AVL Trees

Recall the operations (e.g. `search`, `insert`, `delete`) of a binary search tree. The runtime of these operations were all $O(h)$ where h represents the height of the tree, defined as the length of the longest branch. In the worst case, all the nodes of a tree could be on the same branch. In this case, $h = n$, so the runtime of these binary search tree operations are $O(n)$. However, we can maintain a much better upper bound on the height of the tree if we make efforts to balance the tree and even out the length of all branches. An AVL trees is a binary search tree that balances itself every time an element is inserted or deleted. In addition to the invariants of a BST, **each node of an AVL tree has the invariant property that the heights of the sub-tree rooted at its children differ by at most one**, i.e.:

$$|\texttt{height(node.left)} - \texttt{height(node.right)}| \leq 1 \tag{1}$$

Height Augmentation

In AVL trees, we augment each node to keep track of the node's height.

```
1  def height(node):
2      if node is None:
3          return -1
4      else:
5          return node.height
6
7  def update_height(node):
8      node.height = max(height(node.left), height(node.right)) + 1
```

Every time we insert or delete a node, we need to update the height all the way up the ancestry until the height of a node doesn't change.

Note that augmenting the nodes with additional information (depending on the problem at hand) can be a nice algorithmic technique for solving problems. Examples of these are provided on the practice problem on these recitation notes.

AVL Insertion, Deletion and Rebalance

We can insert a node into or delete a node from a AVL tree like we do in a BST. But after this, the height invariant (1) of the AVL tree may not be satisfied any more.

For insertion, there are 2 cases where the invariant will be violated:

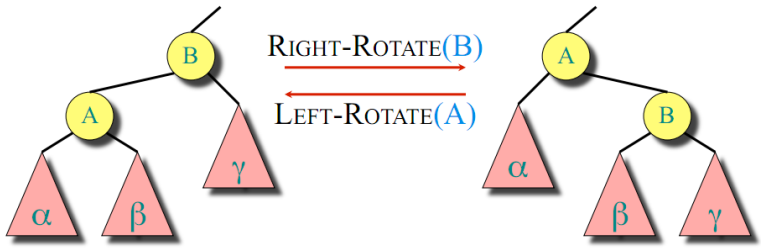
1. The left child of node x is heavier than the right child. Inserting into the left child may imbalance the AVL tree.

2. The right child of node x is heavier than the left child. Inserting into the right child may imbalance the AVL tree.

For deletion, the cases are analogous, however, you are not expected to know deletion in detail.

So we need to rebalance the tree to maintain the invariant, starting from the node inserted (or the parent of the deleted node) and continue up.

There are two operations needed to help balance an AVL tree: a left rotation and a right rotation. Rotations simply re-arrange the nodes of a tree to shift around the heights while maintaining the order of its elements. Making a rotation requires re-assigning `left`, `right`, and `parent` of a few nodes, and **updating their heights**, but nothing more than that. Rotations are $O(1)$ time operations.




```
1  def rebalance(self, node):
2      while node is not None:
3          update_height(node)
4          if height(node.left) >= 2 + height(node.right):
5              if height(node.left.left) >= height(node.left.right):
6                  self.right_rotate(node)
7              else:
8                  self.left_rotate(node.left)
9                  self.right_rotate(node)
10         elif height(node.right) >= 2 + height(node.left):
11             if height(node.right.right) >= height(node.right.left):
12                 self.left_rotate(node)
13             else:
14                 self.right_rotate(node.right)
15                 self.left_rotate(node)
16         node = node.parent
```

```
1  def insert(self, k):
2          node = super(AVL,self).insert(k)
3          self.rebalance(node)
```

Note that `rebalance` includes `upate_height` as well.

Maintaining Size

To maintain the size of each node in an AVL tree, we must slightly alter how we insert and delete from the tree. When we insert a new node, we climb down the tree starting from the root and end up stopping where we insert our new node. We can maintain size by incrementing the size of each node we see along the way. This is enough for a normal BST, but if we have an AVL tree, we must also make corrections when we call rotate. We do a similar thing when we delete a node, but with decrementing size.

