# INTRODUCTION TO DATA STRUCTURES & ALGORITHMS

Data structures provide a way to organize the data for your program in a way that is efficient and easy to use.

**Definition 1:** A **Data Structure** is a data object together with the relationships that exist among the instances & among the individual elements that compose an instance.

**Definition 2:** A **Data Structure** is an arrangement of data in a computer's memory or even disk storage. **Algorithms,** on the other hand, are used to manipulate the data contained in these data structures as in searching and sorting. An Algorithm is a finite set of instructions.

Commonly used algorithms are useful for:

* Searching for a particular data item (or record).
* Inserting a new data item (or record)
* Deleting a specific data item (or record)
* Sorting the data. (Arranging the elements in increasing/decreasing order)
* Iterating through all the items in a data structure. (Visiting each item in turn so as to display it or perform some other action on these items)

A data structure is classified into two categories: **Linear and Non-Linear data structures.**

**1. Linear Data Structures:**

A data structure is said to be linear if the elements form a sequence. It is sequential and continues in nature i.e. access the data in sequential manner.

In linear data structure we cannot insert an item in middle place and it main­tains a lin­ear rela­tion­ship between its ele­ments

There are two ways of representing linear data structures in memory. One way is to have the linear relationship between the elements by means of sequential memory locations. Such linear structures are called arrays. The other way is to have the linear relationship between the elements represented by means of links. Such linear data structures are called linked list.

**Ex: Arrays, Linked Lists, Stacks, Queues etc..**

**2. Non** - **Linear Data Structures:**

A data structure is said to be non-linear if elements do not form a sequence.(not sequential).

It does not main­tain any lin­ear rela­tion­ship between their ele­ments. Every data item is attached to several other data items in a way that is specific for reflecting relationships. The data items are not arranged in a sequential structure. **Ex: Trees, Graphs.**

A data structure is **linear** if every item is related with next and previous item and it is **non linear** if it is attached with many of the items in specific ways to reflect relationship.

**Abstract Data Types**

A **Data Type** is a collection of objects and a set of operations that act on those objects.

An **Abstract Data Type (ADT)** is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects are separated from the representation of the objects and the implementation of the operations.

An **abstract data type (ADT)** define s

* a State of an object and
* Operations that act on the object, possibly changing the state.

ADTs provide the benefits of abstraction, modularity and information hiding. There is a strict separation between the public interface, or specification, and the private implementation of the ADT. This separation facilitates correctness proofs of programs/algorithms that use entities of the ADT.

An Abstract Data Type can be thought of as a "description" of the data in the class and a list of operations that can be carried out on that data and instructions on how to use these operations.

Abstract Data Types: Any set or collection of instances is a data object.

integer = {0, +1, -1, +2, -2, +3, -3, …}

daysOfWeek = {S,M,T,W,Th,F,Sa}

Data object -- relationships that exist among instances and elements that comprise an instance.

**ARRAY AS AN ABSTRACT DATA TYPE**

An array is ordering of data elements so that information can be extracted from them. The size of an array depends on the number of rows and columns.

An array is usually implemented as an consecutive set of memory location. An array is a set of pairs <index,value>,such that each index that is defined has a value associated with it.

The constructor **GeneralArray**(int j,RangeList list,float initValue=defaultValue)produces a new array of the appropriate size and type. All of the items are initially set to the floating point variable initValue. **Retrieve** accept an index and return the value associated with the index if the index is valid or an error if the index is invalid. **Store** accepts an index, and a value of type float, and replaces the <index,old value>pair with the<index, new value> pair.

**class** GeneralArray

{

// A set of pairs <index,value> where for the each value of index in index set there is a //value of type **float**.index set is a finite ordered set of one or more dimensions ,for //example ,{0,…..,n-1}.

**public :**

GeneralArray (**int** j,RangeList list,**float** initValue=defaultValue);

//this constructor creates a j dimensional array of floats ;the range of the kth //dimension is given by kth element of list.for each index i in the index set ,insert //<i , initvalue>into the array.

**float** Retrieve(index i);

//if i is in the index set of the array, return the float associated with i in the //array;otherwise throw an exception.

**void** Store (index i,**float** x);

//if i is in the index set of the array , replace the old value associated with i by x;

//otherwise throw an exception.

};

**1. PROGRAM TO IMPLEMENT ARRAY AS AN ADT**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class array\_adt

{

int a,b[100],count,n;

public:

array\_adt();

void create();

void store(int,double);

void retrieve(int);

};

array\_adt::array\_adt()

{

count=0;

}

void array\_adt::create()

{

cout<<"enter the size of array"<<endl;

cin>>a;

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

b[i]=1;

cout<<"array has been created"<<endl;

}

void array\_adt::store(int index,double value)

{

if(index<a)

{

b[index]=value;

count++;

}

else

cout<<"index out of range";

}

void array\_adt::retrieve(int index)

{

if(b[index]==-1)

cout<<"no data is available at this index";

else

{

if(count==0)

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

else

cout<<"the value is at the index"<<index<<"is:"<<b[index];

}

}

void menu()

{

cout<<"\n 1.create 2.store 3.retrieve 4.exit"<<endl;

cout<<"enter your choice"<<endl;

}

void main()

{

int ch,i;

double v;

array\_adt q;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:q.create();

break;

case 2:cout<<"enter the index and the value you want to store"<<endl;

cin>>i>>v;

q.store(i,v);

break;

case 3:cout<<"enter the index whose value you want to check"<<endl;

cin>>i;

q.retrieve(i);

break;

case 4:exit(0);

}

menu();

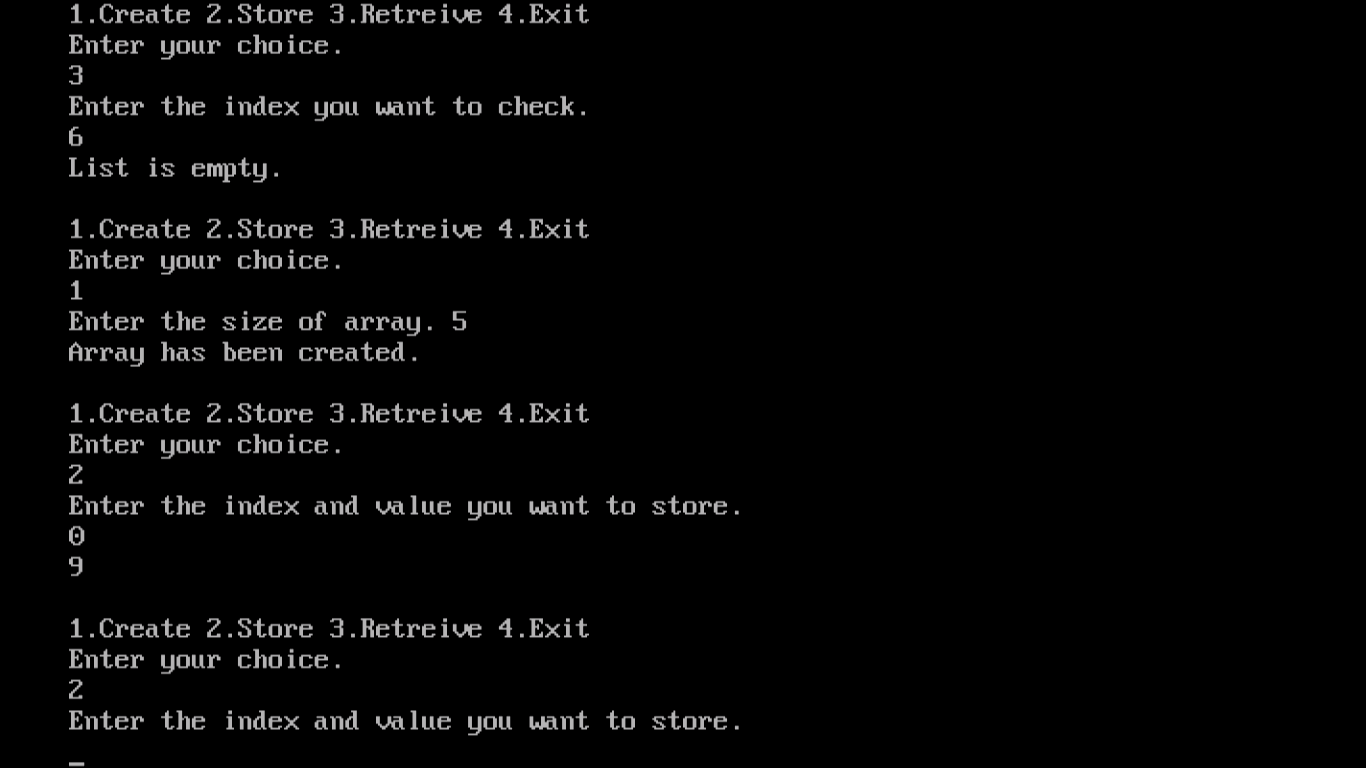
cin>>ch;

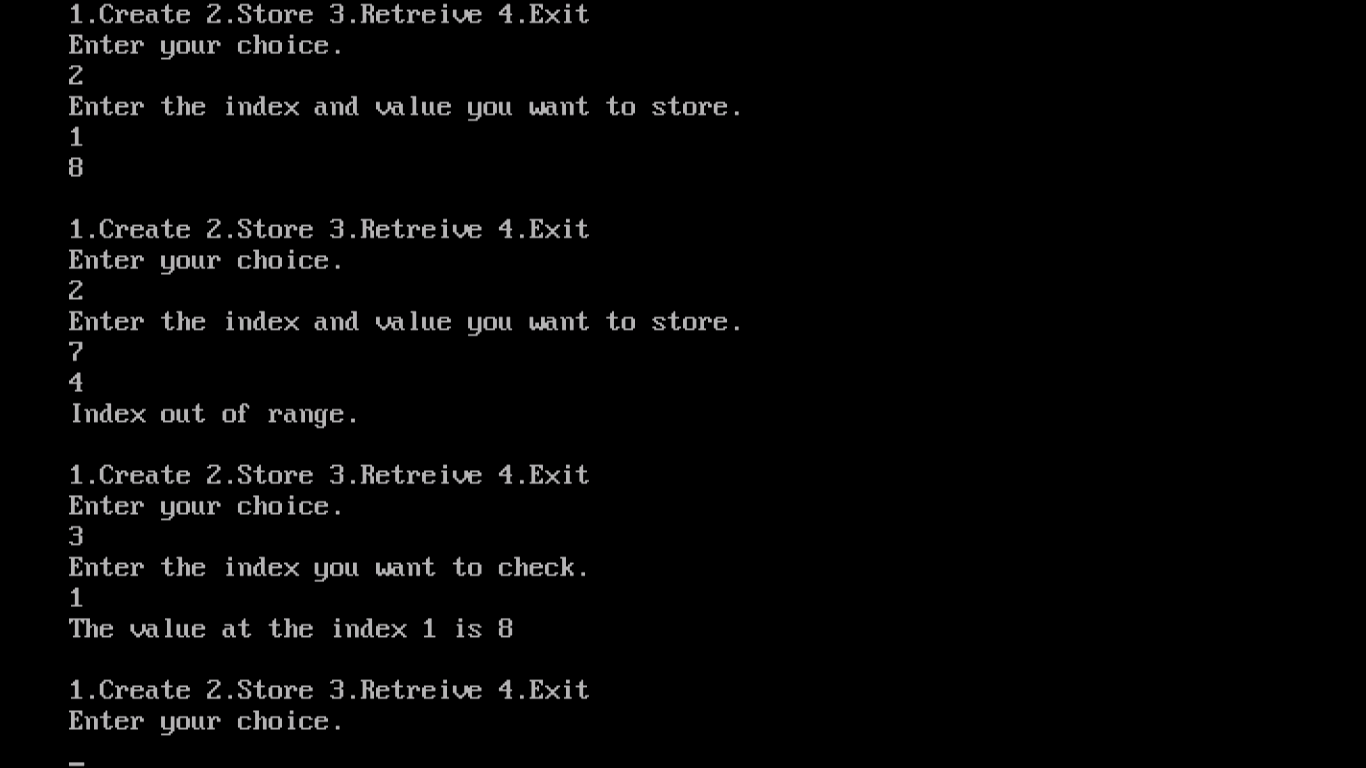
}

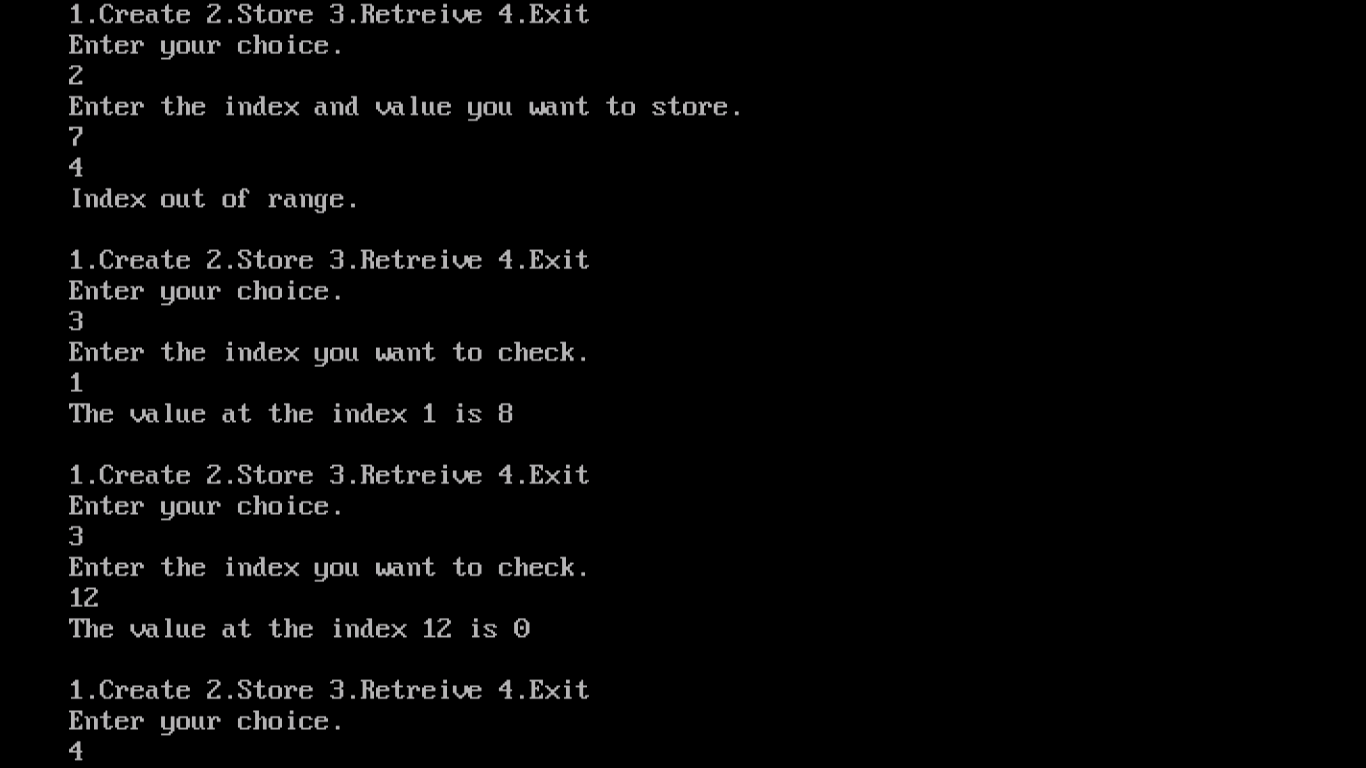
getch();

}

**OUTPUT**







**STRING AS AN ABSTRACT DATA TYPE**

A string contains the components elements as characters. As an ADT, a string is defined to have the form S=S 0 ,……….,S n-1 ,where S i are characters set of the programming language, and n is the length of the string. If n=0, then S is an empty or null string.

There are several useful operations that are specified for strings. Some of these operations are similar to those required for the other ADT’s: creating a new empty string ,reading a string or printing it out, appending two strings together (called concatenation),or copying a string. However, there are other operations that are unique to our new ADT, including comparing strings, inserting a sub string into a string, removing a sub string from a string, or finding a pattern in a string.

**class** String

{

**public:** String (**char** \*init, **int** m); //constructor that initializes \* **this** to string init of length m.

**int** Length (); //Return the number of characters in \***this.**

String Concat (String t);

//Return a string whose elements are those of \***this** followed by those of t.

**int** Find(String pat);

//Return an index i such that pat matches the sub string of \***this** that //begins at position i.

//Return -1 if pat is either empty or not a sub string of \***this.**

};

**String Pattern Matching: A Simple Algorithm**

Assume that we have two strings, s and pat, where pat is a pattern to be searched for in s. This algorithm determines if pat is in s by using the function Find. The invocation s.Find(pat) returns an index i such that pat matches the substring of s that begins at position i. It returns -1 if and only if pat is either empty or is not a substring of s.

Implementation of Find function:

The easiest way but least efficient method to determine whether pat is in s is to serially consider each position of s and determine if this position is the starting point of a match. Let lengthP and lengths denote the lengths of the pattern pat and the string s, respectively. Position of s to the right of position lengthS-lengthP need not be considered, as there are not enough characters to their right to complete a match with pat. Function Find implements this strategy.

The asymptotic complexity of the resulting pattern matching function is **Ο(lengthP . lengthS)**.

**2. PROGRAM TO IMPLEMENT STRING AS AN ADT**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

class string

{

char \*str1,\*str2;

public:

void concat();

void insert();

int find();

};

void string::concat()

{

char str[80];

cout<<"enter the first string"<<endl;

cin>>str1;

cout<<"enter the second string"<<endl;

cin>>str2;

strcpy(str,str1);

strcpy(str,str2);

cout<<"concatenation string is:"<<str<<endl;

}

void string::insert()

{

cout<<"enter 5 letter and any letter string"<<endl;

cin>>str1>>str2;

}

int string::find()

{

int length1=5,length2=2;

for(int start=0;start<=length1-length2;start++)

{

int j;

for(j=0;j<length2&&str1[start+j]==str2[j];j++)

{

if(j==length2-1)

{

cout<<"pattern matched at position:";

return start;

}

}

}

cout<<"pattern did not match!";

return -1;

}

void menu()

{

cout<<endl;

cout<<"enter your choice"<<endl;

cout<<"1.concatenate 2.pattern matching 3.exit"<<endl;

cout<<endl;

}

void main()

{

int ch;

string s;

clrscr();

menu();

cin>>ch;

while(ch!=4)

{

switch(ch)

{

case 1:s.concat();

break;

case 2:s.insert();

cout<<s.find();

break;

case 3:exit(0);

}

menu();

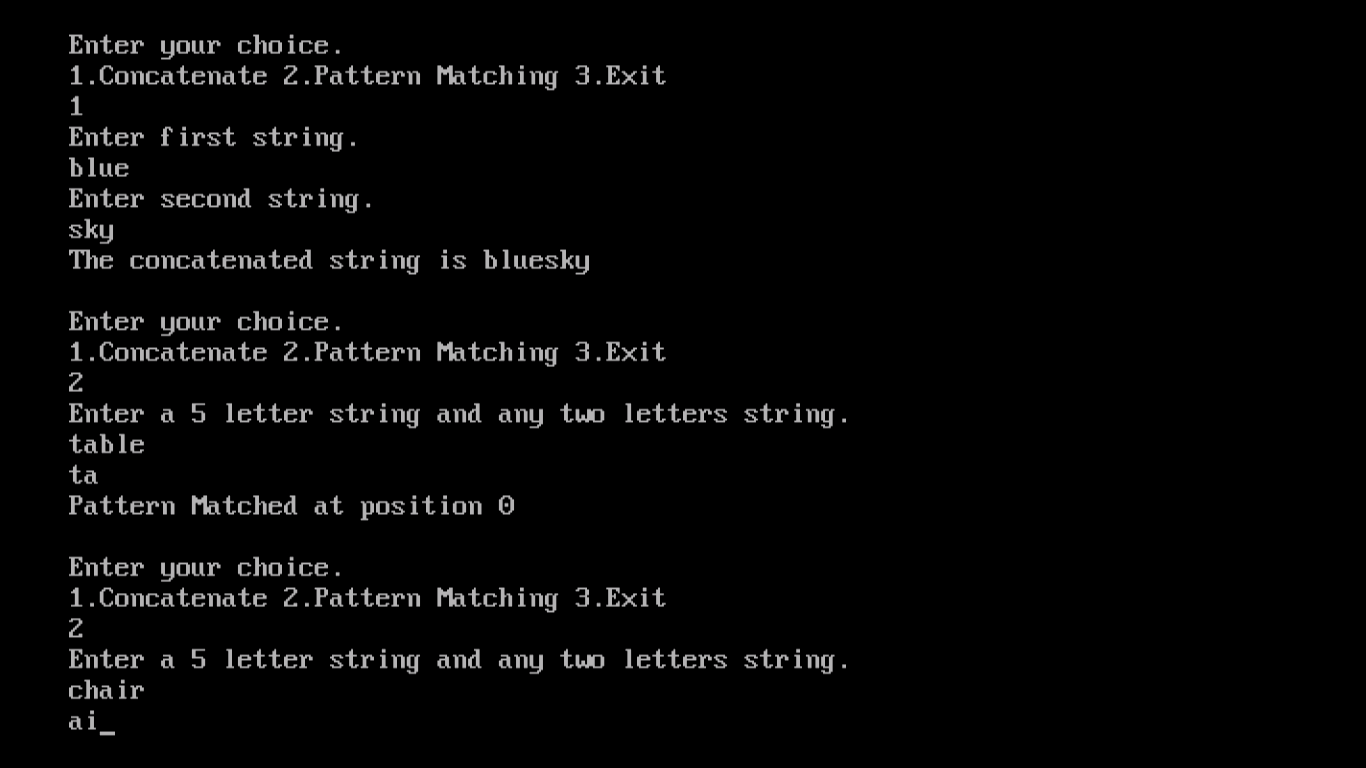
cin>>ch;

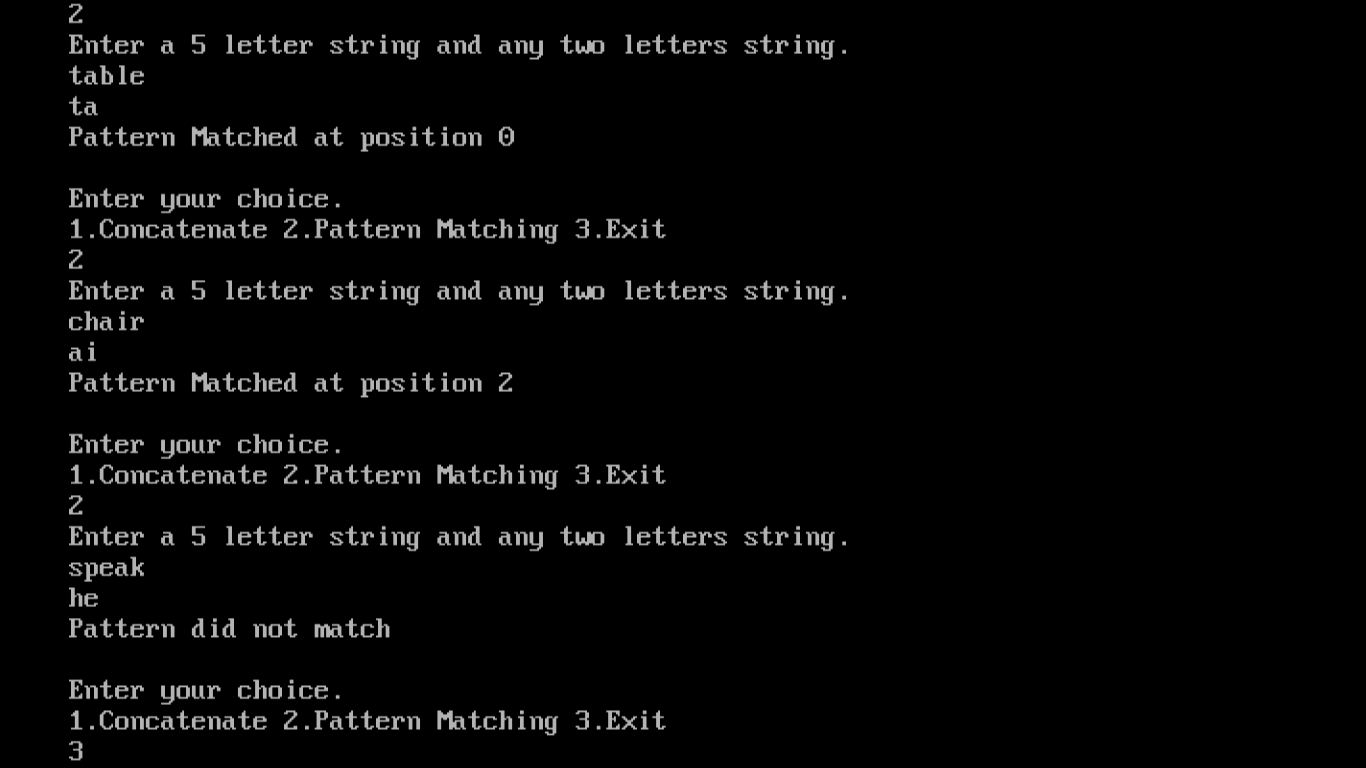
}

getch();

}

**OUTPUT**





**Stacks**

Stacks are common data structures, and are commonly used in many different algorithms.

In stack, when you add a new item, it goes on the top of a stack, and when you remove an item, you remove it from the top. This behavior is called "last in, first out," or LIFO, and makes many algorithms and data collections much easier to work with.

Definition: Stack is an ordered collection of items into which items can be inserted and deleted from only one end called as “Top” of the stack. It is also called as LIFO list(Last In First Out).

Stack implementation provides three main functions: Push, which puts a new item on the top of the stack; Pop, which removes an item from the top of the stack and Top , which examines the element on the top of the stack.

Stack Underflow: If Stack is empty and Pop operation is performed it is not possible to delete the items. This situation is called Stack Underflow.

Stack Overflow: If Stack is full and Push operation is performed it is not possible to insert or Push the new items into the stack. This situation is called Stack Overflow.

Standard operations:

IsEmpty … return true iff stack is empty

Top … return top element of stack

Push … add an element to the top of the stack

Pop … delete the top element of the stack

A stack is a method used to insert and delete items from a linear list. The concept of a stack is of fundamental importance in computing as it is used in so many different applications, and the principle of a stack is illustrated.

E.g. The numbers in the list: 23, 54, 10 & 90.

If the numbers were set out vertically, the list would look like: 23

54

10

90

 If 77 was added to the stack when it is *pushed* on top of the stack. The stack now looks like this:

77

23

54

10

90

If an item is to be removed, is it said to be *popped off the stack* in the \'last number in first number out\' (LIFO - last in first out).

Applications of stack:

1. Stack are used to convert infix expression to postfix expression

2. Stacks are used to evaluate postfix expression.

3. Stack is used in recursion etc.

**3. PROGRAM TO IMPLEMENT STACK AS AN ARRAY**

#include<stdio.h>

#include<conio.h>

#include<iostream.h>

#include<stdlib.h>

template <class T>

class linearstack

{

T top;

T a[5];

public:

linearstack();

T topele();

T push(T x);

T pop();

void display();

};

//function to increament the position of the stack

template<class T>

T linearstack<T>::linearstack()

{

top=-1;

}

//function to return the top element of the stack

template<class T>

T linearstack<T>::topele()

{

if(top==-1)

return 0;

else

return a[top];

}

//function to inset the ne elements in the stack

template<class T>

T linearstack<T>::push(T x)

{

if(top==4)

cout<<"the stack is overflow"<<endl;

else

{

top=top+1;

a[top]=x;

cout<<x<<"is inserted"<<endl;

}

return 0;

}

//function to delete the element from the stack

template<class T>

T linearstack<T>::pop()

{

T x;

if(top==-1)

cout<<"the stack is underflow"<<endl;

else

{

x=a[top];

top=top-1;

cout<<x<<"is deleted"<<endl;

}

return 0;

}

//function to display the elements of the stack

template<class T>

void linearstack<T>::display()

{

if(top==-1)

cout<<"stack is empty"<<endl;

else

{

cout<<"the stack elements are:"<<endl;

for(T x=top;x>=0;x--)

cout<<a[x]<<endl;

}

}

//user options

void menu()

{

cout<<endl;

cout<<"1.top element 2.push 3.pop 4.display 5.exit"<<endl;

cout<<"enter your choice"<<endl;

}

//main function

void main()

{

linearstack<int>s;

int te,e,ch;

clrscr();

menu();

cin>>ch;

while(ch!=6)

{

switch(ch)

{

case 1:te=s.topele();

cout<<"the top element is :"<<te<<endl;

case 2:cout<<"enter the element to be inserted"<<endl;

cin>>e;

s.push(e);

break;

case 3:s.pop();

break;

case 4:s.display();

break;

case 5:exit(0);

break;

}

menu();

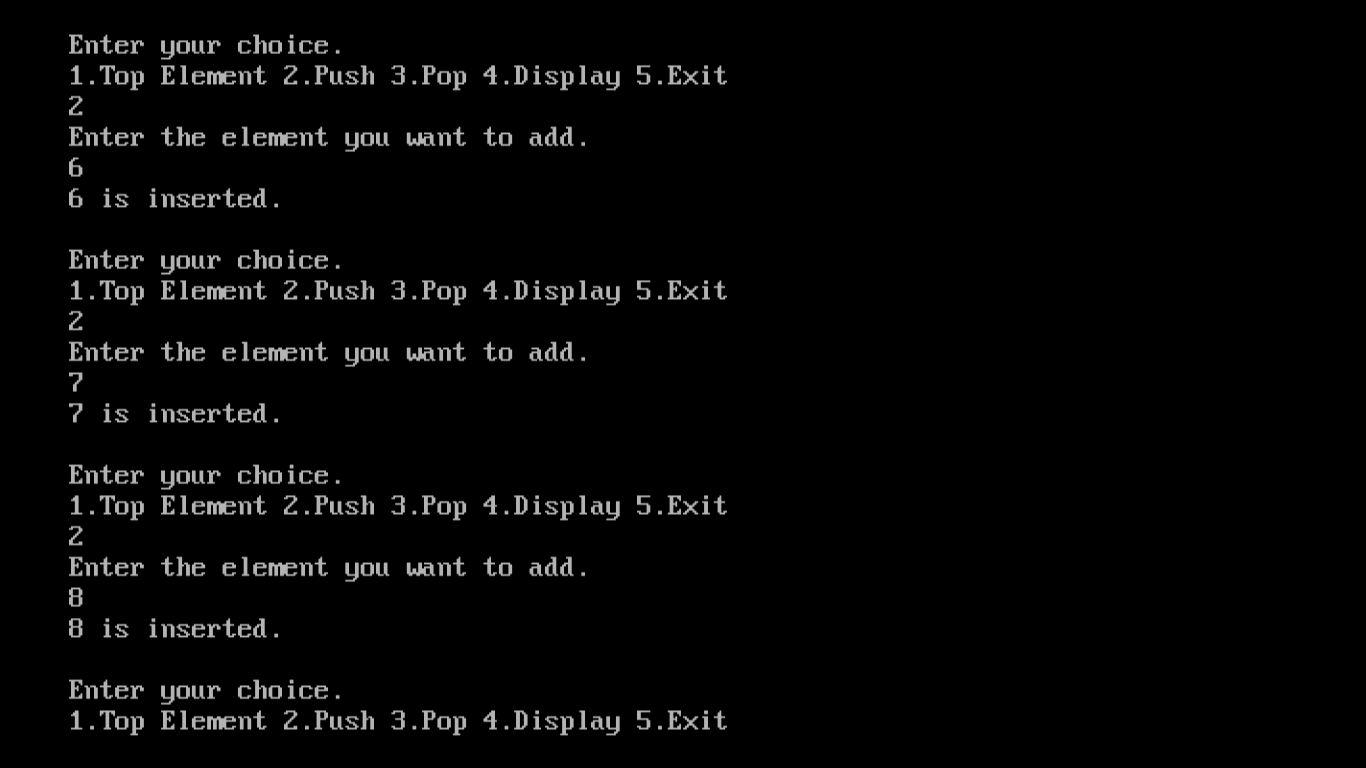
cin>>ch;

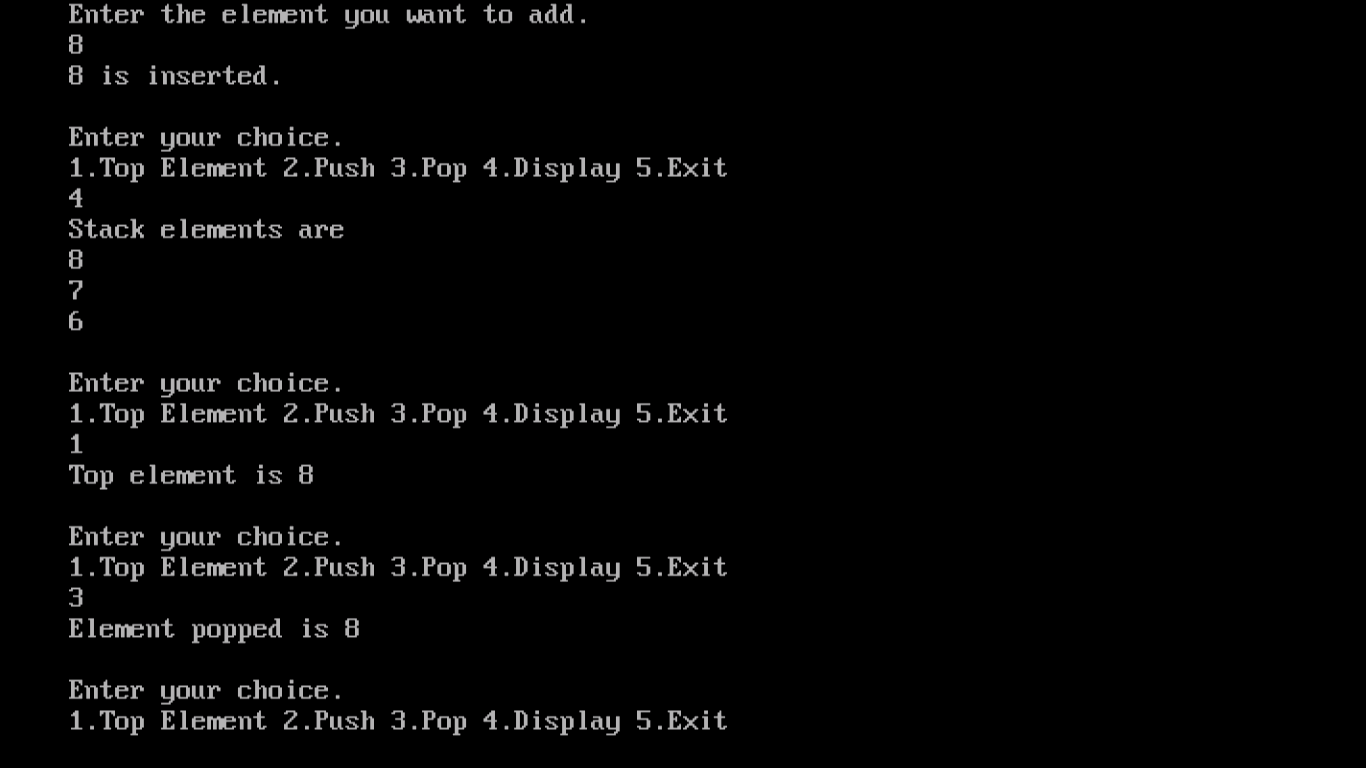
}

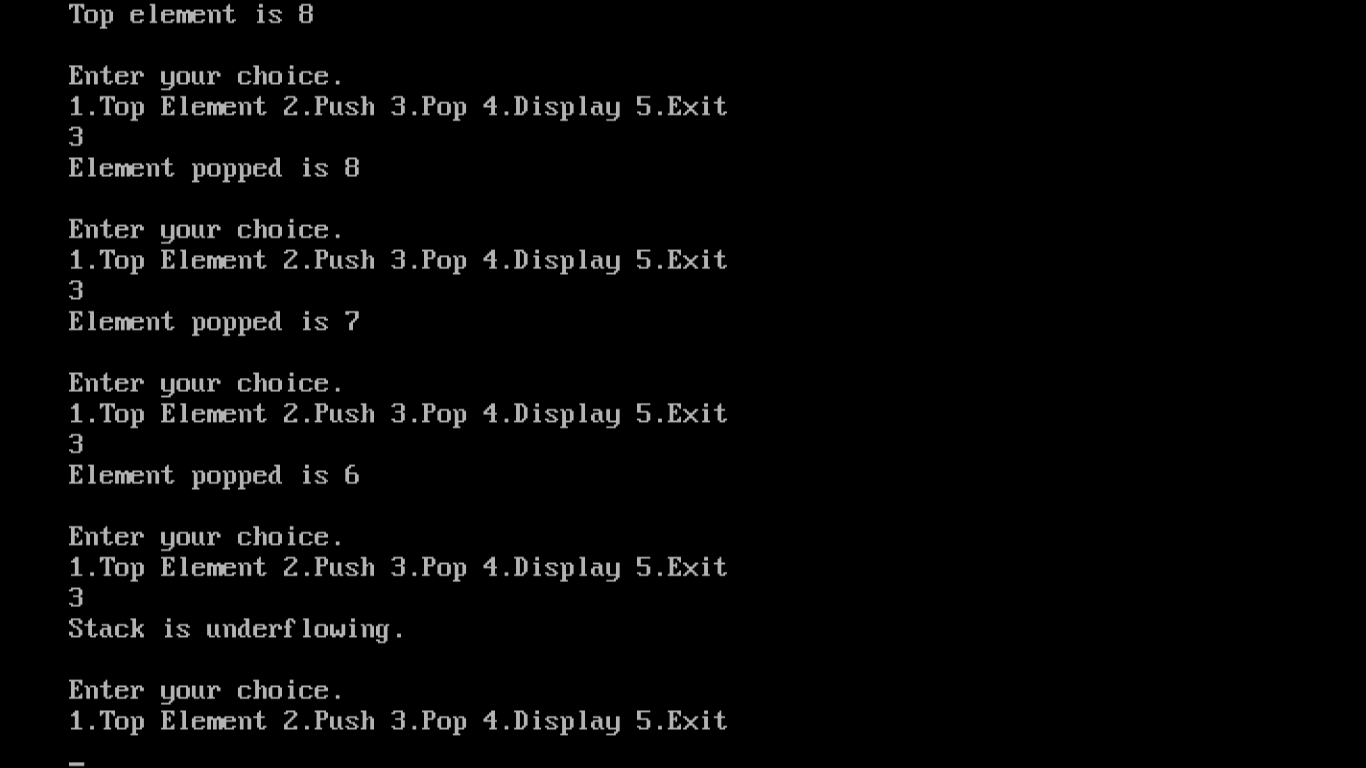
getch();

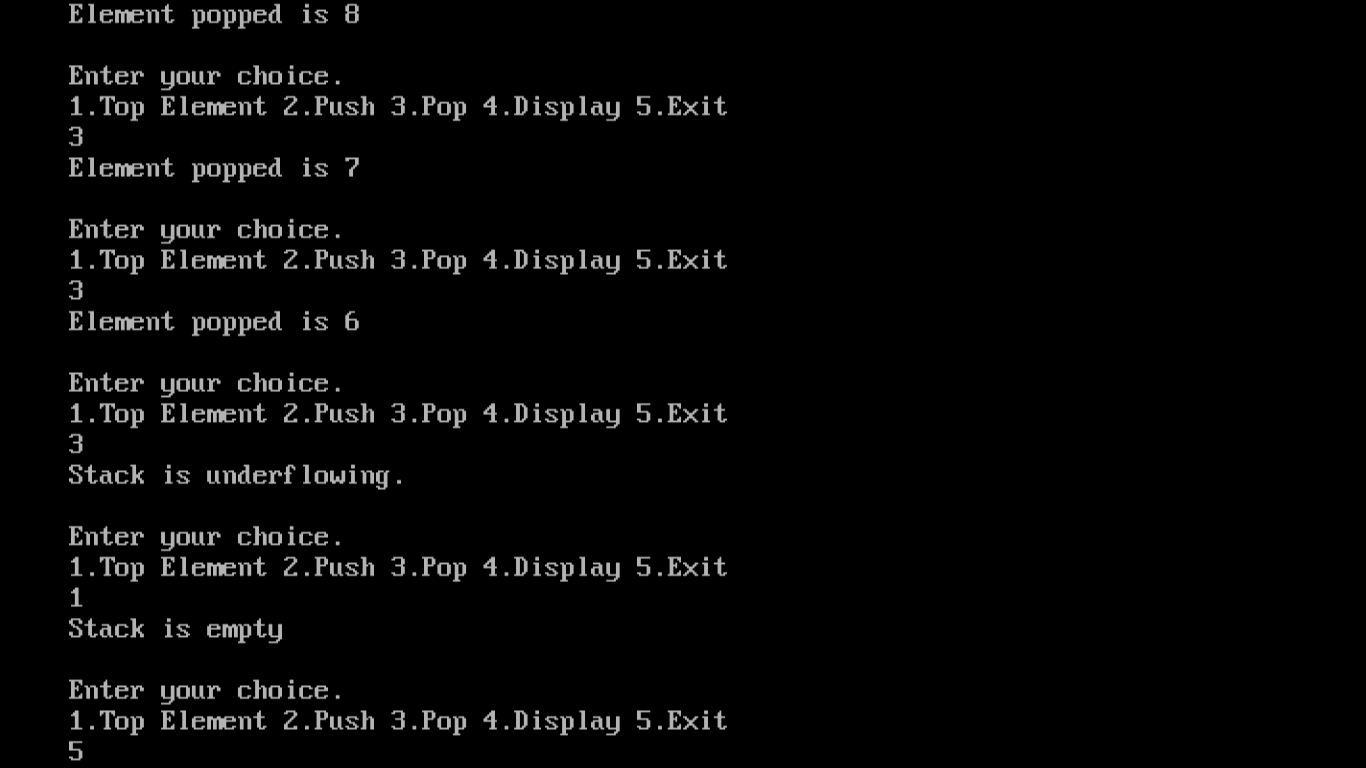
}

**OUTPUT**









**QUEUES**

QUEUE is an ordered collection of items into which items can be inserted from one end called as Rear end and items are deleted from other end called as Front end of the Queue. It is also called as First In First Out(FIFO).

