CMSC 420: Spring 2022

Programming Assignment 1A: Quake Heaps (Insertion and Merging)

Handed out: Thu, Feb 10. Due: **Thu, Feb 24, 11:59pm**.

Overview: This is the first in a two-part assignment to implement an interesting data structure called a *Quake Heap*. As with standard heaps, this data structure implements a *priority queue*. Such a data structure stores key-value pairs, where keys are from a totally ordered domain (such as integers, floats, or strings). At a minimum, a priority queue supports the operations of *insert* (add a new key-value pair) and *extract-min* (remove the entry with the smallest key, and return its associated value).

The most famous example of a heap is the *binary heap*, which is the data structure used by HeapSort. There are numerous variants, which provide improved performance for various operations, notably that of decreasing a key. The quake heap is such a variant. It was developed by Timothy Chan (described in this paper) as a simpler alternative to the Fibonacci Heap. It supports insertion and decreasing keys in $O(1)$ time, and it supports extract-min in $O(\log n)$ amortized time. (More details can be found on the CMSC420 Projects page.)

In Part-A of the assignment, we will implement only a portion of the quake heap functionality. *We will discuss the quake heap in a future lecture, but this part of the assignment is completely self-contained.* In Part-B, we will implement all the functionality.

Quake Heap: The Java class is called `QuakeHeap`. It is generic, templated by two types `Key` and `Value`. The `Key` type implements the Java `Comparable` interface, meaning that it must provide a function `compareTo()` for comparing keys. We also assume that both types support a (stable) `toString()` method.

The quake heap is represented as a collection of binary trees, where each node stores a key-value pair. The nodes of these trees are organized into *levels*. All the leaves reside on level 0, and each key in the heap is stored in exactly one leaf of some tree. (For example, in Fig. 1, the keys consist of the 13 keys in the blue leaf nodes.)

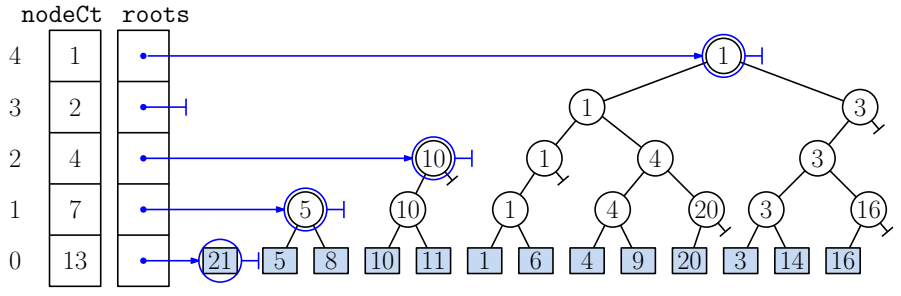

Figure 1: A quake heap storing 13 keys $\{21, 5, 8, 10, \ldots, 16\}$ (values are not shown).

Each internal node has a left child and an optional right child. Its key value is that of its left child. If the right child exists, its key value is greater than or equal to the left child. The root holds the smallest key value over all the leaves, which (by our rule that the left key is

smaller) is that of the leftmost leaf. *It follows that the smallest key in the heap will be stored in one of the roots* (but we don't generally know which).

Each node stores a key-value pair, left and right child pointers, parent pointer, and its level. We maintain two additional arrays organized by level:

- `roots[lev]`: A linked list containing references to the tree roots of level `lev`. (We recommend implementing this as a Java `LinkedList` of nodes).
- `nodeCt[lev]`: Stores the total number of nodes at level `lev`.

Nodes and Locators: The principal objects being manipulated are the nodes in the quake heap. As mentioned above, each node stores a key-value pair, left and right child links, parent link, and its level in the tree. Nodes at level 0 are leaves, so both child links are `null`. A root node (at any level) has a parent link of `null`. Java provides an elegant way to define nodes by simply nesting a class, say `Node`, inside your `QuakeHeap` class (see the code block below).

One tricky element in any heap structure that supports decrease-key is that we need a mechanism for identifying the entry whose key we wish to decrease. When we insert a key-value pair, we create a new leaf node. Since `Node` is a protected object within `QuakeHeap`, we cannot return a pointer directly to it. Instead, we create a special public object, called a `Locator`, to enclose a reference to this newly inserted leaf node. The insert function returns a locator referencing the newly created node. A skeletal example is provided below.

```
package cmsc420_s22;
public class QuakeHeap<Key extends Comparable<Key>, Value> {

    class Node {                                    // a node in the heap
        Key key;
        Value value;
        // ... whatever else you need in your node
    }

    public class Locator {                          // locates a node
        private Node u;                             // the node
        private Locator(Node u) { this.u = u; }     // constructor
        private Node get() { return u; }            // getter
    }

    public Locator insert(Key x, Value v) {         // insert (x,v)
        // ... your code to create a new leaf node u
        return new Locator(u);
    }

    // ... other QuakeHeap members
}
```

Operations: For this part of the project, we will begin by implementing the basic functions needed to insert keys and to build subtrees. Here is a list of the operations you are to implement. (Further details available on the CMSC420 Projects page.)

`QuakeHeap(int nLevels)`: This constructs an empty quake heap. The parameter `nLevels` indicates the number of levels to allocate in your arrays `roots` and `nodeCt`.[1] This allocates and initializes the `roots` and `nodeCt` arrays and any other private data that your class uses.

`void clear()`: This resets the structure to its initial state. In particular, it resets all the node counts to zero and clears all the `roots` lists.[2]

`Locator insert(Key x, Value v)`: This inserts the key-value pair (x, v) in the heap. This creates a "trivial" tree consisting of a single root node at level 0, that stores this key-value pair. It inserts this node into the list `roots[0]`. It returns a `Locator` (see above) referencing the newly created leaf node.

`int getMaxLevel(Locator r)`: Given a locator `r`, this determines the maximum height of an ancestor reachable by following the reversal of left-child links up the tree. (If the keys are unique, this is the highest node in the tree that has the same key as `r`). For example, for the heap in Fig. 1, the max-level of leaf labeled 21 is 0, the max-level of 4 is 2, the max-level of 1 is 4, and the max-level of 16 is 1.

`Key getMinKey()`: This returns the smallest key in the heap and also reorganizes the heap, merging many small trees into one large tree.

- If the heap is empty, throw an `Exception` with the message `"Empty heap"`.
- Otherwise, find the minimum key in the heap by enumerating all the nodes in all the `roots` lists, we find the one with the smallest key. (Ties may be broken arbitrarily.)
- Next, consolidate trees by the following process, called `merge-trees`. Enumerate the levels bottom-up, from zero up to the second highest level (that is, `nLevels-2`). At each level `k`:
  - Sort the nodes of `roots[k]` in increasing order by their keys.[3] Ties may be broken arbitrarily.
  - Next, merge trees in pairs as follows. While `roots[k]` has at least two roots:
    - Extract the first two root nodes from the sorted list. Call them `u` and `v`. Since the list is sorted, we know that `u.key` $\leq$ `v.key`.
    - Create a new root node `w`, with `u` as its left child and `v` as its right child. (Don't forget to set `u` and `v`'s parent links to point to `w`.) By our convention, `w`'s key is set to `u.key`. (We don't care about `w`'s value field. You can just set it to `null`.)
    - Add `w` to `roots[k+1]`.

Observe that when the merge-tree process is finished, every level, except possibly the top one, has at most one root. This is illustrated in Fig. 2.

[1] From a software design perspective, it would be better if the constructor did not have this parameter, and the array just grows dynamically as needed. Limiting the array size will be useful for testing purposes.

[2] You might realize that there is a potential memory leak here since locators may have been generated that refer to entries that have been removed. Fixing this is not trivial, but we won't worry about it.

[3] If you store your `roots[k]` as a Java `LinkedList`, you can invoke `Collections.sort()` to sort them.

