17/1/24

Elementary Data Structures :

→ **Data Structure** : A way of organizing and storing data in a computer's memory or in external storage devices.

→ **Linear Data Structures** : DS that allows data elements to be arranged in a linear/sequencial fashion.

Ex. Array, Stack, Queue, Linked List

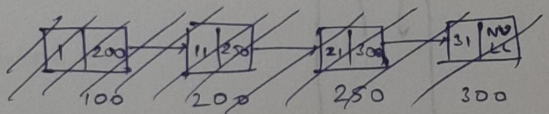
→ **Non-Linear DS** : DS that is consisting of non-linearly arranged data elements

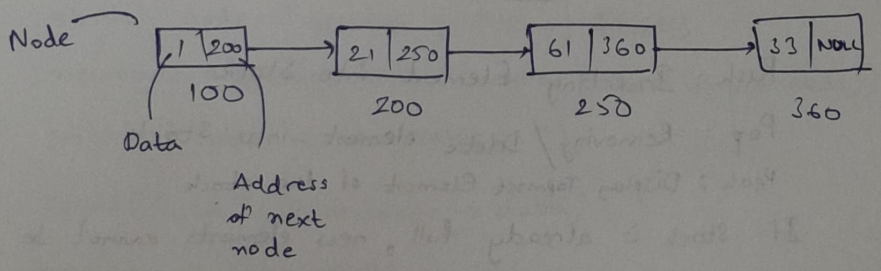
Ex. Tree, Graphs
↳ Loop possible
↳ No Loop posible

- **Linked List** :

Dynamic Memory Allocation (Non-Continuous Memory Allocation)

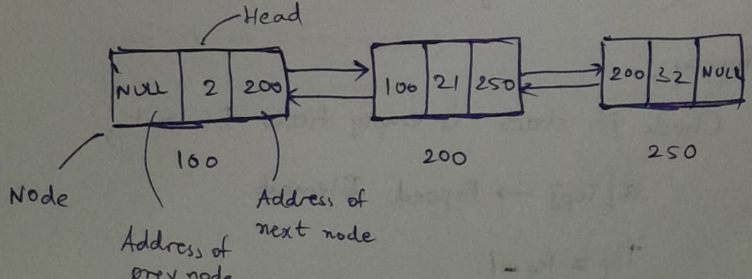
(1) Single Linked List :



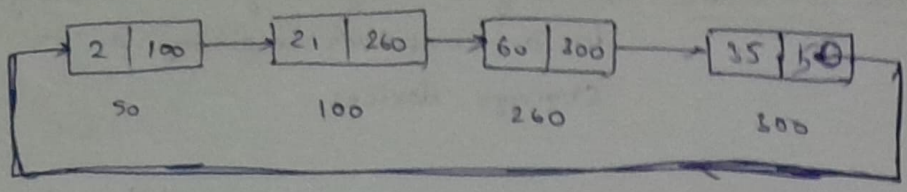


Head : Pointer which holds the address of the first node

(2) Double Linked List :



(1) Circular Linked List:



Circular Singly Linked List

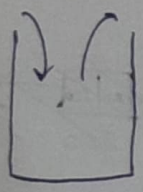
→ Stack: Linear DS

(i) Last In First Out
(ii) First In Last Out

Top: Represents the topmost element in the stack as a pointer.

If Stack is empty, top value == -1

Representation of a Stack:



Push: Inserting Element into Stack
Pop: Removing / Delete element into Stack
Peak: Display Topmost Element of the stack
If Stack is already full, new elements cannot be inserted due to stack overflow.

In that case, Top = Top + 1 (First step)
a[Top] = value (Second step)

(At the stage of Initialisation)

For Popping,

Check if stack is empty first, If not,

a[Top] → Popped Element

Top = Top - 1

- For Recursion, Extra Memory is Required. Which is Stack.

  Non-Recursive Method is better.

- Infix : a + b
  Postfix : * ab +
  Prefix : + ab

} (Operations)

19/1/24

→ Defining a node :

```
struct node
{
   int node;
   struct node *next;
};
struct node *head = NULL;
```

→ Creating a node :

```
Void create()
{
   struct node *newnode, *temp; head
   newnode = (struct node *) malloc (sizeof (struct node));
   printf (" Enter a new value : ");
   scanf ("%d", &newnode -> data);
   newnode -> next = NULL;
   if (head == NULL) {
      head = newnode;
      temp = head; }
   else {
      temp -> next = newnode;
      temp = temp -> next; }
```

→ Displaying elements:

```
void display()
{
    struct node *temp = head;
    if (head == NULL)
        printf("Linked list is empty");
    else {
        while (temp != NULL) {
            printf("%d", temp -> data);
            temp = temp -> next; }
        }
}
```

(1) Create a new node and insert at the beginning of the linked list:

~~Sol: void add()~~
~~{~~
~~struct node *temp;~~
~~if (head == NULL)~~
~~head =~~

Sol: void add()
{
struct node *newnode;
newnode = (struct node *) malloc (sizeof (struct node));
printf("Enter the value: ");
scanf("%d", &newnode -> data);
newnode -> next = NULL;
newnode -> next = head;
head = newnode;

Q) Create a new node and insert at the end of the linked list.

Sol:
```
void add_to_end(){
    struct node *newnode; struct node *temp;
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("Enter the value : ");
    scanf("%d", newnode -> data);
    if (head == NULL) {
        newnode -> next = head;
        head = newnode; }
    else {
    newnode -> next = NULL;
    while (temp -> next != NULL) {
        temp = temp -> next; }
    temp -> next = newnode;
}
```

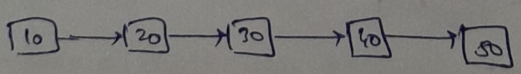
Q) Write a function which returns no. of nodes in the linked list.

Sol:
```
int count_nodes(){
    struct node *temp = head; int count = 0;
    if (head == NULL)
        return count
    else {
        while (temp != NULL) {
            temp = temp -> next;
            count += 1; }
        return count
    }
```

Q) Inserting new node after a particular position

Sol:
```
void insert_node() {
    int pos, i = 1;
    printf("Enter the position: ");
    scanf("%d", &pos);
    int count = count_nodes();
    if (pos > count)
        printf("Invalid Position\n");
    else {
        struct node *newnode, *temp = head;
        newnode = (struct node *) malloc (sizeof (struct node));
        printf("Enter new value: ");
        scanf("%d", &newnode -> data);
        newnode -> next = NULL;
        while (i < pos) {
            temp = temp -> next;
            i += 1; }
        newnode -> next = temp -> next;
        temp -> next = newnode;
    }
}
```

Q) Insert a new node, given a data, find the node number. (M.W)

Ex 

Input : 40
Output: Node 4

24/1/24

Q) ~~Del~~ Deleting a Node @ Beginning

Sol:

```
Void del_at_begin()
{
  if (head == NULL)
    printf("Linked List is Empty");
  else if (head -> next = NULL) {
    printf("The deleted Node is: %d", head -> data);
    free(head);
    head = NULL; }
  else {
    struct node *temp = head;
    head = temp -> next;
    free(temp);
    temp = NULL; }
}
```

Q) ~~Delete~~ Delete at end of a Node:

Sol:

```
void del_at_end()
{
  if (head = NULL)
    printf("Linked List is Empty");
  else if (head -> next = NULL) {
    printf("The deleted node is %d", head -> data);
    free(head);
    head = NULL; }
  else {
    struct node *temp = head, *prev;
    while (temp -> next != NULL) {
      prev = temp;
      temp = temp -> next; }
```

```
        prev -> next = NULL;
        free(temp);
        temp = NULL; }
    }
```

Q) Delete @ Particular Position

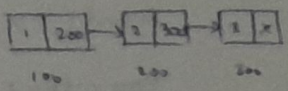
Sol:

```
void del_at_pos() {
    { int pos, i=1;
      printf("Enter Position : ");
      scanf("%d", &pos);
      int count = count_nodes();
      if (pos > count)
          printf("Invalid Input");
      else {
          struct node *prev, *temp = head;
          while (i<pos-1) {
              temp = temp -> next;
              i = i+1; }
          prev = temp -> next;
          temp -> next = prev -> next;
          prev -> next = NULL;
          free(prev);
          prev = NULL;
      }
    }
```

9) Display elements in the reverse direction:

Sol: First reverse Single ~~Input~~ Linked List ⇒ reverse()

```
void display(){
{
    struct node *temp = head;
    if (head == NULL)
        printf("Linked List is empty");
    else{
        while(temp != NULL){
```




```
Void reverse()
{
    struct node *prev, *cur, *nextnode;
    prev = NULL;
    cur = nextnode = head;
    while (nextnode != NULL)
    {
        nextnode = nextnode -> next;
        cur -> next = prev;
        prev = cur;
        cur = nextnode;
    }
    head = prev;
}
reverse();
Struct node *temp = head;
While (temp != NULL) {
    printf("%d", temp -> data);
    temp = temp -> next; }
```

→ Implementation of Stack Using Array:

```
#include<stdio.h>
#include<stdlib.h>
#define size 10
int a[size];
int top = -1;
```

1) void push()

```
{
  int x;
  if (top == size-1)
    printf("Stack is Full\n");
  else {
    printf("Enter the value: ");
    scanf("%d", &x);
    top = top+1;
    a[top] = x; }
}
```

2) void pop()

```
{
  if (top == -1)
    printf("Stack is Empty");
  printf("%d", a[top]);
  top = top-1;
}
```

3) void peek()

```
printf("%d", a[top])
```

```
if (top == -1)
printf("Stack is Empty");
```

4) void display() {

```
if (top == -1)
  printf("Stack is Empty\n");
else {
  for (int i=top; i>=0; i--)
    printf("%d\n", a[i]);
  }
}
```

→ Implementation of Stack Using Linked List:

```
struct node
{
    int data;
    struct node *next;
}*top;
top = NULL;
```

- Creating new node :

```
void push() {
    struct node *newnode;
    newnode = (struct node *) malloc (sizeof (struct node));
    int x;
    printf ("Enter the value : ");
    scanf ("%d", &x);
    newnode -> data = x;
    newnode -> next = top;
    top = newnode;
}
```

- Deleting a node :

```
void pop() {
    if (top == NULL)
        printf ("Stack is Empty");
    else {
        struct node *temp;
        temp = top;
        top = top -> next;
        temp -> next = NULL;
        free (temp);
        temp = NULL;
        }
}
```

25/1/24 (Double Linked List)

→ Defining a Node :

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

struct node *head = NULL;
struct node *tail = NULL;
```

→ Creating a Node :

```
void create()
{
  struct node *newnode;
  newnode = (struct node *) malloc (sizeof(struct node));
  printf("Enter the value");
  scanf("%d", &newnode -> data);
  newnode -> next = NULL;
  newnode -> prev = NULL;
  if (head == NULL) {
      head = tail = newnode;
  }
  else {
      tail -> next = newnode;
      newnode -> prev = tail;
      tail = tail -> next;
  }
}
```

→ Display elements in forward direction :

```
void display()
{
  struct node *temp = head;
  if (head == NULL)
     printf("Linked List is empty");
  else {
     while (temp != NULL) {
        printf("%d", temp -> data);
        temp = temp -> next;
        }
     }
}
```

→ Display elements in reverse direction :

```
void rev-display()
{
  struct node *temp = head;
  if (head == NULL)
     printf("Linked List is empty");
  else {
     while (temp != NULL) {X (temp->next != NULL)
        temp = temp -> next;
        }
     temp = temp -> prev;
     while (temp != NULL) {
        printf("%d", temp -> data);
        temp = temp -> prev;
        }
     }
}
```

→ aft a this, no temp is pointing nowhere

Correct Method:

display elements but *temp = tail instead of *temp = head

→ How to insert a newnode at the beginning:

```
void insert_at_begin(){
    struct node *newnode;
    newnode = (struct node *) malloc(sizeof(struct node));
    printf("Enter the value: ");
    scanf("%d", &newnode->data);
    newnode->next = NULL;
    newnode->prev = NULL;
    if (head == NULL)
        head = tail = newnode;
    else {
        head->prev = newnode;
        newnode->next = head;
        head = newnode;
    }
}
```

→ Inserting a newnode at the end:

```
void insert_at_end(){
    struct node *newnode;
    newnode = (struct node *) malloc(sizeof(struct node));
    printf("Enter the new value: ");
    scanf("%d", &newnode->data);
    newnode->prev = NULL;
    newnode->next = NULL;
    if (head == NULL)
        head = tail = newnode;
    else {
        tail->next = newnode;
        newnode->prev = tail;
        tail = newnode;
    }
}
```

→ Insert a newnode after particular position :

```
void insert_at_pos ()
{
    int pos, i=1;
    printf ("Enter a position : ");
    scanf ("%d", &pos);
    int count = length();
if (pos > count)
    printf ("Invalid position");
else {
    struct node *newnode;
    newnode = (struct node *) malloc (sizeof (struct node));
    printf ("Enter the value : ");
    scanf ("%d", &newnode -> data);
    newnode -> next = NULL;
    newnode -> prev = NULL;
    struct node *temp = head;
    while (i<pos) {
        temp = temp -> next;
        i++; }
    newnode -> prev = temp;
    newnode -> next = temp -> next;
    temp -> next -> prev = newnode;
    temp -> next = newnode;
    }
}
```

→ Delete the Node at Beginning:

```
void del_at_beginning() {
    if (head == NULL)
        printf("Linked List is Empty");
    else if (head -> next = NULL) {
        printf("The deleted Node is %d", head -> data);
        free(head);
        head = NULL;
        tail = NULL;
    }
    else {
        struct node *temp = head;
        head = temp -> next;
        free(temp);
        temp = NULL;
    }
}
```

(OR)

```
else {
    struct node *temp = head;
    head = temp -> next;
    head -> prev = NULL;
    free(temp);
    temp = NULL;
}
```

[temp -> next -> prev = NULL;
temp -> next = NULL;]

→ Delete the Node at the end:

```
void del_at_end() {
    if (head == NULL)
        printf("Linked List is Empty");
    else {
        struct node *temp = tail;
        tail = tail -> prev;
        tail -> next = NULL;
        temp -> prev = NULL;
        free(temp);
        temp = NULL;
    }
}
```

```
else if (head -> next = NULL) {
    free(head);
    head = NULL;
    tail = NULL;
}
```

→ Delete a Node at a Particular Position :

```
void del_at_pos(){
    int pos, i;
    printf("Enter the position : ");
    scanf("%d", &pos);
    int count = length();
    if (pos > count)
        printf("Invalid Choice");
    else {
        struct node *temp = head;
        while (i<pos) {
            temp = temp -> next;
            i = i+1; }

        temp -> prev -> next = temp -> next;
        temp -> next -> prev = temp -> prev;
        temp -> next = NULL;
        temp -> prev = NULL;
        free(temp);
        temp = NULL;
    }
}
```

→ Reverse a double linked list :

```
void rev() {
    struct node *currnode, *nextnode;
    currnode = head;
    while (currnode != NULL) {
        nextnode = currnode -> next;
        currnode -> next = currnode -> prev;
        currnode -> prev = nextnode;
        currnode = nextnode; }
    currnode = head;
    head = tail;
    tail = currnode; }
```

31/1/24

Circular Linked List

- Circular Single Linked List
- Circular Double Linked List

[Singular]

- Defining a Node:

```
struct node {
    int data;
    struct node *next;
};
struct node *head, *tail;
head = tail = NULL;
```

- Creating Node:

```
void create() {
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("Enter the value: ");
    scanf("%d", &newnode->data);
    newnode->next = NULL;
    if (head = NULL) {
        head = tail = newnode;
        tail->next = head; }
    else {
        tail->next = newnode;
        tail = newnode;
        tail->next = head;
    }
}
```

- Insert at Beginning ;

```
void insert_at_begin() {
    struct node *newnode;
    newnode = (struct node *)malloc(sizeof(struct node));
    printf("Enter the value: ");
    scanf("%d", newnode->data);
    newnode->next = NULL;
    if (head == NULL) {
        head = tail = newnode;
        tail -> next = newnode; }
    else {
        newnode -> next = head;
        head = newnode;
        tail-> next = head; }
}
```

H.W

- Insert at Particular Position :

~~Store as linked list~~
~~Write a program~~

- Insert at End:

```
void add_to_end(){
    struct node *newnode, *temp;
    newnode = (struct node*) malloc(sizeof(struct node));
    printf("Enter the value: ");
    scanf("%d", newnode->data);
    if (head == NULL) {
        head = tail = newnode;
        tail->next = newnode; }
    else {
        newnode->next = NULL;
        while (temp->next != head) {
            temp = temp->next }
        temp->next = newnode;
        tail = newnode;
        tail->next = head;
    }
}
```

- Delete at Beginning:

```
Void del_at_begin() {
    if (head == NULL)
        printf("Linked list is Empty");
    else if (head->next == head) {
        free(head);
        head = tail = NULL; }
    else {
        struct node *temp = head;
        head = temp->next;
        tail->next = head;
        free(temp);
        temp = NULL; } }
```

Delete at the end :

```
Void del_at_end {
    if (head == NULL)
        printf("Linked List is empty");
    else if (head->next = head) {
        free(head);
        head = tail = NULL; }
    else {
        struct node *temp = head, *prev;
        while (temp->next != head) {
            prev = temp;
            temp = temp->next; }
        prev->next = head;
        free(temp);
        temp = NULL;
        tail = prev;
    }
}
```

H.W

- Delete at particular position :

2/2/24

- Display elements in Singly Linked List: (Circular)

```
void display() {
    struct node *temp = head;
    if (head == NULL)
        printf("Linked list is Empty");
    else {
        while (temp->next != head) {
            printf("%d", temp->data);
            temp = temp->next; }
    }
    do {
        printf("%d", temp->data);
        temp = temp->next;
    } while (temp != head);
}
}
```

→ Queue:

- Linear Data structure
- First in - First Out principle
- Enqueue: Inserting elements (Done from Rear)
  dequeue: Removing/deleting elements (Done from Front)

- ```
  #include <stdio.h>
  #include <stdlib.h>
  #define size 10
  int queue[size];
  int front = -1;
  int rear = -1;
  ```
  ```

