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.

A simple idea that does not work

Let us first consider a simple strategy that does not work, in the sense that it does not yield an algorithm that is fast enough. We use the same notation from class and from the "Notes on 2-3 trees" handout for the data layout:

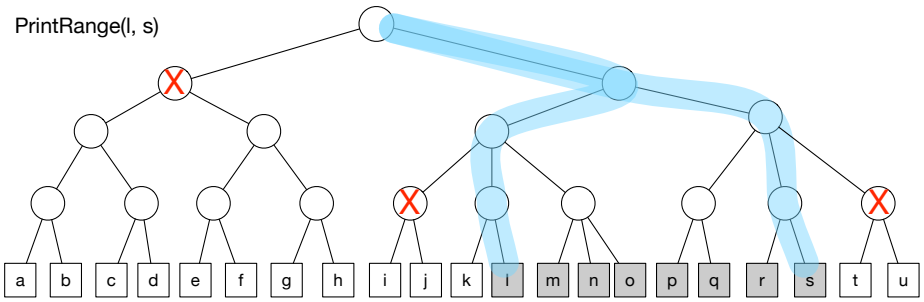
```
class Node {
    KeyType guide;
    // guide is the max key in subtree rooted at node
}
class InternalNode extends Node {
    Node child0, child1, child2;
    // child0 and child1 are always non-null
    // child2 is null iff node has only 2 children
}
class LeafNode extends Node {
    // guide is the key
    ValueType value;
}
```

Consider the following recursive algorithm which is initially invoked as $printRange0(p, x, y, h)$, where p = root of tree and h = height of tree. It prints out the key/value value for all keys in the range $[x, y]$.

```
printRange0(p, x, y, h):
if h = 0 then
    // p points to a leaf node
    if p.guide ∈ [x, y] then print p.guide, p.value
    return

// p points to an internal node
printRange0(p.child0, x, y, h − 1)
printRange0(p.child1, x, y, h − 1)
if p.child2 ≠ null then printRange0(p.child2, x, y, h − 1)
```

The problem with this algorithm is that it visits every node in the tree, so its running time is $O(n)$, instead of $O(m + \log(n))$, as required. To see the problem and one way to fix it, consider the following example:



Here, we want to print all keys in the range $[l, s]$, that is, the keys at the gray shaded leaves. The search paths to l and s are highlighted in blue. You can see there is no point in continuing the recursion at the internal nodes marked with a red X — all of the time spent exploring the descendants of these nodes is wasted.

A slightly less simple idea that *does* work

As the above example illustrates, we would like to "prune" the recursion so that we do not waste time exploring nodes that are to the left of to the right of the search paths to the keys x and y.

This is easy to do. Consider the following recursive algorithm which is initially invoked as *printRange1*(p, x, y, h, lo), where p = root of tree, h = height of tree, and $lo = -\infty$. It prints out the key/value value for all keys in the range $[x, y]$, and it is assumed that all keys below p are *strictly* greater than lo.

printRange1(p, x, y, h, lo):
if $h = 0$ then
    // p points to a leaf node
    if p.guide $\in [x, y]$ then print p.guide, p.value
    return

$hi \leftarrow p$.guide
// p points to an internal node, and all keys below p lie in the interval $(lo, hi]$
if $y \leq lo$ then return *// All leaves below p lie strictly to the right of* $[x, y]$
if $hi < x$ then return *// All leaves below p lie strictly to the left of* $[x, y]$
printRange1$(p$.child0$, x, y, h - 1, lo)$
printRange1$(p$.child1$, x, y, h - 1, p$.child0.*guide*$)$
if p.child2 $\neq$ null then *printRange1*$(p$.child2$, x, y, h - 1, p$.child1.*guide*$)$

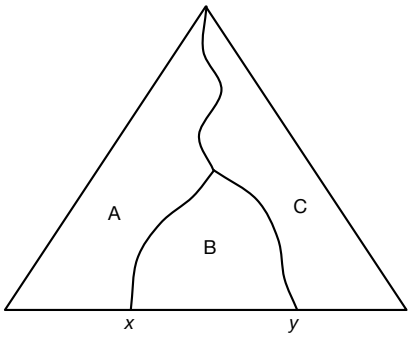
This algorithm works by pruning the recursion as indicated when $y \leq lo$ or $hi < x$. Notice how the value of lo gets updated on the recursive calls.

- When it recurses on p.child1, since the maximum key stored under p.child0 is p.child0.guide, we know that all keys under p.child1 are *strictly* greater than p.child0.guide.
- Similarly, when it recurses on p.child2, since the maximum key stored under p.child1 is p.child1.guide, we know that all keys under p.child2 are *strictly* greater than p.child1.guide.

You may want to run the algorithm on the above example. You will see that the recursion gets pruned exactly at the nodes marked with a red X.

In general, this recursive algorithm will visit nodes on or between the search paths for x and y, which is a total of $O(m + \log(n))$ nodes. The only additional nodes visited are where the pruning occurs, and these are siblings of nodes along the search paths for x and y — so there are only $O(\log(n))$ such nodes.

A general statement. We can make a more general statement about the relationship between the keys *lo* and *hi* associated with an internal node p, and the search paths for x and y. Consider the following diagram:



This represents a 2-3 tree showing search paths to x and y.

- Region A represents all internal nodes strictly to the left of the search path to x.
  Node p is in this region $\iff hi < x$.
- Region B represents all internal nodes strictly between the two search paths.
  Node p is in this region $\iff x \leq lo$ and $hi < y$.
- Region C represents all internal nodes strictly to the right of the search path to y.
  Node p is in this region $\iff y \leq lo$.

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. After you have all that working you should attempt to write the complete program, implementing *printRange1*.

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".