Hashing

The entire second half of this course has been (more or less) dedicated to finding records in files. We started with O(N) and improved to $O(\log_2 N)$ and then to $O(\log_K N)$. How can we possibly improve on that?

Topic	***Folk & Zoellick***
• Direct Mapping	§ N/A
• Hashing	§ 10.1
• Collisions	§§ 10.1.2, 10.2, 10.3.2
• Collision Resolution	§§ 10.5, 10.6, 10.8

Access Progress

Here are the techniques we know for finding a record with given key value in some file:

Technique	***Clusters Read***	***N=262144, K=64***
Sequential Search	O(N/K)	4096
Binary Search	$O(\log_2 N)$	18
Paged Binary Trees	$O(\log_K N)$	3
B-Trees	$O(\log_K N)$	3

☐

Q: What could possibly be better than $O(\log_K N)$?

A:

Q: Ya right!

A:

Direct Mapping

All those techniques work the same way: we know some key value *k*. In order to find the record with that key value, we have to search through some file. It may be the data file, it may be an index file, it may be a B-tree. But we still have to search for *k*.

What if we already knew where k *was in the file?*

Here's an example. We have a data file full of student records. The primary key is student number. We're looking for the record of the student with student number 488652.

Q: Wouldn't that be cool if 488652's record were 488,652 bytes into the file?

A: You're darn tootin' it'd be cool.

The Key is the Address

In the example, we store the student record of each student (with student number s_i) at an offset of s_i bytes into the file.

Now if we’re looking for the record for student s_i we just do a seek to position s_i from the beginning of the file.

Q: What’s wrong with this picture?

A:

The Key is the RRN

Student numbers are too close together to be used as byte offsets. In fact, in the Winter 1999 offerings of CSI 2131, there are 14 students whose student numbers are within 100 of another student's number. There are 4 students within 20.

In general, it's possible for two students to have numbers that differ by only one.

But what if we had fixed length records. Instead of storing the record for a student with student number s_i at an offset of s_i bytes into the file, we store the record as RRN s_i.

Q: What's wrong with *this* picture?

A:

Ok, maybe this isn't such a good idea (although, it doesn't seem like that bad an idea either, now that I type it).

Hashing

Storing records in positions in a file based on the value of their keys is known as *hashing*. In our example, we used student number as RRN. This is about the simplest form of hashing there is: using the primary key itself as the RRN. But there are several problems:

- not all primary keys can be used *as is* for RRN
    - remember `BOYOTT001`?
- the *keyspace* is usually much bigger than N
    - are there really 10,000,000 student records in the U of O database?
- the keyspace may also be too *small*
    - a few years ago student numbers were only six digits at U of O

So instead of using the primary key as RRN, we usually *do something* to the primary key to turn it into a good RRN. This *doing something* is called a *hash function H(k)*.

Did you know?
The term *hashing* comes from the Latin verb `hashio-hashere-heshi-hashitum` meaning *to humble, to crush, to humiliate, to store records in predictable positions in a file to allow direct access, to obliterate*.

Hashing Functions

Let's go back to an old example.

If we believe that 100 records is enough to store all the records for a while, we could use the last two digits of the key as the RRN. In this case our hashing function is

$$\boxed{H(k) = last\_two\_digits(k)}$$

Cor351614	...
Mil155900	...
Lam643080	...
Har018618	...
Ste970555	...
Fei134681	...
Red496173	...
Blu142224	...
Pnu789164	...
Eng466879	...
Gra823015	...

Mil155900	...
⋮	
Cor351614	...
Gra823015	...
⋮	
Har018618	...
⋮	
Blu142224	...
⋮	
Ste970555	...
⋮	
Pnu789164	...
⋮	
Red496173	...
⋮	
Eng466879	...
Lam643080	...
Fei134681	...
⋮	

Q: *Now* what's the problem?

A:

Perfect Hashing Functions

If

$$\forall k,j\ (H(k) \neq H(j)) \rightarrow (k \neq j)$$

Then we say *H(k)* is *perfect*.

In our example it was pure luck that no two keys had the same last two digits. In general, it is very hard to find a perfect hashing function.

□

Q: How hard is it?

A: So hard that we're not even going to try

Q: If we don't try to build a perfect hashing function isn't it possible that $\exists k,j\ k \neq j$ AND $H(k) = H(j)$?

A:

Collisions

Hashing Function

A formula that maps key values to reference values that represent the position of a record in a file.

Hash Value

A reference value produced by the hashing function

- In our example hash values were used as RRNs

Hash Table

The file containing the records in positions determined by the hashing function.

☐

In general we are not guaranteed to have a hashing function that maps distinct keys onto distinct hash values. Instead of trying to fix the function, we’ll let the function be imperfect and deal with the results.

When a given hashing function maps two different keys onto the same hash value we call it a *collision*. All of the best known hashing functions attempt to minimize the likelihood of too many collisions.

Minimizing Collisions

There are two ways to attempt to minimize the likelihood of collisions:

1. make the hash table bigger
2. construct a hashing function that produces an even distribution of hash values

Q: How does making the hash table bigger minimize the likelihood of collisions?

