Trees

Data Structures

Ching-Fang Hsu
Department of Computer Science and Information Engineering
National Cheng Kung University

Introduction -- Terminology

- **Definition** (Tree)
  - A *tree* is a finite set of one or more nodes such that:
    - There is a specially designed node called *root*.
    - The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$, where each of these sets is a tree.
      - $T_1, \ldots, T_n$ are the subtrees of the root.
- Subtrees are prohibited from ever connecting together.
- Every node in the tree is the root of some subtree.

Introduction -- Terminology (contd.)

- The *degree of a node*
  - The number of subtrees of the node
- The *degree of a tree*
  - The maximum degree of the nodes in the tree
- A *leaf/terminal* node
  - A node with degree zero

Introduction -- Terminology (contd.)

- The *parent* (*children*) of a node
  - Given a node X and its subtrees $T_1, \ldots, T_n$, which are rooted at node $r_1, \ldots, r_n$, respectively.
    - X is the parent of $r_1, \ldots,$ and r_n. In other words, $r_1, \ldots,$ and r_n are X's children.
- *Siblings*
  - Children of the same parent
- The *ancestors* of a node
  - All the nodes along the path from the root to the node

Introduction -- Terminology (contd.)

- The *descendents* of a node
  - All the nodes that are in its subtrees
- The *level* of a node
  - The root is at level one.
  - Otherwise, the level is the level of its parent plus one.
- The *height*/*depth* of a tree
  - The maximum level of any nodes in the tree

Introduction -- Representation of Trees

- List Representation
  - Write a tree as a list in which each of the subtrees is also a list
    - Example: (p.193, Fig. 5.2)

      (A (B (E (K, L), F), C(G), D(H (M), I, J)))

Introduction -- Representation of Trees in Memory

- Linked lists
  - A node with varying number of fields
    - p. 195, Fig. 5.4
  - Each link represents a child of the node.
- Left Child-Right Sibling Representation
  - Exactly two link or pointer fields per node
    - p.195 Fig. 5.5
  - The order of children in a tree is not important.
    - Any of the children of a node could be its leftmost child and any of its siblings could be the closest right sibling.
    - Example: p. 196, Fig. 5.6

Binary Trees

- **Definition** (Binary Trees)
  - A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.
- The chief characteristics of a binary tree
  - The degree of any given node must not exceed two.
  - The order of subtrees is not irrelevant any more.
  - May have zero nodes

Binary Trees (contd.)

- The binary tree ADT (p.199, ADT 5.1)
- A binary tree vs. A tree
  - An empty tree is invalid while a binary tree may have zero nodes.
  - The order of subtrees is irrelevant in a tree while the order of children is distinguishable in a binary tree.
    - p. 199, Fig. 5.9

Binary Trees (contd.)

- Two special types of binary trees
  - Skewed trees
    - Skewed to the left or to the right (p. 200, Fig. 5.10(a))
  - Complete binary trees
    - => All the leaf nodes are on two adjacent levels. (p. 200, Fig. 5.9(b))

Binary Trees -- Properties

- **Lemma 5.2** [*Maximum number of nodes*]:
  - The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$
  - The maximum number of nodes in a binary tree of depth k is 2^k -1, $k \geq 1$
  - proof: ref. p. 200~201
- **Lemma 5.3** [*Relation between number of leaf nodes and nodes of degree 2*]:
  - For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Binary Trees -- Properties (contd.)

- **Definition** (Full Binary Trees)
  - A *full binary tree* of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$.
  - A numbering scheme
    - Starting with the root on level 1, continue with the nodes on level 2, and so on.
    - Nodes on any level are numbered from left to right.
    - p. 202, Fig. 5.11

Binary Trees -- Properties (contd.)

- **Definition** (Complete Binary Trees)
  - A binary tree with n nodes and depth k is complete *iff* its nodes correspond to the nodes numbered from 1 to n in the binary tree of depth k.

Binary Trees -- Representation

- Array Representation
  - A one-dimensional array
    - The 0th position of the array is a dummy element.
  - **Lemma 5.4**: If a complete binary tree with n nodes (depth = $\lfloor \log_2 n + 1 \rfloor$) is represented sequentially, then for any node with index i, $1 \leq i \leq n$, we have:
    - *parent* (i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$, i is at the root and has no parent.
    - *left_child* (i) is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.
    - *right_child* (i) is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, then i has no right child.

