- Thuật ngữ và khái niệm
- Độ phức tạp về thời gian của thuật toán
- Các ví dụ
- Kết luận và lưu ý

# THUẬT NGỮ VÀ KHÁI NIỆM

- Thuật toán là một thủ tục xác định bao gồm một dãy hữu hạn các bước cần thực hiện để thu được lời giải bài toán
- Một thuật toán luôn có một tập dữ liệu đầu vào (input) và một tập dữ liệu đầu ra (output) tương ứng với yêu cầu và lời giải bài toán

# THUẬT NGỮ VÀ KHÁI NIỆM

Có thể mô tả thuật toán bằng:

- Ngôn ngữ tự nhiên (Natural language)
- Mã giả (Pseudocode)
- Ngôn ngữ lập trình cấp cao (High programming languages) như Pascal, C/C++ vv

# THUẬT NGỮ VÀ KHÁI NIỆM

**Ví dụ**: Tìm $x$ trong dãy $a_1, a_2, \ldots, a_n$

Đầu vào: Số x, dãy $n$ số $a_1, a_2, \ldots, a_n$

Đầu ra:  Một giá trị logic true hoặc false

Search($x$, $a$, $n$)

1 **for** $i \leftarrow 1$ **to** $n$

2 **do** **if** $a_i = x$

3 **then** **return** true

4 **return** false

# THUẬT NGỮ VÀ KHÁI NIỆM

Độ phức tạp của thuật toán là chi phí về tài nguyên của hệ thống (chủ yếu là thời gian, bộ nhớ, CPU, đường truyền) cần thiết để thực hiện thuật toán

# THUẬT NGỮ VÀ KHÁI NIỆM

Phân tích thuật toán (Analyzing of Algorithm) là quá trình tìm ra những đánh giá về tài nguyên cần thiết để thực hiện thuật toán

# THUẬT NGỮ VÀ KHÁI NIỆM

Độ phức tạp về thời gian của thuật toán:

- Được qui về đếm số lệnh cần thực thi của thuật toán
- Đó là một hàm $T(n)$ phụ thuộc vào kích thước $n$ của input
- Coi như có một máy trừu tượng (abstract machine) để thực hiện thuật toán

# ĐPT THỜI GIAN CỦA THUẬT TOÁN

- Thời gian tối thiểu để thực hiện thuật toán với kích thước đầu vào *n* gọi là thời gian chạy tốt nhất của thuật toán
- Thời gian nhiều nhất để thực hiện thuật toán với kích thước đầu vào *n* được gọi là thời gian chạy xấu nhất của thuật toán
- Thời gian trung bình để thực hiện thuật toán với kích thước đầu vào *n* được gọi là thời gian chạy trung bình của thuật toán

# ĐPT THỜI GIAN CỦA THUẬT TOÁN

**Ví dụ** Đánh giá độ phức tạp về thời gian của thuật toán

Search(*x*, *a*, *n*)

1 **for** $i \leftarrow 1$ **to** $n$

2 **do** **if** $a_i = x$

3 **then** **return** true

4 **return** false

# ĐPT THỜI GIAN CỦA THUẬT TOÁN

- **Giải** Gọi $\alpha$, $\beta$ và $\gamma$ là thời gian thực hiện của phép gán, phép so sánh và trả về của thuật toán
  - Trường hợp tốt nhất: Nếu $a_1 = x$, thì $T(n) = \alpha + \beta + \gamma$
  - Trường hợp xấu nhất: Nếu $x \notin \{a_1, a_2, ..., a_n\}$ thì

$$T(n) = (n+1)\alpha + n\beta + \gamma$$

# ĐPT THỜI GIAN CỦA THUẬT TOÁN

**Giải**

- Trường hợp trung bình: Tồn tại $i$, $a_i = x$, thì $T(n) = f(i) = i\alpha + i\beta + \gamma$ vời xác suất $p(i) = 1/n$

  $T(n) = [(\alpha + \beta + \gamma) + (2\alpha + 2\beta + \gamma) + ... + (n\alpha + n\beta + \gamma)]/n$

  $= [(n(n+1)(\alpha + \beta)/2 + n\gamma]/n = (n+1)(\alpha + \beta)/2 + \gamma$

✓ lưu ý: $T(n)$ là kỳ vọng (expected value) của $f(i) = \alpha i + \beta i + \gamma$ trên không gian xác suất $\{1, 2, ..., n\}$

# ĐPT THỜI GIAN CỦA THUẬT TOÁN

Giả sử $g$ là hàm không âm đối số nguyên dương $n$

- $T(n)$ có bậc không quá $g(n)$ và viết $T(n) = O(g(n))$ nếu có hằng số $c > 0$ và số nguyên dương $N$ sao cho $T(n) \le cg(n)$, $\forall\ n \ge N$

# ĐPT THỜI GIAN CỦA THUẬT TOÁN

**Ví dụ** Xét $T(n) = 20n^2 + 9n + 3$.

- $T(n) \leq 20n^2+9n^2+3n^2 = 32n^2, \forall\ n \geq 1$
- Vậy $T(n) = O(n^2)$, với $c = 32$, $N = 1$

# ĐPT THỜI GIAN CỦA THUẬT TOÁN

- Nếu $T_1(n) = O(f(n))$ và $T_2(n) = O(g(n))$ thì
  - $T_1(n) + T_2(n) = O(max(f(n), g(n)))$
  - $T_1(n).T_2(n) = O(f(n).g(n))$
  - $c.O(f(n)) = O(f(n))$
  - $O(c) \equiv O(1)$

# ĐPT THỜI GIAN CỦA THUẬT TOÁN

- Nếu thuật toán có thời gian chạy tốt nhất (trung bình, xấu nhất) là $T(n)$ và $T(n) = O(g(n))$ thì ta nói thời gian chạy tốt nhất (trung bình, xấu nhất) của thuật toán có bậc không quá $g(n)$ hay thời gian chạy tốt nhất (trung bình, xấu nhất) của thuật toán là $O(g(n))$

# ĐPT THỜI GIAN CỦA THUẬT TOÁN

Lưu ý:

- Bậc của thời gian chạy càng lớn thì thuật toán càng chậm (chẳng hạn, thuật toán có thời gian chạy $T(n) = O(n^2)$ sẽ kém hiệu quả hơn thuật toán có thời gian chạy $T(n) = O(n\lg n)$)

# CÁC VÍ DỤ

- **Ví dụ 1** Viết và phân tích thuật toán tính gần đúng $e^x$ theo khai triển $e^x \approx 1 + x/1! + x^2/2! + \ldots + x^n/n!$
- **Giải** ?

# CÁC VÍ DỤ

Exp($x$, $n$)

```
1   s ← 1
2   for i ← 1 to n
3       do  p ← 1
4           for j ← 1 to i
6               do p ←  p*x/j
5           s ← s + p
6   return s
```

# CÁC VÍ DỤ

Gọi $\alpha$, $\gamma$ là thời gian thực hiện lệnh gán và trả về, thì

- $T(n) = \alpha + n\alpha + (1+2+...+n)\alpha + n\alpha + \gamma$
- $T(n) = (2n + 1)\alpha + [n(n+1)/2]\alpha + \gamma$
- $T(n) = O(n^2)$

# CÁC VÍ DỤ

Exp(*x*, *n*) - Một thuật toán hiệu quả hơn

```
1  s ← 1
2  p ← 1
3  for i ← 1 to n
4      do  p ← p*x/i
5          s ← s + p
6  return s
```

# CÁC VÍ DỤ

Phân tích độ phức tạp thuật toán thứ 2

- $T(n) = 2\alpha + 2n\alpha + \gamma$
- $T(n) = 2(n + 1)\alpha + \gamma$
- $T(n) = O(n)$

# CÁC VÍ DỤ

- **Ví dụ 2** Viết và phân tích thuật toán tính giai thừa của số tự nhiên $n$
- **Giải** ?

# CÁC VÍ DỤ

```
Factorial(n)
1   if n = 0 or n = 1
2         then return 1
3         else if n > 1
4                  then return n*Factorial(n-1)
```

# CÁC VÍ DỤ

Gọi *T*(*n*) là thời gian chạy của thuật giải

$$T(n) = \begin{cases} O(1), & n = 0, 1 \\ T(n-1) + O(1), & n > 1 \end{cases}$$

# CÁC VÍ DỤ

$T(n) = T(n\text{-}1) + c$

$= T(n\text{-}2)+c+c = T(n\text{-}2) + 2c$

$= T(n\text{-}3)+c + 2c = T(n\text{-}3) + 3c$

....

$= T(n\text{-}(n\text{-}1)) + (n\text{-}1)c = T(1) + (n\text{-}1)c = c + (n\text{-}1)c = nc$

$T(n) = O(n)$ (tương đương với thuật toán không đệ qui)

# CÁC VÍ DỤ

Giả sử mỗi bước đòi hỏi 1 $\mu$s = $10^{-6}$sec thì với $n$ = 100

- $T(n) = O(n^3)$ có thời gian chạy $T(n) \approx 100^3.10^{-6} = 1$ (sec)
- $T(n) = O(2^n)$ có thời gian chạy $T(n) \approx 2^{100}.10^{-6}$(sec) =

  $2^{100}.10^{-6}$(sec)/(60.60.24.365) $\approx$ 4.106(năm)

# KẾT LUẬN VÀ LƯU Ý

- Nếu bài toán có thuật giải với thời gian chạy xấu nhất là đa thức, $O(n^m)$, thì bài toán gọi là được giải tốt
- Nếu bài toán không có thuật giải với thời gian chạy xấu nhất là đa thức thì bài toán gọi là khó giải (intractable)
- Nếu bài toán khó đến mức không thể xây dựng được thuật giải thì nó được gọi là không giải được (unsolvable problem)

# KẾT LUẬN VÀ LƯU Ý

- Phân tích độ phức tạp chủ yếu dựa trên kỹ thuật đếm và biểu diễn hệ thức truy hồi
- Phân tích trường hợp trung bình thường phức tạp hơn và cần thêm các công cụ toán học như lý thuyết xác suất, hàm sinh
- Trong nhiều trường hợp chỉ cần tính thời gian chạy xấu nhất

# CÁC CẤU TRÚC DỮ LIỆU CƠ BẢN
## (Elementary Data Structures)

- Chồng xếp (stack)
- Hàng đợi (queue)
- Danh sách liên kết đơn (singly linked list)
- Danh sách liên kết kép (double linked list)

# STACK

- Biểu diễn một tập động (dynamic set) mà thao tác chèn và xóa theo nguyên lý vào sau ra trước (last in, first out hay LIFO)
- Stack là mô hình dữ liệu được ứng dụng nhiều trong thực tế
- Trong khoa học máy tính stack được dùng để quản lý quá trình gọi và thực thi chương trình con

# CÁC THAO TÁC TRÊN STACK

- PUSH(S, x): Chèn vào stack S một phần tử
- POP(S): Loại khỏi stack một phần tử
- STACK_EMPTY(S): Kiểm tra stack rỗng
- STACK_FULL(S): Kiểm tra stack đầy

# BIỂU DIỄN STACK

- Có nhiều cách biểu diễn
- Có thể dùng mảng một chiều n phần tử S[1..n]

# BIỂU DIỄN STACK

- Ký hiệu top[S] là đỉnh stack (chỉ đến phần tử được chèn vào gần nhất)
- Tại một thời điểm stack bao gồm các phần tử S[1], S[2], ..., S[top[S]]
- Trong đó S[1] là phần tử ở đáy và S[top[S]] là phần tử ở đỉnh stack

# BIỂU DIỄN STACK

- Nếu top[S] = 0, stack rỗng
- Nếu top[S] > n, stack tràn
- Mỗi lần thao tác push thực hiện top[S] tăng lên 1
- Mỗi lần thao tác pop thực hiện top[S] giảm đi 1

# BIỂU DIỄN STACK

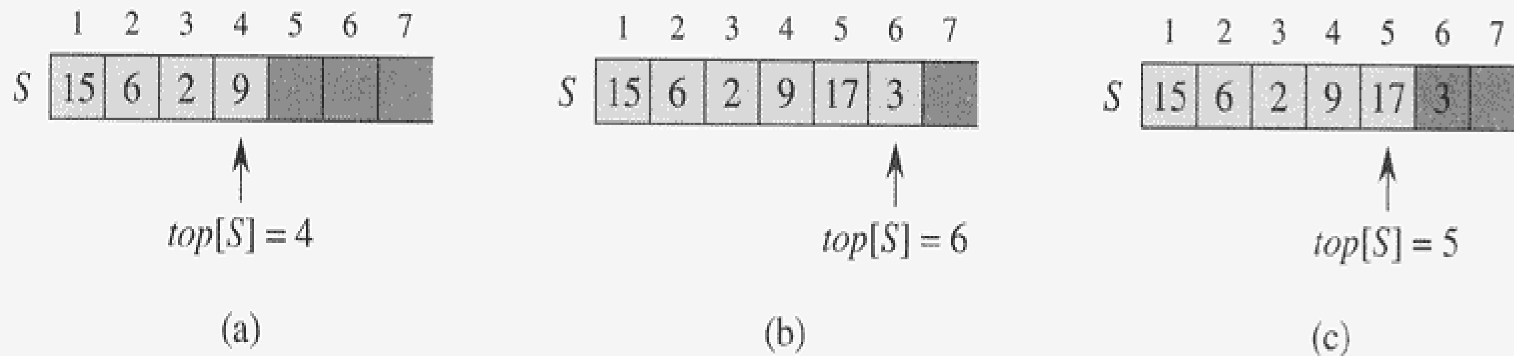



**Figure 10.1** An array implementation of a stack $S$. Stack elements appear only in the lightly shaded positions. **(a)** Stack $S$ has 4 elements. The top element is 9. **(b)** Stack $S$ after the calls PUSH($S$, 17) and PUSH($S$, 3). **(c)** Stack $S$ after the call POP($S$) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

# STACK-EMPTY

```
STACK-EMPTY(S)
1   if top[S] = 0
2       then return TRUE
3       else  return FALSE
```

# PUSH

PUSH$(S, x)$

1 $top[S] \leftarrow top[S] + 1$

2 $S[top[S]] \leftarrow x$

# POP

```
POP(S)
1  if STACK-EMPTY(S)
2      then error "underflow"
3      else top[S] ← top[S] − 1
4           return S[top[S] + 1]
```

# STACK-FULL

STACK-FULL($S$)

1 **if** $top[S] \geq n$

2 **then return** TRUE

3 **else return** FALSE

# ĐỘ PHỨC TẠP

- Thời gian thực hiện các thao tác là O(1)

# QUEUE

- Biểu diễn một tập động (dynamic set) mà thao tác chèn và xóa theo nguyên lý vào trước ra trước (First in, first out hay FIFO)
- Queue là mô hình dữ liệu được ứng dụng nhiều trong thực tế
- Queue có thể được ứng dụng trong quá trình quản lý và thực thi có trật tự các chương trình của một hệ điều hành

# CÁC THAO TÁC TRÊN QUEUE

- ENQUEUE(Q, x): Chèn vào queue một phần tử
- DEQUEUE(Q): Loại khỏi queue một phần tử
- QUEUE_EMPTY(Q): Kiểm tra queue rỗng
- QUEUE_FULL(Q): Kiểm tra queue đầy

