Table of Content

S.No	Date	Experiment Name	Marks obtained	Page No	Sign
1		Array Implementation of List ADT			
2		Array implementation of Stack ADT			
3		Array implementation of Queue ADT			
Application of Stack and Queue ADT					
3 a		Evaluation of Suffix expression			
3 b		Balancing Parenthesis			
3 c		Tower of Hanoi problem with n disks			
3 d		Producer and consumer Problem			
Binary Search Tree					
4 a		Insertion, Deletion operation			
4 b		Tree Traversal			
4 c		Finding Min Element and Max Element			
Binary Heap and Graph					
5 a		Construct max heap to perform insertion and delete operations.			
5 b		Construct min heap to perform insertion and delete operations.			
6		Dijkstra's algorithm			
7		Prim's algorithm			
8		Hashing Technique			
Searching an element					
9 a		Linear Search			
9 b		Binary Search			
Sorting methods to arrange a list of integers in ascending order					
10 a		Insertion Sort			
10 b		Bubble Sort			
11		Heap Sort			
12 a		Quick Sort			
12 b		Merge Sort			
Real Time Problems					
13		Content Beyond Syllabus – Singly Linked List			

Ex.No: 01	Array Implementation of List ADT
Date :	

Aim

To write a C program to implement list ADT using Array.

Theory

A list is a sequential data structure, ie. a collection of items accessible one after another beginning at the head and ending at the tail.

It is a widely used data structure for applications which do not need random access Addition and removals can be made at any position in the list

lists are normally in the form of a_1, a_2, a_3.....an. The size of this list is n. The first element of the list is a1, and the last element is an. The position of element ai in a list is i.

List of size 0 is called as null list.

Algorithm

Step 1: Create nodes first, last; next, prev and cur then set the value as NULL.

Step 2: Read the list operation type.

Step 3: If operation type is create, then process the following steps.

- Allocate memory for node cur.
- Read data in cur's data area.
- Assign cur node as NULL.
- Assign first=last=cur.

Step 4: If operation type is Insert then process the following steps.

- Allocate memory for node cur.
- Read data in cur's data area.
- Read the position the Data to be insert.
- Availability of the position is true then assign cur's node as first and first = cur.
- If availability of position is false then do following steps.
    - Assign next as cur and count as zero.
    - Repeat the following steps until count less than position.

            - Assign prev as next
            - Next as prev of node.
            - Add count by one.
            - If prev as NULL then display the message INVALID POSITION.
            - If prev not qual to NULL then do the following steps.
                - ✓ Assign cur's node as prev's node.
                - ✓ Assign prev's node as cur.

Step5: If operation type is delete then do the following steps.

- Read the position.
- Check list is Empty. If it is true display the message List empty.
- If position is first.
    - Assign cur as first.
    - Assign First as first of node.
    - Reallocate the cur from memory.
    - If position is last.
        - Move the current node to prev.
        - cur's node as Null.
        - Reallocate the Last from memory.
        - Assign last as cur.
    - If position is enter Middle
        - Move the cur to required position.
        - Move the Previous to cur's previous position
        - Move the Next to cur's Next position.
        - Now Assign previous of node as next.
        - Reallocate the cur from memory.

Step 6: If operation is traverse.

- Assign current as first.
- Repeat the following steps until cur becomes NULL

Program without pointer

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
void create();
void insert();
void deletion();
void search();
void display();
int a,b[20], n, p, e, f, i, pos;
void main()
{
//clrscr();
int ch;
char g='y';
do
{
printf("\n main Menu");
printf("\n 1.Create \n 2.Delete \n 3.Search \n 4.Insert \n 5.Display\n 6.Exit \n");
printf("\n Enter your Choice");
scanf("%d", &ch);
switch(ch)
{
case 1:
create();
break;
case 2:
deletion();
break;
case 3:
search();
break;
case 4:
insert();
break;
case 5:
display();
break;
case 6:
exit();
break;
default:
printf("\n Enter the correct choice:");
```

```
}
printf("\n Do u want to continue:::");
scanf("\n%c", &g);
}
while(g=='y'||g=='Y');
getch();
}
void create()
{
printf("\n Enter the number of nodes");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n Enter the  Element:",i+1);
scanf("%d", &b[i]);
}
}
void deletion()
{
printf("\n Enter the position u want to delete::");
scanf("%d", &pos);
if(pos>=n)
{
printf("\n Invalid Location::");
}
else
{
for(i=pos+1;i<n;i++)
{
b[i-1]=b[i];
}
n--;
}
printf("\n The Elements after deletion");
for(i=0;i<n;i++)
{
printf("\t%d", b[i]);
}
}
void search()
{
printf("\n Enter the Element to be searched:");
scanf("%d", &e);
for(i=0;i<n;i++)
```

```
{
if(b[i]==e)
{
printf("Value is in the %d Position", i);
}
else
{
printf("Value %d is not in the list::", e);
continue;
}
}
}
void insert()
{
printf("\n Enter the position u need to insert::");
scanf("%d", &pos);

if(pos>=n)
 {
 printf("\n invalid Location::");
 }
 else
 {
 for(i=MAX-1;i>=pos-1;i--)
 {
 b[i+1]=b[i];
 }
 printf("\n Enter the element to insert::\n");
 scanf("%d",&p);
 b[pos]=p;
 n++;
 }
 printf("\n The list after insertion::\n");
display();
}
void display()
{
printf("\n The Elements of The list ADT are:");
for(i=0;i<n;i++)
{
printf("\n\n%d", b[i]);
}
}
```

Result

Thus, the Array implementation of List ADT were executed successfully and output has been verified.

Ex.No: 02 a	Array Implementation of Stack ADT
Date :	

Aim

To write a C program to implement stack ADT using Array.

Theory

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO(First In Last Out).

This strategy states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first. The stack is formed by using the array. All the operations regarding the stack are performed using arrays.

Algorithm

Step 1: Create stack array variable and top variable then set the top value as -1 and also mention the array size

Step 2: Read the list operation type.

Step 3: Perform the Push operation. It adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.

```
begin
 if stack is full
    return
 endif
else
 increment top
 stack[top] assign value
end else
end procedure
```

Step 4: Perform the pop operation. It removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

begin

if stack is empty

return

endif

else

store value of stack[top]

decrement top

return value

end else

end procedure

Step 5: Perform Peek operation. It returns the top element of the stack.

begin

return stack[top]

end procedure

If the stack is Empty it rReturns true if the stack is empty, else false.

begin

if top < 1

return true

else

return false

end procedure

Program

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
    //clrscr();
    top=-1;
    printf("\n Enter the size of STACK[MAX=100]:");
    scanf("%d",&n);
```

```
    printf("\n\t STACK OPERATIONS USING ARRAY");
    printf("\n\t--------------------------------");
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
    {
        printf("\n Enter the Choice:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                printf("\n\t EXIT POINT ");
                break;
            }
            default:
            {
                printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
            }

        }
    }
    while(choice!=4);
    return 0;
}
void push()
{
    if(top>=n-1)
    {
        printf("\n\tSTACK is over flow");
```

```
    }
    else
    {
        printf(" Enter a value to be pushed:");
        scanf("%d",&x);
        top++;
        stack[top]=x;
    }
}
void pop()
{
    if(top<=-1)
    {
        printf("\n\t Stack is under flow");
    }
    else
    {
        printf("\n\t The popped elements is %d",stack[top]);
        top--;
    }
}
void display()
{
    if(top>=0)
    {
        printf("\n The elements in STACK \n");
        for(i=top; i>=0; i--)
            printf("\n%d",stack[i]);
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }

}
```

Output

Enter the size of STACK[MAX=100]:10

STACK OPERATIONS USING ARRAY

1.PUSH
2.POP
3.DISPLAY
4.EXIT

Enter the Choice:1

Enter a value to be pushed:12

Enter the Choice:1

Enter a value to be pushed:24

Enter the Choice:1

Enter a value to be pushed:98

Enter the Choice:3

The elements in STACK

98

24

12

Press Next Choice

Enter the Choice:2

The popped elements is 98

Enter the Choice:3

The elements in STACK

24

12

Press Next Choice

Enter the Choice:4

Result

Thus, the Array implementation of stack ADT has been executed successfully and output is verified.

Ex.No: 02 b	Array Implementation of Queue ADT
Date :	

Aim

To write a C program to implement stack ADT using Array.

Theory

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple. Just define a one-dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1. Whenever, we want to insert a new value into the queue, increment 'rear' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

Algorithm

Step 1: Include all the header files which are used in the program and define a constant 'SIZE' with specific value.

Step 2: Declare all the user defined functions which are used in queue implementation.

Step 3: Create a one dimensional array with above defined SIZE (int queue[SIZE])

Step 4: Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1)

Step 5: Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

Step 6: enQueue(value) - Inserting value into the queue

- Check whether queue is FULL. (rear == SIZE-1)
- If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
- If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

Step 7: deQueue() - Deleting a value from the Queue

- Check whether queue is EMPTY. (front == rear)
- If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
- If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

Step 8: Displays the elements of a Queue

- Check whether queue is EMPTY. (front == rear)
- If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.
- If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.
- Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value reaches to rear (i <= rear)

Program

```
#include<stdio.h>
#include<conio.h>
#define SIZE 10
void enQueue(int);
void deQueue();
void display();
int queue[SIZE], front = -1, rear = -1;
void main()
{
  int value, choice;
  clrscr();
  while(1){
    printf("\n\n***** MENU *****\n");
    printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
    printf("\nEnter your choice: ");
```

```
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("Enter the value to be insert: ");
                scanf("%d",&value);
                enQueue(value);
                break;
        case 2: deQueue();
                break;
        case 3: display();
                break;
        case 4: exit(0);
        default: printf("\nWrong selection!!! Try again!!!");
    } } }
void enQueue(int value){
  if(rear == SIZE-1)
    printf("\nQueue is Full!!! Insertion is not possible!!!");
  else{
    if(front == -1)
        front = 0;
    rear++;
    queue[rear] = value;
    printf("\nInsertion success!!!");
  }}
void deQueue(){
  if(front == rear)
    printf("\nQueue is Empty!!! Deletion is not possible!!!");
  else{
    printf("\nDeleted : %d", queue[front]);
    front++;
```

```
    if(front == rear)
        front = rear = -1;
  }}
void display(){
  if(rear == -1)
    printf("\nQueue is Empty!!!");
  else{
    int i;
    printf("\nQueue elements are:\n");
    for(i=front; i<=rear; i++)
        printf("%d\t",queue[i]);
  }}
```

Output

Result

Thus, the Array implementation of queue ADT has been executed successfully and output is verified.

Ex.No: 03 a	Application of Stack and Queue ADT - Evaluation of Suffix expression
Date :	

Aim

To implement a Program in C for the evaluation of Suffix expression with single digit operands and operators: +, -, *, /, %, ^ using stack concepts.

Theory

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. As Postfix expression is without parenthesis and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle.

Evaluation rule of a Postfix Expression states:

While reading the expression from left to right, push the element in the stack if it is an operand.

Pop the two operands from the stack, if the element is an operator and then evaluate it.

Push back the result of the evaluation. Repeat it till the end of the expression.

Algorithm

Step 1: Create a stack to store operands (or values).

Step 2: Scan the given expression and do the following for every scanned element.

