B+ Trees

CptS 223 - Fall 2017 - Aaron Crandall

Today’s Agenda

- Announcements
- Humor of the day
- B+ Trees

Announcements

- HW2 is due out - it's all about trees and tree operations
- WSU Tech Expo & Job Fair is Oct 3rd - 10-3 in Beasley Coliseum
    - Get your resume ready! - This takes time to get it right. Have someone read it
    - Make a public GitHub account on github.com (or bitbucket.com)
        - NO joke - put some personal code there and you'll stand out for internships
- Career Fair Prep #5 – Workshop: Interview Techniques & Mock Interviews
    - https://goo.gl/UQM6vN - Sept 25th @4:30pm-4:30pm in Sloan 169

Thing of the day

Foo, bar, baz - This is a pattern used to describe nonsense names in code

- Derived from WWII slang and ported to CS engineers
- Fubar -> foo, bar
- Baz? Apparently just because it flows well

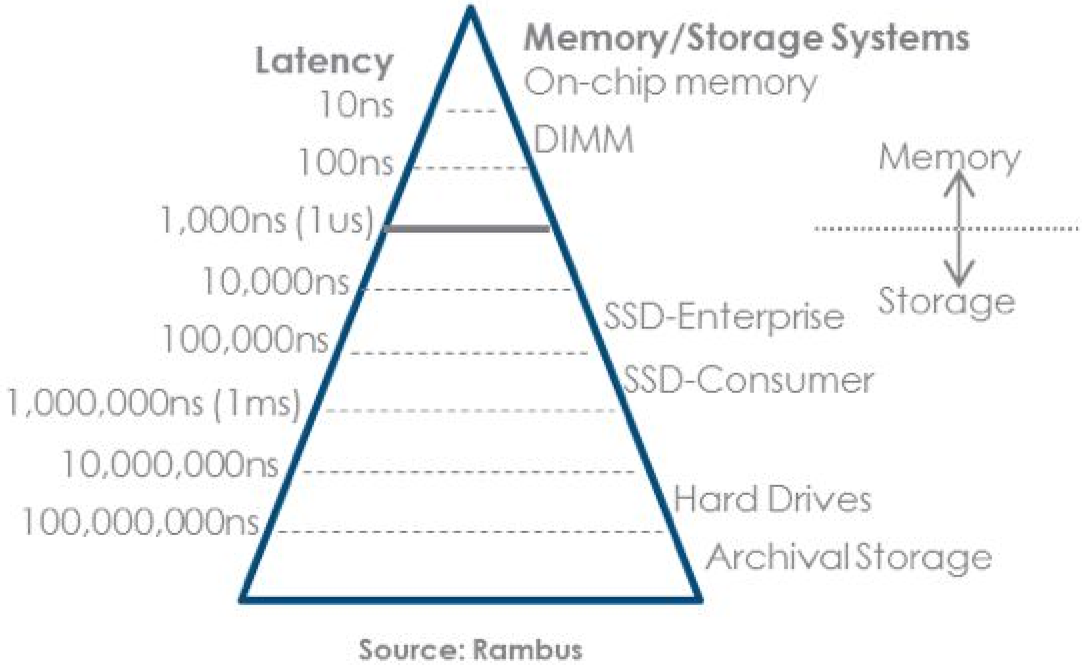
```
Void foo(char baz[100])
{
            strcat (baz, “ quz”);
}

void bar(void)
{
            char            baz[100];

            sprintf(baz, “qok”);
            foo(baz);
            printf(“%s”, baz);
}
```

B+ Trees!

- An M-ary tree (instead of a bin-ary tree)
    - Allows M children per node
- Data is stored at leaf nodes
    - Instead of at all nodes (like a classic B Tree)
- Designed to store data is blocks on disk instead of in RAM
    - Allows handling of larger data sets by optimising for slower read/write speeds
- Used very often for indexing in SQL databases
    - Gets sequential ordered data access O(N) if leaf nodes link to each other

The Memory Hierarchy

- CPU at the top
- Slowest data at bottom
- Managing disk accesses is what B-trees do



Book example numbers are out of date.

- They're still using 2002 numbers in their disk calculations
- SSD drives, M.2 SSD, new RAM tech, DMA channels, and supercap backed memory buffers have made disks very very speedy
- But it's still illustrative for large data storage issues

Old(er) tech	Today?
HDD - Spinning disks - 7,200 RPM w/seek times	SSD - NAND Flash memory - Also have DRAM buffers
Access times: ~9ms	Access times: ~10μs
0.009 seconds	0.00001 seconds

Access time difference?

