INDEX

|  |  |  |
| --- | --- | --- |
| **Sr.No** | **Practical** | **Sign** |
| 1 | Implementation of Different Sorting Techniques. [Bubble, Merge, Insert, Selection, Shell, Quick] |  |
| 2 | Implementation of Searching algorithms. [Linear & Binary] |  |
| 3 | Operations on Arrays. [Traverse, Insert, Delete, Merge, Sort & Search] |  |
| 4 | Implementation of Linked lists. [Traverse, Insert, Delete, Search] |  |
| 5 | Implementation of Stacks. [Using Array and Linked List] |  |
| 6 | Implementation of Queue. [Using Array and Linked List] |  |
| 7 | Implementation of Graph using Adjacency matrix. |  |
| 8 | Implementation of Dijkstra’s Algorithm. |  |
| 9 | Implementation of Prim’s Algorithm. |  |
| 10 | Implementation of Kruskal’s Algorithm. |  |
| 11 | Create and perform various operations on Binary Search Tree. |  |
| 12 | Implementation of Heap. [Max & Min] |  |
| 13 | Implementing of Hashing Techniques. |  |

# Practical 1

### Implementation of Bubble Sorting Technique.

Description:-

Bubble Sort is comparison based sorting algorithm. In this algorithm adjacent elements are compared and swapped to make correct sequence. This algorithm is simpler than other algorithms, but it has some drawbacks also. This algorithm is not suitable for large number of data set. It takes much time to solve the sorting tasks.

Algorithm:-

begin BubbleSort(list) for all elements of list if list[i] > list[i+1] swap(list[i], list[i+1]) end if

end for return list

end BubbleSort Code:- #include<iostream> using namespace std;

void swapping(int &a, int &b) { int temp;

temp = a;

a = b;

b = temp;

}

void display(int \*array, int size) { for(int i = 0; i<size; i++)

cout << array[i] << " "; cout << endl;

}

void bubbleSort(int \*array, int size) { for(int i = 0; i<size; i++) {

int swaps = 0;

for(int j = 0; j<size-i-1; j++) { if(array[j] > array[j+1]) {

swapping(array[j], array[j+1]); swaps = 1;

}

}

if(!swaps) break;

}

}

int main() { int n;

cout << "Enter the number of elements: "; cin >> n;

int arr[n];

cout << "Enter elements:" << endl; for(int i = 0; i<n; i++) {

cin >> arr[i];

}

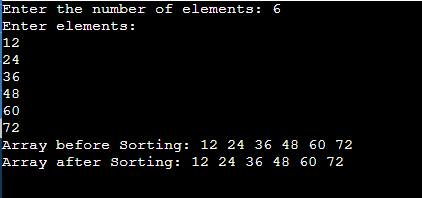
cout << "Array before Sorting: "; display(arr, n);

bubbleSort(arr, n);

cout << "Array after Sorting: "; display(arr, n);

}

Output:-



### Implementation of Merge Sorting Technique.

Description:-

The merge sort technique is based on divide and conquer technique. We divide the while data set into smaller parts and merge them into a larger piece in sorted order. It is also very effective for worst cases because this algorithm has lower time complexity for worst case also.

Algorithm:-

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

Code:-

#include <iostream> using namespace std;

void Merge(int array[],int begining,int midpoint,int endpoint){ int i,j,k;

int size1=midpoint-begining+1;//Size of an array 1 int size2=endpoint-midpoint;//Sie of an array 2

int Left\_Array[size1],Right\_Array[size2]; for(i=0;i<size1;i++){

Left\_Array[i]=array[begining+i];

}

for(j=0;j<size2;j++){ Right\_Array[j]=array[midpoint+1+j];

}

i=0; j=0;

k=begining;

while(i<size1 && j<size2){ if(Left\_Array[i]<=Right\_Array[j]){

array[k]=Left\_Array[i]; i++;

}

else{

array[k]=Right\_Array[j]; j++;

} k++;

}

while(i<size1){ array[k]=Left\_Array[i]; i++;

k++;

}

while(j<size2){ array[k]=Right\_Array[j]; j++;

k++;

}

}

void Merge\_Sort(int a[],int beg,int end){ if(beg<end){

int mid=(beg+end)/2; Merge\_Sort(a,beg,mid); Merge\_Sort(a,mid+1,end); Merge(a,beg,mid,end);

}

}

void Display(int a[],int size){ for(int i=0;i<size;i++){

cout<<a[i]<<" ";

}

}

int main()

{

cout<<"Sorted Array Using Merge\_Sort: \n"; int arr[8]={4,33,27,10,35,19,42,44};

int size = sizeof(arr)/sizeof(arr[0]); Merge\_Sort(arr,0,7); Display(arr,8);

return 0;}

Output:-

C:\Users\Admin\Desktop\merge sort.PNG

### Implementation of Insertion Sorting Technique.

Description:-

This sorting technique is similar with the card sorting technique, in other words we sort cards using insertion sort mechanism. For this technique, we pick up one element from the data set and shift the data elements to make a place to insert back the picked up element into the data set.

Algorithm:-

Step 1 − If it is the first element, it is already sorted. return 1; Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

Code:-

#include<iostream> using namespace std;

void display(int \*array, int size) { for(int i = 0; i<size; i++)

cout << array[i] << " "; cout << endl;

}

void insertionSort(int \*array, int size) { int key, j;

for(int i = 1; i<size; i++) { key = array[i];

j = i;

while(j > 0 && array[j-1]>key) { array[j] = array[j-1];

j--;

}

array[j] = key;

}

}

int main() { int n;

cout << "Enter the number of elements: "; cin >> n;

int arr[n];

cout << "Enter elements:" << endl; for(int i = 0; i<n; i++) {

cin >> arr[i];

}

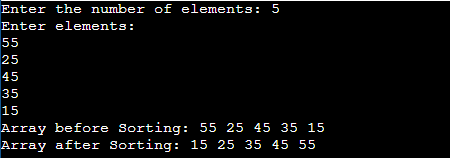
cout << "Array before Sorting: "; display(arr, n);

insertionSort(arr, n);

cout << "Array after Sorting: "; display(arr, n);

}

Output:-



### Implementation of Selection Sorting Technique.

Description:-

In the selection sort technique, the list is divided into two parts. In one part all elements are sorted and in another part the items are unsorted. At first we take the maximum or minimum data from the array. After getting the data (say minimum) we place it at the beginning of the list by replacing the data of first place with the minimum data. After performing the array is getting smaller. Thus this sorting technique is done.

Algorithm:-

Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element Step 5 − Repeat until list is sorted.

Code:- #include<iostream> using namespace std;

void swapping(int &a, int &b) { int temp;

temp = a; a = b;

b = temp;

}

void display(int \*array, int size) { for(int i = 0; i<size; i++)

cout << array[i] << " "; cout << endl;

}

void selectionSort(int \*array, int size) { int i, j, imin;

for(i = 0; i<size-1; i++) { imin = i;

for(j = i+1; j<size; j++) if(array[j] < array[imin])

imin = j;

swap(array[i], array[imin]);

}

}

int main() { int n;

cout << "Enter the number of elements: "; cin >> n;

int arr[n];

cout << "Enter elements:" << endl; for(int i = 0; i<n; i++) {

cin >> arr[i];

}

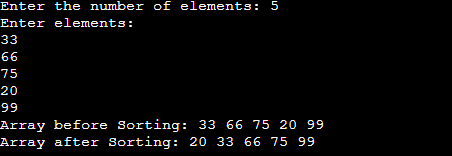
cout << "Array before Sorting: "; display(arr, n);

selectionSort(arr, n);

cout << "Array after Sorting: "; display(arr, n);

}

Output:-



### Implementation of Shell Sorting Technique.

Description:-

The shell sorting technique is based on the insertion sort. In the insertion sort sometimes we need to shift large block to insert item in the correct location. Using shell sort, we can avoid large number of shifting. The sorting is done with specific interval. After each pass the interval is reduced to make smaller interval.

