**Delhi Technological University**

Department Of Computer Science And Engineering



**Data Structure Lab**

Lab File for the course CO201 (P)

**Submitted by :** Vishal Das (2K21/CO/523)

**Submitted to :** Anurag Goel

**Contents**

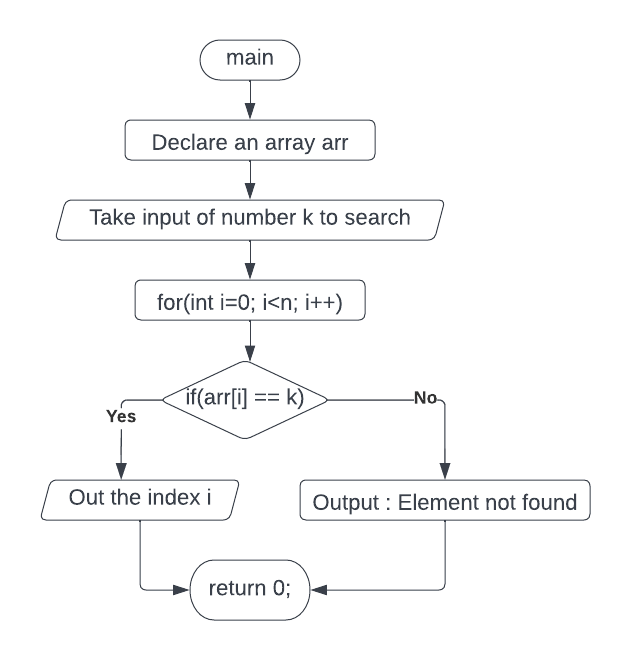
|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Program** | **Date** | **Signature** |
| 1 | Write a C/C++ program to Implement Linear Search. | August 24, 2022 |  |
| 2 | Write a C/C++ program to Implement Binary Search. |
| 3 | Write a C/C++ program to insert an element at the mid-position in the One-dimensional array. | August 31, 2022 |
| 4 | Write a C/C++ program to delete a given row in the two-dimensional array. |
| 5 | Write a C/C++ program to implement a stack data structure and perform its operations. | September 07, 2022 |
| 6 | Write a C/C++ program to implement two stacks using a single array. |
| 7 | Write a C/C++ program to find the minimum element of the stack in constant time with using extra space. | September 14, 2022 |
| 8 | Write a C/C++ program to find the minimum element of the stack in constant time without using extra space. |
| 9 | Write a C/C++ program to represent a sparse matrix in compact form. | September 21, 2022 |
| 10 | Write a C/C++ program to perform the addition, multiplication and transpose operations on sparse matrix given in compact form. |
| 11 | Write a C/C++ program to implement Queue Data structure. |
| 12 | Write a C/C++ program to reverse the first k elements of a given Queue. |
| 13 | Write a C/C++ program to check whether the given string is Palindrome or not using Double Ended Queue (DEQUE). | October 12, 2022 |
| 14 | Write a C/C++ program to implement Tower of Hanoi Problem using Stack. |
| 15 | Write a C/C++ program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position. |
| 16 | Write a C/C++ program to split a given linked list into two sub-list as Front sub-list and Back sub-list, if odd number of the element then add the last element into the front list. |
| 17 | Given a Sorted doubly linked list of positive integers and an integer, finds all the pairs (sum of two nodes data part) that is equal to the given integer value. |
| 18 | Write a C/C++ program to implement Stack Data Structure using Queue. | October 19, 2022 |
| 19 | Write a C/C++ program to implement Queue Data Structure using Stack. |
| 20 | Write a C/C++ program to implement the Binary Tree using linked list and perform In-order traversal. | October 26, 2022 |
| 21 | Write a C/C++ program to check whether the given tree is a Binary Search Tree or not. |
| 22 | Write a C/C++ program to implement insertion in the AVL tree. | November 02, 2022 |
| 23 | Write a C/C++ program to Delete a key from the AVL tree. |
| 24 | Write a C/C++ program to count the number of leaf nodes in an AVL tree. |

**Program 01 :** Write a C/C++ program to Implement Linear Search.

**Theory :**

Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.

**Algorithm :**



**Code :**

#include <iostream>

using namespace std;

int linearSearch(int array[], int size, int key)

{

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

if (array[i] == key)

return i;

return -1;

}

int main(void)

{

int size, key;

cout << "Input the size of array : ";

cin >> size;

int arr[size];

cout << "Input the " << size << " elements : ";

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

cin >> arr[i];

cout << "Element to search for : ";

cin >> key;

int index = linearSearch(arr, size, key);

if (index != -1)

cout << "Element found at index " << index << endl;

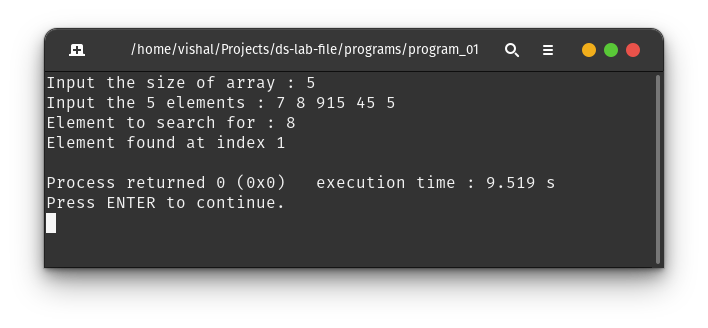
else

cout << "Element doesn't exist in the array" << endl;

return 0;

}

**Output :**



**Program 02 :** Write a C/C++ program to Implement Binary Search.

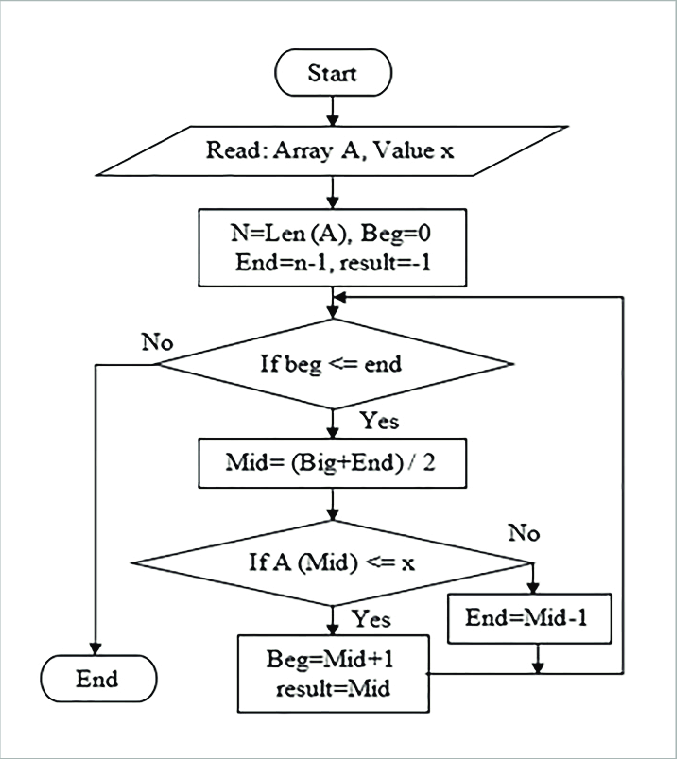
**Theory :**

Binary Search is a searching algorithm used in a sorted array by repeatedly dividing the

search interval in half. The idea of binary search is to use the information that the array is

sorted and reduce the time complexity to O(Log n).

**Algorithm :**



**Code :**

#include <iostream>

using namespace std;

int binarySearch(int array[], int key, int right\_index)

{

int left\_index = 0;

while (left\_index <= right\_index)

{

int mid\_index = (left\_index + right\_index) / 2;

int middle\_element = array[mid\_index];

if (middle\_element == key)

return mid\_index;

else if (middle\_element > key)

right\_index = mid\_index - 1;

else

left\_index = mid\_index + 1;

}

return -1;

}

int main(void)

{

int size, key;

cout << "Input the size of sorted array (ascending) : ";

cin >> size;

int arr[size];

cout << "Input the " << size << " elements : ";

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

cin >> arr[i];

cout << "Element to search for : ";

cin >> key;

int index = binarySearch(arr, key, size);

if (index != -1)

cout << "Element found at index " << index << endl;

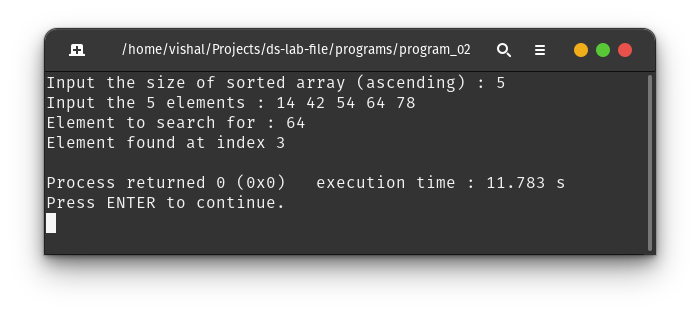
else

cout << "Element doesn't exist in the array" << endl;

return 0;

