**1. BUBBLE SORT**

**Time Complexity : Worst Case : O(n^2);**

**Avg Case : O(n^2);**

**Best Case : O(n);**

**Space Complexity : O(1) a single additional memory space is required i.e.., temp variable;**

**Algo C#**

Int n = arr.Length;

for(int I = 0 ; i<n-1; i++)

{

Int flag = 0;

for(int 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;

flag = 1;

}

}

If(flag == 0)

break;

}

**Algo JS**

let n = arr.length;

for(let i = 0 ; i<n-1; i++)

{

let flag = 0;

for(letj = 0; j<n-i-1; j++)

{

If(arr[j]>arr[j+1]){

lettemp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

flag = 1;

}

}

If(flag == 0)

break;

}

**2.INSERTION SORT**

**Time Complexity : Worst Case : O(N^2)**

**Avg Case : O(N^2)**

**BEST CASE : O(N)**

**SPACE COMPLEXITY : O(1) => a single additional memory space is required i.e.., temp variable**

the **worst case** is when your list is in the exact opposite order you need. In that case:

* For the first item, you make 0 comparisons, of course.
* For the second item, you compare it to the first item and find that they are not in the right position; you've made 1 comparison.
* For the third, you compare it with both, and find that the third has to go to the top. You've made 2 comparisons.
* This goes on; for every following value, you make one more comparison.
* Finally, for the *n*th item, you make *n* - 1 comparisons.

If you add up the number of comparisons you make *for the worst case*, you'll see that it is 0 + 1 + 2 + ... + n-1, which is equal to (n^2 - n) / 2 comparisons for the worst case, which is O(n^2). (The part that determines the complexity is when we consider large n, in which case the n^2 term dominates)

**ALGO C#**

int n = arr.Length;

for(int i = 0; i<n-1; i++)

{

int temp = arr[i+1];

int j = I;

while(j>=0 && arr[j]>temp)

{

arr[j+1] = arr[j];

j--;

}

arr[j+1] = temp;

}

**ALGO JS**

let n = arr.length;

for(let i = 0; i<n-1; i++)

{

let temp = arr[i+1];

let j = I;

While(j>= 0 && arr[j]> temp){

arr[j+1] = arr[j];

j--;

}

arr[j+1] = temp;

}

**3.SELECTION SORT (**in-place comparison sorting algorithm, same as insertion sort**)**

**Time Complexity : Worst Case : O(N^2)**

**Avg Case : O(N^2)**

**BEST CASE : O(N^2)**

**SPACE COMPLEXITY : O(1) => a single additional memory space is required i.e.., temp variable**

**ALGO C#**

int n = arr.Length;

for(int I = 0; i<n-1; i++)

{

Int min = i;

for(int j = i+1; j<n ; j++){

if(arr[j]<arr[min]){

min = j;

}

}

Int temp = arr[min];

arr[min] = arr[i];

arr[i] = temp;

}

**ALGO JS**

let n = arr.length;

for(let I = 0; i<n-1; i++){

let min = I;

for(let j = i+1; j<n; j++){

if(arr[j] < arr[min]){

min = j;

}

}

Let temp = arr[min];

arr[min] = arr[i];

arr[i] = temp;

}

**4.SHELL SORT(**in insertion, we compare adjacent elements; in this we compare distant elements**)**

**Time Complexity : Worst Case : depends on gap sequence**

**Avg Case : depends on gap sequence**

**BEST CASE : O(n log n)**

**SPACE COMPLEXITY : O(1) => a single additional memory space is required i.e.., temp variable**

int n = arr.Length;

int gap = n/2;

int temp,I,j;

While(gap>=1){

for(int I = gap; i<n; i++){

temp = arr[i];

j = I;

while(j>= gap && arr[j-gap] >temp){

arr[j] = arr[j-gap];

j = j-gap;

}

arr[j] = temp;

}

gap = gap/2;

}

**ALGO JS**

Let n = arr.length;

Let gap = Math.floor(n/2);

let temp,I ,j;

while(gap>=1){

for(let i = gap; i<n; i++)

{

temp = arr[i];

j= i;

while(j>= gap && arr[j-gap]>temp){

arr[j] = arr[j-gap];

j= j-gap;

}

arr[j] = temp;

}

gap = gap/2;

}

**5. MERGE SORT**

**Time Complexity : Worst Case : O(n\*Log n)** **; Avg Case : O(n\*Log n)**

**BEST CASE : O(n\*Log n)**

The time complexity of MergeSort is **O(n\*Log n)** in all the 3 cases (worst, average and best) as the mergesort always divides the array into two halves and takes linear time to merge two halves.

**SPACE COMPLEXITY : O(N)**

since the original array has n elements the temporary arrays are of size n respectively

**ALGO C#**

public void mergeSort(int[] arr, int lb, int ub){

if(lb<ub){

int mid = (lb+ub)/2;

mergeSort(arr,lb,mid);

mergeSort(arr,mid+1,ub);

merge(arr,lb,mid,ub);

}

}

Public void merge(int[] arr, int lb, int mid, int ub){

Int I = lb;

Int j = mid+1;

Int k = lb;

Int[] brr = new int[arr.Length];

While(i<= mid && j<= ub){

If(arr[i] < arr[j]){

Brr[k] = arr[i];

I++; k++;

}

Else{

Brr[k] = arr[j];

J++; k++;

}

}

If(i>mid){

While(j<=ub){

Brr[k] = arr[j];

J++;k++;

}

}

Else{

While(i<= mid) {

Brr[k] = arr[i];

I++;k++;

}

}

For(int k = lb; k<=ub; k++){

Arr[k] = brr[k];

}

}

**ALGO JS**

TODO

**6.HEAP SORT**

**Time Complexity : Worst Case : O (n logn)**

**Avg Case : O (n logn)**

**BEST CASE : O (n logn)**

**SPACE COMPLEXITY : O(1)**

public void HeapSort(int[] arr){

int n =arr.Length;

for(int i =n/2; i>=0; i--){

heapify(arr,n,i);

}

For(int I = n-1; i>=0 ; i--){

Int swap = arr[0];

Arr[0] = arr[i];

Arr[i] = swap;

Heapify(arr,n,0);

}

}

Public void heapify(int arr[], int n, int i){

Int largest = I;

Int l = 2\*I;

Int r = (2\*i) +1;

If(l<n && arr[l] > arr[largest]){

Largest = l;

}

If(r<n && arr[r] > arr[largest]){

Largest = r;

}

If(largest != i) {

Int temp = arr[i];

Arr[i] = arr[largest];

Arr[largest] = temp;

Heapify(arr,n,largest);

}

}

**ALGO JS**

function heapSort(arr){

let n = arr.length;

for(let i = Math.floor(n / 2);i>=0; i--){

heapSorting(arr,n,i);

}

for(let i = n-1; i>0 ; i--){

let temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

heapSorting(arr,i,0);

}

}

function heapSorting(arr,n,i){

let largest = i;

let l = 2\*i;

let r = (2\*i)+1;

if(l<n && arr[l]>arr[largest] )

largest = l;

if(r<n && arr[r]>arr[largest] )

largest = r;

if(largest!= i)

{

let temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

heapSorting(arr,n,largest);

}

}

let arr = [15, 6, 5, 4, 7,20, 30];

heapSort(arr);

console.log(arr);

**7.QUICK SORT**

**Time Complexity : Worst Case : O(N^2)**

**Avg Case : O(n\*log(n))**

**BEST CASE : O(n\*log(n))**

**SPACE COMPLEXITY : O(N)**

**I.START**

public void quicksort(int[] arr, int lb, int ub){

if(lb<ub){

int loc = partition(arr,lb,ub);

quicksort(arr,lb,loc-1);

quicksort(arr,loc+1,ub);

}

}

Public int partition(int[] arr, int lb, int ub){

Int pivot = arr[lb];

Int start = lb;

Int end = ub;

While(start<end){

While(arr[start]<pivot && start<end){

Start++;

}

While(arr[end]>pivot){

End--;

}

If(start<end){

Swap(arr[start],arr[end]);

}

}

Swap(arr[end],arr[ lb]);

Return end;

}

**ALGO JS**

function quickSort(arr,lb,ub){

if(lb<ub){

let loc = partition(arr,lb,ub);

quickSort(arr,lb,loc-1);

quickSort(arr,loc+1,ub);

}

return arr;

}

function partition(arr,lb,ub) {

let pivot = arr[lb];

let start = lb, end = ub;

while(start<end){

while(arr[start]<=pivot && start<end){

start++;

}

while(arr[end]>pivot){

end--;

}

if(start<end){

let temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

}

}

let temp2 = arr[lb];

arr[lb] = arr[end];

arr[end] = temp2;

return end;

}

let arr = [68,72,14,23,27,12,8,9];

let sortedArray = quickSort(arr, 0, arr.length - 1);

console.log(sortedArray);

**II. MIDDLE**

public void quicksort(int[] arr, int lb, int ub){

int start = lb;

int end = ub;

int mid = lb +(ub-lb)/2;

int pivot = arr[mid];

while(start< end){

while(arr[start]<pivot){

start++;

}

While(arr[end]> pivot){

End --;

}

If(start<= end){

Swap(arr[start], arr[end]);

Start++;

End--;

}

}

If(lb<end){

Quicksort(arr,lb,end);

}

If(ub>start){

Quicksort(arr,start,ub);

}

}

**ALGO JS**

function quickSortMiddle(arr,lb,ub) {

let mid = Math.floor((lb+ ub)/ 2);

let pivot = arr[mid];

let start = lb;

let end = ub;

while (start <= end) {

while (arr[start] < pivot) {

start++;

}

while (arr[end] > pivot) {

end--;

}

if (start <= end) {

let temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

if (lb < end)

quickSortMiddle(arr, lb, end);

if (ub > start)

quickSortMiddle(arr,start, ub);

}

let arr = [26,14,12,9,23,5,2];

quickSortMiddle(arr, 0, arr.length - 1);

console.log(arr);

**III.END**

public static void quickSort(int[] arr, int lb, int ub)

{

if (lb < ub)

{

int loc = partition(arr, lb, ub);

quickSort(arr, lb, loc - 1);

quickSort(arr, loc + 1, ub);

}

}

public static int partition(int[] arr, int lb, int ub)

{

int start = lb;

int end = ub - 1;

int pivot = ub;

while (start <= end)

{

while (arr[start] < arr[pivot])

{

start++;

}

while (end>= start && arr[end] >= arr[pivot])

{

end--;

}

if (start < end)

{

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

}

}

int temp1 = arr[start];

arr[start] = arr[pivot];

arr[pivot] = temp1;

return start;

}

**IV.RANDOM**

Public void quicksort(int[] arr, int lb, int ub){

If(lb<ub)

{

Int loc= randomPartition(arr,lb,ub);

Quicksort(arr, lb,loc-1);

Quicksort(arr,loc+1,ub);

}

}

Public int randomPartition(int[] arr, int lb, int ub){

Random rand = new Random();

Int random = ran.Next(lb,ub);

Swap(lb,random);//Swap with lb and make it pivot

Return Partition(arr,lb,ub);

}

Public int partition(int[] arr, int lb, int ub){

Int pivot = arr[lb];

Int start = lb;

Int end = ub;

While(start<end){

While(arr[start]<pivot && start<end){

Start++;

}

While(arr[end]>pivot){

End--;

}

If(start<end){

Swap(arr[start],arr[end]);

}

}

Swap(arr[end],arr[ lb]);

Return end;

}

**ALGO JS**

**I.START**

function quickSort(arr,lb,ub){  
 if(lb<ub){  
 let loc = partition(arr,lb,ub);  
 quickSort(arr,lb,loc-1);  
 quickSort(arr,loc+1,ub);  
  
 }  
 return arr;  
}  
function partition(arr,lb,ub) {  
 let pivot = arr[lb];  
 let start = lb, end = ub;  
 while(start<end){  
 while(arr[start]<=pivot && start<end){  
 start++;  
 }  
  
 while(arr[end]>pivot){  
 end--;  
 }  
 if(start<end){  
 let temp = arr[start];  
 arr[start] = arr[end];  
 arr[end] = temp;  
 }  
 }  
 let temp2 = arr[lb];  
 arr[lb] = arr[end];  
 arr[end] = temp2;  
  
 return end;  
}  
let arr = [68,72,14,23,27,12,8,9];  
let sortedArray = quickSort(arr, 0, arr.length - 1);  
console.log(sortedArray);

**II.MIDDLE**

function quickSortMiddle(arr,lb,ub) {  
 let mid = Math.floor((lb+ ub)/ 2);  
 let pivot = arr[mid];  
 let start = lb;  
 let end = ub;  
 while (start <= end) {  
 while (arr[start] < pivot) {  
 start++;  
 }  
 while (arr[end] > pivot) {  
 end--;  
 }  
 if (start <= end) {  
 let temp = arr[start];  
 arr[start] = arr[end];  
 arr[end] = temp;  
 start++;  
 end--;  
 }  
  
 }  
 if (lb < end)  
 quickSortMiddle(arr, lb, end);  
 if (ub > start)  
 quickSortMiddle(arr,start, ub);  
}  
let arr = [26,14,12,9,23,5,2];  
quickSortMiddle(arr, 0, arr.length - 1);  
console.log(arr);

