CHƯƠNG 10

HÀNG ƯU TIÊN

Trong các chương trước chúng ta đã nghiên cứu KDLTT từ điển. Từ điển là một tập đối tượng dữ liệu, mỗi đối tượng được gắn với một giá trị khóa, và các phép toán tìm kiếm, xen, loại trên từ điển được tiến hành khi được cung cấp một giá trị khóa. Trong chương này chúng ta sẽ đưa ra KDLTT hàng ưu tiên. Hàng ưu tiên khác với từ điển ở chỗ, thay cho giá trị khóa, mỗi đối tượng dữ liệu trong hàng ưu tiên được gắn với một giá trị ưu tiên, và chúng ta chỉ quan tâm tới việc tìm kiếm và loại bỏ đối tượng có giá trị ưu tiên nhỏ nhất. Nội dung chính của chương này là:

- Đặc tả KDLTT hàng ưu tiên.
- Trình bày phương pháp cài đặt hàng ưu tiên bởi cây thứ tự bộ phận (heap).
- Đưa ra một ứng dụng của hàng ưu tiên trong nén dữ liệu và xây dựng mã Huffman.

10.1 KIỂU DỮ LIỆU TRỪU TƯỢNG HÀNG ƯU TIÊN

Giả sử chúng ta cần bảo lưu một cơ sở dữ liệu gồm các bản ghi về các bệnh nhân đến khám và chữa bệnh tại một bệnh viện. Các bệnh nhân khi đến bệnh viện sẽ được đưa vào cơ sở dữ liệu này. Nhưng các bác sĩ khám cho bệnh nhân sẽ không phục vụ người bệnh theo thứ tự ai đến trước sẽ được khám trước (như cách tổ chức dữ liệu theo hàng đợi). Mỗi bệnh nhân sẽ được cấp một giá trị ưu tiên và các bác sĩ sẽ gọi vào phong khám bệnh nhân có giá trị ưu tiên nhỏ nhất (người được ưu tiên trước hết tại thời điểm đó). Nhiều hoàn cảnh khác (chẳng hạn, hệ máy tính phục vụ nhiều người sử dụng) cũng đòi hỏi chúng ta cần phải tổ chức một tập đối tượng dữ liệu theo giá trị ưu tiên sao cho các thao tác tìm đối tượng và loại đối tượng có giá trị

ưu tiên nhỏ nhất được thực hiện hiệu quả. Điều đó dẫn đến sự hình thành KDLTT hàng ưu tiên.

Hàng ưu tiên (priority queue) được xem là một tập các đối tượng dữ liệu, mỗi đối tượng có một giá trị ưu tiên. Thông thường các giá trị ưu tiên có thể là các số nguyên, các số thực, các ký tự…; điều quan trọng là chúng ta có thể so sánh được các giá trị ưu tiên . Trên hàng ưu tiên chúng ta chi quan tâm tới các phép toán sau đây:

1. Insert (P,x). Xen vào hàng ưu tiên P đối tượng x.
2. FindMin(P). Hàm trả về đối tượng trong P có giá trị ưu tiên nhỏ nhất (đối tượng được ưu tiên nhất). Phép toán này đòi hỏi P không rỗng
3. DeleteMin(P). Loại bỏ và trả về đối tượng có giá trị ưu tiên nhỏ nhất trong P. P cũng cần phải không rỗng.

Hàng ưu tiên được sử dụng trong các hoàn cảnh tương tự như hoàn cảnh đã nêu trên, tức là khi ta cần quản lý sự phục vụ theo mức độ ưu tiên. Hàng ưu tiên còn được sử dụng để thiết kế các thuật toán trong nhiều ứng dụng. Cuối chương này chúng ta sẽ đưa ra một ứng dụng: sử dụng hàng ưu tiên để thiết kế thuật toán xây dựng mã Huffman.

Trong các mục tiếp theo chúng ta sẽ đề cập tới các phương pháp cài đặt hàng ưu tiên.

10.2 CÁC PHƯƠNG PHÁP ĐƠN GIẢN CÀI ĐẶT HÀNG ƯU TIÊN

Trong mục này chúng ta sẽ thảo luận cách sự dụng các cấu trúc dữ liệu đã quen biết: danh sách, cây tìm kiếm nhị phân để cài đặt hàng ưu tiên và thảo luận về hiệu quả của các phép toán hàng ưu tiên trong các cách cài đặt đơn giản đó.

10.2.1 Cài đặt hàng ưu tiên bởi danh sách

Cài đặt hàng ưu tiên đơn giản nhất là biểu diễn hàng ưu tiên dưới dạng một danh sách được sắp hoặc không được sắp theo giá trị ưu tiên. Đương nhiên là danh sách đó có thể được lưu trong mảng hay DSLK.

Nếu chung ta cài đặt hàng ưu tiên bởi danh sách các phần tử được sắp xếp theo thứ tự ưu tiên tăng dần (hoặc giảm dần) thì phần tử có giá trị ưu tiên nhỏ nhất nằm ở một đầu danh sách, và do đó các phép toán FindMin và DeleteMin chỉ cần thời gian O(1). Nhưng để thực hiện phép toán Insert chúng ta cần tìm vị trí thích hợp trong danh sách để đặt phần tử mới sao cho tính chất được sắp của danh sách được bảo tồn. Vì vậy phép toán Insert đòi hỏi thời gian O(n), trong đó n là độ dài của danh sách.

Nếu chúng ta cài đặt hàng ưu tiên bởi danh sách các phần tử theo một thứ tự tùy ý, thì khi cần xen vào hàng ưu tiên một phần tử mới chúng ta chỉ cần đưa nó vào đuôi danh sách, do đó phép toán Insert chỉ cần thời gian O(1). Song để tìm và loại phần tử có giá trị ưu tiên nhỏ nhất chúng ta cần phải duyệt toàn bộ danh sách, và vì vậy các phép toán FindMin và DeleteMin cần thời gian O(n).

10.2.2 Cài đặt hàng ưu tiên bởi cây tìm kiếm nhị phân

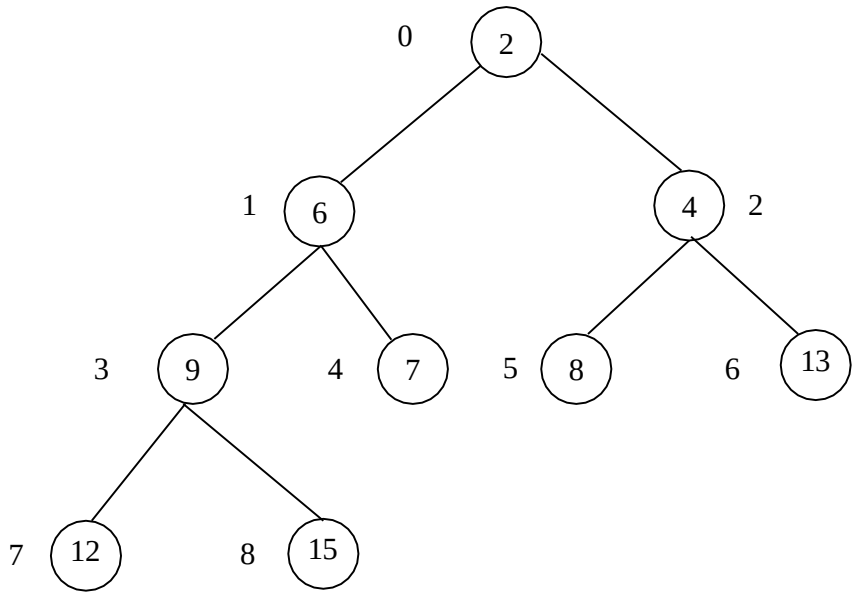
Trong mục 8.4.3 chúng ta đã cài đặt KDLTT tập động bởi cây tìm kiếm nhị phân. Cần lưu ý rằng, có thể xem hàng ưu tiên như là một dạng đặc biệt của tập động khi mà ta lấy giá trị ưu tiên của mỗi phần tử làm khóa và chỉ quan tâm tới các phép toán Insert, FindMin, DeleteMin. Vì vậy đương nhiên chúng ta có thể cài đặt hàng ưu tiên bởi cây tìm kiếm nhị phân. Từ lớp BSTree (trong hình 8.16), bằng thừa kế bạn có thể đưa ra lớp cài đặt hàng ưu tiên. Hiệu quả của các phép toán hàng ưu tiên trong cách cài đặt này đã được thảo luận trong mục 8.5. Trong trường hợp tốt nhất thời gian thực hiện các phép toán hàng ưu tiên là O(log n), trường hợp xấu nhất là O(n).

Trong mục sau đây chúng ta sẽ đưa ra một cách cài đặt mới: cài đặt hàng ưu tiên bởi cây thứ tự bộ phận. Với cách cài đặt này, thời gian thực hiện của các phép toán hàng ưu tiên luôn luôn là O(log n).

