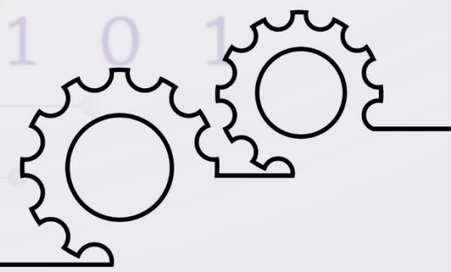
SIMATS
School of Engineering

Data Structures

Computer Science and Engineering



Saveetha Institute of Medical And Technical Sciences,Chennai.

CSA03 - DATA STRUCTURES

Concept Mapping

Topic 1 :- Introduction to Data Structures.

- Data
- Data Object
- Data Structures
- Abstract Data Types [ADT]
- Algorithm
- Performance Analysis of Algorithm
- Time Complexity
- Space Complexity
- Best, Worst, Average Case studies
- Recursion

Topic 2 :- Array

- ADT Array
- Operations on Arrays
- Searching Algorithms
  - * Linear Search
  - * Binary Search.

Topic 3 :- Linked List

- Singly Linked List
- Doubly Linked List
- Circularly Linked List

Topic 4 :- Stack

- Array Implementation
- Linked List Implementation

Topic 5 :- Stack Application

- Arithmetic Expression - Notations
  - * Infix Notation
  - * Prefix Notation
  - * Postfix Notation
- Expression Conversion
- Expression Evaluation
- Balancing Symbols

Topic 6 :- Queue.

- Array Implementation
- Linked List Implementation.
- Circular Queue.
- Applications.

Topic 7 :- Trees

- Preliminaries
- Binary Tree
- Binary Search Tree
- Tree Traversal

Topic 8 :- AVL Trees

- Single Rotations
  - * Left Left [LL]
  - * Right Right [RR]
- Double Rotation
  - * Left Right [LR]
  - * Right Left [RL]

Topic 9 :- Applications of Search Trees

- B-Tree
- TRIE
- 2-3 Tree
- 2-3-4 Tree.

Topic 10 :- Red Black Tree

Topic 11 :- Splay Trees.

- Zig Rotations
- Zag Rotations
- Zig Zig Rotations
- Zag Zag Rotations
- Zig Zag Rotations
- Zag Zig Rotations.

Topic 12 :- Hashing

- Hash Function
- Separate Chaing
- Open Addressing
- Linear Probing

Topic 13 :- Priority Queue.

- Heap
- Heap Sort

Topic 14 :- Sorting

- Insertion Sort
- Merge Sort
- Radix Sort

Topic 15 :- Quick Sort

- Divide & Conquer
- Time complexity

Topic 16 :- Sorting

- Bubble sort
- Selection sort
- Time complexity

Topic 17 :- Graph.

- Terminologies
- Types
- Topological Sort

Topic 18 :- Shortest Path Algo.

- Unweighted Shortest Path
- Dijkstra's Algo.

Topic 19 :- Minimum Spanning Tree. (MST)

- Prim's Algo.
- Kruskal's Algo.

Topic 20 :- Graph Traversal

- Breadth First Search
- Depth First Search

Topic 21 :- Advanced Data Structures

- Future Scope
- Real-Time Applications
- Research Area.

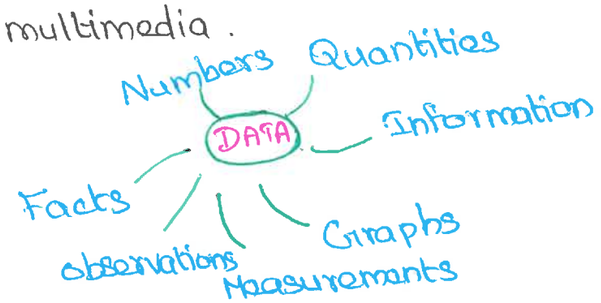
CSA03 : DATA STRUCTURES

- INTRODUCTION
  - DATA
  - DATA OBJECT
  - DATA STRUCTURE
  - ADT
  - ALGORITHM
  - PRIMITIVE AND NON PRIMITIVE
  - LINEAR AND NON LINEAR
  - PERFORMANCE ANALYSIS OF ALGO.
  - TIME COMPLEXITY
  - SPACE COMPLEXITY
  - BEST, WORST, AVERAGE CASESTUDIES
  - RECURSION
- SHORTEST PATH ALGORITHMS
  - Unweighted
  - Dijkstra
- MINIMUM SPANNING TREE
  - Prim's
  - Kruskal
- GRAPH TRAVERSAL
  - BFS
  - DFS
- GRAPH
  - Terminologies
  - Types
  - Topological sort
- DATA STRUCTURES
- HASHING
  - Hash functions
  - Separate chaining
  - Open Addressing
  - Linear Probing
- ADVANCED DS RESEARCH
  - FUTURE SCOPE
  - REAL TIME APPLICATIONS
- LINEAR DATA STRUCTURES
  - ADT ARRAYS
  - Operations on ARRAYS
  - Searching Algo.
    - Linear Search
    - Binary Search.
- LINKED LIST
  - Singly linked list
  - Doubly linked list
  - Circular linked list.
- SORTING
  - Bubble sort
  - Selection sort
  - Time Complexity
  - Priority Queue
  - Heaps
  - Heap Sort
  - Insertion sort
  - Merge sort
  - Radix sort
  - Quick sort
- STACK
  - Array Implemention
  - Pointer Implementation
- APPLICATIONS
  - Arithmetic Expression
  - Expression Conversion
  - Expression Evaluation
  - Symbol Balancing
- QUEUE
  - Array Implementation
  - Pointer Implementation
  - Circular Queue
  - Priority Queue.
  - Applications.
- TREES
  - Preliminaries
  - Binary Tree
  - Binary Search Tree.
  - Tree Traversal.
- AVL TREES ROTATION
  - LL Rotation
  - RR Rotation
  - LR Rotation
  - RL Rotation
- APPLICATION OF SEARCH TREE
  - B -Tree
  - Trie
  - 2-3 Tree
  - 2-3-4 tree
- RED BLACK TREE
- SALARY TREE
- ROTATIONS
  - ZIG
  - ZAG
  - ZIG ZAG
  - ZAG ZIG
  - ZAG ZAG
  - ZIG ZIG

TOPIC 1 : Introduction - Data Structures

DATA:

Data refers to raw facts, figures or information that is typically in the form of numbers, text, or multimedia.



DATA OBJECT:

A data object is a term used in computer Science & programming to refer to an instance of a data structure, which is a way of organizing & storing data in a computer so that it can be used efficiently.

ADT : ABSTRACT DATA TYPE

→type for objects with well-defined interface
→ Set of values & operations
→ to hide how the operation is performed on the data

DATA STRUCTURE:

A data structure is a way of organizing & storing data in a computer so that it can be used efficiently. It defines the relationship between the data & the operations that can be performed on the data

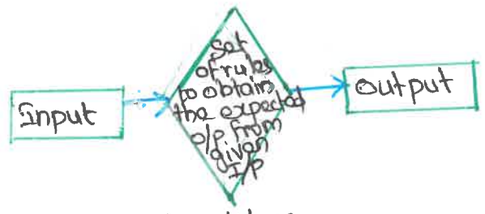
examples:

* Arrays
* Linked lists
* Stacks
* Queues
* Trees
* Graphs
* Hash tables
* Heaps

Linear & Non-linear

* Linear → Data elements are arranged sequentially or linearly
→ easy to implement
(eg) stack, array, Queue, Linked list
* Non-linear – Data elements are not arranged sequentially or linearly
→ utilizes computer memory efficiently
(eg) Trees, Graphs.

What is Algorithm:

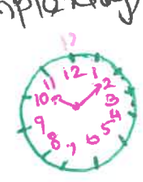


Performance Analysis of Algorithm:

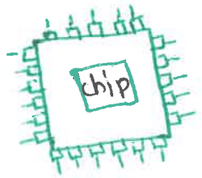
Performance analysis of Algorithms involves evaluating & comparing the efficiency of different algorithms in terms of time & space complexity

Performance analysis of an algorithm is performed by using the following measures.

* Space required to complete the task of that algorithm (space complexity) It includes program space & data space.
* Time required to complete the task of that algorithm (Time Complexity)



Time



Space

* Time complexity:

→ Time complexity measures the amount of time an algorithm takes to complete as a function of the input size.
→ Notation: Big O notation O(f(n))), where f(n) is an upper bound on the growth rate of the algorithm with respect to the input size.
→ (eg): If an algorithm has a time complexity of O(n^2), it means the running time grows quadratically with the input size.

* Space complexity:

→ Space complexity measures the amount of memory an algorithm uses relative to the input size.
→ Notation: Similar to time complexity expressed using Big O notation.
→ (eg): If an algorithm has a space complexity of O(n), it means the space required grows linearly with the input size.

* Best, worst and Average case Analysis

Algorithms may behave differently depending on the input. Analyzing best-case, worst-case & average case scenarios provides a more complete understanding of algorithmic efficiency

Primitive & Non Primitive:

Primitive & Non primitive data structures are two broad categories that classify types of data structures based on their characteristics & how they store & organize data

(eg): Integer, Float, character, Boolean.

Non-primitive data structures

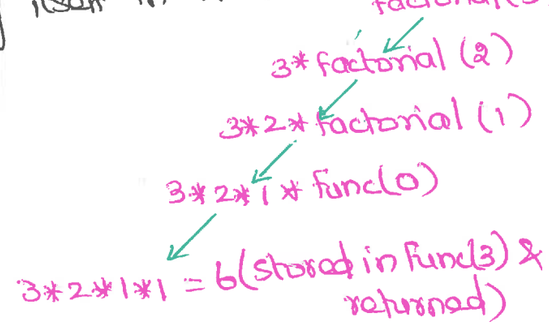
→ are more complex data structures that are constructed using primitive data types.
(eg): Arrays, Linked list, stacks, Queues, Trees, Graphs, Hash Tables.

Performance Analysis of Algorithms:

* Best case: Represents the most favourable conditions under which an algorithm performs optimally.
* Worst case: Represents the conditions under which an algorithm performs the most poorly.
* Average case: Represents the expected performance of an algorithm when considering all possible inputs.

Recursion:

Recursion is a programming concept where a function calls itself in its own definition



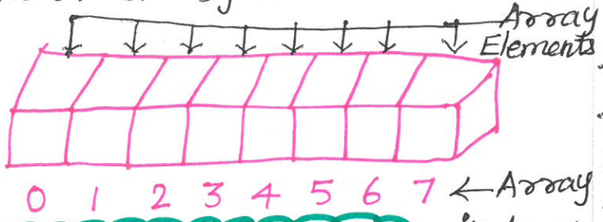
TOPIC 2 : LINEAR DATA STRUCTURES [ARRAY]

ADT Array

→ Abstract Data type (ADT)
→ Way of classifying data structures

Array

→ Array is an abstract data type.
→ Holds a collection of elements accessible by an index.



Array Representation

```
int array[ ] = {10,20,30,40,50,60,70};
```

Name, Array Elements, Type, Size

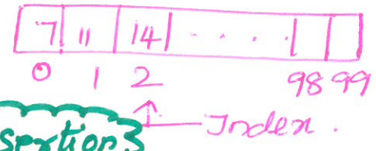
→ Index starts with 0
→ Array length is 7, it can store 7 elements.
→ Each elements can be accessed, its index.

Array Operations

- Traverse – Print all the array elements one by one
- Insertion – Adds an element at given index
- Deletion – Deletes an elements at given index
- Search – Searches an element using the given index or by the value
- Update – Updates an element at the given index

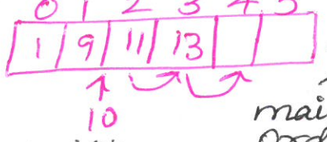
Traverse

→ Visiting every element of an array.



Insertion

→ Insert element at a specific position.
→ For this operation to succeed the array must have enough capacity.
→ insert an element 10 at index 2 then elements after index 1 must get shifted to their adjacent right to make way for a new element.

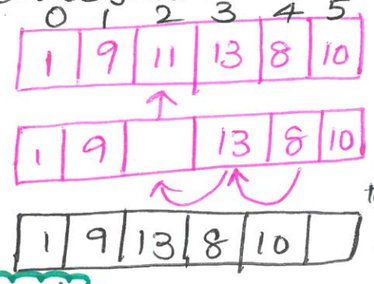


Elements need to be shifted to maintain relative Order.

→ NO position specified then insert element at end and avoid shifting.

Deletion

→ An element at a specified position can be deleted.
→ Creating void that needs to be fix-ed by shifting all elements to their adjacent left.

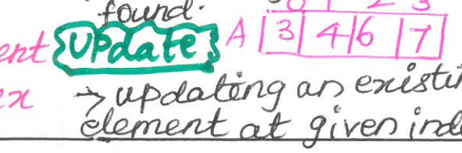


Search

→ Traversing array until the element found.

Update

→ updating an existing element at given index.



Searching Algorithms

Searching:
→ To check whether a particular elements is present in the list.

Types

→ Linear Search
→ Binary Search

Linear Search

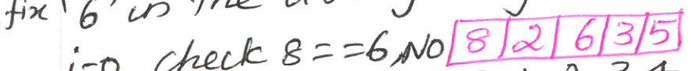
→ Sequential Search and can be applied on unsorted list.

Algorithm Steps

1. Search an element by traversing the array from the Starting, till the element is found.
2. It compares the element to the Searched with all the elements present in the array. When the element is matched, it returns the index of the element, else return '-1'.

Example

To search an element '6' in the array using LS.



i=0, Check 8==6, NO
i=1, check 2==6, NO
i=2, Check 6==6, yes
Output: 2

Code

```
for(int i=0; i<n; i++)
  { if (values[i]==Target)
      return i;
  } return -1; //Not found
```

Advantage

→ Simple, easy to understand and implement
→ Not required any order to store the data.

Disadvantage:

→ Slow & Poor efficiency.

Binary Search

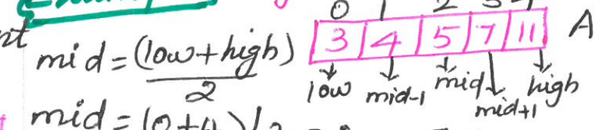
→ Searching is done for a Sorted array.
→ Uses Divide & conquer method

Algorithm Steps:

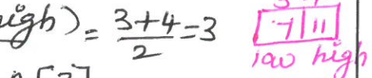
1. Find the middle element of Sorted array.
   low = 0, hight = n-1, mid = $\frac{low+high}{2}$
2. If the search key value is equal to mid element, return index of mid.
3. If not, compare the mid element with search element.
   a) if search element is greater than the mid value, find a new middle element from the right side of mid value.
   b) If search element is less than the mid value, find a new middle element from the left
4. If no matches is found, return -1.

Example

key value: 7



mid = $\frac{(low+high)}{2}$
mid = (0+4)/2 = 2 = A[2]
check 7==5 (A[2])
↳ Not same (7>5)
mid = $\frac{(low+high)}{2} = \frac{3+4}{2} = 3$



Check 7==7 A[3]
return A[3] = 3.

Advantage

→ faster compared to LS
→ No of comparison less
→ More efficient.

Disadvantage

→ Works only on sorted list and complex.

Time complexity:

	Best case	Average	Worst
Linear	O(1)	O(n)	O(n)
Binary	O(1)	O(logn)	O(logn)

Topic: 3. Linked List – Single, Doubly & Circular

Single Linked List (SLL)

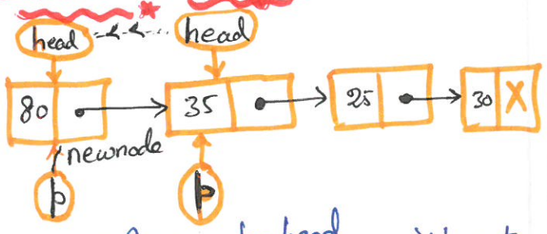
-> Collection of nodes connected from head node to tail.
-> Each node contains two fields.
   1) Data 2) Address.
-> Connection in one direction.

-> Linear data structure, in which nodes are created dynamically in memory & linked to the list.

Operations in SLL:

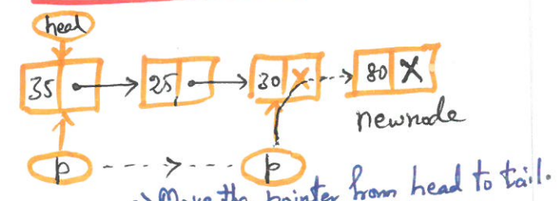
-> Insertion, -> Deletion, -> Searching
-> Sorting -> Updation & Modifications.

Insert at Begin:



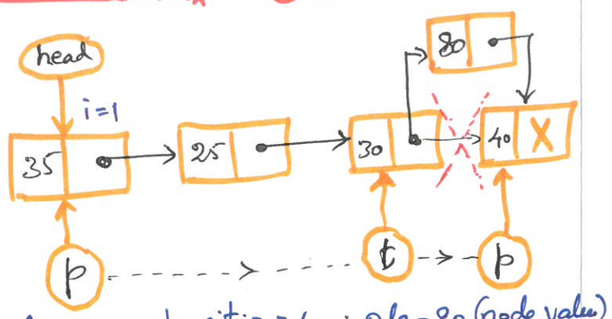
-> Assign p = head
-> newnode -> is equal to 'p' pointer.
-> head is equal to newnode.
-> newnode -> next = p.

Insert at End:



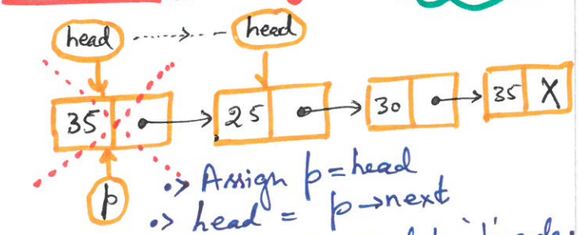
-> Move the pointer from head to tail.
-> p -> next = newnode.
-> p = newnode.

Insert at any position:



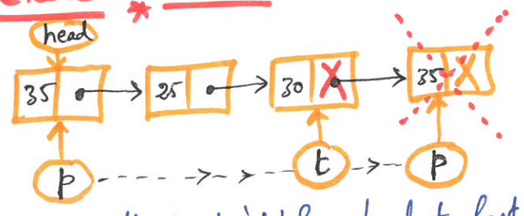
Assume: position = 4 ; ele = 80 (node value)
-> Move the pointer 'p' from head to position node.
-> Also move 't' from head to its previous position.
-> Now t -> next = newnode.
-> newnode -> next = p.

Delete at Begin:



-> Assign p = head
-> head = p -> next
-> Free (p) : Delete 'p' node.

Delete at End:

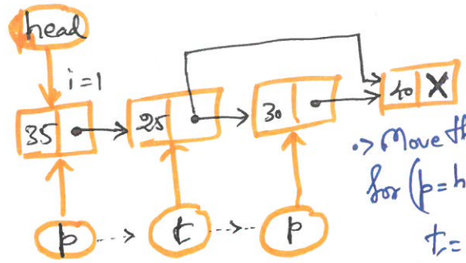


-> Move the pointer 'p' from head to last node.
-> Similarly move 't' to its previous position.
-> Now t -> next = NULL.
-> Free (p) : Delete 'p' node from the list.

APPLICATION: FACEBOOK

Delete at any position:

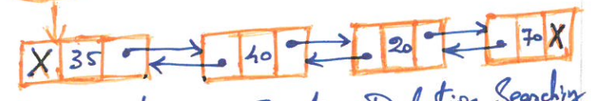
Assume: position = 3



-> Move the 'p' to its position.
for (p = head; i = 1; i < pos; t = p; p = p -> next, i++)
Now t -> next = p -> next.

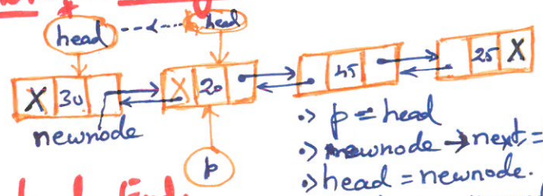
Doubly Linked List: (DLL)

-> Collection of nodes connected in two directions in which node contains two addresses.



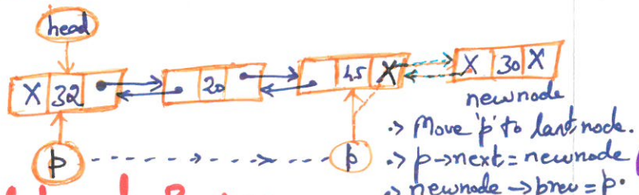
-> Operations -> Insertion, Deletion, Searching Sorting, Modifications.

Insert at Begin:



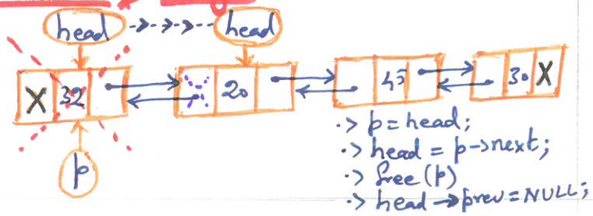
-> p = head
-> newnode -> next = p
-> head = newnode.
-> p -> prev = newnode.

Insert at End:



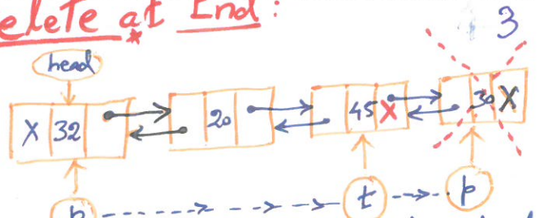
-> Move 'p' to last node.
-> p -> next = newnode
-> newnode -> prev = p.

Delete at Begin:



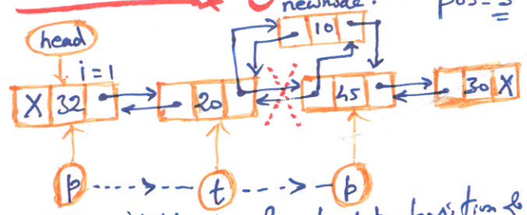
-> p = head;
-> head = p -> next;
-> Free (p)
-> head -> prev = NULL;

Delete at End:



-> Move 'p' pointer from head to end node
-> With respect to make 't' to its previous
-> t -> next = NULL.
-> Free (p) ; Delete the 'p' node.

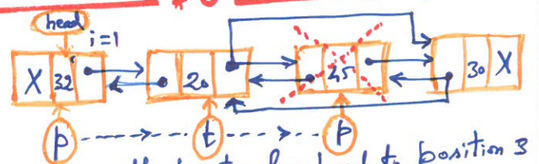
Insert at any Position:



-> Move 'p' pointer from head to position & 't' to its previous position.
-> Now t -> next = newnode;
newnode -> next = p;
newnode -> prev = t;
p -> prev = newnode.

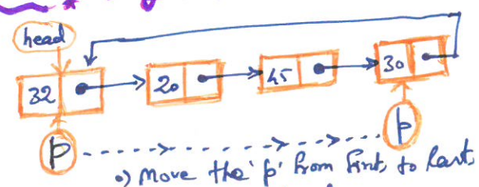
Delete at any position:

pos = 3



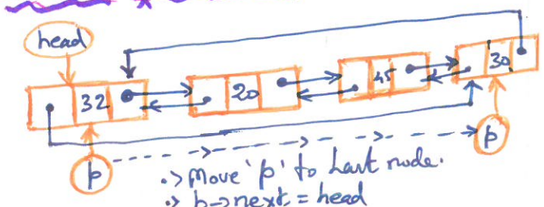
-> Move the pointer from head to position 3 and 't' to its previous.
-> Now t -> next = p -> next;
-> p -> next -> prev = t;

Circular Single Linked List:



-> Move the 'p' from first to last
-> p -> next = head.

Circular Double Linked List



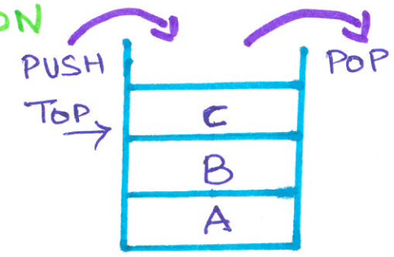
-> Move 'p' to last node.
-> p -> next = head
-> head -> prev = p.

TOPIC -4 STACK IMPLEMENTATION

Stack: An Abstract Data Type (ADT)
* A linear DS in which the operations are performed at only one end - **TOP**
* Eg: Pile of coins, stack of Trays
* **PRINCIPLE**: Last in First out **LIFO**
Lastly Inserted element will be removed first

IMPLEMENTATION: 2 Types
- ARRAYS (Fixed size DS)
- LINKED LIST - (Variable Size DS)

BASIC OPERATION
* PUSH
* POP
* PEEK
* IS FULL
* IS EMPTY



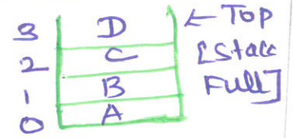
ARRAY IMPLEMENTATION

PUSH Process of inserting a new data element when stack is empty **TOP = -1**

STEPS
* check if the stack is full (overflow)
* If the stack is full, display stack Full and exit
[Top = Maxsize - 1]
* If stack is not Full increment Top Top++
* Add data element to the stack stack[Top] = Value

ISFULL - check if stack is Full

If (top == Maxsize -1)
return true;



IS Empty - check if stack is empty

If (top == -1)
return true;

POP: Process of removing top element from the stack

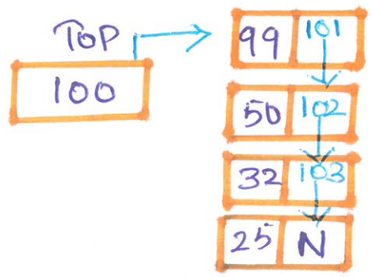
Steps
* check if stack is empty (underflow)
* If empty display stack empty (Top=-1)
* If not empty decrements
* Top value by 1 (Top=Top-1)
Return

PEEK: Returns the value of top most element

Steps
Checks if the stack is empty (underflow)
If empty displays "stack empty" (Top=-1)

If not empty, returns the element which TOP is pointing (Stack[Top])

LINKED LIST IMPLEMENTATION



PUSH: Process of inserting new element into the stack
* Create a newNode with given value
* check whether stack is Empty (Top == NULL)
* If it is Empty then set newNode -> next = NULL
* If it is Not Empty then set newNode -> next = Top
* Finally, set Top = newNode

POP: Process of Deleting an Element from a Stack
* Check whether stack is empty (Top == NULL)
* If it is Empty then "Stack is empty" Deletion is not possible
* If it is Not Empty then define a Node pointer 'temp' and set it to Top
* Then set Top = Top -> next
Finally delete 'temp' (Free(temp))

PEEK: Returns the value of Top most element
* check whether stack is empty (Top == NULL)
* If it is empty then "Stack is empty cannot Display"
* If it is Not Empty then check (Temp -> next != NULL) and display elements

TOPIC-5 (STACK APPLICATIONS)

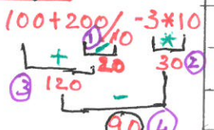
↳ Arithmetic Expression - Types of Notations:

* **Infix Notation** - arithmetic operator appears between operands (op1 operator op2) (eg:- a+b)
* **Prefix Notation** - arithmetic operator appears before operands (Polish Notation) (eg:- +ab)
* **Postfix Notation** - arithmetic operator appears after operands (Reverse Polish Notation) (eg:- ab+)

Precedence of Operators & Associativity:

Operator	Precedence	Value
Exp($,↑,^)	Highest	3
*, /	Next High	2
+, -	Lowest	1

Eg:- 100+200/10-3*10



Algorithms for Infix-Postfix Conversion

1) Scan infix expression from left to right, if it is operand, output it into postfix expression.
2) If the input is an operator and stack is empty, push operator into operator's stack.
3) If the operator's stack is not empty:
   * if the precedence of operator is greater than the top most op of stack, push it into stack.
   * if the precedence of operator is less than the top most operator of stack, pop it from the stack until we find a low precedence op.
   * if the precedence of operator is equal, then check the associativity of the operator. If LTR (Left-to-Right) associativity, then pop the operator from stack. If RTL (Right-to-Left) associativity, then push into the stack.
   * if the input is '(', push it into stack.
   * if the input is ')', pop out all operators until we find '('.
4) Repeat step 1, 2 & 3 till expression has reach '#' [#-Delimiter].
5) Now pop all the remaining operators from the stack and push into the output postfix expression.
6) Exit.

Example: A+(B*(C+D)/E#

i/p	stack	Output Postfix Exp	Action
A		A	Add A into o/p exp.
+	+	A	Push '+' into stack.
(	+(	A	Push '(' into stack
B	+(	AB	Add B into o/p exp.
*	+(*	AB	Push '*' into stack.
C	+(*	ABC	Add C into o/p exp
+	+(+	ABC*	'+' operator has less Precedence than '*' & add to o/p exp. Push '+'
D	+(+	ABC*D	Add D into o/p exp.
)	+	ABC*D+	')' has come, So pop + & add it to o/p exp
/	+/	ABC*D+	'/' has higher Precedence than '+', so push '/' to stack
E	+/	ABC*D+E	Add E into o/p exp.
#	+/	ABC*D+E/	'#'-delimiter, so pop all operators from the stack one by one and add it to o/p expression
	+	ABC*D+E/+	

Output: ABC*D+E/+

↳ Evaluating Arithmetic Expression

Algorithm Steps:

1) Scan the input string from left-right
2) For each input symbol
   a) if it is a digital number, then Push it onto the stack.
   b) if it is an operator, then pop out the top most 2 contents from the stack & apply the operator on them. Later, push the result to stack.

Example: (5+3)*(8-2) ⇒ Infix.

1) Convert into postfix expression.
   53+82-*
2) Evaluate the postfix expression.

READ Symbol	STACK OPERATIONS	EVALUATED Part of Expression
Initial	Stack-Empty	Nothing
5	Push(5)	Nothing
3	Push(3)	Nothing
+	Value 1 = PoP() 3 Value 2 = PoP() 5 result = Value2+Value1 Push(result)	Value 1 = 3 Value 2 = 5 result = 5+3 = 8 Push(8)
8	Push(8)	(5+3)
2	Push(2)	(5+3)
-	Value1 = PoP() Value2 = PoP() result = value2 - value1	Value 1 = 2 Value 2 = 8 result = 8-2 Push(6) (5+3), (8-2)
*	Value 1 = PoP() Value 2 = PoP() result = value2 * value1 Push(result)	Value 1 = 6 Value 2 = 8 Result = 8*6 Push(48) (5+3)*(8-2)
$ end	result = PoP()	Display(result) Result = 48 [(5+3)*(8-2)]

Note:- if input is '0' (end), empty the stack.

↳ Balancing Symbols

Algorithm Steps:

Read 1 character at a time until it encounters the delimiter '#'
1) If the character is an opening symbol, push it onto the stack.
2) If the character is a closing symbol, and if the stack is empty report an error as missing symbol.
3) If it is a closing symbol & if it has a corresponding opening symbol in the stack, pop it from the stack. Otherwise, "error - Mismatched Symbol".
4) At the end of file, if the stack is not empty, report as "Unbalanced Symbol". Otherwise, report as "Balanced Symbol"

Example: (a+b)#
1 2 3 4 5 6

Step	Read character	STACK	Step	Read character	STACK
1	(	(	2	a	(
3	+	(	4	b	(
5	)		6	#	empty

(a+b)# ➔ Balanced Symbol.

Example: ((a+b)#
1 2 3 4 5 6 7

Step	Read character	STACK	Step	Read character	STACK
1	(	(	2	(	((
3	a	((	4	+	((
5	b	((	6	)	(
7	#	(← Stack Not empty			

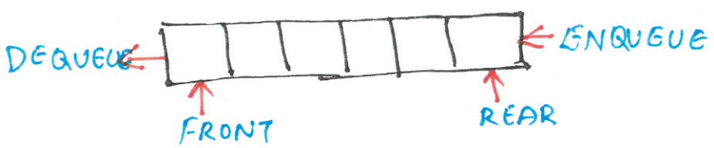
((a+b)# ➔ Unbalanced Symbol.

Other Applications:

* Frequently accessed Websites - Cookies
* Visited Machines - JVM
* Redo/Undo Operations - Google Docs
* Forward-Backward Surfing - Google
* Scratch Card - Google Pay

TOPIC-6 (QUEUE)

→ **Queue ADT**: Linear Data Structure
* Insertion : Rear End - ENQUEUE
* Deletion : Front - DEQUEUE
* First In First Out (FIFO)



→ **Real Time Example**:
Bill Counter, Bank Counter, etc.

→ **Applications**:
Printer, CPU task scheduling, etc.

→ **Implementations**:
*Array *Linked List.

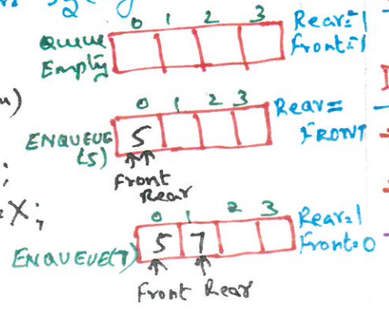
→ **Basic Operations**:
*Enqueue
*Dequeue
*Display.

→ **Implementation - Using Array**.
Fixed Size DS.

Enqueue:
* Before inserting, check whether Queue is full (Overflow Condition)
* If full, display "Queue Overflow" Rear=N FULL
* If not full, element is added to the end of the Array at rear side.

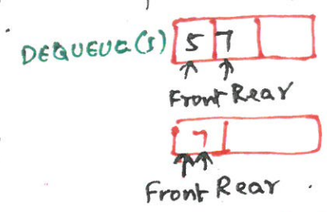
Note:- N - maximum size of Queue.

```
if (rear ≥ maxSize)
    Print ("Overflow")
{   Rear = Rear + 1;
    Queue[Rear] = X;
}
```



Dequeue:
Note:- Before deleting, check whether Queue is empty (Underflow Condition)
* If empty, display "Queue Underflow" Front=-1 (empty)
* If not empty, element in the front is deleted from the Queue and front is pointed to the next element in the Queue.

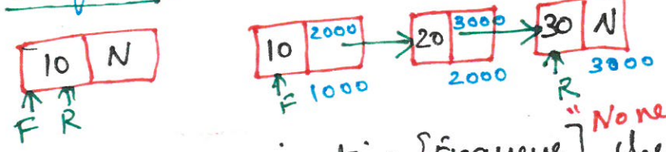
```
if (Front == -1)
    Print ("Underflow")
    else { Front = Front + 1;
}
```



→ **Implementation - Using Linked List**.
Each node contains 2 fields - Data field & Pointer field (Data) (Address of next node)

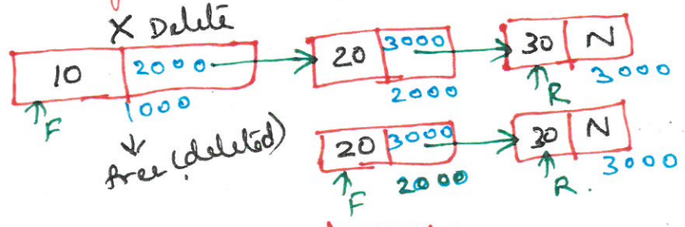
Node: Data | Address of Next

Enqueue: F → Front ; R → Rear



Note:- Before inserting [Enqueue] "No need to" check whether Queue is full (Since dynamic DS).

Dequeue:
Before deleting, check whether Queue is empty (Underflow Condition)

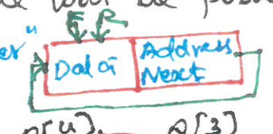


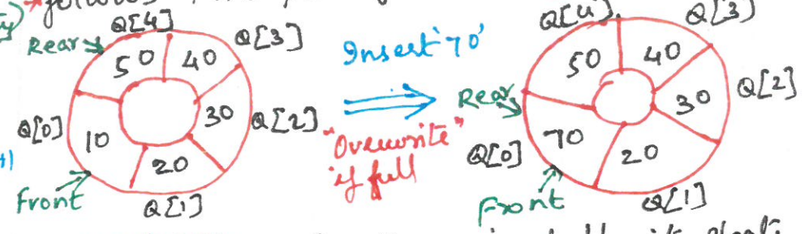
Advantages over Arrays:
Dynamic size, Easy to insert and delete.

Drawbacks
* Random access not allowed.
* Utilization of memory space is more.

→ **Types**:
*Circular Queue. *Priority Queue *DEQUE

Circular Queue: Last node will be pointing to the first node "Ring Buffer"
*follows FIFO principle.





Disadvantage:-
* When the Circular Queue is full, it starts overwriting the existing values when insert.

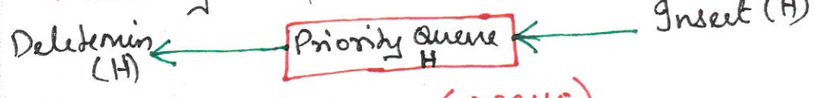
→ **Basic Operations**:- Enqueue & Dequeue.

Priority Queue: Elements will be assigned by a 'priority' & will be processed based on order.
*higher order priority element is processed before lower priority element.

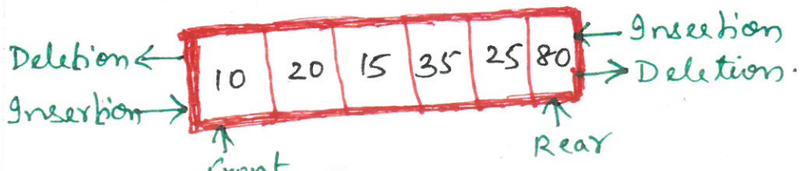
→ **Types**:
*Ascending Priority Queue
*Descending Priority Queue.

→ **Basic Operations** :- Insert & Delete.
(Enqueue) (DeleteMin)

→ **Implementation**:-
* Linked List
* Binary Search Tree (BST)
* Binary Heap.



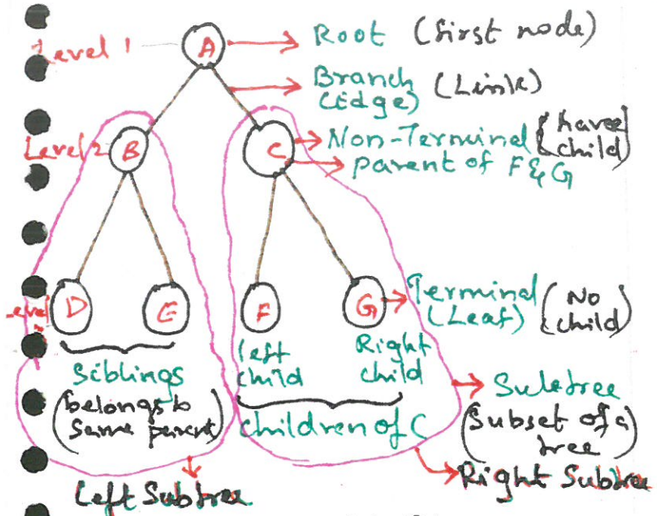
Double-Ended Queue (DEQUE)
Both insertion and deletion operations are performed at both the ends.



Queue Applications:
* Operating Systems - Job scheduling
* CPU Scheduling.
* Printer Spooler.

TOPIC-7 (PRELIMINARIES - BINARY TREE - TREE TRAVERSAL - BINARY SEARCH TREE)

TREES - a Non-Linear DS
*represents hierachial relation



Note:- N nodes → N-1 Edges

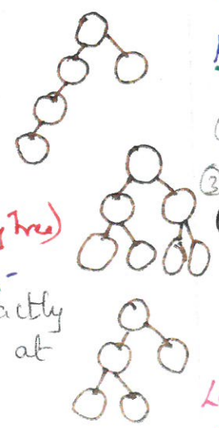
Height - No. of edges from leaf - particular node.
Depth - No. of edges from root - particular node
Length - No. of nodes in that path.
Degree - No. of children of a node.
Climbing - Traversing from leaf → root
Descending - Traversing from root → leaf
Simple Tree - Node can have any no. of children.
Binary Tree - Node can have atmost 2 children (Degree = Max = 2)
Null Tree - Tree with only root node.

Binary Tree - Application:
→ Efficient Searching & Sorting algms.

Binary Tree - Types

***Skewed Binary Tree** - nodes are added only in one side.

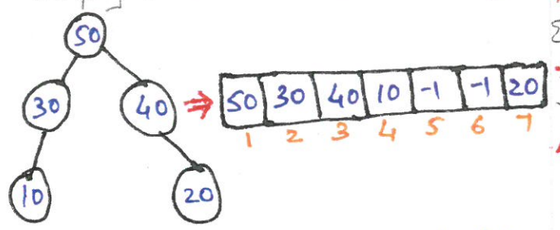
***Strictly Binary Tree** - nodes with either 2 or no node at all (Full Binary Tree)



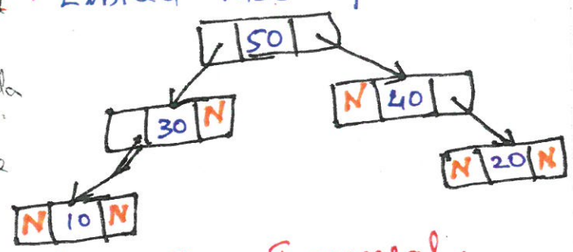
***Complete Binary Tree** - Internal nodes has exactly 2 children & leaf nodes at same level.

Binary Tree - Representation

***Array Representation**
Root → position 1
Parent (i) → i/2 (i → Node Position)
Left child → 2i
Right child → 2i+1
Empty Position → -1 (No Node)

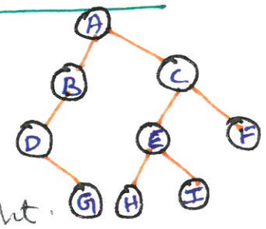


***Linked List Representation**

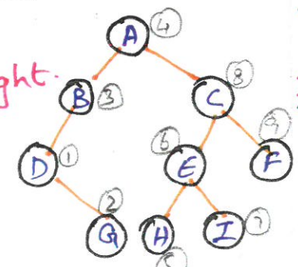


Binary Tree - Traversal.
↳ Inorder
↳ Pre-order
↳ Post-order

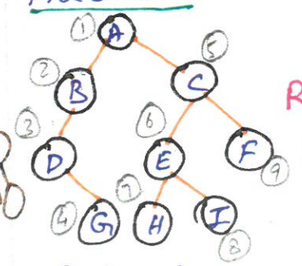
Note: Traverse from left → Right.



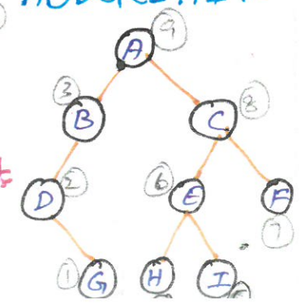
Inorder:
Left → Root → Right
DGBAHEICF



Pre order:
Root → Left → Right
ABDGCEHIF



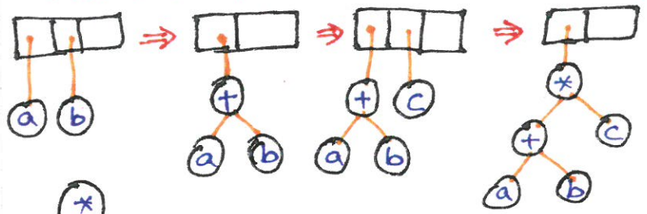
Post-order:
Left → Right → Root
GDBHIEFCA



Binary Tree - Expression Tree
→ Leaf Nodes - Operands;
→ Non-Leaf Nodes - Operators
→ Traversal by inorder, preorder & postorder
→ First, convert infix → postfix expression

Steps:-
*input symbol - operand ⇒ create a new node & push a pointer onto the stack.
*input symbol - operator ⇒ Pop 2 nodes with root as operator.

Eg:- (a+b)*c → ab+c*
[Infix expression → Postfix expression]

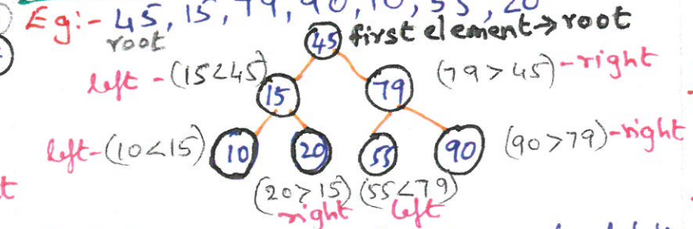


(a+b)*c → Expression Tree

Binary Search Tree (BST)
Left Subtree < Root < Right Subtree
(elements)

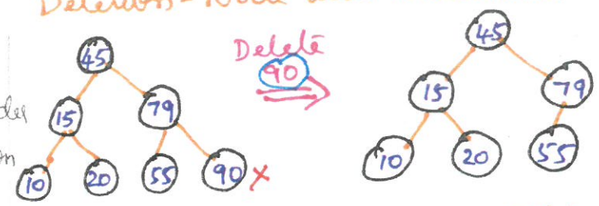
BST - Operations:
Insert, Delete, Search.

Insert: Comparing with root/Parent.
Eg:- 45, 15, 79, 90, 10, 55, 20

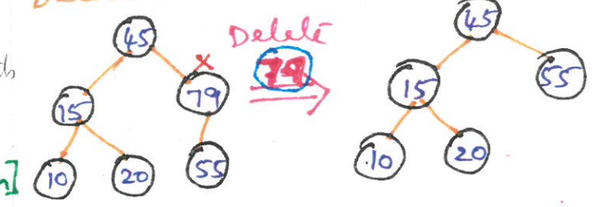


Delete: 3 categories in node deletion
→ Node may not have any children
→ Node may have only one child
→ Node may have two children

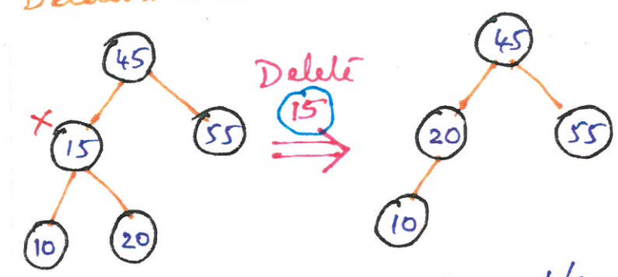
Deletion - Node with No children.



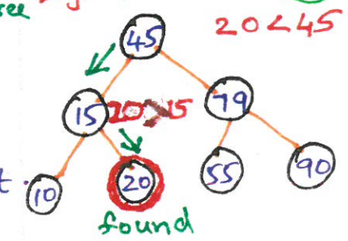
Deletion - Node with one child.



Deletion - Node with two children



Search: Comparing with root/Parent
Eg:- Search (20)



Note:
*Minimum element lies as the left most node.
*Maximum element lies as the right most node.

