Binary Trees

Binary Trees Overview

- Can search a tree quickly, like an ordered array
  - Can use a binary search on an ordered array
  - O(logN) time
- Can insert and delete quickly, like a linked list
  - Inserting and deleting in a linked list is O(1) time
  - Searching is much more tedious

Binary trees

- Nodes – often represent entities or objects
- Edges – represent the way nodes are related
  - Generally restricted to going in one direction – from root downward
  - In a program edges are represented by references kept within the node
- Each node in a binary tree has no more than two children (nodes to which it connects)

Terminology

- Path – sequence of nodes to get to a particular node
- Root – the top of the tree
  - Must be one and only one path from the root to a node
- Parent – single node one level higher than the current node (one parent only)
- Child – nodes below a given node
- Leaf – a node without any children
- Subtree – a given node and all its children
- Visiting – program code arriving at a node to perform some operation on the node
- Traversing – visit all the nodes in a tree in some specified order
- Levels – how many generations from the root
- Keys – value to find or order to maintain
- Binary trees – each node has two or fewer children

Binary search tree

- A tree whereby a node's left child must have a key less than its parent and a node's right child must have a key greater than or equal to its parent
  - Left child is left on a diagram
  - Right child is right on a diagram

Binary trees

- Unbalanced – more nodes one side or the other of the root
  - Data added randomly typically doesn't unbalance a tree
  - Data added in sequence (ascending or descending) unbalances the tree
- Nodes
  - Contain "real" data
  - Contain references to their children

Finding a node

- Given a key value:
  - Start at the root
  - Go to left child if key less than
  - Go to right child if key greater than or equal
  - Repeat till find node or null pointer to next child
- Done in $O(\log_2 N)$ time – why?
- (see applet example and code)

Inserting a node

- First find where to insert it
  - Find where it should *go* – the parent of the one to insert
  - When finds an appropriate child place is null, that's were it is inserted by putting the reference in the parent
  - If no root, make it the root
  - Will always find a place for an insertion
- (see example applet and code)

Traversing a tree

- Inorder
  - Visit nodes in ascending order, based on key value
  - Can be used to create a sorted list
  - 1. Calls itself to traverse the node's left subtree
  - 2. Visit the node
  - 3. Call itself to traverse the node's right subtree
- (see example applet and code)

Preorder and Postorder Transversal

- Used to parse algebraic expressions
- Preorder
  - 1. Visit the node
  - 2. Call itself to traverse the node's left subtree
  - 3. Call itself to traverse the node's right subtree
  - A*(B+C) becomes *A+BC – prefix notation
- Postorder
  - 1. Call itself to traverse the node's left subtree
  - 2. Call itself to traverse then ode's right subtree
  - 3. Visit the node
  - A*(B+C) becomes ABC+* -- postfix notation

Maximum and minimum values

- Minimum: Go to left child until find a null left child, then return its parent
- Maximum: Go to right child until find a null right child, then return its parent

Deleting a node

- Node has no children
  - Set the parent's appropriate child to null
- Node has one child
  - Child of the node to be deleted may either be a left child or a right child
  - Node to be deleted may be either the left child or right child of the parent
- (see example applet and code)

Deleting a node (continued)

- Node has two children
  - Replace the node with its inorder successor (the node with the next highest key to the one to delete)
    - Go to the node's right child, then to successive left children until no more. Last one visited is the inorder successor
    - If successor is right child of node:
      - Unplug current from the right (or left as appropriate) child field of its parent and set the field to point to the successor
      - Unplug the current node's left child and plug it into the left child of the successor
    - If successor is left descendent of right child
      - Plug right child of successor into the left child of the successor's parent
      - Plug the right child of the node to be deleted into the right child of the field of the successor
      - Unplug current form the right child of its parent and set this field to point to the successor
      - Unplug current's left child from current and plug it into the left child of the successor

Efficiency of binary trees

- Operations depend on the number of levels in the tree structure
- Number of levels is related to binary
- Therefore, $O(\log_2 N)$
- For 1,000,000 items that's about 20
  - Unordered array and linked list – about 500,000 to find an item
  - Ordered array is quick in finding but 500,000 to insert an item

Trees represented by arrays

- Position of the reference in the array determines its place in the tree
  - 0 is root
  - 1 is root's left child, 2 is roots right child
  - 3 is left child's left child, 4 is left child's right child
  - Etc.
  - A node's left child is 2 * index of the node + 1
  - A node's right child is 2 * index of the node + 2
  - A node's parent is (index –1) / 2
- Uses some unneeded memory
- Inefficient for deletion (move items in array)

Duplicate keys

- Best if not allowed (as in many real world applications)
- Find needs to check for additional nodes

Summary

- A tree consists of nodes and edges
- Root is the only node without a parent
- No more than two children in a binary tree
- Binary search tree:
  - Left child are less than node
  - Right child is greater than or equal to node
- Perform search, insert, delete in O(logN) time
- Traversing a tree is visiting all nodes in some order (inorder, postorder, preorder)
- Unbalanced tree – root has more left descendents than right or visa versa
- Trees can be represented by arrays but references are more common