CME 2001
Data Structures and Algorithms

Zerrin Işık
zerrin@cs.deu.edu.tr

B-Trees

B-Trees

- B-trees are balanced search trees and designed to work well on disks or other direct access secondary storage devices.
- Many database systems use B-trees, or variations to store information.
- The idea of the B-tree also motivates the design of many other disk-based index structures.

B-Trees

- When data volume is large and does not fit in memory, a B-tree- is used.
- The B-tree is always balanced (all leafs appear at the same level)
- Since each disk access exchanges a whole block of information between memory and disk, a node of the B-tree is expanded to hold more than two child pointers, up to the block capacity.
- The B-tree requires that every node (except the root) has to be at least half full.
- An exact match query, insertion, deletion need to access O(log*B n)* nodes, where
  - ***B***: *the page capacity* in number of child pointers
  - ***n***: *is the number of objects*

Disk –Based Environment

- The computer CPU deals directly with the primary storage (main memory).
- We can access data stored in main memory quickly, but cannot store everything in memory:
  - because memory is expensive.
  - memory is volatile, i.e. if there is a power failure, information stored in memory gets lost.
- The secondary storage stands for magnetic disks. Although it has slower access, it is less expensive and it is non-volatile.

Disk –Based Environment

- The CPU does not deal with disk directly, any data has to be read from disk to memory first.
- Data is stored on disk in units called ***blocks*** or ***pages***.

- If a disk page is 8KB (8192 bytes), while a node in the BST is 16 bytes (four integers: key, value, two child pointers)

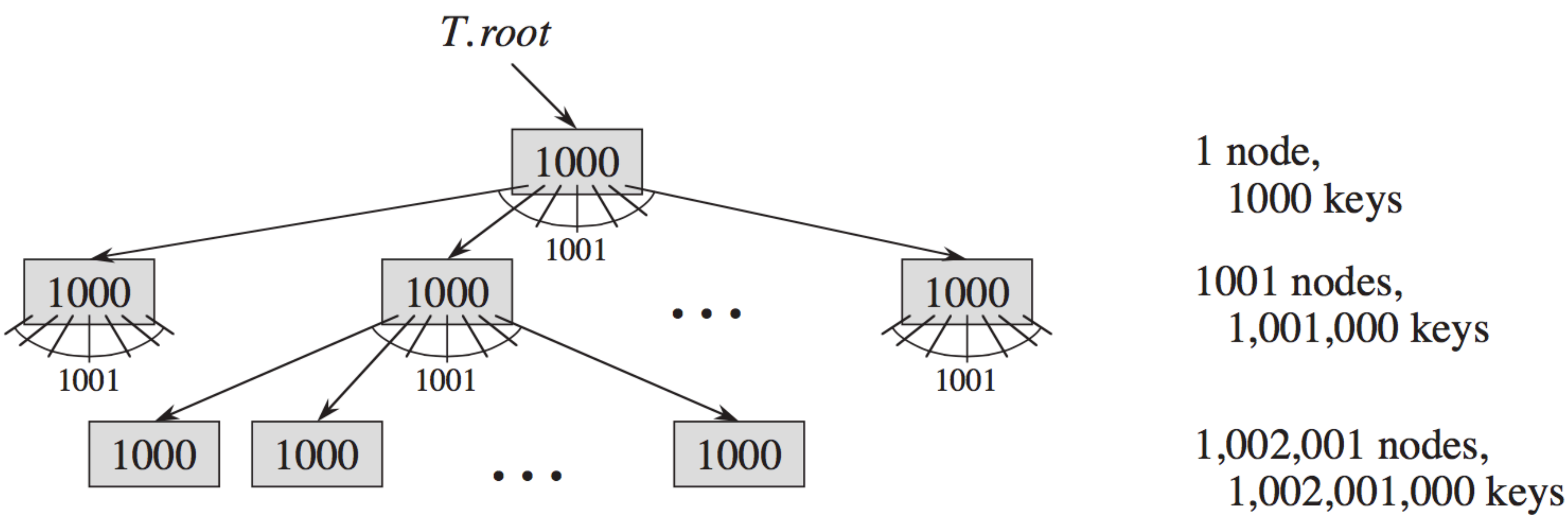
  => every page is only 8192/16 = 0.2% full.
- To improve space efficiency, we should store multiple tree nodes in one disk page.

Disk –Based Environment

- The running time of a B-tree operations highly depends on the number of *DISK-READ* and *DISK-WRITE* operations.
- ***Branching factor***: total # of children of a parent node
- For a large B-tree stored on a disk, ***branching factors*** are between 50 and 2000, depending on the size of a key relative to the size of a page.
- A large *branching factor* reduces :
  - the height of the tree
  - the number of disk accesses required to find any key

B-tree example

- B-tree :
  - branching factor = 1001
  - height = 2 (that can store over ***one billion keys***)
- can keep the root node permanently in main memory,
- can find any key in this tree by making at most only **2** disk accesses.



Definition of B-trees

1. Every node $\boldsymbol{x}$ has the following attributes:
   a) $\boldsymbol{x.n}$: the number of keys currently kept in node $\boldsymbol{x}$
   b) the $\boldsymbol{x.n}$ keys themselves stored in non-decreasing order, so that $\boldsymbol{x.key_1 \leq x.key_2 \leq \ldots \leq x.key_{x.n}}$
   c) $\boldsymbol{x.leaf}$: boolean value, which is *TRUE* if $\boldsymbol{x}$ is a leaf and *FALSE* if $\boldsymbol{x}$ is an internal node.

