Algorithms
Chapter 18
B-Trees

Associate Professor: Ching-Chi Lin
林清池 副教授
chingchi.lin@gmail.com

Department of Computer Science and Engineering
National Taiwan Ocean University

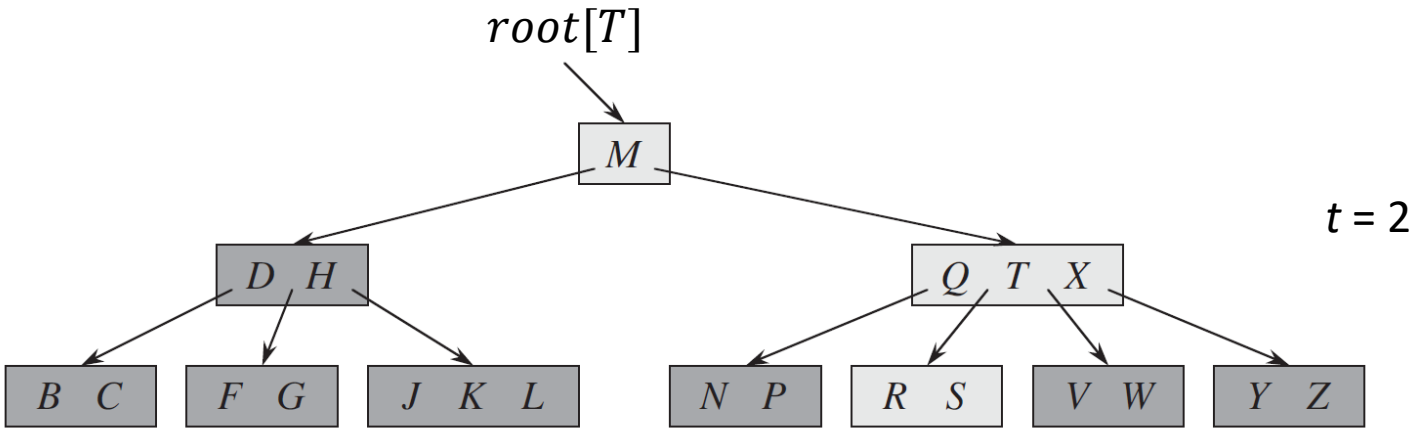
Outline

- **Definition of B-trees**
- Basic operations on B-trees
- Deleting a key from a B-tree

Overview$_{1/2}$

- B-trees are balanced search trees designed to work well on magnetic disks.
- B-trees are similar to red-black trees, but they are better at minimizing disk I/O operations.
- **Red-black trees**
  - A variation of binary search trees.
  - **Balanced**: height is $O(\lg n)$, where n is the number of nodes.
  - Operations will take $O(\lg n)$ time in the worst case.
- **B-trees**
  - Generalize binary search trees in a natural manner.
  - **Balanced**: height is $O(\lg_t n)$, where n is the number of nodes.
  - Operations will take $O(t \lg_t n)$ time in the worst case, where t is the minimum degree of the B-tree.

Overview$_{2/2}$



- Running time of a B-tree algorithm is determined by the number of DISK-READ and DISK-WRITE operations.
  - Thus, a B-tree node is usually as large as a whole disk page.
  - The "branching factors" between 50 and 2000 are often used, depending on the size of a key relative to the size of a page.
- An internal node *x* containing *n*[*x*] keys has *n*[*x*]+1 children.

Properties of B-trees$_{1/2}$

- A **B-tree** *T* is a rooted tree having the following properties:
  - Every node *x* has the following fields:
    - $n[x]$, the number of keys in node *x*,
    - $key_1[x] \le key_2[x] \le \cdots \le key_{n[x]}[x]$,
    - $leaf[x]$, $leaf[x]$ = TRUE if *x* is a leaf, $leaf[x]$ = FALSE if *x* is an internal node.
  - Each internal node *x* also contains $n[x]+1$ pointers $c_1[x]$, $c_2[x]$, ..., $c_{n[x]+1}[x]$ to its children.
  - If k_i is any key stored in the subtree with root $c_i[x]$, then

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

  - All leaves have the same depth, which is the tree's height *h*.
  - Every node *x* other than the root must have $t - 1 \le n[x] \le 2t - 1$, where $\boldsymbol{t \ge 2}$ is the **minimum degree** of the B-tree.
  - If the tree is nonempty, the root has $1 \le n[root] \le 2t - 1$.

Properties of B-trees$_{2/2}$

- 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**.

Height of a B-tree$_{1/2}$

- **Lemma 18.1** If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$, $h \leq \log_t \frac{n+1}{2}$.

  Proof:

  - The root contains at least one key.
  - Thus, there are at least 2 nodes at depth 1.
  - All other nodes contain at least $t - 1$ keys.
  - So, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on.

