Binary Search Trees

Managing Information with Binary Comparisons

What is a Binary Search Tree?

Figure 12.1 Binary search trees. For any node \( x \), the keys in the left subtree of \( x \) are at most \( x.key \), and the keys in the right subtree of \( x \) are at least \( x.key \). Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. (a) A binary search tree on 6 nodes with height 2. (b) A less efficient binary search tree with height 4 that contains the same keys.
Binary Trees

Navigating a Binary Search Tree
We have three procedures for exploring all the nodes of a binary tree. We show one, with the other two (PREORDER-TREE-WALK and POSTORDER-TREE-WALK) being obvious variants.

\[
\text{INORDER-TREE-WALK}(x)
\]
1. \textbf{if } \text{NIL} \neq \text{NIL}\
2. \text{INORDER-TREE-WALK}(x.	ext{left})
3. \text{print } x.	ext{key}
4. \text{INORDER-TREE-WALK}(x.	ext{right})

Binary Trees

Navigating a Binary Search Tree: Complexity

**Theorem 12.1.** If \(x\) is the root of an \(n\)-node subtree, the the call INORDER-TREE-WALK \((x)\) takes \(\Theta(n)\) time.

**Proof.** Let \(T(n)\) denote the time spent by INORDER-TREE-WALK starting from the root of an \(n\)-node tree. Since the procedure accesses all the nodes, we must have \(T(n) = \Omega(n)\). To show: \(T(n) = O(n)\).

Let \(d\) denote the amount of time that INORDER-TREE-WALK spends on activities outside of recursion (in this case, it just prints the key). We assume this amount is constant. Assume further that \(T(0) = c\).

For \(n > 0\), suppose the root has a subtree of size \(k\) and another subtree of size \(n - k - 1\) (we don’t recurse on the root).
Binary Trees

Navigating a Binary Search Tree

Then $T(n) \leq \max_{0 \leq k \leq n-1}(T(k) + T(n-k-1) + d)$.

We use the substitution method, with the guess

$T(n) \leq (c + d)n + c$.

Base case: $T(0) \leq c$. OK.

Induction case:

$T(n) \leq \max_{0 \leq k \leq n-1}((c + d)k + c) + ((c + d)(n-k-1) + c) + d$ 

$\leq \max_{0 \leq k \leq n-1}(c + ((c + d)(n-1) + c) + d) \leq (c + d)n + c$.

Note that the maximization was independent of $k$.

---

Searching a Binary Tree

**Figure 12.2** Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$ from the root. The minimum key in the tree is 2, which is found by following left pointers from the root. The maximum key 20 is found by following right pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.
Binary Trees

Navigating a Binary Tree: Search

\[ \text{TREE-SEARCH}(x, k) \]

1. \textbf{if} \( x = \text{NIL} \) \textbf{or} \( k = x.\text{key} \)
2. \textbf{return} \( x \)
3. \textbf{if} \( k < x.\text{key} \)
4. \textbf{return} \( \text{TREE-SEARCH}(x.\text{left}, k) \)
5. \textbf{else return} \( \text{TREE-SEARCH}(x.\text{right}, k) \)

\[ \text{ITERATIVE-TREE-SEARCH}(x, k) \]

1. \textbf{while} \( x \neq \text{NIL} \) \textbf{and} \( k \neq x.\text{key} \)
2. \textbf{if} \( k < x.\text{key} \)
3. \( x = x.\text{left} \)
4. \textbf{else} \( x = x.\text{right} \)
5. \textbf{return} \( x \)

---

Binary Trees

Navigating a Binary Tree: Search

Min, Max, Successor, Predecessor.

\[ \text{TREE-MINIMUM}(x) \]

1. \textbf{while} \( x.\text{left} \neq \text{NIL} \)
2. \( x = x.\text{left} \)
3. \textbf{return} \( x \)

\[ \text{TREE-MAXIMUM}(x) \]

1. \textbf{while} \( x.\text{right} \neq \text{NIL} \)
2. \( x = x.\text{right} \)
3. \textbf{return} \( x \)

\[ \text{TREE-SUCCESSOR}(x) \]

1. \textbf{if} \( x.\text{right} \neq \text{NIL} \)
2. \textbf{return} \( \text{TREE-MINIMUM}(x.\text{right}) \)
3. \( y = x.p \)
4. \textbf{while} \( y \neq \text{NIL} \) \textbf{and} \( x = y.\text{right} \)
5. \( x = y \)
6. \( y = y.p \)
7. \textbf{return} \( y \)
Binary Trees

Navigating a Binary Tree: Search Complexity

**Theorem 12.2.** We can implement the dynamic set operations **SEARCH**, **MINIMUM**, **MAXIMUM**, **SUCCESSOR**, and **PREDECESSOR** so that each of them runs in $O(h)$ time on a binary tree of height $h$.

