

Project 0: Implementing a Hash Table

CS 265, Big Data Systems, Spring 2017

Goal and Motivation. The goal of Project 0 is to help you refresh basic skills at designing and implementing data structures and algorithms. These skills will be essential for following the class. Project 0 is meant to be done prior to the semester or during the early weeks. If you are doing it before the semester feel free to contact the staff for any help or questions by coming to the office hours or posting in Piazza.

We expect that for most students Project 0 will take between one and three days, depending on the student's background on the above areas. If you are having serious trouble navigating Project 0 then you should reconsider the decision to take CS265. You can expect the final semester project to be much more work.

Basic Project Description. In many computer applications, it is often necessary to store a collection of key-value pairs. For example, consider a digital movie catalog. In this case, keys are movie names and values are their corresponding movie descriptions. Users of the application look up movie names and expect the program to fetch their corresponding descriptions quickly. Administrators can insert, erase, and update movie descriptions, and they want to execute these operations quickly as well. In general, any data structure that stores a collection of key-value pairs is called an associative array. It implements the following interface:

```
put(key , value );  
get(key , values_array );  
erase(key );
```

The *put* command stores a new key-value pair in the associative array. The *get* command retrieves all key-value pairs with a matching key and stores their values in *values_array*. The *erase* command removes all key-value pairs with a matching key.

An associative array can be implemented in many different ways, and the implementation determines the time it takes the three different commands (i.e., *put*, *get* and *erase*) to execute. In project 0, you will implement an associative array as a hash table. Relative to other data structures that can be used to implement an associative array (e.g., sorted array, linked list, etc), the distinguishing characteristic of a hash table is that the time to put, get or erase a key-value pair does not increase with respect to the number of key-value pairs inserted so far.

Programming Language. For your implementation, you will be using the C programming language. C is considered to be a low-level programming language. Low-level programming languages give the programmer fine control over how the program is executed in hardware. This enables the creation of very efficient implementations. However, it also opens up various pitfalls that do not exist in higher-level programming languages (e.g., Java or Python). A brief tutorial on C is located here¹.

¹C tutorial: <http://www.cprogramming.com/tutorial/c-tutorial.html>

Instructions. Below we provide a guideline for implementing a hash table and its basic operations, e.g., putting, getting, and erasing. We also provide a description of common implementation problems that may come up. In addition, we provide test scripts that will help you test your hash table for correctness and performance.

Getting Started: Skeleton code for your implementation is provided in the repository here². To access the code, install the version control system Git, for which an installation guide is located here³ Through the command line, you can then clone the repository onto your own machine by running the command:

```
git clone https://bitbucket.org/HarvardDASlab/project0-hashtable.git
```

More details on how to run the code and get started are in the README file in the repository.

1 Basic Implementation

A hash table uses a so-called hash function h to map from keys onto random yet deterministic slots in an array. Thus, a key-value entry with key K is stored at index $h(K)$ of the array, and looking up a key K requires searching in slot $h(K)$ for a key-value pair with key K . You may use a hash function of your choosing (many common choices can be found online, though be sure to cite your source). A good starting point is modular hashing, which involves taking the key modulo the size of the array.

A general concern is that the hash function may map multiple key-value pairs onto the same slot in the array. These phenomena are called collisions. To handle collisions, each slot in the array should point to a linked list. All key-value entries mapped by the hash function onto the same slot should be inserted to the same linked list. A lookup should quickly find the appropriate linked list and then traverse it to find the target key. This is illustrated in Figure 1. Note that the amount of time to find or insert an entry with key K is proportional to the length of the linked list at slot $h(K)$ of the array.

Ideally, the hash function should uniformly distribute keys across the slots of the array so that the sizes of the linked lists are approximately equal. However, if the hash function results in skew, some linked lists may grow disproportionately larger than others. In particular, suppose the workload has the characteristic that most keys are divisible by some common factor of N , the number of slots in the array. For example, in the case of Figure 1, N is 10, so suppose most keys are divisible by 5, which is a common factor of 10. If so, then most key-value pairs would reside in the same small set of linked lists: the ones at slots 0 and 5. This would result in poor performance. To minimize the chance of this happening, a common strategy is to set N to be a prime number. The intuition is that this minimizes the number of factors that can be common to the keys in the workload and N .

