Binary Search Trees

A rooted binary tree $\underline{t}$ whose nodes are from an ordered set A is called a "Binary Search Tree" (BST for short) if the following condition holds:

- For a node p in $\underline{t}$, let λ be the left sub-tree and ρ denote the right sub-tree. Every node in λ is *less* than p and every note in ρ is *greater* than p.

Recall that a binary tree can be written as an array $(t_0, \ldots, t_k)$ where the entry at i has left child at $2 \cdot i + 1$ and right child at $2 \cdot +2$. With this convention we note that:

- $(0, 1, 2, 3, 4, 5, 6)$ is not a binary search tree.
- $(3, 1, 5, 0, 2, 4, 6)$ is a binary search tree.

Exercise: Draw pictures of the trees above to convince yourself of the statements.

A binary tree of depth d has *at most* $2^{d+1} - 1$ nodes. Thus, if n is the number of nodes we have $n \leq 2^{d+1} - 1$. This is the same as the assertion that $\log_2(n) \leq d$.

Let us see how the size of d affects the "search" aspect of the tree.

Searching

Given a tree $\underline{t}$ with a node c, we let $c.p$ denote its parent or `NIL` if c is the root node; we let $c.l$ denote its left child or `NIL`, and $c.r$ denotes its right child or `NIL`.

Here is how we can search for v in the sub-tree of the BST-tree $\underline{t}$ that lies below the (non-`NIL`) node x.

(Note: In the following algorithms we use $<$ and $>$ symbols for comparisons. The actual comparison functions for nodes may be more complicated.)

```
define bstsearch(t, v, x)
    if x < v
        if x.r is NIL
            There is no v in the tree, so stop with appropriate message.
        else
            return bstsearch(t, v, x.r)
    if x > v
        if x.l is NIL
            There is no v in the tree, so stop with appropriate message.
        else
            return bstsearch(t, v, x.l)
    v is found at x, so stop with appropriate message.
```

Note that the number of steps is roughly equal to the depth d of the tree. Thus, for a quick search we need the depth d to be as small as possible while having the same number of nodes n. This means the depth should be $\lceil \log_2(n) \rceil$. However, maintaining the tree in this fashion would require each mode to be (very close to) the median of the sub-tree that lies below it. Thus, it would seem that this

maintenance would take too many steps. Instead, it may be good enough if the depth is maintained as $O(\log(n))$. We could think of such a binary search tree as *balanced*. We will look at such examples later on.

There are other kinds of searching operations on a BST which also depend only on the depth.

- The minimum of the nodes is the left most leaf node.
- The maximum of the nodes is the left most leaf node.

More generally, we can do this search starting at any non-`NIL` node x.

```
define bstmin(t, x)
    if x.l is not NIL
        return bstmin(t, x.l)
    else
        return x.
```

Exercise: Write `bstmax` which finds the maximum element below a given node x.

The "successor" of a node x in the tree $\underline{t}$ can be found using `bstmax` as below. It is the smallest element of $\underline{t}$ which is greater than x, or if there is no node larger than x, then x itself.

```
define bstsucc(t, x)
    if x.r is not NIL
        return bstmax(t, x.r)
    else
        return x
```

Similarly, the "predecessor" of a node x in the tree $\underline{t}$ can be found using `bstmax` as below. It is the biggest element of $\underline{t}$ which is smaller than x, or x is there is no node smaller than x.

Exercise: Write `bstpred` which finds the predecessor of a given node x.

Inserting

To insert a node v in a tree $\underline{t}$ we use code similar to `bstsearch` above.

The following code inserts v in the sub-tree that lies below x.

```
define bstinsert(t, v, x)
    if x < v
        if x.r is NIL
            Set x.r to be v.
        else
            return bstsearch(t, v, x.r)
    if x > v
        if x.l is NIL
```

Set $x.l$ to be v (almost).
else
return `bstsearch`$(\underline{t}, v, x.l)$
v is already at x, so stop with appropriate message.

Deletion

Deletion of a node x is a little more complicated.

- If x is a leaf node (i.e. it has no left or right child), then we just replace the node with `NIL`.
- If x has only one non-`NIL` child (i.e. either left or right child is `NIL`), then this non-`NIL` child takes its place.
- If x has a left child *and* a right child, then x can be replaced by either its successor or predecessor. We explain how to replace x by its successor y below. Note that $y.l$ is `NIL` in this case.
  - If $y = x.r$, then y takes the place of x.
  - If $y \neq x.r$, then y lies in the sub-tree below $x.r.l$. We replace x by y and y by $z = y.r$ in this case.

