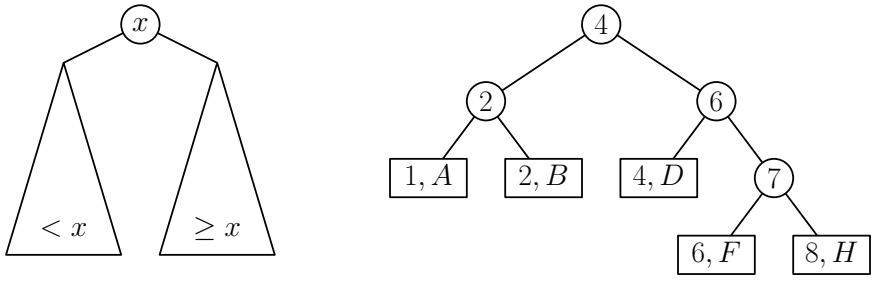
Programming Assignment 1: Extended Binary Search Trees (Preliminary)

Handed out: Tue, Sep 29. Due: TBA (but not before Thu, Oct 8).

Overview: In this assignment you will implement an extended version of a standard (unbalanced) binary search tree. (This data structure is discussed in class on 09/24.) Recall that an extended binary tree is one in which all internal nodes have two non-null children, and all the leaves are called external nodes. An extended binary search tree (which we will call *XBSTree* for short), is an alternative implementation of an ordered dictionary, storing key-value pairs. It differs from the traditional binary search tree in that the internal nodes do not store values, only keys. We refer to these as *splitters*. These splitters do not necessarily correspond to keys in the dictionary, they are merely used as an *index* to help us locate the external node that contains a given key-value pair. Separating the indexing (internal nodes) from the key-value storage (external nodes) has a number of practical advantages, which we will mention later this semester.

Defintion: Given an internal node storing a splitter value x, we make the convention that the key-value pairs whose keys are strictly smaller than x are stored its left subtree and those whose keys are greater than or equal to x are stored in its right subtree (see Fig. 1(a)).



(a) (b)

Figure 1: (a) Convention for internal nodes, and (b) a possible extended binary search tree storing the key-value pairs $\{(1, A), (2, B), (4, D), (6, F), (8, H)\}$.

An example of such a tree is shown in Fig. 1(b). Observe that the splitter values may or may not correspond to keys that appear in the leaf nodes. It is important to note that splitter values that do not arise as an actual key value in an external node (such as 7 in Fig. 1(b)) are not present in the dictionary. Thus, the operation `find(7)` would return `null`, and an attempt to insert a key-value pair with the same key value, such as $(7, G)$, would be allowed.

Operations: The operations for extended binary search trees are very similar to the standard trees we have seen. Here is a formal definition of how the operations work.

Initialization: The initial tree consists of a `root` pointer whose value is `null`. If the tree consists of a single key-value pair, the root points directly to an external node containing this pair. Once the tree has two or more entries, the root will point to an internal node.

`find(x)`: Starting at the root, we traverse the path to a leaf in the natural manner for a binary search tree. (If the root is `null`, then the search fails immediately.) Given an internal node with key y, we recurse on the left subtree if $x < y$ and on the right subtree if $x \geq y$. On arriving at an external node, we report success if x matches the key there and failure otherwise. If we succeed, we return the value associated with this pair, and otherwise we return `null`.

`insert(x, v)`: If the root is `null` (meaning that the tree is empty), we create a single external node containing the pair (x, v) and set the root to point to this node. Otherwise, we apply the same process as in `find` to determine the closest leaf node, say (y, w). If $x = y$, then we generate a duplicate-key error. Otherwise, we create an new external node containing the pair (x, v). There are two cases for how to connect it to the tree. If $x < y$, we create a new internal node in which we store the splitter value y. We make (x, v) its left child, (y, w) its right child, and this new internal node replaces (y, w) as the child of its parent (see Fig. 1(a)). On the other hand, if $x > y$, we create a new internal node in which we store the splitter value x. We make (y, w) its left child, (x, v) its right child, and this new internal node replaces (y, w) as the child of its parent.

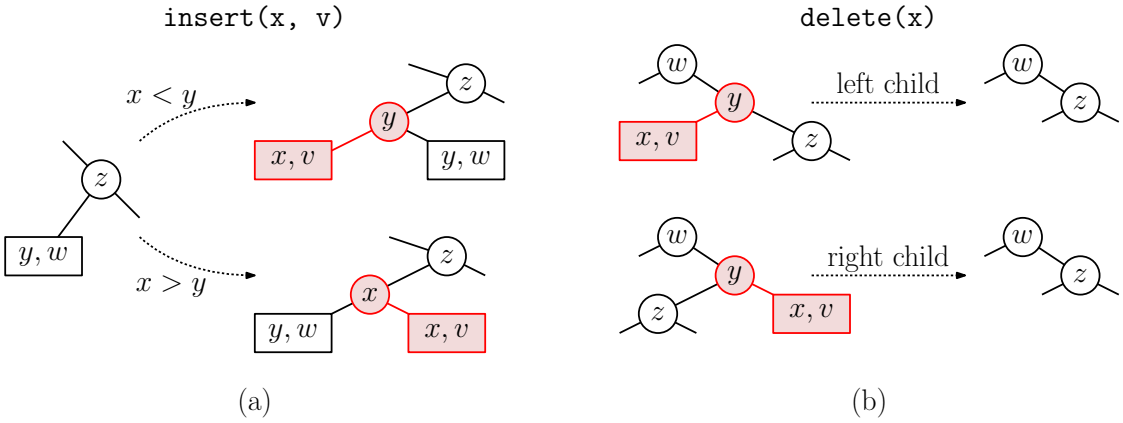


Figure 2: Insertion and deletion.

Observe that both cases are consistent with our convention that the key-value pairs in the left subtree are strictly smaller than the splitter value and those in the right subtree are greater than or equal to the splitter.

`delete(x)`: As usual, we apply the same process as in `find` to locate the leaf containing x. If x does not appear in this leaf, we generate a nonexistent-key error. If this external node has no parent (implying that the tree has only one entry), we delete this node and set the root to `null`. Otherwise, let y denote its parent, and let z denote the other child of y. (Note that y must be an internal node, so by definition of extended trees, it has two non-null children.) We delete both the nodes (x, v) and y, and make y's parent point to z, instead of y (see Fig. 2(b)).

`getPreorderList()`: This operation generates a preorder enumeration of the nodes in the tree. This is represented as a Java `ArrayList` of type `String`. We assume that both keys and values support a `toString` method. For each internal node storing a key x, the

`ArrayList` entry is of the form `"(" + x.toString() + ")"`. For each external node storing the pair (x, v), the entry is `"[" + x.toString() + " " + v.toString + "]"`. (Our testing programs are based on Java `String` equality. Note the use of parentheses for internal nodes and square brackets for external nodes. Also, note the single space between the key and value.)

For example, given the tree of Fig. 1(b), the elements of the `ArrayList` would consist of the following strings:

```
(4)
(2)
[1 A]
[2 B]
(6)
[4 D]
(7)
[6 F]
[8 H]
```

`getMin()` / `getMax()`: These two functions return the values associated with the smallest and largest key values of the dictionary. For example, given the tree of Fig. 1(b), these would return the values `A` and `H`, respectively. If the dictionary is empty, these return `null`.

`findDown(x)` / `findUp(x)`: Recall that the `find` function returns `null` if there is no entry with key x. These functions behave exactly like `find(x)` if there is a key-value pair whose key is x. If there is not such key-value pair, the `findDown` function returns the value associated with the *next smaller key* and `findUp` returns the value associated with the *next larger key*. If there is no such key (e.g., if x is smaller than the smallest key in the case of `findDown` or x is larger than the largest key in the case of `findUp`), the function returns `null`. Here are some examples for the tree of Fig. 1(b),

`findDown(0) =`	`null`	`findUp(0) =`	`A`
`findDown(2) =`	`B`	`findUp(2) =`	`B`
`findDown(3) =`	`B`	`findUp(3) =`	`D`
`findDown(7) =`	`F`	`findUp(7) =`	`H`
`findDown(8) =`	`H`	`findUp(8) =`	`H`
`findDown(10) =`	`H`	`findUp(10) =`	`null`

`clear()`: This removes all the entries from the dictionary, resulting in an empty tree.

Running-time Requirements: All the dictionary operations (`insert`, `delete`, `find`, `getMin`, `getMax`, `findUp`, and `findDown`) should run in time proportional to the height of the tree. The operation `getPreorderList` should run in time proportional to the number of nodes in the tree. We will determine this by inspection of your submitted code.

Program structure: 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. In particular, you will implement a class called `XBSTree`, which has the following public interface:

```
package cmsc420_f20;
```

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

    public XBSTree() { ... } // constructs an empty tree
    public Value find(Key x) { ... }
    public void insert(Key x, Value v) throws Exception { ... }
    public void delete(Key x) throws Exception { ... }
    public ArrayList<String> getPreorderList() { ... }
    public Value getMin() { ... }
    public Value getMax() { ... }
    public Value findDown(Key x) { ... }
    public Value findUp(Key x) { ... }
    public void clear() { ... }

}
```

Observe that the `XBSTree` class is parameterized with two types, `Key` and `Value`. We assume that both of these objects implement a `toString` method. The `Key` type implements `Comparable<Key>`, which means that it defines a comparator function, `compareTo`. To determine whether key `x` is less than key `y`, use `x.compareTo(y) < 0`.

Node structure: The most natural way to represent nodes is through inheritance (and we will check for this in grading). There should be a parent class, called say `Node`, from which we derive two subclasses, one for each type of node. Let's call them `InternalNode` and `ExternalNode` (but you can call them whatever you like). Only internal and external nodes will ever be generated (that is, the parent class is abstract). Thus, the `Node` methods are all declared to be "`abstract`," meaning that you don't supply a function body for them.

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

    private abstract class Node { // generic node (purely abstract)
        Key key;
        abstract Value find(Key x); // no function body for these!
        abstract Node insert(Key x, Value v) throws Exception;
        abstract Node delete(Key x) throws Exception;
    }

    private class InternalNode extends Node { // internal node
        Node left;
        Node right;
        Value find(Key x) { ... }
        Node insert(Key x, Value v) throws Exception { ... }
        Node delete(Key x) throws Exception { ... }
        // ... other functions omitted
    }

    private class ExternalNode extends Node { // external node
        Value value;
        Value find(Key x) { ... }
        Node insert(Key x, Value v) throws Exception { ... }
```

```
        Node delete(Key x) throws Exception { ... }
        // ... other functions omitted
    }

    // ... public functions (see above)
}
```

These are just examples. You are allowed to augment the above node structure by adding additional data fields, modifying the function signatures, or adding additional methods. Because recursive methods such as insert, find, and delete will depend on the type of node, the recursive utility functions should be member functions associated with each node type, rather than passing a reference to the node as an argument. For example, in class, when inserting on the right child of a node `p`, we used the command `p.left = insert(x, v, p.left)`. Using the above structure, the `insert` method associated with the `InternalNode` would have the following command instead: `left = left.insert(x, v)`.

Actual Test Data? What are the actual key and value types that we will use in our testing. The short answer is the you should not know and you should not care. Your program should work correctly, subject to the above assumptions and nothing else.

For this part of the assignment, our tests will involve a data set drawn from a database of airports. The keys will be three-letter codes (so called, IATA codes), such as "`DCA`", "`IAD`", "`LAX`", and such. The values will be stored in a class that contains related information, such as the airport's name, its city, country, and latitude and longitude. The only attributes that you will observe in our test output files are the IATA code and the airport's city. However, none of these will affect your implementation.

Skeleton Code: We will provide you with some skeleton code to start with. This consists of the following:

`XBSTree.java`: A skeletal version of the main class for the extended binary search tree. **This is the only file you need modify.**

`Airport.java`: A class that stores information about airports

`Point2D.java`: A small utility class for storing (x, y) coordinates. Used for each airport's latitude and longitude.

`XBSTreeTester.java`: Main program for testing your implementation. It inputs commands either from a file or standard input and sends output to another file or standard output.

`CommandHandler.java`: A class that processes commands that are read from the input file and produces the appropriate function calls to the member functions of your `XBSTree` class. This also contains a function that converts the output of the `getPreorderList` operation into an indented inorder traversal of the tree, which is a bit easier to read. Here is an example of the output for the above tree:

```
Tree structure:
  | | [1 A]
  | (2)
  | | [2 B]
  (4)
```

```
| | [4 D]
| (6)
| | | [6 F]
| | (7)
| | | [8 H]
```

You may create additional files as well. Other than `XBSTree.java` avoid modifying or reusing any of the above files, since we overwrite them with our own when testing your program. Use the package "`cmsc420_f20`" for all your source files.

Testing/Grading: We will be using Gradescope's autograder and JUnit for testing and grading your submissions. All the tests and the expected results are visible. We will provide a link to the final test data on the class Projects page. Some grading will be done manually, and this will constitute 10% of the final score. We will be checking the following items:

- You should use Java's class inheritance to implement your internal and external nodes.
- All dictionary operations (including `findUp`, `findDown`, `getMin`, and `getMax`) should be implemented so they run in time proportional to the height of the tree (and *not* proportional to the number of nodes in the tree).
- We will not check in detail for adherence to coding standards, but we may deduct points if your code is unusually complex of messy.

Submission Instructions and Late Policy:

Important disclaimer: This is the first time I have used the Gradescope autograder. This is not an "out of the box" process, and we needed to write a fair amount of code to evaluate your programs. If something does not appear to be correct or if you have any questions, please let me know.

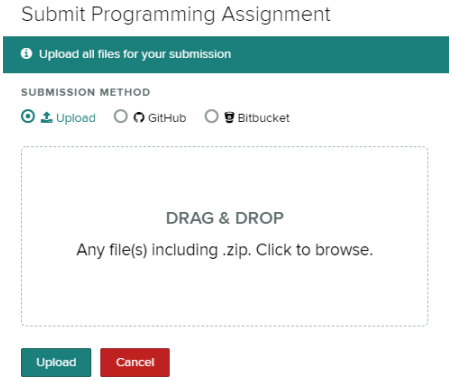


Figure 3: Gradescope submission.

Submissions will be made through Gradescope. Here is what to do:

- Log into the CMSC420 page on Gradescope, select this assignment, and select Submit. A window will pop up (see Fig. 3). You should submit all the files provided in the skeleton

code, `Airport.java`, `Point2D.java`, `XBSTreeTester.java`, and `CommandHandler.java`, and of course, your modified file `XBSTree.java`. Select "`Upload`" and another window will pop up confirming the submission. (There is no limit to the number of submissions you can make.) After a short time (perhaps minutes), Gradescope will display the results (see Fig. 4).

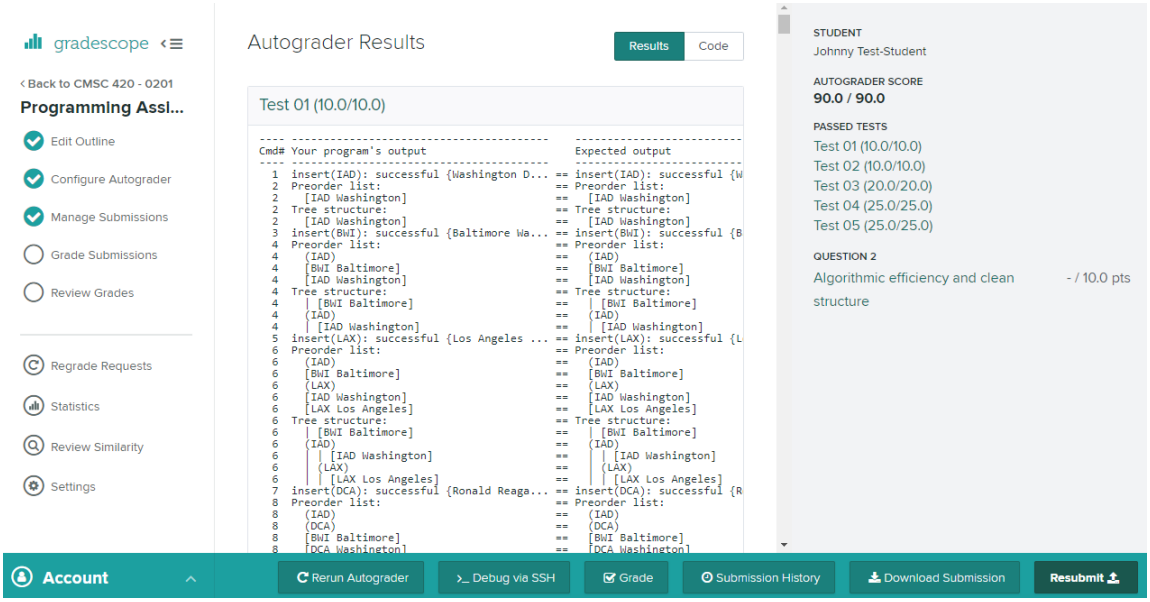


Figure 4: Gradescope autograder results (correct).

On the top-right of the page, it shows the scores of the individual tests as generated by the autograder. (If there are compilation errors, these will be displayed on this page.) The center of the window shows a line-by-line summary, with the output generated by your program on the left and the expected output on the right. If there are mismatches, these will be highlighted (see Fig. 5). The final score is based on the percentage of commands for which your program produced the expected output. Note that the comparison program is very simple. It compares line by line (without considering the possibility of inserted or deleted lines) and is sensitive to changes in case and the addition of white-space.

Late Policy: The late policy is the same as that listed in the course syllabus:

Up to 6 hours late:	5% of total
Up to 24 hours late:	10% of the total
For each additional 24 hours late:	20% of the total

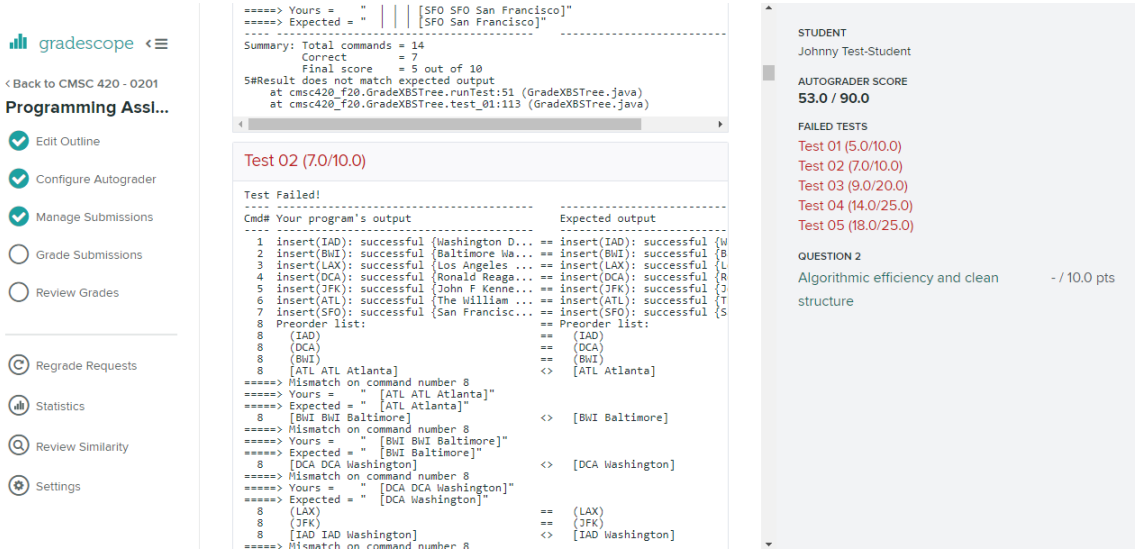


Figure 5: Gradescope autograder results (partially correct).