Thomas Rafeld, M12808058

Ryan Winterhalter, M12727389

Lab 9 Report

**Objectives:**

The objectives of this module were to be able to describe a hash table, understand the approaches to deal with collisions, and build a hash table. The first objective of being able to describe hash tables is important for our success in this class, as being able to describe hash tables is a key part to understanding how they work. In order to describe a hash table, we must understand hash tables and their functionality, which (considering that this lab consisted of making and testing different hash tables) is incredibly important for our success in this class. As hash tables are a common, important data structure, understanding them and being able to describe them is also incredibly important for a career in engineering, as there may be projects on which we work where a hash table implementation is used (or would be the best fit for the implementation). Understanding hash tables and how they work, as well as being able to describe them to others, would be very important for our success on this type of project; we would have to use our knowledge of hash tables and their functionality to implement a solution to our problem and may have to work on a team in doing so, which would require us to be able to understand hash tables enough to be able to describe them. The second objective of being able to understand the approaches to deal with collisions is important for our success in this class because collisions are a major problem with hash tables that need to be handled for their proper functionality. If we are to be able to complete Lab 9 and properly implement different types of hash tables, it is crucial for us to be able to understand how to properly deal with collisions and understand the different approaches on how to deal with them. This understanding is also important for a career in engineering, as different hash tables may be a better fit for the implementation of certain problems (depending on how much you add/remove, how large your data is, how much you search, etc.). Understanding the different types of hash tables and their different approaches on dealing with collisions would allow us to excel in the workplace by allowing us to properly implement the best solution for the type of problem we are facing. The third objective of being able to build a hash table is incredibly important for our success in this class because it is the entire objective of the Lab 9 assignment; if we are to properly complete Lab 9, we should be able to build hash tables. This objective is also incredibly important for a career in engineering, as being able to build a functioning hash table would be critical to our success on certain projects that may require hash table implementations in the solution. Being able to understand hash tables and how they work is one thing, but a career in engineering requires you to -- regardless of your understanding -- create a functioning product. As a result, being able to build a functioning hash table is invaluable to a career in engineering in the instance that your project requires a hash table implementation. Considering hash tables are a common data structure, it is important that we understand how to build them for both our success in this class and a career in engineering.