Your implementation should allow the existence of multiple key-value pairs with the same key. Thus, the *put* command simply appends an entry to the appropriate linked list, the *get* command returns all entries with a matching key, and the *erase* command removes all entries with a matching key. You should implement these commands inside the `hash_table.c` file, which provides further instructions.

Implementation Details. A node of the linked list should be implemented as a C structure⁴. This structure contains one key-value entry as well as a pointer⁵ to the next node in the linked list. A slot in the array contains a pointer to the first node in each linked list. For the terminal node in the linked list, the pointer is set to NULL. When a new key-value pair is inserted into a linked list, a new node should be allocated using a call to `malloc` (the C

²Project 0 repository: <https://bitbucket.org/HarvardDASlab/project0-hashtable.git>

³Git installation guide: <https://confluence.atlassian.com/bitbucket/set-up-git-744723531.html>

⁴A tutorial on C structures can be found here: <http://www.cprogramming.com/tutorial/c/lesson7.html>

⁵A tutorial on C pointers can be found here: <http://www.cprogramming.com/tutorial/c/lesson6.html>

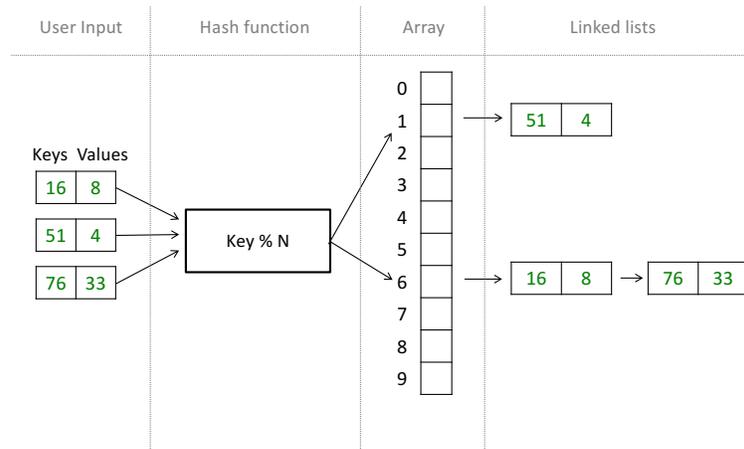


Figure 1: Illustration of a Hash Table with an array comprising 10 slots. The first key-value pair is mapped to slot 6 because the key is 16 and $16 \bmod 10$ is 6. The second key-value pair is mapped to slot 1 because the key is 51 and $51 \bmod 10$ is 1. The third key-value pair is mapped to slot 6 because the key is 76 and $76 \bmod 10$ is 6. Note that since the first and third key-value pairs map into the same slot in the array, they are chained within the same linked list.

memory allocator), and the pointer of the terminal node should be set to point at the new node. Thus, a new node becomes the new terminal node.

Common Problem. In the interest of speed, the programming language C does not perform bound-checking on arrays during runtime. This means that if a program accesses index $X + 1$ of an array that only contains X slots, the program will not necessarily crash (unless you are lucky). Instead, the access will return the wrong data and thereby cause the program to behave unpredictably. Even worse, if the program updates an out-of-bound index, it may corrupt live data that belongs to other data structures in the program. This is known as a buffer overflow. Such errors are difficult to debug, and they should be avoided through vigilance while programming. However, there are tools that exist to profile a program to detect out-of-bound memory accesses. Valgrind is an example of such a tool. If your program crashes in an obscure way, we recommend running it with Valgrind to identify the problem. An example of how to detect an out-of-bound accesses with Valgrind is located here⁶.

Debugging. While implementing a data structure or algorithm, it is common to make small logical mistakes that may cause the program to return incorrect results or even crash. If either of these things happen, you can use a debugger to try to identify the source of the problem. A debugger is a tool that allows stepping through the code, line by line, while showing the values of all the different variables at a given point in the execution. Thus, it can be used to track down the source of an error. GDB is an example of a debugger, and a tutorial on how to use it is located here: <https://kb.iu.edu/d/aqsj>.

