# Practical No. 01

**Problem Statement:Implementation of different Sorting Techniques.**

# Bubble Sort

**Complexity:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithms | Time Complexity | |  | Space  Complexity |
| Best Case | Average Case | Worst case |
| Bubble Sort | O(n) | O(n²) | O(n²) | O(1) |
| Selection Sort | O(n²) | O(n²) | O(n²) | O(1) |
| Insertion Sort | O(n) | O(n²) | O(n²) | O(1) |
| Radix Sort | O(nk) | O(nk) | O(nk) | O(n+k) |
| Shell Sort | O(n\*logn) | O(n\*logn) OR  O(n1.25) | O(n2) | O(1) |

**Radix Sort:** **n**-Length of Array & **k**-Number of digits.

# Code:

#include <iostream> using namespace std;

void Bubble(int entry[],int size)

{

for(int round=1;round<size;round++)

{

for(int i=0;i<size-round;i++)

{

if(entry[i]>entry[i+1])

{

int temp=entry[i];

entry[i]=entry[i+1]; entry[i+1]=temp;

} }

} }

void Display(int entry[], int size)

{

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

{

cout<<entry[i]<<" ";

}

cout<<endl;

}

int main()

{

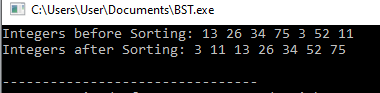
int entryInt[]={13,26,34,75,3,52,11};

cout<<"Integers before Sorting: "; Display(entryInt,7); cout<<"Integers after Sorting: "; Bubble(entryInt,7); Display(entryInt,7);

return 0;

}

# Output:



* **Insertion Sort**

# Algorithm:

insertionSort(array A)

Step 1: begin

for i = 1 to length[A] - 1 do

Step 2: begin

value = A[i]; Step 3: j = i - 1;

while j >= 0 and A[j] > value do

Step 4: begin

A[j + 1] = A[j];

j = j - 1; end;

Step 5: A[j + 1] = value;

end; end;

# Code:

#include <iostream> using namespace std;

void Insertion(int entry[],int size)

{

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

{

int live=entry[i]; int k;

for(k=i-1;k>=0 && entry[k]>live;k--) entry[k+1]=entry[k]; entry[k+1]=live;

}

}

void Display(int entry[], int size)

{

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

{

cout<<entry[i]<<" ";

}

cout<<endl;

}

int main()

{

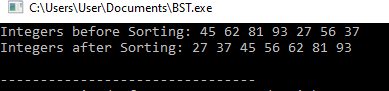
int entryInt[]={45,62,81,93,27,56,37};

cout<<"Integers before Sorting: "; Display(entryInt,7); cout<<"Integers after Sorting: "; Insertion(entryInt,7); Display(entryInt,7);

return 0;

}

# Output:



* **Selection Sort**
  + **Algorithm:** SelectionSort(array A) Step 1: begin

For I = 0 to N-1 do: Smallsub = I

Step 2: For J = I + 1 to N-1 do:

If A(J) < A(Smallsub) Smallsub = J

End-If End-For

Step 3: Temp = A(I)

A(I) = A(Smallsub) A(Smallsub) = Temp End-For

# Code:

#include <iostream> using namespace std;

void Selection(int entry[],int size)

{

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

{

int LiveMin=entry[i]; int LiveMinIndex=i; for(int j=i+1;j<size;j++)

{

if(LiveMin>entry[j])

{

LiveMin=entry[j];

LiveMinIndex=j;

} }

if(LiveMinIndex != i)

{

entry[LiveMinIndex]=entry[i]; entry[i]=LiveMin;

} } }

void Display(int entry[], int size)

{

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

{

cout<<entry[i]<<" ";

}

cout<<endl;

}

int main()

{

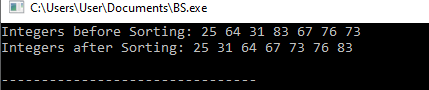
int entryInt[]={25,64,31,83,67,76,73};

cout<<"Integers before Sorting: "; Display(entryInt,7); cout<<"Integers after Sorting: "; Selection(entryInt,7); Display(entryInt,7);

return 0;

}

# Output:



* **Shell Sort**

# Algorithm:

Step 1: Caculate gap size

Step 2: WHILE gap is greater than 0

FOR each element of the list, that is gap apart Extract the current item

Step 3: Locate the position to insert Insert the item to the position END FOR

Step 4: Calculate gap size

END WHILE

# Code:

#include <iostream> using namespace std;

void Shell(int entry[],int size)

{

int gap,i,j;

for(int gap=size/2;gap>=1;gap=gap/2)

{

for(j=gap;j<size;j++)

{

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

{

if(entry[i+gap]>entry[i]){ break;

}

else

{

int temp=entry[i+gap]; entry[i+gap] =entry[i]; entry[i]=temp;

} } }

}}

void Display(int entry[], int size)

{

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

{

cout<<entry[i]<<" ";

}

cout<<endl;

}

int main()

{

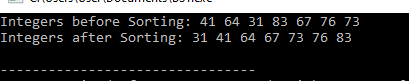
int entryInt[]={41,64,31,83,67,76,73};

cout<<"Integers before Sorting: "; Display(entryInt,7); cout<<"Integers after Sorting: "; Shell(entryInt,7); Display(entryInt,7);

return 0;

}

# Output:



* **Radix Sort**

# Algorithm:

Radix-Sort(A, d)

Step 1: for j = 1 to d do

int count[10] = {0};

Step 2: for i = 0 to n do

count[key of(A[i]) in pass j]++

Step 3: for k = 1 to 10 do

count[k] = count[k] + count[k-1]

Step 4: for i = n-1 downto 0 do

result[ count[key of(A[i])] ] = A[j]

count[key of(A[i])]-- Step 5: for i=0 to n do

A[i] = result[i] Step 6: end for(j)

end func

# Code:

#include <iostream> using namespace std;

int FindMax(int entry[],int size)

{

int large =entry[0]; for(int i=1;i<size;i++) if(entry[i]>large) large=entry[i];

return large;

}

void num(int entry[],int size,int base)

{

int n[10]={0}; int i;

int res[size]; for(i=0;i<size;i++) n[(entry[i]/base)%10]++; for(i=1;i<=9;i++)

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

for(i=size-1;i>=0;i--)

{

res[n[(entry[i]/base)%10]-1]=entry[i];

n[(entry[i]/base)%10]--;

}

for(i=0;i<size;i++) entry[i]=res[i];}

void Radix(int entry[],int size){ int large=FindMax(entry,size); int base;

for(base=1;large/base>0;base=base\*10)

{

num(entry,size,base);

}}

void Display(int entry[], int size){ for(int i=0;i<size;i++){ cout<<entry[i]<<" "; } cout<<endl;}

int main()

{

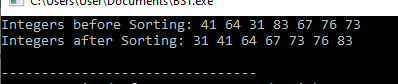
int entryInt[]={41,64,31,83,67,76,73};

cout<<"Integers before Sorting: "; Display(entryInt,7); cout<<"Integers after Sorting: "; Radix(entryInt,7); Display(entryInt,7);

return 0;

}

# Output:



* **Quick Sort**

# Algorithm:

QuickSort( double[] a ) Step 1: if ( a.length ≤ 1 )

return;

Step 2: Select a pivot;

Partition a[] in 2 halves: left[]: elements ≤ pivot right[]: elements > pivot;

Step 3: Sort left[]; Sort right[];

Step 4: Concatenate: left[] pivot right[]

# Code:

#include <iostream> using namespace std;

static void Quick(int entry[], int lside, int rside); static int part(int entry[], int lside, int rside); static void Display(int entry[], int size);

static void Quick(int entry[], int lside, int rside) { if (lside < rside) {

int rotate = part(entry, lside, rside); Quick(entry, lside, rotate-1); Quick(entry, rotate+1, rside);

}

}

static int part(int entry[], int lside, int rside) { int i = lside;

int rotate = entry[rside]; int temp;

for(int j = lside; j <=rside; j++) { if(entry[j] < rotate) {

temp = entry[i]; entry[i] = entry[j];

entry[j] = temp; i++;

}

}

temp = entry[rside]; entry[rside] = entry[i]; entry[i] = temp;

return i;

}

void Display(int entry[], int size)

{

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

{

cout<<entry[i]<<" ";

}

cout<<endl;

}

int main()

{

int entryInt[]={41,64,31,83,67,76,73,25,46,19,37,58,61,34,61,93};

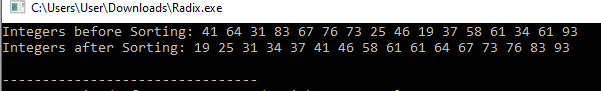
cout<<"Integers before Sorting: "; Display(entryInt,16);

int n = sizeof(entryInt) / sizeof(entryInt[0]); cout<<"Integers after Sorting: "; Quick(entryInt,0,n-1); Display(entryInt,16);

return 0;

}

# Output:



**Practical No. 02**

# Aim:Implementation of Searching Techniques.

**Complexity:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithms | Time Complexity | |  | Space  Complexity |
| Best Case | Average Case | Worst case |
| Linear Search | O(1) | O(n) | O(n) | O(1) |
| Binary Search | O(1) | O(logn) | O(logn) | O(1) |

* **Linear Search**

# Code:

#include<iostream> using namespace std;

int LinearSearch(int arr[],int sel,int n)

{

int loc;

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

{

if(arr[i]==sel)

{

loc=i; break;

}

else

{

loc=0;

}

}

return loc;

}

int main()

{

int n,sel,loc,arr[5]; cout<<"Series Size: "; cin>>n;

cout<<endl<<"Values of Series: "; for(int i=0;i<n;i++)

cin>>arr[i];

cout<<endl<<"Value to be Searched: "; cin>>sel;

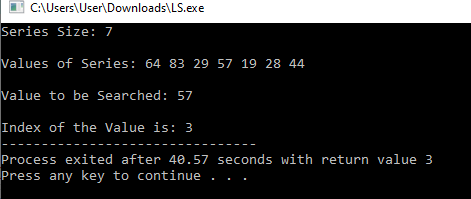
loc=LinearSearch(arr,sel,n); if(loc==-1)

cout<<endl<<"No such Value Exist!"; else

cout<<endl<<"Index of the Value is: "<<loc; return loc;

