CSCI 2270 - Data Structures and Algorithms
Instructor: Hoenigman /Jacobson
Recitation 14[1]

In this recitation, you will be learning hash tables.

Hash Tables

A hash table, also known as a hash map, is a data structure that stores data using a parameter in the data, called a key, to map the data to an index in an array. The data is also called a record, and the array where records are stored is called a hash table.

Two necessary components to a hash table:
The array where the records are stored and a hash function that generates the mapping to an array index.
For example, imagine the hash table is used to store records of movies. Each movie record contains the Title, Ranking, and Year of the movie. The movie could be defined with a struct as follows:

```
struct movie{
        string Title;
        int Ranking;
        int Year;
}
```

Below, in Figure 1, shows the process of how individual movie records are stored in a hash table. Each movie is a record that contains the properties of the movie. The key for the record, which in this case is the title, is input to a hash function, shown in Figure 1 as h(Title). The hash function uses the ASCII characters in the title, generates a unique integer value for that title. That integer value is the index in the hash table array where the movie record is then stored. For example, if the hash function returned a value of 2, then the movie would be stored at index = 2 in the hash table array.

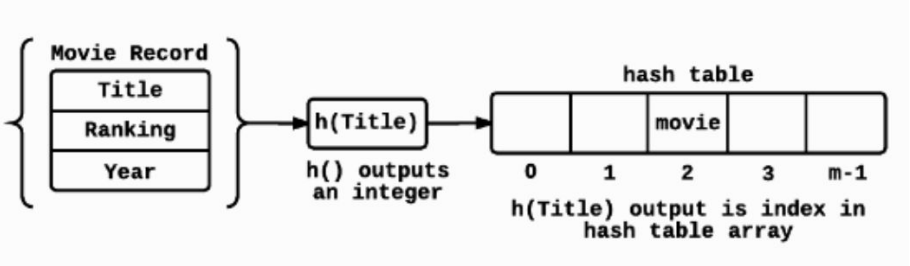

Figure 1. Process showing how individual movie records are stored in a hash table. The movie Title is the input to a hash function, which outputs an integer that serves as the index for the movie in an array.

[1] Excerpt From: Rhonda Hoenigman. “Visualizing Data Structures.” iBooks.

Hash Functions

Hash functions convert the key into an integer index to store the record in the hash table. One of the simplest hash functions converts a string to an integer by summing the ASCII values of all letters in the string and then modding the sum by the array size. The mod operation ensures that the integer is within the bounds of the array. The algorithm for this hash function is shown in Algorithm 13.1. The algorithm takes the key and the size of the hash table as arguments and returns the index where the record is to be stored in the hash table. This algorithm calculates the index in a hash table for a record with a specified key value.

Pre-conditions
key is a string or character array.
tableSize is the size of the array.

Post-conditions
Returns integer index, where 0<= index <= tableSize.

```
int hashSum(key, tableSize):
1.  sum = 0
2.  for x = 1 to key.end
3.          sum = sum + key[x]
4.  sum = sum % tableSize
5.  return sum
```

We will be looking a hash table ADT that uses a linked list to implement collision resolution with chaining. The size of the hash table and the hash table array are private variables. The public methods to insert, search, and delete take the key value as an argument. The *insert()* method adds the record to the hash table chain in alphabetical order. The *search()* method returns a pointer to the record if it is found and NULL if it is not found. The *delete()* method searches the hash table for the value, resets the pointers in the chain to bypass the record, and then frees the memory for that node. In this implementation, the hashTable stores empty records that serve as sentinel nodes to a linked list for each index.

Searching

The search() algorithm, first calculates the hash value for the search key to determine the index in the hashTable array where the record should be stored. On Line 2 (below), the algorithm checks if the hashTable array is NULL at that index. If it is NULL, then the key value is not in the hash table and the algorithm returns NULL on Line 9. If hashTable[index] is not NULL, then the key could exist in the linked list chain for that index. Lines 4 - 8 traverse the chain checking for records where the key matches the search key. Search for a node in the hash table with the specified key value and return a pointer to the node.