Binary Trees -- Representation (contd.)

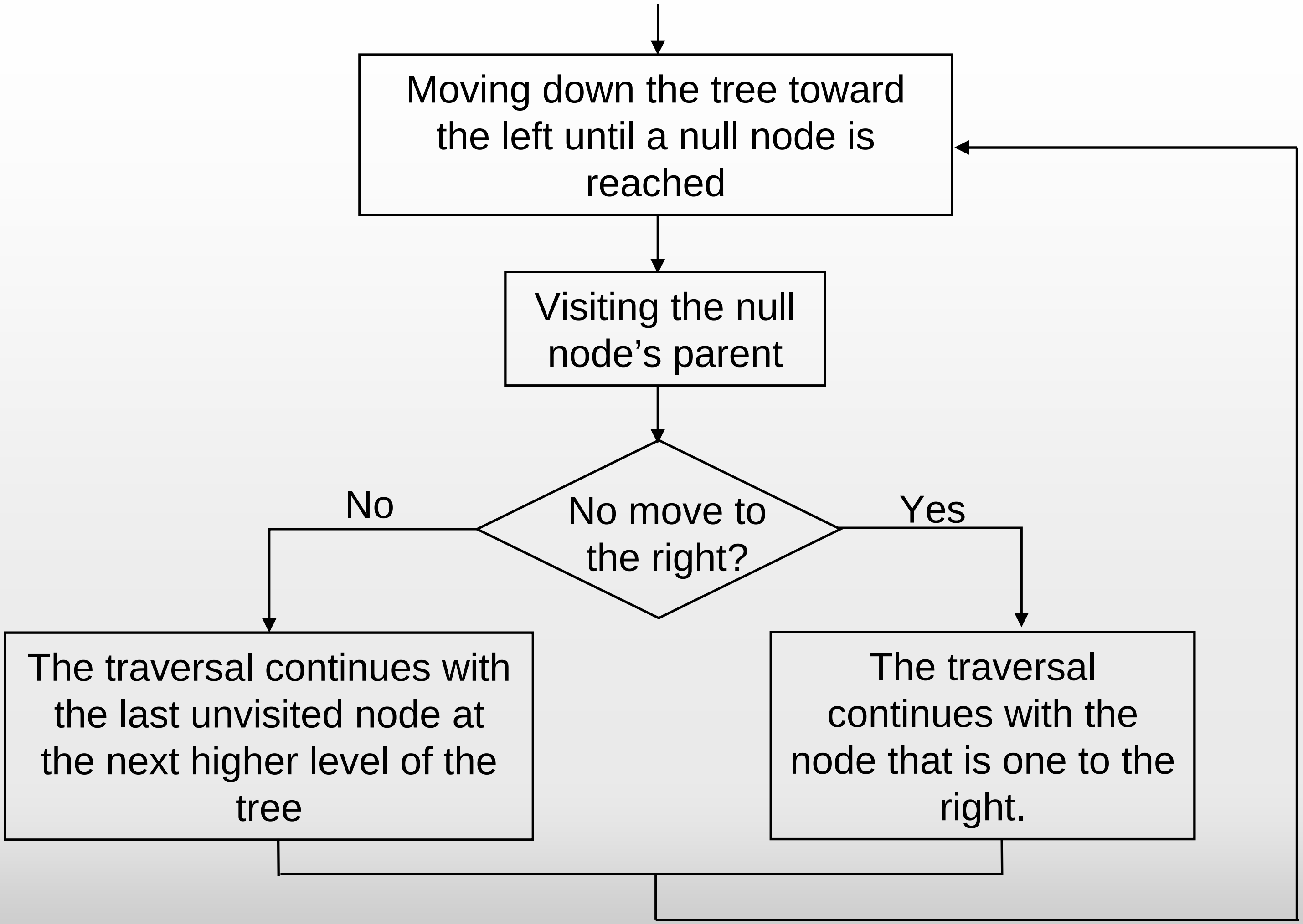
- In the worst case, a skewed tree of depth k requires 2^k-1 spaces
  - Only k spaces will be occupied.
- Disadvantages
  - A waste of space
  - The general inadequacies of sequential representation

- Linked Representation
  - Three fields (p. 204)
    - *left_child*, *data*, and *right_child*
  - A fourth field, *parent*, is added if it is necessary to know the parents of random nodes.

