B-tree

DATA STRUCTURES & ALGORITHMS

Definition

A ***B-tree*** T is a rooted tree (whose root is $T.root$) having the following properties:

1. Every node x has the following attributes:

   a. $x.n$, the number of keys currently stored in node x,

   b. the $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \le x.key_2 \le \cdots \le x.key_{x.n}$,

   c. $x.leaf$, a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.

2. Each internal node x also contains $x.n + 1$ pointers $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.

Definition

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $x.c_i$, then

$$k_1 \le x.key_1 \le k_2 \le x.key_2 \le \cdots \le x.key_{x.n} \le k_{x.n+1} .$$

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

Definition

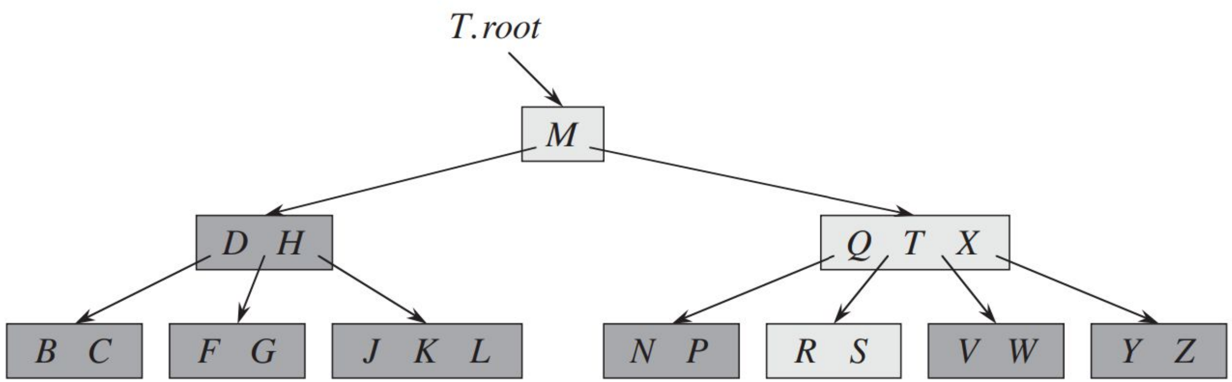
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the ***minimum degree*** of the B-tree:

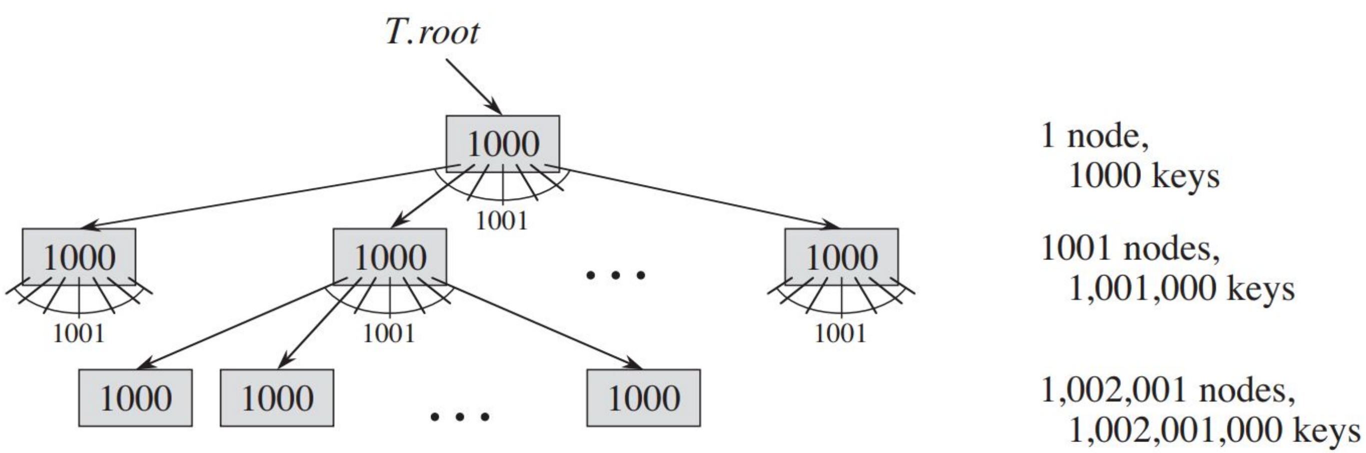
   a. Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

   b. Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t - 1$ keys.[2]

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a ***2-3-4 tree***. In practice, however, much larger values of t yield B-trees with smaller height.

Example

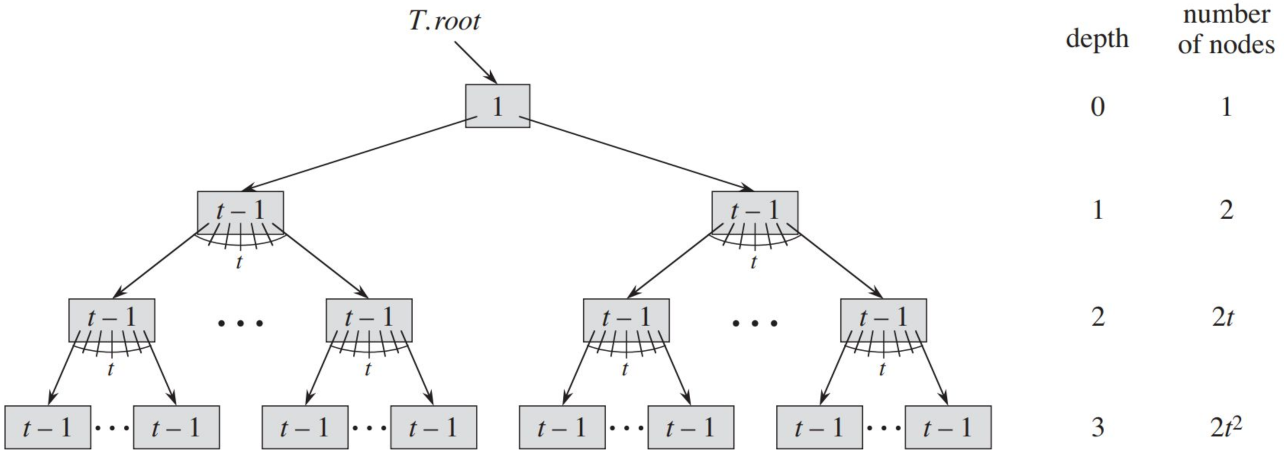

T.root
M
D H
Q T X
B C
F G
J K L
N P
R S
V W
Y Z


T.root
1000
1001
1000
1000
1000
1001
1001
1001
1000
1000
1000
1 node,
1000 keys
1001 nodes,
1,001,000 keys
1,002,001 nodes,
1,002,001,000 keys

The height of a B-tree

If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2} .$$



The height of a B-tree

number n of keys satisfies the inequality

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) \\ &= 2t^h - 1 \,. \end{aligned}$$

By simple algebra, we get $t^h \leq (n+1)/2$. Taking base-t logarithms of both sides proves the theorem. ∎

Searching a B-tree

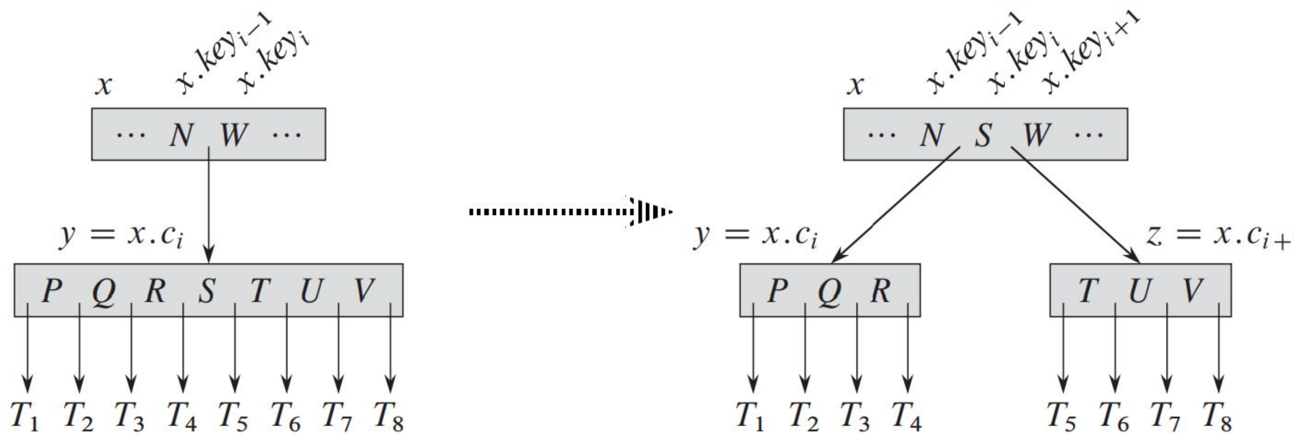
```
B-TREE-SEARCH(x, k)
i = 1
while i ≤ x.n and k > x.key_i
    i = i + 1
if i ≤ x.n and k == x.key_i
    return (x, i)
elseif x.leaf
    return NIL
else DISK-READ(x.c_i)
    return B-TREE-SEARCH(x.c_i, k)
```