Proof. Immediate from an analysis of the loops.

Managing a Binary Tree: Insertion

We maintain a trailing pointer $y$ as parent of $x$.

When $x = \text{NIL}$, then $y$ points to the element we need to attach $z$ to.

```
TREE-INSERT(T, z)
1    y = NIL
2    x = T.root
3    while x \neq NIL
4        y = x
5            if z.key < x.key
6                x = x.left
7            else x = x.right
8    z.p = y
9    if y == NIL
10       T.root = z // tree T was empty
11    elseif z.key < y.key
12       y.left = z
13    else y.right = z
```
Binary Trees
Managing a Binary Tree: Insertion

The time complexity of Tree-Insert is $O(h)$, where $h$ is the height of the tree.

Binary Trees
Managing a Binary Tree: Deletion

This is a much harder operation to implement: insertion always adds a leaf – so we just need to find the right leaf position to implement. Deletion may try to delete a leaf – which would be easy to manage, but may also try to delete an interior node. If the node to be deleted has a single child, life is easy: just attach the single child in place of the node just deleted. If the node to be deleted has two children, life is more complicated...
Binary Trees

Managing a Binary Tree: Deletion

The main ideas (again) are:
1. If \( z \) has no children (a leaf), simply remove it and set the pointer of the parent pointing to \( z \) to NIL.
2. If \( z \) has just one child, then we elevate that child to take \( z \)'s position in the tree by modifying \( z \)'s parent to replace \( z \) by \( z \)'s child.
3. If \( z \) has two children, we find \( z \)'s successor \( y \) (in \( z \)'s right subtree) and have \( y \) take \( z \)'s position in the tree. The rest of \( z \)'s original right subtree becomes \( y \)'s new right subtree, and \( z \)'s left subtree becomes \( y \)'s new left subtree. This is a bit tricky...

Binary Trees

Managing a Binary Tree: Deletion

We have the two simple cases:
Binary Trees

Managing a Binary Tree: Deletion

And the harder ones:

We define a subroutine to manage moving subtrees around: it replaces the subtree rooted at $u$ with the subtree rooted at $v$. Node $u$’s parent becomes $v$’s parent, and $u$’s parent receives $v$ as its appropriate child.

```
TRANSPLANT(T, u, v)
1 if u.p == NIL
2    T.root = v
3 elseif u == u.p.left
4    u.p.left = v
5 else u.p.right = v
6 if v ≠ NIL
7    v.p = u.p
```
Binary Trees

Managing a Binary Tree: Deletion

TREE-DELETE($T$, $z$)

1. if $z$.left == NIL
2. TRANSLANT($T$, $z$, $z$.right)
3. elseif $z$.right == NIL
4. TRANSLANT($T$, $z$, $z$.left)
5. else $y$ = TREE-MINIMUM($z$.right)
6. if $y$.p $\neq$ $z$
7. TRANSLANT($T$, $y$, $y$.right)
8. $y$.right = $z$.right
9. $y$.right.p = $y$
10. TRANSLANT($T$, $z$, $y$)
11. $y$.left = $z$.left
12. $y$.left.p = $y$

Note that the only operation whose time complexity depends on the height of the tree is TREE-MINIMUM – all others involve a fixed number of elementary operations.
Binary Trees

Building a Binary Tree

The major problem is that none of the operations (especially insertion) we have introduced give us a good bound on $h$, the height of the tree. One can prove that a randomly built binary tree has expected height $O(\log n)$, where $n$ is the number of elements inserted. One can also show that the maximum possible height is $n$.

**Question:** can we modify Tree-Insert and Tree-Delete in such a way that the height of the resulting tree is always $O(\log n)$?

The answer will be, as often in such cases, YES, BUT...