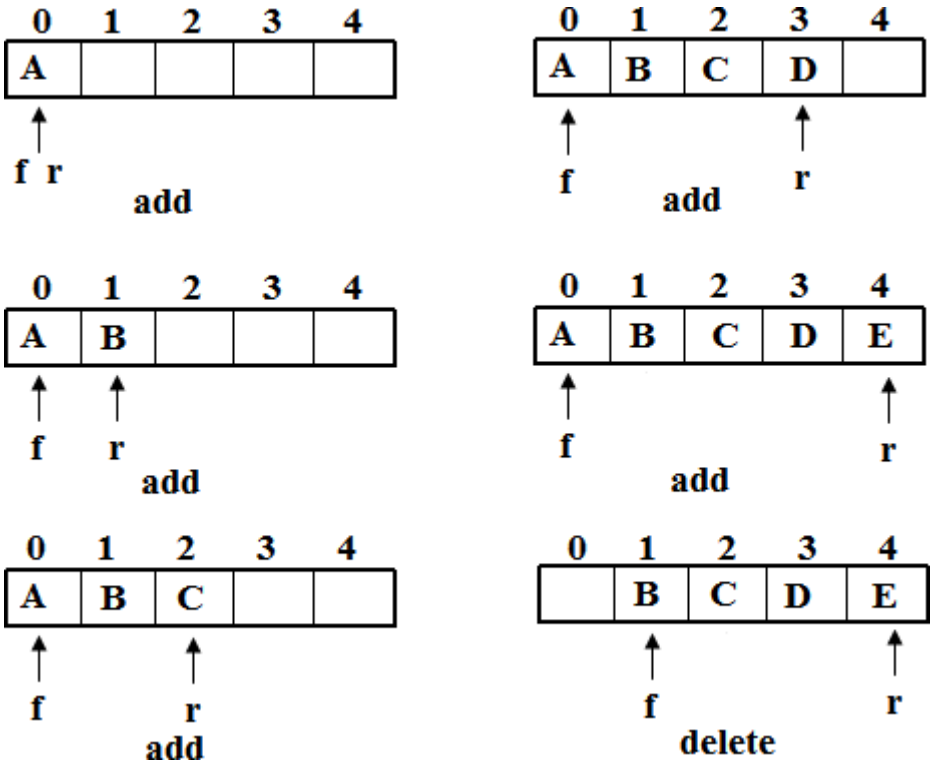
QUEUE REPRESENTATION USING ARRAY

- Queues may be represented by one-way lists or linear arrays.
- Queues will be maintained by a linear array QUEUE and two pointer variables:
  FRONT-containing the location of the front element of the queue
  REAR-containing the location of the rear element of the queue.
- The condition FRONT = NULL will indicate that the queue is empty.

Figure indicates the way elements will be deleted from the queue and the way new elements will be added to the queue.

- Whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment FRONT := FRONT + 1
- When an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment REAR := REAR + 1





QUEUE OPERATIONS

Implementation of the queue operations as follows.

1. Queue Create

```
Queue CreateQ(maxQueueSize) ::=
        #define MAX_QUEUE_SIZE  100          /* maximum queue size */
        typedef struct
                {
                        int key;             /* other fields */
                } element;
        element queue[MAX_QUEUE_SIZE];
        int rear = -1;
        int front = -1;
```

2. Boolean IsEmptyQ(queue) ::= front ==rear

3. Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

In the queue, two variables are used which are **front** and **rear.** The queue increments **rear** in addq( ) and **front** in delete( ). The function calls would be
addq (item); and **item =delete( );**

1. addq(item)

```
void addq(element item)
        {                       /* add an item to the queue */
                if (rear == MAX_QUEUE_SIZE-1)
                        queueFull();
                queue [++rear] = item;
        }
```

Program: Add to a queue

2. deleteq( )

```
element deleteq()
        {                       /* remove element at the front of the queue */
                if (front == rear)
                        return queueEmpty( );           /* return an error key */
                return queue[++front];
        }
```

Program: Delete from a queue

3. queueFull( )

The **queueFull** function which prints an error message and terminates execution

```
void queueFull()
        {
                fprintf(stderr, "Queue is full, cannot add element");
                exit(EXIT_FAILURE);
        }
```

Example: Job scheduling

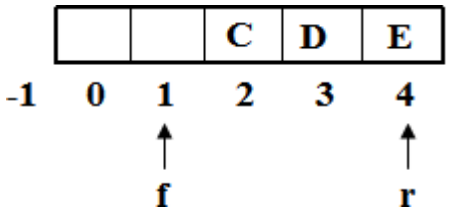
- Queues are frequently used in creation of a **job queue** by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system.
- Figure illustrates how an operating system process jobs using a sequential representation for its queue.

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					Queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Figure: Insertion and deletion from a sequential queue

Drawback of Queue

When item enters and deleted from the queue, the queue gradually shifts to the right as shown in figure.

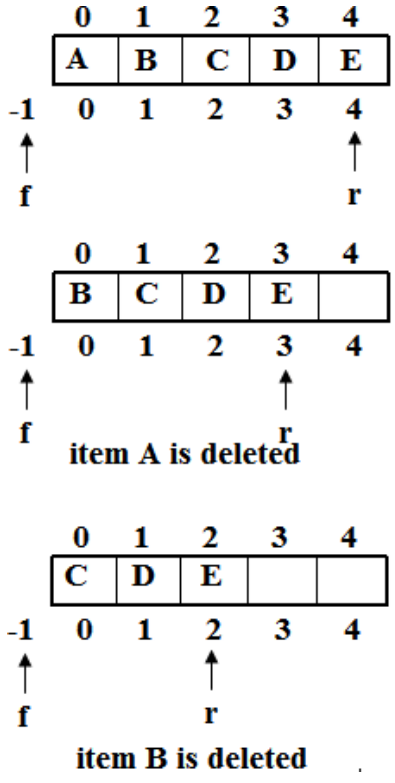


In this above situation, when we try to insert another item, which shows that the **queue is full**. This means that the **rear** index equals to MAX_QUEUE_SIZE -1. But even if the space is available at the front end, rear insertion cannot be done.

Overcome of Drawback using different methods

Method 1:

- When an item is deleted from the queue, move the entire queue to the **left** so that the first element is again at queue[0] and front is at **-1**. It should also recalculate **rear** so that it is correctly positioned.
- Shifting an array is very time-consuming when there are many elements in queue & queueFull has worst case complexity of **O(MAX_QUEUE_SIZE)**



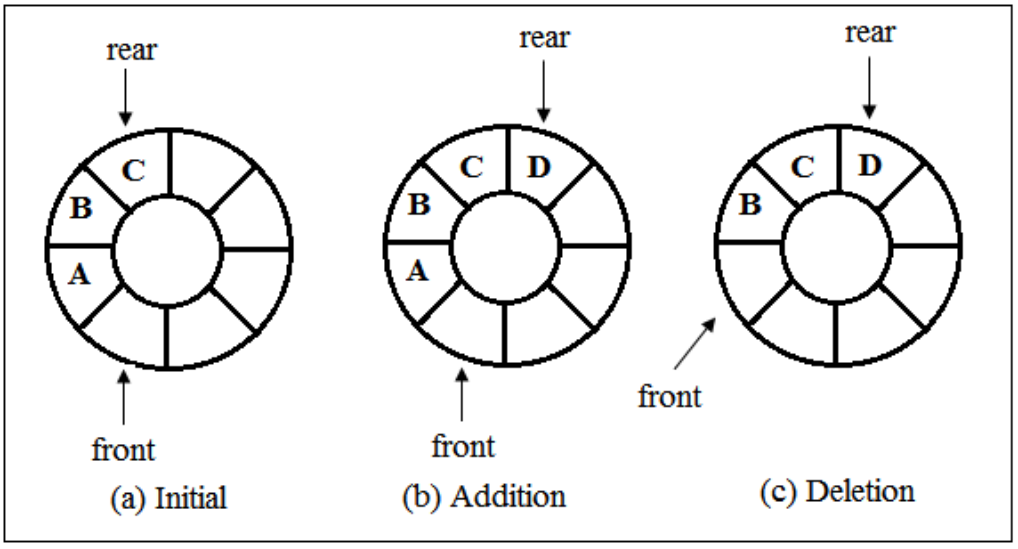
Method 2:

Circular Queue

- It is "The queue which **wrap around** the end of the array." The array positions are arranged in a circle.
- In this convention the variable **front** is changed. **front** variable points one position **counterclockwise** from the location of the front element in the queue. The convention for rear is unchanged.

CIRCULAR QUEUES

- It is "The queue which **wrap around** the end of the array." The array positions are arranged in a circle as shown in figure.
- In this convention the variable **front** is changed. **front** variable points one position **counterclockwise** from the location of the front element in the queue. The convention for rear is unchanged.



Implementation of Circular Queue Operations

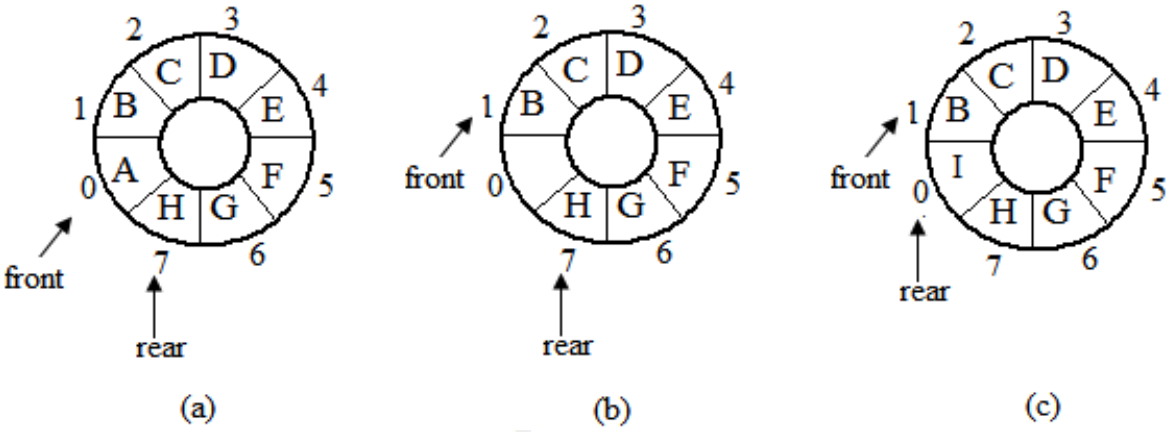
- When the array is viewed as a circle, each array position has a **next** and a **previous** position. The position next to **MAX-QUEUE-SIZE -1** is **0**, and the position that precedes **0** is **MAX-QUEUE-SIZE -1**.
- When the queue **rear** is at MAX_QUEUE_SIZE-1, the next element is inserted at position **0**.
- In circular queue, the variables **front** and **rear** are moved from their current position to the next position in clockwise direction. This may be done using code

```
if (rear = = MAX_QUEUE_SIZE-1)
        rear = 0;
else rear++;
```

Addition & Deletion

- To add an element, increment **rear** one position clockwise and insert at the new position. Here the **MAX_QUEUE_SIZE** is 8 and if all 8 elements are added into queue and that can be represented in below figure (a).
- To delete an element, increment **front** one position clockwise. The element **A** is deleted from queue and if we perform 6 deletions from the queue of Figure (b) in this fashion, then queue becomes empty and that **front =rear.**
- If the element **I** is added into the queue as in figure (c), then **rear** needs to increment by 1 and the value of rear is **8**. Since queue is circular, the next position should be 0 instead of 8.
  This can be done by using the modulus operator, which computes remainders.

(rear +1) % MAX_QUEUE_SIZE




```
void addq(element item)
        {               /* add an item to the queue */
                rear = (rear +1) % MAX_QUEUE_SIZE;
                if (front == rear)
                        queueFull();            /* print error and exit */
                queue [rear] = item;
        }
```

Program: Add to a circular queue

```
element deleteq()
        {               /* remove front element from the queue */
                element item;
                if (front == rear)
                        return queueEmpty( );           /* return an error key */
                front = (front+1)% MAX_QUEUE_SIZE;
                return queue[front];
}
```

Program: Delete from a circular queue

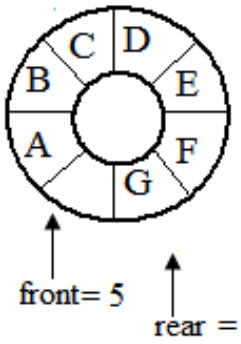
Note:

- When queue becomes empty, then **front =rear.** When the queue becomes full and **front =rear.** It is difficult to distinguish between an empty and a full queue.
- To avoid the resulting confusion, increase the capacity of a queue just before it becomes full.

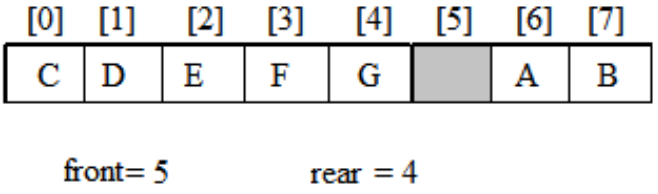
CIRCULAR QUEUES USING DYNAMIC ARRAYS

- A dynamically allocated array is used to hold the queue elements. Let **capacity** be the number of positions in the array queue.
- To add an element to a **full queue**, first increase the size of this array using a function **realloc**. As with dynamically allocated stacks, **array doubling** is used.

Consider the **full queue** of figure (a). This figure shows a queue with seven elements in an array whose capacity is 8. A circular queue is flatten out the array as in Figure (b).

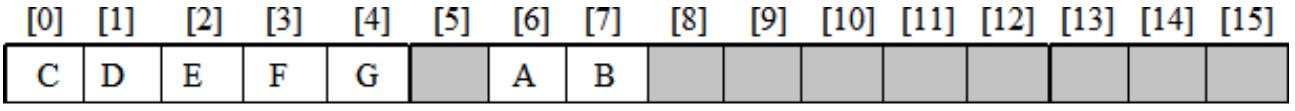


(a) A full circular queue



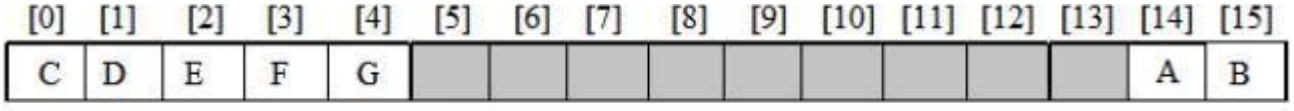
(b) Flattened view of circular full queue

Figure (c) shows the array after array doubling by relloc



(c) After array doubling

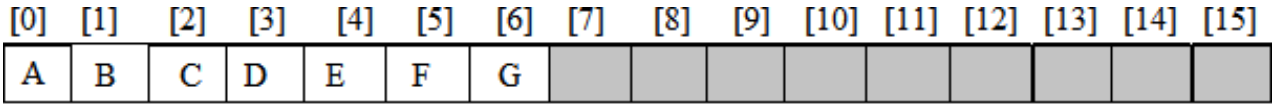
To get a proper circular queue configuration, slide the elements in the right segment (i.e., elements A and B) to the right end of the array as in figure (d)



(d After shifting right segment

To obtain the configuration as shown in figure (e), follow the steps

1) Create a new array **newQueue** of twice the capacity.
2) Copy the second segment (i.e., the elements queue [front +1] through queue [capacity-1]) to positions in **newQueue** beginning at 0.
3) Copy the first segment (i.e., the elements queue [0] through queue [rear]) to positions in newQueue beginning at **capacity – front – 1.**



front= 15 rear = 6

(e) Alternative configuration

Below program gives the code to add to a circular queue using a dynamically allocated array.

```
void addq( element item)
    {           /* add an item to the queue
        rear = (rear +1) % capacity;
        if(front == rear)
            queueFull( );          /* double capacity */
        queue[rear] = item;
    }
```

Below program obtains the configuration of **figure (e)** and gives the code for queueFull. The function copy (a,b,c) copies elements from locations **a** through **b-1** to locations beginning at **c**.

```
void queueFull( )
    {                          /* allocate an array with twice the capacity */
        element *newQueue;
        MALLOC ( newQueue, 2 * capacity * sizeof(* queue));
        /* copy from queue to newQueue */

        int start = ( front + ) % capacity;
        if ( start < 2)                /* no wrap around */
            copy( queue+start, queue+start+capacity-1,newQueue);
        else
        {                          /* queue wrap around */
            copy(queue, queue+capacity, newQueue);
            copy(queue, queue+rear+1, newQueue+capacity-start);
        }
```

```
        /* switch to newQueue*/
        front = 2*capacity – 1;
        rear = capacity – 2;
        capacity * =2;
        free(queue);
        queue= newQueue;
    }
```

Program: queueFull

DEQUEUES OR DEQUE

A deque (double ended queue) is a linear list in which elements can be added or removed at either end but not in the middle.

Representation

- Deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque.
- Figure shows deque with 4 elements maintained in an array with N = 8 memory locations.
- The condition LEFT = NULL will be used to indicate that a deque is empty.

DEQUE

			AAA	BBB	CCC	DDD	
1	2	3	4	5	6	7	8

LEFT: 4 RIGHT: 7

There are two variations of a deque

1. **Input-restricted deque** is a deque which allows insertions at only one end of the list but allows deletions at both ends of the list
2. **Output-restricted deque** is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

PRIORITY QUEUES

A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

(1) An element of higher priority is processed before any element of lower priority.
(2) Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

Representation of a Priority Queue

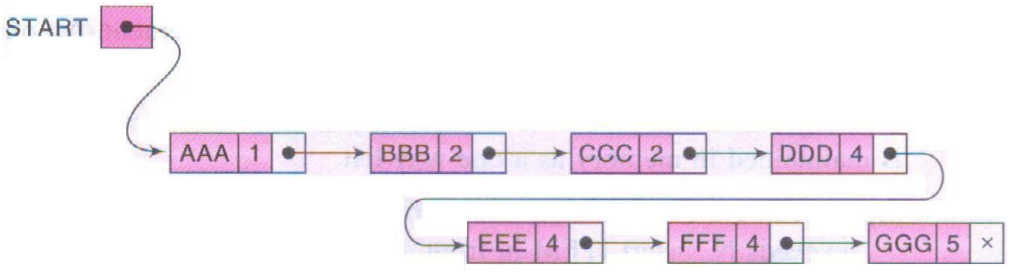
1. One-Way List Representation of a Priority Queue

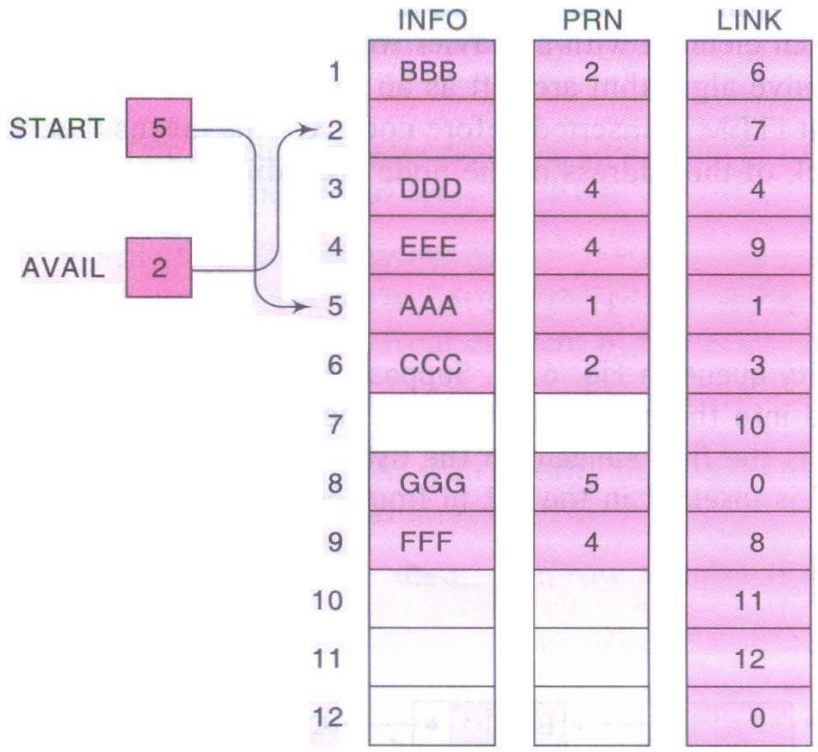
One way to maintain a priority queue in memory is by means of a one-way list, as follows:

1. Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.
2. A node X precedes a node Y in the list
   a. When X has higher priority than Y
   b. When both have the same priority but X was added to the list before Y. This means that the order in the one-way list corresponds to the order of the priority queue.

Example:

- Below Figure shows the way the priority queue may appear in memory using linear arrays INFO, PRN and LINK with 7 elements.
- The diagram does not tell us whether BBB was added to the list before or after DDD. On the other hand, the diagram does tell us that BBB was inserted before CCC, because BBB and CCC have the same priority number and BBB appears before CCC in the list.





The main property of the one-way list representation of a priority queue is that the element in the queue that should be processed first always appears at the beginning of the one-way list. Accordingly, it is a very simple matter to delete and process an element from our priority queue.

Algorithm to deletes and processes the first element in a priority queue

Algorithm: This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list.

1. Set ITEM:= INFO[START] [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

Algorithm to add an element to priority queue

Adding an element to priority queue is much more complicated than deleting an element from the queue, because we need to find the correct place to insert the element.

Algorithm: This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

1. Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.
2. If no such node is found, insert ITEM as the last element of the list.

The main difficulty in the algorithm comes from the fact that ITEM is inserted before node X. This means that, while traversing the list, one must also keep track of the address of the node preceding the node being accessed.

Example:

Consider the priority queue in Fig (a). Suppose an item XXX with priority number 2 is to be inserted into the queue. We traverse the list, comparing priority numbers.

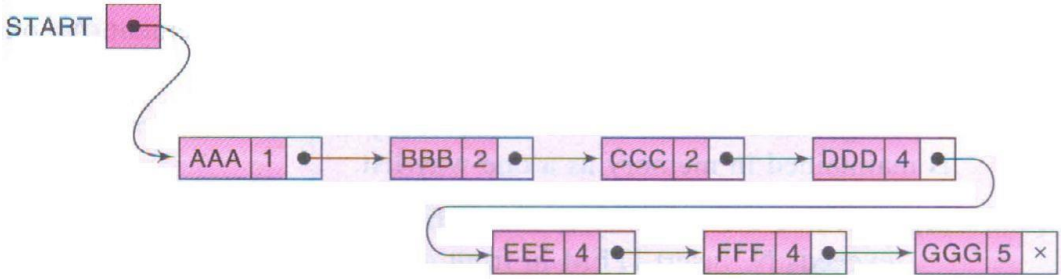
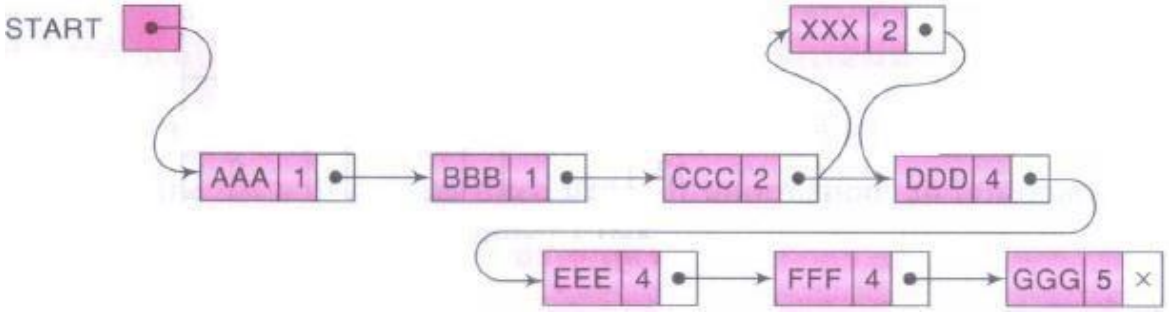


Fig (a)



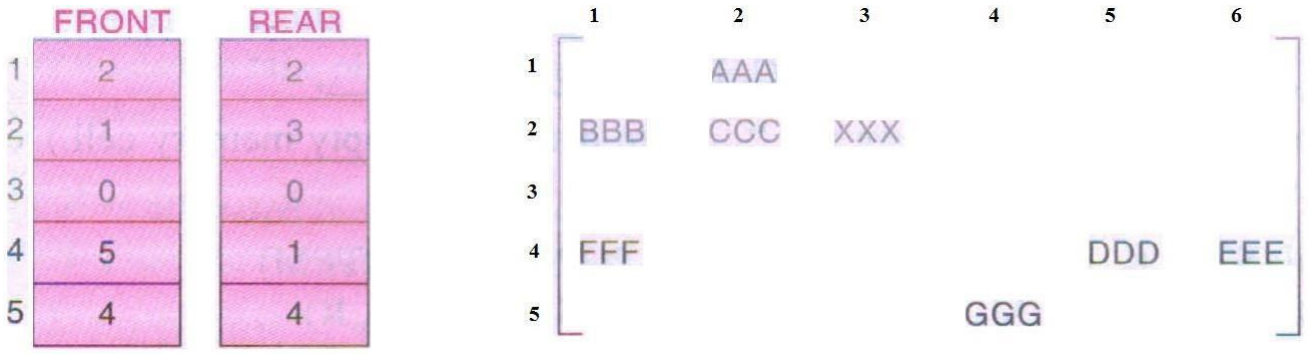
Fig(b)

Observe that DDD is the first element in the list whose priority number exceeds that of XXX. Hence XXX is inserted in the list in front of DDD, as pictured in Fig(b).

Observe that XXX comes after BBB and CCC, which have the same priority as XXX. Suppose now that an element is to be deleted from the queue. It will be AAA, the first element in the List. Assuming no other insertions, the next element to be deleted will be BBB, then CCC, then XXX, and so on.

Array Representation of a Priority Queue

- Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number).
- Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REA R.
- If each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays.



Observe that FRONT[K] and REAR[K] contain, respectively, the front and rear elements of row K of QUEUE, the row that maintains the queue of elements with priority number K.

The following are outlines or algorithms for deleting and inserting elements in a priority queue

Algorithm: This algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

1. [Find the first non-empty queue.]
   Find the smallest K such that FRONT[K] ≠ NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.

Algorithm: This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.

MULTIPLE STACKS AND QUEUES

- In multiple stacks, we examine only **sequential mappings** of stacks into an array. The array is one dimensional which is **memory[MEMORY_SIZE]**. Assume ***n*** stacks are needed, and then divide the available memory into *n* segments. The array is divided in proportion if the expected sizes of the various stacks are known. Otherwise, divide the memory into equal segments.
- Assume that *i* refers to the stack number of one of the *n* stacks. To establish this stack, create indices for both the **bottom** and **top** positions of this stack. ***boundary*[i]** points to the position immediately to the left of the bottom element of stack *i*, ***top[i]*** points to the top element. Stack *i* is empty iff *boundary[i]=top[i]*.

The declarations are:

```
#define MEMORY_SIZE  100          /* size of memory */
#define MAX_STACKS  10            /* max number of stacks plus 1 */
element memory[MEMORY_SIZE];      /* global memory declaration */
int top [MAX_STACKS];
int boundary [MAX_STACKS] ;
int n;                            /*number of stacks entered by the user */
```

To divide the array into roughly equal segments

```
top[0] = boundary[0] = -1;
        for (j= 1;j<n; j++)
                top[j] = boundary[j] = (MEMORY_SIZE / n) * j;
boundary[n] = MEMORY_SIZE - 1;
```

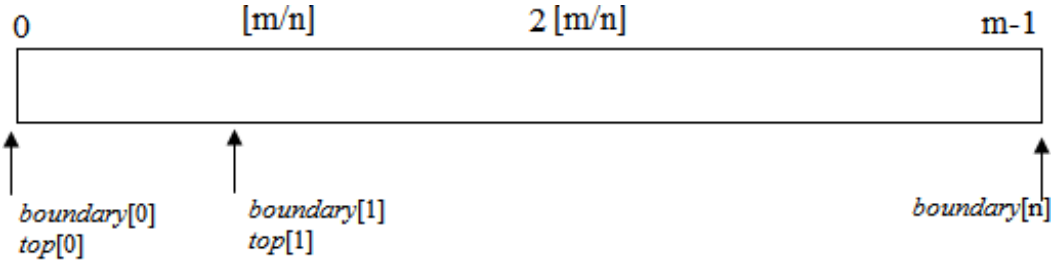


Figure: Initial configuration for *n* stacks in *memory [m]*.

In the figure, **n** is the number of stacks entered by the user, n < MAX_STACKS, and m =MEMORY_SIZE. Stack **i** grow from **boundary[i] + 1** to **boundary [i + 1]** before it is full. A boundary for the last stack is needed, so set **boundary [n]** to **MEMORY_SIZE-1**.

Implementation of the add operation

```
void push(int i, element item)
    {           /* add an item to the ith stack */
        if (top[i] == boundary[i+l])
            stackFull(i);
        memory[++top[i]] = item;
    }
```

Program: Add an item to the ith stack

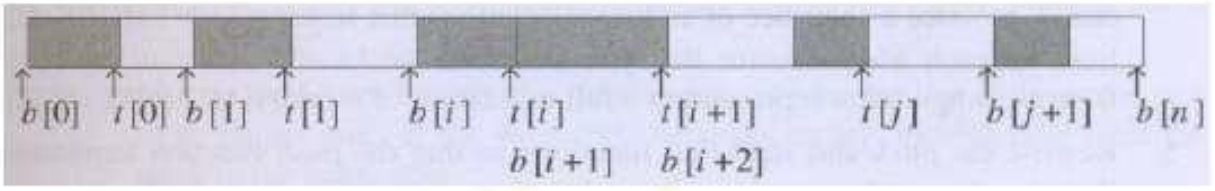
Implementation of the delete operation

```
element pop(int i)
    {           /* remove top element from the ith stack */
        if (top[i] == boundary[i])
            return stackEmpty(i);
        return memory[top[i]--];
    }
```

Program: Delete an item from the ith stack

The top[i] == boundary[i+1] condition in push implies only that a particular stack ran out of memory, not that the entire memory is full. But still there may be a lot of unused space between other stacks in array memory as shown in Figure.

Therefore, create an error recovery function called **stackFull** , which determines if there is any free space in memory. If there is space available, it should shift the stacks so that space is allocated to the full stack.



b boundary , t=top

Method to design stackFull

- Determine the least, *j,* i < *j* < *n,* such that there is free space between stacks *j* and j+1. That is, *top[j] < boundary[j+l]*. If there is a j, then move stacks *i+l,i+2,* .., j one position to the right (treating *memory[O]* as leftmost and *memory[MEMORY_SIZE* - 1] as rightmost). This creates a space between stacks *i* and *i*+1.
- If there is no j as in (1), then look to the left of stack *i.* Find the largest j such that 0 ≤ j ≤ *i* and there is space between stacks j and j+ 1 ie, *top[j] < boundary[j+l]*. If there is a j, then move stacks *j+l, j+2, ... , i* one space to the left. This also creates space between stacks *i* and *i*+1.
- If there is no j satisfying either condition (1) or condition (2), then all *MEMORY_SIZE* spaces of memory are utilized and there is no free space. In this case *stackFull* terminates with an error message.

LINKED LISTS

TOPICS

MODULE – 2: **Linked Lists:** Definition, Representation of linked lists in Memory, Memory allocation; Garbage Collection.
Singly Linked list, SLL operations: Traversing, Searching, Insertion, and Deletion, Lists and Chains, Representing Chains in C, Circular linked lists, and header linked lists, Linked Stacks and Queues, Polynomials.
MODULE – 3: Additional List operations, Doubly Linked lists, Sparse matrix representation.

LINKED LISTS *In this chapter, the list data structure is presented. This structure can be used as the basis for the implementation of other data structures (stacks, queues etc.). The basic linked list can be used without modification in many programs. However, some applications require enhancements to the linked list design. These enhancements fall into three broad categories and yield variations on linked lists that can be used*

The disadvantages of arrays are:

- The size of the array is fixed. Most often this size is specified at compile time. This makes the programmers to allocate arrays, which seems "large enough" than required.
- Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.
- Generally array's allocates the memory for all its elements in one block whereas linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Here is a quick review of the terminology and rules of pointers. The linked list code will depend on the following functions:

- **malloc()** is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype of malloc() and other heap functions are in stdlib.h. malloc() returns NULL if it cannot fulfill the request. It is defined by: *void *malloc (number_of_bytes)* Since a void * is returned the C standard states that this pointer can be converted to mple, any type.

- For ex: char *cp; cp = (char *) malloc (100); Attempts to get 100 bytes and assigns the starting address to cp. We can also use the sizeof() function to specify the number of bytes. For example, int *ip; ip = (int *) malloc (100*sizeof(int));

- **free()** is the opposite of malloc(), which de-allocates memory. The argument to free() is a pointer to a block of memory in the heap — a pointer which was obtained by a malloc() function. The syntax is: *free (ptr);* The advantage of free() is simply memory management when we no longer need a block.

1.1. LINKED LIST CONCEPTS

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list. The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Figure 1: Representation of node

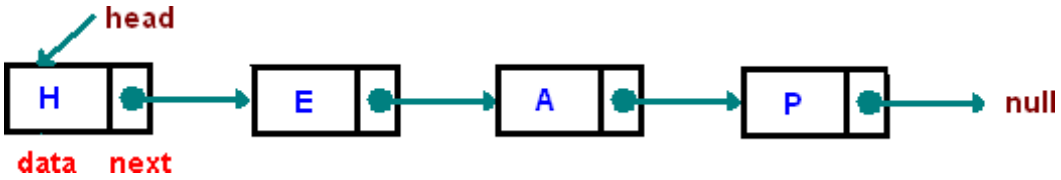


Figure 2: Example of Linked list

Advantages of linked lists: Linked lists have many advantages. Some of the very important advantages are:

- Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
- Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
- Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
- Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

- It consumes more space because every node requires a additional pointer to store address of the next node.
- Searching a particular element in list is difficult and also time consuming.

Comparison between array and linked list:

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

1.2. TYPES OF LINKED LISTS

Basically we can put linked lists into the following four items:

1. Single Linked List.
2. Double Linked List.
3. Circular Linked List.
4. Circular Double Linked List.

- A single linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.
- A double linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.
- A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.
- A circular double linked list is one, which has both the successor pointer and predecessor pointer in the circular manner.

1.3. SINGLE LINKED LIST:

Singly linked lists are the basic type of linked lists where each node has exactly one pointer field.

A singly linked list is comprised of zero/ more number of nodes when the number of nodes is zero , the list is empty otherwise if the linked list is non-empty, the list is pictorially represented as 1st node links to 2nd node and 2nd node links to 3rd node and so on. the last node has zero link whose value of address is set to NULL.

REPRESENTING SLL IN C LANGUAGE / MEMORY ALLOCATION

The following features are used to represent SLL. Use the following 3 steps to create a SLL.