10.3 CÂY THỨ TỰ BỘ PHẬN

Trong mục 8.4 chúng ta đã nghiên cứu CTDL cây tìm kiếm nhị phân. Trong cây tìm kiếm nhị phân, các khóa của dữ liệu chứa trong các đỉnh của cây cần phải thỏa mãn tính chất thứ tự: khóa của một đỉnh bất kỳ lớn hơn khóa của các đỉnh trong cây con trái và nhỏ hơn khóa của các đỉnh trong cây con phải. Trong cây thứ tự bộ phận, chỉ đòi hỏi các khóa chứa trong các đỉnh thỏa mãn tính chất thứ tự bộ phận: Khóa của một đỉnh bất kỳ nhỏ hơn hoặc bằng khóa của các đỉnh con của nó. Mặc dù vậy, CTDL cây thứ tự bộ phận cho phép ta thực hiện hiệu quả các phép toán hàng ưu tiên.

Cây thứ tự bộ phận (partially ordered tree, hoặc heap) là một cây nhị phân hoàn toàn và thỏa mãn tính chất thứ tự bộ phận. Ví dụ, cây nhị phân trong hình 10.1 là cây thứ tự bộ phận.



Hình 10.1 Cây thứ tự bộ phận

Chúng ta cần lưu ý đến một số đặc điểm của cây thứ tự bộ phận. Trước hết, nó cần phải là cây nhị phân hoàn toàn (xem định nghĩa trong mục 8.3), tức là tất cả các mức của cây đều không thiếu đỉnh nào, trừ mức thấp nhất được lấp đầy kể từ bên trái. Tính chất này cho phép ta cài đặt cây thứ tự bộ phận bởi mảng. Mặc khác, từ tính chất thứ tự bộ phận ta rút ra đặc điểm sau: Các giá trị khóa nằm trên đường đi từ gốc tới các nút lá tạo thành một dãy không giảm. Chẳng hạn, dãy các giái trị khóa từ gốc (đỉnh 0) tới đỉnh 8 là 3, 6, 9, 15. Điều đó có nghĩa là, dữ liệu có khóa nhỏ nhất được lưu trong gốc của cây thứ tự bộ phận.

Độ cao của cây thứ tự bộ phận. Giả sử n là số đỉnh của cây nhị phân hoàn toàn có độ cao h. Dễ dàng thấy rằng, $2^{h-1} < n \leq 2^h - 1$ và do đó $2^{h-1} < n+1 \leq 2^h$. Từ đó ta suy ra rằng, độ cao h của cây thứ tự bộ phận có n đỉnh bằng [log(n+1)].

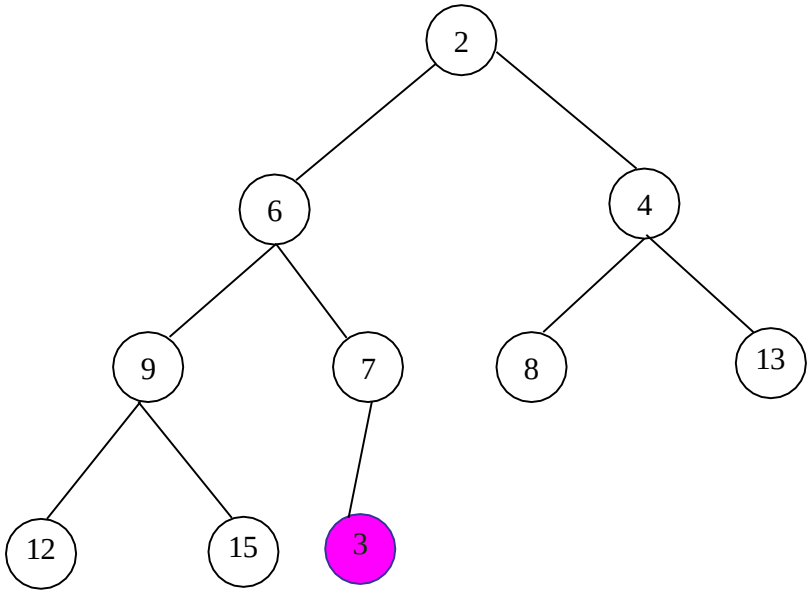
Sau đây chúng ta xét xem các phép toán hàng ưu tiên được thực hiện như thế nào khi hàng ưu tiên biểu diễn bởi cây thứ tự bộ phận (với giá trị khóa của dữ liệu chứa trong mỗi đỉnh được lấy là giá trị ưu tiên).

10.3.1 Các phép toán hàng ưu tiên trên cây thứ tự bộ phận

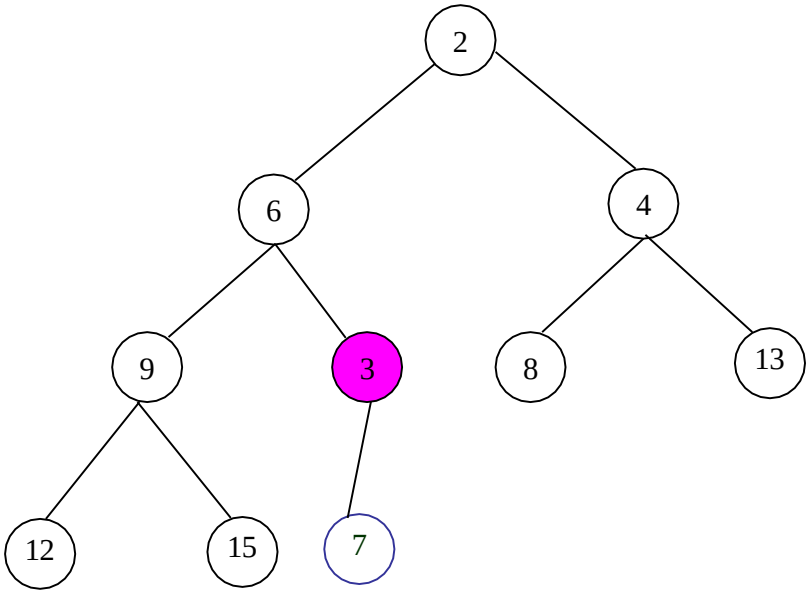
Như trên đã nhận xét, đối tượng dữ liệu có khóa nhỏ nhất được lưu trong gốc của cây thứ tự bộ phận, do đó phép toán FindMin được thực hiện dễ dàng và chỉ đòi hỏi thời gian O(1). Khó khăn khi thực hiện phép toán Insert và DeleteMin là ở chỗ chúng ta phải biến đổi cây như thế nào để sau khi xen vào (hoặc loại bỏ) cây vẫn còn thỏa mãn tính chất thứ tự bộ phận.

Phép toán Insert. Giả sử chúng ta cần xen vào cây một đỉnh mới chứa dữ liệu x. Để đảm bảo tính chất là cây nhị phân hoàn toàn, đỉnh mới được đặt là đỉnh ngoài cùng bên phải ở mức thấp nhất. Chẳng hạn, từ cây trong hình 10.1, khi thêm vào đỉnh mới với khóa là 3 ta nhận được cây trong hình 10.2 a. Nhưng cây nhận được có thể không thỏa mãn tính chất thứ tự bộ phận, vì đỉnh mới thêm vào có thể có khóa nhỏ hơn khóa của đỉnh cha nó,

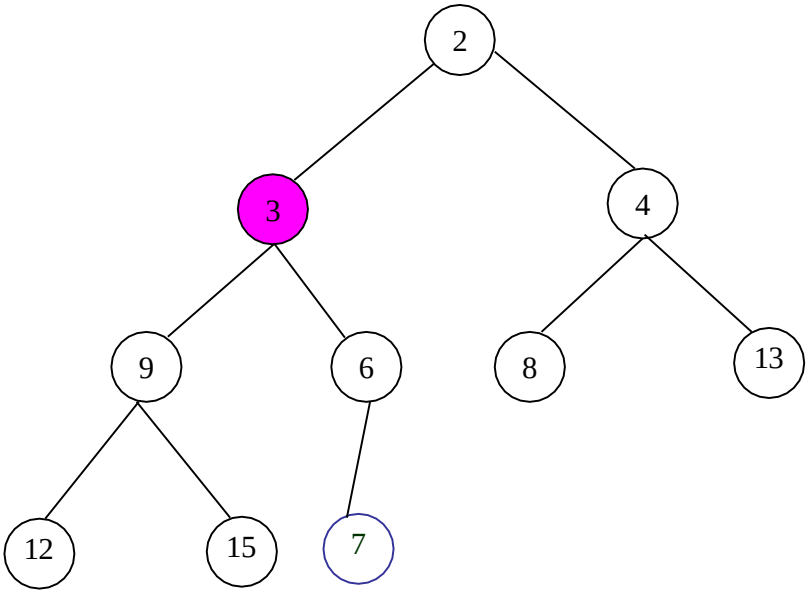
chẳng hạn cây trong hình 10.2 a, đỉnh mới thêm vào có khóa là 3, đỉnh cha nó có khóa là 7. Chúng ta có thể sắp xếp lại dữ liệu chứa trong các đỉnh để cây thỏa mãn tính chất thứ tự bộ phận bằng thuật toán đơn giản sau. Đi từ đỉnh mới thêm vào lên gốc cây, mỗi khi ta đang ở một đỉnh có khóa nhỏ hơn khóa của đỉnh cha nó thì ta hoán vị dữ liệu chuá trong hai đinh đó và đi lên đỉnh cha. Quá trình đi lên sẽ dừng lại khi ta đạt tới một đỉnh có khóa lớn hơn khóa của đỉnh cha nó, hoặc cùng lắm là khi đạt tới gốc cây. Ví dụ, với cây trong hình 10.2 a, sự đổi chỗ các dữ liệu được minh họa trong hình 10.2 a-c. Dễ dàng chứng minh được rằng, quá trình đi từ đỉnh mới thêm vào lên gốc và tiến hành đổi chỗ các dữ liệu như trên sẽ cho cây kết quả là cây thứ tự bộ phận (bài tập).



