CS3310 Data and File Structures with Java
Instructor: Ajay K Gupta
Lab TA: Yu Guo, Zijiang J Yang
Assignment: 3

Jonah Kubath

Software Life Cycle Report – Assignment 3

Phase 1: Specification

- Practice bubble sort, insertion sort, selection sort, and merge sort
- Developing efficient in space and time solutions
- Compare data structures for four different sorting methods

Program Output is at the end of this file.

Phase 2 Design:
Modules and Basic structure
Package – edu.wmich.cs3310.hw3.Kubath.application

Module 1: Class Main
Fields:

1. String[] userInput – holds the user's name and length of array to generate
2. Int length – holds the length of the array / list to generate
3. String name – name of the user
4. Double[] list – holds the run times of the 4 lists
5. Double[] array – holds the run times of the 4 lists
6. Long start – start time of the sorting methods
7. Long end – end time of the sorting methods
8. Long microseconds – total time of the sorting methods
9. Character[] sorted – holds the sorted array for output to the console
10. Character[] data – holds the randomly generated array
11. Character[] copy – holds the copied array data
12. MyMap newMap – object to holds the sorting list values
13. Int timesRun – used to run the program 10 times
14. CharList linkedList – linked list of randomly generated characters
15. CharList copyLi – copied linked list
16. CharNode leftNode – first node of the linked list, used for mergeSort()
17. CharNode next – the second node of the linked list, used for mergeSort()

Methods:

1. CharList generateList(int length) – takes in a length to create a list and generates random characters added to the list
2. Void printArray(Character[] array) – prints the given array
3. Void printList(CharList list) prints the given list
4. String[] getInput() – takes in the user's name and length of array to generate. Returns an array of the user's input
5. Void copyArray(Character[] data, Character[] copy) – copies data to the copy array
6. CharList copyList(CharList orgin) copies the orgin to a new CharList and returns a pointer to the new CharList
7. Character[] copyListToArray(CharList list, Character[] array) – copies the data in the list to the array

Module 2: MyMap

Fields:

1. HashMap<Character, Integer> map – holds the alphabet with their weight as a value to be used during sorting
2. Set<?> entrySet – used to iterate all values of the map
3. Iterator<?> it – used to iterate over the values of the map

Methods:

1. Int test(char left, char right) – takes in two characters and uses the hashmap to determine which character is greater.

Module3: Sort

Fields:

None

Methods:

1. Void bubbleSort(Character[] array, MyMap map, int begin) – Bubble sorts the given array using the map object to determine the sorting of characters.
2. <T extends Comparable<T>> void bubbleSort(T[] array, int begin) – Generic bubble sort.
3. <T> void bubbleSort(T[] array, Comparator<? Super T> comparator) – Generic bubble sort for an array using the comparator object.
4. CharList bubbleSort(CharList list, MyMap map, int begin) – Bubble sort the given linked list using the map list to determine the sorting of characters.
5. Void selectionSort(Character[] array, MyMap map, int begin) – selection sort on the given array using the map object to determine the sorting of characters.

6. <T> void selectionSort(T[] array, Comparator<? Super T> comparator) – Generic selection sort using the comparator.
7. Void selectionSort(CharList list, MyMap map, int begin) – selection sort on the given list using the map object to determine the sorting of characters.
8. Void swap(CharList list, int left, int right) – takes in two positions of the linked list and swaps the values held in those nodes.
9. CharNode sortedMerge(MyMap map, CharNode a, CharNode b) – merges back the CharNode objects a and b using the map object to determine the sorting of characters.
10. CharNode mergeSort(CharNode h, MyMap map) – performs a mergeSort on the given linked list using the map object to determine the sorting of characters.
11. CharNode getMiddle(CharNode h) – returns the node in the middle of the linked list
12. Void mergeSort(Character[] array, Character[] temp, MyMap map, int left, int right) – performs merge sort on the array using the temp array to store data. Uses the map object to determine the sorting of characters.
13. Character[] insertionSort(Character[] array, MyMap map, int begin) – performs insertion sort on the array using the map object to determine the sorting of characters.
14. CharList insertionSort(CharList list, MyMap map, int begin) – performs insertion sort on the list using the map object to determine the sorting of characters.