* GAME ENGINE
* COMPUTER GRAPHICS

TOPIC 8 - AVL TREE

AVL TREE: Balanced Search Tree

Adelson-Velskii - Landis

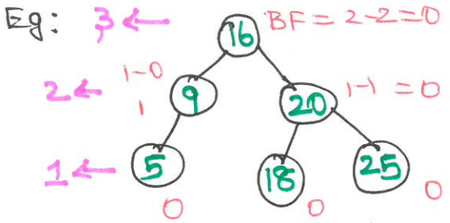
Balance Factor (BF) = Height of the Left Sub Tree − Height of the Right

BF = −1, 0, 1 => Balanced

Eg: 3 ← BF = 2−2 = 0

2 ← 1−0 = 1

1 ← 1−1 = 0



Tree is Balanced

Height (left subtree) = 2

Height (Right subtree) = 2

Balance Factor = 2 − 2 = 0

Types of Rotations:

* If tree not balanced (BF = 1, −1, 0) then perform rotations to balance
* AVL causes Unbalanced when insert

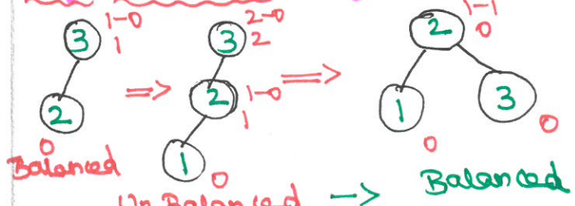
Left Subtree - Left Child (LL) } Single Rotation

Right Subtree - Right Child (RR) }

Left Subtree - Right Child (LR) } Double Rotation

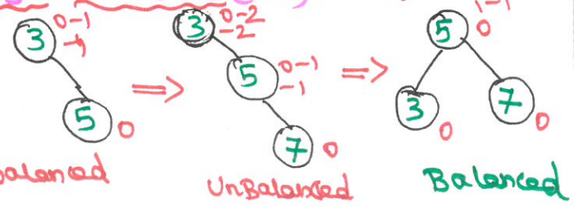
Right Subtree - Left Child (RL) }

Left Rotations (LL): Single



Balanced → Unbalanced → Balanced

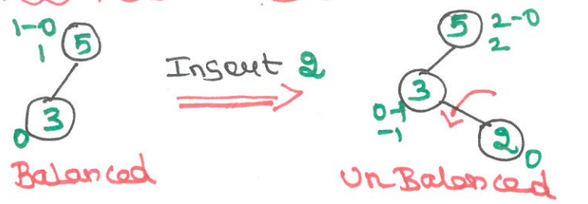
Right Rotations (RR): Single



Balanced → Unbalanced → Balanced

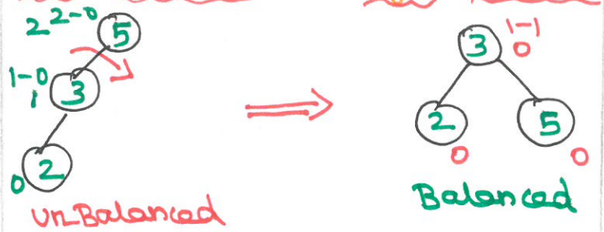
Left Right Rotations (LR): Double

Insert 2



Balanced → Unbalanced

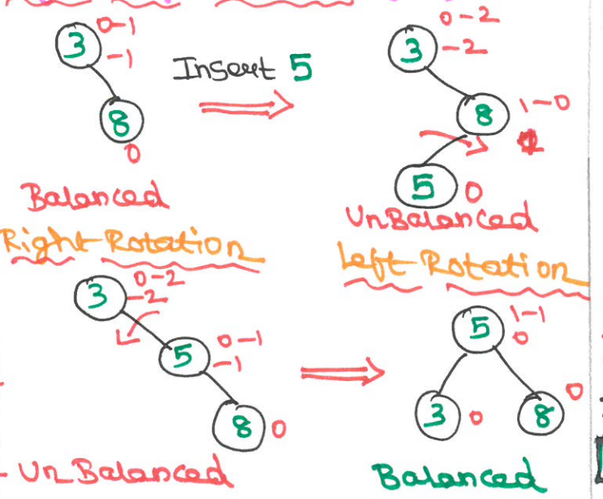
Left Rotation



Unbalanced → Balanced

Right Left Rotations (RL)

Insert 5



Balanced → Unbalanced

Right Rotation

Unbalanced

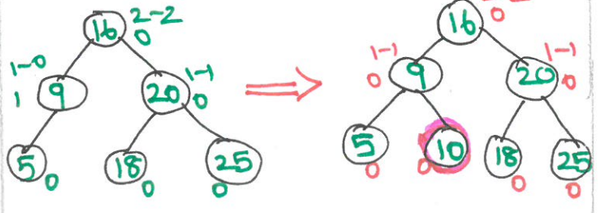
Left Rotation

Balanced

Operations : - AVL Tree

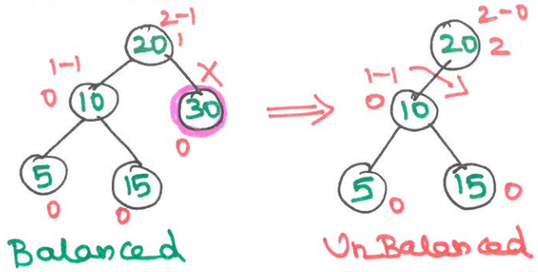
* Insertion
* Deletion
* Search

Insertion: Insert '10'

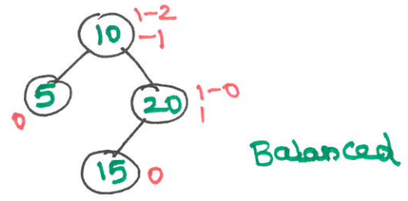


Balanced → Balanced

Deletion: Delete '30'

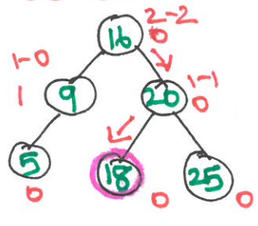


Balanced → Unbalanced



Balanced

Search: Search '18'



18 > 16 -> Traverse Right

18 < 20 -> Traverse Left

18 Found

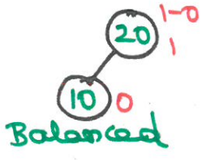
Example: Balance a tree using AVL

Insert 20, 10, 40, 50, 90, 30

Insert 20

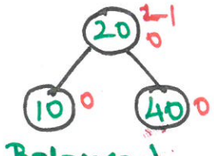
Balanced

Insert 10



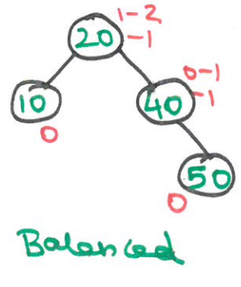
Balanced

Insert 40



Balanced

Insert '50'

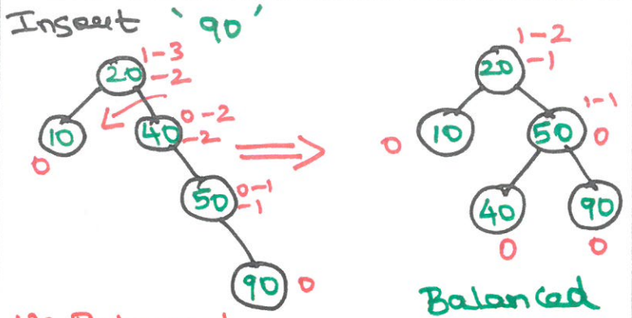


Balanced

If tree is Balanced

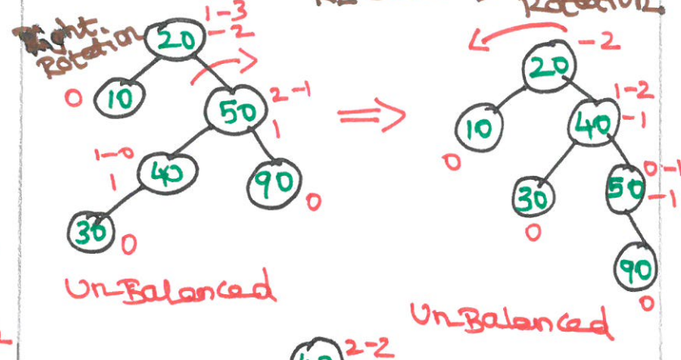
No rotations required

Insert '90'



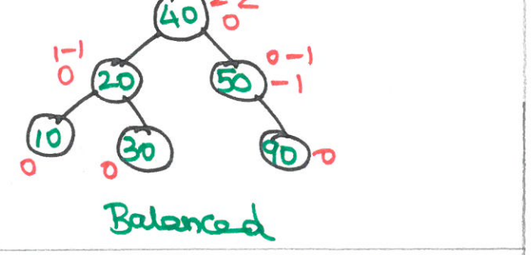
Unbalanced → Balanced

Insert '30'



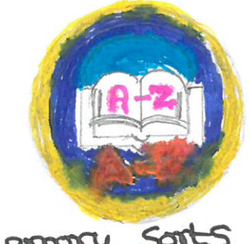
Unbalanced → Unbalanced

RL case -> Left Rotation



Balanced

APPLICATIONS: ONLINE DICTIONARY

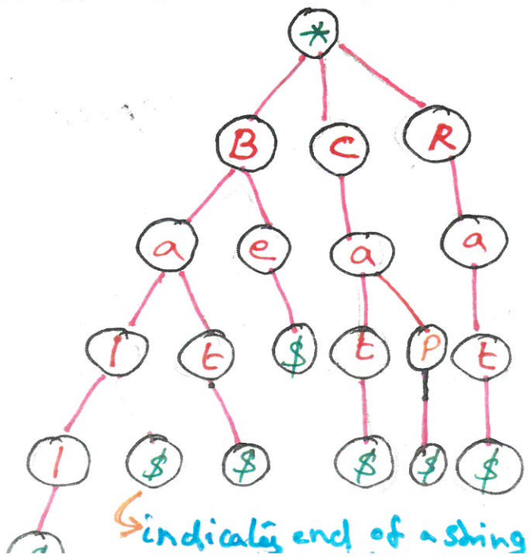


* In memory sorts of Dictionaries
* Search Engine Optimization
* Equal Distance balancing from Server to client node in networks.

TOPIC-9 (APPLICATIONS OF SEARCH TREES - B TREE - TRIE - 2-3 TREE - 2-3-4 TREE)

TRIE

→ All the search trees are used to store the collection of numerical values but they are not suitable for storing the collection of words or strings.
→ TRIE data structure makes retrieval of a string from the collection of strings more easily.
→ Term 'TRIE' comes from the word retrieval
→ It is a special kind of tree that is used to store the dictionary.
→ It is a fast & efficient way for dynamic spell checking.
→ Every node in Trie can have one or a number of children.
→ All the children of a node are alphabetically ordered. If any two strings have a common prefix then they will have the same ancestors.
Eg: Consider the following list of strings to construct Trie Cat, Bat, Ball, Rat, Cap & Be



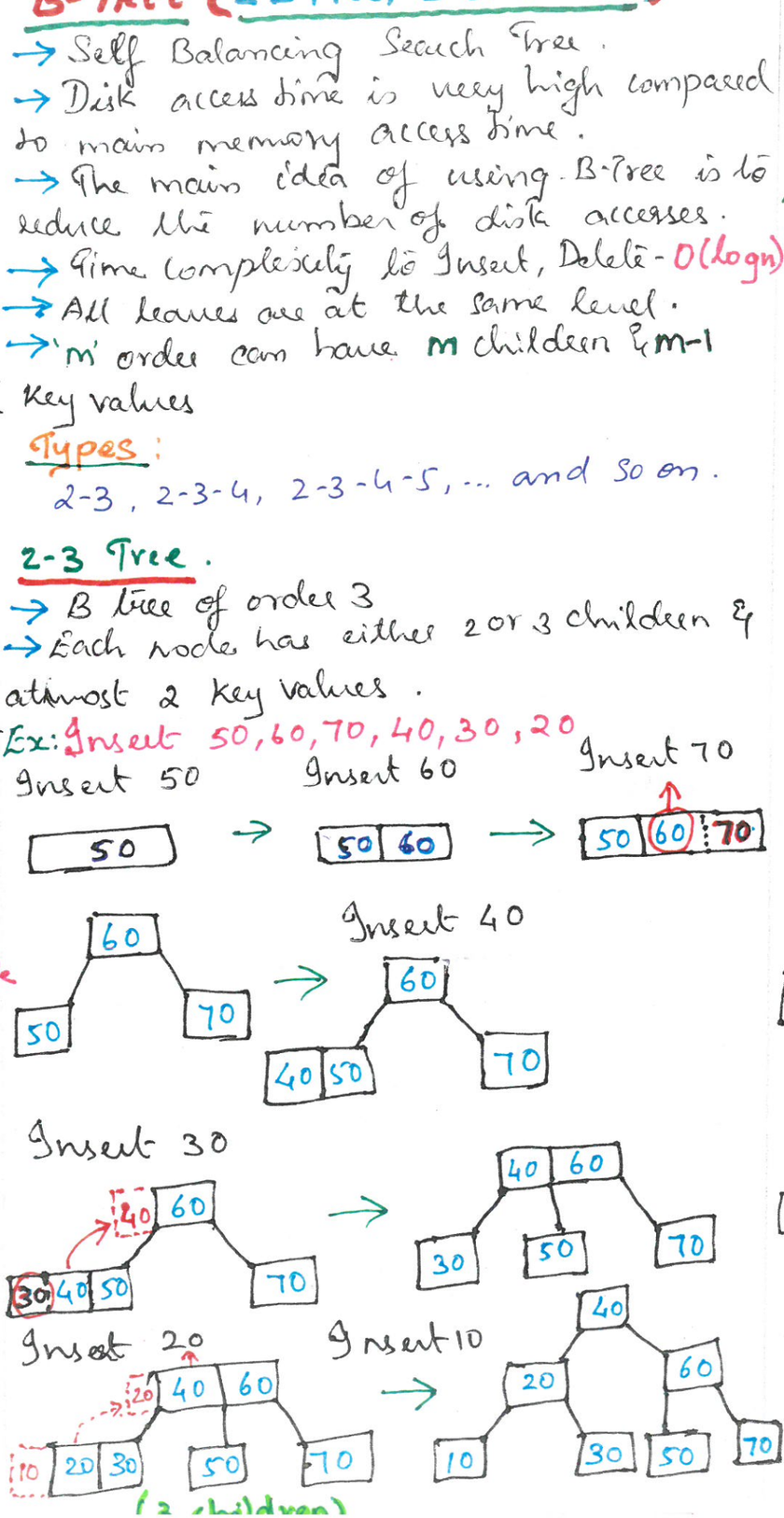
B-TREE (2-3 Tree, 2-3-4 Tree)

→ Self Balancing Search Tree.
→ Disk access time is very high compared to main memory access time.
→ The main idea of using B-Tree is to reduce the number of disk accesses.
→ Time complexity to Insert, Delete - O(log n)
→ All leaves are at the same level.
→ 'm' order can have m children & m-1 key values

Types:
2-3, 2-3-4, 2-3-4-5, ... and so on.

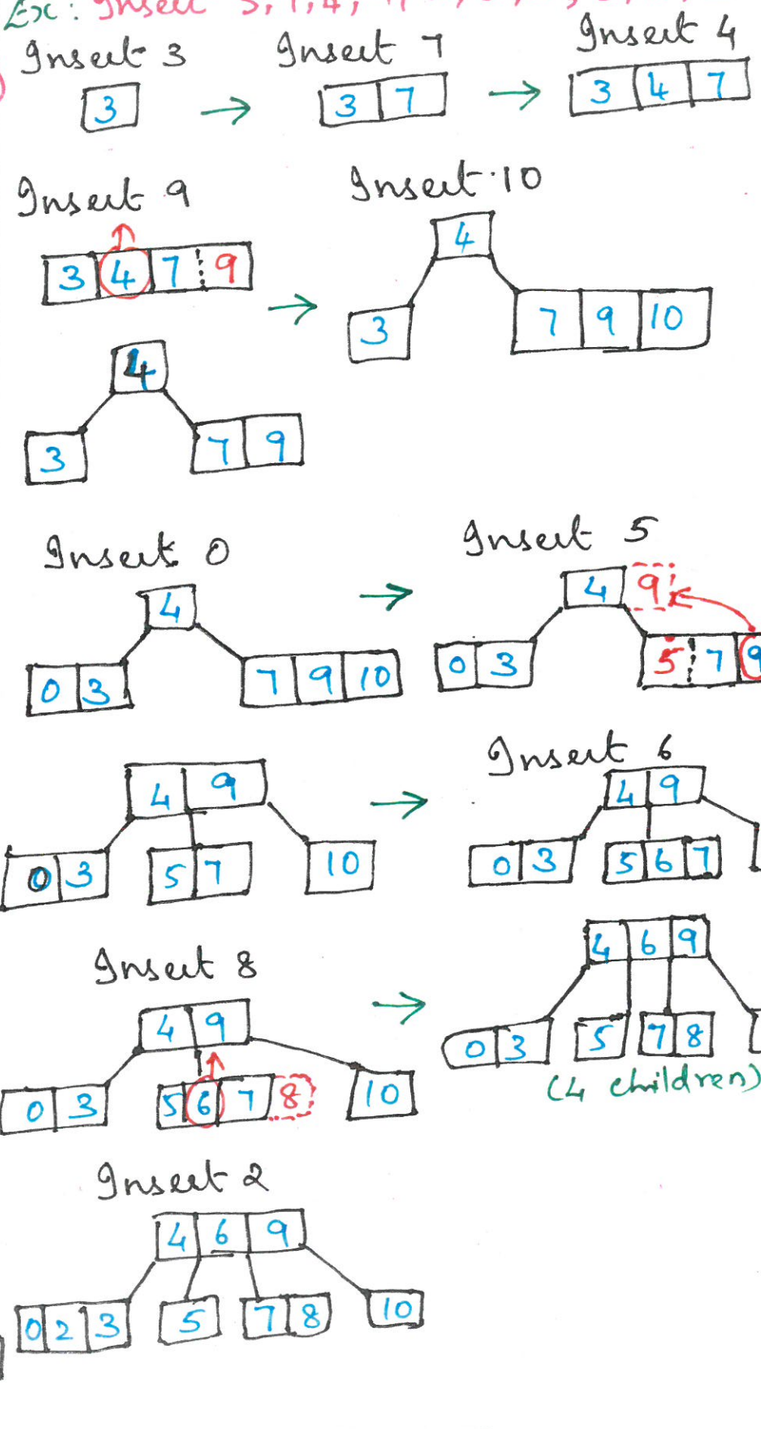
2-3 Tree.

→ B tree of order 3
→ Each node has either 2 or 3 children & atmost 2 key values.
Ex: Insert 50, 60, 70, 40, 30, 20



2-3-4 Tree

→ B tree of order 4
→ Each node has either 2/3/4 children & atmost 3 key values.
Ex: Insert 3, 7, 4, 9, 10, 0, 5, 6, 8, 2



TOPIC:10 RED BLACK TREE

RED-BLACK TREE:

BST in which every node is colored either **RED** or **BLACK**

Properties

→ BST → Newnode - **RED**
→ Root Node - **BLACK**
→ No 2 consecutive **RED** nodes
→ All paths same no of BLACK nodes
→ Leaf node - **BLACK**

Insertion Need to satisfy properties if not, perform operations to make it **RED-BLACK** Tree

1. Recolor
2. Rotations
3. Rotations Followed by Recolor

STEPS

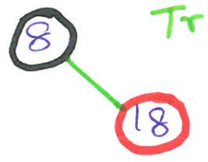
1. check Tree is empty
2. Empty insert Newnode as ROOT (BLACK)
3. Not empty insert New node as a Leaf node (RED)
4. Parent (New Node) BLACK exit
5. Parent (NewNode) RED check Color of Parent node's sibling of New Node
   - Sibling **BLACK** or NULL Rotate and Recolor it
   - Sibling **RED** Recolor repeat the steps until tree beomes **RED-BLACK** Tree

Example: Create **RED-BLACK** Tree by inserting 8, 18, 5, 15, 17, 25, 40, 80

Insert 8

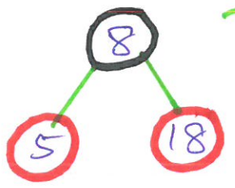
NewNode BLACK

Insert 18



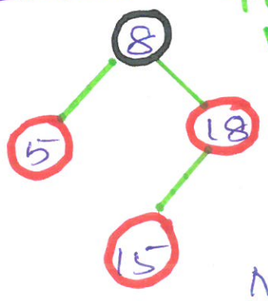
Tree Not empty
New Node **RED**

Insert 5



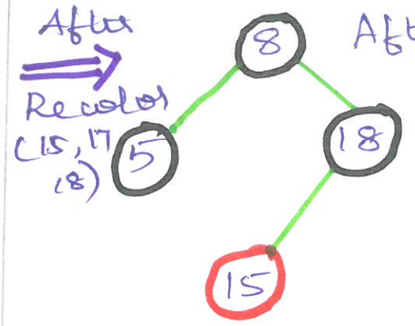
Tree Not empty
New Node **RED**

Insert 15

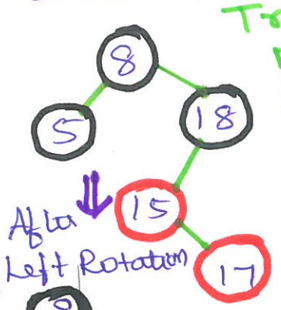


Tree not empty
New Node **RED**
18, 15 **(RED)** (consecutive)
New Node Parent
Sibling **RED** (5)
Recolor

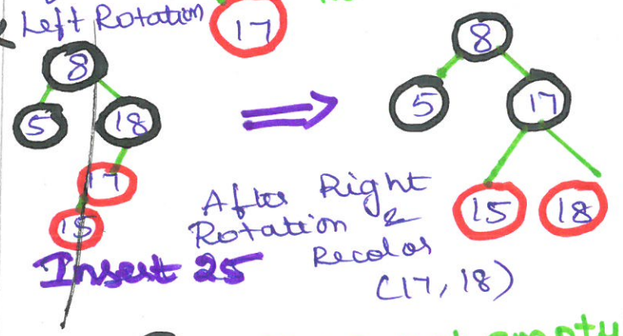
Note: Dont Recolor if Parent - root (BLACK)



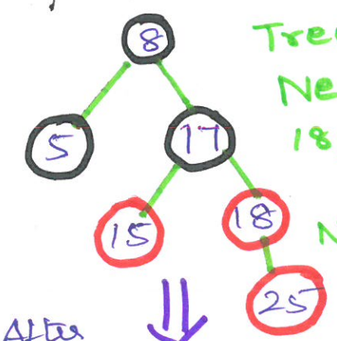
Insert 17



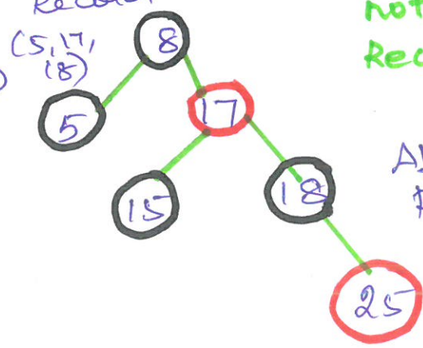
Tree not empty
New Node **Red**
15, 17 **RED** (Consecutive)
New Node Parent
Sibling NULL
Rotate & Recolor



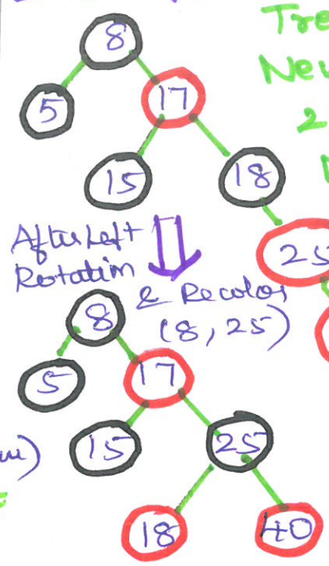
Insert 25



Tree not empty
NewNode **RED**
18, 25 **(RED)** (Consecutive)
New Node Parent
Sibling **RED** (15)
Parent Parent's not root
Recolor

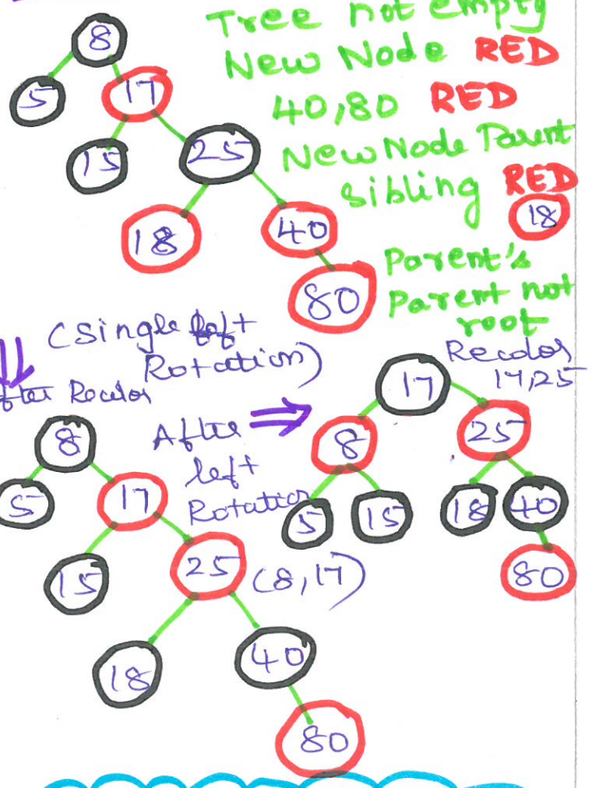


Insert 40



Tree not empty
New Node **RED**
25, 40 **RED**
New Node's Parent's sibling NULL
Rotate & Recolor
RR-Case
(Single left Rotation)

Insert 80



Tree not empty
New Node **RED**
40, 80 **RED**
New Node Parent
Sibling **RED** (18)
Parent's Parent not root
Recolor

Searching words in Dictionaries
Add or Delete items in cart - Online Shopping

TOPIC: 11

Splay tree

Splay tree is a self-adjusted Binary Search tree in which every operation on an element rearrange the tree so that element is placed at the root position of the tree.

