Stacks

Topics

1. Introduction to Stacks
2. Implementation of a Stack using Array in C/C++
3. Implementation of a Stack using Linked List in C/C++
4. Details of Inbuilt Stack library in C++, Java & Python
5. Problems & Solutions

Introduction

Stack is a Linear Data Structure which 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 says that, whatever was the last inserted element onto the stack, will be the first element to be popped out of the stack. FILO is just the same saying as LIFO, but the other way around.

Stacks primarily support the following three basic operations :-

1. **Push:** Adds/Pushes an item on to the top of the stack. If the stack is full while pushing an element, then it is said to be a *Stack Overflow* condition *i.e.*, we can't push any more items onto the stack.
   Complexity : O(1)

2. **Pop:** Removes/Pops an item from the top of the stack. The item that is popped first, is the most recent item that has been pushed on to the stack. If the stack is empty while popping, then it is said to be a *Stack Underflow* condition *i.e.*, There are no more items in the stack to be popped.
   Complexity : O(1)

3. **Top:** Get the topmost item. If the stack is empty, just return -1 (or INT_MIN).
   Complexity : O(1)

How to understand a stack practically?
There are many real life examples of stack. Consider the simple example of plates stacked over one another in a restaurant. The plate which is at the top is the first one to be removed, *i.e.,* the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it simply follows LIFO/FILO order.

In terms of programming, the browser has a forward and backward button to go back to the previously visited page. This functionality of going back/forward is same as popping/pushing an element from/to the top of the stack. Therefore, every browser uses the stack data structure in this manner.

Implementation

There are two ways to implement a stack :-

- Using an Array.
- Using a Linked List.

Implementation of a Stack using an Array in C/C++

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure 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 get top item from stack
int peek(struct Stack* stack) {
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}

int main() {
    struct Stack* stack = createStack(100);

    push(stack, 10);
    push(stack, 20);
    push(stack, 30);

    printf("%d popped from stack\n", pop(stack));
    printf("Top item is %d\n", peek(stack));
    return 0;
}
```

Implementation of a Stack using a Linked List in C/C++

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent a stack
struct StackNode {
    int data;
    struct StackNode* next;
};

// Function to create a new node
struct StackNode* newNode(int data) {
    struct StackNode* stackNode = (struct StackNode*) malloc(sizeof(struct StackNode));
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

// Function to know if the stack is Empty or not
int isEmpty(struct StackNode *root) {
    return !root;
}

// Function to push a new item onto the top of the stack
void push(struct StackNode** root, int data) {
    struct StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    printf("%d pushed to stack\n", data);
}

// Function to pop an item from the top of the stack
int pop(struct StackNode* *root) {
    if (isEmpty(*root))
        return INT_MIN;
    struct StackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);
    return popped;
}

// Function to know the value of the top most item of the stack
int peek(struct StackNode* root) {
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
}

// Driver Program
int main() {
    struct StackNode* root = NULL;
    push(&root, 10);
    push(&root, 20);
    push(&root, 30);
    printf("%d popped from stack\n", pop(&root));
    printf("Top element is %d\n", peek(root));
    return 0;
}
```

- **Linked List Implementation & Array Implementation of Stack in Java.**
- **Linked List Implementation & Array/List Implementation of a Stack in Python.**

Details of Inbuilt Stack

- **C++**

  Declaration:

  ```
  stack<int> myStack;
  ```

  Functions:

  - Get the top element: `myStack.top();`
  - Push a new element: `myStack.push(20);`
  - Pop the top element: `myStack.pop();`
  - Check if the Stack is Empty: `myStack.empty();`
  - Get the Size of the stack: `myStack.size();`

  More details at **CPPReference - Stack.**

- Library Details for **Java's Stack Class: Java Docs** & **How to use Stack Class in Java.**

- In **Python,** a **List** can be used as a **Stack: How to use a Python List as a Stack.**

Problems & Solutions

1. Design a data structure for `top()`, `push()`, `pop()` and `getMin()` operations. Make sure that all the aforementioned operations take only O(1) time to execute.
   Solution: Maintain two stacks, one for the actual stack elements and other for maintaining minimum element in the stack so far.

2. Find the first greater element on right side for every element `ar[i]` in a given array.
   Solution: Start from right side of the array and maintain a stack of elements processed so far. Now, for every element `ar[i]`, pop all elements which are less than current element from the stack and set the top of stack as the next greater element for `ar[i]`. Push `ar[i]` in the stack. Repeat this for all elements (from right hand-side).

3. Evaluate the given Postfix Expression.
   Eg: Input: `2 3 1 * + 9 -`
   Output: `-4`
   [**Note:** The Infix Expression is: `3 * 1 + 2 - 9`]
   Solution: Keep pushing the operands onto a stack. Whenever you encounter an operator, pop two operands from the stack, perform operation and push the result back into the stack.

4. Implement two stacks in a single array.
   Solution: Create 1st stack from the start of the array and grow it rightwards. Create the 2nd stack from the end of the array and grow it leftwards. Use two pointers `top1` and `top2` for the 2 stacks respectively, and check `(top2 - top1) > 1` to ensure the array isn't full before any new element insertion/push.

5. Check for balanced parentheses in an expression.
   Solution: When you encounter an opening bracket, push it onto the stack. When you encounter a closing bracket, if stack is not empty, pop top element from the stack and match it with the current closing bracket type, else report "Invalid". If the stack is empty at the end of the process, report "Valid", else report "Invalid".

6. Reverse a string of words.
   Eg: Input: `"Algorithms and Data Structures"`
   Output: `"Structures Data and Algorithms"`
   Solution: Simply push all the words of the string onto a stack and pop them.

7. The Stock Span problem is a financial problem where we have a series of ***n*** daily price quotes for a stock and we need to calculate span of stock's price for all ***n*** days. The span $\mathbf{S_i}$ of the stock's price on a given day ***i*** is defined as the maximum number of consecutive days just before the given day, for which the price of the stock on the current day is less than or equal to its price on the given day. For example, if an array of 7 days prices is given as `{100, 80, 60, 70, 60, 75, 85}`, then the span values for corresponding 7 days are `{1, 1, 1, 2, 1, 4, 6}`.
   Solution: The solution can be broken down to finding the *next greater element on the left hand-side*. Solution will be similar to Problem-2 above.