Algorithm:-

Step 1 − Initialize the value of h

Step 2 − Divide the list into smaller sub-list of equal interval h Step 3 − Sort these sub-lists using insertion sort

Step 3 − Repeat until complete list is sorted

Code:- #include<iostream> using namespace std;

void ShellSort(int a[], int n)

{

int i, j, k, temp;

for(i = n/2; i > 0; i = i/2)

{

for(j = i; j < n; j++)

{

for(k = j-i; k >= 0; k = k-i)

{

if(a[k+i] >= a[k]) break;

else

{

temp = a[k]; a[k] = a[k+i]; a[k+i] = temp;

}

}

}

}

}

int main()

{

int n, i;

cout<<"\nEnter the number of data element to be sorted: "; cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": "; cin>>arr[i];

}

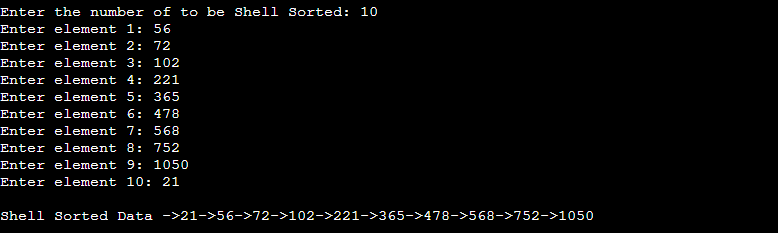
ShellSort(arr, n); cout<<"\nSorted Data "; for (i = 0; i < n; i++)

cout<<"->"<<arr[i];

return 0;

}

Output:-



### Implementation of Quick Sorting Technique.

Description:-

Quick sort is based on divide-and-conquer. The average time complexity of this algorithm is O(n\*log(n)) but the worst case complexity is O(n^2). To reduce the chances of the worst case here Quicksort is implemented using randomization.

Algorithm:-

Step 1 − Choose the highest index value has pivot

Step 2 − Take two variables to point left and right of the list excluding pivot Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right Step 8 − if left ≥ right, the point where they met is new pivot Code:-

#include<iostream> #include<stdlib> using namespace std;

void swap(int \*a, int \*b)

{

int temp; temp = \*a;

\*a = \*b;

\*b = temp;

}

int Partition(int a[], int low, int high)

{

int pivot, index, i; index = low; pivot = high;

for(i=low; i < high; i++)

{

if(a[i] < a[pivot])

{

swap(&a[i], &a[index]); index++;

}

}

swap(&a[pivot], &a[index]);

return index;

}

int RandomPivotPartition(int a[], int low, int high)

{

int pvt, n, temp; n = rand();

pvt = low + n%(high-low+1); swap(&a[high], &a[pvt]);

return Partition(a, low, high);

}

int QuickSort(int a[], int low, int high)

{

int pindex; if(low < high)

{

pindex = RandomPivotPartition(a, low, high);

QuickSort(a, low, pindex-1); QuickSort(a, pindex+1, high);

}

return 0;

}

int main()

{

int n, i;

cout<<"\nEnter the number of data element to be Quick Sorted: "; cin>>n;

int arr[n];

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

{

cout<<"Enter element "<<i+1<<": "; cin>>arr[i];

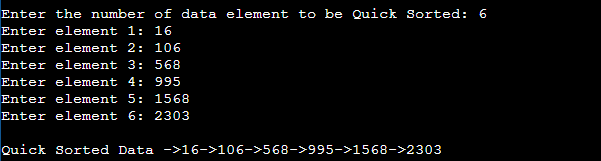
}

QuickSort(arr, 0, n-1); cout<<"\nQuick Sorted Data "; for (i = 0; i < n; i++)

cout<<"->"<<arr[i]; return 0;

}

Output:-



# Practical 2

### Implementation of Linear Search Algorithm.

Description:-

In linear search algorithm, we compare targeted element with each element of the array. If the element is found then its position is displayed. Linear search which is used to find whether a given number is present in an array and if it is present then at what location it occurs. It is also known as sequential search. It is straightforward and works as comparing each element with the element to search until it is found or the list ends.

Algorithm:-

Step 1: Set i to 1

Step 2: if i > n then go to step 7 Step 3: if A[i] = x then go to step 6 Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8 Step 7: Print element not found

Step 8: Exit

Code:-

#include <iostream> using namespace std;

int main()

{

int a[10]={10,20,30,40,5,6,33,65,-4,100},no,pos=-1;

cout<<"Enter Element to be searched"<<endl; cin>>no;

for(int i=0;i<10;i++){ if(a[i]==no){

pos=i; break;

}

}

if(pos!=-1){

cout<<"\n Element is found at "<<pos<<" Index";

}

else{

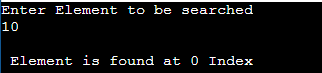
cout<<"\n Element is not found";

}

return 0;

}

Output:-



### Implementation of Binary Search Algorithm.

Description:-

Binary Search is a method to find the required element in a sorted array by repeatedly having the array and searching in the half.

This method is done by starting with the whole array. Then it is halved. If the required data value is greater than the element at the

middle of the array, then the upper half of the array is considered. Otherwise, the lower half is considered. This is done continuously until either the required data value is obtained or the remaining array is empty.

Algorithm:-

1. Compare x with the middle element.
2. If x matches with the middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half.
4. Else (x is smaller) recur for the left half.

Code:-

#include<iostream> using namespace std;

int binarySearch(int arr[], int p, int r, int num) { if (p <= r) {

int mid = (p + r)/2; if (arr[mid] == num)

return mid ;

if (arr[mid] > num)

return binarySearch(arr, p, mid-1, num); if (arr[mid] < num)

return binarySearch(arr, mid+1, r, num);

}

return -1;

}

int main(void) {

int arr[] = {1, 3, 7, 15, 18, 20, 25, 33, 36, 40};

int n = sizeof(arr)/ sizeof(arr[0]); int num;

cout << "Enter the number to search: \n"; cin >> num;

int index = binarySearch (arr, 0, n-1, num); if(index == -1){

cout<< num <<" is not present in the array";

}else{

cout<<"Number is present at index "<< index <<" in the array";

}

return 0;

}

Output:-

C:\Users\Admin\Desktop\binary search.PNG

# Practical 3

**Operations on Arrays:-** Traverse, Insert, Delete, Merge, Sort & Search.

Description:-

Traversal in an array means printing the elements of the array one by one. This means you are traversing through the elements of the array one after the other.

Insertion operation is used to insert an element into the array. Insertion of an element can happen at the beginning, middle or at the end of the array.

Merge sort keeps on dividing the list into equal halves until it can no more be divided. If it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Deletion operation is used to delete an element from the array.

A search operation is used to search for an element in the array. It can also be used to search for an element greater than or less than the specified element.

Sorting operations is used to sort array in asencding order or descesnding order.

Algorithm:- Traversal operation:- 1. C=1

2. Process LIST[C] 3. C= C+1

1. if (C<=N) then repeat 2 and 3
2. End.

Insertion operation:-

1. If(M<N) then BACK=M+1 else STOP
2. While (BACK>i) repeat steps 3 to 4
3. REG[BACK]= REG[BACK-1]
4. BACK=BACK-1;
5. Reg[BACK]='X'

* 6. M=M+1
* 7. End.

Deletion operation:-

1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop

Merge operations:-

MergeSort(A, p, r): if p > r

return

q = (p+r)/2 mergeSort(A, p, q) mergeSort(A, q+1, r) merge(A, p, q, r

Search operation:-

1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop

Code:-

#include <iostream> #include <bits/stdc++.h> using namespace std;

int binarysearch(int a[],int n,int key)

{

int left=0;

int right=n-1; while(left<=right)

{

int mid=(left+right)/2; if(a[mid]==key)

{

return mid;

}

if(a[mid]>key)

{

right=mid-1;

}

else

{

left=mid+1;

}

}

return -1;

}

int bubblesort(int a[],int n)

{

cout<<"\nSorted Array is : "; for(int i=0;i<n-1;i++)

{

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

{

if(a[j]>a[j+1])

{

int temp=a[j+1]; a[j+1]=a[j]; a[j]=temp;

}

}

}

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

{

cout<<a[i]<<" ";

}

return 0;

}

int traverseArray(int a[],int n)

{

cout<<"\nDisplaying Array Elements\t"; for(int i=0;i<n;i++)

{

cout<<a[i]<<" ";

}

return 0;

}

void mergeArray(int a[], int b[], int res[], int n1, int n2)

{

int k=0;

for(int i=0;i<n1;i++)

{

res[k]=a[i]; k++;

}

for(int j=0;j<n2;j++)

{

res[k]=b[j]; k++;

}

sort(res,res+n1+n2);

}

int main()

{

int ch,n1; while(1)

{

int a[1000];

cout<<"\n\nEnter Size of An Array\t"; cin>>n1;

cout<<"\nEnter Array Elements\n"; for(int i=0;i<n1;i++)

{

cin>>a[i];

}

cout<<"\n1 For Traversing\n2 For Sorting\n3 For Searching\n4 For Merging\t";

cin>>ch; switch(ch)

{

case 1:

traverseArray(a,n1); break;

case 2:

bubblesort(a,n1); break;

case 3:

int key,pos;

cout<<"\nEnter Element to be searched\t"; cin>>key;

pos = binarysearch(a,n1,key); if(pos==-1)

{

cout<<"\nElement is not found";

}

else

{

cout<<"\nElement is found at "<<pos<<" index";

}

break; case 4:

int n2,b[1000];

cout<<"\nEnter Size of An Array\t"; cin>>n2;

cout<<"\nEnter Array Elements\n"; for(int i=0;i<n2;i++)

{

cin>>b[i];

}

int res[n1 + n2]; mergeArray(a, b, res, n1, n2);

cout << "The sorted merged array is: "; for (int i = 0; i < n1 + n2; i++)

cout << " " << res[i]; cout << "\n";

break;

}

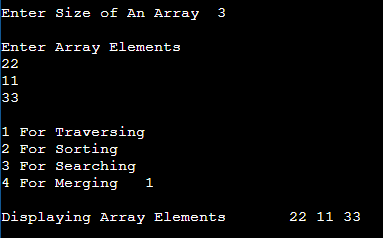
}

return 0;

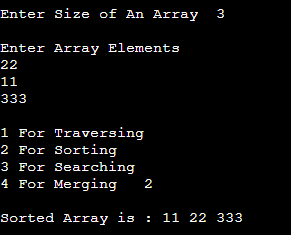
}

Output:-

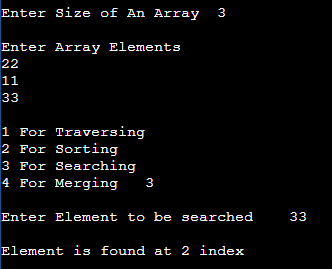
For Traversing:-



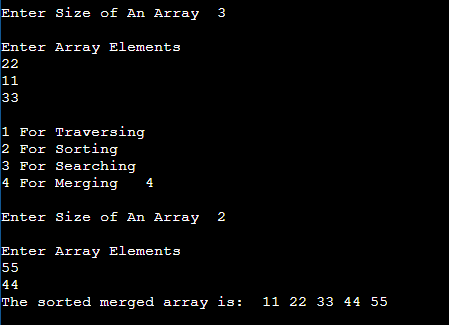
For Sorting:-



For Searching:-



For Merging:-



**Practical 4**

**Implementation of Linked Lists [Singly Linked List]**:-Traverse, Insert, Delete, Search.

Description:-

There are various linked list operations that allow us to perform different actions on linked lists. For example, the insertion operation adds a new element to the linked list.

Here's a list of basic linked list operations that we will cover in this article.

* [Traversal](https://www.programiz.com/dsa/linked-list-operations#traverse) - access each element of the linked list
* [Insertion](https://www.programiz.com/dsa/linked-list-operations#add) - adds a new element to the linked list
* [Deletion](https://www.programiz.com/dsa/linked-list-operations#delete) - removes the existing elements
* [Search](https://www.programiz.com/dsa/linked-list-operations#search) - find a node in the linked list

Algorithm:-

#### Traverse a Linked List

Step 1: [INITIALIZE] SET PTR = HEAD

Step 2: Repeat Steps 3 and 4 while PTR != NULL Step 3: Apply process to PTR -> DATA

Step 4: SET PTR = PTR->NEXT [END OF LOOP]

Step 5: EXIT)

#### Insert Elements to a Linked List

You can add elements to either the beginning, middle or end of the linked list.

#### Insert at the beginning

* + Allocate memory for new node
  + Store data
  + Change next of new node to point to head
  + Change head to point to recently created node

#### Insert at the End

* + Allocate memory for new node
  + Store data
  + Traverse to last node
  + Change next of last node to recently created node

#### Insert at the Middle

* + Allocate memory and store data for new node
  + Traverse to node just before the required position of new node
  + Change next pointers to include new node in between

#### Delete from a Linked List

1. **Delete from beginning**
   * Point head to the second node

#### Delete from end

* + Traverse to second last element
  + Change its next pointer to null

#### Delete from middle

* + Traverse to element before the element to be deleted
  + Change next pointers to exclude the node from the chain

#### Search an Element on a Linked List

You can search an element on a linked list using a loop using the following steps. We are finding item on a linked list.

* + Make head as the current node.
  + Run a loop until the current node is NULL because the last element points to NULL.
  + In each iteration, check if the key of the node is equal to item. If it the key matches the item, return true otherwise return false.

Code:-

#include <iostream> using namespace std; class Node

{

public:

int info; Node \*next;

};

class List: public Node

{

Node \*first,\*last; public:

List()

{

first = NULL; last=NULL;

}

void Create()

{

Node \*temp; temp = new Node; int no;

cout<<"\nEnter an element \n"; cin>>no;

temp->info = no; temp->next=NULL; if(first==NULL)

{

}

else

{

}

}

first=temp; last=first;

last->next=temp; last=temp;

void Insert()

{

int ch; int no;

cout<<"\n1 For Insert at Begining\n2 For Insert at position\n"; cin>>ch;

Node \*prev,\*cur,\*temp; switch(ch)

{

case 1:

prev=first; cur=new Node;

cout<<"\nEnter an Element\n"; cin>>no;

cur->info=no;

cur->next=NULL; if(prev==NULL)

{

}

else

{

}

break; case 2:

first=cur;

cur->next=first; first=cur;

int index;

cout<<"\nEnter an Element\n"; cin>>no;

cout<<"Enter Index Position "; cin>>index;

cur=new Node; cur->info=no;

cur->next=NULL; if(index<1)

{

cout<<"\nposition should be >= 1";

}

else if(index==1)

{

}

else

{

cur->next=first; first=cur;

temp=first;

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

{

if(temp!=NULL)

{

temp=temp->next;

}

}

if(temp!=NULL)

{

}

break;

}

}

}

else

{

}

cur->next=temp->next; temp->next=cur;

cout<<"\nThe previous node is null.";

void Delete()

{

if(first==NULL)

{

}

else

{

cout<<"\nElement is not exist"; return ;

Node \*temp=first; Node \*prev;

int ch;

cout<<"\n1 For First\n2 For Last\n3 For Position\n"; cin>>ch;

switch(ch)

{

case 1:

first=first->next; break;

case 2:

while(temp!=last)

{

prev=temp; temp=temp->next;

}

if(temp==last)

{

prev->next=NULL;

last=prev;

}

break; case 3:

int pos,count=1; cout<<"\nEnter position\n"; cin>>pos; while(count!=pos)

{

prev=temp; temp=temp->next; count++;

}

if(count==pos)

{

prev->next=temp->next;

}

break;

}

}

}

int Search(int element)

{

int position=-1,flag=0; if(first!=NULL)

{

Node \*temp;

temp=first; while(temp!=NULL)

{

position++;

if(temp->info==element)

{

flag =1; break;

}

temp=temp->next;

}

}

if(flag!=0)

{

}

else

{

}

}

return position;

return position;

void Display()

{

Node \*temp=first; if(first==NULL)

{

cout<<"\nList is Empty";

}

else

{

while(temp!=NULL)

{

cout<<temp->info; cout<<"-->"; temp=temp->next;

}

cout<<"NULL";

}

}

};

int main()

{

List l; int ch; while(1)

{

cout<<"\n1 For Creation of Node\n2 For Display List\n3 For Insert at begining\n4 For Deletion of element\n5 For Search\n";

cin>>ch; switch(ch)

{

case 1:

l.Create(); break;

case 2:

l.Display(); break;

case 3:

l.Insert(); break;

case 4:

l.Delete(); break;

case 5:

if(l.Search(50)==-1)

{

cout<<"\nList is empty";

}

else if(l.Search(50)==0)

{

}

else

{

}

break; default:

cout<<"\nElement not found";

cout<<"\nElement found at "<<l.Search(50);

cout<<"\nWrong Choice\n"; break;

}

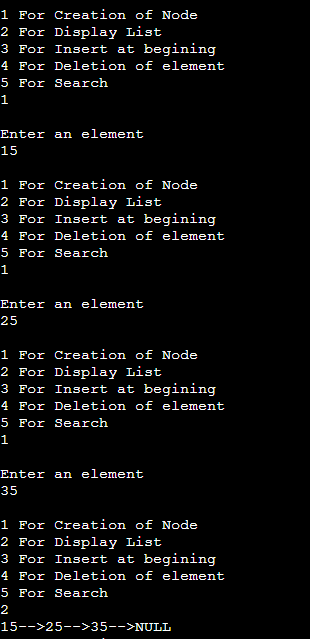
}

return 0;

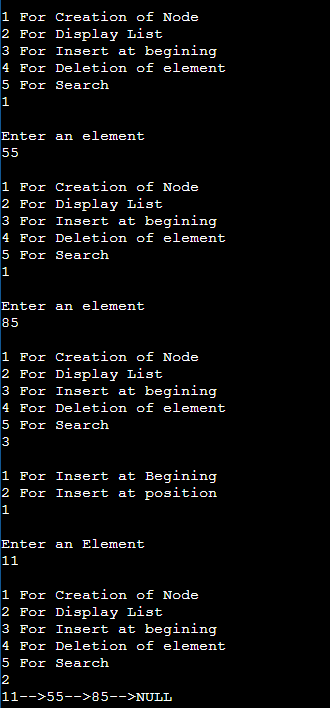
}

Output:-

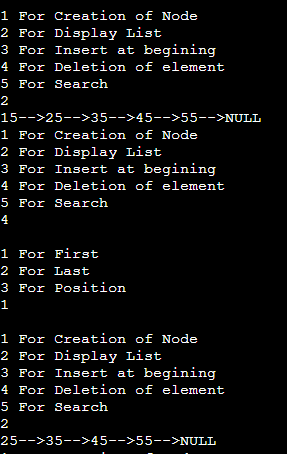
#### For Traversing



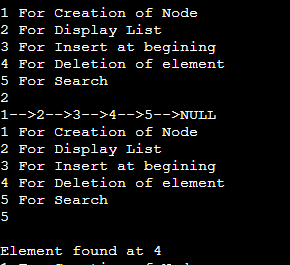
**For Inserting**



#### For Deleting



**For Searching**



## Practical 5

**Implementation of Stack Using Array.**[push, pop, display] Description:-

A stack is an abstract data structure that contains a collection of elements. Stack implements the LIFO mechanism i.e. the element that is pushed at the end is popped out first. Some of the principle operations in the stack are −

* + Push - This adds a data value to the top of the stack.
  + Pop - This removes the data value on top of the stack
  + Display - This returns the top data value of the stack

Algorithm:- For Push:- Step 1: Start

Step 2: Declare Stack[MAX]; //Maximum size of Stack

Step 3: Check if the stack is full or not by comparing top with (MAX-1) If the stack is full, Then print "Stack Overflow" i.e, stack is full and cannot be pushed with another element

Step 4: Else, the stack is not full Increment top by 1 and Set, a[top] = x

which pushes the element x into the address pointed by top.

// The element x is stored in a[top] Step 5: Stop

For Pop:- Step 1: Start

Step 2: Declare Stack[MAX]

Step 3: Push the elements into the stack

Step 4: Check if the stack is empty or not by comparing top with base of array

i.e 0

If top is less than 0, then stack is empty, print "Stack Underflow"

Step 5: Else, If top is greater than zero the stack is not empty, then store the value. Pointed by top in a variable x=a[top] and decrement top by 1.

The popped element is x.

Step 6 - If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top--).

For Display:- Step 1: Start

Step 2: Declare Stack[MAX]

Step 3: Push the elements into the stack

Step 4: Print the value stored in the stack pointed by top. Step 6: Stop

Code:-

#include <iostream> #include<conio.h> #include<stdlib.h> int top=-1;

using namespace std; class Stack{

int arr[1000];

int max; public:

Stack(int a){ max=a;

}

void Push(int x){ if(top==(max-1)){

cout<<"\nStack is full";

}

else{

top++; arr[top]=x;

}

}

void Pop(){ if(top==-1){

cout<<"\nStack is Empty";

}

else{

top--;

}

}

void Display(){ if(top==-1){

cout<<"\nStack is Empty";

}else{

for(int i=0;i<=top;i++){ cout<<arr[i]<<"-->";

}

}

}

};

int main()

{

int max,ch,no;

cout<<"\nEnter Size of An Array"; cin>>max;

Stack s(max); while(1){

cout<<"\n 1 For Push\n2 For Pop\n3 For Display\n"; cin>>ch;

switch(ch){ case 1:

cout<<"\nEnter Element To Insert "; cin>>no;

s.Push(no);

break; case 2:

s.Pop();

s.Display(); break;

case 3:

s.Display(); break;

}

}

return 0;

}

Output:-



**Implementation of Stack Using Linked List.**[push, pop, display] Description:-

A stack is an abstract data structure that contains a collection of elements. Stack implements the LIFO mechanism i.e. the element that is pushed at the end is popped out first. Some of the principle operations in the stack are −

* Push - This adds a data value to the top of the stack.
* Pop - This removes the data value on top of the stack.
* Display - This returns the top data value of the stack.

Algorithm:- For Push:- Step 1: Start

Step 2: Create a node new and declare variable top Step 3: Set new data part to be Null

// The first node is created, having null value and top pointing to it Step 4: Read the node to be inserted.

Step 5: Check if the node is Null, then print "Insufficient Memory"

Step 6: If node is not Null, assign the item to data part of new and assign top to link part of

new and also point stack head to new.

For Pop:- Step 1: Start

Step 2: Check if the top is Null, then print "Stack Underflow."

Step 3: If top is not Null, assign the top's link part to ptr and assign ptr to stack\_head's

link part.

Step 4: Stop

For Display:-

Step 1 - Check whether stack is Empty (top == NULL).

Step 2 - If it is Empty, then display 'Stack is Empty!!!' and terminate the function.

Step 3 - If it is Not Empty, then define a Node pointer 'temp' and initialize with top.

Step 4 - Display 'temp → data --->' and move it to the next node.

Repeat the same

until temp reaches to the first node in the stack. (temp → next != NULL).

