

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**LAB RECORD**

**CSB4146 - DATA STRUCTURES LAB**

**2019-2020**

**STUDENT NAME: K.SUMANTH KUMAR REDDY**\_

**REGISTER NUMBER: 19113105**\_\_\_\_\_\_\_\_\_\_\_\_\_

**SEMESTER & SECTION** : 2nd sem\_& CSE 2B\_\_\_\_



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

REGISTRATION NUMBER: **19113105**\_\_\_\_\_\_\_\_\_\_\_

Certified that this is a bonafide record work done by **K. SUMANTH KUMAR REDDY** of B.Tech. Computer Science and Engineering, for the course  **CSB 4116 – Data Structures Lab in the second semester during the period January 2020 to May 2020** at Hindustan Institute of Technology and Science, Chennai - 603103.

**Faculty Incharge**

**(Angelina Geetha)**

This record is submitted for the practical examination held on \_\_\_\_\_\_\_\_\_\_\_\_

**B.Tech.(CSE) – II Semester – B Section**

**CSB 4146 – Data Structures Lab Assessment Sheet**

Name**: K.SUMANTH KUMAR REDDY** Roll. No: **19113105**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| S.  No. | Date | Title | Program writing  (5) | Debugging & Output (2) | Inference  & Viva  (3) | Total  (10) | Sign |
| 1 | 10-01-2020 | Arrays – To find largest and smallest element |  |  |  |  |  |
| 2 | 24-01-2020 | Stacks using Arrays |  |  |  |  |  |
| 3 | 31-01-2020 | Queues using Arrays |  |  |  |  |  |
| 4 | 07-02-2020 | Operations on a Singly linked list |  |  |  |  |  |
| 5 | 14-02-2020 | Operations on a Doubly linked list |  |  |  |  |  |
| 6 | 28-02-2020 | Stack using Singly Linked List |  |  |  |  |  |
| 7 | 28-02-2020 | Queues using Singly Linked List |  |  |  |  |  |
| 8 | 06-03-2020 | Binary Search Tree   * Insertion * Deletion |  |  |  |  |  |
| 9 | 13-03-2020 | Binary Search Tree   * Insertion * Inorder Traversal * Post order Traversal * Preorder Traversal |  |  |  |  |  |
| 10 | 17-04-2020 | Linear Search |  |  |  |  |  |
| 11 | 17-04-2020 | Binary Search |  |  |  |  |  |
| 12 | 24-04-2020 | Bubble Sort |  |  |  |  |  |
| 13 | 27-04-2020 | Insertion Sort |  |  |  |  |  |
| 14 | 30-04-2020 | Selection Sort |  |  |  |  |  |
| 15 | 04-05-2020 | Shell Sort |  |  |  |  |  |
| 16 | 06-05-2020 | Quick Sort |  |  |  |  |  |
| 17 | 12-05-2020 | Heap Sort |  |  |  |  |  |

Faculty Incharge: 1. Dr. Angelina Geetha 2. Dr. M. Judith Leo

**Title:** Arrays- To find largest and smallest element

**Date:** 10-01-2020

**Program No:** 01

**Problem Definition:**

* An array in C or C++ is a collection of items stored at contiguous memory locations and elements can be accessed randomly using indices of an array.
* They are used to store similar type of elements as in the data type must be the same for all elements.
* They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type.
* To add to it, an array in C or C++ can store derived data types such as the structures, pointers etc. Given below is the picturesque representation of an array.

**PROGRAM FOR ARRAYS- TO find largest and smallest element**

#include<stdio.h>

int main()

{

int a[50],i,n,large,small;

printf("\t Finding smallest and largest element in an array\n\n\n");

printf("Enter number of elements:");

scanf("%d",&n);

printf("\nEnter the elements:");

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

scanf("%d",&a[i]);

large=small=a[0];

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

{

if(a[i]>large)

large=a[i];

if(a[i]<small)

small=a[i];

}

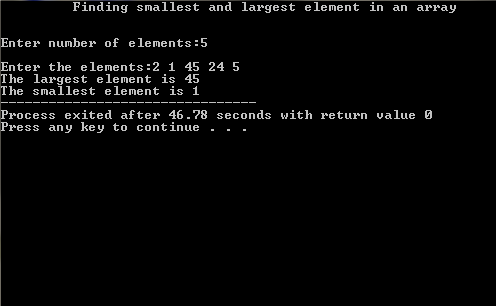
printf("The largest element is %d",large);

printf("\nThe smallest element is %d",small);

return 0;

}

**OUTPUT:**



**Title:** Stack using Arrays

**Date:** 24-01-2020

**Program No:** 02

**Problem Definition:**

* Stack is a LIFO (last in first out) structure.
* It is an ordered list of the same type of elements.
* A stack is a linear list where all insertions and deletions are permitted only at one end of the list.
* When elements are added to stack it grow at one end. Similarly, when elements are deleted from a stack, it shrinks at the same end.
* Applications of stack are: 1. Expression Evaluation 2. Back tracking 3. Function call

**PROGRAM FOR STACKS USING ARRAYS:**

#include<stdio.h>

int stack[100],choice,n,top,x,i;

void push(void);

void pop(void);

void display(void);

int main()

{

    top=-1;

printf("\n Stack using arrays\n");

printf("\n Enter the Stack size:");

    scanf("%d",&n);

    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]);

