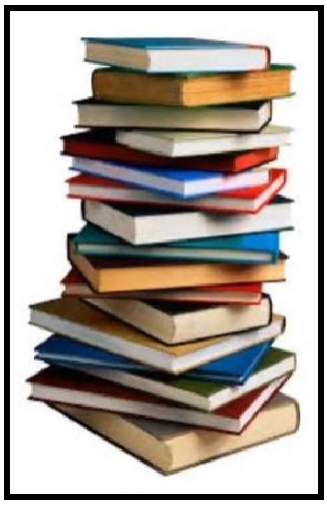
Stacks

Introduction

- It is also a linear data structure like arrays and linked lists.
- It is an abstract data type(ADT).
- This is also known as recursion type data structure.
- It follows LIFO order of data insertion/abstraction. LIFO stands for **Last In First Out.**
- Consider the example of a pile of books:



Here, unless the book at the topmost position is removed from the pile, we can't have access to the second book from the top and similarly, for the books below the second one. When we apply the same technique over the data in our program then, this pile-type structure is said to be a stack.

Like deletion, we can only insert the book at the top of the pile rather than at any other position. This means that the object/data that made its entry at the last would be one to come out first, hence known as **LIFO.**

Operations on the stack:

- **Insertion**: This is known as **push** operation.
- **Deletion**: This is known as **pop** operation.
- **Access to the topmost element:** This operation is performed by the **top** function.
- **Size of the stack:** This operation is performed by the **size** function which returns the number of elements in the stack.
- **To check for filled/empty stack:** This operation is performed by **isEmpty** function which returns boolean value. It returns true for empty stack and false otherwise.

Implementing stack using array

As in the arrays we can access any elements but here we want to use it as a stack which means we should be only allowed to access the topmost element of it. Hence, to use an array as a stack we will be keeping the access to the former as **private** in the class. This way we can put restrictions on the various operations discussed above.

Check the code below for better understanding of the approach (follow-up in the comments)...

```
#include   <climits>
class StackUsingArray {
        //Privately declared
         int *data;                              // Dynamic array created serving as stack
         int nextIndex;                          // To keep the track of current top index
         int capacity;                           // To keep the track of total size of stack

         public :
         StackUsingArray(int totalSize) {   //Constructor to initialise the values
                 data = new int[totalSize];
                 nextIndex = 0;
```

```
        capacity = totalSize;
    }

    // return the number of elements present in my stack
    int size() {
        return nextIndex;
    }

    bool isEmpty() {
        /*
        if(nextIndex == 0) {
            return true;
        }
        else {
            return false;
        }
        */

        return nextIndex == 0;        //Above program written in short-hand
    }

    // insert element
    void push(int element) {
        if(nextIndex == capacity) {
            cout << "Stack full " << endl;
            return;
        }
        data[nextIndex] = element;
        nextIndex++;                  //Size incremented
    }
    // delete element
    int pop() {
//Before deletion we checked if it was initially not empty to prevent underflow
        if(isEmpty()) {
            cout << "Stack is empty " << endl;
            return INT_MIN;
        }
        nextIndex--;                  //Conditioned satisfied so deleted
        return data[nextIndex];
    }
    //to return the top element of the stack
    int top() {
        if(isEmpty())  {              // checked for empty stack to prevent overflow
            cout << "Stack is empty " << endl;
            return INT_MIN;
        }
        return data[nextIndex - 1];
```

```
    }
};
```

Dynamic stack

There is one limitation to the above approach, which is the size of the stack is fixed. In order to overcome this limitation, whenever the size of the stack reaches its limit we will simply double its size. To get the better understanding of this approach, look at the code below…

```
#include <climits>

class StackUsingArray {
    int *data;
    int nextIndex;
    int capacity;

    public :
    StackUsingArray() {
        data = new int[4];              //initially declared with a small size of 4
        nextIndex = 0;
        capacity = 4;
    }

    // return the number of elements present in my stack
    int size() {
        return nextIndex;
    }
    bool isEmpty() {
        return nextIndex == 0;
    }

    // insert element
    void push(int element) {
        if(nextIndex == capacity) {
            int *newData = new int[2 * capacity];       //Capacity doubled
            for(int i = 0; i < capacity; i++) {
                newData[i] = data[i];                   //Elements copied
            }
            capacity *= 2;
            delete [] data;
            data = newData;
            /*cout << "Stack full " << endl;
            return;*/
```

```
            }
            data[nextIndex] = element;
            nextIndex++;
        }

        // delete element
        int pop() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return INT_MIN;
            }
            nextIndex--;
            return data[nextIndex];
        }
        int top() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return INT_MIN;
            }
            return data[nextIndex - 1];
        }
};
```

Stack using templates

While implementing the dynamic stack, we kept ourselves limited to the data of type integer only, but what if we want a generic stack(something that works for every other data type as well). For this we will be using templates. Refer the code below(based on the similar approach as used while creating dynamic stack):

```
#include <climits>

template <typename T>                    // Templates initialised
class StackUsingArray {
        T *data;                         // Template type of data used
        int nextIndex;
        int capacity;

        public :

        StackUsingArray() {
```

```
            data = new T[4];
            nextIndex = 0;
            capacity = 4;
        }
        // return the number of elements present in my stack
        int size() {
            return nextIndex;
        }
        bool isEmpty() {
            return nextIndex == 0;
        }

        // insert element
        void push(T element) {
            if(nextIndex == capacity) {
                T *newData = new T[2 * capacity];
                for(int i = 0; i < capacity; i++) {
                    newData[i] = data[i];
                }
                capacity *= 2;
                delete [] data;
                data = newData;
            }
            data[nextIndex] = element;
            nextIndex++;
        }

        // delete element
        T pop() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return 0;
            }
            nextIndex--;
            return data[nextIndex];
        }
        // For extracting top element
        T top() {
            if(isEmpty()) {
                cout << "Stack is empty " << endl;
                return 0;
            }
            return data[nextIndex - 1];
        }
};
```

You can see that every function whose return type was int initially now returns T type (i.e., template-type).

Generally, the template approach of stack is preferred as it can be used for any data type irrespective of it being int, char, float, etc…

Stack using LL

Till now we have learnt how to implement a stack using arrays, but as discussed earlier, we can also create a stack with the help of linked lists.

All the five functions that stacks can perform could be made using linked lists. Below is the template for you to work upon the same In case you are unable to come up with the program, kindly refer to the code in the solution section of the problem.

```
#include <iostream>
using namespace std;
template <typename T>
class Node {                    //Node class for Linked list
        public :
                T data;
                Node<T> *next;
                Node(T data) {
                        this -> data = data;
                        next = NULL;
                }
                ~Node() {
                        delete next;
                }
};

template <typename T>
class Stack {
        Node<T> *head;
         Node<T> *tail;
        int size;                // number of elements present in stack
```

```
        public :
        Stack() {               // Constructor to initialize the head and tail to NULL and
                                // size to zero
        }

        int getSize() {         // traverse the whole linked list and return its length

        }

        bool isEmpty() {        // Just check if the head pointer is NULL or not

        }

        void push(T element) {      // insert the newNode at the end of the list and
                                    // update the tail node
        }

        T pop() {     // remove the tail node from the list and then update the tail
                      // pointer to the previous position by traversing the list again.
        }

        T top() {    //return the value at the tail pointer. No change needed.

        }
};
```

In-built stack using STL

C++ provides the in-built stack in it's **standard temporary library (STL)** which can be used instead of creating/writing a stack class each time. To use this stack, we need to use the header file:

```
#include <stack>
```

To declare a stack use the following syntax:

```
stack <datatype_of_variables_that_will_be_stored> Name_of_stack;
```

For example: to create a stack of integer type with name 'st':

stack <int> st;

Now, you can simply perform all the operations using the in-built stack functions:

- **st.push(value_to_be_inserted)** : To insert a value in the stack
- **st.top()** : Returns the value at the top of the stack
- **st.pop()** : Deletes the value at the top from the stack.
- **st.size()** : Returns the total number of elements in the stack.
- **st.empty()** : Returns a boolean value (True for empty stack and vice versa).

Practice Problems:

- https://www.hackerrank.com/challenges/equal-stacks/problem
- https://www.hackerrank.com/challenges/simple-text-editor/problem
- https://www.techiedelight.com/design-a-stack-which-returns-minimum-element-without-using-auxiliary-stack/
- https://www.hackerearth.com/practice/data-structures/stacks/basics-of-stacks/practice-problems/algorithm/monk-and-order-of-phoenix/