CS2040S Semester 1 2023/2024

Data Structures and Algorithms

Tutorial+Lab 07

Table ADT 2: Balanced BST

For Week 09

Document is last modified on: October 17, 2023

MODAL ANSWER IS FOR OUR CLASS ONLY; NOT TO BE DISTRIBUTED IN PUBLIC

1 Introduction and Objective

The purpose of this tutorial is to reinforce the concepts of Binary Search Tree (BST) and the importance of having a balanced BST. In CS2040/C/S, we learn Adelson-Velskii Landis (AVL) Tree as one such possible balanced BST implementation (it is a legacy... Prof Halim learned this AVL Tree when he was an undergraduate, hence this is what he pass down for now... Maybe one day, he will add Red-Black Tree (this is the one that is used inside Java TreeMap and TreeSet instead of AVL Tree) visualization at VisuAlgo and change the lecture content too :).

In this tutorial, we will first discuss bBST augmentations by discussing operations that utilizes its 'ordered' property: Select and Rank, as well as a simple augmentation to make the standard bBST able to handle duplicate elements (already integrated in VisuAlgo /bst page since mid 2023).

Then, we will show (again) the versatility of balanced BST data structure as an alternative implementation of Priority Queue ADT that we have learned earlier.

We will also discuss balanced BST versus Hash Table (discussed in the previous tutorial) as implementation for Table ADT.

2 Tutorial 07 Questions

Basic Operations of (balanced) Binary Search Tree: AVL Tree

Q1). (Optional, only when many are still not comfortable with basic bBST operations): We will start this tutorial with a quick review of basic BST operations, but on a balanced BST: AVL Tree. The

tutor will first open `https://visualgo.net/en/avl`, click Create → Random. Then, the tutor will ask students to Search for some integers, find Successor of existing integers, perform Inorder Traversal, Insert a few random integers, and also Remove existing integers (details in Q2).

This part is open ended, up to the tutor, and can be totally skipped (or only briefly mentioned) if the tutor detects that most students are roughly okay with basic bBST tasks (test with simple deletion case(s) that will later be discussed in Q2).

If need be, the review of basic operations can be done quickly and the tutor has to be creative enough to ask for various corner cases, starting from operations that do not change the underlying BST:

1. Search, vary the request between existing versus non-existing integer, and close to root versus as far as possible. Also discuss a new variant: lower_bound(v), that slightly differs from Search(v) when v does not exist.

2. Successor, actually the tutor will ask students to answer Predecessor operation instead to verify understanding of this mirror operation of Successor. The tutor will vary the request between vertex that has Predecessor or not (has left child versus has no left child) and the tutor will also ask about the minimum element (it has no Predecessor). Optional: Discuss the subtle difference between these two strategies to find Successor of v if v has no right child: A). go up to parent(s) until we encounter a right turn (the current setup in VisuAlgo – but this REQUIRES us to maintain parent pointers) versus start from the root, search for v. If we go left, we keep the current vertex as 'potential successor' (the one currently implemented in BSTDemo.cpp – this eliminates the need to maintain parent pointers).

3. Inorder traversal, trivial. Show them shortcut that just echo the content of the BST in sorted order instead of manually performing Inorder traversal. The output of Inorder traversal of a BST is guaranteed to be sorted. Also take this opportunity to introduce two quick variants: Pre-order traversal and Post-order traversal of BST (or any tree) structure.

Then, the tutor can ask about two operations that modify the content of the bBST that may trigger rotation(s) and explain those rotation(s) cases accordingly:

1. Insert, vary the request between inserting to new leaf close to root versus as far as possible and whether such insertion triggers any rotation (one of the four cases).

2. Delete/Remove, vary the request between the three deletion cases (leaf vertex, vertex with one child, vertex with two children), but defer asking deletion that causes rotation to Q2 below that is specifically asked so that students put more thoughts on such delete operation.