The difference is that stacks provide last-in, first-out data storage, whereas queues are first in, first out (FIFO).

The Queue functions are Enqueue, Dequeue, Front and Rear. Enqueue adds an element to the end of the queue, Dequeue removes an element from the front of the queue, Front examines the next element to be de-queued and Rear examines the element that was recently en-queued.

A queue is very similar I principle to how a stack operates. A queue is often called a FIFO stack (First in First out). The operation of the queue is the same as the operation of a \'normal\' queue. (if you were first into a shop you would get server first).

When the data has been processed and the first operation has been used (start pointer) the stack does not shift up, just the pointers are moved. This therefore acts as a circular list, so when all the items have been popped and some more pushed on, the procedure is started again from the top using three pointers.

Applications of queues:

1. Queues are used in breath first traversal of tree.

2. Queues are used in implementation of scheduling algorithms of operating system.

**4. PROGRAM TO IMPLEMENT** **QUEUE AS AN ARRAY**

#include<stdio.h>

#include<conio.h>

#include<iostream.h>

#include<stdlib.h>

template<class T>

class linearqueue

{

T front;

T rear;

T a[5];

public:

linearqueue();

T enque(T x);

T deque();

void display();

};

template<class T>

T linearqueue<T>::linearqueue()

{

front=0;

rear=-1;

}

template<class T>

T linearqueue<T>::enque(T x)

{

if(rear==4)

cout<<"the queue is full"<<endl;

else

{

rear=rear+1;

a[rear]=x;

cout<<x<<"is inserted"<<endl;

}

return 0;

}

template<class T>

T linearqueue<T>::deque()

{

if(front>rear)

cout<<"the queue is empty"<<endl;

else

{

int x=a[front];

front=front+1;

cout<<x<<"is deleted"<<endl;

}

return 0;

}

template<class T>

void linearqueue<T>::display()

{

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

{

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

}

}

void menu()

{

cout<<endl;

cout<<"1.enque 2.deque 3.display 4.exit"<<endl;

cout<<endl;

}

void main()

{

linearqueue<int>q;

int e,ch;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:cout<<"enter the element to be inserted"<<endl;

cin>>e;

q.enque(e);

break;

case 2:q.deque();

break;

case 3:cout<<"the queue elements are:"<<endl;

q.display();

break;

case 4:exit(0);

break;

}

menu();

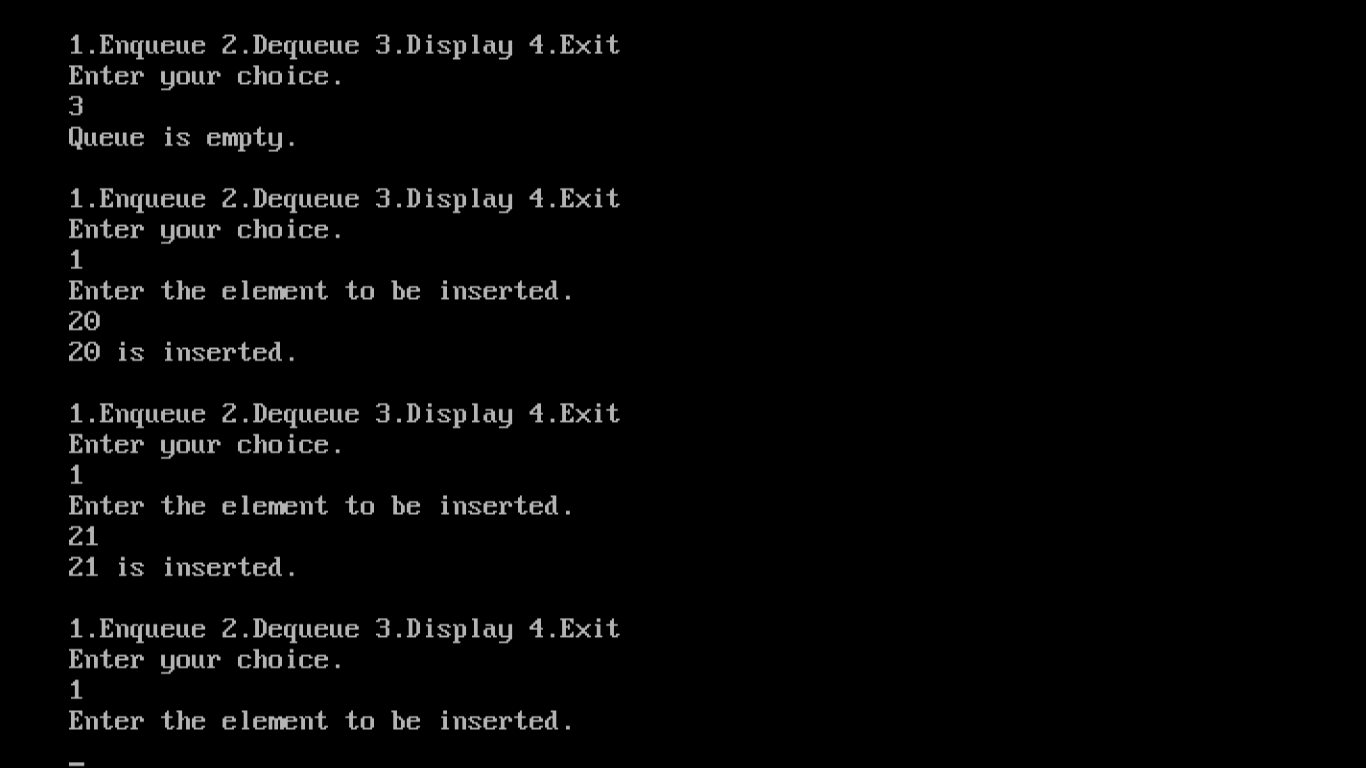
cin>>ch;

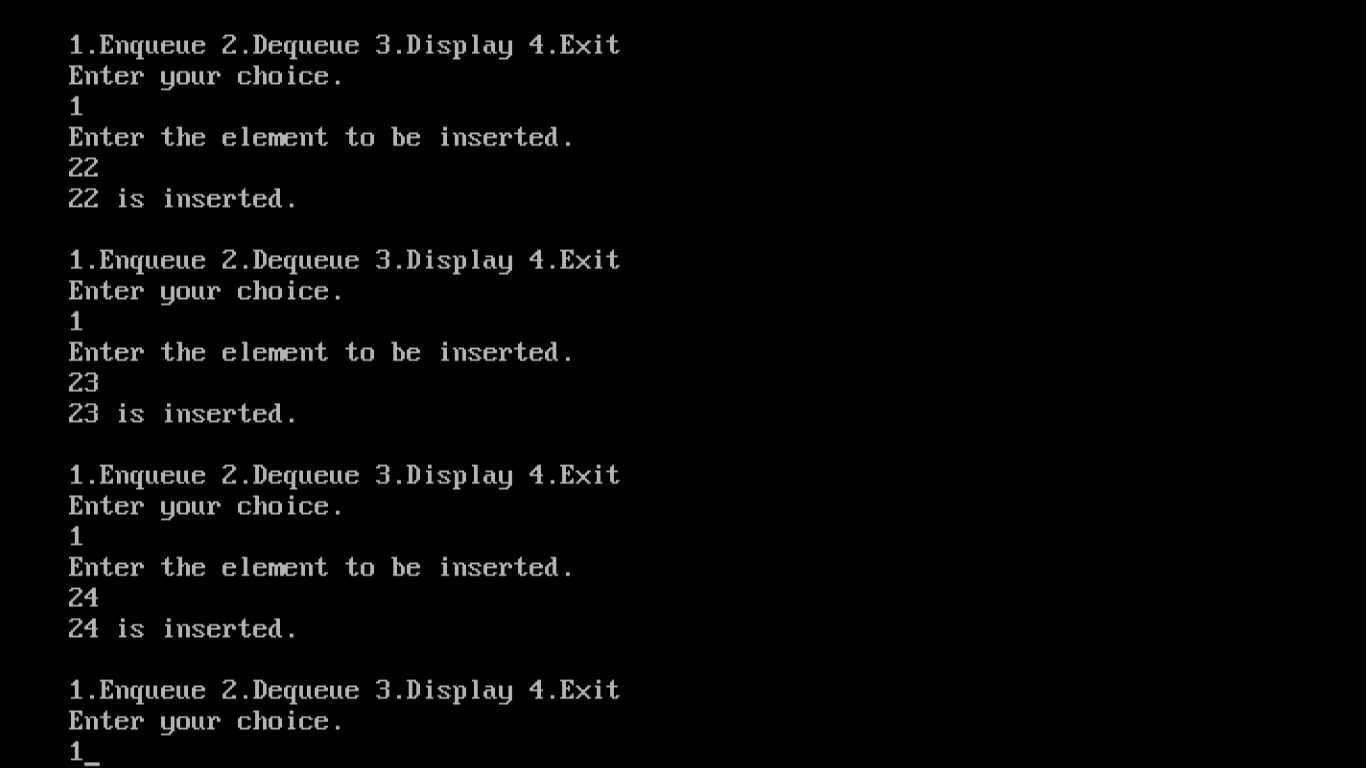
}

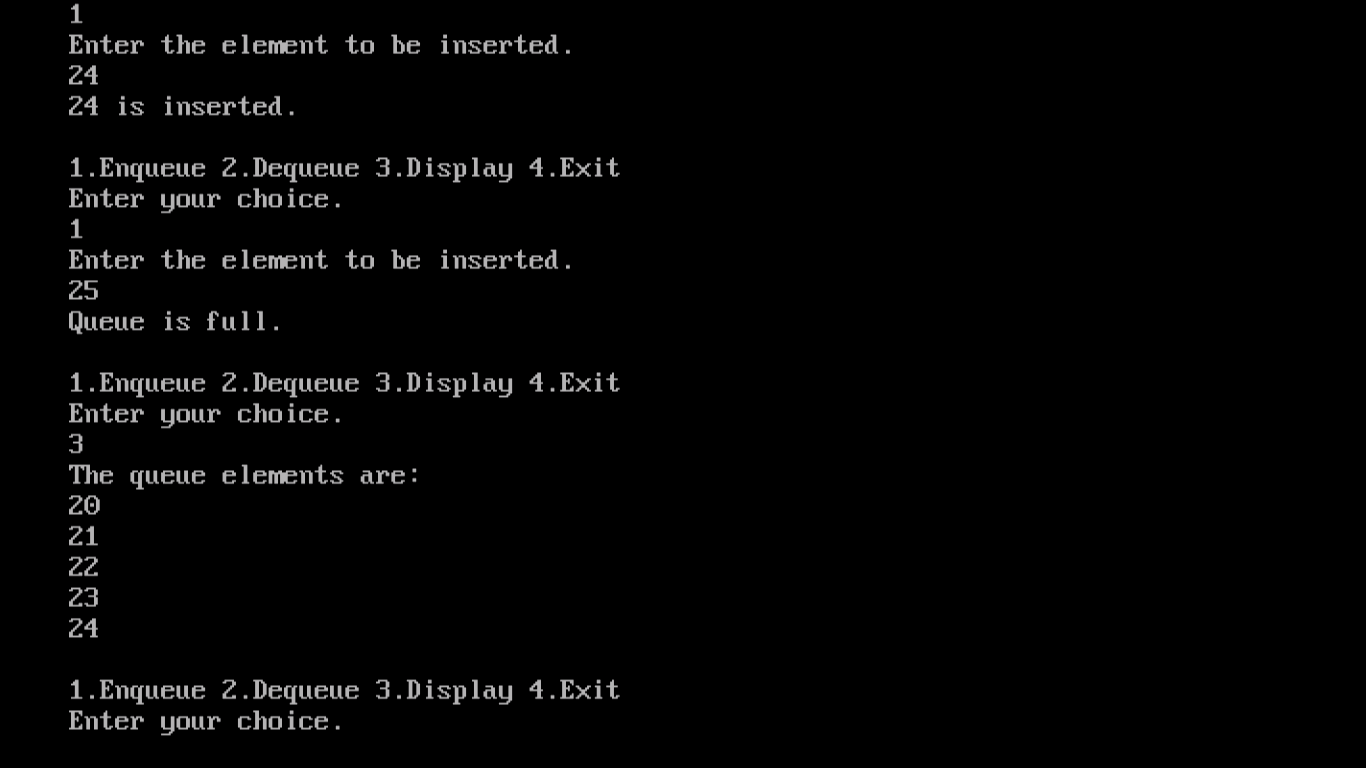
getch();

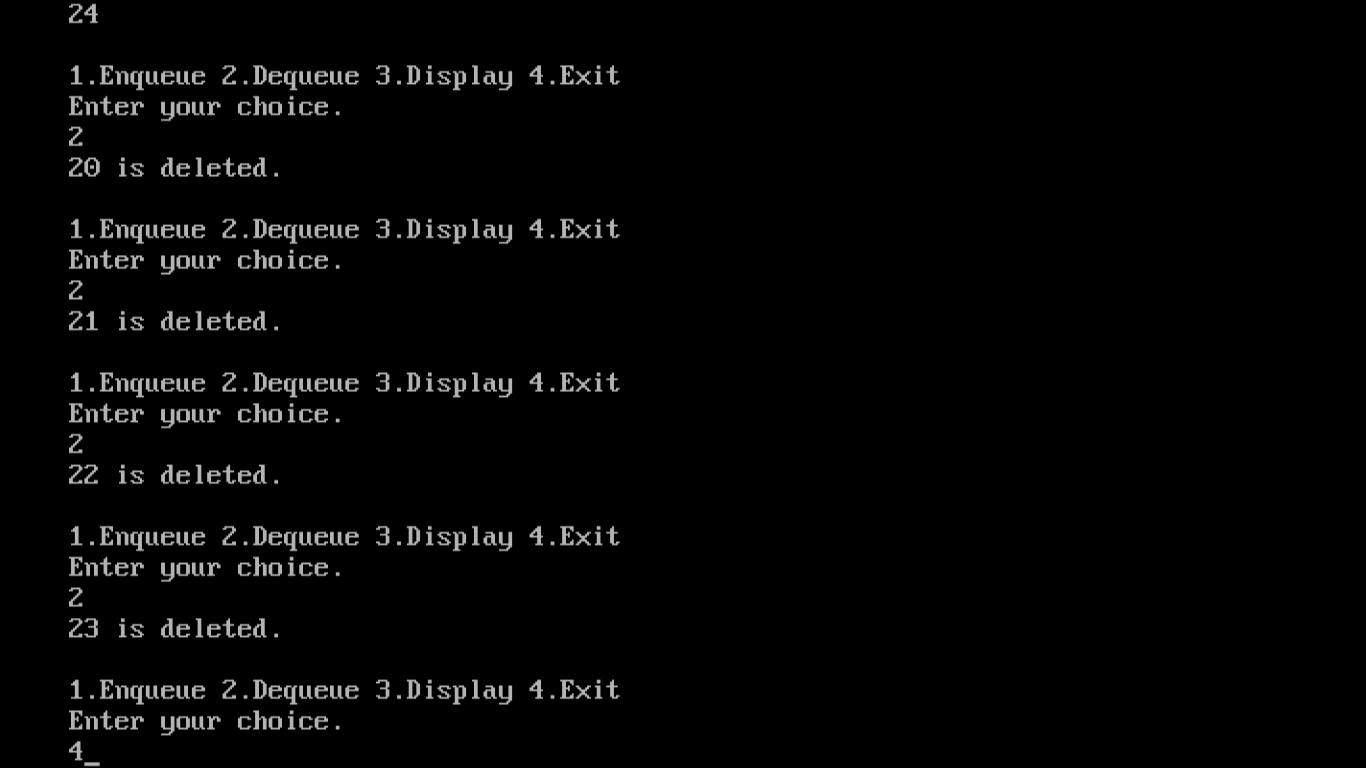
}

**OUTPUT**

****

****

****

****

**5. PROGRAM TO IMPLEMENT STACK AS A LINKED LIST**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

template<class T>

class stackelement

{

T data;

stackelement<T> \*next;

friend linkedstack<T>;

};

template<class T>

class linkedstack

{

stackelement<T> \*top;

public:

linkedstack();

T topele();

T push(T &);

T pop();

void display();

};

template<class T>

T linkedstack<T>::linkedstack()

{

top=0;

}

//function to return the top element of the stack

template<class T>

T linkedstack<T>::topele()

{

if(top==0)

return 0;

else

return top->data;

}

//function to insert the elements in the stack

template<class T>

T linkedstack<T>::push(T &x)

{

stackelement<T> \*p=new stackelement<T>;

p->data=x;

p->next=top;

top=p;

cout<<x<<"is pushed into the stack"<<endl;

return p->data;

}

//function to delete the elements from the stack

template<class T>

T linkedstack<T>::pop()

{

stackelement<T> \*temp=top;

if(top==0)

cout<<"the stack is underflow"<<endl;

else

{

top=top->next;

cout<<temp->data<<"is deleted from the stack"<<endl;

delete temp;

}

return 0;

}

//function to display all the elements of the stack

template<class T>

void linkedstack<T>::display()

{

stackelement<T> \*current;

if(top==0)

cout<<"the stack is empty"<<endl;

else

{

cout<<"the stack elements are:"<<endl;

for(current=top;current!=0;current=current->next)

cout<<current->data<<endl;

}

}

//user options

void menu()

{

cout<<endl;

cout<<"1.top element 2.push 3.pop 4.display 5.exit"<<endl;

cout<<"enter your choice"<<endl;

}

//main function

void main()

{

linkedstack<int>s;

int te,e,ch;

clrscr();

menu();

cin>>ch;

while(ch!=6)

{

switch(ch)

{

case 1:te=s.topele();

cout<<"the top element is:"<<te<<endl;

break;

case 2:cout<<"enter the elements to be inserted"<<endl;

cin>>e;

s.push(e);

break;

case 3:s.pop();

break;

case 4:s.display();

break;

case 5:exit(0);

}

menu();

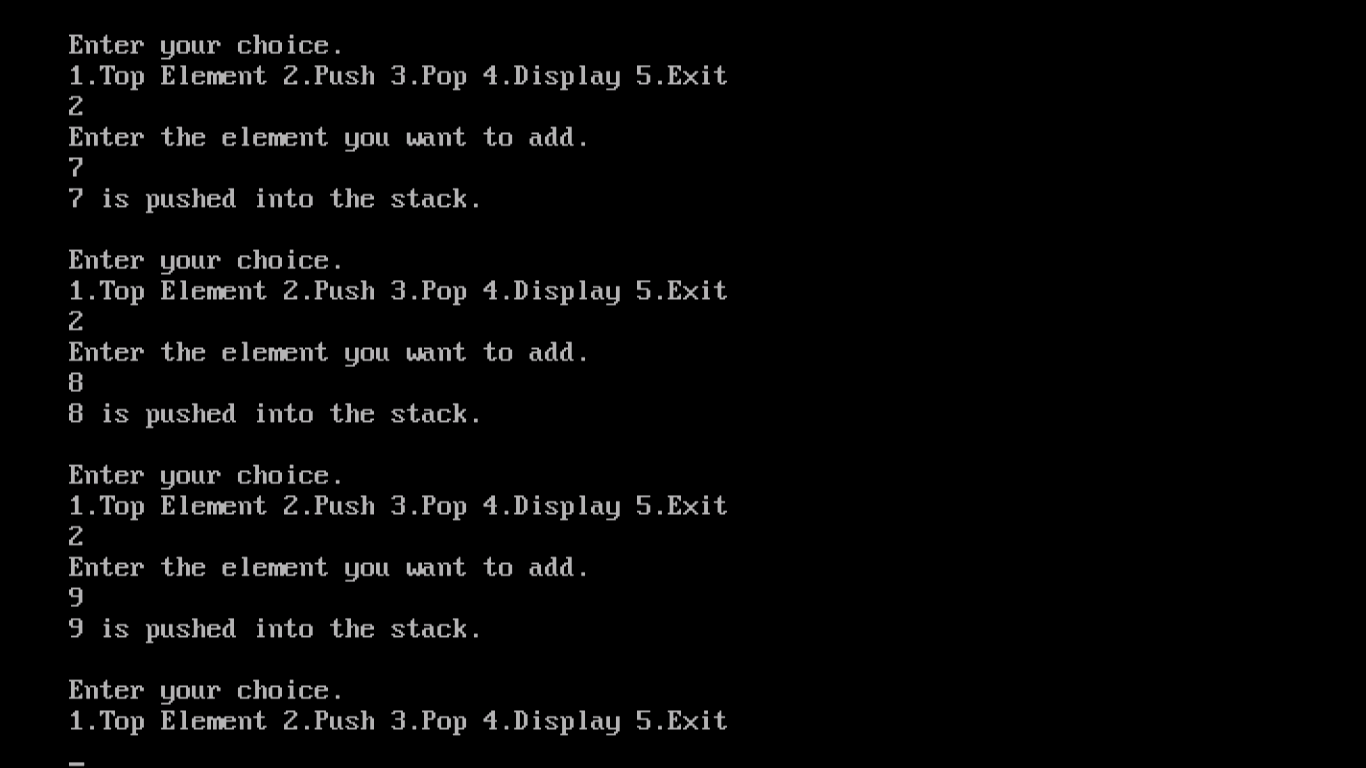
cin>>ch;

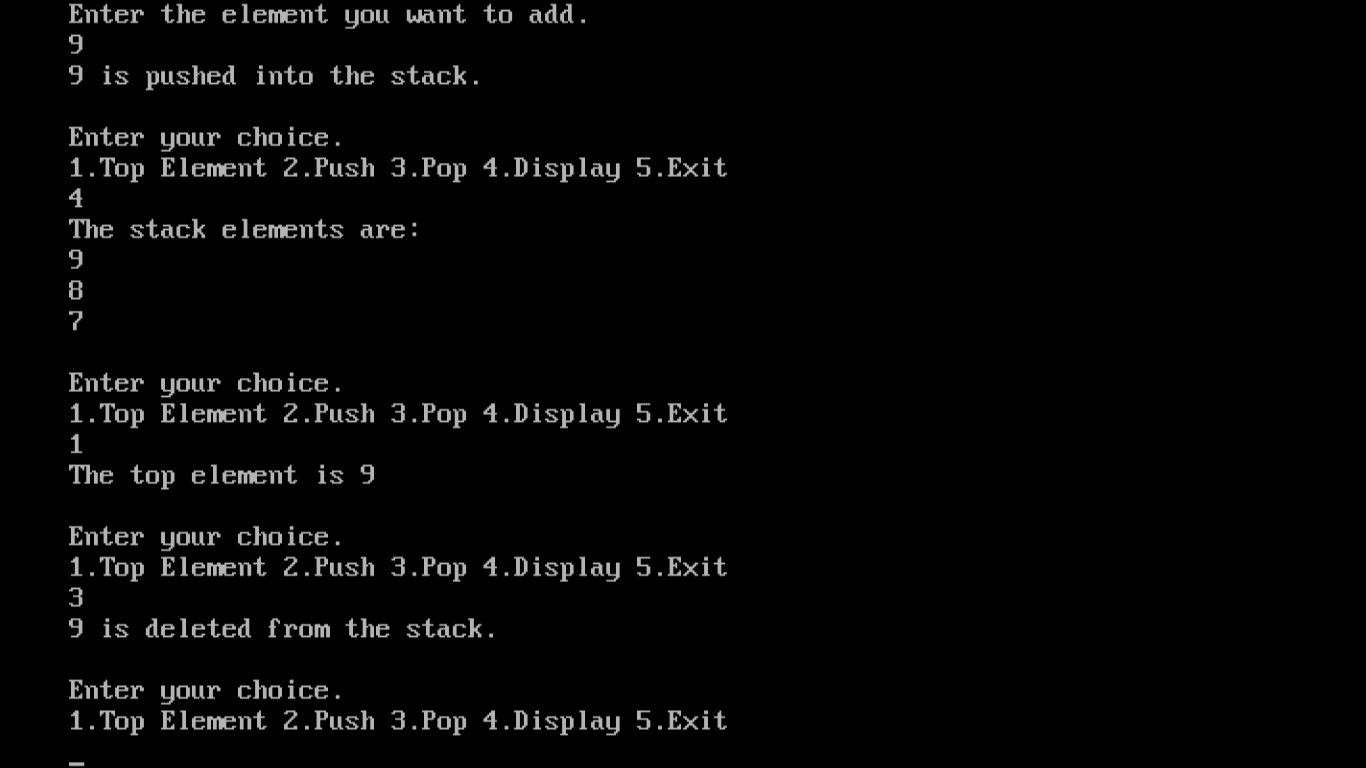
}

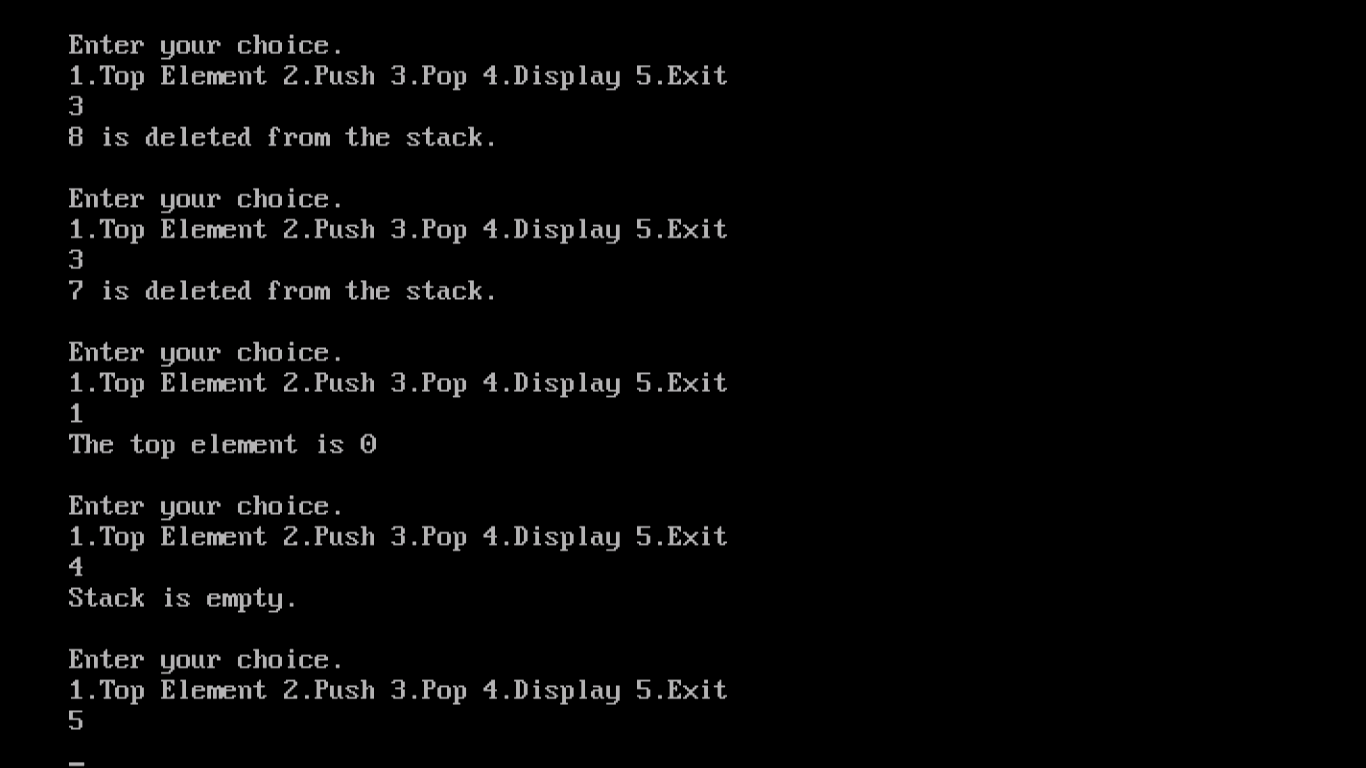
getch();

}

**OUTPUT**







**6. PROGRAM TO IMPLEMENT QUEUE AS A LINKED LIST**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

template<class T>

class queueelement

{

T data;

queueelement<T> \*next;

friend linkedqueue<T>;

};

template<class T>

class linkedqueue

{

queueelement<T> \*front;

queueelement<T> \*rear;

public:

linkedqueue();

T frontele();

T rearele();

T enque(T &);

T deque();

void display();

};

template<class T>

T linkedqueue<T>::linkedqueue()

{

front=0;

rear=0;

}

template<class T>

T linkedqueue<T>::frontele()

{

if(front==0)

return 0;

else

return front->data;

}

template<class T>

T linkedqueue<T>::rearele()

{

if(rear==0)

return 0;

else

return rear->data;

}

template<class T>

T linkedqueue<T>::enque(T &x)

{

queueelement<T> \*p=new queueelement<T>;

p->data=x;

p->next=0;

rear->next=p;

if(front==0)

front=p;

rear=p;

cout<<x<<"is inserted"<<endl;

return p->data;

}

template<class T>

T linkedqueue<T>::deque()

{

if(front==0)

cout<<"the queue is empty"<<endl;

else

{

queueelement<T> \*temp;

temp=front;

front=front->next;

cout<<temp->data<<"is deleted"<<endl;

delete temp;

}

return 0;

}

template<class T>

void linkedqueue<T>::display()

{

queueelement<T> \*current;

if(front==0)

cout<<"queue is empty"<<endl;

else

{

cout<<"the queue element are:"<<endl;

for(current=front;current!=0;current=current->next)

cout<<current->data<<endl;

}

}

void menu()

{

cout<<endl;

cout<<"1.front element 2.rear element 3.enque 4.deque 5.display 6.exit"<<endl;

cout<<"enter your choice"<<endl;

}

void main()

{

linkedqueue<int>q;

int fe,re,e,ch;

clrscr();

menu();

cin>>ch;

while(ch!=7)

{

switch(ch)

{

case 1:fe=q.frontele();

cout<<"the first element is:"<<fe<<endl;

break;

case 2:re=q.rearele();

cout<<"the last element is:"<<re<<endl;

break;

case 3:cout<<"enter the elment to be inserted "<<endl;

cin>>e;

q.enque(e);

break;

case 4:q.deque();

break;

case 5:q.display();

break;

case 6:exit(0);

}

menu();

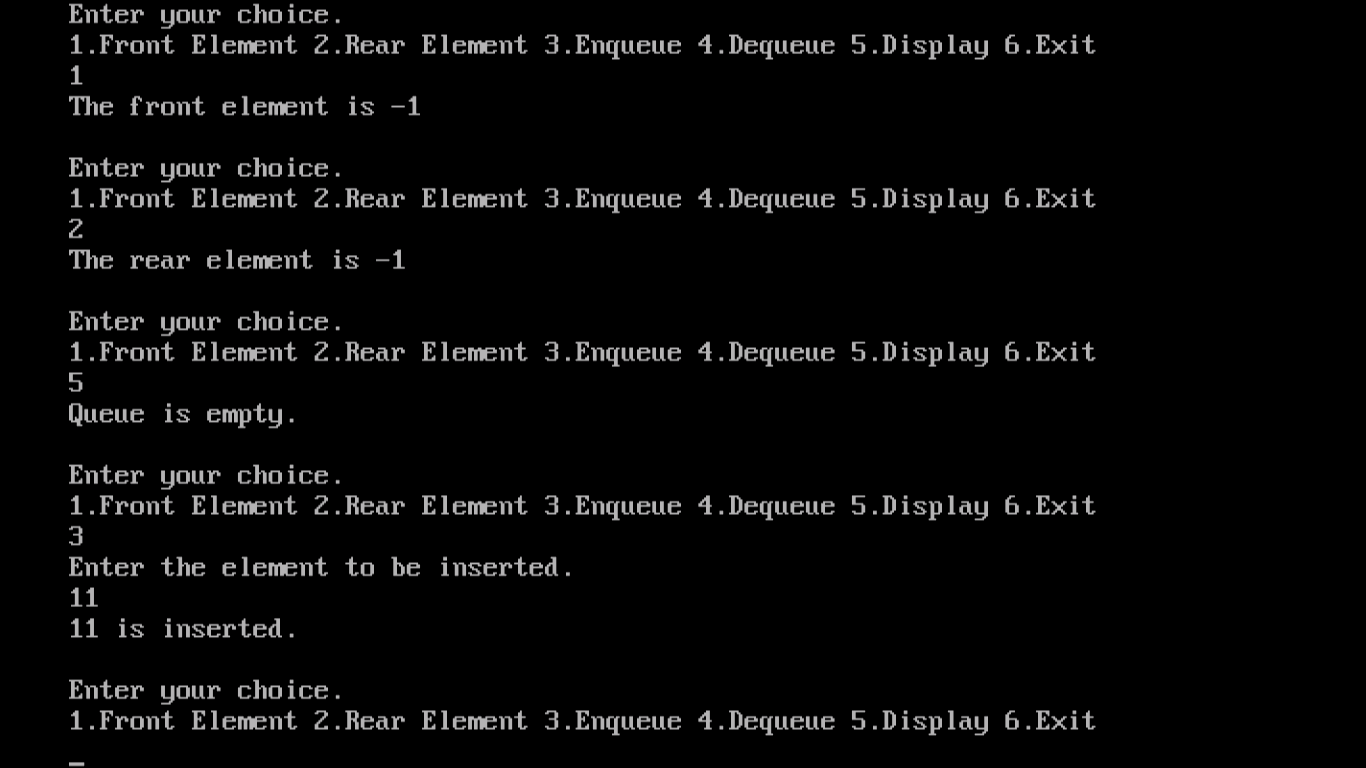
cin>>ch;

