CSE 326: Data Structures

Topic #5: Binary Search Trees

Ashish Sabharwal
Autumn, 2003

Today’s Outline

• Admin: Written homework #1 is out!
• Quick Tree Review
• Binary Trees
• Dictionary ADT / Search ADT
• Binary Search Trees

Tree Calculations

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

Find the height of the tree...

runtime:

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

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

Complete Tree

Perfect Tree

Full Tree

Binary Tree: Some Numbers!

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

What’s the average tree height for $n$ nodes, assuming all distinct trees of $n$ nodes are equally likely?

ADTs Seen So Far

- Stack
  - Push
  - Pop
- Queue
  - Enqueue
  - Dequeue
- List
  - Insert
  - Remove
  - Find
- Priority Queue
  - Insert
  - DeleteMin

Remember decreaseKey?

New! The Search ADT

- Data:
  - unique user-specified keys
  - Or: a set of keys

- Operations:
  - Insert (key)
  - Find (key)
    - Checks for membership
    - Remove (key)

Also New! The Dictionary ADT

- Data:
  - values mapped to user-specified keys
  - Or: a set of (key, value) pairs

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

The Search ADT is sometimes called the "Set ADT"

An easy extension of the Search ADT!
A Modest Few Uses

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

Probably the most widely used ADT!

Naïve Implementations

- Insert
- Find
- Delete

- Unsorted Linked-list
- Unsorted array
- Sorted array

What limits the performance?

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

Find in BST, Iterative

Node

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;
}

Node

Node Find(Object key, Node root) {
  Node root;
  while (root != NULL && root.key != key) {
    if (key < root.key)
      root = root.left;
    else
      root = root.right;
  }
  return root;
}

Runtime:
Binary Search vs. Binary Search Tree

A well balanced binary search tree allows \(O(\log n)\) time binary search!

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.

Analysis of BuildTree

- Worst case: \(O(n^2)\) as we’ve seen
- Average case assuming all orderings equally likely:
  - Sum of all depths:
    - \(D(n) = 2D(i) + D(n-i-1) + (n-1)\)
  - Average depth of a node:
  - Total runtime:

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 – The Leaf Case**

Delete(17)

**Deletion – The One Child Case**

Delete(15)

**Deletion – The Two Child Case**

Delete(5)

**Deletion – The Two Child Case**

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees!

Options:
- \textit{suc}c from right subtree: \textit{findMin}(right)
- \textit{pred} from left subtree: \textit{findMax}(left)

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

\[ \begin{array}{c}
\text{8} \\
\text{7 replaces 5} \\
\text{3} \\
\text{2} \\
\text{1} \\
\end{array} \]

Original node containing 7 gets deleted

To Do

• Start Homework #1
  – Somewhat long but easy
  – Will get you hands on practice with Math background and heaps

• Read chapter 4 in the book