(a)



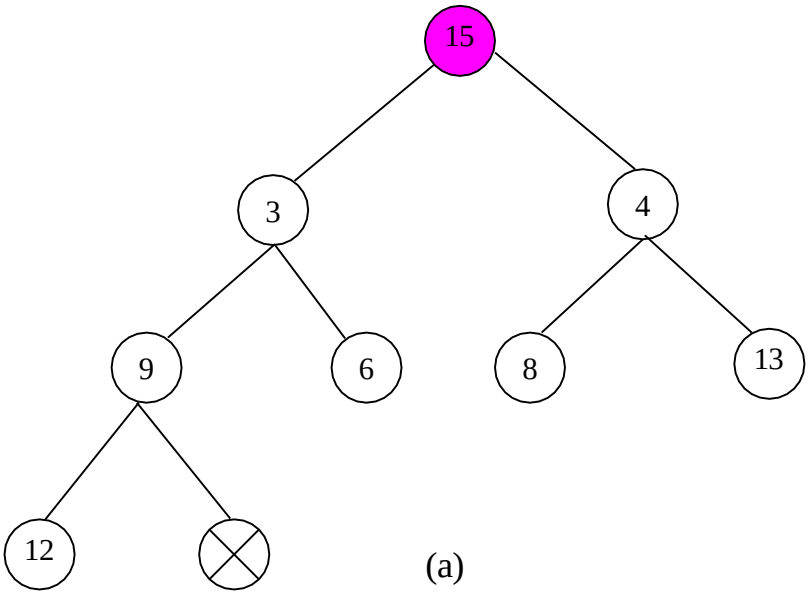
(b)



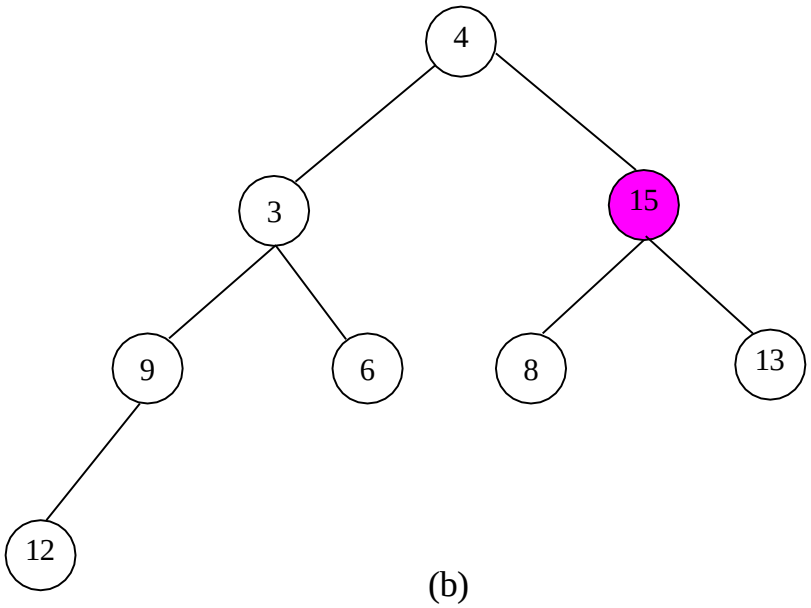
(c)

Hình 10.2. Xen một đỉnh mới vào cây thứ tự bộ phận.

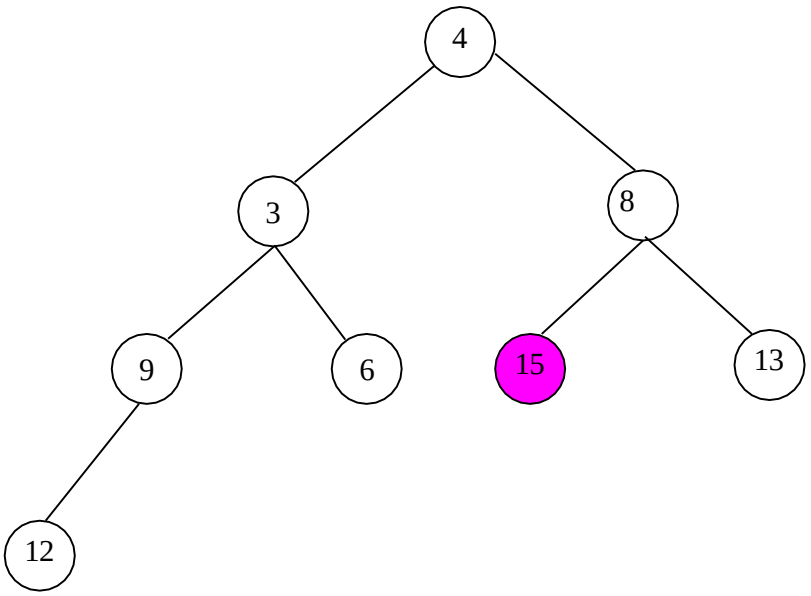
Phép toán DeleteMin. Chúng ta cần loại bỏ khỏi cây thứ tự bộ phận đỉnh có khóa nhỏ nhất, như trên đã nhận xét, đó là gốc cây. Thuật toán loại gốc cây được tiến hành qua hai bước. Đầu tiên, ta đem dữ liệu ở đỉnh ngoài cùng bên phải ở mức thấp nhất lưu vào gốc cây và loại đỉnh đó. Chẳng han, từ cây trong hình 10.1, ta nhận được cây trong hình 10.3a . Hành động trên mới chỉ đảm bảo đỉnh chứa dữ liệu có khóa nhỏ nhất đã bị loại và cây vẫn là cây nhị phân hoàn toàn. Song tính chất thứ tự bộ phận có thể bị vi phạm, vì khóa của đỉnh gốc bây giờ có thể lớn hơn khóa của các đỉnh con của nó, chẳng hạn như cây trong hình 10.3 a. Bước tiếp theo, ta đi từ gốc cây xuống lá và tiến hành hoán vị các dữ liệu chứa trong các đỉnh cha và con khi cần thiết. Giả sử ta đang ở đỉnh p, con trái của p (nếu có) là v_l, con phải (nếu có) là v_r . Giả sử đỉnh p có khóa lớn hơn khóa của ít nhất một trong hai đỉnh con là v_l và v_r. Khi đó, nếu khóa của đỉnh v_l nhỏ hơn khóa của đỉnh v_r thì ta hoán vị các dữ liệu trong p và v_l và đi xuống đỉnh v_l; nếu ngược lại, ta hoán vị các dữ liệu trong p và v_r, rồi đi xuống đỉnh v_r. Quá trình đi xuống sẽ dừng lại khi ta đạt tới một đỉnh có khóa nhỏ hơn khóa của các đỉnh con, hoặc cùng lắm là khi đạt tới một đỉnh lá. Ví dụ, sự đổi chỗ các dữ liệu chứa trong các đỉnh của cây trong hình 10.3a được thể hiện trong các hình 10.3 a-c.



(a)



(b)



(c)

Hình 10.3. Loại đỉnh chứa khóa nhỏ nhất.

Chúng ta đã chỉ ra rằng, độ cao của cây hoàn toàn xấp xỉ bằng log(n), trong đó n là số đỉnh của cây. Các phép toán Insert và DeleteMin chỉ tiêu tốn thời gian để hoán vị các dữ liệu chứa trong các đỉnh nằm trên đường đi từ lá lên gốc (đối với phép toán Insert) hoặc trên đường đi từ gốc xuống lá (phép toán DeleteMin). Vì vậy, thời gian thực hiện các phép toán Insert và DeleteMin trên cây thứ tự bộ phận là O(log n), trong đó n lá số dữ liệu trong hàng ưu tiên.