A:

Q: What is the disadvantage to making the hash table bigger?

A:

Collision Avoidance: Natural Patterns

Let’s say we have 7,000 employees at the bottling plant. The employee file contains records with primary keys consisting of the first two letters of an employee’s last name, two digits for month of birth, two digits for day of birth and three randomly generated digits to make keys unique. Here’s an example:

`BA0227012`	`...`

We’ll allow for a hash table with 10,000 entries, so we need a four digit hash value for each key. Is there any part of the key that already has a fairly even distribution from 0 to 9,999?

Collision Avoidance: Natural Patterns (cont.)

BA0227012	...
BA1220977	...
BI0509121	...
BO1210165	...
HO0406239	...
LA1009998	...
LE0425727	...
LO0426837	...
MA0803402	...
MA0914422	...
PR0315190	...
RA1212053	...
RO0415286	...
SK0405839	...
ST0209773	...
SZ1111650	...
UR0306124	...
WH0802544	...

Q: How about converting the two letters to four digits by letting A=00, B=01, etc.?

A:

Q: How about the four month/day digits?

A:

Q: How about the last digit of the month and the three random digits?

A:

$$\boxed{H(k) = 1000 \times (\text{month mod } 10) + \text{rand\_digits}}$$

Collision Avoidance: Folding

Often (especially if a key contains some information), key values are not random. If there is a meaningful relationship between a key and the record it identifies, it is possible that similar records will have similar keys.

For example, consider the ISBN system of numbering (those numbers on books that look like this: `0-582-51734-6`). The first few digits of an ISBN refer to the publisher. Every publisher has a unique code so every book from the same publisher starts with the same digits. If we used the first digits for a hash value, books from the same publisher would *collide*.

One common technique to *randomize* the inherent similarity in keys is called *folding*. Parts of the key are separated and added together. We could hash the ISBNs by adding the first five digits to the last five:

```
0-201-38596-1 → 02013+85961 = 87974

0-201-41606-9 → 02014+16069 = 18083

0-201-55713-4 → 02015+57134 = 59149

0-582-05530-8 → 05820+55308 = 61128

0-582-51734-6 → 05825+17346 = 23171
```

Collision Avoidance: Scaling

Our folding hashing function is supposed to produce fairly random hash values, giving us fairly even distribution over the *range* of the function.

Q: But how do the hash values distribute over the RRNs in the hash table?

A:

Often, after we’ve hashed our keys to produce good, random hash values, we need to scale the values so that they range over the entire hash table. That is, for the hash function to work properly, we need to scale it:

$$H(k) = P \times h(k) \bmod TABLE\_SIZE$$

This formula ensures that the hash values range over the entire hash table.

Q: What does this mean for the size of the hash table?

A:

Collision Avoidance: Scaling: Example

Let’s say we have a hashing function *h(k)* that produces values between 1,000 and 5,999 and our hash table has 10,000 entries (from 0 to 9,999). Our scaled formula might be:

$$H(k) = 2 \times h(k) \bmod 10{,}000$$

h(k)	***H(k)***
2148	
3065	
5220	
1444	
4557	

Q: What’s wrong with the formula?

A:

Q: What can we do to *P* to fix the formula?

A:

Collision Repair

We can make our hashing function jump through all the hoops we want to avoid collisions. But if the function is not a perfect hashing function (and it almost never will be), collisions will occur. We must know how to deal with them when they *do* occur.

Luckily, there are several good algorithms for *resolving* collisions. We're going to look at three of them:

1. Linear Probing (*aka Progressive Overflow*)
2. Rehashing (*aka Double Hashing*)
3. Buckets

Collision Resolution Algorithms

The basic idea behind collision resolution algorithms is always the same:

- The hashing function *H(k)* produces a hash value *v* for some key *k* belonging to record *r*
- Position *v* in the hash table is already occupied by some other record
- Find a different position in the hash table to store *r* such that the search program (which has the same hashing function *H(k)*) will be able to find *r* in the hash table even though it’s not at the expected position *v*

Linear Probing

We build a hash table by inserting records into positions determined by applying the hash function to their keys. The size of the hash table is known in advance (it must be known in advance because TABLE_SIZE is usually a part of the hashing function).

For our collision resolution algorithms it will help to add a little field to each entry in the hash table. This field will contain one of the values {EMPTY, OCCUPIED, DELETED}. Before building the hash table all entries are set to EMPTY.

Linear Probing is probably the simplest method of collision resolution. It goes like this:

1. For record *r* with key *k*, compute the position in the hash table
   $$v = H(k)$$
2. If position *v* is not OCCUPIED, store *r* at position *v*
3. If position *v* is OCCUPIED, compute a new position
   $$v = (v + 1) \bmod \text{TABLE\_SIZE}$$
   and go back to step 2

Linear Probing Example

Let’s go back to our poor bottlers. Unfortunately, too many employees were getting drunk on the job, so they fired a bunch of the 7,000 employees. Now we only need room for 10 records in the hash table. Appropriately, we modify our hashing function to take the last digit of the primary key only:

$$\boxed{H(k) = last\_digit(k)}$$

The hash table starts out with all elements set to EMPTY.

Records

BA0227012	...
BA1220977	...
BI0509121	...
BO1210165	...
HO0406239	...
LA1009998	...
LE0425727	...
LO0426837	...
MA0803402	...
. . .	

Hash Table

Searching a Linear Probing Hash Table

Searching for a record in a hash table is almost exactly the same as inserting a record in a hash table:

1. For record *r* with key *k*, compute the position in the hash table
$$v = H(k)$$
2. If position *v* is EMPTY, no record with key *k* appears in the table
3. If position *v* is OCCUPIED by a record with key *k*, done!
4. Otherwise, compute a new position
$$v = (v + 1) \bmod \text{TABLE\_SIZE}$$
and go back to step 2

Q: What if position *v* is marked DELETED?

A:

Deleting a record uses the exact same algorithm as searching, except in step 3 mark the table entre DELETED.

Searching a Linear Probing Hash Table Example

Let's do some record searching and deleting in the hash table that has the following primary key values:

- `search for BA1220977`
- `search for BO1210165`
- `search for MA0803402`
- `delete BA0227012`
- `search for LO0426837`
- `search for MO1112727`

LE0425727
BI0509121
BA0227012
LO0426837
MA0803402
BO1210165
EMPTY
BA1220977
LA1009998
HO0406239

Q: Where's your fancy O(1) now?

A:

Chaining: A Slight Improvement on Linear Probing

Synonym

Two records with keys *k* and *j* such that $k \neq j$ but $H(k) = H(j)$.

On our trek through the hash table looking for the non-existent record with key `MO1112727`, we passed some synonyms and some non-synonyms. We can skip some of the work by keeping pointers in the hash table to link one synonym to the next.

Let’s rebuild our hash table, this time adding pointers to synonyms added as records are inserted

```
BA0227012
BA1220977
BI0509121
BO1210165
HO0406239
LA1009998
LE0425727
LO0426837
MA0803402
```


Chaining Still

Now let’s redo our searches on the new hash table with pointers to synonyms:

- `search for BA1220977`
- `search for BO1210165`
- `search for MA0803402`
- `delete BA0227012`
- `search for LO0426837`
- `search for MO1112727`

LE0425727	
BI0509121	
BA0227012	
LO0426837	
MA0803402	
BO1210165	
EMPTY	
BA1220977	
LA1009998	
HO0406239	

Double Hashing

The goal of a hashing function is to convert keys into *random* (yet reproducible) record positions. It is the randomness of the hash values that ensures even distribution across the range of the hash table.

When collisions *do* occur, however, a resolution technique like *linear probing* attempts to place synonyms close together. This can lead to *clustering* of records in the hash table.

Double Hashing resolves collisions by applying a second hashing function to the key to find a new address (instead of just adding 1, like in linear probing). Here’s an example of a Double Hashing algorithm for inserting records in a hash table:

1. For record r with key k, compute the position in the hash table
   $$v = H(k)$$
2. If position v is not OCCUPIED, store r at position v
3. If position v is OCCUPIED, compute a new position
   $$v = (v + G(k)) \bmod \text{TABLE\_SIZE}$$
   and go back to step 2

Double Hashing (cont.)

In our previous example we had the following hash function:

$$\boxed{H(k) = last\_digit(k)}$$

All keys with the same last digit are synonyms (they collide under *H(k)*). With Double Hashing, our second hashing function could be:

$$\boxed{G(k) = 1 + (second\_last\_digit(k) \bmod 7)}$$

The collision resolution formula would be:

$$\boxed{v = (v + 1 + (second\_last\_digit(k) \bmod 7)) \bmod \mathrm{TABLE\_SIZE}}$$

□

Double Hashing (cont.)

In the collision resolution formula from the previous slide:

$$\boxed{v = (v + 1 + (\mathit{second\_last\_digit}(k) \bmod 7) \bmod \text{TABLE\_SIZE}}$$

Q: What's the (+ 1) for?

A:

Q: What's the (mod 7) for?

A:

Now even if two keys map to the same hash value, it is much less likely that they will map to the same *subsequent* hash value. Even records placed following collisions are distributed evenly throughout the hash table.

□

Q: So why does this still not feel right?

A:

Pass the Bucket

Although double hashing attempts to ensure even distribution of records in the hash table, it also basically guarantees that retrieving a record *r* will cost on average *(i+1)/2* cluster reads, where *i* is the number of synonyms of *r* (including *r*).

Not only that, but those clusters will be random clusters within the file.

We can solve this problem by allowing more than one record to be stored in the same table entry. A hash table entry that can hold more than one records is called a *bucket*.

Buckets Example

For example, if each entry in the table were big enough to store three records instead of just one, our previous example would require no collision resolution.

BI0509121		
BA0227012	MA0803402	
BO1210165		
BA1220977	LE0425727	LO0426837
LA1009998		
HO0406239		

In the Bucket

Q: How does the hash table in the example compare to the original, no buckets version?

A:

Q: Do we still need a collision resolution mechanism?

A:

Q: Do we still need a DELETED marker?

A:

Q: How big should we make the buckets?

A: