Arrays → Linear Data Structure

realloc = Allocates Another Memory, not the Same Memory

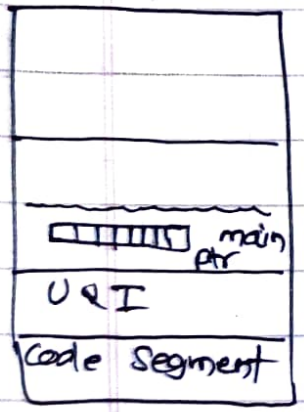
ADTs → Abstract Data Type
→ set of values + set of operations on them

eg 1 int (primitive DT) → 9 | 10 | 12

9 + 12 = 21
↓
operator
↓
Details are abstracted

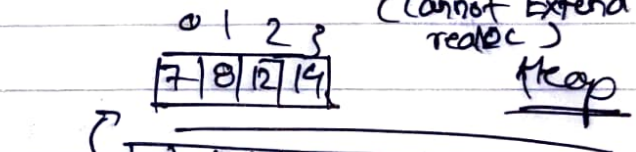
eg 2.
myArray → [representation] → total_size → (6)
used_size → (3)
Base_Address (Pointer)

+ [operations] → max()
get(i)
set(i, num)
add(arr)



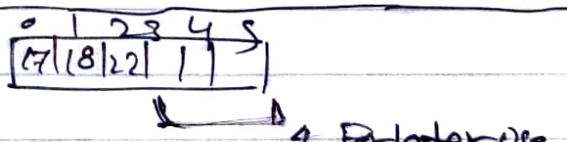
Main Memory (RAM)

(Cannot Extend using realloc)



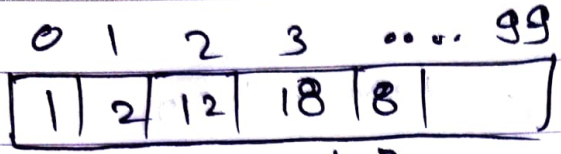
(int *)Malloc(n * sizeof(int))

[If we are using C, we use structures]



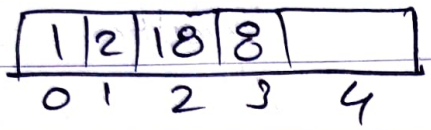

```
Struct myArray {
    int total_size;  (Memory to be reserved)
    int used_size;   (Memory to be used)
```

Insertion and Deletion



(79)

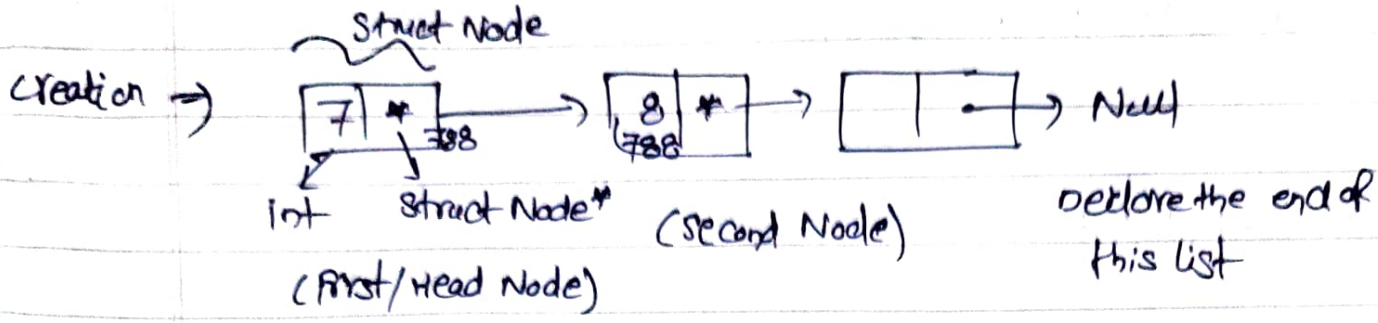
Not Acceptable X



→ After Deletion

Linked List → Linear Data Structure

* Creation + Traversal



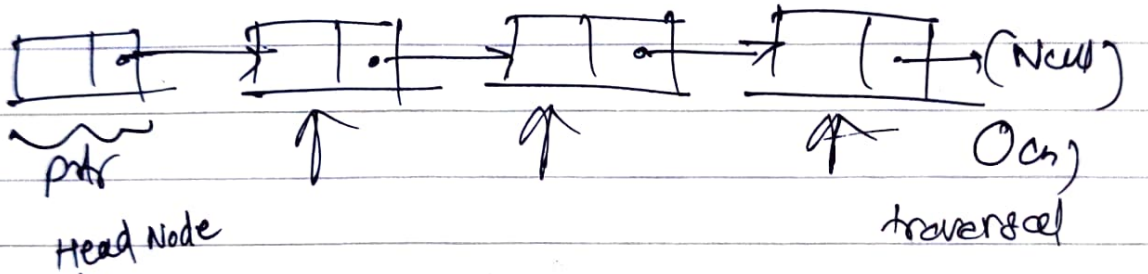
Head = (int*) (Struct Node*) Malloc (sizeof(Struct Node));

(Note: Head is struct pointer)

head → data = 7 ~ (Arrow pointer)

head → next = secondNode;

Traversal →



Insertion →

Case 1 = Insert at First Node / Beginning O(1)

Case 2 = Insert In between O(n) O(1) Best case

worst case

Case 3 = Insert at the End/ last Node O(n) worst case

Case 4 = Insert After a Node O(1)

(Pointer given)

Deletion ⇒

Same cases as Insertion

—x—

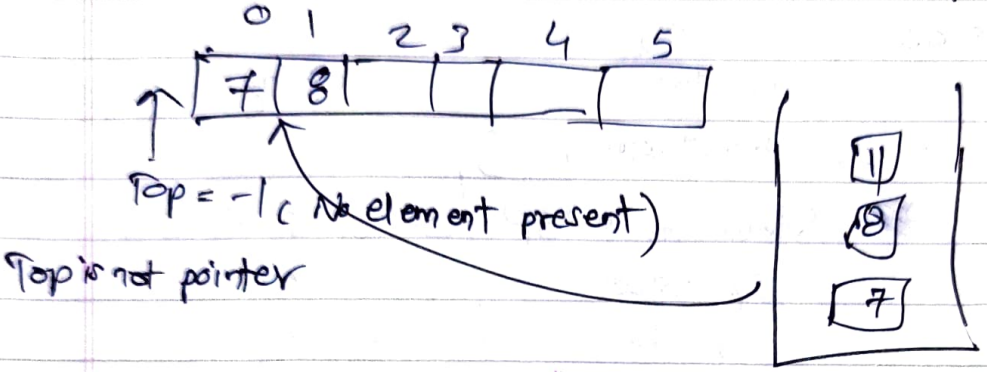
* Circular Linked List ⇒

— last Node points to First Node

* Stack Using an array → Linear Data Structure

→ Stack is a collection of elements following LIFO. Items can be inserted or removed only from one end

pop() → remove element
push() → Insert element



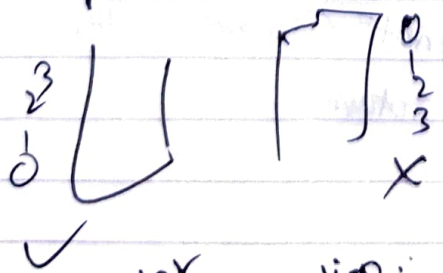
→ Implementation of Stack using Arrays ⇒

1) Need Fixed sized Array creation
2) Top / pointer / value Element (currently int)

3)
```
struct stack {
    int size;
    int top;   //top value
    int *arr;
};
```

4)
```
struct stack S;
S.size = 80;
S.top = -1;
S.arr = (int*) malloc (size of (int));
```
S.size ✗

5) push 7 !



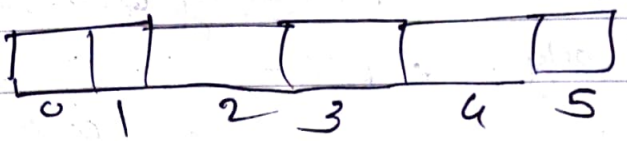
✓

easier
O(1) → Insertion.
Most of the operations happen in

pop?
return arr[top--];

all operations while using Arrays

* Stack Push(), Pop() →



= Cannot push() if full
= Cannot pop() if empty.

```
Struct stack { int size;
               int top;        → creating a stack.
               int * arr; }
```

Struct stack *sp	Struct stack s;
sp→size	s.size;
sp→top	s.size;
sp→arr	s.arr;

1) Push →

```
Struct stack *sp)
sp → size = 8;
sp → top = -1;                    sp→size ×
sp → arr = (int*) malloc (sizeof(int)*8);
// Push starts  if (isFull(sp)) {
                    printf / -ve value // Stack overflow
                  else →
                    sp → top++;
                    sp → arr [sp→top] = val;
```

2) Pop →

```
if (isEmpty(sp)) { printf ("Stack underflow"); ;
                   return -1;
```

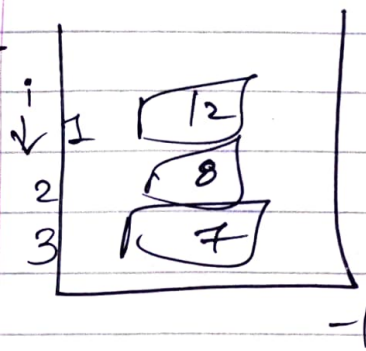
[index starts from 0
position starts from 1]

3) peek →

```
int peek(struct stack *sp; int i)
    if(sp->top - (i+1) < 0)
        Invalid position
        return -1;
    else
        return sp->arr[sp->top-i+1];
```

↑ i is the position of the element of stack (starts from 1 not 0)

eg



Position (i)	Array Index
1	2 = Top - i+1
2	1
3	0

Top 2

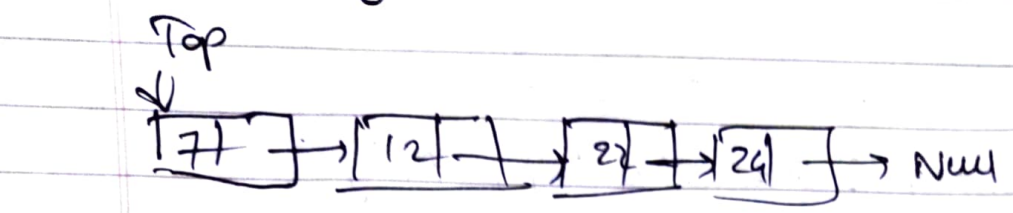
We view Stack from top Always, so that's why we will count the peek function

4) StackTop();

5) StackBottom;

* Time Complexity →
  - StackBottom() → O(1)
  - stackTop() → O(1)
  - push() → O(1)
  - isEmpty() → O(1)
  - isFull() → O(1)
  - pop() → O(1)
  - peek() → O(1)

* Stack using linked list ⇒



Side 1 (Used for push, pop) Side 2

OU) → Insertion, Deletion

```
Struct Node {
    int Data;
    Struct Node * Next;
}
```

- Head is Now referred as top
- StackEmpty Condition → (top == Null)
- stackFull Condition → (Heap Memory Exhausted)
  (you can always set Custom sizeof Stack)
  (ptr == Null)

1) IsEmpty () :→

```
Void isEmpty :
    if (top == Null) {
        return 1;
    }
    else {
        return 0;
    }
```

2) IsFull () ; →

```
Struct Node * n = (Struct Node *) Malloc (sizeof(Node));
    if (n == Null) {
        return 1; }
    else {
        return 0;
    }
```

3) push() :→ [Inserting Node at index 0]

```
Struct Node *n = (Struct Node* ) Malloc(..sizeof(StructNode));
IF(n==Null) {
        printf("Stack overflow");
    }
else {
    n→ data = x;
    n→ next = top;
     top = n;
```

4) pop() :→

```
IF( isEmpty) {
        printf(" Stack underflow");
    }
else {
Struct Node * n = Top;
    top = top→next; int x = n→ data;
    free(n);
    return x;
}
```

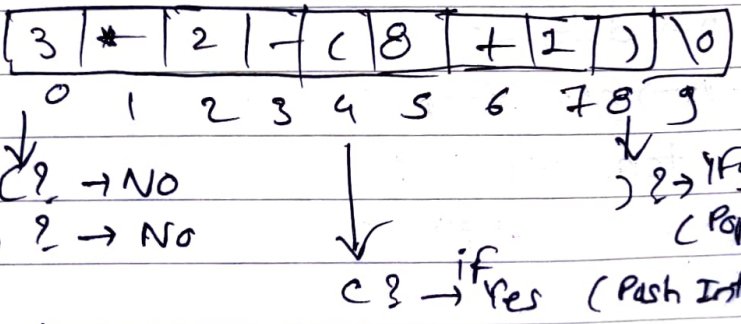
Time complexity = O(1) best / O(n) worst

Parentheses Matching Problem using Stack

((3*2)-1(8-2))

a = &(-3) * 4(8 ;

eg 1 3 * 2 - (8+1) → is Parenthesis Balanced?



Condition for a Balanced Expression →

① while popping - stack should not underflow - Unbalanced

② At End of Expression, stack should be Empty, Must if Not, - Unbalanced

eg 2 1-3) * 4(8) → Unbalanced ①

eg 3 (1*(8-3(7) → Unbalanced ②

eg 4 (7*8)-3(7)) → Unbalanced by ①

7-(8(3+9) + 11+12)) - 8) → 2 pairs pop but Un balanced

* Parenthesis Matching Code ⇒

— use character Array

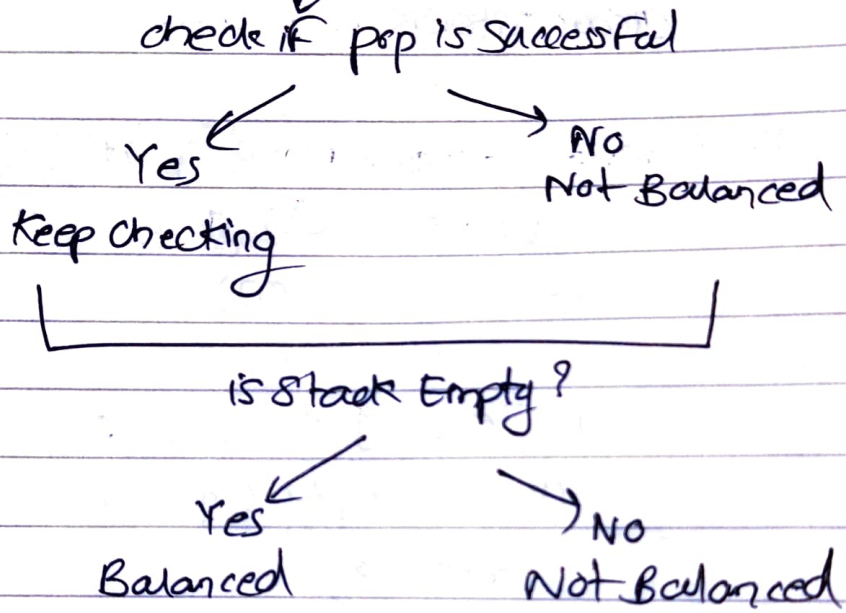
```
int parenthesisMatch (char* exp) {
    struct stack * sp ;
    // create stack

for(int i=0 ; exp[i]!='\0' ; i++) {
    if( exp[i] == '(') {
        push( sp, exp[i]) ;
    }
    else if( exp[i] == ')') {
        if ( isEmpty (sp)) {
            return 0;
        }
        Pop[sp) ;
    }
}

if(isStackEmpty(sp)) { return 1;}
  else { return 0;}}

}
```

Multiple Parentheses Matching using Stack

a = { 7 - (3-2) + [8 + (99 - 11)] }

opening → { ([→ Push

closing → })] → Pop



Infix- Prefix and Postfix

— Notations to write an Expression

1) Infix → a+b, a-b, p/q, x-4 [operand - operator - operand]
→ symbol in between characters
[a*b]

2) Prefix → -ab, +xz, *pb, -pq [operator - operand1 - operand2]
→ character before After symbol
[*ab]

3) Postfix → ab*, pq-, xy+ [operand1 - operand2 - operator]
→ character before symbol
[ab*]

Left to Right

eg1 → Infix ⇒ A*(B+C)*D
③ ① ②
Postfix ⇒ ABC+*D*

Machine Cannot Evaluate Infix So they require Prefix and Postfix

operator	precedence
( )	3
* /	2
+ -	1

Postfix is More Useful than prefix

Q1 x-y*2
Convert to prefix and Postfix

→ Prefix ⇒
Step 1 : Parenthesize expression
(x-(y*2))
(x-[*y2])
-x*y2 → Prefix

→ Postfix ⇒
Step 1 : Parenthesize Expression
(x-(y*2))
(x-[y2*])
xy2*-

(2) p-q-r/a (Always follow operation precedence and Associativity)

Prefix

((p-q)-(r/a))

([-pq]-[/ra])

--pq/ra

Prefix

Postfix

((p-q)-(r/a))

([pq-]-[ra/])

pq-ra/-

Postfix

(3) (m-n) * (p+q)

Prefix

((m-n)*(p+q))

([-mn]*[+pq])

(*-mn+pq)

Prefix

Postfix

((m-n)*(p+q))

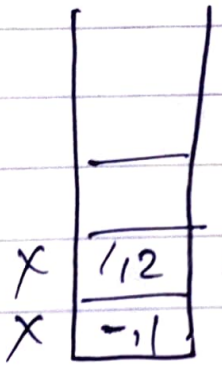
([mn-]*[pq+])

mn-pq+*

Postfix

Infix to Postfix Using Stack →

x-y/z-R*d → Infix



Precedence (next -1)

Postfix Expression :→

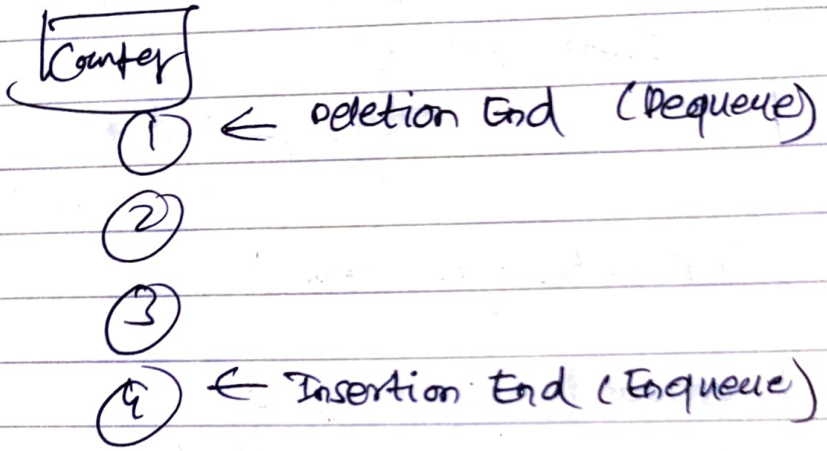
xyz/-Rd*-

+, - → 1
*, / → 2
(→ 3

- Check operand
- Push next operator into the stack
- check precedence of next operator
- If precedence not bigger than Already present element operator then pop previous operator, then check for next operator
- If precedence == previous operator then also pop previous operator of stack and then push current operator
- pop all the operators at the End of the Expression from stack.