**III. END**

function quickSort(arr,lb,ub){  
 if(lb<ub){  
 let loc = partition(arr,lb,ub);  
 quickSort(arr,lb,loc-1);  
 quickSort(arr,loc+1,ub);  
  
 }  
 return arr;  
}  
function partition(arr,lb,ub){  
 let start = lb;  
 let end = ub - 1;  
 let pivot = ub;  
  
 while (start <= end)  
 {  
 while (arr[start] < arr[pivot])  
 {  
 start++;  
 }  
 while (end>= start && arr[end] >= arr[pivot])  
 {  
 end--;  
 }  
 if (start < end)  
 {  
 let temp = arr[start];  
 arr[start] = arr[end];  
 arr[end] = temp;  
 }  
 }  
 let temp1 = arr[start];  
 arr[start] = arr[pivot];  
 arr[pivot] = temp1;  
  
 return start;  
}  
  
let arr = [26,14,12,9,23,5,2];  
let sortedArray = quickSort(arr, 0, arr.length - 1);  
console.log(sortedArray);

**IV Random**

const quicksort = (arr, low, high) => {

if (low < high) {

const pivot = partitionRandom(arr, low, high);

quicksort(arr, low, pivot - 1);

quicksort(arr, pivot + 1, high);

}

return arr;

}

const partitionRandom = (arr, low, high) => {

//Get random number between the given indexes.

const random = Math.floor(Math.random() \* (high - low + 1) + low);

//Swap with the first element

//swap(arr, random, low);

let temp = arr[random];

arr[random] = arr[low];

arr[low] = temp;

//Use first element as a pivot

return partition(arr, low, high);

}

function partition(arr,lb,ub) {

let pivot = arr[lb];

let start = lb, end = ub;

while(start<end){

while(arr[start]<=pivot && start<end){

start++;

}

while(arr[end]>pivot){

end--;

}

if(start<end){

let temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

}

}

let temp2 = arr[lb];

arr[lb] = arr[end];

arr[end] = temp2;

return end;

}

let arr = [68,72,14,23,27,12,8,9];

let sortedArray = quicksort(arr, 0, arr.length - 1);

console.log(sortedArray);

**8. COUNTING SORT**

The time complexity of counting sort algorithm is **O(n+k)** where n is the number of elements in the array and k is the range of the elements. Counting sort is most efficient if the range of input values is not greater than the number of values to be sorted.

**Time Complexity : Worst Case : O(n+k)**

**Avg Case : O(n+k)**

**BEST CASE : O(n+k)**

**SPACE COMPLEXITY : O(max)**

public void countingSort(int[] arr){

int n = arr.Length;

int max = 0;

for(int I = 0; i<n; i++)

{

If(arr[i]>max)

Max = arr[i];

}

Int k = max;

Int[] count = new int[k+1];

Int[] brr = new int[n];

For(int I = 0; i<n; i++){

Count[i] = 0;

}

For(int I = 0; i<n ; i++){

count[Arr[i]] ++;

}

For(I = 1; i<=k ;i++){

Count[i] += count[i-1];

}

For(int I = n-1; i>=0; i--){

Brr[--count[arr[i]] = arr[i];

}

For(int I = 0; i<n; i++){

Arr[i] = brr[i];

}

}

**ALGO JS**

function countingSort(arr){

let n = arr.length;

let max =0;

for(let i =0;i<n; i++){

if(max < arr[i]){

max = arr[i];

}

}

let count = new Array(max+1);

let brr = new Array(n).fill(0);

for(let i=0;i<max+1;i++){

count[i] = 0;

}

for(let i =0; i<n;i++){

count[arr[i]]++;

}

for(let i = 1; i <= max; i++){

count[i] = count[i] + count[i - 1];

}

for(let i = n-1; i>=0; i--)

{

brr[--count[arr[i]]] = arr[i];

}

for(let i=0;i<n;i++){

arr[i] = brr[i];

}

}

let data = [2,1,0,5,9,3,1,2,4,5,2,1];

countingSort(data);

console.log(data);

**9. RADIX SORT**

**O(nd)**, where n is the size of the array and d is the number of digits in the largest number.

The worst case in radix sort occurs when all elements have the same number of digits except one element which has significantly large number of digits. If the number of digits in the largest element is equal to n, then the runtime becomes O(n2). The worst case running time of Counting sort is **O(n+b)**.

**Time Complexity : Worst Case : O(N+B)**

**Avg Case : O(D\*(n+b))**

**BEST CASE : O(ND)**

**SPACE COMPLEXITY : O(n+b)**

public void radixSort(int[] arr){

int n = arr.Length;

int max = 0;

for(int I = 0; i<n; i++)

{

If(arr[i]>max)

Max = arr[i];

}

For(int pos =1; max/pos>0; pos \*=10){

countSort(arr,n,pos)

}

}

Public void countSort(int[] arr, int n, int pos){

Int[] count = new int[10];

Int[] brr = new int[n];

For(int I = 0; i<10; i++){

Count[i] = 0;

}

For(int I = 0; i<n; i++){

Count[(Arr[i]/pos)%10] ++;

}

For(int I = 1; i<10;i++){

Count[i] += count[i-1];

}

For(int I = n-1; i>=0; i--){

Brr[--count[(arr[i]/pos) %10]] = arr[i];

}

For(int I = 0; i<n; i++){

Arr[i] = brr[i];

}

}

**ALGO JS**

function radixSort(arr) {

let n = arr.length;

let max = 0;

for (let i = 0; i < n; i++) {

if (max < arr[i]) {

max = arr[i];

}

}

for (let pos = 1; max / pos; pos \*= 10) {

countingSort(arr, n, pos);

return;

}

}

function countingSort(arr, n, pos) {

let count = new Array(10);

let brr = new Array(n);

for (let i = 0; i < 10; i++) {

count[i] = 0;

}

for (let i = 0; i < n; i++) {

count[(arr[i]/pos)%10]++;

}

for(let i=1;i<10;i++){

count[i] = count[i]+count[i-1];

}

for(let i =n-1;i>=0;i--){

brr[--count[(arr[i]/pos)%10]] = arr[i];

}

for(let i=0;i<n;i++){

arr[i] = brr[i];

}

}

let data = [2,1,0,5,9,3,1,2,4,5,2,1];

radixSort(data);

console.log(data);

**10.Bucket Sort**

**Time Complexity : Worst Case : O(n + k)**

**Avg Case : O(N^2)**

**BEST CASE : O(n + k)**

**SPACE COMPLEXITY : O(N+K)**

**ALGO C#**

Public int[] bucketSort(int[] arr){

int n = arr.Length;

int max = 0;

for(int I = 0; i<n; i++)

{

If(arr[i]>max)

Max = arr[i];

}

For(int pos =1; max/pos>0; pos \*=10){

bucketSort (arr,n,pos)

}

Return arr;

}

Public void bucketSort(int[] arr, int n, int pos){

List<int> buckets = new List<int>[10];

For(int I = 0; i<10; i++){

Buckets[i] = new List<int>();

}

For(int I = 0; i<n; i++){

Int data = (( Arr[i]/pos) %10);

Buckets[data].Add(arr[i]) ;

Arr[i] = 0;

}

Var arrelement = 0;

Foreach(List<int> data in buckets){

Foreach(int x in data){

Arr[arrelemt] = x;

Arrelement++;

}

}

}

**ALGO JS**

function bucketSort(arr) {

let n = arr.length;

let max = 0;

for (let i = 0; i < n; i++) {

if (max < arr[i]) {

max = arr[i];

}

}

for (let pos = 1; max / pos >0; pos \*= 10) {

bucketSortAlgo(arr, n, pos);

}

}

function bucketSortAlgo(arr, n, pos){

let buckets = new Array(10);

for (let i = 0; i < 10; i++)

buckets[i] = [];

for (let i = 0; i < n; i++)

{

let data =Math.floor (Math.floor(arr[i] / pos) %10);

buckets[data].push(arr[i]);

arr[i] = 0;

}

let arrayvar = 0;

buckets.forEach(x=>

x.forEach(myFunction));

function myFunction(z){

arr[arrayvar] = z;

arrayvar++;

}

}

let data = [15, 1, 321, 10, 802, 2, 123, 90, 109, 11];

bucketSort(data);

console.log(data);

**11.Linear Search**

**Time Complexity : Worst Case : O(N)**

**Avg Case : O(N)**

**BEST CASE : O(1)**

**SPACE COMPLEXITY : O(1)**

**ALGO C#**

int n = arr.Length;

int found = 0;

for(int i = 0; i<n ; i++){

If(arr[i] == data){

found ==1;

Console.WriteLine(“Data is found”);

}

}

If(found == 0){

Console.WriteLine(“Data is not found”);

break;

}

**ALGO JS**

function linearSearch(arr,data){

let n = arr.length;

let found = 0;

for(let I = 0; i<n; i++){

If(arr[i] == data){

found = 1;

console.log(“data is found”);

}

}

If(found == 0){

console.log(“data is not found”);

break;

}}

**12. BINARY SEARCH**

**Time Complexity : Worst Case : O(log n)**

**Avg Case : O(log n)**

**BEST CASE :** O(1)

**SPACE COMPLEXITY : O(1)**

**ALGO C#**

Public void BinarySearch(int[] arr, int value){

int l = 0;

int r = arr.Length-1;

int mid = (l+r)/2;

while(l<=r){

if(value == arr[mid]){

Console.WriteLine(“Value is found at index” +mid);

}

if(arr[mid] < value)

L = mid+1;

Else

R = mid-1;

}

Console.WriteLine(“value is not found”);

}

**ALGO JS**

function BinarySearch(arr,value){

let l = 0;

let r = arr.length-1;

let mid = Math.floor((l+r)/2);

while(l<=r){

if(arr[mid] == value)

console.log(“value is found at index”+mid);

if (arr[mid] <value)

l = mid+1;

else

r = mid-1;

}

Console.log(“value is not found”);

}

**13. Single Linked List**

Public class Node{

Public int value;

Public Node next;

Public Node(int val){

Value = val;

next = null;

}

}

Public class SingleLinkedList{

Private Node head;

Public SingleLinkedList(){

Head = null;

}

**INSERT AT FRONT**

Public void insertFront(int data){

Node newNode = new Node (data);

If(head == null){

Head = newNode;

Return;

}

newNode.next = head;

head = newNode;

}

**INSERT AT END**

Public void insertEnd(int data){

Node newNode = new Node(data);

If(head == null){

Head = newNode;

Return;

}

Node p = head;

While(p.next!= null){

P = p.next;

}

p.next = newNode;

return;

}

**Push at Index;**

Public void pushAtPosition(int data, int position){

Node newNode = new Node(data);

If(position<1)

Console.WriteLine(“Index should be greater than 1”);

Else if(position == 1){

newNode.next = head;

head = newNode;

}

Else{

Node p = head;

For(int I = 1; i<position-1; i++){

If(p != null){

P = p.next;

}

}

If(p != null){

newNode.next =p.next;

p.next = newNode;

}

Else{

Console.WriteLine(“Previous Node is Null”);

}

}

}

**Removes element at position**

Public void removeNodeatPosition(int pos){

If(head == null)

Return;

If(pos<1)

Console.WriteLine(“Pos should be >1”);

Else if(pos ==1){

Node temp = head;

Head = head.next;

Temp = null;

}

Else{

Node p = head;

For(int I = 1; i<pos-1; i++){

If(p!= null){

P = p.next;

}

}

If(p != null && p.next != null){

Node nodeToDel = p.next;

p.next = p.next.next;

nodeToDel = null;

}

Else{

Console.WriteLine(“Previous Node is Null”);

}

}

}

**Remove Node AT INDEX**

Public void removeNodeAtIndex(int index){

If(head == null)

Return;

Node p = head;

If(index == 0){

Head = p.next;

}

For(int I = 0; i<index-1; i++){

If(p!= null)

P = p.next;

}

If(p!= null && p.next != null)

{

Node nodeToDel = p.next;

p.next = p.next.next;

nodeToDel = null;

}

Else

Console.WriteLine(“Previous Node is Null”);

}

**Remove last NODE**

Public Node removeLastNode(Node head){

If(head == null || head.next == null )

Return;

Node p = head;

While(p.next.next != null ){

P = p.next;

}

p.next = null;

return head;

}

**REMOVE FIRST NODE**

Public Node removeFirstNode(Node head){

If(head == null )

Return null;

Head = head.next;

Return head;

}

**REVERSE**

Public Node reverse(Node head){

Node prev = null;

While(head != null){

Node nextNode = head.next;

Head.next =prev;

Prev = head;

Head = nextNode;

}

Return prev;

}

}

**ALGO JS**

**Index at front**

class Node {

constructor(data) {

this.val = data

this.next = null

}

}

class SingleLinkedList {

constructor(head = null) {

this.head = head

}

**INSERT AT FRONT**

insertFront(data){

let newNode = new Node (data);

if(head == null){

head = newNode;

return;

}

newNode.next = head;

head = newNode;

return head;

}

**INSERT AT END**

insertEnd( data){

let newNode = new Node(data);

if(head == null){

head = newNode;

return;

