Justin Murillo

Hashing algorithm write up CSCI 2270

**Purpose:**

The purpose of this project was to evaluate and compare four different hashing implementations. These comparisons primarily focus on how each hashing implementation performs when doing certain common functions and the different possible load orders that these implementations will have.

**Procedure:**

I have chosen to use Linked List Chaining, Binary Tree Chaining, Linear Probing, and Cuckoo hashing algorithms because each of them are unique to how they deal with collisions and they also apply what I have learned in my Data Structures class. Two of them use chaining as collision resolution, and the other two use open addressing. Both types methods are commonly used for hash functions. Linked List Chaining works by resolving each collision by creating a hash table of node pointers. These node pointers will grow to form linked lists when a collision occurs. When a collision does occur, the table at that index which stores the original key points to the new key node. So, when we search for that key in the hash table, we first need to iterate through that linked list starting at that index to find that key.

Binary Tree Chaining uses a similar technique, where a collision is resolved by using pointers to point to another key at that index in the hash table. The only difference being how the chain is traversed. It uses recursion to iterate through the tree, which can be faster to find a certain key than a regular linked list. Average case of traversal for a BBT is O(log n) compared to O(n) of a linked list.

Linear Probing does not use pointers to resolve collisions. Instead it uses open addressing. What this means is that when a collision occurs, instead of having something point to the new key from that index in the hash table, the key is just shifted down one on the index so that the new key and original key are placed next to each other. If there is already something occupying that space next to the original key, keep shifting down until a vacant spot is found.

Cuckoo hashing is also another hash function that uses open addressing. But instead of using one hash table, it uses two, and it also uses two hashing functions. When a collision occurs when inserting a key into hash table 1: the new key is ran through the second hashing function. It is then placed into the second hash table. If a collision occurs when inserting a key into the second hash table, the original key at that second hash table is removed and the new key takes its place. The removed key is then placed somewhere on the first hash table. If that place on the first hash table is occupied, the key at that location is displaced and so on and so on until a free spot is located. It is possible for this kind of insertion loop to fail when no vacant slot can be found on either hash table. When this kind of loop repeats infinitely, the hash table needs to be rehashed, usually meaning that the size of both tables needs to be expanded to make room for more keys and typically increased in such a way that the size of each table is double the original size.

For this procedure, I will be using the ctime library in C++ to time how long it takes for each type of operations to be executed under each implementation of hash function. The three different operations will be Delete(), Lookup(), and Insert() respectively. These operations will be performed under different load factors on each hash function and will be performed 100 times with the mean time to complete all 100 operations will be calculated and recorded. This is what the data and results look like for this performance evaluation.

**Results:**

AVERAGE RESULTS FOR INSERTION, DELETION, AND LOOKUP

|  |  |  |  |
| --- | --- | --- | --- |
| **Linked List Chaining** | INSERTION | DELETION | LOOKUP |
| Load Factor 0.1 | 0.0006 | 0.0012 | 0.0034 |
| Load Factor 0.2 | 0.00063 | 0.0037 | 0.001 |
| Load Factor 0.5 | 0.0007 | 0.0044 | 0.0032 |
| Load Factor 0.7 | 0.0012 | 0.0042 | 0.0034 |
| Load Factor 0.9 | 0.0012 | 0.0056 | 0.004 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Linear Probing** | INSERTION | DELETION | LOOKUP |
| Load Factor 0.1 | 0.0043 | 0.00034 | 0.00 |
| Load Factor 0.2 | 0.00395 | 0.0020 | 0.0025 |
| Load Factor 0.5 | 0.0062 | 0.0025 | 0.0038 |
| Load Factor 0.7 | 0.0041 | 0.0022 | 0.00325 |
| Load Factor 0.9 | 0.0054 | 0.00265 | 0.00356 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Tree Chaining** | INSERTION | DELETION | LOOKUP |
| Load Factor 0.1 | 0.0013 | 0.002 | 0.0015 |
| Load Factor 0.2 | 0.0014 | 0.0021 | 0.002 |
| Load Factor 0.5 | 0.0021 | 0.0026 | 0.0014 |
| Load Factor 0.7 | 0.0032 | 0.00235 | 0.001 |
| Load Factor 0.9 | 0.004 | 0.0028 | 0.0015 |

|  |  |  |  |
| --- | --- | --- | --- |
| **Cuckoo** | INSERTION | DELETION | LOOKUP |
| Load Factor 0.1 | NA | NA | NA |
| Load Factor 0.2 | NA | NA | NA |
| Load Factor 0.5 | NA | NA | NA |
| Load Factor 0.7 | NA | NA | NA |
| Load Factor 0.9 | NA | NA | NA |

I personally am not confident with the results of this evaluation. When trying to implement the ctime library on my computer, the resulting output only rounded to the nearest thousandth of a second, which for this kind of project is not acceptable since many of the operations can be completed in the span of microseconds. This sometime lead to the timing of operations to be ‘0.00’ or ‘0.01’ seconds. Obviously, this is not a high enough level of accuracy for a rigorous performance evaluation thus leading me to not be confident with my results. I was also having much difficulty with getting my Cuckoo functions to run properly. I didn’t know how to implement the more complex forms of the function, such as the rehashing and how to check for an infinite loop. Because of this, no measurements could be recorded.