CSE332: Data Abstractions

Lecture 8: Memory Hierarchy & B Trees

Ruth Anderson

Winter 2011

Announcements

- **Project 2** – posted!
  Partner selection due by 11pm Tues 1/25 *at the latest*.
- **Homework 2** – due NOW!
- **Homework 3**– due Friday Jan 28st posted later today

Today

- Dictionaries
  – AVL Trees (finish up)
- The Memory Hierarchy and you
- Dictionaries
  – B-Trees

Why do we need to know about the memory hierarchy?

- One of the assumptions that Big-Oh makes is that all operations take the same amount of time.
- Is that really true?

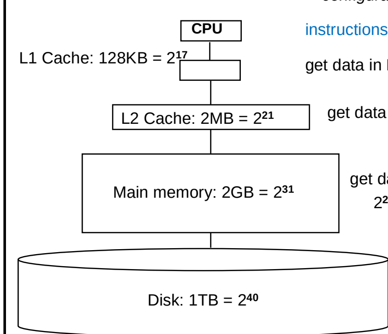
Now what?

- We have a data structure for the dictionary ADT (AVL tree) that has worst-case $O(\log n)$ behavior
  – One of several interesting/fantastic balanced-tree approaches
- We are about to learn another balanced-tree approach: B Trees
- First, to motivate why B trees are better for really large dictionaries (say, over 1GB = 2^{30} bytes), need to understand some ***memory-hierarchy basics***
  – Don't always assume "every memory access has an unimportant *O(1)* cost"
  – Learn more in CSE351/333/471 (and CSE378), focus here on relevance to data structures and efficiency

---

A typical hierarchy

"Every desktop/laptop/server is different" but here is a plausible configuration these days



instructions (e.g., addition): 2^{30}/sec

get data in L1: 2^{29}/sec = 2 instructions

get data in L2: 2^{25}/sec = 30 instructions

get data in main memory: 2^{22}/sec = 250 instructions

get data from "new place" on disk: 2^{7}/sec =8,000,000 instructions

Morals

It is much faster to do:	Than:
5 million arithmetic ops	1 disk access
2500 L2 cache accesses	1 disk access
400 main memory accesses	1 disk access

Why are computers built this way?

- Physical realities (speed of light, closeness to CPU)
- Cost (price per byte of different technologies)
- Disks get much bigger not much faster
  - Spinning at 7200 RPM accounts for much of the slowness and unlikely to spin faster in the future
- Speedup at higher levels (e.g. a faster processor) makes lower levels *relatively slower*
- Later in the course: more than 1 CPU!

"Fuggedaboutit", usually

The hardware automatically moves data into the caches from main memory for you

- Replacing items already there
- So algorithms much faster if "data fits in cache" (often does)

Disk accesses are done by software (e.g., ask operating system to open a file or database to access some data)

So most code "just runs" but sometimes it's worth designing algorithms / data structures with knowledge of memory hierarchy

- And when you do, you often need to know one more thing…

How does data move up the hierarchy?

- Moving data up the memory hierarchy is slow because of *latency* (think distance-to-travel)
  - Since we're making the trip anyway, may as well carpool
    - Get a block of data in the same time it would take to get a byte
  - Sends *nearby memory* because:
    - It's easy
    - And likely to be asked for soon (think fields/arrays)
- Side note: Once a value is in cache, may as well keep it around for awhile; accessed once, a value is more likely to be accessed again in the near future (more likely than some random other value)

Spatial Locality

Temporal locality

Locality

Temporal Locality (locality in time) – If an item is referenced, it will tend to be referenced again soon.

Spatial Locality (locality in space) – If an item is referenced, items whose addresses are close by will tend to be referenced soon.

Block/line size

- The amount of data moved from disk into memory is called the "**block**" size or the "**page**" size
  - Not under program control
- The amount of data moved from memory into cache is called the cache "**line**" size
  - Not under program control

Connection to data structures

- An array benefits more than a linked list from block moves
  - Language (e.g., Java) implementation can put the list nodes anywhere, whereas array is typically contiguous memory

    (Note: "array" doesn't necessarily mean "good")
    - Binary heaps "make big jumps" to percolate (different block)

BSTs?

- Since looking things up in balanced binary search trees is $O(\log n)$, even for $n = 2^{39}$ (512GB) we don't have to worry about minutes or hours
- Still, number of disk accesses matters
  - AVL tree could have height of, say, 55
  - Which, based on our proof, is a lot of nodes
  - Most of the nodes will be on disk: the tree is shallow, but it is still many gigabytes big so the *tree* cannot fit in memory
    - Even if memory holds the first 25 nodes on our path, we still need 30 disk accesses

Note about numbers; moral

- All the numbers in this lecture are "ballpark" "back of the envelope" figures
- Even if they are off by, say, a factor of 5, the moral is the same: If your data structure is mostly on disk, you want to minimize disk accesses
- A better data structure in this setting would exploit the block size and relatively fast memory access to ***avoid disk accesses***…

Trees as Dictionaries

(N= 10 million)

In worst case, each node access is a disk access, number of accesses:

Disk accesses

- BST
- AVL
- B Tree

Our goal

- **Problem**: A dictionary with so much data most of it is on disk
- **Desir**e: A balanced tree (logarithmic height) that is even shallower than AVL trees so that we can minimize disk accesses and exploit disk-block size
- **A key idea**: Increase the branching factor of our tree