SUMEDH

- Topics

1. Stack
2. Queue
3. Trees
4. Hashing
5. Graphs

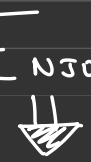
Stacks : Basic Ops
Array
Linked list
Infix Postfix
Hanoi
Function Calls

Queue : Basic Ops
Arr
LL
Apps
Types

Trees : Binary tree
Traversal
BST
AVL
B-TREE , HEAP , HASH

Graphs : BFS
DFS
MST
Topological sort
PRIM
KRUSKAL

ENJOY



- Stacks

Array Implementation

Program :-

```
#include <stdio.h>
#include <stdlib.h>
#define N 5;

int stack [N];
int top = -1;

int isempty () {
    return top == -1;
}

int isfull (){
    return top == N-1;
}

void push () {

    int x;
    printf (" Enter data : ");
    scanf ("%d", &x);

    if ( isfull() ) {
        printf (" Overflow ");
    }
    else {
        top ++;
        stack [top] = x;
    }
}

void pop () {
    int item;
    if ( isempty() ) {
    printf ( " Underflow " );
    }
    else {
        item = stack [top]
        top --;
    printf ("%d deleted ", item);
    }
}

void display () {

    int i

    for (int i= top; i>=0; i--){

    printf ( "%d \n", stack[i]);
    }
}

int main () {

        :


    }
```

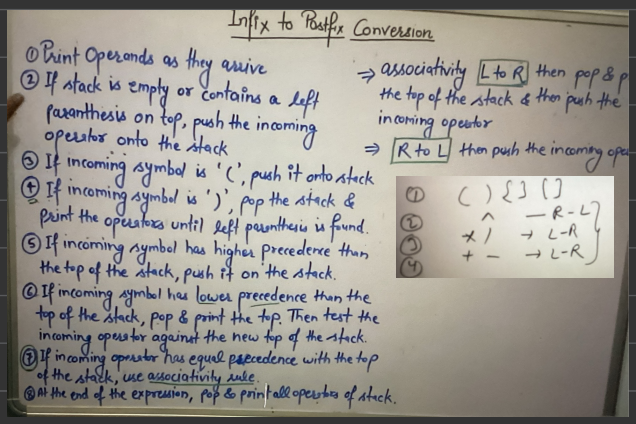
Linked List Implementation

Program :-

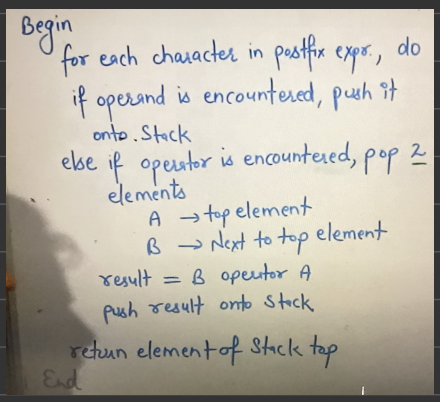
```
#include <stdio.h>
#include <stdlib.h>

Struct node {
    int data ;
    Struct node * next;
}
Struct node * top = 0

 void push (int x) {

   Struct node * newnode ;
newnode = (struct node *) malloc (sizeof (struct node));

    newnode → data = x ;
    newnode → next = top;
        top = newnode ;
}

void pop () {

 struct node * temp
    temp = top;
  if ( top == -1) {
     printf (" Underflow ");
 }
  else {

 printf (" %d deleted ", top -> data );
     top = top → next
       free (temp);
 }
}

void display () {

 Struct node * temp
    temp= top
```

```
if ( top == 0) {
    printf(" empty ");
}
else {
    while (temp != 0){
        printf ("%d \n" temp->data);
            temp = temp -> next
    }
}

int main () {

    }
```

Infix to postfix



Post fix evaluation



Hanoi

```
START
Procedure Hanoi(disk, source, dest, temp)

  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, temp, dest)    // Step 1
    move disk from source to dest          // Step 2
    Hanoi(disk - 1, temp, dest, source)    // Step 3
  END IF

END Procedure
STOP
```

