1. **SOURCE CODE:**

#include <stdio.h>

int main() {

int arr[50], num, temp, i, j;

printf("Please, enter the total no. you want to enter: ");

scanf("%d", &num);

// Use a loop to enter the numbers

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

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

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

}

for (i = 0, j = num - 1; i < num / 2; i++, j--) {

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

printf("\nReverse all elements of the array:\n");

// Use a loop to print the reversed array

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

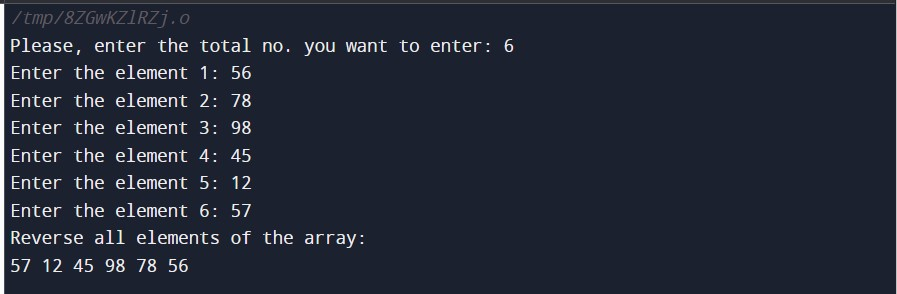
printf("%d ", arr[i]);

}

return 0;

}

**OUTPUT:**



1. **SOURCE CODE:**

**#include <stdio.h>**

**void searchValue(int arr[], int size, int value) {**

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

**if (arr[i] == value) {**

**printf("Value present at index no: %d\n", i + 1);**

**return;**

**}**

**}**

**printf("Value does not exist\n");**

**}**

**int main() {**

**int arr[] = {1, 2, 3, 4, 5, 6, 7};**

**int n = 7;**

**int val;**

**printf("Enter the value you want to search: ");**

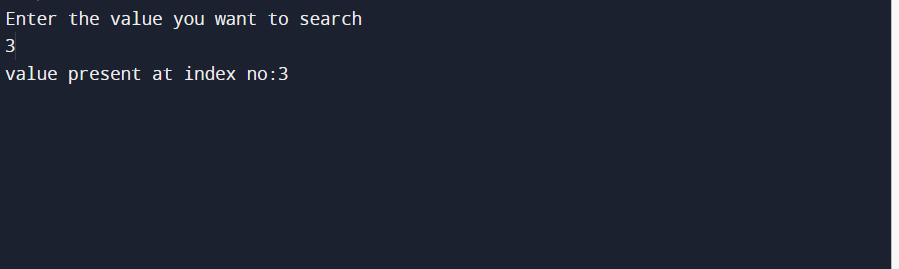
**scanf("%d", &val);**

**searchValue(arr, n, val);**

**return 0;**

**}**

**OUTPUT:**



1. **SOURCE CODE:**

#include <stdio.h>

int main() {

int n, num[50], largest, second;

printf("Enter number of elements: ");

scanf("%d", &n);

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

printf("Enter Array Element %d: ", i + 1);

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

}

/\* Here we are comparing first two elements of the

\* array, and storing the largest one in the variable

\* "largest" and the other one to "second" variable.

\*/

if (num[0] < num[1]) {

largest = num[1];

second = num[0];

} else {

largest = num[0];

second = num[1];

}

for (int i = 2; i < n; i++) {

/\* If the current array element is greater than largest

\* then the largest is copied to "second" and the element

\* is copied to the "largest" variable.

\*/

if (num[i] > largest) {

second = largest;

largest = num[i];

}

/\* If the current array element is less than largest but greater

\* than second largest ("second" variable) then copy the

\* element to "second"

\*/

else if (num[i] > second && num[i] != largest) {

second = num[i];

}

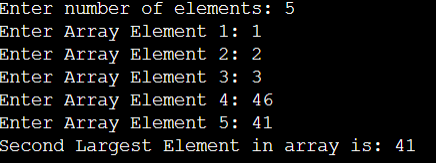
}

printf("Second Largest Element in array is: %d\n", second);

return 0;

}

**OUTPUT:**

****

1. **SOURCE CODE:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

struct student

{

int rollno;

float cpi;

char name[100];

};

void main()

{

struct student s[100];

for(int i=0;i<10;i++)

{

printf("\nEnter name :");

scanf("%s",s[i].name);

printf("\nEnter rollno :");

scanf("%d",&s[i].rollno);

printf("\nEnter cpi :");

scanf("%f",&s[i].cpi);

}

for(int i=0;i<10;i++)

{

printf("Name :%s\n" , s[i].name);