Queue Data Structure → Linear Data Structure

Stack → LIFO
Queue → FIFO → First in First out



Queue ADT

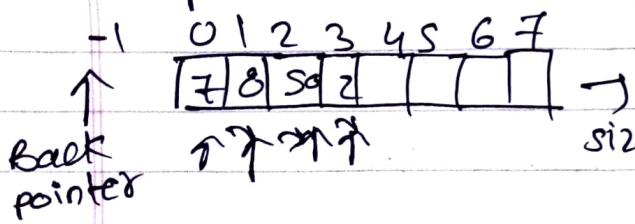
1) Data: Storage
   Insertion End (Pointer)
   Deletion End (Pointer)

2) Methods: EnQueue → onboarding
   Dequeue →
   First Value →
   Last Value →
   Peek the queue Peek(pos) →
   isEmpty →
   isFull →

[Not limited to only counters]
[Queue can be Implemented in Various ways]
   i) Arrays
   ii) Using Linked list
   iii) Using other ADT's

— Implementation of Queue Using Array

Method - II



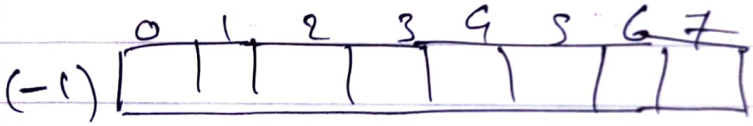
Not useful for Dequeue

— Insert : Increment BackInd — O(1)
(Enqueue) Insert at BackInd

— Remove : Remove element at (0) — O(n)
(Dequeue) Shift all elements

Index

Method - 2



Not useful for Dequeue

(F = -1, B = -1)

Insert : Increment B, Insert at B

Remove : Move F only, not B,
Increment F, remove element at F

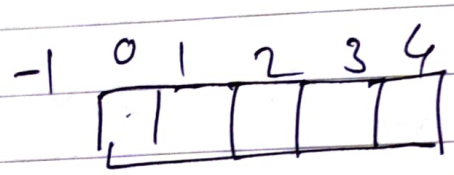
First element = F+1

Rear Element = B

Que Empty Condition ⇒ F == B

Que Full Condition ⇒ B = size-1

Queue using Arrays - Operations

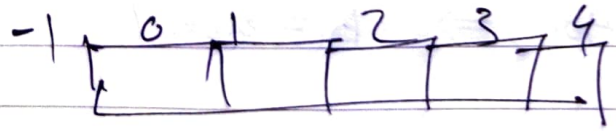


F = -1
R = -1 ⇒ (Move Rear For Insertion)

Index value

```
Struct Queue {
    int size;
    int F; // front value
    int R; // Rear value
    int *arr;
};

int main() {
  Struct Queue q;
    q.size = 10;
    q.F = q.r = -1;
    q.arr = (int *) malloc( q.size(sizeof(int));
}
```

① EnQueue ⇒



F = -1
R = -1

```
Void EnQueue(Struct Queue* q
if(IsFull(q)) {  int val){
    printf("Queue overflow");
} else {
    q->r = q->r+1;
    } q->arr[q->r] = val;
}
```

(2) Dequeue ⇒

(return Integer)

```
int Dequeue( Struct Queue *q) {
    int a = -1;  // return wrong value, failed
    if( q->f == q->r) {
        printf("No Element to Queue");
    } else {
        q->f++;
        a = q->arr[q->f];
    }
    return a;
}
```

→ IsEmpty()

```
// Full() ⇒
if(q->r == q->size-1)
    return 1;
```

* Drawbacks ⇒

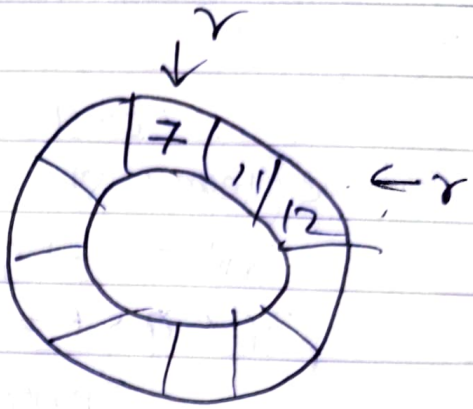
1) Cannot used space Efficiently

Circular Queue

⇒ Circular Increment ⇒

— i = i+1 ; is Linear Increment

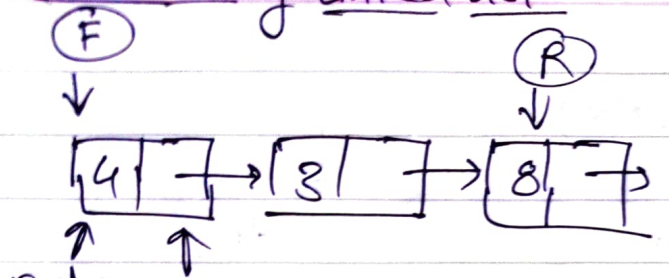
— i = (i+1) % size ⇒ Circular Increment



— Void enqueue (Struct queue *q, int val) {
IF((q→r+1) % q→size == q→F) {
Printf ("Queue is Full"); }
else {
q→r = (q→r+1) % q→size ;
q→array[q→r] = val ;
}
}

— int dequeue(Struct queue *q) {
int val = -1;
IF(q→r == q→F) {
printf("Queue is Empty");
} else {
q→F == (q→F+1) % q→size
val = q→arr[q→F];
}
return val;
}

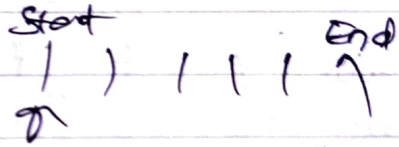
Queue using Linked list



Linked list is like a chain

→ Front and Rear are pointers this time [F & R]

Queue Basics



— Void equeue (Struct queue Node *F, int Val) {

Struct Node *r

Struct Node* n = (Struct Node*) malloc (sizeof (Struct Node));

if (n == Null) {

printf ("Failed memory Allocation");

} else {

Node → data = Val;

n → next = Null;

if ((F == Null) {

F = r = n;

}

else {

r → next = n;

r = n;

}

}

we wont always do r → nxt = Null

R → Null

r → Null

[1] Special Case

isFull (

n == Null

)

isEmpty (

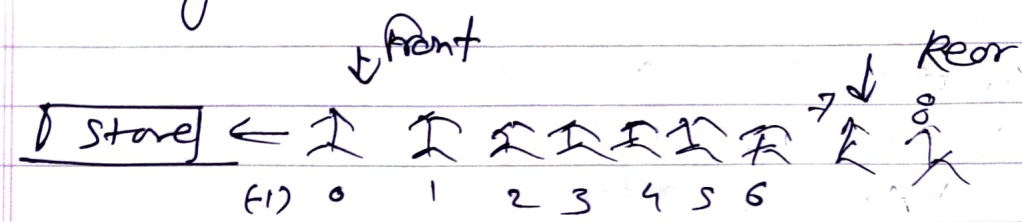
Front == Null

)

```
int dequeue ( N* F, [redacted] ) {
    int val = -1;
    N* ptr = F;
    // check for is Empty
    F = F -> next;
    val = ptr -> data;
    free(ptr);    // Imp Step
    return val;
}
```

If Not queue Empty

Double Ended Queue (DEQueue)



Queue ⇒ FIFO

DEQueue ⇒ ×

- Dequeue
  - → Insertion from Rear + Front
  - → Insertion from Rear + Front. → we can do ALL possible operations.

2 types of DEQueue

- Restricted Input DEQueue
  (Insertn from Front is not Allowed)
- Restricted output DE-queue
  (Deln from rear is not Allowed)

Sorting Algorithms

19827 ...

two ways of sorting

1) Ascending order — 12789
2) Descending order — 98721

Sorting: Newest!
Top Rated!

* Analysis Criteria for Sorting Algorithm ⇒

1) Time Complexity
2) Space Complexity → Inplace sorting Algorithm
3) Stability ⇒ (6)126 ⇝ 1266
4) Internal sorting Algo = All data is loading into the Memory (RAM)
   External sorting Algo = All data is not loaded into the Memory (RAM)
5) Adaptive ⇒ Already Sorted Data takes less time
6) Recursive = if it uses Recursion
   Non Recursive = if it does not use Recursion

Lighter Elements ←

→ Heavier Elements

Bubble Sort

len = 6 (n)
Pass = 5 (n-1)

0	1	2	3	4	5
7	11	9	2	17	4

swap

→ 2, 4, 7, 9, 11, 17

→ largest element to the last

7 11 9 2 17 4 → 0,1

First Pass ⇒ 7 9 ~~11~~ ~~2~~ 11 17 4 → 1,2

7 9 2 11 17 4 → 2,3

[5 comparisons]
5 possible swaps

7 9 2 11 17 4 → 3,4

7 9 2 11 4 (17) → 4,5 ⇒ 1st Pass Completed

skip this

(n-1)

2nd Pass ⇒ 7 9 2 11 4 17 → 0,1

[4 comparisons]
4 possible swaps

7 2 9 11 4 17 → 1,2

7 2 9 11 4 17 → 2,3

7 2 9 4 (11) 17 → 3,4

can't compare
skip this

(n-2)

3rd Pass ~~2 7~~ 9 4 11 17 → 0,1

[3 comparisons]
3 possible swaps

2 7 9 4 11 17 → 1,2

2 7 4 (9) 11 17 → 2,3

can't compare
skip this.

(n-3)

4th Pass ⇒ 2 7 4 9 11 17 → 0,1

[2 comparisons]
2 possible swaps

2 (4) 7 9 11 17 → 1,2

can't compare

Skip and stop the program

(n-4)

5th Pass gives Sorted Array ⇒ [2 4 7 9 11 17] → 0,1

[1 comparison]
1 possible swap

(1)

- Total no of Comparisons = $\frac{n(n-1)}{2} = O(n^2)$ worst case Tc

length (n) = no of elements of array
passes (p) = $n-1$

- Is Bubble Sort Stable? Yes Bubble Sort is Stable sorting Algo order of sorted array same as order of Input Array

- Is Bubble Sort Recursive? NO

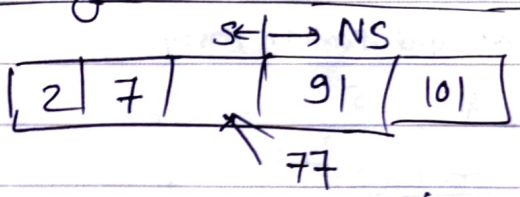
- Is Bubble Sort Adaptive?
  By default NO But we can Make it Adaptive