2. Each internal node $\boldsymbol{x}$ contains $\boldsymbol{x.n+1}$ pointers $\boldsymbol{x.c_1, x.c_2, \ldots, x.c_{n+1}}$ to indicate its children. Leaf nodes have no children, no $\boldsymbol{c_i}$ attributes.

3. The keys $\boldsymbol{x.key_i}$ separate the ranges of keys stored in each subtree. If $\boldsymbol{k_i}$ is any key stored in the subtree with root $\boldsymbol{x.c_i}$ then:

$$\boldsymbol{k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \ldots \leq x.key_{x.n} \leq k_{x.n+1}}$$

Definition of B-trees

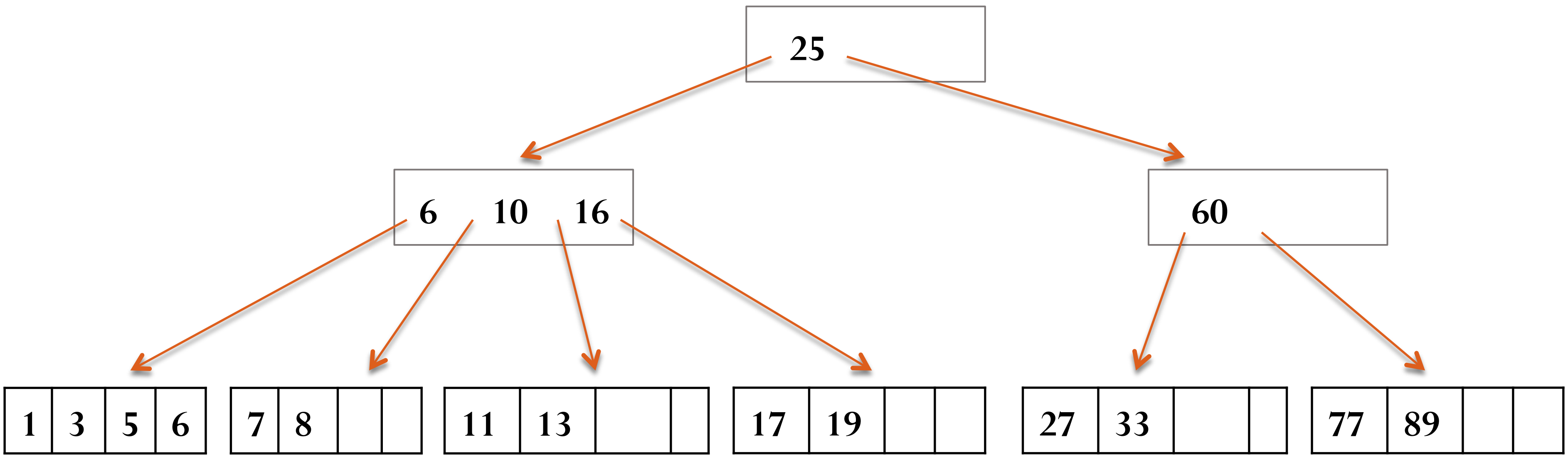
4. All leaves have the same depth, which is the tree's height ***h*.**

5. Nodes have lower and upper bounds on the number of keys they can contain. These bounds are represented by a fixed integer $t \geq 2$, called the ***minimum degree*** of the B-tree:

   a) Every node (except root) must have at least t-1 keys. Every internal node has at least t children. If tree in non-empty, root must have at least one key.

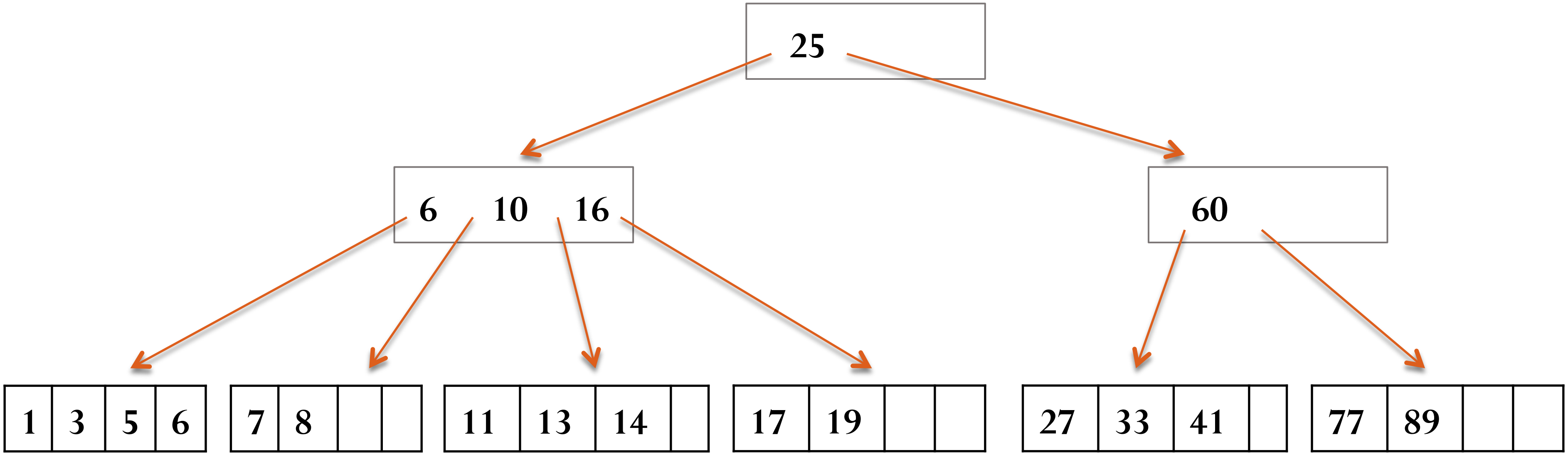
   b) Every node may contain at most 2t-1 keys. An internal node may have at most 2t children. A node becomes ***full***, if its contains exactly 2t-1 keys.

