DATA STRUCTURES

Dr. Uma Priyadarsini P.S
Professor, Deep Learning

TOPIC-1 INTRODUCTION - DATA STRUCTURES

DATA:- a distinct pieces of information

DATA PHRASES:

* Big Data * Data Mining * Data Center * Database.....

DATA OBJECTS:- a region of storage that contains a value or group of values.

ABSTRACT DATA STRUCTURES(ADT)

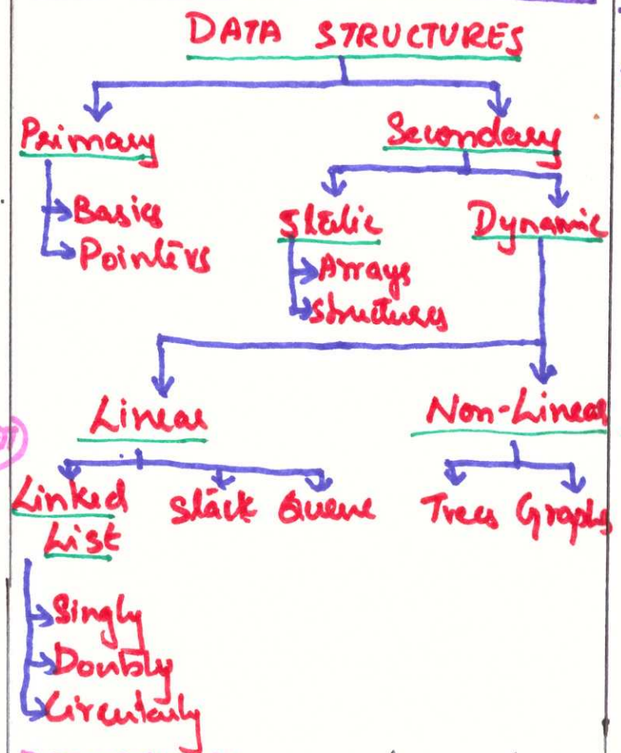
DATA STRUCTURES:- a logical way of organizing data in the memory.

ADT:- a type for objects whose behavior is defined by a set of values and a set of operations.

CLASSIFICATIONS:- Based on applications.

* Static DS - fixed size DS
* Dynamic DS - Variable " "
* Persistent DS - Permanent.
  - Partially Persistent
  - Fully Persistent
  - Confluently Persistent
  - Emphemeral
* Succinct DS - Dictionaries
* Implicit DS - lower bound info.
* Compressed DS - Gif.....
* Search DS - Static & Dynamic
  - Simplest Query.
  - Common Query
  - Multidimensional Space
  - Simultaneous Range Query

ABSTRACT DATA TYPES (ADT)



RECURSION: process of repeating items in a self-similar way.

↳ Recursive Function

↓

eg:- Factorial.

$$Fact(n) = \begin{cases} 1 & , \text{when } n=0. \\ n * fact(n-1) & , \text{when } n>0. \end{cases}$$

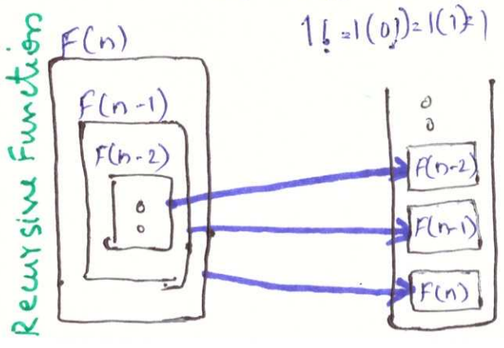
$n! = n \times (n-1) \times (n-2) \times \ldots \times 1.$

$4! = 4(3!) = 4(6) = 24$

$3! = 3(2!) = 3(2) = 6$

$2! = 2(1!) = 2(1) = 2$

$1! = 1(0!) = 1(1) = 1$



REAL TIME APPLICATIONS - LINEAR & NON-LINEAR DS

Linear Data Structures:

- Linked List
  - Image Viewer Software (Singly)
  - Train Coaches (Doubly)
  - Escalators (circular)
- Stack
  - Converting Infix - Postfix Expressions
  - History of visited websites
  - Call logs
  - Recursions
  - Media Playlists
  - Java Virtual Machine
  - Pile of Dinner Plates
  - Stacked chairs.
- Queue
  - Operating Systems - Job Scheduling
  - CPU scheduling
  - Escalators
  - Printer Spooler
  - Handle Website Traffic
  - Sending an Email.

Non-Linear Data Structures:

- Trees
  - Domain Name Server
  - Data Base
  - Compute Graphics (BST)
  - Possible moves in chess Game
  - Query (posting ques. on web)
- Graphs
  - Google's Knowledge Graph
  - FaceBook's Graph API
  - GPS Navigation Systems.
  - Facebook, Instagram.... (User as node)
  - Shortest Paths.
  - D - Game Engine
  - Data Analysis (AVL)
  - Data Mining (AVL)
  - Dictionary App (TRIE)

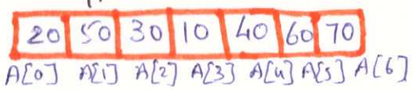
INTRODUCTION - DATA STRUCTURES - CLASSIFICATIONS.

- Data
- Data Objects
- Data Structures
  - Recursion
  - Application Based Types
    - Static Data Structures
    - Dynamic Data Structures
    - Persistent Data Structures
      - Partially Persistent
      - Fully Persistent
      - Confluently Persistent
      - Ephemeral.
    - Succinct Data Structures
    - Implicit Data Structures
    - Compressed Data Structures
    - Search Data Structures
      - Classification of Queries
        - Simplest Query
        - Common Queries
        - Multi-Dimensional Space Queries
        - Simultaneous Range Queries.
  - Abstract Data Types (ADT)
    - Primary Data Structures
      - Basics
      - Pointers
    - Secondary Data Structures
      - Static Data Structures
        - Arrays
        - Structures
      - Dynamic Data Structures
        - Linear Data Structures
          - Linked Lists.
            - Singly
            - Doubly
            - Circular
              - Singly
              - Doubly
            - INSERT
            - DELETE
          - Stack (LIFO)
            - PUSH
            - POP
            - PEEK.
          - Queue (FIFO)
            - ENQUEUE
            - DEQUEUE
        - Non-Linear Data Structures
          - TREES
            - Binary Tree
            - Binary Search Tree
            - Tree Traversal
              - Inorder
              - Preorder
              - Postorder
            - Applications
              - AVL Tree
              - B-Tree (2-3, 2-3-4)
              - TRIE
              - RED BLACK Tree
              - SPLAY Tree.
          - Graphs

TOPIC-2 (ADT ARRAY-OPERATIONS ON ARRAYS- LINKED LIST)

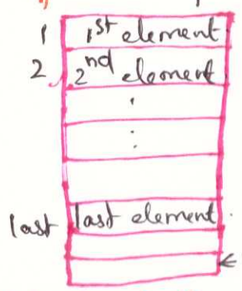
ADT- Arrays.

Array: a variable which can store multiple values of same data type at a time.



Operations:

- Traverse: Print all the elements one by one.
- Insert: Adds an element.
- Delete: Removes an element.
- Search: Search an element.
- Update: updates an element.



***Insert at begining of Array:**

↳ Causes all the existing data items to shift 1 step downward & insert.
↳ Array size - MAX (overflow)
↳ Check if array has empty space to store, if so insert.

***Insert at given index of Array:**

↳ location (index) will be given to store new data.
↳ check if array is full, if not move that data from that location 1 step downward & insert new data.

***Insert at end of Array:**

↳ Check if array is empty (≠MAXSize) NULL if not insert at end.

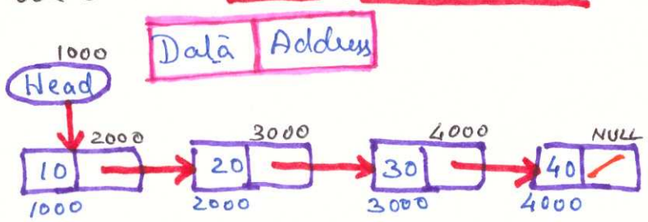
NOTE: We can perform deletion operations similarly.

- Delete at begining
- Delete at given index
- Delete at end.

→ check for underflow condition before deletion.

LINKED LIST: (Singly Linked List)

Collection of nodes connected together. Node contains Data & Address Fields.



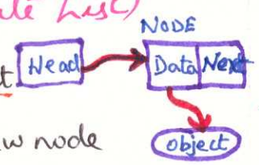
OPERATIONS:

- CREATE
- DISPLAY
- INSERT (Begin, End, Index)
- DELETE (Begin, End, Index)
- SEARCH

→CREATE - Initially NULL.

Add first item (to create list)

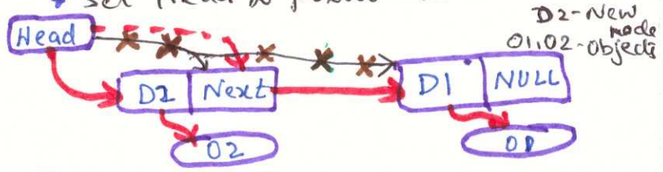
- Allocate space for node
- Set its data pointer to object
- Set Next to NULL
- Set Head to pointer to new node



→INSERT - at Begin:

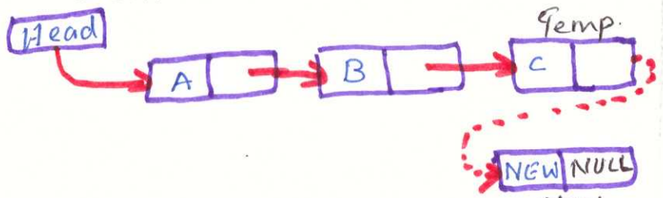
Add second item (to insert new node)

- Allocate space for new node.
- Set its data pointer to object
- Set Next (address) to Head
- Set Head to point the new node.



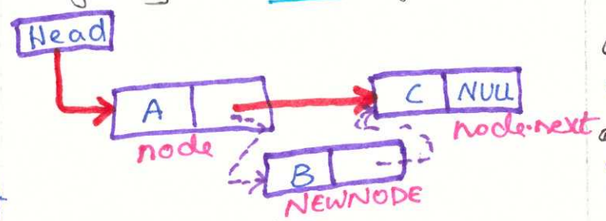
→INSERT - at End:

- Check whether the list is empty, if so assign new node as head.
- If not empty, link the last node (NEXT→NULL) to new node.
- The NEXT of Newnode is set to NULL and insert Data in Data field.



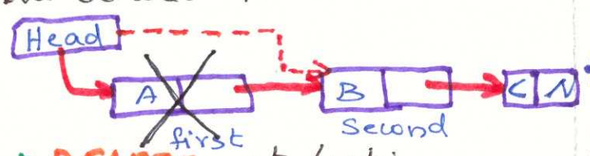
→INSERT - Index (Intermediate)

- Check whether the list is empty, if not get the address of the preceding node after which the new node is to be inserted.
- NEXT field of NEW NODE is made to point to the data field of the next node
- NEXT field of preceding node is made to point to the data field of the NEWNODE by assigning the address of NEWNODE



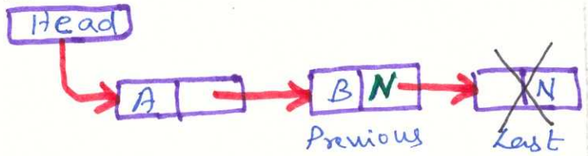
→DELETE - at Begin:

- Check whether the list is empty (head.next⇒NULL) Underflow
- If the list is not empty, set the head.next to the second node in the list (by address)
- Release (free) the memory for the deleted node.



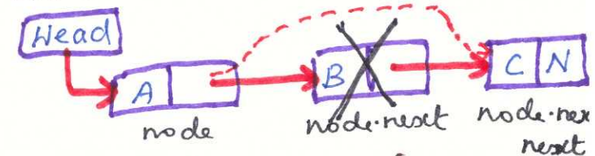
→DELETE - at Last:

- Check whether the list is empty (head.next⇒NULL).
- If the list is not empty, set the next field of the Previous node (before last node of the list to NULL
- Release (free) the memory for the deleted node.



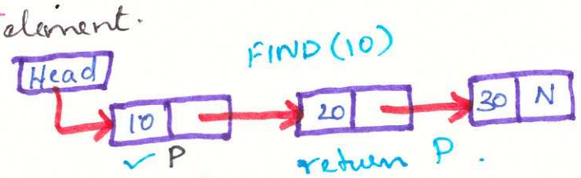
→DELETE - an Intermediate node

- Check whether the list is empty, (head.next⇒NULL)
- If the list is not empty, Set the previous.next node point to data field of the next node
- Release the memory (free) for the deleted node.



→SEARCH - Searching an element using the given key

- Returns the position of searching element.



→DISPLAY

- Check whether list is Empty (Head = NULL)
- If empty, display 'List is empty' & exit
- If not empty, define a Node pointer 'temp' and initialize with head.
- Keep displaying temp→data with an arrow (-->) until temp reaches to the last node.
- Finally display temp→data with arrow pointing to NULL

Note:-
Before insertion, check for Overflow condition (Full or not)
Before deletion, check for Underflow condition (Empty or not)

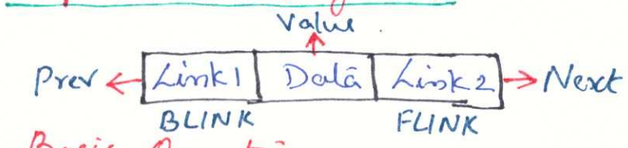
TOPIC-3 (LINKED LIST CLASSIFICATION, SEARCHING ALGORITHM)

LINKED LIST CLASSIFICATION

→Singly Linked List (Linked list)
→Doubly Linked List
→Circular Linked List.