Pre-conditions
Unused indices in the hash table are set to NULL.
value is a valid key search value for the hash table.

Post-conditions
Returns a pointer to the node in the hash table chain where node.key = value.

search(value):

```
1.  index = hashSum(value, tableSize)

2.  if (hashTable[index].next != NULL)          //if there is something there, traverse the
linked list
3.      tmp = hashTable[index].next
4.      while(tmp != NULL)
5.          if (tmp->key == value)
6.              return tmp
7.          else
8.              tmp = tmp.next
9.  return NULL
```

Inserting or Building a HashTable

The algorithm to insert a record into a hash table, shown in Algorithm 13.3, calculates the hash value of the new record from its key, which is an argument to the algorithm. On Lines 2 -3, the key and next properties of the new record are set. Line 4 checks if there are already entries in the hash table for that hash value. If there are no entries, the new record is added as the first element at that location on Lines 5 - 6. If there are entries, Lines 9 - 13 check if the record is already in the hash table. Lines 15 - 16 traverse the chain for the position where the new record will be in alphabetical order. On Lines 17 - 19, the pointers for the existing records are updated to include the new record.

Pre-conditions
Unused indices in the hash table are set to NULL.
value is a valid hash table key value.

Post-conditions
Record inserted into the hash table at the correct location, as specified by the hash function.

insert(value)

```
1.  index = hashSum(value, tableSize)          // hash is calculated
2.  hashElement.key = value                    //set properties of new record
3.  hashElement.next = NULL
4.   hashElement.previous = NULL
```

```
5.  if (hashTable[index].next == NULL)                    // check if index has something there
6.      hashElement.previous = hashTable[index]       // if not, will add the element to that index
7.      hashTable[index].next = hashElement
8.  else                                                  // already taken and have to chain
9.      tmp = hashTable[index].next
10.       while(tmp != NULL)                              //check if record already in table
11.         if(tmp.key == value)
12.             print("duplicate")
13.             return
14.         tmp = tmp.next
15.     tmp = hashTable[index].next                       // reset pointer
16.     while(tmp.next != NULL && hashElement.title > tmp.title)      //traverse linked list
17.           tmp = tmp.next
18.     hashElement.next = tmp
19.     hashElement.previous = tmp.previous
20.     tmp.previous.next = hashElement
21.     tmp.previous = hashElement
// NOTE: This doesn't handle all cases.
22.   //TO DO  Rewire and handle the border cases,
      // i.e. insert at the end when one node is present,
      // insert at the end when more than one node is present,
      // insert at the beginning.
```

Deleting

The steps to delete a record from a hash table, shown in Algorithm 13.4, follow the same pattern initially as the steps to insert and search for a record. The index for the element to delete is identified on Line 1 with a call to the hash function. Once the key value is found in the hash table chain, on Line 5, the next and previous pointers for the surrounding nodes in the chain are updated on Lines 6 - 8. One Line 9, the node is deleted, which frees the memory.

Pre-conditions

Unused indices in the hash table are NULL.
value is a valid search key for a record in the hash table.

Post-conditions

Node with the specified key value deleted from the chain and the memory freed.
Pointers in the linked list updated to bypass the deleted node.

delete (value)

```
1.  index = hashSum(name, tableSize)
2.  if (hashTable[index].next != NULL)                    // there is a linked list.
3.      tmp = hashTable[index].next
```

```
4.      while(tmp != NULL)
5.          if(tmp.key == value)
6.              tmp.previous.next = tmp.next
7.              if(tmp.next != NULL)
8.                  tmp.next.previous = tmp.previous
9.              delete tmp
10.             break
```

In class work:

Given the array: {Whiplash, Star Wars, The Godfather, Se7en, Beauty and the Beast} where after they were sent into the hashSum, the indexes returned were 0, 0, 2, 3, 0 respectively. Draw out a hash table based on the insert function shown in class, if the hashTable size of 4.

Recitation 14 programming exercise:

For this exercise, you will implement a C++ function to build a hash table from an existing hash table using a new hash function. In the hashtable, insert the movies at the front of the linked list when the movies hash to the same index. You will have until Sunday at 5pm to complete the exercise.