AVL Trees

We have already seen one implementation of the *Dictionary* ADT that works well in practice—hash tables. You have very probably already met binary search trees (they are discussed below). Both of these data structures have a worst case running time of $\Theta(n)$ for findElement, insertItem and removeItem. In this lecture note, we study a *balanced* binary search tree that implements the methods findElement, insertItem and removeItem of *Dictionary* with a worst-case running time of $\Theta(\lg(n))$ for the methods. The balanced binary trees studied here were invented in 1962 by G. Adelson-Velskii and E.M. Landis and are called the AVL Trees.

5.1 Binary Search Trees

A tree is *binary* if each of its vertices either has two children, in which case it is called an *internal vertex*, or no children at all, in which case it is called a *leaf* or an *external vertex*. We refer to the two children of an internal vertex v as the *left* child and *right* child of v. The vertices of a search tree store *items* consisting of *keys* and *elements*. It is convenient to set up the tree in such a way that only the internal vertices store items[1]. The keys are taken from an ordered set (in Java, keys must be of some `Comparable` type). A binary tree storing items in its internal vertices is a *binary search tree* if for every vertex v, all keys stored in the left subtree of v (rooted at the left child of v) are less than or equal to the key of v and all keys stored in the right subtree are greater than or equal to the key of v. (If we do not allow duplicate keys then the inequalities are strict.)

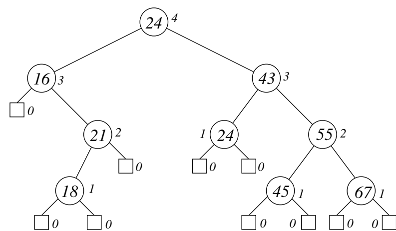

Figure 5.1. A binary search tree. The keys are shown inside the vertices, and the heights are shown next to the vertices.

Figure 5.1 shows an example of a binary search tree. In all our examples, the keys are natural numbers, and we show only the keys of the vertices.

[1] This may seem like a waste of space, but when implementing a search tree we can just represent the leaves as null references.

We now present the implementation of findElement() on any binary search tree.

Algorithm findElement(k)

1. **if** isEmpty(T) **then return** NO_SUCH_KEY
2. **else**
3. $u \leftarrow root$
4. **while** ((u is not null) **and** $u.key \neq k$) **do**
5. **if** ($k < u.key$) **then** $u \leftarrow u.left$
6. **else** $u \leftarrow u.right$
7. **od**
8. **if** (u is not null) **and** $u.key = k$ **then return** $u.elt$
9. **else return** NO_SUCH_KEY

Algorithm 5.2

Recall that the *height* of a vertex v of a tree is the maximum length over all paths from v to a leaf of the tree, obtained by following left or right child vertices (we count the number of vertices on a path but not the leaf). The *height* $h = h(T)$ of a tree T is the height of the root. Figure 5.1 shows the height of each vertex.

Examining Algorithm 5.1, it should be clear that, since we walk one step further away from the root with every iteration of the loop at lines 4–7, the worst-case running time of findElement is $O(h)$, where h is the height of the tree. Hence we have:

Theorem 5.3. *For any binary search tree implementation of the Dictionary ADT, findElement has asymptotic worst-case running time $O(h)$, where h is the height of the tree.*

In fact the basic insertItem and removeItem implementations on a (not necessarily balanced) binary search tree also achieve worst-case running time $O(h)$. The problem is that binary search trees can be completely *unbalanced*. In the degenerate case, a tree storing n items can have height n, in which case it essentially looks like a linked list.

In §§5.3 and 5.4 we will describe special implementations of insertItem and removeItem on an AVL tree, which have the affect of keeping the height of the AVL tree bounded by $O(\lg(n))$, and hence guarantee that findElement, insertItem and removeItem all run in $O(\lg(n))$ time.

5.2 AVL trees

We now define the special balance property we maintain for an AVL tree.

Definition 5.4.

(1) A vertex of a tree is *balanced* if the heights of its children differ by one at most.

(2) An *AVL tree* is a binary search tree in which *all* vertices are balanced.

The binary search tree in Figure 5.1 is not an AVL tree, because the vertex with key 16 is not balanced. The tree in Figure 5.5 is an AVL tree.

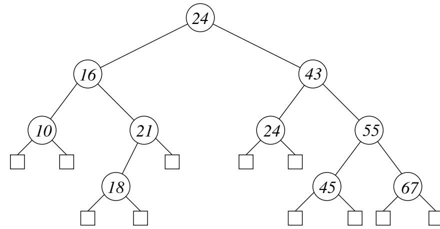

Figure 5.5. An AVL tree.

