COSEQUENTIAL PROCESSING

Reading:

- FZR section 8.1–8.3

Algorithm for Master-file Update

```
read the first master file record, m
read the first transaction file record, t

while (at least one file has not been completely read)

    if (m.key > t.key)
       // No master record exists for this transaction.
       if the transaction can be processed
           process it
           read the next transaction file record into t
       else
           log an error
       end if

    else if (m.key < t.key)
       // No transaction exists for this master record.
       print the [probably unchanged] record m
                             to the new master file
       read the next master file record into m

    else both keys are equal
       // Transaction t applies to record m.
       apply transaction to to record m
       read the next transaction file record into t
       // There may be more transactions for m, so
       // don't read the next master record.

    end if
end while
```


File Structures for Direct Files: Introduction

Reading:

- FZR sections 7.1–7.5 on simple indexes
- FZR sections 7.6–7.8 on fancy indexes
- FZR chapter 9 on B-trees
- FZR chapter 10 on fancy B-trees
- FZR chapter 11 on hashing
- FZR chapter 12 on incremental hashing

These topics constitute most of the remainder of the course. (So you'll have some time to do these readings!)

Recap on sequential files

As we've seen, a number of useful things can be done efficiently using only sequential access to files. Examples:

But other things *cannot* be done efficiently using only sequential access:

How can we do searching more efficiently?

We are exploiting the fact that we don't have to access a file sequentially. We can instead use "direct" access.

Limitations of keeping a sorted data file and using binary search:

- can only perform efficient binary search according to one field
  (the one it's sorted by)
- insertion and deletion are inefficient
- must have fixed-length records, and this is space-inefficient in some circumstances

Let's look more generally at direct access, and some other ways we can exploit it.

Direct files

With direct access, to access any file position is $O(1)$.

So if we have a way to go from
record key $\Rightarrow$ location in file
then we can access any record in $O(1)$ time.

NB: File access may be $O(1)$, but it's *very* slow relative to memory access.
So we still want to minimize jumping around.

There are many ways to organize a direct file! ...

Two general approaches

Approach A: For any possible key, *compute* the location of the record associated with it.

We use a function to map
record key $\Rightarrow$ location in file.

Approach B: Explicitly *store* the location of the record associated with a key.

Use some file structure to index the data file. Each element of the index stores:

- a key plus
- the location of the record with that key.

To find a record, search index by key and then go to the location it gives.

Simple Indexed Files

Reading:

- FZR sections 7.1–7.5

An analogy used in FZR

Searching for topics in a book

One approach:

- Book: Sort the words and use binary search.
- File: Analogous to sorting records in a data file and using binary search.

Problem:

- Book: the order of the words is lost!
- File: We may not be willing to re-order records in a file.

Solution:

Searching for books by multiple fields

One approach:

- Library: Have 3 copies of each book, and 3 libraries, each sorted a different way.
- File: Duplicate the data in multiple files, each sorted a different way.

Problem:

- Library: This requires 3 times the resources!
- File: Ditto. Plus the redundancy is dangerous. What if the copies of a record disagree across files??

Solution:

Advantages of indexes

We get fast searching, but we could get that with sequential sorted files and binary search. Advantages beyond that:

- The data file can be in any order — it doesn't have to be sorted. This allows faster insertion and deletion.
- We can have as many indexes as we like. This allows fast searching according to *several* keys.
- Records do not have to be of fixed length. This can permit the data file to use less space.

Once we create an index, the data records are "**pinned**": Because they are pointed to, if we move them we must update all pointers to them.

File Structure for a Simple Index

(Remember, there may be several indexes on the same data file, even one per field.)

Have one index entry per record in the data file.

Index must consist of fixed-length records, and must be sorted by some field.
Otherwise, we can't do fast binary searching of the index; we might as well have just an unsorted data file without an index.

An index can be in the same or a separate file.

If the index is small enough, we load it into memory while we're using it. (A simple array of structs will do.) This allows for much faster searches etc.

Some of the Basic Operations

Search

- The binary search is done in memory.
- In total, just one seek is required.

Insertion

- Can just add the new record to the end of the data file, since its order doesn't matter.
- Must update the index too. It's sorted, so this usually means some shifting.
- That's $O(n)$, where n is the number of records in the index
- But n is also the size of the data file! Isn't $O(n)$ really slow? Linear search on the data file would have been $O(n)$!

Deletion

- Have to update the index, which means some shifting. Same points as for insertion.
- But updating the data file isn't so simple. What to do??

Creating an index the first time

- Method:
- May have to re-create the index if it is ever destroyed.

Loading the index into memory

- Method:

Storing the index back into a file

- Method:
- Big worry: What if the index isn't rewritten, or is rewritten incompletely?

Speeding Things Up

Problem: If the index won't fit into memory, operations will be *much* slower!

Consider insertion. With our simple index updating the index is

- $O(\log n)$ to find the right spot and
- $O(n)$ to do the inserting

Thus it's $O(n)$ over all.

Solution: How can we speed it up?

Using a BST for an Index

Now what's the complexity of insertion? Search?

Solution:

Then our searching etc. time will much better: $O(\log n)$ in the worst case.

(Cost: **)**

Problem: This improved search time may not be good enough if we're searching in a really large file.

Solution: ??

Another Approach

Instead of using a BST for the index, let's again consider a simple index for a large file.

Example: We have 6,000,000 records of 120 bytes with 8 byte keys. That's 6,000,000 key/reference pairs for the index.

Problem: We can't fit the index in memory

Solution: Build an index to the index. Suppose we can put 100 of these pairs into a single record of the index. Now our second index has only 60,000 key/reference pairs.

Problem: It still doesn't fit into memory

Solution:

With a branching factor (or fan-out) of 100, our multi-level index only needed 4 levels.

Now what's the complexity of search?

This is because the tree is perfectly balanced.

But at what cost did we achieve this?

We need a tree with a branching factor > 2 and restrictions forcing it to be reasonably balanced. We also need efficient algorithms for insertion and deletion which maintain that reasonable balance.

B-trees

A tree with branching factor > 2, and restrictions forcing it to be reasonably balanced.

A B-tree of "order M" must obey these rules:

- all leaves at the same level
- branching factor is M,
  i.e., no more than M children per node
- at *least* $\lceil M/2 \rceil$ children per node,
  except the root.
  (For the leaves, these children are all nil.)
- root has more flexibility:
  at least 2 children, unless it's the only node.

We'll cover B-trees in detail.

B Trees

Reading:

- FZR chapter 9 on B-trees
- FZR chapter 10 on B^{+}-trees

Performance of B-tree Insertion

Best case: ____________________

- # nodes read =
- # nodes written =

Worst case: ____________________

- # nodes read =
- # written read =

Is this good?
It all depends on tree height. ...

Height of a B-tree with n records

Recall that #pointers per node $\geq \lceil m/2 \rceil$ (except for the root, which has ≥ 2).

Minimum #leaves in a B-tree of height h

Height	**Minimum Descendants**
h = 1 (root)	2
2	$2\lceil m/2 \rceil$
3	$2\lceil m/2 \rceil^2$
...	
h	$2\lceil m/2 \rceil^{h-1}$

So, in general, for any level of a B-tree, the minimum number of descendants extending from that level is $2\lceil m/2 \rceil^{h-1}$

For a B-tree with N keys in its leaves, the minimum height of the tree h is expressed by the relationship.

$N \geq 2\lceil m/2 \rceil^{h-1}$

Height of a B-tree with n records

We can flip things around to find the minimum height.

$n \geq 2\lceil m/2 \rceil^{h-1}$

$\frac{n}{2} \geq \lceil m/2 \rceil^{h-1}$

$\log_{\lceil m/2 \rceil}\left(\frac{n}{2}\right) \geq h - 1$

$h \leq \log_{\lceil m/2 \rceil}\left(\frac{n}{2}\right) + 1$

Comparing to BSTs

Worst height for a BST with n nodes?

When do we get height about $\log_2 n$?

Bottom line

For a BST with n nodes
(and therefore n records),

$$\boxed{h \geq \lceil \log_2(n+1) \rceil}$$

For a B-tree with n nodes,

$$\boxed{h \leq \log_{\lceil m/2 \rceil}\left(\frac{n}{2}\right) + 1}$$

B-tree Deletion

Basic algorithm:

- Search for the key to be deleted.
- If it's not the highest key in the node, just delete the key from the node.
- If it is the highest,
  1. Delete the key, and
  2. Modify the index (at possibly multiple levels) to reflect the change.

Problem: Underflow.

Either kind of key deletion may leave a node too small.

Handling underflow

Easier case:

- If an adjacent sibling has > the minimum, steal a record from it.
- This is called **redistribution**.
- Higher level indexes will have to be modified.

Harder case:

- If not, do the opposite of splitting: merge two nodes together.
- A key will have to be deleted from the parent node.
- This is called **concatenation**.
- Changes in the index may ripple all the way to the top.

Hashing

Reading:

- FZR chapter 11 on basic hashing

Recap: Direct files

In a direct file:

- to access any file position is $O(1)$.

So if we have a way to go from
record key ⇒ location in file
then we can access any record in $O(1)$ time.

NB: File access may be $O(1)$, but it's *very* slow relative to memory access.
So we want to minimize jumping around.

There are many ways to organize a direct file! ...

We've already discussed in detail **Approach B:** explicitly *store* the location of the record associated with a key.

Use some data structure to index the records.

Each element of the index stores:

- a key that is in use, plus
- the location of the record with that key.

To find a record, search data structure by key.

Bonus: can have multiple indexes.

Approach A: For any possible key, *compute* the location of the record associated with it.

We use a function to map
record key ⇒ location in file.

We'll look at three ways to do this.

(1) Direct Mapping

The key itself *is* the position of the record in the file.
I.e., $f(key) = key$.

Problems:

(2) Directory lookup

Keep a directory (probably in a separate file). It tells you where in the record file to find the record.

To find the appropriate directory entry, use the record's key directly.
So again, $f(key) = key$.

We save space:

And we lose space:

When does this method beat Direct Mapping?

Bottom line re space:

What about time?

(3) Hashing

Don't bother with a directory. Have just one file, and use the key to find the record's location in it.
(Just like Direct Mapping.)

But this time, use a mapping function that is not "direct". Use one that takes us from
large key space $\Rightarrow$ small address space.

Hashing Issues

Must devise an appropriate hash function.
Because the hash function maps a large space to a small space, we will have "collisions".

We can make each location a "bucket" that can store lots of records.

- But buckets must have fixed size, thus they can still overflow.

We will need a scheme to handle this.

Must decide on the # and size of buckets.

When file gets very full, collisions can be too numerous. May be worthwhile re-organizing the file layout to have more buckets

Hashing Performance

If everything is well designed, retrieval can be very fast — just a few file accesses.

One operation is really slow:

Hash functions

A hash function is a mathematical function that maps from
keys ⇒ locations.

There are some standard types of hash function, including

- **mid-square:** square the number and then take some digits from the middle.
- **folding:** Divide the number in half and combine the two halves, *e.g.*, add them together.
- **modular division:** Mod by some number, preferably a prime.

See the text for more about hash functions. Note that there is a lot of interesting theory about hash functions and their properties. (csc 378 covers this.)

Examples of hash functions

Say our key is a string. Before we hash it, we need to turn it into an integer.

One solution: Concatenate together the alphabetic position of the 1st and the second character. **E.g.** "$\boxed{\text{T}}\boxed{\text{o}}$yota" ⇒ $\boxed{20}\boxed{15}$.

