CMSC 420: Spring 2022

Programming Assignment 1B: Quake Heaps (Decreasing and Extracting)

Handed out: Tue, Mar 8. Due: **Thu, Mar 31, 11:59pm.**

Overview: This assignment is the continuation of Programming Assignment 1A on the Quake Heap. In Part-A, you implemented functions for insertion (and the use of locators), getting the minimum key (and merging trees), and producing a symbolic listing of the structure. In this part, we will complete the remainder of the structure by adding the operations of decrease-key, extract-min, and a few others.

Quake Heap: Please refer to Part-A of the assignment for a general description of the Quake Heap data structure (see Fig. 1. We will employ the same basic elements here. Further details available on quake heap can be found in the CMSC420 Projects page and the Quake Heap lecture notes. The operations from Part-A will be the same as before. This includes the constructor, `clear`, `insert`, `getMaxLevel`, `getMinKey`, and `listHeap`. In this part, you will implement the following additional operations:

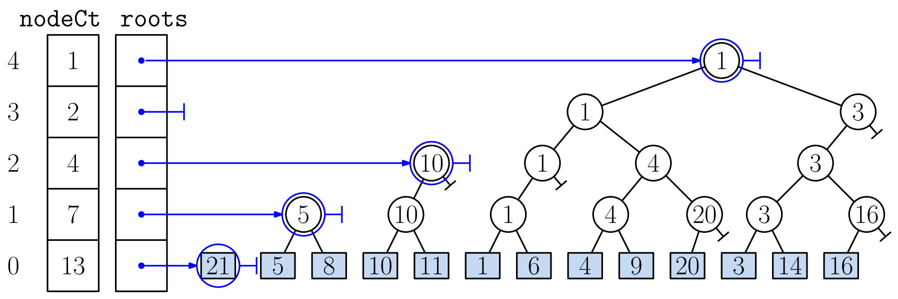

Figure 1: A quake heap storing 13 keys $\{21, 5, 8, 10, \ldots, 16\}$ (values are not shown).

`void decreaseKey(Locator r, Key newKey)`: This decreases the key of the item referenced by locator `r` to have the key value of `newKey`. If `newKey` is strictly larger than the current key for this item, an `Exception` is thrown with the message `"Invalid key for decrease-key"`. Otherwise, the decrease-key operation described in class is performed. (For testing purposes, we would like you to perform the operation even if the new key value is the same as the original key value.)

In class, we presented two methods. One was a simple method running in $O(\log n)$ time and the other was more sophisticated and ran in $O(1)$ time. You may implement the simple method for full credit, and we will leave the sophisticated method as a challenge problem for extra credit points.

The simple method starts with the leaf node `u` specified by the locator `r`. It walks up the tree, always following left-child links. For each node visited (including the initial leaf) the key value is changed to `newKey`. The process stops either when we pop off the top of the tree or when we first visit a parent along a right-child link.

Note that there may be duplicate keys in the heap. For this reason, it is not a good idea to compare keys to determine whether a node is the left child of its parent.

`Value extractMin()`: If the heap is nonempty, this function finds the entry with the smallest key value, removes this entry from the heap, and returns the associated value. This is done by the process described in the class notes, which involves first searching to find the root node with the smallest key, performing cuts along its left-most path, merging trees, and quaking. If there are ties for the minimum key, you may extract any one. If the heap is empty, an `Exception` is thrown with the message `"Empty heap"`.

The processes of searching for the smallest key and merging trees are exactly the same as in Part-A. (Remember that tree roots are to be sorted on each level before merging. This is not required by the quake heap, but it useful for testing purposes.) The processes of cutting and quaking are described in the lecture notes.

Note that unlike `getMinKey()`, this function returns the *value* associated with the minimum key, not the key itself.

`int size()`: This returns the number of entries in the heap, which is equivalent to the number of leaf nodes in all the trees. This should be answered in $O(1)$ time. The easiest way to implement this is just to maintain a counter, which is incremented whenever entries are inserted and decremented when entries are removed.

`void setQuakeRatio(double newRatio)`: This sets the quake ratio from its current value (which is initially 3/4) to the value `newRatio`. If `newRatio` is strictly smaller than 1/2 or strictly greater than 1, an `Exception` is thrown with the message `"Quake ratio is outside valid bounds"`. The current structure is not modified, but in all future instances where the quake operation is performed, this value will be used.

`void setNLevels(int nl)`: This sets the number of levels in the quake heap to `nl`. If `nl` is smaller than 1, an `Exception` is thrown with the message `"Attempt to set an invalid number of levels"`. Otherwise, the function sets the number of levels to this value, and adjusts the `roots` and `nodeCt` arrays accordingly.

If `nl` is greater than or equal to the current number of levels, these new levels are added to the structure. The structure is otherwise unchanged. (In particular, we do *not* invoke merge-trees at this time to take advantage of the fact that there are additional levels. The next time that merge-trees would have been invoked, we will take advantage of the new levels.)

On the other hand, if `nl` is smaller than the current number of levels, we remove all nodes in the tree at levels greater than or equal to `nl` (in the same manner as if we were to force a quake were triggered), and convert all the newly exposed nodes at level `nl` $- 1$ to be new root nodes. Then we reduce the number of levels to the new value.

Skeleton Code: As usual, we will provide skeleton code on the class Projects Page. The only file that you should expect to modify is `QuakeHeap.java`. Use must use the package "`cmsc420_s22`" for all your source files. (This is required for the autgrader to work.) We will provide a driver program that will input a set of commands. You need only implement the data structure and the functions listed above. Here is a portion of the class's public interface (and of course, you will add all the private data and helper functions).

```
package cmsc420_s22;
```

```
import java.util.ArrayList;

public class QuakeHeap<Key extends Comparable<Key>, Value> {

    class Node { ... }
    public class Locator { ... }

    public QuakeHeap(int nLevels) { ... }
    public void clear() { ... }
    public Locator insert(Key x, Value v) { ... }
    public Key getMinKey() throws Exception { ... }
    public int getMaxLevel(Locator r) { ... }
    public ArrayList<String> listHeap() { ... }

    // New functions

    public int size() { ... }
    public void setQuakeRatio(double newRatio) throws Exception { ... }
    public void setNLevels(int nl) throws Exception { ... }
    public Value extractMin() throws Exception { ... }
}
```

Efficiency requirements: As in Part-A, the function `insert()` should run in $O(1)$ time, the function `getMinKey()` should run in time proportional to the number of roots (plus the time needed for sorting each level), and the function `getMaxLevel()` should run in time proportional to the maximum number of levels. For Part-B, the function `size` should run in $O(1)$ time, and `decreaseKey` should run in time proportional to the number of levels.

Testing/Grading: Submissions will be made through Gradescope (you need only upload your modified `QuakeHeap.java` file). We will be using Gradescope's autograder and JUnit for testing and grading your submissions. We will provide some testing data and expected results along with the skeleton code.

The total point value is 50 points. Of these, 45 points will be purely for input/output correctness as tested by the autograder, and the remaining 5 points will be for clean programming style and the above efficiency requirements.

Challenge Problem: The simple decrease-key function described above takes time proportional to the tree height, which is $O(\log n)$ time. Using the methods described in the lecture notes, implement decrease-key to run in $O(1)$ time.

If you attempt this, add a comment in the first line of your program (or somewhere near the top) so we can check it. Please tell us where your `decreaseKey` function can be found in your source code. For example:

```
// I HAVE ATTEMPTED THE CHALLENGE PROBLEM. SEE DECREASEKEY ON LINE 327
package cmsc420_s22;
...
```