

# 19CSE 212: Data structures and Algorithms

## Lab sheet 5

### STACK ADT

---

1. Declare a class (~~StackInt.java~~) for integer ~~StackInt~~ with two attributes:

(a) An array '~~arr~~' of size 5.

(b) A variable '~~top~~' initialized to -1;

2. Write a separate test driver class (~~Test.java~~) and create an object instance of ~~StackInt~~ class.

```
StackInt si = new StackInt();
```

Compile both the .java files and run ~~Test.java~~. Ensure no errors.

3. Now try to access the '~~top~~' attribute of ~~StackInt~~ from ~~Test.java~~ directly.

```
System.out.println("Stack top is " + si.top);
```

Compile and execute.

4. Add a ~~default constructor~~ that will create the stack of some standard top (say 10) in case user does not give the top.

```
StackInt() {  
    arr = new int[10];  
    top = -1;  
}
```

Test cases:

```
StackInt si = new StackInt();  
System.out.println(si.arr.length);
```

5. Add a ~~parameterized constructor~~ that will create the stack with specified size.

```
StackInt(int sz) {  
    arr = new int[sz]; top = -1;  
}
```

Test cases:

```
StackInt si2 = new StackInt(15);  
System.out.println(si2.arr.length);
```

4. Add ~~print~~ function to ~~StackInt.java~~ that will print the contents up to the top of the stack. It

~~should be public:~~

```
public void print() {  
    // Your logic here  
    // Scan through the array from 0 to top  
    print the values with tab space in between.  
}
```

~~5. Implement push() method first without checking the array length limit.~~

```
public void push(int item) {  
    // Your logic here  
    // increment top and set 'top'th position in 'arr' to item  
}
```

~~Test cases:~~

~~(a) Try invoking push operations and check.~~

```
StackInt si = new StackInt(),  
si.push(100),  
si.print(),  
si.push(200),  
si.print(),
```

~~(b) Try to push beyond stack capacity~~

```
StackInt si = new StackInt(),  
.....  
si.push(900), si.print(), si.push(300), si.print(),
```

~~The execution will abort as soon as the last push is invoked.~~

~~ArrayIndexOutOfBoundsException.~~

~~6. Implement the check to push() method. Don't add an item to the array if top exceeds arr.length. Instead print "can't push" message.~~

```
public void push(int item) {  
    // Your enhanced logic here  
}
```

~~Test cases:~~

~~(a) Run this time. No exception will be thrown this time around.~~

~~7. Add a getter method `getTop()` which returns the current top of the stack.~~

```
public int getTop() {  
    //return top;  
}
```

~~Test cases:~~

~~(a) Invoke `getTop()` from test file and print the top.~~

```
System.out.println(si2.getTop());
```

8. Now implement `pop()` method that removes the topmost item in the stack and returns it.

First without lower bound checking logic.

```
public int pop() {  
    // Your logic here  
}
```

Test cases:

(a) Write push and pop few items to check it's working.

```
int item = si.pop();  
si.print();
```

(b) Write pop more items than what were pushed.

```
int item1 = si.pop();  
si.print();  
....
```

The last call to pop should throw Array out of bounds exception.

9. Now implement the check for lower limit ( $< 0$ ) and print "can't pop" message.

~~Run test file. Note that no exception will be thrown this time.~~

~~10. Implement `peek()` method to return topmost item without removing it from the stack~~

```
public int peek() {  
    //Your logic here  
}
```

~~Test cases: (a) Write test file to check the working of peek. This also needs  $< 0$  check.~~

```
StackInt si = new StackInt[5];  
System.out.println(si.peek()); si.push(100);  
System.out.println(si.peek()); si.push(200);
```

```
System.out.println(si.peek()); si.push(300);
```

```
System.out.println(si.peek());
```

~~The first peek() call should print "can't peek" message. Other should return properly.~~

~~11. You can't check if contents of two stacks are same by using ==.~~

~~Test cases:~~

```
StackInt si1 = new StackInt();
```

```
StackInt si2 = new StackInt();
```

```
si1.push(100);
```

```
si2.push(100);
```

```
si1.push(200);
```

```
si2.push(200);
```

```
if (si1 == si2)
```

```
System.out.println("Both si1 and si2 are same");
```

```
else
```

```
System.out.println("Both si1 and si2 are not the same");
```

~~Run it and check. It will print si1 and si2 are not same. Why?~~

~~Because == operator will only compare 2 addresses. Then how to check the contents?~~

12. Implement `equals()` method which will first compare the top. If not same, return false. If same, scan through arrays of both stacks to check if each item in one stack is same as an item in another stack. If so, return true. Else return false.

```
public boolean equals(Stack another) { // Your logic here }
```

Test cases:

(a) Now run test file .

It will print si1 and si2 are same since top and contents are same.

(b) Now write test file to do same number of pushes but contents different.

```
StackInt si1 = new StackInt(5);
```

```
StackInt si2 = new StackInt(5);
```

```
si1.push(100);
```

```
si2.push(100);
```

```
si1.push(200);
```

```
si2.push(300);
```

if (code)

System.out.println("Both si1 and si2 are same");

else System.out.println("Both si1 and si2 are not the same");

~~13. Implement a function `getminElement()` to return the minimum element in a stack.~~

~~14. Implement a `copyStack()` function to return a duplicate stack of original stack.~~

~~15. Implement a function to `reverse` an input string using stack.~~

16. Given a stack with `push()`, `pop()`, `empty()` operations, delete the middle element in the stack.

Input : Stack[] = [1, 2, 3, 4, 5]

Output : Stack[] = [1, 2, 4, 5]

Input : Stack[] = [1, 2, 3, 4, 5, 6]

Output : Stack[] = [1, 2, 4, 5, 6]

17. Implement a function to sort the elements in a stack.

18. Implement Two Stacks in a Single Array