B+ Trees, Splay Trees, Code

CptS 223 - Fall 2017 - Aaron Crandall

Today’s Agenda

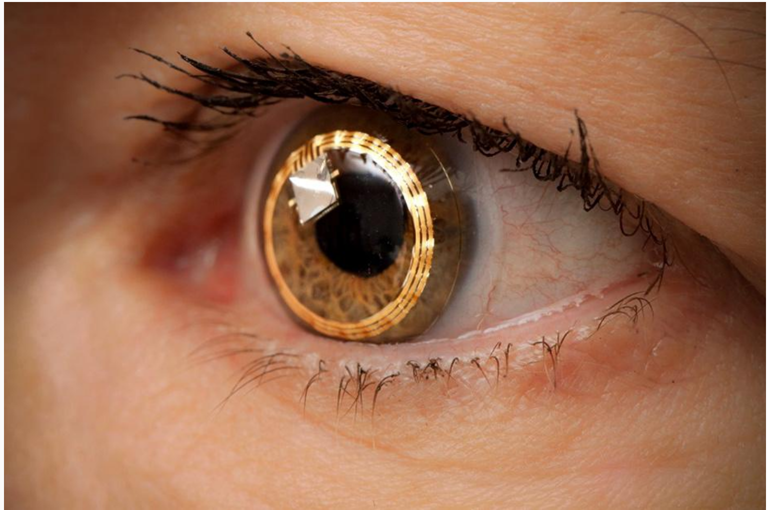
- Announcements
- Humor of the day
- HW1 and solutions
- B+ Trees
- Splay trees (wacky!)
- Code demos

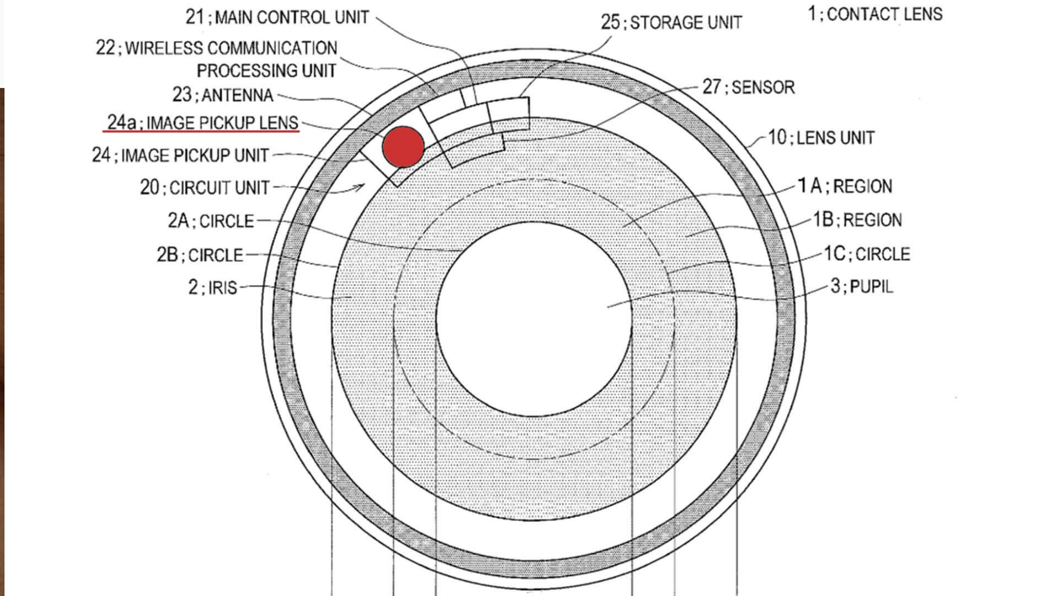
Announcements

- Wednesday will be Red-Black trees
- Friday is all midterm review:
    - Will go over chapters 1-4 + Red-Black trees
    - I promise that there will be no attendance taken, it’s a totally optional session
- Yes, we won our football game on Saturday
    - It wasn’t really a question
    - I do give props to Leach for keeping #3 in after the fumble/interception strategy

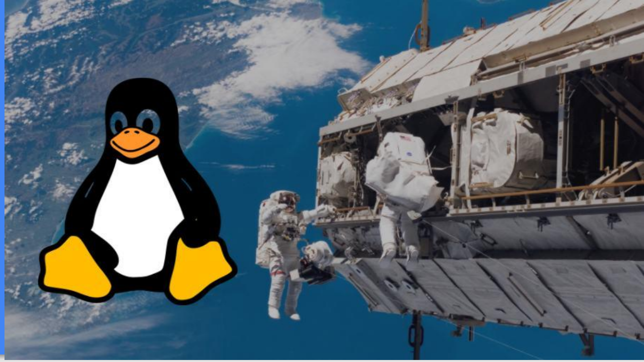
Thing of the day: Sony has patented a contact lens that is blink powered and records video

"The future is already here — it's just not very evenly distributed."
-- William Gibson





Linux in space!



