Lecture 12

HashTable: Implementation

Last time:

Knowing the index value of an array makes us to find an object at constant time.

Thus, our savior to make search faster is a hash function that returns an index value of an object.

However, generally, it turns out that it is not possible to have a hash function that does not have any collision.

There are a few ways to deal with collisions.

- **Linear Probing**: When there is a collision, we try to find an empty cell sequentially and put the value into the nearest empty cell. However, this approach has an issue of forming the primary clusters and the performance can get really bad. For this, we need to keep load factor and, from time to time, we need to rehash.
- There is also **Quadratic Probing** that has another subtle clustering issue called secondary clustering due to the fixed interval of probing.
- To solve this issue with the fixed interval, there is another workaround, called **Double Hashing**, which uses two hash functions. One is to calculate hash value and the other is to decide the step size of probing.
- **Separate Chaining**: The other workaround is to have a linked list at each index. This allows us to not to worry too much about load factor. However, we do not want to make the linked lists become too full either.

Time to get our hands dirty, implementing hash table!

We will implement hash table using ***linear probing as its collision resolution method*.**

Major operations of our hash table are:

- search(int key)
- insert(int key)
- delete(int key)

HashTableInterface interface

```
/*
 * HashTable interface that takes only positive int values
 */
public interface HashTableInterface {
      /**
       * Inserts a new int key to the table
       * @param key int key to be inserted
       */
      void insert(int key);

      /**
       * Returns true when the key is found
       * @param key int key value to be searched
       * @return boolean value true if found, false not found
       */
      boolean search(int key);

      /**
       * Deletes and returns an int value from the table
       * @param key int value to be deleted
       * @return the deleted int value
       */
      int delete(int key);
}
```

HashTable class

```
public class HashTable implements HashTableInterface {
      private static final DataItem DELETED = new DataItem(-1);
      private DataItem[] hashArray;
      private int tableLength;

      // precondition: initialSize is a positive int
      public HashTable(int initialSize) {
            tableLength = initialSize;
            hashArray = new DataItem[tableLength];
      }

     // static nested class
      private static class DataItem {
            private int key;

            public DataItem(int key) {
                  this.key = key;
            }
      }

       // TODO implement main methods here
}
```

First things first: Hashing method

```
// private helper method for hashing a key value
private int hashFunc(int key) {
      return key % tableLength;
}
```

Look too simple? Well… for now!

Searching for a key

```
@Override
public boolean search(int key) {
      int hashVal = hashFunc(_____key_____);
      while(hashArray[hashVal] != _____null_____) {
            if(hashArray[hashVal].___key___ == key)
                  return true; // found

            _____hashVal++_____;
            // hashVal cannot be bigger than tableLength.
            // Need to wrap around
            hashVal = hashVal _____%_____tableLength;
      }
      return false; // cannot find
}
```

Deleting a key value (Practice for you!)

```
@Override
public int delete(int key) {
      int hashVal = hashFunc(key);
      while(hashArray[hashVal] != null) {
            if(hashArray[hashVal].key == key
temp=...
                  hashArray[hashVal] = DELETED;
            hashVal++
            hashVal = hashVal % tableLength
      }
      return temp;
```

Inserting a new int value

```
@Override
public void insert(int key) {
      DataItem item = new DataItem(key);
      int hashVal = hashFunc(_____key_____);

      // find an empty or available (deleted) index
      while(hashArray[_____hashVal_____] != null &&
hA[hV] != DELETED                                    ) {
            hashVal++;
            // wrap around
            hashVal = hashVal _____%TL_____;
      }

      hashArray[_____hV_____] = item;
}
```

A thought about open addressing:
Having "deleted" stage is one of the reasons why open addressing may not be efficient. When there are many deletions, search will take longer. Especially, unsuccessful search could be quite slow.

1, empty
2. occupied
3. deleted

We had a chance to see how a hash table can be implemented, at least, using linear probing as its collision resolution method. ***There is a lot of room for improvement in the code. It is just to show the basic concept and logic.***

Moving on, our next question is how we can implement our own hash algorithm? In other words, what makes a good hash function?

What makes a good hash function?
Quick computation is the key to a good hash function. Thus, a hash function with many multiplications and divisions is NOT a good idea.

The purpose of a hash function is to take a range of key values and transform them into index values in a way that the ***key values are distributed randomly across all the indices*** of the hash table.

Key values may be completely random or not so random.

1. Random key values
If the key values are random, then we can simply find index values by the following simple operation just like our code before.

index = key % tableLength;

2. Non-random key values
For example, there is a database that uses car-part numbers as key values.

033-400-03-94-05-0-535

This is interpreted as follows:

- Digits 0-2: Supplier number (1 to 999, currently up to 70)
- Digits 3-5: Category code (100, 150, 200, 250, up to 850)
- Digits 6-7: Month of introduction (1 to 12)
- Digits 8-9: Year of introduction (00 to 99)
- Digits 10-11: Serial number (1 to 99, never exceeds 100)
- Digits 12: Toxic risk flag (0 or 1)
- Digits 13-15: Checksum (sum of the other fields)

Based on the interpretations provided, the key value should be 0,334,000,394,050,535 for the particular part number shown above.

We can say that there is no guarantee that we will have random numbers between 0 to 9,999,999,999,999,999.

Some work should be done to have these part numbers to form a range of more random numbers.

- **Don't Use Non-Data**: The key values should be squeezed as much as it could. For example, category code has to be changed to be from 0 to 15. Also, the checksum should be removed because it is derived number from other information and does not add any new information.
- **Use All the Data**: Other than the non-data values, we need to use all of the data values. Don't just use the first four digits, etc.
- **Use a Prime Number for the Modulo Base**: Which means the table array length should be a prime number. This can prevent clustering from happening. For example, if the table array length is 50, then all of the multiples of 50 in our car-part numbers will be hashed into the same index.
- **Use Folding**: Another reasonable hash function involves breaking the keys into groups of digits and adding the groups.

  SSN example: 123-45-6789

  In case table array length is 1009: Break the number into three groups of three digits. (123+456+789 = 1368 %1009 = 359).

  In case table array length is 101: Break the number into four two-digit numbers and one one-digit number. (12+34+56+78+9 = 189%101 = 88).

  This way you can *distribute* the numbers better.

The basic idea is to **examine your key values carefully** and implement your hash function to *remove any irregularity in the distribution of the key values*. More on this in the next lecture!