Step 2.1: If the element is a number, push it into the stack

Step 2.2: If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack

Step 3: When the expression is ended, the number in the stack

Program

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
int i, top = -1;
int op1, op2, res, s[20];
char postfix[90], symb;
void push (int item)
{
```

```
        top = top+1;
        s[top] = item;
}
int pop ()
{
        int item;
        item = s[top];
        top = top-1;
        return item;
}

void main()
{
        printf("\nEnter a valid postfix expression:\n");
        scanf("%s", postfix);
        for(i=0; postfix[i]!='\0'; i++)
        {
                symb = postfix[i];
                if(isdigit(symb))
                {
                        push(symb - '0');
                }
                else
                {
                        op2 = pop();
                        op1 = pop();
                        switch(symb)
                        {
                                case '+':       push(op1+op2);
                                                break;
                                case '-':       push(op1-op2);
                                                break;
                                case '*':       push(op1*op2);
                                                break;
                                case '/':       push(op1/op2);
                                                break;
                                case '%':       push(op1%op2);
                                                break;
                                case '$':
                                case '^':       push(pow(op1, op2));
                                                break;
                                default :   push(0);
                        }
                }
```

```
    }
    res = pop();
    printf("\n Result = %d", res);
}
```

Output

```
    Enter a valid postfix expression:
    623+-382/+*2$3+
    Result = 52

    Enter a valid postfix expression:
    42$3*3-84/11+/+
    Result = 46
```

Result

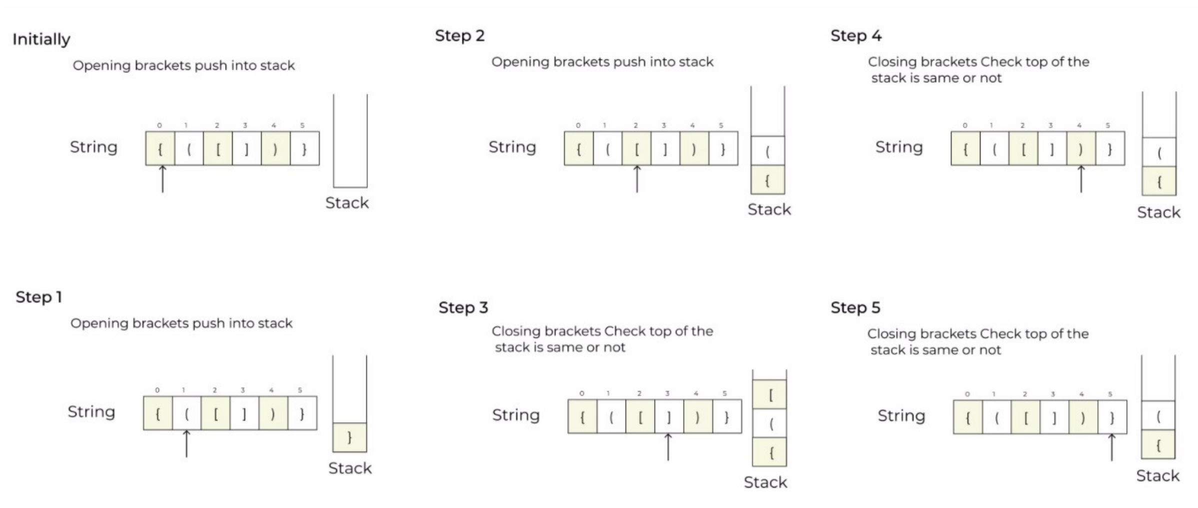
Thus, the C program for evaluation of Suffix expression with single digit operands and operators: +, -, *, /, %, ^ using stack concepts has been executed successfully and output is verified.

Ex.No: 03 b	Application of Stack and Queue ADT - Balancing Parenthesis
Date :	

Aim

To implement a Program in C for the balancing parenthesis **using** stack concepts.

Theory



Algorithm

Step 1: Declare a structure for character stack.

Step 2: Now traverse the expression string exp.

Step 2.1: If the current character is a starting bracket (**'(' or '{' or '['**) then push it to stack.

Step 2.2: If the current character is a closing bracket (**')' or '}' or ']'**) then pop from stack and if the popped character is the matching starting bracket then fine else brackets are not balanced.

Step 3: After complete traversal, if there is some starting bracket left in stack then "NOT BALANCED"

Program

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

```
#include<conio.h>
#define MAX 30
int top=-1;
int stack[MAX];
void push(char);
char pop();
int match(char a,char b);
int check(char []);
int main()
{
        char exp[MAX];
        int valid;
        clrscr();
        printf("Enter an algebraic expression : ");
        gets(exp);
        valid=check(exp);
        if(valid==1)
                printf("Valid expression\n");
        else
                printf("Invalid expression\n");
                getch();
                return 0;
}
int check(char exp[] )
{
        int i;
        char temp;
      for(i=0;i<strlen(exp);i++)
      {
```

```
        if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
            push(exp[i]);
        if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
            if(top==-1)    /*stack empty*/
            {
                printf("Right parentheses are more than left parentheses\n");
                return 0;
            }
            else
            {
                temp=pop();
                if(!match(temp, exp[i]))
                {
                    printf("Mismatched parentheses are : ");
                    printf("%c and %c\n",temp,exp[i]);
                    return 0;
                }
            }
    }
    if(top==-1) /*stack empty*/
    {
        printf("Balanced Parentheses\n");
        return 1;
    }
    else
    {
        printf("Left parentheses more than right parentheses\n");
        return 0;
    }
```

```
}/*End of main()*/
int match(char a,char b)
{
	if(a=='[' && b==']')
		return 1;
	if(a=='{' && b=='}')
		return 1;
	if(a=='(' && b==')')
		return 1;
	return 0;
}/*End of match()*/
void push(char item)
{
	if(top==(MAX-1))
	{
		printf("Stack Overflow\n");
		return;
	}
	top=top+1;
	stack[top]=item;
}/*End of push()*/
char pop()
{
	if(top==-1)
	{
		printf("Stack Underflow\n");
		exit(1);
	}
	return(stack[top--]);
```

```
}/*End of pop()*/
```

Output

```
Enter an algebraic expression: (((5+7)*6)/2)
Balanced Parentheses
Valid expression
```

Result

Thus, the C program for balancing parenthesis using stack concepts has been executed successfully and output is verified.

Ex.No: 03 c	Application of Stack and Queue ADT - Tower of Hanoi problem with n disks
Date :	

Aim

To implement a Program in C for the Tower of Hanoi problem with n disks using stack operation

Theory

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

Algorithm

Step 1: Move n-1 disks from source to aux

Step 2: Move nth disk from source to dest

Step 3: Move n-1 disks from aux to dest

Recursive algorithm for Tower of Hanoi can be driven as follows

```
START
Procedure towerOfHanoi (disk, source, dest, aux)
  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    towerOfHanoi (disk - 1, source, aux, dest)     // Step 1
    move disk from source to dest          // Step 2
    towerOfHanoi (disk - 1, aux, dest, source)     // Step 3
  END IF
 END Procedure
STOP
```

Program

```
#include<stdio.h>
#include<conio.h>
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
   if (n == 1)
   {
      printf("\n Move disk 1 from rod %c to rod %c", from_rod, to_rod);
      return;
   }
   towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
   printf("\n Move disk %d from rod %c to rod %c", n, from_rod, to_rod);
   towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}
 int main()
{
   int n = 4; // Number of disks
    clrscr();
   towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
   getch();
   return 0;
}
```

Output

Move disk 1 from rod A to rod B

Move disk 2 from rod A to rod C

Move disk 1 from rod B to rod C

Move disk 3 from rod A to rod B

Move disk 1 from rod C to rod A

Move disk 2 from rod C to rod B

Move disk 1 from rod A to rod B

Move disk 4 from rod A to rod C

Move disk 1 from rod B to rod C

Move disk 2 from rod B to rod A

Move disk 1 from rod C to rod A

Move disk 3 from rod B to rod C

Move disk 1 from rod A to rod B

Move disk 2 from rod A to rod C

Move disk 1 from rod B to rod C

Result

Thus, the C program for Tower of Hanoi problem with n disks using stack concepts has been executed successfully and output is verified.

Ex.No: 03 d	Application of Stack and Queue ADT – Producer and consumer Problem
Date :	

Aim

To implement a Program in C for the producer and consumer problem

Theory

The producer-consumer problem is an example of a multi-process synchronization problem.. There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

Algorithm

Step 1: Define the maximum buffer size.

Step 2: Enter the number of producers and consumers.

Step 3: The producer produces the job and put it in the buffer.

Step 4: The consumer takes the job from the buffer.

Step 5: If the buffer is full the producer goes to sleep.

Step 6: If the buffer is empty then consumer goes to sleep.

Program

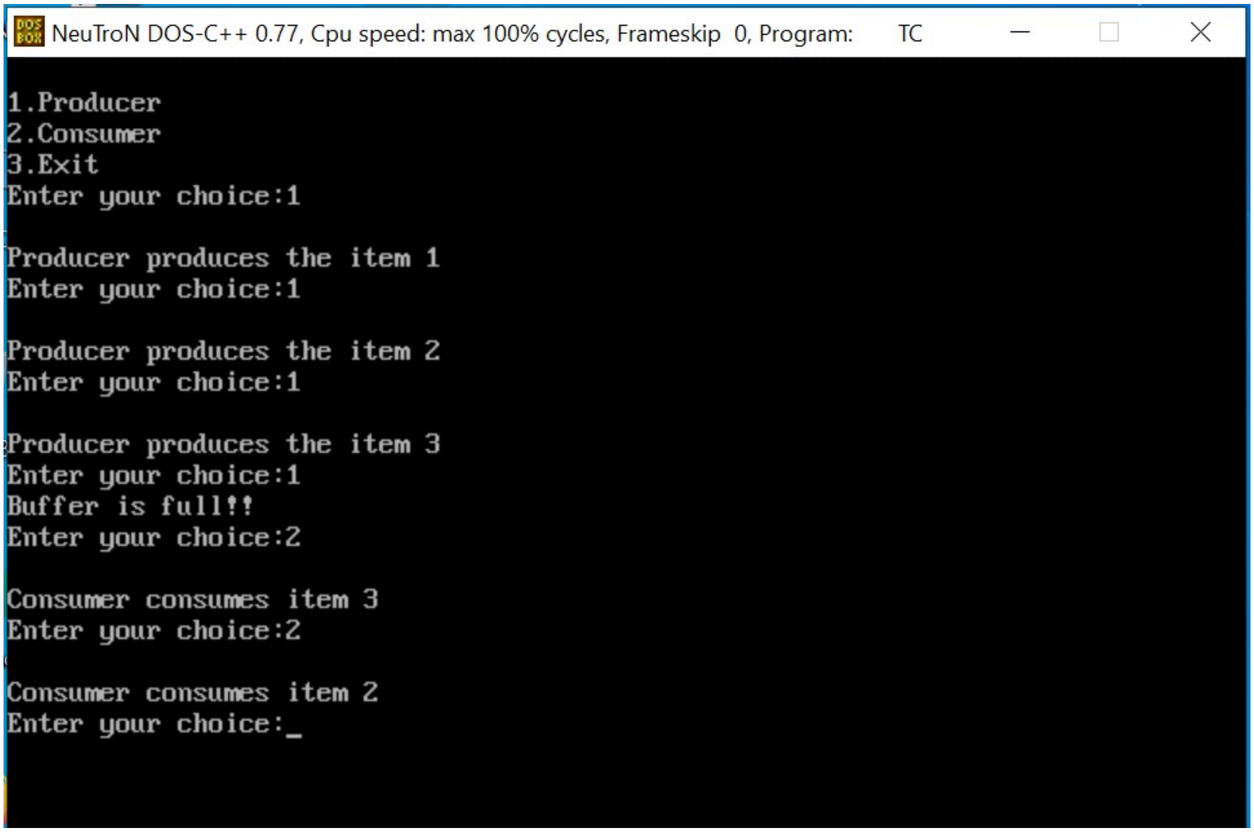
LIFO

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;
int main()
```

```
{
    int n;
    void producer();
    void consumer();
    int wait(int);
    int signal(int);
    printf("\n1.Producer\n2.Consumer\n3.Exit");
    while(1)
    {
        printf("\nEnter your choice:");
        scanf("%d",&n);
        switch(n)
        {
            case 1:    if((mutex==1)&&(empty!=0))
                            producer();
                        else
                            printf("Buffer is full!!");
                        break;
            case 2:    if((mutex==1)&&(full!=0))
                            consumer();
                        else
                            printf("Buffer is empty!!");
                        break;
            case 3:
                        exit(0);
                        break;
        }
    }
    return 0;
```

```
}
int wait(int s)
{
   return (--s);
}
int signal(int s)
{
   return(++s);
}
void producer()
{
   mutex=wait(mutex);
   full=signal(full);
   empty=wait(empty);
   x++;
   printf("\nProducer produces the item %d",x);
   mutex=signal(mutex);
}
void consumer()
{
   mutex=wait(mutex);
   full=wait(full);
   empty=signal(empty);
   printf("\nConsumer consumes item %d",x);
   x--;
   mutex=signal(mutex);
}
```

Output



Result

Thus, the C program for producer and consumer problem using stack concepts has been executed successfully and output is verified.