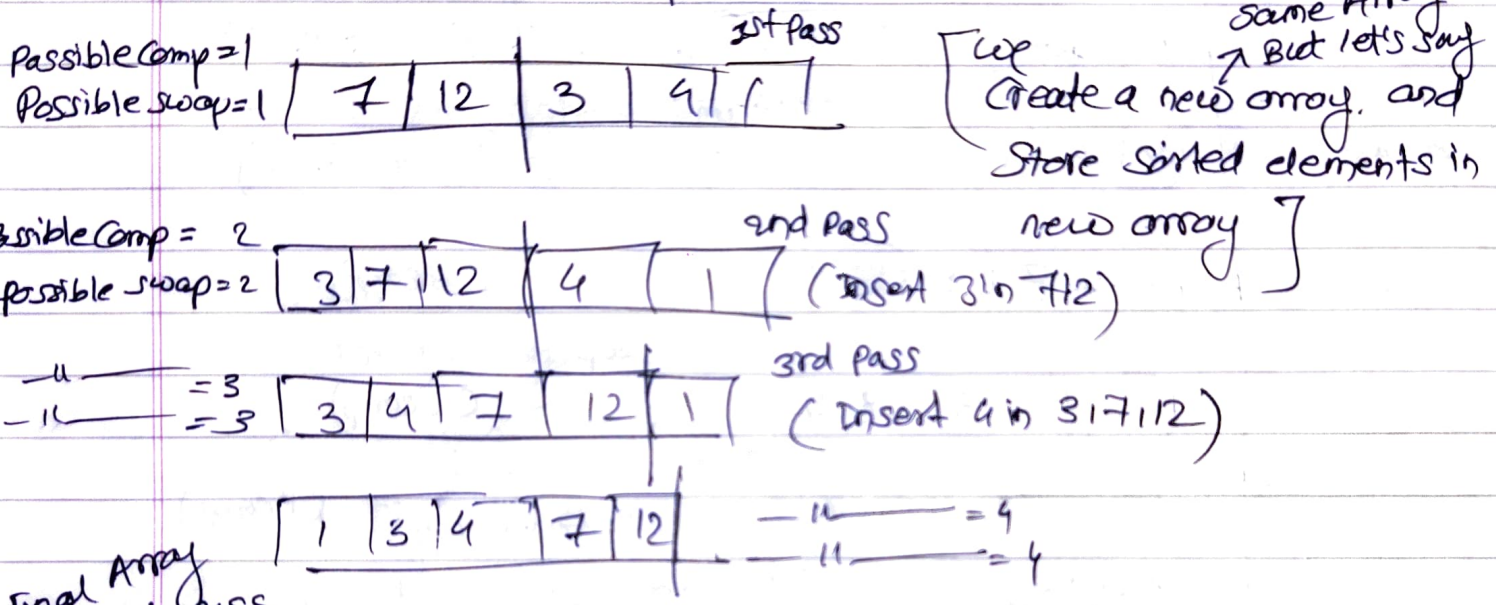
- Time Complexity = $O(n^2)$
- Space Complexity = $O(1)$ — Do not take Additional Space

(We can convert time Complexity from $O(n^2)$ to $O(n)$ to Already sorted elements using Bubble Sort and Make it Adaptive)

Insertion sort

- Inserting an element into the Sorted Array
- Start Inserting from last ~~or first~~





we Create a new array. and Store Sorted elements in new array] ↗ same Array But let's say

Array size = n (length)

① Total Passes = $n-1$

Total Possible Comparison = $1+2+3+\cdots(n-1) = n(n-1)/2$

Total Possible swaps = $1+2+3+\cdots(n-1) = n(n-1)/2$

② Time Complexity = $O(n^2)$ (worst case)

= $O(n)$ (Best case) (Array is Already sorted)

③ Stable ? ⇒ Yes, it is stable

④ Adaptive ? ⇒ Yes, it is Adaptive

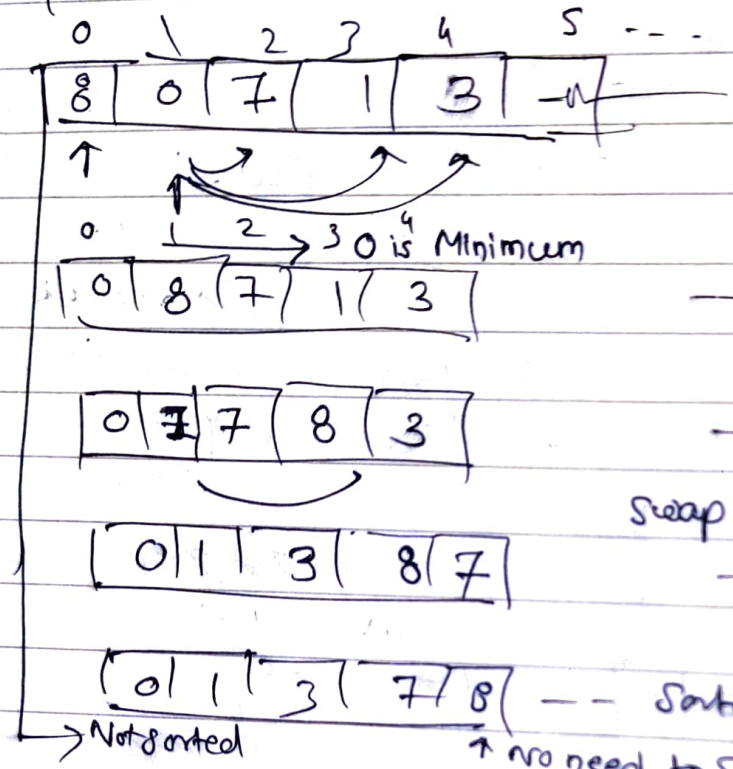
- Intermediate result is not useful.

Selection Sort

n = 5

0	1	2	3	4
8	0	7	1	3

- Identify the minimum Element and sort in Array



length of Array (n) = 5 , No of Passes = n-1

total Possible Comparisons = $1+2+\cdots+(n-1) = \frac{n(n-1)}{2} = O(n^2)$

Time Complexity = $O(n^2)$

Min. No of Swaps = 0

Total Maximum Swaps = n-1

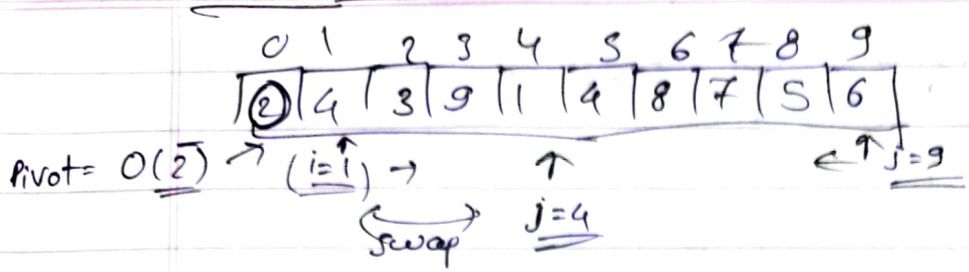
Stability = (maintains order of elements) Not a Stable Algorithm.

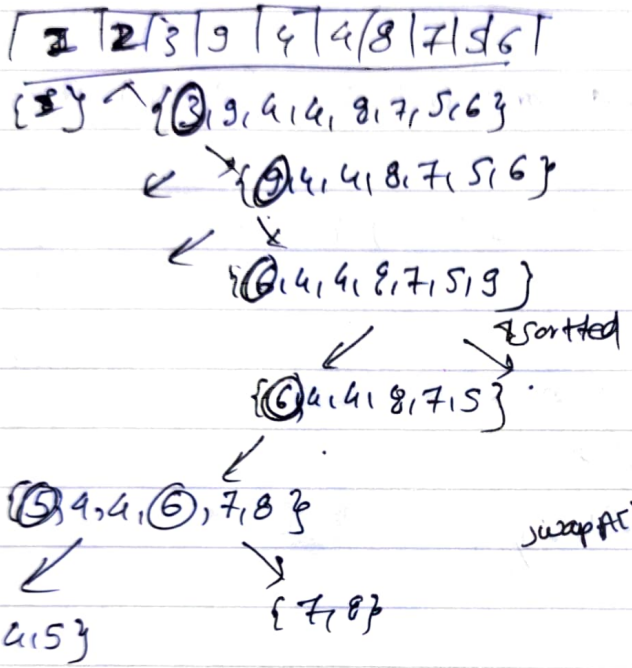
Adaptive = not an Adaptive Algo

Advantage = Sorting in min no of swaps.

Quick Sort Stable? Not stable
Quick Sort Inplace Algo? Yes

Quicksort





⇒ If j<i = Pivot interchange with j

Partitioning ⇒
1) i = low
2) j = high
(Always take Pivot as low)
(3) i++ until element > pivot found
(4) j-- until element < pivot found
(5) Repeat (4)(5) until j ≤ i
(6) Swap Pivot with A[j]

swap A[i], A[j]

— Partitioning Position Element Identification — Split Array into different parts
— Pivot element placed such that all left and Right sided elements are in continuous order (Smaller — Pivot - Bigger)
— Quick sort is Recursive (applies on Subarrays)
— Quick sort don't use Extra Memory Space for Array.