# BIỂU DIỄN QUEUE

- Có nhiều cách biểu diễn
- Có thể dùng một mảng một chiều với n phần tử Q[1..n] để biểu diễn một hàng đợi tối đa n-1 phần tử

# BIỂU DIỄN QUEUE

- Mỗi hàng đợi có một đầu (head) và một đuôi (tail)
- Một phần tử được đưa vào hàng đợi tại đuôi của nó
- Phần tử bị loại khỏi hàng đợi là phần tử ở đầu hàng đợi

# BIỂU DIỄN QUEUE

- Ký hiệu head[Q] và tail[Q] là đầu và đuôi của hàng đợi
- Các phần tử trong hàng đợi được đặt tại các vị trí head[Q], head[Q]+1,..., tail[Q]-1 và coi vị trí 1 đi ngay sau vị trí n trong một trật tự "vòng"

# BIỂU DIỄN QUEUE

- head[Q] = tail[Q] thì hàng đợi rỗng (khởi tạo hàng đợi rỗng: head[Q] = tail[Q] = 1)
- Khi head[Q] = tail[Q] + 1 thì hàng đợi đầy

# BIỂU DIỄN QUEUE

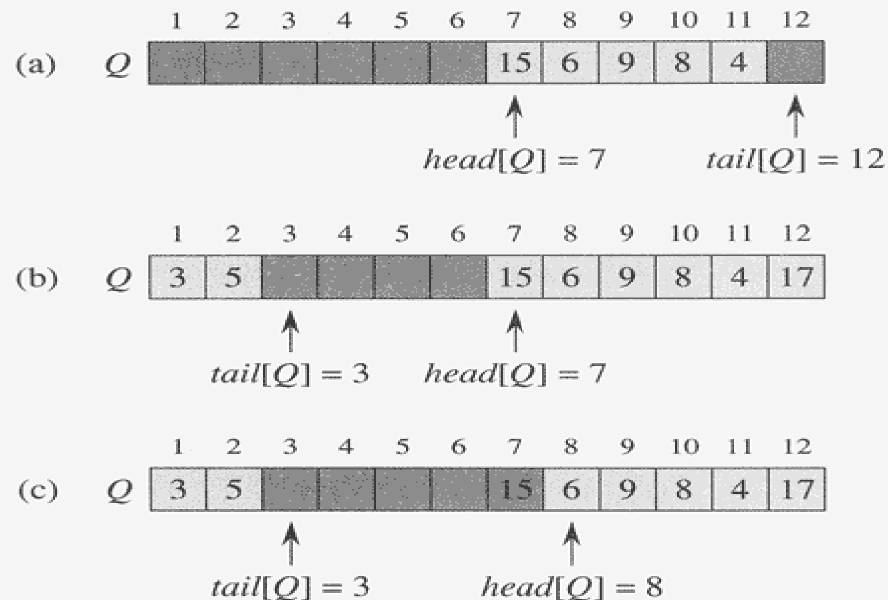



**Figure 10.2** A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE($Q$, 17), ENQUEUE($Q$, 3), and ENQUEUE($Q$, 5). **(c)** The configuration of the queue after the call DEQUEUE($Q$) returns the key value 15 formerly at the head of the queue. The new head has key 6.

# ENQUEUE

ENQUEUE$(Q, x)$

1 $Q[tail[Q]] \leftarrow x$
2 **if** $tail[Q] = length[Q]$
3 **then** $tail[Q] \leftarrow 1$
4 **else** $tail[Q] \leftarrow tail[Q] + 1$

# DEQUEUE

```
DEQUEUE(Q)
1   x ← Q[head[Q]]
2   if head[Q] = length[Q]
3       then head[Q] ← 1
4       else  head[Q] ← head[Q] + 1
5   return x
```

# ĐỘ PHỨC TẠP

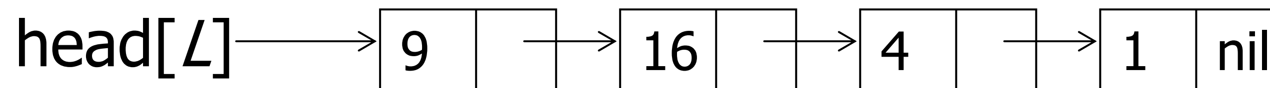
- Các thao tác trên hàng đợi đều có chi phí về thời gian là O(1)
- Lưu ý: Trong các thao tác ENQUEUE và DEQUEUE chưa có kiểm tra hàng đợi đầy và rỗng

# DANH SÁCH LIÊN KẾT ĐƠN

- Danh sách liên kết đơn là cấu trúc dữ liệu trong đó các đối tượng được sắp đặt theo một trật tự tuyến tính
- Trật tự tuyến tính trong danh sách liên kết được xác định bởi các pointer trong mỗi đối tượng

# DANH SÁCH LIÊN KẾT ĐƠN

- Ví dụ về một danh sách liên kết đơn




- Danh sách liên kết cung cấp một sự biểu diễn mềm dẻo và đơn giản cho các tập động, hỗ trợ các thao tác như tìm kiếm, chèn, xóa v.v

# DANH SÁCH LIÊN KẾT ĐƠN

- Mỗi đối tượng trong danh sách chứa một khóa (có thể có thông tin khác) và một pointer next
- Với một phần tử x, next[x] chỉ đến phần tử theo ngay sau nó

# DANH SÁCH LIÊN KẾT ĐƠN

- Nếu next[x]= NIL, thì x không có phần tử đứng sau nó, vì vậy x là phần tử cuối cùng còn gọi là đuôi của danh sách
- head[L] chỉ đến phần tử đầu tiên của danh sách, không có phần tử x mà next[x] chỉ đến phần tử đầu của danh sách
- Nếu head[L] = NIL thì danh sách L là rỗng

# CÁC THAO TÁC TRÊN DANH SÁCH

- LIST-SEARCH(L, k): Tìm kiếm một phần tử có khóa k
- LIST-INSERT(L, x): Chèn phần tử x vào danh sách
- LIST-DELETE(L, x): Xóa phần tử x khỏi danh sách

# LIST-SEARCH

- Thao tác LIST-SEARCH(L, k) tìm một phần tử có khóa k trong danh sách L
- Nếu có phần tử có khóa k, thủ tục sẽ trả về một pointer chỉ đến phần tử này
- Nếu không có đối tượng nào có khóa bằng k trong danh sách, thủ tục trả về NIL

# LIST-SEARCH

**LIST-SEARCH**($L$, $k$)

1 $x \leftarrow head[L]$

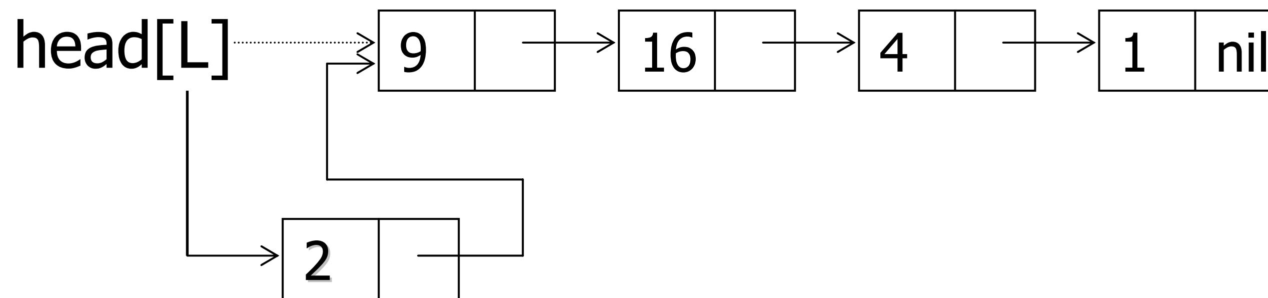
2 **while** $x \neq$ NIL and $key[x] \neq k$

3 **do** $x \leftarrow next[x]$

4 **return** $x$

# LIST-INSERT

- Thao tác LIST-INSERT(L, x) thực hiện đơn giản bằng cách chèn x vào đầu của L



# LIST-INSERT

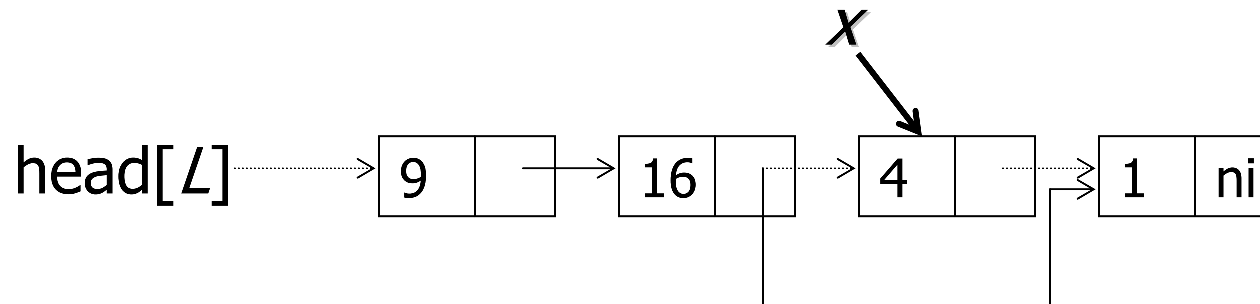
**LIST-INSERT(*L*, *x*)**

1 $next[x] \leftarrow head[L]$

2 $head[L] \leftarrow x$

# LIST-DELETE

- Thao tác LIST-DELETE(L, x) xóa x khỏi L
- Được thực hiện bằng cách cập nhật lại các con trỏ khi biết con trỏ tới x để loại x ra khỏi danh sách



# LIST-DELETE

```
LIST-DELETE(L, x)
1   if head[L] = x
2       then head_delele(L, x)
3       else  p ← head[L]
4             while p ≠ NIL and next[p] ≠ x
5                   do  p ← next[p]
6             if next[p]=x
7                then next[p] ← next[x]
```

# LIST-DELETE

- Lưu ý: Muốn loại một phần tử khi biết khóa, trước hết phải gọi thủ tục LIST-SEARCH để xác định con trỏ đến phần tử này

# ĐỘ PHỨC TẠP

- Thao tác tìm kiếm và xoá một phần tử mất tối đa O(n) nếu danh sách có n phần tử
- Thao tác chèn một phần tử chi phí là O(1)

# DANH SÁCH LIÊN KẾT KÉP

- Danh sách liên kết kép là cấu trúc dữ liệu trong đó các đối tượng được sắp đặt theo một trật tự tuyến tính
- Trật tự tuyến tính trong danh sách liên kết kép được xác định bởi một cặp pointer trong mỗi đối tượng

# DANH SÁCH LIÊN KẾT KÉP

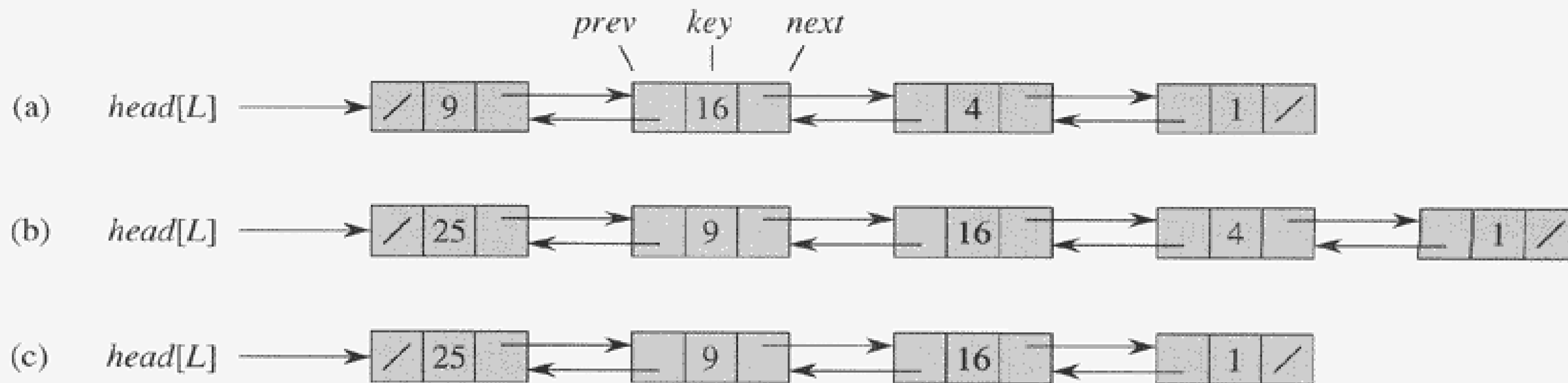



**Figure 10.3** **(a)** A doubly linked list $L$ representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with fields for the key and pointers (shown by arrows) to the next and previous objects. The *next* field of the tail and the *prev* field of the head are NIL, indicated by a diagonal slash. The attribute $head[L]$ points to the head. **(b)** Following the execution of LIST-INSERT$(L, x)$, where $key[x] = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE$(L, x)$, where $x$ points to the object with key 4.

# CÁC THAO TÁC TRÊN DSLK KÉP

- LIST-SEARCH(L, k): Tìm kiếm một phần tử có khóa k
- LIST-INSERT(L, x): Chèn phần tử x vào danh sách
- LIST-DELETE(L, x): Xóa phần tử x khỏi danh sách

# LIST-SEARCH

- Thao tác LIST-SEARCH(L, k) tìm một phần tử có khóa k trong danh sách L
- Nếu có phần tử có khóa k, thủ tục sẽ trả về một pointer chỉ đến phần tử này
- Nếu không có đối tượng nào có khóa bằng k trong danh sách, thủ tục trả về NIL

# LIST-SEARCH

```
LIST-SEARCH(L, k)
1  x ← head[L]
2  while x ≠ NIL and key[x] ≠ k
3      do x ← next[x]
4  return x
```

# LIST-INSERT

- Thao tác LIST-INSERT(L, x) thực hiện đơn giản bằng cách chèn x vào đầu của

# LIST-INSERT

```
LIST-INSERT(L, x)
1   next[x] ← head[L]
2   if head[L] ≠ NIL
3       then prev[head[L]] ← x
4   head[L] ← x
5   prev[x] ← NIL
```

# LIST-DELETE

- Thao tác LIST-DELETE(L, x) xóa x khỏi L
- Được thực hiện bằng cách cập nhật lại các con trỏ khi biết con trỏ tới x để loại x ra khỏi danh sách

# LIST-DELETE

```
LIST-DELETE(L, x)
1  if prev[x] ≠ NIL
2      then next[prev[x]] ← next[x]
3      else head[L] ← next[x]
4  if next[x] ≠ NIL
5      then prev[next[x]] ← prev[x]
```

# ĐỘ PHỨC TẠP

- Thao tác tìm kiếm một phần tử mất tối đa O(n) nếu danh sách có n phần tử
- Thao tác chèn và xóa một phần tử chi phí là O(1)

# BIỂU DIỄN CÂY CÓ GỐC

- Cây nhị phân (tham khảo - Introduction to Algorithms)
- Cây đa phân  (tham khảo - Introduction to Algorithms)

# BẢNG BĂM
## (Hash Table)

- Bảng địa chỉ trực tiếp (direct-address table)
- Bảng băm (Hash table)
- Hàm băm (Hash function)
- Giải quyết đụng độ bằng PP địa chỉ mở

# BẢNG ĐỊA CHỈ TRỰC TIẾP

- Bảng địa chỉ trực tiếp (direct-address table) lưu trữ một tập có khóa trong U={0, 1, ..., m-1} là một mảng T[0..m-1] mà mỗi ví trí (slot) tương ứng với một khoá trong U
- Một áp dụng cần một tập các phần tử không vượt quá m và m không quá lớn thì có thể dùng một bảng địa chỉ trực tiếp

# BẢNG ĐỊA CHỈ TRỰC TIẾP

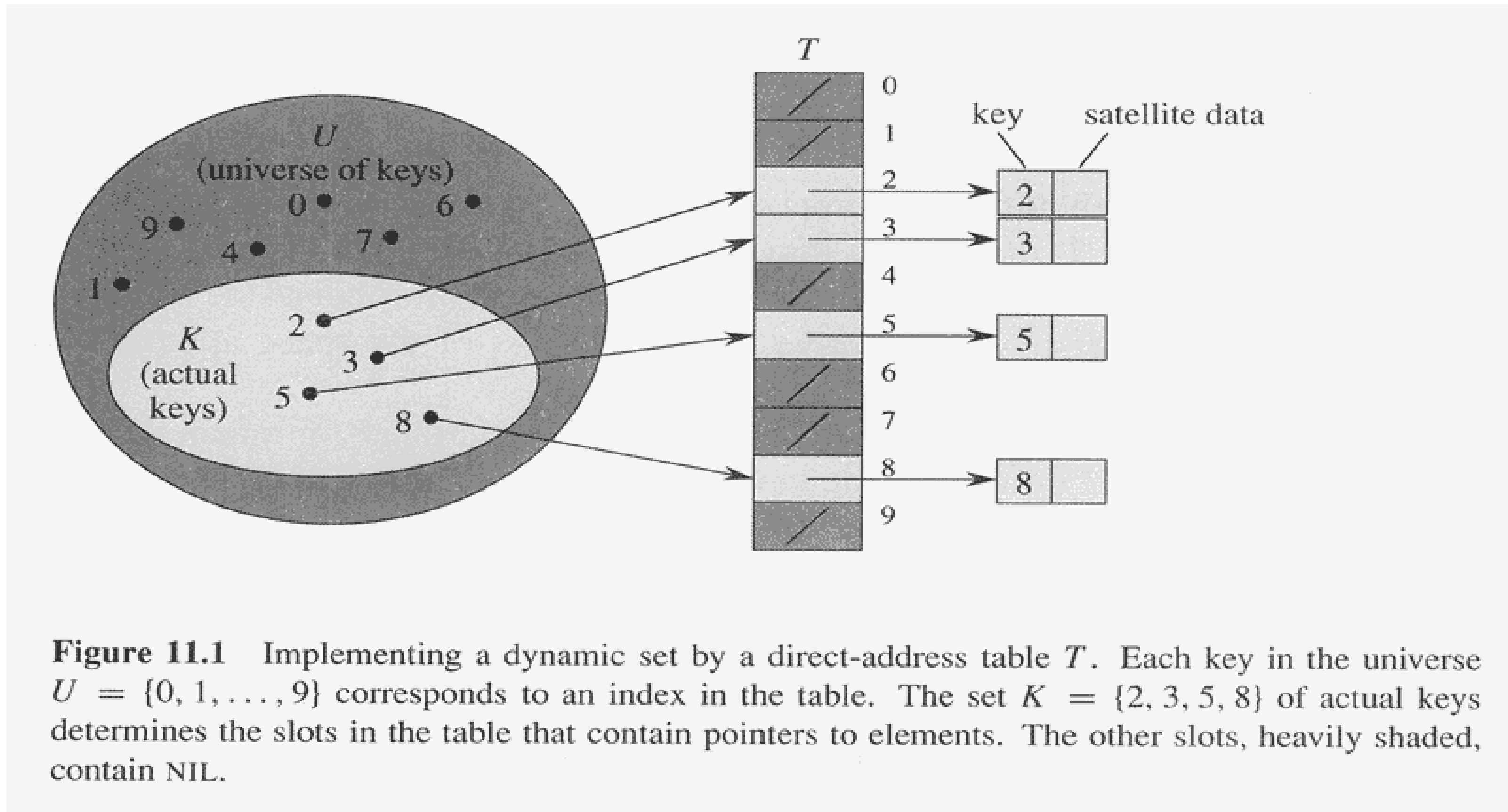



**Figure 11.1** Implementing a dynamic set by a direct-address table $T$. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

# BẢNG ĐỊA CHỈ TRỰC TIẾP

- Slot k ứng với một phần tử trong tập có khóa k
- Nếu tập không chứa phần tử có khoá k thì T[k]=NIL

# CÁC THAO TÁC

- DIRECT-ADDRESS-SEARCH(T, k): Tìm kiếm một phần tử có khoá k trên bảng địa chỉ trực tiếp T
- DIRECT-ADDRESS-INSERT(T, x): Chèn một phần tử x vào bảng địa chỉ trực tiếp
- DIRECT-ADDRESS-DELETE(T, x): Xoá phần tử x khỏi bảng địa chỉ trực tiếp

# DIRECT-ADDRESS-SEARCH

```
DIRECT-ADDRESS-SEARCH(T, k)
  return T[k]
```

# DIRECT-ADDRESS-INSERT

DIRECT-ADDRESS-INSERT( $T$, $x$)

$T[key[x]] \leftarrow x$

# DIRECT-ADDRESS-DELETE

DIRECT-ADDRESS-DELETE( *T*, *x*)

*T*[*key*[*x*]]←NIL

# ĐỘ PHỨC TẠP

- Thời gian thực hiện các thao tác là O(1)
- Lưu ý
  - Là cấu trúc dữ liệu tốt đối với tập kích thước nhỏ
  - Nếu số phần tử của tập biến động lớn dùng bảng này sẽ lảng phí nhiều bộ nhớ và có thể không khả thi

# BẢNG BĂM

- Bảng băm là một cấu trúc dữ liệu có thể lưu trữ một tập các đối tượng có số phần tử tùy ý
- Các thao tác tìm kiếm, chèn, xoá trên bảng băm rất hiệu quả
- Phần tử có khoá k được lưu trữ trong slot h(k), trong đó h là hàm băm (hash function) từ tập U các khóa đến tập các slot của bảng băm T[0..m-1]

# BẢNG BĂM

- Hàm băm có dạng h: U $\rightarrow$ {0,1,..., m-1}
- h(k) là giá trị băm (hash value) của khoá k hay còn gọi phần tử có khoá k băm slot h(k)
- Với hàm băm chỉ cần xử lý m-1 giá trị thay vì |U| giá trị

# BẢNG BĂM

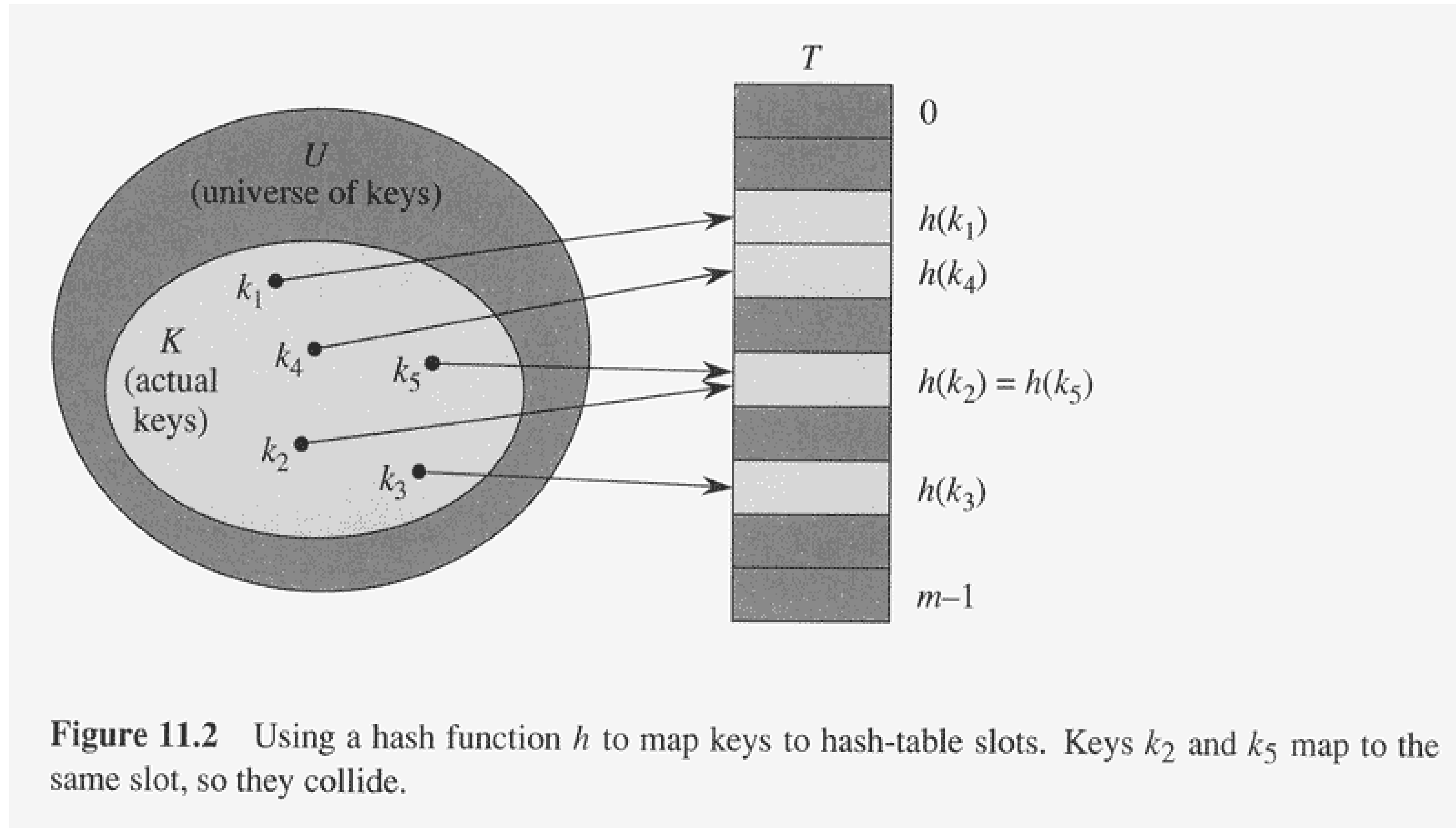



**Figure 11.2** Using a hash function $h$ to map keys to hash-table slots. Keys $k_2$ and $k_5$ map to the same slot, so they collide.

# BẢNG BĂM

- Có thể có sự dụng độ (collision) lưu trữ các khóa, ví dụ $k_2$ dụng độ $k_5$, nghĩa là $h(k_2)= h(k_5)$
- Các phương pháp giải quyết đụng độ
  - Thiết kế hàm băm h(U) phân bố đều trên T (không triệt để)
  - Giải quyết dụng độ bằng kết nối (collision resolution by chaining)
  - Giải quyết dụng độ bằng địa chỉ mở (open addressing)

# GIẢI QUYẾT DỤNG ĐỘ BẰNG KẾT NỐI

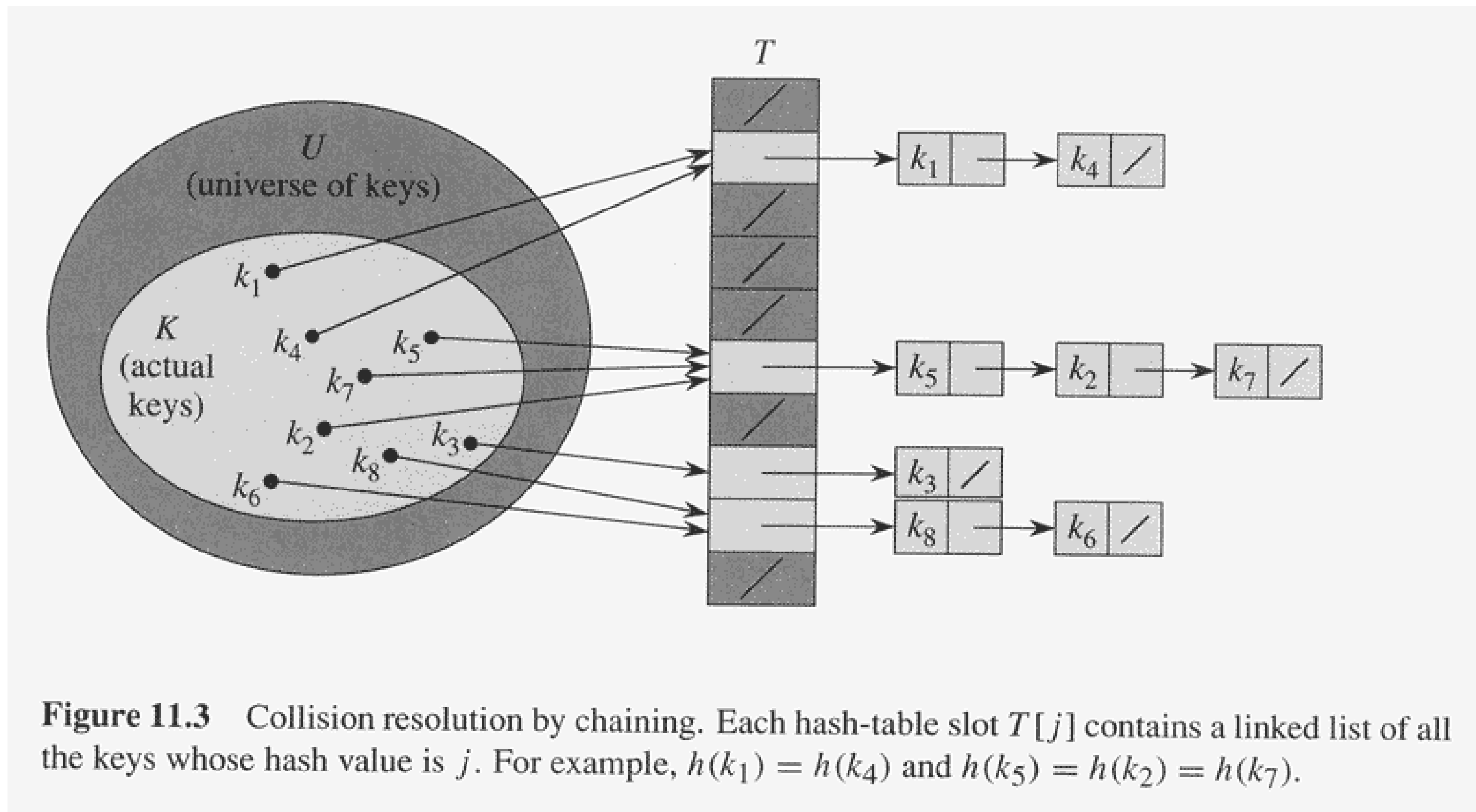



**Figure 11.3** Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is $j$. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

# GIẢI QUYẾT DỤNG ĐỘ BẰNG KẾT NỐI

- Đặt tất cả các phần tử có cùng giá trị hàm băm vào một danh sách liên kết
- Slot j chứa pointer chỉ đến đầu của danh sách liên kết của tất cả các phần tử được lưu trữ có hàm băm là j
- Nếu không có khóa k để h(k)=j thì thì slot j trỏ đến NIL

# CÁC THAO TÁC

- CHAINED-HASH-INSERT(T, x): Chèn một phần tử vào bảng băm T
- CHAINED-HASH-SEARCH(T, k): Tìm một phần tử có khoá k trong T
- CHAINED-HASH-DELETE(T, x): Xoá một phần tử khỏi T

