Design and Analysis of Algorithms

Lecture 6—Binary Search Trees

Lei Wang

Dalian University of Technology

October 8, 2008

Binary Search Trees

Goals

- Binary search trees, tree walks, and operations on binary search trees.

Search Trees

- Data structures that support many dynamic-set operations.
- Can be used as both a dictionary and as a priority queue.
- Basic operations take time proportional to the ***height*** of the tree.
- For complete binary tree with n nodes: worst case $\Theta(\lg n)$.
- For linear chain of n nodes: worst case $\Theta(n)$.
- Different types of search trees include binary search trees, red-black trees (covered in Chapter 13), and B-trees (covered in Chapter 18).

Binary search trees

Binary search trees are an important data structure for dynamic sets.

- Accomplish many dynamic-set operations in $O(h)$ time, where h = height of tree.
- Each node contains the fields
  - *key* (and possibly other satellite data).
  - *left*: points to left child.
  - *right*: points to right child.
  - p: points to parent. $p[root[T]] = \text{NIL}$.
- Stored keys must satisfy the ***binary-search-tree*** property.
  - If y is in left subtree of x, then $key[y] \leq key[x]$.
  - If y is in right subtree of x, then $key[y] \geq key[x]$.

Example

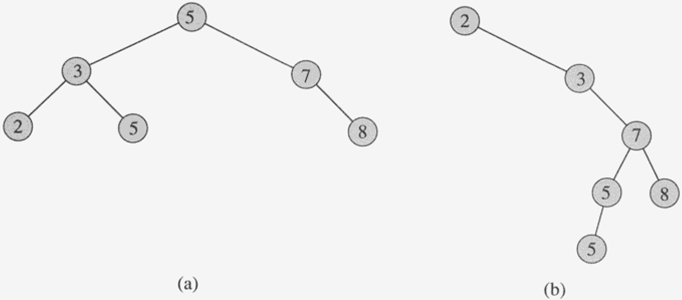

Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $key[x]$, and the keys in the right subtree of x are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional 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.

Inorder tree walk

Elements are printed in monotonically increasing order. How INORDER-TREE-WALK works:

- Check to make sure that x is not NIL.
- Recursively, print the keys of the nodes in x's left subtree.
- Print x's key.
- Recursively, print the keys of the nodes in x's right subtree.

INORDER-TREE-WALK

```
INORDER-TREE-WALK(x)
if x ≠ NIL
   then INORDER-TREE-WALK(left[x])
        print key[x]
        INORDER-TREE-WALK(right[x])
```

Time: Intuitively, the walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once.

Searching

TREE-SEARCH

```
TREE-SEARCH(x, k)
if x = NIL or k = key[x]
    then return x
if k < key[x]
    then return TREE-SEARCH(left[x], k)
    else  return TREE-SEARCH(right[x], k)
```

Time: The algorithm recurses, visiting nodes on a downward path from the root. Thus, running time is $O(h)$, where h is the height of the tree.

Minimum and maximum

The binary-search-tree property guarantees that:

- the minimum key is located at the ***leftmost*** node, and
- the maximum key is located at the bf *rightmost* node.

TREE-MINIMUM

```
TREE-MINIMUM(x)
while left[x] ≠ NIL
    do x ← left[x]
return x
```

TREE-MAXIMUM

```
TREE-MAXIMUM(x)
while right[x] ≠ NIL
    do x ← right[x]
return x
```

Time: Both procedures visit nodes that form a downward path from the root to a leaf. Both procedures run in $O(h)$ time, where h is the height of the tree.

Successor and predecessor

The successor of a node x is the node y such that $key[y]$ is the smallest key $> key[x]$. (We can find x's successor based entirely on the tree structure. No key comparisons are necessary.) If x has the largest key, then x's successor is NIL.

```
TREE-SUCCESSOR(x)
if right[x] ≠ NIL
    then return TREE-MINIMUM(right[x])
y ← p[x]
while y ≠ NIL and x = right[y]
      do x ← y
         y ← p[y]
return y
```

Time: Since we visit nodes on a path down the tree or up the tree, the running time is $O(h)$, where h is the height of the tree.

Insertion

```
TREE-INSERT(T, z)
y ← NIL
x ← root[T]
while x ≠ NIL
      do y ← x
         if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = NIL
   then root[T] ← z                        ▷ Tree T was empty
   else if key[z] < key[y]
           then left[y] ← z
           else right[y] ← z
```

Time: Same as TREE-SEARCH. On a tree of height h, procedure takes $O(h)$ time.

Deletion

TREE-DELETE is broken into three cases.

Case 1: z has no children.

- Delete z by making the parent of z point to NIL, instead of to z.

Case 2: z has one child.

- Delete z by making the parent of z point to z's child, instead of to z.

Case 3: z has two children.

- z's successor y has either no children or one child. (y is the minimum node—with no left child—in z's right subtree.)
- Delete y from the tree (via Case 1 or 2).
- Replace z's key and satellite data with y's.

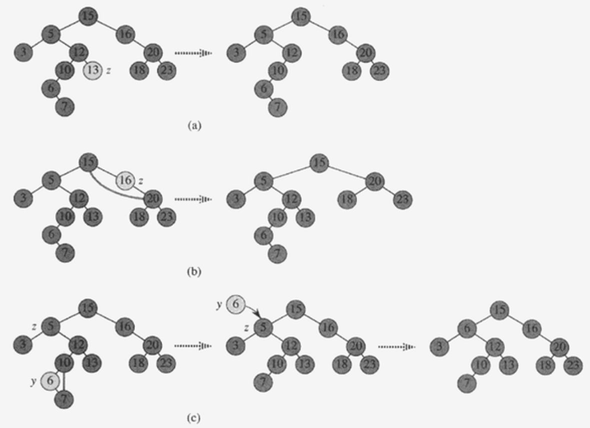

Figure 12.4 Deleting a node z from a binary search tree. Which node is actually removed depends on how many children z has; this node is shown lightly shaded. **(a)** If z has no children, we just remove it. **(b)** If z has only one child, we splice out z. **(c)** If z has two children, we splice out its successor y, which has at most one child, and then replace z's key and satellite data with y's key and satellite data.

Time: $O(h)$, on a tree of height h.

Minimizing running time

We've been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

- ***Problem:*** Worst case for binary search tree is $\Theta(n)$—no better than linked list.
- ***Solution:*** Guarantee small height (balanced tree)—$h = O(\lg n)$. In later chapters, by varying the properties of binary search trees, we will be able to analyze running time in terms of n.
- ***Method:*** Restructure the tree if necessary. Nothing special is required for querying, but there may be extra work when changing the structure of the tree (inserting or deleting).