DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE FILE

Data Structures

(ESTD - 1999)

Vidya Jyothi Institute of Technology

(An AUTONOMOUS Institution)

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi,
Permanently Affiliated to
J.N.T. University, Hyderabad)

(Aziz Nagar, C.B.Post, Hyderabad -500075)

Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

REGULATION: R18

BATCH: 2018-22

ACADEMIC YEAR: 2019-2020

PROGRAM: B.Tech (COMPUTER SCIENCE AND ENGINEERING)

YEAR/SEM: II/I

COURSE NAME: Data Structures

COURSE CODE: A23504

PRE REQUISITE: C - Programming

COURSE COORDINATOR: B.Sailaja

COURSE INSTRUCTORS:

1. PKV Sarma
2. K.Srinivas
3. Y.Praveen Kumar

Course Coordinator

HOD-CSE

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE FILE INDEX

S.NO.	DESCRIPTION
1.	Syllabus
2.	Text Books & Other References
3.	Time Table
4.	Program Outcomes (PO's) , Program Specific Outcomes (PSO's) & PEO's
5.	Mapping of Course Outcomes (CO's) With Program Outcomes (PO's) & Program Specific Outcomes (PSO's)
6.	Academic Calendar
7.	Course Schedule
8.	Lesson Plan
9.	Assignment Questions
10.	Mid Question Papers I & II
11.	Unit Wise Questions
12.	Minutes of Course Review Meeting
13.	Lecture Notes
14.	Power Point Presentations
15.	Semester End Question Papers
16.	Extra Topics delivered (if any)
17.	Innovations In Teaching and Learning
18.	Assessment Sheet – CO Wise (Direct Attainment)
19.	Course End Survey Form

Syllabus

DATA STRUCTURES

B. Tech. II Year I Semester

L	T	P	C
3	0	0	3

Course Outcomes:

At the end of the course student would be able to

1. Understand the concepts of Stacks and Queues with their applications.
2. Analyze various operations on Binary trees.
3. Examine of various concepts of binary trees with real time applications.
4. Analyze the shortest path algorithm on graph data structures.
5. Outline the concepts of hashing, collision and its resolution methods using hash functions.

UNIT --I:

Data Structures: Introduction, Types of data structures, Static and Dynamic representation of data structure and comparison. **Stacks:** Stacks definition, operations on stacks, Representation and evaluation of expressions using Infix, Prefix and Postfix, Algorithms for conversions and evaluations of expressions from infix to prefix and postfix using stack.

Queues: types of Queues- Circular Queue, Deque and operations.

UNIT -- II:

Trees: Basic terminologies, Types of Binary Tree: Complete and Full Binary Tree, Extended Binary Trees, Representation of Trees using Arrays and Linked lists (advantages and disadvantages), Tree Traversal, Representation of Algebraic expressions, Threaded Binary Trees.

UNIT -- III:

Advanced concepts on trees: Representation and Creation of Binary Search Trees (BST), Operations on BST, Representation and advantages of AVL Trees, algorithms& operations on AVL Trees,Multi-way trees, Definition and advantages of B-trees, B+ Trees, Red-Black Trees.

UNIT -- IV:

Graphs-Basic terminology, Representation of graphs: sequential representation, Adjacency, Path Matrix) Linked representation. Graph Traversals-Breadth First Search, Depth First Search algorithms. Spanning Tree, Minimum Spanning Trees- Prim's Algorithm, Kruskals Algorithm, Dijkstra Algorithm.

UNIT --V:

Hashing: General Idea, Hash Functions, collisions, Collision avoidance techniques, SeparateChaining ,Open Addressing-Linear probing, Quadratic Probing, Double Hashing, Rehashing, Extensible Hashing, Implementation of Dictionaries.

Text Books:

1. Data Structures Using C, Second Edition ReemaThereja OXFORD higherEducation
2. Data Structures, A Pseudo code Approach with C, Richard F.Gillberg&Behrouz A. Forouzan, Cengage Learning, India Edition, Second Edition,2005.

Reference Books:

1. Data Structures, Seymour Lipschutz, Schaum's Outlines, Tata McGraw-Hill, Special Second Edition.
2. Data Structures Using C and C++, Aaron M. Tenenbaum, YedidyahLangsam and Moshe J. Augenstein PHI Learning Private Limited, DelhiIndia.
3. Fundamentals of Data Structures, Horowitz and Sahani, *Galgotia Publications* Pvt Ltd Delhi India.
4. Data Structure Using C, A.K. Sharma, Pearson EducationIndia.

Text Book & Other References

Text Books & other References

Text Books	
1	Data Structures Using C, Second Edition, ReemaThereja,OXFORD higher Education
2	Data Structures, A Pseudo code Approach with C, Richard F.Gillberg&Behrouz
Suggested / Reference Books	
1.	Data Structures, Seymour Lipschutz, Schaum's Outlines
2.	Data Structures Using C and C++I, Aaron M. Tenenbaum
3.	Fundamentals of Data StructuresI, Horowitz and Sahani
4.	Data Structure Using C, A.K. Sharma, Pearson Education India
Other Resources	
1	https://www.javatpoint.com/data-structure-tutorial
2	https://www.w3schools.blog/data-structure-tutorial
3	https://www.geeksforgeeks.org/data-structures/
4	https://www.programiz.com/dsa

Time Table

Vidya Jyothi Institute of Technology

Department of Computer Science and Engineering

Sec: **CSE-A**

Year/Sem: **II -I** W.E.F: **17/06/2019** ROOM NO: **C301**

DAY	9.00 – 10.00	10.00 – 11.00	11.00 – 12.00	12.00 – 12.45	12.45 – 1.45	1.45 – 2.45	2.45 – 3.45
MON	MFCS	EDC	PYTHON	LUNCH	DLD	P&S	DS(T)
TUE	EDC	DLD	PYTHON		DS	MFCS	P&S (T)
WED	DS	PYTHON	MFCS		EDC/DLD LAB		
THU	DLD	EDC	P&S		DS/PYTHON LAB		
FRI	P&S	PYTHON(T)	DLD		MFCS	EDC(T)	MC-I
SAT	EDC	MFCS(T)	MC-I		DS	P&S	DLD(T)

Subject		Name of the Faculty
P&S	Probability & Statistics	Ms. Fouzia
MFCS	Mathematical Foundations of Computer Science	Ms.Ch.Deepika
DS	Data Structures	Ms. B.Sailaja
PYTHON	Python Programming	Dr. B. Vijaya Kumar
DLD	Digital Logic Design	Ms. Radha Devi
EDC	Electronic Devices and Circuits	Ms.Renuka
DS LAB	DS&PYTHON LAB	Ms. B.Sailaja/ Dr. B. Vijaya Kumar/Ch.Deepika
EDC/DLD LAB	EDC/DLD LAB	Ms. Radha Devi/ Ms.Renuka
MC – I	Gender Sensitization	Ms. Y.Suneetha

Class Incharge Ms.CH.Deepika

II YEAR Coordinator Ms. B.Sailaja

Time Table I/C

H.O.D

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

Program Outcomes (Po's) & Program Specific Outcomes (Pso's)

Program Outcomes (PO's):

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization for the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools, including prediction and modelling to complex engineering activities, with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSO's):

PSO 1: The ability to design and develop Algorithms to provide optimized solutions for societal needs

PSO 2: Apply standard approaches and practices in Software Project Development through trending technologies

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY
Department of Computer Science and Engineering

Programme Educational Objectives (PEOs)

PEO1: Enhance the employablity of the graduates in Software industries/Public sector/Research organizations

PEO2: Acquire anlytical and computational abilities to pursue higher studies for professional growth

PEO3: Work in multidisciplinary project teams with effective communication skills and leadership qualities

PEO4: Develop professional ethics among the students and promote entrepreneurial abilities

Mapping of
Course Outcomes(CO's)
With
Program Outcomes(PO's)
&
Program Specific Outcomes(PSO's)

Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affliated JNTUH)

(An AUTONOMOUS Institution)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Year & Sem: II year I Sem

Course name: Data Structures

Course Code: A23504

Regulation: R18

COURSE OUTCOMES:

After completing this course the student must demonstrate the knowledge and ability to	
1.	Understand the concepts of Stacks and Queues with their applications
2.	Analyze various operations on Binary trees
3.	Examine of various concepts of binary trees with real time applications
4.	Analyze the shortest path algorithm on graph data structures.
5.	Outline the concepts of hashing, collision and its resolution methods using hash functions

CO –PO MAPPING:

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
CO 1	3	3	3	3	3	2	1	2	2	2	2	2
CO 2	3	3	3	3	3	2	2	2	2	3	2	3
CO 3	3	3	3	3	3	3	3	2	3	2	3	3
CO 4	3	3	3	3	3	2	2	3	2	2	3	3
CO 5	3	3	3	3	3	2	2	1	1	1	1	2
AVG	3	3	3	3	3	2.2	2.6	1.8	2	2	2.2	2.6

CO - PSO MAPPING:

	PSO1	PSO2
CO1	3	3
CO2	3	3
CO3	3	3
CO4	3	3
CO5	2	2
Avg	2.8	2.8

Course Coordinator

HOD-CSE

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

Academic Calendar

Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved by A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)
(Aziz Nagar, C.B.Post, Hyderabad -500075)

II/III/IV B.Tech I & II Semester Academic Calendar for the Academic Year 2019-20

II/III/IV YEAR I SEMESTER		Commencement of Class Work 17.06.2019	
	From	To	Duration
I Spell of Instruction	17.06.2019	10.08.2019	8 WEEKS
I Mid Examinations	13.08.2019	17.08.2019	4 DAYS
II Spell of Instruction	19.08.2019	05.10.2019	7 WEEKS
Dussehra Holidays	07.10.2019	19.10.2019	2 WEEKS
II Spell of Instruction Continuation	21.10.2019	26.10.2019	1 WEEK
II Mid Examinations	28.10.2019	31.10.2019	4 DAYS
Practical Examinations	01.11.2019	03.11.2019	3 DAYS
End Semester Examinations	04.11.2019	20.11.2019	2 WEEKS 3 DAYS
Betterment Examinations	21.11.2019	23.11.2019	3 DAYS
Supplementary Examinations	25.11.2019	07.12.2019	2 WEEKS
II/III/IV YEAR II SEMESTER		**Commencement of Class Work 09.12.2019**	
I Spell of Instruction	09.12.2019	10.01.2020	5 WEEKS
Sankranthi Holidays	11.01.2020	15.01.2020	5 DAYS
Technical/Sports fest	16.01.2020	18.01.2020	3 DAYS
I Spell of Instruction Continuation	20.01.2020	08.02.2020	3 WEEKS
I Mid Examinations	10.02.2020	15.02.2020	1 WEEK
II Spell of Instruction	17.02.2020	11.04.2020	8 WEEKS
II Mid Examinations	13.04.2020	17.04.2020	4 DAYS
Practical Examinations	18.04.2020	22.04.2020	4 DAYS
Betterment Examinations	23.04.2020	25.04.2020	4 DAYS
End Semester Examinations	27.04.2020	12.05.2020	2 WEEKS 2 DAYS
Supplementary Examinations	13.05.2020	30.05.2020	2 WEEKS 4 DAYS
Commencement of classes will be from	15.06.2020		

Vidya Jyothi Institute of Technology, C.B.Post, Hyderabad-75

DIRECTOR

Course Schedule

Course Schedule

Distribution of Hours in Unit – Wise

Unit	Topic	Book1	Book2	Total No. of Hours
I	Data Structures Introduction, Stacks,Applications of Stacks,Queues,Types of Queues	T1/43-49, 219-268	T2/10-24, 79-170	12
II	Trees-Basic Concepts, Representation of trees,Tree traversals	T1/279-314	T2/263-288	9
III	Advanced concepts on Trees-BST,AVL trees,Multi way Search Trees,Red black tree	T1/219-329	T2/299-375, 423-450	10
IV	Graphs-Introduction, Representation,Traversal, Spaaning Trees,MST	T1/383-414	T2/481-524	8
V	Hashing techniques, Collision resolution Techniques, Dictionaries	T1/466-485	T2/611-630	7
Total contact classes for syllabus coverage				46
	Assignment Tests : 02(Before Mid1 & Mid2 Examination)			

(B. Sailaja)

Lesson Plan

Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT: Data Structures **ACADEMIC YEAR:** 2019-20

NAME: B.Sailaja **YEAR/SEM/SECTION:** II B.TECH/I SEM

LESSON PLAN

S.No	Topic Name	Teaching Learning Process
	Unit-I	
1.	Data Structures: Introduction, Types of data structures	From Known to Unknown, Chalk & Board
2.	Static and Dynamic representation of data structure, comparison	Pendulum Method, Chalk & Board
3.	Stacks: Stacks definition, operations on stacks	From Known to Unknown, Chalk & Board
4.	Representation of Infix, Prefix and Postfix	Chalk & Board
5.	Algorithm to convert infix to postfix	Chalk & Board
6.	Evaluation of postfix expression	Chalk & Board
7.	Algorithm to convert infix to prefix	Chalk & Board
8.	Evaluation of prefix expression	Chalk & Board
9.	Queues: def, operations	From Known to Unknown, Chalk & Board
10.	Types-Deque- operations	Chalk & Board
11.	Circular queue operations	Chalk & Board
	Unit-II	
12.	Trees: Basic terminologies	PPT
13.	Types of Binary Tree: Complete and Full Binary Tree, Extended Binary Trees	PPT
14.	Trees using Arrays and Linked lists (advantages	Pendulum Method ,Chalk & Board

	and disadvantages)	
15.	Representation of Algebraic expressions	Pendulum Method ,Chalk & Board
16.	Representation of Tree Traversal	Chalk & Board
17.	Threaded Binary Trees	Chalk & Board
	Unit-III	
18.	Advanced concepts on trees: Representation and Creation of Binary Search Trees (BST)	Role Play, Chalk & Board
19.	Operations on BST	Chalk & Board
20.	Representation and advantages of AVL Trees	Chalk & Board, Seminars
21.	algorithms& operations on AVL Trees	Chalk & Board, Seminars
22.	Multi-way trees	Chalk & Board
23.	Definition and advantages of B-trees	Chalk & Board
24.	B+ Trees	Chalk & Board
25.	Red-Black Trees	Chalk & Board
	Unit-IV	
26.	Graphs-Basic terminology	From Known to Unknown ,Chalk & Board
27.	Representation of graphs: sequential representation, Adjacency, Path Matrix) Linked representation.	Pendulum Method ,Chalk & Board
28.	Graph Traversals-Breadth First Search	Pendulum Method ,Chalk & Board
29.	Depth First Search algorithms	Pendulum Method ,Chalk & Board
30.	Spanning Tree	Chalk & Board
31.	Minimum Spanning Trees- Prim's Algorithm	Chalk & Board
32.	Kruskals Algorithm	Chalk & Board
33.	Dijkstra Algorithm	Inquiry-based instruction, Chalk & Board
	Unit-V	
34.	Hashing: General Idea	From Inquiry-based instruction, Chalk

		& Board
35.	Hash Functions	Chalk & Board
36.	Collisions	Chalk & Board
37.	Collision avoidance techniques, Separate Chaining	Chalk & Board
38.	Open Addressing-Linear probing, Quadratic Probing, Double Hashing	Chalk & Board
39.	Rehashing	Chalk & Board
40.	Extensible Hashing	PPT
41.	Implementation of Dictionaries.	Chalk & Board

Course Coordinator

HOD-CSE
Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-500075.

Assignment Questions

Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ASSIGNMENT 1

Branch: CSE **Year&Sem:** II-I

SUB: Data structures **Academic Year:** 2019-20

Faculty Name: B.Sailaja **Marks:** 25M

S.No	Question	Marks	CO	BL	PO's
1	What is a Data Structure? Explain the representation of Data Structure.	5	1	L2	1-12
2	Convert the following expression A+(B*C)-((D*E+F)/G) into post fix form	5	1	L4	1-12
3	Explain tree traversals with example?	5	2	L2	1-12
4	What is a Binary Tree? Explain the representation of Binary tree	5	2	L1,L2	1-12
5	Define binary search tree. Explain the insertion and deletion in BST	5	3	L1,L2	1-12

Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

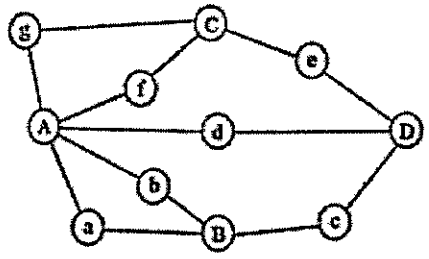
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ASSIGNMENT 2

Branch: CSE **Year&Sem:** II-I

SUB: Data Structures **Academic Year:** 2019-20

Faculty Name: B.Sailaja **Marks:** 25M

S.No	Question	Marks	CO	BL	PO's
1	Explain the various rotations of AVL Trees maintaining balance factor while deletion takes place.	5	3	L2	1-12
2	Define Graph and explain how graphs can be represented in adjacency matrix and adjacency LIST.	5	4	L1,L2	1-12
3.	Illustrate DFS and BFS traversals of following graph 	5	4	L4	1-12
4.	Define hashing and discuss the different hashing functions with an example.	5	5	L1,L2	1-12
5.	Analyze input (371, 323, 173, 199, 344, 679, 989) and hash function h(x)=x mod 10, Show the result using quadratic probing, and double hashing h2(x)=7 - (x mod 7)	5	5	L5	1-12

Mid Question Papers I & II

Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)
(Aziz Nagar, C.B.Post, Hyderabad -500075)

II B.Tech I Semester I Mid Examination	
Branch: CSE	Duration: 90Min
Sub: Data Structures	Marks: 20
Date: 14.08.2019	Session: AN

Course Outcomes:

1. Understand the concepts of Stacks and Queues with their applications.
2. Analyze various operations on Binary trees.
3. Examine of various concepts of binary trees with real time applications.
4. Analyze the shortest path algorithm on graph data structures.
5. Outline the concepts of hashing, collision and its resolution methods using hash functions.

Bloom's Level:

Remember	I	Apply	III	Evaluate	V
Understand	II	Analyze	IV	Create	VI

Q.N	ANSWER ALL THE QUESTIONS	Marks	CO	PO	BL
	PART-A (3Q×2M =6 Marks)				
1	Define data structures.	2	1	1-12	L1
2	Define the node, siblings, left-skewed tree.	2	2	1-12	L1
3	Construct the binary search Tree for the below given data 35, 20, 15, 31, 89,-1, 35, 45.	2	3	1-12	L6
	ANSWER ALL THE QUESTIONS				
	PART-B (5+5+4= 14 Marks)				
4 i.a)	List out the algorithm steps for a stack insertion operation.	3	1	1-12	L4
b)	Write the differences between linear and non linear data structures.	2	1	1-12	L1
	OR				
ii.a)	Convert infix expression into its equivalent prefix expression: A*(B+D)/E- F*(G+H/K).	3	1	1-12	L5
b)	Write short notes on different types of queues.	2	1	1-12	L1
5.i a)	What are the advantages and disadvantages of Static and Dynamic representation of a binary tree	2	2	1-12	L1
b)	Given In order traversal of a binary tree is D,B,H,E,I,A,F,J,C,G and post order traversal is D,H,I,E,B,J,F,G,C,A. Construct binary tree and find the pre order traversal.	3	2	1-12	L6
	OR				
ii. a)	Explain tree traversals with example.	2	3	1-12	L3
b)	Define and explain the following: A) One way threaded Binary Tree. B) Full Binary Tree	3	3	1-12	L3
6.i)	Explain the insertion and searching operation on binary search trees with an example	4	4	1-12	L5
	OR				
ii)	Construct a binary search tree by inserting following elements 20, 70, -1, 66, 11, 55, 90, 45,100,110,91 and delete 66,100	4	4	1-12	L5

Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)
(Aziz Nagar, C.B.Post, Hyderabad -500075)

II Year B.Tech I Semester II Mid Examination

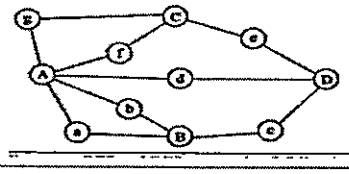
Branch: CSE	Duration: 90Min
Sub: Data Sturctures	Marks: 20
Date: 30.10.2019	Session: AN

Course Outcomes:

1. Understand the concepts of Stacks and Queues with their applications
2. Analyze various operations on Binary trees.
3. Examine of various concepts of binary trees with real time applications
4. Analyze the shortest path algorithm on graph data structures
5. Outline the concepts of hashing, collision and its resolution methods using hash functions.

Bloom's Level:

Remember	I	Analyze	IV
Understand	II	Evaluate	V
Apply	III	Create	VI

	PART-A (3Q×2M =6 Marks)	Outcomes		BL	Marks
	ANSWER ALL THE QUESTIONS	CO	PO		
1.i)	Compare B-tree and B+ tree	3	1-12	2	2
	[OR]				
ii)	Define balance factor and what the balance factor value of avl tree is	3	1-12	1	2
2.i)	Define weighted graph give one example.	4	1-12	1	2
	[OR]				
ii)	Define Minimum Spanning tree with an example.	4	1-12	1	2
3.i)	Define Hashing and hash table	5	1-12	1	2
	[OR]				
ii)	Define dictionaries.	5	1-12	1	2
	PART-B (4+5+5= 14 Marks)	Outcomes		BL	Marks
	ANSWER ALL THE QUESTIONS	CO	PO		
4.i.	Construct AVL Tree with the following elements C,O,M,P,U,T,I,N,G and remove the elements P , U and T	3	1-12	3	4
	[OR]				
ii.a)	What are the advantages of B-trees?	3	1-12	1	2
b)	Differentiate between Binary Search Tree and an AVL Tree	3	1-12	4	2
5.i.a)	Define Graph and explain how graphs can be represented in	4	1-12	2	3
b)	Define in degree, out degree for a graph.	4	1-12	1	2
	[OR]				
ii.a)	 Traverse the graph using BFS and DFS	4	1-12	3	3
b)	List the applications of Graphs	4	1-12	1	2
6.i.a)	Discuss any 2 different hashing functions with an example	5	1-12	2	3
b)	List the uses of hash table	5	1-12	1	2
	[OR]				
ii.	Write a C program to implement Quadratic probing	5	1-12	3	5

Unit Wise Questions

UNIT – I

Short Answers Type Questions

1. **Define** data structure? **Discuss** types of Data structures?
2. **List** linear and nonlinear data structures
3. **List** the operations performed in the Linear Data Structure.
4. **List** the applications of stack?
5. **Distinguish** between static and dynamic representation?
6. **Define** stack?
7. **Define** circular queue?
8. **Define** prefix and infix?
9. **Define** postfix expression?
10. **List** types of Queue?
11. **Define** dequeue?
12. **Define** input restricted queue?
13. **State** the difference between queue and circular queue?
14. **State** operations on stack?
15. **State** overflow and underflow conditions of stack?
16. **Describe** output restricted queue?
17. **List** the ?
18. **State** the rules to be followed during infix to postfix conversions
19. **Convert** the infix expression (a+b)-(c*d) into post fix form
20. **Define** overflow and underflow conditions of Queue

Broad Answers Type Questions

1. **a) Define** Data structure? State and explain the representation of data strcuture?
   b) Explain the differences between static and dynamic representation?
2. **Explain** the operations of stack with example?
3. Convert the expression ((A + B) * C - (D - E) * (F + G)) into post fix form
4. Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 | 3 +
5. Explain the operations of Dequeue with example.

UNIT – II

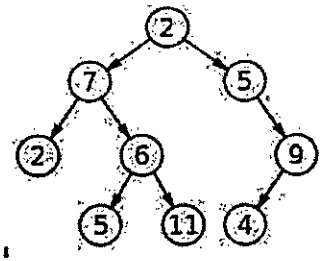
Short Answers Type Questions

1. **Define** tree.Give example
2. **Write** the properties of tree?
3. **Define** the terms node, degree, siblings, depth?
4. **Distinguish** between tree and binary tree?
5. **Define** path?
6. **Define** Binary tree?
7. **List** the applications of Trees
8. **State** the properties of a Binary Tree
9. **Define** full binary tree?
10. **List** types of trees?

11. **List** the different traversing techniques of a binary tree?
12. **Define** an Expression tree?
13. **Draw** an expression tree for the following expression A*(B+D)/E-F*(G+H/K)? ?
14. **Define** complete binary tree
15. **Distinguish** between complete binary tree and full binary tree?
16. **Define** threaded binary tree?
17. **State** applications of trees?
18. **Write** the advantages and disadvantages of array representation?
19. **Define** a right-skewed binary tree and Left-skewed binary tree.?
20. **What** is a tournament tree?

Broad Answers Type Questions

1 **Explain** Binary tree ADT and its representation
2 Write inorder, preorder, post order traversal of the following tree



3 Define an expression tree for the following expression ((A + B) * C - (D - E) * (F + G)) and write the different tree traversals for above expression
4 Define threaded binary tree? Explain the impact of such a representation on the tree traversal procedure?
5 Explain tree traversals with example.

UNIT – III

Short Answers Type Questions

1. **Define** balanced search tree?
2. **State** the operations on binary search tree?
3. **List** the drawbacks of a binary search tree?
4. **Distinguish** binary tree and binary search tree?
5. **Define** balance factor and what the balance factor value of AVL tree is?
6. **List** the different AVL tree rotations to insert a node?
7. **Discuss** the drawbacks of AVL trees?
8. **List** the properties of B-Trees?
9. **Explain** the L-R rotation of an AVL tree with example?
10. **Define** B-tree with example?
11. **List** the properties of B-Trees?
12. **What** are the advantages of B-trees
13. **List** the properties of red black Trees?
14. **Distinguish** between B tree and B+ tree?

15. **Define** M-way tree?
16. **List** the different operations on B-Trees?
17. **Explain** the R-0 rotations in AVL tree?
18. **Construct** BST for 35,20,46,82,49,20,65?
19. **Explain** L-1 rotations in AVL Tree?
20. **Define** red black tree with example ?

Broad Answers Type Questions

1.**Explain** deletion operation of binary search tree with example?

2. **Explain** various rotations of AVL Trees maintaining balance factor while insertion takes place.

3 **Construct** the binary search Tree for the below given data. P, F, B, H, G, S, R, Y, T, W, Z. and delete B, H.

4 **Insert** the following elements into an empty AVL Tree20, 15, 5, 10, 12,17,25,19.?

5 **Define** B-tree, B+ tree? What are the advantages and disadvantages of B-tree? Explain with an example.?

UNIT – IV

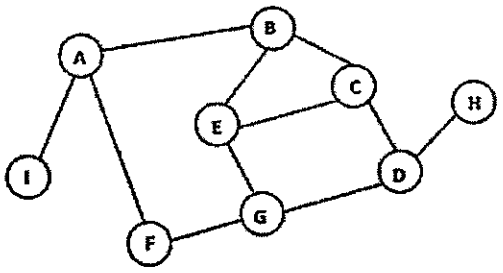
Short Answers Type Questions

1. **Define** graph?
2. **Explain** the Adjacency matrix representation of graph?
3. **Explain** linked representation of the graph?
4. **Distinguish** between BFS and DFS?
5. **Define** weighted graph?
6. **List** the tree traversal?
7. **Discuss** about connected and disconnected graph?
8. **Describe** cycle and path?
9. **Define** spanning tree?
10. **Write** the basic properties of a spanning tree?
11. **List** the applications of Graphs?
12. **Differentiate** between graph and tree?
13. **Explain** in degree and out degree of a directed graph ?
14. **Define** minimum spanning?
15. **Define** directed graph?
16. **Describe** source and sink?
17. **Define** sub graph.?
18. **What** is an isolated vertex?
19. **Write** the algorithms to find spanning tree in the graph?
20. **List** various algorithms to find minimum spanning tree?

Broad Answers Type Questions

1. Explain different ways representation of graphs?

2. Explain DFS graphs traversal algorithms with suitable example?
3. Illustrate DFS and BFS traversals of following graph



4. Define Minimum Spanning tree? Explain Kruskal's Algorithm with exampl
5. Explain DIJKSTRA Algorithm with example

UNIT – V

Short Answers Type Questions

1. **Define** Hashing?
2. **Compare** the time complexities of binary search, linear search, hashing?
3. **Define** Hash Function and hash table?
4. **List** different types of popular hash functions?
5. **Explain** mid square method with example?
6. **Write** the properties of a good hash function ?
7. **Define** Separate Chaining?
8. **Describe** folding method?
9. **Describe** open addressing?
10. **List** the uses of hash table?
11. **Define** Collision?
12. **Define** probe?
13. **State** different types of collision resolving techniques?
14. **Describe** extendible hashing?
15. **List** advantages of hashing?
16. **List** the different types of open addressing methods ?
17. **Describe** rehashing.
18. **Write** disadvantages of rehashing every time?
19. **State** advantages of extendible hashing?
20. **Define** dictionary?

Broad Answers Type Questions

1. Define hashing and discuss the different hashing functions with an example
2. Define collision and discuss any two collision resolution techniques?
3. Use quadratic probing to fill the Hash table of size 11. Data elements are 23,0,52,61,78,33,100,8,90,10,14.?
4. **Explain** Extendable hasing with an example?
5. Analyze input (371, 323, 173, 199, 344, 679, 989) and hash function h(x)=x mod 10, Show the result using quadratic probing, and double hashing h2(x)=7 - (x mod 7)

Minutes of Course Review Meeting

Meeting 1

Date: 16/07/2019

Details of Meeting No – 1	
Date of Meeting	16/07/2019.
Member's Present	1. Ms. B.Sailaja 2. Mr PKV Sarma 3. Mr. K.Srinivas 4. Mr.Y.Praveen Kumar
Details	Points discussed in the meeting: • Preparation of Unit wise questions and give assignment to students • Discussion of Teaching Learning Practices • Status of Syllabus coverage of Mid I and instructions to complete the syllabus
Signatures	1. 2. 3. 4.

Course Coordinator

CSE-HOD

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

Meeting 2

Date: 25/09/2019

Details of Meeting No – 2	
Date of Meeting	25/09/2019
Member's Present	1 Ms. B.Sailaja 2. Mr PKV Sarma 3. Mr. K.Srinivas 4. Mr.Y.Praveen Kumar
Details	• Preparation of Unit wise questions and give assignment to students • Status of Syllabus coverage of Mid II and instructions to complete the syllabus
Signatures	1. 2. 3. 4.

Course Coordinator

CSE-HOD

Head of the Department
Computer Science and Engineering
VJIT, Hyderabad-50075.

Lecture Notes

other end called the front. It can be implement using arrays or linked lists.

Initially front = NULL = -1 and rear = NULL = -1

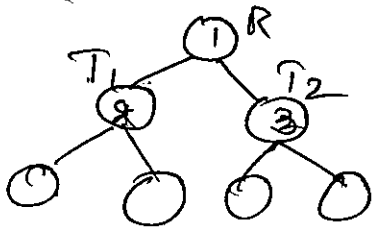
Before inserting an element in the queue, overflow condition has to check. When a queue is full overflow occures

i.e. rear = MAX-1

- Before deleting also under flow condition has to check. i.e. if front > rear or front = rear = -1.

Trees:- A tree is a non linear DS which consists of a collection of nodes arranged in a hierarchical order. One of node is designated as the root node, & the remaining nodes can be partitioned into disjoint sets such that each set is a sub-tree of the root.

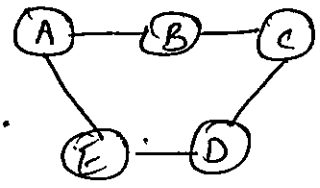
A binary tree consists of a root node & left and right sub-trees, where both sub-trees are also binary trees.



Adv:- provides quick search, insert & delete.

dist :- complicated.

Graph :- A graph is a non-linear DS which is a collection of vertices & edges that connect these vertices.



Best Models for
Adv :- Real world situations

Disadv :- Complex.

Static & Dynamic representation of DS :-

DS can be represented in two catagories.

1. Static DS :- Static data structure is storing the data using array i.e. Memory allocation is fixed it is not changed at the runtime.

Limitations :- 1) fixed size i.e. if more elements are there we can't add.

2. Memory wasted :- If we are allocating more memory & the data is not there then it is wasted.

Due to this the DS can be used Dynamic representation.

2. Dynamic :- Using this when new element has to store it will allocate the memory & if it is not

needed it will discard. This can be implemented using pointers.

Strings:- A string is a null-terminated character array, ie after the last character, a null character ('\0') is stored to signify the end of the character array.

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

- The string can be read in 3 ways.
  1) Using scanf — scanf("%s", str);
  2) Using gets() gets(str)
  3) Using getchar(), getch() or getche()
     ↓ read a sequence of single characters
     ↓ in this null character has to append.

Eg:-
```
ch = getchar();
while (ch != '*')
{ str[i] = ch;
  i++
  ch = getchar();
}
str[i] = '\0';
```

The string can be displayed in 3 ways.

1. Using printf()
2. Using puts()
3. Using putchar().

printf("%.s", str);

printf("%5.3s", str) prints 3 characters in total of five characters.

puts() → puts(str)

→ i=0
while (str[i] != '\0')
{
putchar(str[i]); print the characte
i++
}.

Operations on String:-

Strlen:- Gives the length of the string.
Syn:- strlen(string).

Strcpy:- Copies a string from source to destination
$ strcpy(s2, s1).
destination. source
s1 is copied to s2.

Strcmp:- Compares characters of two strings.
strcmp(source, target)
If same returns zero. otherwise returns numeric difference b/w the ASCII values.

Strcat:- Appends source string to destination str.
strcat(t1, t2).
t2 is append to t1.

∴ **Strrev** :- Reverses all characters of a string.

```
strrev(string);

Void main()
{
   char str1[20], str2[10], str3[5] = "Hai";
   int l
    printf("enter string1");
      gets(str1);
    printf("enter string2");
     scanf("%s", str2);
      l = strlen(str1);
    printf("length of first string %d", l);
     printf("reverse of string1", strrev(str1));
      strcat(str1, str3);
    printf("%s", str1);
     if(strcmp(str1, str2))
      printf("strings are same");
     else
      printf("strings are not equal");
      strcpy(str1, str2);
     printf("%s", str1);
    getch();
}
```

Pointers & Strings:-

```
char str[10] = {'H', 'i', '\0'};
char str[10] = "Hi";
```

When a string is declared, the compiler sets aside a contiguous block of the memory to hold the characters. ~~& initializes its first four.~~

```
Void main()
{
    char str[] = "Hello";
    char *pstr;
    pstr = str;
    printf(" string is");
    while (*pstr != '\0')
    {
        printf("%c", *pstr);
        pstr++;
    }
}
```

→ puts(pstr)

- ~~Pstr~~ The pointer pstr points to str.
- when puts(str) given it passes the address of str[0] to pstr.

Stack :- Stack is a linear data structure in which elements are added and removed only from one end, which is called the "Top". Hence a stack is called LIFO.

Eg:- Function calls.

Operations :-

A stack performs three basic operations

1) push : adds an element to the top of the stack

2) pop : removes the element from the top of the stack.

3) peek : returns the value of the topmost element of the stack.

Stack can be implemented in two ways

1) Using Arrays 2) Using linked list.

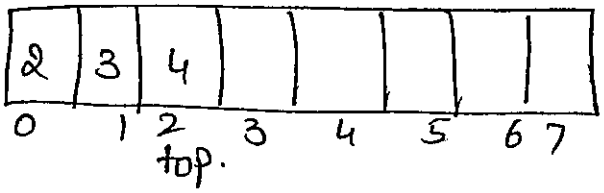
Array representation :-

- Stacks can be represented as a linear array. - Every stack has a variable called Top associated with it.
- Top is the position where element will be added to or deleted from.

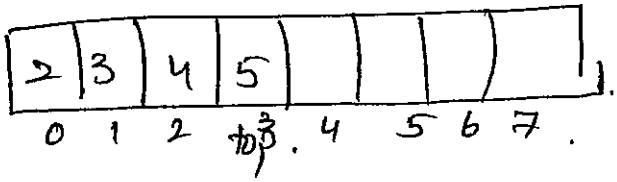
There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.

- If top = NULL, indicates that the stack is empty.
- If top = Max-1, indicates stack is full.

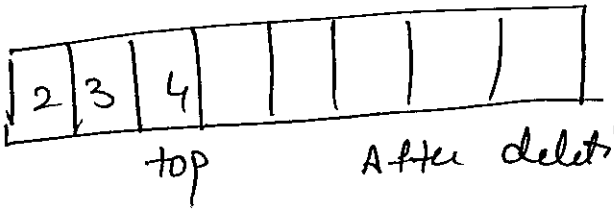
Push :— Before inserting the value, it check if top = Max-1. i.e overflow condition.



to insert 5 it check top = Max-1, if condition false increments top & store the new element.



Pop :- Before deleting it check if top = NULL i.e. underflow condition.



Peek :- Befor it checks top = NULL, else. print the top of the value.

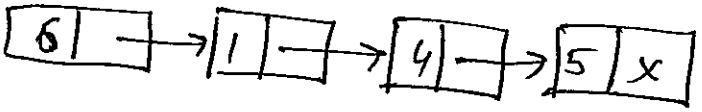
```
#include < stdio.h>
#define MAX 10
int st[MAX], top=-1;
void push();
int pop();
int peek();
void display();
void main();
{
  int option, val;
  do
  {
    printf("1. Push \n");
    printf("2. pop \n");
    printf("3. peek \n");
    printf("4. display \n");
    printf("5. Exit");
    printf("enter ur choice");
    scanf("%d", &option);
  } switch(option)
  {
  case 1: push();
          break;
  case 2: val=pop();
          pf(" element poped is %d", val);
          break;
  case 3: val = peek();
          pf("the top value %d", val);
          break;
  case 4: display();
          break;
  case 5: exit(0);
  }
} while(option!=5);
getch();
}
void push()
{ int ele;
  printf("enter ele to push");
  scanf("%d", &ele);
    if(top == MAX-1)
      pf("overflow");
    else
    {
      top++;
      st[top]= ele;
    }
}
int pop()
{ int ele;
  if(top == -1)
    pf("underflow");
  else
  {
    ele =st[top];
    top--;
    return ele;
  }
  return 0;
}
```

Linked Representation of Stack :-

The draw back with array is that the array must be declared to have some fixed size.

In a linked stack every node has two parts - 1) stores the data
2) stores the address of next node.

- The start pointer of the linked list is used as TOP.
- All insertions & deletions are done at the node pointed by TOP.
- If TOP = NULL., indicates stack is empty.



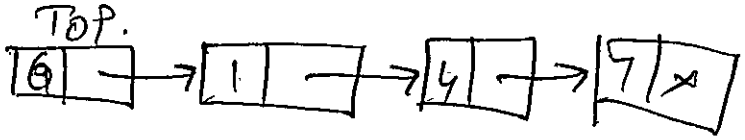
Operations on a linked stack :—

1) Push :— It is used to insert an element into the stack.

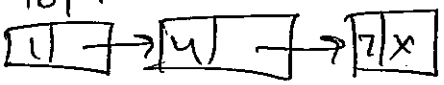
First check if top = NULL, if it is then new node will be called as TOP.

If top != NULL then insert the new node at the begining of stack & name this node as top.

2) Pop operation :— It is used to delete the topmost element from a stack. Before deleting the value, check if Top = NULL, then it means that stack is empty & no more deletions can be done.



