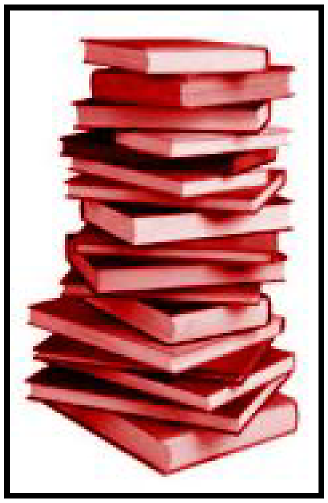
Stacks

Introduction

- Stacks are simple data structures that allow us to store and retrieve data sequentially.
- A stack is a linear data structure like arrays and linked lists.
- It is an abstract data type**(ADT)**.
- In a stack, the order in which the data arrives is essential. It follows the LIFO order of data insertion/abstraction. LIFO stands for **Last In First Out.**
- Consider the example of a pile of books:

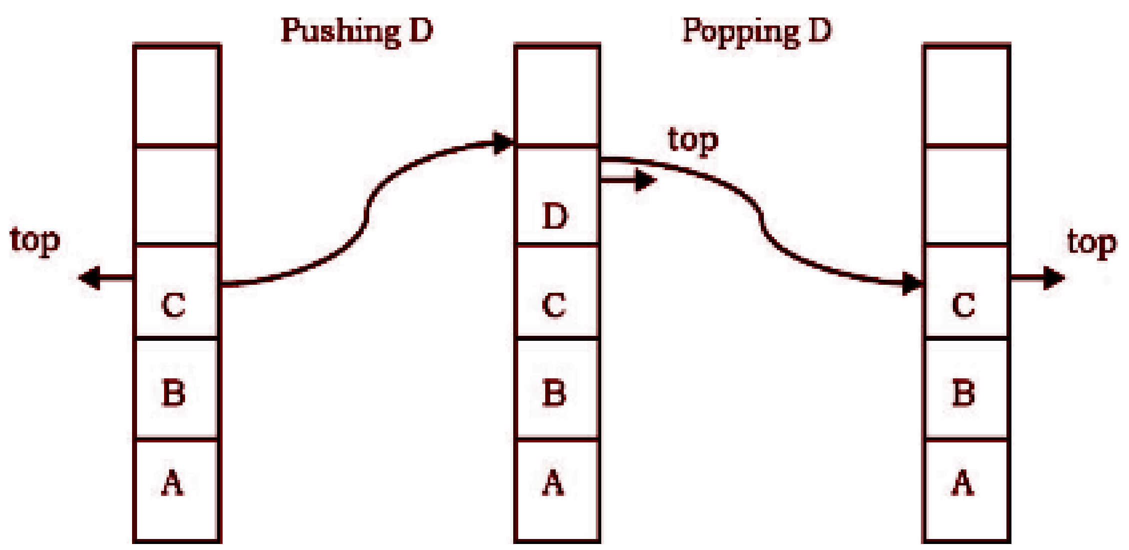


Here, unless the book at the topmost position is removed from the pile, we can't have access to the second book from the top and similarly, for the books below the second one. When we apply the same technique over the data in our program then, this pile-type structure is said to be a stack.

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:

- In a stack, insertion and deletion are done at one end, called **top**.
- **Insertion**: This is known as a **push** operation.
- **Deletion**: This is known as a **pop** operation.



Main stack operations

• `push (int data)`: Insert data onto the stack.

• `int pop()`: Removes and returns the last inserted element from the stack.

Auxiliary stack operations

• `int top()`: Returns the last inserted element without removing it.

• `int size()`: Returns the number of elements stored in the stack.

• `boolean isEmpty()`: Indicates whether any elements are stored in the stack or not.

Performance

Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

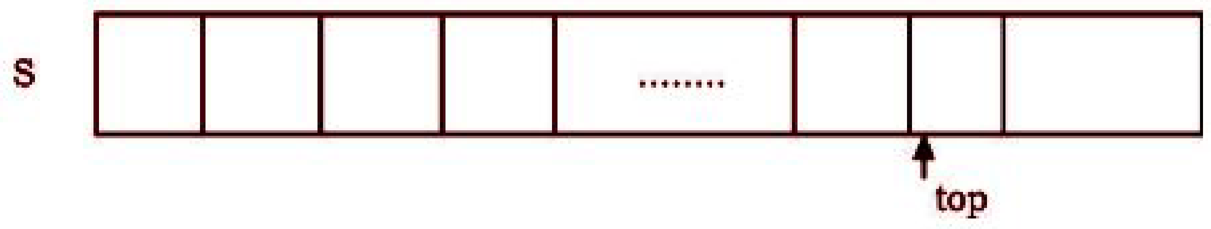
Space Complexity (for n push operations)	O(*n*)
Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

Exceptions

- Attempting the execution of an operation may sometimes cause an error condition, called an exception.
- Exceptions are said to be “thrown” by an operation that cannot be executed.
- Attempting the execution of pop() on an empty stack throws an exception called **Stack Underflow**.
- Trying to push an element in a full-stack throws an exception called **Stack Overflow**.

