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 element can be accessed directly in constant time.
- But, it has its disadvantages such as moving all of the elements when there is new insertion or deletion because it is contiguous structure (no holes allowed).

LinkedList:
- Sequential access data structure where each element 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 singly linked list).
- There is no need to shift elements 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 of the rest of elements in the list. (Not 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 was good enough but you want to go back to the initial page you clicked from the search result page.

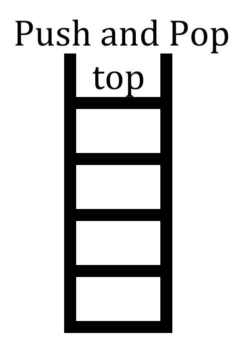
What would you do? Back button!!

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 only two operations allowed.

- Push : push the 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!

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, a LinkedList, etc. The most important thing is that we want a Stack to have the same functionalities no matter what underline data structure it has.

For that, we need a Stack Interface.

```
public interface StackInterface<AnyType> {
    void push(AnyType e); // 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 A which has fixed size of capacity
- Variable top that refers to the top element in the stack
- Variable DEFAULT_CAPACITY that refers to the default array length

When do we know that stack is full? (Remember this if fixed-size array!)

Also, how do we know that stack is empty?

Stack size : In general, stacks in real programs do not have big size. 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 = 15;
	private int top;
	private AnyType[] A;

	@SuppressWarnings("unchecked")
	public ArrayStack(int initialCapacity) {
		if(initialCapacity <= 0) {
			A = (AnyType[]) new Object[DEFAULT_CAPACITY];
		} else {
			A = (AnyType[]) new Object[initialCapacity];
		}

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

	public ArrayStack() {
		this(DEFAULT_CAPACITY);
	}

	 // implements all of the public methods here


}
```

Push

```
/**
 * Inserts a new element onto the top of the stack
 * @throws Exception
 */
@Override
public void push(AnyType e) {
     // Check if stack is full or not
     if(top == A.length-1) {
           throw new StackException("Stack has overflowed");
     }
     top++:
     A[top] = e;
}
```

Pop

```
/**
 * Removes and returns the element at the top
 */
@Override
public AnyType pop() {
     AnyType x = A[top]; // get the object on the top
     A[top] = ______________; // make sure the memory is freed
     top--; // reduce the top variable
     return x;
}
```

Peek

```
/**
 * Returns top item without removing it
 */
@Override
public AnyType peek() {
     if(___________________) throw new StackException("Empty");
     return A[top];
}
```

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?

Can you remember some of the major methods that LinkedList class in Java offers?

+addFirst(element: Object) : void
+removeFirst() : Object

```
LinkedList<Integer> theStack = new LinkedList<Integer>();

// push onto the stack
theStack.addFirst(0);

// pop from the stack
theStack.removeFirst();
```

Do not be confused!: You are using addFirst and removeFirst methods and Stack is LIFO.

Example : Reverse a string!

```
public class Reverser {
	private String input;

	public Reverser(String input) {
		this.input = input;
	}

	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++) {
			char ch = input.charAt(i);
			theStack.________________________;
		}

		// Set the output as a stringbuffer
		StringBuilder output = new StringBuilder();
		while(!theStack._____________________) {
			output.append(___________________________);
		}

		return output.toString();
	}
}
```