“Spaceborne” Linux Supercomputer Starts Running In Space, Achieves 1 Teraflop Speed

https://fossbytes.com/spaceborne/

HPE’s Spaceborne Computer was launched into the space using SpaceX Dragon Spacecraft. This beast was launched as a result of a partnership between Hewlett Packard Enterprise (HPE) and NASA to find out how high-performance computers perform in space. Now, this supercomputer is fully installed and operational in ISS.

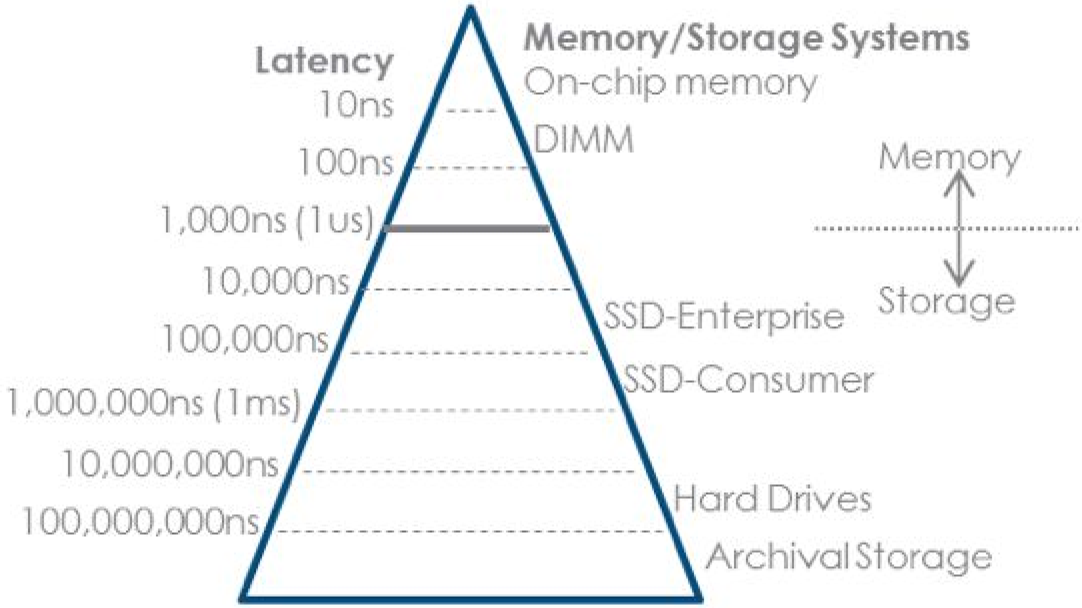
Going over HW1

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
- 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 data access O(N) if leaf nodes link to each other

The Memory Hierarchy

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



Source: Rambus

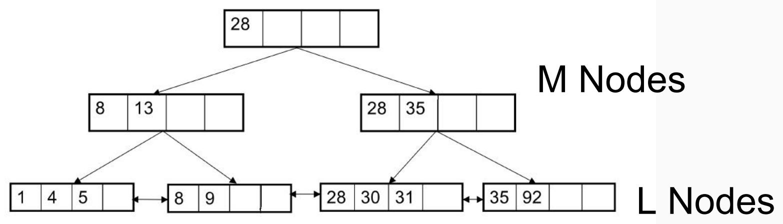
Properties of a B-tree* (really, a B+ tree)

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.

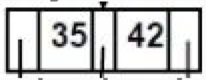
* This is a B+ tree, an old skool B-tree stores data at all nodes, not just leaves

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 many 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.