Q2). Draw a valid AVL Tree and nominate a vertex to be deleted such that if that vertex is deleted:
a). **No rotation** happens
b). **Exactly one** of the four rotation (L, R, LR, RL) cases happens
c). **Exactly two** of the four rotation cases happens (you can**not** use the sample given in VisuAlgo which is `https://visualgo.net/en/bst?mode=AVL&create=8,6,16,3,7,13,19,2,11,15,`

18,10, delete vertex 7; think of your own test case)
d). **Exactly three** of the four rotation cases happens

This part is also open ended and there are many possible answers. The tutor will help students construct the (bigger) answer based on the idea of minimum-sized AVL Tree, i.e., https://visualgo.net/en/bst?mode=AVL&create=21,13,29,8,18,26,32,5,11,16,20,24,28,31,33,3,7,10,12,15,17,19,23,25,27,30,2,4,6,9,14,22,1 and then delete 33.

Extra BST Operations (After Augmentations)

Q3). There are two important bBST operations: Select and Rank that have just been included in VisuAlgo (during May-July 2023 holiday, see https://visualgo.net/en/bst?slide=5-1) but can be quite useful for some **order-statistics** problems. Please discuss the details on how to implement these two operations efficiently after we augment each bBST vertex with an 'extra attribute'.

The key part to make this work is to augment our bBST (e.g., AVL Tree) with one more important information at each vertex: Size of subtree below (and including) that vertex. bBST that is augmented with that 'size' attribute can be modified to support rank and select operations in $O(\log n)$ time (at the moment, this feature can be seen in the lower_bound(v) animation. Here are their pseudo code; some minor detail involving null node and value v that does not exist in the bBST are not considered yet. PS: If the tree is not balanced, these two operations may degenerate back to $O(n)$.

```
int rank(node, v) { // assume that v exists in the BST and size attribute is there
       if (node.key == v) return node.left.size + 1; // found
  else if (node.key <  v) return rank(node.left, v); // v must be on the left
  else                    return node.left.size+1 // v is > node's left and the node
                                 + rank(node.right, v); // and plus this rank
}

// select is very similar to rank and similar with QuickSelect from tut01... :O
int select(node, k) { // assume size attribute is there
  int q = node.left.size;
       if (q+1 == k) return node.key; // this node has rank k
  else if (q+1 >  k) return select(node.left, k); // rank k is in the left subtree
  else               return select(node.right, k-q-1); // do you understand why?
}
```

Q4). What if there are duplicate elements in our bBST? Please discuss on how to implement this feature efficiently after we augment each bBST vertex with yet another 'extra attribute' (different from above). Prof Halim has just added this feature at https://visualgo.net/en/bst during May-July 2023 holiday.

The easiest answer is to augment our bBST (e.g., AVL Tree) with one more important information

at each vertex: the frequency counter of that vertex. If we insert a duplicate of a certain vertex that already exists in our bBST, we simply increase its frequency by one. If we delete a duplicate of a certain vertex that has more than one occurrence in our bBST, we simply decrease its frequency by one. If we insert a totally new item into our bBST, we do a proper bBST insertion and set its frequency to 1. If we delete an existing item in our bBST with frequency 1, this is when we do a proper bBST deletion. This easiest answer is the result of a few iterations of trying to figure out what is the best way to handle duplicates in bBST (this variant is not commonly found in other CS textbooks on bBST).

Binary Heap... or Not? (Quick Review)

Q5 and Q6 are just quick review of an interesting topic that has been discussed in lecture.

Q5). We know that Binary (Max) Heap can be used as Priority Queue and can do `ExtractMax()` in $O(\log n)$ time. What modifications/additions/alterations are required so that *both* `ExtractMax()` and `ExtractMin()` can be done in $O(\log n)$ time for the set of n elements and every other Priority Queue related-operations, especially Insert/Enqueue retains the same $O(\log n)$ running time? The elements are not necessarily distinct. Hint: What is the topic of this tutorial?

There is a long answer that uses two Binary Heaps... but it is super complex to explain, so I will skip that.

Competitive Programmer answer is thus: Just use a (balanced) BST to model the Priority Queue. The keys in this bBST are the priority values. Then, to find the minimum/maximum element, we just need to call $O(\log n)$ `findMin()`/`findMax()`. Insertion/Enqueue to bBST is still $O(\log n)$. Notice that we do NOT have to associate `PriorityQueue` ADT to be always a `Binary Heap` data structure! We will have discussed BST and balanced BST by the time we discuss this tutorial.

Tutors are expected to use `https://visualgo.net/en/bst` and use the Search: Find Minimum/Find Maximum functionality to support this discussion. Throw in Insert (new minimum) and/or (new maximum) and do another Find Minimum/Find Maximum.

This solution still works in the presence of duplicates.

Q6). Quick follow up from the question. above:
Now revisit Q6). of Tut04 (Priority Queue ADT Tutorial).
Would you answer that question differently now?
Competitive Programmer answer: Same as above, just use a bBST to model the Priority Queue.
Insert to PQ = Insert to bBST $O(\log n)$,
Delete from PQ (any position) = Delete from bBST $O(\log n)$,
Update (Increase or Decrease) Key of PQ = Delete the old value and insert the new value to the bBST :O $O(2 \log n = \log n)$,

Get Max = FindMax of bBST, $O(\log n)$ (similarly for Get Min = Find Min of bBST).
All operations are still in order of $O(\log n)$...

Hash Table or Balanced BST?

Q7). As of now, you have been exposed with both possible implementations of Table ADT: Hash Table (and its variations) and BST (including Balanced BST like AVL Tree). Now write down four potential usage scenarios of Table ADT. Two scenarios should favor the usage of Hash Table whereas for the other two scenarios, using Balanced BST is better.

Possible usage scenarios that favor Hash Table (usually Separate Chaining):

1. Pure key to value mappings **without** needing the keys to be in ordered fashion,
2. When even the small gap between $O(\log N)$ versus $O(1)$ matters for your application (and again, there is no need for the keys to be ordered). PS: If there is no need to delete key, switch Separate Chaining to Open Addressing (Double Hashing) for an even faster $O(1)$ performance.

Possible usage scenarios that favor Balanced BST (e.g. AVL):

1. Pure key to value mappings **with** condition that the keys have to be in ordered fashion,
2. Finding min/max keys, finding k-th smallest/largest key, finding the next largest/previous smaller keys of a certain given key, basically operations that require the keys to be ordered, and new keys can by dynamically added/updated/deleted later,
3. Using Balanced BST for another sorting algorithm Tree Sort: We can insert N items into a bBST in $O(N \log N)$ time and then output the content of bBST using $O(N)$ inorder traversal. If the content of the bBST is never updated again, this is overkill. But if the content of the bBST is frequently updated: dynamic data structure (new element added, existing element deleted, or existing values updated), then bBST is probably the way to go.
4. Using Balanced BST as Priority Queue like in Q3-Q4 (similar to sorting on a dynamic data structure above, but concentrating on min/max elements).

Hands-on 7

TA will run the second half of this session with a few to do list:

- PS4 Debrief (Quick one)
- Very quick review of Java TreeSet and TreeMap,
- Do a(nother) sample speed run of VisuAlgo online quiz that are applicable so far, e.g., `https://visualgo.net/training?diff=Medium&n=5&tl=5&module=bst`
- Then, live solve another chosen Kattis problem involving BST/AVL Tree/augmented BST...

Summary of your findings of PS4,
1-2 slides for Java API demonstrations (shorter than last week, as they are technically 'identical'), just show a few techniques or refer students to relevant references, e.g., `https://github.com/stevenhalim/cpbook-code/blob/master/ch2/nonlineards/map_set.java`.
Optional: Again, you may want to share the existence of the multimap/multiset versions.
Show off in 5m :), again (on another set of random questions).
Kattis /courseschedulingh, an easy bBST (TreeMap) application (map String (course code) to an HashSet of Strings (student names, can have duplicates))

Problem Set 5

We will end the tutorial with **high-level** discussion of PS5.
As usual, we still have next week, so the official discussion is shorter.

For PS5A (/kannafriendship), let's start with subtask 1 first: $N \leq 2000$ and $Q \leq 2000$, an $O(QN)$ brute force algorithm using Direct Addressing Table (Boolean array) data structure should be OK. If students can get past that, then let's consider subtask 2: $N \leq 200\,000$ but $s_i = e_i$ for all type 1 queries (so type 1 query runs in $O(1)$). We can make type 2 queries also run in $O(1)$ by updating a counter during type 1 updates. The last two subtasks are reserved for the next tutorial.

For PS5B (/traveltheskies), you will have to understand Lecture 9a - introduction to Graph and Graph Data Structure first. You need another 'combo' data structure, then simulate day to day flights. Hint: n Adjacency Lists (one for each day). For AL of day d, we take note of flight between airport u to airport v with capacity z.