}

**Output :**

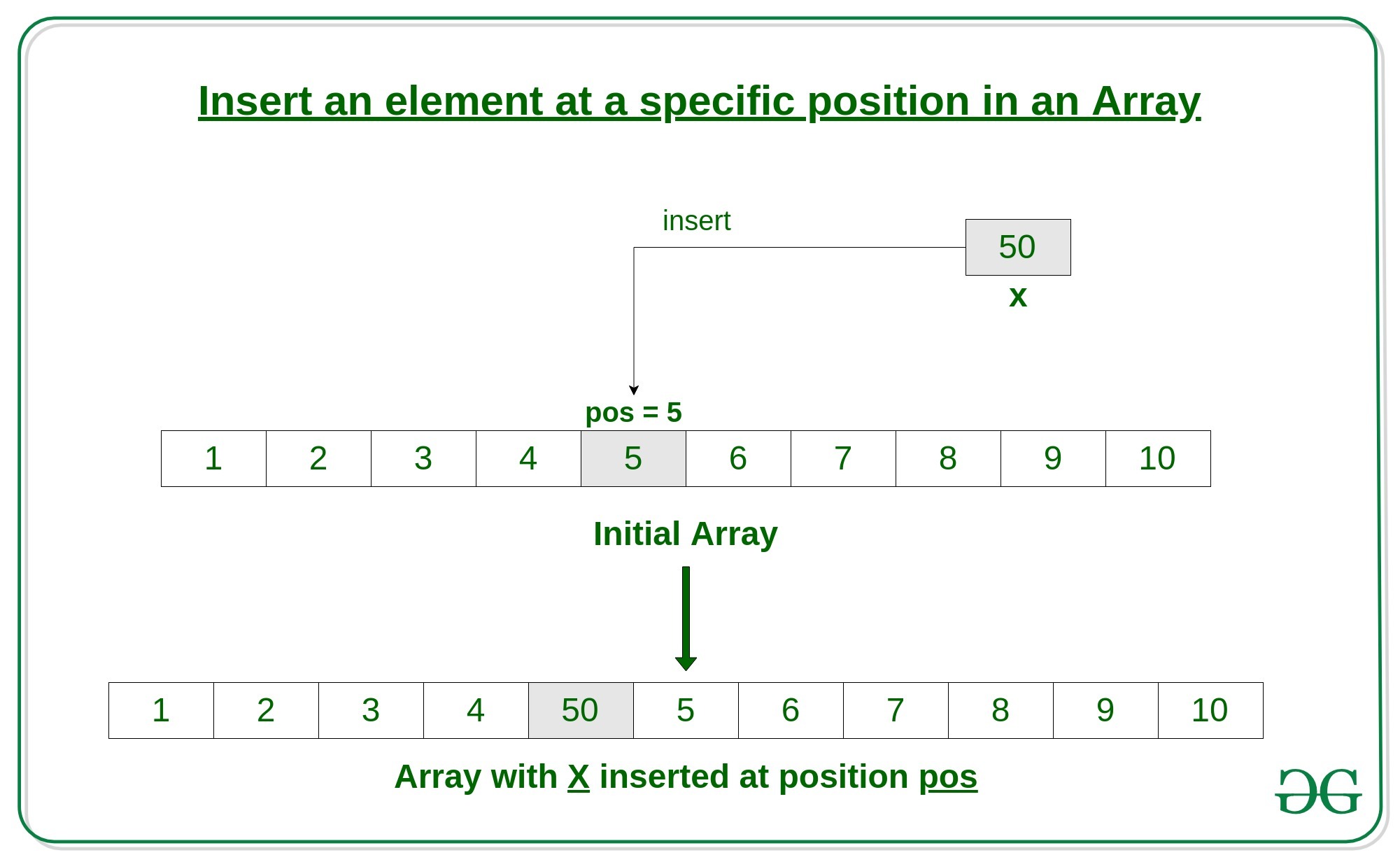


**Program 03 :** Write a C/C++ program to insert an element at the mid-position in the One-dimensional array.

**Theory :**

An array is a collection of items stored at contiguous memory locations. For inserting an element in between of an array, we need to shift the data in the ahead of insertion location one index ahead and put the insertion element at the empty place hence generated.

**Algorithm :**



**Code :**

#include <iostream>

using namespace std;

int main(void)

{

int n, top;

cout << "Input max array size : ";

cin >> n;

int a[n];

cout << "Input number of array elements : ";

cin >> top;

cout << "Input array elements : ";

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

cin >> a[i];

cout << "Array elements : ";

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

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

int mid = top / 2;

for (int i = top; i > mid; i--)

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

cout << endl

<< "Input the element to be inserted : ";

cin >> a[mid];

top++;

cout << "Array elements after insertion : ";

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

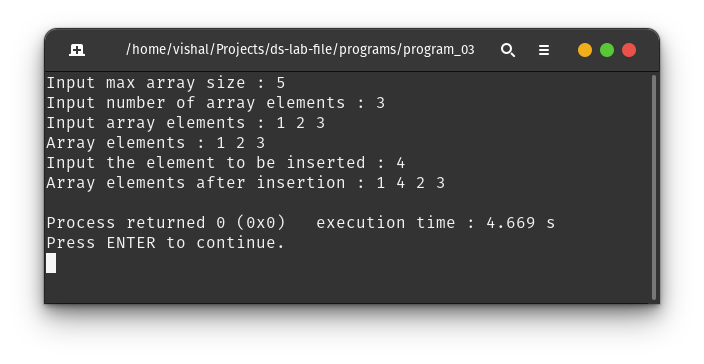
cout << a[i] << " ";

cout << endl;

return 0;

}

**Output :**

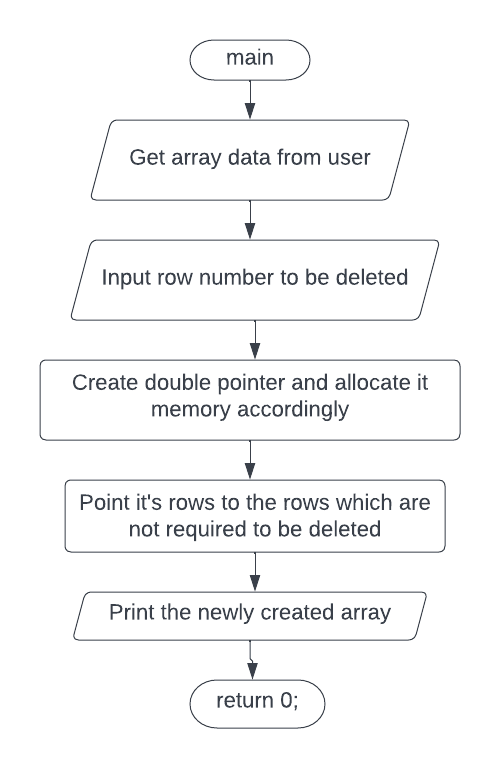


**Program 04 :** Write a C/C++ program to delete a given row in the two-dimensional array.

**Theory :**

2D Arrays in C/C++ are actually double pointer, where first level of pointer represents a row and the second level of pointer represents the element. To remove a particular row, all we need to do is create a new double pointer and not include the memory location of deleted row in it.

**Algorithm :**



**Code :**

#include <iostream>

using namespace std;

int main(void)

{

int n, k, delRow;

cout << "Input dimension of 2D Array : ";

cin >> n >> k;

int arr[n][k];

cout << "Input the elements : " << endl;

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

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

cin >> arr[i][j];

int \*\*newArr = new int \*[n-1];

cout

<< "Input row number to delete : ";

cin >> delRow;

cout << endl

<< "Matrix with deleted row : " << endl;

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

{

if (i < delRow - 1)

\*(newArr + i) = \*(arr + i);

else

\*(newArr + i) = \*(arr + i + 1);

}

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

{

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

cout << newArr[i][j] << " ";

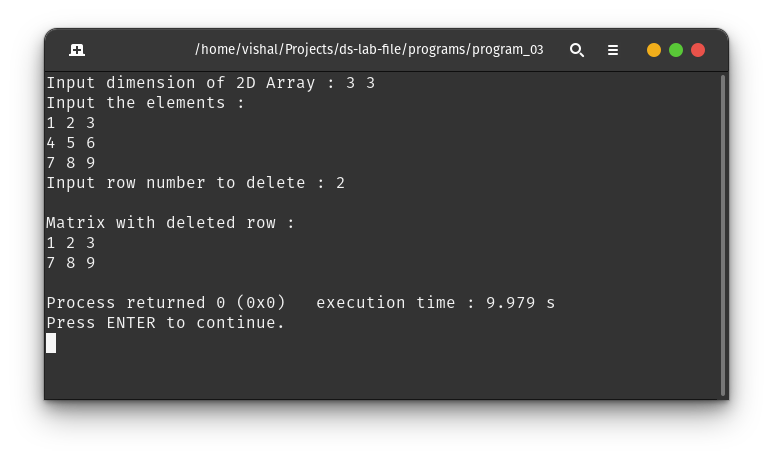
cout << endl;

}

return 0;

}

**Output :**

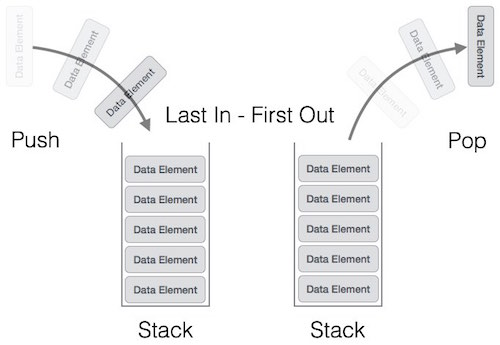


**Program 05 :** Write a C/C++ program to implement a stack data structure and perform its operations.

**Theory :**

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end (top) and an element is removed from that end only.

**Algorithm :**



**Code :**

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int \*array;

int top;

int capacity;

} stack;

stack \*createStack(int capacity) {

stack \*s = (stack \*)malloc(sizeof(stack));

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

s->capacity = capacity;

s->top = -1;

}

void pop(stack \*s) {

if (s->top != -1) {

s->top--;

printf("Top element popped!\n");

} else

printf("ERROR : Stack underflow");

}

int isFull(stack \*s) {

if (s->top == s->capacity - 1)

return 1;

return 0;

}

int isEmpty(stack \*s) {

if (s->top == -1)

return 1;

return 0;

}

void push(stack \*s, int x) {

if (s->top != s->capacity - 1) {

s->array[++s->top] = x;

printf("Pushed %d!\n", x);

} else

printf("ERROR : Stack overflow");

}

int top(stack \*s) {

if (s->top != -1)

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

}

int main() {

stack \*s = createStack(10);

push(s, 10);

push(s, 72);

push(s, 45);

printf("Top element : %d\n", top(s));

pop(s);

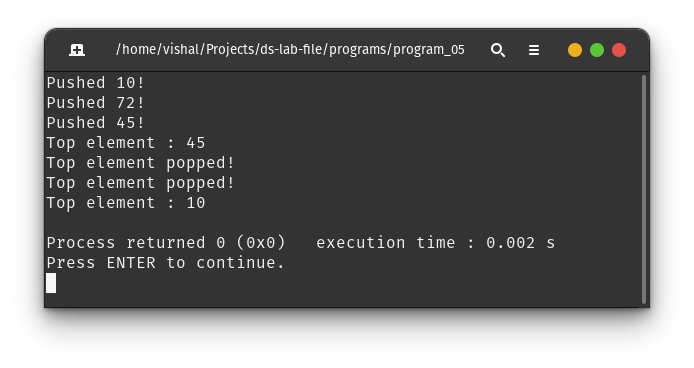
pop(s);

printf("Top element : %d\n", top(s));

return 0;

}

**Output :**

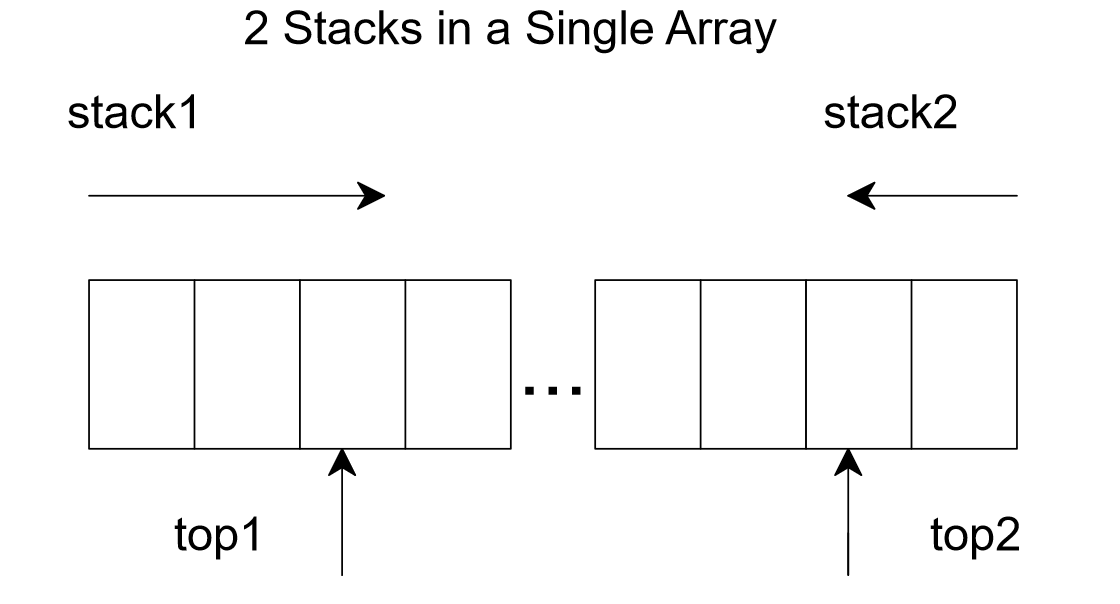


**Program 06 :** Write a C/C++ program to implement two stacks using a single array.

**Theory :**

This method efficiently utilizes the available space. It doesn’t cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction.

**Algorithm :**



**Code :**

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int top1;

int top2;

int capacity;

int \*array;

} dualstack;

dualstack \*createDualStack(int capacity) {

dualstack \*s = (dualstack \*)malloc(sizeof(dualstack));

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

s->top1 = -1;

s->top2 = capacity;

return s;

}

int isEmpty(dualstack \*s, int stack\_id) {

if (stack\_id == 1)

return s->top1 == -1;

if (stack\_id == 2)

return s->top2 == s->capacity;

else

printf("Invalid Stack ID");

return 0;

}

void push(dualstack \*s, int x, int stack\_id) {

if (s->top1 + 1 == s->top2) {

printf("Error : Stack Overflow\n");

return;

}

if (stack\_id == 1)

s->array[++s->top1] = x;

else if (stack\_id == 2)

s->array[--s->top2] = x;

else

printf("Invalid Stack ID\n");

}

void pop(dualstack \*s, int stack\_id) {

if (stack\_id == 1) {

if (s->top1 != -1)

s->top1--;

else

printf("Stack 1 is already empty\n");

} else if (stack\_id == 2) {

if (s->top2 != s->capacity)

s->top2++;

else

printf("Stack 2 is already empty\n");

} else

printf("Invalid Stack ID\n");

}

int top(dualstack \*s, int stack\_id) {

if (stack\_id == 1) {

if (!isEmpty(s, 1))

return s->array[s->top1];

} else if (stack\_id == 2) {

if (!isEmpty(s, 2))

return s->array[s->top2];

} else

printf("Invalid Stack ID");

return -1;

}

void displayStackTop(dualstack \*s) {

printf("Stack 1 : %d\n", top(s, 1));

printf("Stack 2 : %d\n\n", top(s, 2));

}

int isFull(dualstack \*s) { return s->top1 == s->top2 - 1; }

int main(void) {

dualstack \*s = createDualStack(10);

printf("Stack 1 is empty : %d\n", isEmpty(s, 1));

printf("Stack 2 is empty : %d\n", isEmpty(s, 2));

printf("Stack is full : %d\n\n", isFull(s));

push(s, 11, 1);

push(s, 12, 2);

push(s, 13, 1);

push(s, 14, 1);

push(s, 15, 2);

push(s, 16, 1);

displayStackTop(s);

pop(s, 1);

displayStackTop(s);

push(s, 17, 1);

push(s, 18, 2);

push(s, 19, 2);

displayStackTop(s);

push(s, 20, 1);

push(s, 21, 2);

displayStackTop(s);

push(s, 22, 1); // should give stack overflow

pop(s, 2); // make space from stack 2

push(s, 22, 1); // use that empty space for stack 1

displayStackTop(s);

printf("Stack 1 is empty : %d\n", isEmpty(s, 1));

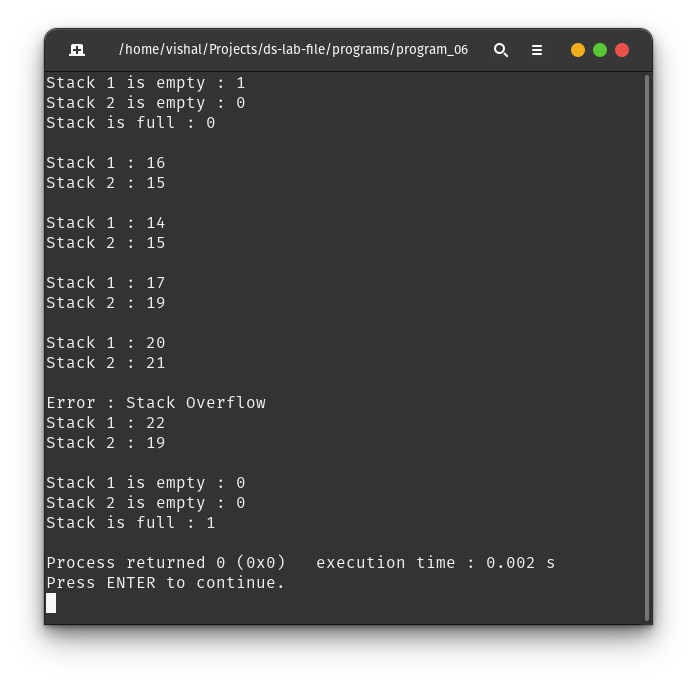
printf("Stack 2 is empty : %d\n", isEmpty(s, 2));

printf("Stack is full : %d\n", isFull(s));

return 0;

}

**Output :**



**Program 07 :** Write a C/C++ program to find the minimum element of the stack in constant time without using extra space.

**Theory :**

The approach is to keep another stack which stores the current smallest number as the original stack is pushed with a number and pop element from it when original stack is popped. Hence the top of stack will always return the smallest number present in the stack currently.

**Algorithm :**

1. Create a different minStack along with the original stack.
2. When pushing the first element in stack, push it in both minStack and stack.
3. When pushing newer elements in stack, push the minimum of (minStack top and element to be pushed) in the minStack.
4. When popping an element, pop from both the stacks.
5. To get the current smallest number, simply return the top of the minStack.

**Code :**

#include "stack\_impl.c"

#include <stdio.h>

#include <stdlib.h>

typedef struct {

stack \*stack;

stack \*minStack;

} minstack;

minstack \*createMinStack(int capacity) {

minstack \*m = (minstack \*)malloc(sizeof(minstack));

m->minStack = createStack(capacity);

m->stack = createStack(capacity);

return m;

}

void minstack\_push(minstack \*m, int x) {

push(m->stack, x);

if (!isEmpty(m->minStack))

push(m->minStack, (top(m->minStack) > x ? x : top(m->minStack)));

else

push(m->minStack, x);

}

int minstack\_top(minstack \*m) {

if (!isEmpty(m->stack))

return top(m->stack);

printf("Stack is empty!\n");

return -1;

}

void minstack\_pop(minstack \*m) {

if (isEmpty(m->stack))

return;

pop(m->stack);

pop(m->minStack);

}

int getMinimum(minstack \*m) { return top(m->minStack); }

int main(void) {

minstack \*s = createMinStack(10);

minstack\_push(s, 31);

minstack\_push(s, 72);

minstack\_push(s, 10);

minstack\_push(s, 45);

readStack(s->stack);

printf("Top element : %d\n", minstack\_top(s));

printf("Minimum element : %d\n\n", getMinimum(s));

minstack\_pop(s);

minstack\_pop(s);

readStack(s->stack);

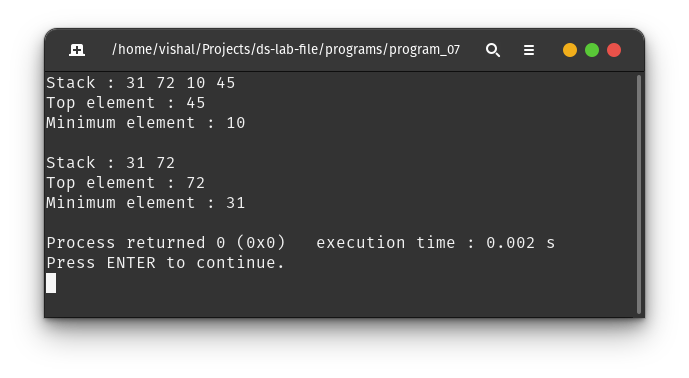
printf("Top element : %d\n", minstack\_top(s));

printf("Minimum element : %d\n", getMinimum(s));

return 0;

}

**Output :**



**Program 08 :** Write a C/C++ program to find the minimum element of the stack in constant time without using extra space.

**Theory :**

The approach is to keep every element of stack corelated to the latest minimum number which comes in the stack.

**Algorithm :**

1. Create a the original stack.
2. When pushing the first element in stack, set its element as the minimum.
3. When pushing newer elements in stack,
   1. Push the difference of the new element and the minimum rather than pushing element itself.
   2. If the difference is negative (new element is smaller than the current minimum), update the new minimum.
4. To get the current smallest number, simply return,
   1. the top of the stack + minimum if top of stack is not zero.
   2. the minimum element is top of stack is zero.

**Code :**

#include "stack\_impl.c"

typedef struct {

stack \*s;

int min;

} dasMinStack;

dasMinStack \*createDasMinStack(int capacity) {

dasMinStack \*dms = malloc(sizeof(dasMinStack));

dms->s = createStack(capacity);

return dms;

}

void dasPush(dasMinStack \*dms, int x) {

if (isEmpty(dms->s)) {

dms->min = x;

push(dms->s, 0);

} else {

push(dms->s, x - dms->min);

if (x < dms->min)

dms->min = x;

}

}

int dasTop(dasMinStack \*dms) {

int x = top(dms->s) + dms->min;

if (x > 0)

return x;

else

return dms->min;

}

void dasPop(dasMinStack \*dms) {

int x = top(dms->s);

pop(dms->s);

if (x < 0)

dms->min = dms->min - x;

}

void printStack(dasMinStack \*dms) {

int min = dms->min;

for (int i = dms->s->top; i >= 0; i--) {

int t = dms->s->array[i];

if (t < 0) {

printf("%d ", min);

min = min - t;

} else

printf("%d ", t + min);

}

printf("\n");

}

int main(void) {

dasMinStack \*s = createDasMinStack(10);

dasPush(s, 31);

printStack(s);

printf("Top element : %d\n", dasTop(s));

printf("Minimum element : %d\n\n", s->min);

dasPush(s, 72);

printStack(s);

printf("Top element : %d\n", dasTop(s));

printf("Minimum element : %d\n\n", s->min);

dasPush(s, 10);

printStack(s);

printf("Top element : %d\n", dasTop(s));

printf("Minimum element : %d\n\n", s->min);

dasPush(s, 45);

printStack(s);

printf("Top element : %d\n", dasTop(s));

printf("Minimum element : %d\n\n", s->min);

printStack(s);

printf("Top element : %d\n", dasTop(s));

printf("Minimum element : %d\n\n", s->min);

dasPop(s);

printStack(s);

printf("Top element : %d\n", dasTop(s));

printf("Minimum element : %d\n\n", s->min);

dasPop(s);

printStack(s);

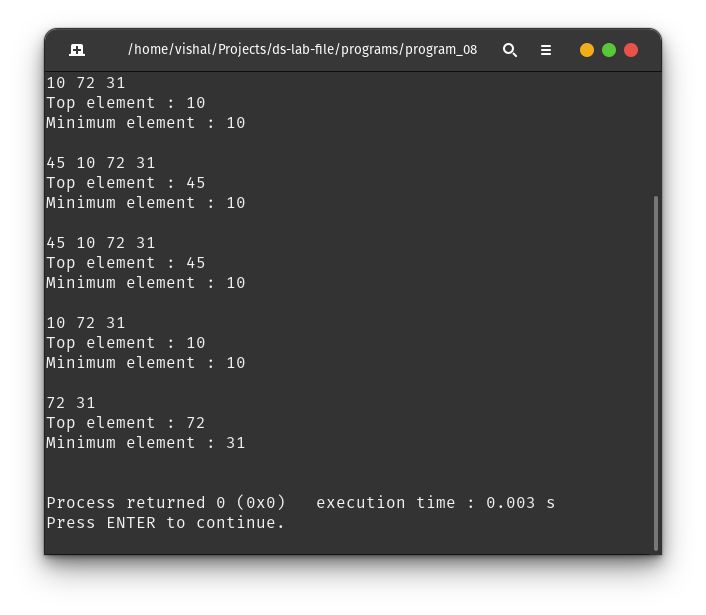
printf("Top element : %d\n", dasTop(s));

printf("Minimum element : %d\n\n", s->min);

return 0;

}

**Output :**



**Program 09 :** Write a C/C++ program to create a sparse matrix using arrays.

**Theory :**

A sparse matrix is a matrix in which many or most of the elements have a value of zero. This is in contrast to a dense matrix, where many or most of the elements have a non-zero value.

**Algorithm :**

Only store the row, column and the element value for the non-zero entries of the matrix in an array of structures.

**Code :**

#define MAX 101

#define TABLE printf("ROW\tCOL\tVAL\n")

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int row;

int col;

int value;

} element;

int main(void) {

element a[MAX];

printf("Enter the number of rows and columns of the matrix : ");

scanf("%d %d", &a[0].row, &a[0].col);

printf("Enter the non zero elements of the matrix : ");

scanf("%d", &a[0].value);

TABLE;

for (int i = 1; i <= a[0].value; i++)

scanf("%d %d %d", &a[i].row, &a[i].col, &a[i].value);

printf("Given Sparse Matrix is represented as :\n");

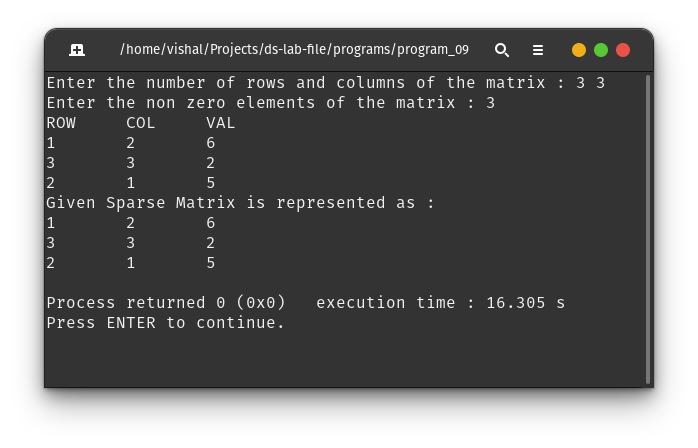
for (int i = 1; i <= a[0].value; i++)

printf("%d\t%d\t%d\n", a[i].row, a[i].col, a[i].value);

return 0;

}

**Output :**



**Program 10 :** Write a C/C++ program to perform the addition, multiplication and transpose operations on sparse matrix given in compact form.

**Theory :**

A sparse matrix is a matrix in which many or most of the elements have a value of zero. This is in contrast to a dense matrix, where many or most of the elements have a non-zero value.

**Algorithm :**

Apply various operation considering all the elements of structures in the given array.

**Code :**

#define MAX 101 /\* max number of elements + 1 \*/

#define TABLE printf("ROW\tCOL\tVAL\n")

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int row;

int col;

int value;

} element;

void printSparseMatrix(element \*a) {

TABLE;

for (int i = 1; i <= a[0].value; i++)

printf("%d\t%d\t%d\n", a[i].row, a[i].col, a[i].value);

}

element \*createSparseMatrix() {

element \*matrix = malloc(MAX \* sizeof(element));

printf("Enter the number of rows and columns of the matrix : ");

scanf("%d %d", &matrix[0].row, &matrix[0].col);

printf("Enter the non zero elements of the matrix : ");

scanf("%d", &matrix[0].value);

TABLE;

for (int i = 1; i <= matrix[0].value; i++)

scanf("%d %d %d", &matrix[i].row, &matrix[i].col, &matrix[i].value);

matrix = realloc(matrix, (matrix[0].value + 1) \* sizeof(element));

return matrix;

}

element \*transposeSparse(element \*a) {

element \*c = (element \*)malloc(sizeof(element) \* (a[0].row \* a[0].col + 1));

c[0].row = a[0].col;

c[0].col = a[0].row;

c[0].value = a[0].value;

for (int i = 1; i <= a[0].value; i++) {

c[i].row = a[i].col;

c[i].col = a[i].row;

c[i].value = a[i].value;

}

return c;

}

element \*additionSparse(element \*a, element \*b) {

if (a[0].row != b[0].row || a[0].col != b[0].col) {

printf("ERROR : Incompatible matrices for addition\n");

exit(EXIT\_FAILURE);

}

element \*c = (element \*)malloc(sizeof(element) \* (a[0].row \* b[0].col + 1));

c[0].row = a[0].row;

c[0].col = a[0].col;

c[0].value = 0;

for (int i = 1; i <= a[0].row; i++) {

for (int j = 1; j <= a[0].col; j++) {

int elem = 0;

for (int k = 1; k <= a[0].value; k++) {

if ((a[k].row == i && a[k].col == j) ||

(b[k].row == i && b[k].col == j)) {

elem += (a[k].row == i && a[k].col == j) ? a[k].value : 0;

elem += (b[k].row == i && b[k].col == j) ? b[k].value : 0;

}

}

if (elem != 0) {

c[0].value++;

c[c[0].value].row = i;

c[c[0].value].col = j;

c[c[0].value].value = elem;

}

}

}

return c;

}

element \*mutiplySparse(element \*a, element \*b) {

if (a[0].col != b[0].row) {

printf("Incompatible matrices for multiplication\n");

exit(EXIT\_FAILURE);

}

element \*c = (element \*)malloc(sizeof(element) \* (a[0].row \* b[0].col + 1));

c[0].row = a[0].row;

c[0].col = b[0].col;

c[0].value = 0;

for (int i = 1; i <= a[0].row; i++) {

int row[a[0].col];

for (int k = 0; k < a[0].col; k++)

row[k] = 0;

for (int j = 1; j <= a[0].value; j++) {

if (a[j].row == i) {

row[a[j].col - 1] = a[j].value;

}

}

for (int m = 1; m <= b[0].col; m++) {

int col[b[0].row];

for (int k = 0; k < b[0].row; k++)

col[k] = 0;

for (int j = 1; j <= b[0].value; j++) {

if (b[j].col == m) {

col[b[j].row - 1] = b[j].value;

}

}

int sum = 0;

for (int l = 0; l < a[0].col; l++)

sum += row[l] \* col[l];

if (sum != 0) {

c[0].value++;

c[c[0].value].row = a[i].row;

c[c[0].value].col = b[i].col;

c[c[0].value].value = sum;

}

}

}

return c;

}

int main(void) {

element \*a, \*b, \*c;

a = createSparseMatrix();

c = transposeSparse(a);

// print A

printf("Matrix A is represented as :\n");

printSparseMatrix(a);

printf("\n");

b = createSparseMatrix();

// print B

printf("Matrix B is represented as :\n");

printSparseMatrix(b);

// transpose of A

printf("\nTranspose(A) is represented as :\n");

printSparseMatrix(c);

// addition of A and B

printf("\nAddition of A and B is represented as :\n");

free(c);

c = additionSparse(a, b);

printSparseMatrix(c);

// multiplication of A and B

free(c);

c = mutiplySparse(a, b);

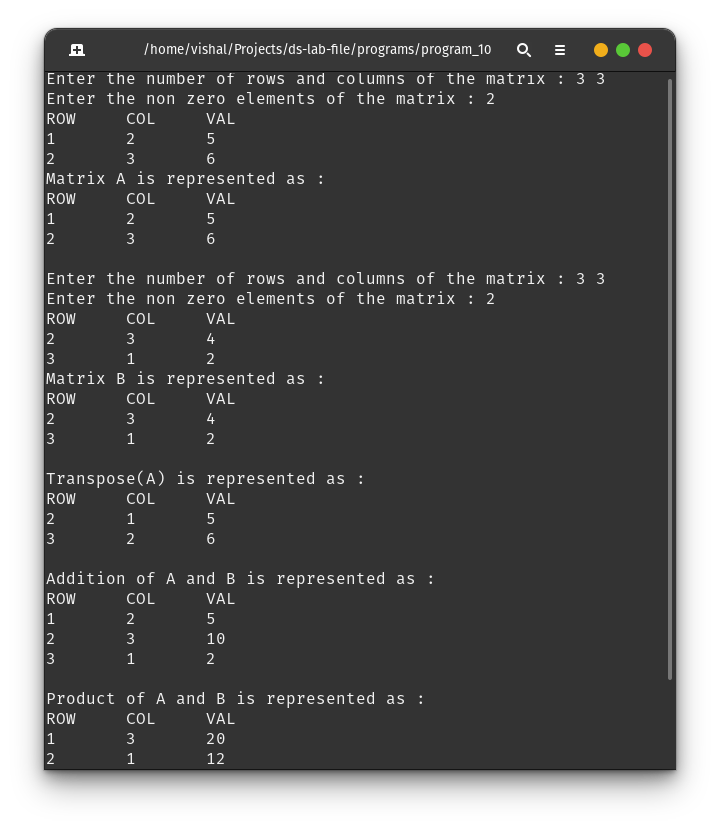
printf("\nProduct of A and B is represented as :\n");

printSparseMatrix(c);

return 0;

}

**Output :**



**Program 11 :** Write a C/C++ program to implement Queue Data structure.

**Theory :** A queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order. We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.

**Algorithm :**

enqueue() − add (store) an item to the queue.

dequeue() − remove (access) an item from the queue.

peek() − Gets the element at the front of the queue without removing it.

isfull() − Checks if the queue is full.

isempty() − Checks if the queue is empty.

**Code :**

#include <stdio.h>

#include <stdlib.h>

typedef struct {

int front;

int rear;

int \*arr;

int capacity;

} queue;

queue \*createQ(int capacity) {

queue \*q = malloc(sizeof(queue \*));

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

q->capacity = capacity;

q->front = q->rear = -1;

return q;

};

char isEmpty(queue \*q) {

if (q->front == q->rear && q->front == -1)

return 1;

return 0;

};

char isFull(queue \*q) {

if (q->rear == q->capacity - 1)

return 1;

return 0;

};

void enqueue(queue \*q, int value) {

if (isFull(q)) {

printf("ERROR : Queue is full\n");

return;

}

if (isEmpty(q)) {

q->arr[++q->rear] = value;

q->front = q->rear;

return;

}

q->arr[++q->rear] = value;

};

void dequeue(queue \*q) {

if (isEmpty(q)) {

printf("ERROR : Queue is empty\n");

return;

}

q->front++;

if (q->front > q->rear)

q->front = q->rear = -1;

};

int peek(queue \*q) {

if (isEmpty(q)) {

printf("ERROR : Queue is empty\n");

exit(-1);

}

return q->arr[q->front];

};

int main(void) {

queue \*q = createQ(10);

enqueue(q, 14);

enqueue(q, 18);

enqueue(q, 5);

printf("Peek : %d\n", peek(q));

dequeue(q);

printf("Peek : %d\n", peek(q));

dequeue(q);

printf("Peek : %d\n", peek(q));

dequeue(q);

dequeue(q);

enqueue(q, 7);

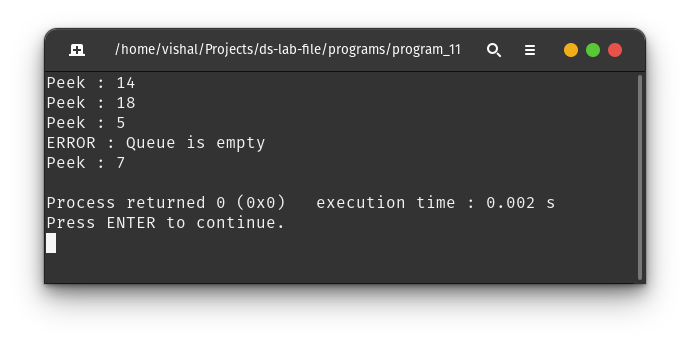
enqueue(q, 2);

printf("Peek : %d\n", peek(q));

return 0;

};

**Output :**



**Program 12 :** Write a C/C++ program to reverse the first k elements of a given Queue.

**Theory :**

The primary difference between Stack and Queue Data Structures is that Stack follows LIFO while Queue follows FIFO data structure type. LIFO refers to Last In First Out.

**Algorithm :**

1. Push first k elements in the stack while dequeuing the queue.
2. Now enqueue the remaining elements in a different queue till original queue is not empty.
3. Pop elements from stack and fill them in the original queue.
4. Enqueue the elements from the second queue to the orignal queue.

**Code :**

#include "./queue\_impl.c"

#include "./stack\_impl.c"

#include <stdio.h>

void \*reverseKElement(queue \*q, int k) {

stack \*s = createStack(k);

int upto = q->front + k;

for (int i = q->front; i < upto; i++) {

push(s, peek(q));

dequeue(q);

}

queue \*t = createQ(q->capacity - k);

while (!isQueueEmpty(q)) {

enqueue(t, peek(q));

dequeue(q);

}

while (!isEmpty(s)) {

enqueue(q, top(s));

pop(s);

}

while (!isQueueEmpty(t)) {

enqueue(q, peek(t));

dequeue(t);

}

}

int main(void) {

queue \*q = createQ(15);

for (int i = 1; i <= 15; i++)

enqueue(q, i);

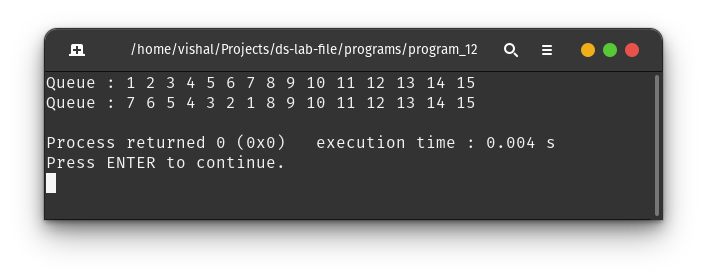
showQ(q);

reverseKElement(q, 7);

showQ(q);

}

**Output :**

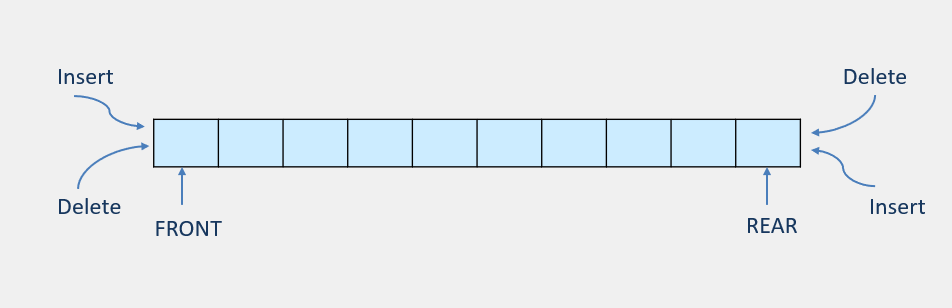


**Program 13 :** Write a C/C++ program to check whether the given string is Palindrome or not using Double Ended Queue (DEQUE).

**Theory :**

A double-ended queue is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).

**Algorithm :**



**Code :**

#include "deque\_impl.c"

char isPalindrome(char \*s) {

int n = 0;

while (s[n] != '\0') {

n++;

}

deque \*q = createDeque(n);

for (int i = 0; i < n / 2; i++) {

push\_back(q, (int)s[i]);

}

for (int i = (n + 1) / 2; i < n; i++) {

if ((char)peek\_back(q) != s[i])

return 0;

else

pop\_back(q);

}

return isQEmpty(q);

}

int main() {

char s1[] = "HeyayeH";

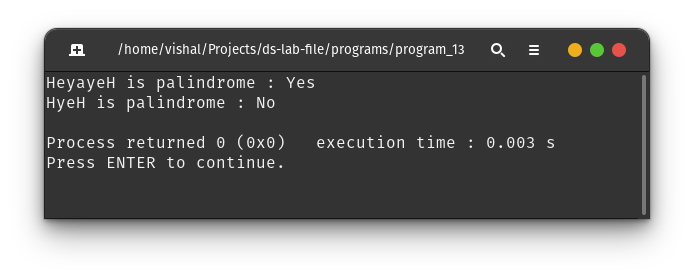
char s2[] = "HyeH";

printf("%s is palindrome : %s\n", s1, isPalindrome(s1) ? "Yes" : "No");

printf("%s is palindrome : %s\n", s2, isPalindrome(s2) ? "Yes" : "No");

}

**Output :**



**Program 14 :** Write a C/C++ program to implement Tower of Hanoi Problem using Stack.

**Theory :**

Tower of Hanoi is a mathematical puzzle consisting of three rods and a number of disks, which can slide onto any rod. The puzzle begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to the last rod, obeying the following rules - Only one disk may be moved at a time, each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod. No disk may be placed on top of a disk that is smaller than it.

**Algorithm :**

1. Solve Tower of Hanoi with ‘N-1’ disks from ‘A’ to ‘B’.
2. Move the last disk from rod A to rod C.
3. Solve Tower of Hanoi with ‘N-1’ disks from ‘B’ to ‘C’.

**Code :**

#include "stack\_impl.c"

#include <stdio.h>

stack \*from\_rod, \*to\_rod, \*aux\_rod;

long long itr = 0;

void solveHanoi(int n, stack \*from, stack \*to, stack \*helper) {

if (n == 0)

return;

itr++;

solveHanoi(n - 1, from, helper, to);

int t = top(from);

push(to, t);

pop(from);

solveHanoi(n - 1, helper, to, from);

}

int main(void) {

int n = 3;

from\_rod = createStack(n);

aux\_rod = createStack(n);

to\_rod = createStack(n);

for (int i = n; i > 0; i--) {

push(from\_rod, i);

}

printf("Tower of Hanoi before solving :\n");

readStack(from\_rod);

readStack(aux\_rod);

readStack(to\_rod);

solveHanoi(n, from\_rod, to\_rod, aux\_rod);

printf("\nTower of Hanoi after solving :\n");

readStack(from\_rod);

readStack(aux\_rod);

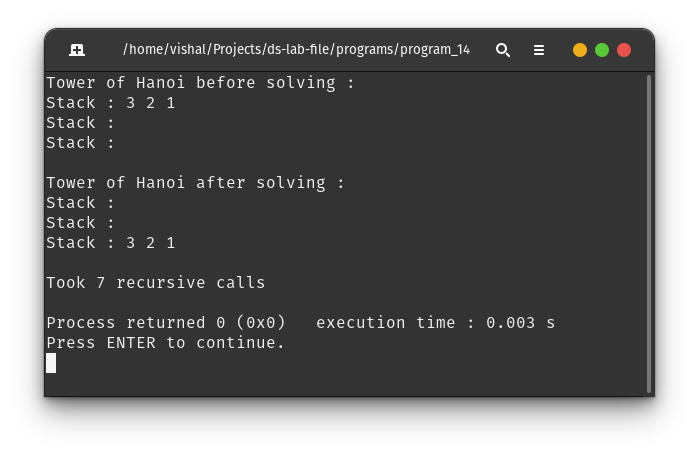
readStack(to\_rod);

printf("\nTook %lld recursive calls\n", itr);

return 0;

}

**Output :**



**Program 15 :** Write a C/C++ program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position.

**Theory :**

Linked List is a linear data structure, but unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers. They include a series of connected nodes.

**Algorithm :**

1. If node is to be inserted at start, then create a new node and connect it’s next to the current head and make it the new head.
2. If node is to be inserted at some position, traverse to the position where node is to be inserted.
3. Connect the new node’s next to the previous node’s next and join the previous node to the new node.

**Code :**

#include <stdio.h>

#include <stdlib.h>

typedef struct node {

int value;

struct node \*next;

} node;

node \*createNode(int value) {

node \*x = (node \*)malloc(sizeof(node));

x->value = value;

x->next = NULL;

}

node \*insertFromStart(node \*\*pointer\_to\_head, int n, int value) {

node fake\_head = {0, \*pointer\_to\_head};

node \*traverse = &fake\_head;

while (--n && traverse->next) {

traverse = traverse->next;

}

node \*x = createNode(value);

x->next = traverse->next;

traverse->next = x;

\*pointer\_to\_head = fake\_head.next;

return fake\_head.next;

}

node \*insertAtStart(node \*\*pointer\_to\_head, int value) {

return insertFromStart(pointer\_to\_head, 1, value);

}

void printLL(node \*head) {

if(head==NULL)

printf("Empty Linked List!\n");

printf("Linked List : ");

while (head) {

printf("%d, ", head->value);

head = head->next;

}

printf("\n");

}

int main() {

node \*head = NULL;

printLL(head);

insertAtStart(&head, 25);

printLL(head);

insertAtStart(&head, 36);

printLL(head);

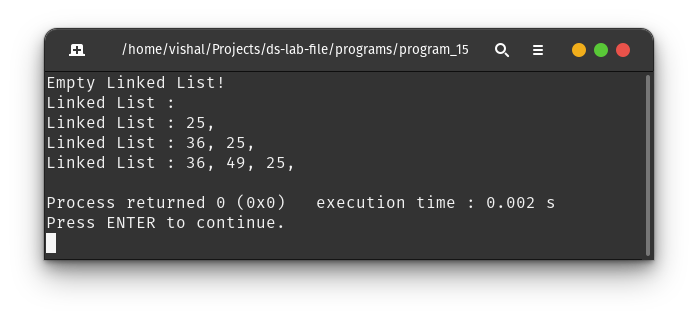
insertFromStart(&head, 2, 49);

printLL(head);

return 0;

}

**Output :**



**Program 16 :** Write a C/C++ program to split a given linked list into two sub-list as Front sub-list and Back sub-list, if odd number of the element then add the last element into the front list.

**Theory :**

Linked List is a linear data structure, but unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers. They include a series of connected nodes.

**Algorithm :**

1. Make the head of first sub list as the head of given linked list.
2. Traverse to the middle of ht linked list using fast-slow pointer concept.
3. Make the next of middle node as head of the second sub list.
4. Break the link of the middle node and connect it to null pointer.

**Code :**

#include "ll\_impl.c"

void splitLinkedList(node \*head, node \*\*l1, node \*\*l2) {

if (head == NULL) {

\*l1 = NULL;

\*l2 = NULL;

return;

}

node \*alpha = head;

node \*sigma = head;

while (sigma->next && sigma->next->next) {

alpha = alpha->next;

sigma = sigma->next->next;

}

\*l2 = alpha->next;

alpha->next = NULL;

\*l1 = head;

}

int main() {

node \*head = NULL, \*l1, \*l2;

for (int i = 1; i <= 7; i++)

insertAtEnd(&head, i);

printLL(head);

printf("\nSplitting original linked list into 2 sublists\n\n");

splitLinkedList(head, &l1, &l2);

printf("Sublist 1 : ");

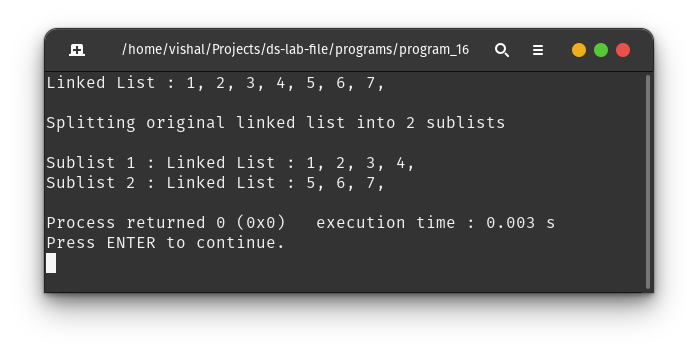
printLL(l1);

printf("Sublist 2 : ");

printLL(l2);

}

**Output :**



**Program 17 :** Given a Sorted doubly linked list of positive integers and an integer, finds all the pairs (sum of two nodes data part) that is equal to the given integer value.

**Theory :**

Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers. They include a series of connected nodes.

**Algorithm :**

1. Take pointers \*a and \*b storing the start and end of linked list respectively.
2. If the sum of a->value and b->value is equal to target sum, output the pair and keep looking for next pair by moving a to next and b to previous node.
3. If target sum is smaller, move a to the next node and repeat step 2.
4. If target sum is greater, move b to the

**Code :**

#include <stdio.h>

#include <stdlib.h>

typedef struct doublyLL {

int value;

struct doublyLL \*left;

struct doublyLL \*right;

} doublyLL;

doublyLL \*createNode(int value, doublyLL \*left, doublyLL \*right) {

doublyLL \*x = (doublyLL \*)malloc(sizeof(doublyLL));

x->value = value;

x->left = left;

x->right = right;

}

void twoSum(doublyLL \*head, int sum) {

if (!head || !head->right) {

printf("ERROR : Invalid Input!");

exit(EXIT\_FAILURE);

}

doublyLL \*a = head, \*b = head->right;

while (b->right) {

b = b->right;

}

int fsum = a->value + b->value;

while (a != b) {

if (fsum > sum) {

b = b->left;

} else if (fsum < sum) {

a = a->right;

} else {

printf("(%d, %d)\n", a->value, b->value);

if (a->right == b)

return;

a = a->right;

b = b->left;

}

fsum = a->value + b->value;

}

}

void printLL(doublyLL \*head) {

printf("Linked List : ");

while (head) {

printf("%d, ", head->value);

head = head->right;

}

printf("\n");

}

int main() {

doublyLL \*head = createNode(1, NULL, NULL);

doublyLL \*l = head;

for (int i = 2; i <= 10; i++) {

doublyLL \*x =

createNode((i \* i) / 7 + i, l, NULL); // some sorted Linked List

l->right = x;

l = x;

}

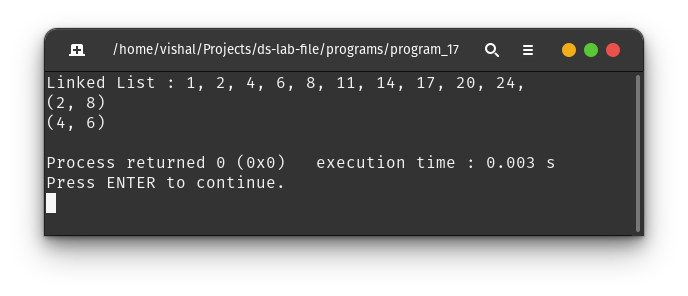
printLL(head);

twoSum(head, 10); // find pair of elements for which sum = 10

return 0;

}

**Output :**

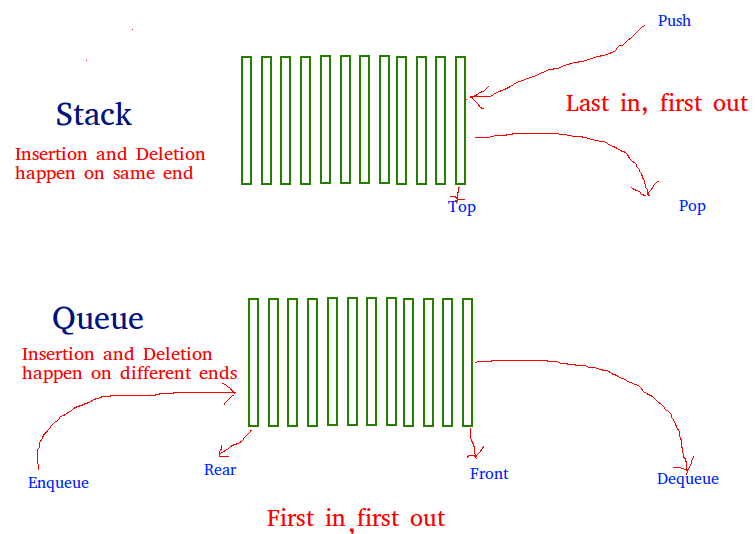


**Program 18 :** Write a C/C++ program to implement Stack Data Structure using Queue.

**Theory :**

In a queue, enqueue inserts element in the end while dequeue operation deletes the element from the front where as in stack, elements are inserted and deleted from the same side. A queue can be used to implement a stack by either making the enqueue operations costly or by making the dequeue operations costly.

**Algorithm :**



**Code :**

#include "circularQ\_impl.c"

typedef struct {

circularQ \*front;

} MyStack;

MyStack \*myStackCreate(int capacity) {

MyStack \*m = ((MyStack \*)malloc(sizeof(MyStack)));

m->front = createQ(capacity);

return m;

}

void myStackPush(MyStack \*obj, int x) { enqueue(obj->front, x); }

void myStackPop(MyStack \*obj) {

if (isQEmpty(obj->front)) {

printf("ERROR : Empty stack!");

return;

}

int size = qSize(obj->front);

while (--size) {

enqueue(obj->front, peek(obj->front));

dequeue(obj->front);

}

dequeue(obj->front);

}

int myStackTop(MyStack \*obj) {

if (isQEmpty(obj->front)) {

printf("ERROR : Empty stack!\n");

exit(EXIT\_FAILURE);

}

int size = qSize(obj->front);

while (--size) {

enqueue(obj->front, peek(obj->front));

dequeue(obj->front);

}

int x = peek(obj->front);

dequeue(obj->front);

enqueue(obj->front, x);

return x;

}

char myStackEmpty(MyStack \*obj) { return isQEmpty(obj->front); }

int main(void) {

MyStack \*stack = myStackCreate(20);

myStackPush(stack, 10);

myStackPush(stack, 11);

myStackPush(stack, 12);

printf("Top Element in stack : %d\n", myStackTop(stack));

myStackPop(stack);

printf("Top Element in stack : %d\n", myStackTop(stack));

myStackPop(stack);

printf("Top Element in stack : %d\n", myStackTop(stack));

myStackPush(stack, 30);

printf("Top Element in stack : %d\n", myStackTop(stack));

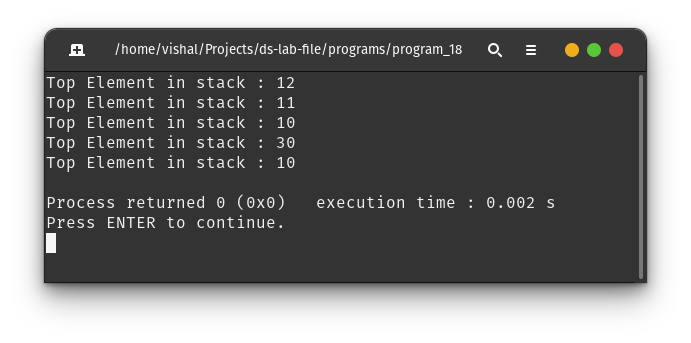
myStackPop(stack);

printf("Top Element in stack : %d\n", myStackTop(stack));

return 0;

}

**Output :**

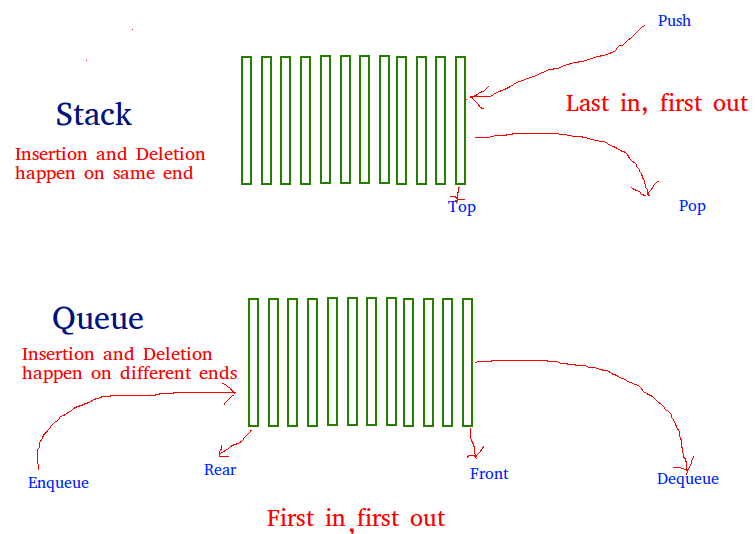


**Program 19 :** Write a C/C++ program to implement Queue Data Structure using Stack.

**Theory :**

In a queue, enqueue inserts element in the end while dequeue operation deletes the element from the front where as in stack, elements are inserted and deleted from the same side. A queue can be used to implement a stack by either making the enqueue operations costly or by making the dequeue operations costly.

**Algorithm :**



**Code :**

#include "stack\_impl.c"

typedef struct {

stack \*s;

} myQueue;

myQueue \*createQ(int capacity) {

myQueue \*q = (myQueue \*)malloc(sizeof(myQueue));

q->s = createStack(capacity);

return q;

}

void enqueue(myQueue \*q, int x) { push(q->s, x); }

void dequeue(myQueue \*q) {

if (isEmpty(q->s)) {

printf("ERROR : Queue empty!\n");

return;

}

int x = top(q->s);

if (size(q->s) != 1) {

pop(q->s);

dequeue(q);

push(q->s, x);

return;

}

pop(q->s);

}

int peek(myQueue \*q) {

if (isEmpty(q->s)) {

printf("ERROR : Queue empty!\n");

exit(EXIT\_FAILURE);

}

int x = top(q->s);

int y = x;

if (size(q->s) != 1) {

pop(q->s);

x = peek(q);

push(q->s, y);

}

return x;

}

char isQEmpty(myQueue \*q) { return isEmpty(q->s); }

int main(void) {

myQueue \*q = createQ(15);

enqueue(q, 10);

enqueue(q, 20);

enqueue(q, 30);

printf("Peek queue : %d\n", peek(q));

dequeue(q);

printf("Peek queue : %d\n", peek(q));

dequeue(q);

enqueue(q, 40);

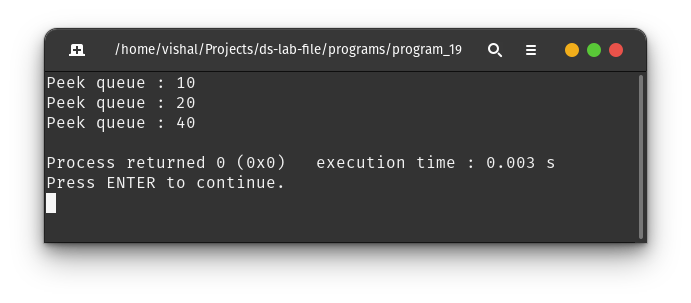
dequeue(q);

printf("Peek queue : %d\n", peek(q));

return 0;

}

**Output :**



**Program 20 :** Write a C/C++ program to implement the Binary Tree using linked list and perform In-order traversal.

**Theory :**

Linked list can be used to store data in a linear fashion unlike binary tree, hence storing the elements of Binary Tree in a linked list is degrading the structure of Binary Tree unless every node is stored which may contain a lots of nullptr. Hence it will not be recommended to store binary tree in linked list unless tree is an almost complete Binary Tree.

**Algorithm :**

Traverse the binary tree and make a Linked list node for each element and connect them properly.

**Code :**

#include "./bt\_impl.c"

#include "./ll\_impl.c"

void flattenToLL(btNode \*root, node \*\*flatten) {

if (!root) {

return;

} else {

if (root->left)

flattenToLL(root->left, flatten);

insertAtEnd(flatten, root->value);

if (root->right)

flattenToLL(root->right, flatten);

}

}

btNode \*generate() {

btNode \*x = createBtNode(5);

x->left = createBtNode(4);

x->right = createBtNode(7);

x->left->left = createBtNode(6);

x->right->right = createBtNode(8);

return x;

}

int main(void) {

btNode \*root = generate();

node \*head = NULL;

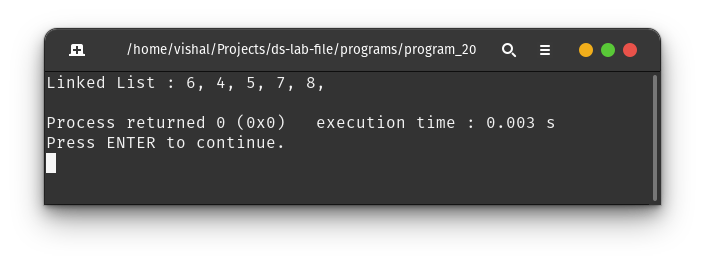
flattenToLL(root, &head);

printLL(head);

return 0;

}

**Output :**



**Program 21 :** Write a C/C++ program to check whether the given tree is a Binary Search Tree or not.

**Theory :**

Binary Search Tree is a node-based binary tree data structure which has the following properties :

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.

**Algorithm :**

On performing inorder traversal of binary tree, if all the elements are greater than their previous element (i.e – traversal is strictly descending order) then the tree will be a Binary Search Tree.

**Code :**

#include "./bt\_impl.c"

char \_isBST(btNode \*root, int \*min, char \*init) {

if (!root) {

return 1;

} else {

char x = \_isBST(root->left, min, init);

if (x == 0 || (\*init && \*min >= root->value))

return 0;

\*init = 1;

\*min = root->value;

char y = \_isBST(root->right, min, init);

return x && y;

}

}

char isBST(btNode \*root) {

int \*min = (int \*)malloc(sizeof(int));

char \*init = (char \*)malloc(sizeof(char));

return \_isBST(root, min, init);

}

btNode \*generate() {

btNode \*x = createBtNode(5);

x->left = createBtNode(4);

x->right = createBtNode(7);

x->right->left = createBtNode(6);

x->right->right = createBtNode(8);

return x;

}

int main(void) {

btNode \*root = generate();

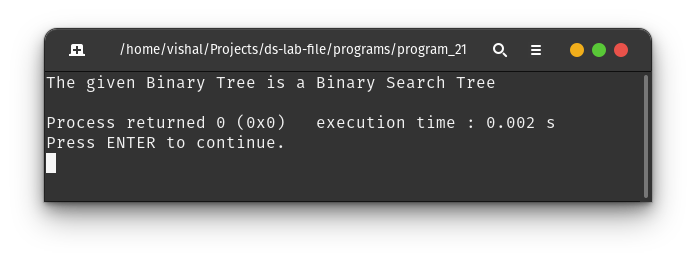
printf("The given Binary Tree is %sa Binary Search Tree\n",

isBST(root) ? "" : "not ");

return 0;

}

**Output :**



**Program 22 :** Write a C/C++ program to implement insertion in the AVL tree.

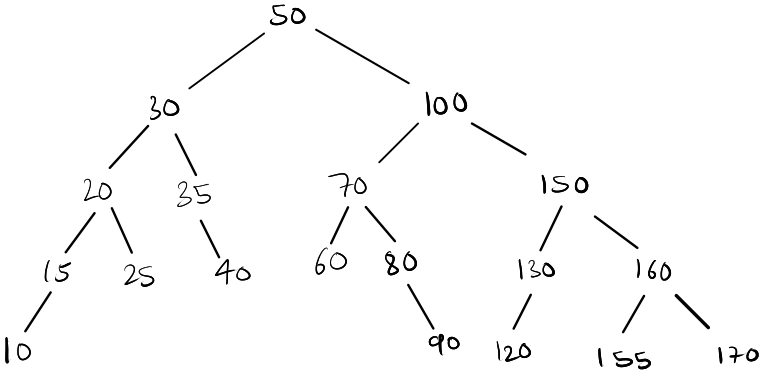
**Program 23 :** Write a C/C++ program to Delete a key from the AVL tree.

**Program 24 :** Write a C/C++ program to count the number of leaf nodes in an AVL tree.

**Theory :**

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes. Most of the BST operations (search, max, min, insert, delete, etc.) take *O(nlogn)*.

**Example of AVL Tree :**



**Algorithm :**

To make sure that the given tree remains AVL after every insertion and deletion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to balance a BST without violating the BST property (*keys(left) < key(root) < keys(right)*).

* Left Rotation
* Right Rotation

While to count leaf nodes, we can traverse the tree and add 1 to the counter whenever a node with no left and right child is reached (leaf node).

**Code :**

#define NEWLINE printf("\n")

#include <stdio.h>

#include <stdlib.h>

typedef struct avlNode {

int value;

struct avlNode \*left;

struct avlNode \*right;

int height;

} avlNode;

int max(int a, int b) { return a > b ? a : b; }

int height(avlNode \*root) {

if (!root)

return 0;

return root->height;

}

avlNode \*createAvlNode(int value) {

avlNode \*x = (avlNode \*)malloc(sizeof(avlNode));

x->value = value;

x->left = NULL;

x->right = NULL;

x->height = 1;

return x;

}

avlNode \*rightRotate(avlNode \*root) {

avlNode \*x = root->left;

avlNode \*y = x->right;

x->right = root;

root->left = y;

root->height = max(height(root->left), height(root->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

avlNode \*leftRotate(avlNode \*root) {

avlNode \*x = root->right;

avlNode \*y = x->left;

x->left = root;

root->right = y;

root->height = max(height(root->left), height(root->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

return x;

}

int balanceFactor(avlNode \*root) {

if (!root)

return 0;

return height(root->left) - height(root->right);

}

avlNode \*insert(avlNode \*root, int value) {

if (!root)

return createAvlNode(value);

if (value < root->value)

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

else if (value > root->value)

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

else

return root; // Let Duplicate values be not allowed in AVL tree

root->height = max(height(root->left), height(root->right)) + 1;

int balance = balanceFactor(root);

if (balance > 1 && value < root->left->value)

return rightRotate(root);

if (balance < -1 && value > root->right->value)

return leftRotate(root);

if (balance > 1 && value > root->left->value) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && value < root->right->value) {

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

// delete a key from avl tree

avlNode \*minValueNode(avlNode \*root) {

avlNode \*current = root;

while (current->left)

current = current->left;

return current;

}

avlNode \*delete (avlNode \*root, int value) {

if (!root)

return root;

if (value < root->value)

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

else if (value > root->value)

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

else {

if (!root->left || !root->right) {

avlNode \*temp = root->left ? root->left : root->right;

if (!temp) {

temp = root;

root = NULL;

} else

\*root = \*temp;

free(temp);

} else {

avlNode \*temp = minValueNode(root->right);

root->value = temp->value;

root->right = delete (root->right, temp->value);

}

}

if (!root)

return root;

root->height = max(height(root->left), height(root->right)) + 1;

int balance = balanceFactor(root);

if (balance > 1 && balanceFactor(root->left) >= 0)

return rightRotate(root);

if (balance > 1 && balanceFactor(root->left) < 0) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && balanceFactor(root->right) <= 0)

return leftRotate(root);

if (balance < -1 && balanceFactor(root->right) > 0) {

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

void printInorder(avlNode \*root) {

if (!root)

return;

printInorder(root->left);

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

printInorder(root->right);

}

// count number of leaf nodes in avl tree

int countLeafNodes(avlNode \*root) {

if (!root)

return 0;

if (!root->left && !root->right)

return 1;

return countLeafNodes(root->left) + countLeafNodes(root->right);

}

int main(void) {

avlNode \*root = NULL;

printf("Input number of elements to be inserted in AVL tree: ");

int n;

scanf("%d", &n);

printf("Input elements to be inserted in AVL tree: ");

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

int x;

scanf("%d", &x);

root = insert(root, x);

}

NEWLINE;

printf("Inorder traversal of the constructed AVL tree is :\n");

printInorder(root);

NEWLINE;

printf("\nNumber of leaf nodes in AVL tree : %d\n", countLeafNodes(root));

printf("\nInput number of elements to be deleted from AVL tree : ");

scanf("%d", &n);

printf("\nInput elements to be deleted from AVL tree : ");

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

int x;

scanf("%d", &x);

root = delete (root, x);

}

printf("\nInorder traversal of the AVL tree after deletion :\n");

printInorder(root);

NEWLINE;

printf("\nNumber of leaf nodes in AVL tree : %d\n", countLeafNodes(root));

return 0;

}

**Output :**

**Tree taken in the given Output after insertions and deletion**

