Trees

CSE 373
Data Structures & Algorithms
Ruth Anderson
Spring 2008

Today’s Outline

• Admin:
  – HW #1 due thurs 4/10 at 11:59pm,
  – Bring printouts to class Friday 4/11
• Math
  • Trees!

Math Fundamentals &
Asymptotic Analysis

Powers of 2

• Many of the numbers we use in Computer
  Science are powers of 2
• Binary numbers (base 2) are easily represented in
digital computers
  – each “bit” is a 0 or a 1
  – an n-bit wide field can represent how many different
    things?

Unsigned binary numbers

• For unsigned numbers in a fixed width field
  – the minimum value is 0
  – the maximum value is $2^n - 1$, where $n$ is the
    number of bits in the field
  – The value is $\sum_{i=0}^{i=n-1} a_i 2^i$
• Each bit position represents a power of 2
  with $a_i = 0$ or $a_i = 1$

Logs and exponents

• Definition: $\log_2 x = y$ means $x = 2^y$
  – 8 = $2^3$, so $\log_8 8 = 3$
  – 65536 = $2^{16}$, so $\log_{65536} 65536 = 16$
• Notice that $\log_2 x$ tells you how many bits are
  needed to hold x values
  – 8 bits holds 256 numbers: 0 to $2^8 - 1 = 0$ to 255
  – $\log_2 256 = 8$
Floor and Ceiling

\[
\left\lfloor X \right\rfloor \quad \text{Floor function: the largest integer } \leq X
\]

\[
\left\lfloor 2.7 \right\rfloor = 2 \quad \left\lfloor -2.7 \right\rfloor = -3 \quad \left\lfloor 2 \right\rfloor = 2
\]

\[
\left\lceil X \right\rceil \quad \text{Ceiling function: the smallest integer } \geq X
\]

\[
\left\lceil 2.3 \right\rceil = 3 \quad \left\lceil -2.3 \right\rceil = -2 \quad \left\lceil 2 \right\rceil = 2
\]

Properties of logs

• We will assume logs to base 2 unless specified otherwise
• \( \log AB = \log A + \log B \)  
  – \( A = 2^{\log_2 A} \) and \( B = 2^{\log_2 B} \)  
  – \( AB = 2^{\log_2 A \times \log_2 B} \)  
  – so \( \log_2 AB = \log_2 A + \log_2 B \)  
  – [note: \( \log AB \neq \log A \times \log B \)]

Other log properties

• \( \log A/B = \log A - \log B \)  
• \( \log (A^B) = B \log A \)  
• \( \log \log X < \log X < X \) for all \( X > 0 \)  
  – \( \log \log X = Y \) means \( 2^Y = X \)  
  – \( \log X \) grows slower than \( X \)  
  • called a “sub-linear” function
A log is a log is a log

- Any base $x$ log is equivalent to base 2 log within a constant factor

$$\log_B x = \log_2 x \cdot \log_2 B$$

Trees
BSTs, and AVL Trees
Chapter 4 in Weiss

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

Traversals

```java
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

Binary Tree: Special Cases

A

B

C

D

E

F

Complete Tree

Perfect Tree

Full Tree

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)

A Modest Few Uses

• Student, Customer records
• Networks: Router tables
• Operating systems: Page tables
• Compilers: Symbol tables

Probably the most widely used ADT!

Implementations

insert find delete

• Unsorted Linked-list

• Unsorted array

• Sorted array
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

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;
}
```

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 \( O(\log n) \)
  – 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 \textit{height}

Potential Balance Conditions
3. Left and right subtrees of \textit{every node} have equal number of nodes

4. Left and right subtrees of \textit{every node} have equal \textit{height}
The AVL Balance Condition
Left and right subtrees of every node have equal heights differing by at most 1

Define: \( \text{balance}(x) = \text{height}(x\text{.left}) - \text{height}(x\text{.right}) \)

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 \( \Theta(\log n) \)

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. 
2. 
3. 

\text{NULL}s have height \(-1\)

An AVL Tree
AVL tree of height \( h=4 \) with the min # of nodes

AVL tree with height information and data labels.
AVL trees: find, insert

- **AVL find:**
  - same as BST find.

- **AVL insert:**
  - same as BST insert, except may need to “fix” the AVL tree after inserting new value.

AVL tree insert

Let \( x \) be the node where an imbalance occurs.

Four cases to consider. The insertion is in the

1. left subtree of the left child of \( x \).
2. right subtree of the left child of \( x \).
3. left subtree of the right child of \( x \).
4. right subtree of the right child of \( x \).

**Idea:** Cases 1 & 4 are solved by a **single rotation**.
Cases 2 & 3 are solved by a **double rotation**.

Bad Case #1

- Insert(6)
- Insert(3)
- Insert(1)

Fix: Apply Single Rotation

AVL Property violated at this node (x)

Single Rotation example
Bad Case #2
Insert(1)
Insert(6)
Insert(3)

Fix: Apply Double Rotation
AVL Property violated at this node (x)

Double Rotation
1. Rotate between x’s child and grandchild
2. Rotate between x and x’s new child

Double rotation in general

Height of tree before? Height of tree after? Effect on Ancestors?

Double rotation, step 1

Double rotation, step 2

Imbalance at node X

Single Rotation
1. Rotate between x and child

Double Rotation
1. Rotate between x’s child and grandchild
2. Rotate between x and x’s new child
Insert into an AVL tree: a b c d

Student Activity: Circle your final answer

Insertion into AVL tree
1. Find spot for new key
2. Hang new node there with this key
3. Search back up the path for imbalance
4. If there is an imbalance:
   - case #1: Perform single rotation and exit
   - case #2: Perform double rotation and exit

   Both rotations keep the subtree height unchanged.
   Hence only one rotation is sufficient!

Easy Insert
Insert(3)

Hard Insert (Bad Case #1)
Insert(33)

Single Rotation

Single and Double Rotations:
Inserting what integer values would cause the tree to need a:
1. single rotation?
2. double rotation?
3. no rotation?
Hard Insert (Bad Case #2)

Insert(18)

Unbalanced?
How to fix?

Single Rotation (oops!)

Double Rotation (Step #1)

Double Rotation (Step #2)