- Inserting:

```
void enqueue() {
    int x;
    printf("Enter value to be inserted: ");
    scanf("%d", &x);
    if (size-1 == rear)
        print("Queue is full");
    else if ((front == -1) && (rear == -1)) {
        front = rare = 0;
        queue[rear] = x; }
    else {
        rear++;
        queue[rear] = x; }
}
```

- Deleting:

```
void dequeue() {
    if ((front == -1) && (rear == -1)) {
        printf("The queue is Empty");
    else if (front == rear) {
        printf("%d", queue[front]);
        front = rear = -1; }
    else {
        printf("%d", queue[front]);
        front = front + 1; }
}
```

- Display elements:

```
void display()
    if ((front == -1) && (rear == -1))
        printf("The Queue is Empty");
    else {
        for (int i=front; i<=rear; i++)
            printf("%d", queue[i]); }
}
```

- Peek function:

```
void peek() {
    if ((front == -1) && (rear == -1)) {
        printf("Queue is Empty");
    else
        printf("%d", queue[front]);
}
```

→ Queue Implementation using a Linked List:

-
```
void enqueue() {
    int x, struct node *newnode;
    newnode = (struct node *) malloc(sizeof(struct node))
    printf("Enter the value : ");
    scanf("%d", &x);
    newnode -> data = x;
    newnode -> next = NULL;
    if (front == NULL) {
        front = newnode;
        rear = newnode;
    }
```

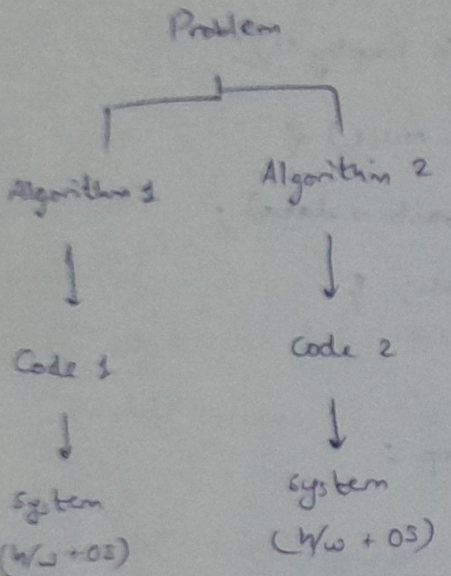
```
else {
    rear -> next = newnode
    rear = newnode
  }
}
```

- ```
  void dequeue() {
    if (front == NULL)
        printf("Queue is Empty");
    else if (front -> next = NULL) {
        printf("%d", front -> data);
        front = rear = NULL; }
    else {
        struct node *temp;
        temp = front;
        front = temp -> next;
        temp -> next = NULL
        free(temp);
        temp = NULL; }
  }
  ```

- ```
  void display() {
    if (front == NULL)
        printf("Queue is Empty");
    else {
        struct node *temp = front;
        while (temp != NULL)
  ```

→ The efficiency of an algorithm depends on the time, storage and resources required to execute the Algorithm.

- Efficiency is measured with the help of assymptotic notations.



(Posterior Analysis)

- Prior Analysis:

  Uses theoretical methods

- Algorithm

  → Should be unambiguous

  → Should be finite, & feasable

  — Should take 0 or 1 input & give atleast one output

  → Two types:

  (i) Iterative Algorithms
  Loops

  (ii) Recursive Algorithm
  Recursive Functions.

Time Complexity:

- Measure of amount of time an algorithm takes to complete as a function of the size of its input
- Computational efficiency of an ~~Program~~ Algorithm.

Space Complexity:

- Total space taken by the Algorithm w.r.t input size the input size. Also includes Auxillary space and space used by the input.
- Auxillary space: Temp space used by the input

→ Asymtotic Notation:

- Big O
- Big Omega
- Big Theta

→ Big O:

worst case Scenario

Ex. $g(n)$ & $f(n)$

$g(n) = O(f(n))$

when $g(n) \leq c f(n)$ $[c > 0]$

Max time an algorithm takes to solve a problem

$\forall n \geq n_0$ $(n_0 > 1)$, There exists a c such that $g(n) > f(n)$

Then $g(n) = O(f(n))$

Ex. $f(n) = 3n + 2$

$g(n) = 4n$

$c = 5$ $(n_0 = 1)$

Ex. $f(n) = 3n+2$

$g(n) = n^2$

$3n+2 = cn^2$

$f(n) \leq g(n) \quad \begin{bmatrix} n=1 \\ c=5 \end{bmatrix}$

→ Big Omega :

Best Case Scenario / Lower-Bound

$f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$

$f(n) = \Omega(g(n))$

Ex. $f(n) = 3n+2$

$g(n) = n$

$3n+2 \geq cn \quad \forall n > n_0$

let $n_0 = 1$

$(3-c)n \geq -2$

$n \leq \frac{2}{c-3}$

$c \leq \frac{3n+2}{n}$

$\Rightarrow c \leq 3 + \frac{2}{n}$

$c = 1$ (min)
$c = 3$ (max)
} for $n_0 = 1$

→ Big Theta:

Represents both lower and upper bound.

$0 \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \quad \forall n \geq n_0$

(Provides tight bound)

$\therefore f(n) = \Theta(g(n))$

Note: Program without loop or Recursive function, Time Complexity is always order of 1. [O(1)]

7/2/24

Ex. A(n) {

```
for (i=0; i<=n; i++)
    printf("Save Me");
}
```

Loop will run for 'n+1' times, for n.

$\therefore O(A(n))$ will be order of 'n'.

Ex. A(n) {

```
for (i=0; i<=n; i++)
    for (j=0; j<=n; j++)
        printf("save me");
}
```

i = 1 2 3 ... n

j(i) 1 4 9 ... n^2

$1+4+9+\dots+n^2$

$= \frac{n(n+1)(2n+1)}{6}$

$= \frac{n^3 + \dots}{6}$

Loop will run for 'n^2' times for n.

$O(A(n))$ will be order of n^2.

Ex. A(n) {

```
i=1; s=1;
while (s<=n) {
    i++;
    s=s+i;
    printf("Save Me");
}
}
```

n=1 → loop runs 1 time
n=2 → loop runs 1 times
n=3 → loop runs 2 times
n=4 → loop runs 2 times
n=5 → loop runs 2 times

i	1	2	3	4	k
s	3	6	10	15	$\frac{(k+1)(k+2)}{2}$

$\frac{(k+1)(k+2)}{2} > n$

$\Rightarrow \frac{k^2}{2} > n$

$k = O(\sqrt{n})$

Ex. A(n) {

```
i=1;
for (i=1; i*i<=n; i++)
    print("Save Me");
}
```

n=1 → L=1
n=2 → L=1
n=3 → L=2
n=9 → L=3

∴ O(A(n)) is of order $\sqrt{n}$.

Ex. A(n) {

```
int i,j,k,n;
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++)
    for (k=1; k<=100; k++)
      printf("Save Me");
}
```

i =	1	2	3	----	n
j(L)	1	2	3	---	n
k(L)	1×100	2×100	3×100	---	n×100

$$100 + 100\times1 + 100\times2 + \cdots + n\times100$$

$$= 100(1+2+\cdots n)$$

$$= 100\ \frac{n(n+1)}{2}$$

$$= 100\left(\frac{n^2+n}{2}\right)$$

order of $\mathcal{O}(A(n)) = \underline{n^2}$

Ex. A(n) {

```
int i,j,k,n;
for (i=1; i<=n; i++)
  for (j=1; j<=i*i; j++)
    for (k=1; k<=n/2; k++)
      printf("Save Me");
}
```

i	1	2	3	4	5	6	7	8	9
j	1	1	1	2	2	2	2	2	3
k	0	1	1	2	2	3	3	4	4

$(1\times0) + (1\times1) + (1\times1) + (2\times2) + (2\times2) + \cdots$

i	1	2	3	.....	n
j	1	4	9	.....	n^2
k	$\frac{n}{2}$	$4\times\frac{n}{2}$	$9\times\frac{n}{2}$	.....	$n^2\times\frac{n}{2}$

$$\frac{n}{2} + 4\times\frac{n}{2} + 9\times\frac{n}{2} + \cdots n^2\times\frac{n}{2}$$

$$\frac{n}{2}\left(\frac{n(n+1)(2n+1)}{6}\right)$$

$$\Rightarrow \boxed{n^4}$$

$O(A(n))$ is of order n^4

(2) A(n) {

```
for (i=1; i<=n; i=i*2)
    printf("Save Me");
```

}

Sol:

n =	1	2	3	4	5	6	7	8
Loop:	1	2	2	3	3	3	3	4

Loop runs k times

$2^{k-1} < n$

$[\log_2 n]$

$\Rightarrow k-1 < \log_2 n$

$\Rightarrow k < \log_2 n + 1$

$O(A(n)) = \log n$

Q) A(n) {

```
int i,j,k;
for (i=n/2; i<=n; i++)
    for (j=1; j<=n/2; j++)
        for (k=1; k<=n; k=k*2)
            printf("Save Me");
```

Sol: i $n/2$

j $n/2$

k $\log_2 n$

$\Rightarrow n/2 \times n/2 \times \log_2 n$

Q) A(n) {

```
int i,j,k;
for (i=n/2; i<=n; i++)
    for (j=1; j<=n; j=j*2)
        for (k=1; k<=n; k=k*2)
            printf("Save Me");
}
```

Sol: i $n/2$

j $\log_2 n$

k $\log_2 n$

$\Rightarrow \frac{n}{2} \times (\log_2 n)^2$

4/2/24

Q) A(n) {

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j=j*j)
        printf("Save Me");
}
```

Sol:

i	1	2	3	4	...	n
j	n	$n/2$	$n/3$	$n/4$	...	n/n

$$n(1 + 1/2 + 1/3 + \ldots 1/n)$$

$$O(n \log n)$$

Q) Assume $n \geq 2$

```
A(n) {
    while (n>1) {
        printf("save me");
        n = n/2;
    }
}
```

Sol:

n = 2	4	8	2^k	5	20
loop: 1	2	2	k	2	4

$$O(\lfloor \log_2 n \rfloor)$$

→ Recursive Algorithm:

Q) A(n) {
  if (n>1) {
    return A(n-1)
  }
}

∴ Recursive Algorithm:

$$T(n) = \begin{cases} 1 + T(n-1) & ; n>1 \\ 1 & ; n=1 \end{cases}$$

Back Substitution Method:

$T(n) = 1 + T(n-1)$ - ①

$T(n-1) = 1 + T(n-2)$ - ②

$T(n-2) = 1 + T(n-3)$ - ③

② in ①:

$T(n) = 1 + 1 + T(n-2)$

③ in ① $1 = 1 + 1 + 1 + T(n-3)$

$\vdots$

$= k + T(n-k)$

$\vdots$ $[n-k=1]$

$= n-1+T(n-(n-1))$

$= n-1+T(1)$

$= n-1+1$

$\therefore T(n) = n$

Q) $T(n) = \begin{cases} n + T(n-1) & : n>1 \\ 1 & : n=1 \end{cases}$

$T(n) = n + T(n-1)$ - ①

$T(n-1) = n + T(n-2) - 1$ - ②

$T(n-2) = n + T(n-3) - 2$ - ③

$T(n) = n + n + T(n-2) - 1$

$= 3n + T(n-3) - 3$

$\vdots$

$\Rightarrow T(n) = kn + T(n-k) - \frac{k(k-1)}{2}$

$$T(n) = kn + T(n-k) - \frac{k(k-1)}{2}$$

$$= kn + T(n-(n-1)) - \frac{(k)(k-1)}{2}$$

$$= (n-1)(n) + T(1) - \frac{(n-1)(n-2)}{2}$$

$$\therefore T(n) = n^2 - n + 1 - \frac{(n-1)(n-2)}{2}$$

$$= n^2 + 1 - 2n$$

$$\therefore T(n) = n^2 - 2n + 2$$

$$T(n) = n^2 - n + 1 - \frac{n^2 - 3n + 2}{2}$$

$$= \frac{n^2}{2} + \frac{n}{2}$$

$$\therefore T(n) = \frac{1}{2}(n^2 + n)$$

13/2/24

→ Sorting Techniques:

Sorting efficiently arranges elements in a specific order.

Various sorting Algorithms:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort
- Heap Sort
- Counting Sort
- Radix Sort

(1) Bubble Sort:

Ex. 5, 9, 2, 15, 6, 11

CE NE

Pass 1 : 5 9 2 15 6 11

5 2 9 15 6 11

5 2 9 15 6 11

5 2 9 6 15 11

5 2 9 6 11 15

if Current Element (CE) is less than Next Element (NE), Continue sorting.

Else Sort

Pass 2 : 2 5 9 6 11 [15]

2 5 9 6 11 15

2 5 6 9 11 15

2 5 6 9 11 15

2 5 6 9 11 15 ✓

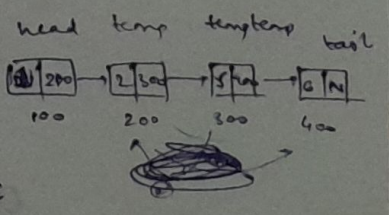
Pass 3 : Runs - No need to Swap 2 5 6 9 [11 15]

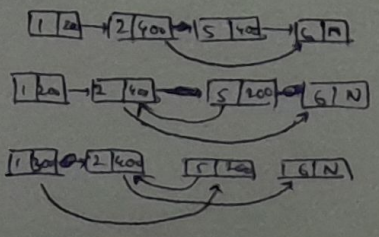
Pass 4 : Runs - NO need to Swap 2 5 6 [9 11 15]

Pass 5 : Runs - No need to Swap 2 5 [6 9 11 15]

There will be 'n-i' passes (Algorithm) for 'n' elements.

```
swap (pos) {
    struct node *temp = head;
    int i = 1;
    while (i < pos) {
        temp = temp -> next;
        i++;
    }
    struct node *temptemp = temp -> next;
    temp -> next = temptemp -> next;
    temptemp -> next = temp;
    temp -> prev -> next = temptemp;
}
```





```
void bubble_sort(int arr[], int n) {
    int i, j;
    for (i=0; i<n-1; i++) {
        for (j=0; j<n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Time Complexity:

Pass 1 : n-1

Pass 2 : n-2

Pass 3 : n-3

Pass 4 : n-4

⋮

1

$1+2+3+\dots+(n-1)$

$$= \frac{(n-1)(n-2)}{2}$$

$$O(n) = n^2$$

ii) Selection Sort :

Ex. 5, 9, 2, 15, 6, 11

2], 9, 5, 15, 6, 11

2, 5], 9, 15, 6, 11

2, 5, 6], 9, 15, 11

2, 5, 6, 9], 15, 11

2, 5, 6, 9, 11], 15

[Finding smallest Element]

↳ By seeing minimum element of an array

1/2/24

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

(iii) Insertion Sort:

- Unsorted Subarray &
  Sorted Subarray
- Works effectively for small input size

Ex. 5, 1, 8, 2, 16

5], 1, 8, 2, 16

Sorted

1, 5], 8, 2, 16

Sorted Unsorted

1, 5, 8 2, 16

Sorted Unsorted

1, 2, 5, 8 16

Sorted Unsorted

1, 2, 5, 8, 16

Sorted

Best case Time Complexity is of the order 'n'. i.e. All elements in increasing order already.

Worst Case : n^2

- void insertionsort (int a[], n) {

```
int i, j, temp;
for (i=1; i<n; i++) {
    temp = a[i];
    j = i-1;
    while (j>=0 && a[j] > temp) {
        a[j+1] = a[j];
        j--; }
    a[j+1] = temp; } }
```

16/2/24

Quick sort:

(H.W: write C program)

Ex. 56, 28, 95, 18, 78, 33, 45, 57, 20

Pivot element - First element (56)

i = 0
j = 9

→ if (a[i] <= Pivot) { → Looped ('while' instead of 'if')
    i++ }

→ if (a[j] > Pivot) { ↗
    j-- }

→ if (i < j)
    swap (a[i], a[j])
  else
    swap (a[j], Pivot) } → One pass is completed once else block is executed

↓↑

Pivot element will be in its place

i.e. Left of Pivot < Pivot
Right of Pivot > Pivot

After one pass of Quick sort,

33, 28, 20, 18, 45, 56, 78, 57, 95

Quick Sort 1 | Quick Sort 2

∴ Divide and Conquer Sorting Mechanism

For 1: 33, 28, 20, 18, 45

i = 4,
j = 3 } ⇒ ~~33, 28, 20, 18, 33~~
⇒ 18, 28, 20, 33, 45

For 2: 78, 57, 95

i = 2
j = 1 } ⇒ 78, 95, 57

→ Merge Sort:

Ex. 40, 30, 45, 5, 11, 84, 2

left → 0 (l)

right → 6 (r)

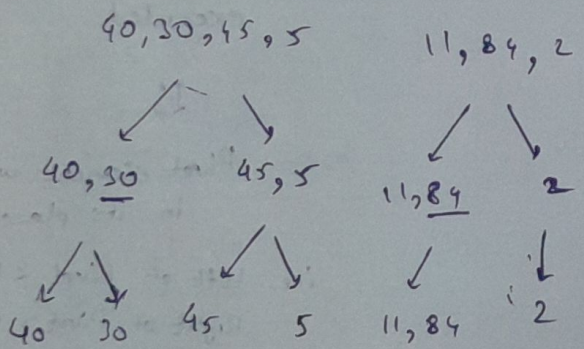
m (middle element) = $\frac{l+r}{2}$

$\therefore m = 3$

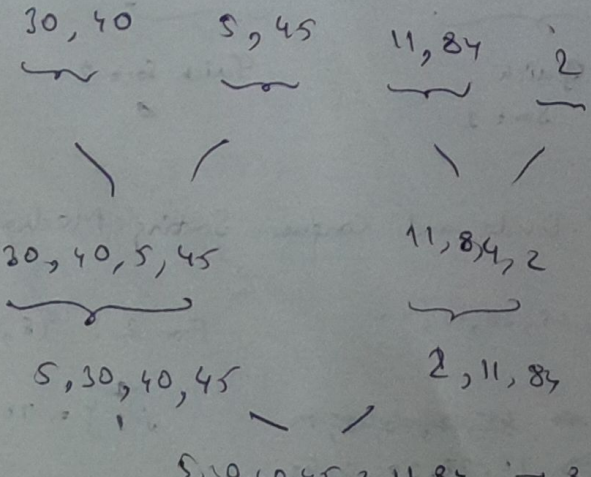
40, 30, 45, 5 | 11, 84, 2

merge sort (a, l, m)

