Ch18: B-Trees

23 Oct 2012
CMPT231
Dr. Sean Ho
Trinity Western University

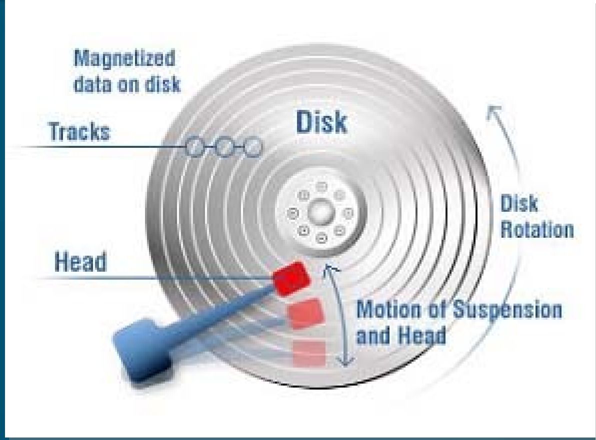
Outline for today

- B-Trees
  - Motivation: properties of spinning disks
  - B-tree concept
  - Search in $O(t \log_t n)$
  - Insert in $O(t \log_t n)$
  - Delete in $O(t \log_t n)$
  - Application to filesystems
- Midterm review (ch6-8, 11)

Balancing search trees

- Complexity of most tree ops depends on height
  - Search, insert, delete
  - Worst case: tree becomes a linked list
- How to keep tree balanced, bushy (low height)?
- BSTs with tree rotations:
  - Red-black trees (ch13)
    - Levels alternate colour: longest path ≤ 2 * shortest
  - AVL trees
    - Rotate after insert/delete
  - Splay trees
    - Search/ins/del rotate node to root and rebalance

Trees for disk storage



- Accessing a spinning disk:
  - Seek: move head to desired track, wait until desired sector comes to head (slow)
  - Throughput: reading consecutive sectors (fast)
- Lots of small iops (I/O operations/sec) are bad
  - ⇒ so buffer and do I/O in large pages at a time
    - Minimise # disk accesses (also good for network fs)
  - Read pages to RAM, modify, and write back
    - Only a limited # pages can be in RAM at a time
- Tree-based disk filesystem: 1 node ↔ 1 page
  - Very low, bushy tree with large degree

victimoftechnology.com

Outline for today

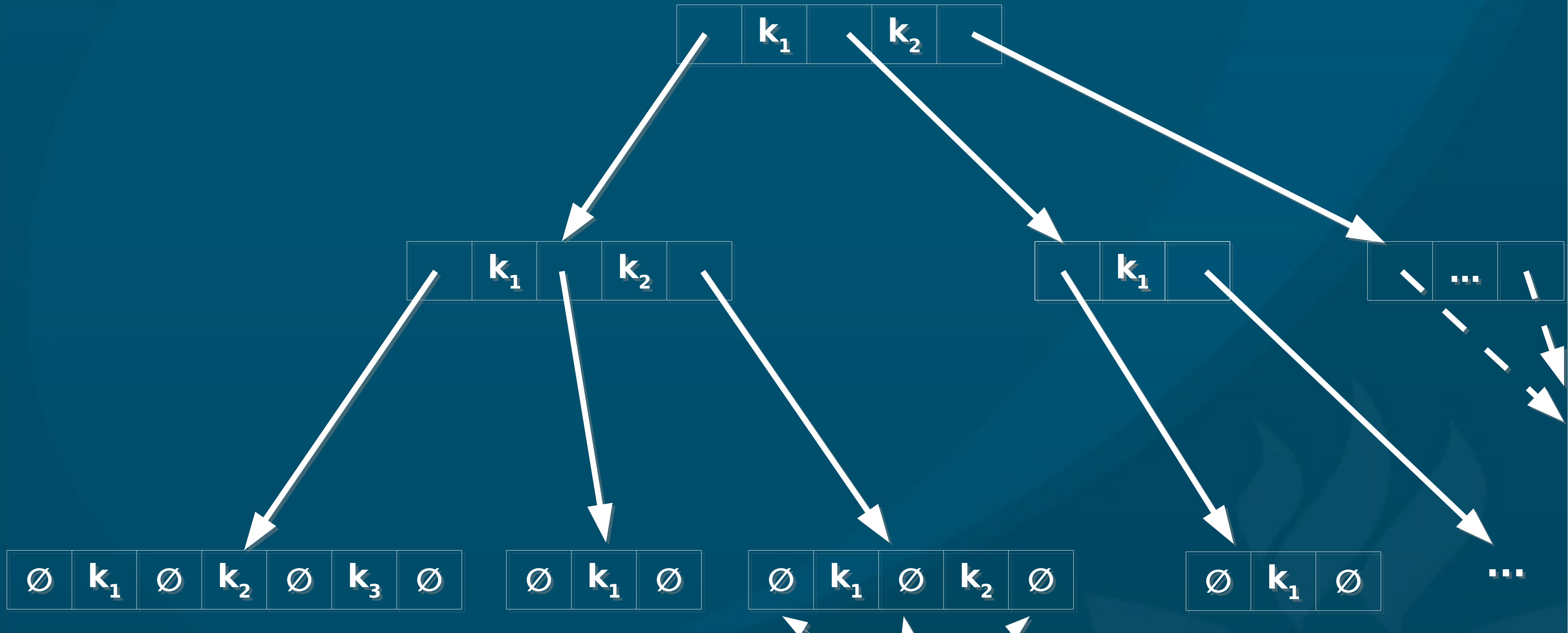
- B-Trees
  - Motivation: properties of spinning disks
  - B-tree concept
  - Search in $O(t \log_t n)$
  - Insert in $O(t \log_t n)$
  - Delete in $O(t \log_t n)$
  - Application to filesystems
- Midterm review (ch6-8, 11)

