CMSC 420: Spring 2023

Programming Assignment 1: Weight-based Leftist Heaps

Handed out: Thu, Feb 16. Due: **Wed, Mar 1, 11:59pm**.

Overview: In this programming assignment you will implement a variant of a leftist heap. Recall that a leftist heap is a mergeable form of the heap structure, which was presented in Lecture 5. Rather than using NPL (null-path length) values to balance the tree, our data structure will be "weight-based," in the sense that balance will be based on the number of nodes in each subtree.

A Weight-Based Leftist Heap: Our data structure, called `WtLeftHeap` will store key-value pairs, where the key represents the entry's priority. The data type is generic and is 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. It is declared as follows:

```
public class WtLeftHeap<Key extends Comparable<Key>, Value>
```

Our weight-based leftist heap differs in a number of respects from the standard leftist heap presented in class:

- Rather than a min-heap, this will be a *max-heap*, meaning that the each node's parent key is greater than or equal to its key (see Fig. 1).

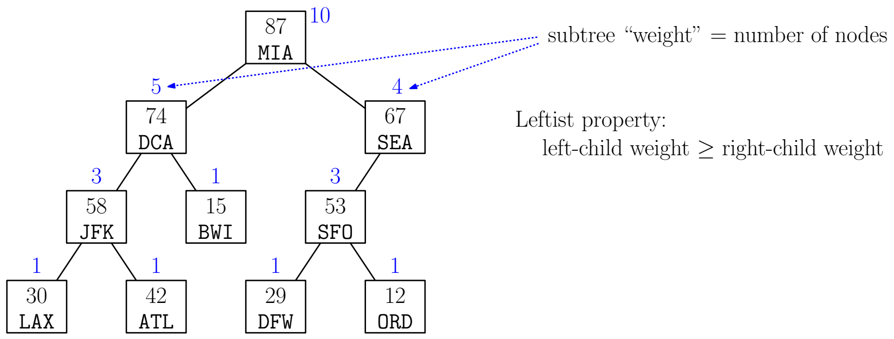


Figure 1: Weight-based leftist heap.

- Rather than basing the tree structure on the NPL (null path length), it will be based on a node's *weight*, which is defined to be the number of nodes in the subtree rooted at this node. Each node will store its weight. The weight of its left subtree can never be smaller than the weight of its right subtree (see Fig. 1).
- In addition to the standard leftist-heap operations, you will also implement a function that *updates* the key (priority) of a given entry. Since this alteration may violate the max-heap structure, the tree structure may need to be updateed as well.

- In order to efficiently support the above update operation, we will need to make two additional enhancements to the tree structure. First, (as with the adjustable stack from the earlier assignment) *locators* will be used to identify the node to be updateed. Since the subsequent reorganization may involve entries moving both up and down the tree, we will need to add a *parent link* to each node of the tree.

Locators and Adjustments: In many applications of heaps, it is necessary to update the priorities of entries that reside in the heap. This raises two questions: (1) how to efficiently identify an entry within the heap and (2) how to efficiently update the heap's structure after making the modification. As in the earlier programming assignment, we will handle the first issue with the use of *locators*.

Whenever we insert a key-value pair in the tree, we return a reference to the node containing the newly created entry. Since a node will be a protected object within our data structure, we cannot just return a reference directly to it. Instead, we create a special public class within our data structure, called a `Locator`, which encapsulates this reference. In addition to creating a new node and adding it to the data structure, the insert function also creates a new `Locator` object that references this node and returns it to the user of our data structure. Now, when the user wants to update a key, it can use this locator to efficiently identify the node to be updateed. A skeletal example of how to set this up is provided below.

```
public class WtLeftHeap<Key extends Comparable<Key>, Value> {
    private class Node { ... }            // a node of the tree (private)
    public class Locator {                // a node locator (public)
        private Node node;                // hidden reference to the node
        ...
    }
    public Locator insert(Key x, Value v) // insert method returns a Locator
    ...
}
```

It is the user's responsibility to save these locators, and it is the data structure's responsibility to see that they are properly updated. As an example, consider the weight-based leftist heap shown in Fig. 2(a), and suppose that the user wants to update the key of `ORD` from 12 to 82. When `ORD` was added to the heap, the `insert` function returned a locator object. To modify the key, we pass the locator into the `updateKey` function, which allows us to access the node directly. We can then change its key to 82. Unfortunately, the heap order is now violated. Since the key has increased, we can fix this by sifting the entry up the tree, repeatedly swapping with its parent, until its parent's key is at least as larger (see Fig. 2(b)). Depending on how you implement this operation (moving nodes or copying their contents) other locators may need to be updated as well.

