

CSE 222 HOMEWORK #8

Multiple Heap

Orhan Aksoy

09104302

TABLE OF CONTENTS

1. PROBLEM	3
1.1. REQUIREMENTS	3
2. ANALYSIS.....	3
3. DESIGN.....	5
4. IMPLEMENTATION	6
4.1. CREATION OF THE MULTIPLE HEAP	6
4.2. ADDITION	7
4.3. FINDFIRST OPERATION.....	7
4.4. GETFIRST OPERATION	8
4.5. REMOVEAT METHOD	8
5. TESTING.....	8
5.1. THE TEST CASES	9
5.2. THE PROGRAM OUTPUT.....	10

LIST OF FIGURES

Figure 1 Multiple Heap Solution	5
Figure 2 Multiple Heap – Typical Memory Organization	6
Figure 3 The Program Output.....	10

1. Problem

Heap data structure provides an efficient way for finding and removing of the smallest/largest element in a collection with respect to a single attribute. However, there may be some situations in which the finding/removing function should operate with respect to various attributes within the same collection.

A new data structure will be implemented in order to fulfill the requirement described above. With this data structure, finding and removing operations will be possible with respect to various attributes within the same collection at the same time.

1.1. Requirements

- 1.1.1. The data structure shall be a generic collection to store user defined objects.
- 1.1.2. It shall allow users to:
 - a. add an item to the collection
 - b. peek/remove the biggest/smallest item in the collection
 - c. get the number of items in the collection
- 1.1.3. It shall accept multiple ordering criteria that it will use for comparing the items in its collection in order fulfill the requirement described in 1.1.2 – b. This will make the collection ordered with respect to more than one criteria at the same time.
- 1.1.4. All of the ordering criteria shall be set during the creation time of the heap, and will not be changed afterwards.
- 1.1.5. The users shall be able to select an ordering criteria during peeking/removing the biggest/smallest element in the collection as described in 1.1.2 – b. These methods shall return the biggest/smallest element according to this criteria.
- 1.1.6. All item finding/addition/removals shall be $O(\log n)$ time.

2. Analysis

In a regular heap, the data collection is stored in a specific ordering so that the retrieval of the biggest/smallest item takes constant time. Addition and removal to/from the heap takes $O(\log(n))$ time. The algorithm that runs during insertion and deletion operations for this capability requires comparison of items with respect to their relative sizes – that is, determining whichever one is bigger/smaller.

In the multiple heap case, the requirements described in section 1.1 drives two changes to the original heap data structure:

1. During the creation of the multiple heap, the users will provide a list of ordering criteria among which they will select a specific one during finding/removal operations afterwards.
2. The users of the multiple heap will still expect the finding/removal operations to take constant time no matter which ordering criteria they select during these operations. So, the specific ordering of data items emphasized above should be done for all criteria at the same time during insertion/deletion operations.

A solution to this problem is to create a regular heap per comparison function. During additions, a new item is inserted into each of the heaps using a unique comparison function for that heap.

Obtaining the first item with respect to a comparison function will become selection of the root of the heap identified by the comparison function.

The removal of an item is not straightforward in the multiple heap case. The removal operation in a heap is typically getting and removing of the root of the heap. Although this operation is trivial for the provided comparison function – that is, the removal of the root -, the same item is supposed to be removed from all of the heaps as well. In this solution, this operation is implemented, where removal of any node in any heap updates all of the heaps accordingly.

3. Design

The class diagram of the solution is shown below:

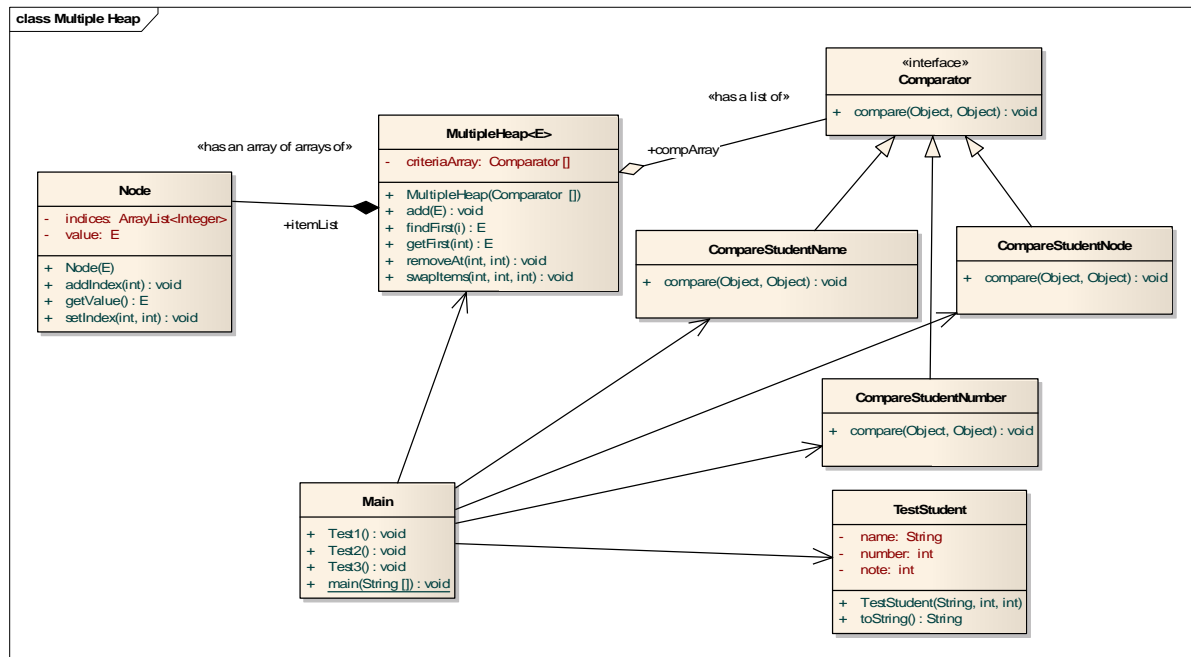


Figure 1 Multiple Heap Solution

The generic class **MultipleHeap** implements the functions described in the requirements. Its constructor receives an array of **Comparator** references. For each of these references, a separate *ArrayList* is created representing the storage area for a new heap. This new *ArrayList* is stored in another *ArrayList* named *itemList* that represents the list of storage areas of the heaps. That is:

`itemList : ArrayList < ArrayList < Node > >`

The *itemList* is a list of heaps, so, the *ArrayList<Node>* is the storage area of a heap. **Node** contains the item that is being stored and also a list of indices into the heaps.

Assume that there are 3 comparator references. This means that there are 3 heaps in the *itemList*. Every time a new item is added to the multiple heap, this new item is added to each of these 3 heaps. However, the added items are NOT separate **Node** objects. Instead, the node object is created for the first heap and this same reference is used for the rest of the heaps. This way, a modification on a node in any heap is automatically reflected to all of the heaps.

The memory organization of a typical case is shown below:

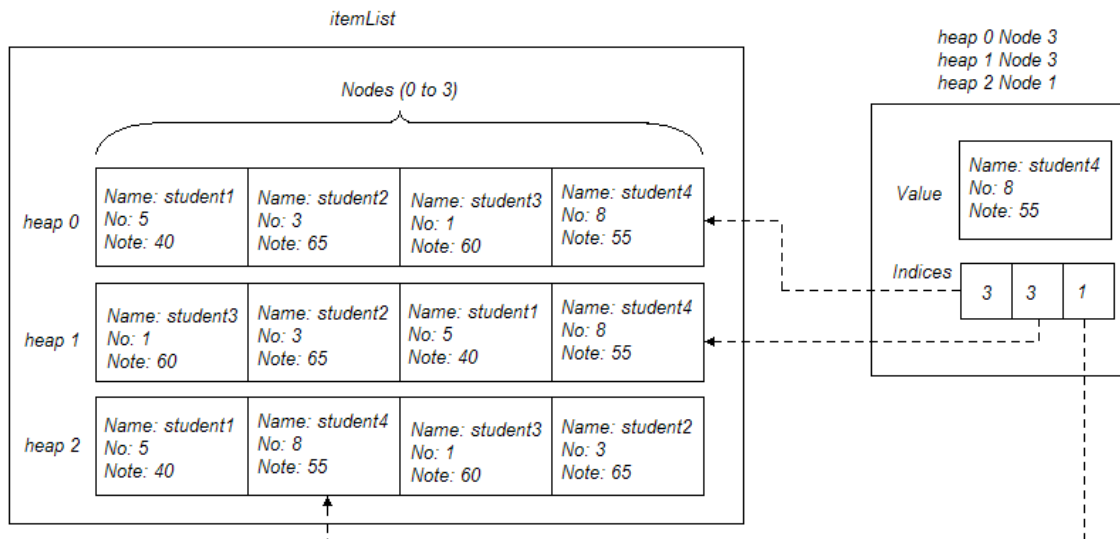


Figure 2 Multiple Heap – Typical Memory Organization

In the figure above, there are three comparator references: The first one (used in heap 0) is with respect to student name, the second one is with respect to student number and the last one with respect to the note.

The addition operation makes three addition operations in three separate heaps. The removal of the root in any heap is followed by the removal of the corresponding nodes in the other heaps using the indices stored within the node. This requires the capability of the removal of an item at any position in the heaps. Querying the multiple heap using a criteria simply returns the root item in the corresponding heap identified by the order of the comparator reference within the comparator array.

4. Implementation

4.1. Creation of the Multiple Heap

During the creation of the multiple heap, an array of Comparator references is presented to the constructor. The constructor definition is:

```
MultipleHeap(Comparator [] compArray) {
```

Each of the Comparator references inside the compArray is supposed to compare two items in a different manner. An example could be the MultipleHeap<Integer>. IN this case, the first comparator could return -1 if the first parameter is less than the second. The second comparator could do the inverse: that is, it could return -1 if the second parameter is less than the first. This way, two references in the compArray would result in two heaps in the multiple heap. The first one would return the smallest integer, and the second one the biggest.

So, the constructor simply creates as many heaps in the itemList as there are comparator references in the compArray.

4.2. Addition

During the addition the following algorithm is implemented:

1. For the first heap, a new node object is created and the item is stored within this node.
2. This node is pushed back of the first heap.
3. All of the indices inside the index array within the node is reset to point the last item in the heap.
4. The node reference created in (1) is pushed back of the rest of the heaps. Note that the node reference, and thus the element and index arrays are same in all of the heaps.
5. For each of the heaps the following steps are taken:
 - a. Using the comparator reference identified by the heap (heap 0 uses comparator 0, heap 1 uses comparator 1, etc.), the last item in the heap is compared with its parent. If it is smaller, it is swapped with its parent.
 - b. (a) is repeated until the parent is smaller.
 - c. During the swap operations, the index array is updated so that the new position of the node within the heap is set in the relative index position in the indices array.

After four additions of student objects in a multiple heap created with 3 comparator references, a typical memory state can be seen in Figure 2.

If there are m comparator references in the compArray explained in section 4.1, then the time complexity of the addition of an item into a multiple heap with n items in it is m times $O(\log(n))$ which is $O(m \log(n))$.

4.3. findFirst operation

The findFirst method definition is:

```
public E findFirst(int criteriaIndex) {
```

The findFirst operation requires an index into the comparator references that were supplied in the constructor. If there are 4 comparator references in the array provided during construction of the multiple heap, then this criteriaIndex can have the values from 0 to 3.

This function simply returns the root item in the heap identified by the provided criteriaIndex.

So, the time complexity of this operation is constant.

4.4. *getFirst* operation

The return value of the *getFirst* operation is the same as *findFirst* operation described in section 4.3. The only difference of this operation is the removal of the returned item.

The removal operation in the multiple heap requires one removal operation per heap. For the heap identified by the *criteriaIndex*, the item to be removed is the root. This operation is exactly same as the regular heap operation. The root is removed, and this empty position is filled with the last item in the heap. Then, this item at the root is replaced with its smaller child consecutively until both children are bigger.

However, once this item is removed in the heap identified by the *criteriaIndex*, the array cells pointing to this node in the other heaps are now invalid. They are supposed to be removed as well. For this purpose, a new method named '*removeAt*' is introduced in the multiple heap. This method accepts an integer representing the criteria index and another integer representing the item position in the corresponding heap array. Using the indices array of the deleted node, the invalid nodes in all of the heap arrays are removed using this new function.

The time complexity of this method is also $O(\log(n))$. Because the *getFirst* operation gets the root of a specific heap (constant time), then for all heaps, moves an item up to the root (number of criteria times $\log(n)$) then replaces it with the last item (constant time) and then moves the root down to its position ($O(\log(n))$). Assuming there are m criteria, then the total time is $\text{constant} + \log(n) + (m-1)*2*\log(n)*\text{constant} = O(\log(n))$. (The removal procedure is described in the following section).

4.5. *removeAt* method

This method accepts two parameters: An index into the criteria array, and another index into the array representing the heap storage. This method removes an item in a heap at any position. TO do it, the following algorithm is implemented:

1. The node to be removed is swapped with its parent until this node becomes the root
2. The root is replaced with the last item in the heap array.
3. The last item is removed.
4. The root is swapped by its smaller child until both children are bigger.

During an operation that moves a node, the indices array is updatd as well.

The details of all methods in all classes can be seen in the javadoc files.

5. Testing

In order to test the application, the following sources are necessary:

The sources:

- Main.java, MultipleHeap.java, TestStudent.java

Before testing, the sources should be compiled using

```
# javac Main.java
```

To run the application, use

```
# java Main
```

5.1. The Test Cases

In order to test the MultipleHeap class, a test class named TestStudent is written. This class represents a student having three members: student name (String), student number (int) and a note (int). Its constructor accepts three parameters each of which correspond to these members.

In the TestStudent class, three inner classes are defined. They all implement the Comparator interface in a different manner. Their names are: CompareStudentName, CompareStudentNumber and CompareStudentNode. In their compare methods, they all compare two TestStudent instances according to the method described in their names.

In the main, an array of TestStudents with four members is created and provided to three test methods named Test1, Test2 and Test 3.

5.1.1. Test Case 1

This method creates a MultipleHeap object with only one comparator: CompareStudentName. Then, it feeds the four student instances into the MultipleHeap using the 'add' method, and then calls getFirst consecutively until there are no more items in the multiple heap and prints them out. In the output, the students are ordered according to the student names.

5.1.2. Test Case 2

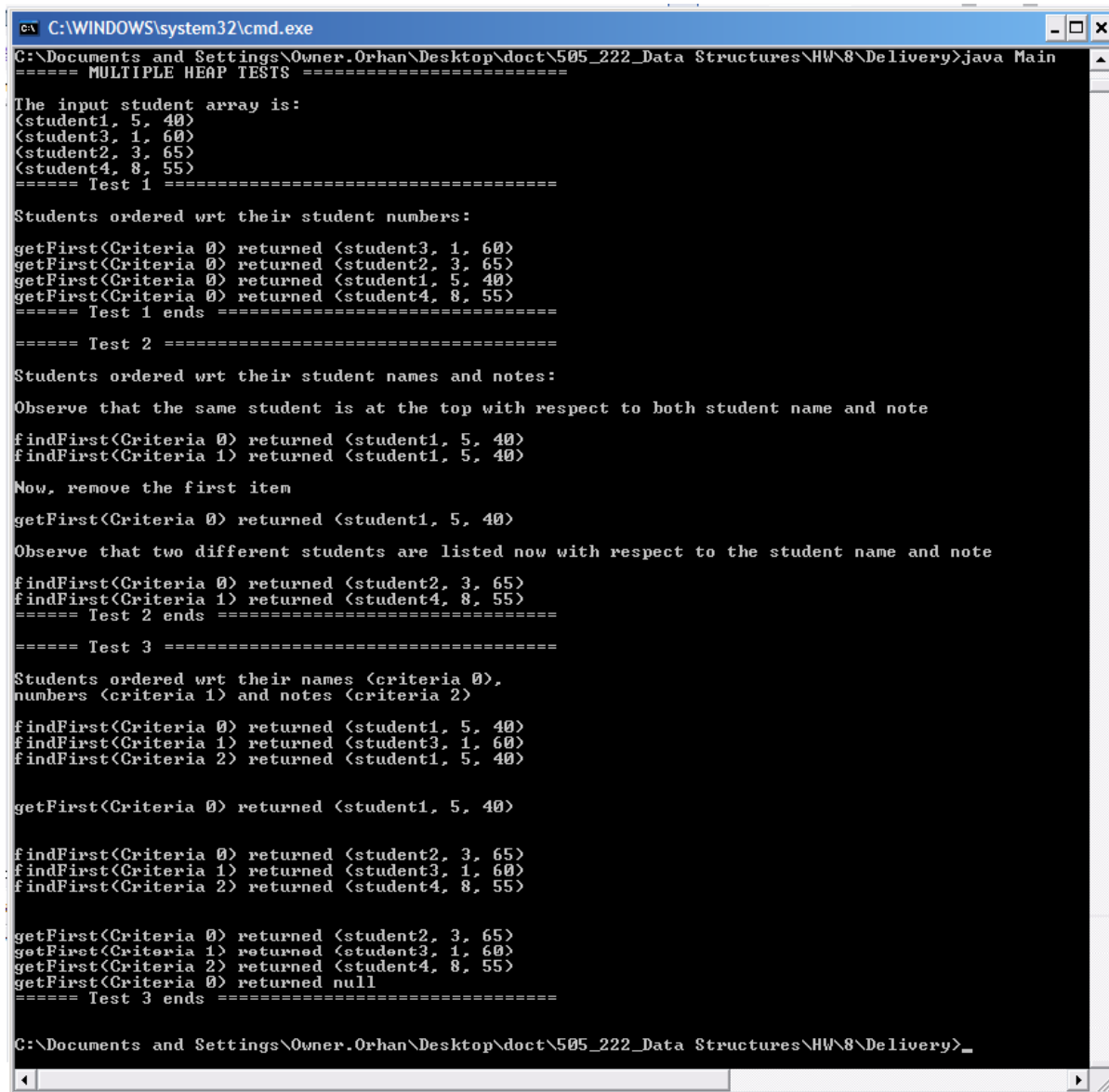
This method creates a MultipleHeap object with two comparators: CompareStudentName and CompareStudentNote. The same four student objects are added to the multiple heap. The findFirst output is shown on the output, and they are same – because, student1 is the smallest item according to student names, and 40 is the smallest note in the list, and they both belong to the first student in the array. Then a 'getFirst' is called. Then, it is observed that the

findFirst outputs using the two criteria now show two different objects after the removal of the first item.

5.1.3. Test Case 2

This method creates a MultipleHeap object with all three comparators, and various findFirst/getFirst calls are made on the heap, and the output is shown.

5.2. The Program Output



```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\Owner.Orhan\Desktop\doct\505_222_Data Structures\HW\8\Delivery>java Main
===== MULTIPLE HEAP TESTS =====

The input student array is:
<student1, 5, 40>
<student3, 1, 60>
<student2, 3, 65>
<student4, 8, 55>
===== Test 1 =====

Students ordered wrt their student numbers:
getFirst(Criteria 0) returned <student3, 1, 60>
getFirst(Criteria 0) returned <student2, 3, 65>
getFirst(Criteria 0) returned <student1, 5, 40>
getFirst(Criteria 0) returned <student4, 8, 55>
===== Test 1 ends =====

===== Test 2 =====

Students ordered wrt their student names and notes:
Observe that the same student is at the top with respect to both student name and note
findFirst(Criteria 0) returned <student1, 5, 40>
findFirst(Criteria 1) returned <student1, 5, 40>

Now, remove the first item
getFirst(Criteria 0) returned <student1, 5, 40>

Observe that two different students are listed now with respect to the student name and note
findFirst(Criteria 0) returned <student2, 3, 65>
findFirst(Criteria 1) returned <student4, 8, 55>
===== Test 2 ends =====

===== Test 3 =====

Students ordered wrt their names (criteria 0),
numbers (criteria 1) and notes (criteria 2)
findFirst(Criteria 0) returned <student1, 5, 40>
findFirst(Criteria 1) returned <student3, 1, 60>
findFirst(Criteria 2) returned <student1, 5, 40>

getFirst(Criteria 0) returned <student1, 5, 40>

findFirst(Criteria 0) returned <student2, 3, 65>
findFirst(Criteria 1) returned <student3, 1, 60>
findFirst(Criteria 2) returned <student4, 8, 55>

getFirst(Criteria 0) returned <student2, 3, 65>
getFirst(Criteria 1) returned <student3, 1, 60>
getFirst(Criteria 2) returned <student4, 8, 55>
getFirst(Criteria 0) returned null
===== Test 3 ends =====

C:\Documents and Settings\Owner.Orhan\Desktop\doct\505_222_Data Structures\HW\8\Delivery>
```

Figure 3 The Program Output