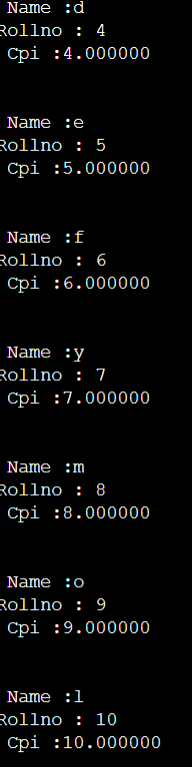
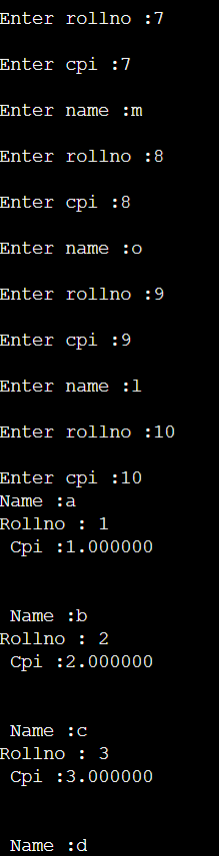
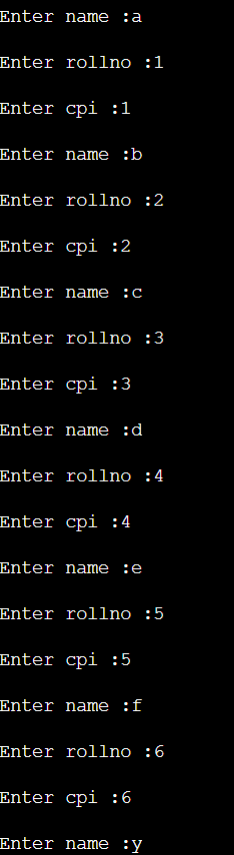
printf("Rollno : %d\n ", s[i].rollno);

printf("Cpi :%f\n\n\n ", s[i].cpi);

}

}

**OUTPUT:**

****

1. **SOURCE CODE:**

#include<stdio.h>

#include<conio.h>

struct student

{

char name[20];

int roll\_no;

float cpi;

};

void main(){

struct student s;

scanf("%s",s.name);

scanf("%d",&s.roll\_no);

scanf("%f",&s.cpi);

printf("%s\n",s.name);

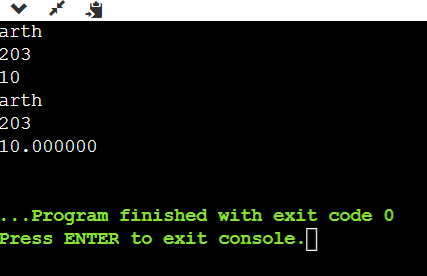
printf("%d\n",s.roll\_no);

printf("%f\n",s.cpi);

getch();

}

**OUTPUT:**

****

**VI. SOURCE CODE:**

#include<stdio.h>

int main() {

    int a=23;

    int \*p, \*\*q;

    p=&a;

    q=&p;

    printf("%d \n", &a);

    printf("%d \n", p);

    printf("%d \n", a);

    printf("%d \n", &p);

    printf("%d \n", \*p);

    printf("%d \n", q);

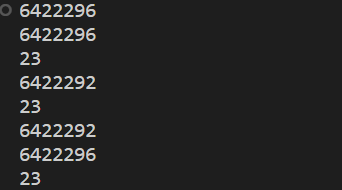
    printf("%d \n", \*q);

    printf("%d \n", \*\*q);

return 0;

}

**OUTPUT:**

****

**VII. SOURCE CODE:**

#include<stdio.h>

int main(){

    int a[5]={10,12,18,15,13};

    int \*p;

     p=a;

    for(int i=0;i<5;i++)

       {printf("%d \n", \*p);

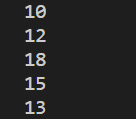
          p++;

    }

return 0;

}

**OUTPUT:**

****

**VIII. SOURCE CODE:**

#include<stdio.h>

struct Employee

{

char name[20];

int EID;

float exp;

};

void main()

{

  struct Employee \*e, k;

  e=&k;

  printf("name:");

scanf("%s",e->name);

printf("EID: ");

scanf("%d",&e->EID);

printf("EXP: ");

scanf("%f",&e->exp);

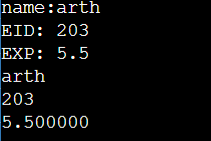
printf("%s\n",e->name);

printf("%d\n",e->EID);

printf("%f\n",e->exp);

}

**OUTPUT:**

****

**IX. SOURCE CODE:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

struct student

{

int roll;

float cpi;

char name[20];

};

void main()

{

struct student s[3];

struct student \*p;

p=s;

for(int i=0;i<3;i++)

{

scanf("%d",&s[i].roll);

scanf("%f",&s[i].cpi);

scanf("%s",s[i].name);

}

for(int i=0;i<3;i++)

{

printf("roll no : %d \n",p->roll);

printf("cpi : %f \n",p->cpi);

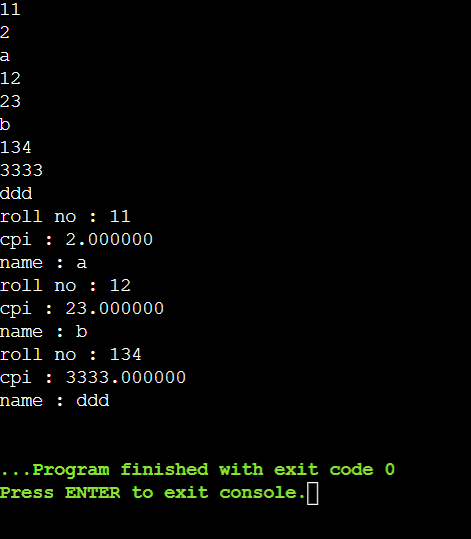
printf("name : %s \n",p->name);

p++;

