Announcements

• Homework 2 released on website
  • Due Oct. 6\textsuperscript{th} at 5:40 PM (next class)
• Post homework to Shared Files, Homework #2
• Distinguished Lecture
CS Distinguished Lecture

- Network Evolution, Network Economics, and Network Innovation
- Andrew Odlyzko
- Monday, October 5th.
- 11:00 AM Davis Auditorium
  Schapiro/CEPSR
Review

- Lists, Stacks, Queues in Linux
- Introduction to Trees
  - Definitions
  - Tree Traversal Algorithms
- Binary Trees
Today’s Plan

• Search Tree ADT
• Binary search tree
  • implementation
• running time analysis
Search (Tree) ADT

- ADT that allows insertion, removal, and searching by key
  - A key is a value that can be compared
  - In Java, we use the Comparable interface
  - Comparison must obey transitive property
- Search ADT doesn’t use any index
Binary Search Tree Property:
- Keys in left subtree are less than root.
- Keys in right subtree are greater than root.

BST property holds for all subtrees of a BST
Inserting into a BST

- Compare new value to current node, if greater, insert into right subtree, if lesser, insert into left subtree

- `insert(x, Node t)`
  - if `(t == null)` return new Node(x)
  - if `(x > t.key)`, then `t.right = insert(x, t.right)`
  - if `(x < t.key)`, then `t.left = insert(x, t.left)`
  - return t
Searching a BST

• findMin(t)  // return left-most node
  if (t.left == null) return t.key
  else return findMin(t.left)

• search(x,t)  // similar to insert
  if (t == null) return false
  if (x == t.key) return true
  if (x > t.key), then return search(x, t.right)
  if (x < t.key), then return search(x, t.left)
Deleting from a BST

- Removing a leaf is easy, removing a node with one child is also easy.
- Nodes with no grandchildren are easy.
- What about nodes with grandchildren?
A Removal Strategy

- First, find node to be removed, \( t \)
- Replace with the smallest node from the right subtree
  - \( a = \text{findMin}(t.\text{right}); \)
  - \( t.\text{key} = a.\text{key}; \)
- Then delete original smallest node in right subtree
  - \( \text{remove}(a.\text{key}, t.\text{right}) \)
Sorting with BST

- Suppose we have a built BST
- How to print out nodes in order?
  - inorder traversal
- Running time?
  - $O(N)$
Worst Case

- Operations: insert, remove, search
- Worst case insert sequence: 1,2,3,4,5,6
- search(6)
- Insert 1,6,5,4,3,2
  Remove(1)
Average Case Analysis

- All operations run in $O(d)$ time, but what is $d$?
  - Worst case $d = N$
  - Best case $d = \log(N+1)-1$
  - Average case?
Average Case Analysis

• Consider the **internal path length**: the sum of the depths of all nodes in a tree

• Let $D(N)$ be the internal path length for some tree $T$ with $N$ nodes*.

• Suppose $i$ nodes are in the left subtree of $T$.

• Then $D(N) = D(i) + D(N - i - 1) + N - 1$
Average Case Analysis

- $D(N) = D(i) + D(N - i - 1) + N - 1$

- Assume all insertion sequences are equally likely

- Subtree sizes only depend on the 1st key inserted
  - all subtree sizes equally likely

- Average of $D(i)$ (and $D(N-i-1)$) is $rac{1}{N} \sum_{j=0}^{N-1} D(j)$
Average Case Analysis

- Average case $D(N)$ then becomes

\[ D(N) = \frac{2}{N} \left[ \sum_{j=0}^{N-1} D(j) \right] + N - 1 \]

- This is a recurrence, which can be solved to show that $D(N) = O(N \log N)$

- (page 272-273 in Weiss)

- Then average depth over all $N$ nodes is $O(\log N)$
How do we implement Search Trees that explicitly avoid worst case O(N) operations?

Intuition: try to keep the tree balanced

What is the cost of this balancing?
Reading

• This class: Section 4.3
• Next class: Section 4.4 – AVL Trees