- Queue — deletion frm front & insertn from rear

Array Implementation

Program :-

```
# include <stdio.h>
# define N 5
int queue [N] ;
int front = -1 ;
int rear = -1 ;

void enqueue (int x)

  if (rear = N-1) {
      printf (" Overflow ");
  }
else if ( front == -1 && rear == -1) {
      front = rear = 0
        queue [rear] = x;
  }
 else {
      rear ++ ;
    queue [rear] = x;
  }
}

void dequeue () {

  if ( front == -1 && rear == -1) {
        printf (" Underflow ");
    }
  else if ( front == rear) {
        front = rear = -1;
    }
   else {
      printf ("%d deleted ", queue [front] );
        front ++ ;
  }
}
```

```
void display () {

    if ( rear == -1 && front == -1) {
        printf(" Underflow ");
    }
    else {
        for (int i= front ; i <= rear ; i++) {
            printf (" %d\n", queue[i]);
        }
}

int main () {

    }
```

Linked list implementation

Program :-

```
#include <stdlib.h>
#include <stdio.h>

struct Node {
    int data;
    struct Node* next;
};
struct Node* front =0;
struct Node* rear = 0;

void enqueue (int x) {
    struct Node* newnode;
newnode = (struct node*) malloc ( sizeof (struct Node*));
    newnode -> data = x
    newnode -> next = 0;

    if ( front == 0 && rear == 0) {
        front = rear = newnode;
    }
    else {
        rear -> next = newnode;
            rear = newnode;
    }
}
```

```
void dequeue () {
    struct Node * temp;
      temp = front
 if ( front == 0 && rear == 0) {
       printf (" Underflow ");
    }
 else {
       printf ("%d deleted ", front -> data);
            front = front -> next;
             free (temp);
      }
}

void display () {
     struct Node * temp ;
       temp = front;
 if ( front == 0 && rear == 0) {
      printf (" Underflow ");
   }
 else {
       while (temp != 0) {
          printf ("%d \n", temp -> data);
             temp = temp -> next;
         }
     }
  }

int main () {

    }
```

Circular Queue

Program :-

```c
#include <stdio.h>
#define MAX_SIZE 5 // Change this to any desired maximum

typedef struct {
    int queue[MAX_SIZE];
    int front;
    int rear;
} CircularQueue;

// Initialize the queue
void initializeQueue(CircularQueue *cq) {
    cq->front = -1;
    cq->rear = -1;
}

// Function to check if the Circular Queue is empty
int is_empty(CircularQueue *cq) {
    return cq->front == -1;
}

// Function to check if the Circular Queue is full
int is_full(CircularQueue *cq) {
    return (cq->rear + 1) % MAX_SIZE == cq->front;
}

// Function to add an element to the Circular Queue
const char* enqueue(CircularQueue *cq, int data) {
    if (is_full(cq)) {
        return "Queue is full";
    }
    if (is_empty(cq)) {
        cq->front = cq->rear = 0;
    } else {
        cq->rear = (cq->rear + 1) % MAX_SIZE;
    }
    cq->queue[cq->rear] = data;
    return "Enqueued element";
}

// Function to remove an element from the Circular Queue
const char* dequeue(CircularQueue *cq) {
    if (is_empty(cq)) {
        return "Queue is empty";
    }
    int data_to_dequeue = cq->queue[cq->front];
    if (cq->front == cq->rear) {
        cq->front = cq->rear = -1; // Queue is now empty
    } else {
        cq->front = (cq->front + 1) % MAX_SIZE;
    }
    printf("Dequeued element: %d\n", data_to_dequeue);
    return "Dequeued element";
}

// Function to get the front element of the Circular Queue
const char* peek_front(CircularQueue *cq) {
    if (is_empty(cq)) {
        return "Queue is empty";
    }
    printf("Front element: %d\n", cq->queue[cq->front]);
    return "Front element retrieved";
}

// Function to get the rear element of the Circular Queue
const char* peek_rear(CircularQueue *cq) {
    if (is_empty(cq)) {
        return "Queue is empty";
    }
    printf("Rear element: %d\n", cq->queue[cq->rear]);
    return "Rear element retrieved";
}

int main() {
    CircularQueue cq;
    initializeQueue(&cq);

    // Testing the circular queue operations
    printf("%s\n", enqueue(&cq, 10));
    printf("%s\n", enqueue(&cq, 20));
    printf("%s\n", enqueue(&cq, 30));
    printf("%s\n", peek_front(&cq));
    printf("%s\n", peek_rear(&cq));
    printf("%s\n", dequeue(&cq));
    printf("%s\n", dequeue(&cq));
    printf("%s\n", dequeue(&cq));
    printf("%s\n", dequeue(&cq)); // Should indicate queue is empty

    return 0;
}
```