Creating an empty B-tree

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

B-TREE-CREATE requires $O(1)$ disk operations and $O(1)$ CPU time.

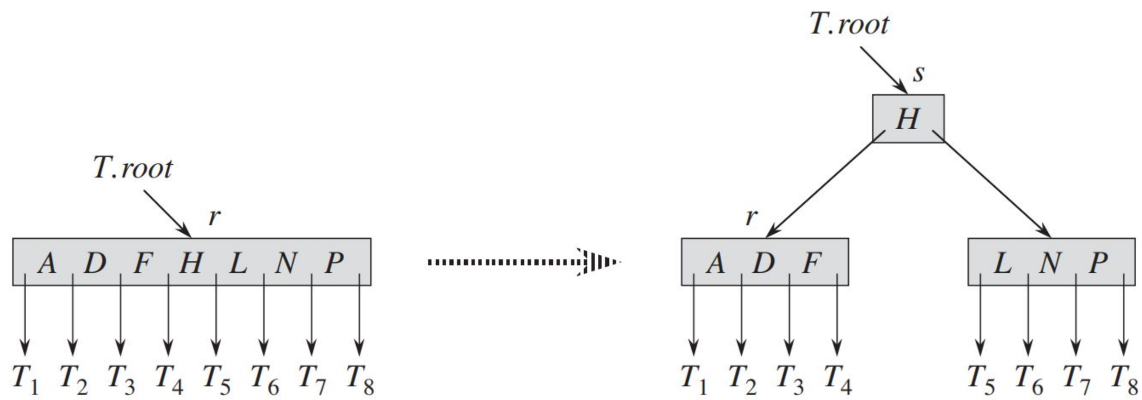
Insert



The procedure B-TREE-SPLIT-CHILD takes as input a nonfull internal node **x** and an index i such that **x.ci** is a full child of **x**.

```
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)
```