Ex.No: 04 a	Binary Search Tree – Insertion and Deletion Operation
Date :	

Aim

To construct the Binary Search Tree and perform the insertion and deletion operation using C

Theory

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

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

The properties that separate a binary search tree from a regular binary tree is

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

Algorithm

Step 1: start

Step 2: declare the necessary variables and function to be used in the program

Step 3: Get the choice of the function that the user need to perform

Step 4: Use. Malloc function for creation and insertion and findmin function for deletion

Step 5: perform the operation and print the result

Step 6: stop the process

Program

```
#include<stdio.h>

#include<conio.h>

#include<process.h>
```

```
#include<alloc.h>
struct tree
{
int data;
struct tree *lchild;
struct tree *rchild;
}*t,*temp;
int element;
void inorder(struct tree *);
struct tree * create(struct tree *, int);
struct tree * find(struct tree *, int);
struct tree * insert(struct tree *, int);
struct tree * del(struct tree *, int);
void main()
{
int ch;
do
{
printf("\n\t\t\tBINARY SEARCH TREE");
printf("\n\t\t\t****** ****** ****");
printf("\nMain Menu\n");
printf("\n1.Create\n2.Insert\n3.Delete\n4.Find\n5.Exit\n");
printf("\nEnter ur choice :");
scanf("%d",&ch);
switch(ch)
{
```

```
case 1:
printf("\nEnter the data:");
scanf("%d",&element);
t=create(t,element);
inorder(t);
break;
case2:
printf("\nEnter the data:");
scanf("%d",&element);
t=insert(t,element);
inorder(t);
break;
case3:
printf("\nEnter the data:");
scanf("%d",&element);
t=del(t,element);
inorder(t);
break;
case4:
printf("\nEnter the data:");
scanf("%d",&element);
temp=find(t,element);
if(temp->data==element)
printf("\nElement %d is at %d",element,temp);
else
printf("\nElement is not found");
```

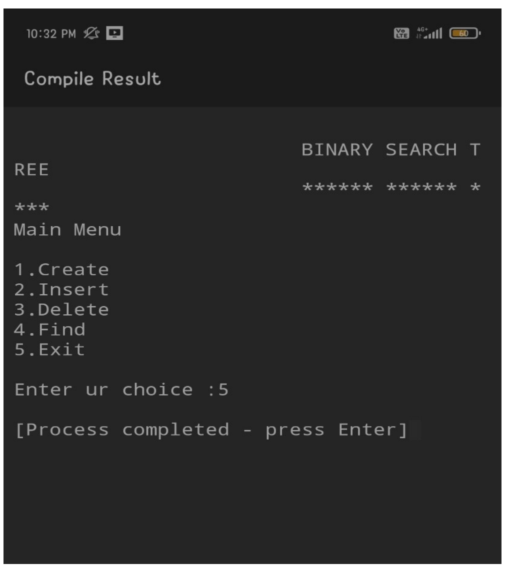
```
break;
case 5:
 exit(0);
}
}while(ch<=5);
}
struct tree * create(struct tree *t, int element)
{
t=(struct tree *)malloc(sizeof(structtree));
t->data=element;
t->lchild=NULL;
t->rchild=NULL;
return t;
}
struct tree * find(struct tree *t, int element)
{
if(t==NULL)
return NULL;
if(element<t->data)
return(find(t->lchild,element));
else
if(element>t->data)
return(find(t->rchild,element));
else
return t;
}
```

```
struct tree *insert(struct tree *t,int element)
{
if(t==NULL)
{
t=(struct tree *)malloc(sizeof(struct tree));
t->data=element;
t->lchild=NULL;
t->rchild=NULL;
return t;
}
else
{
if(element<t->data)
{
t->lchild=insert(t->lchild,element);
}
elseif(element>t->data)
{
t->rchild=insert(t->rchild,element);
}
elseif(element==t->data)
{
printf("element already present\n");
}
return t;
}
```

```
}
struct tree * del(struct tree *t, int element)
{
if(t==NULL)
printf("element not found\n");
elseif(element<t->data)
t->lchild=del(t->lchild,element);
elseif(element>t->data)
t->rchild=del(t->rchild,element);
elseif(t->lchild&&t->rchild)
{
temp=findmin(t->rchild);
t->data=temp->data;
t->rchild=del(t->rchild,t->data);
}
else
{
temp=t;
if(t->lchild==NULL)
t=t->rchild;
else
if(t->rchild==NULL)
t=t->lchild;
free(temp);
}
return t;
```

```
}
void inorder(struct tree *t)
{
if(t==NULL)
return;
else
{
inorder(t->lchild);
printf("\t%d",t->data);
inorder(t->rchild);
}
}
```

Output



Result

Thus, the C program to implement binary search tree was executed and the output was verified successfully.

Ex.No: 04 b	Binary Search Tree – Tree Traversal (pre-order, in-order, and post-order)
Date :	

Aim

To construct the Binary Search Tree and perform the tree traversal operation using C

Theory

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

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

The properties that separate a binary search tree from a regular binary tree is

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

Algorithm

Step 1: start the process

Step 2: define and declare the functions to be used in the program

Step 3: Get the choice from the user which is the type of traversal that the user needs to perform.

Step 4: For inorder traversal, the left child, root and the right child will be visited for preorder, root, left child and tight child will be visited and for post order traversal, left child right child and then the root will be visited.

Step 5: newnode can be inserted using malloc function

Step 6: Display the results

Step 7: stop

Program

```
#include<stdio.h>
#include<conio.h>
struct node
{
```

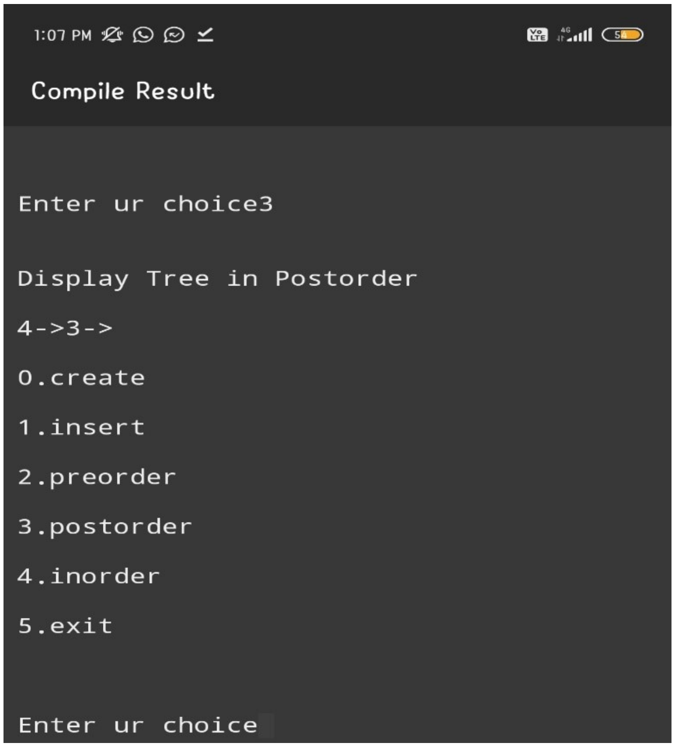
```
int data;
struct node *rlink;
struct node *llink;
}*tmp=NULL;
typedef struct node NODE;
NODE *create();
void preorder(NODE *);
void inorder(NODE *);
void postorder(NODE *);
void insert(NODE *);
void main()
{
int n,i,m;
clrscr();
do
{
printf(\n\n0.create\n\n1.insert \n\n2.preorder\n\n3.postorder\n\n4.inorder\n\n5.exit\n\n);
printf(\n\nEnter ur choice);
scanf("%d",&m);
switch(m)
{
case 0:
tmp=create();
break;
case 1:
insert(tmp);
break;
case 2:
printf(\n\nDisplay tree in Preorder traversal\n\n);
```

```
preorder(tmp);
break;
case 3:
printf(\n\nDisplay Tree in Postorder\n\n);
postorder(tmp);
break;
case 4:
printf(\n\nInorder\n\n);
inorder(tmp);
break;
case5:
exit(0);
}
}
while(n!=5);
getch();
}
void insert(NODE *root)
{
NODE *newnode;
if(root==NULL)
{
newnode=create();
root=newnode;
}
else
{
newnode=create();
while(1)
```

```
{
if(newnode->data<root->data)
{
if(root->llink==NULL)
{
root->llink=newnode;
Break;
}root=root->llink;
}if(newnode-ot->data>root->data)
{
if(root->rlink==NULL)
{
root->rlink=newnode;
break;
}
root=root->rlink;
}
}
}
}
NODE *create()
{
NODE *newnode;
int n;
newnode=(NODE*)malloc(sizeof(NODE));
printf(\n\nEnter the Data );
scanf(%d,&n);
newnode->data=n; newnode->llink=NULL;
newnode->rlink=NULL;
```

```
return(newnode);
}
void postorder(NODE *tmp)
{
if(tmp!=NULL)
{
postorder(tmp->llink);
postorder(tmp->rlink);
printf(%d->,tmp->data);
}}
void inorder(NODE *tmp)
{
If (tmp!=NULL)
{
inorder(tmp->llink);
printf(%d->,tmp->data);
inorder(tmp->rlink);
}}
void preorder(NODE *tmp)
{
if(tmp!=NULL){
printf(%d->,tmp->data);
preorder(tmp->llink); preorder(tmp->rlink);
}}
```

Output




```
Enter ur choice3

Display Tree in Postorder
4->3->
0.create
1.insert
2.preorder
3.postorder
4.inorder
5.exit

Enter ur choice
```

Result

Thus, the C program to perform tree traversal has been implemented and the output was verified successfully.

Ex.No: 04 c	Binary Search Tree – Finding maximum and minimum element in the tree
Date :	

Aim

To construct the Binary Search Tree and finding the maximum and minimum element in the tree using C

Theory

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

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

The properties that separate a binary search tree from a regular binary tree is

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

Algorithm

Approach for finding minimum and maximum element:

Step 1: Traverse the node from root to left recursively until left is NULL.

Step 2: The node whose left is NULL is the node with minimum value.

Step 3: Traverse the node from root to right recursively until right is NULL.

Step 4: The node whose right is NULL is the node with maximum value.

Program

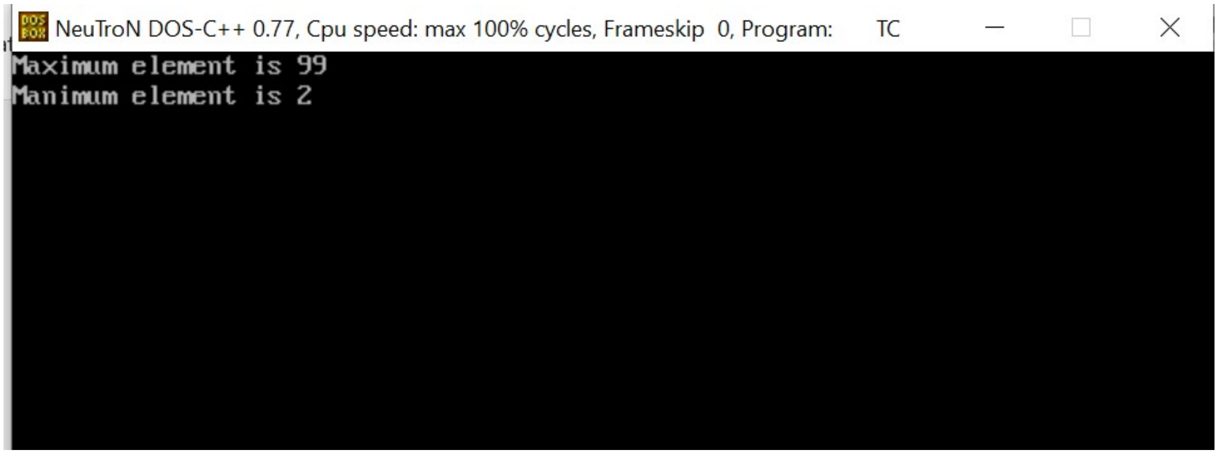
```
#include <limits.h>

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

struct Node {

   int data;
```

```
    struct Node *left, *right;
};
// A utility function to create a new node
struct Node* newNode(int data)
{
    struct Node* node
        = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}
// Returns maximum value in a given Binary Tree
int findMax(struct Node* root)
{
    // Base case
    int res,lres,rres;
    if (root == NULL)
            return INT_MIN;
    // Return maximum of 3 values:
    // 1) Root's data 2) Max in Left Subtree
    // 3) Max in right subtree
    res = root->data;
    lres = findMax(root->left);
    rres = findMax(root->right);
```

```
    if (lres > res)
            res = lres;
    if (rres > res)
            res = rres;
    return res;
}
// Returns minimum value in a given Binary Tree
int findMin(struct Node* root)
{
    // Base case
    int res,lres,rres;
    if (root == NULL)
      return INT_MAX;
    // Return minimum of 3 values:
    // 1) Root's data 2) Max in Left Subtree
    // 3) Max in right subtree
    res = root->data;
    lres = findMin(root->left);
    rres = findMin(root->right);
    if (lres < res)
      res = lres;
    if (rres < res)
      res = rres;
    return res;
```

```
}

// Driver code
int main(void)
{
    struct Node* NewRoot = NULL;
    struct Node* root = newNode(2);
    clrscr();
    root->left = newNode(77);
    root->right = newNode(25);
    root->left->right = newNode(66);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(11);
    root->right->right = newNode(99);
    root->right->right->left = newNode(4);
    // Function call
    printf("Maximum element is %d \n", findMax(root));
    printf("Manimum element is %d \n", findMin(root));
    getch();
    return 0;
}
```

Output



Result

Thus, the C program to find the maximum and minimum element in the tree has been implemented and the output was verified successfully.