Double Ended Queue

Skipped

- Trees

- Binary Tree Representation

Program :-

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data,
    struct node * left, * right ;
};

struct node * create () {

        int x ;
    struct node * newnode ;
newnode = (struct node *) malloc (sizeof (struct node *)) ;
  printf (" Enter data & -1 if no node : ");
  scanf (" %d ", &x ) ;

    if (x == -1) {
        return 0;
      }
   else {
      newnode -> data = x ;

  printf (" Enter left child of %d ", x);
     newnode -> left = create ();

  printf (" Enter right child of %d ", x);
     newnode -> right = create();

    return newnode ;
  }
}

int main () {
   struct node * root;
      root =0 ;
    root = create ();
  }
```

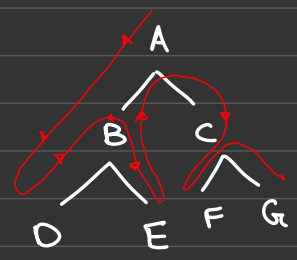
• Tree Traversal

1. Preorder Traversal

(Root, left, right)

- Here, first we traverse the root
- Then go to left subtree n go on
- In left subtree, first the left child then right child
- Then the right subtree

eg

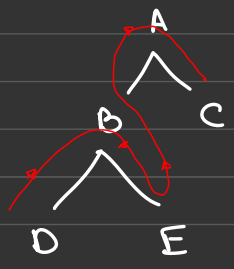


order of traversal : A → B → D → E → C → F → G

2. Inorder Traversal (left, root, right)

- Here go to the left-child of left-most subtree
- The root and the right sub-tree

eg

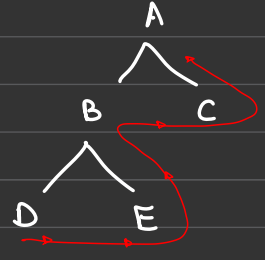


Order D → B → E → A → C

3. Post order traversal

- Here first the left-child of left-most subtree
- Then the right child
- Then parent of left subtree
- The left child of right subtree then the rgh child n parent
- At last, root

eg



Order : D → E → B → C → A

Program :-

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node * left, * right;
};

struct node * create () {

        // same as the the prog above //

}

void preorder (struct node * root) {
    if (root == NULL) {
        return;
    }
  else {
    printf ("%d", root -> data);
    preorder (root -> left);
    preorder (root -> right);
  }
}

void inorder (struct node * root) {
        if (root == NULL) {
            return;
        }
  else { inorder (root -> left);
         printf ("%d", root -> data);
         inorder (root -> right);
    }
}
void postorder (struct node * root) {
    if (root == NULL) {
        return;
    }
  else { preorder (root -> left);
         preorder (root -> right);
```

```
        printf ("%d", root->data);
    }
}

int main () {

    struct node * root = NULL;
      root = create ();

    printf (" Preorder Traversal : ");
    preorder (root);

    printf (" Inorder Traversal : ");
     Inorder (root);

    printf ("Postorder Traversal : ");
    postorder (root);

     return 0;
}
```

- Height of Binary tree

- Function :-

```
int calch (struct node * root) {
      if (root == NULL) {
            return -1;
        }
    else {
     int lh = calch (root->left);
     int rh = calch (root->right);

     return 1+ (lh > rh ? lh : rh);
    }
}

int main () {

    int h = calch (root);
}
```