We now come to our key result.

Theorem 5.6. *The height of an AVL tree storing n items is $O(\lg(n))$.*

PROOF For $h \in \mathbb{N}$, let $n(h)$ be the minimum number of items stored in an AVL-tree of height h. Observe that $n(1) = 1$, $n(2) = 2$, and $n(3) = 4$.

Our first step is to prove, by induction on h, that for all $h \geq 1$ we have

$$n(h) > 2^{h/2} - 1. \tag{5.1}$$

As the induction bases, we note that (5.1) holds for $h = 1$ and $h = 2$ because $n(1) = 1 > \sqrt{2} - 1$ and $n(2) = 2 > 2^1 - 1$.

For the induction step, we suppose that $h \geq 3$ and that (5.1) holds for $h - 1$ and $h - 2$. We observe that by Defn 5.4, we have

$$n(h) \geq 1 + n(h-1) + n(h-2),$$

since one of the two subtrees must have height at least $h-1$ and the other height at least $h-2$. (We are also using the fact that if $h_1 \geq h_2$ then $n(h_1) \geq n(h_2)$, this is very easy to prove and you should do this.) By the inductive hypothesis, this yields

$$\begin{aligned} n(h) &> 1 + 2^{\frac{h-1}{2}} - 1 + 2^{\frac{h-2}{2}} - 1 \\ &= (2^{-\frac{1}{2}} + 2^{-1})\, 2^{\frac{h}{2}} - 1 \\ &> 2^{\frac{h}{2}} - 1. \end{aligned}$$

The last line follows since $2^{-\frac{1}{2}} + 2^{-1} > 1$. This can be seen without explicit calculations as it is equivalent to $2^{-\frac{1}{2}} > 1/2$ which is equivalent to $2^{\frac{1}{2}} < 2$ and this now follows by squaring. This completes the proof of (5.1).

Therefore for every tree of height h storing n items we have $n \geq n(h) > 2^{h/2} - 1$. Thus $h/2 < \lg(n+1)$ and so $h < 2\lg(n+1) = O(\lg(n))$. This completes the proof of the Theorem. □

It follows from our discussion of findElement that we can find a key in an AVL tree storing n items in time $O(\lg(n))$ in the worst case. However, we cannot just insert items into (or remove items from) an AVL tree in a naïve fashion, as the resulting tree may not be an AVL tree. So we need to devise appropriate insertion and removal methods that will maintain the balance properties set out in Definition 5.4.

5.3 Insertions

Suppose we want to insert an item (k, e) (a key-element pair) into an AVL tree. We start with the usual insertion method for binary search trees. This method is to search for the key k in the existing tree using the procedure in lines 1–7 of findElement(k), and use the result of this search to find the "right" leaf location for an item with key k. If we find k in an internal vertex, we must walk down to the largest near-leaf in key value which is no greater than k, and then use the appropriate neighbour leaf. In effect we ignore the fact that an occurrence of k has been found and carry on with the search till we get to a vertex v after which the search takes us to a leaf l (see Algorithm 5.1). The vertex v is the "largest near-leaf." We make a new internal vertex u to replace the leaf l and store the item there (setting $u.key = k$, and $u.elt = e$).

The updating of u from a leaf vertex to an internal vertex (which will have two empty child vertices, as usual) will sometimes cause the tree to become unbalanced (no longer satisfying Definition 5.4), and in this case we will have to repair it.

Clearly, any *newly unbalanced* vertex that has arisen as a result of inserting into u must be on the path from the new vertex to the root of the tree. Let z be the unbalanced vertex of minimum height. Then the heights of the two children of z must differ by 2. Let y be the child of z of greater height and let x be the child of y of greater height. If both children of y had the same height after the insetion, then z would already have been unbalanced before the insertion, which is impossible, because before the insertion the tree was an AVL tree. Note that the newly added vertex might be x itself, or it might be located in the subtree rooted at x. Let V and W be the two subtrees rooted at the children of x. Let X be the subtree rooted at the sibling of x and let Y be the subtree rooted at the sibling of y. Thus we are in one of the situations displayed in Figures 5.7–5.10(a). Now we apply the operation that leads to part (b) of the respective figure. These operations are called *rotations*; consideration of the figures shows why.