}

let p = head;

while(p.next!= null){

p = p.next;

}

p.next = newNode;

return head;

}

**PUSH AT INDEX**

pushAtPosition( data,position) {

let newNode = new Node(data);

if(position<1)

console.log("Index should be greater than 1");

else if(position == 1){

newNode.next = head;

head = newNode;

}

else{

let p = head;

for(let i = 1; i<position-1; i++){

if(p != null){

P = p.next;

}

}

if(p != null){

newNode.next = p.next;

p.next = newNode;

}

else{

console.log("Previous Node is Null");

}

}

return head;

}

**Removes element at position**

removeNodeatPosition( pos){

if(head == null)

return;

if(pos<1)

console.log("Pos should be >1");

else if(pos ==1){

let temp = head;

head = head.next;

temp = null;

}

else{

let p = head;

for(let i = 1; i<pos-1; i++){

if(p!= null){

P = p.next;

}

}

if(p != null && p.next != null){

let nodeToDel = p.next;

p.next = p.next.next;

nodeToDel = null;

}

else{

console.log("Previous Node is Null");

}

}

return head;

}

**REMOVE AT INDEX**

removeNodeAtIndex( index){

if(head == null)

return;

let p = head;

if(index == 0){

head = p.next;

}

for(let i = 0; i<index-1; i++){

if(p!= null)

p = p.next;

}

if(p!= null && p.next != null)

{

let nodeToDel = p.next;

p.next = p.next.next;

nodeToDel = null;

}

else

console.log("Previous Node is Null");

return head;

}

**REMOVE LAST NODE**

removeLastNode(){

if(head == null || head.next == null )

return;

let p = head;

while(p.next.next != null ){

p = p.next;

}

p.next = null;

return head;

}

**REMOVE FIRST NODE**

removeFirstNode(){

if(head == null )

return null;

head = head.next;

return head;

}

**REVERSE**

reverse(head){

let prev = null;

while(head != null){

let nextNode = head.next;

head.next =prev;

prev = head;

head = nextNode;

}

return prev;

}

}

**14. Doubly linked list**

Public class Node{

Public int value;

Public Node next;

Public Node prev;

Public Node(int val){

Value = val;

Next = null;

Prev = null;

}

}

Public class DoublyLinkedList{

Private Node head;

Public DoublyLinkedList(){

Head = null;

}

**INSERT AT FRONT**

Public void insertAtFront(int data){

Node newNode = new Node(data);

If(head == null){

Head = newNode;

Return;

}

Head.prev = newNode;

newNode.next = head;

head = newNode;

}

**INSERT AT END**

Public void insertAtEnd(int data){

If(head == null){

Head = newNode;

Return;

}

Node p = head;

While(p.next != null){

P = p.next;

}

p.next = newNode;

newNode.prev = p;

}

**PUSH AT POSITION**

Public void pushAtIndex(int data, int pos){

Node newNode = new Node(data);

If(head == null)

Return null;

If(pos <1)

Console.WriteLine(“Position should be greater than 1”);

Else if(pos ==1){

head.prev = newNode;

newNode.next = head;

head = newNode;

}

Else{

Node p = head;

For(int I = 1; i<pos-1; i++){

If(p!= null)

P = p.next;

}

If(p!= null && p.next != null ){

newNode.next = p.next;

newNode.prev = p;

p.next = newNode;

newNode.next.prev = newNode;

}

Else{

Console.WriteLine(“Previous Node is null”);

}

}

}

**Remove Node at Position**

Public void removeAtPosition(int data, int position){

If(head == null)

Return null;

If(pos<1)

Console.WriteLine(“pos >=1”);

Else if(pos ==1){

Node nodeToDel = head;

Head = head.next;

nodeToDel = null;

if(head != null)

head.prev = null;

}

Else{

Node p = head;

For(int I = 1; i<pos-1; i++){

If(p!= null)

P = p.next;

}

If(p!= null && p.next != null){

Node toDel = p.next;

p.next = p.next.next;

if(p.next.next != null)

p.next.next.prev = p.next;

toDel = null;

}

Else

Console.WriteLine(“previous Node is null’);

}

}

**Remove Node At Index**

public void RemoveAtIndex (int index)

{

if(head == null)

return;

Node p = head;

if(index == 0)

{

head = p.next;

head.prev = null;

}

else

{

for(int i = 0; i<index-1; i++)

{

if (p != null)

p = p.next;

}

if(p != null && p.next != null)

{

Node toDel = p.next;

p.next = p.next.next;

toDel = null;

if(p.next != null)

p.next.prev = p;

}

else

{

Console.WriteLine("previous Node is null");

}

}

}

**REMOVE FIRST NODE**

public void RemoveFirst()

{

if (head == null)

return;

if(head.next == null)

{

head = null;

return;

}

else

{

head = head.next;

head.prev = null;

}

}

**REMOVE LAST NODE**

public void RemoveLast()

{

if (head == null)

return;

if (head.next == null)

head = null;

else

{

Node p = head;

while(p.next.next != null)

{

p = p.next;

}

Node nodeToDel = p.next;

p.next = null;

nodeToDel = null;

}

}

**REVERSE**

public void reverse()

{

Node temp = null;

while (head != null)

{

temp = head.prev;

head.prev = head.next;

head.next = temp;

head = head.prev;

}

if (temp != null)

{

head = temp.prev;

}

}

}

**ALGO JS**

**INSERT AT FRONT**

insertAtFront(data){

let newNode = new Node(data);

if(head == null){

head = newNode;

return;

}

head.prev = newNode;

newNode.next = head;

head = newNode;

return head;

}

**Insert At End**

insertAtEnd(data){

let newNode = new Node(data);

if(head == null){

head = newNode;

return;

}

let p = head;

while(p.next != null){

p = p.next;

}

p.next = newNode;

newNode.prev = p;

return head;

}

**Push At Position**

pushAtPos(data, pos){

let newNode = new Node(data);

if(head == null)

return null;

if(pos <1)

console.log("Position should be greater than 1");

else if(pos ==1){

head.prev = newNode;

newNode.next = head;

head = newNode;

}

else{

let p = head;

for(let i = 1; i<pos-1; i++){

if(p!= null)

p = p.next;

}

if(p!= null && p.next != null ){

newNode.next = p.next;

newNode.prev = p;

p.next = newNode;

newNode.next.prev = newNode;

}

else{

console.log("Previous Node is null");

}

}

return head;

}

**Remove At Position**

removeAtPosition( data, position){

if(head == null)

return null;

if(position<1)

console.log("pos >=1");

else if(position ==1){

let nodeToDel = head;

head = head.next;

nodeToDel = null;

if(head != null)

head.prev = null;

}

else{

let p = head;

for(let i = 1; i<position-1; i++){

if(p!= null)

p = p.next;

}

if(p!= null && p.next != null){

let toDel = p.next;

p.next = p.next.next;

if(p.next.next != null)

p.next.next.prev = p.next;

toDel = null;

}

else

console.log("previous Node is null");

}

return head;

}

**Remove At Index**

RemoveAtIndex ( index)

{

if(head == null)

return;

let p = head;

if(index == 0)

{

head = p.next;

head.prev = null;

}

else

{

for(let i = 0; i<index-1; i++)

{

if (p != null)

p = p.next;

}

if(p != null && p.next != null)

{

let toDel = p.next;

p.next = p.next.next;

toDel = null;

if(p.next != null)

p.next.prev = p;

}

else

{

console.log("previous Node is null");

}

}

return head;

}

**Remove First**

RemoveFirst()

{

if (head == null)

return;

if(head.next == null)

{

head = null;

return;

}

else

{

head = head.next;

head.prev = null;

}

return head;

}

**Remove Last**

RemoveLast()

{

if (head == null)

return;

if (head.next == null)

head = null;

else

{

let p = head;

while(p.next.next != null)

{

p = p.next;

}

let nodeToDel = p.next;

p.next = null;

nodeToDel = null;

}

return head;

}

**REVERSE**

function reverse() {

var temp = null;

while (head != null) {

temp = head.prev;

head.prev = head.next;

head.next = temp;

head = head.prev;

}

if (temp != null) {

head = temp.prev;

}

}

**15. Circular Linked List**

Public class Node {

Public Node next;

Public int value;

Public Node(int val){

Value = val;

Next = null;

}

}

Public class CLL {

Private Node head;

Public CLL (){

Head = null;

}

**ADD TO EMPTY LIST**

Public void addToEmptyList(int data){

Node newNode = new Node(data);

head = newNode;

head.next = head;

}

**ADD TO BEGINNING LIST**

public void addToBegin(int data)

{

Node newNode = new Node(data);

if (head == null)

{

head = newNode;

head.next = head;

return;

}

Node temp = head;

while(temp.next != head)

{

temp = temp.next;

}

temp.next = newNode;

newNode.next = head;

head = newNode;

}

**ADD TO END OF LIST**

public void addToEnd(int data)

{

Node newNode = new Node(data);

if (head == null)

{

head = newNode;

head.next = head;

return;

}

else

{

Node temp = head;

while (temp.next != head)

temp = temp.next;

temp.next = newNode;

newNode.next = head;

}

}

**DELETE FIRST**

public void deleteFirst()

{

if (head == null)

return;

if (head.next == head)

{

head = null;

}

else

{

Node temp = head;

Node firstNode = head;

while (temp.next != head)

{

temp = temp.next;

}

head = head.next;

temp.next = head;

firstNode = null;

}

}

**DELETE LAST**

public void deleteLast()

{

if (head == null)

return;

if (head.next == head)

{

head = null;

}

else

{

Node temp = head;

while(temp.next.next != head)

{

temp = temp.next;

}

Node lastNode = temp.next;

temp.next = this.head;

lastNode = null;

//temp.next = head;

}

}

**REVERSE**

**//TO DO**

}

**ALGO JS**

class Node {

constructor(data) {

this.val = data

this.next = null

}

}

class circularLinkedList {

constructor(head = null) {

this.head = head

}

**Add to empty list**

addToEmptyList(data){

let newNode = new Node(data);

head = newNode;

head.next = head;

return head;

}

**Add to the beginning of list**

addToBegin( data)

{

let newNode = new Node(data);

if (head == null)

{

head = newNode;

head.next = head;

return;

}

let temp = head;

while(temp.next != head)

{

temp = temp.next;

}

temp.next = newNode;

newNode.next = head;

head = newNode;

return head;

}

**ADD TO END OF LIST**

addToEnd(data)

{

let newNode = new Node(data);

if (head == null)

{

head = newNode;

head.next = head;

return;

}

else

{

let temp = head;

while (temp.next != head)

temp = temp.next;

temp.next = newNode;

newNode.next = head;

}

return head;

}

**DELETE FIRST**

deleteFirst()

{

if (head == null)

return;

if (head.next == head)

{

head = null;

}

else

{

let temp = head;

let firstNode = head;

while (temp.next != head)

{

temp = temp.next;

}

head = head.next;

temp.next = head;

firstNode = null;

}

return head;

}

**DELETE LAST**

deleteLast()

{

if (head == null)

return;

if (head.next == head)

{

head = null;

}

else

{

let temp = head;

while(temp.next.next != head)

{

temp = temp.next;

}

let lastNode = temp.next;

temp.next = head;

lastNode = null;

}

return head;

}

**REVERSE**

}

let head = null;

const p = new circularLinkedList();

console.log( p.addToEmptyList(3));

console.log( p.addToBegin(2));

console.log( p.addToEnd(4));

console.log( p.deleteFirst());

console.log( p.deleteLast());

**16. CIRCULAR DOUBLY LINKED LIST**

public class Node

{

public int value;

public Node next;

public Node prev;

public Node(int x)

{

value = x; next = null;prev = null;

}

}

public class CircularDoublyLinkedList

{

private Node head;

public CircularDoublyLinkedList()

{

head = null;

}

**ADD TO EMPTY LIST**

Public void addToEmptyList(int data){

Node newNode = new Node(data);

head = newNode;

head.next = head;

head.prev = head;

}

**ADD TO THE BEGINNING**

public void createStart(int data)

{

Node newNode = new Node(data);

Node temp = head;

if (head == null)

{

head = newNode;

newNode.prev = head;

newNode.next = head;

}

else

{

while(temp.next!= head)

{

temp = temp.next;

}

temp.next = newNode;

newNode.next = head;

newNode.prev = temp;

head.prev = newNode;

head = newNode;

}

}

**ADD TO THE End**

public void createEnd(int data)

{

Node newNode = new Node(data);

Node temp = head;

if(head == null)

{

head = newNode;

newNode.prev = head;

newNode.next = head;

}

else

{

while(temp.next!= head)

{

temp = temp.next;

}

temp.next = newNode;

newNode.next = head;

newNode.prev = temp;

head.prev = newNode;

}

}

**DELETE FIRST**

public void DeleteFirst()

{

Node firstNode = head;

Node p = head;

if(head== null) {

return;

}

if (head.next == head)

{

head = null;

}

else

{

while(p.next!= head)

{

p = p.next;

}

Node temp = head.next;

head = temp;

p.next = head;

head.prev = p;

firstNode = null;

}

}

**DELET LAST NODE**

public void deleteEnd()

{

Node firstNode = head;

Node p = head;

if (head == null)

return;

if (head.next == head)