If Top != NULL then delete the node pointed by Top & make Top point to the second element of the linked stack.




```
#include<stdio.h>
struct Stack
{ int data;
  struct Stack *next;
};
struct stack *Top=NULL;
struct stack s. s. *temp.
 push( )
{
  struct stack *n;
  n= (struct stack *)malloc( );
   printf("enter the data");
   scanf("%d", n->data);
   if (Top==NULL)
   {
   n->next = NULL;
     top = n;
   }
   else
   {n->next = top;
     top=n;
   }
}
   POP( ) if (TOP==NULL) Pf("empty");
   { else
   {temp=top;
    top=top->next;
    Pf("data deleted%d, temp->data);
    free(temp);
   }.
                }
```

Applications of Stack :-

1. Reversing a list
2. Parentheses checker
3. Conversion of an infix exp into a postfix exp
4. Evaluation of a postfix exp
5. Conversion of an infix exp into a prefix exp.
6. Evaluation of a prefix exp
7. Recursion.
8. Tower of Hanoi.

Arithmetic Exp :-

There are three diff notations to write A.E.

1. Infix 2. Prefix 3. Postfix.

Infix :- The operator is placed in-b/w the operands.

Eg:. A+B

- It is easy to write but comp find it difficult to parse. So comp work more efficiently with exp's written using Prefix & Postfix notations.

Postfix :- It is also known as polish ~~Parenthesis~~ notation. In this the operator is placed after the operands.

Eg:- AB+.

The order of evaluation of a postfix exp. is

always from left to right.

$(A+B)*C$ — $AB+C*$. $(A*B)+(C*D)$.

Eg:- $(A-B)*(C+D)$ Eg:- $(A+B)/(C+D)-D*E$

$AB-CD+*$. $AB+CD+/DE*-$

Postfix :— In this the operator is placed before the operands.

Eg:- $+AB$

- While evaluating the operators are applied to the opends that are present immediately on the right of the operator.

Eg:- $(A+B)*C$. 2) $(A-B)*(C+D)$

$*+ABC$ $*-AB+CD$

3) $-/+AB+CD.*DE$

- Conversion of an Infix into a Postfix :—

Infix exp may contain parentheses, operands & operators.

The precedence of operators can be

high prio — $*, /, \%$

Low prio — $+, -$

Eg:- $A+B*C$ — 1st $B*C$ then A is added.

if have $(A+B)*C$ — 1st $A+B$ then C is multiplied

To convert from infix to postfix it uses

stack to temporarily hold operators. The postfix exp is obtained from left to right.

Step 1: Add ")" to the end of the infix exp.

Step 2: Push "(" on to the stack.

Step 3: Repeat until each character in the infix is scanned.

a) If a "(" is encountered, push it on to the stack.

b) If an operand (digit or char) is encountered, add it to the postfix exp.

c) If a ")" is encounted then

i) Repeatedly pop from stack & add it to the postfix exp until a "(" is encountered

ii) Discard the "(".

d) If an operator op is encountered then

i) Repeatedly pop from stack & add each operator to the postfix exp which has the same precedence or a higher precedence than op.

ii) Push the operator op to the stack.

Step 4: Repeatedly pop from the stack & add it to the postfix exp until the stack is empty

Step 5: Exit.

Eg:- A - (B / C + (D % E * F) / G) * H ;

Infix character	Stack	Post fix exp
	(	
A	(	A
-	(-	A
(	(- (	A
B	(- (	AB
/	(- (/	AB
C	(- (/	ABC
+	(- (+	ABC/
(	(- (+ (	ABC/
D	(- (+ (	ABC/D
%	(- (+ (%	ABC/D
E	(- (+ (%	ABC/DE
*	(- (+ (% *	ABC/DE
F	(- (+ (% *	ABC/DEF
)	(- (+	ABC/DEF*%
/	(- (+ /	ABC/DEF*%
G	(- (+ /	ABC/DEF*%G
)	(- (	ABC/DEF*%G/+
*	(- *	ABC/DEF*%G/+
H	(- *	ABC/DEF*%G/+H
)	(-	ABC/DEF*%G/+H*-

Eg:- A + B - C * D.

```
#include <stdio.h>
#include <ctype.h>
#define MAX 100.
char st[MAX];
int top=-1;
void push(char st[], char);
char pop(char st[]);
void infixtopostfix(char source[], char target[]);
int getpriority(char);
int main()
{
  char infix[100], postfix[100];
  clrscr();
  printf("enter infix exp");
  gets(infix);
  strcpy(postfix,"");
  infixtopostfix(infix, postfix);
  printf("postfix exp is");
  puts(postfix);
  getch();
  return 0;
}
void infixtopostfix(char source[], char target[])
{ int i=0, j=0;
  char temp;
  strcpy(target,"");
  while(source[i]!='\0')
  {
    if(source[i]=='(')
    {
      push(st, source[i]);
      i++;
    }
    else if(source[i]==')')
    {
      while((top!=-1)&&(st[top]!='('))
      {
        target[j] = pop(st);
        j++;
      }
      if(top==-1)
      {
        pf("incorrect exp");
        exit(1);
      }
      temp = pop(st);
      i++;
    }
    else if(isdigit(sour[i]) || isalpha(sour[i]))
    {
      target[j] = source[i];
      j++;
      i++;
    }
    else if(source[i]=='+' || sour[i]=='-'
            || sour[i]=='*' || s[i]=='/'
            || s[i]=='%')
    {
      while((top!=-1) && (st[top]!='(')
            && (getpriority(st[top])
                >getpriority(sour[i])))
      {
        target[j] = pop(st);
        j++;
      }
```

```
        push(st, source[i]);
        i++;
    }
    else
    {
        pf("incorrect exp");
        exit(1);
    }
}
while((top!=-1)&&(st[top]!='('))
{
    target[j] = pop(st);
    j++;
}
target[j] = '\0';
}
int getpriority(char op)
{
    if(op=='/'||op=='*'||op=='%')
        return 1;
    else if(op=='+'||op=='-')
        return 0;
}
void push(char st[], char val)
{
    if(top==MAX-1)
        printf("overflow");
    else
    {
        top++;
        st[top] = val;
    }
}
```

```
char pop(char st[])
{
    char val=' ';
    if(top==-1)
        pf("underflow");
    else
    {
        val = st[top];
        top--;
    }
    return val;
}
```

(a+b) * (c-d) / e

(A-2) * (B+c) - (D * E)

Evaluation of a postfix Exp :-

Using stacks, any postfix exp can be evaluated very easily. Every character of the postfix exp is scanned from left to right.

If the character encountered is an operand it is pushed on to the stack.

If an operator is encountered then the top two values are popped from the stack & the operator is applied to these values. The result is then pushed on to the stack.

Alg Step 1:- Add a ")" at the end of postfix exp

Step 2: Scan every character of the postfix exp & repeat steps 3 & 4 until ")" is encountered

Step 3: If an operand is encountered, push it on the stack.

If an operator op is encountered, then.

a) pop the top two ~~values~~ elements from the stack as A & B as A & B.

b) Evaluate B op A., where A is the topmost ele & B is the ele below A.

c) Push the result of evaluation on to stack.

Step 4: Set result equal to the topmost elem of stack.

Step 5: Exit

9- ((3*4)+8)/4. → 934*8+4/— .

character scanned	stack.	
9	9	
3	9,3	
4	9,3,4	
*	9,12	3*4.
8	9,12,8.	
+	9,20,	12+8
4	9,20,4	
/	9,5	20/4
—	4	9-5

```
#include<ctype.h>
#define MAX 100.
float st[MAX];
int top=-1;
void push(float st[],float val);
float pop(float st[]);
float evaluatepostfixexp(char exp[]);
main()
{
    float val;
    char exp[100];
    clrscr();
    ptl("enter postfix exp");
    gets(exp);
    val=evaluatepostfixexp(exp);
    Printf("Peal of postfix %.2f", val);
    getch();
}
float evaluatepostfixexp(char exp[])
{
    int i=0;
    float op1, op2, value;
    while(exp[i]!='10')
    {
    if(isdigit(exp[i]))
        push(st, float(exp[i]-'0'));
    else
    {
        op2=pop(st);
        op1=pop(st);
        switch(exp[i])
        {
        Case '+': value=op1+op2;
```

```
        break;
case '-' : value = op1-op2;
           break;
case '/' : value = op1/op2;
           break;
case '*' : value = op1*op2;
           break;
case '%' : value = (int)op1 %(int)op2;
           break;
    }
    push(st, value);
    }
    i++
}
 return (pop(st));
}
void push( float st[], float val)
  {  if(top==MAX-1)
        pf("overflow");
     else
      {  top++;
         st[top] = val;
      }
  }
  float pop(float st[])
    {  float val=-1;
       if(top==-1)
            pf("underflow").
```

```
else
{   val = st[top];
    top--;
}
return val;
}.
```

Eg:- 2+3*4.

6 2 / 3 - 4 2 * +.

2 5 7 * 14 - 6 +

Conversion from Infix to Prefix exp:—

Step 1:- Reverse the infix string. While reversing the string must interchange left & right parantheses

Step 2: Obtain postfix exp

Step 3: Reverse the postfix exp to get prefix exp.

Eg:- (A - B / C) * (A / K - L)

1) (L - K / A) * (C / B - A).

2) (LKA/-) * (CB/A-)

LKA/-CB/A-*.

3) *-A/BC-/AKL

Evaluation of a prefix Exp:—

Step 1:- Accept the prefix exp.

Step 2:- Repeat until all the characters in the prefix exp have been scanned.

a) Scan the prefix exp from right, one character at a time.

b) If the scanned character is an operand, push it on the operand stack.

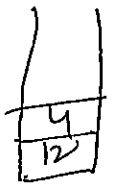
c) If the scanned character is an operator, then i) pop two values from the operand stack.

ii) Apply the operator on the popped operands

iii) push the result on the Operand Stack.

~~Prec~~ ~~+~~ 12. 7 * 8 / 4 12

Character scanned	operand stack
12	12
4	12, 4
/	3 | 12/4
8	3, 8
*	24
7	24, 7
2	24, 7, 2
~~—~~	24, ~~7 2~~ 5
~~+~~	~~24 5~~ 29.
~~out~~	



4

— + 8 / 6 3 2.

2	2
3	23
6	236
/	2 2 6/3
8	2 2 8.
+	2 10
—	8. 10-2

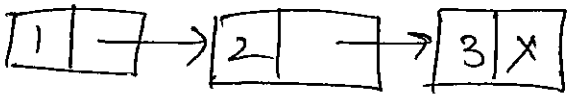
— + 7 * 4 5 + 20

0	0
2	02
+	2
5	2 5
4	2 5 4
*	2 20
7	2 20 7
+	2 27
—	25.

Linked List :-

It is a linear collection of data elements. These are called nodes.

- Every node contains two parts, an integer and a pointer to the next node. The last node store a special value called NULL.



- To create a node.

```
struct node
{int data;
 struct node *next;
};
```

Single Linked list :-

- A SLL is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

Operations on SLL :-

1. Insertion :- A new node is added into a linked list it it is already created.

1) Insertion at beginning
2) Insertion at ending
3) Insertion after a given node
4) Insertion before a given node.

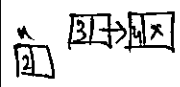
2) Deletion :-

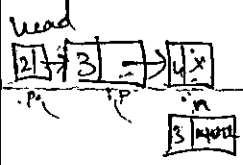
1) Deleting a first node
2) Deleting a last node
3) Deleting after a given node.

3) Search

4) Traverse.

```
insert (struct node *n)
{ int op
  printf("enter where to insert");
  pf("1. At begin 2. At end
      3. After given node");
  sf("%d", &op);
  switch(op)
  {
  case 1: n->next = head;
          head = n;
          break;
  case 2: n->next = NULL;
          for(p=head; p->next != NULL; p=p->next);
          p->next = n;
          break;
  case 3: pf("enter after which node");
          sf("%d", &val);
          p = head;
          while(p->data != val)
          {
            p = p->next;
          }
          n->next = p->next;
          p->next = n;
          break;
  }
}
```

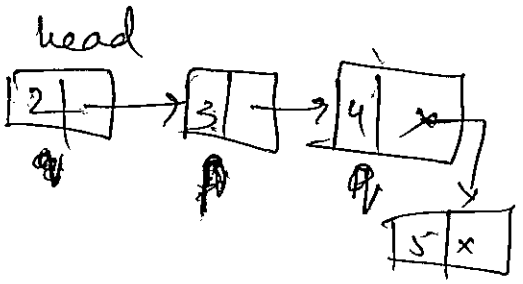




SLL

```
struct node
{ int data;
  struct node *next;
};
struct node *n, *head=NULL
void insertion(struction node *)
        del
        display
        search

void main()
{
  int ch, ele, op;
  clrscr();
while(1)
  {
    pf("Menu");
    pf("1. insert");
    pf("2. del");
    pf("3. search");
    pf("4. display");
    pf("enter ur choice");
    sf("%d", &ch);
  switch(ch)
  {
   case 1: pf("enter the ele");
           sf("%d", &ele);
n = (struct node *) malloc (sizeof (struct node));
  n->data = ele;
  if(head == NULL)
  {
     n->next = NULL;
   } head = n;
   else
   { insert(n);
   pf("enter value to insert");
   pf("1. at begin 2. at end
   break; 3. at
          sf("%d", &o+
case 2: pf("enter the ele to del")
        sf("%d", &ele);
  ----  del();
        break;
case 3: search();
        break;
case 4: display();
        break;
case 5: exit();
        break;
  }
  }
```

```
del()
{ int ch;
  pf(" which node to del");
  pf(" 1. first node ");
  pf(" 2. last node");
  pf(" 3. after a given node");
  sf("%d", &ch);
  switch(ch)
  { case 1: n = head;
            head = head->next;
            free(n);
            break;
    case 2: for(P=head; p->next!=NULL ; P=p->next)
            {
              q = P;
            }
            q->next = NULL
            free(P);
            break;
    case3: pf("enter after which node");
           sf("%d", &val);
           P=head
           while(p->data!= val)
           {
           P=p->next;
```




```
            q = p->next;
            p->next = q->next;
            free(P);
            break;
          }
        }
```

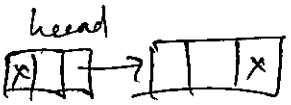
```
search()
{
  Pf("enter ele to search");
   sf("%d", ele);
   p=head
  while(p!=NULL)
   {
      p=p->next;
      if(p->data==ele)
      {
        c=1;
        break;
      }
      else
        p=p->next;
   }
  if(c==1)
    pf("ele found")
  else
    pf("ele not there");
}
```

```
display()
{
 pf
 for(p=head; p!=NULL; p=p->next)
   pf("%d", p->data);
}
```

Double linked list :-

A dLL is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.

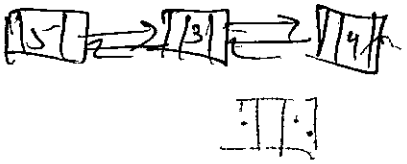
```
struct node
{
   struct node * prev;
    int data;
   struct node *next;
};
```




```
insertion(struct node *n)
{
 n = (struct node *) malloc(sizeof(struct node));
   pf("enter data");
   sf("%d", n->data);
   if(head == NULL)
   {
      n->next = NULL;
      n->prev = NULL;
       head = n;
   }
  else
 {
   pf("enter where to insert");
```

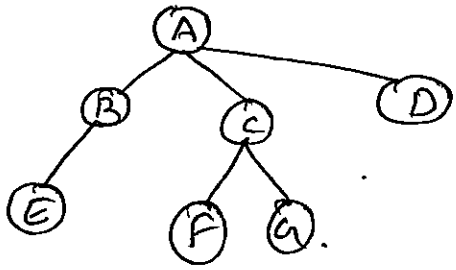
```
pf("1. At begin");
pf("2. At end");
pf("3. After a node");
sf("%d", &ch);
switch(ch)
{
case 1: n->prev = NULL;
        head->prev = n;
        n->next = head;
        head = n;
break;
```

```
case 2: p=head;
 while(p->next != NULL)
 { p = p->next;
 }
 n->prev = p;
  n->next = NULL;
  p->next = n;
 break;

case 3: pf("enter after which node");
          sf("%d", &val);
       p=head;
    while( p->data != NULL)
     {
      if( p->data == val)
      {
         n->next = p->next;
       (p->next)->next = n;
          p->next = n;
          break;
      }
      else
        p = p->next;

     }.

    break;
}
```



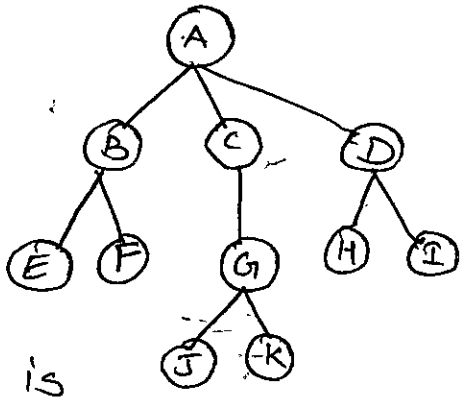
Unit-2.

Tree :- A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree & all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.



Tree.

Basic Terminology :-



Root node :- The root node 'R' is the topmost node in the tree.

If R = NULL, then tree is empty.

Sub trees :- If the root node 'R' is not NULL, then the trees T_1, T_2 & T_3 are called the sub-trees of R.

Leaf node :- A node that has no children is called the leaf node or terminal node.

Path :- A sequence of consecutive edges is called a path.

873540

Ancestor node :- An ancestor of a node is any predecessor node on the path from root to that node.

The root node doesnot have any ancestors.

Descendant node :- A decendant node is any successor node on path from the node to a leaf node.

Leaf nodes do not have any descendant nodes.

Level number :- Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1.

Degree :- It is equal to the number of children that a node has.

The degree of leaf node is zero.

Indegree :- Indegree of a node is the number of edges arriving at that node.

Outdegree :- Outdegree of a node is the number of edges leaving that node.

Types of Trees :- There are 6 types

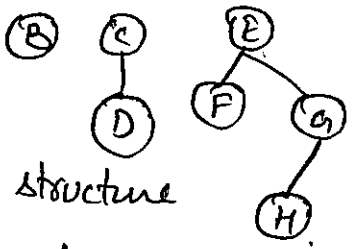
1. General Trees
2. Forests
3. Binary Trees
4. Binary Search Trees
5. Expression Trees
6. Tournament Trees.

General Trees:- It stores elements hierarchically. A node in a general tree may have zero or more sub trees. The no. of sub-trees for any node may be variable.

The problem is when another sub-tree is added to a node that already has the maximum number of sub-trees attached to it. Even for searching, traversing, adding and deleting becomes much more complex.

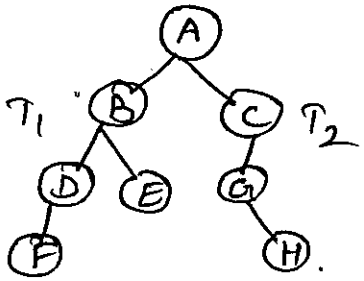
Forests:- A forest is a disjoint union of trees. Every node of a tree is the root of some sub-tree.

A forest can also be defined as an ordered set of zero or more general trees.



Binary Tree:- A B.T. is a tree data structure in which each node has at most two children, which are referred as left child and the right child.

A - root node.
T_1 - is left sub tree
T_2 - is right sub tree.



Parent:- If 'N' is any node in T that has left successor S_1 & right successor S_2 then N is called the parent of S_1 & S_2.

Every node other than the root node has a parent node. Depth of Tree :- In any tree, depth of a tree is the longest path from ~~root~~ leaf to the ~~leaf~~ root node.

Sibling :- All nodes that are at the same level & share the same parent are called siblings.

Depth :- The depth of a node N is the length of the path from the root 'R' to the node N. The depth of the root node is zero.

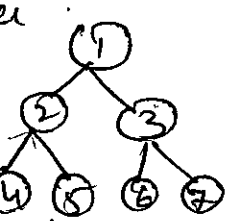
Height of a Tree :- It is the total number of nodes on the path from the root node to the deepest node in the tree. A tree with only a root node has a height of 1.

* A binary tree of ht 'h' has atleast 'h' nodes and at most $2^h - 1$ nodes.

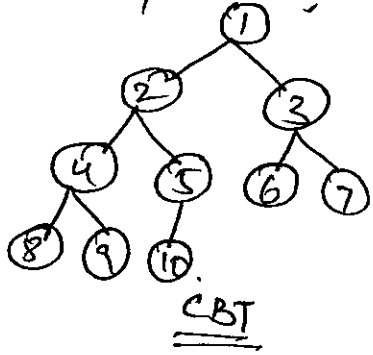
Complete binary Tree :- A CBT is a binary tree that satisfy two properties

1. every level except possibly the last is completely filled
2. All nodes are as far left as possible

Full Binary Tree :- A FBT is a tree in which every node other than the leaves has two children.

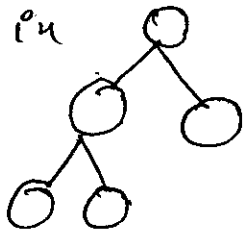


FBT



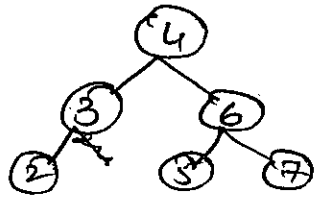
CBT

Extended Binary Tree:— A binary tree T is said to be an EBT if each node in the tree has either no child or exactly two children. (2-Tree)



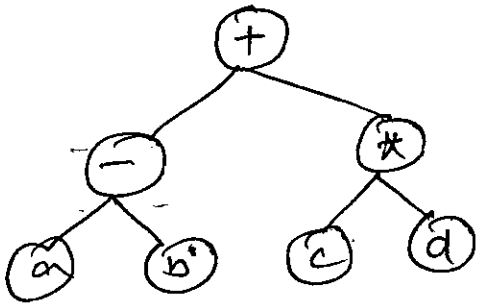
In this nodes having two children are internal nodes & nodes having no children are called external nodes.

Binary Search Trees:— It is also known as an Ordered B.T., in which the nodes are arranged in an order. In this all nodes in the left sub-tree have a value less than that of the root node & all the nodes in the right sub-tree have a value greater than that of the root node.

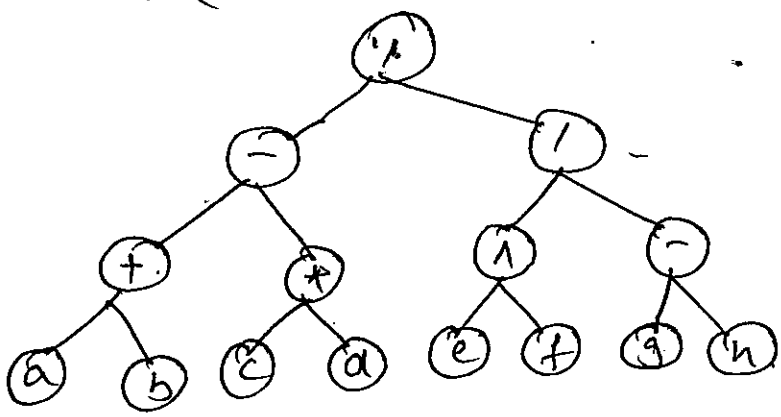


Expression Trees. B.T are used to store algebraic expression, these are called E.Trees.

i) $(a-b) + (c*d)$

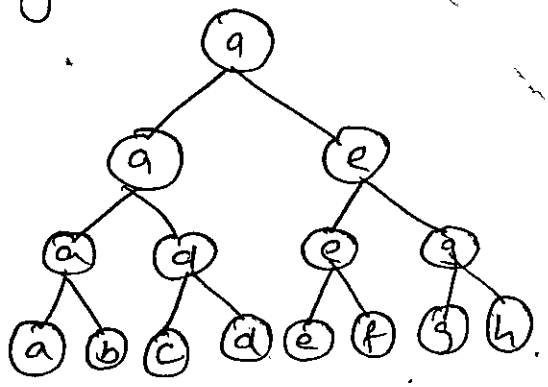


ii) $((a+b)-(c*d)) \uparrow ((e\uparrow f)/(g-h))$



Height of a node:— is the total no. of edges from leaf to that node in longest path.

Tournament trees:— (Selection tree) In this, each external node represents a player & each internal node represents the winner of the match played b/w the players represented by its children nodes. They are also called as winner trees.



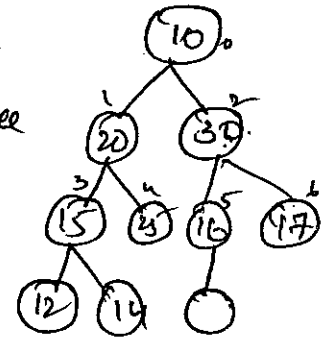
Representation of Binary tree in the Memory:—

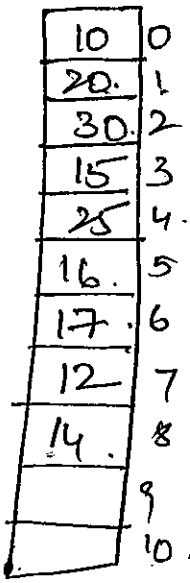
It can be represented using a sequential or linked.

1. Sequential Representation:— It is done using single or one-dimensional array. Though it is simple but it is inefficient as it requires a lot of memory space.

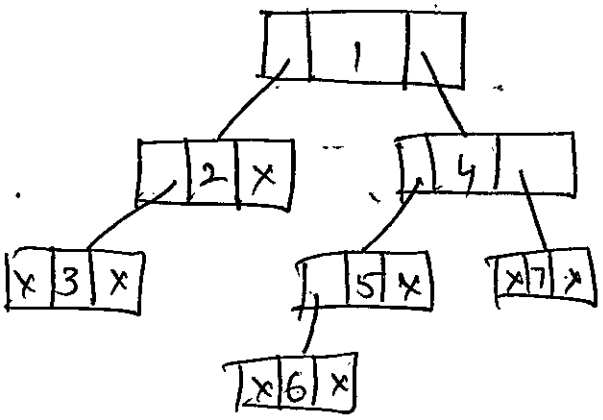
— A one-dimensional array, called tree is used to store the elements of tree

— The root of the tree is stored at '0'th location then children of the nodes as, left child in (2k+1) & right child in (2k+2) where 'k' is the position of the node.



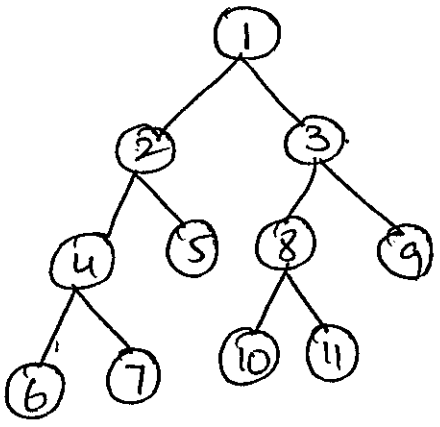


Linked Representation :- Every node have three parts .1) data ele
2) pointer to left node
3) " " right node.




```
Struct node
{
   struct node *left;
   int data;
   Struct node *right;
};
```

- The Root pointer points to the root of the tree.
- If Root = NULL, then the tree is empty.



root →

	Left	data	right
1	10	4	5
2	-1	11	-1
3	7	1	9
4	-1	10	-1
5	-1	7	-1
6	-1	9	-1
7	1	2	12
8			
9	14	3	6
10	-1	6	-1
11			
12	-1	5	-1
13			
14	4	8	2

Adv:- 1. A particular node can be placed at any location in memory
2. Insertions & deletions can be made directly without data movements.
3. Best for any type of trees.

4. flexible because system take care of allocating & freeing of nods.

Dis adv! —

1. Difficult to understand
2. Additional memory is needed to store pointers.
3. Accessing a particular node is not easy.

Seq adv! -

1. Easy to understand
2. Best for complete & full B.T.
3. programming is very easy
4. Easy to move from child to parent & vice versa.

dis adv! —

1. Lots of memory is wasted.
2. Insertions & deletion of nodes needs a lot of data movement.
3. Execution time is high.

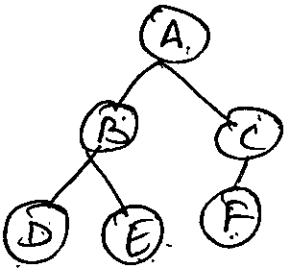
Traversing a Binary Tree! —

It is the process of visiting each node in the tree exactly once in a systematic way.

As tree is a non-linear ds in which the elements can be traversed in many diff ways.

There are 4 traversals.

1. Pre-Order traversal :- The following operations are performed recursively at each node.
   1. Visiting the root node
   2. Traversing the left sub-tree
   3. Traversing the right sub-tree.



In this, first root node, then left subtree & then the right sub-tree.

- It is also called as depth-first traversal.

Alg:- Step 1: Repeat steps 2 to 4 while Tree != NULL
Step 2: Write Tree → Data
Step 3: Preorder (Tree → left)
Step 4: Preorder (Tree → right)
End of Loop.
Step 5: END.

(NLR traversal algorithm).

- It is used to extract a prefix notation from an expression tree.

Inorder Traversal :-

1. Traversing the left sub-tree
2. Visiting the root node
3. Traversing the right sub-tree

- It is also called as symmetric traversal.. (LNR traversal algorithm).
- Used to display the elements of BST.

Alg:- Step 1 : Repeat steps 2 to 4 while Tree != NULL.

Step 2 : Inorder (Tree → left)

Step 3 : write Tree → data

Step 4 : Inorder (Tree → right)

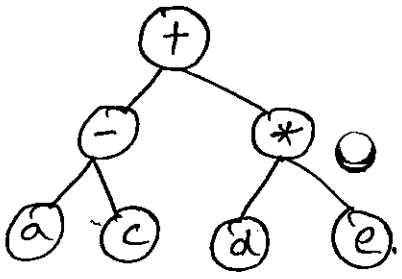
End of Loop.

Step 5 : End.

Post order Traversal :-

1. Traversing the left sub-tree
2. Traversing the right sub-tree
3. Visiting the root node.

- (L R N)
- Used to extract postfix notation.



In - (a-c) + (d*e).
Pre - + - a c * d e.
Post - a c - d e * +

Alg:- Step 1 : Repeat steps 2 to 4 while Tree != NULL.

Step 2 : Postorder (Tree → left)

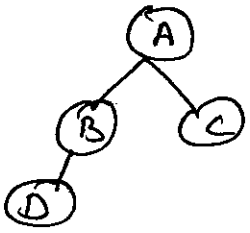
Step 3 : Postorder (Tree → right)

step 4 : Visit the root node.

End of loop

Step 5 : End.

Level order Traversal :- All nodes at a level are accessed before going to the next level. This is also called as breadth first traversal algorithm.



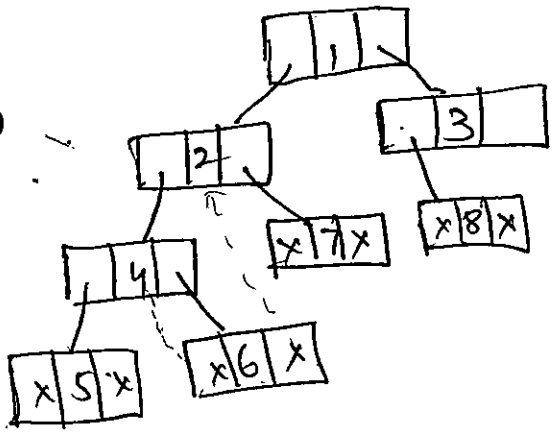
- A B C D.

Threaded Binary Trees:-

It is same as B.T but the difference is storing the NULL pointers.

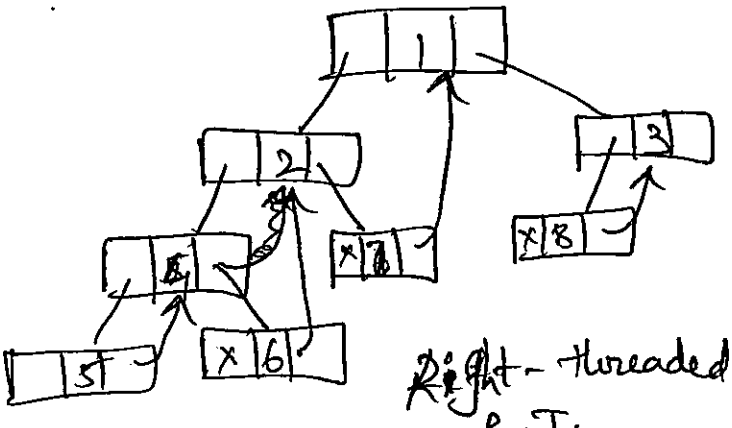
In a linked representation, a number of nodes contain a NULL pointer, either in their left or right fields or in both. This space is wasted in storing a NULL can be efficiently used to store some other useful piece of information.

The NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node. These special pointers are called threads & binary trees containing threads are called threaded trees.

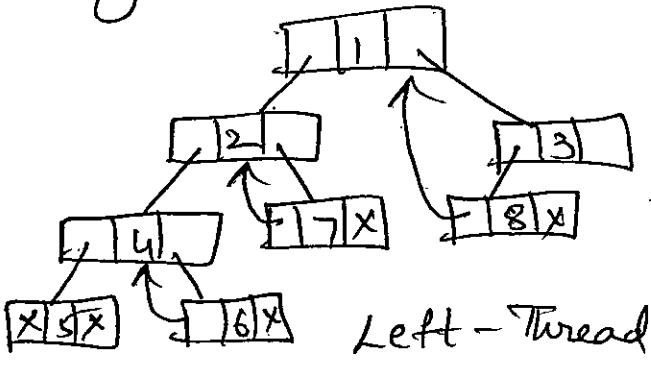


One way Threading:-

If a thread is appeared in only one pointer either left or right then it is called as one way threading.



Right-Threaded B.T.

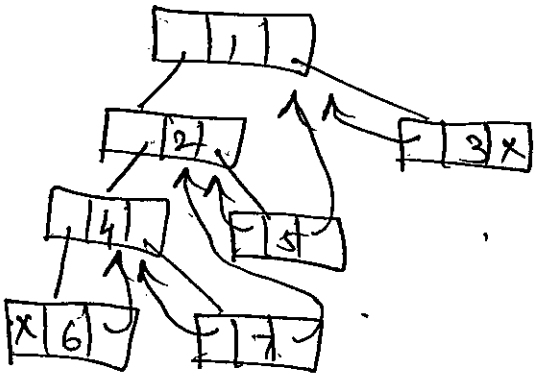


Left-Thread B.T.

Two way Threading :— The threads will appear in both left & right field of a node is called two way threading or double-threaded tree.

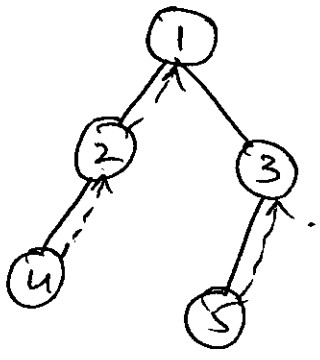
Left – in-order predecessor.

Right – in-order successor.



In-order Threading :—

Traversing a threaded binary tree which is done in in-order.



For every node, visit the left sub-tree first, provided if one exists & has not been visited. Then the node (root) itself is followed by visiting its right sub-tree. Incase there is no right sub-tree, check for threaded link & make the threaded node the current node.

1. Node 1 has leftchild i.e. 2 which has not visited. So add 2
2. 2 has left child i.e. 4 " " " "
   So add 4.
3. Node 4 does not have left & right child, so print

& check for its threaded link. It has a threaded link to node 2, so make node 2 current node.

4. Node 2 has a left child, already been visited. It does not have a right child. Print 2.

5. Node 1 has left child, already visited print 1. 1 has right child 3 which is not visited, so 3 is current node

6. Node 3 has left child which is not visited, so make 5 has current node.

7. Node 5 doesn't have any child. print 5. thread is 3 → current node.

8. '3' is current node print & it doesnot have any right child & no thread.

9. 42153.

Eg :- Tree constructs.

1) Inorder - 20, 30, 35, 40, 45, 50, 55, 60, 70
preorder - 50, 40, 30, 20, 35, 45, 60, 55, 70

2) In - 4, 8, 2, 5, 1, 6, 3, 7
post - 8, 4, 5, 2, 6, 7, 3, 1.

3) Pre - 25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90,
post - 4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

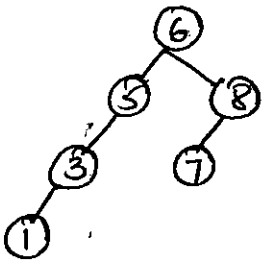
4) Pre - A B D E F C G H J L K
In : D B F E A G C L J H K

5) Pre - A B D G H K C E F
Post - G K H D B E F C A

```
#include<stdio.h>
void insert(struct node *, struct node *);
void inorder(struct node *);
void preorder(struct node *);
void postorder(struct node *);
struct node
{
 int data;
 struct node *left, *right;
};
struct node *root=NULL, *n, *temp

void main()
{
 int ch;
 clrscr();
 while(1)
 {
 printf("1. Insert 2.
 printf("enter ur choice");
 scanf("%d", &ch);
 switch(ch)
 {
 case 1: n=(struct node *)malloc(sizeof(s.n));
 printf("enter data");
 scanf("%d", &val);
 n->data=val;
 n->left=n->right=NULL;
 if(root==NULL)
 root=n;
 else
 insert(root, n);
 break;
 case 2: inorder(root);
 break;
```

```
void insert(struct node *temp, struct node *n)
{
 if(n->data < temp->data)
 {
 if(temp->left==NULL)
 temp->left=n;
 else
 insert(temp->left, n);
 }
 else
 {
 if(temp->right==NULL)
 temp->right=n;
 else
 insert(temp->right, n);
 }
}
```

```
void inorder(s.n *temp)
{
 if(temp!=NULL)
 {
 inorder(temp->left);
 pf("%d", temp->data);
 inorder(temp->right);
 }
}
```



Unit-3

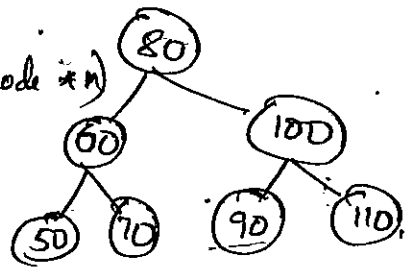
Binary Search Tree :-

Operations on B.S.T :-

On BST three operations can be performed-

1. Insertion
2. Deletion
3. Search

Insert :- It is used to add a new node with a given value at the correct position in the B.S.T.



Alg :-
```
Void Insert (struct node *r, struct node *n)
{
  if ( n->data < r->data)
  {
    if ( r->left == NULL)
      r->left = n;
    else
      insert (r->left, n);
  }
  else
  {
    if(r->right == NULL)
      r->right = n;
    else
      insert (r->right, n);
  }
}
```

When a new node came to insert the ele is checked with root node. If the new element is less than root then it will goes to left subtree. If the left subtree is null the new node is attached otherwise again it will check with the left subtree element & procedure continue until the node is attached.

If the element is greater than root then it will goes to the right sub tree & the procedure is same.

2. Delete :— When a node to be deleted there exists three cases.

1) Deleting a node that has no children.

To delete a node, simply remove that node.

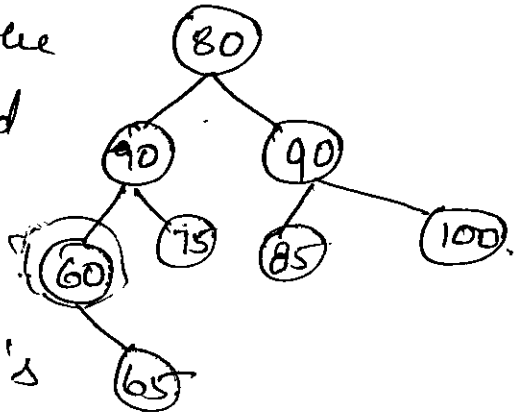
2) Deleting a node with one child

To delete a node which is having one child, set the child as child of node's parent.

