**TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI PHÂN HIỆU TẠI TP. HỒ CHÍ MINH**

**BỘ MÔN CÔNG NGHỆ THÔNG TIN**



**BÁO CÁO BÀI TẬP LỚN**

**ĐỀ TÀI: XÂY DỰNG ỨNG DỤNG QUẢN LÍ TÍNH TIỀN TRONG CÁC CỬA HÀNG**

Giảng viên hướng dẫn: TRẦN THỊ DUNG

Sinh viên thực hiện: NGUYỄN THÀNH TIẾN

VÕ VĂN TRÍ

MẠNH QUỐC TOÀN

ĐOÀN MINH THÔNG

TRẦN THỊ THU TRANG

Lớp: CQ.60.CNTT

Khóa: 60

Tp. Hồ Chí Minh, năm 2020

**LỜI CẢM ƠN**

Đầu tiên, em xin gửi lời cảm ơn chân thành đến Trường Đại học Giao thông Vận tải – Phân hiệu tại thành phố Hồ Chí Minh đã đưa bộ môn Công nghệ thông tin vào giảng dạy. Đặc biệt, em xin gửi lời cảm ơn dến giảng viên bộ môn – Cô Trần Thị Dung đã dạy dỗ, truyền đạt cho chúng em những kiến thức quý báu trong suốt thời gian học tập vừa qua và chúng em đã có thêm cho mình rất nhiều kiến thức để bổ sung thêm vào hành trang tri thức của mình.

Trong chương trình học môn Cấu trúc dữ liệu & giải thuật và Lập trình hướng đối tượng thì chúng em cũng gặp rất nhiều khó khăn trong học tập bởi vì hai môn này đối với chứng em thì cũng không phải gọi là dễ. Nhưng nhờ sự giảng dạy, giúp đỡ tận tình của cô mà chúng em đã biết đượ thêm rất nhiều kiến thức bổ ích. Và cô đã cùng góp ý, chỉ dẫn để xây dựng đề tài cho bài tập lớn là “Xây dựng ứng dụng quản lí tính tiền trong các cửa hàng”.

Sau một thời gian nỗ lực để thực hiện thì đề tài cũng đã hoàn thành. Nhưng không sao tránh khỏi những sai sót do chúng em cũng còn chưa có nhiều kinh nghiệm thực tế cũng như kiến thức còn hạn hẹp. Chúng em kính mong nhận được sự góp ý và nhận xét chân thành từ Cô, để chứng em có thể hoàn thiện và hoàn thành tốt hơn đề tài bài tập lớn của mình. Và cũng như là để chúng em có thể rút ra được những kinh nghiệm để cho những đề tài sau của mình được tốt hơn.

Lời sau cùng thì chúng em chúc Cô luôn có một sức khỏe tốt và đạt được nhiều thành công hơn nữa trên con đường giảng dạy của mình.

Tp. Hồ Chí Minh, ngày tháng năm 2020

Sinh viên thực hiện

MỤC LỤC

CHƯƠNG 1. TỔNG QUAN VỀ ĐỀ TÀI

1. ĐẶT VẤN ĐỀ
   1. Lý do chọn đề tài
   2. Mục tiêu nghiên cứu

CHƯƠNG 2. LISTS, STACKS AND QUEUES

* 1. Các kiểu dữ liệu trừu tượng (ADTS)
  2. Danh sách ADT
     1. Triển khai danh sách theo mảng đơn giản
     2. Danh sách liên kết đơn
  3. Vector và danh sách trong STL
     1. Vòng lặp
     2. Ví dụ sử dụng xóa trên danh sách
     3. Hằng số nguyên
  4. Thực hiện vector
  5. Thực hiện danh sách
  6. Ngăn xếp ADT
     1. Mô hình ngăn xếp
     2. Thực hiện các ngăn xếp
     3. Ứng dụng
  7. Hàng đợi ADT
     1. Mô hình hàng đợi
     2. Triền khai hàng đợi theo mảng
     3. Ứng dụng của hàng đợi

CHƯƠNG 3. HASHING (hàm băm)

* 1. Ý tưởng chung
  2. Hàm băm
  3. Chuỗi riêng biệt
  4. Bảng băm không có danh sách được liên kết
     1. Đo tuyến tính
     2. Đo lường bậc hai
     3. Băm kép
  5. Hàm lại
  6. Bảng băm trong Thư viện chuẩn
  7. Bảng băm với quyền truy cập Worst – Case O (1)
     1. Băm hoàn hảo
     2. Hàm băm Cuckoo
     3. Hàm băm lò cò
  8. Băm chung
  9. Hàm băm mở rộng

CHƯƠNG 4. TREES (cây)

* 1. Sơ bộ
     1. Thực hiện cây
     2. Cay traversal với 1 ứng dụng
  2. Cây nhị phân
     1. Thực hiện
     2. Ví dụ cây biểu hiện
  3. Cây tìn kiếm ADT – Cây nhị phân
     1. Contanis
     2. findMin và findMax
     3. Chèn
     4. Xóa
     5. Bộ tạo hủy và sao chép
  4. Cây AVL
     1. Xoay đơn
     2. Xoay đôi
  5. Phát cây
     1. Ý tưởng
     2. Đánh đòn
  6. Tree traversals
  7. B-Cây
  8. Bộ và bản đồ trong thư viện chuẩn
     1. Bộ
     2. Bản đồ
     3. Thực hiện bộ và bản đồ

CHƯƠNG 1. TỔNG QUAN VỀ ĐỀ TÀI

* 1. Lí do chọn đề tài:

Trong xã hội ngày càng phát triển hiện nay, khoa học công nghệ là thứ không thể thiếu đối với mỗi quốc gia, doanh nghiệp, trường học hay mỗi cá nhân, đặc biệt là công nghệ thông tin. Với sự phát triển mọt cách không ngừng nghỉ như vậy của công nghệ thông tin, nó đã giúp giải quyết các công việc học tập, nghiên cứu, quản lí thông tin một cách dễ dàng và tiện lợi. Thấy được tiềm năng đó, các quốc gia, doanh nghiệp, cá nhân, ….đã ứng dụng nó vào thực tiễn cuộc sống để giải quyết các công việc đó một cách nhanh gọn hơn.

Trong những năm gần đây thì nhu cầu về các phần mềm, ứng dụng học tập giải trí ngày càng nhiều hơn. Và trong đó thì các cửa hàng hay là quán ăn thì họ cũng muốn việc tính tiền, quản lý doanh thu trong quán một cách nhanh gọn và chính xác nhất.

Nhận thấy được vấn đề này thì nhóm em cũng đã tiềm hiểu và thực hiện đề tài “ Xây dựng ứng dụng quản lý tính tiền trong các cửa hàng”. Nhằm tạo ra một ứng dụng có thể giúp cho người bán hàng có thể tính tiền một cách nhanh chóng và quản lý được doanh thu của mình trong ngày đó là bao nhiêu.

1.2. Mục tiêu nghiên cứu:

* Xây dựng ứng dụng quản lý tính tiền bằng ngôn ngữ c++, c#
* Ứng dụng phải đáp ứng được giao diện thân thiện với mắt nhìn của người dùng, đầy đủ các chức năng.
* Giúp giảm sự nhầm lẫn cho người tính tiền, và giúp tính tiền một cách nhanh chóng.

1.3. Phạm vi nghiên cứu:

* Nghiên cứu về xây dựng giao diện trên nên của ngôn ngữ lập trình c#.

1.4. Khảo sát thực tế:

Hiện nay thì các cửa hàng, quán ăn xuất hiện ngày càng nhiều. Thì nhu cầu sử dụng các ứng dụng quản lý, tính tiền ngày càng tăng lên. Dựa trên thực tế đó thì nhóm cũng đã xây dựng một ứng dụng quản lí tính tiền đơn giản để giúp cho việc tính tiền nhanh hơn và chính xác hơn.

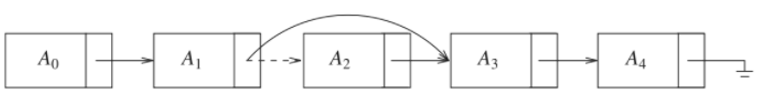
CHƯƠNG 2. LISTS, STACKS AND QUEUES

* 1. Các kiểu dữ liệu trừu tượng (ADTS)
* Là một đối tượng cùng với một tập các phép toán kiểu dữ liệu trừu tượng là những trừu tượng toán học.
* Có thể xem được các đối tượng như: danh sách tập hợp và đồ thị cùng với các hoạt động của chúng như là quảng cáo…
* Đối với tập ADT thì có các hoạt đọng như: thêm vào, xóa bớt, kích cỡ, chứa đựng.
  1. Danh sách ADT
* Chúng ta sẽ có một danh sách có dạng A0,A1,A2…..An-1. Danh sqqachs này có kích thước là N. Vá có một danh sách đặc biệt có kích thước 0 là danh sách trống.
* Đối với bất kì danh sách nào ngoại trừ danh sách trống thì Ai theo sau Ai-1(i<n) và Ai-1 đứng trước Ai(i>0). Phần tử đầu tiên của danh sách là A0 và phần tử cuối cùng là An-1.
* Chúng ta sẽ không xác định tiền nhiệm của A0 hay kế thừa của An-1.
* Một số hoạt động phổ biến trên ADT là printList và makeEmpty, thực hiện những điều hiển nhiên; find trả về vị trí xuất hiện đầu tiên của một mục; chèn và loại bỏ, thường chèn và xóa một số phần tử khỏi một số vị trí trong danh sách; và findKth, trả về phần tử ở một số vị trí (được chỉ định làm đối số)
* Ví dụ: cho một danh sách: 34,12,52,16,12, thì find(52) có thể trả về 2; insert(52) có thể làm cho liệt kê thành 34, 12, x, 52, 16, 12(nếu ta chèn vào vị trí đã cho); và loại bỏ (52) có thể biến danh sách đó thành 34, 12, x, 16, 12.
  + 1. Triển khai danh sách theo mảng đơn giản
* Các mảng cre vớ dung lượng cố định, lớp vecto, lưu trữ bên trong một mảng, cho phép mảng phát triển bằng cách tang gấp đôi công suất của nó khi cần thiết. Điều này giải quyết vấn đề nghiêm trọng nhất với việc sử dụng một mảng cụ thể .
* Triển khai mảng cho phép printList được thực hiện trong tời gian tuyến tính và hoạt động findKth mất thời gian liên tục. Tuy nhiên việc chèn xóa có thể tốn kém tùy thuộc vào vị trí chèn và xóa.
  + 1. Danh sách liên kết đơn
* Để tránh chi phí tuyến tính của việc chèn và xóa ta cần phải đảm bảo rằng danh sách không được lưu trữu liền kề nếu không toàn bộ phần của danh sách sẽ cần được di chuyển.



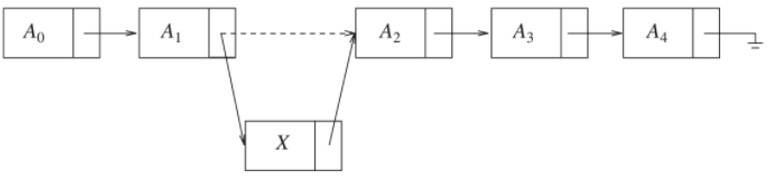
Hình 3.1 Một danh sách liên kết

Danh sách bao gồm một loạt các nút, mỗi nút chứa một phần tử và một liên kết đến nút chứa phần tử kế nhiệm của nó. Liên kết kế tiếp theo của ô cuối cùng trỏ đến nullptr.Để thực thi printList () hoặc find (), chúng ta chỉ bắt đầu ở nút đầu tiên trong danh sách và sau đó duyệt qua danh sách bằng cách nhấp vào các liên kết tiếp theo.



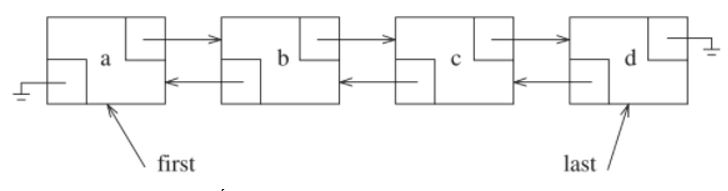
Hình 3.2 Xóa một danh sách khỏi liên kết

Phương thức remove có thể được thực thi trong một lần thay đổi con trỏ tiếp theo. Hình 3.2 cho thấy kết quả của việc xóa phần tử thứ ba trong danh sách ban đầu.



Hình 3.3 Chèn vào danh sách liên kết

Phương thức chèn yêu cầu lấy một nút mới từ hệ thống bằng cách sử dụng lệnh gọi mới và sau đó thực hiện hai thao tác con trỏ tiếp theo. Ý tưởng chung được thể hiện trong Hình 3.3. Đường đứt nét thể hiện con trỏ cũ.



Hình 3.4 Danh sách liên kết kép

Ý tưởng về việc duy trì một liên kết thứ ba tới nút tiếp theo đến cuối cùng không hoạt động vì nó cũng cần được cập nhật trong quá trình xóa. Thay vào đó, chúng tôi có mọi nút duy trì một liên kết đến nút trước đó của nó trong danh sách. Điều này được thể hiện trong Hình 3.4 và được gọi là danh sách liên kết kép.

* 1. Vector và danh sách trong STL

Ngôn ngữ C++ bao gồm trong thư viện của nó việc triển khai các cấu trúc dữ liệu chung. Phần ngôn ngữ này được biết đến với tên gọi Thư viện Mẫu Chuẩn (STL). Danh sách ADT là một trong những cấu trúc dữ liệu được thực hiện trong STL. Chúng ta sẽ thấy một số cấu trúc khác trong Chương B và C. Nói chung, những cấu trúc dữ liệu này được gọi là bộ sưu tập hoặc vùng chứa.

Có hai cách triển khai phổ biến của Danh sách ADT. Vectơ cung cấp triển khai mảng có thể phát triển của Danh sách ADT. Ưu điểm của việc sử dụng vector là nó có thể lập chỉ mục trong thời gian không đổi. Điểm bất lợi là việc chèn các mục mới và loại bỏ các mục hiện có là tốn kém trừ khi các thay đổi được thực hiện ở cuối vector. Danh sách cung cấp việc triển khai danh sách được liên kết kép của Danh sách ADT. Lợi thế của việc sử dụng danh sách là việc chèn các mục mới và loại bỏ các mục hiện có là rẻ với điều kiện là vị trí của các thay đổi được biết đến. Điểm bất lợi là danh sách không dễ lập chỉ mục.

Cả vectơ và danh sách đều không hiệu quả cho các tìm kiếm. Trong suốt danh sách thảo luận này đề cập đến danh sách được liên kết kép trong STL trong khi danh sách (bộ chữ không có phông chữ monospace) đề cập đến Danh sách ADT chung hơn. Cả vectơ và danh sách đều là các mẫu lớp được khởi tạo với loại mục mà chúng lưu trữ. Cả hai đều có một số phương pháp chung. Ba phương pháp đầu tiên được hiển thị thực sự có sẵn cho tất cả các vùng chứa STL:

·       int size () const trả về số phần tử trong vùng chứa.

·       void clear () xóa tất cả các phần tử khỏi vùng chứa.

·       bool blank () const trả về true nếu vùng chứa không chứa phần tử nào và ngược lại là false.

Cả vector và danh sách đều hỗ trợ thêm và xóa khỏi cuối danh sách trong thời gian không đổi. Cả vectơ và danh sách đều hỗ trợ truy cập mục phía trước trong danh sách trong thời gian không đổi. Các hoạt động là :

·       void pushback (const Object x) thêm x vào cuối danh sách.

·       void popback () xóa đối tượng ở cuối danh sách.

·       const Object back () const trả về đối tượng ở cuối danh sách (một trình đột biến trả về tham chiếu cũng được cung cấp).

·       const Object front () const trả về đối tượng ở đầu danh sách (một trình thay đổi trả về tham chiếu cũng được cung cấp).

 Bởi vì một danh sách được liên kết kép cho phép các thay đổi hiệu quả ở phía trước nhưng một vectơ thì không nên hai phương thức sau chỉ khả dụng cho danh sách :

·       void pushfront (const Object x) thêm x vào phía trước danh sách.

·       void popfront () loại bỏ đối tượng ở đầu danh sách.

Vectơ có bộ phương thức riêng không nằm trong danh sách. Hai phương pháp cho phép lập chỉ mục hiệu quả. Hai phương thức còn lại cho phép người lập trình xem và thay đổi dung lượng bên trong. Các phương thức này là:

·       Toán tử đối tượng (int idx) trả về đối tượng tại chỉ mục idx trong vectơ không có giới hạn kiểm tra (một trình truy cập trả về tham chiếu hằng số cũng được cung cấp).

·       Đối tượng tại (int idx) trả về đối tượng tại chỉ mục idx trong vectơ có kiểm tra giới hạn (một trình truy cập trả về một tham chiếu không đổi cũng được cung cấp).

·       int dung lượng () const trả về dung lượng bên trong của vector.

·       Void dự trữ (int newCapacity) đặt dung lượng mới. Nếu có sẵn một ước lượng tốt, nó có thể được sử dụng để tránh mở rộng vectơ

* + 1. Vòng lặp

Một số thao tác trên danh sách quan trọng nhất, những thao tác để chèn và xóa khỏi giữa danh sách yêu cầu khái niệm về vị trí. Trong STL, một vị trí được biểu diễn bằng một trình lặp kiểu lồng nhau. Đặc biệt đối với một chuỗi danh sách, vị trí được đại diện bởi trình điều khiển chuỗi kiểu danh sách đối với một vectơ vị trí được đại diện bởi trình điều khiển vectơ lớp học, v.v. Trong việc mô tả một số phương thức, đơn giản chỉ cần sử dụng trình lặp làm tốc ký nhưng khi viết mã, chúng ta sẽ sử dụng tên lớp lồng nhau thực sự.

Ban đầu, có ba vấn đề chính cần giải quyết đầu tiên là làm thế nào một người có được một trình vòng lặp và những hoạt động nào mà bản thân trình vòng lặp có thể thực hiện, thứ ba là các phương thức List ADT yêu cầu trình vòng lặp làm tham số.

***Bắt một lặp lại:***

Đối với vấn đề đầu tiên, danh sách STL (và tất cả các vùng chứa STL khác) xác định một cặp :

·       phương thức trình lặp begin () trả về một trình vòng lặp thích hợp đại diện cho mục đầu tiên trong vùng chứa.

