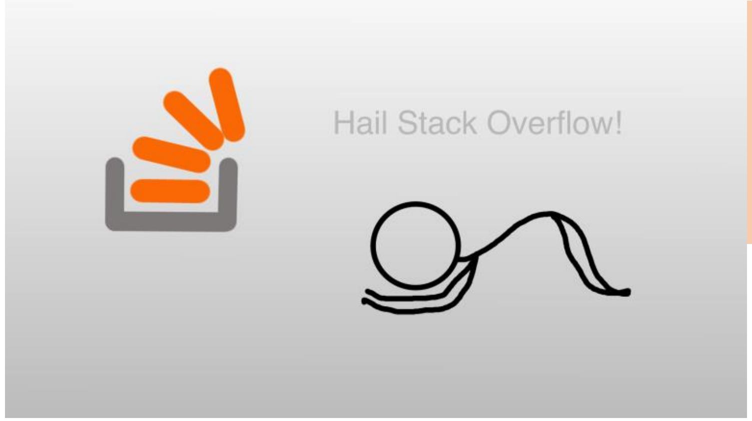
Ch.05



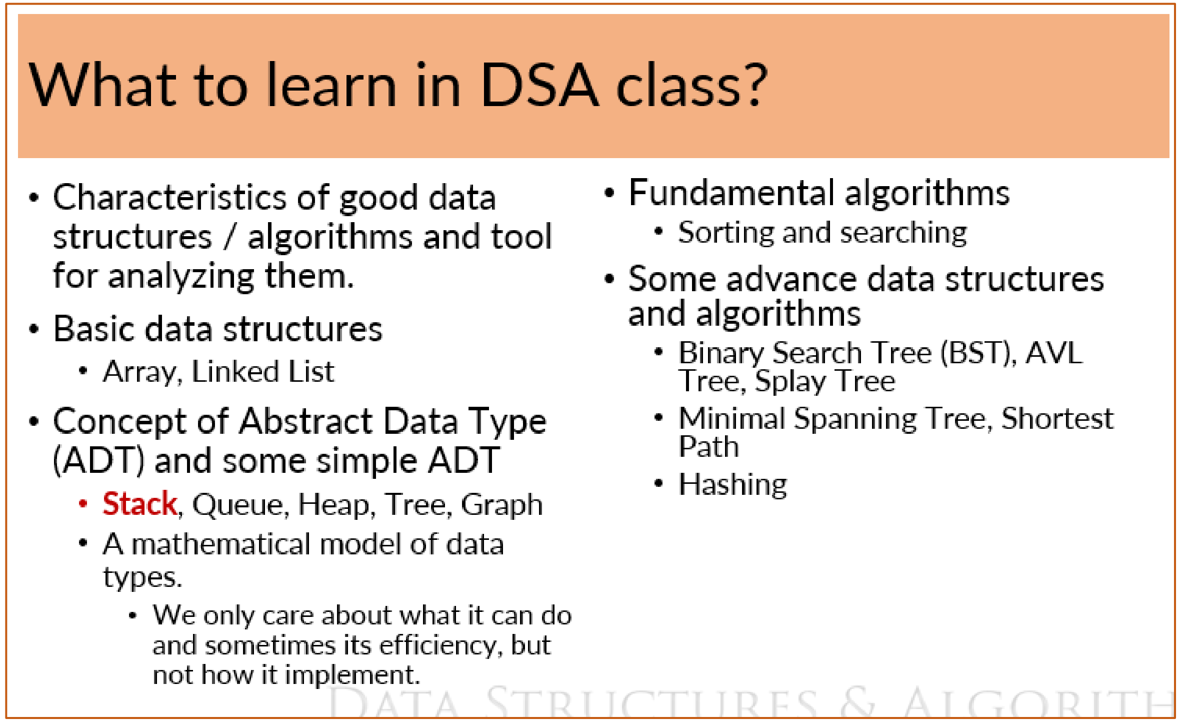
Stack

This is not the stack ADT!

