Hash Tables

(CISS-111 William Jojo)

Overview

[References are made to Big-O notation in this discussion and you should refer to that document for clarification.]

Arrays will always have a purpose as will linked lists. Linked lists, we have learned, help to overcome some housekeeping limitations of arrays that would become performance issues with large data sets. Linked lists, however, have a scalability issue of their own. Recall that to insert into an ordered list required us to walk the chain of nodes to find the insert point. This alone is a performance issue when we are working with more than a few hundred up to a few thousand nodes.

When we get into the tens of thousands and even a hundred-thousand nodes, the linked list no longer makes sense as tool to solve problems when we are searching the contents in a linear fashion. Remember that with an array of four-billion sorted numbers the binary search could ascertain the presence of a value with a maximum of 32 data values inspected or $O(log_2 n)$.

What if we could take the best of both worlds? Arrays and linked lists in the same data structure? First we need to understand why we would want such a data structure. Imagine we have 125,000 items. We could store all of the items in a sorted array and perform binary searches on the data. At most we need to look at 17 items (2^{17} = 131072) on any one search as we keep cutting the list in half with each comparison. A linked list is clearly out of the running since it would equate to a linear search on unsorted array data and we have no method of dividing a linked list in half. (Actually we *can* divide a linked list in half. It is called a binary tree, but that is a topic for another discussion).

The Hash Table

What if we could take the 125,000 items and place them into a series of buckets? How about 4000 or so buckets? Moreover, what if we had a method that when we select an item at

random from our pile of 125,000 we could definitively identify the correct bucket to which it belongs every time? If we do the math, we are looking at about 31 items per bucker if we use 4000 buckets. Someone now asks us to find a particular item. Since we have a bucket selection method that will always choose the correct bucket, we have narrowed the search to a single bucket which will likely have 31 items in it. We have reduced the number of items to inspect from 125,000 to about 31 before we have inspected any at all. Increase the number of buckets and you can decrease the search list further.

If we can increase the buckets to a number greater than the number of data items, then we begin to possibly achieve finding something in the hash table within one lookup or *O(1)*. In fact, this is the goal of the hash table.

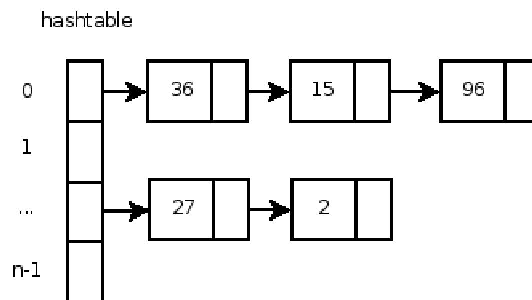

Figure 1: Hash table with chaining as an array of linked lists.

Now imagine we have an array of buckets and each bucket is a linked list as shown in *Figure 1*. This is the essence of a hash table. In this figure we see `hashtable` that is an array of `Node` references of which some or all point to linked lists. The value `n` is used to represent the number of buckets, so our array bounds are from `0` to `(n-1)`.

The method of selecting the correct bucket is known as *hashing* which produces a *hash value*. The hash value is what we use as the index into the array of buckets. Now since our hashing method is likely not to be *perfect* and the because the data set could exceed our number of buckets we need to be prepared for a problem known as a *collision*. Collisions occur because more than one data value hashes to the same bucket. A way to deal with this is called *chaining*. Chaining allows us to place multiple items that hash to the same bucket into a reasonably fast data structure like a small linked list.

Now, when we are asked to find an item in the hash table, we hash the item and this selects the linked list to be searched. Let us turn our attention to the computation of hash values and obtaining consistency.

Hash Values

A hash value is typically nothing more than a number. The number does not necessarily have meaning. However, it must be *consistent*. In other words, the hash value *could* be an index into an array. On the other hand, in a language like Java it could be a hexadecimal value that is some function of the memory location in which an object lives. Regardless of the value the same data passed through the hash program must produce the same hash value every time.

Hashing typically has the best possible distribution among buckets when the number of buckets used is prime numbers. It is often preferable that the prime is as far away from the two adjacent powers of two. Of course, we are free to pick whatever value we want, but we are trying to get the best spread of items to buckets without *clustering*. Clustering occurs when items collect in adjacent buckets in an unusual way. That is, several consecutive buckets may have 50-60 items then the next few buckets have only 10-20 followed by the next 10 buckets having 40-70. With such an uneven spread of items, the hashing method likely needs reworking or a better prime number of buckets.

Now that we have discussed some of the math, what make a good bucket count? Well, it is often a tradeoff of performance and memory consumption. Too small a number and our linear searches become very long. Too large a number and we waste space, but some wasted space is often worth it if we can get the performance we need. In other words, it is often accepted practice to have many empty buckets.

A hash for integers can be something like the following:

```
hash = item % 769;
```

We need to keep in mind that 769 is the number of buckets in our example. If we take `item` and find the modulus of it and 769, then we know what bucket it belongs in or more to the point, the `hash`. This is the simplest of examples, but many variations follow the same general

practice.

Table 1 shows some values that have been suggested as good values to start with and to grow a hash table. Why would we want to grow the table? Remember that we want to get as close to *O(1)* as possible, but we will accept some collisions. Moreover, a few more collisions is not the end of the world, nor is it an indication that performance is guaranteed to suffer.

193	389	769	1543
3079	6151	12289	24593
49157	98317	196613	393241
786433	1572869	3145739	6291469
12582917	25165843	50331653	100663319
201326611	402653189	805306457	1610612741

Table 1: Table of good primes for bucket counts of a hash table.

Using an example we will now hash phone numbers and see how many items we will have per bucket. *Example 1* will generate 10,000 random United States style phone numbers in the range 200-0000 through 999-9999 (no country or area code). These are easily represented with integers. The user can input the number of buckets they want to test with. This value is then used to figure out the spread of numbers across the buckets. Since the numbers are random, the selection of the same number of buckets will yield a statistically different spread. each time. The hash value for each phone number is very simple:

```
hash = pnum % buckets;
```

Remember that the hash value here represents the bucket in which we would place the number. Hash values in general do not need to include the modulo portion in the calculation. This could be left to the programmer to implement within the table implementation.

```
import java.util.*;
import java.io.*;

public class PhoneHash {

        public static Scanner kb = new Scanner(System.in);

        public static void main (String[] args) {

                int x, hash, largest=0, smallest, numempty=0;
                int pnum, buckets;
                int[] hist;

                // find out how many buckets to test
                System.out.print("Enter the number of buckets: ");
                buckets = kb.nextInt();

                // build histogram
                hist = new int[buckets];
                System.out.println("Building Histogram\n");
                for (x = 0; x < 10000; x++) {
                        pnum = (int)(Math.random() * 8000000 + 2000000);
                        hash = pnum % buckets;
                        hist[hash]++;
                }

                // calculate bucket depth
                smallest = Integer.MAX_VALUE;
                for ( x = 0; x < buckets; x++ ) {
                        System.out.printf("%d numbers in bucket %d.\n",
                                        hist[x], x);
                        if ( hist[x] == 0 )
                                numempty++;
                        if ( hist[x] > largest )
                                largest = hist[x];
                        if ( hist[x] < smallest )
                                smallest = hist[x];

                }

                System.out.printf("%d Buckets\n", buckets);
                System.out.printf("Deep bucket is %d entries\n", largest);
                System.out.printf("Shallow bucket is %d entries\n", smallest);

        }
}
```

Example 1: Program to generate phone numbers and help figure the best hash size.

We mentioned earlier the possibility of growing a hash table. To grow a table is actually rather straightforward, but does require a bit of forethought. This is demonstrated in *Example 2* with two tables.

```
public class Rehash {

        private static class Node {
                int data;
                Node next;

                private Node(int item) {
                        data = item;
                        next = null;
                }
        }

        public static void main (String[] args) {

                int x, hash;
                int pnum;
                Node[] hashtable1 = new Node[389];
                Node[] hashtable2 = new Node[769];

                System.out.println("Using 389 Buckets");
                System.out.println("Building Table\n");
                for (x = 0; x < 1000; x++) {
                        pnum = (int)(Math.random() * 8000000 + 2000000);
                        hash = pnum % 389;
                        Node n = new Node(pnum);
                        n.next = hashtable1[hash];
                        hashtable1[hash] = n;
                }
                hist(hashtable1);

                System.out.println("\nRehashing into 769 buckets");
                System.out.println("Building Table\n");
                for (Node p : hashtable1) {
                        while (p != null) {
                                hash = p.data % 769;
                                Node n = new Node(p.data);
                                n.next = hashtable2[hash];
                                hashtable2[hash] = n;
                                p = p.next;
                        }
                }
                hashtable1 = null; // send first table to collector
                hist(hashtable2);
        }
```

Example 2: Program to rehash values from 389 buckets to 769 buckets.

It is easy to see in *Example 2* how the rehashing portion could be made into a method. Also, the statistical histogram code has been turned into a method. In fact, it is about time to make this whole process into a class with some appropriate methods.

```
        public static void hist(Node[] ht) {
                int x, hash, largest=0, smallest=Integer.MAX_VALUE;
                int numempty=0;

                // calculate bucket depth
                for (x = 0; x < ht.length; x++) {
                        Node p = ht[x];
                        int l=0;
                        if ( p == null )
                                numempty++;
                        while (p != null) {
                                l++;
                                p = p.next;
                        }
                        System.out.printf("%d numbers in bucket %d.\n",
                                        l, x);
                        if ( l > largest )
                                largest = l;
                        if ( l < smallest )
                                smallest = l;
                }

                System.out.printf("%d Buckets\n", ht.length);
                System.out.printf("Deep bucket is %d entries\n", largest);
                System.out.printf("Shallow bucket is %d entries\n", smallest);
                System.out.printf("Empty buckets: %d\n", numempty);
        }
}
```

Example 2: Rehashing example continued.

The class in *Example 3* shows the `HashTable` class and some very useful methods. These methods complete the ability to add and remove items, check for the presence of an item and to rehash a table. Keep in mind that the Java `HashSet` class has very similar functionality, but uses a `Set` class for storage. Moreover, the `Hashtable` in Java implements a tuple-based key/value pair and is implemented as a `Map` object.

Our implementation of the `HashTable` class is very simple and straightforward and can easily be adapted to include an iterator. Note that our calculation of the hash value includes performing:

```
item % BUCKETS;
```

Again, this is not required, be we have chosen to do it here to more tightly bind the `hashValue()` method to the object.

```
public class HashTable {

        private Node[] buckets;
        private static int BUCKETS;

        private class Node {
                int data;
                Node next;

                private Node(int item) {
                        data = item;
                        next = null;
                }
        }

        public HashTable(int numBuckets) {
                BUCKETS = numBuckets;
                buckets = new Node[BUCKETS];
        }

        public static int hashValue(int item) {
                if ( item < 0 )
                        item = -item;
                return (item % BUCKETS);
        }

        public void add(int item) {
                int h;
                Node n;

                h = hashValue(item);
                n = new Node(item);
                n.next = buckets[h];
                buckets[h] = n;
        }

        public boolean contains(int item) {
                int h;
                Node p;

                h = hashValue(item);
                p = buckets[h];
                while ( p != null ) {
                        if ( p.data == item )
                                return true;
                        p = p.next;
                }

                return false;
        }
```

Example 3: Complete HashTable class with support for add, remove, contains and rehash.

```
	public boolean remove(int item) {
		int h;
		Node cur, back=null;

		h = hashValue(item);
		cur = buckets[h];
		while ( cur != null ) {
			if (cur.data == item) {
				if ( back == null )
					buckets[h] = cur.next;
				else
					back.next = cur.next;
				return true;
			}
			back = cur;
			cur = cur.next;
		}
		return false;
	}

	public void rehash(int numBuckets) {
		int h;
		Node [] newtable = new Node[numBuckets];

		for (Node p : buckets) {
			while (p != null) {
				h = p.data % numBuckets;
				Node n = new Node(p.data);
				n.next = newtable[h];
				newtable[h] = n;
				p = p.next;
			}
		}
		buckets = newtable;
		BUCKETS = numBuckets;
	}
}
```

Example 3: Complete implementation of HashTable class, continued

One final item to discuss is the hashing of data that is not as simple as an integer. For example, a string would require a bit more processing to enable hashing and placement into the table. This can be accomplished by starting with a hash value of zero and for each character in the string multiply the current hash value by 31 (or 37) then add the current character. This results in the following mathematical sequence assuming we have some string `s`:

$$s[0] * 31^{n-1} + s[1] * 31^{n-2} + ... + s[n-1]$$

In Java we can code this up the like so:

```
String s;
int x, n, h=0;

n = s.length();
for (x = 0; x < n; x++)
   h = h * 31 + s.charAt(x);
```

Once h is computed we simply take that value modulo the number of buckets.

```
return h % NUMBUCKETS;
```

This is no way suggests that this is the only or perfect way to hash strings. It is, however, quite simple to implement and is relatively fast. This algorithm can also produce negative hash values, so you may need to change the sign on the result to avoid an `ArrayIndexOutOfBoundsException`. We must be prepared to deal with strings of differing case as they will not hash to the same value. This would require us to either present the string in a normalized form or require the hashing method to do this automatically.