·       iterator end () trả về một trình lặp thích hợp đại diện cho endmarker trong vùng chứa (tức là vị trí sau mục cuối cùng trong vùng chứa).

Phương thức end có vẻ hơi bất thường bởi vì nó trả về một trình lặp nằm ngoài giới hạn. Để xem ý tưởng, hãy xem mã sau đây thường được sử dụng để in các mục trong vectơ v trước khi giới thiệu các vòng lặp for dựa trên phạm vi trong C++11:

for( int i = 0; i != v.size( ); ++i )

    cout << v[ i ] << endl;

Nếu chúng tôi viết lại mã này bằng cách sử dụng trình vòng lặp, chúng tôi sẽ thấy một sự tương ứng tự nhiên với các phương thức bắt đầu và kết thúc:

for( vector<int>::iterator itr = v.begin( ); itr != v.end( ); itr.??? )

cout << itr.??? << endl;

Trong kiểm tra kết thúc vòng lặp, cả iv.size () và itrv.end () đều nhằm kiểm tra xem bộ đếm vòng lặp có vượt quá giới hạn hay không. Đoạn mã cũng đưa chúng ta đến vấn đề thứ hai là trình lặp phải có các phương thức liên kết với nó (các phương thức không xác định này được đại diện bởi).

***Phương thức lặp lại:***

Dựa trên đoạn mã ở trên, rõ ràng là các trình vòng lặp có thể được so sánh với và có thể có các hàm tạo bản sao và toán tử được xác định. Do đó, các trình vòng lặp có các phương thức và nhiều phương thức sử dụng nạp chồng toán tử. Bên cạnh việc sao chép các hoạt động được sử dụng phổ biến nhất trên trình vòng lặp, bao gồm các thao tác sau:

·       itr và itr tiến trình vòng lặp itr đến vị trí tiếp theo. Cả hai dạng tiền tố và hậu tố đều được phép.

·       itr trả về một tham chiếu đến đối tượng được lưu trữ tại vị trí của trình lặp. Tham chiếu được trả về có thể sửa đổi được hoặc không (chúng ta sẽ thảo luận về những chi tiết này ngay sau đây).

·       itr1itr2 trả về true nếu các trình vòng lặp itr1 và itr2 tham chiếu đến cùng một vị trí và ngược lại là false.

·       itr1itr2 trả về true nếu các trình vòng lặp itr1 và itr2 tham chiếu đến một vị trí khác và ngược lại là false.

Với các toán tử này, mã để in sẽ là:

for( vector<int>::iterator itr = v.begin( ); itr != v.end( ); ++itr )

cout << \*itr << endl;

Việc sử dụng quá tải toán tử cho phép một người truy cập vào mục hiện tại sau đó tiến tới mục tiếp theo bằng cách sử dụng itr. Do đó, một giải pháp thay thế cho đoạn trên là:

vector<int>::iterator itr = v.begin( );

while( itr !=v.end( ) )

cout << \*itr++ << endl;

***Các hoạt động vùng chứa yêu cầu trình lặp***

Đối với vấn đề cuối cùng, ba phương pháp phổ biến nhất yêu cầu trình vòng lặp là những phương pháp thêm hoặc xóa khỏi danh sách (một vectơ hoặc danh sách) tại một vị trí được chỉ định:

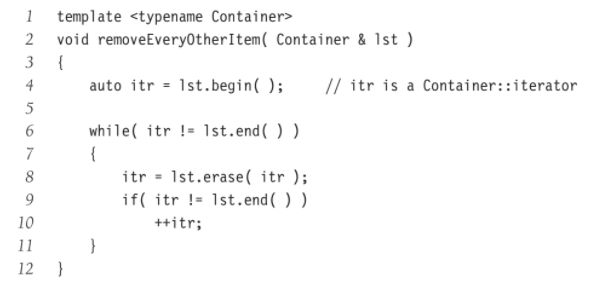
·       chèn trình lặp (iterator pos const Đối tượng x) thêm x vào danh sách trước vị trí được cung cấp bởi pos trình lặp. Đây là một hoạt động thời gian không đổi cho danh sách nhưng không cho vector. Giá trị trả về là một trình lặp đại diện cho vị trí của mục được chèn.

·       phép xóa trình lặp (iterator pos) xóa đối tượng tại vị trí được cung cấp bởi trình lặp. Đây là một hoạt động thời gian không đổi cho danh sách nhưng không cho vector. Giá trị trả về là vị trí của phần tử theo sau pos trước cuộc gọi. Thao tác này làm mất hiệu lực của vị trí hiện đã cũ vì mục vùng chứa mà nó đang xem đã bị xóa.

·       xóa trình lặp (đầu trình lặp bắt đầu trình lặp kết thúc) xóa tất cả các mục bắt đầu từ sau khi khởi động đến nhưng không bao gồm kết thúc. Quan sát rằng toàn bộ danh sách có thể bị xóa bằng lệnh gọi c.erase (c.begin () c.end ()).

1.3.2 Ví dụ sử dụng xóa trên danh sách.

Như một ví dụ, chúng tôi cung cấp một quy trình loại bỏ mọi mục khác trong danh sách bắt đầu bằng mục đầu tiên. Vì vậy, nếu danh sách chứa 6 5 1 4 2 thì sau khi phương thức được gọi, nó sẽ chứa 5 4. Chúng ta thực hiện việc này bằng cách lướt qua danh sách và sử dụng phương pháp xóa trên mỗi mục thứ hai. Trên danh sách, đây sẽ là quy trình thời gian tuyến tính vì mỗi lệnh gọi để xóa cần thời gian không đổi nhưng trong một vectơ, toàn bộ quy trình sẽ mất thời gian bậc hai vì mỗi lệnh gọi xóa không hiệu quả bằng thời gian O (N). Do đó, chúng tôi thường chỉ viết mã cho một danh sách. Tuy nhiên, vì mục đích thử nghiệm, chúng tôi viết một mẫu hàm chung sẽ hoạt động với cả danh sách hoặc vectơ và sau đó cung cấp.



Hình 3.5 Sử dụng trình vòng lặp để xóa mọi mục khác trong Danh sách (hoặc vectơ hoặc danh sách). Hiệu quả cho một danh sách nhưng không hiệu quả cho một vectơ.

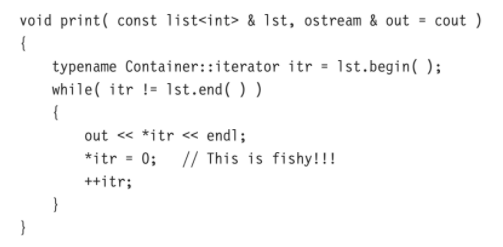
thông tin thời gian. Mẫu chức năng được thể hiện trong Hình 3.1 Việc sử dụng auto ở dòng 4 là một tính năng của C11 cho phép chúng ta tránh được Containeriterator kiểu dài hơn. Nếu chúng tôi chạy mã chuyển một listint, thì mất 0,039 giây cho danh sách 800000 mục và 0,073 giây cho danh sách 1600000 mục và rõ ràng là một quy trình thời gian tuyến tính vì thời gian chạy tăng cùng một yếu tố với kích thước đầu vào. Khi chúng ta vượt qua một vecto, quy trình sẽ mất gần năm phút đối với vectơ 800000 mục và khoảng hai mươi phút đối với vectơ 1600000 mục, thời gian chạy tăng gấp bốn lần khi đầu vào chỉ tăng một hệ số hai là phù hợp với hành vi bậc hai.

* + 1. Hằng số nguyên

Kết quả của itr không chỉ là giá trị của mục mà trình lặp đang xem mà còn là chính mục đó. Sự khác biệt này làm cho các trình lặp rất mạnh mẽ nhưng cũng dẫn đến một số phức tạp. Để xem lợi ích, giả sử chúng ta muốn thay đổi tất cả các mục trong bộ sưu tập thành một giá trị được chỉ định. Quy trình sau đây hoạt động cho cả vectơ và danh sách và chạy trong thời gian tuyến tính. Đây là một ví dụ tuyệt vời về việc viết mã không phụ thuộc vào kiểu chung.



Để xem vấn đề tiềm ẩn, giả sử Vùng chứa c đã được chuyển đến một quy trình bằng cách sử dụng tham chiếu gọi theo hằng số. Điều này có nghĩa là chúng tôi mong đợi rằng không có thay đổi nào được phép đối với c và trình biên dịch sẽ đảm bảo điều này bằng cách không cho phép gọi bất kỳ trình đột biến cs nào. Hãy xem xét đoạn mã sau in ra danh sách các số nguyên nhưng cũng cố gắng tìm cách thay đổi danh sách:



Nếu mã này là hợp pháp thì hằng số của danh sách sẽ hoàn toàn vô nghĩa vì nó sẽ rất dễ bị bỏ qua. Mã không hợp pháp và sẽ không được biên dịch. Giải pháp được cung cấp bởi STL là mọi bộ sưu tập không chỉ chứa một kiểu lồng nhau của trình lặp mà còn chứa kiểu lồng nhau của trình phối hợp. Sự khác biệt chính giữa trình lặp và trình phối hợp là toán tử cho trình phối hợp trả về một tham chiếu không đổi và do đó nó cho trình phối hợp không thể xuất hiện ở phía bên trái của câu lệnh gán.

Hơn nữa, trình biên dịch sẽ buộc bạn phải sử dụng một constiterator để duyệt qua một tập hợp không đổi. Nó làm như vậy bằng cách cung cấp hai phiên bản bắt đầu và hai phiên bản kết thúc như sau:

·       trình lặp bắt đầu()

·       trình phối hợp bắt đầu () const

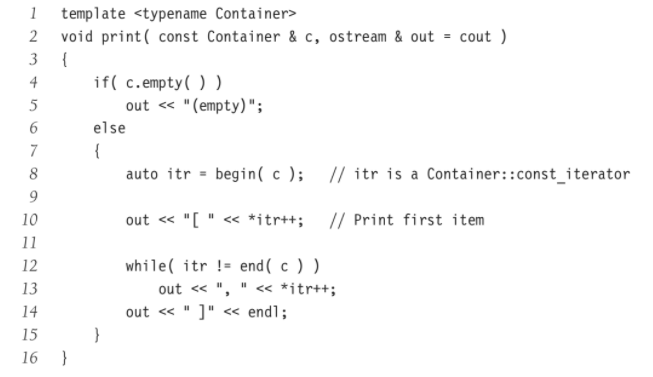
·       trình lặp kết thúc()

·       trình phối hợp kết thúc() const

Hai phiên bản của begin chỉ có thể ở trong cùng một lớp vì hằng số của một phương thức (tức là cho dù nó là một trình truy cập hay trình đột biến) được coi là một phần của chữ ký.

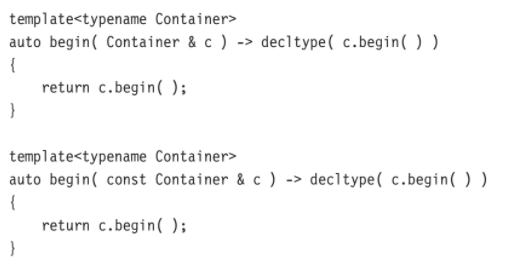
 Nếu bắt đầu được gọi trên một vùng chứa không thay đổi, phiên bản trình đột biến trả về một trình lặp sẽ được gọi. Tuy nhiên, nếu begin được gọi trên một vùng chứa hằng số thì được trả về là một constiterator và giá trị trả về có thể không được gán cho một trình lặp.

 Một tính năng bổ sung trong C++11 cho phép một người viết mã hoạt động ngay cả khi loại Container không có các hàm thành viên bắt đầu và kết thúc. Các chức năng miễn phí không phải thành viên bắt đầu và kết thúc được định nghĩa cho phép một người sử dụng begin (c) ở bất kỳ nơi nào cho phép c.begin (). Viết mã chung bằng begin (c) thay vì c.



Hình 3.6 In bất kì thùng chứa nào

chức năng không thành viên. Việc bổ sung các hàm bắt đầu và kết thúc dưới dạng miễn phí trong C11 được thực hiện bằng cách bổ sung các tính năng ngôn ngữ tự động và kiểu khai báo như được hiển thị trong đoạn mã dưới đây.



Hình 3.7

Trong đoạn mã này, kiểu trả về của begin được suy ra là kiểu của c.begin (). Đoạn mã trong Hình 3.9 sử dụng tự động để khai báo trình vòng lặp (như trong Hình 3.8) và sử dụng các hàm không phải thành viên bắt đầu và kết thúc.

* 1. Thực hiện vector

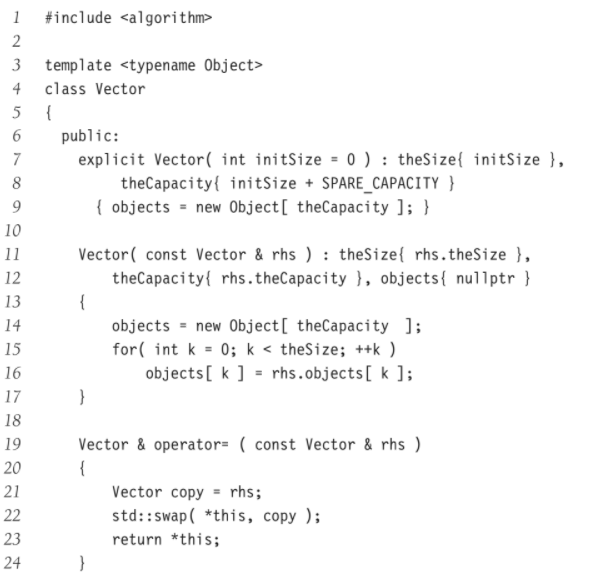
Trong phần này, chúng tôi cung cấp việc triển khai mẫu lớp vectơ có thể sử dụng được. Vectơ sẽ là kiểu hạng nhất có nghĩa là không giống như mảng nguyên thủy trong C++, vectơ có thể được sao chép và bộ nhớ mà nó sử dụng có thể được lấy lại tự động (thông qua trình hủy của nó). Trong Phần 1.5.7, chúng tôi đã mô tả một số tính năng quan trọng của mảng nguyên thủy C++:

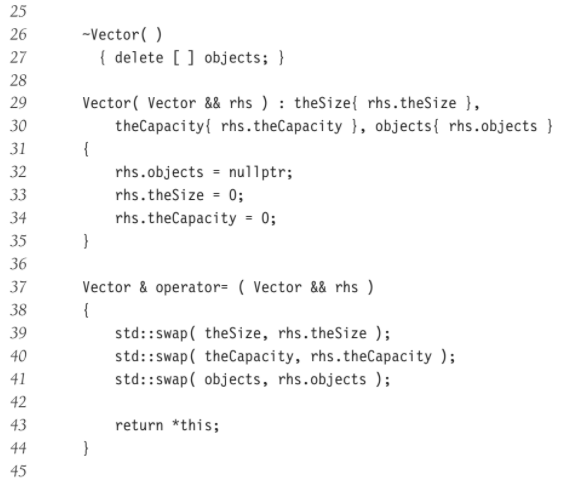
·       Mảng chỉ đơn giản là một biến con trỏ tới một khối bộ nhớ mà kích thước mảng thực tế phải được lập trình viên duy trì riêng biệt.

·       Khối bộ nhớ có thể được cấp phát thông qua mới nhưng sau đó phải được giải phóng thông qua xóa.

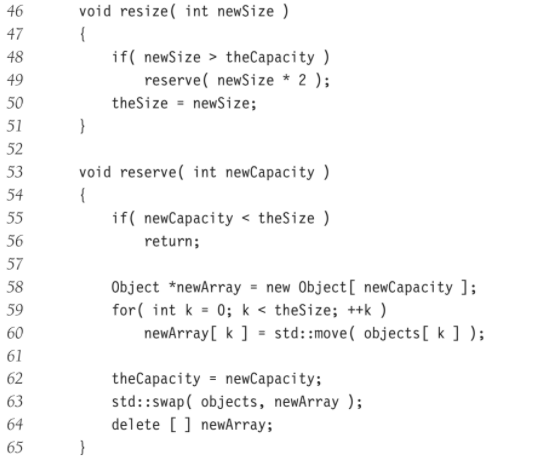
·       Khối bộ nhớ không thể được thay đổi kích thước (nhưng một khối mới có lẽ lớn hơn có thể được lấy và khởi tạo với khối cũ và sau đó khối cũ có thể được giải phóng).

Hình 3.8 và Hình 3.9 cho thấy lớp Vector. Giống như đối tác STL của nó, việc kiểm tra lỗi có giới hạn. Sau đó, chúng tôi sẽ thảo luận ngắn gọn về cách kiểm tra lỗi có thể được cung cấp. Như được hiển thị trên các dòng 118 đến 120, Vector lưu trữ dung lượng kích thước và mảng nguyên thủy làm thành viên dữ liệu của nó. Hàm tạo tại các dòng từ 7 đến 9 cho phép người dùng chỉ định kích thước ban đầu được mặc định là 0. Sau đó, nó khởi tạo các thành viên dữ liệu có dung lượng lớn hơn một chút so với kích thước để có thể thực hiện một số lần phản hồi mà không làm thay đổi dung lượng. Hàm tạo bản sao được hiển thị từ dòng 11 đến dòng 17 tạo một Vector mới và sau đó có thể được sử dụng bởi một toán tử triển khai thông thường sử dụng thành ngữ tiêu chuẩn là hoán đổi trong một bản sao. Thành ngữ này chỉ hoạt động nếu hoán đổi được thực hiện bằng cách di chuyển mà bản thân nó yêu cầu thực hiện hàm khởi tạo di chuyển và toán tử di chuyển được hiển thị ở dòng 29 đến 44. Một lần nữa, những thành ngữ này sử dụng các thành ngữ rất chuẩn. Thực hiện toán tử gán bản sao bằng cách sử dụng phương thức tạo bản sao và hoán đổi trong khi đơn giản chắc chắn không phải là phương pháp hiệu quả nhất, đặc biệt trong trường hợp cả hai Vectơ có cùng kích thước. Trong trường hợp đặc biệt có thể được kiểm tra nó có thể hiệu quả hơn nếu chỉ cần sao chép từng phần tử một bằng toán tử Objects.