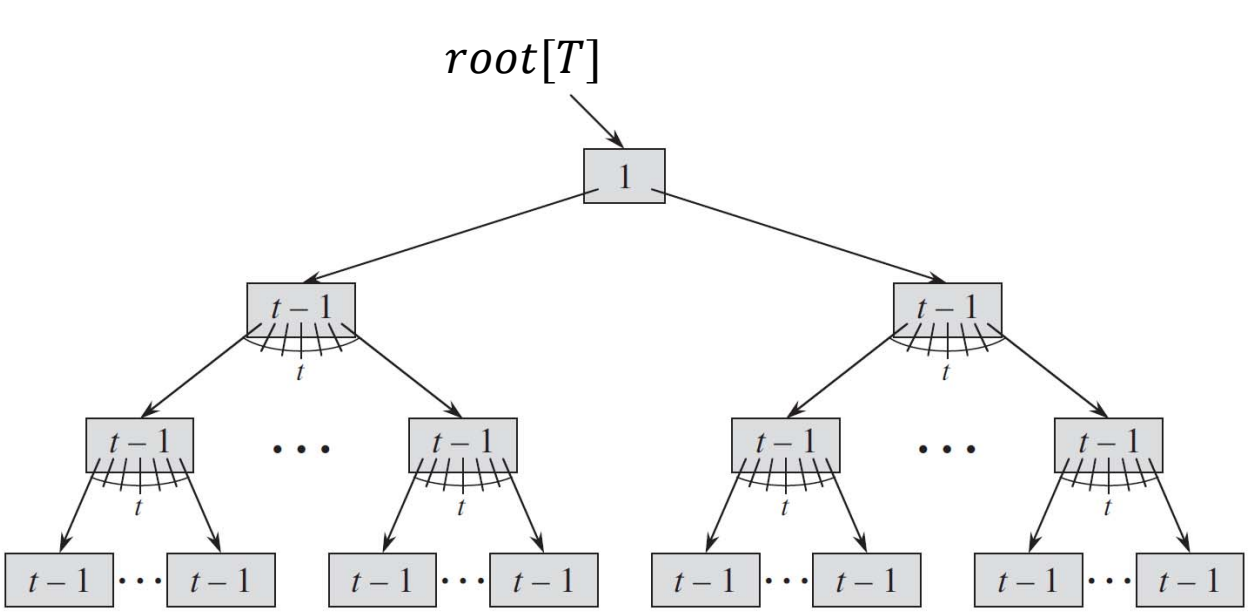
- Then, we have $n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$

$$= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right)$$

$$= 2t^h - 1.$$

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

Height of a B-tree$_{2/2}$



depth	number of nodes
0	1
1	2
2	$2t$
3	$2t^2$

Outline

- Definition of B-trees
- **Basic operations on B-trees**
- Deleting a key from a B-tree

Searching a B-tree

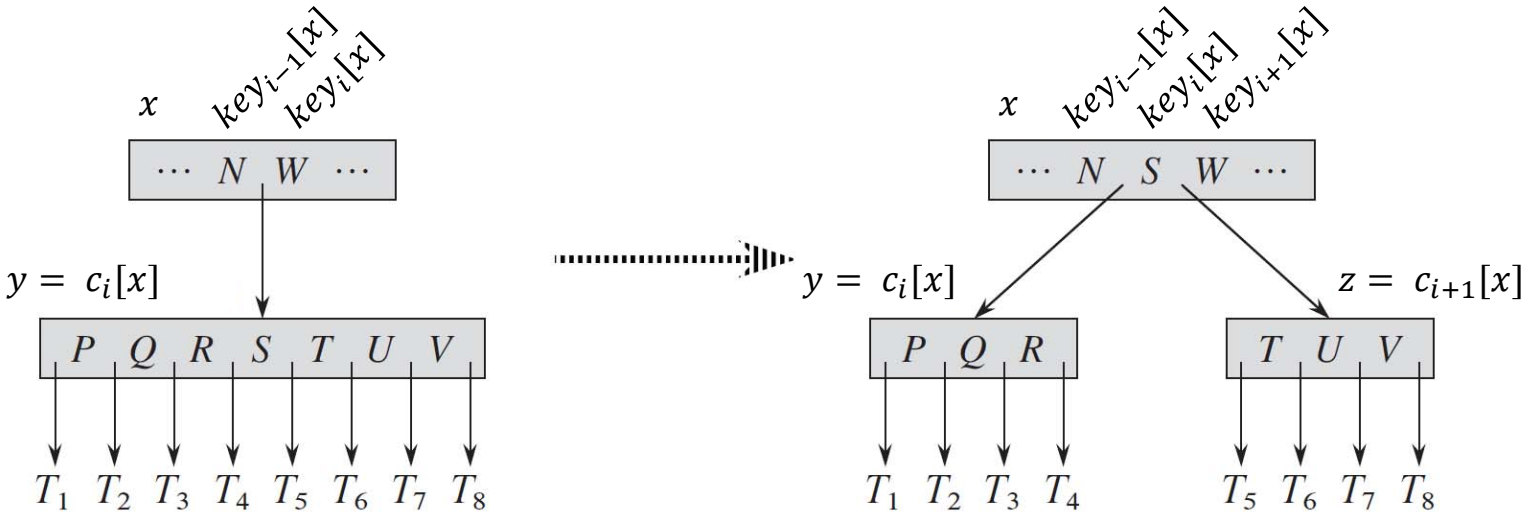
- **B-TREE-SEARCH** is a straightforward generalization of the **TREE-SEARCH** procedure defined for binary search trees.
- Instead of making a binary, or "two-way" branching decision at each node, we make an ($n[x]$+1)-way branching decision.

```
B-TREE-SEARCH(x, k)
1.    i ← 1
2.    while i ≤ n[x] and k > key_i[x]
3.        i ← i + 1
4.    if i ≤ n[x] and k = key_i[x]
5.        return (x, i)
6.    elseif leaf[x]
7.        return NIL
8.    else DISK-READ(c_i[x])
9.        return B-TREE-SEARCH(c_i[x], k)
```

- Since $n[x] < 2t$, the running is $O(th) = O(t \log_t n)$.

Splitting a node in a B-tree$_{1/2}$

- A fundamental operation used during insertion is the **splitting** a full node y (having $2t - 1$ keys) around its **median key** $key_t[y]$ into two nodes having $t - 1$ keys each.



Splitting a node with **$t = 4$**. Node y is split into two nodes, y and z, and the median key S of y is moved up into y's parent.

If y has no parent, then the tree grows in height by one.

