15.5 Hash Tables with Chaining

Seek, and ye shall find.

MATTHEW 7:7

A **hash table** or **hash map** is a data structure that efficiently stores and retrieves data from memory. There are many ways to construct a hash table; in this section we will use an array in combination with singly linked lists. In the previous section we saw that a linked list generally requires linear, or $O(n)$, steps to determine if a target is in the list. In contrast, a hash table has the potential to execute a fixed number of steps to look up a target, regardless of the size of n. We saw that a constant-time lookup is written $O(1)$. However, the hash table we will present may still require n steps, but such a case is unlikely.

hash table
hash map

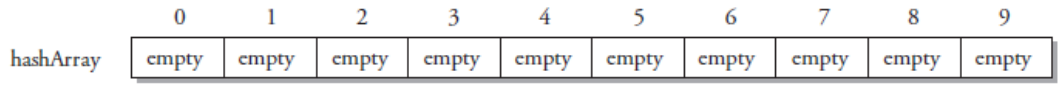
An object is stored in a hash table by associating it with a *key*. Given the key, we can retrieve the object. Ideally, the key is unique to each object. If the object has no intrinsically unique key, then we can use a **hash function** to compute one. In most cases the hash function computes a number.

hash function

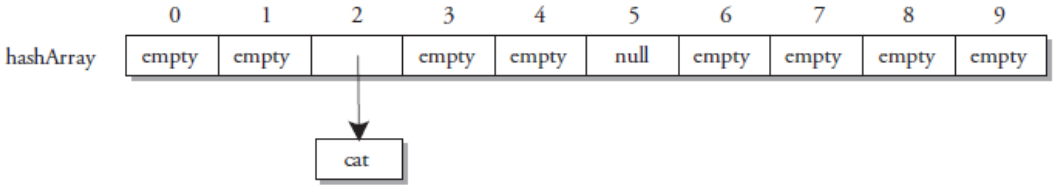
For example, let's use a hash table to store a dictionary of words. Such a hash table might be useful to make a spell checker—words missing from the hash table might not be spelled correctly. We will construct the hash table with a fixed array in which each array element references a linked list. The key computed by the hash function will map to the index of the array. The actual data will be stored in a linked list at the hash value's index. The Display below illustrates the idea with a fixed array of ten entries. Initially each entry of the array `hashArray` contains a reference to an empty singly linked list. First we add the word `"cat"`, which has been assigned the key or hash value of 2 (we'll show how this was computed shortly). Next we add `"dog"` and `"bird"`, which are assigned hash values of 4 and 7, respectively. Each of these strings is inserted as the head of the linked list using the hash value as the index in the array. Finally, we add `"turtle"`, which also has a hash of 2. Since `"cat"` is already stored at index 2, we now have a **collision**. Both `"turtle"` and `"cat"` map to the same index in the array. When this occurs in a hash table with **chaining**, we simply insert the new node onto the existing linked list. In our example there are now two nodes at index 2: `"turtle"` and `"cat"`.

1. Existing hash table initialized with 10 empty linked lists

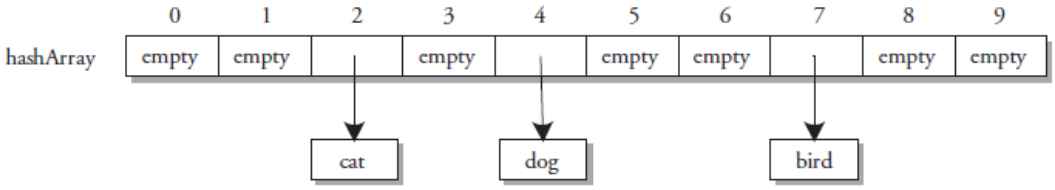
```
hashArray = new LinkedList 3[SIZE]; // SIZE = 10
```



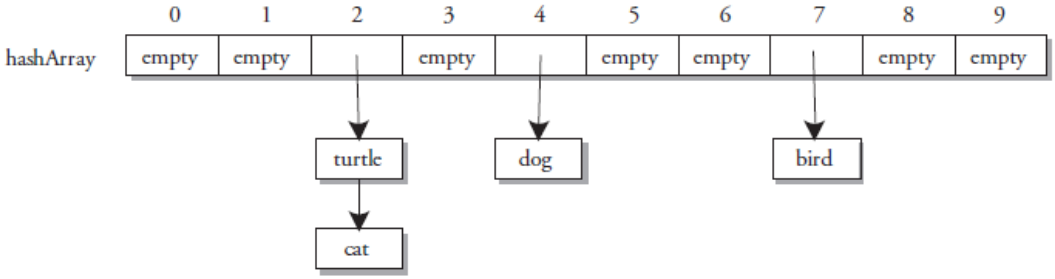
2. After adding "cat" with hash of 2



3. After adding "dog" with hash of 4 and "bird" with hash of 7



4. After adding "turtle" with hash of 2 – collision and chained to linked list with "cat"



collision
chaining

To retrieve a value from the hash table, we first compute the hash value of the target. Next we search the linked list that is stored at `hashArray[hashValue]` for the target, using an iterator to sequentially search the linked list. If the target is not found in this linked list, then the target is not stored in the hash table. If the size of the linked list is small then the retrieval process will be quick.

