Topics for study for the CS 2420 Final

1. Binary-Search-Tree
   Left must be smaller than parent, right must be larger than parent
   a. Insert : if it is smaller it goes left, if it is larger it goes right (lg n)
   b. Find: should follow the same as insert, but have a return true or false (lg n)
   c. Traversals
      i. Pre-order traversal: root first, then left tree, then right tree
      ii. In-order traversal: left tree, then root, then right tree
      iii. Post-order traversal: left tree, right tree, then root

```
cout << n->data << " ";
pre_order(n->left);
pre_order(n->right);
```

   d. Delete: (lg n)
      i. Case 1, the tree is empty, return NULL
      ii. Check if less than and call recursively to left ptr to find
      iii. If larger call recursively to right and find
      iv. When the data is found, check for number of childen
         1. If only 1 child, create pointer and delete node and reattach child
         2. If two childen, find predecessor, by going one left, then all the way to the right.
         3. Switch values of predecessor and N, delete n
   e. Duplicates
      i. Are they allowed?
      ii. If so, we have to find them and then figure out which one to manipulate
2. AVL-Tree, is also a Binary Search Tree
   - By definition this one always maintains balance.
   - Balanced Tree means that every nodes sub-trees has a height that differ by no more than 1
   - Height is equal to 1 plus the number of nodes in the sub tree
   - Insert / fetch / delete / split / join are all Big O (lg n)
   a. Structure
      i. Tree : is a bin-search tree that maintains balance
      ii. Nodes: each one also has a height variable, - Height is equal to 1 plus the number of nodes in the sub tree
   b. How to Rotate
      i. Rotation: if the unbalance is on the outside of the tree or sub-tree it only requires one rotation
      ii. Double Rotation: if the unbalance is on the inside of the tree or sub-tree it will require a two part rotation
   c. When to rotate
      i. We rotate when there is a node added and it creates an imbalance it height.
3. Heaps

   a. Structure: a heap is implemented as an array, and must be a complete tree, or a tree that is full from left to right, without sub- trees.
   b. Insert: if the array is not empty, then add the entry to the end of the array, do integer division by two, to find the parent, if the parent is less than the new entry, swap.   This is done in a for loop to ensure it swaps until the parent is not less than the new entry
   c. Remove: remove the front by swaping it with the back, and then bubble down by checking to see if it is a right side or left side of the tree.
   d. makeHeap: Takes an array and makes it into a heap, with the efficiency of Big O (N).
   e. heapsort: (n lg n), works this way because it takes an array and puts it into a heap, because is heapifying the numbers as it places the numbers they are sorted into the heap.
4. Graphs
   a. Graph Implementations
   b. BFS
   c. DFS
   d. Dykstra’s Algorithm
5. Hash-Tables
   a. Structure
   b. Insertion
   c. Delete
   d. Hash function
   e. Re-hashing
   f. Double hashing
   g. Collisions
   h. Chained Hashing
6. B-Trees
   a. Definitions
   b. Rules
   c. Implementation