Assignment Dictionaries and Hashing

COSC 2336: Data Structures and Algorithms

Summer 2021

Objectives

- Learn about dictionary abstraction, the idea of organizing data as key/value pairs to search by the key to retrieve a value, which is usually a database record or structure.
- Implement and learn about hashing techniques.
- Look at and understand mid-square hashing and quadratic probing for a closed hash table implementation.
- Continue to use class templates and C++ object oriented techniques for organizing abstract data types.
- Understand how hashing techniques can achieve $\mathcal{O}(1)$ constant time performance for usual case of inserting and finding items int a container.

Description

In this assignment you will be implementing some basic mechanisms of a hash table to make a concrete Dictionary API that uses hashing to store and search for items in its collection. You have been given many files for this assignment. I have provided an `Employee` class in `Employee.[hpp|cpp]` and a `KeyValuePair` class in `KeyValuePair.[hpp|cpp]`. You will not need to make any changes to these files or classes, they should work as given for this assignment.

You will be adding and implementing some member functions to the `HashDictionary` class. The initial `HashDictionary.[hpp|cpp]` file contains a constructor and destructor for a `HashDictionary` as well as some other accessors and operators already implemented that are used for testing.

You will be implementing a closed hash table mechanism using quadratic probing of the slots. You will also implement a version of the mid-square hashing function described in our textbook. You should review our textbook discussion of hashing, as we base our assignment on the `Dictionary` and `KeyValuePair` abstractions covered in Shaffer 3.2 C++ Edition section 4.4 and 9.4

Section 4.4 of our text covers the `Dictionary` abstraction. Really the `Dictionary` defines an abstract API with three main functions, `insert`, `remove` and `find`. So actually our previous `BinaryTree` abstraction was really just the same as this weeks `Dictionary` abstraction. And in both cases, we manage key/value pairs, though in this assignment, as discussed in our textbook section 4.4, we have further abstracted this concept into a separate `KeyValuePair` class. You will find headers and implementations for both of these classes in this assignment.

The `Dictionary` class is an abstract base class defining the API for managing a dictionary of items. A dictionary should be efficient to insert a `value` or record that is indexed by a `key`, and should be efficient to search for and remove that record if asked and you give it the `key` that indexes the record. So really our previous assignment, the `LBinaryTree` could be redefined this week to be an implementation of a `Dictionary` abstraction. The tree would be one way of implementing the `Dictionary` API efficiently. Recall from our last unit that `insert`, `find` and `remove` are all $\mathcal{O}(\log_2 n)$ operations when implementing the dictionary as a tree, and assuming the tree remains relatively balanced.

For this assignment, we will be implementing a concrete `Dictionary` type using an allocated/managed array of values. We will use hashing techniques on the array to `insert`, `find` and `remove` values from the container. Thus this implementation is called a `HashDictionary`. If we were to move our previous assignment tree implementation into this framework, we might think of the tree as a linked list implementation of the `Dictionary` API, and rename it `TreeDictionary`.

As discussed in this units materials, using a hash to implement the `Dictionary` API can seem a bit magical. All

three operations, `insert`, `find` and `remove` are constant time $\mathcal{O}(1)$ operations using a hashing technique, with some caveats. As usual, as you have experienced, since we are managing a fixed block of memory as an array, if the array becomes full we do have to grow the array by doubling it and copying the existing `values` into the newly allocated array. So `insert` can be constant time normally, but may be $\mathcal{O}(n)$ occasionally when we need to grow the array to fit more values into the container.

Likewise, `find` and `remove` can degenerate in performance. This depends on the hashing technique used. There are two main techniques, as discussed in our Shaffer textbook section 9.4: open hashing and closed hashing. Open hashing involves resolving collisions by creating a linked list of values that hash to the same index (collide). Thus open hashing is a combination of an array based implementation with the use of linked lists to manage and resolve hash collisions. We are using a closed hashing technique in this assignment. In closed hashing we resolve collisions by performing a linear search (possibly with some probing scheme) in the array. Thus if the hash table becomes very full, `find` and `remove` might end up needing a long linear search to find the `key`, and can thus degenerate to $\mathcal{O}(n)$ performance in the worst case.