# CHAINED-HASH-INSERT

- CHAINED-HASH-INSERT(*T*, *x*)

  Insert *x* at the head of list *T*[*h*(*key*[*x*])]

- Thời gian thực hiện là O(1) (chèn vào đầu danh sách)

# CHAINED-HASH-SEARCH

- CHAINED-HASH-SEARCH(*T*, *k*)

  Search for an element with key *k* in list *T*[*h*(*k*)]

- Thời gian trung bình là $O(1+\alpha)$, trong đó $\alpha = n/m$ với m, n lần lượt là số slot và số phần tử được lưu trữ trong bảng, $\alpha$ gọi là hệ số tải (load factor) của T

# CHAINED-HASH-DELETE

- CHAINED-HASH-DELETE( $T$, $x$)

  Delete $x$ from the list $T[h(key[x])]$

- Thời gian thực hiện trung bình là O(n/m)

# HÀM BĂM

- Một hàm băm là tốt nếu h(k) có thể là bất kỳ slot nào trên bảng băm
- Luôn giả sử mỗi khoá là một số tự nhiên
- Có một số cách xây dựng hàm băm như: phương pháp chia (division method), phương pháp nhân (multiplication method) v.v

# HÀM BĂM

- Với phương pháp chia, tạo hàm băm bằng cách đặt tương ứng mỗi khoá k với số dư trong phép chia k cho số slot m
- Vậy h(k) = k mod m
- Tập các khoá được chia thành m lớp
- Nên chọn m là số nguyên tố không quá gần với lũy thừa của 2

# HÀM BĂM

- Phương pháp nhân, hàm băm có dạng:
  - $h(k) = \lfloor m(k\ A \bmod 1) \rfloor$, A là hằng thỏa $0 < A < 1$
  - Với $k\ A \bmod 1 = kA - \lfloor k\ A \rfloor$
- Phương pháp này không hạn chế việc chọn m

# HÀM BĂM

- Ví dụ A = $(\sqrt{5}-1)/2 \approx$ 0.61803 và k =123456, m =10000,

  Thì

$$
\begin{aligned}
h(k) &= \lfloor 10000(123456 \times 0.61803 \bmod 1) \rfloor \\
&= \lfloor 10000 \times 0.0041151 \rfloor \\
&= \lfloor 41.151 \rfloor \\
&= 41
\end{aligned}
$$

# HÀM BĂM

- Lưu ý:
  - Trong ứng dụng cần tìm các hàm băm thích hợp
  - Còn có phương pháp khác để giải quyết dụng độ khóa, ví dụ phương pháp địa chỉ mở (open addressing-tham khảo)

# GIẢI QUYẾT ĐỤNG ĐỘ BẰNG ĐỊA CHỈ MỞ

- Trong phương pháp địa chỉ mở tất cả các phần tử được lưu trữ ngay trong bảng băm
- Hệ số tải $\alpha = n/m \leq 1$
- Mỗi slot của bảng chứa một phần tử của tập hoặc NIL

# GIẢI QUYẾT ĐỤNG ĐỘ BẰNG ĐỊA CHỈ MỞ

- Để tìm một phần tử, kiểm tra một cách có hệ thống các slot của bảng cho đến khi tìm được nó hoặc biết chắc chắn nó không có trong bảng
- Để chèn một phần tử vào bảng, kiểm tra lần lượt bảng băm cho đến khi tìm được một slot rỗng và chèn nó vào slot này
- Thay vì thăm dò theo trật tự 0, 1, ..., $m$-1, dãy các vị trí được thăm dò phụ thuộc vào khoá đang được chèn vào bảng

# GIẢI QUYẾT ĐỤNG ĐỘ BẰNG ĐỊA CHỈ MỞ

- Để xác định các slot được dò, mở rộng hàm băm sao cho hàm cả số thăm dò (bắt đầu từ 0) như là đối số thứ hai của hàm
- Hàm băm có dạng **$h$: $U \times$ {0,1,..., $m$-1} $\rightarrow$ {0,1,..., $m$-1}**
- Với mỗi *k*, dãy các vị trí cần thăm dò là ($h(k, 0)$, $h(k, 1)$,...,$h(k, m-1)$)
- Dãy này là một hoán vị của (0, 1,..., $m$-1), vị trí đầu tiên còn trống sẽ được chèn khoá *k* vào

# GIẢI QUYẾT ĐỤNG ĐỘ BẰNG ĐỊA CHỈ MỞ

- Trong các thao tác chèn và xoá, giả sử các phần tử chỉ chứa khoá (không có các thông tin khác kèm theo)
- Thủ tục tìm kiếm một phần tử theo khoá *k* sẽ dò theo dãy đã chèn vào bảng. Vì vậy nó có thể kết thúc không thành công nếu gặp một slot rỗng (phần tử có khoá *k* không thể được chèn sau slot rỗng này).
- Dãy các slot được dò tìm (trong cả tìm kiếm và chèn, xoá) phụ thuộc vào dạng của hàm băm. Có một số phương pháp thăm dò như: Thăm tuyến tính (linear probing), thăm bậc hai (quadratic probing) và băm kép (double hashing)

# GIẢI QUYẾT ĐỤNG ĐỘ BẰNG ĐỊA CHỈ MỞ

- Chèn một phần tử vào bảng

```
HASH-INSERT(T, k)
1   i ← 0
2   repeat  j ← h(k, i)
3           if T[j] = NIL
4              then T[j] ← k
5                   return j
6              else  i ← i + 1
7     until i = m
8   error "hash table overflow"
```

# GIẢI QUYẾT ĐỤNG ĐỘ BẰNG ĐỊA CHỈ MỞ

- Tìm một phần tử

```
HASH-SEARCH(T, k)
1  i ← 0
2  repeat j ← h(k, i)
3         if T[j] = k
4            then return j
5         i ← i + 1
6    until T[j] = NIL or i = m
7  return NIL
```

# GIẢI QUYẾT ĐỤNG ĐỘ BẰNG ĐỊA CHỈ MỞ

- Phụ thuộc vào hàm băm sẽ có các phương pháp tìm kiếm sau:
  - Tìm kiếm tuyến tính
  - Tìm kiếm bậc hai
  - Tìm kiếm băm kép

# TÌM TUYẾN TÍNH

- Thứ tự tìm có tính chất tuyến tính
- Hàm băm có dạng: $h(k, i) = (h'(k) + i) \bmod m$, $i = 0, 1, \ldots, m-1$ và $h' : U \rightarrow \{0, 1, \ldots, m-1\}$ là một hàm băm thông thường
- Với khoá k dãy các slot cần thăm dò là $T[h'(k)]$, $T[h'(k)+1], \ldots, T[m-1]$
- Tìm tuyến tính chưa phải là phương pháp tốt (vì các khoá có thể tụ lại từng đoạn trong bảng)

# TÌM BẬC HAI

- Thứ tự tìm được gia một hàm bậc hai
- Hàm băm có dạng: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, trong đó $i = 0, 1, \ldots, m-1$; $h' : U \rightarrow \{0,1,\ldots, m-1\}$ là một hàm băm thông thường và $c_1$, $c_2$ là các hằng khác 0
- Tìm bậc hai khắc phục được yếu điểm của tìm tuyến tính nhưng hạn chế ở chổ các giá trị băm có thể không lấp đầy các slot của bảng

# BĂM KÉP

- Thứ tự thăm được gia một hàm băm
- Hàm băm có dạng: $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$, trong đó $i = 0, 1, \ldots, m-1$; $h_1$ và $h_2$ là các hàm băm thông thường
- Khởi đầu vị trí được dò là $T[h_1(k)]$, các vị trí tiếp theo được gia thêm $h_2(k)$
- Phương pháp băm kép là tốt nhất

# VÍ DỤ BĂM KÉP

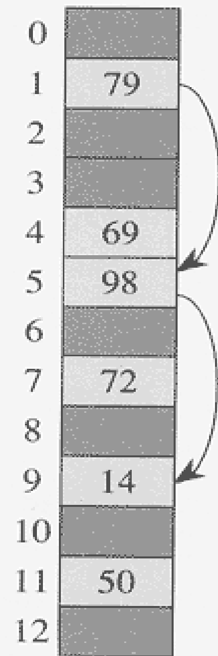



**Figure 11.5** Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

# ĐỘ PHỨC TẠP

- Một bảng băm địa chỉ mở với hệ số tải $\alpha = n/m < 1$, thì số các bước thăm dò trong một tìm kiếm không thành công nhiều nhất là $1/(1- \alpha)$, giả sử quá trình băm là chuẩn.
- Chứng minh?

# ĐỘ PHỨC TẠP

- Một bảng băm địa chỉ mở với hệ số tải $\alpha = n/m < 1$, thì số các bước thăm dò trong một tìm kiếm thành công nhiều nhất là:

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}+\frac{1}{\alpha}$$

- Chứng minh?

# CÂY NHỊ PHÂN TÌM KIẾM
## (BINARY SEARCH TREE-BST)

- Khái niệm cây
- Duyệt cây nhị phân
- Cây BST
- Các thao tác trên cây BST
- Cây BST ngẫu nhiên (randomly BST)
- Biểu diễn cây đa phân

# KHÁI NIỆM CÂY

- Cây là một tập hữu hạn các nút trong đó có một nút đặc biệt gọi là gốc (root)
- Giữa các nút có quan hệ phân cấp gọi là "quan hệ cha con"

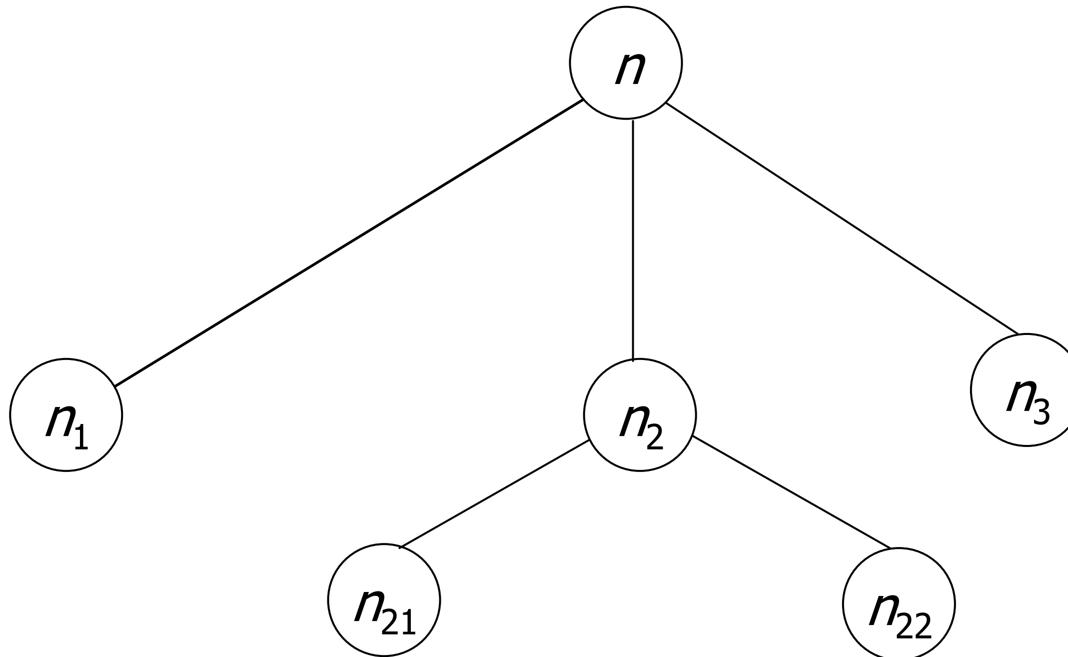
# KHÁI NIỆM CÂY

- Cây được định nghĩa một cách đệ qui:
  - Một nút là một cây, nút đó cũng là gốc của cây
  - Nếu n là một nút và $T_1$, $T_2$, ..., $T_k$ là các cây với $n_1$, $n_2$, ..., $n_k$ lần lượt là các gốc thì một cây mới T sẽ được tạo lập bằng cách cho n trở thành cha của các nút $n_1$, $n_2$, ..., $n_k$
  - Lúc này n là gốc còn $T_1$, $T_2$, ..., $T_k$ là các cây con (subtrees) của gốc ($n_1$, $n_2$, ..., $n_k$ là con của nút n)

# KHÁI NIỆM CÂY

- Qui ước cho phép tồn tại cây không có nút nào và gọi đó là cây rỗng (null tree)
- Cây mà mỗi nút chỉ có tối đa hai con gọi là cây nhị phân

# KHÁI NIỆM CÂY



# BIỂU DIỄN CÂY NHỊ PHÂN

- Biểu diễn cây nhị phân
  - Có thể dùng ba biến p, left, right để lưu trữ các pointer chỉ đến cha, con trái và con phải của mỗi nút trong cây nhị phân T
  - Nếu p[x] = NIL thì x là nút gốc.
  - Nếu left[x] = NIL thì x không có con trái
  - Nếu right[x] = NIL thì x không có con phải
  - Nút gốc của cây T được trỏ bởi root[T]
  - Nếu root[T] = NIL thì cây T là rỗng

# BIỂU DIỄN CÂY NHỊ PHÂN

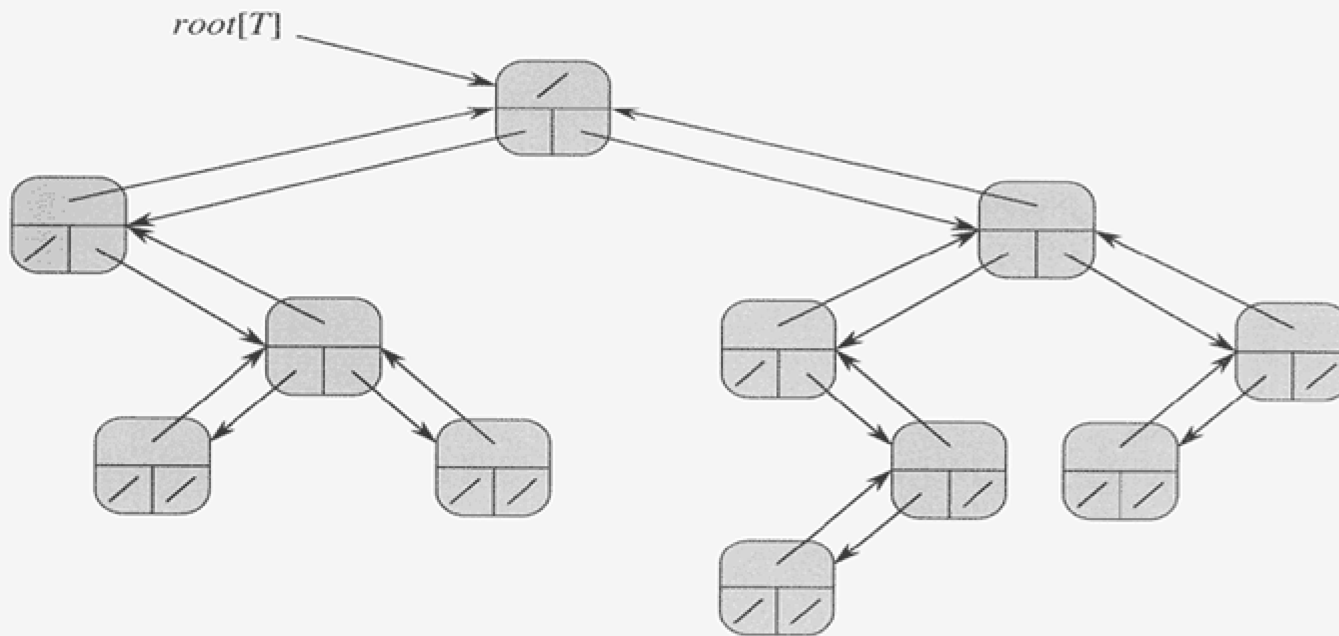



**Figure 10.9** The representation of a binary tree $T$. Each node $x$ has the fields $p[x]$ (top), $left[x]$ (lower left), and $right[x]$ (lower right). The *key* fields are not shown.

