PW SKILLS

Lesson Plan

Stacks - 1

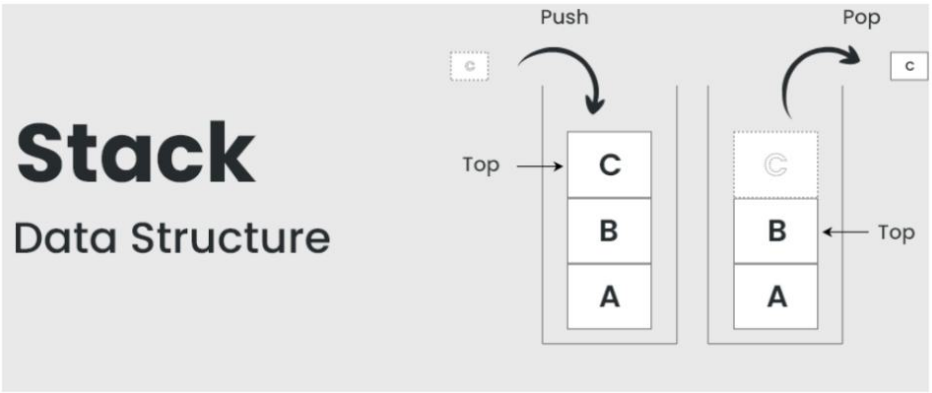
PW SKILLS

Today's checklist:

- Stack data structure
- Basic operations on stack
- Inbuilt functions of stack
- Overflow and underflow
- Practice questions

What is Stack?

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.



Basic Operations on Stack

In order to make manipulations in a stack, there are certain operations provided to us.

- push() to insert an element into the stack
- pop() to remove an element from the stack
- top() Returns the top element of the stack.
- isEmpty() returns true if stack is empty else false.
- size() returns the size of stack.

C++ provide a STL for stack, For creating a stack, we must include the <stack> header file in our code

The functions associated with stack are:

- **empty()** – Returns whether the stack is empty – Time Complexity : O(1)
- **size() –** Returns the size of the stack – Time Complexity: O(1)
- **top() –** Returns a reference to the top most element of the stack – Time Complexity: O(1)

- **Complexity:** O(1)
- **push(g):** Adds the element 'g' at the top of the stack – Time Complexity: O(1)
- **pop():** Deletes the most recent entered element of the stack – Time Complexity: O(1)

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
 stack<int> stack;
 stack.push(21);// The values pushed in the stack should be of the same data which is written during
declaration of stack
 stack.push(22);
 stack.push(24);
 stack.push(25);
 int num=0;
 stack.push(num);
 stack.pop();
 stack.pop();
 stack.pop();

 while (!stack.empty()) {
  cout << stack.top() <<" ";
  stack.pop();
 }
}
```

Q. Reverse a stack

Input: St : {3,2,1,7,6}

Output: {6,7,1,2,3}

Code:

```
//User function Template for C++

class Solution{
public:
    void reverseStack(std::stack<int>& original) {
    std::stack<int> tempStack;

    // Transfer elements from original stack to tempStack
    while (!original.empty()) {
        tempStack.push(original.top());
        original.pop();
    }

    // Transfer elements from tempStack back to original stack
    while (!tempStack.empty()) {
        original.push(tempStack.top());
        tempStack.pop();
    }
}
```

Q. Copy stack into another stack in same order

Code:

```
void copyStack(std::stack<int>& original, std::stack<int>&
destination) {
    std::stack<int> tempStack; // Temporary stack to preserve order

    // Copy elements from original stack to temporary stack
    while (!original.empty()) {
        tempStack.push(original.top());
        original.pop();
    }

    // Copy elements from temporary stack to destination stack
    while (!tempStack.empty()) {
        destination.push(tempStack.top());
        original.push(tempStack.top()); // Restoring the original
stack
        tempStack.pop();
    }
}
```

Q. Display Stack

Code:

```
// Function to display the contents of a stack without modifying it
void displayStack(std::stack<int> myStack) {
    std::cout << "Stack contents: ";
    while (!myStack.empty()) {
        std::cout << myStack.top() << " ";
        myStack.pop();
    }
    std::cout << std::endl;
}
```

Q. Given a stack S and an integer N, the task is to insert N at the bottom of the stack.

Code:

```
stack<int> recur(stack<int> S, int N)
{
    // If stack is empty
    if (S.empty())
        S.push(N);

    else {

        // Stores the top element
        int X = S.top();

        // Pop the top element
        S.pop();

        // Recurse with remaining elements
        S = recur(S, N);

        // Push the previous
        // top element again
        S.push(X);
    }
    return S;
}

// Function to insert an element
// at the bottom of stack
void insertToBottom(
    stack<int> S, int N)
{