Package – LinkedList

Module 1: CharList

Fields:

1. CharNode head – points to the head of the list
2. CharNode tail – points to the tail of the list

Methods:

1. CharList() – default constructor that builds a head node and sets the tail to point to the head.
2. Void insertTail() – creates and insert a node at the tail of the list
3. Void insertTail(char newChar) – creates a new node with newChar set as data and inserts it at the end of the list
4. CharNode deleteTail() – removes the last node in the list and returns the object.

5. CharNode deleteHead() – removes the head node in the list and returns the object.
6. CharNode getHead() – returns the head of the list
7. Void setHead(CharNode newHead) – sets the newHead node to the head of the list.
8. Int length() – returns the length of the list

Module 2: CharNode

Fields:

1. Char myData – holds the data for the node
2. CharNode nextNode – holds a pointer to the next node in the list
3. CharNode previousNode – holds a pointer to the previous node in the list

Methods:

1. CharNode(char setData) – sets the data for the node being created
2. Void setData(char newData) – sets the data for the node as newData
3. Char getData() – returns the data held in the node
4. Void setNext(CharNode next) sets the next pointer
5. Void getNext() – returns the pointer to the next node
6. Void setPrevious(CharNode previous) – sets the previous node pointer
7. CharNode getPrevious() – returns the pointer to the previous node

Pseudocode

Main()

Create an array to hold the user's input
Initialize the array by calling getInput()
Hold the values in two variables
Create two arrays to hold the total time for the list sorts and the array sorts
Create variables to hold the begin, end, and total run time of the sorts
Create two arrays to hold the random characters and the sorted characters
Create a MyMap object to hold the list to determine the sort order
Create a variable to hold the number of times the program is run
Use a for loop to run the program 10 times to create an average run time
Generate and store the random characters
Copy the random characters to the copy array
Print the original list
Copy the random array to the linked list
Call the 4 sort methods on the linked list
The copy list needs to be reinitialized with the data list so that each method is called on an unsorted list
Add the total run time to the list array
Call the 4 sort methods on the array

The copy array needs to be reinitialized with the data array so that each method is called on an unsorted array

Add the total run time to the array time array

Output the data used in sorting

Output the average times of the sort methods

generateArray()

Create a new array to hold the generated characters

Generate random lowercase letters for the given length

Return the new list

generateList()

Create a new linked list object

Generate random lowercase letters for the given length and store in new nodes

printArray()

print each character in the given array

printList()

Create a temp Node to hold the position of the linked list

While the temp Node != null iterate through the list

Print the character held in the node

getInput()

Create a scanner object to read from STDIN

Create an array to store the user's input

Use a try / catch to receive the length of the array to create

Use an string to receive the input

Use Integer.parseInt to convert the string to an integer or throw an error if bad input was given

Use a try / catch to receive the user's name

Use a string to receive the input

Use another string and remove any non letters

If the strings are different lengths, then bad input was given

prompt for new input

return the array of values

copyArray()

Run through the length of the original array and copy the values to the copy array

copyList()

Create a new linked list

Run through the list and copy the values to the copy list

Return the linked list

copyListToArray()
    Create a node to hold the position in the linked list
    Create an integer variable to hold the position of the array
    Iterate through the list until the temp node is null
        Copy the data held in the node to the array[count]
        Iterate the count variable
    Return the array

MyMap

MyMap(String name)
    Create an integer to hold count
    Create a new HashMap object
    Copy the name to the HashMap – skipping already added characters
        Setting the weight of the character to the count variable
        Iterate count
    Add the rest of the alphabet to the HashMap
        Iterate count

Test()
    Compare the two characters
        Use the HashMap to get the weight of the two characters
            If left < right, return -1
            If left > right, return 1
            If left == right, return 0

Sort

bubbleSort()
    Use a for loop starting at I to iterate the list
        Use another for loop starting at I to iterate the remaining of the list
            If the current character is less than the next character swap them

selectionSort()
    Create a variable to hold the lowest value index
    Iterate through the list
        Set the index to i
        Iterate through the remaining portion of the list starting at i
            If current value of the array is less than the array at index
                Index = current position of inner loop
        Swap the value at index with the current position of i

Swap()
    If the positions to swap are the same
        Return
    Create two variables to hold the two nodes to be swapped
    Iterate through the list

Save the nodes
Swap the values held in the those nodes

sortedMerge()
if the left value is null
take values from the right
if the right value is null
take values from the left
if the value of left < right value
take value from the left
else
take value from the right

mergeSort()
if the list is null or the next node point is null
return
Call the getMiddle() method to get the middle node of the list
Create a node object to point to middle.next() node
Make the middle node.next() = null
Call mergeSort() on the left side of the list
Call mergeSort() on the right side of the list

Save the return node value of sortedMerge() method
Return the node

getMiddle()
if the node == null
return
Create two pointers to nodes
Iterate one pointer by 2
Iterate the other pointer by 1
Once the faster pointer has reach become null, the middle of the list is pointed to by the slower pointer
Return the slower pointer

CharList
CharList()
Saves the head node as a new node
Saves tail pointer to the head node

insertTail()
Create a new node
Set the previous point to the current tail
Set the tail to point to the new node

```
deleteHead()
        if tail == head
                return as the list is empty
        else
                Create a temp node to hold the head node
                Set the head node to the head.next()
                Return temp
getHead()
        if tail == head
                return null
        else
                return head

setHead()
        head = newHead

length()
        create a count variable
        Create a node to point to the current position
        While the pointer != null
                Count++
                Pointer = point.getNext()

        Return count
```

CharNode

```
CharNode
        Call setData() with the given value

setData()
        set myData field to the given data

getNext()
        return the nextNode pointer

setPrevious()
        previousNode = new Pointer

getPrevious()
        return the previousNode pointer
```

Summary Analysis

Linked List

Bubble Sort

Time Complexity

The bubble sort method is going to look at the list. When a value is out of place, the value will be swapped with value next to it. Every time a value is found out of place the method will have to look at the list again. In the worst case, the list will be in complete opposite order. This will cause the method to take every value and move it to the opposite side of the list. Looking at every value of the array for every value causes a Big-Ohh of N^2. Every time a switch is needed, the Node pointers are already stored. This allows the swap to be of constant time.

Space

Bubble sort requires N space for the characters and N pointers to the next node.

Selection Sort

Time Complexity

The selection sort method is going to search the list and find the lowest value. It will then swap the lowest value current position. When a swap occurs the lower end of the list is now sorted. The good aspect of selection sort is that the beginning of the loop is iterated each time. Selection sort will only have N^2 / 2 comparison, but this means that the worst case would Big-Ohh of N^2.

Space

Selection sort requires N space for the characters and N pointers to the next node.

Insertion Sort

Time Complexity

Insertion sort starts at the second node the list. This is because the first node is always sorted by itself. The method will then iterate through all the nodes in the list. Each node it returns to the beginning and iterates until the position to be inserted is found. This means that if the list was backwards, the list would be iterated two times per item or Big-Ohh of N^2.

Space

Insertion sort requires N space for the characters and N pointers to the next node.

Merge Sort

Time Complexity

Merge sort starts by splitting the list into two separate lists until a single node is left. It then finishes the sort by adding the nodes back to the list after testing the left and right node. To split the list in half will take Big-Ohh log(N), but to merge list back every node is looked at Big-Ohh of N. Leaving the method with Big-Ohh of (nlog(n)).

Space

Merge sort requires N space for the characters and N pointers to the next node.

Array

Bubble Sort

Time Complexity

Bubble sort for the array is very similar to the list. The array is going to have an outer loop that will iterate through the array. When a value is found to be greater than the value next it, then they are swapped. This swap forces another pass through the array. Worst case scenario is that the array is backwards and the elements will have to swap to the opposite position in the array causing Big-Ohh N^2.

Space

Bubble sort requires N space for the characters.

Selection Sort

Time Complexity

Selection sort is like the linked list selection sort. The method will start at the beginning of the list and search for the smallest item. The item is moved to the front and the sorting is incremented. This allows the method to only make N^2 / 2 comparisons. The Big-Ohh notation for selection sort would be N^2.

Space

Selection sort requires N space for the characters.

Insertion Sort

Insertion sort will start at the beginning of the list and place the items to the left in a sorted order. In the worst case, insertion sort would have to go over N objects. In the assignment we are supposed to use binary search to find the position to insert the character. This would make the insertion of log(n). The issue is that after the position is found, the array still must be shifted to make space for the insertion. The shifting would be O(n). This will result in Insertion sort being Big-Ohh N^2.

Space

Insertion sort requires N space for the characters.

Merge Sort

Merge sort will continually break the array in half until individual nodes are left. The nodes are then merged back into the array. Merge sort needs Big-Ohh nLog(n) because after the log(n) of splitting the array, merge sort takes n time to merge back each individual character.

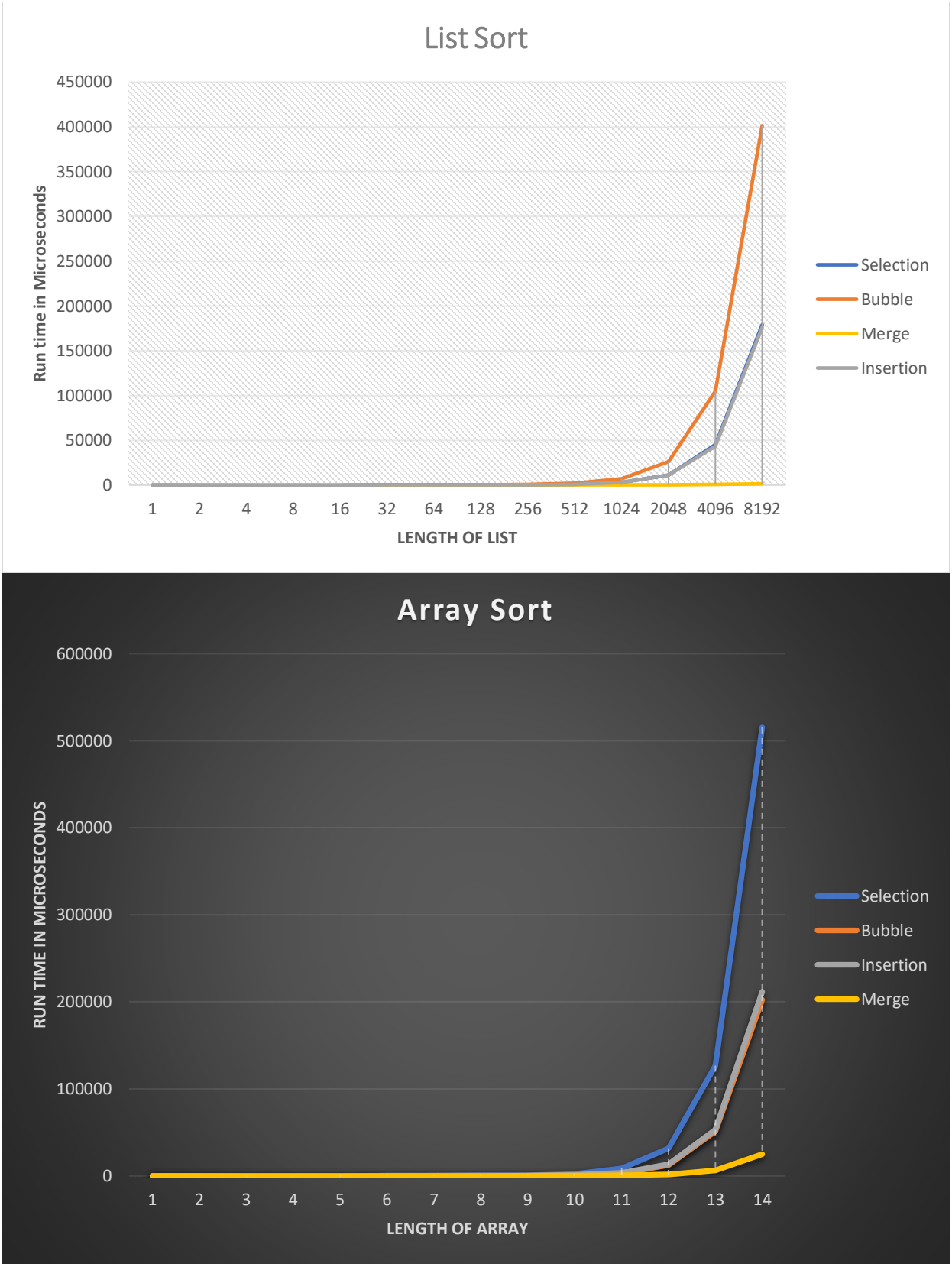
Space

Merge sort requires N space for the characters and also N space for the temp array.

Empirical Analysis

The empirical data truly shows how quick merge sort can be. Compared to the other sorting methods, merge sort is nearly linear. During my test Bubble, Insertion, and Selection all had the same growth rate in terms of Big-Ohh. They all started curving fairly quickly. The strange result was that both Linked-List and the Array had one that grew faster by a constant. For linked-list the bubble sort grew about 2x faster than insertion or selection sort. For array

based, selection sort grew about 2x faster than insertion or bubble sort. In the end, merge sort dominated the other three sorts.


List Sort
Run time in Microseconds
450000
400000
350000
300000
250000
200000
150000
100000
50000
0
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
LENGTH OF LIST
Selection
Bubble
Merge
Insertion
Array Sort
RUN TIME IN MICROSECONDS
600000
500000
400000
300000
200000
100000
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
LENGTH OF ARRAY
Selection
Bubble
Insertion
Merge

Time Complexity

Time complexity is analyzed for each sort. The main method will take the highest time complexity of the sorts used – N^2.

Phase 3: Risk Analysis

The only input from the user is length of the random data to generate and their name. I have a try / catch to catch a non-number and a catch for any non-letters in their name. Their name also must be at least one character.

Phase 4: Verification

The application has been run through with many variations to test.

Phase 5: Coding

The code of this program is included in the zip file. The code is explained with comments and line breaks to make reading easier. A Javadoc is also made to explain the use of each method.

Phase 6: Testing

Test runs have been included at the end of this file.

Phase 7: Refining the program

Refinements can be made to the output upon the user's request.

Phase 8: Production

A zip file including the source files from eclipse and the output of the program have been included.

Phase 9: Maintenance

Changes can be made once feedback is received.

Program Output

“sdb(erlj)((djre)lejr)”
Queue is balanced
Time for Queue - 67 micro s
Stack is balanced
Time for Stack - 48 micro s

“4+(5*4))”
Queue Unbalanced
Time for Queue - 42 micro s
Stack Unbalanced
Time for Stack - 43 micro s

3
Queue is balanced
Time for Queue - 27 micro s
Stack is balanced
Time for Stack - 22 micro s

“ere)(jre”
Queue Unbalanced
Time for Queue - 41 micro s
Stack Unbalanced
Time for Stack - 146 micro s

“((())((”
Queue Unbalanced
Time for Queue - 155 micro s
Stack Unbalanced
Time for Stack - 81 micro s

)(
Queue Unbalanced
Time for Queue - 35 micro s
Stack Unbalanced
Time for Stack - 20 micro s

Average Queue - 56.285714 micro s
Average Stack - 55.571429 micro s