    }

    else

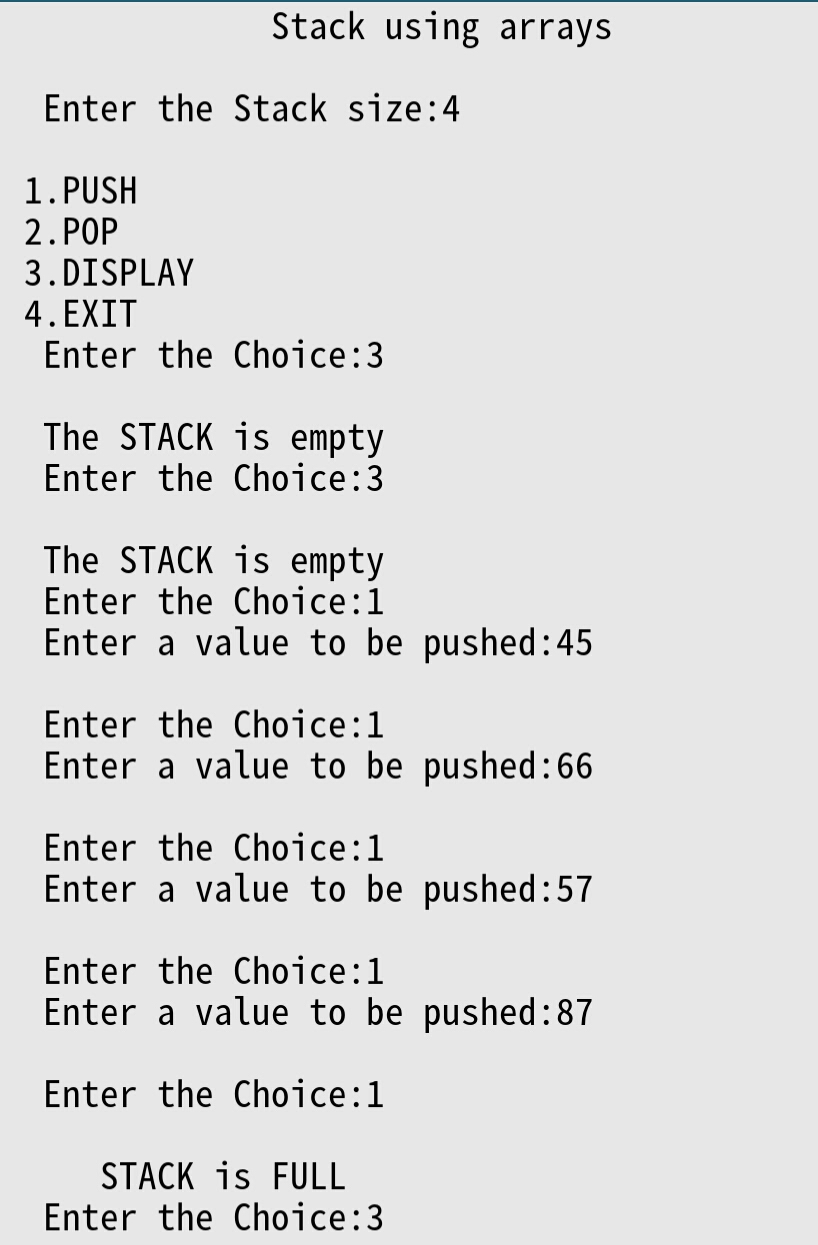
    {

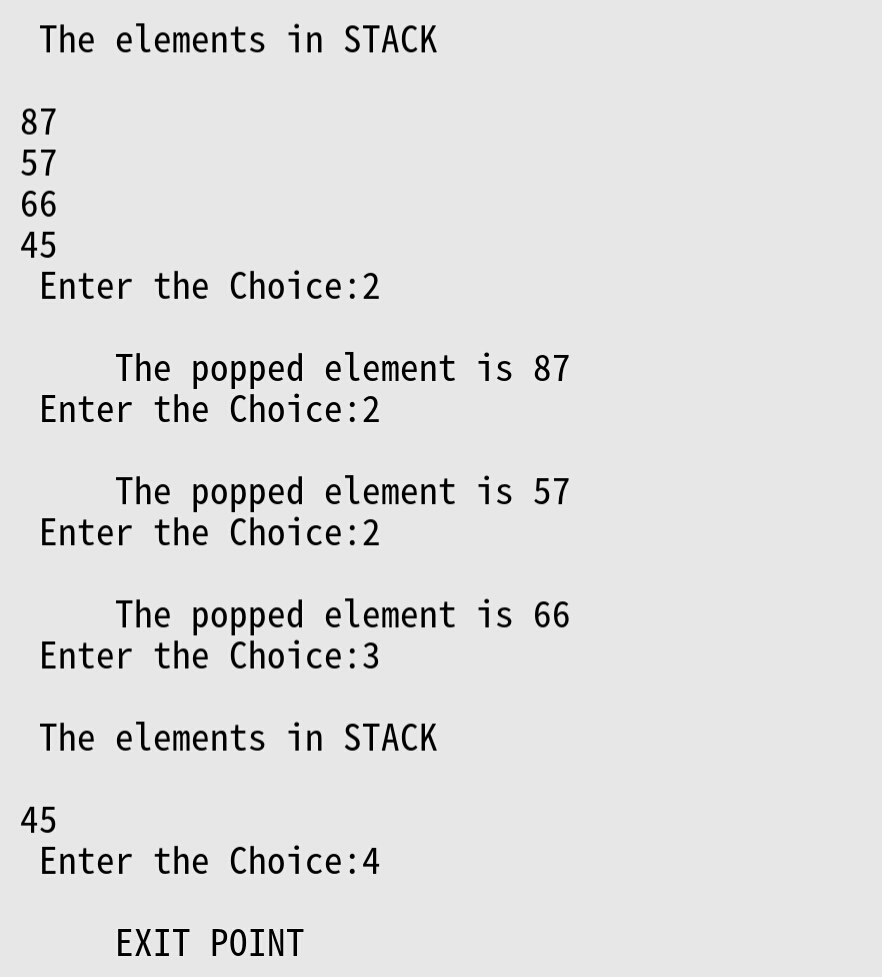
        printf("\n The STACK is empty");

    }

}

**OUTPUT:**



****

**Title:** Queues using arrays

**Date:** 31-01-2020

**Program No:** 03

**Problem Definition:**

* A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.
* Queue follows the First In First Out(FIFO) rule - the item that goes in first is the item that comes out first.
* A major disadvantage of a classical queue is that a new element can only be inserted when all of the elements are deleted from the queue. Reusability of space is not possible. This can be overcome by Circular queues.
* Applications of Queues are:

1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2. Data packets waiting to be transmitted over the Internet.
3. There are various queues quietly doing their job in your computer's (or the network's) operating system.

**PROGRAM FOR QUEUES USING ARRAYS:**

#include<stdio.h>

#include<conio.h>

#define n 5

int main()

{

    int queue[n],ch=1,front=0,rear=0,i,j=1,x=n;

    printf("Queue using Array");

    printf("\n1.Insertion \n2.Deletion \n3.Display \n4.Exit");

    while(ch)

    {

        printf("\nEnter the Choice:");

        scanf("%d",&ch);

        switch(ch)

        {

        case 1:

            if(rear==x)

                printf("\n Queue is Full");

            else

            {

                printf("\n Enter no %d:",j++);

                scanf("%d",&queue[rear++]);

            }

            break;

        case 2:

            if(front==rear)

            {

                printf("\n Queue is empty");

            }

            else

            {

                printf("\n Deleted Element is %d",queue[front++]);

                x++;

            }

            break;

        case 3:

            printf("\n Queue Elements are:\n ");

            if(front==rear)

                printf("\n Queue is Empty");

            else

            {

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

                {

                    printf("%d",queue[i]);

                    printf("\n");

                }

                break;

            case 4:

                printf("Completed\n");

exit(0);

            default:

                printf("Wrong Choice: please see the options");

            }

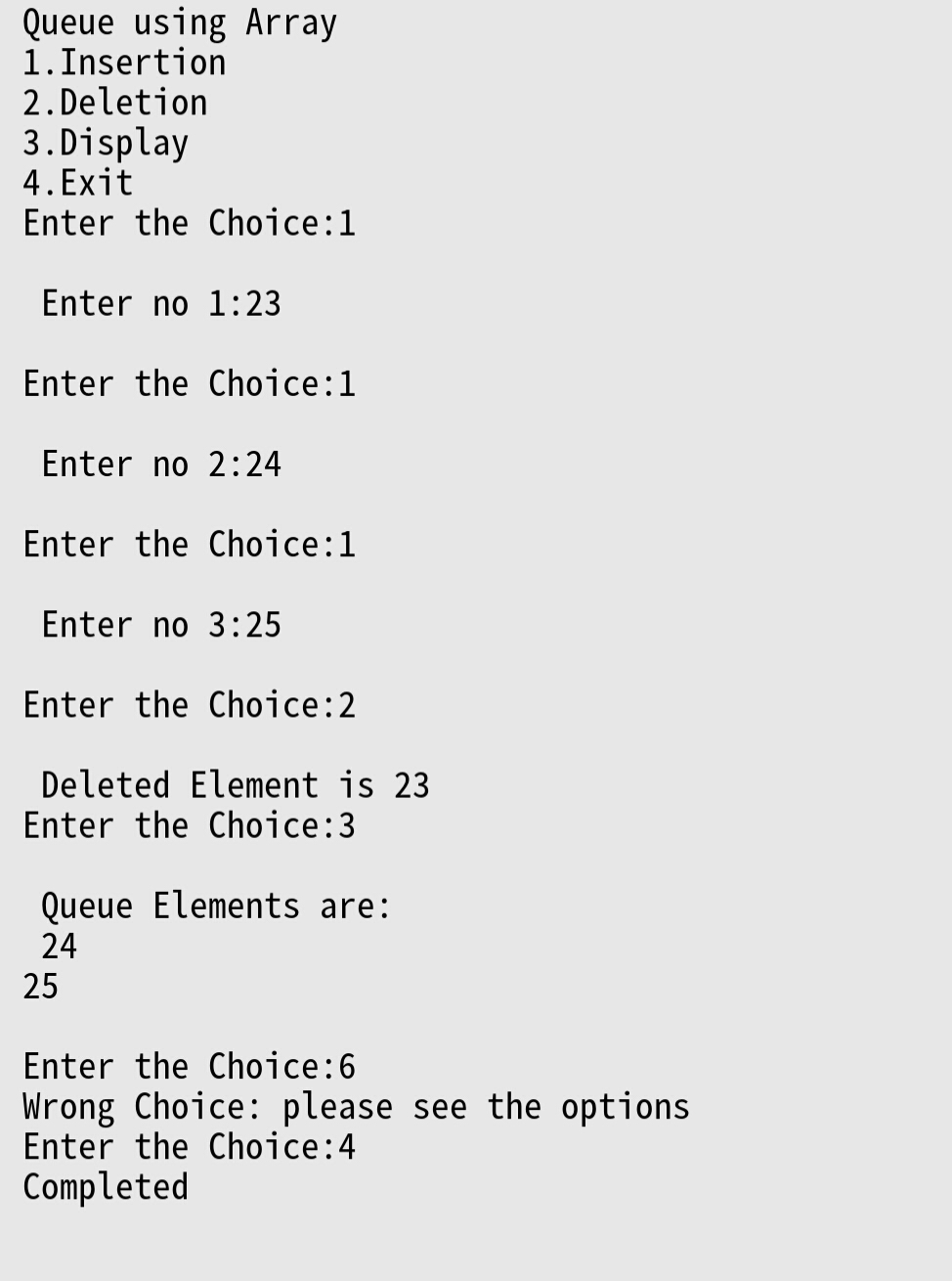
        }

    }

    getch();

}

**OUTPUT:**



**Title:** Operations on Single Linked List

**Date:** 07-02-2020

**Program No:** 04

**Problem Definition:**

* Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.
* It is known as One-way list.
* A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.
* Each node in a list consists of at least two parts:

1. data
2. pointer to the next node
3. Advantages are: 1. Dynamic size 2. Ease of insertion and deletion.

**PROGRAM FOR SINGLE LINKED LIST OPERATIONS:**

#include<stdio.h>

#include<stdlib.h>

void create();

void display();

void insert\_begin();

void insert\_end();

void insert\_pos();

void delete\_begin();

void delete\_end();

void delete\_pos();

struct node

{

int info;

struct node \*next;

};

struct node \*start=NULL;

int main()

{

int choice;

printf("\t\t\tSINGLE LINKED LIST OPERATIONS\n\n");

printf("\n 1.Create \n");

printf("\n 2.Display \n");

printf("\n 3.Insert at the beginning \n");

printf("\n 4.Insert at the end \n");

printf("\n 5.Insert at specified position \n");

printf("\n 6.Delete from beginning \n");

printf("\n 7.Delete from the end \n");

printf("\n 8.Delete from specified position \n");

printf("\n 9.Exit \n");

printf("\n--------------------------------------\n");

do

{

printf("\n Enter your choice:\t");

scanf("%d",&choice);

switch(choice)

{

case 1:

create();

break;

case 2:

display();

break;

case 3:

insert\_begin();

break;

case 4:

insert\_end();

break;

case 5:

insert\_pos();

break;

case 6:

delete\_begin();

break;

case 7:

delete\_end();

break;

case 8:

delete\_pos();

break;

case 9:

exit(0);

break;

default:

printf("\n Wrong Choice:\n");

break;

}

}

while(choice!=9);

return 0;

}

void create()

{

struct node \*temp,\*ptr;

temp=(struct node \*)malloc(sizeof(struct node));

if(temp==NULL)

{

printf("\nOut of Memory Space:\n");

exit(0);

}

printf("\nEnter the data value for the node:\t");

scanf("%d",&temp->info);

temp->next=NULL;

if(start==NULL)

{

start=temp;

}

else

{

ptr=start;

while(ptr->next!=NULL)

{

ptr=ptr->next;

}

ptr->next=temp;

}

}

void display()

{

struct node \*ptr;

if(start==NULL)

{

printf("\nList is empty:\n");

return;

}

else

{

ptr=start;

printf("\nThe List elements are:\n");

while(ptr!=NULL)

{

printf("%d\t",ptr->info );

ptr=ptr->next ;

}

}

}

void insert\_begin()

{

struct node \*temp;

temp=(struct node \*)malloc(sizeof(struct node));

if(temp==NULL)

{

printf("\nOut of Memory Space:\n");

return;

}

printf("\nEnter the data value for the node:\t" );

scanf("%d",&temp->info);

temp->next =NULL;

if(start==NULL)

{

start=temp;

}

else

{

temp->next=start;

start=temp;

}

}

void insert\_end()

{

struct node \*temp,\*ptr;

temp=(struct node \*)malloc(sizeof(struct node));

if(temp==NULL)

{

printf("\nOut of Memory Space:\n");

return;

}

printf("\nEnter the data value for the node:\t" );

scanf("%d",&temp->info );

temp->next =NULL;

if(start==NULL)

{

start=temp;

}

else

{

ptr=start;

while(ptr->next !=NULL)

{

ptr=ptr->next ;

}

ptr->next =temp;

}

}

void insert\_pos()

{

struct node \*ptr,\*temp;

int i,pos;

temp=(struct node \*)malloc(sizeof(struct node));

if(temp==NULL)

{

printf("\nOut of Memory Space:\n");

return;

}

printf("\nEnter the position for the new node to be inserted:\t");

scanf("%d",&pos);

printf("\nEnter the data value of the node:\t");

scanf("%d",&temp->info) ;

temp->next=NULL;

if(pos==0)

{

temp->next=start;

start=temp;

}

else

{

for(i=0,ptr=start;i<pos-1;i++)

{

ptr=ptr->next;

if(ptr==NULL)

{

printf("\nPosition not found:[Handle with care]\n");

return;

}

}

temp->next =ptr->next ;

ptr->next=temp;

}

}

void delete\_begin()

{

struct node \*ptr;

if(ptr==NULL)

{

printf("\nList is Empty:\n");

return;

}

else

{

ptr=start;

start=start->next ;

printf("\nThe deleted element is :%d\t",ptr->info);

free(ptr);

}

}

void delete\_end()

{

struct node \*temp,\*ptr;

if(start==NULL)

{

printf("\nList is Empty:");

exit(0);

}

else if(start->next ==NULL)

{

ptr=start;

start=NULL;

printf("\nThe deleted element is:%d\t",ptr->info);

free(ptr);

}

else

{

ptr=start;

while(ptr->next!=NULL)

{

temp=ptr;

ptr=ptr->next;

}

temp->next=NULL;

printf("\nThe deleted element is:%d\t",ptr->info);

free(ptr);

}

}

void delete\_pos()

{

int i,pos;

struct node \*temp,\*ptr;

if(start==NULL)

{

printf("\nThe List is Empty:\n");

exit(0);

}

else

{

printf("\nEnter the position of the node to be deleted:\t");

scanf("%d",&pos);

if(pos==0)

{

ptr=start;

start=start->next ;

printf("\nThe deleted element is:%d\t",ptr->info );

free(ptr);

}

else

{

ptr=start;

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

{

temp=ptr;

ptr=ptr->next ;

if(ptr==NULL)

{

printf("\nPosition not Found:\n");

return;

}

}

temp->next =ptr->next ;

printf("\nThe deleted element is:%d\t",ptr->info );

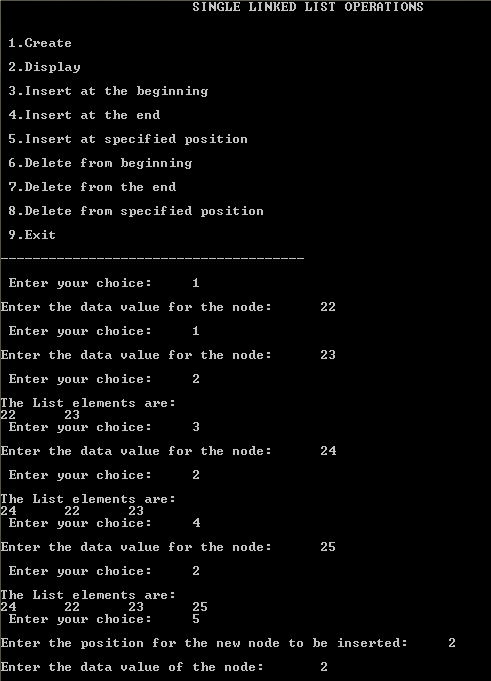
free(ptr);

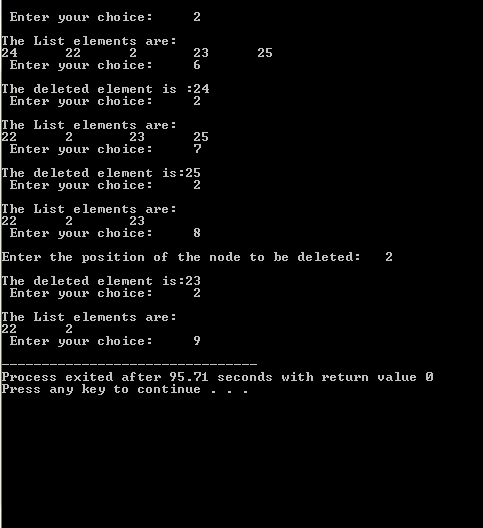
}

}

}

**OUTPUT:**





**Title:** Operations on Double Linked List

**Date:** 14-02-2020

**Program No:** 05

**Problem Definition:**

* Double linked list is known as Two-way list.
* The node structure has two members:
  1. data stores the information
  2. flink pointer holds the address of the next node.
  3. blink pointer holds the address of the previous node
* It can be traversed both in forward and backward directions.
* It uses more memory and space due to two pointers.

**PROGRAM FOR DOUBLE LINKED OPERATIONS:**

#include<stdio.h>

#include<stdlib.h>

struct node

{

struct node \*prev;

struct node \*next;

int data;

};

struct node \*head;

void insertion\_beginning();

void insertion\_last();

void insertion\_specified();

void deletion\_beginning();

void deletion\_last();

void deletion\_specified();

void display();

int main ()

{

int choice =0;

printf("\n\t\t\t OPERATIONS ON DOUBLE LINKED LIST \n\n");

printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n5.Delete from last\n6.Delete the node after the given data\n7.Display\n8.Exit\n");

do

{

printf("\nEnter your choice:");

scanf("\n%d",&choice);

switch(choice)

{

case 1:

insertion\_beginning();

break;

case 2:

insertion\_last();

break;

case 3:

insertion\_specified();

break;

case 4:

deletion\_beginning();

break;

case 5:

deletion\_last();

break;

case 6:

deletion\_specified();

break;

case 7:

display();

break;

case 8:

exit(0);

break;

default:

printf("Please enter valid choice..");

}

}

while(choice!=8);

return 0;

}

void insertion\_beginning()

{

struct node \*ptr;

int item;

ptr = (struct node \*)malloc(sizeof(struct node));

if(ptr == NULL)

{

printf("\nOVERFLOW");

}

else

{

printf("\nEnter Item value:");

scanf("%d",&item);

if(head==NULL)

{

ptr->next = NULL;

ptr->prev=NULL;

ptr->data=item;

head=ptr;

}

else

{

ptr->data=item;

ptr->prev=NULL;

ptr->next = head;

head->prev=ptr;

head=ptr;

}

printf("\nNode inserted\n");

}

}

void insertion\_last()

{

struct node \*ptr,\*temp;

int item;

ptr = (struct node \*) malloc(sizeof(struct node));

if(ptr == NULL)

{

printf("\nOVERFLOW");

}

else

{

printf("\nEnter value:");

scanf("%d",&item);

ptr->data=item;

if(head == NULL)

{

ptr->next = NULL;

ptr->prev = NULL;

head = ptr;

}

else

{

temp = head;

while(temp->next!=NULL)

{

temp = temp->next;

}

temp->next = ptr;

ptr ->prev=temp;

ptr->next = NULL;

}

}

printf("\nnode inserted\n");

}

void insertion\_specified()

{

struct node \*ptr,\*temp;

int item,loc,i;

ptr = (struct node \*)malloc(sizeof(struct node));

if(ptr == NULL)

{

printf("\n OVERFLOW");

}

else

{

temp=head;

printf("Enter the location:");

scanf("%d",&loc);

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

{

temp = temp->next;

if(temp == NULL)

{

printf("\n There are less than %d elements", loc);

return;

}

}

printf("Enter value:");

scanf("%d",&item);

ptr->data = item;

ptr->next = temp->next;

ptr -> prev = temp;

temp->next = ptr;

temp->next->prev=ptr;

printf("\nnode inserted\n");

}

}

void deletion\_beginning()

{

struct node \*ptr;

if(head == NULL)

{

printf("\n UNDERFLOW");

}

else if(head->next == NULL)

{

head = NULL;

free(head);

printf("\nnode deleted\n");

}

else

{

ptr = head;

head = head -> next;

head -> prev = NULL;

free(ptr);

printf("\nnode deleted\n");

}

}

void deletion\_last()

{

struct node \*ptr;

if(head == NULL)

{

printf("\n UNDERFLOW");

}

else if(head->next == NULL)

{

head = NULL;

free(head);

printf("\nnode deleted\n");

}

else

{

ptr = head;

if(ptr->next != NULL)

{

ptr = ptr -> next;

}

ptr -> prev -> next = NULL;

free(ptr);

printf("\nnode deleted\n");

}

}

void deletion\_specified()

{

struct node \*ptr, \*temp;

int val;

printf("\n Enter the data after which the node is to be deleted : ");

scanf("%d", &val);

ptr = head;

while(ptr -> data != val)

ptr = ptr -> next;

if(ptr -> next == NULL)

{

printf("\nCan't delete\n");

}

else if(ptr -> next -> next == NULL)

{

ptr ->next = NULL;

}

else

{

temp = ptr -> next;

ptr -> next = temp -> next;

temp -> next -> prev = ptr;

free(temp);

printf("\nnode deleted\n");

}

}

void display()

{

struct node \*ptr;

printf("\n printing values...\n");

ptr = head;

while(ptr != NULL)

{

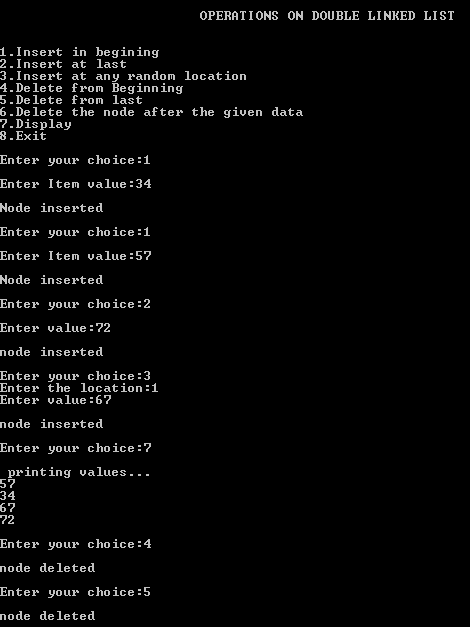
printf("%d\n",ptr->data);

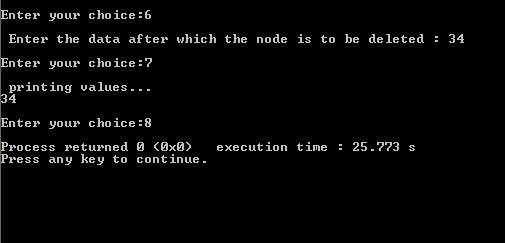
ptr=ptr->next;

}

}

**OUTPUT:**





**Title:** Stack using single linked list

**Date:** 28-02-2020

**Program No:** 06

**Problem Definition:**

* A stack can be easily implemented through the linked list.
* In stack Implementation, a stack contains a top pointer which is “head” of the stack where pushing and popping items happens at the head of the list first node have null in link field and second node link have first node address in link field and so on and last node address in “top” pointer.
* The main advantage of using linked list over an arrays is that it is possible to implements a stack that can shrink or grow as much as needed.
* In using array will put a restriction to the maximum capacity of the array which can lead to stack overflow. Here each new node will be dynamically allocate. So overflow is not possible.

**PROGRAM FOR STACK USING SINGLE LINKED LIST:**

#include<stdio.h>

#include<conio.h>

struct Node

{

int data;

struct Node \*next;

}\*top = NULL;

void push(int);

void pop();

void display();

int main()

{

int choice, value;

printf("\t\t\t Stack using Linked List \n");

printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");

do

{

printf("Enter your choice: ");

scanf("%d",&choice);

switch(choice)

{

case 1: printf("Enter the value to be insert: ");

scanf("%d", &value);

push(value);

break;

case 2: pop(); break;

case 3: display(); break;

case 4: printf("\nExit point");

default: printf("\nWrong selection!!! Please try again!!!\n");

}

}

while(choice!=4);

return 0;

}

void push(int value)

{

struct Node \*newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

if(top == NULL)

newNode->next = NULL;

else

newNode->next = top;

top = newNode;

printf("\nInsertion is Success!!!\n");

}

void pop()

{

if(top == NULL)

printf("\nStack is Empty!!!\n");

else{

struct Node \*temp = top;

printf("\nDeleted element: %d", temp->data);

top = temp->next;

free(temp);

}

}

void display()

{

if(top == NULL)

printf("\nStack is Empty!!!\n");

else{

struct Node \*temp = top;

while(temp->next != NULL){

printf("%d--->",temp->data);

temp = temp -> next;

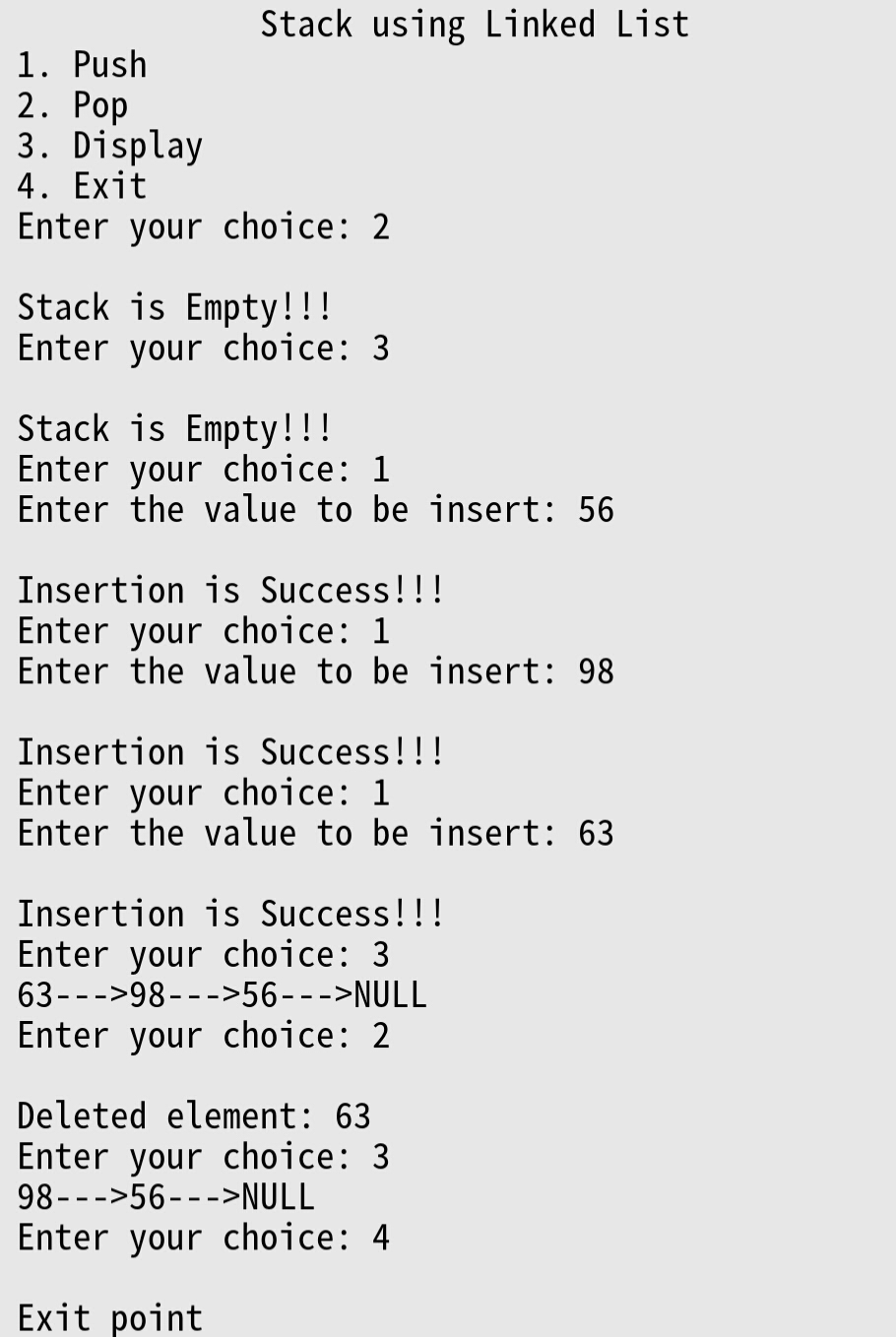
}

printf("%d--->NULL",temp->data);

}

}

**OUTPUT:**



**Title:** Queues using single linked list

**Date:** 28-02-2020

**Program No:** 07

**Problem Definition:**

* In a linked queue, each node of the queue consists of two parts i.e. data part and the link part.
* Each element of the queue points to its immediate next element in the memory.
* In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer.
* The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.
* Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

**PROGRAM FOR QUEUES USING SINGLE LINKED LIST:**

#include<stdio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*next;

};

struct node \*front;

struct node \*rear;

void insert();

void delete();

void display();

int main ()

{

int choice;

printf("\t\t Queues using single link list\n\n\n");

printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");

do

{

printf("\nEnter your choice :");

scanf("%d",& choice);

switch(choice)

{

case 1:

insert();

break;

case 2:

delete();

break;

case 3:

display();

break;

case 4:

exit(0);

break;

default:

printf("\nEnter valid choice??\n");

}

}

while(choice!=4);

return 0;

}

void insert()

{

struct node \*ptr;

int item;

ptr = (struct node \*) malloc (sizeof(struct node));

if(ptr == NULL)

{

printf("\nOVERFLOW\n");

return;

}

else

{

printf("\nEnter value?\n");

scanf("%d",&item);

ptr -> data = item;

if(front == NULL)

{

front = ptr;

rear = ptr;

front -> next = NULL;

rear -> next = NULL;

}

else

{

rear -> next = ptr;

rear = ptr;

rear->next = NULL;

}

}

}

void delete ()

{

struct node \*ptr;

if(front == NULL)

{

printf("\nUNDERFLOW\n");

return;

}

else

{

ptr = front;

front = front -> next;

free(ptr);

}

}

void display()

{

struct node \*ptr;

ptr = front;

if(front == NULL)

{

printf("\nEmpty queue\n");

}

else

{ printf("\nprinting values .....\n");

while(ptr != NULL)

{

printf("\n%d\n",ptr -> data);

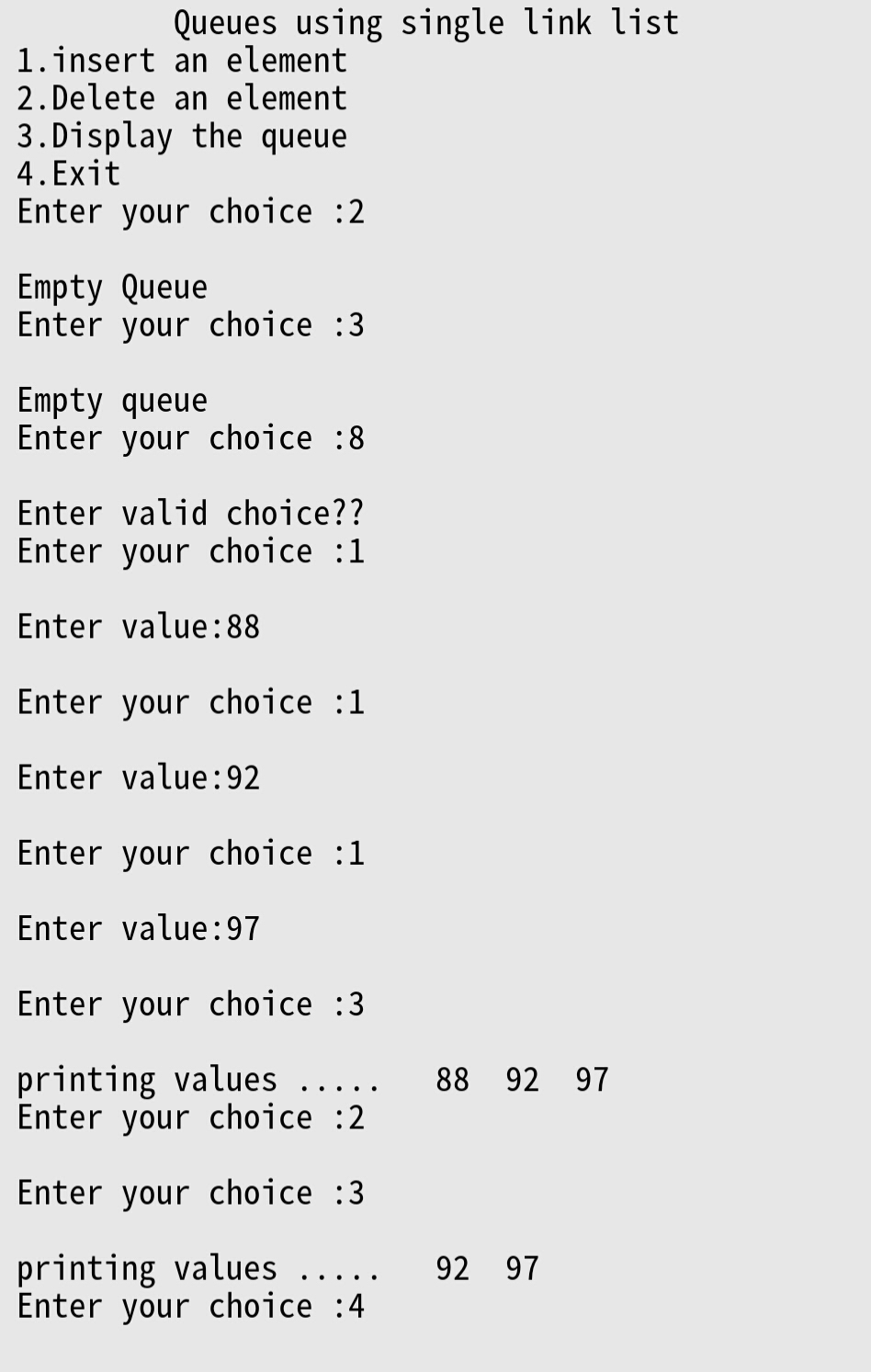
ptr = ptr -> next;

}

}

}

**OUTPUT:**

****

**Title:** Binary search tree operations (insertion, deletion)

**Date:** 06-03-2020

**Program No:** 8

**Problem Definition:**

* 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 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 separates a binary search tree from a regular [binary tree](https://www.programiz.com/data-structures/trees) is

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

**PROGRAM FOR BINARY SEARCH OPERATIONS:**

# include <stdio.h>

# include <malloc.h>

struct node

{

int info;

struct node \*lchild;

struct node \*rchild;

}

\*root;

void find(int item,struct node \*\*par,struct node \*\*loc)

{

struct node \*ptr,\*ptrsave;

if(root==NULL)

{

\*loc=NULL;

\*par=NULL;

return;

}

if(item==root->info)

{

\*loc=root;

\*par=NULL;

return;

}

if(item<root->info)

ptr=root->lchild;

else

ptr=root->rchild;

ptrsave=root;

while(ptr!=NULL)

{

if(item==ptr->info)

{ \*loc=ptr;

\*par=ptrsave;

return;

}

ptrsave=ptr;

if(item<ptr->info)

ptr=ptr->lchild;

else

ptr=ptr->rchild;

}

\*loc=NULL;

\*par=ptrsave;

}

void insert(int item)

{ struct node \*tmp,\*parent,\*location;

find(item,&parent,&location);

if(location!=NULL)

{

printf("Item already present");

return;

}

tmp=(struct node \*)malloc(sizeof(struct node));

tmp->info=item;

tmp->lchild=NULL;

tmp->rchild=NULL;

if(parent==NULL)

root=tmp;

else

if(item<parent->info)

parent->lchild=tmp;

else

parent->rchild=tmp;

}

void case\_a(struct node \*par,struct node \*loc )

{

if(par==NULL)

root=NULL;

else

if(loc==par->lchild)

par->lchild=NULL;

else

par->rchild=NULL;

}

void case\_b(struct node \*par,struct node \*loc)

{

struct node \*child;

if(loc->lchild!=NULL)

child=loc->lchild;

else

child=loc->rchild;

if(par==NULL )

root=child;

else

if( loc==par->lchild)

par->lchild=child;

else

par->rchild=child;

}

void case\_c(struct node \*par,struct node \*loc)

{

struct node \*ptr,\*ptrsave,\*suc,\*parsuc;

ptrsave=loc;

ptr=loc->rchild;

while(ptr->lchild!=NULL)

{

ptrsave=ptr;

ptr=ptr->lchild;

}

suc=ptr;

parsuc=ptrsave;

if(suc->lchild==NULL && suc->rchild==NULL)

case\_a(parsuc,suc);

else

case\_b(parsuc,suc);

if(par==NULL)

root=suc;

else

if(loc==par->lchild)

par->lchild=suc;

else

par->rchild=suc;

suc->lchild=loc->lchild;

suc->rchild=loc->rchild;

}

int del(int item)

{

struct node \*parent,\*location;

if(root==NULL)

{

printf("Tree empty");

return 0;

}

find(item,&parent,&location);

if(location==NULL)

{

printf("Item not present in tree");

return 0;

}

if(location->lchild==NULL && location->rchild==NULL)

case\_a(parent,location);

if(location->lchild!=NULL && location->rchild==NULL)

case\_b(parent,location);

if(location->lchild==NULL && location->rchild!=NULL)

case\_b(parent,location);

if(location->lchild!=NULL && location->rchild!=NULL)

case\_c(parent,location);

free(location);

}

void display(struct node \*ptr,int level)

{

int i;

if ( ptr!=NULL )

{

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

printf(" ");

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

printf(" ");

printf("%d\n", ptr->info);

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

}

}

int main()

{

int choice,num;

root=NULL;

printf("\n\t\t\t BINARY SEARCH TREE OPERATIONS \n");

printf("1.Insert\n");

printf("2.Delete\n");

printf("3.display\n");

printf("4.quit\n");

do

{

printf("\nEnter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("Enter the number to be inserted : ");

scanf("%d",&num);

insert(num);

break;

case 2:

printf("\nEnter the number to be deleted : \n");

scanf("%d",&num);

del(num);

break;

case 3:

display(root,1);

break;

case 4:

break;

default:

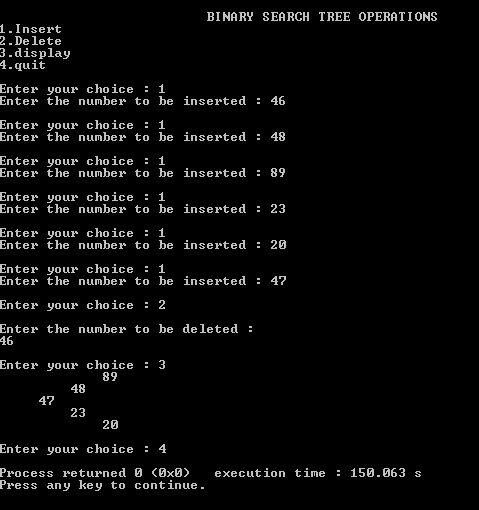
printf("\n Wrong choice\n");

}

}

while(choice!=4);

}

**OUTPUT:  
**

**Title:** Binary search operations(In-order, Pre-order, Post-order)

**Date:** 13-03-2020

**Program No:** 09

**Problem Definition:**

* There are three ways which we use to traverse a tree –

1. In-order Traversal
2. Pre-order Traversal
3. Post-order Traversal

* **In-order Traversal (L Rt R):** In this method the left subtree is visited first, then the root and later the right sub-tree. The output will produce sorted key values in an ascending order.
* **Pre-order Traversal (Rt L R):** In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.
* **Post-order Traversal (L R Rt**): In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

**PROGRAM FOR BINARY SEARCH TREE TRAVERSALS:**

# include <stdio.h>

# include <malloc.h>

struct node

{

int info;

struct node \*lchild;

struct node \*rchild;

}

\*root;

void find(int item,struct node \*\*par,struct node \*\*loc)

{

struct node \*ptr,\*ptrsave;

if(root==NULL)

{

\*loc=NULL;

\*par=NULL;

return;

}

if(item==root->info)

{

\*loc=root;

\*par=NULL;

return;

}

if(item<root->info)

ptr=root->lchild;

else

ptr=root->rchild;

ptrsave=root;

while(ptr!=NULL)

{

if(item==ptr->info)

{ \*loc=ptr;

\*par=ptrsave;

return;

}

ptrsave=ptr;

if(item<ptr->info)

ptr=ptr->lchild;

else

ptr=ptr->rchild;

}

\*loc=NULL;

\*par=ptrsave;

}

void insert(int item)

{ struct node \*tmp,\*parent,\*location;

find(item,&parent,&location);

if(location!=NULL)

{

printf("Item already present");

return;

}

tmp=(struct node \*)malloc(sizeof(struct node));

tmp->info=item;

tmp->lchild=NULL;

tmp->rchild=NULL;

if(parent==NULL)

root=tmp;

else

if(item<parent->info)

parent->lchild=tmp;

else

parent->rchild=tmp;

}

void case\_a(struct node \*par,struct node \*loc )

{

if(par==NULL)

root=NULL;

else

if(loc==par->lchild)

par->lchild=NULL;

else

par->rchild=NULL;

}

void case\_b(struct node \*par,struct node \*loc)

{

struct node \*child;

if(loc->lchild!=NULL)

child=loc->lchild;

else

child=loc->rchild;

if(par==NULL )

root=child;

else

if( loc==par->lchild)

par->lchild=child;

else

par->rchild=child;

}

void case\_c(struct node \*par,struct node \*loc)

{

struct node \*ptr,\*ptrsave,\*suc,\*parsuc;

ptrsave=loc;

ptr=loc->rchild;

while(ptr->lchild!=NULL)

{

ptrsave=ptr;

ptr=ptr->lchild;

}

suc=ptr;

parsuc=ptrsave;

if(suc->lchild==NULL && suc->rchild==NULL)

case\_a(parsuc,suc);

else

case\_b(parsuc,suc);

if(par==NULL)

root=suc;

else

if(loc==par->lchild)

par->lchild=suc;

else

par->rchild=suc;

suc->lchild=loc->lchild;

suc->rchild=loc->rchild;

}

int preorder(struct node \*ptr)

{

if(root==NULL)

{

printf("Tree is empty");

return 0;

}

if(ptr!=NULL)

{

printf("%d ",ptr->info);

preorder(ptr->lchild);

preorder(ptr->rchild);

}

}

void inorder(struct node \*ptr)

{

if(root==NULL)

{

printf("Tree is empty");

return;

}

if(ptr!=NULL)

{

inorder(ptr->lchild);

printf("%d ",ptr->info);

inorder(ptr->rchild);

}

}

void postorder(struct node \*ptr)

{

if(root==NULL)

{

printf("Tree is empty");

return;

}

if(ptr!=NULL)

{

postorder(ptr->lchild);

postorder(ptr->rchild);

printf("%d ",ptr->info);

}

}

void display(struct node \*ptr,int level)

{

int i;

if ( ptr!=NULL )

{

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

printf("\n");

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

printf(" ");

printf("%d", ptr->info);

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

}

}

int main()

{

int choice,num;

root=NULL;

printf("\n\t BINARY SEARCH TREE TRAVERSALS \n");

printf("1.Insert\n");

printf("2.Inorder Traversal\n");

printf("3.Preorder Traversal\n");

printf("4.Postorder Traversal\n");

printf("5.Display\n");

printf("6.Quit\n");

do

{

printf("\nEnter your choice : ");

scanf("%d",&choice);

switch(choice)

{

case 1:

printf("Enter the number to be inserted : ");

scanf("%d",&num);

insert(num);

break;

case 2:

inorder(root);

break;

case 3:

preorder(root);

break;

case 4:

postorder(root);

break;

case 5:

display(root,1);

break;

case 6:

break;

default:

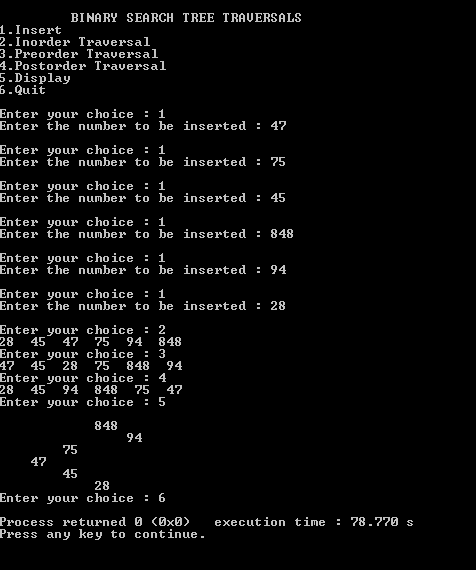
printf("Wrong choice\n");

}

}

while(choice!=6);

}

**OUTPUT:  
**

**Title:** Linear search program

**Date:** 17-04-2020

**Program No:** 10

**Problem Definition:**

* Linear search is known as Sequential search.
* Linear searching is a searching technique to find an item from a list until the particular item not found or list not reached at end.
* We start the searching from 0th index to N-1 index in a sequential manner, if particular item found, returns the position of that item otherwise return failure status or -1.
* Complexity of linear search is O(n).

**PROGRAM FOR LINEAR SEARCH:**

#include <stdio.h>

int main()

{

int array[100], search, c, n;

printf("\n\t\t\t LINEAR SEARCH\n");

printf("Enter number of elements:");

scanf("%d", &n);

printf("Enter %d elements:", n);

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

scanf("%d", &array[c]);

printf("Enter a number to search:");

scanf("%d", &search);

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

{

if (array[c] == search)

{

printf("%d is present at location %d.\n", search, c);

break;

}

}

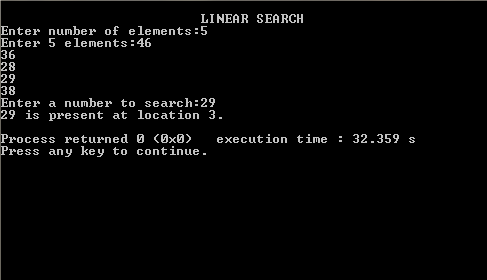
if (c == n)

printf("%d isn't present in the array.\n", search);

return 0;

}

**OUTPUT:**

****

**Title:** Binary search program

**Date:** 17-04-2020

**Program No:** 11

**Problem Definition:**

* It is also known as half-interval search, logarithmic search, or binary chop, is a [search algorithm](https://en.wikipedia.org/wiki/Search_algorithm" \o "Search algorithm) that finds the position of a target value within a [sorted array](https://en.wikipedia.org/wiki/Sorted_array" \o "Sorted array).
* Binary search compares the target value to the middle element of the array. If they are not equal, the half in which the target cannot lie is eliminated and the search continues on the remaining half, again taking the middle element to compare to the target value, and repeating this until the target value is found. If the search ends with the remaining half being empty, the target is not in the array.
* It is special type of search work on sorted list only.
* Complexity of binary search is O{nlogn).

**PROGRAM FOR BINARY SEARCH:**

#include <stdio.h>

void main()

{

int array[10];

int i, j, num, temp, keynum;

int low, mid, high;

printf("\n\t\t\t BINARY SEARCH \n");

printf("Enter the size of array:");

scanf("%d", &num);

printf("Enter the elements:");

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

{

scanf("%d", &array[i]);

}

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

{

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

{

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

{

temp = array[j];

array[j] = array[j + 1];

array[j + 1] = temp;

}

}

}

printf("Sorted array is...\n");

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

{

printf("%d\n", array[i]);

}

printf("Enter the element to be searched:");

scanf("%d", &keynum);

low = 1;

high = num;

do

{

mid = (low + high) / 2;

if (keynum < array[mid])

high = mid - 1;

else if (keynum > array[mid])

low = mid + 1;

} while (keynum != array[mid] && low <= high);

if (keynum == array[mid])

{

printf("%d is found at the location %d",keynum,mid);

}

else

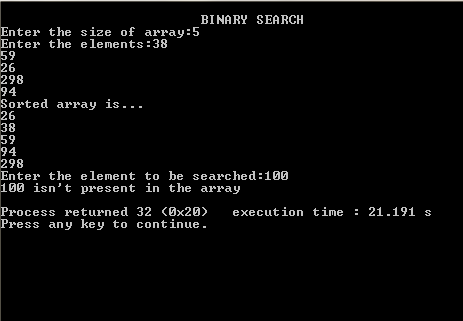
{

printf("%d isn't present in the array \n",keynum);

}

}

**OUTPUT:**



**Title:** Bubble sort program

**Date:** 24-04-2020

**Program No:** 12

**Problem Definition:**

* Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for Eg: an Array with N number of elements.
* Bubble Sort compares all the element one by one and sort them based on their values.
* It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.
* Sorting takes place by stepping through all the data items one by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.
* Complexity of Bubble Sort is O(n).

**PROGRAM FOR BUBBLE SORT :**

#include <stdio.h>

#define MAX 100

int main()

{

int arr[MAX],limit;

int i,j,temp;

printf("\t\t\t\t BUBBLE SORT \n\n");

printf("Enter total number of elements: ");

scanf("%d",&limit);

printf("Enter array elements: \n");

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

{

printf("Enter element %3d: ",i+1);

scanf("%d",&arr[i]);

}

for(i=0; i<(limit-1); i++)

{

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

{

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

{

temp=arr[j];

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

arr[j+1]=temp;

}

}

}

printf("The elements after bubble sorting in ascending order:\n");

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

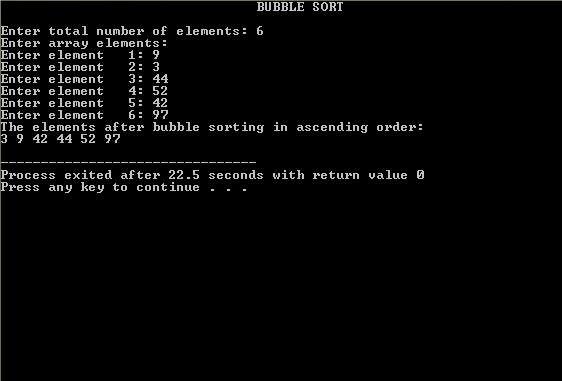
printf("%d ",arr[i]);

printf("\n");

return 0;

}

**OUTPUT:**



**Title:** Insertion Sort program

**Date:** 27-04-2020

**Program No:** 13

**Problem Definition:**

* Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list.
* Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.
* Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array position, it checks the value there against the largest value in the sorted list.
* If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.
* The resulting array after k iterations has the property where the first k + 1 entries are sorted.
* Complexity of Insertion sort is O(n).

**PROGRAM FOR INSERTION SORT:**

#include<stdio.h>

int main()

{

int data[100],n,temp,i,j;

printf("Enter number of elements to be sorted:");

scanf("%d",&n);

printf("Enter elements: ");

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

scanf("%d",&data[i]);

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

{

temp = data[i];

j = i - 1;

while(temp < data[j] && j>=0)

{

data[j + 1] = data[j];

j = j - 1;

}

data[j + 1]=temp;

}

printf("Sorted array: ");

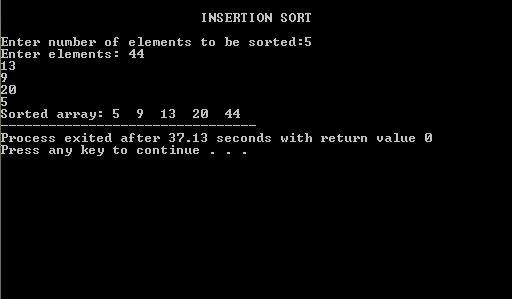
for(i = 0; i < n; i++)

printf("%d ",data[i]);

return 0;

}

**OUTPUT:**



**Title:** Selection SortProgram

**Date:** 30-04-2020

**Program No:** 14

**Problem Definition:**

* The selection sort improves on the bubble sort by making only one exchange for every pass through the list.
* In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location.
* As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place.
* This process continues and requires n−1 passes to sort n items, since the final item must be in place after the (n−1) st pass.
* Complexity of Selection sort is O(n2).

**PROGRAM FOR SELECTION SORT:**

#include <stdio.h>

int main()

{

int array[100], n, pos, temp, i, j;

printf("\n\t\t\t SELECTION SORT \n");

printf("Enter number of elements:");

scanf("%d", &n);

printf("\nEnter the %d values\n", n);

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

{

scanf("%d", &array[i]);

}

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

{

pos = i;

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

{

if (array[pos] > array[j])

pos = j;

}

if (pos != i)

{

temp = array[i];

array[i] = array[pos];

array[pos] = temp;

}

}

printf("Sorted list :\n");

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

{

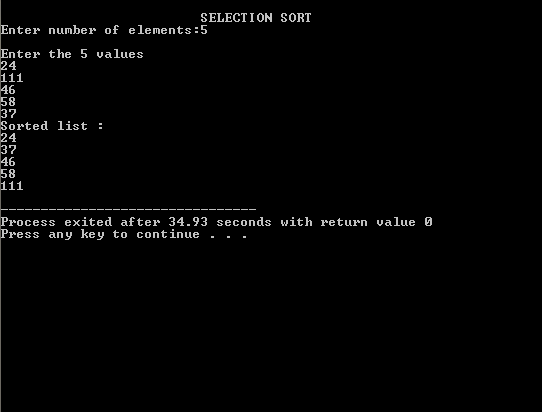
printf("%d\n", array[i]);

}

return 0;

}

**OUTPUT:**



**Title:** Shell Sort program

**Date:** 04-05-2020

**Program No:** 15

**Problem Definition:**

* Shell sort works by comparing elements that are distant rather than adjacent elements in an array.
* Shell sort makes multiple passes through a list and sorts a number of equally sized sets using the insertion sort.
* The distance between comparisons decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared.
* Complexity of shell sort is O(nlogn).

**PROGRAM FOR SHELL SORT:**

#include<stdio.h>

#include<conio.h>

int main()

{

int arr[30];

int i,j,k,tmp,num;

printf("\n\t\t\t SHELL SORT \n");

printf("Enter total no. of elements : ");

scanf("%d", &num);

for(k=0; k<num; k++)

{

printf("\nEnter %d number : ",k+1);

scanf("%d",&arr[k]);

}

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

{

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

{

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

{

if(arr[k+i]>=arr[k])

break;

else

{

tmp=arr[k];

arr[k]=arr[k+i];

arr[k+i]=tmp;

}

}

}

}

printf("\t\*\*\*\* Shell Sorting \*\*\*\*\n");

for(k=0; k<num; k++)

{

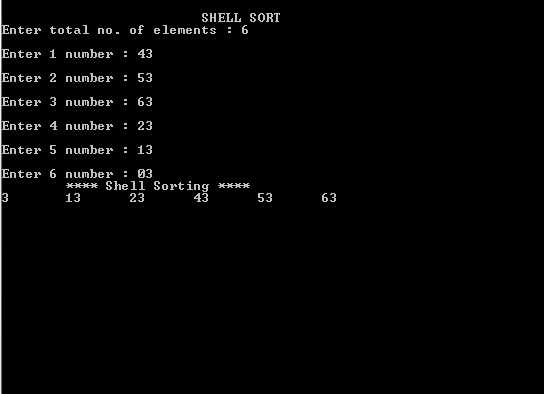
printf("%d\t",arr[k]);

}

getch();

}

**OUTPUT:**



**Title:** Quick Sort program

**Date:** 06-05-2020

**Program No:** 16

**Problem Definition:**

Quicksort is a fast sorting algorithm, which is used not only for educational purposes, but widely applied in practice.

**ALGORITHM:**

The divide-and-conquer strategy is used in quicksort. Below the recursion step is described:

* **Choose a pivot value:** We take the value of the left most element or middle element or right most element as pivot value.
* **Partition:** Rearrange elements in such a way, that all elements which are lesser than the pivot go to the left part of the array and all elements greater than the pivot, go to the right part of the array. Values equal to the pivot can stay in any part of the array. Notice, that array may be divided in non-equal parts.
* **Sort both parts:** Apply quicksort algorithm recursively to the left and the right parts.
* Complexity of quick sort is O(nlogn).

**PROGRAM FOR QUICK SORT:**

#include<stdio.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];

printf("\n\t\t\t QUICK SORT \n");

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: ");

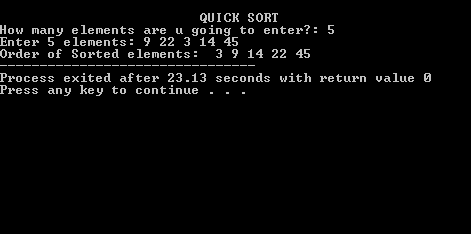
for(i=0;i<count;i++)

printf(" %d",number[i]);

return 0;

}

**OUTPUT:**



**Title:** Heap Sort Program

**Date:** 12-05-2020

**Program No:** 17

**Problem Definition:**

* Heap sort uses the property of Heaps to sort an array. The Heap data structure is an array object that can be viewed as a complete and balanced binary tree.
* Min (Max)-Heap has a property that for every node other than the root, the value of the node is at least (at most) the value of its parent.
* Thus, the smallest (largest) element in a heap is stored at the root, and the subtrees rooted at a node contain larger (smaller) values than does the node itself.
* Complexity of Heap Sort is O(nlogn).

**PROGRAM FOR HEAP SORT:**

#include<stdio.h>

void create(int []);

void down\_adjust(int [],int);

int main()

{

int heap[30],n,i,last,temp;

printf("\t\t\t HEAP SORT \n");

printf("Enter number of elements:");

scanf("%d",&n);

printf("\nEnter elements:");

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

scanf("%d\n",&heap[i]);

heap[0]=n;

create(heap);

while(heap[0] > 1)

{

last=heap[0];

temp=heap[1];

heap[1]=heap[last];

heap[last]=temp;

heap[0]--;

down\_adjust(heap,1);

}

printf("\nArray after sorting:\n");

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

printf("%d\t",heap[i]);

return 0;

}

void create(int heap[])

{

int i,n;

n=heap[0];

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

down\_adjust(heap,i);

}

void down\_adjust(int heap[],int i)

{

int j,temp,n,flag=1;

n=heap[0];

while(2\*i<=n && flag==1)

{

j=2\*i;

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

j=j+1;

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

flag=0;

else

{

temp=heap[i];

heap[i]=heap[j];

heap[j]=temp;

i=j;

}

}

}

**OUTPUT:**

