Part 4 RBST Testing Strategy

Paul Nickerson

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.

The RBST class keeps track of the height and number of children of each subtree with O(1) complexity during each operation that potentially changes those values. Because these values are so crucial to the map's functionality, and considering the difficulty of externally validating those values without adding more public methods, I include a preprocessor macro, _DEBUG_, which, when set to true (it defaults to false in production uses of the class), indicates to the BST base class to recursively verify the value of these values before each public method returns. This is potentially an expensive O(N) operation that significantly slows down map functionality, which is why it is disabled by default and specifically enabled during unit testing. If it is determined that the O(1) calculated values do not match the O(N), "true" values, the class throws an exception, which is caught by the CATCH testing framework and causes the test to fail.

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.

I test remove_random() 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.

tree_structure.cpp

Because I was able to successfully implement the pretty-print bonus method, it is fairly straightforward to verify that the underlying RBST map successfully maintains the correct BST tree structure. I print the tree structure to a file, then make a system() call to pipe that through a few command line utilities that extract just the numerical key (I say numerical, but that number could be in the form of a string of const char*). This results in the natural order of the keys as they exist in the tree. I pipe those through uniq and sort to remove duplicates and induce expected sorting order. If the tree exhibits the correct structure, the output of those two operations will match verbatim.

I start by filling the map to half capacity, verify that load() returns 0.5, then check validate the tree structure. Afterwards, I fill the tree to full capacity and then delete half the nodes via remove() and remove_random() operations in an attempt to destablize tree structure. Then I validate the tree structure again.

This series of checks is implemented for each of the four supported key types.