# BIỂU DIỄN CÂY NHỊ PHÂN

Cấu trúc cây nhị phân trong C++

- Coi mỗi nút của cây biểu diễn một đối tượng có khóa là một số nguyên (kiểu int)
- Cấu trúc dữ liệu cây nhị phân có thể được định nghĩa dựa trên các pointer chỉ đến cha và các con như sau

```
typedef struct CELL *TREE;
struct CELL {
        int key;
        TREE p, left, right;
};
```

# DUYỆT CÂY NHỊ PHÂN

- Có ba thứ tự duyệt cơ bản
  - Duyệt theo thứ tự giữa (inorder tree walk)
  - Duyệt theo thứ tự trước (preorder tree walk)
  - Duyệt theo thứ tự sau (postorder tree walk)
- Các thủ tục duyệt cây chi phí O(n) trên cây có n nút

# DUYỆT CÂY NHỊ PHÂN

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      then INORDER-TREE-WALK(left[x])
3           print key[x]
4           INORDER-TREE-WALK(right[x])
```

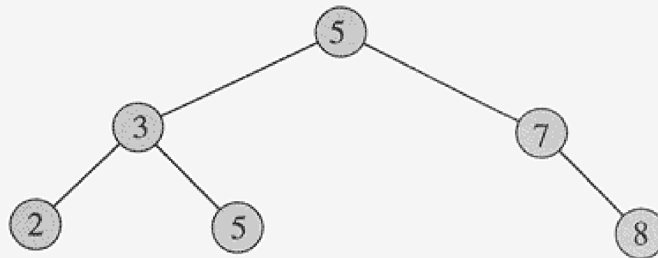
# CÂY BST

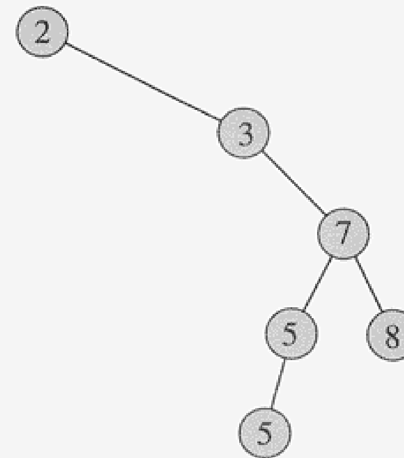
- Là mô hình dữ liệu có nhiều ứng dụng trong thực tế
- Khoá của các phần tử được lưu trữ thỏa mãn tính chất cây nhị phân tìm kiếm (binary-search-tree-property)
  - Nếu y là một nút trong cây con trái của nút x thì

    key[y]< key[x]
  - Nếu y là một nút trong cây con phải của nút x thì

    key[y]> key[x]

# CÂY BST




(a)

(b)

**Figure 12.1** Binary search trees. For any node $x$, the keys in the left subtree of $x$ are at most $key[x]$, and the keys in the right subtree of $x$ are at least $key[x]$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

# CÁC THAO TÁC TRÊN CÂY BST

- Các thao tác tìm kiếm
- Các thao tác chèn và xóa

# CÁC THAO TÁC TÌM KIẾM

- TREE-SEARCH: Tìm một phần tử theo khoá cho trước
- TREE-MINIMUM: Tìm phần tử có khoá nhỏ nhất
- TREE-MAXIMUM: Tìm phần tử có khoá lớn nhất
- TREE-SUCCESSOR: Tìm phần tử đi sau một phần tử
- TREE-PREDECESSOR: Tìm phần tử đi trước một phần tử

# CÁC THAO TÁC TÌM KIẾM

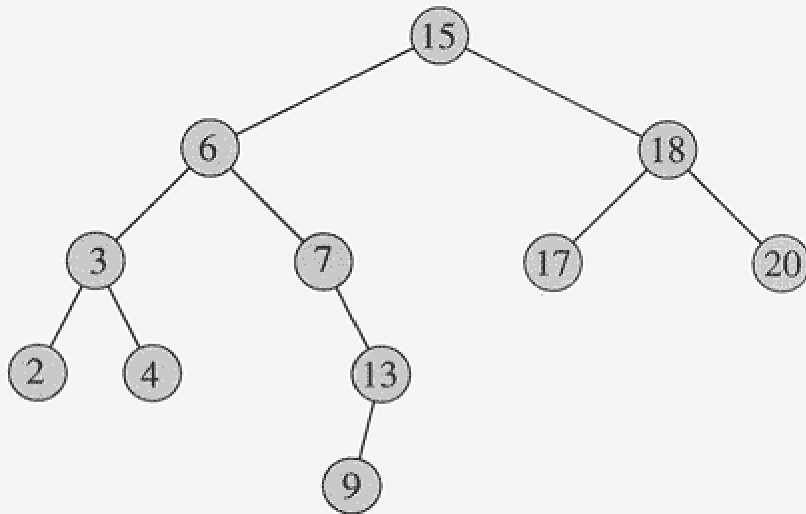



**Figure 12.2** Queries on a binary search tree. To search for the key 13 in the tree, we follow the path 15 → 6 → 7 → 13 from the root. The minimum key in the tree is 2, which can be found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

# TREE-SEARCH

- Đầu vào là một pointer x chỉ đến gốc của cây nhị phân tìm kiếm và khoá k của phần tử cần tìm
- Thủ tục trả về pointer chỉ đến nút có khoá k hoặc NIL
- Dựa vào tính chất của cây nhị phân tìm kiếm để xác định nút nào phải tìm kế tiếp trong các con trái và phải của x
  - Nếu k < key[x], tìm tiếp trong cây con trái
  - Ngược lại, tìm tiếp trong cây con trái

# TREE-SEARCH

```
TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2      then return x
3  if k < key[x]
4      then return TREE-SEARCH(left[x], k)
5      else  return TREE-SEARCH(right[x], k)
```

# TREE-SEARCH

```
ITERATIVE-TREE-SEARCH(x, k)
1   while x ≠ NIL and k ≠ key[x]
2         do if k < key[x]
3               then x ← left[x]
4               else  x ← right[x]
5   return x
```

# TREE-SEARCH

- Thời gian chạy của TREE-SEARCH(x, k) trong trường hợp xấu nhất là O(h), trong đó h là chiều cao của cây
- Trường hợp tốt nhất có thể là O(1)

# TREE-MINIMUM

- Đầu vào là một pointer x chỉ đến gốc của cây nhị phân tìm kiếm
- Thủ tục trả về pointer chỉ đến nút có khoá nhỏ nhất
- Dựa vào tính chất của cây nhị phân tìm kiếm, phần tử có khoá nhỏ nhất được tìm kiếm theo các pointer chỉ đến con trái bắt đầu từ gốc cho đến khi gặp một NIL

# TREE-MINIMUM

```
TREE-MINIMUM(x)
1    while left[x] ≠ NIL
2         do x ← left[x]
3    return x
```

T(n)=O(h), h là chiều cao của cây có n nút

# TREE-MAXIMUM

- Đầu vào là một pointer x chỉ đến gốc của cây nhị phân tìm kiếm
- Thủ tục trả về pointer chỉ đến nút có khoá lớn nhất
- Phần tử có khoá nhỏ nhất được tìm kiếm theo các pointer chỉ đến con phải bắt đầu từ gốc cho đến khi gặp một NIL

# TREE-MAXIMUM

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2       do x ← right[x]
3  return x
```

T(n)=O(h), h là chiều cao của cây có n nút

# TREE-SUCCESSOR

- Phần tử đi sau của x là nút có khoá nhỏ nhất lớn hơn key[x]
- Đầu vào là một nút x trên cây nhị phân tìm kiếm
- Thủ tục trả về pointer chỉ đến nút đi sau x hoặc NIL nếu x có khoá lớn nhất trong cây
- Thủ tục tìm kiếm dựa vào TREE-MINIMUN(right[x]) và tính chất của cây nhị phân tìm kiếm

# TREE-SUCCESSOR

- Thủ tục được chia làm hai trường hợp
  - Nếu cây con bên phải của x khác rỗng thì thực hiện lời gọi thủ tục TREE-MINIMUN(right[x])
  - Ngược lại, nếu y đi sau x thì y là tổ tiên gần nhất của x mà con trái của nó cũng là tổ tiên của x, tìm y bằng cách đi về gốc cho tới khi gặp nút đầu tiên có con trái

# TREE-SUCCESSOR

```
TREE-SUCCESSOR(x)
1   if right[x] ≠ NIL
2       then return TREE-MINIMUM(right[x])
3   y ← p[x]
4   while y ≠ NIL and x = right[y]
5         do x ← y
6            y ← p[y]
7   return y
```

T(n)=O(h), h là chiều cao của cây có n nút

# TREE-PREDECESSOR

- Phần tử đi trước của x là nút có khoá lớn nhất nhỏ hơn key[x]
- Thủ tục này tương tự (đối xứng) với TREE-SUCCESSOR(x)
- Thuật toán ?

# CHÈN VÀ XOÁ

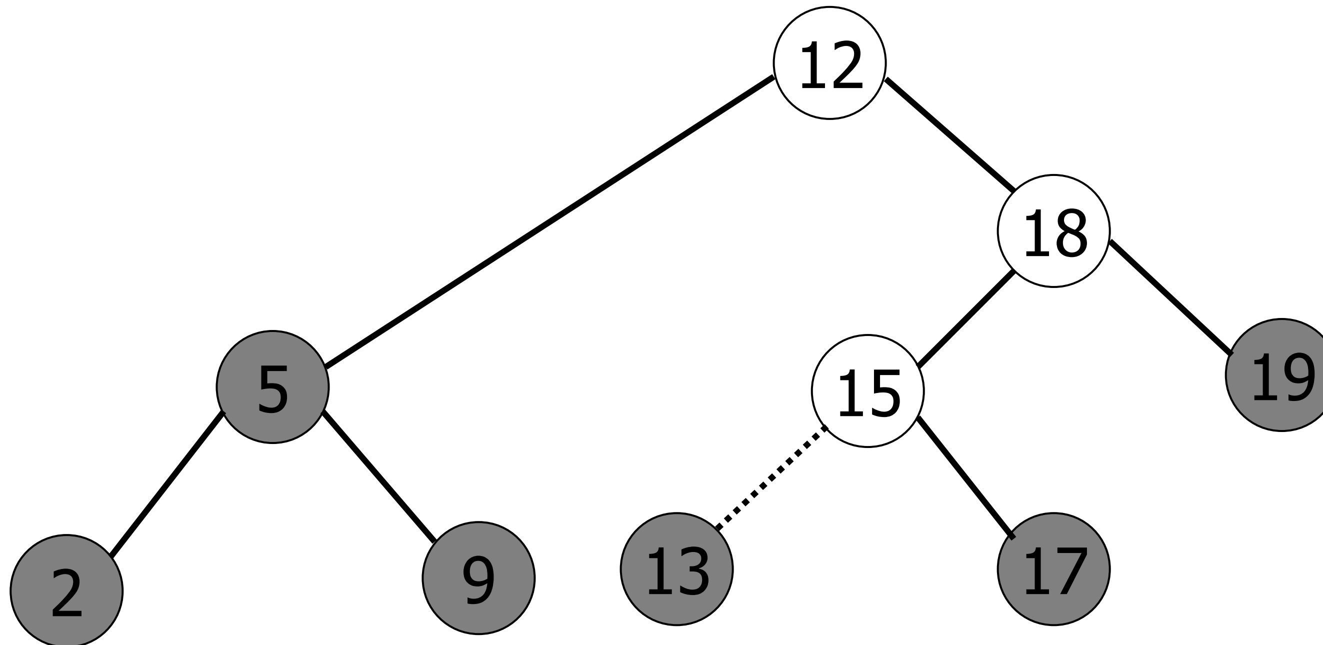
- TREE-INSERT: Chèn một nút vào cây
- TREE-DELETE: Xóa một nút trên cây

# TREE-INSERT

- Thủ tục TREE-INSERT(T, z) tìm một vị trí thích hợp để chèn nút z có khóa key[z] vào cây T sao cho left[z] = NIL và right[z] = NIL
- Cây T tăng một nút và vẫn đảm bảo là cây nhị phân tìm kiếm

# TREE-INSERT

Chèn một phần tử có khoá 13 vào cây, các nút được làm sáng chỉ đường đi từ gốc đến vị trí cần chèn



# TREE-INSERT

```
TREE-INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4       do y ← x
 5          if key[z] < key[x]
 6             then x ← left[x]
 7             else x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z                    ▷ Tree T was empty
11     else if key[z] < key[y]
12             then left[y] ← z
13             else right[y] ← z
```

# TREE-INSERT

- Thủ tục chèn z bắt đầu từ gốc của cây theo một đường đi xuống, pointer x giữ vết đường đi và pointer y duy trì như là cha xủa x
- Vòng lặp 3-7 tạo ra hai pointer di chuyển xuống theo cây trái hoặc phải tuỳ theo sự so sánh key[z] và key[x] cho đến khi x đạt được bằng NIL, z được đặt vào vị trí NIL này

# TREE-INSERT

- Dòng 8-13 đặt lại các pointer để z được chèn vào cây
- Thời gian thực hiện thủ tục là O(h)

# TREE-DELETE

- TREE-DELETE(T, z) xoá z có key[z] = v, xét ba trường hợp
  - Nếu z không có con, sửa cha của z là p[z] để nó có con là NIL
  - Nếu z chỉ có một con, loại bỏ z bằng một liên kết mới giữa con và cha của nó
  - Nếu z có hai con, loại phần tử y đi sau của z và thay thế nội dung của z bởi nội dung của y

