Hash Table

Hashing (in terms of data structures) is a method that provides ease of storing, searching, and removing data. A **struct** or **class** is used to house the data that will be stored in the **hash table**. One component of the data will be a **key** that will be used for the storage, searching, and removal of the data as a whole. The **hash table** will be implemented as a **class** with an array as a private member variable.

For this example, suppose you have a fruit and vegetable store and you want to store your produce by PLU (Product Look-Up) code. The three components of data you would need are the PLU, name of the produce, and the cost of the produce.

```
struct PLU
{
    int pluKey; //Key for lookup in hash table
    string name;
    double cost;
};
```

Now, you could create an array large enough to hold every possible PLU, but this would be extremely wasteful. PLUs are generally 4-digit numbers with organic produce prepending a 9 onto regular PLUs. For example, the PLU for a banana is 4011 and the PLU for an organic banana is 94011. Having an array of 100,000 elements is ridiculous, especially when most of them will never be used.

Let's assume that the store sells a variety of produce throughout the year and all together will have roughly 700 PLUs (some produce have more than one PLU). The usual approach is to use an array that has a capacity larger than needed. The capacity of the array will be 811. The reasoning behind the capacity method has to do with how the data is stored in the hash table (see the section on **double hashing** and **twin primes** on the two pages).

A **hash function** is a function that maps the keys to their respective array indexes. Since the capacity of our hash table is only 811, a PLU **key** such as 4088 (red bell pepper) is much too big to fit into the array. A common hash function is:

```
template<typename RecordType>
size_t HashTable<RecordType>::hash(int key) const
{
    return (key % CAPACITY);
}
```

This ensures that the incoming key will be somewhere between 0 and 810, which is the exact range of the indexes of our array. However, this method may associate two or more keys with the same index. This

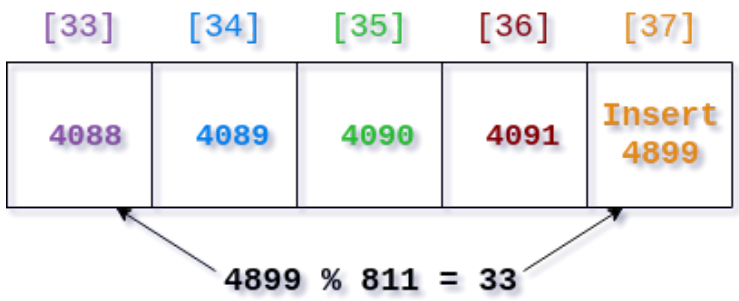
is called a **collision**. For example, 4088 % 811 = 33, so the data for red bell peppers gets placed at index 33. In addition, regular/curly parsley has the plu 4899, and 4899 % 811 = 33. There are a couple methods to fixing this problem.

Two common ways of hashing are **open-address hashing** and **chained hashing**. In **open-address hashing**, the hash function gives the appropriate index the data should be placed. If, however, the spot is taken, try the next index in sequence, and so forth. If the index is, for example, 809, try 810. If those two options do not work, wrap around the array and try index 0, then 1, etc. This is known as **linear probing**. Here is the function to return the next index for potential insertion:

```
template<typename RecordType>
size_t HashTable<RecordType>::nextIndex(size_t index) const
{ //Ensures wraparound when index is 810 as next elem in sequence is 0
    return ((index+1) % CAPACITY);
}
```

In **chained hashing**, each index is actually a linked list and will allow for multiple insertions per index.

Furthermore, the method of **open-address hashing** may cause **clustering** of data which makes insertions take longer. For example, PLUs 4088, 4089, 4090, and 4091 will all be placed in indexes 33, 34, 35, and 36 respectively. When PLU 4899 needs to be inserted, all of the above indexes will be checked as the original 33 index is taken (read paragraph above as to why). Ultimately, PLU 4899 will be placed in index 37 (the next available spot after 33). To avoid this, a technique known as **double hashing** is used. **Double hashing** uses a second hash function to place the data if the first hash function returns an index that is already taken.




```
template<typename RecordType>
size_t HashTable<RecordType>::hash2(int key) const
{
    return 1 + (key % (CAPACITY - 2));
}
```

With this second hash function, PLU 4899 will still originally try to be placed into index 33 (using the first hash function), find out it is taken by PLU 4088, and then try to be placed into an index

`1 + (4899 % (811 - 2)) = 46` spots after index 33. An important note about this function is that it does not produce the next index to check, but how many spots after the previously checked index. Therefore, the **nextIndex** function on the previous page needs updated to:

```
template<typename RecordType>
size_t HashTable<RecordType>::nextIndex(size_t index, int key) const
{
    return ((index + hash2(key)) % CAPACITY);
}
```

With this method, there is yet another problem. The program must examine every array index, and with **double hashing** the program could go back to the starting index before every array index was examined. Using the right capacity is key to preventing this problem.

Determining the capacity boils down to mathematics, and computer scientist Donald Knuth suggests that both the **capacity** and **capacity - 2** be prime numbers. For this example, capacity is 811 (811 - 2 = 809 and that's also a prime number). This method is called **twin primes** (two primes separated by a difference of 2).

Insertion

```
template<typename RecordType>
void HashTable<RecordType>::insert(const RecordType &entry)
```

In order to insert your data entry, you will pass in an instance of your **struct** or **class** to the insert function. Inside the function will have a **bool** to indicate whether or not the data already exists (via the pluKey) and a **size_t index** for appropriate placement in the hash table. The reason for the **size_t** type is because an array index can never be negative.

A check to see if the entry already exists occurs. If the entry already exists, no new insertion takes place.

If the entry is new, check to make sure the hash table is not full as no entries can be inserted if that is the case. If there is still room in the hash table, obtain the appropriate index for placement. If the hash function returns a used index, use **linear probing** to find an appropriate index. Once an index has been found, increment **used** (variable in hash table class to keep track of how many elements are filled in the array) by one for the new entry. Finally, set the entry in the proper array element by using **hashTableArray[index] = entry**;

Searching

```
template<typename RecordType>
void HashTable<RecordType>::find(int key, RecordType &result) const
```

For this example, searching for data in the hash table requires two arguments to the search (find) function. The pluKey and an instance of the **struct** or **class** by reference (to house a copy of the data if found). Just like the insert function, the find function has a **bool** variable and a **size_t index** variable. These are used to check and see if the entry is already in the hash table. If the entry already exists, copy over the entry into the passed-in instance, such as **result = hashTableArray[index];** If the entry does not exist, a message to the user may be output and no further action is to take place. This means that the passed-in instance remains how it was beforehand.

Deletion

```
template<typename RecordType>
void HashTable<RecordType>::remove(int key)
```

The removal function only requires the key to be passed into the function. This function, just like the find function, remove will have **bool** and **size_t** variables for the same reason. If the entry already exists, set **hashTableArray[index] = PREVIOUSLY_USED;** and decrement **used** by one. **PREVIOUSLY_USED** is a negative integer to aid in the finding of entries. A full explanation is in **HashTable.cpp**. If the entry does not exist, a message to the user may be output and no further action is to take place.

Conclusion

Open-address hashing with one hash function is the focus of the example program **HashTable.cpp**. You will be able to see how to implement functions for insertion, searching, and deletion of data in a hash table. If you would like, you are able to add the second hash function as a quick exercise to understand this concept further.

Chained hashing will be a topic in CSIII, so for now here is a visual of how this is implemented. The array is an array of linked lists such that each linked list is able to hold more than one piece of data per hash index.