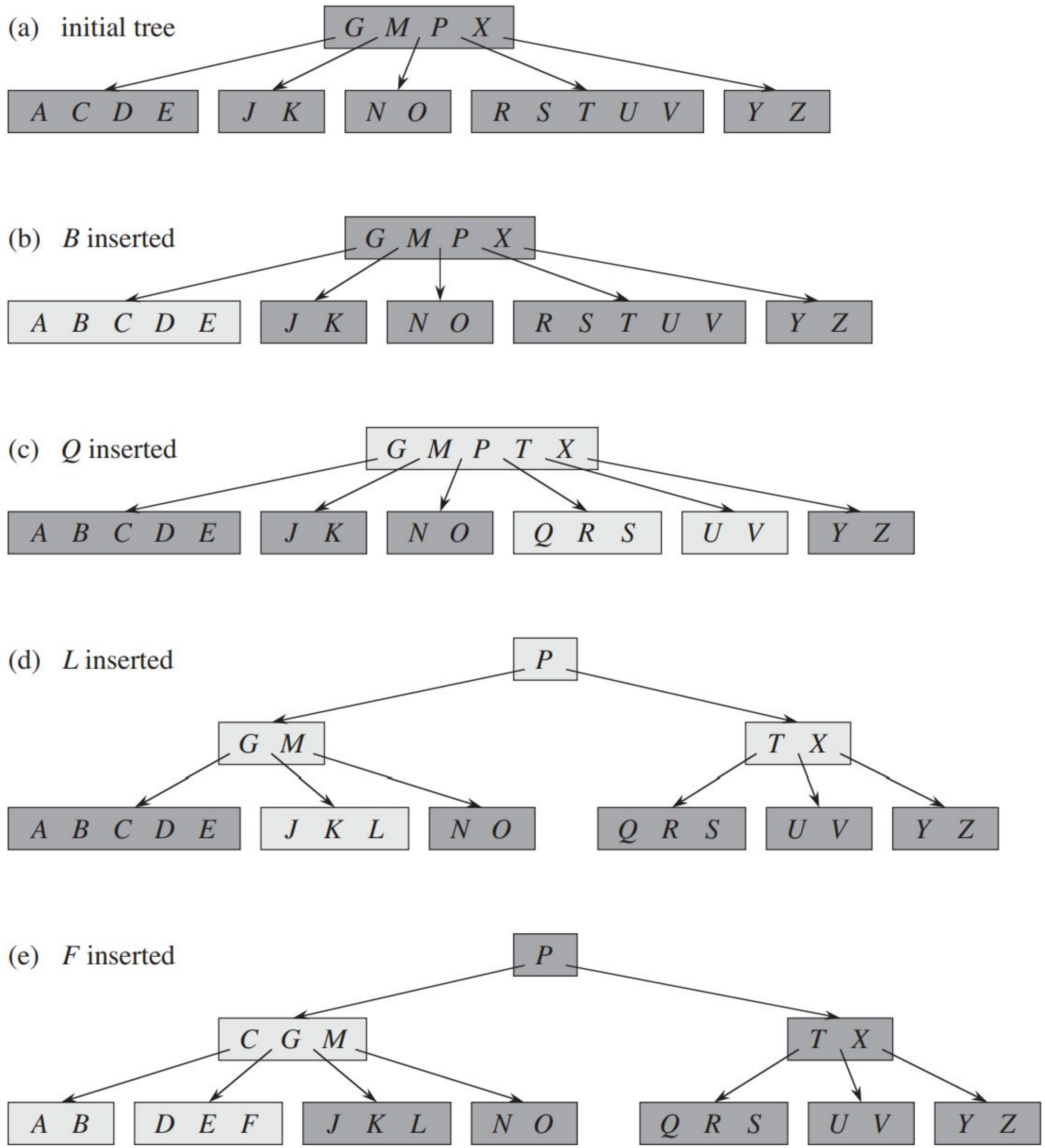
Insert




```
B-TREE-INSERT(T, k)
r = T.root
if r.n == 2t − 1
    s = ALLOCATE-NODE()
    T.root = s
    s.leaf = FALSE
    s.n = 0
    s.c_1 = r
    B-TREE-SPLIT-CHILD(s, 1)
    B-TREE-INSERT-NONFULL(s, k)
else B-TREE-INSERT-NONFULL(r, k)
```

Insert

```
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)
```


(a) initial tree
G M P X
A C D E
J K
N O
R S T U V
Y Z
(b) B inserted
G M P X
A B C D E
J K
N O
R S T U V
Y Z
(c) Q inserted
G M P T X
A B C D E
J K
N O
Q R S
U V
Y Z
(d) L inserted
P
G M
T X
A B C D E
J K L
N O
Q R S
U V
Y Z
(e) F inserted
P
C G M
T X
A B
D E F
J K L
N O
Q R S
U V
Y Z

Delete

1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following:

    a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

    b. If 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 k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)