# TREE-DELETE

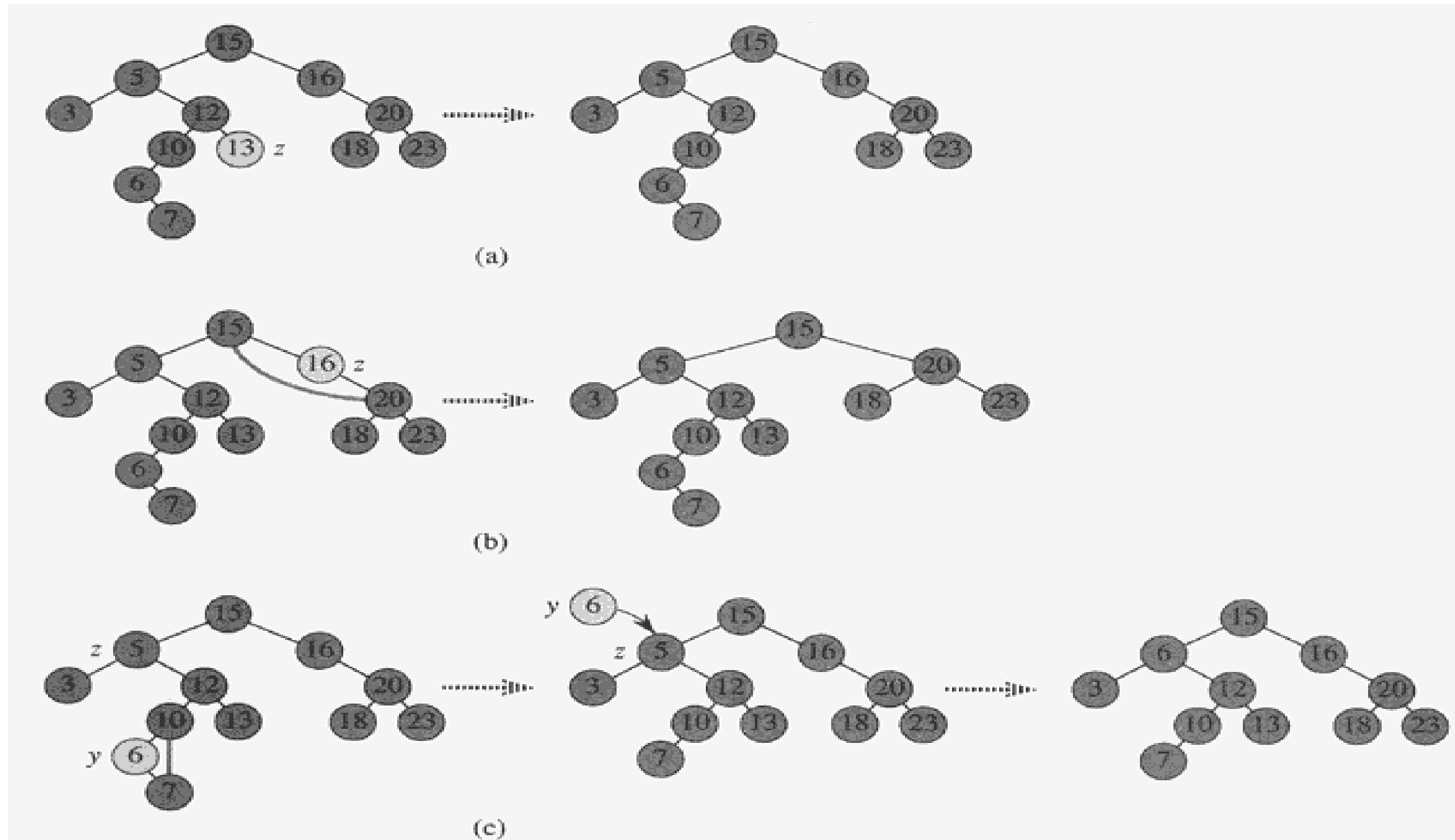



**Figure 12.4** Deleting a node $z$ from a binary search tree. Which node is actually removed depends on how many children $z$ has; this node is shown lightly shaded. **(a)** If $z$ has no children, we just remove it. **(b)** If $z$ has only one child, we splice out $z$. **(c)** If $z$ has two children, we splice out its successor $y$, which has at most one child, and then replace $z$'s key and satellite data with $y$'s key and satellite data.

# TREE-DELETE

# TREE-DELETE

- Dòng 1-3, xác định một nút y để loại bỏ, nút y là z (nếu z có nhiều nhất 1 con) hoặc là đi sau của z (nếu z có 2 con)
- Dòng 4-6, x được gán đến con $\neq$ NIL của y hoặc NIL nếu y không có con

# TREE-DELETE

- Nút y bị loại bỏ trong dòng 7-13 bằng cách sửa các pointer trong p[y] và x, với việc xét các điều kiện biên
- Trong dòng 14-16, nếu đi sau của z phải loại bỏ thì nội dung của z được di chuyển từ y đến và ghi đè lên nội dung trước đó của z
- Cuối cùng, thao tác trả về nút y bị loại khỏi cây

# ĐỘ PHỨC TẠP

- **Định Lý** Các thao tác INSERT và DELETE thực hiện trong thời gian O(h) trên một cây nhị phân tìm kiếm độ cao h

# CÂY BST NGẪU NHIÊN

- Độ cao của cây nhị phân tìm kiếm biến đổi thông qua các thao tác chèn và xoá các nút trên cây
- Thời gian thao tác là O(h), vì vậy khi cây suy biến và có chiều cao là n-1 (n là số nút trên cây) thì các thao tác này sẽ không còn hiệu quả
- Xác suất để xẩy ra sự suy biến khi chèn, xoá một cách ngẩu nhiên là rất nhỏ

# CÂY BST NGẪU NHIÊN

- Một cây nhị phân tìm kiếm được xây dựng ngẩu nhiên trên n khoá phân biệt là cây sinh ra từ thao tác chèn các khoá theo trật tự ngẫu nhiên vào trong một cây khởi đầu rỗng
- Việc tạo các cây nhị phân tìm kiếm một cách ngẩu nhiên thường cho ta một cây cân bằng (tham khảo)

# CÂY BST NGẪU NHIÊN

- **Định Lý** Độ cao trung bình của một cây nhị phân tìm kiếm được xây dựng ngẫu nhiên trên n khoá phân biệt là O(lg n)

# BIỂU DIỄN CÂY ĐA PHÂN

- Cơ chế biểu diễn cây nhị phân có thể mở rộng cho cây đa phân
- Nếu mỗi nút có tối đa k con thì có thể dùng k+1 biến là p (chỉ đến cha) và $child_1$, $child_2$, ..., $child_k$ (chỉ đến k con)
- Phương pháp này không hiệu quả (tốn nhiều bộ nhớ) và không sử dụng được nếu số con tối đa của một nút quá lớn

# BIỂU DIỄN CÂY ĐA PHÂN

- Sử dụng 3 biến p, left-child, right-sibling để lưu trữ các pointer chỉ đến cha, con trái trái nhất và anh, em bên phải của mỗi nút trong T
- root[T] là pointer chỉ đến gốc của T
- left-child[x] chỉ đến con trái nhất của x
- right-sibling[x] chỉ đến anh em trực tiếp bên phải của x
- Nếu x không có con left-child[x] = NIL
- Nếu x là con phải nhất thì right-sibling[x] = NIL

# BIỂU DIỄN CÂY ĐA PHÂN

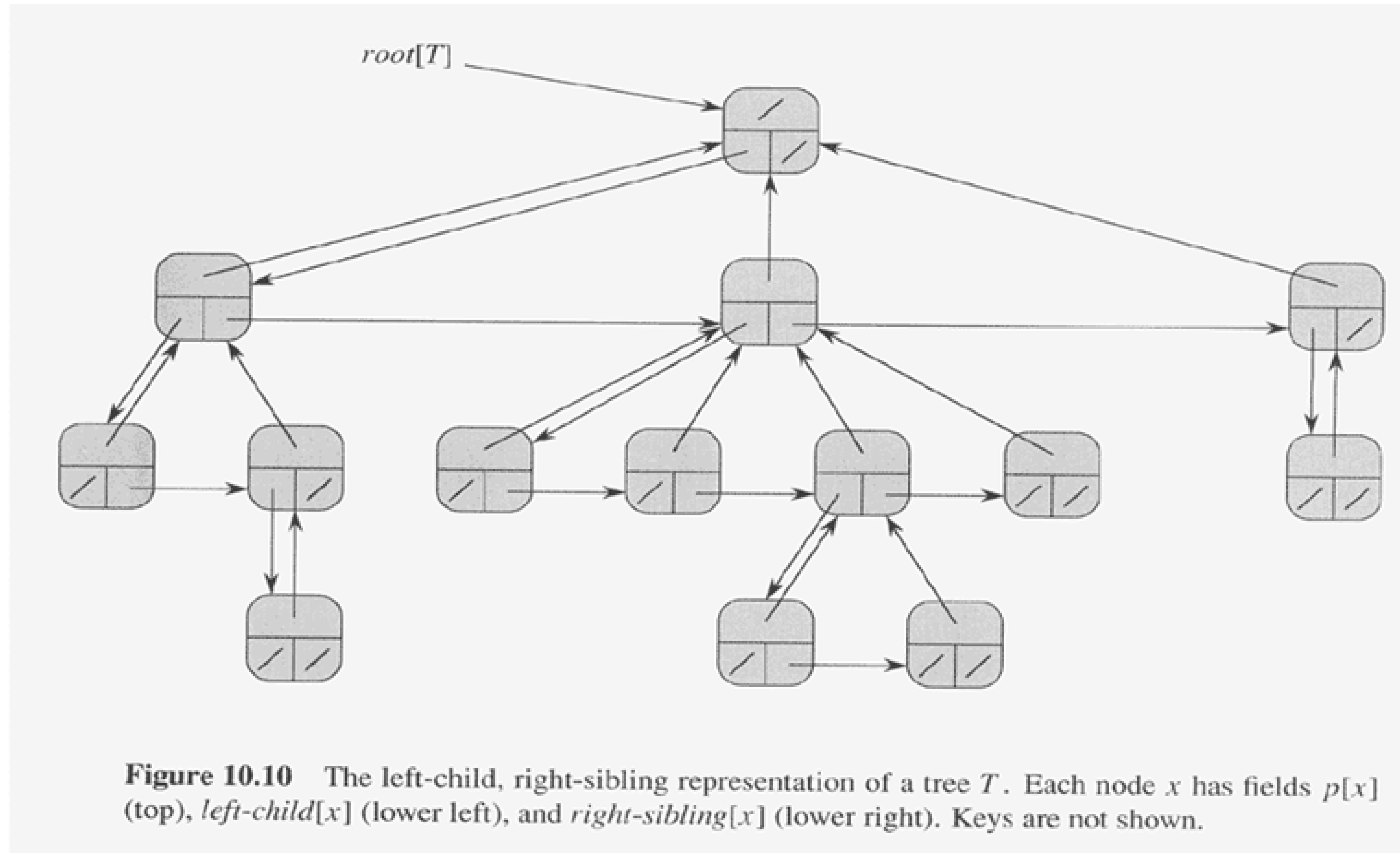



**Figure 10.10** The left-child, right-sibling representation of a tree $T$. Each node $x$ has fields $p[x]$ (top), *left-child*$[x]$ (lower left), and *right-sibling*$[x]$ (lower right). Keys are not shown.

# CÁC BIỂU DIỄN KHÁC CỦA CÂY

- Có thể biểu diễn cây có gốc như một đồ thị có hướng
- Có thể biểu diễn cây có gốc mà không cần pointer chỉ đến cha của nó
- Ngoài ra có thể dùng một mảng một chiều để biểu diễn cây có gốc

# B-CÂY

- Định nghĩa B-Cây
- Các thao tác trên B-Cây
- Hiện thực B-Cây

# ĐỊNH NGHĨA B-CÂY

- Một B-Cây (B-tree) T là một cây có gốc (với gốc root[T]), mỗi nút x có các thành phần sau
  - n[x], số khoá hiện tại được lưu trữ trong nút x
  - Các khoá $key_1[x]$, $key_2[x]$, ..., $key_{n[x]}[x]$ và được lưu trữ theo trật tự tăng
  - leaf[x] có giá trị luận lý là TRUE nếu x là lá và FALSE nếu x là một nút trong

# ĐỊNH NGHĨA B-CÂY

- Nếu x là một nút trong, nó chứa n[x]+1 pointer $c_1[x]$, $c_2[x]$, ...., $c_{n[x]+1}[x]$ đến các con của nó, các nút lá không có con, vì vậy các vùng $c_i$ của nó chưa được định nghĩa
- Các khoá $key_i[x]$ tách các vùng khoá được lưu trữ trong mỗi cây con như sau: nếu $k_i$ là một khoá được lưu trữ trong cây con gốc $c_i[x]$ thì $k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \ldots \leq key_{n[x]}[x] \leq k_{n[x]+1}$
- Mọi lá có cùng độ sâu, đó là chiều cao h của cây

# ĐỊNH NGHĨA B-CÂY

- Mỗi B-Cây có một số nguyên $t \geq 2$, biểu diễn cận trên và cận dưới của số các khoá trong một nút được gọi là bậc nhỏ nhất (minimum degree) của B-cây sao cho
  - Mỗi nút khác gốc phải có ít nhất t-1 khoá; mỗi nút trong khác gốc vì vậy phải có ít nhất t con
  - Nếu cây khác rỗng, nút gốc phải có ít nhất một khoá
  - Mỗi nút có thể chứa nhiều nhất 2t-1 khoá, vì vậy một nút trong có thể có nhiều nhất 2t con
  - Một nút là đầy (full) nếu nó chứa đúng 2t-1 khoá

# ĐỊNH NGHĨA B-CÂY

**Ví dụ 1**: Một B-Cây đơn giản nhất có bậc nhỏ nhất  $t = 2$

- Mỗi nút trong có thể có 2, 3, 4 con
- B-cây này được gọi là 2-3-4-tree
- Thực tế giá trị của t được sử dụng lớn hơn nhiều

# ĐỊNH NGHĨA B-CÂY

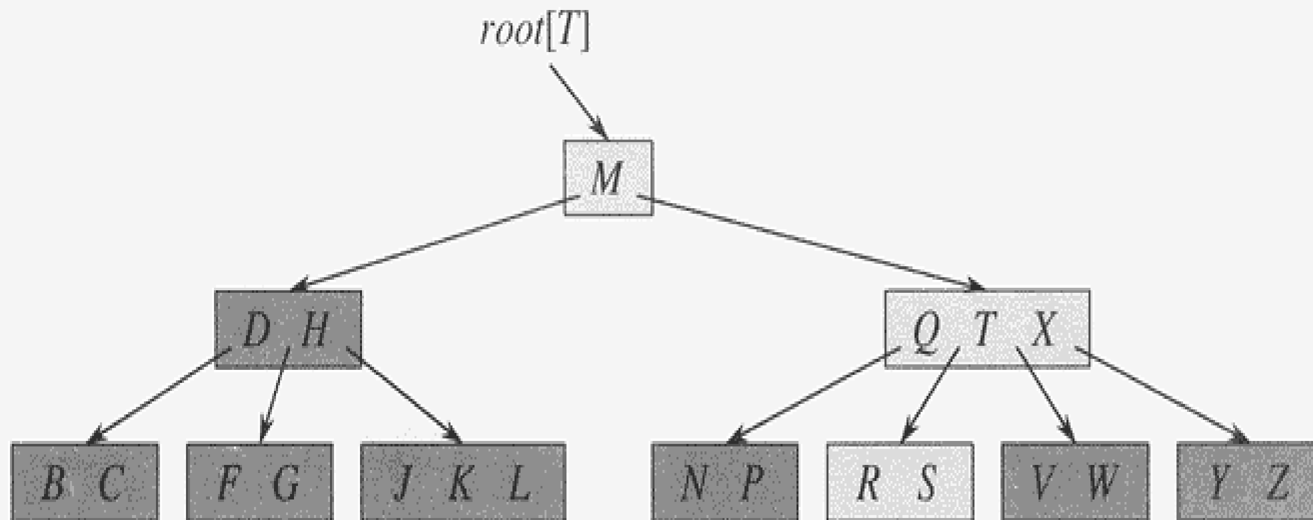



**Figure 18.1** A B-tree whose keys are the consonants of English. An internal node $x$ containing $n[x]$ keys has $n[x] + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter $R$.

# ĐỊNH NGHĨA B-CÂY

- B-Cây do R. Bayer đưa ra 1970
- B-Cây là cấu trúc dữ liệu thích hợp cho việc lưu trữ và thao tác dữ liệu trên bộ nhớ ngoài

# CHIỀU CAO B-CÂY

- **Định lý**: Với mọi B-Cây $T$ có $n$ khoá, chiều cao $h$ và bậc nhỏ nhất $t \geq 2$ thì,

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

- **Chứng minh?**

# CHIỀU CAO B-CÂY

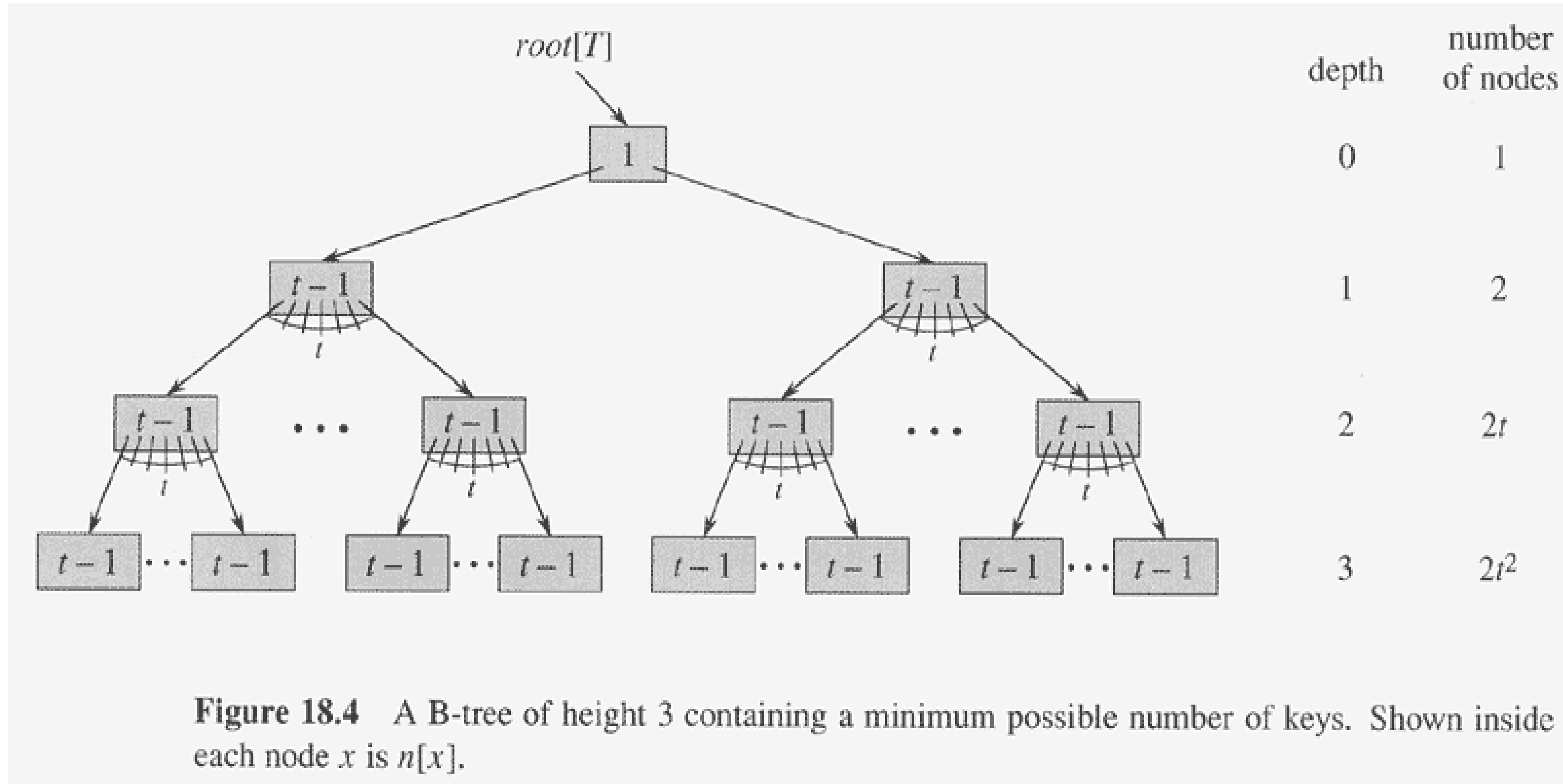



**Figure 18.4** A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node $x$ is $n[x]$.

# CHIỀU CAO B-CÂY

Chứng minh

- Số nút trên B-Cây ít nhất khi gốc có 1 khóa và tất cả các nút khác chứa t-1 khoá
- Trong trường hợp này B-Cây có 2 nút ở độ sâu 1, 2t nút ở độ sâu 2, $2t^2$ nút ở độ sâu 3, ..., tại độ sâu h có $2t^{h-1}$ nút

# CHIỀU CAO B-CÂY

Chứng minh

- Vậy số khoá *n* thỏa mãn bất đẳng thức

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

Từ đây suy ra hệ thức cần chứng minh

# CÁC THAO TÁC TRÊN B-CÂY

- B-TREE-SEARCH: Tìm kiếm trên B-Cây
- B-TREE-CREATE: Tạo một B-Cây rỗng
- B-TREE-SPLIT-CHILD: Tách một nút trong một B-Cây
- B-TREE-INSERT: Chèn một khoá vào một B-Cây
- B-TREE-DELETE: Xóa một khóa khỏi một B-Cây

# B-TREE-SEARCH

- Đầu vào của B-TREE-SEARCH là một pointer chỉ đến nút gốc x của một cây con và một khoá k cần tìm trong cây con này
- Thủ tục B-TREE-SEARCH là một sự mở rộng trực tiếp của TREE-SEARCH trên cây nhị phân tìm kiếm
- Nếu k ở trong B-Cây, thủ tục trả về một cặp có thứ tự (y, i) gồm nút y và chỉ số i sao cho $key_i[y]=k$, ngược lại, NIL được trả về

# B-TREE-SEARCH

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

# B-TREE-SEARCH

- Dòng 1-3 tìm i nhỏ nhất sao cho $k \leq key_i[x]$ ngược lại i được gán là $n[x]+1$
- Dòng 4-5 thủ tục kiểm tra và trả về nếu khoá được phát hiện
- Dòng 6-9 kết thúc tìm kiếm không thành công (nếu x là lá) hoặc đệ qui để tìm trong cây con thích hợp của x (sau khi thực hiện thao tác đọc đĩa)
- Thời gian chi phí cho thao tác tìm kiếm là $O(th) = O(t\log_t n)$

# B-TREE-CREATE

- Thủ tục B-TREE-CREATE tạo một B-Cây rỗng T
- Trước hết nó gọi ALLOCATE-NODE để được cấp phát một trang đĩa (disk page) như là một nút mới với thời gian là O(1)
- Giả định rằng một nút được tạo bởi ALLOCATE-NODE không yêu cầu đọc đĩa vì chưa có thông tin hữu dụng được lưu trữ trên đĩa cho nút này
- B-TREE-CREATE yêu cầu O(1) thời gian thao tác đĩa và O(1) thời gian CPU

# B-TREE-CREATE

B-TREE-CREATE$(T)$

1 $x \leftarrow$ ALLOCATE-NODE()
2 $leaf[x] \leftarrow$ TRUE
3 $n[x] \leftarrow 0$
4 DISK-WRITE$(x)$
5 $root[T] \leftarrow x$

# B-TREE-SPLIT-CHILD

- B-TREE-SPLIT-CHILD nhận đầu vào là một nút trong x chưa đầy, một chỉ số i và một nút con y =$c_i$[x] đã đầy của x
- B-TREE-SPLIT-CHILD tách nút y đầy (có 2t-1 khoá) tại khoá giữa (median key) $key_t$[y] thành 2 nút có t-1 khoá
- Khoá giữa được di chuyển lên nút cha x của y trước đó chưa đầy

# B-TREE-SPLIT-CHILD

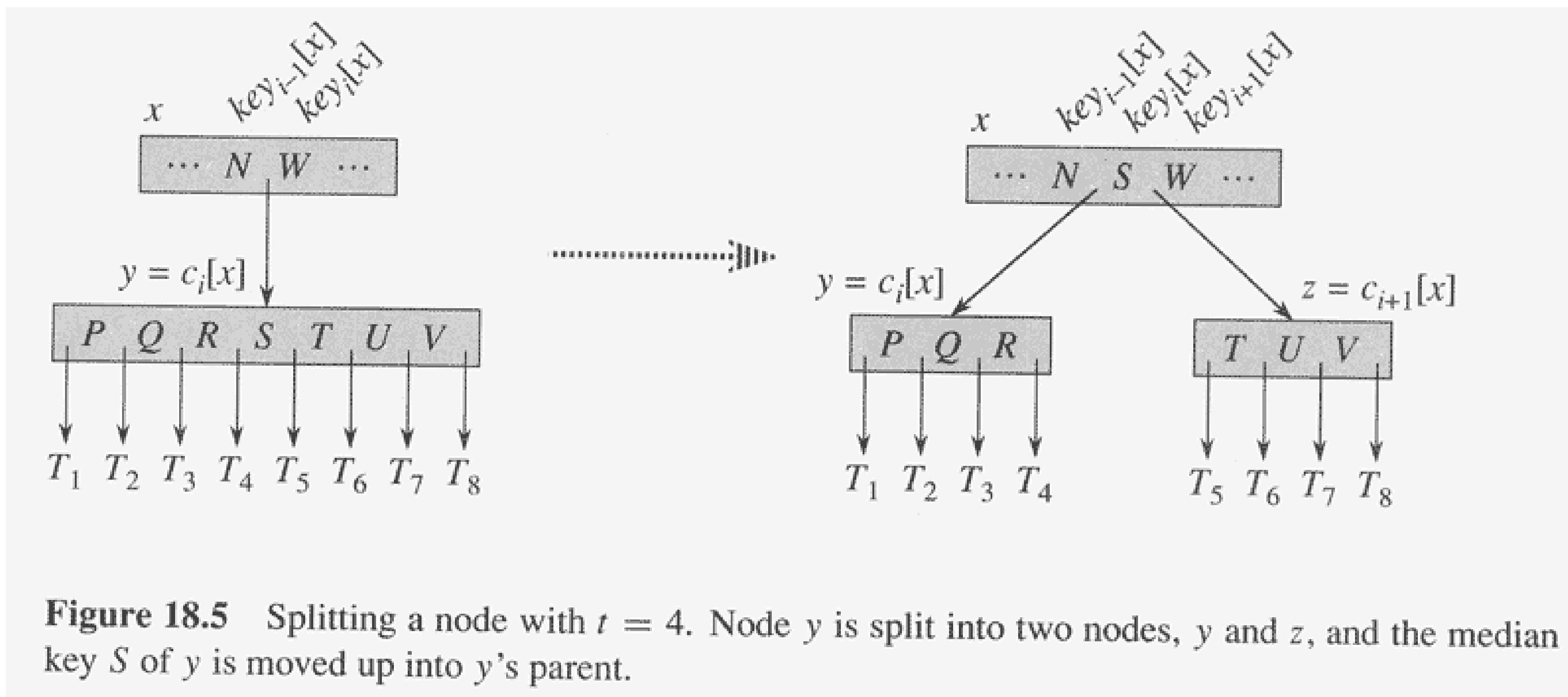



**Figure 18.5** Splitting a node with $t = 4$. Node $y$ is split into two nodes, $y$ and $z$, and the median key $S$ of $y$ is moved up into $y$'s parent.

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

```
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         do key_j[z] ← key_{j+t}[y]
 6   if not leaf[y]
 7      then for j ← 1 to t
 8                 do c_j[z] ← c_{j+t}[y]
 9   n[y] ← t − 1
10   for j ← n[x] + 1 downto i + 1
11         do c_{j+1}[x] ← c_j[x]
12   c_{i+1}[x] ← z
13   for j ← n[x] downto i
14         do 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)
```

# B-TREE-SPLIT-CHILD

- B-TREE-SPLIT-CHILD(x, i, y) tách nút y là con thứ i của nút x, lúc đầu y có 2t-1 khoá sau đó giảm còn t -1 khoá
- Nút z nhận t-1 khoá lớn nhất của y và z trở thành một con mới của x được định vị sau y trong bảng các con của x, khoá giữa của y được di chuyển lên x và phân chia y và z

# B-TREE-SPLIT-CHILD

- Dòng 1-8 tạo z và chuyển cho nó t-1 khoá lớn hơn tương ứng với t con của y
- Dòng 9 chỉnh lại số khoá cho y
- Cuối cùng dòng 10-16 chèn z như là một con của x, di chuyển khoá giữa từ y lên x để tách y và chỉnh lại số khoá của x
- Thời gian chạy là O(t)

# B-TREE-INSERT

- Thủ tục B-TREE-INSERTchèn một khoá k vào một B-tree T yêu cầu O(h) thời gian truy xuất đĩa và $O(th) = O(t\log_t n)$ thời gian CPU
- Thủ tục B-TREE-INSERT sử dụng B-TREE-SPLIT-CHILD để đảm bảo lời gọi đệ qui không bao giờ đi vào một nút đầy

# B-TREE-INSERT

```
B-TREE-INSERT(T, k)
 1  r ← root[T]
 2  if n[r] = 2t − 1
 3     then 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)
```

# B-TREE-INSERT

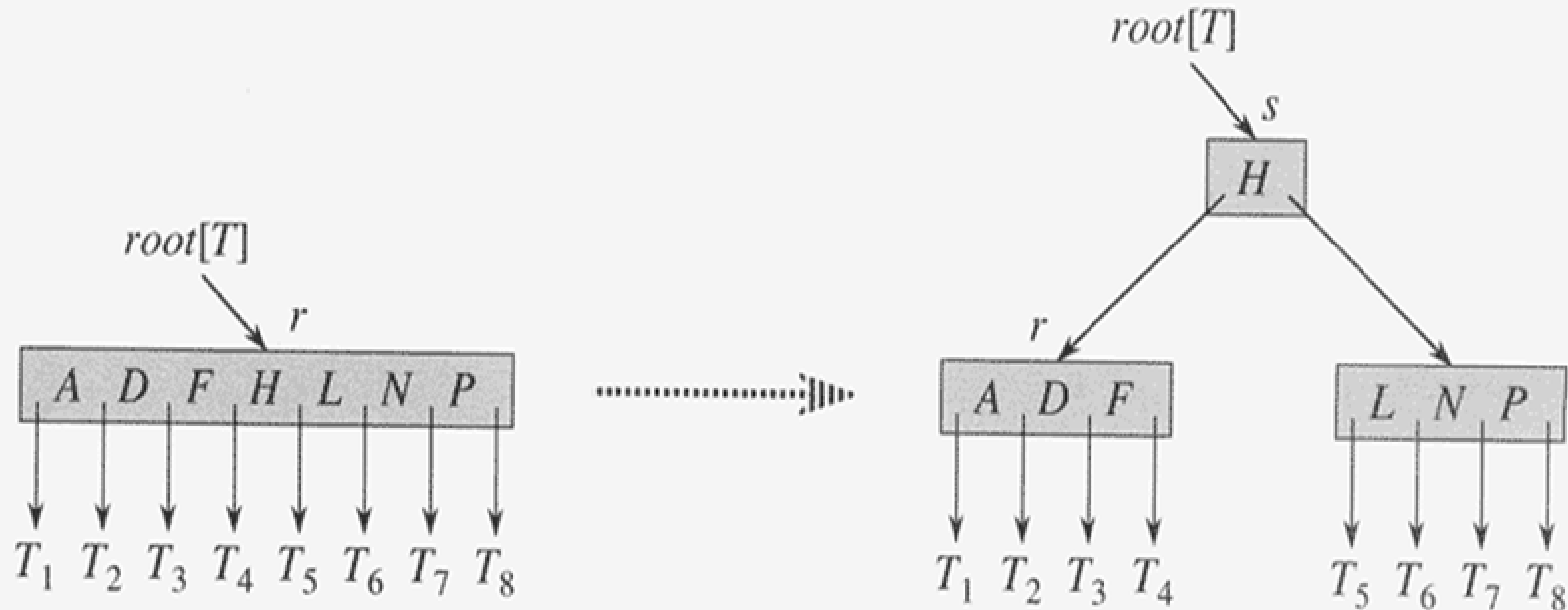



**Figure 18.6** Splitting the root with $t = 4$. Root node $r$ is split in two, and a new root node $s$ is created. The new root contains the median key of $r$ and has the two halves of $r$ as children. The B-tree grows in height by one when the root is split.