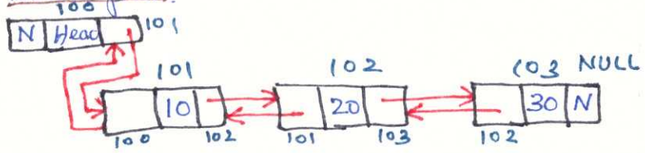
Doubly Linked List: Sequence of elements which has 3 fields:
Link 1 (BLINK): Stores addr. of Prev Node
Link 2 (FLINK): " " " Next "
Data : " actual value.

Representation of Node



Basic Operations
* Insertion (Last, First, Intermediate)
* Deletion (" - ")
* Display

Example:



Advantage:
* Finding Predecessor or Successor of a node is easier.
* Deletion Operation is easier.
* Can allocate & re-allocate memory easily during its execution.

Disadvantage:
* More memory space required since it has 2 pointers.
* Elements are accessed sequentially, so no direct access is allowed.

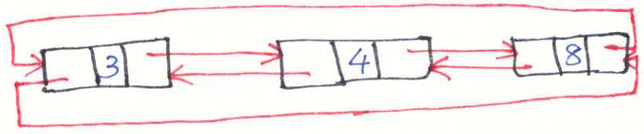
Circular Linked List: Similar to singly linked list except that the last node points to the first node in the list. ie; last node will have the address of first node.
NOTE: No "NULL" terminating pointer.

Basic Operations.
* Insertion
* Deletion
* Display

Types.
→Singly Circular Linked List
→Doubly Circular Linked List

Singly Circular Linked List

Doubly Circular Linked list



Advantage:
* Allows quick access to the first & last records.
* Performs all regular functions of a singly linked list.
* More advantageous for end operations

Disadvantage:
* Complex as compared to Singly/Doubly Linked List
* Harder to find the end of the list.
* Reverse is complex.
* If not handled carefully, then the code may go in infinite loop.

SEARCHING ALGORITHMS

Searching- To check whether a particular element 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 be searched with all the elements present in the array & when the element is matched successfully, it returns the index of the element in array, else returns '-1'

Example: To search an element '6' in the array using Linear Search

i=0, check 8==6, **No**
i=1, check 2==6, **No**
i=2, check 6==6, **Yes** **output= 2**

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

Advantage:
* Simple, easy to understand & implement.
* Not required any order to store the data

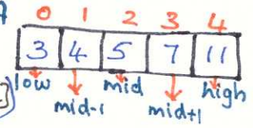
Disadvantage:
* Slow & very poor efficiency.

Time Complexity.
↳ Best Case: O(1)
↳ Average Case: O(n)
↳ Worst Case: O(n)

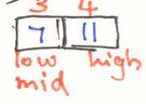
Binary Search:- Searching is done for a sorted array. Uses Divide & Conquer method.

Algorithm steps:
1) Find the middle element of the sorted array.
low=0, high=n-1, mid=(low+high)/2
2) If the search key value is equal to the mid element, then return the index of mid element.
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 side of mid value.
4) If no matches is found, return -1.

Example: key value 'A'



mid=(low+high)/2 = (0+4)/2 = 2 (A[2])
check 7==5 (A[2])
↳ Not same (7>5)



mid=(low+high)/2 = (3+4)/2 ≅ 3
check 7==7 (A[3])
↳ Same (7=7)
Return A[3]
Output=3

Advantage:
* Faster Compared to Linear Search
* No. of Comparison is less.
* More efficient.

Disadvantage:
* Works only on sorted list
* More Complex.

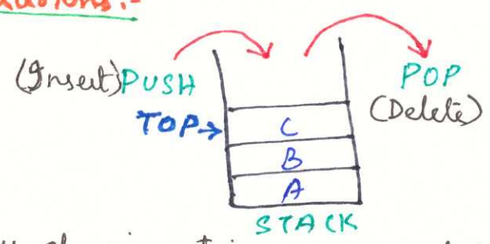
Time Complexity:
↳ O(log n)

TOPIC-4 (STACK)

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) i.e; lastly inserted element will be removed first.

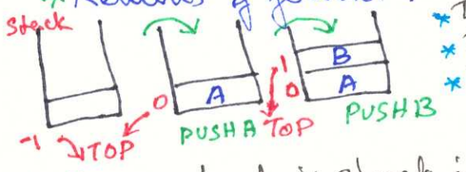
→ **Implementation**:- 2 ways.
- Arrays - fixed size DS.
- Linked list - variable size DS.

→ **Basic Operations**:-
* PUSH
* POP
* PEEK
* IsFULL
* IsEMPTY



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

Steps:
* Check if the stack is full (Overflow Cond)
* If the stack is full, display "stack Full" and exit [Top = MaxSize]
* If not full, increments Top (Top++)
* Adds data element to the location where Top is pointing (stack[TOP] = value)
* Returns if found.



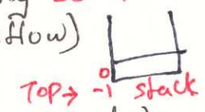
* TOP = -1 => Empty
* TOP = 0 => 1st value
* TOP = MaxSize => FULL

→ **ISFULL** - check if stack is Full (Overflow)

```
if (top == MaxSize) //0-Siz-1, top==Size-1
    returns true;
```

→ **IsEmpty** - check stack is Empty

```
if (top == -1) //Empty. (Underflow)
    returns true;
```



Note:- If the linked list is used to implement stack, we need to allocate space dynamically. Also No need to check for "Overflow Condition"

→ **POP** - Process of removing top element from the stack.

Steps:
* Checks if the stack is empty (Underflow Condition)
* if empty, display "Stack Empty" & exit (Top = -1)
* if not empty, decrements Top value by 1 (Top = Top-1)
* Return.

→ **PEEK** - Returns the value of top most element of stack. without deleting that element from the stack.

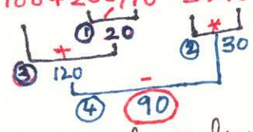
Steps:
* Checks if the stack is empty (Underflow Condition)
* if empty, display "Stack Empty" & exit (Top = -1)
* if not empty, returns the element at which the Top is pointing. (return stack[TOP])

→ **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 Highest	2
+, -	Lowest	1

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



①&② - '/' & '*' both have same precedence but Left to Right (LTR) associativity.
③ - '+' & '-' both have same precedence but Left to Right (LTR) associativity.
'/' and '*' have higher precedence than '+' & '-'.

Algorithm for infix to Postfix Conversion:
1) Scan the 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 operator 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 operator.
* if the precedence of operator is equal, then check the associativity of the operator. If LTR associativity, then pop the operator from stack. If RTL (Right to left) associativity, then put into the stack.
* If the input is '(', push it into stack.
* If the input is ')', pop out all operators until we find '('.

Repeat step 1, 2 & 3 till expression has reach '#'
'#' - Delimiter (End of expression)
4) Now pop all the remaining operators from the stack and push into the output postfix expression
5) Exit.

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

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

Output: ABC*D+E/+

TOPIC-5 (STACK APPLICATIONS)

STACK APPLICATIONS

Evaluating Arithmetic Expression (PostFix)

➤ **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 on to 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
   c) if is '\0' (end) empty the stack.

➤ **Example:** (5+3)*(8-2) ⇒ Infix

1) Convert into postfix expression

5 3 + 8 2 - *

2) Evaluate the postfix expression

READ Symbol	STACK Operations		EVALUATED Part of Expression
Initial	stack-Empty		Nothing
5	Push(5)	5	Nothing
3	Push(3)	3, 5	Nothing
+	Value 1 = PoP() 3 Value 2 = PoP() 5 result = Value2 + Value1 Push (result)	8	Value 1 = 3 Value 2 = 5 result = 5+3 Push(8) (5+3)
8	Push (8)	8, 8	(5+3)
2	Push (2)	2, 8, 8	(5+3)
-	Value1 = PoP() 2 Value2 = PoP() 8 result = value2 - Value1 Push (result)	6, 8	Value 1 = 2 Value 2 = 8 result = 8-2 Push (6) (8-2) (5+3), (8-2)
*	Value 1 = PoP() 6 Value 2 = PoP() 8 result = Value2 - Value1 Push (result)	48	Value 1 = 6 Value 2 = 8 result = 8*6 Push (48) (8*6) (5+3)*(8-2)
$ end	result = pop()		Display (result) (5+3)*(8-2) = 48 (final)

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 on to the stack.
2) If the character is a closing symbol, and if the stack is empty report an error as missing opening 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	#	stack-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.

Tower of Hanoi

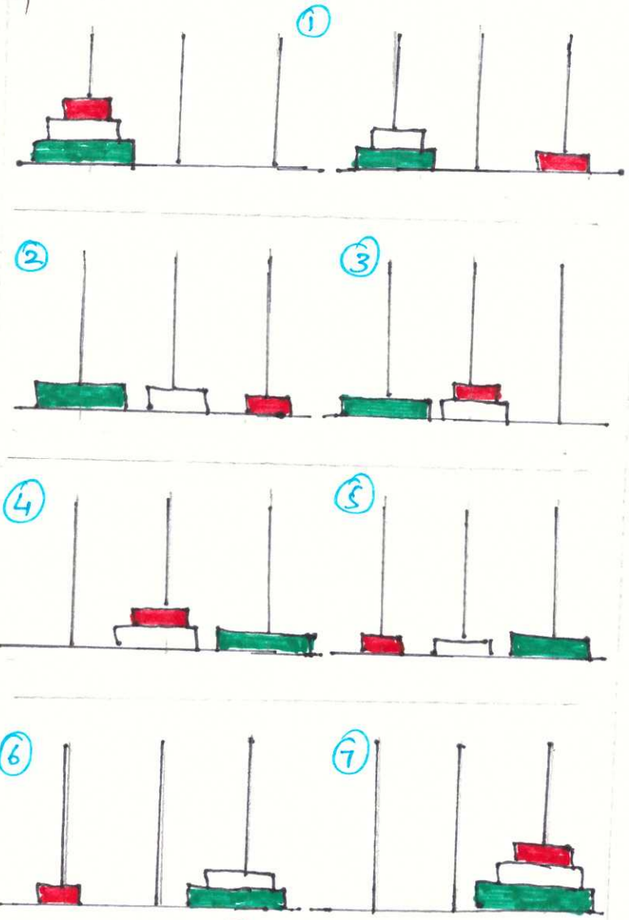
mathematical puzzle where we have 3 rods & 2 disks (Example)

➤ **Objective:** - To move the entire stack to another rod, with following rules.

1) Only one disk can move at a time
2) Each move taking upper disk from one of the stack & placing it on top of another disk.
3) No disk may be placed on top of a smaller disk.

➤ **steps :- 3 Disk**

1) shift 'n-1' disks from A to B
2) shift last disk from A to C
3) shift 'n-1' disks from B to C



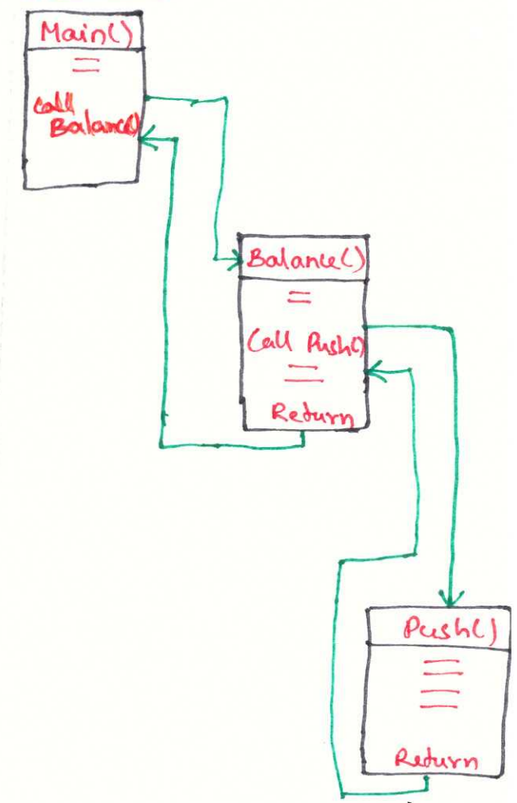
7 moves.

Function Calls.

→ To keep track of the point to which each active sub-routine should return control when it finishes executing.

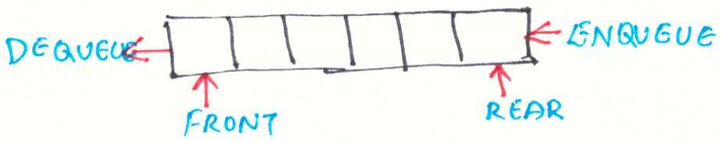
→ Active subroutine is one that has been called but is yet to complete execution after which control should be handed back to the point of call.

➤ **Example**



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.

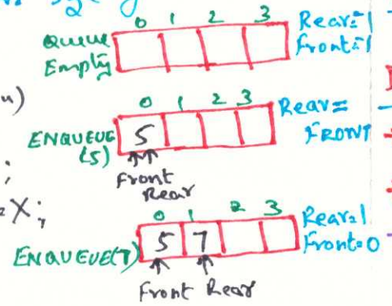
→ **Basic Operations**:
*Enqueue
*Dequeue
*Display.

→ **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.

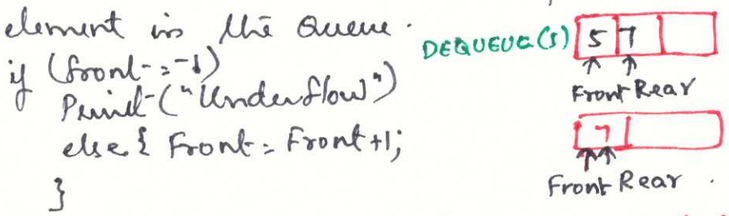
Note:- N - maximum Size of Queue.

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



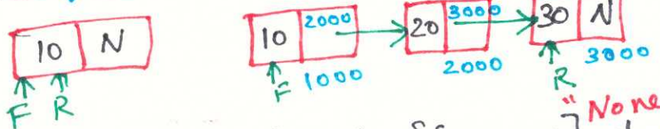
Dequeue:
Note:- Before deleting, check whether Queue is empty (Underflow Condition)
* If empty, display "Queue Underflow" Front=-1 (empty)
* If not empty, element in the front is deleted from the Queue and front is pointed to the next element in the Queue.

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



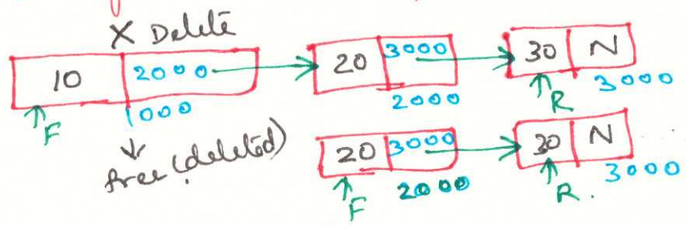
→ **Implementation - Using Linked List**.
Each node contains 2 fields - Data field (Data) & Pointer Field (Address of next node)

Enqueue: F → Front ; R → Rear



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

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

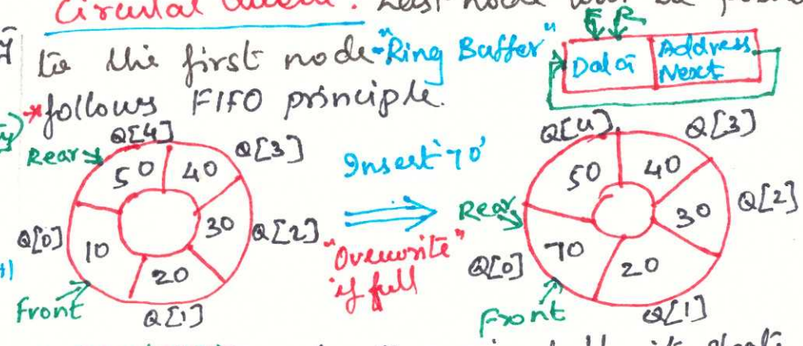


Advantages over Arrays:
Dynamic Size, Easy to insert and delete.

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

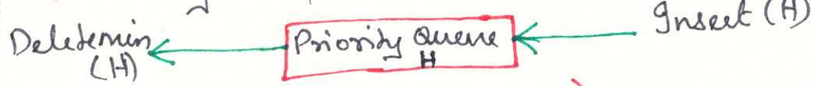
→ **Basic Operations**:- Enqueue & Dequeue.

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.

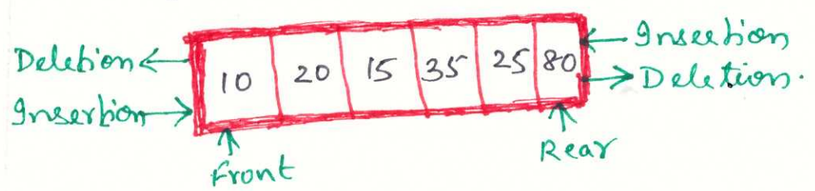
→ **Basic Operations** :- Insert (Enqueue) & Delete (Deletemin)

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



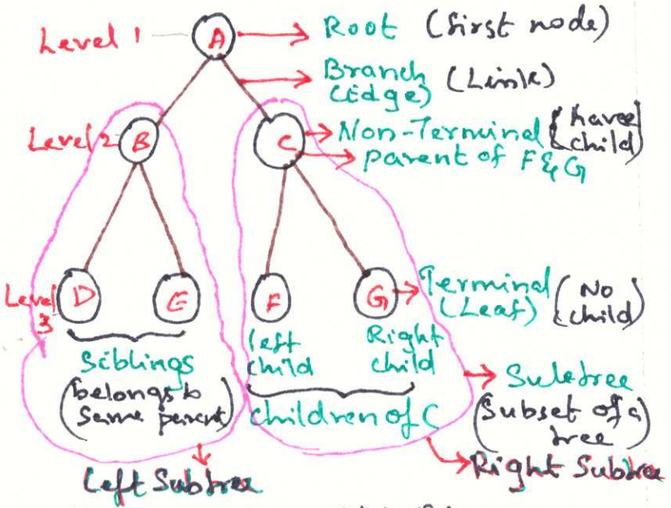
Double-Ended Queue (DEQUE)

Both insertion and deletion operations are performed at both the ends.



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

TREES - a Non-Linear DS
*represents hierachial relation



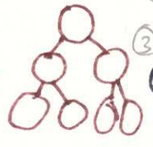
Note:- N nodes → N-1 Edges

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.

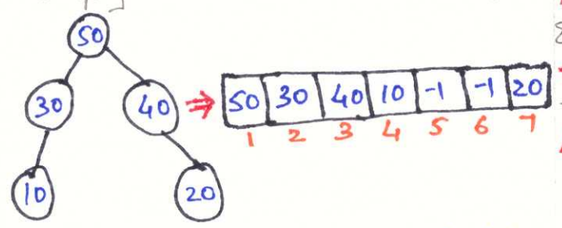
*Strictly Binary Tree - nodes with either 2 or no node at all (Full Binary Tree)



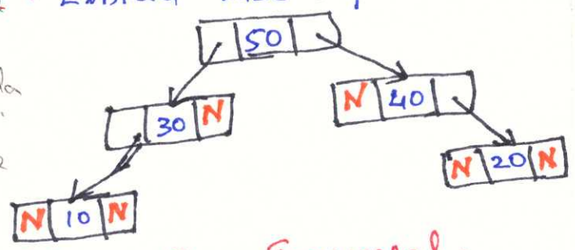
* Complete Binary Tree - Internal nodes has exactly 2 children & leaf nodes at same level.

Binary Tree - Representation

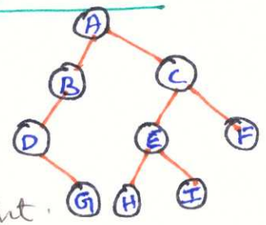
* Array Representation
Root → position 1
Parent (i) → i/2 (i → Node Position)
left child → 2i
Right child → 2i+1
Empty Position → -1 (No Node)



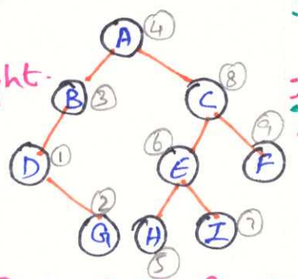
* Linked List Representation



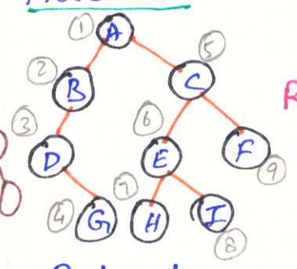
Binary Tree - Traversal.
↳Inorder
↳Pre-order
↳Post-order
Note: Traverse from left → Right.



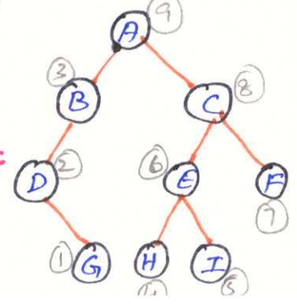
Inorder:
Left → Root → Right.
DGBAHEICF



Pre order:
Root → Left → Right
ABDGCEHIF

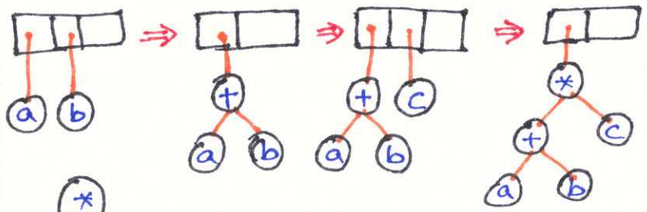


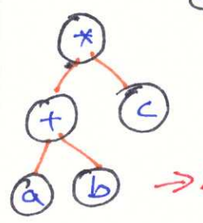
Post-order:
Left → Right → Root
GDBHIEFCA



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]



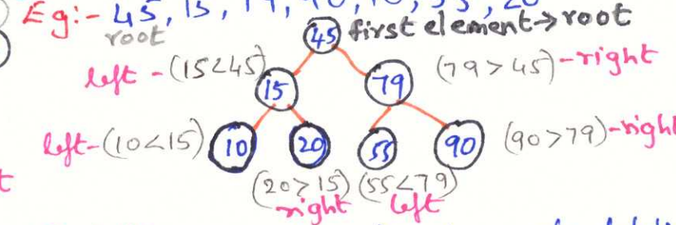


(a+b)*c → Expression Tree

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

BST - Operations:
Insert, Delete, Search.

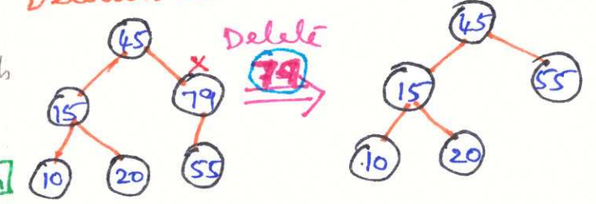
Insert: Comparing with root/Parent.
Eg:- 45, 15, 79, 90, 10, 55, 20



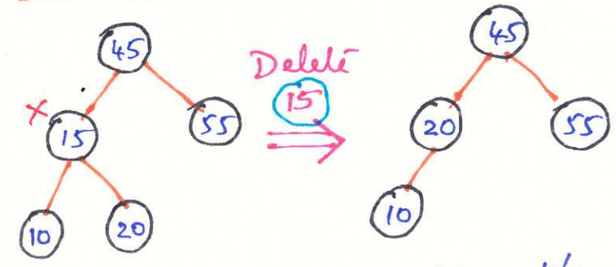
Delete: 3 categories in node deletion
→ Node may not have any children
→ Node may have only one child
→ Node may have two children.

Deletion - Node with No children.

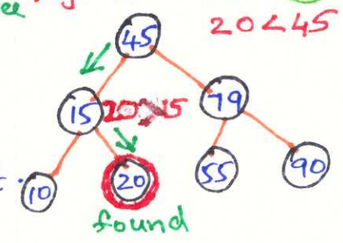
Deletion - Node with one child.



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.

TOPIC - 8 (AVL TREE)

AVL Tree:- Balanced Search Tree.
Adelson-Velskii-Landis.

Balance Factor = Height|Left| - Height|Right|
(BF)

BF = -1, 0, 1 ⇒ Balanced.

Eg:- 3 — 16 BF = 2-2=0 (l, r)
2 — 9 BF=1-0 BF=1; 20 BF=1-1=0
1 — 5 BF=0; 18 BF=0; 25 BF=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 = -1,0,1) then perform rotations to balance
* AVL causes unbalanced when insert
  - Single Rotation: Left Subtree - Left child (LL); Right Subtree - Right child (RR)
  - Double Rotation: Left Subtree - Right child (LR); Right Subtree - Left child (RL)

→ **Left Rotation (LL)** - Single.

3 BF=1 (1-0) → 3 BF=2 (2-0) → 2 BF=0 (1-1)
3 BF=0; 2 BF=0; 1 BF=0; 3 BF=0
Balance BF=0
Unbalanced
Balanced

→ **Right Rotation (RR)** - Single

3 BF=-1 (0-1) → 3 BF=-2 (0-2) → 5 BF=0 (1-1)
5 BF=0; 3 BF=0; 7 BF=0
Balanced
Unbalanced
Balanced

→ **Left-Right Rotation (LR)** ↳ Double

Insert 2

5 BF=1 (1-0); 3 BF=0
Balanced.

5 BF=2 (2-0); 3 BF=-1 (0-1); 2 BF=0
Unbalanced.

Left Rotation

2 BF=2 (2-0); 3 BF=1 (1-0); 5 BF=0
Unbalanced

Right Rotation

3 BF=0 (1-1); 2 BF=0; 5 BF=0
Balanced.

→ **Right-Left Rotation (RL)** ↳ Double.

Insert 5

3 BF=-1 (0-1); 8 BF=0
Balanced

3 BF=-2 (0-2); 8 BF=1 (1-0); 5 BF=0
Unbalanced

Right Rotation

3 BF=-2 (0-2); 5 BF=-1 (0-1); 8 BF=0
Unbalanced

Left Rotation

5 BF=0 (1-1); 3 BF=0; 8 BF=0
Balanced.

Operations - AVL Tree

* Insertion
* Deletion
* Search

Insertion: Insert '10'

16 (2-2); 9 (1-0); 20 (1-1); 5 BF=0; 18 BF=0; 25 BF=0
Balanced

insert '10' ⇒

16 (2-2); 9 (1-1); 20 (1-1); 5 BF=0; 10 BF=0; 18 BF=0; 25 BF=0
Balanced

Deletion: Delete '30'.

20 (2-1); 10 (1-1); 30 BF=0; 5 BF=0; 15 BF=0
Balanced

delete '30' ⇒

20 (2-0); 10 (1-1); 5 BF=0; 15 BF=0
Unbalanced.

10 (1-2); 5 BF=0; 20 (1-0); 15 BF=0
Balanced.

Search: Search '18'

Similar in BST

16 (2-2); 9 BF=1 (1-0); 20 (1-1); 5 BF=0; 18 BF=0; 25 BF=0

18 > 16 ↳ Traverse Right
18 < 20 ↳ Traverse Left
18 Found

Activity: Balance a Tree using AVL
Insert 20, 10, 40, 50, 90, 30

① Insert 20
20 BF=0.
Balanced

② Insert 10
20 BF=1; 10 BF=0
Balanced.