- Binary Search Tree (BST)

- Here, the left child is less than parent/root node
- Right child greater that parent / child

Program :-

```
#include <stdio.h>
#include <stdlib.h>

 Struct node {
    int data;
   Struct node *left, *right;
};

Struct node * createnode (int data) {
    struct node * newnode;
newnode = (struct node *) malloc (sizeof (struct node *));
      newnode -> data = data;
      newnde -> left = newnode -> right = 0;
        return newnode;
  }

Struct node * insert (struct node * root , int data) {

      if (root == 0) {                                  [if data< root then left else righ]
        return createnode (data);
    }
  else {
      if (data < root -> data) {
          root -> left = insert (root -> left, data);
      }
    else {
      root -> right = insert (root -> right , data);
   }
  return root;
   }
}
```

```
struct node * search (struct node * root, int data) {

        if (root == 0){
            return root;
        }
    else    {
        if ( data < root -> data){
            return search (root -> left , data);
            }
        else {
            return search ( root -> right, data);
} } }
```

[if data< then search left orelse rgt]

```
Struct node * delete (struct node * root , int data) {
    Struct node * temp;
      if (root == 0){
          return 0;
      }

  else  {

      if ( data < root -> dat) {
      root ->left = delete (root -> left , data);
    }
else if ( data > root -> data) {
     root -> right = delete ( root -> right , data);
   }
  else {

       if ( root -> left == 0){
            temp = root -> right
            free (root) ;
            return temp;
        }
    else if ( root -> right == 0){
            temp = root -> right
            free (root);
            return temp;
        }
    else  { temp = root -> right
            while ( temp && temp -> left != 0){
                temp = temp -> left
```

- **Case 1: The node has no children (leaf node):**
  - Simply free the node's memory and return `NULL`.
- **Case 2: The node has only one child (left or right):**
  - Return the child node (left or right), replacing the node to be deleted.
- **Case 3: The node has two children:**
  - Find the minimum value node in the right subtree (called the "in-order successor").
  - Replace the node's value with this successor's value.
  - Delete the successor node from the right subtree.

Case - 1

Case - 2

Case - 3

```
            root -> data = temp -> data;
            root -> right = delete (root -> right , temp -> data);
        }
    }
    }
    return root;
}

void inorderdisp (struct node * root){
        if (root != 0){
          inorderdisp (root -> left);
          print ("%d ", root -> data),
          inorderdisp (root -> right);
          }
    }

int main (){
    struct node * root =0;
```

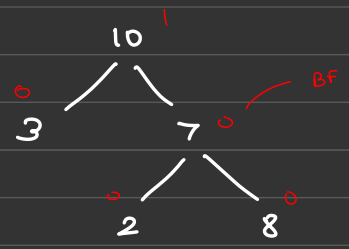
```
    // Manually creating the BST by inserting nodes
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    printf("BST in-order traversal before deletion: ");
    inorderTraversal(root);
    printf("\n");

    // Search for a node
    int searchValue = 40;
    if (search(root, searchValue)) {
        printf("Node %d found in the BST.\n", searchValue);
    } else {
        printf("Node %d not found in the BST.\n", searchValue);
    }

    // Delete a node
    root = deleteNode(root, 20);
    printf("BST in-order traversal after deleting 20: ");
    inorderTraversal(root);
    printf("\n");

    root = deleteNode(root, 30);
    printf("BST in-order traversal after deleting 30: ");
    inorderTraversal(root);
    printf("\n");
```

```
    return 0;
}
```

- AVL Trees
  - It is a BST
  - Here there a BF (Balanced factor)
  - BF = height of (L subtree - R subtree)

L only {-1, 0, 1}

eg



An AVL tree causes imbalance, when any one of the following conditions occur:

case 1 :-

An insertion into the left subtree of the left child of node α.

case 2 :-

An insertion into the right subtree of the left child of node α.

case 3 :-

An insertion into the left subtree of the right child of node α.

case 4 :-

An insertion into the right subtree of the right child of node α.

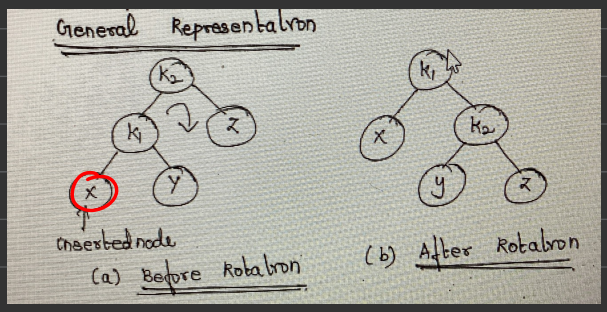
These imbalances can be overcome by

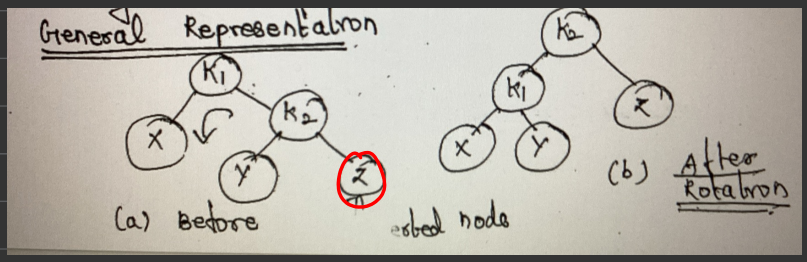
(1) single Rotation
(2) Double Rotation

case 1 & case 4 => single Rotation : (L-L / R-R)

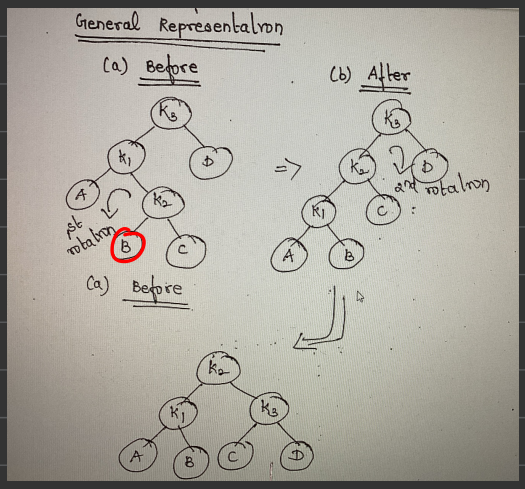
Case 2 & case 3 => Double Rotation (L-R / R-L)

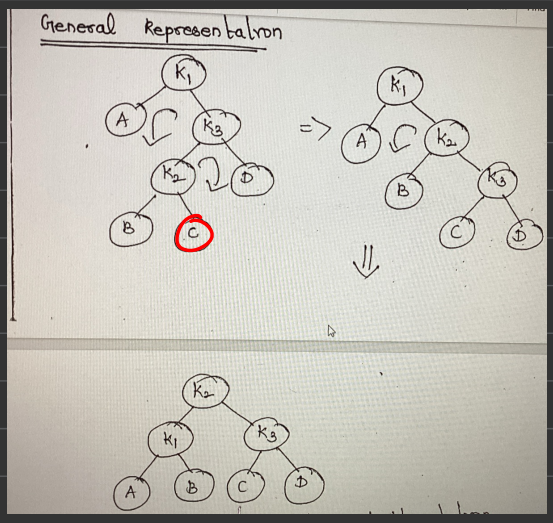
- Single rotation (Case -1 , 4)





- Double Rotation (Case -2 , 3)

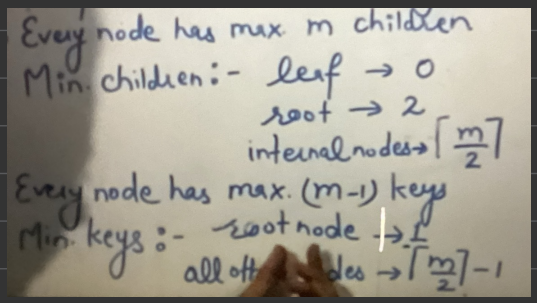




- B- trees

  └ extension of BST

- Can have more than 2 child
- Has order m (5, 3, 4 ...)
- Has more than one key



Skipped HEAP

N

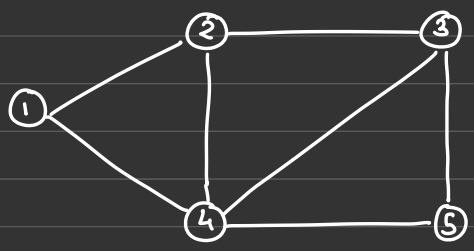
HASHING

- Graphs

  To ways to represent :-

  1. Adjacent Matrix
  2. Adjacent list

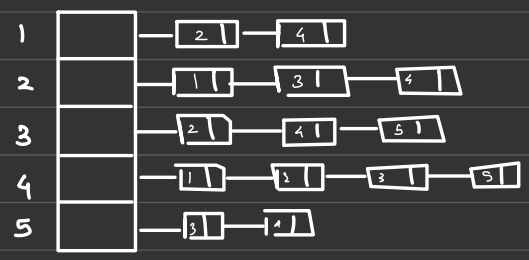
- Adjacent Matrix

eg 

— [here there are no loops i.e 

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

- Adjacent list

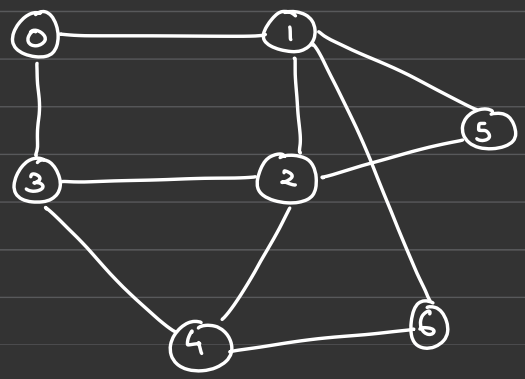


Complexity :- (n+2e)

- Graph Traversal

Two types : 1. BFS (Breadth first Search)
2. DFS (Depth first Search)

- BFS

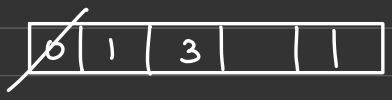
eg 

Step : 1. Take any node as root node
2. follow Queue Structure

eg Take 0 as root node

= 0

Step -3 Go for adjacent most nodes of 0

queue 

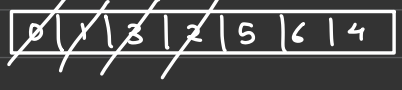
result : 0 1

Step -4 : Go for adjacent (uninserted, unvisited) nodes of '1'

queue : 

result : 0 1 3

Step -4 : Similar to other adj nodes of '3'

queue : 

result : 0 1 3 2

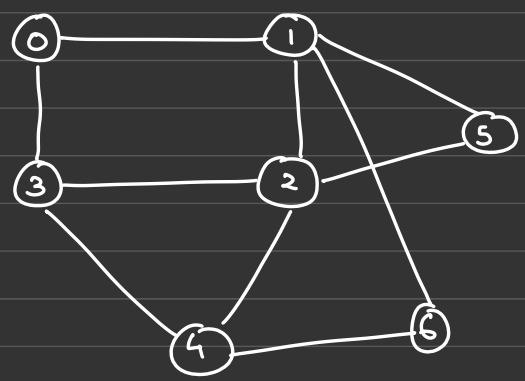
⋮

result : 0 1 3 2 5 6 4

- DFS

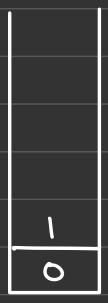
It follows Stack Structure

eg :



Step - 1 : Take 0 as root node

Stack :



res : 0, 1

Step - 2 : Take a adj vertex of '0', lets say 'i', the take any of its adj vertex n go on