Binary Tree Traversals

- What is "tree traversal"?
  - Visiting each node in the tree exactly once
- Notations
  - *L* -- Moving left
  - *V* -- Visiting the node
  - *R* -- Moving right
- Three possible traversals if we traverse left before right (Example: p. 206)
  - *LVR* (inorder), *LRV* (postorder), and *VLR* (preorder)

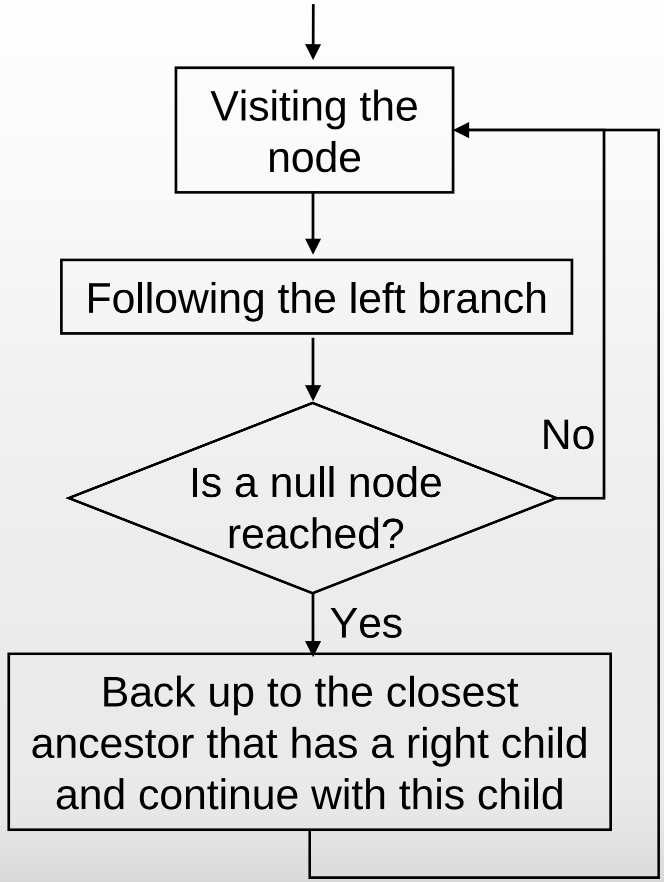
Binary Tree Traversals -- Inorder Traversal



Binary Tree Traversals -- Inorder Traversal (contd.)

- Recursive inorder traversal (p. 207, Program 5.1)
- For a binary tree with an arithmetic expression, the inorder traversal would produce the infix form of the expression.
- Iterative inorder traversal (p. 210, Program 5.4)
  - To simulate the recursion, we must create a stack.
  - The time complexity and space complexity are both $O(n)$.

Binary Tree Traversals -- Preorder Traversal



Binary Tree Traversals -- Preorder Traversal (contd.)