③ Insert 40
20 BF=0; 10 BF=0; 40 BF=0
Balanced.

④ Insert 50
20 BF=-1; 10 BF=0; 40 BF=-1; 50 BF=0
Balanced

⑤ Insert 90
20 BF=-2; 10 BF=0; 40 BF=-2; 50 BF=-1; 90 BF=0
Unbalanced

20 BF=-1; 10 BF=0; 50 BF=0; 40 BF=0; 90 BF=0
Balanced.

⑥ Insert 30 (RL Case)
20 BF=-2; 10 BF=0; 50 BF=1; 40 BF=1; 90 BF=0; 30 BF=0
Unbalanced

Right Rotation

20 BF=-2; 10 BF=0; 40 BF=-1; 30 BF=0; 50 BF=0; 90 BF=0
Unbalanced

Left Rotation

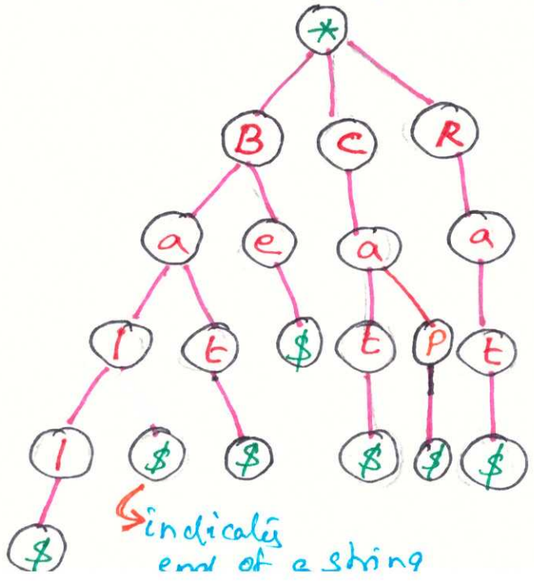
40 BF=0; 20 BF=0; 50 BF=-1; 10 BF=0; 30 BF=0; 90 BF=0
Balanced

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' come 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

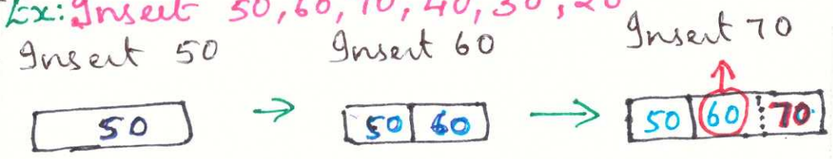
Types:
2-3, 2-3-4, 2-3-4-5, ... and so on.

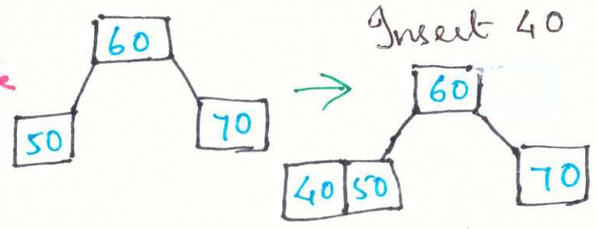
2-3 Tree.

→ B tree of order 3
→ Each node has either 2 or 3 children & atmost 2 key values.

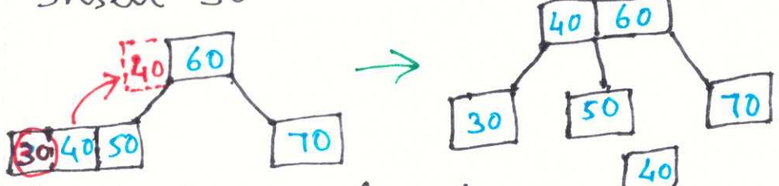
Ex: Insert 50, 60, 70, 40, 30, 20

Insert 50 → Insert 60 → Insert 70





Insert 30



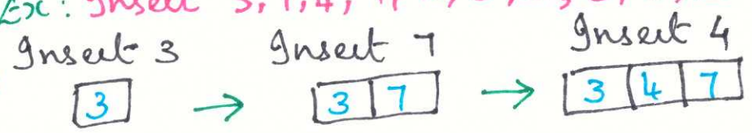
Insert 20 → Insert 10

2-3-4 Tree

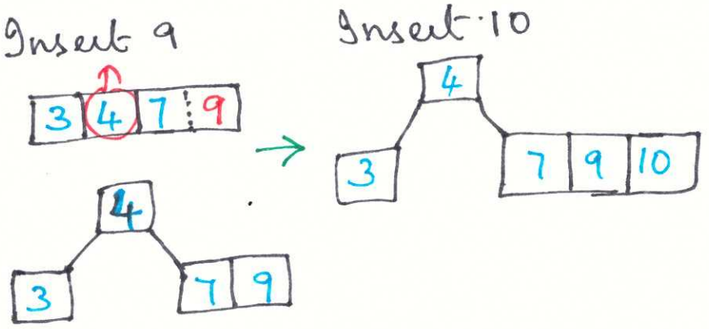
→ B tree of order 4
→ Each node has either 2/3/4 children & atmost 3 key values.

Ex: Insert 3, 7, 4, 9, 10, 0, 5, 6, 8, 2

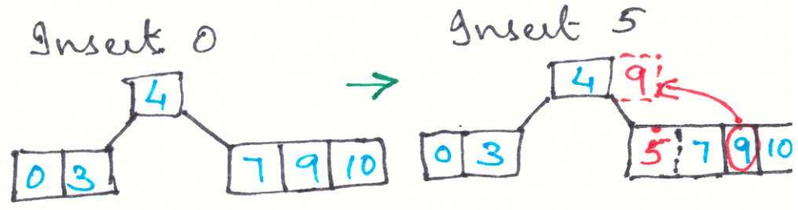
Insert 3 → Insert 7 → Insert 4

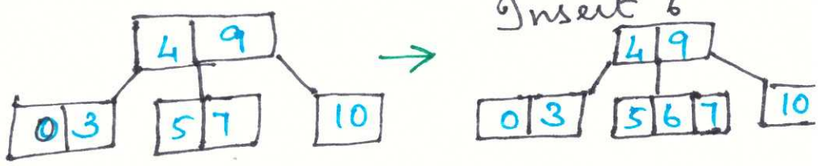


Insert 9 → Insert 10

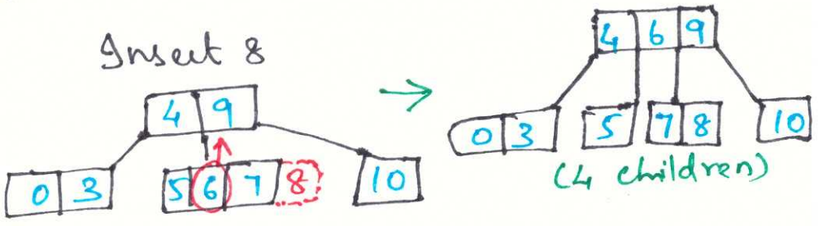


Insert 0 → Insert 5

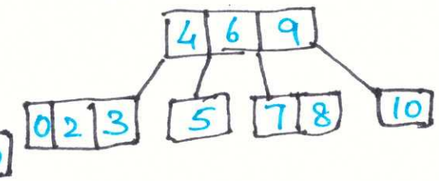




Insert 8



Insert 2



TOPIC-10 (RED-BLACK TREE)

RED-BLACK TREE: BST in which every node is colored either RED or BLACK.

Properties:
- → BST → New node - RED color.
- → Root node - BLACK color.
- → No 2 consecutive RED nodes.
- → All paths - same no. of BLACK node
- → Leaf node - BLACK color (NULL node)

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 Newnode as a Leaf node (RED)
4) parent (NewNode) - BLACK, exit
5) Parent (NewNode) - RED, check color of parent node's sibling of New Node.
6) Sibling - BLACK or NULL, - Rotate and Recolor it.
7) Sibling - RED, - Recolor.

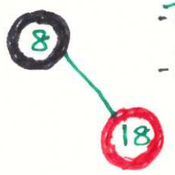
Repeat the steps until tree becomes RED-BLACK Tree.

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

Insert 8

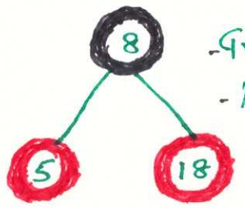
- NewNode - BLACK

Insert 18



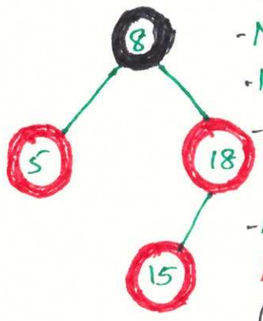
- Tree Not empty
- NewNode - RED

Insert 5



- Tree Not empty
- New Node - RED

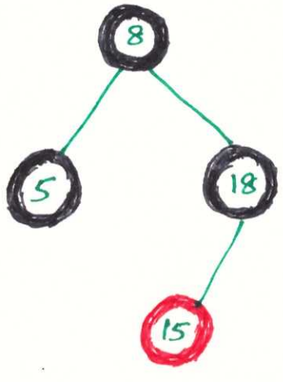
Insert 15



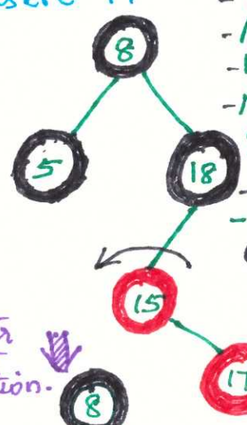
- Tree not empty
- New Node - RED
- 18, 15 - RED (consecutive)
- NewNode Parent's Sibling - RED (5)
- Recolor (Step 7)

Note: Don't recolor if Parent - root (BLACK)

After Recolor (5, 18)

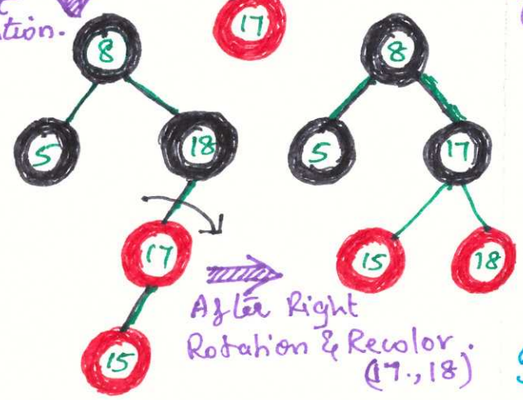


Insert 17



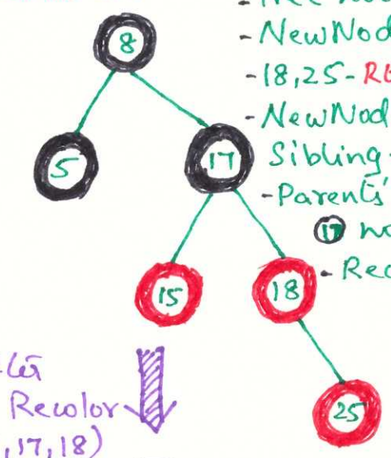
- Tree not empty
- NewNode - RED
- 15, 17 - RED (consecutive)
- NewNode Parent's Sibling - NULL
- Rotate & Recolor
- LR-Case (Left-Right Rotation)

After Left Rotation.



After Right Rotation & Recolor. (17, 18)

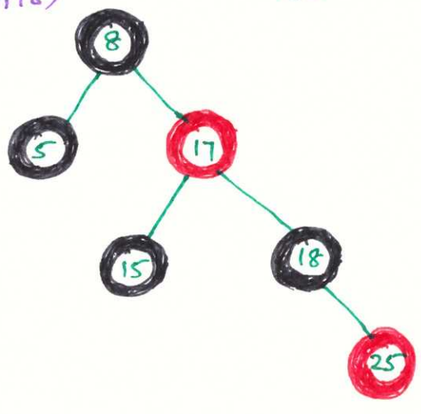
Insert 25



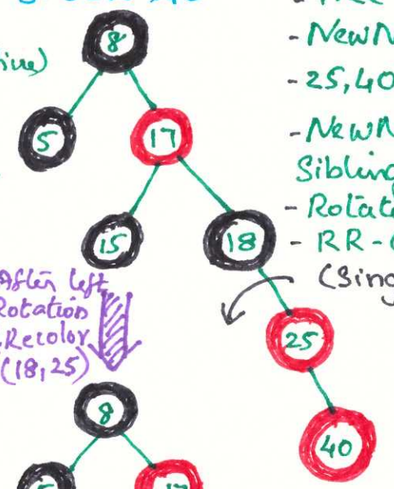
- Tree not empty
- NewNode - RED
- 18, 25 - RED (consecutive)
- NewNode Parent's Sibling - RED (15)
- Parent's Parent - not root (17)
- Recolor.