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.

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

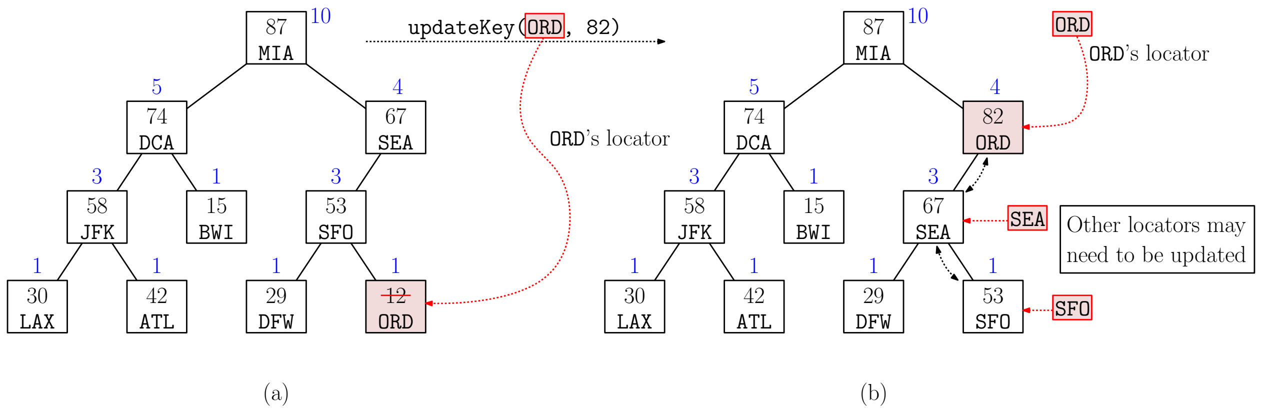


Figure 2: (a) A weight-based leftist heap (with weights indicated in blue) and (b) the `updateKey` operation.

`int size()`: Returns the number of entries in the current heap.

`void clear()`: This resets the structure to its initial state, effectively removing all its existing contents. (You do not need to do anything to existing locators that point into your data structure. It is the responsibility of the user to remove these.)

`Locator insert(Key x, Value v)`: Inserts the key-value pair $(\mathtt{x}, \mathtt{v})$, where `x` is the key and `v` is the value.

Hint: This can be implemented without the need for loops or recursion by making a single call to the `merge` utility function.

`void mergeWith(WtLeftHeap<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. The only difference is that rather than using NPL values to determine which subtree is on the left, use the weight of each subtree, that is, the number of nodes. (As an exercise, convince yourself that the rightmost path of such a tree has $O(\log n)$ nodes.)

`Value extract()`: If the heap is empty, this throws an `Exception` with the error message `"Extract from empty heap"` Otherwise, this locates the entry with the maximum key value, deletes it from the heap, and returns its associated value.

Hint: This can be implemented without the need for loops or recursion by making a single call to the `merge` utility function.

`void updateKey(Locator loc, Key x)`: Change the key associated with entry `loc` to `x`, and update the structure appropriately. You may assume that `loc` is a valid locator for this instance of the data structure.

If the key increases, repeatedly swap this entry with its parent until reaching the root or until the parent key is greater than or equal. If the key decreases, repeatedly swap with the larger of the two children until reaching the leaf level or until both children's keys are less than or equal.

Hint: There are two obvious methods on how to do this. One involves swapping entire nodes by unlinking and relinking them, and the other involves leaving the nodes where they are, but swapping their contents. You may implement whichever version you prefer (since it won't affect the results).

`Key peekKey():` Returns the maximum key in the heap (that is, the key associated with the root node). If the tree is empty, return `null`.

`Value peekValue():` Returns the value associated with the maximum key in the heap (that is, the value associated with the root node). If the tree is empty, return `null`.

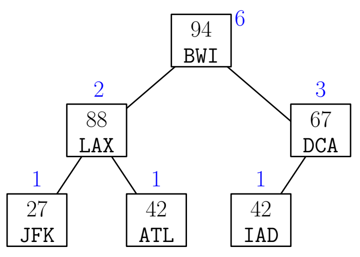
`ArrayList<String> list():` This operation lists the contents of your tree in the form of a Java `ArrayList` of strings. The precise format is important, since we check for correctness by "diff-ing" your strings against ours.

