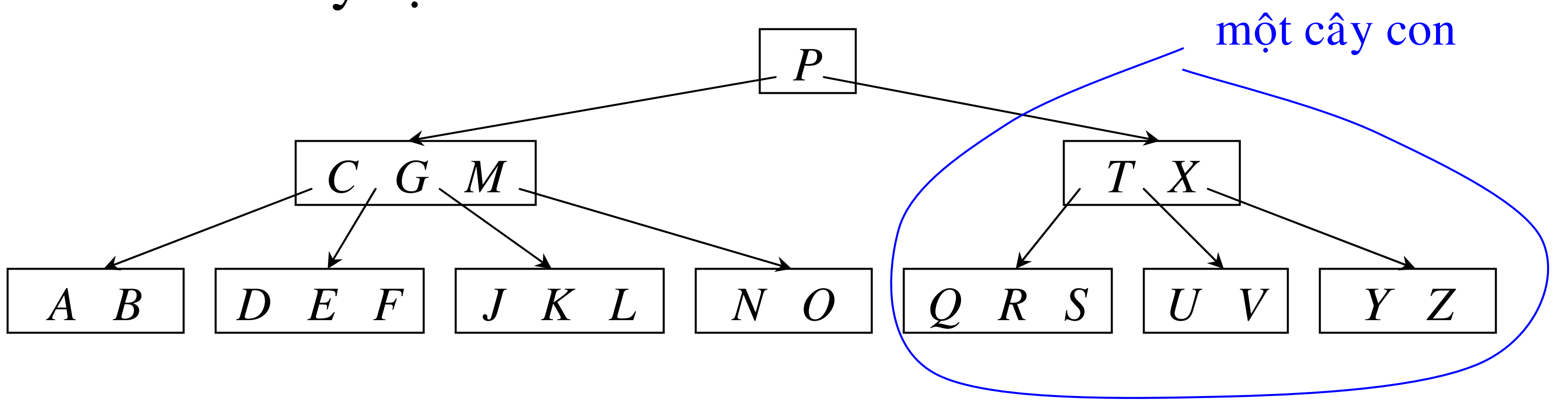
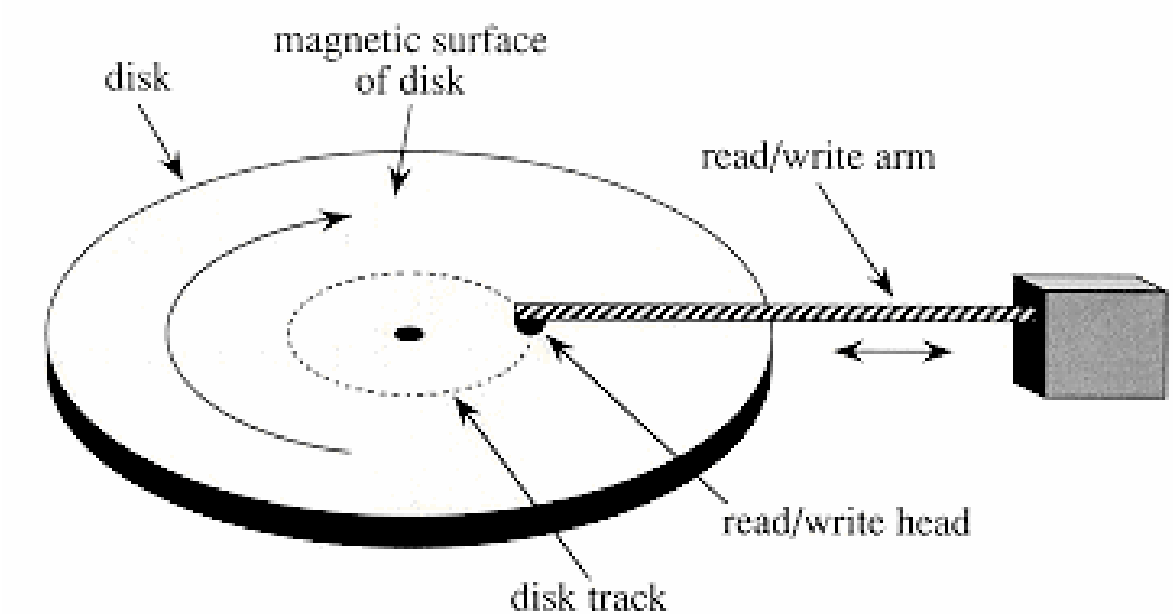
Lecture 6.1: B-Tree

- B-cây tổng quát hoá cây tìm kiếm nhị phân.
- Cho *x* là một nút trong của một B-cây, nút *x* có
  - *n*[*x*] khóa, chia vùng giá trị của các khóa thành *n*[*x*] + 1 vùng giá trị con
  - *n*[*x*] + 1 cây con, lưu trữ khóa trong vùng giá trị con
- Tìm khóa trong B-cây: tại nút *x* có thể cần đến *n*[*x*] phép so sánh
- Ví dụ một mô hình của một B-cây
  - Khóa: ký tự



Cấu trúc dữ liệu trong bộ nhớ ngoài

- Máy tính
  - Bộ nhớ chính (main memory)
  - Bộ nhớ ngoài (secondary storage)
    - Disk
      - Track
      - Cylinder
      - Page (hay sector): số lượng dữ liệu mỗi lần đọc hay viết đĩa là một số nguyên lần của disk page.



Cấu trúc dữ liệu trong bộ nhớ ngoài

- Giả thiết:
  - Số lượng dữ liệu rất lớn, không thể đọc hết cùng một lúc vào bộ nhớ chính
- B-cây là cây tìm kiếm cân bằng được thiết kế để làm việc hữu hiệu trong bộ nhớ ngoài:
  - Số các disk page trong bộ nhớ chính tại mọi thời điểm là hằng.
- Phân tích thời gian chạy của thao tác lên cấu trúc dữ liệu bộ nhớ ngoài, gồm:
  - số các truy cập vào đĩa
  - thời gian CPU

Truy cập đĩa

- Một nút của B-cây thường chiếm nguyên cả một disk page.
- Hệ số phân nhánh tùy thuộc vào tỉ lệ giữa kích thước của khóa và kích thước của disk page.

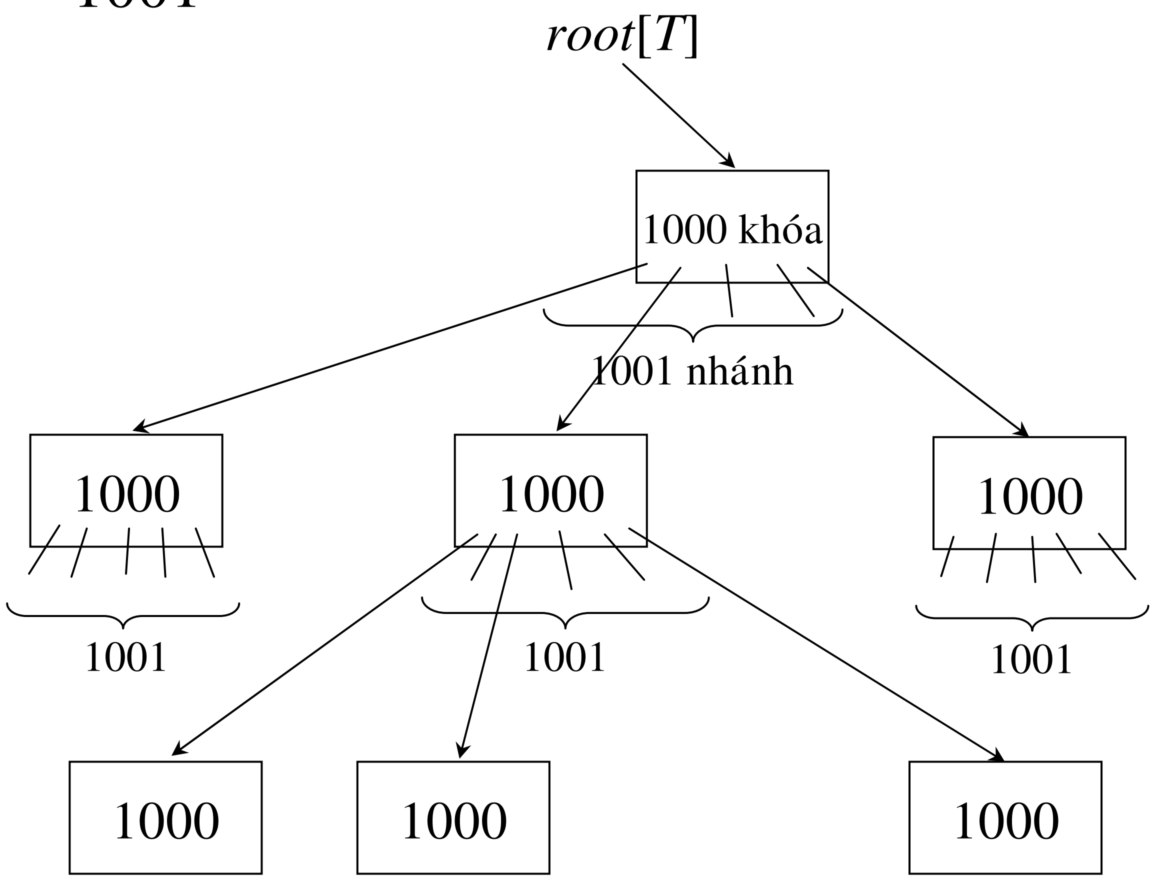
Các thao tác lên một đĩa

- Cho x là một con trỏ đến một đối tượng (ví dụ: một nút của một B-cây). Đối tượng x có thể có nhiều trường
  – Nếu x nằm trong bộ nhớ chính, truy cập các trường của x như thường lệ, ví dụ như *key*[x],...
  – Nếu x còn nằm trên đĩa thì dùng DISK-READ(x) để đọc nó vào bộ nhớ chính.
  – Nếu x đã thay đổi thì dùng DISK-WRITE(x) để trữ nó vào đĩa.
- Cách làm việc tiêu biểu với một đối tượng x

  ...
  $x \leftarrow$ một con trỏ đến một đối tượng nào đó
  DISK-READ(x)
  các thao tác truy cập/thay đổi các trường của x
  DISK-WRITE(x)
  các thao tác không thay đổi một trường của x
  ...

Hệ số phân nhánh

- Ví dụ một B-cây mà:
  - mỗi nút có 1000 khóa, tức là B-cây có hệ số phân nhánh là 1001



1 nút
1000 khóa

1001 nút
1.001.000 khóa

1.002.001 nút
1.002.001.000 khóa

Định nghĩa của B-cây

- Một B-cây T - gốc là $root[T]$, có các tính chất sau:
  - Mỗi nút x có các trường sau
    - $n[x]$, số lượng khóa đang được chứa trong nút x
    - các khóa: có $n[x]$ khóa, được xếp theo thứ tự không giảm, tức là

      $$key_1[x] \le key_2[x] \le \cdots \le key_{n[x]}[x]$$

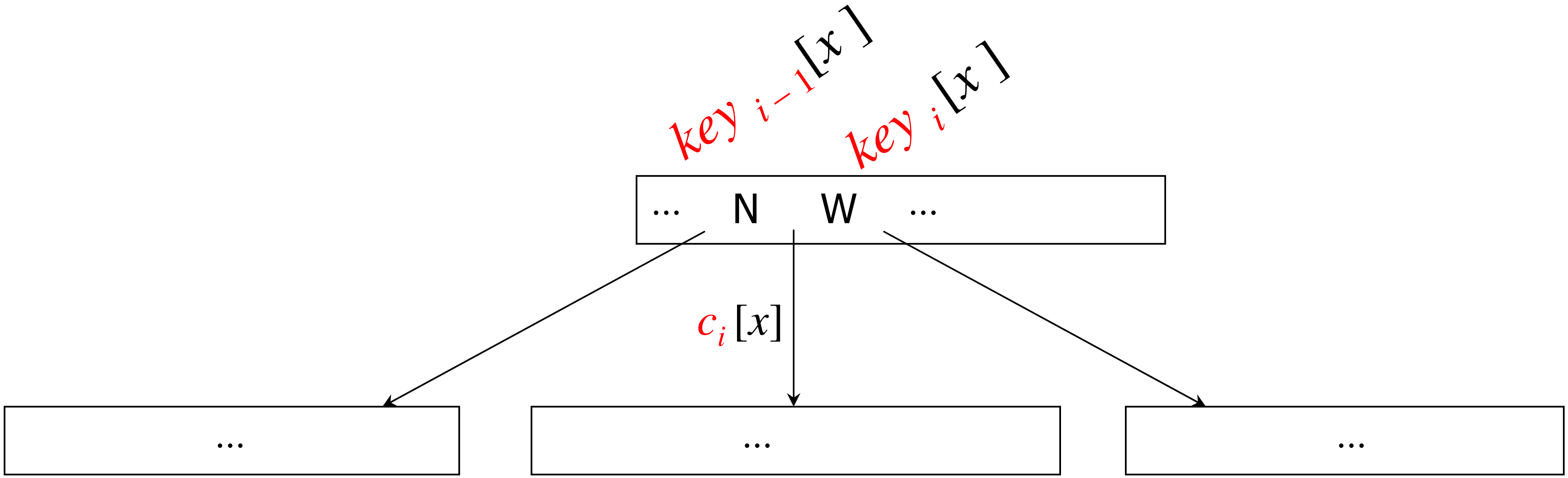
    - $leaf[x]$, có trị bool là

      TRUE nếu x là một lá

      FALSE nếu x là một nút trong
  - Mỗi nút trong chứa $n[x] + 1$ con trỏ $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ đến các nút con của nó.

