# Quiz 3 Study Guide

Below are review questions for you to study for the upcoming quiz. Five major topics have been covered since Quiz 2: General and Binary Trees; Binary Search Trees; Balanced AVL Trees; Priority Queues and Binary Heaps; and Dictionaries and Hash Tables. You should review and understand the basic properties, performance, and implementation of each of these data structures.

The quiz will consist of 3-to-4 questions of the following variety: (1) Comprehension and Recall, where you will be asked to recall and answer knowledge questions; (2) C++ Programming, where you will be asked to program, in syntax correct C++ code, solutions to simple programming problems; and (3) Advancements of Knowledge, where you will be asked to advance your knowledge of a topic by applying that knowledge to a similar but related problem, for example, consider implementing a data structure not covered directly in class.

## 1 Tree Definitions and Terminology

- How is a tree different from a list? Are all trees lists or all lists trees? What makes a tree *non-linear?*
- Define the following terms: Node, Parent, Child, Interior Node, Root Node, and Leaf Node.
- What is the difference between a general tree and a binary tree?
- How many nodes can a binary tree of height $h$ contain? How many nodes can a general tree of height $h$ contain?
- Explain how the height of a tree and the depth of a given node of a tree are related.
- Write down the recursive definition of a binary tree: Explain how you can think of a binary tree as a collection of sub trees and empty trees?
- What are the four different traversal types for a binary tree? Using recursive functions where appropriate, write the pseudo code for each of the traversals?
- Explain why Level-Order traversal is just like Breadth-First-Search.
- Consider adapting the recursive methods of post-, pre-, and in-order traversal of a binary tree to a general tree. The class definition of a node in the general tree is as follows:

```
template <typename T>
class Node{
  T value;
  List<Node<T> *> children;
  //...
}
```

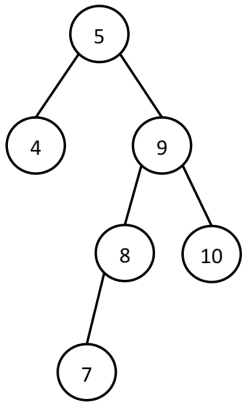
## 2 Binary Search Trees

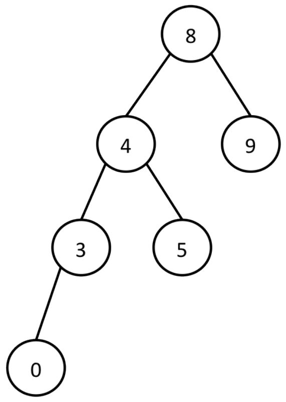
- Explain Binary Search over an array: Why does Binary Search requires the underlying list to be sorted?
- Analyze the performance of Binary Search on a sorted array. Why does it only require $lg(n)$ comparisons to either find (or not find) a given value in the array?
- Explain how a recursive trace of a Binary Search is very tree-like. Convert a trace of a binary search on a sorted array into a Binary Search Tree.
- A Binary Search Tree (BST) can function as a Dictionary: What are the abstract properties of a dictionary and how can a binary search tree implementation meet the requisite properties?
- What is the key property of a binary search tree with respect to the key-values in the left vs. right sub trees?
- In pseudo-code, write code for `find()` and `update()` of a binary search tree.
- Consider the implementation of BST in lab, write, in syntax-correct C++ code, code for `insert()`. Do not consider balancing the tree.
- When performing a remove on a BST, consider the four cases: removing a leaf; removing a node with a left child; removing a node with a right child; and removing a node with a left and right child. Write the pseudo-code for `remove()` for the three easy cases. What makes them easy?
- When performing a "hard case" remove, removing an interior node with a left and right child, what makes this case hard? How do you select the node to replaced the node that is being removed? Draw a visual to demonstrate where this node is in the tree, and how to find it.
- Analyze the runtime of each of the major BST operations for a balanced BST: `find()`, `update()`, `insert()`, `remove()`.
- Consider the *worst-case* runtime of each of the major BST operations: What is the worst case runtime? What does the worst case tree look like?
- Why is it important to maintain balanced binary search trees? Consider the insertion of the following items: $1, 2, 3, 4, 5, 6, 7$. What does the resulting tree look like?
- If you wanted to build a balanced BST using an implementation that does not provide balancing, in what order should the nodes be inserted? For example, does a post-order insertion order produce a balanced tree?

## 3 AVL Trees

- An AVL tree is a binary tree that maintains a balancing condition. What properties of the balancing condition are desirable with respect to the performance of an AVL tree?
- What is the standard AVL balancing condition with respect to the height of the sub trees of a given a node.

- Consider the implementation of an AVL, in what functions must balancing occur?
- Describe the four different types of imbalances. Draw a visual of each. Further classify the different imbalances into two categories: Why can you do this?
- For each imbalance type, describe the rotations that solves the imbalance. Draw a visual of the steps of each of the rotations.
- Draw the resulting AVL tree for inserting: $0, 1, 2, 3, 4, 5, 6, 7$.
- In the trees below identify the imbalance and draw the final balanced tree:



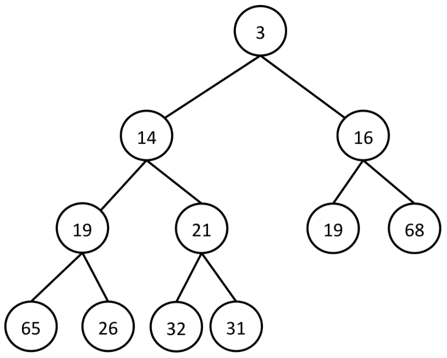


- Explain why we can implement `updateHeight()` non-recursively in our lab implementation of the AVL tree?
- When performing an insert on an AVL tree, how many nodes height value must be updated? Recall that the height value of a node is the height of the sub tree, of which this node is the root.
- When performing a remove on an AVL tree, if the root is removed, how many nodes height must be updated? If the remove occurs at the leaf, how many nodes height must be updated?
- Make an argument why it only requires at most one rotation to solve an imbalance after an insert? Why is this not the case for a remove?
- Demonstrate, that even with updating the height and all the rotations, that maintaining a balanced BST using AVL is still $O(lgn)$ for `insert()`, `remove()`, `update()`, and `find()`.

## 4 Priority Queues and Binary Heaps

- What makes a priority queue different than a general queue?
- Write down the abstract/pure-virtual definition of a priority queue?
- Consider implementing a priority queue using an unsorted array, a sorted array, a sorted linked list, and a binary search tree: What is the cost for each of the basic priority queue operations? (`insert()`, `removeMin()`, `getMin()`, `isEmpty()`, `getEmpty()`)

- What are the two key properties of a binary heap? Given these proprieties, explain how a binary heap can be used to implement a priority queue.
- Describe the necessity of the bubble-up and bubble-down procedure for maintaining heap order for a binary heap. In which basic operation is bubble-up called, and in which basic operation is bubble-down called?
- Draw the heap that results from inserting the following priorities into a heap: $5, 2, 9, 3, 5, 2, 7, 4, 8, 0, 6$. (*If you want to borrow playing cards to perform more exercises, you can borrow than from my office.*)
- Draw the result of performing five `removeMin()`'s from the following heap:



- Does there exist a traversal of a binary heap that will sort the items in the tree? If so, which one and why? If not, why not?
- Write the 1-based index array representation of the heap above. Given an index for a node in the array, how can the index of the parent of the node be found? How can the child of that node be found?
- Given a 1-based index array representation of a heap, write a recursive implementation of bubble-up. The type of the function is: `BinaryHeap::bubbleUp(int curIndex)`, where a BinaryHeap has an array of `items` in the heap and a `size` for the total number of items in the array.
- Write a similar recursive function for bubble-down.
- Consider using a heap to sort an unsorted array: Write a naive implementation that is not space efficient. Why is this not space efficient?
- What is the key to writing space efficient Heap Sort? How is the array placed in heap order, and where does the resulting removed items get placed?
- In general terms, explain why placing an array in heap order is $O(n)$? In general terms, explain why removing all items from a heap is $O(n * lg(n))$
- What is the benefit of in-place Heap Sort over in-place Quick Sort or non-in-place Merge Sort?

## 5 Dictionaries and Hash Tables

- Recall from earlier the costs of implementing a dictionary using an unsorted array, a sorted array, a sorted linked-list, and a balanced tree: where is there room for improvement and where will the cost always be $O(n)$
- What is a hash function and how can does it enable a hash table implementaiton of a dictionary to leverage fast indexing?
- What are the desirable properties of a hash function? Explain why the hash function below does not meet those requirements.

```
int hashCode(string key, int capacity){
    int sum = 0;
    for(int i=0;i<key.length();i++){
        //sum across ascii values
        sum+= (int) key[i]
    }
    //bound sum to the capacity
    return sum % capacity;
}
```

- Describe why the hash function below does meet those requirements:

```
int hashCode(string key, int capacity){
    int sum = 0;
    for(int i=0;i<key.length();i++){
        //sum across ascii values with multiplier
        sum = sum * 37 + (int) key[i]
    }
    //bound sum to the capacity
    return sum % capacity;
}
```

- Describe the process of *linear probing* for managing collisions in a hash table.
- In pseudo code, write code for `insert()` and `get()` for a linear probing hash table?
- What are the two solutions for handling removes in a linear probing hash table? Which is better, and why?
- Why are successful `get()`'s on a linear probing hash table more expensive than unsuccessful searches?
- How is the load on a hash table defined? How does the load affect the performance of a hash table?
- Given a high load, the hash table should be expanded and *rehashed*: What is that process? Write pseduo-code for rehashing a linearly probed hash table.
- Explain the process of *quadratic probing* for managing collisions in a hash table. What problem with linear probing does quadratic probing avoid?

- What is the process of *double hashing* for managing collisions in a hash table. What problems with linear probing and quadratic probing does double hashing avoid? Explain why the choice of the secondary hash is nearly as important as the choice of the primary hash.
- Explain the process *separate chaining* or simply, *chaining* for managing collisions in a hash table.
- In pseduo-code, write code for `insert()` and `get()` of a chained hash table.
- In pseudo-code, write code for rehashing a chained hash table.
- If the load on a chained hash table is 1, what is the expected length of any chain in the hash table?
- If the load on the hash table is $\lambda$, what is the expected cost of a unsuccessful `get()` on a chained hash table? What is the expected cost of a successful `get()` on a chained hash table?
- Explain why the load on a chained hash table should be approximately 1? Use the formulas above in your argument.
- Consider a hash table with the hash function of $h(k) = k\%10$ into a hash table with 10 slots, indexed from 0 to 9: Draw the resulting hash tables for inserting $10, 4, 20, 19, 18, 79, 42, 5, 3, 15, 58$ into the hash table if linear probing, quadratic probing, and separate chaining.