Announcements (1/23/09)

- HW #2 due now
- HW #3 out today, due at beginning of class next Friday.
- Project 2A due next Wed. night.
- Read Chapter 4

ADTs Seen So Far

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

- Priority Queue
  - Insert
  - DeleteMin

Then there is decreaseKey…

The Dictionary ADT

- Data:
  - a set of (key, value) pairs

  \[ \text{insert}(seitz, \ldots) \]

- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)

**The Dictionary ADT is also called the “Map ADT”**
**A Modest Few Uses**

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

*Probably the most widely used ADT!*  

**Implementations**

- Unsorted Linked-list
- Unsorted array
- Sorted array

**Binary Trees**

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

- Representation:

```
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
```

**Binary Tree: Representation**

```
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
```

```
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+
```
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

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**: All levels are fully filled.
- **Perfect Tree**: All levels are fully filled except possibly the last level, which is filled from left to right.
- **“List” Tree**: A list of nodes with no branching.
- **Full Tree**: All levels are fully filled.

Binary Tree: Some Numbers…

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

For binary tree of height $h$:

- max # of leaves:
- max # of nodes:
- min # of leaves:
- min # of nodes:
Recall: depth of a node = path length from node to root.

Consider the space of all possible binary trees of N nodes.
• Sum up the depths of every node in that forest and divide by the number of nodes.
• This is the average depth over all nodes over all binary trees of size N. How big is it?

What would the average depth be for a well-balanced tree?

Binary Search Tree Data Structure

• Structural property
  – each node has ≤ 2 children
  – result:
    • storage is small
    • operations are simple

• 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

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;
  }
}

BINARY SEARCH TREES?
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;
}
```

Runtime:

Bonus: FindMin/FindMax

- Find minimum
- Find maximum

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?

Insert in BST

Insert(13)
Insert(8)
Insert(31)

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.
  – 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 guaranteed to be between the two child subtrees

Options:
• \textit{succ} from right subtree: \texttt{findMin(t.right)}
• \textit{pred} from left subtree: \texttt{findMax(t.left)}

Now delete the original node containing \textit{succ} or \textit{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!