Splitting a node in a B-tree$_{2/2}$

B-TREE-SPLIT-CHILD(x, i, y)

1. $z \leftarrow$ ALLOCATE-NODE()
2. $leaf[z] \leftarrow leaf[y]$
3. $n[z] \leftarrow t - 1$

$\Theta(1)$

4. **for** $j \leftarrow 1$ **to** $t - 1$
5. $\quad key_j[z] \leftarrow key_{j+t}[y]$
6. **if** not $leaf\ [y]$
7. $\quad$ **for** $j \leftarrow 1$ **to** t
8. $\quad\quad c_j[z] \leftarrow c_{j+t}[y]$
9. $n[y] \leftarrow t - 1$
10. **for** $j \leftarrow n[x] + 1$ **downto** $i + 1$
11. $\quad c_{j+1}[x] \leftarrow c_j[x]$
12. $c_{i+1}[x] \leftarrow z$
13. **for** $j \leftarrow n[x]$ **downto** i
14. $\quad key_{j+1}[x] \leftarrow key_j[x]$

$\Theta(t)$

15. $key_i[x] \leftarrow key_t[y]$
16. $n[x] \leftarrow n[x] + 1$
17. DISK-WRITE(y)
18. DISK-WRITE(z)
19. DISK-WRITE(x)

$\Theta(1)$

Time : $O(t)$

Inserting a key into a B-tree$_{1/3}$

- **B-TREE-INSERT** inserts a key k into a B-tree T of height h in a single pass down the tree. Requiring $O(h)$ disk accesses.
- Use B-TREE-SPLIT-CHILD to guarantee that the recursion never descends to a full node.

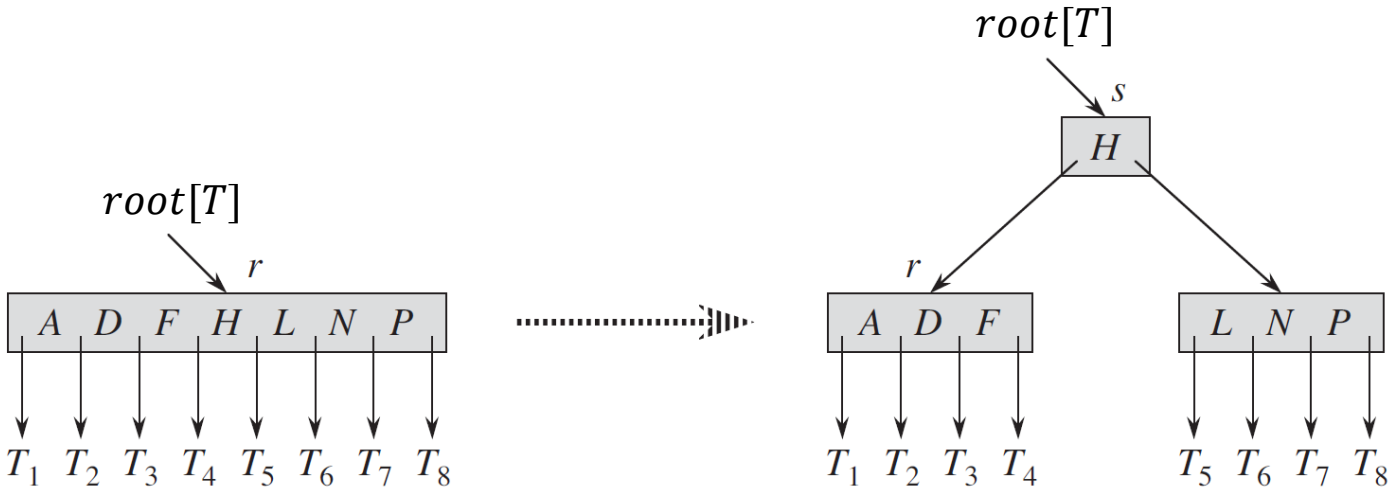
```
B-TREE-INSERT(T, k)
1.   r ← root[T]
2.   if n[r] = 2t − 1
3.       s ← ALLOCATE-NODE()
4.       root[T] ← s
5.       leaf[s] ← FALSE
6.       n[s] ← 0
7.       c1[s] ← r
8.       B-TREE-SPLIT-CHILD(s, 1, r)
9.       B-TREE-INSERT-NONFULL(s, k)
10.  else B-TREE-INSERT-NONFULL(r, k)
```

Handle the case in which the root node r is full: the root is split and a new node s becomes the root.

- The CPU time required is $O(th) = O(t \log_t n)$.

Inserting a key into a B-tree$_{2/3}$

- Unlike a binary search tree, a B-tree increases in height at the top instead of at the bottom.



Splitting the root with ***t* = 4**. Root node *r* is split in two, and a new root node *s* is created.

The B-tree grows in height by one when the root is split.

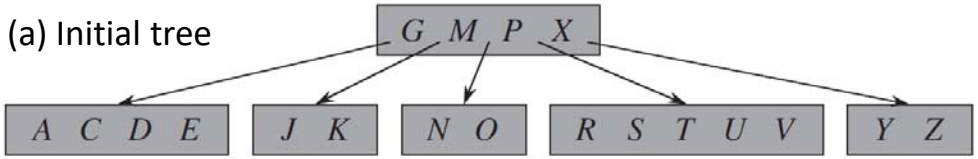
Inserting a key into a B-tree$_{3/3}$

- B-TREE-INSERT-NONFULL inserts key k into node x, which is assumed to be nonfull when the procedure is called.