Hình 3.8 Lớp vecto (Phần 1/2)

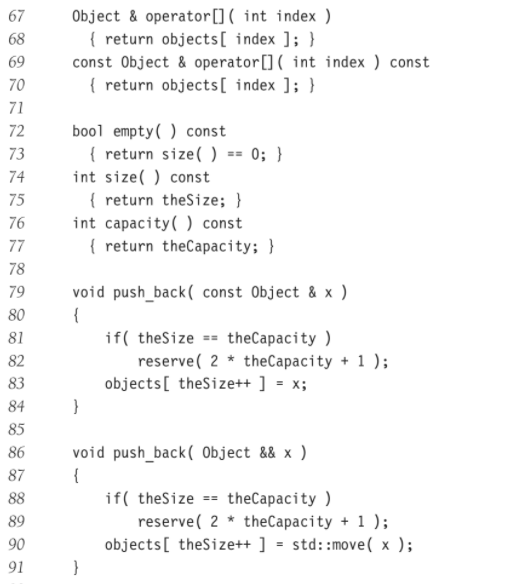


Hình 3.8 Lớp vecto (Phần 2/2)

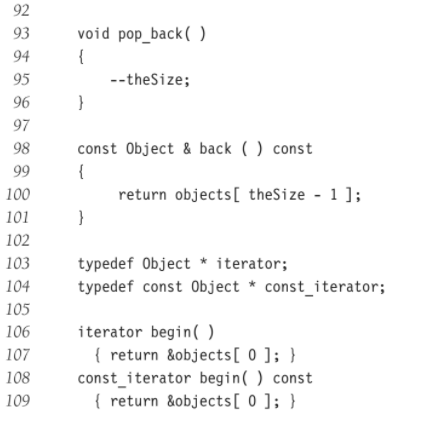
Quy trình thay đổi kích thước được hiển thị tại các dòng 46 đến 51. Mã chỉ cần đặt thành viên dữ liệu theSize sau khi có thể mở rộng dung lượng. Mở rộng năng lực là rất lớn. Vì vậy, nếu dung lượng được mở rộng, nó sẽ được làm lớn gấp đôi so với kích thước để tránh phải thay đổi dung lượng một lần nữa trừ khi kích thước tăng đột ngột (giá trị 1 được sử dụng trong trường hợp kích thước bằng 0). Việc mở rộng dung lượng được thực hiện theo quy trình dự trữ được hiển thị ở dòng 53 đến dòng 65. Nó bao gồm phân bổ một mảng mới ở dòng 58, di chuyển nội dung cũ ở dòng 59 và 60 và thu hồi mảng cũ ở dòng 64. Như được hiển thị ở dòng 55 và 56 quy trình dự trữ cũng có thể được sử dụng để thu nhỏ mảng bên dưới nhưng chỉ khi dung lượng mới được chỉ định ít nhất là lớn bằng kích thước. Nếu nó không phải là yêu cầu dự trữ được bỏ qua.

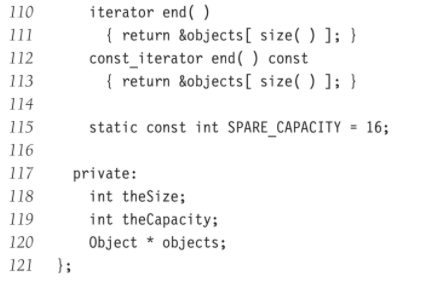
Hai phiên bản của toán tử là tầm và được hiển thị trong các dòng từ 67 đến 70. Việc kiểm tra lỗi có thể dễ dàng thêm vào bằng cách đảm bảo rằng chỉ mục đó trong phạm vi 0 đến size () - 1 bao gồm và ném một ngoại lệ nếu không.

Một loạt các quy trình ngắn cụ thể là popback pushback dung lượng kích thước trống và back được thực hiện ở dòng 72 đến dòng 101. Ở dòng 83 và 90, chúng ta thấy việc sử dụng toán tử postfix sử dụng theSize để lập chỉ mục mảng và sau đó tăng theSize. Chúng tôi đã thấy cùng một thành ngữ khi thảo luận về trình lặp itr sử dụng itr để quyết định xem mục nào và sau đó nâng cấp nó. Vị trí của các vấn đề Trong toán tử tiền tố itr tiến lên itr và sau đó sử dụng itr mới để quyết định mục nào sẽ xem và tương tự như vậy objectstheSize sẽ tăng Kích thước và sử dụng giá trị mới để lập chỉ mục mảng (không phải là thứ chúng ta muốn). popback và back đều có thể được hưởng lợi từ việc kiểm tra lỗi, trong đó một ngoại lệ được đưa ra nếu kích thước là 0.



Hình 3.9 Lớp vecto (Phần 1/3)





Hình 3.9 Lớp vecto (Phần 2/3)

Cuối cùng, ở dòng 103 đến dòng 113, chúng ta thấy khai báo kiểu lồng nhau của trình lặp và trình phối hợp và hai phương thức bắt đầu và hai phương thức kết thúc. Đoạn mã này sử dụng thực tế là trong C++ một biến con trỏ có tất cả các toán tử giống nhau mà chúng ta mong đợi cho một trình lặp. Các biến con trỏ có thể được sao chép và so sánh toán tử mang lại đối tượng được trỏ đến và đặc biệt nhất khi được áp dụng cho một biến con trỏ, biến con trỏ sau đó sẽ trỏ vào đối tượng sẽ được lưu trữ tuần tự tiếp theo Nếu con trỏ đang trỏ vào bên trong một mảng làm tăng con trỏ đặt nó ở phần tử mảng tiếp theo. Những ngữ nghĩa này cho con trỏ có từ đầu những năm 70 với ngôn ngữ lập trình C mà C++ dựa trên đó. Cơ chế trình lặp STL được thiết kế một phần để bắt chước các hoạt động của con trỏ.

Do đó, tại các dòng 103 và 104, chúng ta thấy các câu lệnh typedef cho biết trình lặp và trình biến đổi đơn giản là các tên khác cho một biến con trỏ và bắt đầu và kết thúc chỉ cần trả về các địa chỉ bộ nhớ đại diện cho vị trí mảng đầu tiên và vị trí mảng không hợp lệ đầu tiên tương ứng.

 Sự tương ứng giữa các trình vòng lặp và con trỏ cho kiểu vectơ có nghĩa là việc sử dụng một vectơ thay vì mảng C có thể mang lại ít chi phí. Điểm bất lợi là khi viết mã không có kiểm tra lỗi. Nếu trình lặp itr đi ngang qua điểm đánh dấu kết thúc thì nó cũng không nhất thiết báo hiệu lỗi. Để khắc phục sự cố này sẽ yêu cầu trình lặp và trình phối hợp phải là các kiểu lớp lồng nhau thực tế chứ không chỉ đơn giản là các biến con trỏ. Sử dụng các kiểu lớp lồng nhau phổ biến hơn nhiều và là những gì chúng ta sẽ thấy trong lớp Danh sách trong Phần 5.

* 1. Thực hiện danh sách

Trong phần này, chúng tôi cung cấp việc triển khai mẫu lớp danh sách có thể sử dụng được. Như trong trường hợp của lớp vectơ, lớp danh sách của chúng ta sẽ được đặt tên là Danh sách để tránh sự mơ hồ với lớp thư viện. Nhớ lại rằng lớp List sẽ được triển khai như một danh sách được liên kết kép và chúng ta sẽ cần duy trì các con trỏ đến cả hai đầu của danh sách. Làm như vậy cho phép chúng tôi duy trì chi phí thời gian không đổi cho mỗi hoạt động miễn là hoạt động xảy ra ở một vị trí đã biết. Vị trí đã biết có thể ở cuối hoặc ở vị trí được chỉ định bởi một trình lặp.

Khi xem xét thiết kế, chúng tôi sẽ cần cung cấp bốn lớp:

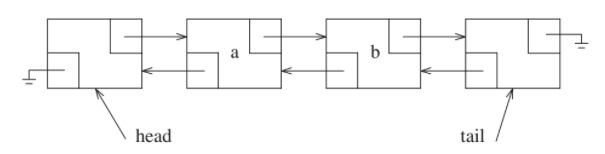
1. Bản thân lớp List chứa các liên kết đến cả kích thước của danh sách và một loạt các phương thức.

2. Lớp Node có khả năng là một lớp lồng nhau riêng tư. Một nút chứa dữ liệu và con trỏ đến các nút trước đó và tiếp theo cùng với các hàm tạo thích hợp.

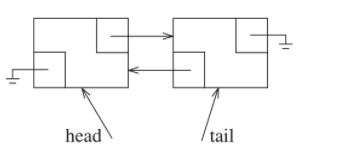
 3. Lớp constiterator trừu tượng hóa khái niệm về một vị trí và là một lớp được lồng vào nhau. Trình phối hợp lưu trữ một con trỏ đến nút hiện tại và cung cấp việc triển khai các hoạt động của trình vòng lặp cơ bản, tất cả ở dạng toán tử được nạp chồng như và.

 4. Lớp trình lặp tóm tắt khái niệm vị trí và là một lớp lồng nhau công khai. Trình lặp có chức năng tương tự như constiterator ngoại trừ toán tử đó trả về một tham chiếu đến mục đang được xem thay vì tham chiếu liên tục đến mục. Một vấn đề kỹ thuật quan trọng là một trình lặp có thể được sử dụng trong bất kỳ trường hợp nào yêu cầu trình phối hợp nhưng không phải ngược lại. Nói cách khác, trình lặp IS-A constiterator.

Chúng ta có thể thấy ở dòng 5 phần đầu khai báo lớp Node lồng nhau riêng tư. Thay vì sử dụng từ khóa class, chúng tôi sử dụng struct. Trong C, struct là một di tích từ ngôn ngữ lập trình C. Một cấu trúc trong C về cơ bản là một lớp trong đó các thành viên được mặc định là công khai. Nhớ lại rằng trong một lớp, các thành viên được mặc định là private. Rõ ràng là cấu trúc



Hình 3.10 Một danh sách được liên kết kép với các nút đầu trang và nút đuôi



Hình 3.11 Một danh sách được liên kết kép trống với các nút đầu trang và nút đuôi

 Tại dòng 9, chúng ta thấy phần đầu khai báo của lớp trình duyệt lồng nhau công khai và ở dòng 12, chúng ta thấy phần đầu của khai báo của lớp trình lặp lồng nhau công khai. Cú pháp bất thường là kế thừa là một cấu trúc mạnh mẽ không được sử dụng trong sách. Cú pháp kế thừa nói rằng trình lặp có chức năng chính xác như trình phối hợp với một số bổ sung có thể có và trình lặp đó tương thích về kiểu với trình phối hợp và có thể được sử dụng ở bất cứ nơi nào cần trình phối hợp. Hãy thảo luận kỹ những chi tiết đó khi chúng ta xem các triển khai thực tế sau này.

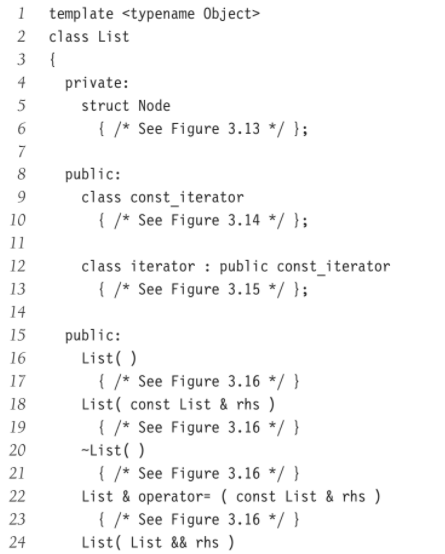
Các dòng từ 80 đến 82 chứa các thành viên dữ liệu cho Danh sách cụ thể là các con trỏ đến các nút đầu và nút đuôi. Chúng tôi cũng theo dõi kích thước trong thành viên dữ liệu để phương pháp kích thước có thể được thực hiện trong thời gian không đổi.

 Phần còn lại của lớp List bao gồm hàm tạo là Big-Five và một loạt các meth. Nhiều phương pháp là một lớp. bắt đầu và kết thúc trả về các trình vòng lặp thích hợp, lệnh gọi ở dòng 30 là điển hình của việc triển khai trong đó chúng ta trả về một trình vòng lặp đã xây dựng (do đó các lớp trình vòng lặp và trình phối hợp đều có một phương thức khởi tạo lấy một con trỏ đến một Node làm tham số của nó).

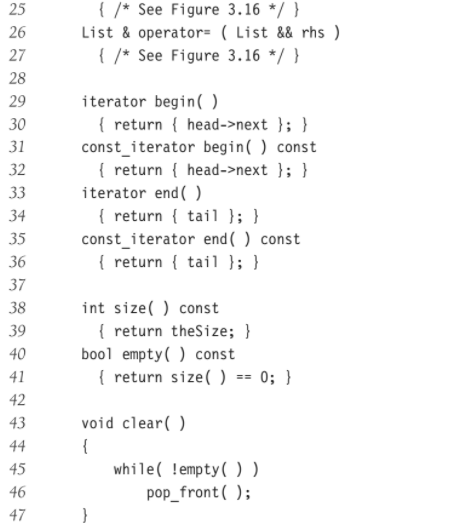
Phương thức xóa ở dòng 43 đến 47 hoạt động bằng cách xóa liên tục các mục cho đến khi Danh sách trống. Sử dụng chiến lược này cho phép clear tránh làm bẩn các nút đòi lại bởi vì việc khai hoang nút bây giờ được chuyển thành cửa sổ bật lên. Các phương thức từ dòng 48 đến dòng 67 đều hoạt động bằng cách khéo léo lấy và sử dụng một trình lặp thích hợp. Nhớ lại rằng phương thức chèn sẽ chèn trước một vị trí, vì vậy, chèn ngược sẽ chèn trước dấu cuối theo yêu cầu. Trong cửa sổ bật lại, lưu ý rằng xóa (-end ()) tạo một trình lặp tạm thời tương ứng với trình đánh dấu sẽ rút lại trình lặp tạm thời và sử dụng trình lặp đó để xóa. Hành vi tương tự xảy ra ở phía sau. Cũng lưu ý rằng trong trường hợp của các hoạt động cửa sổ bật lên và cửa sổ bật lại, một lần nữa chúng tôi tránh xử lý việc cải tạo nút.

Hình 3.12 cho thấy lớp Node bao gồm các mục được lưu trữ con trỏ tới Node trước đó và tiếp theo và một hàm tạo. Tất cả các thành viên dữ liệu là công khai.

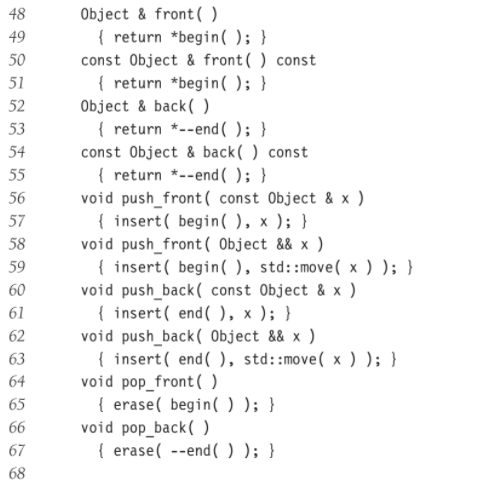
 Hình 3.12 cho thấy lớp constiterator và Hình 3.13 cho thấy lớp lặp. Như chúng ta đã đề cập trước đó, cú pháp ở dòng 39 trong Hình 3.13 chỉ ra một tính năng nâng cao được gọi là kế thừa và có nghĩa là trình phối hợp IS-A của trình lặp. Khi lớp trình lặp được viết theo cách này, nó kế thừa tất cả dữ liệu và phương thức từ trình phối hợp. Sau đó, nó có thể thêm dữ liệu mới, thêm các phương thức mới và ghi đè (tức là xác định lại) các phương thức hiện có. Trong trường hợp chung nhất, có một hành lý cú pháp quan trọng (thường dẫn đến từ khóa ảo xuất hiện trong mã). 1 đối tượng kiểu tên mẫu Khi lớp trình lặp được viết theo cách này, nó kế thừa tất cả dữ liệu và phương thức từ trình phối hợp. Sau đó, nó có thể thêm dữ liệu mới, thêm các phương thức mới và ghi đè (tức là xác định lại) các phương thức hiện có. Trong trường hợp chung nhất, có một hành lý cú pháp quan trọng (thường dẫn đến từ khóa ảo xuất hiện trong mã).

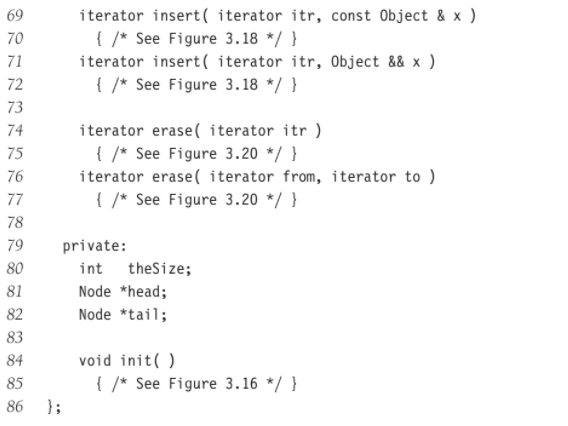


Hình 3.12 Liệt kê lớp (Phần 1/4)

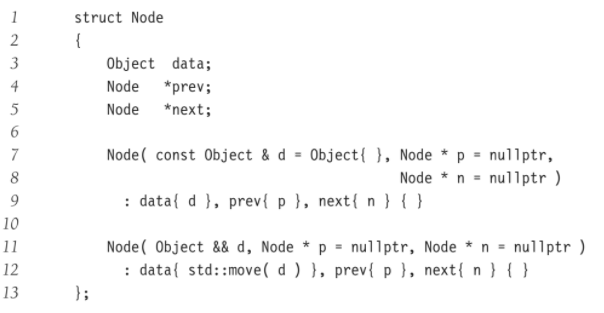


Hình 3.12 Liệt kê lớp (Phần 2/4)





Hình 3.13 Liệt lê lớp (Phần 3/4)



Hình 3.14 Lớp Node chồng nhau cho lớp Danh sách

* 1. Ngăn xếp ADT

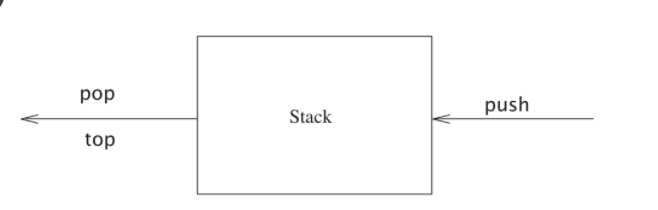
Ngăn xếp là một danh sách có hạn chế là việc chèn và xóa chỉ có thể được thực hiện ở một vị trí cụ thể là cuối danh sách được gọi là đầu.

