Mergesort with Lists & Stacks

By: Matthew S Montoya

1 Introduction

In this lab, a program will take in a user's input, an integer *n*, and create a Linked List of that size. Each node's value will be between 0 and (m-1), where m is also an integer provided by the user. Once the lusts are created, it must be sorted recursively via the mergesort algorithm, and iteratively using stacks.

2 Proposed Solution & Design Implementation

To achieve the end goal, the program will consist of eight different methods. Each method will be used to solve one part of the problem. For example, the main method will call a separate method, in which the user will determine how long they would like the length of list to be, as well as the range of values each node should have. This is done to keep the code organized and solve the problem one part at a time. When asking for user input, the program will use a scanner, and to prevent any errors, will flush itself prior to taking input of a different variable kind (eg. Integers, then doubles), should that be necessary.

2.1— The Main Method

The main method's only line of code will call a second method that will prompt the user for more information about the linked list, to properly construct it. This is done to prevent unnecessary problem-solving solving within the main method.

2.2— The *userInput* Method

The *userInput* method will be a non-return type method in which the program will ask the user integer *n*, the length of the list and integer *m*, the possible values between 0 and (m-1) that each node integer value can randomly be. After the list is created, it is this method which will call other methods in the program to either build, print, or sort the list.

2.3— The *printList* Method

The *printList* method will be a non-return type method in which the contents of a list are printed. Taking in the parameter, *iNode list*, the method will traverse the list and print the contents in order, whether the list is sorted or unsorted. This method is called a series of time during the program to display the contents of the list as it is created, as well as after the list has been sorted.

2.4— The *buildList* Method

The *buildList* method will be an *iNode* return type method in which the linked list is constructed using the parameters *n* and *m*, the length of the list and the range of possible integer values each node may contain. Using a for-loop, the list will be constructed one node at a time, connecting each new node to the front of the previously created node. This method will return the list to the *userInput* method.

2.5— The *mergeSort* Method

The *mergeSort* method will be an iNode return type method in which the LinkedList, created in the *buildList* method, is recursively sorted. Taking in a parameter, *constructedList*, this method will split the list, depending on whether the list the node contain an even or odd integer value. Note: The method will call a helper method, the *merge* method to finish sorting the list using the algorithm, and send the even and odd integers to be merged into an organized list.

2.6— The *merge* Method

A helper method for the *mergesort* method, this method will be an *iNode* return type method in which the individual node values are taken as parameters. If there are no empty nodes, this will continuously compare each even and odd node and place the lower value node after the one with a higher value. If, for example, all the even nodes are used and there are only odd-valued nodes left, those nodes will be attached at the front of the list (and vise-versa for the odd nodes). Using a while-loop, the method will reverse the linked list and return that to the *mergeSort* method, which will then be returned to the *userInput* method. The method will then be printed to show the user that the input has been sorted.

2.7— The *reverseLinkedList* Method

The *reverseLinkedList* method will be the first step in solving the iterative mergesorting of stacks. Using the same for-loop that reversed the linked list in the *merge* method, the linked list will be reversed in this method, so that it will now be in descending order. This is done to push the values into a stack in ascending order, as stacks follow the first-in-last-out rule. This method will then create and print the value of the stacks, and call a helper method, using some of the code that was provided as a part of the lab assignment, to sort the stack.

