COMP 2210 – Exam 3 Study Guide Robert Sanek

**RECURSION**

**Recursive Definitions consist of:**

if (base case) return solution;

else return recursive\_call;

1. Base Case – simple statement/definition not involving recursion
2. Recursive Case – set of rules that reduce all other cases toward base case

* When a method is called, an *activation record* (stack frame) for that method is **pushed** onto the *runtime stack* (call stack). When a method returns, its activation is **popped** from the call stack.
* One of the benefits of recursion is that it **focuses our thinking very clearly on specific cases of the problem**. Our thinking, and the resulting code structure, is based directly on the recursive structure of the object we’re dealing with.
* **Recursive code is generally less efficient than an equivalent iterative version.**
* Generally: If code is “naturally recursive,” code recursively. If this becomes a bottleneck, implement iterative version.

**Tail Recursion** – a special case of recursion where the last operation of the method is the recursive call.

public int factTR(int n, int fact) {

if (n==0) return fact;

else return factTR(n-1, n\*fact); }

public int factorial (int n) { factTR(n, 1); }

public int factorial (int n) {

if (n==0) return 1;

else return n\* factorial(n-1); }

* To write a method in tail recursive form, add parameters as necessary so that the computation is performed on the “up” calls via the parameters instead of performing it on the returns.
* Compilers can eliminate tail recursion automatically, and this is a common optimization (ex. GCC in C programs).
* **Writing a method in tail recursive form can speed things up independently of what the compiler does/doesn’t do** (ex. Fibonacci, improvement from O(2N) to O(N))

public int fib(int n) {

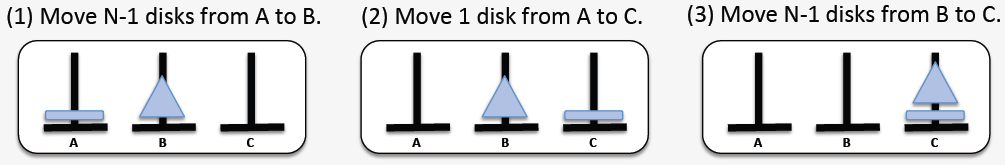
if (n==0) return 0;

else return fibTR(n, 1, 1, 0); }

public int fibTR(int n, int k, int fibk, int fibk1) {

if (n==k) return fibk;

else return fibTR(n, k+1, (fibk + fibk1), fibk); }

**Tower of Hanoi**

public void moveTower(int numDisks, String startPeg, String endPeg, String tempPeg) {

if (numDisks == 1)

moveOneDisk(startPeg, endPeg);

else {

moveTower(numDisks-1, startPeg, tempPeg, endPeg);

moveOneDisk(startPeg, endPeg);

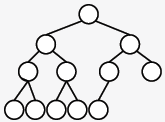
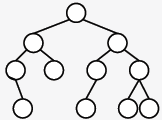
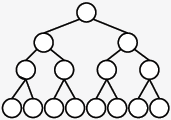
moveTower(numDisks-1, tempPeg, endPeg, startPeg);

}

}

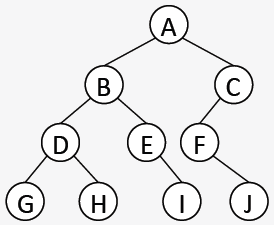
**TREES**

**Tree** – a collection in which elements are arranged in a hierarchy.

* **Lists vs. Trees**:
  + A list is a **one-dimensional** structure: it defines *linear* relationships between elements (predecessor, successor)
  + A tree is a **two-dimensional** structure: it defines *hierarchical* relationships between elements (parent, child).
* Composed of nodes and branches (edges).
* Can be implemented with arrays or nodes and pointers
* **Node Types**: **parent** (1+ children) **leaf** (no children), **child** (1 parent), **root** (no parent).
* **Order** – an integer ≥ 2 that represents the upper limit on the number of children that any node can have.
  + Ex. **binary** (≤2 children), **ternary** (≤3 children), **general** (no specified order/number of children)
* **Path** – sequence of nodes from one node to another, parent to child. **Path length** = number of nodes (or edges) on the path.
  + Node X is an **ancestor/descendant** of node Y iff there is a path from X/Y to Y/X, respectively.
* **Subtrees** – trees within larger trees. There are as many subtrees as there are nodes in the tree (tree itself is a subtree).
* **Height** – measures distance of given node from the “bottom” of the tree. It is the length of the longest path from a given node to a descendent leaf.
  + O(log n) for full/complete/balanced trees
* **Depth** – measures distance of given node from the “top” of the tree. It is the length of the path from the root of the tree to a given node (same concept as “level”).
  + Depth of lowest leaf = height of tree
* Tree terminology
  + **Full** – all leaves have same depth, every parent node has maximum number of children.
    - Height: floor(log2(n)) + 1
  + **Complete** – tree is full to the next-to-last level; all leaves on lowest level are “left-justified.”
    - Shortest possible tree (minimum height) that can store N nodes.
  + **Balanced** – each node’s subtrees have similar heights.
    - Near-optimal height for storing N nodes. Height: O(log N)

**BINARY TREES**

**Binary tree** – a tree of order 2

* Implementation Strategies
  + **Node-and-link based** (matches conceptual picture of a tree)
    - store the element and pointers to right and left trees
  + **Array-based** (can use too much space)
    - store root at index 0; nodes stored at index i:
      * **left** child: 2i + 1 | **right** child: 2i + 2 | **parent**: floor((i-1)/2)
* Common algorithms can be implemented recursively – calculating height, calculating number of nodes, searching for a value, traversing. Generally, these are implemented by doing something at the current node, then handling the left/right subtrees recursively.
* Transversal Types
  + Depth-first:
    - **Preorder**: NLR (A B D G H E I C F J)
    - **Postorder**: LRN (G H D I E B J F C A)
    - **Inorder**: LNR (G D H B E I A F J C)
  + Breadth-first:
    - **Level order:** (A B C D E F G H I J)
      * Visits nodes level by level: can be implemented with a FIFO queue

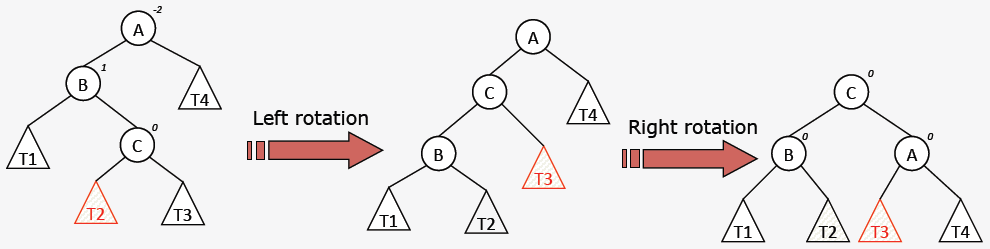
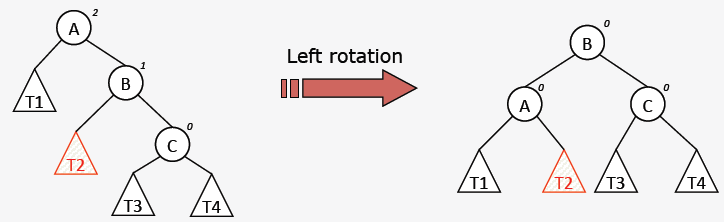
**BINARY SEARCH TREES**

