Project 2 – Design Description

Tony Shen

For this project, I utilized only the two test programs: openhttest.cpp and orderedhttest.cpp. I believed that this would make the code design as simple as possible, avoiding the possibility of duplicate definitions as both test files could share some content sourced from a header file. The different types of collision handling used by the two files however made it an easier decision to include everything in two files. For example, it can clearly be seen that openhttest.cpp utilizes a class called DoubleHash. On the other hand, orderedhttest.cpp uses a struct HashTableEntry which represents our node, followed by the class HashMapTable.

Starting with openhttest.cpp, we first see a global variable MAX set to 0. This represents the size of the hash table, which is later set in function n. The class DoubleHash contains pointers to keys and names, which are defined in the constructor to represent arrays for each piece of information. The int current_size is incremented each time a new number is added to the table, allowing for us to check if the table is full through a comparison to MAX (seen later in function bool isFull). Also inside the constructor we initialize each of the member variables, going through an array to initialize keys and names. Since all the numbers must follow the correct North American phone number format, the initialization to -1 will never coincide with an actual phone number. We can use this to check if a position in our table is empty, allowing us to put an element in that spot.

The next function is bool found, which simply returns whether or not a key exists in the table (true) or not (false). The hash functions h1 and h2 come next, the ladder representing a simple iteration where 1 is added if the result is even. Both return integer values, representing the desired array index once an open spot is found.

We now begin to look at the member functions (all void) representing the commands outlined in the project description. First is n, which sets MAX to the passed in integer value m. Our next function i takes the parameters k and the caller (which have been separated in the main function) and inserts them into their respective arrays. First, we check if the function is full and if the key already exists, using our previously defined functions. Next, we use the two hash functions and iterate the value of probe until it represents an index that is empty (once keys[probe] = -1, the initial value). The values are then saved into the array, and current_size is incremented. It is clear that this simple iteration of h1 and h2 will provide a constant expected average runtime, as some instances will have no need for the while loop while others may need to iterate numerous times to find a spot.

The member function s uses a Boolean flag to determine our cout. We simply step through our keys array and if the desired key is found, the flag is set to true. If the entire array is stepped through and nothing was found, we simply output failure. This gives a constant expected average runtime, depending on how quickly k can be found, as we can exit the for loop once found is true. The function d involves a very similar implementation, with the inclusion of resetting the values of keys and names to their default values. Once again, a constant runtime is expected, as we exit the for loop once the desired key is found and deleted.

Lastly, the destructor simply deletes the arrays and clears the memory.

Now looking at orderedhttest.cpp, we can immediately see a few clear differences in our implementation choices. The struct HashTableEntry represents our node in the linked list, with member variables k, caller, next, and previous pointers. This node is initialized with values passed through, and the next and previous pointers are set to NULL as there are not yet any collisions.

Next, we examine the main class of the function, HashMapTable. We first define hash as a pointer to the hash array, which then points to the start of the linked list. We initialize the array as we do not yet expect any collisions, and with that, any need for the linked list part. We then have the same functions h1 and isFull as openhttest.cpp. The found function is slightly different however, as we must now check through the keys in the linked list (involved in a collision) as well. We first define a variable to represent the index given by the hash function (under which the key should exist), then use a to step through the linked list (a->n) until we either find (true) or don't find (false) the key.

We now look at our command functions (again all void), starting with n, which accomplishes the same goals as in openhttest.cpp. The i function is considerably more complicated, so we will slowly walk through. The same checks for a full table and the key already existing happen first. Next, we use the hash function h1 to generate our index hash_k. We again use a to step through the linked list, involving p to temporarily store the value of a. Once a = NULL, we know that we have reached the end of the list and that we can insert the element. At this point we create a new HashTableEntry with the values passed into i. Next, we check the cases of a being the first node in the linked list and update the pointer accordingly. Finally, our value of current_size is incremented. This function completes in constant average expected time since the only factor affecting runtime is the length of the linked list, which will average out through many runs.

Next, we look at our search function s. Once again, we use a to step through the linked list starting at the index hash_k generated by h1. Our Boolean flag allows us to exit the while loop early once we find the desired number, and also generate the appropriate cout. For the same reasons as our s function in openhttest.cpp, this function completes in average constant time. Looking at function d now, we first use !found to determine if the key even exists. The same steps are then followed (hash_k, a, p) to step through the list and delete our key. If we end up with the last pointer (p) not being equal to NULL, we connect the disconnected pointers (from deleting the node) through the code: p->n = a->n. We then reset our key value and set a = NULL, and delete a. This operates in constant average expected runtime, since runtime is only a matter of the size of the linked list (which averages out).

The last member function is p, unique to orderedhttest.cpp. This function simply uses a (at index i in hash) to step through a linked list and print each element.

Lastly, the destructor simply deletes the array hash from memory.

Now that we have gone through each member function of both test files, we will go through the main function, which is common to both (minus a call to p for orderedhttest.cpp). We first initialize hash, used to access all our member functions and variables. The user then simply enters a command and is taken to the correct if block. The only block of note is i, which uses getline to read the entire string (eg. 1234567890;tony shen), then eliminates the newline character using pop.back. We then define the delimiter ; to separate the number and name using substring. We convert k_str to k, an integer. Finally we erase the 10-digit number from the front of the entire string to get the name.