* + 1. Mô hình ngăn xếp

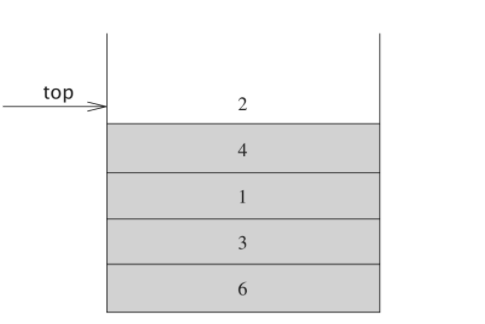
Các hoạt động cơ bản trên ngăn xếp là push tương đương với insert và pop sẽ xóa phần tử được chèn gần đây nhất. Phần tử được chèn gần đây nhất có thể được kiểm tra trước khi thực hiện bật lên bằng cách sử dụng quy trình hàng đầu. Một cửa sổ bật lên hoặc trên cùng trên một ngăn xếp trống thường được coi là một lỗi trong ADT ngăn xếp. Mặt khác, hết dung lượng khi thực hiện đẩy là một giới hạn thực hiện nhưng không phải là lỗi ADT. Các ngăn xếp đôi khi được gọi là danh sách LIFO (cuối cùng vào trước).

 Mô hình được mô tả trong Hình 3.15 chỉ biểu thị rằng push là hoạt động đầu vào và pops và top là đầu ra. Các hoạt động thông thường để tạo ra các ngăn xếp trống và kiểm tra độ trống là một phần của tiết mục nhưng về cơ bản tất cả những gì bạn có thể làm với ngăn xếp là đẩy và bật.

 Hình 3.16 cho thấy một ngăn xếp trừu tượng sau một số hoạt động.



Hình 3.15 Mô hình ngăn xếp Đầu vào cho ngăn xếp là bằng đầu ra đẩy là bằng cửa sổ bật lên và trên cùng.



Hình 3.16 Mô hình ngăn xếp Chỉ có thể truy cập phần tử trên cùng

* + 1. Thực hiện các ngăn xếp

Vì ngăn xếp là một danh sách nên bất kỳ việc triển khai danh sách nào cũng sẽ làm được. Liệt kê rõ ràng và hỗ trợ vector hoạt động ngăn xếp 99 thời gian chúng là sự lựa chọn hợp lý nhất. Đôi khi có thể nhanh hơn để thiết kế một triển khai cho mục đích đặc biệt. Bởi vì các hoạt động ngăn xếp là các hoạt động thời gian không đổi, điều này không có khả năng mang lại bất kỳ cải tiến rõ ràng nào ngoại trừ trong những trường hợp rất đặc biệt.

Đối với những thời điểm đặc biệt này, chúng tôi sẽ đưa ra hai cách triển khai ngăn xếp phổ biến. Một cái sử dụng cấu trúc được liên kết và cái kia sử dụng một mảng và cả hai đều đơn giản hóa logic trong vectơ và danh sách để chúng tôi không cung cấp mã.

***Việc triển khai danh sách được liên kết của ngăn xếp***

Việc triển khai đầu tiên của ngăn xếp sử dụng một danh sách được liên kết đơn lẻ. Chúng tôi thực hiện đẩy bằng cách chèn ở đầu danh sách. Chúng tôi thực hiện một cửa sổ bật lên bằng cách xóa phần tử ở đầu danh sách. Hoạt động hàng đầu chỉ đơn thuần kiểm tra phần tử ở đầu danh sách trả về giá trị của nó. Đôi khi các hoạt động pop và top được kết hợp thành một.

***Thực hiện theo mảng của ngăn xếp***

Một cách triển khai thay thế tránh các liên kết và có lẽ là giải pháp phổ biến hơn. Nó sử dụng sự triển khai backback và popback từ vector nên việc triển khai là không đáng kể. Được liên kết với mỗi ngăn xếp là theArray và topOfStack là 1 cho một ngăn xếp trống (đây là cách một ngăn xếp trống được khởi tạo). Để đẩy một số phần tử x vào ngăn xếp, chúng ta tăng topOfStack và sau đó đặt ArraytopOfStack x. Để bật lên, chúng tôi đặt giá trị trả về làArraytopOfStack và sau đó giảm dần topOfStack.

Lưu ý rằng các hoạt động này không chỉ được thực hiện trong thời gian cố định mà còn rất nhanh. Trên một số máy, các lệnh đẩy và cửa sổ bật lên (của số nguyên) có thể được viết trong một lệnh máy hoạt động trên một thanh ghi có địa chỉ tự động tăng và tự động giảm. Thực tế là hầu hết các máy hiện đại có các hoạt động ngăn xếp như một phần của tập lệnh thực thi ý tưởng rằng ngăn xếp có lẽ là cấu trúc dữ liệu cơ bản nhất trong khoa học máy tính sau mảng.

* + 1. Ứng dụng.

Sẽ không có gì ngạc nhiên nếu chúng ta hạn chế các hoạt động được phép trong một danh sách thì các hoạt động đó có thể được thực hiện rất nhanh chóng. Tuy nhiên, điều ngạc nhiên lớn là số lượng nhỏ các hoạt động còn lại lại rất mạnh mẽ và quan trọng. Chúng tôi đưa ra ba trong số nhiều ứng dụng của ngăn xếp. Ứng dụng thứ ba cung cấp một cái nhìn sâu sắc về cách các chương trình được tổ chức.

***Các ký hiệu cân bằng***

Trình biên dịch kiểm tra chương trình của bạn để tìm lỗi cú pháp nhưng thường xuyên thiếu một ký hiệu (chẳng hạn như dấu ngoặc nhọn hoặc dấu khởi đầu chú thích) có thể khiến trình biên dịch tràn ra hàng trăm dòng chẩn đoán mà không xác định được lỗi thực sự. Một công cụ hữu ích trong tình huống này là một chương trình kiểm tra xem mọi thứ có cân bằng hay không. Vì vậy, mọi dấu ngoặc vuông bên phải và dấu ngoặc đơn phải tương ứng với phần đối vị bên trái của nó.

Dãy () là hợp pháp nhưng () là sai. Rõ ràng là không đáng để viết một chương trình khổng lồ cho việc này nhưng hóa ra lại dễ dàng kiểm tra những thứ này. Để đơn giản, chúng tôi sẽ chỉ kiểm tra sự cân bằng của dấu ngoặc đơn và dấu ngoặc nhọn và bỏ qua bất kỳ ký tự nào khác xuất hiện.

Thuật toán đơn giản sử dụng một ngăn xếp và như sau :

Tạo một ngăn xếp rỗng. Đọc các ký tự cho đến cuối tệp. Nếu ký tự là một ký hiệu mở, hãy đẩy nó lên ngăn xếp. Nếu nó là một biểu tượng đóng và ngăn xếp trống thì báo lỗi. Nếu không, hãy bật ngăn xếp. Nếu biểu tượng bật ra không phải là biểu tượng mở tương ứng thì báo lỗi. Ở cuối tệp nếu ngăn xếp không trống thì báo lỗi.

 Bạn sẽ có thể thuyết phục bản thân rằng thuật toán này hoạt động. Nó rõ ràng là tuyến tính và thực sự chỉ làm cho một lần đi qua đầu vào. Do đó, nó trực tuyến và khá nhanh. Công việc bổ sung có thể được thực hiện để cố gắng quyết định phải làm gì khi một lỗi được báo cáo, chẳng hạn như xác định nguyên nhân có thể xảy ra.

***Biểu thức Postfix***

 Giả sử chúng ta có một máy tính bỏ túi và muốn tính chi phí của một chuyến đi mua sắm. Để làm như vậy, chúng tôi thêm một danh sách các số và nhân kết quả với 1,06, điều này sẽ tính giá mua của một số mặt hàng đã thêm thuế bán hàng địa phương. Nếu các mục là 4,99 5,99 và 6,99 thì một cách tự nhiên để nhập giá trị này sẽ là chuỗi 06 này tính giá mua của một số mặt hàng đã cộng thuế bán hàng nội địa.

* 1. 5.99 + 6.99 ∗ 1.06 =

Tùy thuộc vào máy tính, điều này tạo ra câu trả lời dự định 19.05 hoặc câu trả lời cụ thể 18.39. Hầu hết các máy tính bốn chức năng đơn giản sẽ đưa ra câu trả lời đầu tiên nhưng nhiều máy tính nâng cao biết rằng phép nhân có mức độ ưu tiên cao hơn phép cộng.

 Mặt khác, một số mặt hàng chịu thuế và một số mặt hàng không phải vậy nếu chỉ có mặt hàng đầu tiên và mặt hàng cuối cùng thực sự bị đánh thuế thì trình tự

4.99 ∗ 1.06 + 5.99 + 6.99 ∗ 1.06 =

sẽ đưa ra câu trả lời đúng (18,69) trên máy tính khoa học và câu trả lời sai (19,37) trên máy tính đơn giản. Một máy tính khoa học thường đi kèm với dấu ngoặc để chúng ta luôn có thể nhận được câu trả lời đúng bằng cách đặt dấu ngoặc nhưng với một máy tính đơn giản, chúng ta cần nhớ các kết quả trung gian.

 Một chuỗi đánh giá điển hình cho ví dụ này có thể là nhân 4,99 với 1,06 để lưu câu trả lời này là A1. Sau đó, chúng tôi thêm 5,99 và A1 lưu kết quả trong A1. Chúng tôi nhân 6,99 và 1,06 lưu câu trả lời trong A2 và kết thúc bằng cách thêm A1 và A2 để lại câu trả lời cuối cùng trong A1. Chúng ta có thể viết chuỗi hoạt động này như sau:

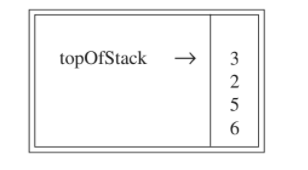
* 1. .06 ∗ 5.99 + 6.99 1.06 ∗ +

Ký hiệu này được gọi là hậu tố hoặc ký hiệu Ba Lan ngược và được đánh giá chính xác như chúng tôi đã mô tả ở trên. Cách dễ nhất để làm điều này là sử dụng ngăn xếp. Khi một số được nhìn thấy, nó được đẩy lên ngăn xếp khi một toán tử được nhìn thấy, toán tử được áp dụng cho hai số (ký hiệu) được bật ra từ ngăn xếp và kết quả được đẩy lên ngăn xếp. Ví dụ: biểu thức hậu tố

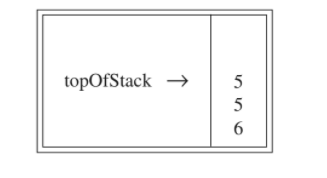
6523 + 8 ∗ +3 + ∗

được đánh giá như sau:

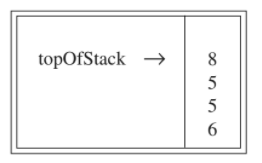
 Bốn biểu tượng đầu tiên được đặt trên ngăn xếp. Ngăn xếp kết quả là:



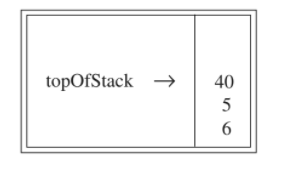
Tiếp theo a được đọc để 3 và 2 xuất hiện từ ngăn xếp và tổng 5 của chúng được đẩy.



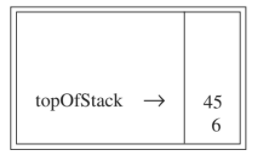
Tiếp theo 8 được đẩy.



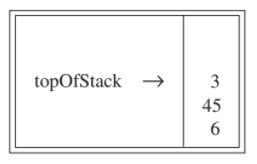
Bây giờ a\* được nhìn thấy nên 8 và 5 được xuất hiện và 5\*8= 40 được đẩy.



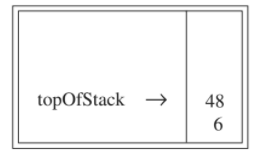
Tiếp theo a+ được nhìn thấy vì vậy 40 và 5 được xuất hiện và 5+40= 45 được đẩy.



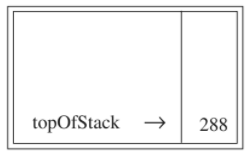
Bây giờ 3 được đẩy.



Tiếp theo, + bật 3 và 45 và đẩy 45+ 3= 48.



Cuối cùng a\* được nhìn thấy và 48 và 6 xuất hiện, kết quả là 6\* 48= 288 được đẩy.



***Chuyển đổi Infix sang Postfix***

 Không chỉ ngăn xếp có thể được sử dụng để đánh giá một biểu thức hậu tố mà chúng ta còn có thể sử dụng ngăn xếp để chuyển đổi một biểu thức ở dạng chuẩn (còn được gọi là infix) thành hậu tố. Chúng tôi sẽ tập trung vào một phiên bản nhỏ của vấn đề chung bằng cách chỉ cho phép các toán tử () và nhấn mạnh vào các quy tắc ưu tiên thông thường. Chúng tôi sẽ giả định thêm rằng biểu thức là hợp pháp. Giả sử chúng ta muốn chuyển đổi biểu thức infix

a+b\*c+(d\*e+f)\*g

vào postfix. Một câu trả lời đúng là abc \* +de \* f+g \* +.

Khi một toán hạng được đọc, nó ngay lập tức được đặt vào đầu ra. Các toán tử không được xuất ra ngay lập tức vì vậy chúng phải được lưu ở đâu đó. Điều chính xác cần làm là đặt các toán tử đã được nhìn thấy nhưng không được đặt trên đầu ra vào ngăn xếp. Chúng tôi cũng sẽ xếp chồng các dấu ngoặc đơn bên trái khi chúng gặp phải. Chúng tôi bắt đầu với một ngăn xếp trống ban đầu.

 Nếu chúng ta thấy một dấu ngoặc phải thì chúng ta bật các ký hiệu viết ngăn xếp cho đến khi chúng ta gặp một dấu ngoặc trái (tương ứng) được xuất hiện nhưng không xuất hiện.

Nếu chúng tôi thấy bất kỳ ký hiệu nào khác (() thì chúng tôi bật các mục nhập từ ngăn xếp cho đến khi chúng tôi tìm thấy mục nhập có mức độ ưu tiên thấp hơn. Một ngoại lệ là chúng tôi không bao giờ xóa a (khỏi ngăn xếp trừ khi xử lý a). Vì mục đích của phép toán này có mức ưu tiên thấp nhất và (cao nhất. Khi thực hiện xong, chúng ta đẩy toán tử vào ngăn xếp.

Cuối cùng, nếu chúng ta đọc đến cuối đầu vào, chúng ta bật ngăn xếp cho đến khi nó trống rỗng viết các ký hiệu vào đầu ra.

* 1. Hàng đợi ADT

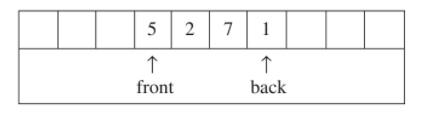
Giống như hàng đợi ngăn xếp là danh sách. Với một hàng đợi, tuy nhiên, việc chèn được thực hiện ở một đầu trong khi việc xóa được thực hiện ở đầu kia.

* + 1. Mô hình hàng đợi

Các hoạt động cơ bản trên hàng đợi là enqueue chèn một phần tử vào cuối danh sách (được gọi là phía sau) và dequeue xóa (và trả về) phần tử ở đầu danh sách (được gọi là phía trước). Hình 3.17 cho thấy mô hình trừu tượng của một hàng đợi.

* + 1. Triền khai hàng đợi theo mảng

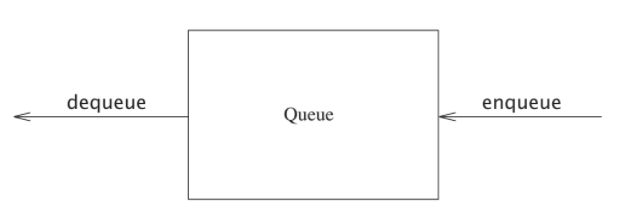
Như với ngăn xếp, bất kỳ triển khai danh sách nào cũng hợp pháp đối với hàng đợi. Giống như ngăn xếp, cả danh sách liên kết và triển khai mảng đều cung cấp thời gian chạy O (1) nhanh cho mọi hoạt động. Việc triển khai danh sách liên kết là đơn giản và được để lại như một bài tập. Bây giờ chúng ta sẽ thảo luận về việc triển khai mảng các hàng đợi. Đối với mỗi cấu trúc dữ liệu hàng đợi, chúng tôi giữ một mảng theArray và các vị trí phía trước và phía sau đại diện cho các phần cuối của hàng đợi. Chúng tôi cũng theo dõi số lượng phần tử thực sự có trong hàng đợi currentSize. Bảng sau đây cho thấy một hàng đợi ở một số trạng thái trung gian.



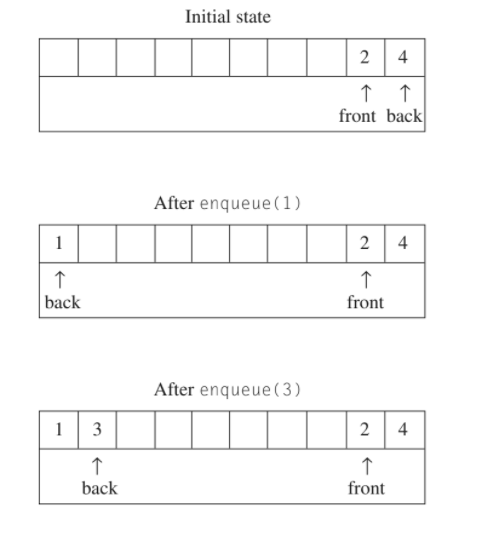
Các hoạt động phải rõ ràng. Để xếp hàng một phần tử x, chúng ta tăng currentSize và quay lại sau đó đặt Arrayback=x. Để dequeue một phần tử, chúng tôi đặt giá trị trả về cho currentSize Arrayfront giảm và sau đó tăng lên phía trước. Các chiến lược khác là có thể (điều này sẽ được thảo luận sau). Chúng tôi sẽ bình luận về việc kiểm tra các lỗi hiện tại.