If the node is the left child of its

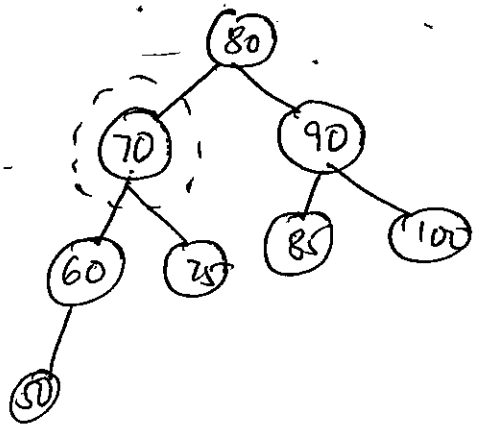
Parent, the node's child becomes the left child of the node's parent.

Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent.



- 3. Deleting a node with Two children

Replace the node's value with its in-order predecessor (largest value in the left subtree) or inorder successor (smallest value in the right subtree).

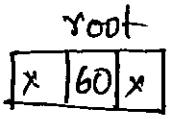


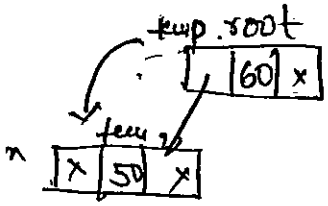
→ 70 — can be replaced with 60 or 75.

3. Search:— It is used to find whether a given value is present in the tree or not. Process starts at root node & it checks until the BST is empty. If it is empty then the

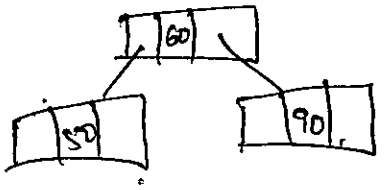
value is not present in the tree.

```
Insert()
{
  n = (struct node *) malloc (sizeof (struct node));
  printf("enter data");
  scanf("%d", &n->data);
  n->left = NULL;
  n->right = NULL;
  if(root == NULL)
    root = n;
  else
  {
    temp = root;
    while(temp != NULL)
    {
      if( n->data < temp->data)
      {
        if(temp->left == NULL)
        {
          temp->left = n;
          break;
        }
        else
          temp = temp->left;
      }
      else
      {
        if (temp->right == NULL)
```



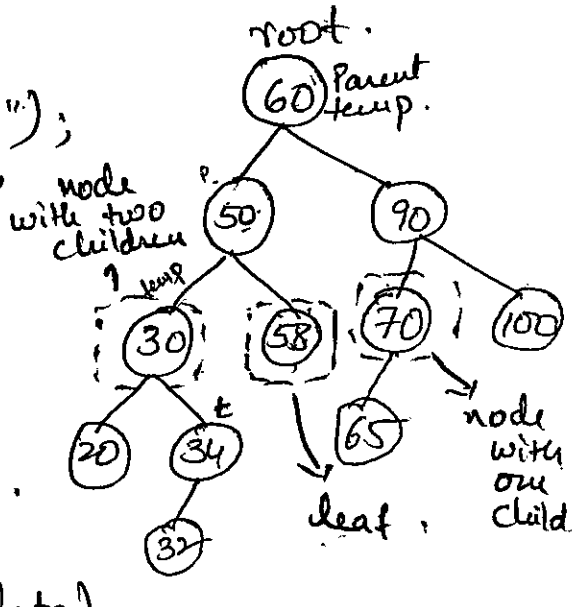


```
        }
            temp -> right = n;
            break;
        }
        else
            temp = temp -> right;
    }
}
```



Delete :—

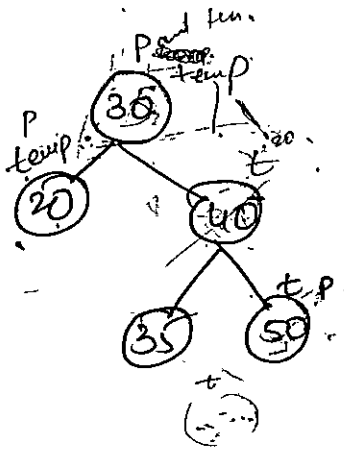
```
Delet )
{   int key; c=0; *P=NULL, *Parent=NULL;
    if ( root == NULL)
      printf("no nodes to delete");
    else
    {temp=root ; Pf("enter ele to del");
     sf("%d", &key);
     while (temp!=NULL)
     {
        Parent=temp;
        if(key < temp->data)
        { Parent=temp;
          temp = temp->left;
        }
        else if (key > temp->data)
        { Parent = temp;
          temp = temp->right;
        }
```



```
else if (key == temp->data)
{
    if (temp->left == NULL && temp->right == NULL)
    {                                          leaf node
        if(Parent == NULL)
        { free(temp); root=NULL ; c++; break; }
        else
            free(temp); c++;
            break;
    }

    else if(temp->left == NULL && temp->right != NULL)
        if(Parent == NULL)                     node with one child
        { root = temp->left;
    {
        if( parent->left == temp)
        {
            parent->left = temp->right;
        }
        else if(parent->right == temp)
        {
            parent->right = temp->right;
        }
        free(temp); c++;
        break;
    }

    else if(temp->left != NULL && temp->right == NULL)
        if(Parent == NULL)
            root = temp->right;
    { else if( p->left == temp)
        p->left = temp->left;
      else
        p->right = temp->left;
```

if(Parent->left == temp)
    Parent->left = NULL;
else
    Parent->right = NULL;

```
        free(temp); ctt;
        break;
    }
else if (temp->left != NULL && temp->right != NULL)
                                    node with two children
{
    t = temp->right;
    P=NULL;
    while(t->left != NULL)
    {   P=t;
        t = t->left;
    }
    temp->data = t->data;
    free(t);          ctt;
    break;
}
```




```
if(P==NULL)
    temp->right = NULL;
else
    P->left = NULL;
}
}
if(c==0)
    Pf("node is not present");
}
}
```

Search:—

```
Search()
{
  if(root == NULL)
   pf("Tree is empty");
  else
  {
    pf("enter key to search");
    sf("%d", &key);
     temp = root;
     while(temp != NULL)
     {
     if(key < temp->data)
        temp = temp->left;
      else if(key > temp->data)
        temp = temp->right;
      else if (key == temp->data)
      {
        pf("the node is present");
          break;
      }
    }
    if(temp == NULL)
     pf("node is not present");
  }
}
```

AVL Trees :-

It is a self balancing BST invented by Adelson – Velsky & Landis.

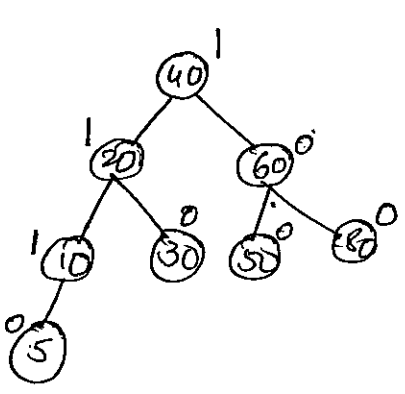
In an AVL tree, the heights of the two sub-trees of a node may differ by at most one. Due to this it is also known as height – balanced tree.

AVL is same as BST with a little difference that, each node stores an additional variable called BalanceFactor.

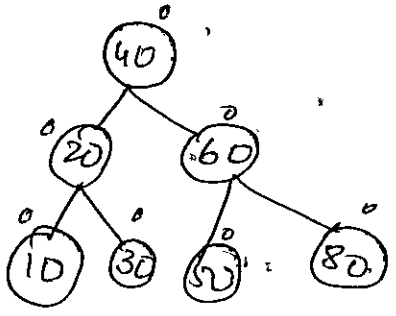
Balance-factor = Height(left sub-tree) – Height(right subtree)

An BST in which every node has a balance factor of -1, 0, or 1 is said to be height balanced.

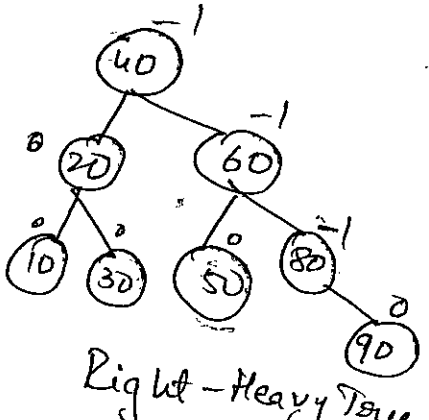
- If the bf = 1, then the left subtree is one level higher than that of the right sub-tree. Such a tree is called Left-heavy tree.



Left heavy tree.



Balanced tree



Right-Heavy Tree

• If bf = 0, then the height of the left sub-tree is equal to the height of the right sub-tree.

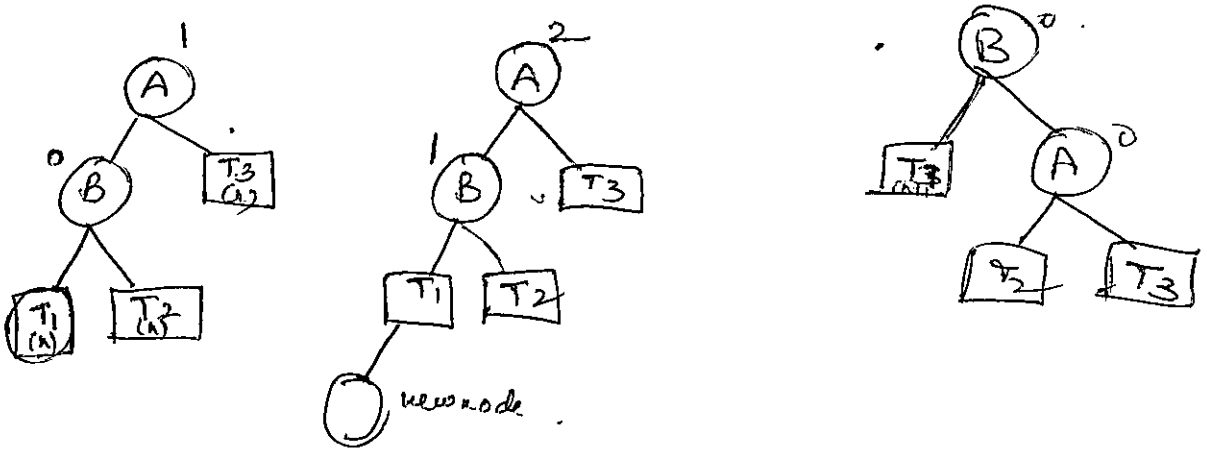
• If bf = −1, then the height of the left sub-tree is one level lower than that of the right sub-tree. Such a tree is called as a right-heavy tree.

Insertions and deletions from an AVL tree may disturb the balance factor of the nodes & thus rebalancing of the tree may have to be done.

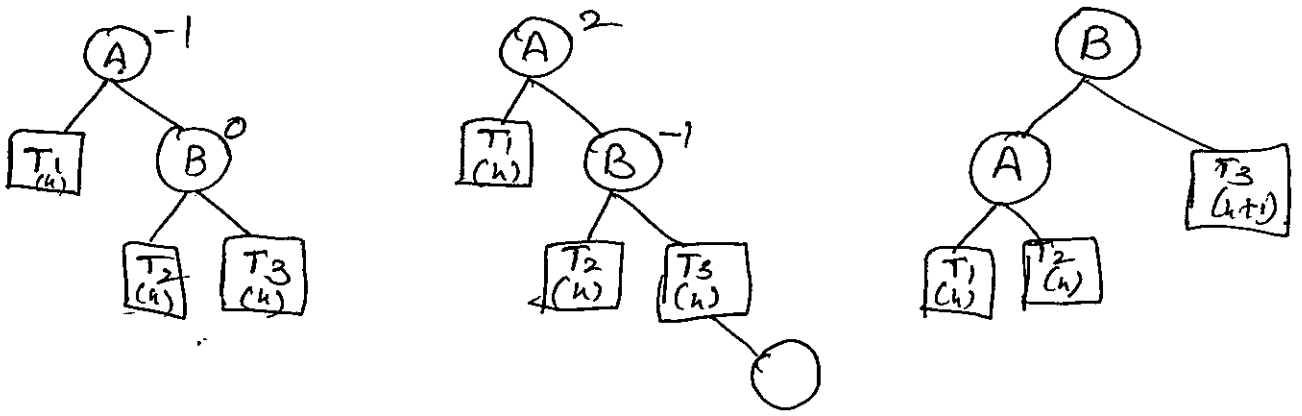
The tree is rebalanced by performing rotation at the critical node.

There are four types of rotations.

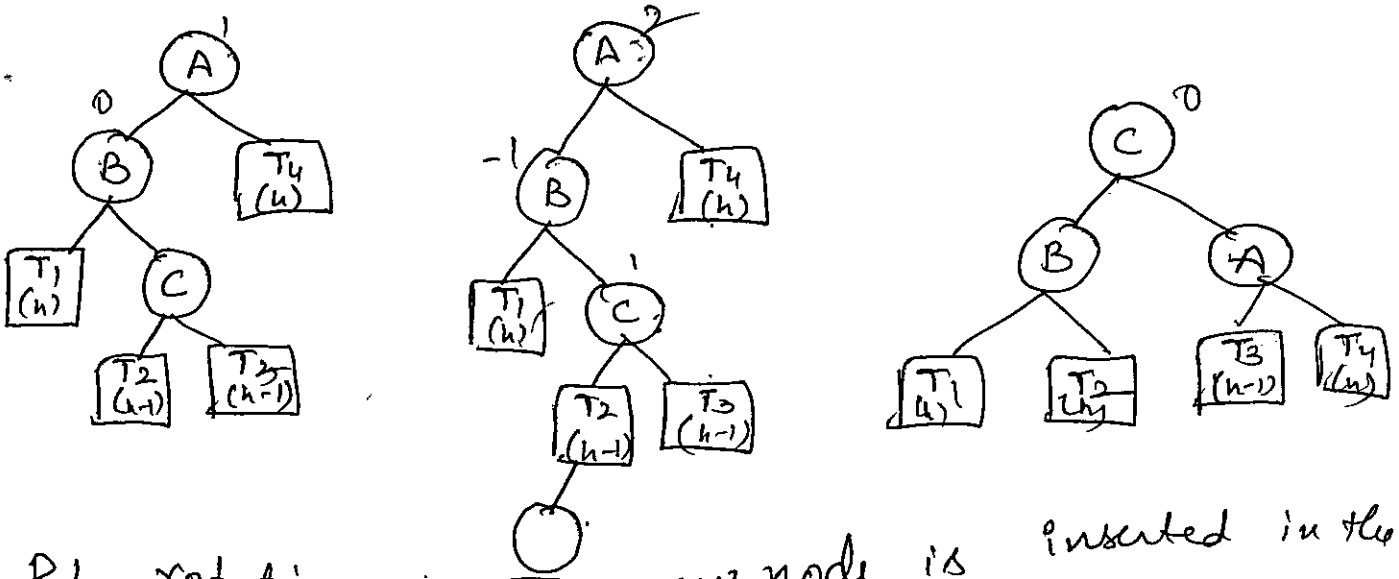
1. <u>LL Rotation</u> :— The new node is inserted in the left sub-tree of the left sub-tree of the critical node.



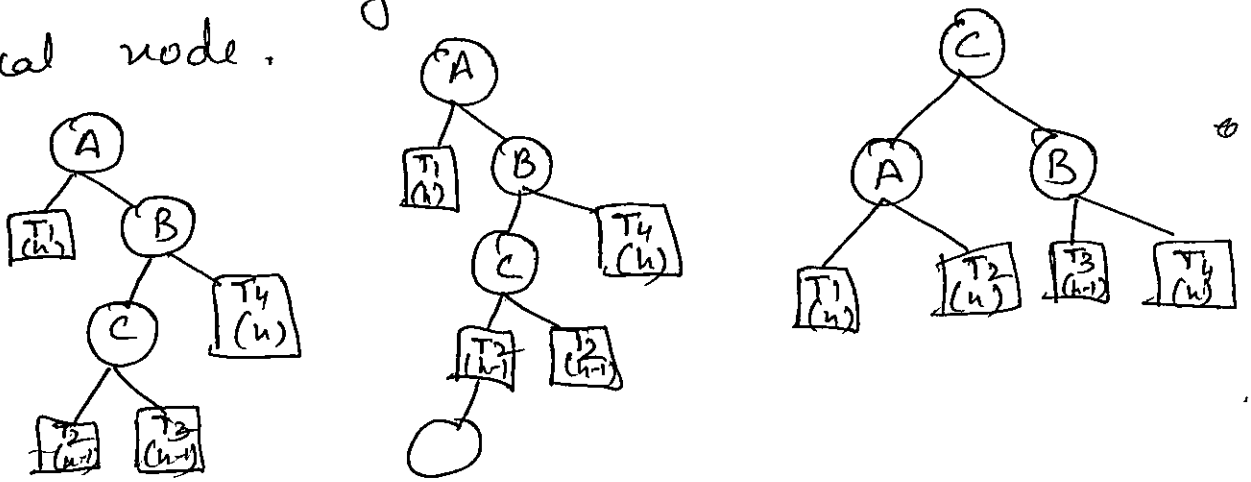
2. RR Rotation :- The new node is inserted in the right sub-tree of the right sub-tree of the critical node.



3. LR rotation :- The new node inserted in the right subtree of the left subtree of the critical node.



4. RL rotation :- The new node is inserted in the left sub tree of the right sub-tree of the critical node.



Operations on AVL Tree :-

1. **Searching** :- It is same as BST.

2. **Insertion** :- It is also same as in BST. The new node is always inserted as the leaf node. But the insertion is followed by an additional step of rotation, to restore the balance of the tree.

If insertion of the new node does not distrub the balance factor then rotations are not required.

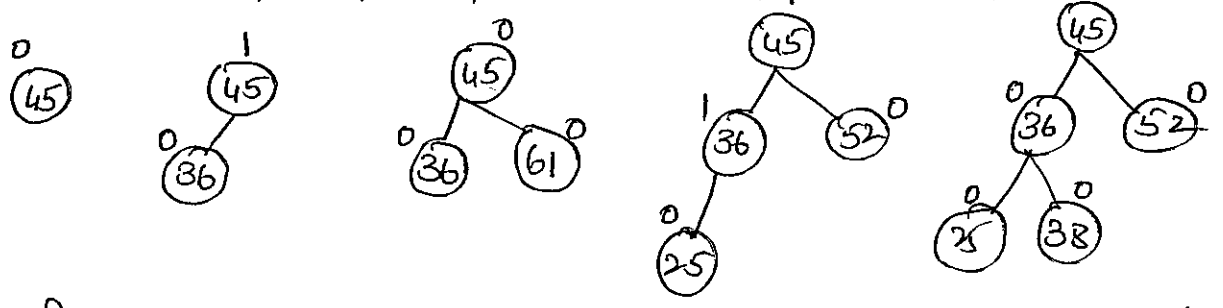
During insertion, the new node is inserted as the leaf node, so it will always have a balance factor equal to zero. The only nodes whose balance factors will change are those which lie in the path b/w the root of the tree & the newly inserted node.

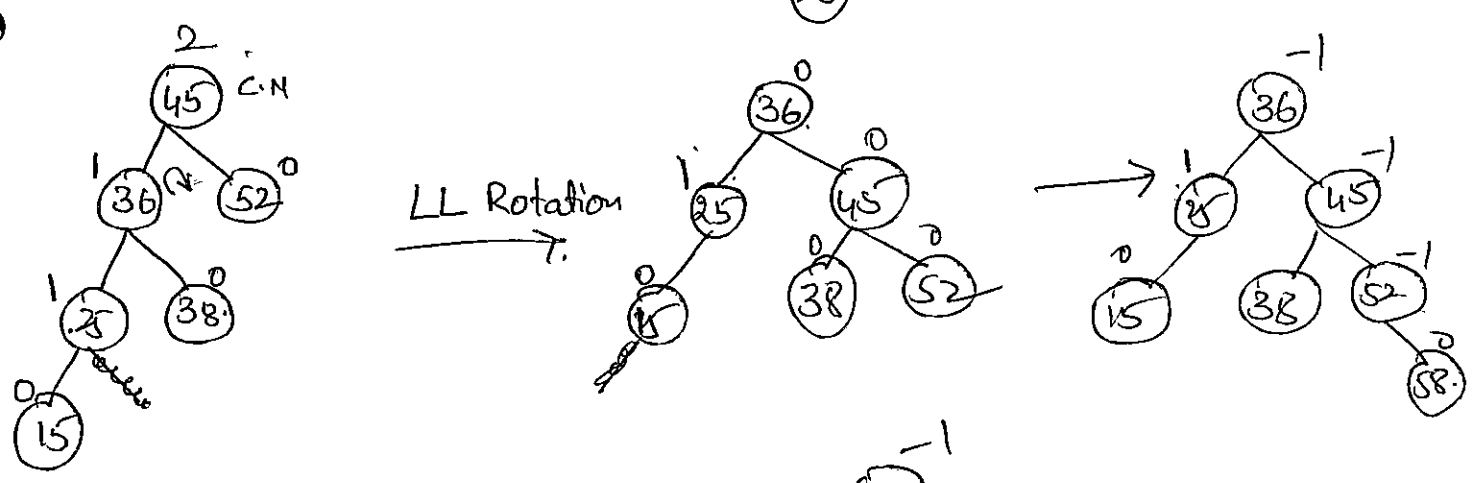
The possible changes are as follows:

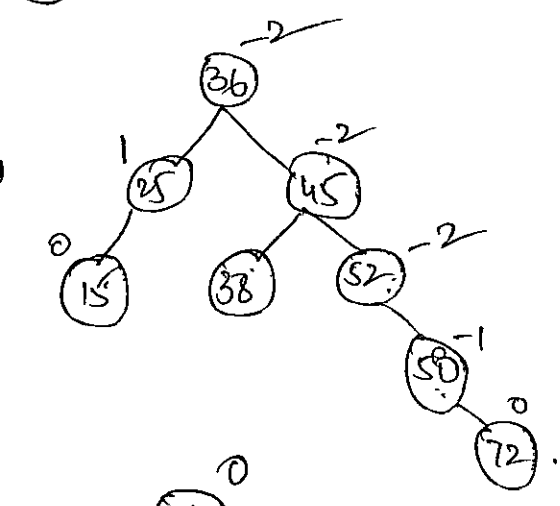
1. Initially, the node was either left or right heavy & after insertion, it becomes balanced.
2. Initially the node was balanced & after insertion, it becomes either left or right heavy.

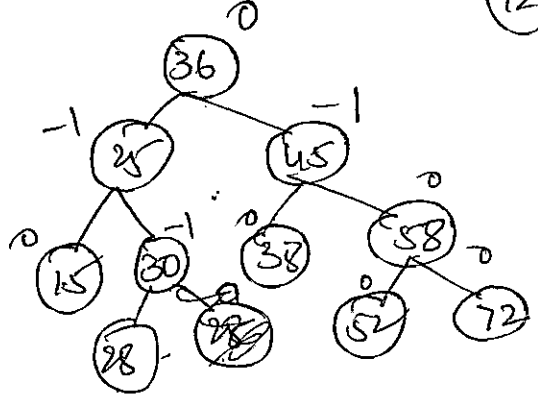
3: Initially the node was heavy & the new node has been inserted in the heavy sub-tree, creating an unbalanced sub-tree. Such node is said to be critical node.

Eg:- 45, 36, 61, 52, 25, 38, 15, 58, 72, 30, 28

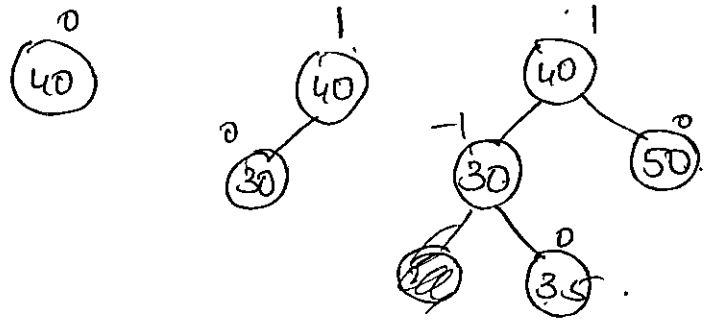


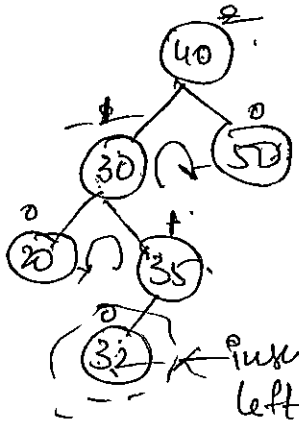




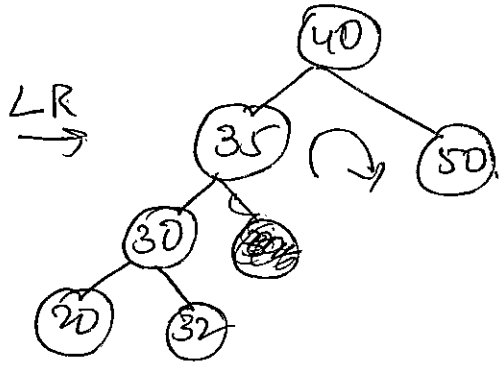


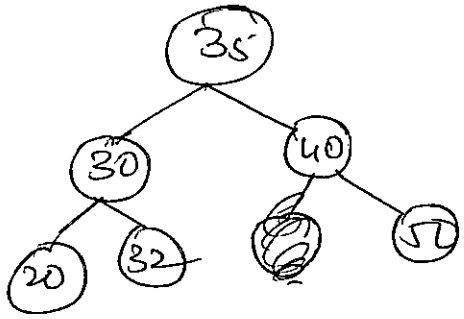
Eg:-



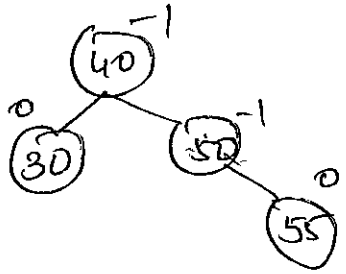


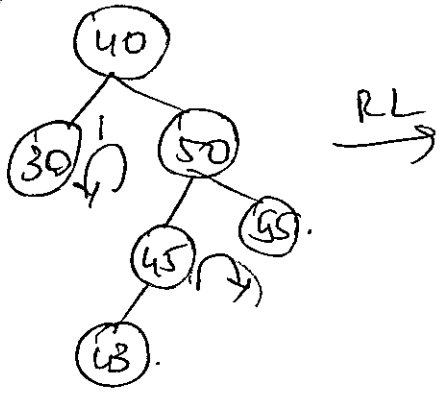
inserted. to the left subtree to right.

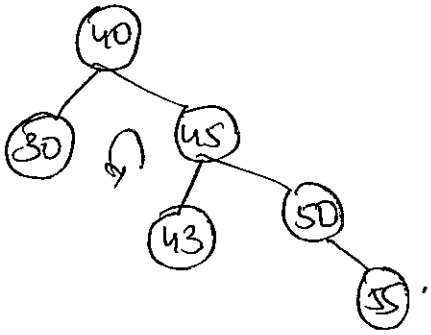


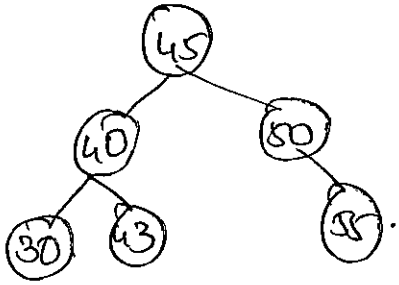


Eg:-









3. Deleting a Node from an AVL Tree :-

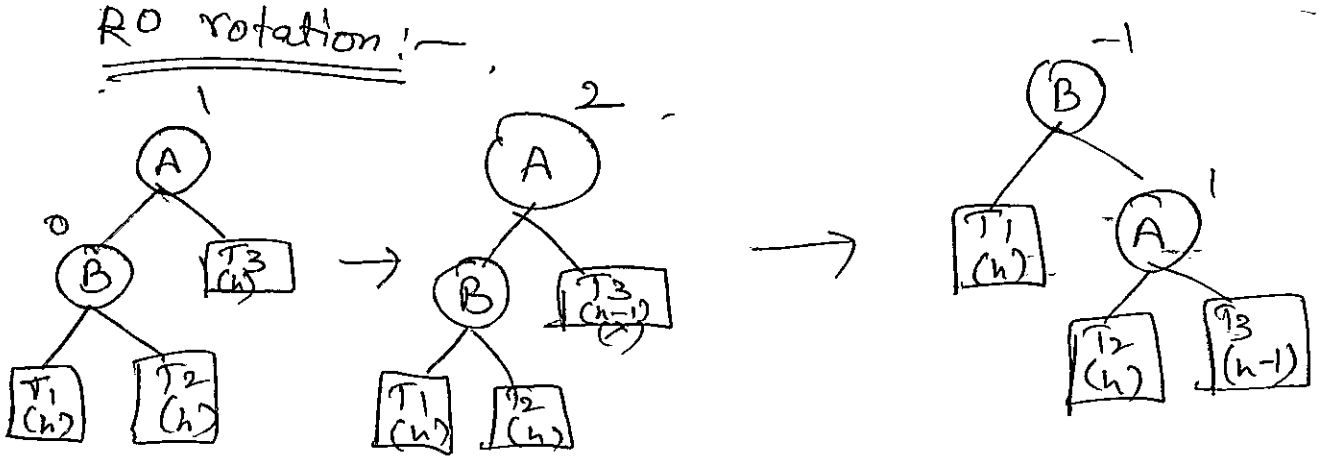
It is also similar to that of BST. But Deletion may disturb the AVLness of the tree, so to rebalance the AVL tree perform rotations. There are R & L rotation.

A node X is deleted from the AVL tree & the node 'A' becomes critical node. If 'X' is present in the left subtree of A then L rotation is applied, else if X is in the right sub-tree, R rotation is performed.

There are three categories of L & R rotations.

- R rotations :—

  RO rotation :—

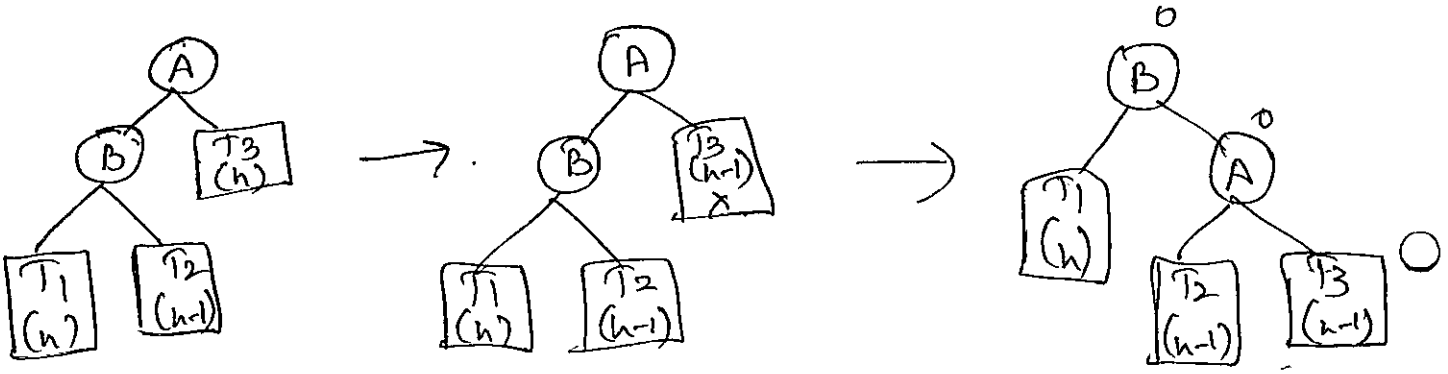


- If B be the root of the left subtree of A (Critical node), RO rotation is applied if the balance factor of B is 0.

  → Tree is an AVL Tree then node X is to be deleted from the right sub tree of the critical node A. Since balance factor of node B is 0, apply RO rotation.

R1 Rotation :— (LL).

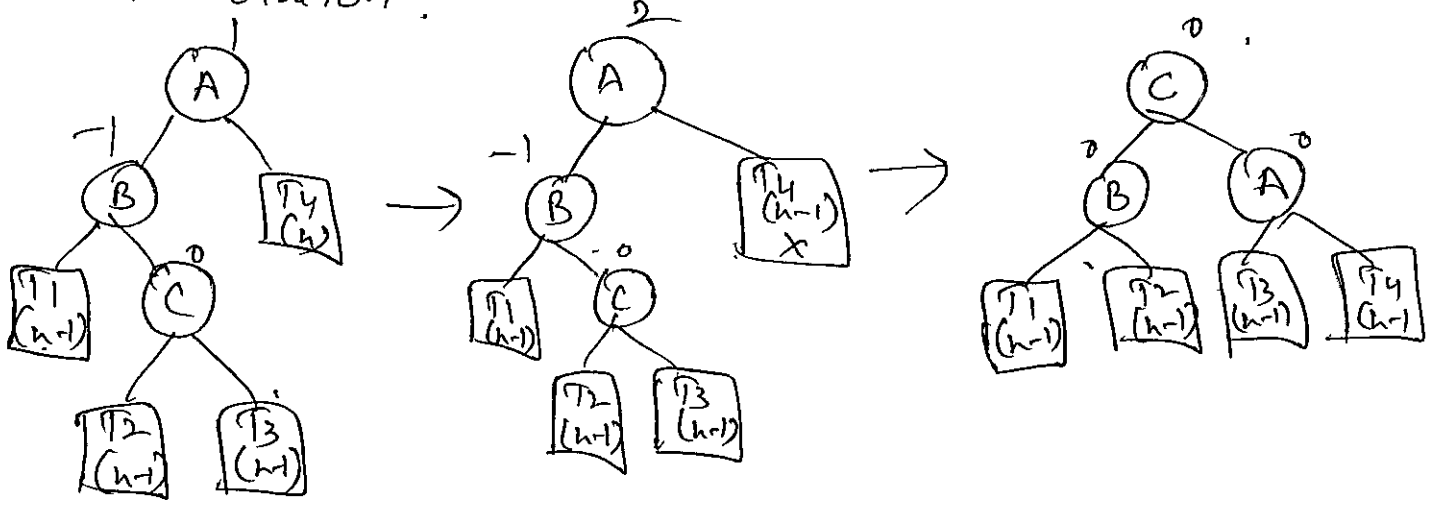
Tree is an AVL tree & the node x is to be deleted from the right sub-tree of the critical node A. Since the balance factor of node B is 1 apply R1 rotation.



(a)

R-1 Rotation :— (LR)

Tree is an AVL & the node x is to be deleted from the right sub-tree of the critical node A. Since the balance factore of node B is −1, apply R-1 rotation.



Red-black Trees:-

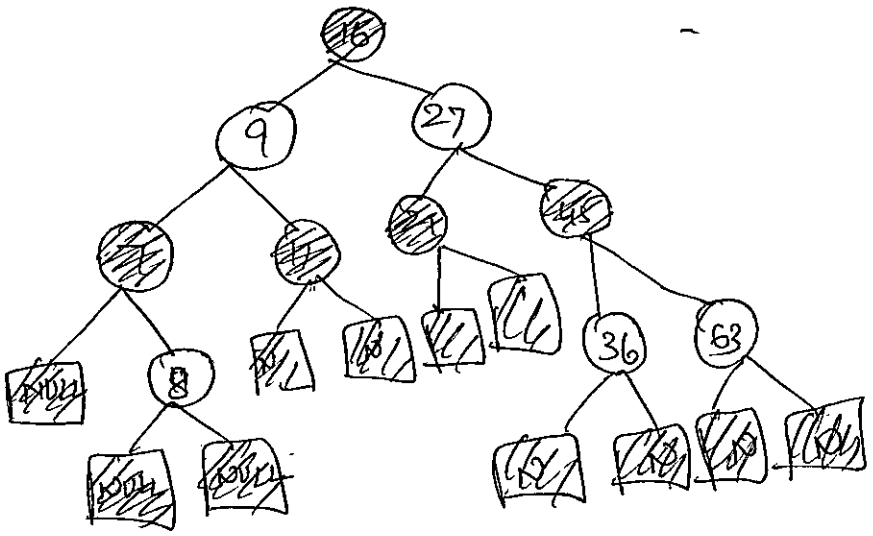
It is a self-balancing BST tree & it is also called as a 'symmetric binary B-Tree'. In this tree, no data is stored in the leaf nodes..

Properties:-.

A red black tree is a BST in which every node has a colour which is either red or black. It should have the following properties.

1. The colour of a node is either red or black
2. The colour of the root node is always black.
3. All leaf nodes are black,
4. Every red node has both the children coloured in black,
5. Every simple path from a given node to any of its leaf nodes has an equal number of black nodes.

   i.e. no path is more than twice as long as any other path.



Red-black Tree

Applications :—

1. It offers worst case time guarantees for insertion, deletion & search operations.
2. Used in real time applications.

M-way Search Trees :—

M-way search tree has, M-1 values per node and M sub-trees. In such a tree, M is called the degree of the tree.

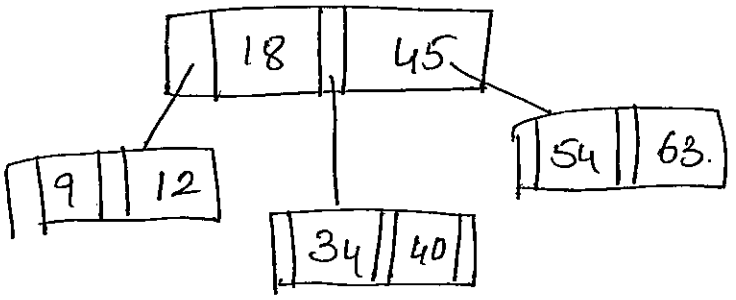
Eg:- M=2, it has one value & two sub-trees.

Every internal node of an M-way search tree consists of pointers to M sub-trees & contains M-1 keys where M>2.

P_0	K_0	P_1	K_1	P_2	K_2	----	P_{n-1}	K_{n-1}	P_n

Structure of M-way search Tree

$P_0, P_1, \ldots P_n$ are pointers to the node's sub-trees & $K_0, K_1, \ldots K_{n-1}$ are the key values of the node. All the key values are stored in ascending order.



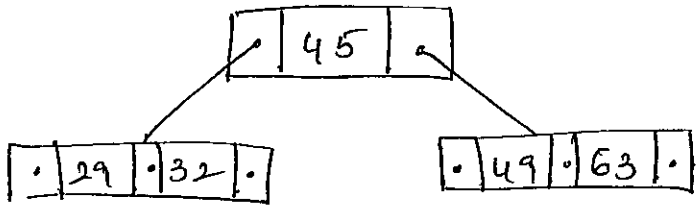
M-way search tree of order 3

B-Trees:-

A B-Tree of oder 'm' can have a maximum of $m-1$ keys & m pointers to its sub-trees. It has the following properties.

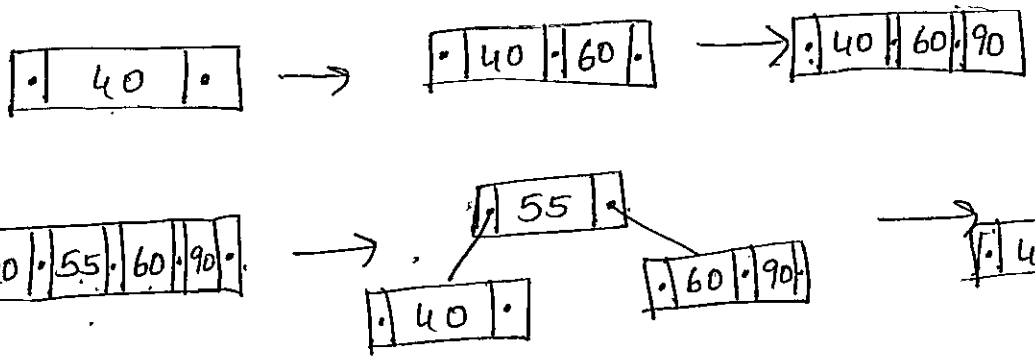
1. Every node in the B tree has at most 'm' children.
2. Every node in the B-tree except the root node & leaf nodes has atleast $m/2$ children. This conditions helps to keep the tree bushy so that path from the root node to the leaf is very short, even in a tree that stores a lot of data.
3. The root node has atleast two children if it is not a terminal node.