By applying the rotation we balance vertex z. Its descendants (i.e., the vertices below z in the tree before rotation) remain balanced. To see this, we make the following observations about the heights of the subtrees V, W, X, Y:

- height$(V) - 1 \leq$ height$(W) \leq$ height$(V) + 1$ (because x is balanced). In the Figures, we have always assumed that W is the higher tree, but it does not make a difference.

- max{height(V), height(W)} = height(X) (as y is balanced and height$(x) >$ height(X)).

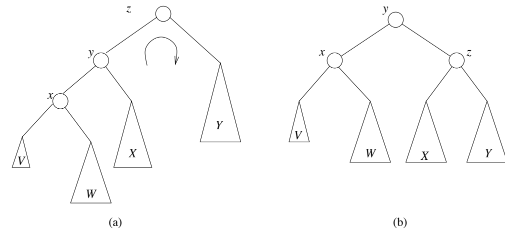

Figure 5.7. A clockwise single rotation

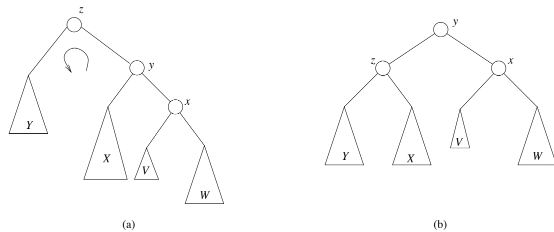

Figure 5.8. An anti-clockwise single rotation

- $\max\{\text{height}(V), \text{height}(W)\} = \text{height}(Y)$ (because $\text{height}(Y) = \text{height}(y) - 2$).

The rotation reduces the height of z by 1, which means that it is the same as it was before the insertion. Thus all other vertices of the tree are also balanced.

Algorithm 5.11 overleaf summarises $\text{insertItem}(k, e)$ for an AVL-tree. In order to implement this algorithm efficiently, each vertex must not only store references to its children, but also to its parent. In addition, each vertex stores the height of its left subtree minus the height of its right subtree (this may be -1, 0, or 1).

We now discuss $T_{\text{insertItem}}(n)$, the worst-case running time of insertItem on a AVL tree of size n. Let h denote the height of the tree. Line 1 of Algorithm 5.11 requires time $O(h)$, and line 2 just $O(1)$ time. Line 3 also requires time $O(h)$, because in the worst case one has to traverse a path from a leaf to the root. Lines 4.-6. only require constant time, because all that needs to be done is redirect a few references to subtrees. By Theorem 5.3, we have $h \in O(\lg(n))$. Thus the overall asymptotic running time of Algorithm 5.11 is $O(\lg(n))$.

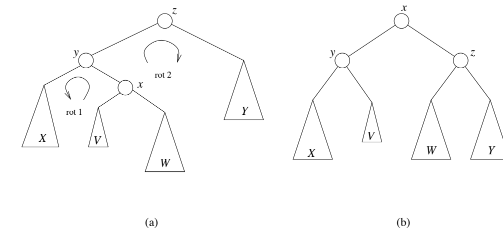

Figure 5.9. An anti-clockwise clockwise double rotation

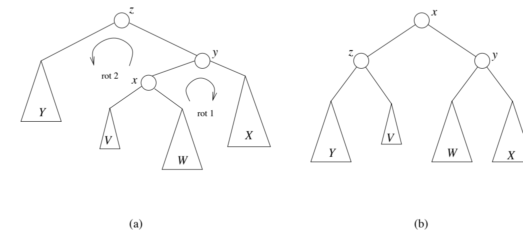

Figure 5.10. A clockwise anti-clockwise double rotation

5.4 Removals

Removals are handled similarly to insertions. Suppose we want to remove an item with key k from an AVL-tree. We start by using the basic removeItem method for binary search trees. This means that we start by performing steps 1–7 of $\text{findElement}(k)$ hoping to arrive at some vertex t such that $t.key = k$. If we achieve this, then we will delete t (and **return** $t.elt$), and replace t with the closest (in key-value) vertex u (u is a "near-leaf" vertex as before—note that this does not imply *both* u's children are leaves, but one will be). We can find u by "walking down" from t to the near-leaf with largest key value no greater than k. Then t gets u's item and u's height drops by 1.