* Quick sort Analysis → (Worst Case = Already sorted Elements)
• (Time Complexity of Worst Case = O(n²))

Noor
Comp Algo Partitioning = Linear function of n = T = (n-1)(K1n+K2)

• Best Case Analysis = O(nlogn)

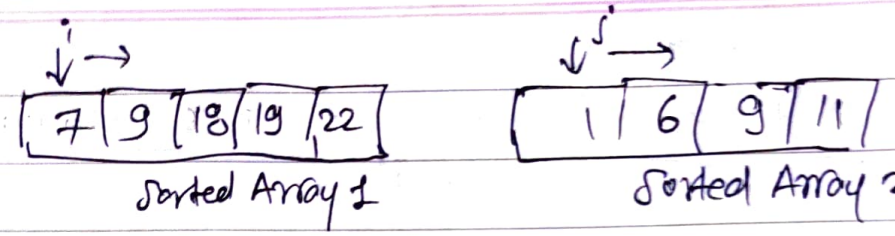
16 → 1 × T(n)
8 × 7 → 2 × (T(n/2))
4 3 3 3 → 2² × T(n/2²)
2 1 1 1 1 1 → 2³ × T(n/2³)

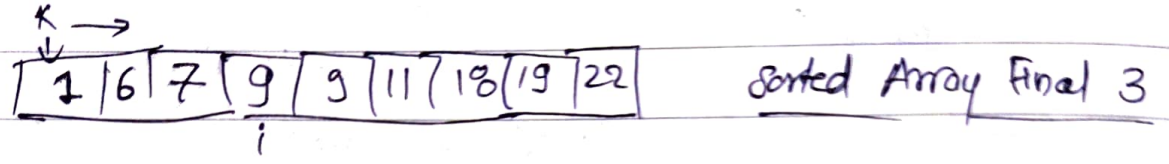
Partition time T(n) = K1n + K2
Total time = $(K_1n+K_2) + 2\left(\frac{K_1n}{2}\right) + 2^2\left(\frac{K_1n}{2^2}\right)$ ... k time (height of tree) = logn

= nxK1n + K
= lognxn = O(nlogn)

Merge Sort

Start condition ⇒ $i=j=k=0$

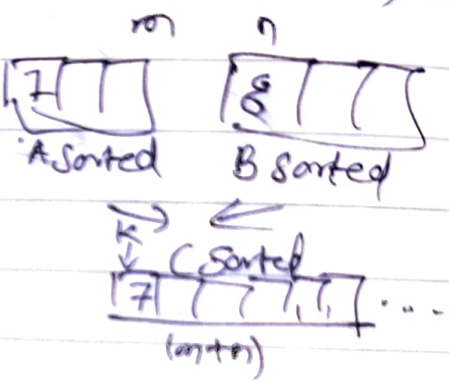




if $i=j$, then fill any one i or j Prefer i (Merging Procedure)

Different Arrays ⇒

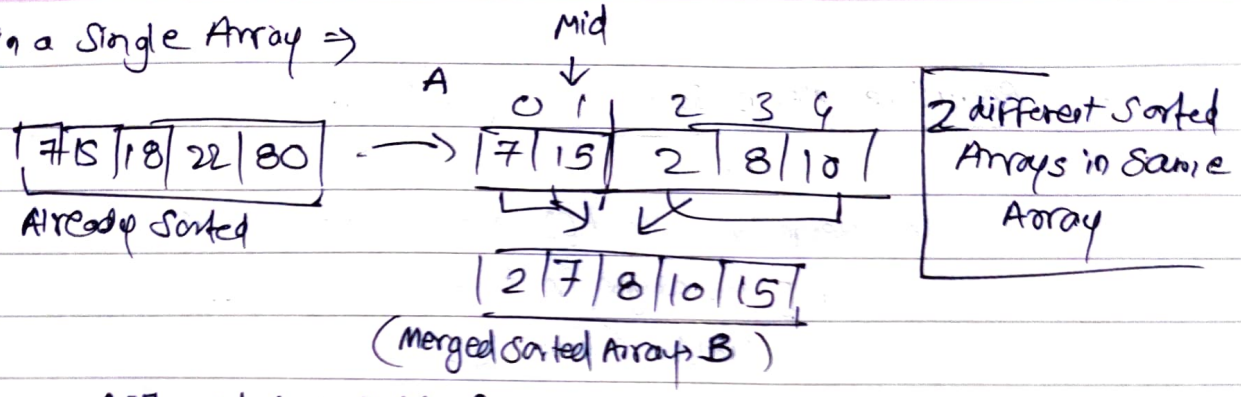
```
Void Merge (A[], B[], C[], m, n) {
    int i,j,k;
    i=j=k=0;
    while( i<m && i<n) {
    if (A[i] < B[j]) {
            C[k] = A[i];
             i++;
             k++;
    } else {
            C[k]= B[j];
             j++, k++; }
    while( i<m ) {
            C[k] = A[i];
             k++;
             i++; }
    while( j<n) {
            C[k] = B[j];
             k++;
             j++; }
```



Copy All remaining elements from A to C

Copy All remaining elements from B to C

- Merging in a Single Array ⇒




```
void Merge (A[], mid, low, high) {
    int i=j; k;   int B[high+1];
    i = low;
    j = mid+1;
    k = low high;
    while (i<mid && j< high) {
        if (A[i] < A[j]) {
            B[k] = A[i];
            i++;
            k++; }
        else {
            B[k] = A[j];
            j++;
            k++; }
        while (i<=mid) {
            B[k] = A[i];
            k++;
            i++; }
        while (j<high) {
            B[k] = B[j];
            k++;
            j++; }
        for (i = low, i<= high, i++) { A[i] = B[j]; }
```

→ First half of Sorted Array

→ Second half of Sorted Array

- **Recursive Merge Sort →**

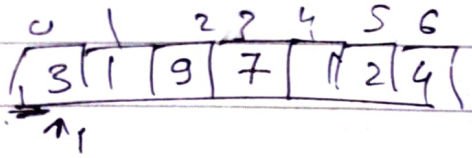
```
void MergeSort(int A[], int low, int high){
    if(low<high){
        mid = (low+high)/2;
        mergeSort(A, low, mid);
        mergeSort(A, mid+1, high);
        merge(A, low, mid, high);
    }
}
```

→ Is Merge Sort Adaptive ? Not Adaptive

→ Is Merge Sort Stable ? Yes

Count Sort

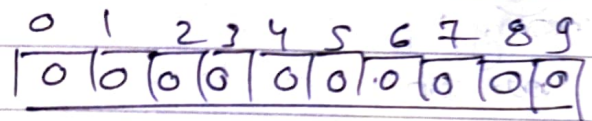
It is one of the fastest



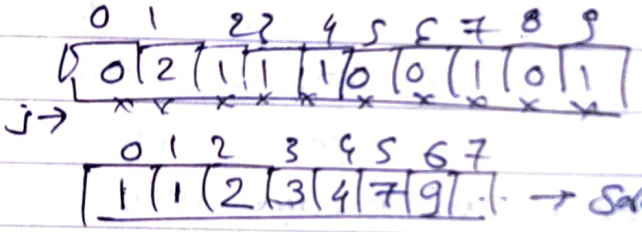
→ Array

→ (9) Max element

- Identify Maximum Element
- create Count Array of size of Maximum element ⇒ let's say 10



- Initialize Count Array with 0 which is Auxiliary Array.
- Traverse the unsorted Array and increment 3rd Index by 1



Traverse and Copy these elements into Another Array

→ Sorted Array

→ we used dynamic Memory Allocation to create Auxiliary Count Array

→ Count Sort is Stable

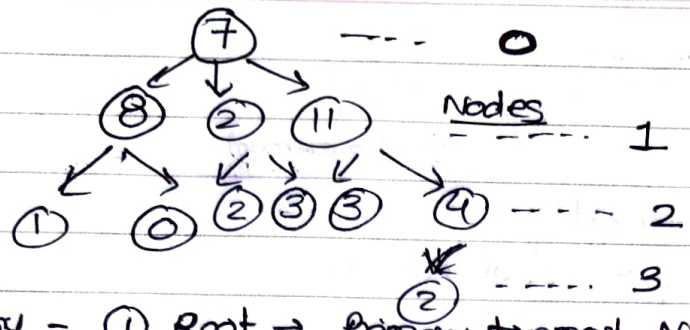
→ Count Sort is Adaptive

→ Time Complexity = O(n+k) k is range of input values

→ Disadvantage is it uses Extra space means Space Complexity is high

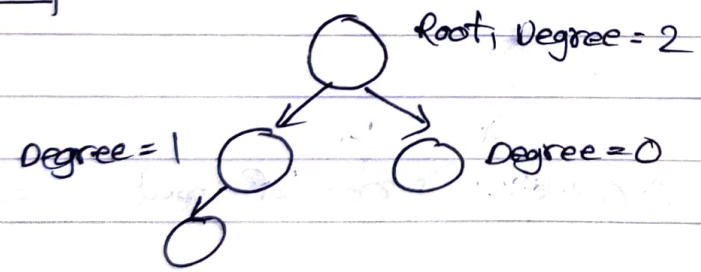
→ Best time Complexity for any Sorting Algorithm But at the cost of Extra Space

Trees → Ideal for representing hierarchical data / Desktop filesystem / organizational etc
→ Non-linear Data Structure



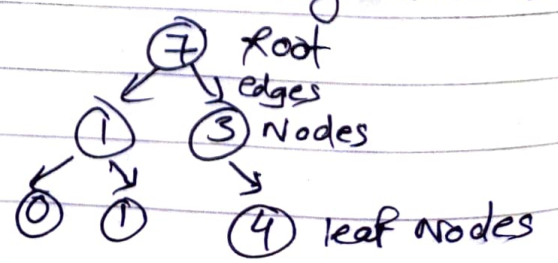
Terminology = ① Root → Primary topmost Node
② Parent → Node which connects to the child
③ Child → Node which is connected by Another Node is it's child.
④ / Leaf (External) → Nodes with no children / without children
⑤ Internal Node → Nodes with atleast 1 child.
⑥ Depth of a Node → No of edges from root to that Node / deepest leaf Node
⑦ Height of tree - No of edges from node to deepest leaf!
⑧ Siblings → Nodes Belonging to the same parent!
⑨ Ancestors / → Descendents
⑩ Degree → How Many child Nodes are connected ?