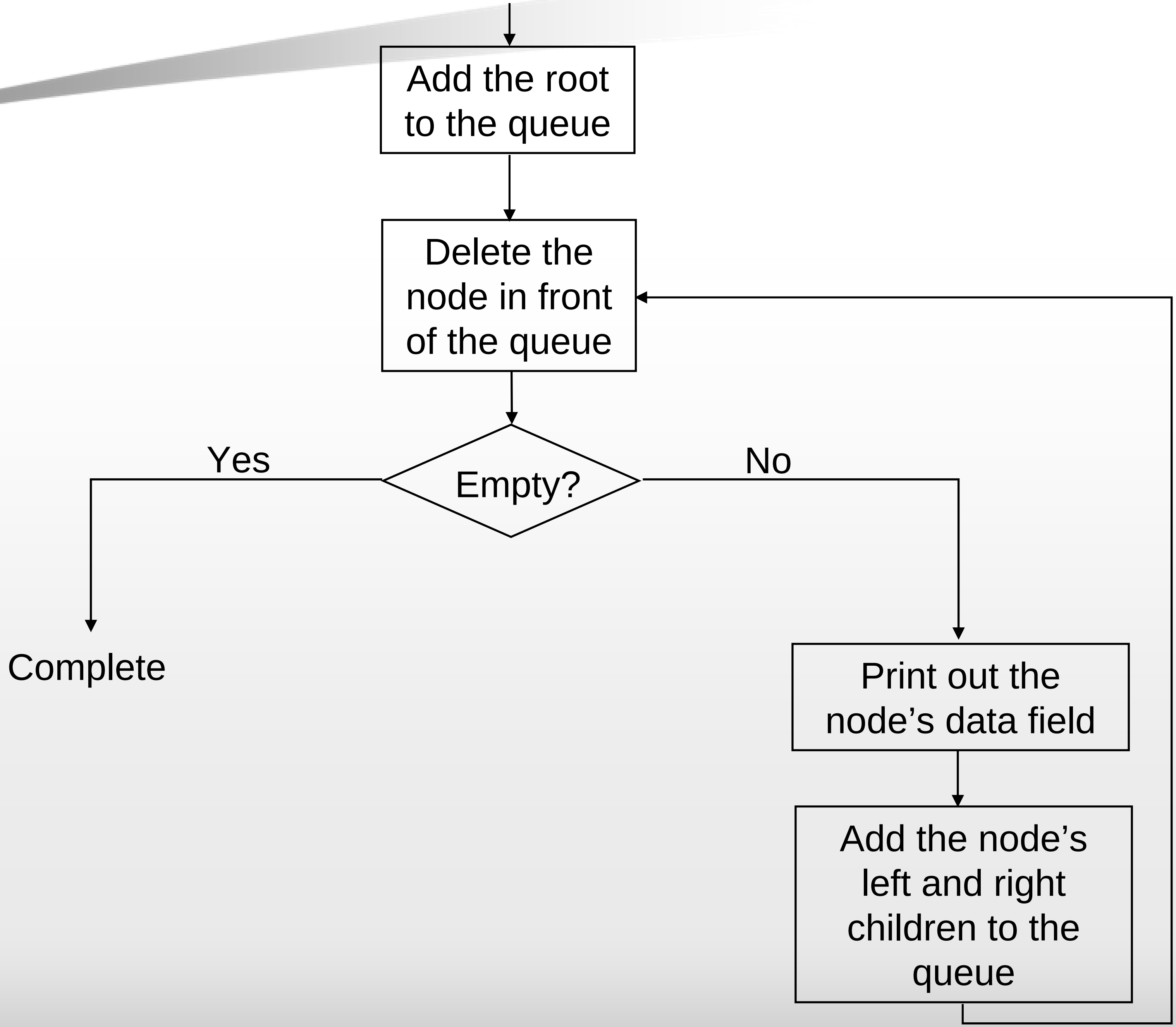
- Recursive preorder traversal (p. 208, Program 5.2)
- Using a preorder traversal, the nodes of a binary tree with arithmetic expression can be output as the prefix form of the expression.

Binary Tree Traversals -- Postorder Traversal

- Postorder traversal visits a node’s two children before it visits the node
  - A node’s children will be output before the node.
  - p. 209, Program 5.3
  - Postfix forms

Binary Tree Traversals -- Level Order Traversal

- This type of traversal requires a queue.
- Visits the nodes using the ordering scheme shown in Fig. 5.11
  - p. 211, Program 5.5
    - A circular queue is used.


Add the root to the queue
Delete the node in front of the queue
Empty?
Yes
No
Complete
Print out the node's data field
Add the node's left and right children to the queue

The Heap Abstract Data Type

- **Definition** (Max (Min) Trees)
  - A *max* (*min*) *tree* is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any).
- **Definition** (Max (Min) Heaps)
  - A *max* (*min*) *heap* is a complete binary tree that is also a max (min) tree.
- An array can be used to represent a heap.
  - The addressing scheme provided by Lemma 5.3

The Heap Abstract Data Type (contd.)

- The basic operations of the ADT of a max heap (p. 223, ADT 5.2)
  - Creation of an empty heap
  - Insertion of a new element into the heap
  - Deletion of the largest element from the heap
- The real challenge is the design of the representation of a heap for efficient insertion and deletion.

The Heap Abstract Data Type -- Priority Queues

- One of applications of heaps
  - Note: Heaps are only one way to implement priority queues.
  - The insertion and deletion times for several representations of priority queues

The Heap Abstract Data Type -- Insertion into A Max Heap

- Example: p. 226, Fig. 5.27
- Implementation of heap insertion
  - Go from an element to its parent
    - How to get a node's parent?
      - A parent field is added if we use linked representation.
      - It is much easier if we choose the array representation for a heap since a heap is a complete binary tree.
  - p. 227, Program 5.13
  - Time complexity: $O(\log n)$

The Heap Abstract Data Type -- Deletion from A Max Heap

- Step 1: Take the deleted element from the root of the heap.
- Step 2: Move down the heap, compare and exchange parent and child nodes until the heap definition is re-established.
  - Example: p. 228, Fig. 5.28
  - p. 229, Program 5.14
- Time complexity: $O(\log n)$

Binary Search Trees

- ❖ A heap is not well suited for applications in which we must delete arbitrary elements.
- ❖ **Definition** (Binary Search Trees)
  - ❑ A *binary search tree* is a binary tree. If it is not empty it satisfies the following properties:
    - ◆ Every element has a key, and no two elements have the same key, i.e., the keys are unique. ⇐ Redundant!
    - ◆ The keys in a nonempty left subtree must be smaller than the key in the root of the subtree.
    - ◆ The keys in a nonempty right subtree must be larger than the key in the root of the subtree.
    - ◆ The left and right subtrees are also binary search trees.

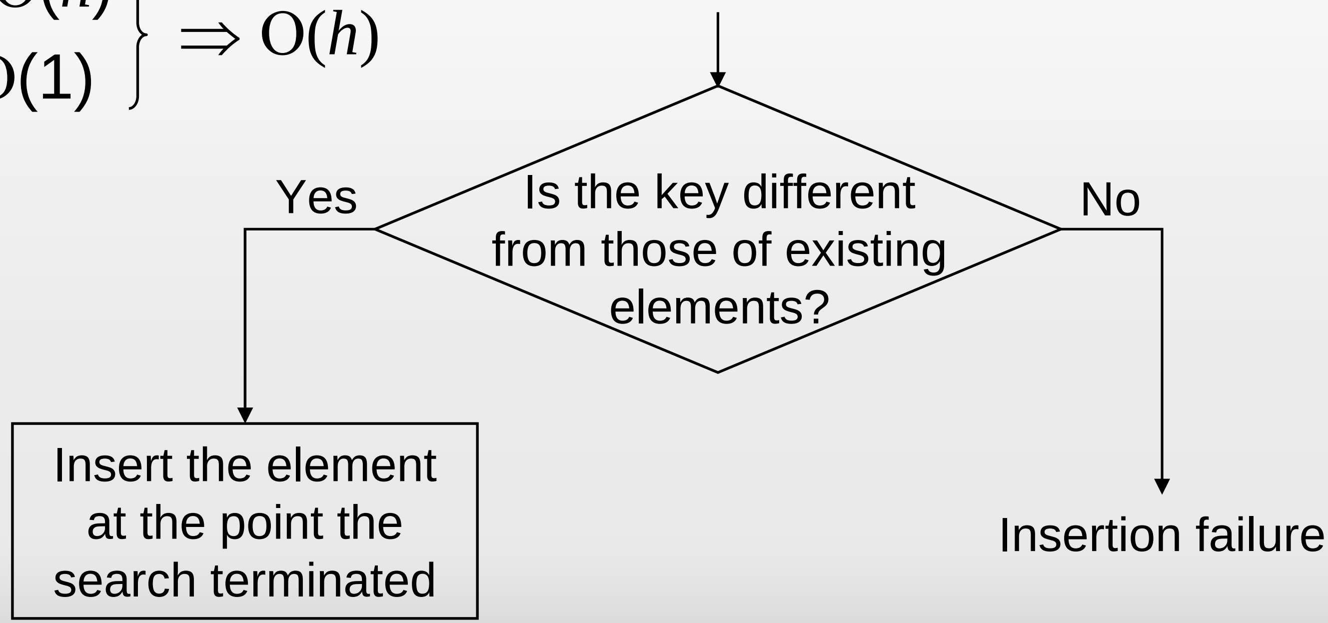
Binary Search Trees -- Search

- p. 233, Program 5.15 (Recursive version)
- p. 233, Program 5.16 (Iterative version)
- Analysis
  - If h is the height of the binary search tree
    - Recursive version: $O(h)$
      - Additional stack space requirement: $O(h)$
    - Iterative version: $O(h)$

Binary Search Trees -- Insertion

- p. 235, Program 5.17
- Analysis

$$\left.\begin{array}{l}\text{Search: } \mathrm{O}(h)\\ \text{Attach: } \mathrm{O}(1)\end{array}\right\} \Rightarrow \mathrm{O}(h)$$



Binary Search Trees -- Deletion

- Deletion of a leaf node is easy.
- For the nonleaf node case
  - Replace the target node with either the largest element in its left subtree or the smallest element in its right subtree.
  - Delete the replacing element from the subtree
- Time complexity: $O(h)$

Set Representation

- Assumptions
  - The elements of the set are the numbers 0, 1, 2, …, n-1.
  - The sets being represented are pairwise disjoint.
    - If S_i and S_j are two sets and $i \neq j$, then there is no element that is in both S_i and S_j.
    - Example: S_1 = {0, 6, 7, 8}, S_2 = {1, 4, 9}, S_3 = {2, 3, 5}
  - For each set, link the nodes from the children to the parent.
    - p. 248, Fig, 5.37

Set Representation (contd.)

- The minimal operations
  - *Disjoint set union (union (i, j))*
    - If S_i and S_j are two disjoint sets, then $S_i \cup S_j$ = { $x \mid x \in S_i$ or $x \in S_j$}
  - *find(i)*
    - Find the set containing the element, i.
- For simplicity, each set is identified by its root of the tree representing it.
  - Example: We refer to S_1 as 0.

Set Representation (contd.)

- Each node needs only one field, the index of its parent.
  - The only data structure needed is an array, as depicted in Fig. 5.40 on p. 249.
  - Root nodes have a parent of -1.
  - *union* (i, j) (p. 250, Program 5.19)
    - Assuming that the convention is that the first tree becomes a subtree of the second, `parent[i] = j`.
  - *find*(i) (p. 250, Program 5.19)
    - Follow the indices starting at i and continue until a negative parent index is reached.

Set Representation (contd.)

- Analysis
  - Performance characteristics are not very good, especially for a series of find operations over a degenerate tree.
  - Example: p. 251, Fig. 5.41
  - The total time needed to process n-1 finds is: $\sum_{i=2}^{n} i = O(n^2)$
- How to avoid the creation of degenerate trees?
  - Solution: Adopt Weighting Rule for union(i, j)!

Definition (Weighting Rule for union(i, j))

- If the number of nodes in tree i is less than the number in tree j then make j the parent of i; otherwise make i the parent of j.

Set Representation (contd.)

- By incorporating the weighting rule, the union operation takes the form given in *WeightedUnion* (p. 252, Program 5.20).
- **Lemma 5.5:** Let T be a tree with n nodes created as a result of *WeightedUnion* . No node in T has level greater than $\lfloor \log_2 n \rfloor + 1$.
  - The time to process a find in an n element tree is $O(\log_2 n)$.
- **Definition [Collapsing rule]:** If j is a node on the path from i to its root then make j a child of the root.

Set Representation (contd.)

- By incorporating the collapsing rule, the find operation takes the form given in *find2* (p. 255, Program 5.21).
  - Roughly doubles the time for an individual find
  - However, the worst case time over a sequence of finds is reduced.
  - Example: p. 253, Example 5.4

Set Representation (contd.)

- **Definition** (Ackermann's function $A(p, q)$)

$$A(p,q) = \begin{cases} 2^q & p = 1 \text{ and } q \geq 1 \\ A(p-1, 2) & p \geq 2 \text{ and } q = 1 \\ A(p-1, A(p, q-1)) & p \geq 2 \text{ and } q \geq 2 \end{cases}$$

- **Definition** ($\alpha(m, n)$, related to a functional inverse of Ackermann's function $A(p, q)$)
  - $\alpha(m, n) = \min \{ z \geq 1 \mid A(z, 4\lceil m/n \rceil) > \log_2 n \}$

Set Representation (contd.)

- **Lemma 5.6** [Tarjan and Van Leeuwen]
  - Let $T(f, u)$ be the maximum time required to process any inter-mixed sequence of f finds and u unions. Assume that $u \geq n/2$ Then:

$$k_1(u + f\alpha(f + u, u)) \leq T(f, u) \leq k_2(u + f\alpha(f + u, u))$$

for some positive constants k_1 and k_2.

Set Representation -- Equivalence Classes

- Regard the equivalence classes to be generated as sets
- How to process an equivalence pair, $i \equiv j$?
  - Determine the sets containing i and j.
    - different $\Rightarrow$ union operation
    - the same $\Rightarrow$ do nothing
  - So, two finds and at most one union are needed to perform for each equivalence pair.
  - Time complexity: $O(n+m\alpha(2m, min\{n-1, m\}))$, if we have n polygons and $m \geq n$ equivalence pairs