B-tree example



Insert : 14, 41

B-tree example



B-tree Operations: Search

```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

- B-TREE-SEARCH applies a similar method with BST-SEARCH.
- Starts with B-TREE-SEARCH (*T.root*, *k*)
- If k is found, it returns (x, i) node x and index i such that $x.key_i = k$
- Otherwise returns *NIL*.
- DISK-READ reads the requested object from secondary memory and puts into main memory.

Running time: O(t.h) = $O(t \lg_t n)$

- ***while*** loop runs ***t*** times

- recursion runs at most the depth of the tree = O(h)

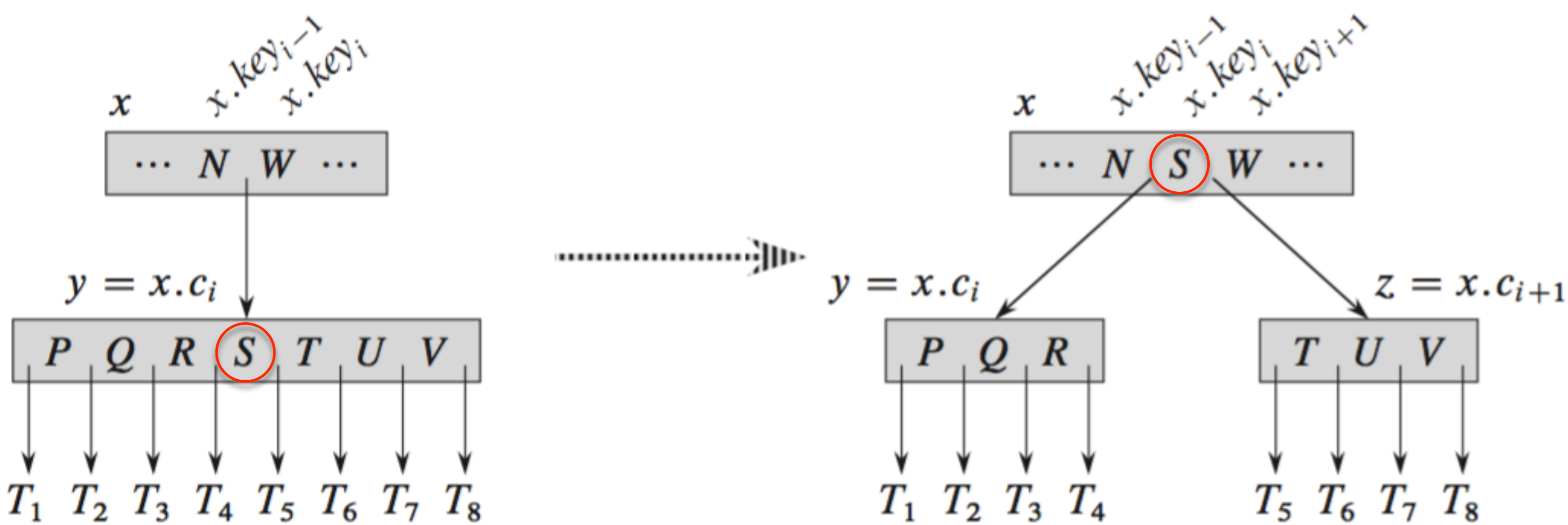
Create an empty B-tree

```
B-TREE-CREATE(T)
x = ALLOCATE-NODE()
x.leaf = TRUE
x.n = 0
DISK-WRITE(x)
T.root = x
```

- ALLOCATE-NODE allocates one disk page to be used as a new node in O(1) time.
- DISK-WRITE saves attributes of new node to the disk.
- Running time: O(1)

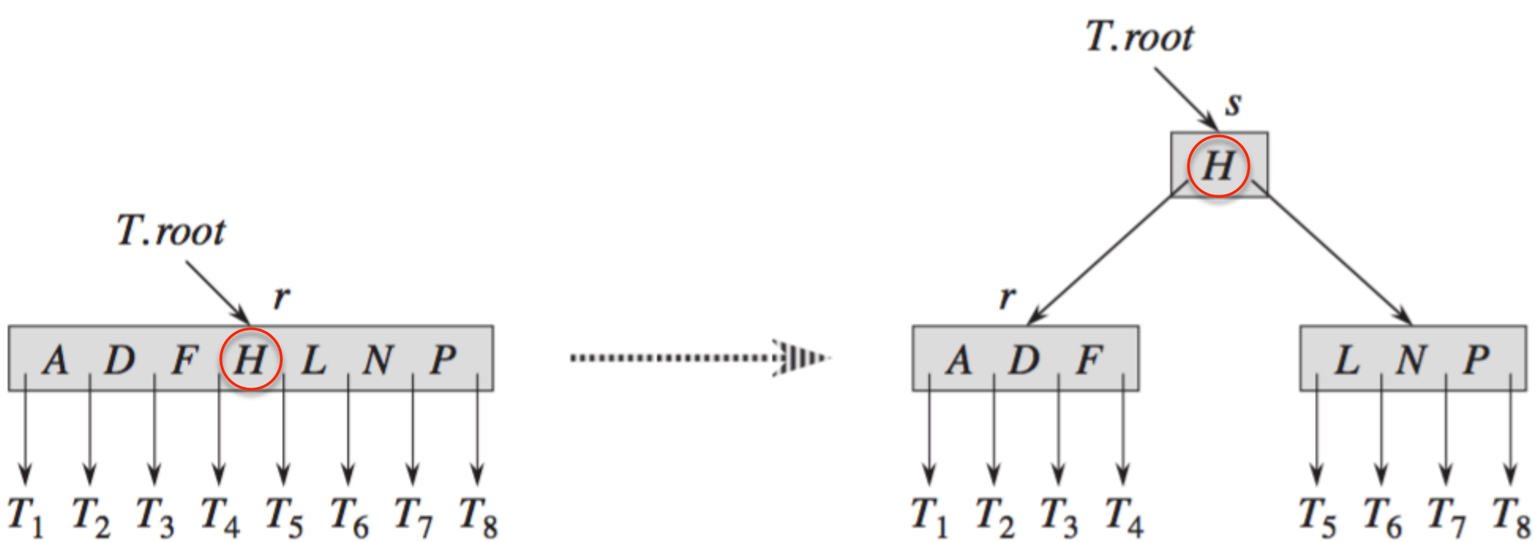
Insert a key into a B-tree

- To insert new key ***X***, find a suitable leaf node:
  - If leaf node is not full, put ***X*** into empty slot (easy case!)
  - If leaf node is full, split leaf node and adjust parents up to root.
- Insert "S" into below B-tree in which **t**=4



Insert a key into a B-tree …

- If the key should be put into a full root node, which should be divided into two new nodes and new root.
- Insert "H" into the root node of below B-tree in which **t**=4



Insert a key into a B-tree

```
B-TREE-INSERT(T, k)
1  r = T.root
2  if r.n == 2t − 1
3      s = ALLOCATE-NODE()
4      T.root = s
5      s.leaf = FALSE
6      s.n = 0
7      s.c_1 = r
8      B-TREE-SPLIT-CHILD(s, 1)
9      B-TREE-INSERT-NONFULL(s, k)
10 else B-TREE-INSERT-NONFULL(r, k)
```

Is Root node is full?

If "yes", then run lines 3-9

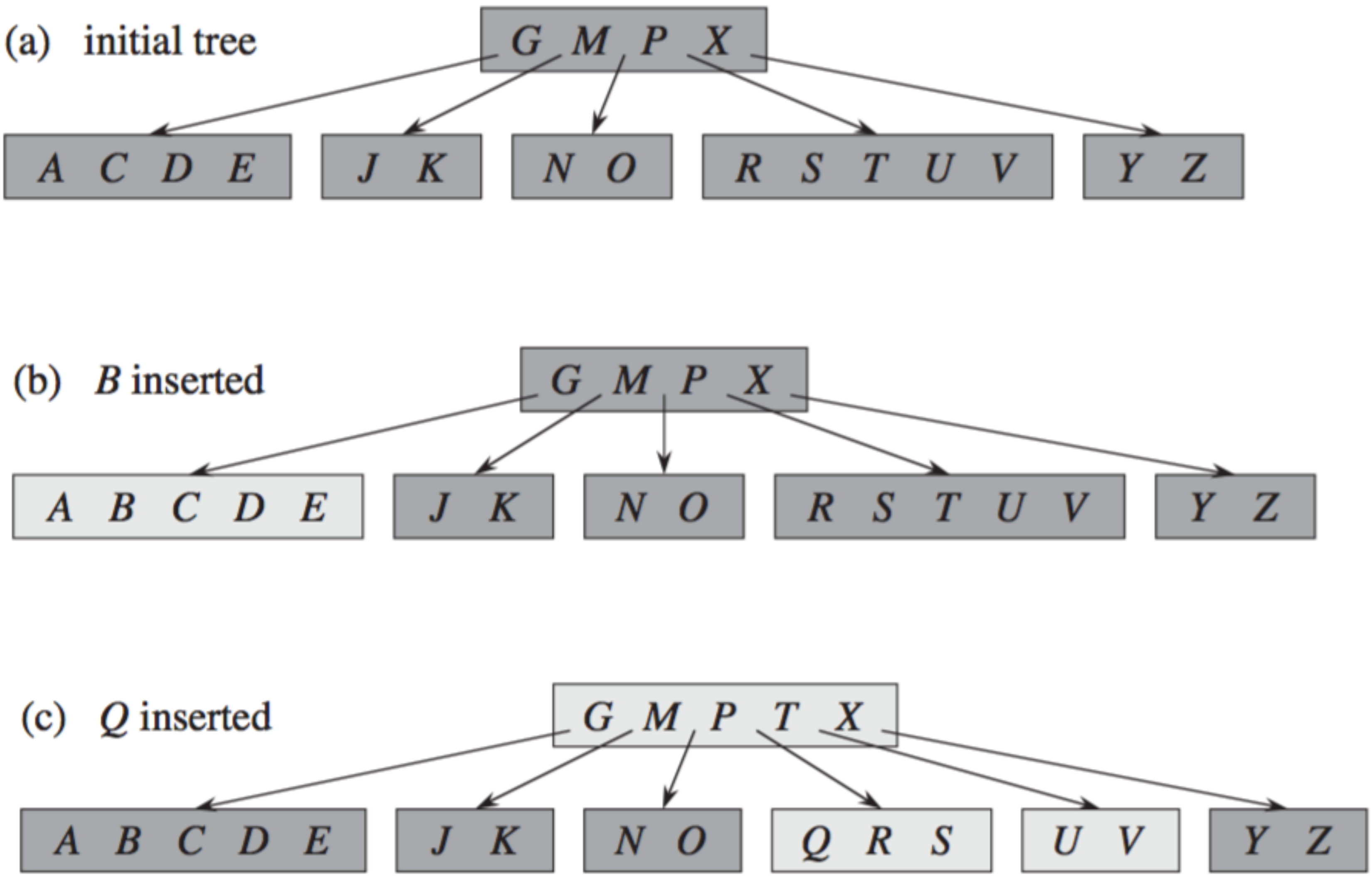
Running time: $O(t \lg_t n)$

```
B-TREE-SPLIT-CHILD(x, i)
  z = ALLOCATE-NODE()
  y = x.c_i
  z.leaf = y.leaf
  z.n = t - 1
  for j = 1 to t - 1
      z.key_j = y.key_{j+t}
  if not y.leaf
      for j = 1 to t
          z.c_j = y.c_{j+t}
  y.n = t - 1
  for j = x.n + 1 downto i + 1
      x.c_{j+1} = x.c_j
  x.c_{i+1} = z
  for j = x.n downto i
      x.key_{j+1} = x.key_j
  x.key_i = y.key_t
  x.n = x.n + 1
  DISK-WRITE(y)
  DISK-WRITE(z)
  DISK-WRITE(x)
```

```
B-TREE-INSERT-NONFULL(x, k)
  i = x.n
  if x.leaf
      while i ≥ 1 and k < x.key_i
          x.key_{i+1} = x.key_i
          i = i - 1
      x.key_{i+1} = k
      x.n = x.n + 1
      DISK-WRITE(x)
  else while i ≥ 1 and k < x.key_i
          i = i - 1
      i = i + 1
      DISK-READ(x.c_i)
      if x.c_i.n == 2t - 1
          B-TREE-SPLIT-CHILD(x, i)
          if k > x.key_i
              i = i + 1
