As with previous chapters, we are going to lay out the learning objectives for this chapter, and split them into what you should get on the first reading vs the second reading.

1 First Reading: Understand Key Ideas

On your first reading, the goal is to get the big ideas, and build the mental scaffolding in which to hold the details on your second reading—the learning objectives for your first reading are generally **Remember** and **Understand** objectives. If you are missing a few details (but not completely lost), it is ok to move ahead, and come back to them the next time through—you might ask questions on the course forums too! Note you should **sleep at least one night between the first and second reading** to give your brain time to process the information.

- **Binary Search Trees: Intro**
  - **Explain** how binary search works, and why it has $O(lg(N))$ runtime.
  - **Explain** the main advantage of Binary Search Trees over Linked Lists.
  - **Explain** what condition the input must fulfil to perform binary search.
- **Binary Search Concepts**
  - **Explain** the structure and operation of a binary search tree.
    - Note: the fields in a Node will vary depending on how you are using the BST, for example, a BST that implements a Map will have key, value, left, and right; however, a BST that implements a Set will just have key, left, and right.
  - **Show** where/why a binary tree violates the ordering rules of a binary search tree (as in Figure 22.1.b).
    - Note: we usually save apply-level learning objectives for the second reading, however, you need to understand the BST-ordering rules well enough to do this task before understanding how/why specific operations work.
  - **Define** *node*, *edge*, *graph*, *path*, *directed path*, *undirected path*, *cycle*, *undirected cycle*, *connected*(graph), *tree*, *rooted tree*, *binary tree*, *binary search tree*, *parent* (of a node), *children* (of a node), *ancestors* (of a node), *descendants* (of a node), *depth* (of a node), *leaf nodes*, *sub-tree*, *height* (of a node), *height* (of a tree), *full* (binary tree), *balanced* (binary tree), *complete (binary tree)*.
  - **Define** *directed acyclic graph (DAG)*.
  - **Define** *totally ordered*.
  - **Explain** how a BST could be used to implement a set or map ADT, and what requirement there is on the key type for such an implementation.
  - **Explain** why/how BSTs can implement some operations efficiently that are not part of a typical map or set ADT.
  - **Define** *abstract syntax tree*.
  - **Recognize** that there are many useful kinds of trees that are not BSTs.
- **Adding to a Binary Search Tree**

- **Explain** why we cannot add to a BST by placing the new node as the root.
- **Explain** why we add BST nodes as leaf nodes.
- **Explain** how to add to a BST using recursion (including the base case, the recursive case, what is returned by each call, and how the root is updated at the end).
- **Explain** how to add to a BST by finding the parent of the node to add (including why the difficulties with this approach).
- **Explain** how to add to a BST with a `Node **` using iteration.
- **Recognize** that adding to a BST is $O(lg(N))$ if the tree is balanced, but may be not be, and instead be $O(N)$ if the tree is imbalanced.

- **Searching a Binary Search Tree**
  - **Explain** how to search a BST recursively for a particular element (including the two base cases and the recursive case).
  - **Explain** how to search a BST iteratively for a particular element (including the two ways we stop our loop, and how we update the current node if we keep going).
    - Note: There is no need for a `Node**` approach as we are not modifying the tree.
  - **Explain** why either approach for searching is $O(lg(N))$ assuming the tree is balanced.

- **Removing From a Binary Search Tree**
  - **Explain** how to remove from a BST when the node to remove has zero or one child.
  - **Explain** why it is more complex to remove a node from a BST that has two children.
  - **Explain** how to remove from a BST when the node to remove has two children.
  - **Explain** why there are three cases (not four) to consider when removing from a BST (including how/why we can collapse two cases into one and it still works).

- **Tree Traversals**
  - **Define** *traverse* (a data structure).
  - **Explain** how to perform an inorder traversal of a binary tree.
  - **Explain** how to perform an preorder traversal of a binary tree.
  - **Explain** why a preorder traversal is useful.
  - **Explain** how to perform an postorder traversal of a binary tree.
  - **Explain** why a postorder traversal is useful.

2 Second Reading

As always, we strongly recommend you sleep between your first and second reading. The second reading is where you want to focus on the higher-level learning objectives, which build on the lower-level learning objectives you worked on in the first reading. Sleeping gives your brain a chance to process the information from the first reading.

- **Categorize** A graph as a tree or not a tree.
- **Categorize** A tree as a binary tree, a binary search tree, or neither.
- **Compute** the height of a node in a tree.
- **Categorize** A binary tree as full or not, balanced or not, and complete or not.
- **Show** where/how to add a node to a binary search tree.
- **Write** code to add to a binary search tree.
- **Show** how a node is found in a binary search tree (or how you conclude it is not there).
- **Write** code to search a binary search tree.
- **Show** how a remove a node from a binary search tree.
- **Write** code to remove from a binary search tree.
- **Perform** an inorder, preorder, or postorder traversal on a binary search tree.
- **Write** code to do a inorder, preorder, or postorder traversal on a binary search tree.
- **Write** a BST to implement an appropriate ADT (*e.g.* a set or a map).
- **Write** code that uses BSTs and/or ADTs implemented with BSTs.
  - Note: `std::set` and `std::map` are STL implementations of the Set and Map ADT (respectively) which are implemented with balanced BSTs.
- **Design** an algorithm and **write** code to do other operations on a binary search tree.
  - Note that interviews commonly ask you to do unusual things to binary search trees. As with linked lists, the internet is full of more BST interview questions than you can imagine.

3 Key Learning Objectives

For interviews, you want to work to an ability to **implement** any wild and crazy operation on a BST. Many interviewers love BSTs as you can do a lot of really interesting operations on them, and they show your ability to work with pointers and develop moderately complex algorithms.

You might end up finding it useful to **write** some complex/strange operations on BSTs for real code. However, you also really want to be able to **write** code which makes good use of `std::set` and `std::map` (of course, understanding generally how these are implemented helps you use them).

As with all things, being able to **draw** diagrams of what is happening is key to your understanding, and helps you debug.

Finally, you will want to be able to **assess** if a BST is the right data structure for the problem you are trying to solve.