B-trees

- Generalisation of BST: (left) ≤ key ≤ (right)
- In a B-tree of min-degree t, every node has:
  - nkeys sorted keys (t-1 < nkeys < 2t-1)
  - (if non-leaf) nkeys+1 links to child nodes, interleaved between the keys
    - Hence degree is between t and 2t
- All leaves are at same depth h
  - For a tree of min-degree t and height h, what are min/max # of keys stored?
- B+-tree: data/payload stored in leaves
- B*-tree: 2t-1 < nkeys < 3t-1

B-tree with t=2

- Also called 2-3-4 tree:



data (logical block addrs) go here in a B+ tree

B-tree operations

- Standard search tree interface:
  - Search, insert, delete
- Track not only CPU complexity, but also # disk accesses: read()s & write()s
  - Complexity in terms of t and $h = \Theta(\log_t n)$
  - Constrained variable degree (between t and $2t$) keeps tree balanced
- Keep root node in RAM
  - Other nodes need to be read from disk
  - Root needs to be written to disk if modified

B-tree: search

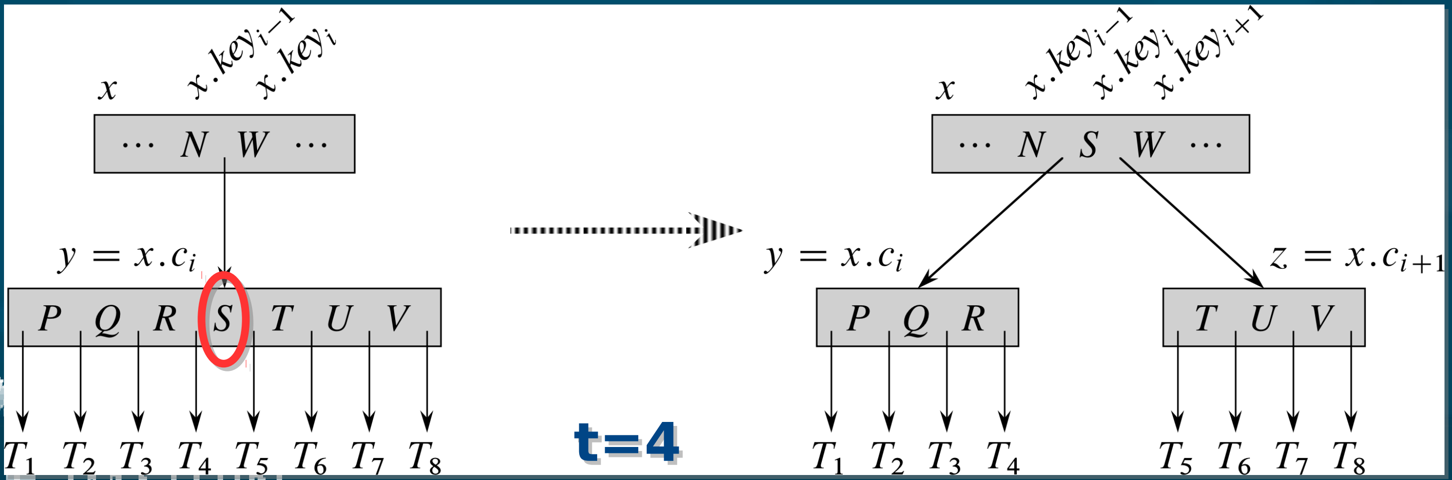
- search(node, key):
  - // Linear search for the right key
    for (i = 1; i ≤ node.size and key > node.key[i]; i++)
  - // Found it in this node!
    if i ≤ node.size and key == node.key[i]:
    return (node, i)
  - // Not here and we're a leaf
    if x.isleaf: return NULL
  - // Load child node from disk and recurse
    read(node.child[i])
    return search(node.child[i], key)
- Complexity (worst-case): $O(th) = O(t \log_t n)$
- Disk accesses (worst-case): $O(h) = O(\log_t n)$

