Programming Assignment 1: Extended AA Trees


Overview: In this assignment you will implement an extended variant of the AA Tree. Recall that an extended binary tree there are two different node types, internal nodes and external nodes. Extended trees are used in many applications. By separating node types, we can better tailor each node to its particular function. Data (that is, the key-value pairs) are stored only in the external nodes, which internal nodes only store keys, called splitters. Together, the internal nodes serve as an index, directing the search to the appropriate external node where the data is stored.

Our extended AA tree, or AAXTree, will be templated by two types Key and Value. Our only assumption regarding these types is that the Key type implements the Java Comparable interface, meaning that it defines a function compareTo() for comparing keys. In our test data, keys will be of type String and values of type Airport.

Extended AA Tree: Recall that a traditional AA tree is a binary variant of the 2-3 tree and is a close relative of red-black trees. Specifications on how to implement an extended version of this tree are given in the Supplemental Lecture on Extended AA Trees, which will be posted on the class’s Projects Page. An example of such a tree is shown in Fig. 1(a).

Figure 1: (a) An extended AA Tree storing the 15 key-value pairs \{(1, b), (3, f), (4, d), \ldots\}, (b) ordering convention. See the Projects Page for further information.

Part 1a Requirements: (Due Mon, Mar 8, 11pm - 30%) This first part involves the basic functionality to find, insert, list the dictionary’s contents, and clearing the dictionary. Because our autograding program will check that your tree matches ours exactly, it is important that you follow the implementation described in the Supplemental Lecture on extended AA Trees.

Value find(Key x): Determines whether there is a key-value pair \((x, v)\), and if so returns a reference to \(v\). Otherwise, it returns null.
void insert(Key x, Value v) throws Exception: Inserts key value \((x, v)\), throwing an Exception with the message "Insertion of duplicate key" if there is a key-value pair in the dictionary with key \(x\).

**ArrayList<String> getPreorderList():** This operation generates a preorder enumeration of the nodes in the tree. This is represented as a Java ArrayList of type String, with one entry per node. The \texttt{key} and \texttt{value} refer to the key and value stored in the node. We assume that both provide a \texttt{toString} method. The \texttt{level} value refers to the node's level in the AA tree.

- Internal nodes: "\((\text{key} + \ "\_\") + \text{level} \) (where \"\_\" is a space character)
- External node: "\([\text{key} + \ "\_\") + \text{value} + \ "\)]"

Our autograder program is sensitive to both case and whitespace. For example, given the tree of Fig. 1, a partial list of the ArrayList contents are shown in Fig. 2.

<table>
<thead>
<tr>
<th>Index</th>
<th>Contents</th>
<th>Index</th>
<th>Contents</th>
<th>Index</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(7) 3</td>
<td>5</td>
<td>(5) 1</td>
<td>10</td>
<td>(8) 1</td>
</tr>
<tr>
<td>1</td>
<td>(4) 2</td>
<td>6</td>
<td>[4 d]</td>
<td>11</td>
<td>[7 u]</td>
</tr>
<tr>
<td>2</td>
<td>(2) 1</td>
<td>7</td>
<td>[5 t]</td>
<td>12</td>
<td>(10) 1</td>
</tr>
<tr>
<td>3</td>
<td>[1 b]</td>
<td>8</td>
<td>(17) 3</td>
<td>13</td>
<td>[9 h]</td>
</tr>
<tr>
<td>4</td>
<td>[3 f]</td>
<td>9</td>
<td>(13) 2</td>
<td>14</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 2: Partial result from \texttt{getPreorderList} for the tree shown in Fig. 1(a).

void clear(): This removes all the entries of the tree (e.g., by setting the root pointer to null).

**Part 1b Requirements:** (Due Wed, Mar 24, 11pm - 70%) In this part you will implement the remaining operations.

int size(): Returns the number of key-value pairs in the dictionary. For example, for the tree of Fig. 1(a), this would return 15.

void delete(Key x) throws Exception: Deletes the entry with key \(x\). If there is no such entry, it throws an Exception with the error message "Deletion of nonexistent key" if there is no key-value pair in the dictionary with key \(x\).

Value getMin(): This returns the value associated with the smallest key of the dictionary. For example, on the tree of Fig. 1(a), this returns "b". If the dictionary is empty, these both return null.

Value getMax(): Same as getMin, but for the largest key of the dictionary.

Value findSmaller(Key x): This returns the value associated with the entry having the largest key that is strictly smaller than \(x\). (Note that \(x\) may or may not be in the dictionary.) If the dictionary is empty or if there is no key smaller than \(x\), this returns null.

For example, on the tree of Fig. 1(a), \texttt{findSmaller(18)} would return "o", the value associated with the next smaller key 14. More generally, \texttt{findSmaller(x)} for \(14 < x \leq 18\) returns "o". Finally, \texttt{findSmaller(1)} or generally \texttt{findSmaller(x)} for any \(x \leq 1\) would return null.
Value findLarger(Key x): Same as findSmaller, but for the next strictly larger key of the dictionary.

Value removeMin(): Deletes the entry from the dictionary associated with the smallest key, and returns its associated value. If the dictionary is empty, this returns null, and the dictionary is unchanged. As with any deletion operation, the tree should be rebalanced after the deletion. For example, on the tree of Fig. 1(a), removeMin() would delete the entry (1, b) and return the value “b”.

Value removeMax(): Same as removeMin, but for the largest key of the dictionary.

Skeleton Code: As in the first assignment, we will provide skeleton code on the class Projects Page. The only file that you should expect to modify is AAXTree.java. Use must use the package “cmsc420_s21” 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).

```java
package cmsc420_s21;

public class AAXTree<Key extends Comparable<Key>, Value> {
    public AAXTree() { ... } // you fill these in
    public Value find(Key x) { ... }
    public Value find(Key x) { ... }
    public void insert(Key x, Value v) throws Exception { ... }
    public void insert(Key x, Value v) throws Exception { ... }
    // ... and so on
}
```

Efficiency requirements: Except for getPreorderList, all operations should run in \( O(\log n) \) time, where \( n \) is the number of entries in the data structure. The operation getPreorderList should run in time \( O(n) \). We will check this by a manual inspection of your code.

Testing/Grading: Submissions will be made through Gradescope (you need only upload your modified AAXTree.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. We will be checking the following items:

- You should use Java’s class inheritance to implement your internal and external nodes.
- All operations (except getPreorderList) should be implemented so they run in \( O(\log n) \) time.
- We will not check in detail for adherence to coding standards, but we may deduct points if your code is unusually complex of messy.