CS2040 Tutorial 7 Suggested Solution

Week 9, starting 10 Oct 2022

Q1 Other BST Operations

You are given a BST implementation below:

```
class Node {
   int item;
   Node left, right;
   Node(int i, Node l, Node r) { item = i; left = l; right = r; }
}
class BST {
   int numNodes;
   Node root;

   int floor(int key) {} // to implement, may create helper method

   void insert(int key) { root = insert(key, root); }
   private Node insert(int key, Node curr) {
      if (curr == null) { numNodes++; return new Node(key, null, null); }
      if (key == curr.item) return curr; // no insertion
      if (key < curr.item) curr.left = insert(key, curr.left);
      else curr.right = insert(key, curr.right);
      return curr;
   }

   void preOrderPrint(Node root) {
      if (root == null) return;
      System.out.print(root.item + " ");
      preOrderPrint(root.left);
      preOrderPrint(root.right);
   }
   void inOrderPrint(Node root) {
      if (root == null) return;
      inOrderPrint(root.left);
      System.out.print(root.item + " ");
      inOrderPrint(root.right);
   }
   void postOrderPrint(Node root) {
      if (root == null) return;
      postOrderPrint(root.left);
      postOrderPrint(root.right);
      System.out.print(root.item + " ");
   }
```

```
    void print() {
        System.out.print("Size: " + numNodes + "\nPreorder: [ ");
        preOrderPrint(root);
        System.out.print("]\nInorder: [ ");
        inOrderPrint(root);
        System.out.print("]\nPostorder: [ ");
        postOrderPrint(root);
        System.out.print("]\n");
    }
}
```

(a) Write another method `int ceil(int key)` in the BST class that finds the *smallest* element that is more than or equals to `key`, or Integer.MAX_VALUE if none exists (i.e. the ceiling).

(b) Can `ceil(int)` be tweaked to implement `int higher(int key)` that returns the *smallest* element strictly greater than `key`, or Integer.MAX_VALUE if none exists? (i.e. the successor)

What is the time complexity of: one call to `higher(int)`, as well as repeated calls to `higher(currentKey)`, starting with `currentKey` being the smallest key in the tree, till we get Integer.MAX_VALUE?

(c) How does the BST implementation need to be changed to support `Node succ(Node curr)` (note the different parameter from (b)), that works similarly to `higher(int)` in (b) but returns the Node instead of the desired element?

Design and implement `succ(Node)`. Why is such an implementation more efficient?

Answer

(a) We can implement this recursively, similar to how binary search can be tweaked to allow similar floor(), or bisect_left() / lower_bound() (in py / C++ respectively) functionality.

When the target is < a node's key, we visit the left subtree but still keep the current node's key as a candidate answer:

```
int ceil(int key) {
    Node ans = ceil(key, root);
    if (ans == null) return Integer.MAX_VALUE;
    return ans.item;
}
private Node ceil(int key, Node curr) {
    if (curr == null) return null;
    if (key == curr.item) return curr; // found ans, since no duplicates
    if (key > curr.item) return ceil(key, curr.right); // curr not valid
    // otherwise target key < curr.item, so curr is a candidate
    Node betterAns = ceil(key, curr.left);
    if (betterAns == null) return curr;
    return betterAns;
}
```

(b) Yes, the only difference is that when the target key is found in a Node's item, the Node is not a candidate solution:

```
int higher(int key) {
   Node ans = higher(key, root);
   if (ans == null) return Integer.MAX_VALUE;
   return ans.item;
}
private Node higher(int key, Node curr) {
   if (curr == null) return null;
   if (key >= curr.item) return higher(key, curr.right); //curr not valid
   // otherwise target key < curr.item, so curr is a candidate
   Node betterAns = higher(key, curr.left);
   if (betterAns == null) return curr;
   return betterAns;
}
```

One call to `higher(int)` runs in O(**h**) time, h being the height of the BST.

When making repeated calls to `higher(int)` from smallest element to largest one key at a time, the algorithm moves down only 1 path from the root each call. Therefore it runs in a total of O(**Nh**) time for a BST with **N** elements.

(c) Each Node needs to be augmented with a parent attribute. BST operations that change the structure of the tree need to maintain this attribute.

```
class BST {
   ...
   private Node insert(int key, Node curr) {
      if (curr == null) { numNodes++; return new Node(key, null, null); }
      if (key == curr.item) return curr; // no insertion
      if (key < curr.item) curr.left = insert(key, curr.left);
      else curr.right = insert(key, curr.right);
      if (curr.left != null) curr.left.parent = curr;
      if (curr.right != null) curr.right.parent = curr;
      return curr;
   }
}
```

Instead of starting from the root each time, we move from node to node, similar to how nodes are conceptually visited in an in-order traversal. When performing in-order traversal, the successor will be the smallest node in the right subtree if exists, otherwise it will be the first ancestor that turns right-ward.

```
class BST {
  ...
  Node succ(Node curr) { // pre-cond: curr not null
    if (curr.right != null) { // leftmost in right subtree
      Node smallest = curr.right;
      while (smallest.left != null) smallest = smallest.left;
      return smallest;
    }
    // no right subtree, so first right-ward ancestor
    Node ancestor = curr.parent;
    while (ancestor != null && ancestor.left != curr) {
      curr = ancestor;
      ancestor = ancestor.parent;
    }
    return ancestor;
  }
}
```

Therefore, while one call to `succ(Node)` still takes O(**h**) time, **N** calls from the smallest key upward take a total of O(**N**) time (better than O(**Nh**) time previously).