Ex.No: 05 a	Construct Max heap to perform insertion and delete operations
Date :	

Aim

To Implement a program to construct max heap to perform insertion and delete operations.

Theory

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

Algorithm

Step 1: Initialize the array

Step 2: Create a complete binary tree from the array

Step 3: Start from the first index of non-leaf node whose index is given by n/2 - 1.

Step 4: Set current element i as largest.

Step 5: The index of left child is given by 2i + 1 and the right child is given by 2i + 2.

Step 6: If left Child is greater than current Element (i.e. element at i^{th} index), set left Child Index as largest.

Step 7: If right Child is greater than element in largest, set right Child Index as largest.

Step 8: Swap largest with current Element

Step 9: Repeat steps 3-7 until the subtrees are also heapified.

Heapify(array, size, i)

set i as largest

leftChild = 2i + 1

rightChild = 2i + 2

if leftChild > array[largest]

set leftChildIndex as largest
if rightChild > array[largest]
set rightChildIndex as largest
swap array[i] and array[largest]

To create a Max-Heap:

MaxHeap(array, size)
loop from the first index of non-leaf node down to zero
call heapify

Insert an element in Max Heap

If there is no node,
create a newNode.
else (a node is already present)
insert the newNode at the end (last node from left to right.)
heapify the array

Delete an element in Min Heap

If nodeToBeDeleted is the leafNode
remove the node
Else swap nodeToBeDeleted with the lastLeafNode
remove noteToBeDeleted
heapify the array

Program

```
#include <stdio.h>

int array[100], n;

main()

{

    int choice, num;

    n = 0;/*Represents number of nodes in the heap*/

    while(1)
```

```
{
   printf("1.Insert the element \n");
   printf("2.Delete the element \n");
   printf("3.Display all elements \n");
   printf("4.Quit \n");
   printf("Enter your choice : ");
   scanf("%d", &choice);
   switch(choice)
   {
   case 1:
      printf("Enter the element to be inserted to the list : ");
      scanf("%d", &num);
      insert(num, n);
      n = n + 1;
      break;
   case 2:
      printf("Enter the elements to be deleted from the list: ");
      scanf("%d", &num);
      delete(num);
      break;
   case 3:
      display();
      break;
   case 4:
```

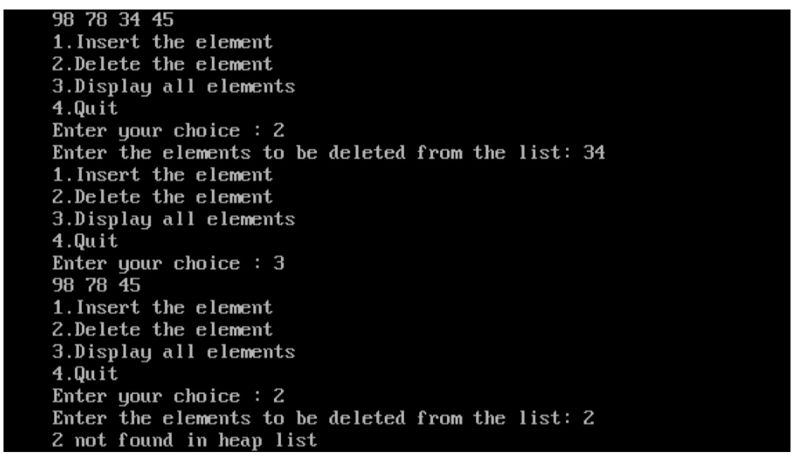
```
            exit(0);
        default:
            printf("Invalid choice \n");
    }/*End  of switch */
}/*End of while */
}/*End of main()*/
 display()
{
    int i;
    if (n == 0)
    {
        printf("Heap is empty \n");
        return;
    }
    for (i = 0; i < n; i++)
        printf("%d ", array[i]);
    printf("\n");
}/*End of display()*/
 insert(int num, int location)
{
    int parentnode;
    while (location > 0)
    {
        parentnode =(location - 1)/2;
```

```
        if (num <= array[parentnode])
        {
            array[location] = num;
            return;
        }
        array[location] = array[parentnode];
        location = parentnode;
    }/*End of while*/
    array[0] = num; /*assign number to the root node */
}/*End of insert()*/
 delete(int num)
{
    int left, right, i, temp, parentnode;

    for (i = 0; i < num; i++) {
        if (num == array[i])
            break;
    }
    if (num != array[i])
    {
        printf("%d not found in heap list\n", num);
        return;
    }
    array[i] = array[n - 1];
```

```
n = n - 1;
parentnode =(i - 1) / 2; /*find parentnode of node i */
if (array[i] > array[parentnode])
{
   insert(array[i], i);
   return;
}
left = 2 * i + 1; /*left child of i*/
right = 2 * i + 2; /* right child of i*/
while (right < n)
{
   if (array[i] >= array[left] && array[i] >= array[right])
      return;
   if (array[right] <= array[left])
   {
      temp = array[i];
      array[i] = array[left];
      array[left] = temp;
      i = left;
   }
   else
   {
      temp = array[i];
      array[i] = array[right];
```

```
        array[right] = temp;

        i = right;

      }

      left = 2 * i + 1;

      right = 2 * i + 2;

  }/*End of while*/

  if (left == n - 1 && array[i])    {

      temp = array[i];

      array[i] = array[left];

      array[left] = temp;

  }}
```

Output



Result

Thus, the C program to construct max heap and its operations has been implemented successfully and the output is verified

Ex.No: 05 b	Construct Min heap to perform insertion and delete operations
Date :	

Aim

To Implement a program to construct min heap to perform insertion and delete operations.

Theory

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.

Algorithm

Step 1: Check if the array has more than two elements. If it does not, remove the element in the first index. If it does, continue with the steps below.

Step 2: Assign the last value to the first index.

Step 3: Remove the last value from the array.

Step 4: Check if the array has three elements remaining. If it is true, check if the first element is greater than the second element. Swap them if the condition is satisfied. If there are more than three elements, continue with the steps below.

Step 5: Define the index of the parent node, left node, and right node.

Step 6: Loop through the array that have both the left child value and right child value. Where the parent value is greater than the left child value or right child value, swap them. If the left node value is greater than the right node value, swap them as well.

Step 7: If there is no right node value but the parent node is greater than the left node value, swap the values.

Program

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
struct Heap{
```

```
    int *arr;
    int count;
    int capacity;
    int heap_type; // for min heap , 1 for max heap
};
typedef struct Heap Heap;
Heap *CreateHeap(int capacity,int heap_type);
void insert(Heap *h, int key);
void print(Heap *h);
void heapify_bottom_top(Heap *h,int index);
void heapify_top_bottom(Heap *h, int parent_node);
int PopMin(Heap *h);
int main(){
    int i;
    Heap *heap = CreateHeap(20, 0); //Min Heap
    clrscr();
    if( heap == NULL ){
        printf("__Memory Issue____\n");
        return -1;
    }
    for(i =9;i>0;i--)
        insert(heap, i);

    print(heap);
    for(i=9;i>=0;i--){
        printf(" Pop Minima : %d\n", PopMin(heap));
        print(heap);
    }
    getch();
```

```
    return 0;
}
Heap *CreateHeap(int capacity,int heap_type){
    Heap *h = (Heap * ) malloc(sizeof(Heap)); //one is number of heap

    //check if memory allocation is fails
    if(h == NULL){
        printf("Memory Error!");
            return;
    }
    h->heap_type = heap_type;
    h->count=0;
    h->capacity = capacity;
    h->arr = (int *) malloc(capacity*sizeof(int)); //size in bytes
    //check if allocation succeed
    if ( h->arr == NULL){
        printf("Memory Error!");
        return;
    }
    return h;
}
void insert(Heap *h, int key){
    if( h->count < h->capacity){
        h->arr[h->count] = key;
        heapify_bottom_top(h, h->count);
        h->count++;
    }
}
void heapify_bottom_top(Heap *h,int index){
```

```
    int temp;
    int parent_node = (index-1)/2;
    if(h->arr[parent_node] > h->arr[index]){
        //swap and recursive call
        temp = h->arr[parent_node];
        h->arr[parent_node] = h->arr[index];
        h->arr[index] = temp;
        heapify_bottom_top(h,parent_node);
    }
}
void heapify_top_bottom(Heap *h, int parent_node){
    int left = parent_node*2+1;
    int right = parent_node*2+2;
    int min;
    int temp;
    if(left >= h->count || left <0)
        left = -1;
    if(right >= h->count || right <0)
        right = -1;
    if(left != -1 && h->arr[left] < h->arr[parent_node])
        min=left;
    else
        min =parent_node;
    if(right != -1 && h->arr[right] < h->arr[min])
        min = right;
    if(min != parent_node){
        temp = h->arr[min];
        h->arr[min] = h->arr[parent_node];
        h->arr[parent_node] = temp;
```

```
        // recursive  call
        heapify_top_bottom(h, min);
    }
}
int PopMin(Heap *h){
    int pop;
    if(h->count==0){
        printf("\n__Heap is Empty__\n");
        return -1;
    }
    // replace first node by last and delete last
    pop = h->arr[0];
    h->arr[0] = h->arr[h->count-1];
    h->count--;
    heapify_top_bottom(h, 0);
    return pop;
}
void print(Heap *h){
    int i;
    printf("____________Print Heap_____________\n");
    for(i=0;i< h->count;i++){
        printf("-> %d ",h->arr[i]);
    }
    printf("->__/\\__\n");
}
```

Output

```
_____________Print Heap______________
      -> 1 -> 2 -> 4 -> 3 -> 7 -> 8 -> 5 -> 9 -> 6 ->__/\__
       Pop Minima : 1
      _____________Print Heap______________
      -> 2 -> 3 -> 4 -> 6 -> 7 -> 8 -> 5 -> 9 ->__/\__
       Pop Minima : 2
      _____________Print Heap______________
      -> 3 -> 6 -> 4 -> 9 -> 7 -> 8 -> 5 ->__/\__
       Pop Minima : 3
      _____________Print Heap______________
      -> 4 -> 6 -> 5 -> 9 -> 7 -> 8 ->__/\__
       Pop Minima : 4
      _____________Print Heap______________
      -> 5 -> 6 -> 8 -> 9 -> 7 ->__/\__
       Pop Minima : 5
      _____________Print Heap______________
      -> 6 -> 7 -> 8 -> 9 ->__/\__
       Pop Minima : 6
      _____________Print Heap______________
      -> 7 -> 9 -> 8 ->__/\__
       Pop Minima : 7
      _____________Print Heap______________
      -> 8 -> 9 ->__/\__
       Pop Minima : 8
      _____________Print Heap______________
      -> 9 ->__/\__
       Pop Minima : 9
      _____________Print Heap______________
      ->__/\__
      __Heap is Empty__
       Pop Minima : -1
      _____________Print Heap______________
      ->__/\__
```

Result

Thus, the C program to construct min heap and its operations has been implemented successfully and the output is verified

Ex.No: 06	Dijkstra's algorithm
Date :	

Aim

To implement a C program to find shortest path in graph using Dijkstra's algorithm

Theory

Dijkstra algorithm is also called single source shortest path algorithm. It is based on greedy technique. The algorithm maintains a list visited [ ] of vertices, whose shortest distance from the source is already known.

If visited [1], equals 1, then the shortest distance of vertex i is already known. Initially, visited[i] is marked as, for source vertex.

At each step, we mark visited[v] as 1. Vertex v is a vertex at shortest distance from the source vertex.

Time Complexity

The program contains two nested loops each of which has a complexity of O(n). n is number of vertices. So the complexity of algorithm is $O(n^2)$.

Algorithm

Step 1: Create cost matrix C[ ][ ] from adjacency matrix adj[ ][ ]. C[i][j] is the cost of going from vertex i to vertex j. If there is no edge between vertices i and j then C[i][j] is infinity.

Step 2: Array visited[ ] is initialized to zero.

