Stacks

Topics

1. Introduction
2. Implementation Using Array and Linked List
3. C++ STL Stack
4. Problems

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).

Stacks primarily supports the following three basic operations

1. **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
   Complexity : O(1)

2. **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
   Complexity : O(1)

3. **Top:** Get the topmost item.
   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 canteen. 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 can be simply seen to follow LIFO/FILO order.

Implementation

There are two ways to implement a stack:

- Using array
- Using linked list

Implementation Using Array

```
#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 Using Linked List

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

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

struct StackNode* newNode(int data) {
    struct StackNode* stackNode = (struct StackNode*) malloc(sizeof(struct StackNode));
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

int isEmpty(struct StackNode *root) {
    return !root;
}
```

```
void push(struct StackNode** root, int data) {
    struct StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    printf("%d pushed to stack\n", data);
}

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;
}

int peek(struct StackNode* root) {
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
}

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;
}
```

Usage Details for C++

Declaration
stack<int> myStack;

Functions

- Top element: myStack.top();
- Push new element: myStack.push();
- Pop top element: myStack.pop();
- Check Empty: myStack.empty();
- Size of the stack: myStack.size();

More details at http://www.cplusplus.com/reference/stack/stack/.

Usage Details for Java can be found at
https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html.

Problems

1. Design a data structure for top(), push(), pop() and getMin() operations in O(1) time complexity.
   Solution: Maintain two stacks, one for the actual array elements and other for maintaining minimum element 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 a[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 for all elements (from right hand-side)

3. Evaluation of Postfix Expression
   Example:
   Input: 2 3 1 * + 9 -
   Output -4 [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 an 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 repectively, and check *(top2 –top1) > 1* to ensure the array isnt full before any new insertion.

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, 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
   Solution: Simple 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 Si 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-size*. Solution will be similar to Problem-2 above.