Correctness and Runtime

Correctness of insert follows from the correctness that rotations does not violate the Binary Search Tree (BST) invariant, so nodes that are smaller are still on the left and bigger are still on the right subtree. Rotations correct the current height difference, so they at least correct height differences locally. Furthermore, rotations only change the height by at most 1, so as the rebalancing is done, rotations do not make keeping the AVL tree invariant harder. Rebalancing does not continue indefinitely because it stops at the root if necessary.

Runtime of insert (and delete) might be $O(h) = O(\log n)$ because the rebalancing might have go up to the root in the worst case.

Quick Review of Height of an AVL tree

Let $n_h =$ be the minimum number of nodes of an AVL tree of height h. Then we have:

$$n_h \geq 1 + n_{h-1} + n_{h-2} \implies n_h > 2n_{h-2}$$

$$n_h > 2^{\frac{h}{2}} \implies h < 2 \log n_h$$

so the height of the tree is $O(\log n)$.

AVL Sort

You can easily sort a list of n elements by creating an AVL tree, inserting each element into the AVL tree, then doing an in-order traversal of the AVL tree. Note that once you have constructed an AVL tree, the in-order traversal takes $O(n)$ time. Unfortunately, to actually construct the AVL tree takes $O(n \lg n)$ time, so using an AVL to sort a list takes $O(n \lg n)$ time, just like merge-sort and heap-sort.

It's interesting to note that the time-consuming part of AVL-sort is constructing the AVL tree, and doing a traversal to get the sorted list is quick once the tree is constructed. It is the opposite case for heaps: to construct a heap is fast, but to get the sorted list from a heap takes $O(n \lg n)$ time. It may help to think of the elements in a binary search tree as more organized than the elements in a heap: because BSTs have more restrictions on the ordering of their elements, they take longer to construct, but are also more powerful.

Practice Problems

Selection with AVL Trees

Suppose you have an AVL tree and you want to define the function SELECT(k) which finds and returns the k^{th} smallest element in the tree. The naive approach is to find the minimum element, and then call successor $k-1$ times. This has a worse-case running time of $O(n)$. We can do better by augmenting the AVL tree.

Augmenting with subtree size

Define the 'size' of each node as the total number of nodes in that node's subtree, including that node. If each node keeps track of its size, then you can easily determine the k^{th} element by walking down the tree in time $O(\lg n)$.

```
1  def select(self, node, k):
2
3      if node.left == None:
4        left_size = 0
5      else:
6        left_size = node.left.size
7
8      if k-1 == left_size:
9        return node
10     elif k-1 < left_size:
11       return select(node.left, k)
12     else:
13       return select(node.right, k - left_size - 1)
```

Overpaid employees

1. *Overpaid employees*: your company stores a database of employees, with relevant salary and age information for each employee. Because your company is not doing too well financially you need to cut down the salaries of the "overpaid" employees. To do so, you decide to identify the employees that are "overpaid" based on their age. You want to quickly find the employee of age $\leq A$ with the highest salary. Design a data structure that has the following operations:

   (a) **Insert**(a, s): insert a new employee with age a and salary s.

   (b) **Delete**(a, s): delete the employee with age a and salary s (if it exists in your database).

   (c) **MaxSal**(a): return the largest salary amongst all employees whose age is $\leq a$. If there is no such employee return -1.

   All operations must run in $O(\log n)$ time.

