Today’s Outline

- Quick Tree Review
- Binary Trees
- Dictionary ADT / Search ADT
- Binary Search Trees

Reading: Weiss ch. 4

Tree Calculations

Recall: height is max number of edges from root to a leaf

Find the height of the tree...

runtime:

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

Inorder Traversal

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

Binary Trees

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

- Representation:

```
Data

left pointer  right pointer

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

Binary Tree: Representation
### Binary Tree: Special Cases

#### Complete Tree
- A
- B
- C
- D
- E
- F

#### Perfect Tree
- A
- B
- C
- D
- E
- F
- G

#### Full Tree
- A
- B
- C
- D
- E
- F
- G
- H
- I

### Binary Tree: Some Numbers!

For binary tree of height $h$:
- max # of leaves:
- max # of nodes:
- min # of leaves:
- min # of nodes:

### 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

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

**insert(perkins, …)**

**find(marius)**

- **perkins**
  - Hal Perkins
  - OH: MW 3:40
  - CSE 360

- **marius**
  - Marius Nita,
  - OH: Th 2:30
  - 3rd floor breakout

- **andy**
  - Andy Sun,
  - OH: Tue 1:30
  - CSE 002

*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 Search Tree Data Structure

- Structural property
  - each node has \( \leq 2 \) 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 TREES?*
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

```java
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.
  - 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:
- \textit{succ} from right subtree: \text{findMin}(t.\text{right})
- \textit{pred} from left subtree: \text{findMax}(t.\text{left})

Now delete the original node containing \textit{succ} or \textit{pred}
- Leaf or one child case – easy!
Finally…

Balanced BST

Observation
- BST: the shallower the better!
- For a BST with $n$ nodes
  - Average height is $O(\log n)$
  - Worst case height 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!

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

Potential Balance Conditions
3. Left and right subtrees of every node have equal number of nodes
4. Left and right subtrees of every node have equal height
The AVL Balance Condition

Left and right subtrees of every node have equal heights differing by at most 1

Define: balance(x) = height(x.left) – height(x.right)

AVL property: \(-1 \leq \text{balance}(x) \leq 1, \text{ for every node } x\)

- Ensures small depth
  - Will prove this by showing that an AVL tree of height \(h\) must have a lot of (i.e. \(O(2^h)\)) nodes
- Easy to maintain
  - Using single and double rotations

The AVL Tree Data Structure

Structural properties
1. Binary tree property
2. Balance property: balance of every node is between -1 and 1

Result:
- Worst case depth is \(O(\log n)\)

Ordering property
- Same as for BST

Proving Shallowness Bound

Let \(S(h)\) be the min # of nodes in an AVL tree of height \(h\)

Claim: \(S(h) = S(h-1) + S(h-2) + 1\)

Solution of recurrence: \(S(h) = O(2^h)\) (like Fibonacci numbers)
Testing the Balance Property

We need to be able to:

1. NULLs have height -1

An AVL Tree

NULLs have height -1

<table>
<thead>
<tr>
<th>data</th>
<th>height</th>
<th>children</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>/</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>/</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>/</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>/</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>/</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>/</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>/</td>
</tr>
</tbody>
</table>