}

# Output:



* **Binary Search**
  + **Algorithm:** BinarySearch Step 1: Begin

Step 2: Set beg = 0

Set end = n-1

Set mid = (beg + end) / 2

Step 3: while ( (beg <= end) and (a[mid] ≠ item) ) do Step 4: if (item < a[mid]) then

Set end = mid - 1 else

Set beg = mid + 1 endif

Step 5: Set mid = (beg + end) / 2 Endwhile

Step 6: if (beg > end) then

Set loc = -1 else

Set loc = mid Endif

Step 7: End

# Code:

#include<iostream> using namespace std;

int BinarySearch(int arr[],int sel,int n)

{

int begin=0; int stop=n-1; int loc;

int cen=int((begin+stop)/2); while(begin<=stop && arr[cen]!=sel){ if(sel<arr[cen])

stop=cen-1; else begin=cen+1;

cen=int((begin+stop)/2);

}

if(arr[cen]==sel) loc=cen;

else loc=-1;

return cen;

}

int main()

{

int n,sel,loc,arr[5]; cout<<"Series Size: "; cin>>n;

cout<<endl<<"Values of Series: "; for(int i=0;i<n;i++)

cin>>arr[i];

cout<<endl<<"Value to be Searched: "; cin>>sel;

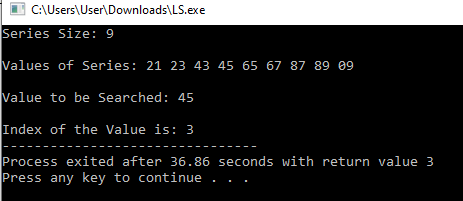
loc=BinarySearch(arr,sel,n); if(loc==-1)

cout<<endl<<"No such Value Exist!"; else

cout<<endl<<"Index of the Value is: "<<loc; return loc;

}

# Output:



**Practical No. 03**

# Aim: Implementation of Stacks.

# Operations:

# PUSH

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Push()** | Time Complexity | | | Space  Complexity |
| Best Case | Average Case | Worst case |
| Stack using Array | O(1) | O(1) | O(n) | O(1) |
| Stack using LinkedList | O(1) | O(1) | O(1) | O(1) |

# POP

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Pop()** | Time Complexity | |  | Space  Complexity |
| Best Case | Average Case | Worst case |
| Stack using Array | O(1) | O(1) | O(1) | O(1) |
| Stack using LinkedList | O(1) | O(1) | O(1) | O(1) |

# PEEK

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Peek()** | Time Complexity | |  | Space  Complexity |
| Best Case | Average Case | Worst case |
| Stack using Array | O(1) | O(1) | O(1) | O(1) |
| Stack using LinkedList | O(1) | O(1) | O(1) | O(1) |

**Implementation of Stack using Array**

# Code:

#include<iostream> using namespace std; #define MSize 25

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

void push()

{

int item;

if(top==MSize-1)

{

cout<<"STACK FULL\n";

}

else

{

cout<<"Enter values to be intrested: "; cin>>item;

stack[++top]=item;

}

}

void pop(){ int item; if(top==-1)

cout<<"EMPTY STACK"<<endl; else

{

item=stack[top-1];

cout<<"Deleted Element: "<<item;

}

}

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

cout<<"EMPTY STACK"<<endl ; else

{

cout<<"Values in Stack: "<<endl ; for(int i=top;i>=0;i--) cout<<"\n"<<stack[i];

}

}

int main(){ int choice; char ch; do{

cout<<"\*\*\*\* Stack Operation \*\*\*\*\n\n";

cout<<"1-Push Value\n\n2- Pop Value\n\n3- Traverse\n\n4-Exit\n"; cin>>choice;

switch(choice){ case 1:

push(); break; case 2:

pop(); break; case 3:

traverse(); break; default:

cout<<"\n Invalid Choice";

}

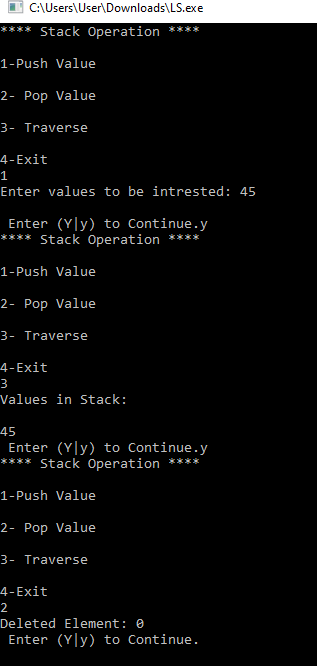
cout<<"\n Enter (Y|y) to Continue."; cin>>ch;

}

while(ch=='Y'||ch=='y'); return 0;

}

# Output:



* **Implementation of Stack using Linked List**

# Code:

#include<iostream> using namespace std; template<typename E> class SNode{

public:

E element; SNode<E>\* next;}; template<typename E> class SLinkL{

public:

SLinkL();

~SLinkL();

void addFore( const E& e); void del();

void show(); private:

SNode<E>\* head;}; template<typename E> SLinkL<E>::SLinkL(){ head=NULL; } template<typename E> SLinkL<E>::~SLinkL(){ del();} template<typename E>

void SLinkL<E>::addFore( const E& e){ SNode<E>\* v=new SNode<E>;

v-> element =e; v->next=head; head=v;}

template<typename E> void SLinkL<E>::del(){ if(head==NULL) cout<<"Stack is Empty"; else {

SNode<E>\* old=head; head=old->next;

delete old; }} template<typename E> void SLinkL<E>::show(){ SNode<E>\* T;

for(T=head; T!=NULL;T=T->next) cout<<T->element<<endl;

}

int main()

{

SLinkL <int> a; a.addFore(22); a.addFore(23);

cout<<"Element in the Stack: "<<endl; a.show();

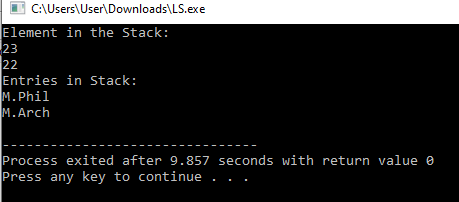
SLinkL <string> b; b.addFore("M.Arch");

b.addFore("M.Phil"); cout<<"Entries in Stack: "<<endl; b.show();

return 0;

}

# Output:



**Practical No. 4: Implementation of stack application**

**Complexity:**

**Stack Application:**

|  |  |  |
| --- | --- | --- |
| Algorithms | Time Complexity | Space Complexity |
| Postfix Evaluation | O(n) | O(n) |
| Balancing of Parenthesis | O(n) | O(n) |

## Aim: Write a program in c++ to implement postfix evaluation

## Code:

#include<iostream> using namespace std; #define MAXSIZE 20 int stack[MAXSIZE]; int top=-1;

int pop()

{

int item; if(top==-1)

cout<<"the stack is empty\n"; else

item=stack[top--]; return item;

}

void push(int item)

{

if(top==MAXSIZE-1)

cout<<"the stack is full\n"; else

stack[++top]=item;

}

void evaluatePostfix(char expr[])

{

int i; for(i=0;expr[i];++i)

{

if(expr[i]==' ') continue;

else if(isdigit(expr[i]))//reading integer

{

int num=0; while(isdigit(expr[i]))

{

num=num\*10+(int)(expr[i]-'0'); i++;

}

}

else

{

push(num);

//push(expr[i]-'0');

int A=pop(); int B=pop(); switch(expr[i])

{

case '+':

push(B+A); break;

case '-':

push(B-A); break;

case '\*':

push(B\*A); break;

case '/':

}

}

}

push(B/A); break;

cout<<stack[top];

}

int main()

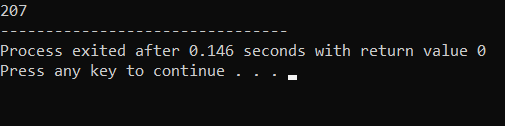
{

char expr[]="50 20 + 3 \* 9 3 / -"; evaluatePostfix(expr);

return 0;

}

## Output:



1. **Aim: Write a program in c++ to implement balancing of parenthesis**

## Algorithm:

Step 1: Declare a char stack

Step 2: Now traverse the expression from left to right till end of the expression.

* + If the character is open bracket ‘(‘ or ‘{‘ or ‘[‘ then push on stack
  + If the character is closing bracket ‘)’ or ‘}’ or ‘]’ then the top character.
  + If the popped character is matching with the open bracket then it is balanced otherwise not balanced.

Step 3: After complete traversal if there is any open bracket left in stack then it is also not balanced.