Implementing stack- Simple Array Implementation

This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the **top** element.



Consider the given implementation in Java for more understanding:

```
class StackUsingArray{
      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
             data = new int[totalSize];
             nextIndex = 0;
             capacity = totalSize;
       }

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

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

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

       // insert element
       public void push(int element) {
             if(nextIndex == capacity) {
                   System.out.println("Stack full");
                   return;
             }
             data[nextIndex] = element;
             nextIndex++;                            //Size incremented
       }

       // delete element
       public int pop() {
          //Before deletion check for empty to prevent underflow
             if(isEmpty()) {
                   System.out.println("Stack is empty");
                   return Integer.MIN_VALUE;
             }
```

```
            nextIndex--;              //Conditioned satisfied so deleted
            return data[nextIndex];
      }

      //to return the top element of the stack
      public int top() {
            if(isEmpty()) {                  // checked for empty stack
                  System.out.println("Stack is empty");
                  return Integer.MIN_VALUE;
            }
            return data[nextIndex - 1];
      }
}
```

Limitations of Simple Array Implementation

In programming languages like C++, Java, etc, the maximum size of an array must first be defined i.e. it is fixed and it cannot be changed.

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...

```
class StackUsingArray{
     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() {    //Constructor
            data = new int[4];
            nextIndex = 0;
            capacity = 4;
      }

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

```
	}

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

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

	// insert element
	public 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;
			data = newData;
		}
		data[nextIndex] = element;
		nextIndex++;                            //Size incremented
	}

	// delete element
	public int pop() {
	   //Before deletion check for empty to prevent underflow
		if(isEmpty()) {
			System.out.println("Stack is empty");
			return Integer.MIN_VALUE;
		}
		nextIndex--;                //Conditioned satisfied so deleted
		return data[nextIndex];
	}

	//to return the top element of the stack
	public int top() {
		if(isEmpty()) {                 // checked for empty stack
			System.out.println("Stack is empty");
			return Integer.MIN_VALUE;
		}
		return data[nextIndex - 1];
```

```
      }
}
```

Stack using templates for Generic Data type Stack

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):

```
class StackUsingArray <T>{
      T[] 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() {    //Constructor
            data = new T[4];
            nextIndex = 0;
            capacity = 4;
      }

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

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

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

      // insert element
      public void push(T element) {
            if(nextIndex == capacity) {
                  T newData[] = new T[2 * capacity]; //Capacity doubled
                  for(int i = 0; i < capacity; i++) {
                        newData[i] = data[i];            //Elements copied
                  }
                  capacity *= 2;
                  data = newData;
            }
            data[nextIndex] = element;
            nextIndex++;                               //Size incremented
      }

      // delete element
      public T pop() {
         //Before deletion check for empty to prevent underflow
            if(isEmpty()) {
                  System.out.println("Stack is empty");
                  return Integer.MIN_VALUE;
            }
            nextIndex--;                 //Conditioned satisfied so deleted
            return data[nextIndex];
      }

      //to return the top element of the stack
      public T top() {
            if(isEmpty()) {                  // checked for empty stack
                  System.out.println("Stack is empty");
                  return Integer.MIN_VALUE;
            }
            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 Generic Linked Lists

Till now we have learned 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:

```
class Node<T> {                  //Node class for Linked list
      T data;
      Node<T> next;

      Node(T data) {
            this.data = data;
            next = NULL;
      }
      Node() {
            next = null;
      }
}


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
      }

      public int getSize() {   // traverse the LL and return its length

      }

      public boolean isEmpty() {// check if the head pointer is NULL or not

      }

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

      public T pop() {  // remove the tail node and then update the tail
                        // pointer to the previous position
      }

      public T top() { //return the value at the tail node.
      }
}
```

Inbuilt Stack in Java

Java provides the in-built stack in it's **library** which can be used instead of creating/writing a stack class each time. To use this stack, we need to use the import following file:

```
import java.util.Stacks;
```

To declare a stack use the following syntax:

```
Stack <datatype_that_will_be_stored> Name_of_stack = new Stack<>();
```

There are various functions available in this module:

- **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.isEmpty()** : Returns a boolean value (True for empty stack and vice versa).

Problem Statement- Balanced Parenthesis

For a given string expression containing only round brackets or parentheses, check if they are balanced or not. Brackets are said to be balanced if the bracket which opens last, closes first. You need to return a boolean value indicating whether the expression is balanced or not.

Approach:

- We will use stacks.

- Each time, when an open parenthesis is encountered, push it in the stack, and when closed parenthesis is encountered, match it with the top of the stack and pop it.
- If the stack is empty at the end, return Balanced otherwise, Unbalanced.

Java Code:

```
public String checkBalanced(inputStr){//Function to check parentheses
    Stack<Character> s = new Stack<>(); //The stack
    for(char i : inputStr.toCharArray){
        if (i==’[’ || i==’{’ || i==’(’)
            s.push(i);
        else if (i==’]’ || i==’}’ || i==’)’)
            if (s.size()>0 && s.top()==i)
                s.pop();
            else
                return "Unbalanced";
    }
    if (s.size() == 0)
        return "Balanced";
    else
        return "Unbalanced";
}
```