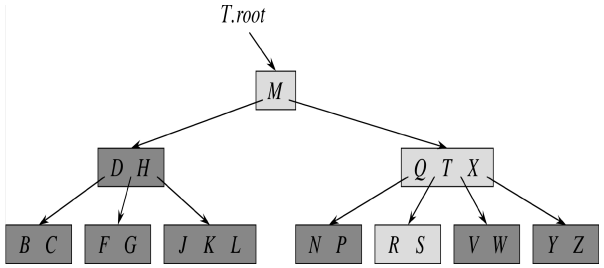


B-Trees
Lecture 18

COSC 242 – Algorithms and Data Structures

Today’s outline

1. Overview
2. Disk storage
3. B-trees
4. Searching
5. Inserting
6. Deletion
7. Cases

Today’s outline

1. Overview
2. Disk storage
3. B-trees
4. Searching
5. Inserting
6. Deletion
7. Cases

Overview

B-trees are balanced search trees designed to work well on disks or other direct access secondary storage devices.

B-trees are similar to RBTs, but they are better at minimizing disk I/O operations. Many database systems use B-trees, or variants of B-trees, to store information.

B-trees differ from red-black trees in that B-tree nodes may have many children, from a few to thousands.

Overview

The “branching factor” of a B-tree can be quite large, although it usually depends on characteristics of the disk unit used.

Similar to red-black trees in that every n-node B-tree has height O(lg n)

The exact height of a B-tree can be considerably less than that of an RBT because its branching factor.

B-trees generalise binary search trees.

Today’s outline

1. Overview
2. Disk storage
3. B-trees
4. Searching
5. Inserting
6. Deletion
7. Cases

Disk storage

Balanced BSTs like RBTs are great for data structures that are small enough to fit in main memory. But what happens when we need to use external storage?

Type	Data Speed	Size	Min addressable unit
DDR4 Ram	≤ 25 GB/s	≤ 128 GB (macOS, Win10 64 Home)*	8 bytes
SSD	500 MB/s - 3.5 GB/s	< 4 TB	256 KB – 4 MB
HDD	180 MB/s	< 16 TB	4 KB

Typically, secondary storage exceeds primary memory by at least two orders of magnitude.

** Many systems nowadays come with ~8 GB RAM, but usable amount is less*

Disk storage

For storage that has relatively slow read/write speeds, it makes sense to design data structures that minimise the number of reads/writes in order to access the data.

It also makes sense to have data structures that use the minimum addressable unit as their base node size.

B-trees do both and are commonly used for database applications.

B-trees and their variants are also used in most major file systems: HFS+ (macOS), ext4 (Linux), and NTFS (Win10).

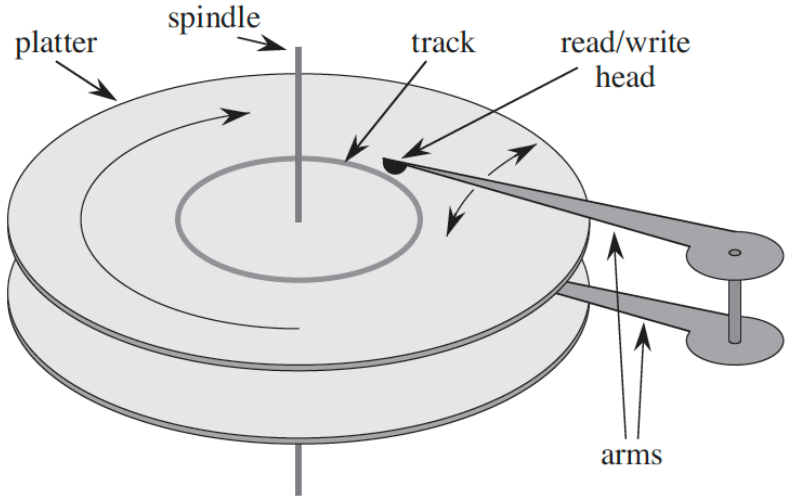
Pages

A page is a fixed-length contiguous block of virtual memory.

Pages have particular utility in HDD. Physical movement of the drive arm is slow, so to optimise the time outlay, the disk will access one or more pages in a single operation.

Pages (and blocks) also play a role in an SSD[1].

Typically, a B-tree node is as large as a whole disk page. This size limits the number of children a B-tree can have.



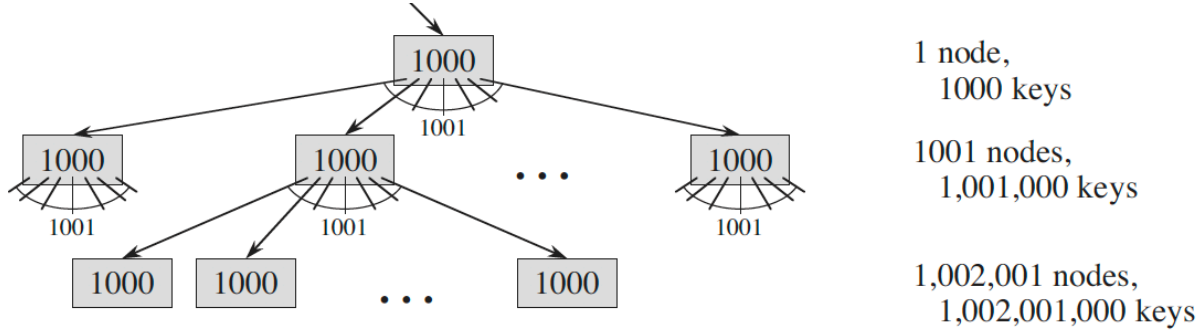
1 - http://pages.cs.wisc.edu/~remzi/OSTEP/file-ssd.pdf

Branching

For large B-trees stored on disk, branching factors are often between 50-2000, depending on the size of the key relative to the size of the page.

A large branching factor dramatically reduces the height of the tree, and the number of disk accesses required to find any key.

Example: B-tree with a branching factor of 1000 and height 2 can store over a billion keys. As we keep the root in main memory, we can find any key with at most two disk accesses.



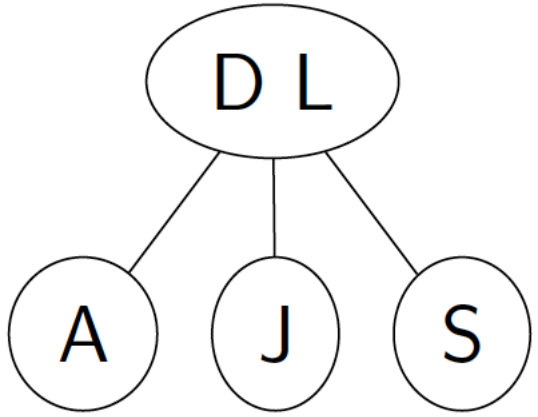
Today’s outline

1. Overview
2. Disk storage
3. B-trees
4. Searching
5. Inserting
6. Deletion
7. Cases

B-Trees

A B-tree is an extension of a BST. Instead of up to 2 children, a B-tree can have up to *m* children for some pre-specified integer m (called the order of the B-tree).

m is typically chosen so that a B-tree node will take up one page on the drive.



Definition

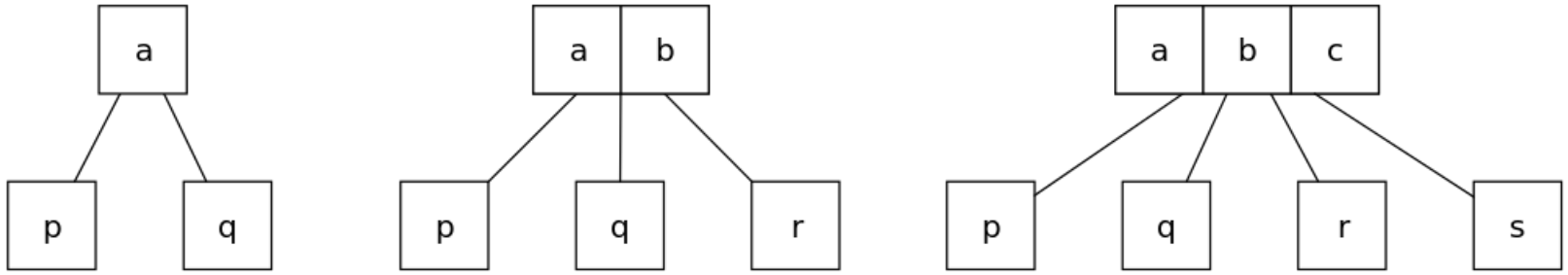
A B-tree of **minimum degree *t*** ≥ 2 has the following properties:

1. Every node *x* has the following attributes:
   a) *x.n* = the number of keys currently stored in node *x*
   b) Keys are stored in increasing order: $x.key_1 \leq x.key_2 \leq \ldots \leq x.key_n$
   c) *x.leaf*, a Boolean, is TRUE if *x* is a leaf, FALSE if *x* is an internal node.
2. All leaves have the same depth, which is the tree's height *h*
3. Nodes have upper and lower bounds on the number of keys.
   a) Every node other than the root has x.n ≥ $t - 1$ keys. Every internal node other than the root has ≥ *t* children.
   b) Every node may contain x.n ≤ $2t - 1$ keys. An internal node may have ≤ 2t children ($m = 2t$). A node is full if it has x.n = $2t - 1$ keys.
4. The root has at least two children if it is not a leaf node.

Simple B-tree

The simplest B-tree occurs when $t = 2$ (i.e., $m = 4$). Every internal node has either 2, 3, or 4 children. This is known as a **2-3-4 tree**[1]. That is, a B-tree of order 4. This structure is commonly used to implement associative arrays.

In practice, much larger values of t yield B-trees with smaller height.



2 node 3 node 4 node

Today’s outline

1. Overview
2. Disk storage
3. B-trees
4. Searching
5. Inserting
6. Deletion
7. Cases

Searching

Searching a B-tree is much like searching a BST, except that instead of making a binary, or “two-way”, branching decision at each node, we make a multiway branching decision according to the node’s children.

At each internal node x, we can make ($x.n$ + 1) branching decisions.

Search takes a pointer to the root node of a subtree x, and a key k.

If k is in the tree, B-Tree-Search returns the ordered pair (y, i) consisting of node y and an index i, such that $y.key_i = k$. Otherwise, return NIL.

Searching


```
function B-Tree-Search(Node x, Key k)
1:   i = 0
2:   while i ≤ x.n and k > x.k_i  // Scan keys in current node
3:       i = i + 1
4:   if i ≤ x.n and k == x.key_i then
5:       return (x, i)              // Found key
6:   elseif x.leaf then
7:       return NIL                 // Scanned node keys, no children to scan
8:   else
9:       return B-Tree-Search(x.c_i, k) // We have children to scan
end function
```

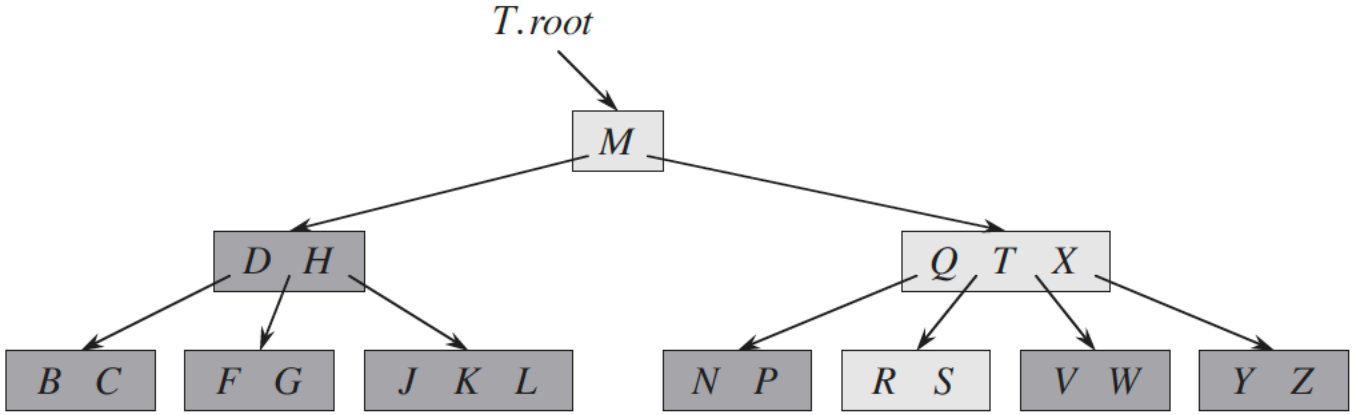
Searching

Lines 1-3: find smallest index i such that $k \leq x.key_i$, or else set $i = x.n + 1$.

Lines 4-5: Check if we have found the key, returning if we have.

Lines 6-7: Or terminate unsuccessfully (if *x* is a leaf)

Lines 8-9: Or recurse to search the appropriate subtree of *x*.



Lightly-shaded nodes are examined in a search for letter 'R'.

Today’s outline

1. Overview
2. Disk storage
3. B-trees
4. Searching
5. Inserting
6. Deletion
7. Cases

Insertion

Inserting a key into a B-tree is more complicated than a BST. As with a BST, we search for the leaf position at which to insert a new node.

With a B-tree however, we cannot create a new left node and insert it, as the result would not be a valid B-tree. The leaf would not be at the same height, and x.n !≥ $t - 1$ keys (when t > 2).

Instead we insert the new key into an existing leaf node. Doing so requires a new operation: **split**.

Splitting

Since we cannot insert a key into a full leaf node, we **split** the full node *y* (having *2t-1* keys) around its **median key** $y.key_i$ into two nodes having only *t-1* keys each.

The median key moves up into *y*'s parent to identify the dividing point between the two new trees.

But if y's parent is also full, we must split it before we can insert the new key, and could potentially do so, all the way up the tree.

To achieve a single pass down the tree insertion, we therefore always split each full node we come to along the way (including the leaf itself).

That way, when we split a full node *y*, its parent is not full.

Inserting


```
function B-Tree-Insert(Node x, Key k)
1:   i = x.n
2:   if x.leaf
3:       while i ≥ 1 and k < x.key_i    // Move left over leaf node keys
4:           x.key_i+1 = k              // Shift keys over by +1 for new key
5:           i = i - 1
6:       x.n = x.n + 1                  // Insert new key
7:   else while i ≥ 1 and k < x.key_i   // Not a leaf, find insert/split point
8:           i = i – 1
9:       i = i + 1                      // Insert/split point
10:      if x.c_i.n == 2t-1 then        // Full child?
11:          B-Tree-Split-Child(x, i) // Yes, split the node
12:          if k > x.key_i then
13:              i = i + 1              // Move over index if key larger
14:      B-Tree-Insert(x.c_i, k)
end function
```

Splitting the root

The special case for splitting the root is omitted for brevity, as is the procedure for splitting a node. These can be found in Section 18.2.

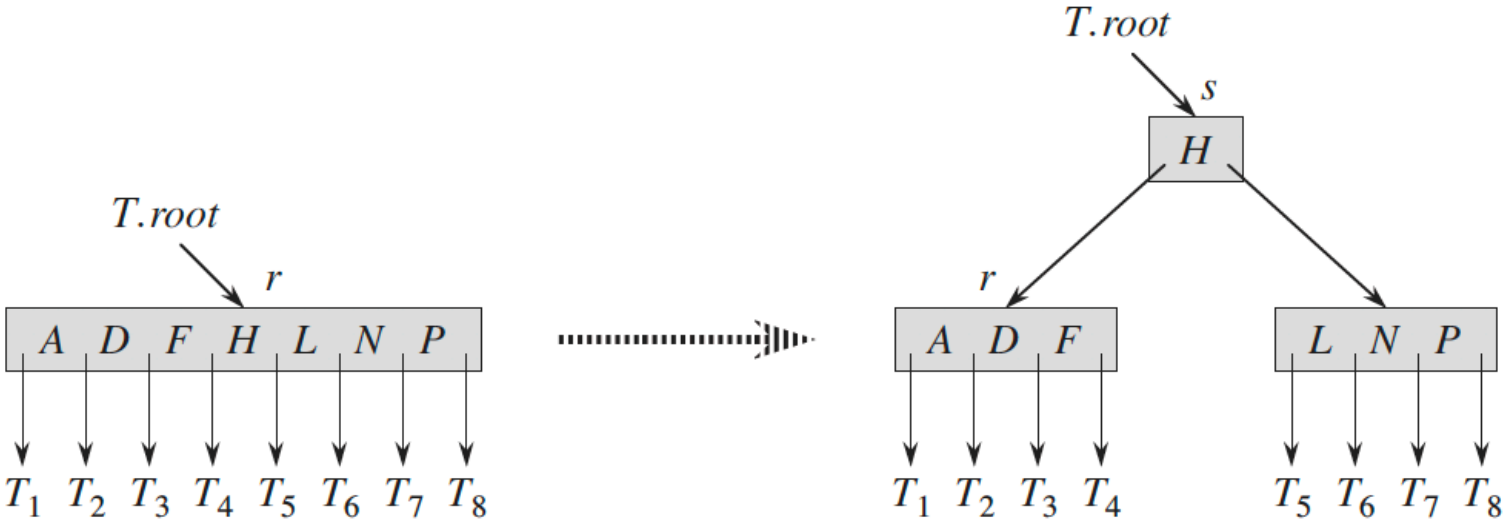
Splitting the root is the only way to increase the height of a B-tree.

