Data Structures & Algorithms

Stacks

B.Bhuvaneswaran, AP (SG) / CSE

9791519152

bhuvaneswaran@rajalakshmi.edu.in

RAJALAKSHMI ENGINEERING COLLEGE
An AUTONOMOUS Institution
Affiliated to ANNA UNIVERSITY, Chennai

Stacks

- A stack is an ordered collection of elements where elements are only added and removed from the same end.

Examples

- In the physical world, an example of a stack would be a stack of plates in a kitchen - you add plates or remove plates from the top of the pile.
- In the software world, a good example of a stack is the history of your current browser's tab.
- Let's say you're on site A, and you click on a link to go to site B, then from B you click on another link to go to site C.
- Every time you click a link, you are adding to the stack - your history is now [A, B, C].
- When you click the back arrow, you are "removing" from the stack - click it once and you have [A, B], click it again and you have [A].

Note

- Another term used to describe stacks is LIFO, which stands for last in, first out.
- The last (most recent) element placed inside is the first element to come out.

Stacks

- Stacks are very simple to implement.
- Some languages like Java have built-in stacks.
- In Python, you can just use a list stack = [] and use stack.append(element) and stack.pop().
- In fact, any dynamic array can implement a stack.
- Typically, inserting into a stack is called pushing and removing from a stack is called popping.
- Stacks will usually also come with operations like peek, which means looking at the element at the top of the stack.

Time Complexity

- The time complexity of stack operations is dependent on the implementation.
- If you use a dynamic array, which is the most common and easiest way, then the time complexity of your operations is the same as that of a dynamic array.
- O(1) push, pop, and random access, and O(n) search. Sometimes, a stack may be implemented with a linked list with a tail pointer.

Note

- The characteristic that makes something a "stack" is that you can only add and remove elements from the same end.
- It doesn't matter how you implement it, a "stack" is just an abstract interface.

Note

- Stacks and recursion are very similar.
- This is because recursion is actually done using a stack. Function calls are pushed on a stack.
- The call at the top of the stack at any given moment is the "active" call.
- On a return statement or the end of the function being reached, the current call is popped off the stack.

Algorithm Problems

- For algorithm problems, a stack is a good option whenever you can recognize the LIFO pattern.
- Usually, there will be some component of the problem that involves elements in the input interacting with each other.
- Interacting could mean matching elements together, querying some property such as "how far is the next largest element", evaluating a mathematical equation given as a string, just comparing elements against each other, or any other abstract interaction.

Interface guide

```
// Declaration: Java supports multiple implementations, but we will be using
// the Stack interface with the Stack implementation. Specify the data type
Stack<Integer> stack = new Stack<>();

// Pushing elements:
stack.push(1);
stack.push(2);
stack.push(3);
```

Interface guide

```
// Popping elements:
stack.pop(); // 3
stack.pop(); // 2

// Check if empty
stack.empty(); // false

// Check element at top
stack.peek(); // 1

// Get size
stack.size(); // 1
```

Example

```
public class Example {
   public static void main(String[] args) {
      Stack<Integer> stack = new Stack<>();

      stack.push(1);
      stack.push(2);
      stack.push(3);

      System.out.println(stack.pop());
      System.out.println(stack.pop());
      System.out.println(stack.pop());

      stack.push(5);

      if (stack.empty()) {
         System.out.println("Stack is empty!");
      } else {
         System.out.println(String.format("Stack is not empty, top is: %d", stack.peek()));
      }
   }
}
```

String problems

- String questions involving stacks are popular.
- Normally, string questions that can utilize a stack will involve iterating over the string and putting characters into the stack, and comparing the top of the stack with the current character at each iteration.
- Stacks are useful for string matching because it saves a "history" of the previous characters.

Valid Parentheses

- Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
- The string is valid if all open brackets are closed by the same type of closing bracket in the correct order, and each closing bracket closes exactly one open bracket.
- For example, s = "({})" and s = "(){}[]" are valid, but s = "(]" and s = "({)}" are not valid.

Examples

- Input:
  - s = "{([]){}}"
- Output:
  - true
- Input:
  - s = "{([}])"
- Output:
  - false

Remove All Adjacent Duplicates In String

- You are given a string s.
- Continuously remove duplicates (two of the same character beside each other) until you can't anymore.
- Return the final string after this.
- For example, given s = "abbaca", you can first remove the "bb" to get "aaca". Next, you can remove the "aa" to get "ca". This is the final answer.

Backspace String Compare

- Given two strings s and t, return true if they are equal when both are typed into empty text editors. '#' means a backspace character.
- For example, given s = "ab#c" and t = "ad#c", return true. Because of the backspace, the strings are both equal to "ac".

Example

- Input:
  - s = "ab#c"
  - t = "ad#c"
- Output:
  - true

Queries?

Thank You…!