Trees

Ming-Hwa Wang, Ph.D.
COEN 279/AMTH 377 Design and Analysis of Algorithms
Department of Computer Engineering
Santa Clara University

Terminology
- node, degree, root, tree, forest, subtree, path, length, height, (grand) children, (grand) parent, (proper) ancestor, (proper) descendant, nonterminal, terminal/leaf, ordered/unordered tree, sibling, perfect k-ary tree, complete k-ary tree

Listing/Traversal of Tree
- preorder, inorder, postorder, level-order
- reconstruct the tree from inorder with any one else

Labeled tree and Expression Tree
- prefix, infix, postfix/Polish, and conversions

Basic Operation
- parent(n, T)
- leftmost_child(n, T)
- right_sibling(n, T)
- label(n, T)
- create(v, T1, T2, .., Ti)
- root(T)
- makenull(T)

Implementation
- array of parent pointers
- list of children
- leftmost-child, right-sibling

Binary Trees and Binary Search Trees
- Basic Operation
  - find_min(T)
  - find_max(T)
  - insert(v, T)
  - delete(v, T)
    - if v is a leaf, just delete it
    - if v has only one child, parent adjust pointer bypass v and point to v's child
  - if v has 2 children, replace the key of v with the smallest key of the right subtree and recursively delete that node as previous case
- average depth O(lgN)
- performance depends heavily on the input being random
- tree tend to skew after lots of insertions/deletions

AVL Trees - balanced binary trees
- Adelson-Velskii and Landis tree
  - same height \( \rightarrow \) perfectly balance tree \( w / 2^{h+1}-1 \) nodes
  - differ by one \( \rightarrow \) min : \( S(h) = S(h-1) + S(h-2) + 1 \)
- Single rotation
  - insertion occurs on the outside (left-left / right-right)
    - rotate right for left-left insertion
    - rotate left for right-right insertion
    - update heights after rotation
- Double rotation
  - insertion occurs on the inside (left-right / right-left)
    - rotate left for its left child and then rotate right for left-right insertion
    - rotate right for it right child and then rotate left for right-left insertion
- Insertion needs at most one single/double rotation to balance the tree, but deletion may need more.

Splay Trees
- self-adjusting, amortized complexity \( O(m * \lg N) \)
- access locality
- no need to maintain height information and simpler
- splaying
  - if v's parent is the root, rotate v and the root
  - if zig-zag, perform a double rotation
  - if zig-zig, \( G-P-X \rightarrow X-P-G \)
- delete
  - access the node to be deleted (puts it to the root)
  - access the largest element in left subtree (no right child)
  - assign the right subtree as the right child

Red Black Trees
- Coloring properties:
  - Every node is colored either red or black.
  - The root is black, every leaf (include nil) is black.
  - If a node is red, its children must be black.
• Every simple path from a node to a nil pointer must contain the same number of black nodes.
• Alternatively, assign black/red to the pointers point to a black/red child.
• The height of a red black tree is at most $2\lg(N+1)$.
• No path is more than twice as long as any other, and guarantee that operations take $O(\lg N)$.
• Basic operations: color changes and rotations.
• Bottom-up insertion:
  • New item $X$ is placed as a red leaf in the tree (except insert a node as black to an empty tree.)
  • If the parent $P$ of the inserted item is black, done.
  • If the parent is red and the parent's sibling $U$ is black (the grandparent $G$ must be black),
    • LLb: if $X$ is the left child of $P$ and $P$ is the left child of $G$, a single right rotation with color change. Finally the subtree's new root is black.
    • LRb: if $X$ is the right child of $P$ and $P$ is the left child of $G$, a left rotation followed by a right rotation with color change. Finally the subtree's new root is black.
    • RRb/RLb will be symmetric to LLb/LRb.
  • If both $P$ and $U$ are red (LLr, LRr, RLRr, RRr), change both color to black, and change the grandparent $G$ to red. If the grand-grandparent is also red, percolate this procedure up until we no longer have two consecutive red nodes or we reach the root (which is changed to black).
  • an example:
    • the original tree:
      11b
      / \
     2r 14b
     / \ / \ \
    1b 7b 15r
      / \ / \
     5r 8r
    • insert 4:
      11b
      / \
     2r 14b
     / \ / \ \
    1b 7b 15r
      / \ / \
     5r 8r ← same color
      \\
• Top-down deletion:
  • Deletion of a red leaf $Y$ is trivial.
  • If the parent of $Y$ is $P$, the sibling ($S$) of $Y$ is black, and $S$ has 0 (Rb0, Lb0), 1 (Rb1L, Rb1R, Lb1L, Lb1R), or 2 (Rb2, Lb2) red children:
    • Rb0/Lb0: change $S$ to red and $P$ to black
    • Rb1L/Lb1R: rotate right, change $S$ to $P'$ color and change $P$ to black (preserve root's color)
    • Rb1R/Lb1L: let the red child of $S$ be the new root (i.e., do a left and then a right rotation) of the subtree and preserve the root's color, change $S$ and $P$ to black
    • Rb2/Lb2: same as Rb1R/Lb1L (preserve)
  • If $S$ is red (then $P$ is black), and $S$ has 0, 1 or 2 red descendants:
    • Rr0/Lr0: right rotation to make $S$ the root of the subtree and change its color to black
- **Rr1RL/Lr1LR**: make the red descendant’s parent the root of the subtree
- **Rr1RR/Lr1LL**: make the red descendant the root of the subtree and change it color to black
- **Rr2/Lr2**: same as Rr1RR/Lr1LL.

The average red black tree is about as deep as an average AVL tree, but less overhead for insertion and rotations occur relatively infrequently.

### B-Trees

- **Structure properties:**
  - The root is either a leaf or has between 2 and $M$ children
  - All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and $M$ children
  - All leaves are at the same depth
  - All data is stored at the leaves. Each leaf has between $\lceil M/2 \rceil$ and $M$ keys
  - Interior node contains $M$ pointers $P_i$ (i in $[1, M]$) and $M - 1$ values $K_j$ (j in $[1, M-1]$), $K_j$ represent the smallest key found in the subtree $P_{i+1}$.

- **Order of B-tree**
  - order 3: 2-3 tree

- **Insert**
  - if not overloaded, just insert
  - if overloaded,
    - split the node into two, or
    - insert in a unoverloaded sibling
  - adjust the entries in the internal nodes

- **Delete**
  - if not underloaded, just delete
  - if underloaded,
    - combine with a sibling, or
    - steal from sibling
  - adjust the entries in the internal nodes

- **Maximum depth of a B-tree is $\lceil \lg \lceil M/2 \rceil \rceil$**

### AA-Trees

**Treaps, Cartesian Tree, or Priority Search Trees**

- A treap is simultaneously a binary search tree for the search keys and a heap for the priorities, and the node with highest priority must be the root. A treap could be formed by inserting the nodes highest-priority-first into a binary search tree without doing any rebalancing.
- If the priorities are independent random numbers then the shape of a treap has the same probability distribution as the shape of a random binary search tree, a search tree formed by inserting the nodes without rebalancing in a randomly chosen insertion order. Because random binary search trees are known to have logarithmic height with high probability, the same is true for treaps.
- The root corresponds to a $T$ whose joint lies on the topmost point. The $T$ splits the plane into three parts. The top part is (by definition) empty; the left and right parts are split recursively. This interpretation has some interesting applications in computational geometry. Treaps were rediscovered and used to build randomized search trees.