Micah Cook

Prof. Zagrodzki

CSCI 2270-103

8 December 2019

Hash Table Project Report

Purpose:

The purpose of this project is to create multiple hash table implementations with different collision resolution methods. To evaluate each hash table, large datasets will be parsed into each hash table and insert, search, and deletion operations will occur at different load factors to evaluate the performance and efficiency of each hash table.

Procedure:

The data structures that will be used in each unique hash table will be linked lists, binary search trees, and arrays with linear probing or cuckoo hashing.

In the linked list hash table, collision resolution will be performed using linked lists in combination with an array for the hash table. If a collision occurs, the inserted value will be stored at the end of a linked list at the calculated index in the hash table. So, the hash table is an array with each array element pointing to a linked list of values.

In the binary search tree hash table, collision resolution will be performed using binary search trees in combination with an array for the hash table. If a collision occurs, the inserted value will be inserted at the end of a subtree at a specific location in the binary search tree depending on the value of the number that will be inserted. So, the hash table is an array with each array element pointing to the root of a binary search tree.

In the linear probing hash table, collision resolution will be performed using linear probing with a step size of 1. If a collision occurs, the insertion function will scan each element ahead of the calculated index until an empty slot in the array has been found, then the function will insert the value there. So, the hash table is just an array, but each value inserted will have its own location in the hash table without chaining together values like in the BST or linked list implementations.

In the cuckoo hashing table, collision resolution will be performed using two hash functions, two hash table arrays, and rehashing. If a collision occurs, the insertion function will calculate the index for the number to be inserted by using the first hash function and attempt to insert it at that calculated index in the first hash table. If a collision occurs here, the insertion function will insert the number at the first hash function location in the first hash table, take the value that was previously there, and recalculate an index for this “displaced” number with the second hash function and attempt to insert it at that index in the second hash table. If a collision occurs here, then the insertion function will take the “displaced” number and insert it at the second hash function location in the second hash table, take the value that was previously there, and recalculate an index for this new “displaced” number with the first hash function and attempt to insert it at that index in the first hash table. This process of displacing values and constantly attempting to re-insert these values into both hash tables can potentially cause an infinite loop of attempting to insert values but constantly getting collisions. In this case, the program will then “rehash” or rebuild the hash tables by increasing both hash tables sizes and this results in new hash functions that will calculate new indexes for numbers. With these new hash functions, the program will re-insert all values into the new hash tables with these new hash functions and then the insertion process will go on like normal.

The metric (unit of time) that will be used to measure the runtime differences between the implementations will be in seconds, but all values of seconds seen in the graphs will be 0.000001 of a second, or a microsecond. So for example, seeing that an average of 100 insertions at a load factor of 0.9 will take approximately 0.00001 seconds translates to 10 microseconds.

Results:

\*\*In my Cuckoo Hashing table, it only works on small datasets that fit within the default tableSize of 10009. Once rehashing starts to occur (which usually occurs around >= 0.5 loading factor), two things can happen: 1. The program hangs and gets stuck in a cycle of re-inserting values or 2. The program terminates successfully but the table contains a lot of garbage values. Because of this, graphs of cuckoo hashing cannot be provided as my cuckoo hashing table does not fully work.

\*Graphs are provided in the other Microsoft word document in this folder.

The mean and standard deviation of the runtimes at each load factor for each implementation is as followed:

* Linear Probing:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Load Factor | 0.1 | 0.2 | 0.5 | 0.7 | 0.9 | 1.0 |
| Mean Time | 0 | 0 | 0 | 0 | 0.0000016225 | 0.00000739625 |
| Standard Deviation | 0 | 0 | 0 | 0 | 0.0000036293 | 0.0000042752 |

* Linked Lists:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Load Factor | 0.1 | 0.2 | 0.5 | 0.7 | 0.9 | 1.0 |
| Mean Time | 0 | 0 | 0 | 0 | 0 | 0.0000049358 |
| Standard Deviation | 0 | 0 | 0 | 0 | 0 | 0.0000049377 |

* Binary Search Trees:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Load Factor | 0.1 | 0.2 | 0.5 | 0.7 | 0.9 | 1.0 |
| Mean Time | 0 | 0 | 0 | 0 | 0.000005075 | 0.000004905 |
| Standard Deviation | 0 | 0 | 0 | 0 | 0.000004916 | 0.000004907 |

Each implementation performs as expected on the data sets because as load factor increased, execution time also increased. The reason why this happened is because as more values are inserted, more records inhabit the hash tables and when more insertions occur, the chance for collisions becomes greater and greater. When a collision occurs in each of these implementations, the perfect O(1) complexity for insertions, searches, and deletions in hash tables starts to approach an O(n) complexity, so naturally execution times will increase.

Cuckoo hashing might not be the best method of hashing for these large data sets because as more values are inserted into the cuckoo hash table, more collisions and thus more cycles can occur, so the chances for rehashing the table increases. Rehashing the hash table is a very expensive process for the program to do as rehashing is essentially just re-inserting every value into a hash table with a larger size than the previous one, so when large data sets are to be inserted, rehashing must occur quite often and can make the program very inefficient.