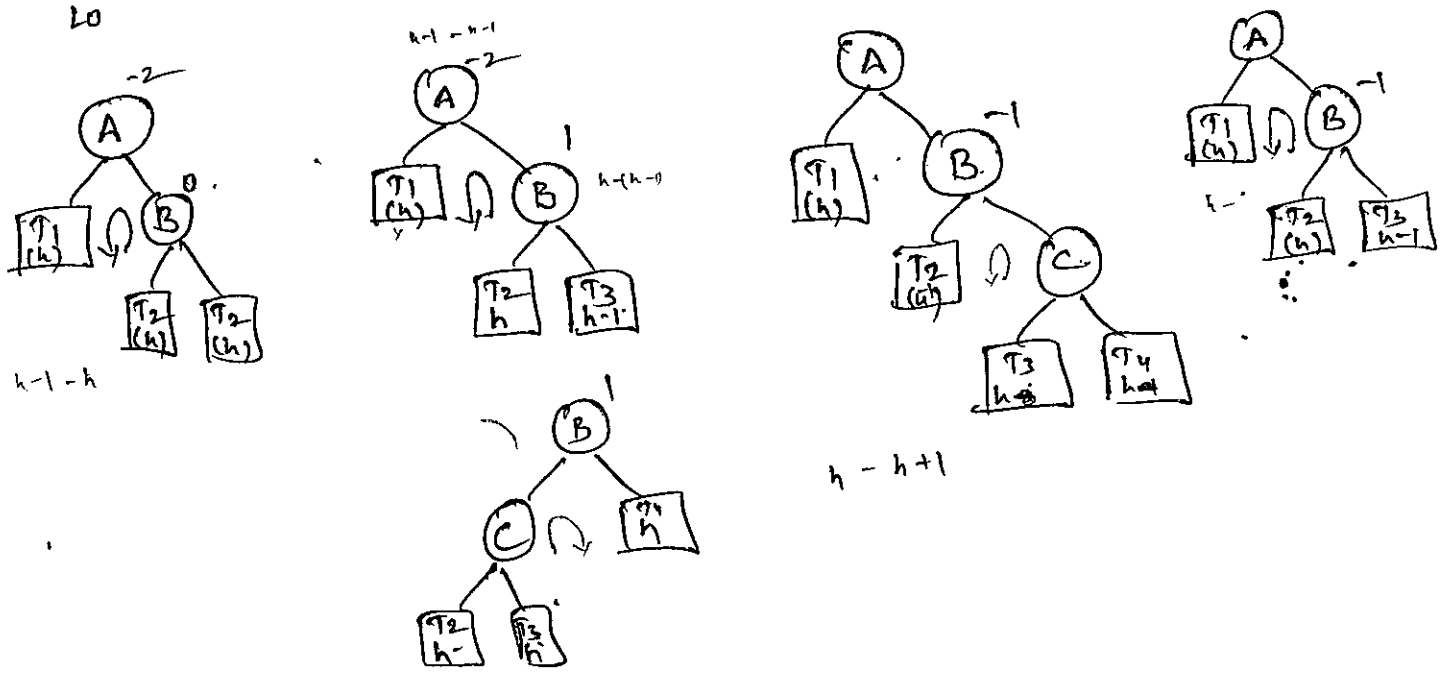
4. All leaf nodes are at the same level.



while performing insertion & deletion number of child nodes may change, so in order to maintain a min number of children, the internal nodes may be joined or split.

Eg:- B-Tree order of 4.

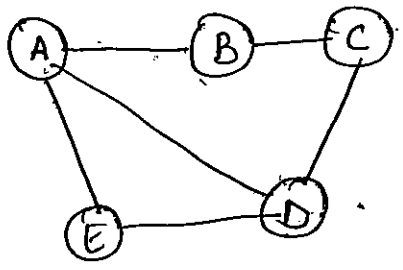




Unit - IV

Graphs

A graph G is defined as an ordered set (V, E) where 'V' represents the set of vertices and 'E' represents the edges that connect these vertices.



$V = \{A, B, C, D, E\}$

$E = \{AB, BC, CD, DE, AE, AD\}$

- A graph can be directed or undirected.

Undirected graph :- In an undirected graph edges do not have any direction associated with them.

Directed graph :- In a directed graph, edges form an ordered pair.

Graph Terminology :-

Adjacent nodes or neighbours :- For every edge, $e = (u, v)$ that connects nodes u and v, the nodes u & v are the end-points and are said to be the adjacent nodes or neighbours.

Degree of a node :- Degree of a node v, $\deg(v)$ is the total number of edges containing the node v.

Path :- A path 'p' of length 'n' from a node v to v is defined as a sequence of $(n+1)$ nodes.

Closed path :- A path 'p' is known as a closed ~~simple~~ path if the edge has the same end-points.

Simple path :- If all the nodes in the path are distinct with an exception that v_0 may be equal to v_n.

If $v_0 = v_n$ then the path is called closed simple path.

Cycle :- A path in which the first and the last vertices are same. A simple cycle has no repeated edges or vertices.

Regular graph :- A graph in which each vertex has the same number of neighbours, i.e. every node has the same degree.

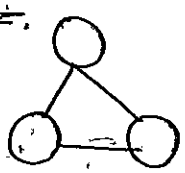
A regular graph with vertices of degree 'k' is called k-regular graph.

Connected graph :- A graph is said to be connected if for any two vertices (u,v) in V there is a path from u to v.

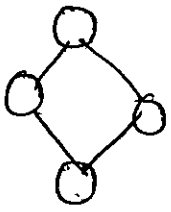
Completegraph :- A graph 'G' is said to be completed if all its nodes are fully connected. That is there is a path from one node to every other node in the graph.

- A C.G. has $n(n-1)/2$ edges where 'n' is the number of nodes in 'G'.

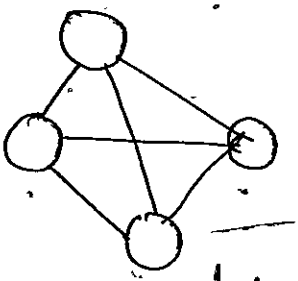
Eg:-



2-Regular graph



Connected graph.

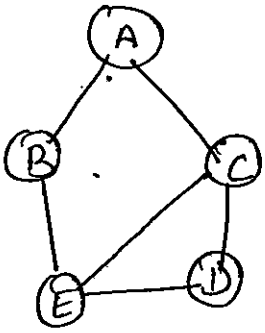


Complete graph.

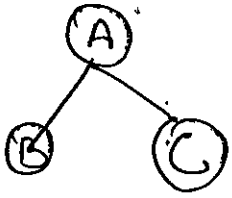
- Sub graph :- A subgraph G_1 of a graph G is defined as $G_1(V_1, E_1)$ where :-

$$V(G_1) \subseteq V(G) \text{ & } E(G_1) \subseteq E(G).$$

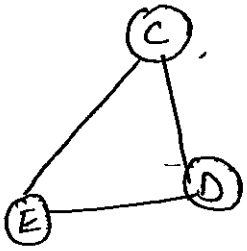
clique :- In an undirected graph $G = (V, E)$ clique is a subset of the vertex set $C \subseteq V$, such that for every two vertices in C, there is an edge that connects two vertices.



Graph G



Subgraph G1



clique 'C'.

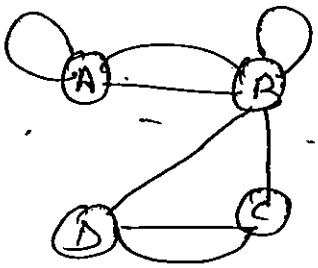
Multiple edges :- Distinct edges which connect the same end-points are called multiple edges.

Loop :- An edge that has identical end-points is called a loop i.e. $e = (v, v)$.

Multi-graph :- A graph with multiple edges and/or loops is called a multi graph.

Multiple edges

loop.

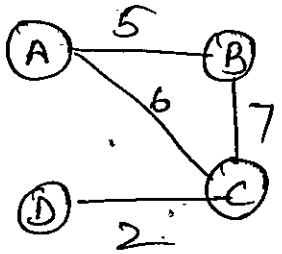


Multi graph.

Size of a graph :- The total number of edges in the graph is its size.

Labelled graph or weighted graph :- A graph is said to be labelled if every edge in the graph is assigned some data.

In a weighted graph, the edges of the graph are assigned some weight or length.



Directed graph :- Also known as digraph, is a graph in which every edge has a direction assigned to it.

For an edge (U, V)

- The edge begins at U & terminates at V.
- 'U' is known as the origin or initial point of 'e' & 'V' is known as the destination or terminal point of 'e'.
- U is the predecessor of V & V is the successor of U.
- Nodes U & V are adjacent to each other.

Terminology of a directed graph :-

out-degree of a node :- outdeg(U), is the number of edges that originate at U.

In-degree of a node :- indeg(U), is the number of edges that terminat at U.

Degree of a node :- deg(v), is equal to the sum of in-degree & out-degree of that node.

$$deg(v) = indeg(v) + outdeg(v).$$

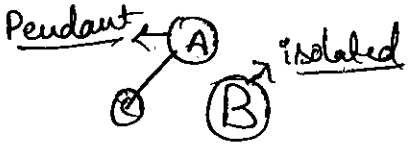
Isolated vertex :- A vertex with degree zero.

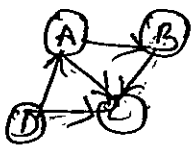
Pendant vertex :- A vertex with degree one.

Cut vertex :- A vertex which when deleted would disconnect the remaining graph.

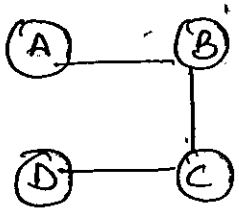
Source :- A node 'v' is known as a source if it has a positive out-degree but a zero in-degree.

Sink :- A node 'v' is known as a sink if it has a positive in-degree but a zero out-degree.





'C' is sink.
'D' is source.



B & C are cut vertex

Parallel edges → same as multiple edges.

Simple directed graph:

Representation of Graphs:-

There are three common ways of storing graphs in the computer's memory.

- Sequential representation
- Linked representation
- Adjacency multi-list.

1. Adjacency Matrix Representation:-

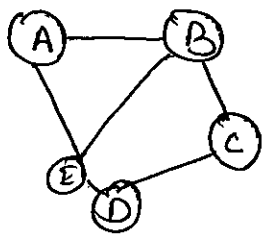
It is used to represent which nodes are adjacent to one another.

a_{ij} = 1 if V_i is adjacent to V_j i.e. there is an edge from V_i to V_j
= 0 otherwise

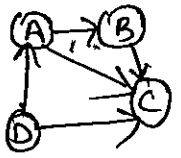
For any graph G having n nodes, the adjacency matrix will have the dimension of nxn.

In an A.M. rows & columns are labelled by graph vertices.

A.M contains only 0's & 1's it is called a bit matrix or a Boolean Matrix.

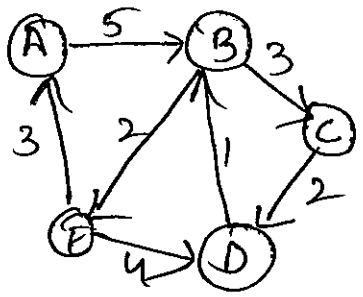


	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	1
C	0	1	0	1	0
D	0	0	1	0	1
E	1	1	0	1	0



	A	B	C	D
A	0	1	1	0
B	0	0	1	0
C	0	0	0	0
D	1	0	1	0

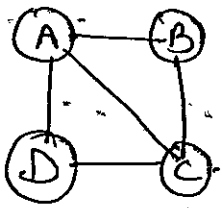
The A.M of for a weighted graph contains the weights of the edges connecting the nodes.



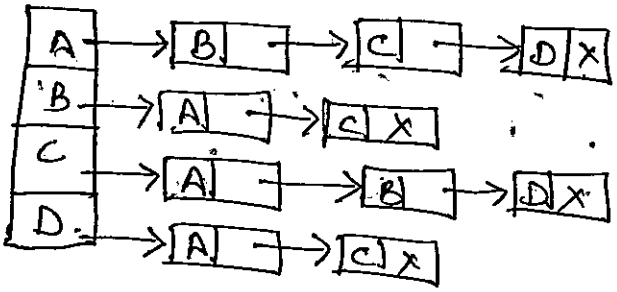
	A	B	C	D	E
A	0	5	0	0	0
B	0	0	3	0	2
C	0	0	0	2	0
D	0	1	0	0	0
E	3	0	0	4	0

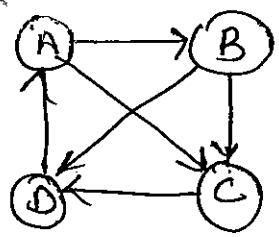
2. Adjacency list Representation:-

This structure consists of a list of all nodes in G. Every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

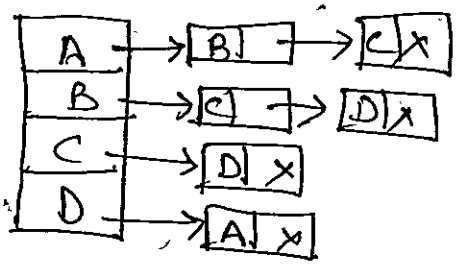


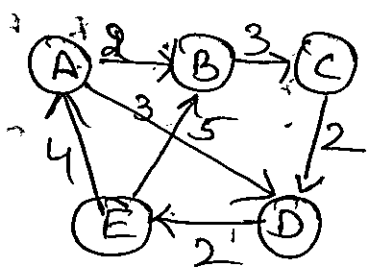
Undirected



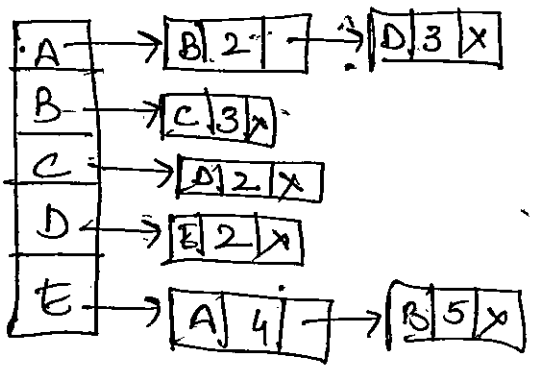


directed.





weighted graph



Advantages :—

1. easy to follow
2. Adding new nodes is easy.

Graph Traversal Algorithms :-

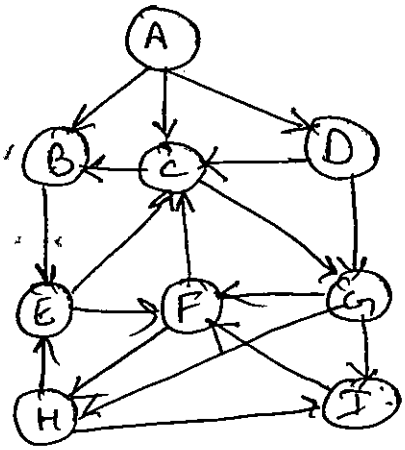
Traversing a graph means examining the nodes and edges of the graph.

There are two methods.

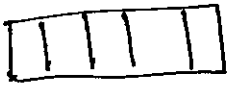
1. Breadth-first search
2. Depth-first Search.

BFS :— It is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes and so on until

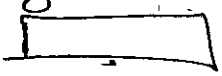
- Queue is used to implement BFS.



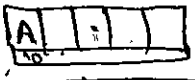
Step 1 :- Queue is initialized.

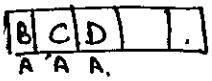


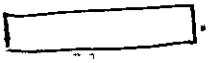
originate

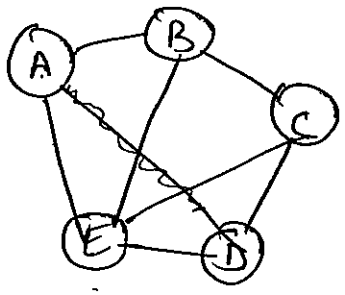


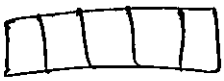
A to I.

1. 

2. 

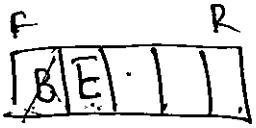
3. 



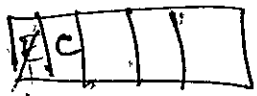
1. 

2. 

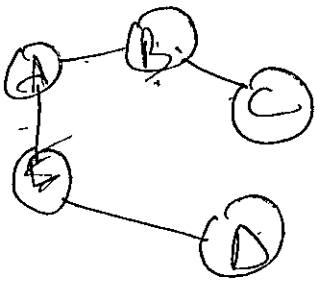
o/p.

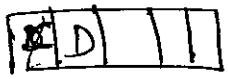
3. 

o/p A.

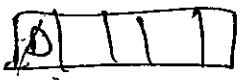
4. 

o/p AB



5. 

o/p ABE

6. 
o/p ABEC.

7. 
ABECD.

Applications :—

1. Finding all connected components
2. Finding the shortest path b/w two nodes.

BFS Alg :—

1. Get an array of size >= no. of vertices in the graph
2. Initialize queue to empty state.
3. Enqueue that first node & mark it as visited.
4. While queue is not empty, do steps 4(a) to 4(d)
   a) Dequeue value from queue.
   b) Enqueue all adjacent vertices into the queue.
   c) Mark them enqueued vertices as visited
   d) If no vertex is left to be enqueued repeat step 4a

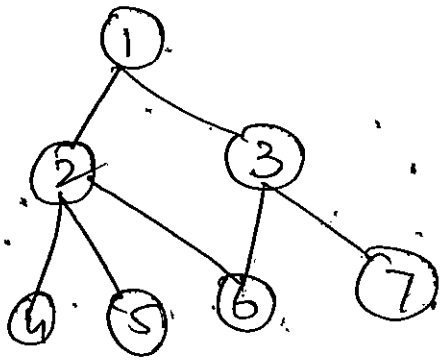
Depth First Search:-

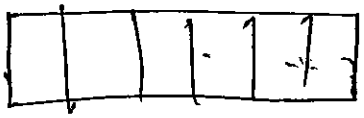
DFS. traverse the graph G, by expanding the starting node of G & then going deeper & deeper until the end of the node is encountered. When a dead-end is reached the alg backtracks, returning to the most recent node that has not been completely explored.

It uses Stack to implement.

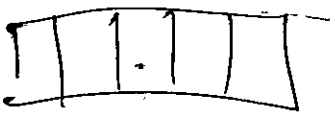
Alg

1. Initialize stack to empty
2. Initially all nodes of the graph are unvisited.
3. choose any node as start node & push this node onto the stack and mark it visited.
4. while (Stack is not empty)
   a) Pop value from stack
   b) All the nodes that are adjacent to this popped value & that are also not visited are pushed onto the stack.
   c) If there are no adjacent nodes, goto(a)



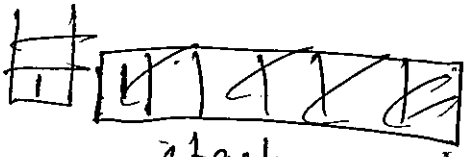


Stack is empty

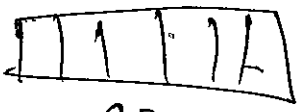


DFS array

Start node 1

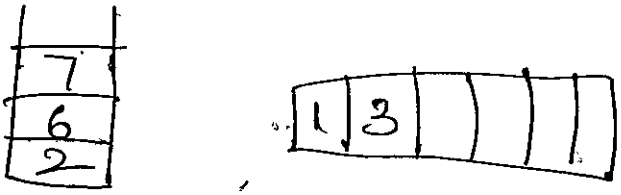


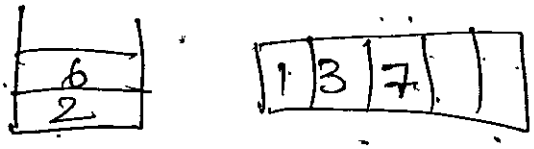
Stack

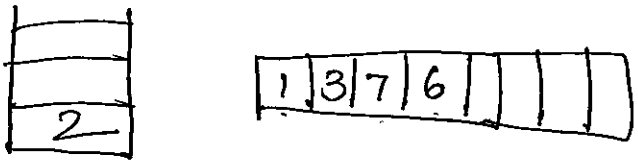


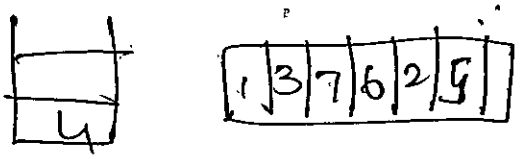
ar

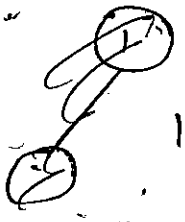
Stack







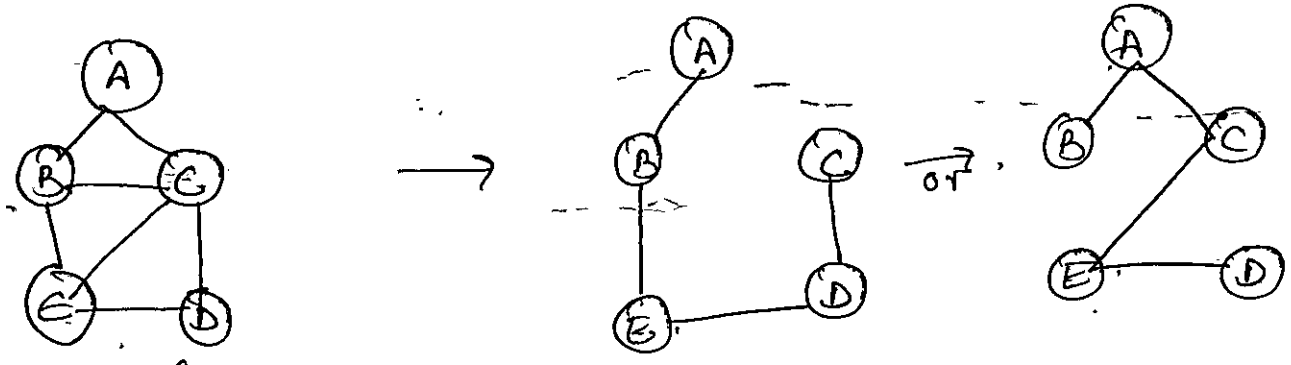




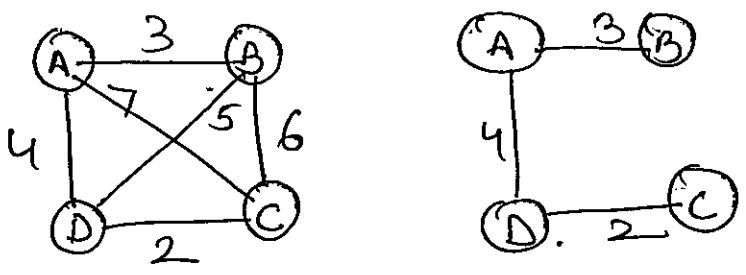
Applications of DFS:—

1. Finding a path b/w two specified nodes.
2. Finding whether a graph is connected or not.
3. Computing the spanning tree of a connected graph.

- **Spanning Tree:—** In an undirected graph, a spanning tree is a subgraph of graph 'G', which is a tree that connects all the vertices together.



- **Minimum Spanning Tree:—** It is a spanning tree that has weights associated with its edges & the total weight of the tree is minimum.



Applications :-

1. Used for designing n/w's.
2. Used to find the cheapest way to connect terminals.
3. applied in routing algorithms for finding the most efficient path.

To find minimum spanning trees two algorithms are used.

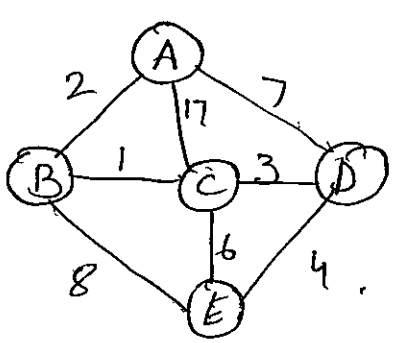
1. Kruskal
2. Prim's.

Kruskal Algorithm :-

Build a minimum spanning tree by adding the smallest weighted edge at each step provided it follows the follows two conditions:

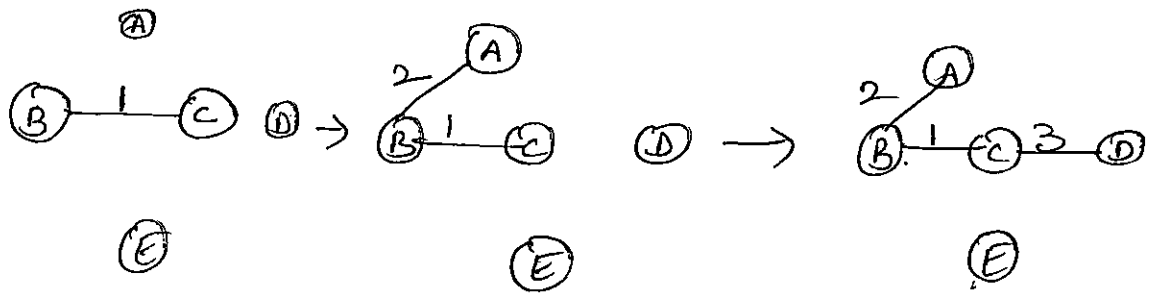
i) It has not been included previously.

ii) Addition of this edge does not create a cycle.

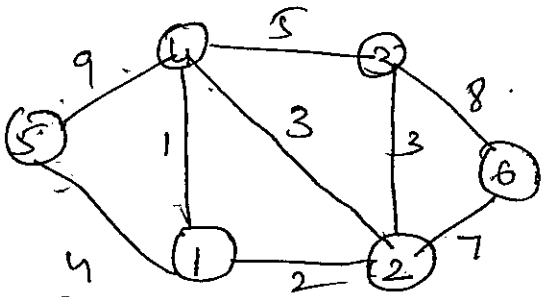


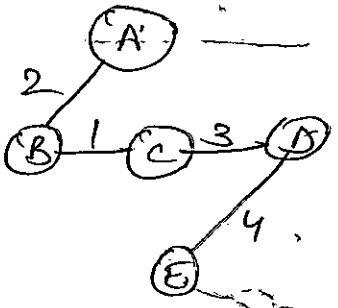
	A	B	C	D	E
A	0	2	17	7	0
B	2	0	1	0	8
C	17	1	0	3	6
D	7	0	3	0	4
E	0	8	6	4	0

1. Find the smallest weighted edge of the graph. This is ~~from~~ ~~B to C~~.
2. Add the edge to graph such that it should not form cycle.
3. Repeat the steps until n-1 edges are ~~to~~ attached.



Eg:-





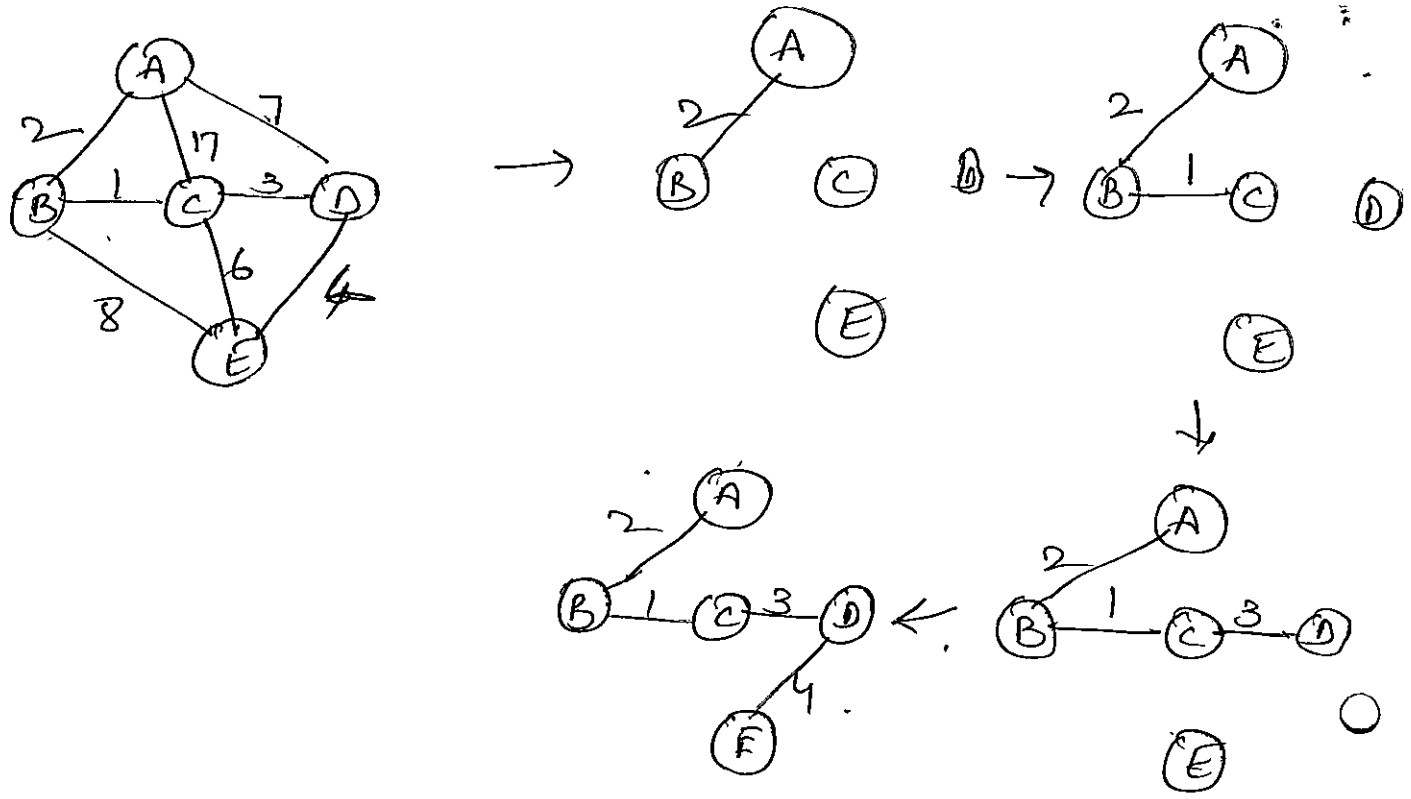
Prim's Algorithm:-

Step 1: Choose the edge vertex as starting node.

Step 2: Find the edge with a least cost from the starting vertex & include in the MST.

Step 3: For all the nodes in the MST, find the edge with the least weight from any of the nodes in MST.

Step 4: Repeat step 3 until ~~cycle is formed~~ all the vertices visited.



Dijkstra's Algorithm:—

Given a graph G(V, E), where 'V' is the set of vertices & 'E' is the set of edges in the graph.

Given a single vertex v, single source shortest path problem is to find the shortest path from 'v' to every other vertex in the graph.

Initially, each vertex is assigned as unreachable & cost of the path is given as infinite to reach that vertex.

Cost of source is given as zero.

An empty set is filled by traversing each vertex & by calculating its shortest

path from source. This set is completed when all the vertices are traversed.

Algorithm:—

1) consider source vertex as a, dis[v] is the cost of reaching vertex v from a, par[v] is the parent of v, weight[v,v] is the cost of traversing from v to v.

2. Initialize dis[v] = ∞ & par[v] = Nill & dis[a] = 0.

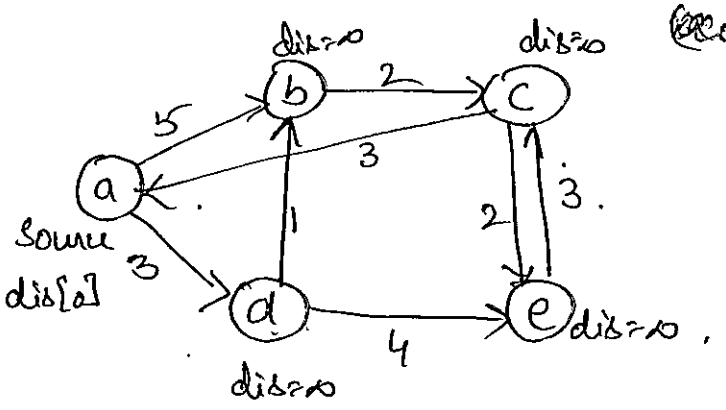
3. select v for which weight[a, v] is minimum.

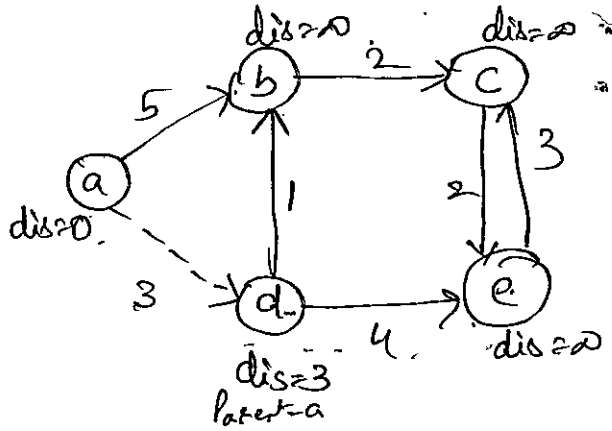
4. E = E ∪ [v].

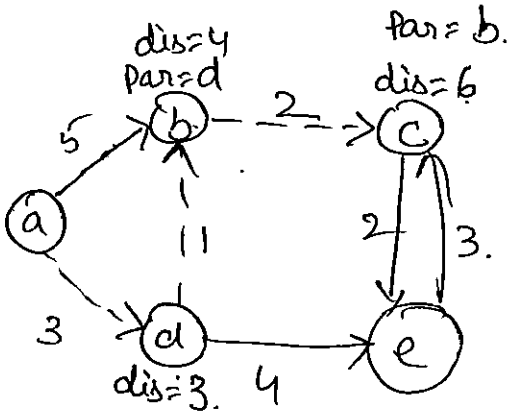
5. For each Vertex v connected with v if dis[v] > dis[v] + weight[v, v] then dis[v] = dis[v] + weight[v, v].

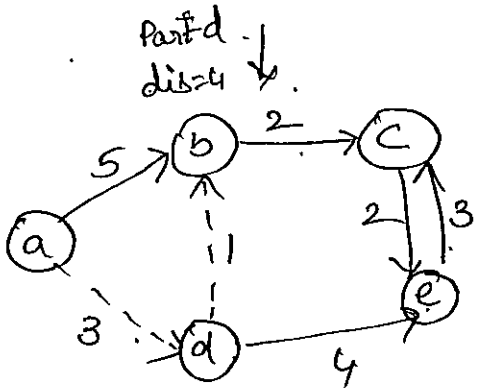
6. par[v] = v.

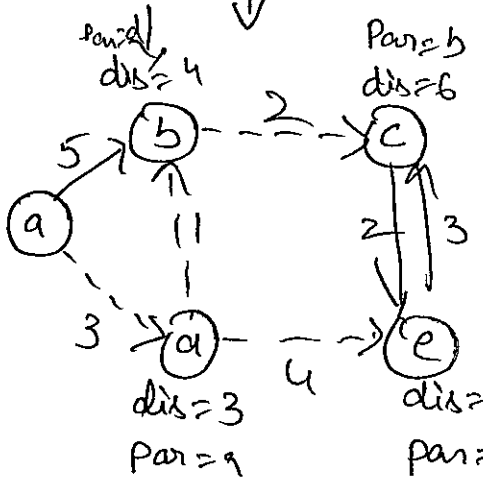
7. Repeat steps 3 to 6 until all the vertices are traversed.





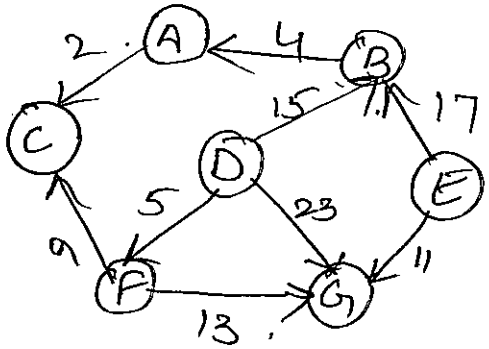


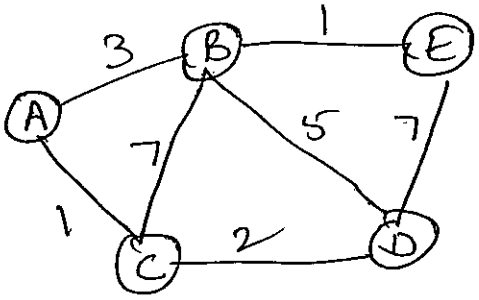


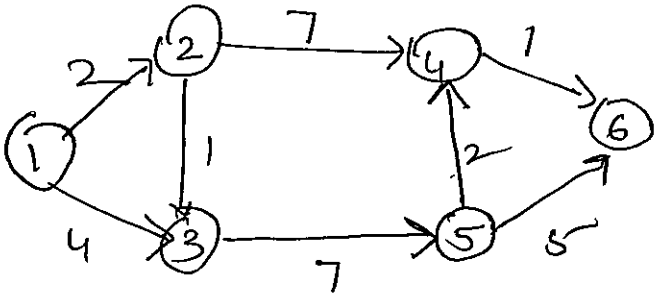


Path	dis
a-d	3
a-d-b	4
a-d-b-c	6
a-d-e	7.

Examples:







BFS

```
Void creategraph();
void bfs(int);
int g[10][10], n, visited[10], f=-1, r=-1, q[20];
void main()
{
  int v;
  clrscr();
  creategraph();
  printf("enter starting vertex");
  scanf("%d", &v);
  bfs(v);
  getch();
}
void creategraph()
{
int i, j;
printf("enter the no. of nodes");
scanf("%d", &n);
for(i=0; i<n; i++)
{
  for(j=0; j<n; j++)
  {
    printf("enter edge from %d to %d:", i, j);
    scanf("%d", &g[i][j]);
  }
}
for(i=0; i<n; i++)
  visited[i]=0;
}
void bfs(int v)
{ int i;
  q[++r]=v;
  visited[v]=1;
while(r!=f)
{
  v=q[++f];
  printf("%d->", v);
for(i=0; i<n; i++)
{ if(g[v][i]==1 && visited[i]==0)
  {
    q[++r]=i;
    visited[i]=1;
  }
}
}
```

ELECTION COMMISSION OF INDIA

FORM-6

(See Rules 13(1) and 26) of Registration of Electors Rule-1960

Acknowledgement No._____
(To be filled by office)

Application for Inclusion of Name in Electoral Roll for First time Voter OR on Shifting from One Constituency to Another Constituency.

To, The Electoral Registration Officer, ...Assembly / Parliamentary Consitituency

I request that my name be included in the electoral roll for the above Constituency. (Tick appropriate box)
As a first time voter ☐ **or due to shifting from another constituency** ☐
Particulars in support of my claim for inclusion in the electoral roll are given below:-

SPACE FOR PASTING ONE RECENT PASSPORT SIZE PHOTOGRAPH (3.5 CM X 3.5 CM) SHOWING FRONTAL VIEW OF FULL FACE WITHIN THIS BOX

Mandatory Particulars

(a) Name	
(b) Surname(if any)	
(c) Name and surname of Relative of Applicant [see item (d)]	
(d) Type of Relation (Tick appropriate box)	Father ☐ Mother ☐ Husband ☐ Wife ☐ Other ☐
(e) Age [as on 1st January of current calendar year.....]	Years ☐☐ Months ☐☐
(f) Date of Birth (in DD/MM/YYYY format)(if known)	☐☐/☐☐/☐☐☐☐
(g) Gender of Applicant (Tick appropriate box)	Male ☐ Female ☐ Third Gender ☐
(h)Current address where applicant is ordinarily resident	House No.
Street/Area/Locality	
Town/Village	
Post Office	Pin Code ☐☐☐☐☐☐
District	State/UT
(i) Permanent address of applicant	House No.
Street/Area/Locality	
Town/Village	
Post Office	Pin Code ☐☐☐☐☐☐
District	State/UT
(j)EPIC No. (if issued)	

Optional Particulars

(k) Disability (if any) (Tick appropriate box)	Visual impairment ☐ Speech & hearing disability ☐ Locomotor disability ☐ Other _____
(l) Email id (optional)	
(m) Mobile No. (optional)	☐☐☐☐☐☐☐☐☐☐

DECLARATION - *I hereby declare that to the best of knowledge and belief –*

(i) I am a citizen of India and place of my birth is Village/Town.....District.....State.....

(ii) I am ordinarily resident at the address given at (h) above since .....(date, month, year).

(iii)I have not applied for the inclusion of my name in the electoral roll for any other constituency.

**(iv)My name has not already been included in the electoral roll for this or any other assembly/* parliamentary *constituency*

OR

**My name may have been included in the electoral roll for_____ Constituency in _____ State in which I was ordinarily resident earlier at the address mentioned below and if so, I request that the same may be deleted from that electoral roll.*

** strike off the option not appropriate*

DFS

```
Void creategraph();
 void dfs(int);
int g[10][10], visited[10], n, st[10], top=-1
 main()
{
  int v;
  creategraph();
  Printf("enter the starting node");
  scanf("%d", &v);
   dfs(v);
   getch();
}
void creategraph()
{
}
void dfs(int v)
 {
   st[++top]=v;
   while(top!=-1)
   {
     v=st[top];
     top--;
    if(visited[v]==0)
   {  printf("%d", v);
   }  visited[v]=1;
for(i=0; i<n; i++)
{
if(g[v][i]==1 && visited[i]==0)
  {  st[++top]=i;
  }
}
}
}
```

Address of earlier place of ordinary residence (if applying due to shifting from another constituency)			
House No.		Street/Area/Locality	
Town/Village			
Post Office		Pin Code	
District		State/UT	

I am aware that making a statement or declaration which is false and which I know or believe to be false or do not believe to be true, is punishable under Section 31 of the Representation of the People Act, 1950 (43 of 1950).

Place.....

Date..... Signature of Applicant.....

Remarks of Field Level Verifying Officer:

Details of action taken
(To be filled by Electoral Registration Officer of the constituency)

The application of Shri / Shrimati/ Kumari .....for inclusion of name in the electoral roll in Form 6 has been accepted/ rejected. Detailed reasons for acceptance [under or in pursuance of rule 18/20/26(4)] or rejection [under or in pursuance of rule 17/20/26(4)] are given below:

Place:

Date: Signature of ERO Seal of the ERO

Intimation of decision taken (to be filled by Electoral Registration Officer of the constituency and to be posted to the applicant on the address as given by the applicant)

The application in Form 6 of Shri/Shrimati/Kumari.....

Postage Stamp to be affixed by the Electoral Registration Authority at the time of dispatch

Current address where applicant is ordinarily resident		House No.	
Street/Area/Locality			
Town/Village			
Post Office		Pin Code	
District		State/UT	

Has been (a) accepted and the name of Shri/Shrimati/Kumari.....

Has been registered at Serial No.....in Part No..... of AC No.....

(b) rejected for the reason.....

Date:

Electoral Registration Officer

Address.....

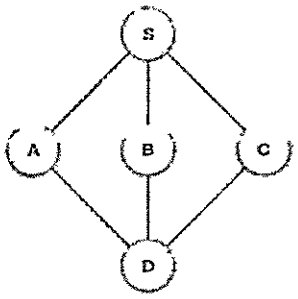
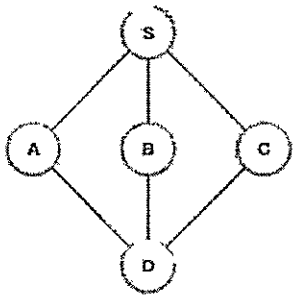
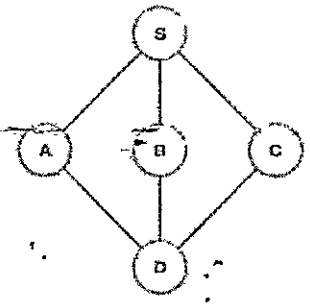
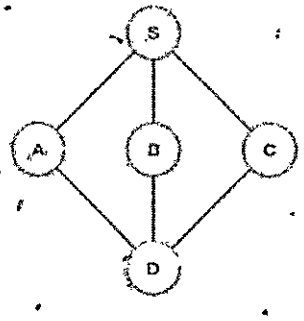
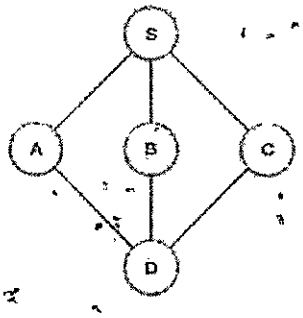
Acknowledgement/Receipt

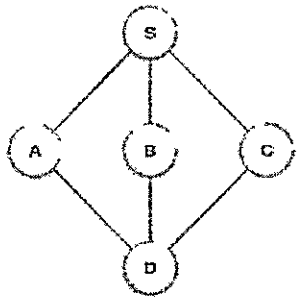
Acknowledgement Number _____ Date _____

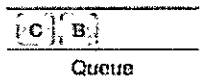
Received the application in form 6 of Shri / Smt. / Ms. _____
[Applicant can refer the Acknowledgement No. to check the status of application].

Name/Signature of ERO/AERO/BLO

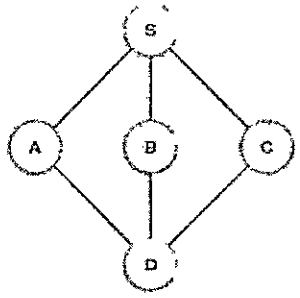
14,19,20,2 5,1,1,5)

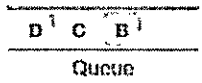
Step	Traversal	Description
1	 Queue	Initialize the queue.
2	 Queue	We start from visiting **S** (starting node), and mark it as visited.
3	 A Queue	We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4	 B A Queue	Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5	 C B A Queue	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

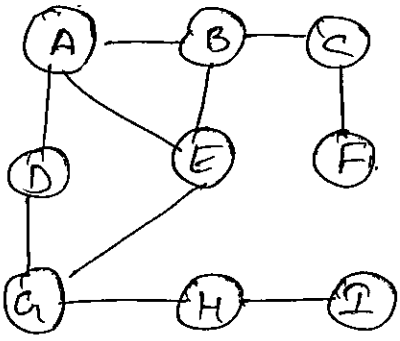


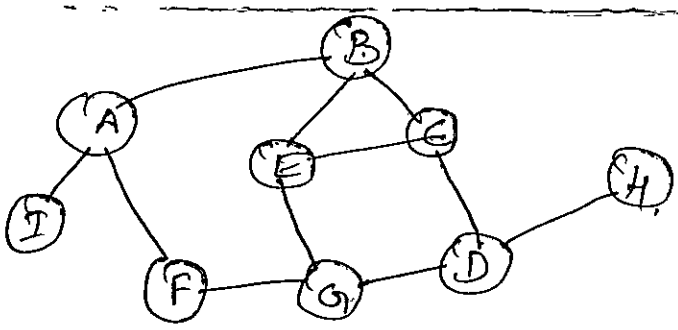


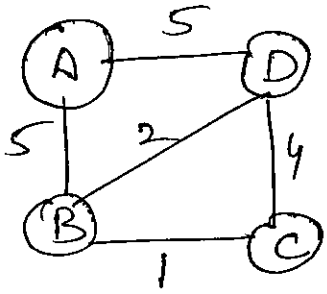
From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

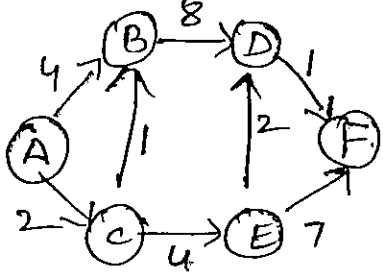
At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

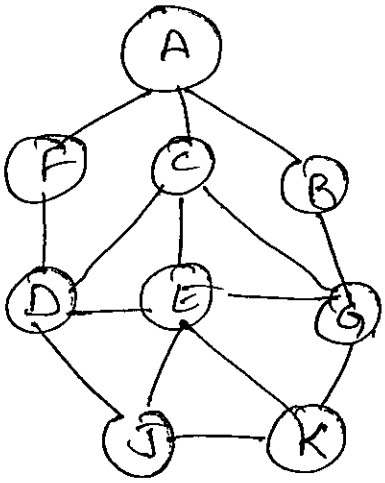
The implementation of this algorithm in C programming language can be seen here .

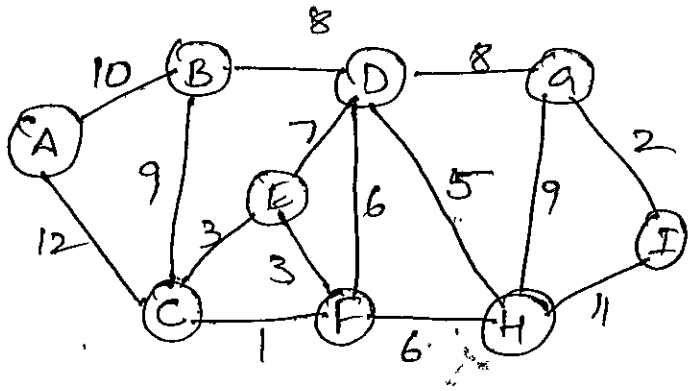












Unit-5

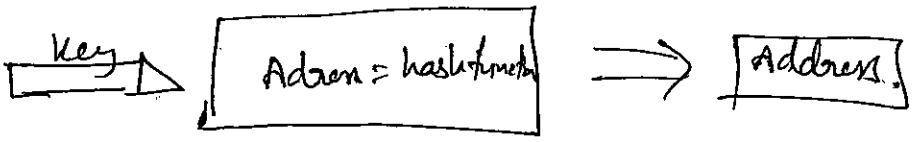
To search a value there are two algorithms Linear & binary search. Linear search takes O(n) & B.S takes O(log n) time complexity.

To search the value to O(1)

- There are two solutions.

First place the elements acording to the values. For example if the key values range from 0000 to 9999 but there are only 100 values the remaining space is wasted.

- Second solution, the elements are converted to a array index using a function. An array is considered as hash table & the function that will carry out the transformation will be called a hash function.



Hash Table :-

A hash table is a data structure that maps keys to array positions using hash function.

A value stored in a hash table can be searched in O(1) time by using a hash function, which generates an address from the key.

In a hash table an element with key K is stored at index h(K) and not K. This process of mapping the keys to appropriate locations in a hash table is called hashing.

If for two or more keys point to the same memory location a collision will occur.

Hash Function :-

A hash function is a mathematical formula which, when applied to key, produces an integer which can be used as an index for the key in the hash table.

The main aim of a hash function is that elements should be relatively, randomly & uniformly distributed.

Properties of a good hash function.

1. Low cost :- Cost of executing a hash function must be small.

2. Determinism :- A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value.

3. Uniformity :- A good hash function must map the keys as evenly as possible over its output range. It also.

- Different Hash Function :-

1. Division Method :-

This method divides x by M & then uses the remainder obtained.

$$h(x) = x \bmod M$$

- This method works very fast, but extra care should be taken to select a suitable value for M.

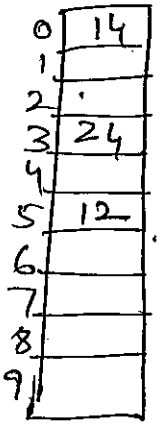
Choose M to be a prime number

Eg:- $M = 7$. $M = 11$

$h(14) = 14 \% 7 = 0$. $= 14 \% 11 = 3$.

$h(12) = 12 \% 7 = 5$. $12 \% 11 = 1$

$h(24) = 24 \% 7 = 3$. $24 \% 11 = 2$.



- Drawback is consecutive keys map to consecutive hash value.

2. Multiplication Method :-

Step1: Choose a constant A such that $0 < A < 1$.

Step2: Multiply the key by A.

Step3: Extract the fractional part of KA

Step 4: Multiply the result of step3 by the size of hash table.

$$h(k) = m (kA \bmod 1)$$

where $(KA \bmod 1)$ gives the fractional part of KA & m is the size of table.

Knuth suggested $A = (\text{Sqrt}5 - 1)/2$

$= 0.6180339887$.

Eg:- $m = 1000$, key = 12345.

$h(12345) = (1000 (12345 \times 0.618033 \bmod 1))$

$= (1000\ (7629.617385 \bmod 1))$

$= 1000 \times 0.617385$

$= 617.385$

$= 617.$

Mid-Square Method:-

Step 1: Square the value of the key i.e. k^2

Step 2: Extract the middle r digits of the result obtained

$$h(k) = s$$

where s is obtained by selecting r digits from k^2

Eg:- $M = 100.$

$K = 1234$

$k^2 = 1522756$

$r = 2$ - 3rd & 4th digits.

$h(k) = 27.$

Folding method:-

Step 1:- Divide the key value into a number of parts. That is divide k into parts $k_1, k_2 \text{----} k_n$ where each part has the same number of digits except the last

part which may have lesser digits than the other parts.

Step2: Add the individual parts, i.e. obtain sum of $k_1 + k_2 + \ldots + k_n$. The hash value is produced by ignoring the last carry, if any.

Eg:- 5678, 321. M = 100.

Parts — 56, 78.

56 + 78 = 134.

h(5678) = 34.

Parts — 32 1

32 + 1 = 33.

h(321) = 33.

Collisions :-

When the hash function maps two different keys to the same location then a collision will occur.

To solve the problem of collision also called collision resolution technique is applied.

There are two methods to solve.

1. Open addressing
2. chaining.

Collision Resolution by Open Addressing

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence & the next record is stored in that position.

The process of finding the next memory location to store the value is called probing.

Open addressing technique can be implemented using
1. Linear probing
2. Quadratic probing
3. double hashing
4. Rehashing.

Linear Probing:—

If a value is already stored at a location generated by $h(x)$, then following hash function is used to resolve the collision

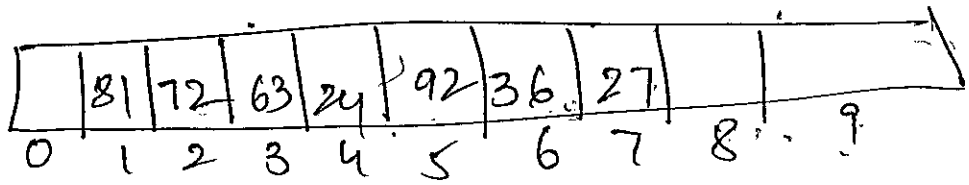
$$h(k, i) = (h'(k) + i) \bmod m$$

where m — size of table.
i — probe number that varies from 0 to $m-1$.

For a given key k, first location generated

by $[h'(k) \bmod m]$ is probed because for the first time $i=0$. If the location is free, value is stored in it, else the second probe generates the address of location given by $[h'(k)+1] \bmod m$ and so on, until a free location is found.

Eg:- 72, 27, 36, 24, 63, 81, 92, 101.



$h(72) = (72 \bmod 10 + 0) \bmod 10$.

$= 2 \bmod 10 = 2$.

$h(27) = (27 \bmod 10 + 0) \bmod 10$.

$= 7 \bmod 10 = 7$

$h(36) = (36 \bmod 10 + 0) \bmod 10$

$= 6 \bmod 10 = 6$.

$h(24) = (24 \bmod 10 + 0) \bmod 10$

$= 4 \bmod 10 = 4$.

$h(63) = (63 \bmod 10 + 0) \bmod 10$

$= 3$.

$h(81) = (81 \bmod 10 + 0) \bmod 10 = 1$.

$h(92) = (92 \bmod 10 + 0) \bmod 10$.

$= 2$.

$h(92) = (92 \bmod 10 + 1) \bmod 10$.

$= (2+1) \bmod 10 = 3$.

$h(92) = (92 \bmod 10 + 2) \bmod 10$

$= (2+2) \bmod 10 = 4$.

$h(92) = (92 \bmod 10 + 3) \bmod 10$.

$= (2+3) \bmod 10 = 5$.

- $h(101) = (101 \bmod 10 + 0) \bmod 10$.

$= 1$ (collision)

$h(101) = (101 \bmod 10 + 1) \bmod 10$.

$= (1+1) \bmod 10 = 2$

$h(101) = (101 \bmod 10 + 2) \bmod 10$.

$= (1+2) \bmod 10 = 3$.

- $h(101) = (101 \bmod 10 + 3) \bmod 10$.

$= (1+3) \bmod 10 = 4$.

$h(101) = (101 \bmod 10 + 4) \bmod 10$.

$= (1+4) \bmod 10 = 5$

$h(101) = (101 \bmod 10 + 5) \bmod 10$.

$= (1+5) \bmod 10 = 6$.

⋮

Search a value using linear probing. Same as for storing value in a hash table.

- The array index is computed & the key is compared with the value if match is found, then search is successful otherwise search function begins a sequential search of the array that continues until:
  - the value is found or
  - search function encounters a vacant location in the array, indicating that the value is not present, or.
  - the search function terminates because it reaches the end of the table & the value is not present.

Draw backs

1. higher risk of more collisions where one collision has already taken place.

Quadratic Probing:-

$$h(k,i) = [h'(k) + c_1 i + c_2 i^2] \bmod m.$$

where $h'(k) = (k \bmod m)$

$c_1 \neq c_2$, $\Rightarrow$ i is the probe number from 0 to $m-1$

Eg:- $m = 10$

72, 27, 36, 24, 63, 81, 101 , $c_1 = 1$, $c_2 = 3$

$h'(k) = k \bmod m$

$h(72, i) = [72 \bmod 10 + c_1 i + c_2 i^2] \bmod 10$, $i = 0$

$= 2$

$h(27,0) = [27 \bmod 10 + 1 \times 0 + 3 \times 0^2] \bmod 10$

$= 7$

$h(36,0) = 6$

$h(24,0) = 4$

$h(63,0) = 3$

$h(81,0) = 1$

$h(101,0) = [101 \bmod 10 + 1 \times 0 + 3 * 0^2] \bmod 10$

$= 1$

$(101, 1) = [101 \bmod 10 + 1 * 1 + 3 *] \bmod 10$

$= [1 + 1 + 3] \bmod 10$

$= 5 \bmod 10 = 5$

0	
1	81
2	72
3	63
4	24
5	101
6	36
7	27
8	
9	

- Quadratic probing resolves the primary clustering problem that exists in the linear probing
- Major drawback is a sequence of successive probes may explore a fraction of the table, & this fraction may be quite

small. If this happens then we will not be able to find an empty location in the table;

3. Double hashing:-

In this it uses two hash functions rather than a single function.

$h(k,i) = [h_1(k) + i h_2(k)] \bmod m$

where m is the size of hash table,

$h_1(k)$ & $h_2(k)$ are two hash functions.

given as $h_1(k) = k \bmod m$,

$h_2(k) = k \bmod m'$

i is the probe no. that varies from 0 to m-1

- m' is chosen to be less than m.

$m' = m-1$ or $m-2$

- It minimizes repeated collisions and the effects of clustering.
- It is a very efficient alg. it always require m to be a prime no.

4. Rehashing :-

When a hash table becomes nearly full, the no. of collisions increases, thereby degrading the performance of insertion & search operations. In such cases, to create a new hash table double the size of the original hash table.

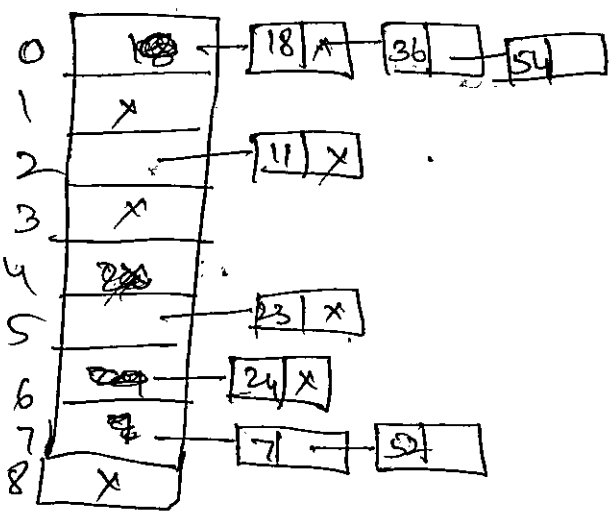
All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, & then inserting it in the new hash table.

Rehashing seems to be simple process, but it is quite expensive & must therefore not be done frequently.

Collision Resolution by chaining :-

In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location.

Eg:- 7, 24, 18, 52, 36, 54, 11 & 23 ; m = 9.



$h(7) = 7 \bmod 9$
$= 7.$

$h(24) = 24 \bmod 9.$
$= 6$

$h(18) = 18 \bmod 9$
$= 0$

$h(52) = 52 \bmod 9$
$= 7.$

$h(36) = 36 \bmod 9$
$= 0.$

$h(54) = 54 \bmod 9$
$= 0$

$h(11) = 11 \bmod 9$
$= 2.$

$h(23) = 23 \bmod 9$
$= 5$

- The main advantage is that it remains effective even when the no. of key values to be stored is much higher than the no. of locations in the hash table,
- Does not degrade the performance even when the table is half full.

- Disadv :- 1) using of linked list
2) traversing a linked list.

```
Void creategraph();
Void bfs()
Void display();
int g[10][10], n ;
Void main()
{
  int v;
  clrscr();
  creategraph()
  Pf("Starting vertex is");
  Scanf("
```

Adv & Disadv of Hashing:—

1. No extra space is required to store the index
2. Fast data access

1. Need hashing tech for inserting & retrieving data values
2. Choosing an effective hash function

Applications:—

1. Database indexing.
2. To implement compiler symbol table in C++.
3. Internet search engines.

A dictionary is a general purpose ds for storing a group of objects. A dictionary has a set of keys & each key has a single associated value.

Dictionaries are implemented as hash tables.

The keys in a dictionary must be simple types (such as integers or strings) while the values can be of any type.

Keys in a dictionary must be unique; an attempt to create a duplicate key will typically overwrite the existing value for that key.

//Program to implement Linear Program in Hashing

```
#include <stdio.h>
#include <conio.h>
int tsize;

int hasht(int key)
{
 int i ;
 i = key%tsize ;
 return i;
}
int rehashl(int key)
{
 int i ;
 i = (key+1)%tsize ;
 return i ;
}
void main()
{
   int key,arr[20],hash[20],i,n,s,op,j,k ;

   printf ("Enter the size of the hash table: ");
   scanf ("%d",&tsize);

   printf ("\nEnter the number of elements: ");
   scanf ("%d",&n);

   for (i=0;i<tsize;i++)
 hash[i]=-1 ;

   printf ("Enter Elements: ");
   for (i=0;i<n;i++)
   {
 scanf("%d",&arr[i]);
   }
   for(k=0;k<n;k++)
    {
 key=arr[k] ;
 i = hasht(key);
 while (hash[i]!=-1)
 {
    i = rehashl(i);
 }
 hash[i]=key ;
   }
   printf("\nThe elements in the array are: ");
   for (i=0;i<tsize;i++)
   {
 printf("\n  Element at position %d: %d",i,hash[i]);
   }
}
```

```
//Program to implement Quadratic Probing in hashing

#include <stdio.h>
#include <conio.h>
int tsize;

int hasht(int key)
{
 int i ;
 i = key%tsize ;
 return i;
}
int rehashq(int key, int j)
{
 int i ;
 i = (key+(j*j))%tsize ;
 return i ;
}
void main()
{
   int key,arr[20],hash[20],i,n,s,op,j,k ;

   printf ("Enter the size of the hash table:  ");
   scanf ("%d",&tsize);

   printf ("\nEnter the number of elements: ");
   scanf ("%d",&n);

   for (i=0;i<tsize;i++)
 hash[i]=-1 ;

   printf ("Enter Elements: ");
   for (i=0;i<n;i++)
   {
 scanf("%d",&arr[i]);
   }
  for(k=0;k<n;k++)
   {
 j=1;
 key=arr[k] ;
 i = hasht(key);
 while (hash[i]!=-1)
 {
   i = rehashq(i,j);
   j++ ;
 }
 hash[i]=key ;
   }
   printf("\nThe elements in the array are: ");
   for (i=0;i<tsize;i++)
   {
 printf("\n Element at position %d: %d",i,hash[i]);
   }
}
```

Extendible Hashing is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic changes.

A *hash table* in which the *hash function* is the last few bits of the *key* and the table refers to *buckets*. Table entries with the same final bits may use the same bucket. If a bucket overflows, it splits, and if only one entry referred to it, the table doubles in size. If a bucket is emptied by deletion, entries using it are changed to refer to an adjoining bucket, and the table may be halved.

Main features of Extendible Hashing: The main features in this hashing technique are:

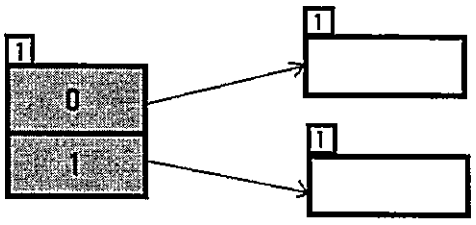
- **Directories:** The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.
- **Buckets:** The buckets are used to hash the actual data.

Example based on Extendible Hashing: Now, let us consider a prominent example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**

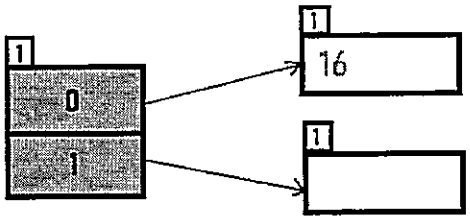
Bucket Size: 3 (Assume)

Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

- **Solution:** First, calculate the binary forms of each of the given numbers.
  16- 10000
  4- 00100
  6- 00110
  22- 10110
  24- 11000
  10- 01010
  31- 11111
  7- 00111
  9- 01001
  20- 10100
  26- 01101
- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:

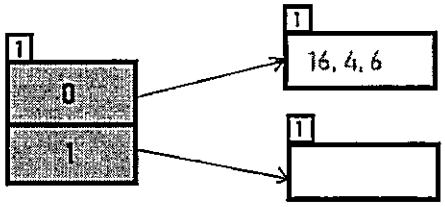


- **Inserting 16:**
  The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.



Hash(16)= 10000

- **Inserting 4 and 6:**
  Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows:

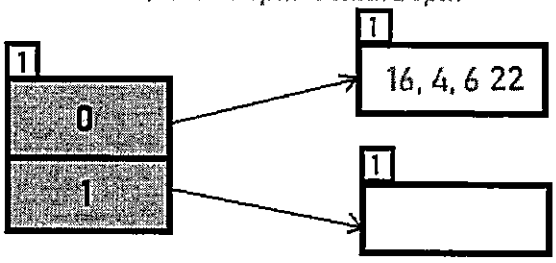


Hash(4)=100
Hash(6)=110

- **Inserting 22:** The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.

OverFlow Condition

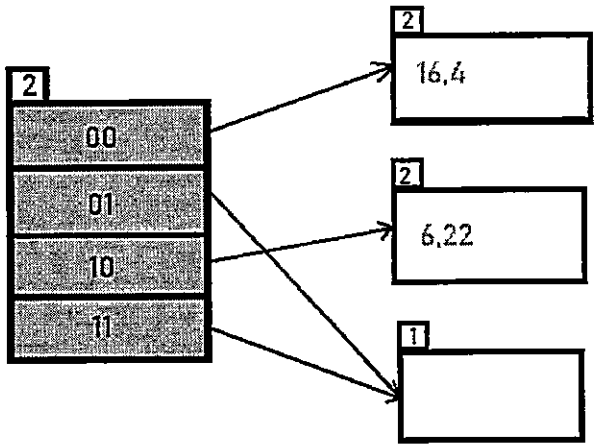
Here, Local Depth=Global Depth



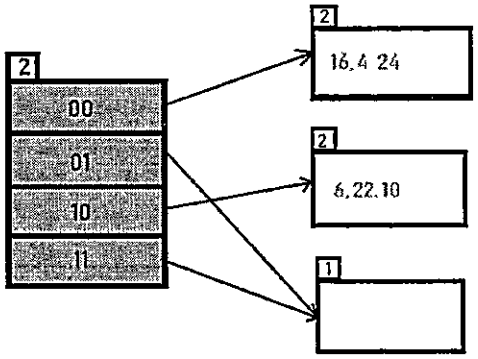
Hash(22)=10110

- As directed by **Step 7-Case 1**, Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now,the global depth is 2. Hence, 16,4,6,22 are now rehashed w.r.t 2 LSBs.[16(100**00**),4(1**00**),6(1**10**),22(1011**0**)]

After Bucket Split and Directory Expansion



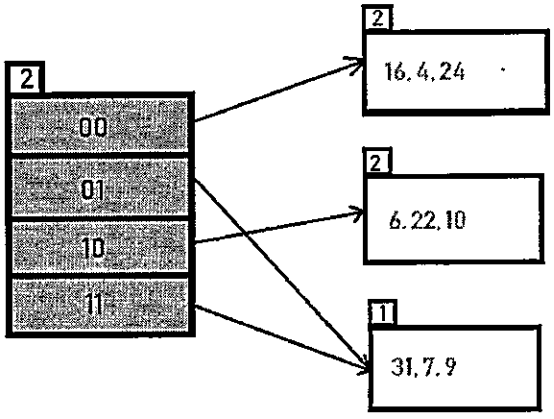
- **Notice that the bucket which was underflow has remained untouched. But, since the number of directories has doubled, we now have 2 directories 01 and 11 pointing to the same bucket. This is because the local-depth of the bucket has remained 1. And, any bucket having a local depth less than the global depth is pointed-to by more than one directories.*
- **Inserting 24 and 10:** 24(11**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



Hash(24)= 11000
Hash(10)=1010

- **Inserting 31,7,9:** All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow

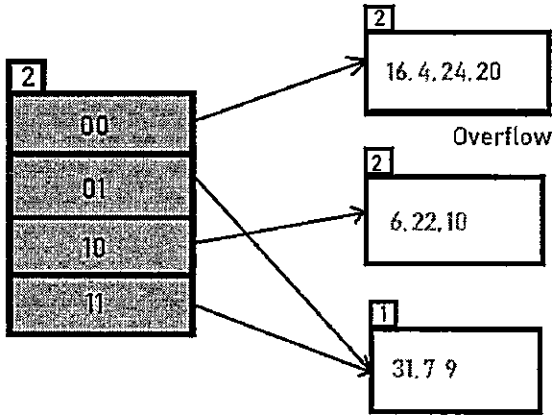
condition here.



Hash(31)= 11111
Hash(7)= 111
Hash(9)= 1001

- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.

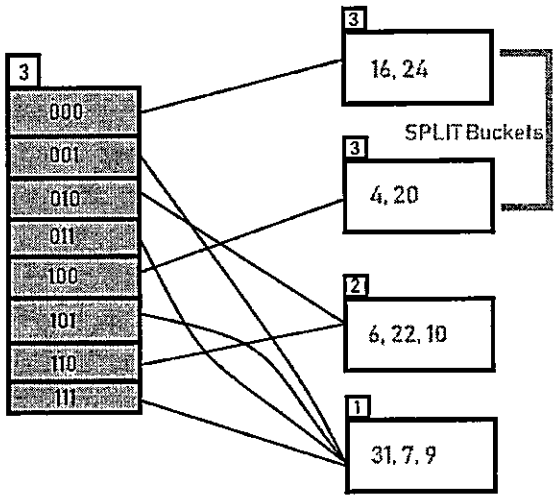
OverFlow, Local Depth= Global Depth



Hash(20)=10100

- 20 is inserted in bucket pointed out by 00. As directed by **Step 7-Case 1**, since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with

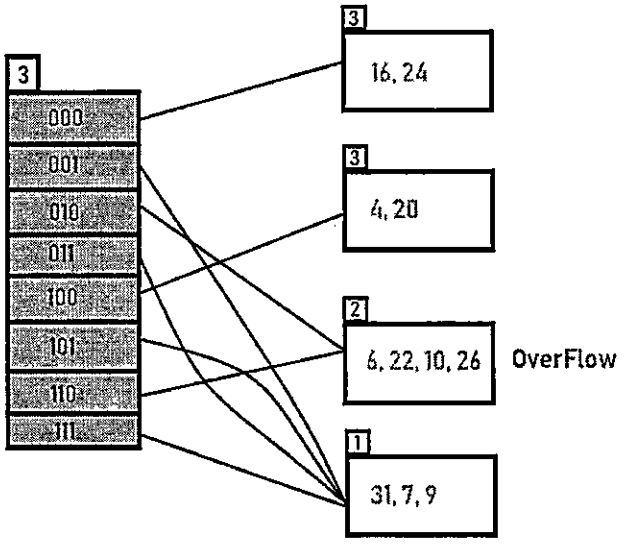
the new global depth. Now, the new Hash table looks like this:



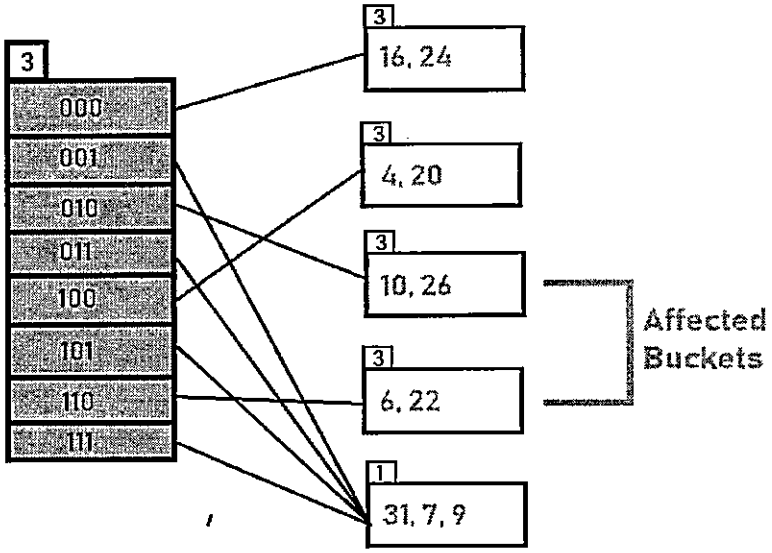
- **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.

Hash(26)=11010

OverFlow, Local Depth <Global Depth



- The bucket overflows, and, as directed by **Step 7-Case 2,** since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed. Finally, the output of hashing the given list of numbers is obtained.



- **Hashing of 11 Numbers is Thus Completed.**

Key Observations:

1. A Bucket will have more than one pointers pointing to it if its local depth is less than the global depth.
2. When overflow condition occurs in a bucket, all the entries in the bucket are rehashed with a new local depth.
3. If Local Depth of the overflowing bucket

is equal to the global depth, only then the directories are doubled and the global depth is incremented by 1.

4. The size of a bucket cannot be changed after the data insertion process begins.

Advantages:

1. Data retrieval is less expensive (in terms of computing).
2. No problem of Data-loss since the storage capacity increases dynamically.
3. With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.

Limitations Of Extendible Hashing:

1. The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
2. Size of every bucket is fixed.
3. Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
4. This method is complicated to code.

Power Point Presentation

Data Structures

Unit-1

Syllabus

- Unit-I: Data Structures, Stacks , Queues
- Unit-II:Trees
- Unit-III:Advanced Concepts in Tree
- Unit-IV:Graphs
- Unit-V:Hashing

Text Books:

- 1. Data Structures Using C, Second Edition Reema Thereja OXFORD higher Education
- 2. Data Structures, A Pseudo code Approach with C, Richard F. Gillberg & Behrouz A. Forouzan, Cengage Learning, India Edition, Second Edition, 2005.

Course Outcomes:

- At the end of the course student would be able to
- 1. Understand the concepts of Stacks and Queues with their applications.
- 2. Analyze Various operations on Binary trees.
- 3. Examine of various concepts of binary trees with real time applications.
- 4. Analyze the shortest path algorithm on graph data structures.
- 5. Outline the concepts of hashing, collision and its resolution methods using hash functions

Unit-1 Data Structure

- Data structure is a representation of data and the operations allowed on that data
- A data structure is a way to store and organize data in order to facilitate the access and modifications
- A **data structure** is a special way of organizing and storing **data** in a computer so that it can be used efficiently.

Advantages of DS

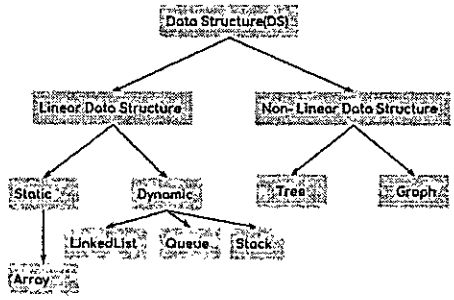
- **Data Organization:** Organizing the data so that it can accessed efficiently when we need that particular data.
- It is secure way of storage of data.
- It allows information stored on disk very efficiently

Data Structure Types

There are two types of data structures:

- 1. Linear Data Structure
  2. Non-linear Data Structure
- **Linear data structures:** Elements of Linear data structure are accessed in a sequential manner, however the elements can be stored in these data structure in any order.
  - Eg:LinkedList, Stack, Queue and Array
- **Non-linear data structures:** Elements of non-linear data structures are stores and accessed in non-linear order.
  - Eg:Tree and Graph

Classification of DS



Representation of Data Structure

- We can represent DS in Two ways:
- 1. Static Representation
- 2. Dynamic Representation

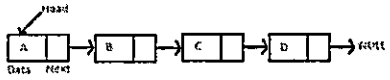
Static Representation:

In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.

Example of Static Data Structures: Array

Representation of Data Structure

- **Dynamic Representation:**
- In Dynamic data structure the size of the structure in not fixed and can be modified during the operations performed on it.
- Dynamic data structures are designed to facilitate change of data structures in the run time.
- Example of Dynamic Data Structures: Linked List



STATIC AND DYNAMIC DATA STRUCTURES

Static

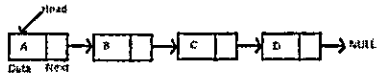
- A static data structure is designed to store a known number of data items.
- The values of the data can be changed but the memory size is fixed
- As static data structures store a fixed number of data items they are easier to program, as there is no need to check on the size of the data structure or the number of items stored.
- For example a high scores board on a video game may record the last 20 high scores. If a new one is entered the last one is lost!

Dynamic

- Dynamic data structures are designed to allow the data structure to grow or shrink at runtime.
- It is possible to add new elements or remove existing elements without having to consider memory space.
- Dynamic data structures make the most efficient use of memory but are more difficult to program, as you have to check the size of the data structure and the location of the data items each time you use the data.
- For example a shopping list may have different amounts of items on it each week. A program to make a shopping list will need to be dynamic and let the number of entries change!

Linked List

- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.



Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

Linked List

- **Advantages over arrays**
  1) Dynamic size
  2) Ease of insertion/deletion
- **Drawbacks of Linked list:**
  1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
  2) Extra memory space for a pointer is required with each element of the list.
  3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Linked List Representation

- A linked list is represented by a pointer to the first node of the linked list.
- The first node is called the head.
- If the linked list is empty, then the value of the head is NULL.
  Each node in a list consists of at least two parts:
  1) data
  2) Pointer (Or Reference) to the next node

- // A linked list node

```
struct Node
{
  int data;
  struct Node* next;
};
```

Types of Linked List

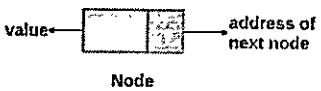
- Single linked List
- Double Linked List
- Circular Linked List

Basic Operations on Linked List

- **Traversal:** To traverse all the nodes one after another.
- **Insertion:** To add a node at the given position.
- **Deletion:** To delete a node.
- **Searching:** To search an element(s) by value.

Single Linked List

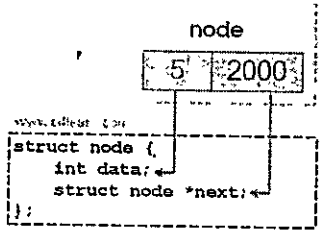
- A Singly-linked list is a collection of nodes linked together in a sequential way where each node of the singly linked list contains a data field and an address field that contains the reference of the next node
- The structure of the node in the Singly Linked List is



Node

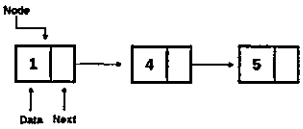
Single Linked List

A single linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.



Single Linked List

- The nodes are connected to each other in this form where the value of the next variable of the last node is NULL i.e. *next = NULL*, which indicates the end of the linked list.

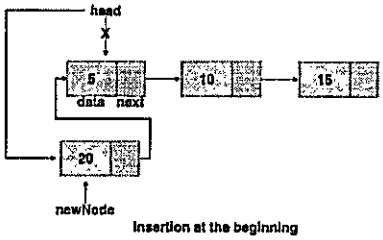


Operations on SLL

- Insertion
  - Insertion at the beginning
  - Insertion at the end. (Append)
  - Insertion after a given node
- Deletion
  - Delete at the beginning
  - Delete at the end.
  - Delete a given node
- Search
  - Search the node
- Traverse
  - To traverse all the nodes one after another.

Insertion

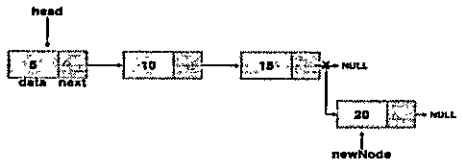
- *Insertion at the beginning:*
- If the list is empty, the new node becomes the head of the list. Otherwise, connect the new node to the current head of the list and make the new node, the head of the list.



Insertion at the beginning

Insertion

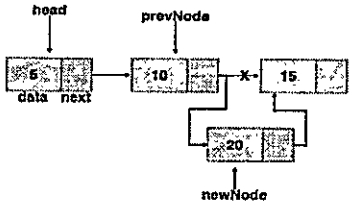
- *Insertion at end:*
- Traverse the list until find the last node. Then insert the new node to the end of the list.
- In case of a list being empty, return the updated head of the linked list because in this case, the inserted node is the first as well as the last node of the linked list.



Insertion at the end

Insertion

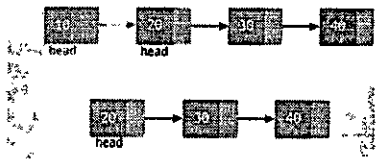
- *Insertion after a given node*
- We are given the reference to a node, and the new node is inserted after the given node.



Insertion after a given node

Deletion

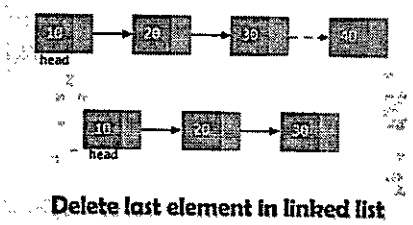
- *Delete a beginning node:*
- First node is pointed by the head, move the head to the next node, then free the first node
- If there is only one node then free the node and head becomes NULL



Delete first element in linked list

Deletion

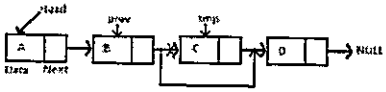
- *Delete the end node:*
- Move the pointer to the last node and free the node.
- Keep the NULL in the before that node



Delete last element in linked list

Deletion

- *Delete a particular node*
- To delete a node from linked list, do following steps.
  1) Find previous node of the node to be deleted.
  2) Change the next of previous node.
  3) Free memory for the node to be deleted.



Search

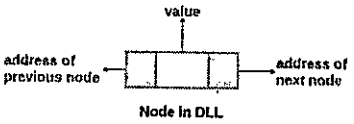
- To search any value in the linked list, traverse the linked list and compares the value present in the node.

Traverse

- To traverse all the nodes one after another.
- Start with the head of the list. Access the content of the head node if it is not null.
- Then go to the next node(if exists) and access the node information
- Continue until no more nodes (that is, you have reached the null node)

Double Linked List

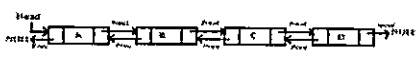
- A Doubly Linked List contains an extra memory to store the address of the previous node, together with the address of the next node and data which are there in the singly linked list. So, here we are storing the address of the next as well as the previous nodes.



Node in DLL

Double Linked List

- The nodes are connected to each other in this form where the first node has **prev = NULL** and the last node has **next = NULL.**



- **Advantages over Singly Linked List-**
- It can be traversed both forward and backward direction.
- **Disadvantages over Singly Linked List-**
- It will require more space as each node has an extra memory to store the address of the previous node.
- The number of modification increase while doing various operations like insertion, deletion, etc.

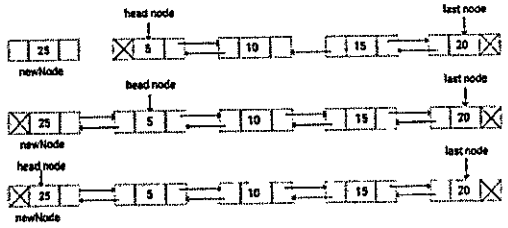
Double Linked List

- **Insertion**

A node can be added in four ways
1) At the front of the DLL
2) After a given node.
3) At the end of the DLL
4) Before a given node.

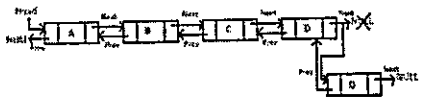
Insertion in DLL

- *At the front of the DLL:*
- The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL.



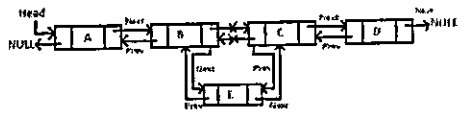
Insertion in DLL

- *At the end of the DLL:*
- A Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.



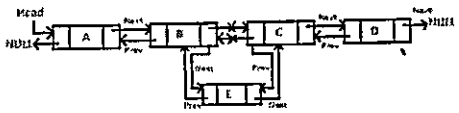
Insertion in DLL

- *Add a node after a given node:*
- We are given pointer to a node as prevnode, and the new node is inserted after the given node.



Insertion in DLL

- *Add a node before a given node:*
- We are given pointer to a node as prevnode, and the new node is inserted after the before given node.

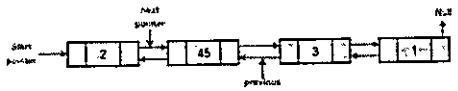


Double linked list

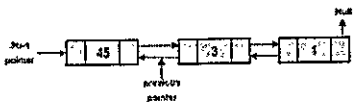
- Deletion
  - Delete the front node
  - Delete the end node
  - Delete the given node

Deletion in DLL

- *Delete a First node:*
- If the node to be deleted is the head node then make the next node as head

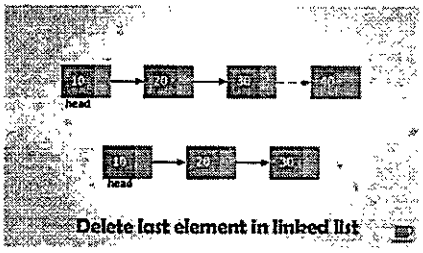


After deleting the first node the linked list becomes:



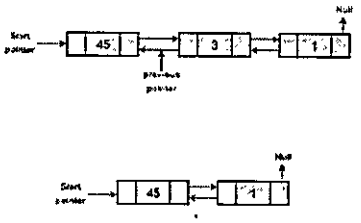
Deletion in DLL

- *Deleting the last node:*



Deletion in DLL

- *Deleting a given node:*



Search

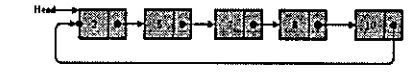
- To search any value in the linked list, traverse the linked list and compares the value present in the node.

Traverse

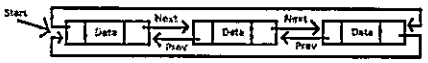
- To traverse all the nodes one after another.
- Start with the head of the list. Access the content of the head node if it is not null.
- Then go to the next node(if exists) and access the node information
- Continue until no more nodes (that is, you have reached the null node)

Circular Linked List

- **Circular linked list** is a linked list where all nodes are connected to form a circle. There is no NULL at the end.
- A circular linked list can be a singly circular linked list or doubly circular linked list.



Circular Single linked List



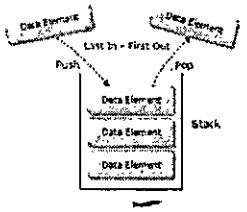
Circular Double linked List

Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2) Useful for implementation of queue. No need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
3) Circular lists are useful in applications to repeatedly go around the list.

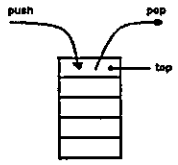
Stacks

- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).



Operations on Stack

- Elements can be inserted or deleted only from one end of the stack i.e. from the top.
- The element at the top is called the *top* element.
- The operations of inserting and deleting elements are called **push()** and **pop()** respectively.



Operations on Stack

- PUSH:
  - Push operation refers to inserting an element in the stack.
  - The new element is inserted at the top of the stack.
- POP:
  - Pop operation Removes an element from the stack
  - The top of the element is removed
- PEEK:
  - Get the top data element of the stack, without removing it.

Implementation of Stack

- Stack can be implemented in two ways:
  - 1.Using Array
  - 2. Using Linked List

Array Implementation:

PUSH Operation:

- The size of the stack is specified at the beginning
- Initially stack is empty and top=-1
- When a new element is pushed it checks the condition whether the stack is full,if it is true it gives "Overflow"
- Otherwise the element is inserted from top end
- Top variable is increased by 1

Array Implementation-PUSH

Step 1 - Check whether stack is FULL. (top == SIZE-1)
Step 2 - If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.
Step 3 - If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

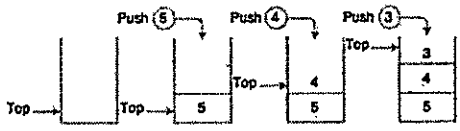
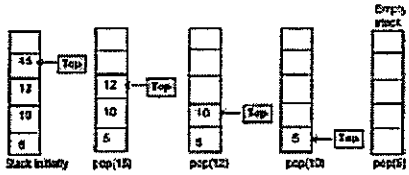


Fig. Insertion of Elements in a Stack

Array Implementation-POP

- Pop():
  - To delete an element from any , it checks whether the stack is empty, if it is true it gives "Underflow"
  - Otherwise top of the element is removed from the stack
  - Top is decreased by 1
  - If there are no more elements top will becomes -1



Pop

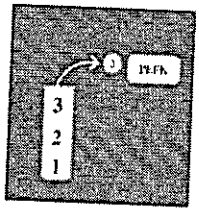
- **Step 1 - Check whether stack is EMPTY. (top == -1)**
- **Step 2 - If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function.**
- **Step 3 - If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).**

Display

- **Step 1 - Check whether stack is EMPTY. (top == -1)**
- **Step 2 - If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.**
- **Step 3 - If it is NOT EMPTY, then define a variable 'i' and initialize with top. Display stack[i] value and decrement i value by one (i--).**
- **Step 3 - Repeat above step until i value becomes '0'.**

Array Implementation-PEEK

- Peek:
  - If the stack is empty it displays "**Stack is empty**"
  - Otherwise it displays the top of the element in the stack

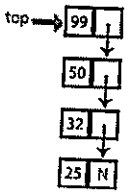


Linked List Representation

- The main advantage of using linked list over an arrays is that it is possible to implements a stack that can shrink or grow as much as needed.
- In using array will put a restriction to the maximum capacity of the array which can lead to stack overflow.
- Here each new node will be dynamically allocate.
- so overflow is not possible.
- The stack implemented using linked list can work for an unlimited number of values.

Linked List Representation

- A stack can be easily implemented through the linked list.
- In stack implementation, a stack contains a top pointer, which is "head" of the stack where pushing and popping items happens at the head of the list.
- First node have null in link field and second node link have first node address in link field and so on and last node address in "top" pointer.



PUSH

- **Step 1** - Create a **newNode** with given value.
- **Step 2** - Check whether stack is **Empty (top == NULL)**
- **Step 3** - If it is **Empty**, then set **newNode → next = NULL.**
- **Step 4** - If it is **Not Empty**, then set **newNode → next = top.**
- **Step 5** - Finally, set **top = newNode.**

POP

- **Step 1** - Check whether **stack is Empty (top == NULL).**
- **Step 2** - If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3** - If it is **Not Empty**, then define a **Node** pointer **'temp'** and set it to **'top'.**
- **Step 4** - Then set **'top = top → next'.**
- **Step 5** - Finally, delete **'temp'. (free(temp)).**

Display

- **Step 1** - Check whether stack is **Empty (top == NULL).**
- **Step 2** - If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
- **Step 3** - If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top.**
- **Step 4** - Display **'temp → data --->'** and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. **(temp → next != NULL).**
- **Step 5** - Finally! Display **'temp → data ---> NULL'.**

Applications of Stack

- Stacks can be used for **expression evaluation.**
- Stacks can be used to check parenthesis matching in an expression.
- Stacks can be used for Conversion from one form of expression to another.
- Stacks can be used for Memory Management.
- Stack data structures are used in backtracking problems.

Representation of an expression

- An expression can be represented in three ways
  - Infix expression
  - Prefix expression
  - Postfix expression

- **Infix Notation**
  - Operators are written in-between their operands
  - This is the usual way we write expressions
  - Eg: A * (B + C) / D
- **Post fixNotation**
  - Operators are written after their operands
  - The postfix notation is called as suffix notation and is also referred to reverse polish notation.
  - Eg: A B C + * D /
- **Prefix Notation**
  - Operators are written before their operands
  - The prefix notation is called as polish notation
  - Eg: / * A + B C D

Infix	Prefix	Postfix
a+b	+ba	ab+
(a+b)*(c+d)	*+dc+ba	ab+cd+*
b*b-4*a*c	-*c*a4*bb	bb*4a*c*-

Algorithm to convert Infix to Postfix

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.
5. If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.
6. Repeat the process until the last symbol

Infix Exp: (A+(B*C-(D/E^F)*G)*H)

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' is at higher precedence than '+'
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that's why '^' Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that's why '^' Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form.

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

So, the Postfix Expression is 23*21-/53*+

Infix to Prefix

Step 1. Reverse the input string
Step 2. Examine the next element in the input
Step 3. If it is operand add it to the output string
Step 4. If it is closing parenthesis push it on stack
Step 5. If it is an operator then
  a. If stack is empty, push operation on stack
  b. If the top of stack is ")" push operator on stack
  c. If it has same or higher priority then the top of stack, push it
  d. Else pop the operator & add it to output string, repeat 5
Step 6. If it is "(" pop operator and add them to string s until a ")" is encountered. POP and discard ")"
Step 7. If there is more input go to step 2
Step 8. If there is no more input, unstack the remaining operators & add them
Step 9. Reverse the output string

(a + (b * c) / (d – e)) = +a/*bc-de

NOTE: scan the infix string in reverse order.

SYMBOL	PREFIX	OPSTACK
)	Empty	)-
)	Empty	))
e	e	))
-	e	)-
d	de	)-
(	-de	)
/	-de	)/
)	-de	)/)
c	c-de	)/)
*	c-de	)/)*
b	bc-de	)/)*
(	*bc-de	)/
+	/*bc-de	)+
a	a/*bc-de	)+
(	+a/*bc-de	Empty

More Problems

- The following infix expressions convert to Postfix and Prefix expressions
- 1. (A + B) * (C + D)
- 2. (A - B/C) * (A/K-L)
- 3.(1+4/2*1+2)*3/2
- 4.9*(12+4)/5*6
- 5.(a*b/c)-(d^e+f)

Postfix Evaluation

- A postfix expression can be evaluated using the Stack data structure. Follow the below steps

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Postfix Evaluation

- 456*+

Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No more elements(pop)	Empty	34(Result)

Applications of Stack

- Stacks can be used for **expression evaluation.**
- Stacks can be used to check parenthesis matching in an expression.
- Stacks can be used for Conversion from one form of expression to another.
- Stacks can be used for Memory Management.
- Stack data structures are used in backtracking problems.

Representation of an expression

- An expression can be represented in three ways
  - Infix expression
  - Prefix expression
  - Postfix expression

Prefix Evaluation

- Step 1: Put a pointer P at the end of the end
- Step 2: If character at P is an operand push it to Stack
- Step 3: If the character at P is an operator pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack
- Step 4: Decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression.
- Step 5: The Result is stored at the top of the Stack, return it Step 6: End

Prefix Evaluation

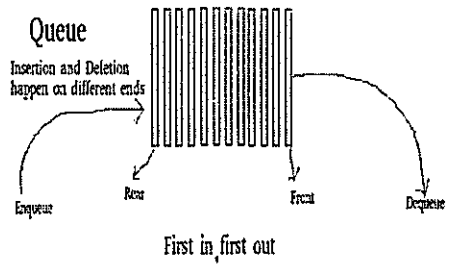
Example: -*-4325

Symbol	opnd1	opnd2	value	opndstack
5				5
2				5, 2
3				5, 2, 3
4				5, 2, 3, 4
-	4	3	7	5, 2
*	7	2	14	5, 7
-	14	5	9	5, 14
				9

result

Queue

- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- In a queue data structure, adding and removing elements are performed at two different positions.
- The insertion is performed at one end and deletion is performed at another end.
- In a queue data structure, the insertion operation is performed at a position which is known as **'rear'** and the deletion operation is performed at a position which is known as **'front'**.
- In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.



Operations on a Queue

- The following operations are performed on a queue data structure...
- **enQueue(value) - To insert an element into the queue**
- **deQueue() - To delete an element from the queue**
- **display() - (To display the elements of the queue)**

Implementation of Queue

- Queue data structure can be implemented in two ways. They are as follows...
  - **1.Using Array**
  - **2.Using Linked List**
- When a queue is implemented using an array, that queue can organize an only limited number of elements.
- When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

Data Structures

Unit-1

Queue Operations using Array

- Queue data structure using array can be implemented as follows... **Step 1** - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2** - Declare all the **user defined functions** which are used in queue implementation.
- **Step 3** - Create a one dimensional array with above defined SIZE **(int queue[SIZE])**
- **Step 4** - Define two integer variables **'front'** and **'rear'** and initialize both with **'-1'**. **(int front = -1, rear = -1)**
- **Step 5** - Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

Insertion

- In a queue data structure, enQueue() is a function used to insert a new element into the queue.
- In a queue, the new element is always inserted at **rear** position.
- **Step 1** - Check whether **queue** is **FULL. (rear == SIZE-1)**
- **Step 2** - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3** - If it is **NOT FULL**, then increment **rear** value by one **(rear++)** and set **queue[rear] = value.**

Deletion

- In a queue data structure, deQueue() is a function used to delete an element from the queue.
- In a queue, the element is always deleted from **front** position
- **Step 1** - Check whether **queue** is **EMPTY. (front == rear)**
- **Step 2** - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3** - If it is **NOT EMPTY**, then increment the **front** value by one **(front ++)**. Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal **(front == rear)**, if it **TRUE**, then set both **front** and **rear** to **'-1' (front = rear = -1)**.

Display

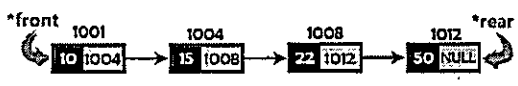
- Use the following steps to display the elements of a queue...
- **Step 1** - Check whether **queue** is **EMPTY. (front == rear)**
- **Step 2** - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3** - If it is **NOT EMPTY**, then define an integer variable **'i'** and set **'i = front+1'**.
- **Step 4** - Display **'queue[i]'** value and increment **'i'** value by one **(i++)**. Repeat the same until **'i'** value reaches to **rear (i <= rear)**

Queue using Linked list

- The major problem with the queue implemented using an array is, It will work for an only fixed number of data values.
- That means, the amount of data must be specified at the beginning itself.
- Queue using an array is not suitable when we don't know the size of data which we are going to use.
- A queue data structure can be implemented using a linked list data structure.
- The queue which is implemented using a linked list can work for an unlimited number of values.

Queue Operations using Linked list

- In linked list implementation of a queue, the last inserted node is always pointed by **'rear'** and the first node is always pointed by **'front'**



Operations

- To implement queue using linked list, we need to set the following things before implementing actual operations.
- **Step 1** - Include all the **header files** which are used in the program. And declare all the **user defined functions.**
- **Step 2** - Define a **'Node'** structure with two members **data** and **next.**
- **Step 3** - Define two **Node** pointers **'front'** and **'rear'** and set both to **NULL.**
- **Step 4** - Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

- use the following steps to insert a new node into the queue...
- **Step 1** - Create a **newNode** with given value and set **'newNode → next'** to **NULL.**
- **Step 2** - Check whether queue is **Empty (rear == NULL)**
- **Step 3** - If it is **Empty** then, set **front = newNode** and **rear = newNode.**
- **Step 4** - If it is **Not Empty** then, set **rear → next = newNode** and **rear = newNode.**

deQueue() - Deleting an Element from Queue

- use the following steps to delete a node from the queue...
- **Step 1** - Check whether **queue** is **Empty (front == NULL).**
- **Step 2** - If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3** - If it is **Not Empty** then, define a Node pointer **'temp'** and set it to **'front'.**
- **Step 4** - Then set **'front = front → next'** and delete **'temp' (free(temp)).**

display() - Displaying the elements of Queue

- Use the following steps to display the elements (nodes) of a queue...
- **Step 1** - Check whether queue is **Empty (front == NULL).**
- **Step 2** - If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3** - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front.**
- **Step 4** - Display **'temp → data --->'** and move it to the next node. Repeat the same until **'temp'** reaches to **'rear' (temp → next != NULL).**
- **Step 5** - Finally! Display **'temp → data ---> NULL'.**

Applications of Queue

- 1. It is used to schedule the jobs to be processed by the CPU.
- 2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
- 3. Breadth first search uses a queue data structure to find an element from a graph.

Types of Queue

- **There are four types of Queue:**
- Simple Queue
- Circular Queue
- Priority Queue
- Dequeue (Double Ended Queue)

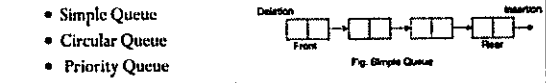


Fig. Simple Queue

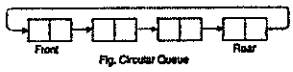


Fig. Circular Queue

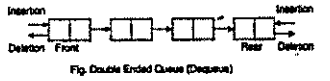
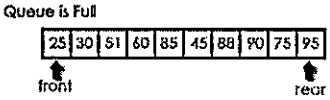


Fig. Double Ended Queue (Dequeue)

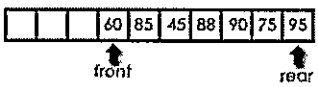
Circular Queue

- In a normal Queue Data Structure, we can insert elements until queue becomes full.
- But once the queue becomes full, we can not insert the next element until all the elements are deleted from the queue.
- The queue after inserting all the elements into it is as follows



Circular Queue

- After deleting three elements from the queue...



- In the above situation, even though we have empty positions in the queue we can not make use of them to insert the new element.
- This is the major problem in a normal queue data structure.
- To overcome this problem we use a circular queue data structure.

Circular Queue

- A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- Graphical representation of a circular queue is as follows..

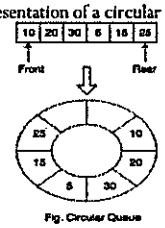


Fig. Circular Queue

Implementation of Circular Queue

- To implement a circular queue data structure using an array
- **Step 1** - Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2** - Declare all **user defined functions** used in circular queue implementation.
- **Step 3** - Create a one dimensional array with above defined SIZE **(int cQueue[SIZE])**
- **Step 4** - Define two integer variables **'front'** and **'rear'** and initialize both with **'-1'. (int front = -1, rear = -1)**
- **Step 5** - Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

- In a circular queue, enQueue() is a function which is used to insert an element into the circular queue.
- In a circular queue, the new element is always inserted at **rear** position.
- **Step 1** - Check whether **queue** is **FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))**
- **Step 2** - If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3** - If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1.**
- **Step 4** - Increment **rear** value by one **(rear++)**, set **queue[rear] = value** and check **'front == -1'** if it is **TRUE**, then set **front = 0.**

deQueue() - Deleting a value from the Circular Queue

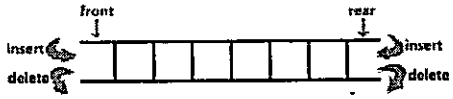
- In a circular queue, deQueue() is a function used to delete an element from the circular queue.
- In a circular queue, the element is always deleted from **front** position.
- **Step 1** - Check whether **queue** is **EMPTY. (front == -1 && rear == -1)**
- **Step 2** - If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3** - If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one **(front ++)**. Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0.** Then check whether both **front - 1** and **rear** are equal **(front -1 == rear)**, if it **TRUE**, then set both **front** and **rear** to **'-1' (front = rear = -1).**

display() - Displays the elements of a Circular Queue

- Step 1 - Check whether queue is EMPTY. (front == -1)
- Step 2 - If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
- Step 3 - If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front'.
- Step 4 - Check whether 'front <= rear', if it is TRUE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.
- Step 5 - If 'front <= rear' is FALSE, then display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i <= SIZE - 1' becomes FALSE.
- Step 6 - Set i to 0.
- Step 7 - Again display 'cQueue[i]' value and increment i value by one (i++). Repeat the same until 'i <= rear' becomes FALSE.

Double Ended Queue-Deque

- Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**).
- That means, we can insert at both front and rear positions and can delete from both front and rear positions.



Operations on Deque

Mainly the following four basic operations are performed on queue:

- ***insertFront()***: Adds an item at the front of Deque.
- ***insertLast()***: Adds an Item at the rear of Deque.
- ***deleteFront()***: Deletes an item from front of Deque.
- ***deleteLast()***: Deletes an item from rear of Deque.

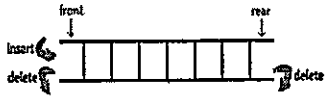
Deque

- Double Ended Queue can be represented in TWO ways, those are as follows...

- Input Restricted Double Ended Queue
- Output Restricted Double Ended Queue

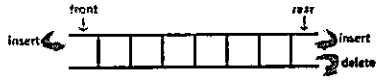
Input Restricted Double Ended Queue

- In input restricted double-ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

- In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.

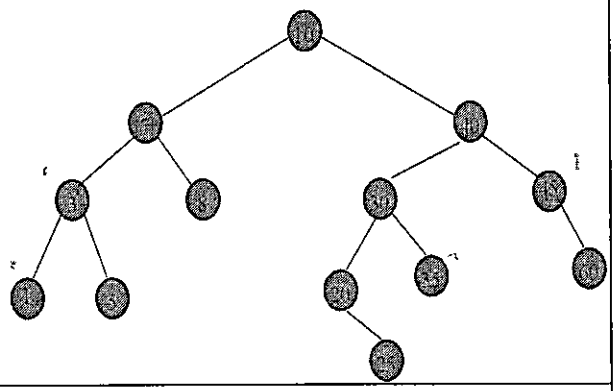


AVL Trees

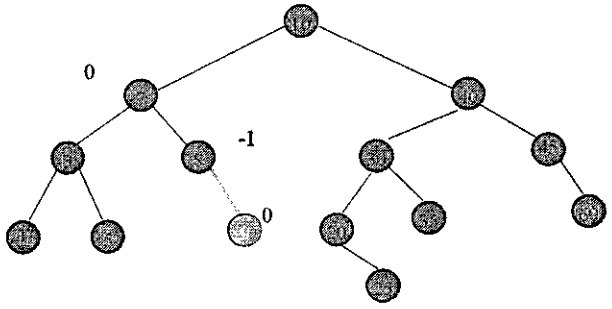
- binary tree
- for every node x

  balance factor of x = height of left subtree of x
  – height of right subtree of x
- balance factor of every node x – 1, 0, or 1
- $\log_2 (n+1) <= \text{height} <= 1.44 \log_2 (n+2)$

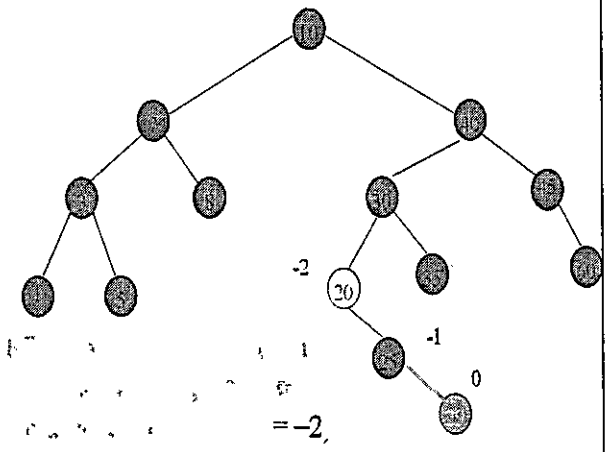
Example AVL Tree



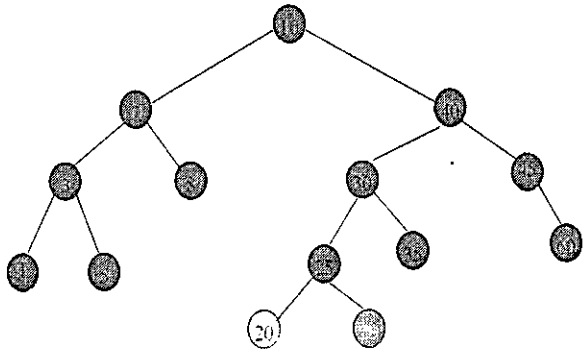
put(9)



put(29)



put(29)



Insert/Put

- Following insert/put, retrace path towards root and adjust balance factors as needed.
- Stop when you reach a node whose balance factor becomes 0, 2, or –2, or when you reach the root.
- The new tree is not an AVL tree only if you reach a node whose balance factor is either 2 or –2.
- In this case, we say the tree has become

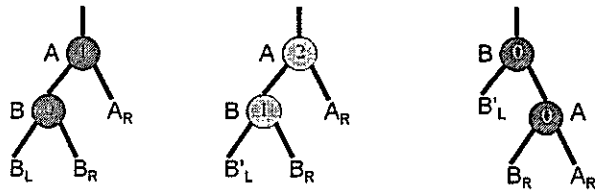
A-Node

- Let A be the nearest ancestor of the newly inserted node whose balance factor becomes +2 or –2 following the insert.
- Balance factor of nodes between new node and A is 0 before insertion.

Imbalance Types

- RR ... newly inserted node is in the right subtree of the right subtree of A.
- LL ... left subtree of left subtree of A.
- RL... left subtree of right subtree of A.
- LR... right subtree of left subtree of A.

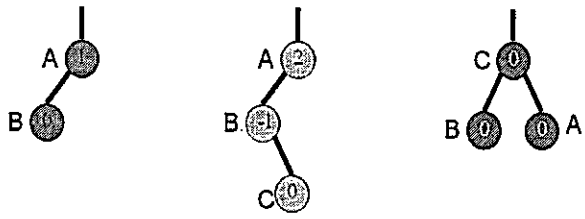
LL Rotation



Before insertion. After insertion. After rotation.

- Subtree height is unchanged.
- No further adjustments to be done.

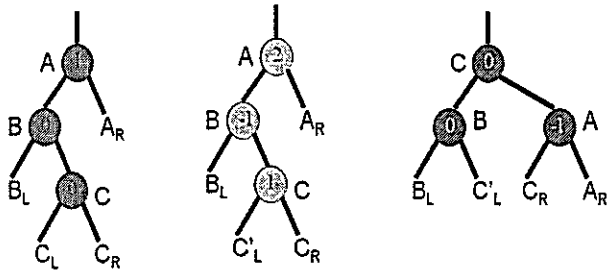
LR Rotation (case 1)



Before insertion. After insertion. After rotation.

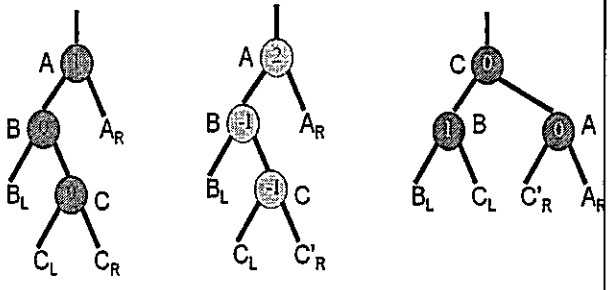
- Subtree height is unchanged.
- No further adjustments to be done.

LR Rotation (case 2)



- Subtree height is unchanged.
- No further adjustments to be done.

LR Rotation (case 3)

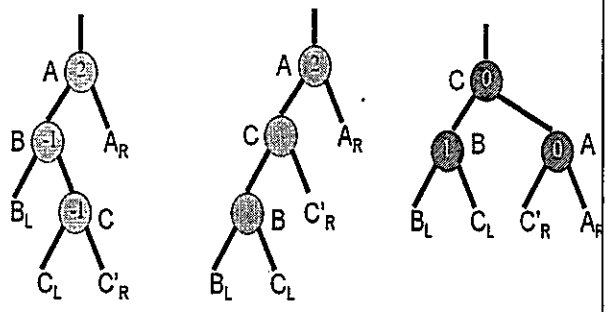


- Subtree height is unchanged.
- No further adjustments to be done.

Single & Double Rotations

- Single
  - LL and RR
- Double
  - LR and RL
  - LR is RR followed by LL
  - RL is LL followed by RR

LR Is RR + LL



After insertion. After RR rotation. After LL rotation.

Remove An Element

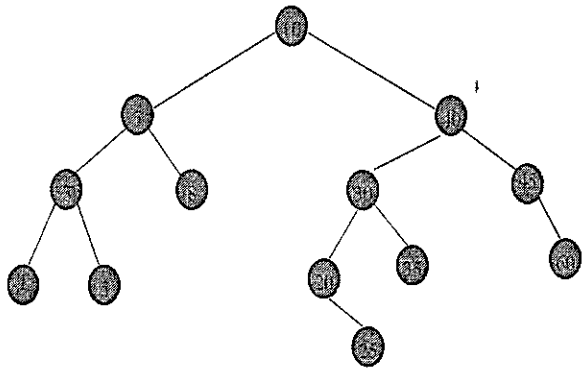
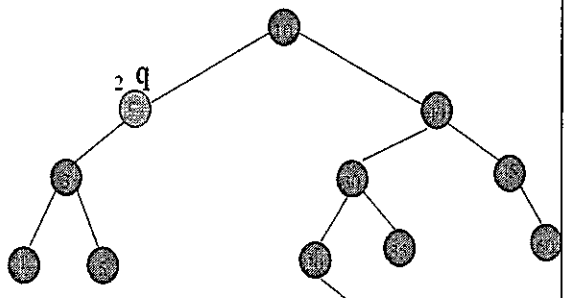


Figure 8.

Remove An Element



- q be parent of deleted node.
- Retrace path from q towards root.

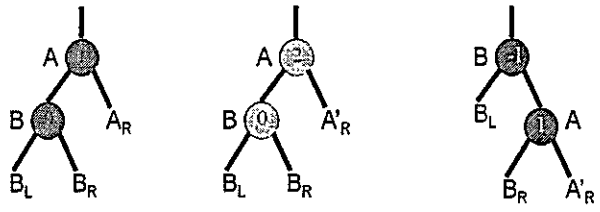
New Balance Factor Of q

- Deletion from left subtree of q bf--.
- Deletion from right subtree of q bf++.
- New balance factor = 1 or –1 no change in height of subtree rooted at q.
- New balance factor = 0 height of subtree rooted at q has decreased by 1.
- New balance factor = 2 or –2 tree is unbalanced at q.

Imbalance Classification

- Let A be the nearest ancestor of the deleted node whose balance factor has become 2 or –2 following a deletion.
- Deletion from left subtree of A type L.
- Deletion from right subtree of A type R.
- Type R new bf(A) = 2.
- So, old bf(A) = 1.
- So, A has a left child B.
  - bf(B) = 0 R0.
  - bf(B) = 1 R1.
  - bf(B) = –1 R-1.

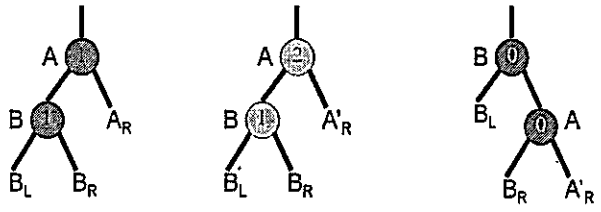
R0 Rotation



Before deletion. After deletion. After rotation.

- Subtree height is unchanged.
- No further adjustments to be done.
- Similar to LL rotation.

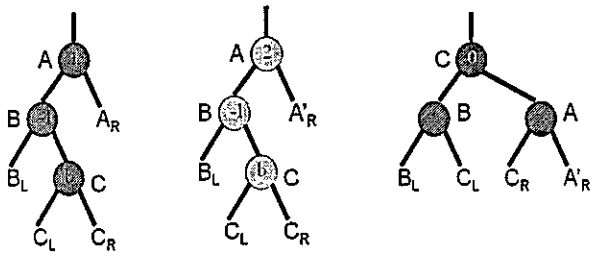
R1 Rotation



Before deletion. After deletion. After rotation.

- Subtree height is reduced by 1.
- Must continue on path to root.
- Similar to LL and R0 rotations.

R-1 Rotation



- New balance factor of A and B depends on b.
- Subtree height is reduced by 1.
- Must continue on path to root.
- Similar to LR.

Number Of Rebalancing Rotations

- At most 1 for an insert.
- O(log n) for a delete.

Red Black Trees

Colored Nodes Definition

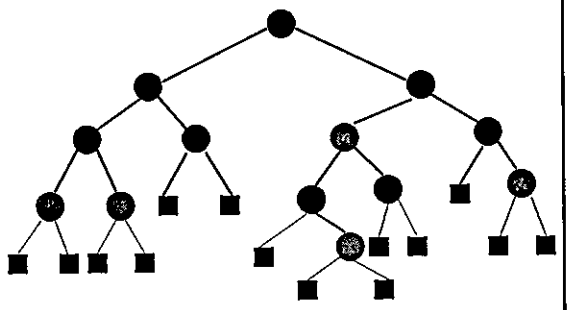
- Binary search tree.
- Each node is colored red or black.
- Root and all external nodes are black.
- No root-to-external-node path has two consecutive red nodes.
- All root-to-external-node paths have the same number of black nodes

Red Black Trees

Colored Edges Definition

- Binary search tree.
- Child pointers are colored red or black.
- Pointer to an external node is black.
- No root to external node path has two consecutive red pointers.
- Every root to external node path has the same number of black pointers.

Example Red-Black Tree



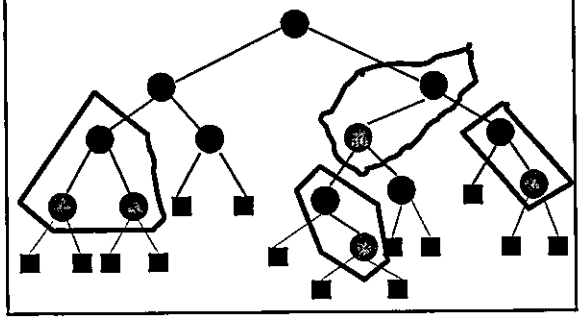
Properties

- The height of a red black tree that has n (internal) nodes is between $\log_2(n+1)$ and $2\log_2(n+1)$.