**Binary search tree** – a tree in which the search property holds on *every* node

* **Search property** – every left child of a node must be less than the parent; every right child must be greater than the parent.
* A binary search tree imposes a **total order** on all its elements. Inorder transversals will necessarily list elements smallest to largest.
* **Searching for values** – the number of comparisons to find a given value is equal to the depth of the node that contains it.
  + Worst-case: searching for a lowest-leaf value. Entire height of tree is traversed.
    - tall and narrow trees: O(N); short and wide trees: O(log N)
* **Inserting values**
  + A new node will always be a new leaf. Use search algorithm to locate insertion point.
  + Worst-case is O(height) (@ lowest leaf) | Tree stays relatively flat when values are added in random order.
* **Deleting values**
  + Use search algorithm to locate value to be deleted. Worst-case is O(height) (@ lowest leaf)
  + Three cases for deletion. If node to be deleted is a:
    - **Case 0** Leaf node: set the parent’s pointer to this node to null.
    - **Case 1** Node with one non-empty subtree: set the parent’s pointer to this node to this node’s child.
    - **Case 2** Node with two non-empty subtrees: find a replacement node, delete the node containing selected replacement.
      * Replacement value can be inorder predecessor or successor
  + Values deleted in random order cause the tree to be less well-structured.

**AVL TREES**

**AVL Tree** – a binary search tree in which the heights of the left and right subtrees of *every* node differ by at most 1

* **Balance Factors**
  + Every node in an AVL tree has a balance factor.
  + ***bfN = hR - hL***
  + Balance factor will be positive if right subtree is greater in height; negative if left subtree is greater in height.
  + A bf of ±2 means that the subtree rooted at that node is out of balance.
    - Balance can be restored with rotations. All rotations occur in the context of a **3-node neighborhood**.
* Rebalancing Operations
* Coding Rotations

B = rotateLeft(A);

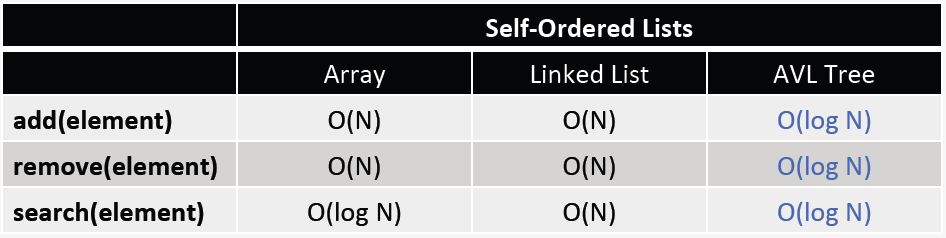
public BTN rotateLeft(BTN n) {

BTN m = n.right;

n.right = m.left;

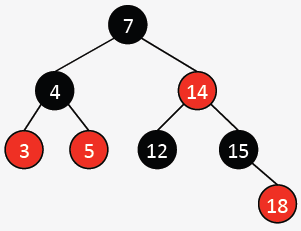
m.left = n;

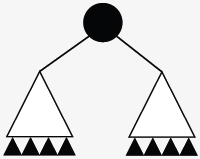
return m; }

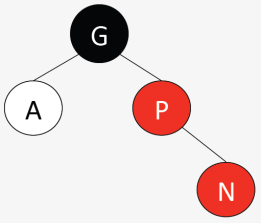
* **Inserting** new elements – use the standard BST insertion algorithm to insert new nodes. Starting with this node, walk the reverse path back towards the root, recalculating balance factors.
  + Stop at the first (lowest) node that has a bf of ±2. This node roots the 3-node neighborhood that will be rotated.
  + **At most one rebalancing operation is required per insertion**.
* **Deleting** elements – use standard BST deletion algorithm to delete the element. Starting at the *point of deletion*, walk the reverse path back towards the root, recalculating balance factors.
  + Stop at the first (lowest) node that has a bf of ±2. This node roots the 3-node neighborhood that will be rotated.
  + **Multiple rebalancing operations may be required per deletion**, so the reverse walk must go to the root each time.
* AVL Trees offer guaranteed **O(log N)** performance on all 3 major collection operations: **add, remove, & search**.