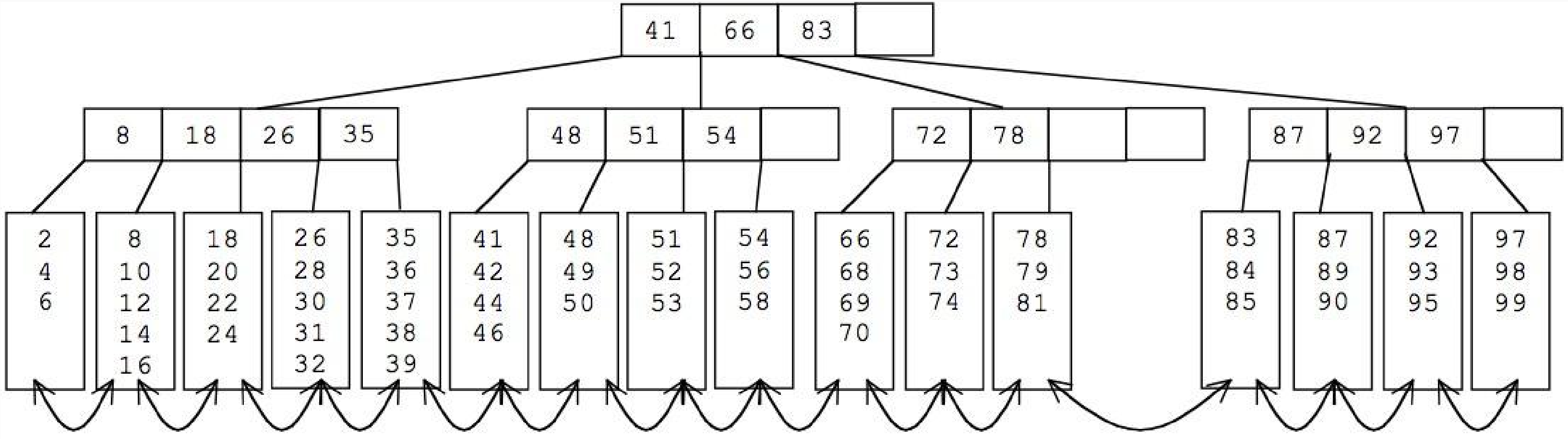
M Nodes

- Only store pointers to other nodes (M or L)
- Have M pointers and M-1 keys (how you index a record)
  - This example is a B-tree of order 3: 

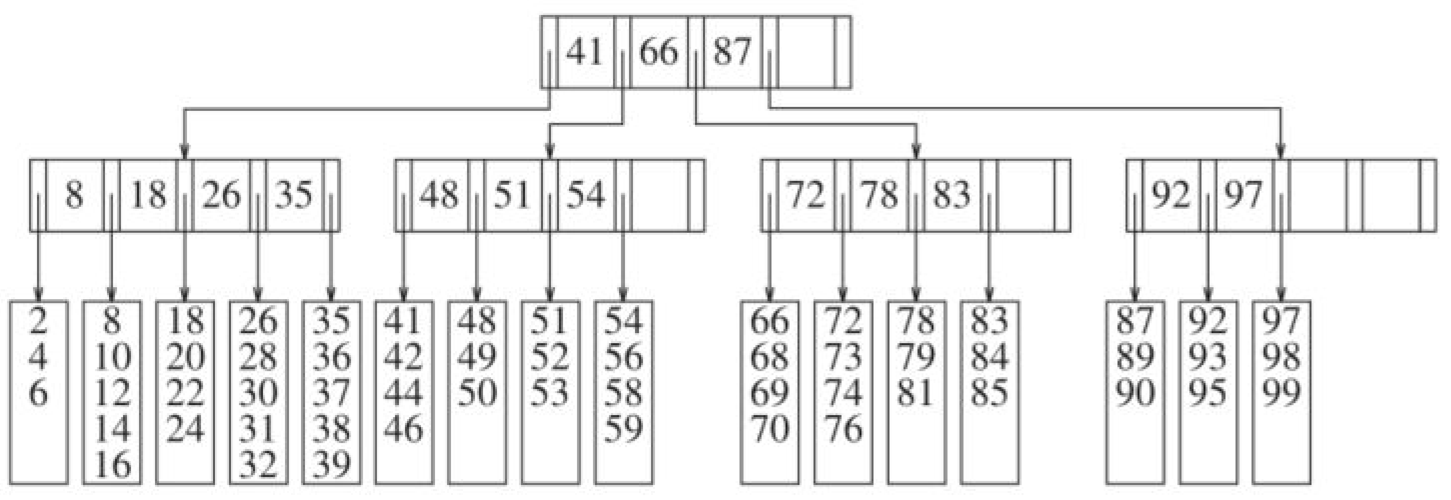
- You calculate the order by how much physical space you use in a fs block:
  - fs blocksize >= M * pointerbytes + (M - 1) * keysize
  - Round down for M
- Question: how big are pointers on your computer? (in bits)

