ECS 60 **Trees Lectures**

I. Trees
- A. Motivation O(N) for lists, average is O(log n)
- B. Definitions
  - 1. Tree = a finite set of one or more nodes such that:
    - a) There is a specially designated node called the root, r.
    - b) The remaining nodes are partitioned into n ≥ 0 disjoint sets T1, …, Tn, where each of these sets is a tree. We call T1, …, Tn the subtrees of the root. The roots of each of these subtrees are connected by a directed edge from r.
    - c) The root of each subtree is said to be a child of r, and r is the parent of each subtree root. Each node has exactly one parent; the root has no parent.
    - d) More family terminology
      - (1) Siblings = Nodes with the same parent.
      - (2) Ancestor, descendent,
    - e) Leaves = nodes with no children
  - 2. A path from n1 to nk (down) is a sequence of nodes n1, n2, …, nk such that n_i is the parent of n_{i+1} for 1 <= i < k. Length = number of edges in a path , namely k-1.
  - 3. Depth of n_i = length of path from the root to n_i . depth of root = 0.
  - 4. Height of n_i is the length of the longest path from n_i to a leaf.
- C. Visual representation using pet tree vs course requirement "tree"
- D. Linked List implementation

```
struct TreeNode {
  Object element;
  TreeNode *firstChild;
  TreeNode *nextSibling;
};
```

II. Binary Trees = a tree in which no node can have more than two children.
- A. Definitions
  - 1. Complete binary tree = every level except deepest must contain as many nodes as possible, and at the deepest level , all the nodes are as far left as possible.
  - 2. Full binary tree = every leaf has the sam depth and every non-leaf as two children.
  - 3. Complete binary tree
- B. Linked List implementation
- C. Array implementation Left child = [2i + 1], Right child = 2i + 2

III. Binary Search Trees
- A. Definition: A binary tree with the property that for every node, X, in the tree, the values of all the items in its left subtree are smaller than the item in X, and the values of all the items in its right subtree are larger than the item in X
- B. ADT
  - 1. Object find(Object)
  - 2. Object findMin()
  - 3. Object findMax
  - 4. void insert(Object x), O(N) 6, 15, 3, 20, 1, 8, 5, 16,18.
  - 5. void remove(Object x), O(N)
    - a) Leaf node, just delete. Delete 1
    - b) parent with one child, link child to its grandparent, and delete the node Delete 5
    - c) parent with two children, replace the data of the node with smallest data of the right subtree and then recursively remove that child node. Delete 15
      - (1) Tends to skew tree, making left subtrees larger.
  - 6. makeEmpty()
  - 7. All operations except makeEmpty() have average O(log N) though there are cases of O(N^2), e.g. insert in order 1,2,3,4

IV. AVL Trees = BST with a balance condition.
- A. For every node in the tree, the height of the left and right subtrees can differ by at most 1. So depth is O(log N)
- B. Insertion. 4, 8, 12, 16,

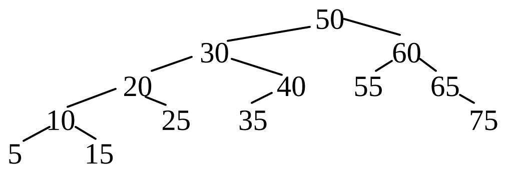
1. After an insertion only the nodes on the path from the insertion point to the root might have their balance altered. Start at lowest altered subtree and work up to root.
2. Four cases:
    a) Left subtree of the left child of r. (outside)
    b) Right subtree of the left child of r.(inside)
    c) Left subtree of the right child of r.(inside)
    d) Right subtree of the right child of r. (outside)
3. Single Rotation for outside cases.
    a) For 2a above:

```
rotateWith LeftChild( AvlNode * & k2){ // k2 is parent of r.
  AvlNode *k1 = k2->left;
  k2->left = k1->right;
  k1->right =k2;
  k2 = k1;
}
```

4. Double Rotatation for inside cases.
    a) For 2b above.
        (1) Single Rotate r with its parent.
        (2) Single Rotate r with its new parent

C. Deletion

1. Two steps
    a) Delete node as with binary search tree. = Leaf just do it; with one child place child in its place, two children then pick min of right.
    b) Retrace path from deleted parent to root balancing using single and double rotation as needed.
    c) Note, it is possible to stop traversing the entire path if the height info



Delete 55 to get two rotations.

V. Splay Trees

A. M operations take at most O(M logN) time, though one operation may take O(N) time.
B. Allowing O(N) but want O(M logN) then must move nodes.
C. No height or balance info needed.
D. Zig-zag case = Double rotation
E. Zig - zig case = move up two and set grandparent as first child.
F. When access paths are long, the rotations tend to be good for future operations.
G. When accesses are cheap the rotations are not as good and can be bad.
H. Deletion by accessing the node, deleting it. Rotate largest element of left tree to its root so root has no right child, and attach right subtree.

VI. Tree Traversals.

A. Inorder O(N) to list all the items in sorted order.
B. Postorder for determining height of a node.
C. Preorder to determine the depth of a child.
D. First case in recursive is null case.
E. Level-order, breadth-first search cannot be done recursively, and is done with a queue.

VII. B Trees

A. Motivation: Big-Oh ignores disk access time. We want to reduce the number of disk accesses to a very small constant. Complicated code is irrelevant in comparison to disk access time. More branching means less height. A balanced binary tree has height $\log_2 N$, while a M-ary search tree has height that is roughly $\log_M N$

B. B+ tree has the following properties.

1. The data items are stored at leaves. Not true for B* trees and B trees, which have data, stored in non-leaf nodes too.