Định nghĩa của B-cây

Mô hình một nút của B-cây

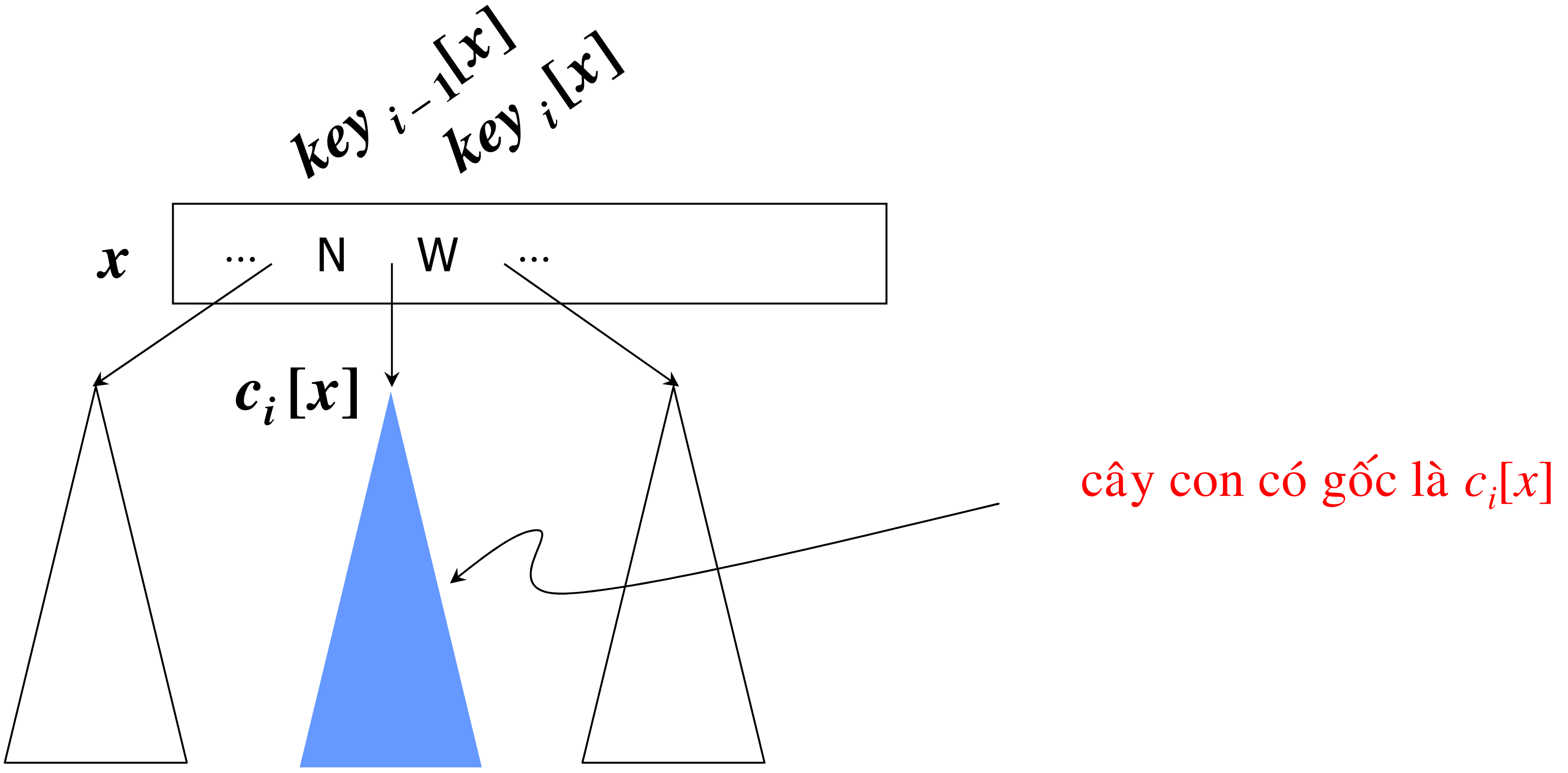


Định nghĩa của B-cây

(tiếp)

– Nếu k_i là một khóa bất kỳ trong cây con có gốc là $c_i[x]$ thì

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



Định nghĩa của B-cây

(tiếp)

- Tất cả các lá có cùng một *độ sâu*, bằng *chiều cao* h của cây
- Có một số nguyên $t \geq 2$ gọi là *bậc tối thiểu* của cây sao cho
  - Mọi nút không phải nút gốc phải có ít nhất $t - 1$ khóa. Nếu cây $\neq \varnothing$ thì nút gốc phải có ít nhất một khóa.
  - Nút có thể chứa tối đa $2t - 1$ khóa. Một nút là *đầy* nếu nó có đúng $2t - 1$ khóa.

k2 Một nút không phải là gốc có ít nhất t-1 khóa
Mỗi nút có tối đa:2t-1 khóa
khmt-hung, 11/26/2006

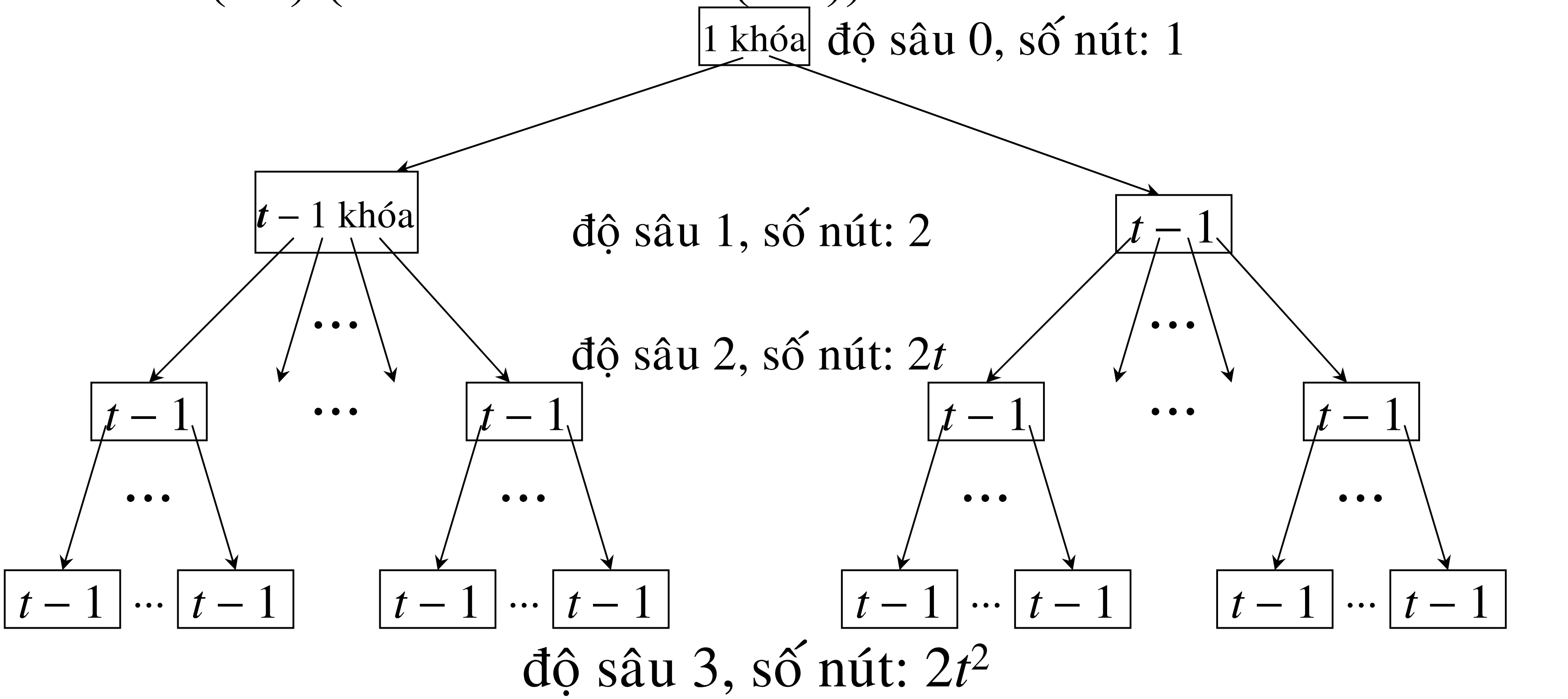
k1

Số khóa tối thiểu trong một B-cây

- B-Cây có bậc là t sẽ có số khóa ít nhất khi mọi nút đề chỉ có t-1 khóa, ngoại trừ nút gốc có đúng 1 khóa.
- Cây có chiều cao là h thì các nút sẽ nằm ở các mức từ 0 đến (h-1).
- Gọi số khóa tối thiểu của B-cây bậc t chiều cao h là n:

n >=1+2(t-1)+2t(t-1) +2t^2*(t-1)+…+2t^(h-1)*(t-1)

= 1+ (t-1) (2t+2t^2+…+ 2t^(h-1))



k1 Nút gốc có ít nhất 1 khóa -> có ít nhất 2 nút con ở độ sau 1.
Mỗi nút con có ít nhất t-1 khóa => mỗi nút con có ít nhất t nút con. Vậy ở độ sâu 2 => có ít nhất 2t nút.

2t nút, mỗi nút có tối thiểu t nút con => ở độ sau 3 có ít nhất 2t*t nút con...
khmt-hung, 11/26/2006

Chiều cao của một B-cây

Định lý

Nếu $n \geq 1$ thì mọi B-cây T với n khóa, chiều cao h, và bậc tối thiểu $t \geq 2$ có $$h \leq \log_t \frac{n+1}{2}$$

Chứng minh

Có tối thiểu 2 nút ở độ sâu 1, $2t$ nút ở độ sâu 2,..., và $2t^{h-1}$ nút ở độ sâu h. Vậy số khóa n thõa

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

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

$$= 2t^h - 1$$

Do đó $t^h \leq \dfrac{n+1}{2}$, từ đây suy ra định lý.

Các thao tác lên một B-cây

- Các thao tác trên B cây
  - B-TREE-SEARCH
  - B-TREE-CREATE
  - B-TREE-INSERT
  - B-TREE-DELETE
- Trong các thủ tục ta quy ước:
  - Gốc của B-cây luôn luôn nằm trong bộ nhớ chính.
  - Một nút bất kỳ mà là một tham số được truyền đi trong một thủ tục thì đều đã thực thi thao tác DISK-READ.