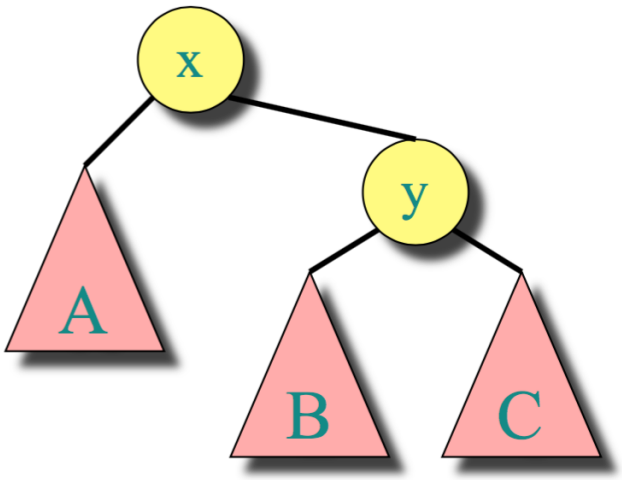
Augment with largest salary and key with age

The main idea to solve the Overpaid employees is to have an augmentation of the AVL tree. The augmentation is to have the key of the nodes to be the age and to have the nodes keep track of the largest salary its subtree. When one inserts a new node (a, s) one simple inserts by age as in a normal AVL tree but simply updates the largest salary in the subtree if necessary. To delete, one also needs to keep track of the salary of each node, however, note that the salary isn't the key of the AVL tree. Finally, to find the max salary for all ages bellow a certain limit a, one simply needs to traverse the tree to find a (if it is in the tree) and at the same time make sure to keep track of the

max salaries of everything less than a as we traverse. All of these operations are simple traversals of the AVL tree which take $O(\log n)$ time.

Review of the cases with some detail

Let x be the lowest node that violates the AVL invariance. We will fix the subtree of x and move up. WLOG, assume the right child of x is deeper than the left child of x (x is "right-heavy").

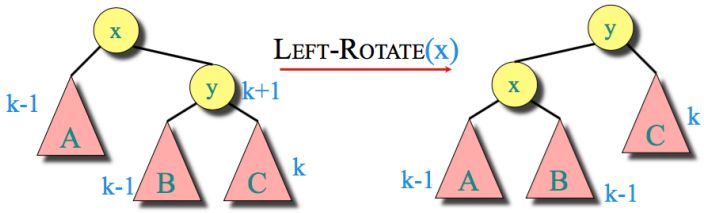


If x is right-heavy then there are 3 scenarios:

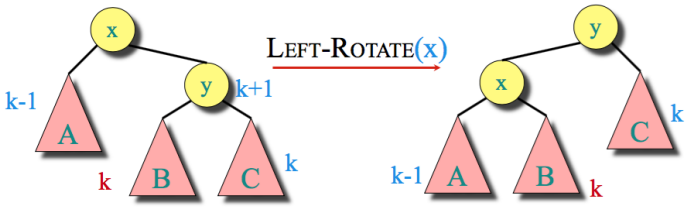
1. Case 1: Right child y of x is right-heavy
2. Case 2: Right child y of x is balanced.
3. Case 3: Right child y of x is left-heavy.

The cases and the re-balancing/rotations will be described with the following diagrams:

Case 1: y is right-heavy



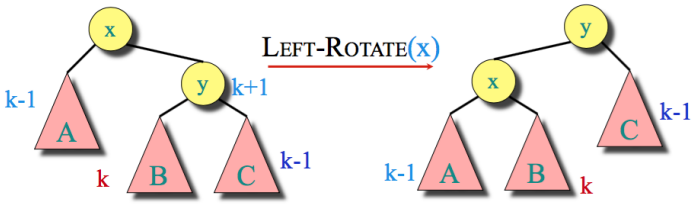
Case 2: y is balanced



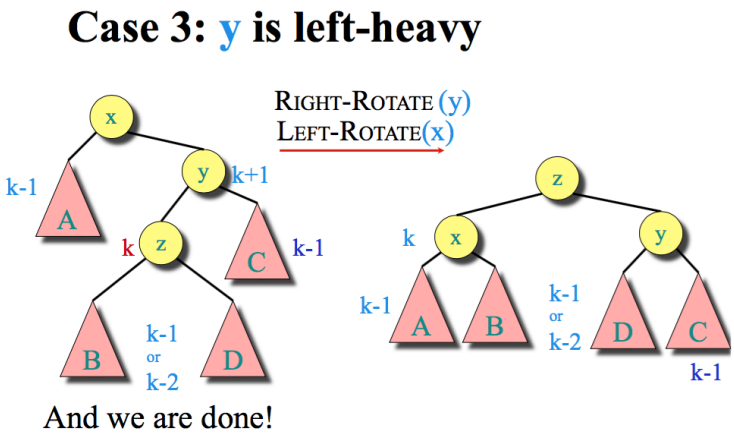
Same as Case 1

When the child of y is left-heavy we need 2 rotations instead of 1 as illustrated as follows:

Case 3: y is left-heavy



Need to do more …



1 Example AVL tree

Example diagrams can be found as a separate PDF on stellar.

For an example, it is common to consider the height of a single node as 0 and -1 for an empty subtree (i.e. a point to nil/null has height -1). Do the following operations:

1. insert(1)
2. insert(2)
3. insert(3). There is a violation of the AVL property. Thus, you rebalance using case 1.
4. insert(4)
5. insert(2.5)
6. insert(2.2). There is a violation of the AVL property. Thus, you rebalance using case 3 (double rotation).s

Notice that you have to make sure that the heights are maintained correctly. Only nodes on the path from insertion point to root node have possibly changed in height. Every time we insert (or delete a node), we need to update the height all the way up the ancestry until the height of a node does not change.

Notice that case 2 wasn't possible to trigger using only insertions because since to get that state we need B and C to have the same height $k-1$ and increase both by 1 with one single insert. Regardless case 2 has the same solutions as case 1.

Sidebar: Data Structures

A **data structure** is a collection of algorithms for storing and retrieving information. The operations that store information are called **updates**, and the operations that retrieve information are called **queries**. For example, a sorted array supports the following operations:

1. Queries: MIN(), MAX(), SEARCH(x)
2. Updates: INSERT(x), DELETE(x)

The salient property of a data structure is its **representation invariant** (**RI**), which specifies how information is stored. Formally, the representation invariant is a predicate which must always be true for the data structure to function propertly. The query operations offered by the data structure are guaranteed to produce the correct result, as long as the representation invariant **holds** (is true). Update operations are guaranteed to **preserve** the representation invariant (if the RI holds before the update, it will also hold after the update).

For example, a sorted array's representation invariant is that it stores keys in an array, and the array must always be sorted. SEARCH is implemented using the binary search algorithm, which takes $O(\log(N))$ time. SEARCH is guaranteed to be correct if the RI holds (the array is sorted). On the other hand, INSERT must preserve the RI and its running time is $O(N)$. INSERT's worst-case input is a key that is smaller than all the keys in the array, as it will require shifting all the elements in the array to the right.

When implementing (and debugging) a data structure, it is useful to write a `check_ri` method that checks whether the representation invariant is met, and raises an exception if that is not the case. While debugging the data structure, every operation that modifies the data structure would call `check_ri` right before completing. This helps you find a bug in your implementation as soon as it happens, as opposed to having to track it down based on incorrect query results. Because it is only used during debugging, `check_ri` can be slower than the data structure's main operations. For example, checking a binary heap's representation invariant takes $O(N)$ time, whereas the usual query (MIN) and update operations (INSERT, UPDATE, EXTRACT-MIN) take $O(\log(N))$ time.

When building a software system, you should stop and think for a bit about the data structures that can perform each task efficiently. Once you have some data structures in mind, you can design the API (interface) between the module implementing the task and the rest of the system, in a way that would allow the module to be implemented using any of the efficient data structures. Once the API is in place, you should initially choose the data structure with the simplest implementation, to minimize development time. If you need to optimize your system later, you will be able to switch in a more efficient data structure easily, because you thought of that possibility in advance, when designing the module's API.