CS 3310 – Data and File Structures

Instructor: Dr. Ajay Gupta, Western Michigan University

Lab TA: Yu Guo

Ken Rivard

**SOFTWARE LIFE CYCLE REPORT – FOR ASSIGNMENT #4 Sorting Problems**

This assignment requires us to do steps first while using a linked list but I will discuss the steps of the problem specification with the explanations for my use of linked list and array list alongside each other to give the program a better flow.

Design and Justifications

The problem specification asks us to solve a series of problems.

1. Read the data in *NamesList.txt* file and store the names in a linked list and array list.
   1. First, for my linked list I used Java’s Scanner to read in the file and stored the contents of the file into a linked list using a while loop. As for storing the contents of the file into the array list, I did the same exact thing, I used Java’s Scanner and a while loop until the end of the file was reached.
2. Sort the data in the linked-list with the 4 types of sorting methods (bubble sort, selection sort, merge sort, binary-insertion sort (i.e., insertion sort that uses binary search to find the insert position).
3. Besides string type, please let your application handle integer, float, double, char, in case we may need to use those in the future.
   1. First I created my own implementation of a linked-list in order to make analyzing the time and space complexities easier. I reused my Node class and LinkedList class from previous assignments in order to ensure that everything was working properly since previous assignments were working properly. I also carried over the use of Generics from Assignment 2 into my Node and LinkedList class.
   2. As for sorting, I implemented all of my linked list sorting methods within the LinkedList class for ease of access and use.
      1. mergeSort() and merge() to sort the list using merge sort
      2. selectionSort() to sort the list using selection sort
      3. insertionSort() to sort the list using binary-insertion sort
      4. bubbleSort() to sort the list using bubble sort
   3. for sorting the of the array list, I used the built-in array list library in Java
      1. merge sort I created a separate class title ArrayListMergeSort.java because my implementation of merge sort did not work when declaring it in this Main.java class.
      2. Binary-Insertion sort was the same as merge sort in the way that I needed to create a separate class titled ArrayListBinaryInsertSort.java.
      3. As for bubble sorting the array list I was able to implement that in the Main.java class directly
      4. For the selection sort of the array list I was able to implement that in the Main.java class as well.
   4. All sorting algorithms while using linked list and array list will work on the types integer, float, double, and char.
4. Please make your application could handle even billions of data (find out what is the maximum size your application can handle in case you cannot get billion size to run at all on your machine).
   1. I ran my program with 10,000 names in the file and the times only increased by no more than 300ms
   2. I then jumped to 500,000 names in a very large file but my computer was slowing down and eventually froze so I had to shut it down and due to time constraints I was unable to run anymore large file tests.
5. Make sure your application is robust, readable, and has a friendly UI.
   1. My program is structured to run the 4 sorting algorithms required first using the linked list and then using an array list. All sorted lists are clearly labeled before sorting and after sorting. After I sorted the lists I also included the time in *ms* in which they were sorted. At the beginning I printed out the non-sorted list to prove that the lists were not already sorted.
6. Analyze these 4 sorting methods (by time and space complexity) empirically as well as theoretically, and specify which is the best one to help YG when linked list is used to store data.

**Comparison of Theoretical v. Empirical Complexities**

Linked List Sorting Algorithms

The mergeSort() algorithm sorts the elements in the linked list in O(n logn) time in all cases.

The empirical analysis of merge sort contains 10 comparisons in the mergeSort() method and 15 comparisons in the merge() method. Since merge sort calls another method to merge the two sorted arrays together that will take n time and depending on the size of the array, the comparisons made will be n as well. From this we are able to conclude that merge sort will run in O(n logn) time.

**Linked List Merge Sort Algorithm**

public Node mergeSort(Node headOrg)

if (headOrg == null || headOrg.next == null)

return headOrg;

Node a = headOrg;

Node b = headOrg.next;

while ((b != null && b.next != null)) {

headOrg = headOrg.next;

b = (b.next).next;

}//end while

b = headOrg.next;

headOrg.next = null;

return merge(mergeSort(a), mergeSort(b));

}//end mergeSort

public Node merge(Node a, Node b) {

Node temp = new Node(a);

Node head = temp;

Node c = head;

while ((a != null) && (b != null)) {

if (a.data.compareTo(b.data) < 1) {

c.next = a;

c = a;

a = a.next;

} else {

c.next = b;

c = b;

b = b.next;

}//end else

}//end while

c.next = (a == null) ? b : a;

return head.next;

}//end merge

The selection sort algorithm sorts the elements in the linked list in O(n^2) time in all cases.

The empirical analysis of the selection sort algorithm contains 16 comparisons across two for loops. Since 2 for loops are needed that is two different instantiations of n time, therefore leading to a complexity of O(n^2).

**Linked List Selection Sort Algorithm**

public void selectionSort(Node head) {

for (Node index = head; ((index != null) && (index.getLinkNext() != null)); index = index.getLinkNext()) {

Node min = index;

for (Node test = min.getLinkNext(); test != null; test = test.getLinkNext()) {

if (test.getData().compareTo(min.getData()) < 1) {

min = test;

}//end if within for loop

}//end for

if(index != min) {

Node temp = new Node(min);

temp = index;

index = min;

min = temp;

}//end if

}//end for loop

}//end selectionSort

The Binary-Insertion sort algorithm sorts the elements in the linked list in O(n logn) time with the worst case being O(n^2)

The empirical analysis of the binary-insertion sort contains 17 comparisons within the while loop. Since we have a while loop running the best case is binary-insertion sort will run in O(n) time but runs an average and worst case of O(n^2).

**Linked List Binary Insertion Sort Algorithm**

public void insertionSort(Node node) {

//declare first node the start of sorted list

Node sortedList = node;

node = node.next;

sortedList.next = null;

while(node != null) {

final Node current = node;

node = node.next;

//find current

if (current.data.compareTo(sortedList.data) < 1) {

//current is now the sorted head

current.next = sortedList;

sortedList = current;

} else {

//search for correct positions

Node search = sortedList;

while (search.next != null && (current.data.compareTo(search.next.data)) > 0) {

search = search.next;

}//end while search

current.next = search.next;

search.next = current;

}//end else

}//end while node

}//end insertionSort

The bubble sort algorithm sorts the elements in the linked list in O(n^2) time in all cases.

The empirical analysis of bubble sort contains two for loops and a total of 15 comparisons within those for loops. Because of the nested for loops the bubble sort algorithm will run in O(n^2) time in the best, average, and worst cases.

**Linked List Bubble Sort**

public void bubbleSort() {

if (size > 1) {

for(int i = 0; i < size; i++) {

Node currentNode = head;

Node next = head.getLinkNext();

for(int j = 0; j < size - 1; j++) {

if(currentNode.getData().compareTo(next.getData()) > 0) {

Node temp = currentNode;

currentNode.data = next.data;

next.data = temp;

}//end if

currentNode = next;

next = next.getLinkNext();

}//end 2nd for

}//end first for

}//end if

}//end bubbleSortAlgorithm

Array List Sorting Algorithms

The merge sort algorithm sorts the elements in the array list in O(n logn) time in all cases.

The mergeSort() method contains 10 comparisons and the merge() method contains 9 comparisons. Since mergeSort() calls method merge() that will require n time and another n time must be taken into account for the size of the array; luckily there are no loops so we conclude that our empirical analysis of merge sort is O(n logn).

**Array List Merge Sort Algorithm**

public ArrayList<String> mergeSort(ArrayList<String> whole) {

ArrayList<String> left = new ArrayList<String>();

ArrayList<String> right = new ArrayList<String>();

int center;

if(whole.size() == 1) {

return whole;

} else {

center = whole.size() / 2;

//copy left half

for(int i = 0; i < center; i++) {

left.add(whole.get(i));

}//end for

//copt the right half

for(int i = center; i < whole.size(); i++) {

right.add(whole.get(i));

}//end for

//sort the left and right halves

left = mergeSort(left);

right = mergeSort(right);

//merge results together

merge(left, right, whole);

}//end else

return whole;

}//end mergeSort()

private void merge(ArrayList<String> left, ArrayList<String> right, ArrayList<String> whole) {

int leftIndex = 0;

int rightIndex = 0;

int wholeIndex = 0;

while(leftIndex < left.size() && rightIndex < right.size()) {

if(left.get(leftIndex).compareTo(right.get(rightIndex)) < 0) {

whole.set(wholeIndex, left.get(leftIndex));

leftIndex++;

} else {

whole.set(wholeIndex, right.get(rightIndex));

rightIndex++;

}//end else

wholeIndex++;

}//end while

ArrayList<String> rest;

int restIndex;

if(leftIndex >= left.size()) {

rest = right;

restIndex = rightIndex;

} else {

rest = left;

restIndex = leftIndex;

}

for(int i = restIndex; i < rest.size(); i++) {

whole.set(wholeIndex, rest.get(i));

wholeIndex++;

}//end for

The selection sort algorithm sorts the elements in the array list in O(n^2) time in all cases.

In selection sort, 13 comparisons are made across 2 for loops. And again, since we have 2 separate for loops that will require n^2 space which leads to the conclusion that selection sort will take O(n^2) time.

**Array List Selection Sort Algorithm**

for(selI = 0; selI < nameArrList.size(); selI++) {

smallest = nameArrList.get(selI);

smallestIndex = selI;

for(selJ = selI + 1; selJ < nameArrList.size(); selJ++) {

if(smallest.compareTo(nameArrList.get(selJ)) > 0) {

smallest = nameArrList.get(selJ);

smallestIndex = selJ;

}

}

temp = nameArrList.get(selI);

nameArrList.set(selI, nameArrList.get(smallestIndex));

nameArrList.set(smallestIndex, temp);

System.out.println(nameArrList.get(selI));

}//end for

The Binary-Insertion sort algorithm sorts the elements in the array list in O(n logn) time with O(n^2) being the worst case.

The binInsSearch() method contains 8 comparisons and one while loop. The binInsSort() method contains 5 comparisons and one for loop. Since we have one for loop we are able to conclude that our empirical analysis of Binary-Insertion sort runs in O(n) time in the best case and O(n^2) time in the average and worst case.

**Array List Binary-Insertion Sort Algorithm**

public int binInsSearch(ArrayList<String> nameArrList, int left, int right, String txt) {

int middle = -1;

while(left <= right) {

middle = (right + left) / 2;

if(txt.compareTo(nameArrList.get(middle)) == 0) {

return middle;

}

if(txt.compareTo(nameArrList.get(middle)) < 0) {

right = middle - 1;

} else {

left = middle + 1;

}

}

return middle;

}//end binInsSearch

public void binInsSort(ArrayList<String> nameArrList, int left, int right) {

for(int i = left + 1; i < nameArrList.size() - 1; i++) {

String txt = nameArrList.remove(i);

int index = binInsSearch(nameArrList, left, i, txt);

nameArrList.add(index, txt);

//System.out.println("LIST: " + nameArrList.toString());

}//end for

The bubble sort algorithm sorts the elements in the array list in O(n^2) time in all cases.

The empirical analysis of bubble sort contains 9 comparisons across 2 for loops. As for all cases with 2 for loops, that will require n^2 time and from that we are able to conclude that bubble sort runs in O(n^2) time in all cases.

**Array List Bubble Sort Algorithm**

for(int bubI = 0; bubI < nameArrList.size(); bubI++) {

for(int bubJ = bubI + 1; bubJ < nameArrList.size(); bubJ++)

if (nameArrList.get(bubI).compareTo(nameArrList.get(bubI+1)) > 0) {

String t = nameArrList.get(bubI);

nameArrList.set(bubI, nameArrList.get(bubJ));

nameArrList.set(bubI+1, t);

}//end if

System.out.println(nameArrList.get(bubI));

}//end for

**Sorting Algorithm Runtime**

Computer Specs

* Lenovo ThinkPad T440s
* Intel Core i7-4600U CPU @ 2.10GHx x 4
* Ubuntu 14.04 LTS – 64bit
* Intel Haswell Mobile Graphics
* 12GB Memory

*\*\*All times were measured using System.currentTimeMillis() from Java\*\**

Linked List Sorting Algorithms Time

* Merge Sort: 1ms
* Selection Sort: 2ms
* Binary-Insertion Sort: 3ms
* Bubble Sort: 2ms

Array List Sorting Algorithms Time

* Merge Sort: 1ms
* Selection Sort: 1ms
* Binary-Insertion Sort: 0ms
* Bubble Sort: 1ms