CITS2200 Data Structures and Algorithms

Topic 19

Hash Tables

- Introduction to hashing — basic ideas
- Hash functions
  - properties, 2-universal functions, hashing non-integers
- Collision resolution
  - bucketing and separate chaining
  - open addressing
  - dynamic tables — linear hashing

Reading: Lambert and Osbourne, Section 13.2

1. Introduction

The Table is one of the most commonly used data structures — central to databases and related information systems.

In the previous section, we briefly examined a List representation for tables, but this had linear time access.

Can we do better?

We have seen a number of situations where constant time access to data can be achieved by indexing directly into a block...

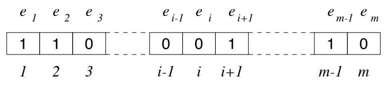
eg. Array uses an addressing function

$$\alpha(i,j) = (i-1) \times n + j \qquad 1 \le i \le m,\ 1 \le j \le n$$

	a	b	c	d	e
a	1	2	3	4	5
b	6	7	8	9	10
c	11	12	13	14	15
d	16	17	18	19	20

eg. Set uses a characteristic function...

$$f(e) = \begin{cases} \mathit{true} \ \ (\text{or } 1) & e \in A \\ \mathit{false} \ \ (\text{or } 0) & \text{otherwise} \end{cases}$$



These approaches:

- often sacrifice space for time — space wasted by all the "holes"
- rely on the ordering of elements, which translates to an ordering of the memory block.

Can we improve on these?

- more compact use of space
- applicable to unordered information (eg Table)

$\Rightarrow$ *hashing...*

2. Basic Hashing

The direct indexing approaches above work by:

- setting aside a big enough block for all possible data items
- spacing these so that the address of any item can be found by a simple calculation from its ordinality

What if we use a block which is not big enough for all possible items?

- Addressing function must map all items into this space.
- Some items may get mapped to the same position $\Rightarrow$ called a *collision*.

Example

Suppose we fill out our Lotto coupons as follows. Each time we notice a positive integer in our travels, we calculate its remainder modulo 45 and add that to our coupon. . .

The first thing we see is a pizza brochure containing the numbers 165, 93898500, 2, 13, 1690. These map to positions 30, 15, 2, 13, 25.

This fills 5 positions in our data store. . .

0	1	$\cancel{2}$	3	4	5	6	7	8
9	10	11	12	$\cancel{13}$	14	$\cancel{15}$	16	17
18	19	20	21	22	23	24	$\cancel{25}$	26
27	28	29	$\cancel{30}$	31	32	33	34	35
36	37	38	39	40	41	42	43	44

Next we ring up to order our Greek Vegetarian, and we're told it'll be ready in 15!
$\Rightarrow$ *collision*

Collisions

When two entries "hash" to the same key, we have a *collision*.

We need a method for dealing with this. However. . .

Advantages:

- Once we allow collisions we have much more freedom in choosing an addressing function.
- It no longer matters whether we know an ordering over the items.

Exercise

Obtain an address from your name into the block 1. . . 10 as follows:

- Count up the number of letters in your name.
- Add 1.
- Double it.
- Add 1.
- Double it.
- Subtract the number of letters in your name.
- Add the digits in your current number together.
- Square it.
- Add the digits in your number together.

What address did you get?

Such a function is called a *hash function*. It takes the item to be looked up or stored and "hashes it" into an address in the block, or *hash table*.

Hashing is used in both data management (via hash tables) and security and cryptography (for example MD5 checksums).

We will consider hash functions in more detail, and then consider methods for dealing with collisions.

3. Hash Functions

To begin with, we'll assume that the element (or key) to be hashed is an integer. We need a function

$$h : \mathcal{N} \to 0 \ldots m-1$$

that maps it into a hash table `block` of size m.

Thus, to store a table t, we would set

$$\texttt{block}[h(a)] = a$$

for all elements a in t, and fill the other elements of `block` with null references.

We call $h(a)$ the *home address* of a.

3.1 Properties of Hash Functions

A hash function that maps each item to a unique position is called *perfect*. Note that if we had an infinitely large storage block we could always design a perfect hash function.

