The Stack ADT

Computer Science 112
Boston University

Christine Papadakis-Kanaris

A Stack Interface: First Version

```
public interface Stack {
    boolean push(Object item);
    Object pop();
    Object peek();
    boolean isEmpty();
    boolean isFull();
}
```

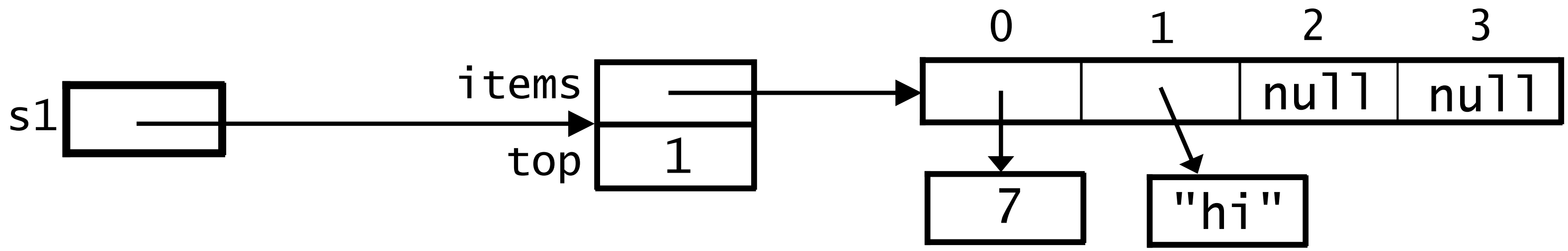
- `push()` returns `false` if the stack is full, and `true` otherwise.

- `pop()` and `peek()` take no arguments, because we know that we always access the item at the top of the stack.
  - return `null` if the stack is empty.

- The interface provides no way to access/insert/delete an item at an arbitrary position.
  - encapsulation allows us to ensure that our stacks are manipulated only in ways that are consistent with what it means to be stack

Collection Classes and Data Types

```
public class ArrayStack implements Stack {
    private Object[] items;
    private int top;    // index of the top item
     ...
}
```



- So far, our collections have allowed us to add objects of any type.

```
ArrayStack s1 = new ArrayStack(4);
s1.push(7);    // 7 is turned into an Integer object for 7
s1.push("hi");
String item = s1.pop();              // won't compile
String item = (String) s1.pop();  // need a type cast
```

- We'd like to be able to limit a given collection to one type.

```
ArrayStack<String> s2 = new ArrayStack<String>(10);
s2.push(7);                     // won't compile
s2.push("hello");
String item = s2.pop();    // no cast needed!
```