$(0+3)/2 = 1.5 = 1$

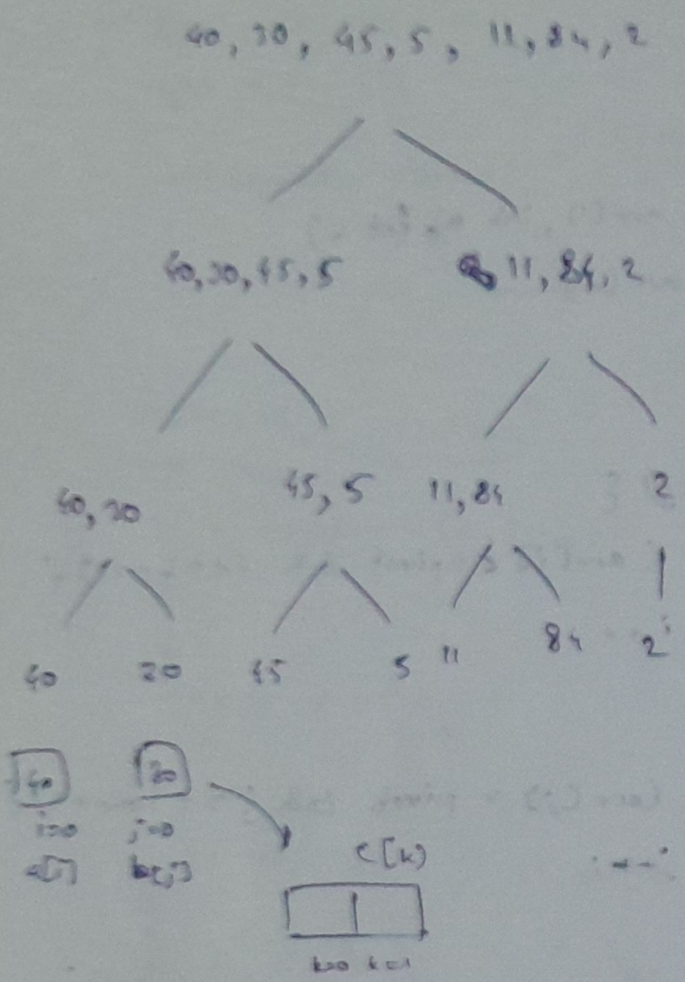


Sort & Combine



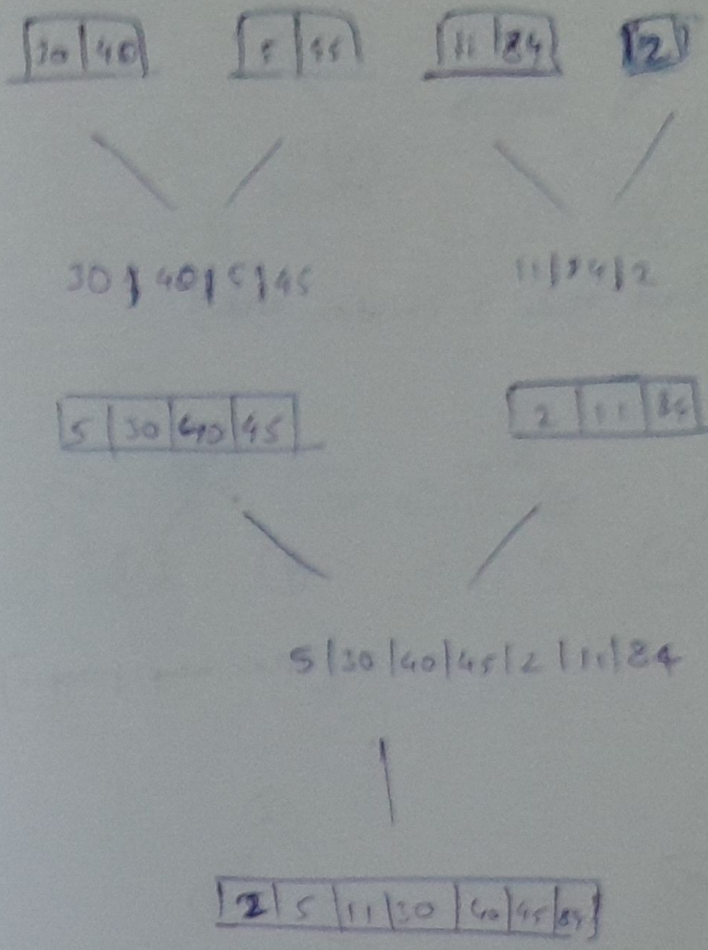
19/8/24

→ Merge Sort:




```
if (a[i]<b[j]) {
    c[k] = a[i];
    i++;
    k++; }

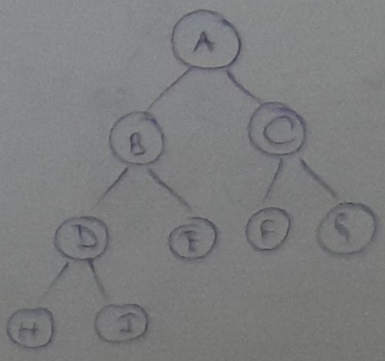
else {
    c[k] = b[j];
    j++;
    k++;
}
```



→ Heap Sort :

Arraying elements in increasing order : Max heap

Arranging elements in decreasing order : Min heap



Complete Binary Tree

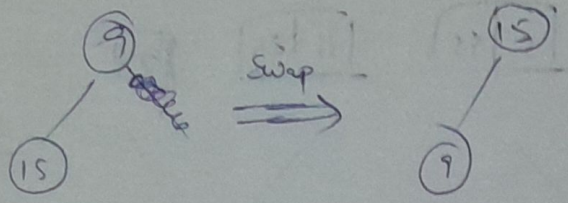
(i) All Nodes should have 2 children

(ii) Left to Right Filling of Leaf Node

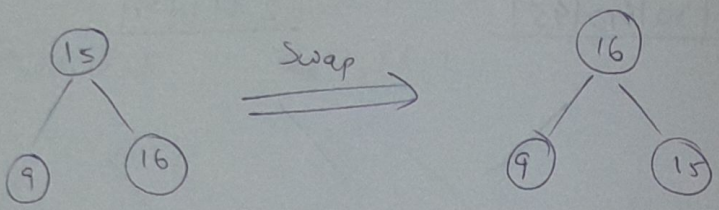
Ex. 9, 15, 16, 1, 15, 10

↓

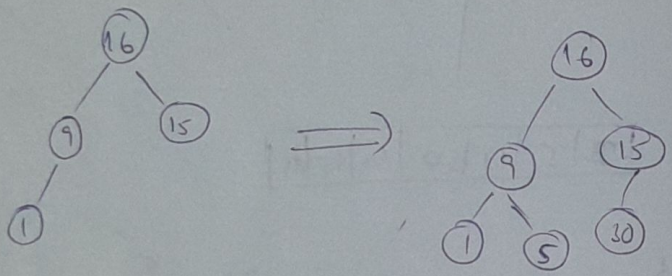
Root Node



[Parent Node > Children Node]

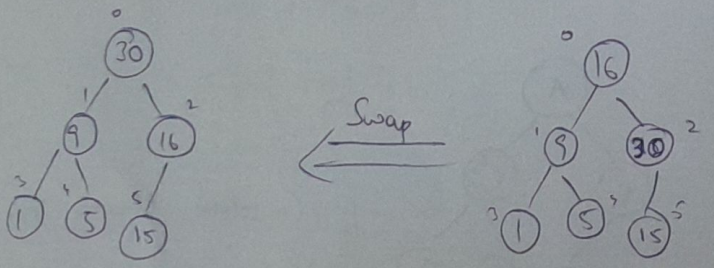


[Max heap process]



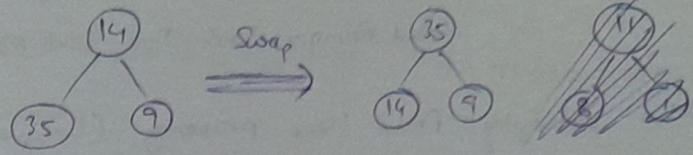
[Complete Binary Tree Rule]

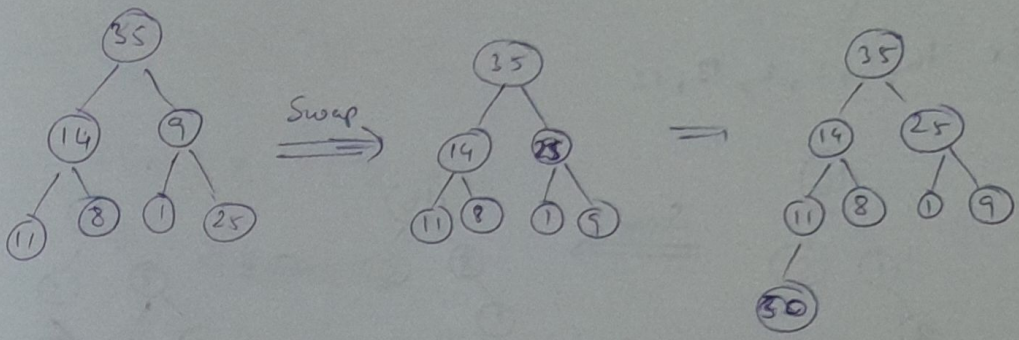
Swap

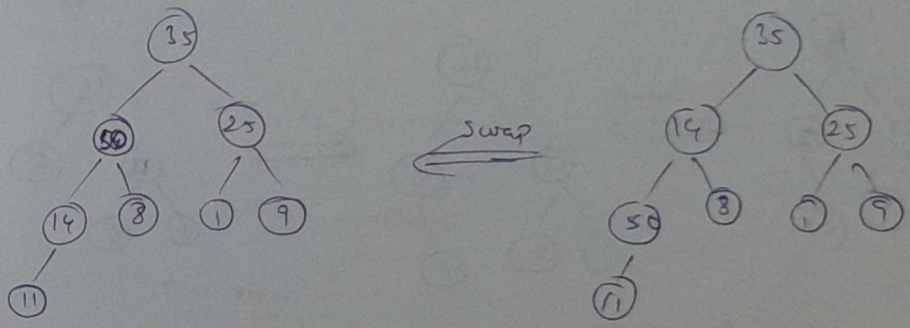


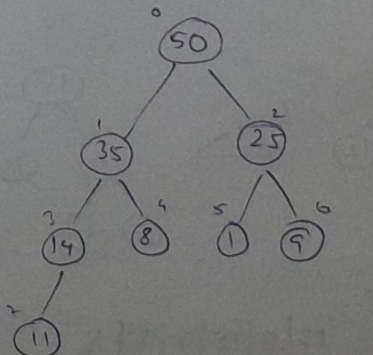
30|9|16|1|5|15

Ex. 14, 35, 9, 11, 8, 1, 25, 50









50 | 35 | 25 | 14 | 8 | 1 | 9 | 11

Swapping process - Heapify Process

Heap Sort : First & Last elements are swapped

15 | 9 | 16 | 1 | 5 | 30 Then Heapify Again

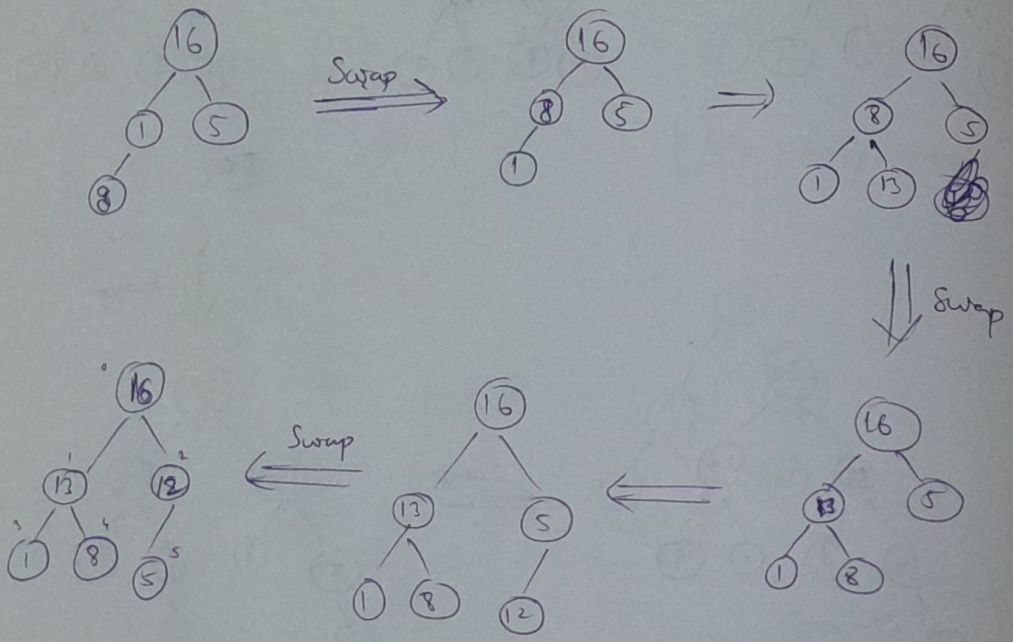
While Heapifying, Remove leaf Nodes one by one.

& Swap first and Last Nodes one by one.

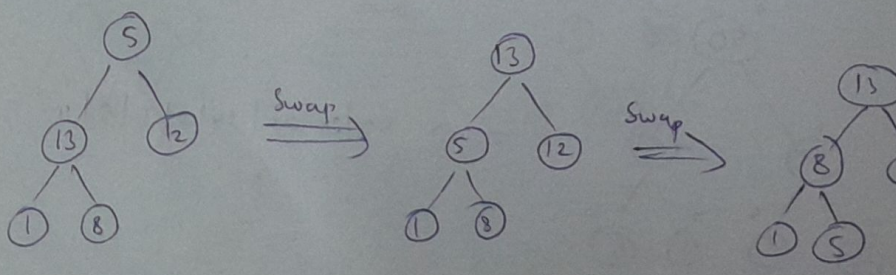
[And Remove Last The Leaf Node]

Also satisfying Max Heap property (Complete Binary Tree)

Ex. 16, 1, 5, 8, 13, 12



16 | 13 | 12 | 1 | 8 | 5



13 | 8 | 12 | 1 | 5 | 16

→ For Quicksort:

$T(1) = 1$

$T(2) = 2 \times T(n/2) + n$ $\qquad T(n) = 2T(n/2) + n$

~~$T(3) =$~~

$T(n/2) = 2 \times T(n/4) + n/2$

$T(n/4) = 2 \times T(n/8) + n/4$

$\therefore T(n) = 4 \times T(n/4) + 2n$

~~$2^3 T(n/8)$~~

$= 4 \times (2 \times T(n/8) + n/4) + 2n$

$= 8 \times T(n/8) + 3n$

$\vdots$

$= 2^k * T(n/2^k) + nk$

$= 2^k \times T(1) + n\log n$

$= n + n\log n$

$\frac{n}{2^k} \geq 1$

$\Rightarrow n \geq 2^k$

$\Rightarrow \log_2 n \geq k$

$\therefore \Omega(n \log n)$

[Best Case]

~~Θ fn~~

$O(n^2)$ → if we need ascending but given descending
i.e. worst case

→ Radix Sort:

- No comparisons
- Distribute elements into buckets based on individual digits.
- Processes digits from LSD to MSD (or) Vice-versa

Ex. 170, 45, 75, 90, 802, 24, 2, 66.

∴ 170, 045, 075, 090, 802, 024, 002, 066

0	170, 090
1	
2	802, 002
3	
4	024
5	045, 075
6	066
7	
8	
9	

170, 090, 802, 002, 024, 045, 075, 066

0	802, 002
1	
2	024
3	
4	045
5	
6	066
7	170, 075
8	
9	090

802, 002, 024, 045, 066, 170, 075, 090

0	002, 024, 045, 066, 075, 090
1	170
2	
3	
4	
5	
6	
7	
8	802
9	

002, 024, 045, 066, 075, 090, 170, 802

∴ 2, 24, 45, 66, 78, 90, 170, 802

Sorted

→ Counting Sort :

- Non-comparitive :

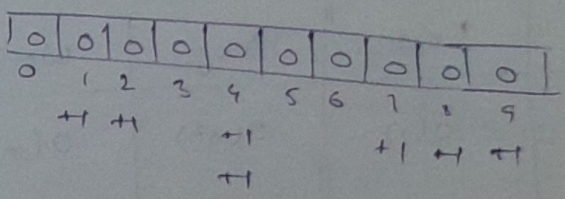
  Counts the n occurances of each element in the input.

  Works well if Δ of range b/w elements is not that large

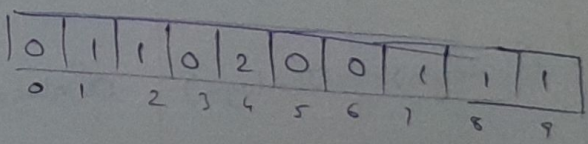
Ex. 2, 9, 7, 4, 1, 8, 4

↳ largest element

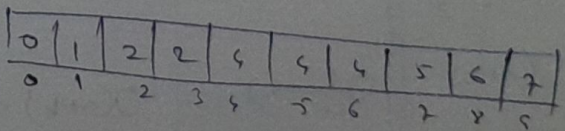
Create Array of (9+1) indexes



Digit by Digit, Adding 1 to each data of created array wherever the index corresponded to the given data.

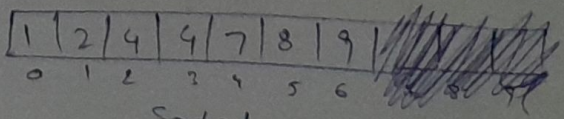


Adding Cumulatively



Start from last : 2, 9, 7, 4, 1, 8, (4)

Check value at 4th Index, Subtract 1 and place at 3rd Index



8 → 6-1 = (5)
1 → 1-1 = (0)
4 → 3-1 = (2)
7 → 5-1 = (4)

Sorted

→ Inplace Sorting Algorithms:

- Doesn't take extra space
  i.e. Sorts within its own memory

Ex. Bubble, Insertion, Selection, Heap, Quick
?