    // Recursively insert
    // N at the bottom of S
    S = recur(S, N);

    // Print the stack S
    while (!S.empty()) {
        cout << S.top() << " ";
        S.pop();
    }
}
```

Q. Reverse stack recursively

Code:

```
class Solution{
public:
    void insert(stack<int>& s,int x){
        if(s.empty()){
            s.push(x);
            return;
        }
        int curr=s.top();
        s.pop();
        insert(s,x);
        s.push(curr);
    }
    void Reverse(stack<int> &St){
        if(St.size()==1) return;
        int curr=St.top();
        St.pop();
        Reverse(St);
        insert(St,curr);
    }
};
```

Overflow:

Stack overflow in C++ occurs when attempting to add elements beyond the predefined capacity of a stack, causing memory overreach. This issue often arises in fixed-size stack implementations, leading to memory corruption or program crashes. To avoid stack overflow, use dynamic data structures or perform capacity checks before adding elements. Handling overflow conditions proactively ensures program stability and prevents unexpected errors.

Underflow:

In C++, stack underflow happens when trying to pop an element from an empty stack, causing an attempt to access or remove an element from a stack that has no elements. This situation often occurs when a pop operation is executed on an already empty stack. Handling underflow involves checking if the stack is empty before performing a pop operation to prevent accessing or removing elements that don't exist, ensuring the program operates safely without encountering unexpected errors.

Array/Vector Implementation

```
// C++ Program to Implement stack using Class Templates

// Including input output libraries
#include <iostream>
// Header File including all string functions
```

```
#include <string>

using namespace std;

// Taking size of stack as 10
#define SIZE 5

// Class
// To represent stack using template by class
// taking class in template
template <class T> class Stack {
 // Public access modifier
public:
 // Empty constructor
 Stack();

 // Method 1
 // To add element to stack which can be any type
 // using stack push() operation
 void push(T k);

 // Method 2
 // To remove top element from stack
 // using pop() operation
 T pop();

 // Method 3
 // To print top element in stack
 // using peek() method
 T topElement();

 // Method 4
 // To check whether stack is full
 // using isFull() method
 bool isFull();

 // Method 5
 // To check whether stack is empty
 // using isEmpty() method
 bool isEmpty();

private:
 // Taking data member top
 int top;

 // Initialising stack(array) of given size
 T st[SIZE];
};

// Method 6
// To initialise top to
// -1(default constructor)
```

```
template <class T> Stack<T>::Stack() { top = -1; }

// Method 7
// To add element element k to stack
template <class T> void Stack<T>::push(T k)
{

 // Checking whether stack is completely filled
 if (isFull()) {
  // Display message when no elements can be pushed
  // into it
  cout << "Stack is full\n";
 }

 // Inserted element
 cout << "Inserted element " << k << endl;

 // Incrementing the top by unity as element
 // is to be inserted
 top = top + 1;

 // Now, adding element to stack
 st[top] = k;
}

// Method 8
// To check if the stack is empty
template <class T> bool Stack<T>::isEmpty()
{
 if (top == -1)
  return 1;
 else

 return 0;
}

// Utility methods

// Method 9
// To check if the stack is full or not
template <class T> bool Stack<T>::isFull()
{
 // Till top in inside the stack
 if (top == (SIZE - 1))
  return 1;
 else

// As top can not exceeds th size
  return 0;
}
```

```
// Method 10
template <class T> T Stack<T>::pop()
{
 // Initialising a variable to store popped element
 T popped_element = st[top];

 // Decreasing the top as
 // element is getting out from the stack
 top--;

 // Returning the element/s that is/are popped
 return popped_element;
}

// Method 11
template <class T> T Stack<T>::topElement()
{
 // Initialising a variable to store top element
 T top_element = st[top];

 // Returning the top element
 return top_element;
}

// Method 12
// Main driver method
int main()
{
 // Creating object of Stack class in main() method
 // Declaring objects of type Integer and String
 Stack<int> integer_stack;
 Stack<string> string_stack;

 // Adding elements to integer stack object
 // Custom integer entries
 integer_stack.push(2);
 integer_stack.push(54);
 integer_stack.push(255);

 // Adding elements to string stack
 // Custom string entries
 string_stack.push("Welcome");
 string_stack.push("to");
 string_stack.push("PhysicsWallah");

 // Now, removing element from integer stack
 cout << integer_stack.pop() << " is removed from stack"
  << endl;
```

```
 // Now, removing element from integer stack
 cout << integer_stack.pop() << " is removed from stack"
  << endl;

 // Removing top element from string stack
 cout << string_stack.pop() << " is removed from stack "
  << endl;

 // Print and display the top element in integer stack
 cout << "Top element is " << integer_stack.topElement()
  << endl;

 // Print and display the top element in string stack
 cout << "Top element is " << string_stack.topElement()
  << endl;

 return 0;
}
```

LinkedList Implementation

```
// C++ program to Implement a stack
// using singly linked list
#include <bits/stdc++.h>
using namespace std;

// creating a linked list;
class Node {
public:
 int data;
 Node* link;

 // Constructor
 Node(int n)
 {
  this->data = n;
  this->link = NULL;
 }
};

class Stack {
 Node* top;

public:
 Stack() { top = NULL; }
```

```
 void push(int data)
 {

  // Create new node temp and allocate memory in heap
  Node* temp = new Node(data);

  // Check if stack (heap) is full.
  // Then inserting an element would
  // lead to stack overflow
  if (!temp) {
   cout << "\nStack Overflow";
   exit(1);
  }

// Initialize data into temp data field
  temp->data = data;

  // Put top pointer reference into temp link
  temp->link = top;

  // Make temp as top of Stack
  top = temp;
 }

 // Utility function to check if
 // the stack is empty or not
 bool isEmpty()
 {
  // If top is NULL it means that
  // there are no elements are in stack
  return top == NULL;
 }

 // Utility function to return top element in a stack
 int peek()
 {
  // If stack is not empty , return the top element
  if (!isEmpty())
   return top->data;
  else
   exit(1);
 }

// Function to remove
 // a key from given queue q
 void pop()
 {
  Node* temp;

  // Check for stack underflow
  if (top == NULL) {
```

```
cout << "\nStack Underflow" << endl;
   exit(1);
  }
  else {

   // Assign top to temp
   temp = top;

   // Assign second node to top
   top = top->link;

   // This will automatically destroy
   // the link between first node and second node

   // Release memory of top node
   // i.e delete the node
   free(temp);
  }
 }

 // Function to print all the
 // elements of the stack
 void display()
 {
  Node* temp;

  // Check for stack underflow
  if (top == NULL) {
   cout << "\nStack Underflow";
   exit(1);
  }
  else {
   temp = top;
   while (temp != NULL) {

    // Print node data
    cout << temp->data;

    // Assign temp link to temp
    temp = temp->link;
    if (temp != NULL)
     cout << " -> ";
   }
  }
 }
};
```

```
int main()
{
 // Creating a stack
 Stack s;

 // Push the elements of stack
 s.push(11);
 s.push(22);
 s.push(33);
 s.push(44);

 // Display stack elements
 s.display();

 // Print top element of stack
 cout << "\nTop element is " << s.peek() << endl;

 // Delete top elements of stack
 s.pop();
 s.pop();

 // Display stack elements
 s.display();

 // Print top element of stack
 cout << "\nTop element is " << s.peek() << endl;

 return 0;
}
```

Linked List VS Vector Implementation

Linked List Implementation:

Advantages:

- **Dynamic Size:** Linked lists allow for a dynamic size without pre-allocating memory, making them suitable for situations where the size of the stack is uncertain.

- **Efficient Insertion/Deletion:** Insertion and deletion of elements at the top of the stack are efficient (constant time complexity - O(1)).

- **No Fixed Capacity:** Linked lists can grow as needed without the limitation of a fixed capacity.

Disadvantages:

- **Extra Memory Overhead:** Each element requires additional memory for the linked list node pointer.

- **Less Efficient Random Access:** Accessing elements at arbitrary positions is slower compared to vectors due to sequential traversal.

Vector (Dynamic Array) Implementation:

Advantages:

- **Random Access:** Vectors allow direct access to elements using indexing, enabling faster access compared to linked lists.

- **Simple Implementation:** Implementing a stack with a vector can be straightforward and easy to understand.

- **Possibility of Preallocation**: Capacity can be preallocated for better performance if the maximum size of the stack is known.

Disadvantages:

- **Dynamic Reallocation:** Resizing a vector might involve reallocation and copying of elements to a new memory location, which can be time-consuming.

- **Fixed Capacity Limitation:** In some scenarios, the vector might have a fixed capacity, leading to potential overflow if exceeded.

When to Use Each:

- Linked List Implementation:
- Suitable when the size of the stack is uncertain or needs to change dynamically.
- Better for scenarios requiring frequent insertion and deletion at the top.
- Vector Implementation:
- Suitable when fast random access and better performance for element access are required.
- Useful when the maximum size of the stack is known and fixed.

Summary:

- Linked list-based stacks offer dynamic sizing and efficient insertion/deletion but may consume more memory due to the node pointers.

- Vector-based stacks provide fast random access and are easier to manage, but resizing can be inefficient and may have a fixed capacity.

The choice between the two implementations depends on factors like the expected size of the stack, the frequency of insertion/deletion, and the need for random access or a fixed capacity.