CS 3310: Data and File Structures

Instructor: Dr. Ajay K. Gupta, Western Michigan University

Lab TA: Yu Guo

Matt Peter

**Software Life Cycle Report – Assignment 2**

**Phase 1: Specification**

Objectives:

1. Write a program that uses multiple techniques to sort arrays and linked-lists
2. Analyze the advantages and disadvantages of arrays and linked-lists
3. Compare the results you get in terms of theoretical and empirical complexities

Write a program that:

1. Generates n random floating-point numbers from 1 to m
2. Stores the randomly generated values in an array and linked-list
3. Sorts the array and linked-list using bubble-sort, merge-sort, quick-sort, and insertion-sort in a non-decreasing order and outputs, at most, the first 100 values

**Phase 2: Design**

2.1 Modules and their Basic Structure

1. Module 1: Class Hw2Main contains:
   1. public static void main (String[] args) – Gets information from the user, generates array and linked list, and calls sort methods
2. Module 2: Class ArraySort contains:
   1. public void bubbleSort (T[] values) – Sorts the array using a bubble-sort sorting method
   2. public void mergeSort (T[] values, int first, int last) – Sorts the array using a merge-sort sorting method
   3. public void quickSort (T[] values, int first, int last) – Sorts the array using a quick-sort sorting method
   4. public void insertionSort (T[] values) – Sorts the array using a insertion-sort sorting method
   5. public void selectionSort (T[] values) – Sorts the array using a selection-sort sorting method
3. Module 3: Class LinkedListSort contains:
   1. public void bubbleSort (LinkedList<T> values) – Sorts the linked list using a bubble-sort sorting method
   2. public void mergeSort (LinkedList<T> values, int first, int last) – Sorts the linked list using a merge-sort sorting method
   3. public void quickSort (LinkedList<T> values, int first, int last) – Sorts the linked list using a quick-sort sorting method
   4. public void insertionSort (LinkedList<T> values) – Sorts the linked list using an insertion-sort sorting method
   5. public void selectionSort (LinkedList<T> values) – Sorts the linked list using a selection-sort sorting method

2.2 Pseudocode for the Modules

2.2.1 Pseudocode for Hw2Main

1a. Hw2Main Pseudocode Refinement #1:

// Declare and Instantiate Scanner and Random objects

// Declare and Instantiate ArraySort and LinkedListSort objects

// Declare and Instantiate array, linked list, and other variables

// Gather input from the user

// Generate values for linked list and array

// Call sort methods using ArraySort and LinkedListSort objects

// Print out resulting data

2.2.2 Pseudocode for ArraySort

2a. ArraySort Pseudocode Refinement #1:

// Method bubbleSort – Sorts the array using a bubble-sort sorting method

// Method mergeSort – Sorts the array using a merge-sort sorting method

// Method quickSort – Sorts the array using a quick-sort sorting method

// Method insertionSort – Sorts the array using an insertion-sort sorting method

// Method selectionSort – Sorts the array using a selection-sort sorting method

2b. ArraySort Pseudocode Refinement #2:

// Method bubbleSort – Sorts the array using a bubble-sort sorting method

// Enter a for loop that sets the upper bound of numbers that will be compared (decreasing as the highest numbers are found)

// Enter another for loop that keeps track of which numbers are currently being compared

// If one value is greater than the next, switch the values

// Method mergeSort – Sorts the array using a merge-sort sorting method

// Check to make sure that there are at least two values in the array currently being sorted

// Find the position of the middle of the array

// Call mergeSort for the first half

// Call mergeSort for the second half

// Declare variables to keep track of what values have already been sorted from both arrays

// Combine the two arrays into one sorted array

// Figure out which value at the current position in both arrays is smaller and put that in the new array

// When all the values in one of the arrays have been used, fill the rest of the new array with the values that are left

// Transfer the values in the sorted array back to the original array

// Method quickSort – Sorts the array using a quick-sort sorting method

// Declare and Instantiate variables to keep track of values being compared as well as a pivot point for the portion being sorted

// Starting from opposite ends, search for values that are less than and greater than the pivot value and swap them

// Starting from the lower end, search the array for a value that is less than the pivot value

// Starting from the upper end, search the array for a value that is greater than the pivot value

// If the greater value is to the left of the lesser value, swap the two values

// Continue this process until values that are less than and greater than the pivot value are separated into two portions of the array

// Call quickSort for the values that are at or above the position where the sorting stopped

// Call quickSort for the values that are at or below the position where the sorting stopped

// Method insertionSort – Sorts the array using an insertion-sort sorting method

// Enter a for loop that sets the number that will be compared to the previous portion of the array

// Enter another for loop that keeps track of which numbers are currently being compared

// If the value that is currently being checked is lesser than the previous, swap the two