Unlike a BST, a B-tree increases its height at the top instead of at the bottom.

Example: Splitting the root

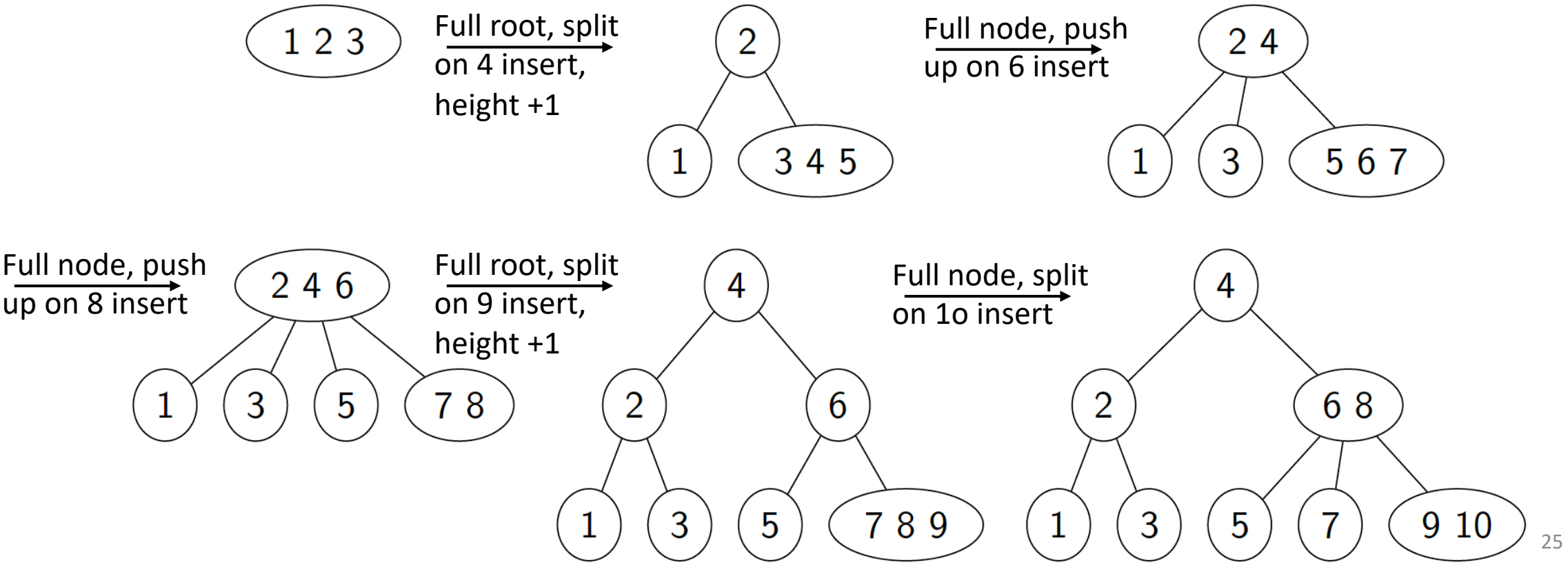
Splitting the root with $t = 4$. The root node r splits in two, and a new root node s is created.

The new root contains the median key of r and has the two halves of r as children. The B-tree grows in height by one when the root is split.

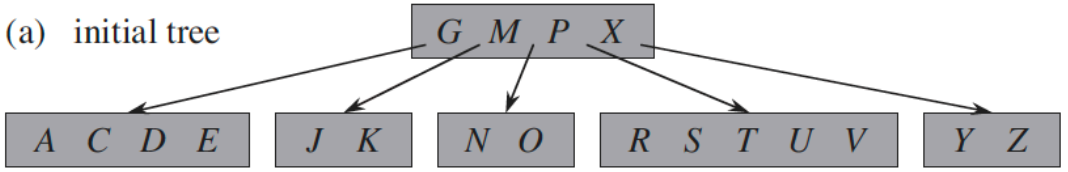


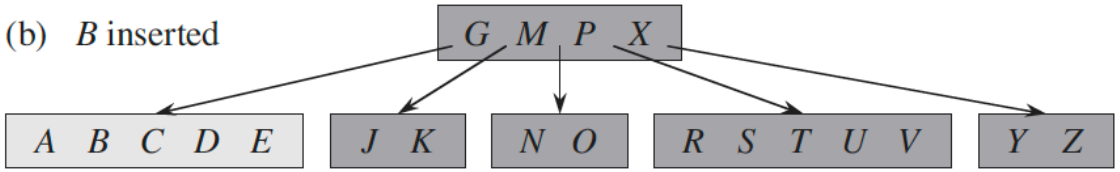
Example: Insert 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

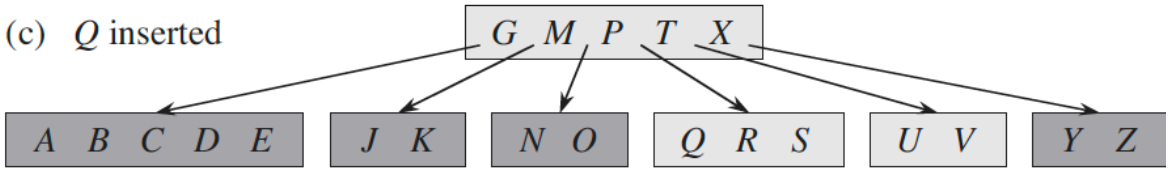
For t = 2, the maximum number of keys is 3 ($2t$-1) and max children m = 4.

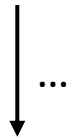


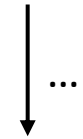
Example: Insert B, Q, L, F

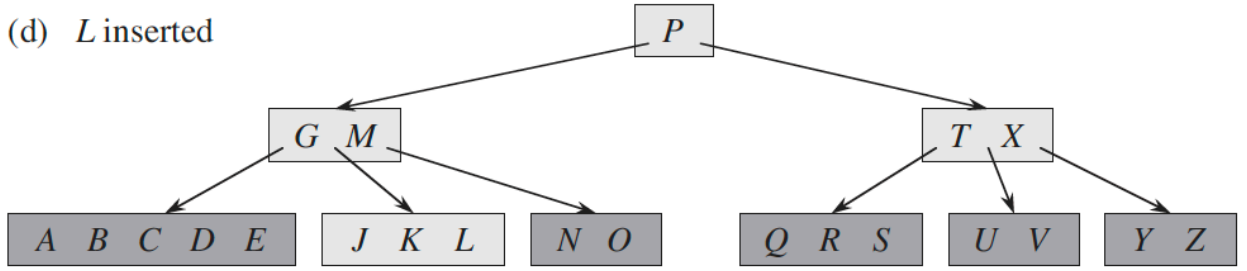


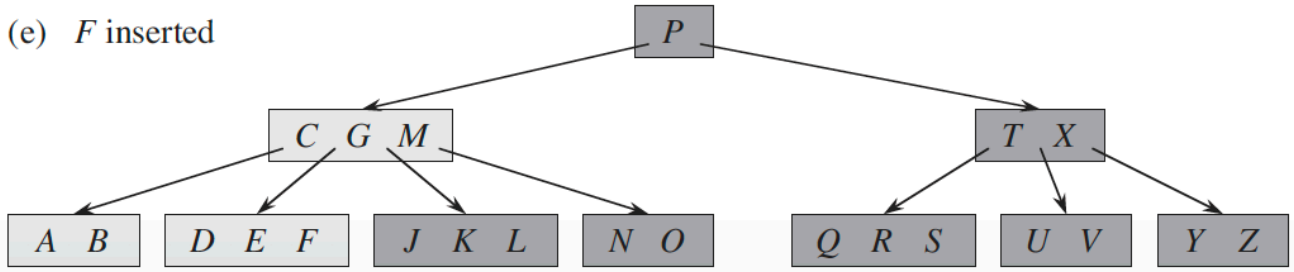












Today’s outline

1. Overview
2. Disk storage
3. B-trees
4. Searching
5. Inserting
6. Deletion
7. Cases

Deletion

Deletion from a B-tree is a bit more complicated than insertion because a key may be deleted from any node, not just a leaf.