However the worst case both for insertion and for `find` and `remove` can be minimized and do not occur often, especially if, for closed hashing, the hash table is kept relatively sparse. Thus you will see that the `growHashTableIfNeeded()` function given to you for this assignment will actually double the size of the table, not when the table is full, but when the table reaches 50% capacity. This is for performance reasons, to try and keep the hash table relatively sparsely populated, so that long linear searches do not occur frequently for `find` and `remove` operations.

For this assignment you will be completing the 3 basic API methods for the `Dictionary` abstraction, `insert`, `find` and `remove`. In addition, you will need to implement two functions `hash` and `probe` to support calculating the hash index for a `key`, which are used by all 3 of the methods to determine where in the array to start looking for a `key` to insert or return its value.

Overview and Setup

For this assignment you will be given the following files that you will be using and adding code to for this assignment.

File Name	Description
`src/test-HashDictionary.cpp`	Unit tests of the array based hashing `HashDictionary` implementation of the `Dictionary` API
`include/Dictionary.hpp`	Header file of the ADT base `Dictionary` class that defines the `Dictionary` interface / abstraction
`include/HashDictionary.hpp`	Header file of the concrete array based hashing implementation of the `Dictionary` abstract data type
`include/KeyValuePair.hpp`	Header file of the `KeyValuePair` template abstraction, used to store key/value record pairs in `Dictionary` containers.
`src/Dictionary.cpp`	Implementation file of common methods of the `Dictionary` base class
`src/HashDictionary.cpp`	Implementation file for the `HashDictionary` member functions that implement the concrete array hashing based `Dictionary` API
`src/KeyValuePair.cpp`	Implementation file of methods for the `KeyValuePair` template abstraction to manage key/value pairs in `Dictionary` containers.

This week you will be adding implementations into the `HashDictionary` class, which is a concrete array based hashing implementation of the `Dictionary` abstraction. You will be using and storing `KeyValuePair` instances in your container. The `Employee` class is used in the testing of your hashing dictionary implementation.

As usual, before starting on the assignment tasks proper, you should make sure you have completed the following setup steps.

1. Copy the assignment repository on GitHub using the provided assignment invitation link for 'Assignment Recursion' for our current class semester and section.
2. Clone the repository using the SSH URL to your local class DevBox development environment.
3. Checkout the 'origins/feedback' branch to your local working DevBox repository.
4. Configure the project by running the `configure` script from a terminal.

5. Confirm that the project builds and runs, though no tests will be defined or run initially. If the project does not build on the first checkout, please inform the instructor.
6. You should create the issue for Task 1 and/or for all tasks for the assignment now before beginning the first task. On your GitHub account, go to issues, and create it/them from the issue templates for the assignment. Also you should close the initial Pull request that should be automatically opened for you, so that you can open your own when committing your work.

Assignment Tasks

Task 1: Implement the `HashDictionary` private `probe()` and `hash()` Methods

As usual make sure that you have created Task 1 on your GitHub repository for this assignment and are ready to create a new Pull request for this assignment. There are three test cases in the test file for the Task 1 functions. You can uncomment the first one and work on the `probe()` function, and then uncomment the next two for the `hash()` function in order, but as usual you should create the declarations for the functions and make sure your code still compiles and runs the uncommented tests successfully before beginning your work on implementing these functions.

Both the `probe()` and `hash()` functions will be private member functions, internal to the `HashDictionary` class. So the declarations of these methods should be put in the `private` section, after the declarations of the member variables of the class.

The `probe()` function will hopefully not be too hard to implement, it is meant as a bit of a warm up for this assignment. Your `probe()` private method should implement a quadratic probing scheme as described in our Shaffer textbook section 9.4.3 on pg. 338. Use `c1 = 1, c2 =2, c3 = 2` as the parameters for your quadratic probe (the tests of `probe()` assume your probe sequence is using these parameter values for the quadratic function). So for example, lets say `probeIndex` is the probe sequence index. The values for the parameters mean we want you to use the quadratic formula

$$\text{probeOffset} = c1 \times \text{probeIndex}^2 + c2 \times \text{probeIndex} + c3 \tag{1}$$

So for example, when the probe index is 0, the probe offset you should return will be $1 \times 0^2 + 2 \times 0 + 2 = 2$. When the probe index is incremented to 1, the probe offset you calculate and return would be $1 \times 1^2 + 2 \times 1 + 2 = 5$, etc. The expected return value from the tests should help you understand what value is expected to be returned by calling the `probe()` function.

The `probe()` method should return an `int` value, which is the value to add onto the hash slot for a hash probe sequence. The function takes two parameters as input, a `const` reference to a `Key`, and an `int` representing the probe sequence index. Make sure that the key and value parameters passed into your functions are all `const` reference parameters, e.g. `const Key& key` in this case. And finally, calling the `probe()` method does not modify the `HashDictionary` instance, so it should be declared as a `const` member function.

The first `key` parameter is not actually used in your function. As discussed in our text, some probe sequences might vary the sequence based on the key, but simple quadratic probing always gives a deterministic sequence of probe offsets no matter the key.

You should also uncomment the tests and implement the `hash()` private member function for task 1. You will be implementing a hash function for integer like keys using the described mid-square hashing function (Shaffer 9.4.1 Example 9.6 pg. 327). You will create a slight variation of this algorithm for our hashing dictionary. First of all the `hash()` member functions should take a 'const' reference to a `Key` as its only input parameter, and it will then return a regular `int` as its result. This is also a private member function, and it should be declared as a `const` member function again, since calling `hash()` does not modify the instance in any way. Since this is a hash function, the integer value should be in the range `0 - allocationSize - 1`, so don't forget to `mod` by the `allocationSize` of the current hash table before returning your hash result.

The algorithm to implement your `hash()` member function is as follows.

1. First of all, you should square the `key` value that is passed in. However do not use the `<cmath>` library function to square the key, since this will return back a `double` result. Simply multiply the `key` by itself to square it. **HINT**: since you should be passing this parameter in as a `const` reference, you might find the compiler

complains if you try and assign a new value back into the `key` parameter you pass in. You may need to make a local copy of the `key` to manipulate for your hash function here.

2. Then, assuming we are working with a 32 bit int, we want to only keep the middle 16 bits of the square of the key to use for our hash, which is the 'mid' part of the mid-square hashing. There are many ways to work with and get the bits you need, but most likely you will want to use C bit-wise operators to do this. For example, a simple method to get the middle 16 bits is to first mask out the upper 8 bits using the bit-wise & operator (e.g. `key & 0x00FFFFFF`) will mask out the high order 8 bits to 0). Then once you have removed the upper most significant 8 bits, you can left shift the key by 8 bits, thus dropping out the lower least significant 8 bits (e.g. `key >> 8`). Performing a mask of the upper 8 bits and shifting out the lower 8 bits will result in you only retaining the middle 16 bits.
3. Finally, after squaring your key and getting the middle bits of the key, you need to recast the resulting index into the range from 0 to the maximum size of the current hash table minus 1. For the `HashDictionary` instance, the member variable `allocationSize` keeps track of the current size of the hash table, so you will need to use modulus arithmetic to recast your result into the correct range to be a proper hash index.
4. Return this calculated hash index slot as the result from your `hash()` function.

When you are satisfied with your work on the `probe()` and `hash()` methods, you should be able to pass all of the tests for task 1 in our testing file. Once working, commit your changes for these two function, and push them to the `feedback` branch of your assignment repository.

Task 2: Implement the `HashDictionary` private `probeForAvailableSlot()` and `probeForKeySlot()` Methods

In this hashing implementation, we are implementing a version of closed hashing. This means that, if a hashing collision occurs, we will use a probing sequence to search for subsequent slots in the hash table to either find the key we are looking for or to find an empty slot if we are trying to insert a new key into the collection.

We can make the actual implementation of the `Dictionary` public API methods almost trivial if we just implement two private methods to perform probe sequences on your closed hash table. The `insert()`, `find()` and `remove()` methods will use these two methods to do almost all of their actual work, so the bulk of the functionality of the closed hashing algorithm comes from the correct implementation of these probe methods.

The next two test cases in the test file are for the task 2 functions. We recommend you do these two functions one at a time, so start by uncommenting the first task 2 test case, and adding a stub for the `probeForAvailableSlot()` and making sure you can compile and run the uncommented tests before beginning your implementation.