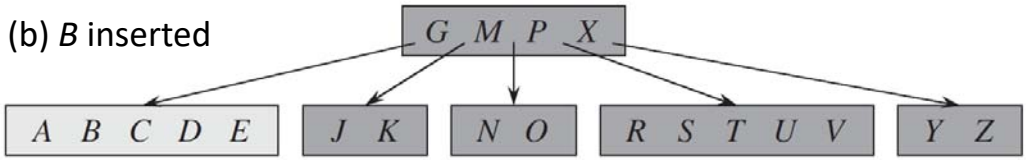
```
B-TREE-INSERT-NONFULL(x, k)
1.   i ← n[x]
2.   if leaf[x]
3.       while i ≥ 1 and k < key_i[x]
4.           key_{i+1}[x] ← key_i[x]
5.           i ← i – 1
6.       key_{i+1}[x] ← k
7.       n[x] ← n[x] + 1
8.       DISK-WRITE(x)
9.   else  while i ≥ 1 and k < key_i[x]
10.              i ← i – 1
11.          i ← i + 1
12.          DISK-READ(c_i[x])
13.          if n[c_i[x]] = 2t – 1
14.              B-TREE-SPLIT-CHILD(x, i, c_i[x])
15.              if k > key_i[x]
16.                  i ← i + 1
17.          B-TREE-INSERT-NONFULL(c_i[x], k)
```

Time : $O(th)$
$= O(t \log_t n)$.

(a) Initial tree

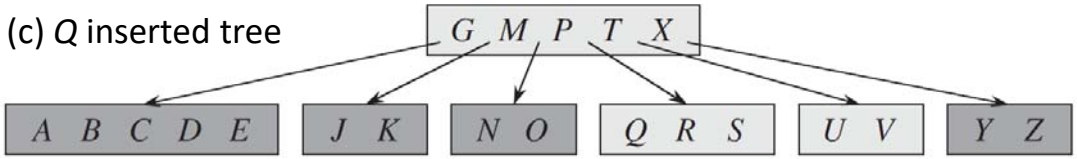


(b) *B* inserted

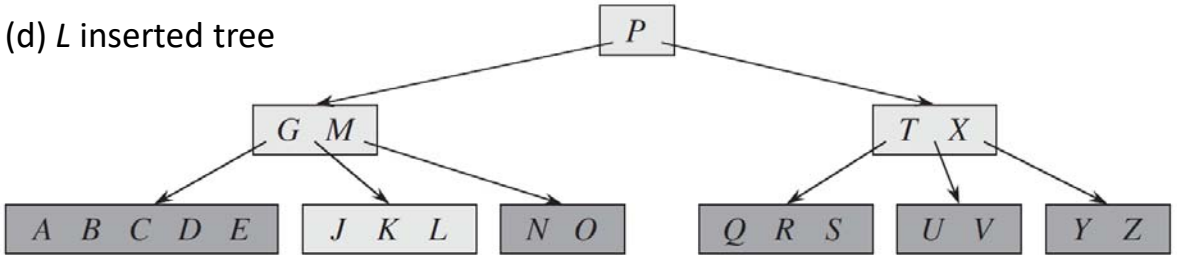


$t = 3$

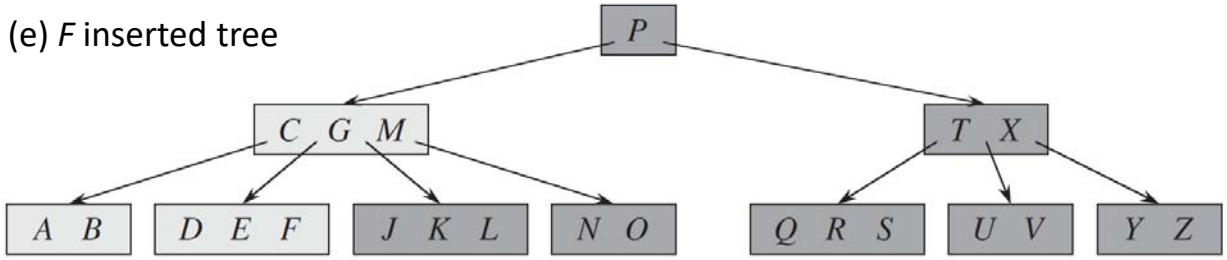
(c) *Q* inserted tree



(d) *L* inserted tree



(e) *F* inserted tree



Outline

- Definition of B-trees
- Basic operations on B-trees
- **Deleting a key from a B-tree**

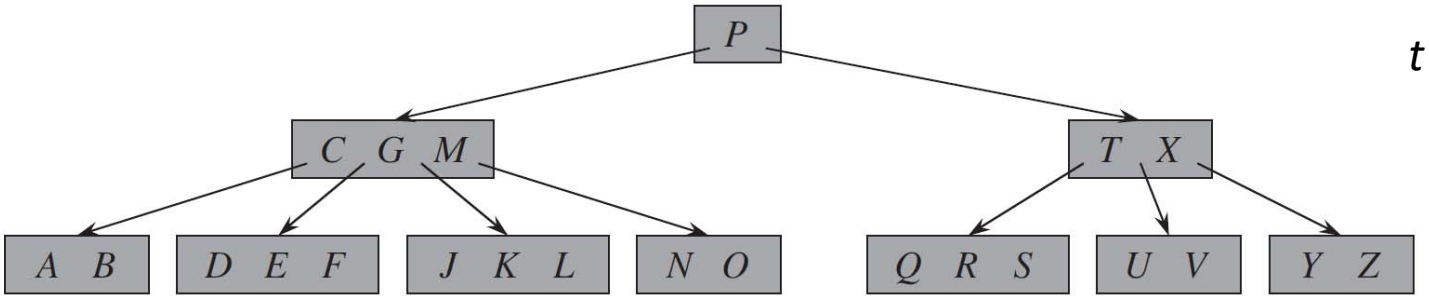
Deleting a key into a B-tree$_{1/3}$

- **B-TREE-Delete** deletes the key *k* from the subtree rooted at *x*.
  - Guarantee that whenever B-TREE-DELETE is called recursively on a node *x*, the number of keys in *x* is at least the minimum degree *t*.
  - Allows us to delete a key from the tree in one downward pass without having to "back up".

- If the root node *x* becomes an internal node having no keys, then (occur in case 3c, below)
  - *x* is deleted,
  - *x*'s only child $c_1[x]$ becomes the new root of the tree,
  - decreasing the height of the tree by one, and
  - preserving the property that the root of the tree contains at least one key.

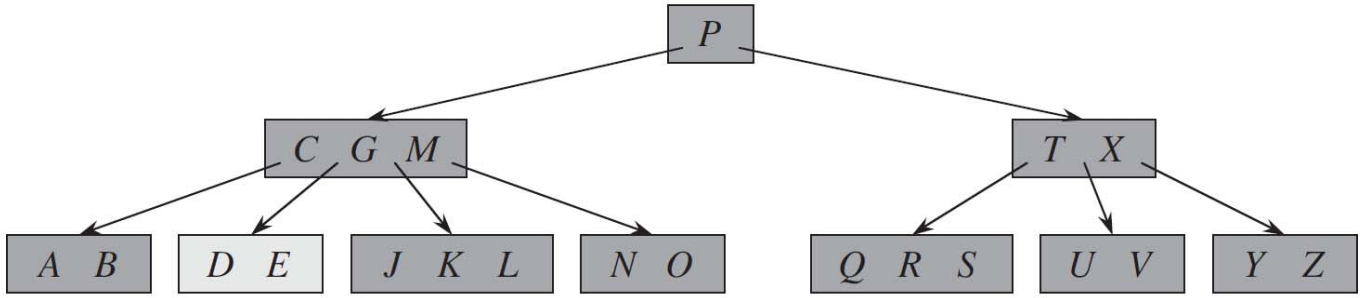
Case 1: k is in node x and x is a leaf

- Delete the key k from x.



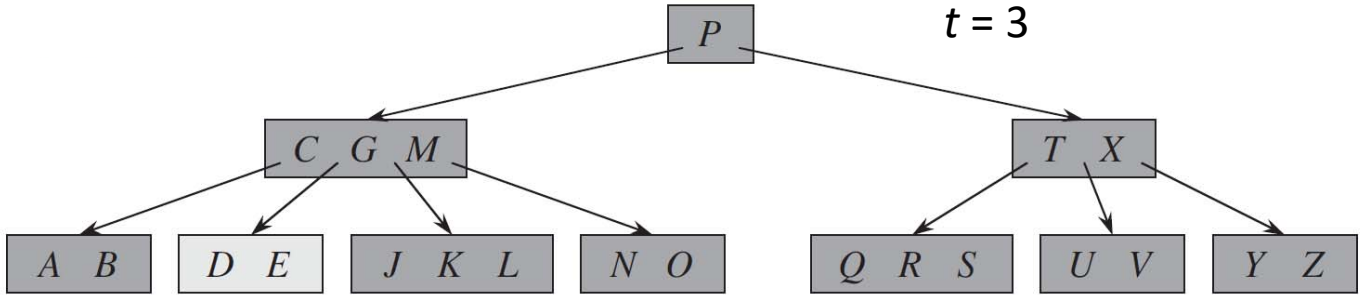
$t = 3$

Case 1: *F* deleted.

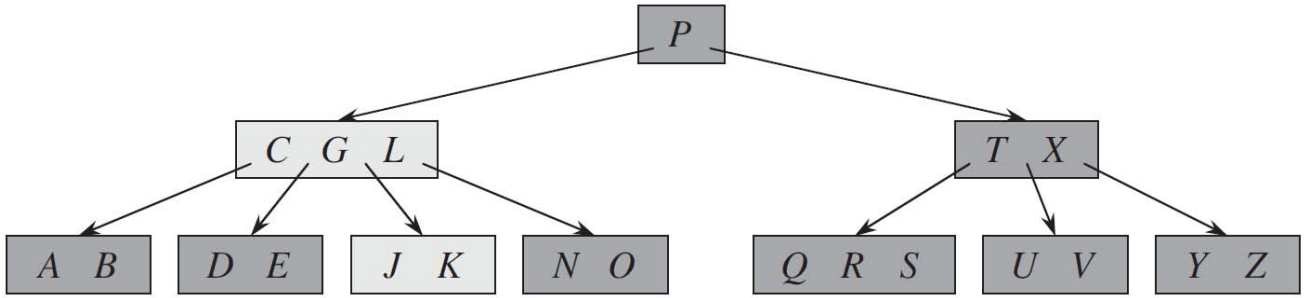


Case 2: k is in node x and x is an internal node$_{1/3}$

- Case 2a: the child y that precedes k has at least t keys.
  - Find the predecessor k' of k in the subtree rooted at y.
  - Recursively delete k', and replace k by k' in x.

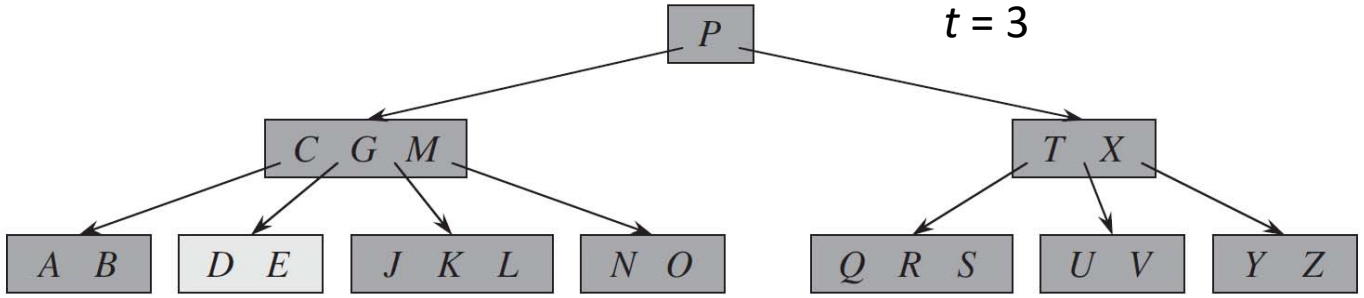


Case 2a: *M* deleted.

