HASH TABLES

1. INTRODUCTION & OVERVIEW

Having now looked at arrays, linked lists, stack, queues and trees we will conclude this series of lectures by looking at hash tables.

The efficiency of storage, retrieval and sorting has been discussed elsewhere and not dealt with in detail when discussing the earlier data structures. Now, however, it becomes one of the main reasons for introducing this final data structure.

We finished discussion of each of our earlier structures by moving for array-based to linked list-based implementations; amongst other benefits, these eliminated the wastage of unoccupied storage while providing an ability to order data. Hash tables, on the other hand, make a virtue of wasted storage and non-ordered storage, and trading these off against large efficiency gains for data access.

1.1 Dictionaries and Skip Lists

The term dictionary can be used to describe a collection of pairs \((k,e)\) where:

- \(k\) = a key (used for looking up data, as in a database); and
- \(e\) = the data which is to be operated on (eg, with a get, put or remove).

Data of this type can be stored in, eg, linked lists or trees. A linked list can be slow for a get and a tree can be awkward to 'balance' (eg, an AVL or red-black tree). Some improvements can be made with the compromise of a skip list. In brief, this uses intermediate pointers to allow a search to 'skip' to the middle of a list if the element being looked for is in the upper half of that list; commonly, several such pointers can be used to further speed up searches. This is, however, simply a way of enhancing data structures which we have already discussed.

2. HASHING

Let us start by going all the way back to storing data in an array, say information about students. If we have a single, small class and the students each have an enrolment number (eg, in the range 1 to 40) then we can store the data in an array and use the enrolment number as an index (assuming for now that the elements have indices in the range 1 .. 40, not the actual 0 .. 39). It could be that there are 7 such classes and our students have enrolment numbers in the range 241 .. 280. We could declare a 280-element array and only use 40 of its elements for our class but a better method would be to stay with a 40-element array and calculate the index (or dictionary key) using the mapping:

\[
\text{index} = \text{enrolment number} - 240.
\]
2.1 Hash Functions

Imagine that we had a university club or society open to all students from all years. We could use the students' matriculation number as the key but, to allow for all possibilities, with would mean a number which was 8 digits long, such as 99123456, and we would need an array holding ten million elements (each element possibly holding many bytes of data).

This is excessive and unnecessary. If the society expects no more than 100 members (a 3rd Division Football team fan club?) then we could simply use the last two digits of the matric number as the key. Information about student 99123456 would be held in array element 56 and that for student 9811111 would be held in array element 11 but what about student 9911111? Wouldn't s/he occupy the same data space as 9811111? (Yes, but we discuss that soon under linear probing!).

The mapping which we would use here would be to take the remainder from dividing the matric number (mn) by 100, ie:  \( key = mn \% 100 \).

Not surprisingly, this is called the division-remainder technique and it is an example of a uniform hash function in that it distributes the keys (roughly) evenly into the hash table. This operation (mod or Java '%'') results in key values in the range 0 .. 99 (assuming now that our arrays start at index 0).

If we wanted to use the students' surnames as the key then the problem becomes even more extreme but it is solved by similar methods. Suppose that we used the first three characters from the surname.

If we treat these as three 8-bit integers and multiply them together we get a 24-bit integer which we can then deal with as shown in Figure 1. [Note: Although Java uses 16-bit Unicode for characters, the bottom 8 bits are the same as the ASCII character set.]

```java
public static long hash(String name, int tableSize) {
    // get bottom 8 bits of first char
    int tmp = name.charAt(0) % 256;

    // now multiply by next two 8-bit chars
    tmp = tmp * (name.charAt(1) % 256);
    tmp = tmp * (name.charAt(2) % 256);

    // Now return an index within table's bounds
    return (tmp % tableSize);
}
```

Fig. 1: Method hash()

Other techniques for creating a hash function include, eg:

- **Folding** can be used, whereby a long number (eg phone number) can be treated as several shorter number which are then added to give a code (shift folding), possibly first reversing the digits of one of the short numbers (boundary folding);
- **A random number generator** using a particular parameter value as a 'seed' to generate the next random number in its sequence. The random number in the range 0.0 .. 1.0 can then be normalised and truncated to give the required key.
2.2 Collisions, Buckets and Slots

If no two values are able to map into the one table position then we have what is called ideal hashing. For hash function $f$, each key, $k$, maps into position $f(k)$. In searching for an element, we compute its expected position: if there is an element present then we operate on it; if not then the element was not present in the table.

Each of our earlier examples suffers from a major problem, however: students with, say, the same last three matric number digits (or the first three characters in their name) will generate the same hash code (ie, array index). If we try to use the same position to store data for both of them, we have a collision.