Splaying an element is the process of bringing it to the root position by performing suitable rotation operation.

Rotation in Splay Tree.

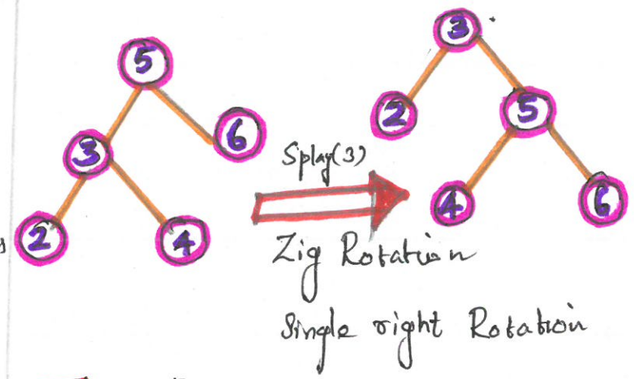
* Zig Rotation
* Zag Rotation
* Zig-Zig Rotatwin
* Zag-Zag Rotation
* Zig-Zig Rotation
* Zag-Zig Rotation

Did you Know?

SplayNet [Facebook]

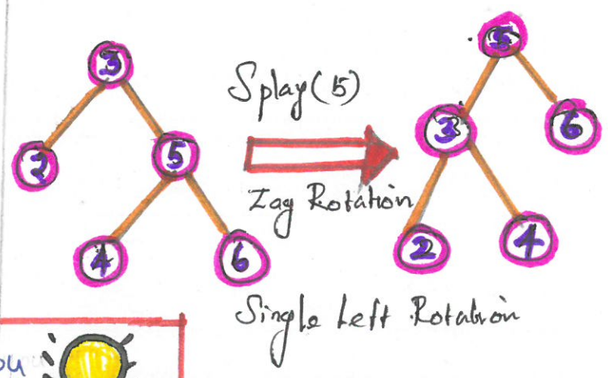
Zig Rotation:

The Zig Rotation in a Splay tree is Single right rotation in AVL Tree rotation. In Zig rotation every node moves one position to the right from its Current position.



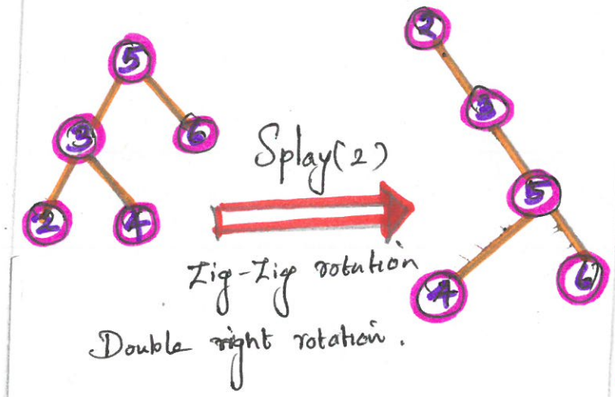
Zag Rotation:

The Zag Rotation in a Splay tree is Similar to the Single Left Rotation in AVL tree rotations. In Zag rotation every one moves one position to the left from its Current position.



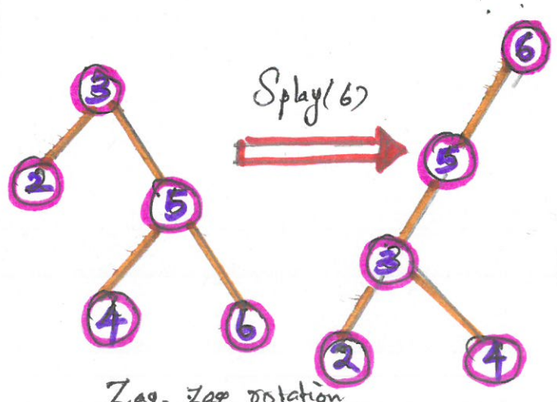
Zig-Zig Rotation:

The Zig-Zig rotation in a Splay tree is a double Zig rotation. In Zig-Zig rotation every nodes Moves two position to the right from its Current position.



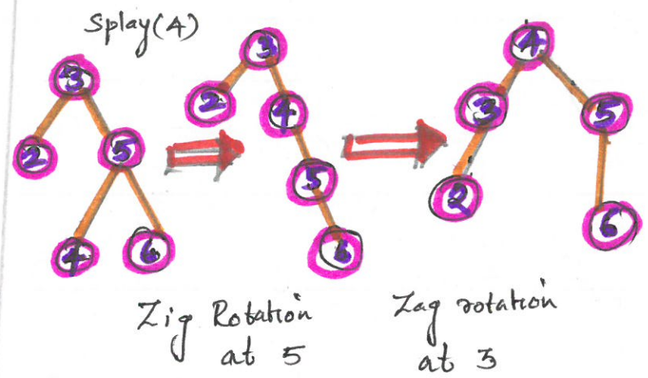
Zag-Zag Rotation:

The Zag-Zag rotation in a splay tree is a double Zag rotation. In Zag-Zag rotation every node moves two position to the left from its Current position.



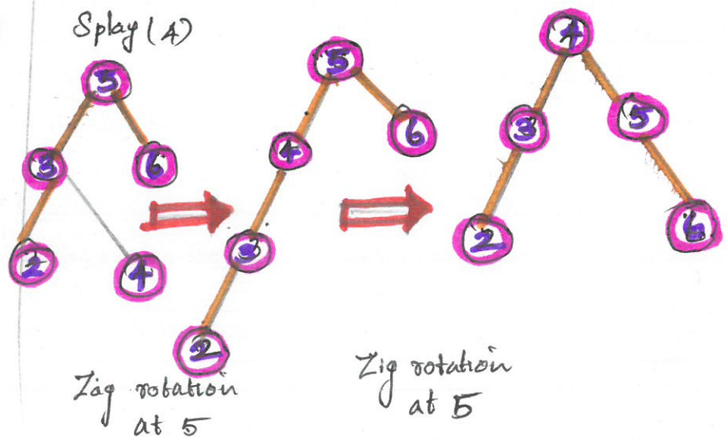
Zig-Zag Rotation:

The Zig-Zag rotation in a Splay tree is a Sequence of Zig-Zag rotation every node moves one position to the right followed by one position to the left from its Current position.



Zag-Zig Rotation:

The Zag-Zig Rotation in a Splay tree is a Sequence of Zag rotation followed by Zig rotation In Zag-Zig rotation every one moves one position to the left- followed by one position to the right from its Current.



TOPIC-12 (HASHING - HASH FUNCTION - SEPARATE CHAINING - OPEN ADDRESSING

Hashing

Hash Table Data Structures - Arrays of keys

Keys - Strings with associated value.

Hashing - Implementation of Hash Table.

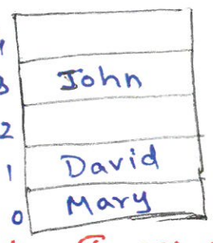
Hash Function - Mapping each Key into Hash Table index with size [0-(Hsize-1)]

Hsize - Hash Table Size

Example:

Hsize = 5

John hashes to 3
Mary " " 0
Devid " " 1

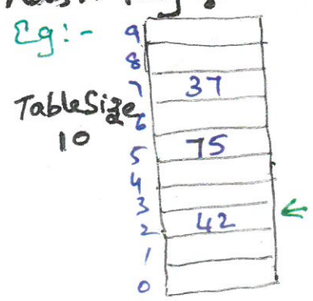


Hash Function - Types:

* Division Method:
  Hash(Key) = Key Mod TableSize
  Eg:- hash(92) = 92 Mod 10 (TableSize=10) = 2
* Mid-Square Method: Key is squared and mid part of result forms Index.
  Eg:- hash(3101) = 3101*3101 = 9616201 ⇒ 162 (mid 3 digit)
* Digit-Folding Method:
  Key is divided into separate parts & combine all by using some operations.
  Eg:- hash(124|655|12) = 124+655+12 = 791

Collision: Hash function returns same hash key for more than one record.

Eg:-



hash(42) = 42%10 = 2
hash(37) = 37%10 = 7
hash(75) = 75%10 = 5
hash(12) = 12%10 = 2 ← 12 ↳ Collision Occurs

⇒ Collision Resolution Techniques is used to resolve Collisions.

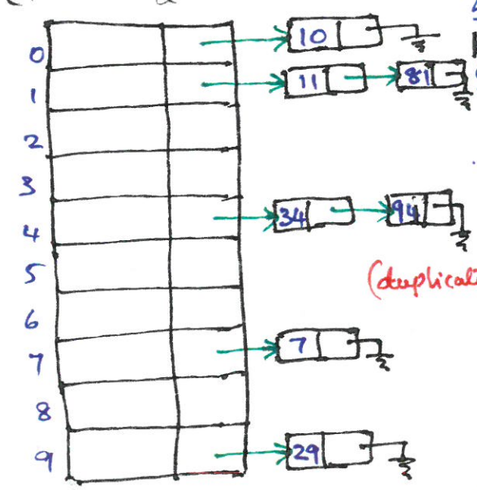
Collision Resolution Techniques

- Separate Chaining
- Open Addressing
  - Linear Probing
  - Quadratic "
  - Double Hashing

Separate Chaining:-

* Open Hashing
* No collision occurs
* Use Linked List Concept
* Utilize more memory space.
* Less Cache Performance

Example: Insert 10, 11, 81, 10, 7, 34, 94, 29.
(Table-Size = 10)



Insert 10, Hash(10) = 10 Mod 10 = 0 (%)
Insert 11, Hash(11) = 11 % 10 = 1
Insert 81, Hash(81) = 81 % 10 = 1
Insert 10, Hash(10) = 10 % 10 = 1 (duplicate)
Insert 7, Hash(7) = 7 % 10 = 7
Insert 34, Hash(34) = 34 % 10 = 4
Insert 94, Hash(94) = 94 % 10 = 4
Insert 29, Hash(29) = 29 % 10 = 9

To overcome the disadvantage of Separate chaining we go for Open Addressing.

Open Addressing:-

* Closed Hashing
* Use Array Concepts
* Alternate cells are tried until empty cell is found.
* 3 Strategies:
  - ↳ Linear Probing
  - ↳ Quadratic Probing
  - ↳ Double Hashing

Linear Probing

Index = {(U+i) % M}
U → Key ; M → TableSize ; i → iteration
Eg:- Insert Key = {89, 18, 49, 58, 69}

TableSize = 10

	After 89	After 18	After 49	After 58	After 69
0			49	49	49
1				58	58
2					69
3					
4					
5					
6					
7		18	18	18	18
8			89	89	89
9	89	89			

Insert 89, U=9
index = (9+0) % 10 = 9 (i=0) Empty
Insert 18, index = 18 % 10 = 8 ⇐ U Empty
i=0, index = (8+0) % 10 = 9 ⇐ U
Insert 49, index = 49 % 10 = 9 Not empty
i=0, index = (9+0) % 10 = 9
i=1, index = (9+1) % 10 = 0 Empty
Insert 58, index = 58 % 10 = 8 ⇐ U
i=0, index = (8+0) % 10 = 8 Not empty
i=1, index = (8+1) % 10 = 9 Not empty
i=2, index = (8+2) % 10 = 0 Not empty
i=3, index = (8+3) % 10 = 1 Empty
Insert 69, index = 69 % 10 = 9 ⇐ U
i=0, index = (9+0) % 10 = 9 Not empty
i=1, index = (9+1) % 10 = 0 Not empty
i=2, index = (9+2) % 10 = 2 Empty

Quadratic Probing

Index = {(U + i²) % M}

U → Key ; M → TableSize ; i → Iteration
* Similiar to Linear Probing

Double Hashing

Index = {(index + i * indexH) % M}

indexH → Hash value computed by another Hash function

Priority Queue:

* Processing of an object based on priority.
* Priority Heap

Implementation:
- → Linked List
- → Binary Search Tree
- → Binary Heap

Binary Heap: Heap DS created using binary tree.

Heapify: process of creating binary heap DS either in Min_Heap or Max_Heap.

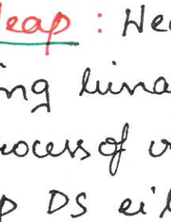
Types:
- → Max_Heap (parent > children)
- → Min_Heap (parent < children)

2 properties:
- → Shape property
  - ↳ Complete Binary Tree (CBT)
- → Heap property
  - ↳ Max_Heap or Min_Heap

Operations:
Build, Insert, Delete.

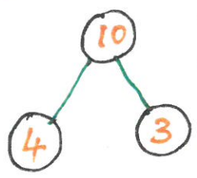
Example: Build binary Heap by inserting {4, 10, 3, 5, 1}

Insert 4,



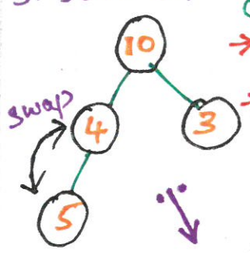
Check properties:
- → Shape property - complete BT
- → Heap property - Not satisfied (Max_Heap). So swap. (10, 4)

Insert 3,

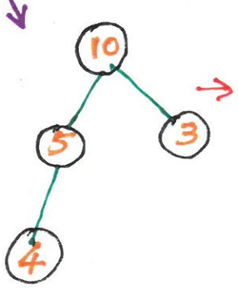


Check properties:
- → Shape property - satisfied
- → Heap property - Satisfied (Max_Heap)

Insert 5,

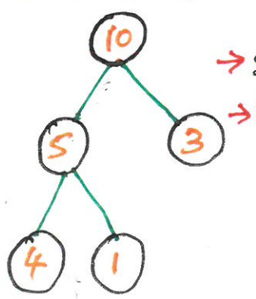


Check properties:
- → Shape property - satisfied
- → Heap property - Not " (Max_Heap). So swap. (4, 5)



→ Heap property satisfied.

Insert 1,



Check Properties:
- → Shape property - Satisfied
- → Heap property - "

Binary Heap (Max_Heap) is constructed successfully.

Note:-
In Max_Heap, maximum element will be at root.
In Min_Heap, minimum element will be at root.

Heap Sort:

Sorting technique based on Binary heap DS.

* 2 ways:
  - → Max_Heap Sort - Descending order
  - → Max_Heap Sort - Ascending "

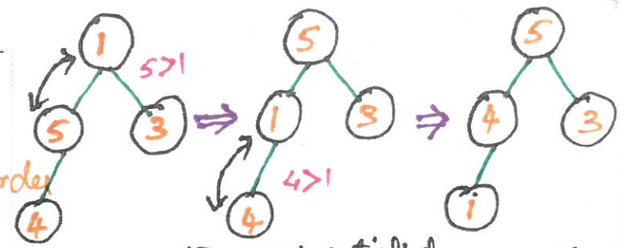
Max_Heap Sort:
uses Delete_Max operation.

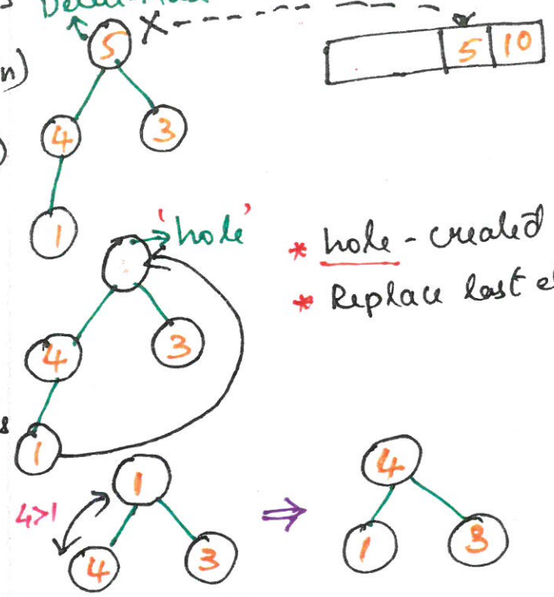
Example: Heap Sort - {4, 10, 3, 5, 1}

Steps:-
1) Construct Binary Heap (Max/Min)
2) Call Delete_Max / Delete_Min operation (root element - deleted)
3) Place the deleted element in the sorting list.
4) "Hole" will be created.
5) Replace lastly inserted element at the root.
6) Check binary heap properties and satisfy it.
7) Repeat steps till get sorted list.

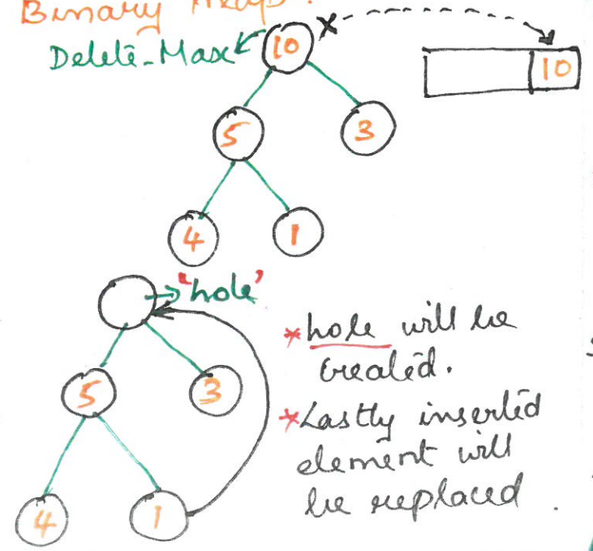
Now, Solve {4, 10, 3, 5, 1} using Max_Heap Sort (Ascending order)

Binary Heap.

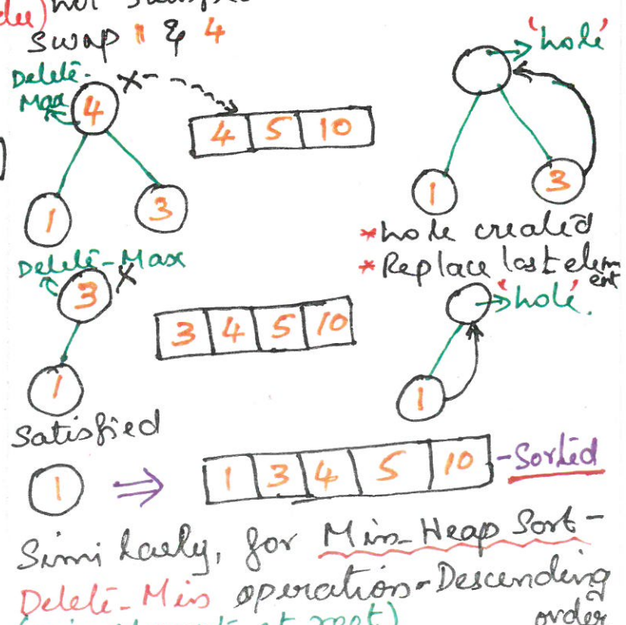




* hole - created
* Replace last element



*hole will be created.
*Lastly inserted element will be replaced.



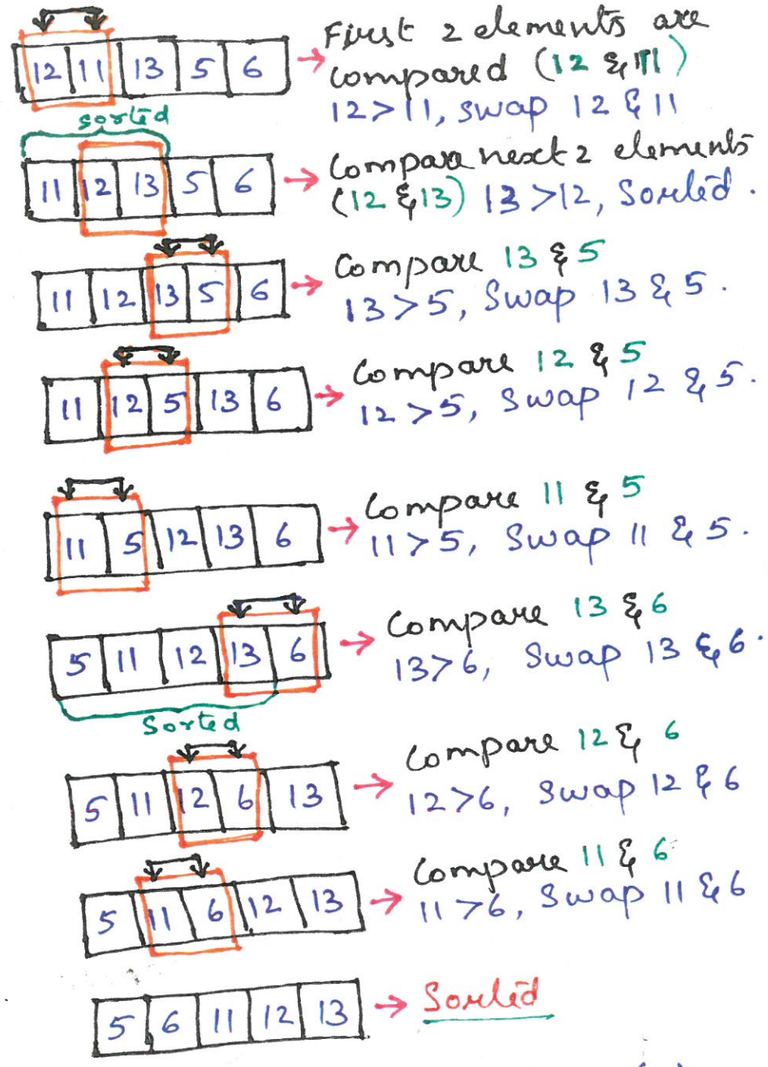
Similarly, for Min_Heap Sort - Delete_Min operation - Descending order (min element at root)

TOPIC-14 (INSERTION SORT, MERGE SORT, RADIX SORT)

Insertion Sorting:

*Simple
*Efficient for small data values.

Example: Sort 12,11,13,5,6 Using Insertion sort.

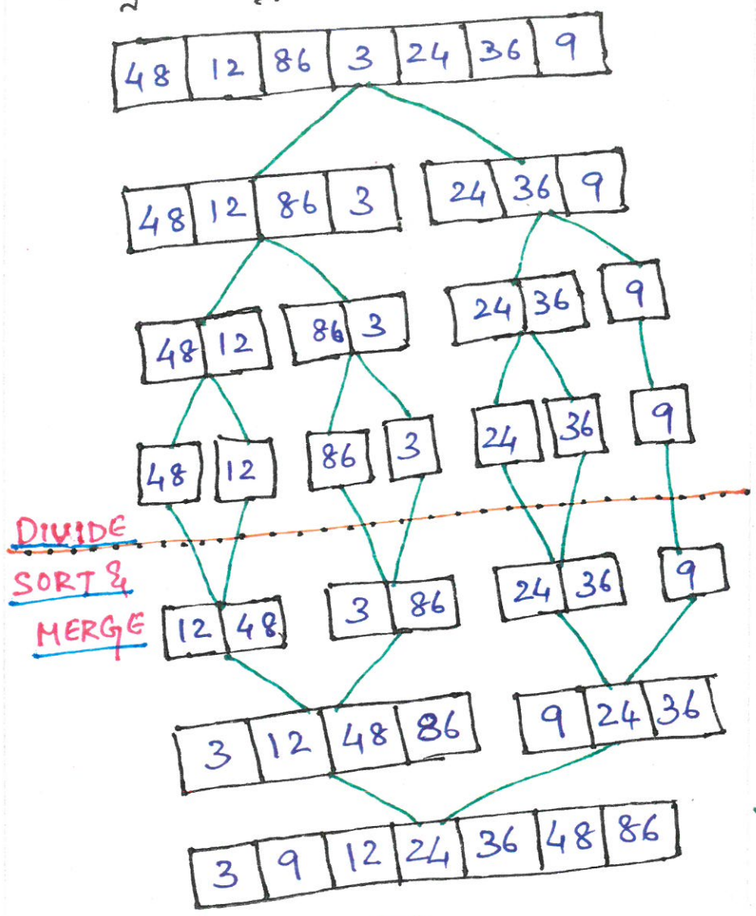


Best Case Time Complexity - $O(n)$
Worst Case Time Complexity - $O(n^2)$
Average Case Time Complexity - $O(n^2)$

Merge Sort:

* Divide & Conquer Method
* Recursively dividing the array into 2 halves, Sort & then merge

Example:- Sort 48,12,86,3,24,36,9 Using merge sort.



Sorted.

Time Complexity - $O(n \log n)$

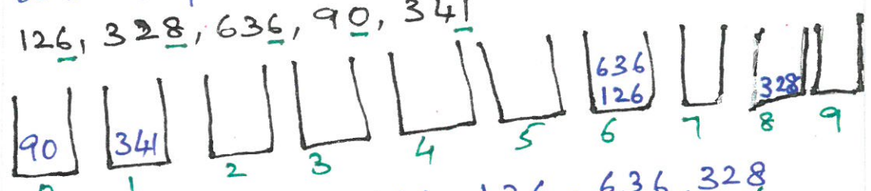
Radix Sort:

*Non Comparative integer Sorting algorithm
* Digit-by-digit Sorting (Sorting done from least significant digit)

Example:- Sort 126, 328, 636, 90, 341 Using Radix Sort.

Pass I: Consider the 1's place and keep it in respective buckets (0-9)

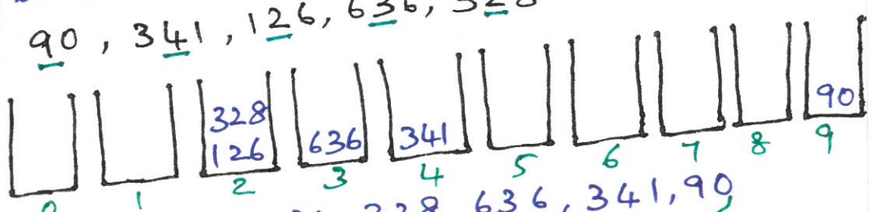
126, 328, 636, 90, 341



→Arrange:- 90, 341, 126, 636, 328 (input for Pass II)

Pass II: Consider the 10's place and keep it in respective buckets (0-9)

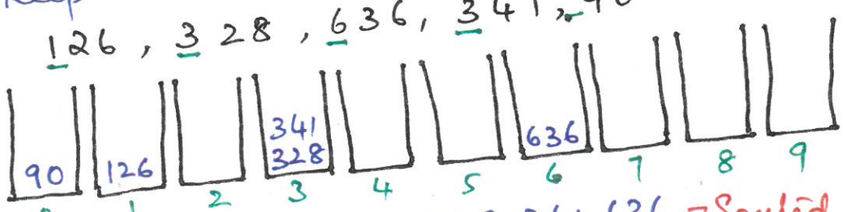
90, 341, 126, 636, 328



→Arrange:- 126, 328, 636, 341, 90 (input for Pass III)

Pass III: Consider the 100's place and keep it in respective buckets (0-9)

126, 328, 636, 341, 90



→Arrange:- 90, 126, 328, 341, 636 - Sorted

Note:- No. of digits = No. of Passes.

Time Complexity - $O(nd)$.
(n → array size, d → no. of digits)

TOPIC -15 (QUICK SORT)

Quick Sort:

* Divide & Conquer Method

Divide :- partition the array into 2 sub-arrays

Conquer :- recursively, sort 2 subarrays.

Combine :- combine the already sorted array.

* choosing a pivot element [Mean, Median, First, Last..]

Steps :-

Step 1: Choose a pivot element from the list.

Step 2: Define 2 variables 'i' & 'j'. i → 1st & j → last value

Step 3: Increment i until list[i] > pivot, then stop

Step 4: Decrement j until list[j] < pivot, then stop

Step 5: If i < j, then swap list[i] and list[j]

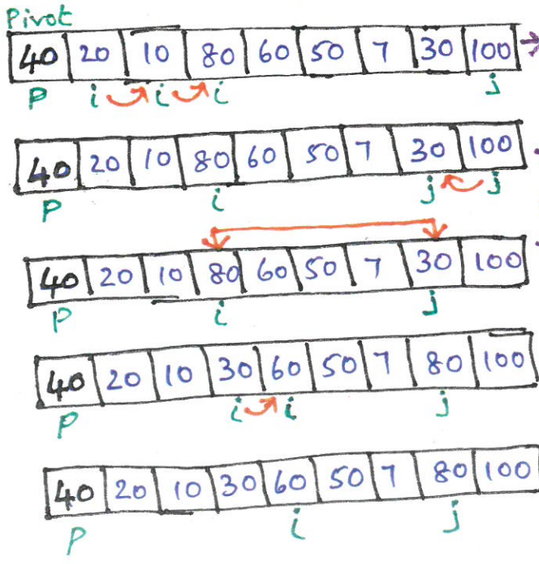
Step 6: Repeat steps 3, 4, 5 until i > j.

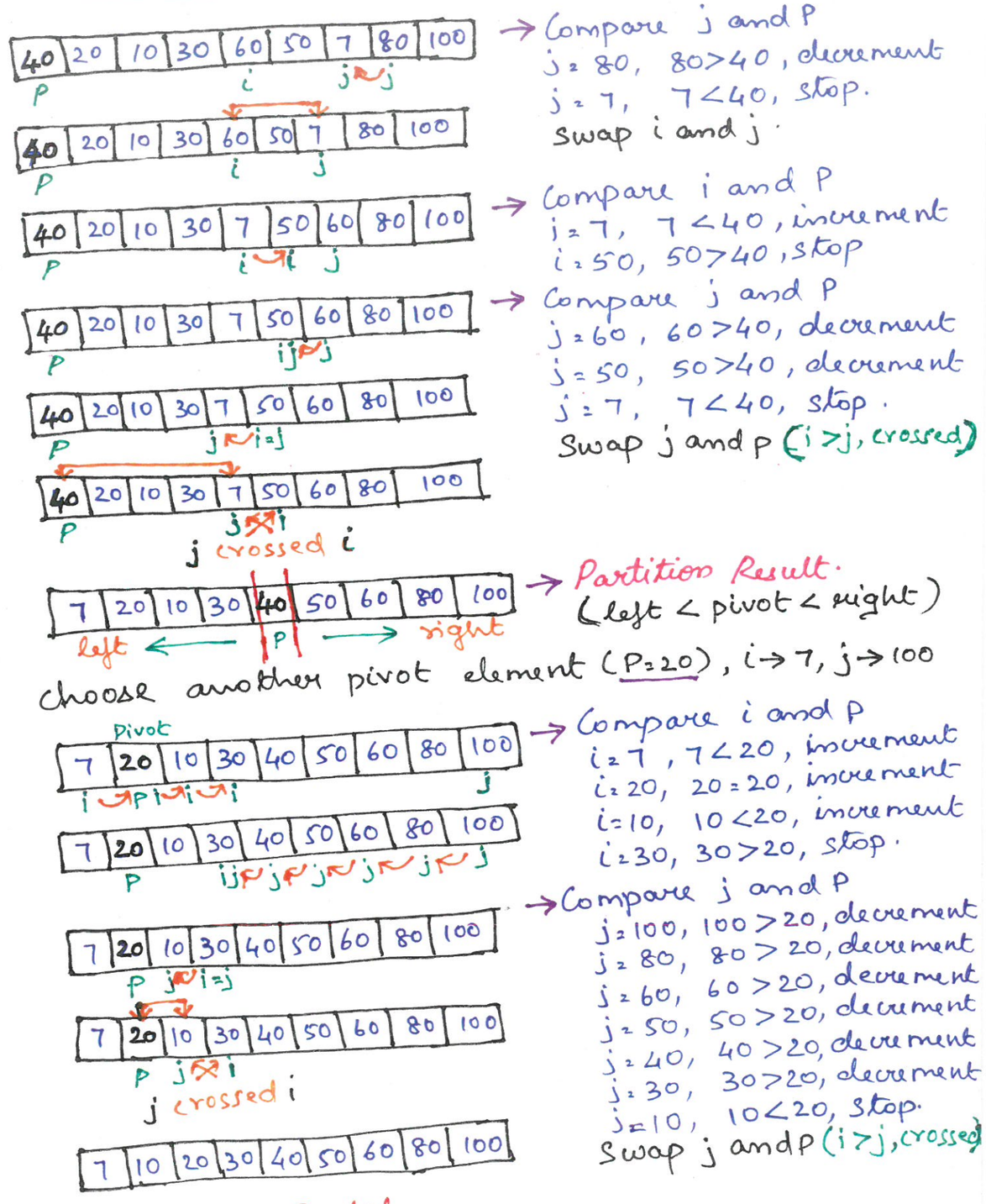
Step 7: If i > j (crossed), then swap crossed index with Pivot element.

* Partition Result → left element < pivot < right element

Step 8: Choose another pivot element & repeat the steps till gets sorted.

Example: Sort 40, 20, 10, 80, 60, 50, 7, 30, 100 Using Quick sort. Pivot (P) → 40, i → 20, j → 100 ← Consider.





Best Case Time Complexity - $O(n \log n)$

Worst Case Time Complexity - $O(n^2)$

Average Case Time Complexity - $O(n \log n)$

TOPIC 16: BUBBLE SORT, SELECTION SORT, TIME COMPLEXITY

BUBBLE SORT:

* Internal Sorting Algorithm
* Comparing two consecutive elements (a[i] with a[i+1])
* Largest element in the array bubbles towards the right end of first pass
* Repeated for all pairs till Sorted
* (n-1) passes where 'n' is the no. of elements to be sorted

Algorithm:

```
for (i=0; i<n-1; i++)   // no. of passes
{  for(j=0; j<n-1; j++)
     {  if(a[j] > a[j+1])
          {  temp = a[j];
             a[j] = a[j+1];
}  }  }  a[j+1] = temp;
```

Example:

Pass 1:

16	17	7	9	6	Unsorted
16	17	7	9	6	No change
16	17	7	9	6	Swap
16	7	17	9	6	Swap
16	7	9	17	6	Swap
16	7	9	6	17	End of pass 1

Largest element is placed at last index

Pass 2:

16	7	9	6	17	Swap
7	16	9	6	17	Swap
7	9	16	6	17	Swap
7	9	6	16	17	No change
7	9	6	16	17	End of pass 2

Pass 3:

7	9	6	16	17	No change
7	9	6	16	17	Swap
7	6	9	16	17	No change
6	7	9	16	17	No change
6	7	9	16	17	End of pass 3

Pass 4:

7	6	9	16	17	Swap
6	7	9	16	17	No change
6	7	9	16	17	No change
6	7	9	16	17	No change
6	7	9	16	17	End of pass 4

O/P: 6 7 9 16 17 — Sorted order

SELECTION SORT:

* Repeat until no unsorted element remain
  - → Search the unsorted part of the data to find Smallest value
  - → Swap Smallest found value with the first element of the unsorted part
* It contains N-1 passes.

Algorithm:

```
for(i=0; i<n-1; i++)
{  int min = i;
   for(j=i+1; j<n; j++)
   {  if (a[j] < a[min])
      {  min = j;
   }  }
   temp = a[i];
   a[i] = a[min];
   a[min] = temp;
}
```

Example:

Sorted | Unsorted — min

	7	4	11	9	3	2
Sort	Unsorted					
2	4	11	9	3	7	Pass 1
Sorted	Unsorted					
2	3	11	9	4	7	Pass 2
Sorted	Unsorted					
2	3	4	9	11	7	Pass 3
Sorted	Unsorted					
2	3	4	7	11	9	Pass 4
2	3	4	7	9	11	Pass 5

Sorted order: 2 3 4 7 9 11

Sorting	Advantage	Disadvantage
Bubble Sort	* Simple & easy for implement * Efficient when input data is sorted	Inefficient for large volume of input data
Selection Sort	* Efficient for small amount of I/p data * 60% more efficient than bubble sort	* Perform all n comparisions for sorted I/p data * Inefficient for large volume of input data * Take more time

Time Complexity:

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Heap Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
Quick Sort	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$

TOPIC - 17 (GRAPH - TERMINOLOGIES - TYPES - TOPOLOGICAL SORT)

Graphs: $G = (V, E)$

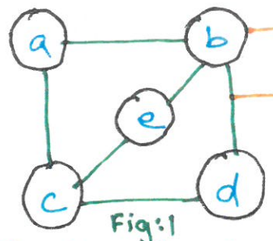


Fig:1

→ Vertex, $V = \{a, b, c, d, e\}$

→ edge, $E = \{(a,b), (a,c), (b,d), (d,c), (c,e), (b,e)\}$

Terminologies: (Refer Fig:1)

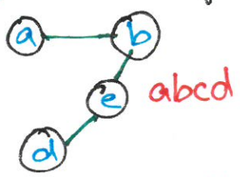
Adjacent vertices → Connected by an edge a → b, c ; c → a, e, d

Length → No. of edges.

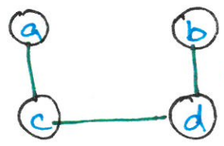
Degree → No. of adjacent vertices

Degree|a| = 2, Degree|b| = 3

Path → Sequence of vertices.

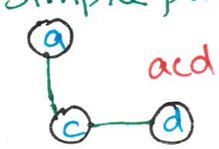


abcd

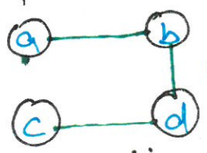


acdb

Simple path → No repeated vertices.

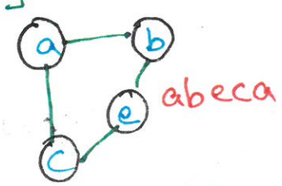


acd

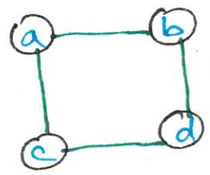


abdc

Cycle → Last & first vertex - Same.

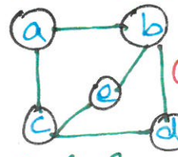


abeca

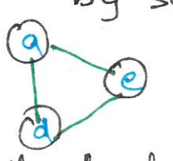


abdca

Connected Graph → 2 vertices connected by some path.

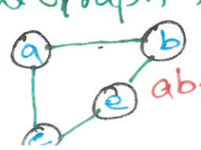


Connected Graph

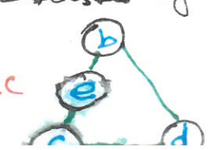


Not Connected Graph

Sub-Graph → Subset of vertices & edges



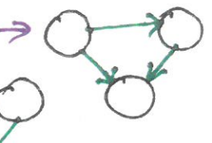
abec



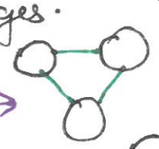
becd

Connected Components - max. connected subgraph.

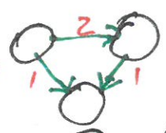
Directed Graph (Digraph): specify directions in edges. →



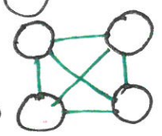
Undirected Graph: No directed edges. →



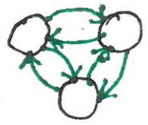
Weighted Graph: have weights in edges →



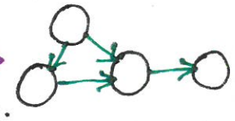
Complete Graph: all vertices are connected to each other (Undirected Graph) →



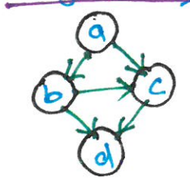
Strongly Connected Graph: → all vertices are connected in both directions (Directed Graph)



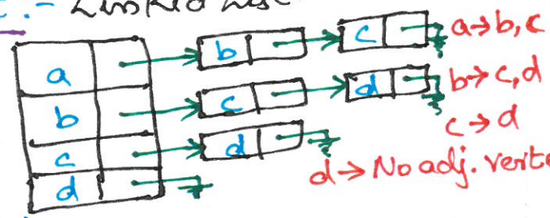
Acyclic Graph: No cycles. →



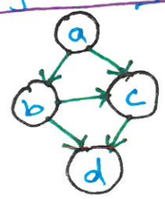
Adjacency List :- Linked list.



→



Adjacency Matrix :-



→

	a	b	c	d
a	0	1	1	0
b	0	0	1	1
c	0	0	0	1
d	0	0	0	0

* 4 Vertices
* 4x4 Matrices

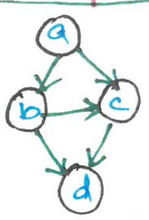
Topological Sort : used in Directed Acyclic Graph (DAG) - directed & No cycles.

* Sorting - DAG.
* if u → v, then v appears after u

Algorithm steps:

→ Compute indegrees of all vertices
→ Find vertex V with indegree 0, place in ordered list.
→ Remove V and its edges (U,V)
→ Update indegree of remaining vertices
→ Repeat till all vertices are in ordered list.

Example:



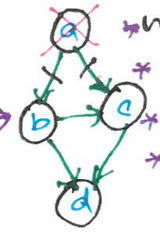
⇒

Adjacency Matrix:

	a	b	c	d
a	0	1	1	0
b	0	0	1	1
c	0	0	0	1
d	0	0	0	0

indegree[a] = 0, indegree[b] = 1
indegree[c] = 2, indegree[d] = 2

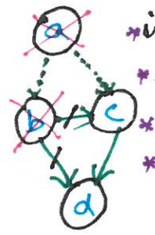
Vertex	1	2	3	4
a	0			
b	1			
c	2			
d	2			
Enqueue : a				
Dequeue : a				



* indegree[a] = 0.
* Enqueue a
* Dequeue a → list.
* Remove adj. edges of a → b, c

indegree[b] = 0, indegree[c] = 1

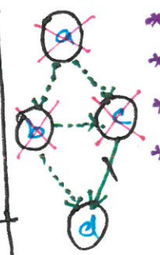
V	1	2	3	4
a	0	0	0	0
b	1	0		
c	2	1		
d	2	2		
Enqueue : ab				
Dequeue : ab				



* indegree[b] = 0
* Enqueue : b
* Dequeue b → list
* Remove adj. edges of b → c, d.

indegree[c] = 0, indegree[d] = 1

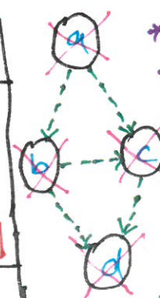
V	1	2	3	4
a	0	0	0	0
b	1	0	0	0
c	2	1	0	
d	2	2	1	
Enqueue : abc				
Dequeue : abc				



* indegree[c] = 0
* Enqueue : c
* Dequeue : c → list
* Remove adj. edges of c → d.

indegree[d] = 0

V	1	2	3	4
a	0	0	0	0
b	1	0	0	0
c	2	1	0	0
d	2	2	1	0
Enqueue : abcd				
Dequeue : abcd				



* Enqueue : c
* Dequeue - c → list
* No adj. edges for d.

Sorted → abcd

TOPIC-18 SHORTEST PATH ALGORITHMS - UNWEIGHTED SHORTEST PATH - DIJKSTRA'S ALGORITHM

Shortest Path Algorithms:

→ Finds minimum cost from Source to other vertex.

Types:

→ Single Source Shortest Path (SSSP)
→ All Pair Shortest Path (APSP)
  ↳ Shortest Path between all pair of vertices.
  ↳ Floyd's warshall, Johnsons.

Single Source Shortest Path

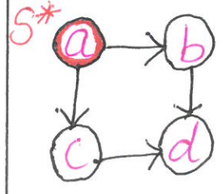
↳ Unweighted

Known → Visited - known (1)
Not visited - Unknown (0)
dv → Distance from Source (initially ∞)
Edge → 1, No Edge → 0
Pv → Actual Path.

Steps:

1. Source Node - 'S', Enqueue
2. Dequeue 'S' → known as 1 and find it adjacency vertex.
3. Distance, dv → Source vertex distance increment by 1 & Enqueue the vertex.
4. Repeat from step ② until queue becomes empty.

Example:

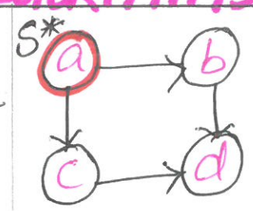


a → Source Node.

Initial configuration

V	Known	dv	Pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

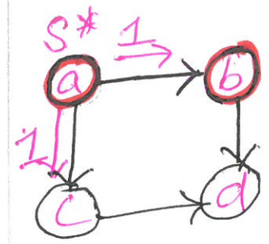
Enqueue :
Dequeue :



V	Known	dv	Pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Enqueue : a
Dequeue : a

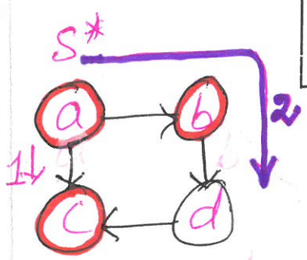
a → known (1)
Enqueue : a
Dequeue : a



V	Known	dv	Pv
a	1	0	0
b	1	1	a
c	0	1	a
d	0	0	0

Enqueue : b c
Dequeue : a b

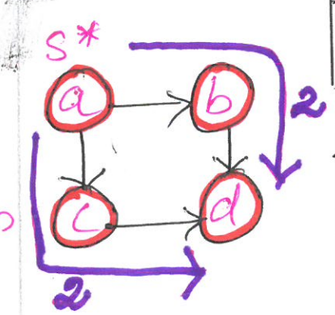
dv(b) = 1
dv(c) = 1
Enqueue : b, c
Dequeue : b



V	Known	dv	Pv
a	1	0	0
b	1	1	a
c	1	1	a
d	0	2	b

Enqueue : c, d
Dequeue : a, b, c

dv(a)
a → b → d (2)
a → c (1)



V	Known	dv	Pv
a	1	0	0
b	1	1	a
c	1	1	a
d	1	2	b

Enqueue : d
Dequeue : a b c d

dv ⇒
a → b → d (2)
a → c → d (2)
Same distance

V	Known	dv	Pv
a	1	0	0
b	1	1	a
c	1	1	a
d	1	2	b

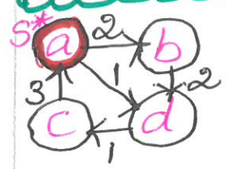
Queue : Empty

Shortest path from Source vertex 'a'
a → b = 1
a → c = 1
a → d = 2

Dijkstra's Algorithm (Weighted)

→ Single Source shortest path algorithm
→ Similar to unweighted SSSPA, needs to calculate the distance using weights.

Example:

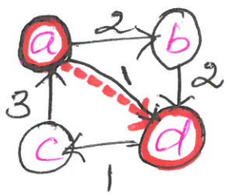


Initial configuration

V	Known	dv	Pv
a	0 →	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Enqueue : a
Dequeue : a

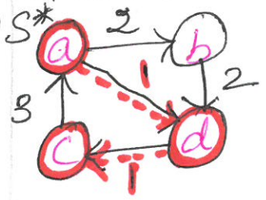
a → known (1)



a → b = 2
a → d = 1
Shortest path
Enqueue : d
Dequeue : d
(Path from a)

V	Known	dv	Pv
a	1	0	0
b	0	2	a
c	0	∞	0
d	1	1	a

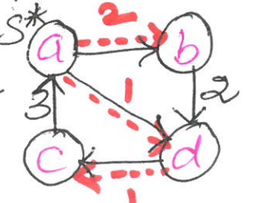
Enqueue : b d
Dequeue : a d



a → d → c = 2
Enqueue : c
Dequeue : c
Path from d

V	Known	dv	Pv
a	1	0	0
b	0	2	a
c	1	2	d
d	1	1	a

Enqueue : c
Dequeue : a d c



a → b = 2
Enqueue : b
Dequeue : b

V	Known	dv	Pv
a	1	0	0
b	1	2	a
c	1	2	d
d	1	1	a

Enqueue : b
Dequeue : a b c b.

Shortest Path from Source Vertex 'a' : a → b = 2, a → c = 2, a → d = 1

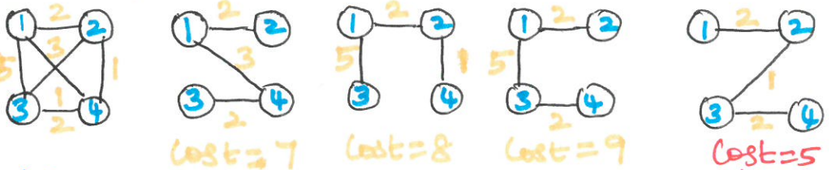
Minimum Spanning Tree (MST)

Spanning Tree - Connected Graph, & No cycles
Subgraph with all Vertices

Minimum Spanning Tree - Spanning tree with minimum cost.

n^{n-2} Spanning tree will obtain for n nodes/ Vertex.