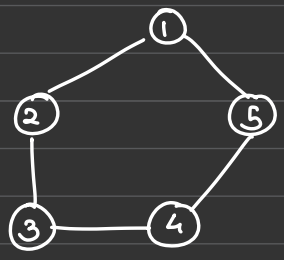
:

result : 0, 3, 2, 4, 6,

Note : When u go deep n find vertices already visited, then u gotta back-track but also pop that element n then go back

- Minimum Spanning tree

eg



vertices edges

G (V, E)

G' (V', E') [Spanning tree]

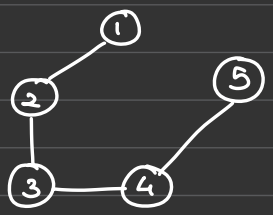
* A graph can have >1 spanning tree

Relation : V' = V

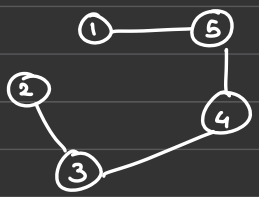
E' = Subset E

= |V| - 1

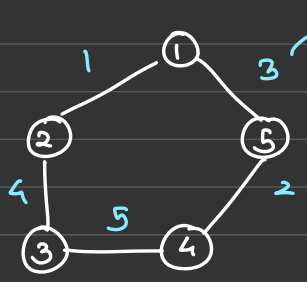
ST :



or



MST



[Here, the ST with less weight is MST]

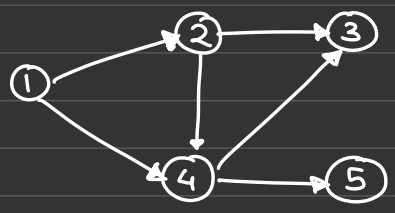
- Topological Sorting

It's a linear ordering of vertices, every directed edge ab, It shld go from vertex a to b in ordr

[shld be DAG (directed acyclic graph)

eg To eat bread-jam
S-1 Take bread out
-2 Apply jam
-3 Toast
-4 Eat

eg



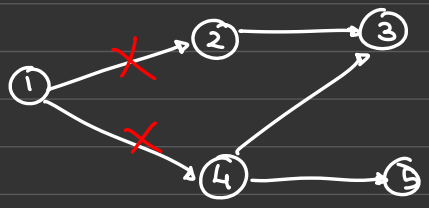
— [* It shld be always DAG]

IN° [toward it]
OUT° [away]

