Software Development (CS2500)

Lecture 17: Hashing

M.R.C. van Dongen

November 10, 2010

Contents

1 Outline

Today we shall study *hashing*. Hashing lets you map keys to positions in arrays. *Perfect hashing* is rarely possible: in general *collisions* will occur. Should they occur, these collisions should be *resolved*. We shall study two classes of collision resolution strategies. The first strategy is *open addressing*. This technique looks for other free cells in the array. The second strategy is *separate chaining*. This technique associates an additional data structure with each occupied cell in the array. Keys that hash to a given cell in the array are stored in the data structure for that cell.

2 The Method `hashCode( )`

Arrays are the basic data structure for many applications. However, storing and retrieving by index isn't always ideal. For example, let's assume you want to implement a `Table` class. Each item in the `Table` has a *key*. Different items have different keys. The `Table` supports the following methods.

`Table( int size ):` Create an *empty* `Table` object. The `Table` should have a maximum capacity of `size`.

`boolean isFull( ):` Determine if the `Table` is full. (We shall not implement this method. However, implementing the method is not too difficult. For example, by maintaining an attribute `size` that counts the number of objects in the `Table`, the `Table` is full if and only if `size` is equal to the maximum capacity of the `Table`.)

`void add( Object o ):` Add a new item to the `Table`. We shall insert the item using its key.

`void remove( Object o ):` Remove an existing item from the `Table`.

`boolean contains( Object o ):` Decide if the `Table` contains a given member.

All operations should be fast. We shall use an array to store the objects in the `Table`. The objects are mapped to positions in the array by using their keys. So how are we going to associate the objects' keys with positions in the array.

Before we continue, it is recalled that the `Object` class defines the instance method `int hashCode( )`. This method returns an `int` which is guaranteed to be unique, unless you override the method. Many applications use `hashCode( )` as an object key. Let's try and use `hashCode( )` to implement our `Table` class. We shall use a method '`int hashFunction( Object o )`' which *hashes* the object `o` to its position in the array. The following is a *partial* implementation, which, for example, doesn't implement `isFull( )`.

Don't Try this at Home

```
public class Table {
    private final Object[] members;

    public Table( int size ) {
        members = new Object[ size ];
    }

    private int hashFunction( Object o ) {
        final int key = o.hashCode( );
        return (members.length + key % members.length) % members.length;
    }

    public void add( Object o ) {
        members[ hashFunction( o ) ] = o;
    }

    public void remove( Object o ) {
        members[ hashFunction( o ) ] = null;
    }

    public boolean contains( Object o ) {
        return o == members[ hashFunction( o ) ];
    }
}
```

The method `hashFunction( )` maps the hash code — the key — of an object to a position in the array. Note that the method cannot rely on returning '`key % members.length`' because `key` may be negative.

You probably already guessed that this implementation is not ideal. Despite the fact that different objects have different keys (hash codes), we cannot guarantee that there won't be *collisions*. Here a collision occurs when two different objects are mapped to the same index position in the array. For example, if the length of the array is 2 then we will always get a collision if we insert two objects with different even keys. The same happens if we add two objects with different odd hash codes.

As mentioned before, the `Object` class defines the instance method `int hashCode( )`. The default implementation of `hashCode( )` may not always be useful: overriding is usually required. Care should be taken when overriding hash codes. Always override `hashCode( )` if you override `equals( )`. You should always make sure that `o1.hashCode( ) == o2.hashCode( )` if `o1.equals( o2 )`. Some classes rely on this contract. The idea is that two objects which are deeply equal should have equal hash codes.

When computing hash codes you should take as much information into account as possible. Here "information" are the attributes which determine `equals( )`. This generally reduces collisions. For example, use all characters to compute the hash code of a `String` attribute. To combine the attributes you usually use the following scheme [Bloch, 2008, Item 7]:

Pseudo Code

```
int hashCode = ⟨constant⟩;
hashCode = hashCode * ⟨prime number⟩ + ⟨hash code of 1st attribute⟩;
hashCode = hashCode * ⟨prime number⟩ + ⟨hash code of 2nd attribute⟩;
hashCode = hashCode * ⟨prime number⟩ + ⟨hash code of 3rd attribute⟩;
…
```