After Recolor (18, 25, 40)

After Recolor (15, 17, 18)



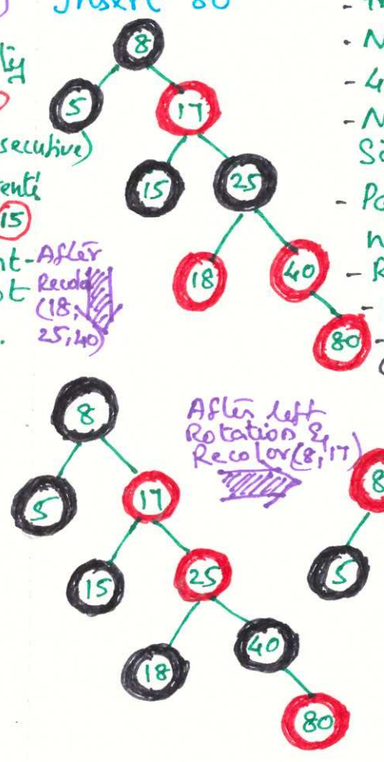
Insert 40



- Tree not empty
- NewNode - RED
- 25, 40 - RED
- NewNode's Parent Sibling - NULL
- Rotate & Recolor
- RR - Case (Single left Rotation)

After Left Rotation & Recolor (18, 25)

Insert 80



- Tree not empty
- NewNode - RED
- 40, 80 - RED
- NewNode Parent's Sibling - RED (18)
- Parent's Parent - not root (25)
- Recolor
- 17, 25 - RED
- RR - Case (Single left Rotation)

After left Rotation & Recolor (8, 17)

Thus, RED-BLACK Tree is created. No. of BLACK nodes in all paths - 2 (same)

TOPIC - 11 (SPLAY TREE)

SPLAY TREE : Self-adjusted BST

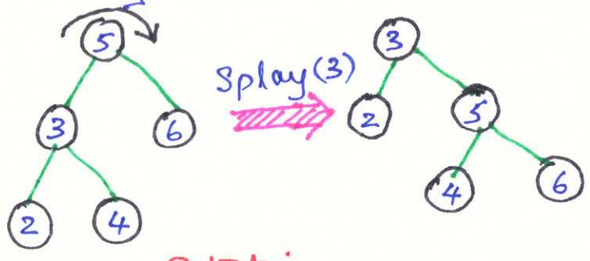
* rearrange the element by placing at the root position = "Splaying".

Splay Tree - Rotations.

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

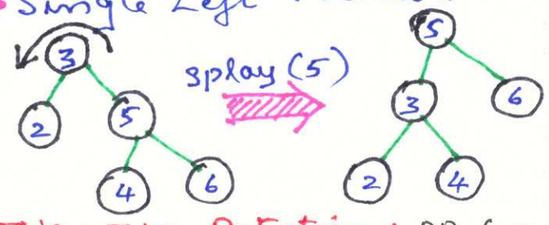
Zig Rotation :

→ Single right Rotation



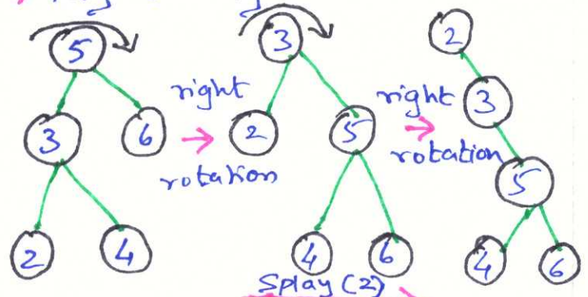
Zag Rotation :

→ Single Left Rotation



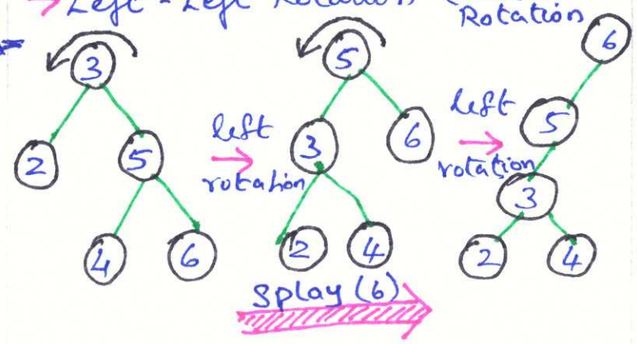
Zig-Zig Rotation : RR-Case

→ Right - Right Rotation



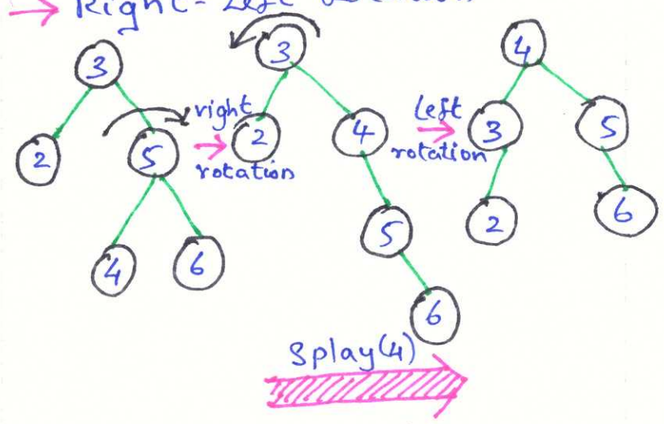
Zag-Zag Rotation : LL-Case

→ Left - Left Rotation (Double Rotation)



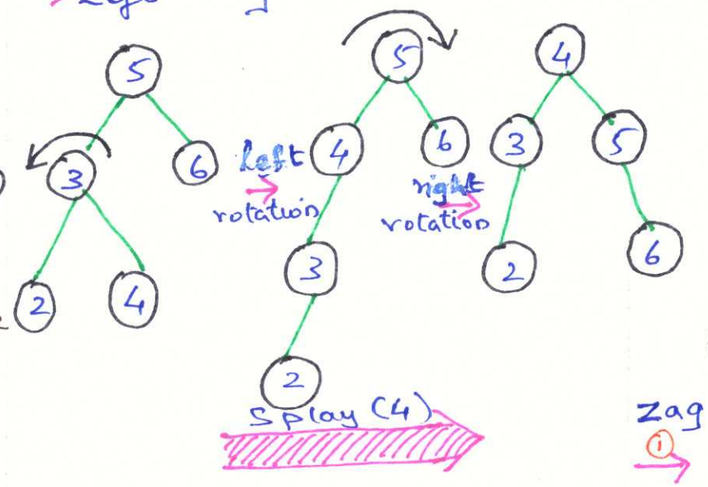
Zig-Zag Rotation : RL-Case

→ Right - Left Rotation



Zag-Zig Rotation : LR-Case

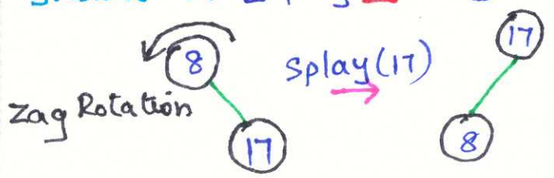
→ Left - Right Rotation.



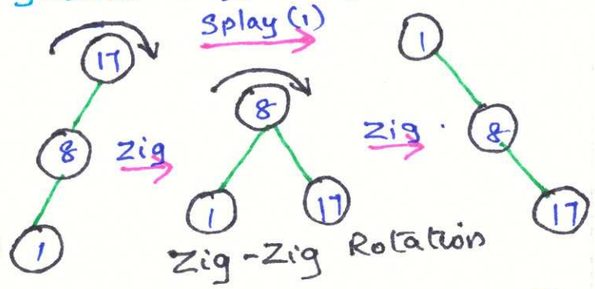
Example :- Insert 8, 17, 1, 14, 16, 15

Insert 8

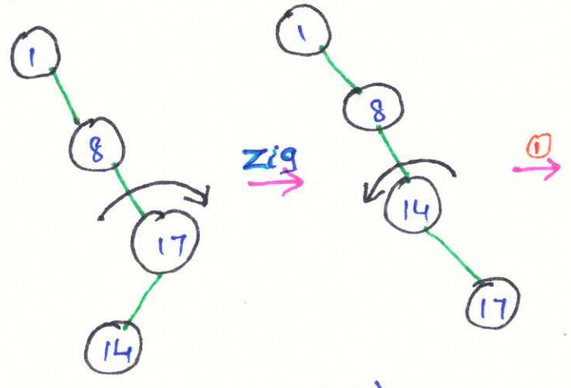
Insert 17 [Splay 17-root]



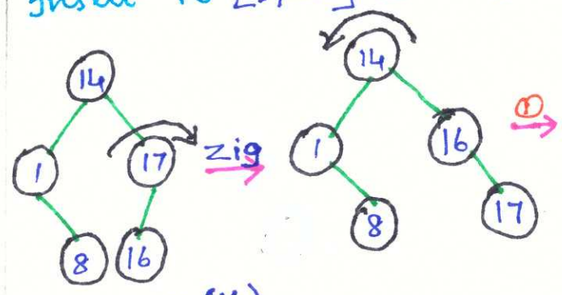
Insert 1 [Splay 1-root]



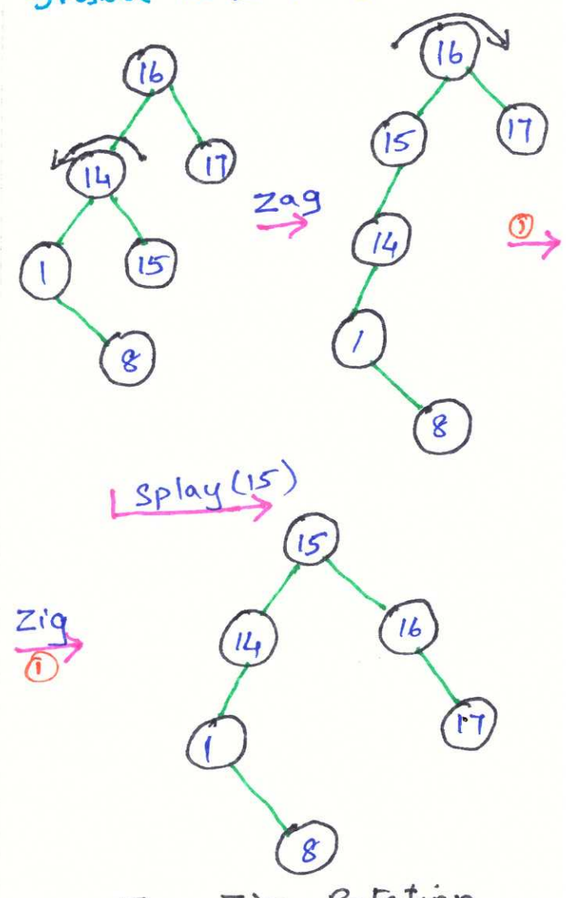
Insert 14 [Splay 14-root]



Insert 16 [Splay 16-root]



Insert 15 [Splay 15-root]



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

Hashing

Hash Table Data Structures - Arrays of keys

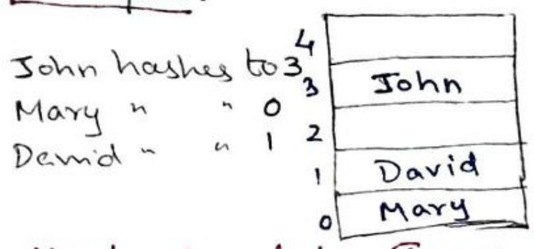
Keys - Strings with associated value.

Hashing - Implementation of Hash Table.

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

Hsize - Hash Table size

Example:



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.

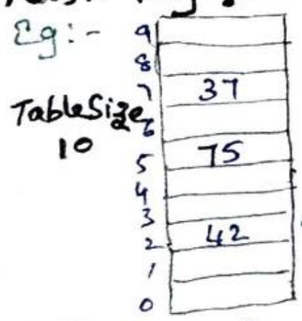
  Eg:- hash(3101) = 3101 * 3101 = 9616201 ⇒ 162 (mid 3 digit)

* 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

⇒ Collision Resolution Techniques is used to resolve Collisions.

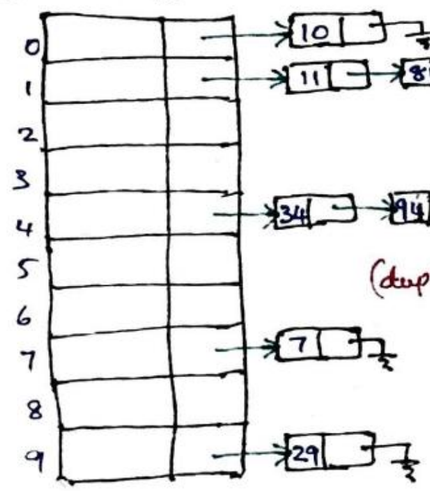
Collision Resolution Techniques

- 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					
8		18	18	18	18
9	89	89	89	89	89

Insert 89, U=9

index = (9+0) % 10 = 9 (i=0) Empty

Insert 18, index = 18 % 10 = 8 ⇐ U

i=0, index = (8+0) % 10 = 8 Empty

Insert 49, index = 49 % 10 = 9 ⇐ U

i=0, index = (9+0) % 10 = 9 Not empty

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

* Similar to Linear Probing

Double Hashing

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

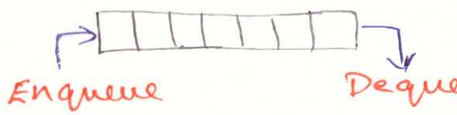
indexH → Hash value computed by another Hash function

TOPIC-13 (PRIORITY QUEUE, HEAP & HEAP SORT)

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.

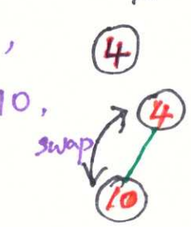
Types:
- → Max_Heap (parent > children)
- → Min_Heap (parent < children)

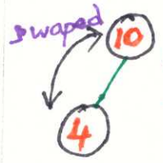
& properties:
- → Shape property
  - ↳ Complete Binary Tree (BT)
- → Heap property
  - ↳ Max_Heap or Min_Heap.

Operations:
Build, Insert, Delete.