A Hash Function for Strings

A simple way to compute a numeric hash value for a string is to sum the ASCII value of every character in the string and then compute the modulus of the sum using the size of the fixed array. Code to compute the hash value is shown below:

```
private int computeHash(String s)
{
  int hash = 0;
  for (int i = 0; i < s.length( ); i++)
  {
      hash += s.charAt(i);
  }
  return hash % SIZE; // SIZE = 10 in example
}
```

For example, the ASCII codes for the string "dog" are as follows:

```
d   ->100
o   ->111
g   ->103
```

The hash function is computed as follows:

```
Sum              = 100 + 111 + 103   = 314
Hash = Sum % 10  = 314 % 10          = 4
```

In this example we first compute an unbounded value, the sum of the ASCII values in the string. However, the array was defined to only hold a finite number of elements. To scale the sum to the size of the array, we compute the modulus of the sum with respect to the size of the array, which is 10 in the example. In practice the size of the array is generally a prime number larger than the number of items that will be put into the hash table.[1] The computed hash value of 4 serves like a fingerprint for the string "dog". However, different strings may map to the same value. We can verify that "cat" maps to (99 + 97 + 116) % 10 = 2 and also that "turtle" maps to (116 + 117 + 114 + 116 + 108 + 101) % 10 = 2.

Note that we can improve our hashing function by using a prime number for modulus. A prime number avoids common divisors after modulus that can lead to collisions if the hash values are not uniformly distributed.

Some demonstration code for a Hash Table follows. In this example we use the ArrayList generic class in place of a Linked List; the end result is similar.

```
public class Node {
    private String key;
    private String payload;
    public Node()
    {
        key="";
        payload="";
    }
    public Node(String key, String payload)
    {
        this.key = key;
        this.payload = payload;
    }
    public String getKey()
    {
        return key;
    }
    public String getPayload()
    {
        return payload;
    }
   public boolean equals(Object o)
    {
        Node other = (Node) o;
        return other.key == this.key;
    }
}
```

```
import java.util.ArrayList;
public class HashTable {
    private ArrayList<Node>[] hashArray;
    private static final int SIZE = 10;

    public HashTable()
    {
        hashArray = (ArrayList<Node>[]) new ArrayList[SIZE];
        for (int i = 0; i < SIZE; i++)
        {
            hashArray[i] = new ArrayList<Node>();
        }
    }

    private int computeHash(String s)
    {
        int hash = 0;
        for (int i = 0; i < s.length(); i++)
        {
            hash += s.charAt(i);
        }
        return (hash % SIZE);
    }

    public String retrievePayload(String target)
    {
        int hash = computeHash(target);
        ArrayList<Node> arrlist = hashArray[hash];
        for (Node n : arrlist)
        {
            if (n.getKey()==target)
                return n.getPayload();
        }
        return "";
    }

    public void add(Node n)
    {
        int hash = computeHash(n.getKey());
        ArrayList<Node> arrlist = hashArray[hash];
        if (!arrlist.contains(n))  // Don't add if duplicate
        {
            arrlist.add(n);
        }
    }
}

public class HashTableExample {
    public static void main(String[] args) {
        HashTable h = new HashTable();
        h.add(new Node("dog","This is a dog"));
        h.add(new Node("cat","This is a cat"));
        h.add(new Node("turtle","This is a turtle"));
        h.add(new Node("bird","This is a bird"));
```

```
        System.out.println("Contains dog? " + h.retrievePayload("dog"));
        System.out.println("Contains cat? " + h.retrievePayload("cat"));
        System.out.println("Contains fish? " + h.retrievePayload("fish"));
        System.out.println("Contains bird? " + h.retrievePayload("bird"));    }
}
```

Efficiency of Hash Tables

The efficiency of our hash table depends on several factors. First, let's examine some extreme cases. The worst-case run-time performance occurs if every item inserted into the table has the same hash key. Everything will then be stored in a single linked list. With *n* items, the find operation will require *O*(*n*) steps. Fortunately, if the items that we insert are somewhat random, the probability that all of them will hash to the same key is highly unlikely. In contrast, the best-case run-time performance occurs if every item inserted into the table has a different hash key. This means that there will be no collisions, so the find operation will require constant, or *O*(1), steps because the target will always be the first node in the linked list.

We can decrease the chance of collisions by using a better hash function. For example, the simple hash function that sums each letter of a string ignores the ordering of the letters. The words `"rat"` and `"tar"` would hash to the same value. A better hash function for a string `s` is to multiply each letter by an increasing weight depending upon the position in the word. For example:

```
int hash = 0;
for (int i = 0; i < s.length( ); i++)
{
    hash = 31 * hash + s.charAt(i);
}
```

Another way to decrease the chance of collisions is by making the hash table bigger. For example, if the hash table array stored 10,000 entries but we are only inserting 1,000 items, then the probability of a collision is much smaller than if the hash table array stored only 1,000 entries. However, a drawback to creating an extremely large hash table array is wasted memory. If only 1,000 items are inserted into the 10,000-entry hash table then at least 9,000 memory locations will go unused. This illustrates the **time-space tradeoff**. It is usually possible to increase run-time performance at the expense of memory space, and vice versa.

Hash Table variants to cover in the video:

- Linear probing
- Quadratic probing
- Rationale and problems with both! Especially with deletion.
- Double hashing