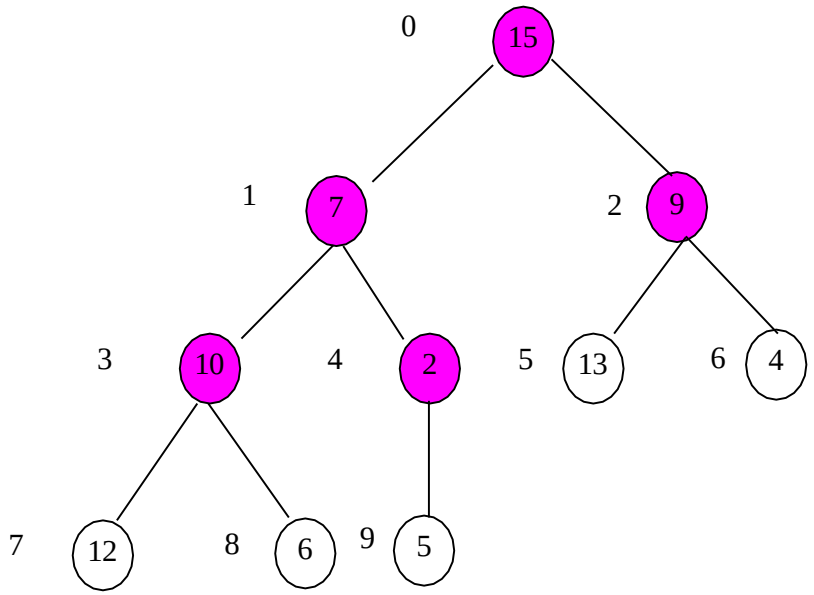
10.3.2 Xây dựng cây thứ tự bộ phận

Trong các ứng dụng, thông thường chúng ta cần phải xây dựng một cây thứ tự bộ phận từ n đối tượng dữ liệu đã có. Một cách đơn giản là, xuất phát từ cây rỗng, áp dụng phép toán Insert để xen vào các đỉnh mới chứa các dữ liệu đã cho. Mỗi phép toán xen vào đòi hỏi thời gian O(log n) và do đó toàn bộ quá trình trên cần thời gian O(n log n). Sau đây chúng ta đưa ra một cách xây dựng khác, chỉ đòi hỏi thời gian là O(n).

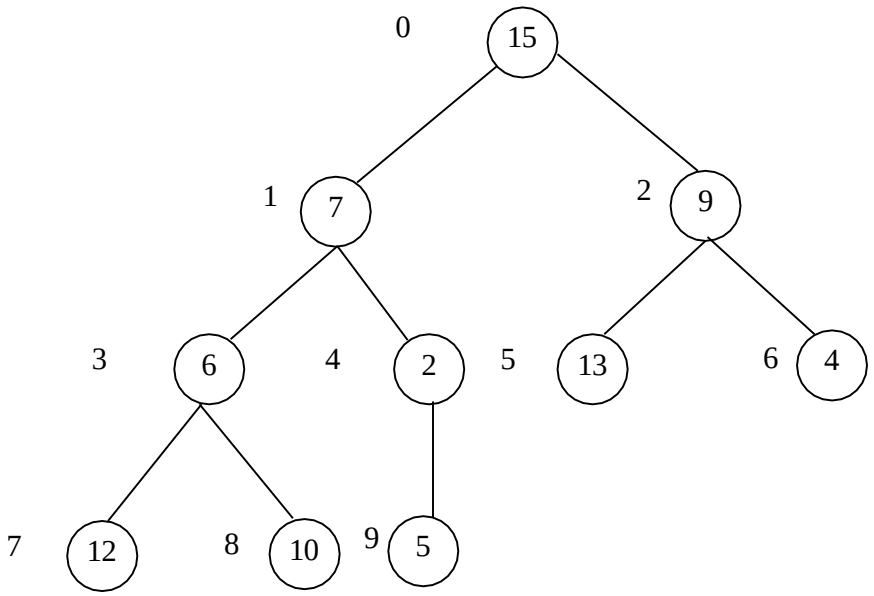
Như đã trình bày ở trên, thuật toán DeleteMin gồm hai bước. Sau bước thứ nhất, ta nhận được cây (chẳng hạn, cây trong hình 10.3 a), cây này có hai cây con trái và phải của gốc đã là cây thứ tự bộ phận, chỉ trừ tại gốc có thể tính chất thứ tự bộ phận không thỏa mãn. Trong bước thứ hai, ta đi từ gốc xuống và tiến hành hoán vị các dữ liệu trong cặp đỉnh cha, con khi cần thiết để cho cây thỏa mãn tính chất thứ tự bộ phận. Chúng ta sẽ nói tới bước này như là bước đẩy dữ liệu chứa trong gốc cây xuống vị trí thích hợp (siftdown). Điều đó là cơ sở của thuật toán xây dựng cây sau đây. Thuật toán này gồm hai giai đoạn:

1. Từ dãy n dữ liệu, ta xây dựng cậy nhị phân hoàn toàn, các dữ liệu được lần lượt đưa vào các đỉnh của cây theo thứ tự từ trên xuống dưới và trong cùng một mức thì từ trái qua phải, bắt đầu từ gốc được đánh số là 0. Chẳng hạn, từ 10 dữ liệu với các khóa là 15 , 7 , 9 , 10 , 2 , 13 , 4 , 12 , 6 và 5, ta xây dựng được cây nhị phân hoàn toàn như trong hình 10.4 a.

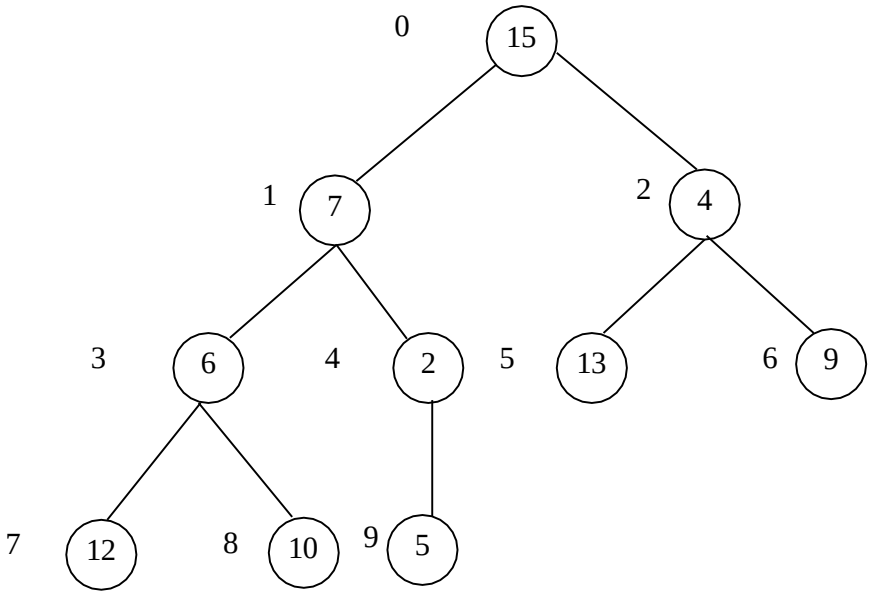
2. Xử lý các đỉnh i của cây bắt đầu từ đỉnh được đánh số là $\lfloor n/2 - 1 \rfloor$, sau mỗi lần i được giảm đi 1 cho tới khi i bằng 0. Chẳng hạn, trong hình 10.4 a, chúng ta lần lượt xử lý các đỉnh i = 4, 3, 2, 1, 0. Mỗi khi xử lý tới đỉnh i, thì đỉnh i đã là gốc của cây nhị phân hoàn toàn với hai cây con đã là cây thứ thứ tự bộ phận. Vì vậy, chúng ta chỉ cần tiến hành đẩy dữ liệu chứa trong đỉnh i xuống vị trí thích hợp, như chúng ta đã làm trong bước hai của thuật toán DeleteMin. Ví dụ, xét cây trong hình 10.4 a. Bắt đầu từ đỉnh i=4, đỉnh này có khóa nhỏ hơn khóa của đỉnh con nó, nên không cần làm gì cả. Với i=3, ta đẩy dữ liệu trong đỉnh 3 xuống đỉnh 8 để có cây trong hình 10.4
c. Với i = 2, đẩy dữ liệu ở đỉnh 2 xuống đỉnh 6, ta có cây trong hình 10.4c. Với i=1, đẩy dữ liệu ở đỉnh 1 xuống đỉnh 9, nhận được cây trong hình 10.4d. Với i=0, ta cần đẩy dữ liệu ở gốc cây xuống đỉnh 9 và nhận được cây thứ tự bộ phận như trong hình 10.4 e.