2.8— The *stackMergeSort* Method

Using part of the code provided to us for the lab, the *stackMergeSort* method will iteratively sort the stack into individual elements. The will be like its recursive counterpart from earlier in the program, and return an iNode type with the sorted stack. Note: this method will call a helper method to merge the elements back into a sorted stack.
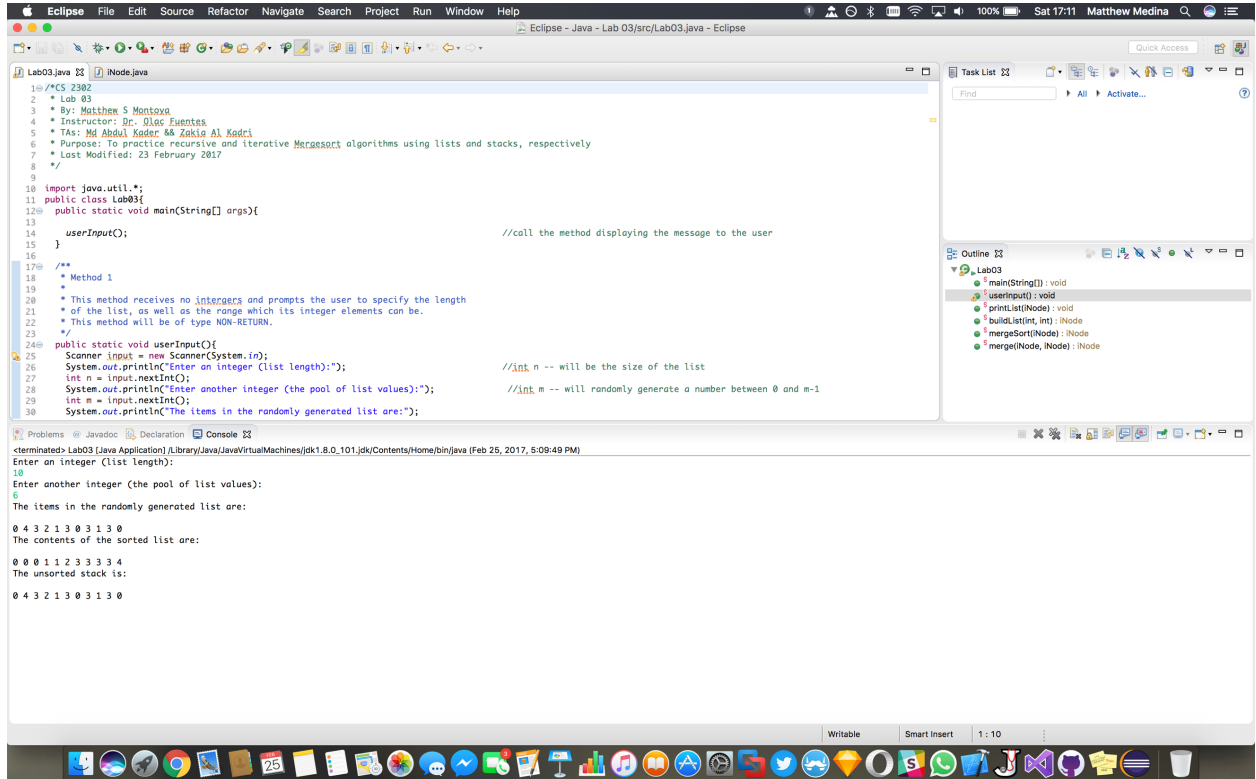
2.9— The *mergeStacks* Method

The *mergeStacks* method will be an iNode return type method in which the sorted stack will be sent back to the *stackMergeSort* method and then sent back to the *userInput* method. From there, it will be printed to show the user it has been sorted. Like the *merge* method, it will compare even and odds elements and should any elements be left over, they will be placed into the stack, to prevent errors, including null pointer exceptions.

3 Experimental Results

3.1 — Program Output

The output of the code (found in the appendix) is shown below. The first two images are from the same test. Note: The first image includes the beginning of the code (which can also be found in the *Appendix* section), as proof the executed code is not the work of others. The following images is an enhanced version of the program's output during various executions.

The program execution showing the before and after for the recursive mergesort, as well as the before for the iterative merge sort of the stacks

```
Enter an integer (list length):
10
Enter another integer (the pool of list values):
6
The items in the randomly generated list are:

0 4 3 2 1 3 0 3 1 3 0
The contents of the sorted list are:

0 0 0 1 1 2 3 3 3 3 4
The unsorted stack is:

0 4 3 2 1 3 0 3 1 3 0
```
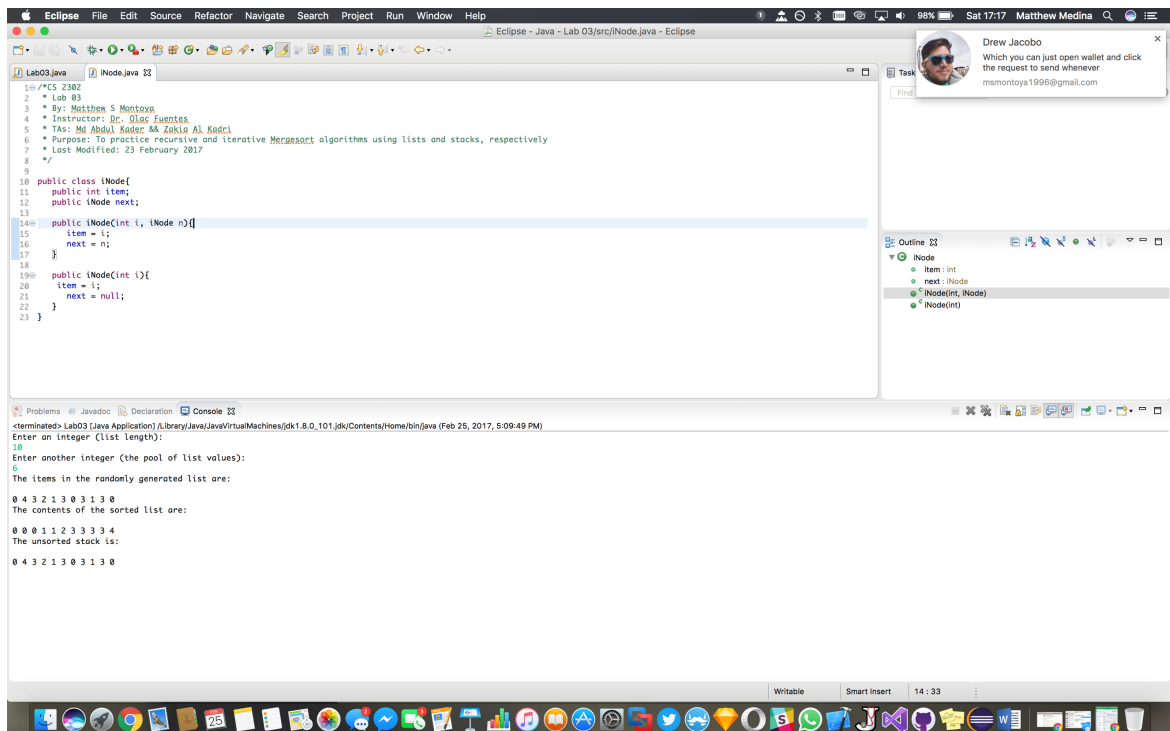
The same output from the previous test but enhanced.
NOTE: The after for the mergesort with stacks is not shown, as I was unable to have the methods successfully compile.