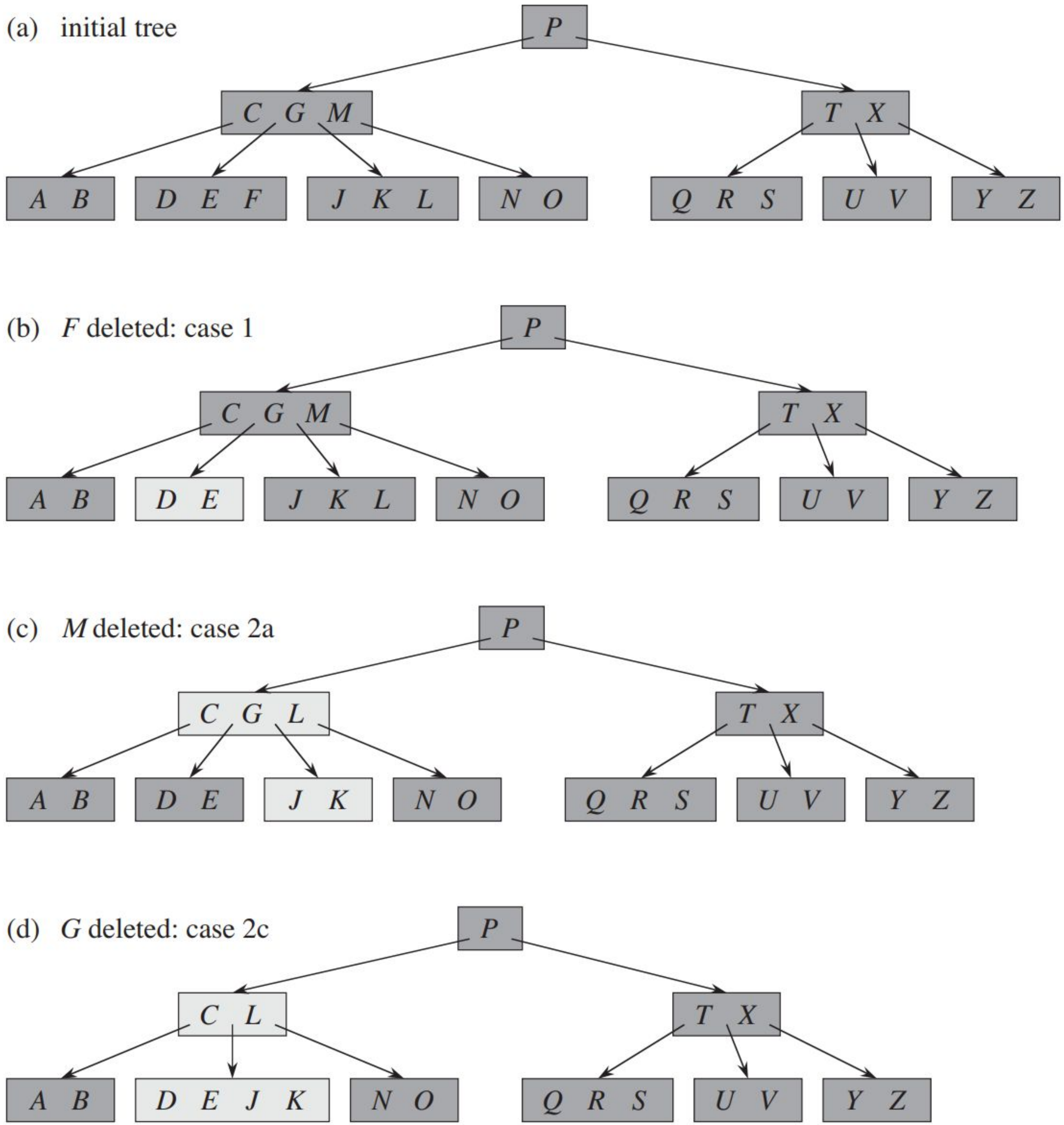
    c. Otherwise, if both y and z have only $t - 1$ keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t - 1$ keys. Then free z and recursively delete k from y.