Tìm trong một B-cây

- Thủ tục để tìm một khóa trong một B-cây
  - Input:
    - Một con trỏ chỉ đến nút gốc x của một cây con, và
    - Một khóa k cần tìm trong cây con.
  - Output:
    - Nếu k có trong cây thì trả về một cặp (y, i) gồm một nút y và một chỉ số i mà $key_i\,[y] = k$
    - Nếu k không có trong cây thì trả về NIL.

k3

Tìm trong một B-cây

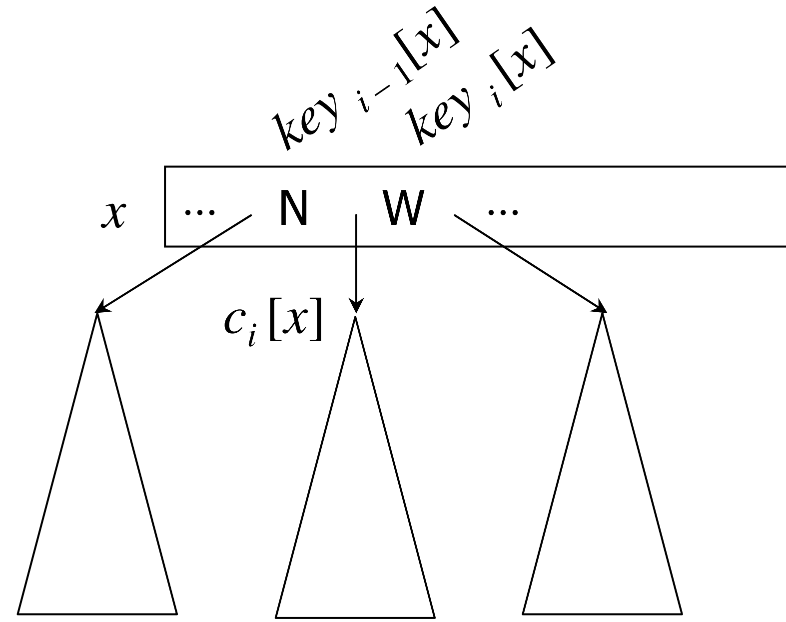
(tiếp)

B-TREE-SEARCH(x, k)

```
i ← 1
while i ≤ n[x] and k > key_i [x]
    do i ← i + 1

if i ≤ n[x] and k = key_i [x]
   then return (x, i)

if leaf [x]
   then return NIL
   else DISK-READ(c_i [x])
        return B-TREE-SEARCH(c_i [x], k)
```



k3 Ci(x) có thể ở giữa KEYi-1 và KEYi
Có thể Ci(x) là con trỏ cuối cùng
khmt-hung, 11/26/2006

k4

Tìm trong một B-cây

(tiếp)

- Các nút mà giải thuật truy cập tạo nên một đường đi từ gốc xuống đến nút có chứa khóa (nếu có).

  Thời gian CPU để xử lý mỗi nút là $O(t)$; t-bậc tối thiểu của cây
- Do đó
  - số disk pages mà B-TREE-SEARCH truy cập là $\Theta(h) = \Theta(\log_t n)$, với h là chiều cao của cây, n là số khoá của cây.
  - B-TREE-SEARCH cần thời gian CPU $O(t\,h) = O(t \log_t n)$.

k4 Mỗi nút có tối đa 2t-1 khóa

khmt-hung, 11/26/2006

Tạo một B-cây trống

- Thủ tục để tạo một nút gốc trống
  - Gọi thủ tục ALLOCATE-NODE để làm một nút mới

```
B-TREE-CREATE(T)
    x ← ALLOCATE-NODE()
    leaf[x] ← TRUE
    n[x] ← 0
    DISK-WRITE(x)
    root[T] ← x
```

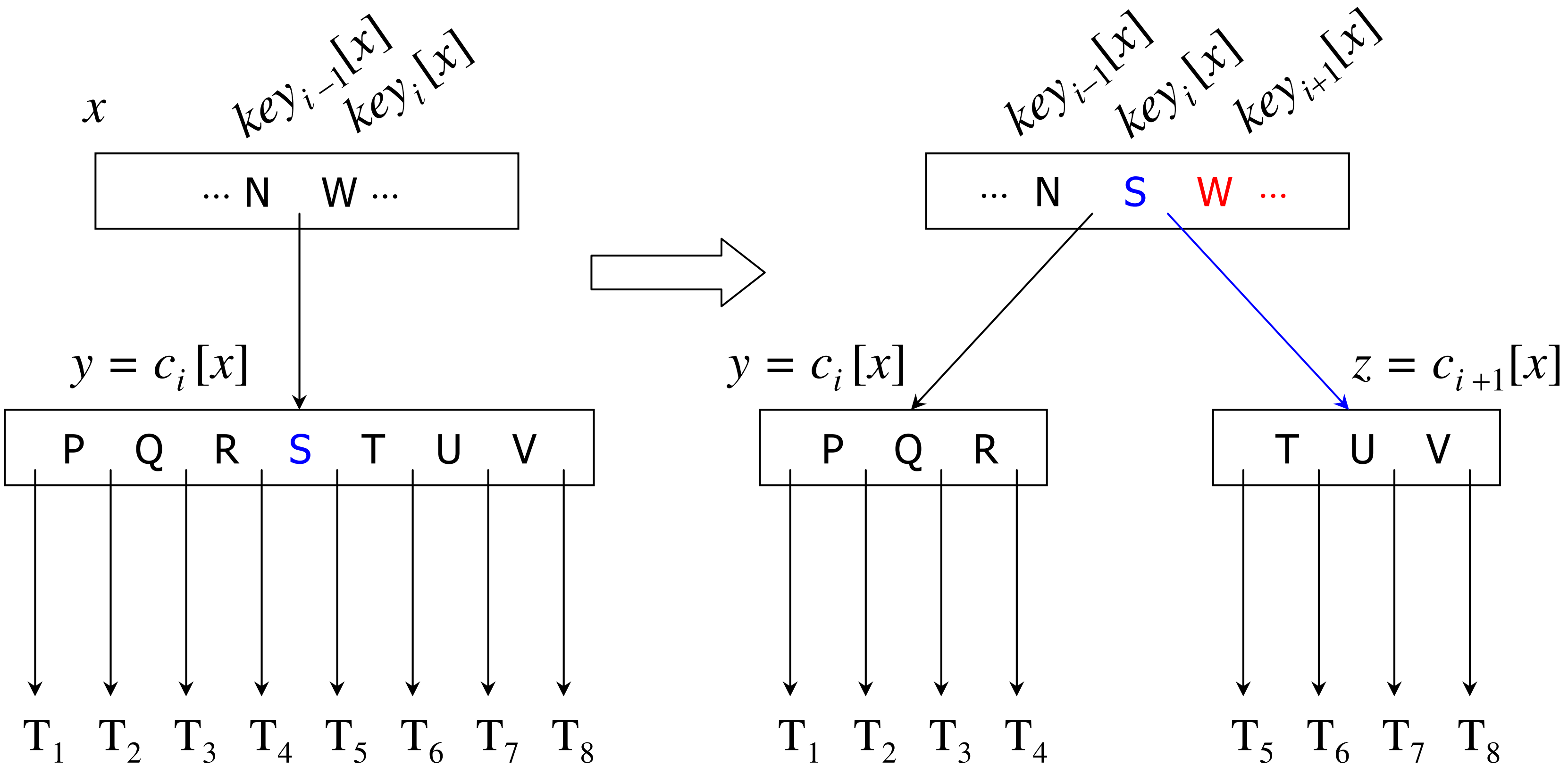
  - B-TREE-CREATE cần $O(1)$ thời gian CPU và $O(1)$ disk operations.

Chèn một khóa vào một B-cây

- Định nghĩa nút đầy *y*: $n[y] = 2t - 1$.
- Định nghĩa *khóa giữa* (median key) của *y* là khóa thứ t.
- Ta sẽ chèn khóa vào một lá của cây. Để tránh trường hợp chèn khóa vào một lá đã đầy, ta cần một thao tác *tách* (split) một nút đầy *y*. Thao tác này
  - tách nút đầy *y* quanh khóa giữa của nó thành hai nút, mỗi nút có $t - 1$ khóa
  - di chuyển khóa giữa lên nút cha của *y* (phải là nút không đầy) vào một vị trí thích hợp.
- Để chèn khóa mà chỉ cần một lượt đi (one pass) từ nút gốc đến một lá, thủ tục sẽ tách mọi nút đầy trên đường đi từ gốc đến nút lá (đảm bảo được rằng khi tách một nút đầy *y* thì nút cha của nó phải là không đầy).

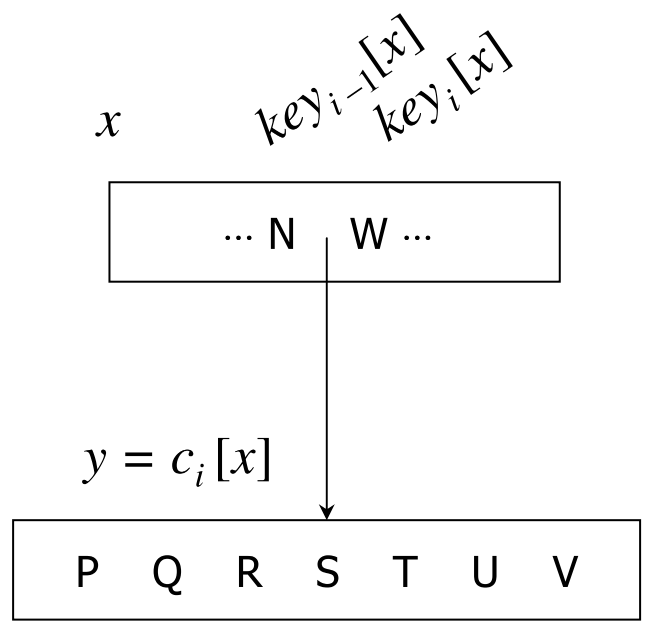
Ví dụ tách một nút đầy

- Bậc tối thiểu $t = 4$. Vậy số khóa tối đa của một nút là 7.
- Tách nút đầy y là con của nút không đầy x.



Tách một nút của một B-cây

- Thủ tục B-TREE-SPLIT-CHILD
  - Input: một nút trong không đầy x, một chỉ số i mà nút $y = c_i[x]$ là một nút đầy
  - Thủ tục tách y thành hai nút và chỉnh x để cho x có thêm một nút con.




```
B-TREE-SPLIT-CHILD(x, i, y)
    z ← ALLOCATE-NODE()// Tạo nút để tách nút y
    leaf[z] ← leaf[y]
    n[z] ← t – 1
  // lấy các khóa nửa bên phải của nút y
   for j ← 1 to t – 1
        do key_j[z] ← key_{j+t}[y]

    if not leaf[y]
      then for j ← 1 to t
              do c_j[z] ← c_{j+t}[y]
    n[y] ← t – 1
```

Tách một nút của một B-cây

(tiếp)

//Điều chỉnh các con trỏ và các khoá trong nút x

for $j \leftarrow n[x] + 1$ **downto** $i + 1$ // đẩy các con trỏ sang phải

  do $c_{j+1}[x] \leftarrow c_j[x]$

$c_{i+1}[x] \leftarrow z$

for $j \leftarrow n[x]$ **downto** i // đẩy các key sang phải

  do $key_{j+1}[x] \leftarrow key_j[x]$

$key_i[x] \leftarrow key_t[y]$

$n[x] \leftarrow n[x] + 1$

DISK-WRITE(y)

DISK-WRITE(z)

DISK-WRITE(x)

Tách một nút của một B-cây

(tiếp)

- B-TREE-SPLIT-CHILD cần
  - $\Theta(t)$ thời gian CPU (các dòng 4-5 và 7-8)
  - $O(1)$ disk operations (các dòng 17-19).