{

head = null;

}

else

{

while(p.next.next != head)

{

p = p.next;

}

Node lastNode = p;

p.next = head;

head.prev = p;

lastNode = null;

}

}

**REVERSE**

**//TODO**

}

**ALGO JS**

class Node {

constructor(data) {

this.val = data

this.next = null

this.prev = null;

}

}

class circulardoublyLinkedList {

constructor(head = null) {

this.head = head

}

**ADD TO EMPTY LIST**

addToEmptyList( data){

let newNode = new Node(data);

head = newNode;

head.next = head;

head.prev = head;

return head;

}

**ADD TO THE BEGINNING**

createStart( data)

{

let newNode = new Node(data);

let temp = head;

if (head == null)

{

head = newNode;

newNode.prev = head;

newNode.next = head;

}

else

{

while(temp.next!= head)

{

temp = temp.next;

}

temp.next = newNode;

newNode.next = head;

newNode.prev = temp;

head.prev = newNode;

head = newNode;

}

return head;

}

**ADD TO THE End**

createEnd(data)

{

let newNode = new Node(data);

let temp = head;

if(head == null)

{

head = newNode;

newNode.prev = head;

newNode.next = head;

}

else

{

while(temp.next!= head)

{

temp = temp.next;

}

temp.next = newNode;

newNode.next = head;

newNode.prev = temp;

head.prev = newNode;

}

return head;

}

**DELETE FIRST**

DeleteFirst()

{

let firstNode = head;

let p = head;

if(head== null) {

return;

}

if (head.next == head)

{

head = null;

}

else

{

while(p.next!= head)

{

p = p.next;

}

let temp = head.next;

head = temp;

p.next = head;

head.prev = p;

firstNode = null;

}

return head

}

**DELETE LAST NODE**

deleteEnd()

{

let firstNode = head;

let p = head;

if (head == null)

return;

if (head.next == head)

{

head = null;

}

else

{

while(p.next.next != head)

{

p = p.next;

}

let lastNode = p;

p.next = head;

head.prev = p;

lastNode = null;

}

return head;

}

}

let head = null;

const p = new circulardoublyLinkedList();

console.log( p.addToEmptyList(3));

console.log( p.createStart(2));

console.log( p.createEnd(4));

console.log( p.DeleteFirst());

console.log( p.deleteEnd());

**17.STACK**

1. **STACK USING ARRAY**

**Time Complexity :**

**PUSH : Append at index n : O(1)**

**Prepend at index 0 : O(N)**

**POP : Delete index n-1 : O(1)**

**Delete index 0 : O(N)**

**PEEK : BOTH AT O ,N : O(1)**

**SPACE COMPLEXITY : O(1)**

public class CreateStack {

private int MAX = 100;

private int top;

private int[] stack;

public CreateStack()

{

stack = new int[MAX];

top = -1;

}

**EMPTY STACK OR NOT**

public void isEmpty()

{

if (top == -1)

{

Console.WriteLine("Stack is empty.");

}

else

{

Console.WriteLine("Stack is not empty.");

}

}

**PUSH ELEMENT INTO STACK**

public void push(int x)

{

if (top == (MAX - 1))

{

Console.WriteLine("Stack size limit reached.");

}

else

{

stack[++top] = x;

Console.WriteLine(x + " is added into the stack.");

}

}

**POP ELEMENT FROM STACK**

public void pop()

{

if (top < 0)

{

Console.WriteLine("Stack is empty.");

}

else

{

int x = stack[top--];

Console.WriteLine(x + " is deleted from the stack.");

}

}

**PEEK ELEMENT FROM STACK**

public int peek()

{

if (top < 0)

{

Console.WriteLine("Stack is empty.");

return 0;

}

else

{

return stack[top];

}

}

**SIZE OF STACK**

public int size()

{

return top + 1;

}

public void display()

{

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

{

Console.WriteLine(stack[i]);

}

}

static void Main(string[] args)

{

CreateStack MyStack = new CreateStack();

MyStack.push(10);

MyStack.push(20);

MyStack.push(30);

MyStack.push(40);

MyStack.display();

MyStack.pop();

MyStack.isEmpty();

int result = MyStack.peek();

Console.WriteLine($"The topElement is : {result}");

}

}

**ALGO JS**

class stack{

constructor() {

this.MAX = 10

this.top = -1;

this.stack = new Array(this.MAX);

}

isEmpty()

{

if (this.top == -1)

{

console.log("Stack is empty.");

}

else

{

console.log("Stack is not empty.");

}

}

size()

{

return this.top + 1;

}

push( x)

{

if (this.top == (this.MAX - 1))

{

console.log("Stack size limit reached.");

}

else

{

this.stack[++this.top] = x;

//console.log(x + " is added into the stack.");

}

}

pop()

{

if (this.top < 0)

{

console.log("Stack is empty.");

}

else

{

let x = this.stack[this.top--];

console.log(x + " is deleted from the stack.");

}

}

peek()

{

if (this.top < 0)

{

console.log("Stack is empty.");

return 0;

}

else

{

//Console.WriteLine("Stack not empty.");

return this.stack[this.top];

}

}

print(){

for (let i = this.top; i >= 0; i--)

{

console.log(this.stack[i]);

}

}

}

const Stack = new stack();

Stack.push(10);

Stack.push(20);

Stack.push(30);

Stack.push(40);

console.log(Stack.print());

Stack.pop();

console.log(Stack.print());

console.log(Stack.peek());

1. **STACK USING LINKED LIST**

**TIME COMPLEXITY**

**PUSH : Append at tail : O(1)**

**Prepend at head: O(1)**

**POP : Delete tail Node: O(N)**

**Delete head Node : O(1)**

**PEEK : BOTH AT head, tail : O(1)**

**SPACE COMPLEXITY : O(1)**

public class Node

{

public int value;

public Node next;

public Node(int x)

{

value = x; next = null;

}

}

public class StackLinkedList

{

Node top;

public StackLinkedList()

{

top = null;

}

public void push(int data)

{

Node newNode = new Node(data);

newNode.next = top;

top = newNode;

}

public void pop()

{

Node temp = top;

if (top == null)

Console.WriteLine("stack is empty, Nothing to delete");

else

{

Console.WriteLine("The deleted element is " + top.value);

top = top.next;

temp = null;

}

}

public void peek()

{

if (top == null)

Console.WriteLine("stack is empty");

else

{

Console.WriteLine("The top element is " + top.value);

}

}

public void display()

{

if (top == null)

{

Console.Write("\nStack is empty");

return;

}

else

{

Node temp = top;

while (temp != null)

{

Console.WriteLine(temp.value);

temp = temp.next;

}

}

}

}

static void Main(string[] args)

{

StackLinkedList data = new StackLinkedList();

data.push(10);

data.push(20);

data.push(30);

data.push(40);

data.display();

data.pop();

data.display();

data.peek();

}

**ALGO JS**

class Node{

constructor(value){

this.value = value;

this.next = null;

}

}

class Stack {

constructor() {

this.top = null;

}

// Insert the element at the top of the stack.

push(value) {

const newNode = new Node(value);

newNode.next = this.top;

this.top = newNode;

}

pop(){

let temp = this.top;

if(this.top == null)

console.log("The stack is empty");

else{

console.log("The deleted value is" +this.top.value);

this.top = this.top.next;

temp = null;

}

}

// Return the top element.

peek() {

if(this.top == null)

console.log("stack is empty");

else

console.log("The top element is :" +this.top.value);

}

printList() {

let list = [];

let currentNode = this.top;

while (currentNode !== null) {

list.push(currentNode.value);

currentNode = currentNode.next;

}

console.log(list + " ");

}

}

const myStack = new Stack();

myStack.push(10);

myStack.push(20);

myStack.push(30);

myStack.push(40);

myStack.printList();

myStack.peek();

myStack.pop();

myStack.pop();

myStack.peek();

**III. STACK USING QUEUE**

**i.MAKING PUSH COSTLY**

**Time Complexity : Push : O(N), POP : O(1), PEEK: O(1)**

**SPACE COMPLEXITY : O(N)**

Public class stackUsingQ{

Queue<int> queue1 = new Queue<int>();

**PUSH VALUES IN QUEUE**

Public void push(int x){

Var queue2 = new Queue<int>();

Queue2.Enqueue(x);

While(queue1.Count>0){

queue2.Enqueue(queue1.Dequeue());

}

queue1 = queue2;

}

**POP**

Public int pop(){

If(queue1.Count == 0)

Console.WriteLine(“Queue iss empty”);

Else

Return queue1.Dequeue();

}

**Peek**

Public int peek(){

If(queue1.Count == 0)

Console.WriteLine(“Queue iss empty”);

Else

Return queue1.Peek ();

}

**IS EMPTY OR NOT**

public bool Empty()

{

if (queue1.Count == 0)

return true;

return false;

}

}

**ALGO JS**

class Queue {

constructor() {

this.storage = {}

this.last = 0

this.first = 0

}

peek() {

return this.storage[this.first]

}

size() {

return this.last -this.first

}

isEmpty() {

return this.last === this.first

}

dequeue() {

let first = this.storage[this.first]

delete this.storage[this.first]

this.first++

return first

}

enqueue(val) {

this.storage[this.last] = val

this.last++

}

}

1. **MAKING POP COSTLY**

**Time Complexity : Push : O(1), POP : O(N), PEEK: O(1)**

**SPACE COMPLEXITY : O(N)**

public class MyStack

{

Queue<int> queue1 = new Queue<int>();

public void Push(int x)

{

queue1.Enqueue(x);

}

public void Pop()

{

var queue2 = new Queue<int>();

if (queue1.Count == 0)

return;

while (queue1.Count != 1)

{

queue2.Enqueue(queue1.Dequeue());

}

Console.WriteLine( "The element deleted is " +queue1.Dequeue());

queue1 = queue2;

}

public int TopPeek()

{

if (queue1.Count == 0)

return -1;

return queue1.Peek();

}

public bool Empty()

{

if (queue1.Count == 0)

return true;

return false;

}

static void Main(string[] args)

{

MyStack p = new MyStack();

p.Push(5);

p.Push(6);

p.Pop();

p.Push(7);

p.Push(8);

p.Pop();

p.Pop();

Console.WriteLine(p.Empty());

int data = p.TopPeek();

Console.WriteLine(data);

}

}

**ALGO JS**

class Queue {

constructor() {

this.storage = {}

this.last = 0

this.first = 0

}

peek() {

return this.storage[this.first]

}

size() {

return this.last -this.first

}

isEmpty() {

return this.last === this.first

}

dequeue() {

let first = this.storage[this.first]

delete this.storage[this.first]

this.first++

return first

}

enqueue(val) {

this.storage[this.last] = val

this.last++

}

}

var MyStack = function() {

this.queue1 = new Queue();

this.queue2 = new Queue();

};

MyStack.prototype.push = function (x) {

this.queue1.enqueue(x);

};

MyStack.prototype.pop = function () {

if (this.queue1.size() == 0)

return;

while (this.queue1.size() != 1)

{

this.queue2.enqueue(this.queue1.dequeue());

}

console.log( "The element deleted is " +this.queue1.dequeue());

let temp = this.queue1

this.queue1 = this.queue2

this.queue2 = temp

};

MyStack.prototype.top = function() {

if (this.queue1.size() == 0)

return -1;

while (this.queue1.size() > 1)

this.queue2.enqueue(this.queue1.dequeue());

let top = this.queue1.dequeue();

this.queue2.enqueue(top);

let temp = this.queue1

this.queue1 = this.queue2

this.queue2 = temp

return top;

};

MyStack.prototype.empty = function() {

if (this.queue1.size() == 0)

return true;

return false;

};

const p = new MyStack();

p.push(5);

p.push(6);

console.log(p.pop());

console.log(p.top());

console.log(p.empty());

**18.Queue**

**i. Queue using Arrays**

**Time Complexity :**

**Enqueue:** O(N) time (worst case), O(1) time (best case), O(1) time (average case)

**Dequeue: O(1)**

**PEEK : O(1)**

**SPACE COMPLEXITY : O(1)**

Public class QueueData{

Int Max = 100;

Int[] queue;

Int front,rear;

Public QueueData(){

Front = -1;rear = -1;

Queue = new int[Max];

}

**QUEUE IS EMPTY OR NOT**

Public void IsEmpty(){

If(front == -1 and rear == -1)

Console.WriteLine(“Queue is empty”);

Else

Console.WriteLine(“Queue is not empty”);

}

**ENQUEUE**

Public void enqueue(int x){

If(rear == Max -1)

Console.WriteLine(“Queue is full”);

Else if(rear == -1 && front == -1){

Rear = front = 0;

Queue[rear] = x;

}

Else{

Rear++;

Queue[rear] = x;

}

}

**DEQUEUE**

Public void dequeue(int x){

If(rear == -1 && front == -1){

Console.WriteLine(“Queue is Empty”);

}

Else if(front == rear){

Front= rear = -1;

}

Else{

Console.WriteLine(queue[front]);

Front++;

}

}

**PEEK**

public int frontElement()

{

if (rear == front)

{

Console.WriteLine("Queue is empty.");

return 0;

}

else

{

return queue[front];

}

}

**DISPLAY**

public void display()

