Heaps and Polling Data

CS 106 - Introduction to Data Structures

Lab 6 - one week

For this assignment, you'll continue your previous investigation of Democratic primary polling data by determining which candidate is currently in the lead.

1 Implementing a Heap

You should start by implementing the given `PriorityQueue` interface as a `LinkedHeap` so that generic objects that implement the compareTo function from the Comparable interface can be inserted into your priority queue.

Requirements:

1. Implement the PriorityQueue interface as a maximum heap. You should extend your previous implementation of the LinkedBinaryTree in LinkedHeap. As in the previous lab, you should *not* use any for or while loops in your implementation.

2. Your implementation should be properly encapsulated, i.e. no implementation details should be made visible outside of the LinkedHeap or LinkedBinaryTree classes. You *may* modify your LinkedBinaryTree implementation from the previous lab if necessary, for example, you may find it useful to make the Node class `protected` instead of private.

3. You should use the `compareTo` method of the given element to determine which values are greater or less within the priority queue. Insertion of candidate results that are already in the tree should *update* the current element in the priority queue while making any updates necessary to guarantee the heap property. When you put your polling data into the tree this will be equivalent to updating the poll numbers for a candidate. Since this heap will be used to store polling data, you should be implementing this as a *maximum* heap, so that we will be able to easily retrieve the current top candidate. You

should make sure that compareTo considers the candidates' last names when determining equality, while considering the polling results when determining whether the result is more or less than another candidate's results. Note that you will need to change your implementation of compareTo from the previous lab.

4. Your implementation of `remove` should ensure the heap property is maintained. If the given element can not be found in the heap (based on the use of the `compareTo` method), this method should return `false`.

Hints:

1. `insert`: Items should be inserted into the heap by inserting into a non-perfect subtree. You may find it useful to create a method within your node class to determine if the subtree rooted at that node is perfect. Recall that in a perfect subtree the size of the subtree $size = 2^{height+1} - 1$, thus, you may also find it useful to keep track of the size and height of each subtree. Once you determine where to insert the new node and insert it, your insert method should then call `heapifyUp` (see below) to fix the heap property.

2. `remove`: In order to remove the given element from the heap, you'll need to first find the node containing that element. You can do this by making a helper function that recursively looks at all nodes of the tree and returns the node where the element is equal to the given one. Once you have the Node object to remove, you'll find the "last" node in the heap (i.e., the one in the position that should be removed to maintain the heap shape), swap that with the node to remove, and then remove the node. Finally, use the `heapifyDown` helper method you should write (see below) to fix the heap property of the swapped node.

3. `heapifyUp`: Create a `void heapifyUp(Node node)` method that recursively swaps the given node up the tree and is called on a node after it has been inserted in the bottom level of the tree or swapped to a part of the tree where it violates the heap property.

4. `heapifyDown`: Create a `void heapifyDown(Node node)` method that recursively swaps the given node with the larger of the two children down the tree until the heap property is satisfied.

5. `swap`: You may find it useful to have a helper method:
   `private void swap(Node nodeA, Node nodeB)`
   that does the work of swapping two nodes as well as all of their associated links that can be called from both heapifyUp and heapifyDown. This method should:

(a) Store all of the relevant original information for each of the nodes to swap: the left child, right child, parent, height, and size.

(b) Set each of the above pieces of information to its swapped value for both nodes. Note that in order to do this, you'll need to have some way to identify whether a node was a left or right child of its parent. You could do this by modifying your Node class to keep track of this information (in which case you'll also need to be careful to set that information correctly within `swap`) or by checking equality using `compareTo`.

(c) Be sure to also set the relevant left child, right child, and parent pointers for the parent of the given nodes and the children of the given nodes.

2 Command Line Input

As in the previous assignment, you will take filenames that store polling data as arguments to your main method. Your resulting heap should contain the polling data for each candidate from the most recent date for which there is data from the files given on the command line. The resulting heap should be ordered so that the candidate with the highest percentage of voters in the most recent poll is at the top of the heap.

You will add an optional argument to remove some candidates from consideration. The full set of arguments you should handle will look like this:
`-r Biden Bloomberg dempres_20190210_1.csv dempres_20190210_2.csv`
but it should remain valid for the arguments to only contain files like this:
`dempres_20190210_1.csv dempres_20190210_2.csv`
In the first case above you would print out the tree and top candidate who is *not* Biden or Bloomberg based on the polling data in the given files.

Requirements:

1. Take filename input from the command line into the main method of your `Main.java`. You may be given multiple filenames. Print out the tree (in the same format as for the last lab) after each file is inserted, followed by the top candidate:

   ```
   Top Candidate:
   Joseph R. Biden Jr.:29.0
   ```

   The above printout should happen *once* after all polling data has been inserted and should *not* change the heap.

2. Process the optional $-r$ flag to remove candidates. This should be done so that the top candidate does *not* include any removed candidates. However you should only perform these remove operations once. One suggested order for handing this flag is to:

   (a) insert all the polling data (printing out the heap after each file is inserted, as in the previous lab),

   (b) remove the candidates, and

   (c) show the top candidate.

   In other words, it is expected that when you print out the heap it *will* include the candidates that will later be removed. Recall from above that your compareTo method of your polling data object should test equality based solely on the candidate's last name - this will be useful for removal of candidates.

3. As in the previous assignment, you should print out the heap after each polling file is inserted.

3 Extra Credit

All extra credit should only be done **after** successful completion of all of the base requirements for this assignment. The number of points awarded for extra credit will be smaller than those for completion of the base requirements and the extra credit is designed to be *harder* than those basic requirements as well. You may choose which of the extra credit options below to pursue and can receive credit for some and not others where that makes sense. *If you implement any of these options, identify the work that you did in the README file.*

While a heap is usually required only to return the maximum (or minimum) element, since this will be used to store polling data, it may be interesting to us to retrieve the top few candidates. Most extra credit options below relate to this idea.

1. Add a method `ArrayList<E> peekTopN(int n)` that returns the top elements of the heap in order. The heap should not be any different after the method was called than it was before the method was called, i.e., this is similar to peek in that it does *not* remove the top element. You should *not* implement this method by removing and then reinserting each element, as this has the potential to modify the heap.

   (a) Describe your design of the peekTopN method in your README file.

(b) Add an option for the users to provide a flag that will run your peekTopN method to determine and print out the top N candidates. The resulting arguments you should handle will look like this:

```
-n 5 dempres_20190210_1.csv dempres_20190210_2.csv
```

In the above case, the top 5 candidates would be displayed. This should work seamlessly with the `-r` flag such that the top 5 candidates *who have not been removed* are displayed. The display format should be the same as for the Top Candidate, but with multiple lines below.

2. Handle tied ranks appropriately for the peekTopN method. For example, in the case where there are two candidates who are tied for the best, peekTopN for $n = 1$ should print out *both* of those candidates.

3. Given a detailed big-Oh analysis for the peekTopN method in your README.

4. Implement the heap interface using an array or ArrayList in a new `ArrayHeap` file.