eg IN° of ① = 0
② = 1

Step - 1 : Choose the vertex with IN° = 0

2 : Now, delete all edges out going to that vertex



Now IN° of ② = 0
④ = 1

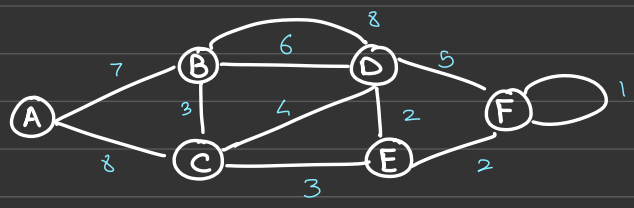
[updated

Step - 3 : Repeat Step 1 then Stp 2

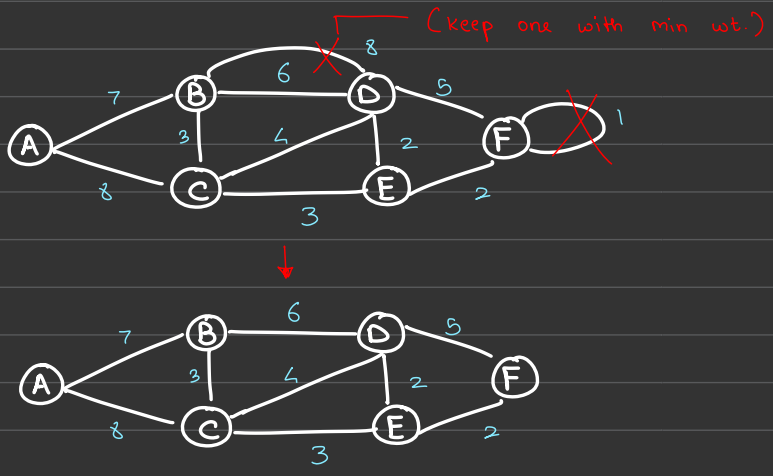
result : 1, 2, 4, 3, 5

• Prim's Algorithm for MST

eg



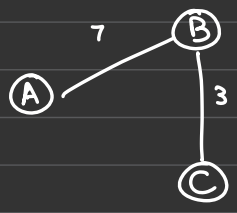
Step - 1 : Remove all loops n parallel edges



Step - 2 : Take any node as root, let's say (A)

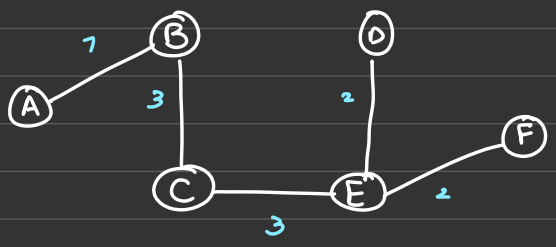
Step - 3 : Take edge from (A), one with min. wt

Step - 4 : Now, check adj edges from (B), one with minimum weight, * but also make sure to compare with (A)'s remaining edge



⋮

Step - 5 : Go on ....

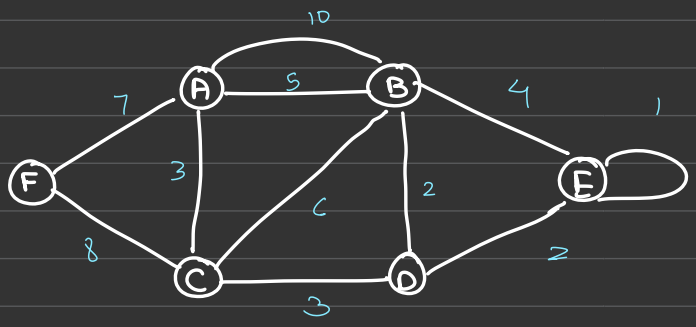


[Do it till u use all vertices n then stop]

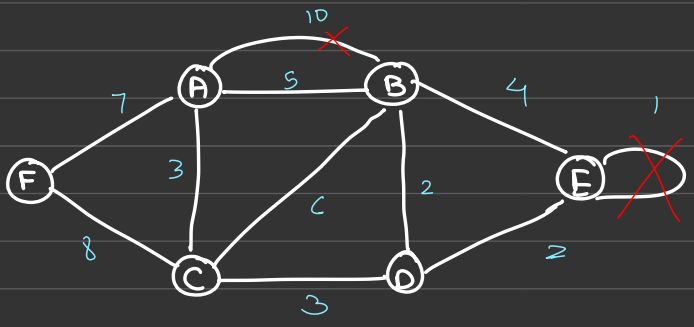
[MST]

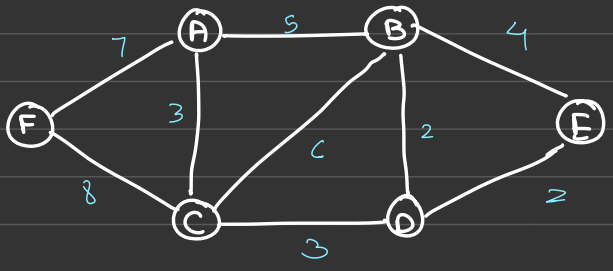
- Kruskal's Algorithm for MST

eg



Step - 1 . Remove all loops n parallel edges





Step - 2 : Arrange all edges in inc order of their weight

BD = 2
DE = 2
AC = 3
CD = 3
BE = 4
AB = 5
BC = 6
AF = 7
FC = 8

Step - 3 : Choose one with min wt. n star connecting (♥ make sure it does not contain cycle)

MST

[To check
V' = V
i.e 6 = 6
E' = V - 1
= 6 - 5
= 5