Compilation. The Gnu C compiler, GCC, gives various options for how to compile a program. One important compiler option is the level of optimization. By default, the compiler's main goal is to minimize compilation time, and so it does not try to improve the performance of the code through compiler optimizations. However, we can tell the compiler using the `-O3` flag to perform as much optimization as possible. While this improves the performance of the code, it increases compilation time. It may also change the order in which lines of the program are executed, which can make debugging confusing.

⁶Out-of-bound error detection with Valgrind: <https://www.cs.swarthmore.edu/~newhall/unixhelp/purify.html>

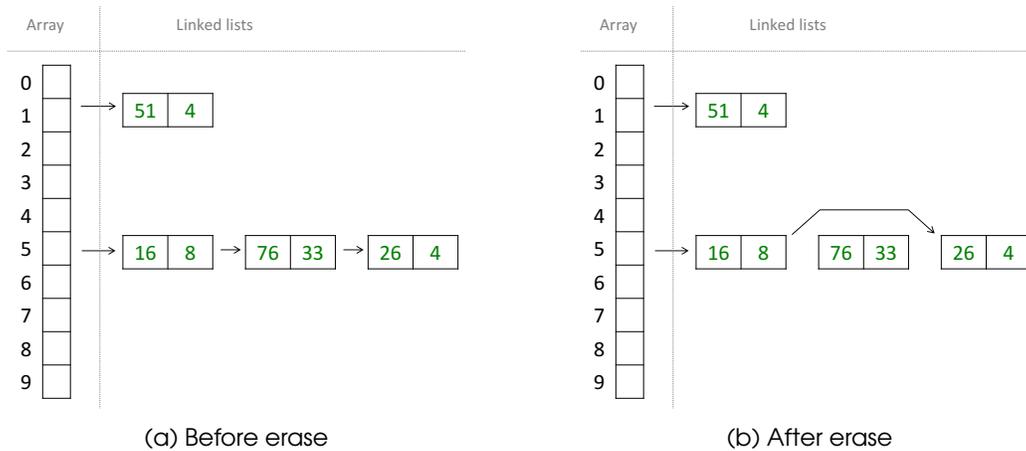


Figure 2: Illustration of before and after erasing a key-value entry with key 76. The erase involves changing the pointer from the predecessor node to point to the successor node. This leads to a memory leak because the erased node still exists and is now not accessible by the application. To avoid a memory leak, we must inform the memory allocator that the node is no longer used.

In the repository, there is a file called `Makefile`, which is a script for how the code should be compiled (instructions for how to use the `Makefile` are in the `README` file in the repository). The compilation command for the benchmark program uses the `-O3` flag, because we want it to run as fast as possible. The other two programs, `main` and `test`, are compiled using the `-O0` option. This option tells the compiler to perform no optimization, and so we retain debugging ability. Another relevant flag is `-Og`, which performs as much optimization as possible while retaining debugging ability. You may experiment with these and other flags throughout the project.

2 Erasing Key-Value Pairs

Now implement the $erase(K)$ function, which removes the key-value entry with key K from the hash table if such a key-value pair exists. The erase function should initially identify the node in the linked list that contains the key-value pair. It should then remove this node from the linked list by changing the preceding node to point to the target node's successor. This is illustrated in Figure 2.

Common Problem: The programming language C does not automatically delete unused data from memory. Thus, when the node is removed from the linked list, the memory is not automatically freed even if it is no longer reachable from the code. This phenomenon is known as a memory leak. Figure 2 illustrated a memory leak resulting from an erase. Memory leaks are undesirable because the memory taken up by inaccessible objects is no longer available to the application. If the amount of non-available memory accumulates over time (e.g. after many erases), the system will run out of available memory. This may cause the program to crash. Thus, it is imperative to delete data from memory as it becomes unused. To this end, use the `free` command of C to inform the memory allocator when a node is removed from the linked list⁷. This informs the memory allocator that the space taken up by the removed node is now available for future memory allocations.

