Trees

- Rooted Tree: collection of nodes and edges
  - Edges go down from root (from parents to children)
  - No two paths to the same node
  - Sometimes, children have “names” (e.g. left, right, middle, etc)
Where are trees useful?

- Some important applications
  - Parse trees (compilers)
  - Search trees
  - String search
  - Game trees (e.g. chess program)
- E.g. parse tree for $3*(4+2)*(2^6)*(8/3)$
Binary Search Trees

• Binary tree: every node has 0, 1 or 2 children
• BST property:
  ➢ If $y$ is in left subtree of $x$, then $\text{key}[y] \leq \text{key}[x]$
  ➢ same for right
Binary Search Trees

• Binary tree: every node has 0, 1 or 2 children
• BST property:
  ➢ If $y$ is in left subtree of $x$, then $\text{key}[y] \leq \text{key}[x]$
  ➢ same for right
Implementation

- Keep pointers to parent and both children
- Each NODE has
  - left: pointer to NODE
  - right: pointer to NODE
  - p: pointer to NODE
  - key: real number (or other type that supports comparisons)
Drawing BST’s

- To help you visualize relationships:
  - Drop vertical lines to keep nodes in right order
- For example, quick look at picture tells you the only legal location to insert a node with key 4
Searching a BST

\[
\text{TREE-SEARCH}(x, k)
\]

\[
\begin{array}{l}
\text{if } x == \text{NIL or } k == \text{key}[x] \\
\quad \text{return } x \\
\text{if } k < x.\text{key} \\
\quad \text{return } \text{TREE-SEARCH}(x.\text{left}, k) \\
\text{else return } \text{TREE-SEARCH}(x.\text{right}, k)
\end{array}
\]

Initial call is \text{TREE-SEARCH}(T.\text{root}, k).

- Running time: $\Theta(\text{height})$
Insertion

\[
\text{TREE-INSERT}(T, z) \\
y = \text{NIL} \\
x = T.\text{root} \\
\text{while } x \neq \text{NIL} \\
\quad y = x \\
\quad \text{if } z.\text{key} < x.\text{key} \\
\text{\quad } x = x.\text{left} \\
\quad \text{else } x = x.\text{right} \\
z.p = y \\
\text{if } y == \text{NIL} \\
\quad T.\text{root} = z \quad // \text{ tree } T \text{ was empty} \\
\text{elseif } z.\text{key} < y.\text{key} \\
\quad y.\text{left} = z \\
\text{else } y.\text{right} = z
\]

- Find location of insertion, keep track of parent during search
- Running time: $\Theta(\text{height})$
Tree Min and Max

TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x

TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x

- Running time: $\Theta(\text{height})$
Tree Successor

\[\text{Tree-Successor}(x)\]

\[\begin{align*}
\text{if } x & \neq \text{NIL} \\
& \quad \text{return Tree-Minimum}(x) \\
y & = x \\
\text{while } y \neq \text{NIL} \text{ and } x == y.
& \quad x = y \\
& \quad y = y \\
& \quad \text{return } y
\end{align*}\]

- Running time: \(\Theta(\text{height})\)
Height of a BST can be...

- ... as little as log(n)
  - full balanced tree

- ... as much as $n-1$
  - unbalanced tree
Exercises

• Write pseudocode for finding
  ➢ max element in a tree
  ➢ predecessor
  ➢ second smallest element in a tree

• Printing elements in order
Inorder traversal

\[ \text{INORDER-Tree-Walk}(x) \]
\[
\text{if } x \neq \text{NIL} \\
\quad \text{INORDER-Tree-Walk}(x \cdot \text{left}) \\
\quad \text{print key}[x] \\
\quad \text{INORDER-Tree-Walk}(x \cdot \text{right})
\]

- Running time: \( \Theta(n) \)
Postorder traversal

• Write pseudocode that computes
  ➢ height of a BST
  ➢ number of nodes in a BST
  ➢ average depth
  ➢ ...

• Example: height
  ➢ Find-height(T)
    • if T==NIL return -1
    • else
      – h1 := Find-height(T.left)
      – h2 := Find-height(T.left)
      – return max(h1, h2) + 1
Preorder Traversal

• Recall that insertion order affects the shape of a BST
  ➢ insertion in sorted order: height n-1
  ➢ random order: height O(log n) with high probability (we will prove that later in the course)

• Write pseudocode that prints the elements of a binary search tree in a plausible insertion order
  ➢ that is, an insertion order that would produce this particular shape

• (print nodes in a preorder traversal)
Exercise on insertion order

- Exercise: Write pseudocode that takes a sorted list and produces a “good” insertion order (that would produce a balanced tree)
- (Hint: divide and conquer: always insert the median of a subarray before inserting the other elements)
Deletion

**Tree-Delete** \((T, z)\)

- **if** \(z.\text{left} \equiv \text{NIL}\)
  - **then** \(\text{TRANSPLANT}(T, z, z.\text{right})\)  // \(z\) has no left child
- **elseif** \(z.\text{right} \equiv \text{NIL}\)
  - **then** \(\text{TRANSPLANT}(T, z, z.\text{left})\)  // \(z\) has just a left child
- **else** // \(z\) has two children.
  - \(y = \text{TREE-MINIMUM}(z.\text{right})\)  // \(y\) is \(z\)’s successor
  - **if** \(y.\text{p} \neq z\)
    - // \(y\) lies within \(z\)’s right subtree but is not the root of this subtree.
      - \(\text{TRANSPLANT}(T, y, y.\text{right})\)
      - \(y.\text{right} = z.\text{right}\)
      - \(y.\text{right.p} = y\)
    - // Replace \(z\) by \(y\).
      - \(\text{TRANSPLANT}(T, z, y)\)
      - \(y.\text{left} = z.\text{left}\)
      - \(y.\text{left.p} = y\)

- **Cases analyzed in book**
- **Running time:** \(\Theta(\text{height})\)