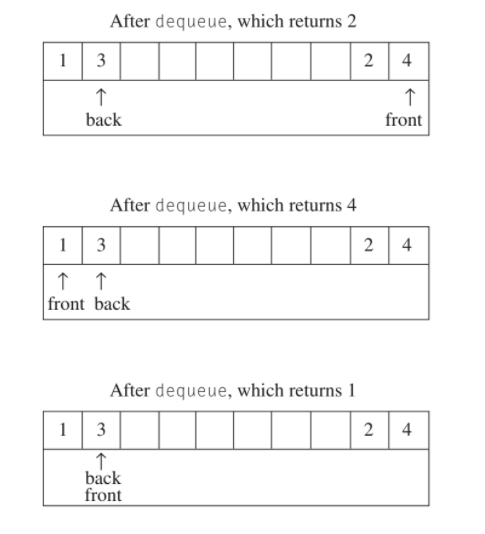
Có một vấn đề tiềm ẩn với việc triển khai này. Sau 10 hàng đợi, hàng đợi dường như đã đầy vì trở lại bây giờ là chỉ mục mảng cuối cùng và hàng đợi tiếp theo sẽ ở vị trí không tồn tại. Tuy nhiên, có thể chỉ có một vài phần tử trong hàng đợi vì một số phần tử có thể đã được xếp thứ tự. Hàng đợi như ngăn xếp thường nhỏ ngay cả khi có nhiều thao tác.

Giải pháp đơn giản là bất cứ khi nào phía trước hoặc phía sau đến cuối mảng, nó sẽ được bao quanh từ đầu. Các bảng sau đây hiển thị hàng đợi trong một số hoạt động. Đây được gọi là triển khai mảng tròn.



Hình 3.17 Mô hình hàng đợi





Mã bổ sung cần thiết để thực hiện bao bọc là tối thiểu (mặc dù nó có thể tăng gấp đôi thời gian chạy). Nếu việc tăng lên phía sau hoặc phía trước khiến nó đi qua mảng, giá trị sẽ được đặt lại về vị trí đầu tiên trong mảng.

Một số lập trình viên sử dụng các cách khác nhau để biểu diễn mặt trước và mặt sau của hàng đợi. Ví dụ, một số không sử dụng một mục nhập để theo dõi kích thước vì chúng dựa vào trường hợp cơ sở mà khi hàng đợi trống trở lại front-1. Kích thước được tính ngầm bằng cách so sánh mặt sau và mặt trước. Đây là một cách rất phức tạp vì có một số trường hợp đặc biệt, vì vậy hãy hết sức cẩn thận nếu bạn cần sửa đổi mã được viết theo cách này. Nếu currentSize không được duy trì như một thành viên dữ liệu rõ ràng thì hàng đợi sẽ đầy khi có phần tử theArray.capacity () - 1 vì chỉ theArray. Dung lượng () có thể phân biệt các kích thước khác nhau và một trong số này là 0. Chọn bất kỳ kiểu nào bạn thích và đảm bảo rằng tất cả các quy trình của bạn đều nhất quán. Vì có một số tùy chọn để triển khai, nên có thể đáng giá một hoặc hai nhận xét trong mã nếu bạn không sử dụng thành viên dữ liệu currentSize.

Trong các ứng dụng mà bạn chắc chắn rằng số lượng hàng đợi không lớn hơn dung lượng của hàng đợi, việc bao bọc là không cần thiết. Như với ngăn xếp, hàng đợi hiếm khi được thực hiện trừ khi các quy trình gọi chắc chắn rằng hàng đợi không trống. Do đó, kiểm tra lỗi thường bị bỏ qua cho hoạt động này ngoại trừ trong mã quan trọng. Điều này thường không chính đáng vì tiết kiệm thời gian mà bạn có thể đạt được là rất ít.

* + 1. Ứng dụng của hàng đợi

Có nhiều thuật toán sử dụng hàng đợi để cung cấp thời gian chạy hiệu quả. Bây giờ chúng ta sẽ đưa ra một số ví dụ đơn giản về cách sử dụng hàng đợi.

 Khi công việc được nộp cho một máy in, chúng được sắp xếp theo thứ tự đến. Vì vậy, về cơ bản các công việc được gửi đến máy in được đặt trên một hàng đợi.

 Hầu như mọi dòng trong cuộc sống thực đều (được cho là) là một hàng đợi. Ví dụ, các hàng tại quầy vé là hàng đợi vì dịch vụ đến trước được phục vụ trước.

 Một ví dụ khác liên quan đến mạng máy tính. Có nhiều thiết lập mạng của máy tính cá nhân trong đó đĩa được gắn vào một máy được gọi là máy chủ tệp. Người dùng trên các máy khác được cấp quyền truy cập vào tệp trên cơ sở ai đến trước được phục vụ trước nên cấu trúc dữ liệu là một hàng đợi.

Các ví dụ khác bao gồm các:

·       Cuộc gọi đến các công ty lớn sau đây thường được xếp vào hàng đợi khi tất cả các nhà khai thác đang bận.

·       Trong các trường đại học lớn, nơi mà nguồn lực hạn chế, sinh viên phải ký vào danh sách chờ nếu tất cả các máy tính đều bị chiếm dụng. Học sinh sử dụng máy tính lâu nhất sẽ bị buộc tắt máy trước tiên và học sinh chờ đợi lâu nhất là người dùng tiếp theo được phép bật.

 Toàn bộ một nhánh của toán học được gọi là lý thuyết xếp hàng đề cập đến việc tính toán theo xác suất thời gian mà người dùng mong đợi sẽ đợi trên một dòng bao lâu thì dòng đó và những câu hỏi khác như vậy. Câu trả lời phụ thuộc vào tần suất người dùng đến hàng và mất bao lâu để xử lý một người dùng sau khi người dùng được phục vụ. Cả hai tham số này được cho dưới dạng hàm phân phối xác suất. Trong những trường hợp đơn giản, một câu trả lời có thể được tính toán một cách phân tích. Ví dụ về một trường hợp đơn giản sẽ là một đường dây điện thoại có một nhà điều hành. Nếu nhà điều hành bận, người gọi sẽ được xếp vào hàng chờ (lên đến một số giới hạn tối đa). Vấn đề này quan trọng đối với các doanh nghiệp vì các nghiên cứu đã chỉ ra rằng mọi người rất nhanh chóng cúp máy.

Nếu có k toán tử thì vấn đề này khó giải hơn nhiều. Các vấn đề khó giải quyết về mặt phân tích thường được giải quyết bằng mô phỏng. Trong trường hợp của chúng tôi, chúng tôi sẽ cần sử dụng một hàng đợi để thực hiện mô phỏng. Nếu k lớn, chúng ta cũng cần các cấu trúc dữ liệu khác để thực hiện điều này một cách hiệu quả. Chúng ta sẽ xem cách thực hiện mô phỏng này trong Chương D. Sau đó, chúng ta có thể chạy mô phỏng cho một số giá trị của k và chọn k tối thiểu để có thời gian chờ hợp lý.

Các ứng dụng bổ sung cho hàng đợi rất nhiều và cũng như với các ngăn xếp, điều đáng kinh ngạc là một cấu trúc dữ liệu đơn giản lại có thể quan trọng đến vậy.

***Tóm tắt***

 Chương này mô tả khái niệm ADT và minh họa khái niệm này bằng ba trong số các kiểu dữ liệu trừu tượng phổ biến nhất. Mục tiêu chính là tách việc triển khai ADT ra khỏi chức năng của chúng. Chương trình phải biết các hoạt động làm gì nhưng thực sự tốt hơn là không biết nó được thực hiện như thế nào.

 Ngăn xếp, danh sách và hàng đợi có lẽ là ba cấu trúc dữ liệu cơ bản trong tất cả khoa học máy tính và việc sử dụng chúng được ghi lại thông qua một loạt các ví dụ. Đặc biệt, chúng ta đã thấy cách các ngăn xếp được sử dụng để theo dõi các lệnh gọi hàm và cách đệ quy thực sự được triển khai. Điều quan trọng cần hiểu là không chỉ vì nó làm cho các lan theo thủ tục trở nên khả thi mà bởi vì biết cách thực hiện đệ quy sẽ loại bỏ rất nhiều bí ẩn xung quanh việc sử dụng nó.

CHƯƠNG 3. HASHING (hàm băm)

Trong Chương 4, chúng ta đã thảo luận về kiểu dữ liệu trừu tượng của cây tìm kiếm , Cho phép các hoạt động khác nhau trên một tập hợp các phần tử. Trong chương này, chúng ta thảo luận về kiểu dữ liệu trừu tượng của bảng băm, kiểu dữ liệu này chỉ hỗ trợ một tập hợp con các phép toán được cây tìm kiếm nhị phân cho phép.

Việc thực hiện các bảng băm thường được gọi là băm Băm là một kỹ thuật được sử dụng để thực hiện chèn, xóa và tìm kiếm trong thời gian trung bình không đổi. Các hoạt động cây yêu cầu bất kỳ thông tin sắp xếp nào giữa các phần tử không được hỗ trợ hiệu quả. Do đó, các hoạt động như findMin, findMax và việc in toàn bộ bảng theo thứ tự đã sắp xếp theo thời gian tuyến tính không được hỗ trợ. Cấu trúc dữ liệu trung tâm trong chương này là bảng băm. Chúng ta sẽ ...

Xem một số phương pháp triển khai bảng băm.

So sánh các phương pháp này một cách phân tích.

Hiển thị nhiều ứng dụng của băm.

So sánh bảng băm với cây tìm kiếm nhị phân.

5.1 Ý Tưởng Chung

Cấu trúc dữ liệu bảng băm lý tưởng chỉ là một mảng có kích thước cố định chứa các mục. Như đã thảo luận trong chương 4, thông thường tìm kiếm được thực hiện trên một số phần (tức là thành viên dữ liệu) của mục. Đây được gọi là chìa khóa. Ví dụ: một mục có thể bao gồm một chuỗi (đóng vai trò là chìa khóa) và các thành viên dữ liệu bổ sung (ví dụ: tên là một phần của cấu trúc nhân viên lớn). Chúng ta sẽ gọi kích thước bảng là TableSize, với sự hiểu biết rằng đây là một phần của cấu trúc dữ liệu băm chứ không chỉ là một số biến trôi nổi trên toàn cầu. Quy ước chung là để bảng chạy từ 0 đến TableSize - 1; chúng ta sẽ thấy lý do ngay sau đây.

Mỗi khóa được ánh xạ thành một số trong phạm vi từ 0 đến TableSize - 1 và được đặt trong ô thích hợp. Ánh xạ được gọi là hàm băm, lý tưởng là hàm này phải đơn giản để tính toán và phải đảm bảo rằng hai khóa riêng biệt bất kỳ nhận được các ô khác nhau. Vì có một số lượng ô hữu hạn và nguồn cung cấp khóa hầu như vô tận, điều này rõ ràng là không thể, và do đó chúng tôi tìm kiếm một hàm băm để phân phối khóa đồng đều giữa các ô. Hình 5.1 là điển hình của một tình huống hoàn hảo. Trong ví dụ này, john băm thành 3, phil băm thành 4, băm dave thành 6, và băm mary thành 7.

|  |
| --- |
|  |
|  |
|  |
| john 25000 |
| phil 31250 |
|  |
| dave 27500 |
| mary 28200 |
|  |
|  |

0

1

2

3

4

5

6

7

8

9

Hình 5.1 Một bảng băm lý tưởng

Đây là ý tưởng cơ bản của băm. Các vấn đề còn lại duy nhất liên quan đến việc chọn một chức năng, quyết định phải làm gì khi hai khóa băm thành cùng một giá trị (đây được gọi là xung đột) và quyết định kích thước bảng.

5.2 Hàm băm

Nếu các khóa đầu vào là số nguyên, thì chỉ cần trả lại khóa mod TableSize nói chung là một chiến lược hợp lý, trừ khi khóa xảy ra một số thuộc tính không mong muốn. Trong trường hợp này, việc lựa chọn hàm băm cần phải được xem xét cẩn thận. Ví dụ: nếu kích thước bảng là 10 và tất cả các khóa đều kết thúc bằng 0, thì hàm băm tiêu chuẩn là một lựa chọn tồi. Vì những lý do mà chúng ta sẽ xem ở phần sau, và để tránh những trường hợp như trên, bạn nên đảm bảo rằng kích thước bảng là nguyên tố. Khi các phím đầu vào là số nguyên ngẫu nhiên, thì chức năng này không chỉ rất đơn giản để tính toán mà còn phân phối các phím một cách đồng đều.

Một hàm băm được thể hiện trong Hình 5.3. Hàm băm này giả định rằng Khóa có ít nhất ba ký tự. Giá trị 27 đại diện cho số chữ cái trong bảng chữ cái tiếng Anh, cộng với khoảng trống và 729 là 272. Hàm này chỉ kiểm tra ba ký tự đầu tiên, nhưng nếu chúng là ngẫu nhiên và kích thước bảng là 10.007, thì chúng tôi mong đợi phân phối công bằng hợp lý. Thật không may, tiếng Anh không phải là ngẫu nhiên. Mặc dù có 263 = 17.576 sự kết hợp có thể có của ba ký tự (bỏ qua khoảng trống), việc kiểm tra một từ điển trực tuyến khá lớn cho thấy số lượng các kết hợp khác nhau thực sự chỉ là 2.851. Ngay cả khi không có sự kết hợp nào trong số này xảy ra va chạm, chỉ 28 phần trăm của bảng có thể được băm thành. Do đó, hàm này, mặc dù có thể dễ dàng tính toán, nhưng cũng không thích hợp nếu bảng băm quá lớn.

1 int hash( const string & key, int tableSize )

2 {

3 int hashVal = 0;

4

5 for( char ch : key )

6 hashVal += ch;

7

8 return hashVal % tableSize;

9 }

Hình 5.2 Một hàm băm đơn giản

1 int hash( const string & key, int tableSize )

2 {

3 return ( key[ 0 ] + 27 \* key[ 1 ] + 729 \* key[ 2 ] ) % tableSize;

4 }

Hình 5.3 Một hàm băm có thể có khác - không quá tốt

1 /\*\*

2 \* A hash routine for string objects.

3 \*/

4 unsigned int hash( const string & key, int tableSize )

5 {

6 unsigned int hashVal = 0;

7

8 for( char ch : key )

9 hashVal = 37 \* hashVal + ch;

10

11 return hashVal % tableSize;

12 }

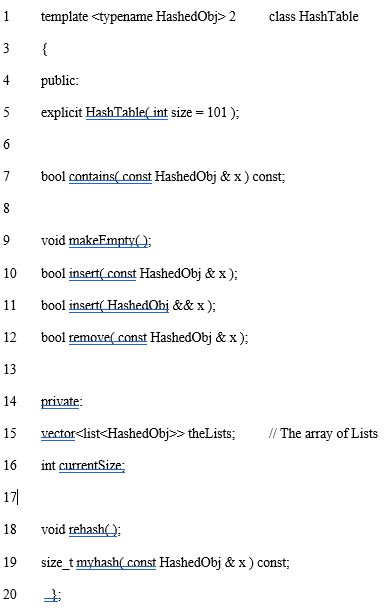
Hình 5.4 Một hàm băm tốt.

Hình 5.4 Cho thấy nỗ lực thứ ba đối với một hàm băm. Hàm băm này liên quan đến tất cả các ký tự trong khóa và thường có thể được mong đợi để phân phối tốt (nó tính KeySize − 1 Key [KeySize - i - 1] 37i và đưa kết quả vào phạm vi thích hợp). Mã tính toán một hàm đa thức (trong số 37) bằng cách sử dụng quy tắc Horner. Ví dụ, một cách khác để tính hk = k0 + 37k1 + 372k2 là theo công thức hk = ((k2) ∗ 37 + k1) ∗ 37 + k0. Quy tắc của Horner mở rộng điều này thành đa thức bậc n.

Hàm băm tận dụng thực tế là cho phép over ow và sử dụng unsigned int để tránh đưa vào một số âm.

5.3 Chuỗi riêng biệt

Chiến lược đầu tiên, thường được gọi là chuỗi riêng biệt, là giữ một danh sách tất cả các phần tử băm với cùng một giá trị. Chúng ta có thể sử dụng triển khai danh sách Thư viện chuẩn. Nếu không gian chật hẹp, bạn nên tránh sử dụng chúng (vì các danh sách này được liên kết kép và lãng phí không gian). Chúng tôi giả định cho phần này rằng các khóa là 10 ô vuông hoàn hảo đầu tiên và hàm băm chỉ đơn giản là băm (x) = x mod 10. (Kích thước bảng không phải là số nguyên tố nhưng được sử dụng ở đây để đơn giản hóa



Hình 5.6 Khai báo kiểu cho bảng băm chuỗi chuỗi riêng biệt

Giao diện lớp cho việc triển khai chuỗi riêng biệt được thể hiện trong Hình 5.6. Bảng băm lưu trữ một mảng danh sách được liên kết, được cấp phát trong hàm tạo.

Giao diện lớp minh họa một điểm cú pháp: Trước C ++ 11, trong phần khai báo theLists, cần có một khoảng trắng giữa hai> s; vì >> là mã thông báo C ++ và vì nó dài hơn> nên >> sẽ được công nhận là mã thông báo. Trong C ++ 11, điều này không còn xảy ra nữa. Cũng giống như cây tìm kiếm nhị phân chỉ hoạt động cho các đối tượng có thể so sánh được, các bảng băm trong chương này chỉ hoạt động đối với các đối tượng cung cấp hàm băm và các toán tử bình đẳng (toán tử == hoặc toán tử! =, hoặc có thể cả hai).