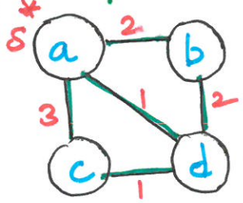
Example: 4 Nodes [$n^{n-2} = 4^2 = 16$ Spanning trees]



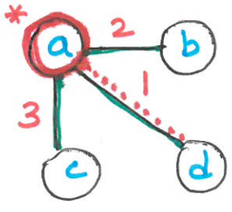
Algorithms:
- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm:
↳ Uses Greedy Techniques
↳ Based on Vertices.

Example:

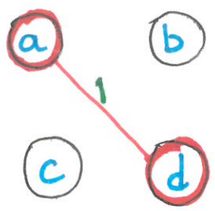


V	KNOWN	dv	Pv
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

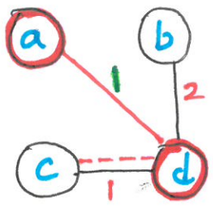


V	KNOWN	dv	Pv
a	1	0	0
b	0	2	a
c	0	3	a
d	0	1	a

a → d (1) minimum distance
d → KNOWN (1), path from 'a'

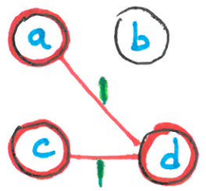


V	KNOWN	dv	Pv
a	1	0	0
b	0	2	a
c	0	3	a
d	1	1	a



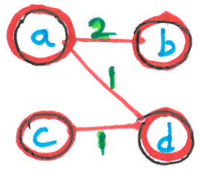
V	KNOWN	dv	Pv
a	1	0	0
b	0	2	a
c	0	1	d
d	1	1	a

adjacent Vertices
d → b, c
d → c = 1
d → b = 2
(min. cost)



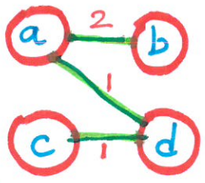
V	KNOWN	dv	Pv
a	1	0	0
b	0	2	a
c	1	1	d
d	1	1	a

C → KNOWN
Path from 'd'



V	KNOWN	dv	Pv
a	1	0	0
b	1	2	a
c	1	1	d
d	1	1	a

Remaining Vertex 'b',
a → b = 2,
d → b = 2
(Same)



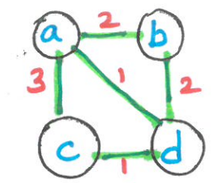
Minimum Spanning Tree
Minimum Cost:

$C_{a,b} + C_{a,d} + C_{c,d} = 2+1+1 = 4$

Kruskal's Algorithms:
↳ Uses Greedy Techniques.
↳ Based on Edges.

Steps:
1) Remove all loops & parallel edges (High cost)
2) Arrange all edges in Ascending Order.
3) Add the edges which has the least cost.

Example:



Edge	dv
(a,b)	2
(a,c)	3
(a,d)	1
(b,d)	2
(c,d)	1

⇒

Edge	dv
(a,d)	1
(c,d)	1
(a,b)	2
(b,d)	2
(a,c)	3

Sort edges in Ascending order

⇐

Edge	dv	Visit
(a,d)	1	✓
(c,d)	1	✓
(a,d)	1	✓
(b,d)	2	x
(a,c)	3	x

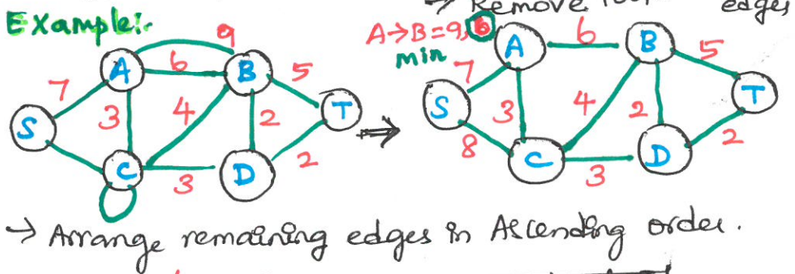
cycle (bdab)
cycle (adca)

Minimum Spanning Tree
Minimum Cost:

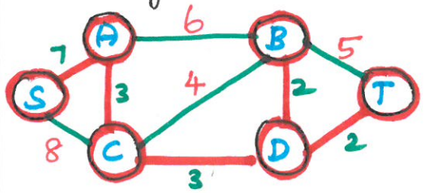
$C_{a,b} + C_{a,d} + C_{c,d} = 2+1+1 = 4$

→ Remove loops & Parallel edges

Example:

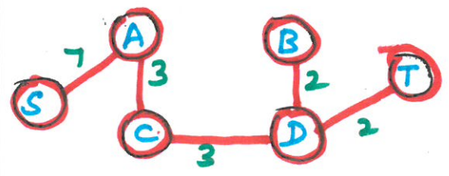


→ Arrange remaining edges in Ascending order.



Edge	dv	Visit
(B,D)	2	✓
(D,T)	2	✓
(A,C)	3	✓
(C,D)	3	✓
(B,C)	4	✓
(B,T)	5	X
(A,B)	6	X
(S,A)	7	✓
(S,C)	8	X

cycle (BDTB)
cycle (ACDBA)
cycle (SACS)



Minimum Spanning Tree
Minimum Cost: $C_{S,A} + C_{A,C} + C_{C,D} + C_{D,B} + C_{D,T}$
$= 7+3+3+2+2 = 17$

Applications — **Google Maps**

TOPIC-20 (BREADTH FIRST SEARCH, DEPTH FIRST SEARCH)

Breadth First Search (BFS)

* Traverses Bread-wise
* Uses Queue Concept (FIFO)

Applications:

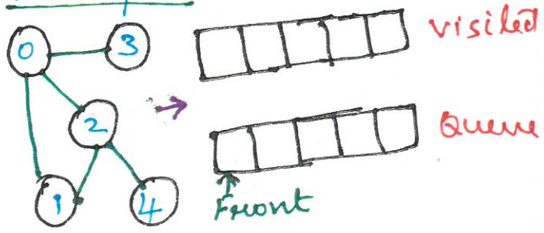
* GPS Navigation
* Path Finding Algorithms
* Minimum Spanning Tree

2 categories - BFS Implementation
↳ Visited & Not visited

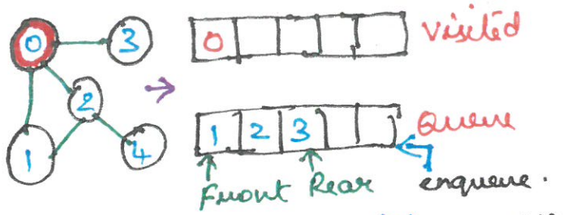
Algorithm steps:

1) Create Queue - size (No of vertices)
2) Enqueue any one (source) vertex - Visited list
3) Enqueue front item from Queue to Visited list. Also, enqueue its adjacent vertices to Queue.
4) Repeat the steps till Queue is empty.

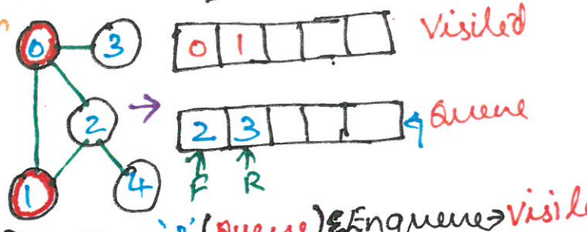
Example:



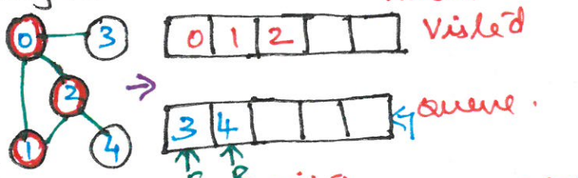
Vertex, 0 → Source.
Enqueue 0 → Visited
Adjacent vertices (0) → 1, 2, 3 Queue



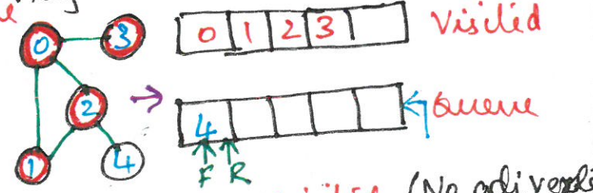
Dequeue front item '1' & move to visited list
Enqueue 1 → visited
Adj. vertices (1) → Queue (0,2)
'0' → already visited



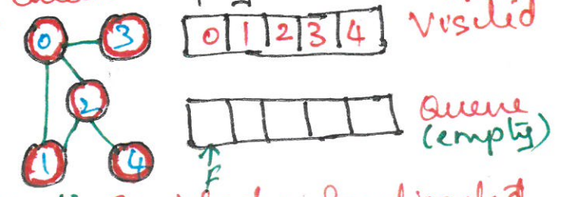
Dequeue '2' (Queue) & Enqueue → Visited
Adjacent vertices (2) → 0, 1, 4 → Queue
visited



Enqueue 3 → Visited
Adjacent vertices (3) → 0 → visited



Enqueue 4 → Visited (No adj. vertex)
Queue Empty.



Note: Similarly, for directed graphs, follow adjacent nodes (outdegree)

Advantage:

* less memory space & time period

Depth First Search (DFS)

* Traverse depthwise (top-bottom)
* Uses Stack concept (LIFO)

Applications:

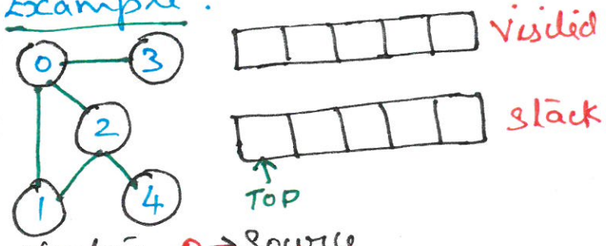
* Topological sorting.
* Detect cycles in the graph.

2 categories - DFS Implementation
↳ Visited & Not Visited

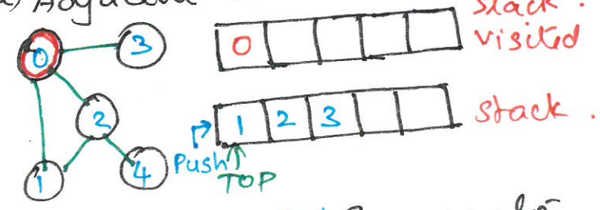
Algorithm steps:

1) Create stack - size (No of vertices)
2) Insert any one (source) vertex - Visited list.
3) Push top item from Stack to visited list. Also, push its adjacent vertices to Stack.
4) Repeat the steps till Stack is empty.

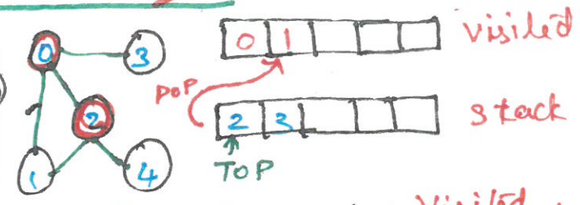
Example:



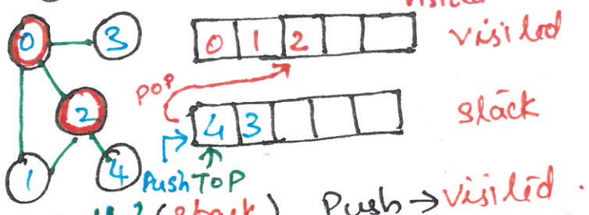
Vertex, 0 → Source
Insert 0 → Visited
Adjacent vertices (0) → 1, 2, 3 Stack.



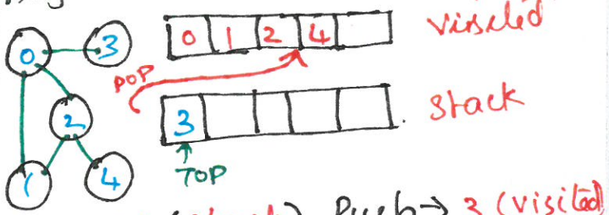
Pop Top item '1' & move to Visited list.
Adj. vertices (1) → Stack (0,2) visited



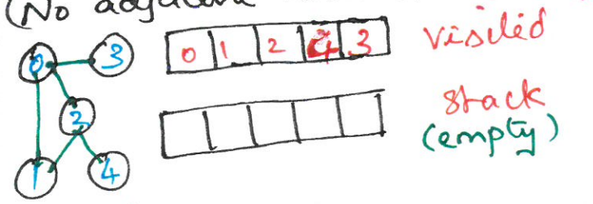
Pop '2' (stack), Push → Visited.
Adjacent vertices (2) → 0, 1, 4 → Stack visited



Pop '4' (stack), Push → visited.
Adjacent vertices (4) → 2 (visited)



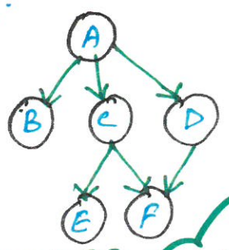
Pop '3' (stack), Push → 3 (visited)
(No adjacent vertices). stack empty



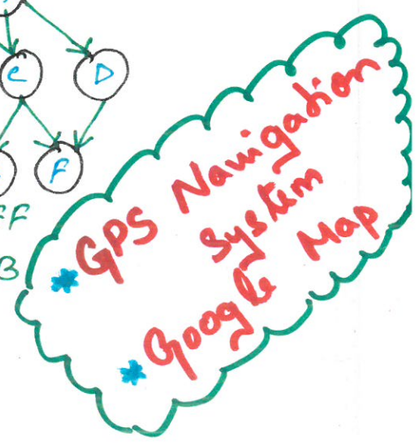
Advantage:

* solutions will definitely found

Activity:



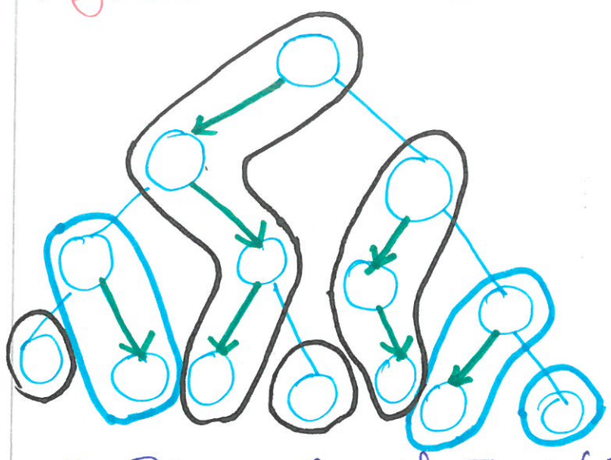
BFS : ABCDEF
DFS : ADFCEB



Advanced Data Structures,

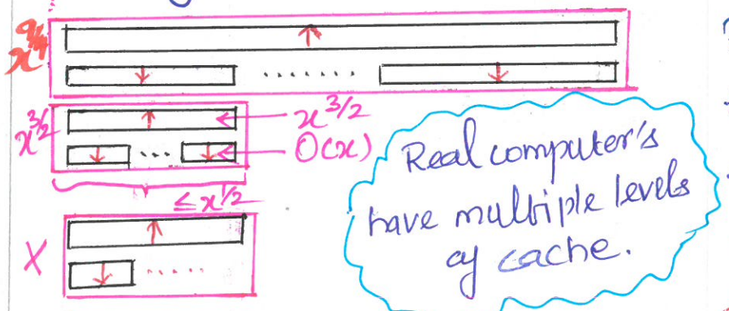
TOPIC - 21

Dynamic optimality:



* Binary Search Tree (BST) that's as good as all others? we still don't know, but we're close.

Memory Hierarchy



* We can optimize the number of cache misses, often without even knowing the size of the cache.

Future of Data Structure

* Problems in handling data
* Never-ending growth of data
* Increased demand for Data Structure.

* Emerging Sources of data.

Research Area in Data structure

* MATLAB experience in accelerating Direct-Gice Algorithm for Constrained global optimization through dynamic data structure and parallelization.
* K-long DNA Sequences, Data Structure.
* A data-driven analysis on bridging techniques for heterogeneous materials and structures.
* Data Structure in Protine - RNA interaction Structure.
* Parallel Algorithms in cached shared memory.
* Sensor data aquisition and smart maintenance using DS.
* "Big Data" processing algorithms for organizing and searching and deducting connection's in vast data collection.
* "Numerical" Algorithm for accurate and stable solution of differential equations, especially in parallel.
* Real time command and control Algorithms for vehicles and industrial processes.

Real Time Applications:

Linked List

* Image viewer software (Single)
* Train Coaches (Doubly)
* Escalator's (Circular)

Stack

* Converting infix - postfix expression
* History of visited website.
* call Logs
* Recursions
* Media play lists
* Java virtual Machine
* Pile of Dinner Plates
* Stacked chairs.

Queue

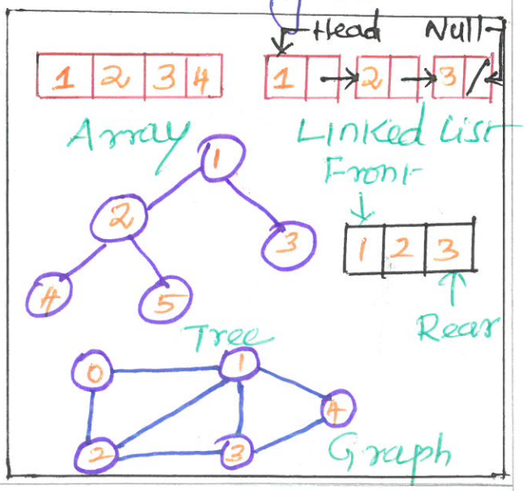
* Operating systems - Job scheduling.
* CPU scheduling
* Escalator's
* Printer Spooler
* Handle website traffic
* Sending email
* Ticket window
* Vehicles - Toll - tax badge
* Networking - Routers and Switches

Trees

* Domain Name server
* Data Base
* Computer Graphics (BST)
* Moves - Chess Game
* Quora (posting Ques)

Graphs:

* Google's Knowledge Graph
* faceBook's Graph API
* Gps Navigation System
* FB, Instagram....
* Shortest path
* D-Game Engine
* Data Analysis (AVL)
* Data Mining (AVL)
* Dictionary App (TRIE)
* Airline Scheduling
* Sudoku's puzzles
* Search Engine



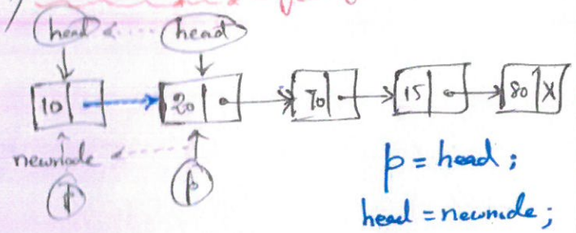
ANALYTICAL CONCEPT MAP

Linked List - Stack - Queue Data Structure

Linked List operations:

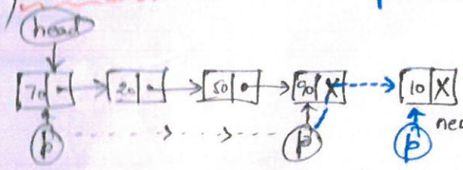
- Insertion – at, Begin, End & Any Position.
- Deletion – at, Begin, End & Any Position.

Q1) Insertion at Begining:



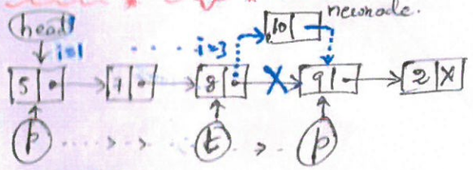

```
p = head;
head = newnode;
```

Q2) Insert at End



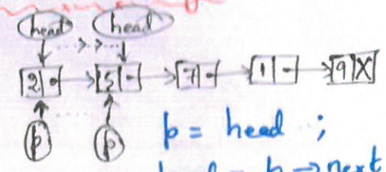

```
newnode -> next = p;
p = newnode;
for(p=head; p->next != NULL; p=p->next);
p->next = newnode;
p = newnode;
```

Q3) Insert at Any Position Let pos = 4



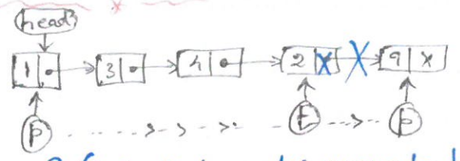

```
for(p=head, i=1; i<pos; p=p->next, i++)
    t=p;
t->next = newnode;
newnode->next = p;
```

Q4) Delete at Begin:



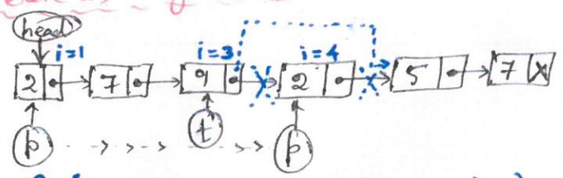

```
p = head;
head = p->next;
free(p);
p = head;
```

Q5) Delete at End:



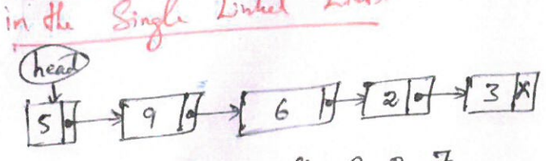

```
for(p=head; p->next != NULL; p=p->next)
    t=p;
t->next = NULL;
free(p);
```

Q6) Delete at Any Position: Let pos = 4




```
for(p=head, i=1; i<pos; p=p->next, i++)
    t=p;
t->next = p->next;
free(p);
```

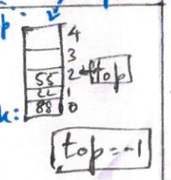
Q7) Find difference between maximum & Minimum in the Single Linked List.



max = 9 min = 2 diff = 9 - 2 = 7

Q8) Searching key=2 ; print "Found"
key=10 ; print "NOT Found"

Q9) Stack operations:



- **push:** `top = top+1; stk[top] = ele;`
- **pop:** `ele = stk[top]; top = top-1;`
- **Peek:** `print "stk[top]";`
- **Display:** `for( i=top ; i>=0; i--) printf("%d", stk[i]);`

```
push(88);
push(22);
push(55);
push(11);
push(77);
pop();
pop();
```

Stack Applications: Infix to Postfix Conversion

Q10) a ← (b/c + (d%e * f)/g) * h

Infix Character Scanned	Stack	Postfix Expression
a		a
-	-	a
(	-(	a
b	-(	ab
/	-(/	ab
c	-(/	abc
+	-(+	abc/
(	-(+(	abc/
d	-(+(	abc/d
%	-(+(%	abc/de
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
over		abc/def*%g/+h**

Use priority of operators
/ * % → 2
+ - → 1

Practice Coding.

Q11) Result : abc/def*%g/+h**

Post fix Expression Evaluation:

I) 8493/*+ II) 84+931/*

Scanning character	Stack operation
8	8
4	8 4
9	8 4 9
3	8 4 9 3
/	8 4 9/3 → 9/3=3
*	8 4*3 → 4*3=12
+	8+12 → 8+12=20

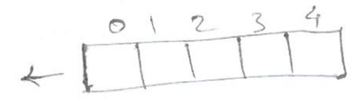
Result = 20

Scanning	Stack operation
8	8
4	8 4
+	12 → 8+4=12
9	12 9
3	12 9 3
/	12 3 → 9/3=3
*	36 → 12*3=36

Result = 36

Queue Data Structure

Size = 5



Front = Rear = -1

Q13) Illustrate the positions:

Operations	Front	Rear
Initially :	-1	-1
Enqueue (22)	0	0
Enqueue (77)	0	1
Enqueue (99)	0	2
Dequeue()	1	2
Enqueue (55)	1	3
Enqueue (33)	1	4
Enqueue (25)	1	4 OVERFLOW
Dequeue()	2	4
Dequeue()	3	4
Dequeue()	4	4
Dequeue()	-1	-1

Enqueue operation:

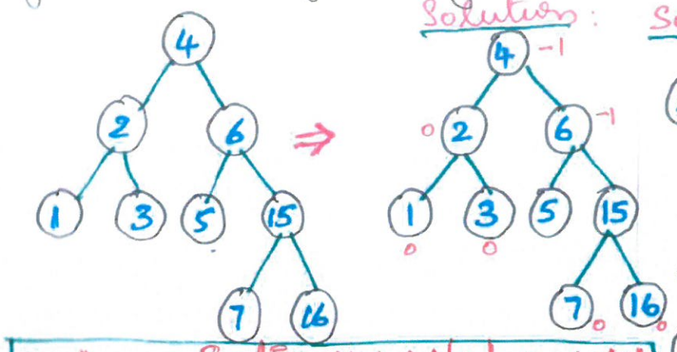
```
if (size-1 == Rear)
    printf("Queue is OverFlow");
else
    if (Rear == -1 && Front == -1)
    {
        Rear = 0;
        Front = 0;
    }
    else
        Rear = Rear + 1;
```

Dequeue operation:

```
if (Rear == -1 && Front == -1)
    printf("Queue is Empty / UnderFlow");
else if (Front == Rear)
    {
        Front = -1;
        Rear = -1;
    }
    else
        Front = Front + 1;
```

PROBLEMS - TREES

Q1. Calculate the Balance factor of each node for the given DS.