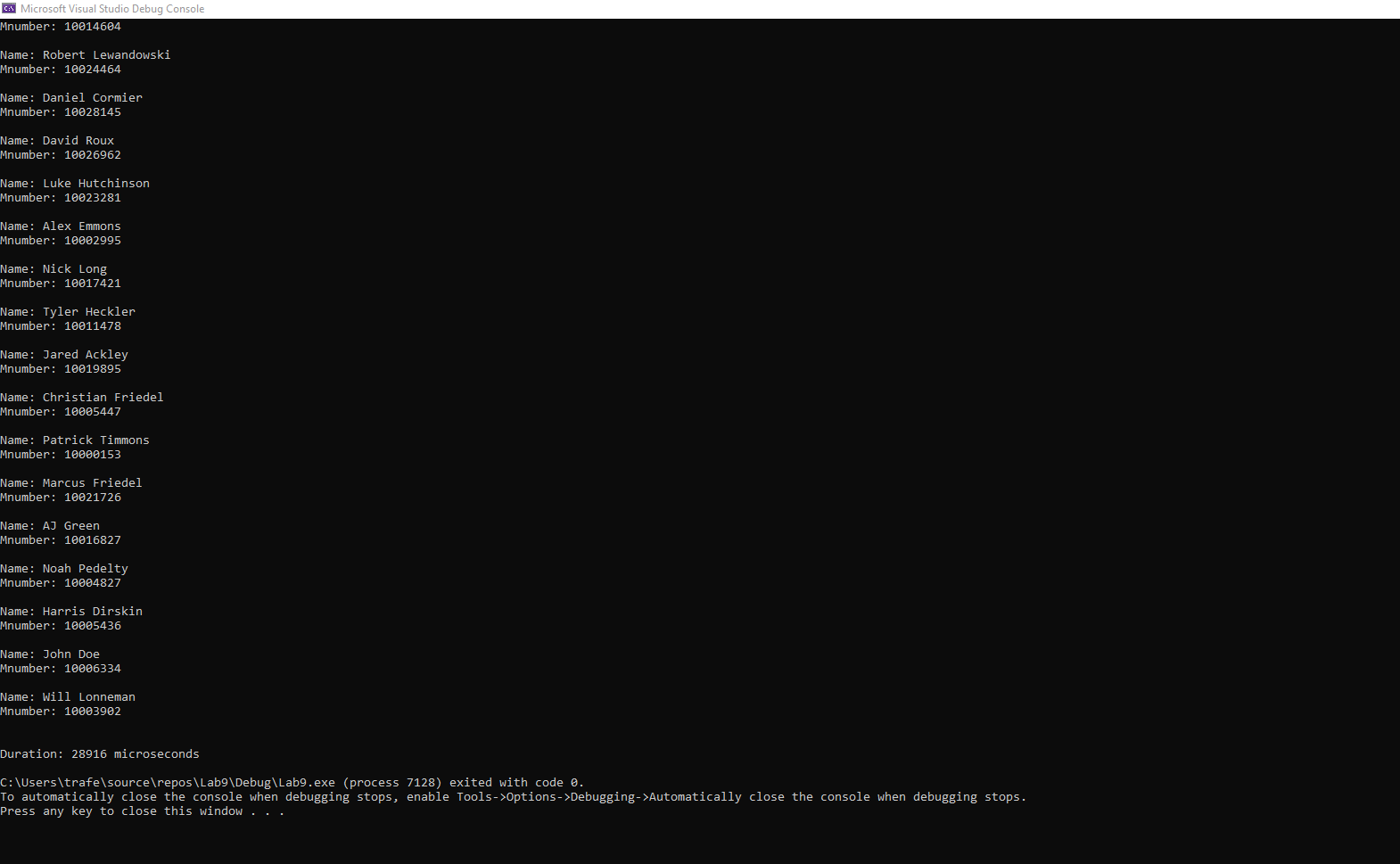
**Task 3:  
**

Figure 1: Screenshot of testing for Lab9 task 3, Hash Table implementation

**Task 4:**

****

Figure 2: Screenshot of testing for the Lab 9, Task 4 chained hash table implementation

**Task 5:**

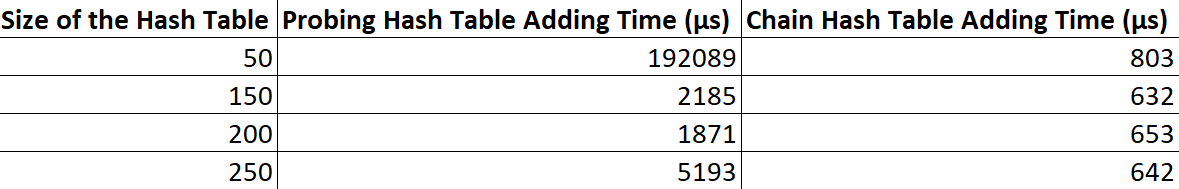
****

Table 1: Adding times for the different hash tables of their different sizes, respectively

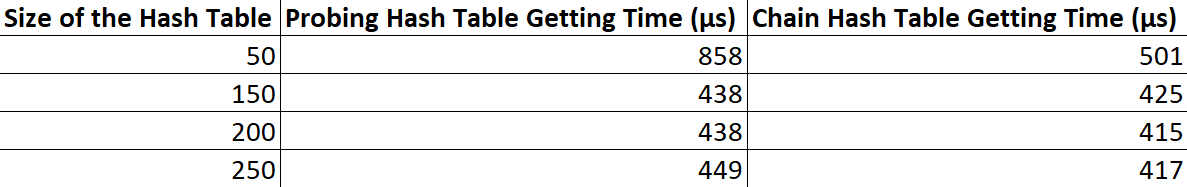


Table 2: Getting times for the different hash tables of their different sizes, respectively

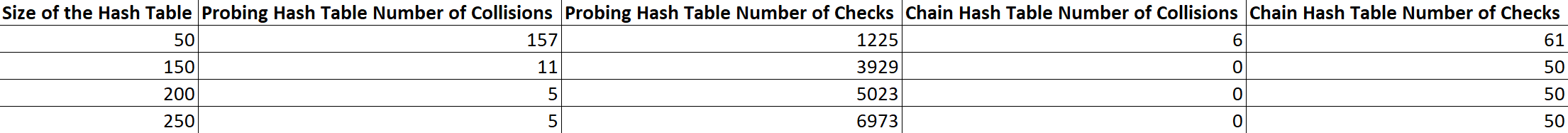
****

Table 3: Collision (adding) and check (getting) counts for the different hash tables of their different sizes, respectively

**Task 2 Modifications:**

For Task 2 we had to add a class that would store the value for a student name and their M-number. Then we had to change the add function to be able to take a pointer value to both the M-number and the name to be added to the hash table. The hash function was also modified to take the M-number value and use that to determine the index of the name and number. The Remove and GetItem functions were modified to take the M-number and find the student using just that.