The iNode class used for the linked list.

4 Conclusions

In this lab, I learned how to implement a better version of mergesort, when compared to the version I implemented in lab 1. Following the TA's advice to insert the remainder of the elements into an array/linked list to help prevent null-pointer exceptions and unsorted lists, I could correctly sort data using merge sort. Although I was unable to correctly implement an iterative version of mergesort on the stacks, I still look forward to learning how to do so.

5 Appendix

5.1— Documentation & Disclosure

- Methods are documented using the following format—
  ```
  /*METHOD NUMBER:
  *
  * BRIEF MENTION OF METHOD'S PURPOSE
  public static void methodName(parameters){
  }
  ```
- Non-obvious code is documented using the following format—
  ```
  for(int i=0; i<x.length-1; i++){        //summarizes the function of the loop
        x[i] = i;           //may include brief logical reasoning
  }
  ```
- Disclosure: Though keeping the original digital format of the code (indention lengths, spaces, etc.) the use of Microsoft Word has reformatted the code in the document to be of different space and tab lengths. The font size has been reduced to help alleviate some visual misrepresentations.

5.2— Student Code

```
/* By: Matthew S Montoya
 * Purpose: To practice recursive and iterative Mergesort algorithms using lists and stacks,
respectively
 * Last Modified: 23 February 2017
 */

import java.util.*;
public class Lab03{
 public static void main(String[] args){

   userInput(); //call the method displaying the message to the user
 }

 /**
  * Method 2
  *
  * This method receives no intergers and prompts the user to specify the length
```

```
    * of the list, as well as the range which its integer elements can be.
    * This method will be of type NON-RETURN.
    */
  public static void userInput(){
    Scanner input = new Scanner(System.in);
    System.out.println("Enter an integer (list length):");                          //int n -- will be the
size of the list
    int n = input.nextInt();
    System.out.println("Enter another integer (the pool of list values):");              //int m -
- will randomly generate a number between 0 and m-1
    int m = input.nextInt();
    System.out.println("The items in the randomly generated list are:");
    iNode constructedList = buildList(n, m);//method 3

    printList(constructedList);//method 2; will print the contents of the LinkedList
    iNode mergeSortedList = mergeSort(constructedList);//method 4

    System.out.println("\nThe contents of the sorted list are:");
    printList(mergeSortedList);//method 2; will print the contents of the merge-sorted LinkedList

    System.out.println("\nThe unsorted stack is:");
    printList(constructedList);//method 2; will inform user this is the unsorted stack

    //Stack stack = reverseLinkedList(constructedList);//method 6; will reverse the LinkedList to use
only one stack
  }

  /**
   * Method 3
   *    printList();
   *
   * This helper method prints the contents of a LinkedList, to show the user the
   * program is executing properly.
   * This method will be a NON-RETURN method.
   */
  public static void printList(iNode list){
    System.out.println();
    while(list != null){
      System.out.print(list.item+" ");
      list=list.next;
    }
  }

  /**
   * OBJECTIVE 1
   *    Method 4
   *
   * This method receives two integers, n and m, representing the
   * length of the list and the the range (0 to m-1) which the node.item can be.
   * This method WILL RETURN the constructed list.
   */
  public static iNode buildList(int n, int m){
    iNode list = new iNode(0, null);
```

```java
      for(int i=0; i<n; i++){
         xlist= new iNode((int)(Math.random()*(m-1)),list);//populates the list with integers from 0 to (m-
1)
      }
      return list;
   }

   /**
    * OBJECTIVE 2
    *    Method 5
    *
    * This method receives the constructed list that was created in method 3, and
    * split the list into indivudal elements for sorting with mergeSort.
    * This method WILL RETURN a list sorted via merge sort.
    */

   public static iNode mergeSort(iNode constructedlist){
      iNode temp = constructedlist;
      if(constructedlist==null || constructedlist.next == null)
         return constructedlist;

      iNode leftHalf = null;
      iNode rightHalf = null;
      int nodeLength = 0;

      while(constructedlist != null){           //get length of list
         nodeLength++;
         constructedlist = constructedlist.next;
      }