Step 5 - Finally! Display 'temp → data ---> NULL'. Code:-

#include<iostream> using namespace std; class Node

{

public:

Node\*head; int data; Node\*next; Node(int val)

{

data=val; next=NULL;

}

};

void push(Node\* &head,int val){ Node\* n =new Node(val); if(head==NULL)

{

head=n; return;

}

Node \*temp=head; while(temp->next!=NULL){ temp=temp->next;

}

temp->next=n;

}

void pop(Node\* &head){ if(head==NULL){ cout<<"Stack Is Empty "; return;

}

Node \*temp=head;

while(temp->next->next!=NULL){ temp=temp->next;

}

temp->next=NULL;

}

void display(Node\* &head){ Node \*temp=head;

while(temp!=NULL){ cout<<"\n"<<temp->data; temp=temp->next;

}

}

int main(){

int ch,element; Node\* head=NULL; while(1){

cout<<"\n1 For Push\n2 For Pop\n3 For Display\n"; cin>>ch;

switch(ch){ case 1:

cout<<"\nEnter Element"; cin>>element; push(head,element); break;

case 2:

pop(head); break;

case 3:

display(head); break;

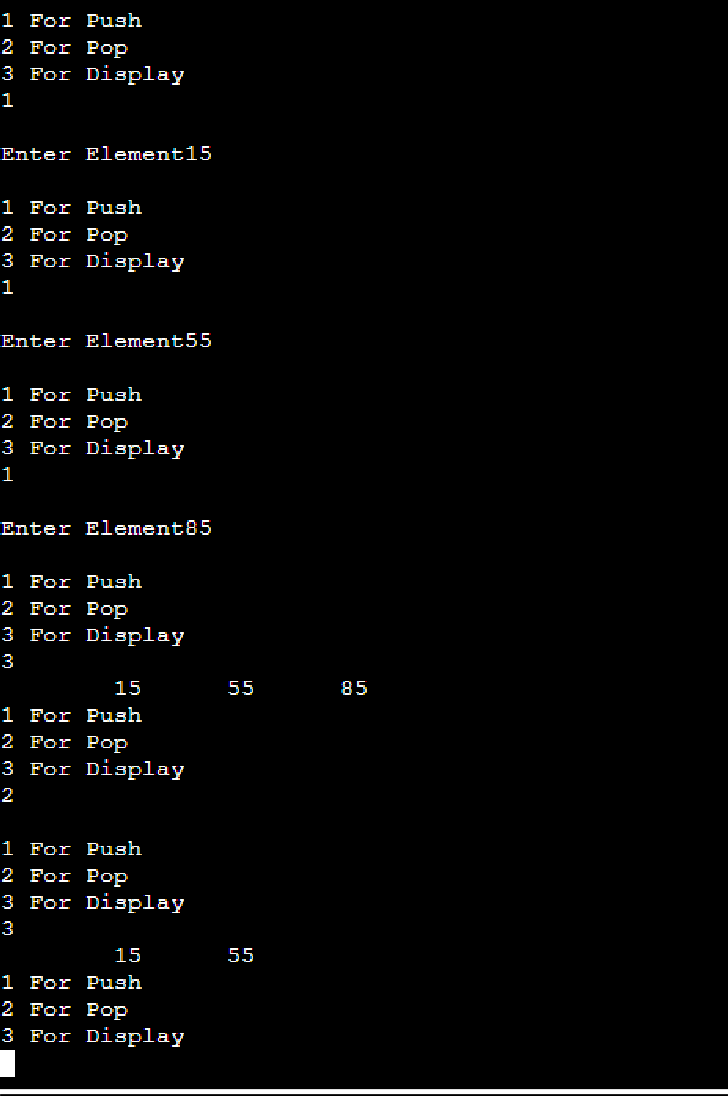
}

}

return 0;

}

Output:-



# Practical 6

**Implementation of Queue Using Array.**[Enqueue & Dequeue] Description:-

A queue is an abstract data structure that contains a collection of elements. Queue implements the FIFO mechanism i.e. the element that is inserted first is also deleted first. In other words, the least recently added element is removed first in a queue.

Algorithm:-

Enqueue:-

**Step 1** − Check if the queue is full.

**Step 2** − If the queue is full, produce overflow error and exit.

**Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.

**Step 4** − Add data element to the queue location, where the rear is pointing.

**Step 5** − return success.

Dequeue:-

**Step 1** − Check if the queue is empty.

**Step 2** − If the queue is empty, produce underflow error and exit.

**Step 3** − If the queue is not empty, access the data where **front** is pointing. **Step 4** − Increment **front** pointer to point to the next available data element. **Step 5** − Return success.

Code:-

#include <iostream> #include<conio.h> #include<stdlib.h> using namespace std;

int queue[100], n, first = - 1, last = - 1; void Insert() {

int val;

if (last == n - 1)

cout<<"Queue Overflow"<<endl; else {

if (first == - 1) first = 0;

cout<<"Insert the element in queue : "<<endl; cin>>val;

last++; queue[last] = val;

}

}

void Delete() {

if (first == - 1 || first > last) { cout<<"Queue Underflow "; return ;

}

else {

cout<<"Element deleted from queue is : "<< queue[first] <<endl; first++;;

}

}

void Display() { if (first == - 1)

cout<<"Queue is empty"<<endl; else

{

cout<<"Queue elements are : "; for (int i = first; i <= last; i++) cout<<queue[i]<<" "; cout<<endl;

}

}

int main() { int ch;

cout<<"\nEnter Size of Array "; cin>>n;

while(1){

cout<<"1) Insert element to queue"<<endl;

cout<<"2) Delete element from queue"<<endl; cout<<"3) Display all the elements of queue"<<endl; cout<<"Enter your choice : "<<endl;

cin>>ch; switch (ch) { case 1: Insert(); break;

case 2: Delete(); break;

case 3: Display(); break;

default: cout<<"Invalid choice"<<endl;

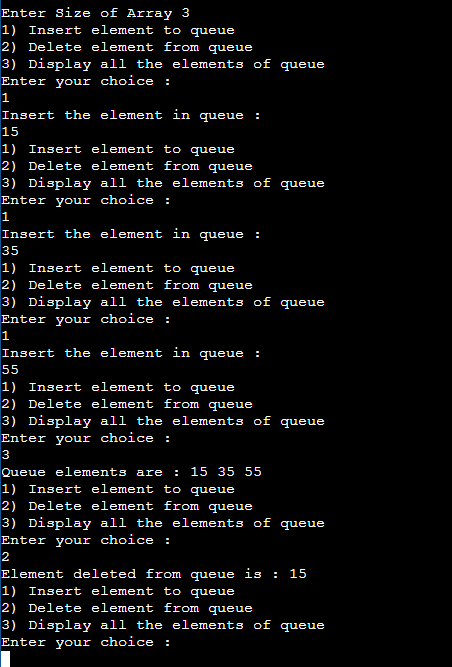
}

}

return 0;

}

Output:-



**Implementation of Queue Using Linked List.**[Enqueue & Dequeue] Description:-

A queue is an abstract data structure that contains a collection of elements. Queue implements the FIFO mechanism i.e the element that is inserted first is also deleted first. In other words, the least recently added element is removed first in a queue.

Algorithm:-

Enqueue

newNode -> data = data newNode -> next = NULL if ( REAR == NULL)

FRONT = REAR = newNode end ifelse

REAR -> next = newNode REAR = newNode

end Algorithm\_Enqueue

Dequeue

if(FRONT == NULL)

print "EMPTY QUEUE" and exit. end ifend Algorithm\_Dequeue else

temp = FRONT

FRONT = FRONT -> NEXT

free(temp)

end elseend Algorithm\_Dequeue

Code:-

#include <iostream>

using namespace std;

struct Node{ int data;

struct Node \*next;

}\*front=NULL,\*rear,\*temp;

void Insert(){ temp=new Node;

cout<<"\nEnter Data\n"; cin>>temp->data;

temp->next=NULL;

if(front==NULL){ front=rear=temp;

}else{

rear->next=temp; rear=temp;

}

}

void Delete(){ if(front==NULL){

cout<<"\nQueue is empty\n";

}else{

temp=front; front=front->next;

cout<<"\nElement Deleted From Queue is "<<temp->data; delete(temp);

}

}

void Display(){ if(front==NULL){

cout<<"\nQueue is empty\n";

}else{

temp=front; while(temp!=NULL){

cout<<temp->data<<"-->"; temp=temp->next;

}

}

}

int main()