      B-TREE-INSERT-NONFULL(x.c_i, k)
```

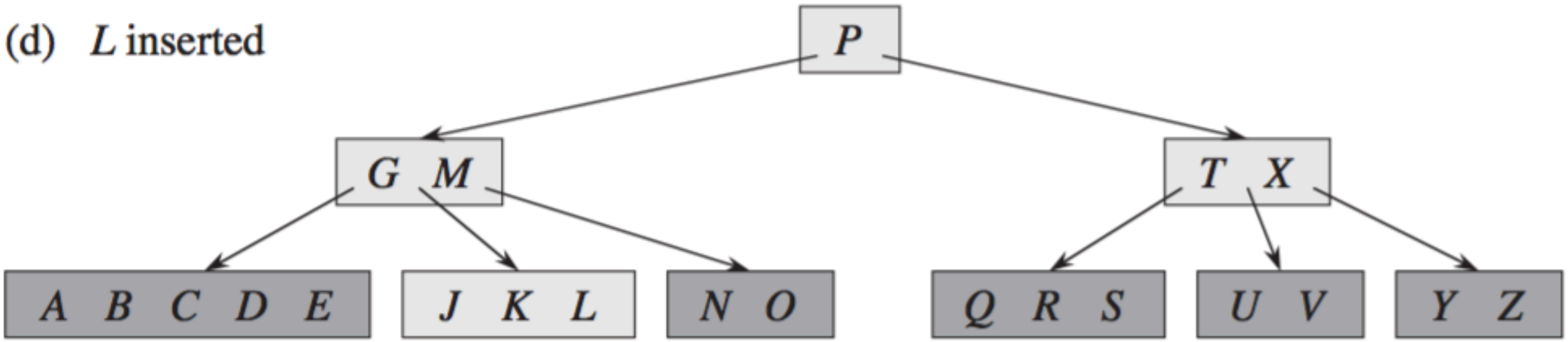
Insertion example

min. $t = 3$

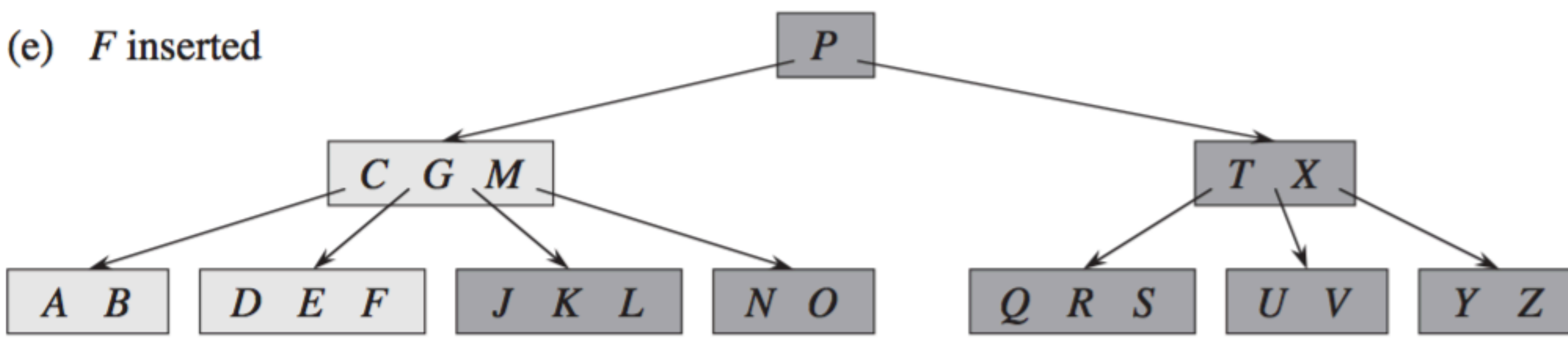


Insertion example

(d) *L* inserted



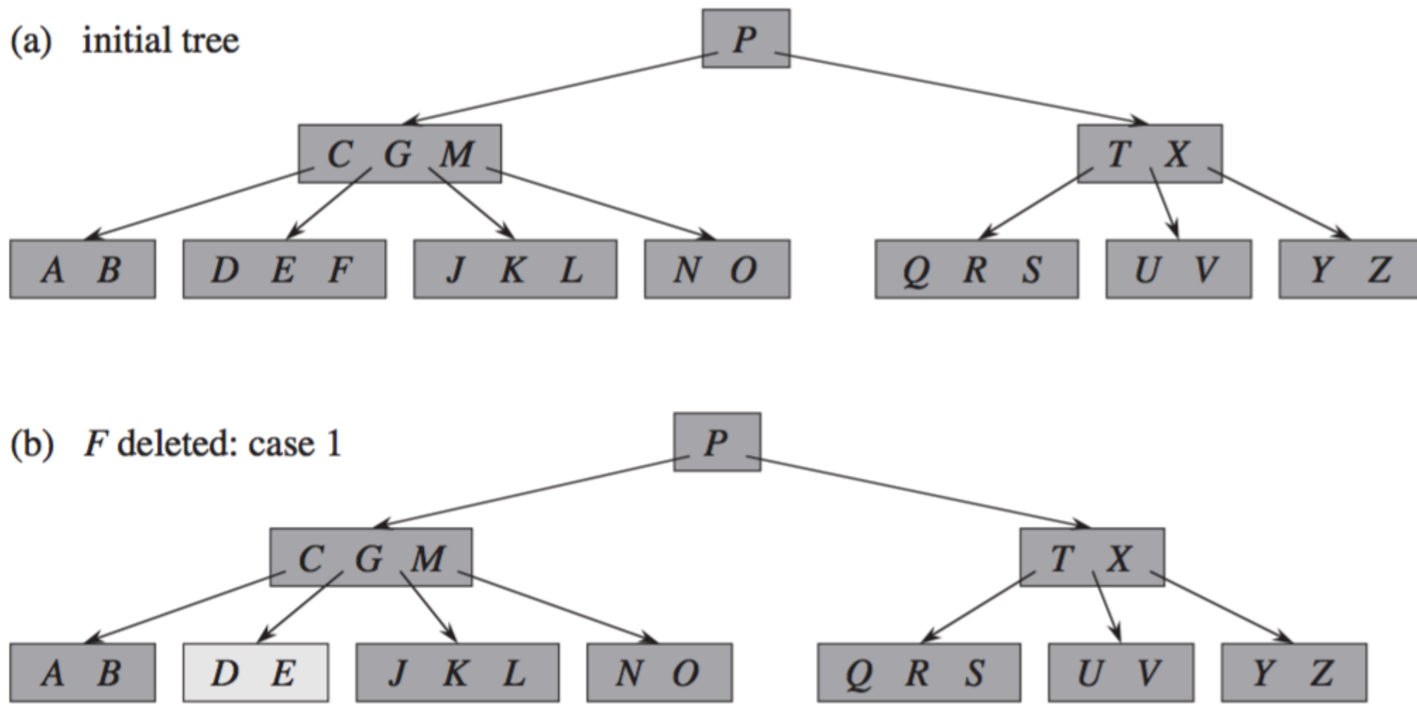
(e) *F* inserted



Delete a key

There could be several cases while delete a key ***k*** from B-tree

Case 1: If ***k*** is in node *x* and *x* is a leaf

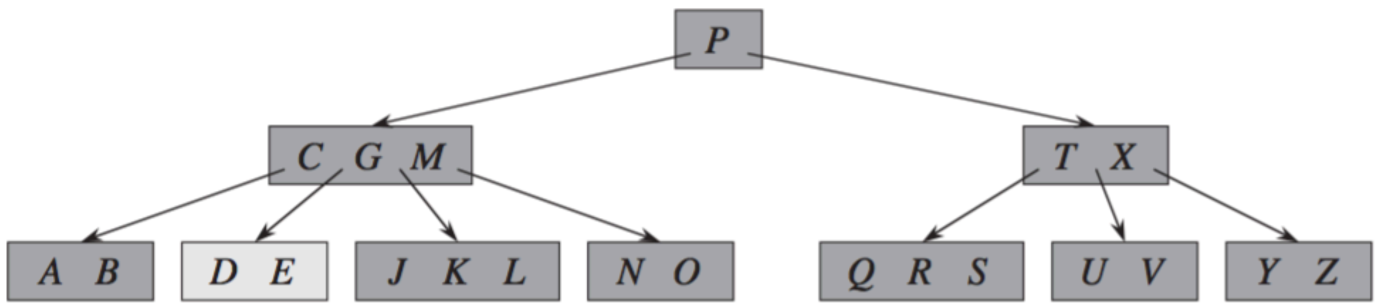


$t = 3$

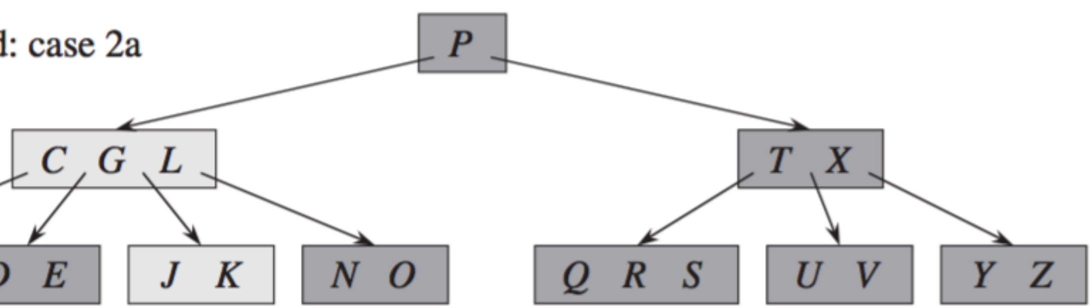
Delete a key …

Case 2: If ***k*** is in node *x* and *x* is an internal node:

2a) If the child *y* that precedes ***k*** in node *x* has at least ***t*** keys, then find the predecessor of ***k*** in the subtree rooted by *y*.



(c) *M* deleted: case 2a

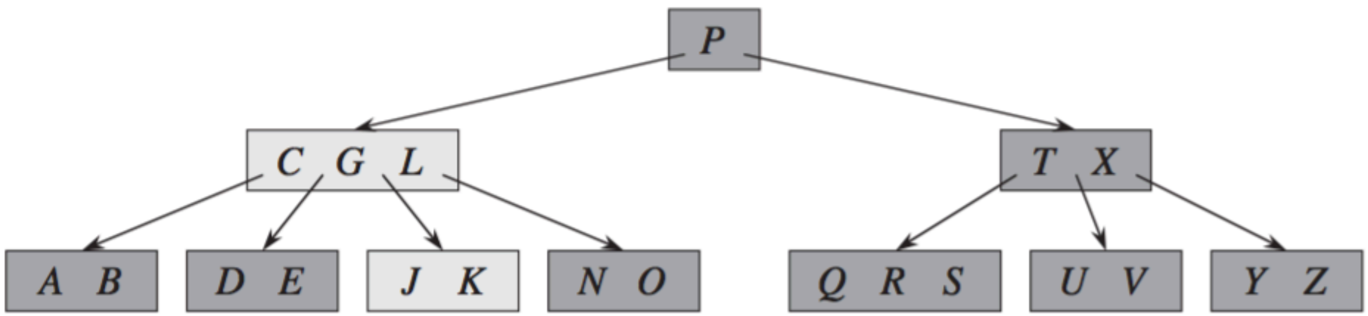


Delete a key …

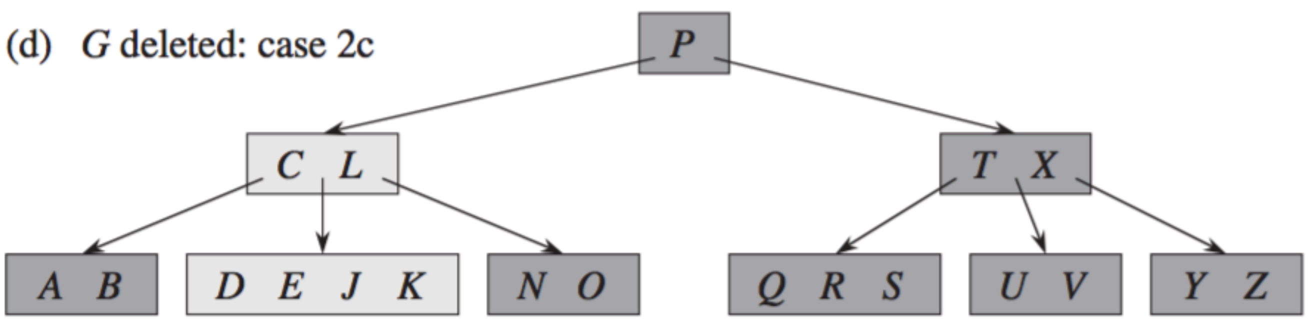
Case 2b) If child y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor of k in the subtree rooted at z.

Delete a key …

Case 2c) if both *y* and ***z*** have only ***t*** -1 keys, merge ***k*** and all of ***z*** into *y*, so that *x* loses both ***k*** and the pointer to ***z***, and y now contains 2***t*** -1 keys. Then free ***z*** and recursively delete ***k*** from *y*.



(d) *G* deleted: case 2c



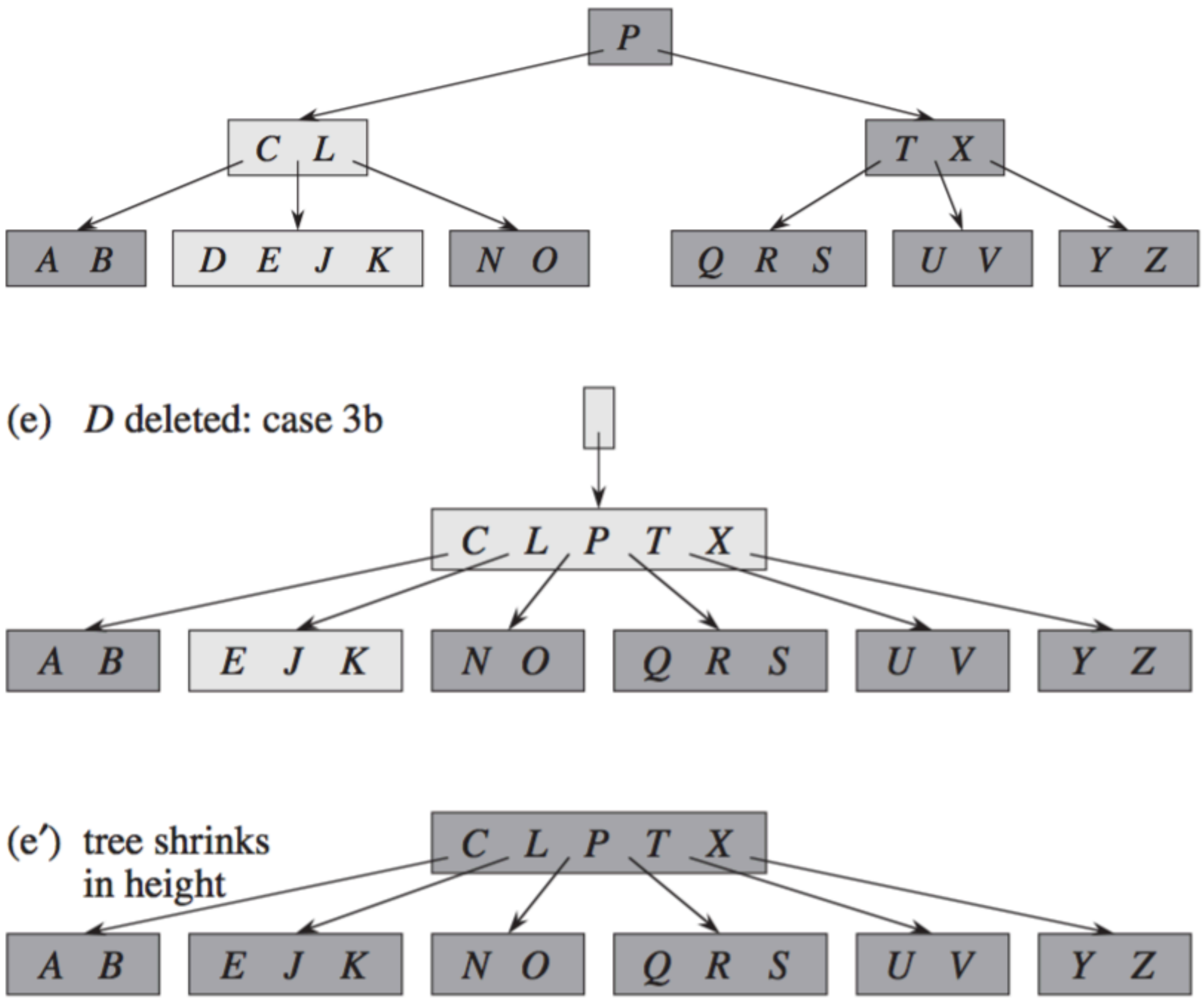
Delete a key …

Case 3: If $\boldsymbol{k}$ does not present in internal node $\boldsymbol{x}$, determine the root $\boldsymbol{x.c_i}$ of the appropriate subtree that must contain $\boldsymbol{k}$, if $\boldsymbol{k}$ is in the tree at all. If $\boldsymbol{x.c_i}$ has only $\boldsymbol{t}$-1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least $\boldsymbol{t}$ keys. Then finish by recursing on the appropriate child of $\boldsymbol{x}$.

3a) If $\boldsymbol{x.c_i}$ has only t -1 keys but has an immediate sibling with at least $\boldsymbol{t}$ keys, give $\boldsymbol{x.c_i}$ an extra key by moving a key from x down into $\boldsymbol{x.c_i}$, moving a key from $\boldsymbol{x.c_i}$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $\boldsymbol{x.c_i}$.

3b) If $\boldsymbol{x.c_i}$ and both of $\boldsymbol{x.c_i}$'s immediate siblings have $\boldsymbol{t}$ -1 keys, merge $\boldsymbol{x.c_i}$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

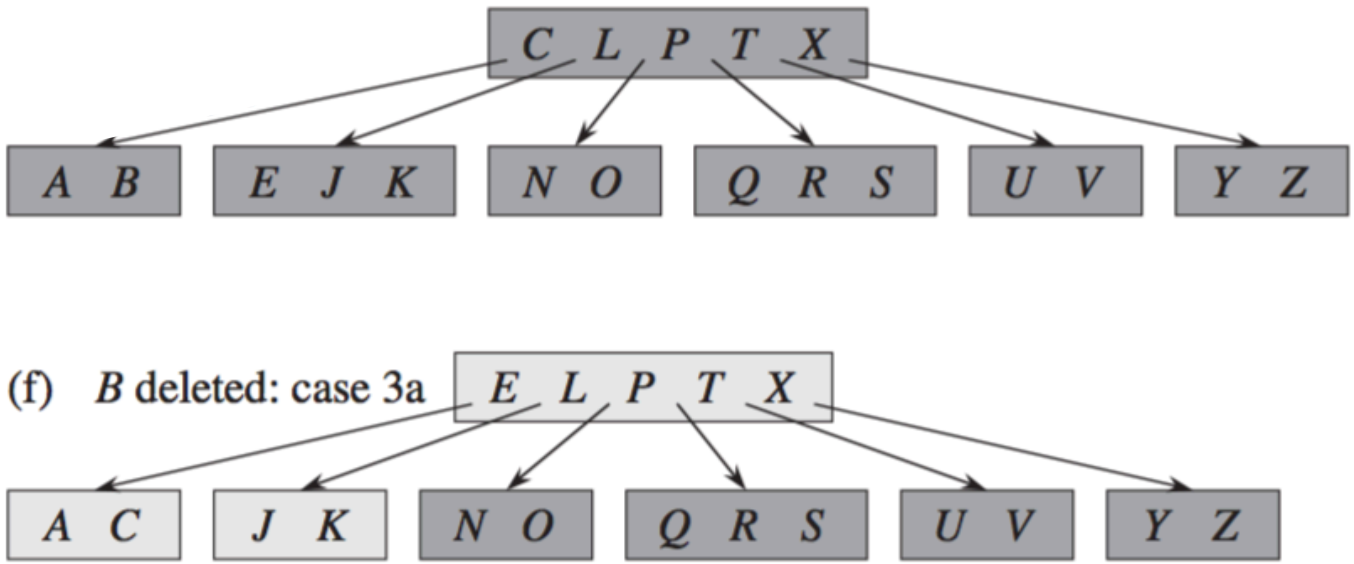
Delete a key …



(e) *D* deleted: case 3b

(e′) tree shrinks in height

Delete a key …



C L P T X
A B
E J K
N O
Q R S
U V
Y Z
(f) B deleted: case 3a
E L P T X
A C
J K
N O
Q R S
U V
Y Z

Next Week Topics

- Elementary Graph Algorithms (Chapter 22)