## Code:

#include<iostream> #include<string.h> using namespace std; #define MAX 20

class Stack //creating stack

{

public:

};

char stk[MAX]; int top;

Stack s;//stk and top

//to access stk and top we have to write s.stk[],s.top void push(char item)

{

if(s.top==(MAX-1)) cout<<"stack is full\n"; else

{

s.top++;//0,1,2,3,4 s.stk[s.top]=item;

}

}

void pop()

{

if(s.top==-1)

{

}

else

{

}

}

cout<<"stack is empty\n";

s.top=s.top-1;

bool balancedParenthesis(string expr)

{

int i=0; char x; s.top=-1;

for(i=0;i<expr.length();i++)

{

if(expr[i]=='('||expr[i]=='{'||expr[i]=='[')

{

push(expr[i]); continue;

}

switch(expr[i])

{

case ')':

x=s.stk[s.top]; pop(); if(x=='{'||x=='[') return false; break;

case '}':

x=s.stk[s.top]; pop(); if(x=='('||x=='[') return false; break;

case ']':

}

}

x=s.stk[s.top]; pop(); if(x=='{'||x=='(') return false; break;

if(i==expr.length()&&s.top==-1) cout<<"stack is balanced\n";

else

cout<<"stack is not balanced\n";

}

int main()

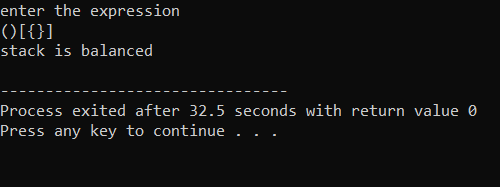
{

}

## Output:

string expr;

cout<<"enter the expression\n"; cin>>expr; balancedParenthesis(expr); return 0;



# Practical No. 5:

# Aim: Implement all different types of queues.

**Complexity:**

Different types of Queue

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Data structure | Time Complexity | |  | Space complexity |
| enqueue | dequeue | peek |
| Circular Queue | O(1) | O(1) | O(1) | O(n) |
| Priority Queue | O(log n) | O(log n) | O(1) | O(n) |

**Circular Queue**

## Code:

#include<iostream> #define max 4

using namespace std;

class CircularQ

{

public:

};

int cq[max]; int front,rear; CircularQ();

void enqueue(); void dequeue(); void display();

CircularQ::CircularQ()

{

front=rear=-1;

cout<<"Enter the number "; cin>>num;

//queue is empty if(front==-1) rear=front=0; else

rear=(rear+1)%max;

}

cq[rear]=num; cout<<num<<"is inserted..";

}

void CircularQ::dequeue()

{

int num; if(front==-1)

cout<<"Queue is empty"; else

{

num=cq[front];

cout<<"deleted item is"<<num; if(front==rear)

front=rear=-1; else

front=(front+1)%max;

}

}

void CircularQ::display()

{

int i; if(front==-1)

cout<<"Queue is empty"; else

{

cout<<"\n Queue elements are \n"; for(i=front;i<=rear;i++) cout<<cq[i]<<"\t";

}

if(front>rear)

{

for(i=front;i<max;i++) cout<<cq[i]<<"\t"; for(i=0;i<=rear;i++) cout<<cq[i]<<"\t";

}

}

int main()

{

CircularQ c; int choice; while(1)

{

cout<<"\n ---- Circular Queue Operation \n";

cout<<"\n 1. Enqueue \n 2. Dequeue \n 3. Display \n 4. Exit \n"; cout<<"Enter the choice ";

cin>>choice; switch(choice)

{

case 1:

case 2:

case 3:

case 4:

c.enqueue(); break;

c.dequeue(); break;

c.d isplay(); break;

exit(0);

default:

cout<<"wrong choice";

}

}

return 0;

}

**Output:**

1. . Enqueue
2. Dequeue
3. D T spley
4. Ex It

Ente n the r ho T r e 1 Enten the nutr be r 2 3 23 i s in s erted..

C T nr u1a n gueue Ope nat hon

1. . Enqueue
2. Dequeue
3. D T spley
4. Ex It

Ente n the r ho Tr e 3



Queue e1emen t c e ce

1. . Enqueue
2. Dequeue
3. D T spley
4. Ex It

Ente n the r ho Tr e 2

de1eted it em i s 23

C T nr u1a n gueue Ope nat hon



1. . Enqueue
2. Dequeue
3. D T spley
4. Ex It

Ente n the r ho Tr e 3

1 . Enqueue

1. Dequeue
2. D T spley
3. Ex It

Ente n the r ho Tr e 4



P ror e s s ex ited aTten 12.86 s er and s with return va lue 0

**Priority Queue**

## Code:

#include<iostream> using namespace std; struct node

{

int priority; int data;

struct node \*link;

};

class PriorityQueue

{

private:

node\* front;

public:

};

PriorityQueue(); void insert(); void deleteItem(); void display();

PriorityQueue::PriorityQueue()

{

front=NULL;

}

//This function will insert a data and its priority void PriorityQueue::insert()

{

node \*tmp,\*q;

int added\_item,item\_priority; tmp=new struct node;

cout<<"\nInput the item value to be added in the queue:";

if(front == NULL || item\_priority > front->priority)

{

}

else

{

}

}

tmp->link = front; front = tmp;

q = front;

while(q->link != NULL && q->link->priority >= item\_priority) q=q->link;

tmp->link = q->link; q->link = tmp;

void PriorityQueue::deleteItem()

{

node\* tmp; if(front == NULL)

cout<<"\nQueue Underflow\n"; else

{

tmp = front;

cout<<"\nDeleted item is %d\n"<<tmp->data; front = front->link;

delete(tmp);

}

}/\*End of del()\*/

void PriorityQueue::display()

{

node\* ptr; ptr = front;

if(front == NULL) cout<<"\nQueue is empty\n"; else

{

cout<<"\nQueue is:\n"; cout<<"\nPriority Item\n"; while(ptr != NULL)

{

cout<<"|"<<ptr->priority<<" "<<ptr->data<<"|->"; ptr = ptr->link;

}

}/\*End of else \*/

}/\*End of display() \*/ int main()

{

int choice; PriorityQueue p; while(1)

{

cout<<"\n 1.Insert \n"; cout<<"\n 2.Delete \n"; cout<<"\n 3.Display \n"; cout<<"\n 4.Quit \n"; cout<<"\n Enter your choice ";

cin>>choice; switch(choice)

{

case 1:

case 2:

case 3:

case 4:

p.insert(); break;

p.deleteItem(); break;

p.display(); break;

exit(1);

default :

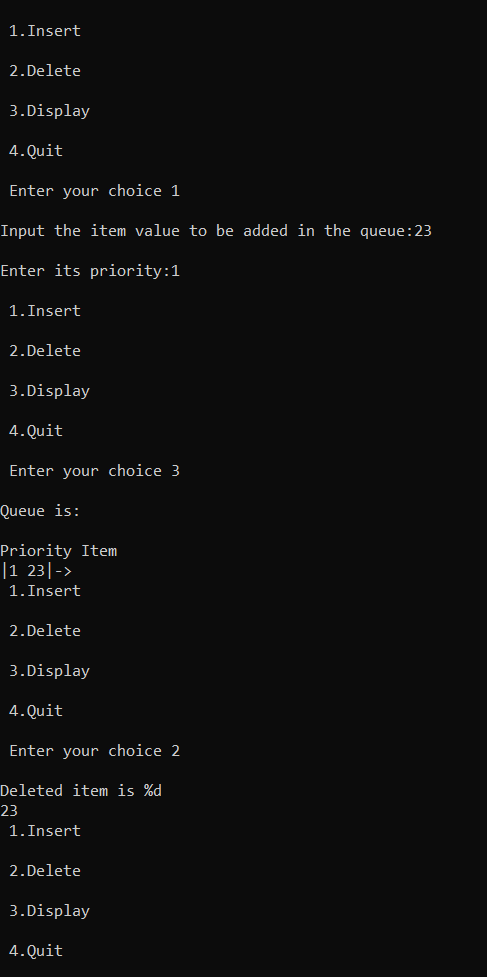
cout<<"\nWrong choice\n";

}/\*End of switch\*/

}/\*End of while\*/

}/\*End of main()\*/

## Output:



# Practical No. 6

# Aim: Implement all types of linked list

# Complexity:

Different types of LinkedList:

Insert():

|  |  |  |  |
| --- | --- | --- | --- |
| Data structure | Time Complexity | | |
| Add at head | Add at tail | Add in middle |
| Singly  LinkedList | O(1) | O(1) or O(n) | O(n) |
| Doubly  LinkedList | O(1) | O(1) | O(1) |
| Circular LinkedList | O(1) | O(n) | O(n) |

# Delete()

|  |  |  |  |
| --- | --- | --- | --- |
| Data structure | Time Complexity | |  |
| Delete first node | Delete last node | Delete in middle |
| Singly LinkedList | O(1) | O(n) | O(n) |
| Doubly  LinkedList | O(1) | O(1) | O(1) |
| Circular LinkedList | O(1) | O(n) | O(n) |

# Search()

|  |  |  |  |
| --- | --- | --- | --- |
| Operation in LinkedList | Time Complexity | | |
| Singly  LinkedList | Doubly  LinkedList | Circular LinkedList |
| Search() | O(n) | O(n) | O(n) |

## Code:

#include<iostream> using namespace std; class SLinked\_List

{

//create a node struct node

{

int info;//data section

struct node\* link;//address section

};

struct node\* head; public:

SLinked\_List()

{

head = NULL;

}

void createList(int); void addAtBeg(int);

void addAfterPos(int,int); void deleteData();

void display();

};

void SLinked\_List::createList(int data)//insert 10,insert 20 head is not NULL

{

//create a node

struct node \*temp,\*q; temp = new struct node; temp->info=data;

temp->link = NULL; if(head == NULL)

{

}

else

{

}

}

head = temp;//temp as starting node

q = head;

while(q->link!=NULL) q = q->link;

q->link = temp;

void SLinked\_List::addAtBeg(int data)

{

struct node\* temp; temp = new struct node; temp->info=data;

temp->link=head; head=temp;

}

void SLinked\_List::addAfterPos(int data,int pos)//15 at pos = 3 swipe to 4 one

{

struct node\* temp,\*q; int i;

q = head; for(i=0;i<pos-1;i++)

{

q = q->link; if(q==NULL)

{

cout<<"\n there are less than "<<pos<<"elements"; return;

}

}

temp = new struct node; temp->link= q->link; temp->info=data;

q->link=temp;

}

void SLinked\_List::deleteData()

{

struct node\* temp,\*q; int data;

if(head == NULL)

{

cout<<"list is empty"; return;

}

cout<<"\n enter the element for deletion"; cin>>data;

if(head->info==data)//if the data is in first node

{

temp=head;

head = head->link; delete(temp);

return;

}

//if the data is in between the list q=head;

while(q->link != NULL)

{

if(q->link->info==data)

{

temp=q->link;

q->link = temp->link; delete(temp);

}

q=q->link;

}

//if data is at last

if(q->link->info==data)

{

temp=q->link; delete(temp);

q->link = NULL; return;

}

}

void SLinked\_List::display()

{

struct node \*q; if(head==NULL)

{

cout<<"\n List is empty"; return;

}

q=head;

cout<<"\n List of elements:"; while(q!=NULL)

{

cout<<q->info<<" ";

q=q->link;//passing to next address

}

}

int main()

{

int choice,size,element,pos;

SLinked\_List sl; while(1)

{

cout<<"1:Create list\n"; cout<<"2: Add element at first\n"; cout<<"3: Add after\n"; cout<<"4: Delete\n";

cout<<"5: Display\n"; cout<<"6: Quit\n"; cout<<"Enter choice:\n"; cin>>choice; switch(choice)

{

case 1:

cout<<"\nHow many nodes to create:"; cin>>size;

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

{

case 2:

}

break;

cout<<"Enter the Element:"; cin>>element; sl.createList(element);

insterted";

case 3:

case 4:

case 5:

case 6:

cout<<"\n enter the element:"; cin>>element; sl.addAtBeg(element);

break;

cout<<"\n enter the element: "; cin>>element;

cout<<"enter the position where the elemenet to be

cin>>pos; sl.addAfterPos(element,pos); break;

sl.deleteData(); break;

sl.display(); break;

exit(0); default:

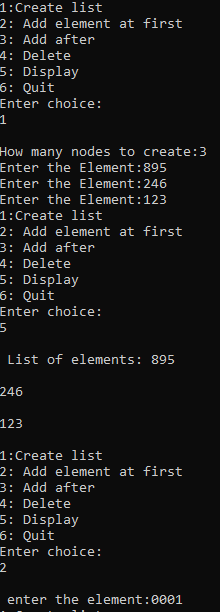
cout<<"\nwrong choice";

}

}

}

## Output:



**Doubly LinkedList**

**Code:**

#include<iostream> using namespace std; class dlinked\_list { struct node {

int data;

struct node\* prev; struct node\* next;

};

struct node\* head; int data;

public:

dlinked\_list();

void insertAtFront(); void insertAtEnd();

void insertAtposition(int); void deleteAtFront(); void deleteAtend();

void deleteAtpos(int); void display();//traverse

};

dlinked\_list :: dlinked\_list() { head=NULL;

}

void dlinked\_list :: insertAtFront() { struct node\* temp;

cout<<"enter data into the node"; cin>>data;

temp=new struct node; temp->data=data; temp->prev=NULL;

temp->next=NULL; if(head==NULL) {

head=temp;

} else {

temp->next=head; head->prev=temp; head=temp;

}

}

void dlinked\_list :: insertAtEnd() {

struct node\* temp,\*t;

cout<<"enter data into the node"; cin>>data;

temp=new struct node; temp->data=data; temp->prev=NULL; temp->next=NULL; if(head==NULL) {

head=temp;

} else {

t=head;

while(t->next!=NULL) {

t=t->next;

}

t->next=temp; temp->prev=t;

}

}

void dlinked\_list :: insertAtposition(int pos) { struct node\* temp,\*pr,\*aft;

int index=0;

cout<<"enter data into the node"; cin>>data;

temp=new struct node; temp->data=data; temp->prev=NULL; temp->next=NULL;

if(head==NULL) { //if it is empty head=temp;

} else {

pr=aft=head; if(pos==0) {

temp->next=head; head=temp;

} else {

while(index<pos) {

index++; pr=aft; aft=aft->next;

}

pr->next=temp; temp->prev=pr; temp->next=aft; aft->prev=temp;

}

}

}

void dlinked\_list :: deleteAtFront() { struct node\* t;

t=head; head=head->next;

head->prev=NULL;

cout<<t->data<<"deleted successfully"; delete(t);

}

void dlinked\_list :: deleteAtend() { struct node \*pr,\*aft; pr=aft=head; if(head==NULL)

cout<<"list is empty";

else {

while(aft->next!=NULL) { pr=aft;

aft=aft->next;

}

pr->next=NULL;

cout<<aft->data<<"deleted successfully"; delete(aft);

}

}

void dlinked\_list :: deleteAtpos(int pos){ struct node \*pr,\*aft; pr=aft=head;

int count=0; if(head==NULL)

cout<<"list is empty";

else

{

if(pos==0){

deleteAtFront();}

else{

while(count<pos) { count++;

pr=aft; aft=aft->next;

}

pr->next=aft->next;

aft->next->prev=pr;

cout<<aft->data<<"is deleted"; delete(aft);

}

}

}

void dlinked\_list :: display() { struct node \*t; if(head==NULL) {

cout<<"list is empty"<<endl;

} else {

cout<<"the elements in the list are"<<endl; t=head;

while(t!=NULL) {

cout<<t->data<<"<=>";

t=t->next;//incrementing the node

}

}

}

int main() {

int choice,size; int element,pos; dlinked\_list d1; while(1) {

cout<<"1: Add Element at First\n"; cout<<"2: Add At end\n"; cout<<"3: add at position\n";

cout<<"4: delete Element at First\n"; cout<<"5: delete At end\n";

cout<<"6: delete element at position\n"; cout<<"7: Display\n";

cout<<"8: Quit\n"; cout<<"1: Enter Choice\n"; cin>>choice; switch(choice) {

case 1:

d1.insertAtFront(); break;

case 2:

d1.insertAtEnd(); break;

case 3:

cout<<"enter the position"<<endl; cin>>pos; d1.insertAtposition(pos);

break;

case 4:

case 5:

d1.deleteAtFront(); break;

case 6:

case 7:

d1.deleteAtend(); break;

cout<<"enter the position"<<endl; cin>>pos;

d1.deleteAtpos(pos); break;

d1.display(); break;

case 8:

exit(0);

default:

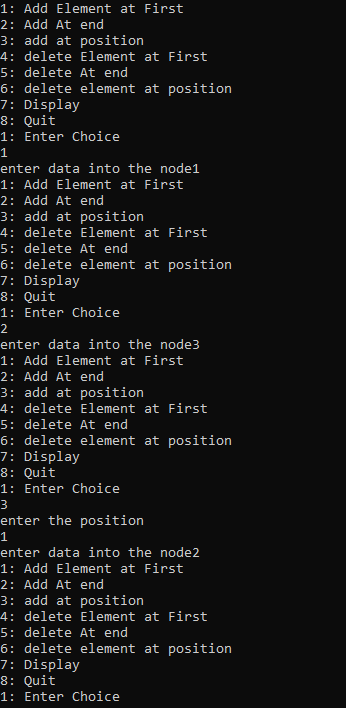
cout<<"Wrong Choice";

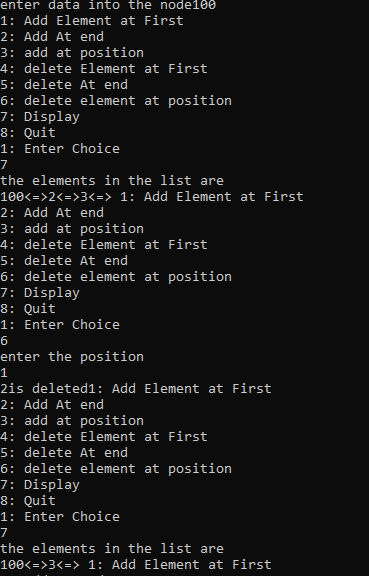
}

}

}

## Output:





**Circular LinkedList**

## Code:

#include<iostream> using namespace std; struct node

{

int data;

struct node\* link;

};

class CList

{

int data;

struct node\* head; public:

CList();

void insertFront(); void insertEnd(); void insertPos(int); void display(); void deleteFront(); void deleteEnd();

void deletePos(int);

};

CList::CList()

{

head=NULL;

}

void CList::insertFront()

{

struct node \*temp, \*t; temp = new struct node;

cout<<"enter element"<<endl; cin>>data;

temp->data=data; if(head == NULL)

{

head=temp;

temp->link=head;

}

else

{

temp->link=head; t = head;

while(t->link != head)

{

t = t->link;

}

t->link= temp; head = temp;

}

cout<<"inserted successfully"<<endl;

}

void CList::insertEnd()

{

struct node\* temp,\*t; int data;

cout<<"enter data to insert"; cin>>data;

temp=new struct node; temp->data=data; temp->link=NULL;

if(head==NULL)//if there no element in the list

{

head=temp;

temp->link=head;

}

else

{

t=head;

if(t->link==head)//list containing one node

{

t->link=temp; temp->link=t;

}

else

{

while(t->link!=head)

{

t=t->link;

}

t->link=temp;

temp->link=head;

}

}

cout<<"node inserted successfully";

}

void CList::insertPos(int pos)

{

struct node \*temp, \*t; int i;

temp = new struct node; cout<<"enter element"<<endl; cin>>data;

temp->data = data; if(head == NULL)

{

cout<<"List is empty";

}

else

{

t = head;

for(i=0; i<=pos-3; i++)

{

t = t->link;

}

temp->link = t->link; t->link = temp;

cout<<"inserted sucessfully"<<endl;

}

}

void CList::deleteFront()

{

struct node \*temp,\*t; t=head;

data=head->data; while(t->link!=head)

{

t=t->link;

}

temp=head;

head=head->link; t->link=head; delete(temp);

cout<<data<<" deleted sucessfully"<<endl;

}

void CList::deleteEnd()

{

struct node \*t,\*temp;//here t is current node and temp is previous node t=head;

while(t->link!=head)

{

temp=t; t=t->link;

}

temp->link=head; data=t->data; delete(t);

cout<<data<<" deleted successfully"<<endl;

}

void CList::deletePos(int pos)

{

struct node \*temp,\*t; t=head;

int count=0; if(head==NULL) cout<<"list is empty"; else

{

if(pos==0)

{

deleteFront();

}

else

{

while(count<pos) {

count++; temp=t; t=t->link;

}

temp->link=t->link;

cout<<t->data<<"is deleted"; delete(t);

}

}

}

void CList::display()

{

struct node \*t; if(head==NULL)

{

cout<<"list is empty\n";

}

else

{

t=head;

if(t->link==head){//if there is only one node cout<<t->data<<"->";

}

else

{

cout<<t->data<<"->"; t=t->link; while(t!=head)

{

cout<<t->data<<"->"; t=t->link;

}

}

}

}

int main()

{

CList c;

int choice,pos; while(1)

{

cout<<"\nCList operations:\n";

cout<<"\n 1.insertAtFront\n 2.insertAtEnd\n 3.insertAtpos\n 4.deleteAtFront\n 5.deleteAtEnd\n 6.deleteAtPos\n 7.display\n 8.exit\n"; cout<<"\n enter choice ";

cin>>choice; switch(choice)

{

case 1:

c.insertFront(); break;

case 2:

c.insertEnd(); break;

case 3:

cout<<"enter position"<<endl; cin>>pos;

c.insertPos(pos); break;

case 4:

c.deleteFront(); break;

case 5:

c.deleteEnd(); break;

case 6:

cout<<"enter the position"<<endl; cin>>pos;

c.deletePos(pos); break;

case 7:

c.display(); break;

case 8:

exit(0); default:cout<<"wrong choice";

}

}

return 0;

}

## Output:

|  |  |  |
| --- | --- | --- |
| T . In se ntAt F nont | | |
| 2. In sentAt End | |  |
| 1. In se ntAtpo s 2. de1eteAt Fnont | | |
| 5. de1eteAt End | |  |
| 6. deleteAt Pos | |
| 7.di splay |  | |

|  |
| --- |
|  |
| ente n e1ement |
|  |
| i n se n te d s u c c e s s I u lly |

|  |  |  |
| --- | --- | --- |
| T . In se ntAt F nont | | |
| 2. In sentAt End | |  |
| 1. In se ntAtpo s 2. de1eteAt F nont | | |
| 5. de1eteAt End | |  |
| 6. deleteAt Pos | |
| 7.di splay |  | |



enten data to In se nt 50

node In sented suse essTu11\

|  |  |  |
| --- | --- | --- |
| T . In se ntAt F nont | | |
| 2. In sentAt End | |  |
| 1. In se ntAtpo s 2. de1eteAt F nont | | |
| 5. de1eteAt End | |  |
| 6. deleteAt Pos | |
| 7.di splay |  | |



ente c e lesen t

|  |
| --- |
|  |
|  |
| 2. In se nt At End |
|  |
| 4. de leteAt Fcont |
| 5. de1eteAt End |
| 6. deleteAt Po s |
| 1. di sp lay 2. ex It |

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| T0 | de1eted | suc e s sI | u11y |

|  |  |  |
| --- | --- | --- |
|  | | |
| 2. In se nt At End | |  |
|  | | |
| 4. de leteAt Fcont | | |
| 5. de1eteAt End | |  |
| 6. deleteAt Po s | |
| 1. di sp lay 2. ex It |  | |

|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| 50 | de1eted | suc c es stu | 1 1 y |

|  |  |  |
| --- | --- | --- |
|  | | |
| 2. In se nt At End | |  |
|  | | |
| 4. de leteAt Fcont | | |
| 5. de1eteAt End | |  |
| 6. deleteAt Po s | |
| 1. di sp lay 2. ex It |  | |

ente n cho Ice 6

49 i s de leted

4. de1eteAt Fnont

1 . InsentAt Fnont

2. In se ntAtEnd

|  |  |
| --- | --- |
| 5. de1eteAt End | |
| 6 . de l ct eAt Pos | |
| 7.di splay |  |

|  |  |  |
| --- | --- | --- |
| T . In se ntAt F nont | | |
| 2. In sentAt End | |  |
| 1. In se ntAtpo s 2. de1eteAt F nont | | |
| 5. de1eteAt End | |  |
| 6. deleteAt Pos | |
| 7.di splay |  | |

enten cho ice 8

P no c e s s exited aPte n 9 5.76 second s uTth netunn va1ue 0

## Practical No. 7:

## Aim: Create and perform various operations on BST

1. Inserting node in BST
2. Deleting the node from BST
3. To find height of Tree
4. To perform Inorder
5. To perform Preorder
6. To perform Postorder
7. To find Maximum value of tree

**Complexity:**

Operations on BST(Binary Search tree):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operations | Time Complexity | |  | Space  Complexity |
| Best Case | Average Case | Worst case |
| Insertion | O(log n) | O(log n) | O(n) | O(n) |
| Deletion | O(log n) | O(log n) | O(n) | O(n) |
| Search | O(log n) | O(log n) | O(n) | O(n) |

|  |  |  |
| --- | --- | --- |
| Operation | Time Complexity | Space Complexity |
| BST Height | O(n)  n-No. of Nodes in BST | O(n) |
| Pre Order Traversal  Post Order Traversal  In-Order Traversal | O(n)  n-Size of Binary Tree | O(1) OR O(h) h-Height of Tree |

## Code:

#include<iostream> using namespace std; #define SPACE 10

//creating a tree node class treenode

{

public:

int data; treenode\* left; treenode\* right; treenode()

{

data=0; left=NULL;

right=NULL;

}

};

class BinarySearchTree

{

public:

treenode\* root;//node BinarySearchTree()

{

root=NULL;

}

void insertNode(treenode\* newnode)

{

if(root==NULL)//there is no node in the tree

{

}

else

{

root=newnode;

cout<<"node is inserted at root level"<<endl;

treenode\* temp=root;//to traverse the tree while(temp!=NULL)

{

if(newnode->data==temp->data)

{

cout<<"duplicacy is not allowed"<<endl; return;

}

else if((newnode->data<temp->data)&&(temp

>left==NULL))

{

temp->left=newnode;

cout<<"the node is inserted at left"<<endl; break;

}

else if(newnode->data<temp->data)

{

temp=temp->left;

}

else if((newnode->data>temp->data)&&(temp->right==NULL))

{

}

else

{

}

}

}

temp->right=newnode;

cout<<"the node is inserted at right"<<endl; break;

temp=temp->right;

}

treenode\* deleteNode(treenode\* r,int v)

{

bool found=false; if(root==NULL)

{

cout<<"tree is empty"<<endl; return NULL;

}

treenode\* curr; treenode\* parent; curr=root; while(curr!=NULL)

{

if(curr->data==v)

{

found=true;

}

else

{

break;

parent=curr; if(v>curr->data)

{

}

else

{

}

}

}

curr=curr->right;

curr=curr->left;

if(!found)

{

cout<<"the value is not present"<<endl; return NULL;

}

if(r==NULL)

{

return NULL;

}

else if(v<r->data)

{

r->left=deleteNode(r->left,v);

}

else if(v>r->data)

{

}

else

{

r->right=deleteNode(r->right,v);

if(r->left==NULL)

{

treenode\* temp=r->right; delete r;

return temp;

}

else if(r->right==NULL)

{

treenode\* temp=r->left; delete r;

return temp;

}

else

{

}

}

}

treenode\* temp=minValueNode(r->right); r->data=temp->data;

r->right=deleteNode(r->right,temp->data);

treenode\* minValueNode(treenode\* node)

{

treenode\* curr=node; while(curr->left!=NULL)

{

curr=curr->left;

}

return curr;

}

void display(treenode\* r, int space)

{

if(r==NULL)

{

return;

}

space+=SPACE; display(r->right,space); cout<<endl;

for(int i=SPACE;i<space;i++)

{

cout<<" ";

}

cout<<r->data<<"\n"; display(r->left,space);

}

int height(treenode\* r)

{

if(r==NULL)

{

}

else

{

return -1;

int lheight=height(r->left); int rheight=height(r->right); if(lheight>rheight)

{

}

else

{

}

}

}

return (lheight+1);

return (rheight+1);

void printPreorder(treenode\* r)

{

if(r==NULL)

{

return;

}

cout<<r->data<<" ";//read node printInorder(r->left);//read left printInorder(r->right);//read right

}

void printInorder(treenode\* r)

{

if(r==NULL)

{

return;

}

printPreorder(r->left);//read left cout<<r->data<<" ";//read node printPreorder(r->right);//read right

}

void printPostorder(treenode\* r)

{

if(r==NULL)

{

return;

}

printPostorder(r->left);//read left printPostorder(r->right);//read right cout<<r->data<<" ";//read node

}

int findMax(treenode\* root)

{

if(root==NULL)

{

return 0;

}

int res=root->data;

int lres= findMax(root->left);

int rres=findMax(root->right); if(lres>res)

{

res=lres;

}

if(rres>res)

{

res=rres;

}

return res;

}

};

int main()

{

BinarySearchTree bst; int choice,val; while(1)

{

cout<<"select bst opertaion:"<<endl; cout<<"1.insert node\n"; cout<<"2.delete node\n"; cout<<"3.display\n"; cout<<"4.display height of tree\n"; cout<<"5. preorder \n";

cout<<"6. Inorder \n"; cout<<"7. Postorder \n"; cout<<"8. Find max value \n"; cout<<"9.exit\n"; cin>>choice;

treenode t;

treenode\* newnode=new treenode(); switch(choice)

{

case 1:

case 2:

case 3:

cout<<"enter value to be inserted in the node:"<<endl; cin>>val;

newnode->data=val; bst.insertNode(newnode); break;

cout<<"enter value to delete"<<endl; cin>>val; bst.deleteNode(bst.root,val);

break; bst.display(bst.root,5);

case 4:

case 5:

case 6:

case 7:

case 8:

case 9:

int h;

cout<<"height is:"<<endl; h=bst.height(bst.root); cout<<h<<endl;

break;

bst.printPreorder(bst.root); break;

bst.printInorder(bst.root); break;

bst.printPostorder(bst.root); break;

int max; max=bst.findMax(bst.root);

cout<<"max value is:"<<max<<endl; break;

exit(0);

}

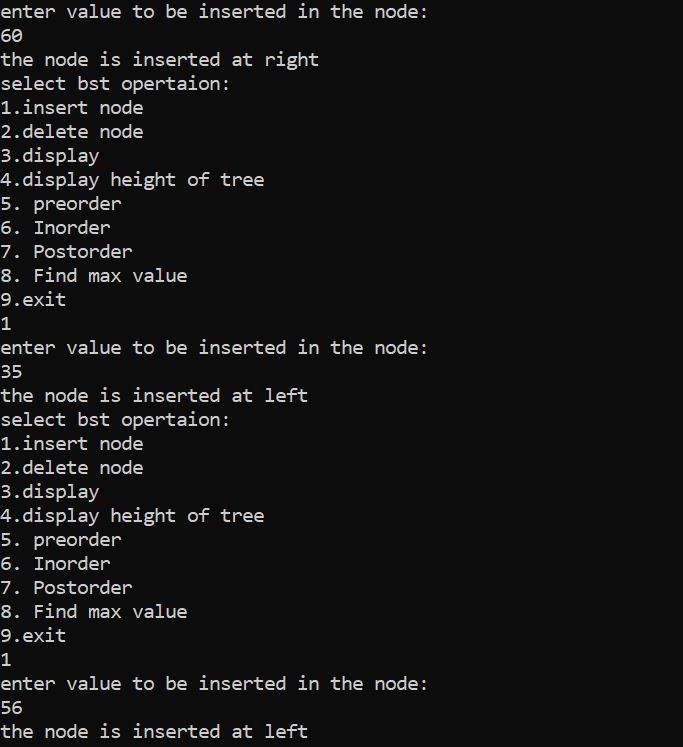
}

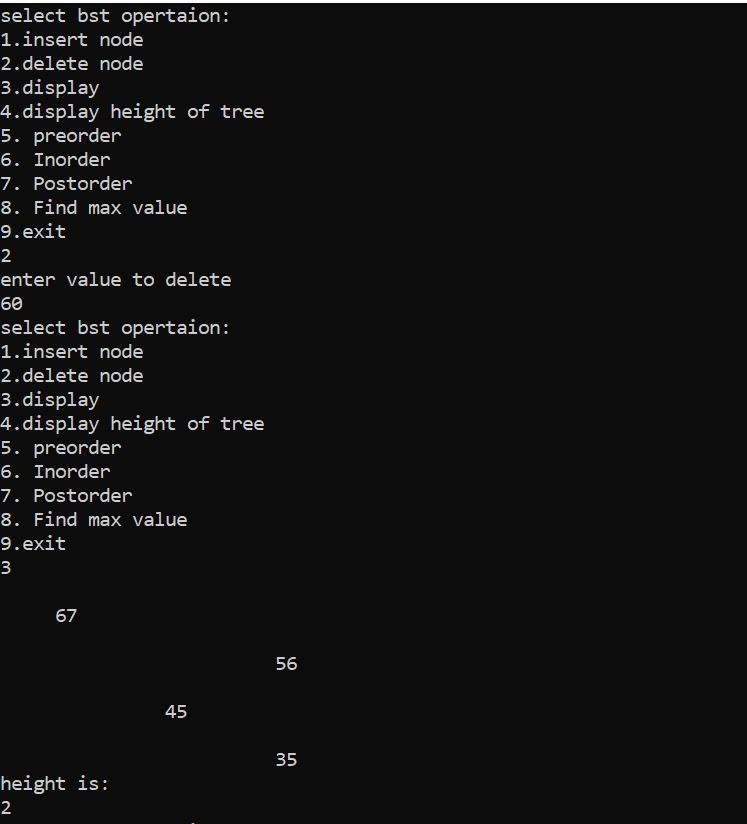
}

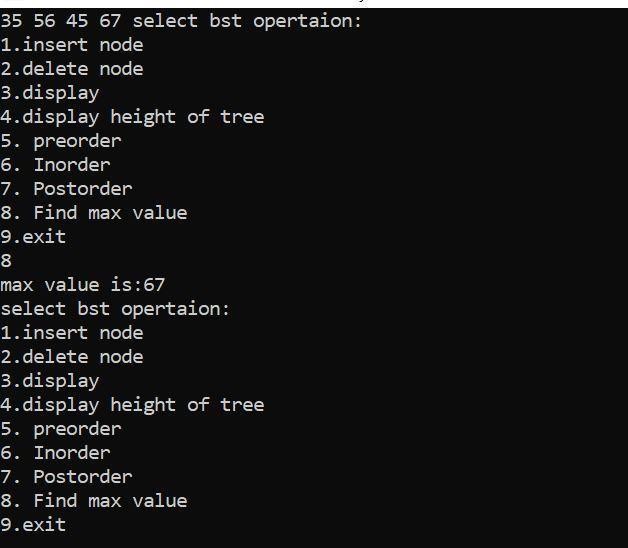
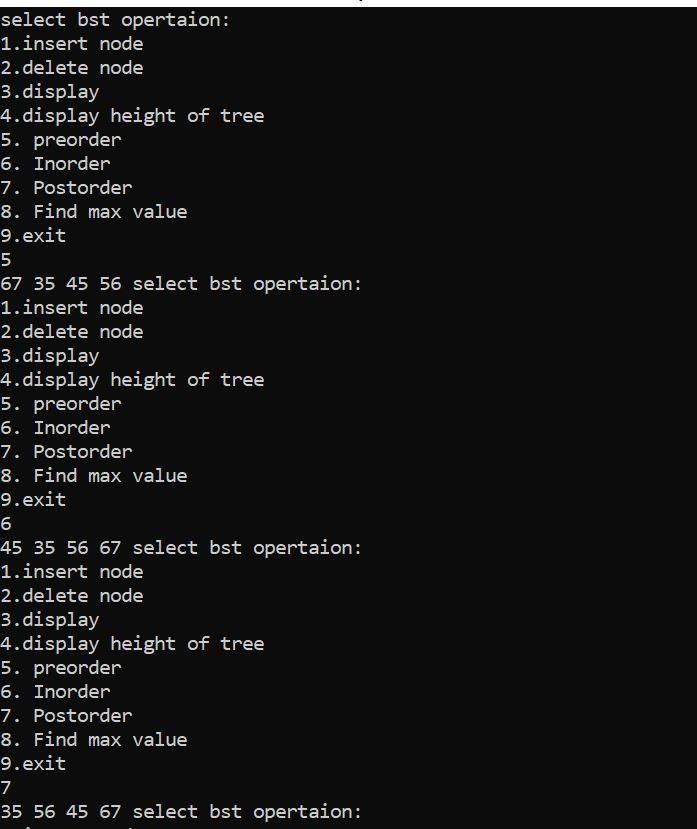
default:

cout<<"wrong choice"<<endl;

Output







## Practical No. 8

**Aim: Perform various hashing techniques with Linear Probe as collision resolution scheme.**

**Complexity:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operations | Time Complexity | |  | Space  Complexity |
| Best Case | Average Case | Worst case |
| Insertion | O(1) | O(1) or O(n) | O(n) | O(1) |
| Deletion | O(1) | O(1) or O(n) | O(n) | O(1) |
| Search | O(1) | O(1) (good hash function) O(n) (bad hash function) | O(n) | O(1) |

1. **implement linear probing**

**Code:**

#include <iostream>

#include <cstdio>

#include <cstdlib>

using namespace std;

const int T\_S = 5;

class HashTable {

public:

int k;

int v;

HashTable(int k, int v) {

this->k = k;

this->v = v;

}

};

class DelNode:public HashTable {

private:

static DelNode \*en;

DelNode():HashTable(-1, -1) {

}

public:

static DelNode \*getNode() {

if (en == NULL)

en = new DelNode();

return en;

}

};

DelNode \*DelNode::en = NULL;

class HashMapTable {

private:

HashTable \*\*ht;

public:

HashMapTable() {

ht = new HashTable\* [T\_S];

for (int i = 0; i < T\_S; i++) {

ht[i] = NULL;

}

}

int HashFunc(int k) {

return k % T\_S;

}

void Insert(int k, int v) {

int hash\_val = HashFunc(k);

int init = -1;

int delindex = -1;

while (hash\_val != init && (ht[hash\_val] == DelNode::getNode() || ht[hash\_val] != NULL && ht[hash\_val]->k != k)) {

if (init == -1)

init = hash\_val;

if (ht[hash\_val] == DelNode::getNode())

delindex = hash\_val;

hash\_val = HashFunc(hash\_val + 1);

}

if (ht[hash\_val] == NULL || hash\_val == init) {

if(delindex != -1)

ht[delindex] = new HashTable(k, v);

else

ht[hash\_val] = new HashTable(k, v);

}

if(init != hash\_val) {

if (ht[hash\_val] != DelNode::getNode()) {

if (ht[hash\_val] != NULL) {

if (ht[hash\_val]->k== k)

ht[hash\_val]->v = v;

}

} else

ht[hash\_val] = new HashTable(k, v);

}

}

int SearchKey(int k) {

int hash\_val = HashFunc(k);

int init = -1;

while (hash\_val != init && (ht[hash\_val] == DelNode::getNode() || ht[hash\_val] != NULL && ht[hash\_val]->k!= k)) {

if (init == -1)

init = hash\_val;

hash\_val = HashFunc(hash\_val + 1);

}

if (ht[hash\_val] == NULL || hash\_val == init)

return -1;

else

return ht[hash\_val]->v;

}

void Remove(int k) {

int hash\_val = HashFunc(k);

int init = -1;

while (hash\_val != init && (ht[hash\_val] == DelNode::getNode() || ht[hash\_val] != NULL && ht[hash\_val]->k!= k)) {

if (init == -1)

init = hash\_val;

hash\_val = HashFunc(hash\_val + 1);

}

if (hash\_val != init && ht[hash\_val] != NULL) {

delete ht[hash\_val];

ht[hash\_val] = DelNode::getNode();

}

}

~HashMapTable() {

delete[] ht;

}

};

int main() {

HashMapTable hash;

int k, v;

int c;

while(1) {

cout<<"1.Insert element into the table"<<endl;

cout<<"2.Search element from the key"<<endl;

cout<<"3.Delete element at a key"<<endl;

cout<<"4.Exit"<<endl;

cout<<"Enter your choice: ";

cin>>c;

switch(c) {

case 1:

cout<<"Enter element to be inserted: ";

cin>>v;

cout<<"Enter key at which element to be inserted: ";

cin>>k;

hash.Insert(k, v);

break;

case 2:

cout<<"Enter key of the element to be searched: ";

cin>>k;

if(hash.SearchKey(k) == -1) {

cout<<"No element found at key "<<k<<endl;

continue;

} else {

cout<<"Element at key "<<k<<" : ";

cout<<hash.SearchKey(k)<<endl;

}

break;

case 3:

cout<<"Enter key of the element to be deleted: ";

cin>>k;

hash.Remove(k);

break;

case 4:

exit(1);

default:

cout<<"\nEnter correct option\n";

}

}

return 0;

}

## Output

## Text Description automatically generated

## Text Description automatically generated

## Text Description automatically generated

## Practical No. 9

## Aim: Implementing Heap with different operations performed

1. To perform insertion operation
2. To create Heap using Heapify method
3. To perform Heap sort
4. To delete the value in heap

**Complexity:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operations | Time Complexity | |  | Space  Complexity |
| Best Case | Average Case | Worst case |
| Insertion | O(1) | O(log n) | O(log n) | O(n) |
| Deletion | O(1) | O(log n) | O(log n) |
| Sorting | O(n\*logn) | |  |
| Heapify | O(n) | |  |  |

## Code:

#include<iostream> using namespace std; #define height 10

int arr[20],n;

//Function to insert an element to the heap

void insert(int num,int loc)//35 4

{

int par; while(loc>0)//0

{

par = (loc-1)/2;//1st element,0th

if (num<=arr[par])//[0]=15,[1]=35 [4]=20

{

arr[loc]=num; return;

}

arr[loc]=arr[par];// loc=par;//recursive,loc=4,loc=1,loc=0

}/\*End of while\*/ arr[0]=num;

}/\*End of insert()\*/

//This function to create a heap void create\_heap()

{

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

//maxHeapify( arr, n,largest); insert(arr[i],i);

}/\*End of create\_heap()\*/

//Function to display the elements in the array void display()

{

int i;

for(i=0;i<n;i++) cout<<arr[i]<<endl; cout<<" ";

}/\*End of display()\*/

void maxHeapify(int arr[],int n, int i)

{

int largest = i;//i=3 int l=2\*i;//6,

int r=(2\*i)+1;//7

//comparing the root with its left and right child while(l<= n && arr[l]>arr[largest])

{

largest=l;

}

while(r<=n && arr[r]>arr[largest])

{

largest=r;

}

if(largest!=i)

{

int temp=arr[i]; arr[i]=arr[largest]; arr[largest]=temp; maxHeapify( arr, n,largest);

}

}

void build(int a[],int n)//create heap

{

int i; for(i=n/2;i>=0;i--) maxHeapify(a,n,i);

}

void del\_root(int last)

{

int left,right,i,temp;

i=0; /\*Since every time we have to replace root with last\*/

/\*Exchange last element with the root \*/ temp=arr[i];

arr[i]=arr[last]; arr[last]=temp;

left=2\*i+1; /\*left child of root\*/ right=2\*i+2;/\*right child of root\*/ while( right < last)

{

if ( arr[i]>=arr[left] && arr[i]>=arr[right] ) return;

if ( arr[right]<=arr[left] )

{

}

else

{

temp=arr[i]; arr[i]=arr[left]; arr[left]=temp; i=left;

temp=arr[i];

}

left=2\*i+1; right=2\*i+2;

arr[i]=arr[right]; arr[right]=temp; i=right;

}/\*End of while\*/

if(left==last-1 && arr[i] < arr[left])

//if (left==last–1 && arr[i] < arr[left] )/\*right==last\*/

{

temp=arr[i]; arr[i]=arr[left]; arr[left]=temp;

}

}/\*End of del\_root\*/

void deleteRoot(int arr[],int n)

{

int lastElement = arr[n-1]; arr[0]=lastElement;

n=n-1; maxHeapify(arr,n-2,0);

}

//Function to sort an element in the heap void heap\_sort()

{

int last;

for(last = n-1; last>=0;last--) del\_root(last);

}

int main()

{

int i;

cout<<"enter number of elements:"; cin>>n;

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

{

cout<<"enter elements:"; cin>>arr[i];

}

}

## Output:

## Text Description automatically generated

# Practical No. 10

# Aim: Create a graph storage structure

## implement adjancency matrix Algorithm: Create a graph

**Time complexity**: O(V+E), where V is number of vertices in the graph and E is number of edges in the graph.

**Space Complexity**: O ( V 2 ) O(V^{2}) O(V2),where V represents the number of vertices and E represents the number of edges in the graph.

**Code:** #include<iostream> using namespace std; #define MAX 20 class Graph

{

int adj[MAX][MAX]; int node;

int edge;

public:

Graph()

{

int i,j; for(i=0;i<MAX;i++) for(j=0;j<MAX;j++) adj[i][j]=0;

node=0; edge=0;

}

void createGraph();//bidirectional void createDGraph();

void display();

};

void Graph::createGraph()

{

int origin,dest,i;

cout<<"enter the no of nodes"<<endl; cin>>node;

cout<<"neter the no of edges"; cin>>edge; for(i=1;i<=edge;i++)

{

cout<<"\enter edge"<<i<<endl; cout<<"enter origin"<<endl; cin>>origin;

cout<<"enter dest";

cin>>dest; adj[origin][dest]=1; adj[dest][origin]=1; cout<<endl;

}

}

void Graph::createDGraph()

{

int origin,dest,i;

cout<<"enter the no of nodes"<<endl; cin>>node;

cout<<"neter the no of edges"; cin>>edge; for(i=1;i<=edge;i++)

{

cout<<"\enter edge"<<i<<endl; cout<<"enter origin"<<endl; cin>>origin;

cout<<"enter dest"; cin>>dest; adj[origin][dest]=1; adj[dest][origin]=0; cout<<endl;

}

}

void Graph::display()

{

int i,j;

for(i=1;i<=node;i++)

{

for(j=1;j<=node;j++) cout<<adj[i][j]<<" "; cout<<endl;

}

}

int main()

{

Graph g; g.createGraph(); g.display(); g.createDGraph(); g.display();}

**Output:**

Text

Description automatically generated

# Practical No. 12

# Aim: Create a minimum spanning tree using any method Kruskal’s algorithm or Prim’s algorithm

## implement minimum spanning tree

## Code:

#include<iostream> #define MAX 20 using namespace std; struct edge

{

int u; int v;

int weight;

struct edge \*link;

}\*front=NULL; int father[MAX];

struct edge tree[MAX]; int n;

int wt\_tree=0; int count=0;

void make\_tree();

void insert\_tree(int i,int j,int wt); void insert\_pque(int i,int j,int wt); struct edge\* del\_pque();

void create\_graph()

{

int i,wt,max\_edges,origin,destin; cout<<"enter no of nodes"; cin>>n;

max\_edges=n\*(n-1)/2; for(i=1;i<=max\_edges;i++)

{

cout<<"enter edges"<<i; cin>>origin>>destin; if((origin==0 )&&(destin==0)) break;

cout<<"enter weight for the edge:"; cin>>wt; if(origin>n||destin>n||origin<=0||destin<=0)

{

}

else

{

cout<<"ivalid edge"<<endl; i--;

insert\_pque(origin,destin,wt);

}

}//end of for if(i<n-1)

{

cout<<"spanning tree not possible"; exit(1);

}

}

int main()

{

int i; create\_graph(); make\_tree();

cout<<"edges to be included in spanning tree"<<endl; for(i=1;i<=count;i++)

{

cout<<tree[i].u; cout<<tree[i].v; cout<<endl;

}

cout<<"\nweight of minimum spanning tree is"<<wt\_tree;

}

void make\_tree()

{

struct edge\* temp;

int node1,node2,root\_n1,root\_n2; while(count<n-1)

{

temp=del\_pque(); node1=temp->u; node2=temp->v; cout<<"n1="<<node1; cout<<"n2="<<node2; while(node1>0)

{

root\_n1=node1; node1=father[node1];

}

while(node2>0)

{

root\_n2=node2; node2=father[node2];

}

cout<<"rootn1="<<root\_n1<<endl; cout<<"rootn2="<<root\_n2<<endl; if(root\_n1!=root\_n2)

{

insert\_tree(temp->u,temp->v,temp->weight); wt\_tree=wt\_tree+temp->weight; father[root\_n2]=root\_n1;

}

}

}

void insert\_tree(int i,int j,int wt)

{

cout<<"the edges inserted in the spanning tree:"<<endl;

count++; tree[count].u=i; tree[count].v=j; tree[count].weight=wt;

}

void insert\_pque(int i,int j,int wt)

{

struct edge\* temp,\*q;

//temp=(struct edge\*)malloc(sizeof(struct edge)); temp=new struct edge();

temp->u=i; temp->v=j;

temp->weight=wt;

if(front==NULL || temp->weight<front->weight)

{

}

else

{

}

temp->link=front; front=temp;

q=front;

while(q->link!=NULL && q->link->weight<=temp->weight) q=q->link;

temp->link=q->link; q->link=temp;

if(q->link==NULL) temp->link=NULL;

}

struct edge\* del\_pque()

{

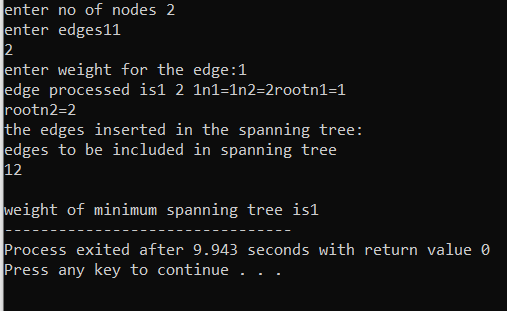
}

## Output:

struct edge\* temp; temp=front;

cout<<"edge processed is"<<temp->u<<" "<<temp->v<<" "<<temp->weight; front=front->link;

return temp;



# Practical No. 12

# Aim: Implementation of graph traversal

# Complexity:

Time Complexity:0(V)

Space Complexity:0(h)

## 

## implement Depth First Search (DFS)

## Code:

#include<iostream> #include<stdio.h> #define max 10 using namespace std;

/\* a function to build adjacency matrix of a graph \*/ void buildadjm(int adj[][max], int n)

{

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

{

cout<<"enter 1 or 0:"<<i<<j; cin>>adj[i][j];

}

}

/\* a function to visit the nodes in a depth first order \*/ void dfs(int x,int visited[],int adj[][max],int n)

{

int j; visited[x] = 1;

//printf(“\nThe node visited id %d\n”,x); cout<<"the node visited id is"<<x; for(j=1;j<=n;j++)

{

if (adj[x][j] ==1 && visited[j] ==0) dfs(j,visited,adj,n);

}

}

int main()

{

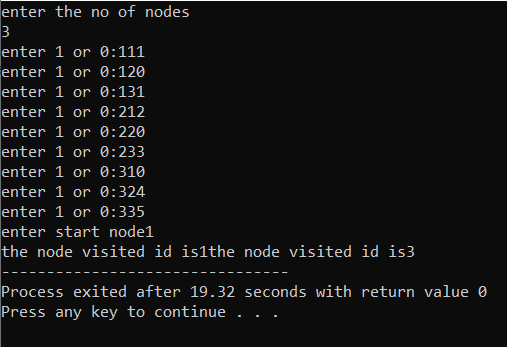
int adj[max][max],node,n; int i, visited[max];

cout<<"enter the no of nodes"<<endl; cin>>n;

buildadjm(adj,n); for(i=1;i<=n;i++) visited[i] =0; cout<<"enter start node"; cin>>node; if(visited[node] ==0) dfs(node,visited,adj,n);

}

## Output:



**implement Breath First Search (BFS)**

## Code:

#include<iostream> #define MAX 50 using namespace std; struct node

{

int vertex; node \*next;

};

node \*adj[MAX];

int totNodes;//number of nodes in graph int queue[MAX],front=-1,rear=-1;

void enqueue(int item)

{

rear=rear+1; queue[rear]=item; if(front==-1) front=0;

}

int dequeue()

{

int delItem=queue[front]; if(front==rear) front=rear=-1;

else front=front+1; return(delItem);

}

int isQueueEmpty()

{

if(front==-1) return 1; else

return 0;

}

void createGraph()

{

node \*new1,\*last;

int neighbours,neighbour\_val; cout<<"Proceeding for graph creation..."<<endl; cout<<"enter the number of nodes"<<endl; cin>>totNodes;

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

{

last=NULL;//store address of next node cout<<"enter num of nodes neighbour to"<<i<<endl; cin>>neighbours;

cout<<"neighbours of"<<i<<"are: "<<endl; for(int j=1;j<=neighbours;j++)

{

cout<<"enter name of the neighbour: "<<endl; cin>>neighbour\_val;

new1 = new node;//creation of node new1->vertex=neighbour\_val; new1->next=NULL; if(adj[i]==NULL) adj[i]=last=new1;

else

{

last->next=NULL; last=new1;

}

}

}

}

void BFS\_traversal()

{

node \*temp;

int startNode,status[MAX],N,v;

const int ready=1,wait=2,processed=3; cout<<"enter the start node"; cin>>startNode;

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

{

status[i]=ready;

}

enqueue(startNode); status[startNode]=wait; while(isQueueEmpty()!=1)

{

N=dequeue(); status[N]=processed; cout<<" "<<N; temp=adj[N]; while(temp!=NULL)

{

v=temp->vertex; if(status[v]==ready)

{

enqueue(v); status[v]=wait;

}

temp=temp->next;

}

}

}

int main()

{

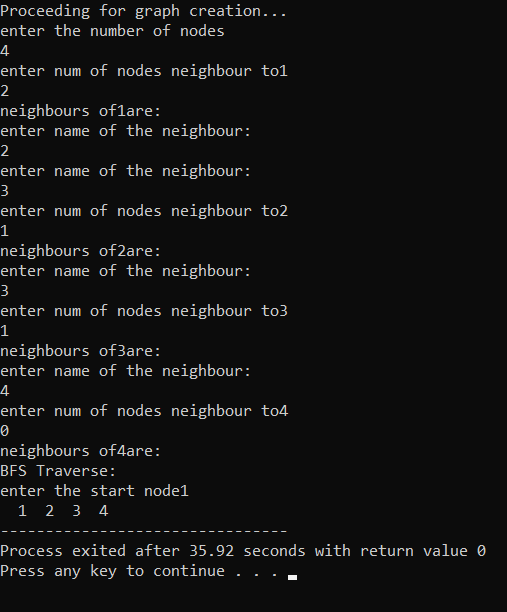
}

## Output:

createGraph();

cout<<"BFS Traverse: "<<endl; BFS\_traversal();

return 0;



**Practical No. 13:**

**Aim:To demonstrate application of linked list**

**implement polynomial addition**

## Code:

#include<iostream> using namespace std; class PolyAdd

{

private: //creation of node struct polynode

{

float coeff; int exp;

polynode \*link;

};

struct polynode\* head; public:

PolyAdd();

void createpoly(float c, int e); void displaypoly();

void addpoly(PolyAdd &p1,PolyAdd &p2);

//~PolyAdd();

};

PolyAdd :: PolyAdd()

{

head=NULL;

}

void PolyAdd :: createpoly(float c, int e)

{

polynode \*temp,\*ptr; temp=new struct polynode; temp->coeff=c;

temp->exp=e;

temp->link=NULL; if(head==NULL || e>head->exp)

{

}

else

{

temp->link=head; head=temp;

ptr=head;

while(ptr->link!=NULL && ptr->link->exp>e)

{

ptr=ptr->link;

}

ptr->link=temp; ptr=ptr->link;

}

}

void PolyAdd ::addpoly(PolyAdd &p1, PolyAdd &p2)

{

struct polynode\* result; if(p1.head==NULL && p2.head==NULL)

{

return;

}

polynode \*temp1,\*temp2; temp1=p1.head; temp2=p2.head;

while(temp1!=NULL && temp2!=NULL)

{

if(head==NULL)

{

}

else

{

}

head=new polynode; result=head;

result->link=new polynode; result=result->link;

if(temp1->exp < temp2->exp)

{

result->coeff=temp2->coeff; result->coeff=temp2->exp; temp2=temp2->link;

}

else if(temp1->exp > temp2->exp)

{

result->coeff=temp1->coeff; result->exp=temp1->exp; temp1=temp1->link;

}

else if(temp1->exp==temp2->exp)

{

result->coeff=(temp1->coeff)+(temp2->coeff); result->exp=temp1->exp;

temp1=temp1->link; temp2=temp2->link;

}

}

while(temp1!=NULL)

{

if(head==NULL)

{

}

else

{

}

head=new polynode; result=head;

result->link=new polynode; result=result->link;

result->coeff=temp1->coeff; result->exp=temp1->exp;

temp1=temp1->link;

}

while(temp2!=NULL)

{

if(head==NULL)

{

}

else

{

}

head=new polynode; result=head;

result->link=new polynode; result=result->link;

result->coeff=temp2->coeff; result->exp=temp2->exp; temp2=temp2->link;

}

result->link=NULL;

}

void PolyAdd :: displaypoly()

{

polynode \*q; q=head; while(q!=NULL)

{

if(q->exp!=0)

{

}

else

{

}

cout<<q->coeff<<"x^"<<q->exp; cout<<"+";

cout<<q->coeff;

q=q->link;

}

}

int main()

{

PolyAdd p1; p1.createpoly(3,3); p1.createpoly(5,4); p1.createpoly(5,0);

cout<<"the first polynomial is: "<<endl; p1.displaypoly();

PolyAdd p2; p2.createpoly(4,4); p2.createpoly(2,3); p2.createpoly(8,0);

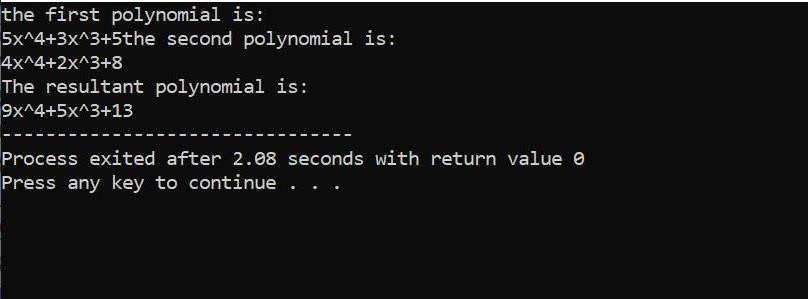
cout<<"the second polynomial is: "<<endl; p2.displaypoly();

PolyAdd p3; p3.addpoly(p1,p2);

cout<<"\nThe resultant polynomial is: "<<endl; p3.displaypoly();

return 0;

}



## implement Sparse Matrix

## Code:

// C++ program for sparse matrix representation.

// Using Link list #include<iostream> using namespace std;

// Node class to represent link list class Node

{

public:

int row; int col; int data;

Node \*next;

};

// Function to create new node

void create\_new\_node(Node \*\*p, int row\_index,int col\_index, int x)

{

Node \*temp = \*p;

Node \*r;

// If link list is empty then

// create first node and assign value. if (temp == NULL)

{

temp = new Node(); temp->row = row\_index; temp->col = col\_index; temp->data = x;

temp->next = NULL;

\*p = temp;

}

// If link list is already created

// then append newly created node else

{

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

r = new Node();

r->row = row\_index; r->col = col\_index;

r->data = x;

r->next = NULL; temp->next = r;

}

}

// Function prints contents of linked list

// starting from start

void printList(Node \*start)

{

Node \*ptr = start;

cout << "row\_position:"; while (ptr != NULL)

{

cout << ptr->row << " "; ptr = ptr->next;

}

cout << endl;

cout << "column\_position:"; ptr = start;

while (ptr != NULL)

{

cout << ptr->col << " "; ptr = ptr->next;

}

cout << endl; cout << "Value:"; ptr = start;

while (ptr != NULL)

{

cout << ptr->data << " "; ptr = ptr->next;}}

int main()

{

int sparseMatrix[4][5] = { { 0 , 0 , 3 , 0 , 4 },

{ 0 , 0 , 5 , 7 , 0 },

{ 0 , 0 , 0 , 0 , 0 },

{ 0 , 2 , 6 , 0 , 0 } };

// Creating head/first node of list as NULL Node \*first = NULL;

for(int i = 0; i < 4; i++){

for(int j = 0; j < 5; j++){

// Pass only those values which

// are non - zero

if (sparseMatrix[i][j] != 0) create\_new\_node(&first, i, j,sparseMatrix[i][j]);}}

printList(frst);

return 0;

}