2. The non leaf nodes store up to M - 1 keys to guide the searching; key i represents the smallest key in subtree i+1.
    a) An M tree is said to have degree M.
3. The root is either a leaf or has between two and M children.
4. All nonleaf nodes (except the root) have between ceil(M/2) and M children.
5. All leaves are at the same depth and have between ceil(L/2) and L children, for some L. L is determined based on the size of data vs disk block size. L = floor (disk block size / sizeof(object))

C. Insertion--Always follow tree to leaf for insertion.
1. If room in leaf then add data. Done.
2. If not room in leaf then if next leaf has room then change key in parent and move to next leaf. Called adopting
3. If not room in leaf or adjacent leaf, then create new leaf by splitting leaf, and adding key to parent. If parent is full then split parent recursively.

D. Deletion--Always follow tree to leaf and delete the data
1. If number of items in the leaf fall below minimum (ceil(L/2)) then
    a) Adopt an item from a neighbor if they are not at a minimum and adjust parent keys.
        (1) Sean rule: look to left neighbor first.
    b) If neighbor at minimum, then combine both, and remove a key from parent. If parent keys fall below minimum (ceil(M/2)) then it must parent follows the same procedure, i.e. adopt from else combine. This continues to be done recursively up the tree. If root left with one child then make child the root.

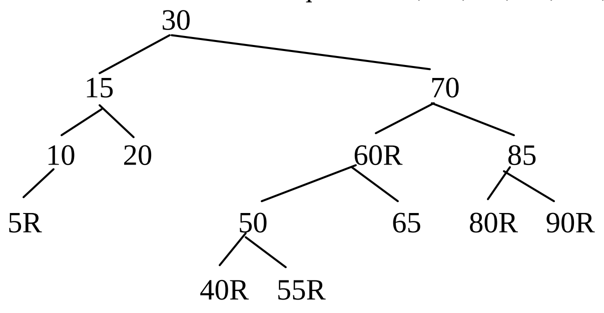
VIII. Red-Black Trees

A. a BST with four rules
1. Every node is colored either red or black.
2. The root is black
3. IF a node is red, its children must be black
4. Every path from a node to a NULL pointer must contain the same number of black nodes.

B. Consequence of coloring rules is the height of a red-black tree is a most 2 log(N + 1) so O(logN). Advantage of red-black trees over AVL is the relatively low overhead required to perform insertion, and the fact that in practice rotations occur relatively infrequently.

C. Insertion
1. Place in leaf. Must be red otherwise rule #4 is violated.
2. If parent is black, then we are done.
3. If parent is red, then assume NULL pointers are black and proceed with the following:
    a) If sibling of the parent is black then look at parent and grandparent
        (1) If zig-zig, then do single rotation of parent with grandparent, and make original parent black, and grandparent red.
        (2) If zig-zag, then do double rotation, and make inserted node black, and original grandparent red.
    b) If sibling of the parent is red and we do rotations, we may have percolate up. We can avoid the possibility of having two red children by altering the tree on the way down to the leaf.
        (1) On the way down to insertion position, if we see a node, X, that has two red children, then make the X red, and the two children black. If X's parent is red then, go to 3 a).
4. Top Down Deletion
    a) Simplifies to deleting a leaf. If node has two children or only a right child, then copy the minimum of right subtree to the position of the deleted node, and then delete the moved node. If node has only a left child, then copy the maximum of the left subtree.
    b) If deleted leaf node is red, then do nothing.
    c) If deleted leaf node is black, then we would have problems violating Rule #4. To avoid this, we can start at top and try to ensure that the leaf is red.
        (1) Make the root red.
        (2) Let P be the parent, X be the child in the path to the leaf, and T be the other child of P.
            (a) P will be red, and X and T will be black.
            (b) If X has two black children, and T has two black children then we can flip the colors of P, T, and X.
            (c) If X has two black children, and T has at least one red child, then treat P,T, and T's red child as zig-zig (single rotation), or zig-zag (double rotation).
            (d) If X has at least one red child, then just move down the path. If the new X is red we continue onward,. If the new X is black then the new T must be red, and X and new P (old X) will be black. We can then rotate T and P, making X's new parent(T) red, and its grand parent is black.

Insertion sequence 10, 85, 15, 70, 20 , 60, 30 , 50 , 65, 80, 90, 40, 5, 55.



I. Tries

A. A tree that uses parts of the key to navigate the search is called a trie, pronounced "try"

B. The leaves may contaoin only the unprocessed suffices of the words.

C. When making a comparison with in a binary search tree, the comparison is made between the key search for and the key in the current node, whereas in the trie only one character is used in each comparison except when comparing with a key in a leaf. Therefore, in situations where the speed of access is vital, such as in spell checkers, a trie is a very good choice.

D. Immune to order of entries.

E. Height determine by longest identical prefix in two words.

F. Space can be reduced by having each node contain a linked list of only those letters encountered, but this means the random access is replaced with sequential search of the node. This ends up being a child-sibling tree.

G. Huffman's algorithm

1. Create single node trees which contain count of each letter.
2. Merge the two trees with the smallest count into one tree.
3. While more than one tree go to 2.
4. Traversing to left is a zero, traversing to the right is a one.