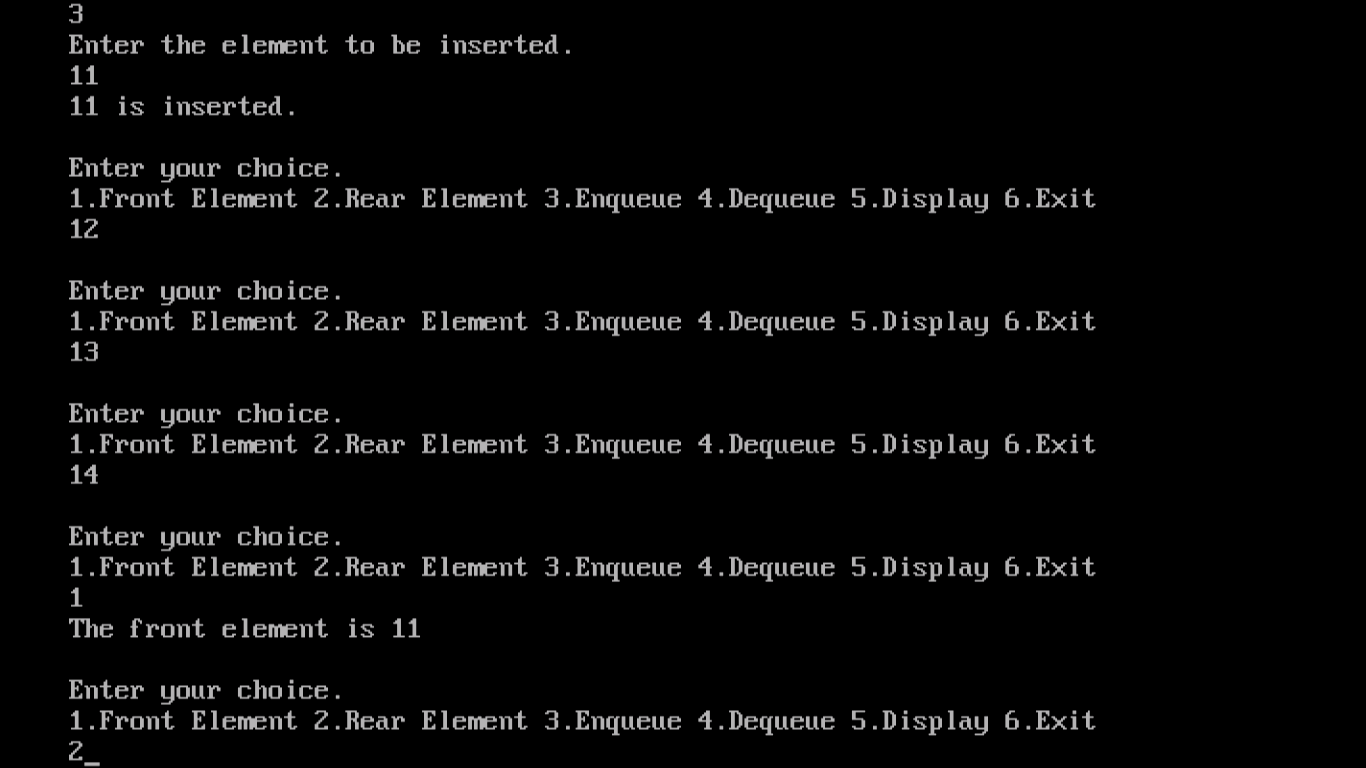
}

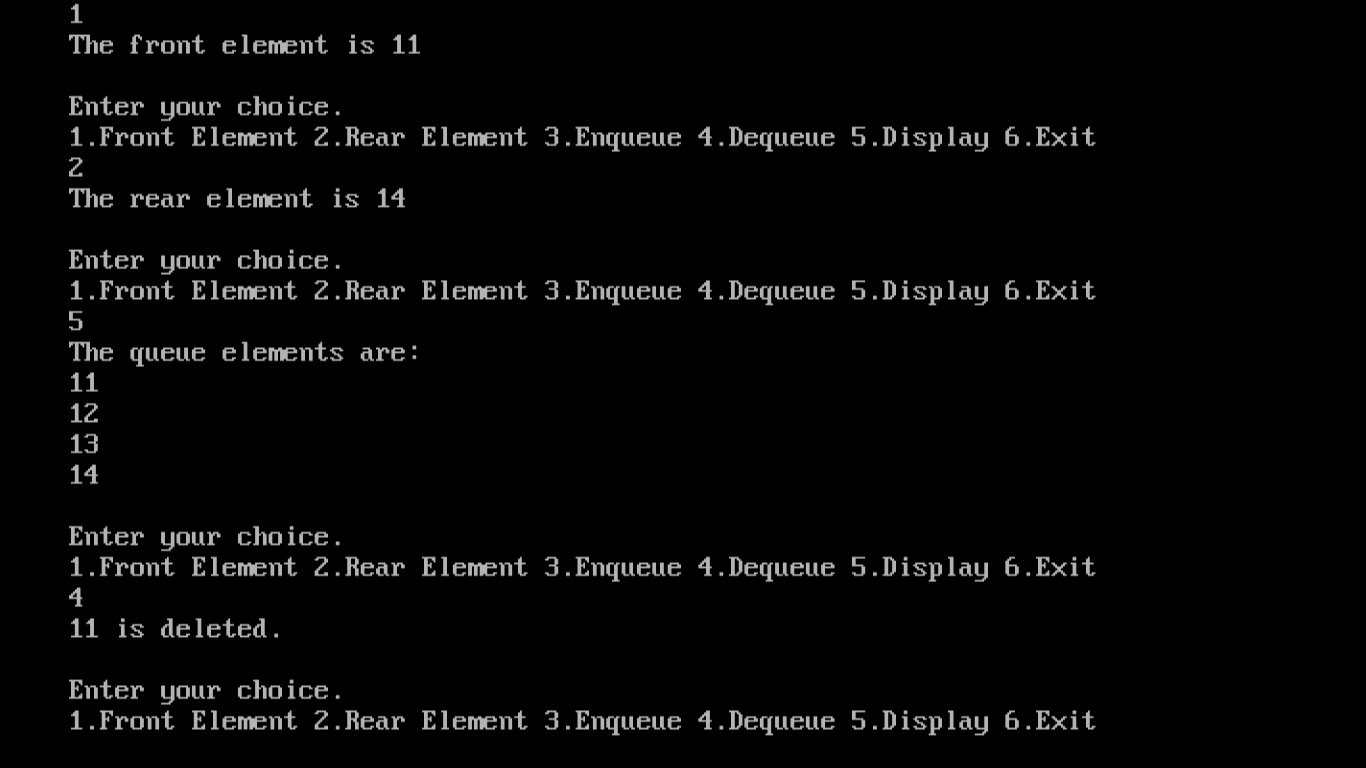
getch();

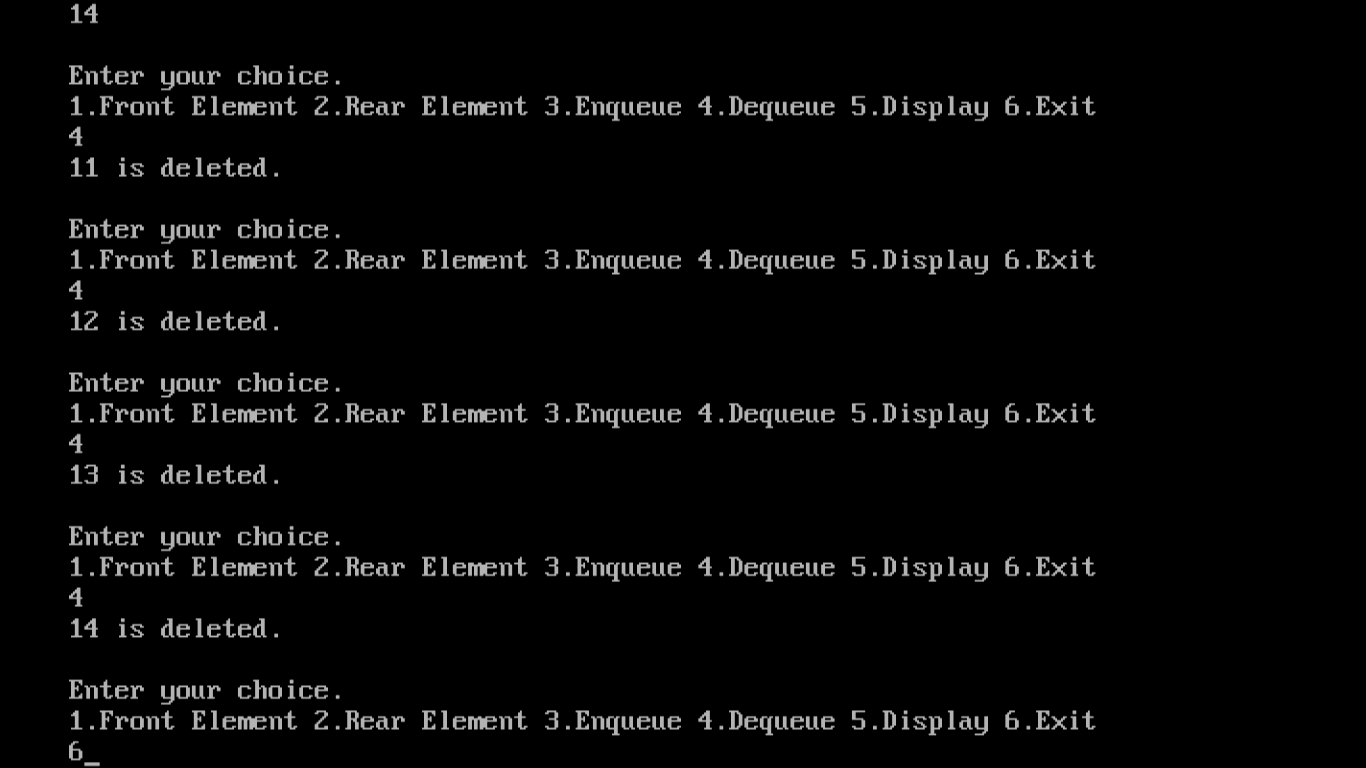
}

**OUTPUT**

****

****

****

****

**INFIX TO POSTFIX CONVERSION**

Infix Expression: Any expression in the standard form like "2\*3-4/5" is an Infix (Inorder) expression.

Postfix Expression: The Postfix (Postorder) form of the above expression is "23\*45/-".  
   
  
In normal algebra we use the infix notation like a+b\*c. The corresponding postfix notation is abc\*+. The algorithm for the conversion is as follows:

Scan the Infix string from left to right.

Initialize an empty stack.

If the scanned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty Push the character to stack.

If the scanned character is an Operand and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack). If topStack has higher precedence over the scanned character Pop the stack else Push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.

Repeat this step till all the characters are scanned.

(After all characters are scanned, we have to add any character that the stack may have to the Postfix string.) If stack is not empty add topStack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty.

Return the Postfix string.

Example:   
  
Let us see how the above algorithm will be implemented using an example.   
  
Infix String: a+b\*c-d   
  
Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/img1.gif Stack | http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/img2.gif Postfix String |

Next character scanned is 'b' which will be placed in the Postfix string. Next character is '\*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '\*', so '\*' will be pushed to the stack.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/img3.gif Stack | http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/img4.gif Postfix String |

The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '\*' which has a higher precedence than '-'. Thus '\*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the topmost element of the stack is '+' which has equal priority to '-'. So pop the '+' from the stack and add it to the Postfix string. The '-' will be pushed to the stack.

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/img5.gif Stack | http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/img6.gif Postfix String |

Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be :

|  |  |
| --- | --- |
| http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/img7.gif Stack | http://scriptasylum.com/tutorials/infix_postfix/algorithms/infix-postfix/img8.gif Postfix String |

End result:

Infix String: a+b\*c-d

Postfix String: abc\*+d-

**7. PROGRAM TO IMPLEMENT INFIX TO POSTFIX CONVERSION**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

#include<string.h>

class stack

{

public:

int top;

char \*stk;

stack(int stacksize)

{

stk=new char[stacksize];

}

char push(char x)

{

top=top+1;

stk[top]=x;

return 1;

}

char pop()

{

char x=stk[top];

top=top-1;

return x;

}

};

class exp

{

private:

char \*infix;

public:

exp();

int prec(char);

int rank(char);

char nextchar(int);

void intopost(void);

char \*append(char \*,char);

};

exp::exp()

{

cout<<"Enter any unparanthesized expression in infix notation:"<<endl;

cout<<"Delimited by #";

cin>>infix;

}

char \*exp::append(char \*postfix,char ch)

{

int len=strlen(postfix);

postfix[len++]=ch;

postfix[len++]='\0';

return postfix;

}

int exp::prec(char ch)

{

switch(ch)

{

case '+':

case '-':return 1;

case '\*':

case '/':

case '%':return 2;

case '$':return 3;

case '#':return 0;

default:return 4;

}

}

int exp::rank(char ch)

{

switch(ch)

{

case '+':

case '-':

case '\*':

case '/':

case '%':

case '$':return -1;

default:return 1;

}

}

char exp::nextchar(int count)

{

return infix[count];

}

void exp::intopost()

{

int r=0,count=0;

char temp,next;

char \*postfix=" ";

stack s(strlen(infix));

s.top=0;

s.stk[s.top]='#';

next=nextchar(count++);

while(next!='#')

{

while(prec(next)<=prec(s.stk[s.top]))

{

temp=s.pop();

postfix=append(postfix,temp);

r=r+rank(temp);

}

if(r<0)

{

cout<<"Invalid expression"<<endl;

getch();

exit(0);

}

s.push(next);

next=nextchar(count++);

}

while(s.stk[s.top]!='#')

{

temp=s.pop();

postfix=append(postfix,temp);

r=r+rank(temp);

if(r<1)

{

cout<<"Invalid expression"<<endl;

getch();

exit(0);

}

}

if(r==1)

{

cout<<"Valid expression"<<endl;

cout<<"Postfix expression is:"<<postfix<<endl;

getch();

exit(0);

}

else

{

cout<<"Invalid expression"<<endl;

getch();

exit(0);

}

}

void main()

{

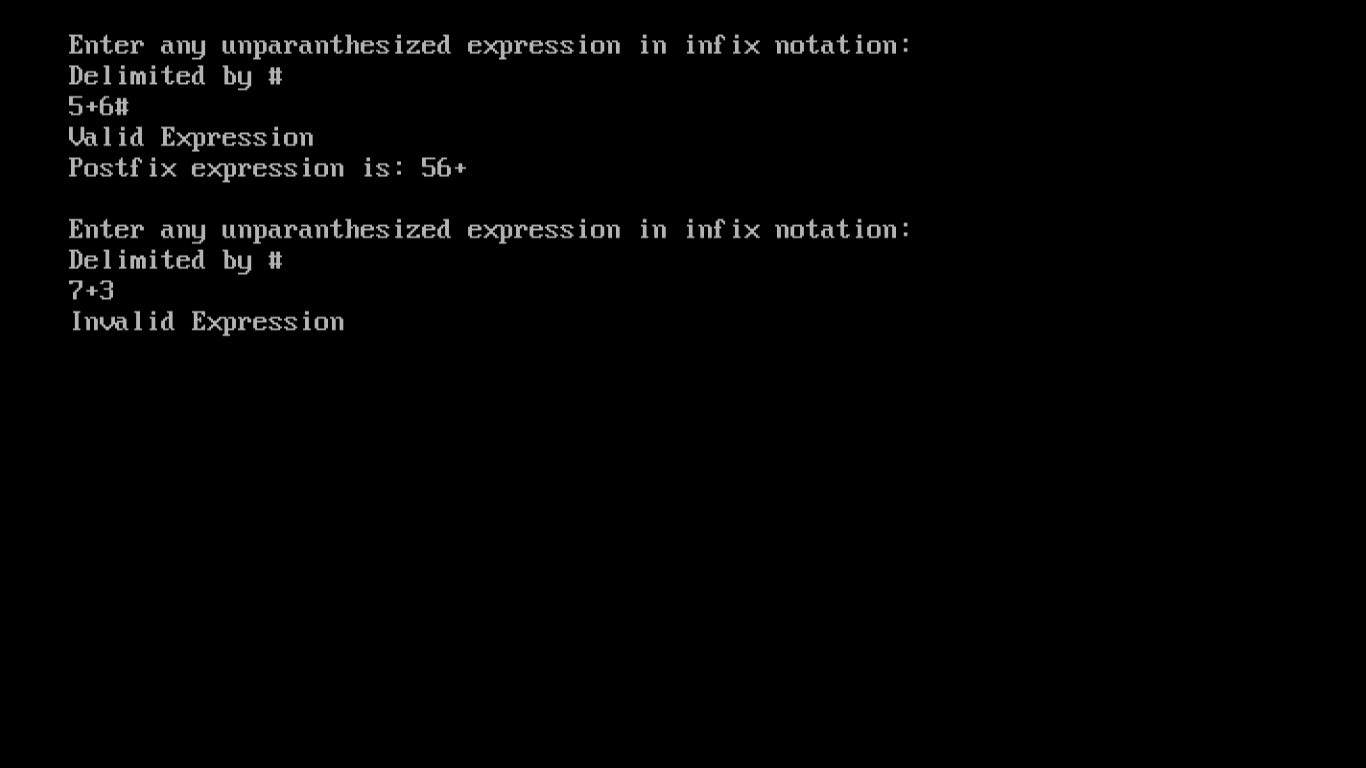
clrscr();

exp e;

e.intopost();

}

**OUTPUT**

****

**EVALUATION OF POSTFIX EXPRESSION**

If e is an expression with operators and operands , the conventional way of writing e is called infix because operators come in between the operands.(unary operators precede there operand)the postfix form of an expression calls for each operator to appear after its operands.

For example, Infix A\*B/C has postfix AB\*C/

If we study the postfix form of A\*B/C, we see that the multiplication comes immediately after its two operands A and B. Now imagine that A\*B is computed and stored in T. Then we have the division operator, /, coming immediately after its two operands T and C. Let us look at example

Infix: A/B-C+D\*E-A\*C

Postfix: AB/C-DE\*+AC-

If we read postfix expression left to right, the first operation division. The two operands that precede this are A and B. The result is stored in T6.Thus, (A/B)-C+(D\*E)-A\*C will have the same postfix form as the previous expression without parenthesis. But (A/B)-(C+D)\*(E-A)\*C will have the postfix form AB/CD+EA-\*C\*-

ALGORITHM.

void Eval(Expression e)

{//Evaluate the postfix expression e. It is assumed that the last token(a token is either an //operator, operand, or’#’)in e is ‘#’.A function Next Token is //used to get the next token from e. //The function uses the stack *stack.*

Stack<Token> *stack*; //initialize stack

for(Token x = NextToken(e); x!=’#’; x = NextToken(e))

if(x is an operand)stack.Push(x)//add to *stack*

else

{//operator

Remove the correct number of operands for operator x from stack;

Perform the operation x and store the result(if any) onto the stack;

}

}

**Uses of Postfix Evaluation:**

Postfix expressions are easy to evaluate as postfix expression does not make use of operator precedence not does it requires the use of parenthesis.

**8. PROGRAM TO IMPLEMENT EVALUATION OF POSTFIX EXPRESSION**

#include<iostream.h>

#include<math.h>

#include<string.h>

#include<stdlib.h>

#include<conio.h>

enum bool{false,true};

class stack

{

public:

int top;

char \*stk;

stack(int stacksize)

{

stk=new char[stacksize];

top=-1;

}

char \*push(char x)

{

stk[++top]=x;

return stk;

}

char pop()

{

char x=stk[top--];

return x;

}

};

bool isop(int ch)

{

switch(ch)

{

case '+':

case '-':

case '\*':

case '/':

case '$':return true;

default:return false;

}

}

float evaluate(char \*p)

{

float res,t1,t2;

stack s(strlen(p));

for(int i=0;i<strlen(p);i++)

{

if(!isop(p[i]))

{

s.push(float(p[i]-'0'));

}

else

{

t1=s.pop();

t2=s.pop();

switch(p[i])

{

case '+':res=t2+t1;

break;

case '-':res=t2-t1;

break;

case '\*':res=t2\*t1;

break;

case '/':res=t2/t1;

break;

case '$':res=pow(t2,t1);

break;

default:cout<<"invalid statement"<<endl;

exit(0);

}

s.push(res);

}

}

return((float)s.stk[s.top]);

}

void main()

{

char \*p;

clrscr();

cout<<"enter a postfix expression:"<<endl;

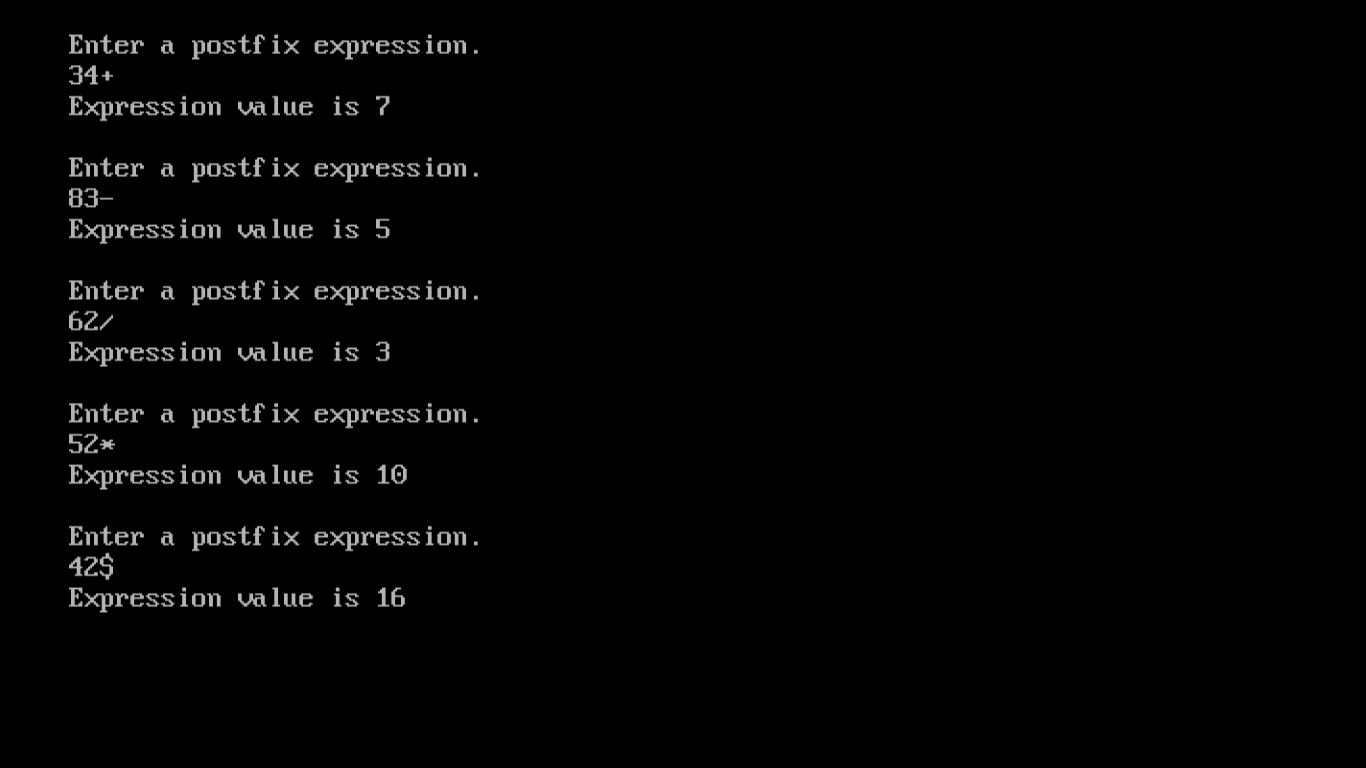
cin>>p;

cout<<"expression value is:"<<evaluate(p)<<endl;

getch();

}

**OUTPUT**

****

**POLYNOMIAL ADDITION USING LINKED LIST**

## Polynomial [Representation using Linked List](http://anubnair.posterous.com/polynomial-representation-using-linked-list):

A **Polynomial** can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term.

First of all note that in a polynomial all the terms may not be present, especially if it is going to be a very high order polynomial.

Consider 5 x12 + 2 x9 + 4x7 + 6x5 + x2 + 12 x .Now this 12th order polynomial does not have all the 13 terms (including the constant term).It would be very easy to represent the polynomial using a linked list structure, where each node can hold information pertaining to a single term of the polynomial.

Each node will need to store the variable x, the exponent and the coefficient for each term.

A Polynomial has mainly two fields. Coefficient and Exponent.

**Node of a Polynomial:**  

For example 3x2 + 5x + 7 will represent as follows.

https://lh6.googleusercontent.com/W2Jx58NLE620O9xJ27Craz4cYLG61OIQ-iyy9K9JSXGOfQhclovSKTl4txQSK_vjuQEEaEa3z8D8NkDX33gbP1tmELHOZpSt4_sNGqrQXsJT0cqwazA

In each node the exponent field will store the corresponding exponent and the coefficient field will store the corresponding coefficient. Link field points to the next item in the polynomial.

**Addition of two polynomials:**

Consider addition of the following polynomials:

5 x12 + 2 x9 + 4x7 + 6x6 + x3

7 x8 + 2 x7 + 8x6 + 6x4 + 2x2 + 3 x + 40

The resulting polynomial is going to be

5 x12 + 2 x9 + 7 x8 + 6 x7 + 14x6 + 6x4 +x3+2x2 + 3 x + 40

To add two polynomials a and b, we use the list iterators ai and bi to examine their starting at the nodes pointed to by a.first and b.first.

If the exponents of two terms are equal, then the coefficients are added. A new term is created for the result if the sum of the coefficients is not zero. If the exponent of the current term in a is less than the exponent of the current term in b, then a copy of the term in b is created and attached to the Polynomial object c. The iterator ib is advanced to the next term in b. Similar action is taken on a if **ai->exp>bi->exp.**

Each time a new node is created, its coef and exp data members are set and the node is appended to the end of c.

The basic algorithm is straightforward, using a merging process that streams along the two polynomials, either copying terms or adding them to the result.

**Analysis:** The following tasks contribute to the computing time:

1. Coefficient additions

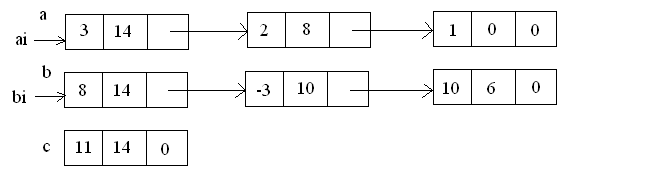
2. Exponent comparisons

3. Inserting a term at the end of the list.

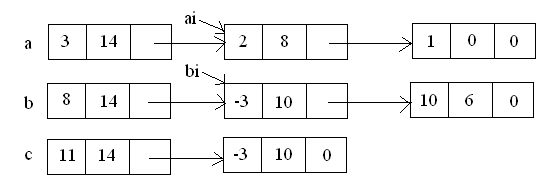
**Example:** a = 3x14+2x8+1, b= 8x14 -3x10+10x6 ,c= ?

Generating the first three terms of c= a+b

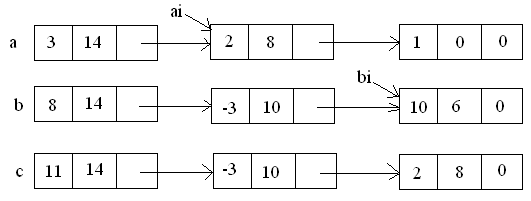
1. **ai -> exp = = bi -> exp**

****

1. **ai ->exp < bi -> exp**

****

1. **ai ->exp > bi -> exp**

****

**9. PROGRAM TO IMPLEMNT POLYNOMIAL ADDITION USING LINKED LIST**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

template<class T>

struct node

{

T coeff,expo;

node<T> \*next;

};

template<class T>

class polynomial

{

private:

node<T> \*exp1,\*exp2,\*result;

public:

polynomial();

void read();

void add();

void show();

};

template<class T>

polynomial<T>::polynomial()

{

exp1=NULL;

exp2=NULL;

result=NULL;

}

template<class T>

void polynomial<T>::read()

{

T m;

cout<<"enter the number of term in equation1:"<<endl;

cin>>m;

T c,e;

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

{

cout<<"enter coefficient and exponent:"<<endl;

cin>>c>>e;

node<T> \*n=new node<T>;

n->coeff=c;

n->next=NULL;

n->expo=e;

n->next=NULL;

if(exp1==NULL)

exp1=n;

else

{

n->next=exp1;

exp1=n;

}

}

cout<<"enter the number of term in equation2:"<<endl;

cin>>m;

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

{

cout<<"enter coefficient and exponent:"<<endl;

cin>>c>>e;

node<T> \*n=new node<T>;

n->coeff=c;

n->next=NULL;

n->expo=e;

n->next=NULL;

if(exp2==NULL)

exp2=n;

else

{

n->next=exp2;

exp2=n;

}

}

}

template<class T>

void polynomial<T>::add()

{

node<T>\*t1,\*t2;

for(t1=exp1;t1;t1=t1->next)

{

for(t2=exp2;t2;t2=t2->next)

{

if(t1->expo==t2->expo)

{

node<T> \*n=new node<T>;

n->expo=t1->expo;

n->coeff=t1->coeff+t2->coeff;

n->next=NULL;

if(result==NULL)

result=n;

else

{

n->next=result;

result=n;

}

t1->expo=0;

t2->expo=0;

}

}

}

for(t1=exp1;t1;t1=t1->next)

{

if(t1->expo!=0)

{

node<T> \*n=new node<T>;

n->expo=t1->expo;

n->coeff=t1->coeff;

n->next=result;

result=n;

}

}

for(t1=exp2;t1;t1=t1->next)

{

I f(t1->expo!=0)

{

node<T> \*n=new node<T>;

n->expo=t1->expo;

n->coeff=t1->coeff;

n->next=result;

result=n;

}

}

}

template<class T>

void polynomial<T>::show()

{

node<T> \*t;

cout<<"result of polynomial addition is:"<<endl;

for(t=result;t;t=t->next)

{

if(t->coeff>0)

{

cout<<"+"<<t->coeff<<"x"<<t->expo;

}

else

cout<<t->coeff<<"x"<<t->expo;

}

}

void main()

{

clrscr();

polynomial<int> p;

p.read();

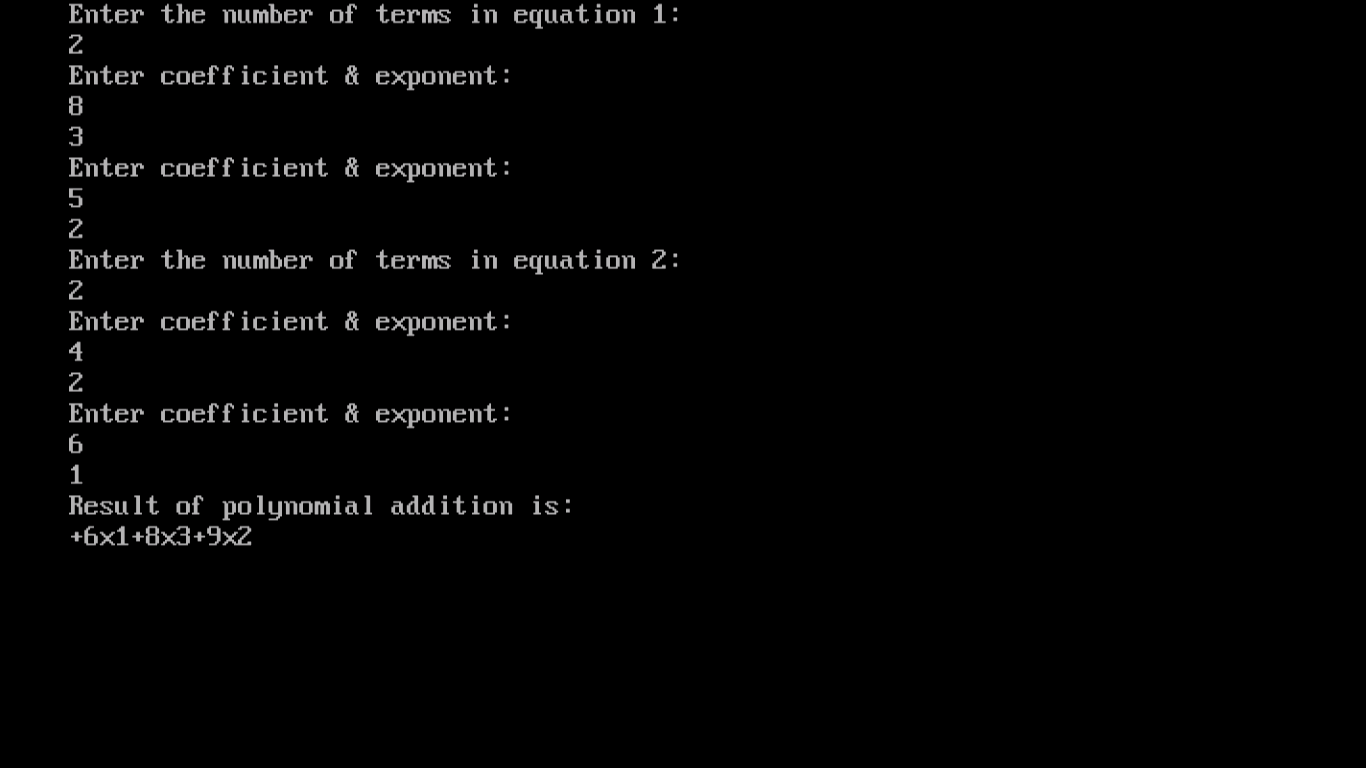
p.add();

p.show();

getch();

}

**OUTPUT**

****

**10.PROGRAM TO IMPLEMENT HASH TABLES**

#include<iostream>

#include<cstdlib>

#include<string>

#include<cstdio>

using namespace std;

const int TABLE\_SIZE = 128;

/\* HashEntry Class Declaration \*/

class HashEntry

{

public:

int key;

int value;

HashEntry(int key, int value)

{

this->key = key;

this->value = value;

}

};

/\* HashMap Class Declaration \*/

class HashMap

{

private:

HashEntry \*\*table;

public:

HashMap()

{

table = new HashEntry \* [TABLE\_SIZE];

for (int i = 0; i< TABLE\_SIZE; i++)

{

table[i] = NULL;

}

}

/\* Hash Function\*/

int HashFunc(int key)

{

return key % TABLE\_SIZE;

}

/\*Insert Element at a key \*/

void Insert(int key, int value)

{

int hash = HashFunc(key);

while (table[hash] != NULL && table[hash]->key != key)

{

hash = HashFunc(hash + 1);

}

if (table[hash] != NULL)

delete table[hash];

table[hash] = new HashEntry(key, value);

}

/\* Search Element at a key \*/

int Search(int key)

{

int hash = HashFunc(key);

while (table[hash] != NULL && table[hash]->key != key)

{

hash = HashFunc(hash + 1);

}

if (table[hash] == NULL)

return -1;

else

return table[hash]->value;

}

/\* Remove Element at a key\*/

void Remove(int key)

{

int hash = HashFunc(key);

while (table[hash] != NULL)

{

if (table[hash]->key == key)

break;

hash = HashFunc(hash + 1);

}

if (table[hash] == NULL)

{

cout<<"No Element found at key "<<key<<endl;

return;

}

else

{

delete table[hash];

}

cout<<"Element Deleted"<<endl;

}

~HashMap()

{

for (int i = 0; i < TABLE\_SIZE; i++)

{

if (table[i] != NULL)

delete table[i];

delete[] table;

}

}

};

/\*Main Contains Menu\*/

int main()

{

HashMap hash;

int key, value;

int choice;

while (1)

{

cout<<"\n----------------------"<<endl;

cout<<"Operations on Hash Table"<<endl;

cout<<"\n----------------------"<<endl;

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>>choice;

switch(choice)

{

case 1:

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

cin>>value;

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

cin>>key;

hash.Insert(key, value);

break;

case 2:

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

cin>>key;

if (hash.Search(key) == -1)

{

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

continue;

}

else

{

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

cout<<hash.Search(key)<<endl;

}

break;

case 3:

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

cin>>key;

hash.Remove(key);

break;

case 4:

exit(1);

default:

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

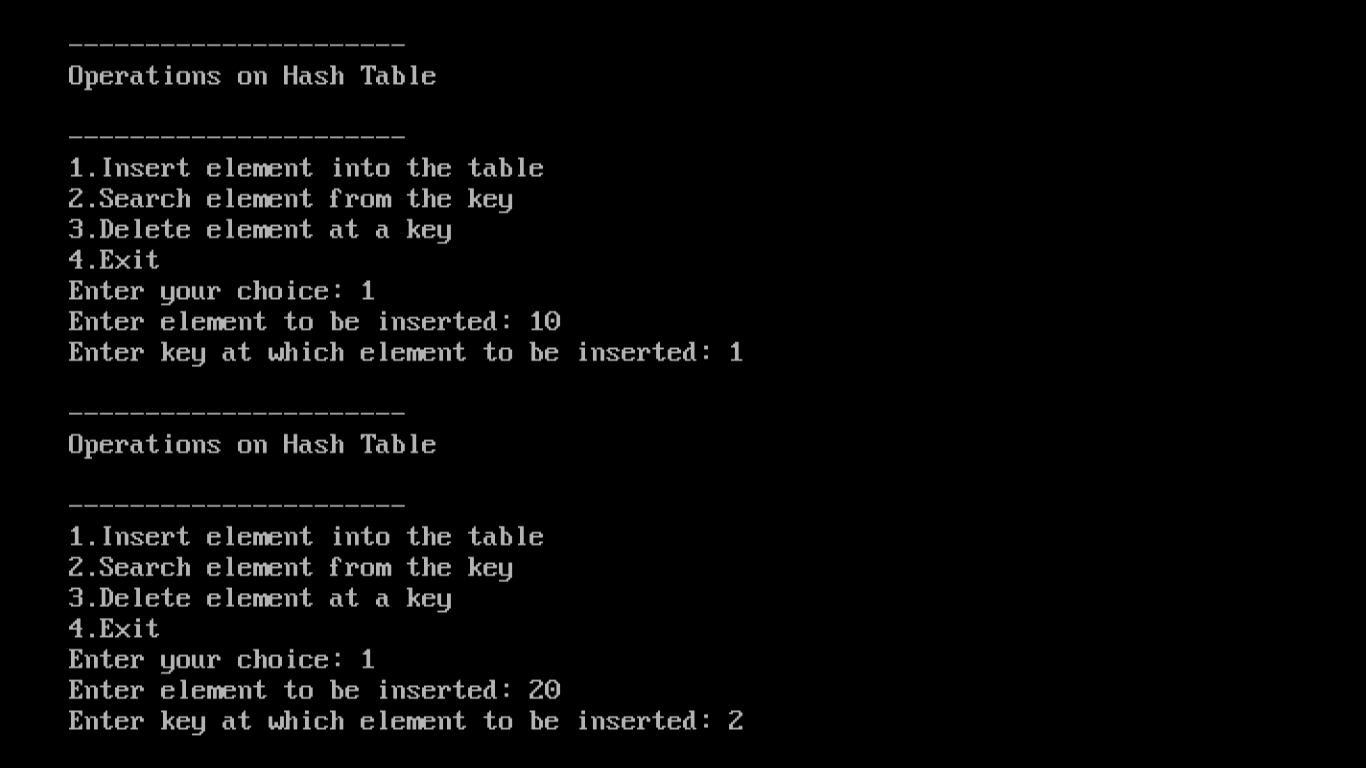
}

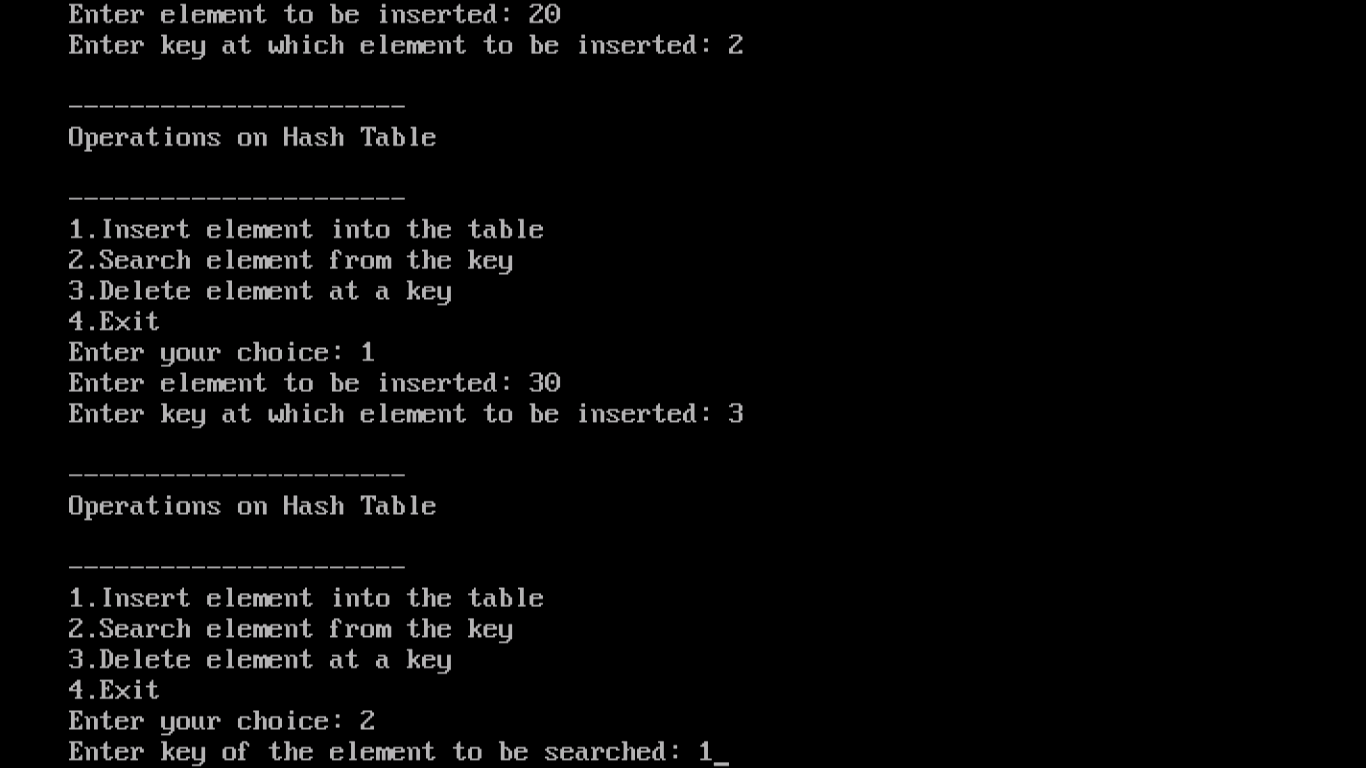
}

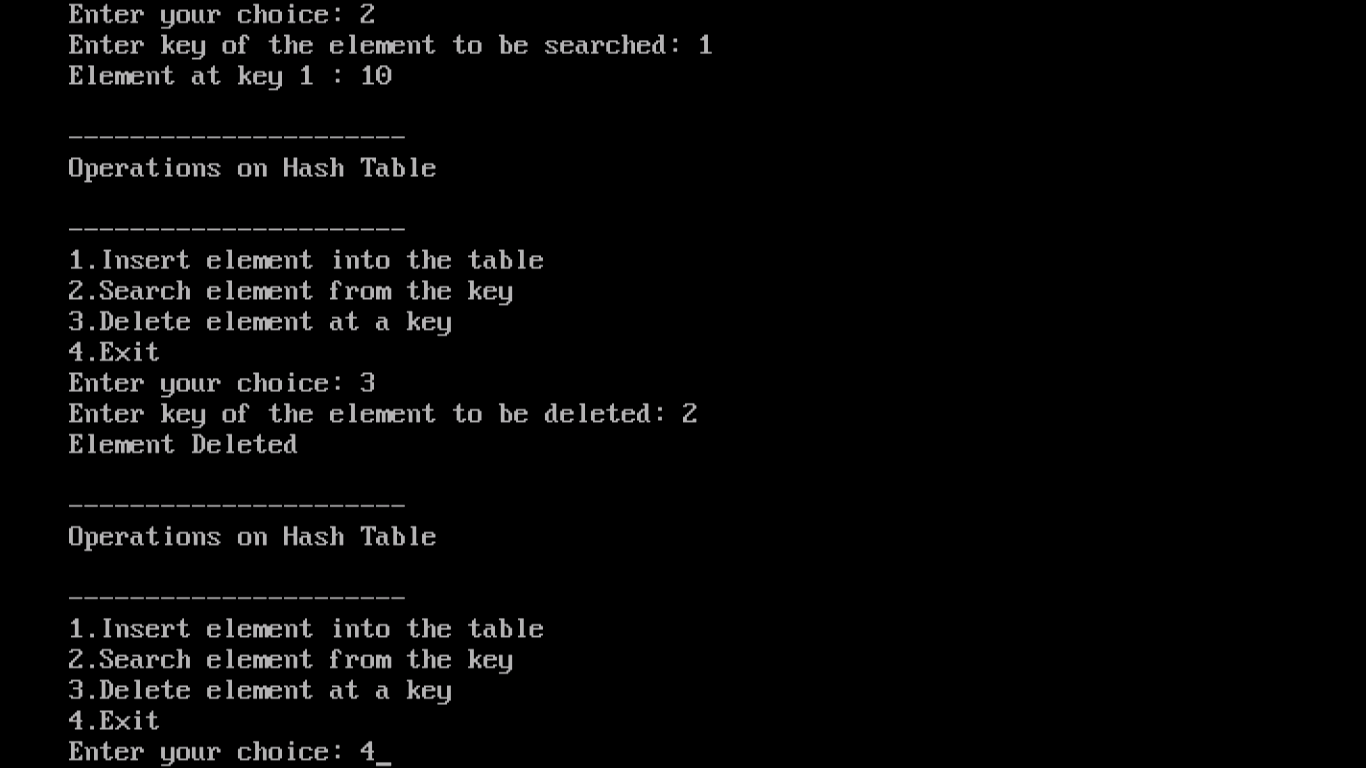
return 0;

}

**OUTPUT**

****

****

****

**SEARCHING**

Searching refers to the operation of searching the particular record from the existing information.

Searching refers to the operation of finding the location of a given item in a collection of items.

The search is said to be successful if ITEM does appear in DATA and unsuccessful otherwise.

The following are the searching algorithms:

1. Linear Search

2. Binary Search

**1. Linear Search**

Linear search or Sequential search is a method for finding a particular value in a [list](http://en.wikipedia.org/wiki/List_(computing)), that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Linear search is the simplest [search algorithm](http://en.wikipedia.org/wiki/Search_algorithm); if all list elements are equally likely to be searched for. Therefore, if the list has more than a few elements, other methods (such as [binary search](http://en.wikipedia.org/wiki/Binary_search) or [hashing](http://en.wikipedia.org/wiki/Hash_table)) will be faster, but they also impose additional requirements.

## Application: When many values have to be searched in the same list, it often pays to pre-process the latter in order to use a faster method. For example, one may [sort](http://en.wikipedia.org/wiki/Sort_(computing)) the list and use binary search, or build any efficient [search data structure](http://en.wikipedia.org/wiki/Search_data_structure) from it. Should the content of the list change frequently, repeated re-organization may be more trouble than it is worth.

## Pseudo code:

### (i).Forward iteration:

The following [pseudo code](http://en.wikipedia.org/wiki/Pseudocode) describes a typical variant of linear search, where the result of the search is supposed to be either the location of the list item where the desired value was found; or an invalid location Λ, to indicate that the desired element does not occur in the list.

For each item in the list:

if that item has the desired value,

stop the search and return the item's location.

Return Λ.

In this pseudo code, the last line is executed only after all list items have been examined with none matching.

If the list is stored as an [array data structure](http://en.wikipedia.org/wiki/Array_data_structure), the location may be the [index](http://en.wikipedia.org/wiki/Array_index) of the item found (usually between 1 and n, or 0 and n−1). In that case the invalid location Λ can be any index before the first element (such as 0 or −1, respectively) or after the last one (n+1 or n, respectively).

If the list is a [simply linked list](http://en.wikipedia.org/wiki/Linked_list), then the item's location is its [reference](http://en.wikipedia.org/wiki/Reference_(computer_science)), and Λ is usually the [null pointer](http://en.wikipedia.org/wiki/Null_pointer).

### (ii)Recursive version:

repeats its action on the sub-array to the left of the middle element or, if the input key is greater, on the sub-array to the right. If the remaining array to be searched is reduced to zero, then the key cannot be found in the array and a special "Not found" indication is returned.

Linear search can also be described as a [recursive algorithm](http://en.wikipedia.org/wiki/Recursion):

LinearSearch(value, list)

if the list is empty, return Λ;

else

if the first item of the list has the desired value, return its location;

else return LinearSearch(value, remainder of the list)

**2. Binary Search:**

Binary search or half-interval search [algorithm](http://en.wikipedia.org/wiki/Algorithm) finds the [position](http://en.wikipedia.org/wiki/Index_(information_technology)) of a specified value (the input "key") within a [sorted array](http://en.wikipedia.org/wiki/Sorted_array). At each stage, the algorithm compares the input key value with the key value of the middle element of the array. If the keys match, then a matching element has been found so its index, or position, is returned. Otherwise, if the sought key is less than the middle element's key, then the algorithm

#### 1.Iterative:

The following incorrect (see notes below) algorithm is slightly modified (to [avoid overflow](http://en.wikipedia.org/wiki/Binary_search_algorithm#Numerical_difficulties)) from [Niklaus Wirth](http://en.wikipedia.org/wiki/Niklaus_Wirth)'s in standard [Pascal](http://en.wikipedia.org/wiki/Pascal_(programming_language)):[[8]](http://en.wikipedia.org/wiki/#cite_note-7)

min := 1;

max := N; {array size: var A : array [1..N] of integer}

repeat

mid := (min+max) div 2;

if x > A[mid] then

min := mid + 1;

else

max := mid - 1;

until (A[mid] = x) or (min > max);

Note 1: This code assumes [1-based array indexing](http://en.wikipedia.org/wiki/Index_(computer_science)). For languages that use 0-based indexing (most modern languages), min and max should be initialized to 0 and N-1.

Note 2: The code above does not return a result, nor indicates whether the element was found or not.

Note 3: The code above will not work correctly for empty arrays, because it attempts to access an element before checking to see if min > max.

Note 4: After exiting the loop, the value of mid does not properly indicate whether the desired value was found in the array. One would need to perform one more comparison to determine if the value A[mid] is equal to x.

This code uses inclusive bounds and a three-way test (for early loop termination in case of equality), but with two separate comparisons per iteration. It is not the most efficient solution.

#### 2. Recursive:

A simple, straightforward implementation is [tail recursive](http://en.wikipedia.org/wiki/Tail_recursive); it recursively searches the subrange dictated by the comparison:

BinarySearch(A[0..N-1], value, low, high)

{

if (high < low)

return -1 // not found

mid = low + (high - low) / 2

if (A[mid] > value)

return BinarySearch(A, value, low, mid-1)

else if (A[mid] < value)

return BinarySearch(A, value, mid+1, high)

else

return mid // found

}

It is invoked with initial low and high values of 0 and N-1

**11. PROGRAM TO IMPLEMENT BINARY SEARCH**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class sortedlist

{

int a[10],count,i;

public:

sortedlist()

{

count=0;

i=0;

}

int insert(int);

int find(int);

void display();

};

int sortedlist::insert(int x)

{

if(count==10)

cout<<"The list is full"<<endl;

else

{

for(int i=count;i>0 && a[i-1]>x;i--)

{

a[i]=a[i-1];

}

a[i]=x;

count++;

cout<<x<<" is inserted at position:"<<i<<endl;

}

return i;

}

int sortedlist::find(int x)

{

int left=0;

int right=count;

while(left<=right)

{

int mid=(left+right)/2;

if(x>a[mid])

left=mid+1;

else if(x<a[mid])

right=mid-1;

else

{

cout<<x<<" is found at position "<<mid<<endl;

return mid;

}

}

cout<<x<<" is not found."<<endl;

return x;

}

void sortedlist::display()

{

if(count==0)

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

else

{

cout<<"The list elements are:"<<endl;

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

cout<<a[i]<<endl;

}

}

void menu()

{

cout<<endl;

cout<<"1.Insert 2.Find 3.Display 4.Exit"<<endl;

cout<<"Enter your choice: ";

}

void main()

{

sortedlist s;

int ch,el,e;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:cout<<"Enter the element to be inserted"<<endl;

cin>>el;

s.insert(el);

break;

case 2:cout<<"Enter the element to be searched"<<endl;

cin>>el;

s.find(el);

break;

case 3:s.display();

break;

case 4:exit(0);

}

menu();

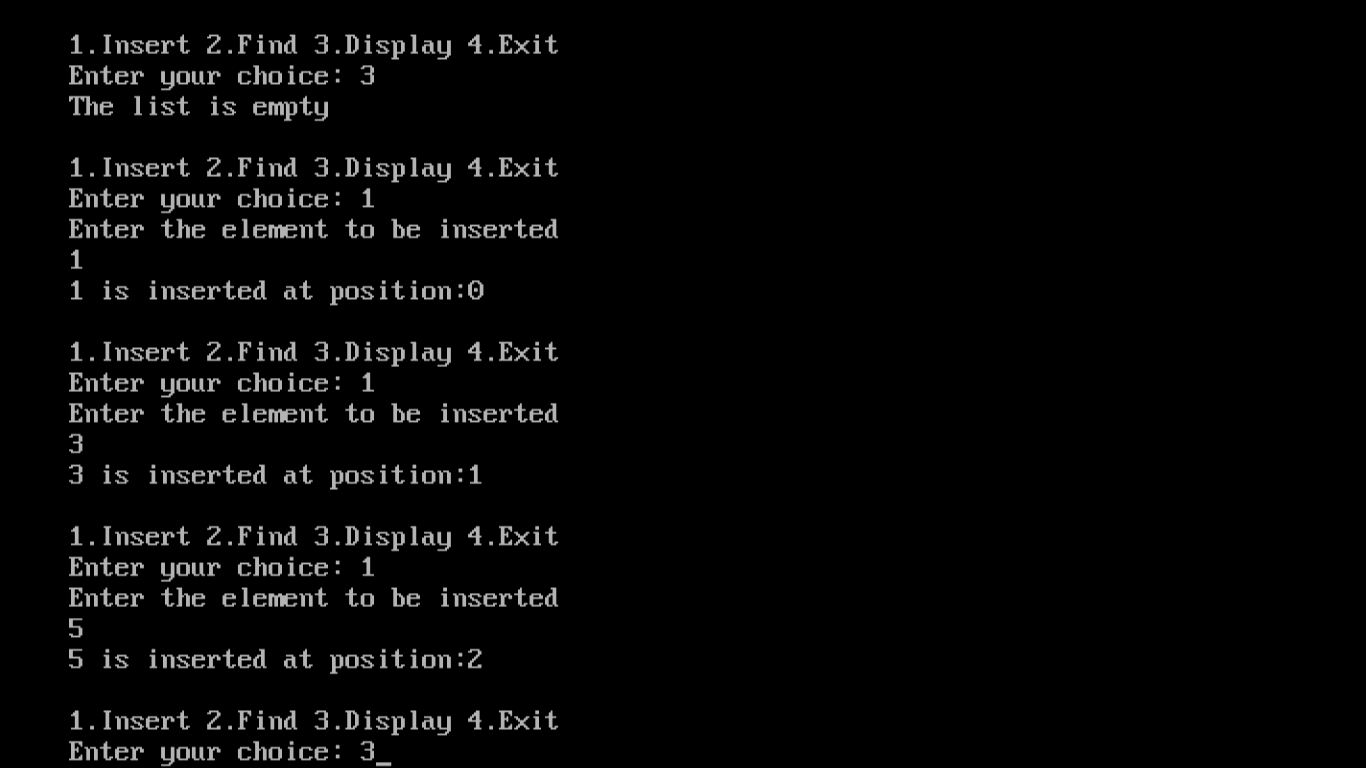
cin>>ch;

}

getch();

}

**OUTPUT**

****

****

**Binary Search TreeS**

A binary search tree T is a finite set of keys. Either the set is empty T=NULL, or the set consist of a root “r” and exactly two binary search trees TL and TR,T={r,TL,TR} with the following properties.

i) All the keys contained in the left sub tree are less than the root key.

ii) All the keys contained in the right sub tree are greater than the root key.

Suppose T is a binary tree. Then T is called a binary search tree if each node N of T has the following property: “The value at N is greater than every value in the left subtree of N and is less than every value in the right sub tree of N.

**Searching and Inserting in Binary Search Trees:**

Suppose an ITEM of information is given. The following algorithm finds the location of

ITEM in the binary search tree T, or inserts ITEM as a new node in its appropriate place in the tree.

(a) Compare ITEM with the root node N of the tree.

(i) If ITEM < N, proceed to the left child of N.

(ii) If ITEM > N, proceed to the right child of N.

(b) Repeat Step (a) until one of the following occurs:

(i) We meet a node N such that ITEM = N. In this case the search is successful.

(ii) We meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree.

In other words, proceed from the root R down through the tree T until finding ITEM in T or inserting ITEM as a terminal node in T.

**Deleting in a Binary Search Tree:**

Suppose T is a binary search tree, and suppose an ITEM of information is given. This section gives an algorithm which deletes ITEM from the tree T.

**Case 1.** N has no children. Then N is deleted from T by simply replacing the location of N in

the parent node P(N) by the null pointer.

**Case 2**. N has exactly one child. Then N is deleted from T by simply replacing the location of

N in P(N) by the location of the only child of N.

**Case 3**. N has two children. Let S(N) denote the inorder successor of N. ( The reader can verify that S(N) does not have a left child.) Then N is deleted from T by first deleting

S(N) from T (by using Case 1 or Case 2) and then replacing node N in T by the node S(N).

Observe that the third case is much more complicated than the first two cases. In all three cases, the memory space of the deleted node N is returned to the AVAIL list.

**Binary SEARCH tree traversals**

Definition: The process of walking through the tree visiting each and every node in the tree is known as **Tree Traversal (or)** Visiting all nodes of a tree is called **Tree Traversal.**

Different types of tree traversals are

1.Depth first traversal

2.Breadth first traversal or Level Order Traversal

**1.Depth First Traversal:** It has three different tree traversal techniques.

**(i)Preorder Traversal (N L R):** In Preorder Traversal,

* Visit the root first
* Traverse the left subtree in Preorder then
* Traverse the right subtree in Preorder

Preorder Traversal can be easily described recursively as follows, for a subtree rooted at n:

void preorder(node \* n)

{

visit(n);

preorder(leftChild(n));

preorder(rightChild(n));

}

**(ii) Inorder Traversal (L N R):** In Inorder Traversal,

* Traverse the left subtree in Inorder
* Visit the root the
* Traverse the right subtree in inorder

Inorder Traversal can be easily described recursively as follows, for a subtree rooted at n:

void inorder(node \* n)

{

inorder(leftChild(n);

visit(n);

inorder(rightChild(n);

}

**(iii)Postorder Traversal (L R N):** In Postorder traversal,

* Traverse the left subtree in Postorder
* Traverse the right subtree in Postorder
* Visit the root

Postorder Traversal can be easily described recursively as follows, for a subtree rooted at n:

void postorder(node \* n)

{

postorder(leftChild(n));

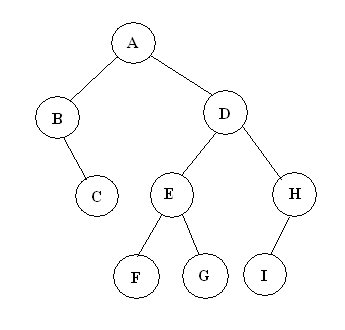
postorder(rightChild(n));

visit(n);

}

**2. Breadth First Traversal OR Level Order Traversal**: It always start with root node. Visit the root node of level 0 then level 1 from left to right until all the levels are completed. (Visiting level by level from left to right).

**Example:**



Preorder Traversal : A,B,C,D,E,F,G,H,I

Inorder Traversal : B,C,A,F,E,G,D,I,H

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

Breadth First Traversal : A,B,D,C,E,H,F,G,I

**12. PROGRAM TO IMPLEMENT BINARY SEARCH TREE – PREORDER**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class btnode

{

int data;

btnode \*lchild;

btnode \*rchild;

public:

btnode(int item)

{

data=item;

lchild=0;

rchild=0;

}

friend btree;

};

class btree

{

public:

btnode \*root;

btree()

{

root=NULL;

}

void create(int);

void insert(int);

void preorder(btnode \*);

};

void btree::create(int n)

{

int item;

if(n<=0)

return;

cout<<"Enter the root node:";

cin>>item;

root=new btnode(item);

int i=2;

cout<<"Input the remaining child nodes:"<<endl;

while(i<=n)

{

cin>>item;

insert(item);

i++;

}

}

void btree::insert(int item)

{

btnode \*p=root;

btnode \*q=0;

while(p)

{

q=p;

if(item==p->data)

{

cout<<"Duplicates are not allowed"<<endl;

cout<<"Insert another value"<<endl;

cin>>item;

insert(item);

return;

}

if(item<p->data)

p=p->lchild;

else

p=p->rchild;

}

p=new btnode(item);

if(root==NULL)

root=p;

else

{

if(item<q->data)

q->lchild=p;

else

q->rchild=p;

}

}

void btree::preorder(btnode \*current)

{

if(root==NULL)

cout<<"Tree is empty."<<endl;

else

{

if(current!=NULL)

{

preorder(current->lchild);

preorder(current->rchild);

cout<<current->data<<endl;

}

}

}

void menu()

{

cout<<endl;

cout<<"1.Create Tree 2.Insert 3.Display in PREORDER 4.Exit"<<endl;

cout<<"Enter your choice."<<endl;

}

void main()

{

btree b;

int ch,n;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:cout<<"Enter the nodes to create a tree:";

cin>>n;

b.create(n);

break;

case 2:if(b.root==NULL)

cout<<"Create the tree first."<<endl;

else

{

cout<<"Enter the node value to be inserted:";

cin>>n;

b.insert(n);

}

break;

case 3:cout<<"Tree nodes after traversing in PREORDER are:"<<endl;

b.preorder(b.root);

break;

case 4:exit(0);

}

menu();

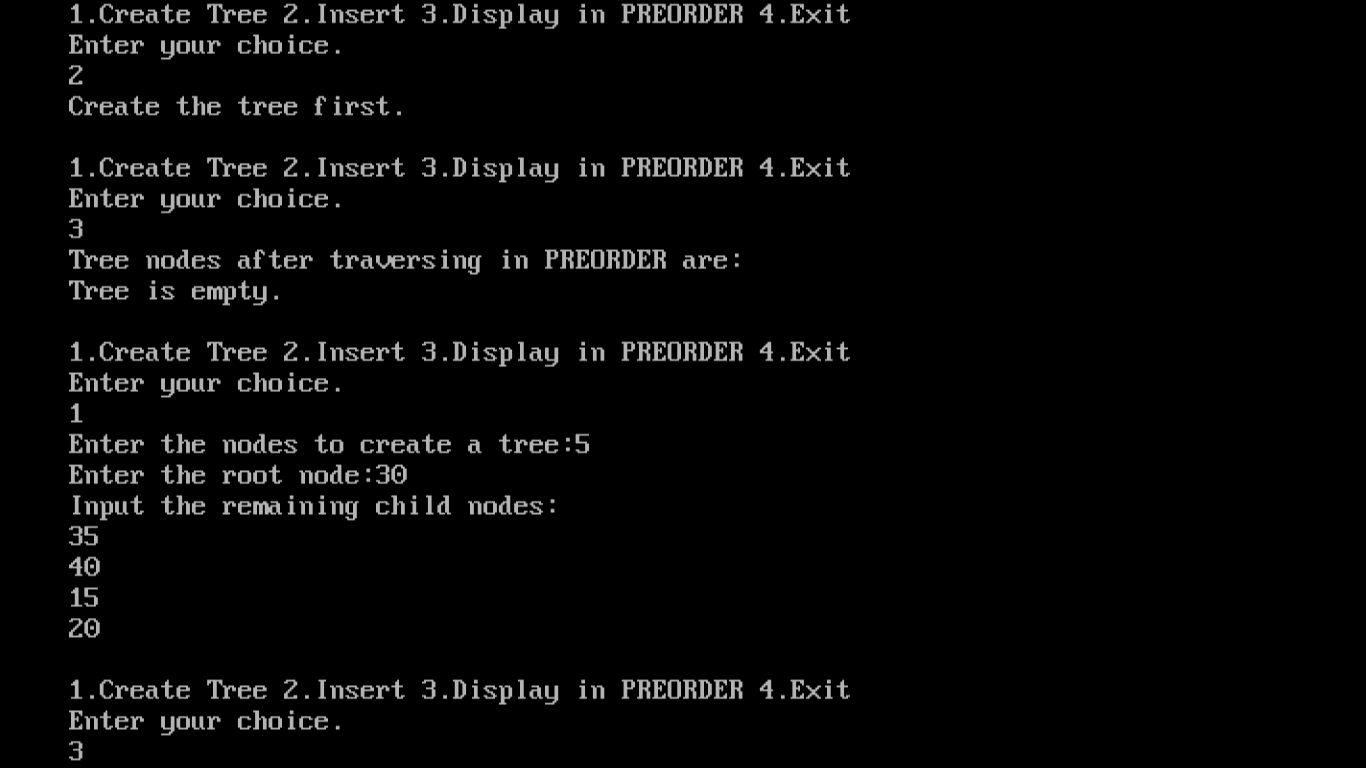
cin>>ch;

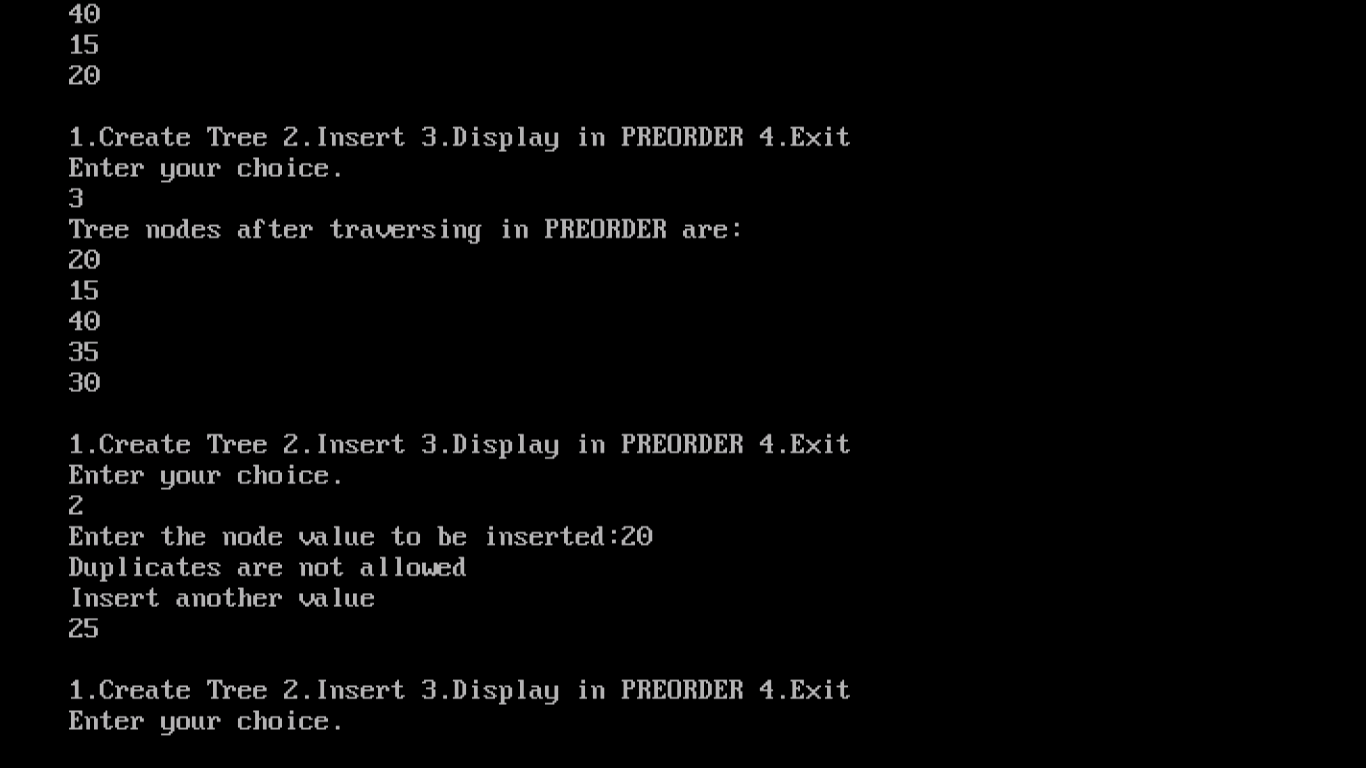
}

getch();

}

**OUTPUT**

****

****

**

**13. PROGRAM TO IMPLEMENT BINARY SEARCH TREE – INORDER**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class btnode

{

int data;

btnode \*lchild;

btnode \*rchild;

public:

btnode(int item)

{

data=item;

lchild=0;

rchild=0;

}

friend btree;

};

class btree

{

public:

btnode \*root;

btree()

{

root=NULL;

}

void create(int);

void insert(int);

void inorder(btnode \*);

};

void btree::create(int n)

{

int item;

if(n<=0)

return;

cout<<"Enter the root node:";

cin>>item;

root=new btnode(item);

int i=2;

cout<<"Input the remaining child nodes:"<<endl;

while(i<=n)

{

cin>>item;

insert(item);

i++;

}

}

void btree::insert(int item)

{

btnode \*p=root;

btnode \*q=0;

while(p)

{

q=p;

if(item==p->data)

{

cout<<"Duplicates are not allowed."<<endl;

cout<<"Insert another value."<<endl;

cin>>item;

insert(item);

return;

}

if(item<p->data)

p=p->lchild;

else

p=p->rchild;

}

p=new btnode(item);

if(root==NULL)

root=p;

else

{

if(item<q->data)

q->lchild=p;

else

q->rchild=p;

}

}

void btree::inorder(btnode \*current)

{

if(root==NULL)

cout<<"Tree is empty"<<endl;

else

{

if(current!=NULL)

{

inorder(current->lchild);

cout<<current->data<<endl;

inorder(current->rchild);

}

}

}

void menu()

{

cout<<endl;

cout<<"1.Create tree 2.Insert 3.Display in INORDER 4.Exit"<<endl;

cout<<"Enter your choice."<<endl;

}

void main()

{

btree b;

int ch,n;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:cout<<"Enter the nodes to create a tree:";

cin>>n;

b.create(n);

break;

case 2:if(b.root==NULL)

cout<<"Create the tree first"<<endl;

else

{

cout<<"Enter the node value to be inserted:";

cin>>n;

b.insert(n);

}

break;

case 3:cout<<"Tree nodes after traversing in INORDER are:"<<endl;

b.inorder(b.root);

break;

case 4:exit(0);

}

menu();

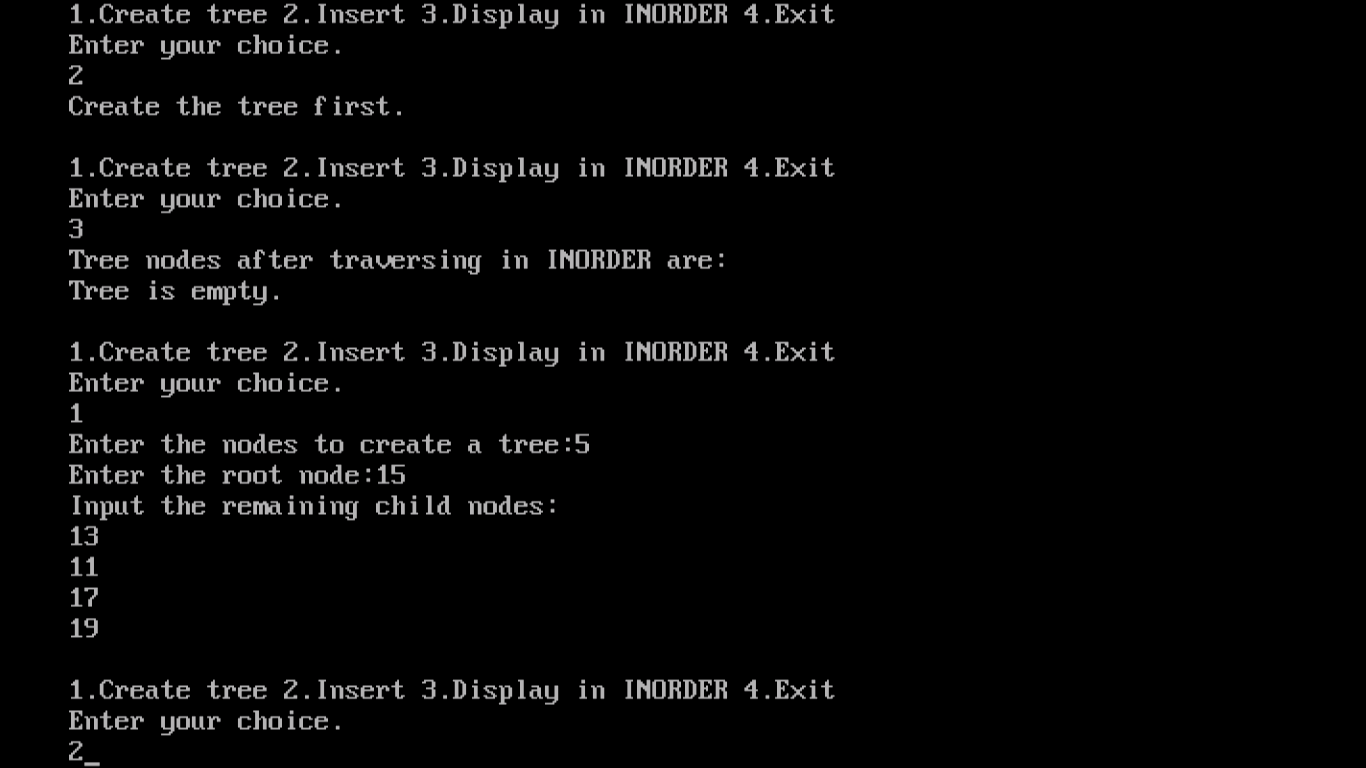
cin>>ch;