```
for(i=0;i<n;i++)
        visited[i]=0;
```

Step 3: If the vertex 0 is the source vertex then visited[0] is marked as 1.

Step 4: Create the distance matrix, by storing the cost of vertices from vertex no. 0 to n-1 from the source vertex 0.

```
for(i=1;i<n;i++)
        distance[i]=cost[0][i];
```

Initially, distance of source vertex is taken as 0. i.e. distance[0]=0;

Step 5: Execute the loop - for(i=1;i<n;i++)

Step 5.1: Choose a vertex w, such that distance[w] is minimum and visited[w] is 0. Mark visited[w] as 1.

Step 5.2: Recalculate the shortest distance of remaining vertices from the source.

Step 5.3: Only, the vertices not marked as 1 in array visited[ ] should be considered for recalculation of distance. i.e. for each vertex v

```
        if(visited[v]==0)
                distance[v]=min(distance[v],
                distance[w]+cost[w][v])
```

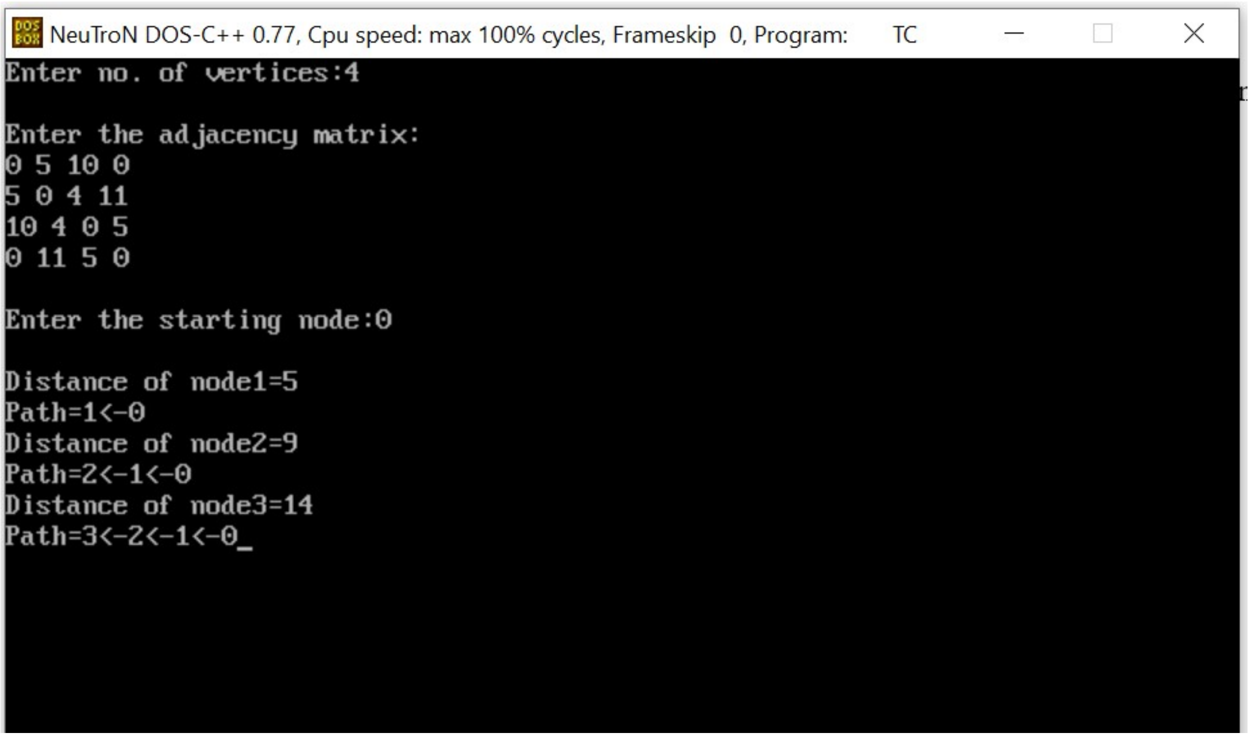
Program

```
#include<stdio.h>
#include<conio.h>
#define INFINITY 9999
#define MAX 10
void dijkstra(int G[MAX][MAX],int n,int startnode);
int main()
{
int G[MAX][MAX],i,j,n,u;
printf("Enter no. of vertices:");
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&G[i][j]);
printf("\nEnter the starting node:");
scanf("%d",&u);
dijkstra(G,n,u);
return 0;
}
```

```
void dijkstra(int G[MAX][MAX],int n,int startnode)
{
int cost[MAX][MAX],distance[MAX],pred[MAX];
int visited[MAX],count,mindistance,nextnode,i,j;
for(i=0;i<n;i++)
for(j=0;j<n;j++)
if(G[i][j]==0)
cost[i][j]=INFINITY;
else
cost[i][j]=G[i][j];
for(i=0;i<n;i++)
{
distance[i]=cost[startnode][i];
pred[i]=startnode;
visited[i]=0;
}
distance[startnode]=0;
visited[startnode]=1;
count=1;
while(count<n-1)
{
mindistance=INFINITY;
for(i=0;i<n;i++)
if(distance[i]<mindistance&&!visited[i])
{
mindistance=distance[i];
nextnode=i;
}
visited[nextnode]=1;
```

```
for(i=0;i<n;i++)
if(!visited[i])
if(mindistance+cost[nextnode][i]<distance[i])
{
distance[i]=mindistance+cost[nextnode][i];
pred[i]=nextnode;
}
count++;
}
for(i=0;i<n;i++)
if(i!=startnode)
{
printf("\nDistance of node%d=%d",i,distance[i]);
printf("\nPath=%d",i);
j=i;
do
{
j=pred[j];
printf("<-%d",j);
}while(j!=startnode);
}
}
```

Output



Result

Thus, the C program to find shortest path in graph using Dijkstra's algorithm has been implemented successfully and the output is verified

Ex.No: 07	Prim's algorithm
Date :	

Aim

To implement a C program to find minimum cost spanning tree in graph using prim's algorithm

Theory

Spanning tree - A spanning tree is the subgraph of an undirected connected graph.

Minimum Spanning tree - Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree from a graph. Prim's algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

Algorithm

Step 1: Create a set mstSet that keeps track of vertices already included in MST.

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

Step 3: While mstSet doesn’t include all vertices

Step 3.1: Pick a vertex u which is not there in mstSet and has a minimum key value.

Step 3.2: Include u to mstSet.

Step 3.3: 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 the weight of edge u-v is less than the previous key value of v, update the key value as the weight of u-v

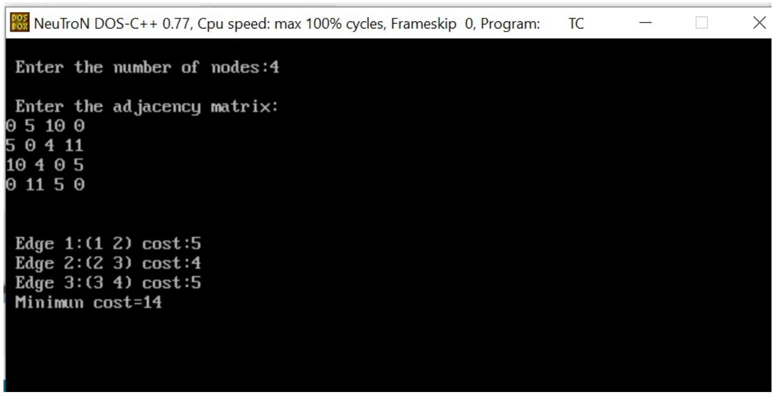
Program

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

```
int a,b,u,v,n,i,j,ne=1;
int visited[10]= {
  0
}
,min,mincost=0,cost[10][10];
void main() {
  clrscr();
  printf("\n Enter the number of nodes:");
  scanf("%d",&n);
  printf("\n Enter the adjacency matrix:\n");
  for (i=1;i<=n;i++)
    for (j=1;j<=n;j++) {
          scanf("%d",&cost[i][j]);
          if(cost[i][j]==0)
              cost[i][j]=999;
  }
  visited[1]=1;
  printf("\n");
  while(ne<n) {
          for (i=1,min=999;i<=n;i++)
             for (j=1;j<=n;j++)
              if(cost[i][j]<min)
               if(visited[i]!=0) {
                    min=cost[i][j];
                    a=u=i;
                    b=v=j;
          }
          if(visited[u]==0 || visited[v]==0) {
                    printf("\n Edge %d:(%d %d) cost:%d",ne++,a,b,min);
                    mincost+=min;
                    visited[b]=1;
          }
          cost[a][b]=cost[b][a]=999;
```

```
  }
  printf("\n Minimun cost=%d",mincost);
  getch();
}
```

Output



Result

Thus, the C program to find minimum cost spanning tree in graph using prim's algorithm has been implemented successfully and the output is verified

Ex.No: 08	Hashing Technique
Date :	

Aim

To implement a C program for hashing technique.

Theory

The Hash table data structure stores elements in key-value pairs where

- Key- unique integer that is used for indexing the values
- Value - data that are associated with keys.

In a hash table, a new index is processed using the keys. And, the element corresponding to that key is stored in the index. This process is called **hashing**.

Let k be a key and h(x) be a hash function.

Here, h(k) will give us a new index to store the element linked with k.

Algorithm

Step 1: Define a key and Initialize an array to store all key-value pairs.

Step 2: Perform the Insert Operation Insert (Key, Value): Insert the pair {Key, Value} in the Hash Table

Step 3: Search the data in the hash table. Find (Key): Finds the value of the Key in the Hash Table.

Step 4: Delete the data in the hash table. Delete (Key): Deletes the Key from the Hash Table

Program

```
// Implementing hash table in C
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct set
{
  int key;
  int data;
```

