Stack

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).

- **Push**: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **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.
- **Peek** or **Top**: Returns top element of stack.
- **isEmpty**: Returns true if stack is empty, else false.
- **Peep** and **Change**

stack practically/ real life

- 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.

Time Complexities

- push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

Applications

- Balancing of **symbols**
- Infix to **Postfix /Prefix** conversion
- **Redo-undo** features at many places like editors, photoshop.
- **Forward** and **backward** feature in web browsers
- Used in many algorithms like **Tower of Hanoi**, **tree traversals**, **stock span problem**, **histogram problem**.
- Other applications can be **Backtracking**, **Knight tour problem**, **rat in a maze**, **N queen problem** and **sudoku solver**
- In Graph Algorithms like **Topological Sorting** and **Strongly Connected Components**

Implementation

- Using array
  - Pros: Easy to implement. Memory is saved as pointers are not involved.
  - Cons: It is not dynamic. It doesn’t grow and shrink depending on needs at runtime.
- Using linked list
  - Pros: The linked list implementation of stack can grow and shrink according to the needs at runtime.
  - Cons: Requires extra memory due to involvement of pointers.

What this function does?

```
void fun(int n){
    Stack S;  // Say it creates an empty stack S
    while (n > 0) {
      push(&S, n%2);
      n = n/2;
    }
    while (!isEmpty(&S))
      printf("%d ", pop(&S));
}
```

It prints binary representation of n

```
void bin(unsigned n) {
    /* step 1 */
    if (n > 1)
        bin(n/2);
    /*step 2 */
    cout << n % 2;
}
int main(void) {
    bin(7);
    cout << endl;
    bin(4);
}
```

```
step 1) if NUM > 1
    a) push NUM on stack
    b) recursively call function with 'NUM / 2'
step 2)
    a) pop NUM from stack, divide it by 2 and print
it's remainder.
```

```
Output:
111
100
```

Queue using Stacks

- Method 1 (By making enQueue/ Insert operation costly)
- This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

Queue using Stacks

- Method 1 (By making enQueue/ Insert operation costly)
- enQueue(q, x)
- 1) While stack1 is not empty, push everything from stack1 to stack2.
- 2) Push x to stack1 (assuming size of stacks is unlimited).
- 3) Push everything back to stack1.
- Here time complexity will be O(n)
- deQueue(q)
- 1) If stack1 is empty then error
- 2) Pop an item from stack1 and return it
- Here time complexity will be O(1)

Queue using Stacks

- Method 2 (By making deQueue operation costly)
- In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

Queue using Stacks

- Method 2 (By making deQueue operation costly)
- enQueue(q, x)
- 1) Push x to stack1 (assuming size of stacks is unlimited).
- Here time complexity will be O(1)
- deQueue(q)
- 1) If both stacks are empty then error.
- 2) If stack2 is empty
- While stack1 is not empty, push everything from stack1 to stack2.
- 3) Pop the element from stack2 and return it.
- Here time complexity will be O(n)

Queue using Stacks

- Method 2 is definitely better than method 1.
- Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only if stack2 empty.

Special Implementation

- Design a Data Structure SpecialStack that supports all the stack operations like push(), pop(), isEmpty(), isFull() and an additional operation getMin() which should return minimum element from the SpecialStack.
- All these operations of SpecialStack must be O(1). To implement SpecialStack, you should only use standard Stack data structure and no other data structure like arrays, list, .. etc.

- Consider the following SpecialStack
- 16 --> TOP
- 15
- 29
- 19
- 18
- When getMin() is called it should return 15, which is the minimum
- element in the current stack.

- If we do pop two times on stack, the stack becomes
- 29 --> TOP
- 19
- 18
- When getMin() is called, it should return 18 which is the minimum in the current stack.

Solution

- Use two stacks: one to store actual stack elements and other as an auxiliary stack to store minimum values. The idea is to do push() and pop() operations in such a way that the top of auxiliary stack is always the minimum.

Solution contd...

- Push(int x) // inserts an element x to Special Stack
- 1) push x to the first stack (the stack with actual elements)
- 2) compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.
- a) If x is smaller than y then push x to the auxiliary stack.
- b) If x is greater than y then push y to the auxiliary stack.

Solution contd...

- int Pop() // removes an element from Special Stack and return the removed element
- 1) pop the top element from the auxiliary stack.
- 2) pop the top element from the actual stack and return it.

- The step 1 is necessary to make sure that the auxiliary stack is also updated for future operations.

Solution contd...

- int getMin() // returns the minimum element from Special Stack
- 1) Return the top element of auxiliary stack.

Push(18)
Push(19)
Push(29)
Push(15)
push(16)

Actual Stack	Auxiliary Stack
16 <--- top	15 <---- top
15	15
29	18
19	18
18	18

Solution Contd...

- The above approach can be optimized.
- We can limit the number of elements in auxiliary stack.
- We can push only when the incoming element of main stack is smaller than or equal to top of auxiliary stack.
- Similarly during pop, if the pop off element equal to top of auxiliary stack, remove the top element of auxiliary stack.

Two Stacks in One Array

- Method 1 (Divide the space in two halves)
- A simple way to implement two stacks is to divide the array in two halves and assign the half half space to two stacks, i.e.,
- use arr[0] to arr[n/2] for stack1, and
- arr[(n/2) + 1] to arr[n-1] for stack2
- where arr[] is the array to be used to implement two stacks and size of array be n.

The problem with this method is inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[].

- Method 2 (A space efficient implementation)
- The idea is to start two stacks from two extreme corners of arr[].
- stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0.
- The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1).
- Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks.
- This method efficiently utilizes the available space. It doesn’t cause an overflow if there is space available in arr[]

- Similarly you can design an implentation of N stacks in single array.

Stack - Merge

- Design a stack with following operations.

- a) push(Stack s, x): Adds an item x to stack s
- b) pop(Stack s): Removes the top item from stack s
- c) merge(Stack s1, Stack s2): Merge contents of s2 into s1.

- Time Complexity of all above operations should be O(1).

- If we use array implementation of stack, then merge is not possible to do in O(1) time as we have to do following steps.
- a) Delete old arrays
- b) Create a new array for s1 with size equal to size of old array for s1 plus size of s2.
- c) Copy old contents of s1 and s2 to new array for s1
- The above operations take O(n) time.

- We can use a linked list with two pointers,
- one pointer to first node (also used as top when elements are added and removed from beginning).
- The other pointer is needed for last node so that we can quickly link the linked list of s2 at the end of s1.
- Following are all operations.
- a) push(): Adds the new item at the beginning of linked list using first pointer.
- b) pop(): Removes an item from beginning using first pointer.
- c) merge(): Links the first pointer second stack as next of last pointer of first list.

Can we do it if we are not allowed to use extra pointer?

- We can do it with circular linked list. The idea is to keep track of last node in linked list. The next of last node indicates top of stack.
- a) push(): Adds the new item as next of last node.
- b) pop(): Removes next of last node.
- c) merge(): Links the top (next of last) of second list to the top (next of last) of first list. And makes last of second list as last of whole list.