Since our hash functions will generally not be perfect, we want a function that distributes evenly over the hash table — that is, one that is not *biased*.

Q: What is an example of a "worst case" hash function in terms of bias?

3.2 2-universal Functions

In the Lotto example earlier, we used the hash function

$$h(i) = \quad i \bmod 45.$$

A commonly used class of hash functions, called *2-universal* functions, extends this idea...

A *2-universal* hash function has the form

$$h(i) = ((c_1 i + c_2) \bmod p) \bmod m$$

where m is the size of the hash table, $p > m$ is a large prime ($p > 2^{20}$), $c_1 < p$ is a positive integer, and $c_2 < p$ is a non-negative integer.

— $(c_1 i + c_2) \bmod p$: "scrambles" i

— $\bmod m$: maps into the block

Small changes (eg to c_1) lead to completely different hash functions.

Another 2-universal hash function that can be used in languages with bit-string operations...

Assume items are b-bit strings for some $b > 0$, and $m = 2^l$ for some $l > 0$. The *multiplicative hash function* has the form:

$$h(i) = (a\,i \bmod 2^b) \operatorname{div} 2^{b-1}$$

for odd a such that $0 < a \leq 2^b - 1$.

Advantage — *mod* and *div* operations can be evaluated by shifting rather than integer division $\Rightarrow$ very quick

3.3 Hashing Non-integers

Non-integers are generally mapped (hashed!) to integers before applying the hash functions mentioned earlier.

Example

Assume we have a program with 3 variables

```
float abc, abd, bad;
```

and we wish to hash to a location to store their values. We could obtain an integer from:

- the length of each word (as in the earlier example) — will lead to a lot of collisions
  $\Rightarrow$ in this case all variables will hash to the same location
- the ordinality of the first character of each word — fewer collisions, but still poor
  $\Rightarrow$ the first two will hash to the same location

- summing the ordinality of all characters — likely to be even fewer collisions
  $\Rightarrow$ but here, the last two will collide
- "weight" the characters differently, eg 3 times the first plus two times the second plus one times the third — collisions will be much rarer (but may still occur)
  $\Rightarrow$ no collisions in this set

Extending the weighting idea, a typical hash function for strings is to treat the characters as digits of an integer to some base b.

Assume we have a character string $s_1 s_2 \ldots s_k$. Then, we calculate

$$[ord(s_1).b^{k-1} + ord(s_2).b^{k-2} + \ldots + ord(s_k)b^0] \bmod 2^B$$

Here:

- b is a small odd number, such as 37...
- $\bmod 2^B$ gives the least significant B bits of the result — eg 16 or 32.

Q: Why the *least* significant bits?

4. Collision Resolution Techniques

There are many variations on collision resolution techniques. We consider examples of three common types.

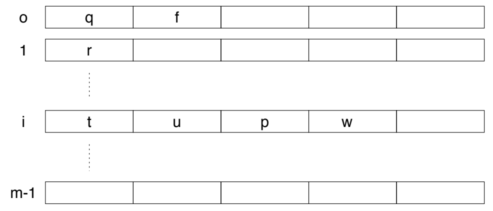
4.1 Bucketing and Separate Chaining

The simplest solution to the collision problem is to allow more than one item to be stored at each position in the hash table

⇒ associate a List with each hash table cell...

Bucketing

— each list is represented by a (fixed size) block.



"Advantage"

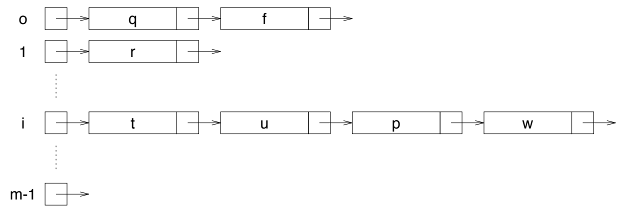
- Simple to implement — hash to address then search list.

Disadvantages

- Searching the List slows down Table access.
- Fixed size ⇒ may waste a lot of space (both in hash table and buckets).
- Buckets may *overflow!* ⇒ back where we started (a collision is just an overflow with a bucket size of 1).

Separate Chaining (Variable Size Bucketing)

— each List is represented by linked list or *chain*.



Advantages

- Simple to implement.
- No overflow.

Disadvantages

- Searching the List slows down Table access.
- Extra space for pointers (if we are storing records of information the space used by pointers will generally be small compared to the total space used).
- Performance deteriorates as chain lengths increase.

Performance

Worst case for separate chaining $\Rightarrow$ all items stored in a single chain. Worst case performance same as List: $O(n)$ — nothing gained!

But *expected case* performance is much better...

The *load factor* λ of a hash table is the number of items in the table divided by the size m of the table.

Assume that each entry in a hash table is equally likely to be accessed, and that each sequence of n insertions is equally likely to occur. Then a hash table that uses separate chaining and has load factor λ has the following expected case performance:

- $s(\lambda) = 2 + \lambda/2$ *probes (read accesses) for successful search*
- $u(\lambda) = 2 + \lambda$ *probes for unsuccessful search*

See Wood, Section 9.2.

4.2 Open Addressing

Separate chaining (and bucketing) require additional space. Yet there will normally be space in the table that is wasted.

Alternative $\Rightarrow$ *open addressing* methods

- store all items in the hash table
- deal with collisions by incrementing hash table index, with wrap-around

Linear probing — increment hash index by one (with wrap-around) until the item, or *null*, is found.

Problem — items tend to "cluster".

Double hashing — increment hash index using an "increment hash function"! $\Rightarrow$ may jump to anywhere in table.

Advantages

- All space in the hash table can be used.

Disadvantages

- Insertions limited by size of table.
- Deletions are problematic. . .

Deleting items means others may not be able to be reached — requires reorganizing table, or marking (flagging) items as deleted.

The latter is most common, but means erosion of space in the hash table.

4.3 Dynamic Tables — Linear Hashing

Finally, there are methods that consider the hash table to be dynamic rather than static!

Linear hashing is an extension of separate chaining — rather than allowing the variable-length buckets (chains) to grow indefinitely, we limit the average size of the buckets.

- Insertions: if the average chain size exceeds a predefined upper bound, split the "next" unsplit bucket, and hash the items in the bucket by a function with double the previous base (ie m, $2m$, $4m$,. . .).
- Deletions: if the average bucket size drops below a predefined minimum and the table is no smaller than the original table, shrink the table.

Effectively, we use two hash functions parameterised by $split$ (the split point) and M (roughly the table size). Assume h is the hash function that maps keys to large integers.

Then, the hashed address is:

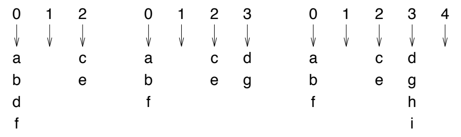
$$H(k) = \left\{ \begin{array}{ll} h(k) \bmod 2M, & \text{if } h(k) \bmod M < split \\ h(k) \bmod M, & \text{otherwise} \end{array} \right\}$$

If the loading factor becomes too high (eg more than $0.8M$ elements in the table), then:

1. $split = split + 1$
2. rehash all items in bucket $split - 1$
3. if $split = M$, set $split = 0$ and $M = 2M$.

Example

Assume table is initially of size 3, and maximum loading is 2. . .



Disadvantages

- (More complicated to code.)
- Requires movement of items.

Advantages

- Maximum load is maintained — improves expected efficiency.
- Insertions — no overflow, not bounded by size of table.
- Deletions — no erosion.

This approach is used by `java.util.Hashtable` — have a look at its API documentation.

5. Summary

Hash tables can be used to:

- improve the space requirements of some ADTs for which bounded representations are suitable
- improve the time efficiency of some ADTs, such as Table, which require unbounded representations

We have seen a number of methods for collision resolution in hash tables:

- bucketing and separate chaining
- open addressing, including linear probing and double hashing
- dynamic methods, such as linear hashing

6. Caution!

Note that while performance can be very good, *this is not a panacea!* For many applications, such as those naturally represented by trees, hashing would lose the structure.

ie. *don't "hash first, ask questions later"*

(Hash can be addictive!)