B-Trees

Introduction

- Similar to Red-Black tree or other balanced search trees.
- Different from other balanced search tree, nodes may have many children. Data is stored in a disk. So mainly used for Disk I/O.
- Provides index structures for large amount of data.

Introduction – contd.

- Up to now, all data that has been stored in the tree has been in memory.
- If data gets too big for main memory, what do we do?
- All data cannot be resident in the main memory.
- If we keep a pointer to the tree in main memory, we could bring in just the nodes that we need.
- For instance, to do an insert with a BST, if we need the left child, we do a disk access and retrieve the left child.
- If the left child is NIL, then we can do the insert, and store the child node on the disk.
- Not too good for a BST

Introduction – contd.

- The problem with BST: storing the data requires disk accesses, which is expensive, compared to execution of machine instructions.
- If we can reduce the number of disk accesses, then the procedures run faster.
- The only way to reduce the number of disk accesses is to increase the number of keys in a node.
- The BST allows only one key per leaf.
- Very good and often used for Search Engines!
  - (when collection size gets very big → the index does not fit in memory)

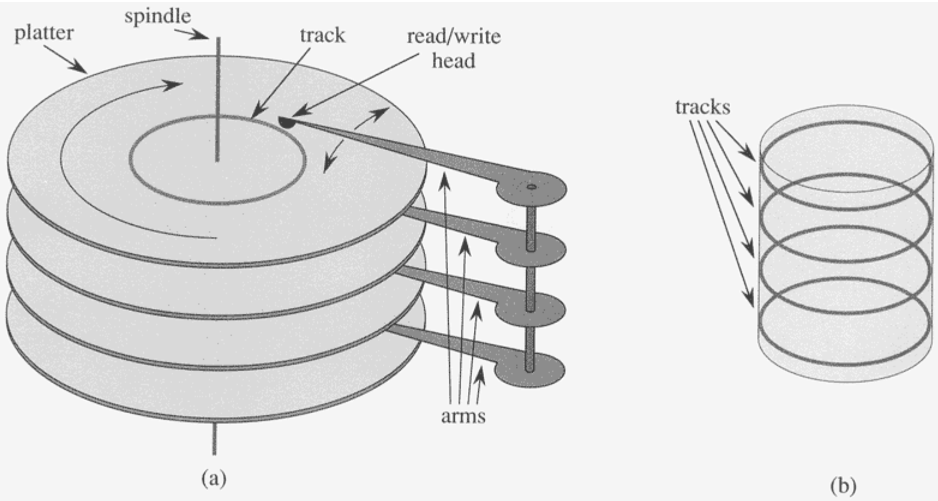
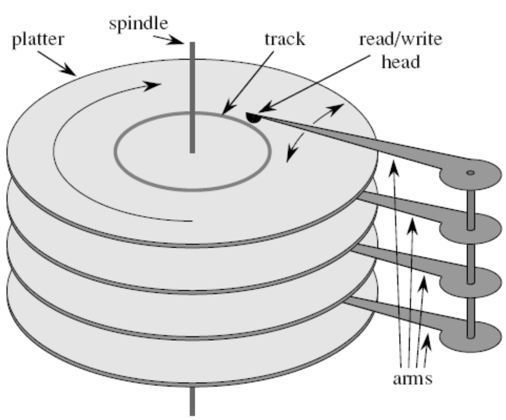


Figure 18.2 **(a)** A typical disk drive. It is composed of several platters that rotate around a spindle. Each platter is read and written with a head at the end of an arm. The arms are ganged together so that they move their heads in unison. Here, the arms rotate around a common pivot axis. A track is the surface that passes beneath the read/write head when it is stationary. **(b)** A cylinder consists of a set of covertical tracks.

Disk Operations

- In a disk, data is organized in disk pages.
- To read data in a disk,
  - The disk rotates, and the R/W head moves t the page containing th data.
  - Then, the whole disk page is read.



From: Cormen, T., C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 2001.