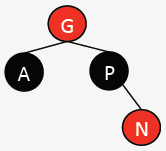
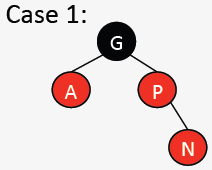
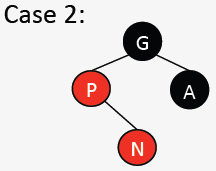
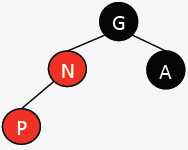
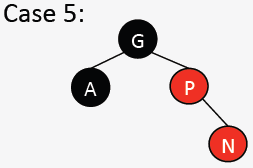
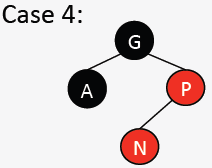
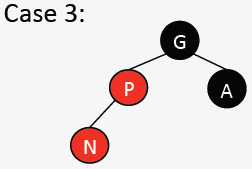
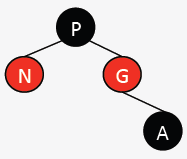
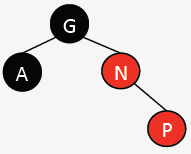
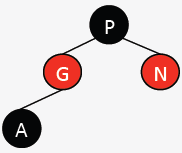
**RED-BLACK TREES**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Max Height** | **Insertion** | **If unbalanced** | **Repair** |
| **AVL** | ~1.44log2N | Insert according to value, then check balance with reverse walk   * bf = ±2? | Identify a 3-node neighborhood | Rotations   * ≤1 repair per insertion * Many repairs possible for deletion |
| **R-B** | 2log2(N+1) | Insert according to value, then check balance with a reverse walk.   * red-red? | Identify a 4-node neighborhood | Rotations + Re-colorings   * Many repairs possible for both insertion and deletion. |

**Red-black tree** – a binary search tree with the following node color rules:

1. **Each node is either red or black.**
   1. Defines legal nodes: black and red.
2. **The root and all empty trees are black.**
   1. Defines “boundaries” of red-black trees
3. **All paths from the root to an empty tree contain the same number of black nodes.**
   1. First half of balance requirement. It makes a statement about the height of the tree in terms of black nodes. Often called the **black height**.
   2. Constrains black node usage. Without red nodes, red-black trees could only be **full**.
4. **A red node can’t have a red child.** 
   1. A red node is used like “filler.” It allows the red-black tree to obey rules 1-3 without being full.
   2. Constrains red node usage.

* Tensions between Rules 3 and 4 force rotations and re-colorings, keeping the tree balanced.
* **Inserting** elements
  + Use standard BST insertion algorithm to insert new node. **Color it red.**
  + Beginning with the red node just inserted, walk the reverse path back toward the root, looking for violations of Rule 4 (red-red).
  + Stop at the first (lowest) red node that has a red parent. This node’s grandparent roots the 4-node neighborhood that will be repaired.
* **The 4-node neighborhood**
  + Bottom node (**N**) is first node with a red parent (**P**). Grandparent (**G**) of N is the root of the neighborhood, and it is black. The ancle (**A**) of N is the fourth node.
  + The repair needed is determined first by A’s color and second by the structural configuration of these four nodes.
* **5 cases for repair**:

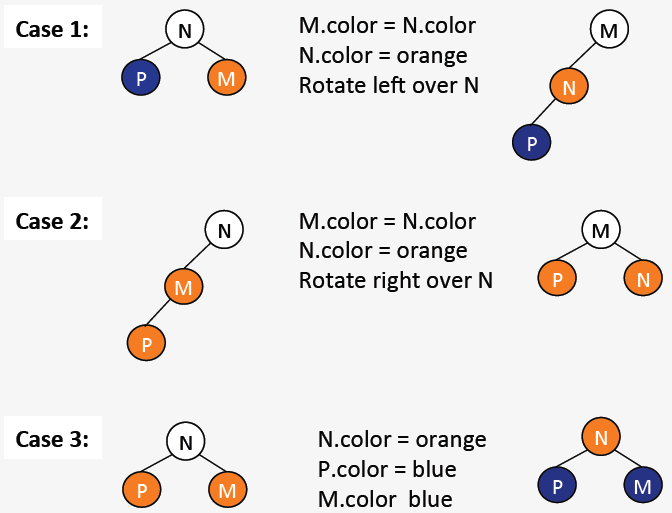
1. A is red. Re-coloring.
   1. Color flip P, color flip A, color flip G
2. A is black.
   1. Rotate left around P, GOTO Case 3
3. A is black.
   1. Color flip P, color flip G, rotate right around G
4. A is black.
   1. Rotate right around P, GOTO Case 5
5. A is black.
   1. Color flip P, color flip G, rotate left around G

* Red-Black trees offer guaranteed **O(log N)** performance on all three major collection operations: **add, remove, & search**.

**LEFT-LEANING RED-BLACK/WAR EAGLE TREES**

**War Eagle trees** are left-leaning red-black trees. They have the same rules as red-black trees, plus:

1. **Blue** corresponds to **black**, **orange** corresponds to **red**
2. An orange node can only be a left child.

* This fifth rule greatly simplifies the add and remove algorithms by reducing the number of cases to three, with **3-node neighborhoods**. Cases must be checked in this order:

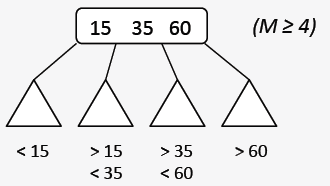
Change N to orange, change P & M to blue.

Swap colors of M & N, rotate right over N.

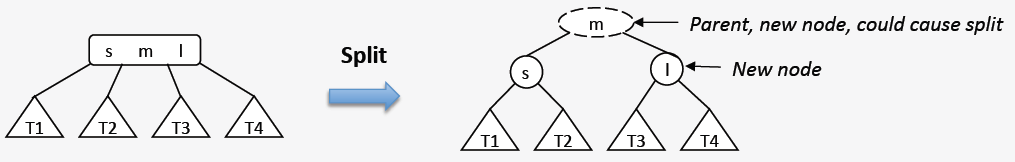
Swap colors of M & N, rotate left over N.

**MULTI-WAY SEARCH & 2-4 TREES**

**Multi-way search tree** (an **M-way tree**) – a tree of order M ≥ 2 in which the search property (total order) holds on every node and in which all leaves are at the same depth.

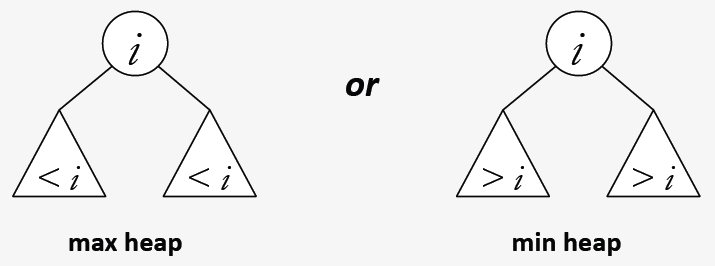
* M-way tree properties:
  + Each node holds between 1 and M-1 values in sorted order.
  + A non-leaf node with K values has K+1 non-empty subtrees that are M-way search trees.
  + The ith subtree of a node that holds values [v0…vk] (0 ≤ i ≤ K) can only store values v such that vi-1 < v < vi

**2-4 tree** – a 4-way search tree where each non-leaf node must have at least two non-empty subtrees.

* **Inserting** values
  + To add a new value, use the total order to find the leaf that should hold this value. **New values are always added in the context of an existing leaf node** (never a new leaf!)
  + Nodes in 2-4 trees can store at most 3 values. 2-4 trees grow “up” by adding a new root rather than down by adding a new (lower) leaf. When a 2-4 node is full but needs to store another value, perform a **split**:
  + Worst-case add in 2-4 trees causes the root to change.