Q2 Contiguous Strip with Same Colour

Given positive integers **N** and **K**, suppose you have created a computer game where there is a large 1 x **N** strip of land painted black (colour = 0). The leftmost cell is indexed 0 while the rightmost cell is indexed **N**-1. Each cell has a colour value within 0 to **K** (a positive integer which could be >> **N**) inclusive.

Design and implement a solution to perform the following operations, **each** running in O(log **N**) time or better:

> `void paint(int cell, int newColor)`
> paints the cell at the given index with a different new colour
>
> `int findContigLength(int cell)`
> returns the length of the longest contiguous sub-strip of land with the same colour that includes the (valid) cell with the given index

You may perform some initialization in O(1) time.

e.g. **N** = 5, so the 5 cells start off with colours [0, 0, 0, 0, 0]. findContigLength(1) returns 5
Next paint(3, 5) causes the colours to become [0, 0, 0, 5, 0]. findContigLength(1) returns 3
Then paint is called 3 more times, giving colours [0, 5, 5, 5, 5]. findContigLength(1) returns 4
Finally, paint(3, 4) is called, colours is now [0, 5, 5, 4, 5]. findContigLength(1) returns 2

Answer

Suppose we have the 5 cells shown in the above example, they can be represented as [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0)] where each pair is a (index, colour) pair/object.

If we use an array to store this, paint() will be efficient but findContigLength() will be O(**A**), where **A** is the answer of that operation. In the worst case, O(**A**) can become O(**N**).

We can just store strips that contain contiguous cells with the same colour instead, i.e. [(0, 4, 0)] on the initial state, and [(0, 0, 0), (1, 2, 5), (3, 3, 4), (4, 4, 5)] for the final state in the example. Each pair is now a (startIndex, endIndex, colour) triple/object. Once we find the desired strip, the strip length can be found in O(1) time.

If the colours never change, we can use a sorted (by fromIndex) array and perform binary search. However, strips can be broken (added) and combined (removed) due to colour changes, so a sorted array becomes inefficient. Using a balanced BSTree set helps to achieve O(log **N**) insertions, removals and searches. However, there is a need to **maintain the invariant** that each strip of contiguous cells with the same colour should only appear in 1 element for the operation to be efficient.

To simplify the representation of the data, notice that each toIndex is always the next strip's fromIndex-1. Therefore, we just need to store (fromIndex, colour) pairs/objects [(0, 0), (1, 5), (3, 4), (4, 5)]]. This saves coding time because we can use a balanced BSTree map instead of creating an object to represent the strip.

To prevent null values from appearing when querying near the first and last strip, we can insert dummy/sentinel strips before the first and last cells [(-1, ☻), (0, 0), (1, 5), (3, 4), (4, 5), (5, ☻)].

Operations:

Initialization creates an empty bBST and puts the 2 dummy strips into the bBST

findContigLength(cell) finds floor(cell) and higher(cell) and returns the distance between them

paint(cell, newColor) is tricky. It may create 1 or 2 new strips, and/or remove 1 or 2 existing strips. The trick is to find a way to make different decisions sequentially rather than consider all possibilities at once

```
class Strips {
  TreeMap<Integer, Integer> strips;

  Strips(int N, int K) { // K doesn't affect solution
    strips = new TreeMap<>(); strips.put(0, 0);
    strips.put(-1, -1); strips.put(N, -1); // dummy/sentinel strips
  }

  int findContigLength(int cell) {
    return strips.higherKey(cell) - strips.floorKey(cell);
  }

```

```
  void paint(int cell, int newColor) {
    Integer prevColor = strips.lowerEntry(cell).getValue(),
      existingColor = strips.get(cell);

    strips.put(cell, newColor); // this cell, possibly replacing

    // 3 cases for breaking strips up:
    //  (a) this cell was standalone, or at end of strip - do nothing
    //  (b) not a standalone cell, and at middle of a strip
    if (1 + cell < strips.higherKey(cell) && existingColor == null)
      strips.put(1 + cell, prevColor); // next strip
    //  (c) not a standalone cell, and at front of a strip
    else if (1 + cell < strips.higherKey(cell))
      strips.put(1 + cell, existingColor);

    // check if can combine prev strip
    if (prevColor == newColor)
      strips.remove(cell);

    // next (not mututally exclusive), check if can combine next strip
    Integer nextColor = strips.higherEntry(cell).getValue();
    if (newColor == nextColor)
      strips.remove(1 + cell);
  }
}
```

Question 3 (Online Discussion) – Conditional Average

Modify the BST and or Node class in Q1, without changing the time complexity of any operation, such that it is possible to implement **`double`** `findCondAverage(`**`int`** `upperBound)` that returns, in O(h) time, the average of all elements in the BST that are ≤ upperBound.