# B-TREE-INSERT

- Dòng 3-9 trong B-TREE-INSERT xử lý trường hợp trong đó nút gốc là đầy
- Nút gốc bị tách và một nút mới s (có hai con) trở thành nút gốc
- Tách nút gốc là cách duy nhất làm tăng chiều cao của một B-Cây

# B-TREE-INSERT

- Thủ tục hoàn thành bằng cách gọi B-TREE-INSERT-NONFULL để chèn khoá k vào trong cây được định gốc tại nút chưa đầy này (s hoặc r)
- B-TREE-INSERT-NONFULL là thủ tục đệ qui và nó đảm bảo không đi vào nút đầy bằng cách gọi B-TREE-SLIT-CHILD

# B-TREE-INSERT

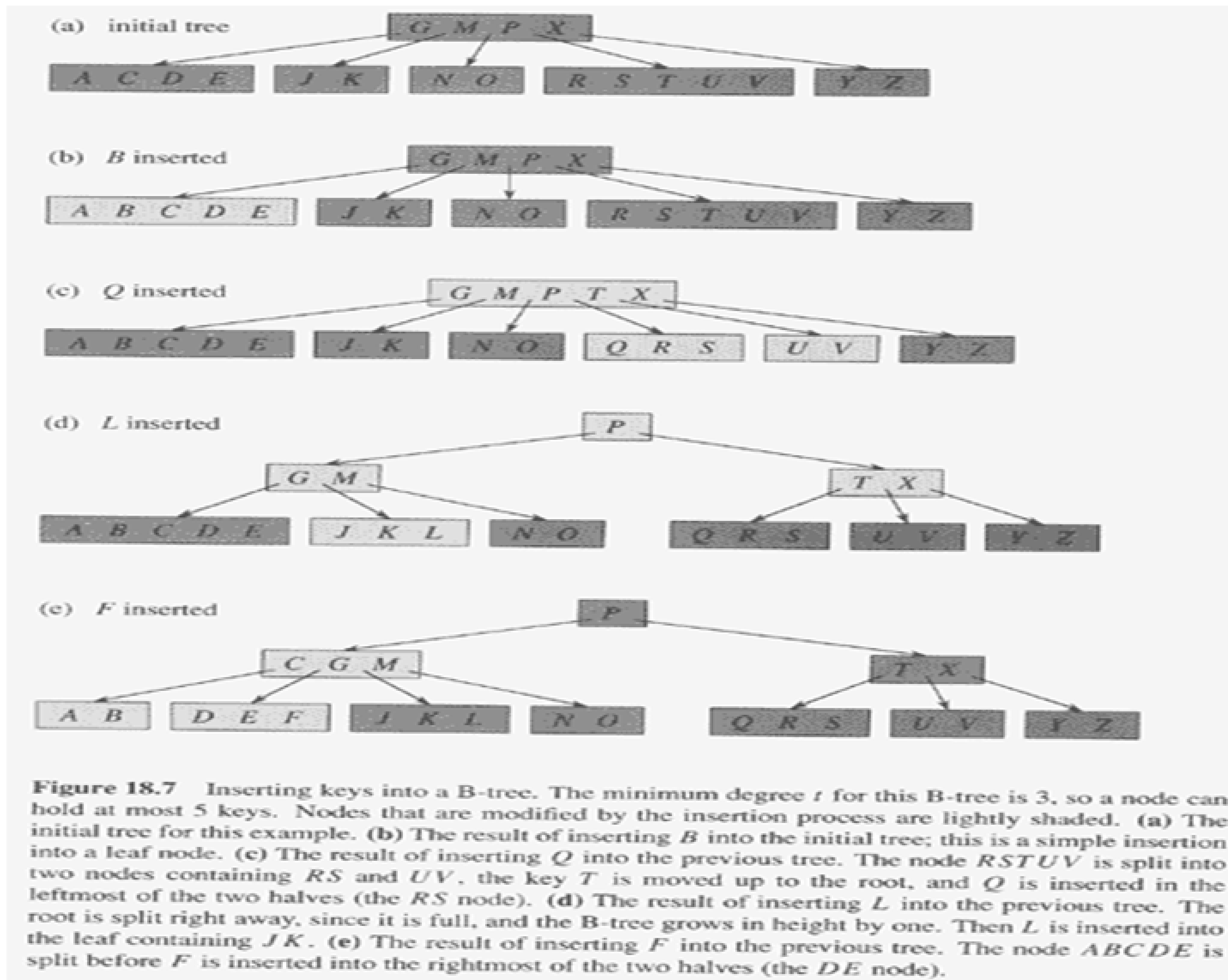



**Figure 18.7** Inserting keys into a B-tree. The minimum degree $t$ for this B-tree is 3, so a node can hold at most 5 keys. Nodes that are modified by the insertion process are lightly shaded. **(a)** The initial tree for this example. **(b)** The result of inserting $B$ into the initial tree; this is a simple insertion into a leaf node. **(c)** The result of inserting $Q$ into the previous tree. The node $RSTUV$ is split into two nodes containing $RS$ and $UV$, the key $T$ is moved up to the root, and $Q$ is inserted in the leftmost of the two halves (the $RS$ node). **(d)** The result of inserting $L$ into the previous tree. The root is split right away, since it is full, and the B-tree grows in height by one. Then $L$ is inserted into the leaf containing $JK$. **(e)** The result of inserting $F$ into the previous tree. The node $ABCDE$ is split before $F$ is inserted into the rightmost of the two halves (the $DE$ node).

# B-TREE-INSERT-NONFULL

```
B-TREE-INSERT-NONFULL(x, k)
 1  i ← n[x]
 2  if leaf[x]
 3     then while i ≥ 1 and k < key_i[x]
 4              do 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              do i ← i − 1
11          i ← i + 1
12          DISK-READ(c_i[x])
13          if n[c_i[x]] = 2t − 1
14             then B-TREE-SPLIT-CHILD(x, i, c_i[x])
15                  if k > key_i[x]
16                     then i ← i + 1
17          B-TREE-INSERT-NONFULL(c_i[x], k)
```

# B-TREE-INSERT-NONFULL

- Dòng 3-8 xử lý trường hợp trong đó x là một nút lá bằng cách chèn khoá k vào trong x
- Nếu x không phải là nút lá, phải chèn k vào nút lá thích hợp trong cây con định gốc tại nút x

# B-TREE-INSERT-NONFULL

- Trong trường hợp này, dòng 9-11 xác định con của x theo đó sự đệ qui đi vào
- Dòng 13 kiểm tra cho trường hợp đệ qui đi vào nút con đầy, khi đó dòng 14 sẽ tách nút con đó thành 2 nút con không đầy, dòng 15-16 xác định đúng nút con để gọi đệ qui
- Dòng 17 thực hiện lời gọi đệ qui để chèn k vào cây con thích hợp

# B-TREE-DELETE

- B-TREE-DELETE(x, k) xóa khóa k khỏi cây con định gốc tại x, và đảm bảo rằng bất kỳ khi nào nó được gọi đệ qui trên nút x thì số khoá trong x ít nhất là bằng bậc nhỏ nhất t (điều kiện này đòi hỏi do tính chất của B-cây)
- Bất kỳ khi nào nút gốc x trở thành nút trong không có khoá thì nút con $c_1[x]$ trở thành nút gốc và cây giảm chiều cao

# B-TREE-DELETE

1. Nếu khoá k ở trong x và x là lá thì xoá k khỏi x
2. Nếu khoá k ở trong x và x là một nút trong thì làm như sau:
   a. Nếu con y đi trước k trong nút x có ít nhất là t khoá, thì tìm đi trước k′ của k trong cây con được định gốc tại y, một cách đệ qui xoá k′ và thay thế k bởi k′ trong x
   b. Tương tự, nếu con z đi sau k có ít nhất t khoá, thì tìm đi sau k′ của k trong cây con được định gốc tại z, một cách đệ qui xoá k′ và thay thế k bởi k′ trong x
   c. Nếu cả y và z có chỉ t-1 khoá, trộn k và tất cả các khóa của z vào trong y sao cho x mất cả k và pointer đến z và y bây giờ chứa 2t-1 khoá, giải phóng z và đệ qui xoá k khỏi y

# B-TREE-DELETE

3. Nếu khoá k không hiện diện trong nút trong x, thì xác định gốc $c_i[x]$ của cây con thích hợp chứa khoá k. Nếu $c_i[x]$ chỉ có t -1 con, thực hiện bước 3a hoặc 3b để đảm bảo rằng không gọi đệ qui vào nút có ít hơn t khoá. Sau đó hoàn tất bằng cách gọi đệ qui trên con thích hợp của x
   a. Nếu $c_i[x]$ chỉ có t-1 khoá nhưng có một anh em với t khoá, cho $c_i[x]$ một khoá bổ sung bằng cách di chuyển một khoá từ x xuống $c_i[x]$, di chuyển một khoá từ anh em trái hoặc phải trực tiếp của $c_i[x]$ lên x
   b. Nếu $c_i[x]$ và tất cả các anh em của $c_i[x]$ có t-1 khóa, trộn $c_i[x]$ với một anh em và yêu cầu di chuyển một khoá từ x vào nút được trộn để trở thành khoá giữa của nút đó

# B-TREE-DELETE

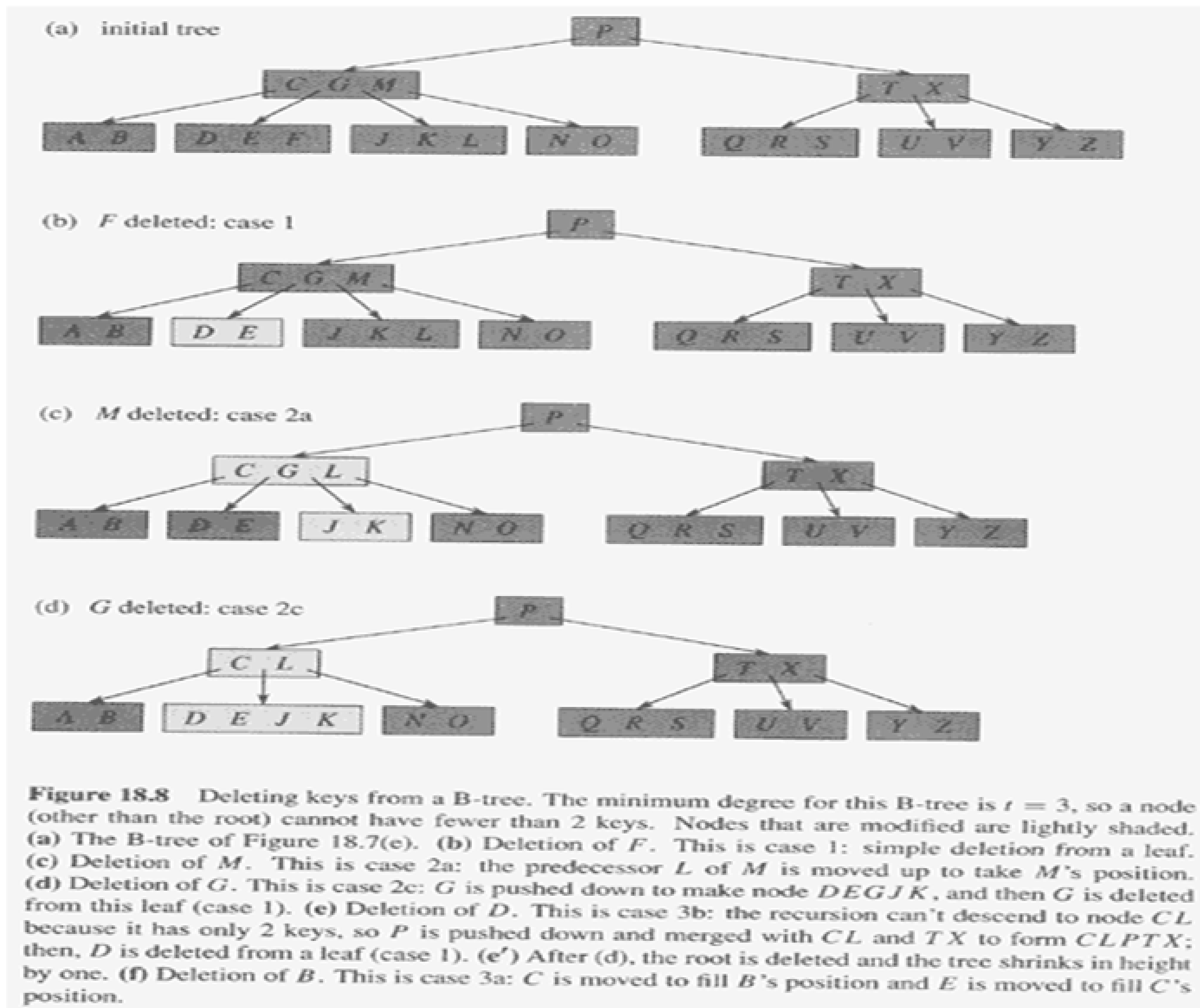



**Figure 18.8** Deleting keys from a B-tree. The minimum degree for this B-tree is $t = 3$, so a node (other than the root) cannot have fewer than 2 keys. Nodes that are modified are lightly shaded. **(a)** The B-tree of Figure 18.7(e). **(b)** Deletion of $F$. This is case 1: simple deletion from a leaf. **(c)** Deletion of $M$. This is case 2a: the predecessor $L$ of $M$ is moved up to take $M$'s position. **(d)** Deletion of $G$. This is case 2c: $G$ is pushed down to make node $DEGJK$, and then $G$ is deleted from this leaf (case 1). **(e)** Deletion of $D$. This is case 3b: the recursion can't descend to node $CL$ because it has only 2 keys, so $P$ is pushed down and merged with $CL$ and $TX$ to form $CLPTX$; then, $D$ is deleted from a leaf (case 1). **(e′)** After (d), the root is deleted and the tree shrinks in height by one. **(f)** Deletion of $B$. This is case 3a: $C$ is moved to fill $B$'s position and $E$ is moved to fill $C$'s position.

# B-TREE-DELETE