```
};
struct set *array;
int capacity = 10;
int size = 0;
int hashFunction(int key)
{
 return (key % capacity);
}
int checkPrime(int n)
{
 int i;
 if (n == 1 || n == 0)
 {
 return 0;
 }
 for (i = 2; i < n / 2; i++)
 {
 if (n % i == 0)
 {
  return 0;
 }
 }
 return 1;
}
int getPrime(int n)
{
 if (n % 2 == 0)
 {
 n++;
```

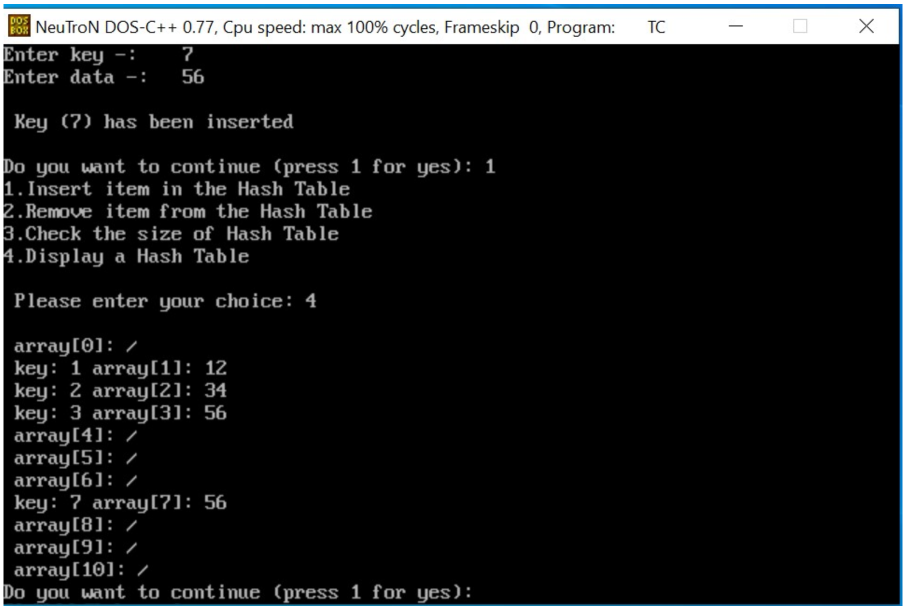
```
 }
 while (!checkPrime(n))
 {
 n += 2;
 }
 return n;
}
void init_array()
{
 int i;
 capacity = getPrime(capacity);
 array = (struct set *)malloc(capacity * sizeof(struct set));
 for (i = 0; i < capacity; i++)
 {
 array[i].key = 0;
 array[i].data = 0;
 }
}
void insert(int key, int data)
{
 int index = hashFunction(key);
 if (array[index].data == 0)
 {
 array[index].key = key;
 array[index].data = data;
 size++;
 printf("\n Key (%d) has been inserted \n", key);
 }
 else if (array[index].key == key)
```

```
 {
 array[index].data = data;
 }
 else
 {
 printf("\n Collision occured  \n");
 }
}
void remove_element(int key)
{
 int index = hashFunction(key);
 if (array[index].data == 0)
 {
 printf("\n This key does not exist \n");
 }
 else
 {
 array[index].key = 0;
 array[index].data = 0;
 size--;
 printf("\n Key (%d) has been removed \n", key);
 }
}
void display()
{
 int i;
 for (i = 0; i < capacity; i++)
 {
 if (array[i].data == 0)
```

```
 {
  printf("\n array[%d]: / ", i);
 }
 else
 {
  printf("\n key: %d array[%d]: %d \t", array[i].key, i, array[i].data);
 }
 }
}
int size_of_hashtable()
{
 return size;
}
int main()
{
 int choice, key, data, n;
 int c = 0;
 clrscr();
 init_array();
 do
 {
 printf("1.Insert item in the Hash Table"
   "\n2.Remove item from the Hash Table"
   "\n3.Check the size of Hash Table"
   "\n4.Display a Hash Table"
   "\n\n Please enter your choice: ");
 scanf("%d", &choice);
 switch (choice)
 {
```

```
    case 1:
      printf("Enter key -:\t");
      scanf("%d", &key);
      printf("Enter data -:\t");
      scanf("%d", &data);
      insert(key, data);
      break;
    case 2:
      printf("Enter the key to delete-:");
      scanf("%d", &key);
      remove_element(key);
      break;
    case 3:
      n = size_of_hashtable();
      printf("Size of Hash Table is-:%d\n", n);
      break;
    case 4:
      display();
      break;
    default:
      printf("Invalid Input\n");
    }
    printf("\nDo you want to continue (press 1 for yes): ");
    scanf("%d", &c);
    } while (c == 1);
    getch();
  }
```

Output



Result

Thus, the program to perform hashing operation using C has been implemented successfully and the output is verified

Ex.No: 09 a	Linear Search
Date :	

Aim

To write a C program to search an element in the array using linear search

Theory

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

Linear search is also called as sequential search algorithm. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL. It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted. The worst-case time complexity of linear search is O(n).

Algorithm

Step 1: First, we have to traverse the array elements using a for loop.

Step 2: In each iteration of for loop, compare the search element with the current array element,

Step 2.1: If the element matches, then return the index of the corresponding array element.

Step 2.2: If the element does not match, then move to the next element.

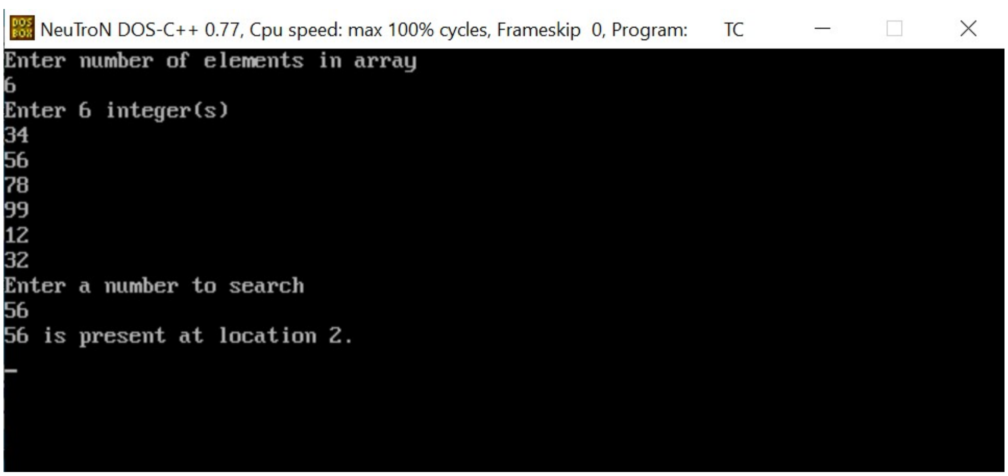
Step 3: If there is no match or the search element is not present in the given array, return -1.

Program

```
#include <stdio.h>
int main()
{
  int array[100], search, c, n;
  printf("Enter number of elements in array\n");
  scanf("%d", &n);
  printf("Enter %d integer(s)\n", n);
```

```
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter a number to search\n");
  scanf("%d", &search);
  for (c = 0; c < n; c++)
  {
    if (array[c] == search)    /* If required element is found */
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d isn't present in the array.\n", search);
  return 0;
}
```

Output



Result

Thus, the program to search an element in the array using linear search has been implemented successfully and the output is verified

Ex.No: 09 b	Binary Search
Date :	

Aim

To write a C program to search an element in the array using Binary search

Theory

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned.

Algorithm

Step 1: Begin with the mid element of the whole array as a search key.

Step 2: If the value of the search key is equal to the item then return an index of the search key.

Step 3: Or if the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.

Step 4: Otherwise, narrow it to the upper half.

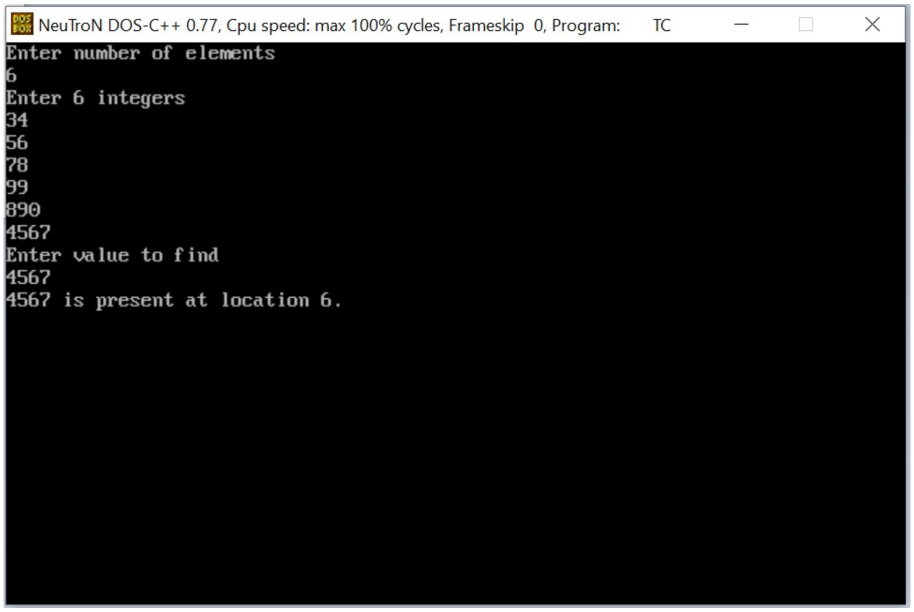
Step 5: Repeatedly check from the second point until the value is found or the interval is empty.

Program

```
#include <stdio.h>
#include <conio.h>
int binarySearch(int [], int, int, int);
int main()
{
  int c, first, last, n, search, array[100], index;
  clrscr();
```

```
  printf("Enter number of elements\n");
  scanf("%d", &n);
  printf("Enter %d integers\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter value to find\n");
  scanf("%d", &search);
  first = 0;
  last = n - 1;
  index = binarySearch(array, first, last, search);
   if (index == -1)
    printf("Not found! %d isn't present in the list.\n", search);
  else
    printf("%d is present at location %d.\n", search, index + 1);
   getch();
  return 0;
}
int binarySearch(int a[], int s, int e, int f) {
  int m;
   if (s > e) // Not found
     return -1;
  m = (s + e)/2;
  if (a[m] == f)  // element found
    return m;
  else if (f > a[m])
    return binarySearch(a, m+1, e, f);
  else
    return binarySearch(a, s, m-1, f);
}
```

Output



Result

Thus, the program to search an element in the array using binary search has been implemented successfully and the output is verified.

Ex.No: 10 a	Insertion Sort
Date :	

Aim

To write a C program to arrange a list of integers in ascending order using Insertion sort Algorithm

Theory

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Characteristics of Insertion Sort

This algorithm is one of the simplest algorithms with simple implementation

Basically, Insertion sort is efficient for small data values

Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

Algorithm

Step 1: If the element is the first one, it is already sorted.

Step 2: Move to next element

Step 3: Compare the current element with all elements in the sorted array

Step 4: If the element in the sorted array is smaller than the current element, iterate to the next element. Otherwise, shift all the greater element in the array by one position towards the right

Step 5: Insert the value at the correct position

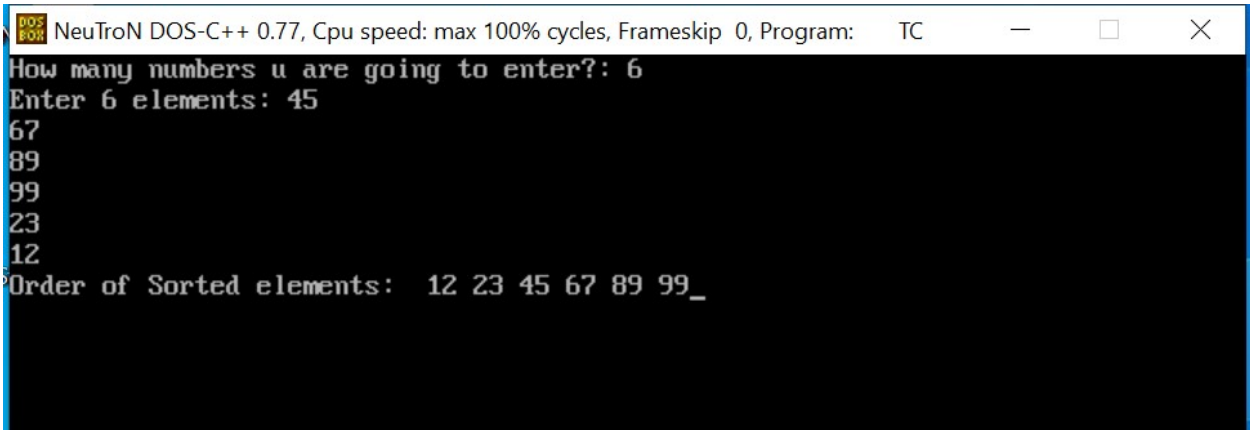
Step 6: Repeat until the complete list is sorted

Program

```
#include<stdio.h>
#include<conio.h>
int main(){
  int i, j, count, temp, number[25];
  clrscr();
```