Outline for today

- B-Trees
  - Motivation: properties of spinning disks
  - B-tree concept
  - Search in $O(t \log_t n)$
  - Insert in $O(t \log_t n)$
  - Delete in $O(t \log_t n)$
  - Application to filesystems
- Midterm review (ch6-8, 11)

B-tree: insert

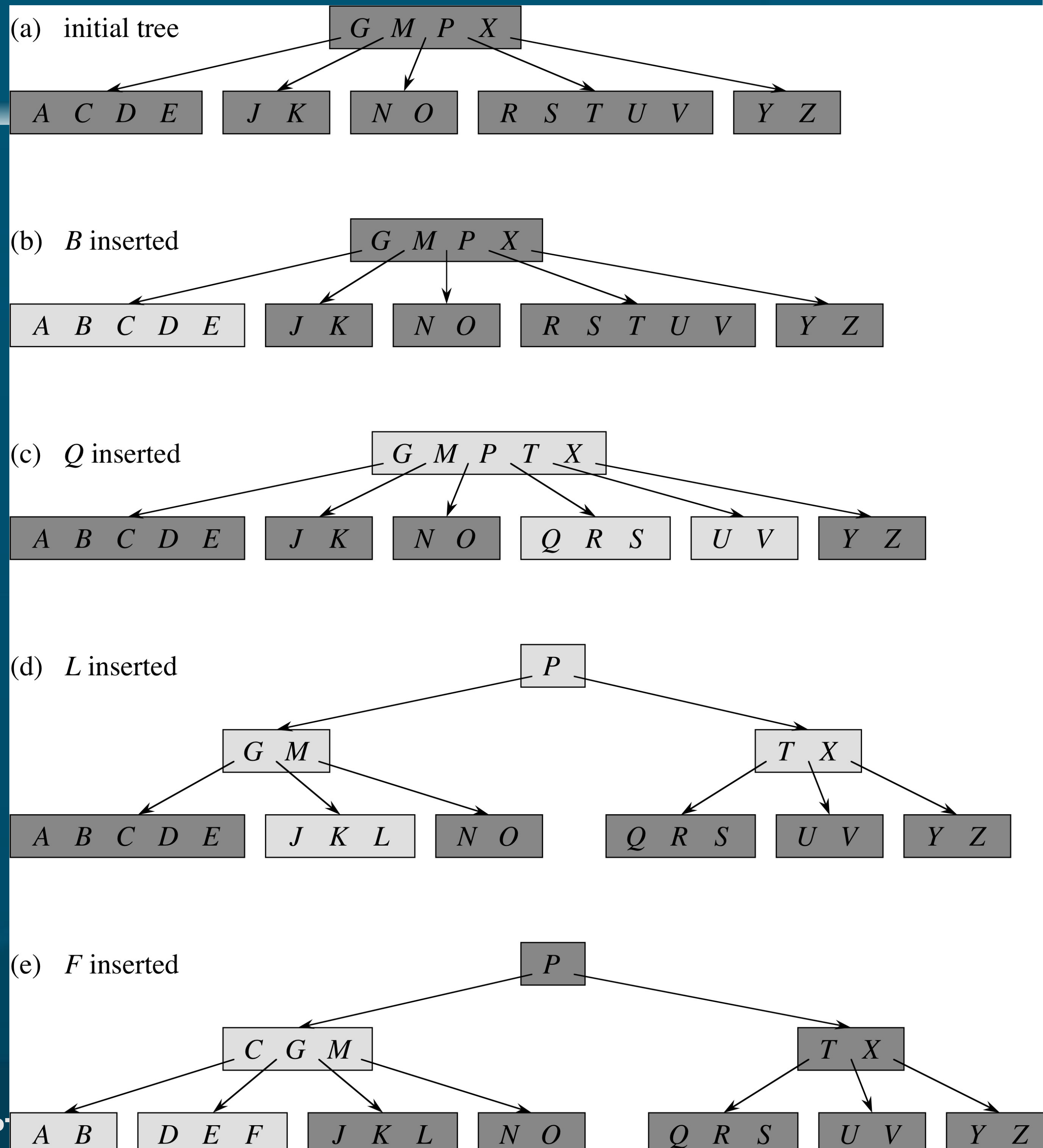
- As with BST, first search for where key should go
  - Search down to leaf node
- If leaf node is not full, just insert new key there
- If leaf node is full (2t-1 keys), need to split:
  - Create two nodes each with t-1 keys
  - Median key (key_t) moves up to parent
  - Iterate on parent, splitting as needed



CPU: O(th)
Disk: O(h)

Insert ex.

- (a) initial tree (t=3)
- (b) insert into non-full leaf
- (c) insert into full leaf: split
- (d) insert and split up to root
- (e) 1-level split

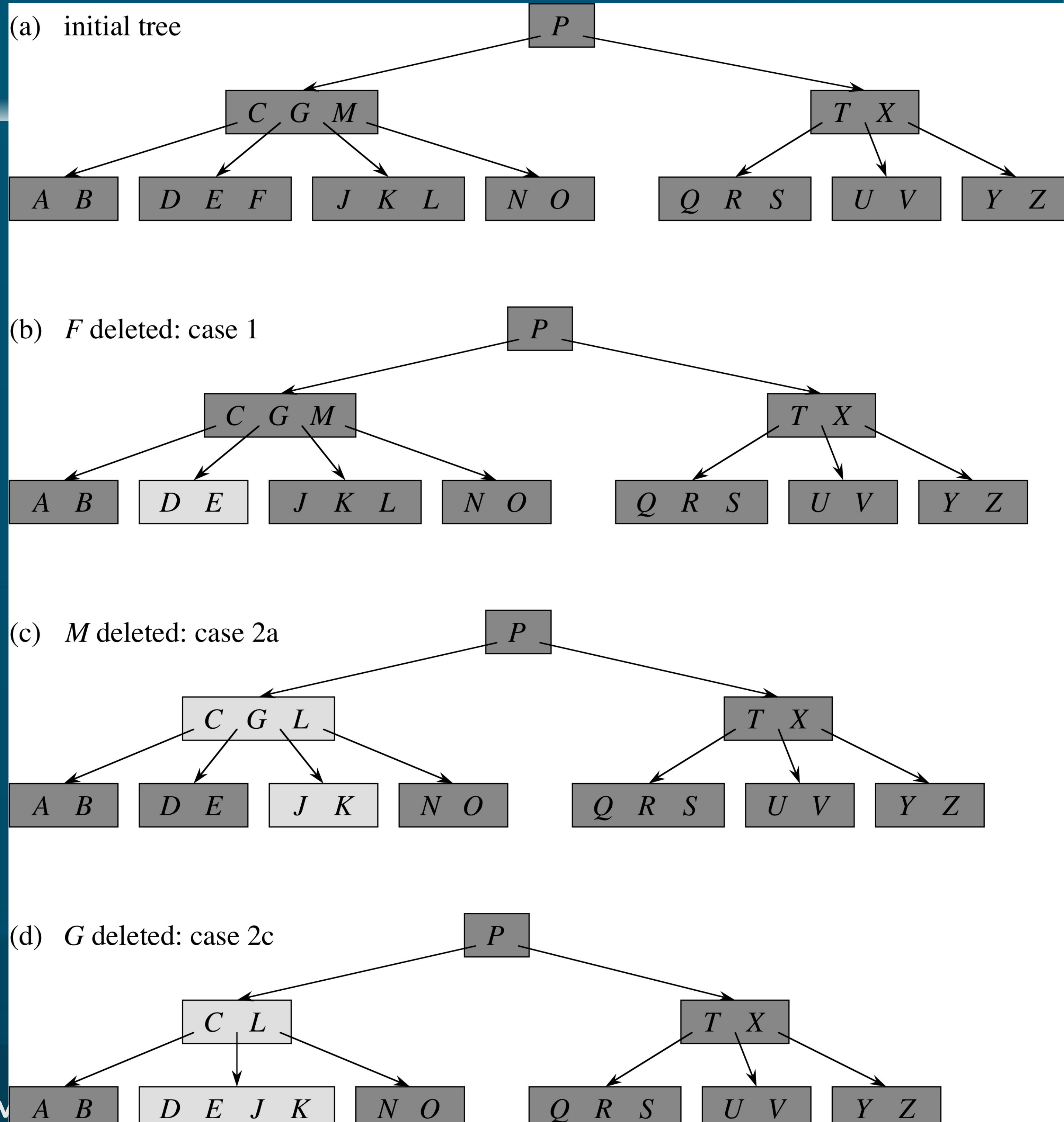


B-tree: delete

- Descend tree, ensuring each node has $\geq t$ keys before we examine it (space for deletion):
- If key is in node and it's a leaf, just delete it
- If key is in node and it's not a leaf:
  - If left child has $\geq t$ keys, replace w/predecessor
  - If right child has $\geq t$ keys, replace w/successor
  - Else, merge left+right children & delete key
- If key is not in node and it's not a leaf:
  Find the child that key should be in: if $t-1$ keys,
  - If left/right sibling has $\geq t$ keys, steal one
  - Else, merge child with a sibling

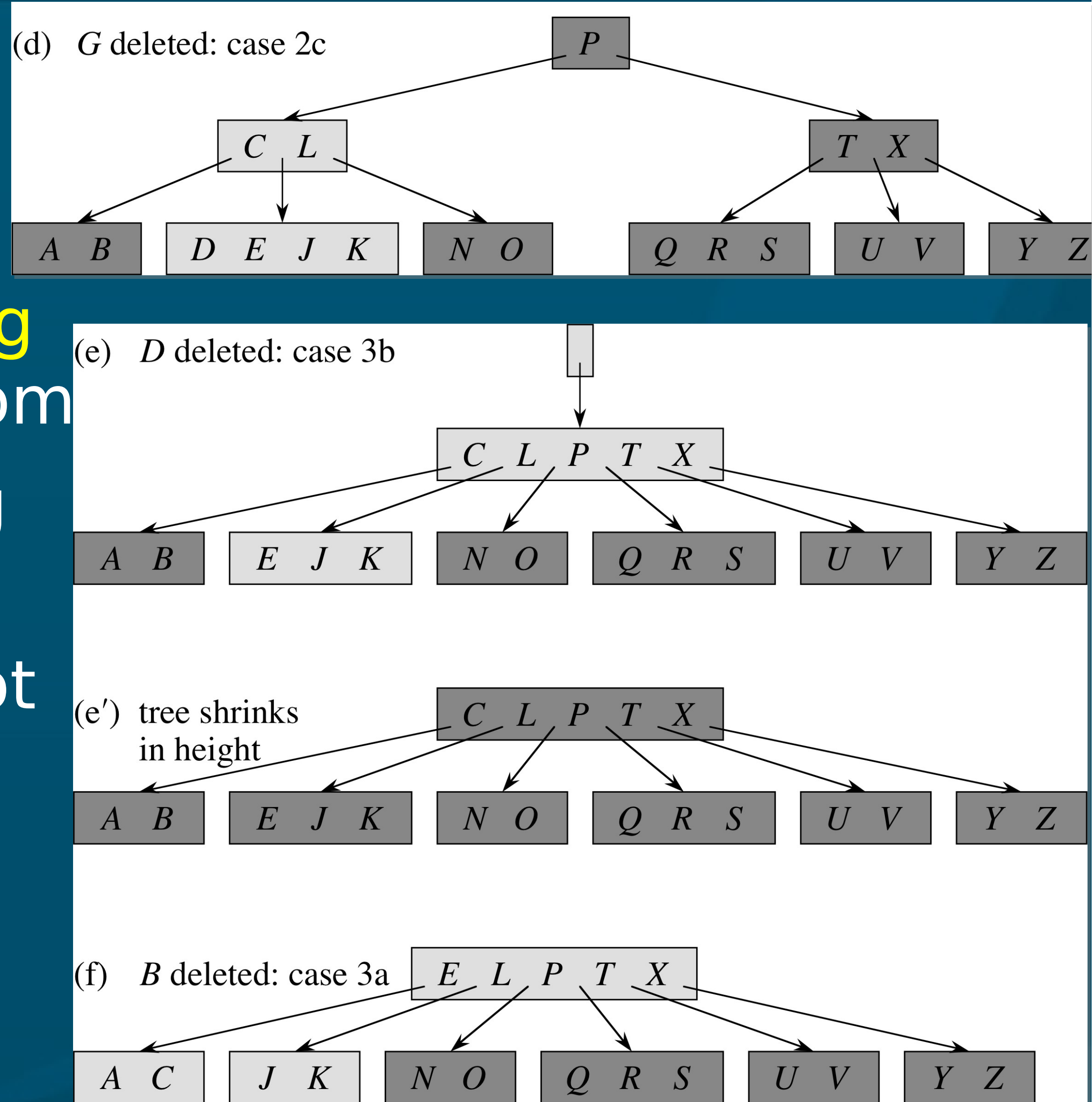
Delete

- (a) (t=3)
- (b) internal nodes ≥ t, key in leaf
- (c) key in non-leaf: use predecessor
- (d) key in non-leaf: merge children



Delete, cont.

- (e) internal node CL too small, and sibling too small to steal from
  - ⇒ Merge w/sibling
- (e') Merging pushes key P down from root
- (f) B not in CLPTX, AB child too small:
  - ⇒ Steal from EJK



B-tree summary

- Generalisation of BST, but:
  - All leaves at same height h (= $\Theta(\log_t n)$)
  - Degree of each node is between t and 2t
- Operations:
  - Create: CPU O(1), disk O(1)
  - Search, insert, delete: CPU O(th), disk O(h)
  - When modifying tree, need to ensure that degree of every node stays between t and 2t (so # keys is between t-1 and 2t-1)

Outline for today

- B-Trees
  - Motivation: properties of spinning disks
  - B-tree concept
  - Search in $O(t \log_t n)$
  - Insert in $O(t \log_t n)$
  - Delete in $O(t \log_t n)$
  - Application to filesystems
- Midterm review (ch6-8, 11)

B-trees in filesystems

- Filesystems store: files, directories, metadata (e.g., name, owner, permissions, update time)
- Contents of files are in (1 or more) extents on disk
  - Logical Block Addresses interpretable by HDD
- Filesystems can use B-trees for lookup tables:
  - Inode table: metadata for each object
    - Indexed by inode, unique to each object
  - Extents table: LBAs for each extent
    - Or the actual data, if it's small enough
  - Journal: transaction log
    - Preserve integrity in case a long write fails

Filesystems that use B-trees

- NTFS indexes (inode tables)
- Mac HFS catalog records (inodes): B+-trees
- Linux ext3/ext4 directory indexes: Htrees
  - store hashed filenames for fast lookup
- Linux BTRFS (“B-TRee FileSystem”):
  - B-trees used for everything:
    - Directory trees (with hashed filenames)
    - Extent tree (file data as LBA or actual data)
    - Log tree
    - Root tree storing links to all other trees!
    - … much more

Outline for today

- B-Trees
  - Motivation: properties of spinning disks
  - B-tree concept
  - Search in $O(t \log_t n)$
  - Insert in $O(t \log_t n)$
  - Delete in $O(t \log_t n)$
  - Application to filesystems
- Midterm review (ch6-8, 11)

Review for exam2: ch6-8, 11

- Hand-simulation, complexity analysis
- Ch6: Heapsort
  - Trees
  - Max heaps: max-heap property, heapify()
  - Heapsort: building a heap, using it for sorting
  - Priority queue: ops, complexity
- Ch7: Quicksort
  - Naive quicksort with fixed pivot
  - Randomised pivot
  - Complexity analysis: expected running time E[]

Review for exam2: ch6-8, 11

- Ch8: Linear-time sorts (assumptions!)
  - Decision tree model, why $\Omega(n \lg n)$ comparisons
  - Counting sort (census + move): $\Theta(n + k)$
  - Radix sort (with r-bit digits): $\Theta(d(n + k))$
  - Bucket sort: $\Theta(n)$ expected time
- Ch11: Hash tables
  - Hash function, hash collisions, chaining
  - Load factor α = n/(# buckets), search in $\Theta(1+\alpha)$
  - Hashes: div, mul, universal hashing
  - Open addressing: linear, quad, double-hash