Lucas Kitaev

Mr. Combs

Senior Projects

8 December 2017

Journal 2

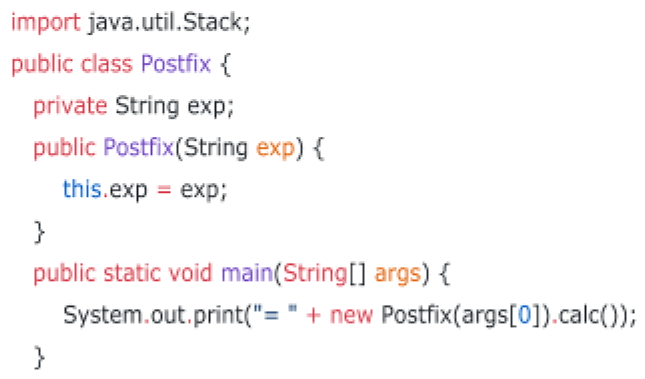
Goals

- Complete lesson on stacks
- Complete lesson on queues
- Complete lesson on linked lists
- Complete lesson on trees
- Complete lesson on heaps
- Configure NetBeans project. Research JavaServer Faces, Struts, Hibernate, and Spring
- Research Node Package Manager, jQuery, and AngularJS
- Outline the functional and infrastructural requirements for the application

Research

For this phase of the project's research, I focused on finishing the remaining curriculum I had left in the AP Computer Science AB material. I was able to skip the sections on basic algorithms, number systems, and abstract classes because I feel confident of my knowledge in those areas. I also skipped the lesson on graphs because it didn't have much content, and after skimming what was there, I didn't think it would be particularly useful. I also wanted to expand my knowledge of JavaScript, by learning about package managers, bundlers, libraries, and frameworks.

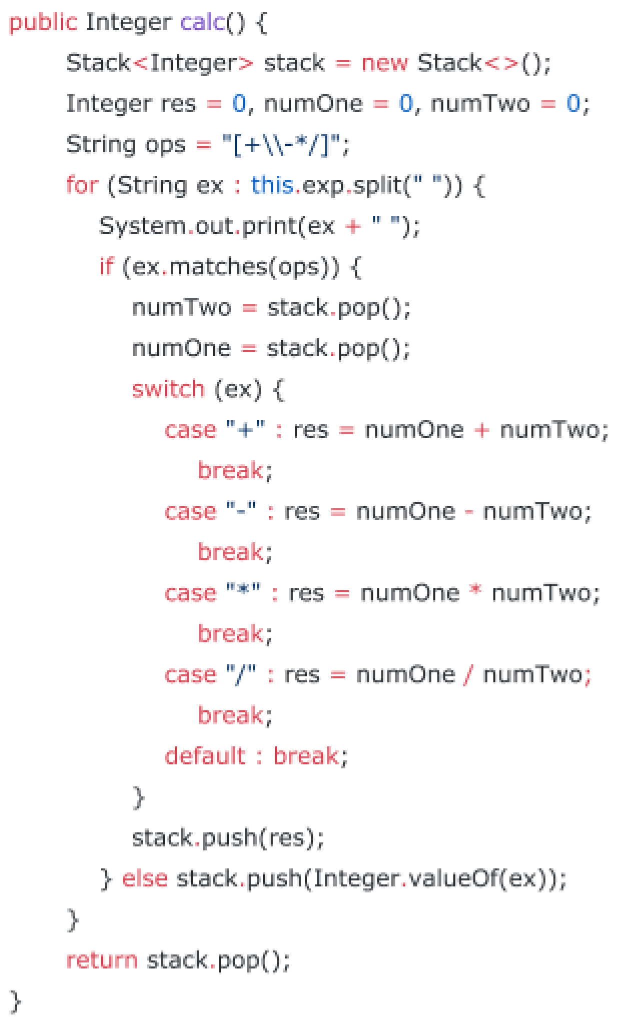
The first Java feature I researched was the stack. Stacks are data structures ordered by first in, last out (FILO). Objects are "pushed" onto the stack when they need to be stored, and "popped" off the stack when they need to be accessed. The stack can also be searched for a specific element.. Stacks can also be "peeked," which returns the object on the top of the stack without removing it from the stack. I made a short program that serves as an example of usage of stacks, which I've included here:



```java
import java.util.Stack;
public class Postfix {
  private String exp;
  public Postfix(String exp) {
     this.exp = exp;
  }
  public static void main(String[] args) {
     System.out.print("= " + new Postfix(args[0]).calc());
  }
```

This is a standard setup for a Java application, including an instance variable, constructor, and main method. Postfix is an alternate method of writing mathematical expressions, where the operator comes after the operands. In terms of computation performance, postfix is much more efficient than infix (the standard way expressions are written, with operators separated by operands) because there are no parentheses required, and therefore no need to worry about order of operations. The user inputs a presumably valid postfix expression as the program argument (given before the code is executed). It includes a shorthand version of creating a new instance of the class and accessing its calculation method. The keyword `this` refers to the current instance of the object.

