Definition of a Stack

What is a stack?

A stack is a data structure referred to as a “collection” that operates on the principle of adding or removing data in a LIFO (last in; first out) manner. A collection is any data structure that binds together groups of objects of a given type together.

What makes a stack unique among other types of collections is that:

- You can only add to or remove from one point
- With a stack, where you add to or remove from are the same point

In a classic model, stacks are thought of like a stack of plates at a restaurant buffet:

The diagram above can demonstrate the nature of how we model a stack data structure.

When we create a new stack, the stack is empty. Therefore, it contains no data in it. We typical refer to a piece of data inside of a stack as an “item”.

When we add the first item to a stack, it appears at the bottom. Just like adding a plate to an empty portion of a tray that has no plates. When we add subsequent plates to the stack, we do not pick up the first plate and put the new stack under it. We simply place the new stack on top of the previous plate that was there. To use the classic model diagram as an example, let’s imagine a stack of integers where the first plate was the number 4. Let’s say we want to add a new item to the stack after that and it is the number 6. Our stack model would look like:

Once our model looks like this, we can either add more to the stack, remove something from the stack, or “peek” at the item at the top of the stack. If we add, say, a 7 to the top of the stack, it would be placed above the 6 on the model. If we removed an item from the stack instead, the 6 would be removed from the stack leaving the 4 as the only remaining item in the stack.

If we wanted instead to “peek” at the stack, this would return the value at the top of the stack (6 in this case) but it would not remove the item from the stack.

In a stack structure, there is no way to access arbitrary elements within the stack. You can only access the top item. You could, if you wanted, access lower items by removing higher items first but this is inefficient and largely defeats the purpose of the stack.

Purpose of a Stack

What purpose does a Stack serve? (vs. Array or others)

Like all data structures, stacks serve a particular purpose. We should never attempt to make a data structure fit a purpose it wasn't intended to. Much like tools in a toolbox, we want the right tool for each job.

Due to the nature of stacks, they are good for two things:

1. When we have a need for adding and removing often
2. When we need data to be ordered in a recursive fashion

One classic example of what Stacks are finely suited for are Undo/Redo operations in a program (like an Art or Word Processing program.)

Arrays are amazing data structures as you have learned in past courses. But arrays also have weaknesses; the largest of them is that add/remove operations are not possible in a fixed-sized array without creating a new array and copying over the contents from the previous array. This is computationally expensive. Therefore, if we have need to add and remove often, we should use a data structure made to accomplish that task specifically.

Stacks are not the only data structure that can handle add/remove operations easily however. The next question is HOW we want the data to be arranged. Stacks are recursive; they operate on a "what goes up must come down" sort of philosophy. Therefore, if you want your data to be arranged in this fashion, you would use a stack.

Contrast this with a Queue (which we will study next week) which is ordered like a line at DMV. You add to the back and remove from the front. A Queue is still a collection which is appropriate for when you need trivial add/remove but the order of the data also matters for matching well with what you are doing (the algorithm). In this way, you can say that data structures and algorithms are intimately connected.

We are currently discussing stacks as a general data structure but later in this course, we will discuss a very important stack called the "program stack." This is key to understanding how the stack portion of memory works and how the concept of recursion is possible. For now, just understand that a stack is a recursive data structure because as stacks add items and remove items, the removed items always returns us to previously added items like a boomerang (what goes up must come down).

Common Operations of a Stack

There are many different implementations of stack classes in different languages. A common set of behavioral elements (functions to support) are:

- Adding an item to the stack (push operation)
- Removing an item from the stack (pop operation)
- Looking at the item at the top of the stack without removing it (peek operation)
- Knowing whether a stack is empty or not (isEmpy operation)
- Knowing how many items are in the stack (size operation)
- Resetting the stack to empty (clear operation)

This is, in my opinion, the bare functionality that any stack class should support. The process of adding an item to a stack is called “pushing an item” on the stack. The origin of this term stems from assembly languages of old. When we remove an item from the stack, we call it “popping the item” from a stack.

The purpose of peek (as previously mentioned) is when we want to view the item at the top of the stack but not remove it from the stack. The purpose of “isEmpty” is to return a boolean value (true/false) informing us if the stack contains items or not. This can present a safe guard for pop and peek operations so that we are not trying to read from an empty stack. Without safe guards or bounds checking, this could result in throwing a “NullPointerException”.

The size operation can be useful for when we need to know how many items are in the stack for purposes of conversion to an array or for setting up a counter for a loop to iterate through the stack while popping the items from it.

The clear operation gives us a handy way to reset the stack to empty so we don’t have to create a new stack but can reuse the resource already created. This is especially useful in Java (which has automatic garbage collection) so we aren’t wasting cycles allocating and deallocating objects on the heap.

Creating and Using Stacks using the Java API

How would we use Stacks in Java?

The Java API includes Stack as part of it. Here is the key information:

```
import java.util.Stack; // This references the Stack class of the Java API

// Inside method:

Stack<String> st = new Stack<>(); // Declared and initializes empty string stack named “st”

st.push(“String #1”); // Add first string to stack

st.push(“String #2”); // Add second string to stack

String s = st.pop(); // Capture removed item into string “s”

String s2 = st.peek(); // Save string at top of stack into string “s2”

boolean b = st.isEmpty(); // Saves whether stack is empty (it isn’t currently so this is false)

int n = st.size(); // Save number of items in stack in “n”

st.clear(); // Resets stack to empty (size() == 0; isEmpty() == true)
```

We have no need to deallocate memory for the stack object when we are done with it since Java has automatic garbage collection.

Creating Stacks from Scratch in Java (Array Implementations)

How to create Stacks from scratch? (Array implementation; fixed vs dynamic size considerations)

When creating an array implementation of a stack, we need to ask a question. Do we want a fixed-size stack (fixed **maximum** size, that is) or do we want the stack to dynamically resize?

Fixed-size Implementation

```
package Main;

/* This is a fixed-size (max size) array implementation of a stack */

public class Fstack{

	// Fields

	private int[] a;

	private int cursor;

	// Constructor

	public Fstack(){

		a = new int[256];

		cursor = 0;

	}

	// Methods

	public void push(int item){

		if(cursor >= a.length) // Prevent buffer overrun

			return;

		a[cursor] = item;

		cursor++;

	}

	public int pop(){

		int ret = -1; // Set to error code

		if(cursor > 0){

			cursor--;

			ret = a[cursor];

		}

		return ret;

	}
```

```
	public int peek(){
		int ret = -1;
		if(cursor > 0){
			ret = a[cursor-1];
		}
		return ret;
	}
	public boolean isEmpty(){
		return cursor == 0;
	}
	public int size(){
		return cursor;
	}
	public void clear(){
		cursor = 0; // Be lazy and save cycles
	}
}
```

Dynamically sized Implementation

If we want a dynamically-sized version, we need to put out a bit more work. We need to add a private helper method called “resize” that will resize the array to double the previous size if we reached the end of the buffer.

This resize method needs to create the resized array, copy all of the old array data to the new array, and update the reference (pointer) of the class field for the array. We also need to modify the push method to check for this condition and resize, if needed.

Here is the Dstack version with everything the same EXCEPT the resize method and modified push:

```
package Main;

/* This is a dynamically-sized array implementation of a stack */

public class Dstack{
	// Fields
	private int[] a;
	private int cursor;
	// Constructor
	public Dstack(){
		a = new int[256]; // Start at a power of two
		cursor = 0;
	}
```

```
// Methods
private void resize(){
      int n = a.length * 2;
      int[] b = new int[n];
      for(int i = 0; i < a.length; i++)
            b[i] = a[i];
      a = b;
}
public void push(int item){
      if(cursor >= a.length) // Resize here...
            resize();
      a[cursor] = item;
      cursor++;
}
public int pop(){
      int ret = -1; // Set to error code
      if(cursor > 0){
            cursor--;
            ret = a[cursor];
      }
      return ret;
}
public int peek(){
      int ret = -1;
      if(cursor > 0){
            ret = a[cursor-1];
      }
      return ret;
}
public boolean isEmpty(){
      return cursor == 0;
}
public int size(){
      return cursor;
```

```
	}

	public void clear(){

		cursor = 0; // Be lazy and save cycles

	}

}
```

These classes are “safe” versions. Many implementations of stack classes will NOT be safe. This means that the burden of bounds checking for pop and peek operations will be on the user of the class and not on the class itself.

Create Stacks from Scratch in Java (Linked List Implementation)

How to create Stacks from scratch? (Linked List implementation)

A linked list is a collection that is bound together not by contiguous memory but by memory references or pointers. This arrangement is what gives it the strength over arrays giving us trivial add/remove but also gives us the weakness of not having constant time random access like an array.

Here is a version of a Stack implementation using a linked list:

```
package Main;

/* This is a linked-list implementation of a stack */

public class Lstack{

	// Fields

	private Node top; // Top of stack

	private int n; // How many items

	// Small subclass; I normally am not a fan of subclassing but this is a tiny implementation

	private class Node{

		int item; // Holds the discrete item in node

		Node next; // Reference to next node in list

	}

	// Constructor

	public Lstack(){

		top = null;

		n = 0;

	}

	// Methods

	public void push(int item){

		Node oldTop = top;

		top = new Node();

		top.item = item;

		top.next = oldTop;

		n++;

	}

	public int pop(){
```

```
            if(n == 0) return -1;
            int ret = top.item;
            top = top.next;
            n--;
            return ret;
      }
      public int peek(){
            if(n == 0) return -1;
            return top.item;
      }
      public boolean isEmpty(){
            return top == null;
      }
      public int size(){
            return n;
      }
      public void clear(){
            top = null;
            n = 0;
      }
}
```