{

if (rear == -1 && front == -1)

{

Console.WriteLine("queue is empty");

}

else

{

for(int i = front; i<=rear; i++)

{

Console.WriteLine(queue[i]);

}

}

}

}

**ALGO JS**

class queue{

constructor(){

this.data= [];

this.Max = 6;

this.front = -1;

this.rear = -1;

this.data = new Array(this.Max);

}

enQueue(x){

if (this.rear == (this.Max - 1))

{

console.log("Queue size limit reached.");

}

else if(this.front==-1 && this.rear == -1)

{

this.front = this.rear = 0;

this.data[this.rear] = x;

console.log(x + " is added into the queue.");

}

else

{

this.data[++this.rear] = x;

console.log(x + " is added into the queue.");

}

}

peek(){

if (this.rear == this.front)

{

console.log("Queue is empty.");

}

else

{

return this.data[this.front];

}

}

deQueue(){

if (this.rear == -1 && this.front == -1)

{

console.log("Queue is empty.");

}

else if(this.rear == this.front)

{

this.front = this.rear = -1;

}

else

{

let x = this.data[this.front];

console.log(x + " is deleted from the queue.");

this.front++;

}

}

isEmpty(){

if (this.rear == -1 && this.front == -1)

{

console.log("Queue is empty.");

}

else

{

console.log("Queue is not empty.");

}

}

size()

{

return (this.rear - this.front)+1;

}

printList(){

if(this.front== -1 && this.rear == -1)

console.log("queue is empty");

else{

for(let i =0; i<this.rear+1; i++){

console.log(this.data[i]);

}

}

}

}

**ii.Queue using linked list**

**Time Complexity :**

**Enqueue:** O(N) time (worst case), O(1) time (best case), O(N) time (average case)

**Dequeue:** O(N) time (worst case), O(1) time (best case), O(N) time (average case)

**PEEK : O(1)**

**SPACE COMPLEXITY : O(1)**

public class Node

{

Public int data;

Public Node next;

Public Node(int x){

Data = x;

Next = null;

}

}

Public class LinkedListQueue{

Node front;

Node rear;

Public LinkedListQueue(){

Front = null;

Rear = null;

}

**Queue is Empty or not**

Public void IsEmpty(){

If(front == null && rear == null)

Console.WriteLine(“Queue is Empty”);

Else

Console.WriteLine(“Queue is Not Empty”);

}

**ENQUEUE**

Public void enQueue(int value){

Node newNode = new Node(value);

If(front == null && rear == null){

Front = rear = newNode;

}

Else{

Rear.next = newNode;

Rear = newNode;

}

}

**DEQUEUE**

Public void deQueue(){

If(front == null && rear == null)

Console.WriteLine(“Queue is empty”);

Else {

Node temp = front;

Front = front.next;

Console.WriteLine("Item deleted is {0}", temp.data);

Temp = null;

}

}

Public void peek(){

If(front == null && rear == null){

Console.WriteLine(“Queue is empty”);

}

Else{

Console.Write("The top element is " + this.front.data);

}

}

**DISPLAY THE DATA**

public void display()

{

if (this.front == null && this.rear == null)

{

Console.WriteLine("Queue is empty");

return;

}

else

{

Node temp = this.front;

while (temp != null)

{

Console.WriteLine("\n" + temp.data);

temp = temp.next;

}

}

}

}

**ALGO JS**

class Node{

constructor(value){

this.data = value;

this.next = null;

}

}

class queueLinkedList{

constructor(){

this.front =this.rear= null;

}

enQueue(value){

const newNode = new Node(value);

if(this.rear == null && this.front == null)

this.front= this.rear = newNode;

else{

this.rear.next = newNode;

this.rear = newNode;

}

}

deQueue(){

if(this.rear == null && this.front == null){

console.log("queue is empty");

}

const temp = this.front;

this.front = this.front.next;

console.log("the deleted item is " +temp.data);

this.temp = null;

}

peek(){

if(this.rear == null && this.front == null){

console.log("queue is empty");

}

else

console.log("The top data is " +this.front.data);

}

}

**III.QUEUE USING STACK**

Class createStack{

private int[] stack1, stack2;

private int Max;

private int top1;

private int top2;

int count;

public createStack()

{

Max = 5;

stack1 = new int[Max];

stack2 = new int[Max];

top1 = -1;

top2 = -1;

count = 0;

}

public void enqueue(int data)

{

push1(data);

count++;

}

public void push1(int x)

{

if (top1 == Max - 1)

{

Console.WriteLine("Stack is full");

}

else

{

stack1[++top1] = x;

}

}

public void dequeue()

{

if(top1== -1 && top2 ==-1)

{

Console.WriteLine("Stack is empty");

}

else

{

for(int i =0; i<count; i++)

{

int x = pop1();

push2(x);

}

int y = pop2();

Console.WriteLine("The dequeued element is:" + y);

count--;

for(int i=0;i<count; i++)

{

int z = pop2();

push1(z);

}

}

}

public int pop1()

{

if (top1 == -1)

{

Console.WriteLine("Stack is empty");

return 0;

}

else

{

int a = stack1[top1];

top1--;

return a;

}

}

public int pop2()

{

if (top2 == -1)

{

Console.WriteLine("Stack is empty");

return 0;

}

else

{

int pop = stack2[top2];

top2--;

return pop;

}

}

public void push2(int x)

{

if (top2 == Max - 1)

{

Console.WriteLine("Stack is full");

}

else

{

stack2[++top2] = x;

}

}

public void peek()

{

if (top1 == -1 && top2 == -1)

{

Console.WriteLine("Stack is empty");

}

else

{

for (int i = 0; i < count; i++)

{

int x = pop1();

push2(x);

}

int y = stack2[top2];

Console.WriteLine("The top element is :" + y);

for (int i = 0; i < count; i++)

{

int z = pop2();

push1(z);

}

}

}

public void display()

{

for (int i = 0; i <= top1; i++)

{

Console.WriteLine(stack1[i] + " ");

}

}

}

**ALGO JS**

class queue {

constructor() {

this.stack1 = [];

this.stack2 = [];

this.Max = 6;

this.top1 = -1;

this.top2 = -1;

this.count = 0;

this.stack1 = new Array(this.Max);

this.stack2 = new Array(this.Max);

}

enQueue(data)

{

this.push1(data);

this.count++;

}

push1(data){

if(this.top1 == this.Max-1){

console.log("Queue is full");

}

else{

this.stack1[++this.top1] = data;

console.log(data + "is added to queue");

}

}

deQueue(){

console.log("Dequeue :")

if(this.top1 == -1 && this.top2 == -1){

console.log("The Queue is full");

}

else{

for(let i = 0; i<this.count; i++){

let x = this.pop1();

this.push2(x);

}

}

let y = this.pop2();

//console.log("the deleted item is" +y);

this.count--;

for(let i=0;i<this.count; i++)

{

let z = this.pop2();

this.push1(z);

}

}

pop1(){

if(this.top1 == -1){

console.log("Queue is empty");

}

else{

let temp = this.stack1[this.top1];

//console.log("The popped item is" +temp);

this.top1--;

return temp;

}

}

pop2()

{

if (this.top2 == -1)

{

console.log("Stack is empty");

}

else

{

let pop = this.stack2[this.top2];

this.top2--;

return pop;

}

}

push2(x){

if(this.top2 == this.Max-1){

console.log("Queue is full");

}

else{

this.stack2[++this.top2] = x;

console.log(x + "is added to queue");

}

}

peek()

{

if (this.top1 == -1 && this.top2 == -1)

{

console.log("Stack is empty");

}

else

{

for (let i = 0; i < this.count; i++)

{

let x = this.pop1();

this.push2(x);

}

let y = this.stack2[this.top2];

console.log("The top element is :" + y);

for (let i = 0; i < this.count; i++)

{

let z = this.pop2();

this.push1(z);

}

}

}

}

**19. Circular QUEUE**

**i.USING ARRAY**

class createQueueArray

{

private int Max;

private int front;

private int rear;

private int[] queue;

public createQueueArray()

{

Max = 5;

queue = new int[Max];

front = -1;rear = -1;

}

public void enqueue(int data)

{

if(front ==-1&& rear== -1)

{

front = rear = 0;

queue[rear] = data;

}

else if(((rear+1)%Max) == front)

{

Console.WriteLine("queue is full");

return;

}

else

{

rear = (rear + 1) % Max; //As the queue has to be circular, we will check this condition instead of incrementing.

queue[rear] = data;

}

}

public void dequeue()

{

if (front == -1 && rear == -1)

{

Console.WriteLine("The queue is empty");

}

else if (front == rear)

{ //only 1 element in the queue

front = rear = -1; //To make queue empty;

}

else

{

// Console.WriteLine(queue[front]);

front = (front+1)% Max;

}

}

public void peek()

{

if (front == -1 && rear == -1)

{

Console.WriteLine("The queue is empty");

}

else

{

Console.WriteLine("The top value is :"+queue[front]);

}

}

public void display()

{

if(front==-1 && rear == -1)

{

Console.WriteLine("The queue is empty");

}

else

{

//while(i != rear+1)

//{

// Console.WriteLine(queue[i]);

// i= (i+1)% Max;

//}

for (int i = 0; i < queue.Length; i++)

{

Console.WriteLine(queue[i]);

}

}

}

}

**ALGO JS**

class queue {

constructor() {

this.queue = [];

this.Max = 6;

this.front = -1;

this.rear = -1;

this.queue = new Array(this.Max);

}

enQueue(data)

{

if(this.front ==-1&& this.rear== -1)

{

this.front = this.rear = 0;

this.queue[this.rear] = data;

}

else if(Math.floor((this.rear+1)%this.Max) == this.front)

{

console.log("queue is full");

return;

}

else

{

this.rear = Math.floor((this.rear + 1) % this.Max); //As the queue has to be circular, we will check this condition instead of incrementing.

this.queue[this.rear] = data;

}

}

dequeue()

{

if (this.front == -1 && this.rear == -1)

{

console.log("The queue is empty");

}

else if (this.front == this.rear)

{ //only 1 element in the queue

this.front = this.rear = -1; //To make queue empty;

}

else

{

// Console.WriteLine(queue[front]);

this.front = Math.floor((this.front+1)% this.Max);

}

}

peek()

{

if (this.front == -1 && this.rear == -1)

{

console.log("The queue is empty");

}

else

{

console.log("The top value is :"+this.queue[this.front]);

}

}

display()

{

if(this.front==-1 && this.rear == -1)

{

console.log("The queue is empty");

}

else

{

for (let i = 0; i < this.queue.length; i++)

{

console.log(this.queue[i]);

}

}

}

}

**II.USING LINKED LIST**

public class Node

{

public int value;

public Node next;

}

public class LinkedList

{

public Node front ,rear;

public LinkedList()

{

front = null; rear = null;

}

public void enqueue(int x)

{

Node newNode = new Node();

newNode.value = x;

newNode.next = null;

if(rear == null)

{

front = rear = newNode;

rear.next = front;

}

else

{

rear.next = newNode;

rear = newNode;

rear.next = front;

}

}

public void dequeue()

{

Node temp = front;

if (front == null && rear == null)

Console.WriteLine("Queue is empty");

else if (front == rear)

{

front = rear = null;

temp = null;

}

else

{

front = front.next;

rear.next = front;

temp = null;

}

}

public void peek()

{

if (front == null && rear == null)

Console.WriteLine("Queue is empty");

else

{

Console.WriteLine("The top value is " +front.value);

}

}

public void display()

{

Node temp = front;

if(front== null && rear== null)

{

Console.WriteLine("The List is empty");

}

else

{

while(temp.next != front)

{

Console.WriteLine(temp.value);

temp = temp.next;

}

Console.WriteLine("{0:D}", temp.value);

}

}

}

**ALGO JS**

class Node{

constructor(value){

this.data = value;

this.next = null;

}

}

class queueLinkedList{

constructor(){

this.front =this.rear= null;

}

enQueue(value){

const newNode = new Node(value);

if(this.rear == null && this.front == null)

this.front= this.rear = newNode;

else{

this.rear.next = newNode;

this.rear = newNode;

}

}

deQueue(){

if(this.rear == null && this.front == null){

console.log("queue is empty");

}

const temp = this.front;

this.front = this.front.next;

console.log("the deleted item is " +temp.data);

this.temp = null;

}

peek(){

if(this.rear == null && this.front == null){

console.log("queue is empty");

}

else

console.log("The top data is " +this.front.data);

}

}

**20. CREATE BINARY TREE**

**ALGO C#**

public class Node

{

public int data;

public Node leftChild;

public Node rightChild;

public Node(int value)

{

data = value; leftChild = null; rightChild = null;

}

}

class Program

{

public Node root;

public Node createBinaryTree(int[] arr,int i,Node root)

{

if (i < arr.Length)

{

Node temp = new Node(arr[i]);

root = temp;

root.leftChild = createBinaryTree(arr, 2 \* i + 1, root.leftChild);

root.rightChild = createBinaryTree(arr, 2 \* i + 2, root.rightChild);

}

return root;

}

public void inorderTraversal(Node root)

{

if (root != null)

{

inorderTraversal(root.leftChild);

Console.WriteLine(root.data + "");

inorderTraversal(root.rightChild);

}

}

public void preOrderTraversal(Node root)

{

if (root != null)

{

Console.WriteLine(root.data + "");

preOrderTraversal(root.leftChild);

preOrderTraversal(root.rightChild);

}

}

public void postOrderTraversal(Node root)

{

if (root != null)

{

postOrderTraversal(root.leftChild);

postOrderTraversal(root.rightChild);

Console.WriteLine(root.data + "");

}

}

static void Main(string[] args)

{

Program p = new Program();

int[] arr = { 7,6,5,4,3,2,1};

p.root = p.createBinaryTree(arr,0,p.root);

Console.WriteLine("\n Inorder: ");

p.inorderTraversal(p.root);

Console.WriteLine("\n Preorder: ");

p.preOrderTraversal(p.root);

Console.WriteLine("\n Postorder: ");

p.postOrderTraversal(p.root);

}

}

**ALGO JS**

class Node

{

constructor(data)

{

this.left = null;

this.right = null;

this.val = data;

}

}

class createBT {

constructor(root = null) {

this.root = root;

}

createBinaryTree(arr, i, root) {

if (i < arr.length) {

let temp = new Node(arr[i]);

root = temp;

root.left = this.createBinaryTree(arr, 2 \* i + 1, root.left);

root.right = this.createBinaryTree(arr, 2 \* i + 2, root.right);

}

return root;

}

preOrderTraversal(root) {

if (root != null) {

console.log(root.val);

this.preOrderTraversal(root.left);

this.preOrderTraversal(root.right);

}

}

inOrderTraversal(root) {

if (root != null) {

this.inOrderTraversal(root.left);

console.log(root.val);

this.inOrderTraversal(root.right);

}

}

postOrderTraversal(root) {

if (root != null) {

this.postOrderTraversal(root.left);

this.postOrderTraversal(root.right);

console.log(root.val);

}

}

}

let arr = [7,6,5,4,3,2,1];

const p = new createBT();

p.root = p.createBinaryTree(arr,0,p.root);

console.log("\n inorder: ");

p.inOrderTraversal(p.root);

console.log("\n Preorder: ");

p.preOrderTraversal(p.root);

console.log("\n Postorder: ");

p.postOrderTraversal(p.root);

**21. BINARY SEARCH TREE**

**ALGO C#**

public class TreeNode

{

public int val;

public TreeNode leftChild;

public TreeNode rightChild;

public TreeNode(int val = 0, TreeNode leftChild = null, TreeNode rightChild = null)

{

this.val = val;

this.leftChild = leftChild;

this.rightChild = rightChild;

}

}

class Program

{

public TreeNode root;

public void Add(int data)

{

// Create a new node

TreeNode node = new TreeNode(data);

if (this.root == null)

{

// When adds a first node in bst

this.root = node;

}

else

{

TreeNode parent = this.root;

// Add new node to proper position

while (parent != null)

{

if (data < parent.val)

{

if (parent.leftChild == null)

{

// When leftChild child empty

// So add new node here

parent.leftChild = node;

return;

}

else

{

// Otherwise

// Visit leftChild sub-tree

parent = parent.leftChild;

}

}

else if(data > parent.val)

{

if (parent.rightChild == null)

{

// When rightChild child empty

// So add new node here

parent.rightChild = node;

return;

}

else

{

// Visit rightChild sub-tree

parent = parent.rightChild;

}

}

else

{

return;

}

}

}

}

public void TraversePreOrder(TreeNode parent)

{

if (parent != null)

{

Console.Write(parent.val + " ");

TraversePreOrder(parent.leftChild);

TraversePreOrder(parent.rightChild);

}

}

public void TraverseInOrder(TreeNode parent)

{

if (parent != null)

{

TraverseInOrder(parent.leftChild);

Console.Write(parent.val + " ");

TraverseInOrder(parent.rightChild);

}

}

public void TraversePostOrder(TreeNode parent)

{

if (parent != null)

{

TraversePostOrder(parent.leftChild);

TraversePostOrder(parent.rightChild);

Console.Write(parent.val + " ");

}

}

public int GetTreeDepth(TreeNode parent)

{

if (parent == null)

return 0;

else

{

int leftChildDepth = GetTreeDepth(parent.leftChild);

int rightChildDepth = GetTreeDepth(parent.rightChild);

if(leftChildDepth > rightChildDepth)

return leftChildDepth + 1;

else

return rightChildDepth + 1;

}

}

public bool search(TreeNode parent,int value)

{

while(parent!= null)

{

if (value == parent.val)

return true;

else if (value < parent.val)

return search(parent.leftChild, value);

else

return search(parent.rightChild, value);

}

return false;

}

public int findMax(TreeNode node)

{

while (node.leftChild != null)

{

node = node.leftChild;

}

return node.val;

}

public TreeNode remove(TreeNode parent, int key)

{

if (parent == null) return parent;

if (key < parent.val)

parent.leftChild = remove(parent.leftChild, key);

else if (key > parent.val)

parent.rightChild = remove(parent.rightChild, key);

// if value is same as parent's value, then this is the TreeNode to be deleted

else

{

// TreeNode with only one child or no child

if (parent.leftChild == null)

return parent.rightChild;

else if (parent.rightChild == null)

return parent.leftChild;

// TreeNode with two children: Get the inorder successor (smallest in the rightChild subtree)

parent.val = findMax(parent.rightChild);

// Delete the inorder successor

parent.rightChild = remove(parent.rightChild, parent.val);

}

return parent;

}

static void Main(string[] args)

{

Program binaryTree = new Program();

binaryTree.Add(12);

binaryTree.Add(11);

binaryTree.Add(14);

binaryTree.Add(20);

binaryTree.Add(15);

binaryTree.Add(6);

binaryTree.Add(8);

binaryTree.Add(17);

binaryTree.Add(5);

Console.WriteLine("PreOrder Traversal:");

binaryTree.TraversePreOrder(binaryTree.root);

Console.WriteLine();

Console.WriteLine("InOrder Traversal:");

binaryTree.TraverseInOrder(binaryTree.root);

Console.WriteLine();

Console.WriteLine("PostOrder Traversal:");

binaryTree.TraversePostOrder(binaryTree.root);

Console.WriteLine();

int depthData = binaryTree.GetTreeDepth(binaryTree.root);

Console.WriteLine("The height of tree is : "+depthData);

bool data = binaryTree.search(binaryTree.root, 20);

Console.WriteLine(data== true?"The data is present " :"Not present");

binaryTree.remove(binaryTree.root, 12);

Console.Write("The list after deletion :");

binaryTree.TraverseInOrder(binaryTree.root);

// //Console.ReadLine();

}

}

**ALGO JS**

class Node

{

constructor(data)

{

this.left = null;

this.right = null;

this.val = data;

}

}

class createBT {

constructor(root = null) {

this.root = root;

}

Add(data){

let node = new Node(data);

if(this.root == null)

this.root = node;

else{

let parent = this.root;

while(parent != null) {

if(data<parent.val) {

if (parent.left == null) {

parent.left = node;

return;

}

else

{

parent = parent.left;

}

}

else if(data>parent.val){

if(parent.right == null){

parent.right = node;

return;

}

else{

parent = parent.right

}

}

else{

return;

}

}

}

}

preOrderTraversal(root) {

if (root != null) {

console.log(root.val);

this.preOrderTraversal(root.left);

this.preOrderTraversal(root.right);

}

}

inOrderTraversal(root) {

if (root != null) {

this.inOrderTraversal(root.left);

console.log(root.val);

this.inOrderTraversal(root.right);

}

}

postOrderTraversal(root) {

if (root != null) {

this.postOrderTraversal(root.left);

this.postOrderTraversal(root.right);

console.log(root.val);

}

}

GetTreeDepth(parent)

{

if (parent == null)

return 0;

else

{

let leftChildDepth = this.GetTreeDepth(parent.left);

let rightChildDepth = this.GetTreeDepth(parent.right);

if(leftChildDepth > rightChildDepth)

return leftChildDepth + 1;

else

return rightChildDepth + 1;

}

}

search( parent, value)

{

while(parent!= null)

{

if (value == parent.val)

return true;

else if (value < parent.val)

return this.search(parent.left, value);

else

return this.search(parent.right, value);

}

return false;

}

findMax( node)

{

while (node.left != null)

{

node = node.left;

}

return node.val;

}

remove( parent, key)

{

if (parent == null) return parent;

if (key < parent.val)

parent.left = this.remove(parent.left, key);

else if (key > parent.val)

parent.right = this.remove(parent.right, key);

// if value is same as parent's value, then this is the TreeNode to be deleted

else

{

// TreeNode with only one child or no child

if (parent.left == null)

return parent.right;

else if (parent.right == null)

return parent.left;

// TreeNode with two children: Get the inorder successor (smallest in the rightChild subtree)

parent.val = this.findMax(parent.right);

// Delete the inorder successor

parent.right = this.remove(parent.right, parent.val);

}

return parent;

}

}

const binaryTree = new createBT();

binaryTree.Add(12);

binaryTree.Add(11);

binaryTree.Add(14);

binaryTree.Add(20);

binaryTree.Add(15);

binaryTree.Add(6);

binaryTree.Add(8);

binaryTree.Add(17);

binaryTree.Add(5);

console.log("PreOrder Traversal:");

binaryTree.preOrderTraversal(binaryTree.root);

console.log("InOrder Traversal:");

binaryTree.inOrderTraversal(binaryTree.root);

console.log("PostOrder Traversal:");

binaryTree.postOrderTraversal(binaryTree.root);

console.log("Tree height is :");

console.log(binaryTree.GetTreeDepth(binaryTree.root));

console.log("The value is present :");

console.log(binaryTree.search(binaryTree.root,17));

console.log("The list after deletion :");

console.log(binaryTree.remove(binaryTree.root, 12));

console.log("InOrder Traversal:");

binaryTree.inOrderTraversal(binaryTree.root);

**22. CONSTRUCT BINARY TREE FROM POST & INORDER TRAVERSAL**

**ALGO C#**

class Node

{

public int data;

public Node leftChild;

public Node rightChild;

public Node(int value)

{

data = value; leftChild = rightChild = null;

}

}

class Program

{

public Node traverse(int[] post,int postIdx, int[] inOrdr,int start,int end)

{

if (start > end || postIdx<0)

return null;

Node tNode = new Node(post[postIdx]);

if (start == end)

return tNode;

int i;

for(i = start;i<end; i++) {

if (inOrdr[i] == tNode.data)

break;

}

tNode.leftChild = traverse(post, postIdx - 1 - (end - i), inOrdr, start, i - 1);

tNode.rightChild = traverse(post, postIdx-1, inOrdr, i+1, end);

return tNode;

}

public void printPostOrder(Node node)

{

if(node!= null)

{

printPostOrder(node.leftChild);

printPostOrder(node.rightChild);

Console.Write(node.data + " ");

}

}

static void Main(string[] args)

{

Program p = new Program();

int[] postOrder = { 9, 1, 2, 12, 7, 5, 3, 11, 4, 8 };

int[] inOrder = { 9, 5, 1, 7, 2, 12, 8, 4, 3, 11 };

int length = postOrder.Length;

Node root = p.traverse(postOrder, postOrder.Length-1, inOrder,0, length-1);

p.printPostOrder(root);

//Console.WriteLine("Hello World!");

}

}

**ALGO JS**

class Node

{

constructor(data)

{

this.left = null;

this.right = null;

this.val = data;

}

}

function traverse(post,postIdx, inOrdr,start,end)

{

if (start > end || postIdx<0)

return null;

let tNode = new Node(post[postIdx]);

if (start == end)

return tNode;

let i;

for( i = start;i<end; i++) {

if (inOrdr[i] == tNode.val)

break;

}

tNode.left = traverse(post, postIdx - 1 - (end - i), inOrdr, start, i - 1);

tNode.right = traverse(post, postIdx-1, inOrdr, i+1, end);

return tNode;

}

function postOrderTraversal(root) {

if (root != null) {

postOrderTraversal(root.left);

postOrderTraversal(root.right);

console.log(root.val);

}

}

let postOrder = [9, 1, 2, 12, 7, 5, 3, 11, 4, 8 ];

let inOrder = [9, 5, 1, 7, 2, 12, 8, 4, 3, 11 ];

let root = traverse(postOrder, postOrder.length-1, inOrder,0, postOrder.length-1);

postOrderTraversal(root);

**23. CONSTRUCT BT FROM PRE & IN ORDER**

**ALGO C#**

public class Node

{

public int data;

public Node leftChild;

public Node rightChild;

public Node(int value)

{

data = value; leftChild = rightChild = null;

}

}

class Program

{

public static int preIndex = 0;

public Node traverseList(int[] inOrdr, int[]pre,int start, int end)

{

if (start > end )

return null;

Node tnode = new Node(pre[preIndex++]);

//If node has No children

if (start == end)

return tnode;

int i;

for (i = start; i < end; i++)

{

if (inOrdr[i] == tnode.data)

{

break;

}

}

//else find the index of tnode in inorder

tnode.leftChild = traverseList(inOrdr, pre, start, i - 1);

tnode.rightChild = traverseList(inOrdr, pre,i+1,end);

return tnode;

