INTRODUCTION TO CSC228

Reading:

- none on this topic
- but see the week-by-week handout for readings you should start (now) on upcoming topics

Scenario: You work for Statistics Canada and your job is to write a program that will give government statisticians easy access to the census data, going back over several decades. They want to retrieve data and perform interesting queries.

Scenario: You want to make your first million by writing a web search engine with some amazing new capabilities.

Coming fresh out of csc148, what will be your biggest stumbling blocks?

Problem 1: The data is HUGE; it won't all fit in memory.

Solution: Leave it in a file but bring smaller pieces in memory when needed.

Problem 2: How are you going to perform operations like searching?

The only way we've taught you to read a file is to start at the top and read through to the bottom. This rules out fast algorithms like binary search. With so much data, that's disastrous.

Solution: Adapt clever data structures (like trees), but build them *in the file* rather than in memory. Now we call them "file structures".

A pointer will now reference a location in the file, rather than a location in memory. (We'll learn how to implement file pointers in C.)

Facing another reality

When you follow a pointer in memory, what has to happen?

When you follow a pointer in a file, what has to happen?

Problem 3: Following a file pointer is so slow that we have to be extremely careful to minimize the number of times we do it.

Solution: Use a binary search tree, but make sure it stays balanced.

Think of the census data. $O(\log n)$ used to sound good, but now the constants are so big that we have to care about them.

- What we have to do $O(\log n)$ times is follow a file pointer (plus some faster in-memory work) — and that's a very slow operation.

Plus, if n is huge, $O(\log n)$ itself can be big.

- If your file contains an index of web pages around the world, n could be 20,000,000. What's $\log_2 20,000,000$?

How can we speed things up?

What 228 is all about

This course is about how to manage LARGE amounts of data.

- Understanding how files work
  so that we can better appreciate these issues
- Clever data structures and algorithms
  that try to offer efficiency in the face of these issues
- Programming techniques
  that allow us to implement the clever data structures and algorithms

It's also about how to manage large software projects. More realistic.

- You will work on larger programs.
- You will be more responsible for figuring things out yourself.
- You will be more responsible for making your own design decisions.
- You will work in groups.

Students who effectively manage

- the software design process
  (i.e., have a serious plan for designing, implementing, and testing their code)
- and their team
  (i.e., have a serious plan for who will do what, when, and a strategy for coping when that doesn't happen)

have a huge advantage over everyone else.

ADMINISTRATIVE DETAILS

Reading:

- csc228 Course Information Sheet
- csc228 Course Guide

You

How many ...

- have taken csc209:
- have taken csc258:
- have taken csc238:
- have taken some third year:

How many ...

- are very comfortable in C:
- feel somewhat weak in C:
- are very comfortable in C++:
- feel somewhat weak in C++:

Computers (St George only)

You will be using the CDF lab.

Account name: `a228xxxx`
Password: your student number
Internet address: `@cdf.utoronto.ca`

You can phone the CDF facility from your home computer. See the CDF guide for details.

You are also free to do your assignments on your own computer.

But, warning: you will be required to hand in all assignments electronically, and they must run on the CDF machines.

Plagiarism; Helping each other

The usual.

- In brief: you must submit work that is your own.
- But you *should* get to know other students and work together to learn course material.
- You will have an opportunity for group work on the project.

BASIC CONCEPTS: PHYSICAL FILES, LOGICAL FILES, RECORDS

Reading:

- FZR chapter 1
- FZR sections 2.1–2.5
- FZR sections 3.1–3.7
- FZR section 4.1
- FZR chapter 5

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.