→ Stable Sorting Algorithms:

- Relative order of equal elements in the sorted output follow their relative order in the unsorted input

Ex. Bubble, Insertion, Selection, Counting, Merge, Radix

→ Time Complexity of Algorithms:

	Best	Avg	Worst	
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	
Merge & Heap	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	
Quick	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	
Counting	$O(n+k)$	$O(n+k)$	$O(n+k)$	k: range of input of non-negative key-values
Radix	$O(n-k)$	$O(n-k)$	$O(n-k)$	k: n(digits) in the largest element.

→ Master's Theorem:

$$T(n) = aT(n/b) + f(n) \qquad \left[\text{where } f(n) = \Theta(n^k \log^p n)\right]$$

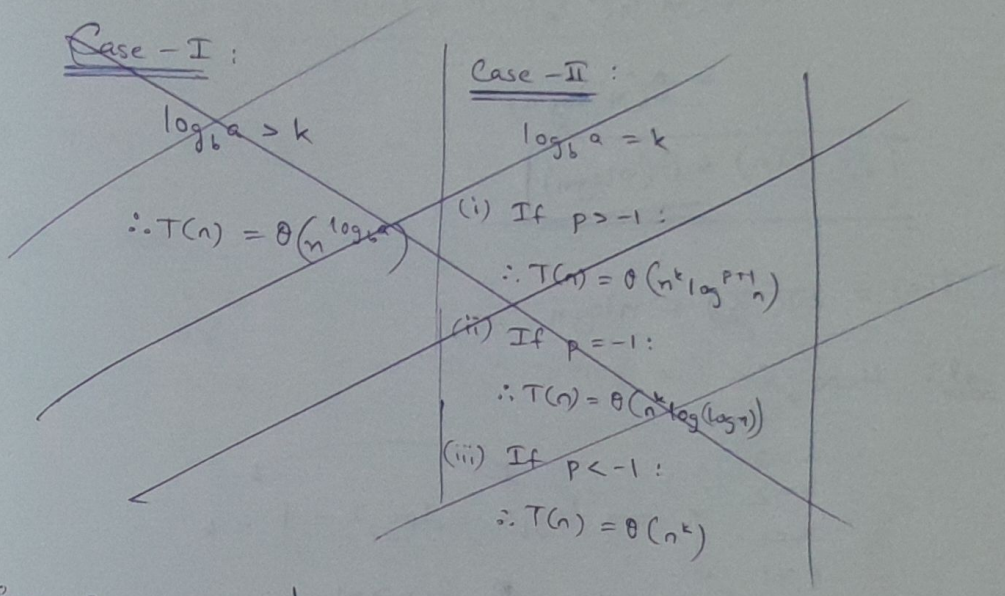
Conditions:

$a >= 1$,

$b > 1$,

$k >= 0$,

$p \in R$



Case - I : $\log_b a > k$

$\therefore T(n) = \Theta(n^{\log_b a})$

Case - II : $\log_b a = k$

(i) If $p > -1$:

$\therefore T(n) = \Theta(n^k \log^{p+1} n)$

(ii) If $p = -1$:

$\therefore T(n) = \Theta(n^k \log(\log n))$

(iii) If $p < -1$:

$\therefore T(n) = \Theta(n^k)$

Case - III : $\log_b a < k$

(i) If $p >= 0$:

$\therefore T(n) = \Theta(n^k \log^p n)$

(ii) If $p < 0$:

$\therefore T(n) = \Theta(n^k)$

Ex. $T(n) = 2T(n/2) + n$

Sol: Here,

$a = 2$
$b = 2$
$k = 1$
$p = 0$

$\Rightarrow \log_b a = \log_2 2 = 1 = k$

& $p > -1$

~~$\therefore \theta(n) = \theta(n)$~~

$\therefore \theta(n^k \log^{p+1} n)$

$= \theta(n^1 \log n)$

$\therefore T(n) = \theta(n \log n)$

Q) $T(n) = 2T(n/2) + n \log n$

Sol: Here,

$a = 2$
$b = 2$
$k = 1$
$p = 1$

$\Rightarrow \log_b a = \log_2 2 = 1 = k$

& $p > -1$

$\theta(n^k \log^{p+1} n)$

$\therefore T(n) = \theta(n \log^2 n)$

Q) $T(n) = 2T(n/2) + n/\log n$

Sol: Here,

$a = 2$
$b = 2$
$k = 1$
$p = -1$

$\Rightarrow \log_b a = \log_2 2 = 1 = k$

& $p = -1$

$\theta(n^k \log(\log n))$

$\therefore T(n) \Rightarrow \theta(n \log(\log n))$

Q) $T(n) = 3T(n/2) + n^2$

Sol: Here,

$a = 3$
$b = 2$
$k = 2$
$p = 0$

$\Rightarrow \log_b a = \log_2 3 < k$

& $p \geq 0$

$\therefore \Theta(n^k \log^p n)$

$= \Theta(n^2 \log^0 n)$

$\therefore T(n) = \Theta(n^2)$

Q) $T(n) = 2T(n/2) + n^{0.51}$

Sol: Here,

$a = 2$
$b = 2$
$k = 0.51$
$p = 0$

$\Rightarrow \log_b a = \log_2 2 = 1 > k$

$\therefore \Theta(n^{\log_b a})$

$\Rightarrow \Theta(n^{\log_2 2})$

$\therefore T(n) = \Theta(n)$

Q) $T(n) = (0.5)T(n/2) + 1/n$

Sol: Here,

$a = 0.5$
$b = 2$
$k = -1$
$p = 0$

$\Rightarrow$ No Master's Theorem

$\because$ Conditions broken are

$k \geq 0$ & $a \geq 1$

Q) $T(n) = 3T(n/2) + (\log n)^2$

Here: $a = 3$, $b = 2$, $k = 0$, $p = 2$ $\Rightarrow \log_b a = \log_2 3 > k$

$\theta(n^{\log_b a})$

$\Rightarrow \theta(n^{\log_2 3})$

$\therefore T(n) = \theta(n^{\log_2 3})$

Q) $T(n) = 2T(n/2) + n(\log n)^2$

Sol: Here, $a = 2$, $b = 2$, $k = 1$, $p = 2$ $\Rightarrow \log_b a = k$ & $p > -1$

$\therefore \theta(n^k \log^{p+1} n)$

$\Rightarrow \theta(n^1 \log^3 n)$

$\therefore T(n) = \theta(n \log^3 n)$

Q) $T(n) = 16T(n/4) + n$

Sol: Here, $a = 16$, $b = 4$, $k = 1$, $p = 0$ $\Rightarrow \log_b a > k$

$\theta(n^{\log_b a})$

$\Rightarrow T(n) = \theta(n^2)$

$T(n) = 2^n T(n/2) + n^n$

↳ Here, Master's Theorem cannot be applied

∴ Apply Back Substitution.

→ Little OH :

- If $f(n) <= cg(n)$,

  Then $f(n) = O(g(n))$

- $f(n) < cg(n)$; $c > 0$

  $f(n) = n$ &

  $g(n) = n^2$

  If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$; Then $f(n) = O(g(n))$ ↘ Little OH

  $\lim_{n \to \infty} \frac{n}{n^2} = \lim_{n \to \infty} \frac{1}{n} = 0$ | ∴ $f(n) = O(g(n))$

Find c S.T $f(n) < c \cdot g(n)$ Always

Ex $f(n) = \log n$

$g(n) = n$

$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{\log n}{n} = \lim_{n \to \infty} \frac{1/n}{1} = 0$

$\left[\frac{\infty}{\infty}\right]$

$f(n) = O(g(n))$

$\log(n) = O(n)$

→ Little OMEGA :

↳ lower bound, Not tight

↳ If $f(n) = \omega(g(n))$, then $|f(n)| > c|g(n)|$

↳ $f(n)$ grows ~~most~~ much faster than $g(n)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) \to \omega(g(n))$$

Ex $f(n) = n$; $g(n) = \log n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{n}{\log n} = \frac{1}{1/n} = n = \infty$$

$$\therefore f(n) = \omega(g(n)) \Rightarrow n = \omega(\log n)$$

Ex $f(n) = 3^n$, $g(n) = 2^n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{2}\right)^n = \infty$$

$$\therefore f(n) = \omega(g(n))$$

$$3^n = \omega(2^n)$$

→ Space Complexity:

Measure of Amount of Storage space an Algorithm / program uses to solve a problem as a function of size of the input.

Expresied in 'big OH' notation

Ex. printf("%d", a+b); [OR] printf S = a+b; printf("%d", s);

$O(2) \equiv O(1)$

~~SP~~

$SP = C + S_p$

C: Independent

S_p: Dependent

```
int n, a[n];
sum = 0;
for (int i=0; i<n; i++) {
    sum += a[i]; }
```

$SP = c + s_p$

$= 3 + n$ → (n Integers)

$= O(n+3)$

$= O(n)$

Another Definition : Extra memory of storage used is SP.

For Sorting :

- Bubble : O(1)
- Selection : O(1)
- Insertion : O(1)
- Merge : O(n)
- Quick : O(nlogn)
- Heap : O(1)
- Radix : O(n+k)
- Counting : O(k) or O(n+k)

Searching Algorithms :

- **Linear Search : (Sequential Search)**

```
int linear_search (int a[], int target, int n) {
    for (int i=0; i<n; i++) {
        if (a[i] == target)
            return i; // found, returns index
    }
    return -1; // Not found
}
```

Best Case : O(1)
Avg. Case : O(n)
Worst Case : O(n)
} TIME COMPLEXITIES

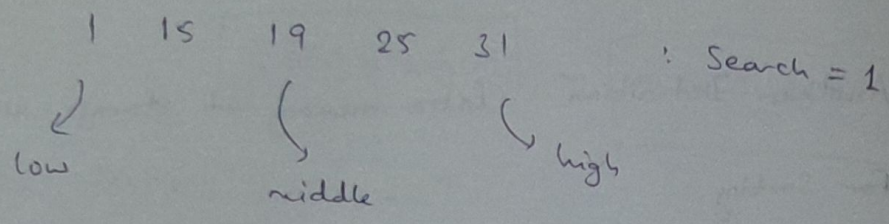
Space Complexity : O(1)

- Binary Search:

↳ For Sorted Arrays

↳ Divides Search in ~~array~~ half:

Narrows down possible locations for the target value



low = 0

high = 4

middle = (0+4)/2 = 4/2 = 2

$$\left[\text{middle} = \frac{\text{low} + \text{high}}{2}\right]$$

Compare a[m] == Target

(19 ≠ 1) ⇒ false

Since false, we narrow search to LHS of m as target < middle

~~middle~~

low = 0,

high = middle - 1

= 2 - 1

= 1

middle = $\frac{0+1}{2}$ = 1/2 = 0

Compare a[middle] == Target

⇒ True

So found, This is binary search

if a[middle] < target,

low = ~~m~~ mid + 1 and high does not change

→ int binary_search (int a[], int low, int high, int target) {

```
while (low <= high) {
    int mid = (low+high)/2;
    if (a[mid] == target)
        return mid;
    else if (a[mid] < target)
        low = mid + 1;
    else
        high = mid - 1;
}
return -1;
}
```

Time Complexity:	Space Complexity:
Best: $O(1)$	$O(1)$
Avg: $O(\log n)$	
Worst: $O(\log n)$	

→ for Binary Search,

$$T(n) = T(n/2) + 1$$

$$\Downarrow$$

$$T(n) = aT(n/b) + \theta(n^k \log^p n)$$

Comparing, we get

$a = 1$
$b = 2$
$k = 0$
$p = 0$

$\log_b a = \log_2 1 = 0 = k$ & $p > -1$

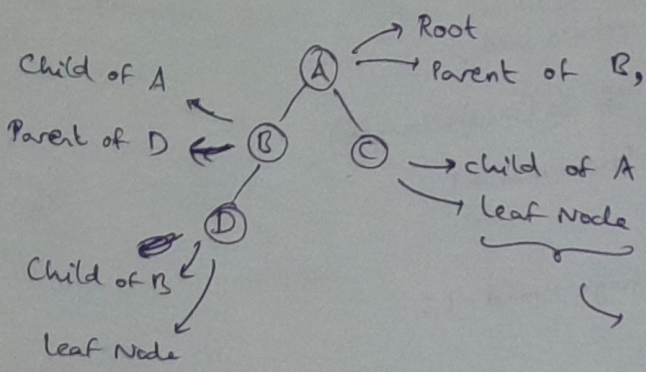
$$\therefore T(n) = \theta(n^k \log^{p+1} n)$$
$$= \theta(n^0 \cdot \log^{0+1} n)$$
$$= \theta(\log n)$$

$$\therefore T(n) = \theta(\log n)$$

14/3/24

TREES

→ It is a Hirarchial data structure of nodes connected by edges.



Internal Nodes are Nodes other than leaf Nodes.

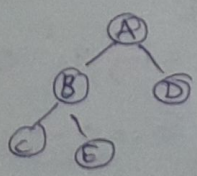
→ Binary Tree :

↳ Each Node has atmost 2 children

Strict (or) Full Binary Tree :

↳ Should be a binary Tree with 0 or 2 children only

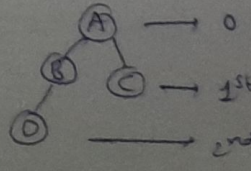
Ex.

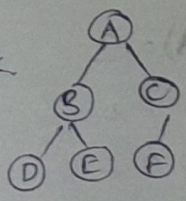


Complete Binary Tree :

↳ All levels of a Binary Tree are completely filled except for possibly last node

Nodes on the last level are filled from left to right.



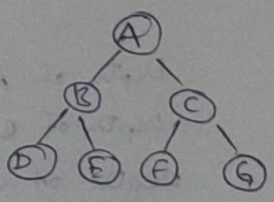
Ex 

✗ ⇒ It is not a complete Binary Tree.

→ Perfect Binary Tree :

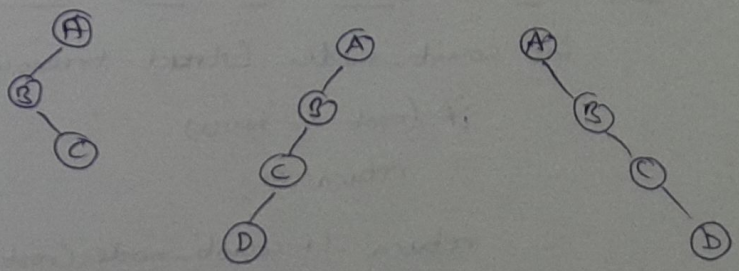
↳ All Internal Nodes should have 2 children

A full Binary Tree where all leaf nodes are at the same level

Ex. 

→ Degenerate Binary Tree :

↳ A binary tree where each parent node has only one associated child node.

Ex 

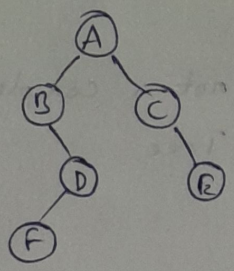
Left Skewed Right Skewed

→ Balanced Binary Tree :

Height of a Node : The no. of edges in the longest path from the node to any leaf node in the tree (binary)

Height of a Tree (binary) : The no. of edges in the longest path from the root node to any leaf node in the tree (binary)

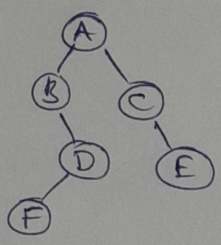
Ex.



Height of Root (A) = 3
Height of Node B = 2
Height of Node D = 1
Height of Node F = 0
Height of Node C = 1
Height of Node E = 0

→ Depth of a Node : No. of edges in the path formed from root Node to given node.

Ex.



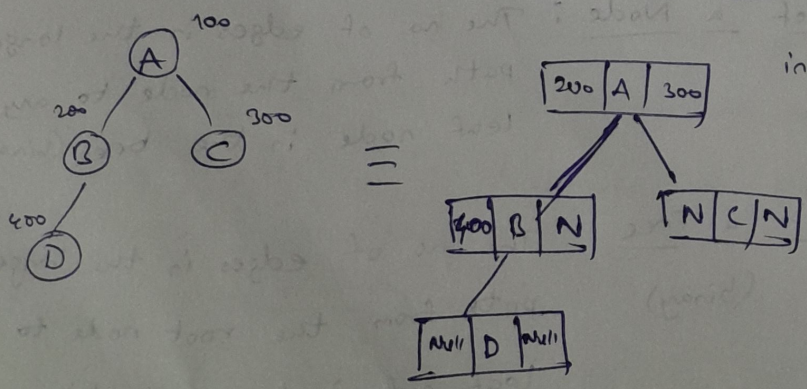
Depth of A = 0
Depth of B = 1
Depth of C = 1
Depth of D = 2
Depth of E = 2
Depth of F = 3

Depth of a Tree = Height of the tree

- Count No. of Nodes in binary Tree :

```
int count_nodes (struct treenode *root) {
    if (root == NULL)
        return 0;
    return 1+count_nodes(root->left) + count_nodes
                                        (root->right);
}
```

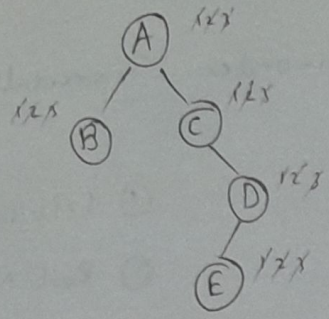
- To Implement Binary Tree, we use Double linked list :




```
int height (node *p){
    int h;
    if (p==NULL)
        return -1;
    int left = height(p->left);
    int right = height(p->right);
    if (left>right)
        h = left+1;
    else
        h = right+1;
    return h;
}
```

→ Post order Traversal:

① Left child
② Right child
③ Root Node



BEDCA

→ In-order Traversal Code:

```
void inOrderTraversal (struct TreeNode *root) {
    if (root == NULL)
        return;
    inOrderTraversal (root -> left);
    printf ("%d", root -> data);
    inOrderTraversal (root -> right);
}
```

→ Pre-Order Traversal Code:

```
Void preOrderTraversal (struct TreeNode *root) {
    if (root == NULL)
        return;
    printf("%d", root -> data);
    preOrderTraversal (root -> left);
    preOrderTraversal (root -> right);
}
```

→ Post-Order Traversal :

```
void postOrderTraversal (struct TreeNode *root) {
    if (root == NULL)
        return;
    postOrderTraversal (root -> left);
    postOrderTraversal (root -> right);
    printf ("%d", root -> data);
}
```

* → Expression Tree :

Binary Tree representing Arithmetic expressions.