Binary Trees → Tree which Every Node has degree less than 2 (or) 2 in



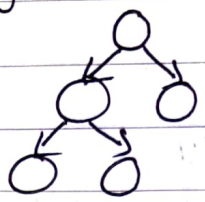
(Binary tree is a tree which has atmost 2 children for all the nodes)

- Tree is made up of nodes and edges
- n nodes = n-1 edges
- Degree = no of direct children (for a Node)
- Tree of Degree 2 is called Binary Tree (Nodes can have 0, 1, 2 children)
- Degree of tree is highest degree of a Node among all the nodes present in the tree

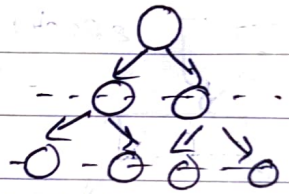


- Types of Binary Trees →

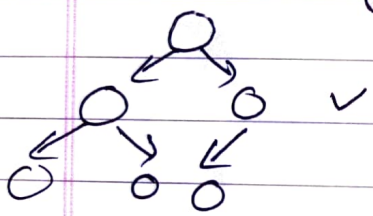
① Full or Strict Binary Tree = All Nodes have either 0 or 2 children



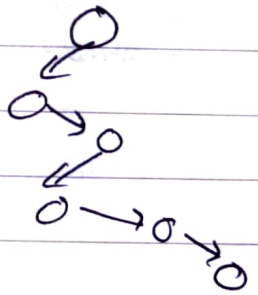
② Perfect Binary Tree = All Internal Nodes have 2 children + All leaf nodes are on same level



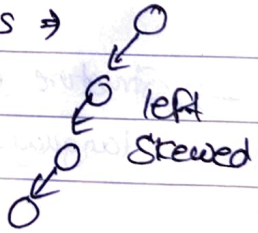
③ Complete Binary Tree = All levels are completely filled except possibly the last level + last level must have it's keys as left as possible

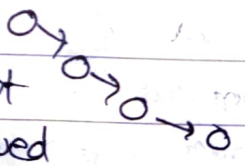


④ Degenerate Tree = Every Parent Node has Exactly one child



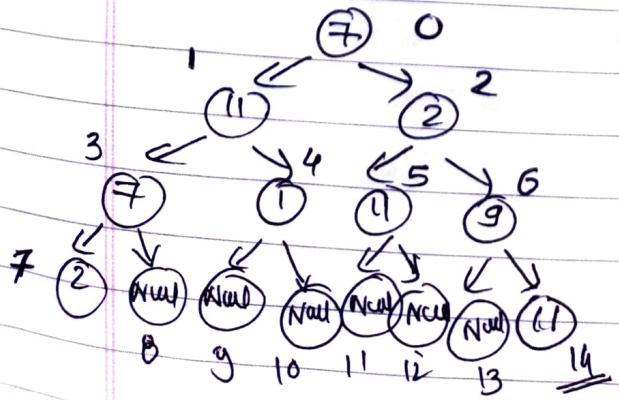
⑤ Skewed Trees →





- Representation of a Binary Tree →

① Array Representation = (Not used Popularly)

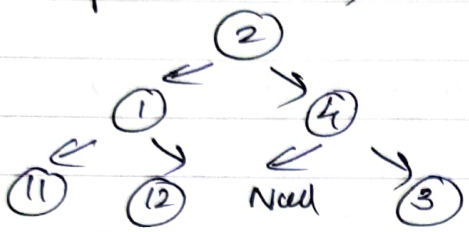


Arr = 

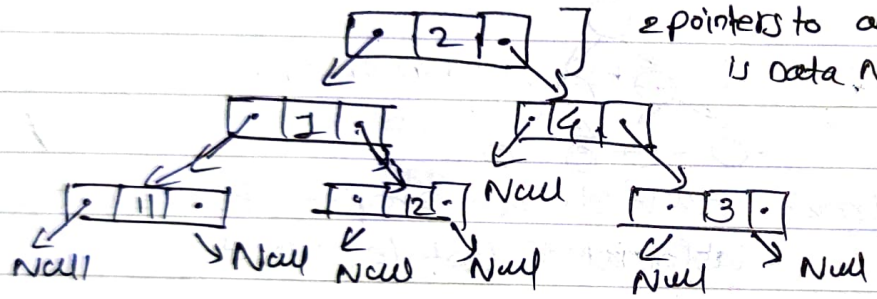
Size is Array is Needed

(Insertion, Deletion and Extension of Tree is Difficult, hectic and Nearly Impossible)

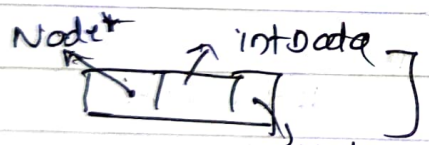
(2) Linked Representation → (Efficient Implementation)



- helps to visualize binary tree quickly
- uses doubly linked list for linked representation
- linked list is linear, linked representation is non-linear



(Pointer to Another structure)




```
Struct Node {
    int data;
    Struct Node * left;
    Struct Node * right
    };
```

→ Structure of a Node in C language

→ ×

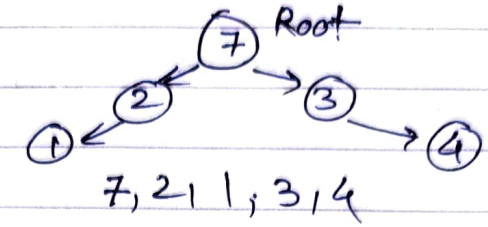
→ Linear Data Structure: Front to Rear / Rear to Front ⇄ Array!

- Traversal in a Binary Tree →

→ Non-Linear Data Structure : Tree

① Preorder Traversal ⇒

- Root, left subtree ; Right subtree



7, 2, 1, 3, 4

② Postorder Traversal ⇒

- left subtree - Right subtree - Root

1, 2, 4, 3, 7 ✗

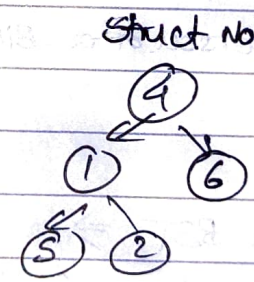
1, 4, 3, 2, 7 ✓

③ Inorder Traversal ⇒

- left subtree - Root - Right subtree

1, 2, 7, 3, 4

- Binary Tree - PreOrder Traversal

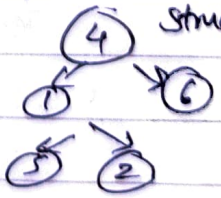


4, 1, 5, 2, 6

```
Void preorder( Struct Node * root) {
    if(root != Null) {
        printf(root → data);
        preorder ( root → left);
        preorder ( root → right); }
```

(uses Recursion)

- Binary Tree - Postorder Trorersal

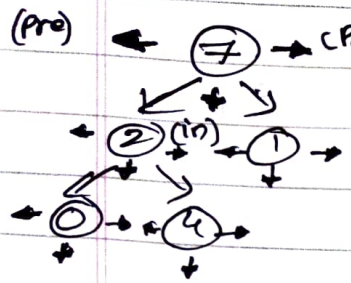


left, Right, Root

5, 2, 1, 6, 4

```
Void postorder( Struct Node* root) {
    If (root != Null) {
        preorder ( root → left) ;
        posterder ( root → right) ;
        printf ( root → data);
```

- How to find Postorder, Inorder and Preorder ⇒



Preorder ⇒ 7 2 0 4 1

Postorder ⇒ 0 4 2 1 7

Inorder ⇒ 0, 2, 4, 7, 1

---

* Binary Search Tree ⇒
  - It is a type of Binary tree!

- Properties = All nodes of
  1) left subtree are lesser.
  2) All nodes of Right subtree are Greater.
  3) left and Right subtrees are also BST
  4) There are no duplicate nodes.
  5) Inorder traversal of a Binary Search Tree gives an ascending sorted Array.

(1) Searching in a BST ⇒

Time Complexity = ($\log n \leq h \leq n$)

```
node* search(Node* root, int key) {
    if root(==null)
        return null;
    if (root→data==key)
        return root;
    else if (root→data > key)
        return search(root→left, key)
    else
        return search(root→right, key)
```

(2) Iterative Search in BST ⇒

```
struct Node* SearchIter(struct Node* root, int key) {
while (root != NULL) {
    if (key == root->data) {
        return root; }
    else if (key < root->data) {
        root = root->left; }
    else {
        root = root->right; }
    }
return NULL;
}
```

(3) Insertion in a BST ⇒ 1) No Duplicates Allowed

(4) Deletion in BST ⇒

(i) Case 1 :- The node is a leaf Node

(ii) Case 2 :- The node is a non-leaf node

(iii) Case 3 :- The node is the root node

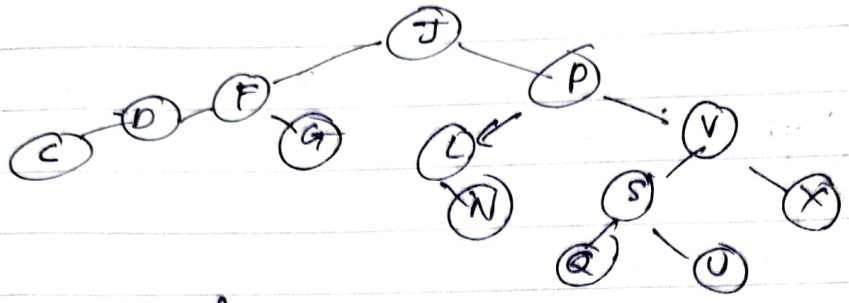
- node is leaf node ⇒
  - Search the node
  - Delete the node

- Node is non-leaf (non-root) node ⇒
  - Inorder pre / Inorder post

- Node is a root node ⇒ - Search for the node
  - Replace with leaf node
  - If Condition is only Inorder Post; condition Becomes Complex
  - Applies to Node in between BST too.

- Search for Inorder pre and Inorder post.
- keep doing this until tree has no empty nodes

AVL Trees → self Balanced BST



Why we need AVL trees?

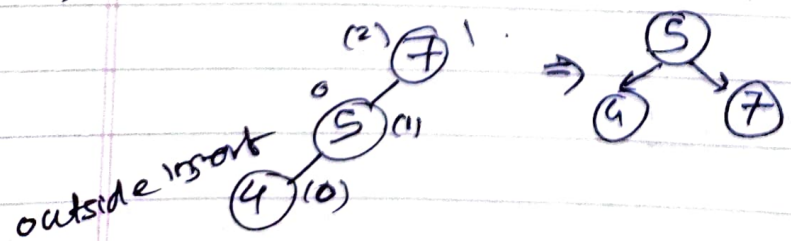
- Almost All operations in BST are of order O(h) where h is height of the tree
- If we don't plan our tree property, this height can get as high as n where n is the no. of nodes in a BST (skewed tree)
- To Guarantee an Upper bound of O(logn) for all these operations, we use Balanced Trees
- AVL avoids a BST to be turned into a list type

What is an AVL Tree?