}

public void printInorder(Node node)

{

if (node == null)

{

return;

}

printInorder(node.leftChild);

Console.Write(node.data + " ");

printInorder(node.rightChild);

}

static void Main(string[] args)

{

Program p = new Program();

int[] preOrder = { 1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7 };

int[] inorder = { 8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7 };

int length = inorder.Length;

Node root = p.traverseList(inorder,preOrder, 0, length - 1);

p.printInorder(root);

}

}

**ALGO JS**

class Node

{

constructor(data)

{

this.left = null;

this.right = null;

this.val = data;

}

}

let preIndex = 0;

function traverseList(inOrdr, pre,start,end)

{

if (start > end )

return null;

let tnode = new Node(pre[preIndex++]);

//If node has No children

if (start == end)

return tnode;

let i;

for (i = start; i < end; i++)

{

if (inOrdr[i] == tnode.val)

{

break;

}

}

//else find the index of tnode in inorder

tnode.left = traverseList(inOrdr, pre, start, i - 1);

tnode.right = traverseList(inOrdr, pre,i+1,end);

return tnode;

}

function inOrderTraversal(root) {

if (root != null) {

inOrderTraversal(root.left);

console.log(root.val);

inOrderTraversal(root.right);

}

}

let preOrder = [1, 2, 4, 8, 9, 10, 11, 5, 3, 6, 7 ];

let inorder = [8, 4, 10, 9, 11, 2, 5, 1, 6, 3, 7 ];

let root = traverseList(inorder,preOrder, 0, inorder.length - 1);

inOrderTraversal(root);

**24. CONSTRUCT BT FROM POSTPRE ORDER**

**ALGO C#**

public class Node

{

public int data;

public Node rightChild;

public Node leftChild;

public Node(int x)

{

data = x; rightChild = leftChild = null;

}

}

class Program

{

int preIndex = 0;

public Node traverse(int[] pre, int[] post, int start,int end, int size) {

if (start > end)

return null;

Node newNode = new Node(pre[preIndex++]);

if (start == end)

return newNode;

int i;

for(i = start; i <= end; i++)

{

if(post[i]== pre[preIndex])

break;

}

newNode.leftChild = traverse(pre, post, start, i, size);

newNode.rightChild = traverse(pre, post, i + 1, end - 1, size);

return newNode;

}

public void printPre(Node node)

{

if(node!= null)

{

Console.WriteLine(node.data + " ");

printPre(node.leftChild);

printPre(node.rightChild);

}

}

static void Main(string[] args)

{

Program p = new Program();

int[] postOrder = { 8, 9, 4, 5, 2, 6, 7, 3, 1 };

int[] preOrder = { 1, 2, 4, 8, 9, 5, 3, 6, 7 };

int size = preOrder.Length;

Node root = p.traverse(preOrder, postOrder, 0, size - 1, size);

p.printPre(root);

}

}

**ALGO JS**

class Node

{

constructor(data)

{

this.left = null;

this.right = null;

this.val = data;

}

}

let preIndex = 0;

function traverse(pre, post,start,end,size) {

if (start > end)

return null;

let newNode = new Node(pre[preIndex++]);

if (start == end)

return newNode;

let i;

for(i = start; i <= end; i++)

{

if(post[i]== pre[preIndex])

break;

}

newNode.left = traverse(pre, post, start, i, size);

newNode.right = traverse(pre, post, i + 1, end - 1, size);

return newNode;

}

function preOrderTraversal(root) {

if (root != null) {

console.log(root.val);

preOrderTraversal(root.left);

preOrderTraversal(root.right);

}

}

let postOrder = [ 8, 9, 4, 5, 2, 6, 7, 3, 1 ];

let preOrder = [1, 2, 4, 8, 9, 5, 3, 6, 7 ];

let size = preOrder.length;

let root = traverse(preOrder, postOrder, 0, size - 1, size);

preOrderTraversal(root);

**25. GRAPH USING BFS**

**Time Complexity : O(V+E) => V= VERTICES, E = EDGES;**

**SPACE COMPLEXITY : O(V);**

**ALGO C#**

class Graph

{

// No. of vertices

private int vertex;

//Adjacency Lists

LinkedList<int>[] \_adj;

public Graph(int vertices)

{

\_adj = new LinkedList<int>[vertices];

for (int i = 0; i < \_adj.Length; i++)

{

\_adj[i] = new LinkedList<int>();

}

vertex = vertices;

}

// Function to add an edge into the graph

public void AddEdge(int node, int edge)

{

\_adj[node].AddLast(edge);

//\_adj[edge].AddLast(node);

}

// Prints BFS traversal from a given source s

public void BFS(int s)

{

// Mark all the vertices as not

// visited(By default set as false)

bool[] visited = new bool[vertex];

for (int i = 0; i < vertex; i++)

visited[i] = false;

// Create a queue for BFS

//LinkedList<int> queue = new LinkedList<int>();

Queue<int> queue = new Queue<int>();

// Mark the current node as

// visited and enqueue it

visited[s] = true;

queue.Enqueue(s);

//queue.AddLast(s);

while (queue.Any())

{

// Dequeue a vertex from queue

// and print it

s = queue.First();

Console.Write(s + " ");

//queue.RemoveFirst();

queue.Dequeue();

// Get all adjacent vertices of the

// dequeued vertex s. If a adjacent

// has not been visited, then mark it

// visited and enqueue it

LinkedList<int> list = \_adj[s];

foreach (var val in list)

{

if (!visited[val])

{

visited[val] = true;

//queue.AddLast(val);

queue.Enqueue(val);

}

}

}

}

// Driver code

static void Main(string[] args)

{

Graph g = new Graph(7);

g.AddEdge(0, 1);

g.AddEdge(0, 3);

g.AddEdge(1, 2);

g.AddEdge(3, 2);

g.AddEdge(2, 4);

g.AddEdge(2, 5);

g.AddEdge(3, 4);

g.AddEdge(4, 6);

g.AddEdge(5, 1);

g.AddEdge(6, 1);

Console.Write("Following is Breadth First " +

"Traversal(starting from " +

"vertex 2)\n");

g.BFS(0);

}

}

**ALGO JS**

class Graph {

constructor(v) {

this.V = v;

this.adj = new Array(v);

for (let i = 0; i < v; i++)

this.adj[i] = [];

}

AddEdge(n, e) {

this.adj[n].push(e);

}

BFS(s){

const visited =new Array(this.V).fill(false);

let queue = new Queue();

visited[s] = true;

queue.enqueue(s);

while(queue.length) {

let getQueueElmnt = queue.dequeue();

console.log(getQueueElmnt + " ");

let list = this.adj.get(getQueueElmnt );

for(var iin list){

let data = list[i];

if(!visited[data]){

visited[data] = true;

q.enqueue(data);

}

}

}

}

}

const g = new Graph(7);

g.AddEdge(0, 1);

g.AddEdge(0, 3);

g.AddEdge(1, 2);

g.AddEdge(3, 2);

g.AddEdge(2, 4);

g.AddEdge(2, 5);

g.AddEdge(3, 4);

g.AddEdge(4, 6);

g.AddEdge(5, 1);

g.AddEdge(6, 1);

console.log("Following is Breadth First " +"Traversal(starting from " + "vertex 2)\n");

g.BFS(0);

**26. GRAPH USING DFS**

**Time Complexity : O(V+E) => V= VERTICES, E = EDGES;**

**SPACE COMPLEXITY : O(V);**

**ALGO C#**

class Graph

{

private int vertex; // No. of vertices

// Array of lists for

// Adjacency List Representation

private LinkedList<int>[] adj;

// Constructor

Graph(int vertices)

{

vertex = vertices;

adj = new LinkedList<int>[vertices];

for (int i = 0; i < vertices; ++i)

adj[i] = new LinkedList<int>();

}

// Function to Add an edge into the graph

void AddEdge(int node, int edge)

{

adj[node].AddLast(edge); // Add w to v's list.

}

// A function used by DFS

void DFSUtil(int v, bool[] visited)

{

// Mark the current node as visited

// and print it

visited[v] = true;

Console.Write(v + " ");

// Recur for all the vertices

// adjacent to this vertex

LinkedList<int> vList = adj[v];

foreach (var n in vList)

{

if (!visited[n])

DFSUtil(n, visited);

}

}

// The function to do DFS traversal.

// It uses recursive DFSUtil()

void DFS(int v)

{

// Mark all the vertices as not visited

// (set as false by default in c#)

bool[] visited = new bool[vertex];

// Call the recursive helper function

// to print DFS traversal

DFSUtil(v, visited);

}

// Driver Code

public static void Main(String[] args)

{

Graph g = new Graph(7);

g.AddEdge(0, 1);

g.AddEdge(0, 3);

g.AddEdge(1, 2);

g.AddEdge(3, 2);

g.AddEdge(2, 4);

g.AddEdge(2, 5);

g.AddEdge(3, 4);

g.AddEdge(4, 6);

g.AddEdge(5, 1);

g.AddEdge(6, 1);

Console.WriteLine(

"Following is Depth First Traversal "

+ "(starting from vertex 2)");

g.DFS(0);

Console.ReadKey();

}

}

**ALGO JS**

class Graph  
{  
  
 constructor(v)  
 {  
 this.V = v;  
 this.adj = new Array(v);  
 for(let i = 0; i < v; i++)  
 this.adj[i] = [];  
 }  
  
 addEdge(n, e)  
 {  
  
 this.adj[n].push(e);  
 }  
  
 DFSUtil(v, visited)  
 {  
  
 visited[v] = true;  
 console.log(v + " ");  
  
 for(let i of this.adj[v].values())  
 {  
 let n = i  
 if (!visited[n])  
 this.DFSUtil(n, visited);  
 }  
 }  
  
 DFS(v)  
 {  
 let visited = new Array(this.V);  
 for(let i = 0; i < this.V; i++)  
 visited[i] = false;  
  
 this.DFSUtil(v, visited);  
 }  
}  
  
const g = new Graph(4);  
  
g.addEdge(0, 1);  
g.addEdge(0, 2);  
g.addEdge(1, 2);  
g.addEdge(2, 0);  
g.addEdge(2, 3);  
g.addEdge(3, 3);  
  
console.log("Following is Depth First Traversal " + "(starting from vertex 2)");  
  
g.DFS(2);

**27. detect cycle using union find**

**ALGO C#**

using System;

namespace DetectCycleUsingUnionFindAlgo

{

class Graph {

public int V, E;

public Edge[] edge;

public Graph(int nV, int nE)

{

V = nV;

E = nE;

edge = new Edge[E];

for (int i = 0; i < E; i++)

{

edge[i] = new Edge();

}

}

// class to represent edge

public class Edge {

public int src, dest;

}

// class to represent Subset

class subset

{

public int parent;

public int rank;

}

// A utility function to find

// set of an element i (uses

// path compression technique)

int find(subset[] subsets, int i)

{

if (subsets[i].parent != i)

subsets[i].parent = find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// A function that does union

// of two sets of x and y

// (uses union by rank)

void Union(subset[] subsets, int x, int y)

{

int xroot = x;

int yroot = y;

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[yroot].rank < subsets[xroot].rank)

subsets[yroot].parent = xroot;

else {

subsets[xroot].parent = yroot;

subsets[yroot].rank++;

}

}

// The main function to check whether

// a given graph contains cycle or not

int isCycle(Graph graph)

{

int V = graph.V;

int E = graph.E;

subset[] subsets = new subset[V];

for (int v = 0; v < V; v++)

{

subsets[v] = new subset();

subsets[v].parent = v;

subsets[v].rank = 0;

}

for (int e = 0; e < E; e++)

{

int x = find(subsets, graph.edge[e].src);

int y = find(subsets, graph.edge[e].dest);

if (x == y)

return 1;

Union(subsets, x, y);

}

return 0;

}

// Driver Code

static public void Main(String[] args)

{

/\* Let us create the following graph

0

| \

| \

1-----2 \*/

int V = 3, E = 3;

Graph graph = new Graph(V, E);

// add edge 0-1

graph.edge[0].src = 0;

graph.edge[0].dest = 1;

// add edge 1-2

graph.edge[1].src = 1;

graph.edge[1].dest = 2;

// add edge 0-2

graph.edge[2].src = 0;

graph.edge[2].dest = 2;

if (graph.isCycle(graph) == 1)

Console.WriteLine("Graph contains cycle");

else

Console.WriteLine(

"Graph doesn't contain cycle");

}

}

}

**ALGO JS**

class Edge {

constructor()

{

this.src = 0;

this.dest = 0;

}

}

class subset

{

constructor(v)

{

this.parent = 0;

this.rank = 0;

}

}

class Graph{

constructor(nV,nE){

this.V = nV;

this.E = nE;

this.edge = new Array(this.E);

for (let i = 0; i < this.E; i++)

{

this.edge[i] = [];

}

}

// class to represent Subset

Union(subsets, x,y)

{

let xroot = this.find(subsets, x);

let yroot = this.find(subsets, y);

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[yroot].rank < subsets[xroot].rank)

subsets[yroot].parent = xroot;

else {

subsets[xroot].parent = yroot;

subsets[yroot].rank++;

}

}

find(subsets,i)

{

if (subsets[i].parent != i)

subsets[i].parent = this.find(subsets, subsets[i].parent);

return subsets[i].parent;

}

isCycle(graph)

{

let V = graph.V;

let E = graph.E;

let subsets = new subset(V);

for (let v = 0; v < V; v++)

{

subsets[v] = new subset();

subsets[v].parent = v;

subsets[v].rank = 0;

}

for (let e = 0; e < E; e++)

{

let x = this.find(subsets, graph.edge[e].src);

let y = this.find(subsets, graph.edge[e].dest);

if (x == y)

return 1;

this.Union(subsets, x, y);

}

return 0;

}

}

let V = 3, E = 3;

const graph = new Graph(V, E);

graph.edge[0].src = 0;

graph.edge[0].dest = 1;

graph.edge[1].src = 1;

graph.edge[1].dest = 2;

graph.edge[2].src = 0;

graph.edge[2].dest = 2;

if (graph.isCycle(graph) == 1)

console.log("Graph contains cycle");

else

console.log("Graph doesn't contain cycle");

**28. KRUSKAL’S MST ALGO**

**ALGO C#**

class Graph

{

// A class to represent a graph edge

class Edge : IComparable<Edge>

{

public int src, dest, weight;

// Comparator function used for sorting edges

// based on their weight

public int CompareTo(Edge compareEdge)

{

return this.weight

- compareEdge.weight;

}

}

// A class to represent

// a subset for union-find

public class subset

{

public int parent, rank;

};

int V, E; // V-> no. of vertices & E->no.of edges

Edge[] edge; // collection of all edges

// Creates a graph with V vertices and E edges

Graph(int v, int e)

{

V = v;

E = e;

edge = new Edge[E];

for (int i = 0; i < e; ++i)

edge[i] = new Edge();

}

// A utility function to find set of an element i

// (uses path compression technique)

int find(subset[] subsets, int i)

{

// find root and make root as

// parent of i (path compression)

if (subsets[i].parent != i)

subsets[i].parent

= find(subsets, subsets[i].parent);

return subsets[i].parent;

}

// A function that does union of

// two sets of x and y (uses union by rank)

void Union(subset[] subsets, int x, int y)

{

int xroot = find(subsets, x);

int yroot = find(subsets, y);

// Attach smaller rank tree under root of

// high rank tree (Union by Rank)

if (subsets[xroot].rank < subsets[yroot].rank)

subsets[xroot].parent = yroot;

else if (subsets[xroot].rank > subsets[yroot].rank)

subsets[yroot].parent = xroot;

// If ranks are same, then make one as root

// and increment its rank by one

else

{

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

// The main function to construct MST

// using Kruskal's algorithm

void KruskalMST()

{

// This will store the

// resultant MST

Edge[] result = new Edge[V];

int e = 0; // An index variable, used for result[]

int i

= 0; // An index variable, used for sorted edges

for (i = 0; i < V; ++i)

result[i] = new Edge();

// Step 1: Sort all the edges in non-decreasing

// order of their weight. If we are not allowed

// to change the given graph, we can create

// a copy of array of edges

Array.Sort(edge);

// Allocate memory for creating V subsets

subset[] subsets = new subset[V];

// Create V subsets with single elements

for (int v = 0; v < V; ++v)

{

subsets[v] = new subset();

subsets[v].parent = v;

subsets[v].rank = 0;

}

i = 0; // Index used to pick next edge

// Number of edges to be taken is equal to V-1

while (e < V - 1)

{

// Step 2: Pick the smallest edge. And increment

// the index for next iteration

Edge next\_edge = new Edge();

next\_edge = edge[i++];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

// If including this edge does't cause cycle,

// include it in result and increment the index

// of result for next edge

if (x != y)

{

result[e++] = next\_edge;

Union(subsets, x, y);

}

// Else discard the next\_edge

}

// print the contents of result[] to display

// the built MST

Console.WriteLine("Following are the edges in "

+ "the constructed MST");

int minimumCost = 0;

for (i = 0; i < e; ++i)

{

Console.WriteLine(result[i].src + " -- "

+ result[i].dest

+ " == " + result[i].weight);

minimumCost += result[i].weight;

}

Console.WriteLine("Minimum Cost Spanning Tree"

+ minimumCost);

Console.ReadLine();

}

// Driver Code

public static void Main(String[] args)

{

/\* Let us create following weighted graph

10

0--------1

| \ |

6| 5\ |15

| \ |

2--------3

4 \*/

int V = 4; // Number of vertices in graph

int E = 5; // Number of edges in graph

Graph graph = new Graph(V, E);

// add edge 0-1

graph.edge[0].src = 0;

graph.edge[0].dest = 1;

graph.edge[0].weight = 10;

// add edge 0-2

graph.edge[1].src = 0;

graph.edge[1].dest = 2;

graph.edge[1].weight = 6;

// add edge 0-3

graph.edge[2].src = 0;

graph.edge[2].dest = 3;

graph.edge[2].weight = 5;

// add edge 1-3

graph.edge[3].src = 1;

graph.edge[3].dest = 3;

graph.edge[3].weight = 15;

// add edge 2-3

graph.edge[4].src = 2;

graph.edge[4].dest = 3;

graph.edge[4].weight = 4;

// Function call

graph.KruskalMST();

}

}

**ALGO JS**

class subset  
{  
 constructor(v)  
 {  
 this.parent = 0;  
 this.rank = 0;  
 }  
}  
class Edge {  
 constructor()  
 {  
 this.src = 0;  
 this.dest = 0;  
 this.weight = 0;  
 }  
}  
class Graph {  
 constructor(nV, nE) {  
 this.V = nV;  
 this.E = nE;  
 this.edge = new Array(this.E);  
 for (let i = 0; i < this.E; i++) {  
 this.edge[i] = [];  
 }  
 }  
 Union(subsets, x,y)  
 {  
 let xroot = this.find(subsets, x);  
 let yroot = this.find(subsets, y);  
  
 if (subsets[xroot].rank < subsets[yroot].rank)  
 subsets[xroot].parent = yroot;  
 else if (subsets[yroot].rank < subsets[xroot].rank)  
 subsets[yroot].parent = xroot;  
 else {  
 subsets[xroot].parent = yroot;  
 subsets[yroot].rank++;  
 }  
 }  
 find(subsets,i)  
 {  
 if (subsets[i].parent != i)  
 subsets[i].parent = this.find(subsets, subsets[i].parent);  
 return subsets[i].parent;  
 }  
 KruskalMST()  
 {  
  
 let result = new Edge(V);  
 let e = 0;  
 let i = 0;  
 for (i = 0; i < V; ++i)  
 result[i] = new Edge();  
  
 this.edge.sort(function(a, b) {  
 return a.weight - b.weight;  
 });  
  
 let subsets = new subset(V);  
 for (let v = 0; v < V; ++v)  
 {  
 subsets[v] = new subset();  
 subsets[v].parent = v;  
 subsets[v].rank = 0;  
 }  
  
 i = 0;  
 while (e < V - 1)  
 {  
 let next\_edge = new Edge();  
 next\_edge = this.edge[i++];  
 let x = this.find(subsets, next\_edge.src);  
 let y = this.find(subsets, next\_edge.dest);  
 if (x != y)  
 {  
 result[e++] = next\_edge;  
 this.Union(subsets, x, y);  
 }  
 }  
 console.log("Following are the edges in " + "the constructed MST");  
  
 let minimumCost = 0;  
 for (i = 0; i < e; ++i)  
 {  
 console.log(result[i].src + " -- " + result[i].dest + " == " + result[i].weight);  
 minimumCost += result[i].weight;  
 }  
  
 console.log("Minimum Cost Spanning Tree" + minimumCost);  
 }  
  
}  
let V = 4, E = 5;  
const graph = new Graph(V, E);  
// add edge 0-1  
graph.edge[0].src = 0;  
graph.edge[0].dest = 1;  
graph.edge[0].weight = 10;  
  
// add edge 0-2  
graph.edge[1].src = 0;  
graph.edge[1].dest = 2;  
graph.edge[1].weight = 6;  
  
// add edge 0-3  
graph.edge[2].src = 0;  
graph.edge[2].dest = 3;  
graph.edge[2].weight = 5;  
  
// add edge 1-3  
graph.edge[3].src = 1;  
graph.edge[3].dest = 3;  
graph.edge[3].weight = 15;  
  
// add edge 2-3  
graph.edge[4].src = 2;  
graph.edge[4].dest = 3;  
graph.edge[4].weight = 4;  
  
// Function call  
graph.KruskalMST();

**29. PRIMS MST ALGO**

**TIME COMPLEXITY : O ( ( V + E ) l o g V )** because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

**SPACE COMPLEXITY :** We need an array to maintain Min-Heap. Space O(E). So, Total space complexity is of order **O(V+E)**.

**ALGO C#**

class MST

{

// Number of vertices in the graph

static int V = 5;

// A utility function to find

// the vertex with minimum key

// value, from the set of vertices

// not yet included in MST

static int minKey(int[] key, bool[] mstSet)

{

// Initialize min value

int min = int.MaxValue, min\_index = -1;

for (int v = 0; v < V; v++)

if (mstSet[v] == false && key[v] < min)

{

min = key[v];

min\_index = v;

}

return min\_index;

}

// A utility function to print

// the constructed MST stored in

// parent[]

static void printMST(int[] parent, int[,] graph)

{

Console.WriteLine("Edge \tWeight");

int minCost = 0;

for (int i = 1; i < V; i++)

{

Console.WriteLine(parent[i] + " - " + i + "\t" + graph[i, parent[i]]);

minCost += graph[parent[i], i];

}

Console.WriteLine("The minimum spanning tree is" + minCost);

}

// Function to construct and

// print MST for a graph represented

// using adjacency matrix representation

static void primMST(int[,] graph)

{

// Array to store constructed MST

int[] parent = new int[V];

// Key values used to pick

// minimum weight edge in cut

int[] key = new int[V];

// To represent set of vertices

// included in MST

bool[] mstSet = new bool[V];

// Initialize all keys

// as INFINITE

for (int i = 0; i < V; i++)

{

key[i] = int.MaxValue;

mstSet[i] = false;

}

// Always include first 1st vertex in MST.

// Make key 0 so that this vertex is

// picked as first vertex

// First node is always root of MST

key[0] = 0;

parent[0] = -1;

// The MST will have V vertices

for (int count = 0; count < V - 1; count++)

{

// Pick thd minimum key vertex

// from the set of vertices

// not yet included in MST

int u = minKey(key, mstSet);

// Add the picked vertex

// to the MST Set

mstSet[u] = true;

// Update key value and parent

// index of the adjacent vertices

// of the picked vertex. Consider

// only those vertices which are

// not yet included in MST

for (int v = 0; v < V; v++)

// graph[u][v] is non zero only

// for adjacent vertices of m

// mstSet[v] is false for vertices

// not yet included in MST Update

// the key only if graph[u][v] is

// smaller than key[v]

if (graph[u, v] != 0 && mstSet[v] == false

&& graph[u, v] < key[v])

{

parent[v] = u;

key[v] = graph[u, v];

}

}

// print the constructed MST

printMST(parent, graph);

}

// Driver Code

public static void Main()

{

/\* Let us create the following graph

2 3

(0)--(1)--(2)

| / \ |

6| 8/ \5 |7

| / \ |

(3)-------(4)

9 \*/

int[,] graph = new int[,] { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

{ 0, 5, 7, 9, 0 } };

// Print the solution

primMST(graph);

}

}

**ALGO JS**

const V = 5;  
function minKey(key,mstSet)  
{  
 let min = Number.MAX\_SAFE\_INTEGER, min\_index = -1;  
  
 for (let v = 0; v < V; v++)  
 if (mstSet[v] == false && key[v] < min)  
 {  
 min = key[v];  
 min\_index = v;  
 }  
  
 return min\_index;  
}  
function primMST(graph)  
{  
 let parent = new Array(V).fill(0);  
 let key = new Array(V).fill(Number.MAX\_SAFE\_INTEGER);  
 let mstSet = new Array(V).fill(false);  
 key[0] = 0;  
 parent[0] = -1;  
 for (let count = 0; count < V - 1; count++)  
 {  
 let u = minKey(key, mstSet);  
 mstSet[u] = true;  
 for (let v = 0;v < V;v++)  
 {  
let data = graph[u];  
let keyValue = key[v];  
 if (data[v] != 0 && mstSet[v] == false && data[v] < keyValue) {  
 parent[v] = u;  
 key[v] = data[v];  
 }  
 }  
 }  
 printMST(parent, graph);  
}  
  
function printMST(parent, graph)  
{  
 console.log("Edge \tWeight");  
 let minCost = 0;  
 for (let i = 1; i < V; i++)  
 {  
 let par = graph[parent[i]]  
 console.log(parent[i] + " - " + i + "\t" + par[i]);  
 minCost += par[i];  
 }  
 console.log("The minimum spanning tree is" + minCost);  
}  
let graph = [ [0, 2, 0, 6, 0 ],  
 [2, 0, 3, 8, 5 ],  
 [0, 3, 0, 0, 7 ],  
 [6, 8, 0, 0, 9 ],  
 [ 0, 5, 7, 9, 0 ] ];  
primMST(graph);