An example of B Tree

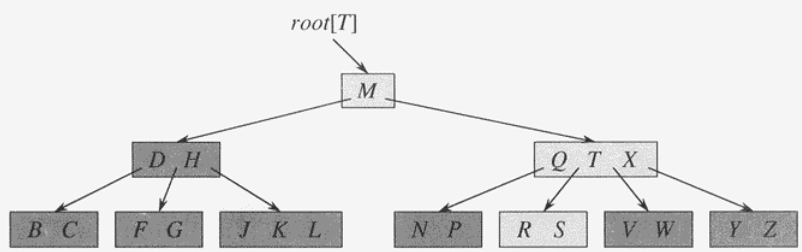
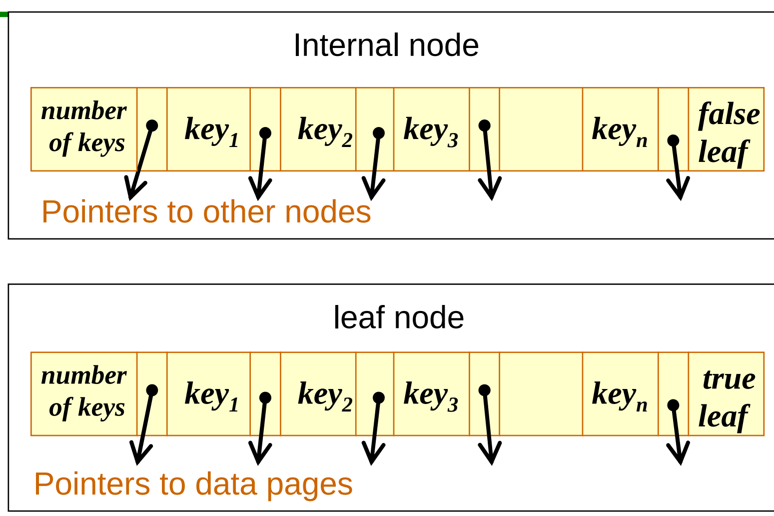


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $n[x]$ keys has $n[x]+1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

B-Trees: Nodes



B-Trees: Nodes – contd.

A B-Tree is a rooted tree (whose root is root[T])) having the following properties:

1. Every internal node x has the following fields:

n[x]	key_1[x]	key_2[x]	key_3[x]	key_4[x]	key_5[x]	key_6[x]	key_7[x]	key_8[x]	leaf[x]

c_1[x]	c_2[x]	c_3[x]	c_4[x]	c_5[x]	c_6[x]	c_7[x]	c_8[x]	c_9[x]

n[x] is the number of keys in the node. n[x] = 8 above.

leaf[x] = false for internal nodes, since x is not a leaf.

The key_i[x] are the values of the keys, where $key_i[x] \leq key_{i+1}[x]$.

c_i[x] are pointers to child nodes. All the keys in c_i[x] have values that are between key_{i-1}[x] and key_i[x].

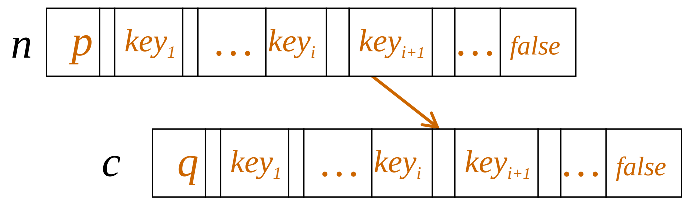
B-Trees: Nodes – contd.

- Leaf nodes have no child pointers
- leaf[x] = true for leaf nodes.
- All leaf nodes are at the same level

n[x]	key_1[x]	key_2[x]	key_3[x]	key_4[x]	key_5[x]	key_6[x]	key_7[x]	key_8[x]	leaf[x]

Properties of B-Trees - contd.

For any node *n* in a B-tree *T*



- $n.key_i \le c.key_1 \le c.key_2 \le \ldots \le c.key_q \le n.key_{i+1}$
- If *n* is a leaf node, the depth of *n* is *h*, where *h* is the tree's height.

Properties of B-Trees contd.

- Root must have 1 key .
- The ***minimum degree, t,*** of the B-tree is the lower bound on the number of keys a node can contain. ($t \ge 2$)
- 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.
- Every node can contain at most $2t - 1$ keys. Therefore, an internal node can have at most $2t$ children.
- We say that a node is ***full*** if it contains exactly $2t - 1$ keys.

