Basic Algorithms:
Guidelines for Programming Assignment 1

For this programming assignment, you are given "starter code" (available on the course home page) that provides you with data structures and algorithms for insertion into a 2-3 tree. To receive full credit:

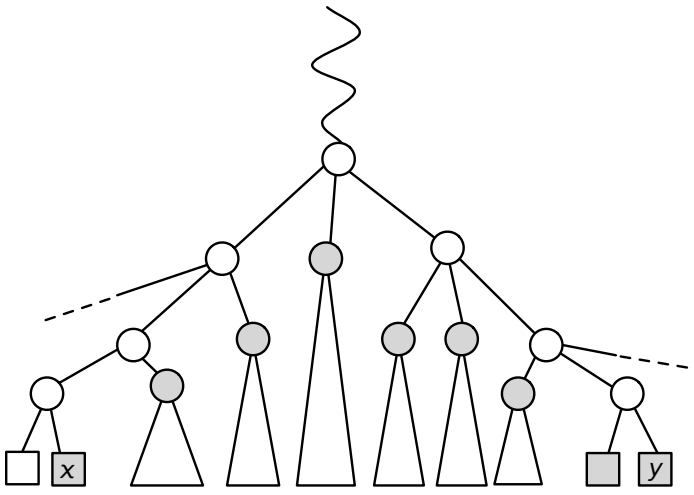
- you *must* make use of the given starter code, and in particular, you *must* use the given data layout of the given Node class, and you *may not* add any additional data members to this class;
- you *must* follow the high-level outline provided here — there are a number of different approaches to solve this problem, but you *must* use the approach given here.

Submissions that do not adhere to the above rules will receive very low grades.

You should read the problem statement on HackerRank for details of the input/output format. The main thing you have to do is implement an operation *printRange*(x, y) that prints out all leaves whose keys lie in the interval $[x, y]$. Here, we will assume that $x \leq y$; however, note that the data in the input file may not be of this form.

The goal is to implement this with an algorithm whose running time is $O(m + \log(n))$, where m is the number of keys in the interval $[x, y]$ and n is the total number of keys.

The strategy you are to use to implement *printRange*(x, y) is best understood by looking at the following diagram:



The strategy is as follows:

1. Perform a search for x, and record the search path; that is, if the tree has height h, construct an array of size $h+1$, and store pointers to all nodes along the search path.

   Iterative pseudocode for search was presented in class. You should adapt the logic of this to also record the search path.

2. Perform a search for y, and record the search path.

3. Calculate the position where the search paths for x and y diverge.

4. Conditionally print out the leaf node labeled x (since x may not be in the tree, you have to check if the key stored at that leaf actually lies in the interval $[x, y]$).

5. Walk the search path from x to the divergence point, printing out all the leaves of each tree hanging off to the *right* of the search path.

   In the above diagram, these are the trees rooted at the two left-most shaded internal nodes. To print out all the leaves in such a tree, you should use a recursive routine *printAll* — see below.

6. Process the divergence point.

   The node where the search paths split may have three children, and if (as in this particular diagram) the search path for x proceeds down the left-most child of that node, and the search path for y proceeds down the right-most child of that node, then print out all the leaves of the tree rooted at that middle child.

7. Walk the search path from the divergence point to y, printing out all the leaves of each tree hanging off to the *left* of the search path.

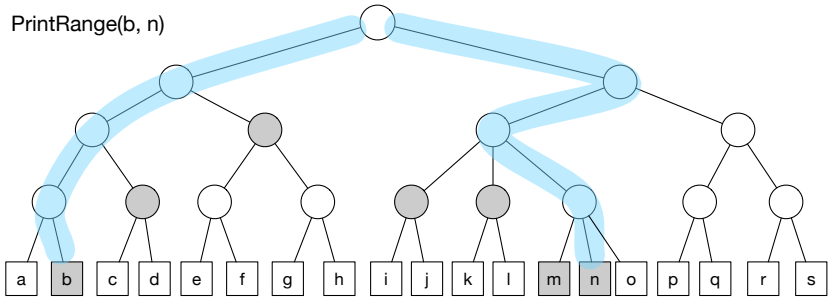
   In the above diagram, these are the trees rooted at the three right-most shaded internal nodes plus the shaded leaf node adjacent to the leaf node labeled y.

8. Conditionally print out the leaf node labeled y (again, since y may not be in the tree, you have to check if the key stored at that leaf actually lies in the interval $[x, y]$).

At a high level, this strategy prints out all the nodes in each tree rooted at the shaded nodes in the diagram, visiting the shaded nodes in a clockwise direction (and with the leaf nodes on the edges handled specially).

The *printAll* function. To implement the above, you will need to implement a recursive function $printAll(p, h)$ that prints all leaves in the subtree rooted at the node p. The routine should have an additional argument h that indicates the height of the node p — this makes it easy to know when to stop the recursion. This is the *only* recursive function that you should implement — everything else should be *iterative*. (Note that the "starter code" also contains recursive functions.)

A more concrete example. The following diagram gives a more concrete example:



Here, the keys are the letters a–s. To perform $printRange(b, n)$, one searches for the keys b and n. The corresponding search paths are highlighted in light green. The function *printAll* would be invoked on all of shaded internal nodes, as well as the shaded leaf holding the key m. The leaves holding b and n would be printed conditionally.

Corner cases. You may have to treat the following "corner cases" specially:

- Performing *printRange* on an empty tree.
- The search paths for x and y do not diverge.

Suggested development strategy. You should develop and do initial testing of your code on your own machine. You should try to develop and test small components, rather than try to develop the entire solution all at once. For example, you could write the high-level driver program that reads and parses the input, and prints out the results of your parsing. You could then write the *printAll* function, and then test it out with a program that just builds a 2-3 tree and calls *printAll* on the root. Only after you have all that working should you attempt to write the complete program.

Java specific issues. For this programming assignment, your program will generate a *lot* of output. It turns out that the usual way of outputting data in Java is horribly inefficient. Therefore, for this assignment you *must* use the `BufferedWriter` class for output. To access this class, you should import from `java.io`:

```
import java.io.*;
```

Then you can create a BufferedWriter as follows:

```
BufferedWriter output =
   new BufferedWriter(new OutputStreamWriter(System.out, "ASCII"), 4096);
```

Now, you will have to pass this `output` object as a parameter to any method that will produce output, or alternatively, you can access it via a "global variable". To output a `String s`, you do this:

```
output.write(s);
```

Finally, when your program is done producing output, you need to "flush" the `output` object:

```
output.flush();
```

NOTES:

- If you use `System.out` directly, instead of `BufferedWriter`, your program will definitely time out on several test cases, and you will not receive a very high score.
- You should construct *only one* `BufferedWriter` object during the execution of your program.
- `BufferedWriter` methods may throw some exceptions, and because of this, you will have to "decorate" your methods that invoke these methods directly or indirectly as follows:

```
static void myMethodForDoingSomething() throws Exception
{
   ...
}
```

- You should *not* use any try/catch blocks in your code.

Other programming languages. You may use Python, or even `C` or `C++`. However, "starter code" is only available for Java and Python. If you do use `C` or `C++`, you still have to have a Node structure with the same data members as in the "starter code".