}

}

**OUTPUT:**

****

**X SOURCE CODE:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

struct student

{

int roll;

float cpi;

char name[20];

};

void main()

{

struct student s,t;

struct student \*p[2];

p[0]=&s;

p[1]=&t;

for(int i=0;i<2;i++)

{

scanf("%d",&p[i]->roll);

scanf("%f",&p[i]->cpi);

scanf("%s",p[i]->name);

}

for(int i=0;i<2;i++)

{

printf("address : %d \n",p[i]);

printf("roll no : %d \n",p[i]->roll);

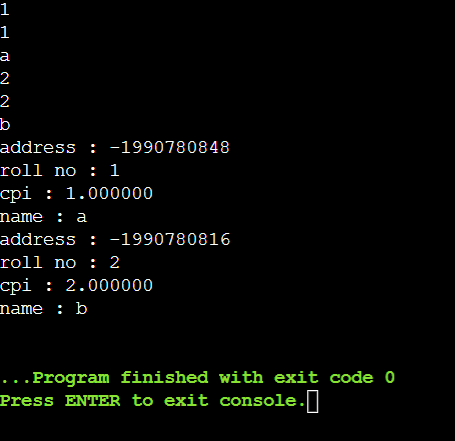
printf("cpi : %f \n",p[i]->cpi);

printf("name : %s \n",p[i]->name);

}

}

**OUTPUT:**

****

**XI SOURCE CODE:**

Write the C program to perform following tasks on a linear linked list:

1.Traversing

2.Inserting node at: a) start b) last

3. Delete node at: a) start b) last

**Source code:**

**#** include <stdio.h>

# include <stdlib.h>

// Defining the structure of node

struct Node

{

int data;

struct Node \*next;

};

// User Defined function for traversing the linked list

void Traverse(struct Node \*START)

{

if (START == NULL)

{

printf("Linear linked list is empty.\n");

return;

}

struct Node \*current = START;

while (current != NULL)

{

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

current = current->next;

}

}

// Function for inserting element at start

void InsertStart(struct Node \*\*START,int newData)

{

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

newNode->data = newData;

newNode->next = \*START;

\*START = newNode;

}

// Function for inserting element at end

void InsertEnd(struct Node\* START,int newData)

{

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

newNode->data = newData;

newNode->next = NULL;

struct Node \*current = START;

while (current->next != NULL)

{

current = current->next;

}

current->next = newNode;

}

// Function for deleting first node

void DeleteStart(struct Node \*START)

{

if (START == NULL)

{

printf("Underflow!");

}

\*START = \*START->next;

}

// Function for deleting last node

void DeleteEnd(struct Node \*START)

{

if (START == NULL)

{

printf("Underflow!");

}

struct Node \*current = START;

while ((current->next)->next != NULL)

{

current = current->next;

}

current->next = NULL;

}

void main()

{

// Making nodes

struct Node \*first, \*second, \*third, \*START;

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

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

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

// Assigning the values to the nodes

first->data = 10;

first->next = second;

second->data = 20;

second->next = third;

third->data = 30;

third->next = NULL;

START = first; // Pointing start to first node

printf("Initial link list: \n");

Traverse(START);

InsertStart(&START, 40);

printf("Link list after adding new node at start: \n");

Traverse(START);

InsertEnd(START, 50);

printf("Link list after adding new node at end: \n");

Traverse(START);

DeleteStart(START);

printf("Link list after deleting first node: \n");

Traverse(START);

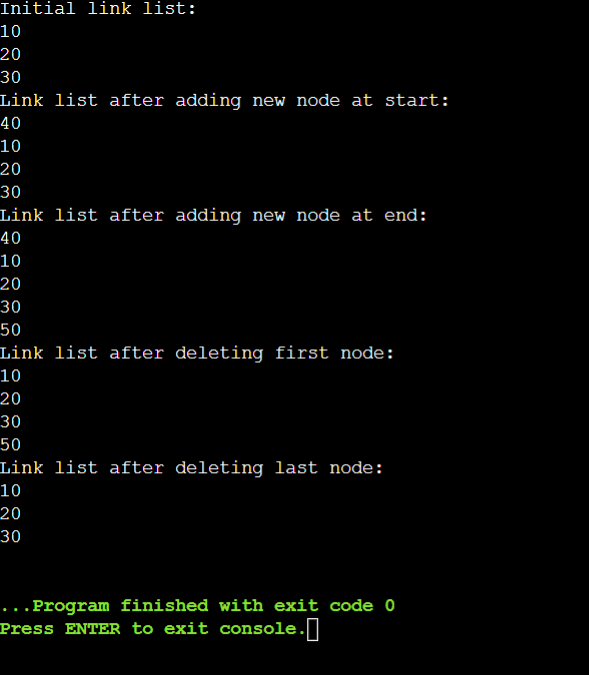
DeleteEnd(START);

printf("Link list after deleting last node: \n");