Thay vì yêu cầu các hàm băm lấy cả đối tượng và kích thước bảng làm tham số, chúng ta có các hàm băm chỉ lấy đối tượng làm tham số và trả về kiểu tích phân thích hợp. Cơ chế tiêu chuẩn để thực hiện việc này sử dụng các đối tượng hàm và giao thức cho bảng băm đã được giới thiệu trong C ++ 11. Cụ thể, trong C ++ 11, các hàm băm có thể được thể hiện bằng mẫu đối tượng hàm:

template <typename Key>

class hash

{

public:

size\_t operator() ( const Key & k ) const; }

Các triển khai mặc định của mẫu này được cung cấp cho các kiểu tiêu chuẩn như int và string; do đó, hàm băm được mô tả trong Hình 5.4 có thể được thực hiện như

template <>

class hash<string> {

public:

size\_t operator()( const string & key ) {

size\_t hashVal = 0;

for( char ch : key )

hashVal = 37 \* hashVal + ch;

return hashVal; }

};

Kiểu size\_t là kiểu tích phân không dấu thể hiện kích thước của một đối tượng; do đó, nó được đảm bảo có thể lưu trữ một chỉ mục mảng. Sau đó, một lớp thực hiện thuật toán bảng băm có thể sử dụng các lệnh gọi đến đối tượng hàm băm chung để tạo ra kiểu tích phân size\_t và sau đó chia tỷ lệ kết quả thành một chỉ số mảng phù hợp. Trong bảng băm của chúng tôi, điều này được thể hiện trong hàm thành viên riêng myhash, được hiển thị trong Hình 5.7.

Hình 5.8 minh họa một lớp Employee có thể được lưu trữ trong bảng băm chung, sử dụng thành viên tên làm khóa. Lớp Employee thực hiện các yêu cầu HashedObj bằng cách cung cấp các toán tử bình đẳng và một đối tượng hàm băm.

Mã để thực hiện makeEmpty, chứa và loại bỏ được hiển thị trong Hình 5.9.

Tiếp theo là quy trình chèn. Phần tử có thể được đặt ở bất kỳ đâu trong danh sách; sử dụng push\_back là thuận tiện nhất trong trường hợp của chúng tôi .whichList là một biến tham chiếu; xem Phần 1.5.2 để thảo luận về việc sử dụng các biến tham chiếu này.

Chúng tôi coi hệ số tải, λ, của bảng băm là tỷ lệ giữa số phần tử trong bảng băm với kích thước bảng.. Trong một tìm kiếm không thành công, số lượng nút cần kiểm tra trung bình là λ. Một tìm kiếm thành công yêu cầu khoảng 1 + (λ / 2) liên kết được duyệt qua. Số "nút khác" dự kiến ​​trong bảng N phần tử và M danh sách là (N − 1) / M = λ − 1 / M, về cơ bản là λ, vì M được cho là lớn. Trung bình, một nửa số “nút khác” được tìm kiếm, vì vậy kết hợp với nút phù hợp, chúng tôi thu được chi phí tìm kiếm trung bình là 1 + λ / 2 nút. Phân tích này cho thấy kích thước bảng không thực sự quan trọng mà là yếu tố tải trọng. Quy tắc chung cho phép băm chuỗi riêng biệt là làm cho kích thước bảng lớn bằng số phần tử dự kiến ​​(nói cách khác, giả sử λ ≈ 1).

1 size\_t myhash( const HashedObj & x ) const

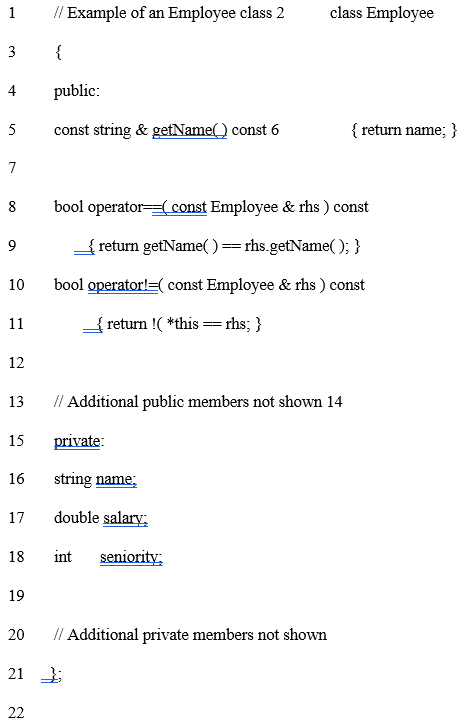
2 {

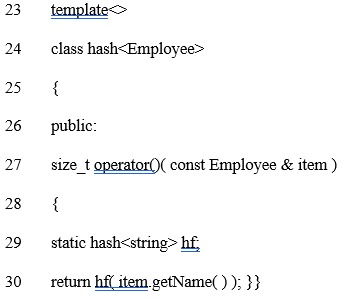
3 static hash<HashedObj> hf;

4 return hf( x ) % theLists.size( );

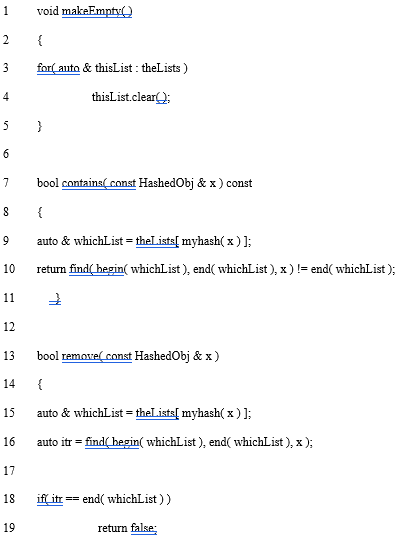
5 }

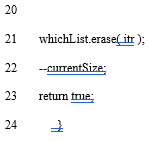
Hình 5.7 Hàm thành viên myHash cho bảng băm



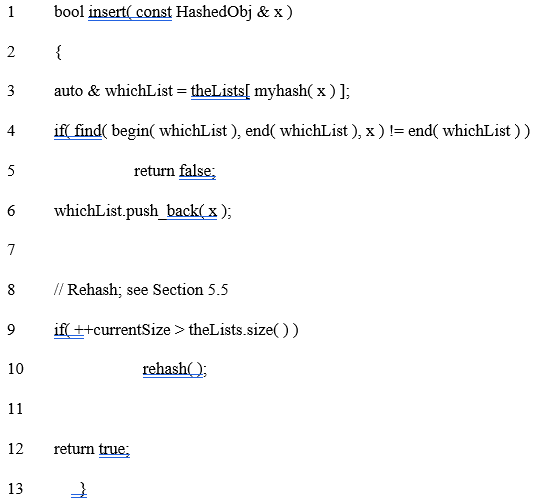


Hình 5.8 Ví dụ về một lớp có thể được sử dụng làm đối tượng băm





Hình 5.9 makeEmpty, chứa và loại bỏ các quy trình cho bảng băm chuỗi riêng biệt



Hình 5.10 quy trình chèn cho bảng băm chuỗi riêng biệt

5.4 Bảng băm không có danh sách được liên kết

Băm chuỗi riêng biệt có nhược điểm là sử dụng danh sách liên kết. Điều này có thể làm chậm thuật toán một chút do thời gian cần thiết để phân bổ các ô mới (đặc biệt là trong các ngôn ngữ khác) và về cơ bản yêu cầu triển khai một cấu trúc dữ liệu thứ hai. Một giải pháp thay thế để giải quyết xung đột với danh sách được liên kết là thử các ô thay thế cho đến khi tìm thấy ô trống. Chính thức hơn, các ô h0 (x), h1 (x), h2 (x), ... được thử liên tiếp, trong đó hi (x) = (hash (x) + f (i)) mod TableSize, với f ( 0) = 0. Hàm, f, là chiến lược phân giải col-lision.

5.4.1 Đo tuyến tính

Trong thăm dò tuyến tính, f là một hàm tuyến tính của i, điển hình là f (i) = i. Điều này tương đương với việc thử các ô một cách tuần tự (có bao bọc) để tìm kiếm một ô trống. Hình 5.11 cho thấy kết quả của việc chèn các khóa {89, 18, 49, 58, 69} vào một bảng băm bằng cách sử dụng cùng một hàm băm như trước và chiến lược giải quyết xung đột, f (i) = i.

Mặc dù chúng tôi sẽ không thực hiện các phép tính ở đây, nhưng có thể chỉ ra rằng số lượng đầu dò dự kiến ​​sử dụng thăm dò tuyến tính là khoảng 1 (1 + 1 / (1 - λ) 2) cho các lần chèn và tìm kiếm không thành công và 1 (1 + 1 / (1 - λ)) để tìm kiếm thành công. Các tính toán có liên quan phần nào. Từ mã có thể dễ dàng nhận thấy rằng các lần chèn và tìm kiếm không thành công yêu cầu cùng một số lượng đầu dò. Suy nghĩ của một thời điểm cho thấy rằng, trung bình, các tìm kiếm thành công sẽ mất ít thời gian hơn các tìm kiếm không thành công.

Lưu ý là λ thay đổi từ 0 đến giá trị hiện tại của nó, do đó các lần chèn trước đó rẻ hơn và sẽ làm giảm giá trị trung bình. Ví dụ, trong bảng trong Hình 5.11, λ = 0,5, nhưng chi phí truy cập 18 được xác định khi 18 được chèn vào. Tại thời điểm đó, λ = 0,2. Vì 18 đã được chèn vào một bảng tương đối trống, nên việc truy cập nó sẽ dễ dàng hơn so với việc truy cập vào một phần tử được chèn gần đây, chẳng hạn như 69

I (λ) = λ 0 1 - xdx = λ ln 1 - λ

Hình 5.12 so sánh hiệu suất của thăm dò tuyến tính (đường cong đứt gãy) với những gì mong đợi từ độ phân giải va chạm ngẫu nhiên hơn. Các tìm kiếm thành công được ký hiệu bằng chữ S, và các tìm kiếm và chèn không thành công được đánh dấu lần lượt bằng U và I.

Nếu λ = 0,75, thì công thức trên chỉ ra rằng 8,5 đầu dò được mong đợi để chèn trong đầu dò tuyến tính. Nếu λ = 0,9, thì có 50 đầu dò được mong đợi, điều này là không hợp lý. Điều này so sánh với 4 và 10 đầu dò cho các hệ số tải tương ứng nếu phân cụm không phải là vấn đề. Tuy nhiên, nếu λ = 0,5, trung bình chỉ cần 2,5 đầu dò để chèn và trung bình chỉ cần 1,5 đầu dò để tìm kiếm thành công.

EmptyTable After89 After18 After49 After58 After69

0 49 49 49

1 58 58

2 69

3

4

5

6

7

8 18 18 18 18

9 89 89 89 89 89

Hình 5.11 Bảng băm với thăm dò tuyến tính, sau mỗi lần chèn

5.4.2 Đo lường bậc hai

Thăm dò bậc hai là một phương pháp giải quyết va chạm giúp loại bỏ vấn đề phân cụm chính của thăm dò tuyến tính. Thăm dò bậc hai là những gì bạn mong đợi — hàm va chạm là bậc hai. Sự lựa chọn phổ biến là f (i) = i2. Hình 5.13 cho thấy bảng băm kết quả với hàm xung đột này trên cùng một đầu vào được sử dụng trong ví dụ thăm dò tuyến tính.

Đối với thăm dò tuyến tính, việc để bảng băm gần đầy là một ý tưởng tồi, vì mỗi dạng suy giảm. Đối với thăm dò bậc hai, tình hình thậm chí còn nghiêm trọng hơn: Không có gì đảm bảo sẽ chốt một ô trống khi bảng đầy hơn một nửa, hoặc thậm chí trước khi bảng đầy một nửa nếu kích thước bảng không phải là số nguyên tố. Điều này là do nhiều nhất một nửa chiếc bàn có thể được sử dụng làm vị trí thay thế để giải quyết các vụ va chạm.

U,I U,I

15.0

12.0 S

9.0

6.0

3.0 S

0.0

.10 .15 .20 .25 .30 .35 .40 .45 .50 .55 .60 .65 .70 .75 .80 .85 .90 .95

Hình 5.12 Số lượng đầu dò được vẽ biểu đồ dựa trên hệ số tải cho chiến lược thăm dò tuyến tính (gạch ngang) và ngẫu nhiên (S là tìm kiếm thành công, U là tìm kiếm không thành công và I là chèn)

EmptyTable After89 After18 After49 After58 After69

0 49 49 49

1

2 58 58

3 69

4

5

6

7

8 18 18 18 18

9 89 89 89 89 89

Hình 5.13 Bảng băm với thăm dò bậc hai, sau mỗi lần chèn

Định lý 5.1

Nếu sử dụng thăm dò bậc hai và kích thước bảng là số nguyên tố, thì một phần tử mới luôn có thể được chèn vào nếu bảng trống ít nhất một nửa.

Bằng chứng

Để kích thước bảng, TableSize, là một số nguyên tố (lẻ) lớn hơn 3. Chúng tôi chỉ ra rằng các vị trí thay thế TableSize / 2 đầu tiên (bao gồm cả vị trí ban đầu h (x)) đều khác biệt. Hai trong số các vị trí này là h (x) + i2 (mod TableSize) và h (x) + j2 (mod TableSize), trong đó 0 ≤ i, j ≤ TableSize / 2. Giả sử, vì mâu thuẫn, các vị trí này giống nhau, nhưng i = j. Sau đó

0

h(x) + i2 = h(x) + j2 (mod TableSize)

i2 = j2 (mod TableSize)

i2 − j2 = 0 (mod TableSize)

(i − j)(i + j) = 0 (mod TableSize)

Vì TableSize là số nguyên tố, theo đó, một trong hai (i - j) hoặc (i + j) bằng 0 (mod TableSize). Vì i và j là khác nhau, lựa chọn đầu tiên là không thể. Vì 0 ≤ i, j ≤ TableSize / 2, tùy chọn thứ hai cũng là không thể. Do đó, các vị trí thay thế TableSize / 2 đầu tiên là khác biệt. Nếu có nhiều nhất TableSize / 2 vị trí thì luôn có thể tìm thấy một vị trí trống.

Không thể thực hiện xóa tiêu chuẩn trong bảng băm thăm dò, vì ô có thể đã gây ra xung đột đi qua nó. Ví dụ, nếu chúng ta loại bỏ 89, thì hầu như tất cả các thao tác tìm còn lại sẽ không thành công. Do đó, bảng băm thăm dò yêu cầu xóa lười biếng, mặc dù trong trường hợp này thực sự không có hàm ý lười biếng.

Giao diện lớp cần thiết để triển khai bảng băm thăm dò được thể hiện trong Hình 5.14. Thay vì một mảng danh sách, chúng ta có một mảng các ô mục nhập bảng băm. Lớp lồng nhau HashEntry lưu trữ trạng thái của một mục nhập trong thành viên thông tin; trạng thái này là ACTIVE, EMPTY hoặc DELETED.

Chúng ta sử dụng một kiểu liệt kê tiêu chuẩn.

enum EntryType { ACTIVE, EMPTY, DELETED };

Xây dựng bảng (Hình 5.15) bao gồm thiết lập thành viên thông tin thành EMPTY cho mỗi ô. Hàm chứa (x), được hiển thị trong Hình 5.16, gọi các hàm thành viên riêng isActive và findPos. Hàm thành viên riêng findPos thực hiện giải quyết xung đột. Trong quy trình chèn, chúng tôi đảm bảo rằng bảng băm lớn hơn ít nhất gấp đôi số phần tử trong bảng, vì vậy độ phân giải bậc hai sẽ luôn hoạt động. Đang thực hiện

Nếu kích thước bảng là một số nguyên tố có dạng 4k + 3 và chiến lược phân giải xung đột bậc hai f (i) = ± i2 được sử dụng, thì toàn bộ bảng có thể được thăm dò. Chi phí là một quy trình phức tạp hơn một chút.

1 template <typename HashedObj>

2 class HashTable

3 {

4 public:

5 explicit HashTable( int size = 101 );

6

7 bool contains( const HashedObj & x ) const;

8

9 void makeEmpty( );

10 bool insert( const HashedObj & x );

11 bool insert( HashedObj && x );

12 bool remove( const HashedObj & x );

13

14 enum EntryType { ACTIVE, EMPTY, DELETED };

15

16 private:

17 struct HashEntry

18 {

19 HashedObj element;

20 EntryType info;

21

22 HashEntry( const HashedObj & e = HashedObj{ }, EntryType i = EMPTY )

23 : element{ e }, info{ i } { }

24 HashEntry( HashedObj && e, EntryType i = EMPTY )

25 : element{ std::move( e ) }, info{ i } { }

26 };

27

28 vector<HashEntry> array;

29 int currentSize;

30

31 bool isActive( int currentPos ) const;

32 int findPos( const HashedObj & x ) const;

33 void rehash( );

34 size\_t myhash( const HashedObj & x ) const;

35 };

Hình 5.14 Giao diện lớp cho bảng băm sử dụng chiến lược thăm dò, bao gồm lớp HashEntry lồng nhau

Trong Hình 5.16, các phần tử được đánh dấu là đã xóa được tính là có trong bảng. Điều này có thể gây ra vấn đề, vì bàn có thể bị đầy quá sớm. Chúng ta sẽ thảo luận về mục này ngay bây giờ.

Các dòng từ 12 đến 15 thể hiện cách nhanh chóng để thực hiện giải bậc hai. Theo định nghĩa của hàm phân giải bậc hai, f (i) = f (i - 1) + 2i - 1, do đó ô tiếp theo cần thử cách ô trước đó một khoảng cách và khoảng cách này tăng lên 2 trên các đầu dò liên tiếp.

1 explicit HashTable( int size = 101 ) : array( nextPrime( size ) )

2 { makeEmpty( ); }

3

4 void makeEmpty( )

5 {

6 currentSize = 0;

7 for( auto & entry : array )

8 entry.info = EMPTY;

9 }

Hình 5.15 Các quy trình khởi tạo bảng băm thăm dò bậc hai

1 bool contains( const HashedObj & x ) const

2 { return isActive( findPos( x ) ); }

3

4 int findPos( const HashedObj & x ) const 5 {

6 int offset = 1;

7 int currentPos = myhash( x ); 8

9 while( array[ currentPos ].info != EMPTY &&

10 array[ currentPos ].element != x )

11 {

12 currentPos += offset; // Compute ith probe

13 offset += 2;

14 if( currentPos >= array.size( ) )

15 currentPos -= array.size( );

16 }

17

18 return currentPos;

19 }

20

21 bool isActive( int currentPos ) const

22 { return array[ currentPos ].info == ACTIVE; }

Hình 5.16 chứa các trình trợ giúp thường quy (và các trình trợ giúp riêng) để băm với thăm dò bậc hai

Nếu vị trí mới nằm ngoài mảng, nó có thể được đưa trở lại trong phạm vi bằng cách trừ đi Kích thước bảng. Phương pháp này nhanh hơn phương pháp hiển nhiên, bởi vì nó tránh được phép nhân và phép chia dường như là bắt buộc. Một cảnh báo quan trọng: Thứ tự kiểm tra ở dòng 9 và 10 là quan trọng. Đừng chuyển nó!

Thói quen cuối cùng là chèn. Như với băm chuỗi riêng biệt, chúng tôi không làm gì nếu x đã có mặt. Nó là một modi đơn giản để làm một cái gì đó khác. Nếu không, chúng tôi đặt nó tại vị trí được đề xuất bởi quy trình findPos. Mã được hiển thị trong Hình 5.17. Nếu hệ số tải vượt quá 0,5, bảng đã đầy và chúng tôi phóng to bảng băm. Điều này được gọi là băm lại và được thảo luận trong Phần 5.5. Hình 5.17 cũng cho thấy loại bỏ.

1 bool insert( const HashedObj & x )

2 {

3 // Insert x as active

4 int currentPos = findPos( x );

5 if( isActive( currentPos ) )

6 return false;

7

8 array[ currentPos ].element = x;

9 array[ currentPos ].info = ACTIVE;

10

11 // Rehash; see Section 5.5

12 if( ++currentSize > array.size( ) / 2 )

13 rehash( );

14

15 return true; 16 }

17

18 bool remove( const HashedObj & x )

19 {

20 int currentPos = findPos( x );

21 if( !isActive( currentPos ) )

22 return false;

23

24 array[ currentPos ].info = DELETED;

25 return true;

26 }

Hình 5.17 Một số quy trình chèn và loại bỏ cho bảng băm với thăm dò bậc hai

5.4.3 Băm kép

Phương pháp giải quyết xung đột cuối cùng mà chúng ta sẽ kiểm tra là băm kép. Đối với hàm băm kép, một lựa chọn phổ biến là f (i) = i · hash2 (x). Công thức này nói rằng chúng tôi áp dụng một hàm băm thứ hai cho x và thăm dò ở khoảng cách hash2 (x), 2hash2 (x), ..., v.v. Một sự lựa chọn không tốt của hash2 (x) sẽ là một thảm họa. Ví dụ: lựa chọn rõ ràng hash2 (x) = x mod 9 sẽ không hữu ích nếu 99 được chèn vào đầu vào trong các ví dụ trước. Do đó, hàm không bao giờ được đánh giá bằng 0. Điều quan trọng nữa là đảm bảo tất cả các ô đều có thể được thăm dò (điều này không thể thực hiện được trong ví dụ dưới đây, vì kích thước bảng không phải là số nguyên tố). Một chức năng như vậy

như hash2 (x) = R - (x mod R), với R là số nguyên tố nhỏ hơn TableSize, sẽ hoạt động tốt. Nếu ta chọn R = 7, thì Hình 5.18 cho thấy kết quả của việc chèn các phím giống như trước đây.

Lần va chạm đầu tiên xảy ra khi 49 được chèn vào. hash2 (49) = 7−0 = 7, do đó 49 được chèn vào vị trí 6. hash2 (58) = 7 - 2 = 5, do đó 58 được chèn vào vị trí 3. Cuối cùng, 69 va chạm và được chèn vào khoảng cách hash2 ( 69) = 7−6 = 1 đi. Nếu chúng tôi cố gắng chèn 60 vào vị trí 0, chúng tôi sẽ xảy ra va chạm. Vì hash2 (60) = 7 - 4 = 3, sau đó chúng tôi sẽ thử các vị trí 3, 6, 9 và sau đó là 2 cho đến khi tìm thấy một vị trí trống. Nói chung là có thể xảy ra một số trường hợp xấu, nhưng không có quá nhiều ở đây.

EmptyTable After89 After18 After49 After58 After69

0 49 49 49

1

2 58 58

3 69

4

5

6

7

8 18 18 18 18

9 89 89 89 89 89

Hình 5.18 Bảng băm với phép băm kép, sau mỗi lần chèn

5.5 Làm lại

Nếu bảng quá đầy, thời gian chạy các thao tác sẽ bắt đầu mất quá nhiều thời gian và các thao tác chèn có thể không thành công đối với băm địa chỉ mở với độ phân giải bậc hai. Điều này có thể xảy ra nếu có quá nhiều lần xóa xen kẽ với phần chèn. Sau đó, một giải pháp là xây dựng một bảng khác lớn gấp đôi (với một hàm băm mới được liên kết) và quét toàn bộ bảng băm ban đầu, tính toán giá trị băm mới cho mỗi phần tử (không bị xóa) và chèn nó vào phần tử mới bàn.

|  |
| --- |
| 6 |
| 15 |
|  |
| 24 |
|  |
|  |
| 13 |

0

1

2

3

4

5

6

Hình 5.19 Bảng băm với đầu dò tuyến tính với đầu vào 13, 15, 6, 24

Ví dụ, giả sử các phần tử 13, 15, 24 và 6 được chèn vào bảng băm thăm dò tuyến tính có kích thước 7. Hàm băm là h (x) = x mod 7. Bảng băm kết quả xuất hiện trong Hình 5.19.

Nếu 23 được chèn vào bảng, bảng kết quả trong Hình 5.20 sẽ đầy hơn 70 phần trăm. Vì bảng đã quá đầy nên một bảng mới được tạo ra. Kích thước của bảng này là 17, vì đây là số nguyên tố đầu tiên lớn gấp đôi kích thước của bảng cũ. Khi đó, hàm băm mới là h (x) = x mod 17. Bảng cũ được quét và các phần tử 6, 15, 23, 24 và 13 được chèn vào bảng mới. Bảng kết quả xuất hiện trong Hình 5.21.

Hình 5.20 Bảng băm với đầu dò tuyến tính sau khi chèn 23

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
| 6 |
| 23 |
| 24 |
|  |
|  |
|  |
|  |
| 13 |
|  |
| 15 |
|  |

0

1

2

3

4

5

6

7

8

9

10

11

12

Hình 5.21 Bảng băm sau khi băm lại

Việc băm lại có thể được thực hiện theo một số cách với thăm dò bậc hai. Một cách thay thế là rehash ngay khi bàn đầy một nửaVì hiệu suất suy giảm khi hệ số tải tăng lên, nên chiến lược thứ ba, được thực hiện với mức giới hạn tốt, có thể là tốt nhất.

5.6 Bảng băm trong Thư viện chuẩn

Trong C ++ 11, Thư viện chuẩn bao gồm các triển khai bảng băm của các tập hợp và bản đồ— cụ thể là tập hợp và bản đồ không có thứ tự, là tập hợp và ánh xạ song song. Các mục trong bộ\_tập\_hợp\_đặt\_hàng (hoặc các khoá trong sơ đồ không có thứ tự) phải cung cấp toán tử đã được nạp chồng == và hàm băm, như được mô tả trước đó, trong Phần 5.3. Cũng giống như mẫu tập hợp và bản đồ cũng có thể được khởi tạo bằng một đối tượng hàm cung cấp (hoặc ghi đè một hàm so sánh mặc định), bộ không có thứ tự và bản đồ không có thứ tự có thể được khởi tạo bằng các đối tượng hàm cung cấp hàm băm và toán tử bình đẳng

1 /\*\*

2 \* Rehashing for quadratic probing hash table.

3 \*/

4 void rehash( )

5 {

6 vector<HashEntry> oldArray = array;

7

8 // Create new double-sized, empty table

9 array.resize( nextPrime( 2 \* oldArray.size( ) ) );

10 for( auto & entry : array )

11 entry.info = EMPTY;

12

13 // Copy table over

14 currentSize = 0;

15 for( auto & entry : oldArray )

16 if( entry.info == ACTIVE )

17 insert( std::move( entry.element ) );

18 }

19

20 /\*\*

21 \* Rehashing for separate chaining hash table.

22 \*/

23 void rehash( )

24 {

25 vector<list<HashedObj>> oldLists = theLists;

26

27 // Create new double-sized, empty table

28 theLists.resize( nextPrime( 2 \* theLists.size( ) ) );

29 for( auto & thisList : theLists )

30 thisList.clear( );

31

32 // Copy table over

33 currentSize = 0;

34 for( auto & thisList : oldLists )

35 for( auto & x : thisList )

36 insert( std::move( x ) );

37 }

Hình 5.22 Băm lại cho cả bảng băm chuỗi riêng biệt và bảng băm thăm dò

1 class CaseInsensitiveStringHash

2 {

3 public:

4 size\_t operator( ) ( const string & s ) const

5 {

6 static hash<string> hf;

7 return hf( toLower( s ) ); // toLower implemented elsewhere

8 }

9

10 bool operator( ) ( const string & lhs, const string & rhs ) const

11 {

12 return equalsIgnoreCase( lhs, rhs ); // equalsIgnoreCase is elsewhere

13 }

14 };

15

16 unordered\_set<string,CaseInsensitiveStringHash,CaseInsensitiveStringHash > s;

Hình 5.23 Tạo một tập hợp không có thứ tự phân biệt chữ hoa chữ thường

Các lớp không có thứ tự này có thể được sử dụng nếu nó không quan trọng đối với các mục có thể xem được theo thứ tự được sắp xếp. Ví dụ, trong ví dụ thay đổi từ trong Phần 4.8, có ba bản đồ:

1. Bản đồ trong đó khóa là độ dài từ và giá trị là tập hợp tất cả các từ có độ dài từ đó.

2. Bản đồ trong đó khóa là đại diện và giá trị là tập hợp tất cả các từ có đại diện đó.

3. Bản đồ trong đó khóa là một từ và giá trị là tập hợp tất cả các từ chỉ khác một ký tự từ từ đó.

Hiệu suất của bản đồ không có thứ tự thường có thể vượt trội hơn bản đồ, nhưng thật khó để biết chắc chắn nếu không viết mã theo cả hai cách.

5.7 Bảng băm với quyền truy cập Worst-Case O (1)

Các bảng băm mà chúng tôi đã kiểm tra cho đến nay đều có thuộc tính mà với các hệ số tải có thể hợp lý và các hàm băm thích hợp, chúng tôi có thể mong đợi chi phí trung bình là O (1) cho các lần chèn, loại bỏ và tìm kiếm. Nhưng trường hợp xấu nhất dự kiến xảy ra đối với một tìm kiếm giả sử một hàm băm hoạt động tốt là gì?

Đối với chuỗi riêng biệt, giả sử hệ số tải bằng 1, đây là một phiên bản của bài toán bi và thùng cổ điển: Cho N quả bóng được đặt ngẫu nhiên (đồng đều) trong N thùng, số quả bóng dự kiến trong thùng nhiều nhất là bao nhiêu? Câu trả lời được nhiều người biết đến là Θ (logN / loglogN), nghĩa là trung bình, chúng tôi mong đợi một số truy vấn mất thời gian gần như logarit. Các loại giới hạn tương tự được quan sát (hoặc có thể chứng minh được) cho độ dài của chuỗi thăm dò dự kiến dài nhất trong bảng băm thăm dò.

5.7.1 Băm hoàn hảo

Giả sử, với mục đích đơn giản hóa, tất cả N mục đều được biết trước. Nếu việc triển khai chuỗi riêng biệt có thể đảm bảo rằng mỗi danh sách có nhiều nhất một số mục không đổi, chúng ta sẽ hoàn thành. Chúng tôi biết rằng khi chúng tôi tạo nhiều danh sách hơn, trung bình các danh sách sẽ ngắn hơn, vì vậy về mặt lý thuyết, nếu chúng tôi có đủ danh sách, thì với xác suất cao hợp lý, chúng tôi có thể mong đợi không có va chạm nào cả!

Nhưng có hai vấn đề cơ bản với cách tiếp cận này: Thứ nhất, số lượng danh sách có thể lớn một cách bất hợp lý; thứ hai, ngay cả với rất nhiều danh sách, chúng tôi vẫn có thể gặp xui xẻo.

Thật không may, M cần phải khá lớn; speci = cally M = (N2). Tuy nhiên, nếu M = N2, chúng ta có thể chứng minh rằng bảng không có va chạm với xác suất ít nhất là 1, và kết quả này có thể là

được sử dụng để tạo ra một phương pháp sửa đổi khả thi cho cách tiếp cận cơ bản của chúng tôi.

Định lý 5.2

Nếu cho N viên bi vào thùng M = N2 thì xác suất để thùng nào có nhiều hơn một viên bi nhỏ hơn 2 viên.

1

Bằng chứng

Nếu một cặp (i, j) quả bóng được đặt trong cùng một thùng, chúng ta gọi đó là va chạm. Gọi Ci, j là số va chạm dự kiến tạo ra bởi hai quả bóng bất kỳ (i, j). Rõ ràng xác suất để hai quả bóng cụ thể va chạm là 1 / M, và do đó Ci, j là 1 / M, vì số lần va chạm liên quan đến cặp (i, j) là 0 hoặc 1. Do đó, số lần va chạm dự kiến trong toàn bộ bảng là (i, j), i <j Ci, j. Vì có N (N − 1) / 2 cặp nên tổng này là N (N - 1) / (2M) = N (N - 1) / (2N2) <2. Vì số va chạm dự kiến là dưới 2, xác suất để có một vụ va chạm cũng phải dưới 2.

Tất nhiên, việc sử dụng danh sách N2 là không thực tế. Tuy nhiên, phân tích trước cho thấy giải pháp thay thế fol-lowing: Chỉ sử dụng N thùng, nhưng giải quyết các xung đột trong mỗi thùng bằng cách sử dụng bảng băm thay vì danh sách được liên kết. Ý tưởng là bởi vì mỗi thùng dự kiến ​​chỉ có một vài mục, bảng băm được sử dụng cho mỗi thùng có thể là bậc hai trong kích thước thùng. Hình 5.24 cho thấy cấu trúc cơ bản. Ở đây, bảng băm chính có mười thùng. Các thùng 1, 3, 5, 7 đều trống. Các thùng 0, 4 và 8 có một mục, vì vậy chúng được phân giải bằng bảng băm phụ với một vị trí. Thùng 2 và 6 có hai mục, vì vậy chúng sẽ được phân giải thành một bảng băm phụ có bốn (22) vị trí. Và thùng 9 có ba mục, vì vậy nó được phân giải thành một bảng băm phụ với chín (32) vị trí.

Định lý 5.3

Nếu N mục được đặt vào một bảng băm chính chứa N thùng, thì tổng kích thước của các bảng băm phụ có giá trị mong đợi nhiều nhất là 2N.

0

22 = 4

1

2

3

4

22 = 4

5

6

7

8

9

32 = 9

Hình 5.24 Bảng băm hoàn hảo sử dụng bảng băm phụ

5.7.2 Hàm băm Cuckoo

Từ thảo luận trước, chúng ta biết rằng trong bài toán bi và thùng, nếu ném ngẫu nhiên N vật vào N thùng, thì kích thước của thùng lớn nhất được mong đợi là Θ (logN / loglogN). Vì giới hạn này đã được biết đến từ lâu và vấn đề đã được các nhà toán học nghiên cứu kỹ lưỡng, nên thật ngạc nhiên khi vào giữa những năm 1990, người ta đã chỉ ra rằng nếu ở mỗi lần tung, hai thùng được chọn ngẫu nhiên và vật phẩm đó là ném vào thùng rỗng hơn (tại thời điểm đó), khi đó kích thước của thùng lớn nhất sẽ chỉ là Θ (loglogN), một con số thực sự thấp hơn đáng kể. Một cách nhanh chóng, một loạt các thuật toán và cấu trúc dữ liệu tiềm năng đã nảy sinh từ khái niệm mới này về “sức mạnh của hai lựa chọn”.

Một trong những ý tưởng là hàm băm cuckoo. Trong hàm băm cuckoo, giả sử chúng ta có N mục. Chúng tôi duy trì hai bảng, mỗi bảng trống hơn một nửa và chúng tôi có hai hàm băm độc lập có thể gán mỗi mục vào một vị trí trong mỗi bảng. Cuckoo băm duy trì sự bất biến mà một mục luôn được lưu trữ ở một trong hai vị trí này.

Ví dụ, Hình 5.25 cho thấy một bảng băm cuckoo tiềm năng cho sáu mục, với hai bảng có kích thước 5 (những bảng này quá nhỏ, nhưng cũng dùng làm ví dụ). Dựa trên các hàm băm được chọn ngẫu nhiên, mục A có thể ở vị trí 0 trong Bảng 1 hoặc vị trí 2 trong Bảng 2. Mục F có thể ở vị trí 3 trong Bảng 1 hoặc vị trí 4 trong Bảng 2, v.v. Ngay lập tức, điều này ngụ ý rằng một tìm kiếm trong bảng băm cuckoo yêu cầu tối đa hai quyền truy cập vào bảng và việc xóa là nhỏ, khi mục đã được định vị (không cần xóa lười bây giờ!).

Bản thân thuật toán băm cuckoo rất đơn giản: Để chèn một mục mới, x, trước tiên hãy đảm bảo rằng nó chưa có ở đó. Sau đó, chúng ta có thể sử dụng hàm băm đầu tiên, và nếu vị trí bảng (đầu tiên) trống, mục có thể được đặt. Vì vậy, hình 5.26 cho thấy kết quả của việc chèn A vào một bảng băm trống.