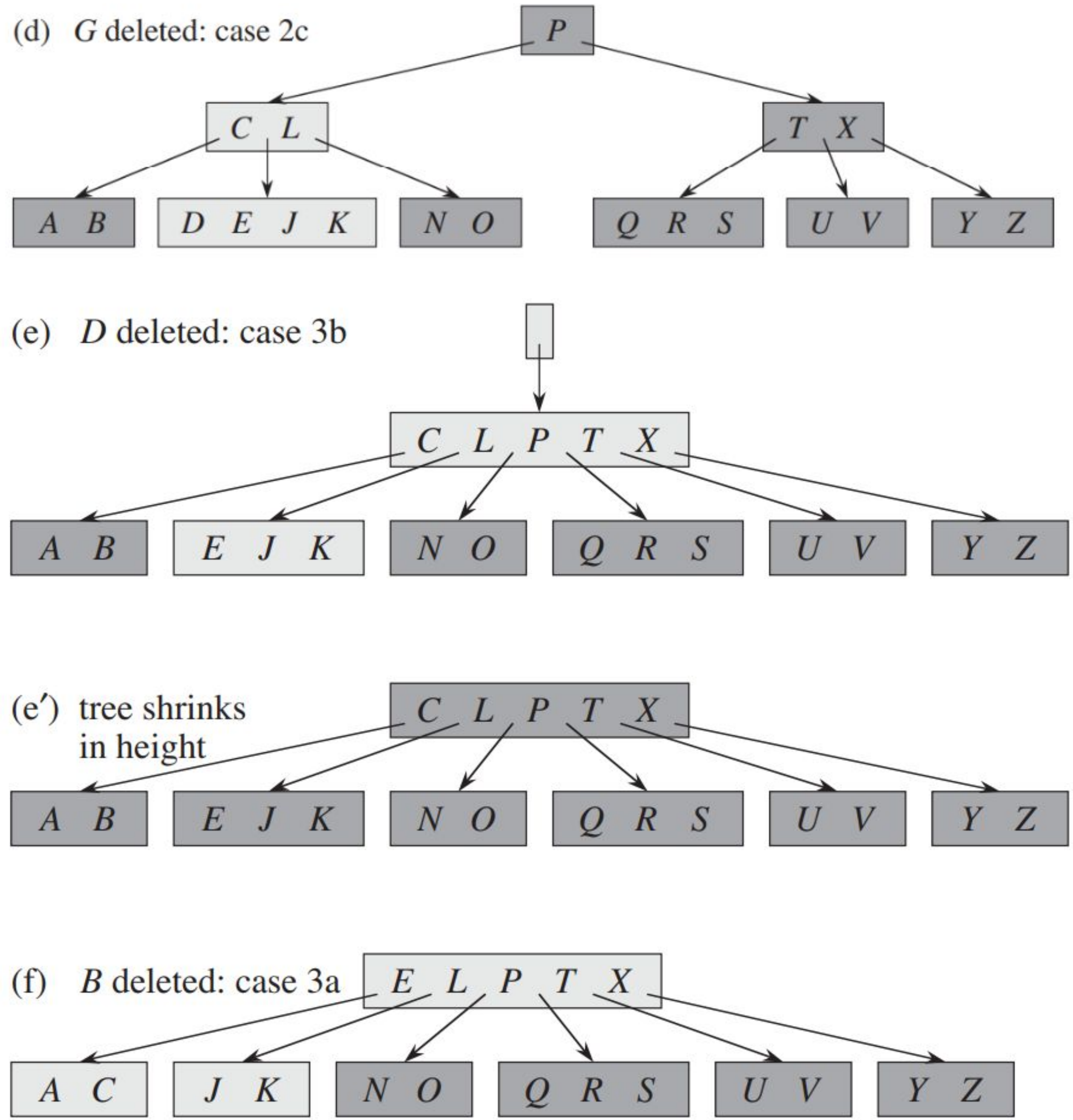
Delete

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

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

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


(a) initial tree
P
C G M
T X
A B
D E F
J K L
N O
Q R S
U V
Y Z
(b) F deleted: case 1
P
C G M
T X
A B
D E
J K L
N O
Q R S
U V
Y Z
(c) M deleted: case 2a
P
C G L
T X
A B
D E
J K
N O
Q R S
U V
Y Z
(d) G deleted: case 2c
P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z


(d) G deleted: case 2c
P
C L
T X
A B
D E J K
N O
Q R S
U V
Y Z
(e) D deleted: case 3b
C L P T X
A B
E J K
N O
Q R S
U V
Y Z
(e′) tree shrinks in height
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