Valgrind, the tool already mentioned in the previous section, can be used to profile a

⁷A tutorial on the `free` command is given here: http://www.tutorialspoint.com/c_standard_library/c_function_free.htm

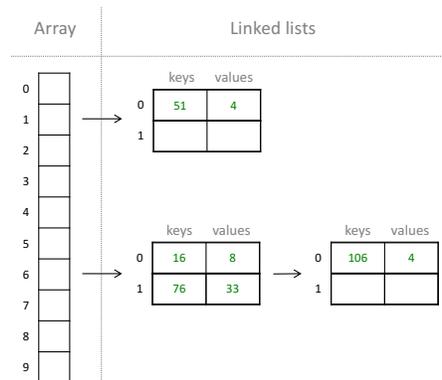


Figure 3: Using a linked list of arrays.

program to identify memory leaks. A tutorial on using Valgrind for this purpose is located here⁸. It is a good idea to regularly check your program with valgrind to verify that there are no memory leaks.

3 Modern Hardware Considerations

In modern computers, fetching data from memory into the processor takes a long time, typically much longer than actually using this data when it is already in the processor. In the context of your hash table implementation, traversing a linked list may involve multiple memory accesses, one for each node. These memory accesses become increasingly expensive as the length of linked lists grow.

To reduce this cost, we can take advantage of how modern computers are fast at sequential access. We can do so by placing multiple key-value pairs in each node of the linked list. This is shown in Figure 3, where each node of the linked list contains 2 key-value pairs. The effect of doing so is that the processor can fetch multiple key-value pairs through a single sequential access, reading an entire node at the same time. This reduces the overall number of memory accesses per traversal of a linked list, and so performance improves.

On the other hand, you should not set the number of key-value pairs that fit into a node too large. The reason is that the amount of data that the processor can store in each of its cache lines is small, typically 64 or 128 bytes. If the size of a node exceeds this amount, then reading each node would involve more than one memory access. This would reduce the benefit of storing multiple key-value pairs in each node. In your implementation, you can experiment to find the node size that results in the best performance.

4 Running out of Capacity

As more keys are added to the structure, the average length of each linked list grows. Thus, the overheads of inserting or retrieving a key-value pair begin to increase because it takes a longer time to traverse the appropriate linked list. To avoid such a degradation in performance, you should enlarge the size of the array in proportion to the number of key-value pair inserted so far.

⁸Tutorial on valgrind: <http://valgrind.org/docs/manual/quick-start.html>

Implementation Details: First, implement bookkeeping within the hash table to keep track of the number of key-value pairs P . Whenever the number of key-value pairs grows beyond a threshold value, the size of the array should be enlarged. The threshold value is typically defined as $\frac{Q \cdot P}{N}$, where P is the number of entries in the hash table, N is the number of slots in the array, and Q is a tuning parameter. You may experiment with different values of Q to find the one that results in the best performance.

To enlarge the hash table, allocate a new array that is larger than the original. Note that the number of slots in the new array should also be a prime number. Moreover, the size of the new array should be approximately double the size of the original array. Then iterate through every key-value pair of the original hash table and insert them into the new hash table. Finally, deallocate the original hash table to ensure no memory leaks have taken place.

5 Experimentation

We provide you with two scripts that automatically test the correctness and performance of your implementation. The correctness test ensures that your implementations of the put, get, and erase commands return correct results. For example, after we insert a key-value pair, we should be able to retrieve the value associated with the key. Similarly, after erasing a key-value pair, we should no longer be able to retrieve a value that was associated with the key before.

The performance test measures the amount of time that it takes to insert 50 million key-value pairs into your hash table. You can use the performance test to iteratively improve your hash table, e.g., to lower response time for queries and inserting new data. You can also use it to experiment with alternative designs and see their effect. For the script to work you will need to support the following API.

```
void init(hashtable** ht);  
void put(hashtable* ht, keyType key, valueType value);  
int get(hashtable* ht, keyType key, keyType *values, int num_values);  
void erase(hashtable* ht, keyType key);
```

Note that this API is expressed in terms of generic data types for keys and values. This is done to emphasize that much of the hash table infrastructure that you will create is applicable to any data types. However, for this project you may assume that `keyType` and `valType` are always integers.

View the README file in the repository for further details on how to implement this interface and how to run the correctness and performance tests.