// Otherwise, exit the second loop

// Method selectionSort – Sorts the array using a selection-sort sorting method

// Enter a for loop that sets the lower bound of the numbers that will be compared (increases each time through)

// Declare variables to keep track of the lowest value found and its position in the array

// Enter another for loop that will go through all of the values in the array from the lower bound to the end

// Compare the value at the current position to lowest value found in this interval of the first for loop

// If it is less than the lowest value, set this current value as the new lowest value

// Once the lowest value in the unsorted portion of the array has been found, swap it with the value at its proper position in the array

2.2.3 Pseudocode for LinkedListSort

3a. LinkedListSort Pseudocode Refinement #1:

// Method bubbleSort – Sorts the linked list using a bubble-sort sorting method

// Method mergeSort – Sorts the linked list using a merge-sort sorting method

// Method quickSort – Sorts the linked list using a quick-sort sorting method

// Method insertionSort – Sorts the linked list using an insertion-sort sorting method

// Method selectionSort – Sorts the linked list using a selection-sort sorting method

3b. LinkedListSort Pseudocode Refinement #2:

// Method bubbleSort – Sorts the linked list using a bubble-sort sorting method

// Enter a for loop that sets the upper bound of numbers that will be compared (decreasing as the highest numbers are found)

// Enter another for loop that keeps track of which numbers are currently being compared

// If one value is greater than the next, switch the values

// Method mergeSort – Sorts the linked list using a merge-sort sorting method

// Check to make sure that there are at least two values in the linked list currently being sorted

// Find the position of the middle of the linked list

// Call mergeSort for the first half

// Call mergeSort for the second half

// Declare variables to keep track of what values have already been sorted from both linked lists

// Declare and Instantiate a new linked list to hold the sorted values

// Combine the two linked lists into one sorted linked list

// Figure out which value at the current position in both linked lists is smaller and put that in the new linked list

// When all the values in one of the linked lists have been used, fill the rest of the new linked list with the values that are left

// Transfer the values in the sorted linked list back to the original linked list

// Method quickSort – Sorts the linked list using a quick-sort sorting method

// Declare and Instantiate variables to keep track of values being compared as well as a pivot point for the portion being sorted

// Starting from opposite ends, search for values that are less than and greater than the pivot value and swap them

// Starting from the lower end, search the linked list for a value that is less than the pivot value

// Starting from the upper end, search the linked list for a value that is greater than the pivot value

// If the greater value is to the left of the lesser value, swap the two values

// Continue this process until values that are less than and greater than the pivot value are separated into two portions of the linked list

// Call quickSort for the values that are at or above the position where the sorting stopped

// Call quickSort for the values that are at or below the position where the sorting stopped

// Method insertionSort – Sorts the linked list using an insertion-sort sorting method

// Enter a for loop that sets the number that will be compared to the previous portion of the linked list

// Enter another for loop that keeps track of which numbers are currently being compared

// If the value that is currently being checked is lesser than the previous, swap the two

// Otherwise, exit the second loop

// Method selectionSort – Sorts the linked list using a selection-sort sorting method

// Enter a for loop that sets the lower bound of the numbers that will be compared (increases each time through)

// Declare variables to keep track of the lowest value found and its position in the linked list

// Enter another for loop that will go through all of the values in the linked list from the lower bound to the end

// Compare the value at the current position to lowest value found in this interval of the first for loop

// If it is less than the lowest value, set this current value as the new lowest value

// Once the lowest value in the unsorted portion of the linked list has been found, swap it with the value at its proper position in the linked list

**Phase 3: Risk Analysis**

There are no known risks associated with this application.

**Phase 4: Verification**

All the steps of the algorithm were checked to ensure correct results in all circumstances. A variety of tests with a wide range of values were also done to check for correct output.

**Phase 5: Coding**

5a. Code Refinement #1:

File Hw2Main.java

package edu.wmich.cs3310.MPeter.hw2;

import java.util.\*;

public class Hw1Main {

public static void main(String[] args) {

// Declare and Instantiate Scanner and Random objects

// Declare and Instantiate ArraySort and LinkedListSort // objects

// Declare and Instantiate array, linked list, and other // variables

// Gather input from the user

// Generate values for linked list and array

// Call sort methods using ArraySort and LinkedListSort // objects

// Print out resulting data

}

}

File ArraySort.java

package edu.wmich.cs3310.MPeter.hw2;

public class ArraySort<T extends Comparable<T> implements IArraySort<T> {

public void bubbleSort(T[] values) {

// Enter a for loop that sets the upper bound of numbers // that will be compared (decreasing as the highest // numbers are found)

// Enter another for loop that keeps track of which // numbers are currently being compared

// If one value is greater than the next, // switch the values

}

@SuppressWarnings(“unchecked”)

public void mergeSort(T[] values, int first, int last) {

// Check to make sure that there are at least two values in // the array currently being sorted

// Find the position of the middle of the array

// Call mergeSort for the first half

// Call mergeSort for the second half

// Declare variables to keep track of what values have // already been sorted from both arrays

// Combine the two arrays into one sorted array

// Figure out which value at the current position in // both arrays is smaller and put that in the new // array

// When all the values in one of the arrays have been // used, fill the rest of the new array with the // values that are left

// Transfer the values in the sorted array back to the // original array

}

public void quickSort(T[] values, int first, int last) {

// Declare and Instantiate variables to keep track of // values being compared as well as a pivot point for // the portion being sorted

// Starting from opposite ends, search for values that are // less than and greater than the pivot value and swap // them

// Starting from the lower end, search the array for // a value that is less than the pivot value

// Starting from the upper end, search the array for // a value that is greater than the pivot value

// If the greater value is to the left of the lesser // value, swap the two values

// Continue this process until values that are less than // and greater than the pivot value are separated into // two portions of the array

// Call quickSort for the values that are at or above the // position where the sorting stopped

// Call quickSort for the values that are at or below the // position where the sorting stopped

}

public void insertionSort(T[] values) {

// Enter a for loop that sets the number that will be // compared to the previous portion of the array

// Enter another for loop that keeps track of which // numbers are currently being compared

// If the value that is currently being checked // is lesser than the previous, swap the two

// Otherwise, exit the second loop

}

public void selectionSort(T[] values) {

// Enter a for loop that sets the lower bound of the // numbers that will be compared (increases each time // through)

// Declare variables to keep track of the lowest // value found and its position in the array

// Enter another for loop that will go through all of // the values in the array from the lower bound to // the end

// Compare the value at the current position to // lowest value found in this interval of // the first for loop

// If it is less than the lowest value, // set this current value as the new // lowest value

// Once the lowest value in the unsorted portion of // the array has been found, swap it with the // value at its proper position in the array

}

}

File LinkedListSort.java

package edu.wmich.cs3310.MPeter.hw2;

import java.util.\*;

public class LinkedListSort<T extends Comparable<T>> implements ILinkedListSort<T> {

public void bubbleSort(LinkedList<T> values) {

// Enter a for loop that sets the upper bound of numbers // that will be compared (decreasing as the highest // numbers are found)

// Enter another for loop that keeps track of which // numbers are currently being compared

// If one value is greater than the next, // switch the values

}

@SuppressWarnings(“unchecked”)

public void mergeSort(LinkedList<T> values, int first, int last) {

// Check to make sure that there are at least two values in // the linked list currently being sorted

// Find the position of the middle of the linked list

// Call mergeSort for the first half

// Call mergeSort for the second half

// Declare variables to keep track of what values have // already been sorted from both linked lists

// Declare and Instantiate a new linked list to hold the // sorted values

// Combine the two linked lists into one sorted linked list

// Figure out which value at the current position in // both linked lists is smaller and put that in // the new linked list

// When all the values in one of the linked lists // have been used, fill the rest of the new linked // list with the values that are left

// Transfer the values in the sorted linked list back to // the original linked list

}

public void quickSort(LinkedList<T> values, int first, int last) {

// Declare and Instantiate variables to keep track of // values being compared as well as a pivot point for // the portion being sorted

// Starting from opposite ends, search for values that are // less than and greater than the pivot value and swap // them

// Starting from the lower end, search the linked // list for a value that is less than the pivot // value

// Starting from the upper end, search the linked // list for a value that is greater than the pivot // value

// If the greater value is to the left of the lesser // value, swap the two values

// Continue this process until values that are less than // and greater than the pivot value are separated into // two portions of the linked list

// Call quickSort for the values that are at or above the // position where the sorting stopped

// Call quickSort for the values that are at or below the // position where the sorting stopped

}

public void insertionSort(LinkedList<T> values) {

// Enter a for loop that sets the number that will be // compared to the previous portion of the linked list

// Enter another loop that keeps track of which // numbers are currently being compared

// If the value that is currently being checked // is lesser than the previous, swap the two

// Otherwise, exit the second loop

}

public void selectionSort(LinkedList<T> values) {

// Enter a for loop that sets the lower bound of the // numbers that will be compared (increases each time // through)

// Declare variables to keep track of the lowest // value found and its position in the linked list

// Enter another for loop that will go through all of // the values in the linked list from the lower // bound to the end

// Compare the value at the current position to // lowest value found in this interval of // the first for loop

// If it is less than the lowest value, // set this current value as the new // lowest value

// Once the lowest value in the unsorted portion of // the linked list has been found, swap it with // the value at its proper position in the linked // list

}

}

**Phase 6: Testing**

|  |  |  |
| --- | --- | --- |
| Input/Output Analysis | | |
|
| **Method** | **Input Size** | **Output Size** |
| Hw2Main | 0 | 0 |
| bubbleSort (ArraySort) | n | 0 |
| mergeSort (ArraySort) | n + 2 | 0 |
| quickSort (ArraySort) | n + 2 | 0 |
| insertionSort (ArraySort) | n | 0 |
| selectionSort (ArraySort) | n | 0 |
| bubbleSort (LinkedListSort) | n | 0 |
| mergeSort (LinkedListSort) | n + 2 | 0 |
| quickSort (LinkedListSort) | n + 2 | 0 |
| insertionSort (LinkedListSort) | n | 0 |
| selectionSort (LinkedListSort) | n | 0 |

The table above shows an analysis of the input and output for each method.

The table below shows an analysis of the space used within each method as well as the big O notation for each of the resulting equations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Space Complexity Analysis | | | | |
|
| **Method** | | **Space Usage** | **Space Complexity** | |
| Hw2Main | | 4n + 8 | O(n) | |
| bubbleSort (ArraySort) | | 3 | O(1) | |
| mergeSort (ArraySort) | | n + 4 | O(n) | |
| quickSort (ArraySort) | | 4 | O(1) | |
| insertionSort (ArraySort) | | 3 | O(1) | |
| selectionSort (ArraySort) | | 5 | O(1) | |
| bubbleSort (LinkedListSort) | | 3 | O(1) | |
| mergeSort (LinkedListSort) | | n + 3 | O(n) | |
| quickSort (LinkedListSort) | | 4 | O(1) | |
| insertionSort (LinkedListSort) | | 3 | O(1) | |
| selectionSort (LinkedListSort) | | 5 | O(1) | |
| Time Complexity Analysis | | | | |
|
| **Method** | **Instruction Count** | | | **Time Complexity** |
| Hw2Main | ~((32/3)\*(n^3)+7(n^2)\*log(n)) | | | O(n^3) |
| bubbleSort (ArraySort) | 2(n^2) - 2n | | | O(n^2) |
| mergeSort (ArraySort) | 2n\*log(n) + 6n | | | O(n\*log(n)) |
| quickSort (ArraySort) | 8n\*log(n) + 5n | | | O(n\*log(n)) |
| insertionSort (ArraySort) | 2(n^2) - 2n | | | O(n^2) |
| selectionSort (ArraySort) | (3/2)\*(n^2)+(7/2)n | | | O(n^2) |
| bubbleSort (LinkedListSort) | 4(n^3)-(9/2)\*(n^2)+(1/2)n | | | O(n^3) |
| mergeSort (LinkedListSort) | 4(n^2)\*log(n)+4n\*log(n)+4n-4 | | | O((n^2)\*log(n)) |
| quickSort (LinkedListSort) | 3(n^2)\*log(n)+4n\*log(n) | | | O((n^2)\*log(n)) |
| insertionSort (LinkedListSort) | 4(n^3)-(9/2)\*(n^2)+(1/2)n | | | O(n^3) |
| selectionSort (LinkedListSort) | (8/3)\*(n^3)-(n^2)-(2/3)n-1 | | | O(n^3) |

All of the time complexities that I calculated (shown in table above) ended up agreeing fairly well with the data that I collected (shown in the graphs below). As it can be seen, the time complexity for sorting using an array is better than sorting using a linked list by a factor of n in all cases (due to the fact that it takes a factor of n time to reach relative values in different linked lists (such as value at the first quarter mark, middle, and third quarter mark), whereas it takes constant time no matter what value you want it an array). This shows one clear advantage of using arrays over linked lists, and normally, linked lists would hold the advantage over arrays of only ever requiring the amount of space that is required by the data you have at a given time. However, in this case, since the amount of space you’ll need for holding the data is decided right away by the user and is never changed throughout the course of the program, this advantage doesn’t help at all in this case. In fact, because each node in a linked list requires space for the data as well as a reference to the next node in the list, the linked list takes up a lot more space than the array. Overall, it is a much better choice to use arrays in this situation over linked lists.

Bubble Sort, Insertion Sort, and Selection Sort time seen increasing at much faster rate than Merge Sort, Quick Sort, and Built-In Sort when sorting arrays.

Insertion Sort, Bubble Sort, and Selection Sort increasing at the fastest rate, O(n^3)

Merge Sort and Quick Sort increasing at the medium rate, O((n^2)\*log(n))

Built-In Sort increasing at the slowest rate, O(n) (although would probably become O(n\*log(n)))

**Phase 7: Refining the Program**

All required features are included in the program so no refinements are needed.

**Phase 8: Production**

A zip file containing source files, a Javadoc, and test data have been submitted.

**Phase 9: Maintenance**

Any maintenance can be performed once feedback has been obtained.