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

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

Our command handler program takes the result of the `list` command and generates a nicely formatted tree (see Fig. 3(c)). The reason for performing the traversal in right-to-left order, rather than the traditional left-to-right, is so that the printed result looks like a 90° rotation of the tree.



(a)

Raw output

```
(94, BWI) [6]
(67, DCA) [2]
[]
(42, IAD) [1]
[]
[]
(88, LAX) [3]
(48, ATL) [1]
[]
[]
(27, JFK) [1]
[]
[]
```

(b)

Formatted output

```
  | (67, DCA) [2]
  | | (42, IAD) [1]
  (94, BWI) [6]
  | | (48, ATL) [1]
  | (88, LAX) [3]
  | | (27, JFK) [1]
```

(c)

Figure 3: (a) A heap, (b) the result of `list`, and (c) the formatted output produced by our program.

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 `WtLeftHeap.java`.

Remember that you must use the package "`cmsc420_s23`" 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.

Class Structure: The high-level `WtLeftHeap` class structure is presented below. There are two inner classes, a private one for the node and a public one for the locator. 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 `Node`, 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 WtLeftHeap<Key extends Comparable<Key>, Value> {

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

    // ... any private and protected data and utility functions

    public WtLeftHeap() { ... }
    public int size() { ... }
    public void clear() { ... }
    public Locator insert(Key x, Value v) { ... }
    public void mergeWith(WtLeftHeap<Key, Value> h2) { ... }
    public Value extract() throws Exception { ... }
    public void updateKey(Locator loc, Key x) { ... }
    public Key peekKey() { ... }
    public Value peekValue() { ... }
    public ArrayList<String> list() { ... }
}
```

Efficiency requirements: (10% of the grade) The functions `insert`, `mergeWith`, and `extract` should all run time proportional to the length of the rightmost chain of the trees involved. The function `updateKey` should run in time proportional to the distance that the node needs to travel during the sifting process. (This is not guaranteed to be $O(\log n)$, since the height of the tree may be much larger than this.) The functions `size()`, `peekKey()`, and `peekValue()` should all run in $O(1)$ time. The function `list()` should run in time proportional to the number of nodes in the tree. A portion of your grade will depend on the efficiency of your program.

Style requirements: (5% of the final grade) Good style is not a major component of the grade, but you should demonstrate some effort here. Part of the grade is based on clean, elegant coding, and a reasonable amount of comments. There is no hard rules here, and we will not be picky. If we deduct points, it will because you used an excessively complicated structure to implement a relatively simple computation.

The other part is based on commenting. You should have a comment at the top of each file you submit. This identifies you as the author of the program and provides a short description of what the program does. For each function (other than the most trivial), you should also include a comment that describes what the function does, what its parameters are, and what it returns. (If you would like to see an example, check out our *canonical solution* to Programming Assignment 0, on the class Project Page.)

Testing/Grading: Submissions will be made through Gradescope. You need only upload your modified `WtLeftHeap.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.

Challenge Problem: (Remember that challenge problems are for extra credit points, which are not part of the standard grade.)

One of the dangers of locators is that they contain a link pointing to a node within your data structure. If a user makes an error and attempts to use an invalid locator, it can destroy the integrity of your data structure. Modify your program by adding an additional check to `updateKey` that verifies that the given locator is valid for the given heap. If not, your program should throw an `Exception` with the error message `"Invalid locator"`. This may necessitate changes to other heap functions as well.

There are two ways that a locator may become invalid:[1]

- An entry is extracted from the heap, and later an attempt is made to access its locator to update the key of this nonexistent entry.
- A locator `loc` points to a node that is currently in one heap (say, `h1`) but the user attempts to access it through an `updateKey` on a different heap (say, `h2.updateKey(loc, 999)`).

The test run `testEC1` tests the above functionality. You may assume that the user is not allowed to copy the contents of one locator to another.

If you do the challenge problem, add a comment to the top of your `WtLeftHeap.java` stating that you attempted the challenge problem. Also, explain how you implemented it (what was your method and which additional class objects and functions were added).

Grading will depend in part on how efficient your implementation is. For basic credit, your locator validation should run in time proportional to the height of the tree. Note, however, that a leftist tree is generally not of $O(\log n)$ height, because left-side paths can be arbitrarily long. For additional credit (extra-extra credit!), perform the validation in $O(\log n)$ time, and provide a careful explanation of how your algorithm works.

[1]There is actually a third way. All the locators become invalid when the tree is cleared. Don't worry about this case, however, since we never test it.