ECS 20 **Chapter 10, Binary Trees**

1. Introduction
   1.1. Because of their property of dividing data into two, not necessarily equal parts, binary trees can be used to organize data in a variety ways.
2. Binary Trees
   2.1. Tree = a finite set of one or more nodes such that:
      2.1.1. There is a specially designated node called the root, *r*.
      2.1.2. The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, T_2, \ldots, T_n$, where each of these sets is a tree. We call $T_1, T_2, \ldots, T_n$ the subtrees of the root. The roots of each of these subtrees are connected by a directed edge from *r*.
      2.1.3. 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.
      2.1.4. More family terminology
         2.1.4.1. Siblings = Nodes with the same parent.
         2.1.4.2. Ancestor = any node on the path between the root and the specified node, including the root.
         2.1.4.3. Descendent = any node in the subtree that has the specified node as its root.
      2.1.5. Leaves = nodes with no children
      2.1.6. Internal node = a node with at least one child.
   2.2. A path from n_1 to n_k (down) is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 <= i < k$. "Path Length" = number of edges in a path , namely k-1.
   2.3. "Depth" of n_i = length of path from the root to n_i . (Called "level number" in the text.) Depth of root = 0.
   2.4. "Height" of n_i is the length of the longest path from n_i to a leaf.
   2.5. "Binary Tree" = a tree in which no node can have more than two children. The children are often assigned the labels *left* and *right*.
   2.6. A "*d*-ary" tree = a tree in which no node can have more than *d* children, e.g. 3-ary tree, or 5-ary tree.
3. Complete and Extended Binary Trees
   3.1. Complete Tree
      3.1.1. A tree is said to be "complete" if all levels, except possibly the deepest, have the maximum number of possible nodes, and if all the nodes at the deepest level appears as far left as possible.
      3.1.2. Height of a complete tree is $\log_d n$ for a d-ary tree, e.g. $\log_2 n$ for a complete binary tree.
      3.1.3. For a any *d*-ary tree, if stored in a zero-based array, *parent index* = $\lfloor \frac{child\ index}{d} \rfloor$, e.g. $\lfloor \frac{child\ index}{2} \rfloor$ for a complete binary tree. The children indices start at *parent index* * *d* + 1 and continue up to *parent index* * *d* + *d*, e.g. for a complete binary tree *parent index* * 2 + 1 to *parent index* * 2 + 2
   3.2. Extended Binary Trees (skipped)
4. Representing Binary Trees in Memory
   4.1. Linked representation with each tree node containing its data, and two pointers to other tree nodes.
   4.2. Sequential representation in an array can work well with complete binary trees, but is wasteful for non-complete trees.
5. Traversing Binary Trees
   5.1. Three standard traversals rely on a stack. The prefix of the names of the traversals indicates when the parent is processed.
      5.1.1. Preorder = parent, then left child, and then right child. Can be used to assign the depth of a node.
      5.1.2. Inorder = left child, then parent, and then right child. Can be used to access nodes in numerical order in a binary search tree (next section).
      5.1.3. Postorder = left child, then right child, and then parent. Can be used to assign the height of a node = 1 + max(heights of children).
   5.2. Level order traversal relies on a queue = parent, children of parent, grandchildren of parent…
6. Binary Search Trees = 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*.

6.1. Searching Algorithm for item *M*

1. Set current node to the root of tree.
2. If the current node is NULL, then exit and indicate the *M* was not found.
3. Compare *M* with the data, *X*, of the current node.
4. If $M = X$, then exit and indicate that *M* was found.
5. If $M < X$, then set the current node to the left child of *X*, and go to step 2.
6. If $M > X$, then set the current node to the right child of *X*, and go to step 2.

6.2. Inserting Algorithm for item *M* is the similar to the searching algorithm, except we check for NULL children BEFORE we set the current node to a child.

1. If the root is NULL, then create a new node with *M* as its data, set it as the root, and exit.
2. Set current node to the root of tree.
3. Compare *M* with the data, *X*, of the current node.
4. If $M < X$, then
   4.1 If the left child of the current node is NULL, create a new node with *M* as its data, and make the new node as the left child of the current node and exit.
   4.2 If the left child of the current node is not NULL, then set the current node to the left child of *X*, and go to step 3.
6. If $M \geq X$, then
   6.1 If the right child of the current node is NULL, create a new node with *M* as its data, and make the new node as the right child of the current node and exit.
   6.2 If the right child of the current node is not NULL, then set the current node to the right child of *X*, and go to step 3.

6.3. To delete an item, search for the item, and if it is found then remove a node.

6.3.1. Deleting has three cases based on the number of children of the node to be deleted:

6.3.1.1. Leaf node: just delete it.
6.3.1.2. Parent with one child, link child to its grandparent, and delete the node.
6.3.1.3. Parent with two children: replace the data of the node with smallest data of the right subtree and then recursively remove that child node. This tends to skew tree, making left subtrees larger. So alternate by taking the largest data of the left subtree, and removing that child node.

7. Priority Queues, Heaps

7.1. Need three operations: findMaximum, deleteMaximum, insert()

7.2. Could use an unsorted array: findMaximum and deleteMaximum O(n), insert O(1).

7.3. Could use a sorted array with the largest at the end of the array: findMaximum and deleteMaximum O(1), insert O(n).

7.4. Using a binary heap findMaximum I s O(1), insert and deleteMaximum are O(log n).

7.5. Structure complete binary tree. Height of the tree is log n.

7.6. Heap order for every node X the key in the parent of X is larger than or equal to the key in X with the exception of the root.

7.7. findMaximum, just return a copy of object at the root of the tree.

7.8. Insert uses percolating up algorithm.

1. Create a “hole” at the next location on the complete binary tree.
2. Compare the object inserted with the parent of the hole.
3. If the object is less than or equal to the parent, then put the object in the hole, and exit.
4. If the object is greater than the parent, then copy the parent into hole, and consider the hole to be where the parent was. Go to step 2.

7.9. Delete Min uses percolating down algorithm

1. Copy the item that was in the last position in the complete binary tree into a temporary variable, called temp.
2. Think of there being a “hole” at the root.
3. Compare temp with the two children of the hole.
4. If temp is the largest then copy temp into the hole, and exit.
5. If one of the children is the largest, then copy its value into the hole, and move the hole down to position previously held by the largest child. Go to 3.

7.10.

8. Path Lengths, Huffman's Algorithm
   8.1. Tries
      8.1.1. A tree that uses parts of the key to navigate the search is called a trie, pronounced "try"
      8.1.2. The leaves may contain only the unprocessed suffices of the words.
      8.1.3. 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.
      8.1.4. Immune to order of entries.
      8.1.5. Height determine by longest identical prefix in two words.
      8.1.6. 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.
   8.2. 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
9. General (ordered Rooted) Trees Revisited (skipped)