This is the function called in the main method. It creates a stack object containing integers, and integers for the current result and operands. A regular expression is used to define the operators; it specifies a group of the four main operators, with the minus sign escaped by two backslashes, since it would otherwise specify a range of characters. A loop over each part of the input string prints out the value of it and then checks if there is a match for one of the operators. If so, it pops the top two numbers off the stack, and performs the specified calculation using a switch statement, which is then pushed onto the stack. If there is no match for the operators, the program assumes it's a number and pushes it onto the stack. The method returns the final value on the stack., as that would be the result after completing all of the calculations in the expression.



```
public Integer calc() {
    Stack<Integer> stack = new Stack<>();
    Integer res = 0, numOne = 0, numTwo = 0;
    String ops = "[+\\-*/]";
    for (String ex : this.exp.split(" ")) {
        System.out.print(ex + " ");
        if (ex.matches(ops)) {
            numTwo = stack.pop();
            numOne = stack.pop();
            switch (ex) {
                case "+" : res = numOne + numTwo;
                    break;
                case "-" : res = numOne - numTwo;
                    break;
                case "*" : res = numOne * numTwo;
                    break;
                case "/" : res = numOne / numTwo;
                    break;
                default : break;
            }
            stack.push(res);
        } else stack.push(Integer.valueOf(ex));
    }
    return stack.pop();
}
```

The next data structure I looked at is the queue. Queues work similar to stacks, but in a first in, first out (FIFO) order. Elements can be added to the back of the queue or removed from the front queue, and just like with stacks, elements can be peeked at. The front of the queue is known as the head of the queue, and any element in the stack that is not the first or last element cannot be operated on. In Java, a queue is an interface, meaning it does not have any inherent functionality. A common implementation of the queue interface is the priority queue, which uses natural ordering (essentially alphabetical) to arrange its elements. Another common implementation of the queue interface is the linked list. The main feature that differentiates linked lists from array lists is that all linked list elements contain a reference to its neighboring elements, which can be used to verify if an element is in its proper position in the list. Linked lists also contain the same functionalities that stacks and queues have.

The last two data structures I learned about were trees and heaps, which actually have no standard implementation in the Java API (application programming interface). Trees and heaps essentially work the same way, consisting of nodes with leaves. The top node is called the root, and each node below the root has a parent node; each parent node has references to its left and right children (Eck). There are three main descriptors for tree structures: width, height, and fullness. Width is the distance (number of nodes) between the two furthest leaves in the tree; height is the longest path from the root to a leaf; a full tree is one where all parents have either two or no children. Trees can be traversed in four ways: in-order, rev-order, pre-order, and post-order. In-order starts at the bottom left node and goes to its parent, then its right child, and then to the parent of its parent until it gets to the root, and then goes back down in the opposite order; rev-order is simply the reverse of in-order. Pre-order starts from the root and then to each left child, then back up and to each right child, ending when it gets to the bottom right node. Post-order starts from the bottom left node like in-order, but goes directly to the right child of its parent, and then up to the parent, ending when it reaches the tree (Eck). Heaps follow the same rules as trees, but are ordered with either the maximum element as the root (max-heap), or the minimum element as the root (min-heap). The way heaps are ordered makes them very similar to priority queues.

The second part of my research for this journal focused on improving my knowledge of JavaScript. I learned how to use Node Package Manager (npm), which has become the standard package manager for web developers, having replaced Bower (Jang). Package managers help developers organize all of the dependencies (other projects that their project is dependent on to function), by installing and updating the needed packages. npm uses a simple text file to describe all of the packages the project uses, this means that developers don't need to include dependencies in their code repository, as other people can just use the text file to install all of the dependencies themself. Two important npm packages I'm using are webpack and babel. webpack is used to bundle up all of the project dependencies into a single .js file, which not only improves web page load times, but greatly simplifies development workflow. babel is a transpiling program, meaning that it converts code in one format to another format. I'm mainly using babel to convert my JavaScript with ECMAScript 2015 features (ECMAScript is the standard for web scripting languages) to more compatible ECMAScript 5, otherwise website functionality would be broken for old browsers like Internet Explorer (Jang).

Accomplishments

- Understanding of advanced data structures in Java
- Configured NetBeans project with npm package in GitHub repository

Reflection

While I didn’t cover the entire computer science curriculum, I feel like I’ve learned enough to put to use the knowledge. I didn’t fully complete two of my goals (researching Java and JavaScript frameworks and libraries, and completing the program functionality outline), so that means I’m still two days behind on my timeline. This delay means that I most likely won’t be able to start development on the web application until winter break. I’ll need to spend the week before finals doing my job-shadow write-up and program functionality outline. During finals week, I should be able to complete the remaining research. I’ve also done some searching around to find other recommendation engines, which has gotten me a head start on the second phase of my research. My advisor also helped me set up an online SQL Server database, but I haven’t experimented with it much besides adding a connection to it in NetBeans; I’ll need to start doing lessons on database programming as a part of my second research phase. So far, I’m happy with how my project is coming along, although I haven’t gotten to the hard part yet.

Works Cited

Eck, David J. “Binary Trees.” *Javanotes 7.0, Section 9.4*, HOBART AND WILLIAM SMITH COLLEGES, 10 Dec. 2016, math.hws.edu/javanotes/c9/s4.html.

Jang, Peter. “Modern JavaScript Explained For Dinosaurs – Peter Jang – Medium.” *Medium*, Medium, 18 Oct. 2017, medium.com/@peterxjang/modern-javascript-explained-for-dinosaurs-f695e9747b70.