Three types of notations :

(i) In-fix Notation :
a + b [operator comes between operand]

(ii) Pre-fix Notation :
+ ab [operator comes before operand]

(iii) Post-fix Notation : [operator comes after operand]
ab +

Ex. a + b x c

Prescedence : x > +

Associativity : Left to Right

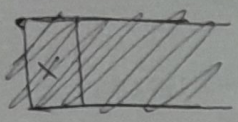
① b x c
② a + (b x c)

Prefix Notation : ① x bc
② + a x bc [+ a b'] (b' = x bc)

Postfix Notation : ① bc x
② bc x a + [b' c' +] (b' = x bc)

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

Sol: Stack: (1) Postfix Expression (2)



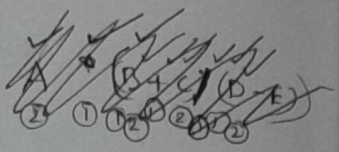
A

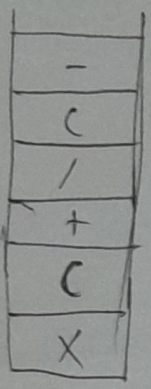
AB

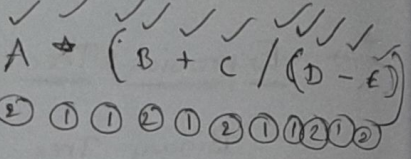
ABC

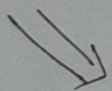
ABCD

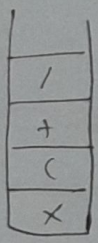
ABCDE



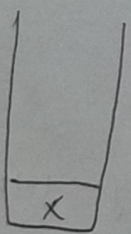








ABCDE-



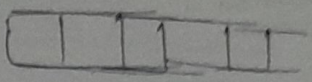
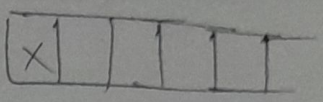
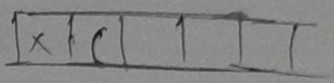
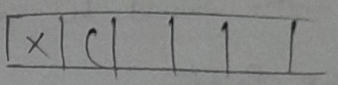
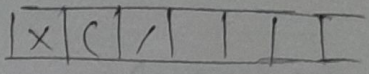
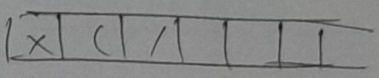
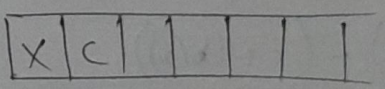
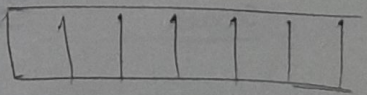
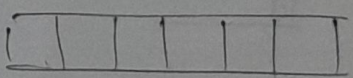
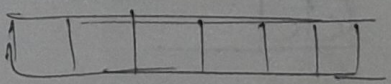
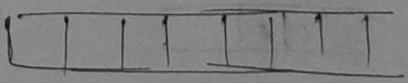
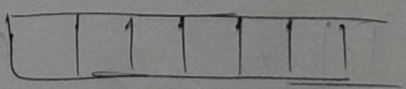
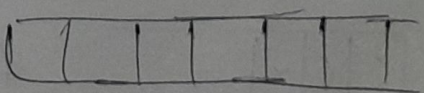
ABCDE-/+

ABCDE-/+X

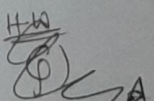
Postfix

(9) A x (B/C + (D - E))

sol.

Scanned	stack	Postfix expression
A		A
x		A
(		A
B		AB
/		AB
C		ABC
+		ABC
(		ABC
D		
-		
E		
)		
)		

HW



(g) A + ((B * C - (D / E ^ F) * G)

18/3/24

→ Infix to Prefix conversion :

Step 1 : Reverse the Infix expression.

Note : (→)

) → (

Step 2 : Reversed Infix should be converted to Postfix

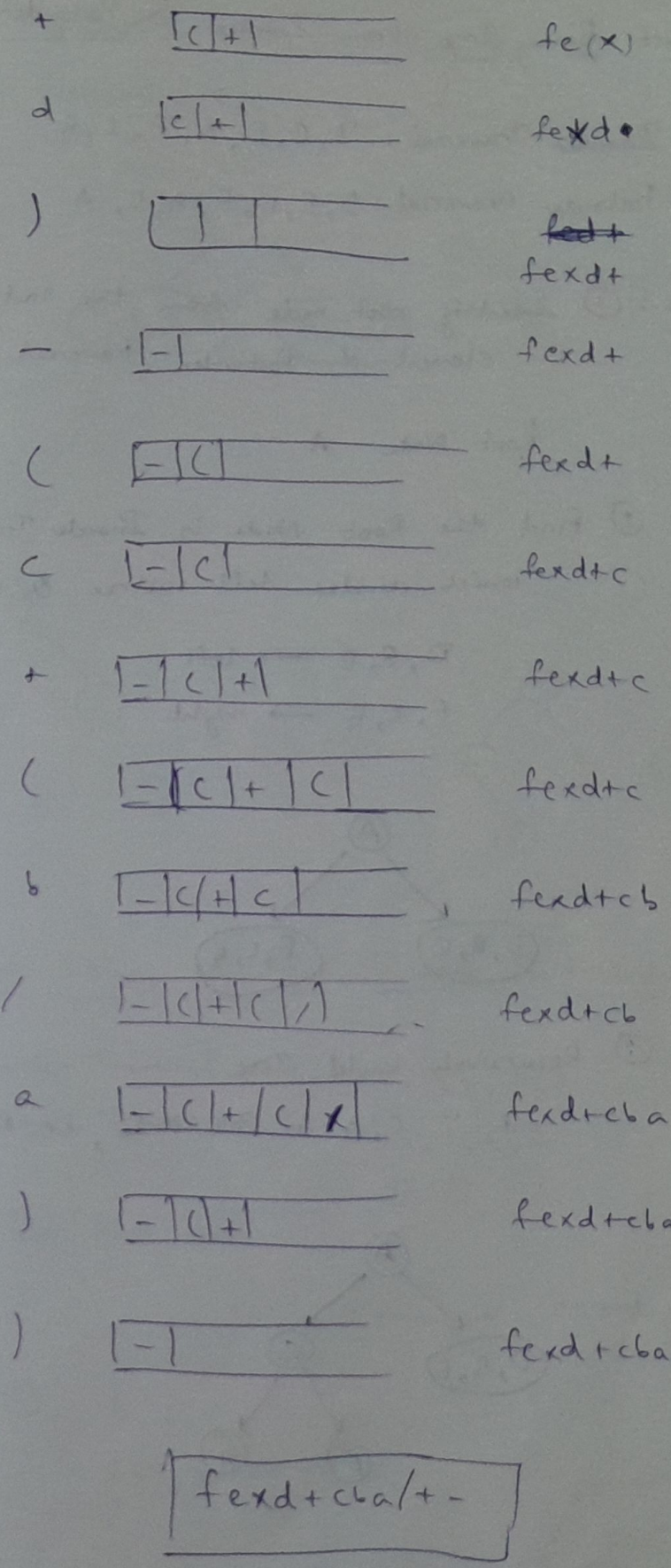
Step 3 : Revere the Postfix

Ex. ((a/b) + c) - (d + (e * f))

Reversing :

① ((f ^ e) + d) - (c + (b/a))

Scanned Symbol	Stack	Postfix
(	[( ]	
(	[(( ]	
f	[(( ]	f
x	[((x]	f
e	[((x]	fe
)	[( ]	fe


+
(|+|
fe(x)
d
(|+|
fexd*
)
(| | |
~~fed+~~
fexd+
-
|-|
fexd+
(
|-|(|
fexd+
c
|-|c|
fexd+c
+
|-|c|+|
fexd+c
(
|-|c|+|c|
fexd+c
b
|-|c|+|c|
fexd+cb
/
|-|c|+|c|/|
fexd+cb
a
|-|c|+|c|x|
fexd+cba
)
|-|c|+|
fexd+cba/
)
|-|
fexd+cba/+
fexd+cba/+-
Postfix : -+/abc+d*ef

→ Construct Binary Tree from Inorder & Postorder :

Ex. Inorder Traversal : D, B, E, A, F, C, G

Postorder Traversal : D, E, B, F, G, C, A

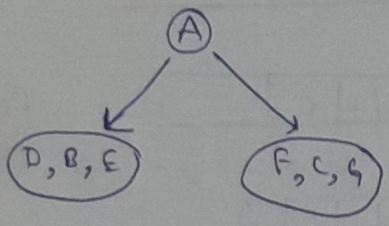
Steps : ① Identify root node from the last ~~node~~ element of Postorder Traversal.

Root Node : A

② Find the Root Node in Inorder Traversal which divides left subtree & right subtree

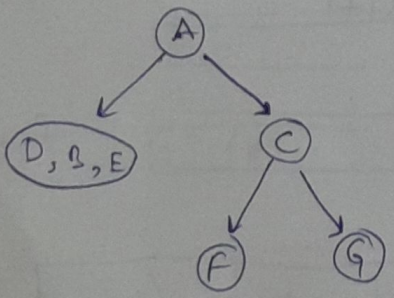
D, B, E → left

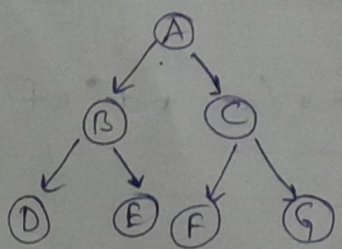
F, C, G → right



③ Recursively build Tree

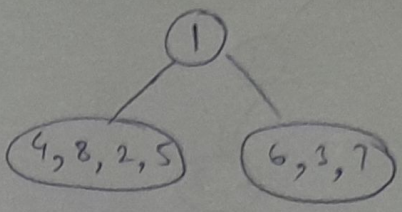
∵ C, A in Post order, i.e. Right to left scanning.

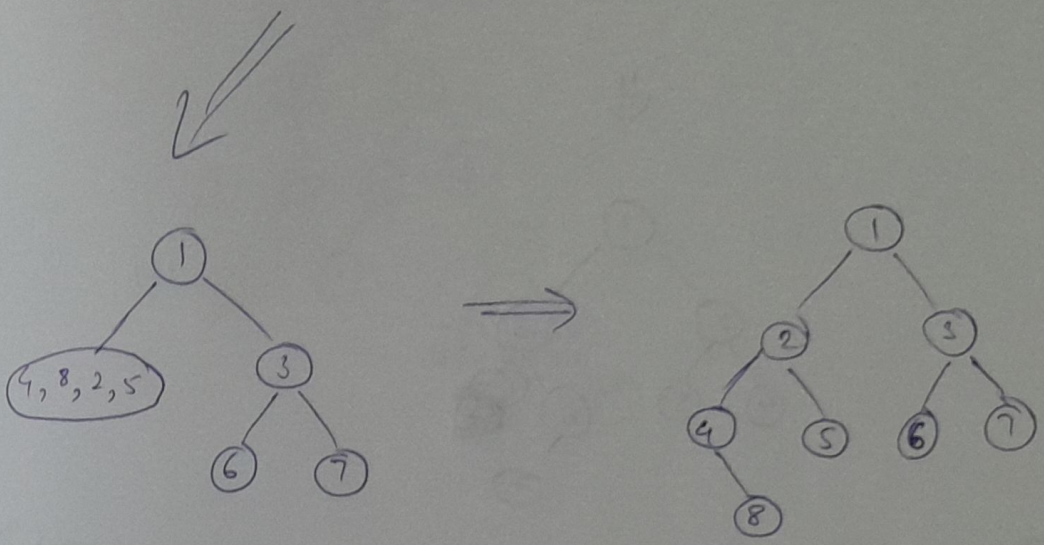




g) Inorder Traversal : 4, 8, 2, 5, 1, 6, 3, 7

Postorder Traversal : 8, 4, 5, 2, 6, 7, 3, 1





→ Construct Binary Tree from Inorder & Preorder:

Ex. Inorder Traversal : 4, 2, 5, 1, 6, 7, 3

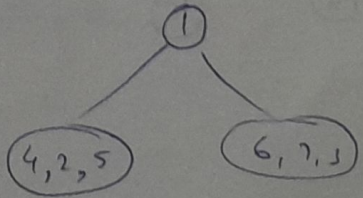
Preorder Traversal : 1, 2, 4, 5, 3, 6, 7

Steps ① Root Node : ①

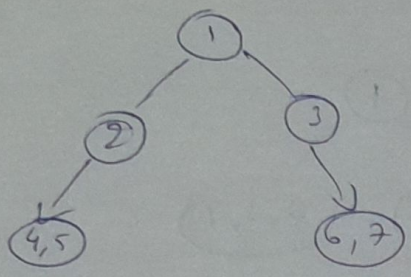
[First element of Pre-order Traversal]

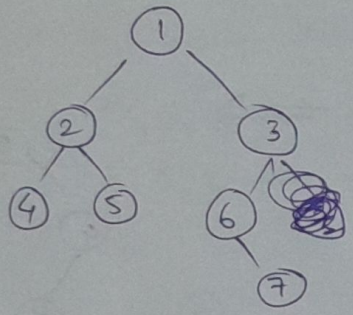
② 4, 2, 5 → left

6, 7, 3 — right



③ Left to right scanning

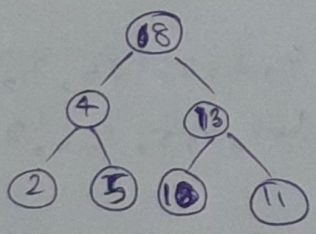




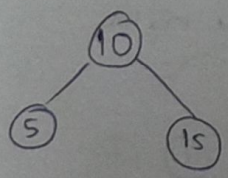
19/3/24

→ Binary Search Tree :

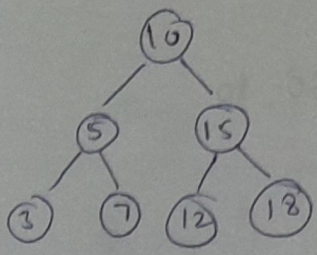
Ex

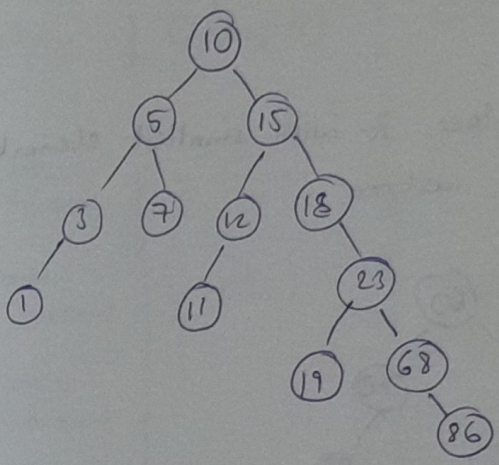


Ex. 10, 5, 15, 3, 7, 12, 18, 1, 23, 11, 19, 68, 86



Check:
10 > 5
10 < 15



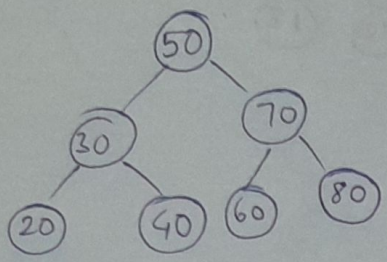


→ Pseudo-code :

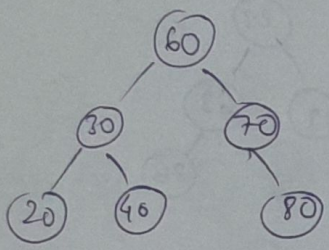
```
struct node {
    int data;
    [illegible];
    struct node *left;
    struct node *right;
};

struct node *createNode (int value) {
    struct node *newnode = (struct node *) malloc (sizeof (struct node));
    newnode -> data = value;
    newnode -> left = NULL;
    newnode -> right = NULL;
    return newnode;
}
```

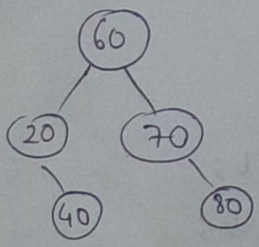
Ex. 50, 30, 20, 40, 70, 60, 80



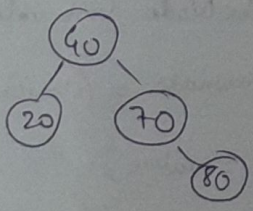
Delete 50 and replace it with smallest element from right subtree



Del 30 with left subtree (largest)



Del 60 with left subtree (largest)



→ Time Complexity in Binary Search Tree:

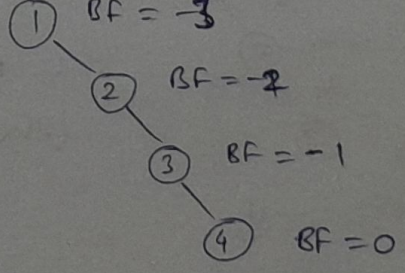
~~Search~~	Avg.	Worst
Search	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(n)$
Inorder Traversal	$O(n)$	$O(n)$
Preorder Traversal		
Postorder Traversal		
Finding min/ max element	$O(\log n)$	$O(n)$

↳ Un-balanced Binary Tree

↔ **Balanced Binary Tree**

Balance Factor (BF) = Height of the ~~right~~ left subtree − Height of the right subtree

Ex.

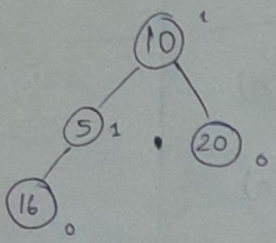


Balanced Binary Tree : BF for each Node lies in $[-1, 1]$

↳ Ex. ~~Actual~~ AVL Trees

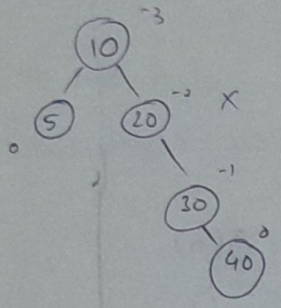
20/3/24

→ AVL Trees:



BF ∈ [-1, 1]

∴ AVL Tree



BF ∉ [-1, 1]

∴ Not a AVL Tree

AVL : Self Balancing Binary Search Tree

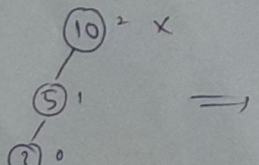
If not AVL,

1. LL Rotation
2. RR Rotation
3. LR Rotation
4. RL Rotation

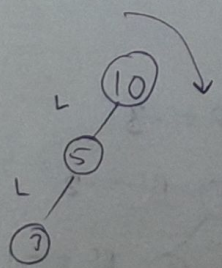
[L : Left
R : Right]

① Left - Left Rotation :

Ex.