**Task 3 Modifications:**

For Task 3, we had to modify our main code for testing our header from task 2 to allow us to import student data (name and M-number) from a text file that would then be inserted into the hash table we built in Task 2. We also had to add our calculation of the run-time for adding all of the different students into the hash table by including the ‘chrono’ library.

**Task 5 Results:**

The adding times for our probing hash table generally decreased as our hash table size increased. This matches our expectations, as we predicted that a smaller array size for the hash table would result in a lot more collision that would need to be dealt with, which would cause a lot more time for the program to execute than for a program with a larger array size for its hash table, which would have fewer collisions that would need to be handled. For our chained hash table, we saw that our adding times also generally decreased as our hash table size increased. This matches our expectations, as we predicted that a smaller array size for the hash table would result in more collisions that would need to be dealt with, similar to that of the probing hash table. This would cause more time for the program to execute than for a program with a larger array size for its hash table, which would have fewer collisions that would need to be handled. However, the add times for our chain hash table were shorter than those for our probing hash table. We expected this, especially as the hash table array size is small, because the handling of collisions with the chain hash table would simply add the collided student to the end of the linked list at that index, whereas the probing hash table would have to iterate through the array until it would find an open spot for the collided student to go.

The getting times for our probing hash table generally decreased as the size of our hash table’s array increased. This did not match our expectations for our probing hash table, as we expected the increased array size to cause a slight increase in our get times, as it would take slightly longer to iterate through the larger array to find if a value is there and compare it with what we are trying to find. This is an area for further investigation. The getting times for our chained hash table were mostly constant as the size of our hash table’s array increased. This matched our expectations as we believed that, even with a larger array size, the get time would remain fairly constant due to our particular implementation, which starts with finding the key of what we are trying to find and immediately going to that index. With the vast majority of the linked list sizes being fairly small, we expected the times to iterate through the linked lists to find if the item was present in the list or not, which we believed would be fairly consistent even as the array size changed. Our getting times for our probing hash table were only slightly longer than that of our chained hash table, which fits our expectations as we believed our specific implementation of the GetItem member function of the chained linked list would be slightly faster because we would find the key immediately and then go straight to the index to look for the item, whereas with the other implementation we had to iterate through the entire list until the item was found.

The number of collisions for the probing hash table did, in fact, drop when our array size increased, as we predicted. We also see that our number of checks for the probing hash table increased as our array size increased, which matches with our prediction. This is why we are unsure of why the getting times generally decreased as our array size increased. This is an area we would like to investigate further. The number of collisions for the chained hash table decreased from 5 to 0 as our array size increased. This matches with what we expected, as the larger array size would cause for a smaller number of collisions to take place, hence explaining why the add times decreased as the array size increased (much like that for probing hash tables). The number of checks for the chained hash table remained generally constant as our array size increased, which matches with our prediction. As the number of collisions reduces so much that almost each data item in the chained hash table is in its own index, the get times will remain generally constant (and might even decrease a little bit, as our data proves, because there will ultimately be some reduction in linked list sizes as the array size increases and the number of collisions decreases). Similar to what we predicted, our probing hash tables number of collisions were greater than those for our chained hash table because the chained hash table only has to collide once with a filled index to be added to the end of the linked list at that index, whereas the probing hash table would have to keep iterating until an open index is found for the new item. Additionally, our predictions matched with our results in that the number of checks for our probing hash table was greater than that of our chained hash table, which we believe is due to our implementation for the chained hash table’s GetItem function being slightly different than the GetItem function for the probing hash table; the GetItem function for the probing hash table has to iterate through the entire array until the value is found, whereas the GetItem function for the chained hash table goes straight the index where that input would have been stored by keying the value that we are looking for and then searching that individual linked list for the value to see if it is in the chained hash table.

**Group Contributions:**

The lab was worked on together by both Ryan and Thomas while on a call together in Microsoft Teams. We worked on the header file for task 1 together and ended up using Ryan’s header file for that. The main cpp written for Task 3 was written by Thomas and then modified later to be used with task 4 and task 5 by Ryan. For the final grade each member of the group should receive 100 percent of the grade as we feel that we both evenly contributed to the lab and worked together for almost the whole time it was being worked on.