Limiting a Stack to Objects of a Given Type

- A *generic* interface and class.
- Here's a generic version of our `Stack` interface:

```
public interface Stack<T> {
    boolean push(T item);
    T pop();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

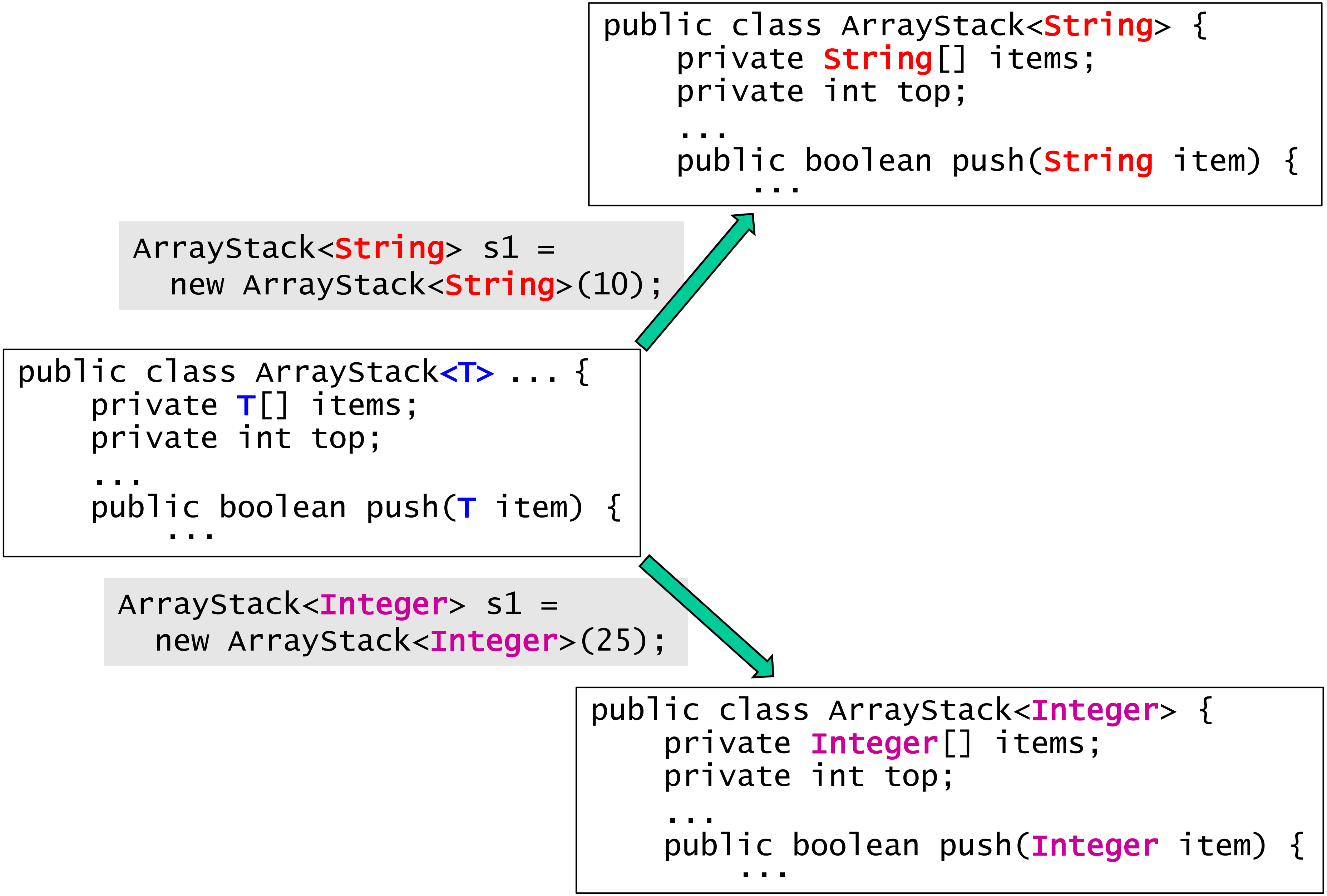
- It includes a *type variable* `T` in its header and body.
  - used as a placeholder for the actual type of the items

A Generic `ArrayStack` Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

- Once again, a type variable `T` is used as a placeholder for the actual type of the items.

Using a Generic Class



```
public class ArrayStack<String> {
    private String[] items;
    private int top;
    ...
    public boolean push(String item) {
        ...
```

```
ArrayStack<String> s1 =
  new ArrayStack<String>(10);
```

```
public class ArrayStack<T> ... {
    private T[] items;
    private int top;
    ...
    public boolean push(T item) {
        ...
```

```
ArrayStack<Integer> s1 =
  new ArrayStack<Integer>(25);
```

```
public class ArrayStack<Integer> {
    private Integer[] items;
    private int top;
    ...
    public boolean push(Integer item) {
        ...
```

`ArrayStack` Constructor

- Java doesn’t allow you to create an object or array using a type variable. Thus, we *cannot* do this:

```
public ArrayStack(int maxSize) {
    items = new T[maxSize];    // not allowed
    top = -1;
}
```

- To get around this limitation, we create an array of type `Object` and cast it to be an array of type `T`:

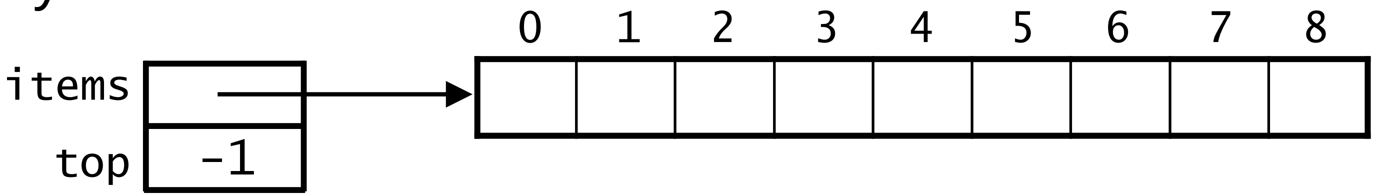
```
public ArrayStack(int maxSize) {
    items = (T[])new Object[maxSize];
    top = -1;
}
```

- The cast generates a compile-time warning, but we’ll ignore it.

- Java’s built-in `ArrayList` class takes this same approach.

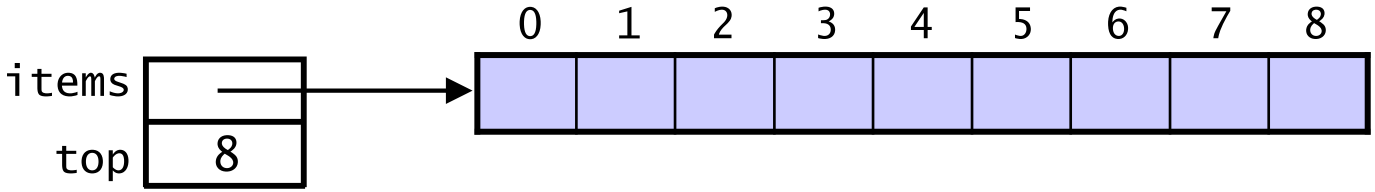
Testing if an `ArrayStack` is Empty or Full

- Empty stack:



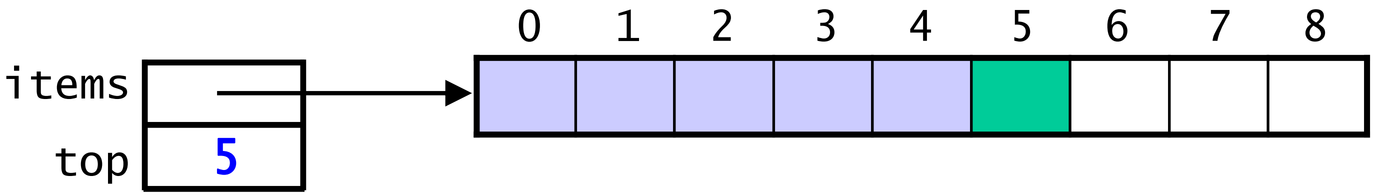
```
public boolean isEmpty() {
    return (top == -1);
}
```

- Full stack:



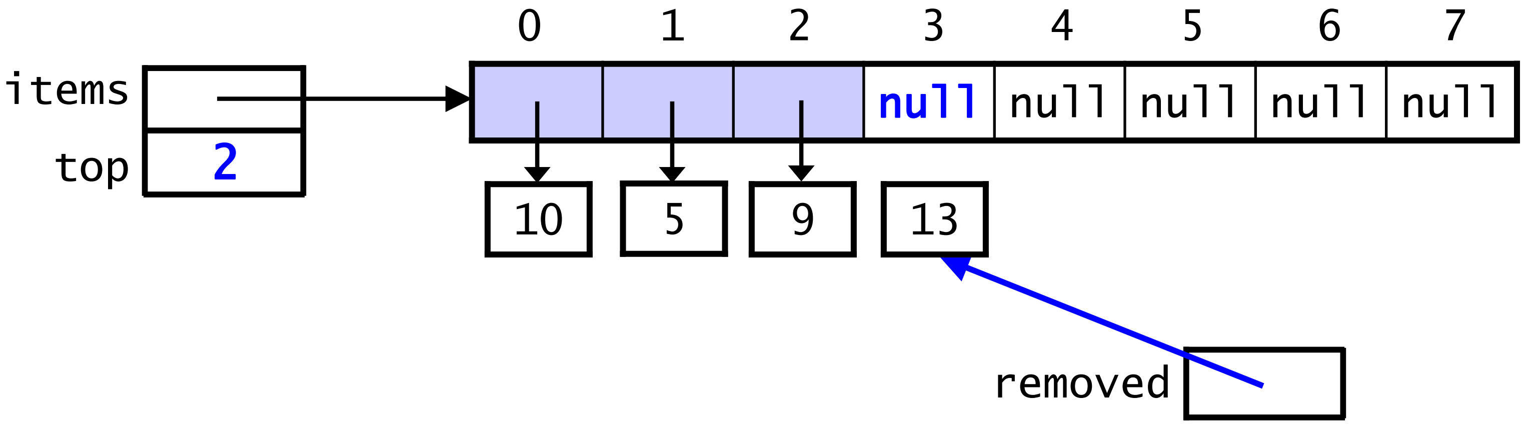
```
public boolean isFull() {
    return (top == items.length - 1);
}
```

Pushing an Item onto an ArrayStack