After this deletion and re-arrangement, any *newly unbalanced* vertex must be on the path from u to the root of the tree. In fact, there can be at most one unbalanced vertex (why?). Let z be this unbalanced vertex. Then the heights of the two children of z must differ by 2. Let y be the child of z of greater height and x the child of y of greater height. If both children of y have the same height, let x be the left child of y if y is the left child of z, and let x be the right child of y if y is the right child of z. Now we apply the appropriate rotation and obtain a tree where z is balanced and where x, y remain balanced. However, we may have

Algorithm insertItem(k, e)

1. Perform lines 1.-7. of findElement()(k, e) on the tree to find the "right" place for an item with key k (if it finds k high in the tree, walk down to the "near-leaf" with largest key no greater than k).
2. Neighbouring leaf vertex u becomes internal vertex, $u.key \leftarrow k$, $u.elt \leftarrow e$.
3. Find the lowest unbalanced vertex z on the path from u to the root.
4. **if** no such vertex exists, **return** (tree is still balanced).
5. **else**
6. Let y and x be child and grandchild of z on $z \rightarrow u$ path.
7. Apply the appropriate rotation to x, y, z.

Algorithm 5.11

reduced the height of the subtree originally located at z by one, and this may cause the parent of z to become unbalanced. If this happens, we have to apply a rotation to the parent and, if necessary, rotate our way up to the root. Algorithm 5.12 on the following page gives pseudocode for removeItem(k) operating on an AVL-tree.

Algorithm removeItem(k)

1. Perform lines 1–7 of findElement(k) on the tree to get to vertex t.
2. **if** we find t with $t.key = k$,
3. **then** remove the item at t, set $e = t.elt$.
4. Let u be "near-leaf" closest to k. Move u's item to t.
5. **while** u is not the root **do**
6. let z be the parent of u
7. **if** z is unbalanced **then**
8. do the appropriate rotation at z
9. Reset u to be the (possibly new) parent of u
10. **return** e
11. **else return** NO_SUCH_KEY

Algorithm 5.12

We now discuss the worst-case running-time $T_{\text{removeItem}}(n)$ of our AVL implementation of removeItem. Again letting h denote the height of the tree, we recall that line 1 requires time $O(h) = O(\lg(n))$. Lines 2. and 3. will take $O(1)$ time, while line 4. will take $O(h)$ time again. The loop in lines 5–9 is iterated at most h times. Each iteration requires time $O(1)$. Thus the execution of the whole loop requires

time $O(h)$. Altogether, the asymptotic worst-case running time of removeItem is $O(h) = O(\lg(n))$.

5.5 Ordered Dictionaries

AVL trees are the first data structures for *Dictionaries* we have seen that implement all methods with a worst case running time of $O(\lg(n))$. There are a few similar tree-based dictionary implementations with the same running time. Still, hash tables are more efficient in practice. The main advantage of AVL trees and similar tree-based *Dictionary* implementations is that they support the order of the keys quite efficiently. The *OrderedDictionary* ADT is an extension of the *Dictionary* ADT by methods such as closestKeyBefore(k), which finds the closest key less than or equal to k that is stored in the dictionary. Such methods can easily be implemented on AVL trees with a running time of $O(\lg(n))$, but there is no efficient way of supporting them on hash tables. The same holds for so-called *range queries* such as findAllElementsBetween(k_1, k_2), i.e., find all elements whose key is between k_1 and k_2. Range queries are very important in database applications.

5.6 Reading

[GT] covers AVL trees, but [CLRS] only has details of a different balanced tree called the *Red-Black Tree*.

Note there are detailed worked examples for insertItem and removeItem in the slides to accompany this lecture note.

Exercises

1. Perform the following sequence of operations on an initially empty AVL tree: insertItem(44), insertItem(17), insertItem(32), insertItem(78), insertItem(50), insertItem(88), insertItem(48), insertItem(62), removeItem(32), removeItem(88).

2. Describe a family of AVL-trees in which a single removeItem operation may require $\Theta(\lg(n))$ rotations.

3. Suppose we wanted to implement an extra method findByRank(r) (as described in LN 3 for the *Vector* class) on the AVL tree data structure, and we want to get $O(\lg(n))$ time for this method also. Sketch a plan of how we can accomplish this (while guaranteeing $O(\lg(n))$ running time for our three existing methods)?

4. Give a pseudo-code implementation of the range query

   $$\text{findAllElementsBetween}(k_1, k_2)$$

   on AVL-trees. What is the running time of your method? Is it possible to implement findAllElementsBetween(k_1, k_2) with a running time of $O(\lg(n))$?

Hint: The number of elements returned by the method has an effect on the running time.