Chèn một khóa vào trong một B-cây

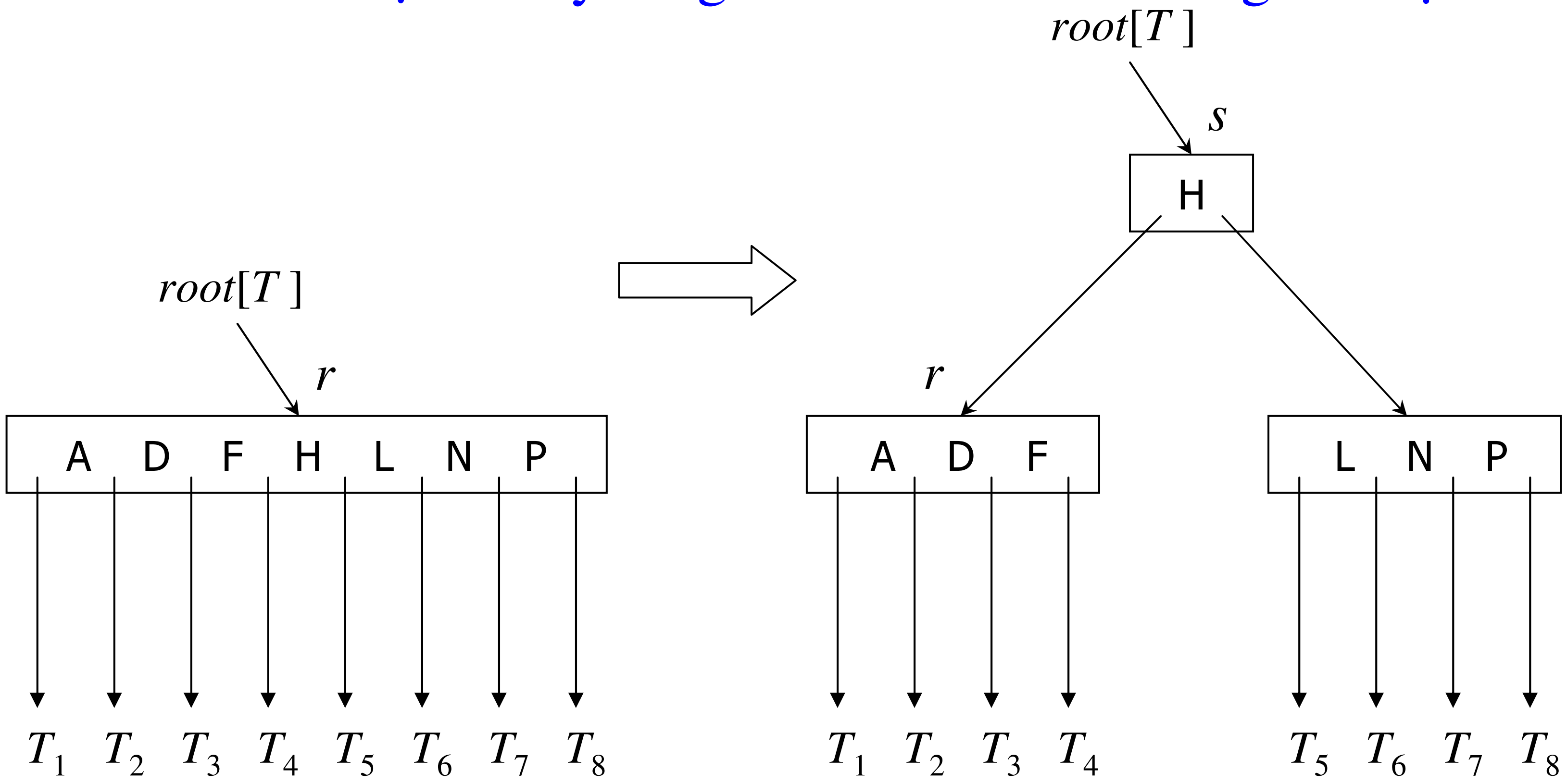
- Thủ tục B-TREE-INSERT để chèn một khóa k vào một B-cây T.
  - Thủ tục gọi B-TREE-SPLIT-CHILD để đảm bảo khi gọi đệ quy thì sẽ không bao giờ xuống một nút đã đầy.

B-TREE-INSERT(T, k)

```
r ← root[T]
if n[r] = 2t – 1
   then s ← ALLOCATE-NODE()// Nếu gốc đầy thì tách gốc
        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)
```

Tách một nút gốc đầy

- Ví dụ: tách một nút gốc đầy của một B-cây mà bậc tối thiểu là $t = 4$.
- Nút gốc mới là *s*. Nút gốc cũ *r* được tách thành hai nút con của *s*.
- Chiều cao của một B-cây tăng thêm 1 mỗi khi nút gốc được tách.



Chèn một khóa vào một nút không đầy

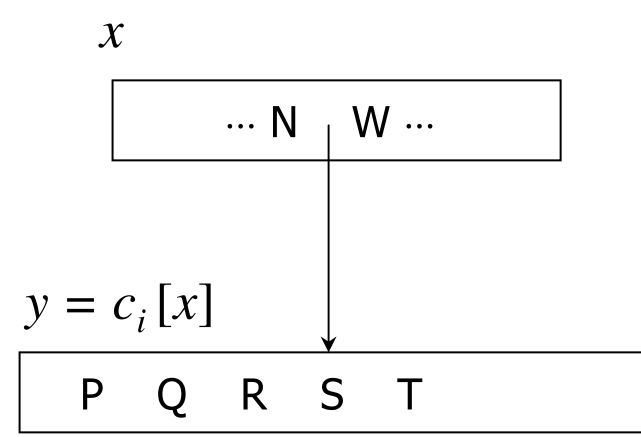
- Thủ tục để chèn một khóa vào một nút không đầy

nút lá

B-TREE-INSERT-NONFULL(x, k)

$i \leftarrow n[x]$
if *leaf* $[x]$
then while $i \geq 1$ and $k < key_i\,[x]$ // đẩy dần sang phải để lấy 1 chỗ để chèn k
do $key_{i+1}[x] \leftarrow key_i\,[x]$
$i \leftarrow i - 1$
$key_{i+1}[x] \leftarrow k$
$n[x] \leftarrow n[x] + 1$
DISK-WRITE(x)

(tiếp)



else while $i \geq 1$ and $k < key_i[x]$

  do $i \leftarrow i - 1$

 $i \leftarrow i + 1$

 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 $i \leftarrow i + 1$

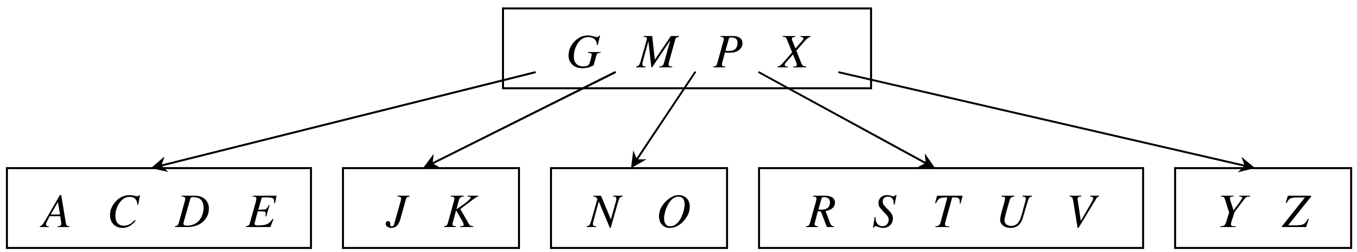
 B-TREE-INSERT-NONFULL($c_i[x]$, k)

Phân tích chèn một khóa vào trong một B-cây

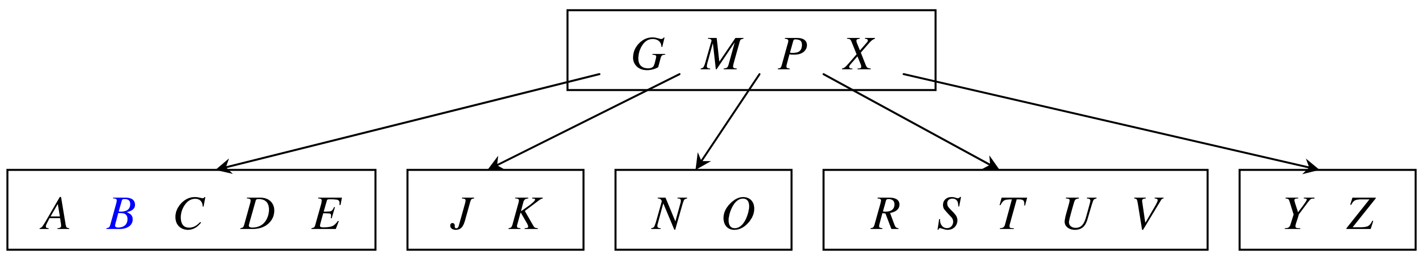
- Thủ tục B-TREE-INSERT cần
  - số truy cập đĩa là $O(h)$ – tương ứng với số lần gọi DISK-READ và DISK-WRITE .
  - thời gian CPU là $O(t\ h) = \mathrm{O}(t \log_t n)$

Các trường hợp khi chèn một khóa vào một B-cây

- Cho một B-cây với bậc tối thiểu $t = 3$
- Cây lúc đầu



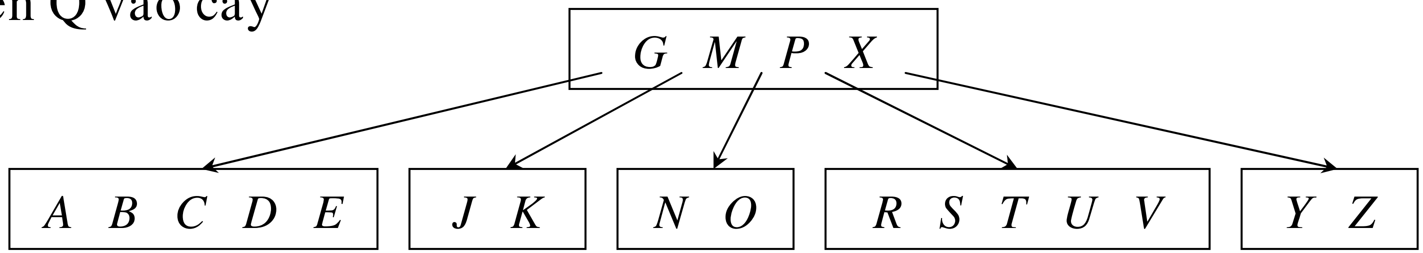
- Đã chèn *B* vào

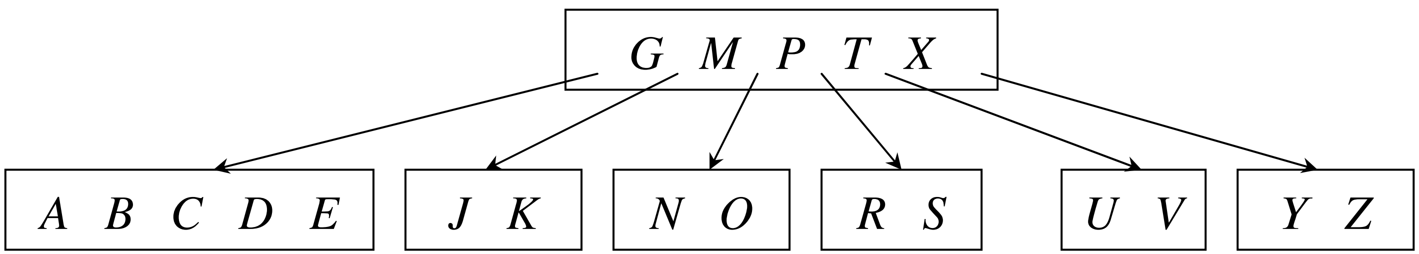


Ví dụ cho các trường hợp khi chèn một khóa vào một B-cây

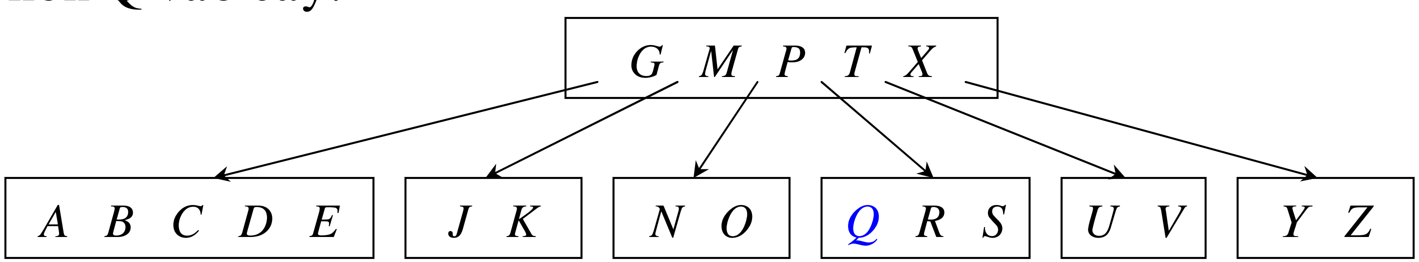
(tiếp)

Chèn Q vào cây





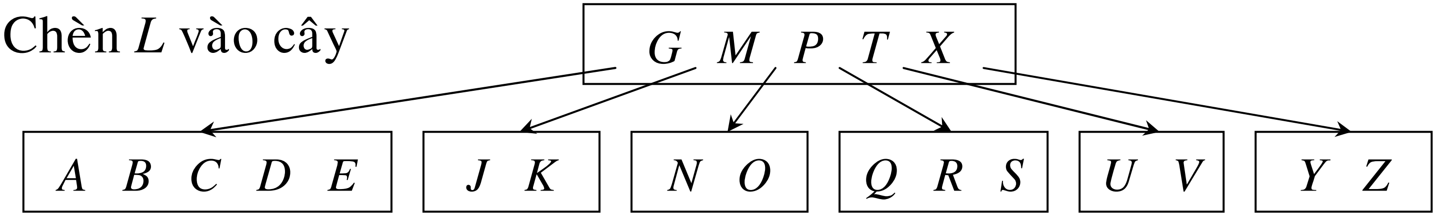
Đã chèn Q vào cây:

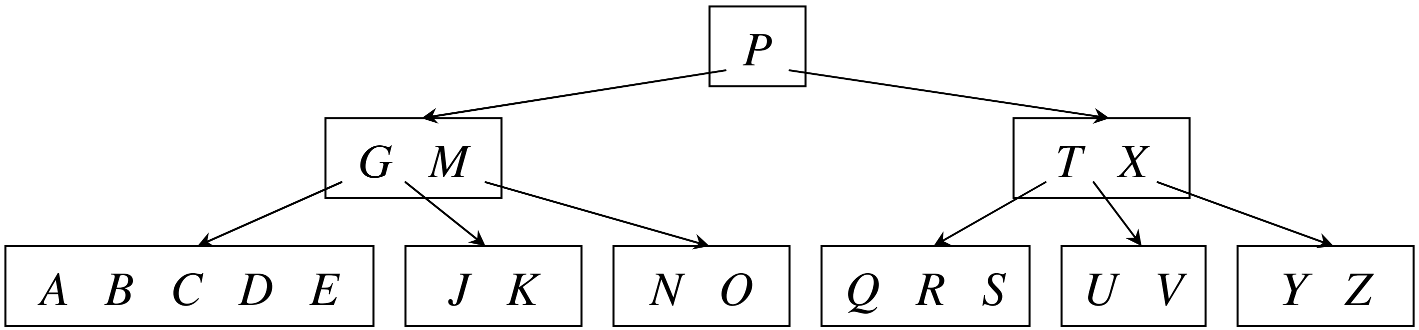


Ví dụ cho các trường hợp khi chèn một khóa vào một B-cây

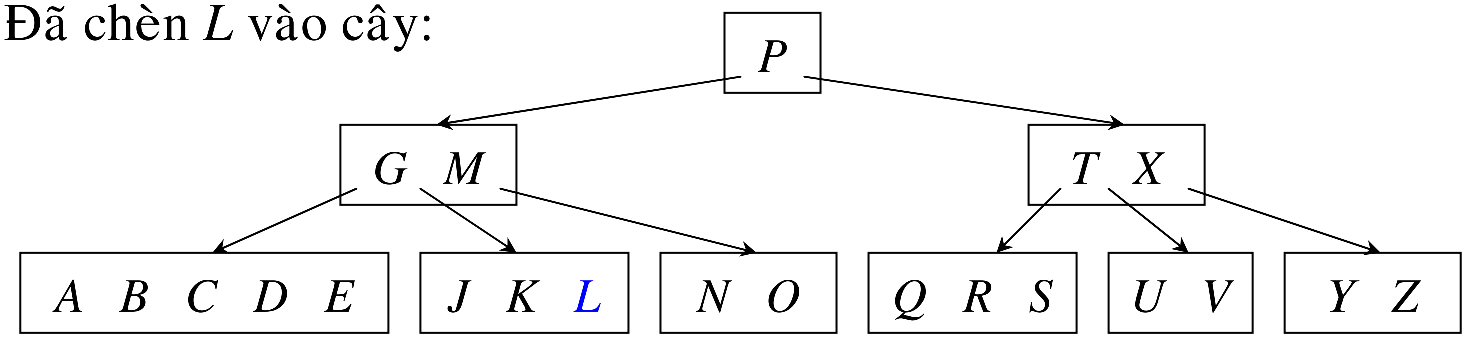
(tiếp)

- Chèn *L* vào cây





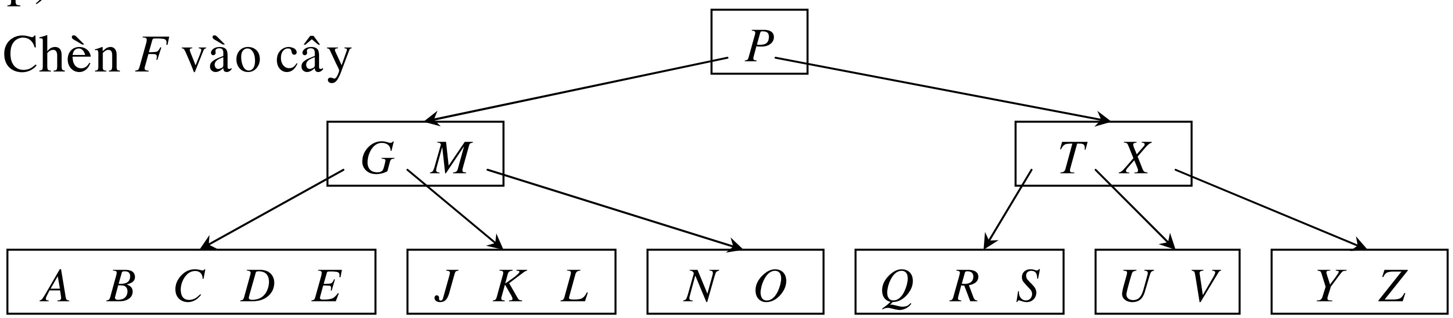
- Đã chèn *L* vào cây:

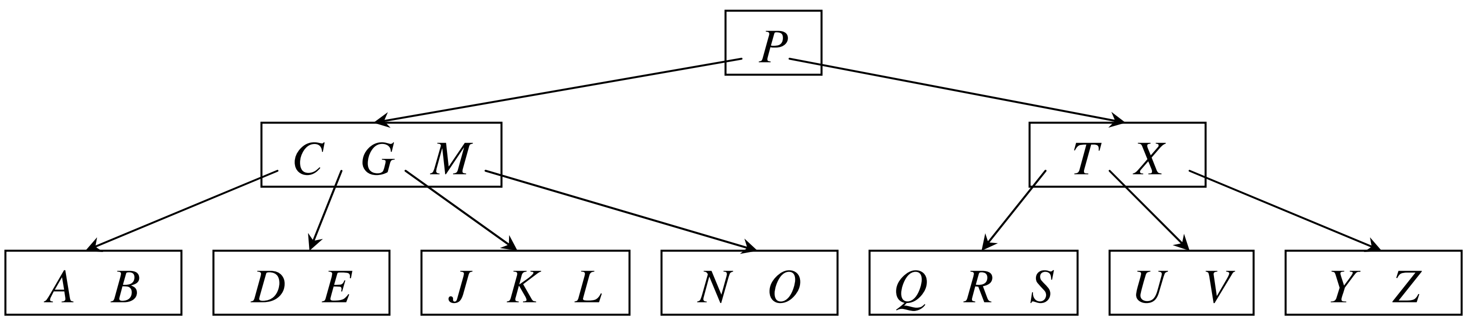


Ví dụ cho các trường hợp khi chèn một khóa vào một B-cây

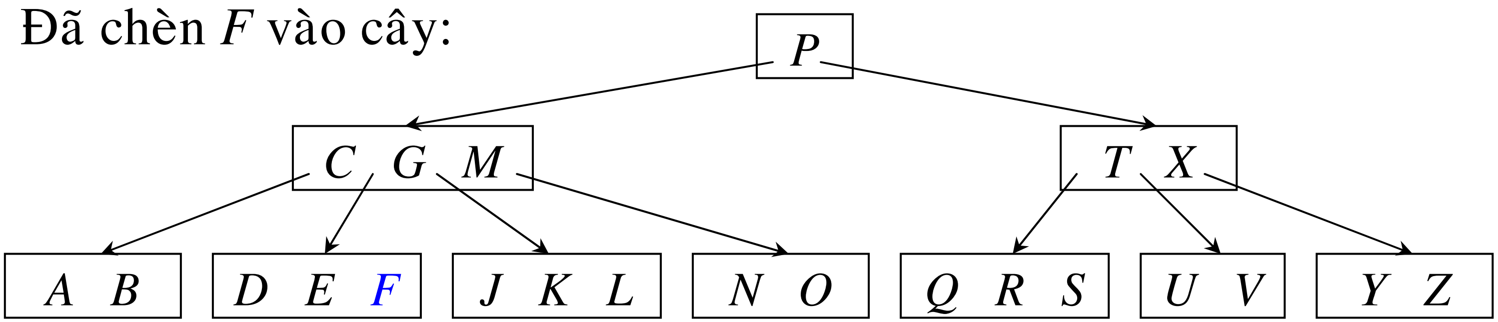
(tiếp)

- Chèn *F* vào cây





- Đã chèn *F* vào cây:



Xóa một khóa khỏi một B-cây

Thủ tục B-TREE-DELETE(x, k) để xóa khóa k khỏi cây con có gốc tại x bảo đảm rằng khi B-TREE-DELETE được gọi đệ quy lên x thì số khóa trong x phải $\geq t$ *với* t - bậc tối thiểu của cây (trừ nút gốc)
.

Do đó, khi thủ tục thực thi, đôi khi một khóa được di chuyển (từ một nút thích hợp khác) vào một nút trước khi đệ quy xuống nút đó.

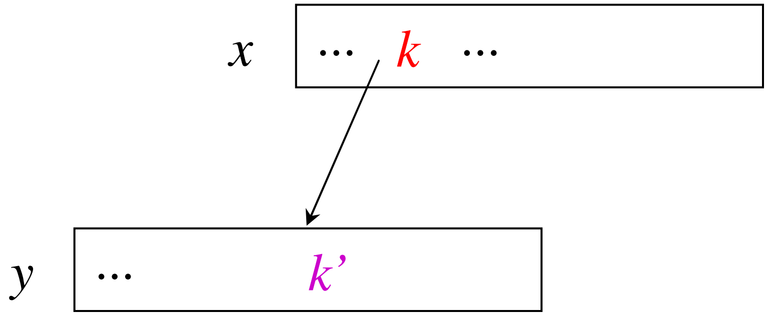
Xóa một khóa khỏi một B-cây

B-TREE-DELETE(x, k)

1. Nếu khóa k có trong nút x và x là một nút lá thì xóa k khỏi x.

2. Nếu khóa k có trong nút x và x là một nút trong thì

a. Nếu nút con y ở trước k có ít nhất t khóa thì tìm khóa trước (predecessor) k' của k trong cây con có gốc tại y. Xóa k' bằng cách gọi đệ quy B-TREE-DELETE(y, k'), kế đó trong x thay k bằng k'.



Xóa một khóa khỏi một B-cây

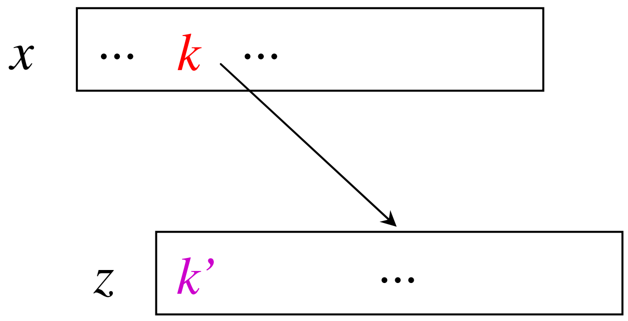
B-TREE-DELETE(x, k)

1. ...

2. Nếu khóa k có trong nút x và x là một nút trong thì

a. ...

b. Tương tự, nếu nút con z ở sau k có ít nhất t khóa thì tìm khóa sau (successor) k' của k trong cây con có gốc tại z. Xóa k' bằng cách gọi đệ quy B-TREE-DELETE(z , k'), kế đó trong x thay k bằng k'.



Xóa một khóa khỏi một B-cây

B-TREE-DELETE(x, k)

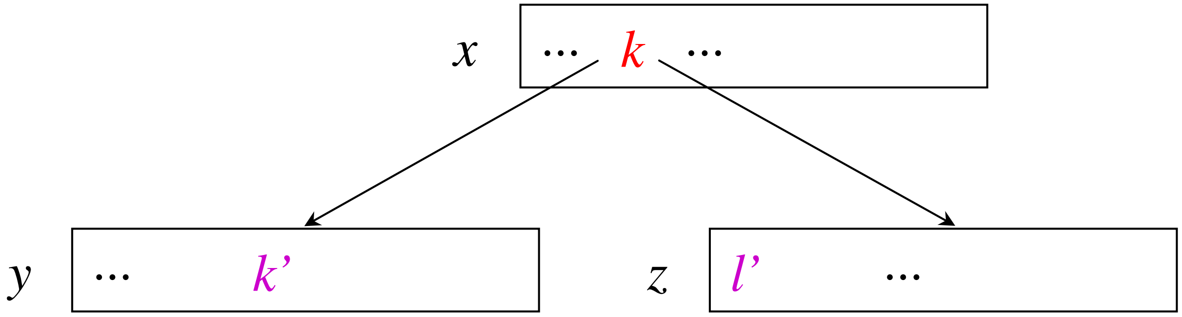
1. ...

2. Nếu khóa k có trong nút x và x là một nút trong thì

a. ...

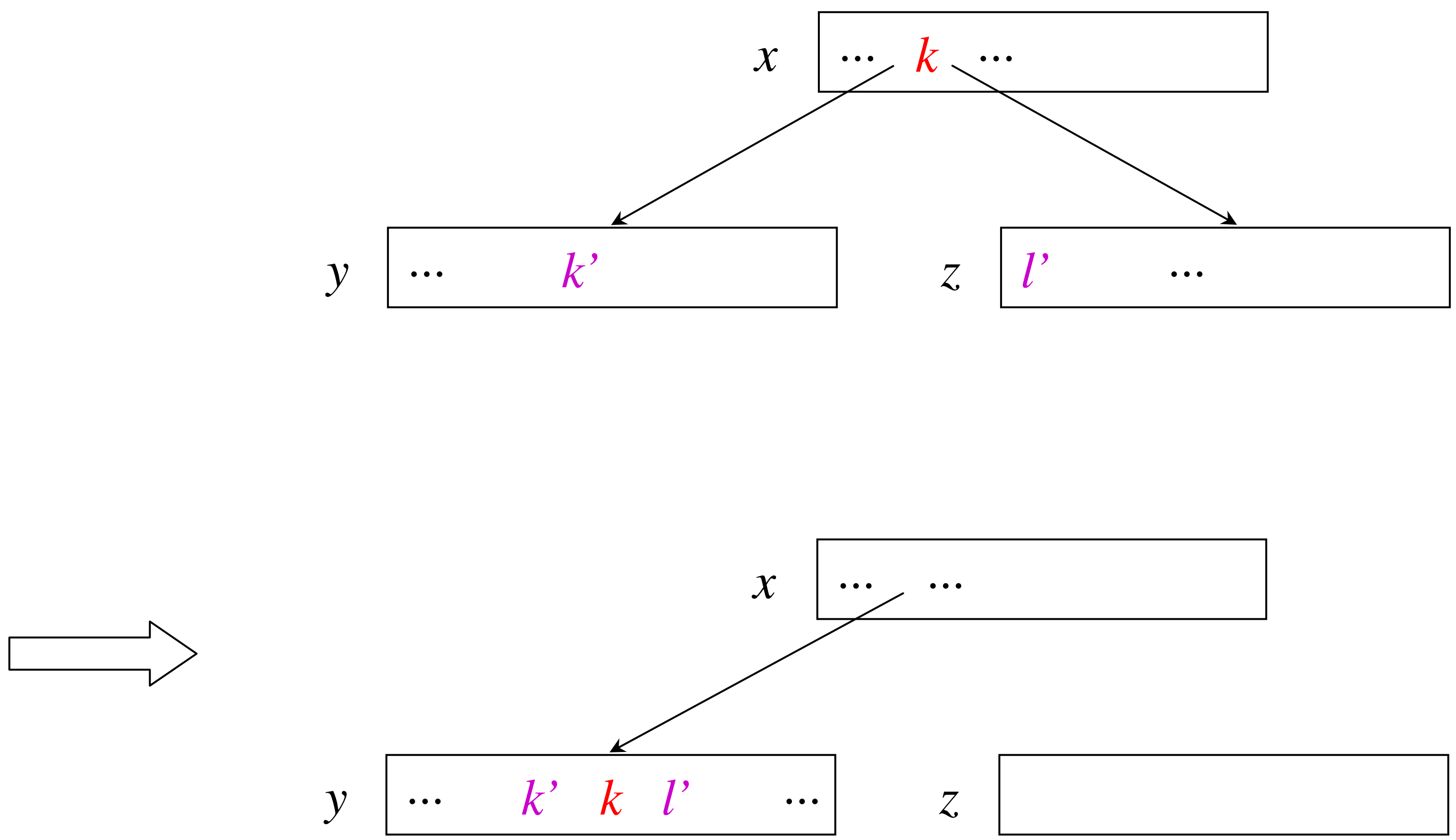
b. ...

c. Nếu không, cả y và z đều chỉ có $t - 1$ khóa, hợp nhất k và nguyên cả z vào y, thành ra x mất k và con trỏ đến z , và bây giờ y chứa $2t - 1$ khóa. Giải phóng (free) z và gọi đệ quy B-TREE-DELETE(y, k) để xóa k khỏi cây có gốc y.



Xóa một khóa khỏi một B-cây

(tiếp)



Xóa một khóa khỏi một B-cây

B-TREE-DELETE(x, k)

1. ...

2. ...

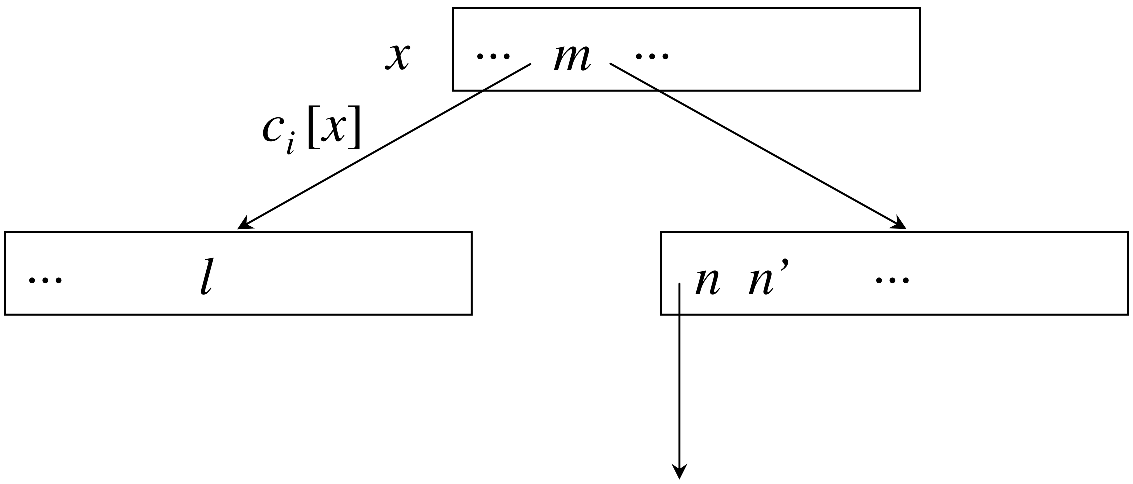
3. Nếu khóa k không có trong nút trong x thì xác định gốc $c_i[x]$ của cây con chứa k, nếu k có trong cây. Nếu $c_i[x]$ chỉ có $t - 1$ khóa, thực thi bước 3a hoặc 3b để đảm bảo rằng ta sẽ xuống đến một nút chứa ít nhất t khóa. Xong rồi gọi B-TREE-DELETE lên nút con thích hợp của x.

Xóa một khóa khỏi một B-cây

(tiếp)

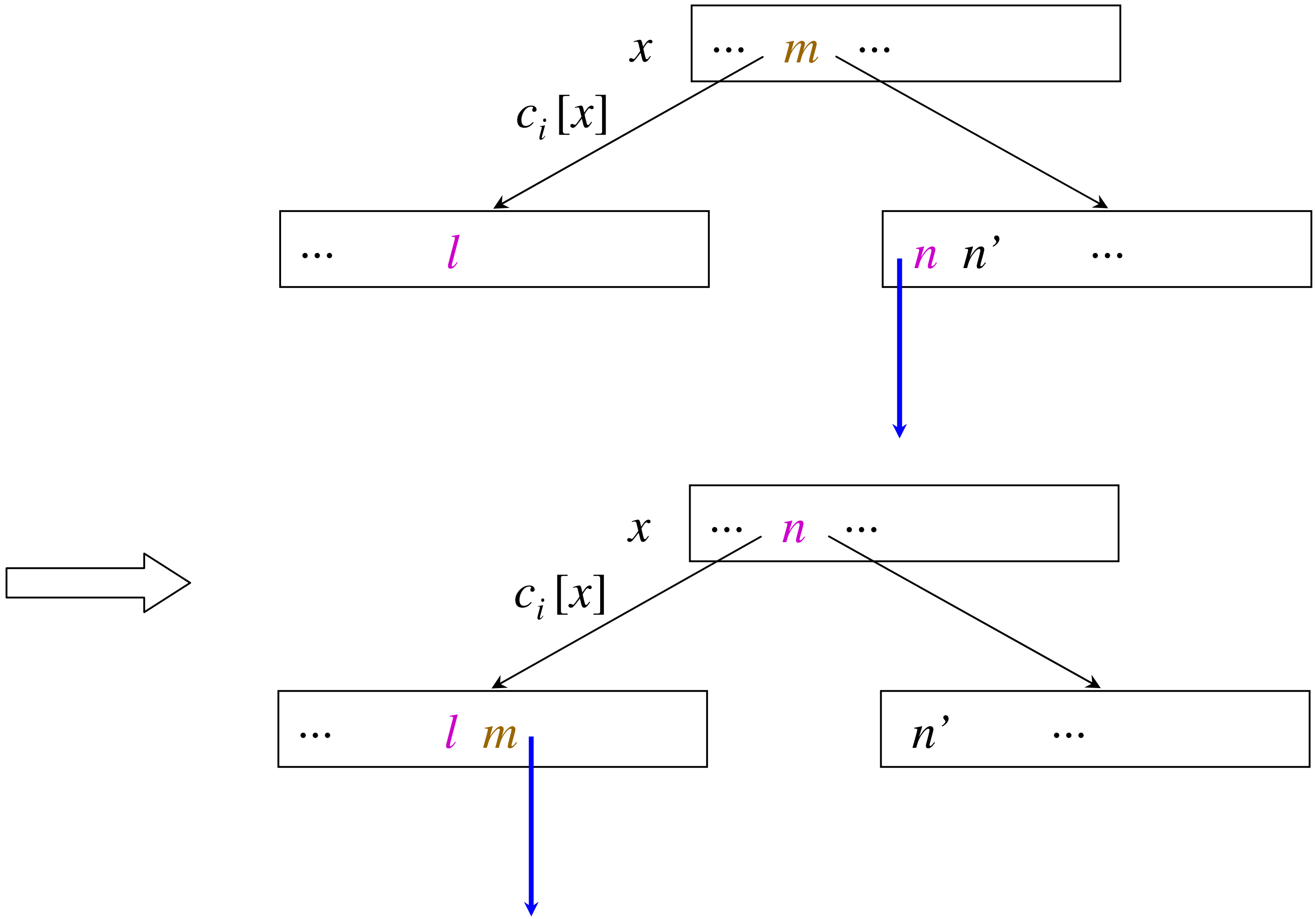
3. ...

a. Nếu $c_i[x]$ chỉ có $t - 1$ khóa, nhưng lại có một nút anh em với ít nhất t khóa, thì cho $c_i[x]$ thêm một khóa bằng cách đem một khóa từ x xuống $c_i[x]$, đem một khóa từ nút anh em ngay bên trái hay ngay bên phải của $c_i[x]$ lên x, và đem con trỏ tương ứng từ nút anh em vào nút $c_i[x]$.



Xóa một khóa khỏi một B-cây

(tiếp)



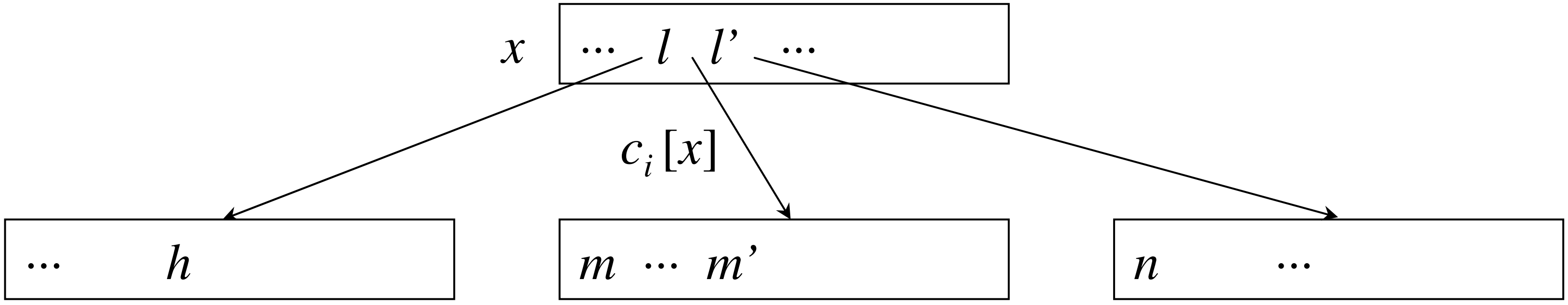
Xóa một khóa khỏi một B-cây

(tiếp)

3. ...

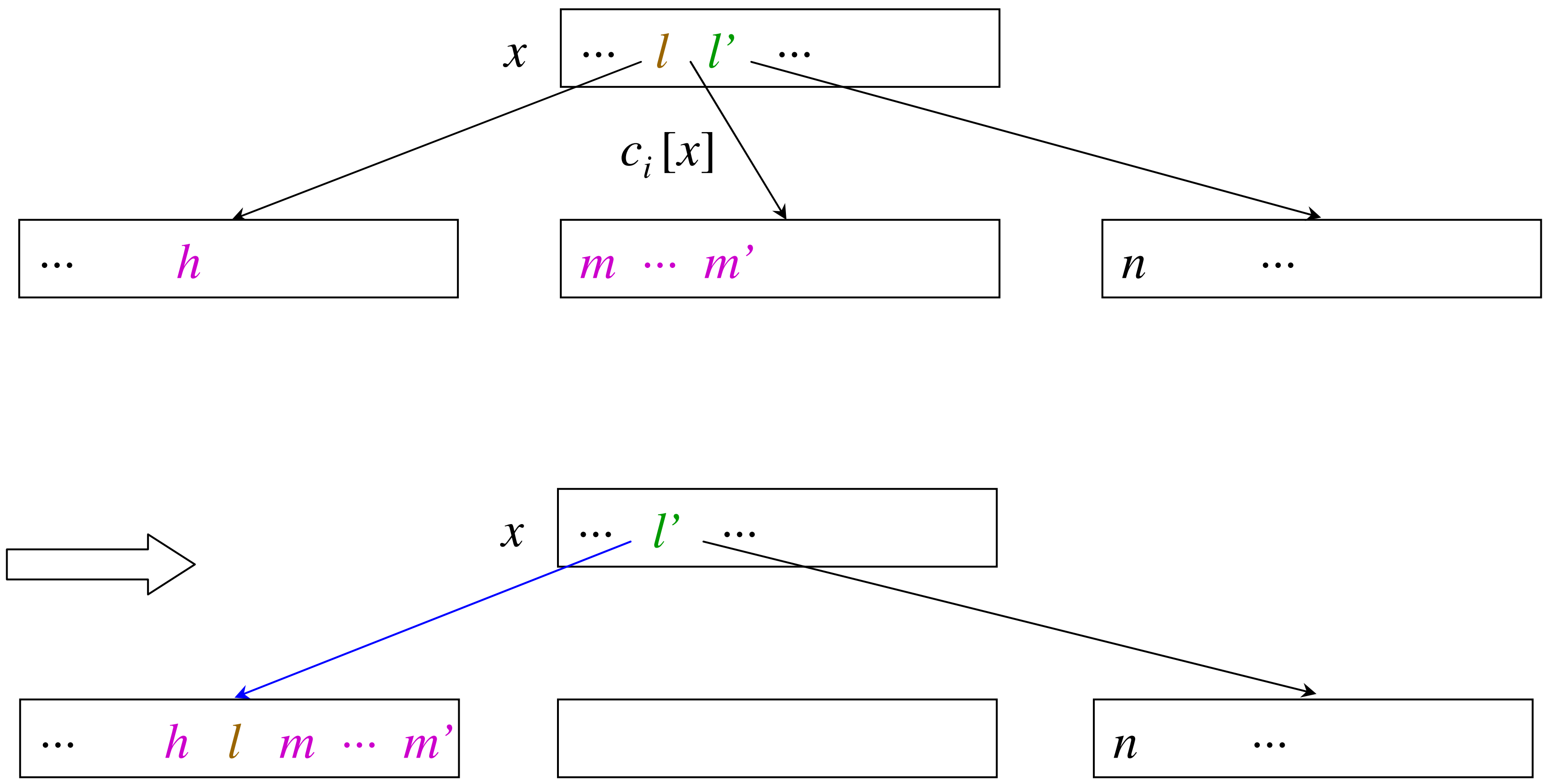
a. ...

b. Nếu $c_i[x]$ và mọi nút anh em của nó chỉ có $t - 1$ khóa, thì hợp nhất $c_i[x]$ và một nút anh em bằng cách đem một khóa từ x xuống nút mới tạo, khóa này sẽ là khóa giữa của nút.



Xóa một khóa khỏi một B-cây

(tiếp)



Xóa một khóa khỏi một B-cây

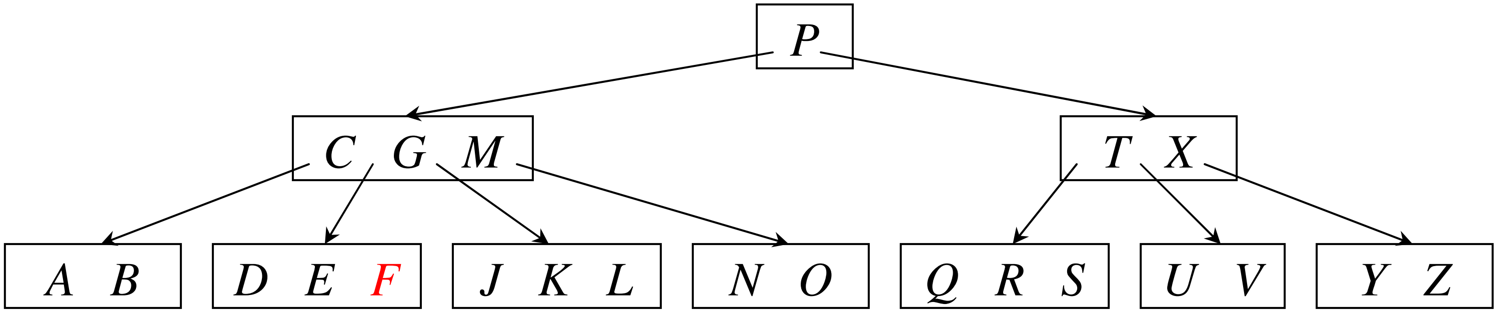
(tiếp)

Thủ tục B-TREE-DELETE cần

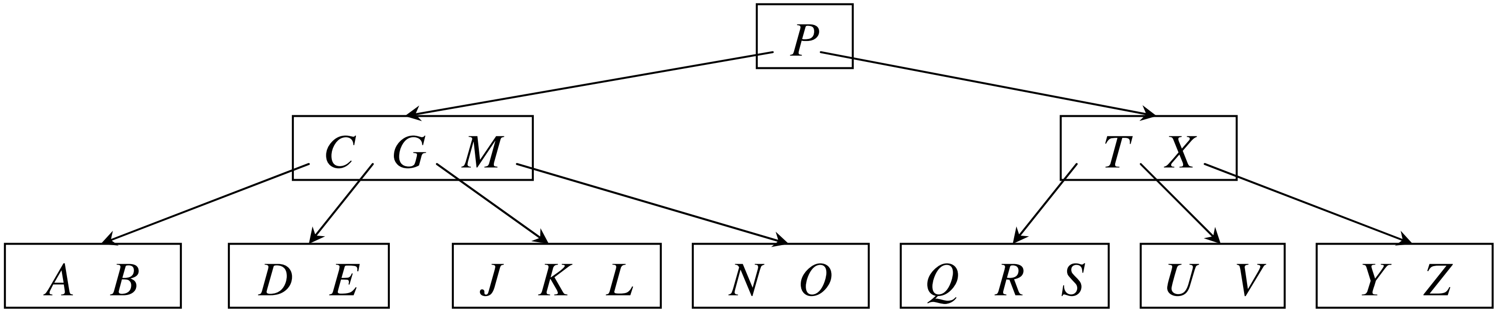
- số truy cập lên đĩa là $O(h)$ vì có $O(1)$ lần gọi DISK-READ và DISK-WRITE giữa các gọi đệ quy của thủ tục.
- thời gian CPU của thủ tục là $O(t\,h) = O(t \log_t n)$.

Ví dụ cho các trường hợp khi xóa một khóa khỏi một B-cây

- Cho một B-cây có bậc tối thiểu $t = 3$
- Cây lúc đầu, xóa *F* khỏi cây

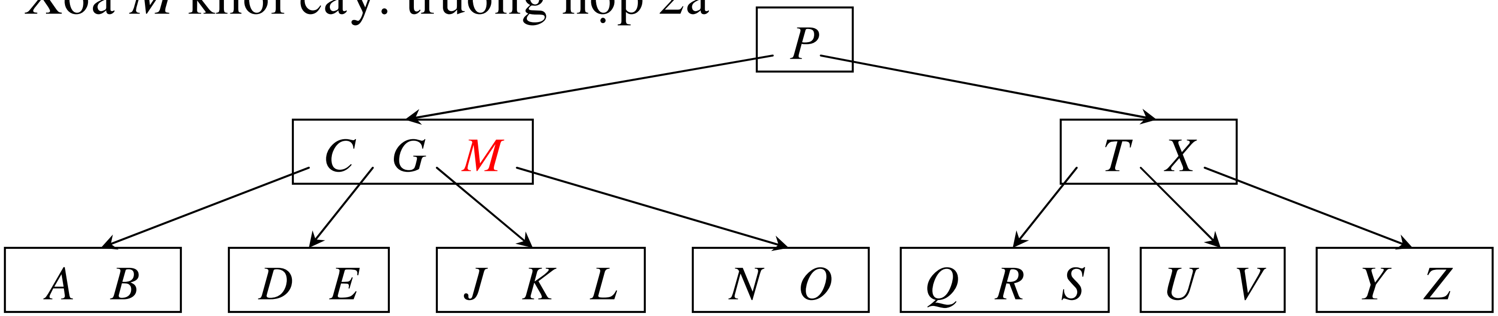


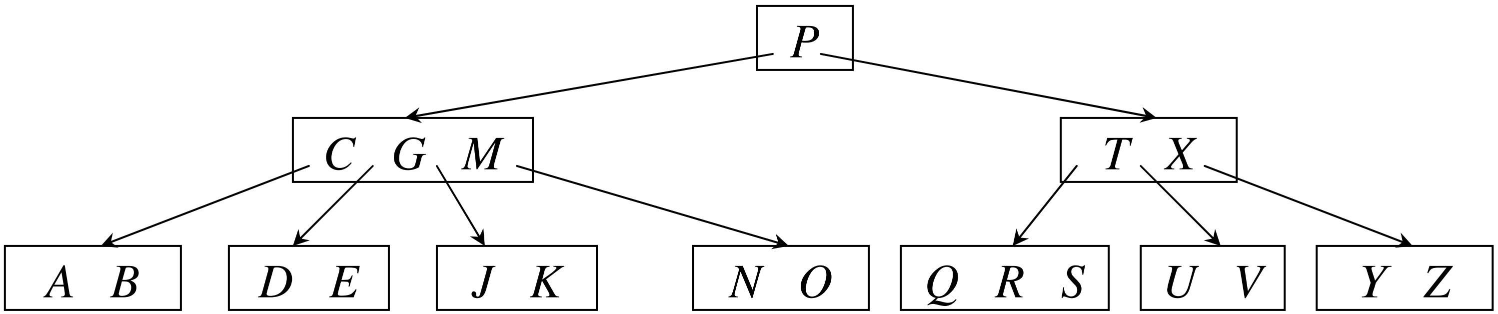
- *F* đã được xóa: trường hợp 1



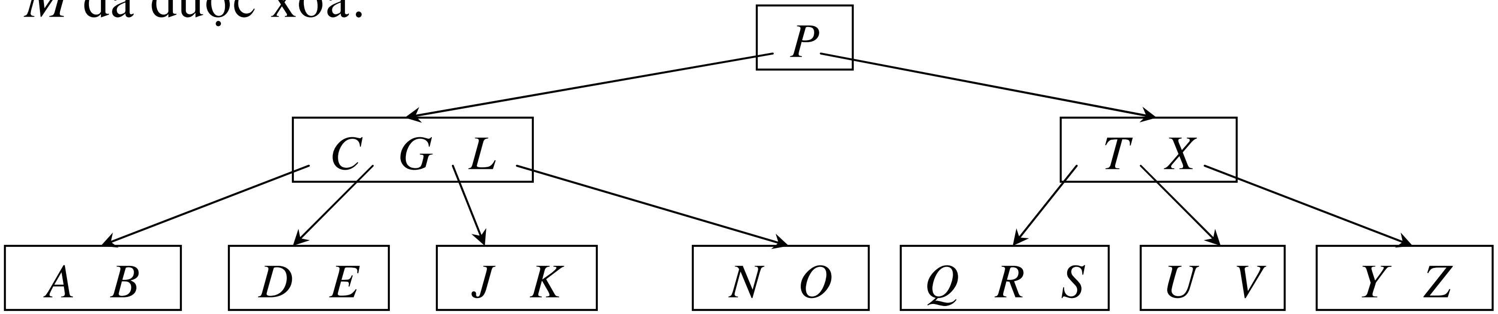
Ví dụ cho các trường hợp khi xóa một khóa khỏi một B-cây