```
public boolean push(T item) {
    if (isFull()) {
        return false;
    }
    top++;
    items[top] = item;
    return true;
}
```

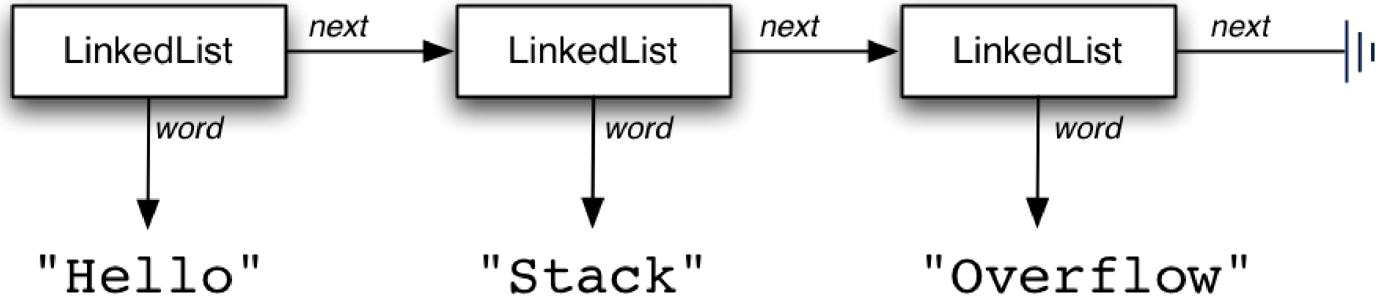
ArrayStack pop() and peek()



```
public T pop() {
    if (isEmpty()) {
        return null;
    }
    T removed = items[top];
    items[top] = null;
    top--;
    return removed;
}
```

- peek just returns `items[top]` without decrementing `top`.

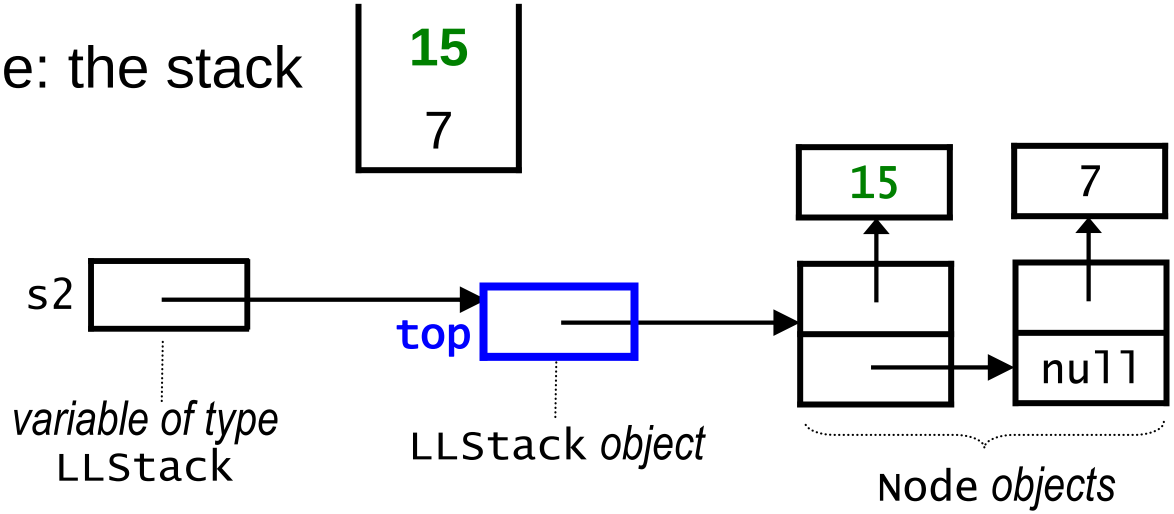
Linked List Implementation of a Stack



Implementing a Generic Stack Using a Linked List

