Benjamin Fenster
CS-104 (Lee)
10 October 2003

Binary Trees

I. Properties
   a. A binary tree with N nodes has N + 1 null links
      i. Prove by induction
      ii. Base Case (1 node): 2 links, both null
      iii. Inductive Case: Assume property is true for N = i nodes
      iv. N = i nodes (i + 1 null links) Then for all i + 1 we're adding two null links but using one to attach the new node.
   b. The maximum number of nodes in a binary tree of height h is $2^{h+1} – 1$
   c. Let h be the height of a binary tree with N nodes. Then $\text{ceil}(\log_2 N) \le h \le N – 1$
      i. Height is at most N – 1, at least $\text{ceil}(\log_2 N)$
      ii. With N nodes, height is max when completely skewed, min when completely balanced.
      iii. If completely skewed, h = N – 1
      iv. If completely balanced, we already know the height
         1. min = 2^h
         2. max = $2^{h+1} – 1$
         3. $2^h \le N \le 2^{h+1}$
         4. $h \le \log(N)$
         5. If we pick an $N_1 > N$, then $h_1 = \log(N_1)$ and $\log(N_1) > \log(N)$ so $h_1 > h$. Thus, $h \le \log(N)$
         6. On the other side, get $\log(N) < h + 1$ by the same reasoning.
         7. So $\log(N – 1) < h \le \log(N)$ So h = ceil(log N)
         8. QED
   d. Binary Search Tree
      i. Definition: A binary tree that has a key value associated with each node and satisfies the following property:
      ii. For all n_j in the left subtree and for all j_k in the right subtree, $n_j.\text{value} \le n_i.\text{value} < n_k.\text{value}$

II. Trees Overview
   a. A single node / vertex is a tree
   b. A node with one or more trees linked to the node is a tree (subtrees)
   c. "Recursive data structure"
   d. An edge is a link between exactly two nodes.
   e. The number of edges (connecting nodes) = the number of nodes – 1
   f. Taxonomy
      i. (From most to least general)
      ii. Free Tree: No root, any nodes linked through edges. That is, could pick any node as the root
      iii. Unordered Tree: Has a root but no ordering
      iv. Ordered Tree
         1. Impose an ordering
         2. All child nodes for any particular node are ordered
         3. This is most general definition we mean when we say "tree."
      v. M-Ary Tree: All children for any node number ≤ M
      vi. Binary Tree: M-Ary Tree where M = 2
   g. Terminology
      i. Paths
         1. Path: A list of nodes connected by edges
         2. Path length = Number of edges in the path
         3. Depth of Node = Path length from the root to that node
         4. Height of Node = Longest path from the node to a leaf.
         5. Height of Root called the Tree Height
      ii. Definitions

1. Height Balanced Tree (Balanced Tree)
    a. No leaf is "much farther away" from the root than any other leaf
    b. The definition of "much farther away" depends on which balancing scheme is used.
2. Forrest: A collection of one or more trees

h. Representation
    i. M-Ary Tree
        1. Remember that tree is a recursive definition, so the declaration will also be recursive.
        2. Need only declare "a node," not "a tree."
        3. Either use explicitly named pointers ("left", "right", "middle", .etc) or an array of pointers.
    ii. Varying Number of Nodes
        1. The number of children varies from one node to another
        2. Cannot name links since there's no inherent maximum number
        3. Could use a linked list
        4. Each node needs a pointer to a child and a pointer to a sibling.
        5. By arranging the tree this way, every node has two pointers: it's a binary tree!
        6. Thus *every* ordered tree can be made into a binary tree