      for(int i=0; i<nodeLength; i++){           //split into evens and odds
         if(i%2==0)
            leftHalf = new iNode(temp.item, leftHalf);
         if(i%2==1)
            rightHalf = new iNode(temp.item, rightHalf);

         temp = temp.next;
      }

      leftHalf = mergeSort(leftHalf);
      rightHalf = mergeSort(rightHalf);

      iNode mergedList = merge(leftHalf, rightHalf);//method 5; merge everything back together
      return mergedList;
   }

   /**
    * OBJECTIVE 2
    *    Method 6
    *
    * This method receives two iNodes, leftHalf and rightHalf of the lists,
    * which will build a new list of sorted items, sorted by Merge Sort
    * This method WILL RETURN the sorted list.
```

```java
     */
     public static iNode merge(iNode leftHalf, iNode rightHalf){
       iNode mergeSorted = null;
       while(leftHalf != null && rightHalf != null){
        if(leftHalf.item <= rightHalf.item){
          mergeSorted = new iNode(leftHalf.item, mergeSorted);
          leftHalf = leftHalf.next;
        }
        else{
          mergeSorted = new iNode(rightHalf.item, mergeSorted);
          rightHalf = rightHalf.next;
        }
       }

       //Lesson from lab 1: if one of the halves is empty, daisy-chain the rest of the elements from the
     other half into the last part of the list
       while(leftHalf != null){                    //while the rightHalf is empty
         mergeSorted = new iNode(leftHalf.item, mergeSorted);//place the rest of the leftHalf elements into
     the list
         leftHalf = leftHalf.next;
       }

       while(rightHalf != null){//while the leftHalf is empty
         mergeSorted = new iNode(rightHalf.item, mergeSorted);//add the rest of the rightHalf elements to
     the list
         rightHalf = rightHalf.next;
       }
       //printList(sorted);//prove to user that the list is sorted

       //okay just kidding, switch the list around
       iNode returnMergeSorted = null;
       while(mergeSorted != null){
        returnMergeSorted = new iNode (mergeSorted.item, returnMergeSorted);
        mergeSorted = mergeSorted.next;
       }
       return returnMergeSorted;
     }


     /**
      * OBJECTIVE 3
      *    Method 7
      *
      * This method receives the origional constructed LinkedList, in which
      * it will reverse it, so that the values can be pushed into a stack.
      * This WILL RETURN
      *
      *
     public static void reverseLinkedList(iNode constructedlist){
     iNode reversed = null;          //FILO: So switch the link list around
     int counter = 0;                     //Apply the same logic as the recursive method
     while(constructedlist != null){
     reversed = new iNode (constructedlist.item, reversed);
```

```java
        counter++;
        constructedlist = constructedlist.next;
        }
        System.out.println("The reversed list is:\n");
        printList(reversed);
        return stackMergeSort(reversed, counter);
        }

        public static Stack merge(iNode reversed, int counter){
        Stack stack = new Stack();
        while(reversed != null){
        stack1.push(reversed.item);
        reversed = reversed.next;
        }



        }

        * OBJECTIVE 3
        *    Method 8
        *
        * This method receives the original constructed LinkedList, in which
        * it will apply iterative sorting (the first half of mergeSort) to stacks.
        * This WILL RETURN a sorted iNode stack
        *
        public static iNode stackMergeSort(iNode list, int n){   //Apply the code Dr. Fuentes provided and
make minor edits
        Stack<MergesortRecord> stack = new Stack();   //use stacks
        MergesortRecord msR = new MergesortRecord(false, list);
        stack.push(msR);
        while(!stack.empty()){
        msR=stack.pop();                                //Keep poping elements and create new stacks
        if(msR.getSorted()){
        list = mergeStacks(list, n);

        * OBJECTIVE 3
        *    Method 9
        *
        * This method receives the stacks created in method 7, and will return
        * the mergesorted stacks to method 7, which will then return it to the main method
        * This WILL RETURN a sorted iNode stack
        *
        public static iNode mergeStacks(iNode list, int n){
        iNode temp = list;
        for(int i =0; i<(n/2-1); i++){
        temp=temp.next;
        }
        iNode list2 = recursiveMerge(temp.next, n-(n/2));//apply the same logic as mergesort in the lists
        temp.next=null;   //assign the final value to null
        }
        */
}
```