(kmitl) cs-department

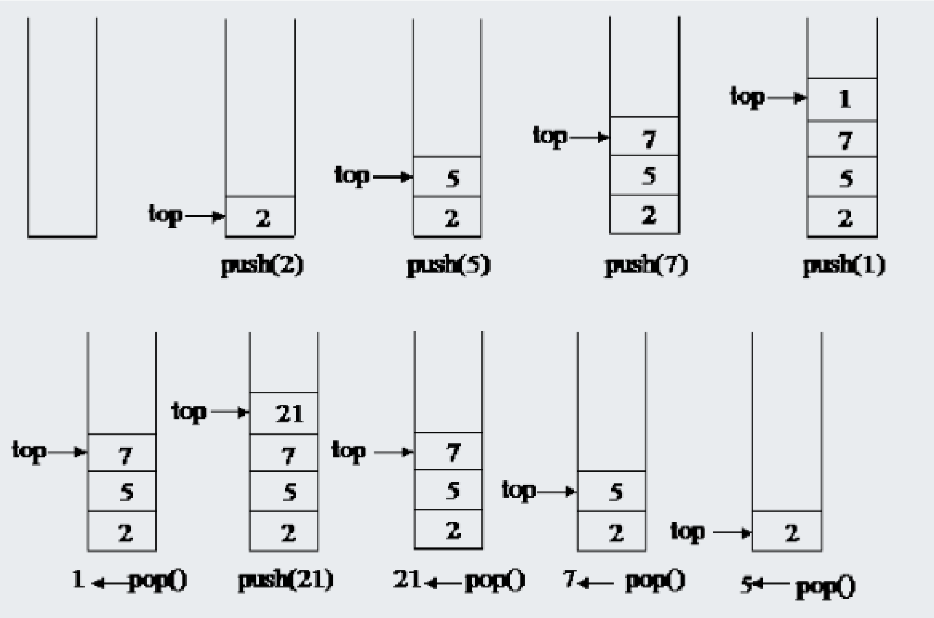
Outline

- Introduction
- Stack: Array Implementation
- Stack: Linked List Implementation
- Some interesting application. (Reverse Polish Notation - RPN)



Introduction : Stack

- A collection of data where you can add/remove a data to/from its top.
  - This make the last data in become the first data out.
  - Sometimes referred to as LIFO – last in, first out data structure
- There are two main operations in stack
  - Push – put a data on the stack
  - Pop – take a data out of stack
- With one convenient operation
  - Top – take a look at the data on the top of stack



https://www.collegenote.net/pastpapers/2898/question/

Note: **stack overflow** is an error occurred when trying to put a data to a full stack.

MyStack.java

MyStack.java

```
public class MyStack {
    public void push(int d) {
        // your code here
    }
    public int pop() {
        // your code here
    }
    public int top() {
        // your code here
    }
    public boolean isFull() {
        // your code here
    }
    public boolean isEmpty() {
        // your code here
    }
    public String toString() {
        // your code here
    }
}
```

Must be O(1)

StackTester.java

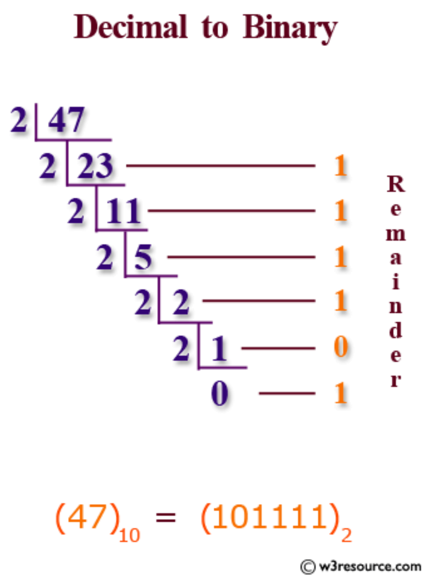
```
public class StackTester {
    public static void main(String[] args) {
        MyStack stack = new MyStack();

        // your code here

    }
}
```

Stack Usage: Decimal to Binary Conversion

- To convert decimal to binary
  we have to keep divide the decimal
  by two and record the remainders.
- Then, the result binary can be read from
  the remainders in reverse order.
- We can put the remainders in a list and reverse..
- Or we can use Stack!

```
public printBinary(int decimal) {
     Stack s = new Stack;
     while(decimal>0) {
          s.push(decimal%2);
          decimal/=2;
     }
     while(!s.isEmpty()) {
        System.out.print(s.pop());
     }
}
```

```
decimal=decimal/2;
=
decimal=decimal>>1;
=
decimal>>=1;
```



