G205
Fundamentals of Computer Engineering
Stefano Basagni
Fall 2004
M-W, 1:30pm-3:10pm
Search Trees

- Dynamic data structures supporting dynamic set operations:
  - Search
  - Minimum, maximum
  - Predecessor, successor
  - Insert, delete
- Implement dictionary and priority queues
Binary Search Trees (BSTs)

- Binary trees
- Each node is an object that contains:
  - Key, satellite data
  - Pointer left to the left child
  - Pointer right to the right child
  - Pointer p to the parent
- The root node is the only one with the pointer p = NIL
- Each leaf of the tree has left = right = NIL
Binary Search Tree Property

Keys satisfy the

BINARY SEARCH PROPERTY

Let $x$ be a node in a BST. If $y$ is a node in the left subtree of $x$ then $\text{key}[y] \leq \text{key}[x]$. If $y$ is a node in the right subtree of $x$ then $\text{key}[x] \leq \text{key}[y]$
Visiting a Tree

- Printing out the keys of the tree nodes
- **Preorder** tree visit: Prints the keys in the root, then the keys in the left subtree, then the keys in the right subtree
- **Inorder** tree visit: Left subtree, root, right subtree
- **Postorder** tree visit: Left subtree, right subtree, root
Visiting a BST

- An inorder visit prints out all the keys in sorted order

Inorder-Tree-Walk(x)

if x ≠ NIL then
  Inorder-Tree-Walk(left[x])
  print(key[x])
  Inorder-Tree-Walk(right[x])
Correctness and Analysis

- Induction, from the binary search tree property
- Theorem: If $x$ is the root of an $n$-node, then $\text{Inorder-Tree-Walk}(x)$ takes $\Theta(n)$ time
- Proof: Substitution method by proving that $T(n) = (c+d)n + c$ (complete induction)
Querying a Binary Search Tree

Typical search operation in a BST:
- Searching for a key stored in the tree
- Minimum
- Maximum
- Successor
- Predecessor

All in \(O(h)\) time, where \(h\) is the height of the tree
Searching, Recursive Version

Input: pointer t to the root of the tree and the key to be searched

Output: pointer to the node with k or NIL

Tree-Search(x,k)

if x=NIL or k=key[x] return x

if k < key[x]
then return Tree-Search(left[x],k)
else return Tree-Search(right[x],k)
Searching, Iterative Version

- Recursion is “unrolled” into a while loop
  
  It-Tree-Search(x,k)
  
  while x ≠ NIL and k ≠ key[x] do
    if k < key[x]
      then x = left[x]
    else x = right[x]
  
  return x

- Initial call, in both cases: Tree-Search(t,k)
- Both cases: O(h)
Minimum and Maximum

- The minimum element is found by following the left child pointers to a NIL.

Tree-Minimum(x)

\[
\text{while left}[x] \neq \text{NIL do } x = \text{left}[x] \\
\text{return } x
\]

- The binary search property guarantees the correctness of Tree-Minimum.

- Similar code for the maximum.

- Clearly $O(h)$.
Successor, 1

- The successor of a node $x$ is the node with the smallest key bigger than $\text{key}[x]$
- BS property $\Rightarrow$ No comparisons are needed!
- Two cases:
  - If the right subtree is non-empty: The successor of $x$ is the minimum in its right subtree
  - If the right subtree is empty: If $x$ has a successor $y$, then $y$ is the lowest ancestor of $x$ whose left child is also an ancestor of $x$
Successor, 2

Tree-Successor(x)
  if right[x] ≠ NIL
    then return Tree-Minimum(right[x])
  y = p[x]
  while y ≠ NIL and x = right[y] do
    x = y
    y = p[y]
  return y

◆ The time complexity is O(h) (we either go down a path, or up)
Insertion and Deletion

- Insertion and deletion cause a dynamic set to change
- The BST that represents that set changes too
- The BST must be modified to reflect the change but the binary search property must continue to hold
Insertion, 1

**Input:**
- root $t$
- pointer $z$ to a node such that
  - $\text{key}[z] = v$
  - $\text{left}[z] = \text{NIL}$
  - $\text{right}[z] = \text{NIL}$

**Begins at the root and traces a path downward.**

**Pointer $x$ traces a path and $y$ is maintained as the parent of $x$.**
Insertion, 2

Tree-Insert(t, z)
  y = NIL
  x = t
  while x ≠ NIL do // Traces the path
    y = x
    if key[z] < key[x]
      then x = left[x]
      else x = right[x]
    p[z] = y // z’s parent is initialized
Insertion, 3

if y = NIL
    then \( t = z \) // Tree was empty
else if \( \text{key}[z] < \text{key}[y] \)
    then \( \text{left}[y] = z \)
else \( \text{right}[y] = z \)

A new node is always inserted as a leaf

Time complexity: \( O(h) \)
Deletion, 1

Input:
- root t
- pointer z to the node to be deleted

Three cases:
1. z has no children \( \rightarrow p[z]=\text{NIL} \)
2. z has only one child: z is “spliced out”
3. z has two children: z’s successor y with no left child is spliced out and z and y data are swapped

Time complexity: \( O(h) \)
Deletion, 1

Tree-Delete(t,z)
if left[z]=NIL or right[z]=NIL
then y=z       // Find y in three cases
else y=Tree-Successor(z)
if left[y]≠NIL // x=non-NIL child of y
then x=left[y]
else x=right[y]
if x≠NIL then p[x]=p[y] // y is spliced out
Deletion, 2

if \( p[y] = \text{NIL} \)
then \( t = x \)
else if \( y = \text{left}[p[y]] \)
then \( \text{left}[p[y]] = x \)
else \( \text{right}[p[y]] = x \)

if \( y \neq z \)
then \( \text{key}[z] = \text{key}[y] \)

\( \text{copy y's satellite data into z} \)

return \( y \)  // To be recycled
Randomly Built BSTs

- All operations on a BST take $O(h)$
- $h$ can vary depending on insertion
- Worst case: items are inserted in strictly increasing order: $h = n-1$
- It can be shown that $h \geq \log n$
- It can be proven that the average height $h$ of a BST is in $O(\log n)$
Assignments

- Textbook, Chapter 12, pages 253—264
- Updated information on the class web page:

  www.ece.neu.edu/courses/eceg205/2004fa