{

int ch; while(1){

cout<<"\n1 For Insert\n2 For Delete\n3 For Display\n"; cin>>ch;

cout<<"\n"; switch(ch){ case 1:

Insert(); break;

case 2:

Delete(); break;

case 3:

Display(); break;

default:

cout<<"\nWrong Choice"; break;

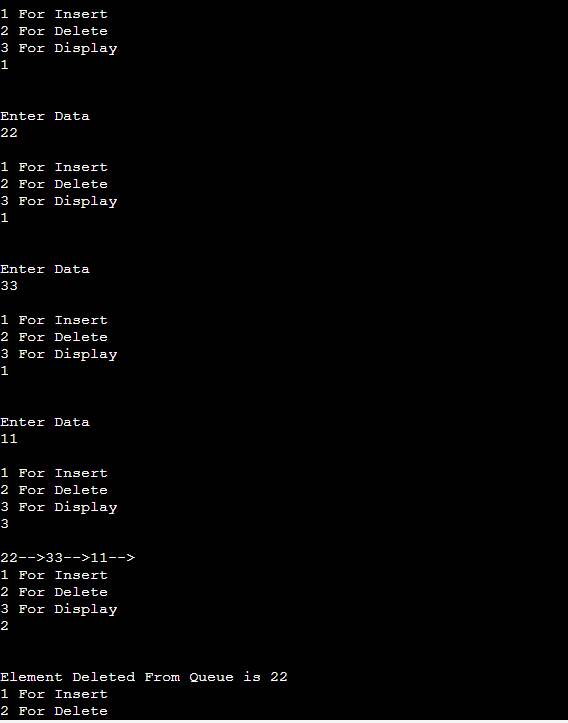
}

}

return 0;

}

Output:-



# Practical 7

### Implementation of Graph using Adjacency Matrix.

Description:-

An adjacency matrix is a way of representing a graph as a matrix of booleans (0's and 1's). A finite graph can be represented in the form of a square matrix on a computer, where the boolean value of the matrix indicates if there is a direct path between two vertices.

Algorithm:-

Begin

adj\_matrix[u, v] := 1 adj\_matrix[v, u] := 1 End

Code:-

#include <iostream> using namespace std; class Graph

{

bool \*\*adjMatrix; int vertices;

public:

Graph (int NoVertex)

{

vertices = NoVertex;

adjMatrix = new bool \*[vertices];

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

{

adjMatrix[i] = new bool[vertices]; for (int j = 0; j < vertices; j++)

{

adjMatrix[i][j] = false;

}

}

}

void addEdge (int i, int j)

{

adjMatrix[i][j] = true; adjMatrix[j][i] = true;

}

void removeEdge (int i, int j)

{

adjMatrix[i][j] = false; adjMatrix[j][i] = false;

}

void display ()

{

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

{

cout << i << "---> ";

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

{

cout << adjMatrix[i][j] << " ";

}

cout << endl;

}

}

};

int

main ()

{

Graph g1 (3);

g1.addEdge (0, 2);

g1.addEdge (2, 0);

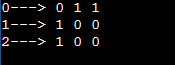
g1.addEdge (1, 0);

g1.addEdge (0, 1);

g1.display (); return 0;

}

Output:-



# Practical 8

### Implementation of Dijkstra’s Algorithm.

Description:-

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

Algorithm:-

1. Create a set sptSet (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
2. Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3. While sptSet doesn’t include all vertices

….a) Pick a vertex u which is not there in sptSet and has a minimum distance value.

….b) Include u to sptSet.

….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v,

then update the distance value of v.

Code:-

#include <iostream> #include <vector>

#define INT\_MAX 10000000 using namespace std;

void DijkstrasTest();

int main() { DijkstrasTest(); return 0;

}

class Node; class Edge;

void Dijkstras();

vector<Node\*>\* AdjacentRemainingNodes(Node\* node); Node\* ExtractSmallest(vector<Node\*>& nodes);

int Distance(Node\* node1, Node\* node2);

bool Contains(vector<Node\*>& nodes, Node\* node); void PrintShortestRouteTo(Node\* destination);

vector<Node\*> nodes;

vector<Edge\*> edges;

class Node { public:

Node(char id)

: id(id), previous(NULL), distanceFromStart(INT\_MAX) { nodes.push\_back(this);

}

public:

char id;

Node\* previous;

int distanceFromStart;

};

class Edge { public:

Edge(Node\* node1, Node\* node2, int distance)

: node1(node1), node2(node2), distance(distance) { edges.push\_back(this);

}

bool Connects(Node\* node1, Node\* node2) { return (

(node1 == this->node1 &&

node2 == this->node2) || (node1 == this->node2 && node2 == this->node1));

}

public:

Node\* node1; Node\* node2; int distance;

};

void DijkstrasTest() { Node\* a = new Node('a'); Node\* b = new Node('b'); Node\* c = new Node('c'); Node\* d = new Node('d'); Node\* e = new Node('e'); Node\* f = new Node('f'); Node\* g = new Node('g');

Edge\* e1 = new Edge(a, c, 1); Edge\* e2 = new Edge(a, d, 2); Edge\* e3 = new Edge(b, c, 2); Edge\* e4 = new Edge(c, d, 1);

Edge\* e5 = new Edge(b, f, 3); Edge\* e6 = new Edge(c, e, 3); Edge\* e7 = new Edge(e, f, 2); Edge\* e8 = new Edge(d, g, 1); Edge\* e9 = new Edge(g, f, 1);

a->distanceFromStart = 0; // set start node Dijkstras();

PrintShortestRouteTo(f);

}

void Dijkstras() {

while (nodes.size() > 0) {

Node\* smallest = ExtractSmallest(nodes); vector<Node\*>\* adjacentNodes = AdjacentRemainingNodes(smallest);

const int size = adjacentNodes->size(); for (int i = 0; i < size; ++i) {

Node\* adjacent = adjacentNodes->at(i);

int distance = Distance(smallest, adjacent) + smallest->distanceFromStart;

if (distance < adjacent->distanceFromStart) { adjacent->distanceFromStart = distance; adjacent->previous = smallest;

}

}

delete adjacentNodes;

}

}

Node\* ExtractSmallest(vector<Node\*>& nodes) { int size = nodes.size();

if (size == 0) return NULL; int smallestPosition = 0; Node\* smallest = nodes.at(0); for (int i = 1; i < size; ++i) { Node\* current = nodes.at(i);

if (current->distanceFromStart < smallest->distanceFromStart) { smallest = current; smallestPosition = i;

}

}

nodes.erase(nodes.begin() + smallestPosition);

return smallest;

}

vector<Node\*>\* AdjacentRemainingNodes(Node\* node) { vector<Node\*>\* adjacentNodes = new vector<Node\*>(); const int size = edges.size();

for (int i = 0; i < size; ++i) { Edge\* edge = edges.at(i); Node\* adjacent = NULL; if (edge->node1 == node) { adjacent = edge->node2;

} else if (edge->node2 == node) { adjacent = edge->node1;

}

if (adjacent && Contains(nodes, adjacent)) { adjacentNodes->push\_back(adjacent);

}

}

return adjacentNodes;

}

int Distance(Node\* node1, Node\* node2) { const int size = edges.size();

for (int i = 0; i < size; ++i) { Edge\* edge = edges.at(i);

if (edge->Connects(node1, node2)) { return edge->distance;

}

}

return -1; // should never happen

}

bool Contains(vector<Node\*>& nodes, Node\* node) { const int size = nodes.size();

for (int i = 0; i < size; ++i) { if (node == nodes.at(i)) { return true;

}

}

return false;

}

void PrintShortestRouteTo(Node\* destination) { Node\* previous = destination;

cout << "Distance from start: "

<< destination->distanceFromStart << endl;

while (previous) {

cout << previous->id << " "; previous = previous->previous;

}

cout << endl;

}

vector<Edge\*>\* AdjacentEdges(vector<Edge\*>& Edges, Node\* node); void RemoveEdge(vector<Edge\*>& Edges, Edge\* edge);

vector<Edge\*>\* AdjacentEdges(vector<Edge\*>& edges, Node\* node) { vector<Edge\*>\* adjacentEdges = new vector<Edge\*>();

const int size = edges.size(); for (int i = 0; i < size; ++i) { Edge\* edge = edges.at(i);

if (edge->node1 == node) {

cout << "adjacent: " << edge->node2->id << endl; adjacentEdges->push\_back(edge);

} else if (edge->node2 == node) {

cout << "adjacent: " << edge->node1->id << endl; adjacentEdges->push\_back(edge);

}

}

return adjacentEdges;

}

void RemoveEdge(vector<Edge\*>& edges, Edge\* edge) { vector<Edge\*>::iterator it;

for (it = edges.begin(); it < edges.end(); ++it) { if (\*it == edge) {

edges.erase(it); return;

}

}

}

Output:-

C:\Users\Admin\Desktop\dijkstra's.PNG

# Practical 9

### Implementation of Prim’s Algorithm.

Description:-

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

* form a tree that includes every vertex
* has the minimum sum of weights among all the trees that can be formed from the graph

Algorithm:-

* 1. Initialize the minimum spanning tree with a vertex chosen at random.
  2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
  3. Keep repeating step 2 until we get a minimum spanning tree

Code:-

#include <cstring> #include <iostream> using namespace std;

#define INF 9999999

// number of vertices in graph #define V 5

// create a 2d array of size 5x5

//for adjacency matrix to represent graph

int G[V][V] = {

{0, 9, 75, 0, 0},

{9, 0, 95, 19, 42},

{75, 95, 0, 51, 66},

{0, 19, 51, 0, 31},

{0, 42, 66, 31, 0}};

int main() {

int no\_edge; // number of edge

// create a array to track selected vertex

// selected will become true otherwise false int selected[V];

// set selected false initially memset(selected, false, sizeof(selected));

// set number of edge to 0 no\_edge = 0;

// the number of edge in minimum spanning tree will be

// always less than (V -1), where V is number of vertices in

//graph

// choose 0th vertex and make it true selected[0] = true;

int x; // row number int y; // col number

// print for edge and weight cout << "Edge"

<< " : "

<< "Weight"; cout << endl;

while (no\_edge < V - 1) { int min = INF;

x = 0;

y = 0;

for (int i = 0; i < V; i++) { if (selected[i]) {

for (int j = 0; j < V; j++) {

if (!selected[j] && G[i][j]) { // not in selected and there is an edge if (min > G[i][j]) {

min = G[i][j]; x = i;

y = j;

}

}

}

}

}

cout << x << " - " << y << " : " << G[x][y];

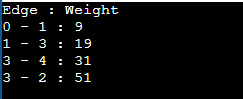
cout << endl; selected[y] = true; no\_edge++;

}

return 0;

}

Output:-



# Practical 10

### Implementation of Krushkal’s Algorithm.

Description:-

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which:-

* form a tree that includes every vertex
* has the minimum sum of weights among all the trees that can be formed from the graph

Algorithm:-

1. Sort all the edges from low weight to high
2. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
3. Keep adding edges until we reach all vertices.

Code:-

#include <algorithm> #include <iostream> #include <vector> using namespace std;

#define edge pair<int, int> class Graph {

private:

vector<pair<int, edge> > G; // graph vector<pair<int, edge> > T; // mst int \*parent;

int V; // number of vertices/nodes in graph public:

Graph(int V);

void AddWeightedEdge(int u, int v, int w); int find\_set(int i);

void union\_set(int u, int v); void kruskal();

void print();

};

Graph::Graph(int V) { parent = new int[V];

//i 0 1 2 3 4 5

//parent[i] 0 1 2 3 4 5 for (int i = 0; i < V; i++) parent[i] = i;

G.clear();

T.clear();

}

void Graph::AddWeightedEdge(int u, int v, int w) { G.push\_back(make\_pair(w, edge(u, v)));

}

int Graph::find\_set(int i) {

// If i is the parent of itself if (i == parent[i])

return i; else

// Else if i is not the parent of itself

// Then i is not the representative of his set,

// so we recursively call Find on its parent return find\_set(parent[i]);

}

void Graph::union\_set(int u, int v) { parent[u] = parent[v];

}

void Graph::kruskal() { int i, uRep, vRep;

sort(G.begin(), G.end()); // increasing weight for (i = 0; i < G.size(); i++) {

uRep = find\_set(G[i].second.first); vRep = find\_set(G[i].second.second); if (uRep != vRep) { T.push\_back(G[i]); // add to tree union\_set(uRep, vRep);

}

}

}

void Graph::print() { cout << "Edge :"

<< " Weight" << endl;

for (int i = 0; i < T.size(); i++) {

cout << T[i].second.first << " - " << T[i].second.second << " : "

<< T[i].first; cout << endl;

}

}

int main() { Graph g(6);

g.AddWeightedEdge(0, 1, 4);

g.AddWeightedEdge(0, 2, 4);

g.AddWeightedEdge(1, 2, 2);

g.AddWeightedEdge(1, 0, 4);

g.AddWeightedEdge(2, 0, 4);

g.AddWeightedEdge(2, 1, 2);

g.AddWeightedEdge(2, 3, 3);

g.AddWeightedEdge(2, 5, 2);

g.AddWeightedEdge(2, 4, 4);

g.AddWeightedEdge(3, 2, 3);

g.AddWeightedEdge(3, 4, 3);

g.AddWeightedEdge(4, 2, 4);

g.AddWeightedEdge(4, 3, 3);

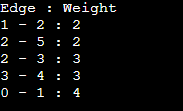
g.AddWeightedEdge(5, 2, 2);

g.AddWeightedEdge(5, 4, 3); g.kruskal();

g.print(); return 0;

}

Output:-



# Practical 11

### Implementation of various operation on Binary search tree.

Description:-

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

* It is called a binary tree because each tree node has a maximum of two children.
* It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

Algorithm:-

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties

Code:-

#include <iostream> using namespace std; struct tree\_node{ tree\_node \*left; tree\_node \*right;

int data;

};

class BST{ tree\_node \*root;

public:

BST(){

root=NULL;

}

int isEmpty(){

return (root==NULL);

}

void insert(){ int item;

cout<<"\nEnter Item To Insert"; cin>>item;

tree\_node \*p=new tree\_node; tree\_node \*parent;

p->data=item; p->left=NULL;

p->right=NULL; parent=NULL;

if(isEmpty()){

root=p;

}

else{ tree\_node \*ptr; ptr=root;

while(ptr!=NULL){ parent=ptr; if(item>ptr->data){ ptr=ptr->right;

}

else{

ptr=ptr->left;

}

}

if(item<parent->data){ parent->left=p;

}

else{

parent->right=p;

}

}

}

void inOrderTraversal(){ inorder(root);

}

void inorder(tree\_node \*ptr){ if(ptr!=NULL){

inorder(ptr->left);

cout<<" "<<ptr->data<<" "; inorder(ptr->right);

}

}

void preOrderTraversal(){ preorder(root);

}

void preorder(tree\_node \*ptr){ if(ptr!=NULL){

cout<<" "<<ptr->data<<" "; inorder(ptr->left); inorder(ptr->right);

}

}

void postOrderTraversal(){ postorder(root);

}

void postorder(tree\_node \*ptr){ if(ptr!=NULL){

inorder(ptr->left); inorder(ptr->right); cout<<" "<<ptr->data<<" ";

}

}

};

int main()

{

int ch; BST bst; while(1){

cout<<"\n1 For Insert \n2 For In-Order Traversal\n3 For Pre-Order Traversal\n4 For Post-Order Traversal\n";

cin>>ch; switch(ch){ case 1:

bst.insert(); break;

case 2: bst.inOrderTraversal(); break;

case 3: bst.preOrderTraversal(); break;

case 4: bst.postOrderTraversal(); break;

}

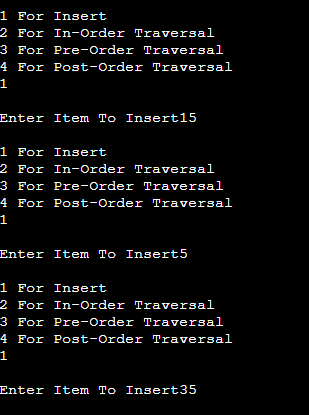
}

return 0;

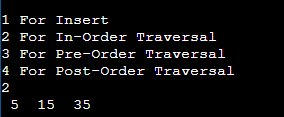
}

Output:-

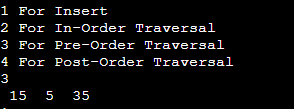
Inserting elements



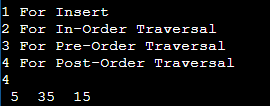
In-Order Traversal



Pre-Order Traversal



Post-Order Traversal



# Practical 12

### Implementation of Max Heap.

Description:-

A Binary Heap is a complete binary tree which is either Min Heap or Max Heap. In a Max Binary Heap, the key at root must be maximum among all keys present in Binary Heap. This property must be recursively true for all nodes in Binary Tree.

Algorithm:- Max Heap:- Begin

Declare function max\_heap () Declare j, t of the integer datatype. Initialize t = a[m].

j = 2 \* m;

while (j <= n) do

if (j < n && a[j+1] > a[j]) then j = j + 1

if (t > a[j]) then break

else if (t <= a[j]) then a[j / 2] = a[j]

j = 2 \* j a[j/2] = t return

End.

For build\_maxheap:

Begin

Declare function build\_maxheap(int \*a,int n).

Declare k of the integer datatype. for(k = n/2; k >= 1; k--)

Call function max\_heap(a,k,n)

End.

Code:-

#include <iostream> #include <conio.h> using namespace std;

void make\_heap(int \*a, int i, int n)

{

int j, temp; temp = a[i]; j = 2 \* i;

while (j <= n)

{

if (j < n && a[j+1] > a[j]) j = j + 1;

if (temp > a[j]) break;

else if (temp <= a[j])

{

a[j / 2] = a[j]; j = 2 \* j;

}

}

a[j/2] = temp; return;

}

void build\_heap(int \*a,int n)

{

int i;

for(i = n/2; i >= 1; i--)

{

make\_heap(a,i,n);

}

}

int main()

{

int n,i,a[100];

cout<<"Enter Array Size\n"; cin>>n;

for(i=1;i<=n;i++){ cout<<"\nEnter Array Element\n"; cin>>a[i];

}

build\_heap(a,n);

cout<<"\nMax Heap is Implemented\n"; for(i=1;i<=n;i++){

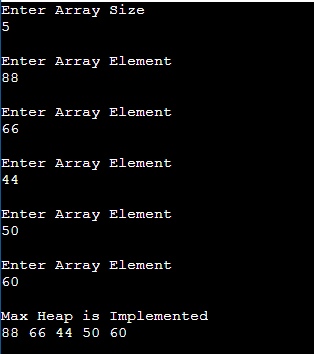
cout<<a[i]<<" ";

}

return 0;

}

Output:-



### Implementation of Min Heap.

Description:-

A Binary Heap is a complete binary tree which is either Min Heap or Max Heap. In a Max Binary Heap, the key at root must be maximum among all keys present in Binary Heap. This property must be recursively true for all nodes in Binary Tree.

Algorithm:- Min Heap:- Begin

Declare function min\_heap(int \*a, int m, int n) Declare j, t of the integer datatype.

Initialize t = a[m]. j = 2 \* m;

while (j <= n) do

if (j < n && a[j+1] < a[j]) then j = j + 1

if (t < a[j]) then break

else if (t >= a[j]) then a[j / 2] = a[j]

j = 2 \* j a[j/2] = t

return End.

For build\_minheap:

Begin

Declare function build\_minheap(int \*a,int n).

Declare k of the integer datatype.

for(k = n/2; k >= 1; k--)

Call function min\_heap(a,k,n)

End.

Code:-

#include <iostream> #include <conio.h> using namespace std;

void make\_heap(int \*a, int i, int n)

{

int j, temp; temp = a[i]; j = 2 \* i;

while (j <= n)

{

if (j < n && a[j+1] < a[j]) j = j + 1;

if (temp < a[j]) break;

else if (temp >= a[j])

{

a[j / 2] = a[j]; j = 2 \* j;

}

}

a[j/2] = temp; return;

}

void build\_heap(int \*a,int n)

{

int i;

for(i = n/2; i >= 1; i--)

{

make\_heap(a,i,n);

}

}

int main()

{

int n,i,a[100];

cout<<"Enter Array Size\n"; cin>>n;

for(i=1;i<=n;i++){ cout<<"\nEnter Array Element\n"; cin>>a[i];

}

build\_heap(a,n);

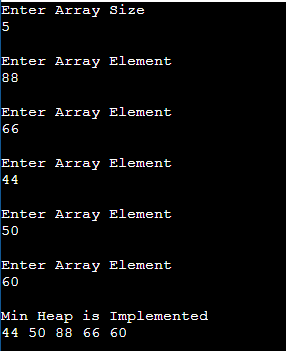
cout<<"\nMax Heap is Implemented\n";

for(i=1;i<=n;i++){ cout<<a[i]<<" ";

}

return 0;

}

Output:-

# Practical 13

### Implementation of Hashing Technique.

Description:-

A hash table is a data structure which is used to store key-value pairs. Hash function is used by hash table to compute an index into an array in which an element will be inserted or searched.

Algorithm:-

Begin

Initialize the table size T\_S to some integer value.

Create a structure hashTableEntry to declare key k and value v. Create a class hashMapTable:

Create a constructor hashMapTable to create the table. Create a hashFunc() function which return key mod T\_S. Create a function Insert() to insert element at a key.

Create a function SearchKey() to search element at a key. Create a function Remove() to remove element at a key.

Call a destructor hashMapTable to destroy the objects created by the constructor.

In main, perform switch operation and enter input as per choice. To insert key and values, call insert().

To search element, call SearchKey(). To remove element, call Remove().

End

Code:-

// Implementing hash table in C++ #include <iostream>

#include <list>

using namespace std;

class HashTable

{

int capacity; list<int> \*table;

public: HashTable(int V);

void insertItem(int key, int data); void deleteItem(int key);

int checkPrime(int n)

{

int i;

if (n == 1 || n == 0)

{

return 0;

}

for (i = 2; i < n / 2; i++)

{

if (n % i == 0)

{

return 0;

}

}

return 1;

}

int getPrime(int n)

{

if (n % 2 == 0)

{ n++;

}

while (!checkPrime(n))

{

n += 2;

}

return n;

}

int hashFunction(int key)

{

return (key % capacity);

}

void displayHash();

};

HashTable::HashTable(int c)

{

int size = getPrime(c); this->capacity = size;

table = new list<int>[capacity];

}

void HashTable::insertItem(int key, int data)

{

int index = hashFunction(key); table[index].push\_back(data);

}

void HashTable::deleteItem(int key)

{

int index = hashFunction(key);

list<int>::iterator i;

for (i = table[index].begin();

i != table[index].end(); i++)

{

if (\*i == key) break;

}

if (i != table[index].end()) table[index].erase(i);

}

void HashTable::displayHash()

{

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

{

cout << "table[" << i << "]"; for (auto x : table[i])

cout << " --> " << x; cout << endl;

}

}

int main()

{

int key[] = {231, 321, 212, 321, 433, 262};

int data[] = {123, 432, 523, 43, 423, 111};

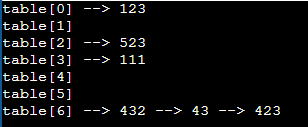
int size = sizeof(key) / sizeof(key[0]);

HashTable h(size);

for (int i = 0; i < size; i++) h.insertItem(key[i], data[i]);

//h.deleteItem(12); h.displayHash();

}

Output:-