Basic Algorithms:
Guidelines for Programming Assignment 2

To receive full credit, 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 $addRange(x, y, \Delta)$ that adds the value Δ to the values associated with all keys in the interval $[x, y]$.

The goal is to implement this with an algorithm whose running time is $O(\log(n))$ using the "lazy update" strategy discussed in class (and in the 2-3 tree handout).

You should use the following data layout:

```
class Node {
      KeyType guide;
      ValueType value;
}
class InternalNode extends Node {
      Node child0, child1, child2;
}
class LeafNode extends Node { }
```

Compared to Programming Assignment 1, this change essentially moves the value field from the *LeafNode* class to the base class *Node*. Recall that guide value at an internal node is used as a guide to facilitate search, while the guide value at a leaf represents the key stored at that leaf. The starter code you used for that programming assignment will still work with this modified data layout.

Recall that in this "lazy update" strategy, every node in the tree has a value field and the "effective value" associated with any key stored at a leaf is the sum of all the value fields in the nodes on the path from the root to that leaf.

In addition to implementing the *addRange* operation, you also have to implement the *lookup* operation and make a small modification to the *insert* operation in the starter code.

AddRange

Consider the following recursive algorithm which is initially invoked as

$$addRange(p, x, y, h, lo, \Delta),$$

where p = root of tree, h = height of tree, and $lo = -\infty$. It is assumed that all keys below p are *strictly* greater than lo.

addRange(p, x, y, h, lo, Δ):

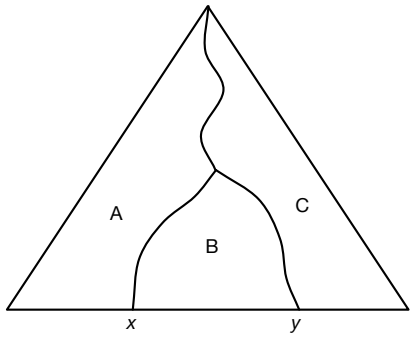
if $h = 0$ then
    // p points to a leaf node
    if p.guide $\in [x, y]$ then p.value $\leftarrow p$.value $+ \Delta$
    return

$hi \leftarrow p$.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* $[x, y]$
if $x \leq lo$ and $hi \leq y$ then
    // All leaves below p are in $(x, y]$
    p.value $\leftarrow p$.value $+ \Delta$
    return
addRange$(p$.child0$, x, y, h - 1, lo, \Delta)$
addRange$(p$.child1$, x, y, h - 1, p$.child0.guide$, \Delta)$
if p.child2 $\neq$ null then *addRange*$(p$.child2$, x, y, h - 1, p$.child1.guide$, \Delta)$

This algorithm works much like the *printRange1* algorithm from Programming Assignment 1. Like *printRange1*, this algorithm works by pruning the recursion as indicated when $y \leq lo$ or $hi < x$. However, it also prunes the recursion when $x \leq lo$ and $hi \leq y$: when this condition occurs, we know that all leaves below p are in the range $(x, y]$, and so we just add Δ to p.value and recurse no further.

In general, this recursive algorithm will visit nodes on the search paths for x and y, which is a total of $O(\log(n))$ 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.

To see in more detail why this works, recall the general statement, presented in the handout for Programming Assignment 1, 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:

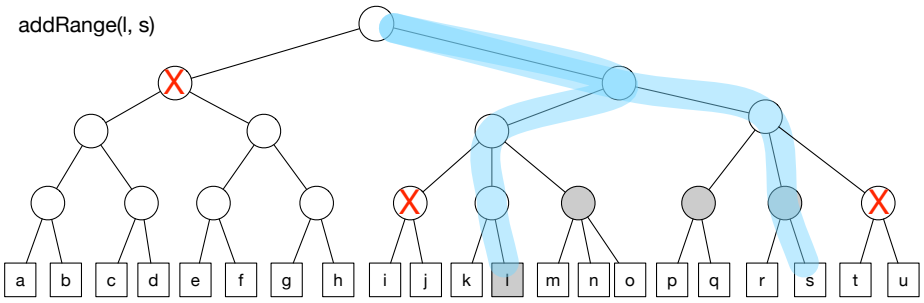


This represents a 2-3 tree showing search paths to x and y.

- 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 $\iff$ $y \leq lo$.

If we wanted to implement directly the strategy outlined in lecture, we would prune the recursion by adding Δ to p.value and stopping the recursion as soon as the recursion enters region B, i.e., when $x \leq lo$ and $hi < y$. However, in the above implementation of *addRange*, we are a bit more aggressive, and prune the recursion when $x \leq lo$ and $hi \leq y$.

Finally, consider again the example from the handout for Programming Assignment 1:



Here, we prune the recursion at the internal nodes marked with a red X, as well as the internal nodes that are shaded gray — the algorithm adds Δ to all the value fields of all of the gray nodes (both leaves and internal nodes). While this example illustrates the mechanics of this algorithm, it is too small to really illustrate how much faster it is than the more straightforward algorithm that would add Δ to the value fields of all the leaves in the given range. However, to avoid time-out errors, it is essential that you use this strategy.

Lookup

You need to implement the lookup operation. Remember that to compute the "effective value" associated with a key, you have to add up all of the values on the search path from the root to the leaf containing that key.

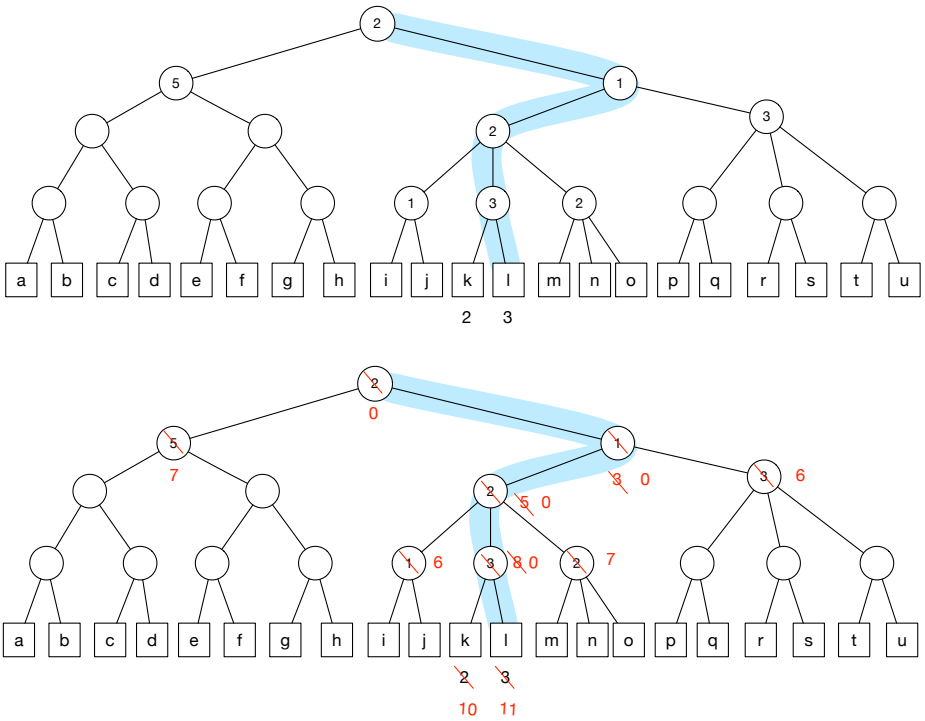
Insertion

You also need to make a small adjustment to the insertion logic. The easiest way to modify the insertion logic is as follows. If you look at the starter code, you see that the insertion

logic invokes a routine *doInsert* that recursively walks down the search path from the root to a leaf. You should modify this routine so that it has the effect of zeroing out the value fields of all internal nodes on this search path as it walks down this path. However, you have to do this in such a way that the "effective values" associated with the keys do not change. To do this, if an internal node p has a nonzero value p.value, you first add p.value to the value fields of each of p's children, and then set p.value to 0. One can easily verify that this has the desired effect. (This strategy is analogous to "shoveling snow".)

Once you zero out the value fields of all internal nodes on the search path, all of the other (rather tricky) logic of insertion (like splitting nodes) takes care of itself. Done right, all of this amounts to adding just a few lines of code to the *doInsert* routine.

Here is an example:



In this example, we are inserting a new key between k and l. The search path would take us to the leaf containing the key l, as highlighted. In the top figure, we have indicated the value associated with each node along this path, as well as a few more for context. In the bottom figure, we have indicated the new values in red — except for the changes indicated, no other value fields in the tree will change. You can see that the root initially had a value of 2, but we clear out the value on the root (i.e., set the value field of the root to 0), and push it onto its children (i.e., we add 2 to the value fields of its children). Now when we come to the next node on the search path, while its value field was initially 1, it is now 3. So again, we clear out the value on this node, pushing it onto its children. We continue in this way all the way down the search path.

Summary

To summarize, for this programming assignment, you need to do the following:

- modify the data layout as discussed above;
- implement the *addRange* function as discussed above;
- implement the *lookup* function as discussed above;
- modify the *doInsert* function as discussed above.