```
public class LLStack<T> implements Stack<T> {
    private Node top;    // top of the stack
    …
}
```

- Example: the stack



- Things worth noting:
  - our LLStack class needs only a single instance variable—a reference to the first node, which holds the top item
  - top item = leftmost item (vs. rightmost item in ArrayStack)
  - we don’t need a dummy node
    - only one case: always insert/delete at the front of the list!

Other Details of Our `LLStack` Class

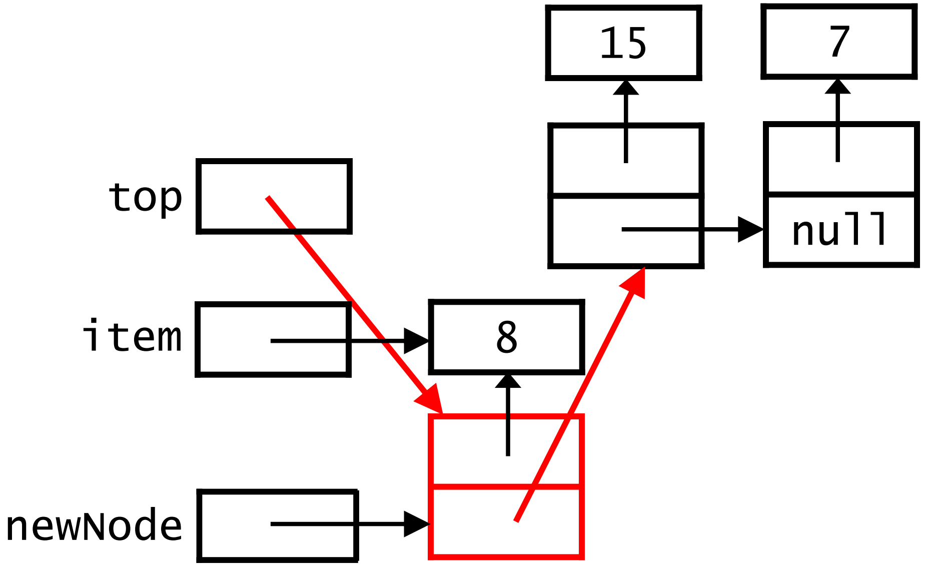
```
public class LLStack<T> implements Stack<T> {
    private class Node {
        private T item;
        private Node next;
        ...
    }

    private Node top;

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

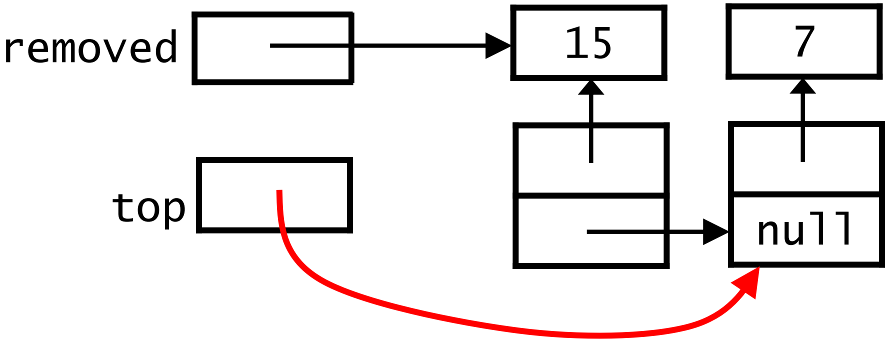
- The inner `Node` class uses the type parameter `T` for the item.
- We don’t need to preallocate any memory for the items.
- The stack is never full!

