Project 3: Hash Tables and Splay Trees

Due: 03/08/2024 at 11:59PM

General Guidelines:

The APIs given below describe the required methods in each class that will be tested. You may need additional methods or classes that will not be directly tested but may be necessary to complete the assignment. Keep in mind that anything that does not need to be public should generally be kept private (instance variables, helper methods, etc.). Additional file creation isn't allowed as Vocareum will only compile the files given in the startercode.

In general, you are not allowed to import any additional classes or libraries in your code without explicit permission from the instructor! Adding any additional imports will result in a 0 for the entire project.

Note on Academic Dishonesty:

Please note that it is considered academic dishonesty to read anyone else's solution code to this problem, whether it is another student's code, code from a textbook, or something you found online. **You MUST do your own work**! You are allowed to use resources to help you, but those resources should not be code, so be careful what you look up online!

Note on implementation details:

Note that even if your solution passes all test cases, if a visual inspection of your code shows that you did not follow the guidelines in the project, you will receive a 0.

Note on grading and provided tests:

The provided tests are to help you as you implement the project. The Vocareum tests will be similar but not exactly the same. Keep in mind that the points you see when you run the local tests are only an indicator of how you are doing. The final grade will be determined by your results on Vocareum.

<u>**Project Overview:**</u>

In this project, you will work on three problems. First, you will implement a hash table data structure with linear probing for collision management. Next, you will implement a type of self-adjusting binary search tree known as a Splay Tree. Finally, you will combine the two data structures into an adaptive hash table using Splay Trees for collision management.

Note: throughout this project, you are expected to use C++ smart pointers as they will greatly simplify memory management and safety. See Appendix A for details.

<u>**Part 1: Hash Table with Open Addressing (Linear Probing) (40 Points):**</u>

<u>What is a Hash Table?</u>

A hash table is a data structure that offers fast insertion, deletion, and lookup of key-value pairs. Each 'key' is processed through a hash function that converts it into an index to an array element (the table) where the 'value' is stored. Because the hash function typically computes the index in constant time, operations in a hash table can be extremely fast and efficient.

<u>Open Addressing and Linear Probing</u>

When two keys hash to the same index, a collision occurs. There are several strategies to handle collisions, and open addressing is one of the most common. In open addressing, all elements are stored within the array itself, and the collision is resolved by finding another spot within the array for the colliding item.

Linear probing is a simple and efficient method for resolving collisions in an open-addressed hash table. When a collision occurs, linear probing searches for the next available slot by moving sequentially through the table from the point of collision. When it reaches the end of the table, it wraps around to the beginning and continues the search.

Deletion in an open-addressed hash table typically involves replacing the deleted item with a dummy element, or "tombstone", such that searching operations can quickly determine whether an element exists in the table. However, in this project we **do not** use tombstones; when deleting an item, it is replaced by null. As a consequence, searching operations must search the entire table to determine whether an element exists.

By completing this project, you will deepen your understanding of hash tables and gain valuable experience with algorithm design and data structure implementation.

Class: hash_map:

In hash_map.hpp, you will implement the methods of the *hash_map* template class. This class has two template arguments K and V, representing the datatypes for the keys and values respectively. The nested *hash_map_node* class contains the key-value pair. The hash table itself is represented by a vector of pointers to *hash_map_nodes*. If a bucket in the table is a null pointer, then that bucket should be considered empty.

Public methods/function to implement:

You will find the functions and class members listed within the starter codes provided in this project for you to implement. Documentations of each function and class members are also provided. Please read carefully before you get started. You may need to throw two exceptions, *duplicate_key* and *nonexistent_key* as described in the functions below.

Implement the following functions:

1. ***template <typename K, typename V> hash_map<K,V>::hash_map():***
   template <typename K, typename V> hash_map<K,V>::hash_map(const size_t bucketCount):
   Initialize an empty hash table with the specified number of buckets (or 1 if *bucketCount* is omitted).
2. ***template <typename K, typename V> size_t hash_map<K,V>::hash_code(K key) const:***
   Calculate and return the hash code of the given key. For this project, we will use a very simple hash function:

   ```
   hash = key % bucketCount;
   ```

   Since *key* is a template argument, it can be of any datatype. For this project, you can assume that the modulus operator % is overloaded for all key types.
3. ***template <typename K, typename V> void hash_map<K,V>::resize(const size_t bucketCount):***
   Resize the hash table to *bucketCount* buckets, and rehash all the existing elements. If the *bucketCount* is less than the current number of elements, return without resizing the table.
4. ***template <typename K, typename V> void hash_map<K,V>::insert(const K& key, std::unique_ptr<V> value):***
   Insert the key and its corresponding value in the hashtable. Throw a *duplicate_key* exception if the key is already present in the hashtable. Otherwise, if the hashtable is full, double its size (rehashing all existing elements), and then insert the pair.
5. ***template <typename K, typename V> const std::unique_ptr<V>& hash_map<K,V>::peek(const K& key):***
   Return a reference to the value corresponding to the given key. Throw a *nonexistent_key* exception if the key doesn't exist in the hashtable.
6. ***template <typename K, typename V> std::unique_ptr<V> hash_map<K,V>::extract(const K& key):***
   Remove and return the value corresponding to the given key from the hashtable. Throw a *nonexistent_key* exception if the key doesn't exist in the hashtable.
7. ***template <typename K, typename V> size_t hash_map<K,V>::size() const:***
   Return the number of elements in the hashtable.
8. ***template <typename K, typename V> size_t hash_map<K,V>::bucket_count() const:***
   Return the current number of buckets in the hashtable.
9. ***bool hash_map<K,V>::empty() const:***
   Return true if the hashtable has zero elements, and false otherwise.

Part 2: Splay Tree (40 Points):

What is a Splay Tree?

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up, and removal in amortized *O(log n)* time.

What makes splay trees unique is their ability to "splay" on every operation. Splaying involves a series of tree rotations that move the accessed item to the root of the tree, thus optimizing the tree for frequent access to elements.

The Splay Operation

The splay operation is the heart of the splay tree. When a node is accessed, the splay tree performs a splay operation that consists of a sequence of *rotations* to move that node to the root. The splay operation not only serves to optimize access times but also helps in keeping the tree balanced. There are three types of rotations used in the splay operation:

Zig: A single rotation is used when the node is a child of the root.
Zig-Zig: A double rotation is used when both the node and its parent are either left or right children.
Zig-Zag: A double rotation of mixed types (one left and one right rotation) is used when the node is a left child and the parent is a right child, or vice versa.

When a node is splayed, the above operations are repeated until that node becomes the root. Refer to the below figures for diagrams of the different rotation operations. Note that in all cases, the respective order of nodes within the tree is preserved, i.e. X < Y < Z.

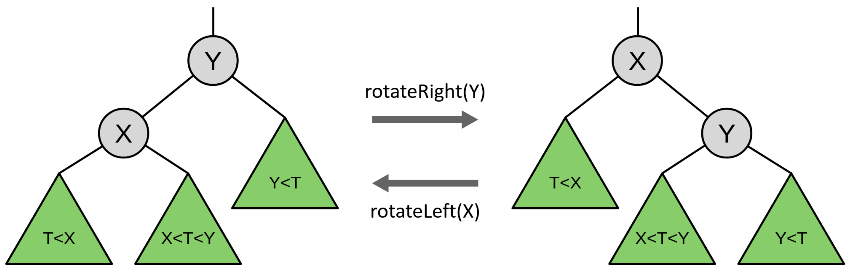

Figure 1: Basic rotation operations

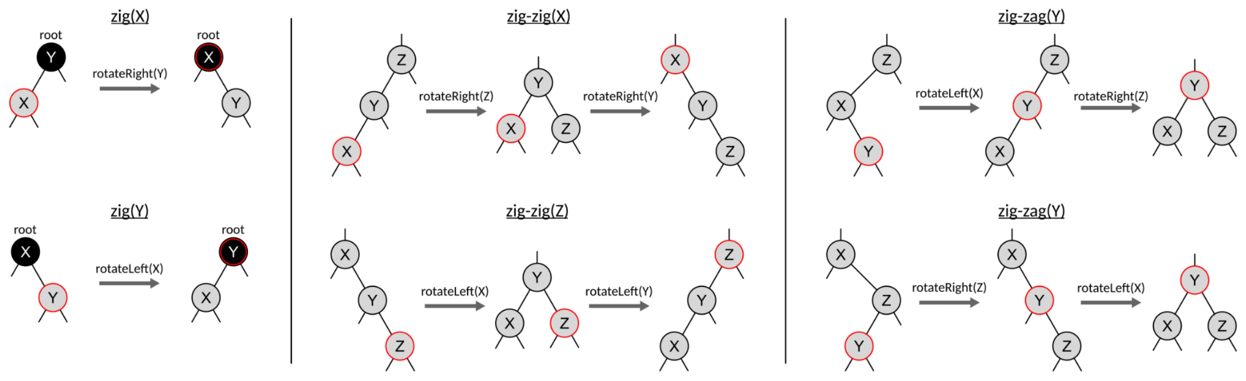

Figure 2: zig, zig-zig, and zig-zag operations for each possible configuration. The node being operated on is highlighted in red.