HDD - 9ms - 0.009 seconds   (111 per second)

SSD - 10µs - 0.00001 seconds   (100,000 per second)

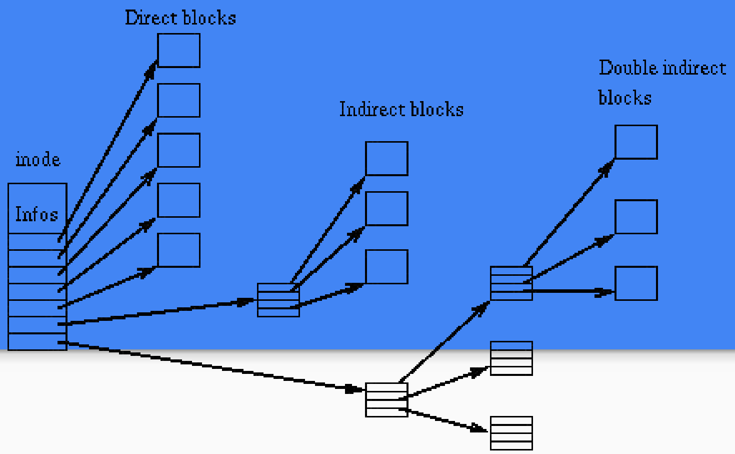
3 GHz CPU: 3,000,000,000 cycles/second

So… even with SSD the CPU will execute 3000-ish instructions before the SSD can access the data. *WAY* better than the 3000000 for a HDD, but it's still noticeable!

I did like the unbalanced binary tree example

- BST with 10,000,000 records degraded into a linked list?
- Each access takes ⅙ sec, so... 10M * ⅙ sec ~= 1666666 sec (19.2 days)
    - With an SSD drive array, it's more like 0.5 days!
- Do you run into 10M element trees?
    - Our smart home data has about 300M data points
    - The database index for that is a tree structure

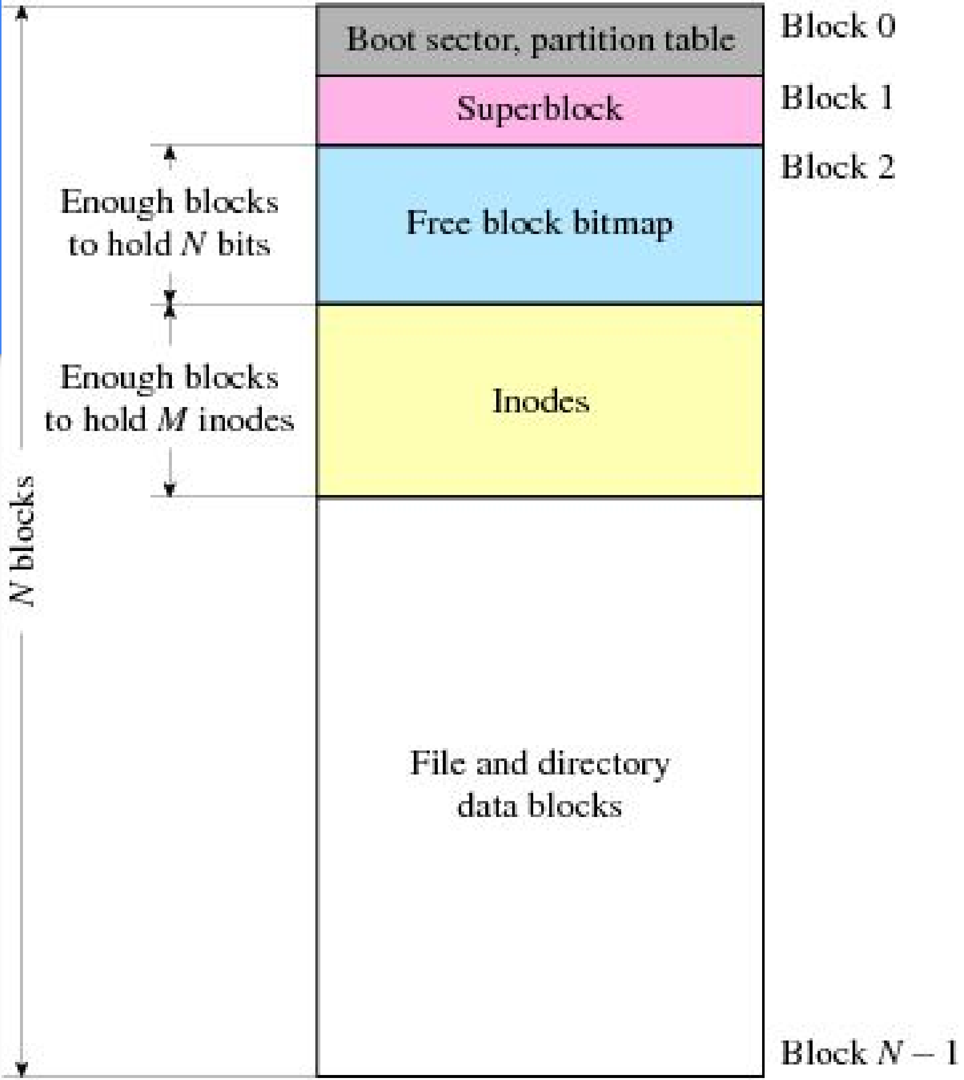
Optimize our large tree's read/write to disk



- Disk filesystems are based on blocks
- What are blocks? - The size of data that can be read/write in a single op
- The disk partitions are formatted with a filesystem
  - The filesystem keeps pointers to blocks (sections) of data with a given size
  - Normally 4 or 8k, but can be made bigger or smaller when you format the partition
- Defaults for some filesystems (fs):
  - NTFS: 4k	Ext4: 4k	Btrfs: 4k
- Data is always aligned with blocks, even if it's not a full block
  - Fragmentation and wasted space do happen (a lot)!

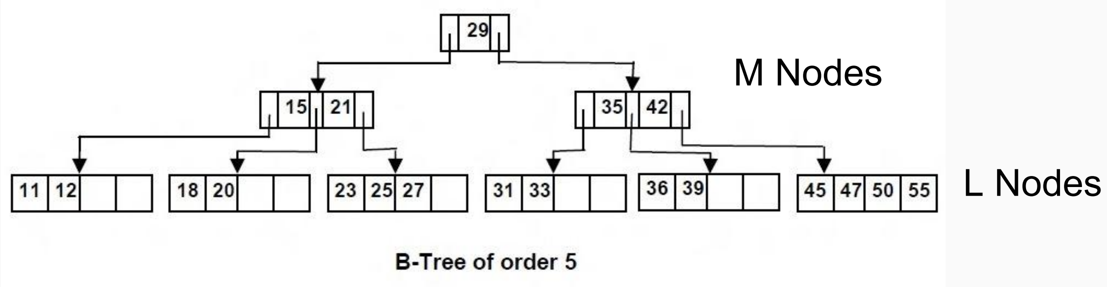
Visually-ish:

- Ever wonder why your actual storage space on disk is much smaller than the size of the disk?
- It's the overhead of the partition tables, inodes, and other bookkeeping.
- You'll know this by heart after CptS 360, and then 460

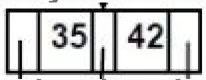


Why the book talks about M & L

- There are two kinds of nodes in a B+ tree:
  - Index (or M) nodes of size M: The number of children a node can have
  - Data (or L) nodes of size L: The number of records a leaf can store
- M & L are calculated by how big of a node you could stuff in a single file system block without going over.
  - Can only be whole numbers, since you can't store part of a node. Always round down.



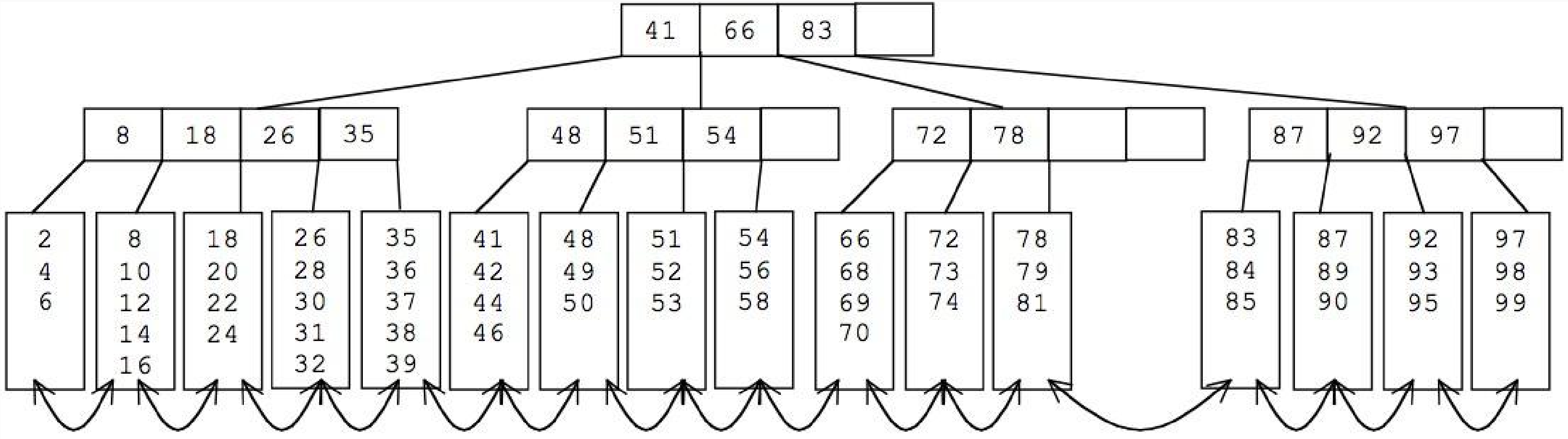
M Nodes

- Only store pointers to other nodes (M or L) and keys for subtrees
  - The keys stored tell you which subtree to descend into to find your data
- Have M pointers and M-1 keys (how you index a record)
  - This example is a B-tree of order 3: 

- You calculate the order by how much physical space you use in a fs block:
  - fs blocksize >= M * pointerbytes + (M - 1) * keysize
  - Round down for M (you can't store 1.5 pointers)
- Question: how big are pointers on your computer? (in bits)

L nodes - only store data records (and maybe a pointer to the next L node)

- L nodes store the actual data records
- The keys in M nodes are just the index value for records, not the data
- fs blocksize >= L * sizeof(record) + optional pointer



Some calculation examples

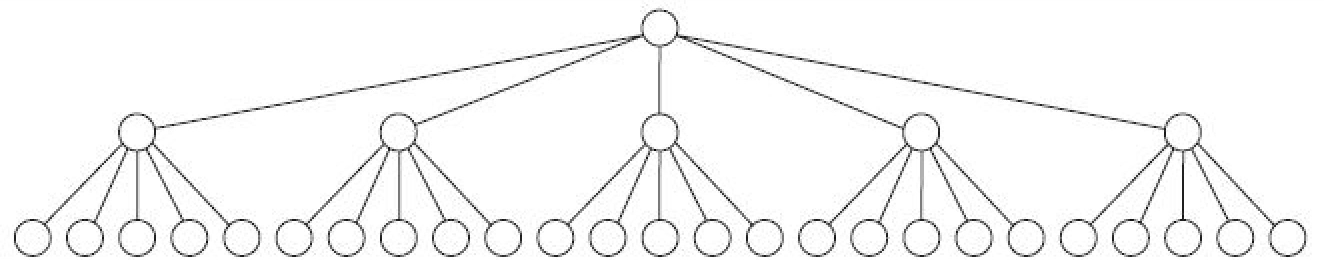
1) You'll need to know the fs blocksize
2) You need to know the pointer size for your architecture
3) You need to know the total size of a record (struct)
   a) How big are int, float, double, char, etc?
   b) Simple solution is to ask the computer: sizeof(type)
4) Alternatively, use specified data sizes:
   a) uint8_t - unsigned integer of 8 bits
   b) int8_t - signed integer of 8 bits
   c) uint16_t | uint32_t

```
struct product {
  int weight;
  double price;
} ;
```

sizeof(int) == 4;
sizeof(double) == 8;

Height of an M-ary tree

- BST is a 2-ary tree for a height of: log_2(N)
- M-ary tree for a height of: log_M(N)
- The bigger the M, the fewer the disk access to get to a leaf

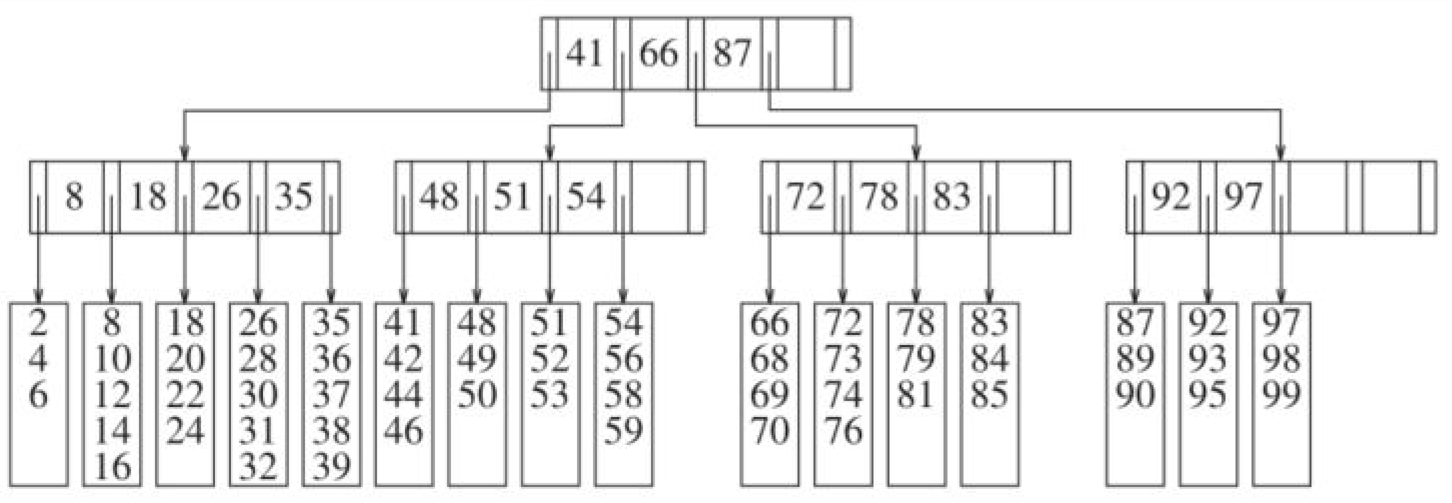


Properties of a B-tree* (really, a B+ tree)

1. The data items are stored at leaves.
2. The nonleaf nodes store up to M − 1 keys to guide the searching:
   key[i] represents the smallest key in subtree i + 1.
3. The root is either a leaf or has between two and M children.
4. All nonleaf nodes (root can be smaller) have between [M/2] and M children.
5. All leaves are at the same depth and have between [L/2] and L data items.

* This is a B+ tree, an old skool B-tree stores data at all nodes, not just leaves
When you search online for B-tree, you’ll get both kinds of trees, FYI

B-Tree Example of order 5



Calculating B-tree order M

- Block: 8,192 bytes (8k)
- Key size: 32 bytes
- Order M, so M-1 keys: (32M - 32) bytes
- Branch (pointer) of 4 bytes (32 bit system? - old skool!) for 4*M bytes
- Total size of a node: 36M - 32
- Max size is 8,192 bytes, so M = 228
  - 36M - 32 <= 8,192 (solve for M ->) 36*228 - 32 <= 8,192)
  - Always round down! Actual answer is: M = 228.4444, but can't store 0.4444 of a key + ptr

Calculating size of leaf nodes: L

- L - number of data records stored in a leaf node
- Blocksize: 8,192 bytes
- Data record: 256 bytes
- L = floor(8,192 / 256) = 32

With an optional pointer:

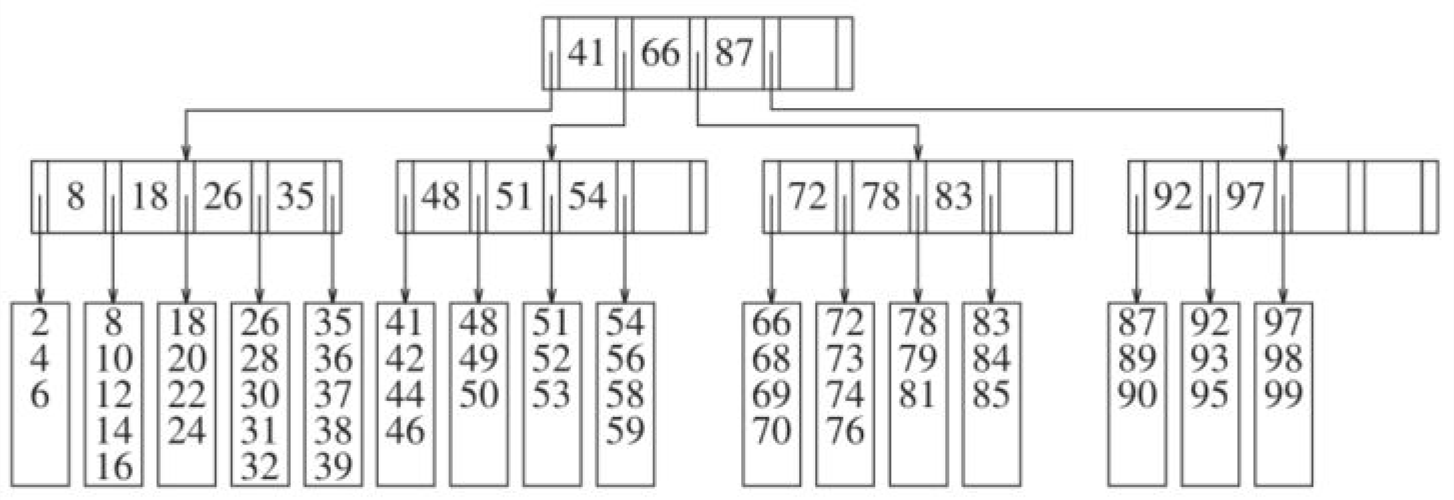
256 L - 4 <= 8,192   (Solve for L ->)   L = (8192 - 4) / 256 = 31.98 (31)

Can nodes be stored in memory?