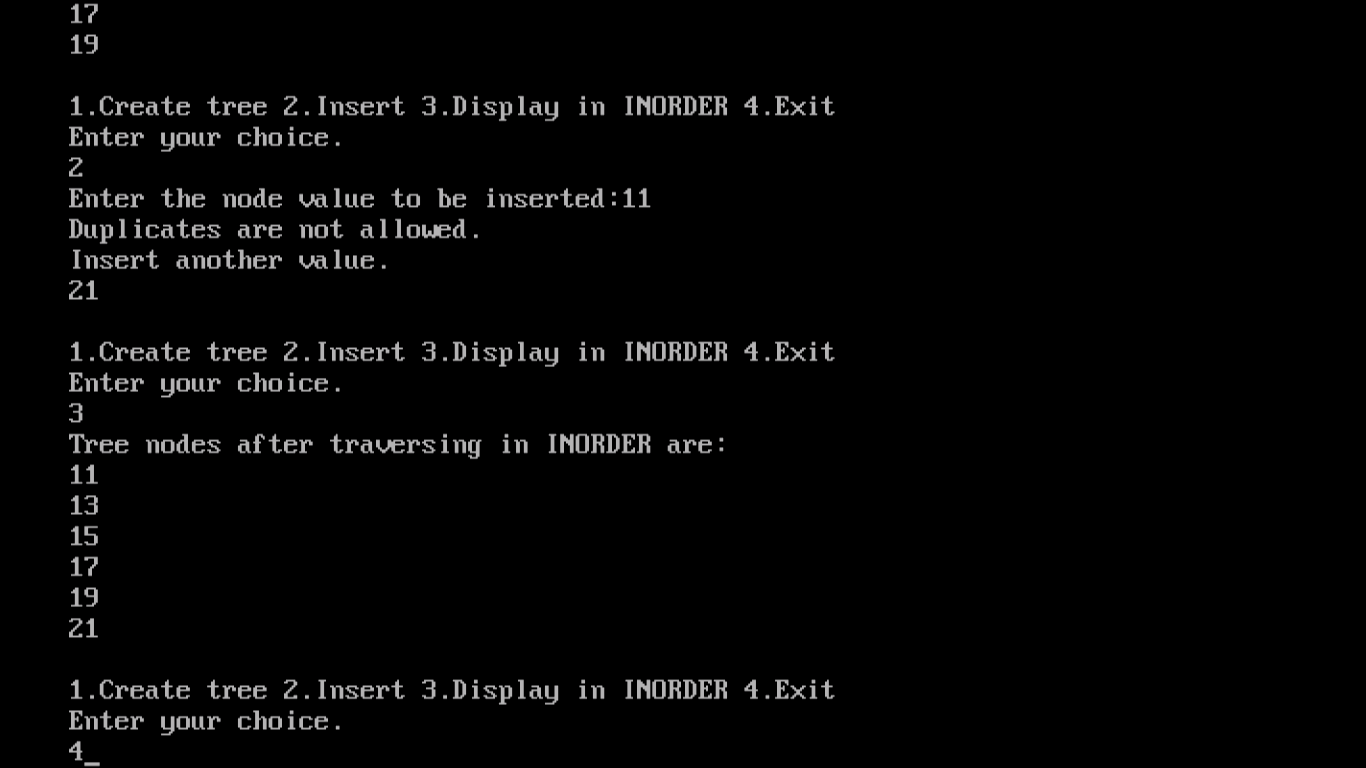
}

getch();

}

**OUTPUT**

**

******

**14. PROGRAM TO IMPLEMENT BINARY SEARCH TREE –POSTORDER**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class btnode

{

int data;

btnode \*lchild;

btnode \*rchild;

public:

btnode(int item)

{

data=item;

lchild=0;

rchild=0;

}

friend btree;

};

class btree

{

public:

btnode \*root;

btree()

{

root=NULL;

}

void create(int);

void insert(int);

void postorder(btnode \*);

};

void btree::create(int n)

{

int item;

if(n<=0)

return;

cout<<"Enter the root node:";

cin>>item;

root=new btnode(item);

int i=2;

cout<<"Input the remaining child nodes:"<<endl;

while(i<=n)

{

cin>>item;

insert(item);

i++;

}

}

void btree::insert(int item)

{

btnode \*p=root;

btnode \*q=0;

while(p)

{

q=p;

if(item==p->data)

{

cout<<"Duplicates are not allowed."<<endl;

cout<<"Insert another value."<<endl;

cin>>item;

insert(item);

return;

}

if(item<p->data)

p=p->lchild;

else

p=p->rchild;

}

p=new btnode(item);

if(root==NULL)

root=p;

else

{

if(item<q->data)

q->lchild=p;

else

q->rchild=p;

}

}

void btree::postorder(btnode \*current)

{

if(root==NULL)

cout<<"Tree is empty."<<endl;

else

{

if(current!=NULL)

{

postorder(current->lchild);

postorder(current->rchild);

cout<<current->data<<endl;

}

}

}

void menu()

{

cout<<endl;

cout<<"1.Create tree 2.Insert 3.Display in POSTORDER 4.Exit"<<endl;

cout<<"Enter your choice."<<endl;

}

void main()

{

btree b;

int ch,n;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:cout<<"Enter the nodes to create a tree:";

cin>>n;

b.create(n);

break;

case 2:if(b.root==NULL)

cout<<"Create the tree first."<<endl;

else

{

cout<<"Enter the node value to be inserted:";

cin>>n;

b.insert(n);

}

break;

case 3:cout<<"Tree nodes after traversing in POSTORDER are:"<<endl;

b.postorder(b.root);

break;

case 4:exit(0);

}

menu();

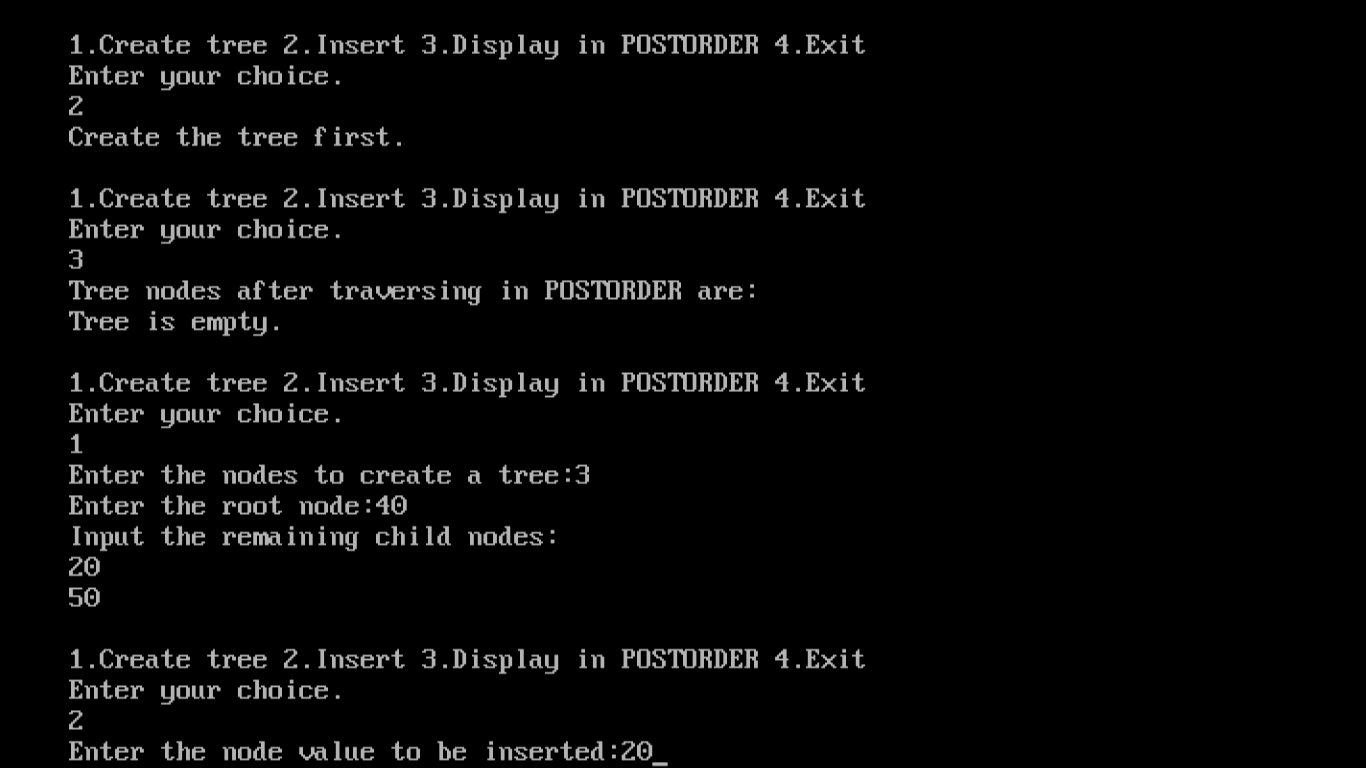
cin>>ch;

}

getch();

}

**OUTPUT**

****

****

**B - Tree**

B-Tree is a self-balancing search tree. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations require O(h) disk accesses where h is height of the tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, and Red Black Tree.

**Properties of B-Tree**

**1)** All leaves are at same level.

**2)** A B-Tree is defined by the term minimum degree ‘t’. The value of t depends upon disk block size.

**3)** Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.

**4)** All nodes (including root) may contain at most 2t – 1 keys.

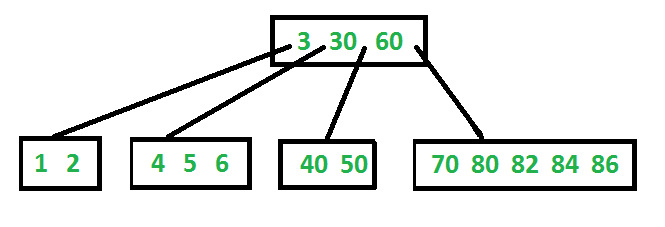
**5)** Number of children of a node is equal to the number of keys in it plus 1.

**6)** All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in range from k1 and k2.

**7)** B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

**8)** Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.

[](http://www.geeksforgeeks.org/wp-content/uploads/BTreeIntro1.png)

**15. PROGRAM TO IMPLEMENT B-TREE**

#include<stdio.h>

#include<conio.h>

#include<iostream>

using namespace std;

struct BTreeNode

{

int \*data;

BTreeNode \*\*child\_ptr;

bool leaf;

int n;

}\*root = NULL, \*np = NULL, \*x = NULL;

BTreeNode \* init()

{

int i;

np = new BTreeNode;

np->data = new int[5];

np->child\_ptr = new BTreeNode \*[6];

np->leaf = true;

np->n = 0;

for (i = 0; i < 6; i++)

{

np->child\_ptr[i] = NULL;

}

return np;

}

void traverse(BTreeNode \*p)

{

cout<<endl;

int i;

for (i = 0; i < p->n; i++)

{

if (p->leaf == false)

{

traverse(p->child\_ptr[i]);

}

cout<< " " << p->data[i];

}

if (p->leaf == false)

{

traverse(p->child\_ptr[i]);

}

cout<<endl;

}

void sort(int \*p, int n)

{

int i, j, temp;

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

{

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

{

if (p[i] > p[j])

{

temp = p[i];

p[i] = p[j];

p[j] = temp;

}

}

}

}

int split\_child(BTreeNode \*x, int i)

{

int j, mid;

BTreeNode \*np1, \*np3, \*y;

np3 = init();

np3->leaf = true;

if (i == -1)

{

mid = x->data[2];

x->data[2] = 0;

x->n--;

np1 = init();

np1->leaf = false;

x->leaf = true;

for (j = 3; j < 5; j++)

{

np3->data[j - 3] = x->data[j];

np3->child\_ptr[j - 3] = x->child\_ptr[j];

np3->n++;

x->data[j] = 0;

x->n--;

}

for (j = 0; j < 6; j++)

{

x->child\_ptr[j] = NULL;

}

np1->data[0] = mid;

np1->child\_ptr[np1->n] = x;

np1->child\_ptr[np1->n + 1] = np3;

np1->n++;

root = np1;

}

else

{

y = x->child\_ptr[i];

mid = y->data[2];

y->data[2] = 0;

y->n--;

for (j = 3; j < 5; j++)

{

np3->data[j - 3] = y->data[j];

np3->n++;

y->data[j] = 0;

y->n--;

}

x->child\_ptr[i + 1] = y;

x->child\_ptr[i + 1] = np3;

}

return mid;

}

void insert(int a)

{

int i, temp;

x = root;

if (x == NULL)

{

root = init();

x = root;

}

else

{

if (x->leaf == true && x->n == 5)

{

temp = split\_child(x, -1);

x = root;

for (i = 0; i < (x->n); i++)

{

if ((a > x->data[i]) && (a < x->data[i + 1]))

{

i++;

break;

}

else if (a < x->data[0])

{

break;

}

else

{

continue;

}

}

x = x->child\_ptr[i];

}

else

{

while (x->leaf == false)

{

for (i = 0; i < (x->n); i++)

{

if ((a > x->data[i]) && (a < x->data[i + 1]))

{

i++;

break;

}

else if (a < x->data[0])

{

break;

}

else

{

continue;

}

}

if ((x->child\_ptr[i])->n == 5)

{

temp = split\_child(x, i);

x->data[x->n] = temp;

x->n++;

continue;

}

else

{

x = x->child\_ptr[i];

}

}

}

}

x->data[x->n] = a;

sort(x->data, x->n);

x->n++;

}

int main()

{

int i, n, t;

cout<<"enter the no of elements to be inserted\n";

cin>>n;

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

{

cout<<"enter the element\n";

cin>>t;

insert(t);

}

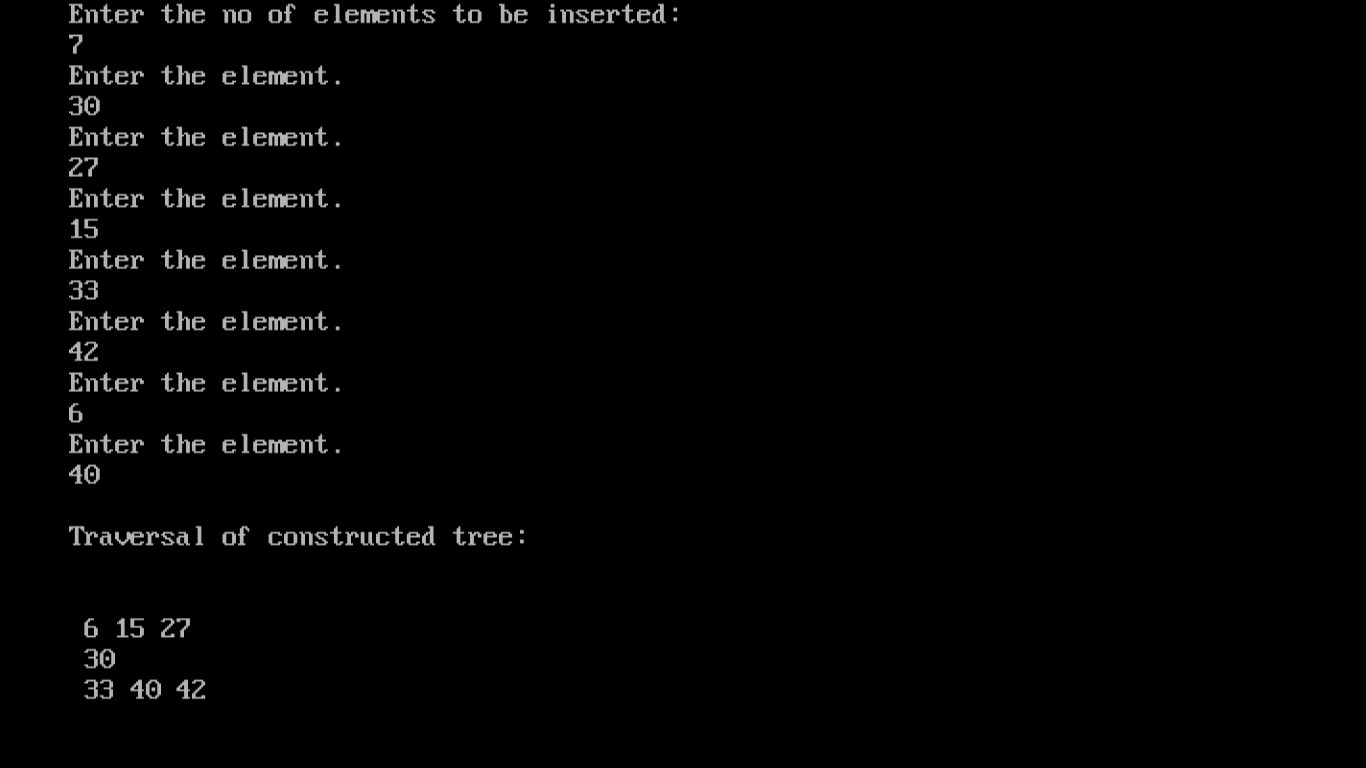
cout<<"traversal of constructed tree\n";

traverse(root);

getch();

}

**OUTPUT**

**

**16. PROGRAM TO IMPLEMENT AVL TREE**

#include<iostream>

#include<cstdio>

#include<sstream>

#include<algorithm>

#define pow2(n) (1 << (n))

using namespace std;

/\* Node Declaration\*/

struct avl\_node

{

int data;

struct avl\_node \*left;

struct avl\_node \*right;

}\*root;

/\*Class Declaration\*/

class avlTree

{

public:

int height(avl\_node \*);

int diff(avl\_node \*);

avl\_node \*rr\_rotation(avl\_node \*);

avl\_node \*ll\_rotation(avl\_node \*);

avl\_node \*lr\_rotation(avl\_node \*);

avl\_node \*rl\_rotation(avl\_node \*);

avl\_node\* balance(avl\_node \*);

avl\_node\* insert(avl\_node \*, int );

void display(avl\_node \*, int);

void inorder(avl\_node \*);

void preorder(avl\_node \*);

void postorder(avl\_node \*);

avlTree()

{

root = NULL;

}

};

/\* Main Contains Menu\*/

int main()

{

int choice, item;

avlTree avl;

while (1)

{

cout<<"\n---------------------"<<endl;

cout<<"AVL Tree Implementation"<<endl;

cout<<"\n---------------------"<<endl;

cout<<"1.Insert Element into the tree"<<endl;

cout<<"2.Display Balanced AVL Tree"<<endl;

cout<<"3.InOrder traversal"<<endl;

cout<<"4.PreOrder traversal"<<endl;

cout<<"5.PostOrder traversal"<<endl;

cout<<"6.Exit"<<endl;

cout<<"Enter your Choice: ";

cin>>choice;

switch(choice)

{

case 1:

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

cin>>item;

root = avl.insert(root, item);

break;

case 2:

if (root == NULL)

{

cout<<"Tree is Empty"<<endl;

continue;

}

cout<<"Balanced AVL Tree:"<<endl;

avl.display(root, 1);

break;

case 3:

cout<<"Inorder Traversal:"<<endl;

avl.inorder(root);

cout<<endl;

break;

case 4:

cout<<"Preorder Traversal:"<<endl;

avl.preorder(root);

cout<<endl;

break;

case 5:

cout<<"Postorder Traversal:"<<endl;

avl.postorder(root);

cout<<endl;

break;

case 6:

exit(1);

break;

default:

cout<<"Wrong Choice"<<endl;

}

}

return 0;

}

/\*Height of AVL Tree\*/

int avlTree::height(avl\_node \*temp)

{

int h = 0;

if (temp != NULL)

{

int l\_height = height (temp->left);

int r\_height = height (temp->right);

int max\_height = max (l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

/\*Height Difference \*/

int avlTree::diff(avl\_node \*temp)

{

int l\_height = height (temp->left);

int r\_height = height (temp->right);

int b\_factor= l\_height - r\_height;

return b\_factor;

}

/\*Right- Right Rotation\*/

avl\_node \*avlTree::rr\_rotation(avl\_node \*parent)

{

avl\_node \*temp;

temp = parent->right;

parent->right = temp->left;

temp->left = parent;

return temp;

}

/\*Left- Left Rotation \*/

avl\_node \*avlTree::ll\_rotation(avl\_node \*parent)

{

avl\_node \*temp;

temp = parent->left;

parent->left = temp->right;

temp->right = parent;

return temp;

}

/\* Left - Right Rotation\*/

avl\_node \*avlTree::lr\_rotation(avl\_node \*parent)

{

avl\_node \*temp;

temp = parent->left;

parent->left = rr\_rotation (temp);

return ll\_rotation (parent);

}

/\*Right- Left Rotation \*/

avl\_node \*avlTree::rl\_rotation(avl\_node \*parent)

{

avl\_node \*temp;

temp = parent->right;

parent->right = ll\_rotation (temp);

return rr\_rotation (parent);

}

/\*Balancing AVL Tree\*/

avl\_node \*avlTree::balance(avl\_node \*temp)

{

int bal\_factor = diff (temp);

if (bal\_factor> 1)

{

if (diff (temp->left) > 0)

temp = ll\_rotation (temp);

else

temp = lr\_rotation (temp);

}

else if (bal\_factor< -1)

{

if (diff (temp->right) > 0)

temp = rl\_rotation (temp);

else

temp = rr\_rotation (temp);

}

return temp;

}

/\*Insert Element into the tree\*/

avl\_node \*avlTree::insert(avl\_node \*root, int value)

{

if (root == NULL)

{

root = new avl\_node;

root->data = value;

root->left = NULL;

root->right = NULL;

return root;

}

else if (value < root->data)

{

root->left = insert(root->left, value);

root = balance (root);

}

else if (value >= root->data)

{

root->right = insert(root->right, value);

root = balance (root);

}

return root;

}

/\*Display AVL Tree \*/

void avlTree::display(avl\_node \*ptr, int level)

{

int i;

if (ptr!=NULL)

{

display(ptr->right, level + 1);

printf("\n");

if (ptr == root)

cout<<"Root -> ";

for (i = 0; i < level &&ptr != root; i++)

cout<<" ";

cout<<ptr->data;

display(ptr->left, level + 1);

}

}

/\*Inorder Traversal of AVL Tree\*/

void avlTree::inorder(avl\_node \*tree)

{

if (tree == NULL)

return;

inorder (tree->left);

cout<<tree->data<<" ";

inorder (tree->right);

}

/\* Preorder Traversal of AVL Tree\*/

void avlTree::preorder(avl\_node \*tree)

{

if (tree == NULL)

return;

cout<<tree->data<<" ";

preorder (tree->left);

preorder (tree->right);

}

/\* Postorder Traversal of AVL Tree\*/

void avlTree::postorder(avl\_node \*tree)

{

if (tree == NULL)

return;

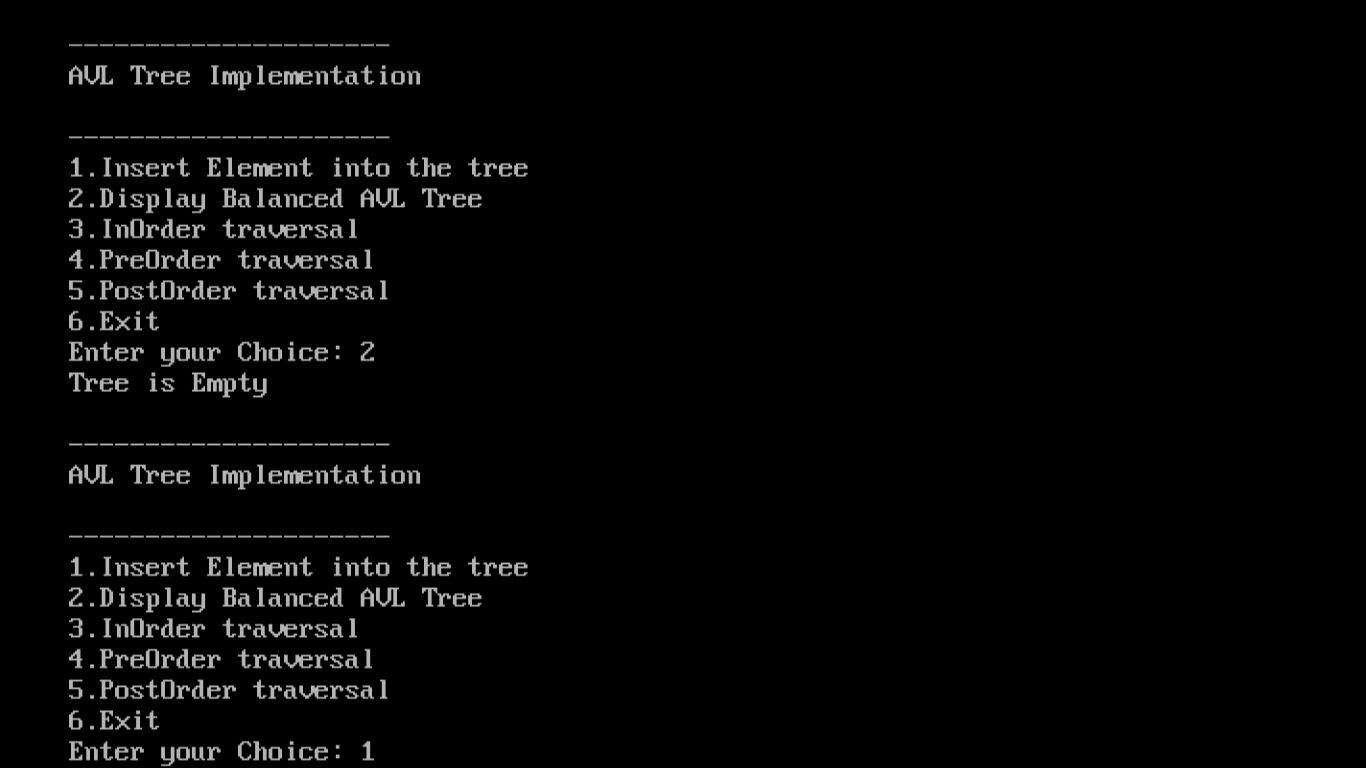
postorder ( tree ->left );

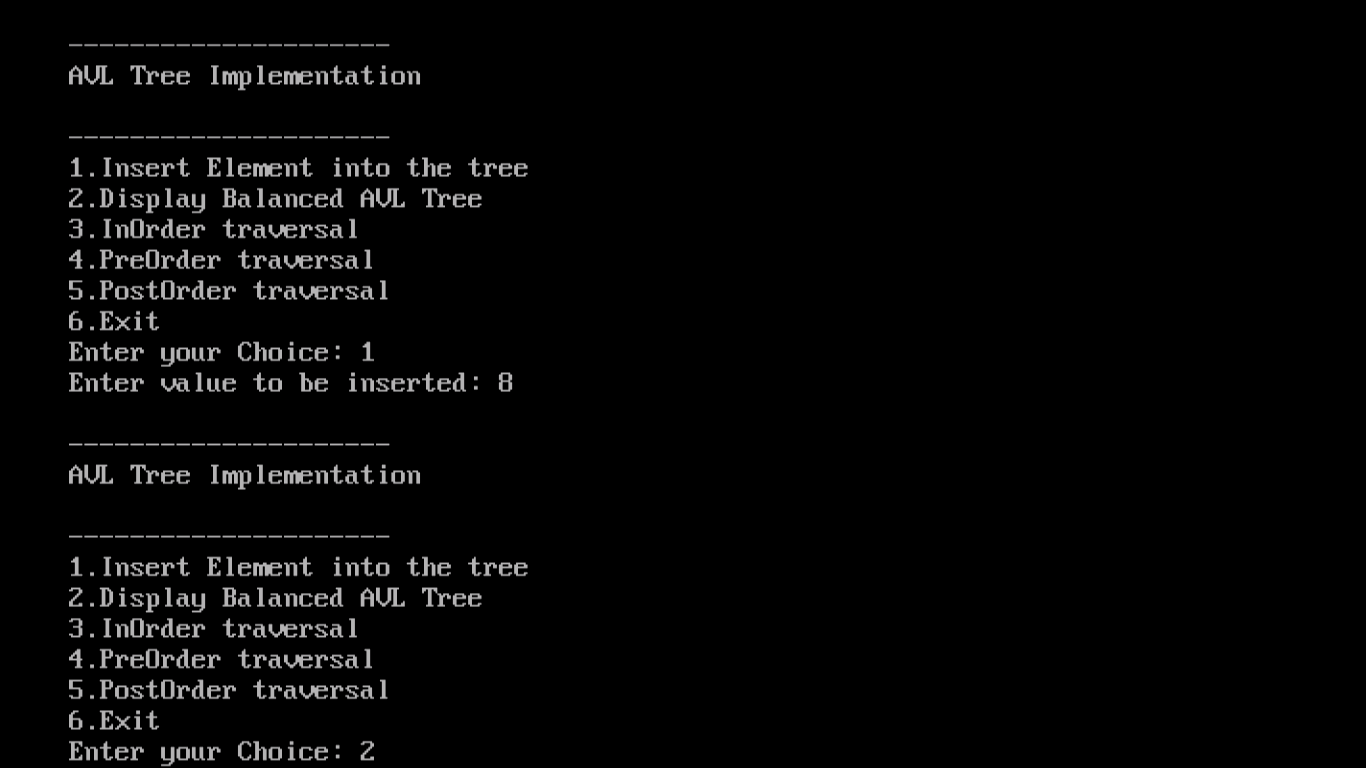
postorder ( tree ->right );

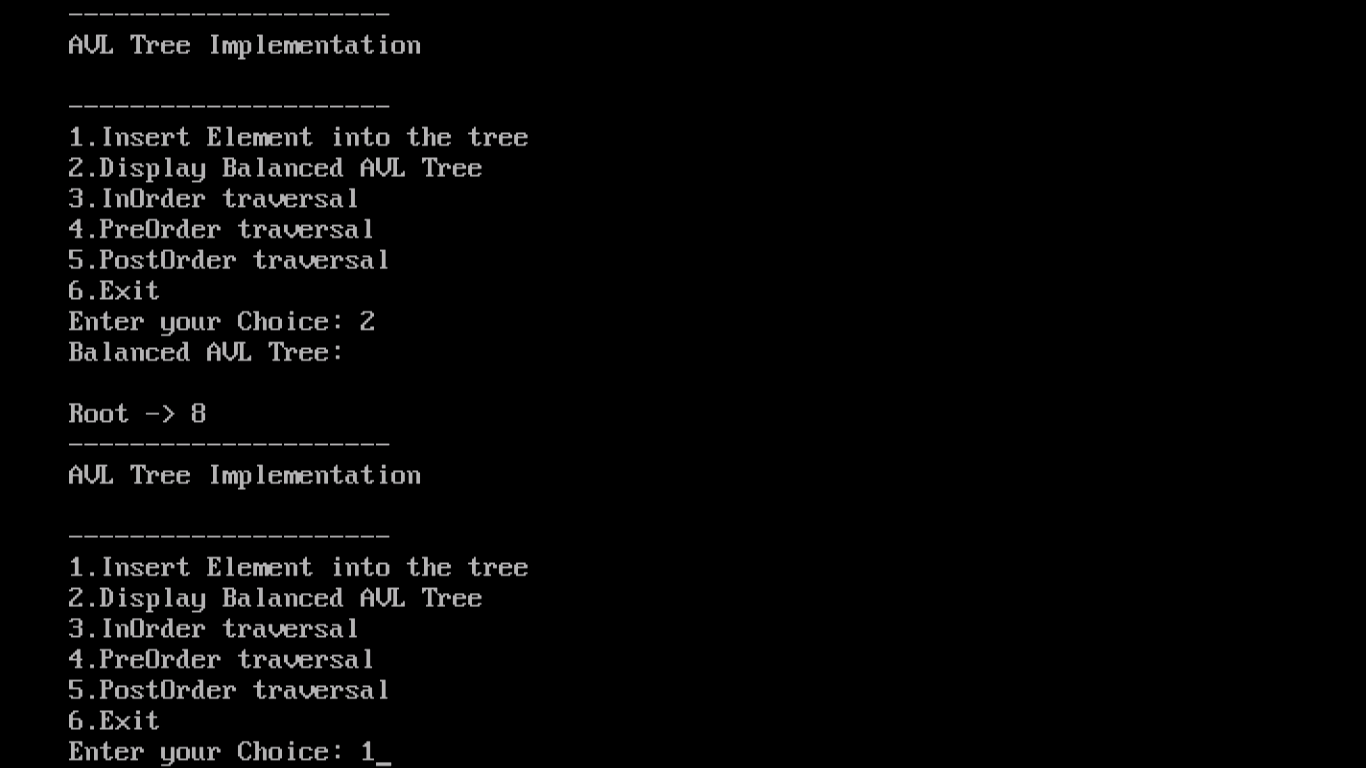
cout<<tree->data<<" ";

