Algorithms and Data Structures

Chapter 12

Catherine Durso

cdurso@cs.du.edu

Chapter 12

Balanced binary search trees support SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $\Theta(\lg n)$ time. In addition, and in contrast to hash tables, values may be added indefinitely while retaining this performance.The structures are elaborations of the binary search tree, the topic here.

Binary Search Tree Property

A binary search tree is a binary tree. Each node x has data members $x.key$, $x.left$, $x.right$, and $x.p$, the key for the data at the node, and the pointers to the left child, right child, and parent, respectively. The keys have an order operation.

The ***binary-search-tree property*** asserts that if x is a node in a binary search tree, and y is a node in the left subtree of x then $y.key \leq x.key$. If y is a node in the right subtree of x then $y.key \geq x.key$.

In-order Tree Walk

INORDER-TREE-WALK(x)

1. if $x \neq NIL$
2. INORDER-TREE-WALK $(x.left)$
3. print $(x.key)$
4. INORDER-TREE-WALK $(x.right)$

Try an example. What can you say about the order in which the keys are printed? Can you prove this claim inductively on the number on items in the tree?

Search

TREE-SEARCH(x, k)

1. if $x = NIL$ or $k = x.key$
2. return x
3. if $k < x.key$
4. return TREE-SEARCH $(x.left, k)$
5. else
6. return TREE-SEARCH $(x.right, k)$

Iterative SEARCH

ITERATIVE-TREE-SEARCH(x, k)

1. while $x \neq NIL$ and $k \neq x.key$
2. if $k < x.key$
3. $x = x.left$
4. else
5. $x = x.right$
6. return x

This will typically be faster than the recursive implementation. Can you give a $\Theta-$bound on the running time?

Minimum

TREE-MINIMUM(x)

1. while $x.left \neq NIL$
2. $\quad x = x.left$
3. return x

Any guesses on how to find the maximum key in the subtree rooted at x?

PREDECESSOR

TREE-PREDECESSOR(x)
1. if $x.left \neq NIL$
2. return TREE-MAXIMUM$(x.left)$
1. $y = x.p$
1. while $y \neq NIL$ and $x = y.left$
5. $x = y$
5. $y = x.p$
6. return y

If x has a non-empty left subtree, the predecessor of x is the maximum value in x's left subtree. If x has an empty left subtree, then x is the minimum value in the right subtree of the lowest ancestor of x

Why?

If x has a non-empty left subtree, the predecessor of x is the maximum value in x's left subtree.

If x has an empty left subtree, then x is the minimum value in the right subtree of the lowest ancestor of x to have x in its right subtree. This is the y found by following parent pointers upward to the first parent to have the previous node as a right child.

The correctness of this is seen by considering the in-order tree walk near x.

How do you suppose you would find the successor?

Insertion

The basic idea for insertion of a node z is to search for $z.key$. The search will fail when the search algorithm moves to a NIL node. This is where to put z.

The text gives pseudocode for an iterative algorithm for insertion, assuming that the node to be inserted has NIL children. Under the same assumption, we can write a two-part recursive insertion.

Recursive SUBTREE-INSERT

SUBTREE-INSERT(x, z) /called on non-NIL x

1. if $z.key < x.key$
2. if $x.left = NIL$
3. $x.left = z$
4. else SUBTREE-INSERT$(x.left, z)$
5. else
6. if $x.right = NIL$
7. $x.right = z$
8. else SUBTREE-INSERT$(x.right, z)$

TREE-INSERT

TREE-INSERT(T, z)

1. if $T.root = NIL$
2. $\quad T.root = z$
3. else SUBTREE-INSERT$(T.root, z)$

Deletion

Deletion handles three cases separately.

- If the node z to be deleted has no children, simply modify $z.p$ to replace the child z with NIL.
- If z has a single non-NIL child, splice out z by linking $z.p$ directly to z's child.
- If z has two non-NIL children, then z's successor, y, is in z's right subtree and y has no left child. Copy y's data over z's then splice out y. Under these circumstances, the binary search property is preserved by replacing the data in z with the data in the successor. (The pseudocode handles the cases $y = z.p$ and $y \neq p$ separately, but the effect is as above.)

Performance

Each of these operations has the worst-case time bound $\Theta(n)$ where n is the number of items in T. (For SUCCESSOR, the worst case occurs if the successor is at the end of a degenerate branch with length proportional to n. Can you construct such a tree?)

These worst-case behaviors are generated by trees that are unbalanced, in the sense of most items in subtrees that are nearly linked lists.

Importance of Height

A tree node is called a ***leaf*** if it has only NIL children. Define the ***height*** of a tree node to be the number of edges on the longest simple downward path from the node to a leaf. More instructively, we can bound the time required for each of the tree operations by $\Theta(h)$, where h is the height of the tree. This indicates that if we could control the height to grow as $\lg n$, these operations would be $\Theta(\lg n)$.