1 Background

In this project, we will be learning about hash tables. There are several aspects to consider when creating a hash table, including

1. How to handle collisions
2. How to resize the hash table

In this project we will be exploring different design choices concerning these areas, and determining when one design might be better than another.

1.1 Collision Strategy

Remember that there are several ways in which we can design our hash table to handle collisions, but in general the two main categories are **Chaining** and **Open Addressing**. Recall that open addressing was described in discussion to be when instead of storing multiple entries at the same array index as we do in chaining, we search through the table in a predetermined manner to find an available position. Within open addressing, there are two strategies that we will focus on in this project, one being linear probing, and the other being double hashing.

1.2 Dynamic Resizing

In lecture, the hash table was presented as a table with a predetermined, static size. However, as noted in discussion, a key ability of any data structure is its ability to resize itself dynamically. We want to be able to expand our hash table as elements are added to it, and we also want to contract it as elements are removed so that we aren't using an unnecessary amount of space. When implementing this, we must know when to expand/contract, and how much to expand/contract when we do. Note that when the array resizes, all of the elements that were in the array prior to resizing need to be explicitly added into the new table.

2 Implementation

2.1 Introduction

Before you start building your hash tables, you first need a hash function, so this function has been provided for you. Additionally, the secondary hash that you are to use when implementing the double hashing strategy has also been provided to you. That being said, if you are feeling ambitious, you can design your own hash function to use, though this is not required.

Now we need to build our hash tables. In this part, we aren't dynamically resizing the hash table, so the table will remain a fixed size. Two skeleton files have been provided for you. In one, you will develop a hash table based upon chaining, and in the other, you will build a hash table based on the two open addressing schemes mentioned above. Note that in the second file you can use a command line argument or constant to specify the hash function that you would like to use (this will be given in the skeleton file).

Task 1: (5%) Develop a static hash table with size 100 that handles collisions with chaining.

Task 2: (5%) Develop a static hash table with size 100 that handles collisions through linear probing.

Task 3: (5%) Develop a static hash table with size 100 that handles collisions through double hashing. The second hash function to use has been provided for you.

2.2 Dynamic Resizing

Now we are going to allow our table to dynamically resize. Make it such that the table doubles in size when reaching a load factor of 1, and halves in size when reaching a load factor of .25.

Task 4: (5%) Modify both of your implementations so that they can dynamically resize using the parameters described above.

3 Evaluation

3.1 Load Factor

Now we are going to evaluate how well our hash tables perform. Files have been provide that will allow you to benchmark each of your designs. Each line in the test file will have a number followed by a 0, 1, or 2. A 0 indicates that the preceding key should be removed and a 1 indicates that the preceding key should be added (code to parse these files will be given to you). A 2 indicates that a boolean value should be returned; true if the key is present in the hash table, and false otherwise. Now we are ready to analyze our implementations. We are first going to run our test files through our implementations while changing the load factor at which the table expands. Start at expanding at a load factor of 1 as you were doing before, and reduce it in increments of .02 to .7. For each load factor, record the number of collisions, and note that a single insert can have multiple collisions.

Task 5: (20%) For each analysis file provide a graph that displays the number of collisions each implementation had versus the expanding load factor. Therefore, for each load factor, there should be 3 points on the graph; 1 point for the collisions seen in the chaining implementation, 1 point for the linear probing implementation, and 1 point for the double hashing implementation.

3.2 Locality

Now recall that in discussion we mentioned that open addressing has a benefit over chaining due to the fact that it makes better use of spatial locality; things being close in physical addresses means that you are more likely to have a cache hit when accessing one and then the other. To simulate this, we will introduce a method that approximates whether something will be be a cache hit or a cache miss. We will be discussing a very limited, toy version of cache, but those who are interested are encouraged to investigate it further [1]. To simulate a cache, you will create an additional data structure that can store 4 values. If we access the hash table at index i, call this value $A[i]$, if $A[i]$ is not among the 4 values that are in our cache, we call this a *cache miss*. We then fill the cache with the values $A[i],\ A[i+1],\ A[i+2],\ A[i+3]$. For example, if I am trying to insert the value 13 into my hash table, and using my hash function, the value 13 hashes to the index 61. I now take $A[61]$ and check to see if it is in my cache. Suppose that it is not in the cache, then the values $A[61],\ A[62],\ A[63],\ A[64]$ will be stored in the cache. Assume now that there is a collision at index 61, and assume that I am using the linear probing method. Then I will next see if index 62 is available. Upon accessing $A[62]$, I check to see if it is in the cache, and indeed it is (since I just put it there). Suppose now that I am very unlucky, and index 62, 63, and 64 are all already occupied. Note that each of these accesses to check their values would be cache *hits*. However, since I got very unlucky, I must now look at index 65. I check to see if $A[65]$ is in my cache, and it is not. This is therefore a cache *miss*, and I then load my cache with the values $A[65],\ A[66],\ A[67],\ A[68]$.

Armed with this information, we would like to determine the average number of cache misses while the hash table has various load factors. To do this, for every access into the hash table (insert, delete, or existence check), record the current load factor, as well as the number of cache misses. Note that under our current definition, every single access will incur at least 1 cache miss (since we have to load our 4 values in initially). You can group load factors together in groups of 10, as in treat accesses with load factor 0.0-0.9 as the same, .10-.19 as the same, .20-.29 as the same, etc. Based on this information, produce a graph that has the grouping on the x-axis, and then the *average* number of cache misses on the y-axis.

Task 6: (30%) For each analysis file provide a graph that displays the average number of cache misses (as defined above) versus load size.

3.3 Analysis

Task 7: (10%) Based on your analysis, when might you use one implementation of the hash function versus another? Support your decision with the analysis that you performed. Carefully describe any aspect of your implementation that could have been improved, and how it affected your results or decision.

Task 8: (20%) Prove that arbitrary inserts and deletes on a hash table that handles collisions with chaining when expanding and contracting at load factors of 1 and .25 have a time complexity of $O(1)$ on average. Show why the constraints of 1 and .25 on the resizing points are necessary to maintain this average time complexity.

4 Academic Integrity

This project is to be done individually. Any implementation or analysis ideas that are not purely your own should be cited, and in particular ideas taken from books or the Internet. Please do not collaborate with other students while developing your programs, and please abide by the student honor code.

References

[1] Cache (computing). https://en.wikipedia.org/wiki/Cache_(computing). Accessed: 2016-04-09.