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

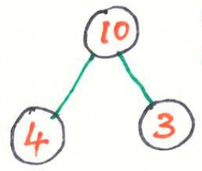
Insert 4,





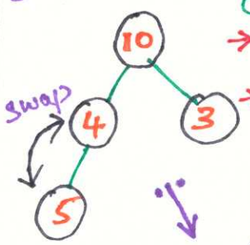
Check properties:
- → Shape property - complete BT
- → Heap property - Not satisfied (Max_Heap). So swap. (10, 4)

Insert 3,

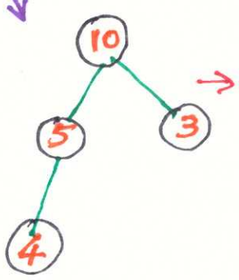


Check properties:
- → Shape property - satisfied
- → Heap property - satisfied (Max_Heap)

Insert 5,

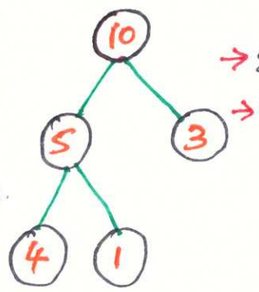


Check properties:
- → Shape property - satisfied
- → Heap property - Not " (Max_Heap). So swap. (4, 5)



→ 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:
  - → Min. Heap Sort - Descending order
  - → Max-Heap Sort - Ascending "

Max-Heap Sort.

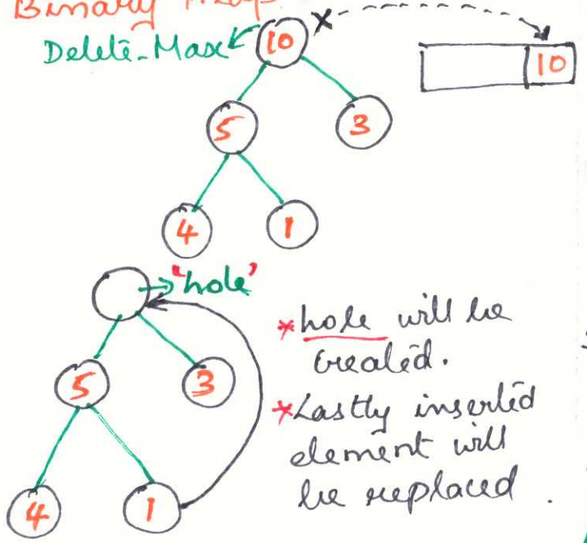
uses Delete-Max operation.

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

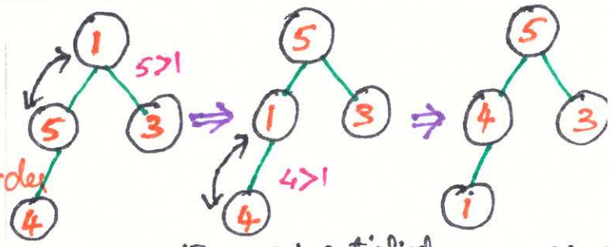
Steps:-
1) Construct Binary Heap (Max/Min)
2) Call Delete-Max / Delete-Min operation (root element - deleted)
3) Place the deleted element in the sorting list.
4) "Hole" will be created.
5) Replace lastly inserted element at the root.
6) Check binary heap properties and satisfy it.
7) Repeat steps till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

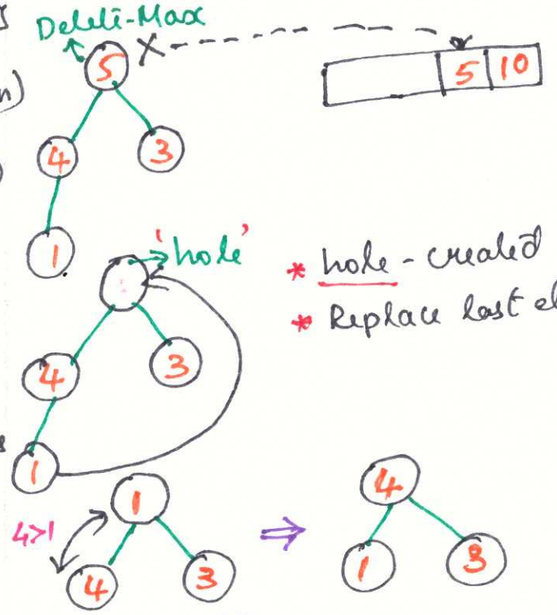
Now,
Solve {4, 10, 3, 5, 1} using Max-Heap Sort (Ascending order)

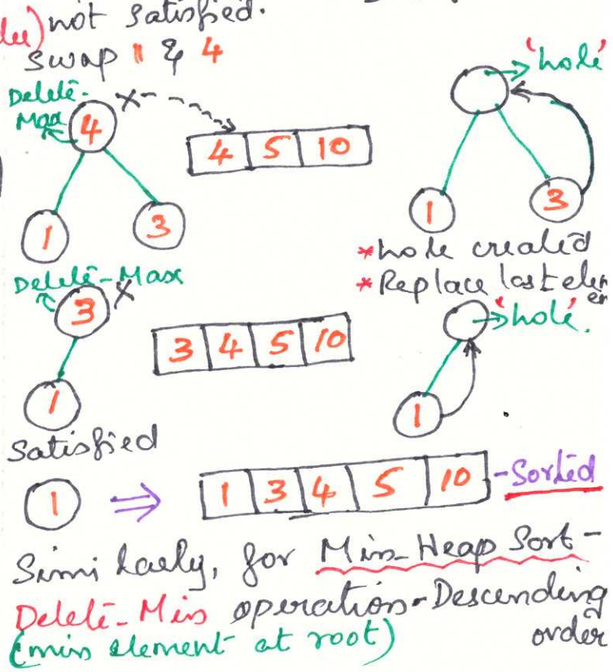
Binary Heap.



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







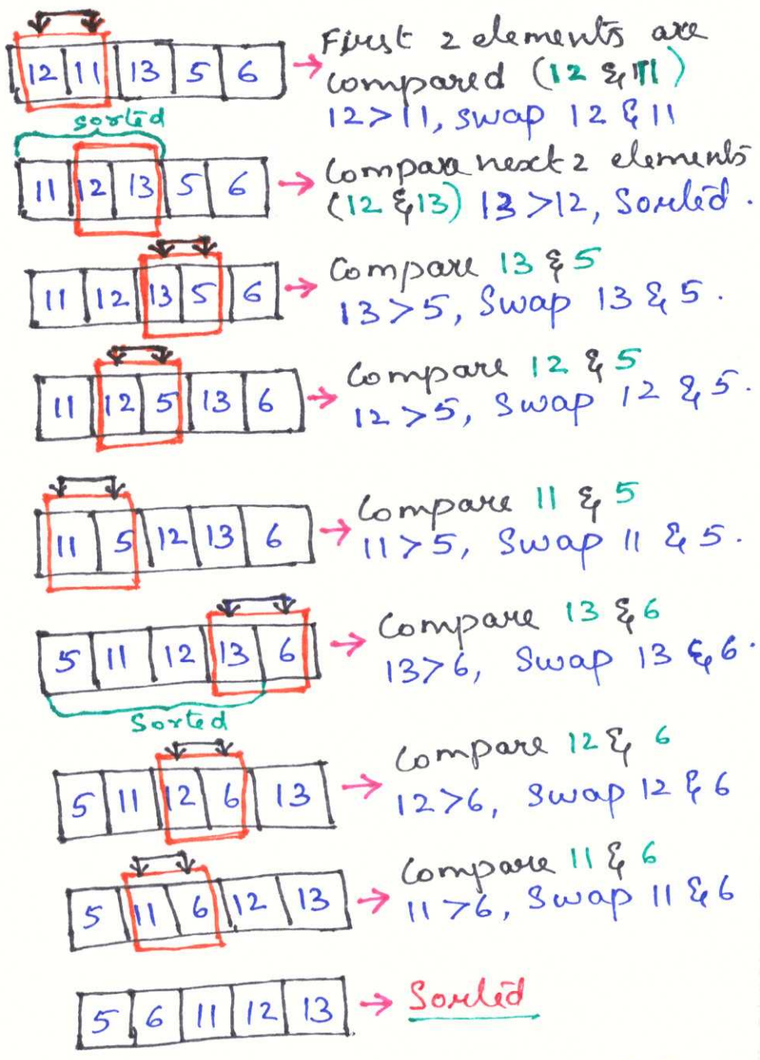
Similarly, for Min-Heap Sort - Delete-Min operation - Descending order (min element at root)

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.

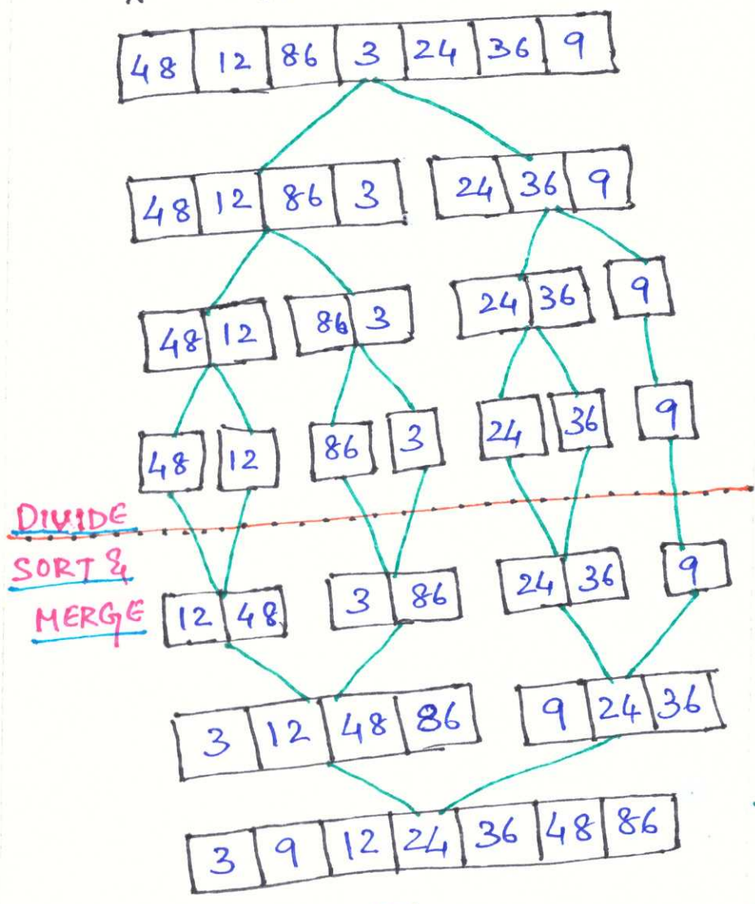


Best Case Time Complexity - $O(n)$
Worst Case Time Complexity - $O(n^2)$
Average Case Time Complexity - $O(n^2)$

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 - $O(n \log n)$

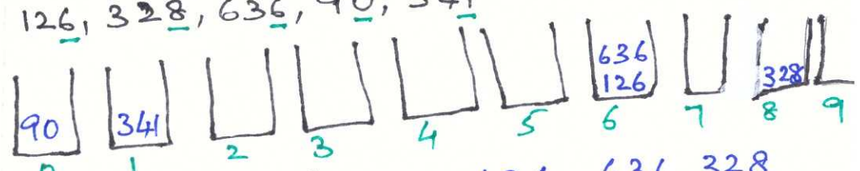
Radix Sort:

* Non Comparative integer Sorting algorith
* 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

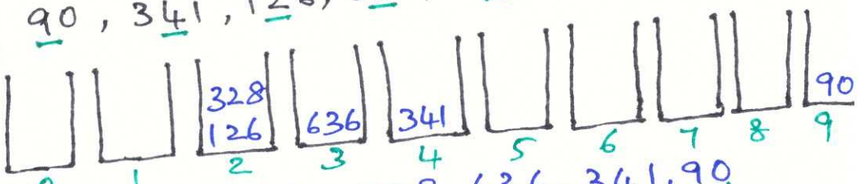


→Arrange:- 90, 341, 126, 636, 328

input for Pass II

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

90, 341, 126, 636, 328

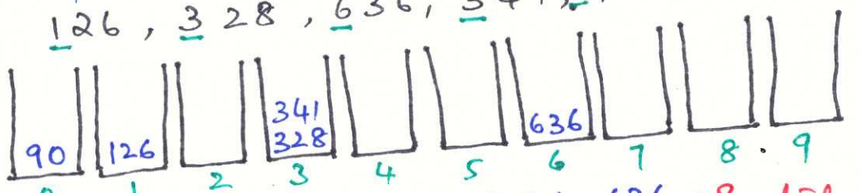


→Arrange:- 126, 328, 636, 341, 90

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...]

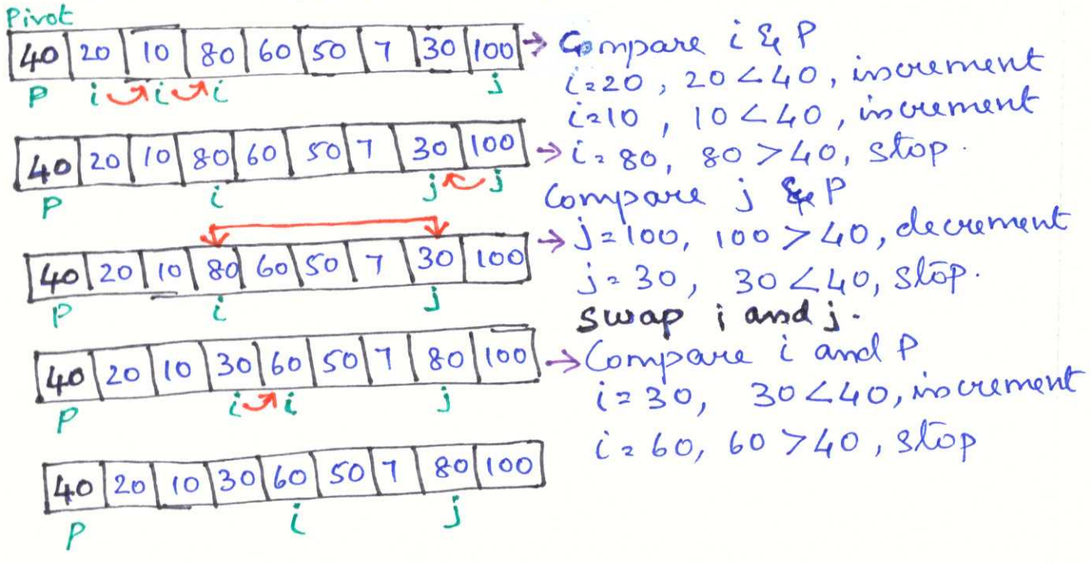
Steps :-

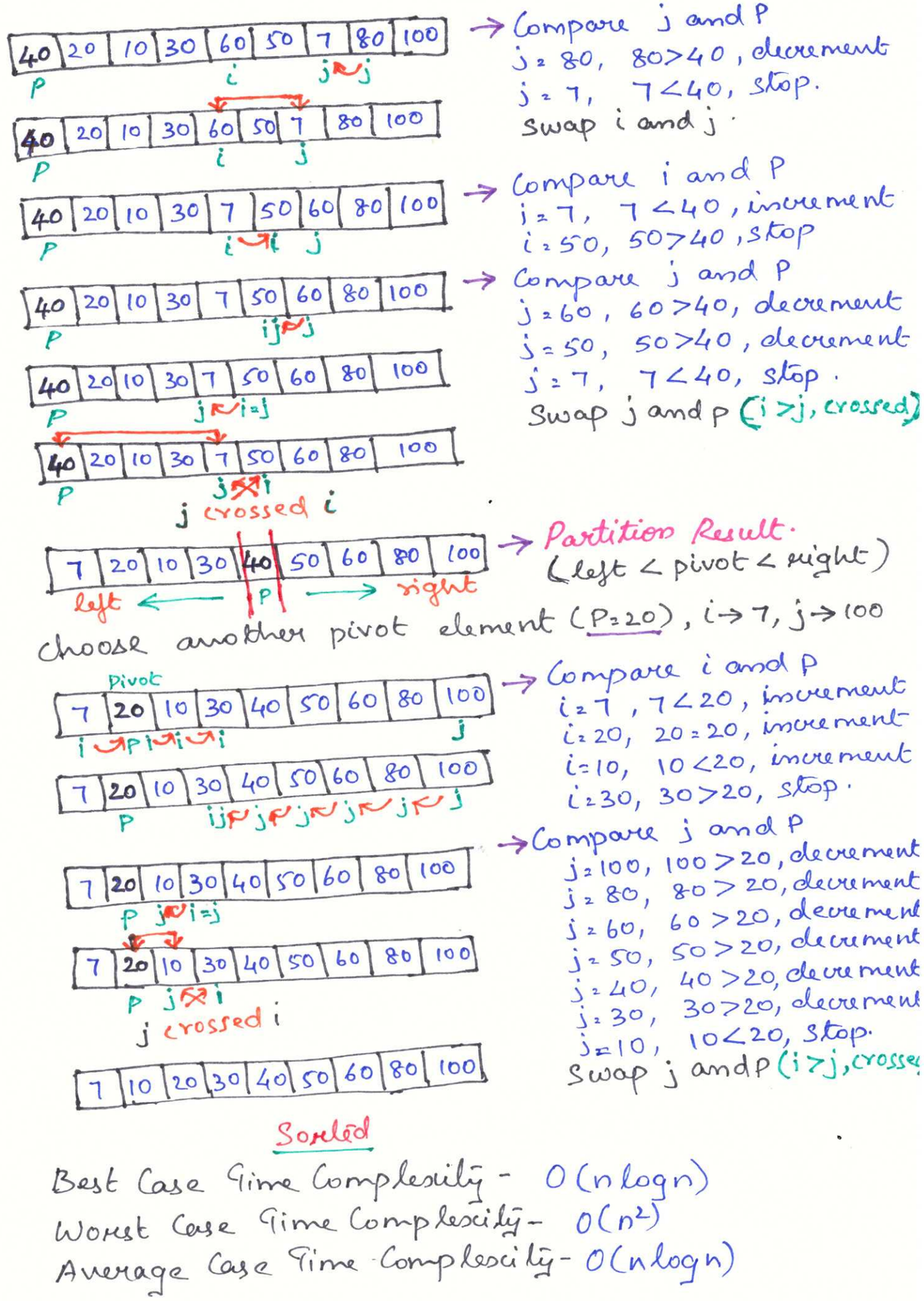
Step 1: Choose a pivot element from the list.
Step 2: Define 2 variables 'i' & 'j'. i → 1st & j → last value
Step 3: Increment i until list[i] > pivot, then stop
Step 4: Decrement j until list[j] < pivot, then stop
Step 5: If i < j, then swap list[i] and list[j]
Step 6: Repeat steps 3, 4, 5 until i > 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(n²)
Average Case Time Complexity - O(n log n)

TOPIC - 16 (GRAPH - TERMINOLOGIES - TYPES - TOPOLOGICAL SORT)

Graphs: $G = (V, E)$

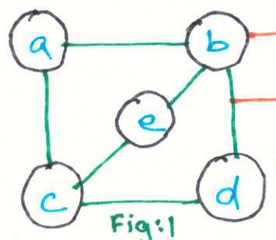


Fig:1

→ 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)

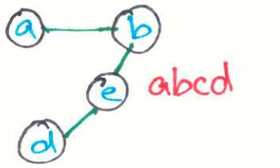
Adjacent vertices → Connected by an edge a → b, c ; c → a, e, d

Length → No. of edges.

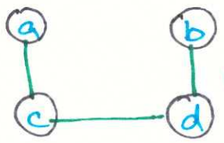
Degree → No. of adjacent vertices

Degree[a] = 2, Degree[b] = 3

Path → Sequence of vertices.

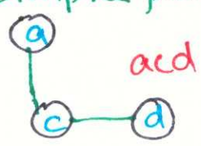


abcd

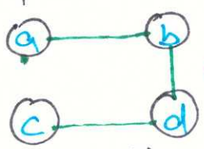


acdb

Simple path → No repeated vertices.

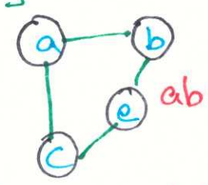


acd

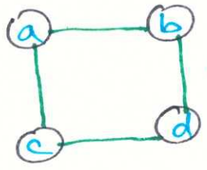


abdc

Cycle → Last & first vertex - Same.

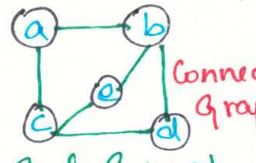


abeca

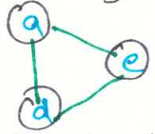


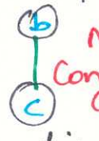
abdca

Connected Graph → 2 vertices connected by some path.



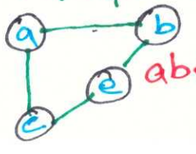
Connected Graph





Not Connected Graph

Sub-Graph → Subset of vertices & edges

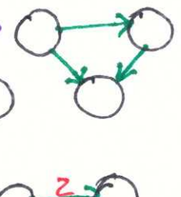


abec

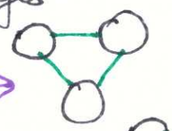
becd

Connected Components - max. connected Subgraph.

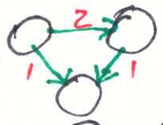
Directed Graph (Digraph): specify directions in edges. →



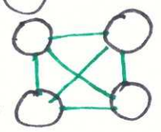
Undirected Graph: No directed edges. →



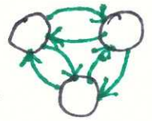
Weighted Graph: have weights in edges →



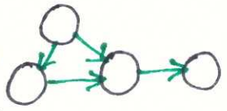
Complete Graph: all vertices are connected to each other (Undirected Graph) →



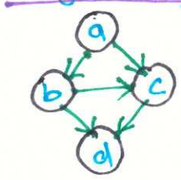
Strongly Connected Graph: all vertices are connected in both directions (Directed Graph) →



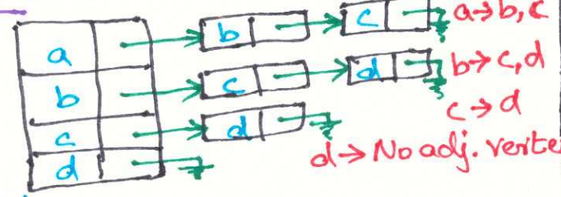
Acyclic Graph: No cycles. →



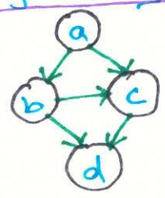
Adjacency List: - Linked list.



→



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
* 4×4 Matrices

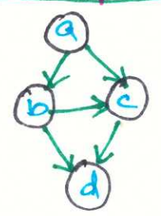
Topological Sort: used in Directed Acyclic Graph (DAG) - directed & No cycles.

* 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:



⟹

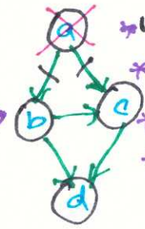
Adjacency Matrix.

	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
a	0
b	1
c	2
d	2
Enqueue : a	
Dequeue : a	

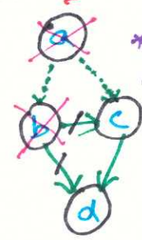
→



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

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

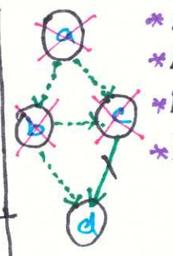
V	1	2	3	4
a	0	0	0	0
b	1	0		
c	2	1		
d	2	2		
Enqueue : ab				
Dequeue : ab				



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

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

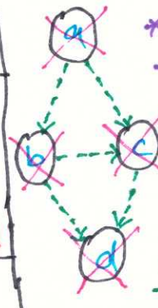
V	1	2	3	4
a	0	0	0	0
b	1	0	0	0
c	2	1	0	
d	2	2	1	
Enqueue : abc				
Dequeue : abc				



* 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 - 17 (SHORTEST PATH ALGORITHMS - UNWEIGHTED SHORTEST PATH - DIJKSTRA'S ALGORITHM)

Shortest Paths Algorithms:

* Finds minimum cost from source to other vertex.

Two types:

→ Single Source Shortest Path (SSSP)

→ All pair Shortest Path (APSP)
  ↳ Shortest paths between all pair of vertices.
  ↳ Floyds - Warshall, Johnson.

Single Source Shortest Path - Unweighted

KNOWN → visited - KNOWN (1)
Not visited - UNKNOWN (0)

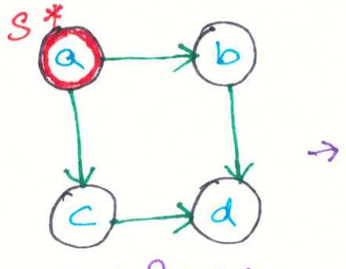
d_v → Distance from source (initially - ∞)
Edge → 1 No Edge → 0

P_v → Actual Paths.

Steps:

1) Source Node - 'S', Enqueue.
2) Dequeue 'S', → KNOWN as 1 & find its adjacency vertex.
3) Distance, d_v → Source vertex distance increment by 1 & Enqueue the vertex
4) Repeat from step ② until Queue becomes empty.

Example:

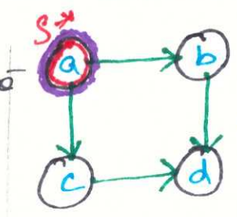


a → Source.

Initial Configuration

V	KNOWN	d_v	P_v
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

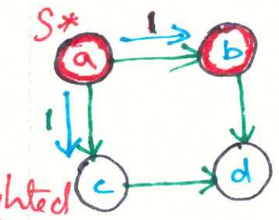
Enqueue :
Dequeue :



V	KNOWN	d_v	P_v
a	1	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Enqueue : a
Dequeue : a

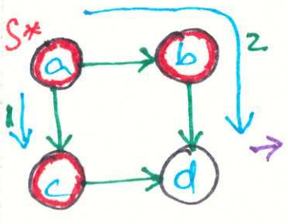
a → KNOWN (1)
Enqueue a
Dequeue a



V	KNOWN	d_v	P_v
a	1	0	0
b	1	1	a
c	0	1	a
d	0	0	0

Enqueue : bc
Dequeue : ab

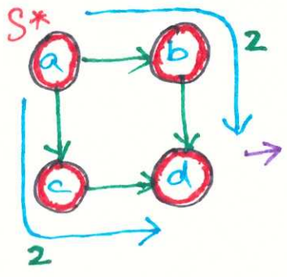
$d_v(b) = 1$
$d_v(c) = 1$
Enqueue b, c
Dequeue b



V	KNOWN	d_v	P_v
a	1	0	0
b	1	1	a
c	1	1	a
d	0	2	b

Enqueue : cd
Dequeue : abc

Enqueue: c, d
Dequeue: c
d_v ⇒ a→b→d(2)
a→c(1)
(Shortest Path)



V	KNOWN	d_v	P_v
a	1	0	0
b	1	1	a
c	1	1	a
d	1	2	b

Enqueue : d
Dequeue : abcd

Enqueue: d
d_v ⇒ a→b→d(2)
a→c→d(2)
(same distance)
Dequeue: d

V	KNOWN	d_v	P_v
a	1	0	0
b	1	1	a
c	1	1	a
d	1	2	b

Queue: Empty

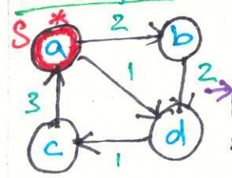
Shortest Path from source vertex 'a'

a → b = 1
a → c = 1
a → d = 2

Dijkstra's Algorithm: (Weighted)

* Single Source Shortest Paths Algorithm
* Similar to Unweighted SSSPA, needs to calculate the distance using weight

Example

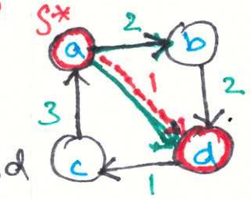


Initial Configuration.

V	KNOWN	d_v	P_v
a	0→1	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

Enqueue : a
Dequeue : a

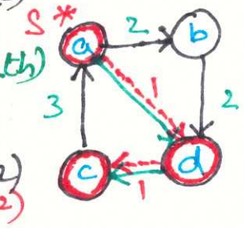
a → KNOWN (1)
Enqueue: a
Dequeue: a



V	KNOWN	d_v	P_v
a	1	0	0
b	0	2	a
c	0	∞	0
d	1	1	a

Enqueue : b d
Dequeue : ad

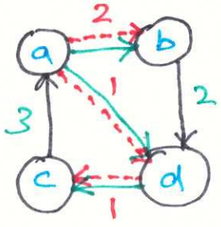
a → b = 2
a → d = 1
(shortest path)
Enqueue: d
Dequeue: d
(path from a)



V	KNOWN	d_v	P_v
a	1	0	0
b	0	2	a
c	1	2	d
d	1	1	a

Enqueue : c
Dequeue : adc

a → d → c = 2
Enqueue: c
Dequeue: c
(path from a)



V	KNOWN	d_v	P_v
a	1	0	0
b	1	2	a
c	1	2	d
d	1	1	a

Enqueue : b
Dequeue : adcb

a → b = 2
Enqueue: b
Dequeue: b
(path from a)

Shortest Path from source vertex 'a'

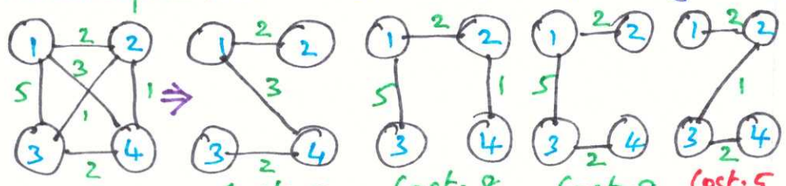
a → b = 2
a → c = 2
a → d = 1

TOPIC - 18 (MINIMUM SPANNING TREE - PRIM'S & KRUSKAL'S ALGORITHMS)

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]

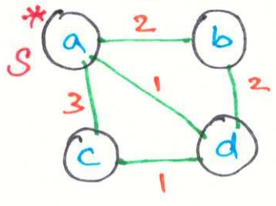


Algorithms:
* Prim's Algorithms
* Kruskal's Algorithms

Prim's Algorithms

↳ Uses Greedy Techniques.
↳ Based on vertices.

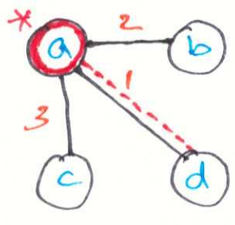
Example:



Initial Configuration

V	KNOWN	d_v	P_v
a	0	0	0
b	0	∞	0
c	0	∞	0
d	0	∞	0

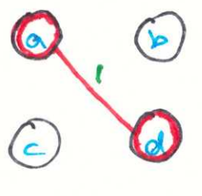
a → Source



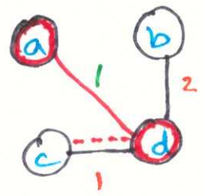
V	KNOWN	d_v	P_v
a	1	0	0
b	0	2	a
c	0	3	a
d	0	1	a

a → KNOWN (1) visited

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

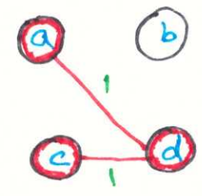


V	KNOWN	d_v	P_v
a	1	0	0
b	0	2	a
c	0	3	a
d	1	1	a



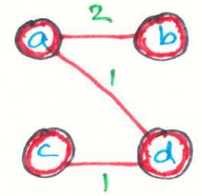
V	KNOWN	d_v	P_v
a	1	0	0
b	0	2	a
c	0	1	d
d	1	1	a

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



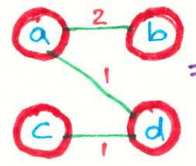
V	KNOWN	d_v	P_v
a	1	0	0
b	0	2	a
c	1	1	d
d	1	1	a

c → KNOWN (1)
Path from 'd'



V	KNOWN	d_v	P_v
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:

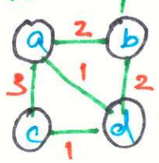
$C_{a,b} + C_{a,d} + C_{c,d} = 2+1+1 = 4$

Kruskal's Algorithms

↳ Uses Greedy Techniques
↳ Based on edges.

Steps:
1) Remove all loops & parallel edges (high cost) from graph if any.
2) Arrange all edges in ascending order.
3) Add the edges which has the least cost.

Example: ①

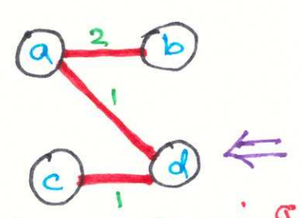


Edge	d_v
(a,b)	2
(a,c)	3
(a,d)	1
(b,d)	2
(c,d)	1

⇒

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

Sort edges in ascending order.



Edge	d_v	Visit
(a,d)	1	✓
(c,d)	1	✓
(a,b)	2	✓
(b,d)	2	×
(a,c)	3	×

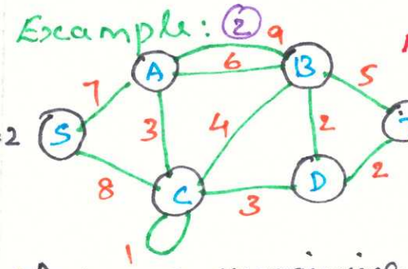
cycle (bda)
cycle (adca)

Minimum Spanning Tree

Minimum Cost:

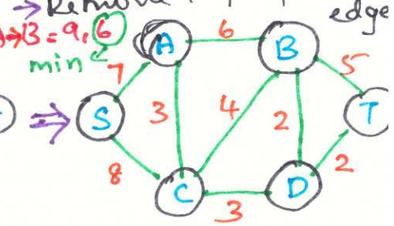
$C_{a,b} + C_{a,d} + C_{c,d} = 2+1+1 = 4$

Example: ②

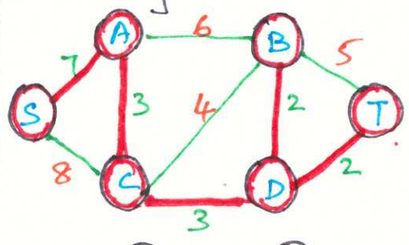


→ Remove loops & Parallel edge

A → B = 9, 6 min

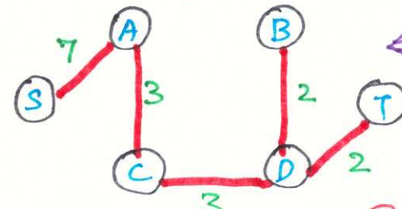


→ Arrange remaining edges in ascending order.



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

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



Minimum Spanning Tree

Minimum Cost:

$C_{S,A} + C_{A,C} + C_{C,D} + C_{D,B} + C_{D,T} = 7+3+3+2+2 = 17$

TOPIC-19 (BREADTH FIRST SEARCH, DEPTH FIRST SEARCH)

Breadth First Search (BFS)

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

Applications:

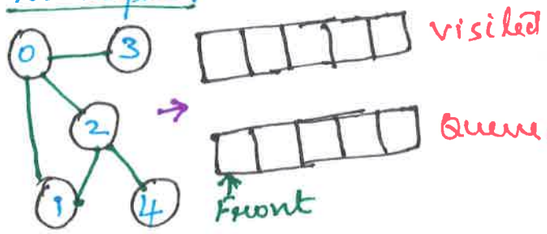
* GPS Navigation
* Paths Finding Algorithms
* Minimum Spanning Tree

2 categories - BFS Implementation
↳ Visited & Not visited

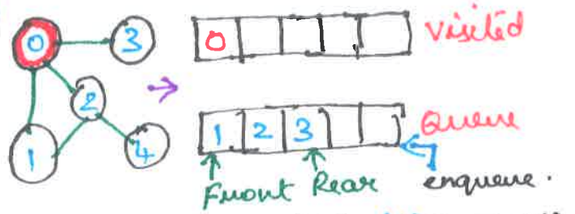
Algorithm steps:

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.

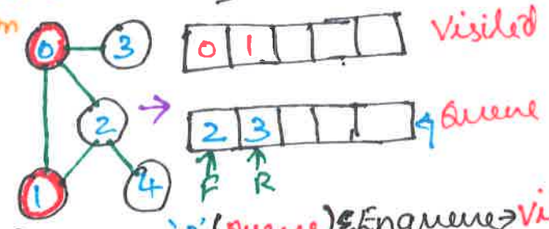
Example:



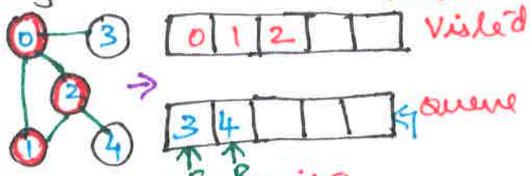
Vertex, 0 → Source.
Enqueue 0 → Visited
Adjacent vertices (0) → 1, 2, 3 → Queue



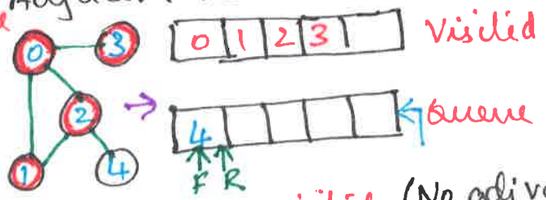
Dequeue front item '1' & move to visited list
Enqueue 1 → visited
Adj. vertices (1) → Queue (0,2)
'0' → already visited



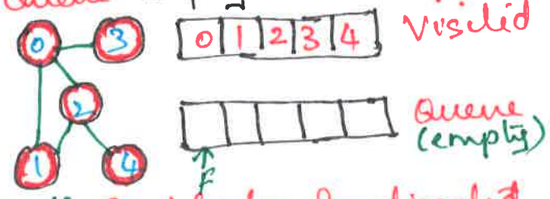
Dequeue '2' (Queue) & Enqueue → Visited
Adjacent vertices (2) → 0, 1, 4 → Queue
visited



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:

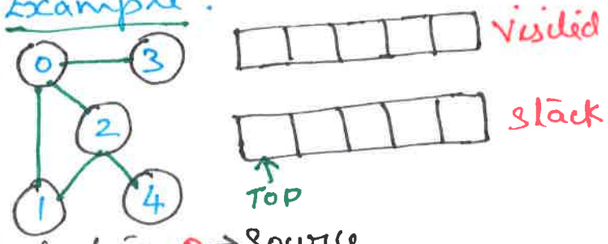
* Topological sorting.
* Detect cycles in the graph.

2 categories - DFS Implementation
↳ Visited & Not Visited

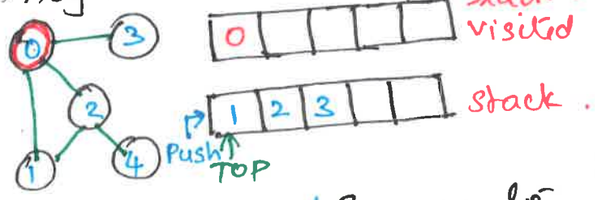
Algorithm steps:

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.

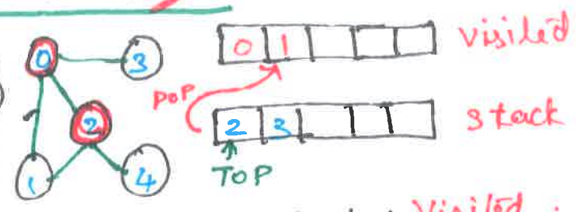
Example:



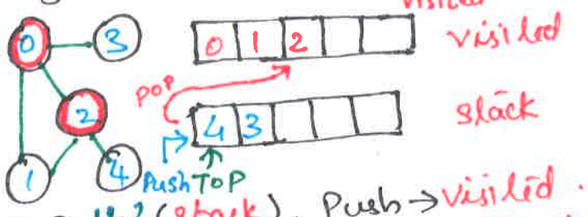
Vertex, 0 → Source
Insert 0 → Visited
Adjacent vertices (0) → 1, 2, 3 → Stack.



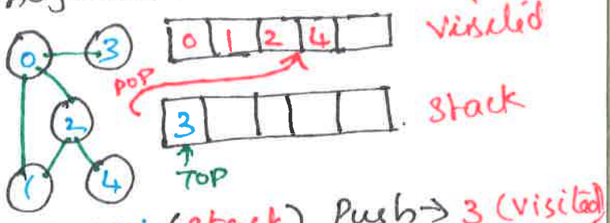
Pop Top item '1' & move to Visited list.
Adj. vertices (1) → Stack (0,2) visited



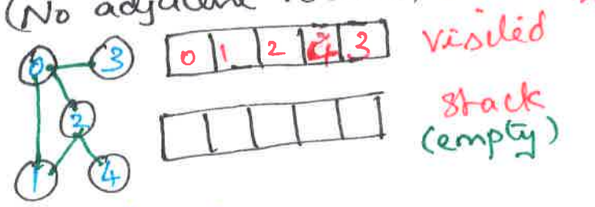
Pop '2' (Stack), Push → Visited.
Adjacent vertices (2) → 0, 1, 4 → Stack visited



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



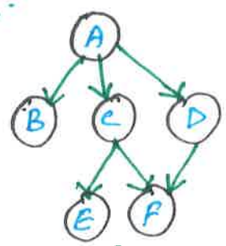
Pop '3' (Stack), Push → 3 (visited)
(No adjacent vertices). stack empty



Advantage:

* solutions will definitely found

Activity:



BFS : ABCDEF
DFS : ADFCEB