Lecture 6

Stack
A limited data structure and LIFO

It's safer to use the right tool for the job!

Last time:

Array or ArrayList:

- Random access data structure where each item can be accessed in constant time.
- But, it has its disadvantages such as shifting the items when there is a new insertion or a deletion because it is a contiguous structure (no holes allowed).

LinkedList:

- Sequential access data structure where each item can be assessed only in particular order.
- It is also a list made of nodes that contain a data item and a reference to the next node (in case of a singly linked list).
- There is no need to shift items to insert and delete in the middle of the list because we can simply link a node to insert or unlink a node to delete using references without moving all the rest of the items in the list. (Not a contiguous structure!)

So far, we have mainly talked about general-purpose list data structures and insertion, deletion, and searching on them.

Now, it is time to **talk about something very restricted**.

I am a web browser!

Suppose you searched for “data structures,” and found some interesting pages.

And click a link from that page and another link from that page again. Now, it is good enough but you want to go back to the initial page you clicked from the search result page.

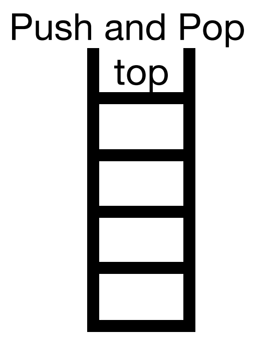
What would you do? Back button!!

How many times?

What if you are to develop this functionality?

There is a stack.

Step 1. Conceptual View
A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle.



Simply speaking, there are two major operations.

- Push: push an item onto the top of the stack.
- Pop: pop the item from the top of the stack.

Think of a stack of chairs in a classroom or Pringles!

Step 2: Implementation View

As it is mentioned in lecture 1, a stack is built on top of other data structures. It could be an array, an ArrayList, or a LinkedList. But, the most important thing is that we want a Stack to have the same functionalities no matter what underlying data structure it relies on.

For that, we need a Stack Interface.

```
public interface StackInterface<AnyType> {
    void push(AnyType item); // O(1)
    AnyType pop();           // O(1)
    AnyType peek();          // O(1)
    boolean isEmpty();       // O(1)
}
```

Array-based implementation

We need three fields for this implementation.

- An array **elements** which has a fixed capacity
- Variable **top** that refers to the top item in the stack
- Constant **DEFAULT_CAPACITY** that refers to the default array length

When do we know that stack is full? (This is based on fixed-length array!)

Also, how do we know that stack is empty?

Stack size: In general, stacks in programs do not grow too big. In fact, it is surprising how small a stack needs to be.

ArrayStack class

```
public class ArrayStack<AnyType> implements
StackInterface<AnyType> {

    private static final int DEFAULT_CAPACITY = 10;
    private int top;
    private _______________________ elements;

    public ArrayStack(int initialCapacity) {
        if (initialCapacity <= 0) {
            elements = new _________________[DEFAULT_CAPACITY];
        } else {
            elements = new _________________[initialCapacity];
        }

        // Set the top to be -1, indicating the stack is empty
        top = -1;
    }

    public ArrayStack() {
        this(DEFAULT_CAPACITY);
    }

    // implements all the methods here



}
```

Push

```
/**
 * Inserts a new item onto the top of the stack.
 * @throws StackException
 */
@Override
public void push(AnyType item) {
    // Check if stack is full or not
    if (top == ________________________) {
        throw new StackException("Stack is full");
    }
    elements[_______________] = item;
}
```

Pop

```
/**
 * Removes and returns the item at the top.
 */
@Override
public AnyType pop() {
    if (________________________) {
        throw new StackException("Empty");
    }
    // get the element on the top

    AnyType item = __________________  elements[top];
    elements[top] = _________________;
    top--; // reduce the top variable
    return item;
}
```

Peek

```
/**
 * Returns top item without removing it.
 */

@Override
public AnyType peek() {
    if (______________________) {
        throw new StackException("Empty");
    }
    return ________________________________________;
}
```

Revisit to the time complexity (Big O):

Push:

Pop:

Thinking back!

Now, what if you ***do not*** want to create your own Stack interface and a Stack class that implements the interface but still want to have LIFO type of functionality in your code?

What would you use?
Remember some of the major methods that LinkedList class in Java offers?

```
addFirst(Object) : void
removeFirst() : Object
```

```
LinkedList<Integer> theStack = new LinkedList<Integer>();
// push onto the stack
theStack.addFirst(0);
// pop from the stack
theStack.________________________________;
```

Do not be confused!: You are using addFirst and removeFirst methods and Stack is LIFO.
Spend some time to check out other methods in LinkedList that you can use such as push(E) and pop().

Example: Reverse a string!

```
public class Reverser {
    private String input;

    public Reverser(String str) {
        input = str;
    }

    public String doReverse() {
        ArrayStack<Character> theStack = new
ArrayStack<Character>(________________________________);

        // push all characters of given string onto the stack
        for (int i = 0; i < input.length(); i++) {
            theStack.________________________________________;
        }

        // Set output as String.
        String output = “”;
        while (_________________________________________) {
            output = output + theStack.___________________;
        }

        return output;
    }
}
```