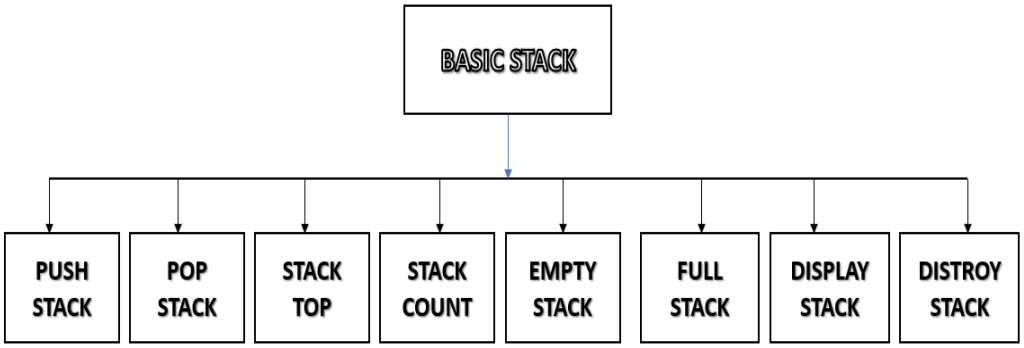
Programs on Stacks

ASSIGNMENT 1

Write a generic C program to implement Stack ADT (stack.h) for the following stack operations: i) create stack ii) push stack iii) pop stack iv) stack top v) empty stack vi) full stack vii) destroy stack viii) display stack. Write an application program (stackmain.c) to create an integer stack and perform all the Stack operations defined in the Stack ADT (stack.h)



stack.h

//Stack ADT type definitions

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
typedef struct node
{
   void* dataptr;
   struct node* link;
}stack_node;
typedef struct
{
   int count;
  stack_node* top;
}stack_head;
/*Prototype declarations*/
stack_head* createstack(void);
bool pushstack(stack_head* stack,void* datainptr);
void* popstack(stack_head* stack);
void* stacktop(stack_head* stack);
```

```
bool emptystack(stack_head* stack);
bool fullstack(stack_head* stack);
int stackcount(stack_head* stack);
stack_head* destroystack(stack_head* stack);
void displaystack(stack_head* stack);

/*   ==========create stack==========
Creates an empty stack
 Pre:Nothing;
 Post:Returns pointer to a Null stack -or- Null if Overflow*/
stack_head* createstack(void)
{
   stack_head* stack;
   stack=(stack_head*)malloc(sizeof(stack_head));
   if(stack)
   {
      stack->count = 0;
      stack->top   = NULL;
   }//if
  return stack;
}//create stack

/*     ==========push stack ==========
Function pushes an item onto the stack.
     Pre: stack is pointer to the stack;  datainptr pointer to data to be inserted
     Post:data inserted into stack
     return true if successful or  false if underflow
*/
bool pushstack(stack_head* stack,void* datainptr)
{
   //local definitions
    stack_node* newptr;
   //statements
   newptr=(stack_node*)malloc(sizeof(stack_node));
   if(!newptr)   return false;
   newptr->dataptr = datainptr;
   newptr->link    = stack->top;
   stack->top      = newptr;
   (stack->count)++;
   return true;
```

```
} //push stack
/*  ==========**pop stack**==========
Function pops item on the top of the stack.
   Pre: stack is a pointer to a stack; Post:returns pointer to user data if successful  null if
   Underflow
*/
void* popstack(stack_head* stack)
{
   void* dataoutptr;
   stack_node* temp;
//statements
   if(stack->count ==0)
  dataoutptr=NULL;
   else
   {
      temp      = stack->top;
     dataoutptr= stack->top->dataptr;
      stack->top= stack->top->link;
      free(temp);
      (stack->count)--;
   }
 return dataoutptr;
} //pop stack
/*     ==========**stack top**==========
Retrives data from top of stack without changing the stack.
     Pre: Stack is the pointer to the stack
     Post:Returns data pointer to the stack if successful  null pointer if stack empty
*/
void* stacktop(stack_head* stack)
{
 //statements
   if(stack->count == 0) return NULL;
   else
      return (stack->top->dataptr);
} //stack top
/* ==========empty stack==========
      Function determines if stack is empty.
      pre: stack is pointer to the stack;
      post:returns true if empty;false if data in the stack.
*/
```

```
bool emptystack(stack_head* stack)
{
//statements
   return(stack->count  ==   0);
}   //empty stack
/*==============full stack====================
 function determines if stack is full. full is defined as heap full.
    pre: stack is pointer to stack
    return true if heap full ;false if heap has room
*/
bool fullstack(stack_head* stack)
{
   //local definitions
stack_node* temp;
   //statements
   temp=(stack_node*)malloc(sizeof(stack_node));
   if(temp)
   {
      free(temp);
      return false;
   }// if
   //malloc failed
   return true;
}   //full stack
/*  ================stack count=====================
returns number of elements in a stack
     pre: stack is the pointer to the stack;   post:count returned
*/

int stackcount(stack_head* stack)
{
 //statements
  return (stack->count);
}  //stack count
/*  ==============destroy stack=====================
this function releases all nodes to the heap.
   pre: stack;   post:return null pointer
*/
stack_head* destroystack(stack_head* stack)
{
```

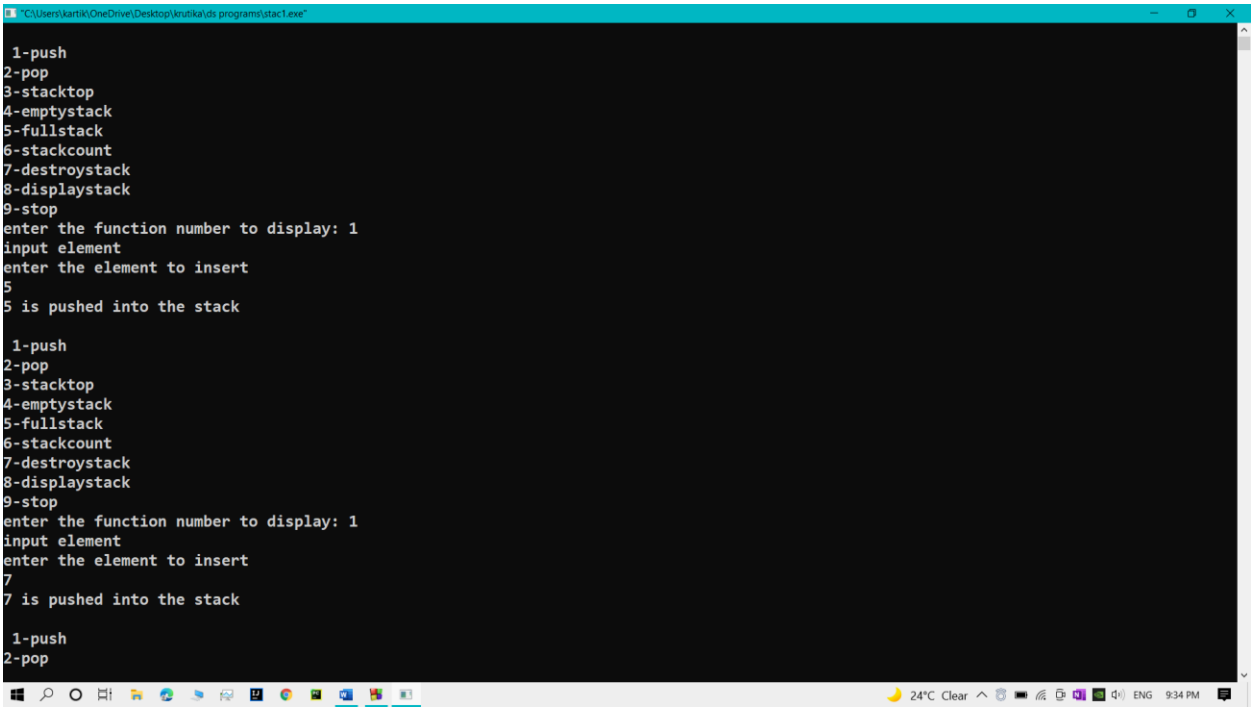
```
/local definitions
stack_node* temp;
//statements
  if(stack)
  {
//delete all nodes in the stack
   while(stack->top != NULL)
   {
//delete data entry
     free(stack->top->dataptr);
     temp=stack->top;
     stack->top=stack->top->link;
     free(temp);
   } //While
  //stack now empty,destroy stack head node.
  free(stack);
  } //if stack
 return NULL;
}   //destroy stack
/* ====================display stack==================
  this function display the data in the stack
  pre: a stack; post:data has been displayed
*/
void displaystack(stack_head* stack)
{
   //local declarations
stack_node* temp;
//statements
   if(stack->count==0)
     printf("stack is empty");
   else
   {
      printf("contents of stack are:\n");
      temp=stack->top;
      while(temp)
      {
         printf("%d",(*(int*)(temp->dataptr)));
         temp=temp->link;
      }
   } }  //display stack
```

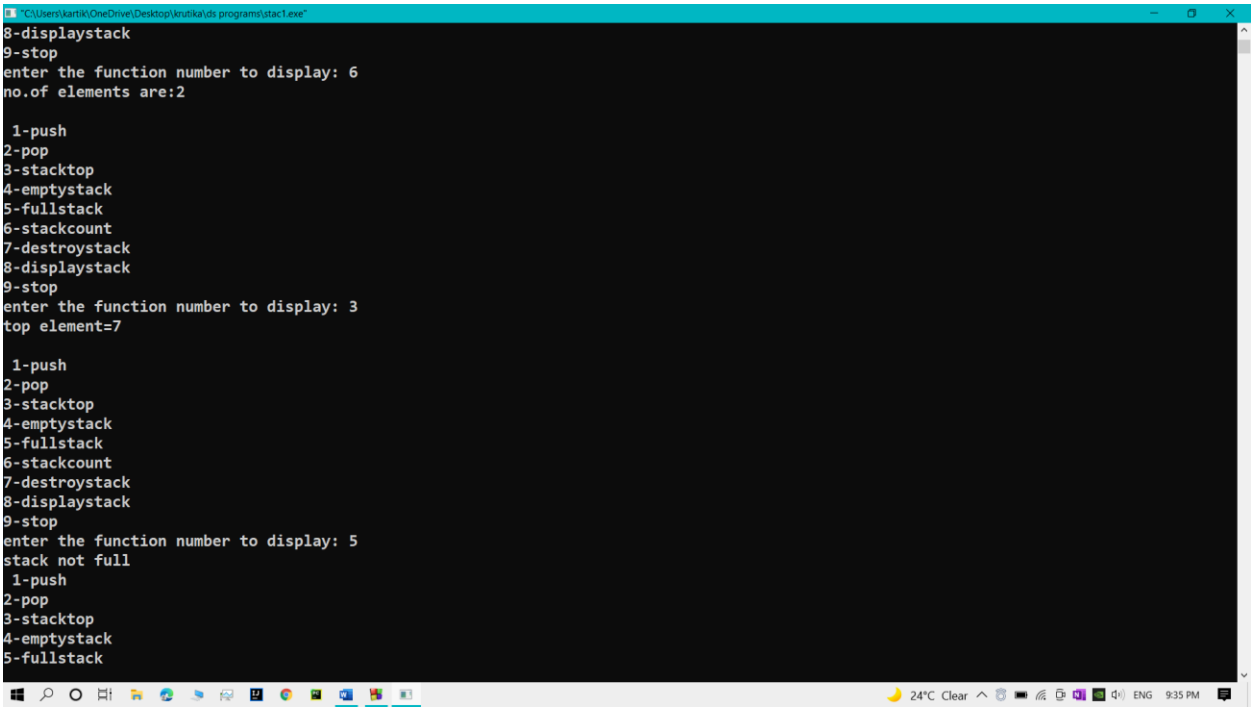
stackmain.c

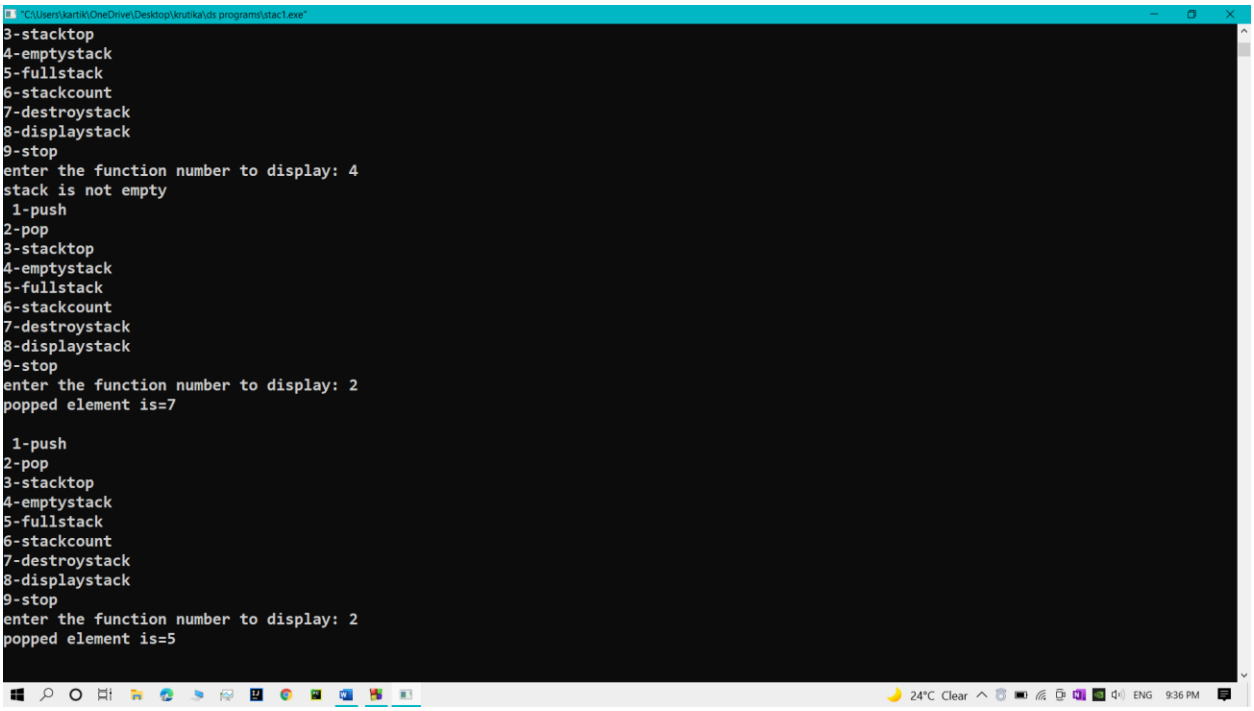
```
/* Performs stack operations on integer stack using stack ADT*/
#include<stdio.h>
#include<stdlib.h>
#include"stack.h"
int main()
{
   stack_head* stack=NULL;
   int* pele;
   int ch;
   int c;
stack=createstack();
stack_head* t;
   while(true)
   {
     printf("\n 1-push\n");
     printf("2-pop\n");
     printf("3-stacktop\n");
     printf("4-emptystack\n");
     printf("5-fullstack\n");
     printf("6-stackcount\n");
     printf("7-destroystack\n");
     printf("8-displaystack\n");
     printf("9-stop\n");
     printf("enter the function number to display: ");
     scanf("%d",&ch);
      switch(ch)
      {

         case 1:printf("input element\n");
                  pele=(int*)malloc(sizeof(int));
                  printf("enter the element to insert\n");
                  scanf("%d",pele);
                  if(pushstack(stack,pele))
                     printf("%d is pushed into the stack\n",*pele);
                 else
                     printf("%d could not be pushed into stack",*pele);
                   break;
         case 2:pele=(int*)popstack(stack);
```

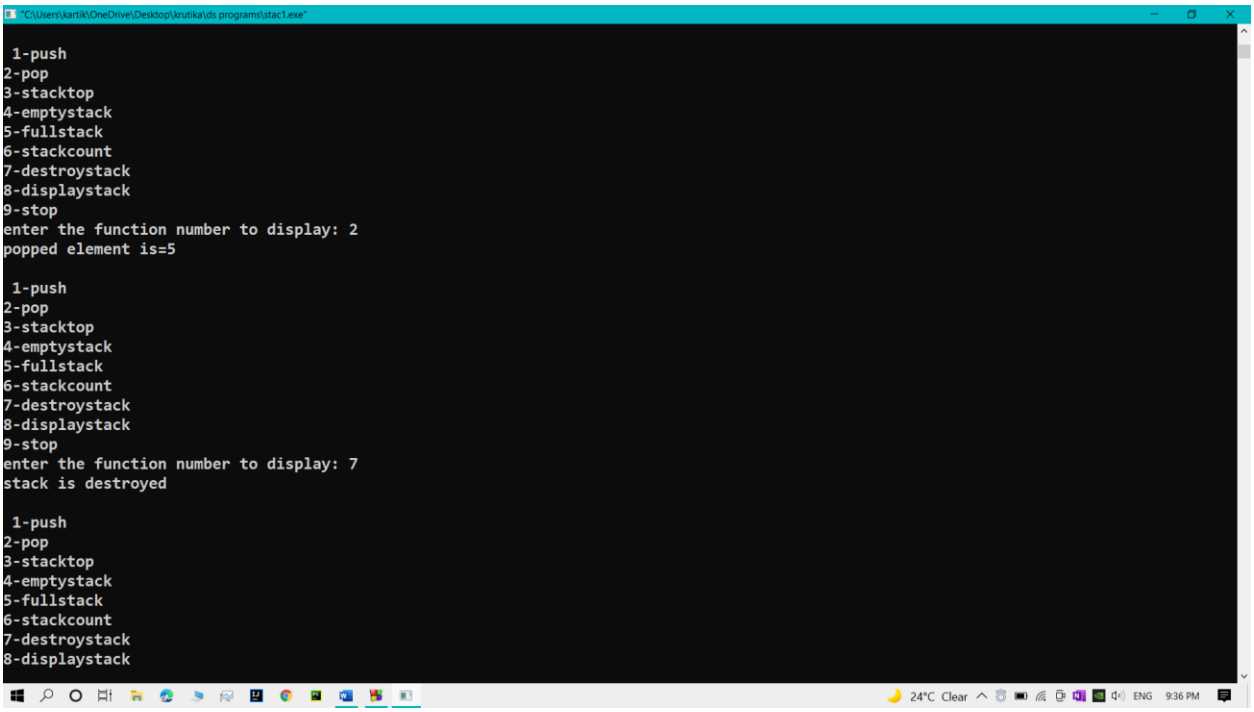
```
                if(pele)
                  printf("popped element is=%d\n",*pele);
                else
                  printf("could not pop element",*pele);
                 break;
        case 3:pele=(int*)stacktop(stack);
                if(pele)
                  printf("top element=%d\n",*pele);
               else
                printf("could not seek");
               break;
        case 4:if(emptystack(stack))
                  printf("stack is empty\n");
                else
                  printf("stack is not empty");
                break;
        case 5:if(fullstack(stack))
                  printf("stack is full\n");
                else
                  printf("stack not full");
                break;
        case 6:c=stackcount(stack);
                printf("no.of elements are:%d\n",c);
                break;
        case 7:t=destroystack(stack);
                if(t==NULL)
                  printf("stack is destroyed\n");
               else
                  printf("stack not destroyed\n");
              break;
        case 8:displaystack(stack);
               break;
              default:exit(0);
      }
}
return 0;
}
```

OUTPUT:


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\stac1.exe"
1-push
2-pop
3-stacktop
4-emptystack
5-fullstack
6-stackcount
7-destroystack
8-displaystack
9-stop
enter the function number to display: 1
input element
enter the element to insert
5
5 is pushed into the stack
1-push
2-pop
3-stacktop
4-emptystack
5-fullstack
6-stackcount
7-destroystack
8-displaystack
9-stop
enter the function number to display: 1
input element
enter the element to insert
7
7 is pushed into the stack
1-push
2-pop
24°C Clear
ENG
9:34 PM

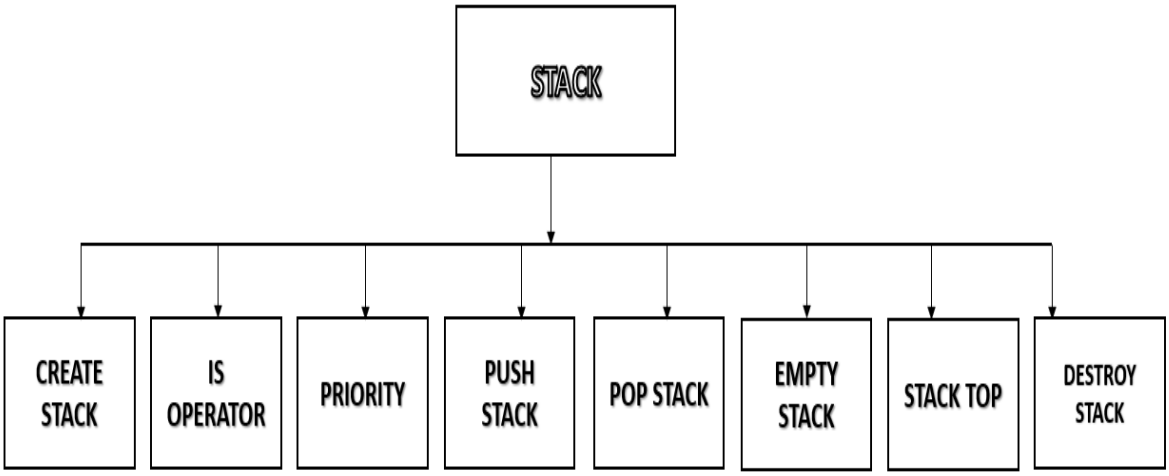

"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\stac1.exe"
8-displaystack
9-stop
enter the function number to display: 6
no.of elements are:2
1-push
2-pop
3-stacktop
4-emptystack
5-fullstack
6-stackcount
7-destroystack
8-displaystack
9-stop
enter the function number to display: 3
top element=7
1-push
2-pop
3-stacktop
4-emptystack
5-fullstack
6-stackcount
7-destroystack
8-displaystack
9-stop
enter the function number to display: 5
stack not full
1-push
2-pop
3-stacktop
4-emptystack
5-fullstack
24°C Clear
ENG
9:35 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\stac1.exe"
3-stacktop
4-emptystack
5-fullstack
6-stackcount
7-destroystack
8-displaystack
9-stop
enter the function number to display: 4
stack is not empty
1-push
2-pop
3-stacktop
4-emptystack
5-fullstack
6-stackcount
7-destroystack
8-displaystack
9-stop
enter the function number to display: 2
popped element is=7
1-push
2-pop
3-stacktop
4-emptystack
5-fullstack
6-stackcount
7-destroystack
8-displaystack
9-stop
enter the function number to display: 2
popped element is=5
24°C Clear
ENG
9:36 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\stac1.exe"
1-push
2-pop
3-stacktop
4-emptystack
5-fullstack
6-stackcount
7-destroystack
8-displaystack
9-stop
enter the function number to display: 2
popped element is=5
1-push
2-pop
3-stacktop
4-emptystack
5-fullstack
6-stackcount
7-destroystack
8-displaystack
9-stop
enter the function number to display: 7
stack is destroyed
1-push
2-pop
3-stacktop
4-emptystack
5-fullstack
6-stackcount
7-destroystack
8-displaystack
24°C Clear
ENG
9:36 PM

ASSIGNMENT 2

Write an application program (infixtopostfic.c) to convert the given infix expression into postfix expression using Stack ADT



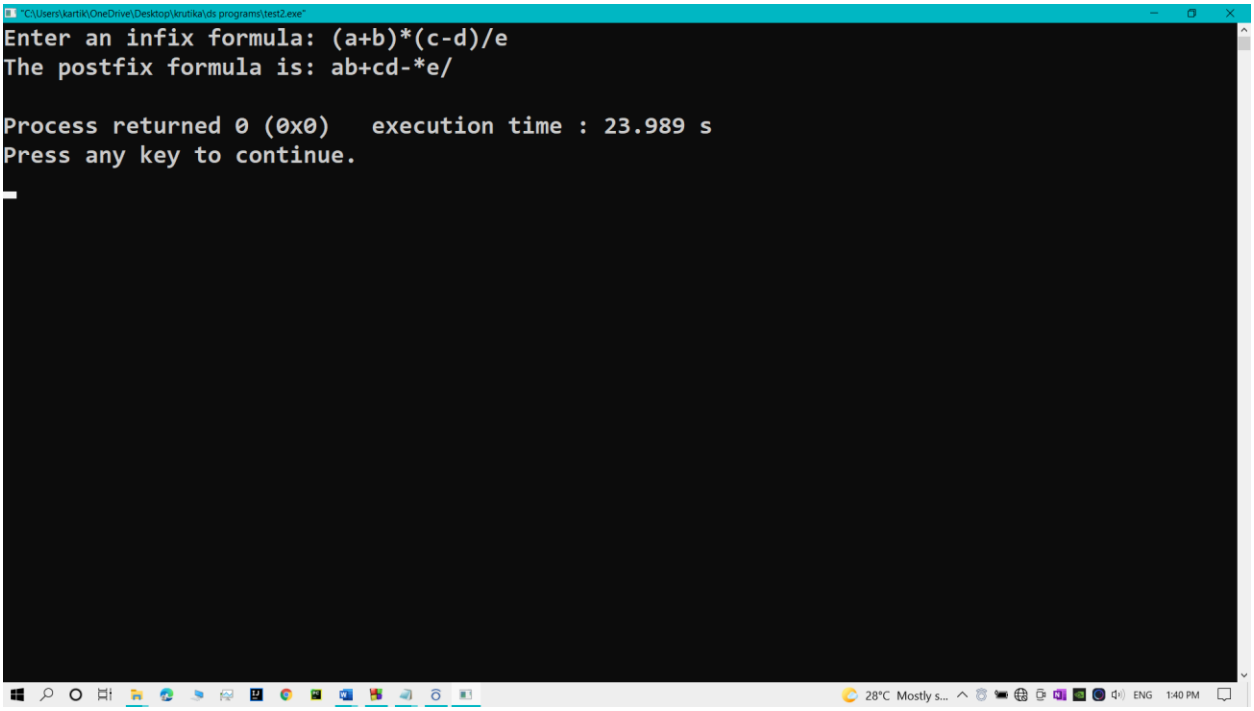

```
/* Converts the given infix expression into postfix expression using stack ADT*/
#include <stdio.h>
#include <string.h>
#include "stack.h"
// Prototype Declarations
int  priority (char token);
bool isOperator (char token);
int main (void)
{
   // Local Definitions
         char   postfix [80] = {0};
         char   temp [2] = {0};
         char   token;
         char*  dataPtr;
         stack_head* stack;
// Statements
// Create Stack
 stack = createstack();
```

```
 // read infix formula and parse char by char
printf("Enter an infix formula: ");
while((token = getchar ())!= '\n')
{
        if (token == '(')
        {
                dataPtr  = (char*) malloc (sizeof(char));
                *dataPtr = token;
                pushstack (stack, dataPtr);
        }// if
        else if (token == ')')
                {
                    dataPtr = (char*)popstack (stack);
                     while (*dataPtr != '(')
                        {
                                temp [0]= *dataPtr;
                                strcat (postfix , temp);
                                dataPtr = (char*)popstack (stack);
                         }// while
                  }     // else if

         else if (isOperator (token))
                {
                // test priority of token at stack top
                dataPtr =(char*)stacktop(stack);
                while (!emptystack(stack)&& priority(token) <= priority(*dataPtr))
                {
                 dataPtr  = (char*)popstack (stack);
                 temp [0] = *dataPtr;
                strcat (postfix , temp);
                dataPtr  = (char*)stacktop (stack);
```

```
                }
                dataPtr  = (char*) malloc (sizeof (char));
               *dataPtr = token;
                pushstack (stack , dataPtr);
      }
     else
       {
         temp[0]= token;
         strcat (postfix , temp);
       }
}
// Infix formula empty. Pop stack to postfix
while (!emptystack (stack))
   {
    dataPtr =(char*)popstack(stack);
    temp[0] = *dataPtr;
   strcat (postfix , temp);
   }
printf ("The postfix formula is: ");
puts (postfix);
destroystack (stack);
 return 0;
}// main
/* ================== priority ====================
Determine priority of operator.
Pre: token is a valid operator
Post: token priority returned
*/
int priority (char token)
{
if (token == '*' || token == '/')
return 2;
if (token == '+' || token == '-')
return 1;
return 0;
}
/* ================== isOperator ====================
Determine if token is an operator.Pre token is a valid operator. Post return true if operator; false
if not
*/
bool isOperator (char token)
{
if (token == '*'    || token == '/'    || token == '+'    || token == '-')
return true;
```

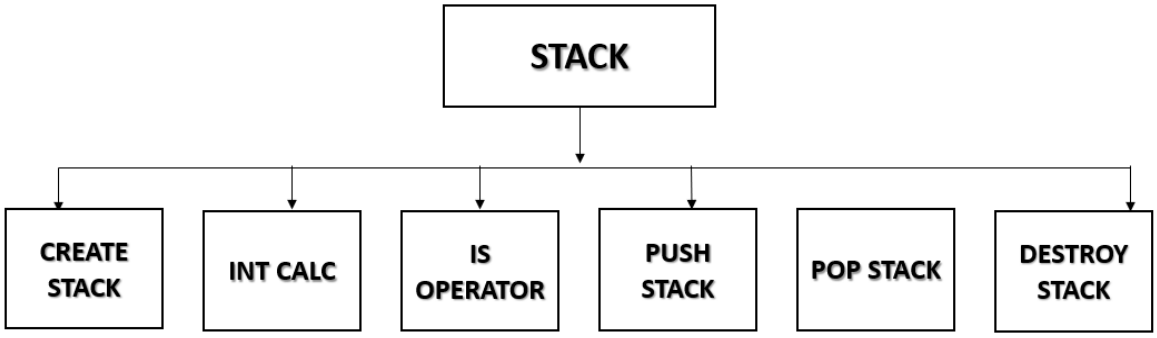
```
return false;
}
```

OUTPUT:



ASSIGNMENT 3

Write an application program to evaluate post fix expression (postfix.c) using Stack ADT.



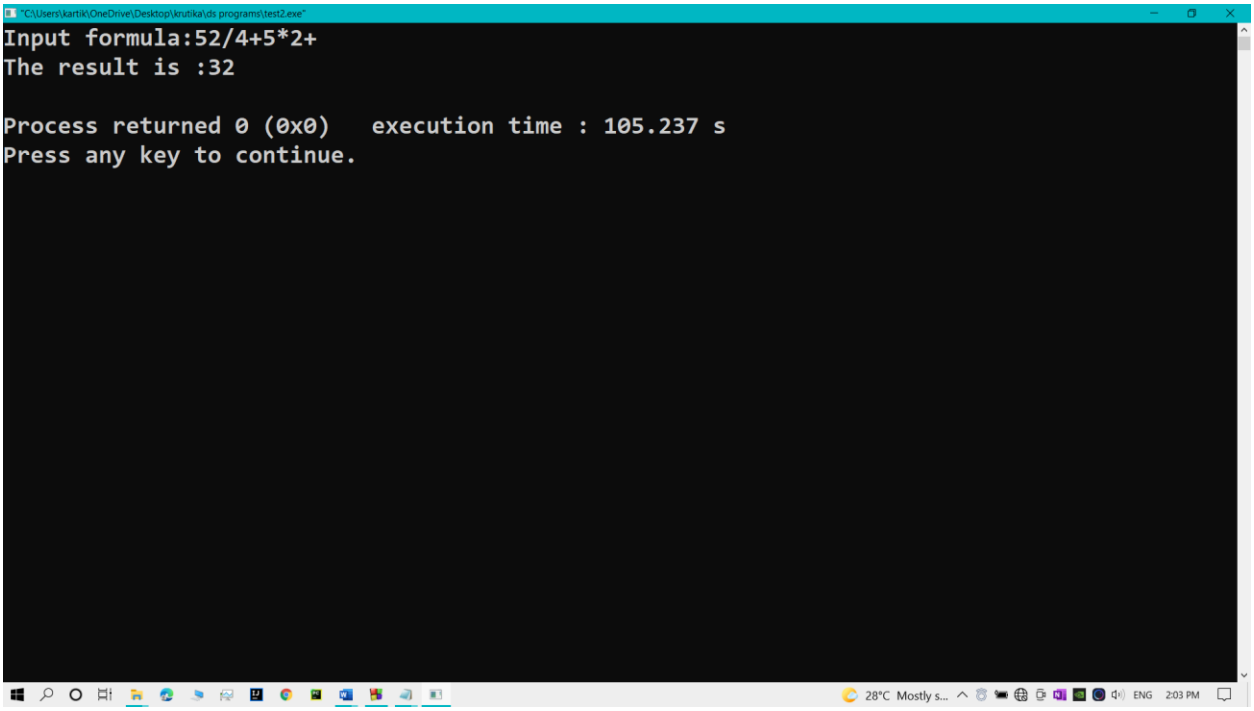
/* This program evaluates a postfix expression using stack ADT andreturns its value */

```
#include<stdio.h>
#include<stdlib.h>
#include"stack.h"
// Prototype Declarations
bool isOperator (char token);
int calc (int operand1,int oper,int operand2);
int main(void)
{
   // Local Definitions
        char token;
        int operand1;
        int operand2;
        int value;
        int* dataptr;
        stack_head* stack;
        // Statements
// Create Stack
        stack = createstack();
        printf("Input formula:");
        while((token = getchar())!='\n')
        {
                if(!isOperator(token))
                {
                        dataptr = (int*)malloc(sizeof(int));
                        *dataptr = atoi(&token);
                        pushstack(stack,dataptr);
```

```
                }
        else
        {
                dataptr = (int*)popstack(stack);
                operand2=*dataptr;
                dataptr=(int*)popstack(stack);
                operand1=*dataptr;
                value=calc(operand1,token,operand2);
                dataptr=(int*)malloc(sizeof(int));
                *dataptr=value;
                pushstack(stack,dataptr);
        }
}
// The final result is in stack. Pop it print it
dataptr=(int*)popstack(stack);
value= *dataptr;
printf("The result is :%d\n",value);

destroystack(stack);
return 0;
}// main
/* ==================== isOperator ==================
Validate operator.
Pre token is operator to be validated
Post return true if valid, false if not
*/
bool isOperator(char token)
{
        if(token =='*'
        || token =='/'
        || token =='+'
        || token =='-')
           return true;
   return false;
}// isOperator
/* ==================== calc ==================
Given two values and operator, determine value of formula.
Pre: operand1 and operand2 are values; oper is the operator to be used
Post: return result of calculation
*/
```

```
int calc(int operand1, int oper,int operand2)
{
        int result;
        switch(oper)
        {
                case '+':result=operand1 + operand2;
                      break;
                case '-':result=operand1 - operand2;
                       break;
                case '*':result=operand1 * operand2;
                       break;
                case '/':result=operand1 / operand2;
                       break;
        }
        return result;
}// calc
```

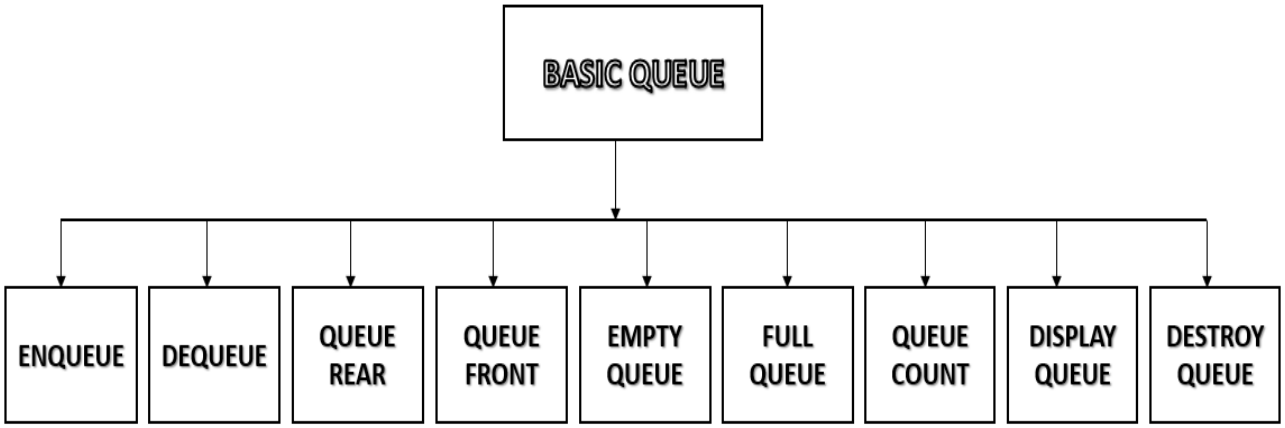
OUTPUT:



Programs on Queue

ASSIGNMENT 4

Write a generic C program to implement Queue ADT (queue.h) for the following queue operations: i) create queue ii) enqueue iii) dequeue iv) queue front v) queue rearvi)emptyqueue vi) full queue viii) queue count ix) destroy queue x) display queue. Write an application program (queuemain.c) to create queue of floating point numbers and perform all the queue operations defined in the Queue ADT.



queue.h

```
#include <stdio.h>
#include <stdlib.h>
#include<stdbool.h>
// queue ADT type definitions
typedef struct node
{
   void* dataptr;
   struct node* next;
}queue_node;
typedef struct
{
queue_node* front;
queue_node* rear;
   int count;
}queue;
```

```
/* Prototype declarations*/
queue* createqueue(void);
bool enqueue(queue* qp,void* itemptr);
bool dequeue(queue* qp,void** itemptr);
bool queuefront(queue* qp,void** itemptr);
bool queuerear(queue* qp,void** itemptr);
bool emptyqueue(queue* qp);
bool fullqueue(queue* qp);
int queuecount(queue* qp);
queue* destroyqueue(queue* qp);
void displayqueue(queue* qp);

/*================= createQueue ================
Allocates memory for a queue head node from dynamicmemory and returns its address to the
caller.
Pre:nothing; Post: head has been allocated and initialized
Return head if successful; null if overflow
*/
queue* createqueue(void)
{
//Local Definitions
   queue* qp;
   //Statements
  qp=(queue*)malloc(sizeof(queue));
   if(qp)
   {
     qp->front=NULL;
     qp->rear=NULL;
     qp->count=0;
   }// if
  return(qp);
}// createQueue
/*================= enqueue ================
This algorithm inserts data into a queue.
Pre:queue has been created; Post :data have been inserted
Return true if successful, false if overflow
*/
bool enqueue(queue* qp,void* itemptr)
{
   //Local Definitions
```

```
    queue_node* newptr;
   //Statements
  newptr=(queue_node*)malloc(sizeof(queue_node));
   if(!newptr)   return false;
   newptr->dataptr=  itemptr;
   newptr->next=     NULL;
   if(qp->count==0)
      // Inserting into null queue
     qp->front=   newptr;
   else
     qp->rear->next= newptr;
   (qp->count)++;
   qp->rear=newptr;
   return(true);
}// enqueue
```

/*================= dequeue ================

deletes a node from the queue.

Pre:queue has been created; Post: Data pointer to queue front returned and front element deleted and recycled.

Return true if successful; false if underflow*/

```
bool dequeue(queue* qp,void** itemptr)
{
   //Local Definitions
  queue_node* deleteloc;
   //Statements
   if(!qp->count)
      return false;
   *itemptr=qp->front->dataptr;
   deleteloc=qp->front;
   // Deleting only item in queue
   if(qp->count==1)
     qp->rear=qp->front=NULL;
   else
     qp->front=qp->front->next;
   (qp->count)--;
   free(deleteloc);
   return true;
}// dequeue
```

/*================= queueFront ================

retrieves data at front of thequeue without changing the queue contents.

```
Pre :qp is pointer to an initialized queue;Post: itemptr passed back to caller
Return true if successful; false if underflow
*/
bool queuefront(queue* qp,void** itemptr)
{
   //Statements
   if(!qp->count)
      return false;
   else
   {
    *itemptr=qp->front->dataptr;
    return(true);
   }
}// queueFront

/*================== queueRear ================
Retrieves data at the rear of the queuewithout changing the queue contents.
Pre:qp is pointer to initialized queue; Post: data passed back to caller
Return true if successful; false if underflow
*/
bool queuerear(queue* qp,void** itemptr)
{
   //Statements
   if(!qp->count)
      return false;
   else
   {
    *itemptr=qp->rear->dataptr;
    return(true);
   }
}// queueRear
/*================== emptyQueue ================
checks to see if queue is empty.
Pre:qp is a pointer to a queue head node
Return true if empty; false if queue has data
*/
bool emptyqueue(queue* qp)
{
   return(qp->count  ==0);
}// emptyQueue
```

```
/*================== fullQueue =================
checks to see if queue is full. Itis full if memory cannot be allocated for next node.
Pre:qp is a pointer to a queue head node
Return true if full; false if room for a node
*/
bool fullqueue(queue* qp)
{
   //Local Definitions
queue_node* temp;
   //Statements
   temp=(queue_node*)malloc(sizeof(queue_node));
   if(temp)
   {
      free(temp);
      return false;
   }// if
// Heap full
   else
      return true;
}// fullQueue
/*================== queueCount =================
Returns the number of elements in the queue.
Pre:qp is pointer to the queue head node
Return queue count
*/
int queuecount(queue* qp)
{
   //Statements
   return(qp->count);
}// queueCount
/*================== destroyQueue =================
Deletes all data from a queue and recycles itsmemory, then deletes & recycles queue head
pointer.
Pre :Queue is a valid queue ;Post: All data have been deleted and recycled
Return null pointer
*/
queue* destroyqueue(queue* qp)
{
   //Local Definitions
queue_node* deleteptr;
```

```
   //Statements
 if(qp)
   {
      while(qp->front!=NULL)
      {
         free(qp->front->dataptr);
         deleteptr=qp->front;
         qp->front=qp->front->next;
         free(deleteptr);
      }
      free(qp);
   }
   return NULL;
}// destroyQueue
```

/*================== display queue ================

Displays contents of the queue

Pre: Queue is a valid queue ;Post: All data have been displayed

*/

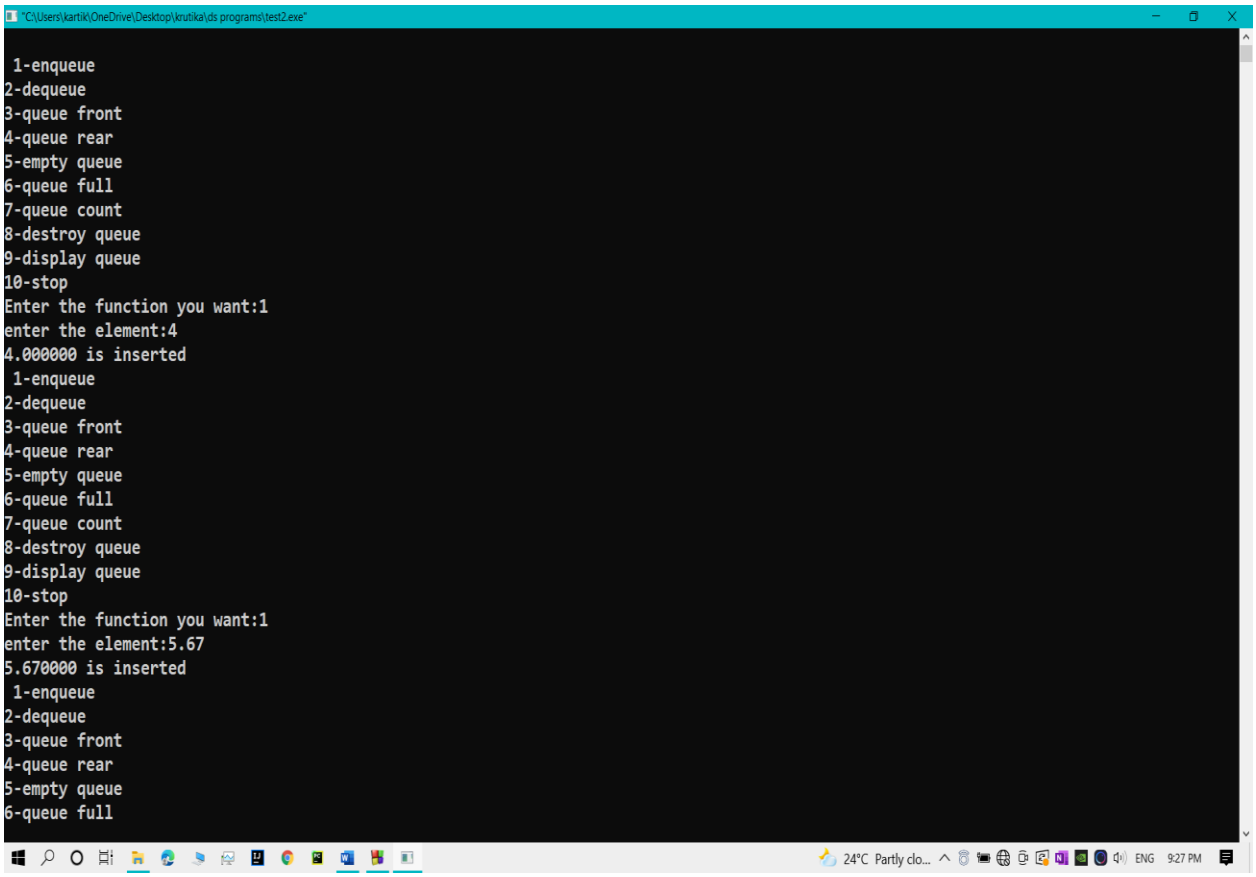
```
void displayqueue(queue* qp)
{
queue_node* t;
   if(!qp->count)
      printf("empty queue");
   else
       t=qp->front;
  printf("contents are:");
  while(t)
    {
       printf("%f",(*(float*)t->dataptr));
       t=t->next;
    }
}
```

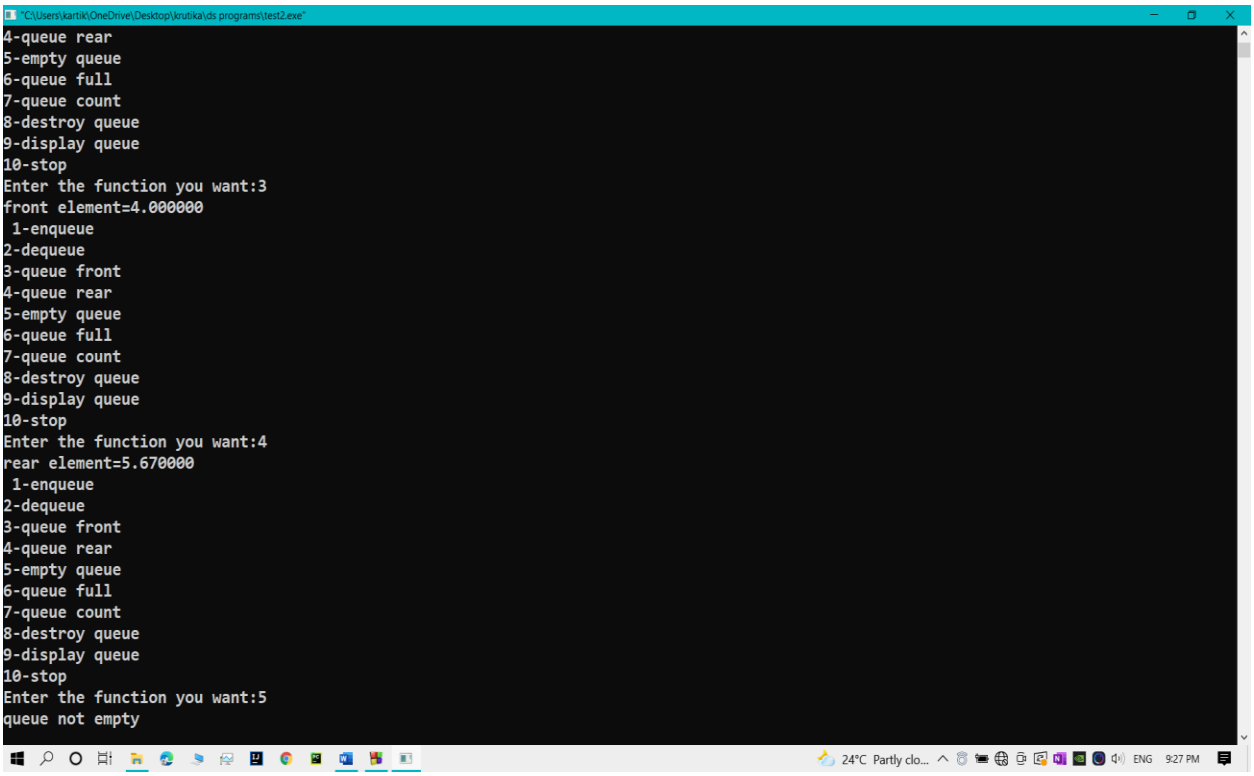
queuemain.c

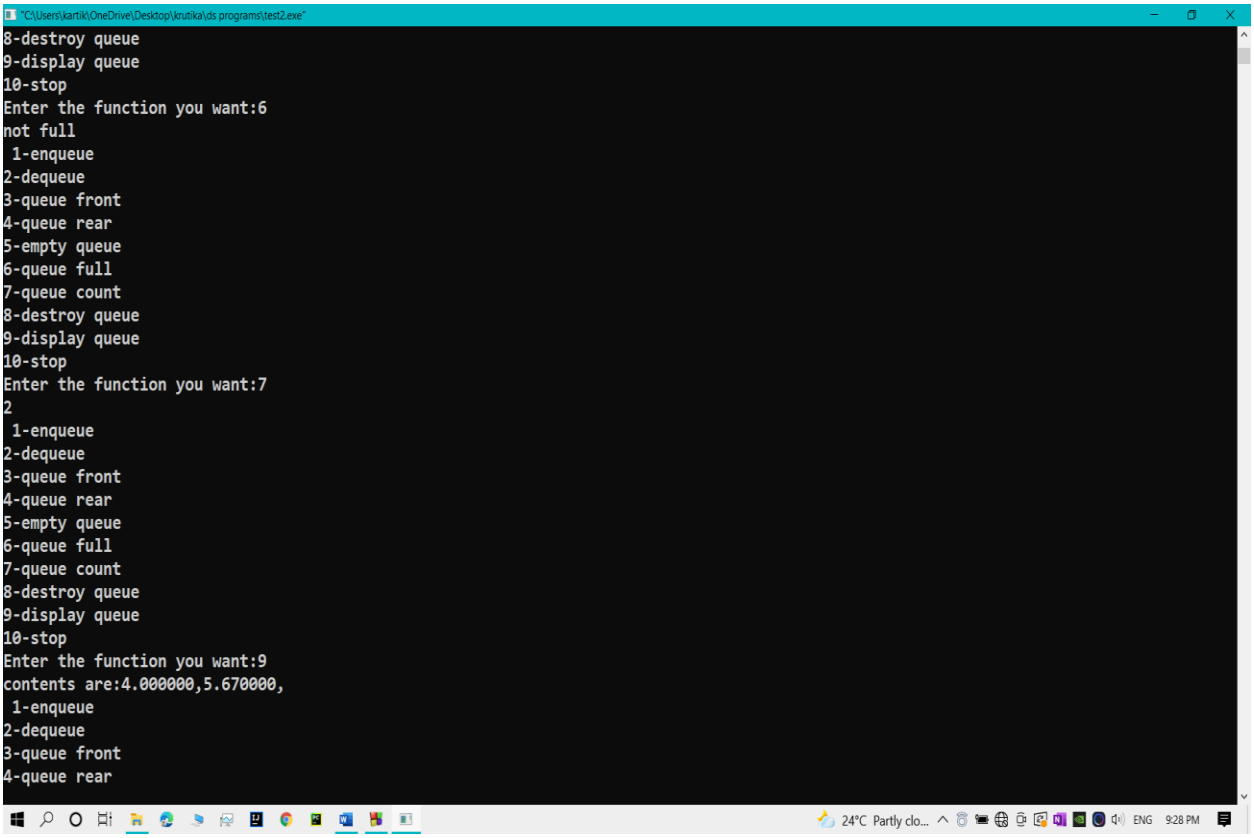
```
/* Performs queue operations on floating point queue using queue ADT*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "queue.h"
int main()
{
   queue* qp;
   qp=createqueue();
   int ch;
   float* dp;
   int c;
   queue* t;
   while(true)
   {
     printf("\n 1-enqueue\n");
     printf("2-dequeue\n");
     printf("3-queue front\n");
     printf("4-queue rear\n");
     printf("5-empty queue\n");
     printf("6-queue full\n");
     printf("7-queue count\n");
     printf("8-destroy queue\n");
     printf("9-display queue\n") ;
     printf("10-stop\n");
    printf("Enter the function you want:");
    scanf("%d",&ch);
    switch(ch)
          {
             case 1:dp=(float*)malloc(sizeof(float));
                    printf("enter the element:");
                    scanf("%f",dp);
                    if(enqueue(qp,dp))
                    printf("%f is inserted",*dp);
                    else
                    printf("%f is not inserted",*dp);
                   break;
             case 2:if(dequeue(qp,(void*)&dp))
                      printf("dequeued element is=%f",*(float*)dp);
```

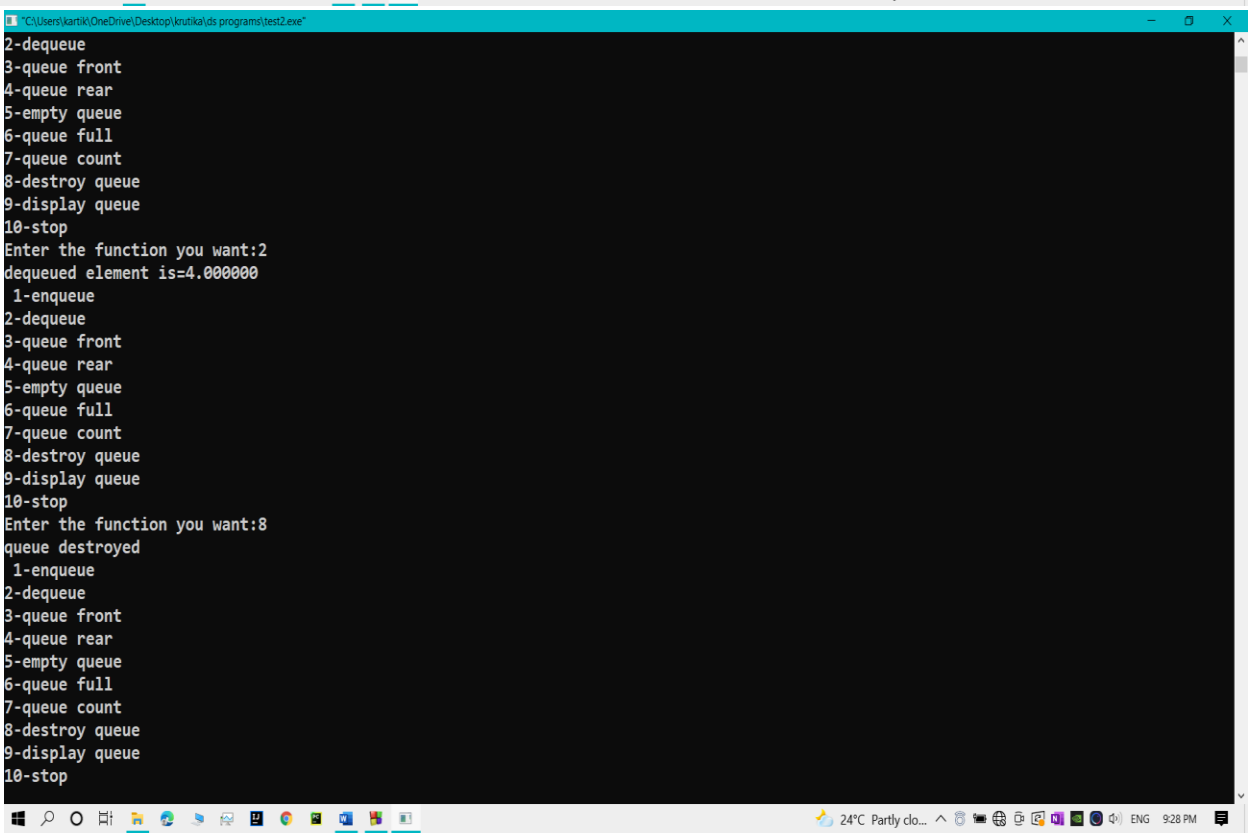
```
                else
                  printf("not dequeued");
                break;
        case 3:if(queuefront(qp,(void*)&dp))
                   printf("front element=%f",(*(float*)dp));
                else
                   printf("queue empty");
                break;
        case 4:if(queuerear(qp,(void*)&dp))
                   printf("rear element=%f",(*(float*)dp));
               else
                  printf("queue empty");
                break;
        case 5:if(emptyqueue(qp))
                  printf("queue empty");
               else
                  printf("queue not empty");
                break;
        case 6:if(fullqueue(qp))
                  printf("queue full");
               else
                  printf("not full");
              break;
        case 7:c=queuecount(qp);
                  printf("%d",c);
                  break;
        case 8:t=destroyqueue(qp);
                   if(t==NULL)
                     printf("queue destroyed");
                  else
                     printf("not destroyed");
             break;
        case 9:displayqueue(qp);
                  break;
       default:exit(0);
      }
}
return 0;}
```

OUTPUT:


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test2.exe"
1-enqueue
2-dequeue
3-queue front
4-queue rear
5-empty queue
6-queue full
7-queue count
8-destroy queue
9-display queue
10-stop
Enter the function you want:1
enter the element:4
4.000000 is inserted
1-enqueue
2-dequeue
3-queue front
4-queue rear
5-empty queue
6-queue full
7-queue count
8-destroy queue
9-display queue
10-stop
Enter the function you want:1
enter the element:5.67
5.670000 is inserted
1-enqueue
2-dequeue
3-queue front
4-queue rear
5-empty queue
6-queue full
24°C Partly clo...
ENG
9:27 PM

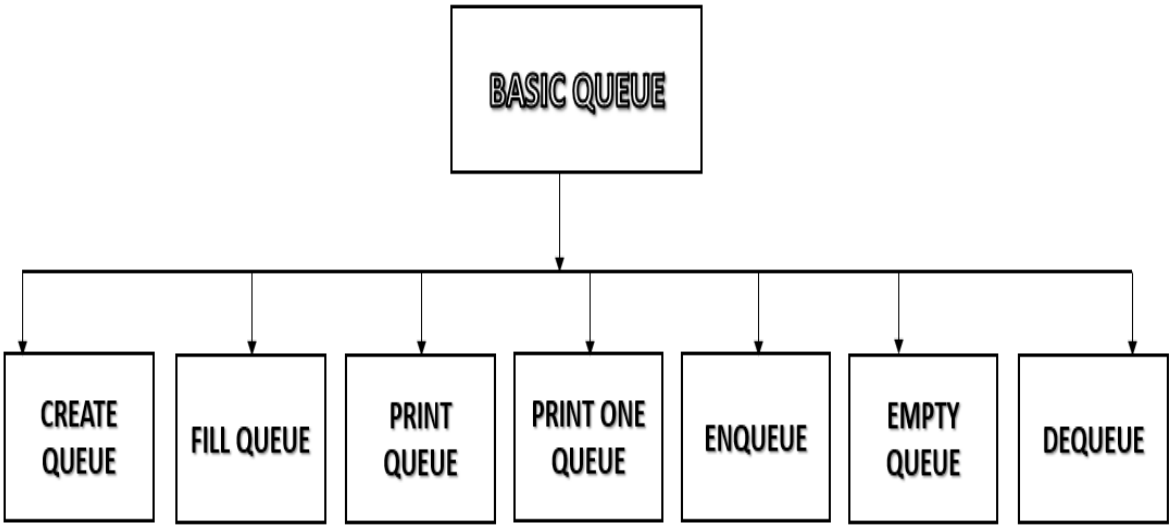

"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test2.exe"
4-queue rear
5-empty queue
6-queue full
7-queue count
8-destroy queue
9-display queue
10-stop
Enter the function you want:3
front element=4.000000
1-enqueue
2-dequeue
3-queue front
4-queue rear
5-empty queue
6-queue full
7-queue count
8-destroy queue
9-display queue
10-stop
Enter the function you want:4
rear element=5.670000
1-enqueue
2-dequeue
3-queue front
4-queue rear
5-empty queue
6-queue full
7-queue count
8-destroy queue
9-display queue
10-stop
Enter the function you want:5
queue not empty
24°C Partly clo...
ENG
9:27 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test2.exe"
8-destroy queue
9-display queue
10-stop
Enter the function you want:6
not full
1-enqueue
2-dequeue
3-queue front
4-queue rear
5-empty queue
6-queue full
7-queue count
8-destroy queue
9-display queue
10-stop
Enter the function you want:7
2
1-enqueue
2-dequeue
3-queue front
4-queue rear
5-empty queue
6-queue full
7-queue count
8-destroy queue
9-display queue
10-stop
Enter the function you want:9
contents are:4.000000,5.670000,
1-enqueue
2-dequeue
3-queue front
4-queue rear


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test2.exe"
2-dequeue
3-queue front
4-queue rear
5-empty queue
6-queue full
7-queue count
8-destroy queue
9-display queue
10-stop
Enter the function you want:2
dequeued element is=4.000000
1-enqueue
2-dequeue
3-queue front
4-queue rear
5-empty queue
6-queue full
7-queue count
8-destroy queue
9-display queue
10-stop
Enter the function you want:8
queue destroyed
1-enqueue
2-dequeue
3-queue front
4-queue rear
5-empty queue
6-queue full
7-queue count
8-destroy queue
9-display queue
10-stop

ASSIGNMENT 5

Write an application program to categorize list of numbers (categorizedata.c) using Queue ADT.



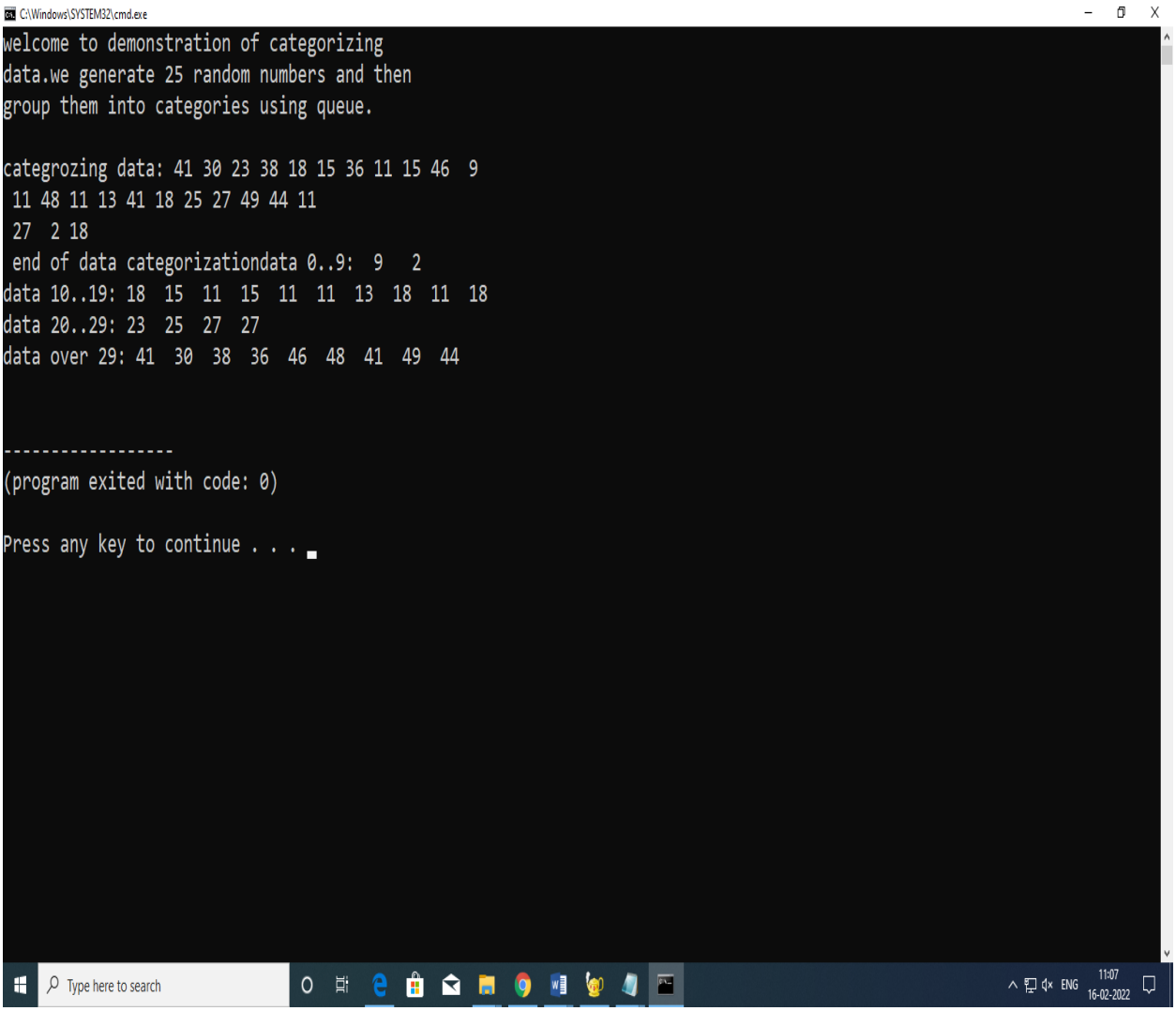

```
/* categorizes list of numbers (categorizedata.c) using Queue  ADT*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "queue.h"
//Prototype Statements
void fillqueues(queue*,queue*,queue*,queue*);
void printqueues(queue*,queue*,queue*,queue*);
void printonequeue(queue* pqueue);
int main(void)
{
   //Local Definitions
         queue* q0to9;
         queue* q10to19;
         queue* q20to29;
         queue* qover29;
         printf("welcome to demonstration of categorizing \n"
               "data.we generate 25 random numbers and then \n"
               "group them into categories using queue.\n\n");
   //Statements
   q0to9=createqueue();
```

```
   q10to19=createqueue();
   q20to29=createqueue();
   qover29=createqueue();
  fillqueues(q0to9,q10to19,q20to29,qover29);
  printqueues(q0to9,q10to19,q20to29,qover29);
   return 0;
}// main

/*================= fillQueues ================
This function generates data using rand() andplaces them in one of four queues.
Pre: All four queues have been created
Post: Queues filled with data
*/
void fillqueues(queue* q0to9,queue* q10to19,queue* q20to29,queue* qover29)
{
   //Local Definitions
int category;
int item;
int* dataptr;
   //Statements
        printf("categrozing data:");
        srand(79);
         for(int i=1;i <=25;i++)
        {
                if(!(dataptr=(int*)malloc(sizeof(int))))
                printf("overflow in fillqueue\a\n"),
                exit(100);
                *dataptr=item=rand()% 51;
                category=item/10;
                printf("%3d",item);
                if(!(i % 11))
        // Start new line when line full
                printf("\n");
                switch(category)
                {
                        case 0:enqueue(q0to9,dataptr);
                                break;
                        case 1:enqueue(q10to19,dataptr);
                                break;
                        case 2:enqueue(q20to29,dataptr);
```

```
                                break;
                        default:enqueue(qover29,dataptr);
                                break;
                }
        }
        printf("\n end of data categorization");
        return ;
}// fillQueues

/*================= printQueues =================
This function prints the data in each of the queues.
Pre: Queues have been filled
Post :Data printed and dequeued
*/
void printqueues(queue* q0to9,queue* q10to19, queue* q20to29,queue* qover29)
{
   //Statements
        printf("data 0..9:");
        printonequeue(q0to9);
        printf("data 10..19:");
        printonequeue(q10to19);
        printf("data 20..29:");
        printonequeue(q20to29);
        printf("data over 29:");
        printonequeue(qover29);
        return ;
}// printQueues
/*================= printOneQueue =================
This function prints the data in one queue,ten entries to a line.
Pre: Queue has been filled
Post :Data deleted and printed. Queue is empty
*/
void printonequeue(queue* pqueue)
      {
        int linecount;
        int* dataptr;
        linecount=0;
        while(!emptyqueue(pqueue))
        {
                dequeue(pqueue,(void*)&dataptr);
```

```
                if(linecount++ >= 10)
                {
                        linecount=1;
                        printf("\n");
                }
                printf("%3d ",*dataptr);
        }
        printf("\n");
return ;
}// printOne Queue
```

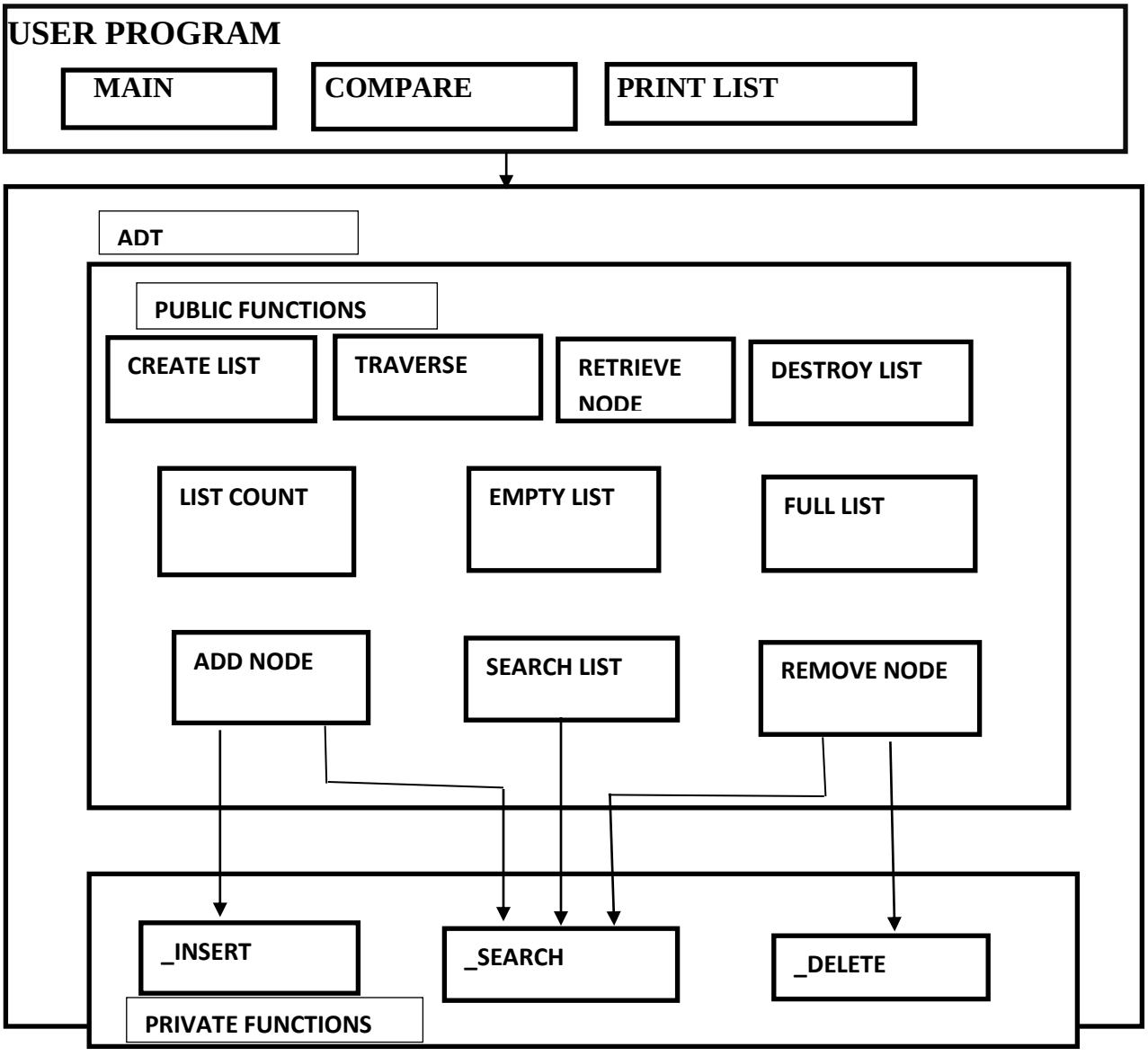
OUTPUT:



Programs on Linked Lists

ASSIGNMENT 6

Write a generic C program to implement singly linked list (ordered list) ADT (singlyll.h) for the following operations: i) create list ii) add node iii) remove node iv) search list v) retrieve node vi) print list vii) empty list viii) full list ix) list count x) destroy list.Write an application program (listmain.c) to create an integer singly linked list and perform all the linked list operations defined in the ADT.



singlyll.h

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
//List ADT Type Defintions
typedef struct node
{
        void* dataptr;
        struct node* link;
}NODE;

typedef struct
{
        int count;
        NODE* pos;
        NODE*head;
        NODE*rear;
        int (*compare)(void* argu1,void* argu2);
}LIST;

//Prototype Declarations
LIST* createlist(int(*compare)(void*argu1,void* argu2));
LIST* destroylist(LIST* list);
int addnode(LIST* plist,void* datainptr);
bool removenode(LIST* plist,void* keyptr,void** dataoutptr);
bool searchlist(LIST* plist,void* pargu,void** pdataout);
static bool retrievenode(LIST* plist,void* pargu,void** dataoutptr);
bool traverse(LIST* plist,intfromwhere,void** dataoutptr);
int listcount(LIST* plist);
bool emptylist(LIST* plist);
bool fulllist(LIST* plist);
static bool _insert(LIST* plist,NODE* ppre,void* datainptr);
static void _delete(LIST* plist,NODE* ppre,NODE* ploc,void** dataoutptr);
static bool _search(LIST* plist,NODE** ppre,NODE** ploc,void* pargu);

/*=============== createList ==============
Allocates dynamic memory for a list head node and returns its address to caller
Pre: compare is address of compare function  used to compare two nodes.
Post:   head has allocated or error returned
     Return head node pointer or null if overflow*/
```

```
LIST* createlist(int(*compare)(void* argu1,void* argu2))
{
    //Local Definitions
        LIST* list;
    //Statements
        list = (LIST*)malloc(sizeof(LIST));
        if(list)
        {
                list->head    = NULL;
                list->pos     = NULL;
                list->rear    = NULL;
                list->count   = 0;
                list->compare = compare;
        }//if
        return list;
}//create list
/*================== addNode ================
 Inserts data into list.
Pre:plist is pointer to valid list; datainptr pointer to insertion data
Post: data inserted or error
       Return  -1 if overflow
       0 if successful
       1 if dup key       */
int addnode(LIST* plist,void* datainptr)
{
    //Local Definitions
        bool found;
        bool success;
        NODE* ppre;
        NODE* ploc;
     //Statements
        found = _search(plist,&ppre,&ploc,datainptr);
        if(found)
       // Duplicate keys not allowed
        return (+1);
success = _insert(plist,ppre,datainptr);
if(success)
 // Overflow
return -1;
```

```
   return 0;
} // addnode
/*================== _insert ==================
Inserts data pointer into a new node.
Pre:plist pointer to a valid list; ppre:  pointer to data's predecessor
datainPtr :data pointer to be inserted
Post: data have been inserted in sequence
      Return boolean, true  if successful,
      false if memory overflow */
static bool _insert(LIST* plist,NODE* ppre,void* datainptr)
{
      //Local Definitions
          NODE* pnew;
          pnew = (NODE*)malloc(sizeof(NODE));
      //Statements
          if(!pnew)
          return false;
          pnew->dataptr = datainptr;
          pnew->link = NULL;
          if(ppre == NULL)
          {
       // Adding before first node or to empty list
          pnew->link  = plist->head;
          plist->head = pnew;
          if(plist->count == 0)
        // Adding to empty list. Set rear
          plist->rear = pnew;
      } // if pPre
  else
   {
      // Adding in middle or at end
          pnew->link = ppre->link;
          ppre->link =  pnew;
           // Now check for add at end of list
          if(pnew->link == NULL)
          plist->rear = pnew;
} // if else
(plist->count)++;
return true;
}//_insert
```

```
/*================= removeNode ===============
Removes data from list.
Pre: plist pointer to a valid list; pointer to key to be deleted;dataoutptr: pointer to data pointer
Post: Node deleted or error returned.
      Return false not found; true deleted */
bool removenode(LIST* plist,void* keyptr,void** dataoutptr)
{
       //Local Definitions
          bool found;
          NODE* ppre;
          NODE* ploc;
      //Statements
          found = _search(plist,&ppre,&ploc,keyptr);
          if(found)
          _delete(plist,ppre,ploc,dataoutptr);
          return found;
}//removenode
/*================= _delete ===============
Deletes data from a list and returns pointer to data to calling module.
Pre:plist pointer to valid list.
Ppre: pointer to predecessor node
ploc: pointer to target node
dataoutptr: pointer to data pointer
Post   Data have been deleted and returned
      Data memory has been freed     */
void _delete(LIST* plist,NODE* ppre,NODE* ploc,void** dataoutptr)
{
       //Statements
          *dataoutptr = ploc->dataptr;
          if(ppre == NULL)
         // Deleting first node
          plist->head = ploc->link;
          else
        // Deleting any other node
          ppre->link = ploc->link;
          // Test for deleting last node
          if(ploc->link == NULL)
          plist->rear = ppre;
          (plist->count)--;
          free(ploc);
```

```
        return ;
}//_delete
/*================== searchList ================
Interface to search function.
Pre :plist pointer to initialized list;pargu pointer to key being sought
Post: pdataout contains pointer to found data  -or- NULL if not found
    Return boolean true successful; false not found */
bool searchlist(LIST* plist,void* pargu,void** pdataout)
{
    //Local Definitions
        bool found;
        NODE* ppre;
        NODE* ploc;
        //Statements
        found = _search(plist,&ppre,&ploc,pargu);
        if(found)
           *pdataout = ploc->dataptr;
        else
           *pdataout = NULL;
        return found;
}//searchlist
/*================== _search ================
Searches list and passes back address of node containing target and its logical predecessor.
Pre : plist pointer to initialized list; ppre  pointer variable to predecessor:ploc  pointer variable to
receive node: pargu pointer to key being sought
Post: ploc points to first equal/greater key -or- null if target> key of last node; ppre points to
largest node < key  -or- null if target < key of first node
    Return boolean true found; false not found
*/
bool _search(LIST* plist,NODE** ppre,NODE** ploc,void* pargu)
{
     //Macro Definition
        #define COMPARE\
        (((*plist->compare)(pargu,(*ploc)->dataptr)))
        #define COMPARE_LAST\
        ((*plist->compare)(pargu,plist->rear->dataptr))
        //Local Definitions
        int result;
     //Statements
        *ppre = NULL;
```

```
        *ploc = plist->head;
        if(plist->count == 0)
          return false;
        // Test for argument > last node in list
        if(COMPARE_LAST > 0)
        {
                *ppre=plist->rear;
                *ploc=NULL;
                return false;
        }//if
        while((result = COMPARE) > 0)
        {
       // Have not found search argument location
                *ppre=*ploc;
                *ploc=(*ploc)->link;
        }//while
        if(result == 0)
       // argument found--success
          return true;
        else
          return false;
}//_search

/*================== retrieveNode ================
 retrieves data in the list without changing the list contents.
Pre: plist pointer to initialized list; pargu pointer to key to be retrieved
Post : Data (pointer) passed back to caller
     Return boolean true success; false underflow */
static bool retrievenode(LIST* plist,void* pargu,void** dataoutptr)
{
     //Local Definitions
        bool found;
        NODE* ppre;
        NODE* ploc;
      //Statements
        found = _search(plist,&ppre,&ploc,pargu);
        if(found)
        {
                *dataoutptr = ploc->dataptr;
                return true;
```

```
        }//if
        *dataoutptr = NULL;
        return false;
}//retrievenode
/*================ emptyList ================
Returns boolean indicating whether or not the list is empty
Pre: plist is a pointer to a valid list
    Return boolean true empty; false list has data
*/
bool emptylist(LIST* plist)
{
        return(plist->count == 0);
}//emptylist
/*================== fullList =================
Returns boolean indicating no room for more data.This list is full if memory cannot be allocated
for another node.
Pre: plist pointer to valid list
Return boolean true if full
    false if room for node */
bool fulllist(LIST* plist)
{
     //Local Definitions
        NODE* temp;
      //Statements
        temp = (NODE*)malloc(sizeof(NODE));
        if(temp)
        {
                free(temp);
                return false;
        }//if
      // Dynamic memory full
        return true;
}//fulllist
/*================== listCount ==================
Returns number of nodes in list.
Pre: plist is a pointer to a valid list
     Return count for number of nodes in list
*/
int listcount(LIST* plist)
{
```

```
    //Statements
        return plist->count;
}//listcount
/*================== traverse ================
Traverses a list. Each call either starts at the beginning of list or returns the location of the
next element in the list.
Pre : plist  pointer to a valid list : fromwhere:   0 to start at first element
Dataptrout:  address of pointer to data
Post:   if more data, address of next node
    Return true node located; false if end of list
*/
bool traverse(LIST* plist,intfromwhere,void** dataptrout)
{
     //Statements
        if(plist->count == 0)
        return false;

         // Start from first node
        if(fromwhere == 0)
        {
                plist->pos = plist->head;
                *dataptrout = plist->pos->dataptr;
                return true;
        } // if fromwhere
        else
        {     // Start from current position
                if(plist->pos->link == NULL)
                return false;
                else
                {
                        plist->pos  = plist->pos->link;
                        *dataptrout = plist->pos->dataptr;
                        return true;
                }//if else
        }//if fromwhere else
}//traverse
/*================== destroyList ================
Deletes all data in list and recycles memory
Pre: List is a pointer to a valid list.
Post: All data and head structure deleted
```

```
      Return null head pointer
*/
LIST* destroylist(LIST* plist)
{
       //local definitions
          NODE* deleteptr;
      //Statements
          if(plist)
          {
                    while(plist->count  >0)
                    {
                    // First delete data
                              free(plist->head->dataptr);
                    // Now delete node
                              deleteptr   = plist->head;
                              plist->head = plist->head->link;
                              (plist->count)--;
                              free(deleteptr);
                    }//while
                    free(plist);
                 }//if
     return NULL;
}//destroy list
/*================== compare =================
Compare the two integers and returns the result.
Pre : This function takes two integers as arguments
Post: returns  -1  if integer1 less than integer 2.
        returns +1 if integer1 is greater than integer2.
  returns 0 if integer1 is equal to integer2.
*/
int compare(void* pargu1,void* pargu2)
{
          int i1,i2;
          int result;
          i1=(*(int*)pargu1);
          i2=(*(int*)pargu2);
          if(i1<i2)
           result = -1;
          else if(i1>i2)
           result = +1;
```

```
        else
         result = 0;
        return result;
}
/*================== printlist ================
Prints the data present in the list.
Pre : list is a meta data structure.
Post: displays all the elements of the list.
  return true node located; false if end of list
*/
void printlist(LIST* plist)
{
        int* partptr;
        plist->pos = NULL;
        if(plist->count == 0)
        printf("sorry");
        else
        {
                printf("linked list contents are:");
                traverse(plist,0,(void**)&partptr);
                do
                {
                        printf("%d,",*partptr);
                }while(traverse(plist,1,(void**)&partptr));
        }
}
```

listmain.c

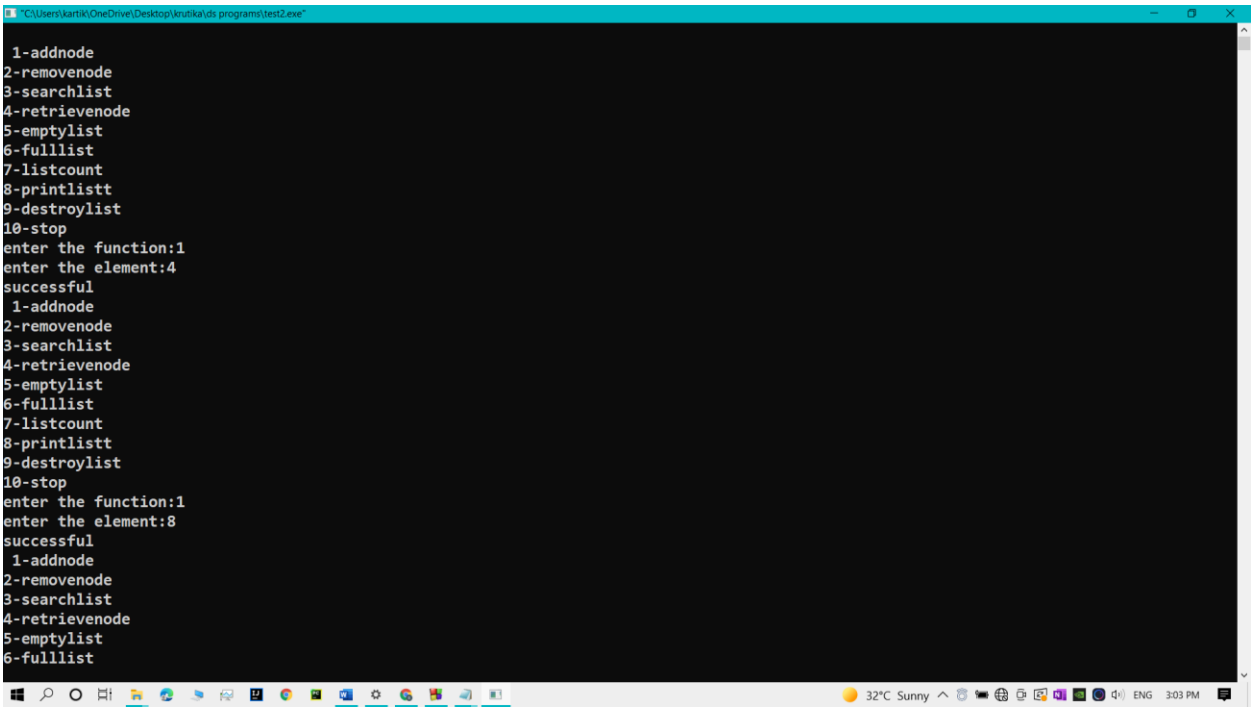
/* create an integer singly linked list and perform all the linked list operations defined in the ADT*/

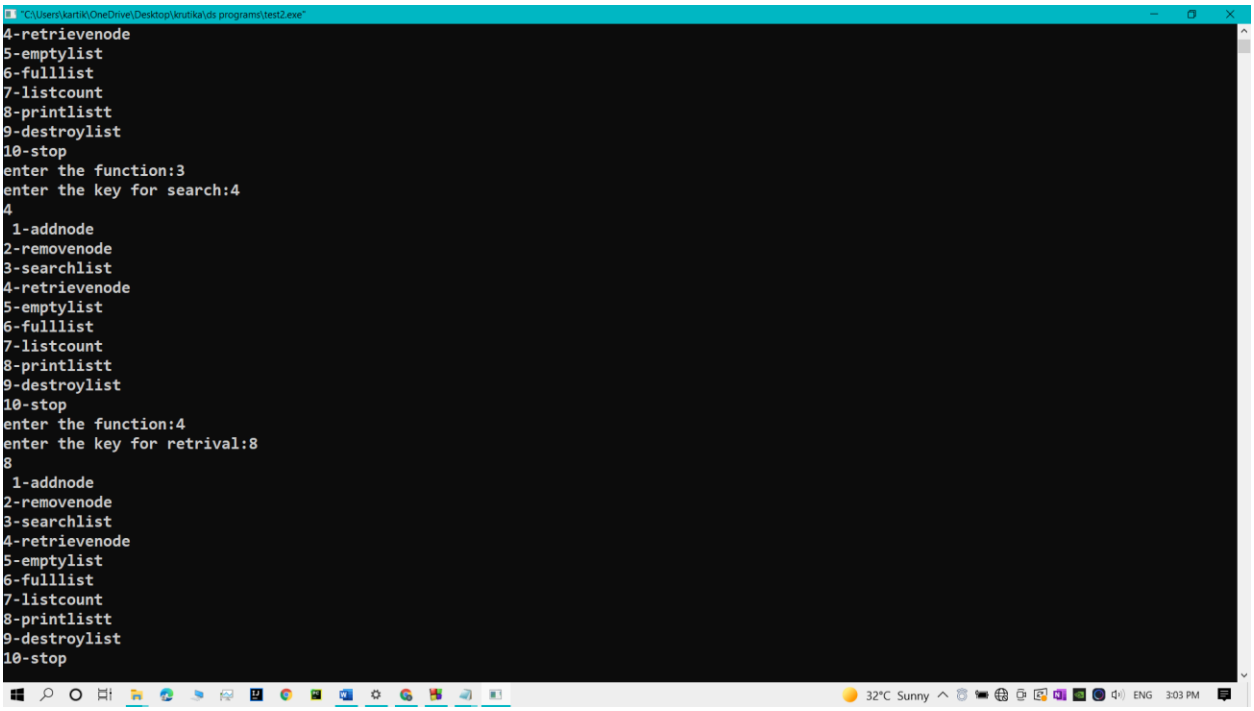
```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "singlyll.h.h"

int main()
{
        LIST* list;
        int t1;
        LIST* t;
        bool found;
        int partnum;
        int* partptr;
        list=createlist(compare);
        int ch,c;
        while(true)
        {
                printf("\n 1-addnode\n");
                printf("2-removenode\n");
                printf("3-searchlist\n");
                printf("4-retrievenode\n");
                printf("5-emptylist\n");
                printf("6-fulllist\n");
                printf("7-listcount\n");
                printf("8-printlistt\n");
                printf("9-destroylist\n");
                printf("10-stop\n");
                printf("enter the function:");
                scanf("%d",&ch);
                switch(ch)
                        {
                         case 1:printf("enter the element:");
                                 int *pele=(int*)malloc(sizeof(int));
                                scanf("%d",pele);
                                t1 = addnode(list,pele);
                                if(t1 == 1)
```

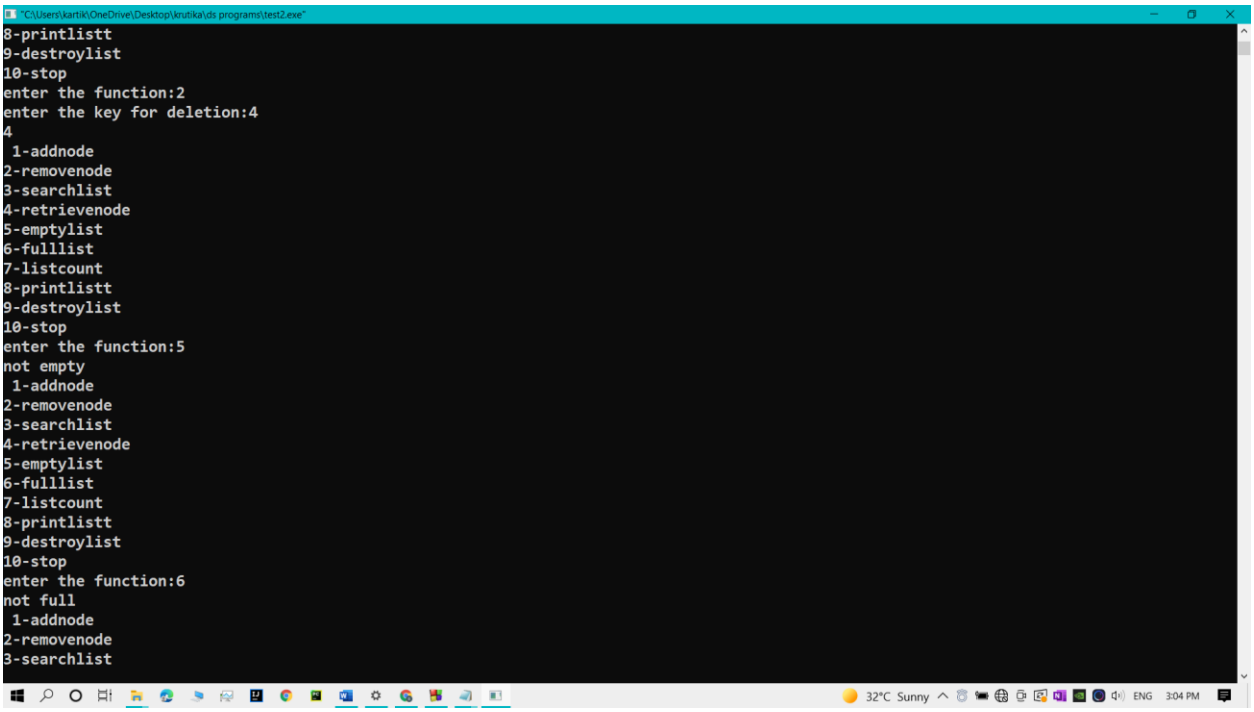
```
            printf("duplicate key");
           else if(t1 == 0)
           printf("overflow");
           else
            printf("successful");
            printf("\telement is inserted.");
           break;


case 2:printf("enter the key for deletion:");
            scanf("%d",&partnum);
            found = removenode(list,&partnum,(void*)&partptr);
            if(found)
              printf("%d",*partptr);
           else
              printf("not removed");
           break;
case 3:printf("enter the key for search:");
            scanf("%d",&partnum);
            found = searchlist(list,&partnum,(void*)&partptr);
            if(found)
             printf("%d",*partptr);
            else
               printf("not found");
            break;
case 4:printf("enter the key for retrival:");
            scanf("%d",&partnum);
found = retrievenode(list,&partnum,(void*)&partptr);
 if(found)
printf("%d",*partptr);
 else
printf("not found");
break;
case 5: if(emptylist(list))
printf("list empty");
else
printf("not empty");
  break;
case 6: if(fulllist(list))
printf("full list");
```

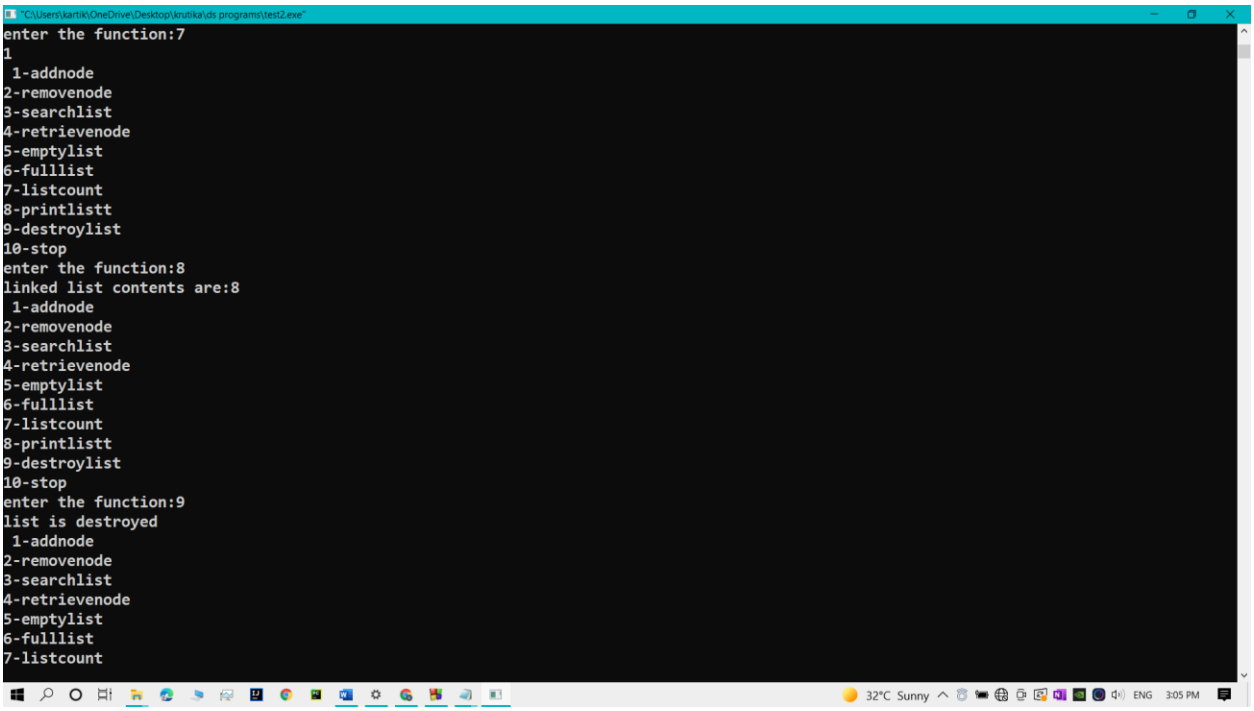
```
                    else
                printf("not full");
                  break;
                case 7:c = listcount(list);
                printf("%d",c);
                  break;
                case 8:printlist(list);
                break;
                case 9:t = destroylist(list);
                if(t == NULL)
                printf("list is destroyed");
                 else
                printf("not destroyed");
                break;
                default:exit(0);
            }
        }
        return 0;
}
```

OUTPUT:



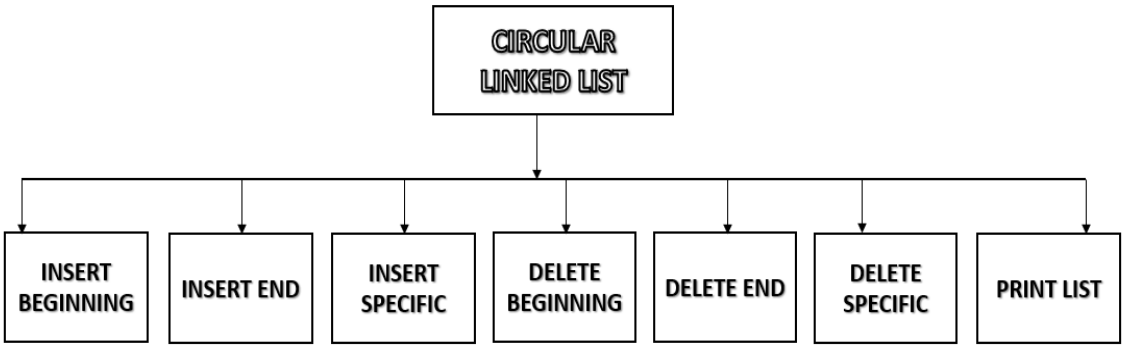

"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test2.exe"
4-retrievenode
5-emptylist
6-fulllist
7-listcount
8-printlistt
9-destroylist
10-stop
enter the function:3
enter the key for search:4
4
1-addnode
2-removenode
3-searchlist
4-retrievenode
5-emptylist
6-fulllist
7-listcount
8-printlistt
9-destroylist
10-stop
enter the function:4
enter the key for retrival:8
8
1-addnode
2-removenode
3-searchlist
4-retrievenode
5-emptylist
6-fulllist
7-listcount
8-printlistt
9-destroylist
10-stop
32°C Sunny
ENG
3:03 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test2.exe"
8-printlistt
9-destroylist
10-stop
enter the function:2
enter the key for deletion:4
4
1-addnode
2-removenode
3-searchlist
4-retrievenode
5-emptylist
6-fulllist
7-listcount
8-printlistt
9-destroylist
10-stop
enter the function:5
not empty
1-addnode
2-removenode
3-searchlist
4-retrievenode
5-emptylist
6-fulllist
7-listcount
8-printlistt
9-destroylist
10-stop
enter the function:6
not full
1-addnode
2-removenode
3-searchlist
32°C Sunny
ENG
3:04 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test2.exe"
enter the function:7
1
1-addnode
2-removenode
3-searchlist
4-retrievenode
5-emptylist
6-fulllist
7-listcount
8-printlistt
9-destroylist
10-stop
enter the function:8
linked list contents are:8
1-addnode
2-removenode
3-searchlist
4-retrievenode
5-emptylist
6-fulllist
7-listcount
8-printlistt
9-destroylist
10-stop
enter the function:9
list is destroyed
1-addnode
2-removenode
3-searchlist
4-retrievenode
5-emptylist
6-fulllist
7-listcount
32°C Sunny
ENG
3:05 PM

ASSIGNMENT 7

Write a C program to implement circular linked list (un ordered list) (circularll.c) for the following list operations: i) insert (beginning, end, insert after) ii) delete (beginning, end, delete specific) iii) print list.




```
/* Performs CLL operations*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
struct node
{
        int data;
        struct node* next;
};
struct node* list = NULL;
void insertbeginning(int );
void insertend(int );
void insertafter(int ,int );
void deletebeginning();
void deleteend();
void deletespecific(int );
void printlist();
int main()
{
        int value;
        int loc;
        int ch;
        while(true)
        {
```

```
        printf("\n 1-insert at beginning\n");
        printf(" 2-insert at end\n");
        printf("3-insert after location\n");
        printf("4-delete at beginning\n");
        printf("5-delete at end\n");
        printf("6-delete at specific\n");
        printf("7-print list\n");
        printf("8-stop\n");
        printf("enter the function :");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1:printf("enter the element:");
                        scanf("%d",&value);
                        insertbeginning(value);
                        break;
                case 2:printf("enter the element:");
                        scanf("%d",&value);
                        insertend(value);
                        break;
                case 3:printf("enter the element:");
                         scanf("%d",&value);
                        printf("enter the location:");
                       scanf("%d",&loc);
                       insertafter(value,loc);
                       break;
                case 4:deletebeginning();
                          break;
                case 5:deleteend();
                         break;
                case 6:printf("enter the location:");
                        scanf("%d",&loc);
                       deletespecific(loc);
                        break;
                case 7:printlist();
                          break;
                default:exit(0);
        }
}
return 0; }
```

```
/*================== insertbeginning ================
Inserts data at the beginning of list.
Pre:value holds the data to be inserted into the list.
Post: Inserts data at the beginning of the list .
*/
void insertbeginning(int value)
{
        struct node* newnode;
        newnode = (struct node*)malloc(sizeof(struct node));
        newnode->data = value;
   //no node in the list
        if(list == NULL)
        {
                list = newnode;
                newnode->next = list;
        }
        else
        {
                struct node* temp = list;
                while(temp->next != list)
                temp = temp->next;
                newnode->next = list;
                list = newnode;
                temp->next = list;
        }
return;
}
/*================== insertend ================
 Inserts data at the end of list.
Pre:value holds the data to be inserted into the list.
Post: data has been inserted at the end of the list.
*/
void insertend(int value)
{
        struct node* newnode;
        newnode = (struct node*)malloc(sizeof(struct node));
        newnode->data = value;
  //no node in the list
        if(list == NULL)
```

```
	{
		list = newnode;
		newnode->next = list;
	}
	else
	{
		struct node* temp = list;
		//locate last node
		while(temp->next != list)
		temp = temp->next;

		temp->next = newnode;
		newnode->next = list;
	}
	return;
}
/*================== insertafter =================
 Inserts data after the specified location in the list.
Pre : list has been created and value holds the data to be inserted.
Post: data has been inserted after specified location.
*/
void insertafter(int value,int loc)
{
	struct node* newnode;
	newnode = (struct node*)malloc(sizeof(struct node));
	newnode->data = value;

	struct node* temp = list;
	//locate the location by comparing data and location
	while(temp->data != loc)
	{
		if(temp->next == list)
		{
			printf("no need to search");
			return;
		}
		else
		temp = temp->next;
	}
	newnode->next = temp->next;
```

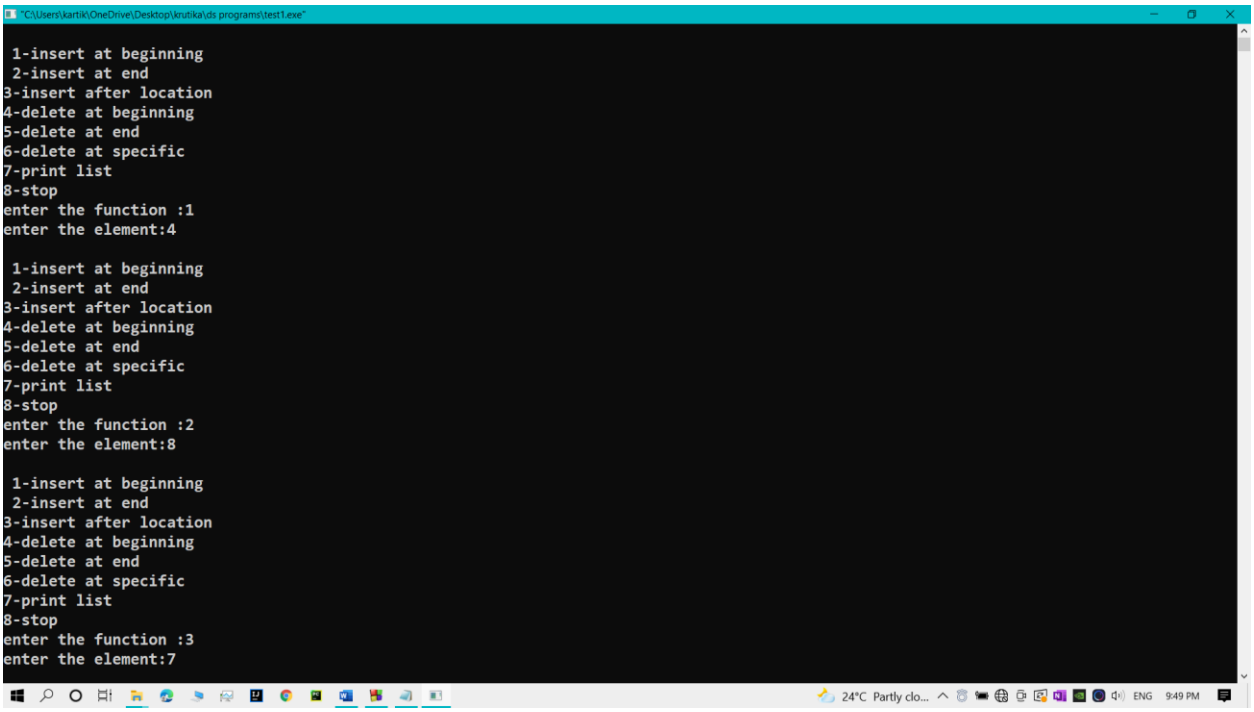
```
        temp->next = newnode;
        return;
}
/*================== deletebeginning ================
 deletes the data present at the beginning of list.
Pre :list has been created .
Post: data present at the beginning has been deleted from the list.
*/
void deletebeginning()
{
        if(list == NULL)
        printf("list is empty");
        else
        {
                struct node* temp1 = list;
                struct node* temp2 = list;

                if(temp2->next == list)
                {
                        list = NULL;
                        free(temp2);
                        return;
                }
                else
                {
                        while(temp1->next != list)
                        temp1 = temp1->next;

                        list = temp2->next;
                        temp1->next = list;
                        free(temp2);
                }
        }
        return;
}

/*================== delete end================
deletes the data present at the end of list.
Pre :list has been created .
Post: data present at the end has been deleted from the list.*/
```

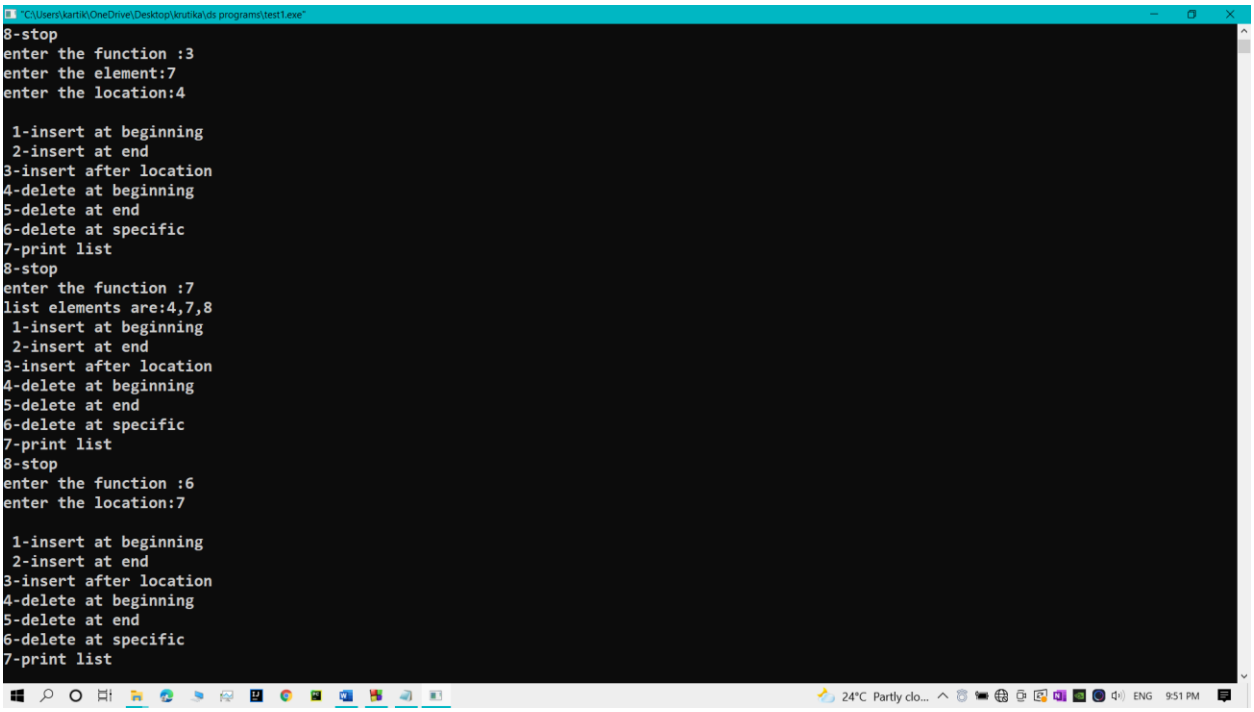
```
void deleteend()
{
        if(list == NULL)
        printf("empty");
        else
        {
                struct node* temp1;
                struct node* temp2;
                temp1 = list;
                if(temp1->next == list)
                {
                        list = NULL;
                        free(temp1);
                        return;
                }
                else
                {
                        while(temp1->next != list)
                        {
                                temp2 = temp1;
                                temp1 = temp1->next;
                        }
                        temp2->next = list;
                        free(temp1);
                }
        }
        return;
}
/*================== deletespecific================
deletes the data present at the end of list.
Pre :list has been created .
Post: data present at the specified location has been deleted from the list.
*/
void deletespecific(int loc)
{
        if(list == NULL)
        {
                printf("empty");
                return;
        }
```

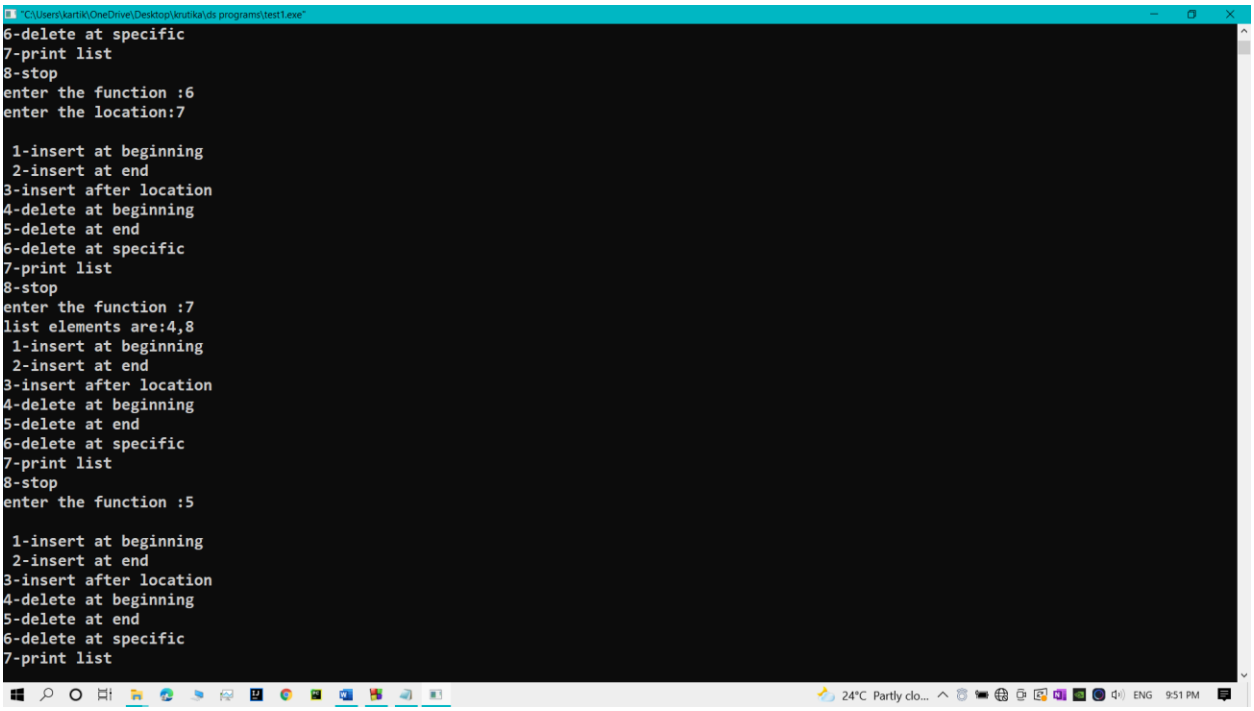
```
else
{
        struct node*temp1;
        struct node* temp2;
        temp1 = list;
        while(temp1->data != loc)
        {
           if(temp1->next == list)
                printf("not found");
        else
        {
                temp2 = temp1;
                temp1 = temp1->next;
        }
}
        // single node in list
        if((temp1->next == list)&&(temp2 == NULL))
        {
                list = NULL;
                free(temp1);
                return;
        }
        else //deletespecific is first node
        if(temp1 == list)
        {
                temp2 = list;
                while(temp2->next != list)
                temp2 = temp2->next;
                list = list->next;
                free(temp1);
        }
        else   //deletespecific is last node
        {
                if((temp1->next == list)&&(temp2 == NULL))
                temp2->next = list;

                else //DELETE SPECIFIC IS MIDDLENODE
                temp2->next = temp1->next;
                free(temp1);
        }
```

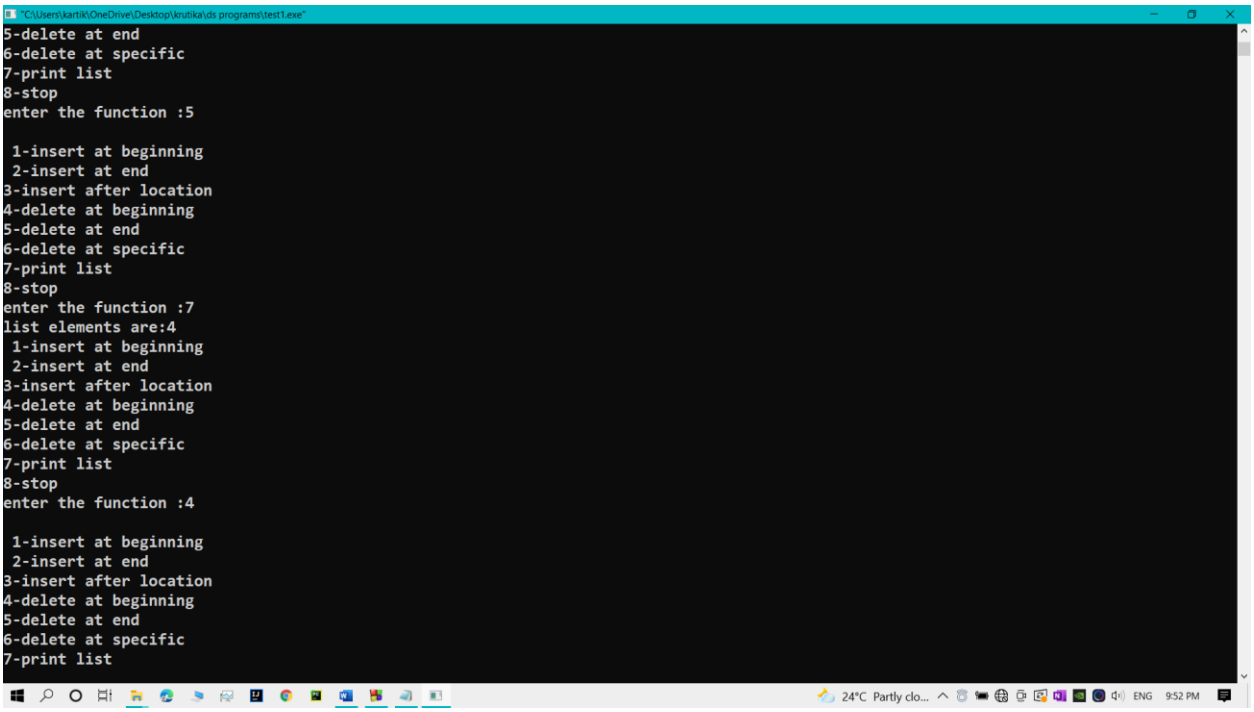
```
        }
        return;
}
/*================== printlist=================
Prints the data present in the list
Pre :list has been created .
Post: all the elements of the list are displayed.
*/
void printlist()
{
        if(list == NULL)
        {
                printf("list is empty");
                return;
        }
        else
        {
                struct node* temp;
                temp = list;
                while(temp->next != NULL)
                {
                        printf("%d,",temp->data);
                        temp = temp->next;
                }
                printf("%d",temp->data);
        }
        return;
}
```

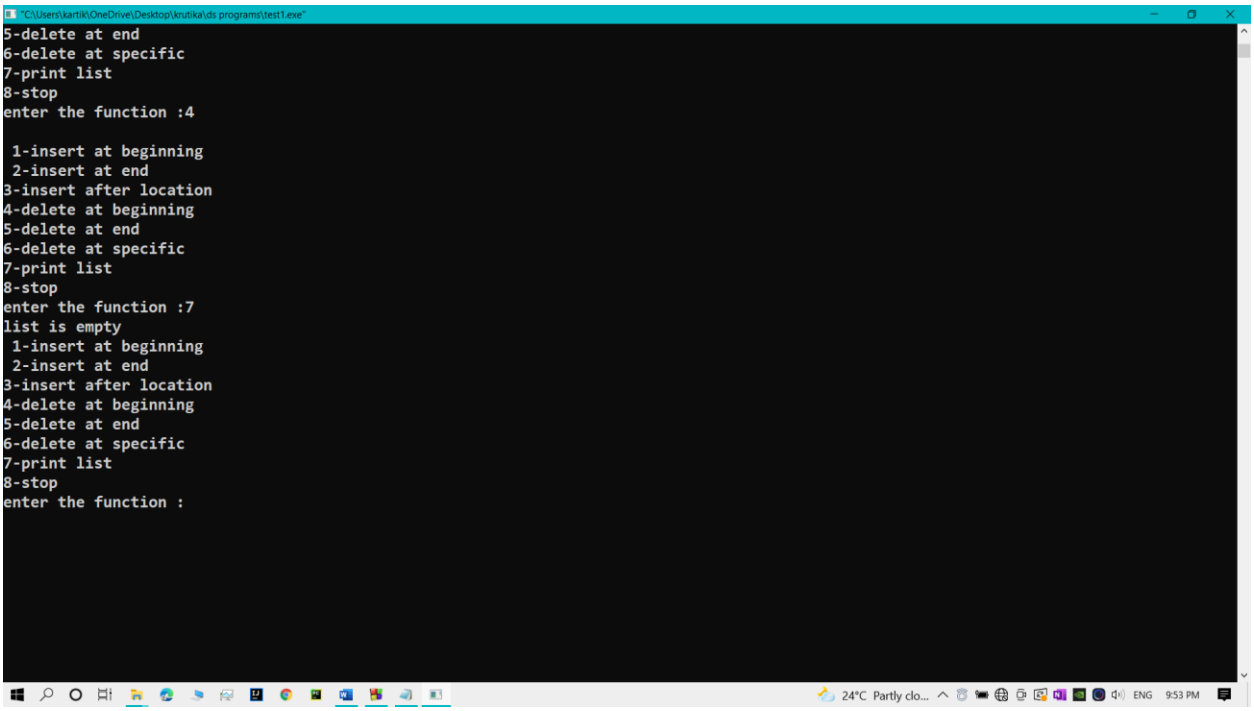
OUTPUT:


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test1.exe"
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :1
enter the element:4
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :2
enter the element:8
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :3
enter the element:7
24°C Partly clo...
ENG
9:49 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test1.exe"
8-stop
enter the function :3
enter the element:7
enter the location:4
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :7
list elements are:4,7,8
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :6
enter the location:7
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
24°C Partly clo...
ENG
9:51 PM

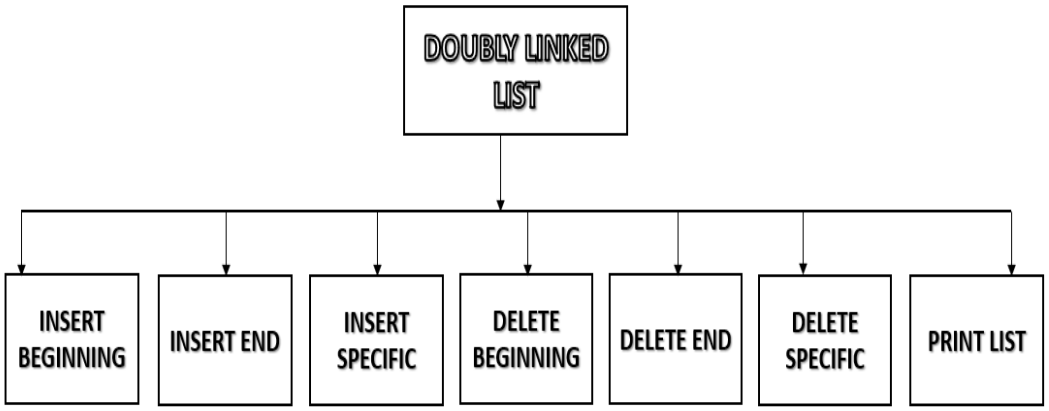

"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test1.exe"
6-delete at specific
7-print list
8-stop
enter the function :6
enter the location:7
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :7
list elements are:4,8
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :5
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
24°C Partly clo...
ENG
9:51 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test1.exe"
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :5
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :7
list elements are:4
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :4
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
24°C Partly clo...
ENG
9:52 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test1.exe"
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :4
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :7
list is empty
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-print list
8-stop
enter the function :
24°C Partly clo...
ENG
9:53 PM

ASSIGNMENT 8

Write a C program to implement doubly linked list (un ordered list) (doublyll.c) for the following list operations: i) insert (beginning, end, insert after) ii) delete (beginning, end, delete specific) iii) print list.




```
/*performs DLL operations*/
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
struct node
{
        int data;
        struct node*prev;
        struct node* next;
};
struct node* list = NULL;
void insertbeginning(int value);
void insertend(int value);
void insertafter(int value,int loc);
void deletebeginning();
void deleteend();
void deletespecific(int delvalue);
void display();
```

```
int main()
{
        int value;
        int delvalue;
        int loc;
        int ch;

        while(true)
        {
                printf("\n 1-insert at beginning\n");
                printf(" 2-insert at end\n");
                printf("3-insert after location\n");
                printf("4-delete at beginning\n");
                printf("5-delete at end\n");
                printf("6-delete at specific\n");
                printf("7-printlist\n");
                printf("8-stop\n");
                printf("enter the function :");
                scanf("%d",&ch);
                switch(ch)
                {

                 case 1:printf("enter the element:");
                        scanf("%d",&value);
                        insertbeginning(value);
                        break;
                case 2:printf("enter the element:");
                        scanf("%d",&value);
                        insertend(value);
                         break;
                case 3:printf("enter the element:");
                        scanf("%d",&value);
                        printf("enter the location:");
                        scanf("%d",&loc);
                        insertafter(value,loc);
                         break;
                case 4:deletebeginning();
                        break;
                 case 5:deleteend();
                         break;
```

```
                case 6:printf("enter the element to be deleted:");
                        scanf("%d",&delvalue);
                        deletespecific(delvalue);
                        break;
                case 7:display();
                        break;
                        default:exit(0);
        }
}
return 0;
}
/*================== insertbeginning =================
Inserts data at the beginning of list.
Pre:value holds the data to be inserted into the list.
Post: Inserts data at the beginning of the list .
*/
void insertbeginning(int value)
{
        struct node* newnode;
        newnode=(struct node*)malloc(sizeof(struct node));
        newnode->data = value;
        newnode->prev = NULL;
      //empty list
        if(list == NULL)
        {
                newnode->next = NULL;
                list = newnode;
        }
        else
        {
                newnode->next = list;
                list->prev = newnode;
                list = newnode;
        }
        return;
}
```

```
/*================== insertend ================
 inserts data at the end of list.
Pre:value holds the data to be inserted into the list.
Post: data has been inserted at the end of the list.
*/
void insertend(int value)
{
        struct node*newnode;
        newnode=(struct node*)malloc(sizeof(struct node));
        newnode->data = value;
        newnode->next = NULL;
      //empty list
        if(list == NULL)
        {
                newnode->prev = NULL;
                list        = newnode;
        }
        else    //locate last node
        {
                struct node*temp;
                temp = list;
                while(temp->next != NULL)
                temp = temp->next;
                temp->next  = newnode;
                newnode->prev = temp;
        }
        return;
}
/*================== insertafter ================
 Inserts data after the specified location in the list.
Pre : list has been created;value holds the data to be inserted.
Post: data has been inserted after specified location.
*/
void insertafter(int value,int loc)
{
        struct node*newnode;
        newnode=(struct node*)malloc(sizeof(struct node));
        newnode->data     =  value;
        struct node*temp1 = list;
        struct node*temp2;
```

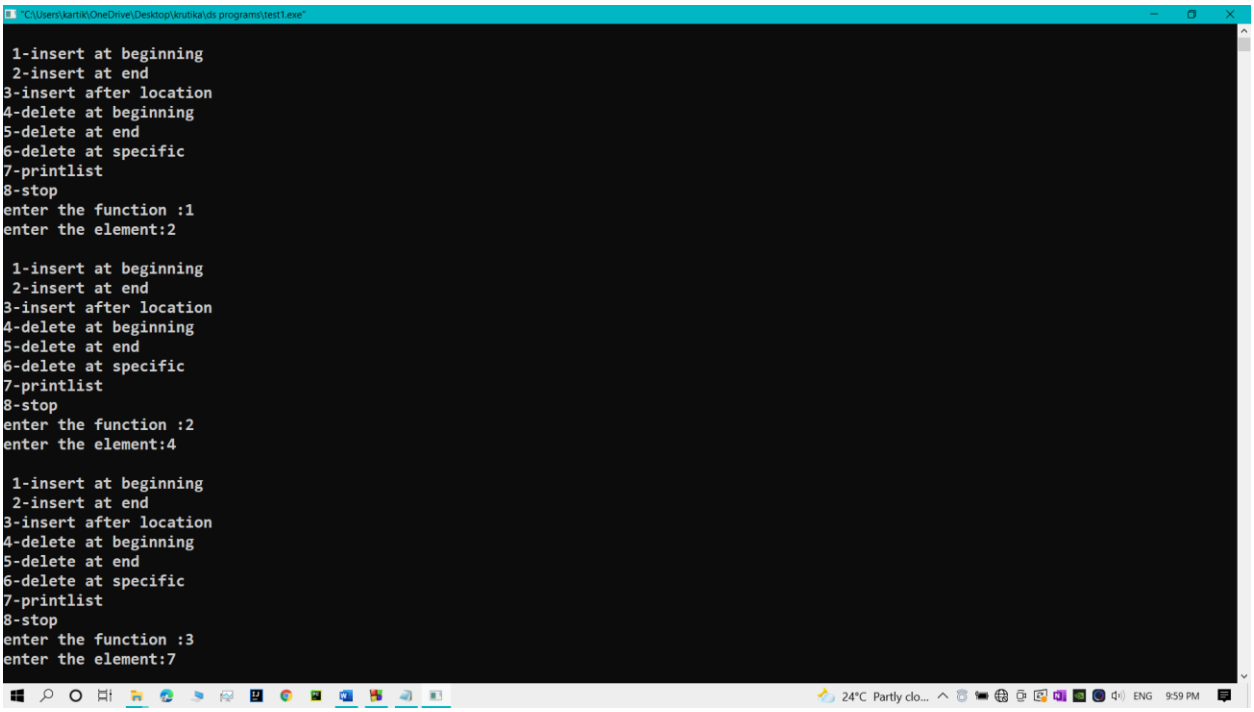
```
//locate the location for insertion
        while(temp1->data != loc)
        {
                if(temp1->next == NULL)
                {
                        printf("node not found in the list");
                        return;
                }
                else    //multiple nodes
                temp1=temp1->next;
        }
        temp2   =  temp1->next;
        temp1->next   = newnode;
        newnode->prev = temp1;
        newnode->next = temp2;
        temp2->prev   =  newnode;
        return;
}
/*================== deletebeginning ================
deletes the data present at the beginning of list.
Pre :list has been created .
Post: data present at the beginning has been deleted from the list.
*/
void deletebeginning()
{
        if(list == NULL)
        printf("list is empty");
        else
        {
                struct node* temp = list;
                //SINGLE NODE IN DLL
                if(temp->prev == temp->next)
                {
                        list = NULL;
                        free(temp);
                }
                else //multiple nodes
                {
                        list = temp->next;
                        list->prev = NULL;
```

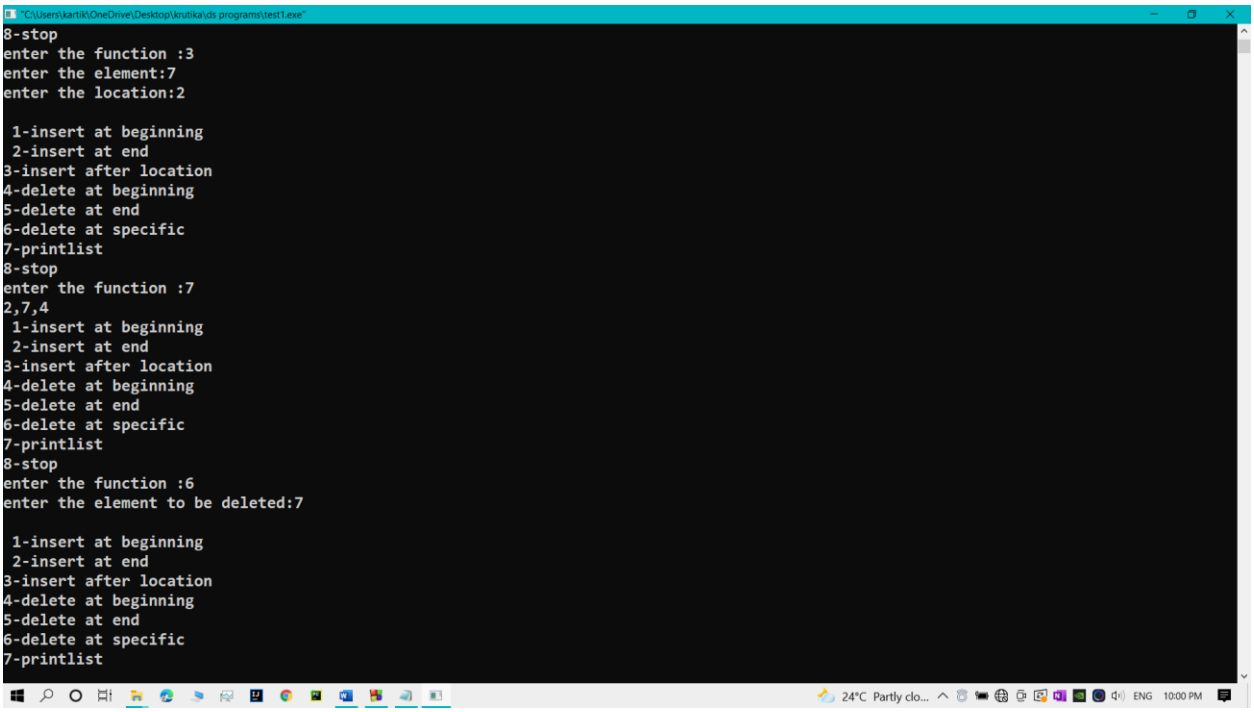
```
                        free(temp);
                }
        }
        return;
}
/*================== deleteend=================
deletes the data present at the end of list.
Pre :list has been created .
Post: data present at the end has been deleted from the list.
*/
void deleteend()
{
        //check for empty list
        if(list == NULL)
        {
                printf("list is empty");
                return;
        }
        //check for single node
        struct node* temp = list;
        if(temp->prev == temp->next)
        {
                list = NULL;
                free(temp);
                return;
        }
        while(temp->next != NULL)
        {
                temp = temp->next;
        }
        temp->prev->next = temp->next;
        free(temp);
        return;
}
/*================== deletespecific=================
deletes the data present at the end of list.
Pre :list has been created .
Post: data present at the specified location has been deleted from the list.
*/
```

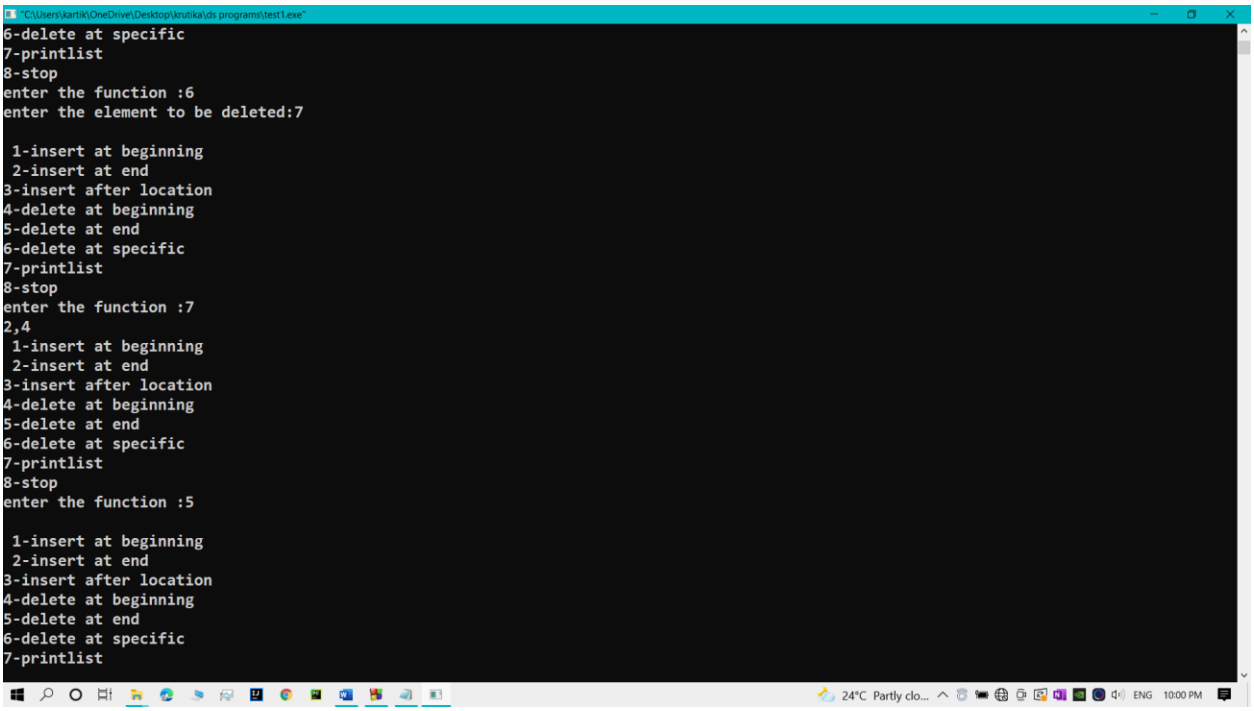
```
void deletespecific(int delvalue)
{
        if(list == NULL)
        {
                printf("empty list");
                return;
        }

        else
        {
                struct node* temp = list;
                //locate deleting node
                while(temp->data != delvalue)
                {
                        //check for single node
                        if(temp->next == NULL)
                        {
                                list = NULL;
                                free(temp);
                                return;
                        }
                        else
                        {
                                temp = temp->next;
                        }
                }//end of while
                if(temp == list)
                {
                        list=NULL;
                        free(temp);
                        return;
                }
                else
                {
                        temp->prev->next = temp->next;
                        temp->next->prev = temp->prev;
                        free(temp);
                }
                return;
        }
```

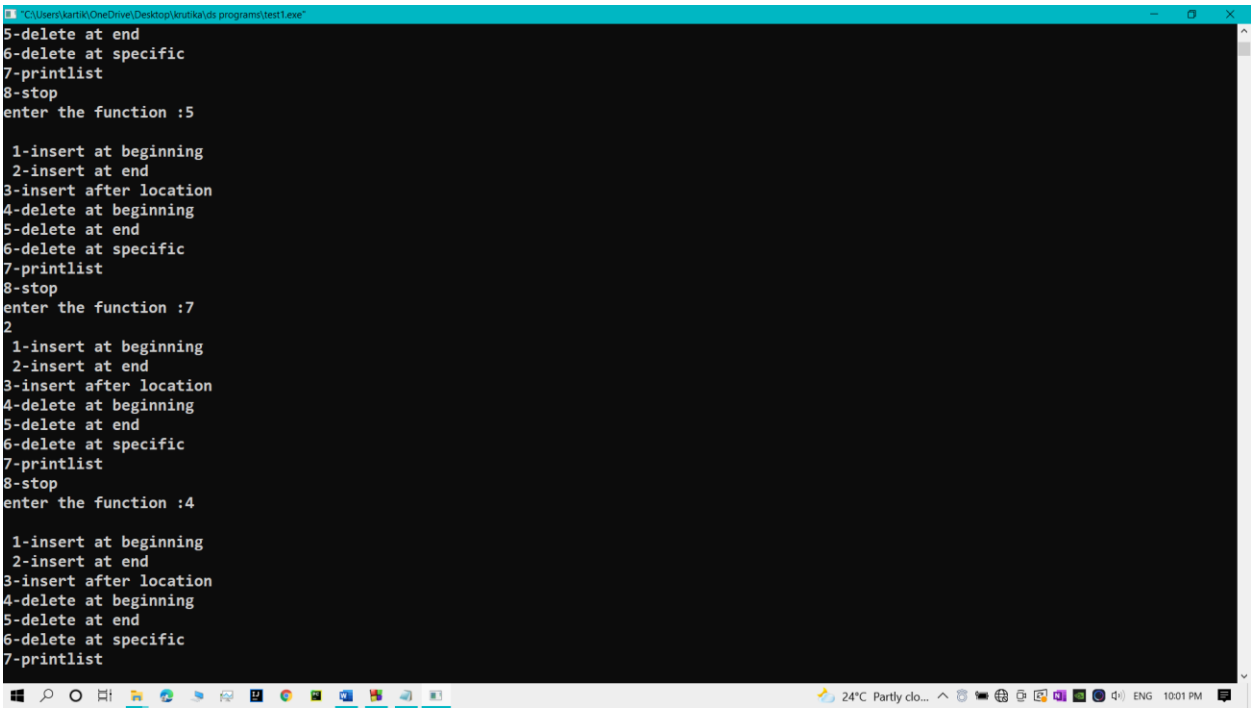
```
}
/*================== Display================
displays the data present in the list
Pre :list has been created .
Post: all the elements of the list are displayed.
*/
void display()
{
        if(list == NULL)
        {
                printf("list is empty");
                return;
        }
        else
        {
                struct node* temp;
                temp = list;
                while(temp->next != NULL)
                {
                        printf("%d,",temp->data);
                        temp = temp->next;
                }
                printf("%d",temp->data);
        }
        return;
}
```

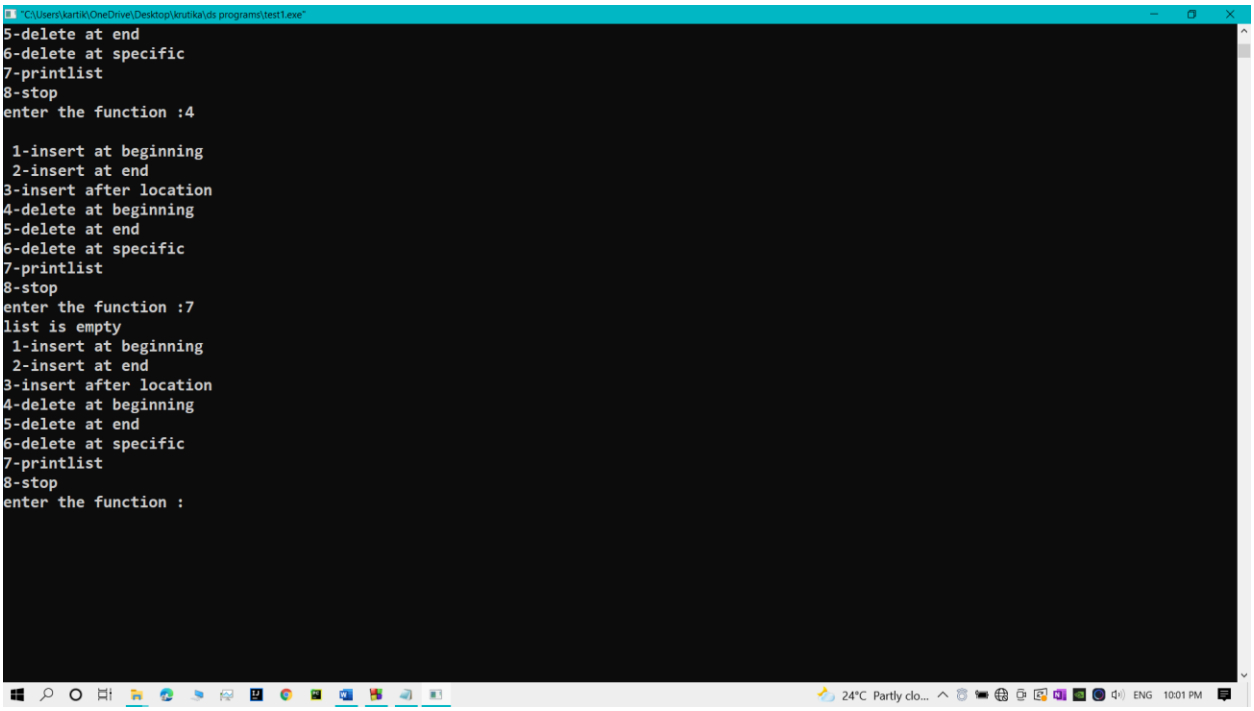
OUTPUT:


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test1.exe"
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :1
enter the element:2
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :2
enter the element:4
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :3
enter the element:7
24°C Partly clo...
ENG 9:59 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test1.exe"
8-stop
enter the function :3
enter the element:7
enter the location:2
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :7
2,7,4
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :6
enter the element to be deleted:7
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
24°C Partly clo...
ENG 10:00 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test1.exe"
6-delete at specific
7-printlist
8-stop
enter the function :6
enter the element to be deleted:7
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :7
2,4
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :5
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
24°C Partly clo...
ENG
10:00 PM

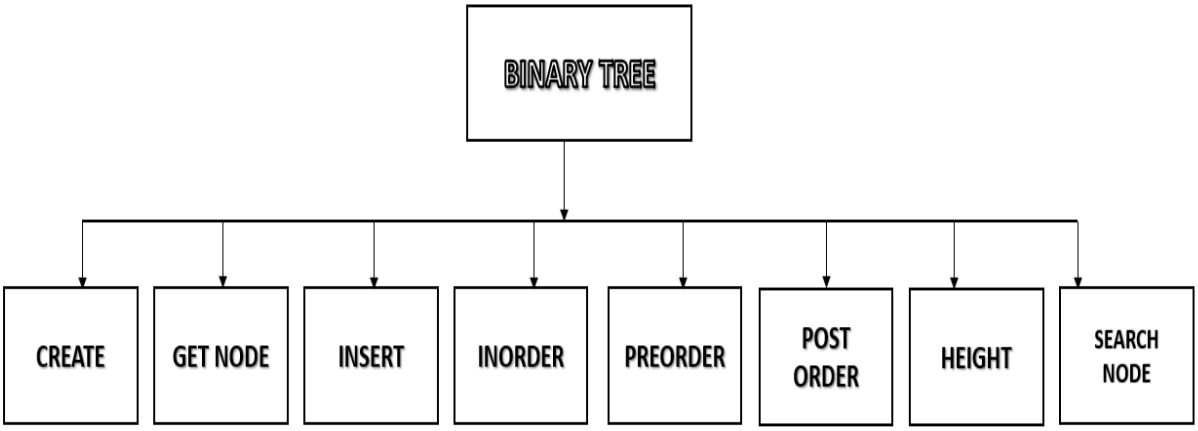

"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test1.exe"
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :5
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :7
2
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :4
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
24°C Partly clo...
ENG
10:01 PM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\test1.exe"
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :4
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :7
list is empty
1-insert at beginning
2-insert at end
3-insert after location
4-delete at beginning
5-delete at end
6-delete at specific
7-printlist
8-stop
enter the function :
24°C Partly clo...
ENG
10:01 PM

PROGRAMS ON TRESS

ASSIGNMENT 9

Write a C program to create a binary tree (BT) of integers and write C functions to perform the following operations on the BT: i) create a BT ii) preorder traversal iii) inorder traversal iv) post order traversal v) height of BT vi) Search node.




```
/*this program performs binary tree operations*/
#include<stdio.h>
#include<stdio.h>
#include<ctype.h>
#include<stdbool.h>
#include<stdlib.h>
struct bt
{
int data;
struct bt* left;
struct bt* right;
};
typedef struct bt node;
node* new = NULL;
node* root = NULL;

/*================== getnode=================
allocates memory for the node
Pre :nothing.
Post: new node is created and pointed to by new.
*/
```

```
node* getnode()
{
        node* temp;
        temp=(node*)malloc(sizeof(node));
        temp->left=NULL;
        temp->right=NULL;
        return temp;
}//get node

//Prototype Declarations
void insert(node*,node*);
void inorder(node*);
void postorder(node*);
void preorder(node*);
bool searchnode(node* root,int key);
int height(node* root);
void create();

/*================== create node=================
initilizes the node
Pre :node is created
Post: data is initilized in the node.
*/
void create()
{
new=getnode();
printf("enter the data\n");
scanf("%d",&new->data);
//No node in the binary tree
if(root==NULL)
   root = new;
else
     insert(root,new);
return;
}
/*================== insert node=================
inserts the node at specified position
Pre :root node is present.
Post:data node is been inserted at specified position.
*/
```

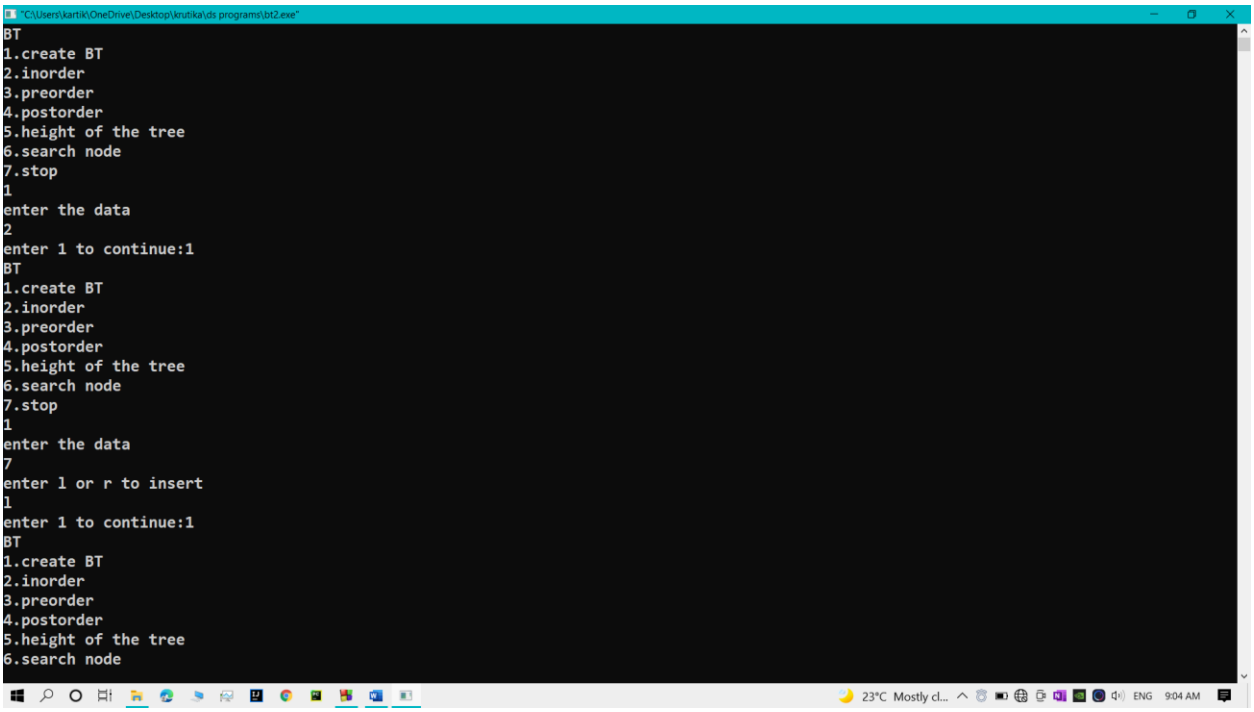
```
void insert(node* root,node* new)
{
char ch;
printf("enter l or r to insert\n");
scanf(" %c",&ch);
if(ch=='r'|| ch=='R')
{
  if(root->right==NULL)
   {
     root->right=new;
    }
else
   {
     insert(root->right,new);
    }
}
else
{
    if(root->left==NULL)
      {
       root->left=new;
       }
   else
      {
        insert(root->left,new);
       }
}
return;
}
/*================== inorder =================
Traverse the binary tree in left-node-right sequence
Pre :root is entry node of tree or subtree.
Post:each node has been processed in order.
*/
void inorder(node* temp)
{
if(temp!=NULL)
{
inorder(temp->left);
printf("%d",temp->data);
```

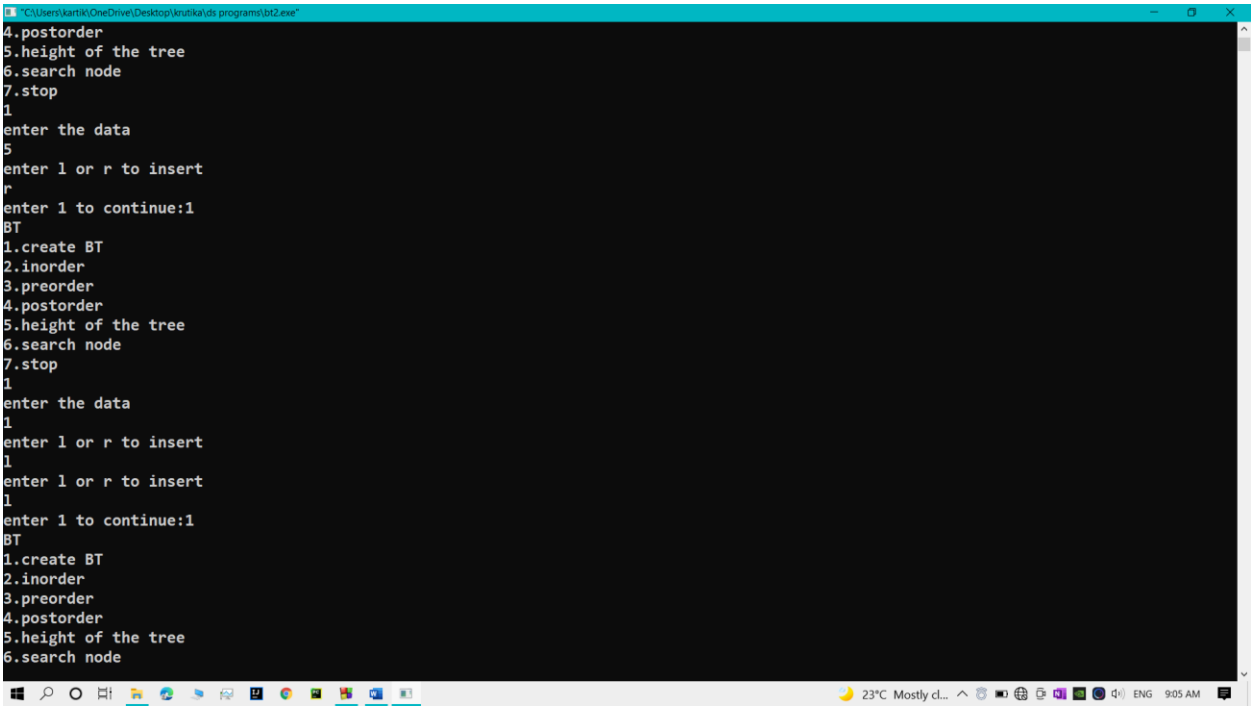
```
inorder(temp->right);
}
return;
}

/*================== preorder =================
Traverse the binary tree in node-left-right sequence
Pre :root is entry node of tree or subtree.
Post:each node has been processed in order.
*/
void preorder(node* temp)
{
if(temp!=NULL)
{
printf("%d",temp->data);
preorder(temp->left);
preorder(temp->right);
}
return;
}

/*================== postorder =================
Traverse the binary tree in left-right-node sequence
Pre :root is entry node of tree or subtree.
Post:each node has been processed in order.
*/
void postorder(node* temp)
{
if(temp!=NULL)
{
postorder(temp->left);
postorder(temp->right);
printf("%d",temp->data);
}
return;
}

/*================== height =================
Returns the height of a tree
Pre :root is entry node of tree or subtree.
```

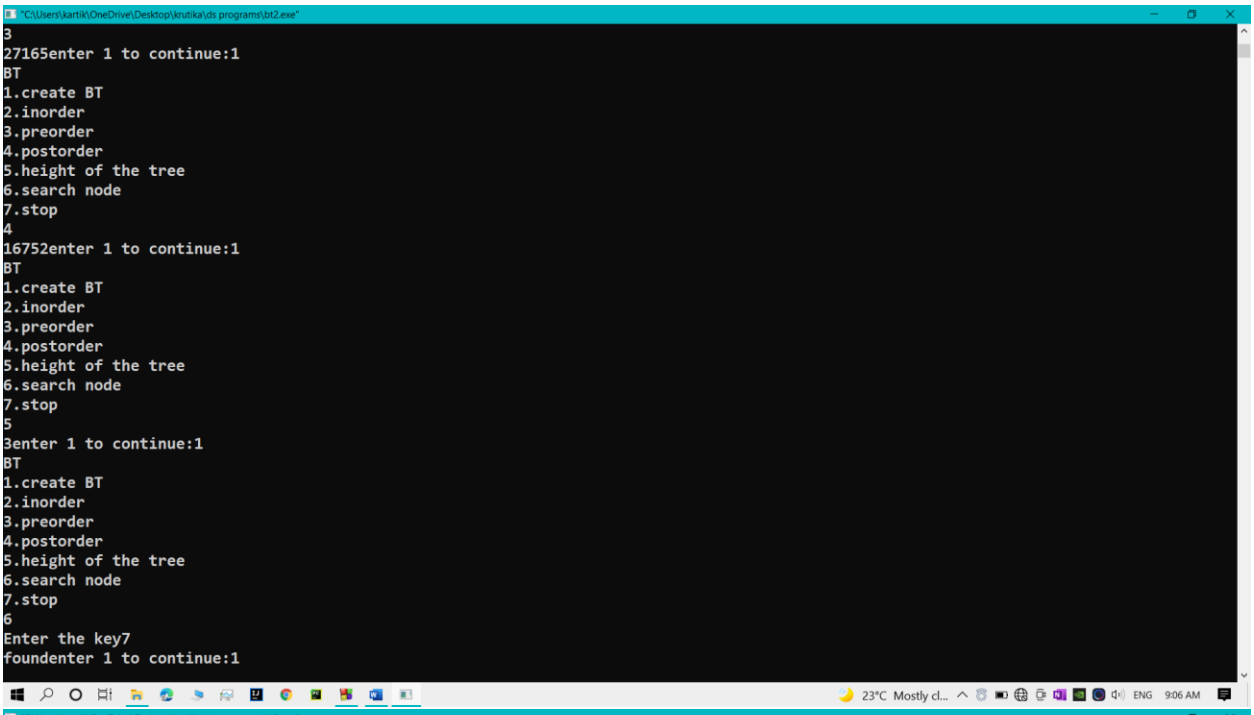
```
Post:height value is returned.
*/
int height(node* root)
{
if(root==NULL)
return(0);
int left=height(root->left);
int right=height(root->right);
if(left>right)
   return(left+1);
else
   return(right+1);
}
/*================== search node =================
search the node in the tree
Pre :root is entry node of tree or subtree and key value is passed to search
Post:returns found if node is present.
     returns not found if node is not present.
*/
bool searchnode(node* root,int key)
{
 if(root==NULL)
   return(false);
if(root->data==key)
   return(true);
bool res1=searchnode(root->left,key);
if(res1)
   return(true);
bool res2=searchnode(root->right,key);
return(res2);
}

int main()
{
int ch,op,h,key;
do
{
printf("BT\n");
printf("1.create BT\n");
printf("2.inorder\n");
```

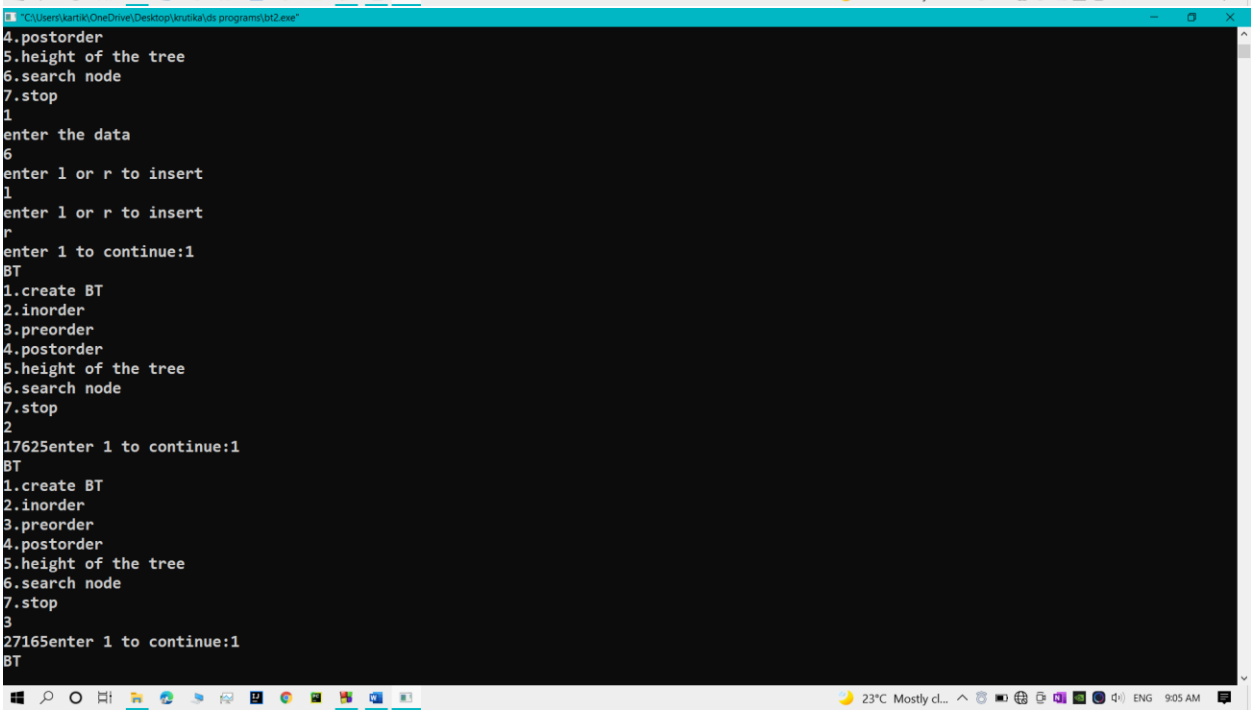
```
printf("3.preorder\n");
printf("4.postorder\n");
printf("5.height of the tree\n");
printf("6.search node\n");
printf("7.stop\n");
scanf("%d",&ch);
switch(ch)
{
case 1:create();
          break;
case 2:inorder(root);
          break;
case 3:preorder(root);
         break;
case 4:postorder(root);
         break;
case 5:h=height(root);
         printf("%d",h);
         break;
case 6:printf("Enter the key");
        scanf("%d",&key);
        if(searchnode(root,key))
             printf("found");
        else
             printf("not found");
      break;
default:exit(0);
           break;
}
printf("enter 1 to continue:");
scanf("%d",&op);
}while(op==1);
return(0);
}
```

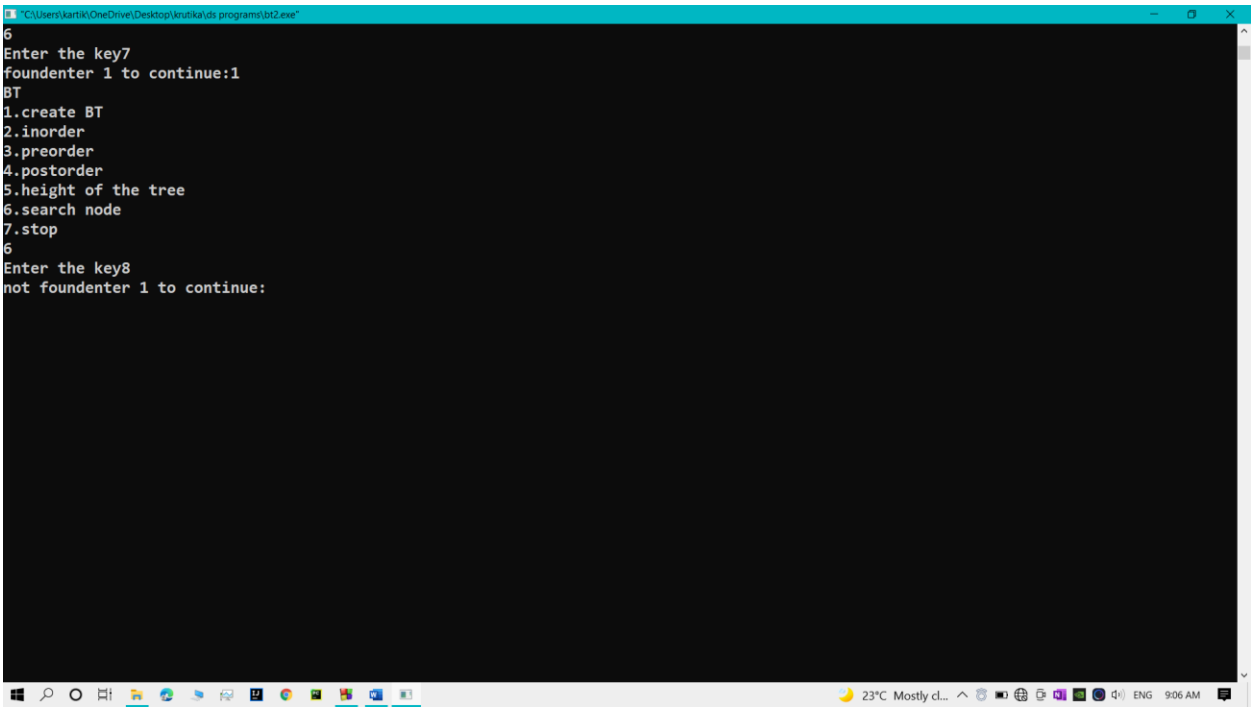
OUTPUT:


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\bt2.exe"
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
7.stop
1
enter the data
2
enter 1 to continue:1
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
7.stop
1
enter the data
7
enter l or r to insert
l
enter 1 to continue:1
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
23°C Mostly cl...
ENG
9:04 AM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\bt2.exe"
4.postorder
5.height of the tree
6.search node
7.stop
1
enter the data
5
enter l or r to insert
r
enter 1 to continue:1
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
7.stop
1
enter the data
1
enter l or r to insert
l
enter l or r to insert
l
enter 1 to continue:1
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
23°C Mostly cl...
ENG
9:05 AM

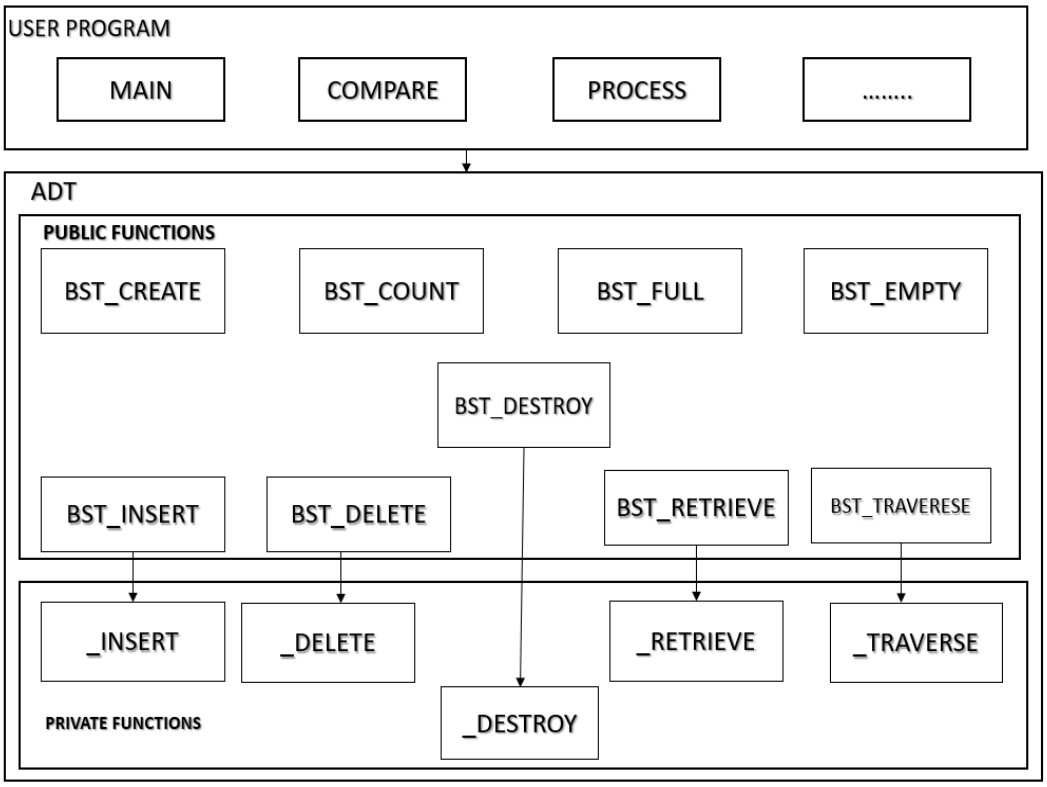

"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\bt2.exe"
3
27165enter 1 to continue:1
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
7.stop
4
16752enter 1 to continue:1
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
7.stop
5
3enter 1 to continue:1
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
7.stop
6
Enter the key7
foundenter 1 to continue:1
23°C Mostly cl...
ENG
9:06 AM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\bt2.exe"
4.postorder
5.height of the tree
6.search node
7.stop
1
enter the data
6
enter l or r to insert
1
enter l or r to insert
r
enter 1 to continue:1
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
7.stop
2
17625enter 1 to continue:1
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
7.stop
3
27165enter 1 to continue:1
BT
23°C Mostly cl...
ENG
9:05 AM


"C:\Users\kartik\OneDrive\Desktop\krutika\ds programs\bt2.exe"
6
Enter the key7
foundenter 1 to continue:1
BT
1.create BT
2.inorder
3.preorder
4.postorder
5.height of the tree
6.search node
7.stop
6
Enter the key8
not foundenter 1 to continue:
23°C Mostly cl...
ENG
9:06 AM

ASSIGNMENT 10

Write a generic C program to implement BST ADT (bst.h) for the following BST operations: i) Create BST ii) Insert BST iii) Delete a BST vi) Retrieve a BST v) Traverse a BST vi) Empty a BST vii) Full BST viii) BST Count ix) Destroy a BST. Write an application program (bstmain.c) to create an integer BST and perform all the Stack operations defined in the BST ADT .



bst.h

```
/*PERFORMS THE BST OPERATIONS*/
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
// Structure Declarations
typedef struct node
{
 void* dataPtr;
 struct node* left;
 struct node* right;
} NODE;
```

```
typedef struct
{
 int count;
 int (*compare) (void* argu1, void* argu2);
 NODE* root;
} BST_TREE;
// Prototype Declarations
BST_TREE* BST_Create
 (int (*compare) (void* argu1, void* argu2));
BST_TREE* BST_Destroy (BST_TREE* tree);
bool BST_Insert (BST_TREE* tree, void* dataPtr);
bool BST_Delete (BST_TREE* tree, void* dltKey);
void* BST_Retrieve (BST_TREE* tree, void* keyPtr);
void BST_Traverse (BST_TREE* tree,
 void (*process)(void* dataPtr));
bool BST_Empty (BST_TREE* tree);
bool BST_Full (BST_TREE* tree);
int BST_Count (BST_TREE* tree);
static NODE* _insert  (BST_TREE* tree, NODE* root, NODE* newPtr);
static NODE* _delete  (BST_TREE* tree, NODE* root, void* dataPtr, bool* success);
static void* _retrieve  (BST_TREE* tree,  void* dataPtr, NODE* root);
static void _traverse  (NODE* root, void (*process) (void* dataPtr));
static void _destroy (NODE* root);

/* ================= BST_Create ================
Allocates dynamic memory for an BST tree head node and returns its address to caller
 Pre: compare is address of compare function  used when two nodes need to be compared
 Post: head allocated or error returned
 Return head node pointer; null if overflow
*/
BST_TREE* BST_Create(int (*compare) (void* argu1, void* argu2))
{// Local Definitions
  BST_TREE* tree;
// Statements
  tree = (BST_TREE*) malloc (sizeof (BST_TREE));
  if (tree)
   { tree->root = NULL;
     tree->count = 0;
     tree->compare = compare;
   } // if
return tree;
} // BST_Create
```

```
/* ================= BST_Insert ==================
This function inserts new data into the tree.
 Pre: tree is pointer to BST tree structure
 Post: data inserted or memory overflow
 Return Success (true) or Overflow (false)*/
bool BST_Insert (BST_TREE* tree, void* dataPtr)
{
// Local Definitions
NODE* newPtr;
// Statements
newPtr = (NODE*) malloc (sizeof(NODE));
if (!newPtr)
 return false;
newPtr->right = NULL;
newPtr->left = NULL;
newPtr->dataPtr = dataPtr;
 if (tree->count == 0)
   tree->root = newPtr;
else
 _insert(tree, tree->root, newPtr);

(tree->count)++;
return true;
} // BST_Insert

/* ================== _insert ==================
This function uses recursion to insert the new data into a leaf node in the BST tree.
 Pre: Application has called BST_Insert, which  passes root and data pointer
 Post:Data have been inserted
 Return pointer to [potentially] new root*/
 NODE* _insert (BST_TREE* tree, NODE* root, NODE* newPtr)
{
// Statements
if (!root)
 // if NULL tree
 return newPtr;
// Locate null subtree for insertion
if (tree->compare(newPtr->dataPtr, root->dataPtr) < 0)
 {
            root->left = _insert(tree, root->left, newPtr);
    return root;
 } // new < node
else
 // new data >= root data
```

```
 {
 root->right = _insert(tree, root->right, newPtr);
 return root;
 } // else new data >= root data
return root;
} // _insert
/* ================== BST_Delete ==================
This function deletes a node from the tree and rebalances it if necessary.
 Pre: tree initialized--null tree is OK;dltKey is pointer to data structure  containing key to be deleted
 Post: node deleted and its space recycled  -or- An error code is returned
 Return Success (true) or Not found (false)
*/
bool BST_Delete (BST_TREE* tree, void* dltKey)
{
// Local Definitions
bool success;
NODE* newRoot;
// Statements
newRoot = _delete (tree, tree->root, dltKey, &success);
if (success)
 { tree->root = newRoot;
 (tree->count)--;
 if (tree->count == 0)
 // Tree now empty
 tree->root = NULL;
 } // if
return success;
} // BST_Delete

/* ==================== _delete ====================
Deletes node from the tree and rebalances tree if necessary.
 Pre: tree initialized--null tree is OK; dataPtr contains key of node to be deleted
 Post: node is deleted and its space recycled -or- if key not found, tree is unchanged
 success is true if deleted; false if not  Return pointer to root
*/
NODE* _delete (BST_TREE* tree, NODE* root, void* dataPtr, bool* success)
{
// Local Definitions
NODE* dltPtr;
NODE* exchPtr;
NODE* newRoot;
void* holdPtr;
// Statements
if (!root)
```

```
 { *success = false;
 return NULL;
 } // if
if (tree->compare(dataPtr, root->dataPtr) < 0)
 root->left = _delete (tree, root->left,
 dataPtr, success);
else if (tree->compare(dataPtr, root->dataPtr) > 0)
 root->right = _delete (tree, root->right,
 dataPtr, success);
else
 // Delete node found--test for leaf node
 {
 dltPtr = root;
 if (!root->left)
 // No left subtree
 {
 free (root->dataPtr); // data memory
 newRoot = root->right;
 free (dltPtr); // BST Node
 *success = true;
 return newRoot; // base case
 } // if true
 else
 if (!root->right)
 // Only left subtree
 {
 newRoot = root->left;
 free (dltPtr);
 *success = true;
 return newRoot; // base case
 } // if
 else
 // Delete Node has two subtrees
 {
exchPtr = root->left;
 // Find largest node on left subtree
 while (exchPtr->right)
 exchPtr = exchPtr->right;
 // Exchange Data
 holdPtr = root->dataPtr;
 root->dataPtr = exchPtr->dataPtr;
 exchPtr->dataPtr = holdPtr;
 root->left =
 _delete (tree, root->left,
```

```
 exchPtr->dataPtr, success);
 } // else
 } // node found
return root;
} // _delete

/* ================== BST_Retrieve ==================
Retrieve node searches tree for the node containing the requested key and returns pointer to its data.
 Pre: Tree has been created (may be null)  data is pointer to data structure  containing key to be located
 Post: Tree searched and data pointer returned
 Return Address of matching node returned
 If not found, NULL returned
*/
void* BST_Retrieve (BST_TREE* tree, void* keyPtr)
{
// Statements
if (tree->root)
 return _retrieve (tree, keyPtr, tree->root);
else
 return NULL;
} // BST_Retrieve
/* ================== _retrieve ==================
Searches tree for node containing requested key and returns its data to the calling function.
 Pre: _retrieve passes tree, dataPtr, root  dataPtr is pointer to data structure  containing key to be located
 Post: tree searched; data pointer returned
 Return Address of data in matching node
 If not found, NULL returned
*/
void* _retrieve (BST_TREE* tree,  void* dataPtr, NODE* root)
{
// Statements
if (root)
{
 if (tree->compare(dataPtr, root->dataPtr) < 0)
 return _retrieve(tree, dataPtr, root->left);
 else if (tree->compare(dataPtr,  root->dataPtr) > 0)
 return _retrieve(tree, dataPtr, root->right);
 else
 // Found equal key
 return root->dataPtr;
 } // if root
else
 // Data not in tree
 return NULL;
```

```
} // _retrieve

/* ================== BST_Traverse ==================
Process tree using inorder traversal.
 Pre: Tree has been created (may be null)  process "visits" nodes during traversal
 Post: Nodes processed in LNR (inorder) sequence
*/
void BST_Traverse (BST_TREE* tree, void (*process) (void* dataPtr))
{
// Statements
_traverse (tree->root, process);
return;
} // end BST_Traverse

/* ================== _traverse ==================
Inorder tree traversal. To process a node, we use the function passed when traversal was called.
 Pre: Tree has been created (may be null)
 Post: All nodes processed
*/
void _traverse (NODE* root, void (*process) (void* dataPtr))
{
// Statements
if (root)
 {
 _traverse (root->left, process);
 process (root->dataPtr);
 _traverse (root->right, process);
 } // if
return;
} // _traverse

/* ================== BST_Empty ==================
Returns true if tree is empty; false if any data.
 Pre: Tree has been created. (May be null)
 Returns True if tree empty, false if any data
*/
bool BST_Empty (BST_TREE* tree)
{
// Statements
return (tree->count == 0);
} // BST_Empty
/* ================== BST_Full ==================
If there is no room for another node, returns true.
 Pre: tree has been created
```

```
 Returns true if no room for another insert  false if room
*/
bool BST_Full (BST_TREE* tree)
{
// Local Definitions
NODE* newPtr;
// Statements
newPtr = (NODE*)malloc(sizeof (*(tree->root)));
if (newPtr)
 {
 free (newPtr);
 return false;
 } // if
else
 return true;
} // BST_Full

/* ================== BST_Count ==================
Returns number of nodes in tree.
 Pre: tree has been created
 Returns tree count
*/
int BST_Count (BST_TREE* tree)
{
// Statements
return (tree->count);
} // BST_Count
/* =============== BST_Destroy ==============
Deletes all data in tree and recycles memory.The nodes are deleted by calling a recursive function to
traverse the tree in inorder sequence.
 Pre: tree is a pointer to a valid tree
 Post: All data and head structure deleted
 Return null head pointer
*/
BST_TREE* BST_Destroy (BST_TREE* tree)
{
// Statements
if (tree)
_destroy (tree->root);
// All nodes deleted. Free structure
free (tree);
return NULL;
} // BST_Destroy
```

```
/* =============== _destroy ==============
Deletes all data in tree and recycles memory. It also recycles memory for the key and data nodes.
The nodes are deleted by calling a recursive function to traverse the tree in inorder sequence.
Pre: root is pointer to valid tree/subtree
 Post: All data and head structure deleted
 Return null head pointer
*/
void _destroy (NODE* root)
{
// Statements
if (root)
 {
 _destroy (root->left);
 free (root->dataPtr);
 _destroy (root->right);
 free (root);
 } // if
return;
} // _destroy

/*================== compare =================
Compare the two integers and returns the result.
Pre : This function takes two integers as arguments
Post: returns  -1  if integer1 less than integer 2.
       returns +1  if integer1 is greater than integer2.
       returns   0   if integer1 is equal to integer2.
*/
int compare(void* pargu1,void* pargu2)
{
          int i1,i2;
          int result;
          i1=(*(int*)pargu1);
          i2=(*(int*)pargu2);
          if(i1<i2)
           result = -1;
          else if(i1>i2)
           result = +1;
          else
           result = 0;
          return result;
}
/*================== printBST ================
Print one integer from BST
Pre : num1 is a pointer to integer
```

Post: integer printed and line advanced

*/

```
void  printBST(void* num1)
{
// Statements
 printf("%d\n",*(int*)num1);
 return;
}//printBST
```

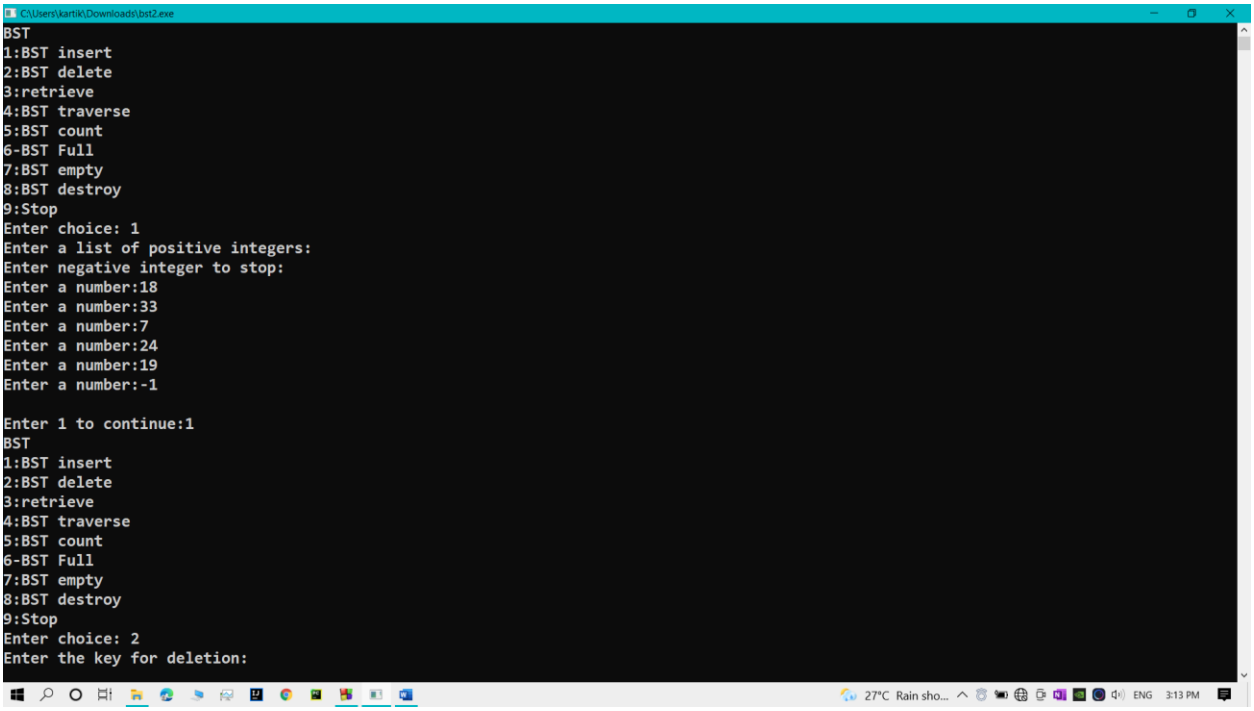
bst.c

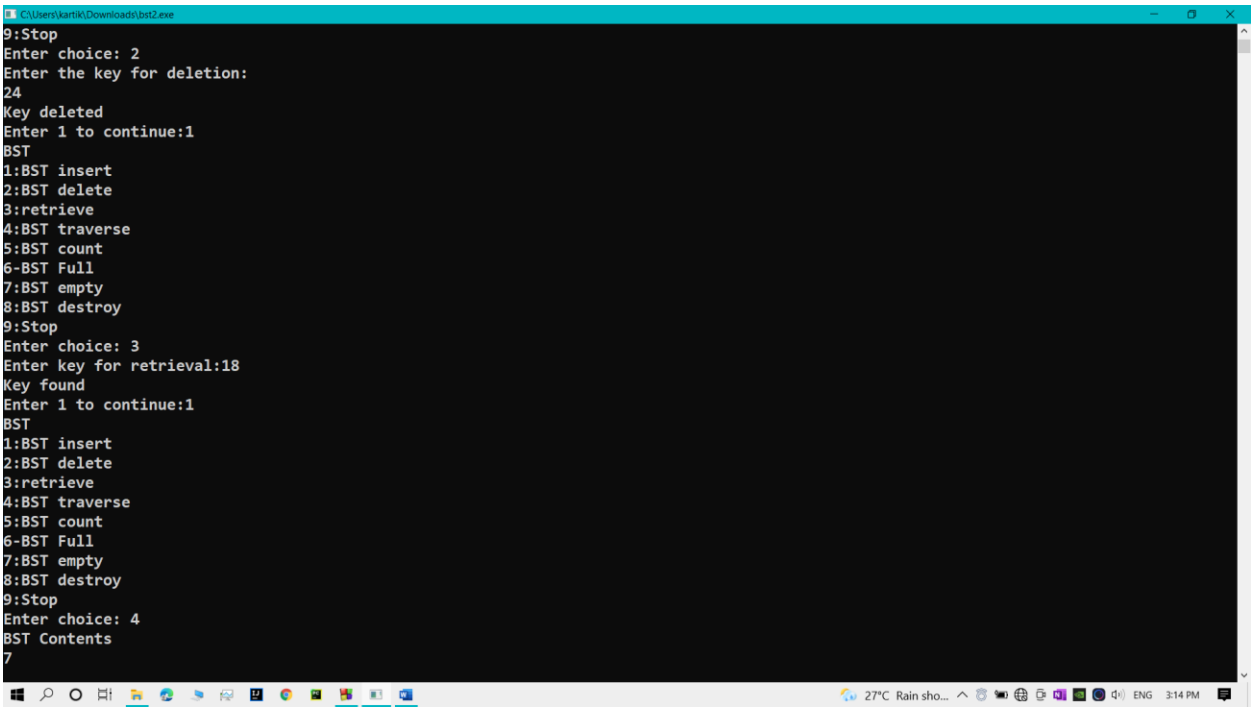
```
#include<stdio.h>
#include<stdlib.h>
#include "bst22.h"

int main()
{
  BST_TREE *BSTROOT;
  int *dataptr;

  BST_TREE *t;
  int c;
  int dataIn=+1;
  int ch,op,Key;
  BSTROOT=BST_Create(compare);
do
 {
   printf("BST\n");
   printf("1:BST insert\n");
   printf("2:BST delete\n");
   printf("3:retrieve\n");
   printf("4:BST traverse\n");
   printf("5:BST count\n");
   printf("6-BST Full\n");
   printf("7:BST empty\n");
   printf("8:BST destroy\n");
   printf("9:Stop\n");
   printf("Enter choice: ");
   scanf("%d",&ch);

    switch(ch)
   {
    case 1: //Build tree
              printf("Enter a list of positive integers:\n");
              printf("Enter negative integer to stop:\n");
     do
            {
             printf("Enter a number:");
             scanf("%d",&dataIn);
             if(dataIn>-1)
              {
               dataptr=(int*)malloc(sizeof(int));
               if(!dataptr)
               {
```

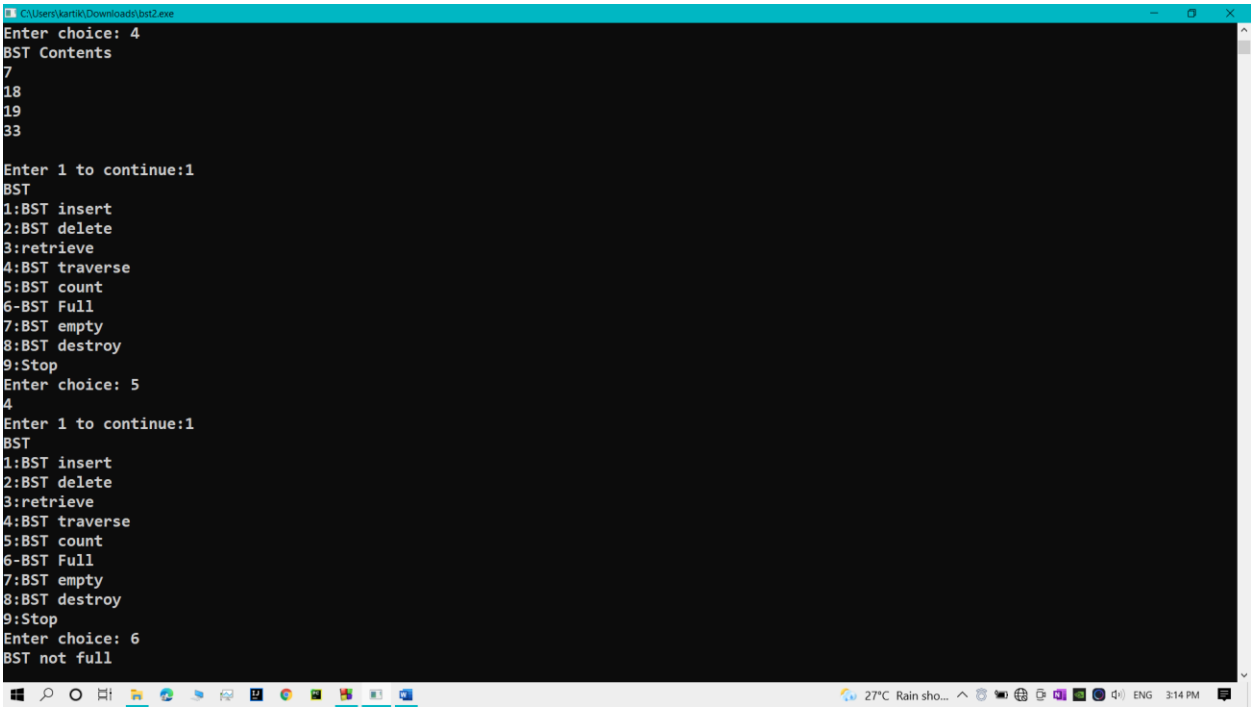
```
            printf("Memory overflow\n");
            exit(100);
           }
           *dataptr=dataIn;
           BST_Insert(BSTROOT,dataptr);
          }
         }while(dataIn>-1);
         break;
case 2: printf("Enter the key for deletion:\n");
        scanf("%d",&Key);
        if(BST_Delete(BSTROOT,&Key))
          printf("Key deleted");
        else
          printf("Key not deleted");
        break;
case 3: printf("Enter key for retrieval:");
        scanf("%d",&Key);
        if(BST_Retrieve(BSTROOT,&Key))
        {
          printf("Key found");
         }
        break;

case 4:printf("BST Contents \n");
        BST_Traverse(BSTROOT,printBST);
        break;
case 5: c=BST_Count(BSTROOT);
         printf("%d",c);
        break;
case 6:if(BST_Full(BSTROOT))
           printf("BST full");
         else
           printf("BST not full");
         break;
case 7:if(BST_Empty(BSTROOT))
          printf("BST empty");
        else
          printf("BST not empty");
        break;
case 8:t=BST_Destroy(BSTROOT);
        if(t==NULL)
          printf("BST destroyed");
        else
          printf("BST not destroyed");
```

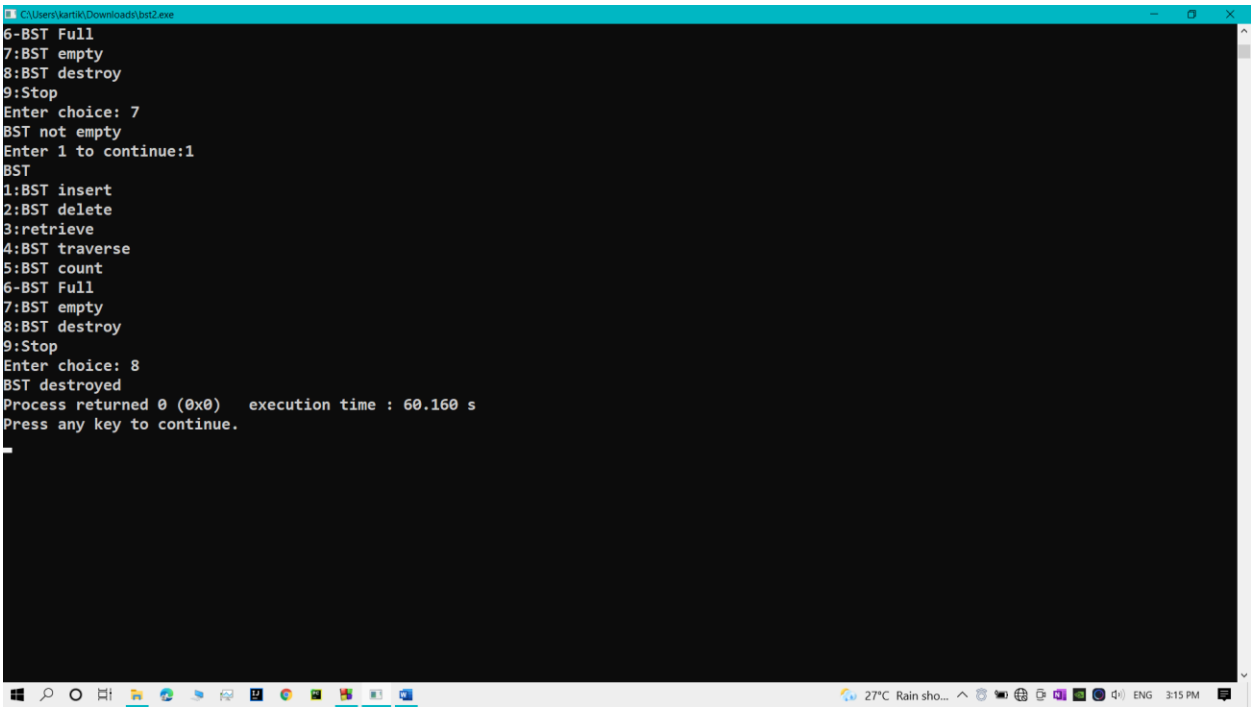
```
    default: exit(0);
   }
            printf("\nEnter 1 to continue:");
           scanf ("%d",&op);
}while(op == 1);
 return (0);
  }
```

OUTPUT:




C:\Users\kartik\Downloads\bst2.exe
9:Stop
Enter choice: 2
Enter the key for deletion:
24
Key deleted
Enter 1 to continue:1
BST
1:BST insert
2:BST delete
3:retrieve
4:BST traverse
5:BST count
6-BST Full
7:BST empty
8:BST destroy
9:Stop
Enter choice: 3
Enter key for retrieval:18
Key found
Enter 1 to continue:1
BST
1:BST insert
2:BST delete
3:retrieve
4:BST traverse
5:BST count
6-BST Full
7:BST empty
8:BST destroy
9:Stop
Enter choice: 4
BST Contents
7


C:\Users\kartik\Downloads\bst2.exe
Enter choice: 4
BST Contents
7
18
19
33
Enter 1 to continue:1
BST
1:BST insert
2:BST delete
3:retrieve
4:BST traverse
5:BST count
6-BST Full
7:BST empty
8:BST destroy
9:Stop
Enter choice: 5
4
Enter 1 to continue:1
BST
1:BST insert
2:BST delete
3:retrieve
4:BST traverse
5:BST count
6-BST Full
7:BST empty
8:BST destroy
9:Stop
Enter choice: 6
BST not full


C:\Users\kartik\Downloads\bst2.exe
6-BST Full
7:BST empty
8:BST destroy
9:Stop
Enter choice: 7
BST not empty
Enter 1 to continue:1
BST
1:BST insert
2:BST delete
3:retrieve
4:BST traverse
5:BST count
6-BST Full
7:BST empty
8:BST destroy
9:Stop
Enter choice: 8
BST destroyed
Process returned 0 (0x0)   execution time : 60.160 s
Press any key to continue.
27°C Rain sho...
ENG
3:15 PM