- Xóa *M* khỏi cây: trường hợp 2a



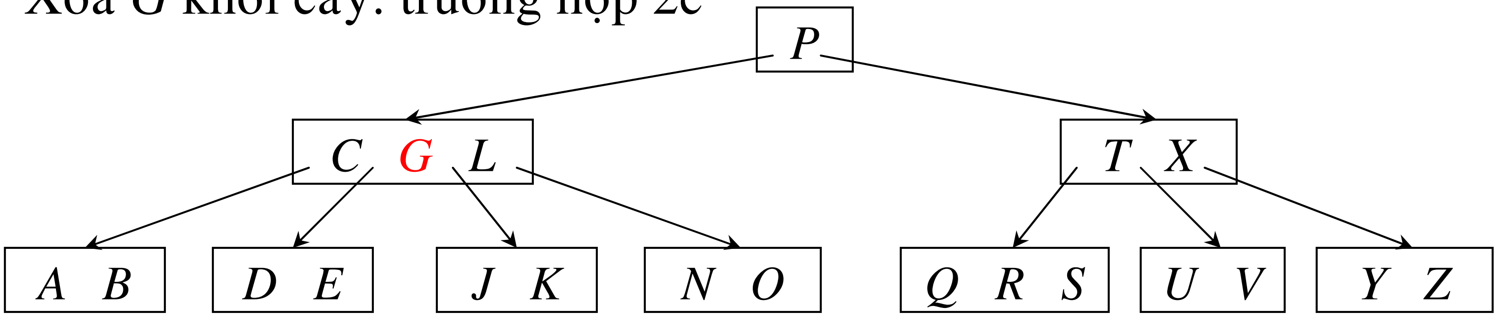


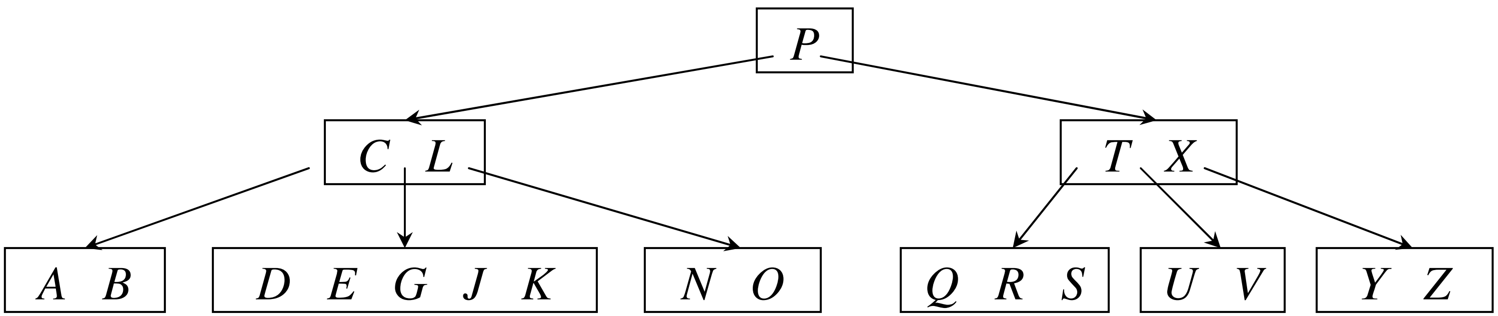
- *M* đã được xóa:



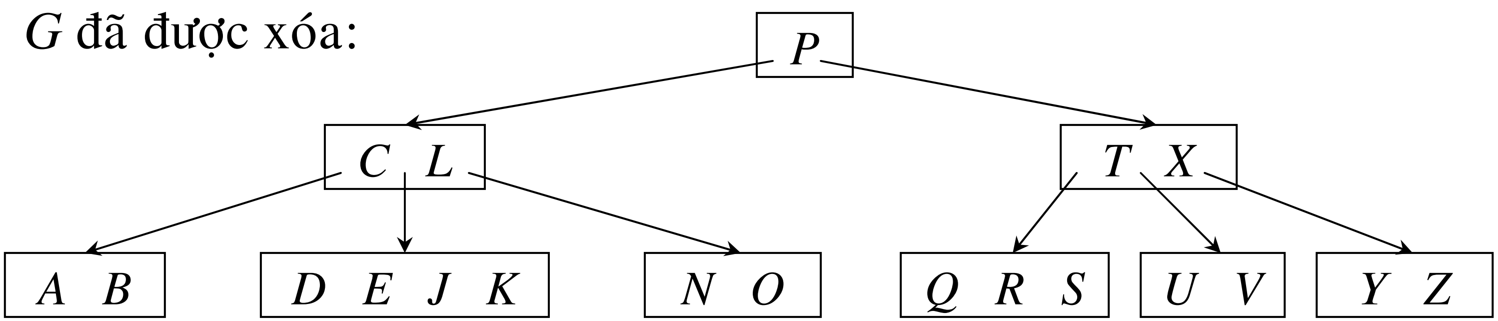
Ví dụ cho các trường hợp khi xóa một khóa khỏi một B-cây

- Xóa *G* khỏi cây: trường hợp 2c



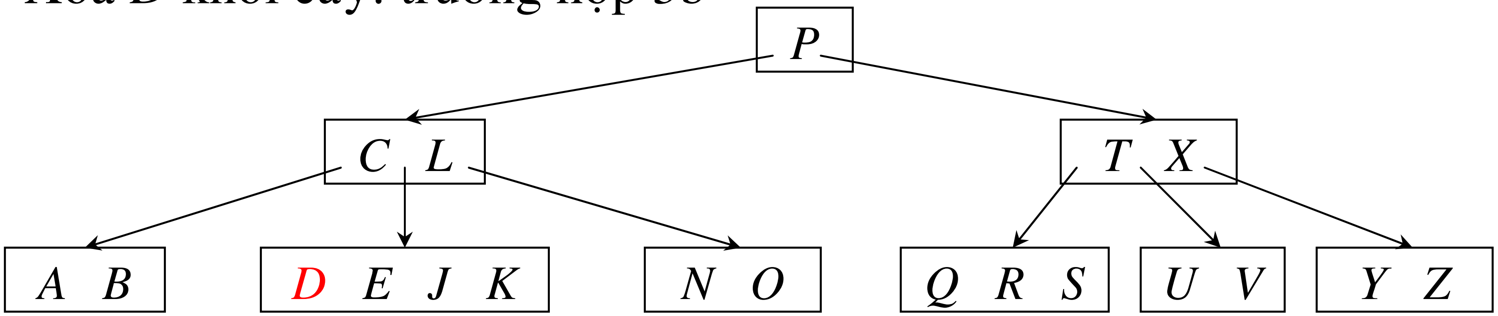


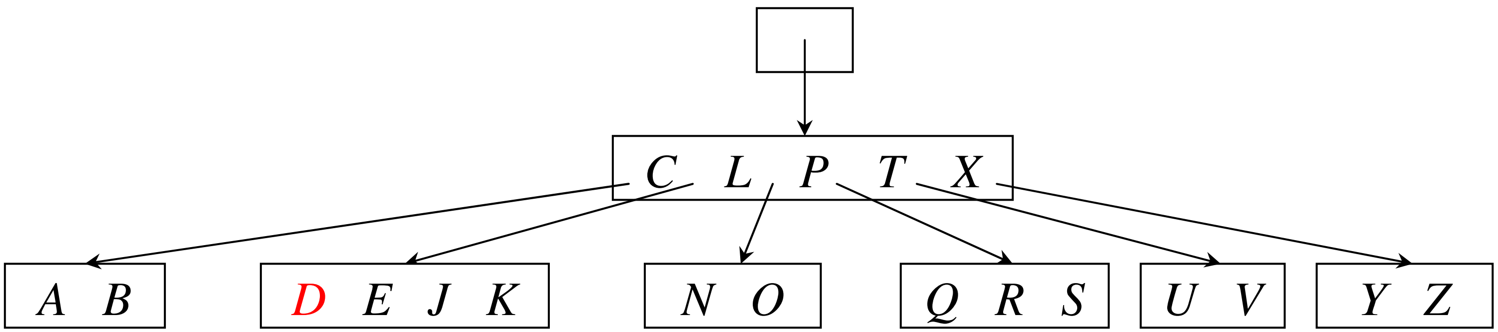
- *G* đã được xóa:



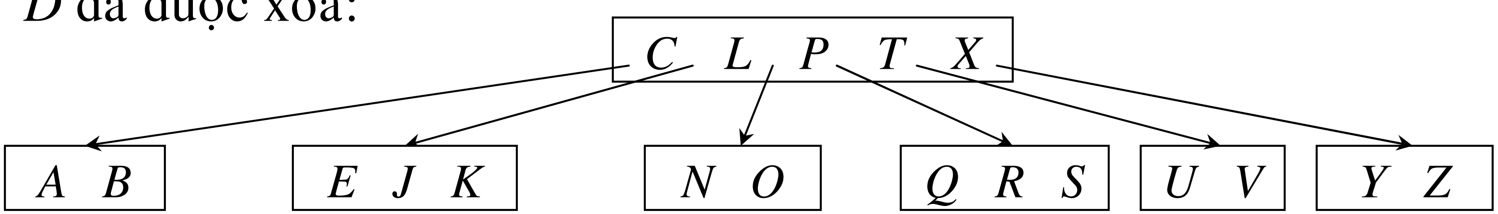
Ví dụ cho các trường hợp khi xóa một khóa khỏi một B-cây

- Xóa *D* khỏi cây: trường hợp 3b



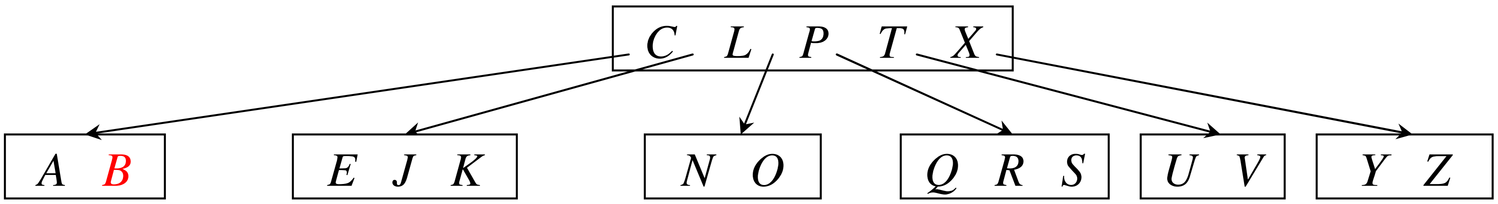


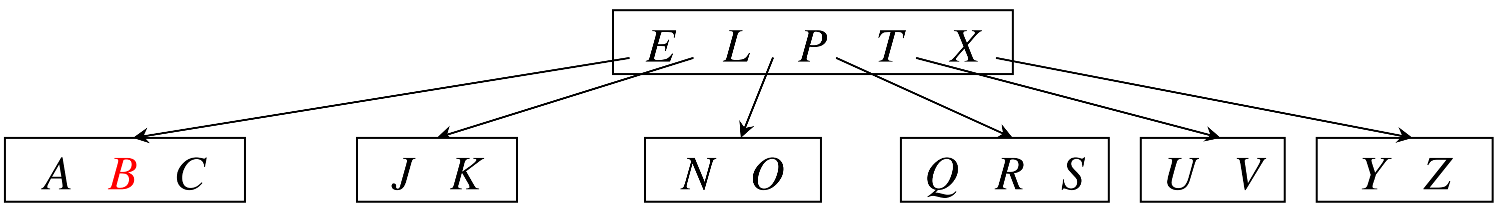
- *D* đã được xóa:



Ví dụ cho các trường hợp khi xóa một khóa khỏi một B-cây

- Xóa *B* khỏi cây: trường hợp 3a





- *B* đã được xóa khỏi cây: