Part I

Part 1 was fairly straightforward. Linear probing lets us make several assumptions about its behavior. For one, from any arbitrary array slot, we can probe to every other slot in exactly M iterations (where M is the map capacity). No infinite loops to worry about! The trickiest part of part 1 was ensuring that, once we remove an item from the array, we probe forward and remove/re-insert each item in the cluster. If I needed to implement a quick and dirty hashing implementation, I would likely choose open addressing with linear probing as my first choice unless there was a good reason that it did not work well for my use-case.

Part II

I really enjoyed part 2, simply because I love working with linked lists. At each slot, I hold on to a head pointer to the first item in the chain, and include a tail dummy node at the end, but I don't store a pointer to the tail. This makes it easy to update the chain. For instance, when I insert a new item, I traverse until I either find the key (in which case I simply update that node's value), or encounter the tail dummy node. If I hit the tail, that means the key was not yet stored in the list, and I transform the tail into a regular node holding the new item and append a new tail.

When removing, I simply traverse the list until I either hit the tail (item not found), or find an item. In that case, I replace the current item with it's next node using a copy assignment operator, and delete the next node's pointer.

Part III Bucket

Part 3 introduced new key types, which were all straightforward to implement except for the c-strings. I hate passing pointers around, especially if they come from the user and refer to character arrays that may go out of scope and get. What I did to solve this was wrap all keys and values in their own class, Key and Value respectively. These classes use another wrapper class, GenericContainer, to hold their raw value. I use template specialization to treat c-strings as special. When GenericContainer is initialized with a c-string, it makes a copy for itself. Because Keys and Values can be replaced during normal map operations, we don't want to lose the pointers. Therefore, whenever a raw value must persist (ie, when returning the value to the client), I create another copy of the raw value and pass the pointer to the client. The client is then responsible for deleting the c-string if he wishing to avoid memory leaks. This strategy allows me to abstract away the pointer logic, greatly reducing code complexity.

In addition, part III requested a new method, cluster_distribution. To implement this, I create an integer array of size M (where M is the map capacity()). Each index in that array refers to a cluster size, and the associated value is used as a counter to track the number of encountered cluster instances of that size. Once I take inventory of every cluster, I traverse that array and pass extant clusters to a priority queue and return that, which sorts the clusters by size in N lg N time. The client can then read each cluster inventory item off the priority queue in order.

Aside from that, the bucket aspect of part 3 was failry straightforward and used essentially the same architecture as part 2.

Part III Open Addressing

Part 3's open addressing section was the most difficult aspect of the entire project. It turns out that implementing an effective hashing method is very tricky, and I settled on fairly rudimentary functions for each of the supported keys. I experimented with functors - structs that expose the function call operators and rely upon function overloading for behavior specific to their types. This is how the c++ standard library seems to support plug-and-play coding, and I found it very effective.

I allow the client to specify three functors - a map capacity planner, a primary hash, and a secondary hash. The default map capacity planner takes the minimum capacity desired by the client, and returns the next-highest prime number to use as the true map capacity.

I was not able to get the remove() methods working for the quadratic probing or the double hashing instances. Quadratic probing tends to have a nasty habit of not reliably visiting every potential slot. Double hashing uses the key itself to pick the next slot, which leads to the following scenario:

- Keys A and B give equalivalent primary hashing values but different double hashing values
- Key A exists in slot 1
- Key B is inserted, skipping slot 1 and probing forward to slot 2
- Key A is removed, and afterwards we attempt to resolve the cluster and reposition Key B so that it can be visited. However, Key A's probing value is dependent on Key A, so it is impossible to know that Key B originally intended to take Key A's slot, so when subsequently searching for Key B we will visit an empty slot where A was and encounter an empty slot

I'm sure there is a way to get the remove() methods working with double hashing and quadratic probing, but I ran out of time, and honestly I got really tired of fighting with the subtle mathematical landmine that hashing functions turns out to be. I have a new-found respect for cryptography researchers who deal with that kind of thing all the time.

Part IV

Part 4 was really fun. I implemented a BST base class and then exposed that to RBST and AVL. RBST simply overrode the insertion functionality to randomly insert at the root. Implementing all the methods for RBST was an exercise in recursive thinking, especially the pretty-print function which I was able to implement. I got a little carried away with playing with the pretty-print function, and I wanted to stress-test it to see how far I could push it. Because I allocate the output buffer (an array of lines) and write to it in memory before dumping it to standard output, a simple array requires contiguous memory and did not support the hundreds of millions of nodes that I wanted to print. Therefore, I used CDAL from the last project to store large chunks of lines together, but in different parts of memory.

As mentioned in the testing strategies, the BST class includes a few methods to manually and recursively calculate the height and number of children at each subtree if the _DEBUG_preprocessor macro is set to true. This is disabled by default because it is an expensive operation, but the unit tests set it to true, which makes ensuring the tree maintains correct structure throughout its lifetime.

Part IV Bonus: AVL Tree

The AVL tree was a bit tricky because I had to keep track of the height at every operation. I solved this once I went through and determined every operation that could potentially change the height of the subtree. Since each operation that could potentially change the height of a subtree was a recursive function, the base case would only change the height by a maximum of one. Therefore, at the base case, I call an update_height method, which simply calculates the height of the current node as one plus the maximum height between the two child nodes. After that operation completes, the recursion bubbles up, and the parent subtree does the same thing.

An elegant consequence of this strategy is that the AVL tree is simply the BST, but with the insert() and remove() functions wrapped in code that balances the current subtree at every recursion level. The balance() method is the largest block of code in the class, which ends up being relatively short and easy to read through and see what's going on. In addition, I wrap a few other BST methods with code that, if _DEBUG_is set to true, will recursively verify that abs(balance factor) is less than or equal to one at every subtree. Like the code that does the same for height and number of children, this is an expensive operation and is disabled by default, but it is very useful for unit testing.