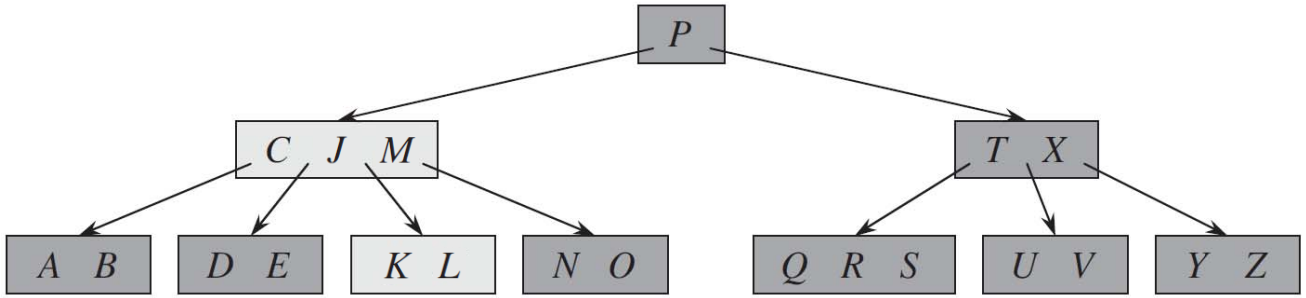


Case 2: k is in node x and x is an internal node$_{2/3}$

- Case 2b: the child z that follows k has at least t keys.
  - Find the successor k' of k in the subtree rooted at z.
  - Recursively delete k', and replace k by k' in x.

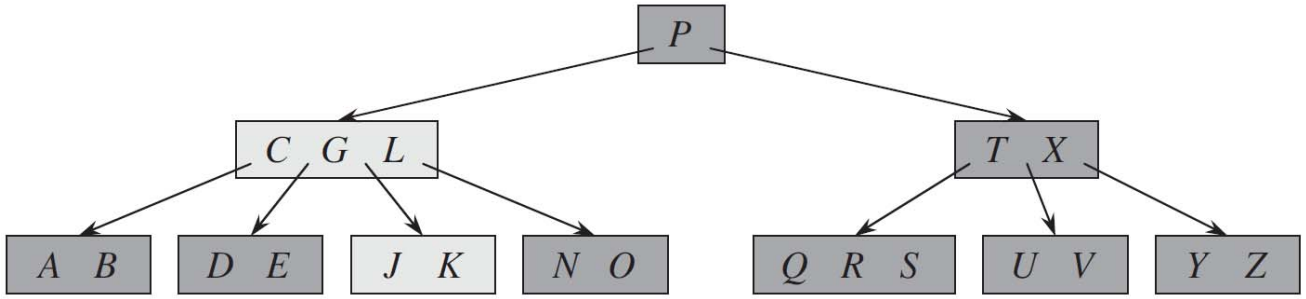


Case 2a: G deleted.



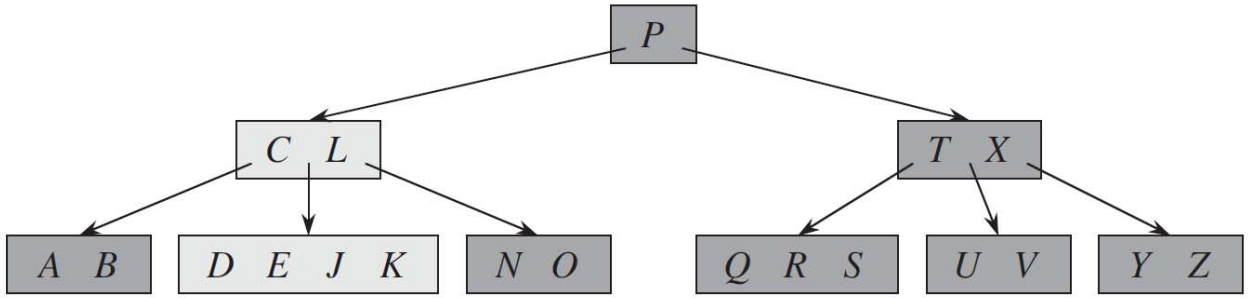
Case 2: k is in node x and x is an internal node$_{3/3}$

- Case 2c: both y and z have only $t - 1$ keys.
  - Merge k and all of z into y.
  - Free z and recursively delete k from y.



$t = 3$

Case 2c: G deleted.

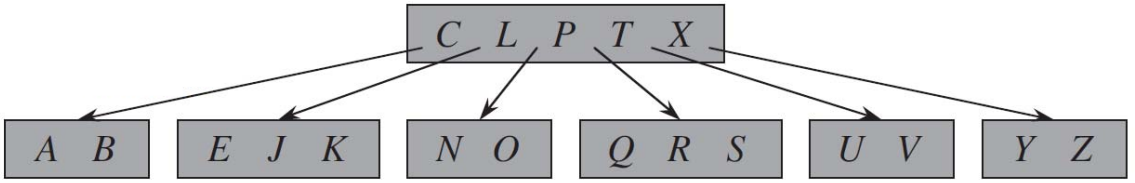


Case 3: k is not present in internal node $x_{1/3}$

- Determine the root $c_i[x]$ of the appropriate subtree that must contain k.
- Case 3a: $c_i[x]$ has at least t keys
  - Recursively delete k from $c_i[x]$.
- Case 3b: $c_i[x]$ has only $t - 1$ keys but has an immediate sibling with at least t keys.
- Case 3c: $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t - 1$ keys.