Now we need a hash function to hash up the integer. Examples of the three general types:

- Mid-square, taking the middle 2 digits. **E.g.** $2015^2 = 406\boxed{02}25$.
- Folding: adding the two halves.
  E.g. 2015 ⇒ 20 + 15 = 35.
- Mod by 97.
  E.g. 2015 mod 97 = 75.

key		h(key)		
as string	converted	mid-square	folding	mod 97
Toyota	2015	$406\boxed{02}25$	35	75
Chev	0308	$94\boxed{86}4$	11	17
Ford	0615	$37\boxed{82}25$	21	33
Chrysler	0308	$94\boxed{86}4$	11	17
Volkswagen	2215	$490\boxed{62}25$	37	81
Nissan	1409	$198\boxed{52}81$	23	51
Plymouth	1612	$259\boxed{85}44$	28	60
Dodge	0415	$17\boxed{22}5$	19	27
Renault	1805	$325\boxed{80}25$	23	59
Saab	1901	$361\boxed{38}01$	20	58
Isuzu	0919	$84\boxed{45}61$	28	46
Pontiac	1615	$260\boxed{82}25$	31	63
Fiat	0609	$37\boxed{08}81$	15	27

Two kinds of collisions:

- collisions that occur because we begin with the same 2 integers. Are collisions for every hash function we might choose.
- collisions that occur even though we begin with 2 different integers. Not necessarily collisions with a different hash function.

Couldn't we use the 2015 as the hashed value?

This would be a bad idea. With 4 digits, there are 10,000 possible values (0...9999), yet only a few will be used.

- Some are unlikely to crop up
  E.g. "aa" ⇒ 0101.
- Some *cannot* crop up
  E.g. ?? ⇒ 2701; 27 is out of range.

Yet we need our hash table to be continuous, and therefore to have all 10,000 slots.
So our hash table will be largely empty.

With each of the 3 hash functions we looked at, the range of h(key) is 0...99 (or less with mod 97).
So our hash table only needs 100 slots.

Of course we might overflow it, but we have to deal with this anyway.

Avoiding collisions?

Upon doing a new insertion, how likely is a collision?

It's certainly more likely when many items have already been inserted.

Loading factor: $= \frac{\#records\ currently\ in\ the\ file}{\#records\ that\ the\ file\ can\ hold}$

In our cars example, the loading factor is only $\frac{13}{100} = 0.13$, yet we already have collisions!

We could reduce collisions by making the capacity of the file bigger (and hence the loading factor smaller). But ...

Exactly how likely are collisions?

For a given file capacity, how likely are collisions as file gets more loaded?

Example: A file of 365 buckets. Let $Q(n)$ be the probability that NO collisions occur during n insertions.

$Q(1) =$

$Q(2) =$

$Q(3) =$

In general,
$Q(n) =$

Q(1) =

Solution to the recurrence relation:

$Q(n) = \frac{365!}{365^n(365-n)!}$

The probability that collisions DO occur is $1 - Q(n)$.

n	$1 - Q(n)$
10	0.1169
20	0.4114
23	0.5073
30	0.7063
40	0.8912
50	0.9704
60	0.9941

If the loading factor is only $\frac{23}{365}$, 50% chance.

If the loading factor is only $\frac{47}{365}$, $>$ 95% chance!

So yes, collisions are a problem!

Buckets

The hash function $h(k)$ tells us where to store (or retrieve) a record with key k.

This could be a slot in an array in memory, or a slot in a file. (For this course, a file.) Either way, we often use the term "hash table".

The slots are called "buckets", because they have capacity for > 1 record. We choose the capacity based on the number of records that can be read or written in one file access.

Analogy: your address book.
key:

hash function:

bucket:

Handling collisions

What do we do when a collision occurs?

Easy case: the bucket has room for the record.

Hard case: the bucket doesn't have room for the record. We call this "overflow".
We need to figure out two things:

- A place to put the record that won't fit it its home bucket.
- A way to find that record later!

How do you handle overflow in your address book?

Handling overflow

Two kinds of approach: either compute or store where to try next. (Gee, where have we heard that before?)

Open addressing

Compute another bucket to try, based on some rule.

Closed addressing (or chaining)

Store the location of another bucket to try, using some sort of pointer.

The "overflow" records may be kept in a separate overflow area, perhaps in a separate file.

Open Addressing

Compute where to look, based on the key.

General method for insertion:
(search is analogous; why?)

Let A_i be where to look on the ith try.

- Use the hash function (h) to find the first bucket where the record might go, A_0.
  $A_0 = h(key)$
- If that bucket is full,
  use a new function (f) to find the next bucket to try. Repeat as necessary.
  $A_i = f(i, key)$
- Stop when we hit a bucket with room, or the sequence of A_i's starts to repeat (*i.e.*, there is no room anywhere).

Many ways to design the function f ...

I. Linear Probing

Step through a sequence of buckets always using the same step size.

Use mod to wrap around when we hit the end of the hash table.

$$\boxed{A_i = (A_{i-1} + stepSize) \bmod n}$$

(We can also express A_i in terms of A_0.)

Example:

$$\begin{aligned} A_i &= (A_{i-1} + 2) \bmod n \\ &= (A_0 + 2i) \bmod n \end{aligned}$$

The sequence of buckets considered is called the "probe sequence".

Linear probing example

n (# buckets) = 13 h(key) = (sum 1st 3) mod 13
bucket size = 1
step size = 2 So $A_i = (A_{i-1} + 2) \bmod 13$

key	h(key)	probe sequence
Chevrolet	16 mod 13 = 3	
Chrysler	29 mod 13 = 3	
Jaguar	18 mod 13 = 5	
Nissan	42 mod 13 = 3	
Karman Ghia	30 mod 13 = 4	

Bucket #	Bucket Contents
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	

II. Non-linear Probing

Problem: **Primary clustering**.

If several records hash to the same spot, or even *any* spot along the probe sequence, they will all follow that same probe sequence.

Example: n = 100; step size = 2.
hash to: 5 probe seq:
hash to: 9 probe seq:

Solution: Make the step size depend on the step number, i.
This is called **non-linear probing**.

E.g., $A_i = (A_{i-1} + 2i^2) \bmod n$

Example: n = 100; step size = $2i^2$.
hash to: 5 probe seq:
hash to: 9 probe seq:

III. Double Hashing

Problem: **Secondary clustering**.

All records that hash to the same spot still have the same probe sequence.

Example: n = 100; step size = $2i^2$.
key 1; hash to: 5 probe seq:
key 2; hash to: 5 probe seq:

Solution: Make step size depend on the *key* (but differently than in original hash function). This is called **double hashing**.

E.g., $A_i = (A_{i-1} + h_2(key)) \bmod n$

Closed Addressing

Instead of computing where to look next, store it, using a "pointer".

Each full bucket has a pointer to an overflow area,

- in the same file (for example at the end)
- or in another file

There are many ways to organize this, since pointers are so flexible.

Cost vs open addressing:
Savings:

Do deletions introduce problems?

Table-assisted Hashing

With ordinary hashing,

- the hash function tells us where to start looking, and
- the collision resolution scheme tells us where to go from there if necessary.

Can we instead come up with a way to be *sure* which one bucket to look in, before we go into the file?

What would we need to know about a bucket that would tell us whether to look there?

Idea: Keep a table in memory that tells, for each bucket, what's the largest key in it.

This is called **table-assisted hashing**.

Example: Searching for key value 30, and we hash to bucket 52.

bucket	largest key in it
26	
⋮	
52	13
⋮	
85	27
⋮	
92	36
⋮	
116	

The table tells us our record is definitely in bucket 92 (if it's anywhere).

Benefits? Costs?

Can we likely keep the table in memory?
Would it be ok to keep it in a file?

"Incremental" Hashing

Reading:

- FZR chapter 12

A Problem

Performance degrades if the file becomes heavily loaded,
i.e., if $\frac{\text{actual}-\text{number}-\text{of}-\text{recs}}{\text{num}-\text{buckets}\times\text{bucket}-\text{size}}$ gets large.

To make things better, it may be worthwhile to increase the number of buckets (and reorganize the data).

This general idea is called **incremental hashing**.

Guess what? There are many ways to do it.

Incremental Hashing

General Approach

As records are inserted, if performance becomes too low, grow the file.

- *I.e.*, "split" one bucket and disperse its records; some stay put and others go to a new bucket.
- This reduces overflow (collisions to full buckets) and hence reduces the # of file accesses during search.

As records are deleted, if space usage becomes too poor, shrink the file.

- *I.e.*, merge two buckets into one.
- This reduces the total # of buckets, and hence reduces waste.

File growth and shrinkage is incremental, *i.e.*:

- It happens on the fly.
  We do it during insertions and deletions, if needed.
- It happens in small amounts.
  We split one bucket rather than rehashing the whole file.

Possible measures of performance include:

- load factor
- average # of disk accesses per search.

Method I: Linear Hashing

Method

- When performance becomes too poor, split bucket 0. (Yes, this is arbitrary.)
- Split it by doubling the mod factor and rehashing its contents. *E.g.*,
  $h(k) = k$ mod 3 becomes
  $h(k) = k$ mod 6.
- Next time, split bucket 1, then 2, etc.
- Keep a counter to remember which buckets have been split.
  Unsplit ones use the old hash function.
  Split ones use the new.

Merging is analogous but opposite.

	old	new
# buckets	$T : 0 \ldots (T-1)$	$T+1 : 0 \ldots T$
hash fcn	$h(k) = k$ mod 3	$h(k) = k$ mod 6
	$h(k) = k$ mod T	$h(k) = k$ mod $2T$

Guarantee: Every element of bucket 0 will either stay put, or land in the new bucket T.

More generally, if we split bucket b, every record will either stay put, or land in the new bucket $T + b$.

Let k be the record's key.
If it was in bucket b originally, we know

k mod $T = b$.

So k must have been one of these:

$$h \quad T+b \quad 2T+b \quad 3T+b \quad 4T+b \quad 5T+b \quad \ldots$$

So when we hash k with the new hash function $h(k) = k$ mod $2T$, we get either:

- b, in which case the record stays put, or
- $T + b$, in which case it goes to the new bucket, $T + b$.

Questions

Will linear hashing work if we use open addressing to solve collisions?

Why split the "next" bucket? Why not the culprit, *i.e.*, the one we inserted to when we passed the performance threshold?

Decision: What if the split fails, *i.e.*, everything happens to stay put? We could split again.

What happens when we've split all the original buckets?

Method II: Extendible Hashing

Build a dynamic directory (in memory for speed) that copes with the varying load factor.

- Hash function takes you to a *directory* entry, rather than directly to a bucket.
- Because buckets are pointed to, needn't be consecutive in the file. So can add and remove buckets as desired.
- Directory must grow and shrink with number of buckets.
- So # of places to hash to changes. Cope by using only the first so many bits of h(key); change this as necessary to change size of directory.
- If using d bits, directory size is 2^d.
- So have capacity for 2^d buckets, but can start with fewer; even just one.

How to "grow" the file

When a bucket overflows:

- Split the one bucket in two.
- Half of the directory entries that pointed to the old bucket will still do so, and half will point to the new bucket.

Eventually, we may reach a point where we can't split a bucket this way.

- This occurs when only one directory entry points to the bucket we want to split.
- Then we double the directory size, and reorganize.

Map of the World

A quick summary of topics that we've covered concerning direct files.

Direct files ⇒ store location

- Simple static index
  (index structure never changes)
- Dynamic index
  (index structure changes)
  - BST
  - (Not covered: AVL tree; M-way tree)
  - B-tree
    * plain B-tree
    * B^* tree
    * B^+ tree

Direct files ⇒ compute location

- Direct mapping
- Directory lookup
- Hashing
  1. Need a hash function
     - folding
     - mid-square
     - modular division
  2. Must handle bucket overflow
     - overflow ⇒ compute next bucket ("open addressing")
       * Linear probing
       * Non-linear probing
       * Double hashing
     - overflow ⇒ store location of next bucket ("closed addressing / chaining")
  3. May be table-assisted
  4. May be incremental
     - Linear hashing
     - Extendible hashing

PROVIDING ACCESS BY SECONDARY KEYS

Reading:

- FZR sections 7.6–7.8

DATABASES

Reading:

- None required