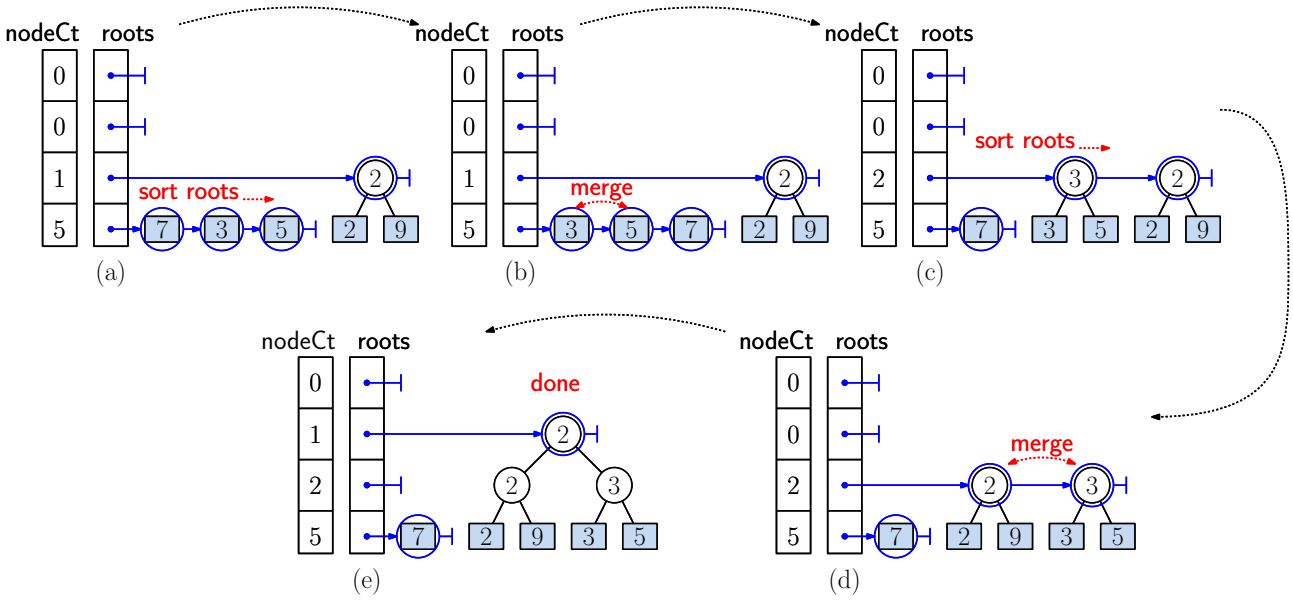

Figure 2: Merging trees. Working bottom-up, we sort the roots at each level, and then merge consecutive pairs until either zero or one root remains.

`ArrayList<String> listHeap()`: This operation lists the contents of your structure in the form of an array-list of strings. The precise format is important, since we check for correctness by "diff-ing" your strings against ours.

Enumerate the levels of the tree from bottom up. For each level, do the following:

- If the the node count for this level is zero, skip this level and go on to the next. Otherwise, sort the root nodes of this level by their key values, just as in `getMinKey()`.
- Generate a *level header* in the form of a string "{`lev:` *xxx* `nodeCt:` *yyy*}" and add it to the array-list. Here, "*xxx*" is the level index and "*yyy*" is the node count for this level. For example, if there are four nodes on level two, this generates the string, "`{lev: 2 nodeCt: 4}`".
- For each root node r in the `roots` list for this level, enumerate the the nodes of this tree based on a preorder traversal. For each node `u` visited in this traversal, we do the following:

  Internal: (`u.level` ≥ 1) Generate the string `"(" + u.key + ")"`. Recursively visit `u.left` and `u.right`.

  Leaf: (`u.level` $= 0$) Generate the string `"[" + u.key + "␣" + u.value + "]"` (where "␣" denotes a single space) and return.

  Null: (`u` $=$ `null`) This cannot happen in Part-A, but it can in Part-B. If so, generate the string `"[null]"` and return.

As an example, invoking `listHeap` on the structure appearing in to Fig. 2(e) would result in the 11-element array-list shown below. (For simplicity, we set the value of key x to be the string "`X0`x".)

Index	Array-List Contents
0:	{lev: 0 nodeCt: 5}
1:	[7 X07]
2:	{lev: 1 nodeCt: 2}
3:	{lev: 2 nodeCt: 1}
4:	(2)
5:	(2)
6:	[2 X02]
7:	[9 X09]
8:	(3)
9:	[3 X03]
10:	[5 X05]

Unfortunately, it is not easy to interpret the tree structure from this preorder listing, but we have provided a function in `CommandHandler.java` that reformats the tree so it is easier to read. For example, given the above array-list, our function would generate the following output for you. (Contrast this with the tree of Fig. 2(e).)

```
Structured list:
  {lev: 0 nodeCt: 5}
    Tree: 0
      [7 X07]
  {lev: 1 nodeCt: 2}
  {lev: 2 nodeCt: 1}
    Tree: 0
      | | [2 X02]
      | (2)
      | | [9 X09]
      (2)
      | | [3 X03]
      | (3)
      | | [5 X05]
```

Skeleton Code: As in the earlier assignment, we will provide skeleton code on the class Projects Page. The only file that you should need to modify is `QuakeHeap.java`. Remember that you must use the package "`cmsc420_s22`" in all your source files in order for the autgrader to work. As before, we will provide the programs `Tester.java` and `CommandHandler.java` to process input and output. You need only implement the data structure and the functions listed above. Below is a short summary of the contents of `QuakeHeap.java`.

```
package cmsc420_s22; // don't change this!
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() { ... }
}
```

Efficiency requirements: 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. A portion of your grade will depend on the efficiency of your program.

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 30 points. Of these, 25 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.