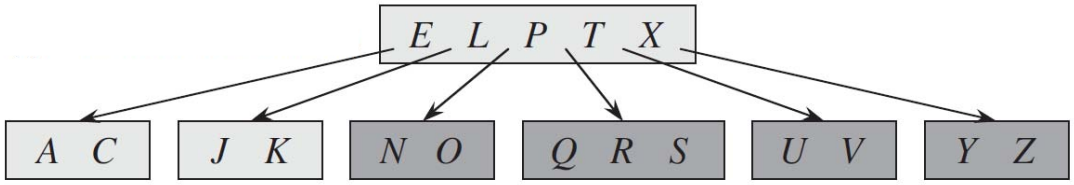
Case 3: k is not present in internal node $x_{2/3}$

- Case 3b: $c_i[x]$ has only $t - 1$ keys but has an immediate sibling with at least t keys.
  - Give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$.
  - Moving a key from $c_i[x]$'s immediate left or right sibling up into x.
  - Moving the appropriate child pointer from the sibling into $c_i[x]$.



$t = 3$

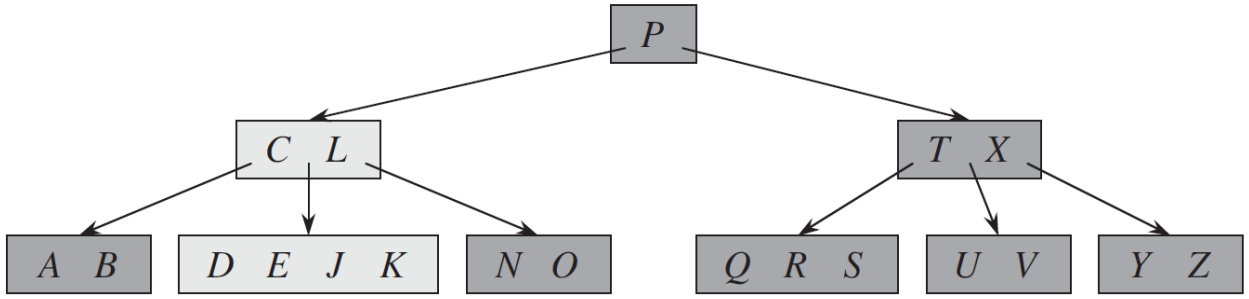
Case 3b：B deleted.



Case 3: k is not present in internal node $x_{3/3}$

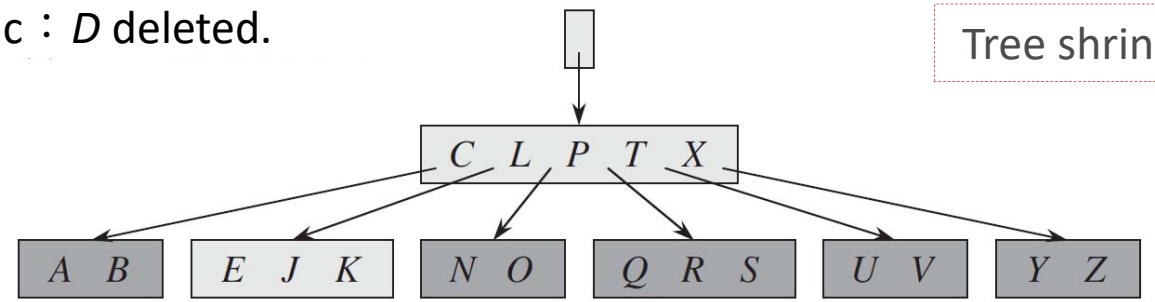
- Case 3c: $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t - 1$ keys.
  - Merge $c_i[x]$ with one sibling.
  - Moving a key from x down into the new merged node to become the median key for that node.

$t = 3$

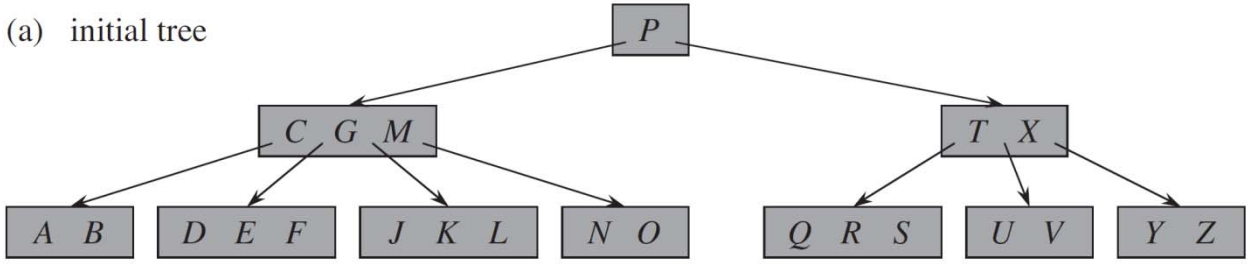


Case 3c：D deleted.

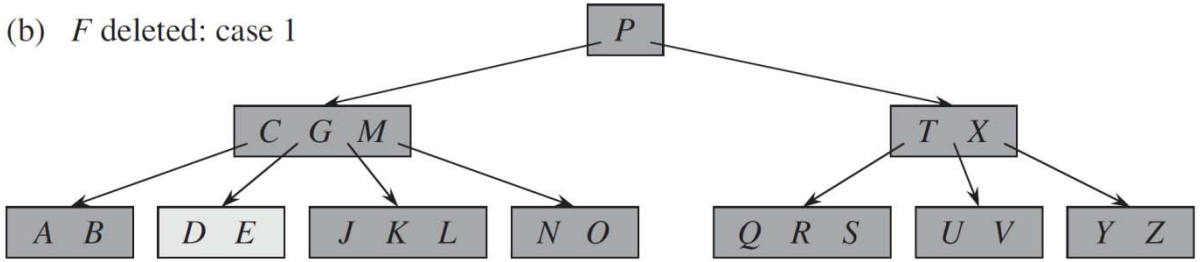
Tree shrinks in height.