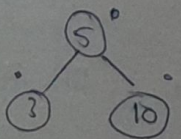


⟹



∴ LL Rotation
[Clockwise]

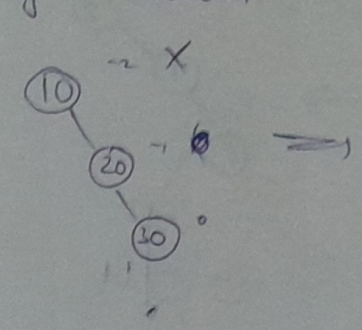
⇓



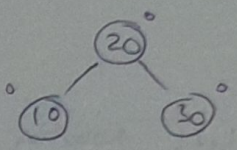
∴ AVL Tree

(2) Right - Right Rotation

Ex.



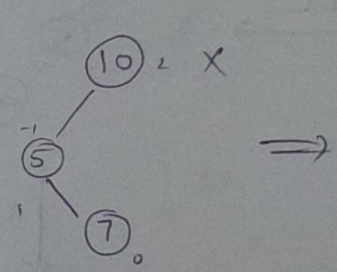
∴ RR Rotation
(Anti-Clockwise)



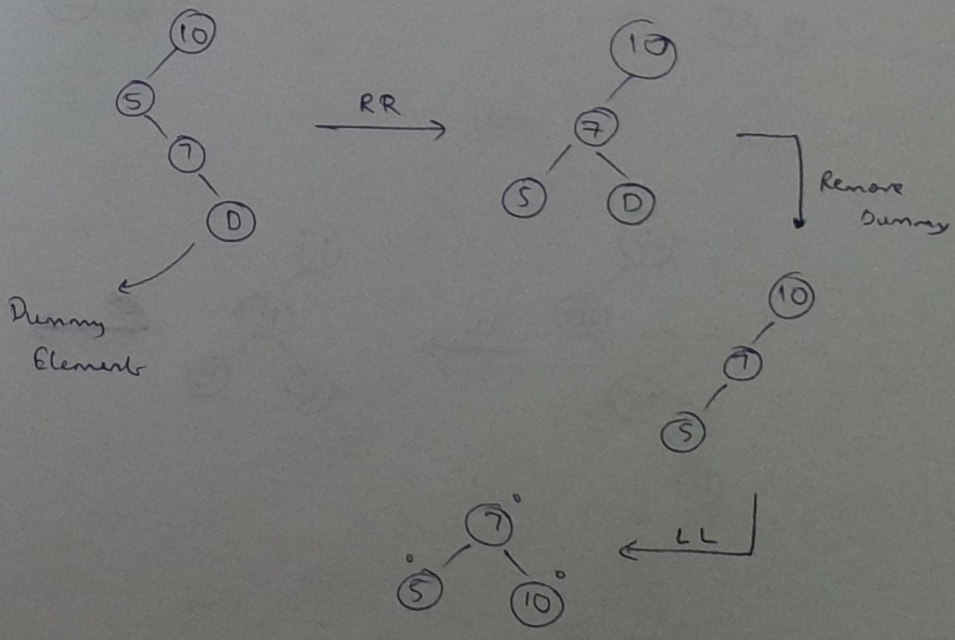
∴ AVL Tree

(3) Left - Right Rotation

Ex.

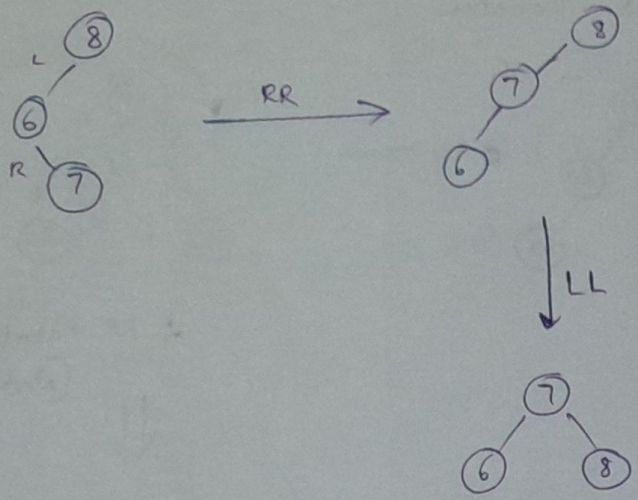


(1) RR Rotation
(2) LL Rotation

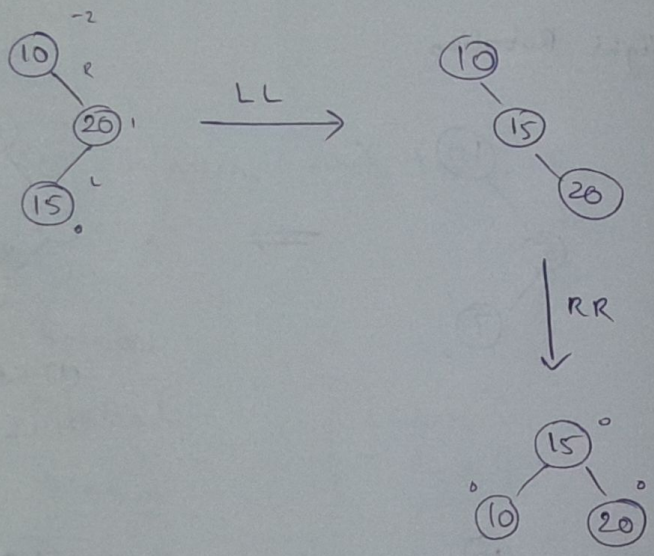


∴ AVL Tree

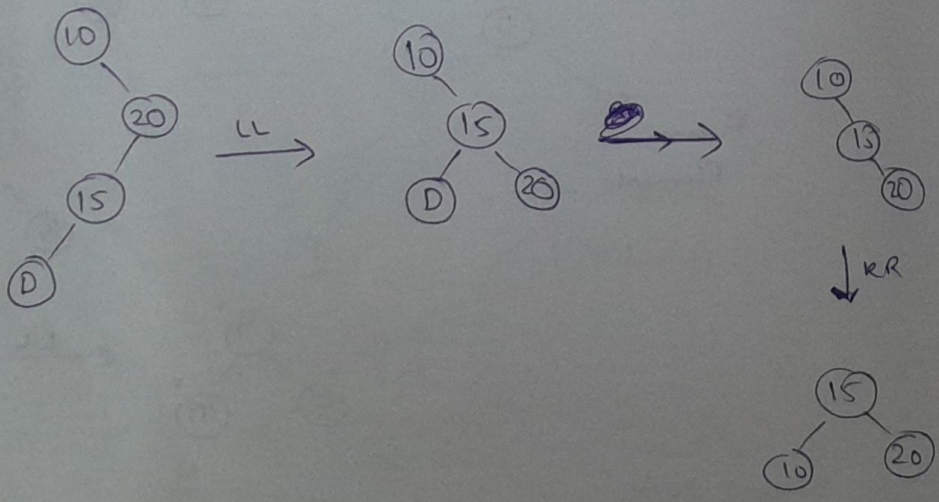
g)



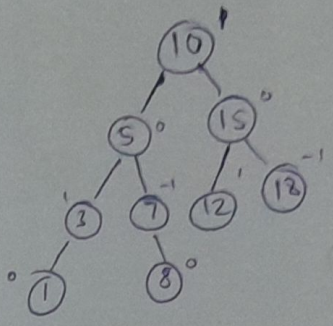
④ Right - left Rotation :



(OR)

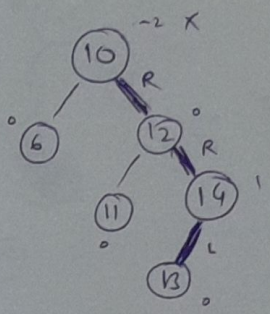


Ex. 10, 5, 15, 3, 7, 12, 8, 18, 1



AVL ✓

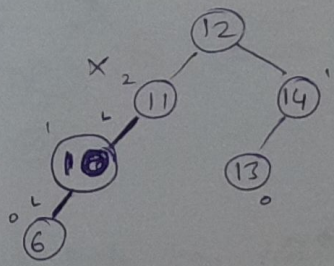
Ex. 10, 6, 12, 11, 14, 13

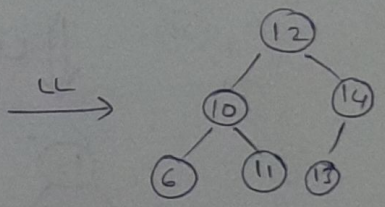


10 ⟶ 13 : RRL

(only first 2)

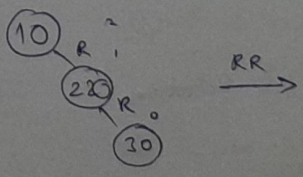
RR



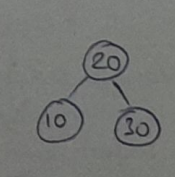


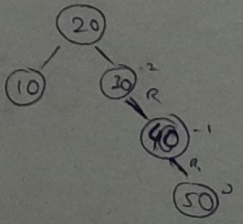
AVL ✓

Ex. 10, 20, 30, 40, 50

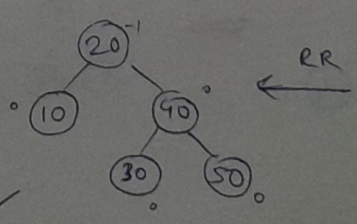


RR





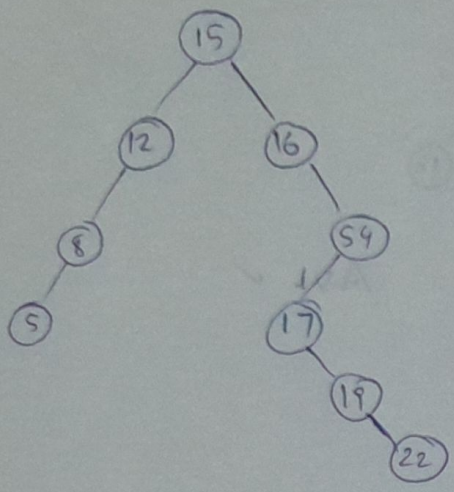
RR



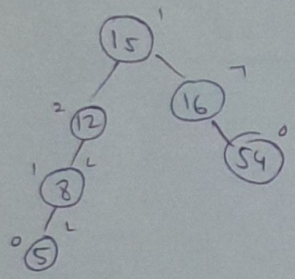
AVL ✓

22/2/24

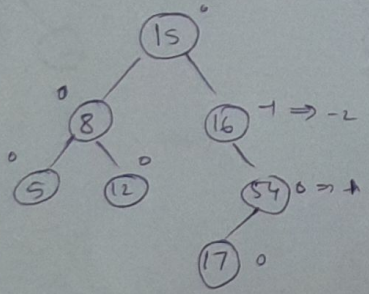
Ex. 15, 16, 12, 8, 54, 5, 17, 19, 22



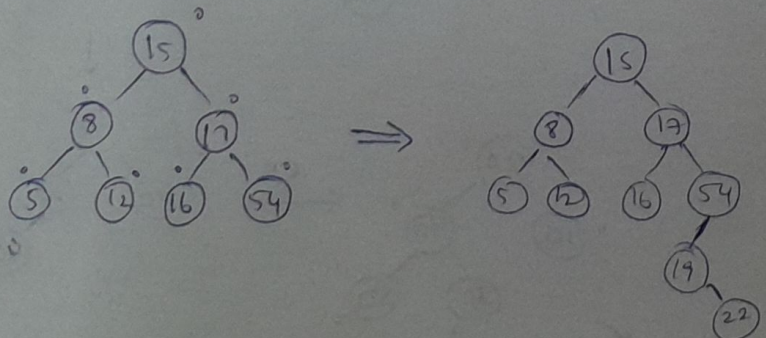
But for AVL Tree:

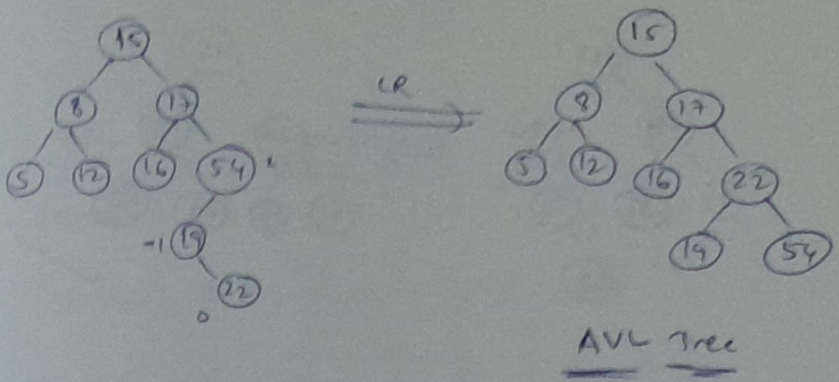


LL



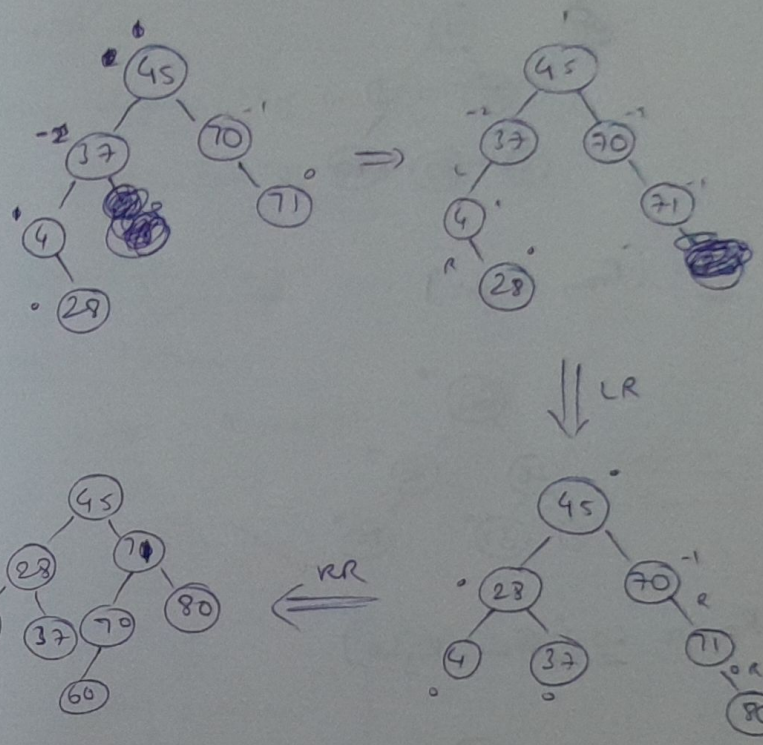
RL Rotation



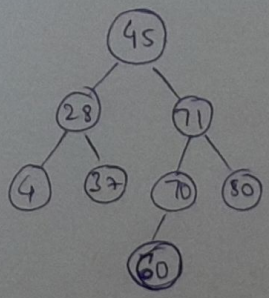


AVL Tree

Ex. 45, 70, 37, 4, 71, 28, 80, 60



→ Deletion of Nodes from AVL Tree :

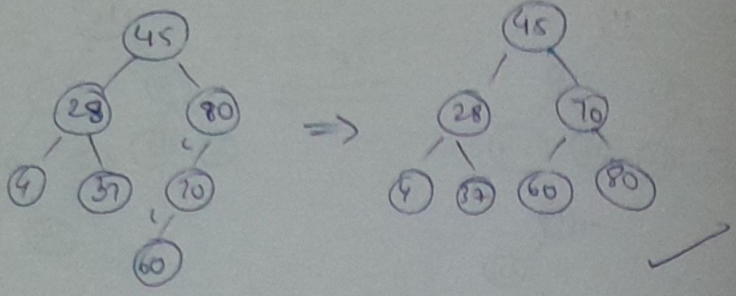


① Find (i) Largest element of Left Subtree
(ii) Smallest element (OR) of Right Subtree

② Delete 71 and replace it with above

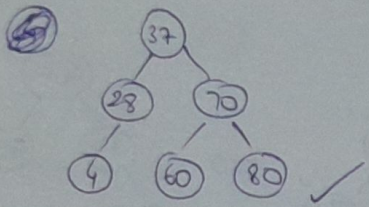
③ Apply Appropriate Rotation

Deleting 71 :

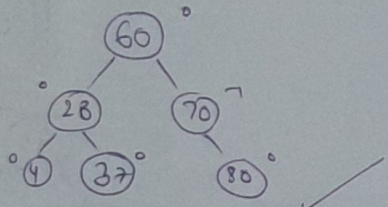


~~Delete~~

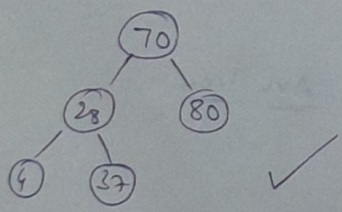
Delete 45 : [from left]



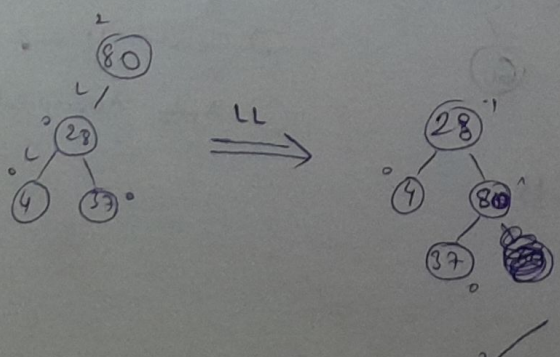
Delete 45 : [from Right]



Remove 60 : [from Right]



Delete 70 : [from Right]



27/3/24

AVL Tree Time complexity:

Search : O(log n)

Insertion : O(log n)

Deletion : O(log n)

→ ## Hashing :

Hash functions :

Takes Input (Key or Identifier),

Produces a hash code

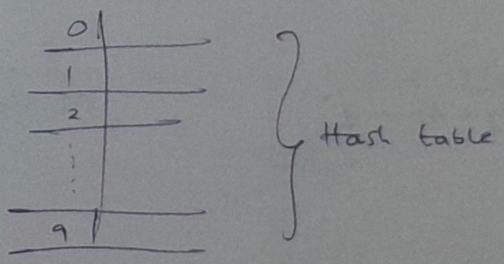
Types of Hash functions

(1) Division Method :

$h(k) = k \bmod M$

Collision :

Seperate - chaining



(2) Mid Square Method :

Square of the Data,

No. of elements as the middle element

= ~~No. of~~ size of the hash table

Ex. $(36)^2 = 1296$

If Hash size = 10, Middle element = 2 (or) 9

If Hash size = 100, Middle element = 29

(3) Digit Folding Method:

Divide data digit by digit

Ex. Data : 123456

Split into 3 sets

12 | 34 | 56

k_1 k_2 k_3

$12 + 34 + 56 = 102$ → Should be placed in Hash table 02.

[Ignore 1 As carry]

(2 Partitions — @ 10^2 Hash size)

(4) Multiplication Method:

Constant A

$$[M \times ((key \times A) \% 1))]$$ M: Hash size

1/4/24

Q) Hash table size =, $m = 10^4$

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

for $A = \frac{\sqrt{5}-1}{2}$

key 123456 mapped to location

Sol:

$$A = \frac{2.236-1}{2} = \frac{1.236}{2} = 0.618$$

$$k \times A = 123456 \times 0.618$$

$$= 76295.808$$

~~Infix to Prefix:~~

$76295.808 \bmod 1 = 0.808$

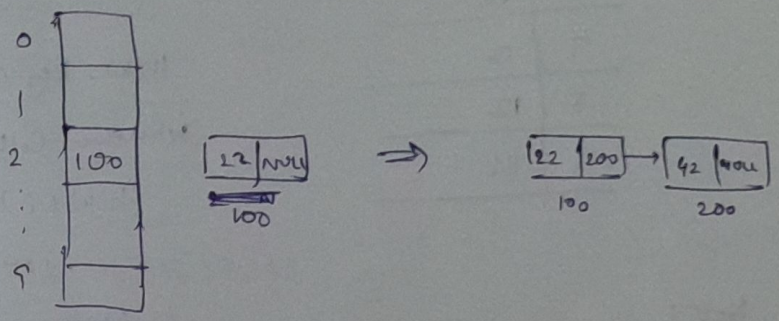
$0.808 \times 10^4 = 8080$

$\lfloor 8080 \rfloor = 8080$

∴ Stored in Hash table 8080

Seperate chaining Technique:

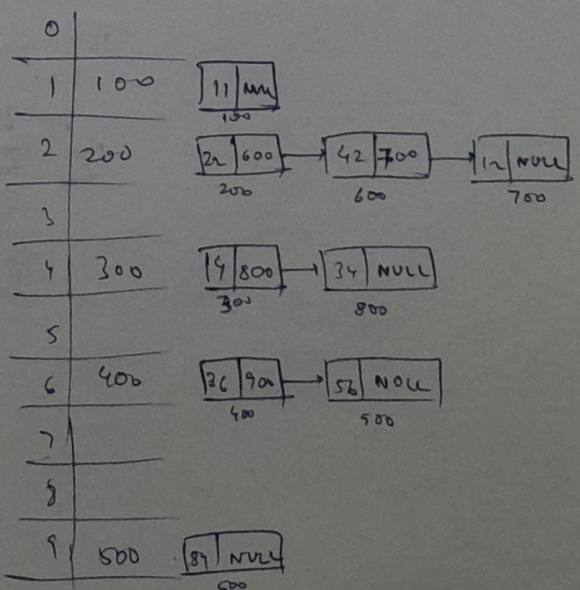
Ex. 22, 36, 11, 4, 42



Implement Array as Linked List as chain.

↳ size 10

Q) 22, 36, 11, 4, 42, 56, 89, 34, 12



2/4/24

→ Open Addressing Techniques :

(1) Linear Probing :

Ex. 22, 36, 11, 4, 42, 56, 89, 34, 12

0	
1	11
2	22
3	42
4	4
5	34
6	36
7	56
8	12
9	89

$22 \% 10 = 2$
$36 \% 10 = 6$
$11 \% 10 = 1$
$4 \% 10 = 4$
$42 \% 10 = 2$

$H(k) = key \% HT$

Whenever Collision,

$(h(k) + i) \% 10$

$i \rightarrow 1, 2, 3, \ldots$

$42 \% 10 = 2$

$(2+1) \% 10 = 3$ ✓

$56 \% 10 = 6$

$(6+1) \% 10 = 7$ ✓

$89 \% 10 = 9$

$34 \% 10 = 4$

$(4+1) \% 10 = 5$ ✓

$12 \% 10 = 2$

$2+1, 2+2, 2+\ldots, (2+6) \% 10 = 8$

↓ Not free ↓ Not free ↓ Not free ↓ free

Note :

No. of elements < Hash Size

Q) 12, 18, 13, 2, 3, 23, 5, 15

Hash size → 10

h(k) = k mod 10 & Linear Probing

Resultant Hash table?

Sol:

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

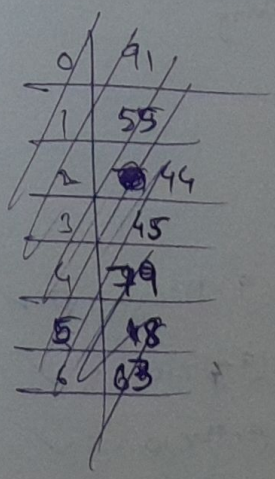
Q) 44, 45, 79, 55, 91, 18, 63

Hash size → 7

h(k) = k mod 7 & Linear Probing

Resultant Hash table?

Sol:



0	91
1	63
2	44
3	45
4	79
5	18
6	55

(2) Quadratic Probing:

Ex. 42, 16, 91, 33, 18, 27, 36, 62

$(h(k) + (i \times i)) \% HT$

0	36
1	91
2	42
3	33
4	
5	
6	16
7	27
8	18
9	

$(36 + 1) \% 10$
$= 7$ ✗

$(36 + 4) \% 10$
$= 0$ ✓

$(62 + 1) \% 10 = 3$ ✗
$(62 + 4) \% 10 = 6$ ✗
$(62 + 9) \% 10 = 1$ ✗
$(62 + 16) \% 10 = 8$ ✗
$(62 + 25) \% 10 = 7$ ✗
$(62 + 36) \% 10 = 8$ ✗
$(62 + 49) \% 10 = 1$ ✗

Q) 9, 19, 29, 39, 49, 59, 69

Hash size – 10

H = k mod 10, Quadratic Probing

59 – Which Index

Sol:

0	19
1	
2	
3	19
4	49
5	39
6	
7	59
8	29
9	9

9: $(9 + 1) \% 10 = 0$ ✓

19: $(9 + 4) \% 10 = 3$ ✓

29: $(9 + 9) \% 10 = 8$ ✓

39: $(9 + 16) \% 10 = 5$ ✓

49: $(9 + 25) \% 10 = 4$ ✓

59: $(9 + 36) \% 10 = 5$ ✗
$(9 + 49) \% 10 = 7$ ✓

21/4/24

Double Hashing :

$$[H_1(k) + i \cdot H_2(k)] \% N$$

↳ Hash size

Ex. 79, 69, 98, 72, 14, 50

$N = 13$ [Hash Table size]

$H_1(k) = k \bmod 13$

$H_2(k) = 1 + (k \bmod 11)$

Sol: $[H_1(k) + i H_2(k)] \% 13$

$\Rightarrow [k \bmod 13 + i + i(k \bmod 11)] \bmod 13$

0	
1	79
2	
3	
4	69
5	14
6	
7	98
8	72
9	
10	
11	50
12	

$79 \% 13 = 1$ ✓

$69 \% 13 = 4$ ✓

$98 \% 13 = 7$ ✓

$72 \% 13 = 7$ ✗ Collision

$H_2(k) = 1 + 6 = 7$ ✗

Collision Again

$(7 + i\,7) \% 13$

$i = 1$ ✗

$i = 2$ ✓ 8

$14 \% 13 = 1$ ✗

$1 + 3 = 4$ ✗

$(1 + 4i) \% 13$

$i = 1$ ✓ 5

$50 \% 13 = 11$ ✓

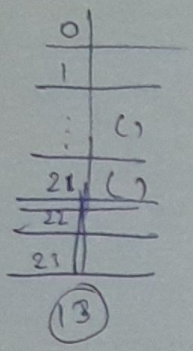
Q) $h_1 = k \bmod 23$

$h_2 = 1 + k \bmod 19$

$N = 23$

$23 \times 3 = 69$

$19 \times 4 = 76$



$90 \bmod 23 = 21 = h_1$

$1 + 90 \bmod 19 = 15 = h_2$

$(21 + 15i) \% 23$

$36 \% 23 = 13$

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

$h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod n)$

where $n = m - 1$

$m = 701$ for key $= 123456$

$h_1(k) = k \bmod 701$

$h_2(k) = 1 + k \bmod 700$

$h_1(k) = 123456 \bmod 701$

$= 80$

$h_2(k) = 1 + 123456 \bmod 700$

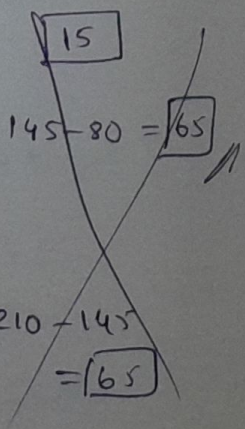
$= 1 + 256 = 257$

$h(k,i) = (80 + 257i) \bmod 701$

$= 337$

$h(k,i) = (80 + 514) \bmod 701$

$= 594$



$594 - 337$

257

GRAPHS:

→ **Node**: Fundamental building block of Graph

Isolated Graph: No Edges

(degree 0) ⟶ Isolated Vertices

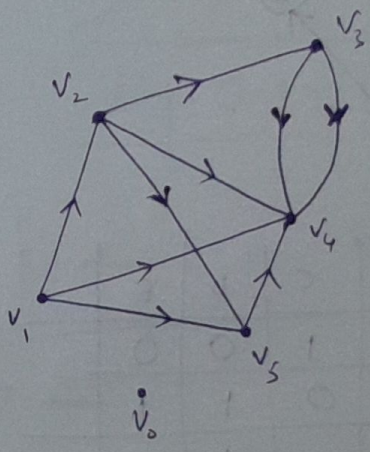
→ **Edge**: Connection between Nodes

→ **Weighted Graph**:

Edges have Weight

→ **Degree**:

A vertex is directly connected with how many other vertices is called Degree.



	(In-degree) Incoming	(Out-degree) Outgoing	Power
V_0	0	0	Total = 0 → Isolated Vertex
V_1	0	3	3
V_2	1	3	4
V_3	1	2	3
V_4	5	0	5
V_5	2	1	3

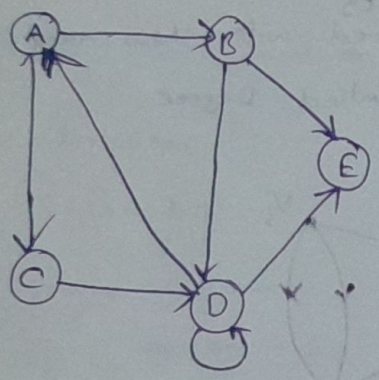
Power = ID + OD

→ Path : No. of Edges is length of Path from one node to another

→ Cycle : Starts and ends at same node.

→ Representation of Graphs:

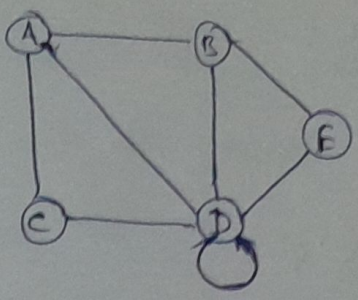
- Adjacency Matrix :



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

2D Array → Connection b/w vertex i and vertex j

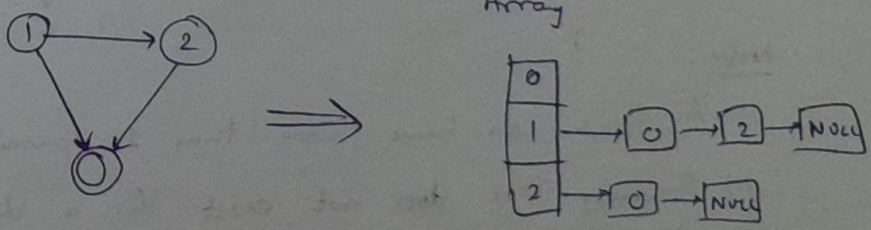
(i, j) represents connection b/w vertex i to vertex j



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

Hence symmetric for Undirected Graph.

- Adjacency List :



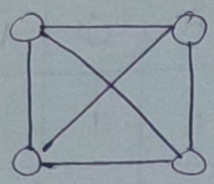
More ~~efficient~~ Memory-efficient for Sparse Graphs.

→ Application of Graph :

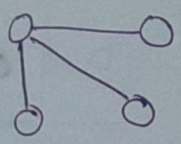
- Spanning Tree :

  ↳ Subset of Graph 'G' such that all vertices are connected using minimum possible number of edges.

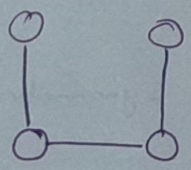
Ex Graph G =



G_1 (Spanning Tree) :



G_2 (Spanning Tree) :



Spanning Tree → Does not have cycles

Note :

(1) A Graph can have more than 1 Spanning Tree.

(2) Spanning Tree does not exist for a discontinuous graph

(3) Graph – 'N' vertices

Spanning tree – 'N-1' edges

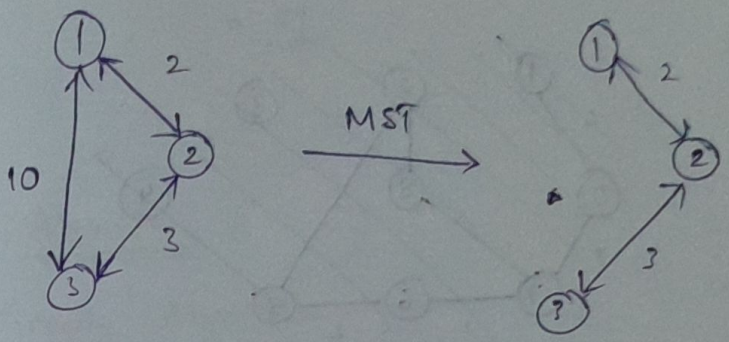
10/1/24

- **Minimum Spanning Tree :**

→ Applicable to weighted Graphs

→ Spanning Tree which has least summation of weights

Ex.



Note :

Loops are not allowed

- **Kruskal's Algorithm :**

Greedy Algorithm

Finds Minimum Spanning ~~Graph~~ Tree of the connected weighted Graph.

i.e Finds Minimum weight.

Steps :

(1) Sort Edges

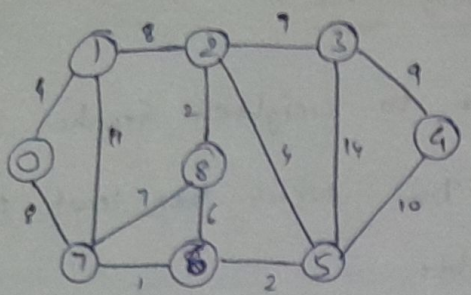
(2) Create empty set to store edges of MST

(3) Iterate over edges

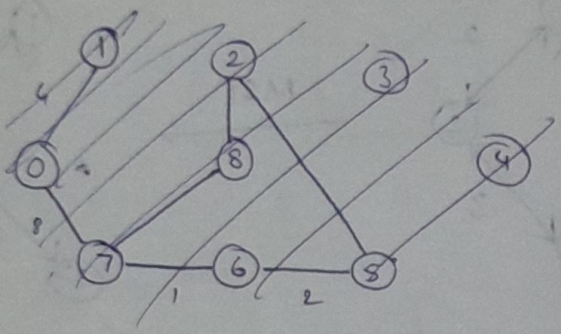
(4) Check for cycle

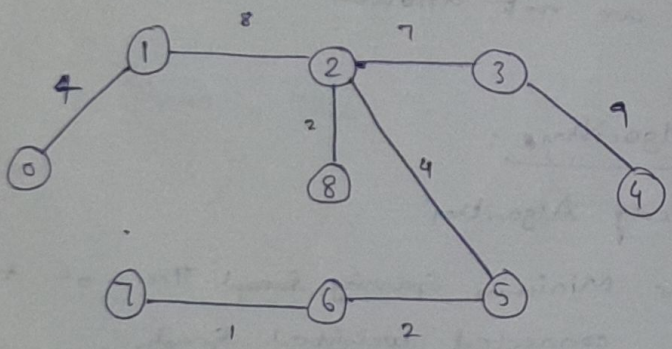
Ex.

Q)



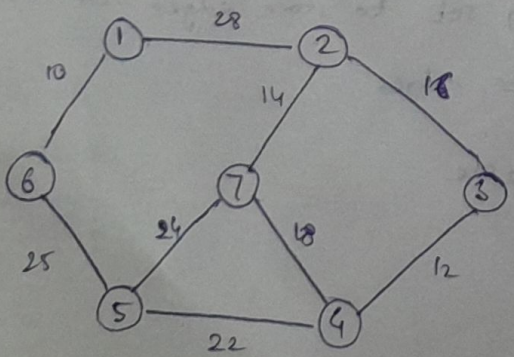
Sol:



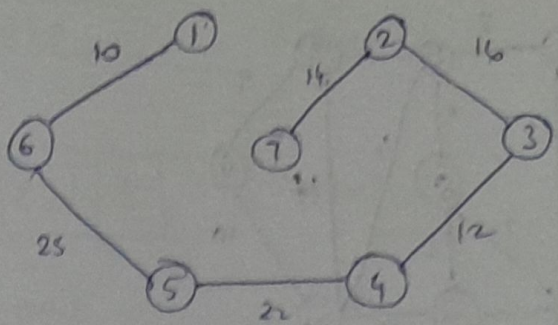


1 + 2 + 2 + 4 + 4 + 8 + 7 + 9 = 37

Q)



Sol:



10 + 12 + 14 + 16 + 22 + 28

= 89

10 ✓
12 ✓
14 ✓
16 ✓
18 X cycle
22 ✓
24 X cycle
25 ✓

- Prim's Algorithm :

  — Greedy Algorithm

  → Starts with a vertex & Visit all vertices such that min. no. of edges & min. summation of edges.

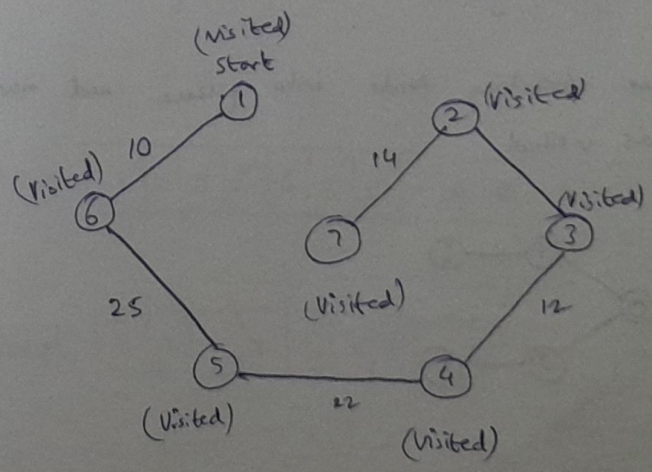
Steps:

(1) start vertex — mark it as visited (starting vertex)

(2) Choose edges with lowest weight & visit all visited vertices

(3) Add vertices

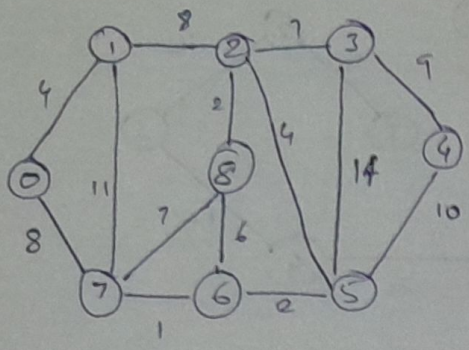
Ex.



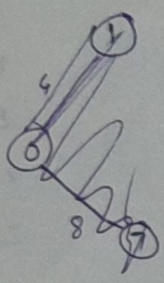
1→6 : 10 (min)
6→5 : 25 (min)
5→4 : 22 (min)
4→3 : 12 (min)
3→2 : 16 (min)
2→7 : 14 (min)

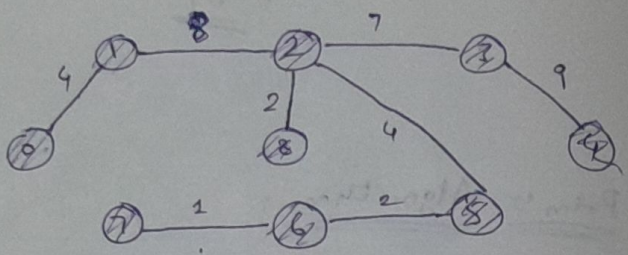
12/4/24

8)



Sol:





~~Sum = 37~~ Sum = 37

→ Time Complexity:

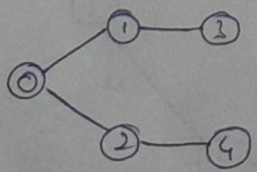
$O(V^2)$ → Adjacency List

$O(E \times \log V)$ → Heap

→ Breadth First Search : (BFS)

Vertex → Neighbours

(1) Enqueue starting Node into queue and mark it as visited.



1	2	3	4

Queue

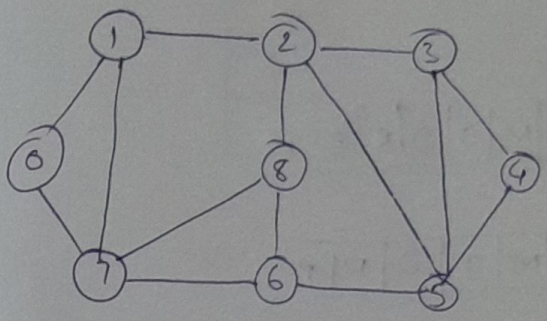
0	1	2	3	4

Visited

(or)

0	2	1	4	3

q)



Start with vertex 1 :

..

Sol:

0	2	4	8	5	3	6	

Queue

1	0	7	2	8			

Visited

Queue

1	0	7	2	6	8	5	3	4	0

Visited

1	0	7	2	6	8	5	3	4	0

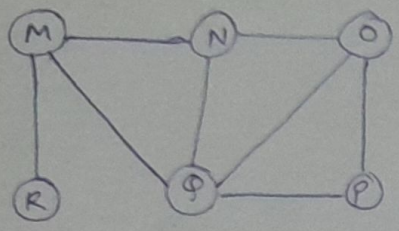
— Time Complexity : $O(V+E)$

V : No of Vertices

E : No of Edges

15/4/24

9)



Queue | N | M | Q | O | R | P |

VV | N | M | Q | O | R | P |

— Depth first search : (DFS)

Steps :

(1) Choose any vertex as starting

(2) Mark it as visited

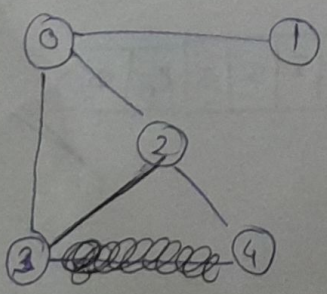
(3) Explore one of its neighbours that hasn't been visited.

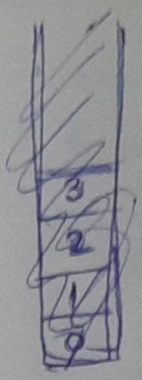
Repeat until no unvisited neighbours

(4) Backtrack to previous and find unvisited neighbours

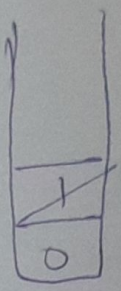
DFS ⟶ Stack is used

Ex.

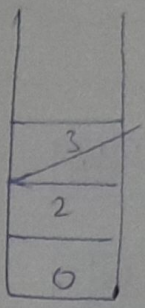




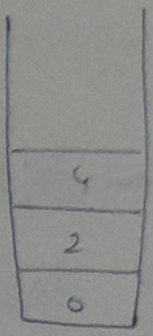
VV: 01



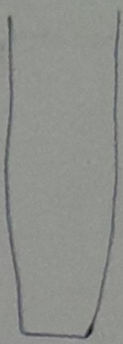
VV: 0 1



VV: 0 1 2 3



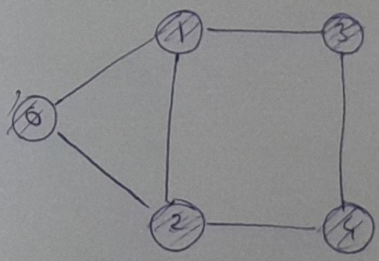
VV: 0 1 2 3 4

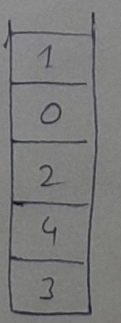


VV: 0 1 2 3 4

DFS complete when stack is empty

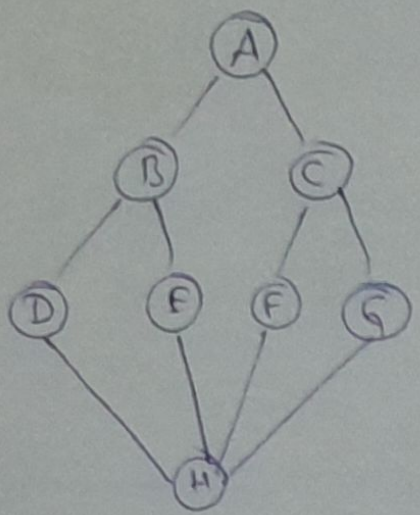
(3)





VV: 3 4 2 0 1

Q1)

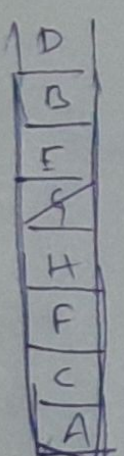


BFS :

Queue | A | ~~B~~ | ~~C~~ | ~~D~~ | E | F | G | H |

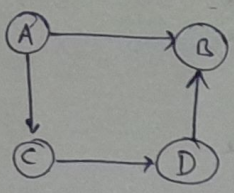
VV | A | B | C | D | E | F | G | H | ✓

DFS :



VV : ACFHGEBD ✓

17/4/24:



ABCD

Directed Acyclic Graph (DAG)

Simplex Communication
A → B and not B → A

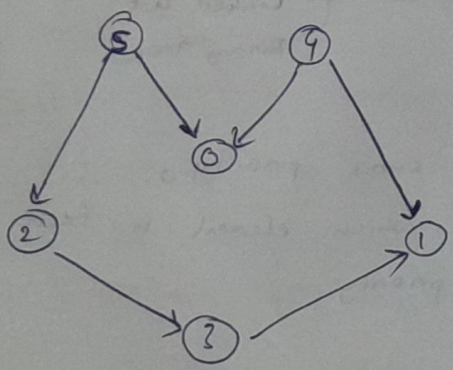
Note:

Topological Sorting only possible for DAGs

Two ways:

(1) DFS
(2) Inorder

(1) Ex



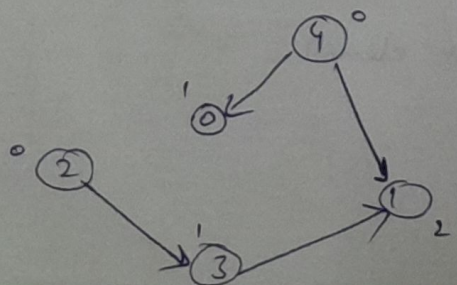
In Degrees:

V(0) = 2
V(1) = 2
V(2) = 1
V(3) = 1
V(4) = 0
V(5) = 0

Lowest In-Degree: 4 and 5

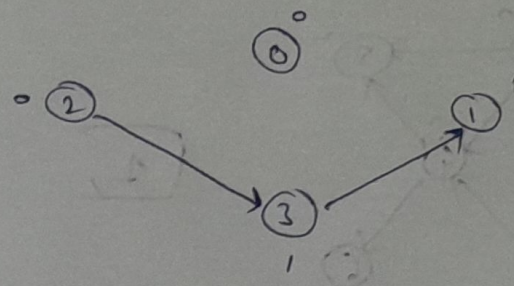
Take 5:

Remove 5 and disconnect edges:



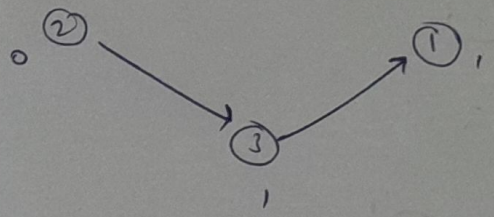
5,

Disconnect 2 (or 4):



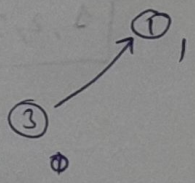
5, 4

Disconnect 0:



5, 4, 0

Disconnect 2:



5, 4, 0, 2

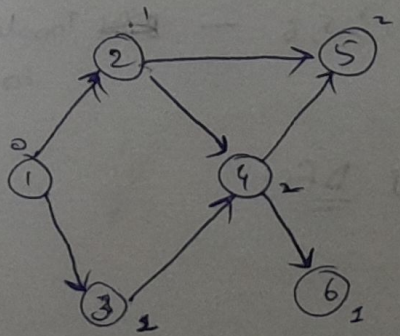
Disconnect 3

| 5 | 4 | 0 | 2 | 3 |

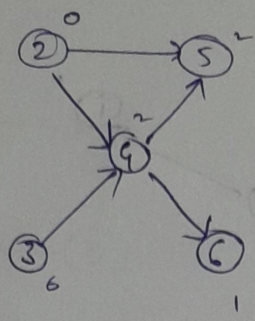
Disconnect 0

5, 4, 0, 2, 3 → ~~DFS~~ Topological Sorting

g)



Disconnect 1

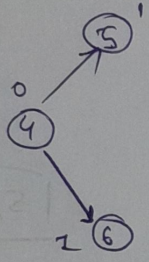


1,

Disconnect 2

1, 2

Disconnect 3:



1, 2, 3

Disconnect 4:

5 0

1, 2, 3, 4

6 0

Disconnect 5 — 1, 2, 3, 4, 5

Disconnect 6 — 1, 2, 3, 4, 5, 6 — ~~DFS~~ Topological sorting

(2) Topological Sorting Using DFS:

Steps:

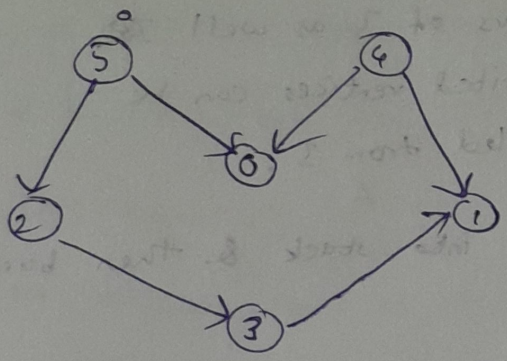
(i) Initialise

(ii) Perform DFS

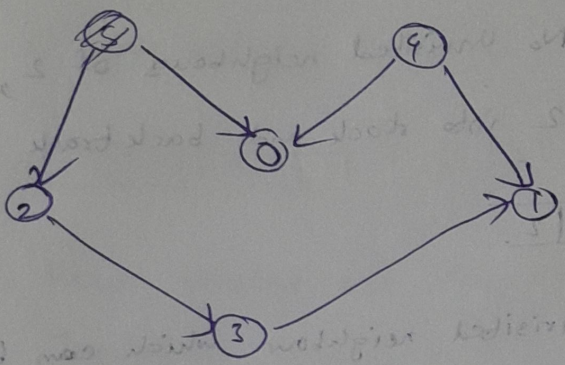
(iii) Recursion

(iv) Backtrack

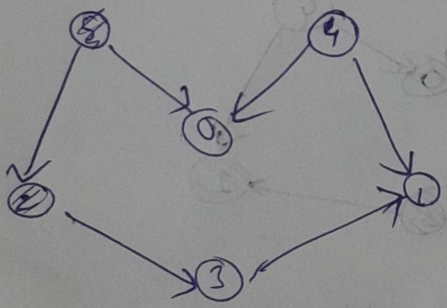
Ex.



First, 5 is visited

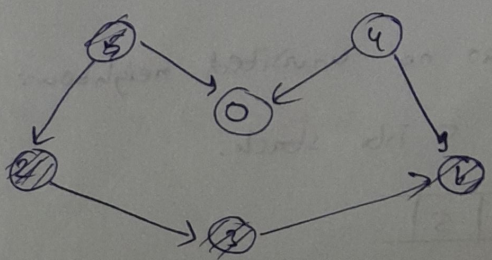


2 is visited



Then Go to 3,

Then Go to 1



Now, No other vertex can be visited

∴ Push 1 into the stack and then Backtrack to 3.

No Neighbours of 3 as well.

i.e No Unvisited vertices can be travelled from 3.

∴ Push 3 into stack & then backtrack to 2

Stack:

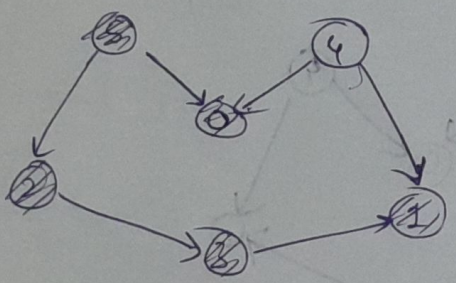
1	3

Then No Unvisited neighbours of 2,

Push 2 into stack and backtrack to 5

1	3	2

5 has unvisited neighbour which can be travelled to. So 0 is visited



Now, We reached deadend again

∴ Push 0 into stack & Backtrack to 5

1	3	2	0

Now, 5 has no unvisited neighbours.

∴ Push 5 into stack.

1	3	2	0	5

As 5 is the node we started from, we have come back to 5. DFS is complete.

Now, start DFS from unvisited nodes

↑ i.e. Here, only one node → 4

Start DFS from 4:

No neighbours can be visited.

∴ Push 4 into stack.

Again, We have come back to starting node of DFS,

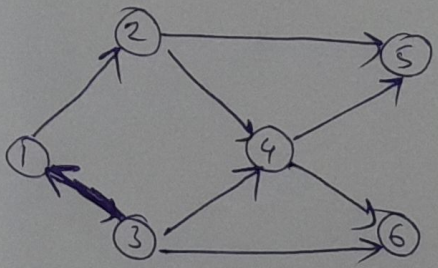
∴ DFS ended.

1	3	2	0	5	4

Then, keep popping and the resultant is the Topological sorting.

i.e. 4, 5, 0, 2, 3, 1

(ף)



Start from 1

1 → 2 → 5, Deadend, Push 5 into stack & Backtrack to 2

2 → 4 → 6, Deadend, Push 6 into stack & Backtrack to 4

↓ Push into stack

Backtrack to 3

↓ Push into stack

Backtrack to 1

Stack: | 5 | 6 | 4 | 2 |

Push 1 into stack As no other unvisited neighbours can be travelled to.

DFS from 1 is complete

Start DFS from unvisited i.e. 3

No other neighbour, Push into stack

Stack: | 5 | 6 | 4 | 2 | 1 | 3 |

Topological Sorting:

5, 6, 4, 2, 1, 3

26/4/24

Priority Queue :

- Type of Queue → front & rear
  → Enqueue
  → Dequeue, ...

Priority Queue

Ascending order Priority Queue	Descending order Priority Queue
(Priority increases throughout)	(Priority reduces throughout)

Highest Priority element — Enqueued first, dequeued first

Priority Queue → Array
→ Linked List
→ Binary Tree

If two elements have same priority, arrival time of which element is faster, that is given more priority.

top to bottom & left to right

Enqueue : Dequeue all elements with less priority, enqueue element and then enqueue rest of the elements.

Dequeue : Element having highest priority will be dequeued first

peek : front element