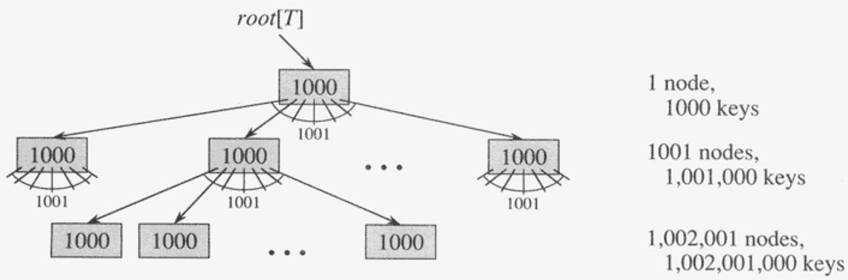


Figure 18.3 A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is $n[x]$, the number of keys in x.

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

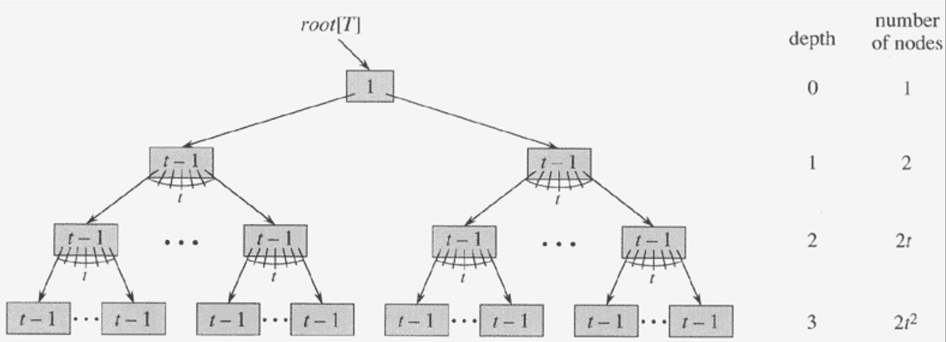


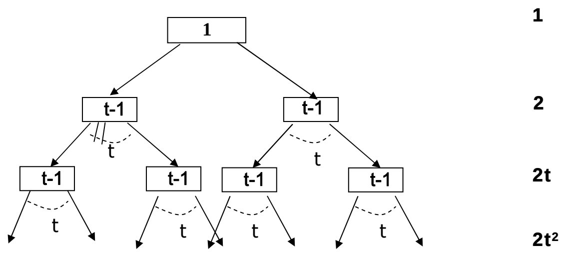
Figure 18.4 A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is $n[x]$.

- Thm ：

  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：



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

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

$$\frac{n+1}{2} \geq t^h. \qquad \log_t \frac{n+1}{2} \geq h.$$

Height of B-Tree

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

  height = $h \leq \log_t[(n+1)/2]$
- The important thing to notice is that the height of the tree is log base t. So, as t increases, the height, for any number of nodes n, will decrease.
- Using the formula $\log_a x = (\log_b x)/(\log_b a)$, we can see that
  - $\log_2 10^6 = (\log_{10} 10^6)/(\log_{10} 2) \approx 6/0.30102999566398 \approx 19$
  - $\log_{10} 10^6 = 6$

  So, 13 less disk accesses to get to the leafs!

B-tree of degree t

- Insertion and deletions:
  - More complicate but still log (n)
  - Split and merge operation.

- Convention：
  - Root of the B-tree is always in main memory.
  - Any nodes that are passed as parameters must already have

  had a DISK_READ operation performed on them.
- Operations：
  - Searching a B-Tree.
  - Creating an empty B-tree.
  - Splitting a node in a B-tree.
  - Inserting a key into a B-tree.
  - Deleting a key from a B-tree.

B-Tree-Search(x,k) :

- Algorithm :

B-Tree-Search(x,k)

$\{ \quad i \leftarrow \mathbf{1}$

$\quad$ while $i \le n[x]$ and $k > key_i[x]$

$\quad\quad$ do $i \leftarrow i + \mathbf{1}$

$\quad$ *if* $i \le n[x]$ and $k = key_i[x]$

$\quad\quad$ then return(x, i)

$\quad$ *if* $leaf[x]$ then return NULL

$\quad\quad\quad$ else DISK - READ$(C_i[x])$

$\quad\quad\quad\quad$ return B - Tree - Search$(C_i[x], k)$

$\}$

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

B-Tree-Created(T) :