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

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

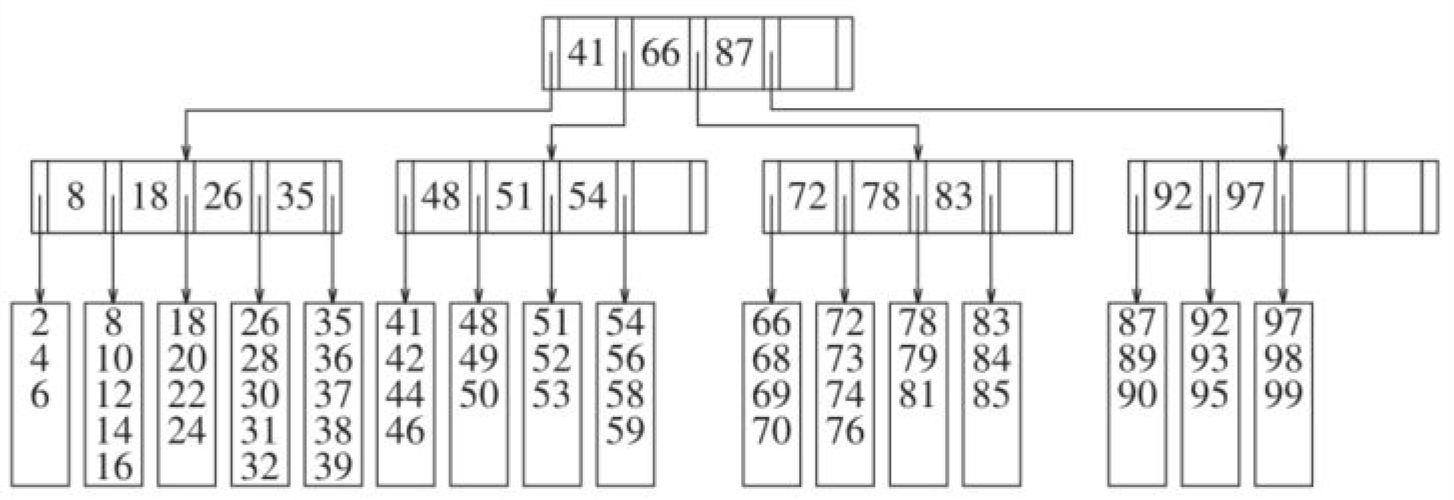


B-Tree Example of order 5



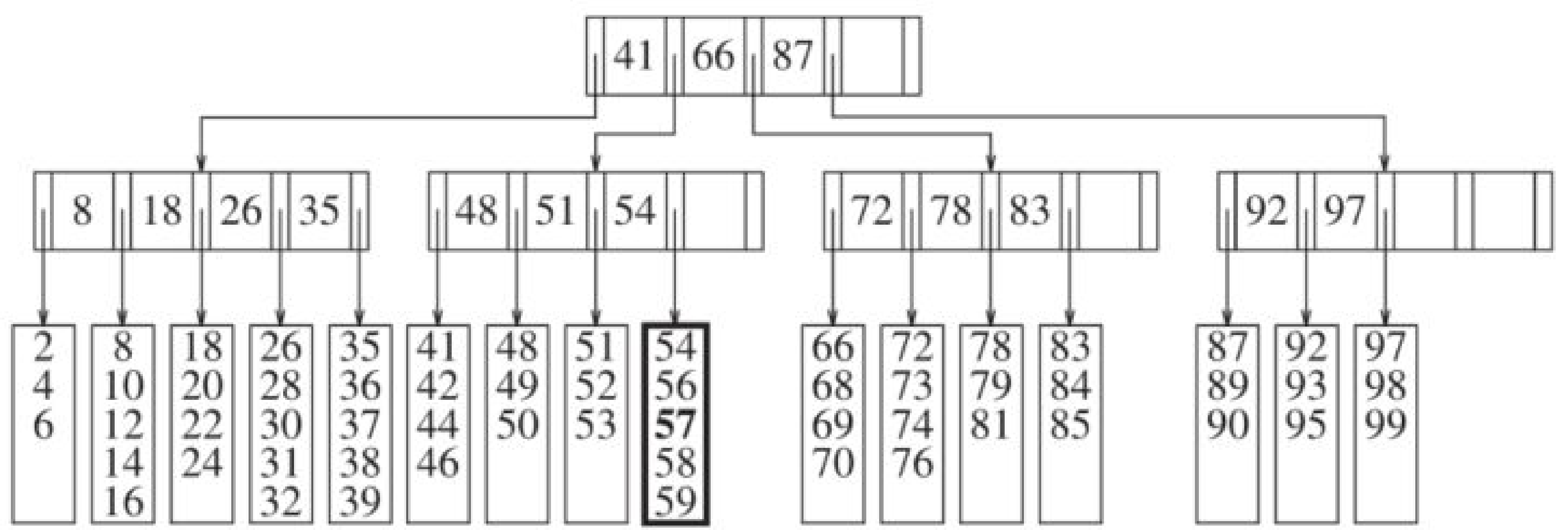
B-tree: Insert, Exists, Delete

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



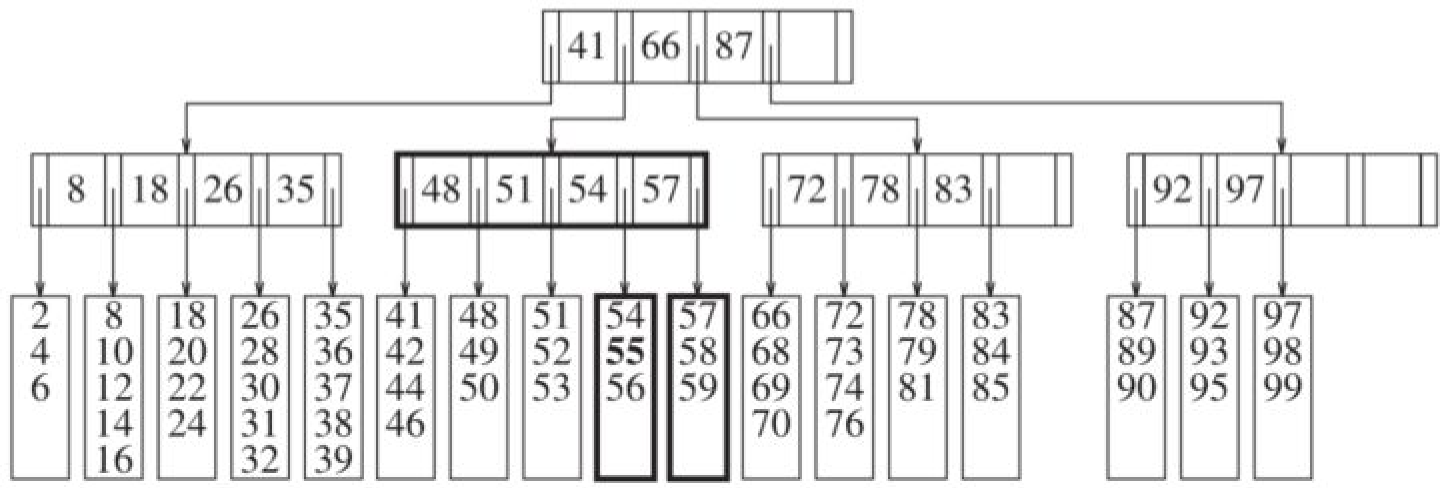
insert(57) result

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



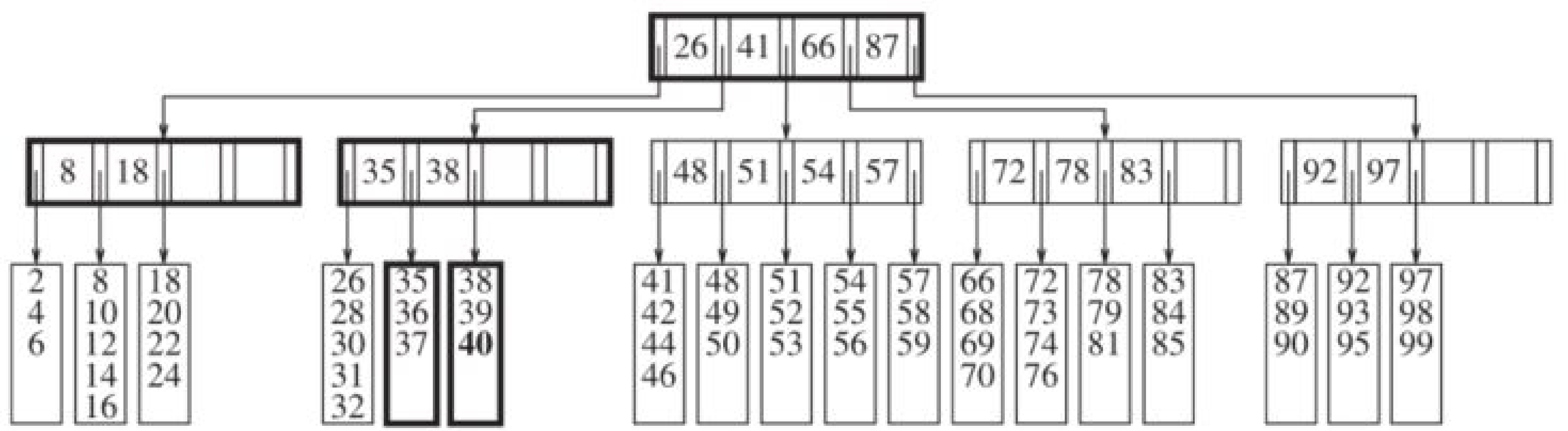
insert(55) causes a leaf split

- 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)



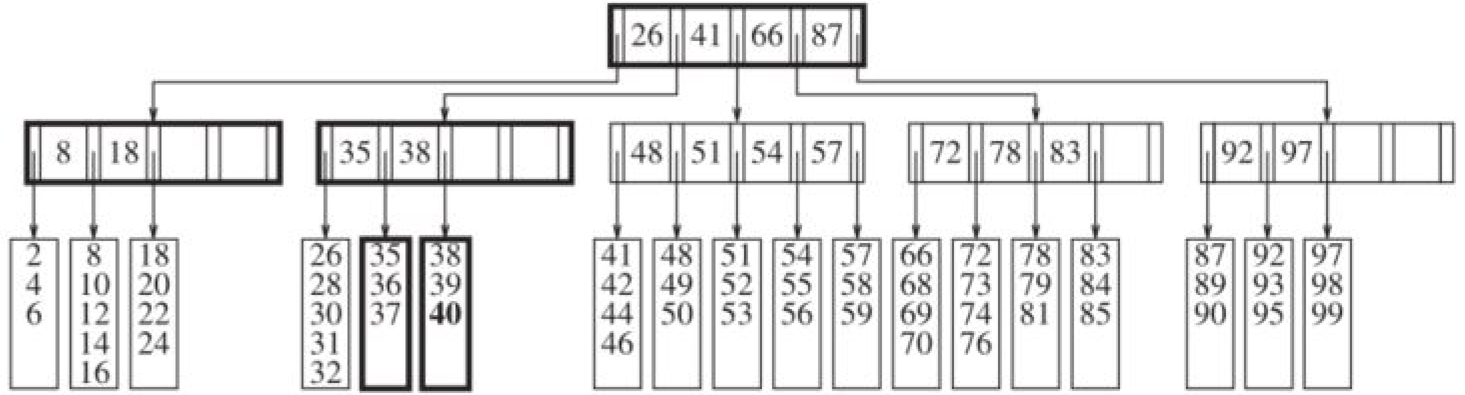
insert(40) causes a node split

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

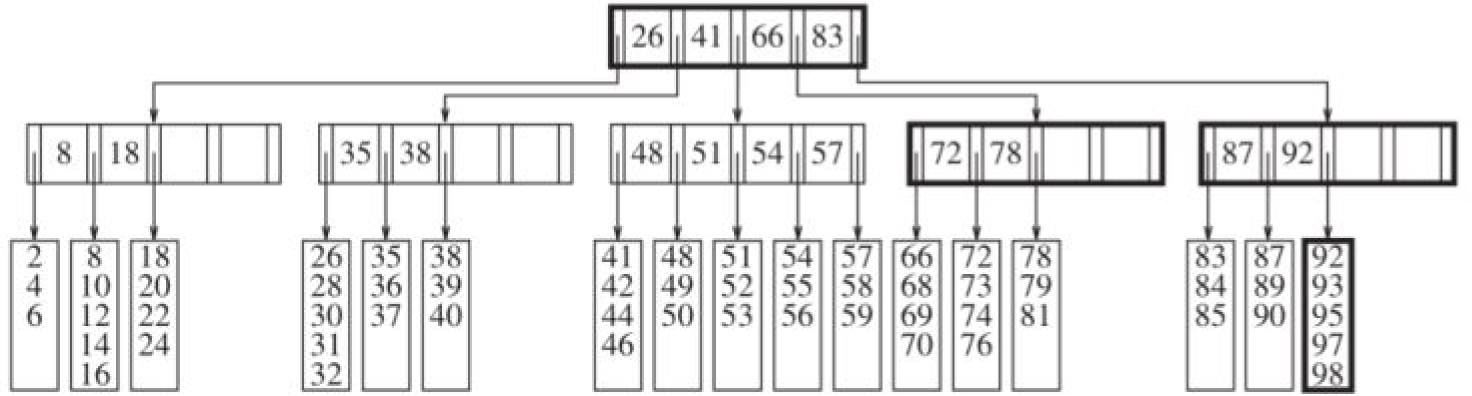


delete

- Delete from leaf first, but…
- If leaf falls below minimum:
  - 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 file system
- Exploit the memory hierarchy whenever possible
- Used heavily in database implementations
- Visualization page:
    - http://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Sets & Maps in STL

- Sets: Can be used as vector or list, but has basic efficient searching
    - Internally, a binary tree
    - 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

Splay Trees - An odd thing.

CptS 223 - Fall 2017 - Aaron Crandall

Splay trees - Very different approach

- Book does this tree in passing. I had to suss out details here and there.
- “Simple” data structure, but only by less decision making
- Does guarantee no bad input sequences (akin to the BST linked list issue)
- Ends up with an amortized running time of O(M f(N)) ~~ O(log(N)) time
- **Every** time a node is accessed (created, read, modified), it is pulled to the root
- Exploits the locality of reference: Once you access a piece of data, you’re highly likely to access it again, or one right nearby
    - https://en.wikipedia.org/wiki/Locality_of_reference

Wait! *EVERY* time a node is accessed?

- Yes, every time you read a node, you rotate and/or bring it to the root
- This means the trees is continually changing shape in significant ways
- It means we don’t keep height or balance information because we’ll always bring accessed nodes to the root, regardless of tree shape

How to move the node to the root?

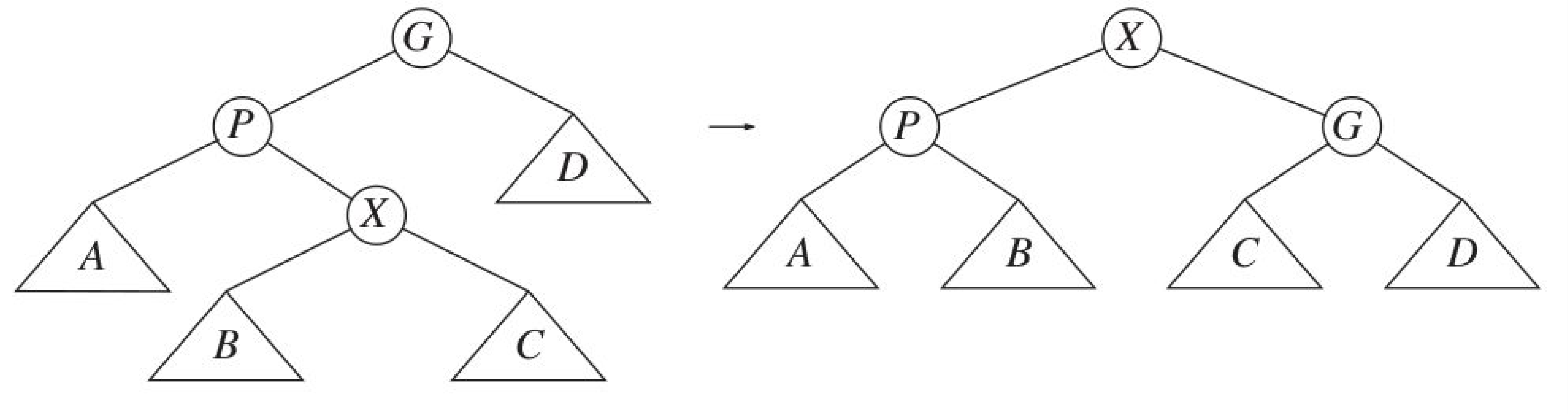
- Not by single rotations, it just makes the tree unbalanced in the other direction
- The book points out that a bad series of inserts: [1..N], single rotations looks good, but gives $\Omega(N^2)$ access times for in-order reads!
- So, instead of single rotations, we splay (zig-zag and zig-zig) instead
  - These kinds of rotations are used in Red-Black trees too

Splaying

- Zig-Zag - The same as an AVL tree double rotation
- Zig-Zig - Rotating over the accessed node until it’s the root

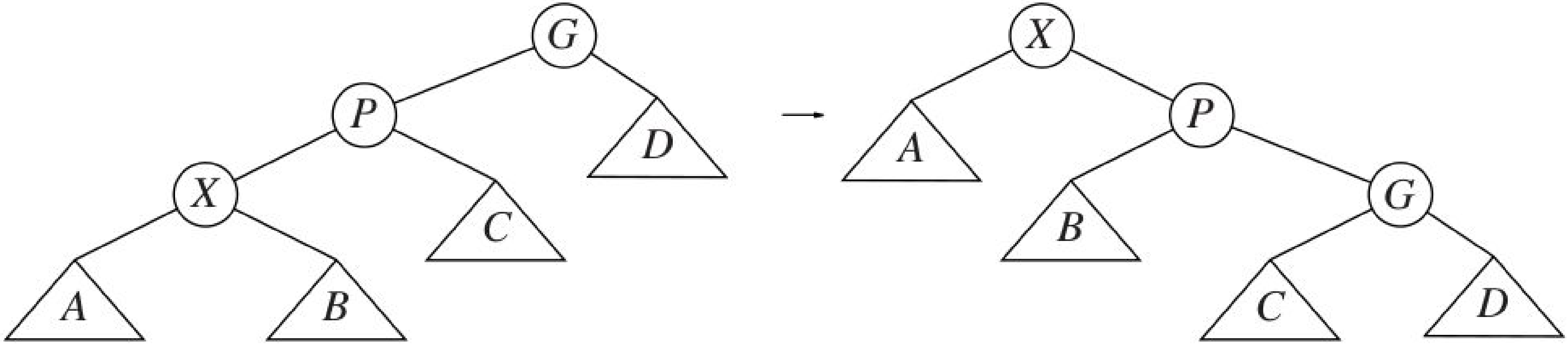
Zig-Zag - Same as an AVL double rotate

- Look at grandparent of accessed node, find double rotation case, rotate!

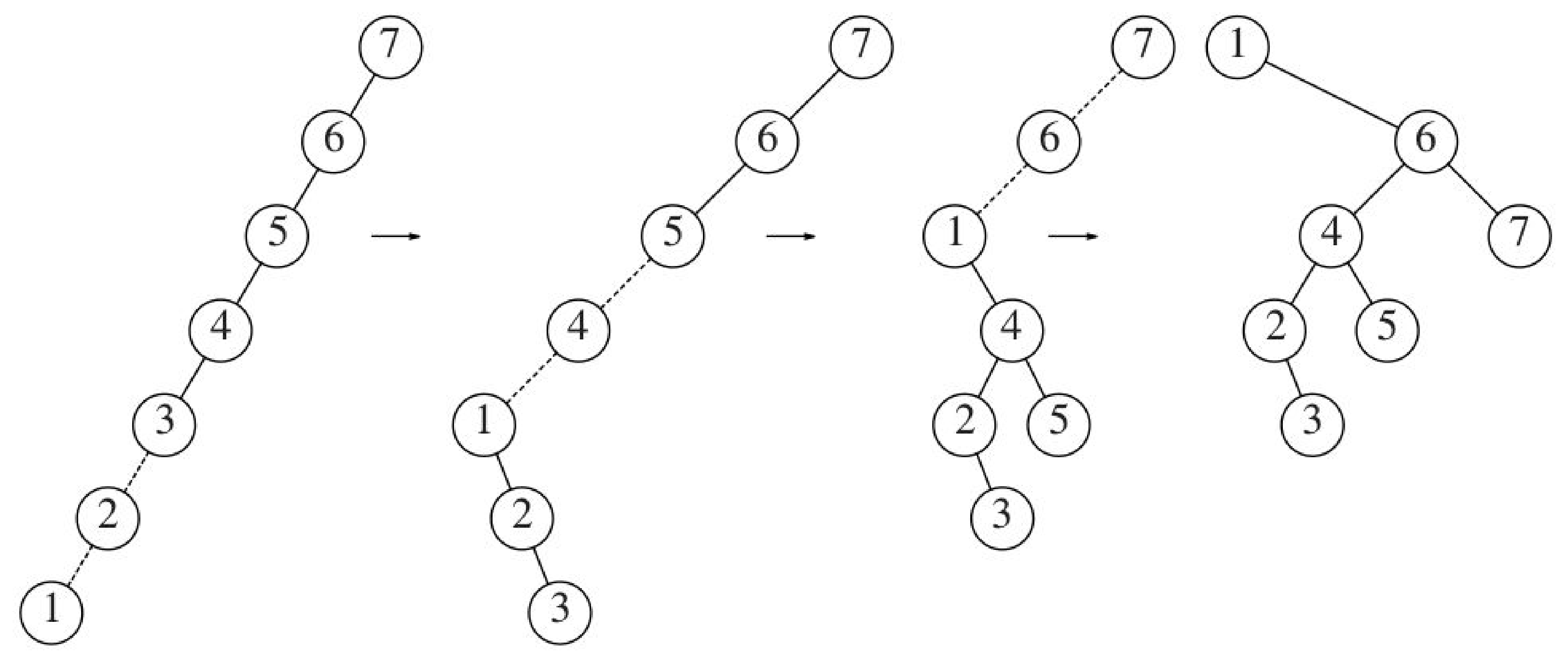


Zig-Zig - Pull accessed node to root over parent and grandparent

- Move X up to root over Parent and Grandparent



Then repeat until X is root of tree



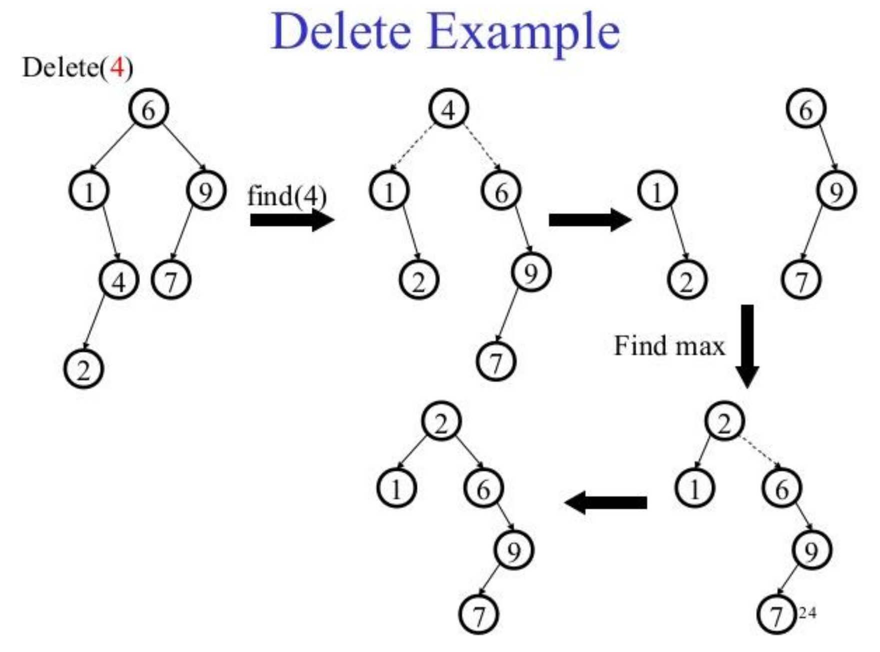
Prior example

- First access of “1” takes O(N), but then access of 2 takes ~ N/2
- Nodes along the access path end up with their height about halved
- As this continues, the overall height ends up about log(N)

Result: When accesses are long, future accesses end up cheap

Concept of locality of reference: Most programs access data in a pattern based around either space or time. Splay trees help this by bringing relatively local values higher up the tree for the next access

Deleting from a Splay Tree?



1) Access node to be deleted
    a) Which brings it to the root
2) Treat children as two subtrees
3) Access max node in left tree
    a) Which brings it to the root of the left tree, and it has no right child
4) Make the right tree the child of the new root of the left tree
5) Forget the node to be deleted
6) Done!

Lookin’ at code - if there’s time.

AVL tree reminder

- Enforces balance of height difference of sub trees of no more than +/- 1
- Access time of log(N) by rotations on insert/delete
- L-L and R-R inserts take single rotations
- L-R and R-L inserts take double rotations