CSE 332: Data Structures
Binary Search Trees

Richard Anderson, Steve Seitz
Winter 2014
Announcements

• HW #2 due next Wednesday
• Project 2 out today
  – can work with partners (optional). Must sign up
  – **harder** than project 1 (16 files to implement)
  – start early!
• Read Chapter 4.1-4.3, 4.6
• No class on Monday
ADTs Seen So Far

- **Stack**
  - Push
  - Pop

- **Queue**
  - Enqueue
  - Dequeue

- **Priority Queue**
  - Insert
  - DeleteMin

None of these support “find”
The Dictionary ADT

• Data:
  – a set of (key, value) pairs

• Operations:
  – Insert (key, value)
  – Find (key)
  – Remove (key)

The Dictionary ADT is also called the "Map ADT"
Many Uses

• Networks: router tables
• Operating systems: page tables
• Compilers: symbol tables
• Search: phone directories, ...
• Biology: genome maps
• Vision: object recognition
• ...

Probably the most widely used ADT!
Implementations

- Unsorted Linked-list
  - \( O(1) \) for insert
  - \( O(n) \) for find
  - \( O(n) \) for delete

- Unsorted array
  - \( O(1) \) for insert
  - \( O(n) \) for find and delete

- Sorted array
  - \( O(n) \) for insert
  - \( O(\log n) \) for find
  - \( O(n) \) to shift

\( = O(n) \)
Binary Trees

• Binary tree is
  – a root
  – left subtree (*maybe empty*)
  – right subtree (*maybe empty*)

• Representation:

```
data
+------------------+
| left | right |
+------------------+
```

```
A
/|
B C
/|
D E F
/|
G  H I J
```


Binary Tree: Representation

```
A
  left pointer  right pointer

B
  left pointer  right pointer
  D
  left pointer  right pointer
  E
  left pointer  right pointer

C
  left pointer  right pointer
  F
  left pointer  right pointer

A
  B
    D
    E
    F

C
```

8
Tree Traversals

A *traversal* is an order for visiting all the nodes of a tree

Three types:
- **Pre-order**: Root, left subtree, right subtree
  
  \[
  + \ast 2 4 5
  \]
- **In-order**: Left subtree, root, right subtree
  
  \[
  2 \ast 4 + 5
  \]
- **Post-order**: Left subtree, right subtree, root
  
  \[
  24 \ast 5 +
  \]
Inorder Traversal

```c
void traverse(BNode t){
    if (t != NULL)
        traverse (t.left);
    process t.element;
    traverse (t.right);
}
```
Binary Tree: Special Cases

- **Complete Tree**: Every level of the tree is fully filled.
- **Perfect Tree**: Every level is fully filled and all leaves are at the same depth.
- **Full Tree**: Every node has 0 or 2 children.
- **“List” Tree**: Only one node exists at each level of the tree.
Binary Tree: Some Numbers...

Recall: height of a tree = longest path from root to leaf.

For binary tree of height $h$:
- max # of leaves: $2^h$
- max # of nodes: $2^{h+1} - 1$
- min # of leaves: 1
- min # of nodes: $h+1$
Binary Search Tree Data Structure

- **Structural property**
  - each node has $\leq 2$ children

- **Order property**
  - all keys in left subtree smaller than root’s key
  - all keys in right subtree larger than root’s key
Example and Counter-Example

BINARY SEARCH TREES?
Find in BST, Recursive

Node Find(Object key, Node root) {
    if (root == NULL) {
        return NULL;
    }
    if (key < root.key) {
        return Find(key, root.left);
    } else if (key > root.key) {
        return Find(key, root.right);
    } else { 
        return root;
    }
}
Find in BST, Iterative

Node Find(Object key,
           Node root) {

    while (root != NULL &&
           root.key != key) {
        if (key < root.key)
            root = root.left;
        else
            root = root.right;
    }

    return root;
}
Bonus: FindMin/FindMax

- Find minimum
- Find maximum
Insert in BST

Insertions happen only at the leaves – easy!

Runtime:

$O(n)$
BuildTree for BST

• Suppose keys 1, 2, 3, 4, 5, 6, 7, 8, 9 are inserted into an initially empty BST.

    If inserted in given order, what is the tree? What big-O runtime for this kind of sorted input?

    If inserted in reverse order, what is the tree? What big-O runtime for this kind of sorted input?
BuildTree for BST

- Suppose keys 1, 2, 3, 4, 5, 6, 7, 8, 9 are inserted into an initially empty BST.
  - If inserted median first, then left median, right median, etc., what is the tree? What is the big-O runtime for this kind of sorted input?
Deletion in BST

Why might deletion be harder than insertion?
Deletion

• Removing an item disrupts the tree structure.
• Basic idea: find the node that is to be removed. Then “fix” the tree so that it is still a binary search tree.
• Three cases:
  – node has no children (leaf node)
  – node has one child
  – node has two children
Deletion – The Leaf Case

Delete(17)
Deletion – The One Child Case

Delete(15)
Deletion – The Two Child Case

Delete(5)

What can we replace 5 with?
Deletion – The Two Child Case

Idea: Replace the deleted node with a value *between* the two child subtrees

Options:

- *succ* from right subtree: findMin(t.right)
- *pred* from left subtree: findMax(t.left)

Now delete the original node containing *succ* or *pred*

- Leaf or one child case – easy!
Finally…

7 replaces 5

Original node containing 7 gets deleted
Balanced BST

Observations

• BST: the shallower the better!
• For a BST with \( n \) nodes
  – Average depth (averaged over all possible insertion orderings) is \( O(\log n) \)
  – Worst case maximum depth is \( O(n) \)
• Simple cases such as insert(1, 2, 3, ..., \( n \)) lead to the worst case scenario

Solution: Require a **Balance Condition** that

1. ensures depth is \( O(\log n) \) – strong enough!
2. is easy to maintain – not too strong!