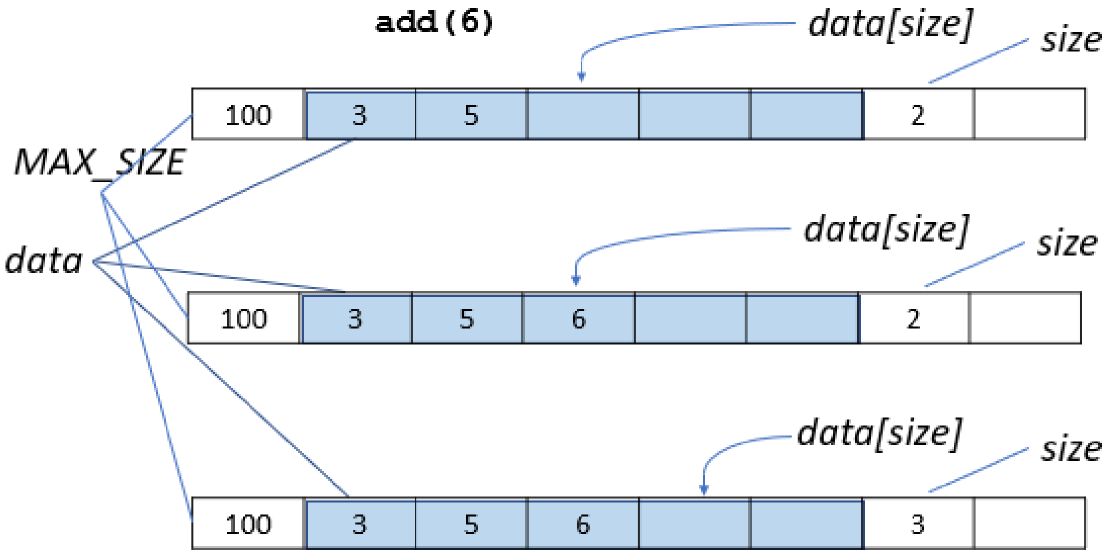
Stack: Array Implementation

- Required operations
```
void push(int d)
int pop()
```
- Convenient operation
```
int top()
```
- Utility operations
```
boolean isFull()
boolean isEmpty()
String toString()
```

MyStackA.java

```
public class MyStackA {
    int MAX_SIZE = 100;
    int stack[] = new int[MAX_SIZE];
    int top = 0;

    // your code here
}
```

- Recall Array Add
  - Pushing on a stack is like add to an array.
  - Popping from a stack is just taking out (delete) the last data and return.



push()/pop()

```
public void push(int d) {
    stack[top++] = d;
}
```

```
public int pop() {
    return stack[--top];
}
```

Again, must be O(1)!

top()

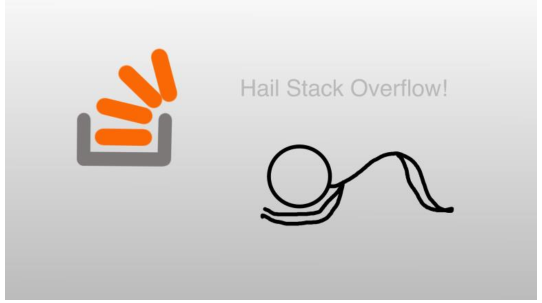
Method `top()` and variable `top` have the same name.
Valid, but not very good practice.

```
public int top() {
    return stack[top-1];
}
```

isFull()/isEmpty()

```
public boolean isFull() {
    return top==MAX_SIZE;
}
```

Stack overflow




```
public boolean isEmpty() {
    return top==0;
}
```

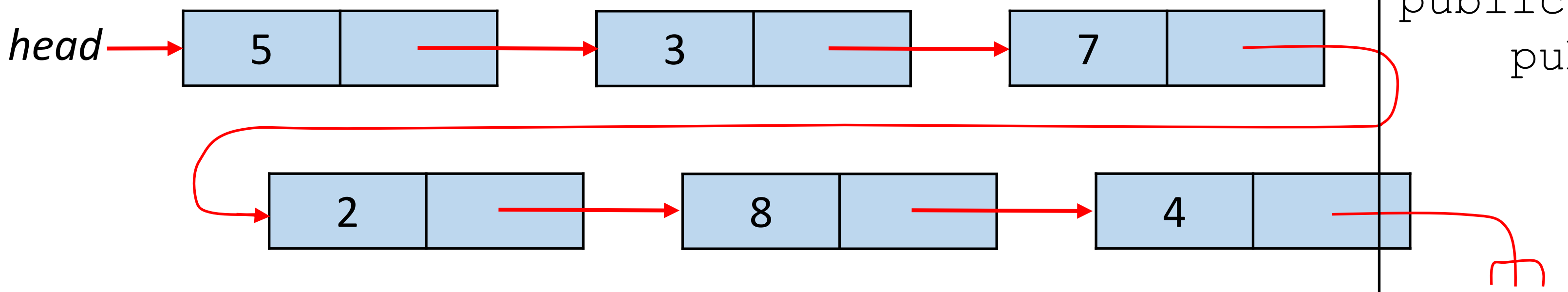
Stack underflow(?)

Recap

- Array implementation of stack is very simple.
  - Look like simpler version of MyArray.
- Next:
  - Linked List implementation.
  - Some interesting application.

Stack: Linked List Implementation

- Recall Linked List add()




```
public void add(int d) {
    Node p = new Node(d);
    p.next = head;
    head = p;
}
```

```
MyStackL.java
```

```
public class MyStackL {
    public class Node {
        int data;
        Node next;
        public Node(int d) {
            data = d;
        }
    }
    Node top=null;

    // your code here

}
```

- Now, how to remove it from the same end?

push()/pop()

```
public void push(int d) {
    Node p = new Node(d);
    p.next = top;
    top = p;
}
```

```
public int pop() {
    int d = top.data;
    top = top.next;
    return d;
}
```

Again, must be O(1)!

top()

```
public int top() {
    return top.data;
}
```

isFull()/isEmpty()

Stack overflow(?)

```
public boolean isFull() {
    return false;
}
```

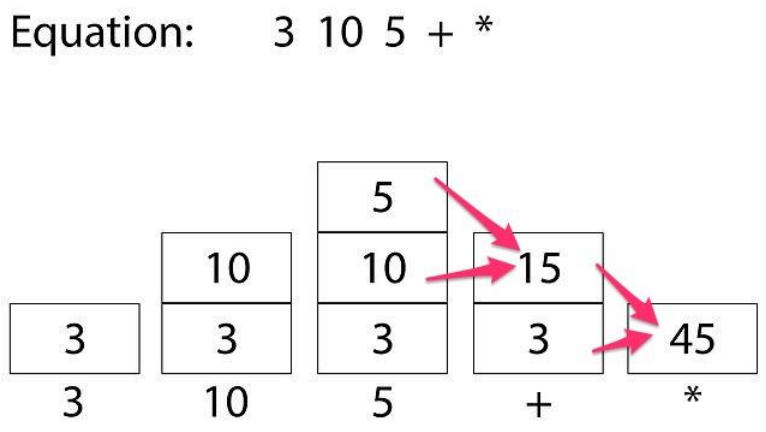
```
public boolean isEmpty() {
    return top==null;
}
```

Reverse Polish Notation - RPN

- In 1954, Arthur Burks, Don Warren, and Jesse Wright proposed Reverse Polish Notation (RPN).
  - Put number first to reduce memory and make use of stack
  - Ex: 3 * (10 + 5) becomes 3 10 5 + *
  - Aka. Polish postfix notation, postfix notation.
  - In 1924, Jan Łukasiewicz invented Polish Notation.
  - It is a way to write mathematical expression without parenthesis. (Prefix notation)
  - Ex: (3-1)*(4+5) can be rewritten as * - 3 1 + 4 5
- HP and other company adopted RPN and build applications and calculators to help engineer with **expression calculation**.

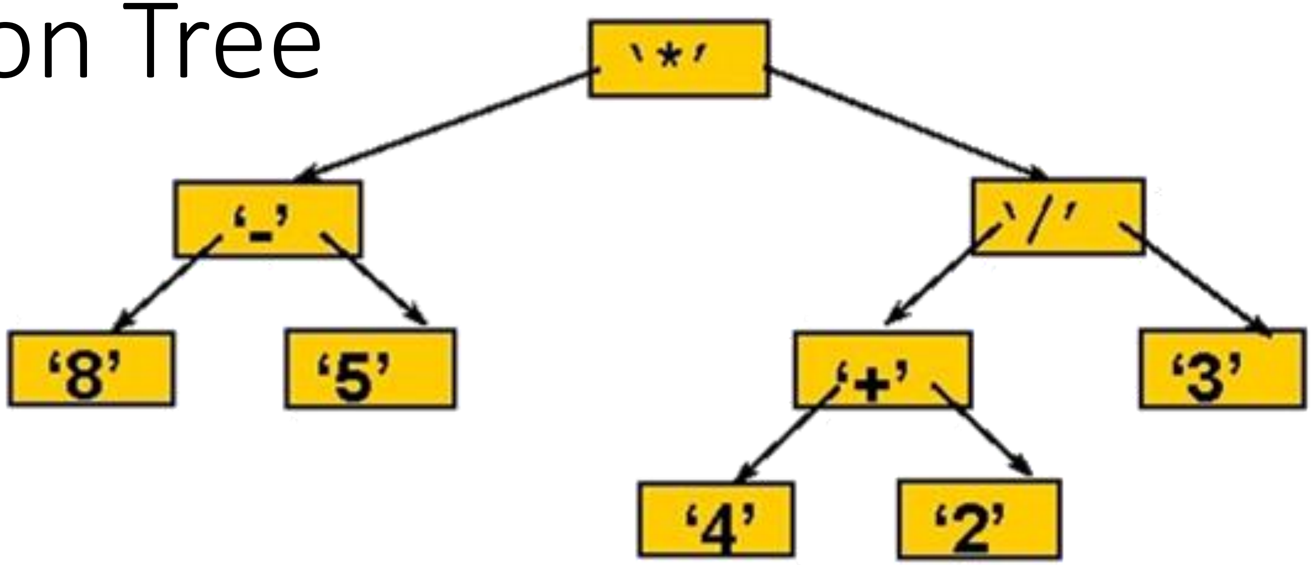
Note that both notation require every that operator takes fix number of operands in order to compute correctly

