Binary Search Trees

Today’s Outline

• Announcements
  – Assignment #1 due (tonite) Fri, April 10 at 11:45pm
  – Assignment #2 due Fri, April 17, coming soon!
  – Midterm Dates:
    • Midterm #1: Friday, April 24th
    • Midterm #2: Wednesday, May 20th

• Today’s Topics:
  – Asymptotic Analysis
  – Binary Search Trees

Today’s Outline

• Announcements
  – Assignment #1 due (tonite) Fri, April 10 at 11:45pm
  – Assignment #2 due Fri, April 17, coming soon!
  – Midterm Dates:
    • Midterm #1: Friday, April 24th
    • Midterm #2: Wednesday, May 20th

• Today’s Topics:
  – Asymptotic Analysis
  – Binary Search Trees

Tree Calculations Example

How high is this tree?

More Recursive Tree Calculations:

Tree Traversals

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

Three types:
• Pre-order: Root, left subtree, right subtree
• In-order: Left subtree, root, right subtree
• Post-order: Left subtree, right subtree, root

Traversals

void traverse(BNode t){
  if (t != NULL)
    traverse (t.left);
    print t.element;
    traverse (t.right);
  }
}

Binary Trees

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

• Representation:
Binary Tree: Representation

A
  /  \
B    C
  / \
D    E
  / \
F    F

Binary Tree: Special Cases

A
  /  \
B    C
  / \
D    E
  / \
F

A
  /  \
B    C
  / \
D    E
  / \
F

A
  /  \
B    C
  / \
D    E
  / \
F

ADTs Seen So Far

- Stack
  - Push
  - Pop
- Queue
  - Enqueue
  - Dequeue

The Dictionary ADT

- Data:
  - a set of (key, value) pairs
- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)

The Dictionary ADT is sometimes called the "Map ADT"

Implementations

- Unsorted Linked-list
- Unsorted array
- Sorted array

A Modest Few Uses

- Sets
- Dictionaries
- Networks: Router tables
- Operating systems: Page tables
- Compilers: Symbol tables

Probably the most widely used ADT!
Binary Search Tree Data Structure

- **Structural property**
  - each node has ≤ 2 children
  - result:
    - storage is small
    - operations are simple
    - average depth is small

- **Order property**
  - all keys in left subtree smaller than root’s key
  - all keys in right subtree larger than root’s key
  - result: easy to find any given key

- What must I know about what I store?

---

Example and Counter-Example

BINARY SEARCH TREE

NOT A BINARY SEARCH TREE

---

Find in BST, Recursive

```java
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

```java
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;
}
```

---

Insert in BST

Insertions happen only at the leaves – easy!

---

BuildTree for BST

- Suppose keys 1, 2, 3, 4, 5, 6, 7, 8, 9 are inserted into an initially empty BST.
  - Runtime depends on the order:
    - in given order
    - in reverse order
    - median first, then left median, right median, etc.
**Bonus: FindMin/FindMax**

- Find minimum
- Find maximum

**Deletion in BST**

Why might deletion be harder than insertion?

**Lazy Deletion**

Instead of physically deleting nodes, just mark them as deleted

+ simpler
+ physical deletions done in batches
+ some adds just flip deleted flag
- extra memory for deleted flag
- many lazy deletions slow finds
- some operations may have to be modified (e.g., min and max)

**Non-lazy 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

**Non-lazy 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 guaranteed to be 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

Observation
• BST: the shallower the better!
• For a BST with n nodes
  – Average height is $\Theta(\log n)$
  – Worst case height is $\Theta(n)$
• 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 $\Theta(\log n)$ — strong enough!
2. is easy to maintain — not too strong!

Potential Balance Conditions

1. Left and right subtrees of the root have equal number of nodes
2. Left and right subtrees of the root have equal height
3. Left and right subtrees of every node have equal number of nodes
4. Left and right subtrees of every node have equal height