Alison Ostlund

CSCI2270/Data Structures

Final Project Write-up

Hash Table Collision Written

The main concept used throughout this assignment was the use and implementation of hash tables and how to handle collisions of data within the hash table. For starters, a hash table is a map of stored data, which can map keys to values. A hash table uses a hash function to compute an index for the data out of ASCII values, then stores the data into an array of “slots” or “buckets” based off this value. I imagine a hash table as large shelving system with buckets in a particular order on particular shelves. The data you build into the hash table is stored into a particular bucket and a particular shelve based on its value, so that if the data is desired, it can easily be found. In a perfect world, each set of data would fit perfectly into the shelves and belong to its own unique bucket uniformly. Unfortunately, this is the case and some of the data have the same value as another causing what is called a collision. In order to handle these collisions, certain algorithms must be used to store data with the same value. Two ways to resolve this problem is by chaining or by open-addressing.

The collision resolution by chaining manipulates singly-linked lists in the hash table array. Each bucket of the array contains a linked list containing all the data with the same hash value. So when a collision occurs at a particular value, the same value is placed in the same index of the array, just in a linked list. In terms of the shelve metaphor, each bucket with the same value is placed on a particular shelve and that shelve is only stores buckets with the same value. When a same data value is added to the row, it is added to end of the shelve. As you move up and down the shelving unit, you are essentially moving up and down the index of the hash table array. When a new data value is added for the first time into the table, or shelving system, it is added to the next empty shelve, or a NULL spot in the array.

The open-addressing strategy is another way to resolve collisions. In this algorithm, all index – value pairs are stored in the hash table itself, so there is no need for external data structure methods. When a new data element is inserted into the table, it is assigned to one bucket. This operation relies on empty buckets to help with collisions as long as the amount of data being stored is smaller than the table size. A simple method of open-addressing is liner probing. Which is when there is a collision with a pre-existing data element at a particular bucket, then the algorithm just checks the next index in the array, or next row up in the shelving system, for three possible outcomes. It looks for the key to equal the search key, for an empty position, or for the key to not equal the search key. Essentially, it searches for alternative locations in the array or alternative shelves in the system.

Within this Project we were asked to take a set of data of baseball players and build a hash table from the data. This data contained every Major League Baseball player on every team since 1985. Each player was listed with a certain field which contained more data about the player. Each player had a field for their first and last name, birth year, country born in, weight, height, bats, throws and teams played on and salary. The data was then put into a hash table using the players first and last name as the key. Each player’s information is then placed into the array based off the hash value calculated through the hash function. Since there are only 5147 unique players in the very large set of data given, there is a large number of collisions. In order to handle the collisions, I used both solutions decried above, chaining and open-addressing.

To get the data, I implemented a command line argument to take in the data on each player, each piece of data associated with the player was broken up into strings per line on the data file. The first and last name of the player was then passed into a hash function. This hash function calculated the ASCII values for the first and last name of the player, then moduloed the calculated values by the table size in order to obtain the hash code. This value is then used to set up and build the table passing the data into the hash function and creating new nodes for each new player. Which is then placed into the insert chaining and inset OpenAddress functions. The insert chaining function basically implements the linked lists in the array of data and takes in the nodes of players and first checks if there is a blank spot in the array, if so it sets the new node to the head of a linked list at the assigned index of the array. It sets the pointers for the linked list in the case of a collision. The insert OpenAddress function takes in the data of the new player node and checks to see if there are any empty index’s in the array, or if the next index is empty and sorts the data and collisions. The first part is essential building the hash table and dealing with the collisions that occur in the data.

The user is then prompted a menu to choose to use chaining, use open addressing, or to exit the program. When the user chooses the first option, then the collisions are resolved using the chaining method. The getchainCollisions and getChainSearchCount functions are primarily used for this option to then calculate the number of collisions that have occurred and the search count. The getchainCollisons is a counter that is returned through this function, the counter chainCollisions is incremented in the insertChaining function, which kept track of the amount of collisions while they occurred while the data was being placed in the hash table. The getChainSearchCount returns the chainSearchcounter used in the findChaining functions. Which takes in a name and last name and search’s the hash table for the player. It returns the search count. The program returns the results of a hash table size of 5147, the amount of collisions as 775084, and the search count of 26420.

If the user choose option number two instead of one, the program will essentially do the same process as it did for option one, except for open addressing resolution for collisions. The getOpenCollisoions and getOpenSearchcount functions are primarily used for this option to then calculate the number of collisions that have occurred and the search count. getchainCollisons is a counter that is returned through this function, the counter openCollisions is incremented in the insertOpenAdress function, which kept track of the amount of collisions as they occurred while the data was being placed in the hash table. The getOpenearchCount returns the openSearch counter used in the findOpenAddress functions. Which takes in a name and last name and search’s the hash table for the player. It returns the search count. The program returns the results of a hash table size of 5147, the amount of collisions as 93982599, and the search count of 26420.

The results are telling as to which collision resolution was more effective. The open address method counted a significantly larger number of collisions than the chaining did. This is because this method is less effective with larger amounts of data such like the baseball player’s data. Although it uses less memory than chaining, in this case it is less effective and harder to implement. I personally preferred the chaining method due to its organized structure and easy to implement algorithm. It had fewer number of collisions due the fact that it allows more data to be stored than the table size of the hash table. Whereas open address has a storing factor proportional to the (table size) / (1 – table size). Although the chaining method requires more storage and the use of other data structure methods, I believe it was the more effective resolution for the collisions in this set of data.