# Algorithms
# Chapter 18
# B-Trees

①可以控制高度的搜尋樹
②配合磁碟的特性做最佳化

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

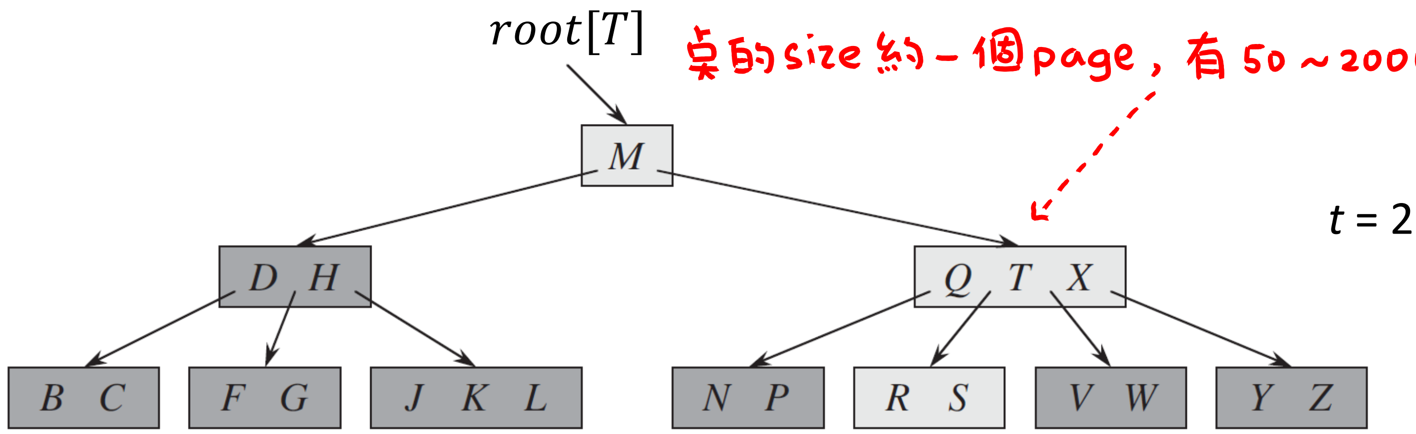
- **Definition of B-trees** B-tree 的性质
- Basic operations on B-trees search + insert
- Deleting a key from a B-tree delete

紅黑樹：高度 O(lgn)
insert, delete, search 時間 O(lgn)
B-trees：高度 $O(\log_t n)$　孩子的個數 t ~ 2t
insert, delete, search 的時間 $O(t\log_t n)$

# Overview$_{1/2}$

- B-trees are balanced search trees designed to work well on magnetic disks. 配合磁碟的特性做最佳化
- B-trees are similar to red-black trees, but they are better at minimizing disk I/O operations.
- **Red-black trees** = searchs + balanced
  - 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** = search tree + balanced + magnetic disks
  - 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$_{2/2}$




桌的size約一個page，有50～2000個小孩

找R時需拜訪的桌

- 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.

  有n[x]個keys ⇒ 有n[x]+1個兒子

重點1 ⇒ 非 root，key 的個數要在 $t-1 \le n[x] \le 2t-1$ 之間
是 root，key 的個數要在 $1 \le n[x] \le 2t-1$ 之間

# 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.

n[x] = key 值的個數

小 $\longrightarrow$ 大

  - 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

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

重點2 ⇒ 所有 leaves 有相同深度

  - 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. 最簡單的 B-tree，当 t=2 時
- Every internal node then has either 2, 3, or 4 children, and we have a **2-3-4 tree**. t=2 時，只可能有 2~4 個兒子

  ⇒ 2, 3, 4 樹

樹高：$O(\log_t n)$

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

$\log_t n = \frac{\lg n}{\lg t}$，樹高較紅黑樹小 $\lg t$ 倍

- **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.

第0層

第1~h層 每個點至少有 t-1 個鍵值

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

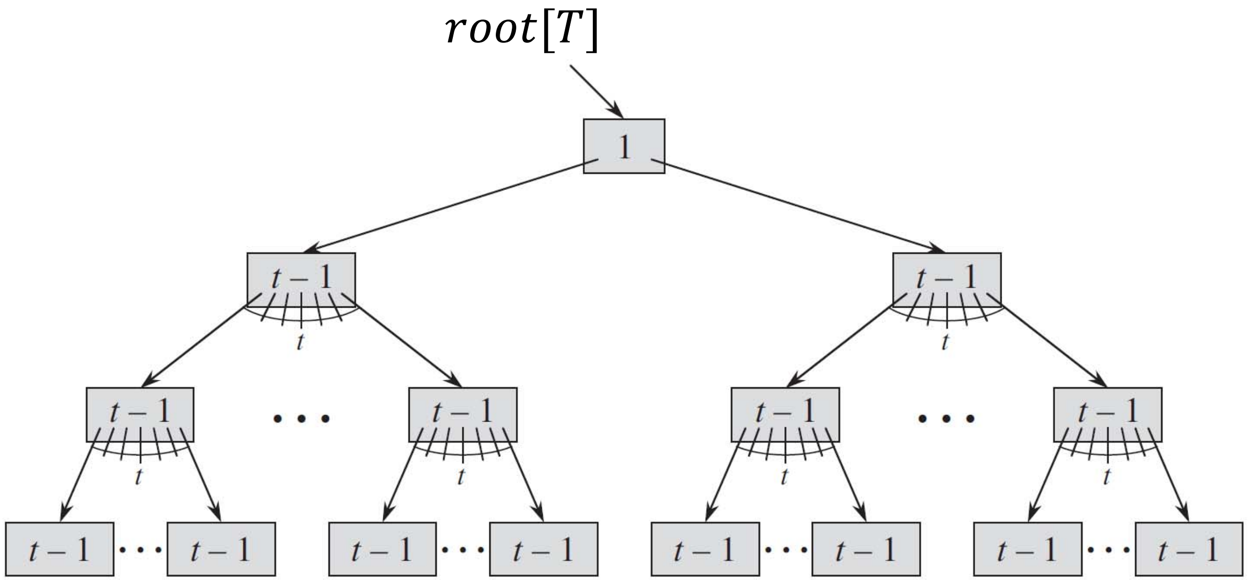
$$= 1 + 2(t-1)\left(\frac{t^h - 1}{t - 1}\right)$$

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

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

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

想法: 樹要高
⇒ 每一層的點數愈少愈好
⇒ 每一個點的 key 值愈少愈好




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

第0層: 是 root，只有一點，最少有一個 key 值

第1層: 因為 root 只有一個 key 值至少有 2 點，

第2層: 第1層每一個點至少有 t-1 個 key 值，
也就是 t 個兒子，故至少有 2t 個點

# Outline

- Definition of B-trees
- **Basic operations on B-trees** 搜尋與新增
- Deleting a key from a B-tree

# Searching a B-tree 和binary tree search 相似，只是要比 n[x] 個鍵值

- **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($x$, $k$) 小 → 大

$i \leftarrow 1$

**while** $i \leq n[x]$ and $k > key_i[x]$ 由左到右一個一個比

  $i \leftarrow i + 1$

**if** $i \leq n[x]$ and $k = key_i[x]$ 相同：找到了

  **return** ($x$, $i$)

**elseif** $leaf[x]$ leaf：不存在

  **return** NIL

**else** DISK-READ($c_i[x]$) 往子樹，繼續找

  **return** B-TREE-SEARCH($c_i[x]$, $k$)

時間：$O(t \log_t n)$

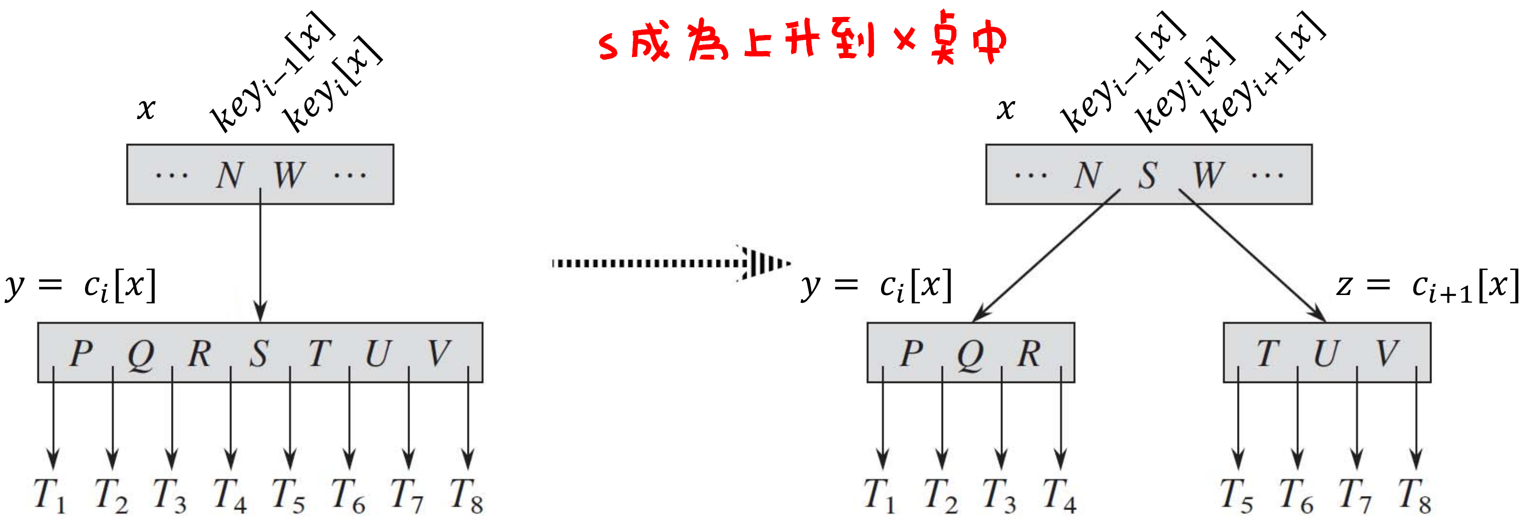
- Since $n[x] < 2t$, the running is $O(th) = O(t\log_t n)$.

insert 的过程中可能需做 splitting 的动作

將一個有 2t-1 個鍵值的点分割成 2 個有 t-1 個鍵值的点

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

- A fundamental operation used during insertion is the **splitting** a full node *y* (having 2*t* – 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.