pain point



https://mrtan.me/post/19.html

Expression Tree



Infix: ((8 - 5) * ((4 + 2) / 3))

Prefix: * - 8 5 / + 4 2 3

Postfix: 8 5 - 4 2 + 3 / *

Only infix need parenthesis

Infix to Postfix Conversion: Visual Method

- Example Infix: **A + B * C**
- Step-by-Step Conversion:
  1. Fully Parenthesize the Expression. (Add parentheses to show precedence)
     (A + (B * C))
     - Notice that each operator is between its operands and enclosed in parentheses.
  2. Postfix Rule (Operators to the Right): Move the operator to the right of its operand pair, and remove parentheses.
     - (B * C) → B C *
     - (A + (B C *)) → A B C * +
- Final Postfix : **A B C * +**

How to Compute RPN

- Given a list of token t in a form of RPN.
  - Ex: 3 5 + the tokens are 3, 5, and + and they are in the form of RPN
- The following pseudocode can evaluate the result

```
Create a stack s
While there are more tokens
  t = next token
  case t is a number
    s.push(t)
  case t is an operator opr
    b = s.pop()
    a = s.pop()
    s.push(a opr b)
result = s.pop()
return result
```

An Example

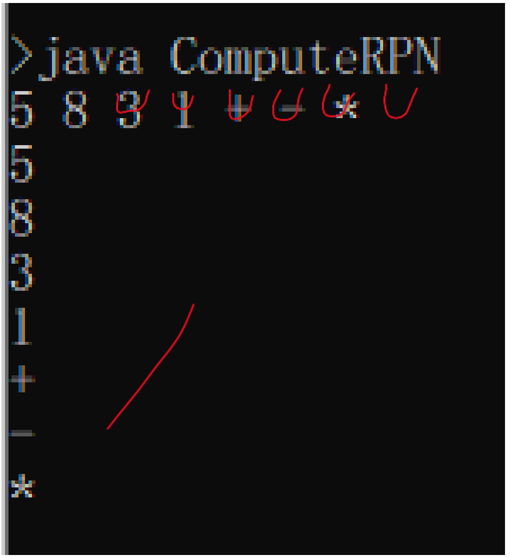
Given a list of tokens: 8 5 – 4 2 + 3 / *

1. t=8, s.push(8)
   - Stack: top -> 8 -> bottom
2. t=5, s.push(5)
   - Stack: top -> 5 -> 8 -> bottom
3. t= –, b=s.pop(), a=s.pop(), s.push(a-b)
   - Stack: top -> 3 -> bottom
4. t=4, s.push(4)
   - Stack: top -> 4 -> 3 -> bottom
5. t=2, s.push(2)
   - Stack: top -> 2 -> 4 -> 3 -> bottom
6. t=+, b=s.pop(), a=s.pop(), s.push(a+b)
   - Stack: top -> 6 -> 3 -> bottom
7. t=3, s.push(3)
   - Stack: top -> 3 -> 6 -> 3 -> bottom
8. t=/, b=s.pop(), a=s.pop(), s.push(a/b)
   - Stack: top -> 2 -> 3 -> bottom
9. t=*, b=s.pop(), a=s.pop(), s.push(a*b)
   - Stack: top -> 6 -> bottom
10. result s.pop(), return result
    - Stack: top -> bottom

Implementation: Testing StringTokenizer

ComputeRPN.java

```
import java.util.Scanner;
import java.util.StringTokenizer;

public class ComputeRPN {
    public static void main(String[] args) {
        MyStack stack = new MyStack();
        Scanner in = new Scanner(System.in);
        String rpn = in.nextLine();
        StringTokenizer st = new StringTokenizer(rpn);

        while(st.hasMoreTokens()) {
            String t = st.nextToken();
            System.out.println(t);
        }
        in.close();
    }
}
```



Implementation: Testing isNumeric

```java
import java.util.Scanner;
import java.util.StringTokenizer;
import java.util.regex.Pattern;

public class ComputeRPN {
private static Pattern pattern = Pattern.compile("-?\\d+(\\.\\d+)?");
    public static boolean isNumeric(String strNum) {
        if (strNum == null) {
            return false;
        }
        return pattern.matcher(strNum).matches();
    }
    public static void main(String[] args) {
        MyStack stack = new MyStack();
        Scanner in = new Scanner(System.in);
        String rpn = in.nextLine();
        StringTokenizer st = new StringTokenizer(rpn);

        while(st.hasMoreTokens()) {
            String t = st.nextToken();
            System.out.println(t+" is a number -> "+isNumeric(t));
        }
        in.close();
    }
}
```

```
>java ComputeRPN
5 8 3 1 + 3 - * /
5 is a number -> true
8 is a number -> true
3 is a number -> true
1 is a number -> true
+ is a number -> false
3 is a number -> true
- is a number -> false
* is a number -> false
/ is a number -> false
```

Implementation:

```
while(st.hasMoreTokens()) {
    String t = st.nextToken();
    if(isNumeric(t))
        stack.push(Double.parseDouble(t));
    else {
        if(t.equals("-")) {
            double b = stack.pop();
            double a = stack.pop();
            stack.push(a-b);
        } // else ...your code here...
    }
}
System.out.print("result: "+stack.pop());
```

```
>java ComputeRPN
3 1 -
result: 2.0
>java ComputeRPN
5 3 1 - -
result: 3.0
>java ComputeRPN
5 3 - 1 -
result: 1.0
>
```

Example challenge

1544. Make The String Great

Solved

Easy | Topics | Companies | Hint

Given a string `s` of lower and upper case English letters.

A good string is a string which doesn't have **two adjacent characters** `s[i]` and `s[i + 1]` where:

- `0 <= i <= s.length - 2`
- `s[i]` is a lower-case letter and `s[i + 1]` is the same letter but in upper-case or **vice-versa**.

To make the string good, you can choose **two adjacent** characters that make the string bad and remove them. You can keep doing this until the string becomes good.

Return *the string* after making it good. The answer is guaranteed to be unique under the given constraints.

Notice that an empty string is also good.

Example 1:

```
Input: s = "leEeetcode"
Output: "leetcode"
Explanation: In the first step, either you choose i = 1 or i = 2, both
will result "leEeetcode" to be reduced to "leetcode".
```

Example 2:

```
Input: s = "abBAcC"
Output: ""
Explanation: We have many possible scenarios, and all lead to the same
answer. For example:
"abBAcC" --> "aAcC" --> "cC" --> ""
"abBAcC" --> "abBA" --> "aA" --> ""
```

Example 3:

```
Input: s = "s"
Output: "s"
```

```java
public class Solution {
  public String makeGood(String str) {
    // beat 55.50%
    Stack<Character> stack = new Stack<>();
    char ch, tmp = '/'; // dummie
    for (int i = 0; i < str.length(); i++) {
        ch = str.charAt(i);
        if (stack.isEmpty()) {
            stack.push(ch);
            continue;
        }
        tmp = stack.peek();
        // lower-case vs upper-case
        if (Math.abs(tmp - ch) == 32) {
            stack.pop();
            continue;
        }
        stack.push(ch);
    }
    StringBuilder sb = new StringBuilder();
    while (!stack.isEmpty()) {
        sb.append(stack.pop());
    }
    return sb.reverse().toString();
  }
}
```

Summary

- Stack is a linear ADT in which the insertion and deletion operations are performed at only the end.
  - Top/Tail
- Its compulsory methods are .pop() and .push()
  - Very useful method is .top()
  - Utilities methods are .isFull(), .isEmpty(), .toString(), etc.
- Array or Linked List underlying data structure is demonstrated on implementing stack
- The LIFO characteristic of a stack can be used for solving many computer science problems.