Deletion from an internal node requires that the node's children be rearranged.

Just as we had to ensure that nodes didn't get too big due to insertion, with deletion we have to ensure that nodes don't get too small.

Recall: Every node other than the root has $x.n \geq t - 1$ keys. Note, the root can have fewer than t - 1 keys.

Deletion

The procedure `B-Tree-Delete` deletes a key *k* from subtree rooted at *x*.

This procedure is designed to guarantee that whenever it calls itself recursively on a node x, the number of keys in *x* (that is, *x.n*) is at least the minimum degree *t*.

This is done by ensuring that, before a key is deleted from a node, that node has at least *t* keys. That is, the node has one more key than the minimum required by the usual B-tree conditions.

Sometimes a key will have to be moved into a child node before recursion descends to that child.

Deletion

This **strengthened** condition allows us to delete a key in one downward pass without having to “back up”.

There are two ways of moving in an extra key:

- We may borrow a key from a nearby node that has more than it needs
- If we can't borrow, then we may merge two nodes that have no keys to spare.

To delete key *k*, we search from the root for the node containing *k*, strengthening each node we visit on the way if it has fewer than *t* keys.

Cases

There are 3 cases for deleting from a B-tree. We reach these cases via recursion.

As we recurse down the tree, we are checking which of the conditions we are in and recursively calling delete as necessary.

In the following cases, assume we have reached node *x*:

Cases

1. *x* is a leaf and contains the key (it will have at least *t* keys). This case is trivial – just delete the key.
2. *x* is an internal node and contains the key. There are 3 subcases:
   2a. predecessor child node has at least *t* keys
   2b. successor child node has at least *t* keys
   2c. neither predecessor nor successor child has *t* keys
3. *x* is an internal node, but doesn't contain the key. Find the child subtree of *x* that contains the key if it exists (call the child *c*). There are three subcases:
   3a. *c* has at least t keys. Simply recurse to c.
   3b. *c* has *t* - 1 keys and one of its siblings has *t* keys.
   3c. *c* and both siblings have *t* - 1 keys.

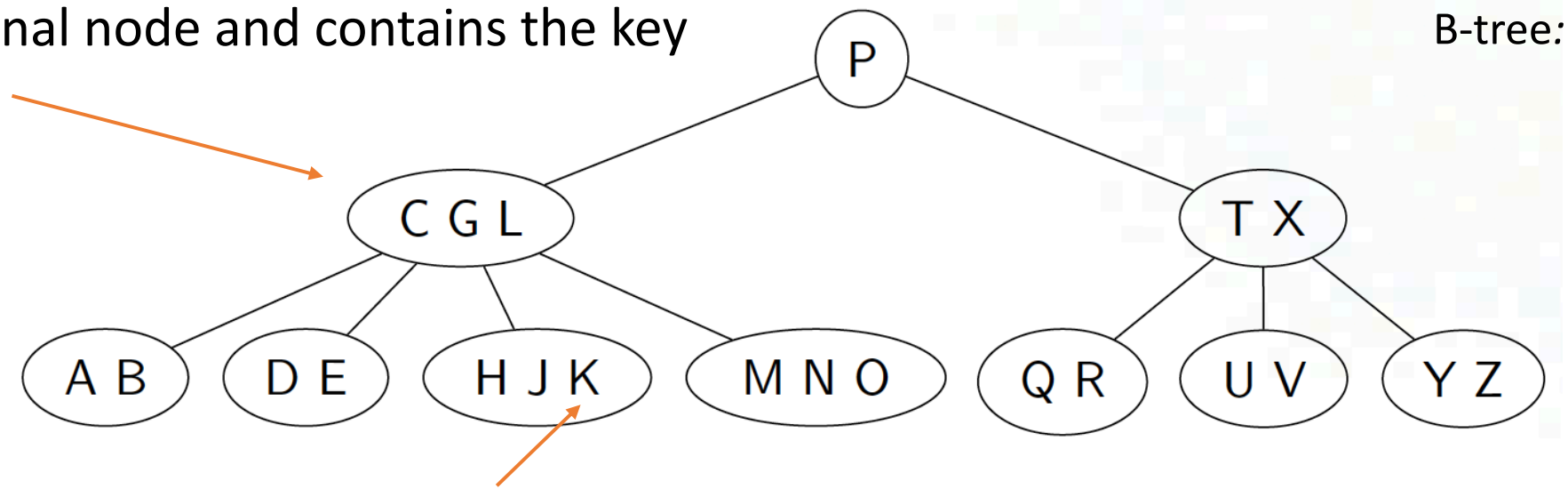
Today’s outline

1. Overview
2. Disk storage
3. B-trees
4. Searching
5. Inserting
6. Deletion
7. Cases

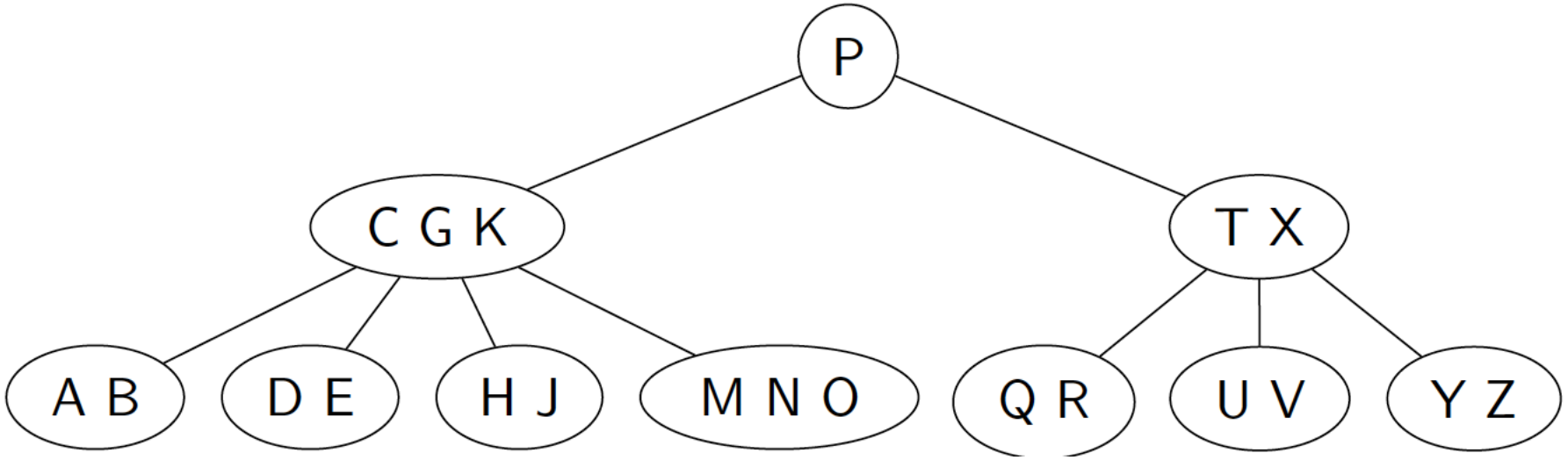
Case 2a – predecessor has ≥ *t* keys – delete L

x is an internal node and contains the key

B-tree: $t = 3 :: 2 \le x.n \le 5$



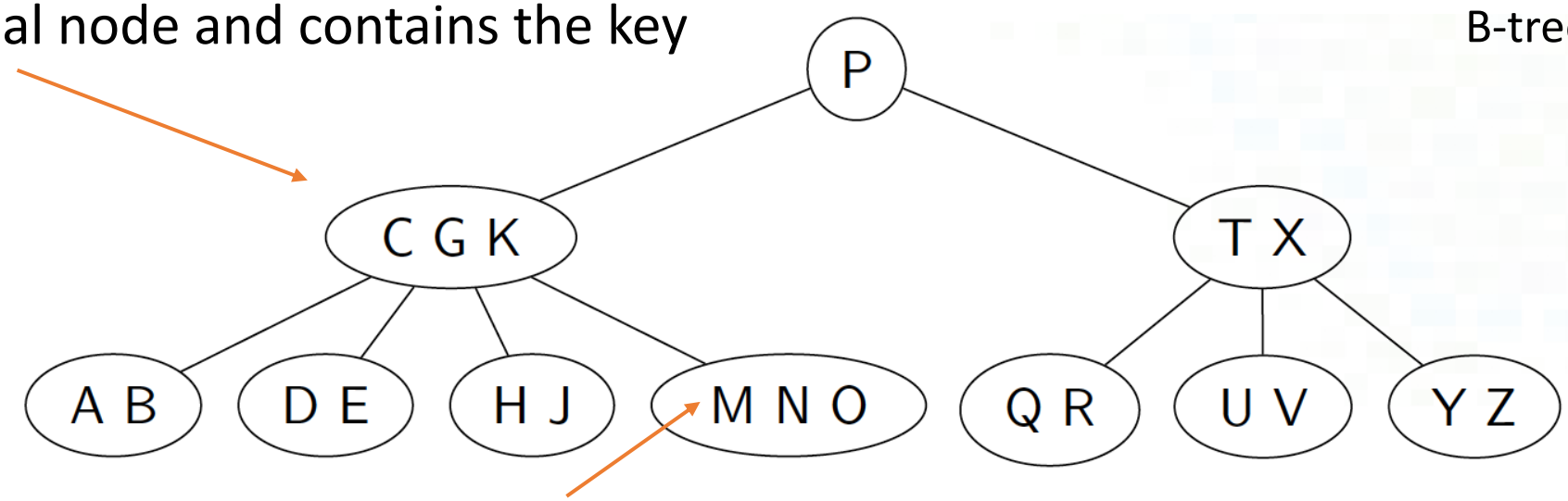
Replace *L* by its inorder predecessor, and recursively delete the predecessor:



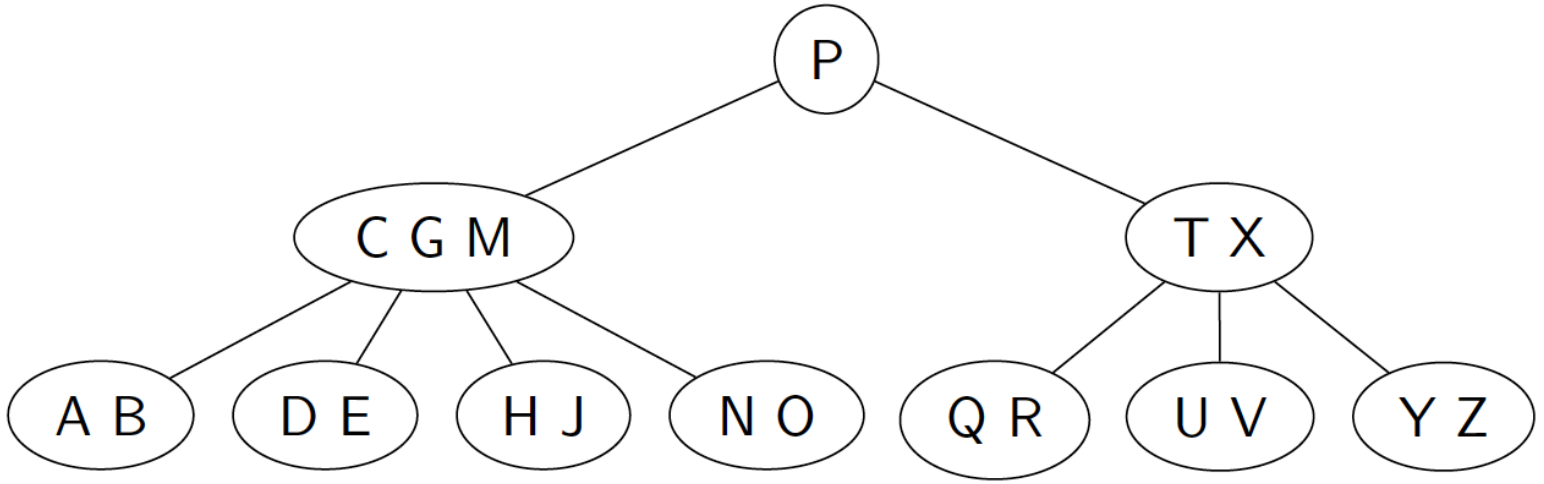
Case 2b – successor has ≥ *t* keys – delete K

x is an internal node and contains the key

B-tree*: t* = 3 :: 2 ≤ *x.n* ≤ 5



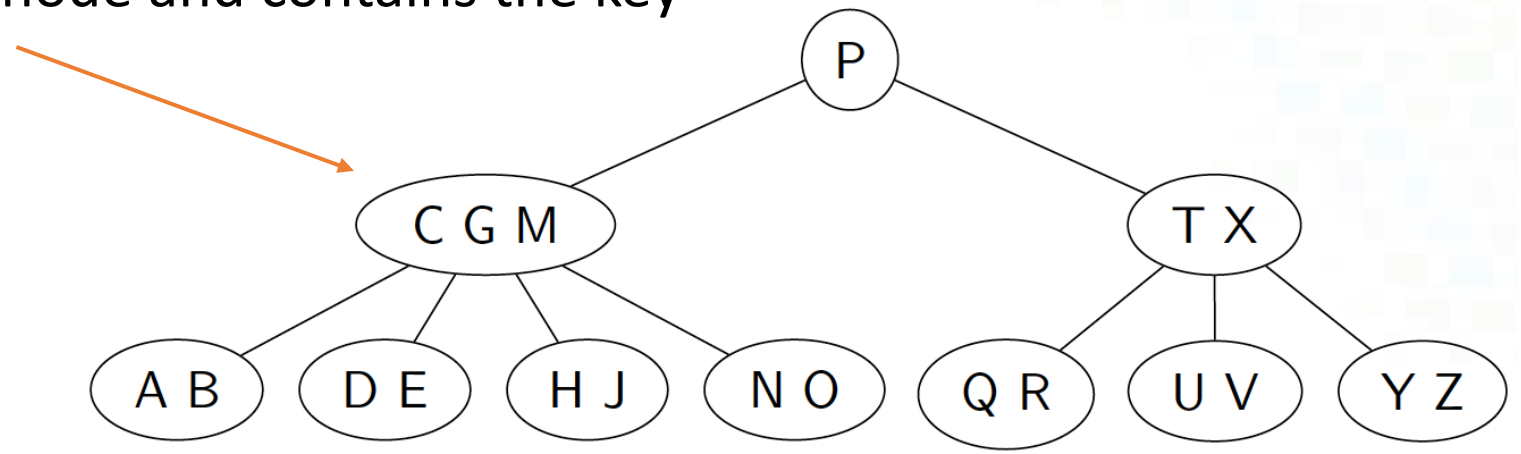
Replace *K* by its successor, and recursively delete the successor:



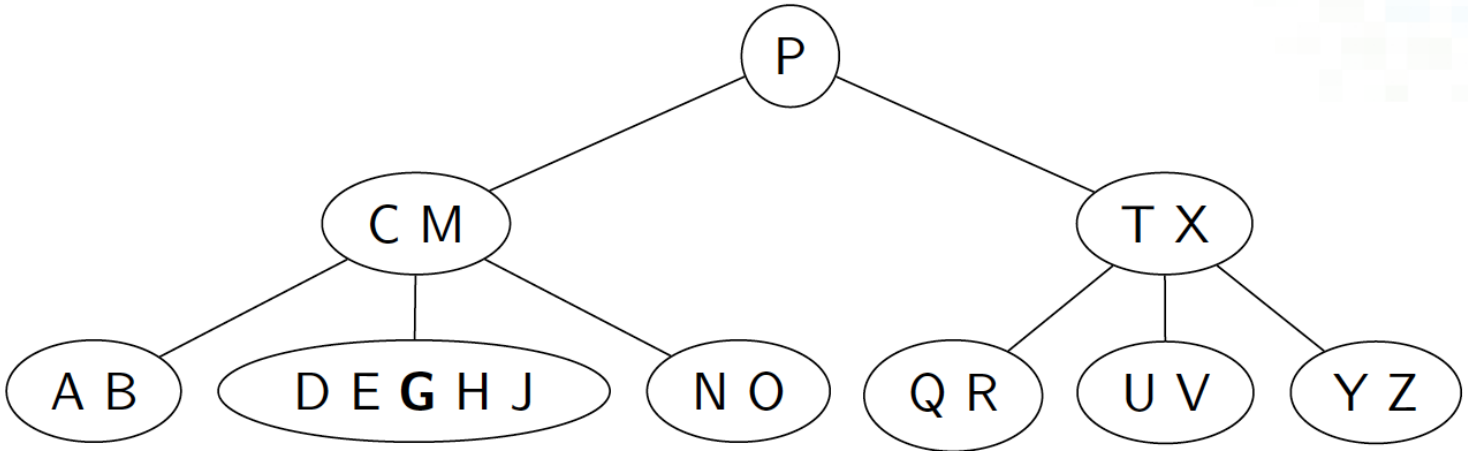
Case 2c – neither predecessor nor successor has ≥ *t* keys – delete G

x is an internal node and contains the key

B-tree*: t* = 3 :: 2 ≤ *x.n* ≤ 5



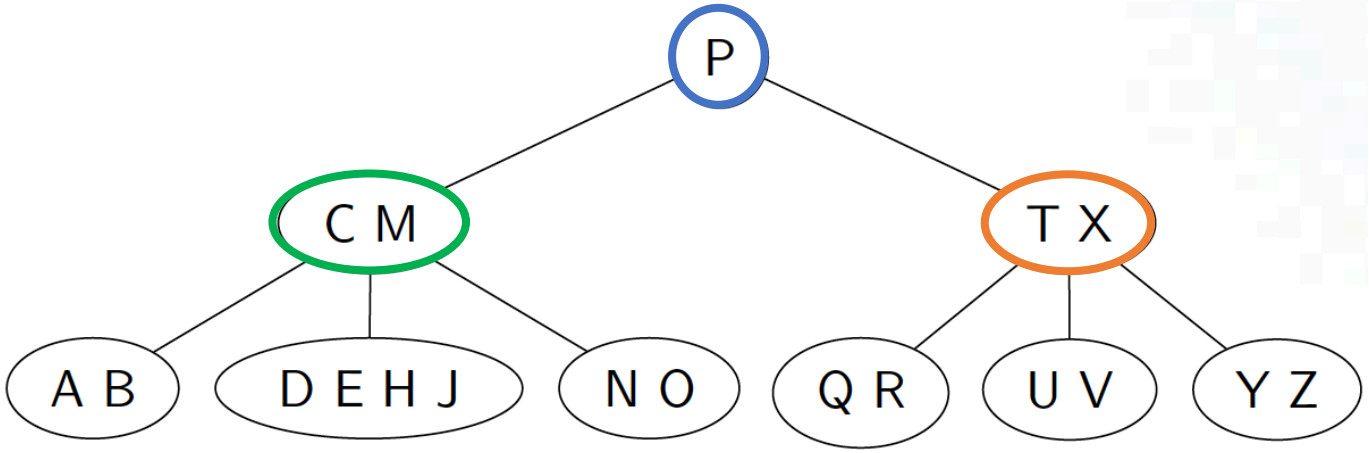
Merge the two children and push ***G*** down into the new child. Then recursively delete ***G*** (which could involve any of the three cases):



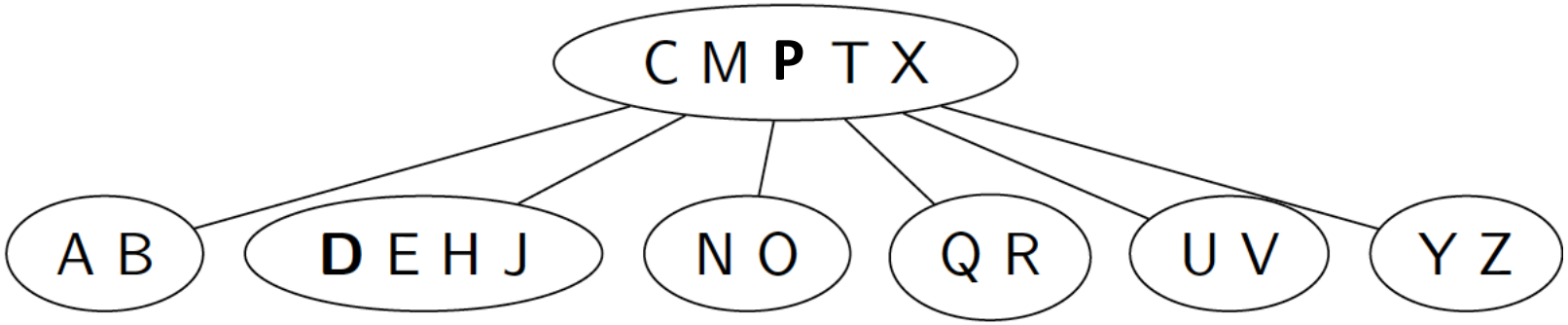
Case 3c - *c* and both siblings have *t* - 1 keys – Delete D

B-tree: $t = 3 :: 2 \leq x.n \leq 5$

x is the node (P), x.*c* is the node (C M):



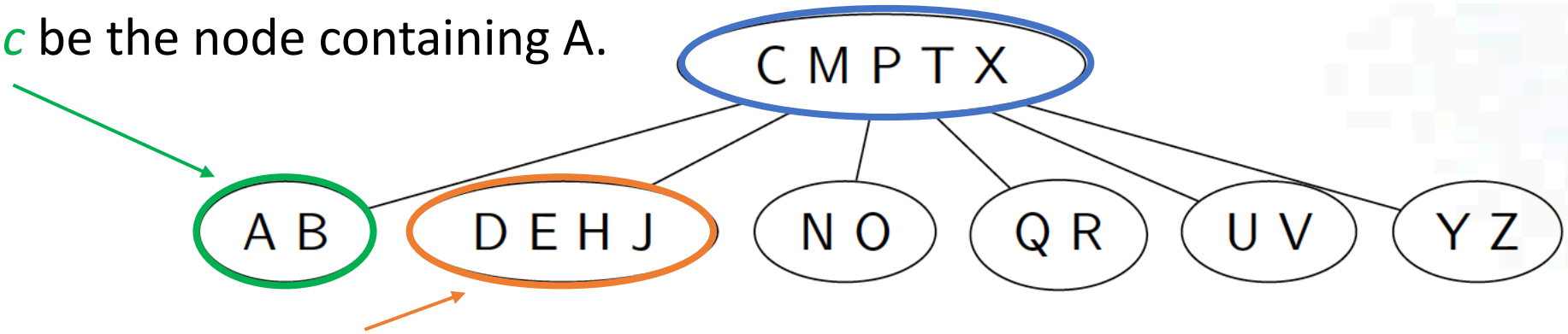
We cannot descend into (C M) as it only has 2 keys. *x.c* has only one sibling (T X) and it has *t* - 1 keys. Therefore we have to merge with one of the siblings. We push **P** down, merge, then recursively delete **D**:



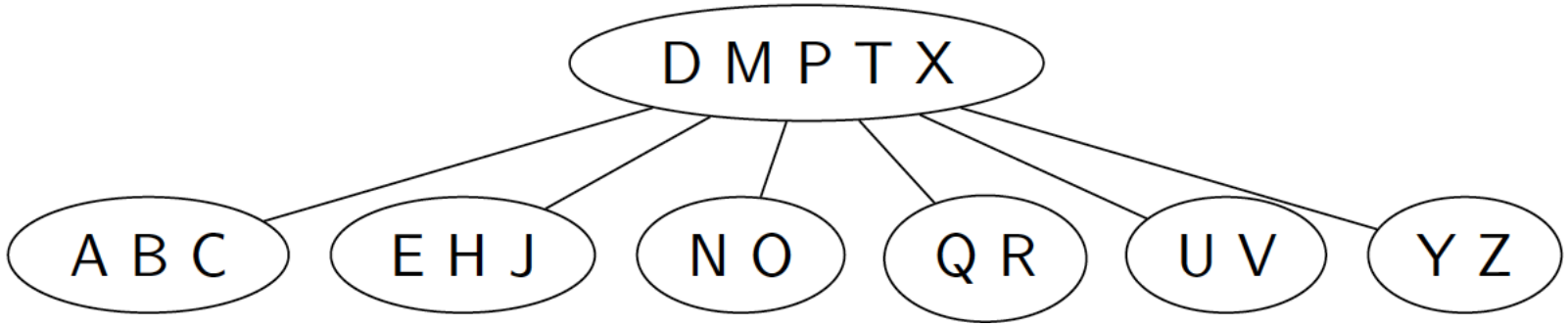
Case 3b - *c* has *t* - 1 keys and one of its siblings has *t* keys – Delete A

B-tree: $t = 3 :: 2 \leq x.n \leq 5$

x is the node (CMPTX).
Let x.*c* be the node containing A.



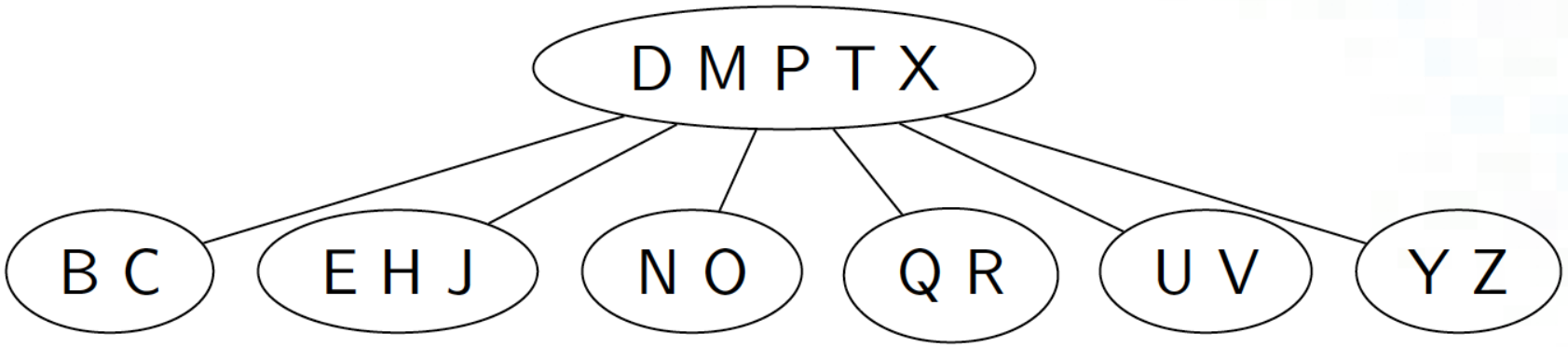
Since x.*c*'s sibling has ≥ *t* keys, we can borrow a key from its sibling. We first move a key down from *x* (key C), then push a key up from our sibling into *x*:



Now we recursively delete A.

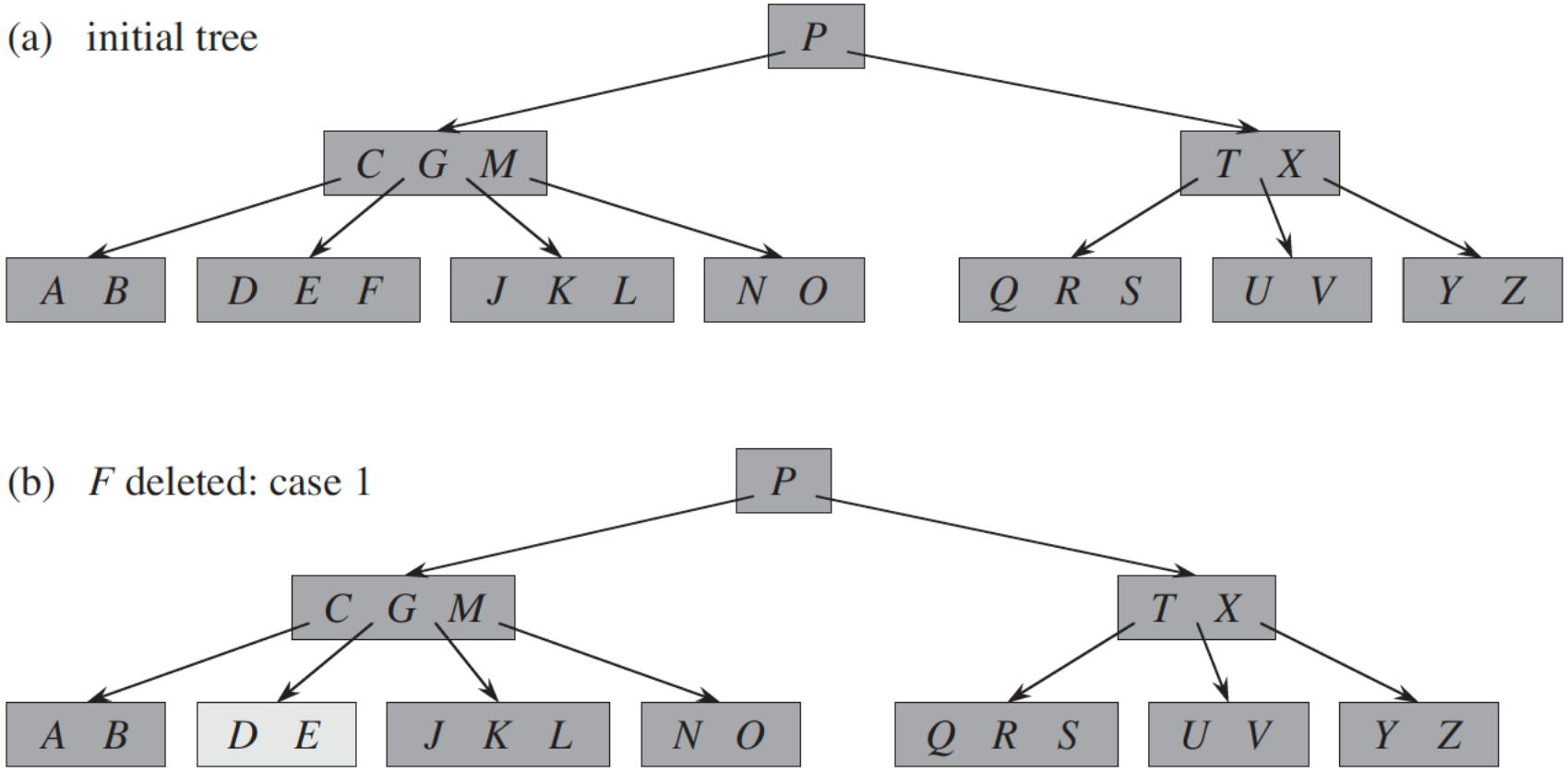
Class challenge

Given the B-tree of degree t = 3, delete z. State relevant cases:



One more example

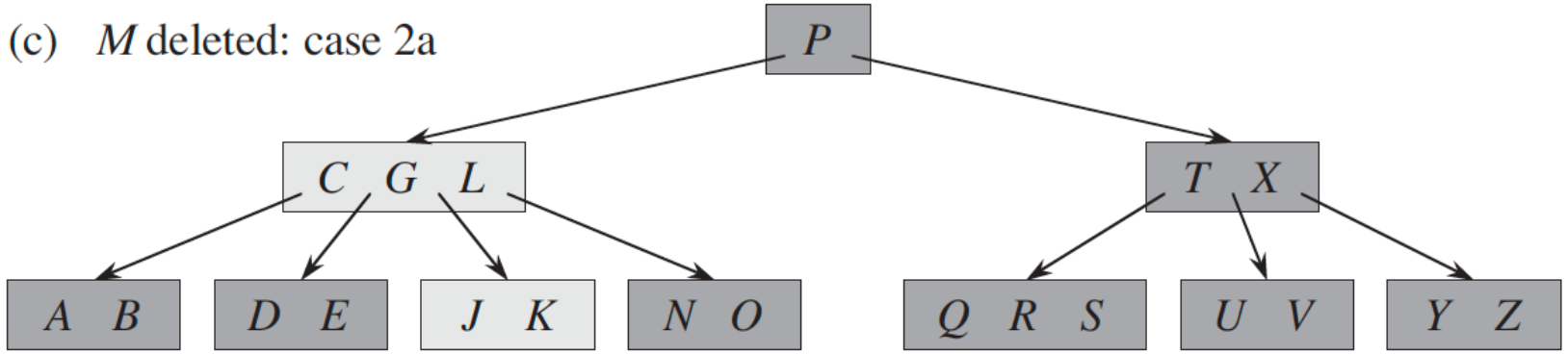
The minimum degree for this B-tree is t = 3. So a node (other than the root) cannot have fewer than 2 keys.



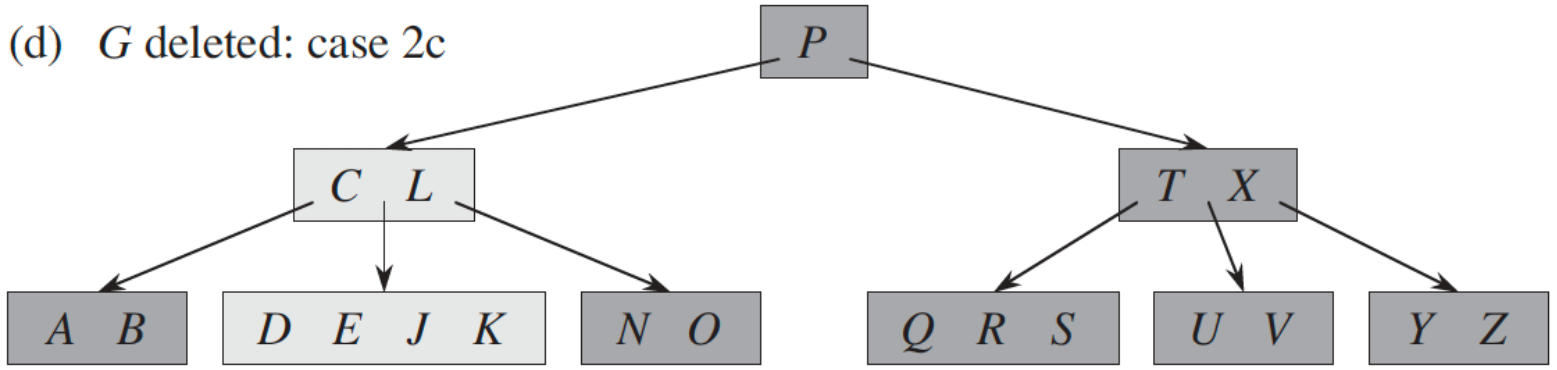
Case 1: Simple deletion from a leaf that contains ≥ t keys

One more example

The minimum degree for this B-tree is t = 3. So a node (other than the root) cannot have fewer than 2 keys.



Case 2a: the predecessor L of M moves up to take M's position.

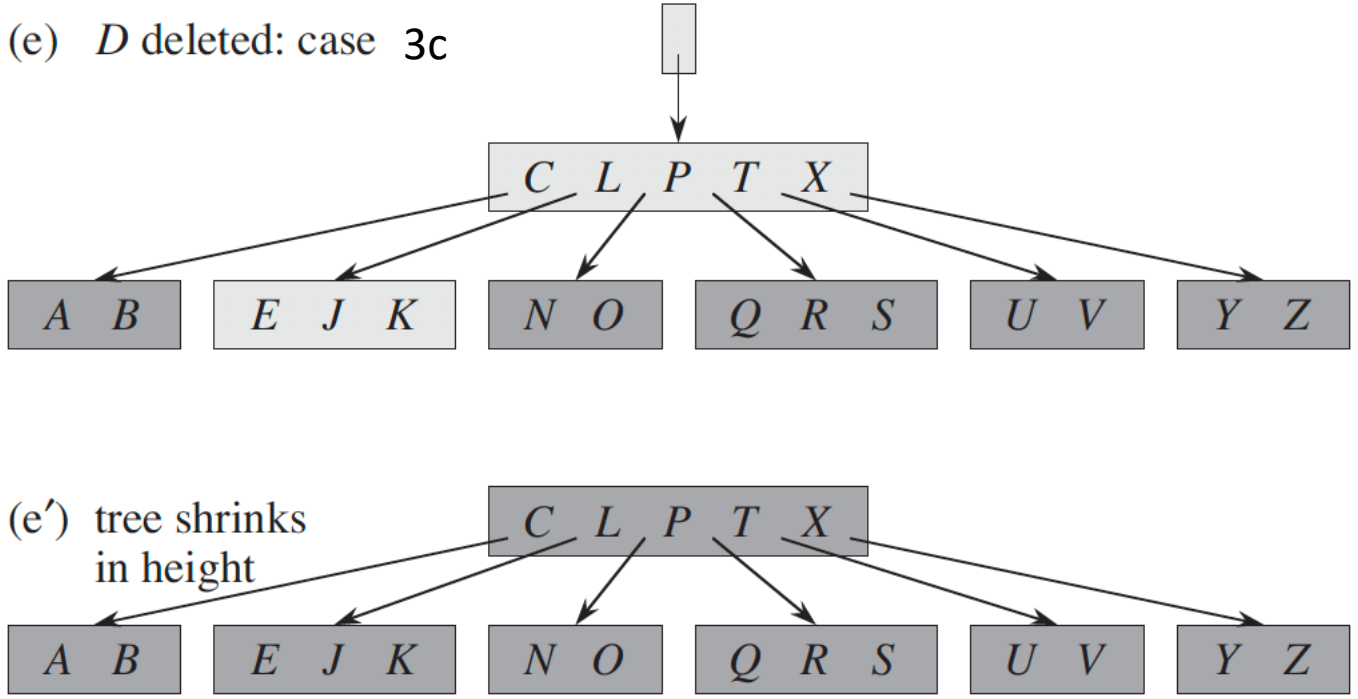


Case 2c: Push G down to make node D E G J K, then delete G from leaf (case 1)

One more example

The minimum degree for this B-tree is t = 3. So a node (other than the root) cannot have fewer than 2 keys.

Case 3c: the recursion cannot descend to node (C L) as it has *t* - 1 keys. Push P down, merge with CL and TX to form CLPTX. Then delete D from a leaf (case 1).

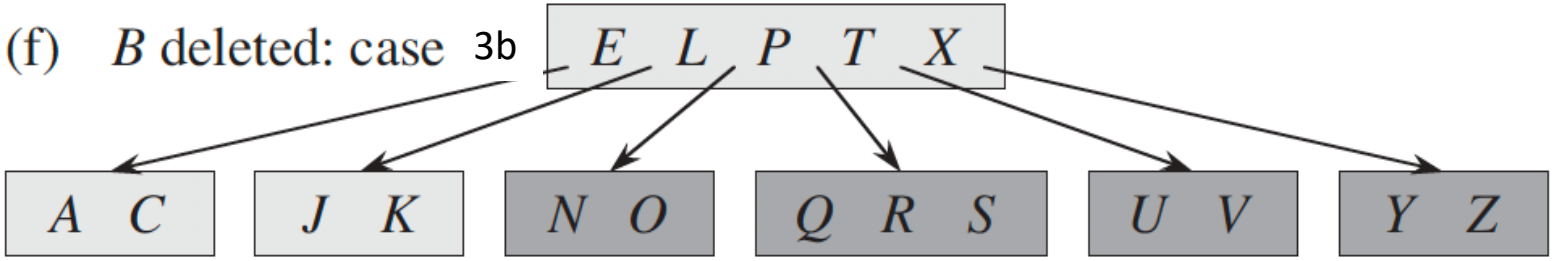


Continued: After (e), we delete the root and the tree shrinks in height by one.

One more example

The minimum degree for this B-tree is t = 3. So a node (other than the root) cannot have fewer than 2 keys.

Case 3b: C moves to fill B's position and E moves to fill C's position.



Suggested reading

B-trees are discussed in Section 18.

Solutions

Image attributions

This Photo by Joshldt is licensed under CC BY-SA

This Photo by Joshldt is licensed under CC BY-SA

This Photo by Joshldt is licensed under CC BY-SA

Disclaimer: Images and attribution text provided by PowerPoint search. The author has no connection with, nor endorses, the attributed parties and/or websites listed above.