There are some guidelines as to how to compute ⟨`hash code of` *n*`th attribute`⟩. For example, if it is a `boolean` then you return 1 if the `boolean` is `true` and 0 if it is `false`. If it is `byte`, `char`, or a `short` then you cast it to an `int`. It it is an `int` then you use the value of the attribute. If it is an object type then you return the hash code of the object. If the attribute is an array then you should treat what's in the array as if they were attributes themselves. For example, the following class overrides `hashCode( )` because it overrides `equals( )`, which dependds on the attributes `age` and `name`. Overriding `hashCode( )` starts by assigning a constant to the result. Next the result is combined with the attribute `age`. Finally, the result is combined with each of the hash codes of the `char`s in the attribute `name`.

```java
public class Person {
    final static int CONSTANT = 17;
    final static int PRIME = 13;
    private final String name;
    private final int age;

    …
    @Override
    public boolean equals( Object that ) {
        final Person p = (Person)that;
        return name.equals( p.name ) && age == p.age;
    }

    @Override
    public int hashCode( ) {
        int hashCode = CONSTANT;
        hashCode = hashCode * PRIME + age;
        for (char c : name) {
            hashCode = hashCode * PRIME + c.hashCode( );
        }
        return hashCode;
    }
}
```

There won't be any questions in the exam about how to override `hashCode( )`. More details about how to compute hash codes may be found in [Bloch, 2008, Item 7].

3 Random Hash Functions

In this section we shall solve some of the problems with our initial implementation of the `Table` class.

Our objective is to use an object's hash code to find a position for the object in the `Table`'s array. This is known as *hashing* the object to a position in the array. The following formally defines the notion of a hash function.

Definition 1 (Hash Function)**.** A *hash function* is a function that maps its argument to an index position of an array.

In the `Table` class our task was hashing each object's key to an index in an array. Since each object's key is its hash code, we may reduce this task to that of hashing `int`s to positions in an `int` array:

```java
public class IntTable {
    int[] members;

    private int hashFunction( int key ) {
        return (members.length + key % members.length) % members.length;
    }
    …
}
```

The following are some desirable properties which every good hash function should have:

- Good hash functions should be correct: the result should be an allowed index.

- They should be fast. Hash functions get called a lot. If they are slow they cause an unnecessary delay.
- They should minimise collisions. Resolving collisions is expensive in terms of time. The fewer there are, the better.
- Every index position should be generated with about the same probability. With this property, the average number of collisions per index position stays low. This is good because low collision counts per index position usually guarantee fast collision resolutions.

Despite your best attempts, overriding `hashCode( )` may still not always work. For example, let's assume we have three objects. The hash codes of the objects are 0, 1, and 5. Let's also assume we have two `Table` objects. `Table` 1 has a capacity of 3: its hash function is $h_3(c) = c \,\%\, 3$. `Table` 2 has a capacity of 4: its hash function is $h_4(c) = c \,\%\, 4$. The first hash function gives no collisions: $h_3(0) = 0$, $h_3(1) = 1$, and $h_3(5) = 2$. The second hash function does give a collision: $h_4(0) = 0$, $h_4(1) = 1$, and $h_4(5) = 1$. Perhaps this came as a surprise because the capacity of the second array was greater than that of the first. So, how do we fix this problem?

One solution is to let each `Table` object compute its own hash function. They start with a random hash function. They use it until a collision arises. Should this happen, the `Table` may resolve the collision by computing another random hash function.

In the following we assume the hash codes are in the domain $U = \{0, \ldots, m-1\}$. Furthermore, assume our array has length $n \leq m$. Note that usually $n \ll m$ because you almost always want to save space by hashing values from a larger domain into a smaller index set. Let $p \geq m$ be a random prime. Finally, let $0 < a < p$ and $0 \leq b \leq p$ be random integers. Consider the following hash function:

$$h_{a,b}(x) = ((ax + b) \,\%\, p) \,\%\, n\,.$$

If x and y are random members from U, then

$$\mathbb{P}\left(h_{a,b}(x) = h_{a,b}(y)\right) \leq n^{-1}\,.$$

Here the notation $\mathbb{P}(X)$ is the probability of the event X. See [Mitzenmacher and Upfal, 2005, Lemma 13.6] for a proof.

We may use this class of hash functions to improve our implementation of our `IntTable` hash function. Note that $n \leq m$, so we may decide to choose $m = n$. The implementation of the method `int randomPrime( int m )` is not provided. The method should return a random prime which greater than or equal to `m`.

```java
public class IntTable {
    final Random rand;
    int[] array;
    int a, b, m, p;

    public IntTable( int size, int m ) {
        array = new int[ size ];
        rand  = new Random( );
        this.m = m;
        computeNewHashFunctionConstants( );
    }

    private void computeNewHashFunctionConstants( ) {
        p = randomPrime( m );
        a = 1 + rand.nextInt( p - 2 );
        b = rand.nextInt( p + 1 );
    }

    // ASSUMPTION 0 <= x <= m.
    private int hashFunction( int x ) {
        final int n = array.length;
        return ((a * x + b) % p) % n;
    }
    …
}
```

Note that we assume that the argument of `hashFunction( )` is non-negative and does not exceed `m`. If this assumption isn't reasonable, then we should first transform the argument to the range $\{0, \ldots, m-1\}$. This should be done in such a way that each transformation result is equally likely. A reasonable way to do this may be the following, but it depends on the input.

```java
private int hashFunction( int key ) {
    final int x = (m + key % m) % m;
    final int n = array.length;
    return ((a * x + b) % p) % n;
}
```

4 Collision Resolution

The class of hash functions which we studies in the previous section is pretty much defenceless when it comes to collisions. In this section we shall study different kinds of hash functions which are more robust because they actively resolve collisions. The following is a global overview.

A *perfect hash function* maps each key to a unique index. Non-perfect hash function results in collisions. If a collision occurs when adding a key we must *resolve the collision*. There are two techniques.

Open addressing: Use a different, free index.

Buckets: Allow multiple keys per index.

In the remainder of this section we shall study three collision resolution strategies. The first two use open addressing techniques. The other strategy uses buckets.

4.1 Running Example

In the following section we shall use a running example to demonstrate the difference between the resolution strategies. In the running example, we have a number of characters which should be hashed into an array. Each character is a uppercase letter. When using a letter, we shall give the letter a subscript which is equal to the letter's position in the alphabet: B_2, J_{10}, S_{19} and so on. The length of the array is 7. For simplicity we shall use the hash function, $h(\Xi_n) = n \,\%\, 7$, which maps the character with subscript n to $n \,\%\, 7$. So $h(B_2) = 2$, $h(J_{10}) = 3$, $h(S_{19}) = 4$, and so on.

We shall start our running example by inserting the letters B_2, J_{10}, and S_{19} into the array. Figure 1 depicts the final situation. So far there are no collisions. In the remaining sections we shall study some collision resolution strategies.

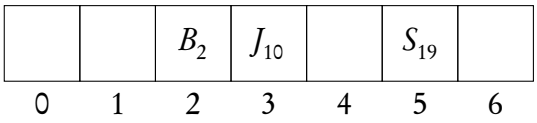

Figure 1: Final situation after inserting B_2, J_{10}, and S_{19}.

4.2 Linear Probing

This section studies the first of two open addressing resolution strategies: linear probing.

Let $n > 2$ be a prime and let's assume we have an n-sized array which isn't full. Furthermore, let's assume we want to insert a key, k. However, this time a collision occurs. To resolve the collision we need to find a free cell. To find the free cell we start at $h(k)$, next visit $(h(k) - 1) \,\%\, n$, visit $(h(k) - 2) \,\%\, n$, ... Here n is the length of the array. Eventually, we should find some free cell.This collision resolution policy is called *linear probing*.

To see how linear probing works, we shall continue our running example and insert the letters N_{14}, X_{24}, and W_{23}.

Inserting N_{14} poses no problem. However, when we try to insert X_{24} we have a collision: $h(X_{24}) = h(J_{10}) = 3$. To resolve the collision at the current cell, Cell 3, we probe the previous cell, Cell 2, which also happens to be full. We continue by probing the next previous cell, Cell 1. Cell 1 happens to be free, and we resolve the collision by inserting X_{24} in Cell 1.

Inserting W_{24} also results in a collision because $h(B_2) = h(W_{24})$. This time we have to probe three more cells before we find the free cell: Cell 1, Cell 0, and Cell 6. Cell 6 is free and we may resolve the conflict by inserting W_{24} in Cell 6. The resulting situation is depicted in Figure 2.

Linear probing visits a sequence of occupied and free indices. The sequence starts with a sequence of occupied cells — this sequence may be empty — and ends with zero or one free cell. All cells are occupied if and only if the `Table` is full. This is called the *probe sequence*. The hash function determines the sequence's first index. The remaining indices are determined by the collision resolution policy. Given

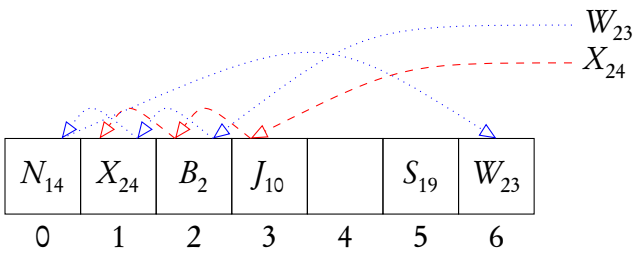

Figure 2: Running example after inserting N_{14} and resolving the collisions of X_{24} and W_{23}. The probe sequence of X_{24} is indicated by the red, dashed lines. The probe sequence of W_{23} is indicated by the blue, dotted lines.

current index i, the next index is $(i - p(i)) \% n$, Where $p(\cdot)$ is the *probe decrement function*.[1] For linear probing the probe decrement function is given by $p(i) = 1$.

Having studied how to insert keys into the `Table`, we continue by studying how to remove keys from the `Table`. When removing a value we shouldn't break a probe sequence. E.g. let's assume we remove B_2 by making Cell 2 empty. If we do this we'll have problems locating X_{24}: A free cell in the probe sequence should indicate that B_2 is not in the table.

