University of Calgary

Assignment 2 - Application of Lists, Stacks, and Queues

Anmol Ratol

CPSC 331

Dr. Ahmad Nasri

May 31, 2024

Exercise 1

See the attached Java files.

Exercise 2

```java
    // remainder is code copied from the template
    public static void main ( String [ ] args ) {
        Random random = new Random ( ) ;
        int [ ] randomNumbers = new int [ 10 ] ; // Array t o s t o r e 10 random numbers
        // Generate random numbers and s t o r e them i n t h e a r r a y
        for ( int i = 0 ; i < randomNumbers.length ; i ++) {
            randomNumbers [ i ] = random.nextInt( 100 ) + 1 ;
        }
        // P r i n t t h e g e n e r a t e d random numbers
        System.out.println ( "Unsorted Numbers numbers:" ) ;
        for ( int number : randomNumbers ) {
            System.out.println ( number ) ;
        }
        // C a l l t h e f u n c t i o n s t o e x e c u t e t h e s t e p s
        Unsorted_Stack ( randomNumbers ) ;
        System.out.println ( "Unsorted numbers in Stack:" ) ;
        printArray ( randomNumbers ) ;

        Stack_To_Queue ( randomNumbers ) ;
        System.out.println ( "Unsorted numbers in Queue:" ) ;
        printArray ( randomNumbers ) ;

        Sorting_Queue ( randomNumbers ) ;
        System.out.println ( "Sorted numbers in Queue :" ) ;
        printArray ( randomNumbers ) ;

        Sorted_Stack ( randomNumbers ) ;
        System.out.println( "Sorted numbers in Stack :" ) ;
        printArray ( randomNumbers ) ;
    }
```

As per the second rule of time complexity as discussed in class, the time complexity taken to sort the values will be the sum of the time complexities of all the functions it calls:

(1)

$$O(Exercise1) = O(GeneratingRandomNumbers) + O(UnsortedStack) + O(StackToQueue) + O(SortingQueue) + O(SortedStack) + 4O(printArray)$$

Firstly the time complexity of generating a new array is *O(n)* time complexity. The cost of then populating arrays with random numbers ends up being *O(n)* by the second rule (multiplication) as it is a loop that iterates *n* times and that `random.nextInt()` has time complexity *O(1)*. Likewise the following printing loop is also *O(n)* as `System.out.println()` is *O(n)*, but since all of the strings being printed are length 2 or less and it doesn't depend on the size of the array, it is essentially *O(1)*.

```java
    // code to copy the contents of one array to another
    private static void clone(int[] array, int[] array2) {
        for (int i = 0; i < array.length; i++){
            array[i] = array2[i];
        }
    }

    // moving the values of the list into a stack, with the bottom being the end of t
    public static void Unsorted_Stack(int[] array) {

        stack = new int[array.length];

        for (int i = 0; i < array.length; i++){
            stack[array.length-i-1] = array[i];
        }

        clone(array, stack);
    }
```

The `clone` function to make two arrays identical has time complexity of *O(n)* as assignment is an *O(1)* function and the loop iterates *n* times. This function is periodically throughout the remaining functions. For `Unsorted_Stack`, the time complexity of initializing the stack is *O(n)*. The following for loop is also *O(n)*, as is the `clone` function. The time complexity of the whole `Unsorted_Stack` ends up being roughly *3*n*, and therefore by the first rule the time complexity is *O(n)*.

```java
    // simulating popping the values into the queue, with the back being the front
    public static void Stack_To_Queue(int[] array) {

        queue = new int[array.length];

        for (int i = 0; i < array.length; i++){
            queue[array.length-i-1] = array[i];
        }

        clone(array, queue);

    }
```

Given that this is essentially identical to the above function, it is also of *O(n)* time complexity.

```java
    // sorting the queue from smallest to largest, with the larges
    public static void Sorting_Queue(int[] array) {

        // standard insertion sort code
        for (int i = 0; i < array.length; i++){
            int j = i;
            while ((j > 0) && (array[j] > array[j - 1])){
                int tmp = array[j];
                array[j] = array[j-1];
                array[j-1] = tmp;
                j--;
            }
        }


    }
```

The average case time complexity of the insertion sort is *O(n^2)*, as its best case implementation is *O(n)* while the worst case ends up *O(n^2)*. As such, `Sorting_Queue` also has *O(n^2)* time complexity.

```java
    // dequeuing the values into the stack
    public static void Sorted_Stack(int[] array) {

        for (int i = 0; i < array.length; i++){
            stack[array.length-i-1] = array[i];
        }

        clone(array, stack);

    }
```

`Sorted_Stack` is essentially the same code as `Unsorted_Stack`, but without the initialization code and as such still ends up being roughly *2*n* in time complexity, so it is *O(n)* overall.

```java
    // U t i l i t y method t o p r i n t an a r r a y
    public static void printArray ( int [ ] array ) {
        for ( int num : array ) {
        System.out.print (num + " " ) ;
        }
        System.out.println( ) ;
    }
```

`printArray` ends up being of the same time complexity as the initial print statements (*O(n)*) when generating the random numbers, as we are printing strings of length 3 or less (as one character is taken up by the blank string) and we are printing *n* numbers and printing an empty string once.

Finally, the total time complexity of these functions can be calculated by substituting the individual time complexities into the original equation (1):

$$O(Exercise1) = O(n)+O(n)+O(n^2)+O(n)+O(n)+4O(n) = O(8n+n^2)$$

Then, intuitively:

$$8n + n^2 \leq cn^2 = 9n^2, \forall n \geq 1$$

Thus, by asymptotic analysis where $c = 9$ and $n_0 = 1$, *Exercise1* is of *O(n^2)* time complexity.

Exercise 3

See the attached Java files.

Exercise 4

```java
    // copied from template
    public static void main ( String [] args ) throws Exception {
        String expression1 = "2+3*1" ;
        String expression2 = "3*2^4-7" ;
        String postfix1 = convertToPostfix ( expression1 ) ;
        String postfix2 = convertToPostfix ( expression2 ) ;
        System.out.println ( expression1 + "->Postfix: " + postfix1 + ", Evaluation: "  + evaluatePostfix ( postfix1 ) ) ;
        System.out.println ( expression2 + "->Postfix: " + postfix2 + ", Evaluation: "  + evaluatePostfix ( postfix2 ) ) ;
    }


}
```

As mentioned in the proof for the time complexity of *Exercise1*, the total time complexity will be the sum of the time complexities of the individual functions called:

(1)

$$O(Exercise3) = O(convertToPostfix) + O(evaluatePostfix)$$

```java
public class Modified_ShuntingYard_Evaluation {
	static String acceptables = "0123456789^*/+-";

	// converting to postfix
	public static String convertToPostfix(String expression) throws Exception{

		// error checking
		if (expression.length() == 0){
			throw new Exception("The input string is empty!");
		}

		// splitting the string into an array
		String[] expArray = expression.split("");
		Stack<String> opStack = new Stack<String>();
		String outputString = "";

		for (String elem: expArray){

			// basic error checking
			if (!acceptables.contains(elem)){
				throw new Exception(String.format("The input string contains invalid character <%s>!", elem));
			}

			// if the string elem is an operand, add it to the outputString
			if (isOperand(elem)) {
				outputString += elem;
			}

			// otherwise, place the operator in the stack after moving all other scanned operators
			// of higher precedence into the outputString, if there are any
			else {
				while((!opStack.isEmpty()) && (hasHigherPrecedence(opStack.peek(), elem))){
					outputString += opStack.pop();
				}

				opStack.push(elem);

			}

		}

		// shuffle the remaining operators into the output string and return
		while (!opStack.isEmpty()){
			outputString += opStack.pop();
		}

		return outputString;
	}
```

```java
    // check the array to see if the string matches anything in the operators list, if n
    private static boolean isOperand(String token){

        for (String operator: operators){
            if (token.compareTo(operator) == 0){
                return false;
            }
        }

        return true;
    }
```

```java
    // check whether the first operand has higher precedence than the second
    private static boolean hasHigherPrecedence(String op1, String op2){
        return (getPrecedence(op1) >= getPrecedence(op2));
    }
```

```java
    // assigning precedence according to PEMDAS or BEDMAS rules
    private static int getPrecedence(String operator)   {

        if (operator.compareTo("^") == 0){
            return 3;
        }

        else if ((operator.compareTo("*") == 0) || (operator.compareTo("/") == 0)){
            return 2;
        }

        else {
            return 1;
        }

    }
```

Starting with the time complexity of `convertToPostfix`, the first set of statements prior to the for

loop are of *O(1)* time complexity, as the if-else statement, stack initialization, and `outputString` initialization are all constant and independent of the size of the input string. The sole exception is the `expression.split("")` operation, as this is *O(n)* time complexity as it must search for each instance of the delimiter and split the string wherever it finds it. `isOperand` ends up being of *O(1)* time complexity, since it is the equivalent of many if-else statements. `getPrecedence` is similarly of *O(1)* time complexity, and consequently `hasHigherPrecedence` is also *O(1)* since it just executes it twice and then compares the values. Moving ahead to the for loop within `convertToPostfix`, it begins with an error check to make sure the character is valid. Since only one character is checked at a time and the length of acceptables is constant and not dependent on the input expression, this operation can be considered to be *O(1)* time complexity. In the string expression, there will always be one more operand than there are operators in the string. The code for handling operands is of *O(1)* time complexity, since it is just being added to the end of the `outputString`.

This occurs *ceiling((n - 1) / 2)* times. The code for handling operators is more complex, and there are in total *floor((n - 1) / 2)* operators to handle. From the implementation a few things can be surmised: firstly each operator will at some point be pushed onto `opStack` at most one time, and that it will be queued on to `outputString` once. The former is true as each operator is pushed onto the stack, and will be popped either when a lower order operator is detected, or in the final mass popping while loop. So the total number of operations performed on the operators end up being *3*floor((n - 1) / 2)* operations total, once each for pushing and popping from the `opStack`, and once each for adding to `outputString`. Therefore, the time complexity for `convertToPostfix` ends up being (disregarding the *O(1)* operations in the first portion by the fourth rule):

$$O(convertToPostfix) = O(expression.split(\text{""})) + O(handlingOperands) + O(handlingOperators)$$

$$O(convertToPostfix) = O(n + ceiling(\frac{n-1}{2}) + 3 * floor(\frac{n-1}{2})) \leq O(3n)$$

So therefore, by the first rule, `convertToPostfix` is *O(n)* time complexity.

```java
    // actually evaluating the expression
    public static double evaluatePostfix(String expression){

        // split the input string into an array
        String[] expArray = expression.split("");
        Stack<String> evalStack = new Stack<String>();
        double op1 = 0.0;
        double op2 = 0.0;
        double expValue = 0.0;

        // cycle through the arrray
        for (String elem: expArray){

            // if it is an operand, then push it directly into the stack
            if (isOperand(elem)){
                evalStack.push(elem);
            }

            // otherwise, take the operator and the previous two elements from the st
            else {
                op2 = Double.parseDouble(evalStack.pop());
                op1 = Double.parseDouble(evalStack.pop());

                evalStack.push(String.valueOf(performOperation(op1, op2, elem)));
            }
        }

        // return the final value resting on top of the stack
        return Double.parseDouble(evalStack.pop());

    }
```

```java
    // converting the inputs to the double value based on the operator
    private static double performOperation(double operand1, double operand2, String operator){

        if (operator.compareTo("^") == 0){
            return Math.pow(operand1, operand2);
        }

        else if (operator.compareTo("*") == 0) {
            return operand1*operand2;
        }

        else if (operator.compareTo("/") == 0){
            return operand1/operand2;
        }

        else if (operator.compareTo("+") == 0){
            return operand1+operand2;
        }

        else {
            return operand1-operand2;
        }
    }
```

When it comes to `evaluatePostfix`, once again all of the statements prior to the for loop are *O(1)* time complexity, with the exception of `expression.split("")` which as previously mentioned is *O(n)* time complexity. The for loop then iterates *n* times and all of the statements within are of *O(1)* time complexity, since `performOperation` is just a series of if-else statements and all operations within are also *O(1)* statements. The loop performs three *O(1)* statements for each operator within the string, and one *O(1)* operation for each operand, so in total there are *ceiling((n – 1) / 2) + 3*floor((n - 1) / 2)* operations, which is roughly equal to *2n*. `parseDouble` is in general an *O(n)* time complexity, but since we are only passing along small strings it is essentially *O(1)* for our purposes. Therefore the function has the following time complexity (disregarding the *O(1)* operations based on the fourth rule):

$$O(evaluatePostfix) = O(expression.split("")) + O(handlingOperands) + O(handlingOperators)$$

$$O(evaluatePostfix) = O(n + 2n) = O(3n)$$

So overall, `evaluatePostfix` has *O(n)* time complexity by the first rule. Now returning to the original equation (1):

$$O(Exercise3) = O(convertToPostfix) + O(evaluatePostfix) = O(3n + 3n) = O(6n)$$

$$6n \leq cn = 7n, \forall n \geq 1, c = 7$$

Therefore by asymptotic analysis where $n_o = 1$ and $c = 7$ means that the overall time complexity of *Exercise3* is *O(n)* .