Properties

- Start with a red black tree whose height is h; collapse all red nodes into their parent black nodes to get a tree whose node-degrees are between 2 and 4, height is >= h/2, and all external nodes are at the same level.

Properties



Properties

- Let h'>= h/2 be the height of the collapsed tree.
- In worst-case, all internal nodes of collapsed tree have degree 2.
- Number of internal nodes in collapsed tree >= $2^{h'}-1$.
- So, n >= $2^{h'}-1$
- So, h <= 2 $\log_2 (n + 1)$

Properties

- At most 1 rotation and O(log n) color flips per insert/delete.
- Priority search trees.
  - Two keys per element.
  - Search tree on one key, priority queue on other.
  - Color flip doesn't disturb priority queue property.
  - Rotation disturbs priority queue property.
  - O(log n) fix time per rotation => $O(\log^2 n)$ overall time.

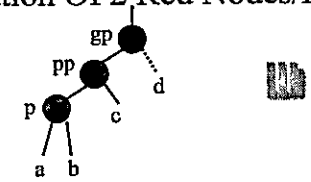
Properties

- O(1) amortized complexity to restructure following an insert/delete.
- C++ STL implementation
- java.util.TreeMap => red black tree

Insert

- New pair is placed in a new node, which is inserted into the red-black tree.
- New node color options.
  - Black node => one root-to-external-node path has an extra black node (black pointer).
    - Hard to remedy.
  - Red node => one root-to-external-node path may have two consecutive red nodes (pointers).
    - May be remedied by color flips and/or a rotation.

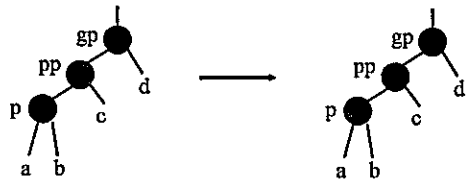
Classification Of 2 Red Nodes/Pointers



- XYz
  - X => relationship between gp and pp.
    - pp left child of gp => X = L.
  - Y => relationship between pp and p.
    - p right child of pp => Y = R.
  - z = b (black) if d = null or a black node.
  - z = r (red) if d is a red node.

XYr

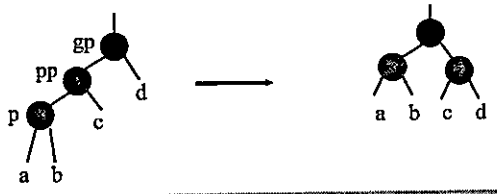
- Color flip.



- Move p, pp, and gp up two levels.
- Continue rebalancing if necessary.

LLb

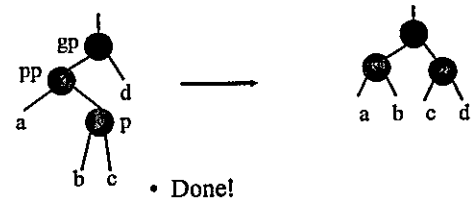
- Rotate.



- Done!
- Same as LL rotation of AVL tree.

LRb

- Rotate.

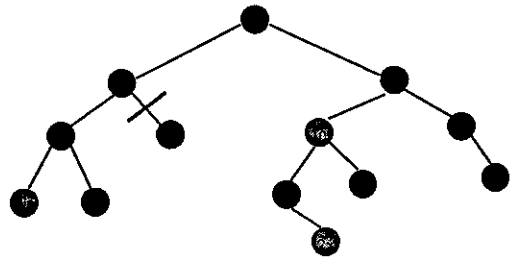


- Done!
- Same as LR rotation of AVL tree.
- RRb and RLb are symmetric.

Delete

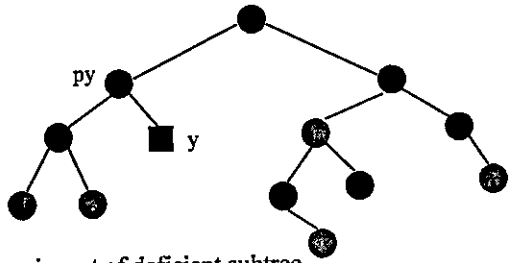
- Delete as for unbalanced binary search tree.
- If red node deleted, no rebalancing needed.
- If black node deleted, a subtree becomes one black pointer (node) deficient.

Delete A Black Leaf



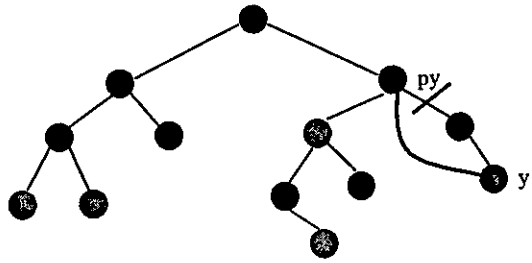
- Delete 8.

Delete A Black Leaf



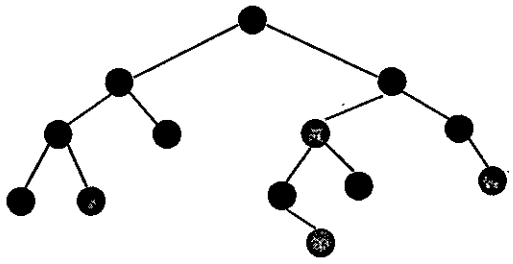
- y is root of deficient subtree.
- py is parent of y.

Delete A Black Degree 1 Node



- Delete 45.
- y is root of deficient subtree.

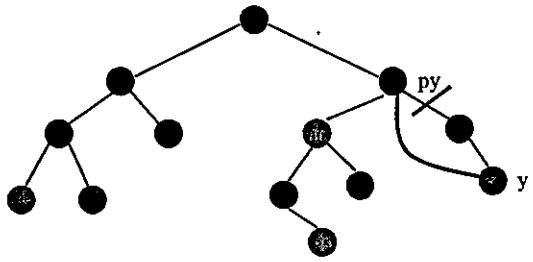
Delete A Black Degree 2 Node



- Not possible, degree 2 nodes are never deleted.

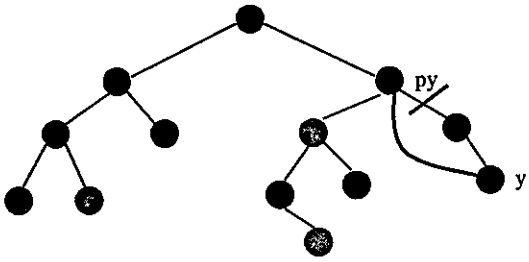
Rebalancing Strategy

- If y is a red node, make it black.



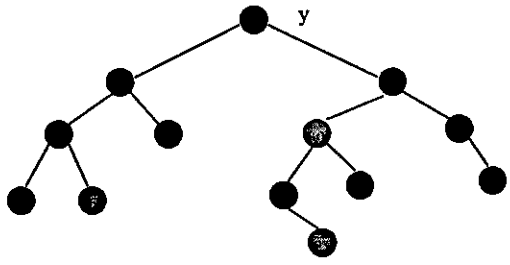
Rebalancing Strategy

- Now, no subtree is deficient. Done!



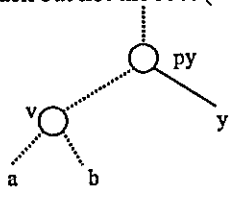
Rebalancing Strategy

- y is a black root (there is no py).
- Entire tree is deficient. Done!



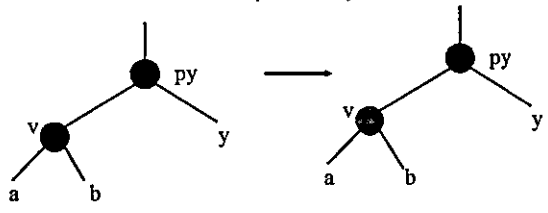
Rebalancing Strategy

- y is black but not the root (there is a py).



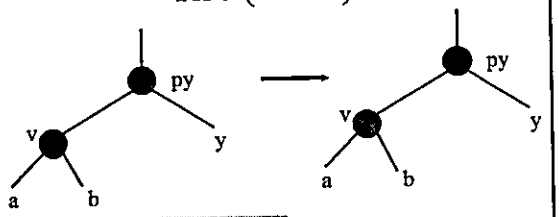
- Xcn
  - y is right child of py => X = R.
  - Pointer to v is black => c = b.
  - v has 1 red child => n = 1.

Rb0 (case 1)



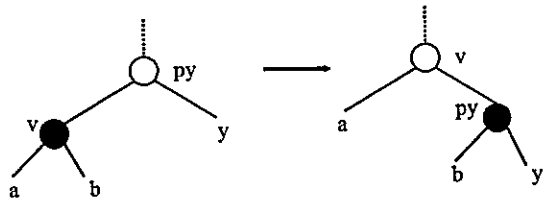
- Color change.
- Now, py is root of deficient subtree.
- Continue!

Rb0 (case 2)



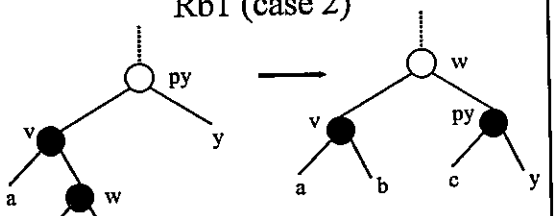
- Color change.
- Deficiency eliminated.
- Done!

Rb1 (case 1)



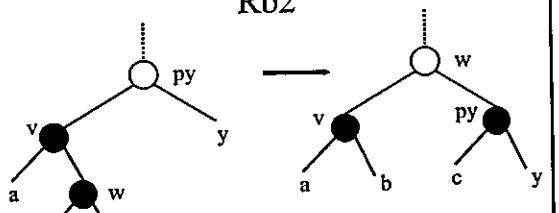
- LL rotation.
- Deficiency eliminated.
- Done!

Rb1 (case 2)



- LR rotation.
- Deficiency eliminated.
- Done!

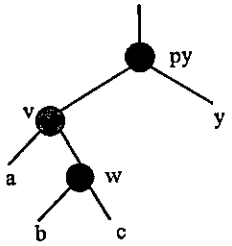
Rb2



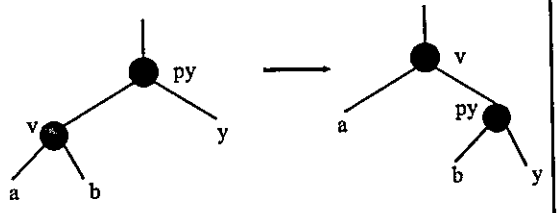
- LR rotation.
- Deficiency eliminated.
- Done!

Rr(n)

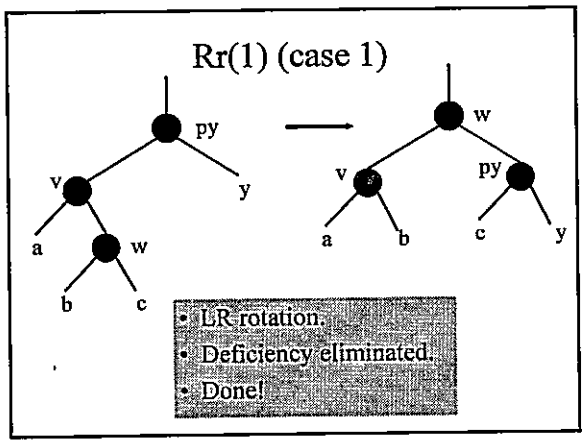
- n = # of red children of v's right child w.

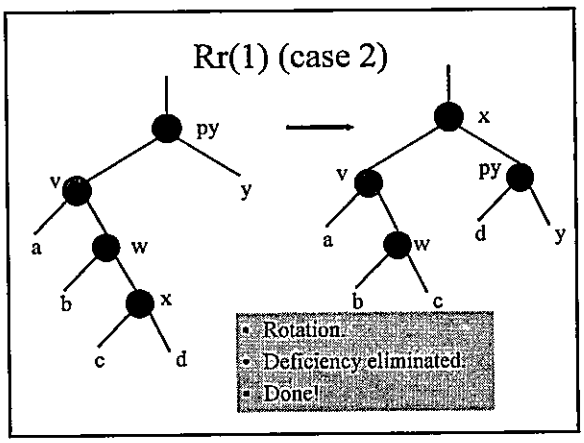


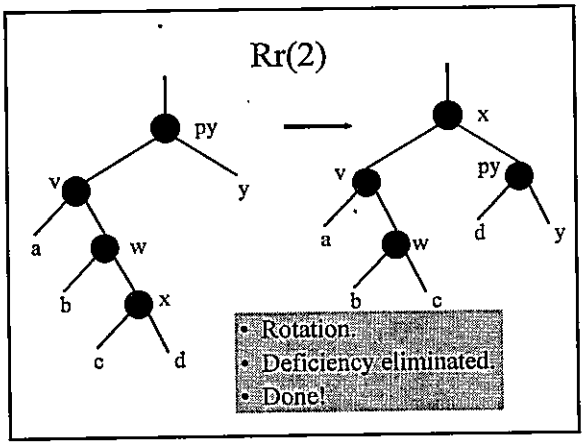
Rr(0)



- LL rotation.
- Done!


Rr(1) (case 1)
py
v
y
a
w
b
c
w
v
py
a
b
c
y
• LR rotation.
• Deficiency eliminated.
• Done!


Rr(1) (case 2)
py
v
y
a
w
b
x
c
d
x
v
py
a
w
d
y
b
c
• Rotation.
• Deficiency eliminated.
• Done!


Rr(2)
py
v
y
a
w
b
x
c
d
x
v
py
a
w
d
y
b
c
• Rotation.
• Deficiency eliminated.
• Done!

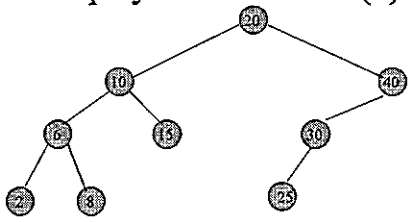
Splay Trees

- Binary search trees.
- Search, insert, delete, and split have amortized complexity O(log n) & actual complexity O(n).
- Actual and amortized complexity of join is O(1).
- Priority queue and double-ended priority queue versions outperform heaps, deaps, etc. over a sequence of operations.
- Two varieties.
  - Bottom up.
  - Top down.

Bottom-Up Splay Trees

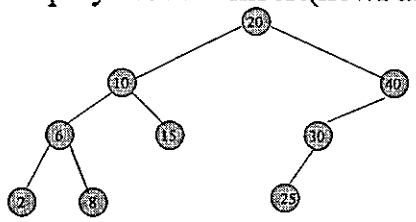
- Search, insert, delete, and join are done as in an unbalanced binary search tree.
- Search, insert, and delete are followed by a splay operation that begins at a splay node.
- When the splay operation completes, the splay node has become the tree root.
- Join requires no splay (or, a null splay is done).
- For the split operation, the splay is done in the middle (rather than end) of the operation.

Splay Node – search(k)



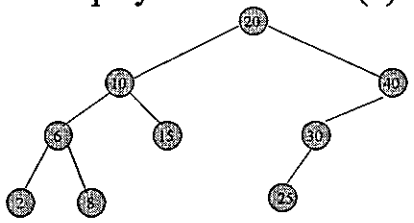
- If there is a pair whose key is k, the node containing this pair is the splay node.
- Otherwise, the parent of the external node where the search terminates is the splay node.

Splay Node – insert(newPair)



- If there is already a pair whose key is newPair.key, the node containing this pair is the splay node.
- Otherwise, the newly inserted node is the splay node.

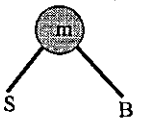
Splay Node – delete(k)



- If there is a pair whose key is k, the parent of the node that is physically deleted from the tree is the splay node.
- Otherwise, the parent of the external node where the search terminates is the splay node.

Splay Node – split(k)

- Use the unbalanced binary search tree insert algorithm to insert a new pair whose key is k.
- The splay node is as for the splay tree insert algorithm.
- Following the splay, the left subtree of the root is S, and the right subtree is B.



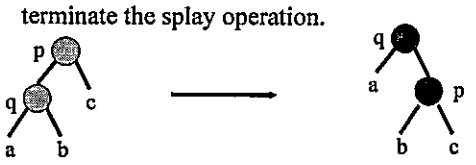
- m is set to null if it is the newly inserted pair.

Splay

- Let q be the splay node.
- q is moved up the tree using a series of splay steps.
- In a splay step, the node q moves up the tree by 0, 1, or 2 levels.
- Every splay step, except possibly the last one, moves q two levels up.

Splay Step

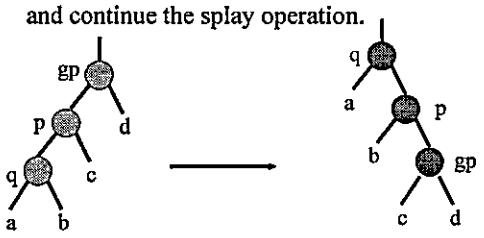
- If q = null or q is the root, do nothing (splay is over).
- If q is at level 2, do a one-level move and terminate the splay operation.



- q right child of p is symmetric.

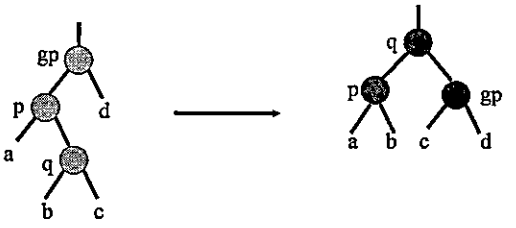
Splay Step

- If q is at a level > 2, do a two-level move and continue the splay operation.



- q right child of right child of gp is symmetric.

2-Level Move (case 2)



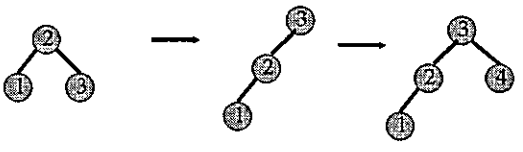
- q left child of right child of gp is symmetric.

Per Operation Actual Complexity

- Start with an empty splay tree and insert pairs with keys 1, 2, 3, ..., in this order.

Per Operation Actual Complexity

- Start with an empty splay tree and insert pairs with keys 1, 2, 3, ..., in this order.



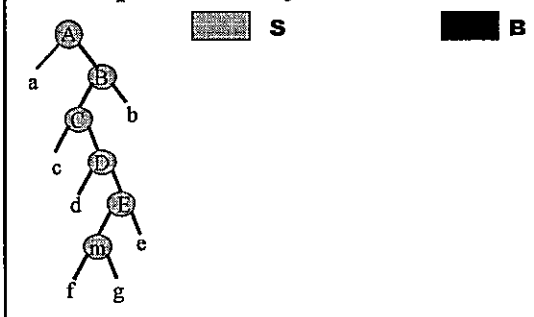
Per Operation Actual Complexity

- Worst-case height = n.
- Actual complexity of search, insert, delete, and split is O(n).

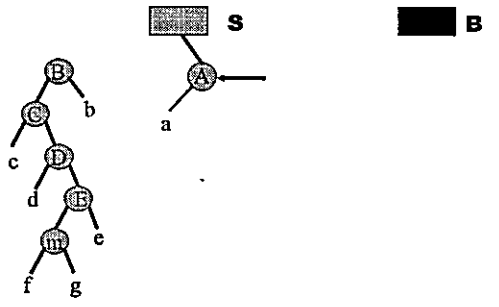
Top-Down Splay Trees

- On the way down the tree, split the tree into the binary search trees **S** (small elements) and **B** (big elements).
  - Similar to split operation in an unbalanced binary search tree.
  - However, a rotation is done whenever an LL or RR move is made.
  - Move down 2 levels at a time, except (possibly) in the end when a one level move is made.
- When the splay node is reached, **S**, **B**, and the subtree rooted at the splay node are combined into a single binary search tree.

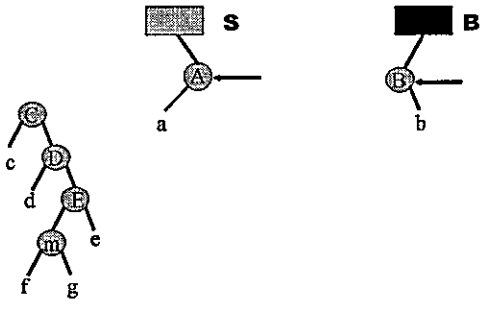
Split A Binary Search Tree



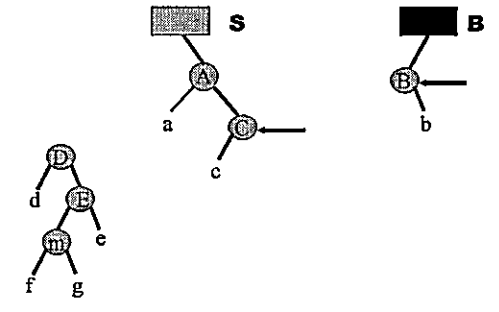
Split A Binary Search Tree



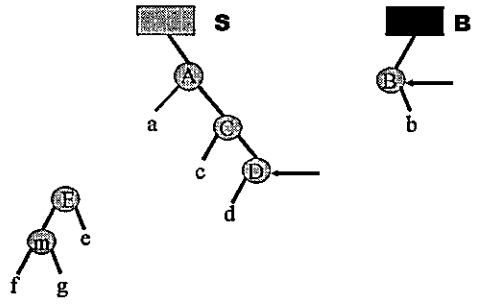
Split A Binary Search Tree



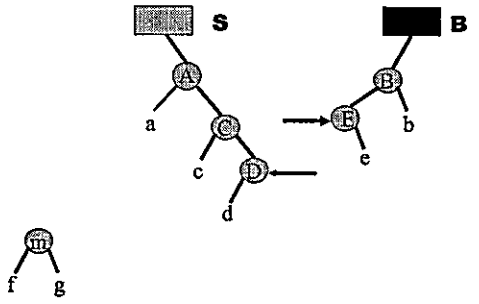
Split A Binary Search Tree



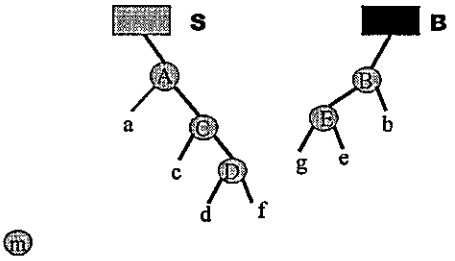
Split A Binary Search Tree



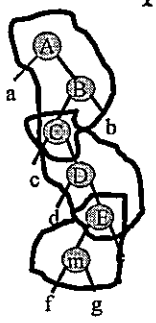
Split A Binary Search Tree



Split A Binary Search Tree

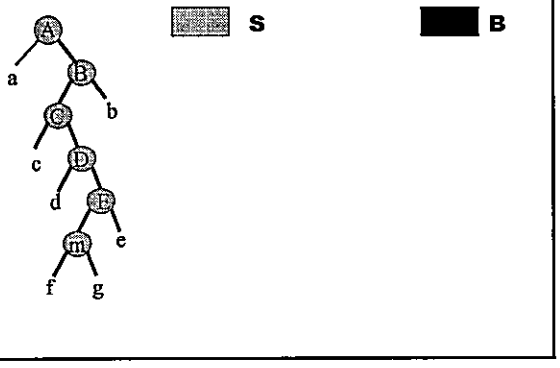


Two-Level Moves

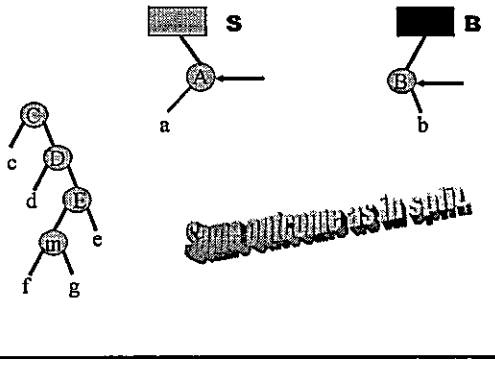


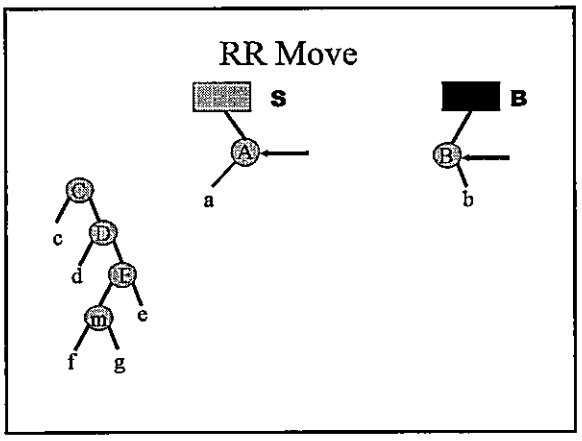
- Let m be the splay node.
- RL move from A to C.
- RR move from C to E.
- L move from E to m.

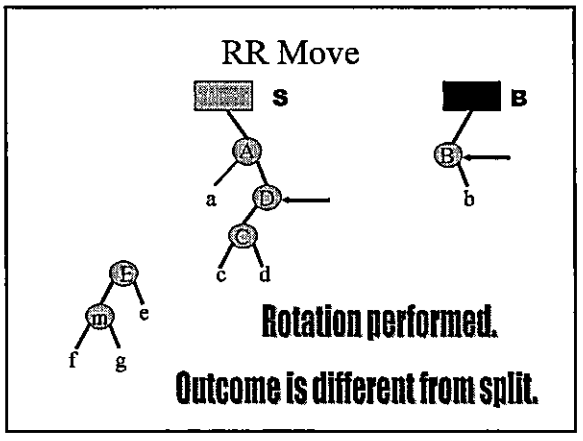
RL Move

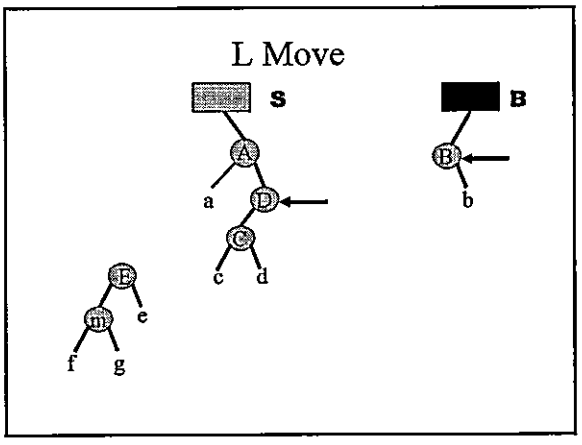


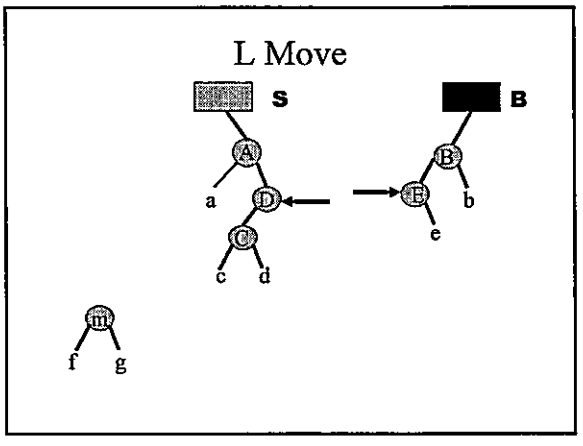
RL Move

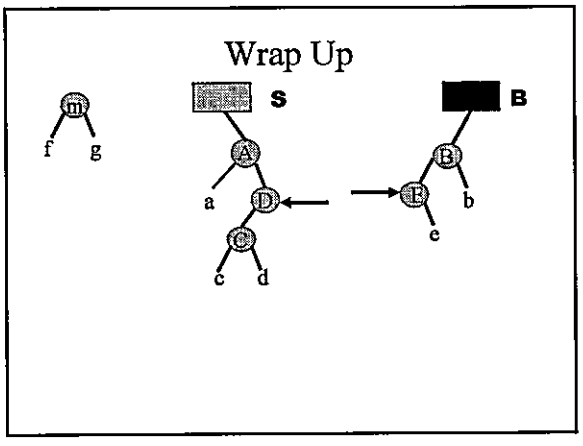


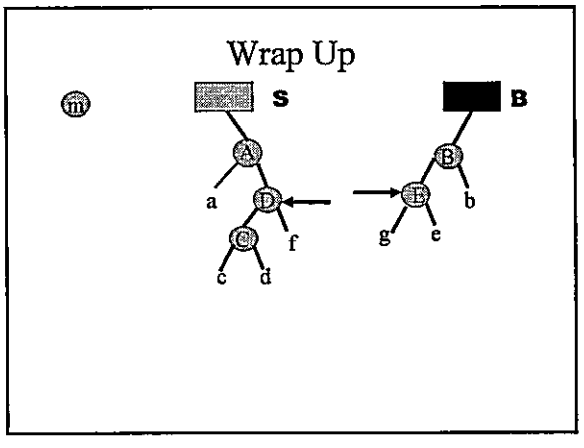

RR Move
S
B
A
B
a
b
C
c
D
d
E
e
m
f
g


RR Move
S
B
A
B
a
D
b
C
c
d
E
e
m
f
g
Rotation performed.
Outcome is different from split.

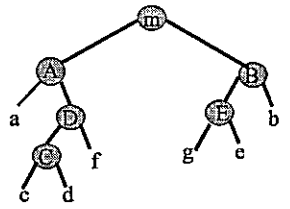

L Move
S
B
A
B
a
D
b
C
c
d
E
e
m
f
g


L Move
S
B
A
B
a
D
E
b
C
e
c
d
m
f
g


Wrap Up
m
f
g
S
B
A
B
a
D
E
b
C
e
c
d


Wrap Up
m
S
B
A
B
a
D
E
b
C
f
g
e
c
d

Wrap Up



Bottom Up vs Top Down

- Top down splay trees are faster than bottom up splay trees.

Semester End Question Papers

Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)

(Aziz Nagar, C.B.Post, Hyderabad -500075) **Subject code: A13505**

II B. Tech I SEM REGULAR EXAMINATION - NOVEMBER 2018

DATA STRUCTURES

(COMMON TO CSE &IT)

Time: 3hrs **Max.Marks:75**

Note: This question paper contains two PARTs A and B.
PART A is compulsory which carries 25 marks. Answer all questions.
PART B consists of 5questions. Answer all the questions.

PART - A

ANSWER ALL THE QUESTIONS **25 M**

1) Define linear data structure and list out types? 2M
2) Write the importance of dynamic memory allocation. 3M
3) What do you mean by complete binary tree and represent with diagram? 2M
4) What are different linear representations of binary trees? 3M
5) Define the structure of an AVL tree 2M
6) What are the different ways of traversing a BST? 3M
7) Define a pseudo graph? 2M
8) What are the appropriate data structures for Prim's and Kruskal MST? 3M
9) Define the collision? 2M
10) What are different collision techniques in hashing? 3M

PART-B

ANSWER ALL THE QUESTIONS **5QX10M=50M**

11) .i) Discuss the implementation of singly linked list with suitable example and, illustrate various operations. [10M]

OR

ii) a) Define Stack? Explain the Stack operations in detail. [5M]
b) Define string. Explain String manipulation functions with examples? [5M]

12).i) a) List the disadvantages of Binary Search Trees? Explain the insertion operation in Binary Search Tree. [7M]
b) Define Graph and write the applications of Graph. [3M]

OR

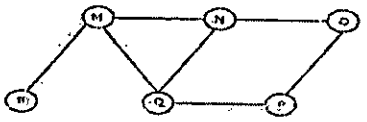
ii) a) Discuss various operations of the threaded binary tree. [6M]
b) If the tree is not a complete binary tree then what changes can be made for easy to access of children of a node in the array? Justify it. [4M]

13).i) a) Explain about the LLR, LRR imbalances in a Red-Block Tree? [6M]
b) What are the differences between B-Tree and M-way searchTree? [4M]

OR

ii) Define AVL Tree and How do we define the height of it? Explain about the balance factor associated with a node of an AVL tree. [10M]

14).i) a) Find the one possible order of the visiting nodes of the following graph using BFS algorithm has been used queue data structure. [7M]



b) Write a short note on DFS. [3M]

OR

ii)) a) Discuss in about Red-Black Trees. [6M]
b) Define the Spanning Tree. Discuss the properties of Spanning Tree. [4M]

15).i) a) Write the differences between separate chaining vs closed hashing give an example. [5M]
b) What is mean by hashing? Explain hash table and function. [5M]

OR

ii) Write a C program to implement Quadratic Probing. [10M]

VJIT(A)

Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By A.I.C.T.E., New Delhi, Permanently Affiliated to JNTU, Hyderabad)

(Aziz Nagar, C.B.Post, Hyderabad -500075)

Subject code: A13505

II B. Tech I SEM REGULAR/SUPPLY EXAMINATION- NOVEMBER 2017

DATA STRUCTURES

(COMMON TO CSE& IT)

Time: 3hrs **Max.Marks:75**

Note: This question paper contains two PARTS, PART A and B.
PART A is compulsory which carries 25 marks. Answer all questions.
PART B consists of 5 Units. Answer any one full question from each unit.

PART – A

ANSWER ALL THE QUESTIONS **25 M**

1) List various string handling functions. (2M)
2) State the rules to be followed during infix to postfix conversions. (3M)
3) Explain the properties of a binary tree. (2M)
4) Define an Expression tree. (3M)
5) Compare AVL Tree and a binary search tree. (2M)
6) State the properties of red black tree. (3M)
7) Define minimum spanning trees (2M)
8) Explain the advantage of DIJKSTRA algorithm. (3M)
9) State and explain different types of collision resolving techniques. (2M)
10) Define dictionaries and its uses. (3M)

PART-B

ANSWER ALL THE QUESTIONS **5X10M=50M**

11 i). (a) Write an algorithm for basic operations on Stack.
(b) Convert the following expression A+(B*C)-((D*E+F)/G) into post fix form.

[OR]

ii). (a) Write C programs to implement queue ADT using Arrays
(b) Write C programs to implement stack ADT using Linked list..

12 i). (a) Discuss how to represent Binary Tree and properties of a Binary Tree.
(b) Explain tree traversals with example.

[OR]

ii). (a) Define an expression tree for the following expression ((A + B) * C - (D - E) * (F + G))
(b) Write the different tree traversals for above expression.

13 i). (a) Define B-tree, B+ tree? What are the advantages and disadvantages of B-tree? Explain with an example.
(b) State the properties of Red-Black trees with example.

[OR]

ii). (a) Explain the searching and deletion operation on AVL Tress.
(b) Explain various rotations of AVL Trees maintaining balance factor while insertion takes place.

14 i). (a) Discuss about graph ADT and its uses.
(b) Define Graph and explain how graphs can be represented in adjacency matrix and adjacency LIST.

[OR]

ii). (a) Differentiate BFS and DFS.
(b) Write the advantages of using BFS over DFS or using DFS over BFS? What are the applications and downsides of each?

15 i).(a) Explain collision and any two collision resolution techniques.
(b) Explain the implementation of dictionaries using hashing.

[OR]

ii). (a) Compare the time complexities of binary search, linear search, hashing.
(b) Define hashing and discuss the different hashing functions with an example..

***VJIT (A) ***

Vidya Jyothi Institute of Technology (Autonomous)

(Accredited by NAAC & NBA, Approved By AICTE, New Delhi, Permanently Affiliated to JNTU, Hyderabad)
(Aziz Nagar, C.B.Post, Hyderabad -500075)

Subject code: A13505

II B. Tech I SEM REGULAR EXAM 2016
DATA STRUCTURES
(COMMON TO CSE and IT)

Time: 3hrs **Max.Marks:75**

Note: This question paper contains two PART A and B.
PART A is compulsory which carries 25 marks. Answer all questions in PART A.
PART B consists of 5 Units. Answer any one full question from each unit. Each question carries 10 marks and may have a, b, c as sub questions.

PART - A

ANSWER ALL THE QUESTIONS **25 M**

1. Define data structure. Differentiate static and dynamic representation of data structures 2M
2. Using strrev() function write a C program to reverse a given string? 3M
3. What are the two methods of binary tree implementation? 2M
4. What are the applications of binary tree? 3M
5. Define B-tree? 2M
6. Show the result of inserting 2, 1, 4, 5, 9, 3, 6, 7 into an initially empty AVL-tree? 3M
7. What is an adjacency matrix? What are the different ways for implementing it? 2M
8. Compare directed and undirected graph. 3M
9. Define hashing and List various hash functions 2M
10. What is extendible hashing? 3M

PART-B

ANSWER ALL THE QUESTIONS **5x10M=50M**

11 a) i) What are the advantages & disadvantages of stack? Write a program to illustrate stack operation? 6M
ii) Draw the expression tree and find the prefix and postfix expressions for the following infix expression: (C + D + A x B) x (E + F) 4M

OR

b) i) Demonstrate String Handling functions with an example program 5M
ii) Explain how the following "infix" expression is evaluated with the help of Stack :
5 * (6 + 2) - 12 / 4 5M

12 a) i) What are the different tree traversals? Explain with examples. 5M
ii) How do you insert an element in a binary tree? Explain with examples. 5M

OR

b) i) Explain Representing lists as Binary tree? Write algorithm for finding Kth element and deleting an element? 6M
ii) Explain the operations of threaded binary trees? 4M

13 a) i) Write an Algorithm to insert an item into a binary search tree and trace the algorithm with the items 6, 2, 8, 1, 4, 3, 5
ii) Explain Insertion Algorithm to Red-Black Tree and insert the following keys 5M
40, 10, 30, 35, 25, 27, 26, 60, 55, 61, 80 5M

OR

b) Explain the following routines in AVL tree with example 10M
i) insertion
ii) deletion
iii) single rotation
iv) double rotation

14 a) i) Define graph. Explain the properties of a graph. What is the difference between strongly and weakly connected graphs? 5M

ii) Write in detail about Depth First search of a graph? 5M

OR

b) Define minimum spanning tree. Write a pseudo code for Prim's algorithm. Also give an example to construct a minimum spanning tree. 10M

15 a) Given input {4371, 1323, 6173, 4199, 4299, 9669, 1989} and a hash function h(X)=X(mod10), show the resulting: 10M
(i) Separate chaining hash table.
(ii) Open addressing hash table using linear probing.
(iii) Open addressing hash table using quadratic probing.
iv) Open addressing hash table with second hash function h2(x)=7-(x mod 7)

OR

b) i) Illustrate Collision Resolution Techniques in Hashing. 5M

ii) Write short notes on Implementation of Dictionaries? 5M

VJIT(A)

Code No: 113BP **R13**

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD
B.Tech II Year I Semester Examinations, May/June-2015
DATA STRUCTURES
(Common to CSE, IT)

Time: 3 Hours **Max. Marks: 75**

Note: This question paper contains two parts A and B.
Part A is compulsory which carries 25 marks. Answer all questions in Part A.
Part B consists of 5 Units. Answer any one full question from each unit.
Each question carries 10 marks and may have a, b, c as sub questions.