Traverse(START);

}

**OUTPUT:**

****

**XII SOURCE CODE:**

Write the C program to perform following tasks on a circular linked list

I) Traversing

II) Inserting node at: a) start b) last

III) Delete node at: a) start b) last

**SOURCE CODE:**

# include <stdio.h>

# include <stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

void Traverse(struct Node \*START)

{

if (START == NULL)

{

printf("Circular linked list is empty.\n");

return;

}

struct Node \*current = START;

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

current = current->next;

while (current != START)

{

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

current = current->next;

}

}

void InsertStart(struct Node \*\*START, int newData)

{

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

newNode->data = newData;

struct Node \*current = \*START;

while (current->next != \*START)

{

current = current->next;

}

current->next = newNode;

newNode->next = \*START;

\*START = newNode;

}

void InsertEnd(struct Node \*START, int newData)

{

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

newNode->data = newData;

struct Node \*current = START;

while (current->next != START)

{

current = current->next;

}

current->next = newNode;

newNode->next = START;

}

void DeleteStart(struct Node \*\*START)

{

if (\*START == NULL)

{

printf("Underflow!");

}

struct Node \*current = \*START;

while(current->next != \*START)

{

current = current->next;

}

current->next = (\*START)->next;

\*START = (\*START)->next;

}

void DeleteEnd(struct Node \*START)

{

if (START == NULL)

{

printf("Underflow!");

}

struct Node \*current = START->next;

struct Node \*SAVE = START;

while(current->next != START)

{

SAVE = current;

current = current->next;

}

SAVE->next = START;

}

void main()

{

struct Node \*first, \*second, \*third, \*START;

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

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

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

first->data = 1;

first->next = second;

second->data = 2;

second->next = third;

third->data = 3;

third->next = first;

START = first;

printf("Initial circular link list is: \n");

Traverse(START);

InsertStart(&START, 4);

printf("Circular Link list after adding new node at start: \n");

Traverse(START);

InsertEnd(START, 5);

printf("Circular Link list after adding new node at end: \n");

Traverse(START);

DeleteStart(&START);

printf("Circular Link list after deleting first node: \n");

Traverse(START);

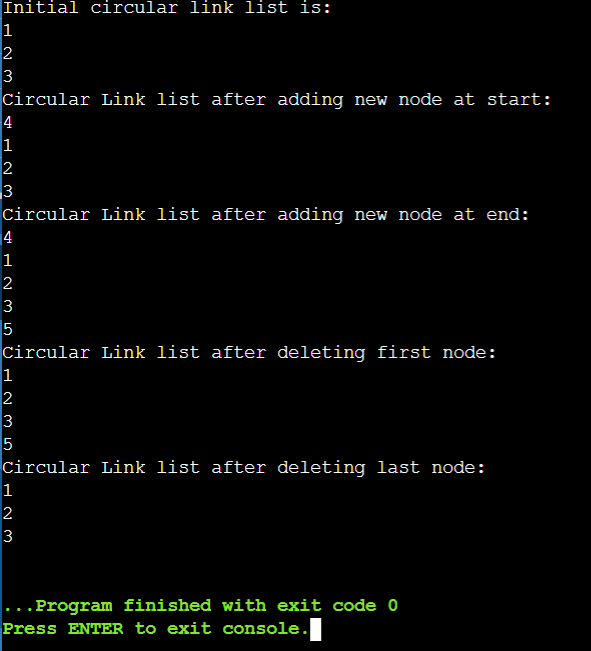
DeleteEnd(START);

printf("Circular Link list after deleting last node: \n");

Traverse(START);

}

**Output:**

****

**XIII SOURCE CODE:**

Write the C program to merge two linked list and the sort it in ascending order

**SOURCE CODE:**

# include <stdio.h>

# include <stdlib.h>

struct Node

{

int data;

struct Node \*next;

};

void Traverse(struct Node \*START)

{

struct Node \*current = START;

while (current != NULL)

{

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

current = current->next;

}

}

void Merge(struct Node \*START1, struct Node \*START2)

{

struct Node \*current = START1;

while (current->next != NULL)

{

current = current->next;

}

current->next = START2;

}

void Sort(struct Node \*START)

{

int swapped;

struct Node \*current = START;

do

{

swapped = 0;

current = START;

while (current->next != NULL)

{

if (current->data > (current->next)->data)

{

int tempData = current->data;

current->data = (current->next)->data;

(current->next)->data = tempData;

swapped = 1;

}

current = current->next;

}

} while (swapped);

}

void main()

{

struct Node \*first, \*second, \*third, \*fourth, \*START1, \*START2;

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

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

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

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

first->data = 80;

first->next = second;

second->data = 50;

second->next = NULL;

START1 = first;

third->data = 40;

third->next = fourth;

fourth->data = 20;

fourth->next = NULL;

START2 = third;

printf("First linked list is: \n");

Traverse(START1);

printf("Second linked list is: \n");

Traverse(START1);

printf("Merged linked list is: \n");

Merge(START1, START2);