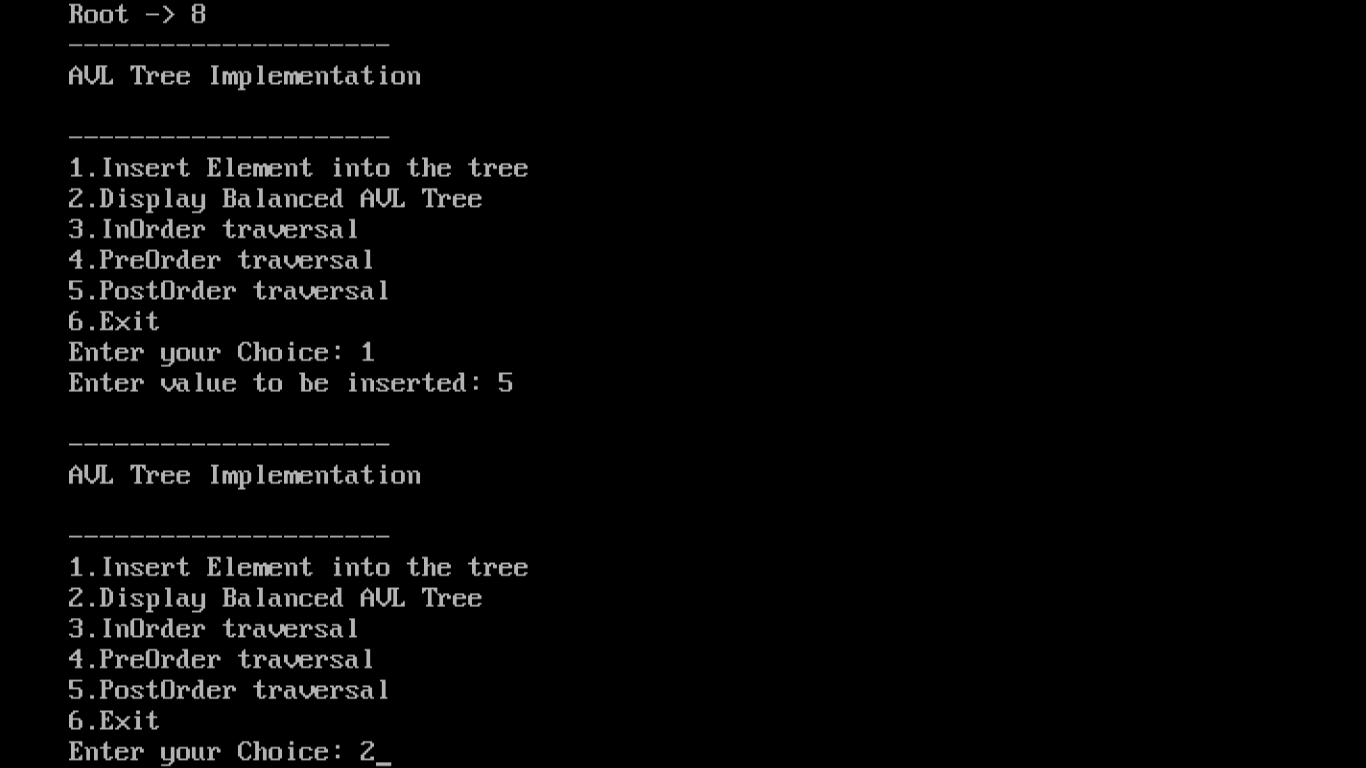
}

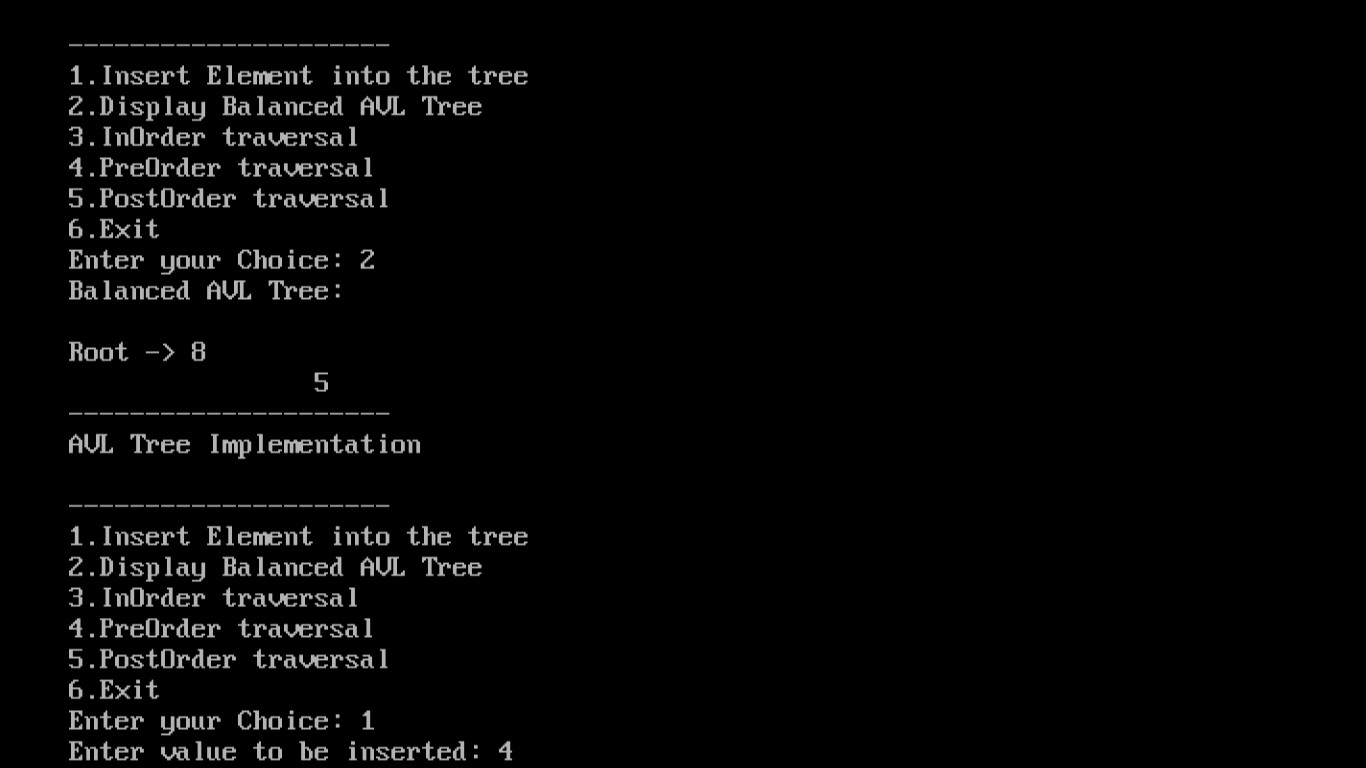
**OUTPUT**

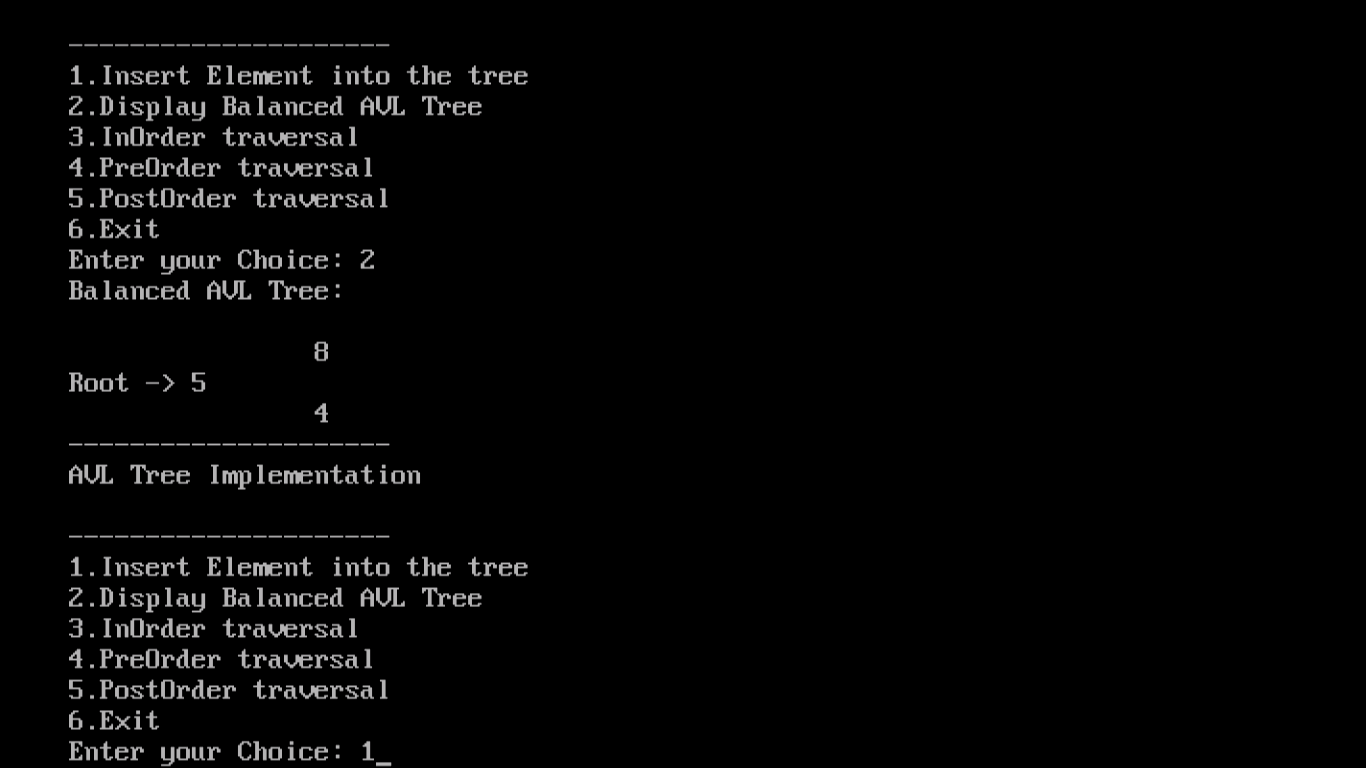
****

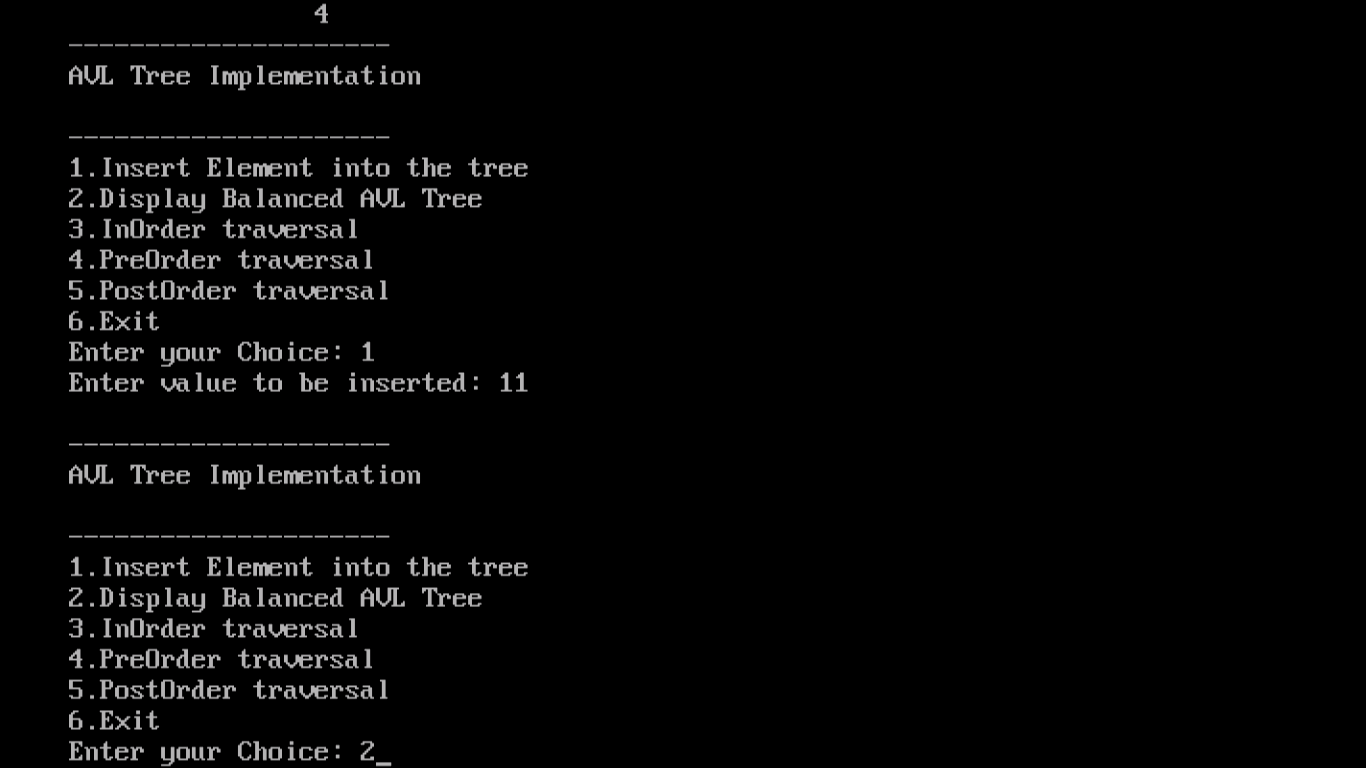
****

****

****

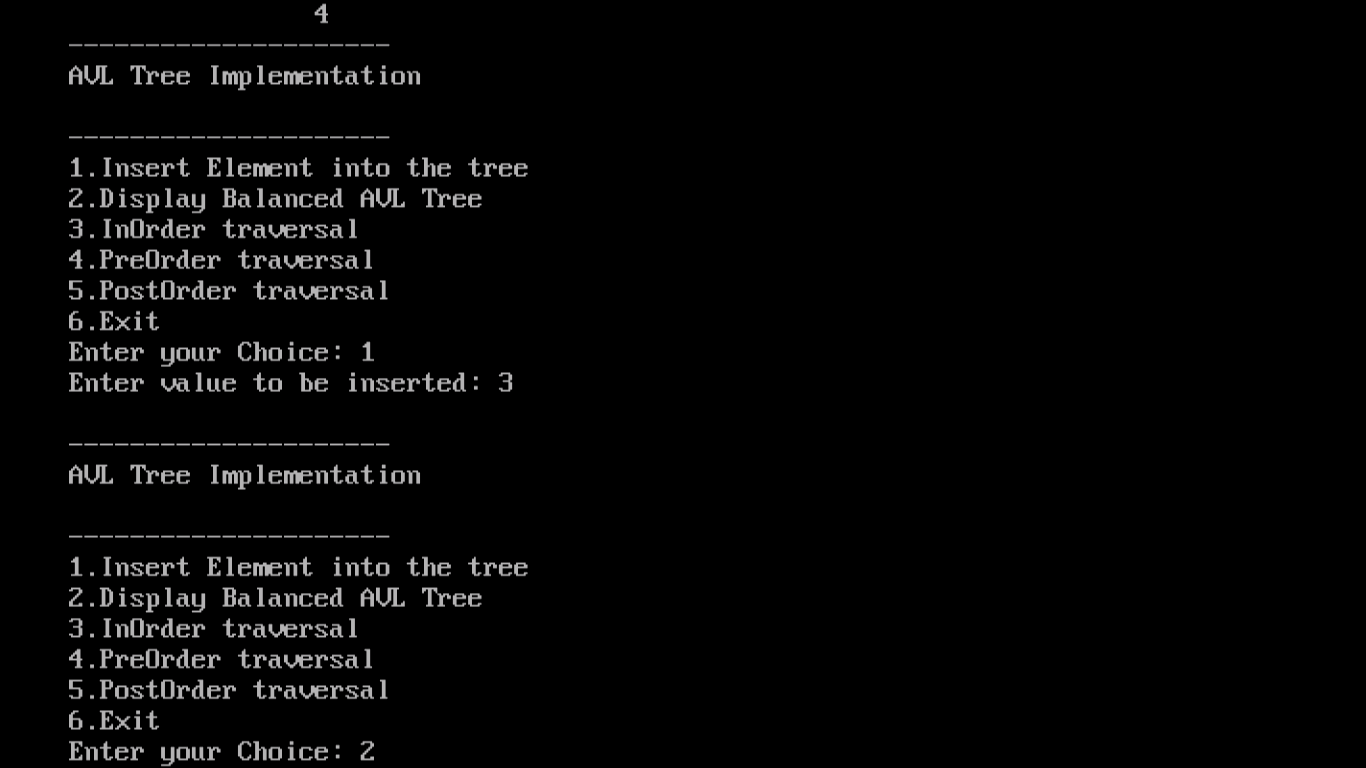
****

****

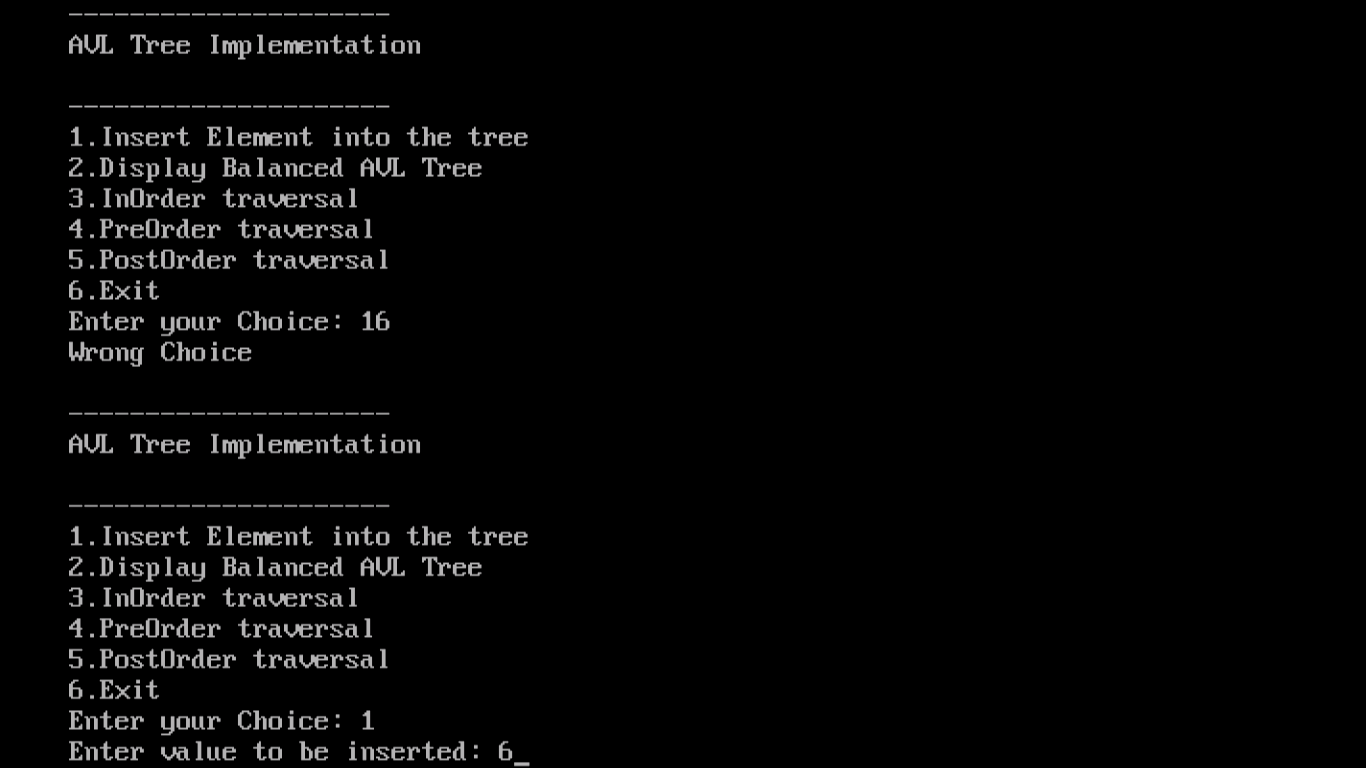
****

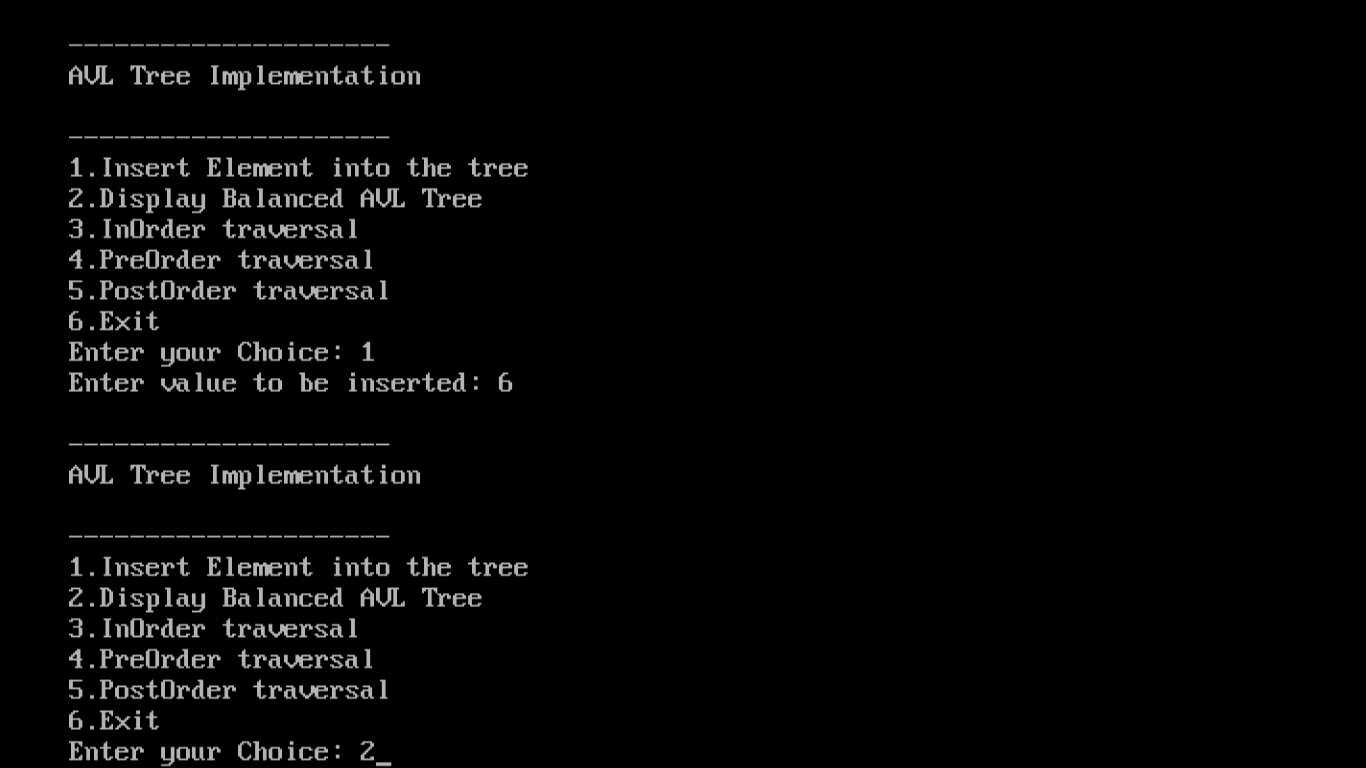
****

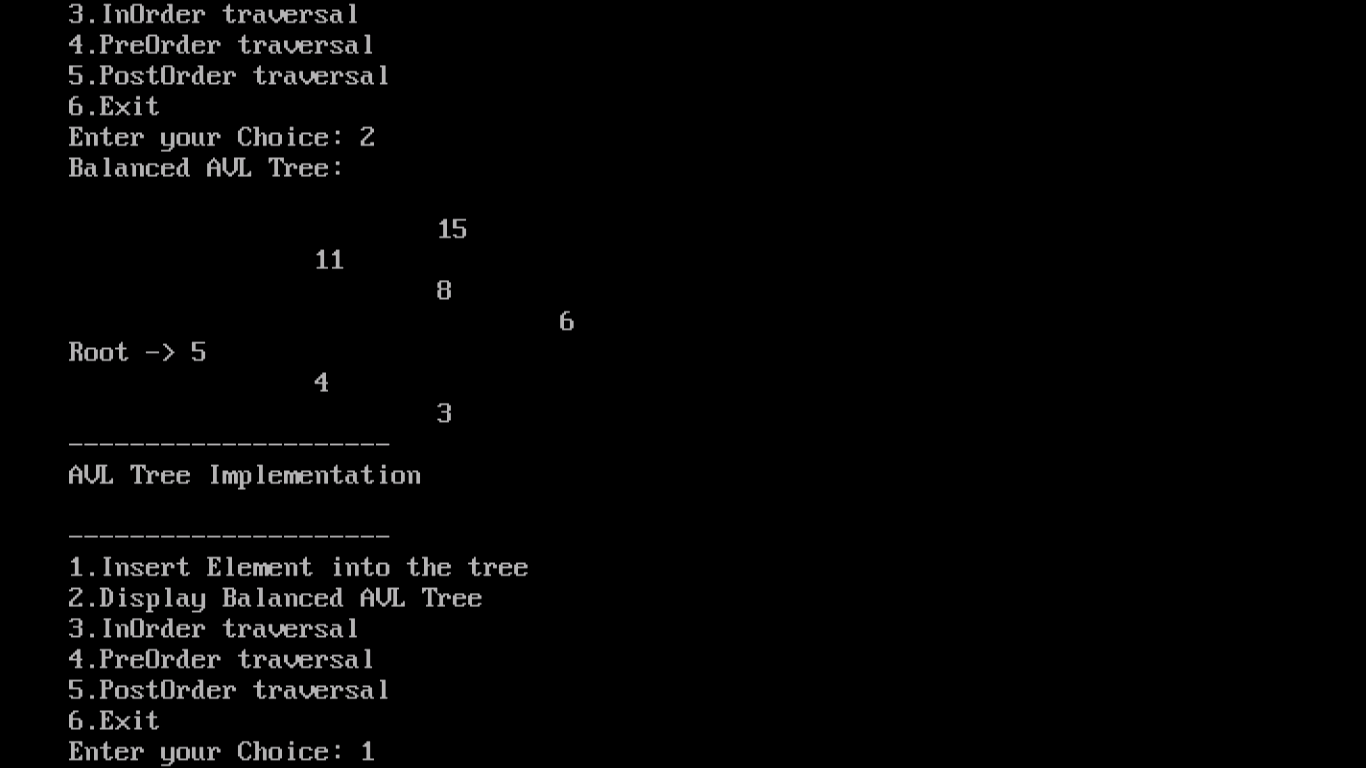
****

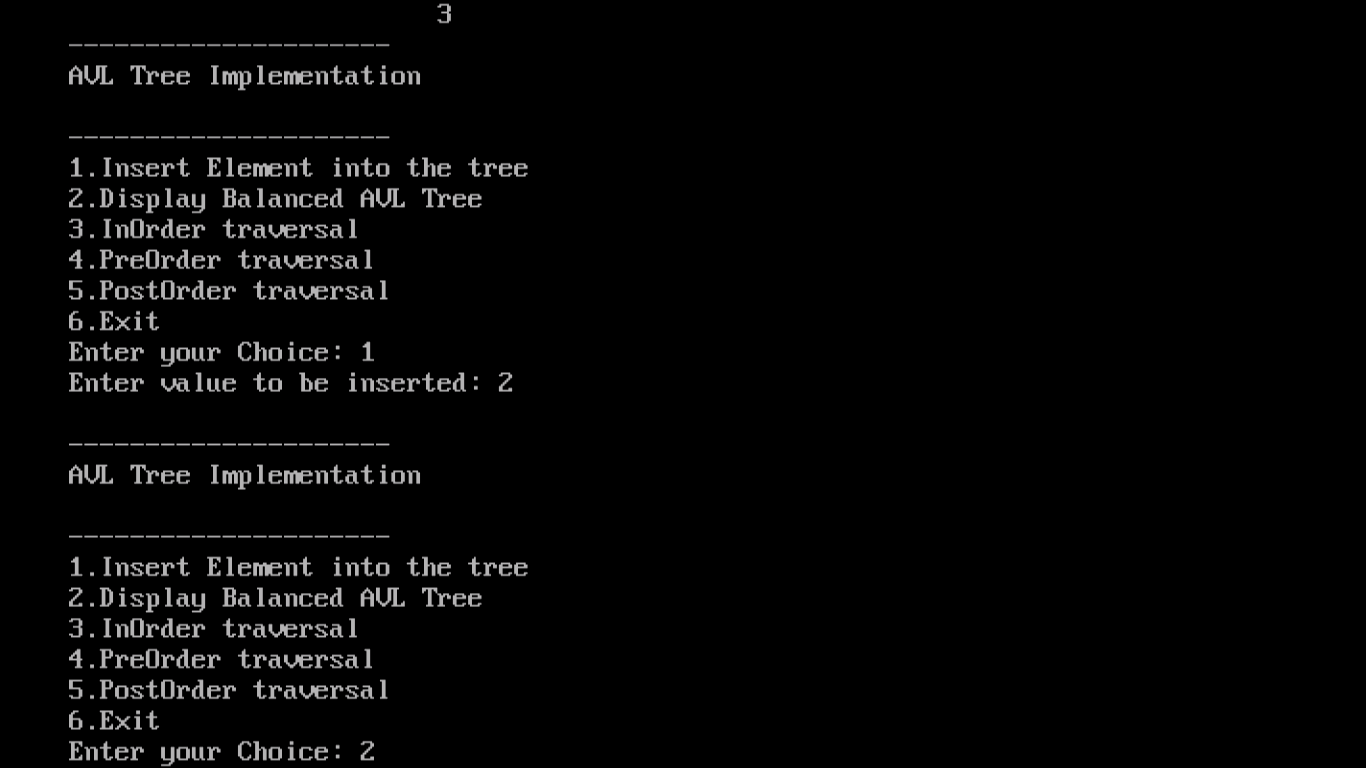
****

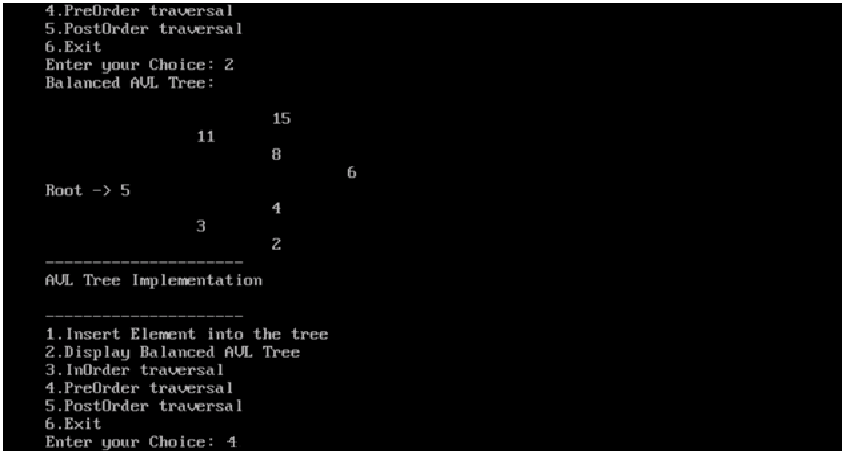
****

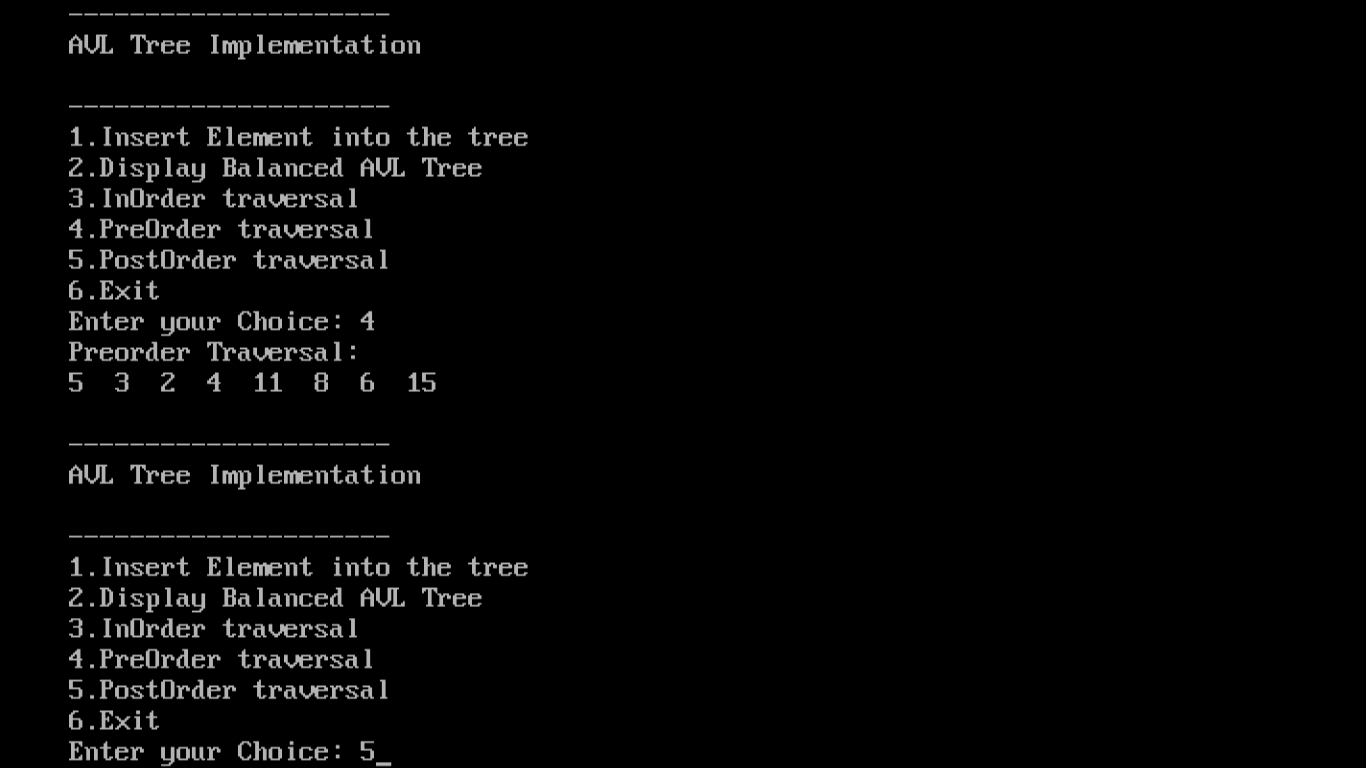
****

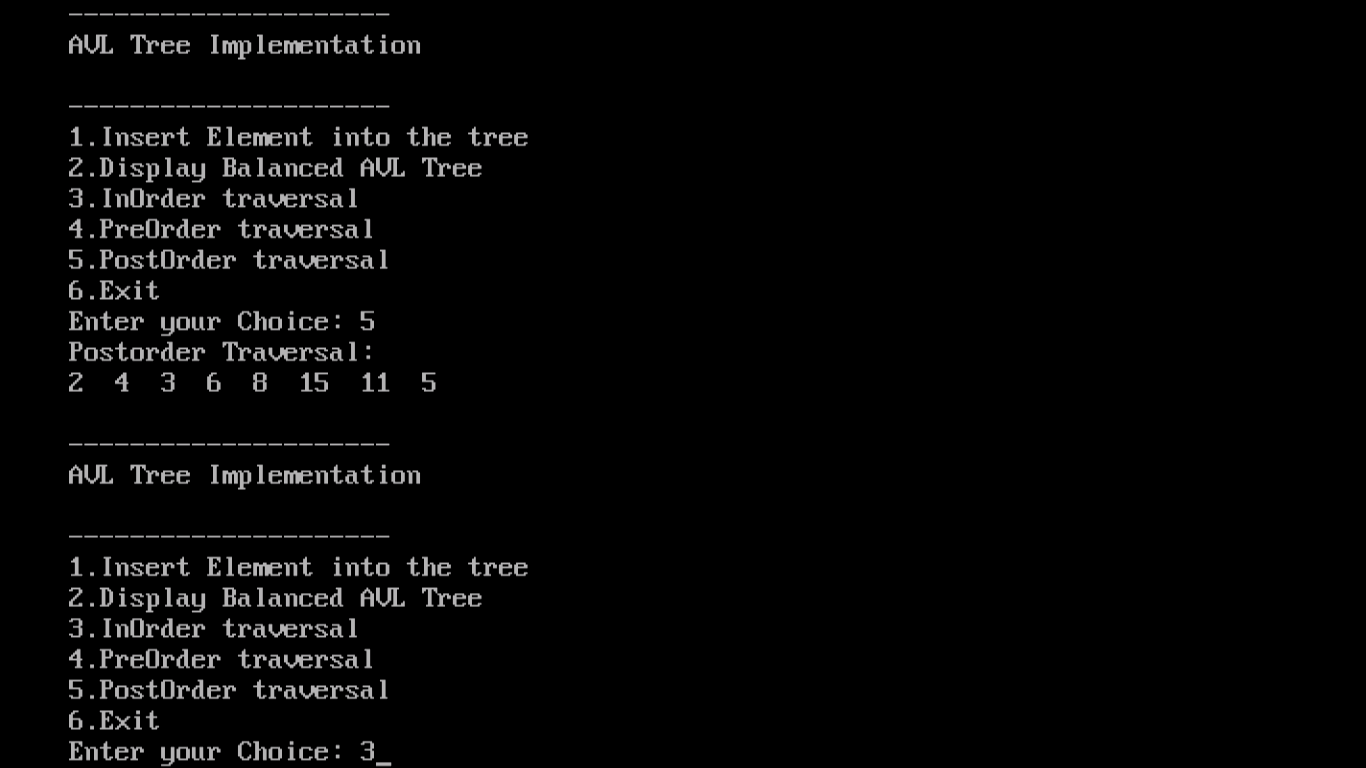
****

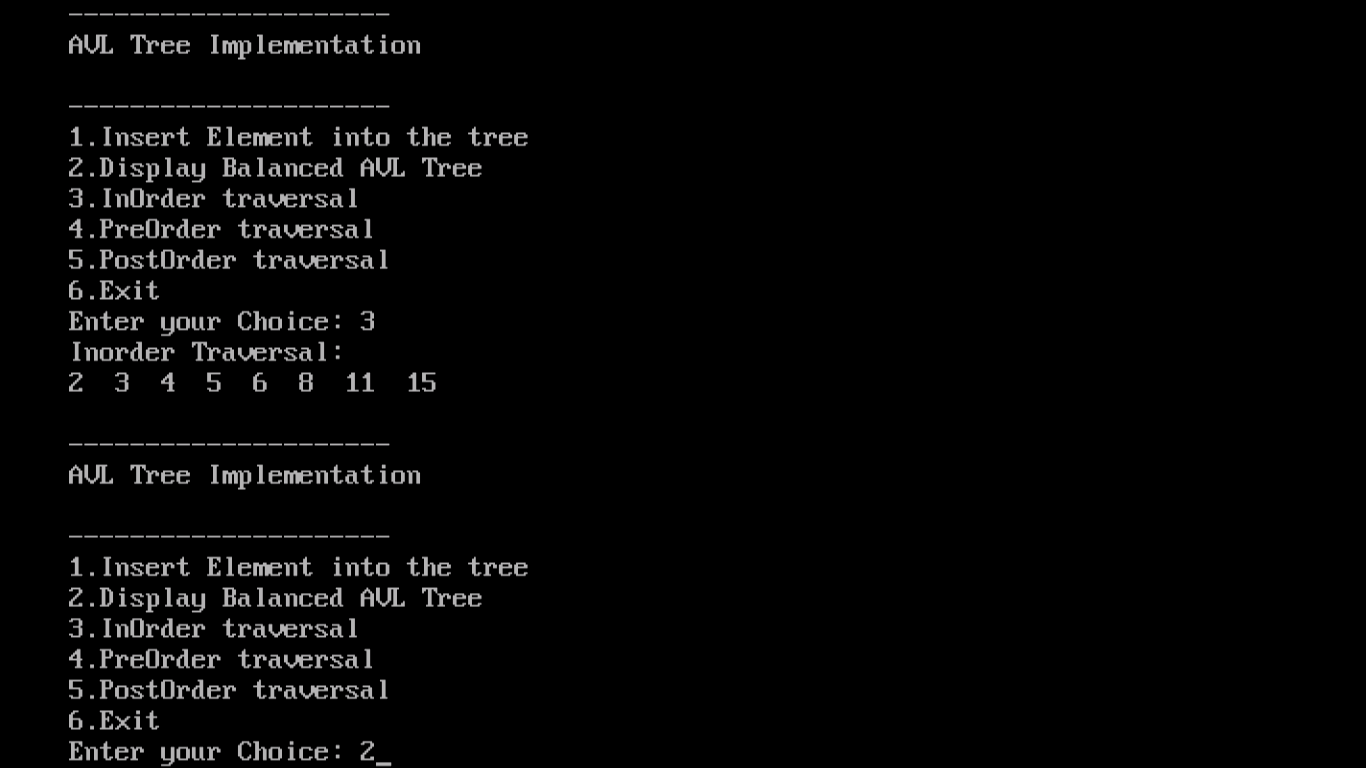
****

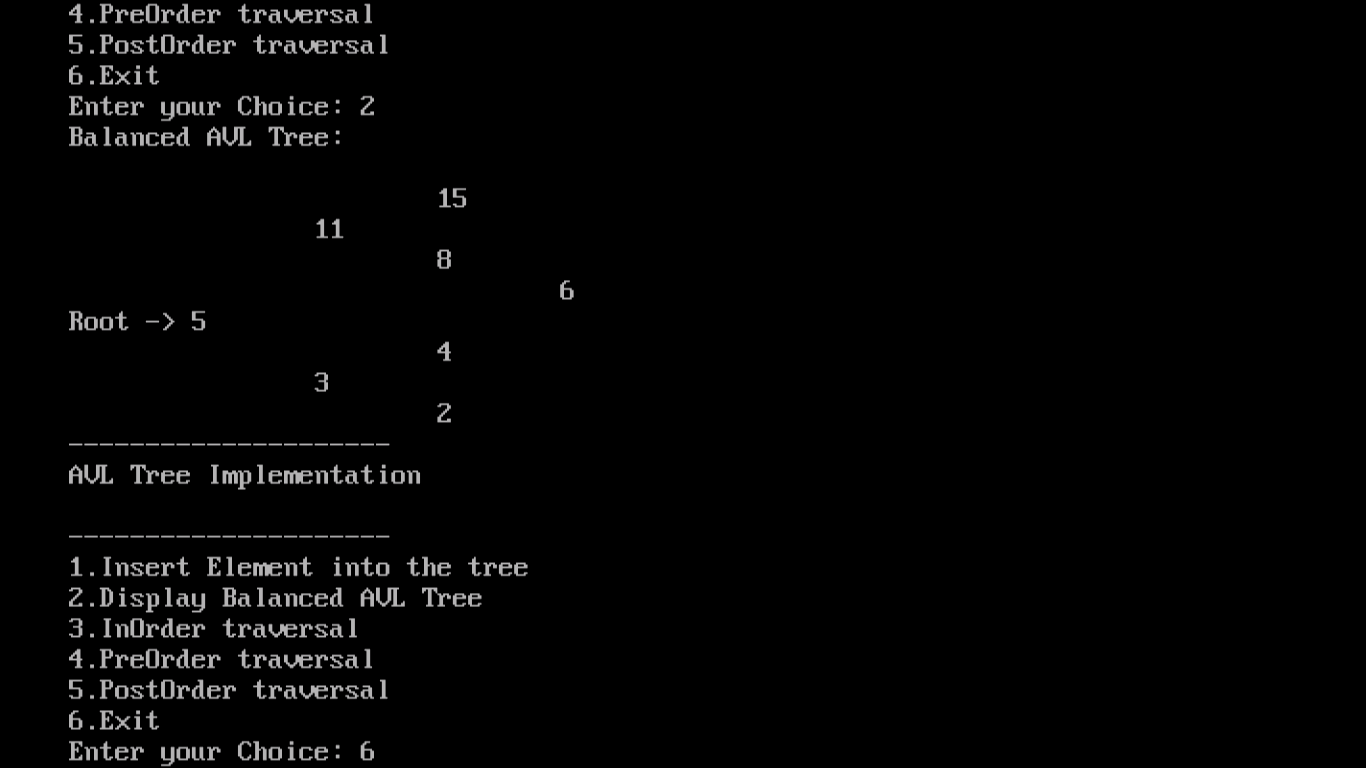
****

****

****

****

****

****

**17. PROGRAM TO IMPLEMENT BFS**

// Program to print BFS traversal from a given source vertex. BFS(int s)

// traverses vertices reachable from s.