(a) initial tree

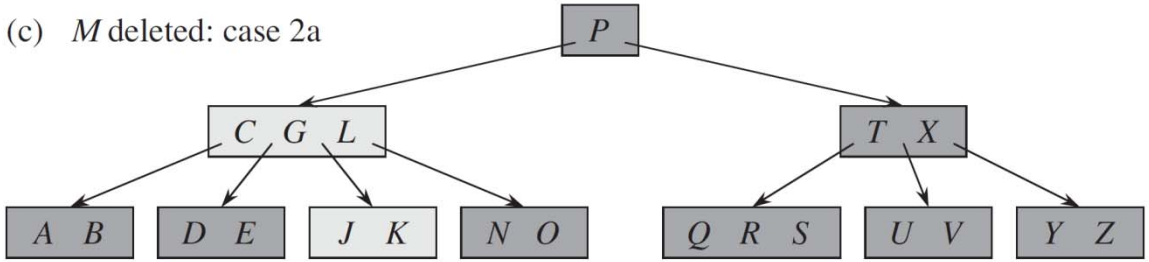


(b) *F* deleted: case 1

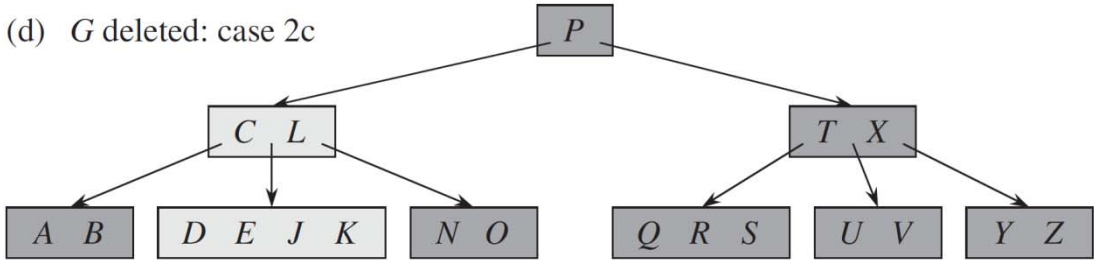


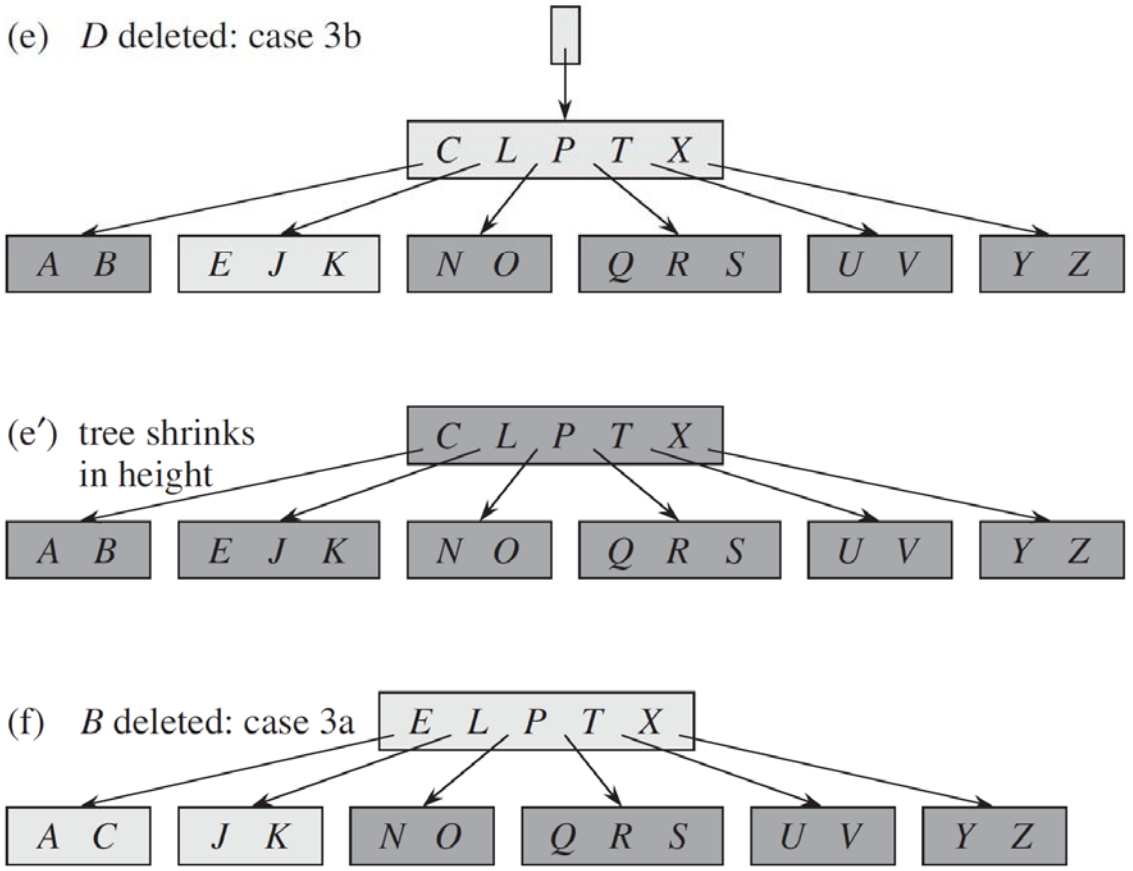
$t = 3$

(c) *M* deleted: case 2a



(d) *G* deleted: case 2c




(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

$t = 3$

Time complexity

- Only $O(h)$ disk operations for a B-tree of height h.
- Only $O(1)$ calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure.
- The CPU time required is $O(th) = O(t \log_t n)$.