Exercise: Convert the above description of deletion into a program or pseudo-code.

Symmetry and Balancing

Suppose x and $y = x.l$ are non-`NIL` nodes in a binary search tree. Let α be the sub-tree on the left of y, β be the sub-tree on the right of y and γ be the sub-tree on the right of x. We claim that we can make another binary-search tree in which y takes the place of x and we put x as the right child of y; moreover *alpha* as the sub-tree on the left of y (as before), β is the sub-tree on the left of x and γ is the sub-tree on the right of x (as before).

This operation is called a *right rotation* of the tree around x and its left child y.

In the reverse direction, if y and $x = y.r$ are non-`NIL` nodes in a binary search tree, let α be the sub-tree on the left of y, β be the sub-tree on the left of x and γ be the sub-tree on the right of x. We can make another binary-search tree in which x takes the place of y and we put y as the left child of x; moreover α remains the sub-tree to the left y, β becomes the sub-tree to the right of y and γ remains the sub-tree to the right of x.

This operation is called a *left rotation* of the tree around y and its right child x.

Now, suppose x and $y = x.l$ are non-`NIL` nodes and the notation α, β, γ is as above. The depth of the sub-tree below y is the maximum of the depths of α and β. Thus, the depth of the sub-tree on the left of the top node (x) is 1 more than this maximum. The depth of the tree on the right of x is the depth of γ.

After a right rotation, y becomes the top node and its left tree is α. So the depth of the tree on the left of the top node has *reduced* by at least one. On the other hand, the depth of the tree on the right of y is now 1 more than the maximum of the depths of β and γ, Thus, the depth of the tree on the right of the top node has increased by at least 1.

This basic idea of rotations allows us to "shift" the depth from the left to the right and vice versa. Thus, we can balance trees by such rotations.

AVL Trees

AVL trees are named after their discoverers Georgy Adelson-Velsky and Evgenii Landis. The idea is to create and maintain binary search trees where the depth of the left sub-tree differs from the depth of the right sub-tree by at most 1. Moreover, we keep data at each node which indicates whether it is the left or right sub-tree is deeper by storing -1 for the left being deeper, 0 for both being equal and $+1$ for the right being deeper.

Note that an insertion or deletion can only affect the nodes that lie above the node which is inserted or deleted, and the change of depth is at most one. Thus, one can show that one needs at most one rotation between a parent and a child, or one pair of opposite rotations at a grand-parent, parent and child to ensure that the AVL property is restored. These nodes lie in the path joining the inserted/deleted node to the root node. Thus, the number of steps required to find these nodes is at most d, the depth of the AVL tree.

What can we say about the minimum number $f(d)$ of nodes of an AVL tree of depth d?

Note that the minimum number of nodes of an AVL tree of depth 0 is 1; the minimum number of nodes of an AVL tree of depth 1 is 2 since one child of the root could be missing.

The left and right sub-trees of an AVL tree of depth d are trees of depth at most $d-1$. In fact, one of them *must* have depth exactly $d-1$. Moreover, by the AVL condition, the other must have depth at least $d-2$. It follows that we have $f(d) \geq f(d-1) + f(d-2)$. Since $f(1) = 2$ is the third Fibonacci number $F_3 = 2$ and $f(0) = 1$ is the second Fibonacci number $F_2 = 1$, we show by induction that $f(d) \geq F_{d+2}$.

It follows that we have that the depth of AVL trees with n nodes is $O(\log(n))$.

By the above informal description it follows that restoring the AVL property after insertion or deletion takes at most $O(\log(n))$ steps.

Another structure on binary search trees that can be balanced in $O(\log(n))$ steps and have depth $O(\log(n))$ is Red-Black trees. In such trees:

- Each node is coloured Red or Black.
- All `NIL` nodes are Black.

- A child of a Red node is Black.
- Every path from a node x to its descendent `NIL` nodes has the same number $b(x)$ of block nodes in it.

As in the case of AVL trees, the above conditions can be restored after insertion/deletion by rotations in the path joining the changed node to the root node.