The purpose of the `probeForAvailableSlot()` function is to search the hash table probe sequence to find either an empty or missing slot. This method will be used when we need to insert a new value into the hash table. A `slot` in the hash table is initially empty when the hash table is created, before any value is assigned to that slot. However, as discussed in our text, there are issues we need to deal with when removing key/value pairs for a closed hashing implementation. Searching for a key in the next function stops when we come to an empty slot in the probe sequence. So if we simply make a slot `empty` when we remove a key/value pair from the table, this can cause incorrect behavior if the value we are searching for is further along the probe sequence of the table. So we need a second state for a hash table slot, that we call a `missing` state. When a key/value pair is removed from the table, we simply set its state to be `missing`. This means when we search for a `key`, we will only stop if we hit an `empty` slot, we should keep going in the search if we reaching `missing` slots, as the key we are searching for could still be further along in the probe sequence. But when searching for a location to insert a new value, as you are doing with this first helper method, we want to stop on the first `empty` or the first `missing` slot we find in the probe sequence.

So the `probeForAvailableSlot()` method takes a `const` reference to a `Key` as input. It is again a `const` member function, as doing a probe does not actually change anything yet in the hash table, it is only returning information. The result returned from this function is an index or `slot` into the hash table, so the returned value should be in the range from 0 to the table `allocationSize - 1`.

For both of these probe methods, the general idea to implement the probe sequence is as follows. Lets say that we have a table with an `allocationSize` of 7, this means that the valid hash table slots or indexes range from 0 to 6. Start by using the `hash()` function to determine the initial home slot. Lets say that for the key you are probing for you get a hash index of 3. Then to do the probe, you need to loop through the probe sequence using the `probe()` functions, starting with a probe index of 0. Recall, that for your quadratic probe function, it will return a probe sequence of `2, 5, 10, 17, 26, ...` for probe indexes `0, 1, 2, 3, 4, ....` So for the initial probe index of 0,

you need to add the probe sequence 2 to the initial hash key home of 3 in this example, meaning the first slot you will check in the hash table will be slot 5. If this slot is not available (it holds a key, it is neither empty nor missing), then you need to calculate the next slot in the sequence, which will be at index `3 + 5 = 8`, though notice this goes beyond the end of the current hash table `allocationSize`, so after adding in the probe sequence to the base hash, you need to do a mod by the `allocationSize`, resulting in looking at index 1 next, etc.

So the general algorithm for `probeForAvailableSlot()` is as follows:

1. Determine the home slot for the key using the `hash()` function.
2. Start at probe index 0
3. While not done
   - Calculate the probe slot by adding the home slot has and the probe offset calculated from `probe()` for the current probe index
   - test the resulting probe slot, if it is `empty` or `missing` then you are done, and you want to return this calculated probe slot as your result of the location of the first missing or empty slot along the probe sequence for this key.
   - otherwise keep searching by incrementing the probe index and returning back to the top of this loop task 3.

Once you have the `probeForAvailableSlot()` method working, you can uncomment the tests and work on the `probeForKeySlot()` method. This method should be almost identical to the previous method. It has the same function signature, and it returns a valid hash table slot. The difference is this method is searching for the slot of the indicated key given as input. It should return the hash table slot where it finds the key, or if it ever comes to an `empty` slot during the probe sequence, it should return that empty slot which indicates a failed search for the key. So this method has an identical algorithm to the previous one, you just need to test if the slot is empty or is equal to the key you are searching for in the loop of the algorithm.

Once you have implemented this work for the two private member functions you should be able to pass the test cases now for task 2. When you are satisfied with your work, commit your changes and push your commit to the `feedback` branch of your assignment repository.

Task 3: Implement the `HashDictionary` public `insert()` API Method

The last 3 tasks are to implement the actual public API methods of our `Dictionary` base class we are making a concrete implementation. Most of the actual work for the API methods has already been completed, and is being done by the 4 private helper methods you should have completed, so the implementation of the public API methods should be relatively simple.

As usual start by uncommenting the test for task 3 in the test file. Also, in `Dictionary.hpp` the declaration for the virtual `insert()` method has been commented out, you should uncomment this declaration. The declaration in `Dictionary.hpp` gives you the signature you need for your `insert()` method (minus the `virtual` keywords). The `insert()` method takes a `Key` and a `Value` as inputs, which are the key/value pair that needs to be inserted into the table. Both of these parameters are `const` reference parameters. The insert method is a `void` function, it does not return any explicit result from performing the insertion of the new key/value pair into the dictionary.

To implement `insert()` perform the following steps:

1. First determine if an attempt is being made to insert a duplicate key. For the dictionary collections we don't allow duplicate keys in the dictionary, each key must be unique. So use the `probeForKeySlot()` method to first search for the key being inserted. This function returns a slot in the hash table. If the returned slot is not empty, or equivalently if the key in the returned slot is the same as the key that is trying to be inserted, then we need to throw an exception to inform the caller that they are trying to do something bad. Throw a `DictionaryDuplicateKeyInsertionException` if you detect an attempt to insert a duplicate key into the collection.
2. If the key is not a duplicate, this means we will be inserting the new key/value pair into the dictionary. Call the `growHashDictionaryIfNeeded()` method before doing the actual insert, to ensure that the hash table is currently of adequate size to insert the new key/value (or to grow to a bigger size if needed before you do the insert).
3. After calling the grow function, you are ensured the hash table is of adequate size now to perform the insertion. Call the `probeForAvailableSlot()` for the `key` to determine the correct slot along the probe sequence where the new key/value pair should be inserted.

4. Insert the key/value pair into the table at the indicated available slot. You need to create a new instance of a `KeyValuePair<Key, Value>` with the input `key` and `value` parameters, and assign this into the `hashTable` slot.
5. Finally your hash table has grown in `size` by 1 with the insertion. Don't forget to increment the `size` member variable, which is defined in the `Dictionary` base class.

NOTE: In step 2 you call `growHashDictionaryIfNeeded()`. Actually this method calls the `insert()` method if it needs to grow the hash table, which is the method you are currently working on. This is a bit of a catch-22. The call of `insert()` is commented out. But once you have a stub function of `insert()` compiling, you should uncomment the call to `insert()` in the `growHashDictionaryIfNeeded()` method. There is also a call to `insert()` in the array based constructor which should also be uncommented at this point. This is an example of indirect recursion. The `insert()` method calls the `growHashDictionaryIfNeeded()` which in turn might recursively call `insert()` if it needs to grow and reinsert values back into the table. Also note that the test for a duplicate key needs to happen before calling the grow function. The table should not grow prematurely if a call is made to insert a duplicate key, the insertion is not actually going to be performed. Thus in the steps above, growing the hash table only happens after we are sure that a new key/value pair is in fact going to be inserted into the table.

Once you are satisfied with your work and passing the tests, commit your implementation and push it to the `feedback` branch of your repository. Some tests for task 3 or later may not pass unless you uncomment the calls to `insert()` in the `growHashDictionaryIfNeeded()` and in the array based constructor, so make sure you have uncommented those calls there when working on your `insert()` method.

Task 4: Implement the `HashDictionary` public `find()` API Method

The `find()` and `remove()` member methods are even simpler than insert, and are very similar to one another. Again start by uncommenting the `find()` virtual declaration in the `Dictionary.hpp` header file. This will give you the signature of the `find()` method. `find()` takes a `const` reference to a `Key` as input and returns a `Value` as its result. As the name suggests, this method performs a search for the indicated `key` and returns the `value` associated with the `key` in the collection. If the search for the `key` is unsuccessful, then the method should throw a `DictionaryKeyNotFoundException` to notify the caller that the `key` is not present.

The steps than to implement the `find()` method are as follows:

1. Use the `probeForKeySlot()` method to search for the slot for the given input `key`.
2. Test the returned hash table slot. If the slot is empty it means the search for the `key` failed and you should throw the `DictionaryKeyNotFoundException`.
3. Otherwise you found the key, so you can simply retrieve and return the value from the hash table slot that contains that key/value pair.

And that is it for the `find()` method. Once you are satisfied with your work and can pass the tests for task 4, commit your changes and push them to the `feedback` branch of your assignment repository.

Task 5: Implement the `HashDictionary` public `remove()` API Method

Finally implement the `remove()` method. The `remove()` method is almost identical to the `find()` method, down to the API for `remove()` is defined to return the `value` of the `key` you are removing, the same as is done for a `find()`. There are two test cases defined for task 5 in the test file. You can start by uncommenting only the first test case. Try the second set of test cases after you get your function implemented, which test specifically that the slot is being set to `missing` for successful removes, and that `find()` and `insert()` still work correctly when slots are set to `missing`.

The algorithm for `remove()` is identical to `find()`. The only difference is that in step 3, you should set the slot where you find the value to `missing` before you return the value. Be careful, calling `setMissing()` on the slot will actually cause the key/value pair in the slot to be removed. So before calling `setMissing()` you will need to make a copy of the value you are about to return.

Once you are passing the first test case of the `remove()` function, don't forget to uncomment the second test case and make sure these tests pass as well. Once you are satisfied with your work, commit your changes for the `remove()` function and push them to the `feedback` branch of your assignment repository.

Assignment Submission

For this class, the submission process is to correctly create pull request(s) with changes committed and pushed to your copied repository for grading and evaluation. For the assignments, you may not be able to complete all tasks and have all of the tests successfully finishing. This is ok. However, you should endeavor to have as many of the tasks completed before the deadline for the assignment as possible. Also, try and make sure that you only push commits that are building and able to run the tests. You may loose points for pushing a broken build, especially if the last build you submit is not properly compiling and running the tests.

In this problem, up to 25 points will be given for having at least 1 commit that compiles and runs the tests (and at least some attempt was made to work on the first task). Thereafter 15 points are awarded for completing each of the 5 tasks. However you should note that the autograder awards either all points for passing all tests, or no points if any test is failing for one of the tasks. Also note that even if you pass all tests, when the instructor evaluates your assignment, they may remove points if you don't follow the requirements for implementing the code (e.g. must reuse functions here as described, need to correctly declare parameters or member functions as `const` where needed, must have function documentation correct). You may also loose points for style issues. The instructor may give back comments in pull requests and/or create new issues for you if you have issues such as these, so it is good to have work committed early before the due date, so that the instructor may give feedback requesting you to fix issues with your current submission.

Program Style

At some point you will be required to follow class style and formatting guidelines. The VSCode environment has been set up to try and format your code for some of these guidelines automatically to conform to class style requirements. But not all style issues can be enforced by the IDE/Editor. The instructor may give you feedback in your pull comments and/or create issues for you for the assignment that you need to address and fix. You should address those if asked, and push a new commit that fixes the issue (or ask for clarification if you don't understand the request). In general the following style/formatting issues will be required for programs for this class:

1. All programs must be properly indented. All indentation must be consistent and lined up correctly. Class style requires 2 spaces with no embedded tabs for all code indentation levels. The editor style checker should properly indent your code when you save it, but if not you may need to check or correct this if code is misaligned or not properly indented.
2. Variable and function names must use `camelCaseNameingNotation`. All variable and function names must begin with a lowercase letter. Do not use underscores between words in the variable or function name. Often function names will be given to you, but you will need to create variables, and maybe some functions, that conform to the naming conventions.
    - Global constants should be used instead of magic numbers. Global constants are identified using `ALL_CAPS_UNDERLINE_NAMING`.
    - User defined types, such as classes, structures and enumerated types should use camel case notation, but should begin with an initial upper case letter, thus `MyUserDefinedClass`.
3. You are required to use meaningful variable and function names. Choosing good names for code items is an important skill. The code examples and starting code tries to give examples of good and meaningful names. In general, do not use abbreviations. Single variable names should be avoided, except maybe for generic loop index variables `i`, `j`, etc. Make your code readable, think of it as writing a document to communicate with other developers (and with your instructor who will be evaluating your code).
4. There are certain white space requirements. In general there should usually never be more than 1 blank line in a row in your code. Likewise there should usually not be more than 1 blank space on a line. There should be 1 blank space before and after all binary operators like `+`, `*`, `=`, `or`.
5. Function documentation is required for all regular functions and all class member functions. You need to follow the correctly formatted Doxygen function documentation format. We will use function documentation generation, and you should be sure your documentation can be built without emitting warnings or errors. Likewise all files should have a file header documentation at the top. You should edit the file header of files where you add in new code (not simply uncommenting existing code). Make sure the information has your correct name, dates, and other information.
6. Practice using proper Git commit messages. You should refer to issues and tasks correctly in commit messages.

Additional Information

The following are suggested online materials you may use to help you understand the tools and topics we have introduced in this assignment.

- U12-1 Dictionaries and Hashing
- C++ Classes and Objects
- C++ Inheritance
- C++ Templates