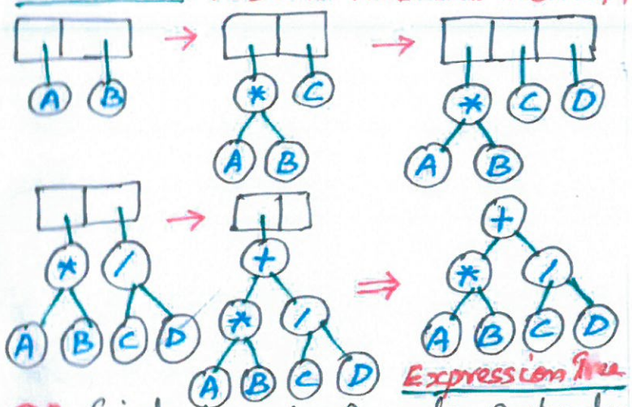


Solution:

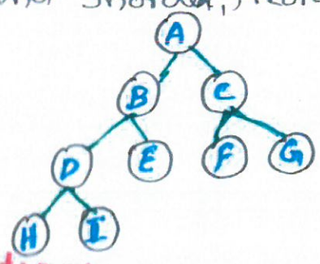
Balance Factor = Height[L] - Height[R]

Q2. Draw a Binary Tree for expression A*B+C/D.

Solution: Postfix Notation: AB*CD/+



Q3. Find Inorder, Preorder, Postorder



Solution:

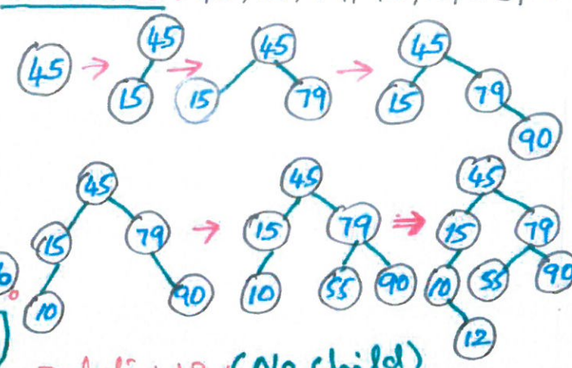
INORDER: Left - Root - Right
HDIBEAFCG

PREORDER: Root - Left - Right
ABDHIECFG

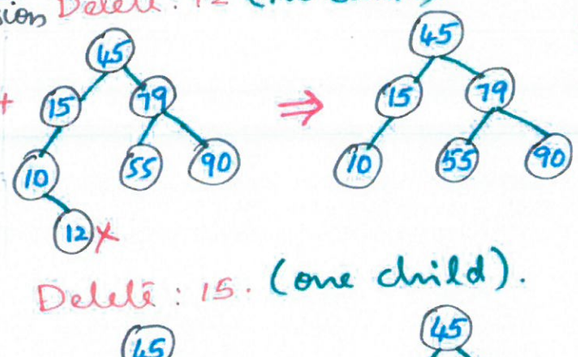
POSTORDER: Left - Right - Root
HIDEBFGCA

Q4. Construct Binary Search Tree 45, 15, 79, 90, 10, 55, 12. Show step-wise deletion of 12, 15 and 79.

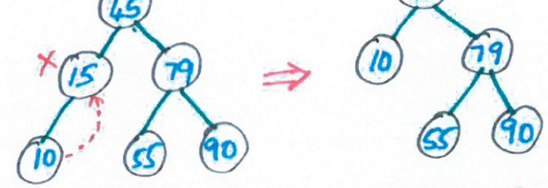
Solution: 45, 15, 79, 90, 10, 55, 12



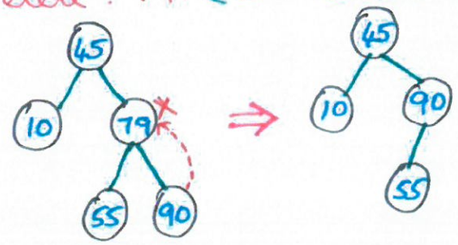
Delete: 12 (No child)



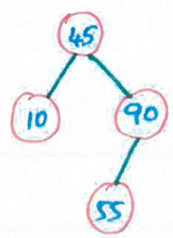
Delete: 15. (one child).



Delete: 79. (Two children)

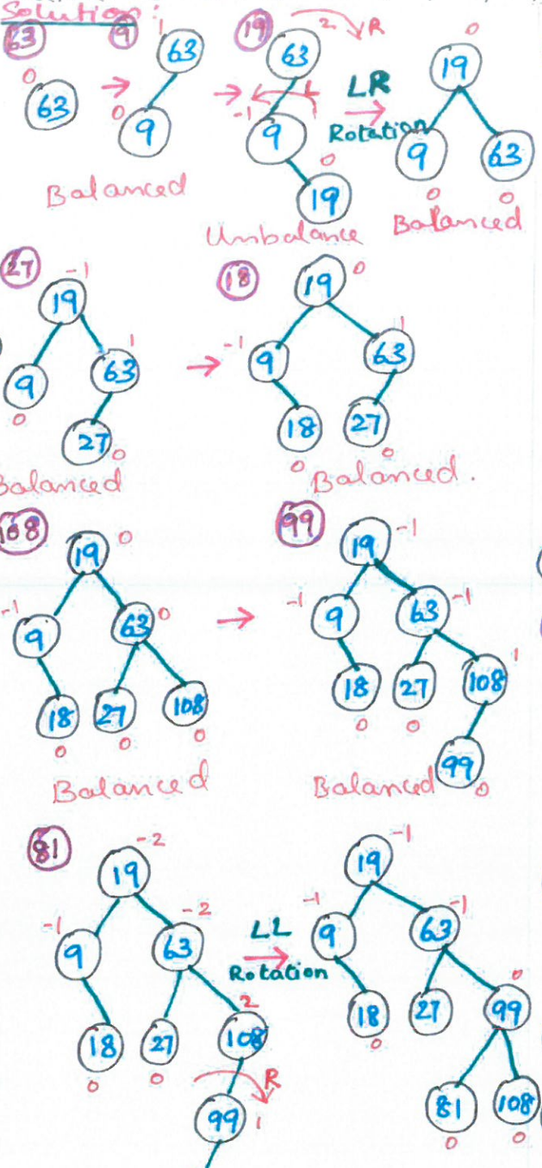


Solution:



Q5. Construct AVL Tree for the elements. 63, 9, 19, 27, 18, 108, 99, 81.

Solution:



Balanced

Balanced

Unbalance

Balanced

Balanced.

Balanced

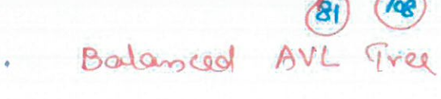
Balanced

Balanced

Balanced.

Unbalanced

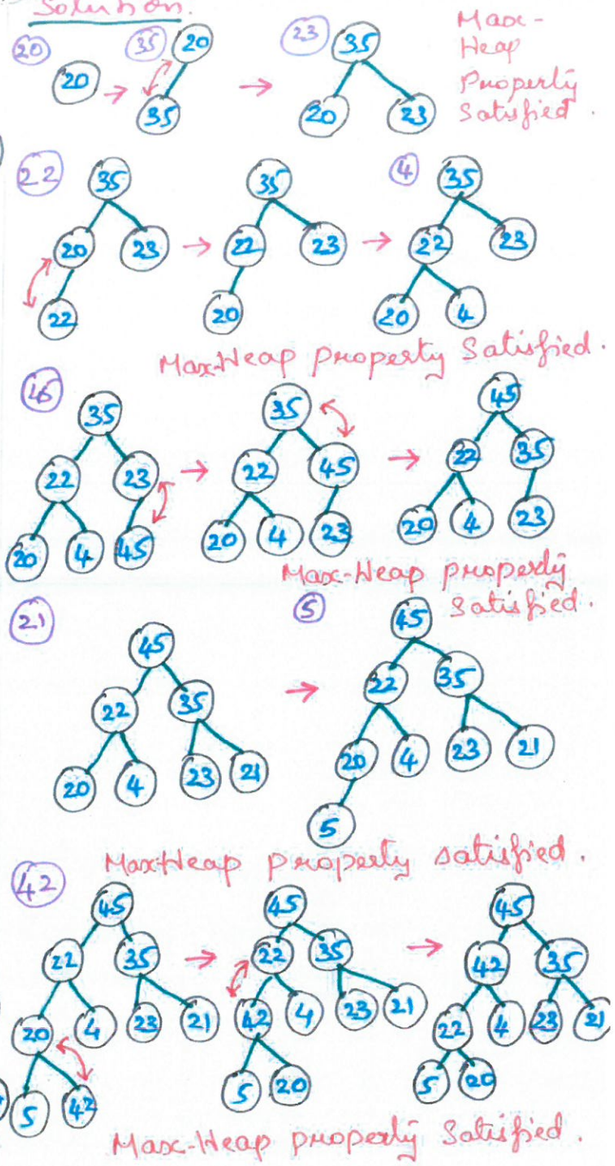
Solution:



Balanced AVL Tree.

Q6. Build Max Heap for elements 20, 35, 23, 22, 4, 45, 21, 5, 42.

Solution:



Max-Heap Property Satisfied.

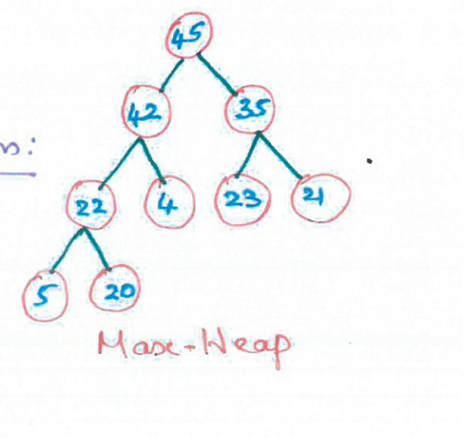
Max-Heap property Satisfied.

Max-Heap property Satisfied.

Max-Heap property Satisfied.

Max-Heap property Satisfied.

Solution:



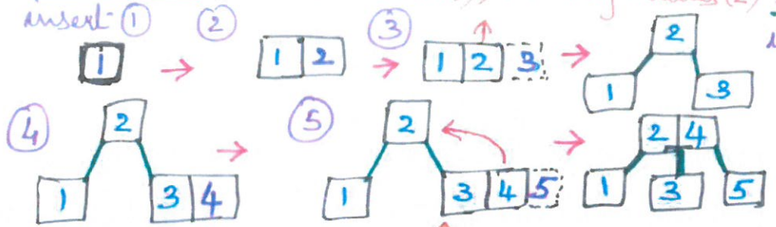
Max-Heap

PROBLEMS - TREES

Q7. Construct a B-Tree of order 3 by inserting numbers from 1 to 8.

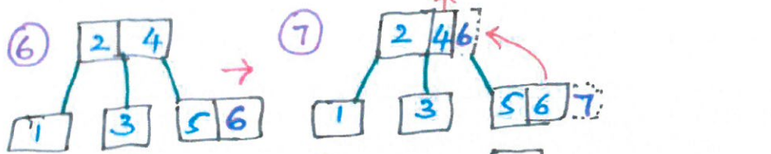
Solution: m=3 (children); m-1 Key values (2)

insert ① ② ③

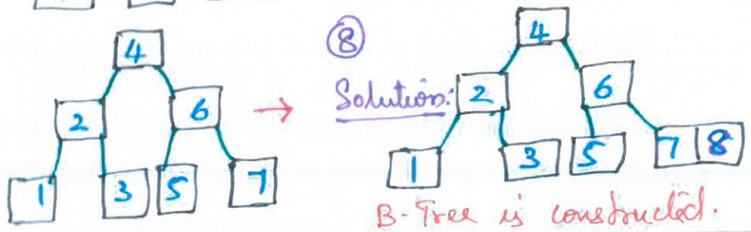


④ ⑤

⑥ ⑦



⑧

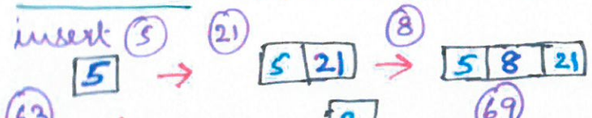


B-Tree is constructed.

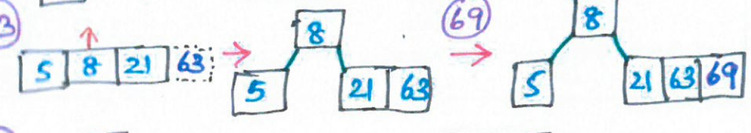
Q8. Construct a 2-3-4 Tree for the sequence of numbers 5, 21, 8, 63, 69, 32, 7, 19, 25.

Solution: m=4 (children) m-1=3 (Key values)

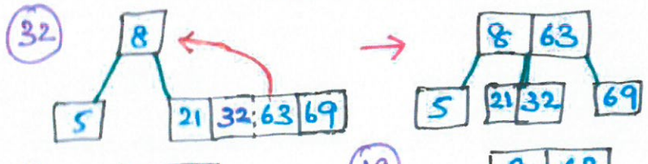
insert ⑤ ㉑ ⑧



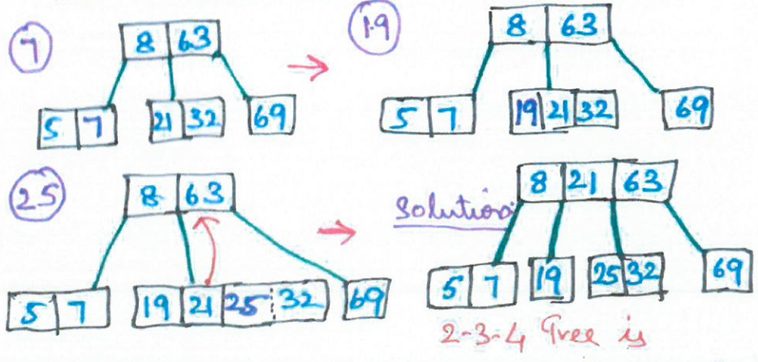
⑥③ ⑥⑨



㉜ ⑦



⑲ ㉕



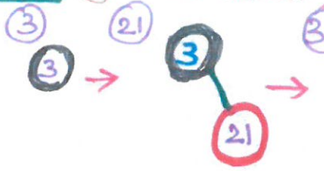
Solution:

2-3-4 Tree is Constructed.

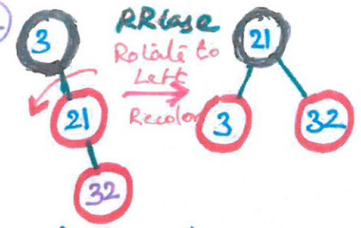
Q9. Construct Red-Black Tree by inserting numbers 3, 21, 32, 15, 40, 25, 10.

Solution: ③ → root (Black); New Node → Red color.

insert ③ ㉑



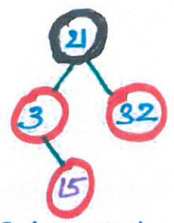
㉜



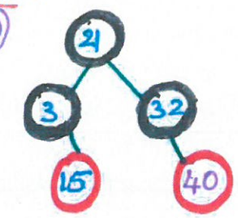
RR case
Rotate to Left
Recolor

21, 32 - Red, New node (32), parents Sibling empty
So rotate & recolor.

⑮

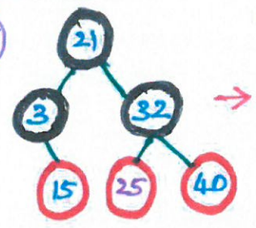


㊵

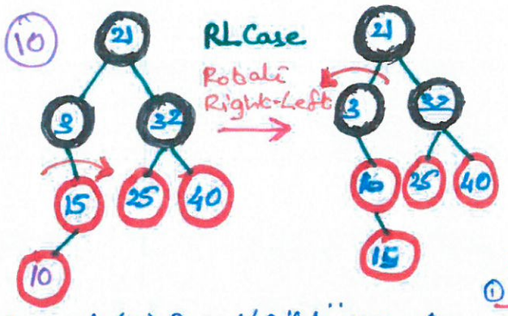


3, 15 - Red, 32 - Red
So recolor (3, 32)

㉕

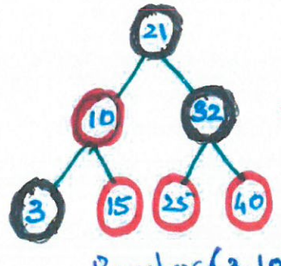


⑩



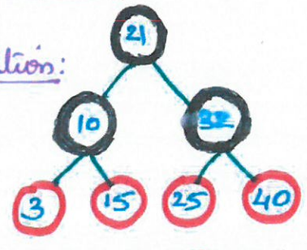
RL Case
Rotate Right-Left

15, 10 - Red, Newnode (10) Parents' Sibling empty.
So rotate & recolor.



Recolor (3, 10)

Solution:



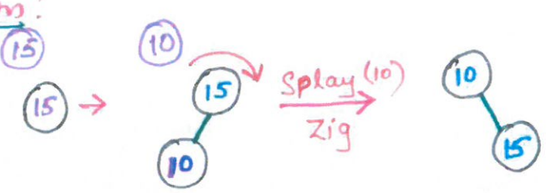
Red Black Tree is Constructed.

Number of black nodes in each path = 2.

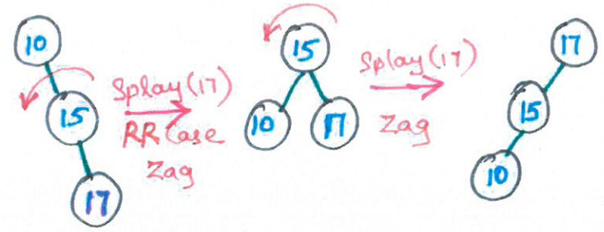
Q10. Construct Splay Tree by inserting the sequence of numbers 15, 10, 17, 7

Solution:

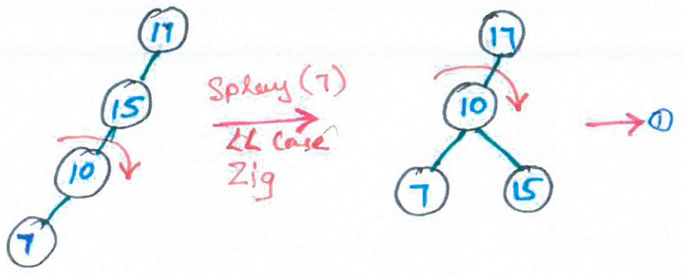
insert ⑮



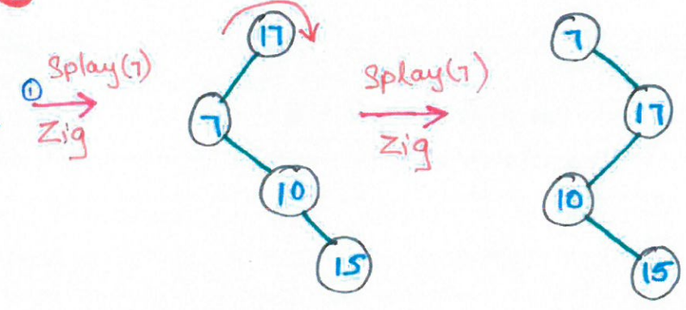
⑰



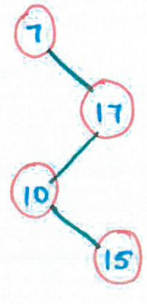
⑦



①



Solution:



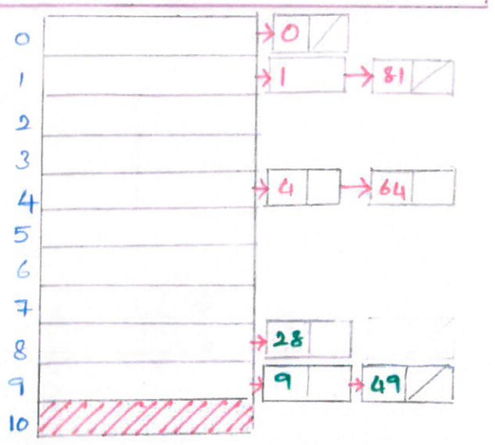
Splay Tree is constructed.

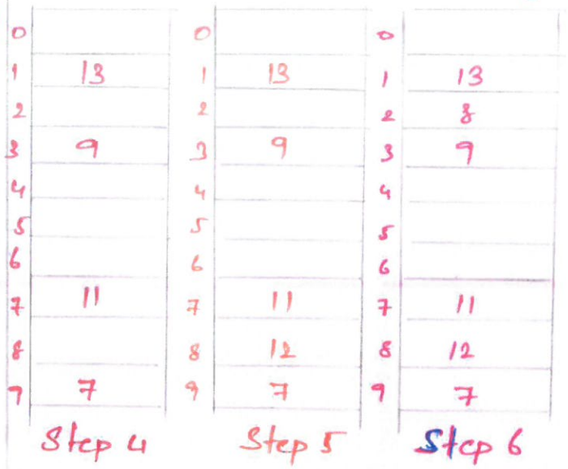
Hashing - Seperate chaining, Hashing - Open Addressing (Linear Probing), Priority Queue - Heap Sort

Hashing (Seperate chaining)

0, 1, 4, 9, 49, 81, 64, 28

Hash(key-value) = (key value MOD Recordsize)



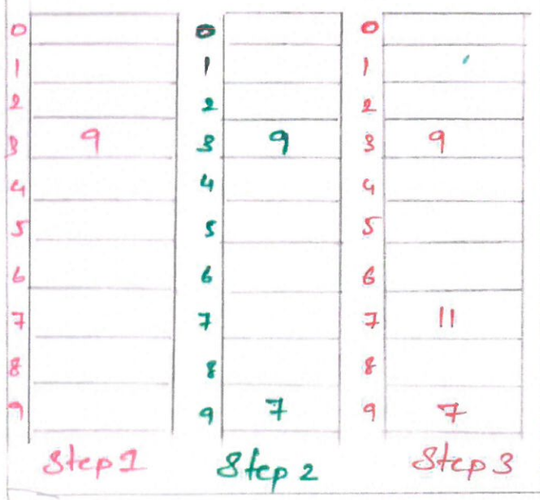


Linear Probing (Open Addressing)

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

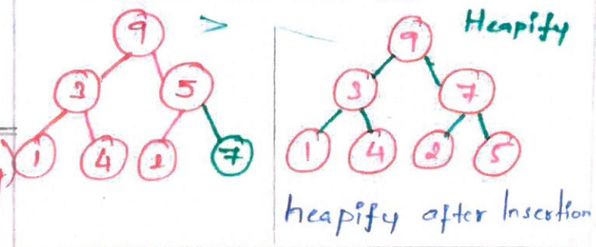
Keys = {9, 7, 11, 13, 12, 8}

Size = 10

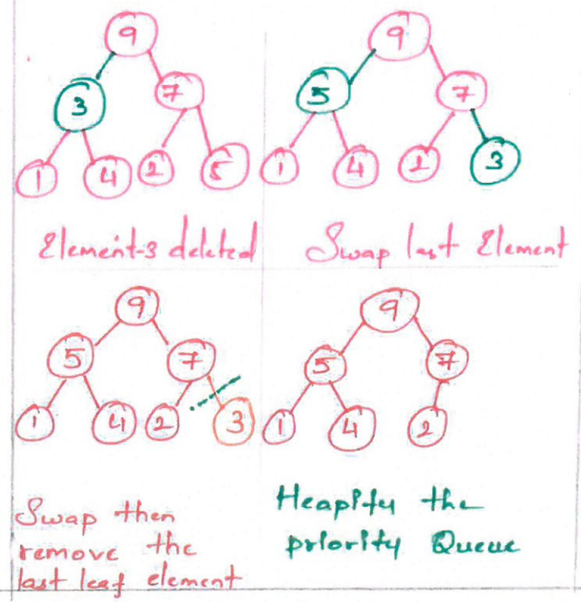


Priority Queue (Max-heap)

Insert element 7



Delete element priority Queue



Heap Sort Problems:

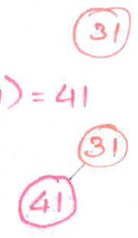
Inserting:

A = | 31 | 41 | 59 | 26 | 58 | 58 | 97 |
(0 1 2 3 4 5 6)

* Highest element must be placed as the root.

Steps:

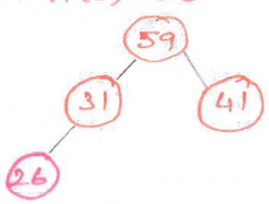
1 → A(0) = 81



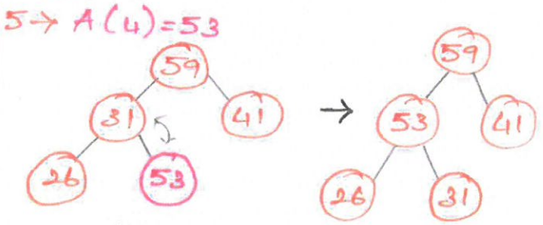
2 → A(1) = 41

3 → A(2) = 59

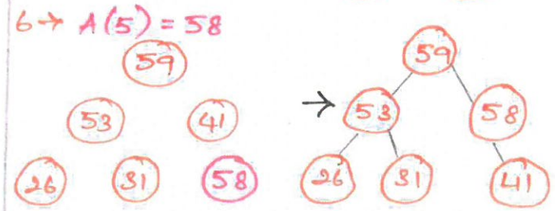
4 → A(3) = 26



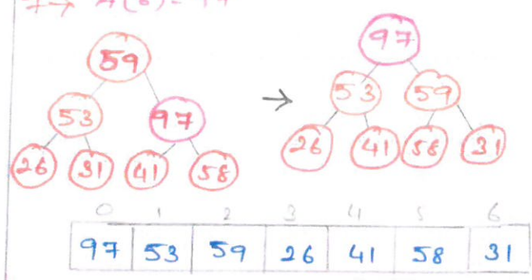
5 → A(4) = 53



6 → A(5) = 58



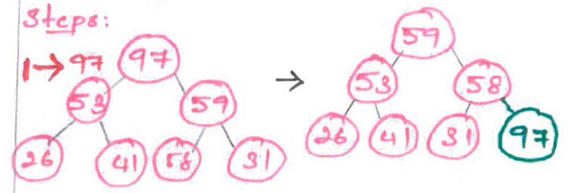
7 → A(6) = 97



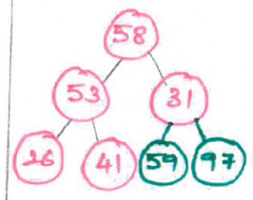
Deleting:

Steps:

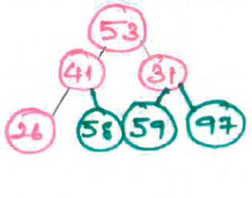
1 → 97



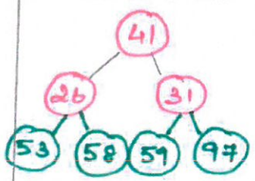
2 → Remove 59



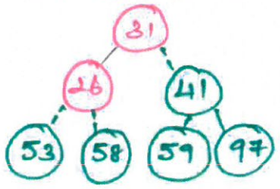
3 → Remove 58



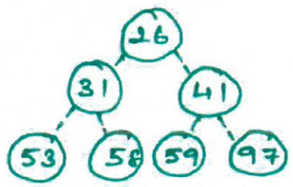
3 → Remove 53



4 → Remove 41



4 → Remove 31



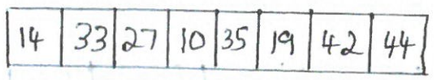
Answer: | 97 | 59 | 58 | 53 | 41 | 31 | 26 |

SORTING

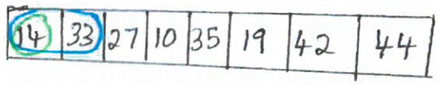
Insertion Sorting :-

* Simple
* Efficient for small data values.

Example : Sort 14, 33, 27, 10, 35, 19, 42, 44

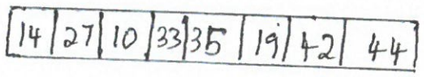


we're take an unsorted array

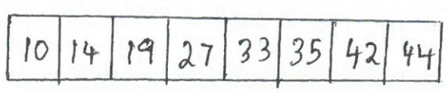


Insertion sort compares 1st two elements.

33 and 27 gets swapped It compares 33 and 10 as next pair.



33 and 10 gets swapped as 33 > 10. This process goes on until all the unsorted values are covered in a sorted sub-list.

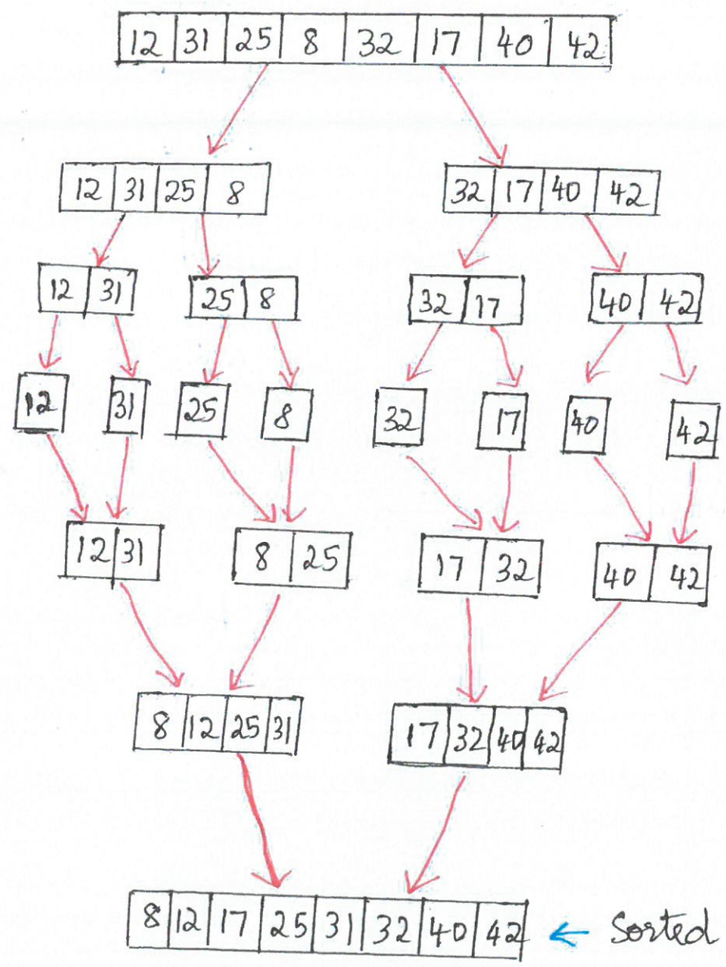


→ Sorted

Merge Sort :-

* Divide and Conquer method
* Recursively dividing the array into 2 halves, sort & then merge

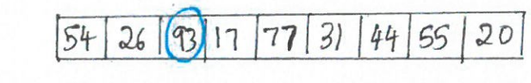
Example : Sort 12, 31, 25, 8, 32, 17, 40, 42

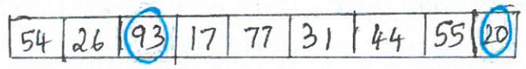


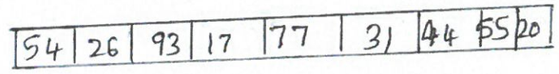
Quick Sort :-

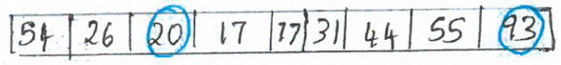
* Divide & conquer method
* choosing a priot element [mean, median, first, Last]

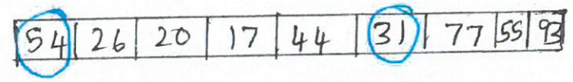
Example : Sort 54, 26, 93, 17, 77, 31, 44, 55, 20

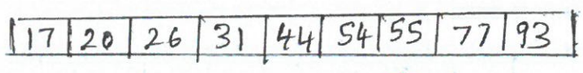












Sorted

∵ Comparing rightmark & leftmark and swapping in ascending order.

Radix Sort :-

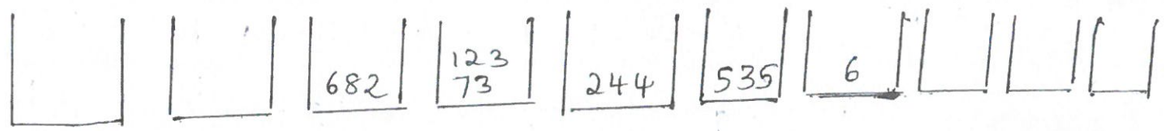
* Algorithm that uses counting sort as a subordinate to sort an array of integers in ascending / descending order.

* The main idea of radix sort revolves around applying counting sort digit by digit on the given array.

* It makes assumption about the data like the data must be between the range of elements.

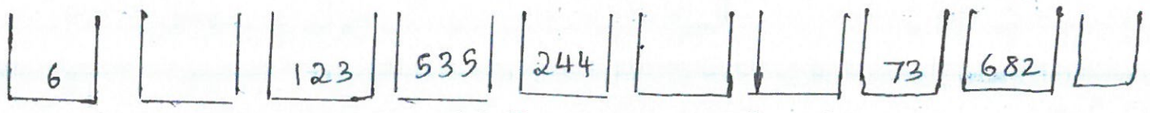
Example : sort 682, 244, 73, 6, 535, 123

Step 1 :-



682, 73, 123, 244, 535, 6

Step 2 :-



6, 123, 535, 244, 73, 682

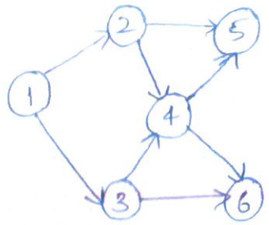
Step 3 :-

6, 73, 123, 244, 535, 682

Hence, the sorted list above is thus the required sorted array

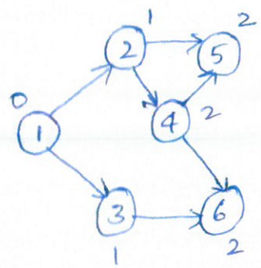
TOPOLOGICAL SORTING, SHORTEST PATH ON WEIGHTED AND UN-WEIGHTED GRAPH

PROBLEM 1: TOPOLOGICAL SORTING



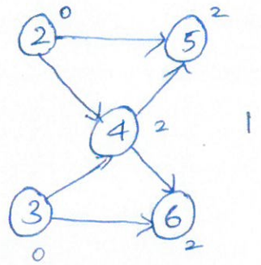
In degree of each vertex

Step 1:



Step 2:

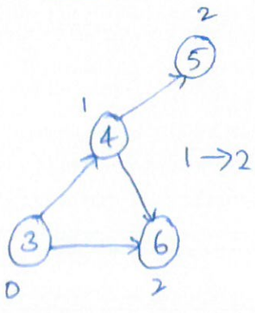
Vertex 1 has least in degree; remove and update 2 degree



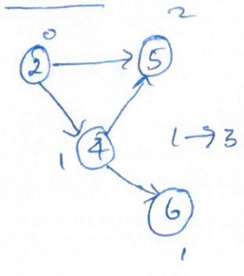
Step 3:

Two vertices are least degree

Case 1

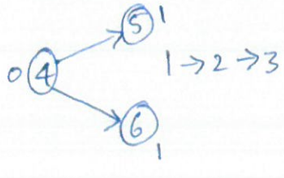


Case 2

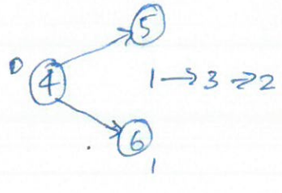


Step 4: Remove vertex 3

Case 1



Case 2:



Step 5: Remove vertex 4

Case 1

$1 \to 2 \to 3 \to 4$ (5)° (6)°

Case 2

$1 \to 3 \to 2 \to 4$ (5)° (6)°

Step 6: Case 1

$1 \to 2 \to 3 \to 4 \to 5 \to 6$

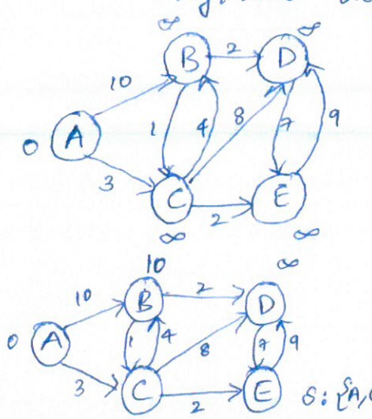
$1 \to 2 \to 3 \to 4 \to 6 \to 5$

Case 2:

$1 \to 3 \to 2 \to 4 \to 5 \to 6$

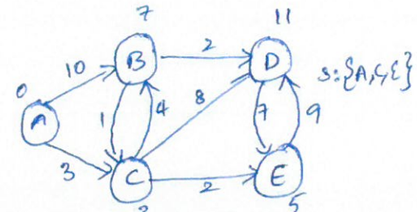
$1 \to 3 \to 2 \to 4 \to 6 \to 5$

PROBLEM 2: Shortest path on weighted Graph (Dijkstra's algorithm)



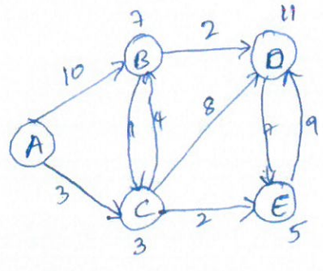
Q:

A	B	C	D	E
0	∞	∞	∞	∞



S: {A, C}

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞



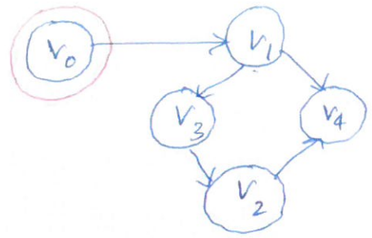
S: {A, C, E, B}

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: {A, C, E, B, D}

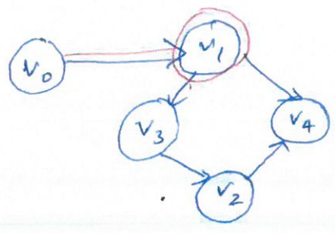
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

PROBLEM 3: Shortest path for unweighted graph

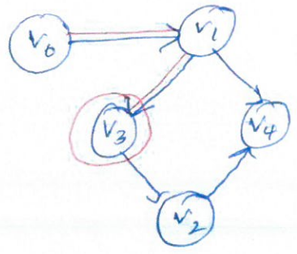


Using DFS

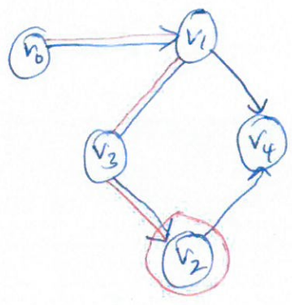
Path to v_1



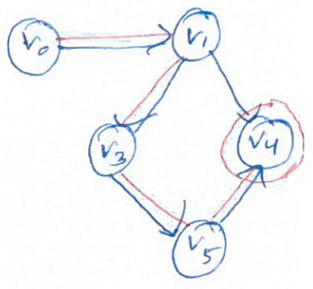
Path to v_3, len 2



Path to v_2, 3



Path to v_4, len 4



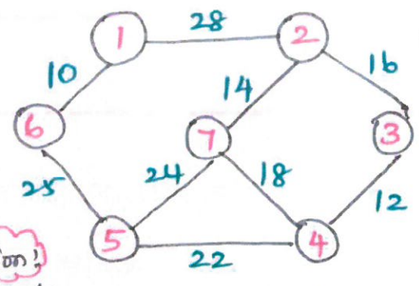
Path

DFS: $V_0 \to V_1 \to V_3 \to V_2 \to V_4$

BFS: $V_0 \to V_1 \to V_3 \to V_4 \to V_2$

PRIM'S, KRUSKAL ALGORITHM, BFS & DFS

Problem:
Construct Minimum Spanning Tree (MST) for the given graph using Prim's Algorithm.

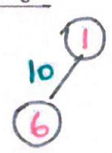


Initial Table:

V	Know	dv	Pv
1	0	0	0
2	0	∝	0
3	0	∝	0
4	0	∝	0
5	0	∝	0
6	0	∝	0
7	0	∝	0

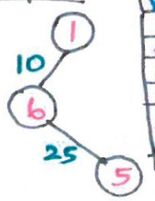
Solution:

Step 1:-



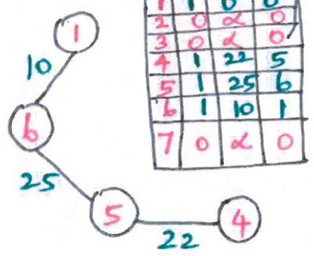
V	Know	dv	Pv
1	1	0	0
2	0	∝	0
3	0	∝	0
4	0	∝	0
5	0	∝	0
6	1	10	1
7	0	∝	0

Step 2:



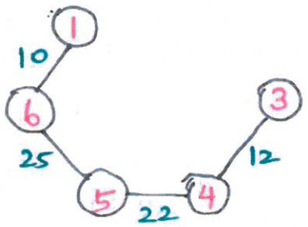
V	K	dv	Pv
1	1	0	0
2	0	∝	0
3	0	∝	0
4	0	∝	0
5	1	25	6
6	1	10	1
7	0	∝	0

Step 3:

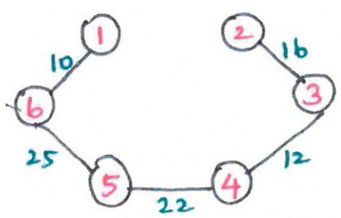


V	K	dv	P
1	1	0	0
2	0	∝	0
3	0	∝	0
4	1	22	5
5	1	25	6
6	1	10	1
7	0	∝	0

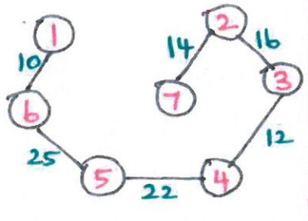
Step 4:



Step 5:



Step 6:

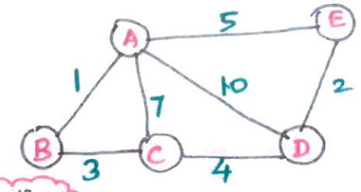


Solution:
All the Vertices included in MST, The cost of MST = Sum of all edge weights

$= 10 + 25 + 22 + 12 + 16 + 14$

Ans = 99 Units

Problem:
Construct Minimum Spanning Tree (MST) for the given graph using Kruskal's Algorithm

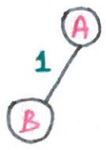


In Sort order:

Edge	dv
(A,B)	1
(E,D)	2
(B,C)	3
(C,D)	4
(A,E)	5
(A,C)	7
A,D	10

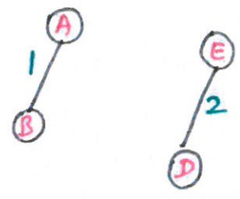
Solution:

Step 1:- Pick edge AB with weight 1



Edge	dv	Visit
A,B	1	✓

Step 2: Pick edge DE with weight 2

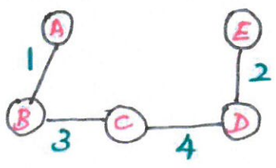


Edge	dv	visit
AB	1	✓
ED	2	✓

Step 3: Add edge BC with weight 3

Edge	dv	visit
AB	1	✓
ED	2	✓
BC	3	✓

Step 4:- Pick edge CD with weight 4



Edge	dv	visit
AB	1	✓
ED	2	✓
BC	3	✓
CD	4	✓
AE	5	× cycle
AC	7	×
AD	10	×

Step 5: Pick edge AE with weight 5 ⇒ forms cyle, discard

Step 6: Pick edge AC with weight 7 ⇒ discard

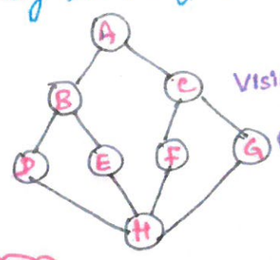
step 7: Pick edge AD with weight 10 ⇒ discard

Minimum cost : = Sum of all edge weights

$= C_{A,B} + C_{E,D} + C_{B,C} + C_{CD}$

Ans = 1+2+3+4 = 10

Problem:
Construct Breadth First Traversal by having the foll. graph of 8 Nodes:

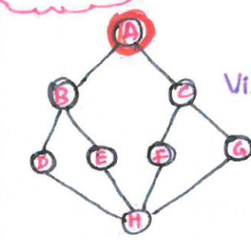


Initial Config

Visited

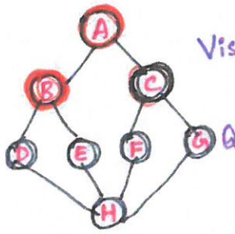
Queue ← EMPTY →

Solution



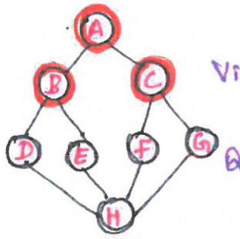
Visited A

Queue B C



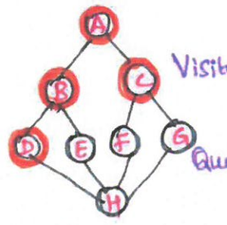
Visited A B

Queue C D E



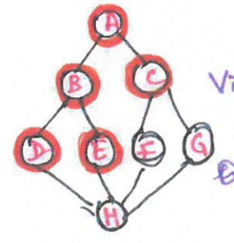
Visited A B C

Queue D E F G



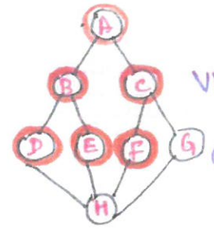
Visited A B C D

Queue E F G H



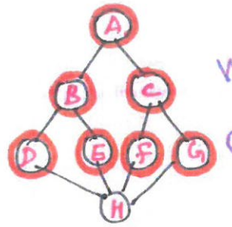
Visited A B C D E

Queue F G H



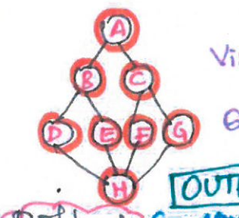
Visited A B C D E F

Queue G H



Visited A B C D E F G

Queue H

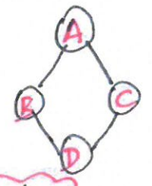


Visited A B C D E F G H

Queue Empty

OUTPUT : A B C D E F G H

Problem: Construct Depth First Traversal for the following graph:

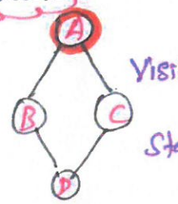


Initial Representation

Visited

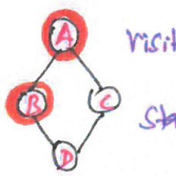
Stack EMPTY

Solution



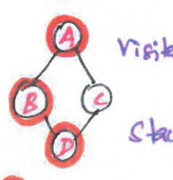
Visited A

Stack B C



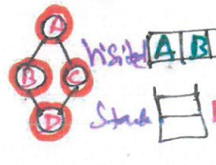
Visited A B

Stack D C



Visited A B D

Stack C



Visited A B D C

Stack EMPTY

OUTPUT: A B D C

SIMATS

SCHOOL OF ENGINEERING

Approved by AICTE | IET-UK Accreditation

Engineer to Excel

Saveetha Nagar, Thandalam, Chennai - 602 105, TamilNadu, India