The following is what we do when we remove a value from our array. Instead of making B_2's cell available, we mark it with a special value $\perp$. The value $\perp$ should be a value which cannot occur as a key. When looking for items we treat $\perp$ as a used cell. When looking for free cells we treat $\perp$ as a free cell.

4.3 Double Hashing

Linear probing is not a good collision resolution policy. It tends to lead to clusters, which in their turn lead to bigger clusters, ... Inserting new items in the presence of large clusters leads to long probe sequences. The larger the clusters get the faster they grow. In general it is better to have *different* probe decrements for different keys. This is called *double hashing*.

With double hashing we have two hash functions. The first hash function is $h_1(k)$. This function is used to find the starting position in the array. The second hash function, $h_2(k)$, is a random random probe decrement as function of k. We use the two to compute the probe sequence (modulo n):

$$h_1(k), (h_1(k) - h_2(k)) \% n, (h_1(k) - 2h_2(k)) \% n, (h_1(k) - 3h_2(k)) \% n, \ldots .$$

The value of $h_2(k)$ should be a random value which is *relative prime* to n [Cormen *et al.*, 2001, Page 240]. Here k and n are relative prime if $\gcd(k, n) = 1$. Note that this makes sense, because we may need the probe sequence to visit *each* of the $n - 1$ remaining cells. If $\gcd(h_2(k), n) \neq 1$ then we won't be able to

[1] Note that if the probe decrement is in the range $\{1, \ldots, n - 1\}$, then the probe sequence should let us find any free cell. This follows from the "Square Hopping" observation in the previous lecture because n is a prime.

reach some of the cells. The following may be a reasonable choice if $k \,\%\, n$ "behaves" randomly and n is large.

$$h_2(k) = \begin{cases} 1 & \text{if } k = 0; \\ h_2(k/n) & \text{if } n \mid k; \\ k \,\%\, n & \text{if } n \nmid k. \end{cases}$$

The assumption that $k \,\%\, n$ "behaves" randomly is quite reasonable for many applications. If n is small, then the following is probably better because it avoid recursion:

$$h_2(k) = \begin{cases} 1 & \text{if } k \mid k; \\ k \,\%\, n & \text{if } n \nmid k. \end{cases}$$

4.4 Limitation of Open Addressing

Open addressing techniques as linear and double hashing have some undesirable properties. The first disadvantage is that frequent additions soon lead to clustering, which slows down the hashing because of long probe sequences. The second disadvantage is that frequent additions will eventually fill the table. Both problems may be overcome by resizing the table. Java has many classes based on hashing. Many of them actively re-size so they can ensure good performance.

4.5 Separate Chaining

Another collision resolution strategy is *separate chaining*. It changes the structure of the hash table. Here table locations now store *multiple* values. Each non-empty table index now has a *bucket*. If a location is free then we add an object to a new bucket. If a location is occupied, we add the object to the location's bucket. A possible implementation for the bucket is a linked list.[2] Separate chaining provides a simple way to resolve collisions. However, it requires more memory than open addressing. Figure 3 depicts the final situation for separate chaining and the running example.

5 Acknowledgements

The running example is based on [Standish, 1994, Chapter 11].

References

[Bloch, 2008] Joshua Bloch. *Effective Java*. Addison-Wesley, 2008.

[Cormen *et al.*, 2001] Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, and Clifford Stein. *Introduction to the Analysis of Algorithms*. MIT Press and McGraw-Hill, 2001. Check Year and add isbn.

[2] A linked list is a data structure which we will study after Christmas.

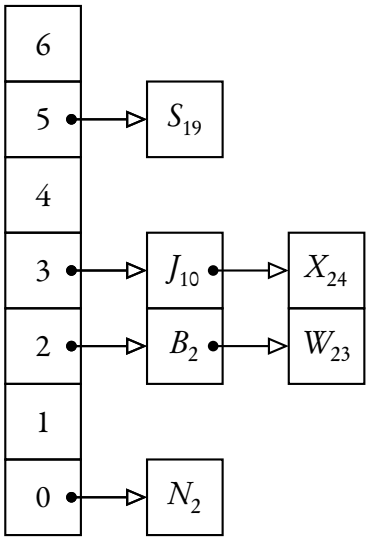

Figure 3: Final result of separate chaining after having inserted the letters B_2, J_{10}, S_{19}, N_{14}, X_{24}, and W_{24}.

[Mitzenmacher and Upfal, 2005] Michael Mitzenmacher and Eli Upfal. *Probability and Computing* Randomized Algorithms and Probabilistic Analysis. Cambridge, 2005.

[Standish, 1994] T.A. Standish. *Data Structures, Algorithms and Software Principles in C*. Addison-Wesley, 1994.

6 For Friday

Study the notes.