```
   printf("How many numbers u are going to enter?: ");
   scanf("%d",&count);
   printf("Enter %d elements: ", count);
   for(i=0;i<count;i++)
     scanf("%d",&number[i]);
   // Implementation of insertion sort algorithm
   for(i=1;i<count;i++){
     temp=number[i];
     j=i-1;
     while((temp<number[j])&&(j>=0)){
          number[j+1]=number[j];
          j=j-1;
     }
     number[j+1]=temp;
   }
   printf("Order of Sorted elements: ");
   for(i=0;i<count;i++)
     printf(" %d",number[i]);
   getch();
   return 0;
}
```

Output



Result

Thus, the program to arrange a list of integers in ascending order using Insertion sort Algorithm has been implemented successfully and the output is verified.

Ex.No: 10 b	Bubble Sort
Date :	

Aim

To write a C program to arrange a list of integers in ascending order using Bubble sort Algorithm

Theory

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Algorithm

Step 1: Starts from the first index: arr[0] and compares the first and second element: arr[0] and arr[1]

Step 2: If arr[0] is greater than arr[1], they are swapped

Step 3: Similarly, if arr[1] is greater than arr[2], they are swapped

Step 4: The above process continues until the last element arr[n-1]

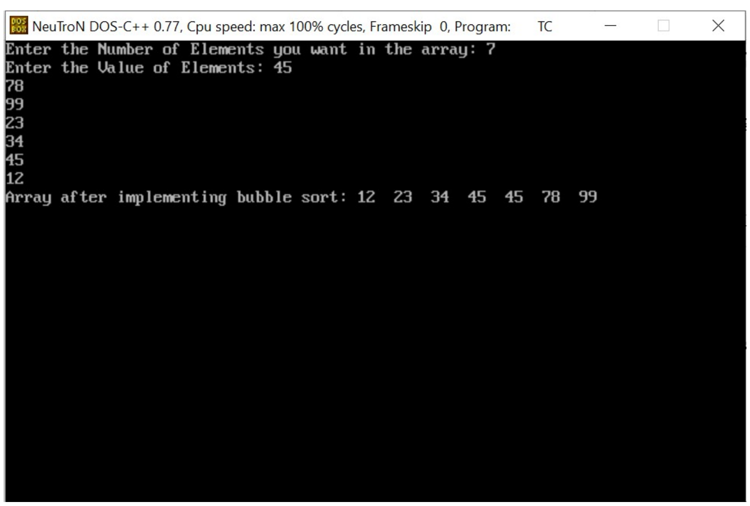
Step 5: All four steps are repeated for each iteration. Upon completing each iteration, the largest unsorted element is moved to the end of the array. Finally, the program ends when no elements require swapping, giving us the array in ascending order.

Program

```
#include <stdio.h>
#include <conio.h>
void bubbleSortExample(int arr[], int num){
   int x, y, temp;
   for(x = 0; x < num - 1; x++){
      for(y = 0; y < num - x - 1; y++){
         if(arr[y] > arr[y + 1]){
            temp = arr[y];
            arr[y] = arr[y + 1];
```

```
            arr[y + 1] = temp;          }       }    }}
int main(){
   int arr[50], n, x;
   clrscr();
   printf("Please Enter the Number of Elements you want in the array: ");
   scanf("%d", &n);
   printf("Please Enter the Value of Elements: ");
   for(x = 0; x < n; x++)
      scanf("%d", &arr[x]);
   bubbleSortExample(arr, n);
   printf("Array after implementing bubble sort: ");
   for(x = 0; x < n; x++){
      printf("%d  ", arr[x]);    }
   getch();
   return 0;}
```

Output



Result

Thus, the program to arrange a list of integers in ascending order using Bubble sort Algorithm has been implemented successfully and the output is verified.

Ex.No: 11	Heap Sort
Date :	

Aim

To write a C program to arrange a list of integers in ascending order using Heap sort Algorithm

Theory

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified. Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees.

Algorithm

Step 1: Get the array of data from the user for perform the sorting Algorithm

Step2: Once the array is received, to create a heap for sorting the elements in ascending order.

Step 3: Now out of the heap, a max heap is needed to be created. Remember, the value of the root node/parent node is always greater than or equal to the value of the children nodes.

Step 4: After building the tree, the above condition must be checked. If the value of the child node is greater than the child node, need to swap the values and repeat the process until it satisfies the max-heap property.

Step 5: Once all the conditions are satisfied, the root node needs to be swapped with the last node.

Step 6: As it is now sorted, can remove the last node from our heap.

Step 7: The previous three steps (Steps 4,5, & 6) need to be repeated until there is only one element left in the heap.

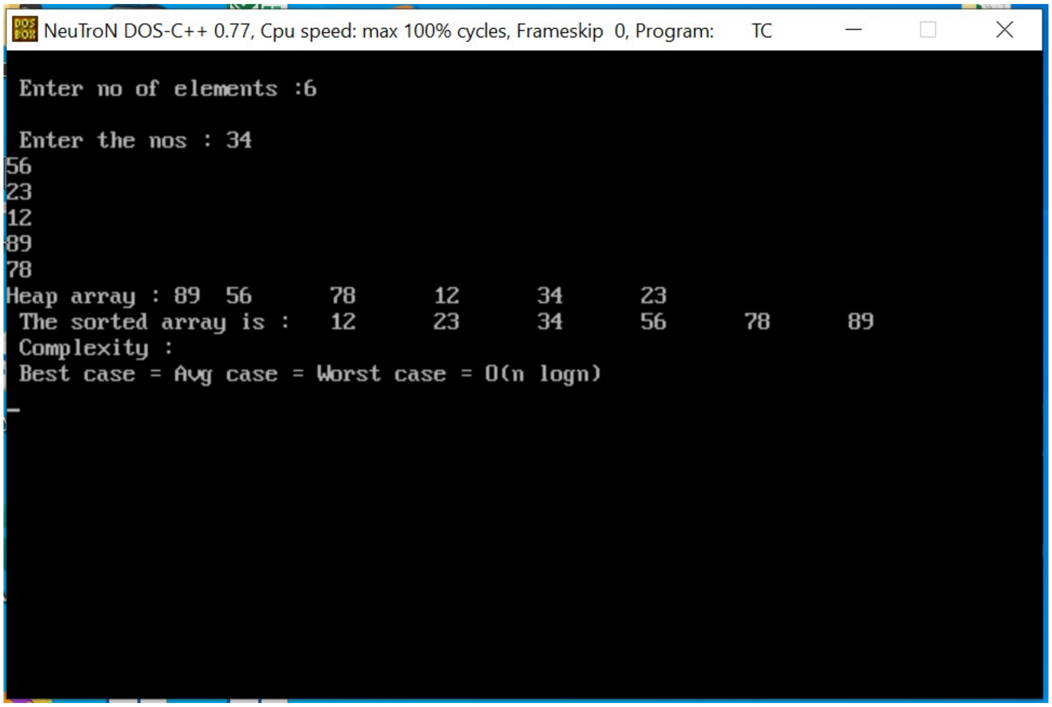
Program

```
#include <stdio.h>
#include<conio.h>
void main()
```

```
{
   int heap[10], no, i, j, c, root, temp;
   clrscr():
   printf("\n Enter no of elements :");
   scanf("%d", &no);
   printf("\n Enter the nos : ");
   for (i = 0; i < no; i++)
     scanf("%d", &heap[i]);
   for (i = 1; i < no; i++)
   {
      c = i;
      do
      {
         root = (c - 1) / 2;
         if (heap[root] < heap[c])   /* to create MAX heap array */
         {
            temp = heap[root];
            heap[root] = heap[c];
            heap[c] = temp;
         }
         c = root;
      } while (c != 0);
   }
    printf("Heap array : ");
   for (i = 0; i < no; i++)
      printf("%d\t ", heap[i]);
   for (j = no - 1; j >= 0; j--)
   {
      temp = heap[0];
```

```
        heap[0] = heap[j    /* swap max element with rightmost leaf element */
        heap[j] = temp;
        root = 0;
        do
        {
            c = 2 * root + 1;    /* left node of root element */
            if ((heap[c] < heap[c + 1]) && c < j-1)
                c++;
            if (heap[root]<heap[c] && c<j)    /* again rearrange to max heap array */
            {
                temp = heap[root];
                heap[root] = heap[c];
                heap[c] = temp;
            }
            root = c;
        } while (c < j);
    }
    printf("\n The sorted array is : ");
    for (i = 0; i < no; i++)
      printf("\t %d", heap[i]);
    printf("\n Complexity : \n Best case = Avg case = Worst case = O(n logn) \n");
getch();
}
```

Output



Result

Thus, the program to arrange a list of integers in ascending order using heap sort Algorithm has been implemented successfully and the output is verified.

Ex.No: 12 a	Quick Sort
Date :	

Aim

To write a C program to arrange a list of integers in ascending order using Quick sort Algorithm

Theory

Quicksort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Algorithm

Step 1: Pick an element from an array, call it as pivot element.

Step 2: Divide an unsorted array element into two arrays.

Step 3: If the value less than pivot element come under first sub array, the remaining elements with value greater than pivot come in second sub array.

QUICKSORT (array A, start, end)
{
if (start < end)
{
p = partition(A, start, end)
QUICKSORT (A, start, p - 1)
QUICKSORT (A, p + 1, end)
}
}
	PARTITION (array A, start, end)
	{
	pivot ? A[end]
	i ? start-1
	for j ? start to end -1 {
	do if (A[j] < pivot) {
	then i ? i + 1
	swap A[i] with A[j]
	}}
	swap A[i+1] with A[end]
	return i+1

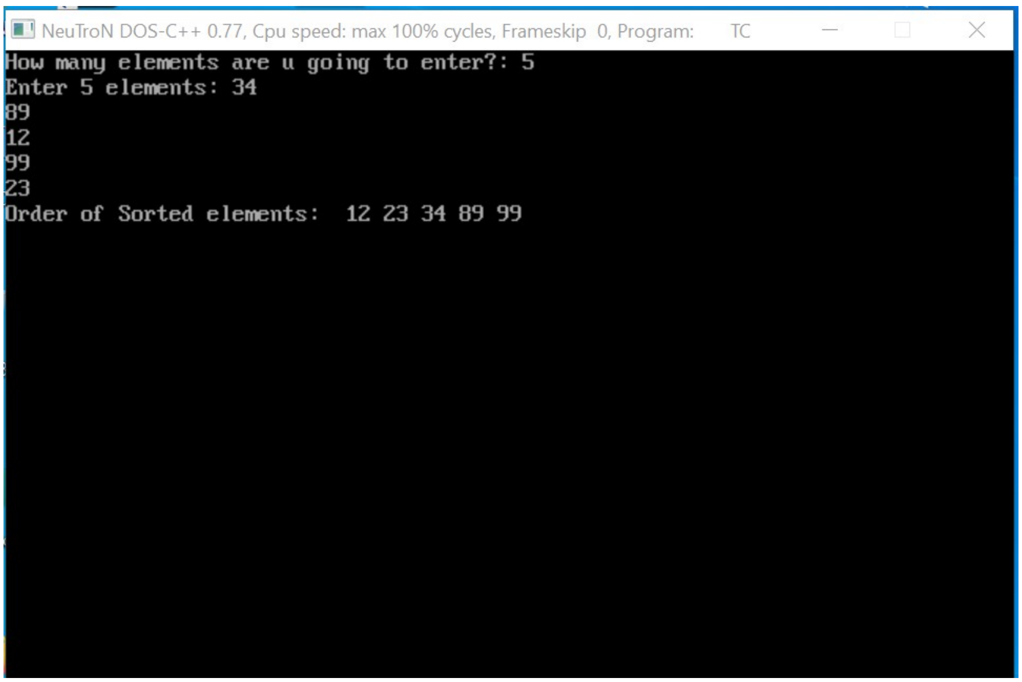
Program

```
#include<stdio.h>
#include<conio.h>
void quicksort(int number[25],int first,int last){
```

```
  int i, j, pivot, temp;
  if(first<last){
    pivot=first;
    i=first;
    j=last;
    while(i<j){
      while(number[i]<=number[pivot]&&i<last)
        i++;
      while(number[j]>number[pivot])
        j--;
      if(i<j){
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }
    temp=number[pivot];
    number[pivot]=number[j];
    number[j]=temp;
    quicksort(number,first,j-1);
    quicksort(number,j+1,last);
  }
}
int main(){
  int i, count, number[25];
  clrscr();
  printf("How many elements are u going to enter?: ");
  scanf("%d",&count);
  printf("Enter %d elements: ", count);
```

```
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);
  quicksort(number,0,count-1);
  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);
  getch();
  return 0;
}
```

Output



Result

Thus, the program to arrange a list of integers in ascending order using Quick sort Algorithm has been implemented successfully and the output is verified.

Ex.No: 12 b	Merge Sort
Date :	

Aim

To write a C program to arrange a list of integers in ascending order using Merge sort Algorithm

Theory

The Merge Sort algorithm is a sorting algorithm that is considered an example of the divide and conquer strategy. So, in this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner. It is a recursive algorithm that continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, we split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both the halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Algorithm

Step 1: Divide the array into 2 parts of lengths n/2 and n - n/2 respectively. Call these arrays as left half and right half respectively.

Step 2: Recursively sort the left half array and the right half array.

Step 3: Merge the left half array and right half-array to get the full array sorted.

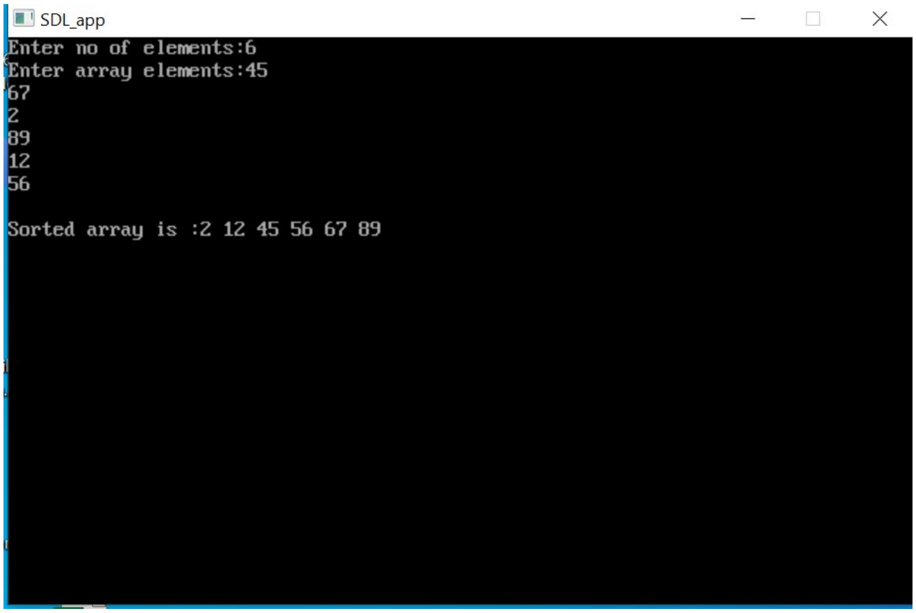
```
MERGE_SORT(arr, beg, end)
        if beg < end
        set mid = (beg + end)/2
        MERGE_SORT(arr, beg, mid)
        MERGE_SORT(arr, mid + 1, end)
        MERGE (arr, beg, mid, end)
        end of if
END MERGE_SORT
```

Program

```
#include<stdio.h>
#include<conio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);
 int main()
{
int a[30],n,i;
clrscr();
printf("Enter no of elements:");
scanf("%d",&n);
printf("Enter array elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
mergesort(a,0,n-1);
printf("\nSorted array is :");
for(i=0;i<n;i++)
printf("%d ",a[i]);
getch();
return 0;
}
 void mergesort(int a[],int i,int j)
{
int mid;
if(i<j)
{
mid=(i+j)/2;
mergesort(a,i,mid); //left recursion
mergesort(a,mid+1,j); //right recursion
```

```
merge(a,i,mid,mid+1,j); //merging of two sorted sub-arrays
}
}
 void merge(int a[],int i1,int j1,int i2,int j2)
{
int temp[50]; //array used for merging
int i,j,k;
i=i1; //beginning of the first list
j=i2; //beginning of the second list
k=0;
while(i<=j1 && j<=j2) //while elements in both lists
{
if(a[i]<a[j])
temp[k++]=a[i++];
else
temp[k++]=a[j++];
}
while(i<=j1) //copy remaining elements of the first list
temp[k++]=a[i++];
while(j<=j2) //copy remaining elements of the second list
temp[k++]=a[j++];
//Transfer elements from temp[] back to a[]
for(i=i1,j=0;i<=j2;i++,j++)
a[i]=temp[j];
}
```

Output



Result

Thus, the program to arrange a list of integers in ascending order using Merge sort Algorithm has been implemented successfully and the output is verified.

Ex.No: 13	Content Beyond Syllabus – Singly Linked List
Date :	

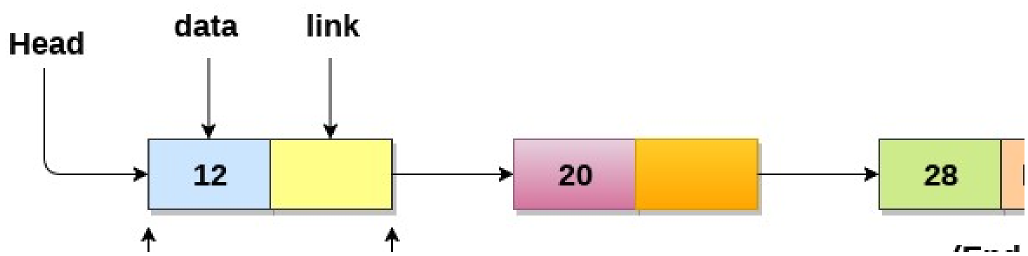
Aim

To write a C program to perform the various operations in singly linked list

Theory

A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail).

Each element in a linked list is called a node. A single node contains data and a pointer to the next node which helps in maintaining the structure of the list.



Algorithm

Traverse

Step 1: [INITIALIZE] SET PTR = HEAD

Step 2: Repeat Steps 3 and 4 while PTR != NULL

Step 3: Apply process to PTR -> DATA

Step 4: SET PTR = PTR->NEXT

[END OF LOOP]

Step 5: EXIT

Insert at beginning

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 7

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = HEAD

Step 6: SET HEAD = NEW_NODE

Step 7: EXIT

Insert at end

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 10

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = NULL

Step 6: SET PTR = HEAD

Step 7: Repeat Step 8 while PTR -> NEXT != NULL

Step 8: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 9: SET PTR -> NEXT = NEW_NODE

Step 10: EXIT

Insert after an Element

Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 12

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET PTR = HEAD

Step 6: SET PREPTR = PTR

Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA != NUM

Step 8: SET PREPTR = PTR

Step 9: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 10: PREPTR -> NEXT = NEW_NODE

Step 11: SET NEW_NODE -> NEXT = PTR

Step 12: EXIT

Delete from Beginning

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 5

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: SET HEAD = HEAD -> NEXT

Step 4: FREE PTR

Step 5: EXIT

Delete from End

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 6: SET PREPTR -> NEXT = NULL

Step 7: FREE PTR

Step 8: EXIT

Delete after a Node

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 10

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: SET PREPTR = PTR

Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM

Step 5: SET PREPTR = PTR

Step 6: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 7: SET TEMP = PTR

Step 8: SET PREPTR -> NEXT = PTR -> NEXT

Step 9: FREE TEMP

Step 10: EXIT

Search

Step 1: [INITIALIZE] SET PTR = HEAD

Step 2: Repeat Steps 3 and 4 while PTR != NULL

Step 3: If ITEM = PTR -> DATA

SET POS = PTR

Go To Step 5

ELSE

SET PTR = PTR -> NEXT

[END OF IF]

[END OF LOOP]

Step 4: SET POS = NULL

Step 5: EXIT

Program

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
    int choice =0;
    clrscr();
    while(choice != 9)
    {
        printf("\n\n*********Main Menu*********\n");
        printf("\nChoose one option from the following list ...\n");
        printf("\n===============================================\n");
```

```
        printf("\n1.Insert in beginning\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n5.Delete from last\n6.Delete node after specified location\n7.Search for an element\n8.Show\n9.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
      switch(choice)
      {
        case 1:
        beginsert();
        break;
        case 2:
        lastinsert();
        break;
        case 3:
        randominsert();
        break;
        case 4:
        begin_delete();
        break;
        case 5:
        last_delete();
        break;
        case 6:
        random_delete();
        break;
        case 7:
        search();
        break;
        case 8:
```

```
            display();
            break;
            case 9:
            exit(0);
            break;
            default:
            printf("Please enter valid choice..");
        }
    }
getch();
}
void beginsert()
{
    struct node *ptr;
    int item;
    ptr = (struct node *) malloc(sizeof(struct node *));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value\n");
        scanf("%d",&item);
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nNode inserted");
    }
```

```
}
void lastinsert()
{
    struct node *ptr,*temp;
    int item;
    ptr = (struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr->data = item;
        if(head == NULL)
        {
            ptr -> next = NULL;
            head = ptr;
            printf("\nNode inserted");
        }
        else
        {
            temp = head;
            while (temp -> next != NULL)
            {
                temp = temp -> next;
            }
```

```
            temp->next = ptr;
            ptr->next = NULL;
            printf("\nNode inserted");

        }
    }
}
void randominsert()
{
    int i,loc,item;
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
```

```
            printf("\ncan't insert\n");
            return;
          }

        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
    }
}
void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nList is empty\n");
    }
    else
    {
        ptr = head;
        head = ptr->next;
        free(ptr);
        printf("\nNode deleted from the begining ...\n");
    }
}
void last_delete()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
```

```
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nOnly node of the list deleted ...\n");
    }

    else
    {
        ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
        ptr1->next = NULL;
        free(ptr);
        printf("\nDeleted Node from the last ...\n");
    }
}
void random_delete()
{
    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which you want to perform deletion \n");
    scanf("%d",&loc);
```

```
    ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node %d ",loc+1);
}
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
```

```
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }      }              }
  void display()
{
    struct node *ptr;
    ptr = head;
    if(ptr == NULL)
    {
        printf("Nothing to print");
    }
    else
    {
        printf("\nprinting values . . . . .\n");
```

```
    while (ptr!=NULL)
    {
      printf("\n%d",ptr->data);
      ptr = ptr -> next;
    }     }  }
```

Output

```
*********Main Menu*********
Choose one option from the following list ...
===============================================
1.Insert in beginning
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
1
Enter value
1
Node inserted
*********Main Menu*********
Choose one option from the following list ...
===============================================
1.Insert in beginning
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
2
Enter value?
2
Node inserted
*********Main Menu*********
```

Choose one option from the following list ...
===
1.Insert in beginning
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
3
Enter element value1
Enter the location after which you want to insert 1
Node inserted
*********Main Menu*********
Choose one option from the following list ...
===
1.Insert in beginning
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
8
printing values . . . . .
1
2
1

*********Main Menu*********
Choose one option from the following list ...
===
1.Insert in begining
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location

7.Search for an element
8.Show
9.Exit
Enter your choice?
2
Enter value?
123
Node inserted
*********Main Menu*********
Choose one option from the following list ...
===
1.Insert in beginning
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
1
Enter value
1234
Node inserted
*********Main Menu*********
Choose one option from the following list ...
===
1.Insert in beginning
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
4
Node deleted from the beginning ...
*********Main Menu*********
Choose one option from the following list ...
===
1.Insert in beginning

2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
5
Deleted Node from the last ...
*********Main Menu*********
Choose one option from the following list ...
===
1.Insert in beginning
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit

Enter your choice?
6
Enter the location of the node after which you want to perform deletion
1
Deleted node 2
*********Main Menu*********
Choose one option from the following list ...
===
1.Insert in beginning
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
8
printing values . . . . .

1
1
*********Main Menu*********
Choose one option from the following list ...
===
1.Insert in beginning
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
7
Enter item which you want to search?
1
item found at location 1
item found at location 2
*********Main Menu*********
Choose one option from the following list ...
===
1.Insert in beginning
2.Insert at last
3.Insert at any random location
4.Delete from Beginning
5.Delete from last
6.Delete node after specified location
7.Search for an element
8.Show
9.Exit
Enter your choice?
9

Result

Thus, the program to perform various operations using singly linked list has been implemented successfully and the output is verified.