We will now describe each position in the table as a bucket with position $f(k)$ being the home bucket (the number of buckets in a table equals the table length). Since several keys can map into the same bucket, it is possible to design buckets with several slots (this is common for disk-stored hash trees), each slot holding a different value. We will generally have just one slot per bucket, although the linked list approach is different again.

People solve the 'collision' problem on a regular basis when visiting the cinema or sporting event! If you are allocated seat number 47 in a half-full stadium and, on arrival, you discover someone sitting there (but the next seat is free), what would you do? You could ask the usurper to move … or you could (calmly!) take the next available seat. But what if a complete stranger was looking for you and only knew your ticket number? If they had been told that you were a placid, reasonable person and discovered seat 47 was otherwise occupied then they might keep walking by the higher-numbered seats asking the occupants their ticket numbers until they found you (or discovered that you weren't in the stadium, after also checking seats 1 .. 46).

3. LINEAR PROBING

The above process is called linear probing. Taking the cinema/stadium analogy a bit further, what if you sold 1000 tickets (numbered 0 .. 999) and then discovered that no more than 400 people were going to attend, causing you to move the event to, say, a 500-seat capacity venue? If you take the ticket number ($t$) and find its value mod 400 ($t \% 400$) then you will have a new ticket number ($n$) in the range 0 .. 399. Up to three people (eg, tickets 73, 473 and 873) may be given the same value of $n$ but they will still find a seat somewhere using linear probing.

A simple numerical example would be where we wished to insert data with keys 28, 19, 59, 68, 89 into a table of size 10.

These entries are made in the table with the following decisions, as shown in Figure 2:
(i) Enter 28 \( \rightarrow 28 \mod 10 = 8; \) position is free, 28 placed in element 8;
(ii) Enter 19 \( \rightarrow 19 \mod 10 = 9; \) position is free, 19 placed in element 9;
(iii) Enter 59 \( \rightarrow 59 \mod 10 = 9; \) position is occupied; try next place in table (using wraparound), ie 0; 59 placed in element 0;
(iv) Enter 68 \( \rightarrow 68 \mod 10 = 8; \) position is occupied; try next place in table (0) - occupied; try next (1); 68 placed in element 1;
(v) Enter 89 \( \rightarrow 89 \mod 10 = 9; \) position is occupied; next two places in table are occupied; try next (2); 89 placed in element 2.

3.1 Finding and Deleting Data

Finding data is not too difficult: given a particular key, \( k \), we look for an element in the home bucket, \( f(k) \). If the data is not there then we keep looking further up the table (using wraparound) until we find it or have ended up back at the home bucket again (ie, the data is not in the table).

Deleting data is trickier. Having first found the data as described, we cannot simply reinitialize the table element. Consider a deletion of 59 from Fig. 2(v). If we found 59 in position 0 and reinitialized that position then, in looking later for 68, we would look first in its home bucket (position 8) then position 9 then 0 - but position zero is now empty so we would (wrongly) exit the search loop. Reordering the table is an option - but a costly one! To avoid this problem, a process of lazy deletion may be used whereby each element of the array has an additional Boolean data member which may be marked used or not used.
4. DESIGNING A GOOD HASH FUNCTION

If we use the division-remainder method then we should take care in the choice of the remainder to use. In particular, we would want to minimise the effects of clustering.

4.1 Clustering

Think back to our stadium example where we switched to a venue with a capacity of 500 because only 400 people were expected. It could get even trickier if the number turning up is 40 and you move to a venue of capacity 50. Up to 20 people could be seeking the same seat and 19 of them would need to find an alternative seat (ie the next higher numbered one which was free). A situation often arises where several keys map into the same home bucket and, through linear probing, have to seek another higher-numbered bucket. The resultant bunching together of elements is one example of what is called clustering and we would hope to avoid the multiple collisions it causes - especially as one cluster can easily merge with another cluster.

4.2 Choice of Divisor

Not all hashing techniques result in an even distribution of keys and it is quite common to have an imbalance between odd and even keys. If we are using the division-remainder method with \( f(k) = k \mod b \) then consider the following:

- If \( b \) is even and there are more even than odd values of \( k \) then \( f(k) \) will produce an excess of even values; similarly, more odd than even values of \( k \) would produce an excess of odd results for \( f(k) \);
- If \( b \) is odd then either kind of excess of \( k \) values would still give a balanced distribution of even/odd results.

Clearly, the divisor should be an odd number. Also, if \( b \) itself is divisible by a small odd number (eg 3, 5, 7) then the results end up less well distributed. Ideally, \( b \) should be a prime number but, if no such prime is available near our table size then we should use an odd number which has no small factors (in the range, say, of 3 .. 19).

Fig. 3 (from Sahni\(^1\)) shows a possible implementation of a hash table in Java. The type HashEntry is yet to be defined elsewhere but it will contain a key, an Object and a field used for lazy deletion. Methods are needed to implement search, get and put operations.

```java
public class HashTable {
    //------------- attributes -------------
    protected int divisor;
    protected HashEntry[] table;
    protected int size;

    //------------- constructor -------------
    HashTable (int theDivisor)
    {
        divisor = theDivisor;
        size = 0;
        table = new HashEntry[divisor];
    }
}
```

Fig. 3: class HashTable()

---


Greg McCarra HashTables (Lec 4) - P.5 Rev.1.1, 16/05/02
5. **ANALYSIS of LINEAR PROBING**

Let us assume for now that we are using a large hash table and that each probe (search, get or put) is independent of the previous one [we shall see that this is a false assumption but read on …].

The proportion of the table that is full is called a **load factor**, usually represented by the symbol \( \lambda \) (lambda). So, a 40 element table holding only 10 elements has a \( \lambda \) of 0.25 - which is also the probability of finding an occupied cell. Therefore the probability of finding an empty cell during a probe is \((1 - \lambda)\). And the expected number of probes needed to find an empty element is \(1/(1 - \lambda)\). [Think of rolling dice - if the probability of throwing a three is 1/6 then the average number of rolls needed to get a three result is 6 rolls. Also, each roll is independent of the last - with four consecutive twos, the chances of a two next time are still only 1/6.]

For example, if our table is a fifth full then you would expect an average of 1.25 probes (= \(1/(1 - 0.20)\)) to find an empty element. If it is nine-tenths full, ten probes would be expected on average.

Unfortunately, this is not true! Clustering means that one failed insertion in a bucket (causing an insertion further up the table) has a knock-on effect on other attempted insertions. Knuth\(^2\) has demonstrated that the **actual** number of probes needed is:

\[
(1 + 1/(1 - \lambda)^2)/2
\]

Recalculating our examples above, we now have that
- For a fifth-full table, we expect an average of 1.28 probes (not 1.25, but close)
- For a half-full table, we expect an average of 2.5 probes (not 2.0, but close-ish)
- For a nine-tenths full table, we expect an average of 50.5 probes (nothing like the 10.0 probes predicted earlier).

Most of these problems are caused by clustering. The type which we have discussed so far is actually called **primary clustering**. We can alleviate many of its problems by moving from linear probing to **quadratic probing**.

6. **QUADRATIC PROBING**

Having calculated our bucket with \( f(k) = k \% b \) then, with linear probing, we checked position \( f(k) + 1 \) for availability, then \( f(k) + 2 \) and so on until we found a free position.

**Quadratic probing** uses the search \( f(k) + i^2 \), then \( f(k) + 2^2 \) then \( f(k) + 3^2 \), etc, (hence the term ‘quadratic’). Stepping on distances of 1, 4, 9, … rather than 1, 2, 3, … greatly reduces the incidence of primary clustering [but introduces **secondary** clustering which it is less problematic - more later!].

Let us try a numerical example similar to that of Fig. 2, this time using quadratic probing. Fig. 4 shows where the values end up with quadratic probing and the decisions involved are:

(i) Enter 28 \( \Rightarrow 28 \% 10 = 8 \); position is free, 28 placed in element 8 (result is same as before);
(ii) Enter 19 \( \Rightarrow 19 \% 10 = 9 \); position is free, 19 placed in element 9 (result is same as before);
(iii) Enter 59 \( \Rightarrow 59 \% 10 = 9 \); position is occupied; try 1 (=1^2) places along in table (using wraparound), ie 0; 59 placed in element 0 (result is same as before);
(iv) Enter 68 \( \Rightarrow 68 \% 10 = 8 \); position is occupied; try next place in table (0) - occupied; try next (1); 68 placed in element 1 (result is same as before for same reasons as iii);
(v) Enter 89 \( \Rightarrow 89 \% 10 = 9 \); position is occupied; previously we tried positions 0 and 1 then put data in 2. This time, we try 0 then go directly to 3, jumping over the existing cluster.

<table>
<thead>
<tr>
<th>Index:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>(ii)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>28</td>
<td>19</td>
</tr>
<tr>
<td>(iii)</td>
<td>59</td>
<td></td>
<td></td>
<td></td>
<td>28</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(iv)</td>
<td>59</td>
<td>68</td>
<td></td>
<td></td>
<td>28</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(v)</td>
<td>59</td>
<td>68</td>
<td>89</td>
<td></td>
<td>28</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Quadratic probing

6.1 Effectiveness of Quadratic Probing

This brief example - which only saves one collision - might not seem proof of effectiveness but a full statistical analysis of quadratic probing shows its benefits over linear probing.
It can be shown\(^3\) that, if the table is less than half full and its size is prime, then quadratic probing guarantees that a new element can always be inserted into the table and no cell is probed twice in the process.

The fact that we are adding \emph{squares} to location numbers instead of just incrementing them might suggest that there is a much larger computational cost from squaring. This is not so, as simpler processes can be used.

In a table of size \(S\), if our home bucket is called \(P_0\) then the our \(i\)-th probe after that will be:

\[
P_i = P_0 + i^2 \pmod{S}
\]

and the one prior to that will be:

\[
P_{i-1} = P_0 + (i-1)^2 \pmod{S}
\]

Subtracting the second equation from the first gives:

\[
P_i - P_{i-1} = i^2 - (i-1)^2 = (2i-1) \pmod{S}
\]

Therefore

\[
P_i = P_{i-1} + (2i-1) \pmod{S}
\]

Now we are dealing with a multiplication by 2 and adding 1. Multiplication by 2 is simply a shift-left of a binary number and an increment is a very fast operation for a microprocessor. The \((\pmod{S})\) is still needed to deal with wraparound, but it is simply dealt with by subtracting \(S\) as and when the element index overshoots the table's last element.

### 6.2 Secondary Clustering

As an exercise, consider entering the values 3, 13, 23, 33 and 43 into Figure 2 using linear probing and then into Figure 4 using linear probing. Although the \emph{places} where collisions will differ, the \emph{amount} of collisions will be the same. This is called secondary clustering. Fortunately, simulations show that the average cost of this effect is only about half of an extra probe for each search and then only for high load factors. One method of resolving secondary clustering is \emph{double hashing} which calculates the distance away of the next probe by using another hash function (instead of the linear +1 or quadratic +\(i^2\))

In general, the only main drawback of quadratic hashing is the need to ensure that the table is no more than half full (ie, \(\lambda < 0.5\)). This is usually a reasonable price to pay.

---

\(^3\) Weiss, MA, "Data Structures and Problem Solving using Java", Addison-Wesley, 2002
7. SEPARATE CHAINING HASHING

Earlier in section 2.2 we introduced the term slots which allow several keys to map into the same bucket. These are commonly used in disk storage and they can be implemented using arrays, as long as dynamic array expansion (ie array doubling) is allowed. As with our earlier discussions of array, however, we find that linked lists are often a more versatile solution. Consider the (rather extreme) example where we were trying to place the following ten values into a hash table with divisor 13: 42, 39, 57, 3, 18, 5, 67, 13, 70 and 26.

With linear probing, this would result in the table shown in Figure 5(a). Now, instead, we will make each bucket simply the starting point of a linked list (initialized to null). Strictly, this means zero slots but effectively we can have a large and variable number of slots. The first element placed there is put in a node pointed to by the bucket. Each time we have a collision at that bucket, we simply add another node in the chain as shown in Figure 5(b), where elements underlined have been placed outside their home bucket:

A slightly more sensible way to implement this would be to have the list sorted by key value using structures like those discussed earlier in this series of lectures.

To find an element with key \( k \), we would calculate the home bucket (\( f(k) = k \mod d \)) and then search along the chain until we found the element.
Tutorial Exercise - 4

4_1 A fundraiser has sold raffle tickets in the range 250 – 749. Ten winners - drawn at random - will be allocated seats at a concert, the seat numbers being (strangely!) 0 through to 9.
   a) Derive a hash function which could be used to allocate the seat numbers (assuming that linear probing was also being used);
   b) How would the experience of the first two arriving winners differ from that of the last two?

4_2 A hash table is to be created with room for 11 data. Assuming linear probing is used:
   a) Allocate data with the following keys to the table (arriving in the order shown): 81, 20, 34, 42, 21, 45;
   b) Re-do the table, assume that the data is arriving in ascending order.

4_3 Using your table from exercise 4_2(a)
   a) Describe how data element with key 42 would be deleted from the table;
   b) What would then be the result of a search for key 21?

4_4 Explain which of the following divisors could be described as ‘good’:
   8, 17, 47, 49, 103, 105

4_5 Referring to lecture notes section 5, describe how the experience of the 5th arriving winner would probably differ from that of the 3rd.

4_6 Re-do exercise 4_2(a), this time using quadratic probing.

4_7 Re-do exercise 4_2(a), this time using a chained hash table.