- Algorithm :

B-Tree-Create(T)

$\{ \quad x \leftarrow$ Allocate – Node()

$\quad$ Leaf$[x] \leftarrow$ TRUE

$\quad$ n[x] $\leftarrow$ 0

$\quad$ DISK - WRITE(x)

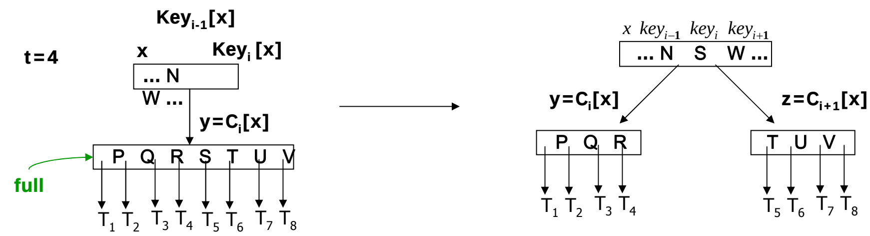
$\quad$ root[T] $\leftarrow$ x

$\}$

$O(\mathbf{1})$

- time :

- B-Tree-Split-Child(x,i,y)：
  - Splitting a node in a B-Tree：



Splitting a full node y (have 2t-1 keys) around its median key $key_t[y]$ into 2 nodes having (t-1) keys each.

- Algorithm：

B-Tree-Split-Child(x,i,y)

{ $z \leftarrow \text{Allocate-Node}()$

$\text{leaf}[z] \leftarrow \text{leaf}[y]$

$n[z] \leftarrow t - \mathbf{1}$

$\textit{for}\ \ j \leftarrow \mathbf{1}\ \ \textit{to}\ \ t\text{-}1\ \ \textit{do}\ \ \text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$

$\textit{if}\ \ \text{not leaf}[y]\ \ \textit{then}$

$\quad \textit{for}\ \ j \leftarrow \mathbf{1}\ \ \textit{to}\ \ t\ \ \textit{do}\ \ C_j[z] \leftarrow C_{j+t}[y]$

$n[y] \leftarrow t - \mathbf{1}$

$\textit{for}\ \ j \leftarrow n[x] + \mathbf{1}\ \ \textit{downto}\ \ i+1\ \ \textit{do}\ \ C_{j+1}[x] \leftarrow C_j[x]$

$C_{j+1}[x] \leftarrow z$

$\textit{for}\ \ j \leftarrow n[x]\ \ \textit{downto}\ \ i\ \ \textit{do}\ \ \text{Key}_{j+1}[x] \leftarrow Key_j[x]$

$\text{Key}_i[x] \leftarrow \text{Key}_t[y]$

$n[x] \leftarrow n[x] + \mathbf{1}$

DISK - WRITE(y)

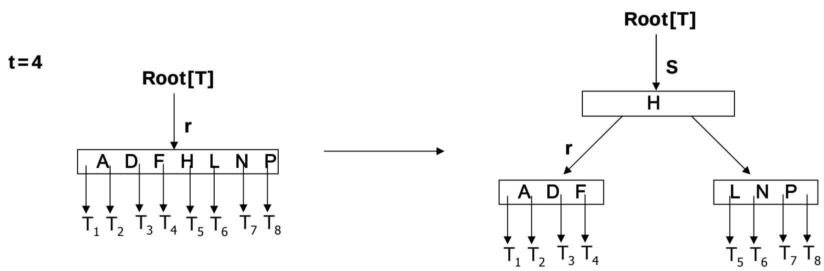
DISK - WRITE(z)

DISK - WRITE(x)

}

B-Tree-Insert(T,k)：

- Insert a key in a B-Tree：



- Algorithm：

B-Tree-Insert(T,k)

{ $\quad$ $r \leftarrow root[T]$

$\quad$ *if* $n[r] = 2t - 1$ *then*

$\quad$ { $\quad$ $S \leftarrow$ Allocate - Node()

$\quad\quad$ $root[T] \leftarrow S$

$\quad\quad$ $leaf[S] \leftarrow$ FALSE

$\quad\quad$ $n[S] \leftarrow \mathbf{0}$

$\quad\quad$ $C_i[S] \leftarrow r$

$\quad\quad$ B-Tree-Split-Child(S,l,r)

$\quad\quad$ B-Tree-Insert-Nonfull(S,k)

$\quad$ }

$\quad$ *else* B - Tree - Insert - Nonfull(r,k)

}

B-Tree-Insert-Nonfull(x,k) ：

Algorithm ：

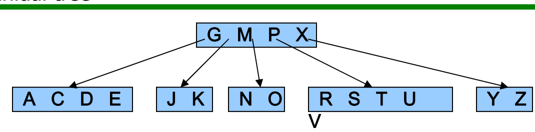
B-Tree-Insert-Nonfull(x,k)

```
{     i ← n[x]
      if leaf[x] then
      { while i ≥ 1 and k < key_i[x]
            do { 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]
            do i ← i - 1
        i ← i + 1
        DISK-READ(C_i[x])
        if n[C_i[x]] = 2t - 1
            then B-Tree-Split-Child(x,i,C_i[x])
                 if k > key_i[x] then i ← i + 1
        B-Tree-Insert-Nonfull(C_i[x],k) }
}
```

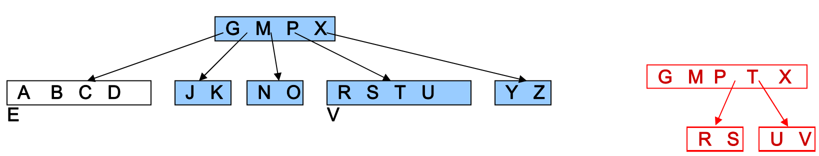
Example ： Inserting keys into a B-Tree.

t=3

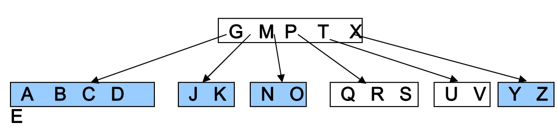
(a) Initial tree



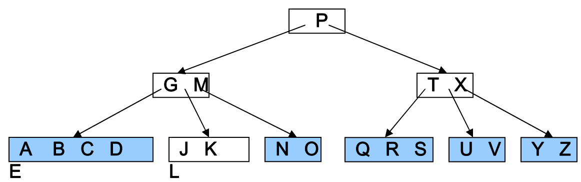
(b) B inserted



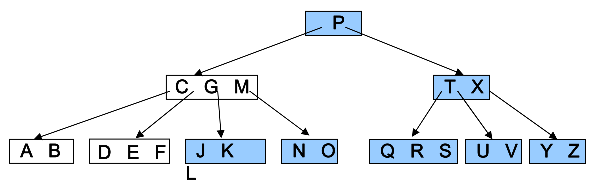
(c) Q inserted



(d) L insert



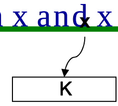
(e) F insert



Deleting a key from a B-Tree :

(x has ≥ t keys)

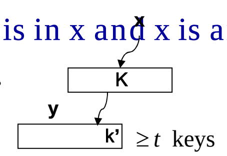
1. K is in x and x is a leaf :



delete k from x.

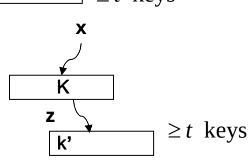
2. K is in x and x is an internal node :

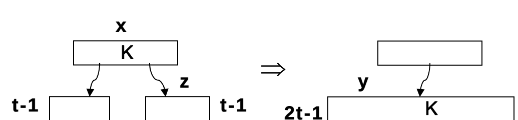
a.



Recursively delete k' and replace k by k' in x.

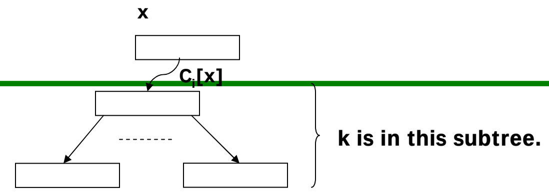
b.



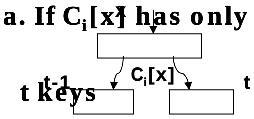


c. if both y,z has t-1 keys.

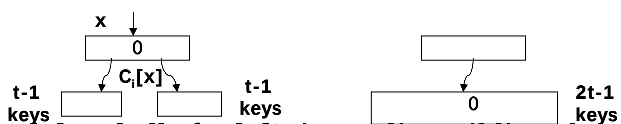
3. If K is not in internal node x :



a. If $C_i[x]$ has only t-1 keys but has an immediate sibling with t keys



- **Move a key from x down to $C_i[x]$.**
- **Move a key from $C_i[x]$'s immediate sibling to x.**
- **Move an appropriate child to $C_i[x]$ from its immediate sibling.**

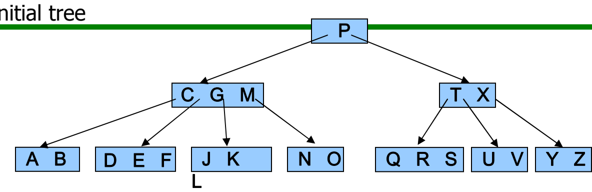


b. If $C_i[x]$ and all of $C_i[x]$'s immediate siblings have t-1 keys, merge $C_i[x]$ with one immediate sibling.

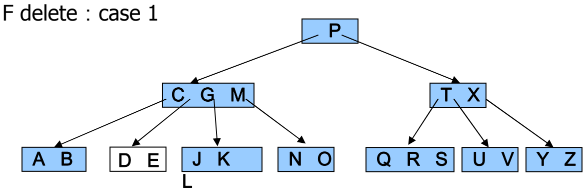
Example : Deleting a key from a B-Tree.

t=3

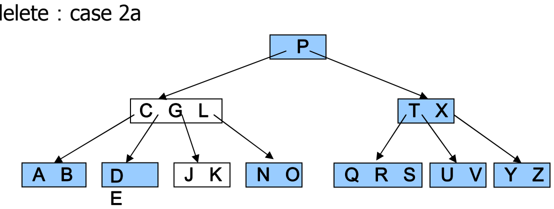
(a) Initial tree

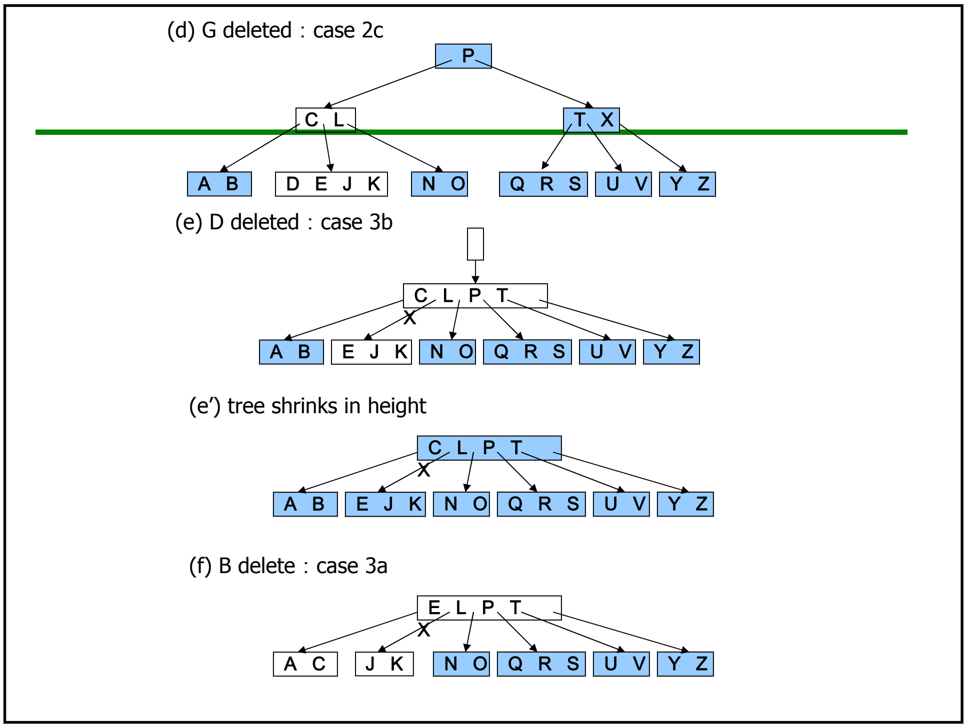


(b) F delete : case 1



(c) M delete : case 2a




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