如果 y 沒有父親，則 B-tree 的高度加 1

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

```
B-TREE-SPLIT-CHILD(x, i, y)
1.      z ← ALLOCATE-NODE()
2.      leaf[z] ← leaf[y]
3.      n[z] ← t – 1
4.      for j ← 1 to t – 1
5.           key_j[z] ← key_{j+t}[y]
6.      if not leaf [y]
7.           for j ← 1 to t
8.                c_j[z] ← c_{j+t}[y]
9.      n[y] ← t – 1
10.     for j ← n[x] + 1 downto i + 1
11.          c_{j+1}[x] ← c_j[x]
12.     c_{i+1}[x] ← z
13.     for j ← n[x] downto i
14.          key_{j+1}[x] ← key_j[x]
15.     key_i[x] ← key_t[y]
16.     n[x] ← n[x] + 1
17.     DISK-WRITE(y)
18.     DISK-WRITE(z)
19.     DISK-WRITE(x)
```

y是x的第i個兒子

Lines 1–3: $\Theta(1)$ 1~3 建立z点

Lines 4–14: $\Theta(t)$

4~8 建立z的鍵值和兒子

9 設定y有t-1個兒子

10~14 因為y一分為二作調整

①將x的key和child指標右移

②放入x的第i個兒子

③設定x有n[x]+1個兒子

Lines 15–19: $\Theta(1)$

Time : $O(t)$

Binary search tree: 從 root 往下找一個合適的父親
B-tree: 作法相同, 但保證尋找過程中的点都还沒滿

# 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. 使用 splitting 保證往下的过程中都不会滿

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

先檢查 root 有沒有滿
如果有, 將 root 一分為二

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

已經確認 root 沒滿

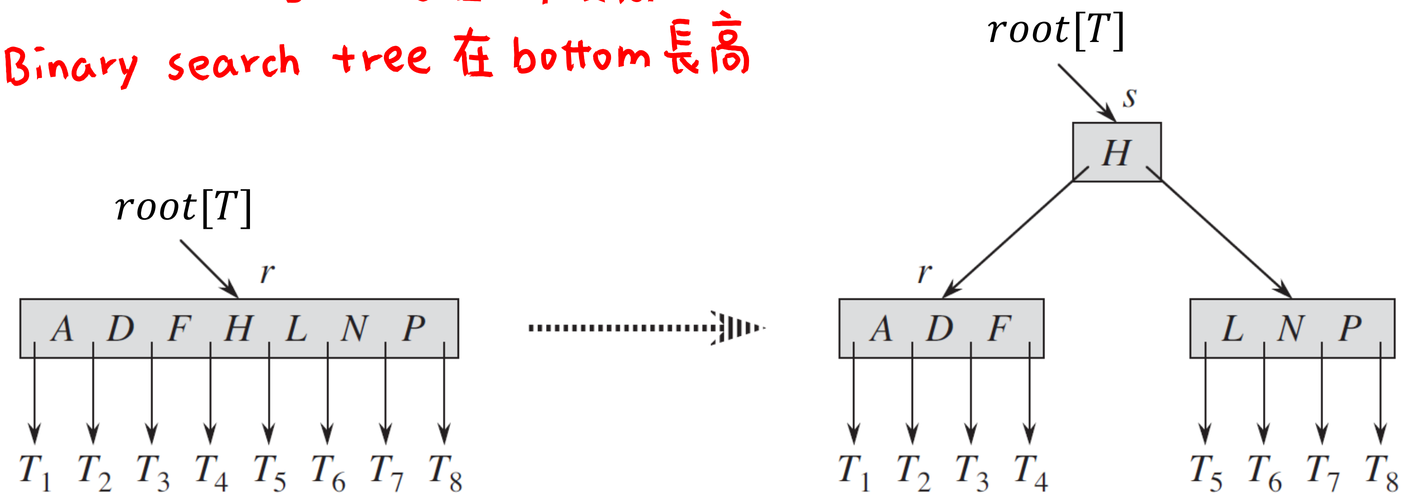
- The CPU time required is $O(th) = O(t\log_t n)$. 時間: $O(t\log_t n)$

# 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.

B-tree 在 top 長高

Binary search tree 在 bottom 長高




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.

如果 root 分割，則 B-tree 的高度加1

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

已知x沒有滿

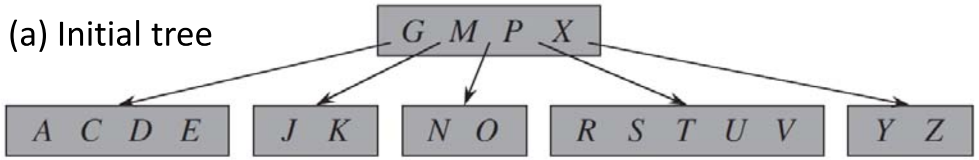
小 ← 大

leaf: 從右往左比，找合適的地方放

internal: 從右往左比，找合適的子樹，在往下之前先檢查兒子有沒有滿

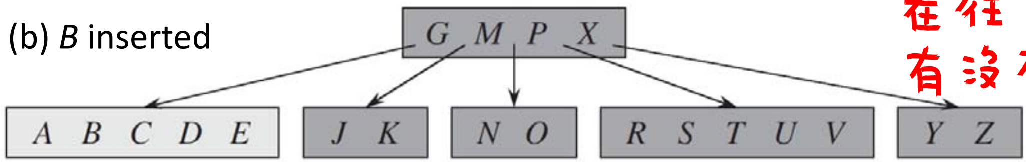
Time : $O(th)$
$= O(t\log_t n)$.