Class: splay_tree:
In splay_tree.hpp, you will implement the methods of the *splay_tree* template class. As with part 1, this class has two template arguments K and V, representing the datatypes for the keys and values respectively. The nested *splay_tree_node* struct contains a key-value pair along with pointers to the left and right children and parent nodes.

Note that the children nodes are referenced by *shared_ptr*s, while the parent node is referenced by a *weak_ptr*. This is to avoid cycles of references. To access a parent node from a *weak_ptr*, use the lock() method. For example, if you have a node *node*, to access it's parent you would do:

```
std::shared_ptr<splay_tree_node> parent = node->parent.lock();
```

If the parent happens to be null, it will return a null pointer. More details can be found in Appendix A.

Public methods/function to implement:

You will find the functions and class members listed within the starter codes provided in this project for you to implement. Documentations of each function and class members are also provided. Please read carefully before you get started. You may need to throw the exceptions *duplicate_key, nonexistent_key* and *empty_tree* as described in the functions below.

Implement the following functions:

1. ***template <typename K, typename V>splay_tree<K,V>::splay_tree():***
   Perform any needed initialization here.
2. ***template <typename K, typename V>void splay_tree<K,V>::insert(const K& key, std::unique_ptr<V> value):***
   Insert the key-value pair in the tree in a BST fashion, by comparing with the keys of the existing nodes in the tree. Throw a *duplicate_key* exception if the key is already present. After insertion, splay the inserted node to the root.
3. ***template <typename K, typename V>const std::unique_ptr<V>& splay_tree<K,V>::peek(const K& key):***
   Return a reference to the value corresponding to the given key in the tree. Throw a *nonexistent_key* exception if the key doesn't exist. Splay the node containing the key if it was found.
4. ***template <typename K, typename V>std::unique_ptr<V> splay_tree<K,V>::extract(const K& key):***
   Remove and return the value corresponding to the given key. Throw a *nonexistent_key* exception if the key doesn't exist. When deleting the node from the tree, first splay the node and delete it using standard BST deletion.
5. ***template <typename K, typename V>K splay_tree<K,V>::minimum_key():***
   Return the minimum key in the tree. Throw an *empty_tree* exception if the tree is empty. The node with the minimum key should be splayed once it is located.
6. ***template <typename K, typename V>K splay_tree<K,V>::maximum_key():***
   Return the maximum key in the tree. Throw an *empty_tree* exception if the tree is empty. The node with the maximum key should be splayed once it is located.
7. ***template <typename K, typename V>bool splay_tree<K,V>::empty() const:***
   Return true if the tree is empty (i.e., root is null) and false otherwise.
8. ***template <typename K, typename V>size_t splay_tree<K,V>::size() const:***
   Return the number of elements present in the tree.

Part 3: Adaptive Hash Table with Splay Tree (20 Points):

What are Adaptive Hash Tables?
Adaptive hash tables are advanced data structures that aim to combine the best attributes of hash tables and self-adjusting binary search trees to efficiently handle a mix of associative array operations. This hybrid structure uses hashing to quickly narrow down the search space and then employs a splay tree to manage collisions, which not only handles the collision resolution but also adapts to access patterns.

Role of Splay Trees in Adaptive Hash Tables
In a traditional hash table with chaining, each bucket typically contains a linked list of all the elements that hash to the same slot. In an adaptive hash table, these linked lists are replaced with splay trees. The splay trees are "adaptive" in that they rearrange themselves with each access to quickly bring frequently accessed elements to the top, thus exploiting temporal locality in the access patterns.

Unlike with open addressing, the adaptive hash table will never run out of room to insert elements. Thus, the hash table will never be resized. For particularly small hash tables, this can mean the splay trees in each bucket can grow to be quite large.

Class: adaptive_hash_map:
The adaptive_hash_map.hpp file contains the *adaptive_hash_map* template class for you to implement. The hash table is represented by a vector of *splay_tree*s. You will use your implementation of part 2 to represent each bucket.

Public methods/function to implement:
You will find the functions and class members listed within the starter codes provided in this project for you to implement. Documentations of each function and class members are also provided. Please read carefully before you get started. Any exceptions to be thrown are defined in splay_tree.hpp.