PART-A **(25 Marks)**

1.a) Define Time Complexity. [2M]
b) Write brief note on Sparse Matrix. [3M]
c) Write the Stack ADT. [2M]
d) Write the steps for converting expression from infix to postfix. [3M]
e) Define Graph. [2M]
f) Explain about Threaded Binary Trees. [3M]
g) Define Sorting and list the Sorting Methods. [2M]
h) Write about Hash Functions. [3M]
i) Write the properties of Binary Search Trees. [2M]
j) Write about Standard Trie. [3M]

PART-B **(50 Marks)**

2.a) Explain Omega and Theta notations with examples.
b) Explain about Circular linked lists. [5+5]

OR

3.a) Explain Big O Notation with an example.
b) List and Explain about Double Linked List operations. [5+5]

4.a) Write a program to implement Circular Linked List.
b) Define queue. Discuss about the various representations of a queue. [5+5]

OR

5.a) Write a C Program to describe implementation of recursion.
b) What is ADT? Write the ADT for Queue Operations. [5+5]

6.a) Write a C Program to implement BFS.
b) Write a C program to implement Binary Tree. [5+5]

OR

7.a) Explain Adjacency matrix Graph Representation method.
b) Explain about MaxHeap operations with an example. [5+5]

Extra Topics Delivered

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Add on Topics

S. No	Name of the Topic
1	Priority Queues
2	Time Complexities of various Algorithms

Priority Queue

Introduction:

The **priority queue in the data structure** is an extension of the "normal" queue. It is an abstract data type that contains a group of items. It is like the "normal" queue except that the dequeuing elements follow a priority order.

It is an abstract data type that provides a way to maintain the dataset. The "normal" queue follows a pattern of first-in-first-out. It dequeues elements in the same order followed at the time of insertion operation. However, the element order in a priority queue depends on the element's priority in that queue. The priority queue moves the highest priority elements at the beginning of the priority queue and the lowest priority elements at the back of the priority queue.

It supports only those elements that are comparable. Hence, a **priority queue in the data structure** arranges the elements in either ascending or descending order.

We can think of a priority queue as several patients waiting in line at a hospital. Here, the situation of the patient defines the priority order. The patient with the most severe injury would be the first in the queue.

Characteristics:

A queue is termed as a priority queue if it has the following characteristics:

- Each item has some priority associated with it.
- An item with the highest priority is moved at the front and deleted first.
- If two elements share the same priority value, then the priority queue follows the first-in-first-out principle for de queue operation.

Types of Priority Queue:

A priority queue is of two types:

- Ascending Order Priority Queue
- Descending Order Priority Queue

Ascending Order Priority Queue

An ascending order priority queue gives the highest priority to the lower number in that queue. For example, you have six numbers in the priority queue that are 4, 8, 12, 45, 35, 20. Firstly, you will arrange these numbers in ascending order. The new list is as follows: 4, 8, 12, 20. 35, 45. In this list, 4 is the smallest number. Hence, the ascending order priority queue treats number 4 as the highest priority.

4	8	12	20	35	45

In the above table, 4 has the highest priority, and 45 has the lowest priority.

Descending Order Priority Queue

A descending order priority queue gives the highest priority to the highest number in that queue. For example, you have six numbers in the priority queue that are 4, 8, 12, 45, 35, 20. Firstly, you will arrange these numbers in ascending order. The new list is as follows: 45, 35, 20, 12, 8, 4. In this list, 45 is the highest number. Hence, the descending order priority queue treats number 45 as the highest priority.

45	35	20	12	8	4

In the above table, 4 has the lowest priority, and 45 has the highest priority.

Searching Techniques & Time Complexities

- Searching is the process of finding a given value position in a list of values.
- It decides whether a search key is present in the data or not.
- It is the algorithmic process of finding a particular item in a collection of items.
- It can be done on internal data structure or on external data structure.

Searching Techniques

To search an element in a given array, it can be done in following ways:

1. Sequential Search
2. Binary Search

1. Sequential Search

- Sequential search is also called as Linear Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

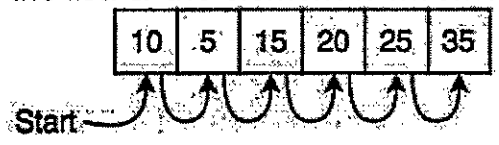


Fig. Sequential Search

The above figure shows how sequential search works. It searches an element or value from an array till the desired element or value is not found. If we search the element 25, it will go step by step in a sequence order. It searches in a sequence order. Sequential search is applied on the unsorted or unordered list when there are fewer elements in a list.

Complexities in linear search are given below:

Space Complexity:

Since linear search uses no extra space, its space complexity is O(n), where n is the number of elements in an array.

Time Complexity:

- Best-case complexity = O(1) occurs when the searched item is present at the first element in the search array.
- Worst-case complexity = O(n) occurs when the required element is at the tail of the array or not present at all.
- Average- case complexity = average case occurs when the item to be searched is in somewhere middle of the Array.

2. Binary Search

- Binary Search is used for searching an element in a sorted array.
- It is a fast search algorithm with run-time complexity of O(log n).
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.

5	10	15	20	25	30

- The above array is sorted in ascending order. As we know binary search is applied on sorted lists only for fast searching.
  For example, if searching an element 25 in the 7-element array, following figure shows how binary search works:

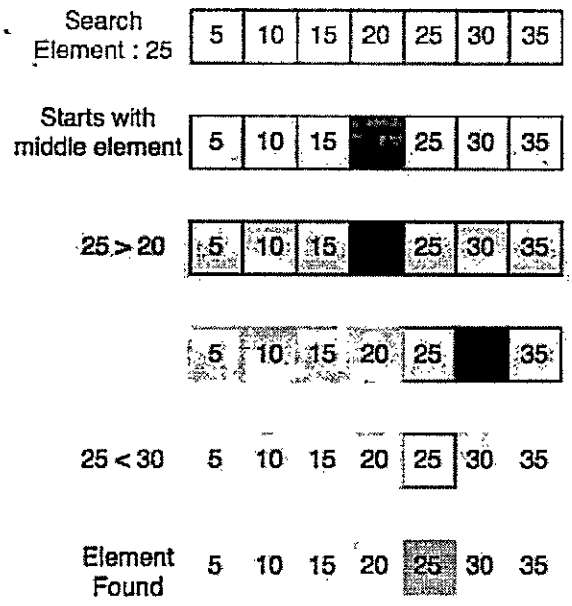


Fig. Working Structure of Binary Search

Binary searching starts with middle element. If the element is equal to the element that we are searching then return true. If the element is less than then move to the right of the list or if the element is greater than then move to the left of the list. Repeat this, till you find an element.

Binary search needs sorted order of items of the array. It works faster than a linear search algorithm. The binary search uses the divide and conquers principle.

Run-time complexity = O(log n)

Complexities in binary search are given below:

- The worst-case complexity in binary search is O(n log n).
- The average case complexity in binary search is O(n log n)
- Best case complexity = O (1)

Innovation in Teaching and Learning

Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Innovative /Student Centric Teaching Method Form

Innovative Technique implemented: Role Play

Subject: Data Structures

Topic: Binary Search Tree

Name of the Faculty: B.Sailaja

Class/ Section: II B.Tech I-Sem CSE-A

Implementation:

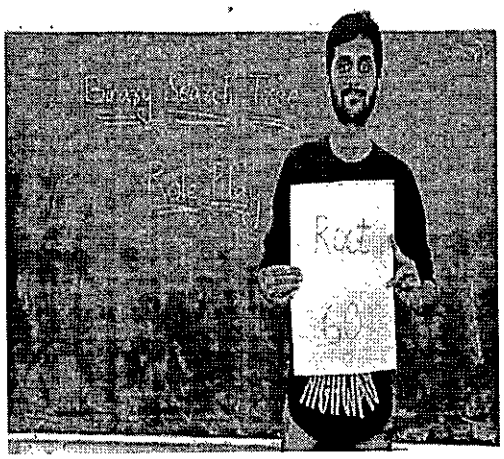
Objective: "Students gain a better understanding of the subject"

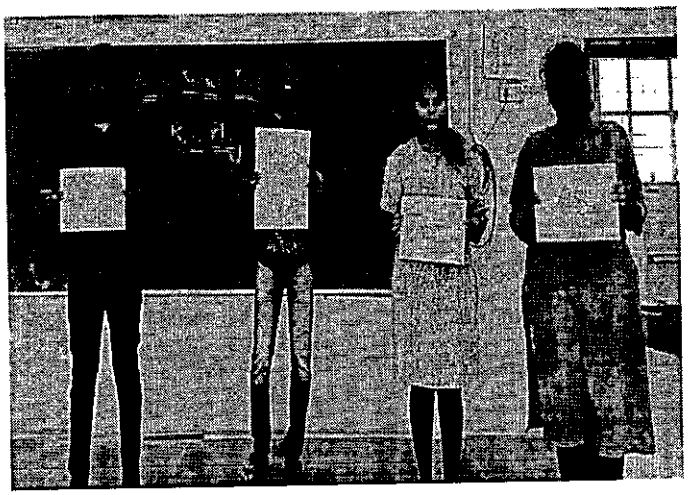
Out Class activity: Made one team to play the roles and informed the students how to present their roles

In class activity:

Each student of the team represents a
number , with a growing tree formed by adding one student at a time

Outcome: It provides real-world scenarios to help students learn





(Course Coordinator)

(HOD-CSE)

Vidya Jyothi Institute of Technology

(Accredited by NAAC & NBA , Approved By A.I.C.T.E., New Delhi, permanently affiliated to JNTUH)

(An AUTONOMOUS Institution)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Innovative /Student Centric Teaching Method Form

Innovative Technique implemented: Interactive Learning (Seminars)

Subject: Data Structures

Topic: AVL Tree

Name of the Faculty: B.Sailaja

Class/ Section: II B.Tech I-Sem CSE-A

Implementation:

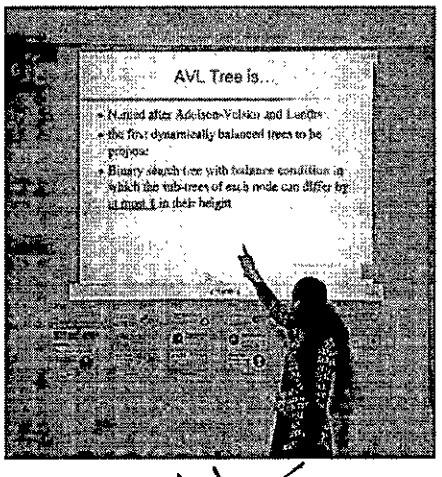
Objective: "Students gain a better understanding of the subject"

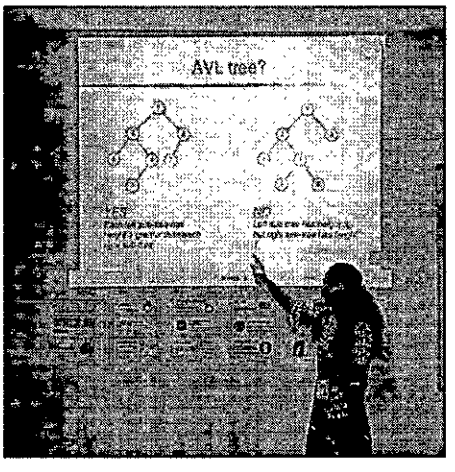
Out Class activity: Made groups and Topic Sent via Channel/Source : "WhatsApp"

Given detailed instructions to students how to give the seminar.

In-class Activity: Seminars are presented by groups on a specific topic allocated to them by presenting ppt

Outcome: Ensure positive learning outcomes related to communication, depth of knowledge of subjects, and the ability to synthesize, evaluate and reflect on information.





(Course Coordinator)

(HOD-CSE)

Assessment Sheet

(CO Attainment)

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

Department of Computer Science Engineering

BATCH: 2018-22

Academic Year: 2019-20

II B.Tech- I Sem

Course: DS

Faculty:

S.No	Reg.No	MID I Threshold 60%								MID II Threshold 60%								Threshold 60% (45M)
		ASM I (5)	PART-A				PART-B			ASM II (5)	PART-A				PART-B			End Exam (75M)
			Q1(2M)	Q2(2M)	Q3 A (1M)	Q3 B (1M)	Q4(5M)	Q5(5M)	Q6(4M)		Q1(2M)	Q2(2M)	Q3 A (1M)	Q3 B (1M)	Q4(4M)	Q5(5M)	Q6(5M)	
1	17911A0593	5	2	2	1	1	4	4	4	5	2	2	1	1	4	4	4	48
2	18911A0501	5	2	2	1	1	2	2	3	5	2	2	1	1	3	2	2	52
3	18911A0502	5	2	1	0	1	2	2	2	5	2	2	1	1	2	2	2	49
4	18911A0503	5	2	1	0	0	2	2	2	5	2	2	1	1	2	2	2	56
5	18911A0504	5	2	1	1	1	2	4	4	5	2	2	1	1	4	2	4	54
6	18911A0505	5	2	2	1	1	3	3	3	5	2	2	1	1	3	3	3	50
7	18911A0506	5	2	2	1	1	3	3	2	5	2	2	1	1	2	3	3	45
8	18911A0507	5	2	2	1	1	4	4	4	5	2	2	1	1	4	4	4	49
9	18911A0508	5	2	2	1	1	5	5	4	5	2	2	1	1	4	5	5	47
10	18911A0509	5	2	1	1	1	3	3	3	5	2	2	1	1	1	3	3	54
11	18911A0510	5	2	1	1	1	1	3	3	5	2	2	0	0	2	2	2	48
12	18911A0511	5	2	2	1	1	3	3	4	5	2	2	1	1	4	3	3	55
13	18911A0512	5	2	2	1	1	4	5	4	5	2	2	1	1	4	4	5	54
14	18911A0513	5	2	1	1	1	5	5	4	5	2	2	1	1	4	5	5	60
15	18911A0515	5	2	1	1	1	3	3	3	5	2	2	1	1	1	3	3	46
16	18911A0516	5	2	2	0	0	2	2	2	5	2	2	1	1	3	1	3	49
17	18911A0518	5	2	2	1	1	3	3	4	5	2	2	1	1	4	3	3	52
18	18911A0519	5	2	2	1	1	4	5	4	5	2	2	1	1	4	4	5	49
19	18911A0521	5	2	2	1	1	3	3	3	5	2	2	1	1	3	3	3	53
20	18911A0522	5	2	2	0	1	3	3	2	5	2	2	1	1	2	3	3	61
21	18911A0523	5	2	2	0	1	4	4	4	5	2	2	1	1	4	4	4	57
22	18911A0524	5	2	2	1	1	3	3	3	5	2	2	1	1	3	3	3	53
23	18911A0525	5	2	2	1	1	3	3	2	5	2	2	1	1	2	3	3	56
24	18911A0527	5	2	2	1	1	4	4	4	5	2	2	1	1	4	4	4	49
25	18911A0528	5	2	2	1	1	5	5	4	5	2	2	1	1	4	5	5	52
26	18911A0529	5	2	2	1	1	3	3	3	5	2	2	1	1	1	3	3	56
27	18911A0530	5	2	2	1	1	1	3	3	5	2	2	1	1	3	1	3	59
28	18911A0531	5	2	2	1	1	3	3	4	5	2	2	0	0	2	2	2	48
29	18911A0532	5	2	2	1	1	4	5	4	5	2	2	1	1	4	4	5	53
30	18911A0533	5	2	2	1	1	5	5	4	5	2	2	1	1	4	5	5	59
31	18911A0534	5	2	2	1	1	3	3	3	5	2	2	1	1	1	3	3	51
32	18911A0535	5	2	2	1	1	1	3	3	5	2	2	1	1	3	1	3	68
33	18911A0536	5	2	2	1	1	3	3	4	5	2	2	1	1	4	3	3	45
34	18911A0537	5	2	2	1	1	4	5	4	5	2	2	1	1	4	4	5	54
35	18911A0538	5	2	2	1	1	3	3	3	5	2	2	1	1	3	3	3	48
36	18911A0539	5	2	2	1	1	3	3	2	5	2	2	1	1	2	3	3	59
37	18911A0540	5	2	2	1	1	4	4	4	5	2	2	1	1	4	4	4	54

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

Department of Computer Science Engineering

38	18911A0541	5	2	2	1	1	3	3	3	5	2	2	1	1	3	3	3	50
39	18911A0542	5	2	2	1	1	3	3	2	5	2	2	1	1	2	3	3	49
40	18911A0543	5	2	2	0	0	2	2	2	5	2	2	0	0	2	2	2	55
41	18911A0544	5	2	1	0	1	3	3	3	5	2	2	1		1	3	3	37
42	18911A0546	5	2	1	0	0	2	2	2	5	1	1	1	1	2	2	2	26
43	18911A0547	5	2	1	0	1	4	4	4	5	2	2	1	1	4	4	4	47
44	18911A0548	5	2	2	1	1	2	2	3	5	2	2	1	1	3	2	2	45
45	18911A0549	5	2	2	0	1	2	2	2	5	2	2	1	1	2	2	2	40
46	18911A0550	5	2	2	0	0	2	2	2	5	2	2	1	1	2	2	2	39
47	18911A0551	5	2	2	1	0	2	4	4	5	2	2	1	1	4	2	4	48
48	18911A0552	5	2	1	1	0	3	3	3	5	2	2	1	1	3	3	3	46
49	18911A0553	5	2	1	1	1	3	3	2	5	2	2	0	1	2	3	3	43
50	18911A0554	5	2	2	1	1	4	4	4	5	2	2	0	1	4	4	4	44
51	18911A0555	5	2	2	1	1	5	5	4	5	2	2	1	1	4	5	5	55
52	18911A0556	5	2	1	1	1	3	3	3	5	2	2	1	1	1	3	3	46
53	18911A0557	5	2	1	1	1	1	3	3	5	2	2	1	1	3	1	3	41
54	18911A0559	5	2	2	0	0	2	2	2	5	2	2	0	0	2	2	2	46
55	18911A0560	5	2	2	1	1	4	5	4	5	2	2	1	1	4	4	5	32
56	18911A0561	5	2	2	1		5	5	4	5	2	2	1	1	4	5	5	38
57	18911A0562	5	2	2	1		3	3	3	5	2	2	0	1	1	3	3	46
58	18911A0563	5	2	1	1	1	1	3	3	5	2	2	0	1	3	1	3	44
59	18911A0564	5	2	1	1	1	3	3	4	5	2	2	1	1	4	3	3	26
60	18911A0565	5	2	1	1	1	4	5	4	5	2	2	1	1	4	4	5	46
61	18911A0566	5	2	2	1	1	3	3	3	5	2	2	1	1	3	3	3	30
62	18911A0567	5	2	2	1	1	3	3	2	5	2	2	1	1	2	3	3	48
63	18911A0568	5	2	2	1	1	4	4	4	5	2	2	1	1	4	4	4	30
64	18911A0569	5	2	2	1	0	3	3	3	5	2	2	1	1	3	3	3	43
65	18911A0570	5	2	1	1	0	3	3	2	5	2	2	0	0	2	2	2	36
66	18911A0571	5	2	1	1	0	4	4	4	5	2	2	1	0	4	4	4	53
67	18911A0572	5	2	2	1	0	5	5	4	5	2	2	0	0	2	2	2	50
68	18911A0573	5	2	2	1	0	3	3	3	5	2	2	1	0	1	3	3	41
69	18911A0574	5	2	1	1	0	1	3	3	5	2	2	1	1	3	1	3	29
70	18911A0575	5	2	1	1	0	3	3	4	5	1	1	1	1	4	3	3	44
71	18911A0576	5	2	1	1	0	4	5	4	5	2	2	1	1	4	4	5	47
72	18911A0577	5	2	1	1	0	5	5	4	5	2	2	1	1	4	5	5	53
73	18911A0578	5	2	2	0	0	2	2	2	5	2	2	1	1	1	3	3	44
74	18911A0579	5	2	2	1	0	1	3	3	5	2	2	1	1	3	1	3	40
75	18911A0580	5	2	2	1	1	3	3	4	5	2	2	1	1	4	3	3	49
76	18911A0581	5	2	2	1	1	4	5	4	5	2	2	1	1	4	4	5	41
77	18911A0582	5	2	1	1	1	3	3	3	5	2	2	1	1	3	3	3	47
78	18911A0583	5	2	1	1	1	3	3	2	5	2	2	1	1	2	3	3	43
79	18911A0584	5	2	2	1	1	4	4	4	5	2	2	1	1	4	4	4	46
80	18911A0585	5	2	2	1	1	3	3	3	5	2	2	1	1	3	3	3	37
81	18911A0586	5	2	1	1	1	3	3	2	5	2	2	0	1	2	3	3	31
82	18911A0587	5	2	1	1	1	4	4	4	5	2	2	0	1	4	4	4	33
83	18911A0588	5	2	2	1	1	3	3	3	5	2	2	0	1	1	3	3	49

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

Department of Computer Science Engineering

84	18911A0589	5	1	2	1	1	2	2	2	5	2	2	0	1	2	2	2	39
85	18911A0590	5	2	2	1	1	4	4	4	5	2	2	0	1	4	4	4	52
86	18911A0591	5	2	2	1	1	2	2	3	5	2	2	0	1	3	2	2	36
87	18911A0592	5	2	2	0	1	2	2	2	5	2	1	0	1	2	2	2	40
88	18911A0593	5	2	2	0	0	2	2	2	5	2	0	0	1	2	2	2	43
89	18911A0594	5	2	2	1	1	2	4	4	5	2	1	0	1	4	2	4	45
90	18911A0595	5	2	2	1	1	3	3	3	5	2	1	1	1	3	3	3	42
91	18911A0596	5	2	2	1	1	3	3	2	5	2	0	1	1	2	3	3	50
92	18911A0597	5	2	2	0	0	2	2	2	5	2	0	0	1	4	4	4	41
93	18911A0598	5	2	2	1	1	5	5	4	5	2	0	1	1	4	5	5	42
94	18911A0599	5	2	2	1	1	3	3	3	5	2	0	1	1	1	3	3	27
95	18911A05A0	5	2	2	1	1	1	3	3	5	2	0	1	0	3	1	3	49
96	18911A05A1	5	2	2	1	1	3	3	4	5	2	1	0	0	4	3	3	36
97	18911A05A2	5	2	2	1	1	4	5	4	5	2	2	1	1	4	4	5	37
98	18911A05A3	5	2	2	1	1	5	5	4	5	2	2	0	1	4	5	5	36
99	18911A05A4	5	2	2	1	1	3	3	3	5	2	2	0	1	1	3	3	30
100	18911A05A5	5	2	2	1	1	1	3	3	5	2	2	0	1	3	1	3	44
101	18911A05A6	5	2	2	1	1	3	3	4	5	2	2	0	1	4	3	3	49
102	18911A05A7	5	2	2	1	1	4	5	4	5	2	1	1	1	4	4	5	42
103	18911A05A8	5	2	2	1	1	3	3	3	5	2	0	1	1	3	3	3	41
104	18911A05A9	5	2	2	1	1	3	3	2	5	2	1	1	1	2	3	3	40
105	18911A05B0	5	2	2	1	1	4	4	4	5	2	1	1	1	4	4	4	48
106	18911A05B1	5	2	2	1	1	3	3	3	5	2	0	1	1	3	3	3	40
107	18911A05B2	5	2	2	0	0	2	2	2	5	2	0	1	1	2	3	3	31
108	18911A05B3	5	2	2	1	1	4	4	4	5	2	0	1	1	4	4	4	45
109	18911A05B4	5	2	2	1	1	5	5	4	5	2	0	1	1	4	5	5	26
110	18911A05B6	5	2	2	1	1	3	3	3	5	2	0	1	1	1	3	3	53
111	18911A05B7	5	2	2	1	1	1	3	3	5	2	1	1	1	3	1	3	59
112	18911A05B8	5	2	2	1	1	3	3	4	5	2	2	1	1	4	3	3	30
113	18911A05B9	5	2	2	1	1	4	5	4	5	1	2	1	1	4	4	5	56
114	18911A05C1	5	2	2	1	1	5	5	4	5	2	2	1	1	4	5	5	51
115	18911A05C2	5	2	2	1	1	3	3	3	5	1	1	1	1	1	3	3	46
116	18911A05C3	5	2	1	1	1	1	3	3	5	2	2	1	1	3	1	3	51
117	18911A05C4	5	2	1	1	1	3	3	4	5	2	2	1	1	4	3	3	37
118	18911A05C5	5	2	1	1	1	4	5	4	5	1	2	1	1	4	4	5	36
119	18911A05C6	5	2	2	1	1	3	3	3	5	0	2	0	0	2	2	2	39
120	18911A05C7	5	2	2	1	1	3	3	2	5	0	2	1	1	2	3	3	46
121	18911A05C8	5	1	2	1	1	4	4	4	5	1	2	1	1	4	4	4	32
122	18911A05C9	5	0	2	1	1	3	3	3	5	1	2	1	1	3	3	3	49
123	18911A05D0	5	0	1	1	1	3	3	2	5	1	2	1	1	2	3	3	47
124	18911A05D1	5	1	1	1	1	4	4	4	5	0	2	1	1	4	4	4	45
125	18911A05D2	5	1	2	1	1	3	3	3	5	1	2	1	1	1	3	3	38
126	18911A05D4	5	1	2	1	1	2	2	2	5	1	2	1	1	2	2	2	58
127	18911A05D5	5	1	1	1	1	4	4	4	5	0	2	1	1	4	4	4	40
128	18911A05D7	5	1	1	1	1	2	2	3	5	0	2	1	1	3	2	2	47
129	18911A05D8	5	1	1	0	1	2	2	2	5	1	1	1	1	2	2	2	56

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

Department of Computer Science Engineering

130	18911A05D9	5	1	1	0	0	2	2	2	5	2	0	1	1	2	2	2	54
131	18911A05E0	5	1	1	1	1	2	4	4	5	1	1	1	1	4	2	4	31
132	18911A05E1	5	1	2	1	1	3	3	3	5	0	1	1	1	3	3	3	55
133	18911A05E2	5	1	2	1	1	3	3	2	5	0	0	1	0	2	3	3	52
134	18911A05E3	5	1	2	1	1	4	4	4	5	1	0	1	0	4	4	4	50
135	18911A05E4	5	1	2	1	0	5	5	4	5	1	0	1	1	4	5	5	60
136	18911A05E5	5	1	1	1	1	3	3	3	5	1	0	1	1	1	3	3	36
137	18911A05E6	5	1	1	1	1	1	3	3	5	0	0	1	1	3	1	3	60
138	18911A05E7	5	1	2	1	1	3	3	4	5	1	1	1	1	4	3	3	51
139	18911A05E8	5	1	2	1	1	4	5	4	5	1	2	1	1	4	4	5	26
140	18911A05E9	5	1	1	1	1	5	5	4	5	0	2	1	1	4	5	5	51
141	18911A05F0	5	1	1	1	1	3	3	3	5	0	1	1	1	1	3	3	57
142	18911A05F1	5	0	1	1	0	1	3	3	5	1	0	1	1	3	1	3	55
143	18911A05F2	5	0	1	1	1	3	3	4	5	1	1	1	1	4	3	3	56
144	18911A05F3	5	0	2	1	0	4	5	4	5	2	1	1	1	4	4	5	26
145	18911A05F4	5	0	2	1	1	3	3	3	5	2	0	0	1	3	3	3	52
146	18911A05F5	5	1	2	1	1	3	3	2	5	2	0	0	1	2	3	3	39
147	18911A05F6	5	0	2	1	0	4	4	4	5	2	0	1	1	4	4	4	32
148	18911A05F7	5	0	1	1	1	3	3	3	5	2	0	1	1	3	3	3	59
149	18911A05F8	5	1	1	1	1	3	3	2	5	2	0	1	1	2	3	3	49
150	18911A05F9	5	1	2	1	1	4	4	4	5	2	1	1	1	4	4	4	60
151	18911A05G0	5	1	2	1	1	5	5	4	5	2	2	1	1	4	5	5	47
152	18911A05G1	5	1	1	1	1	3	3	3	5	2	2	1	1	1	3	3	44
153	18911A05G2	5	1	1	1	1	1	3	3	5	2	1	1	1	3	1	3	57
154	18911A05G3	5	1	2	1	1	3	3	4	5	2	0	1	1	4	3	3	55
155	18911A05G4	5	1	2	1	1	4	5	4	5	2	1	1	1	4	4	5	52
156	18911A05G5	5	0	2	1	1	5	5	4	5	2	1	1	1	4	5	5	52
157	18911A05G6	5	1	2	1	1	3	3	3	5	2	0	1	1	1	3	3	42
158	18911A05G7	5	1	2	1	1	1	3	3	5	2	0	1	1	3	1	3	39
159	18911A05G8	5	1	2	1	1	3	3	4	5	2	0	1	1	4	3	3	47
160	18911A05G9	5	1	2	1	1	4	5	4	5	2	0	1	1	4	4	5	62
161	18911A05H0	5	1	1	1	1	3	3	3	5	2	0	1	1	3	3	3	41
162	18911A05H1	5	1	1	1	1	3	3	2	5	2	1	1	1	2	3	3	51
163	18911A05H2	5	1	1	1	1	4	4	4	5	2	2	1	1	4	4	4	63
164	18911A05H3	5	1	2	1	1	3	3	3	5	2	2	1	1	3	3	3	45
165	18911A05H5	5	1	2	1	1	3	3	2	5	2	2	1	1	2	3	3	49
166	18911A05H6	5	1	2	1	1	4	4	4	5	2	2	1	1	4	4	4	52
167	18911A05H7	5	1	2	1	1	3	3	2	5	2	2	1	1	2	3	3	55
168	18911A05H8	5	1	1	1	1	4	4	4	5	2	2	1	1	4	4	4	38
169	18911A05H9	5	1	1	1	0	5	5	4	5	2	2	1	1	4	5	5	60
170	18911A05J0	5	1	2	1	0	3	3	3	5	2	2	1	1	1	3	3	57
171	18911A05J1	5	1	2	1	0	1	3	3	5	2	2	1	1	3	1	3	36
172	18911A05J2	5	1	1	1	0	3	3	4	5	2	2	1	1	4	3	3	57
173	18911A05J3	5	1	1	1	1	4	5	4	5	1	2	1	1	4	4	5	39
174	18911A05J4	5	1	2	1	0	5	5	4	5	1	2	1	1	4	5	5	27
175	18911A05J5	5	0	1	1	0	3	3	3	5	0	2	1	1	1	3	3	37

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

Department of Computer Science Engineering

176	18911A05J6	5	1	2	1	0	1	3	3	5	0	2	1	1	3	1	3	51
177	18911A05J7	5	1	2	1	0	3	3	4	5	1	2	1	1	4	3	3	40
178	18911A05J8	5	1	2	1	1	4	5	4	5	2	2	1	1	4	4	5	40
179	18911A05J9	5	1	1	1	0	3	3	3	5	2	2	1	1	3	3	3	38
180	18911A05K0	5	1	2	1	0	3	3	2	5	2	2	1	1	2	3	3	57
181	18911A05K1	5	1	2	1	0	4	4	4	5	1	2	1	1	4	4	4	41
182	18911A05K2	5	1	2	1	1	3	3	3	5	0	2	1	1	3	3	3	39
183	18911A05K4	5	1	2	1	0	3	3	2	5	0	2	1	1	2	3	3	28
184	18911A05K5	5	1	1	1	0	4	4	4	5	1	2	1	1	4	4	4	26
185	18911A05K6	5	1	1	1	1	3	3	2	5	1	2	1	1	2	3	3	46
186	18911A05K7	5	1	2	1	1	4	4	4	5	1	2	1	1	4	4	4	46
187	18911A05K8	5	1	2	1	1	5	5	4	5	0	2	1	1	4	5	5	53
188	18911A05K9	5	1	2	1	1	3	3	3	5	1	1	1	1	1	3	3	32
189	18911A05L0	5	2	1	1	1	1	3	3	5	1	2	1	1	3	1	3	33
190	18911A05L1	5	2	1	1	1	3	3	4	5	0	2	1	1	4	3	3	46
191	18911A05L2	5	2	1	1	1	4	5	4	5	0	2	1	1	4	4	5	58
192	18911A05L3	5	2	1	1	1	5	5	4	5	1	1	1	1	4	5	5	60
193	18911A05L4	5	2	2	1	1	3	3	3	5	2	0	1	1	1	3	3	34
194	18911A05L5	5	2	2	1	1	1	3	3	5	1	1	1	1	3	1	3	55
195	18911A05L7	5	2	2	1	1	3	3	4	5	0	1	1	1	4	3	3	35
196	18911A05L8	5	2	1	1	1	4	5	4	5	0	0	1	1	4	4	5	46
197	18911A05L9	5	2	1	1	1	3	3	3	5	1	0	1	1	5	3	3	45
198	18911A05M0	5	2	2	1	1	3	3	2	5	1	0	1	1	2	3	3	42
199	18911A05M1	5	2	2	1	1	4	4	4	5	1	0	1	1	4	4	4	64
200	18911A05M2	5	2	1	1	1	3	3	3	5	0	0	1	0	3	3	3	43
201	18911A05M3	5	2	1	1	1	1	3	3	5	1	1	1	1	3	1	3	16
202	18911A05M4	5	2	2	1	1	3	3	4	5	1	2	1	1	4	3	3	42
203	18911A05M5	5	2	1	1	1	4	5	4	5	0	2	1	1	4	4	5	45
204	18911A05M6	5	2	1	1	1	5	5	4	5	0	2	1	1	4	5	5	32
205	18911A05M7	5	2	2	1	1	3	3	3	5	1	2	1	0	5	3	3	57
206	18911A05M8	5	2	2	1	1	1	3	3	5	0	2	1	0	3	1	3	46
207	18911A05M9	5	2	2	1	1	3	3	4	5	0	2	1	1	4	3	3	34
208	18911A05N0	5	2	1	1	1	4	5	4	5	1	1	1	1	4	4	5	28
209	18911A05N1	5	2	2	1	1	3	3	3	5	1	0	1	0	3	3	3	44
210	18911A05N2	5	2	1	1	1	3	3	2	5	1	1	1	0	2	3	3	37
211	18911A05N3	5	2	2	1	1	4	4	4	5	0	1	1	0	4	4	4	51
212	18911A05N5	5	2	1	1	1	3	3	3	5	1	0	1	0	3	3	3	64
213	18911A05N6	5	2	1	1	1	3	3	2	5	1	0	1	0	5	3	3	52
214	18911A05N7	5	2	1	1	0	4	4	4	5	0	0	1	0	4	4	4	37
215	18911A05N8	5	2	2	1	1	3	3	3	5	0	0	1	1	3	3	3	28
216	18911A05N9	5	2	2	1	1	4	4	4	5	1	0	1	1	4	4	4	45
217	18911A05P0	5	2	2	1	0	3	3	2	5	2	1	1	1	5	3	3	54
218	18911A05P1	5	2	1	1	0	4	4	4	5	2	2	1	1	4	4	4	49
219	18911A05P2	5	2	1	1	0	5	5	4	5	2	2	1	1	5	5	5	45
220	18911A05P3	5	2	1	1	0	3	3	3	5	2	2	0	1	5	3	3	52
221	18911A05P4	5	2	2	1	0	1	3	3	5	2	1	1	1	3	1	3	52

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

Department of Computer Science Engineering

222	18911A05P5	5	2	1	1	0	3	3	4	5	2	0	1	1	4	3	3	55
223	18911A05P6	5	2	1	1	0	4	5	4	5	2	1	0	1	4	4	5	51
224	18911A05P7	5	2	1	1	1	5	5	4	5	2	1	0	1	4	5	5	45
225	18911A05P8	5	2	2	1	1	3	3	3	5	2	0	0	1	5	3	3	54
226	18911A05P9	5	2	2	1	0	1	3	3	5	2	0	0	1	3	1	3	13
227	18911A05Q0	5	2	2	1	0	3	3	4	5	2	0	0	1	4	3	3	52
229	19915A0501	5	2	1	1	1	4	5	4	5	2	0	1	1	4	4	5	48
230	19915A0502	5	2	1	1	1	3	3	3	5	2	0	1	0	3	3	3	61
231	19915A0503	5	2	1	1	1	3	3	2	5	2	1	1	0	2	3	3	57
232	19915A0504	5	2	2	1	1	4	4	4	5	2	2	1	0	4	4	4	45
233	19915A0505	5	2	1	1	1	3	3	3	5	2	2	1	0	3	3	3	58
234	19915A0506	5	2	1	1	1	3	3	2	5	2	2	1	0	5	3	3	51
235	19915A0507	5	2	1	1	0	4	4	4	5	1	1	1	0	4	4	4	54
236	19915A0508	5	2	2	1	1	3	3	3	5	2	0	1	1	3	3	3	12
237	19915A0509	5	2	2	1	1	4	4	4	5	2	1	1	1	4	4	4	14
238	19915A0510	5	2	2	1	0	3	3	2	5	2	1	1	1	5	3	3	14
239	19915A0511	5	2	1	1	0	4	4	4	5	2	0	1	1	4	4	4	54
240	19915A0512	5	2	1	1	0	5	5	4	5	2	0	1	1	5	5	5	15
241	19915A0513	5	2	1	1	0	3	3	3	5	2	0	0	1	5	3	3	56
242	19915A0514	5	2	2	1	0	1	3	3	5	2	0	1	1	3	1	3	51
243	19915A0515	5	2	2	1	0	3	3	4	5	2	0	1	1	4	3	3	53
244	19915A0516	5	2	2	1	0	4	5	4	5	2	1	0	1	4	4	5	46
245	19915A0517	5	2	2	1	1	5	5	4	5	2	2	0	1	4	5	5	48
246	19915A0518	5	2	1	1	1	3	3	3	5	2	2	0	1	5	3	3	57
247	19915A0519	5	2	1	1	0	1	3	3	5	2	2	0	1	3	1	3	59
248	19915A0520	5	2	2	1	0	3	3	4	5	1	2	0	1	4	3	3	54
249	19915A0521	5	2	2	1	0	4	5	4	5	2	2	0	1	4	4	5	48
250	19915A0522	5	2	1	1	1	4	4	4	5	1	1	0	1	4	4	4	59
251	19915A0523	5	2	1	1	0	3	3	3	5	2	2	0	1	3	3	3	14
252	19915A0524	5	2	2	1	0	3	3	2	5	2	2	0	1	5	3	3	54
Average marks		5	1.7	1.6	0.9	0.7	3.1	3.5	3.3	5	1.6	1.4	0.8	0.9	3.2	3.1	3.5	45.3
No of students attemped		251	251	251	251	249	251	251	251	251	251	251	251	250	251	251	251	251
No of students scored		251	182	159	232	186	201	230	203	251	179	156	207	222	184	200	229	155
%of students scored		100	72.51	63.35	92.43	74.70	80.08	91.63	80.88	100	71.31	62.15	82.47	88.80	73.31	79.68	91.24	61.75
CO ATTAINMENT		3	3.0	2.0	3.0	3.0	3.0	3.0	3.0	3	3.0	2.0	3.0	3.0	3.0	3.0	3.0	2.0

VIDYA JYOTHI INSTITUTE OF TECHNOLOGY

Department of Computer Science Engineering

ASSESSMENT OF COs FOR THE COURSE

CO	Method	value	Avg	CO Attainment (Internal)	CO Attainment (End Exam)	Overall CO Attainment
CO 1	ASM I	3	3.0	2.90	2.00	2.23
	MID I - PART A - Q1	3.0				
	MID I - PART A - Q3 A	3.0				
	MID I - PART B - Q4	3.0				
CO 2	ASM I	3	2.8			
	MID I - PART A - Q2	2.0				
	MID I - PART A - Q3 B	3.0				
	MID I - PART B - Q5	3.0				
CO 3	ASM I	3	3.0			
	ASM II	3.0				
	MID I - PART B - Q6	3.0				
	MID II - PART B - Q4	3.0				
CO 4	ASM I	3	3.0			
	MID II - PART A - Q1	3.0				
	MID II - PART A - Q3 A	3.0				
	MID II - PART B - Q5	3.0				
CO 5	ASM I	3	2.8			
	MID II - PART A - Q2	2.0				
	MID II - PART A - Q3 B	3.0				
	MID II - PART B - Q6	3.0				

Course End Survey Form