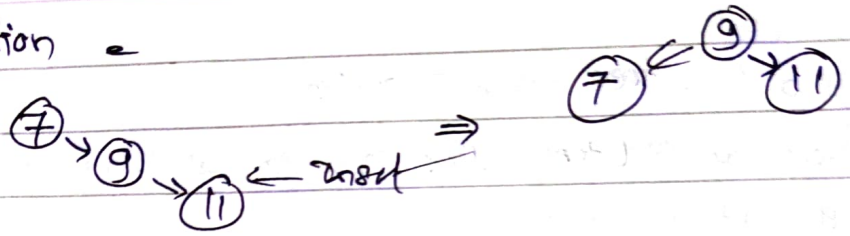
- Height balanced binary search trees.
- Height difference between heights of left and right subtrees for every node is less than or equal to 1
- Balanced Factor = Height of right subtree − Height of left subtree
- Can be −1, 0 or 1 for a Node to be balanced in a Binary search tree
- Can be −1, 0 or 1 for all nodes of an AVL tree.

- Insertion and Rotation in AVL → (Rotation is Imp because maintaining Balanced Factor)

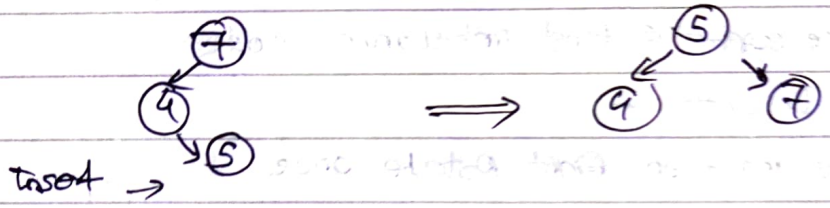
1) LL Rotation =



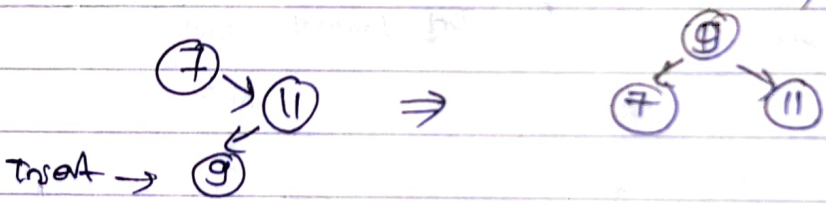
2) RR Rotation :



3) LR Rotation = (double rotation requires) (Anti then clockwise)



4) RL Rotation ⇒ (Requires double rotation) (clockwise then Anticlockwise)

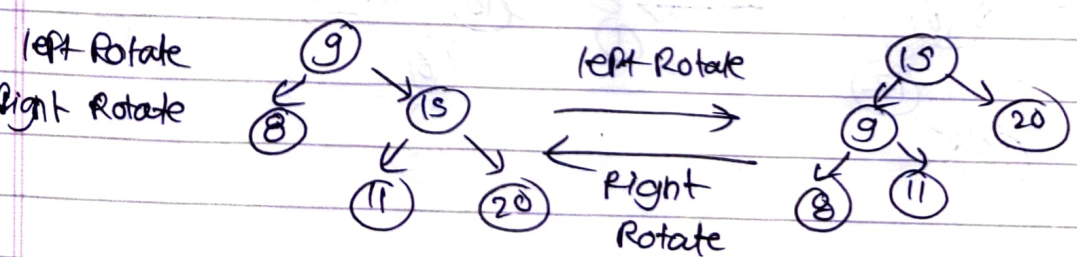


- Rotation in AVL trees with Multiple Nodes →

Rotate operations =

we can perform rotate operations to balance a binary search tree such that the newly formed tree satisfies all the properties of a BST.

1) Left Rotate on a Node → Node is moved towards the left
2) Right Rotate on a node → Node is moved towards the right.



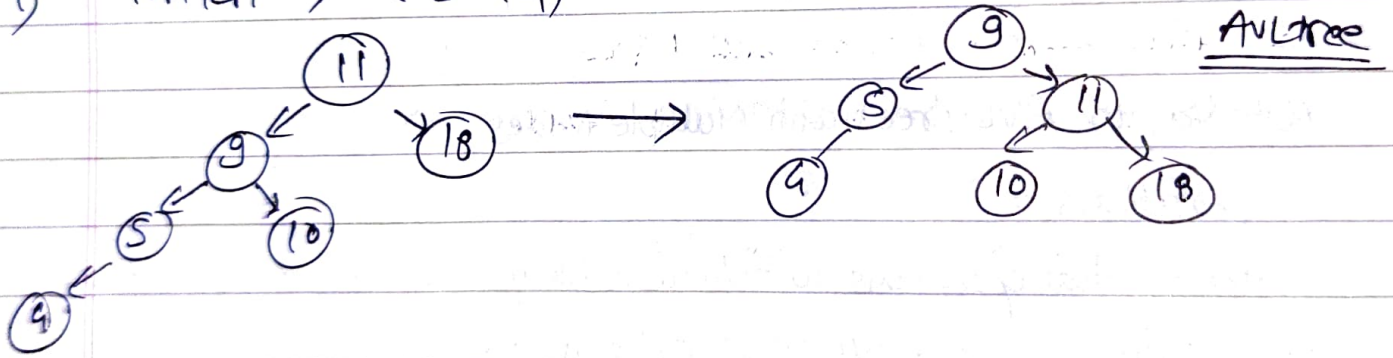
1 & 3 are same trees

- Balancing a AVL tree after Insertion ⇒

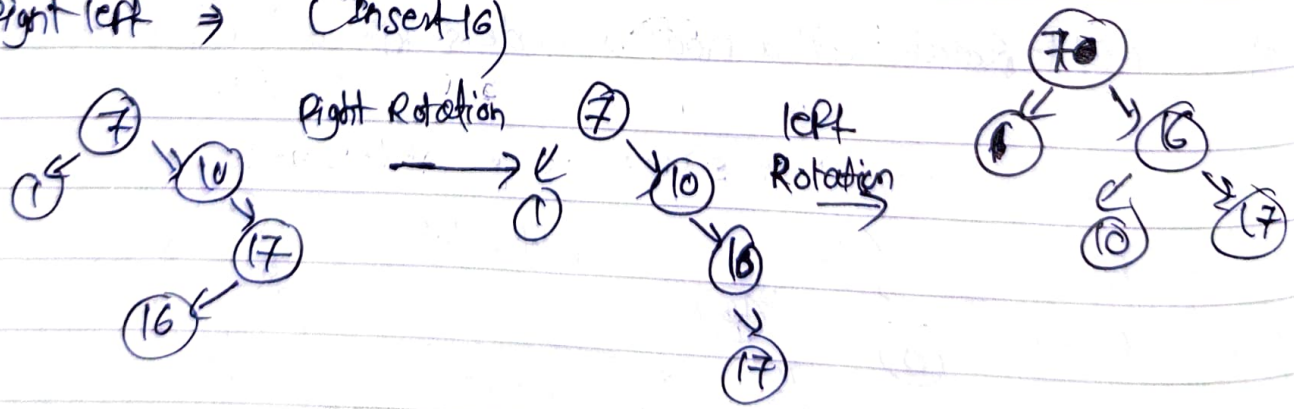
In order to Balance an AVL tree after insertion, we can follow rules –

1) For a left-left Insertion =
   Right Rotate once wrt the first imbalanced node
2) For a Right-Right Insertion =
   Left Rotate once wrt the first imbalanced node
3) For a Left-Right Insertion –
   Left Rotate once and then Right Rotate once
4) For a Right-left Insertion =
   (Right Rotate once and then (left Rotate once
   (child of first imbalanced node in path of Inserted node) First Imbalanced Node)

1) left left ⇒ (Insert 4)

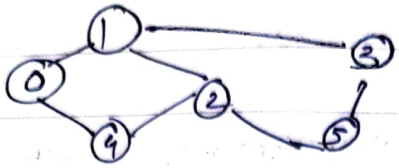


2) Right left ⇒ (Insert 16)



- Example of Non linear Data Structure
- Collection of Nodes/Vertices Connected through edges

Graphs →

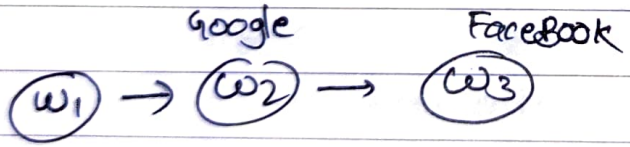


Nodes + Edges
Vertices

- Tree is a Type of Graph

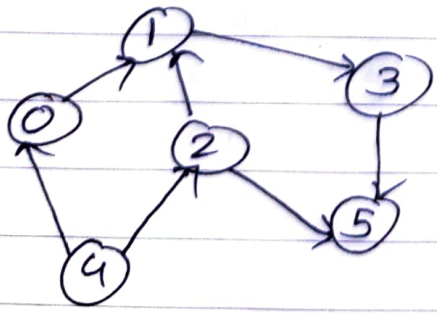
- Formal Definition of a Graph =

1) A Graph G = (V, E) is a Collection of Vertices and edges connecting these vertices
2) Used to model paths in a city, Social Networks, website Backlinks, internal Employee network etc
3) A vertex or Node is one fundamental unit/ entity of which graphs are formed.
4) An edge is uniquely defined by it's 2 endpoints.
5) Directed Edge - one way connection (follow)
6) Undirected Edge - Two way Connection (follow back)
7) Directed Graph - All directed Edges
8) Undirected Graph - All Undirected Edges.



- Indegree and outdegree of a node →

1) Indegree = No of edges going out of the node
2) Outdegree = No of edges coming into the node



- FaceBook - a graph of users

- Graphs are used to model relationships between nodes
- We can Apply graph algorithms to suggest friends to people, calculate no. of mutual friends etc
- Other examples of graph include result of a web crawl for a website or for the entire world wide web, city routes etc
- Is site A linking to site B ?

- Representation of Graph ⇒

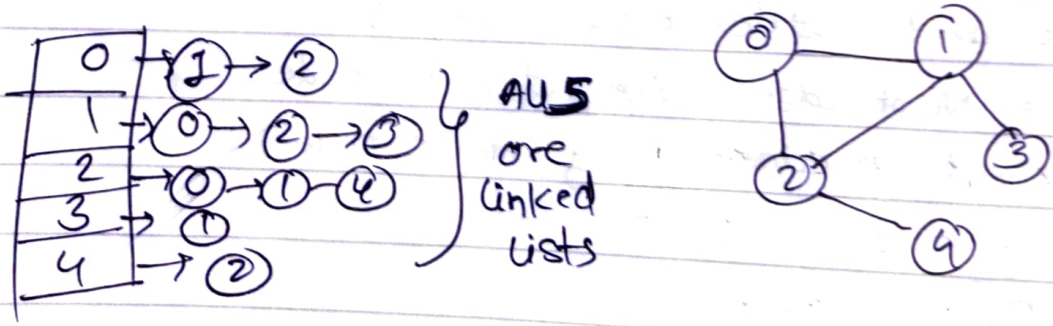
★ v1) Adjacency List = Mark the nodes with the list of it's neighbors.

★ v2) Adjacency Matrix - $A_{ij} = 1$ for an edge between i and j, 0 Otherwise!

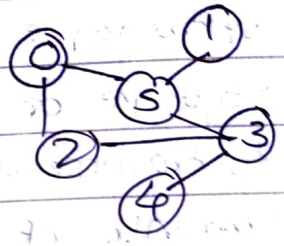
3) Edge set = Store the pair of nodes/ Vertices connected with an edge. eg {(0,1), (0,4), (1,4)}

4) Other Implementations to represent a graph also Exists. For eg. Compact list representation, Cost adjacency list, Cost adjacency matrix etc
(Entire Graph is stored in 1D Array)

- Adjacency list ⇒
- Mark the nodes with the list of it's neighbors



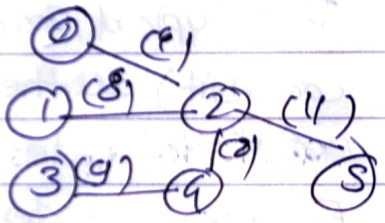
- Adjacency Matrix ⇒
- Aij = 1 for an edge between i and j, 0 otherwise!



	0	1	2	3	4	5
0	0	0	1	0	0	1
1	0	0	0	0	0	1
2	1	0	0	1	0	0
3	0	0	1	0	1	1
4	0	0	0	1	0	0
5	1	1	0	1	0	0

— Applies For Cost Adjacency Matrix where 1 will be replaced by some edge/ value eg city, state, etc

- Cost Adjacency Matrix ⇒
- Aij = Cost For an edge between i and j, 0 otherwise!



— If the cost can be 0 :
Aij = Cost For an edge between i and j, -1 otherwise

	0	1	2	3	4	5
0	0	0	4	0	0	0
1	0	0	8	0	0	0
2	4	8	0	0	2	11
3	0	0	0	0	9	0
4	0	0	2	9	0	0
5	0	0	11	0	0	0

- Graph traversal → (uses Queue Data Structure)

- Graph traversal refers to the process of visiting (checking and/or updating) each vertex (node) in a graph.
- Sequence of steps known as Graph traversal algorithm can be used to traverse a graph.
- Two Algo of Graph traversal are :→
  1) Breadth First Search (BFS) → uses Queue
  2) Depth First Search (DFS) → uses Stack

↓

⊙ Exploring a Vertex (Node) →

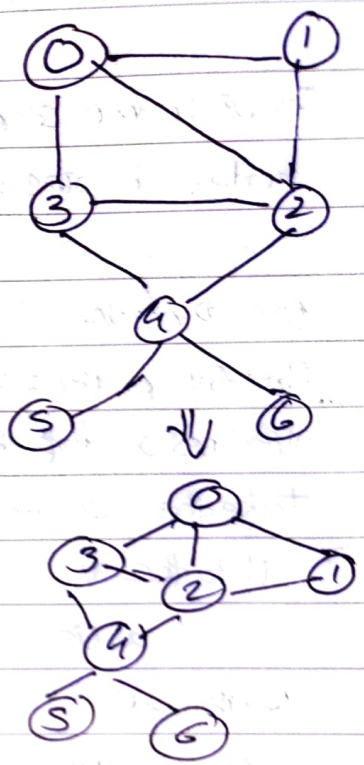
- We have already looked into tree traversal algorithms in our section on trees.
- In a typical Graph traversal Algorithm, we traverse through (or visit) all the nodes of a graph and add it to the collection of visited nodes
- Exploring a vertex in a graph means visiting all the connected vertices.

#1) BFS Algo in Graphs ⇒

- In BFS, we start with a node and start Exploring its connected nodes. The same process is repeated with all the connecting nodes until all the nodes are visited

- Imp in BFS =
  - We can start with any vertex
  - There can be multiple BFS results for a given graph
  - the order of visiting the vertices can be anything.
  -

- BFS spanning Tree ⇒



- Consider the graph shown at the right!
- We can start with any source Node
- Let's start with 0
- Try to construct a tree with 0 as the root.
- Mark all the sideways or duplicate edges (above a node) as dashed
- This constructed tree is called as BFS Spanning Tree
- Level order traversal of a BFS spanning tree is a valid BFS traversal of a graph!

- BFS Traversal ⇒
  - Consider the graph shown at the right!
  - We can start with any source node
  - Let's start with 0 and insert it in the queue.
  - Visit all the connected vertices and Enqueue them for Exploration

Visited: 0, 1, 2, 3, 4, 5, 6

(Not visited) Exploration Queue : 0, 1, 3, 2, 4, 5, 6

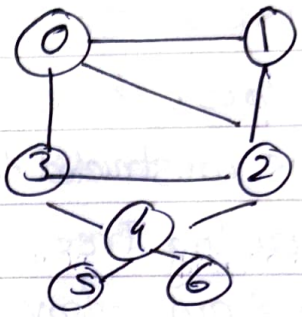
  - Finally Repeat the same for other elements in the Queue

- Algo BFS ⇒
  - Input: A graph G = (V, E) and Source Node is v.
  - Algo:

```
Mark all the nodes in V as unvisited
Mark source nodes as visited
enq(Q,s) // first-in-First out Queue Q
while( Q is not Empty) {
    u := deq(Q);
    For each unvisited neighbour v of u {
        mark v as visited;
        enq(Q,v); } }
```

*2) DFS → (Depth First Search) →

- In DFS, we start with a Node and start Exploring it's connected nodes keeping on suspending the exploration of previous nodes



- DFS Procedure →
- Start by putting any one of the graph's vertices on top of the stack.
- Take the top item of the stack and add it to the visited list.
- Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack
- Keep repeating steps 2 and 3 until the stack is empty.

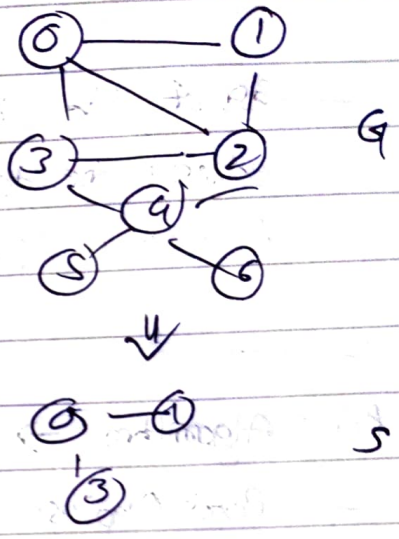
- DFS Algo →

- Input: A Graph G = (V, E) and source code s in V
- Algo:

```
DFS(G,u)
    u.visited = true
    for each v∈G.adj(u)
    IF(v.visited) == false
        DFS(G,v)

init()
    for each u∈G
        u.visited = false
    for each u∈G
        DFS(G,u)
```

Spanning Trees

- Subgraphs =
  - A subgraph of a graph G is a graph whose vertices and edges are subsets of the original graph G



- Connected and Complete Graphs
  - A Connected graph is a graph that is connected in the sense of a topological space i.e. there is a path from any point in the graph. A Graph that is not Connected is said to be disconnected

  - A Complete graph is a simple undirected graph in a which every pair of distinct vertices is connected by a unique edge (Every Node Connected to each other)

- What is Spanning Tree ?.
  - A Connected Subgraph 's' of Graph G(V,E) is said to be a Spanning tree of graph G iff (if and only if):
    1) All vertices of G must be present in S
    2) No of edges in S should be V-1

- No of Spanning trees for Complete Graphs ⇒
  - A Complete graph has $n^{(n-2)}$ Spanning Trees where n is the no. of vertices in the graph

- Spanning Tree Cost ⇒

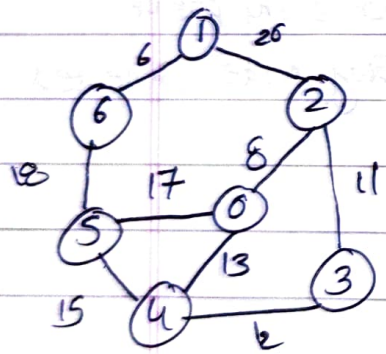
- Cost of the Spanning tree is the sum of the weights of all the edges in the tree
- A minimum Spanning tree is the spanning tree with minimum cost.

—x—

Prims Algorithm ⇒

- Prim's Algo uses Greedy Approach to find the minimum Spanning tree
- We start with any node and start creating a MST.
- In Prim's Algo, We grow the spanning tree from a starting position until n-1 edges are formed (or n nodes are covered)

eg →



Not included in MST

V = {0,1,2,3,4,5,6}

V = {0,1,2,5,4,5}

V = {0,1,2,3,4}

V = {0,2,3}

V = {0,2}

V = {0}

V = { }

Included in MST

A = {1}

options ?

1 → 2 = 20 ✗

1 → 6 = 6 ✓

A = {1,6}

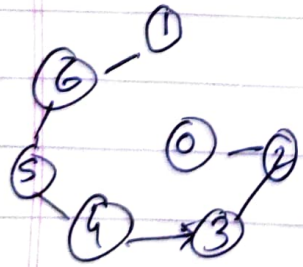
A = {1,5,6}

A = {1,4,5,6}

A = {1,3,4,5,6}

A = {1,2,3,4,5,6}

A = {1,1,2,3,4,5,6,6}



Prims Algo cost = 6 + 18 + 15 + 12 + 11 + 8

= 67