Part 3 Buckets Testing Strategy

Paul Nickerson

In addition to separating tests by operations that are expected to fail and those that are expected to succeed, I also included a scenario for testing the cluster_distribution() function.

operation_failures.cpp

Within the failure operations scenario, I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string). To each instance, I start with an empty map and try to search() and remove() an item whose key does not exist in the map. Both calls should return a value less than zero (indicating key not found). I then fill the map with a bunch of items (it is impossible to run out of space because collisions are resolved via an arbitrarily-growable liked list). From this newly-filled map, I attempt to remove() a key that doesn't exist in the map, and search() for a key that does not exist in the map, both of which should return a value less than zero.

operation_successes.cpp

Within the success operations scenario, I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string). To each instance, I start by filling the map with a bunch of items, clearing it, then filling it up again. The map should then report the correct size. I check that several keys which are expected to exist in the map actually do exist (including the lowest possible key, the highest possible key, and one in the middle).

I check the print() function by routing it to an output string stream and count the number of hyphens in the output, which indicate empty slots. Since load factor = occupied buckets / capacity, we can get the number of unoccupied buckets as capacity * (1 - load). This should equal the number of hyphens in the print() output.

I then attempt to remove() several keys which are known to exist, and check that their associated values are what were expected. After these items are removed, I try to both search() and remove() them, which should all return false.

Part 3 added another function - remove_random() - which I test in a similar way to the preceeding remove() check. I remove a random key, then try to search() for it and remove() it. Both checks should fail and return a value less than zero.

cluster_distribution.cpp

Since cluster_distribution() returns a priority queue of clusters, and each cluster has a minimum size of one, all the clusters taken together should encompass every occupied slot. Therefore, I fill the map with a bunch of items, then clear it and fill it again to try and destablize the map. From there, I take the summation, over every cluster, of the cluster's size times the number of clusters having that size. The result of that summation should equal the output from the map's size() method. I do this four times, once for each of the four key types to be supported.