Implement the following functions:

1. ***template <typename K, typename V> adaptive_hash_map<K,V>::adaptive_hash_map():***
   template <typename K, typename V> adaptive_hash_map<K,V>::adaptive_hash_map(const size_t bucketCount):
   Initialize an empty hash table with the specified number of buckets (or 1 if *bucketCount* is omitted).
2. ***template <typename K, typename V>size_t adaptive_hash_map<K,V>::hash_code(K key) const:***
   Calculate and return the hash code of the given key, using the same hash function as in part 1.
3. ***template <typename K, typename V>void adaptive_hash_map<K,V>::insert(const K& key, std::unique_ptr<V> value):***
   Insert the key and its corresponding value in the hashtable. Throw a *duplicate_key* exception if the key is already present in the hashtable.
4. ***template <typename K, typename V>const std::unique_ptr<V>& adaptive_hash_map<K,V>::peek(const K& key):***
   Return a reference to the value corresponding to the given key. Throw a *nonexistent_key* exception if the key doesn't exist in the hashtable.

5. ***template <typename K, typename V>std::unique_ptr<V> adaptive_hash_map<K,V>::extract(const K& key):***
   Remove and return the value corresponding to the given key from the hashtable. Throw a *nonexistent_key* exception if the key doesn't exist in the hashtable.
6. ***template <typename K, typename V>size_t adaptive_hash_map<K,V>::size() const:***
   Return the number of elements in the hashtable.
7. ***template <typename K, typename V>size_t adaptive_hash_map<K,V>::bucket_count() const:***
   Return the number of buckets in the hashtable.
8. ***template <typename K, typename V>bool adaptive_hash_map<K,V>::empty() const:***
   Return true if the hashtable has zero elements, and false otherwise.

Building and Testing Your Code:

The following instructions are for Linux or Mac systems. For Windows, we recommend installing WSL as detailed in the Windows section below.

CMake:

Included with the starter code is a CMake configuration file. CMake is a cross-platform build system that generates project files based on your operating system and compiler. The general procedure for building your program is to run CMake, and then build using the generated project files or makefile. On Linux that process looks like this:

1. Navigate to your project directory, where CMakeLists.txt is located
2. Create a build directory for compiled executables:
   ```
   $ mkdir build
   ```
   Note: the $ symbol indicates the shell prompt - do not actually type $
3. Navigate to the new build directory
   ```
   $ cd build
   ```
4. Run CMake, which generates a Makefile
   ```
   $ cmake ..
   ```
   The .. indicates the parent directory where your CMake configuration is
5. Compile the code using the Makefile
   ```
   $ make
   ```

If you want to compile with debug symbols, replace step 4 with

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
```

If successful, the program will compile three executables: hash_map_app, splay_tree_app, and adaptive_hash_map_app. These programs accept input from stdin and print to stdout. You can use them to test your implementations. For details on how they work, consult the source code.

Testing:

Provided with the starter code are several sample inputs and expected outputs for tree-app and filesystem-app. To see if your implementation is correct, run the programs with the inputs, and compare the output with the expected outputs. For example:

```
$ ./hash_map_app < ../sample_tests/input/part1_test_00.txt > part1_test_00_output.txt
$ diff -qsZB part1_test_00_output.txt ../sample_tests/expected/part1_test_00.txt
```

If diff reports the files as identical, then your implementation matches the expected output.

Windows Setup:

For Windows users, we recommend using Windows Subsystem for Linux (WSL) and Visual Studio Code as your development environment. Using WSL, you can use all the same commands above to build and run on your local Windows computer. WSL can be installed through the Microsoft Store or by following the instructions here. The link also contains some customization/setup directions. We also recommend downloading Ubuntu from the Microsoft Store and using it to navigate your WSL and install packages.

You will need to install compilers and CMake in order to run and test your project. The required compilers as well as the "make" command can be installed using Ubuntu's build-essential package. This and CMake can be installed with the commands:

```
$ sudo apt update

$ sudo apt install build-essential cmake
```

You will have to enter your WSL password before installation begins to allow privileged access. You can check that the needed items were installed by running the corresponding command with the --version argument. If a version is printed, then that command was installed.

After downloading the work folder from Vocareum, you can move it into your desired directory in WSL using the WSL command line. You can access your Windows home directory from WSL using the path "/mnt/c/Users/<user>/"

To work on your code in WSL, you could use a text editor like Vim or Nano, or you could install the WSL extension in Visual Studio Code. After installing the extension, you can select the blue box at the bottom left of the Visual Studio window and select "Connect to WSL" from the dropdown window. This will allow you to open any WSL files and directories in your Visual Studio environment, and if you open a Terminal you can navigate your directories using the WSL Command Line. (Note: you may need to reinstall other extensions for WSL that you have already installed on your Windows version of VSCode.) From the Terminal, you can execute the `cmake` and `make` commands as specified above to build locally.

You should also test your code on Vocareum. After uploading your files, press the **Run** button on the top right. This will execute a script that will compile your code and run it against all of the sample test cases, and print a summary of which tests failed. You should run your code before submitting it to make sure it compiles and executes properly, since you will have a limited number of submissions.

Watch out for infinite loops! If your program enters into an infinite loop during a test, you will receive a 0 for the entire assignment. Make sure you have safeguards in your code to prevent this from happening.

Note that the sample test cases are not exhaustive and purposefully do not cover all possible cases. When you submit your code it will be tested against many more tests with more thorough cases, which will not be made available to inspect. It is your responsibility to ensure your program is correct, by devising new tests and manually inspecting their outputs.

Appendix A: Smart Pointers and Move Semantics

Introduction

Smart pointers are a key feature of C++ that enable automatic memory management, helping to ensure that memory is properly deallocated when it is no longer needed. This helps prevent memory leaks and dangling pointers—common problems in large programs. C++ offers several types of smart pointers, each with its own use case and characteristics. The three main smart pointers provided by the standard library are *std::shared_ptr<T>*, *std::unique_ptr<T>*, and *std::weak_ptr<T>.*

Characteristics of Smart Pointers

Resource Management: Smart pointers take ownership of dynamic memory allocations, ensuring that memory is released when it is no longer in use.

Type-Safety: Unlike raw pointers, smart pointers provide type safety and are less prone to errors.

Overhead: Smart pointers introduce some overhead due to their control mechanisms (like reference counting in the case of shared_ptr).

Use Cases for Each Smart Pointer

std::unique_ptr<T>: Represents unique ownership of a resource. unique_ptr is the lightest and fastest smart pointer, with no overhead for reference counting. It is used when you want a single pointer to an object that will be reclaimed when that single pointer is destroyed. A unique_ptr cannot be copied, so only one such pointer to an object may exist at a time.

std::shared_ptr<T>: Allows multiple pointers to own a single resource. The resource is only released when the last shared_ptr pointing to it is destroyed or reset. This is useful when you need to share ownership of an object across multiple parts of your program. Each shared_ptr to a resource keeps track of a reference count, which tells how many shared_ptrs refer to the same resource. When the reference count is reduced to 0, the object is released.

std::weak_ptr<T>: Designed to complement shared_ptr by providing a non-owning "weak" reference to the resource. When you want to access the resource that may be owned by one or more shared_ptrs without affecting its ownership status, you use weak_ptr. It is necessary to convert a weak_ptr to a shared_ptr to access the actual object it references, which is done by calling lock() on the weak_ptr. lock() returns a shared_ptr to the resource, increasing its reference count.

Transferring ownership of Smart Pointers

In C++, *std::move* is a standard library function that converts its argument into an *rvalue reference*. This allows the efficient transfer of resources from one object to another, a process known as move semantics. When it comes to smart pointers, *std::move* plays a crucial role in transferring ownership between smart pointer instances.

std::unique_ptr<T>: Since unique_ptr maintains unique ownership of its resource, it cannot be copied—only moved. Using *std::move* with unique_ptr transfers the ownership of the resource to another unique_ptr, leaving the original unique_ptr in a valid but unspecified state (typically null).

std::shared_ptr<T>: Although shared_ptr allows multiple ownerships over the same resource and can be copied, using *std::move* can be beneficial. It transfers the ownership without increasing the reference count, which can be more efficient than copying when the original shared_ptr is no longer needed.

std::weak_ptr<T>: weak_ptr does not own the resource, but *std::move* can be used to transfer the weak reference from one weak_ptr to another. This can be done without affecting the reference count of the shared resource.

Using *std::move* signals the intention to transfer ownership and allows the compiler to perform optimizations related to move semantics. This makes *std::move* a vital tool in modern C++ for managing resources efficiently, especially in the context of smart pointers where ownership semantics are key.

For further reading, consult the links below:

LearnCPP tutorial:
https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics/

CppCon 2016 Herb Sutter:
https://www.youtube.com/watch?v=JfmTagWcqoE

C++ Reference:
https://en.cppreference.com/w/cpp/memory