Part 3 Open Addressing 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_(probe type/double hash).cpp

I include three failure operations scenarios, one for each of the secondary hashing methods to be supported (linear probing, quadratic probing, and double hashing). Within each scenario, I test 4 versions of the map - one for each of the keys to be supported (int, double, string, and c-string).

Linear Probing

To the linear probing 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 to capacity, and attempt to insert an item where there is no space for it, which is expected to return a value less than zero. 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.

Quadratic Probing and Double Hashing

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

Because of these issues, I did not test the remove() functionality of the quadratic probing and double hashing instances, opting instead to simply fill the map to capacity and then search for a key that should not exist in the map. Note that quadratic hashing, as previously mentioned, has a tendency to sometimes ignore some slots, and the probability of this occuring increases drasticly as the load factor approaches 1. I early-abandon when this becomes evident to avoid an infinite loop, but, as a result, it was not possible to include a test to try and insert into a completely filled map like I could do in linear probing, since one key may induce an infinite loop while another key successfully finds one of the last remaining slots.

operation_successes_(probe type/double hash).cpp

Within the three success operations scenarios (one for each of the hashing methods), 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. The number of hyphens in the print() output should equal capacity() - size(), ie the number of unoccupied slots.

As previously mentioned, I was unable to successfully implement remove() functionality in quadratic probing and double hashing, but within linear probing I 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.

cluster_distribution_(probe type/double hash).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 for each of the hashing method instances, once for each of the key types supported (a total of 12 times).