(a) Initial tree




internal：從右往左比，找合適的子樹，
在往下之前先檢查兒子
有沒有滿

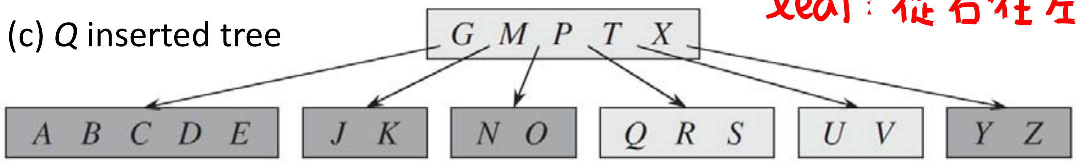
(b) *B* inserted



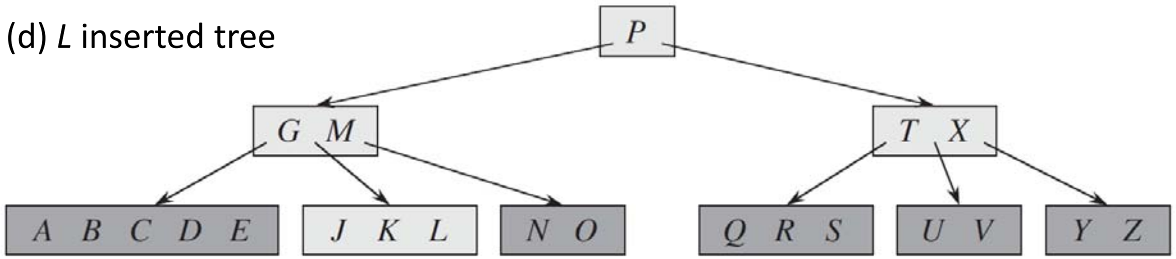

$t = 3$

leaf：從右往左比，找合適的地方放

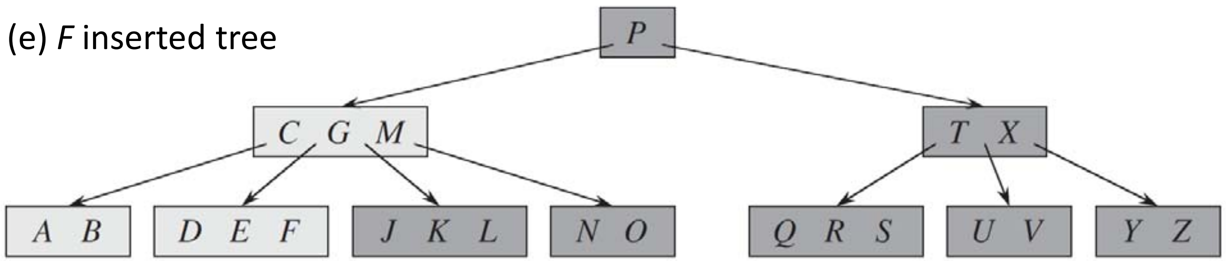
(c) *Q* inserted tree




(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** 刪除一個鍵值

從 root 開始往下拜訪，拜訪的過程中保證拜訪点至少有t個 keys

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

⇒ 如此可保證只需由上往下一次

- **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".

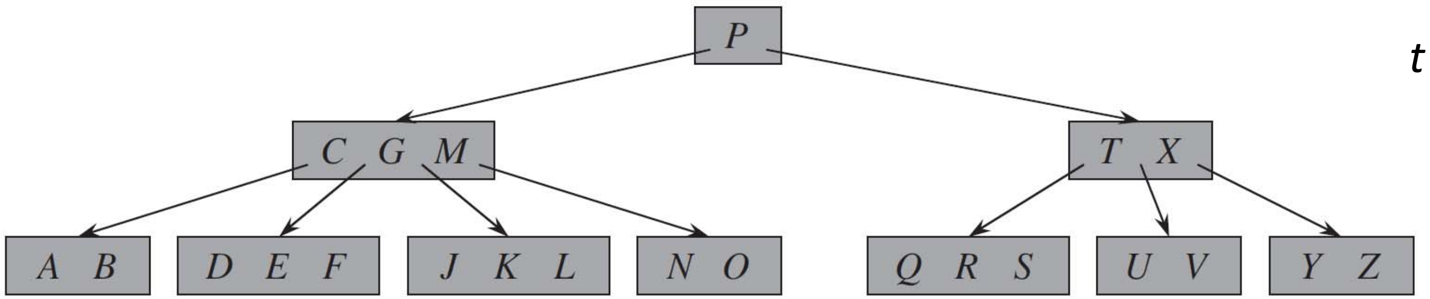
如果 root 沒有 contain 任何鍵值 ⇒ 將 root 刪掉 ⇒ 樹的高度少 1

- 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.

x 是 leaf, k 在 node x ①
x 是 internal node
- k 在 node x ②
- k 不在 node x ③

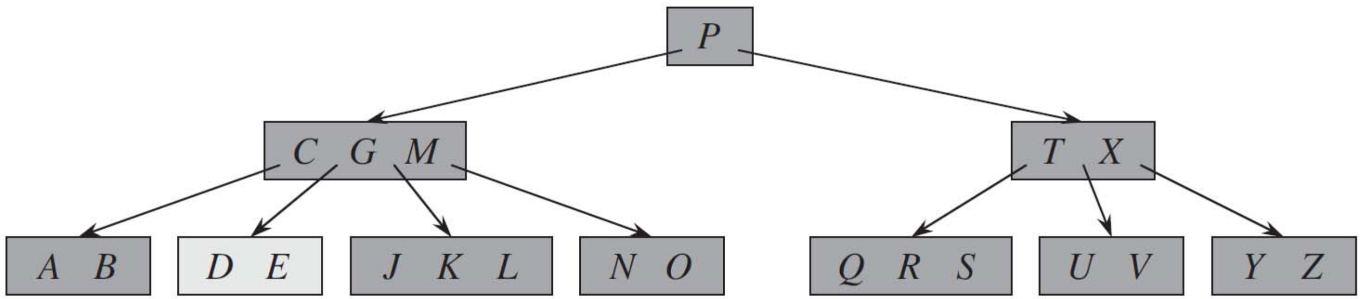
# Case 1: *k* is in node *x* and *x* is a leaf

- Delete the key *k* from *x*.




$t = 3$

Case 1: *F* deleted. x 是 leaf, k 在 node x → 直接刪



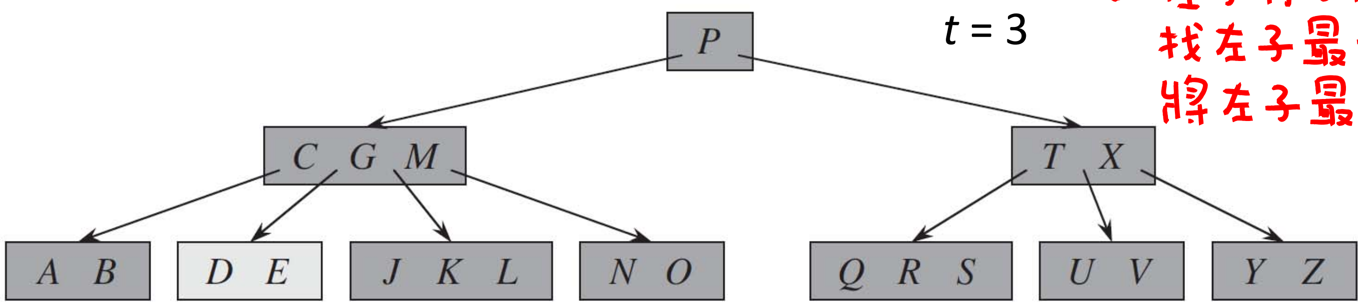
Binary search tree : x 有2子 ⇒ ①左子找最大
or ②右子找最小 來replace

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

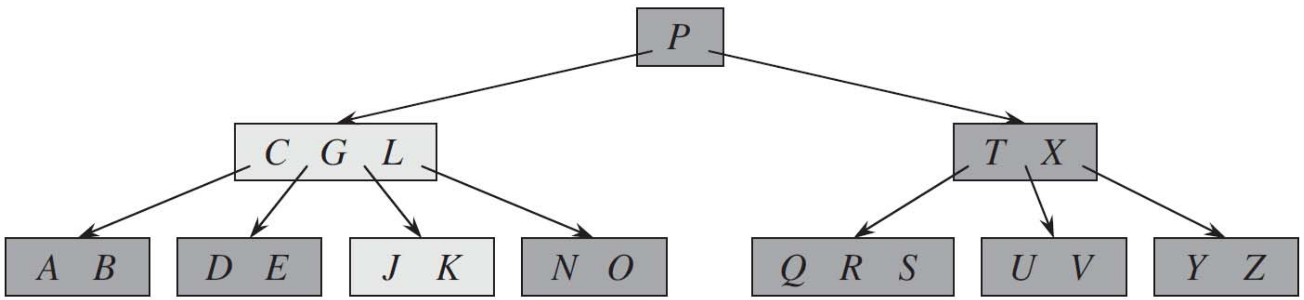
- Case 2a: the child *y* that precedes *k* has at least *t* keys.
  - Find the predecessor *k*′ of *k* in the subtree rooted at *y*.
  - Recursively delete *k*′, and replace *k* by *k*′ in *x*.

雖然有t個,但不可以直接刪

① 左子有t個鍵值,
找左子最大來replace
將左子最大刪除




Case 2a: *M* deleted.



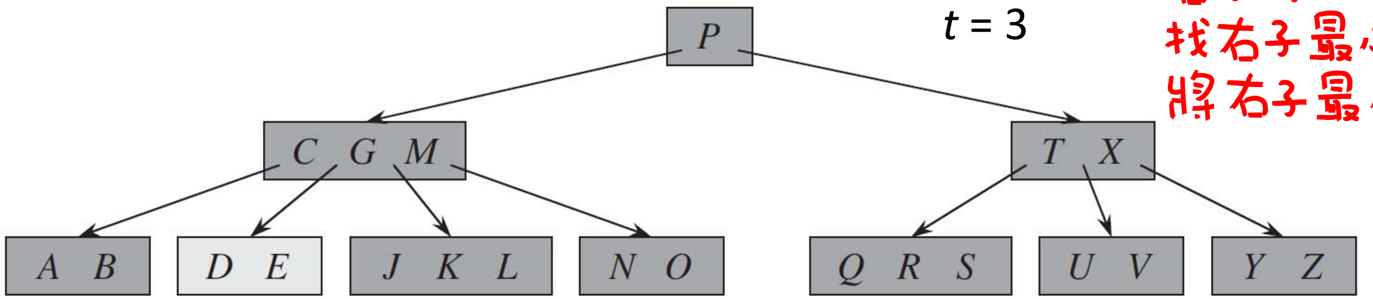
# Case 2: *k* is in node *x* and *x* is an internal node$_{2/3}$

- 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*.

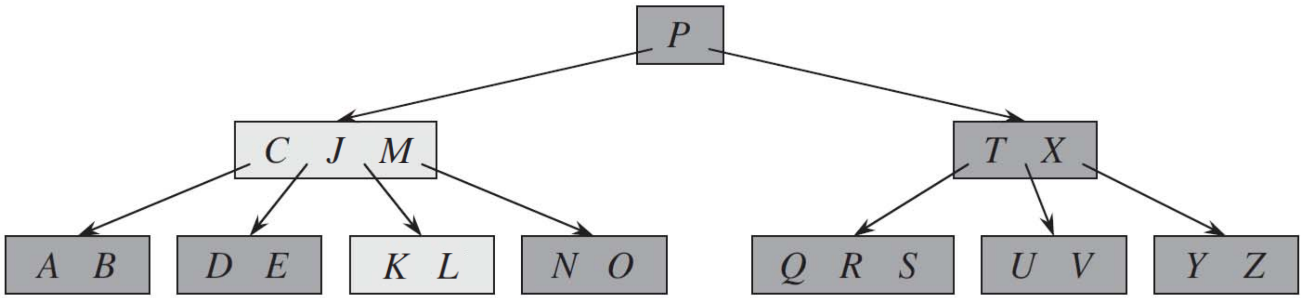
雖然有t個,但不可以直接刪

② 右子有t個鍵值，
找右子最小來replace
將右子最小刪除

$t = 3$




Case 2a: *G* deleted.

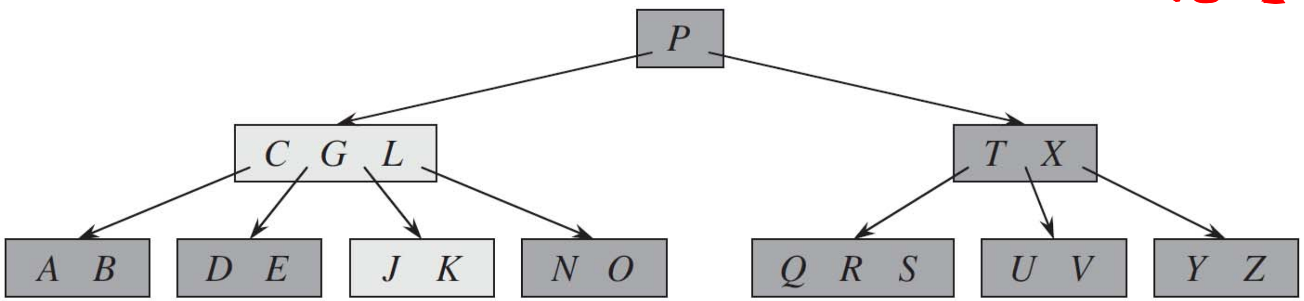


# Case 2: *k* is in node *x* and *x* is an internal node$_{3/3}$

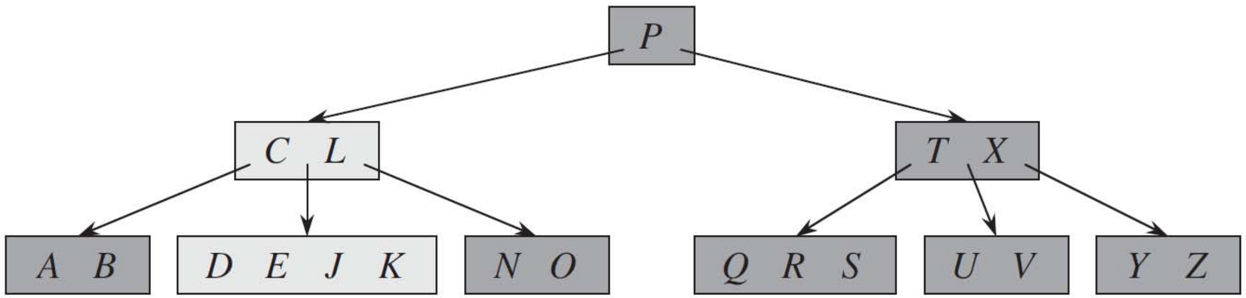
- Case 2c: both *y* and *z* have only $t - 1$ keys.
  - Merge *k* and all of *z* into *y*.
  - Free *z* and recursively delete *k* from *y*.

③ 右子，左子都只有 t-1 個鍵值
⇒ 將右子，左子，k 合併
⇒ 用遞迴將鍵值刪除
（已往下一層）

$t = 3$




Case 2c: *G* deleted.



k 不在 internal node x 中, k 在 Ci[x] 子樹中

# 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 $k$.

(I) Ci[x] 有 t 個
→ 用遞迴將 k 從 Ci[x] 子樹中刪除

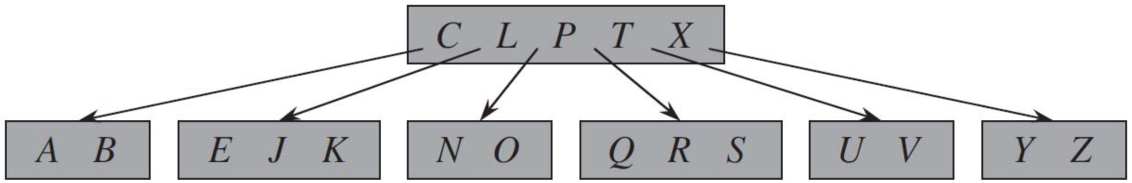
- Case 3a: $c_i[x]$ has at least $t$ keys
  - Recursively delete $k$ from $c_i[x]$.
- Case 3b: $c_i[x]$ has only $t-1$ keys but has an immediate sibling with at least $t$ keys.
- Case 3c: $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t-1$ keys.

(II) Ci[x] 只有 t-1 個
→ 兄有 t 個, 跟兄借
→ 弟有 t 個, 跟弟借

(III) Ci[x], 兄,弟都只有 t-1 個
→ 合併

# 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 $c_i[x]$.

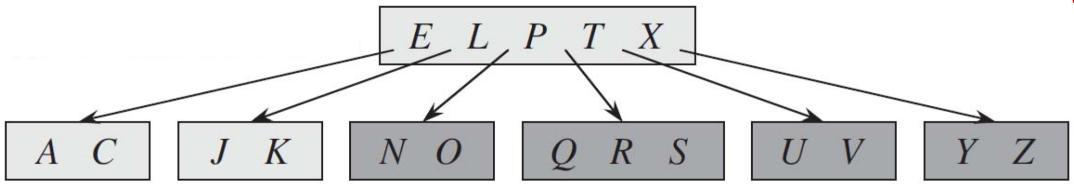



$t = 3$

Case 3b：$B$ deleted.

① 兄有t個，跟兄借

② 弟有t個，跟弟借

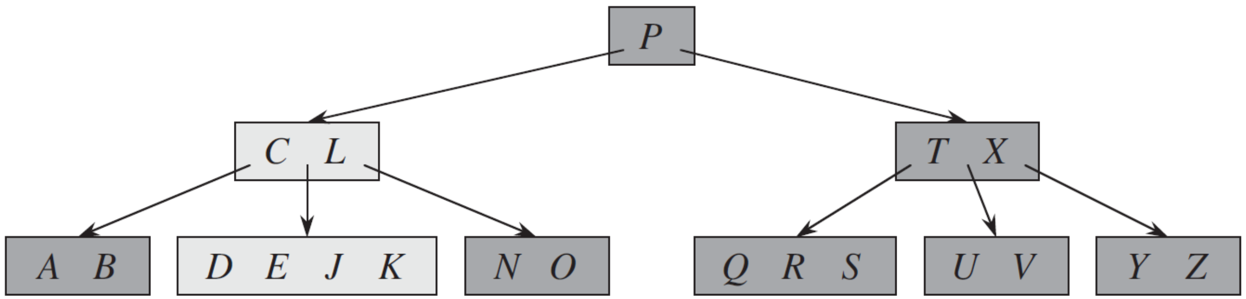


# Case 3: $k$ is not present in internal node $x_{3/3}$

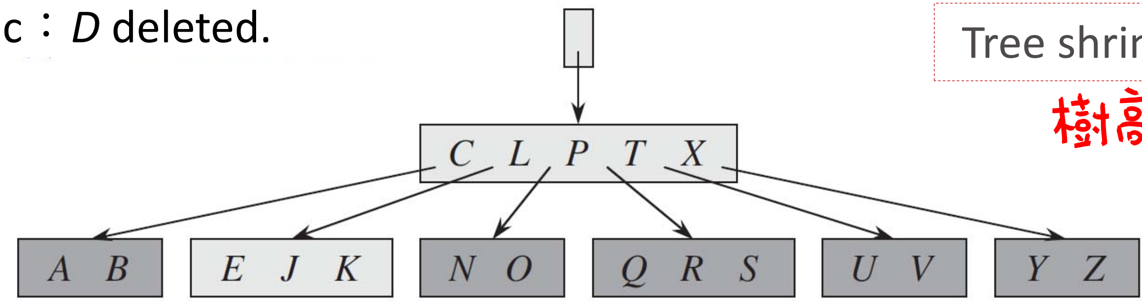
- Case 3c: $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t-1$ keys.
  - Merge $c_i[x]$ with one sibling.
  - Moving a key from $x$ down into the new merged node to become the median key for that node.

③ 兄弟都只有 t-1 個，合併

$t = 3$

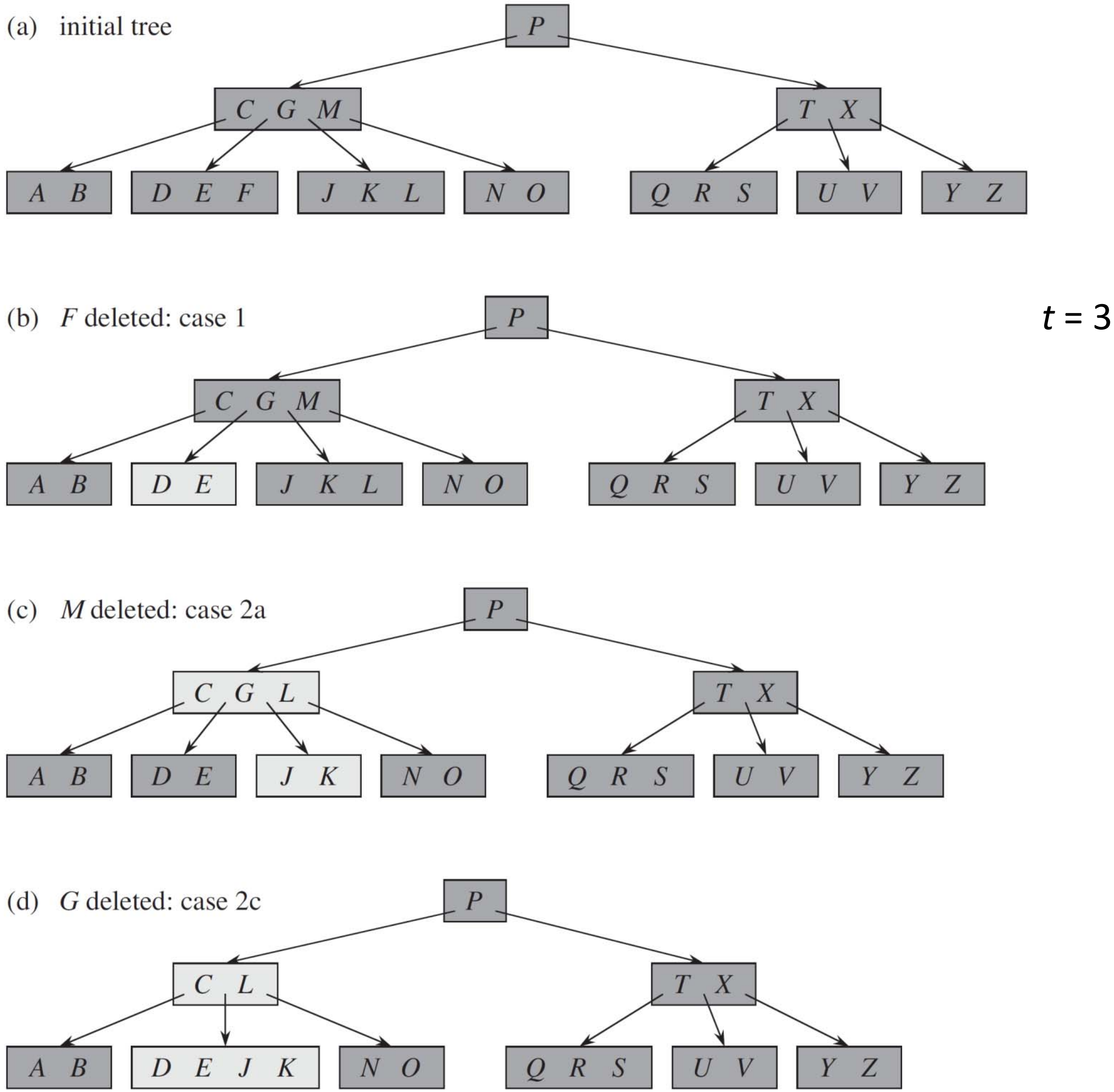


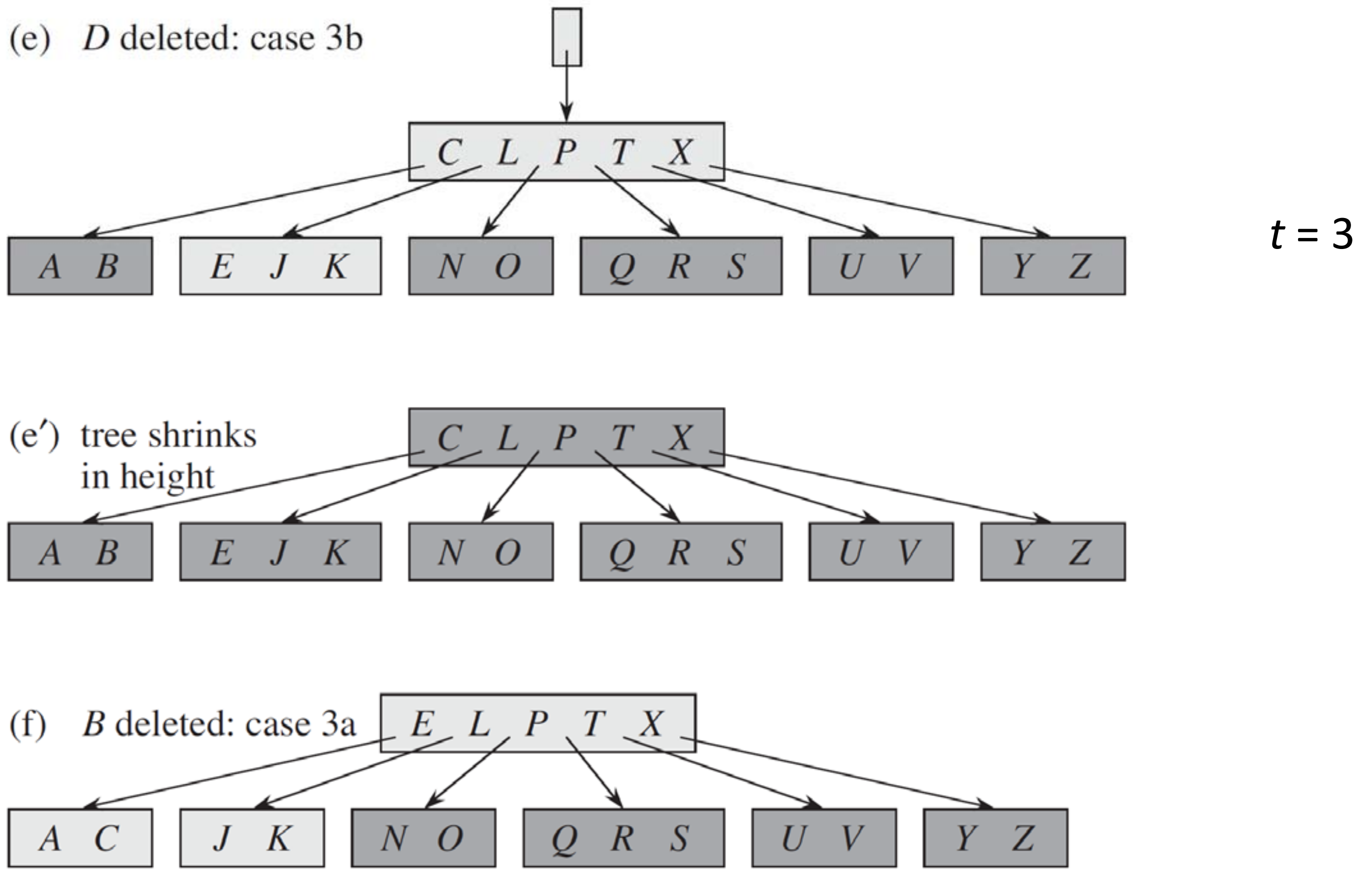

Case 3c：$D$ deleted.




Tree shrinks in height.

樹高減1


(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
t = 3
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


(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.

  遞迴程式只有O(1)的DISK-READ和DISK-WRITE
- The CPU time required is $O(th) = O(t\log_t n)$.