CMSC 420: Fall 2022

Programming Assignment 1: Leftist Heaps

Handed out: Tue, Sep 27. Due: **Tue, Oct 11, 11:59pm**.

Overview: In this programming assignment you will implement a leftist heap, the mergeable heap structure presented in Lecture 5. Your implementation will support all the basic functions of a mergeable priority queue, namely `insert`, `extract-min`, and `merge`. There will also be a couple of additional operations including an operation to list the contents of your tree structure so we can check its correctness.

Operations: You will implement the following public functions. Subject to the efficiency requirements described below, you are free to create whatever additional private/protected data and utility functions as you like.

`public LeftistHeap():` This constructs an empty leftist heap. This creates an empty tree by initializing the `root` to `null` (and any other initializations as needed by your particular implementation).

`boolean isEmpty():` Returns `true` if the current heap has no entries and `false` otherwise.

`void clear():` This resets the structure to its initial state, removing all its existing contents.

`void insert(Key x, Value v):` This inserts the key-value pair $(\mathtt{x}, \mathtt{v})$, where `x` is the key and `v` is the value. (**Hint:** This can be done with a single call to the utility function `merge`, without the need of loops or recursion.)

`void mergeWith(LeftistHeap<Key, Value> h2):` This merges the current heap with the heap `h2`. If `h2` is `null` or it references this same heap (that is, `this == h2`) then this operation has no effect. Otherwise, the two heaps are merged, with the current heap holding the union of both heaps, and `h2` becoming an empty heap.

For testing purposes, you should implement merge operation so it produces exactly the same tree as in the lecture notes.

`Key getMinKey():` This returns the smallest key in the heap, but makes no changes to the heap's contents or structure. If the heap is empty, it returns `null`.

`Value extractMin():` If the heap is empty, this throws an `Exception` with the error message `"Empty heap"` Otherwise, this locates the entry with the minimum key value, deletes it from the heap, and returns its associated value. (**Hint:** This can be done with a single call to the utility function `merge`, without the need of loops or recursion.)

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

Starting at the root node, visit all the nodes of this tree based on a **right-to-left preorder traversal**. In particular, when visiting a node reference `u`, we do the following:

Null: $(\mathtt{u} = \mathtt{null})$ Add the string `"[]"` to the end of the array-list and return.

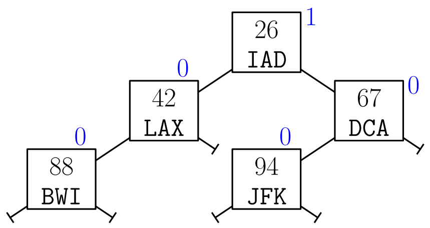

Figure 1: A leftist heap.

Non-null: (u ≠ `null`) Add the string `"(" + u.key + ",␣" + u.value + ")␣[" + u.npl + "]"` with the node's key, value, and npl value to the end of the array-list. (The symbol "␣" is a space.) Then recursively visit `u.right` and then `u.left`.

An example of the output on the tree shown in Fig. 1 is shown below.

Index	Array-List Contents
0:	`(26, IAD) [1]`
1:	`(67, DCA) [0]`
2:	`[]`
3:	`(94, JFK) [0]`
4:	`[]`
5:	`[]`
6:	`(42, LAX) [0]`
7:	`[]`
8:	`(88, BWI) [0]`
9:	`[]`
10:	`[]`

This format has been chosen for a particular reason. It is very easy to produce a nicely formatted output based on this. Given the above output, our program will generate the following structured output. If you rotate it 90° clockwise, it looks quite similar to the tree structure of Fig. 1.

```
Formatted structure:
  | (67, DCA) [0]
  | | (94, JFK) [0]
  (26, IAD) [1]
  | (42, LAX) [0]
  | | (88, BWI) [0]
```

Split: The operations described above follow directly from the code given in class. We would like you to implement one more operation. This is more challenging. It is worth 10 points. So, if you do not implement it correctly, you will still get most of the credit for the assignment.

`LeftistHeap<Key, Value> split(Key x)`: Given a key `x`, this splits the heap into two, the current one contains all the entries whose keys are less than or equal to `x` and all the entries whose keys are strictly greater than `x` are moved into a new heap, which is returned.

For the sake of efficiency (and so we can test your output), the process should be implemented as follows. First, we create an empty list (e.g., a Java `ArrayList`) of nodes. Next, perform a left-to-right preorder traversal of the current tree. When we visit a node `u`, if $\texttt{u.key} \leq \texttt{x}$, then leave the node unchanged and apply the traversal recursively, first to the left subtree and then to the right subtree. On the other hand, if $\texttt{u.key} > \texttt{x}$, unlink this node from the current tree and append it to the end of the list. By heap ordering, we know that all the nodes of this subtree will be placed in the new heap.

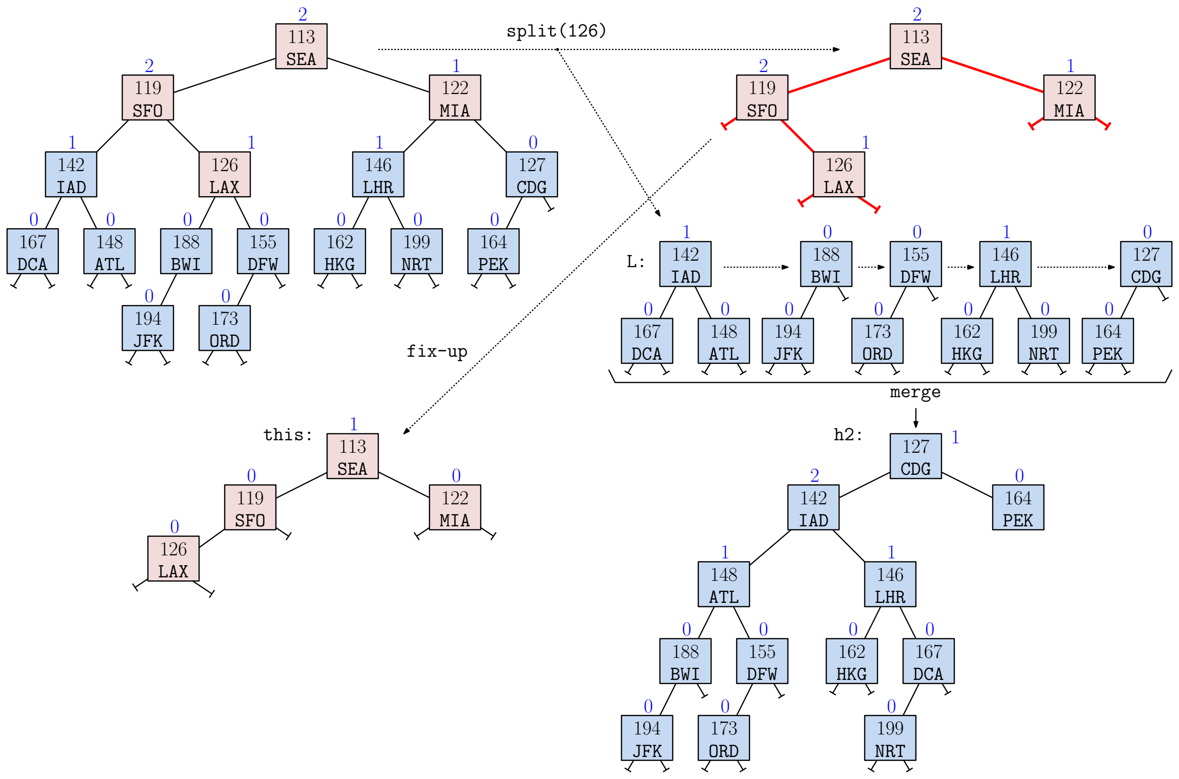

Figure 2: Splitting a leftist heap. We traverse the tree, unlinking all subtrees whose key value strictly exceeds `x = 126`. We merge these trees (from left to right) to form the final result `h2`. We traverse the current tree, update the npl values and swap subtrees so that the leftist property holds.

When this process returns, we have a list $L = \langle u_1, \ldots, u_k \rangle$ of maximal subtrees whose nodes are to be placed in the new heap. These are ordered from left to right. We create a new empty leftist heap, called `h2`, and we merge each of the elements of L into this new heap, from left to right. That is, we perform `h2.mergeWith`(u_i) for $i = 1, 2, \ldots, k$. (Unfortunately, this statement is not kosher because u_i is not a heap, it is just a node. But this is effectively what your program should perform.)

Finally, the original tree has had a number of subtrees removed from it. As a result, the npl values may be wrong and the leftist property may be violated. (Note that the other tree, `h2` will be valid, because it was formed through merges.) To fix it, perform a traversal of the tree, compute the proper npl values, and perform left-right child swaps whenever needed to enforce the leftist property.

Now, both trees have their proper contents and satisfy the required properties. Finally, return a reference to the leftist heap `h2` as the final result.

Note that there are many different valid leftist heaps containing a given set of nodes, and if your implementation differs from the one described above, you will obtain a different tree and your results will not match ours.

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 `LeftistHeap.java`. Remember that you must use the package "`cmsc420_f22`" in all your source files in order for the autgrader to work. As before, we will provide the programs `Part1Tester.java` and `Part1CommandHandler.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 `LeftistHeap.java`.

Class Structure: The high-level `LeftistHeap` class structure is presented below. The entries each consist of a key (priority) and associated value. These can be any two types, but it must be possible to make comparisons between keys. Our class is parameterized with two types, `Key` and `Value`. We assume that the `Key` object implements Java's `Comparable` interface, which means that is supports a method `compareTo` for comparing two such objects. This is satisfied for all of the Java's standard number types, such as `Integer`, `Float`, and `Double` as well as for `String`.

We recommend that the tree's node type, called `LHNode`, is declared to be an inner class. (But you can implement it anyway you like and give it any name you like.) This way, your entire source code can be self contained in a single file.

```
public class LeftistHeap<Key extends Comparable<Key>, Value> {

    class LHNode {                          // recommended node (you may change)
        Key key;                            // key (priority)
        Value value;                        // value (application dependent)
        LHNode left, right;                 // children
        int npl;                            // null path length
        // ... any utility functions you want to define
    }

    // ... any private and protected members you need

    public LeftistHeap() { ... }            // constructor
    public boolean isEmpty() { ... }        // is the heap empty?
    public void clear() { ... }             // clear its contents
    public void insert(Key x, Value v) { ... }                  // insert (x,v)
    public void mergeWith(LeftistHeap<Key, Value> h2) { ... } // merge with h2
    public Key getMinKey() { ... }                              // get min key
    public Value extractMin() throws Exception { ... }          // extract min
    public ArrayList<String> list() { ... }                     // list entries
}
```

Efficiency requirements: The functions `insert`, `mergeWith`, and `extractMin` should all run in

$O(\log n)$ time. The function `getMinKey()` should run in $O(1)$ time. The function `list()` should run in time proportional to the size of the tree. A portion of your grade will depend on the efficiency of your program.

The function `split` should run in time $O(k \log n)$, where k is the number of subtrees that need to be merged together. (In the worst case, k may be as high as $O(n)$, but your function should be more efficient when k is small. If you follow the outline that we have given for `split`, you should achieve this running time.)

The public interface should match what you are given in the skeleton code. You are free to add whatever private and protected members (both data and functions, subject to these efficiency requirements.)

Testing/Grading: Submissions will be made through Gradescope (you need only upload your modified `LeftistHeap.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 80 points. Of these, 60 points will be for the heap operations excluding `split`. Correctly implementing `split` is worth 10 points, and additional 10 points is reserved for clean programming style and the above efficiency requirements.