- Why all this talk of disks?
- Why not just keep it in memory?
- Yes, they can be kept in memory, but if it grows too big what happens?
- Relying on the OS to swap your process out intelligently is asking for trouble since the kernel is designed for general purpose work, not ours
- Also, data size is growing exponentially these days. Plan for the worst
    - Every 2 days humanity is creating as much data as it did up until 2003...
        - about 2.5 Exabytes per day.
    - https://techcrunch.com/2010/08/04/schmidt-data/

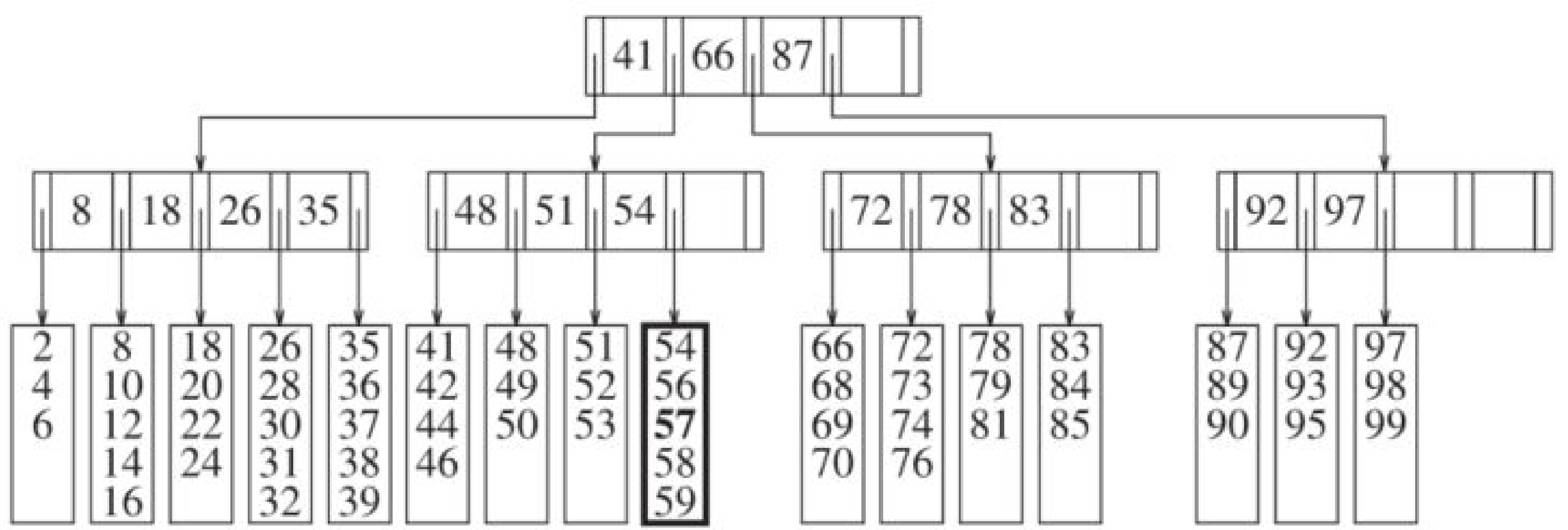
B-tree: Insert, Exists, Delete

- Start at root, find pointer to next node (or leaf) based on keys, recurse
- Insert into leaf node if there's room: insert(57)



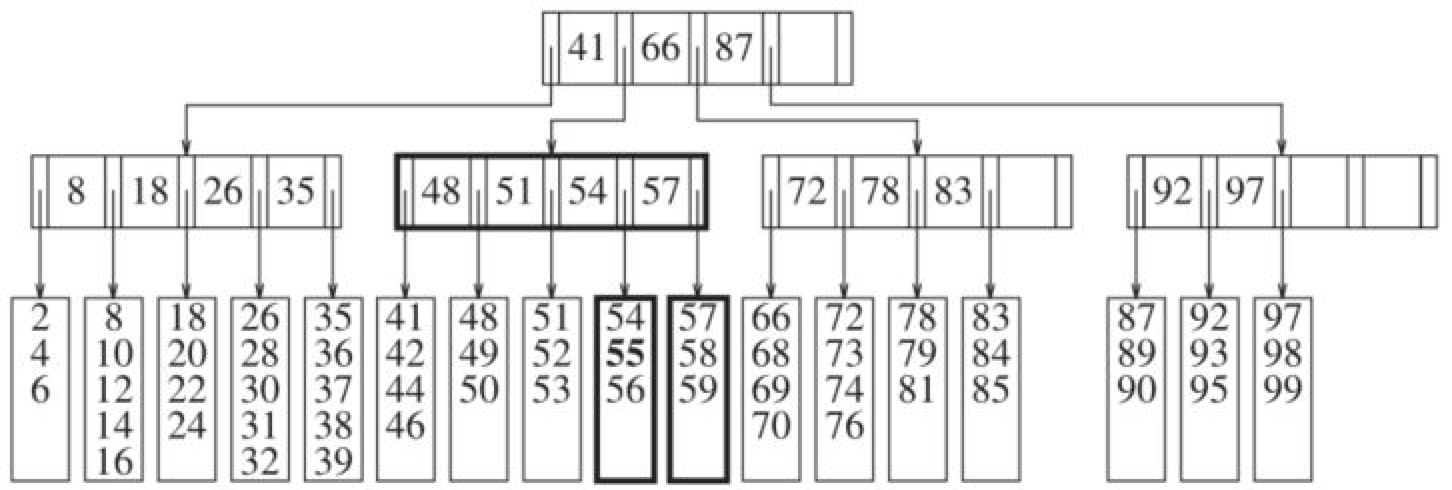
insert(57) result

- There was room in the leaf node, so it just gets added
  - Have to read 3 blocks and write 1



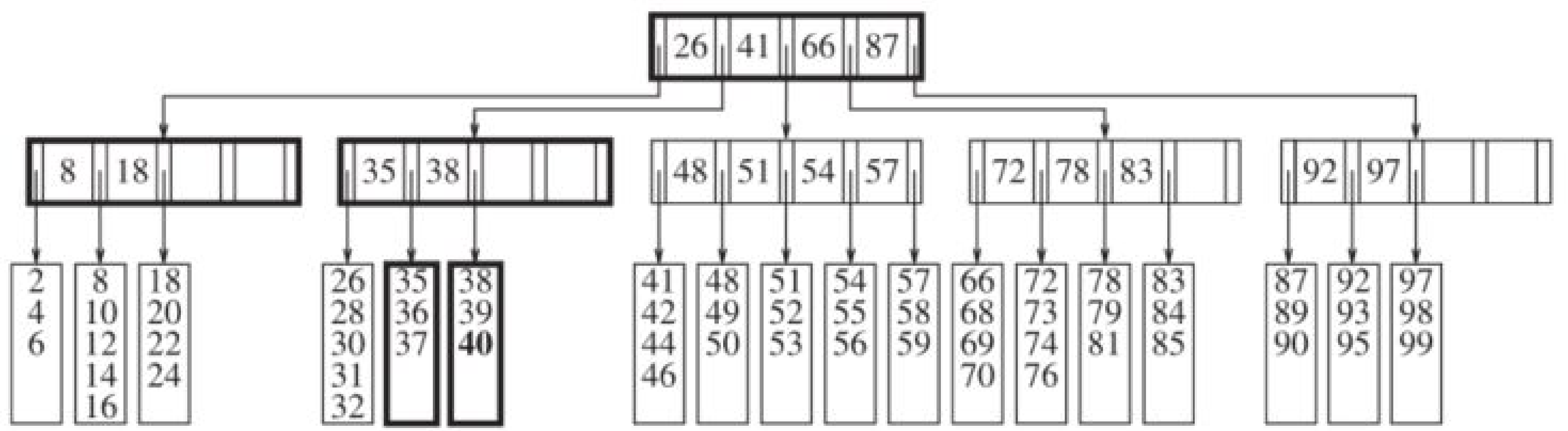
insert(55) causes a leaf split

- insert(55) doesn't have room, so we split the leaf and update the node
- New leaves are guaranteed to have [L/2] or greater elements (3 rd, 3 wr)



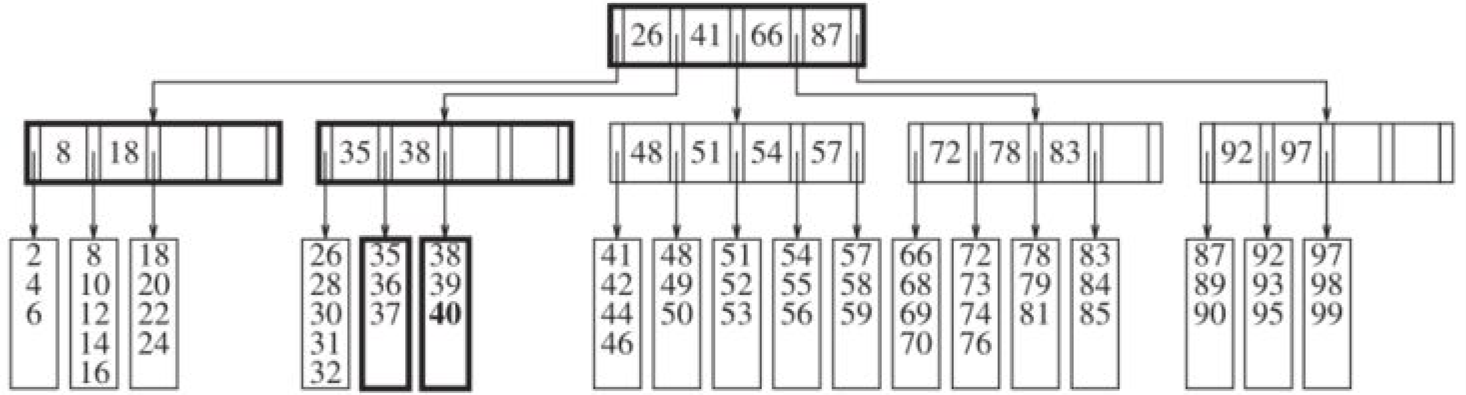
insert(40) causes a node split

- insert(40) fills a leaf, then the parent node
- Parent splits recursively, up to root - 3 reads, then 5 writes

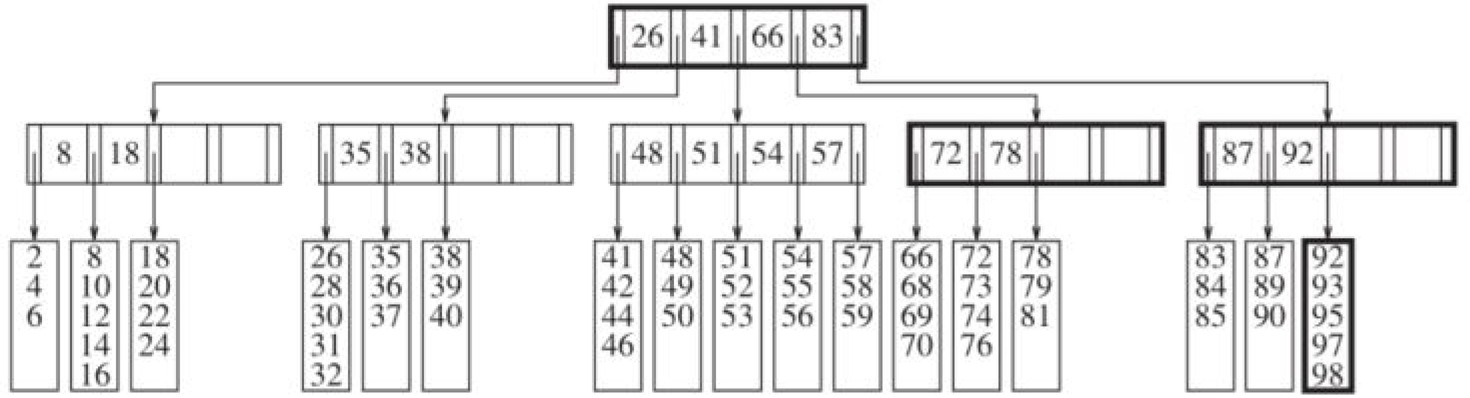


delete

- Delete from leaf first, but…
- If leaf falls below minimum:
  - The tree MUST preserve the M/2 and L/2 minimums in the nodes so...
  - Borrow from node's other leaves (if they're not at the minimum)
  - Could merge with other node (if they are at the minimum) [M/2 + M/2] should have room
    - Can cause recursive node mergers, eventually shrinking the tree if root vanishes



Delete(99) causes a leaf to merge, then the parent to borrow a leaf from next door



B-Tree summary

- Designed to align with filesystem nuances to speed up real-world searches
- Take more bookkeeping than other trees
- Allow for high speed in-order accesses if leaves are linked
- Take some more planning to fit with your hardware and filesystem
- Exploit the memory hierarchy whenever possible
- Used heavily in database design

Sets & Maps in STL

- Sets: Can be used as vector or list, but has basic efficient searching
  - Internally, a BST
  - Takes hints to guess where to insert or search (normally local to last data accessed)
    - Which can give it O(1) access times if you're right. :-)
  - Normally implemented with top-down red black trees (see Chapter 12.2)
- Maps: Actually hashes internally
  - Uses a <key, value> for storing
  - Lookups done on key alone to retrieve value
  - See chapter 5, or lectures after the midterm
  - Key can be things like strings, so you could do lookups on names

Monday: RedBlack trees

- RedBlack trees
- Probably do a note about Splay Trees on Monday too