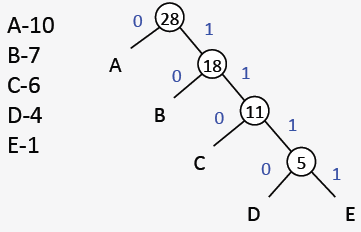
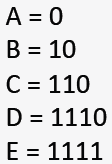
**BINARY HEAPS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PQ Method | Unsorted List | Sorted List | Balanced BST | Binary Heap |
| Add | O(1) | O(N) | O(log N) | O(log N) |
| Remove | O(N) | O(1) | O(log N) | O(log N) |
| Peek | O(N) | O(1) | O(log N) | O(1) |
|  | Nodes/Arrays | Nodes/Arrays | AVL, R-B, etc. | Nodes/**Arrays** |

**Binary heap** – a complete binary tree in which each node obeys a partial order property

* Binary heaps are almost always **implemented as arrays** because:
  + Acceptable space efficiency (complete shape)
  + Easy transversal (parent to child via multiplication, child to parent via division). Store values as in BST:
    - store root at index 0; nodes stored at index i:
      * **left** child: 2i + 1 | **right** child: 2i + 2 | **parent**: floor((i-1)/2)
* Binary Heap **Insertion**
  1. Insert new element in the only location that will maintain the complete shape.
  2. Swap values as necessary on leaf-to-root path to maintain partial order.
* Binary Heap **Deletion**
  1. Maintain the complete shape by replacing the root value with the value in the lowest, right-most leaf. Then delete that leaf.
  2. Swap values as necessary on root-to-leaf path to maintain partial order.
* Heapsort – an in-place comparison sort with O(N log N) time complexity
  + Important because N log N is lower bound (optimal) on number of comparisons necessary for comparison sorts.
  + 2 Phases of heapsort:
    - Rearrange the array elements into max heap order: beginning with the lowest, right-most parent and continuing to the root, heapify each subtree.
    - Repeatedly move the maximum element to its final stored place toward the end of the array, and heapify the remaining elements.
* Heapsort is an in-place sort with guaranteed N log N worst-case performance, but it is not stable and typically has larger constant factors than quicksort.

**Huffman’s algorithm** – generates a variable-length encoding for a given alphabet for the purposes of data compression.

* ASCII – binary character encoding scheme (a sequence of 0s and 1s (bits) used to encode characters) that includes English alphabet, punctuation, digits, and “control” characters.
* ASCII is a **fixed length code**. Each character is represented by the same number of bits. 8 bits = 1 parity bit + 7 bits to encode character (27 = 128 different characters)
* **Variable length codes** – number of bits per character determined by the char’s relative frequency of occurrence. Most frequently occurring chars should use the fewest bits.
  + Generating a vlc: The code for one char can’t be a prefix of another char’s code.
* Code trees – binary trees in which the leaves contain the characters to be coded. Interior nodes are just place-holders; the root of every subtree is annotated with the cumulative frequency of all its descendent leaves.
* Character codes are generated by root to leaf traversals.
* L\left(C\right) = \sum_{i=1}^{n}{w_{i}\times\mathrm{length}\left(c_{i}\right)}There are many possible code trees and many possible char codes. Since char codes are defined by the root to leaf paths, the tree’s shape determines “average” code length.

(left branch = 0, right branch = 1)

wi = weight(ai)

L(C) = “average” code length.

* Huffman’s algorithm generates a code tree with an average code length that is as least as small as any other code tree that could be generated.
  + Create a single node code tree for each character
  + Insert each of these trees into a priority queue (min heap).

while (pq has more than one element) {

c1 = pq.deletemin();

c2 = pq.deletemin();

c3 = new codetree(c1, c2);

pq.add(c3); }