Sorting and Searching Assignment 1

Yacob Ben Youb en Youssef Ben Youb
500672040 - 500775494

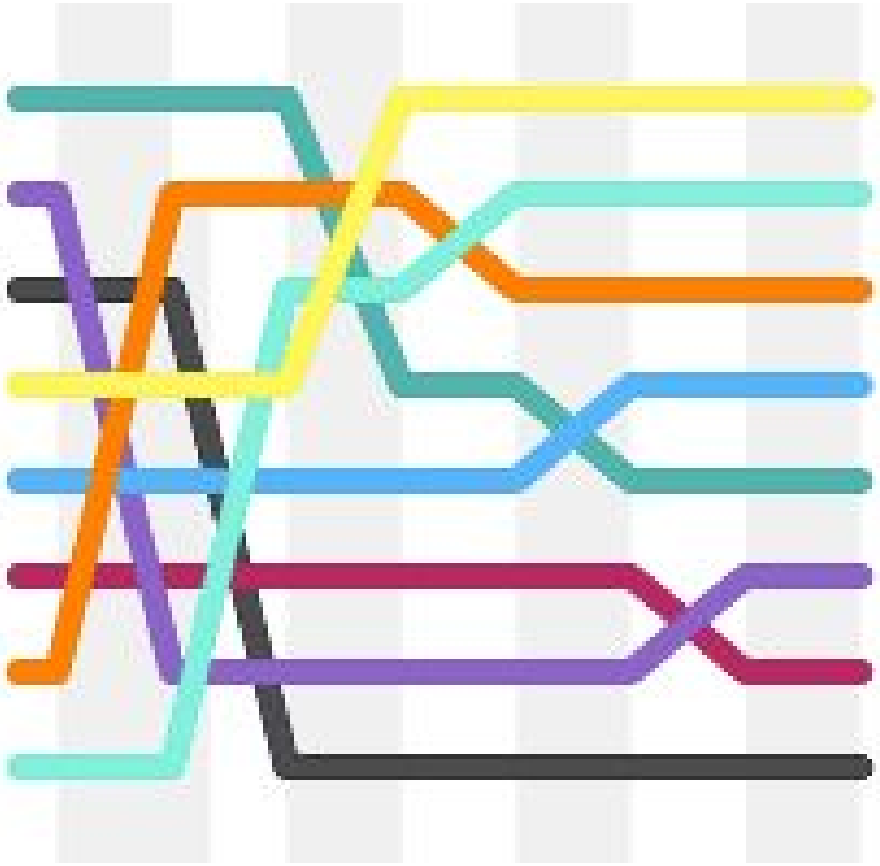


Table of contents:

Introduction

This report will discuss the methods used for implementing a quicksort for an array and a linked list and what the big O notations for these quicksorts are.

Classes

Player

In this class the method compareTo and compareHighScore have been implemented. The compareHighScore method is used by the compareTo method. The compareTo method is used by the sorts in the quicksort classes. It sorts by ascending order. This class also holds the player object which the quick sorts sort.

ArrayQuickSort

This class has the methods to quicksort an array
The methods to implement a quicksort for the array are present in this class.
These methods are:
-sort(E[])
-quickSort(E[], int, int)
-private exchangeNumbers(E[], int, int)

LinkedListQuickSort

This class is used to quicksort a linked list. It makes use of the compareTo method from the player class. Besides that the quicksort all happens in one method:
sortLinkedList(Linked List <E>)

Methods()

compareTo()

This method is from the player class. We have written a custom method which can tell us which of two players is bigger. If the player is bigger than the player it's being compared to, it shall output the number 1. If it's equal, it's 0. If it's smaller, it's -1.

```
@Override
public int compareTo(Player other) {
  if (compareHighScore(other.getHighScore()) != 0) {
     return compareHighScore(other.getHighScore());
```

If the highscores are not equal to each other it will sort by highscore as that is the highest priority. It does this by using the compareHighsScore method.

```
  } else if (lastName.compareTo(other.getLastName()) != 0) {
     return lastName.compareTo(other.getLastName());
```

The second priority to sort by is last name. If the last names are not the same it will compare them using the java String compareTo method.

```
  } else {
     return firstName.compareTo(other.getFirstName());
  }
```

The last priority is sorting by first names. This also makes use of the java String compareTo method.

```
}
```

CompareHighScore

Because java does not have a built in comparator for long's it was necessary to make a method to compare longs. This sorts by ascending order just like the java compareTo methods.

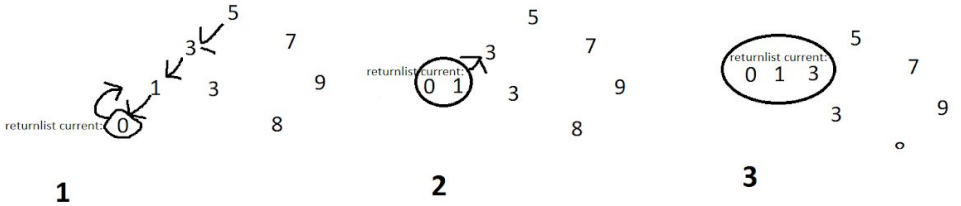
```
public int compareHighScore(long otherHighscore) {
  if (highScore < otherHighscore) {
     return -1;
  } else if (highScore > otherHighscore) {
     return 1;
  }

  return 0;
}
```

sortLinkedList()

The sortLinkedList method does not make use of another method to check if the size is 0 as it does this itself. The sortLinkedList method sorts by dividing the linked list into 2 linked lists if its size is not 0. It does this recursively. On one side it puts the elements which are lower than the pivot and on the other side the ones higher.. Once there is only one element left it will stop calling recursively but still add that element. This way all the lowest elements will be added from low to high eventually sorting the list.

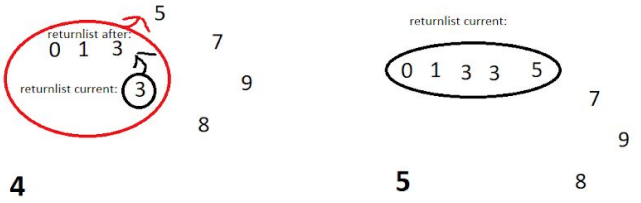
Here is an image further explaining this concept:



1 **2** **3**

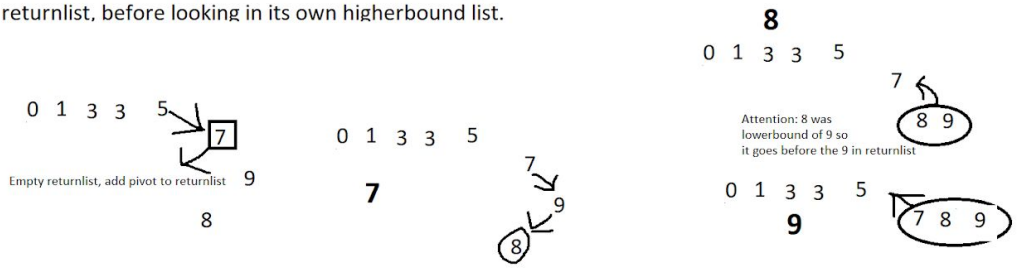
For every list: If it has a lowerbound list, go to the lowerbound list. Do this again for the lowerboundlists until it has no lowerboundlists left. Then append those to your lowerbound list. The lowerbound however will take from their lowerbounds first before returning to their original list.

If an item has no lowerbounds to look in, return that item. The list above it will append it to his own returnlist.

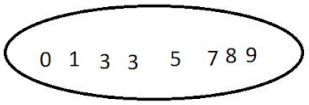


4 **5**

AFTER appending the entire lowerbound list and appending the pivot to our list, sort our higherbound in the same way. Higherbound will look for numbers in its own lowerbound and put it before his own pivot in his returnlist, before looking in its own higherbound list.



6 **7** **8** **9**



10

Now lowerbound.size and higherbound.size are 0, the methods have ended. and the original returnlist will be returned.

```
public static <E extends Comparable<E>> LinkedList<E> sortLinkedList(LinkedList<E>
toSortList) {
    if (toSortList == null ||toSortList.size() == 0) {
        return toSortList;
    }
```
If the list is empty it returns the list as is.
```
    if (toSortList.size() == 1) {
        return (toSortList);
```
If the list only has one element it is already sorted so it is returned as is.
```
    } else {

        LinkedList returnList = new LinkedList();



        E pivot = toSortList.getFirst();
```
Selecting the first element as pivot.
```
        LinkedList<E> lowerBound = new LinkedList<>();
```
Creating a linked list in which the elements lower than the pivot are put in.
```
        LinkedList<E> higherBound = new LinkedList<>();
```
Creating a linked list in which the elements higher than the pivot are put in.


```
        while (toSortList.size() != 1) {
            E listObject = toSortList.pollLast();
```
As long as the list has a size of higher than 1 it stores the value of its last element in an object and deletes the object. It then compares this object to the pivot and if it is smaller it is put in the lowerbound list. Otherwise it is put into the higherbound list.
```
            if (pivot.compareTo(listObject) >= 0) { //Last element is smaller or equal to first
element and must go left
                lowerBound.add(listObject);

            }
            else {

                higherBound.add(listObject);
            }
        }

        lowerBound = sortLinkedList(lowerBound);
```
As this method is called recursively it will eventually be set equal to the last lowerbound. That is the lowest element in the lowerbound list. This one is not added by the pivot
```
        while (lowerBound.size()!=0){

            returnList.add(lowerBound.pollFirst());
```

```
        }


        returnList.add(pivot);
```
Adds the pivot because it is otherwise not added.
```
        higherBound = sortLinkedList(higherBound);
```
Sorts the higherbound list by continuously adding the lowest element from its lowerbound list.
```
        while (higherBound.size()>0){
           returnList.add(higherBound.pollFirst());
        }

        return returnList;
    }
  }
```

Array sorting:

Sort()

This method is from the ArrayQuickSort class.
This method makes sure that the array to be sorted has at least one element in it. Otherwise it would cause problems for the quicksort method.

```
public <E extends Comparable<E>> E[] sort(E[] inputArr) {
```
It expects input of an array which holds E.
```
  if (inputArr == null || inputArr.length == 0) {
     return inputArr;
```
If its length is equal to 0 it will return the array as is without calling the quicksort method.
```
  }
  inputArr = quickSort(inputArr, 0, inputArr.length - 1);
```
If the length is not equal to 0 it wil set inputArr to the same value as the inputArr after the quicksort.
```
  return inputArr;
}
```

ExchangeNumbers()

This method is from the ArrayQuickSort class.
The ExchangeNumbers method swaps the places of 2 items in an array.

```
private <E extends Comparable<E>> E[] exchangeNumbers(E[] inputArray, int lowerIndex,
int higherIndex) {

  E temp = inputArray[lowerIndex];
```
First it creates an object in which the lowerindex value is stored.
```
  inputArray[lowerIndex] = inputArray[higherIndex];
```
Then it sets the lowerIndex value to that of the higherIndex value
```
  inputArray[higherIndex] = temp;
```
Afterwards it sets the higherIndex value to that of the object in which the lowerIndex value was stored.
```
  return inputArray;
}
```

Quicksort()

This is the method the sort() method uses the sort() method uses to do the actual sorting. It sorts by dividing the array in 2: 1 array larger than the pivot and 1 array lower than the pivot. It does this until there is only 1 element left in the array. After this it combines the arrays back.

```
int i = lowerIndex;
  int j = higherIndex;
```
The middle number in the array is selected as pivot.
```
  E pivot = inputArray[lowerIndex + (higherIndex - lowerIndex) / 2];
```

As long as i is not higher than j it will
```
  while (i <= j) {

     while (inputArray[i].compareTo( pivot) == -1) {
        i++;

     }
     while (inputArray[j].compareTo(pivot) == 1) {
        j--;
     }
     if (i <= j) {
        inputArray = exchangeNumbers(inputArray, i, j);
        //move index to next position on both sides
        i++;
        j--;
     }
  }


  if (lowerIndex < j)
     quickSort(inputArray, lowerIndex, j);
```

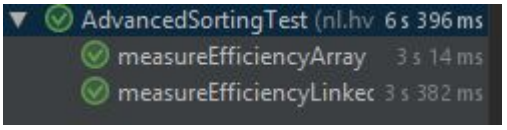
If the there is still more than one indexnumber in the array it will be quicksorted.

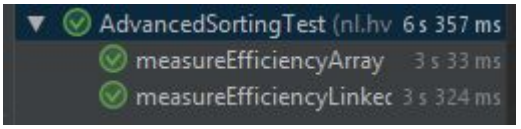
```
  if (i < higherIndex)
      quickSort(inputArray, i, higherIndex);
  return inputArray;
}
```

Efficiency

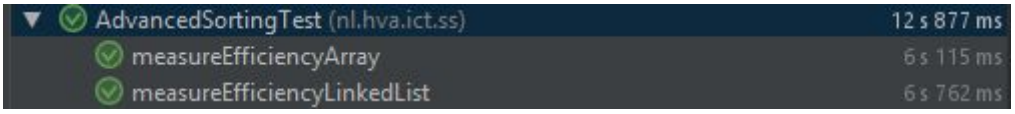
In this chapter we will be determining the efficiency of the quicksort of array and linked list. This is the efficiency of the array and linked list quicksort in practice:

500 thousand records:

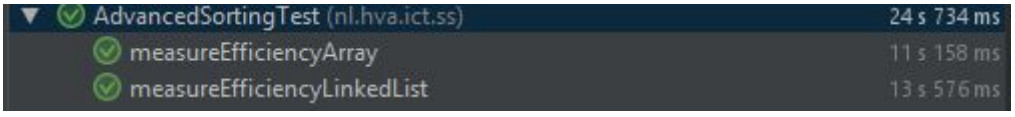


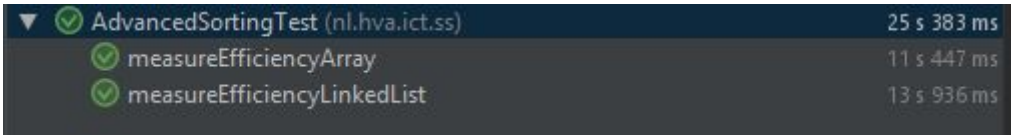


1 million records:



2 million records:





Calculating the correlation between amount of records and time for array quicksort:
Array:
500000 ÷ 3140 = 159
500000 ÷ 3330= 150

1000000 ÷ 6140 =162

1000000 ÷ 6115 = 164

2000000 ÷ 11158 = 179
2000000 ÷ 11447= 175

The efficiency appears to O= N log N looking at these results: The correlation is almost linear, however it still increases somewhat. This is a characteristic of N log N:

Log N becomes increasingly less influential the higher the amount of records. This means Log N takes up less time in correlation to the amount of records. Because this takes less time the amount of records are divided by a lower number causing the result to be higher. This is proof that the efficiency is likely N log N, however it can be argued that not enough tests have been performed to conclude this.

Calculating the correlation between amount of records and time for linked list quicksort:

500000 ÷ 3382 = 148
500000 ÷ 3324 = 150

1000000 ÷ 6723 = 149
1000000 ÷ 6762 = 148

2000000 ÷ 13576 = 147
2000000 ÷ 13936 = 144

The efficiency of the linked list quicksort is likely N √N:

N √N has the same characteristic as N log N, however √N does not decrease quite as quickly.
The results show that the linked list takes more time to sort than the array which backs up this hypothesis.

Comparison

In this chapter we will be comparing the qualities of the linked list quicksort and the array quicksort.

The quality properties i find most important are:
-Amount of memory required
-speed

In the aspect of amount of memory required the linked list quicksort wins:
The linked list creates a new node everytime a new object is added and that's it. that is optimal memory utilization. The array quicksort on the other hand creates a new array with double that size, pastes everything into that array and then deletes the old array. When both arrays are existent a lot of memory is being used and even when there is only one array existent, there is memory being used for slots in the array which don’t contain anything.

In the aspect of speed the array quicksort wins. It simply completes the quicksort process faster and is thus faster. Although it may not be by a large margin, if the user has enough space anyways, it is always good to have more speed over memory.

Conclusion

Linked list quicksort takes longer than the array quicksort. However, array quicksort takes up more space than linked list quicksort. The user should determine for themselves which one they will use for their application.

Thought experiment

In this chapter we will be discussing our thoughts on what would change if we used array lists instead of linked lists for a quicksort.

If linked lists would be replaced with arraylists it would take up a bit more memory, however, it would likely be even faster than the quicksort for the normal array as it does not have to create a new array if it is full.

Possible improvements

In this chapter we will be discussing potential improvements for the quicksorts on array,arraylist,linkedlist

The median of three method would most likely be a good improvement for an arraylist quicksort and array quicksort. This is because the first pivot would be better placed making it so that the list will immediately be better divided in 2 and thus causing less need for more divisions. This would also remove any need for shuffling the list before sorting it.

The median of sample would probably be a good improvement for the linked list. Just like with the median of three method mentioned before, it would make for a better pivot and thus making the sorting process faster.

View on ADT’s

This assignment has brought into view the importance of selecting what data type to use for your application. Besides that it also brought into view how different the code for linked lists and arrays can be when it comes to sorting.