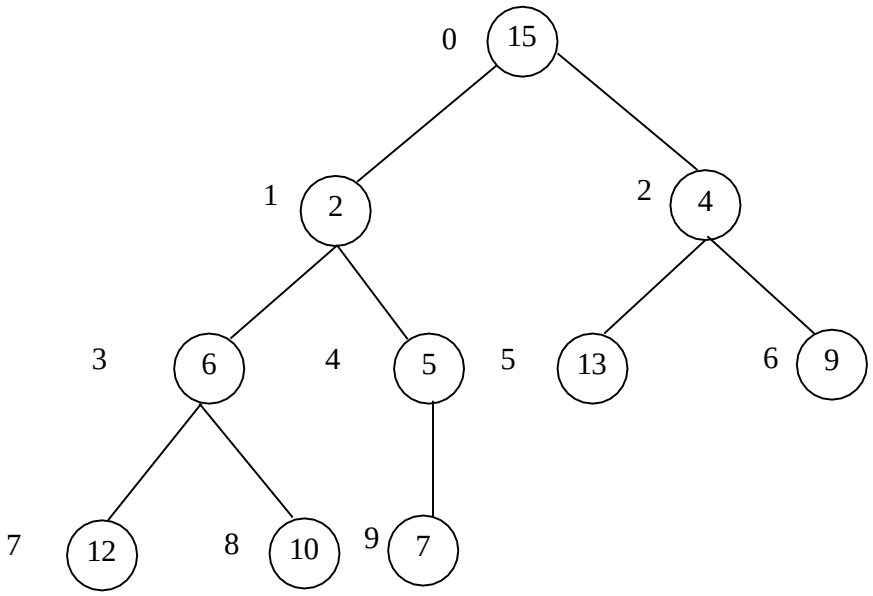
(a)



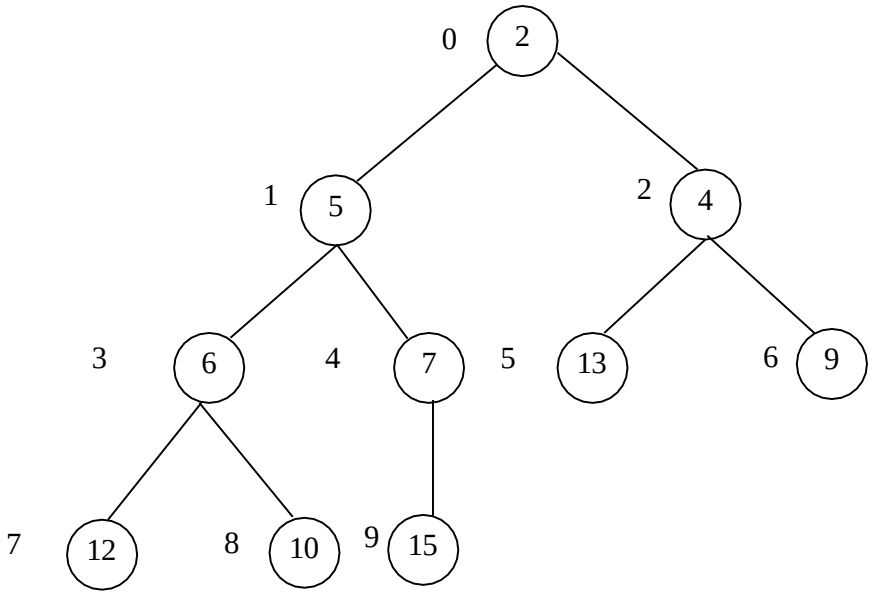
(b)



(c)



(d)



(e)

Hình 10.4. Xây dựng cây thứ tự bộ phận.

Bây giờ chúng ta chứng tỏ rằng, thuật toán xây dựng cây thứ tự bộ phận trên chỉ đòi hỏi thời gian O(n). Rõ ràng rằng, giai đoạn xây dựng cây hoàn toàn từ n dữ liệu chỉ cần thời gian O(n). Vì vậy chúng ta chỉ cần chỉ ra rằng giai đoạn hai cũng chỉ cần thời gian O(n). Trong giai đoạn hai, chúng ta tiến hành xử lý các đỉnh của cây theo thứ tự ngược lại từ đỉnh được đánh số là n-1 tới đỉnh 0. Với mỗi đỉnh ta tiến hành đẩy dữ liệu chứa trong đỉnh đó tới vị trí thích hợp, vị trí này cùng lắm là một lá của cây. Vì vậy thời gian cho xử lý mỗi đỉnh là độ cao của đỉnh đó. Chúng ta tính tổng độ cao của tất cả các đỉnh trong cây. Đỉnh gốc có độ cao h, ở mức 1 có 2 đỉnh với độ cao là h-1,…. Ở mức i có 2^i đỉnh với độ cao là h-i, do đó tổng độ cao của các đỉnh là:

$$f(n) = 2^0.h + 2^1.(h - 1) + \ldots + 2^{h-2}.2 + 2^{h-1}.1$$

$$= \sum_{i=0}^{h-1} 2^i (h - i)$$

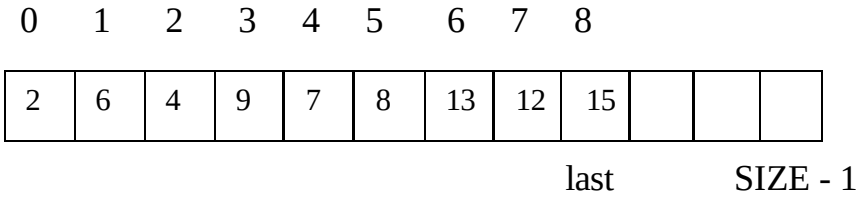
$$= h\sum_{i=0}^{h-1} 2^i - \sum_{i=0}^{h-1} i2^i$$

$$= h(2^h - 1) - [(h - 2)2^h + 2] = 2^{h+1} - h - 2$$

Chúng ta đã biết rằng, độ cao h của cây hoàn toàn n đỉnh là $h \approx \log(n+1)$. Từ đó, $f(n) \approx 2n - \log(n+1)$ hay f(n) = O(n). Như vậy giai đoạn hai chỉ đòi hỏi thời gian O(n).

10.4 CÀI ĐẶT HÀNG ƯU TIÊN BỞI CÂY THỨ TỰ BỘ PHẬN

Trong mục 8.3 chúng ta đã đưa ra cách cài đặt cây nhị phân hoàn toàn bởi mảng. Cây thứ tự bộ phận là cây nhị phân hoàn toàn, do đó có thể cài đặt cấy thứ tự bộ phận bởi mảng. Chẳng hạn, cây thứ tự bộ phận trong hình 10.1 có thể được lưu trong mảng như sau:



Chúng ta sẽ sử dụng một mảng data với cỡ là SIZE để lưu các dữ liệu chứa trong các đỉnh của cây thứ tự bộ phận. Dữ liệu chứa trong đỉnh được đánh số i sẽ được lưu trong thành phần data[i] của mảng, i sẽ chạy từ 0 đến chỉ số sau cùng là last, và không gian trong mảng từ chỉ số last+1 tới SIZE-1 là không gian chưa sử dụng. Nhớ lại rằng, với cách lưu này, nếu một đỉnh được lưu trong data[i] thì đỉnh con trái của nó được lưu trong data[2*i+1], còn đỉnh con phải được lưu trong data[2*i+2] và đỉnh cha của nó được lưu trong data[(i-1)/2].

Sử dụng phương pháp biểu diễn hàng ưu tiên bởi cây thứ tự bộ phận (khóa của các đỉnh là giá trị ưu tiên), chúng ta cài đặt KDLTT hàng ưu tiên bởi lớp PriQueue được mô tả trong hình 10.5. Lớp PriQueue là lớp phụ thuộc tham biến kiểu Item, trong đó Item là kiểu của các phần tử trong hàng ưu tiên. Các phần tử trong hàng ưu tiên chứa một thành phần là giá trị ưu tiên (priority), và để đơn giản cho cách viết ta giả thiết rằng, có thể truy cập trực tiếp tới giá trị ưu tiên của các phần tử. Chúng ta có thể cài đặt hàng ưu tiên bởi mảng động, như chúng ta đã làm khi cài đặt danh sách (xem mục 4.3). Song để tập trung sự chú ý tới các kỹ thuật được sử dụng trong các phép toán hàng ưu tiên, chúng ta cài đặt hàng ưu tiên bởi mảng tĩnh data. Cài đặt bởi mảng động để lại cho độc giả, xem như bài tập. Lớp PriQueue chứa hai hàm kiến tạo, hàm kiến tạo mặc định làm nhiệm vụ tạo ra một hàng ưu tiên rỗng, và một hàm kiến tạo khác xây dựng nên hàng ưu tiên từ các dữ liệu đã được lưu trong một mảng theo phương pháp trong mục 10.3.2. Các hàm thành phần còn lại là các hàm thực hiện các phép toán trên hàng ưu tiên. Một điều cần lưu ý là, chúng ta đưa vào lớp PriQueue một hàm ẩn ShiftDown(int i), hàm này thực hiện bước hai trong thuật toán DeleteMin. Hàm này được sử dụng để cài đặt hàm kiến tạo thứ hai và hàm DeleteMin.

```
template <class Item>
class       PriQueue
{
  public:
    static const int SIZE = 1000;
    PriQueue()  // Hàm kiến tạo mặc định
        {last = -1}
    PriQueue(Item* element, int n);
    //Xây dựng hàm ưu tiên từ n phần tử được lưu trong mảng element.
    bool   Empty() const
         { return last < 0;}
    Item &   FindMin() const;
    // Trả về phần tử có giá trị ưu tiên nhỏ nhất
    void     Insert(const Item &  object, bool &  suc);
    // Xen object vào hàng ưu tiên. Nếu thành công biến suc nhận giá
    // trị true, nếu không giá trị của nó là false.
    Item &   DeleteMin();
    // Loại và trả về đối tượng có giá trị ưu tiên nhỏ nhất.
  private:
     Item     data[SIZE];
     int      last;
     void     ShiftDown(int i);
     // Đẩy dữ liệu trong đỉnh i xuống vị trí thích hợp
};
```

Hình 10.5. Lớp PriQueue.

Sau đây chúng ta cài đặt các hàm thành phần của lớp PriQueue. Trước hết là hàm tìm đối tượng có giá trị ưu tiên nhỏ nhất.

```
template <class Item>
Item &   PriQueue <Item> ::  FindMin()
{
      assert( last >= 0);
      return      data[0];
}
```

Hàm ẩn ShiftDown(int i) thực hiện bước hai trong thuật toán DeleteMin(). Chúng ta sử dụng biến parent ghi lại chỉ số của đỉnh cha và biến child ghi lại chỉ số của đỉnh con có giá trị ưu tiên nhỏ hơn giá trị ưu tiên của đỉnh con kia (nếu có). Biến x lưu lại phần tử chứa trong đỉnh i. Ban đầu giá trị của parent là i, mỗi khi giá trị ưu tiên của x lớn hơn giá trị ưu tiên của đỉnh child thì phần tử chứa trong đỉnh child được đẩy lên đỉnh parent và ta đi xuống đỉnh child. Đến khi giá trị ưu tiên của x nhỏ hơn hoặc bằng giá trị ưu tiên của đỉnh child hoặc khi đỉnh parent là đỉnh lá thì phần tử x được đặt vào đỉnh parent. Hàm ShiftDown được cài đặt như sau:

```
template <class Item>
void  PriQueue <Item> ::  ShiftDown(int i)
{
        Item  x = data[i];
        int      parent = i;
        int      child  =  2*parent+1; //đỉnh con trái.
while (child <= last)
  {
        int     right  = child+1; //đỉnh con phải
        if (right <= last  && data[right].priority < data[child].priority)
                child = right;
        if (x.priority > data[child].priority)
                {
                   data[parent] = data[child];
                   child = 2*parent+1;
                }
        else    break;
  }
        data[parent] = x;
}
```

Sử dụng hàm ShiftDown, chúng ta dễ dàng cài đặt được các hàm DeleteMin và hàm kiến tạo.

```
template <class Item>
Item &  PriQueue<Item> ::  DeteleMin()
{
```

```
    assert(last >= 0);
    data[0] = data[last--];  //chuyển phần tử trong đỉnh lá ngoài cùng
                             //bên phải ở mức thấp nhất lên gốc.
    ShiftDown(0);
}

template <class Item>
PriQueue<Item> ::  PriQueue(Item* element, int n)
{
     int i;
     for (i = 0 ; i < n ; i++)
              data[i]  =  element[i];
     last  =  n-1;
     for (i = n/2 - 1; i >= 0 ; i--)
              ShiftDown(i);
}
```

Bây giờ chúng ta sẽ cài đặt hàm Insert. Hàm Insert được cài đặt theo kỹ thuật tương tự như hàm ShiftDown. Biến child ban đầu nhận giá trị là last+1. Biến parent được tính theo biến child, parent=(child-1)/2. Mỗi khi giá trị ưu tiên của đỉnh cha lớn hơn giá trị ưu tiên của đối tượng cần xen vào thì phần tử trong đỉnh cha được đẩy xuống đỉnh con và ta đi lên đỉnh cha. Khi đạt tới đỉnh cha có giá trị ưu tiên nhỏ hơn hay bằng giá trị ưu tiên của đối tượng cần xen vào thì đối tượng được đặt vào đỉnh con. Hàm Insert được cài đặt như sau:

```
template <class Item>
void  PriQueue<Item>:: Insert(const Item & object, bool & suc)
{
  if (last = = SIZE-1)
             suc =  false;
  else {
          suc =  true;
          int   child = ++last;
          while (child > 0)
            {
               int   parent = (child-1)/2;
               if (data[parent].priority > object.priority)
```

```
                {
                    data[child] =  data[parent];
                    child =  parent;
                }
            else     break;
        }
    data[child] = object;
  }
}
```

10.5 NÉN DỮ LIỆU VÀ MÃ HUFFMAN

Nén dữ liệu (data compression) hay còn gọi là nén file là kỹ thuật rút ngắn cỡ của file được lưu trữ trên đĩa. Nén dữ liệu không chỉ nhằm tăng khả năng lưu trữ của đĩa mà còn để tăng hiệu quả của việc truyền dữ liệu qua các đường truyền bởi modem. Chúng ta luôn luôn giả thiết rằng, file dữ liệu chứa các dữ liệu được biểu diễn như là một xâu ký tự được tạo thành từ các ký tự trong bảng ký tự ASCII. Để lưu các dữ liệu dưới dạng file, chúng ta cần phải **mã hoá** các dữ liệu dưới dạng một xâu nhị phân (xâu bit), trong đó mỗi ký tự được biểu diễn bởi một dãy bit nào đó được gọi là từ mã (code word). Quá trình sử dụng các từ mã để biến đổi xâu bit trở về xâu ký tự nguồn được gọi là **giải mã**.

Trước hết ta nói tới cách mã hoá đơn giản: các từ mã có đội dài bằng nhau. Bởi vì bảng ASCII chứa 128 ký tự, nên chỉ cần các từ mã 8 bit là đủ để biểu diễn 128 ký tự. Cần chú ý rằng, nếu xâu ký tự được tạo thành từ p ký tự thì mỗi từ mã cần ít nhất $\lceil \log p \rceil$ bit. Ví dụ, nếu xâu ký tự nguồn chỉ chứa các ký tự a,b,c,d,e và f thì chỉ cần 3 bit cho mỗi từ mã, chẳng han ta có thể mã a = 000, b = 001, c = 010, d = 011, e = 100, và f = 101. Phương pháp mã hoá sử dụng các từ mã có độ dài bằng nhau có ưu điểm là việc giải mã rất đơn giản, nhưng không tiết kiệm được không gian trên đĩa, chẳng hạn trong ví dụ trên, nếu xâu nguồn có độ dài n thì

số bít cần thiết là 3.n bất kể số lần xuất hiện của các ký tự trong xâu nhiều hay ít.

Chúng ta có thể giảm số bít cần thiết để mã xâu ký tự nguồn bằng cách sử dụng các từ mã có độ dài thay đổi, tức là các ký tự được mã hoá bởi các từ mã có độ dài khác nhau. Chúng ta sẽ biểu diễn ký tự có tần xuất xuất hiện cao bởi từ mã ngắn, còn ký tự có tần xuất xuất hiện thấp với các từ mã dài hơn. Nhưng khi mà các từ mã có độ dài thay đổi, để giải mã chúng ta cần phải có cách xác định bit bắt đầu và bit kết thúc một từ mã trong xâu bit. Nếu một từ mã có thể là đoạn đầu (tiền tố) của một từ mã khác thì không thể giải mã được duy nhất. Chẳng hạn, khi xâu nguồn chứa các ký tự a, b, c, d, e và f và ta sử dụng các từ mã: a = 00, b = 111, c = 0011, d = 01, e = 0 và f = 1 (từ mã 00 là tiền tố của từ mã 0011, từ mã 0 là tiền tố của từ mã 01, 00, 0011, từ mã 1 là tiền tố của từ mã 111); khi đó, xâu bit 010001 có thể giải mã là các từ dad, deed, efad,… Do đó, để đảm bảo một xâu bit chỉ có thể là mã của một xâu nguồn duy nhất, chúng ta có thể sử dụng các từ mã sao cho không có từ mã nào là tiền tố của một từ mã khác.

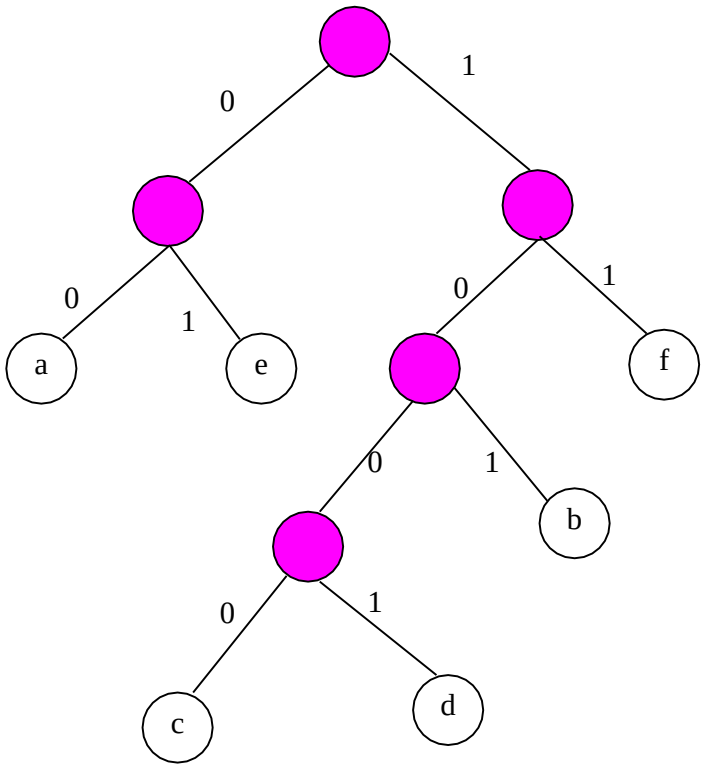
Một hệ từ mã mà không có từ mã nào là tiền tố của từ mã khác sẽ được gọi là mã tiền tố (prefix code). Người ta xây dựng mã tiền tố dựa trên tần xuất của các ký tự trong xâu nguồn. Phương pháp mã này cho phép rút bớt đáng kể số bit cần thiết. Ví dụ, chúng ta lại giả thiết xâu nguồn được tạo thành từ các ký tự a, b, c, d, e và f, xâu có độ dài 10000 ký tự, trong đó ký tự a xuất hiện 3000 lần, b xuất hiện 1000 lần, c xuất hiện 500 lần, d xuất hiện 500 lần, e xuất hiện 3000 lần và f xuất hiện 2000 lần. Chúng ta có thể sử dụng mã tiền tố như sau, a = 00, b = 101, c = 1000, d = 1001, e = 01 và f = 11, khi đó số bit cần thiết là:

3000.2+1000.3+500.4+500.4+3000.2+2000.2 = 23000 bit

Trong khi đó, nếu sử dụng các từ mã cùng độ dài là 3 bit thì số bit đòi hỏi là 10000.3 = 30000 bit.

Như vậy, vấn đề được đặt ra bây giờ là, làm thế nào từ một xâu ký tự nguồn, ta xây dựng được mã tiền tố sao cho số bit cần thiết trong xâu mã là ít nhất có thể được. Mã tiền tố thoả mãn tính chất đó được gọi là mã tiền tố tối ưu. Dưới đây chúng ta sẽ trình bày phương pháp mã được đề xuất bởi Huffman.

Chúng ta sẽ biểu diễn mã tiền tố dưới dạng cây nhị phân, tại mỗi đỉnh của cây, nhánh bên trái được gắn nhãn là 0, nhánh bên phải được gắn nhãn là 1, mỗi ký tự được lưu trong một đỉnh lá của cây. Khi đó, từ mã của mỗi ký tự là xâu bit tạo thành từ các nhãn trên đường đi từ gốc tới đỉnh lá chứa ký tự đó. Chẳng hạn, mã tiền tố đã đưa ra ở trên (a=00, b=101, c=1000, d=1001, e=01 và f=11) được biểu diễn bởi cây nhị phân trong hình 10.6.



Hình 10.6. Cây nhị phân biểu diễn mã tiền tố.

Chúng ta có nhận xét rằng, khi một mã có thể biểu diễn bởi cây nhị phân như đã mô tả ở trên thì mã sẽ là mã tiền tố, điều đó được suy ra từ tính chất các ký tự chỉ được chứa trong các đỉnh lá của cây. Mặt khác, chúng ta có thể giải mã rất đơn giản nếu chúng ta biểu diễn mã tiền tố bởi cây nhị phân. Thuật toán giải mã như sau: đọc xâu bit bắt đầu từ bit đầu tiên, và bắt đầu từ gốc cây ta đi theo nhánh trái nếu bit được đọc là 0 và đi theo nhánh phải nếu bit là 1. Tiếp tục đọc bit và đi như thế cho tới khi gặp một đỉnh lá, ký tự được lưu trong đỉnh lá đó được viết ra xâu kết quả. Lại đọc ký tự tiếp theo và lại bắt đầu từ gốc đi xuống, lặp lại quá trình đó cho tới khi toàn bộ xâu bit được đọc qua.

Thuật toán Huffman sử dụng hàng ưu tiên để xây dựng mã tiền tố dưới dạng cây nhị phân. Cây kết quả được gọi là **cây Huffman** và mã tiền tố ứng với cây đó được gọi là **mã Huffman.** Trước hết xâu ký tự nguồn cần được đọc qua để tính tần xuất của mỗi ký tự trong xâu. Ta ký hiệu tần xuất của ký tự c là f(c). Với mỗi ký tự xuất hiện trong xâu nguồn, ta tạo ra một đỉnh chứa ký tự đó, đỉnh này được xem như gốc của cây nhị phân chỉ có một
đỉnh. Chúng ta ký hiệu tập đỉnh đó là Γ. Tập Γ được lấy làm đầu vào của thuật toán Huffman. Ý tưởng của thuật toán Huffman là từ tập các cây chỉ có một đỉnh, tại mỗi bước ta kết hợp hai cây thành một cây, và lặp lại cho đến khi nhận được một cây nhị phân kết quả. Chú ý rằng, chúng ta cần xây dựng được cây nhị phân sao cho đường đi từ gốc tới đỉnh lá chứa ký tự có tần xuất cao cần phải ngắn hơn đường đi từ gốc tới đỉnh lá chứa ký tự có tần xuất thấp hơn. Vì vậy, trong quá trình xây dựng, mỗi đỉnh gốc của cây nhị phân được gắn với một giá trị ưu tiên được tính bằng tổng các tần xuất của các ký tự chứa trong các đỉnh lá của nó, và tại mỗi bước ta cần chọn hai cây nhị phân có mức ưu tiên nhỏ nhất để kết hợp thành một. Do đó trong thuật toán, ta sẽ sử dụng hàng ưu tiên P để lưu các đỉnh gốc của các cây nhị phân, giá trị ưu tiên của đỉnh v sẽ được ký hiệu là f(v), đỉnh con trái của đỉnh v được ký

hiệu là leftchild(v), đỉnh con phải là rightchild(v). ban đâu hàng ưu tiên P được tạo thành từ các tập đỉnh Γ đã nói ở trên, ta giả sử Γ chứa n đỉnh.

Thuật toán Huffman gồm các bước sau:

1. Khởi tạo hàng ưu tiên P chứa các đỉnh trong tập Γ
2. Bước lặp

For (i = 1 ; i <= n – 1 ; i++)

v_1 = DeleteMin(P);

v_2 = DeleteMin(P);

Tạo ra đỉnh v mới với:

leftchild(v) = v_1;

leftchild(v) = v_2;

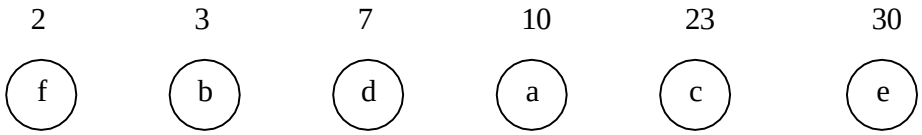
f(v) = f(v_1) + f(v_2);

Insert(P,v);

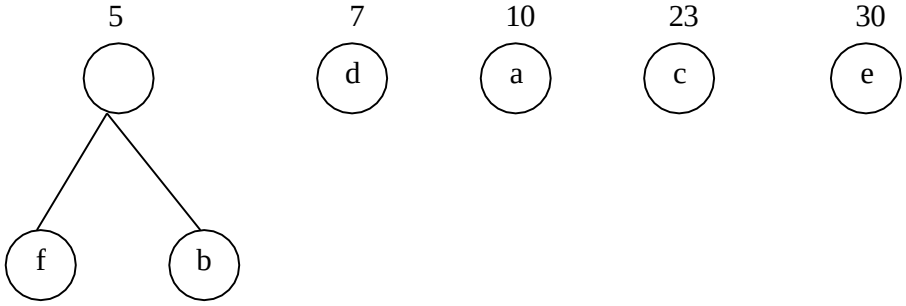
Có thể chứng minh được rằng, mã Huffman là mã tiền tố tối ưu. Một điều nữa cần lưu ý là mã Huffman không phải là duy nhất, bởi vì trong bước 2.3 chúng ta cũng có thể đặt rightchild(v) = v_1, leftchild(v) = v_2.

Để hiểu rõ hơn thuật toán Huffman, ta xét ví dụ sau. Giả sử xâu ký tự nguồn chứa các ký tự với tần xuất như sau: a xuất hiện 10 lần, b xuất hiện 3 lần, c xuất hiện 23 lần, d xuất hiện 7 lần, e xuất hiện 30 lần và f xuất hiện 2 lần. Như vậy ban đầu hàng ưu tiên P chứa các đỉnh được chỉ ra trong hình 10.7a, trong đó giá trị ưu tiên(tần xuất) được ghi trên mỗi đỉnh. Tại mỗi bước ta kết hợp hai cây có giá trị ưu tiên của gốc nhỏ nhất thành một cây mới. Vì có 6 ký tự, nên số bước là 5, kết quả của mỗi bước là một rừng cây được cho trong các hình từ 10.6 b đến 10.6 f

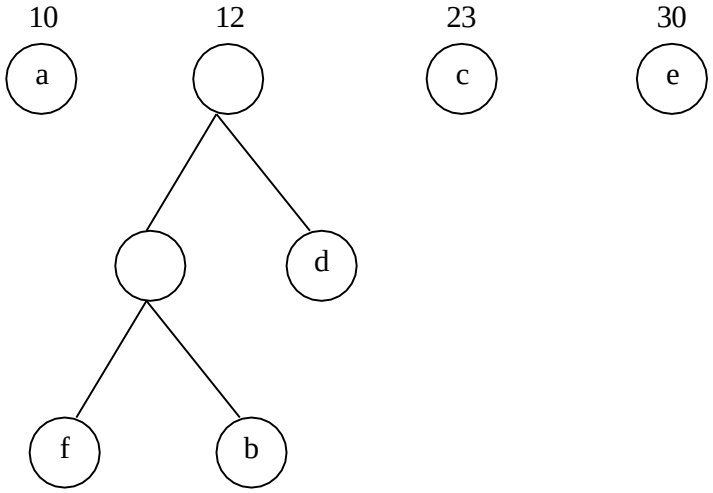
(a)


2
3
7
10
23
30
f
b
d
a
c
e

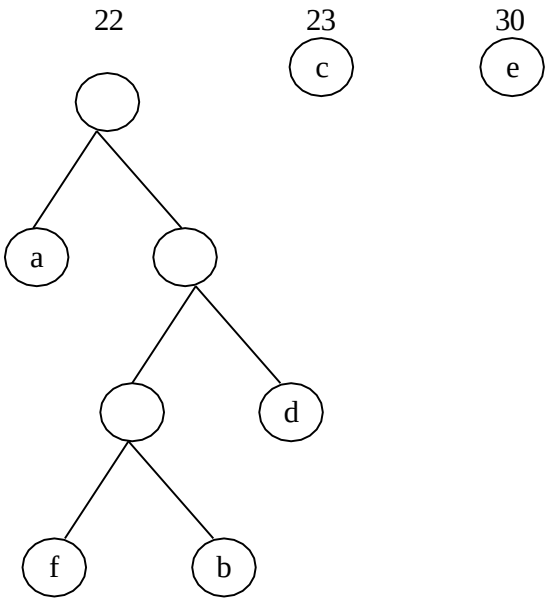
(b)


5
7
10
23
30
d
a
c
e
f
b

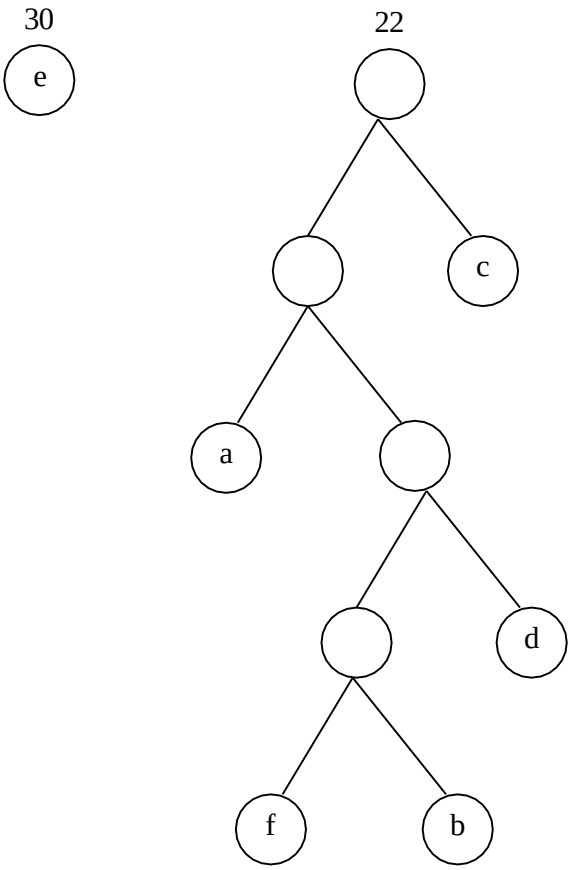
(c)


10
12
23
30
a
c
e
d
f
b

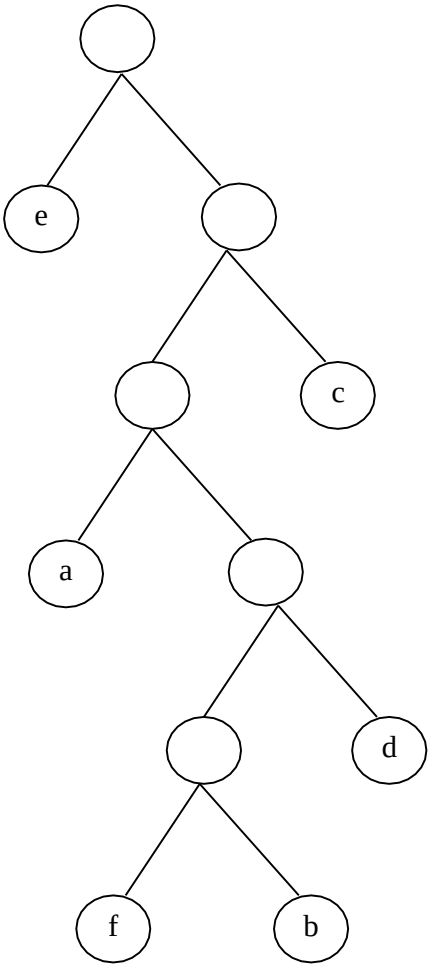
(d)


22
23
30
c
e
a
d
f
b

(e)



(f)



Hình 10.6. Quá trình xây dựng cây Huffman.

Từ cây Huffman trong hình 10.6, ta thành lập được các từ mã như sau: e = 0, c = 11, a = 100, d = 1011, f = 10100, và b = 10101.

BÀI TẬP

1. Ta có thể biểu diễn hàng ưu tiên bởi danh sách theo thứ tự bất kỳ hoặc bởi danh sách được sắp theo giá trị ưu tiên tăng dần (hoặc giảm dần). Hãy cài đặt lớp hàng ưu tiên bằng cách thừa kế private từ lớp cơ sở Dlist (hoặc Llist) trong các trường hợp sau:

a. Hàng ưu tiên được biểu diễn bởi danh sách theo thứ tự bất kỳ.
b. Hàng ưu tiên được biểu diễn bởi danh sách được sắp theo giá trị ưu tiên giảm dần (tăng dần).

2. Giả sử rằng, các đối tượng khác nhau của hàng ưu tiên có thể có cùng một giá trị ưu tiên. Lấy giá trị ưu tiên làm khoá, hãy đưa ra cách biểu diễn hàng ưu tiên dưới dạng cây tìm kiếm nhị phân. Hãy thiết kế và cài đặt lớp hàng ưu tiên khi hàng ưu tiên được biểu diễn bởi cây tìm kiếm nhị phân theo cách đã đưa ra.

3. Cho một dãy đối tượng với các giá trị ưu tiên là 10, 12, 1, 14, 6, 5, 8, 15 và 9.
   a. Từ cây thứ tự bộ phận rỗng, hãy xen vào từng đối tượng theo thứ tự đã liệt kê. Vẽ ra cây kết quả.
   b. Hãy sử dụng thuật toán xây dựng cây thứ tự bộ phận cho dãy đối tượng trên. So sánh cây này với cây nhận được bằng cách xen vào từng đối tượng ở câu a