#include<iostream>

#include <list>

using namespace std;

// This class represents a directed graph using adjacency list representation

class Graph

{

int V; // No. of vertices

list<int> \*adj; // Pointer to an array containing adjacency lists

public:

Graph(int V); // Constructor

void addEdge(int v, int w); // function to add an edge to graph

void BFS(int s); // prints BFS traversal from a given source s

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w); // Add w to v's list.

}

void Graph::BFS(int s)

{

// Mark all the vertices as not visited

bool\*visited = new bool[V];

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

visited[i] = false;

// Create a queue for BFS

list<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push\_back(s);

// 'i' will be used to get all adjacent vertices of a vertex

list<int>::iterator i;

while(!queue.empty())

{

// Dequeue a vertex from queue and print it

s = queue.front();

cout<< s << " ";

queue.pop\_front();

// Get all adjacent vertices of the dequeued vertex s

// If a adjacent has not been visited, then mark it visited

// and enqueue it

for(i = adj[s].begin(); i != adj[s].end(); ++i)

{

if(!visited[\*i])

{

visited[\*i] = true;

queue.push\_back(\*i);

}

}

}

}

int main()

{

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout<< " \n Breadth First Traversal "

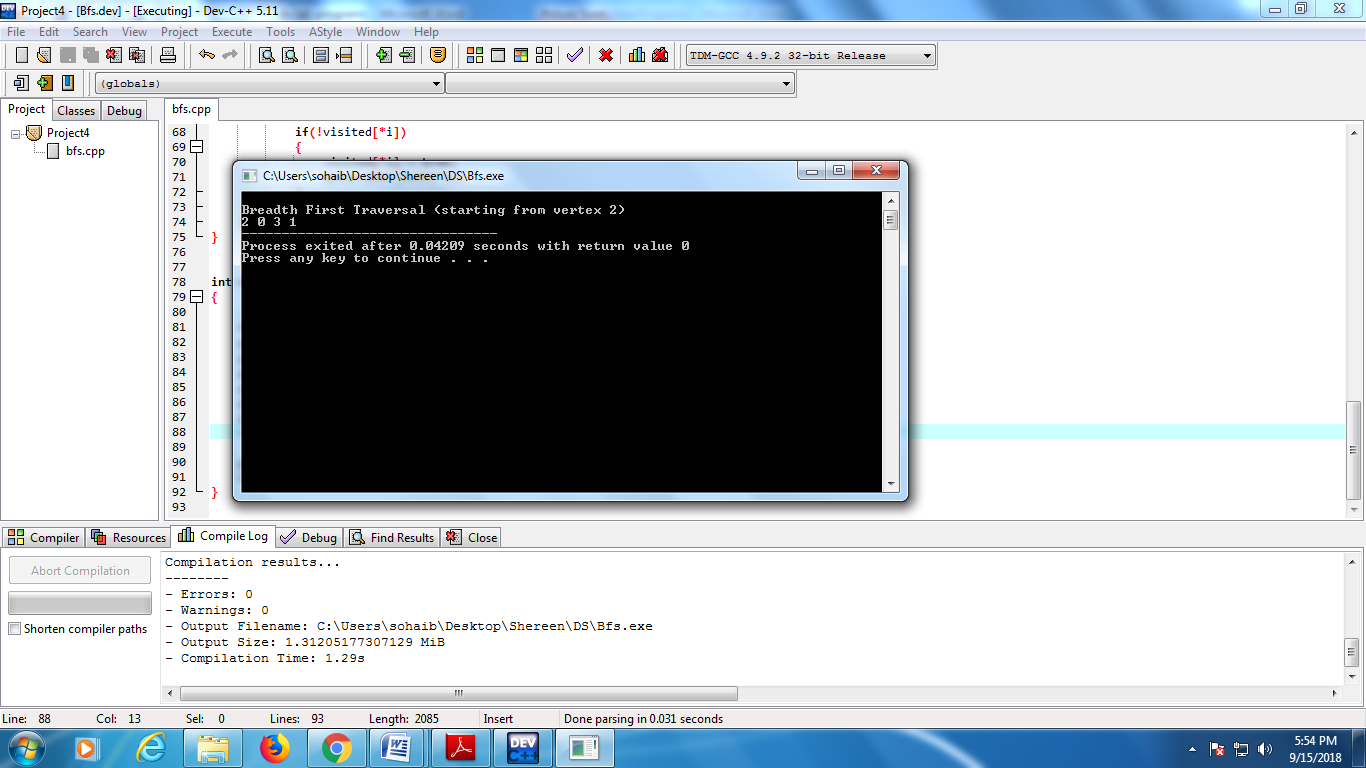
<< "(starting from vertex 2) \n";

g.BFS(2);

return 0;

}

**OUTPUT**

****

**18. PROGRAM TO CHECK WHETHER DIRECTED GRAPH IS CONNECTED USING DFS**

#include <iostream>

#include <list>

#include <stack>

using namespace std;

/\* Class Declaration\*/

class Graph

{

private:

int V;

list<int> \*adj;

void DFSUtil(int v, bool visited[]);

public:

Graph(int V)

{

this->V = V;

adj = new list<int>[V];

}

~Graph()

{

delete [] adj;

}

void addEdge(int v, int w);

bool isConnected();

Graph getTranspose();

};

/\*A recursive function to print DFS starting from v\*/

void Graph::DFSUtil(int v, bool visited[])

{

visited[v] = true;

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFSUtil(\*i, visited);

}

/\*Function that returns reverse (or transpose) of this graph\*/

Graph Graph::getTranspose()

{

Graph g(V);

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

{

list<int>::iterator i;

for(i = adj[v].begin(); i != adj[v].end(); ++i)

{

g.adj[\*i].push\_back(v);

}

}

return g;

}

/\* Add Edge to connect v and w \*/

void Graph::addEdge(int v, int w)

{

adj[v].push\_back(w);

}

/\* Check if Graph is Connected\*/

bool Graph::isConnected()

{

bool visited[V];

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

visited[i] = false;

DFSUtil(0, visited);

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

if (visited[i] == false)

return false;

Graph gr = getTranspose();

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

visited[i] = false;

gr.DFSUtil(0, visited);

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

if (visited[i] == false)

return false;

return true;

}

/\* Main Contains Menu\*/

int main()

{

Graph g1(5);

g1.addEdge(0, 1);

g1.addEdge(1, 2);

g1.addEdge(2, 3);

g1.addEdge(3, 0);

g1.addEdge(2, 4);

g1.addEdge(4, 2);

if (g1.isConnected())

cout<<"The Graph is Connected"<<endl;

else

cout<<"The Graph is not Connected"<<endl;

Graph g2(4);

g2.addEdge(0, 1);

g2.addEdge(1, 2);

g2.addEdge(2, 3);

if (g2.isConnected())

cout<<"The Graph is Connected"<<endl;

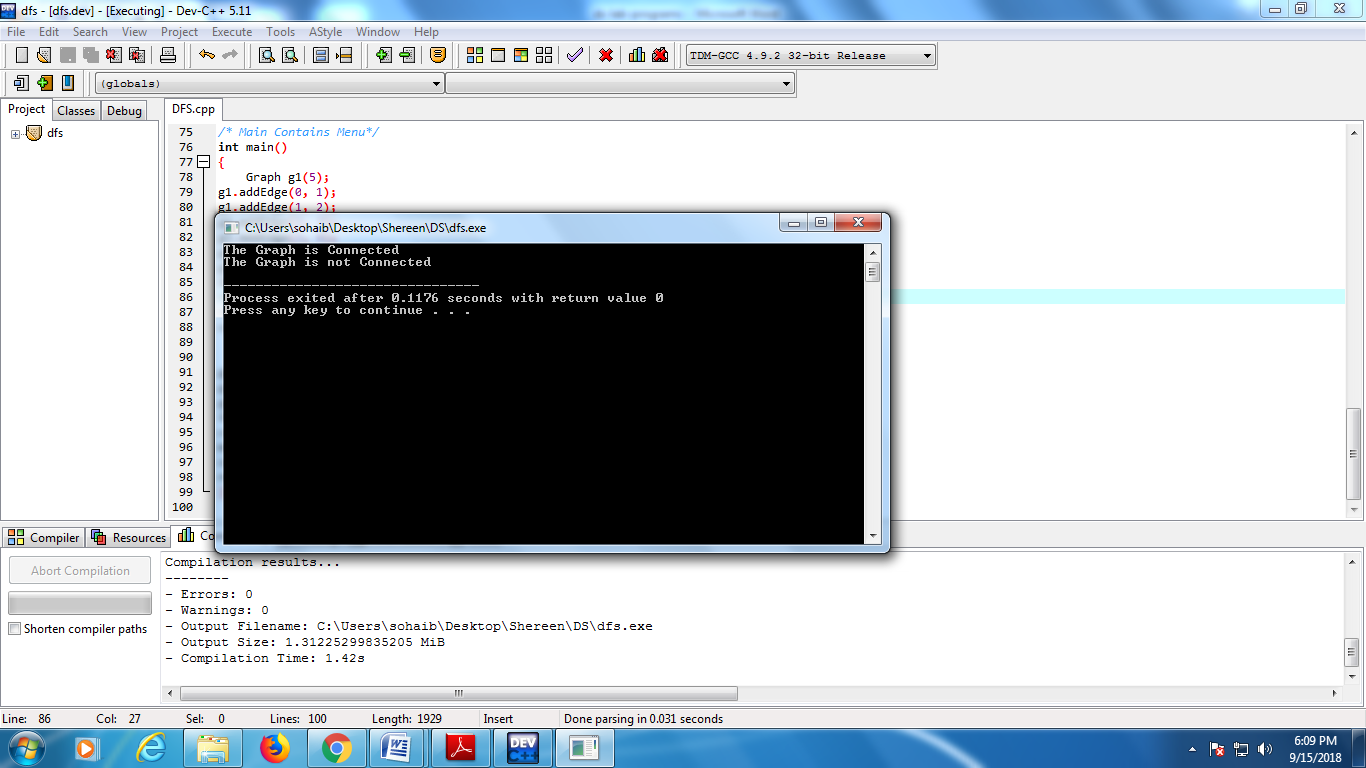
else

cout<<"The Graph is not Connected"<<endl;

return 0;

}

**OUTPUT**

****

**KRUSKAL’S ALGORITHM**

**Minimum Spanning Tree:**

Given a connected and undirected graph, a spanning tree of that graph is a sub graph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree. A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

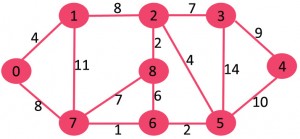
**Steps for finding MST using Kruskal’s algorithm**

1. Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.

3. Repeat step 2 until there are (V-1) edges in the spanning tree.

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not make a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

After sorting:

Weight Src Dest

1 7 6

2 8 2

2 6 5

4 0 1

4 2 5

6 8 6

7 2 3

7 7 8

8 0 7

8 1 2

9 3 4

10 5 4

11 1 7

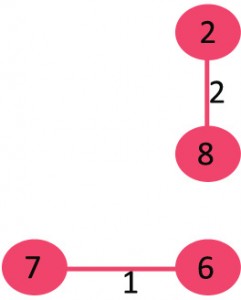
14 3 5

Now pick all edges one by one from sorted list of edges

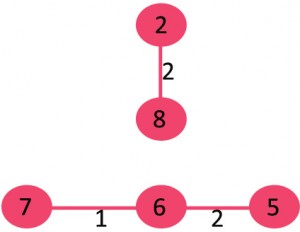
**1.** Pick edge 7-6: No cycle is formed, include it.



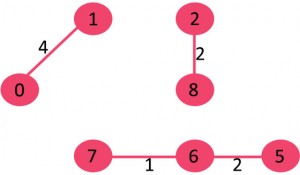
**2.** Pick edge 8-2: No cycle is formed, include it.



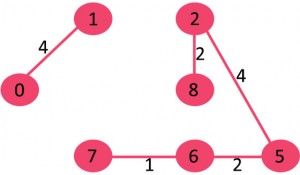
**3.** Pick edge 6-5: No cycle is formed, include it.



**4.** Pick edge 0-1: No cycle is formed, include it.

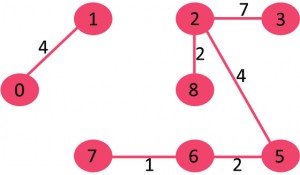


**5.** Pick edge 2-5: No cycle is formed, include it.



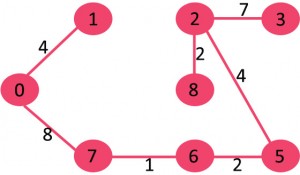
**6.** Pick edge 8-6: Since including this edge results in cycle, discard it.

**7.** Pick edge 2-3: No cycle is formed, include it.



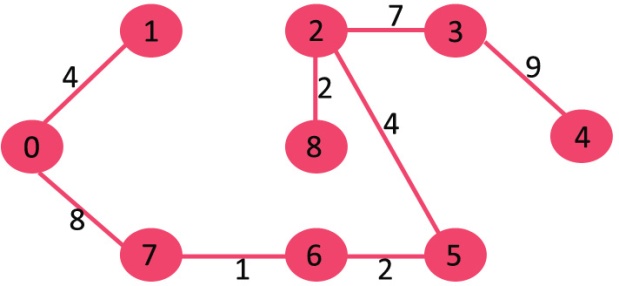
**8.** Pick edge 7-8: Since including this edge results in cycle, discard it.

**9.** Pick edge 0-7: No cycle is formed, include it.



**10.** Pick edge 1-2: Since including this edge results in cycle, discard it.

**11.** Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals (V – 1), the algorithm stops here.

**19. PROGRAM TO IMPLEMENT KRUSKAL’S ALGORITHM**

// C++ program for Kruskal's algorithm to find Minimum Spanning Tree

// of a given connected, undirected and weighted graph

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// a structure to represent a weighted edge in graph

struct Edge

{

int src, dest, weight;

};

// a structure to represent a connected, undirected and weighted graph

struct Graph

{

// V-> Number of vertices, E-> Number of edges

int V, E;

// graph is represented as an array of edges. Since the graph is

// undirected, the edge from src to dest is also edge from dest

// to src. Both are counted as 1 edge here.

struct Edge\* edge;

};

// Creates a graph with V vertices and E edges

struct Graph\* createGraph(int V, int E)

{

struct Graph\* graph = (struct Graph\*) malloc( sizeof(struct Graph) );

graph->V = V;

graph->E = E;

graph->edge = (struct Edge\*) malloc( graph->E \* sizeof( struct Edge ) );

return graph;

}

// A structure to represent a subset for union-find

struct subset

{

int parent;

int rank;

};

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

// (uses path compression technique)

int find(struct subset subsets[], int i)

{

// find root and make root as parent of i (path compression)

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

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

return subsets[i].parent;

}

// A function that does union of two sets of x and y

// (uses union by rank)

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

{

int xroot = find(subsets, x);

int yroot = find(subsets, y);

// Attach smaller rank tree under root of high rank tree

// (Union by Rank)

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

subsets[xroot].parent = yroot;

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

subsets[yroot].parent = xroot;

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

// its rank by one

else

{

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

// Compare two edges according to their weights.

// Used in qsort() for sorting an array of edges

int myComp(const void\* a, const void\* b)

{

struct Edge\* a1 = (struct Edge\*)a;

struct Edge\* b1 = (struct Edge\*)b;

return a1->weight > b1->weight;

}

// The main function to construct MST using Kruskal's algorithm

void KruskalMST(struct Graph\* graph)

{

int V = graph->V;

struct Edge result[V]; // Tnis will store the resultant MST

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

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

// Step 1: Sort all the edges in non-decreasing order of their weight

// If we are not allowed to change the given graph, we can create a copy of

// array of edges

qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

// Allocate memory for creating V ssubsets

struct subset \*subsets =

(struct subset\*) malloc( V \* sizeof(struct subset) );

// Create V subsets with single elements

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

{

subsets[v].parent = v;

subsets[v].rank = 0;

}

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

while(e < V - 1)

{

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

// for next iteration

struct Edge next\_edge = graph->edge[i++];

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

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

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

// in result and increment the index of result for next edge

if(x != y)

{

result[e++] = next\_edge;

Union(subsets, x, y);

}

// Else discard the next\_edge

}

// print the contents of result[] to display the built MST

printf("Following are the edges in the constructed MST\n");

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

printf("%d -- %d == %d\n", result[i].src, result[i].dest,

result[i].weight);

return;

}

int main()

{

/\* Let us create following weighted graph

             10

        0--------1

        |  \     |

       6|   5\   |15

        |      \ |

        2--------3

            4       \*/

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

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

struct Graph\* graph = createGraph(V, E);

// add edge 0-1

graph->edge[0].src = 0;

graph->edge[0].dest = 1;

graph->edge[0].weight = 10;

// add edge 0-2

graph->edge[1].src = 0;

graph->edge[1].dest = 2;

graph->edge[1].weight = 6;

// add edge 0-3

graph->edge[2].src = 0;

graph->edge[2].dest = 3;

graph->edge[2].weight = 5;

// add edge 1-3

graph->edge[3].src = 1;

graph->edge[3].dest = 3;

graph->edge[3].weight = 15;

// add edge 2-3

graph->edge[4].src = 2;

graph->edge[4].dest = 3;

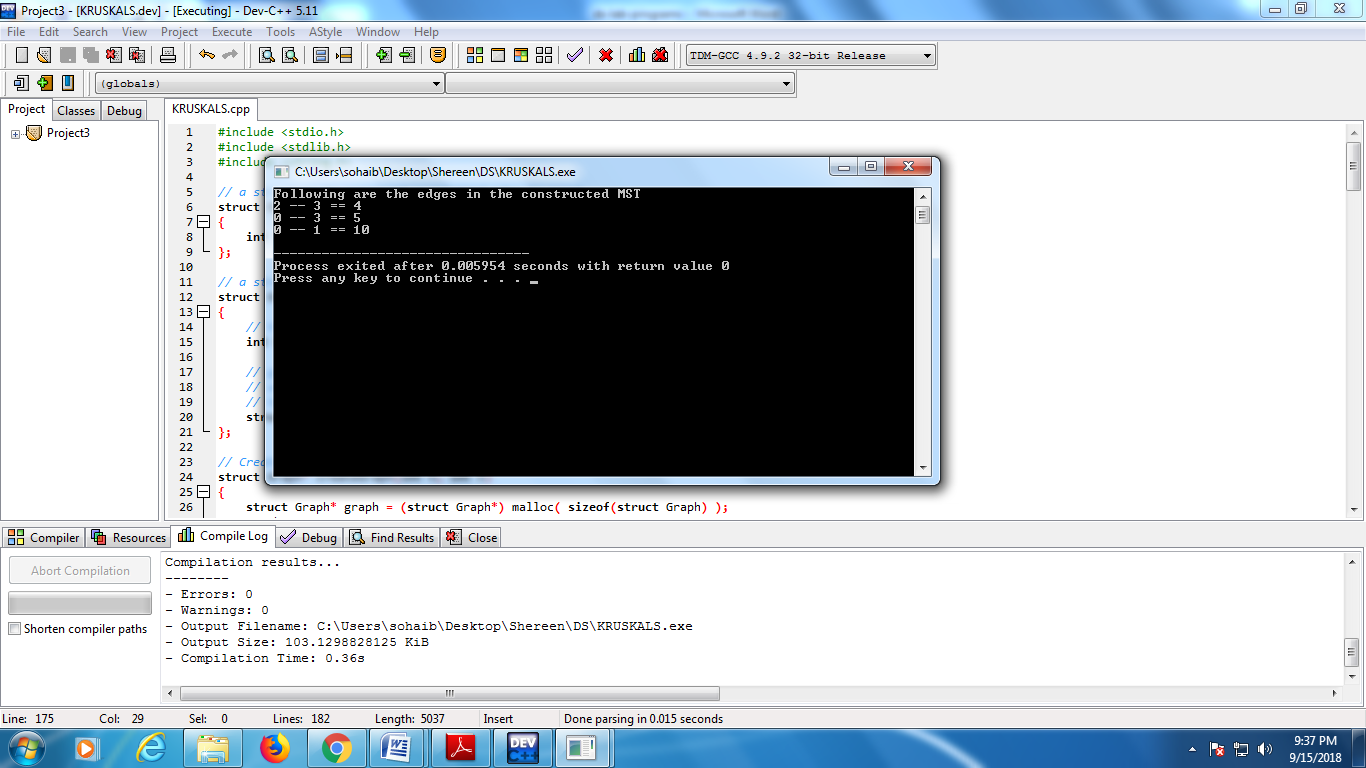
graph->edge[4].weight = 4;

KruskalMST(graph);

return 0;

}

**OUTPUT**



**PRIM’S ALGORITHM**

Like Kruskal’s algorithm, Prim’s algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST; the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other end point of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called cut in graph. So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

A spanning tree means all vertices must be connected. So the two disjoint subsets of vertices must be connected to make a Spanning *Tree*. And they must be connected with the minimum weight edge to make it a Minimum *Spanning Tree*.

**Algorithm**:

**1)** Create a set mstSet that keeps track of vertices already included in MST.

**2)** Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.

**3)** While mstSet doesn’t include all vertices

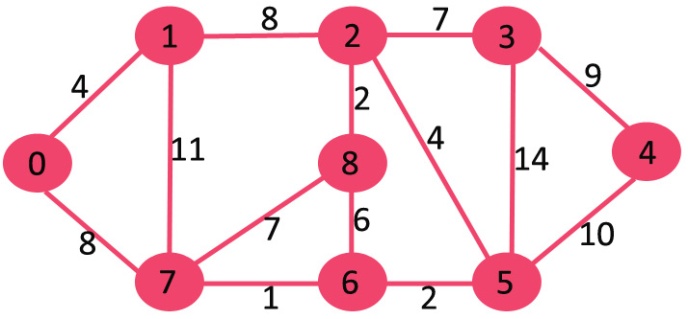
**a)** Pick a vertex u which is not there in mstSet and has minimum key value.

**b)** Include u to mstSet.

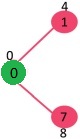
**c)** Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

Let us understand with the following example:



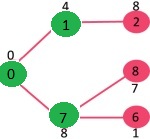
The set mstSet is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in mstSet. So mstSet becomes {0}. After including to mstSet, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in green color.



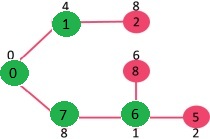
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until mstSet includes all vertices of given graph. Finally, we get the following graph.



**20. PROGRAM TO IMPLEMENT PRIM’S ALGORITHM TO GEMERATE A MINIMUM SPANNING TREE**

// A C / C++ program for Prim's Minimum Spanning Tree (MST) algorithm.

// The program is for adjacency matrix representation of the graph

#include <stdio.h>

#include <limits.h>

// Number of vertices in the graph

#define V 5

// A utility function to find the vertex with minimum key value, from

// the set of vertices not yet included in MST

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

{

// Initialize min value

int min = INT\_MAX, min\_index;

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

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

min = key[v], min\_index = v;

return min\_index;

}

// A utility function to print the constructed MST stored in parent[]

int printMST(int parent[], int n, int graph[V][V])

{

printf("Edge Weight\n");

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

printf("%d - %d %d \n", parent[i], i, graph[i][parent[i]]);

}

// Function to construct and print MST for a graph represented using adjacency

// matrix representation

void primMST(int graph[V][V])

{

int parent[V]; // Array to store constructed MST

int key[V]; // Key values used to pick minimum weight edge in cut

bool mstSet[V]; // To represent set of vertices not yet included in MST

// Initialize all keys as INFINITE

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

key[i] = INT\_MAX, mstSet[i] = false;

// Always include first 1st vertex in MST.

key[0] = 0; // Make key 0 so that this vertex is picked as first vertex

parent[0] = -1; // First node is always root of MST

// The MST will have V vertices

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

{

// Pick the minimum key vertex from the set of vertices

// not yet included in MST

int u = minKey(key, mstSet);

// Add the picked vertex to the MST Set

mstSet[u] = true;

// Update key value and parent index of the adjacent vertices of

// the picked vertex. Consider only those vertices which are not yet

// included in MST

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

// graph[u][v] is non zero only for adjacent vertices of m

// mstSet[v] is false for vertices not yet included in MST

// Update the key only if graph[u][v] is smaller than key[v]

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

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

}

// print the constructed MST

printMST(parent, V, graph);

}

int main()

{ /\* Let us create the following graph

2 3

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

| / \ |

6| 8/ \5 |7

| / \ |

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

9 \*/

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

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

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

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

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

};

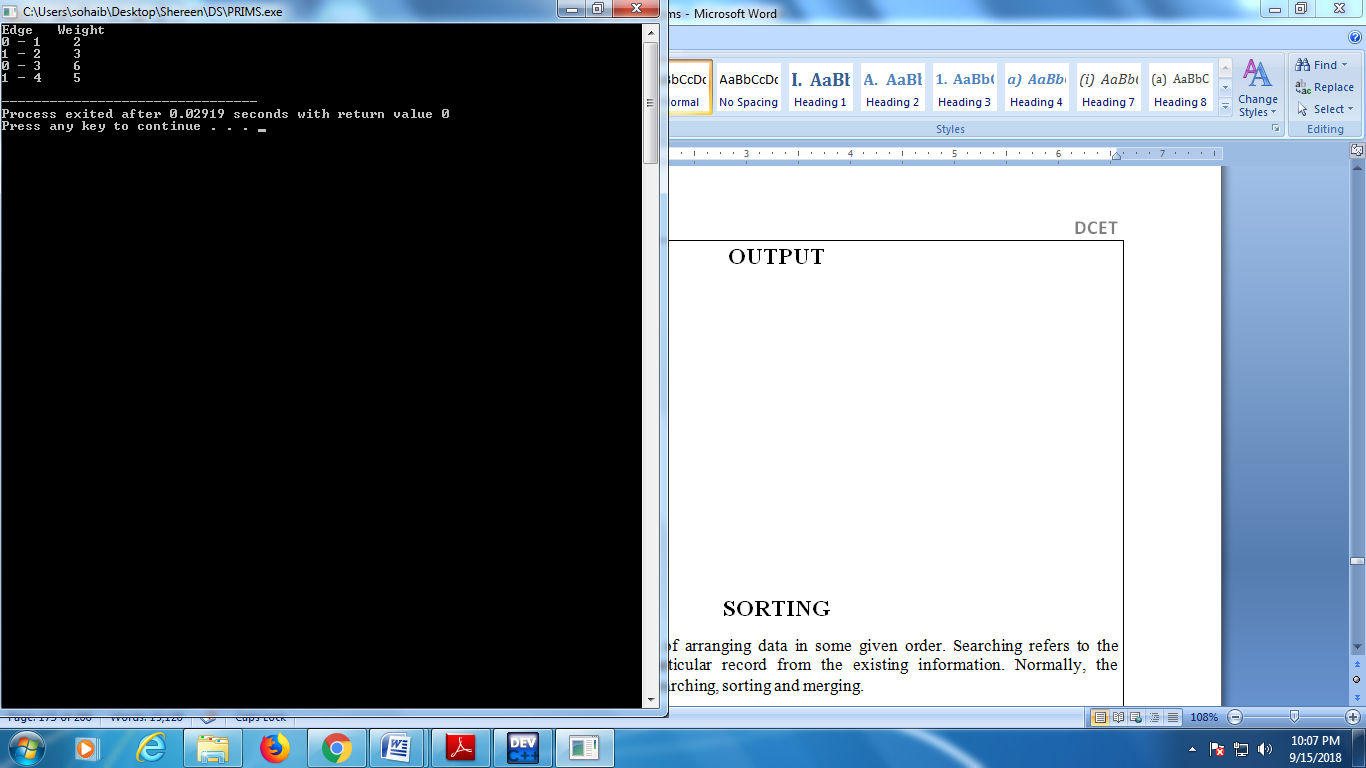
// Print the solution

primMST(graph);

return 0;

}

**OUTPUT**

****

**SORTING**

Sorting refers to the operation of arranging data in some given order. Searching refers to the operation of searching the particular record from the existing information. Normally, the information retrieval involves searching, sorting and merging.

Sorting is very important in every computer application. Sorting refers to arranging of data elements in some given order. Many Sorting algorithms are available to sort the given set of elements.

The two sorting techniques are:

Internal Sorting

External Sorting

**Internal Sorting:**

Internal Sorting takes place in the main memory of a computer. The internal sorting methods are applied to small collection of data. It means that, the entire collection of data to be sorted in small enough that the sorting can take place within main memory.

The following are the methods of internal sorting

1. Insertion sort

2. Selection sort

3. Merge Sort

4. Quick Sort

5. Heap Sort

**1. Insertion Sort:**

In this sorting we can read the given elements from 1 to n, inserting each element into its proper position. At every step, we insert the item into its proper place.

This sorting algorithm is frequently used when n is small. The insertion sort algorithm scans A from A[l] to A[N], inserting each element A[K] into its proper position in the previously sorted subarray A[l], A[2], . . . , A[K-1]. That is:

Pass 1. A[l] by itself is trivially sorted.

Pass 2. A[2] is inserted either before or after A[l] so that: A[l], A[2] is sorted.

Pass 3. A[3] is inserted into its proper place in A[l], A[2], that is, before A[l], between A[l] and A[2], or after A[2], so that: A[l], A[2], A[3] is sorted.

Pass 4. A[4] is inserted into its proper place in A[l], A[2], A[3] so that: A[l], A[2], A[3], A[4] is sorted.

………………………………………………………

Pass N. A[N] is inserted into its proper place in A[l],A[2], . . .,A[N - 1] so that:A[l],A[2],...,A[N] is sorted.

**Algorithm: Insertion Sort**

It works the way you might sort a hand of playing cards:

We start with an empty left hand [sorted array] and the cards face down on the table [unsorted array].

Then remove one card [key] at a time from the table [unsorted array], and insert it into the correct position in the left hand [sorted array].

To find the correct position for the card, we compare it with each of the cards already in the hand, from right to left.

Note that at all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

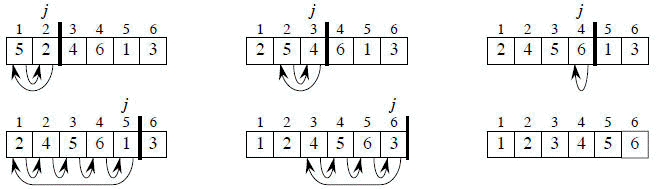
Pseudocode

We use a procedure INSERTION\_SORT. It takes as parameters an array A[1.. n] and the length n of the array. The array A is sorted in place: the numbers are rearranged within the array, with at most a constant number outside the array at any time.

  INSERTION\_SORT (A)

1.     for j ← 2 to length[A]   
2.             do  key ← A[j]      
3.                   {Put A[j] into the sorted sequence A[1 . . j − 1]}     
4.                    i ← j − 1      
5.                    while i > 0 and A[i] > key  
6.                                 do A[i +1] ← A[i]              
7.                                         i ← i − 1       
8.                     A[i + 1] ← key

**Example**: Following figure shows the operation of INSERTION-SORT on the array A= (5, 2, 4, 6, 1, 3). Each part shows what happens for a particular iteration with the value of j indicated. j indexes the "current card" being inserted into the hand.



Read the figure row by row. Elements to the left of A[j] that are greater than A[j] move one position to the right, and A[j] moves into the evacuated position.

**2. Selection Sort**:

In this sorting we find the smallest element in this list and put it in the first position. Then find the second smallest element in the list and put it in the second position. And so on.

Pass 1. Find the location LOC of the smallest in the list of N elements A[l], A[2], . . . , A[N], and then interchange A[LOC] and [1] .Then A[1] is sorted.

Pass 2. Find the location LOC of the smallest in the sublist of N – 1 Elements A[2],A[3],.,A[N], and then interchange A[LOC] and A[2]. Then: A[l], A[2] is sorted, since A[1]<A[2].

Pass 3. Find the location LOC of the smallest in the sublist of N – 2 elements A[3],A[4],...,A[N], and then interchange A[LOC] and A[3]. Then: A[l], A[2], . . . , A[3] is sorted, since A[2] <A[3].

………………………………

Pass N - 1. Find the location LOC of the smaller of the elements A[N - 1),A[N], and then interchange A[LOC] and A[N- 1]. Then: A[l],A[2], . . . , A[N] is sorted, since A[N - 1] < A[N].

Thus A is sorted after N - 1 passes.

 This type of sorting is called "Selection Sort" because it works by repeatedly element. It works as follows: first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

SELECTION\_SORT (A)

for i ← 1 to n-1 do  
    min j ← i;  
    min x ← A[i]  
    for j ← i + 1 to n do  
        If A[j] < min x then  
            min j ← j  
            min x ← A[j]  
    A[min j] ← A [i]  
    A[i] ← min x

 Selection sort is among the simplest of sorting techniques and it work very well for small files. Furthermore, despite its evident "naïve approach "Selection sort has a quite important application because each item is actually moved at most once, Section sort is a method of choice for sorting files with very large objects (records) and small keys.

**3.Merge Sort:**

Combing the two lists is called as merging. For example A is a sorted list with r elements and B is a sorted list with s elements. The operation that combines the elements of A and B into a single sorted list C with n = r + s elements is called merging. After combing the two lists the elements are sorted by using the following merging algorithm

The total computing time = O (n log2 n).

The disadvantages of using merge sort is that it requires two arrays of the same size and type for the merge phase.

Merge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with sub problems, we state each sub problem as sorting a subarray A[p .. r]. Initially, p = 1 and r = n, but these values change as we recurse through sub problems.

To sort A[p .. r]:

1. **Divide Step:** If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two subarrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].

2. **Conquer Step:** Conquer by recursively sorting the two subarrays A[p .. q] and A[q + 1 .. r].

3. **Combine Step:** Combine the elements back in A[p .. r] by merging the two sorted subarrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

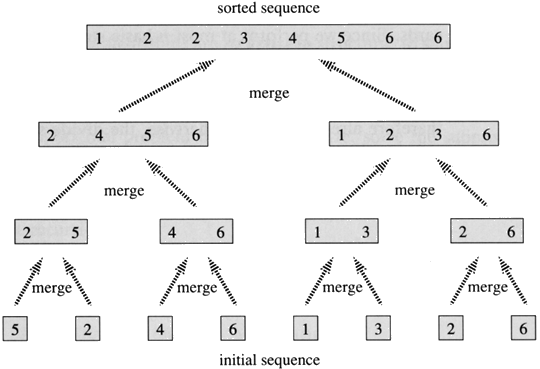
**Algorithm: Merge Sort**

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, 1, n).

MERGE-SORT (A, p, r)

1.     if p < r                                                    // Check for base case  
2.         then q = FLOOR[(p + r)/2]                 // Divide step  
3.                 MERGE (A, p, q)                          // Conquer step.  
4.                 MERGE (A, q + 1, r)                     // Conquer step.  
5.                 MERGE (A, p, q, r)                       // Conquer step.

Example: Bottom-up view of the merge sort procedure for n = 8.



**4. Quick Sort**

This is the most widely used internal sorting algorithm. It is based on divide-and-conquer type i.e. Divide the problem into sub-problems, until solved sub problems are found.

The Quick sort algorithm uses the O (n log n) comparisons on average.

It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good "general purpose" sort and it consumes relatively fewer resources during execution.

Quick sort works by partitioning a given array A[p . . r] into two non-empty sub array A[p . . q] and A[q+1 . . r] such that every key in A[p . . q] is less than or equal to every key in A[q+1 . . r]. Then the two subarrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

Given a sequence, S={S1,S2,…..,Sn}.Quick Sort performs the following steps.

Select one of the elements of ‘S’,the selected element ‘P’ is called Pivot/Partitioning Element.

Remove ‘P’ from ‘S’ and partition the remaining elements of ‘S’ into 2 subarrays L & G such that every element in ‘L’ is lesser than or equal to the Pivot element and every element in ‘G’ is greater than or equal to ‘P’.

Rearrange the elements of the sequences as follows.

S' = {L1,L2,L3,….,Ln,P,G1,G2,G3,……..,Gn}

Since all the elements to the left of partitioning element are less than or equal and all the elements to the right are greater than or equal to it, Partitioned element is kept in the sorted sequence.

Recursively Quick sort the unsorted sequence.

 QuickSort

1. If p < r then
2. q Partition (A, p, r)
3. Recursive call to Quick Sort (A, p, q)

   4. Recursive call to Quick Sort (A, q + r, r)

 Note that to sort entire array, the initial call Quick Sort (A, 1, length[A])

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

### Partitioning the Array

Partitioning procedure rearranges the subarrays in-place.

 PARTITION (A, p, r)

x ← A[p]

i ← p-1

j ← r+1

while TRUE do

    Repeat j ← j-1

    until A[j] ≤ x

    Repeat i ← i+1

    until A[i] ≥ x

    if i < j

        then exchange A[i] ↔ A[j]

        else return j

  Partition selects the first key, A[p] as a pivot key about which the array will partitioned:

Keys ≤  A[p] will be moved towards the left .  
Keys ≥ A[p] will be moved towards the right.

**5. Heap Sort:**

A Heap is a complete binary tree, in which each node satisfies the heap condition.

Heap condition: The key of each node is greater than or equal to the key in its children. Thus the root node will have the largest key value.

MaxHeap: Suppose H is a complete binary tree with n elements. Then H is called a heap or

maxheap, if the value at N is greater than or equal to the value at any of the children of N.

MinHeap: The value at N is less than or equal to the value at any of the children of N.

The operations on a heap

(i) New node is inserted into a Heap

(ii) Deleting the Root of a Heap

(i) Inserting a new element into a heap tree. Suppose H is a heap with N elements. We insert ITEM into the heap H as follows:

a. First join ITEM at the end of H.

b. Then the ITEM rise to its “appropriate place” in H.

(ii) Deleting the Root of a Heap: Deleting the root element from a heap tree. Suppose H is a heap with N elements. We delete the root R from the heap H as follows:

a. Assign the root R to some variable ITEM

b. Replace the deleted node R by the last node L of H

c. L sinks to its appropriate place in H.

The running time of heapsort is O(n log n) i.e. it achieves the lower bound for computational based sorting.

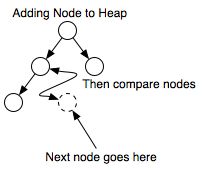
The HEAP data structure is an array object which can be easily visualized as a complete binary tree. There is a one to one correspondence between elements of the array and nodes of the tree. The tree is completely filled on all levels except possibly the lowest, which is filled from the left upto a point. All nodes of heap also satisfy the relation that the key value at each node is at least as large as the value at its children.

Step I: The user inputs the size of the heap (within a specified limit).The program generates a corresponding binary tree with nodes having randomly generated key Values.

Step II: Build Heap Operation: Let n be the number of nodes in the tree and i be the key of a tree. For this, the program uses operation Heapify. When Heapify is called both the left and right subtree of the i are Heaps. The function of Heapify is to let i settle down to a position(by swapping itself with the larger of its children, whenever the heap property is not satisfied)till the heap property is satisfied in the tree which was rooted at (i).This operation calls   
  
Step III: Remove maximum element: The program removes the largest element of the heap(the root) by swapping it with the last element.

Step IV: The program executes Heapify(new root) so that the resulting tree satisfies the heap property.

Step V: Goto step III till heap is empty



**EXTERNAL SORTING:**

The External sorting methods are applied only when the number of data elements to be sorted is too large. These methods involve as much external processing as processing in the CPU. To study the external sorting, we need to study the various external devices used for storage in addition to sorting algorithms. This sorting requires auxiliary storage. The following are the examples of external sorting

* Sorting with Disk
* Sorting with Tapes
* **TIME COMPLEXITIES OF SORTING TECHNIQUES:**

|  |  |  |  |
| --- | --- | --- | --- |
| SORTING TECHNIQUE | BEST CASE | AVG CASE | WORST CASE |
| MERGE SORT | O(nlogn) | O(nlogn) | O(nlogn) |
| QUICK SORT | O(nlogn) | O(nlogn) | O(n\*n) |
| HEAP SORT | O(nlogn) | O(nlogn) | O(nlogn) |
| INSERTION SORT | O(n) | O(n\*n) | O(n\*n) |
| SELECTION SORT | O(n\*n) | O(n\*n) | O(n\*n) |

**21. PROGRAM TO IMPLEMENT SELECTION SORT**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class selectsort

{

int a[50];

public:

void create(int);

int sort(int);

void display(int);

};

void selectsort::create(int n)

{

cout<<"Enter the numbers:";

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

{

cin>>a[i];

}

}

int selectsort::sort(int n)

{

for(int i=n;i>1;i--)

{

int max=0;

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

{

if(a[j]>a[max])

max=j;

}

int t;

t=a[i-1];

a[i-1]=a[max];

a[max]=t;

}

return 1;

}

void selectsort::display(int n)

{

cout<<"The elements are:"<<endl;

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

cout<<a[k]<<endl;

}

void menu()

{

cout<<endl;

cout<<"1.Create 2.Sort 3.Display 4.Exit"<<endl;

cout<<"Enter the choice"<<endl;

}

void main()

{

selectsort s;

int ch,n;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:cout<<"Enter the size of array:";

cin>>n;

s.create(n);

break;

case 2:s.sort(n);

cout<<"The array is sorted"<<endl;

break;

case 3:s.display(n);

break;

case 4:exit(0);

}

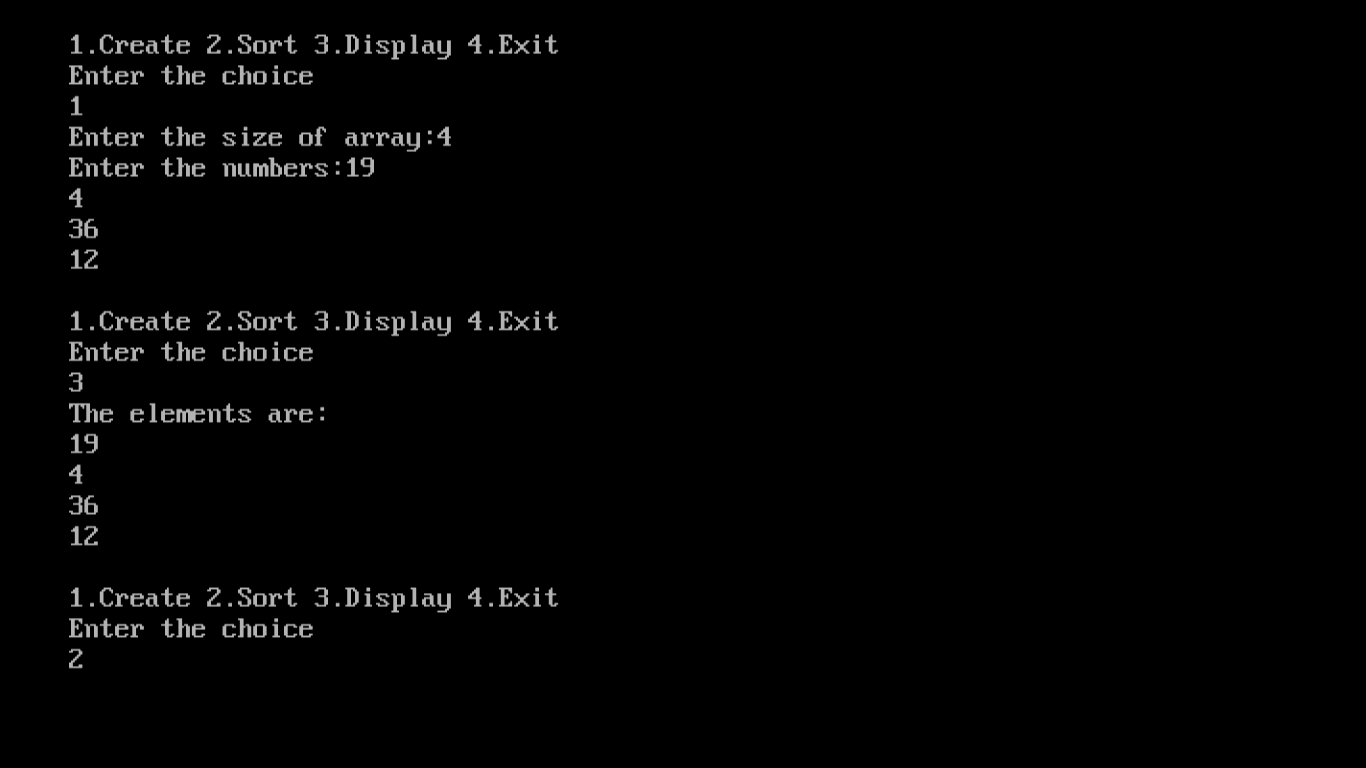
menu();

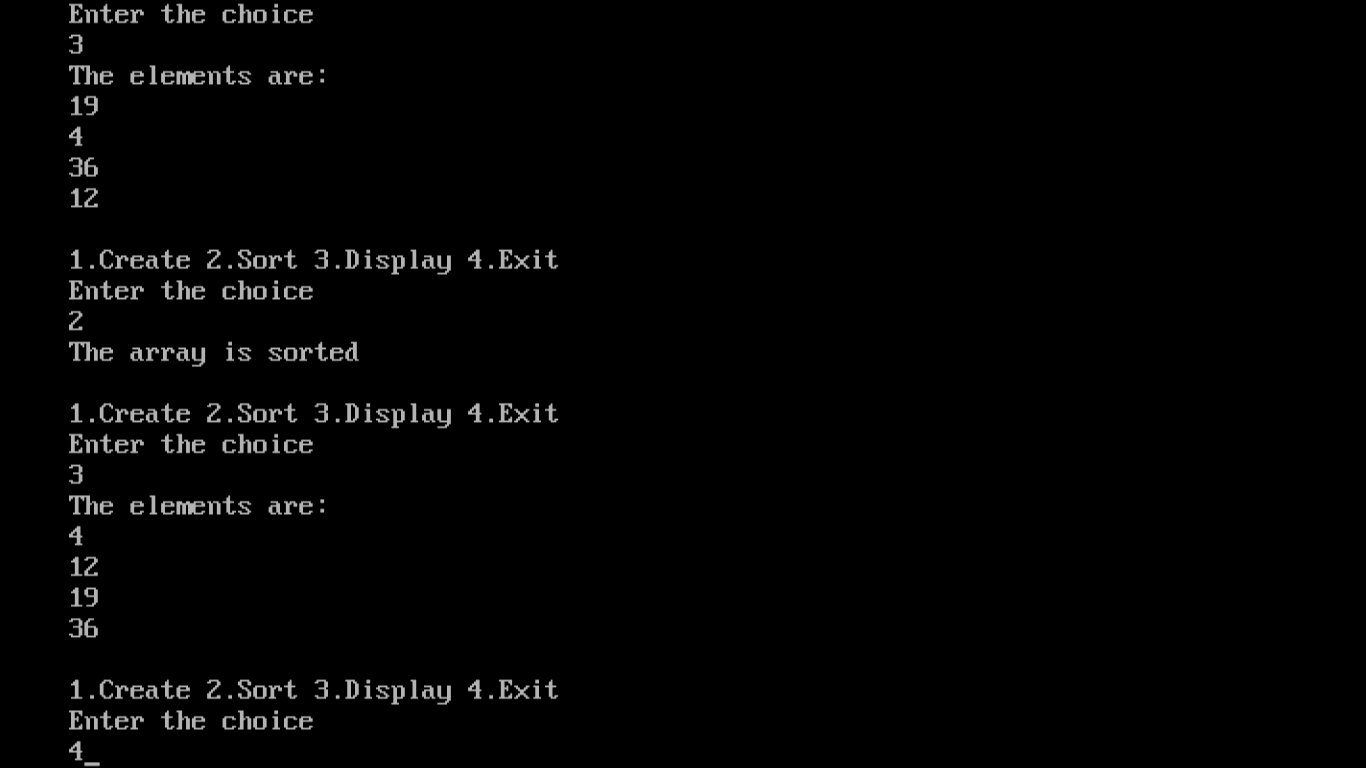
cin>>ch;

}

getch();}

**OUTPUT**

**

******

**22. PROGRAM TO IMPLEMENT QUICK SORT**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class quick

{

int a[50];

public:

void create(int);

void sort(int,int);

int partition(int \*,int,int);

void display(int);

};

void quick::create(int n)

{

cout<<"Enter the numbers:"<<endl;

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

{

cin>>a[i];

}

}

void quick::sort(int p,int q)

{

if(p<q)

{

int j=partition(a,p,q+1);

sort(p,j-1);

sort(j+1,q);

}

}

int quick::partition(int \*a,int m,int p)

{

int v=a[m];

int i=m,j=p;

while

(i<j)

{

do

{

i++;

}

while(a[i]<v);

do

{

j--;

}

while(a[j]>v);

if(i<j)

{

int t;

t=a[i];

a[i]=a[j];

a[j]=t;

}

}

a[m]=a[j];

a[j]=v;

return j;

}

void quick::display(int n)

{

cout<<"The sorted array is:"<<endl;

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

cout<<a[i]<<endl;

}

void menu()

{

cout<<endl;

cout<<"1.Create 2.Sort 3.Display 4.Exit"<<endl;

cout<<"Enter the choice:"<<endl;

}

void main()

{

quick q;

int ch,n;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:cout<<"Enter the size of array:";

cin>>n;

q.create(n);

break;

case 2:q.sort(1,n);

cout<<"Array is sorted"<<endl;

break;

case 3:q.display(n);

break;

case 4:exit(0);

}

menu();

cin>>ch;

}

getch();

}

**OUTPUT**

******

******

**23. PROGRAM TO IMPLEMENT MERGE SORT**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class mrgsrtdemo

{

int a[50];

public:

void create(int);

void merge\_sort(int,int);

void merge(int,int,int);

void display(int);

};

void mrgsrtdemo::create(int n)

{

cout<<"\nEnter the numbers to be sorted:"<<endl;

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

{

cin>>a[i];

}

}

void mrgsrtdemo::merge\_sort(int low,int high)

{

int mid;

if(low<high)

{

mid=(low+high)/2;

merge\_sort(low,mid);

merge\_sort(mid+1,high);

merge(low,mid,high);

}

}

void mrgsrtdemo::merge(int low,int mid,int high)

{

int h,i,j,k,b[30];

h=low;

i=low;

j=mid+1;

while((h<=mid)&&(j<=high))

{

if(a[h]<=a[j])

{

b[i]=a[h];

h++;

}

else

{

b[i]=a[j];

j++;

}

i++;

}

if(h>mid)

{

for(k=j;k<=high;k++)

{

b[i]=a[k];

i++;

}

}

else

{

for(k=h;k<=mid;k++)

{

b[i]=a[k];

i++;

}

}

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

a[k]=b[k];

}

void mrgsrtdemo::display(int n)

{

cout<<"\n The sorted array is :"<<endl;

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

cout<<a[i]<<endl;

}

void menu()

{

cout<<endl;

cout<<"1.Create 2.Sort 3.Display 4.Exit"<<endl;

cout<<"Enter your choice"<<endl;

}

void main()

{

mrgsrtdemo m;

int ch,n;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:cout<<"Enter the size of array:";

cin>>n;

m.create(n);

break;

case 2:m.merge\_sort(1,n);

cout<<"Array is sorted"<<endl;

break;

case 3:m.display(n);

break;

case 4:exit(0);

}

menu();

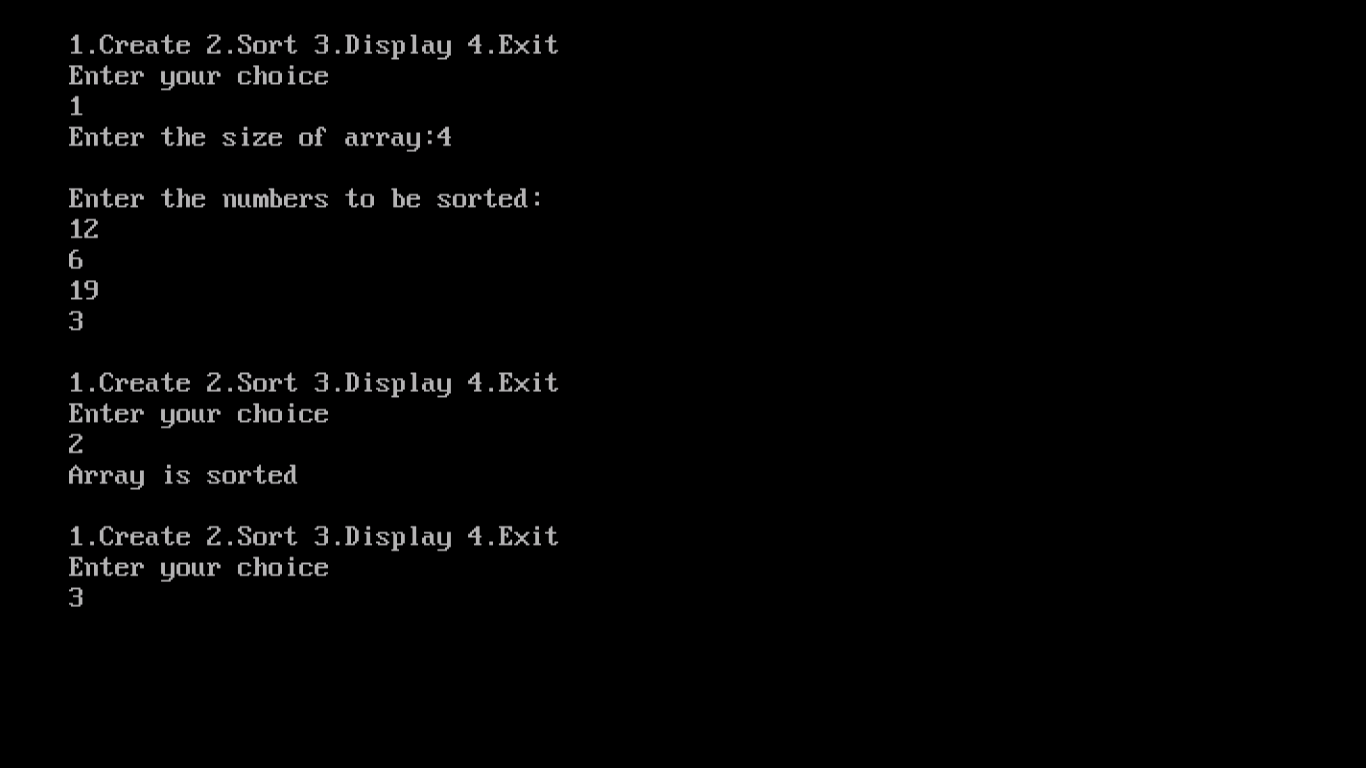
cin>>ch;

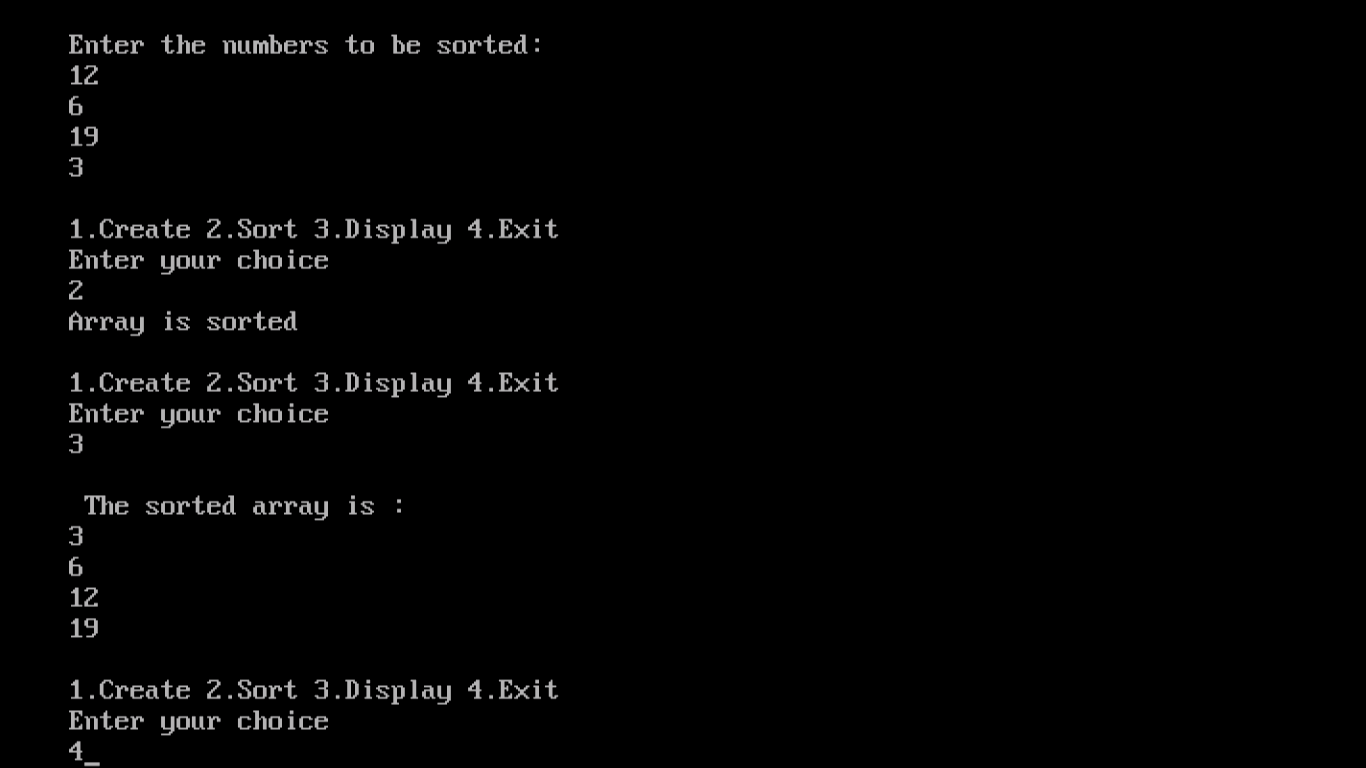
}

getch();

}

**OUTPUT**

**

******

**24. PROGRAM TO IMPLEMENT HEAP SORT**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class heap

{

int a[100];

public:

void create(int);

void heaps(int);

void hs(int \*,int);

void heapify(int \*,int);

void adjust(int \*,int,int);

void display(int);

};

void heap::create(int n)

{

cout<<"Enter the numbers:"<<endl;

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

{

cin>>a[i];

}

}

void heap::heaps(int n)

{

hs(a,n);

}

void heap::hs(int \*a,int n)

{

heapify(a,n);

for(int i=n;i>=2;i--)

{

int t=a[i];

a[i]=a[1];

a[1]=t;

adjust(a,1,i-1);

}

}

void heap::heapify(int \*a,int n)

{

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

adjust(a,i,n);

}

void heap::adjust(int \*a,int i,int n)

{

int j=2\*i;

int item=a[i];

while(j<=n)

{

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

{

j++;

}

if(item>=a[j])

break;

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

j=2\*j;

}

a[j/2]=item;

}

void heap::display(int n)

{

cout<<"The sorted array is:"<<endl;

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

cout<<a[i]<<endl;

}

void menu()

{

cout<<endl;

cout<<"1.Create 2.Sort 3.Display 4.Exit:"<<endl;

cout<<"Enter the choice:"<<endl;

}

void main()

{

heap h;

int a[100];

int ch,n;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:cout<<"Enter the size of array:";

cin>>n;

h.create(n);

break;

case 2:h.heaps(n);

cout<<"Array is sorted:"<<endl;

break;

case 3:cout<<"Heap elements are :"<<endl;

h.display(n);

break;

case 4:exit(0);

}

menu();

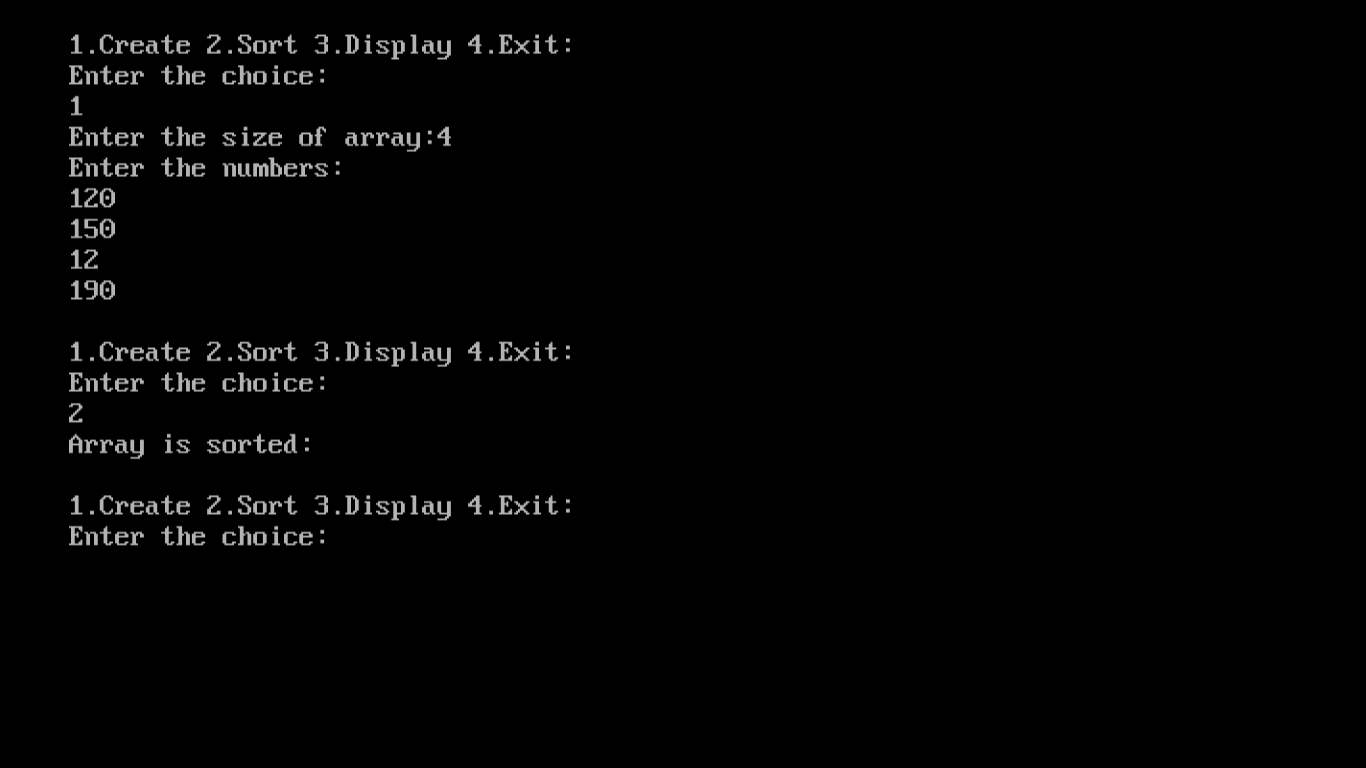
cin>>ch;

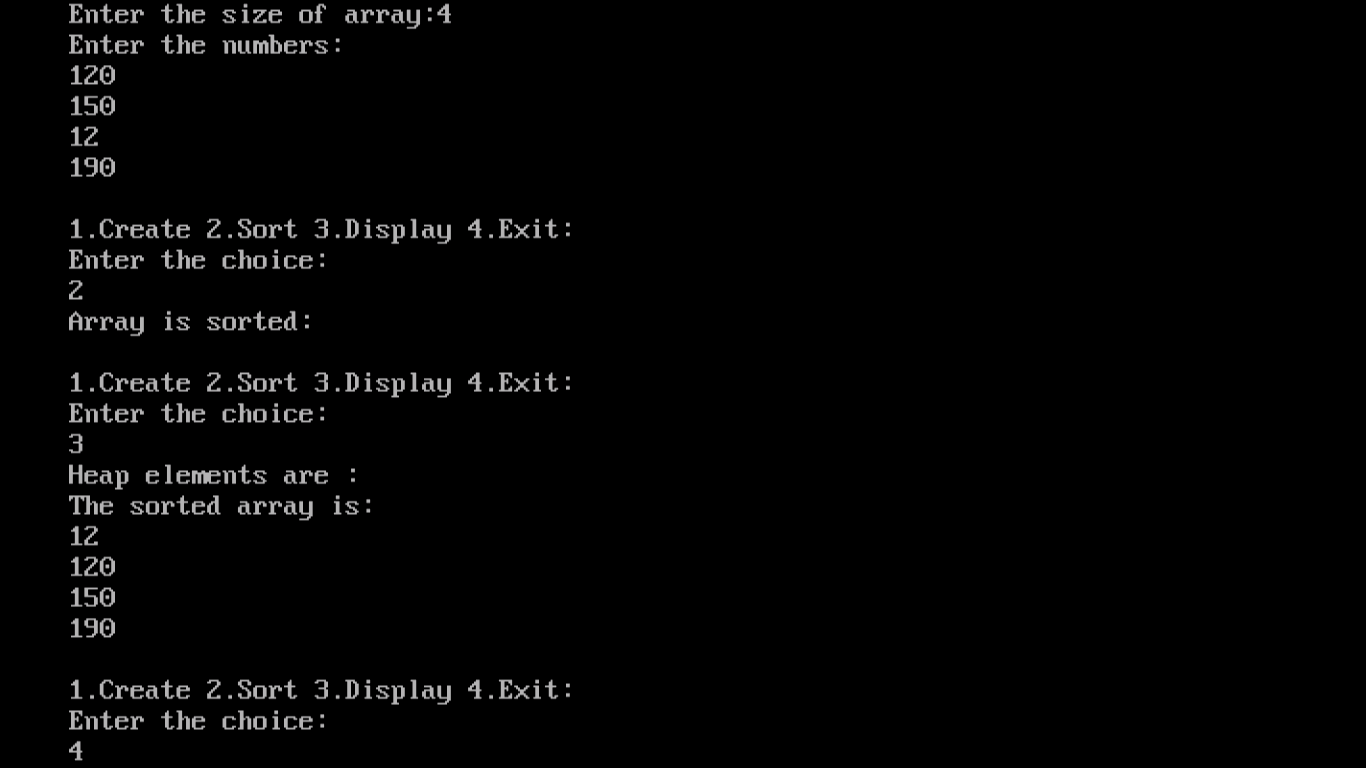
}

getch();

}

**OUTPUT**

******

**

**25. PROGRAM TO IMPLEMENT INSERTION SORT**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

class insort

{

int a[50];

public:

void create(int);

int sort(int);

void display(int);

};

void insort::create(int n)

{

cout<<"Enter the numbers:";

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

{

cin>>a[i];

}

}

int insort::sort(int n)

{

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

{

for(int j=i;j>0&&a[j-1]>a[j];j--)

{

int t;

t=a[j-1];

a[j-1]=a[j];

a[j]=t;

}

}

return 1;

}

void insort::display(int n)

{

cout<<"The elements are:"<<endl;

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

cout<<a[k]<<endl;

}

void menu()

{

cout<<endl;

cout<<"1.Create 2.Sort 3.Display 4.Exit"<<endl;

cout<<"Enter the choice"<<endl;

}

void main()

{

insort i;

int ch,n;

clrscr();

menu();

cin>>ch;

while(ch!=5)

{

switch(ch)

{

case 1:cout<<"Enter the size of the array";

cin>>n;

i.create(n);

break;

case 2:i.sort(n);

cout<<"The array is sorted"<<endl;

break;

case 3:i.display(n);

break;

case 4:exit(0);

}

menu();

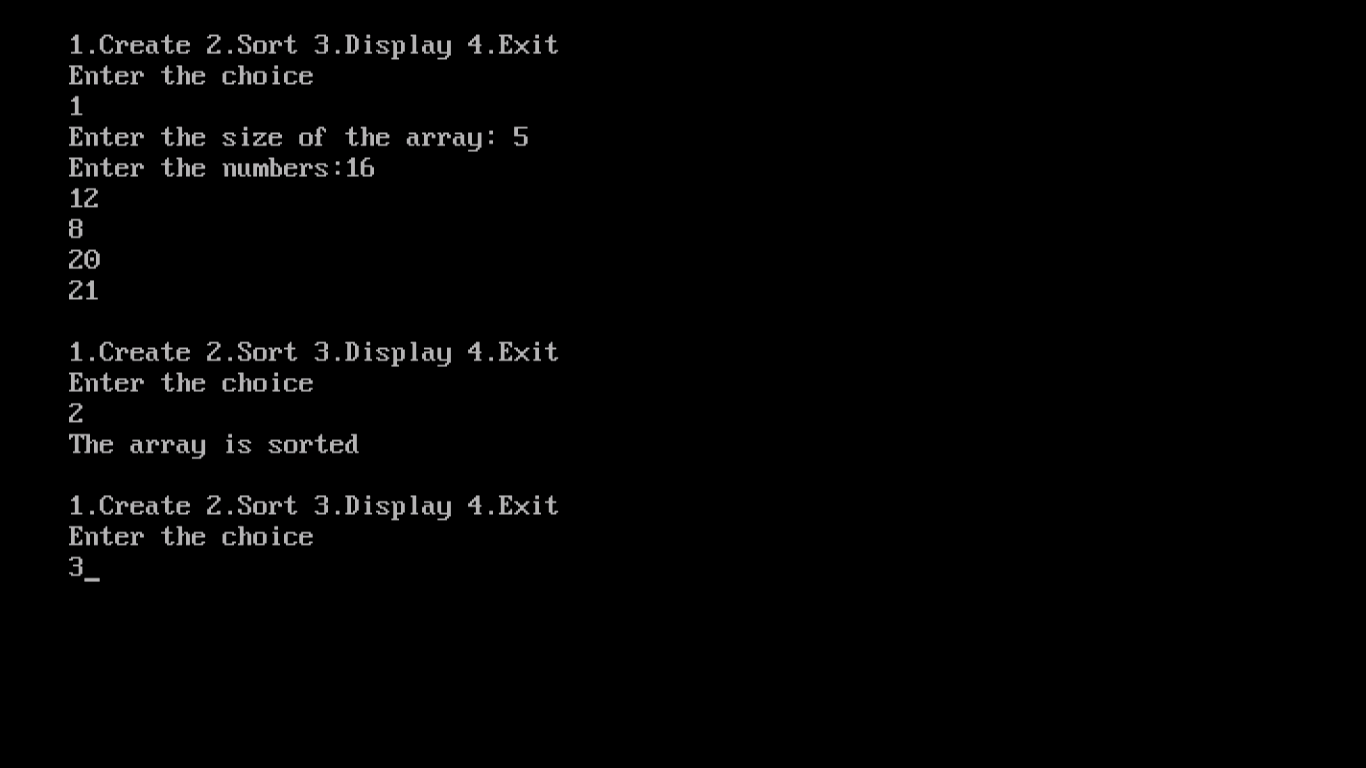
cin>>ch;

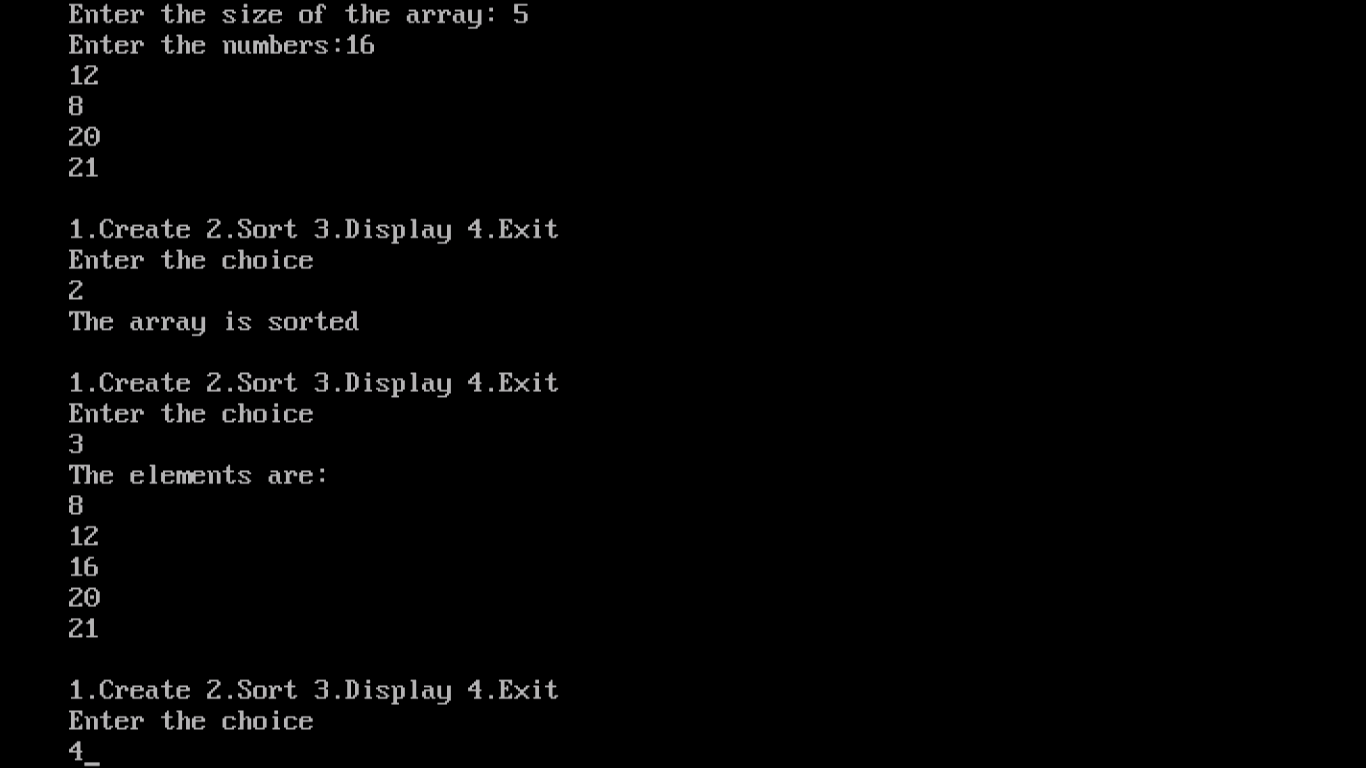
}

getch();

}

**OUTPUT**

**

****