Traverse(START1);

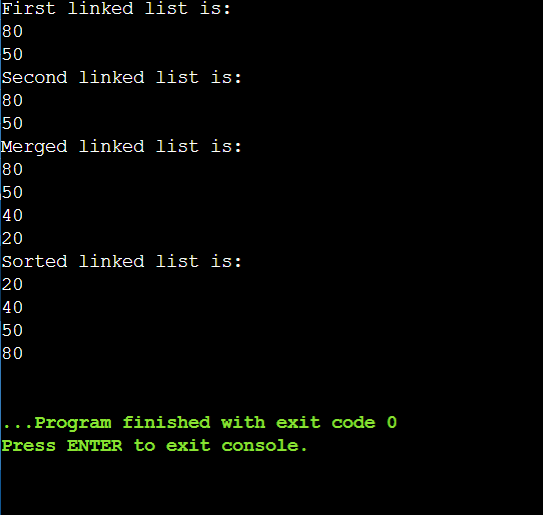
printf("Sorted linked list is: \n");

Sort(START1);

Traverse(START1);

}

**Output:**

****

**14) Push and Pop operations on stack.**

**Source Code :**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#include<limits.h>

// a stucture to represent a stack

struct Stack

{

int top;

unsigned capacity;

int\* array;

};

// FUNCTION TO CREATE A STACK OF GIVEN capacity. it initializes size of stack as 0

struct Stack\* createStack(unsigned capacity)

{

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

stack->capacity = capacity;

stack->top=-1;

stack->array = (int\*)malloc(stack->capacity \* sizeof(int));

return stack;

}

// stack is full when top is equal to the last index

int isFull(struct Stack\* stack)

{

return stack->top==stack->capacity-1;

}

// stack is empty when top is equal to -1

int isEmpty(struct Stack\* stack)

{

return stack->top==-1;

}

// function to add an item to stack.it increases top by -1

void push(struct Stack\* stack, int item)

{

if(isFull(stack))

return;

stack->array[++stack->top]=item;

printf("%d pushed to stack\n",item);

}

// function to remove an item from stack.it decreases top by -1

int pop(struct Stack\* stack)

{

if (isEmpty(stack))

return INT\_MIN;

return stack->array[stack->top--];

}

// function to return the top from stack without removing it

int peek(struct Stack\* stack)

{

if(isEmpty(stack))

return INT\_MIN;

return stack->array[stack->top];

}

// driver program to test above function

int main()

{

struct Stack\* stack = createStack(100);

push(stack,10);

push(stack,20);

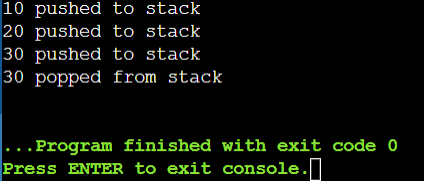
push(stack,30);

printf("%d popped from stack\n",pop(stack));

return 0;

}

**OUTPUT:**

****

**15) Program to check if a string is balanced or not.**

**Source code:**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

// Structure to represent a stack

struct Stack {

char data;

struct Stack\* next;

};

// Function to create a new stack node

struct Stack\* newNode(char data) {

struct Stack\* stackNode = (struct Stack\*)malloc(sizeof(struct Stack));

stackNode->data = data;

stackNode->next = NULL;

return stackNode;

}

// Function to check if the stack is empty

bool isEmpty(struct Stack\* root) {

return (root == NULL);

}

// Function to push a character onto the stack

void push(struct Stack\*\* root, char data) {

struct Stack\* stackNode = newNode(data);

stackNode->next = \*root;

\*root = stackNode;

}

// Function to pop a character from the stack

char pop(struct Stack\*\* root) {

if (isEmpty(\*root))

return '\0';

struct Stack\* temp = \*root;

\*root = (\*root)->next;

char popped = temp->data;

free(temp);

return popped;

}

// Function to check if the given string has balanced parentheses

bool isBalanced(char\* expression) {

struct Stack\* stack = NULL;

int length = strlen(expression);

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

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

push(&stack, expression[i]);

} else if (expression[i] == ')' || expression[i] == '}' || expression[i] == ']') {

if (isEmpty(stack))

return false;

char top = pop(&stack);

// Check if the popped character matches the current closing character

if ((expression[i] == ')' && top != '(') ||

(expression[i] == '}' && top != '{') ||

(expression[i] == ']' && top != '[')) {

return false;

}

}

}

return isEmpty(stack); // If the stack is empty at the end, it's balanced

}

int main() {

char expression[100];

printf("Enter a string containing parentheses: ");

fgets(expression, sizeof(expression), stdin);

if (isBalanced(expression))

printf("Balanced\n");

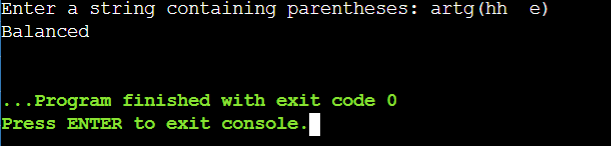
else

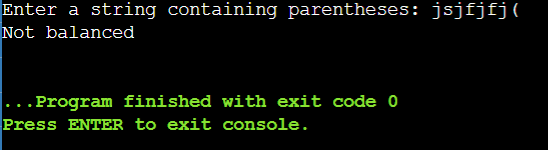
printf("Not balanced\n");

return 0;

}

**OUTPUT:**

****

****

**16) Program to reverse a string.**

**Source code :**

#include <stdio.h>

#include <string.h>

#define MAX 100 /\*maximum no. of characters\*/

/\*stack variables\*/

int top=-1;

int item;

/\*string declaration\*/

char stack\_string[MAX];

/\*function to push character (item)\*/

void pushChar(char item);

/\*function to pop character (item)\*/

char popChar(void);

/\*function to check stack is empty or not\*/

int isEmpty(void);

/\*function to check stack is full or not\*/

int isFull(void);

int main()

{

char str[MAX]; int i;

printf("Input a string: ");

scanf("%[^\n]s",str); /\*read string with spaces\*/

/\*gets(str);-can be used to read string with spaces\*/

for(i=0;i<strlen(str);i++) pushChar(str[i]);

for(i=0;i<strlen(str);i++) str[i]=popChar();

printf("Reversed String is: %s\n",str);

return 0;

}

/\*function definition of pushChar\*/

void pushChar(char item)

{

/\*check for full\*/

if(isFull())

{

printf("\nStack is FULL !!!\n"); return;

}

/\*increase top and push item in stack\*/

top=top+1;

stack\_string[top]=item;

}

/\*function definition of popChar\*/

char popChar()

{

/\*check for empty\*/

if(isEmpty())

{

printf("\nStack is EMPTY!!!\n"); return 0;

}

/\*pop item and decrease top\*/

item = stack\_string[top];

top=top-1; return item;

}

/\*function definition of isEmpty\*/

int isEmpty()

{

if(top==-1) return 1;

else

return 0;

}

/\*function definition of isFull\*/

int isFull()

{

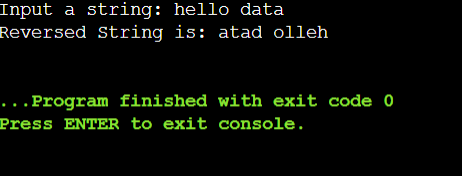
if(top==MAX-1) return 1;

else

return 0;

}

**OUTPUT:**



**17) Evaluation from infix to postfix.**

**Source code :**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

struct Stack {

int top;

unsigned capacity;

char\* array; // Changed data type to 'char\*' for characters

};

struct Stack\* createStack(unsigned capacity)

{

struct Stack\* stack = (struct Stack\*)malloc(sizeof(struct Stack));

if (!stack)

return NULL;

stack->top = -1;

stack->capacity = capacity;

stack->array = (char\*)malloc(stack->capacity \* sizeof(char));

return stack;

}

int isEmpty(struct Stack\* stack)

{

return stack->top == -1;

}

char peek(struct Stack\* stack)

{

return stack->array[stack->top];

}

char pop(struct Stack\* stack)

{

if (!isEmpty(stack))

return stack->array[stack->top--];

return '$';

}

void push(struct Stack\* stack, char op)

{

stack->array[++stack->top] = op;

}

int isOperand(char ch)

{

return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');

}

// A utility function to return precedence of a given operator

// Higher returned value means higher precedence

int Prec(char ch)

{

switch (ch) {

case '+':

case '-':

return 1;

case '\*':

case '/':

return 2;

case '^':

return 3;

}

return -1;

}

// The main function that converts given infix to postfix expression

void infixToPostfix(char\* exp)

{

int i, k;

struct Stack\* stack = createStack(strlen(exp));

if (!stack) // Check if stack was created successfully

{

printf("Stack creation failed.\n");

return;

}

for (i = 0, k = -1; exp[i]; ++i) {

// If the scanned character is an operand, add it to output

if (isOperand(exp[i]))

exp[++k] = exp[i];

// If the scanned character is an '(', push it to the stack

else if (exp[i] == '(')

push(stack, exp[i]);

// If the scanned character is an ')', pop and output from the stack

// until a '(' is encountered.

else if (exp[i] == ')') {

while (!isEmpty(stack) && peek(stack) != '(')

exp[++k] = pop(stack);

if (!isEmpty(stack) && peek(stack) != '(') {

printf("Invalid expression\n");

return;

} else

pop(stack);

} else // An operator is encountered

{

while (!isEmpty(stack) && Prec(exp[i]) <= Prec(peek(stack)))

exp[++k] = pop(stack);

push(stack, exp[i]);

}

}

while (!isEmpty(stack))

exp[++k] = pop(stack);

exp[++k] = '\0';

printf("%s\n", exp);

}

int main()

{

char exp[] = "a+b\*(c^d-e)^(f+g\*h)-i";

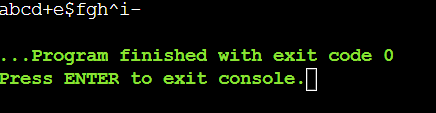
// Function call

infixToPostfix(exp);

return 0;

}

**OUTPUT:**



**18) postfix evaluation**

**Source code :**

**#include <ctype.h>**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**// Stack type**

**struct Stack {**

**int top;**

**unsigned capacity;**

**int\* array;**

**};**

**// Stack Operations**

**struct Stack\* createStack(unsigned capacity)**

**{**

**struct Stack\* stack**

**= (struct Stack\*)malloc(sizeof(struct Stack));**

**if (!stack)**

**return NULL;**

**stack->top = -1;**

**stack->capacity = capacity;**

**stack->array**

**= (int\*)malloc(stack->capacity \* sizeof(int));**

**if (!stack->array)**

**return NULL;**

**return stack;**

**}**

**int isEmpty(struct Stack\* stack)**

**{**

**return stack->top == -1;**

**}**

**char peek(struct Stack\* stack)**

**{**

**return stack->array[stack->top];**

**}**

**char pop(struct Stack\* stack)**

**{**

**if (!isEmpty(stack))**

**return stack->array[stack->top--];**

**return '$';**

**}**

**void push(struct Stack\* stack, char op)**

**{**

**stack->array[++stack->top] = op;**

**}**

**// The main function that returns value**

**// of a given postfix expression**

**int evaluatePostfix(char\* exp)**

**{**

**// Create a stack of capacity equal to expression size**

**struct Stack\* stack = createStack(strlen(exp));**

**int i;**

**// See if stack was created successfully**

**if (!stack)**

**return -1;**

**// Scan all characters one by one**

**for (i = 0; exp[i]; ++i) {**

**// If the scanned character is an operand**

**// (number here), push it to the stack.**

**if (isdigit(exp[i]))**

**push(stack, exp[i] - '0');**

**// If the scanned character is an operator,**

**// pop two elements from stack apply the operator**

**else {**

**int val1 = pop(stack);**

**int val2 = pop(stack);**

**switch (exp[i]) {**

**case '+':**

**push(stack, val2 + val1);**

**break;**

**case '-':**

**push(stack, val2 - val1);**

**break;**

**case '\*':**

**push(stack, val2 \* val1);**

**break;**

**case '/':**

**push(stack, val2 / val1);**

**break;**

**}**

**}**

**}**

**return pop(stack);**

**}**

**// Driver code**

**int main()**

**{**

**char exp[] = "231\*+9-";**

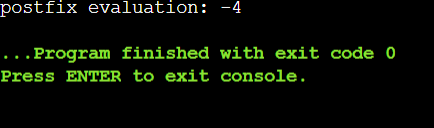
**// Function call**

**printf("postfix evaluation: %d", evaluatePostfix(exp));**

**return 0;**

**}**

**OUTPUT:**



**19) QUEUE**

**Source code :**

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

// A structure to represent a queue

struct Queue {

int front, rear, size;

unsigned capacity;

int\* array;

};

// function to create a queue

// of given capacity.

// It initializes size of queue as 0

struct Queue\* createQueue(unsigned capacity)

{

struct Queue\* queue = (struct Queue\*)malloc(

sizeof(struct Queue));

queue->capacity = capacity;

queue->front = queue->size = 0;

// This is important, see the enqueue

queue->rear = capacity - 1;

queue->array = (int\*)malloc(

queue->capacity \* sizeof(int));

return queue;

}

// Queue is full when size becomes

// equal to the capacity

int isFull(struct Queue\* queue)

{

return (queue->size == queue->capacity);

}

// Queue is empty when size is 0

int isEmpty(struct Queue\* queue)

{

return (queue->size == 0);

}

// Function to add an item to the queue.

// It changes rear and size

void enqueue(struct Queue\* queue, int item)

{

if (isFull(queue))

return;

queue->rear = (queue->rear + 1)

% queue->capacity;

queue->array[queue->rear] = item;

queue->size = queue->size + 1;

printf("%d enqueued to queue\n", item);

}

// Function to remove an item from queue.

// It changes front and size

int dequeue(struct Queue\* queue)

{

if (isEmpty(queue))

return INT\_MIN;

int item = queue->array[queue->front];

queue->front = (queue->front + 1)

% queue->capacity;

queue->size = queue->size - 1;

return item;

}

// Function to get front of queue

int front(struct Queue\* queue)

{

if (isEmpty(queue))

return INT\_MIN;

return queue->array[queue->front];

}

// Function to get rear of queue

int rear(struct Queue\* queue)

{

if (isEmpty(queue))

return INT\_MIN;

return queue->array[queue->rear];

}

// Driver program to test above functions./

int main()

{

struct Queue\* queue = createQueue(1000);

enqueue(queue, 10);

enqueue(queue, 20);

enqueue(queue, 30);

enqueue(queue, 40);

printf("%d dequeued from queue\n\n",

dequeue(queue));

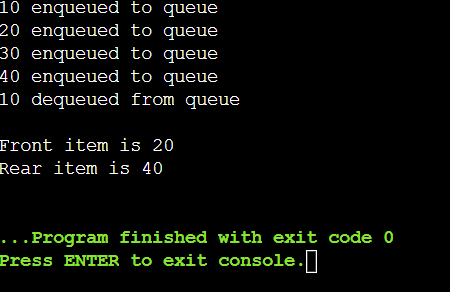
printf("Front item is %d\n", front(queue));

printf("Rear item is %d\n", rear(queue));

return 0;

}

**OUTPUT:**



**20) Preorder**

**Source code :**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct node

{

int data;

struct node\* left;

struct node\* right;

};

struct node\* createnode(int x){

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

root->data=x;

root->left=NULL;

root->right=NULL;

return root;

}

void preorder(struct node\* root)

{

if(root==NULL)

return;

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

preorder(root->left);

preorder(root->right);

}

void main()

{

struct node\* root=createnode(1);

root->left=createnode(2);

root->right=createnode(3);

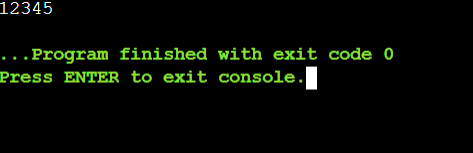
(root->right)->left=createnode(4);

(root->right)->right=createnode(5);

preorder(root);

}

**OUTPUT:**



**21) Inorder**

**Source code :**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct node

{

int data;

struct node\* left;

struct node\* right;

};

struct node\* createnode(int x){

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

root->data=x;

root->left=NULL;

root->right=NULL;

return root;

}

void inorder(struct node\* root)

{

if(root==NULL)

return;

inorder(root->left);

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

inorder(root->right);

}

void main()

{

struct node\* root=createnode(1);

root->left=createnode(2);

root->right=createnode(3);

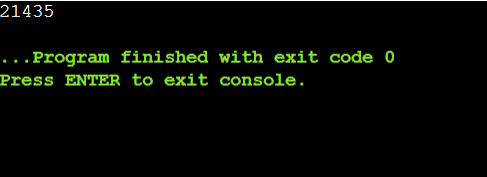
(root->right)->left=createnode(4);

(root->right)->right=createnode(5);

inorder(root);

}

**OUTPUT:**



**22) Postorder**

**Source code :**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct node

{

int data;

struct node\* left;

struct node\* right;

};

struct node\* createnode(int x){

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

root->data=x;

root->left=NULL;

root->right=NULL;

return root;

}

void postorder(struct node\* root)

{

if(root==NULL)

return;

postorder(root->left);

postorder(root->right);

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

}

void main()

{

struct node\* root=createnode(1);

root->left=createnode(2);

root->right=createnode(3);

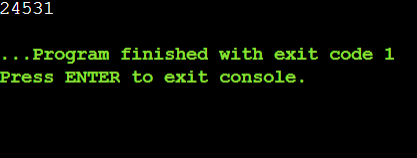
(root->right)->left=createnode(4);

(root->right)->right=createnode(5);

postorder(root);

}

**OUTPUT:**



**23) Searching**

**Source code :**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct node

{

int data;

struct node\* left;

struct node\* right;

};

struct node\* createnode(int x){

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

root->data=x;

root->left=NULL;

root->right=NULL;

return root;

}

void postorder(struct node\* root)

{

if(root==NULL)

return;

postorder(root->left);

postorder(root->right);

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

}

struct node\* search(struct node\* root, int data)

{

// Base Cases: root is null or key is present at root

if (root == NULL || root->data == data)

return root;

// Key is greater than root's key

if (root->data < data)

return search(root->right, data);

// Key is smaller than root's key

return search(root->left, data);

}

void main()

{

struct node\* root=createnode(1);

root->left=createnode(2);

root->right=createnode(3);

(root->right)->left=createnode(4);

(root->right)->right=createnode(5);

int data=10;

if (search(root,data) == NULL)

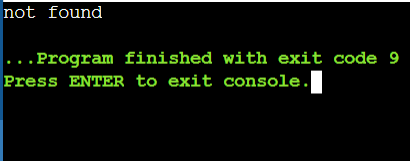
printf("not found");

else

printf("found");

}

**OUTPUT:**



**24) Implementing graph as adjacency list.**

**Source code :**

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

struct graph

{

//declaring graph data structure

struct vertex \*point[5];

};

struct vertex

{

//declaring vertices

int end;

struct vertex \*next;

};

struct Edge

{

//declaring edges

int end,start;

};

struct graph \*create\_graph(struct Edge edges[],int x)

{

int i;

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

for(i=0; i<5;i++)

{

graph->point[i]=NULL;

}

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

{

int start = edges[i].start;

int end = edges[i].end;

struct vertex \*v = (struct vertex \*) malloc(sizeof(struct vertex));

v->end = end;

v->next = graph->point[start];

graph->point[start]=v;

}

return graph;

}

void main()

{

struct Edge edges[]={ {0,1},{0,2},{0,3},{1,2},{1,4},{2,4},{2,3},{3,1}};

int n= sizeof(edges)/sizeof(edges[0]);

struct graph \*graph=create\_graph(edges, n);

int i;

for(i=0;i<5;i++)

{

struct vertex \*ptr = graph->point[i];

while(ptr!=NULL)

{

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

ptr=ptr->next;

}

printf("\n");

}

}

**OUTPUT:**