1. **Define node's structure**

   To define a node, self-referential structures are used
2. **Create a new node**

   malloc( ) or MALLOC macro is used to allocate memory for the defined structures nodes of the size needed for structure node considered.
3. **Removal of nodes**

   At any point if the allocated nodes are not in use, they are removed by free( ).

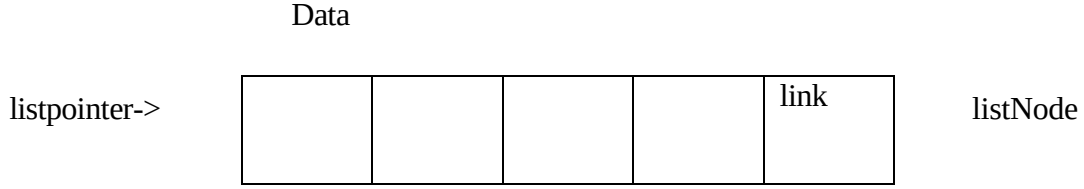
Example: To create a linked list of words following the above steps follow the steps given below.

1. Defining a node

Using self-referential structures nodes are created. for a list of words , in every node the data field should store words, so define datatype accordingly.

```
typedef struct listnode *listpointer;
typedef struct
{
    char data[4];
    listpointer link;
}listNode;
```

This definition will result into a node by name listNode containing char data field of size 4 and a field by name link , which is a pointer variable of type listpointer , where listpointer is a pointer to whole structure.



- Here, first is a variable of type pointer i.e. listpointer, initially making it as NULL and hence , a new list is created by name first and it is empty,

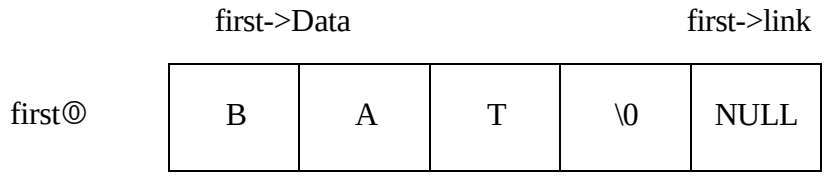
- To create a new node in list, malloc() function is used.

first = malloc(sizeof(*first));

3. To assign the value to the fields of the node. Here, the operator ⓪ is used, which is referred as the structure member operator.

strcpy(first->data,"BAT");
first->link=NULL;

These statements are represented as below:



Here, B = first-> data [0];
A = first-> data [1];
T = first-> data [2];
\0 = first-> data [3];
NULL=first ⓪ link;

GARBAGE COLLECTION

- When the memory is allocated to the linked lists, a special list is maintained which consists of unused memory cells. This list, which has its own pointer, is called the list of available space/ the free storage list or the free pool.
- Thus, the memory is allocated from free pool.
- When node is deleted from a list or a entire list is deleted from a program, the memory space has to be inserted into free storage list, so that it will be reusable.
- The operating system of a computer may periodically collect all the deleted space onto the free-storage list. Any technique which does this collection is called **garbage collection.**
- **Garbage collection** usually takes place in 2 steps:

    1. The computer runs through all list, tagging those cells which are currently in use and then the computer runs through the memory, collecting all untagged space onto the free – storage list.
    2. the garbage collection may take place when there is only some minimum amount of space or no space at all left in the free-storage list or when the CPU is idle and has time to do the collection

The garbage collection is invisible to the programmer.

LINKED LIST OPERATIONS

The following operations are performed on the linked list

1. INSERTION

Insertion operation is used to insert new node to the list created. This operation is performed depending on many scenarios of linked lists like

- If the linked list is empty, then new node after insertion becomes the first node.
- If the list already contains nodes the new node is attached either at front end of the list or at the last end.
- If the insertion is based on the data element/position, then search the list to find the location and then insert the new node.

NOTE: same conditions are checked for deleting a node from the list as well.

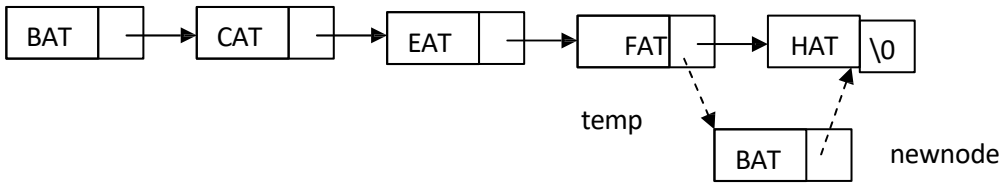


Figure 4: Insertion Operation

Here, if we need **to insert the data item GAT between FAT and HAT, the following steps are** followed.

- Get a node temp
- Set the data field to GAT
- Set the link field of temp to point to the node after FAT, which contains HAT
- Set FAT link field to temp

In figure, if we need to delete FAT, then find the element that immediately precedes the element to be deleted.

EX :- Here, identify EAT.

- Set that element link to the position of GAT i.e. EAT link should point to GAT.
- Use free() to delete FAT node.

Function to create a two-node SLL

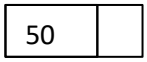
```
listpointer create2()
{
    listpointer first,second;
    first = malloc(sizeof(*first));
    second = malloc(sizeof(*second));
    first⓪data=10;
    second⓪data=20;
    first⓪link=second;
    second⓪link=NULL;
    return first;
```

```
}
```

C function to insert new node with data value 50 into the SLL by name first after the node X.

```
void insert(listpointer *first, listpointer X)
{
    listpointer temp;
    temp=malloc(sizeof(*temp));
    temp->data = 50;
    if(*first)
    {
        temp-> link = x->link;
        x->link = temp;
    }
    else
    {
        temp-> link = NULL;
        *first=temp;
    }
}
```

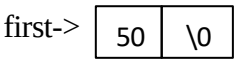
from main function, call this function as insert(*first , x);



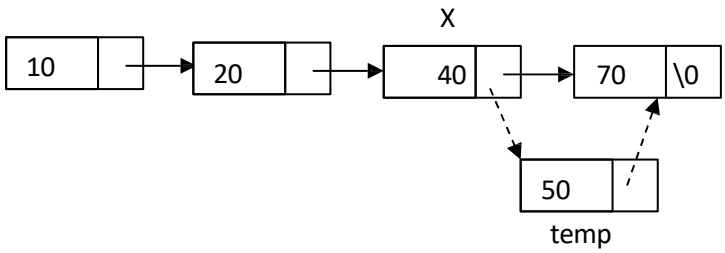
Node to be inserted

temp

- If it is empty, then will be the first node in SLL first.



- So, here we are passing the address of first, the second argument is the X.



Insert at front of the SLL, insert at end of SLL and Insert at mid of the SLL

Refer PPT

2. LIST DELETION

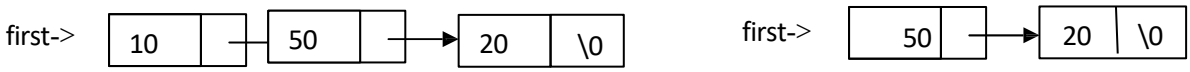
Example deletes X from the list first , where trail is the preceding node of X.

```
void delete( listpointer *first, listpointer trail, listpointer X)
{
    if(trail)
        trail->link = x->link;
    else
        *first = (*first) ->link;
    free(x);
}
```

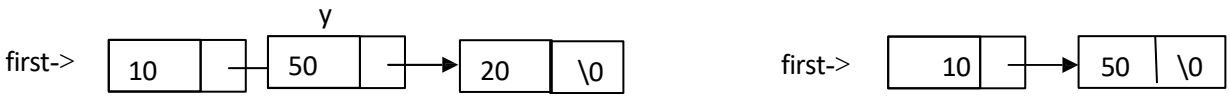
From main function call this function as below:-

delete (&first, NULL, first);
OR
delete (&first, y, y->link);

- Any node is deleted from a linked list by another function delete.
- Assuming that we have 3 pointers, first which points to the start of the list, x points to the node that we wish to delete, &trail points to the node that precedes x node



- In this example, the node x , which has to be deleted is the first node itself. so, after deleting that node, resultant linked list should be like , next figure. so, we must change the value of first to point to node with data 50.



- In above example, deletion corresponds to the function call, **delete (&first, y, y->link);** where, y is the trail node for the deleting node x, which is y->link, i.e, the next node after y. so deleting node containing data 20 is performed and y->link set to NULL.

Deletion at front of SLL, Deletion at End of SLL, Deletion at Mid of List

Refer PPT

3. TRAVESING/PRINTING THE LIST

To print the data fields of the nodes in a list. First print the contents of first's data field. Then, replace first with the address in its link field. So, continue printing out the data field and moving to the next node until end of the list is reached.

```
void printlist (listpointer first)
{
    printf("\n The list contains");
    for( ; first ; first->link)
        printf("%4d", first->data);
}
```

3. SEARCHING

- Searching operation performs the process of finding the node containing the desired value in linked list.
- Searching starts from the first node of the linked list, so that the complete linked list can be searched to find the element. if found search is successful, else unsuccessful.

```
void search(listpointer first, int key)
        {
                int found = 0;
                while( first!=NULL && found = = 0)
                {
                        if(first->data!=key)
                                first=first->next;
                        else
                                found = = 1;
                }
    }
```

DOUBLY LINKED LISTS

- Doubly linked list contains the node which contains the following fields: data field and two link fields, one linking in forward direction and other linking in backward direction.
- figure below shows the DLL

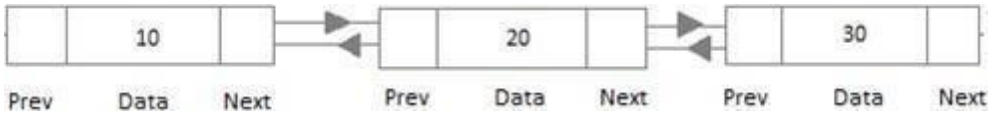


Figure 5: DLL

- In singular linked list, it is possible to traverse in only one direction (forward) in the linked list.
- If we are pointing to a specific node say p, then we can move only in the direction of the links.
- To find a node before p, i.e. preceding node p is difficult unless we start from beginning to reach its previous node.
- Same problem exists, when delete or insertion operations are done on any arbitrarily node in SLL.
- These problems can be overcome using DLL, as they have both direction links, from any node p, where we can find next node/ preceding node easily.

C Representation of DLL

A node in a doubly linked list has at least three fields, a left link field (llink), a data field(data), and a right link field(rlink).

```
typedef struct node *nodepointer;
typedef struct
{
    nodepointer llink;
    element data;
```

```
    nodepointer rlink;
  }node;
```

if ptr points to a node in a DLL, then ptr=ptr->llink->rlink= ptr->rlink->llink;

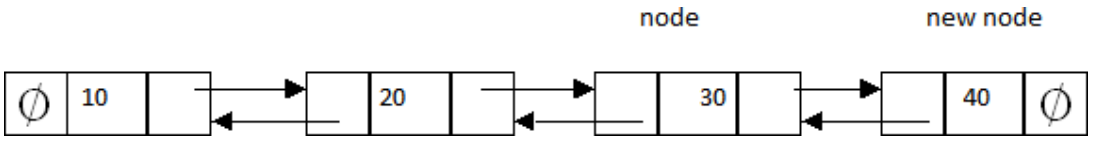
Operations performed on DLL

1. Insert

Insertion into a doubly linked list is fairly easy. Assume there are two nodes, node and newnode, node may be either a header node or an interior node in a list. The function dinsert performs the insertion operation in constant time.

It inserts new node to the right of the node.

```
void dinsert (nodepointer node,nodepointer newnode)
{
        newnode->llink=node;
        node->rlink=newnode;
}
```



2. Deletion

Deletion from a doubly linked list is equally easy. The function *ddelete* deletes the node deleted from the list pointed to by node.

To accomplish this deletion, we only need to change the link fields of the nodes that precede (deleted→llink→rlink) and follow (deleted→rlink→llink) the node we want to delete.

It deleted the node from the list pointed to by node.

```
void delete(nodepointer node,nodepointer deleted)
{
    if(node == deleted)
        printf("\n deletion of header node is not permitted");
    else
    {
        deleted->llink->rlink=deleted->rlink;
        deleted->rlink->llink=deleted->llink;
        free(deleted);
    }
}
```

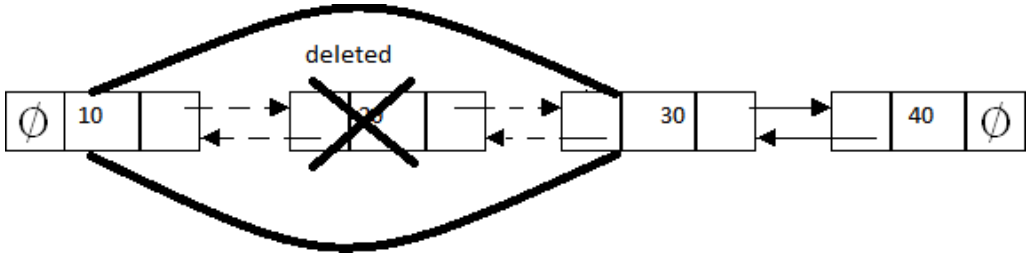


Figure 7: Deletion in DLL

CIRCULAR LINKED LISTS

A linked list whose last node points back to the first node instead of containing a null pointer is called **circular list.**

1. Circular singly linked list

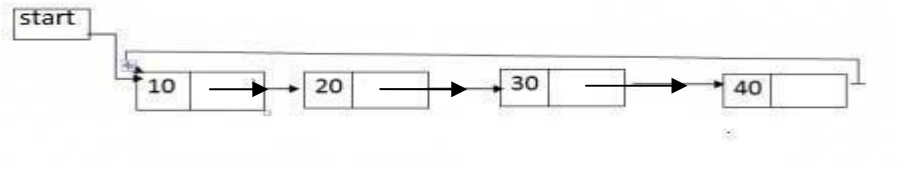


Figure 8: CSLL

In a singly linked circular list, the pointer field of the last node stores the address of the starting node In the list. Hence it is easy to traverse the list given the address of any node in the list.

2. Circular doubly linked list

A doubly linked list whose last node rlink points to first node and first node llink points to last node, making it is a circular called as circular DLL.

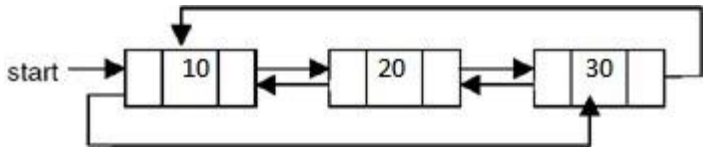


Figure 9: CDLL

To insert a new node into circular DLL at the end

```
void dinsert(nodepointer node,nodepointer newnode)
{
    newnode->llink=node;
    newnode->rlink=node->rlink;
    node->rlink->llink=newnode;
    node->rlink=newnode;
}
```

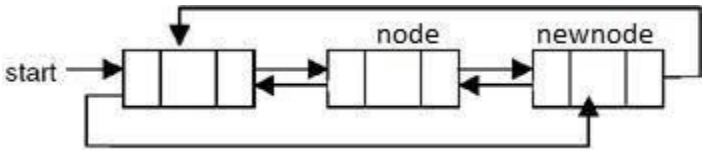


Figure 10: Insertion in CDLL

Advantages of CLL

- Linked list made as circular can connect to the first node easily.
- Insertion/deletion operations can be performed quickly.
- Accessing previous node of any node X, can be achieved from X©end of the list and end to that particular node.
- Circular linked list even can be adapted for DLL, which are doubly linked CLL.

HEADER LINKED LIST

A header linked list is a linked list which always contains a special node called the ***header node*** at the beginning of the list. It is an extra node kept at the front of a list. **Such a node does not represent an item in the list**. The information portion might be unused.

This header node allows us to perform operations more easily and also differentiaties the nodes,first/last especially when the list is circular.The header node may contain some useful about linked list such as number of nodes in the list , address of last node/ some specific distinguishing information . the address of starting node is refereed by headerpointer.

There are two types of header list

1. **Grounded header list**: is a header list where the last node contains the null pointer.
2. **Circular header list**: is a header list where the last node points back to the header node.

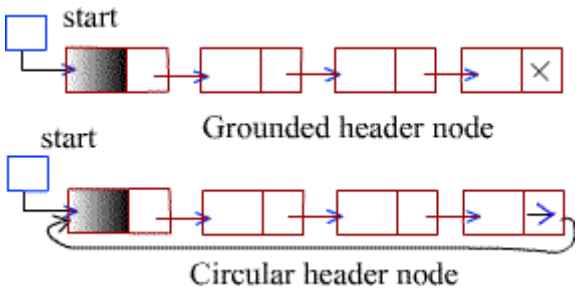


Figure 11: Grounded and Circular header Linked List

Observe that the list pointer START always points to the header node.

- If START→LINK = NULL indicates that a grounded header list is empty
- If START→LINK = START indicates that a circular header list is empty.

The first node in a header list is the node following the header node, and the location of the first node is START→LINK, not START, as with ordinary linked lists.

Below algorithm, which uses a pointer variable PTR to traverse a circular header list

1. Begins with PTR = START→LINK (not PTR = START)
2. Ends when PTR = START (not PTR = NULL).

The two properties of circular header lists:

1. The null pointer is not used, and hence all pointers contain valid addresses.
2. Every (ordinary) node has a predecessor, so the first node may not require a special case.

Linked Stacks and Queues:-

- To represent several queues and stacks sequentially, linked list is the efficient way.
- The linked stack and linked queue are pictorially shown below:

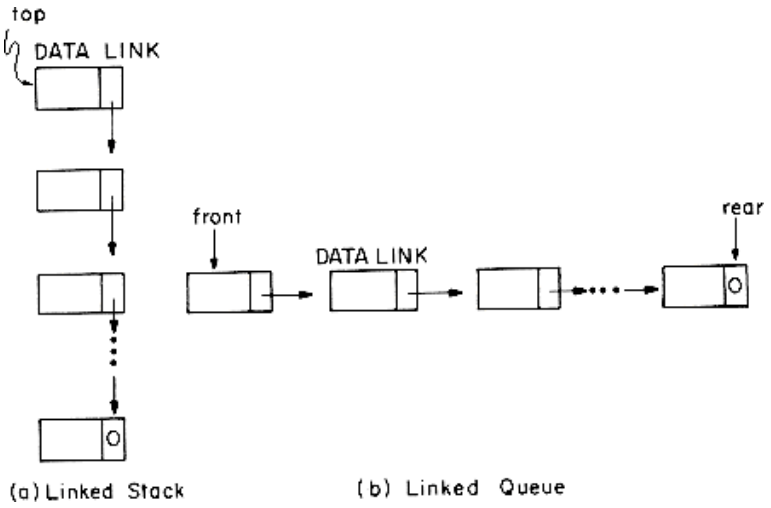


Figure 12: (a) Linked Stack and (b) Linked queue

- The directions of arrows in both stack queue representation help us to easily understand the operations i.e insertion and deletion of nodes. i.e in stack, push/pop operation performed from ht etop of the stack.
- In Figure (b) above, in linked queue, node is easily inserted and deleted using rear and front respectively.
- C declarations to represent 'n', number of stacks in memory simultaneously, where n<= MAX_STACKS.

```
typedef struct
{
        int key;
}Element;
typedef struct stack * stackPointer;
typedef struct
{
        Element data;
        stackPointer link;
}stack;
stackPointer top[MAX_STACKS];
```

- The initial condition for the stack is top[i]=NULL, 0<i<=MAX_STACKS.
- Boundary condition is top[i]=NULL if the ith stack is empty.

Operations on Mulitple stacks (Linked stack):-

1) PUSH:-

- The push function creates a new node by name temp & places item in the data field & top in the link field. The variable top is then changed to point to temp.

```
void push(int i, Element item)
{
        stackPointer temp;
        temp = malloc(sizeof(*temp));
        temp->data = item;
        temp->link = tep[i];
        top[i] = temp;
}
```

- The above C function is to add item to the ith stack.

2) POP:-

```
Element pop(int i)
{
        stackPointer temp = top[i];
        Element item;
        if(!temp)
                return stackEmpty();
        item = temp->data;
        top[i] = temp->link;
        free(temp);
        return item;
}
```

- The above C function is used to delete top element from ith stack.

LINKED QUEUES:-

- To represent 'm' queues simultaneously, declarations are as follows:- where m<=MAX_QUEUES.

```
#define MAX_QUEUES 10
typedef struct queue *queuePointer;
typedef struct
{
        Element data;
        queuePointer link;
}queue;
queuePointer front[MAX_QUEUES], rear[MAX_QUEUES];
```

- In initial condition for the queue, front[i] = NULL, 0<=i<MAX_QUEUES and the boundary is front[i] = NULL iff the ith queue is empty.

Operations of Linked queue:-

1) **Insert**:- add an item to the rear end of a linked queue.

```
void addq(i, item)
{
        queuePointer temp;
        temp = malloc(sizeof(*temp));
        temp->data = item;
        temp->link = NULL;
        if(front[i])
                rear[i]->link = temp;
        else
                front[i] = temp;
        rear[i] = temp;
}
```

Function addq is more complex than push because we must check for an empty queue. If the queue is empty, then change front to point to the new node; otherwise change rear's link field to point to the new node. In either case, we then change rear to point to the new node.

2) **Delete**:- Deletes an item from the front of a linked queue.

```
Element deleteq( int i)
{
    queuePointer temp = front[i];
    Element item;
    if(!temp)
            return queueEmpty();
    item = temp->dtaa;
    front[i] = temp->link;
    free(temp);
    return item;
 }
```

Function deleteq is similar to pop since nodes are removing that is currently at the start of the list. Typical function calls would be addq (i, item); and item = deleteq (i);

APPLICATIONS OF LINKED LISTS

1) **Polynomial Addition:-**

- For adding 2 polynomials, the following terms are compared and checked starting at the nodes pointed to by a & b.
    - If the exponents are equal – add 2 coefficients and create new term for the result. Move a & b to point to next nodes.
    - If the exponent of the term in a is less than the exponent of current item in b, then,
        - Create a duplicate term b.
        - Attach this term to the result called c.
        - Advance the pointer to the next term only in b.
    - If the exponent of the term in a is greater then the exponent of curret item in b, then,
        - Create a duplicate term a.
        - Attach this term to the result, called c.
        - Advance the pointer to next term only in a.

Polynomial is represented as:-

$$A(x) = a_{m-1}\, x^{cm-1} + \ldots + a_0 x^{c0}$$

where, ai are non zero co- efficients and the ci are non negative integer exponents such that $c_{m-1} > c_{m-2} > \ldots > c_1 > c_0 >= 0$.

C Declaration:-

```
typedef struct polyNode *polyPointer;
typedef struct
{
        int coef;
        int expon;
        polyPointer link;
}polyNode;
polyPointer a, b;
```

polyNodes looks as:-

$a = 3x^{14}+2x^{8}+1$, $b = 8x^{14}-3x^{10}+10x^{6}$

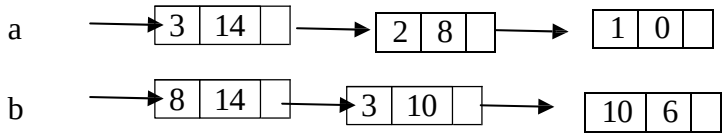


Figure 13: Representaion of a & b polynomials.

C Function for polynomial addition is given below:-

```
polyPointer padd(polyPointer a, polyPointer b)
{       /* return a polynomial which is the sum of a & b */
        polyPointer c, rear, temp;
        int sum;
        MALLOC(rear, sizeof(*rear));
        c=rear;
        while(a && b)
                switch(COMPARE (a->expon, b->expon))
                {
                        case -1:/* a->expon < b->expon */
                                attach(b->coef, b->expon, & rear);
                                b=b->link;
                                break;
                        case 0: /*a->expon == b->expon */
                                sum = a->coef + b->coef;
                                if(sum)
                                        attach(sum, a->expon, & rear);
                                a=a->link;
                                b=b->kink;
                                break;
                        case 1: /* a -> expon > b->expon */
                                attach(a->coef, a->expon, & rear);
                                a=a->link;
                }
                /* copy rest of the list a and then list b */
                for(;a;a->link)
                        attach(a->coef, a->expon, & rear);
                for(;b;b->link)
                        attach(b->coef, b->expon, & rear);
                rear->link = NULL;
                /* delete extra initial node */
                temp = c;
                c=c->link;
                free(temp);
                return c;
}
```

- The above function uses streaming process, that moves along the 2 polynomials, either copying terms directly / adding them to the result.
- Thus, while loop has 3 cases, depending on whether next pair of elements are =, < or >.
- To create a new node and append it to the end of c, the above addition function uses attach().

```
void attach( float coefficient, int exponent, polyPointer *ptr)
{       /* create a new node with coef = coefficient & expon = exponent, attach is to the node pointed to
by ptr. ptr is updated to point to this new nodes */
        polyPointer temp;
        MALLOC(temp, sizeof(*temp));
        temp->coef = coefficient;
        temp->expon = exponent;
        (*ptr)->link = temp;
        *ptr = temp;
}
```

Erasing Polynomials:-

- While using polynomials for different computations, temporary nodes which are having actually waste data can be erased.
- Example:- For performing e(x) = a(x) * b(x) + d(x);
- Main function is as below:-

```
polyPointer a,b,d,e;
a=readPoly();
b=readPoly();
d=readPoly();
temp=pmult(a,b);
e=padd(temp,d);
printPoly(e);
```

- Here, temp is a node, which need to be erased. So, the following function is used to erasenodes.

```
void erase(polyPointer *ptr)
{
    polyPointer temp;
    while(*ptr)
    {
            temp=*ptr;
            *ptr=(*ptr)->link;
            free(temp);
    }
}
```

Circular list representation of polynomials:-

- In a linked structure, the link of the last node points to the first node in the list, it is called as circular list.
- Nodes of a polynomial can be freed efficiently if circular list representation is used.

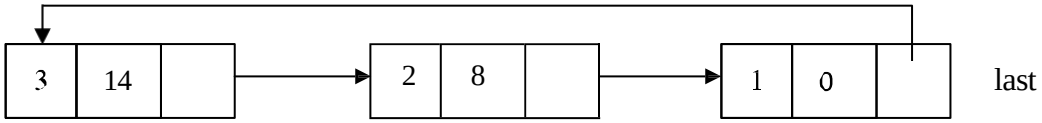


Figure 14: Circular representation of $3x^{14}+2x^{8}+1$

- Free nodes that is no longer in use can be reused by maintaining our own list of nodes that have been freed.
- When we need a new node, freed nodes list is examined, if it is not empty, then use those nodes. If not, use malloc() to create a new node.
- getNode() & retNode() functions are used to use a node and free the node as malloc() and free().

```
polyPointer getNode(void)
{
        polyPointer node;
        if(avail)
        {
                node=avail;
                avail=avail->link;
        }
        else
                MALLOC(node, sizeof(*node));
        return node;
}
void retNode(polyPointer node)
{
        /*return a node to the available list */
        node->link=avail;
        avail = node;
}
```

Erasing Circular list:-

- We can erase circular list in a fixed amount of time independent of the number of nodes in the list using cerase() function

```
void cerase(polyPointer *ptr)
{
        polyPointer temp;
        if(*ptr)
                temp=(*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
}
```

Circular lists with header nodes:-

- In order to handle the zero polynomial, each polynomial with a header node is introduced i.e. each polynomial zero / non-zero contains 1 additional node.
- The expon & coef fields of this node are irrelavant.

Figure 15: Zero Polynomial

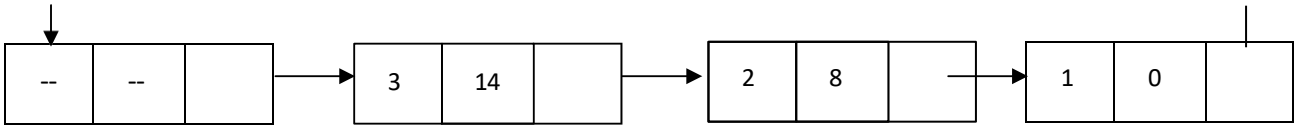


Figure 16: Polynomial $3x^{14}+2x^{8}+1$

```
polyPointer cpadd(polyPointer a, polyPointer b)
{
    polyPointer startA, c, lastC;
    int sum, done=FALSE;
    startA=a;
    a=a->link;
    b=b->link;
    c=getNode();
    c->expon = -1, lastC=c;
    do
    {
            switch(COMPARE(a->expon, b->expon))
            {
                    case -1: attach(b->coef, b->expon, &lastC);
                             b=b->link;
                            break;
                    case 0: if(startA == a)
                                    done = TRUE;
                            else
                            {
                                    sum=a->coef+b->coef;
                                    if(sum)
```

```
                                        attach(sum, a->expon, &lastC);
                                    a=a->link;
                                    b=b->link;
                                }
                                break;
                        case 1: attach(a->coef, a->expon, lastC);
                                a=a->link;
                    }
            }while(!done);
            lastC->link=c;
            return c;
    }
```

SPARSE MATRIX:-

- Sparse Matrix is a matrix with more number of zero enteries than non-zero enteries.
- Each non-zero term is represented by a node with 3 fields:- row, column and value.
- Linked list representation for sparse matrix are more efficient than array representation.
- In this represenation, each column of a sparse matrix is represented as a circularly linked list with a header node.
- Similarly, representation is used for each row of the sparse matirx.
- Each node has a tag field, which distinguish between header nodes and entry nodes. Each header node has 3 additional fields:- down, right & next.
    - down field to link into a column list.
    - right field to link into a row list.
    - next field links the header nodes together.
- Each element entry node has 3 fields in addition to the tag field:- row, col, down, right value.
    - down field to link to next non-zero term in the same column.
    - right field to link to next non-zero term in the same row.
- Thus, if $a_{ij} \neq 0$, there is a node into tag field = entry, value =a_{ij}, row=i & cal=j.

C declarations to represent sparse matrix using likked list.

```
#define MAX_SIZE 50
typedef Enum {head,entry} tagfield;
typedef struct matrixNode *matrixpointer;
typedef struct
{
   int   row;
  int     col;
  int value;
} entryNode;
typedef struct
{
     matrix pointer down;
    matrix pointer right;
    tag field tag;
    union
```

```
 {
    matrix pointer next;
    entry nodes entry;
  }u;
}matrix Node;
matrix pointer hnode[MAX_SIZE];
```

The figure shows linked representation of sparse matrix for the following sparse matrix shown below:-

0 0 0 0 6 0

0 4 0 0 0 0

4 0 0 8 0 0

0 0 0 0 0 4

0 0 7 0 0 0

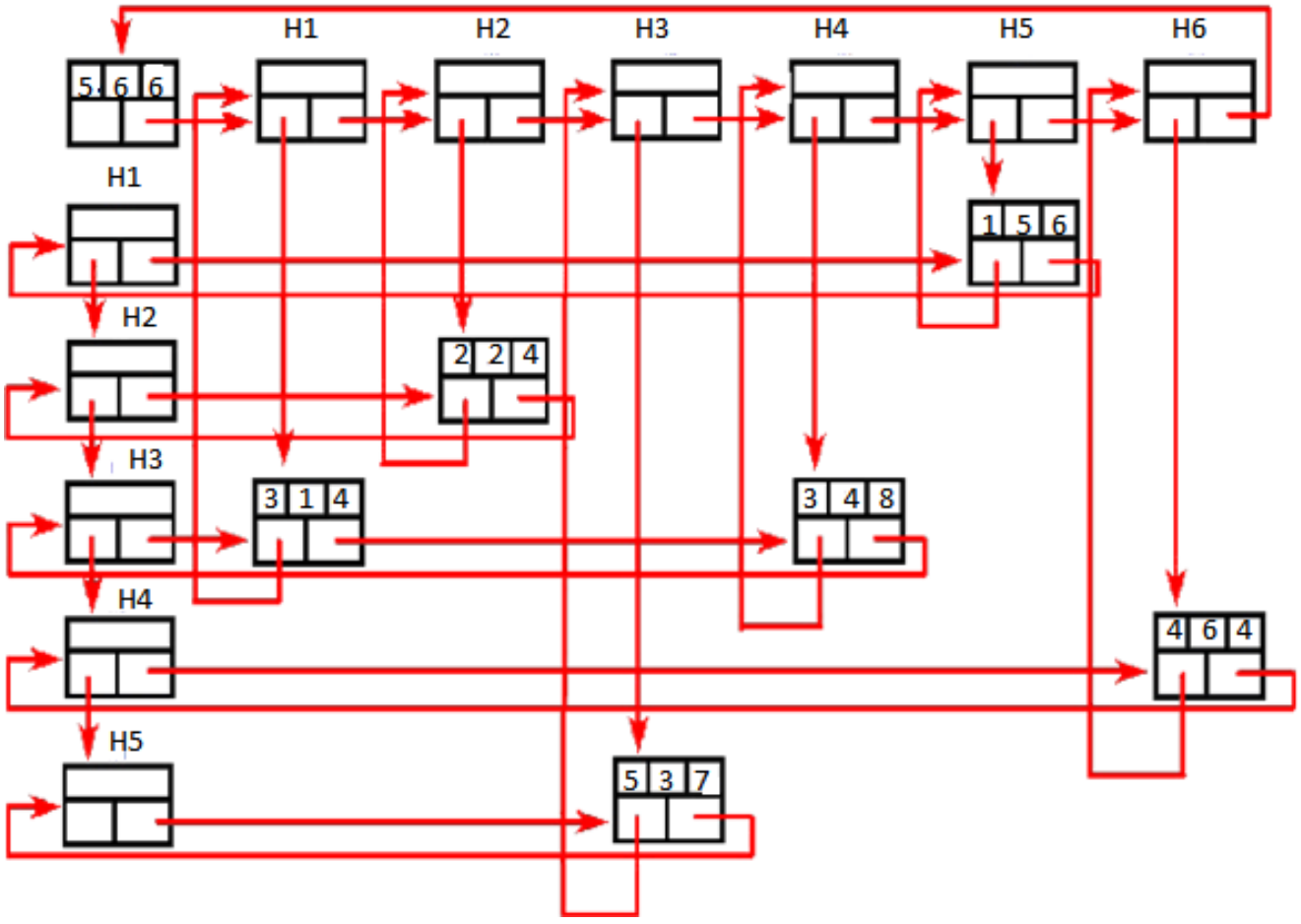


Figure 17:-Linked representation of the sparse matrix

Sparse Matrix Operations:- Input, Output & Erase.

1. **Sparse Matrix Input:-**

- The first operation is of reading in a sparse matrix and obtaining it's linked representation. The first input line consists of the number of rows, number of columns and the number of non zero

terms. This line is followed by numTerms, lines of input, each of which is of the form:- row, col, value.

- The sample input for sparse matrix is given below:-

next	
down	right

(a) Header Node

row	col	value
down	right	

(b) Element Node

Figure 18:- Node Structure for sparse Matrices

- The function mread first sets up the header nodes & then sets up each row list while simultaneously building the column lists. The next field of a header node,i,is initially used to keep track of the last node in column i.

```
matrixPointer mread(void)
{
    /* read in a matrix & sets up its linked representation */
    int num Rows,numCols,numTerms,numHead,i;
    int row,col,value,currentRow;
    matrix pointer temp,last,node;
    printf("enter the number of rows,columns,non-zero terms");
    scanf("%d%d%d",&numRows,&numCols,&numTerms);
      numHeads=(numcols>numRows) ? numCols:numRows;
      node=newNode();
      node⓪tag=entry;
      node⓪u .entry.row=numRows;
       node⓪u.entry.col=numCols;
    if(!numHeads)
        node⓪right=node;
      else
        /*initialize the header nodes8/
       {
          for(i=0;i<numHeads;i++)
           {
               temp=newNode;
               hdnode[i]=temp;
               hdnode[i]->tag = head;
               hdnode[i]->right = temp;
               hdnode[i]->u.next = temp;
           }
           currentRow = 0;
           last = hdnode[0];
           for(i=0;i<numTerms;i++)
           {
                   printf("Enter row, column & value");
                   scanf("%d %d %d", & row, & col, & value);
                   if(row>currentRow)
                   {
                        last->right = hdnode[currentRow];
                        currentRow = row;
                        last=hdnode[row];
```

```
                        }
                        MALLOC(temp, sizeof(*temp));
                        temp->tag = entry;
                        temp->u.entry.row=row;
                        temp->u.entry.col=col;
                        temp->u.entry.value=value;
                        last->right = temp;
                        hdnode[col]->u.next->down= temp;
                        hdnode[col]->u.next = temp;
                    }
                    last->right = hdnode[currentRow];
                    for(i=0;i<numCols;i++)
                        hdnode[i]->u.next->down = hdnode[i];
                    hdnode[numHeads-1]->u.next=node;
                    node->right = hdnode[0];
                }
                return node;
        }
```

2. Sparse Matrix Output:-

To print out the contents of a sparse matrix in a form. The function mwrite is written as below:

```
void mwrite(matrixPointer node)
{
    /*print out the matrix in row major form */
    int i;
    matrixPointer temp, head=node->right;
    printf("numRows=%d, numCols=%d\n", node->u.entry.row, node->u.entry.col);
    for(i=0;i<node->u.entry.row;i++)
    {
        for(temp=head->right;temp!=head;temp=temp->right)
            printf("%5d %5d %5d\n", temp->u.entry.row, temp->u.entry.col,
            temp->u.entry.value);
            head=head->u.next;
        }
}
```

Module 4
GRAPHS

Topics:

Graphs: Definitions, Terminologies, Matrix and Adjacency List Representation of Graphs, Traversal methods: Breadth First Search and Depth First Search.

Hashing: Hash Table organizations, Hashing Functions, Static and Dynamic Hashing.

INTRODUCTION: GRAPH

• A graph G consists of 2 sets, V and E.

V is a finite, on empty set of vertices.

E is a set of pairs of vertices, these pairs are called edges. V(G) and E(G) represents the set of vertices and edges respectively of graph G (Figure 1).

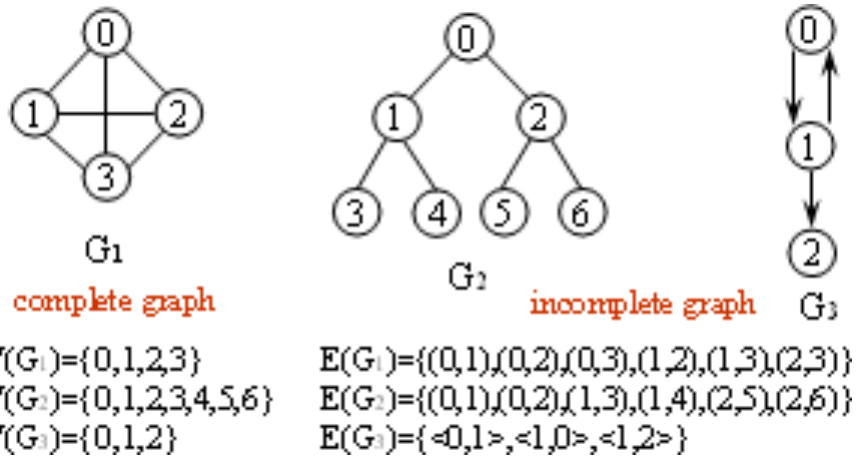


Figure 1: **Three sample graphs**

There are two types of graphs

1. Undirected graph
2. Directed graph

• In an **undirected graph**, the pair of vertices representing any edge is unordered. Thus, the pairs (u,v) and (v,u) represent the same edge.

• In a **directed graph**, each edge is represented by a directed pair <u,v>; u is the tail and v is the head of the edge. Therefore, <u,v> and <v,u> represent two different edges.

Following are the restrictions on graphs

1) A graph may have an edge from a vertex v back to itself. Such edges are known as **self loops** (Fig 2).

2) A graph may not have multiple occurrences of the same edge. If we remove this restriction, a data object referred to as **multigraph**.

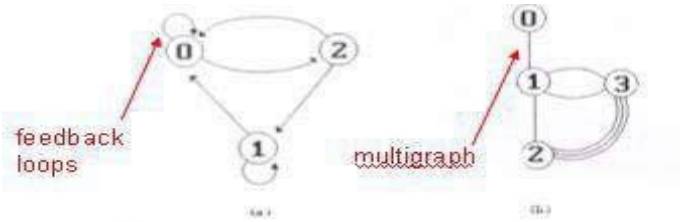


Figure 6.3: Examples of graph like structures

• Maximum number of edges in any n-vertex, undirected graph is n(n-1)/2.
• Maximum number of edges in any n-vertex, directed graph is n(n-1).

TERMINOLOGIES USED IN A GRAPH

• **Subgraph** of G is a graph G' such that V(G') belongs V(G) and E(G') belongs E(G) (Figure 3).

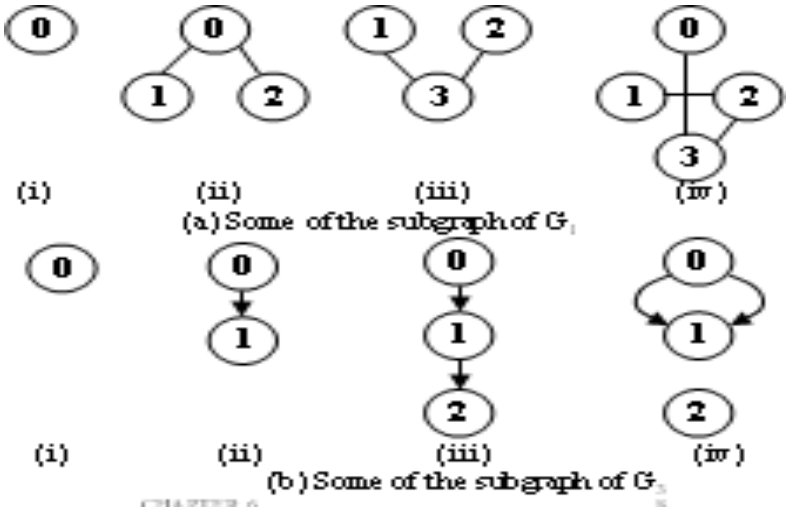


Figure 3:Some subgraphs

• A **path** from vertex u to vertex v in graph G is a sequence of vertices u,i1,i2 . . . ik, v such that (u,i1),(i1,i2) ..... (ik, v) are edges in E(G).
• A **simple path** is a path in which all vertices except possibly the first and last are distinct.
• A **cycle** is a simple path in which the first and last vertices are the same.
• A undirected graph is said to be **connected** iff for every pair of distinct vertices u & v in V(G) there is a path from u to v in G.
• A **connected component H** of an undirected graph is a maximal connected subgraph. (Figure 4).

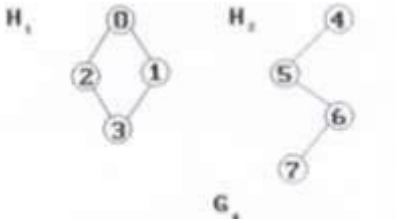


Figure 4:A graph with two connected components

• A **tree** is a connected acyclic(i.e. has no cycles) graph.
• A directed graph G is said to be **strongly connected** iff for every pair of distinct vertices u and v in V(G),there is a directed path from u to v and also from v to u (Figure 5). The graph G3 is not strongly connected as there is no path from vertex2 to 1. A strongly connected component is a maximal subgraph that is strongly connected. G3 has two strongly connected components.

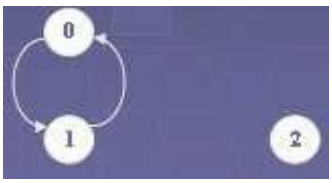


Figure 5: Strongly connected components of G3

• The **degree** of a vertex is the number of edges incident to that vertex. (Degree of vertex 0 is 3)

• In a directed graph G, **in-degree** of a vertex v defined as the number of edges for which v is the head. The **out-degree** is defined as the number of edges for which v is the tail. (Vertex 1 of G3 has in-degree 1, out-degree 2 and degree 3).

GRAPH REPRESENTATIONS

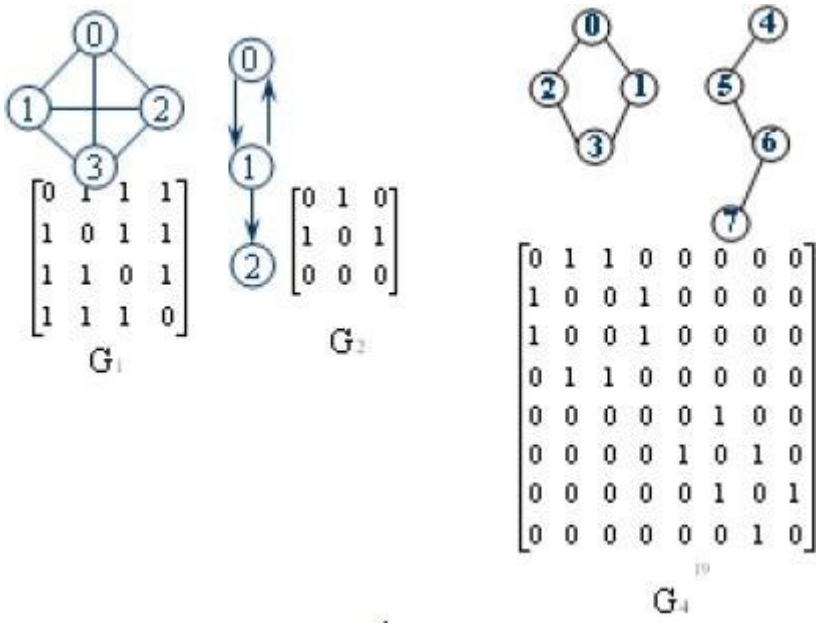
• Three commonly used representations are:

1) Adjacency matrices,
2) Adjacency lists and
3) Adjacency Multilists

1) Adjacency Matrix

• Let G=(V,E) be a graph with n vertices, n>=1.

• The adjacency matrix of G is a two-dimensional n*n array(say a) with the property that a[i][j]=1 iff the edge (i,j) is in E(G). a[i][j]=0 if there is no such edge in G (Figure 6).



• The space needed to represent a graph using its adjacency matrix is n^2 bits.

• About half this space can be saved in the case of undirected graphs by storing only the upper or lower triangle of the matrix. Figure 6 Adjacency matrices

2) Adjacency Lists

• The n rows of the adjacency matrix are represented as n chains.

• There is one chain for each vertex in G.

• The data field of a chain node stores the index of an adjacent vertex (Figure 6.8).

• For an undirected graph with n vertices and e edges, this representation requires an array of size n and 2e chain nodes.

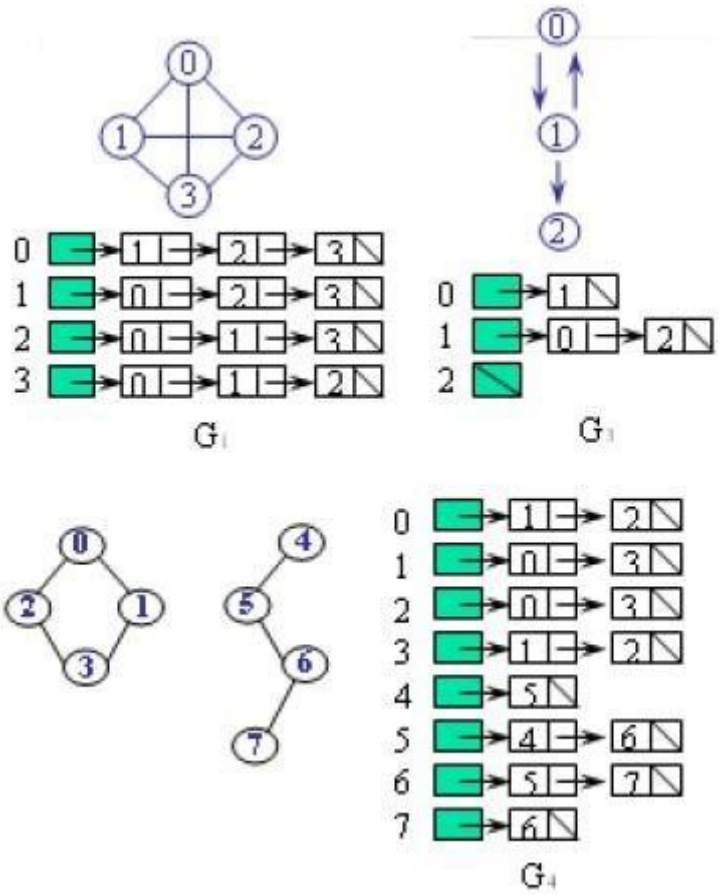


Figure 7 adjacency lists.

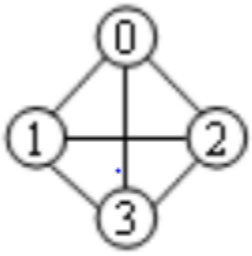
3) Adjacency Multilist

An edge in an undirected graph is represented by two nodes in adjacency list representation.

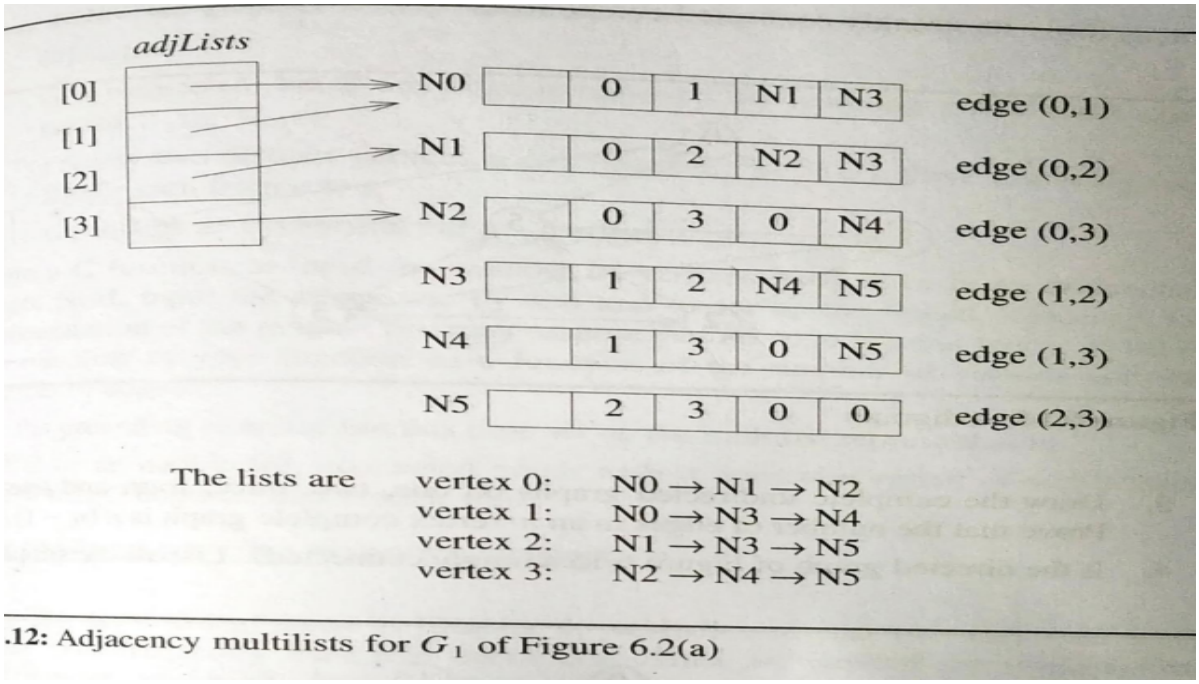
Adjacency Multilists are lists in which nodes may be shared among several lists. (an edge is shared by two different paths). For each edge there will be exactly one node, but this node will be in two lists(i,e, the adjacency lists for each of the two nodes to which it is incident). The node structure is

m	vertex1	vertex2	link1	link2

Where m is single bit field to indicate whether the edge has been examined or not.



G1



.12: Adjacency multilists for G_1 of Figure 6.2(a)

GRAPH ABSTRACT DATA TYPE

structure Graph is
objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices
functions: for all graph ∈ Graph, v, v1 and v2 ∈ Vertices
Graph Create()::=return an empty graph
Graph InsertVertex(graph, v)::= **return** a graph with v inserted. v has no incident edge.
Graph InsertEdge(graph, v1,v2)::= **return** a graph with new edge between v1 and v2
Graph DeleteVertex(graph, v)::= **return** a graph in which v and all edges incident to it are removed
Graph DeleteEdge(graph, v1, v2)::=**return** a graph in which the edge (v1, v2) is removed
Boolean IsEmpty(graph)::= **if** (graph==empty graph)
return TRUE
else
return FALSE
List Adjacent(graph,v)::= **return** a list of all vertices that are adjacent to v

ADT OF GRAPH

ELEMENTARY GRAPH OPERATIONS/ GRAPH TRAVERSALS

1. Depth First Search (DFS)
2. Breadth First Search (BFS)

DFS and BFS are common methods of graph traversal, which is the process of visiting every vertex of a graph. Stacks and queues are two additional concepts used in the DFS and BFS algorithms.

A stack is a type of data storage in which only the last element added to the stack can be retrieved. It is like a stack of plates where only the top plate can be taken from the stack.

The three stacks operations are:

- Push – put an element on the stack
- Peek – look at the top element on the stack, but do not remove it
- Pop – take the top element off the stack.

A queue is a type of data storage in which the elements are accessed in the order they were added. It is like a cafeteria line where the person at the front of the line is next.

The two queues operations are:

- Enqueue – add an element to the end of the queue.
- Dequeue – remove an element from the start of the queue.

Depth First Search (DFS) (It is similar to a preorder tree traversal)

- We begin the search by visiting the start vertex, v.
- Next, we select an unvisited vertex, w, from v's adjacency list and carry out a depth first search on w.
- We preserve our current position in v's adjacency list by placing it on a stack.
- Eventually our search reaches a vertex, u, that has no unvisited vertices on its adjacency list.
- At this point, we remove a vertex from the stack and continue processing its adjacency list.
- Previously visited vertices are discarded; unvisited vertices are visited and places on the stack.

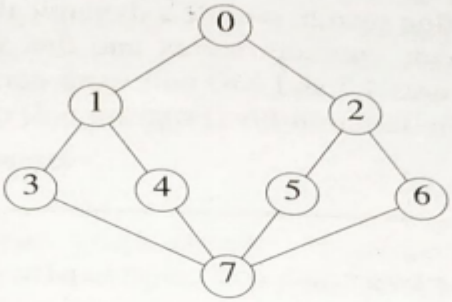
The recursive implementation of ***dfs*** is presented below. This function uses a global array, visited[MAX_VERTICES], that is initialized to *FALSE.*

When we visit a vertex, I, we change visited[i] to *TRUE.* The declarations are:

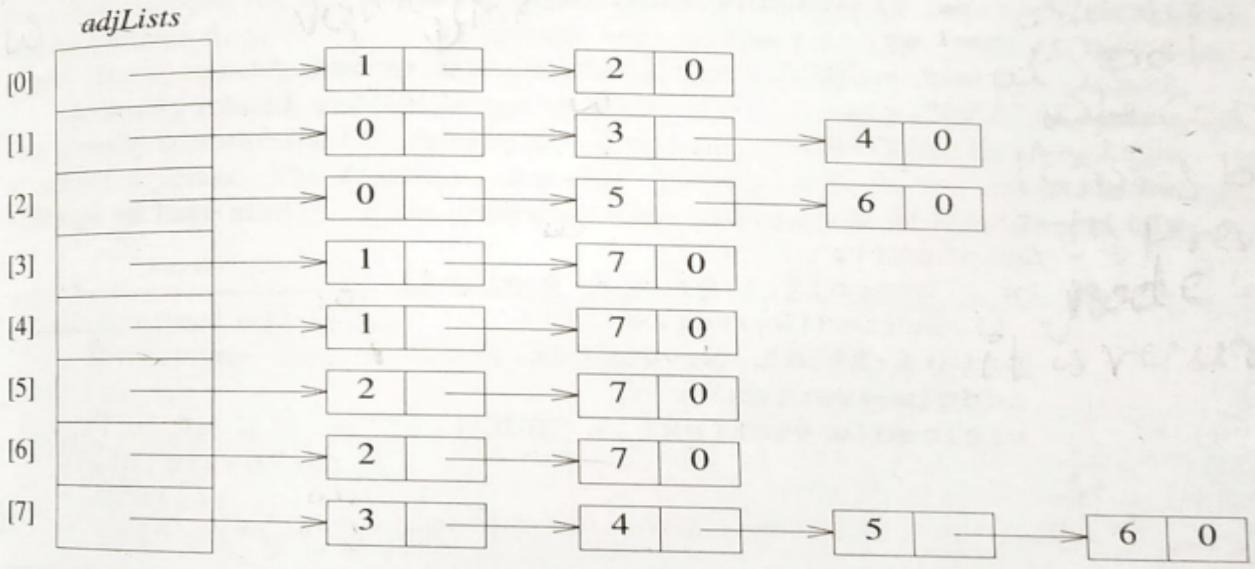
```
#define FALSE 0
#define TRUE 1
short int visited[MAX_VERTICES];
```

```
void dfs(int v)
{/* depth first search of a graph beginning at v */
   nodePointer w;
   visited[v] = TRUE;
   printf("%5d",v);
   for (w = graph[v]; w; w = w→link)
      if (!visited[w→vertex])
         dfs(w→vertex);
}
```

Program 6.1: Depth first search



(a)



(b)

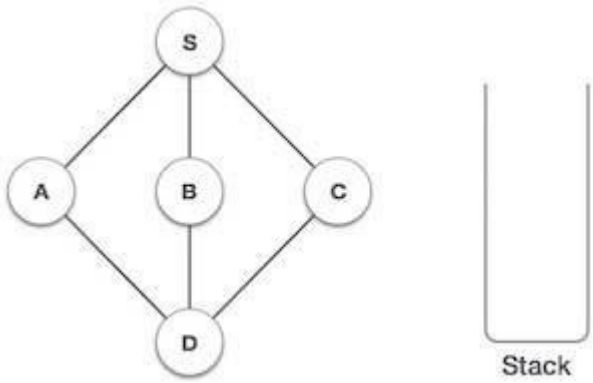
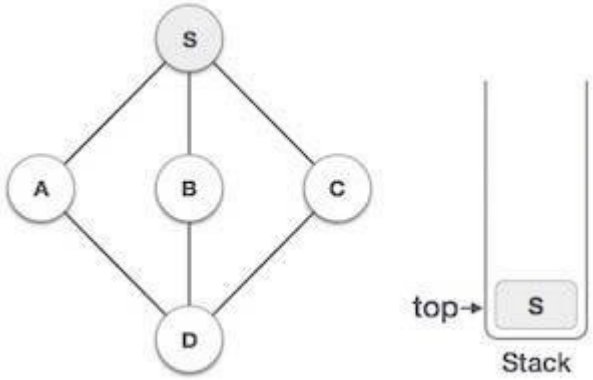
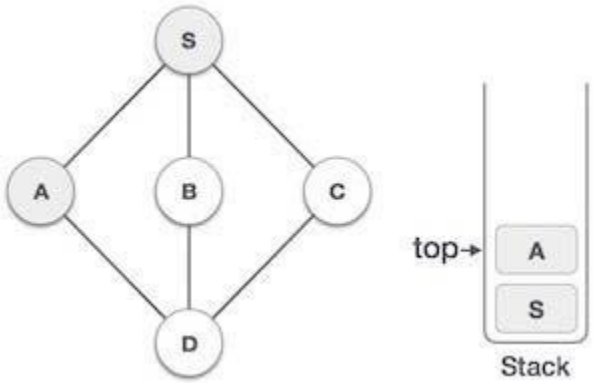
Figure 6.16: Graph *G* and its adjacency lists

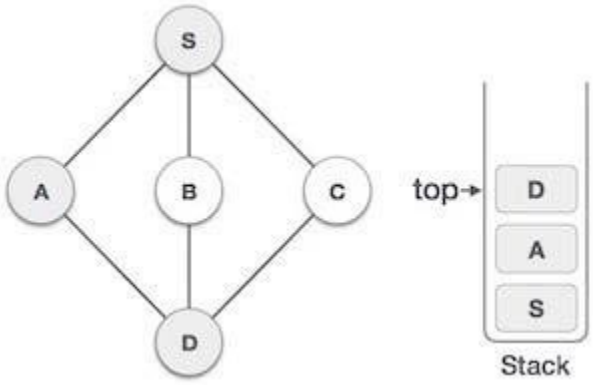
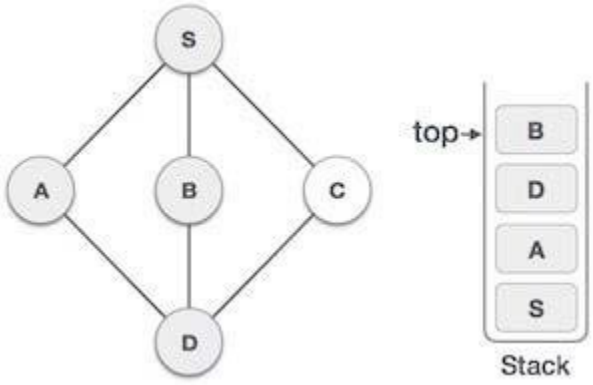
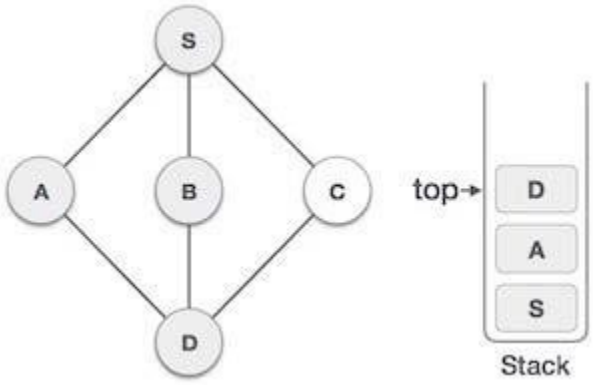
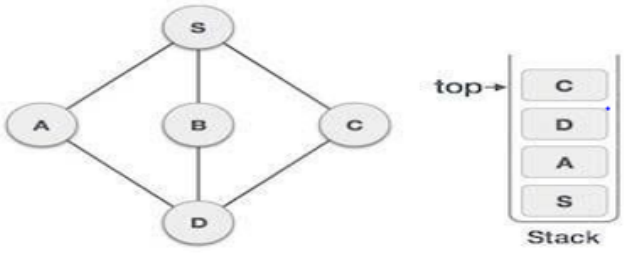
Easy understanding steps with diagram

Note: This is just for understanding you have to write above dfs function and adjacency list

DFS will visit the child vertices before visiting siblings using this algorithm:

- Mark the starting node of the graph as visited and push it onto the stack
- While the stack is not empty
- Peek at top node on the stack
- If there is an unvisited child of that node Mark the child as visited and push the child node onto the stack
- Else Pop the top node off the stack.

Step	Traversal	Description
1.		Initialize the stack.
2.		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S**and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4.		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5.		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6.		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7.		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Breadth First Search (BFS) (It is similar to a level order tree traversal)

- BFS starts at vertex v and marks it as visited. It then visit each of the vertices on v's adjacency list.
- When we have visited all the vertices on v's adjacency list, we visit all the unvisited vertices that are adjacent to the first vertex on v's adjacency list.
- As we visit each vertex we place the vertex in a queue.
- When we have exhausted an adjacency list, we remove a vertex from the queue and proceed by examining each of the vertices on its adjacency list.
- Unvisited vertices are visited and then placed on the queue; visited vertices are ignored.
- We have finished the search when the queue is empty.

The *bfs* function is implemented below:

```
void bfs(int v)
{/* breadth first traversal of a graph, starting at v
    the global array visited is initialized to 0, the queue
    operations are similar to those described in
    Chapter 4, front and rear are global */
  nodePointer w;
  front = rear = NULL; /* initialize queue */
  printf("%5d",v);
  visited[v] = TRUE;
  addq(v);
  while (front) {
    v = deleteq();
    for (w = graph[v]; w; w = w→link)
      if (!visited[w→vertex]) {
        printf("%5d", w→vertex);
        addq(w→vertex);
        visited[w→vertex] = TRUE;
      }
    }
}
```

Program 6.2: Breadth first search of a graph

The queue definition and the function prototypes used by *bfs* are:

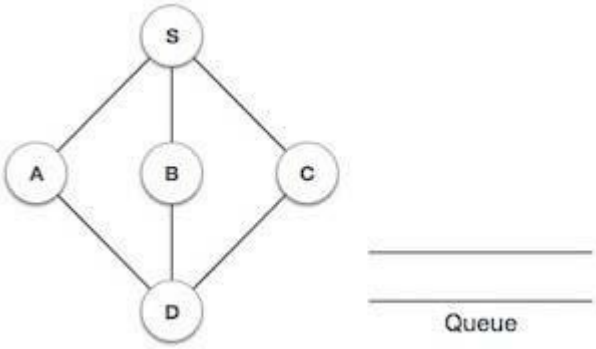
```
typedef struct queue *queuePointer;
typedef struct {
        int vertex;
        queuePointer link;
        } queue;
queuePointer front, rear;
void addq(int);
int deleteq();
```

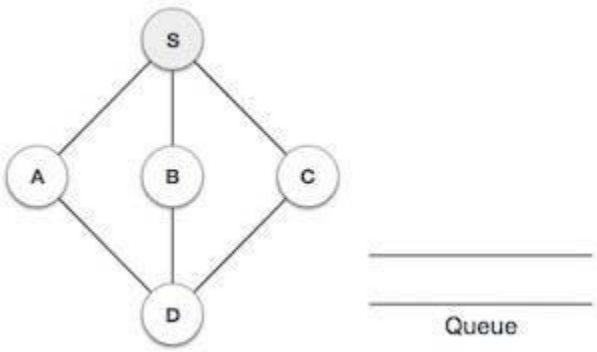
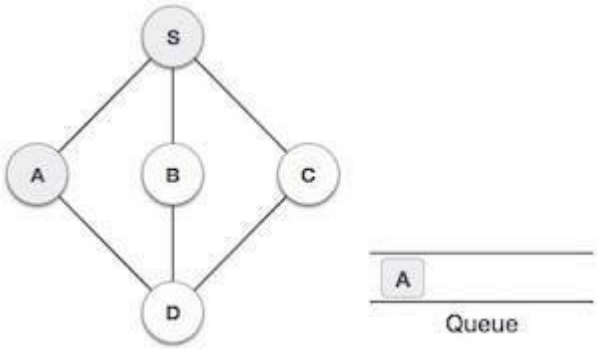
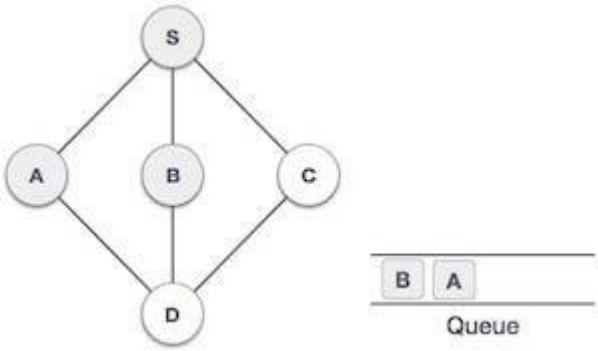
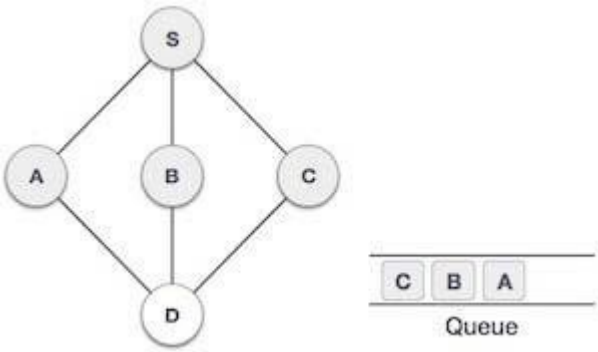
Easy understanding steps with diagram

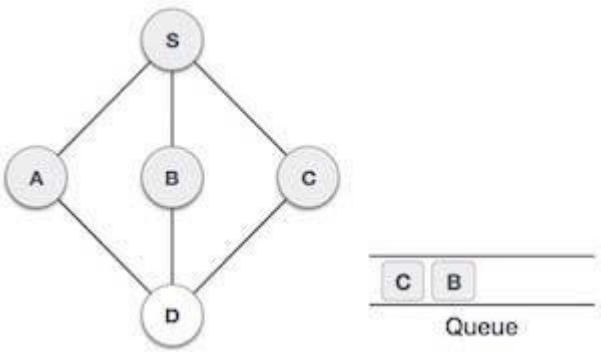
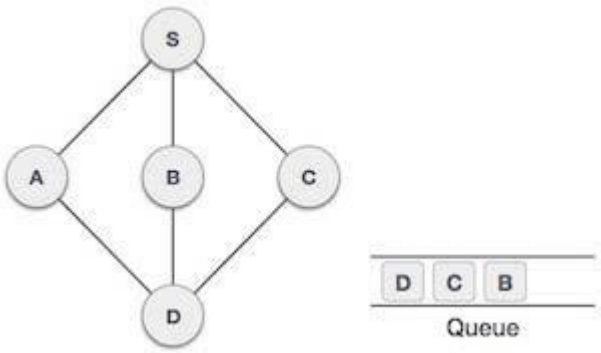
Note: This is just for understanding you have to write above bfs function and adjacency list

BFS will visit the sibling vertices before the child vertices using this algorithm:

- Mark the starting node of the graph as visited and enqueue it into the queue.
- While the queue is not empty.
- Dequeue the next node from the queue to become the current node.
- While there is an unvisited child of the current node .
- Mark the child as visited and enqueue the child node into the queue.

Step	Traversal	Description
1.		Initialize the queue.

2.		We start from visiting **S**(starting node), and mark it as visited.
3.		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4.		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5.		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

6.		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7.		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

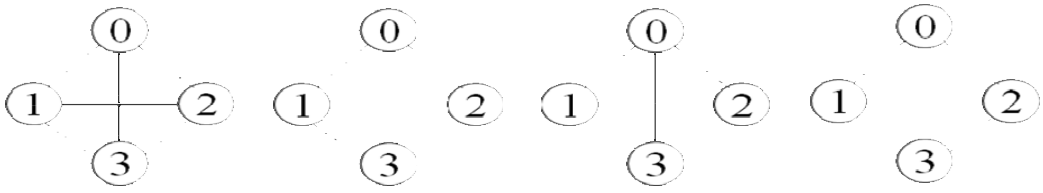
At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the Queue Gets Emptied, The Program Is Over.

Connected components:

DFS or BFS functions can be used to check if the graph is connected or not.

Spanning Trees:

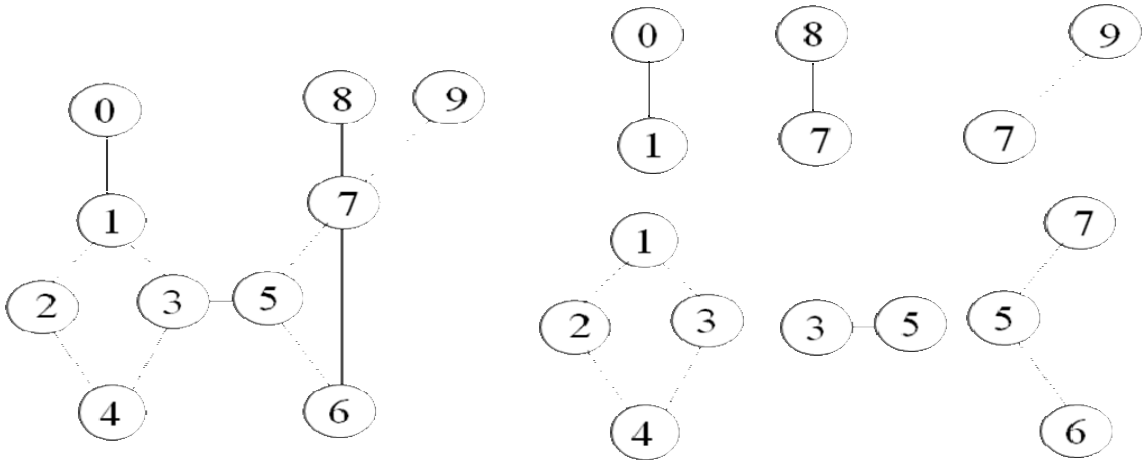
A spanning tree is any tree that consists solely of edges in G and that includes all the vertices in G.



Biconnected Components:

An articulation point is a vertex v of G such that the deletion of v, together with all edges incident on v, produces a graph G', that has atleast 2 connected components. Ex: 1,3,5 and 7 are the articulation points in the following graph.

A biconnected graph is a connected graph that has no articulation points.



Graph Biconnected components

Difference between BFS and DFS

BFS	DFS
BFS visit nodes level by level.	DFS visit nodes of graph depth wise.
We can't visit a level without visiting it's previous level.	It is not necessary to maintain level.
To implement BFS we need to use Queue.	To implement DFS we need to use Stack.
BFS is slower and need more memory.	DFS is faster and required lss memory.
It is used to find Shortest path, testing a graph for bipartite.	It is used for topological sorting, solving puzzles such as maze.

Module 5

HASHING

Hashing is an effective way to store the elements in some data structure. It allows to reduce the number of comparisons. Using the hashing technique we can obtain the concept of direct access of stored record.

The dictionary operations search, insert and delete in arrays, linked list and in BST take O(n) time. If the tree is balanced it will take O(logn) time.

Hashing enables us to perform the dictionary operations in O(1) expected time.

There are two types of hashing

1. Static Hashing
2. Dynamic hashing

STATIC HASHING

Two important aspects associated with hashing are

1. HASH TABLE
2 HASH FUNCTIONS

HASH TABLE

It is a data structure used for storing and retrieving data very quickly. Inserting data in to this table is based on key value.

Example: Storing an employee record in the table, Employee ID is used as key.

The hash key is used to search the data in the hash table. The efficient representation of dictionary can be done using hash table. The dictionary entries in the hash table are filled using hash function.

*In **static hashing** t*he dictionary pairs are stored in a table, ht, called the hash table is partitioned into b buckets, ht [0], , ht[b - 1]. A bucket is said to consist of s slots, each slot being large enough to hold one dictionary pair. Usually s=1, and each bucket can hold exactly one pair.

The address or location of a pair whose key is k is determined by a hash function, h which maps keys into buckets. Thus, for any key k, h (k) is an integer in the range 0 through b - 1. h (k) is the hash or home address of k.

Example: Consider the hash table ht with b = 26 buckets and s =2. We have n = = 10 distinct identifiers, each representing a C library function. The hash function must map each of the possible identifiers on to one of the numbers, 0-25. We can construct a fairly simple hash function by associating the letters, a-z, with the numbers, 0-25, respectively, and then defining the hash function, f (x), as the first character

of x.

Using this scheme, the library functions **acos, define, float, exp, char, atan, ceil, floor, clock**, and **ctime** hast into buckets 0, 3, 5, 4, 2, 0, 2, 5, 2, and 2, respectively. Figure shows the first 8 identifiers entered in to the hash table.

	Slot 0	Slot 1
0	**acos**	**atan**
1		
2	**char**	**ceil**
3	**define**	
4	**exp**	
5	**float**	**floor**
6		
. . .		
25		

: Hash table with 26 buckets and two slots per bucket

The identifiers **acos** and **atan** are synonyms, as are **float** and **floor**, and **ceil** and **char.** The next identifier, **clock**, hashes into the bucket ht [2]. Since this bucket is full, we have an overflow.

When no overflows occur, the time required to insert, delete or search using hashing depends only on the time required to compute the hash function and the time to Search one bucket. Hence, the insert, delete and search times are independent of n, the number of entries in the dictionary.

The hash function of above example is not well suited for most practical applications because of the very large number of collisions and resulting overflows that occur. This is because it is not unusual to find dictionaries in which many of the keys begin with the same letter. Ideally, we would like to choose a hash function that is both easy to compute and results in very few collisions.

Hashing schemes use a hash function to map keys into hash-table buckets. It is desirable to use a hash function that is both easy to compute and minimizes number of collisions. Hence, a mechanism to handle overflows is needed.

HASH FUNCTIONS

It is a function which is used to map a key into a bucket in the hash table. The integer returned by hash function is called hash key.

A hash function is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table.

DIFFERENT HASH FUNCTIONS

1. Division method

It is the most simple method of hashing an integer x. The home bucket is obtained by using the modulo(%) operator. This method divides k by D and then uses the remainder as the home bucket for k. In this case, the hash function can be given as:

$$h(k) = k \bmod D$$

where k is key and D is size of the hash table

Example: Let the keys are 13, 74, 11, 15, 16 and D=10 . The bucket addresses ranges from 0 to D-1 i,e 0 to 9

h(13) = 13 % 10 =3
h(74) = 74 % 10 =4
h(11) = 11 % 10 =1
h(15) = 15 % 10 =5
h(16) = 16 % 10 =6

0	
1	11
2	
3	13
4	74
5	15
6	16
7	
8	
9	

2. Multiplication Method

The steps involved in the multiplication method are as follows:

***Step 1*:** Choose a constant A such that 0 < A < 1.

***Step 2*:** Multiply the key k by A.

***Step 3*:** Extract the fractional part. kA.

***Step 4*:** Multiply the result of Step 3 by the size of hash table (m).

Hence, the hash function can be given as:

h(k) = floor(m *(kA -floor(kA)))

where A= 0.65 i,e 0<A<1 and m is the total number of buckets in the hash table.

Example1: Given a hash table of size m=10, map the key 60 to an appropriate location in the hash table.

Solution Assuming A = 0.65, Given: m = 10, and k = 60

h(60) = floor(10* (60 *0.65 – floor(60 *0.65)))

h(60) = floor(10* (39 – 39))

h(60) = floor(10* 0)

h(60) = 0

Example2: Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table.

Solution Assuming A = 0.618033. Given: m = 1000, and k = 12345

h(k) = floor(m *(kA mod 1))

h(12345) = floor(1000 (12345 *0.618033 mod 1))

h(12345) = floor(1000 (7629.617385 mod 1))

h(12345) = floor(1000 (0.617385))

h(12345) = floor(617.385)

h(12345) = 617

3. Mid-Square Method

The mid-square method is a hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2.

Step 2: Extract the middle r digits of the result obtained in Step 1.

Example 15.3 Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.

Solution Note that the hash table has 100 memory locations whose indices vary from 0 to 99.

This means that only two digits are needed to map the key to a location in the hash table, so r = 2.

After squared choose middle part of the key.

When k = 5642, k^2 = 318**32**164, h (5642) = 32, place the key in the bucket 32

If table size is 1000 whose indices vary from 0 to 999

When k = 1234, k^2 = 15**227**56, h (1234) = 227, place the key in the bucket 227

4. Folding Method

In this method the key k is partitioned into several parts, all but possibly the last being the same length. These partitions are then added together to obtain the hash address for k.

There are two ways of carrying out this addition.

In the **first**, all but the last partition shifted to the right so that the least significant digit of each lines up with corresponding digit of the last partition. The different partitions are now added together to get h (k). This method is known as ***shift folding***.

In the **second** method, ***folding at the boundaries***, the key is folded at the partition boundaries, and digits falling into the same position are added together to obtain h (k). This is equivalent to reversing every other partition and then adding.

Example: Suppose that k = 12320324111220, and we partition it into parts that three decimal digits long.

The partitions are P1 = 123, P2 = 203, P3 = 241, P4=112 and P5 = 20.

Using shift folding, we obtain

$h(k) = \sum P_i = P1 + P2 + P3 + P4 + P5 = 123 + 203 + 241 + 112 + 20 = 699$

when ***folding at the boundaries is used***, we first reverse P2 and P4 to obtain 302 and 211, respectively. Next, the five partitions are added to obtain h (k) = 123 + 302 + 241 + 211 + 20 = 897

5. Converting Keys to Integers

To use some of the described hash functions, keys need to first be converted to non-negative integers. we consider only the conversion of strings into non-negative integers.

```
unsigned int stringToInt(char *key)
{/* simple additive approach to create a natural number
    that is within the integer range */
   int number = 0;
   while (*key)
      number += *key++;
   return number;
}
```

Program 8.1: Converting a string into a non-negative integer

The above program converts each character into a unique integer and sums these unique integers. Since each character maps to an integer in the range 0 through 255.

OVERFLOW HANDLING

There are two popular ways to handle overflows

1. open addressing
2. chaining

1. OPEN ADDRESSING

we describe four open addressing methods

1. linear probing, which also is known linear open addressing,
2. quadratic probing,
3. rehashing
4. random probing.

Linear probing

In linear probing, when inserting a new pair whose key is k, we search the hash table buckets in the order, **ht [h (k) + i] % b**, $0 \leq i \leq b-1$ where h is the hash function and b is the number of buckets. This search terminates when we reach the first unfilled bucket and the new pair is inserted into this bucket. In case no such bucket is found, the hash table is full and it is necessary to increase the table size. Notice that when we resize the hash table, we must change the hash function as well.

For example, when the division hash function is used, the divisor equals the number of buckets. This change in the hash function potentially changes the home bucket for each key in the hash table. So, all dictionary entries need to be remapped into the new larger table.

Example: Assume we have a 15-bucket table with one slot per bucket. As our data we use the words **for, do, while, If, else, and function**. Figure 8.2 shows the hash value for each word using the simplified scheme or program 8.1 and the division hash function. Inserting the first five words into the table poses no problem since they have different hash addresses, However, the last 1dentifier, **function**, hashes to the same bucket as **if.** Using a **circular rotatio**n, the next available bucket is at ht[0], which is where we place **function** (Figure 8.3).

Identifier	Additive Transformation	x	Hash
for	102 + 111 + 114	327	2
do	100 + 111	211	3
while	119 + 104 + 105 + 108 + 101	537	4
if	105 + 102	207	12
else	101 + 108 + 115 + 101	425	9
function	102 + 117 + 110 + 99 + 116 + 105 + 111 + 110	870	12

Figure 8.2: Additive transformation

[0]	**function**
[1]	
[2]	**for**
[3]	**do**
[4]	**while**
[5]	
[6]	
[7]	
[8]	
[9]	**else**
[10]	
[11]	
[12]	**if**

Figure 8.3: Hash table with linear probing (13 buckets, one slot per bucket)

When $s = 1$ and linear probing is used to handle overflows, a hash table search for the pair with key k proceeds as follows:

(1) Compute $h(k)$.

(2) Examine the hash table buckets in the order $ht[h(k)]$, $ht[(h(k) + 1) \% b]$, $\cdots$, $ht[(h(k) + j) \% b]$ until one of the following happens:

(a) The bucket $ht[(h(k) + j) \% b]$ has a pair whose key is k; in this case, the desired pair has been found.

(b) $ht[h(k) + j]$ is empty; k is not in the table.

(c) We return to the starting position $ht[h(k)]$; the table is full and k is not in the table.

Program 8.3 is the resulting search function. This function assumes that the hash table *ht* stores pointers to dictionary pairs. The data type of a dictionary pair is *element* and data of this type has two componenets *item* and *key*.

```
element* search(int k)
{/* search the linear probing hash table ht (each bucket has
    exactly one slot) for k, if a pair with key k is found,
    return a pointer to this pair; otherwise, return NULL */
   int homeBucket, currentBucket;
   homeBucket = h(k);
   for (currentBucket = homeBucket; ht[currentBucket]
                && ht[currentBucket]->key != k;) {
      currentBucket = (currentBucket + 1) % b;
                      /* treat the table as circular */
      if (currentBucket == homeBuket)
         return NULL; /* back to start point */
   }
   if (ht[currentBucket]->key == k)
      return ht[currentBucket];
   return NULL;
}
```

Program 8.3: Linear probing

When linear probing is used to resolve overflows, keys tend to **cluster** together and adjacent clusters tend to coalesce, thus **increasing the search time**.

For example, suppose we enter the C built-in functions **acos**, **atoi, char**, **define**, **exp**, **ceil**, **cos**, **float**, **atol**, **floor**, and **ctime** into a 26-bucket hash table in that order. For illustrative purposes, we assume that the hash function uses the first character in each function name.

Figure 8.4 shows the bucket number, the identifier contained in the bucket, and the number of comparisons required to insert the identifier. Notice that before we can insert atol, we must examine ht [0], ... , ht[8], a total 9 nine comparisons. The average number of buckets examined would be 41/11 = 3.72 per identifier.

Disadvantage of liner probing**:** keys tend to **cluster** together and adjacent clusters tend to coalesce, thus **increasing the search time**.

bucket	*x*	buckets searched
0	**acos**	1
1	**atoi**	2
2	**char**	1
3	**define**	1
4	**exp**	1
5	**ceil**	4
6	**cos**	5
7	**float**	3
8	**atol**	9
9	**floor**	5
10	**ctime**	9
. . .		
25		

Figure 8.4: Hash table with linear probing (26 buckets, one slot per bucket)

Some improvement in the growth of clusters and hence in the average number of comparisons needed for searching can be obtained by *quadratic probing*.

Quadratic probing

In *quadratic probing*, a quadratic function of i is used as the increment. In particular, the search is carried out by examining buckets h (k), (h (k) + i^2) % b, and (h (k) – i^2) % b for 1≤(b- 1)/2. When b is a prime number of the form 4j + 3, for j an integer, the quadratic search described above examines every bucket in the table. Figure 8.5 lists some primes of the form 4j + 3.

Prime	*j*	Prime	*j*
3	0	43	10
7	1	59	14
11	2	127	31
19	4	251	62
23	5	503	125
31	7	1019	254

Figure 8.5: Some primes of the form $4j + 3$

An alternative method to retard the growth of clusters is to use a series of functions h1, h2,….., hm. This method is known as ***Rehashing***. Buckets $h_i(k)$, $1 \le i \le m$ are examined in that order.

Yet another alternative is, ***random probing***, a pseudo-random number generator is used to obtain a random sequence R(i), $1 \le i \le b$ where R(1), R(2),….., R(b-1) is a permutation of [1, 2,……, b-1].
Random hashing is easy to analyze but because of the expense of random number generation it is not often used.

2. Chaining

Linear probing and its variations perform poorly because the search for a key involves comparison with keys that have different hash values.

Many of the comparisons can be saved if we maintain lists of keys, one list per bucket, each list containing all the synonyms for that bucket. If this is done, a search involves computing the hash address h(k) and examining only those keys in the list for h(k). We typically use an array ht [0:b-1] with ht[i] pointing to the first node of the chain for bucket i.
Program 8.4 gives the search algorithm for chained hash tables.

```
element* search(int k)
{/* search the chained hash table ht for k, if a pair with
    this key is found, return a pointer to this pair;
    otherwise, return NULL.
   nodePointer current;
   int homeBucket = h(k);
   /* search the chain ht[homeBucket] */
   for (current = ht[homeBucket]; current;
                                  current = current→link)
      if (current→data.key == k) return &current→data;
   return NULL;
}
```

Program 8.4: Chain search

Figure 8.6 shows the chained hash table corresponding to the linear table found in figure 8.4. The number of comparisons needed to search for any of the identifiers is now one each for **acos, char, define**, **exp** and **float**; two each for **atoi**, **ceil**, and **float**; three each for **atol** and **cos**; and four for **ctime**. The average number of comparisons is now 21/11 = 1.91.

[0] → **acos atoi atol**
[1] → *NULL*
[2] → **char ceil cos ctime**
[3] → **define**
[4] → **exp**
[5] → **float floor**
[6] → *NULL*
. . .
[25] → *NULL*

Figure 8.6: Hash chains corresponding to Figure 8.4

To insert a new key, k, into a chain, we must first verify that it is not currently on chain. Following this, k may be inserted at any position of the chain. Deletion from a chained hash table can be done by removing the appropriate node from its chain.

When chaining is used along with a uniform hash function, the expected average number of key comparisons for a successful search is ≈ 1 + α/2, where α is the loading density n/b (b = number of buckets). For α = 0.5 this number is 1.25, and for a α = 1 it is 1.5. The corresponding numbers for linear probing are 1.5 and b, the table size.

DYNAMIC HASHING (Extendible hashing)

Motivation for Dynamic Hashing

To ensure good performance, it is necessary to increase the size of a hash table whenever its loading density exceeds a prespecified threshold. So. for example. if we currently have b buckets in our hash table and are using the division hash function with divisor D=b, then, when an insert causes the loading density to exceed the prespecified threshold, we use *array doubling* to increase the number of buckets to 2b + 1.

At the same time, the hash function divisor changes to 2b +1. This change in divisor requires us to rebuild the hash table by collecting all dictionary pairs in the original smaller size table and reinserting these into the new larger table. We cannot simply copy dictionary entries from the smaller table into corresponding buckets of the bigger table as the home bucket for each entry has potentially changed. For very large dictionaries that must be accessible on a 24/7 basis, the required rebuild means that dictionary operations must operations must be suspended for unacceptably long periods while the rebuild is in progress.

Dynamic hashing, which also is known as ***extendible hashing***, aims to reduce the rebuild time by ensuring that each rebuild changes the home bucket for the entries in only 1 bucket. In other words, although

table doubling increases the total time for a sequence of n dictionary operations by only O(n), the time required to complete an insert that triggers the doubling is excessive in the context of a large dictionary that is required to respond quickly on a per operation basis.

The *objective of dynamic hashing* is to provide acceptable hash table performance on a per operation basis. We consider *two forms of dynamic hashing*- ***one uses a directory*** and the ***other does not***.

For both forms, we use a hash function h that maps keys into non-negative integers. The range of h is assumed to be sufficiently large and we use h(k,p) to denote the integer formed by the p least significant bits of h(k).

For the examples, we use a hash function h(k) that transforms keys into 6-bit non-negative integers. Our example keys will be two characters each and h transforms letters such as A, B and C into the bit sequence 100, 101, and 110, respectively. Digits 0 through 7 are transformed into their 3-bit representation.

Figure 8.7 shows 8 possible 2 character keys together with the binary representation of h(k) for each. For our example hash function, h(A0,1) = 0, h (A1,3) = 1, h (B1,4) = 1001 = 9, and h (C1,6) = 110 001 = 49.

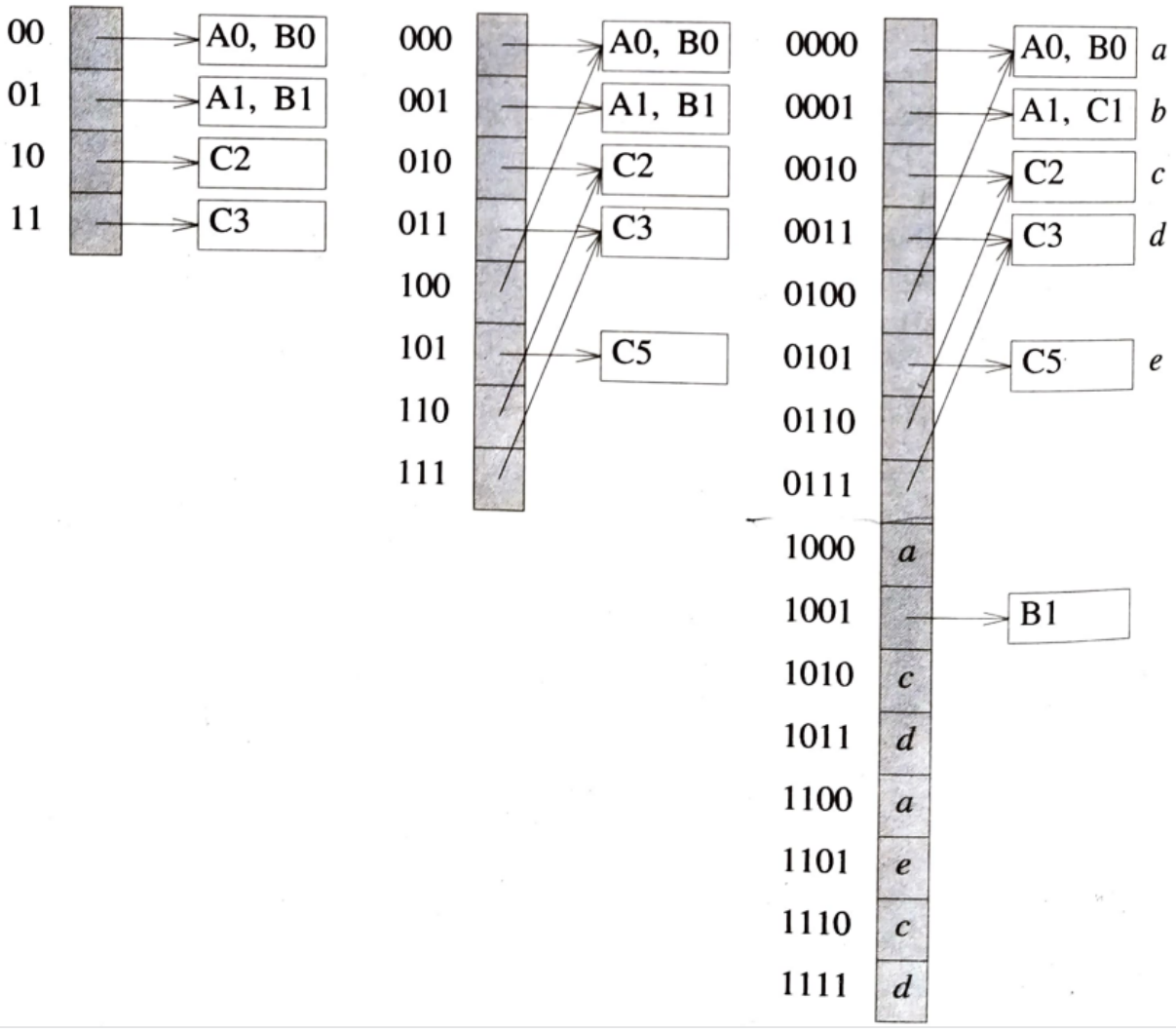
k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

Figure 8.7: An example hash function

Dynamic Hashing Using Directories

A directory, d, of pointers to buckets is used. The size of the directory depends on the number of bits of h(k) used to index into the directory. When indexing is done using, h(k, 2), the directory size is $2^2 = 4$; when h (k, 5) is used, the directory size is 32. The number of bits of h (k) used to index the directory is called the ***directory depth*.**

The size of the directory is 2^t, where t is the directory depth and the number of buckets is at most equal to the directory size. Figure 8.8 (a) shows a dynamic hash table that contains the keys A0, B0, A1, B1, C2, and C3. This hash table uses a directory whose depth is 2 and uses buckets that have 2 slots each.



(a) depth = 2 (b) depth = 3 (c) depth = 4

Figure 8.8: Dynamic hash tables with directories

Suppose we ***insert C5*** into the hash table of Figure 8.8 (a). Since, h (C5,2) = Since, h (C5,2) = 01 This gets us to the bucket with Al and BI. This bucket is full and we get a bucket overflow.

To resolve the overflow, we determine the least u such that h(k,u) is not the same for all keys in the overflowed bucket. In case the least u is greater than the directory depth, we increase the directory depth to this least u value. This requires us to increase the directory size but not the number of buckets. When the directory size doubles, the pointers in the original directory are duplicated so that the pointers in each half of the directory are the same. A quadrupling of the directory size may be handled as two doublings and so on.

For our example, the least u for which h (k, u) is not the same for Al, B1, and C5 is 3. So, the directory is expanded to have depth 3 and size 8. Following the expansion, d[i] = d[i + 4],$0 \le i \le 4$.

Following the resizing of the directory, we split the overflowed bucket using h (k,u). In our case, the overflowed bucket is split using h (k, 3). For **Al** and **B1**, h(k, 3)= **001** and for **C5**, h (k, 3) = **101**. So, we create a new bucket with C5 and place a pointer to this bucket in d[101]. Figure 8.8 (b) shows the result.

Notice that each dictionary entry is in the bucket pointed at by the directory position h (k, 3), although, in some cases the dictionary entry is also pointed at by other buckets. For example, bucket 100 also points to A0 and BO, even though h (A0,3) = h (B0,3) ≠ 000.

Suppose if we want to ***insert C1***. The pointer in position h (C1,2) = 01 of the directory of Figure 8.8 (a) gets us to the same bucket as when we were insert. This bucket overflows. The least u for which h(k,u) isn't the same for Al, BI and C1 is 4. So, the new directory depth is 4 and its new size is 16. The directory size is quadrupled and the pointers d [0:3] are replicated 3 times to fill the new directory. When the overflowed bucket is split, **A1** and **Cl** are placed into a bucket that is pointed at by **d[0001**] and B1 into a bucket pointed at by **d[1001].**

Consider ***inserting A4*** (h (A4) = 100 100) into Figure 8.8 (b). Bucket d[100] overflows. The least u is 3, which equals the directory depth. So, the size of the directory is not changed. Using h (k, 3), A0 and B0 hash to 000 while A4 hashes to 100. So, we create a new bucket for A4 and set d [100] to point to this new bucket.

Directoryless Dynamic Hashing

This method is also known as *linear dynamic hashing*. In the previous case we use an array, ht, of buckets with large size. To avoid initializing such a large array, we use two variables q and r, $0 \le q < 2^r$, to keep track of the active buckets. At any time, only buckets 0 through $2^r + q - 1$ are active. Each active bucket is the start of a chain of buckets. The remaining buckets on a chain are called *overflow buckets*.

Informally,

r is the number of bits of h (k) used to index into the hash table and

q is the bucket that will split next.

More accurately, buckets 0 through q - 1 as well as buckets 2^r through $2^r + q - 1$ are indexed using h (k, r+ 1) while the remaining active buckets are indexed using h(k, r). Each dictionary pair is either in an active or an overflow bucket.

Figure 8.9 (a) shows a directoryless hash table ht with r = 2 and q = 0. The hash function is h(B4) = 101 100, and h(B5) = 101 101. The number of active buckets is 4 (indexed 00, 01, 10, and 11). The index of an active bucket identifies its chain.

Each active bucket has 2 slots and bucket 00 contains B4 and A0. There are 4 bucket active chains, each chain begins at one of the 4 active buckets and comprises only that active bucket (i.e., there are no overflow buckets). In Figure 8.9 (a), all keys have been mapped into chains using h (k, 2).

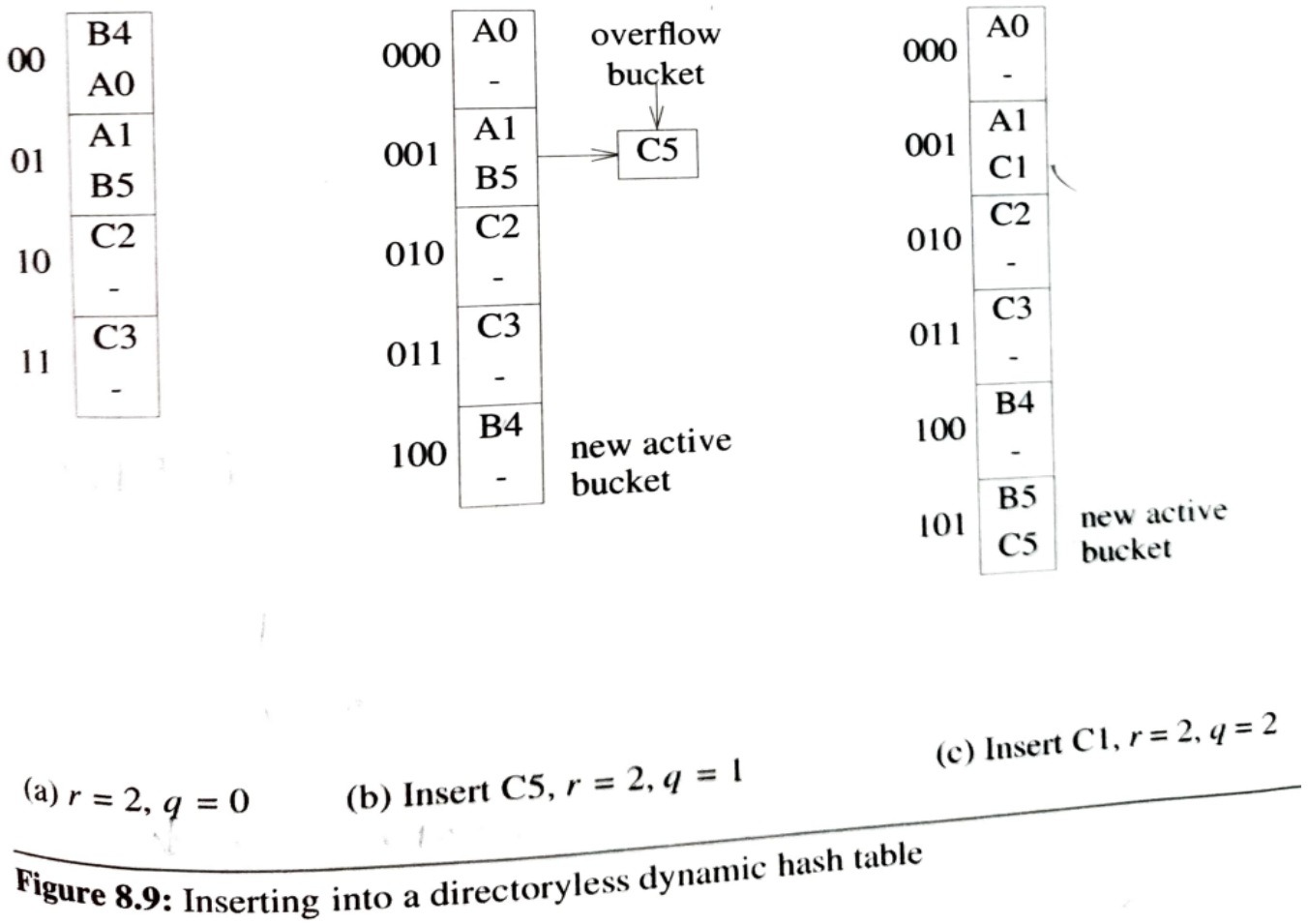


Figure 8.9: Inserting into a directoryless dynamic hash table

In Figure 8.9 (b), r = 2 and q = 1; h (k, 3) has been used for chains 000 and 100 while h (k, 2) has been used for chains 001, 010, and Chain 001 has an overflow bucket; the capacity of an overflow bucket may or may not be the same as that of an active bucket.

To search for k, we first compute h (k,r). If h (k,r) < q, then k, if present, is in a chain indexed using h (k, r+ 1). Otherwise, the chain to examine is given by h (k,r). Program 8.5 gives the algorithm to search a directoryless dynamic hash table.

```
if (h(k,r) < q) search the chain that begins at bucket h(k,r+1);
else search the chain that begins at bucket h(k,r);
```

Program 8.5: Searching a directoryless hash table

To insert C5 table into the table of Figure 8.9 (a), we use the search algorithm of program to determine whether or not C5 is in the table already. Chain 01 is examined and we verity that C5 is not present. Since the active bucket for the searched chain is full, we get an overflow. An overflow is handled by activating bucket $2^r + q$; reallocating the entries in the chain q between q and the newly activated bucket (or chain) $2^r + q$, and incrementing q by 1.

The bucket 4 = 100 is activated and the entries in chain 00 (q = 0) are rehashed using r + 1 = 3 bits. B4 hashes to the new bucket 100 and A0 to bucket 000. Now q = 1 and r = 2. A search for C5 would examine chain 1 and so C5 is added to this chain using an overflow bucket (see Figure 8.9 (b)).

Let us now insert C1 into the table of Figure 8.9 (b). Since, h (C1,2) = 01 = q, chain 01 = 1 is examined by our search algorithm (Program 8.5). The search verifies that C1 is not in the dictionary. Since the active bucket 01 is full, we get an overflow. we activate bucket 2^r +q = 5 = 101 and rehash the keys **A1, B5**, and **C5** that are in chain q. The rehashing is done using 3 bits. Al is hashed into bucket 001 while B5 and C5 hash into bucket 101. q is incremented by 1 and the new key C1 is inserted into bucket 001. Figure 8.9 (c) shows the result.

OPTIMAL BINARY SEARCH TREES

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion.

If probabilities of searching for elements of a set are known e.g., from accumulated data about past searches it is natural to pose a question about an optimal binary search tree for which the average number of comparisons in a search is the smallest possible.

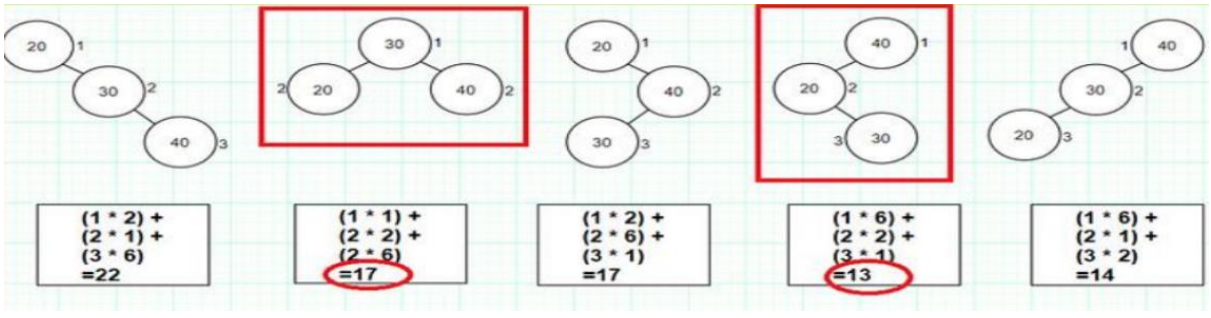
In many applications the cost of searching is very important. So, it is required that the overall cost of searching should be as less as possible. And we know that search time of BST is more than the Balanced Binary Search Tree, as Balanced Binary Search tree has less number of levels than the BST. And there is one way which can further reduce the cost than the Balanced BST, which is Optimal Binary Search Tree. Let us understand by following example

Key	20	30	40
Frequencies	2	1	6

As there are 3 different keys, so we can have total 5 various BST by changing order of keys. The total number of binary search trees with n keys is equal to

$$c(n) = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1,$$

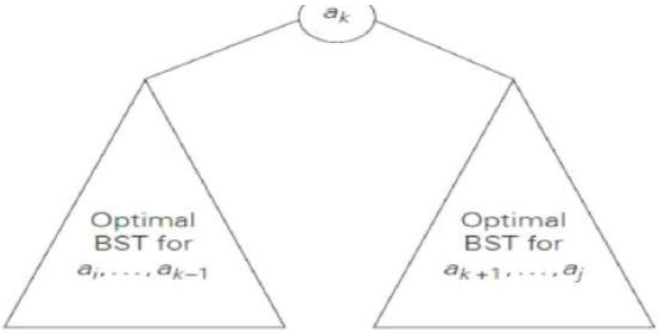
So following are the various possible BST of the above data. And also the overall cost for searching for each BST. The cost of computed by multiplying each node's frequency with the level of tree (Here we are assuming that the tree starts from level 1) and then add them to compute the overall cost of BST



As it is shown in above figure that 2nd BST is balanced and the 4th BST is not balanced, though it's cost is less than the cost of Balanced BST and its cost is the least among all, so it is our Optimal Binary Search Tree for the given data.

As a general algorithm, this exhaustive-search approach is unrealistic as n increases: Therefore dynamic programming approach, we will find values of C(i, j) for all smaller instances of the problem, although we are interested just in C(1, n).

To derive a recurrence underlying a dynamic programming algorithm, we will consider all possible ways to choose a root a_k among the keys $a_i, \ldots, a_j$. For such a binary search tree , the root contains key a_k, the left subtree T_i^{k-1} contains keys $a_i, \ldots, a_{k-1}$ optimally arranged, and the right subtree T^j_{k+1} contain s keys $a_{k+1}, \ldots, a_j$ also optimally arranged.



Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^{j}.

If we count tree levels starting with 1 to make the comparison numbers equal the keys' levels, the following recurrence relation is obtained:

General formula for calculating the minimum cost is:

C[i,j] = min{c[i, k-1] + c[k,j]} + w(i,j). if i<j

i<k<=j

= Pi if i=j

= 0 i>j

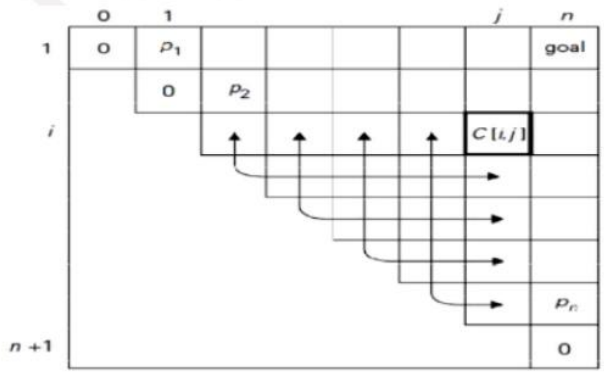


FIGURE 8.9 Table of the dynamic programming algorithm for constructing an optimal binary search tree.

The algorithm computes C(1, n)—the average number of comparisons for successful searches in the optimal binary tree.

Example : Find Optimal BST for the given data.

		a1	a2	a3	a4
Keys		10	15	20	25
p		3	3	1	1
q	2	3	1	1	1

p: successful search
q: unsuccessful search.

Soln:

Step1: Fill w_{00} — w_{44} with q_0 — q_4.
Fill c_{00} — c_{44} with 0
Fill R_{00} — R_{44} with 0.

Step2: Fill all weights using
$w[i,j] = w[i,j-1] + p_j + q_j$

	0	1	2	3	4
j-i=0	0	$w_{00}=2$ $c_{00}=0$ $R_{00}=0$	$w_{11}=3$ $c_{11}=0$ $R_{11}=0$	$w_{22}=1$ $c_{22}=0$ $R_{22}=0$	$w_{33}=1$ $c_{33}=0$ $R_{33}=0$	$w_{44}=1$ $c_{44}=0$ $R_{44}=0$.
j-i=1	1	$w_{01}=8$ $c_{01}=8$ $R_{01}=1$	$w_{12}=7$ $c_{12}=7$ $R_{12}=2$	$w_{23}=3$ $c_{23}=3$ $R_{23}=3$	$w_{34}=3$ $c_{34}=3$ $R_{34}=4$	
j-i=2	2	$w_{02}=12$ $c_{02}=19$ $R_{02}=1$	$w_{13}=9$ $c_{13}=12$ $R_{13}=2$.	$w_{24}=5$ $c_{24}=8$ $R_{24}=3$		
j-i=3	3	$w_{03}=14$ $c_{03}=25$ $R_{03}=2$.	$w_{14}=11$ $c_{14}=19$ $R_{14}=2$.			
j-i=4	4	$w_{04}=16$ $c_{04}=32$ $R_{04}=2$.				

Note: i j

j-i=0 c[0,0] c[1,1] c[2,2] c[3,3] c[4,4

j-i=1 c[0,1] c[1,2] c[2,3] c[3,4] etc

$w_{01} = w_{00} + p_1 + q_1 = 2+3+3 = 8$

$w_{12} = w_{11} + p_2 + q_2 = 3+3+1 = 7$

$w_{23} = w_{22} + p_3 + q_3 = 1+1+1 = 3$

$w_{34} = w_{33} + p_4 + q_4 = 1+1+1 = 3$.

$c_{01} = w_{01} = 8$ $\quad$ $R_{01} = 1$

$c_{12} = w_{12} = 7$ $\quad$ $R_{12} = 2$

$c_{23} = w_{23} = 3$ $\quad$ $R_{23} = 3$

$c_{34} = w_{34}\ 3$ $\quad$ $R_{34} = 4$.

Step3 $\quad j - i = 2$.

$w_{02} = w_{01} + p_2 + q_2 = 8+3+1 = 12$

$w_{13} = w_{12} + p_3 + q_3 = 7+1+1 = 9$.

$w_{24} = w_{23} + p_4 + q_4 = 3+1+1 = 5$

$$c[0,2] = \min \left\{ \begin{matrix} c[0,0] + c[1,2] \\ c[0,1] + c[2,2] \end{matrix} \right\} + w[0,2].$$

$$= \min \left\{ \begin{matrix} 0+7 \\ 8+0 \end{matrix} \right\} \quad 7+12 = 19.$$

Root

$$c[1,3] = \min \left\{ \begin{matrix} c[1,1] + [2,3] \\ c[1,2] + [3,3] \end{matrix} \right\} + w[1,3]$$

$$= \min \left\{ \begin{matrix} 0+3 \\ 7+0 \end{matrix} \right\} + 9 = 12.$$

Root

$$c[2,4] = \min \left\{ \begin{matrix} c[2,2] + c[3,4] \\ c[2,3] + c[4,4] \end{matrix} \right\} + w[2,4]$$

$$= \min \left\{ \begin{matrix} 0+3 \\ 3+0. \end{matrix} \right\} + 5 = 8$$

Root

Step 4 : $j - i = 3$.

$w_{03} = w_{02} + p_3 + q_3 = 12 + 1 + 1 = 14$

$w_{14} = w_{13} + p_4 + q_4 = 9 + 1 + 1 = 11$

$$c[0,3] = \min \begin{cases} c[0,0] + c[1,3] \\ c[0,1] + c[2,3] \\ c[0,2] + c[3,3] \end{cases} + w_{03}$$

Root

$$= \min \begin{cases} 0 + 12 \\ 8 + 3 \\ 19 + 0 \end{cases} + 14 = 25$$

$$c[1,4] = \min \begin{cases} c[1,1] + c[2,4] \\ c[1,2] + c[3,4] \\ c[1,3] + c[4,4] \end{cases} + w[1,4]$$

Root.

$$= \min \begin{cases} 0 + 8 \\ 7 + 3 \\ 12 + 0 \end{cases} + 11 = 19$$

Step 5 : $j - i = 4$

$w[0\,4] = w_{03} + p_4 + q_4 = 14 + 1 + 1 = 16$.

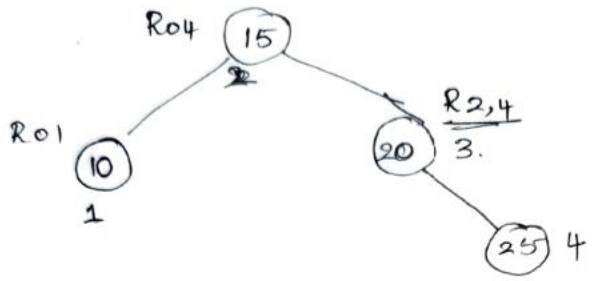
$$c[0,4] = \min \begin{cases} c[0,0] + c[1,4] \\ c[0,1] + c[2,4] \\ c[0,2] + c[3,4] \\ c[0,3] + c[4,4] \end{cases} + w_{04}$$

root.

$$= \min \begin{cases} 0 + 19 \\ 8 + 8 \\ 19 + 3 \\ 25 + 0 \end{cases} + 16 = 32$$

Construction of Optimal BST from table.

Root is $R_{04} \Rightarrow$ 2nd key = 15



MODULE-4
TREES

TOPICS:-Terminology, Binary Trees, Properties of Binary trees, Array and linked Representation of Binary Trees, Binary Tree Traversals - Inorder, postorder, preorder; Additional Binary tree operations. Threaded binary trees, Binary Search Trees – Definition, Insertion, Deletion, Traversal, Searching, Application of Trees-Evaluation of Expression.
Text 1: Ch 5: 5.1 –5.5, 5.7
Text 2: Ch 7: 7.1 – 7.9

This chapter discuss about the tree data structure in detail:-

4.1 DEFINITION

Tree is a finite set of one or more nodes such that

1) There is a specially designated node called root.

2) Remaining nodes are partitioned into disjoint sets T1, T2. . . Tn where each of these are called subtrees of root. (As shown in below figure1)

(Or)

A **Tree** is a set of nodes that either: is empty or has a designated node, called the root, from which hierarchically descend zero or more subtrees, which are also trees. If a tree is not empty, the first node is called the root .The indegreee of root node is zero. With the exception of the root, all the nodes in the tree must have an indegreee exactly one and the out degree of zero, one, or more.

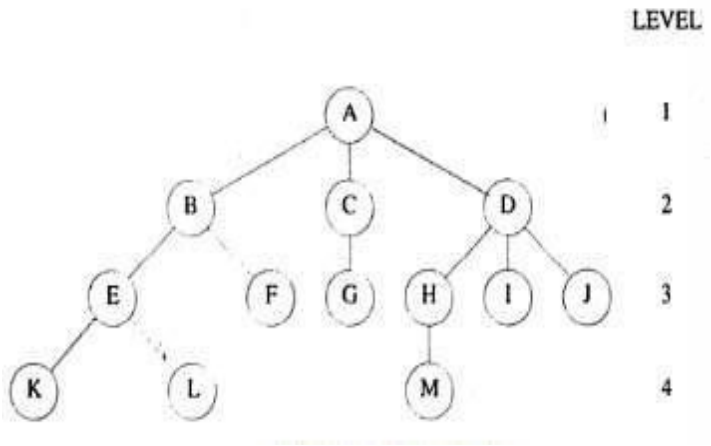


Figure 1: Sample Tree

A tree consists of a finite set of elements called **nodes** and a finite set of directed lines called **branches** that connect the nodes. The number of branches associated with a node is the degree of the node. When the branch is directed toward the node, it is indegree branch. When the branch is directed away from the node it is an outdegree branch.

4.2 TERMINOLOGIES OF TREES

- **Node:** contains item of information & links to other nodes.
- **Root:** A node with an indegree zero i.e. a node with no parent is called root; A non-empty tree has exactly one Root.

- **Degree:** Number of subtrees of a node. For e.g., degree of A=3; degree of C=1
- **Degree of a tree** is the maximum of the degree of the nodes in the tree.

  Degree of given tree=3.
- **Leaf:** Any nodes with an outdegree of zero i.e. a node with no successors or children's. For e.g., K, L, F, G, M, I, J
- **Internal Node:** A node that is not a root or a leaf is known as an internal node. For e.g., B, E, F, C, H, I, J
- **Parent and Child:** The subtrees of a node A are the children of A. A is the parent of its children (OR) a node is a parent if it has successor nodes i.e. if it has an outdegree greater than zero. Conversely, a node with predecessor is a child. For e.g., children of D are H, I and J. Parent of D is A.
- **Siblings/Brothers/Sisters:** Children of same parent are called siblings. For e.g., H, I and J are siblings.
- **Ancestor:** An ancestor is any node in the path from the root to the node. For e.g., ancestors of M are A, D and H.
- **Descendent:** A descendent is any node in the path below the parent node i.e. all nodes in the paths from a given node to a leaf are descendents of that node.
- **Path:** a sequence of nodes in which each node is adjacent to the next one. Every node in the tree can be reached by following a unique path starting from the root. the length of a path is the number of edges in the path, or 1 less than the number of nodes in it
- **Level:** the level of a node is its distance from the root. If a node is at level 'l', then its children are at level 'l+1'.
- **Height or depth of a tree** is defined as maximum level of any node in the tree. For e.g., Height of given tree = 4.
- A **Sub tree** is any connected structure below the root.
- A **tree** is a set of nodes that either: Is empty , or Has a designated node, called the root, from which hierarchically descend zero or more subtrees, which are also trees.

REPRESENTATION OF TREES

The first is the General Tree organization chart format, which is basically the notation used to represent in figure 2. This notation is called general tree representation.

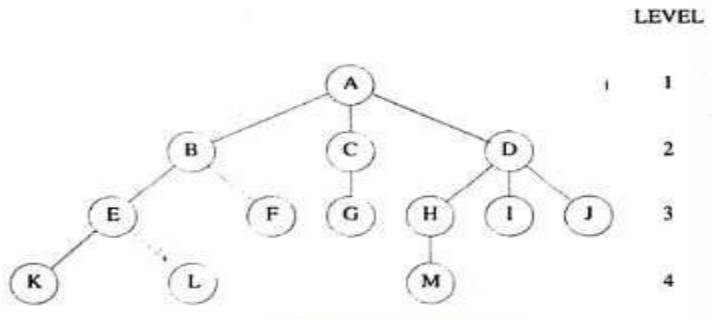


Figure 2: General tree

There are **three different user representations for trees.**

1) List representation
2) Left-child right-sibling representation
3) Degree-two tree representation (Binary Tree)

1) List representation

This user format is the **parenthetical listing**. This format is used with algebraic expressions. When a tree is represented in parenthetical notation, each **open parenthesis indicates the start of a new level, each closing parenthesis completes the current level, and each closing parenthesis completes the current level and moves up one level in the tree.**

Consider the tree shown in the figure 2. Its parenthetical notation is The tree can be drawn as a list:

(A(B(E(K,L),F),C(G),D(H(M),I,J))).

For a tree of degree 'k', we can use the node-structure as shown(Fig. 3).

Figure 3: Node structure with degree k

The information in the root node comes first, followed by a list of subtrees of that node. Each tree-node can be represented by a memory-node that has fields for data & pointers to children of tree-node (Figure. 4).

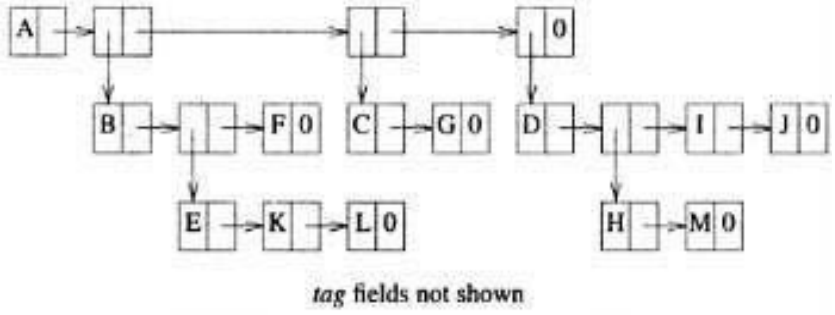


Figure 4: List Representation of Tree

Property of general tree: if T is a k-ary tree with n nodes, each having a fixed size as in figure 3, then n(k-1)+1 of the nk child fields are 0,n>=1.

2) Left child-right sibling representation

Figure 5 shows the node-structure used in left child-right sibling representation.

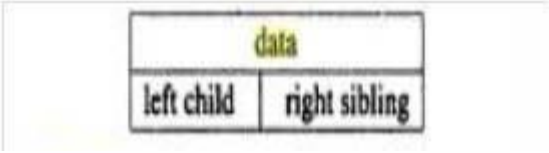


Figure 5 : Left Child-Right Sibling Node

To convert the tree of figure 2 into this representation, every node must note it has at most one leftmost child and at most one closest right sibling.

For example, the leftmost child of A is B, and the leftmost child of D is H. The closest right sibling of B is C, and the closest right sibling of H is I. The Left-child field of each node points to its leftmost child (if any), and right-sibling field points to the closest right sibling (if any). Figure 6 shows the left child right sibling representation.

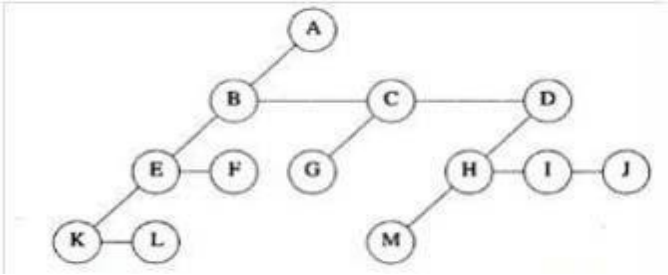


Figure 6: Left Child Right Sibling Representation.

3) Degree-two tree representation

To obtain the degree two tree representation of a tree, simply rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees. This gives the degree two tree displayed in figure 7, In this representation, it refer to 2 children of a node as left & right children. Left child-right child trees are also known as binary trees.

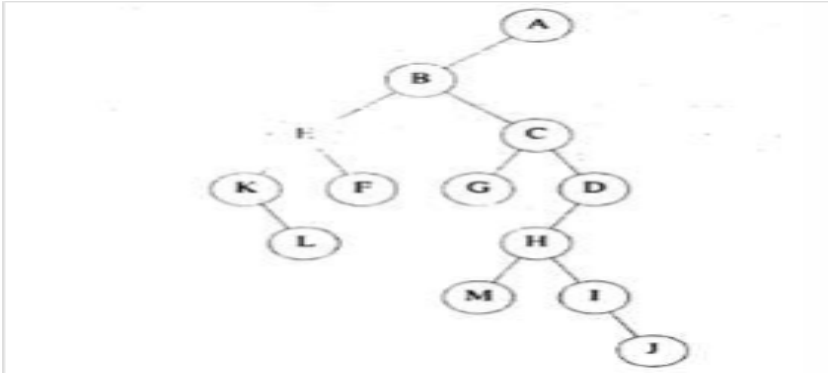


Figure 7: Left Child-Right Child Representation

Figure 8 shows the two additional examples of trees represented as left child-right sibling trees and as left child-right child trees. Left child-right child trees are also known as binary trees.

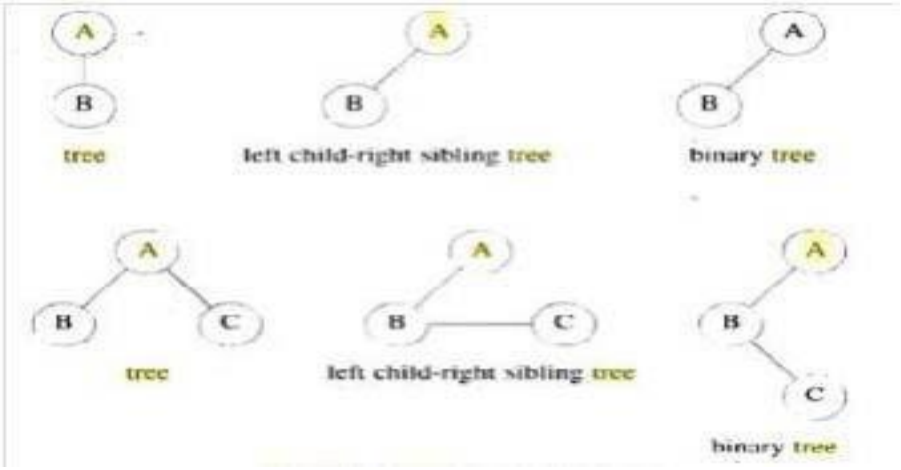


Figure 8: Binary trees

4.3 BINARY TREES

A **Binary tree** is a finite set of elements that is either empty or is partitioned into two disjoint subsets. The **first subset** contains a single element called the **root of the tree**. The **other two subsets** are themselves binary trees, called the **left and right subtrees of the original tree**. A left or right sub tree can be empty. Each element of a binary tree is called a node of the tree as illustrated in figure 8.

(OR)

A **Binary tree** is a tree in which no node can have more than two subtrees. The maximum outdegree for a node is two. In other words, a node can have zero, one or two subtrees. These subtrees are designated as the left sub tree and right sub tree.

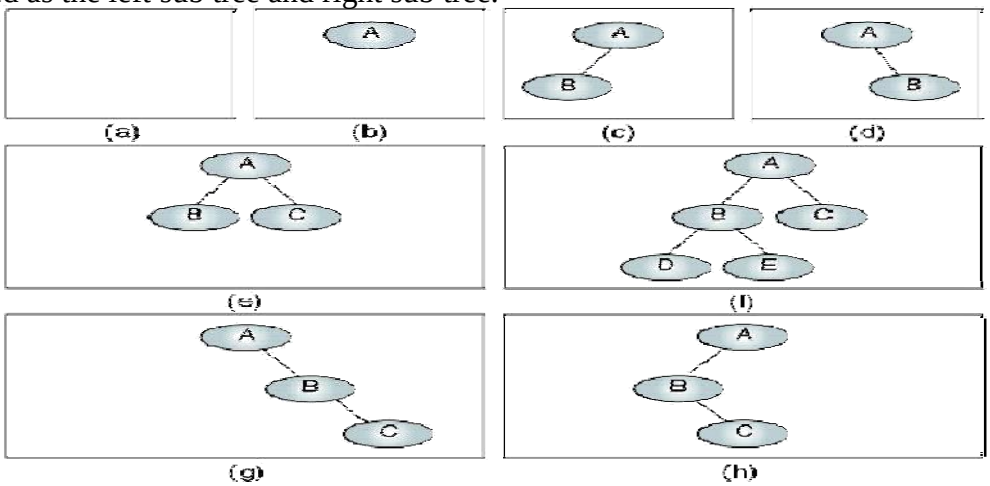


Figure 9: Various binary trees

Consider the below binary tree as an example:

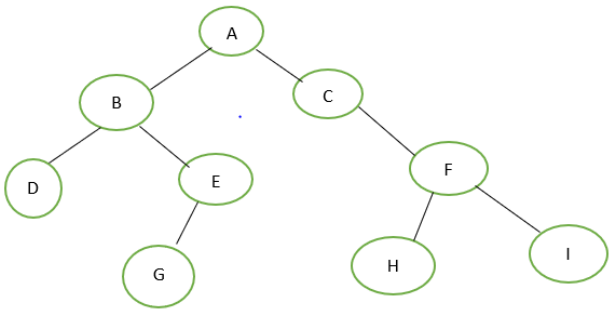


Figure 9 (a): Binary tree

If A is the **root** of a binary tree and B is the root of its left or right sub tree, then A is said to be the father of B and B is said to be the left or right son of A and collection of binary shown in figure 9(a).A node that has no sons is called a **leaf**. Node n1 is an ancestor of node n2 (and n2 is a descendant of n 1) if n1 is either the father of n2 or the father of some ancestor of n2. node n2 is a left descendant of node n1 if n2 is either the left son of n1 or a descendant of the left son of n1.A right descendant may be similarly defined. Two nodes are brothers if they are left and right sons of the same father.

ADT of Binary Tree

```
structure Binary_Tree(abbreviated BinTree) is
objects: a finite set of nodes either empty or consisting of a root node, left Binary_Tree,
and right Binary_Tree.
functions:
for all bt, bt1, bt2 ∈ BinTree, item ∈ element
Bintree Create()::= creates an empty binary tree
Boolean IsEmpty(bt)::= if (bt==empty binary tree)
                           return TRUE
                       else
                           return FALSE
BinTree MakeBT(bt1, item, bt2)::= return a binary tree whose left subtree is bt1, whose
                                  right subtree is bt2, and whose root node contains the data item
Bintree Lchild(bt)::= if (IsEmpty(bt))
                          return error
                      else
                          return the left subtree of bt
element Data(bt)::= if (IsEmpty(bt))
                        return error
                    else
                        return the data in the root node of bt
Bintree Rchild(bt)::= if (IsEmpty(bt))
                          return error
                      else
                          return the right subtree of bt
```

Differentiate between a binary tree and a tree.

1. There is no tree having zero nodes, but there is an empty binary tree.
2. In a binary tree, we distinguish between the order of the children while in a tree we do not.

TYPES OF BINARY TREE

1) **Skewed tree** is a tree consisting of only left sub tree or only right sub tree (Figure 10a).

2) **Full binary tree** is a binary tree of depth k having 2^k-1 nodes, k>=1 (Figure 11).

3) **Complete tree** is a binary tree in which every level except possibly last level is completely filled and all nodes are as far left as possible. A binary tree with n nodes & depth k is complete iff its nodes correspond to nodes numbered from 1 to n in full binary tree of depth k (Figure 10b).

4) If every non leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a **strictly binary tree**. A strictly binary tree with n leaves always contains 2n-1 nodes as illustrated in figure 12.

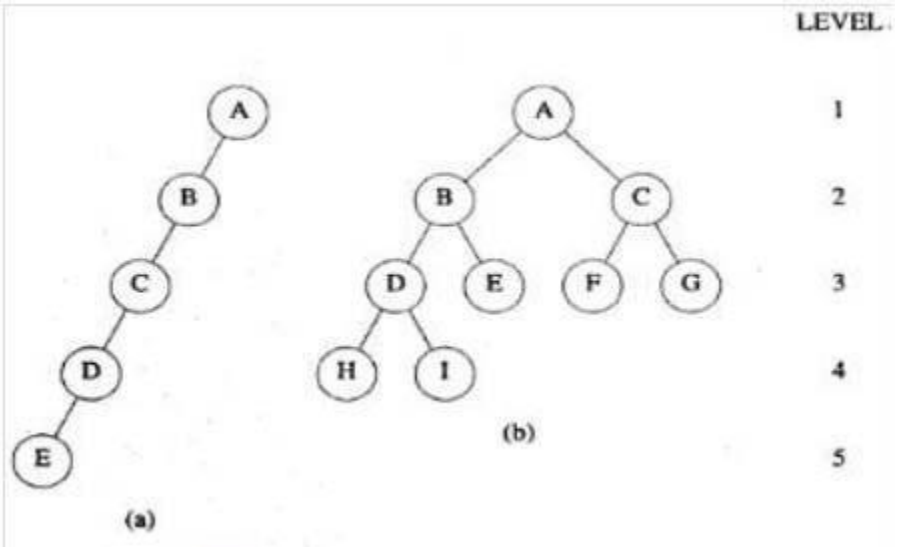


Figure 10(a): **skewed tree** 10(b).**complete binary tree**

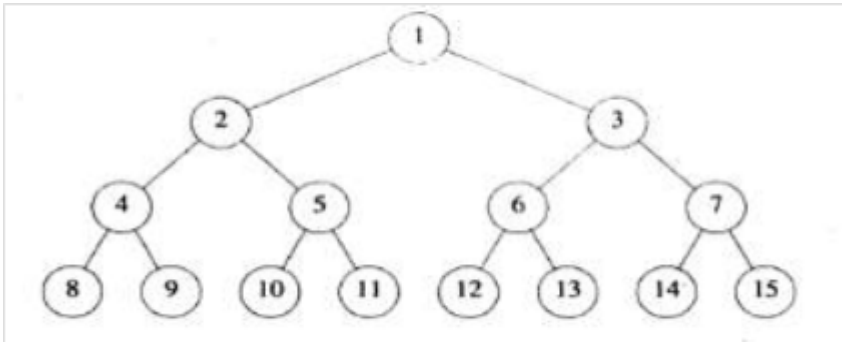


Figure 11:**Full binary tree representation of depth 4 with sequential node numbering**

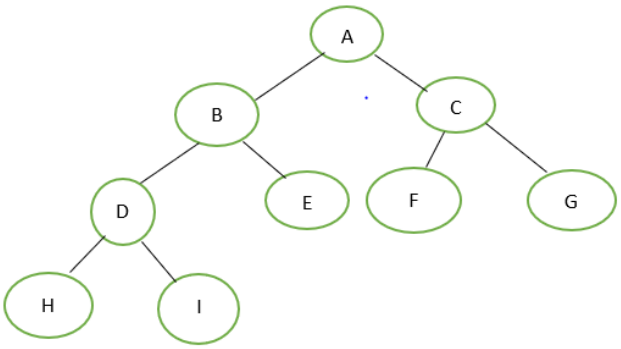


Figure 12: Strictly binary tree

4.4 PROPERTIES OF BINARY TREES

- The **maximum number of nodes on level 'i'** of a binary tree is **2^{i-1}, i >=1.**

(For e.g. maximum number of nodes on level 4=$2^{4-1}=2^3=8$).

- The **maximum number of nodes in a binary tree of depth 'k'** is **2^k-1, k>=1.**

(For e.g. maximum number of nodes with depth 4=$2^4-1=16-1=15$).

- **Relation between number of leaf nodes and degree-2 nodes** : For any non-empty binary tree 'T', if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then **$n_0=n_2+1$.**

- **A full binary tree of depth k is a binary tree of depth k having 2^k-1 nodes, k>=0.**

- **A binary tree with n nodes and depth k is complete** iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k.

- **Minimum number of nodes binary trees:** Given a height of the binary tree, H, the minimum number of nodes in the tree is equal to height of tree.

- **Height of Binary Trees:** If the given binary tree has N nodes in a binary tree, the maximum height is given by Hmax=N. If the given binary tree has N nodes in a binary tree, the minimum Height Is Given By Hmin=Log2n+1

- **Level of binary trees:** The level of a node in a binary tree is defined as follows. The root has level 0, and the level of any other node in the tree is one more than the level of its father.

- **Depth of binary trees:** The depth or Height of a binary tree is the maximum level of any leaf in the tree. By definition the height of any empty tree is -1. This equals the length of the longest path from the root to any leaf.

4.5 BINARY TREE REPRESENTATIONS

A binary tree can be represented in two forms, namely:

1) Array Representation

2) Linked Representation

1. ARRAY REPRESENTATION

A one-dimensional array can be used to store nodes of binary tree (Figure 13).

- If a complete binary tree with 'n' nodes is represented sequentially, then for any node with index i (1<=i<=n) , we have

1) Parent (i) is **at [i/2] if i!=1.** If **i=1**, i is the **root and has no parent**.
2) Left Child (i) is at **2i , if 2i<=n**. If **2i>n**, then i has **no left child**.
3) Right Child (i) is at **2i+1<=n**. If **2i+1>n**, then i has **no right child.**

Consider the tree shown below

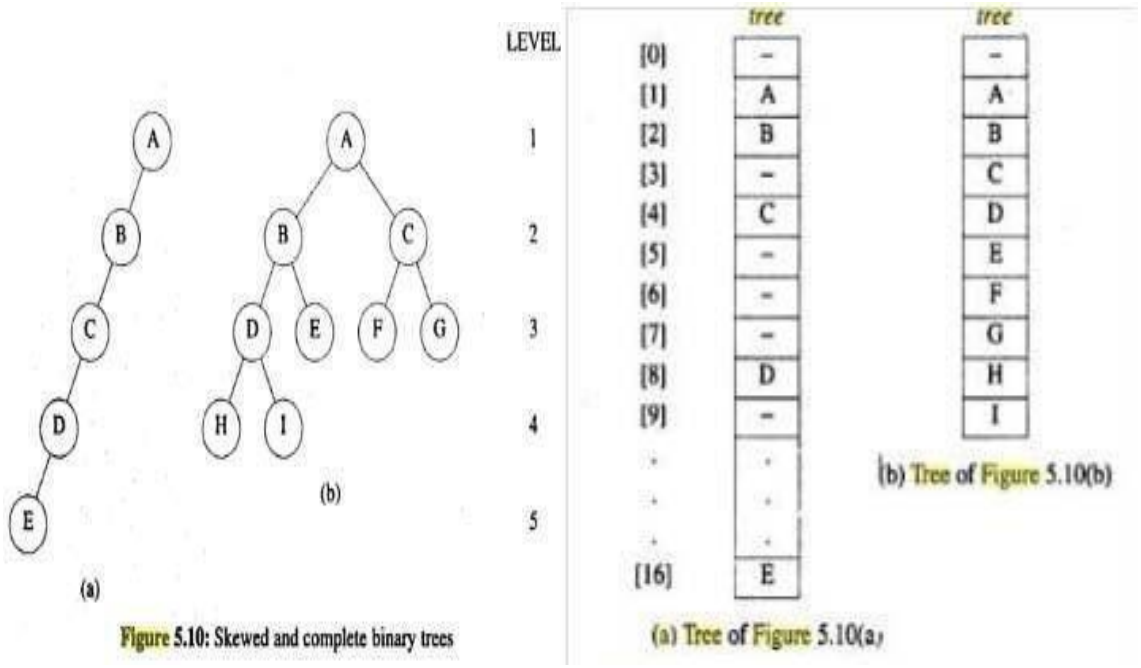


Figure 13: Array Representation of binary tree

Advantage: For complete binary tree, array representation is ideal, as no space is wasted.

Disadvantage: For skewed tree, less than half the array is utilized. In the worst case, a skewed tree of depth k will require 2k-1 spaces. Of these, only k will be used.

2. LINKED REPRESENTATION

• **Shortcoming of array representation**: Insertion and deletion of nodes from middle of a tree requires movement of potentially many nodes to reflect the change in level number of these nodes. These problems can be overcome easily through the use of a linked representation (Figure 15).

• Each **node** has three fields:

1) Left Child,
2) Data and
3) Right Child (Figure 14) and it is defined in C as shown below:

```
typedef struct node *treePointer;
typedef struct
{
        int data;
        treePointer leftChild,rightChild;
}node;
```

Root of tree is stored in the data member 'root' of Tree. This data member serves as access-pointer to the tree.

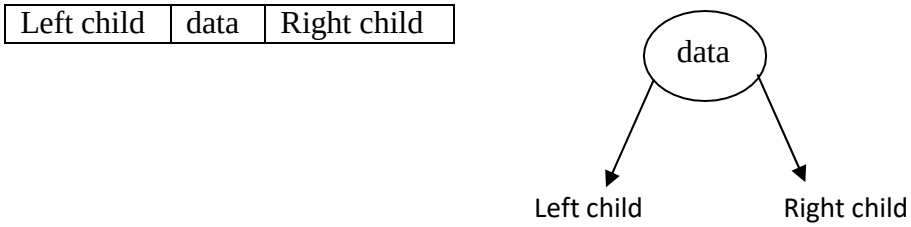


Figure 14: Node representation

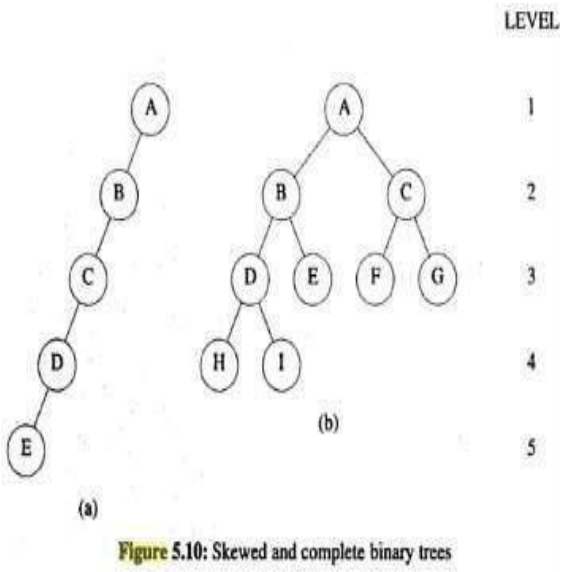


Figure 5.10: Skewed and complete binary trees

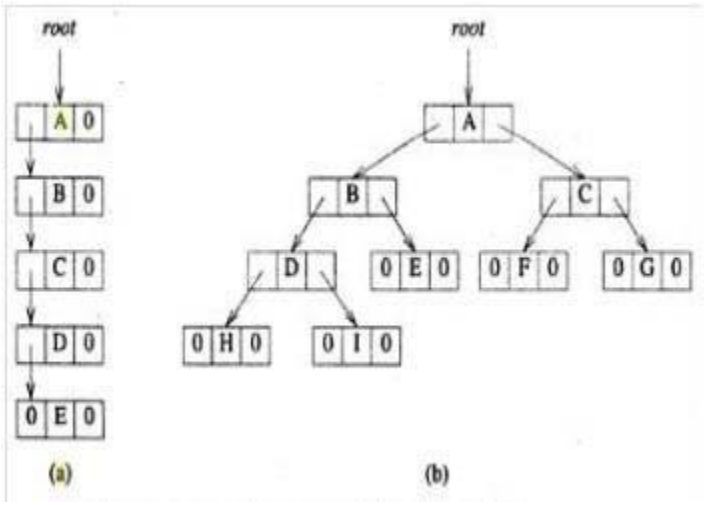


Figure 15: Linked representation of binary tree

4.6 BINARY TREE TRAVERSALS

The order in which the nodes of a linear list are visited in a traversal is clearly from first to last. However, there is no such "natural" linear order for the nodes of a tree. Thus, different orderings are used for traversal in different cases.

Let L, V, and R stand for moving left, visiting the node, and moving right. There are six possible combinations of traversal. LVR, LRV, VLR, VRL, RVL, RLV. Adopt convention that we traverse left before right, only 3 traversals remain LVR, LRV, VLR and in fact they are inorder, postorder and preorder approaches are explained in figure 16.

Tree Traversal Approaches

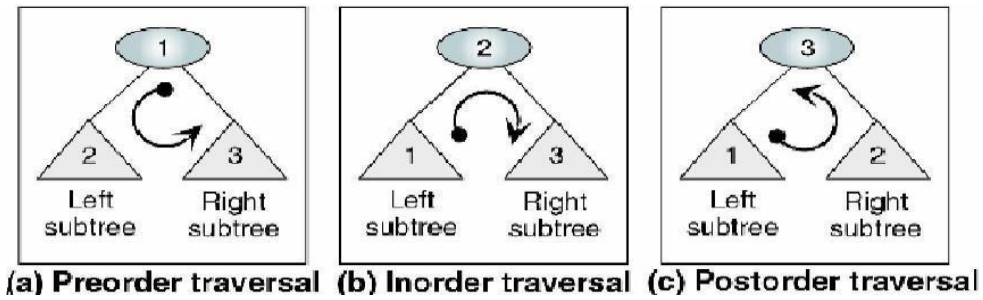


Figure 16: Tree traversal methods

There are 3 techniques, namely:

1) In order traversal (LVR)

2) Preorder traversal (VLR)

3) Post order traversal (LRV).

(Let L=moving left, V= visiting node and R=moving right).

Consider a binary tree with arithmetic expressions as shown in figure 17 and perform traversals

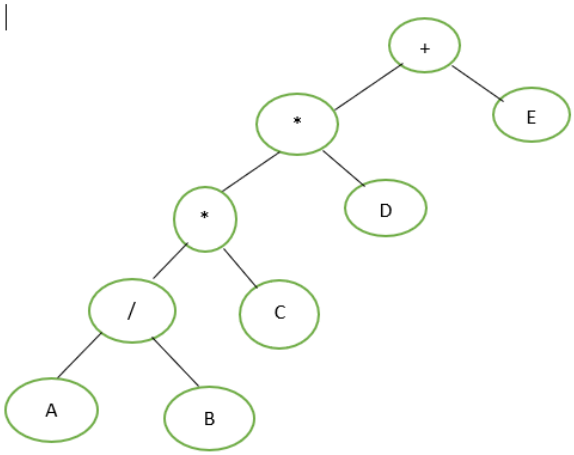


Figure 17: Binary tree with arithmetic expression

1. INORDER TRAVERSAL

Inorder traversal calls for moving down tree toward left until you can go no farther. Then, you "visit" the node, move one node to the right and continue. If you cannot move to the right, go back one more node.

```
void inorder(tree_pointer ptr)
{       /* inorder tree traversal */
        if (ptr)
        {
                inorder(ptr->left_child);
                printf("%d", ptr->data);
                indorder(ptr->right_child);
        }
}
```

Figure 18: Inorder traversal

The steps for traversing a binary tree **in inorder traversal** are:

1. **Visit the left subtree, using inorder.**
2. **Visit the root.**
3. **Visit the right subtree, using inorder.**

The inorder traversal tracing for the above given expression tree by using recursion is shown below:(figure 19)

Call of inorder	Value in Current Node	Action
Driver	+	
1	*	
2	*	
3	/	
4	A	
5	0	
4	A	Printf(A)
6	0	
3	/	Printf(/)
7	B	
8	0	
7	B	Printf(B)
9	0	
2	*	Printf(*)
10	C	
11	0	
10	C	Printf(C)
12	0	
1	*	Printf(*)
13	D	
14	0	
13	D	Printf(D)
15	0	
Driver	+	Printf(+)

16	E	
17	0	
16	E	Printf(E)
18	0	

Figure 19: Tracing of inorder traversal

Each step of the trace shows the call of inorder, the value in the root, and whether or not the printf function is invoked (Figure 19).Since there are 19 nodes in the tree, inorder() is invoked 19 times for the complete traversal. The nodes of figure 17 would be output in an inorder as **A/B*C*D+E**

2. PREORDER TRAVERSAL

Visit a node, traverse left, and continue (figure 20). When it is not possible to continue, move right and begin again or move back until you can move right and resume. The nodes of figure 17 would be output in preorder as **+**/ABCDE**

```
void preorder(tree_pointer ptr)
{       /* preorder tree traversal */
        if (ptr)
        {
                printf("%d", ptr->data);
                preorder(ptr->left_child);
                predorder(ptr->right_child);
         }
}
```

Figure 20: Preorder traversal

The steps for traversing a binary tree in **preorder traversal** are:

1. Visit the root.

2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

3. POSTORDER TRAVERSAL

Post order traversal calls for moving down tree toward left until you can go no farther. Then, move one node to the right and continue. Then when it is not possible to move to the right, go back one more node and visit the node. The nodes of figure 5.16 would be output in postorder as **AB/C*D*E+**

```
void postorder(tree_pointer ptr)
{
        if(ptr)
        {
                postorder(ptr->left_child);
                postorder(ptr->right_child);
                printf("%d",ptr->data);
        }
}
```

The steps for traversing a binary tree in **postorder traversal** are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

4. ITERATIVE INORDER TRAVERSAL

With respect to the figure 17, a node that has no action indicates that the node is added to the stack, while a node that has a printf action indicates that the node is removed from the stack (Figure 21). The left nodes are stacked until a null node is reached, the node is then removed from the stack, and the node's right child is stacked. The traversal continues with the left child. The traversal is complete when the stack is empty. The output of iterative inorder traversal for expression tree is given below as **A/B*C*D+E**

```
void iter_inorder(tree_pointer node)
{
        int top= -1;      /* initialize stack */
        tree_pointer stack[MAX_STACK_SIZE];
        for (;;)
        {
                for (; node; node=node->left_child)
                add(&top, node);            /* add to stack */
                node= delete(&top);
                  /* delete from stack */
                if (!node) break;           /* empty stack */
                        printf("%D", node->data);
                node = node->right_child;
        }
}
```

Figure 21: Inorder Iterative traversal

5. LEVEL-ORDER TRAVERSAL

This traversal uses a queue (Figure 22). We visit the root first, then the root's left child followed by the root's right child. We continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node. function addq( ) adds a tree node to queue and function deleteq( ) deletes the front tree node from the queue.

level order traversal begins by adding the root to the queue.the function operates by deleting the node at the front of the queue, printing out the node's data field, and adding the node's left and right children to the queue. since a node's children are at the next lower level, and we add the left child before the right child, the function prints out the nodes using the ordering scheme and thus level order traversal of arithmetic expression tree is **+*E*D/CAB**

```
void level_order(tree_pointer ptr)
{       /* level order tree traversal */
        int front = rear = 0;
        tree_pointer queue[MAX_QUEUE_SIZE];
        if (!ptr) return;           /* empty queue */
        addq(front, &rear, ptr);
        for (;;)
        {
                ptr = deleteq(&front, rear);
                if (ptr)
                {
                    printf("%d", ptr->data);
                    if (ptr->left_child)
                        addq(front, &rear, ptr->left_child);
                    if (ptr->right_child)
                        addq(front, &rear, ptr->right_child);
                }
                else
                    break;
        }
}
```

Figure 22: Level Order Traversal

4.7 THREADED BINARY TREES

Shortcoming of linked representation of binary tree: There may be more null links than actual pointers. **A.J.Perlis and C.Thornton** have devised a way to make use of these null links. **Solution:** This drawback can be overcome by replacing null links by pointers, called **threads**, to other nodes in the tree.

To construct the threads, we use the following **rules:**

1) If ptr->leftChild =NULL, we replace ptr->leftChild with a pointer to the node that would be visited before ptr in an inorder traversal. i.e. we replace the null link with a pointer to the inorder predecessor

of ptr (Figure 5.20).

2) If ptr->rightChild =NULL, we replace ptr->rightChild with a pointer to the node that would be visited after ptr in an inorder traversal. i.e. we replace the null link with a pointer to the inorder successor of ptr.

The node structure is given by following **C declarations**:

```
typedef struct threadedtree *threadedPointer;
typedef struct
{
        short int leftThread;
        threadedPointer leftChild;
        char data;
        threadedPointer rightChild;
        short int rightThread;
}threadedTree;
```

A empty threaded binary tree is represented by its header node as in figure 23

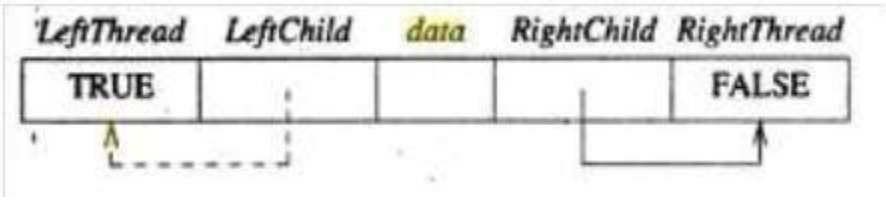


Figure 23: Empty node structure for threaded binary tree

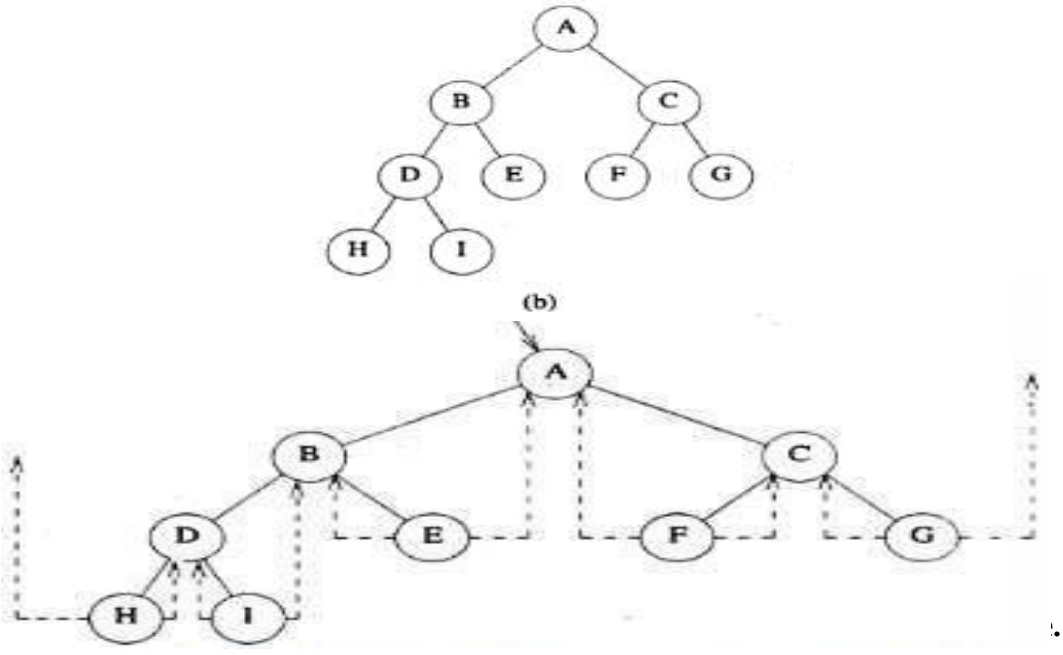


Figure 24 shows the binary tree with its new threads drawn in as broken lines. This tree has 9 nodes and 10 NULL links ,which have been replaced by threads. if we traverse the tree in inorder, the nodes will be visited in the order H,D,I,B,E,A,F,C,G. for example, node E has a predecessor thread

that points to B and a successor thread that points to A

When we represent the tree in memory, we must be able to distinguish between threads and normal pointers. This is done by adding two additional fields to the node structure, leftThread and rightThread (Figure 23).

Assume that ptr is an arbitrary node in a threaded tree. If ptr->leftThread=TRUE, then ptr->leftChild contains a thread; otherwise it contains a pointer to the left child (Fig 24). Similarly, if ptr->rightThread=TRUE, then ptr->rightChild contains a thread; otherwise it contains a pointer to the right child (Figure 24).

In figure two threads have been left dangling: one in the left child of H , the other in the right child of G. We handle the problem of the loose threads by having them point to the header node, root. The variable 'root' points to the header as shown in figure 24.

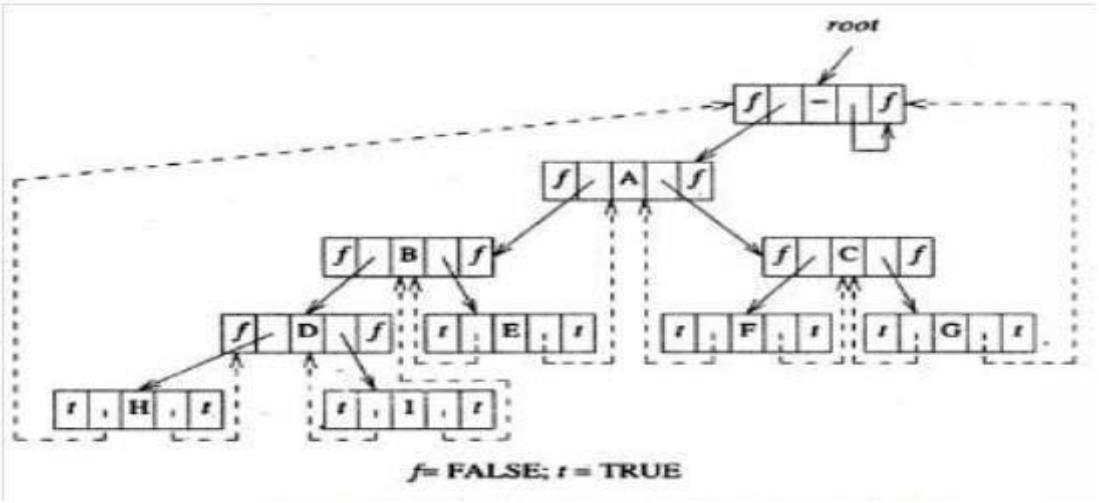


Figure 24: Memory representation of threaded binary tree

1. Inorder traversal of a threaded binary tree

```
threaded_pointer insucc(threaded_pointer tree)
{
        threaded_pointer temp;
        temp = tree->right_child;
        if (!tree->right_thread)
                while (!temp->left_thread)
                        temp = temp->left_child;
        return temp;
}
```

```
void tinorder(threaded_pointer tree)
{
        /* traverse the threaded binary tree inorder */
        threaded_pointer temp = tree;
        for (;;)
        {
            temp = insucc(temp);
            if (temp==tree) break;
                printf("%3c", temp->data);
        }
}
```

2. Inserting A Node Into A Threaded Binary Tree

Let new node 'r' be has to be inserted as the right child of a node 's' (Figure 25). The cases for insertion are 1) If s has an empty right subtree, then the insertion is simple and diagrammed in fig 25 a.

2) If the right subtree of s is not empty, then this right subtree is made the right subtree of r after insertion. When this is done, r becomes the inorder predecessor of a node that has a leftThread==true field, and consequently there is a thread which has to be updated to point to r. The node containing this thread was previously the inorder successor of s. Figure 25b illustrates the insertion for this case.

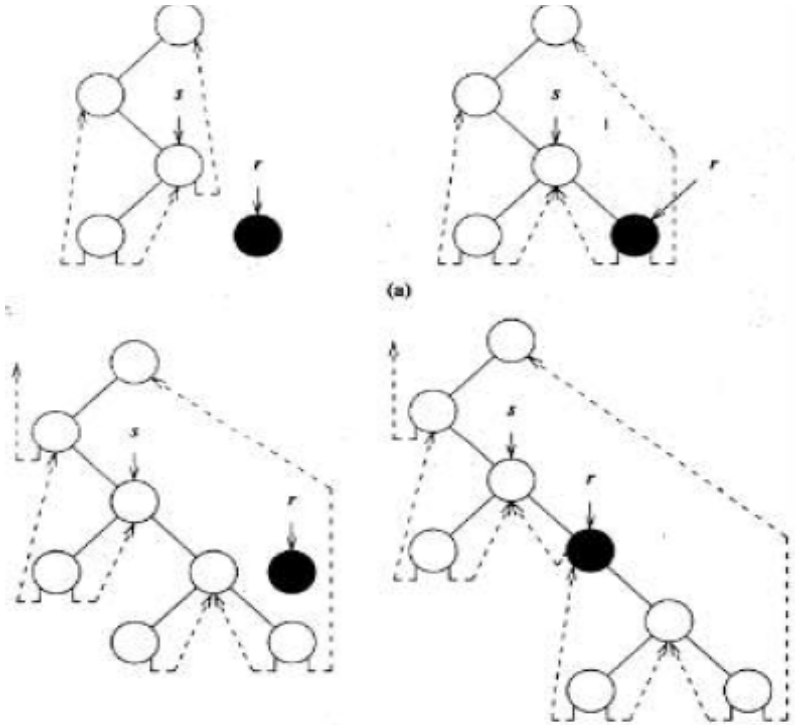


Figure 25: Insertion of node into threaded binary tree

Insertion of node into threaded binary tree:-

```
void insertRight(threadedPointer s,threadedPointer r)
        {
                r->rightChild=parent->rightChild;
                r->rightThread=parent->rightThread;
                r->leftChild=parent;
                r->rightThread=TRUE;
                s->rightChild= child;
                s->rightThread=FALSE;
                if(!r->rightThread)
                {
                        temp=insucc(r);
                        temp->leftChild=r;
                }
 }
```

4.8 BINARY SEARCH TREE (BST)

A binary search tree is a binary tree. it may be empty. if it is not empty then,it satisfies the following properties:

1) Each node has exactly one key and the keys in the tree are distinct (Figure 26).
2) The keys in the left subtree are smaller than the key in the root.
3) The keys in the right subtree are larger than the key in the root.
4) The left and right subtrees are also binary search trees.

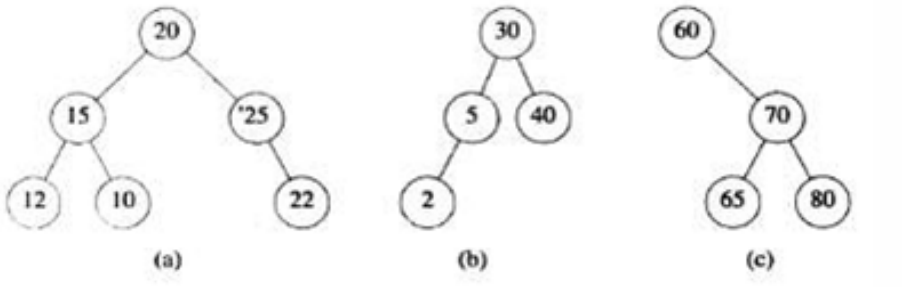


Figure 26: Binary search trees

Fig(a) is not a Binary search tree fig(b) and (c) are Binary search trees

1. Searching A Binary Search Tree

Assume that we have to search for a node whose key value is k. The search begins at the root (Program A and program B)

- If the root is NULL, then the tree contains no nodes and hence the search is unsuccessful.
- If the key value k matches with the root's data then search terminates successfully.
- If the key value is less than the root's data, then we should search in the left subtree.
- If the key value is greater than the root's data, then we should search in the right subtree.

Analysis: If h is the height of the binary search tree, then the search operation can be performed in O(h) time.

```
tree_pointer search(tree_pointer root, int key)
{
        /* return a pointer to the node that contains key. If there is no such
        node, return NULL */
        if (!root)
                return NULL;
        if (key == root->data)
                return root;
        if (key < root->data)
                return search(root->left_child,key);
        else
                search(root->right_child,key);
}
```

Program A: recursive search algorithm for BST

```
tree_pointer search2(tree_pointer tree, int key)
{
        while (tree)
        {
                if (key == tree->data)
                        return tree;
                if (key < tree->data)
                        tree = tree->left_child;
                else
                        tree = tree->right_child;
        }
        return NULL;
}
```

Program B: Iterative search algorithm for BST

2. Inserting into a binary search tree

- Firstly verify, if the tree already contains the node with the same data (Figure 27 &Program C).
- If the search is successful, then the new node cannot be inserted into the binary search tree.
- If the search is unsuccessful, then we can insert the new node at that point where the search terminated.

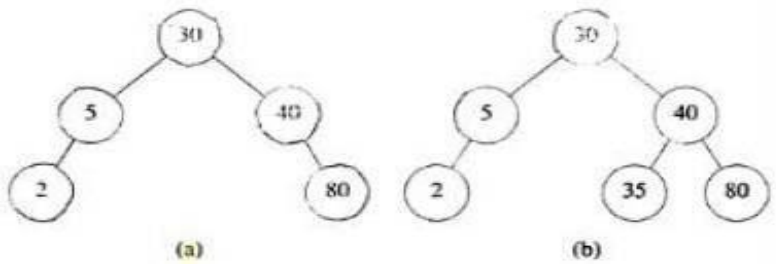


Figure 27: insertion of node 35 to existing BST

```
void insert_node(tree_pointer *node, int num)
{
        tree_pointer ptr,
        temp = modified_search(*node, num);
        if (temp || !(*node))
        {
                ptr = (tree_pointer) malloc(sizeof(node));
                if (IS_FULL(ptr))
                {
                        fprintf(stderr, "The memory is full\n");
                        exit(1);
                }
                ptr->data = num;
                ptr->left_child = ptr->right_child = NULL;
                if (*node)
                        if (num<temp->data)
                                temp->left_child=ptr;
                        else
                                temp->right_child = ptr;
                else
                        *node = ptr;
        }
}
```

Program C: Node insertion to BST

3. Deletion from a binary search tree

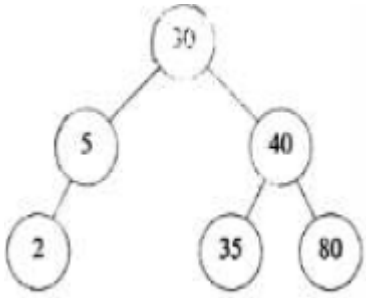


Figure 28: BST

- **Deletion of a leaf:** To delete 35 from the tree of figure 28, the left-child field of its parent is set to NULL.

- **Deletion of a non-leaf that has only one child:** The node containing the dictionary pair to be deleted is freed, and its single-child takes the place of the freed node. So, to delete the 5 from the tree in figure 29, we simply change the pointer from the parent node to the single-child node.

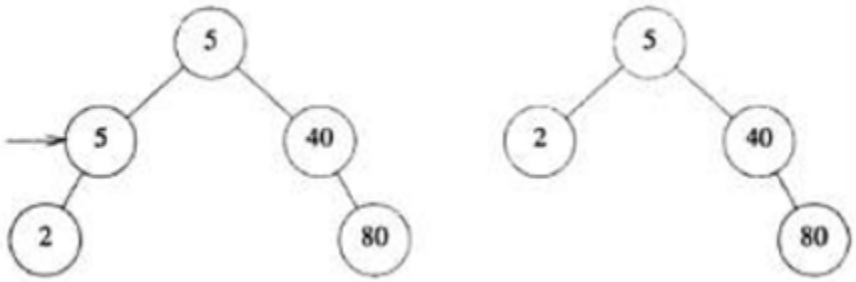


Figure 29(a): node 5 deletion (b): node 30 deletion

- **The pair to be deleted is in a non-leaf node that has two children:** The pair to be deleted is replaced by either the largest pair in its left subtree or the smallest one in its right subtree. For instance, if we wish to delete the pair with key 30 from the tree in figure 28, then we replace it by key 5 as shown in figure 29(b)

4. Joining and splitting binary tree

There are two types of join operation on a binary search tree:

1) **ThreeWayJoin(small,mid,big):** this creates a binary search tree consisting of the pairs initially in the BST small and big as well as pair mid. it is assumed that each key in small is smaller than mid.key and that each key in big is greater than mid.key. following the join , both small and big are empty.

2) **TwoWayJoin(small,big):** this joins the two binary search tree small and big to obtain a single BST that contains all the pairs originally in small and big. it is assumed that all keys of small are smaller

than all keys of big and that following the join both small and big are empty.

3)**split(theTree,k,small,big,mid)**

Splitting a binary search tree will produce three trees: small, mid and big.

- If key is equal to root->data, then root->llink is the small, root->data is mid & root->rlink is big.
- If key is lesser than the root->data, then the root's along with its right subtree would be in the big.
- if key is greater than root->data, then the root's along with its left subtree would be in the small**.**

4. 9 Application of Trees: EXPRESSION TREES

Expression tree is a binary tree, because all of the operations are binary. It is also possible for a node to have only one child, as is the case with the unary minus operator. The leaves of an expression tree are operands, such as constants or variable names, and the other (non leaf) nodes contain operators.

Once an expression tree is constructed we can traverse it in three ways:

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

Figure 30 shows some more expression trees that represent arithmetic expressions given in infix form.

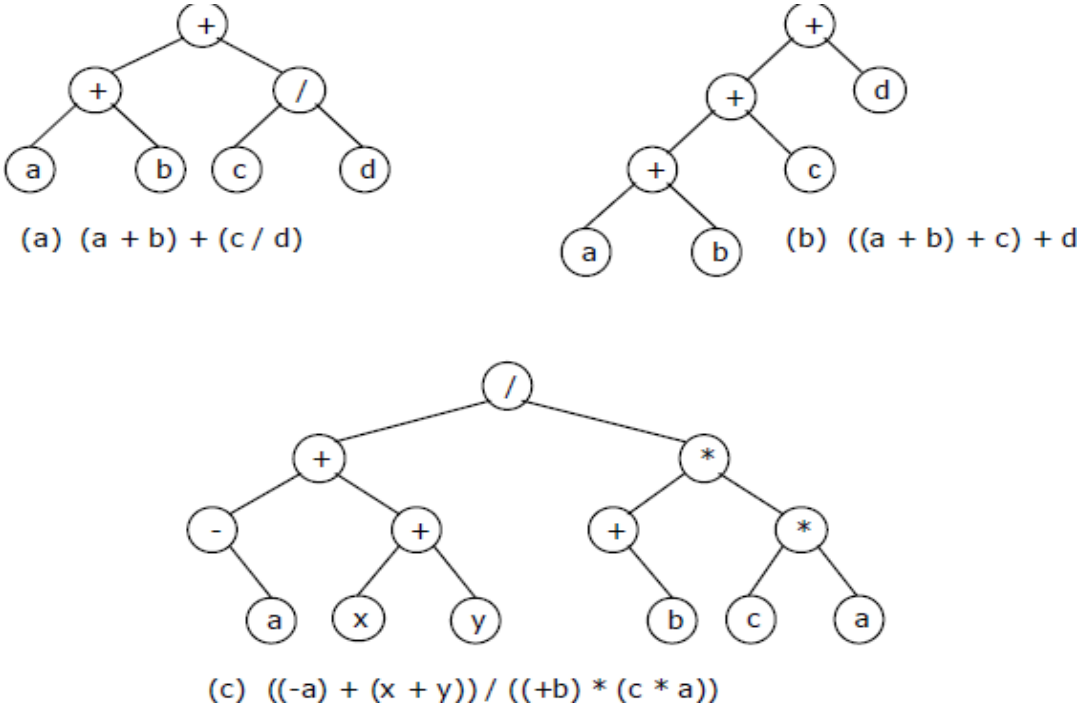


Figure 30: Expression trees

1. Construction of expression tree

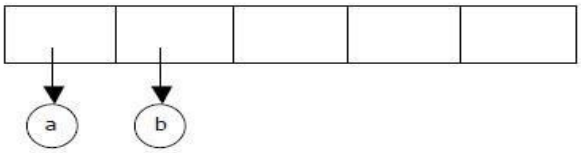
An expression tree can be generated for the infix and postfix expressions. An algorithm to convert a postfix expression into an expression tree is as follows:

- Read the expression from left to right and one symbol at a time.
- If the symbol is an **operand,** we create a one-node tree and push a pointer to it onto a stack.
- If the symbol is an **operator,** we pop pointers to two trees T1 and T2 from the stack (T1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively. A pointer to this new tree is then pushed onto the stack.

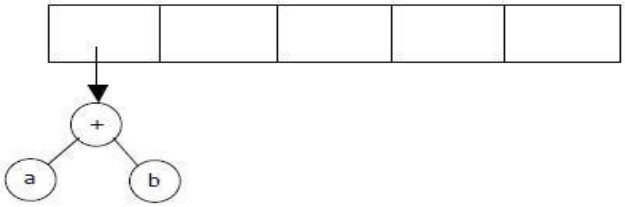
Construct an expression tree for the postfix expression: a b + c d e + * *

Solution:

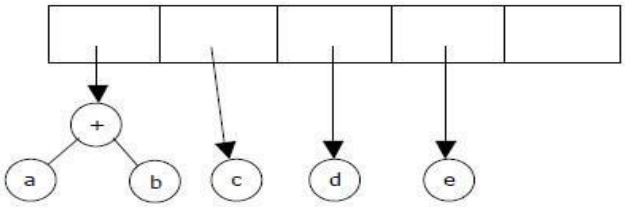
The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.



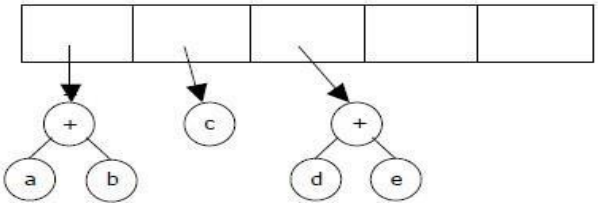
Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



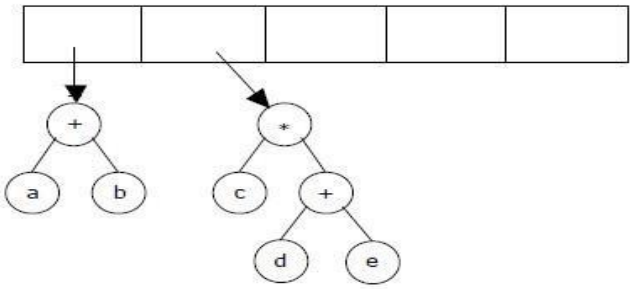
Next, c, d, and e are read, and for each one–node tree is created and a pointer to the corresponding tree is pushed onto the stack.



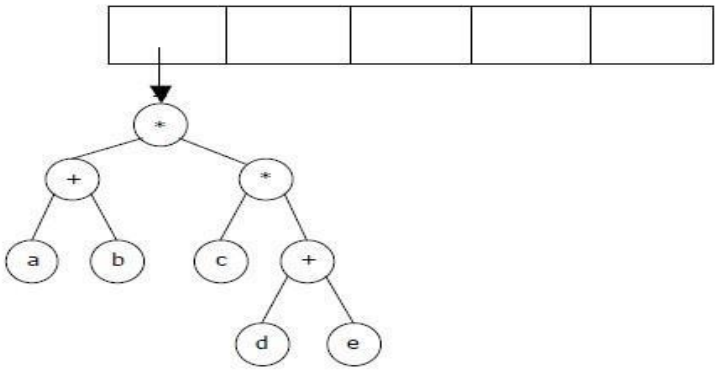
Now a '+' is read, so two trees are merged.



Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.



For the above tree:

Inorder form of the expression: a + b * c * d + e

Preorder form of the expression: * + a b * c + d e

Postorder form of the expression: a b + c d e + * *

2. Building binary tree from traversal pairs

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder,
- Inorder and postorder,
- Inorder and level order

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.Same technique can be applied repeatedly to form sub-trees.

Example 1:

Construct a binary tree from a given preorder and inorder sequence:

Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

Solution:

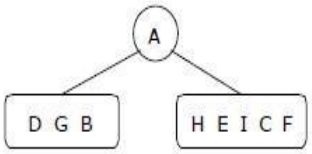
*From Preorder sequence **A** B D G C E H I F, the root is: A*

From Inorder sequence D G B ***A*** H E I C F, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree upto this point looks like:

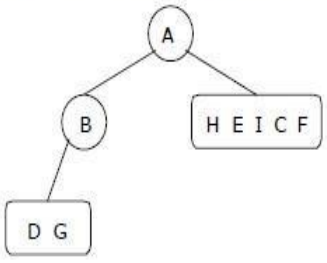


To find the root, left and right sub trees for D G B:

*From the preorder sequence **B** D G, the root of tree is: B*

From the inorder sequence D G **B,** we can find that D and G are to the left of B.

The Binary tree upto this point looks like:

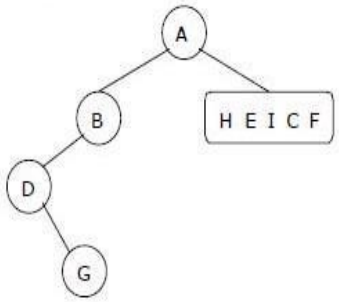


To find the root, left and right sub trees for D G:

*From the preorder sequence **D** G, the root of the tree is: D*

From the inorder sequence ***D*** G, we can find that there is no left node to D and G is at the right of D.

The Binary tree upto this point looks like:

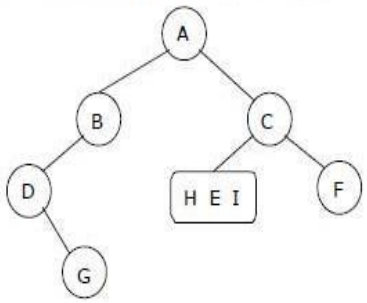


To find the root, left and right sub trees for H E I C F:

*From the preorder sequence **C** E H I F, the root of the left sub tree is: C*

From the inorder sequence *H E I **C** F*, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:

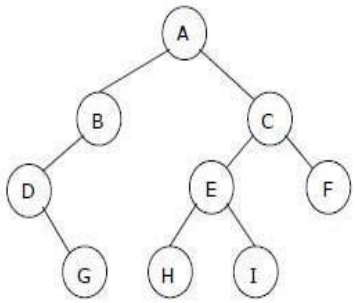


To find the root, left and right sub trees for H E I:

*From the preorder sequence **E** H I, the root of the tree is: E*

From the inorder sequence *H **E** I*, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:



Example 2:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: D G B A H E I C F
Postorder: G D B H I E F C A

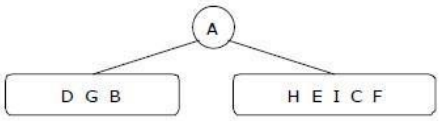
Solution:

From Postorder sequence G D B H I E F C **A**, *the root is: A*

From Inorder sequence *D G B* **A** *H E I C F*, we get the left and right sub trees:

Left sub tree is: D G B
Right sub tree is: H E I C F

The Binary tree upto this point looks like:

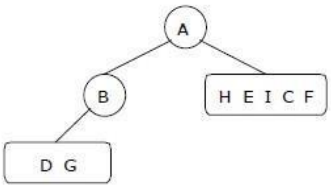


To find the root, left and right sub trees for D G B:

From the postorder sequence G D B, the root of tree is: B

From the inorder sequence *D G* **B,** we can find that D G are to the left of B and there is no right subtree for B.

The Binary tree upto this point looks like:

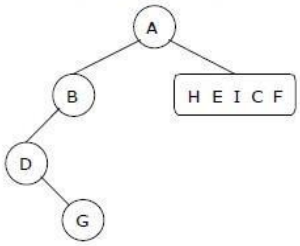


To find the root, left and right sub trees for D G:

*From the postorder sequence G **D**, the root of the tree is: D*

From the inorder sequence **D** *G*, we can find that is no left subtree for D and G is to the right of D.

The Binary tree upto this point looks like:

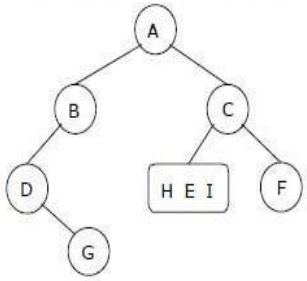


To find the root, left and right sub trees for H E I C F:

*From the postorder sequence H I E F **C**, the root of the left sub tree is: C*

From the inorder sequence H E I ***C*** F, we can find that H E I are to the left of C and F is the right subtree for C.

The Binary tree upto this point looks like:

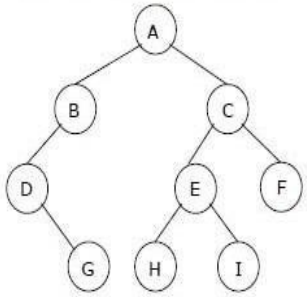


To find the root, left and right sub trees for H E I:

*From the postorder sequence H I **E**, the root of the tree is: E*

From the inorder sequence *H* **E** *I*, we can find that H is left subtree for E and I is to the right of E.

The Binary tree upto this point looks like:



4.10 ADDITIONAL BINARY TREE OPERATIONS

Copying Binary Trees:-

This function is a slightly modified version of postorder. Function which returns an exact copy of the original tree:

```
treepointer copy(treepointer original)
{
        treepointer temp;
        if(original)
        {
                MALLOC(temp, sizeof(*temp));
                temp->leftchild = copy(original->leftchild);
                temp->rightchild = copy(original->rightchild);
                temp->data = original->data;
                return temp;
        }
        return NULL;
}
```

Testing Equality:-

Equivalent binary trees have the same structure and the same information in the corresponding nodes. By same structure, we mean that every branch in one tree corresponds to a branch in the second tree. i.e the branching of the 2 trees is identical. This function returns TRUE if the two trees are equivalent and FALSE if they are not.

```
int equal(treepointer first, treepointer second)
{
        return(( !first && !second) || (first && second && (first->data == second->data) &&
        equal(first->leftchild, second->leftchild) && equal (first->rightchild, second-> rightchild))
}
```

The Satisfiability Problem:-

- Consider the formula that is constructed by set of variables: $x_1, x_2, \ldots, x_n$ and operators $\wedge$(*and*), $\vee$(*or*), $\neg$ (*not*).
- The variables can hold only of two possible values, *true* or *false*.
- The expression can form using these variables and operators is defined by the following rules.
    - A variable is an expression
    - If x and y are expressions, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions
    - Parentheses can be used to alter the normal order of evaluation ($\neg > \wedge > \vee$)

Example: $x_1 \vee (x_2 \wedge \neg x_3)$ If x_1 and x_3 are *false* and x_2 is *true*

$= false \vee (true \wedge \neg false)$

$= false \vee true$

$= true$

The satisfiablity problem for formulas of the propositional calculus asks if there is anassignment of values to the variable that causes the value of the expression to be *true*.

Let's assume the formula in a binary tree

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

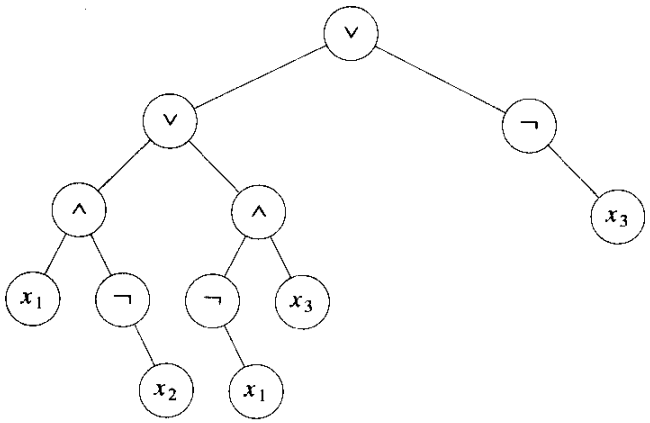


Figure : Propositional formula in a binary tree

The inorder traversal of this tree is

$$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_3 \vee \neg x_3$$

The algorithm to determine satisfiablity is to let (x_1, x_2, x_3) takes on all the possiblecombination of true and false values to check the formula for each combination.

For *n* value of an expression, there are 2^n possible combinations of *true* and *false*
For example n=3, the eight combinations are (t,t,t), (t,t,f), (t,f,t), (t,f,f), (f,t,t), (f,t,f), (f,f,t), (f,f,f).
The algorithm will take $O(g\ 2^n)$, where g is the time to substitute values for x_1, x_2,… x_n andevaluate the expression.
To evaluate an expression, we traverse its tree in postorder,

The node structure for this problem is

leftChild	Data	value	rightChild

Node structure in C:

```
typedef enum {not, and, or, true, false} logical;
typedef struct node *treePointer;
typedef struct
{
        treePointer leftChild;
        logical data;
        short int value;
        treePointer rightChild;
}node;
```

Assume that node->data contains the current value of the variable represented at the leaf node. In the above tree, data field in x1, x2 and x3 contains either TRUE or FALSE. Expression tree with n variables is pointed at by root. With these assumptions we can write our first version of satisfiability algorithm

As below

The first version of Satisfiability algorithm

```
for (all 2^n possible combinations) {
   generate the next combination;
   replace the variables by their values;
   evaluate root by traversing it in postorder;
   if (root→value) {
      printf(<combination>);
      return;
   }
}
printf("No satisfiable combination\n");
```

The C function that evaluates the tree is modified version of postorder traversal, given below:

```
void postordereval(treepointer node)
{
    if( node)
   {
        postordereval(node->leftchild);
        postordereval(node->leftchild);
        switch( node->data)
        {
          case not : node->value = ! node->rightchild->value;
                     break;
          case and : node->value = node->rightchild->value &&
                                   node->leftchild->value ; break;
          case or : node->value = node->rightchild->value ||
                                  node->leftchild->value ; break;
          case true : node->value = TRUE;
                      break;
           case false : node->value = FALSE;
                      break;
        }
     }
```

MODULE-1
STACKS

TOPICS
Stacks: Definition, Stack Operations, Array Representation of Stacks, Stacks using Dynamic Arrays, Stack Applications: Polish notation, Infix to postfix conversion, evaluation of postfix expression.
Recursion - Factorial, GCD, Fibonacci Sequence, Tower of Hanoi, Ackerman's function.

STACK

"**Stack** is an ordered collection of elements or items of same type can be inserted and deleted at only one end called **Top of stack**". (OR)

STACK is an ordered-list in which insertions (called *push*) and deletions (called *pop*) are made at one end called the top. Since last element inserted into a stack is first element removed, a stack is also known as a **LIFO list (Last In First Out).** Stack can be implemented using the Linked List or Array.

Stack belongs to **non-primitive linear data structure**.

Consider a stack s=(a_0,a_1,….,a_{n-1}).here a0 is the bottom element , a_{n-1} is the top element and generally, element a_i is on the top of element a_{i-1},0<i<n.

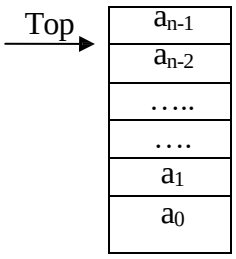


Figure 1a: Stack S

If A, B, C, D, E are the elements added into stack in that order then E is the first element to be deleted. Figure shows insertion and deletion of elements from the stack.

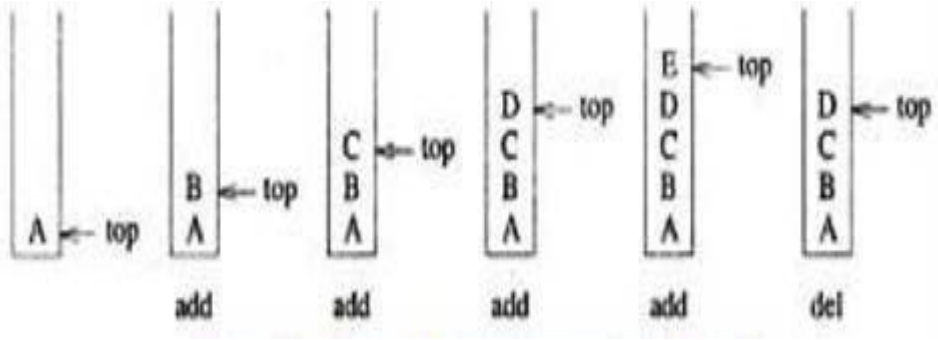


Figure 1b: Insertion and deletion of elements from the stack

Thus, Stack is LIFO Structure i,e **Last in First Out** as shown in figure 1. Example, we can place or remove a card or plate from top of the stack only.

Figure 2: **stack of cards and plates**

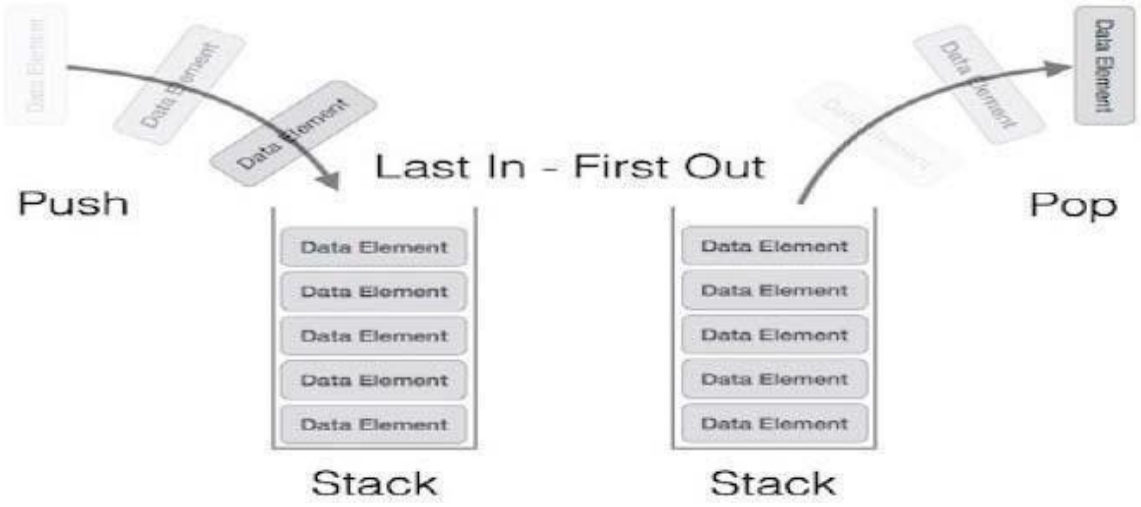


Figure 3: Stack representation

Figure 2 shows the example stacks and figure 3 shows stack representation indicting LIFO operation.

EXAMPLES FOR STACKS:- a stack of coins, a stack of plates , a stack of books , a stack of flooded towels , phone call log, tennis balls in a container, undo and redo operations etc.

2.1 SYSTEM STACK

- A stack used by a program at run-time to process function-calls is called **system-stack** (Fig. 4).

• When functions are invoked, programs create a **stack-frame (or activation-record)** & place the stack-frame on top of system-stack

• Initially, **stack-frame/activation record** for invoked-function contains only **pointer to previous stack-frame & return-address.**

• The **previous stack-frame pointer** points to the stack-frame of the invoking-function, while **return-address contains** the location of the statement to be executed after the function terminates.

• If one function invokes another function, local variables and parameters of the invoking-function are added to its stack-frame.

• A new stack-frame is then created for the invoked-function & placed on top of the system-stack.

• When this function terminates, its stack-frame is removed (and processing of the invoking-function, which is again on top of the stack, continues).

• Frame-pointer(fp) is a pointer to the current stack-frame.

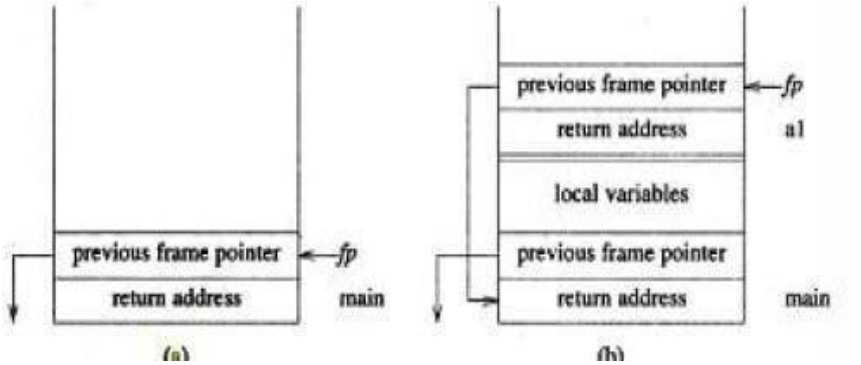


Figure 4: System stack before and after function a1 invoked

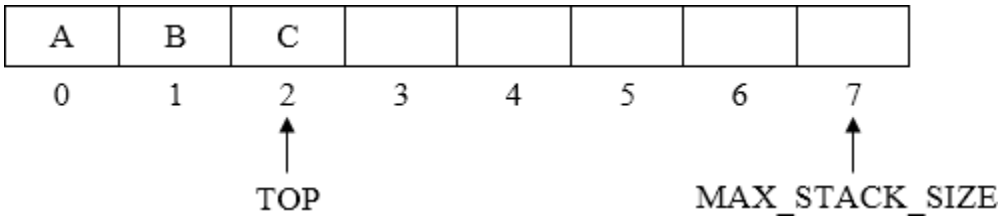
STACK ADT

```
structure Stack is
  objects: a finite ordered list with zero or more elements.
  functions:
   for all stack ∈ Stack, item ∈ element, max_stack_size ∈ positive integer
   Stack CreateS(max_stack_size) ::=
            create an empty stack whose maximum size is max_stack_size

   Boolean IsFull(stack, max_stack_size) ::=
            if(number of elements in stack == max_stack_size)
                return TRUE
            else
                return FALSE

   Stack Add(stack, item) ::=
            if (IsFull(stack))
                stack_full
            else
                 insert item into top of stack and return

    Boolean IsEmpty(stack) ::=
            if(stack == CreateS(max_stack_size))
                return TRUE
            else
                return FALSE

    Element Delete(stack) ::=
           if(IsEmpty(stack))
               return
           else
               remove and return the item on the top of the stack
```

ARRAY REPRESENTATION OF STACKS

- Stacks may be represented in the computer in various ways such as one-way linked list (Singly linked list) or linear array.
- Stacks are maintained by the two variables such as TOP and MAX_STACK_SIZE.
- TOP which contains the location of the top element in the stack. If TOP= -1, then it indicates stack is empty.
- MAX_STACK_SIZE which gives maximum number of elements that can be stored in stack.

Stack can represented using linear array as shown below



2.2 STACK BASIC OPERATIONS

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − pushing (storing / inserting) an element on the stack.
- **pop()** − removing (accessing/ deleting) an element from the stack.

When data is pushed onto stack, to use a stack efficiently we need to check status of stack as well. For the same purpose, the following functionality is added to stacks

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full or overflow.
- **isEmpty()** − check if stack is empty or underflow.

At all times, we maintain a pointer to the last pushed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

PUSH Operation

The process of putting a new data element onto stack is known as **PUSH** Operation. Push operation involves series of steps −

- **Step 1** − Check if stack is full.
- **Step 2** − If stack is full, produce error and exit.
- **Step 3** − If stack is not full, increment **top** to point next empty space.
- **Step 4** − Add data element to the stack location, where top is pointing.
- **Step 5** − return success.

if linked-list is used to implement stack, then in step 3, we need to allocate space dynamically. Push operation is shown in figure 5.

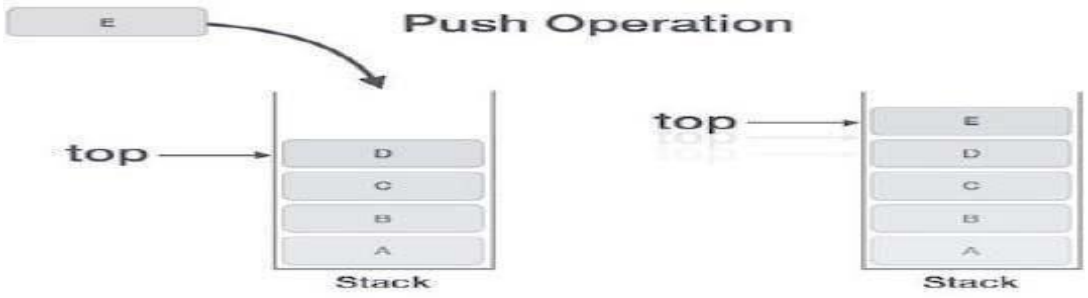


Figure 5: Push operation

POP Operation

Accessing the content while removing it from stack, is known as pop operation. In array implementation of pop() operation, data element is not actually removed, instead **top** is decremented to a lower position in stack to point to next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space. Pop operation is shown in figure6.

A **POP** operation may involve the following steps −

- **Step 1** − Check if stack is empty.
- **Step 2** − If stack is empty, produce error and exit.
- **Step 3** − If stack is not empty, access the data element at which **top** is pointing.
- **Step 4** − Decrease the value of top by 1.
- **Step 5** − return success.

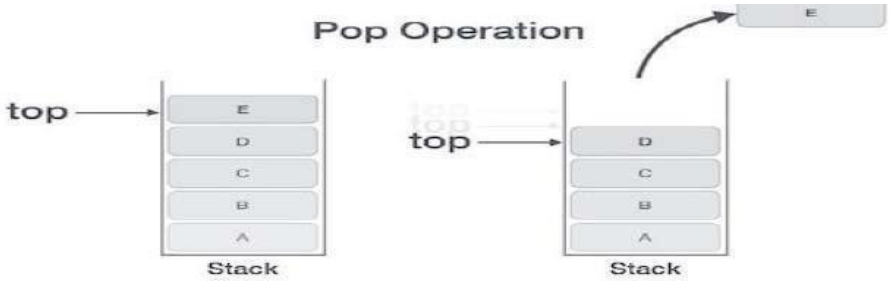


Figure 6: pop operation

IMPLEMENTATION OF STACK OPERATIONS

The easiest way to implement stack ADT is using **one-dimensional array.**

stack[MAX_STACK_SIZE],where MAX_STACK_SIZE =maximum number of entries in the stack.

• The first element of the stack is stored in stack [0],stack[1] is second element and stack[i-1] is the ith element.

• ‟top‟ points to the top element in the stack (top=-1 to denote an emptystack).

• The **CreateS()** function can be implemented as follows and it creates stack of size 100.

```
Stack CreateS(maxStackSize)::=
        #define MAX_STACK_SIZE 100
        struct element
        {
                int key;
        };
element stack[MAX_STACK_SIZE];
int top=-1;
Boolean IsEmpty(Stack)::= top<0;                    //used to check if stack is empty
Boolean IsFull(Stack) ::= top>=MAX_STACK_SIZE-1;    //used to check if stack is full
```

```
void add(int top, element item)
{
    if (top >= MAX_STACK_SIZE-1)
    {
        stack_full( );
        return;
    }
    stack[++top] = item;
}
```

Function push/add() checks to see if the stack is full. If it is, it calls stack_full(), which prints an error message and terminates execution. When the stack is not full, we increment top and assign item to stack[top].

```
element delete(int top)
{
    if (top == -1)
        return stack_empty( );  /* returns and error key */
    return stack[(top)--];
}
```

Function pop/delete() checks to see if the stack is empty using top . If top reaches -1,then it calls stack_empty(), which prints an error message and terminates execution. When the stack is not empty, we return the top most element stack[top] and decrement top.

```
void stack_full()
{  printf(stderr,"stack  is  full,  can"t  add  element");
        exit(EXIT_FAILURE);
}
void stack_empty ()
{       printf(stderr,"stack is empty, can"t deleteelement");
        exit(EXIT_FAILURE);
}
```

2.3 STACK USING DYNAMIC ARRAYS

Shortcoming of static stack implementation: is the need to know at compile-time, a good bound (MAX_STACK_SIZE) on how large the stack will become.

- This shortcoming can be overcome by using a dynamically allocated array for the elements & then

• increasing the size of the array as needed Initially, capacity=1 where capacity=maximum no. of stack-elements that may be stored in array.

• The CreateS() function can be implemented as follows

```
Stack CreateS()::=
        struct element
        {
                int key;
        };

element *stack;
MALLOC(stack,sizeof(*stack));
int capacity=1;
int top=-1;
Boolean IsEmpty(Stack)::= top<0;
Boolean IsFull(Stack) ::= top>=capacity-1;

void stackFull()
{
        REALLOC(stack,2*capacity*sizeof(*stack));
        capacity=2*capacity;
}
```

```
void add(int top, element item)
{
   if (top >= MAX_STACK_SIZE-1)
   {
        stack_full( );
        return;
   }
   stack[++top] = item;
}
```

```
element delete(int top)
{
   if (top == -1)
        return stack_empty( );  /* returns and error key */
   return stack[(top)--];
}
```

```
void stack_full()
{       printf(stderr,"stack is full, can"t addelement");
        exit(EXIT_FAILURE);
}
void stack_empty ()
{       printf(stderr,"stack is empty, can"t deleteelement");
        exit(EXIT_FAILURE);
}
```

Once the stack is full, realloc() function is used to increase the size of array.In array-doubling, we double array-capacity whenever it becomes necessary to increase the capacity of an array.

ANALYSIS

In worst case, the realloc function needs to allocate 2*capacity*sizeof(*stack) bytes of memory and copy capacity*sizeof(*stack) bytes of memory from the old array into the new one. The total time

spent over all array doublings = O(2k) where capacity=2k. Since the total number of pushes is more than 2k-1, the total time spend in array doubling is O(n) where n=total number of pushes.

2.4 APPLICATIONS OF STACKS

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor"s stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

EXPRESSIONS

- An algebraic expression is a legal combination of operators and operands. "The sequence of operators and operands that reduces to a single value after evaluation is called **Expression".**
- **Operand** is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc.
- **Operator** is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

  An algebraic expression can be represented using three different notations. They **are infix,**

postfix and prefix notations:

- **Infix:** It is the form of an arithmetic expression in which we fix (place) the arithmetic operator in between the two operands. Example: (A + B) * (C - D)
- **Prefix:** It is the form of an arithmetic notation in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as **polish notation** (due to the polish mathematician Jan Lukasiewicz in the year 1920). Example: * + A B – C D
- **Postfix:** It is the form of an arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as ***suffix notation*** and is also referred to ***reverse polish notation*****.** Example: A B + C D - *

The **three important features of postfix expression are**:

- Postfix expression is parenthesis-free expression.
- While evaluating the postfix expression the precedence and Associativity of the operators is no longer required
- all expressions given to the system , will be converted into postfix form by the complier since it is easy and more efficient to evaluate.

Converting expressions using Stack:

Let us convert the expressions from one type to another. These can be done as follows

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

Precedence of the operators

The first problem with understanding the meaning of expressions and statements is finding out the order in which the operations are performed.

Example: assume that a =4, b =c =2, d =e =3 in below expression

X = a / b – c + d * e – a * c

((4/2)-2) + (3*3)-(4*2)
=0+9-8
=1

OR

(4/ (2-2 +3)) *(3-4)*2
= (4/3) * (-1) * 2
= -2.66666

The first answer is picked most because division is carried out before subtraction, and multiplication before addition. If we wanted the second answer, write expression differently using parentheses to change the order of evaluation

X= ((a / (b – c + d)) * (e – a) * c

In C, there is a precedence hierarchy that determines the order in which operators are evaluated. Below figure contains the precedence hierarchy for C.

Token	Operator	Precedence	Associativity
( ) [ ] ⟶	function call array element struct or union member	17	left-to-right
-- ++	Increment, Decrement	16	left-to-right
--++ ! ~ -+ & * sizeof	decrement, increment logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	Multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	Bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	Bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right -to-left
,	comma	1	left-to-right

CONVERSION FROM INFIX TO POSTFIX

Procedure to convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. If the scanned symbol is **left parenthesis**, push it onto the stack.
3. If the scanned symbol is an **operand**, then place directly in the postfix expression (output).
4. If the symbol scanned is a **right parenthesis**, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
5. If the scanned symbol is an **operator**, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (*or greater than or equal*) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example 1:
Convert ((A – (B + C)) * D) ↑ (E + F) infix expression to postfix form:

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

EVALUATION OF POSTFIX EXPRESSION

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage. Although infix notation is the most common way of writhing expressions, it is not the one used by compilers to evaluate expressions. Instead compilers typically use a parenthesis-free postfix notation.

Steps for evaluating postfix expression

1) Scan the symbol from left to right.
2) If the scanned-symbol is an **operand**, push it on to the stack.
3) If the scanned-symbol is an **operator**, pop two operands from the stack. The first popped operand acts as operand2 and the second popped operand act as operand 1.Now perform the indicated operation and Push the result on to the stack.
4) Repeat the above procedure till the end of input is encountered

Example 1:

Evaluate the postfix expression: 6 5 2 3 + 8 * + 3 + *

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK	REMARKS
6				6	
5				6, 5	
2				6, 5, 2	
3				6, 5, 2, 3	The first four symbols are placed on the stack.
+	2	3	5	6, 5, 5	Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed
8	2	3	5	6, 5, 5, 8	Next 8 is pushed
*	5	8	40	6, 5, 40	Now a '*' is seen, so 8 and 5 are popped as 8 * 5 = 40 is pushed
+	5	40	45	6, 45	Next, a '+' is seen, so 40 and 5 are popped and 40 + 5 = 45 is pushed
3	5	40	45	6, 45, 3	Now, 3 is pushed
+	45	3	48	6, 48	Next, '+' pops 3 and 45 and pushes 45 + 3 = 48 is pushed
*	6	48	288	288	Finally, a '*' is seen and 48 and 6 are popped, the result 6 * 48 = 288 is pushed

Example 2:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

2.5 RECURSION

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion()
{
recursion(); /* function calls itself */
}
void main()
{
recursion();
}
```

The C programming language supports **recursion, i.e., a function to call itself**.

A recursive function is a function that calls itself during its execution.

But while using recursion, programmers need to be careful to define an exit condition from the function; otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

RECURSION PROPERTIES

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have −

- **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Recursive criteria** − the recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

1. FACTORIAL NUMBER

The product of the positive integers from 1 to n, inclusive is called **n factorial** and is usually denoted by n!

n!=1*2*3*4*……*(n-2)*(n-1)*n.

Factorial function may be defined as

a. if n=0 then return 1

b. if n>0, then return n*(n-1)!

The following example **calculates the factorial of a given number using a recursive function**

```
#include <stdio.h>
int factorial(unsigned int n)
{
        if(n <= 1)
         {
                return 1;
        }
return n * factorial(n - 1);
}
```

```
void main()
{
        int n,res;
        printf("\n enter the value for n");
        scanf("%d",&n);
        res=factorial(n);
        printf("Factorial of %d is %d\n", n, res);
}
```

When the above code is compiled and executed, it produces the following result − **Factorial of 15 is 2004310016**

2. GCD OF TWO NUMBERS

GCD is calculated by using GCD(a,b)=GCD(b,a mod b).

Euclid‟s algorithm

1. Take a and b, and calculate the remainder by performing a%b.
2. Assign the value of b to a and value of remainder to b.
3. Repeat the steps 1 & 2 until value of b becomes 0.
4. If b=0, then return value of „a‟ as the GCD value of a & b.

```
#include <stdio.h>
int gcd(int m, int n);
void main()
{
        int a, b,res;
        printf("Enter two positive integers: ");
        scanf("%d %d", &a, &b);
        res=gcd(a,b);
        printf("G.C.D of %d and %d is %d.", n1, n2, res);
}
int gcd(int a, int b)
{
        int rem;
        if (b==0)
                return a;
        else
        {
             rem=a%b;
                a=b;
                b=rem;
                return gcd(a,b);
}
```

Output

Enter two positive integers:
366 60
G.C.D of 366 and 60 is 6

3. FIBONACCI SERIES

The Fibonacci sequence is the sequence of integers, where each number in this sequence is the sum of two preceding elements.

A formal definition is

a. If n=0 or 1,return n(0/1)

b. If n>1,returnfib(n-1)+fib(n-2)

```
#include <stdio.h>
int fib(int i)
{
        if(i == 0)
        {
                return 0;
        }
        if(i == 1)
         {
                return 1;
        }
        return (fib(i-1) + fib(i-2));
}
void main()
{
        int i;
        for (i = 0; i < 10; i++)
         {
                printf("%d\t", fib(i));
        }
}
```

When the above code is compiled and executed, it produces the following result −

0 1 1 2 3 5 8 13 21 34

4. TOWER OF HANOI PROBLEM

Tower of Hanoi, is a mathematical puzzle which consists of three tower (pegs) and more than one rings; as depicted in below figure 7

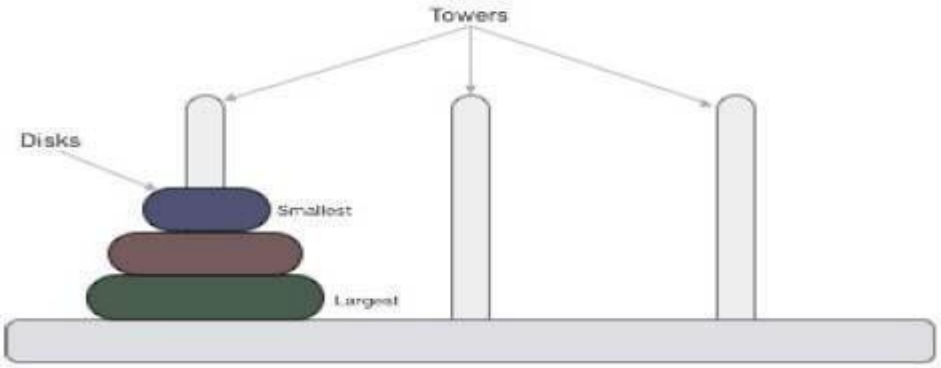


Figure 7: Tower of Hanoi

These rings are of different sizes and stacked upon in ascending order i.e. the smaller one sits over the larger one. There are other variations of puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. The below mentioned are few rules which are to be followed for tower of hanoi −

- **Only one disk can be moved among the towers at any given time.**
- **Only the "top" disk can be removed.**
- **No large disk can sit over a small disk.**

Here is an animated representation in figure 8 of solving a tower of hanoi puzzle with three disks −Tower of hanoi puzzle with **n** disks can be solved in minimum **2n−1** steps. This presentation shows that a puzzle with 3 disks has taken **23−1** = 7 steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

- First we move the smaller one (top) disk to aux peg
- Then we move the larger one (bottom) disk to destination peg
- And finally, we move the smaller one from aux to destination peg.

So now we are in a position to design algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nthdisk) is in one part and all other (n-1) disks are in second part. Our ultimate aim is to move disk n from source to destination and then put all other (n-1) disks onto it. Now we can imagine to apply the same **in recursive way** for all given set of disks.

So steps to follow are −

- **Step 1 −** Move n-1 disks from **source** to **aux**
- **Step 2 −** Move nth disk from **source** to **dest**
- **Step 3 −** Move n-1 disks from **aux** to **dest**

A recursive algorithm for Tower of Hanoi can be driven as follows −

START

```
Procedure tower(disk, source, dest, aux)
if n == 1, THEN
        move disk from source to dest
```

```
else
        tower(n - 1, source, aux, dest) // Step 1
        move disk from source to dest // Step 2
        tower(n - 1, aux, dest, source) // Step 3
END IF
END Procedure
```
STOP

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include <stdio.h>
void towers(int, char, char, char);
int main()
{
    int num;
    printf("Enter the number of disks : ");
   scanf("%d", &num);
    printf("The sequence of moves involved in the Tower of Hanoi are :\n");
    towers(num, 'A', 'C', 'B');
   getch();
    return 0;
}
void towers(int num, char source, char dest, char aux)
{
    if (num == 1)
    {
        printf("\n Move disk 1 from peg %c to peg %c", source,dest);
        return;
    }
    towers(num - 1, source, aux,dest);
    printf("\n Move disk %d from peg %c to peg %c", num, source, dest);
    towers(num - 1, aux, dest, source);
}
```

The below figure shows the disks movements in tower of Hanoi for 3 disks

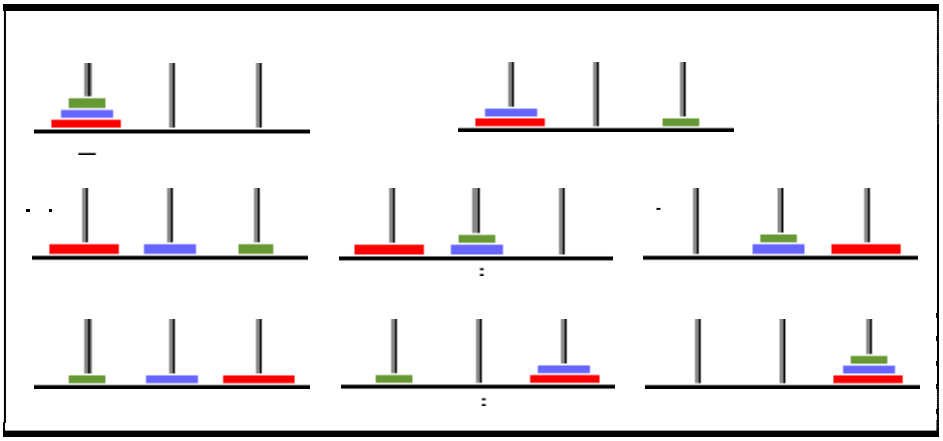
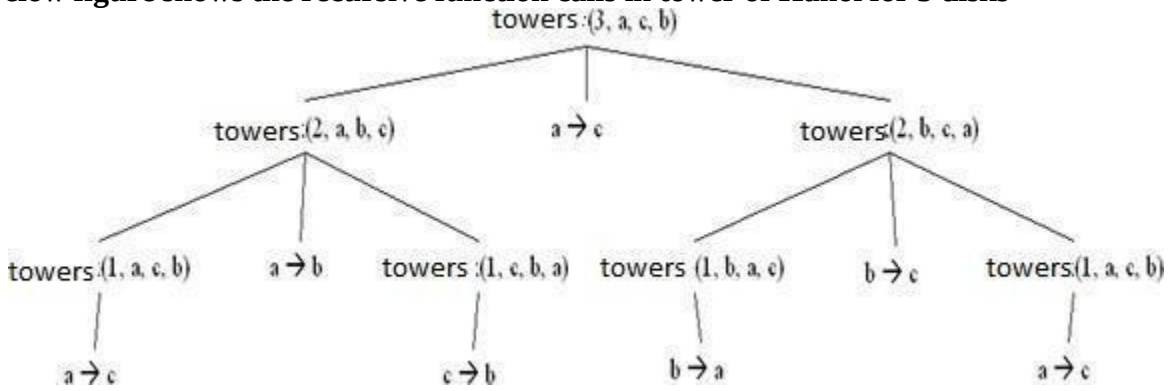


Figure 8: Disk movements

The below figure shows the recursive function calls in tower of Hanoi for 3 disks



ACKERMANN'S FUNCTION

Ackermann–Péter function, is defined as follows for nonnegative integers *m* and *n*:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

To solve A(2,1) using Ackerman‟s function: (m=2, n=1)

A(2,1) = A(1,A(2,0))
= A(1,A(1,1))
=A(1,A(0,A(1,0)))
=A(1,A(0,A(0,1)))
=A(1,A(0,2))
=A(1,3)
=A(0,A(1,2))
=A(0,A(0,A(1,1)))
=A(0,A(0,A(0,A(1,0))))
=A(0,A(0,A(0,A(0,1))))
=A(0,A(0,A(0,2)))
=A(0,A(0,3))
=A(0,4)
A(2,1) = 5

Program:-

```
#include<stdio.h>
int ackerman(int m, int n)
{
        if(m==0)
                return (n+1);
        if(n==0 && m>0)
                return ackerman(m-1,1);
        if(m>0 && n>0)
                return ackerman(m-1,ackerman(m,n-1));
}
void main()
```

```
{
        int m,n;
        printf("Enter the value of m and n\n");
        scanf("%d %d",&m,&n);
        printf("%d",ackerman(m,n));
}
```

K.S SCHOOL OF ENGINEERING AND MANAGEMENT
DEPARTMENT OF COMPUTER SCIENCE & ENGG.
Session 2023-2024 (Odd Semester)
BCS304 Data Structures & Applications

Advanced Learners-1

Sl No.	USN	NAME	SIGNATURE
1	1KG22CS083	Peddinti Mohammad	

Staff Incharge

HOD

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S SCHOOL OF ENGINEERING AND MANAGEMENT
DEPARTMENT OF COMPUTER SCIENCE & ENGG.
Session 2023-2024 (Odd Semester)
BCS304 Data Structures & Applications

Advances Learners-2

Sl No.	USN	NAME	SIGNATURE
1	1KG22CS078	Padmashree M M	
2	1KG22CS080	Pavan Kumar	
3	1KG22CS083	Peddinti Mohammad	
4	1KG22CS087	Prajwal koushik c	
5	1KG22CS093	Raghu Kisthannavar	
6	1KG22CS094	Rakesh V	
7	1KG22CS095	Rakshitha. N	
8	1KG22CS115	Taanish M	
9	1KG22CS119	T Venkata Praneeth	

Staff Incharge

HOD

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S SCHOOL OF ENGINEERING AND MANAGEMENT

DEPARTMENT OF COMPUTER SCIENCE & ENGG.

Attendance for Remedial Class

YEAR / SEMESTER - III - 'B'

COURSE TITLE - Data Structures and Application

COURSE CODE -BCS304

S.NO	USN	NAME	Signature
1	1kG22CS066	Marineni Hansika	
2	1KG22CS068	MohammadSami Sayyad	
3	1KG22CS069	Mohith reddy k	
4	1KG22CS071	Meghana	
5	1KG22CS072	M.Dheekshith	
6	1KG22CS073	Nishanth R	
7	1KG22CS074	Niswana N Swamy	
8	1KG22CS075	NITHISH KUMAR V	
9	1KG22CS079	P Devish Chowdary	
10	1KG22CS081	PAVAN T L	
11	1KG22CS082	PAVAN U	
12	1KG22CS084	Pooja S	
13	1KG22CS088	Pranav Ramesh	
14	1KG22CS089	Prapul u	
15	1KG22CS090	Priya RK	
16	1KG22CS093	RAGHU KISTHANNAVAR	
17	1KG22CS099	Ramya.P	
18	1KG22CS100	Rani	
19	1KG22CS103	Sakesh P	
20	1KG22CS104	Sanjana B	
21	1KG22CS105	Sanjay M D	
22	1KG22CS106	Sanjay S	
23	1KG22CS107	Santosh Kumar nagur	
24	1KG22CS108	Saran R	
25	1KG22CS109	Shashank m goudar	
26	1KG22CS110	Shashank D Urs	
27	1KG22CS111	Sindhushree.k	
28	1KG22CS113	SUHAS S	
29	1KG22CS114	T kavya	
30	1KG22CS116	TARUN.R	
31	1KG22CS118	T Abhishek	
32	1KG22CS120	Toluchuru Haritha	
33	1KG22CS121	UDAYKIRAN	
34	1KG22CS124	Vismaya N	
35	1KG22CS125	Yashwanth R	
36	1KG21CS020	Bhoomika P Desai	
37	DIP	Venkatachal S	
38	DIP	Rajesh PC	
39	DIP	Suhas Madhusudan Shandilya	

Faculty Incharge

HOD

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S SCHOOL OF ENGINEERING AND MANAGEMENT

DEPARTMENT OF COMPUTER SCIENCE & ENGG.

Attendance for Remedial Class

YEAR / SEMESTER - III - 'B'

COURSE TITLE - Data Structures and Application

COURSE CODE -BCS304

S.NO	USN	NAME	Signature
1	1kG22CS066	Marineni Hansika	
8	1KG22CS075	Nithish Kumar V	
13	1KG22CS088	Pranav Ramesh	
14	1KG22CS089	Prapul u	
17	1KG22CS099	Ramya.P	
	1KG22CS102	Sadhvika Godavarthi	
22	1KG22CS106	Sanjay S	
24	1KG22CS108	Saran R	
26	1KG22CS110	Shashank D Urs	
28	1KG22CS113	Suhas S	
29	1KG22CS114	T kavya	
30	1KG22CS116	Tarun R	
31	1KG22CS118	T Abhishek	
33	1KG22CS121	Udaykiran	
35	1KG22CS125	Yashwanth R	
37		Venkatachal S	
38		Rajesh PC	
39		Suhas Madhusudan Shandilya	

HOD

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT,BENGALURU-560 109
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING.
2023-24 ODD SEMSTER III SEM B SEC

SL. NO.	USN	NAME OF THE STUDENT	DATA STRUCTURES AND APPLICATION BCS304 BINDU K P									
			IA1	IA2	IA3	Best of 2	Scale down (IA)	AS1	AS2	AVG of AS	Total IA + AS	Signature
			25	25	25	50	25	25	25	25	50	
1	1KG22CS064	Mahalakshmi P.S	15	14	25	40	20	25	25	25	45	
2	1KG22CS065	Mansi M	12	13	18	31	16	25	25	25	41	
3	1kG22CS066	Marineni Hansika	4	6	13	19	10	25	23	24	34	
4	1KG22CS067	Marpuri Sowmya	10	18	25	43	22	25	25	25	47	
5	1KG22CS068	MohammadSami Sayyad	9	12	13	25	13	25	23	24	37	
6	1KG22CS069	Mohith reddy k	8	15	9	24	12	23	23	23	35	
7	1KG22CS070	Monika k	13	15	22	37	19	25	25	25	44	
8	1KG22CS071	Meghana	9	15	20	35	18	25	25	25	43	
9	1KG22CS072	M.Dheekshith	5	18	20	38	19	25	20	23	42	
10	1KG22CS073	Nishanth R	5	14	9	23	12	23	23	23	35	
11	1KG22CS074	Niswana N Swamy	AB	14	10	24	12	25	25	25	37	
12	1KG22CS075	NITHISH KUMAR V	2	2	18	20	10	23	23	23	33	
13	1KG22CS076	Niveda B	11	11	18	29	15	25	23	24	39	
14	1KG22CS077	P Harshitha	12	15	AB	27	14	25	25	25	39	
15	1KG22CS078	Padmashree M M	17	23	24	47	24	25	25	25	49	
16	1KG22CS079	P Devish Chowdary	5	14	11	25	13	22	22	22	35	
17	1KG22CS080	Pavan Kumar	14	24	21	45	23	25	25	25	48	
18	1KG22CS081	PAVAN T L	6	19	18	37	19	23	23	23	42	
19	1KG22CS082	PAVAN U	6	12	16	28	14	25	25	25	39	
20	1KG22CS083	Peddinti Mohammad	24	25	AB	49	25	25	25	25	50	
21	1KG22CS084	Pooja S	9	12	16	28	14	25	23	24	38	
22	1KG22CS085	Poojitha S	12	12	19	31	16	25	25	25	41	
23	1KG22CS086	PRAGNA PS	12	15	20	35	18	25	25	25	43	
24	1KG22CS087	Prajwal koushik c	17	21	25	46	23	24	25	25	48	
25	1KG22CS088	Pranav Ramesh	9	AB	14	23	12	25	23	24	36	
26	1KG22CS089	Prapul u	AB	9	11	20	10	20	20	20	30	
27	1KG22CS090	Priya RK	9	13	19	32	16	25	23	24	40	
28	1KG22CS091	Punith.B	11	16	AB	27	14	23	23	23	37	
29	1KG22CS092	R.PRUDVI GANESH	11	19	19	38	19	25	25	25	44	
30	1KG22CS093	RAGHU KISTHANNAVAR	5	20	18	38	19	24	24	24	43	
31	1KG22CS094	Rakesh V	17	25	24	49	25	25	25	25	50	
32	1KG22CS095	Rakshitha. N	13.5	21.5	23	45	23	25	25	25	48	
33	1KG22CS096	RAKSHITHA S	16	18	24	42	21	25	22	24	45	
34	1KG22CS097	Rakshitha.S	13	18	AB	31	16	25	25	25	41	
35	1KG22CS098	Ramitha.k	11	14	19	33	17	25	25	25	42	
36	1KG22CS099	Ramya.P	6	5	15	20	10	25	23	24	34	
37	1KG22CS100	Rani	7	18	20	38	19	25	23	24	43	
38	1KG22CS101	Rayan Nadeem	12	14	4	26	13	25	25	25	38	
39	1KG22CS102	Sadhvika godavarthi	11	AB	21	32	16	25	25	25	41	
40	1KG22CS103	Sakesh P	6	18	11	29	15	23	25	24	39	
41	1KG22CS104	Sanjana B	1	15	14	29	15	25	25	25	40	
42	1KG22CS105	Sanjay M D	6	16	18	34	17	24	25	25	42	

43	1KG22CS106	Sanjay S	AB	5	15	20	10	20	20	20	30	
44	1KG22CS107	Santosh Kumar nagur	5	14	19	33	17	23	23	23	40	
45	1KG22CS108	Saran R	3	10	12	22	11	20	20	20	31	
46	1KG22CS109	Shashank m goudar	1	12	9	21	11	25	20	23	34	
47	1KG22CS110	Shashank D Urs	8	8	13	21	11	25	25	25	36	
48	1KG22CS111	Sindhushree.k	9	12	12	24	12	25	23	24	36	
49	1KG22CS112	Sowjanya k s	10	15	21	36	18	25	23	24	42	
50	1KG22CS113	SUHAS S	8	11	16	27	14	25	25	25	39	
51	1KG22CS114	T kavya	5	7	13	20	10	25	23	24	34	
52	1KG22CS115	Taanish M	14	23	21	44	22	25	24	25	47	
53	1KG22CS116	TARUN.R	8	10	13	23	12	25	23	24	36	
54	1KG22CS117	Tejaswini RM	10	15	15	30	15	25	24	25	40	
55	1KG22CS118	T Abhishek	2	3	18	21	11	20	20	20	31	
56	1KG22CS119	T.VENKATA PRANEETH	14	21	18	39	20	25	25	25	45	
57	1KG22CS120	Toluchuru Haritha	9	13	16	29	15	25	25	25	40	
58	1KG22CS121	UDAYKIRAN	2	9	14	23	12	25	25	25	37	
59	1KG22CS122	Vandana Basavaraj Patil	13	16	20	36	18	25	25	25	43	
60	1KG22CS123	Vinayak C	12	11	20	31	16	25	25	25	41	
61	1KG22CS124	Vismaya N	7	15	15	30	15	25	25	25	40	
62	1KG22CS125	Yashwanth R	3	3	18	21	11	22	22	22	33	
63	1KG21CS020	Bhoomika P Desai	9	19	11	30	15	25	25	25	40	
64	1KG23CS403	Rajesh PC	0	2	18	20	10	20	20	20	30	
65	1KG23CS405	Suhas M Shardilya	1	3	19	20	10	23	22	23	32	
66	1KG23CS406	Venkatachal S	1	3	20	23	12	22	22	22	34	

Faculty In-Charge

HOD

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE

Branch : CS Semester : 3

Sl NO.	USN	BCS304
1	1KG21CS020	40
2	1KG22CS001	35
3	1KG22CS002	48
4	1KG22CS003	46
5	1KG22CS004	43
6	1KG22CS005	38
7	1KG22CS006	50
8	1KG22CS007	40
9	1KG22CS008	37
10	1KG22CS009	43
11	1KG22CS010	46
12	1KG22CS011	37
13	1KG22CS012	37
14	1KG22CS013	43
15	1KG22CS014	42
16	1KG22CS015	39
17	1KG22CS016	44
18	1KG22CS017	26
19	1KG22CS018	41
20	1KG22CS019	44
21	1KG22CS020	43
22	1KG22CS021	35
23	1KG22CS022	48
24	1KG22CS023	44
25	1KG22CS024	36
26	1KG22CS025	35
27	1KG22CS026	40
28	1KG22CS027	38
29	1KG22CS028	46
30	1KG22CS029	37
31	1KG22CS030	47
32	1KG22CS031	36
33	1KG22CS032	44
34	1KG22CS033	50
35	1KG22CS034	39
36	1KG22CS035	42

Sl NO.	USN	BCS304
37	1KG22CS036	34
38	1KG22CS037	36
39	1KG22CS038	41
40	1KG22CS039	48
41	1KG22CS040	45
42	1KG22CS041	39
43	1KG22CS042	42
44	1KG22CS043	38
45	1KG22CS044	43
46	1KG22CS045	44
47	1KG22CS046	45
48	1KG22CS047	35
49	1KG22CS048	35
50	1KG22CS049	44
51	1KG22CS050	42
52	1KG22CS051	36
53	1KG22CS052	36
54	1KG22CS053	31
55	1KG22CS054	40
56	1KG22CS055	44
57	1KG22CS056	44
58	1KG22CS057	45
59	1KG22CS058	42
60	1KG22CS059	36
61	1KG22CS060	39
62	1KG22CS061	38
63	1KG22CS062	41
64	1KG22CS063	35
65	1KG22CS064	45
66	1KG22CS065	41
67	1KG22CS066	34
68	1KG22CS067	47
69	1KG22CS068	37
70	1KG22CS069	35
71	1KG22CS070	44
72	1KG22CS071	43
73	1KG22CS072	42
74	1KG22CS073	35
75	1KG22CS074	37

Sl NO.	USN	BCS304
76	1KG22CS075	33
77	1KG22CS076	39
78	1KG22CS077	39
79	1KG22CS078	49
80	1KG22CS079	35
81	1KG22CS080	48
82	1KG22CS081	42
83	1KG22CS082	39
84	1KG22CS083	50
85	1KG22CS084	38
86	1KG22CS085	41
87	1KG22CS086	43
88	1KG22CS087	48
89	1KG22CS088	36
90	1KG22CS089	30
91	1KG22CS090	40
92	1KG22CS091	37
93	1KG22CS092	44
94	1KG22CS093	43
95	1KG22CS094	50
96	1KG22CS095	48
97	1KG22CS096	45
98	1KG22CS097	41
99	1KG22CS098	42
100	1KG22CS099	35
101	1KG22CS100	43
102	1KG22CS101	38
103	1KG22CS102	41
104	1KG22CS103	39
105	1KG22CS104	40
106	1KG22CS105	42
107	1KG22CS106	30
108	1KG22CS107	40
109	1KG22CS108	35
110	1KG22CS109	34
111	1KG22CS110	36
112	1KG22CS111	36
113	1KG22CS112	42
114	1KG22CS113	39

Sl NO.	USN	BCS304
115	1KG22CS114	34
116	1KG22CS115	47
117	1KG22CS116	36
118	1KG22CS117	40
119	1KG22CS118	31
120	1KG22CS119	45
121	1KG22CS120	40
122	1KG22CS121	37
123	1KG22CS122	43
124	1KG22CS123	41
125	1KG22CS124	40
126	1KG22CS125	33
127	1KG23CS400	38
128	1KG23CS401	31
129	1KG23CS402	38
130	1KG23CS403	30
131	1KG23CS404	33
132	1KG23CS405	32
133	1KG23CS406	34

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU-5[illegible]09

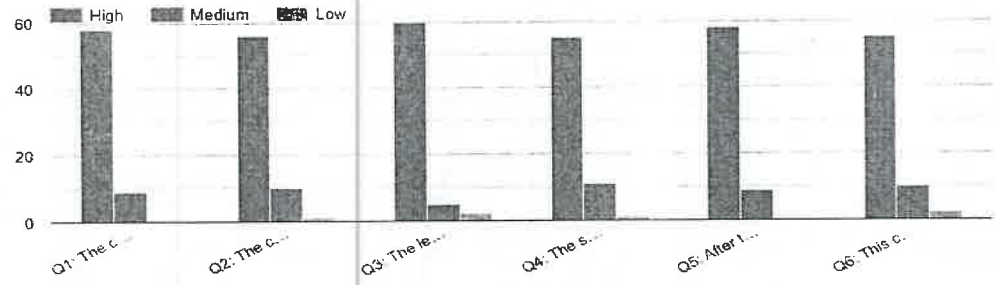
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING.

2023-24 ODD SEMESTER - III SEM B SECTION

DATA STRUCTURES AND APPLICATIONS (BCS304D) - COURSE END SURVEY

Timestamp	Email Address	NAME OF THE STUDENT	USN	[Q1: The course increased your level of interest?]	[Q2: The course content was appropriate and was presented in a structured manner]	[Q3: The learning material, theory/practical sessions were relevant to the course outcomes]	[Q4: The self-study (including reading) required for this course will ensure better achievement of course objectives]	[Q5: After this course, you will be able to solve analyze real life engineering problems related to this course]	[Q6: This course has given you enough understanding to take next level courses]	Signature
3-6-2024 15:24:34	naseemsayyad388@gmail.co	MohammadSami Shakeelahma	1KG22CS068	High	High	High	High	High	High	
3-6-2024 15:25:07	prajwalkoushik27@gmail.com	PRAJWAL KOUSHIK C	1KG22CS087	High	High	High	High	High	High	
3-6-2024 15:25:41	pavankumar462004@gmail.c	Pavankumar	1KG22CS080	High	High	High	High	High	High	
3-6-2024 15:27:59	prudviganesh007@gmail.com	R PRUDVI GANESH	1KG22CS092	High	High	High	High	Medium	Medium	
3-6-2024 15:28:21	t.praneeth123@gmail.com	T V PRANEETH	1KG22CS119	High	High	High	High	High	High	
3-6-2024 15:44:59	sanjumadala2424@gmail.con	Sanjay M D	1KG22CS105	High	High	High	High	High	High	
3-6-2024 15:50:45	prajwalnaidu215@gmail.com	Venkatachal s	1KG24CS406	High	High	High	High	High	High	
3-6-2024 16:09:27	challaeswaraiah14379@gmai	Marpuri Sowmya	1KG22CS067	High	High	High	High	High	High	
3-6-2024 16:12:22	sowjanyaks343@gmail.com	Sowjanya K S	1KG22CS112	High	High	High	Medium	High	High	
3-6-2024 16:16:01	maha34493@gmail.com	Mahalakshmi PS	1KG22CS064	High	High	High	High	High	High	
3-6-2024 16:17:31	mohammadaslam62819@gm	Peddinti Mohammad	1kg22cs083	High	High	High	High	High	High	
3-6-2024 16:23:18	rakshitha.blossom@gmail.cor	RAKSHITHA S	1KG22CS096	High	High	High	High	High	High	
3-6-2024 16:31:11	harithat2004@gmail.com	Toluchuru Haritha	1KG22CS120	High	High	High	High	High	High	
3-6-2024 16:35:16	rakshithasubramanyaachara(	Rakshitha S	1KG22CS097	High	High	High	High	High	High	
3-6-2024 16:45:40	22pspragna@gmail.com	PRAGNA PS	1KG22CS086	High	High	High	Medium	High	Medium	
3-6-2024 19:52:36	sanarathod717@gmail.com	Sanjana B	1KG22CS104	Medium	High	High	High	High	High	
3-6-2024 20:14:29	rayannadeem7@gmail.com	Rayan Nadeem	1KG22CS101	High	High	High	High	High	High	
3-6-2024 20:50:59	pavangowdatl0704@gmail.co	PAVAN T L	1KG22CS081	High	High	High	High	High	High	
3-7-2024 12:03:30	bhoomikapdesai@gmail.com	Bhoomika P Desai	1KG21CS020	High	High	High	High	High	High	
3-7-2024 12:22:14	mummaneedimeghana1@gm	Meghana	1KG22CS71	High	High	High	High	Medium	High	
3-7-2024 12:48:51	vinay740ir@gmail.com	VINAYAK C	1KG22CS123	High	Medium	High	High	Medium	High	
3-7-2024 13:14:15	vandu9426@gmail.com	Vandana Basavaraj Patil	1KG22CS122	High	High	High	Medium	High	High	
3-7-2024 15:35:50	sadhvikagodavarthi@gmail.c	SADHVIKA GODAVARTHI	1KG22CS102	Medium	High	High	High	High	Medium	
3-7-2024 16:11:11	maranenihansika@gmail.com	M Hansika	1KG22CS066	High	High	High	High	High	High	
3-7-2024 17:08:06	padmashree1384@gmail.com	Padmashree	1KG22CS078	High	High	High	High	High	High	
3-7-2024 17:24:39	poojasampath0305@gmail.co	Pooja S	1KG22CS084	Medium	Medium	Medium	Medium	Medium	Medium	
3-7-2024 18:37:38	nithishkumarv22122003@gm	Nithish Kumar V	1kg22cs075	High	High	High	High	High	High	
3-7-2024 18:37:43	naveenvismya@gmail.com	Vismaya N	124	High	High	High	High	High	High	
3-7-2024 18:38:04	ramyapramya178@gmail.com	Ramya.P	1KG22CS099	High	High	High	High	High	High	
3-7-2024 18:40:02	ry352004@gmail.com	Rakesh V	1KG22CS094	High	High	High	High	Medium	Medium	
3-7-2024 18:40:28	sanjayshanthraju123@gmail.(	Sanjay S	1KG22CS106	High	Medium	High	Medium	High	High	
3-7-2024 18:40:47	niswananarayanswamy@gma	Niswana N Swamy	1KG22CS074	High	High	High	High	Medium	Medium	
3-7-2024 18:43:10	acchiharshitha15@gmail.com	P Harshitha	1KG22CS077	Medium	Medium	Medium	Medium	Medium	Medium	
3-7-2024 18:45:26	rakshitha2735@gmail.com	Rakshitha. N	1KG22CS095	Medium	Medium	Medium	Medium	Medium	Low	
3-7-2024 18:52:16	priyark0809@gmail.com	Priya R K	1KG22CS090	Medium	Low	Low	Low	Medium	High	
3-7-2024 19:34:56	kavyanaidu359@gmail.com	Kavya	1kg22cs114	High	High	High	High	High	High	
3-7-2024 20:04:34	nr99910@gmail.com	Nishanth R	1KG22CS073	Medium	Medium	High	High	High	Medium	
3-7-2024 20:24:12	raghukishtannavar261122@g	RAGHU KISTHANNAVAR	1KG22CS093	High	High	High	High	High	High	
3-9-2024 15:50:36	suhasmshandilya30@gmail.c	Suhas Madhusudan Shandilya	1KG2023CS405	High	High	High	High	High	High	
3-9-2024 15:57:24	rameshpraanav@gmail.com	Pranav Ramesh	1KG22CS088	High	High	High	High	High	High	
3-9-2024 15:57:27	tarunr200418@gmail.com	TARUN R	1KG22CS116	High	High	High	High	High	High	
3-9-2024 15:57:37	mohitreddy9342@gmail.com	Mohith reddy k	1KG22CS069	Medium	Medium	Low	Medium	Medium	Low	
3-9-2024 15:57:45	Devishpapani@gmail.com	Devish chowdary P	1KG22CS079	High	High	High	High	High	High	

3-9-2024 15:57:52	taanishraj732004@gmail.com	Taanish M	1kg22cs115	High	High	High	High	High	High	
3-9-2024 15:58:03	a18172649@gmail.com	T Abhishek	1kg22cs118	High	High	High	High	High	High	
3-9-2024 16:01:01	ramithak2003@gmail.com	Ramitha.K	1KG22CS098	High	High	High	High	High	High	
3-9-2024 16:02:18	tejaswinirm2004@gmail.com	Tejaswini RM	1KG22CS117	High	High	High	High	High	High	
3-9-2024 16:10:06	shashankmgoudar80@gmail.	Shashank m goudar	1kg22cs109	High	High	Medium	High	High	Medium	
3-9-2024 16:10:31	hamsuniveda@gmail.com	Niveda B	1KG22CS076	High	High	High	High	High	High	
3-9-2024 16:21:31	dheekshithnaidu001@gmail.c	M.dheekshith	1kg22cs072	High	High	High	High	High	High	
3-9-2024 16:34:32	Karthikeyanmonika23@gmail	Monika k	1KG22CS070	High	High	High	High	High	High	
3-9-2024 16:36:54	udaykiran990197@gmail.com	Udaykiran	1kg22cs121	High	High	High	High	High	High	
3-9-2024 17:36:53	sindhusk7019@gmail.com	Sindhushree k	1KG22CS111	Medium	Medium	Medium	Medium	Medium	Medium	
3-10-2024 20:59:07	vijay.shriyan@gmail.com	POOJITHA.S	1KG22CS085	High	Medium	High	Medium	High	Medium	
3-11-2024 9:20:55	pavanumesh221@gmail.com	Pavan u	1KG22CS082	High	High	High	High	High	High	
3-11-2024 9:22:41	srisuhasgowda24@gmail.com	Suhas S	1kg22cs113	High	High	High	High	High	High	
3-11-2024 10:45:01	sakesh28sk@gmail.com	Sakesh p	1kg22cs103	High	High	High	High	High	High	
3-11-2024 10:45:28	yashurockz01@gmail.com	YASHWANTH R	1KG22CS125	High	High	High	High	High	High	
3-11-2024 11:46:31	rajeshnaidu2848@gmail.com	Rajesh pc	1KG23CS403	High	High	High	High	High	High	
3-11-2024 12:24:35	mansii7584@gmail.com	Mansi M	1kg22cs065	High	High	High	High	High	High	
3-11-2024 12:53:14	shashankdurs77@gmail.com	Shashank D Urs	1kg22cs110	High	High	High	High	High	High	
3-12-2024 14:15:06	prapul.2004@gmail.com	Prapul u	1KG22CS089	High	High	High	High	High	High	
3-12-2024 17:27:40	ranicshinde123@gmail.com	Rani	1kg22cs100	High	High	High	High	High	High	
3-12-2024 17:27:52	1808194saran@gmail.com	Saran	1KG22CS108	High	High	High	High	High	High	
3-12-2024 17:58:24	poojashriyan0@gmail.com	POOJITHA.S	1KG22CS085	High	Medium	High	Medium	High	High	
3-12-2024 18:28:56	punithb165@gmail.com	Punith.B	1KG22CS091	High	High	High	High	High	High	
3-15-2024 12:41:58	santoshkumarnagur11@gma	Santosh Kumar nagur	1KG22C107	High	High	High	High	High	High	



Faculty In-Charge

HOD

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K. S. School of Engineering & Management, Bangalore - 560109

Department of Computer Science Engineering

Staff Feedback (2023-24) Odd Sem

Third Sem 'B' Section

Faculty Name: Mrs. Bindhu K P

Course Name & Code: Data Structures & Application /BCS304

Class Strength:66

Sl. No.	1. Effective Planning & organisation	2. Punctuality / Class time Utilization	3. Ability to teach /explain / effective use of board	4. Interaction / Motivating students	5. Subject knowledge	6. Presentation of the subject / communication	7. Linking subject with practical application	8. Syllabus covearage / exam point of view	9. Evaluation of test / counselling	10. Attitude towards students
1	5	5	5	5	5	5	5	5	5	5
2	5	5	5	5	5	5	5	5	5	5
3	5	5	5	5	5	5	5	5	5	5
4	5	5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5	5	5
6	5	5	5	5	5	5	5	5	5	5
7	5	5	5	5	5	5	5	5	5	5
8	5	5	5	5	5	5	5	5	5	5
9	5	5	5	5	5	5	5	5	5	5
10	5	5	5	5	5	5	5	5	5	5
11	5	5	5	5	5	5	5	5	5	5
12	5	4	5	4	5	4	4	4	4	4
13	5	5	5	5	5	5	5	5	5	5
14	5	5	5	5	5	5	5	5	5	5
15	5	4	5	4	5	4	5	4	5	4
16	5	5	5	5	5	5	5	5	5	5
17	5	5	5	5	5	5	5	5	5	5
18	5	5	5	5	5	5	5	5	5	5
19	5	5	5	5	5	5	5	5	5	5
20	5	5	5	5	5	5	5	5	5	5
21	5	5	5	5	5	5	5	5	5	5
22	5	5	5	5	5	5	5	5	5	5
23	5	5	5	5	5	5	5	5	5	5
24	5	5	5	5	5	5	5	5	5	5
25	5	5	5	5	5	5	5	5	5	5
26	5	5	5	5	5	5	5	5	5	5
27	5	5	5	5	5	5	5	5	5	5
28	5	5	5	5	5	5	5	5	5	5
29	4	5	4	5	3	4	5	4	5	4
30	5	5	5	5	5	5	5	5	5	5
31	5	5	5	5	5	5	5	5	5	5
32	4	5	4	4	4	4	3	4	4	4
33	5	5	5	5	5	5	5	5	5	5
34	5	5	5	5	5	5	5	5	5	5
35	5	4	5	4	5	4	5	4	5	4
36	5	5	5	5	5	5	5	5	5	5
37	5	5	5	5	5	5	5	5	5	5

38	5	5	5	5	5	5	5	5	5	5
39	5	5	5	5	5	5	5	5	5	5
40	5	5	5	4	5	5	5	5	5	5
41	5	5	5	5	5	5	5	5	5	5
42	5	5	5	5	5	5	5	5	5	5
43	5	5	5	5	5	5	5	5	5	5
44	4	5	5	4	5	5	4	4	5	5
45	5	5	4	4	4	4	5	5	5	5
46	5	5	5	5	5	5	5	5	5	5
47	4	4	4	4	4	4	4	4	4	4
48	5	5	5	5	5	5	5	5	5	5
49	4	4	4	4	4	4	4	4	4	4
50	5	5	5	5	5	5	5	5	5	5
51	5	5	5	5	5	5	5	5	5	5
52	5	5	5	5	5	5	5	5	5	5
53	5	5	5	5	5	5	5	5	5	5
54	5	5	5	5	5	5	5	5	5	5
55	5	5	5	5	5	5	5	5	5	5
56	5	5	5	4	4	4	5	5	5	5
57	5	5	5	5	5	5	5	5	5	5
58	5	5	5	5	5	5	5	5	5	5
Col. Total	285	285	285	280	283	281	284	282	286	283
Col. Avg.	4.91	4.91	4.91	4.83	4.88	4.84	4.90	4.86	4.93	4.88
Over all %	97.72									

Head of Department

HOD

Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

Principal