III. Operations
- a. Traversal
    - i. Preorder (like prefix math expression) Display, Left, Right
    - ii. Inorder: Left, Display, Right
    - iii. Postorder: Left, Right, Display
- b. Removing a Node
    - i. Promote the maximum value from the left subtree
    - ii. Promote the minimum value from the right subtree
    - iii. Just an implementation decision.
- c. Run-Time
    - i. Single operation (find, findMin, findMax, insert, remove)
        1. Single Node
        2. O(log N) average, O(N) worst
        3. Worst case: A completely skewed tree. Need to look at *every* node to get to the bottom.
        4. Average Case: Maximum height with N nodes, $N = 2^{h+1}$ so max height is $\theta(\log_2 N)$
    - ii. Finding M Nodes
        1. O(M log N) average
        2. O(MN) worst
    - iii. Construction
        1. Worst case, height is always N – 1
        2. So runtime for construction of N nodes using comparisons as metric is 1 + 2 + 3 + … + N-1
        3. Proof of Average Runtimes
            a. Internal path length is the sum of the depth of all nodes of a binary search tree
            b. Let D(N) be the internal path length of a binary search tree with N nodes.
            c. Then, $D(N) = D(i) + D(N - i - 1) + N - 1$ if $N \geq 1$ where $0 \leq i \leq N - 1$ (any value)
            d. $D(i) + D(N - i - 1)$ refers to the fact that all the nodes (excluding one in the root) are in one of the two subtrees.

e. Any node in one of the subtrees is one level lower than it would be if measured from the top. So there are N – 1 nodes in subtrees. Add that amount.
f. Assume equal probability of having 1, 2, …, N-1 nodes in the left subtree. That is, the distribution of i is uniform.
g. $E[D(i)] = {}^{1}/_{N}D(0) + {}^{1}/_{N}\,D(1) + \ldots + {}^{1}/_{N}D(N-1)$
h. $E[D(N-i-1)] = {}^{1}/_{N}D(0) + {}^{1}/_{N}\,D(1) + \ldots + {}^{1}/_{N}D(N-1)$
i. $\therefore E[D(N)] = E[D(i)] + E[D(N-i-1)] =$
$2 \sum(j = 0 \text{ to } N-1)(D(J) + N - 1)$

IV. Balancing Binary Search Trees
  a. Three Approaches
    i. Amortized
      1. Splay trees well-known example
      2. Whenever we insert a new key, bring it to the root (via rotations).
      3. *Tends* to be more balanced
      4. Do work now so it pays off over time.
    ii. Randomized
      1. Randomly decide whether to insert each new node at the leaf or bring the new node to the root via rotations
      2. That decision is made at *every* node on the way down.
      3. Tends to be more balanced
      4. Easy to implement
    iii. Optimized
      1. AVL Tree (oldest approach), Red-Black tree
      2. If we insert a node and discover that it violates a given rule, fix it.
  b. AVL Trees
    i. Simple optimization approach.
    ii. If a property is violated as a result of an insertion / deletion, fix it.
    iii. The AVL Tree Property: “The heights of the two subtrees differ by at most 1.”
    iv. Worst Case Lookup: O(log N)
    v. Fix Heights by Rotation
      1. Zig-Zig
        a. The property is violated because a new node was inserted in the left subtree of the left child.
        b. Rotate Right.
      2. Zig-Zig
        a. Right subtree of the right child.
        b. Rotate Left
      3. Zig-Zag
        a. Right subtree of the left child.
        b. Rotate Left, Rotate Right
        c. Double rotation
      4. Zig-Zag
        a. Left subtree of the right child
        b. Rotate Right, Rotate left
    vi. Runtime of a single node search is O(N)
      1. Proof
      2. It will be good enough to prove that, for an AVL tree of N nodes, height (h) = O(log N)
      3. Let S(h) be the minimum number of nodes with height h in an AVL tree of N nodes. $S(h) \le N$
      4. $S(h) = S(h-1) + S(h-2) + 1$ for $h \ge 2$. $S(0) = 1$, $S(1) = 2$
        a. h = 0, minimum height is 1 node
        b. h = 1, minimum = 2
        c. Imagine left & right subtrees are the same height.

            i. Then $S(h – 1) + S(h – 1) + 1 = S(h)$
            ii. Subtree-Left + Subtree-Right + Root = Height
        d. What if the difference in height is 1? (The maximum allowed)
            i. Then $S(h) = S(h – 1) + S(h – 2) + 1$
            ii. Height = One Side + Other Side + Root
        e. The minimum is where the difference in height is 1. So we use that equation.
    5. Note: $S(h) = Fib(h + 2) – 1$ and $Fib(h) = \phi^h / \sqrt{5}$
    6. $\therefore S(h) \cong (\phi^{h+2} / \sqrt{5}) - 1 \leq N$
    7. $\phi^{h+2} \leq (N + 1) \sqrt{5})$
    8. $h \leq \log_\phi((N + 1) \sqrt{5}) – 2$
    9. $h = O(\log N)$
    10. QED

c. Splay Trees
    i. Balanced binary search tree, built using Amortization approach
    ii. Worst-Case runtime is still O(N). When doing M operations, still have worst-case = O(M log N)
    iii. The idea is to do more work at insertion or search (every operation) with the hope that subsequent searches will be faster.
    iv. If the same node is accessed again soon, our work will pay off.
    v. Based on the notion of temporal locality
        1. If we just accessed something it's very likely we'll need it again soon.
        2. Caching works on the same principle.
        3. Common concept in computer science.
    vi. Do the work with double rotations
        1. Called "splaying"
        2. Only time single rotation is used in splay trees is if an odd number of total rotations is needed
        3. *Tendency* is to keep the splay tree more balanced.
        4. Zig-Zag Case
            a. Order: Bottom-up
            b. Result: Depth of most nodes on the path from the root to the accessed node is reduced to *half*
        5. Zig-Zig Case
            a. Order: Top-down
            b. Result: Height reduced by 1
    vii. Runtime
        1. Average: O(log N), worst O(N)
        2. Worst-case runtime is the same, but that case occurs less frequently than in plain binary search trees.
        3. When dealing with many nodes at once, worst *and* average case is O(M log N)
        4. Faster than AVL trees in terms of main memory

d. 2-3-4 Trees
    i. *Not* a binary search tree! Used as an introduction to red-black trees.
    ii. Three types of node
        1. 2-node
            a. One value, two children
            b. Insert a new value, it becomes a 3-node
        2. 3-node
            a. Two values, three children
            b. Insert a new value, it becomes a 4-node
        3. 4-node
            a. Three values, four children
            b. Cannot insert a new value, so it needs to split.

        iii. Hard to implement, but it's flexible
        iv. It's always *perfectly* balanced!
        v. 4-Node Splits
            1. Move the middle item to the parent
            2. The other two items become children of the parent
            3. The new item is then inserted appropriately into one of the two children
        vi. Bottom-Up Insertion
            1. Find the new leaf; insert the node.
            2. If the node needs to be split, do so.
            3. The parent may then need to be split too.
            4. The split thus may propagate upward toward the root.
        vii. Top-Down
            1. While looking for the new node's home, spilt every 4-node encountered.
            2. This way there's no chance for propagation. It's more efficient.
        viii. Runtime
            1. Searches in 2-3-4 tree with N nodes visit at most log(N) + 1 nodes.
            2. Insertions require fewer visits than that.
    e. Red-Black Trees
        i. The idea is essentially to implement a 2-3-4 tree in binary tree form.
        ii. This has the fastest search/insertion runtime of any binary search tree.
        iii. Coloring
            1. Each node is colored either red or black.
            2. The root is always black.
            3. Whenever a new node is inserted it's colored red.
            4. A "red node" is the same as an incoming "red edge"
            5. Red edges are indicated with a double line.
            6. Red edges connect nodes to make them behave like "clusters"
            7. Cannot have two red edges consecutively
        iv. 4-Node Splits
            1. Case 1
                a. 4-Node is a child of a 2-node
                b. Color-flip the parent in the 4-node
            2. Case 2
                a. 4-node is a child of a 3-node
                b. 2r
                    i. 2r.i: Zig-Zag: Color flip parent in 4-node
                    ii. 2r-ii: Zig-Zig: Color flip the same way, but then rotate
                c. 2l
                    i. 2l-i: Zig-Zig: Color flip and rotate
                    ii. 2l-ii: Zig-Zag: Color flip parent in 4-node
                d. 2m
                    i. 2m-i: Zig-Zag: Color flip, bottom-up double rotation
                    ii. 2m-ii: Zig-Zag: Color flip, double rotation
        v. Insertion
            1. On the way down, perform colorFlip() if both children are red.
            2. Insert the new node as red
            3. Wherever two red edges are connected, do bottom-up rotations
        vi. Formal Definition
            1. A red-black tree is a binary search tree where each node is marked either red or black and no two red edges appear consecutively.
            2. A *balanced* red-black tree has the same number of black nodes on all paths from the root to any leaf.
    f. B-Trees
        i. Last balanced binary search tree structure we'll study

ii. Randomized: Effect is the same as if keys were inserted in random order
iii. Introduction
    1. Optimization approach
    2. Considered an extension of the 2-3-4 tree
    3. The difference between this tree and the others: B-Trees are disk-resident, whereas others are main memory resident)
        a. One disk page access takes the same time as hundreds of thousands of machine instruction executions.
        b. Thus, for this tree disk I/O cost should be used as the performance metric (measured by number of pages accessed). CPU time just doesn't matter by comparison.
iv. Concepts
    1. Multi-way balanced tree for external searching
    2. This is a disk-resident version of the 2-3-4 tree.
    3. One B-Tree node represents hundreds to thousands of bytes (512, 1024, …)
v. Node Structure
    1. Non-Leaf Node
        a. p_i, $1 \le i \le m$ is a pointer.
        b. k_i, $1 \le i \le m$ is a key.
        c. M-ary node: $p_1k_1p_2k_2 \ldots p_{m-1}k_{m-1}p_m$
        d. Just like the 2-3-4 tree, but there could be many more than 4 elements.
    2. Leaf Node
        a. Stores from L/2 to L items, where L is determined by the record size and disk page size.
        b. Example: Page = 8,000 bytes, Record = 80 bytes, L = 100.
    3. Internal Node (except root)
        a. Stores ceil(M/2) to M pointers, (M-1)/2 to M-1 keys.
        b. L and M need not be the same!
        c. Could take ceiling or floor to convert to an integer. Since M is usually very large, it doesn't really matter which.
        d. Page Size B = 8kB, Pointer Size P = 4B, Key K = 16B
            i. Then one entry = 20B
            ii. Number of entries in each node = floor((B – P) / (P + K)) = 400 = M
        e. In practice, leaf nodes have the same pkpk..p structure as internal nodes, but the pointer is to an actual record. The ith value is the smallest in the (i + 1)th leaf.
    4. Root Node: From 1 to M-1 keys, 2 to M pointers.
vi. Insertion
    1. (Assume duplicates are ignored)
    2. Find the leaf node, insert the new key.
    3. If the leaf overflows, …
        a. Split it.
        b. Get a new node (acquire another disk page).
        c. Move half the entries to the new node.
vii. Deletion
    1. This is more complicated than insertion.
    2. Find and remove the key.
    3. If the node is less than half full, …
        a. Try to get keys from siblings.
        b. If neither sibling can afford to give away keys, merge them together. It can't overflow, or one would've been able to give up some keys.

c. Remove the smallest key of the larger sibling from the parent.
d. If that results in less than half in the parent node, propagate begging-or-merging upward.

viii. Run-Time

1. Again, this is measured in terms of disk accesses (in pages)
2. The number of pages accessed is the number of nodes accessed.
3. Because the root and perhaps the next echelon are cached, we must approximate the tree height.
4. In a binary search tree the worst case is when there's only one child per node. The best case is where there's two children.
5. For a B-Tree the worst case is $\log_{M/2}(N)$, best = $\log_M(N)$.
6. The more empty a tree is, the worse its performance.
7. Average case = $\log_{(2/3)M}(MN)$ based on empirical data – insert/delete a large number of nodes and see what happens.