LLStack push()



```
public boolean push(T item) {
    Node newNode = new Node(item, top);
    top = newNode;
    return true;
}
```

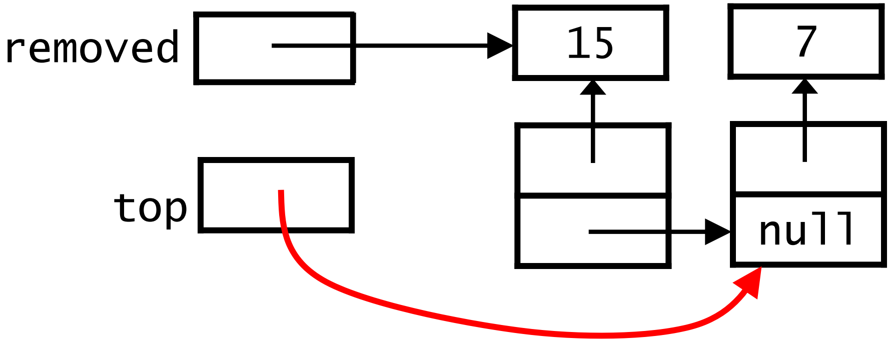
LLStack pop() and peek()



```
public T pop() {
    if (isEmpty()) {
        return null;
    }

    T removed = top.item;
    top = top.next;
    return removed;
}
```

LLStack pop() and peek()



```
public T pop() {
    if (isEmpty()) {
        return null;
    }

    T removed = top.item;
    top = top.next;
    return removed;
}
```

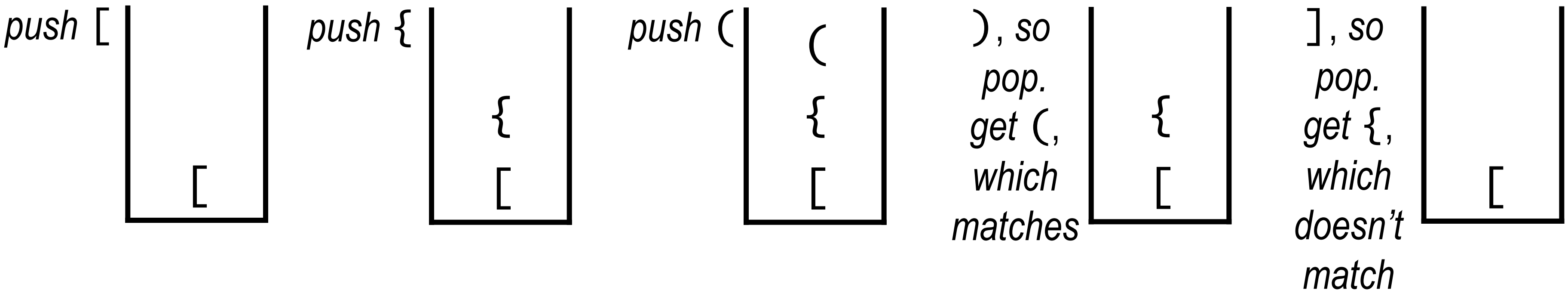
```
public T peek() {
    if (isEmpty()) {
        return null;
    }
    return top.item;
}
```

Efficiency of the Stack Implementations

	`ArrayStack`	**`LLStack`**
`push()`	*O*(1)	*O*(1)
`pop()`	*O*(1)	*O*(1)
`peek()`	*O*(1)	*O*(1)
space efficiency	*O*(m) where m is the *anticipated* maximum number of items	*O*(n) where n is the number of items currently on the stack

Applications of Stacks

- The runtime stack in memory
- Converting a recursive algorithm to an iterative one by using a stack to emulate the runtime stack
- Making sure that delimiters (parens, brackets, etc.) are balanced:
  - push open (i.e., left) delimiters onto a stack
  - when you encounter a close (i.e., right) delimiter, pop an item off the stack and see if it matches
  - example: `5 * [3 + {(5 + 16 - 2)]`



push [

push {

push (

), *so pop. get* (, *which matches*

], *so pop. get* {, *which doesn't match*

- Evaluating arithmetic expressions