Giả sử bây giờ chúng ta muốn chèn B, có các vị trí băm 0 trong Bảng 1 và 0 trong Bảng 2. Đối với phần còn lại của thuật toán, chúng tôi sẽ sử dụng (h1, h2) để xác định hai vị trí, vì vậy vị trí của B được cho bởi (0, 0). Bảng 1 đã được chiếm giữ ở vị trí 0. Tại thời điểm này, có hai lựa chọn: Một là tìm trong Bảng 2. Vấn đề là vị trí 0 trong Bảng 2 cũng có thể bị chiếm. Điều đó xảy ra trong trường hợp này là không, nhưng thuật toán mà bảng băm cuckoo tiêu chuẩn sử dụng không cần quan tâm. Thay vào đó, nó đặt trước mục mới B vào Bảng 1. Để làm như vậy, nó phải thay thế A, vì vậy A di chuyển đến Bảng 2, sử dụng vị trí băm Bảng 2 của nó, đó là vị trí 2. Kết quả được hiển thị trong Hình 5.27 . Có thể dễ dàng chèn C, và điều này được thể hiện trong Hình 5.28. Tiếp theo, chúng ta muốn chèn D, với các vị trí băm (1, 0). Nhưng vị trí Bảng 1(vị trí 1) đã được thực hiện. Cũng lưu ý rằng vị trí Bảng 2 chưa được thực hiện, nhưng chúng tôi không nhìn ở đó. Thay vào đó, chúng ta có D thay thế C, và sau đó C đi vào Bảng 2 ở vị trí 4, như được gợi ý bởi hàm băm thứ hai của nó. Bảng kết quả được thể hiện trong Hình 5.29

|  |  |
| --- | --- |
| Table 1 | |
| 0 | B |
| 1 | C |
| 2 |  |
| 3 | E |
| 4 |  |

|  |  |
| --- | --- |
| Table 2 | |
| 0 | D |
| 1 |  |
| 2 | A |
| 3 |  |
| 4 | F |

A: 0, 2

B: 0, 0

C: 1, 4

D: 1, 0

E: 3, 2

F: 3, 4

Hình 5.25 Bảng băm cuckoo tiềm năng. Hàm băm được hiển thị ở bên phải. Đối với sáu mục này, chỉ có ba vị trí hợp lệ trong Bảng 1 và ba vị trí hợp lệ trong Bảng 2, do đó không rõ ràng là có thể dễ dàng tìm thấy sự sắp xếp này.

|  |  |
| --- | --- |
| Table 2 | |
| 0 |  |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |

|  |  |
| --- | --- |
| Table 1 | |
| 0 | A |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |

A: 0, 2

Hình 5.26 Bảng băm Cuckoo sau khi chèn A

|  |  |
| --- | --- |
| Table 1 | |
| 0 | B |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |

|  |  |
| --- | --- |
| Table 2 | |
| 0 |  |
| 1 |  |
| 2 | A |
| 3 |  |
| 4 |  |

A: 0, 2

B: 0, 0

Hình 5.27 Bảng băm Cuckoo sau khi chèn B

|  |  |
| --- | --- |
| Table 1 | |
| 0 | B |
| 1 | C |
| 2 |  |
| 3 |  |
| 4 |  |

|  |  |
| --- | --- |
| Table 2 | |
| 0 |  |
| 1 |  |
| 2 | A |
| 3 |  |
| 4 |  |

A: 0, 2

B: 0, 0

C: 1, 4

Hình 5.28 Bảng băm Cuckoo sau khi chèn C

|  |  |
| --- | --- |
| Table 2 | |
| 0 |  |
| 1 |  |
| 2 | A |
| 3 |  |
| 4 | C |

|  |  |
| --- | --- |
| Table 1 | |
| 0 | B |
| 1 | D |
| 2 |  |
| 3 | E |
| 4 |  |

A: 0, 2

B: 0, 0

C: 1, 4

D: 1, 0

E: 3, 2

Hình 5.29 Bảng băm Cuckoo sau khi chèn D

|  |  |
| --- | --- |
| Table 1 | |
| 0 | B |
| 1 | D |
| 2 |  |
| 3 | F |
| 4 |  |

|  |  |
| --- | --- |
| Table 2 | |
| 0 |  |
| 1 |  |
| 2 | A |
| 3 |  |
| 4 | C |

A: 0, 2

B: 0, 0

C: 1, 4

D: 1, 0

E: 3, 2

F: 3, 4

Hình 5.30 Bảng băm Cuckoo bắt đầu chèn F vào bảng trong Hình 5.29. Đầu tiên, F thay thế E.

|  |  |
| --- | --- |
| Table 1 | |
| 0 | B |
| 1 | D |
| 2 |  |
| 3 | F |
| 4 |  |

|  |  |
| --- | --- |
| Table 2 | |
| 0 |  |
| 1 |  |
| 2 | E |
| 3 |  |
| 4 | C |

A: 0, 2

B: 0, 0

C: 1, 4

D: 1, 0

E: 3, 2

F: 3, 4

Hình 5.31 Tiếp tục chèn F vào bảng trong Hình 5.29. Tiếp theo, E thay thế A.

|  |  |
| --- | --- |
| Table 1 | |
| 0 | A |
| 1 | D |
| 2 |  |
| 3 | F |
| 4 |  |

|  |  |
| --- | --- |
| Table 2 | |
| 0 |  |
| 1 |  |
| 2 | E |
| 3 |  |
| 4 | C |

A: 0, 2

B: 0, 0

C: 1, 4

D: 1, 0

E: 3, 2

F: 3, 4

Hình 5.32 Tiếp tục chèn F vào bảng trong Hình 5.29. Tiếp theo, A thay thế B.

|  |  |
| --- | --- |
| Table 1 | |
| 0 | A |
| 1 | D |
| 2 |  |
| 3 | F |
| 4 |  |

|  |  |
| --- | --- |
| Table 2 | |
| 0 | B |
| 1 |  |
| 2 | E |
| 3 |  |
| 4 | C |

A: 0, 2

B: 0, 0

C: 1, 4

D: 1, 0

E: 3, 2

F: 3, 4

Hình 5.33 Hoàn thành việc chèn F vào bảng trong Hình 5.29. Thật kỳ diệu (?), Nằm ở vị trí trống trong Bảng 2.

|  |  |
| --- | --- |
| Table 1 | |
| 0 | B |
| 1 | C |
| 2 |  |
| 3 | E |
| 4 |  |

|  |  |
| --- | --- |
| Table 2 | |
| 0 | D |
| 1 |  |
| 2 | A |
| 3 |  |
| 4 | F |

A: 0, 2

B: 0, 0

C: 1, 4

D: 1, 0

E: 3, 2

F: 3, 4

G: 1, 2

Hình 5.34 Chèn G vào bảng trong Hình 5.33. G thay thế D, thay thế B, thay thế A, thay thế E, thay thế F, thay thế C, thay thế G. ở vị trí 2. Tuy nhiên, nói chung điều đó có thể thành công, trong trường hợp này có một chu kỳ và việc chèn sẽ không kết thúc.

Sau khi làm xong, E có thể dễ dàng lắp vào. Cho đến nay, rất tốt, nhưng bây giờ chúng ta có thể chèn F không? Hình 5.30 đến 5.33 cho thấy thuật toán này chèn F thành công, bằng cách dịch chuyển E, A, rồi B.

Sau khi công bố băm cuckoo, nhiều phần mở rộng đã được đề xuất. Ví dụ, thay vì hai bảng, chúng ta có thể sử dụng số lượng bảng cao hơn, chẳng hạn như 3 hoặc 4. Trong khi điều này làm tăng chi phí tra cứu, nó cũng làm tăng đáng kể việc sử dụng không gian lý thuyết. Trong một số ứng dụng, việc tra cứu thông qua các hàm băm riêng biệt có thể được thực hiện song song và do đó tốn ít hoặc không tốn thêm thời gian. Một phần mở rộng khác là cho phép mỗi bảng lưu trữ nhiều khóa; một lần nữa, điều này có thể tăng khả năng sử dụng không gian và giúp việc chèn dễ dàng hơn và có thể thân thiện với bộ nhớ cache hơn. Có thể có nhiều cách kết hợp khác nhau, như trong Hình 5.35. Và cuối cùng, các bảng băm cuckoo thường được triển khai dưới dạng một bảng khổng lồ với hai (hoặc nhiều) hàm băm thăm dò toàn bộ bảng và một số biến thể cố gắng đặt một mục trong bảng băm thứ hai ngay lập tức nếu có một vị trí khả dụng, thay vì bắt đầu một chuỗi các phép dời hình.

Triển khai bảng băm Cuckoo

Thực hiện băm cuckoo yêu cầu một tập hợp các hàm băm; đơn giản chỉ sử dụng Mã băm để tạo ra tập hợp các hàm băm không có ý nghĩa gì, vì bất kỳ xung đột nào trong Mã băm sẽ dẫn đến xung đột trong tất cả các hàm băm. Hình 5.36 cho thấy một giao diện đơn giản có thể được sử dụng để gửi các họ hàm băm đến bảng băm cuckoo.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 1 item per cell | 2 items per cell | 4 items per cell |
| 2 hash functions | 0.49 | 0.86 | 0.93 |
| 3 hash functions | 0.91 | 0.97 | 0.98 |
| 4 hash functions | 0.97 | 0.99 | 0.999 |

Hình 5.35 Hệ số tải tối đa cho các biến thể của hàm băm cuckoo

1 template <typename AnyType>

2 class CuckooHashFamily

3 {

4 public:

5 size\_t hash( const AnyType & x, int which ) const;

6 int getNumberOfFunctions( );

7 void generateNewFunctions( );

8 };

Hình 5.36 Giao diện HashFamily chung để băm cuckoo

Hình 5.37 cung cấp giao diện lớp để băm cuckoo. Chúng tôi sẽ viết mã một biến thể cho phép một số lượng hàm băm tùy ý (được xác định bởi kiểu tham số mẫu HashFamily) sử dụng một mảng duy nhất được giải quyết bởi tất cả các hàm băm.

Trong hình 5.37, chúng tôi chỉ định rằng tải tối đa cho bảng là 0,4; nếu hệ số tải của bảng sắp vượt quá giới hạn này, một mở rộng bảng tự động sẽ được hình thành. Chúng tôi cũng xác định ALLOWED\_REHASHES, trong đó xác định số lần lặp lại chúng tôi sẽ thực hiện nếu việc trục xuất diễn ra quá lâu. Về lý thuyết, ALLOWED\_REHASHES có thể ở mức tối thiểu, vì chúng tôi mong đợi chỉ cần một số lần lặp lại không đổi nhỏ; trong thực tế, tùy thuộc vào một số yếu tố như số lượng hàm băm, chất lượng của hàm băm và hệ số tải, các lần lặp lại có thể làm chậm mọi thứ đáng kể và có thể đáng giá để mở rộng bảng, mặc dù điều này sẽ tốn kém không gian. Biểu diễn dữ liệu cho bảng băm cuckoo rất đơn giản: Chúng tôi lưu trữ một mảng đơn giản, kích thước hiện tại và tập hợp các hàm băm, được biểu diễn trong một phiên bản HashFamily. Chúng tôi cũng duy trì số lượng hàm băm, mặc dù luôn có được từ phiên bản HashFamily.

Hình 5.38 cho thấy các phương thức constructor và makeEmpty, và đây là các phương thức thẳng hàng. Hình 5.39 cho thấy một cặp phương thức riêng. Đầu tiên, myHash, được sử dụng để chọn hàm băm thích hợp và sau đó chia tỷ lệ thành một chỉ số mảng hợp lệ. Thứ hai, findPos, tham vấn tất cả các hàm băm để trả về chỉ mục chứa mục x hoặc −1 nếu không tìm thấy x. findPos sau đó được sử dụng bởi hàm chứa và loại bỏ tương ứng trong Hình 5.40 và 5.41, và chúng ta có thể thấy rằng các phương pháp đó rất dễ thực hiện.

Thói quen đình đám khác là chèn. Trong hình 5.42, chúng ta có thể thấy rằng kế hoạch cơ bản là kiểm tra xem mặt hàng đã có mặt chưa, nếu có thì quay lại. Nếu không, chúng tôi kiểm tra xem bảng đã được tải đầy đủ chưa và nếu có, chúng tôi sẽ mở rộng nó. Cuối cùng, chúng tôi gọi một thói quen trợ giúp để làm tất cả các công việc bẩn thỉu.

1 template <typename AnyType, typename HashFamily>

2 class CuckooHashTable

3 {

4 public:

5 explicit CuckooHashTable( int size = 101 );

6

7 void makeEmpty( );

8 bool contains( const AnyType & x ) const;

9

10 bool remove( const AnyType & x );

11 bool insert( const AnyType & x );

12 bool insert( AnyType && x );

13

14 private:

15 struct HashEntry

16 {

17 AnyType element;

18 bool isActive;

19

20 HashEntry( const AnyType & e = AnyType( ), bool a = false )

21 : element{ e }, isActive{ a } { }

22 HashEntry( AnyType && e, bool a = false )

23 : element{ std::move( e ) }, isActive{ a } { }

24 };

25

26 bool insertHelper1( const AnyType & xx );

27 bool insertHelper1( AnyType && xx );

28 bool isActive( int currentPos ) const;

29

30 size\_t myhash( const AnyType & x, int which ) const;

31 int findPos( const AnyType & x ) const;

32 void expand( ); 33 void rehash( );

34 void rehash( int newSize );

35

36 static const double MAX\_LOAD = 0.40;

37 static const int ALLOWED\_REHASHES = 5;

38

39 vector<HashEntry> array;

40 int currentSize;

41 int numHashFunctions;

42 int rehashes;

43 UniformRandom r;

44 HashFamily hashFunctions; }

Hình 5.37 Giao diện lớp để băm cuckoo

1 explicit HashTable( int size = 101 ) : array( nextPrime( size ) )

2 {

3 numHashFunctions = hashFunctions.getNumberOfFunctions( );

4 rehashes = 0;

5 makeEmpty( );

6 }

7

8 void makeEmpty( )

9 {

10 currentSize = 0;

11 for( auto & entry : array )

12 entry.isActive = false; }

Hình 5.38 Các quy trình khởi tạo và làm trống bảng băm cuckoo

1 /\*\*

2 \* Compute the hash code for x using specified function. 3 \*/

4 int myhash( const AnyType & x, int which ) const 5 {

6 return hashFunctions.hash( x, which ) % array.size( );

7 }

8

9 /\*\*

10 \* Search all hash function places. Return the position 11 \* where the search terminates or -1 if not found.

12 \*/

13 int findPos( const AnyType & x ) const

14 {

15 for( int i = 0; i < numHashFunctions; ++i )

16 {

17 int pos = myhash( x, i );

18

19 if( isActive( pos ) && array[ pos ].element == x ) 20 return pos;}

23 return -1;

24 }

Hình 5.39 Các quy trình tìm kiếm vị trí của một mục trong bảng băm cuckoo và tính toán mã băm cho một bảng nhất định

1 /\*\*

2 \* Return true if x is found. 3 \*/

4 bool contains( const AnyType & x ) const 5 {

6 return findPos( x ) != -1; 7 }

Hình 5.40 Quy trình tìm kiếm bảng băm cuckoo

1 /\*\*

2 \* Remove x from the hash table.

3 \* Return true if item was found and removed.

4 \*/

5 bool remove( const AnyType & x )

6 {

7 int currentPos = findPos( x );

8 if( !isActive( currentPos ) )

9 return false;

10

11 array[ currentPos ].isActive = false;

12 --currentSize;

13 return true;

14 }

Hình 5.41 Quy trình xóa khỏi bảng băm cuckoo

1 bool insert( const AnyType & x )

2 {

3 if( contains( x ) )

4 return false;

5

6 if( currentSize >= array.size( ) \* MAX\_LOAD )

7 expand( );

8

9 return insertHelper1( x );

10 }

Hình 5.42 Quy trình chèn công khai để băm cuckoo

Logic cơ bản của chúng tôi khác với sơ đồ cổ điển. Chúng tôi đã kiểm tra rằng mục cần chèn vẫn chưa có mặt. Tại dòng 15 đến dòng 25, chúng tôi kiểm tra xem có vị trí hợp lệ nào trống không; nếu vậy, chúng tôi đặt mục của chúng tôi ở vị trí có sẵn đầu tiên và chúng tôi đã hoàn tất. Nếu không, chúng tôi sẽ loại bỏ một trong các mặt hàng hiện có. Tuy nhiên, có một số vấn đề phức tạp:

1 static const int ALLOWED\_REHASHES = 5;

2

3 bool insertHelper1( const AnyType & xx )

4 {

5 const int COUNT\_LIMIT = 100;

6 AnyType x = xx;

7

8 while( true )

9 {

10 int lastPos = -1;

11 int pos;

12

13 for( int count = 0; count < COUNT\_LIMIT; ++count )

14 {

15 for( int i = 0; i < numHashFunctions; ++i )

16 {

17 pos = myhash( x, i );

18

19 if( !isActive( pos ) )

20 {

21 array[ pos ] = std::move( HashEntry{ std::move( x ), true } );

22 ++currentSize;

23 return true;

24 }

25 }

26

27 // None of the spots are available. Evict a random one

28 int i = 0;

29 do

30 {

31 pos = myhash( x, r.nextInt( numHashFunctions ) );

32 } while( pos == lastPos && i++ < 5 );

33

34 lastPos = pos;

35 std::swap( x, array[ pos ].element );

36 }

37

38 if( ++rehashes > ALLOWED\_REHASHES )

39 {

40 expand( ); // Make the table bigger

41 rehashes = 0; // Reset the # of rehashes

42 }

43 else

44 rehash( ); // Same table size, new hash functions

45 }

46 }

Hình 5.43 Quy trình chèn để băm cuckoo sử dụng một thuật toán khác chọn mục để loại bỏ một cách ngẫu nhiên, cố gắng không loại bỏ lại mục cuối cùng. Bảng sẽ cố gắng chọn các hàm băm mới (rehash) nếu có quá nhiều lần loại bỏ và sẽ mở rộng nếu có quá nhiều lần lặp lại.

Việc chứng minh mục đầu tiên không hoạt động tốt trong các thí nghiệm.

Việc chứng minh mục cuối cùng không hoạt động tốt trong các thí nghiệm.

Việc loại bỏ các mục theo trình tự (tức là loại bỏ lần đầu tiên sử dụng hàm băm 0, các mục tiếp theo sử dụng hàm băm 1, v.v.) không hoạt động tốt trong các thử nghiệm.

Việc loại bỏ mục hoàn toàn ngẫu nhiên không hoạt động tốt trong các thí nghiệm: Đặc biệt, chỉ với hai hàm băm, nó có xu hướng tạo ra các chu kỳ.

1 void expand( )

2 {

3 rehash( static\_cast<int>( array.size( ) / MAX\_LOAD ) );

4 }

5

6 void rehash( )

7 {

8 hashFunctions.generateNewFunctions( );

9 rehash( array.size( ) );

10 }

11

12 void rehash( int newSize )

13 {

14 vector<HashEntry> oldArray = array;

15

16 // Create new double-sized, empty table

17 array.resize( nextPrime( newSize ) );

18 for( auto & entry : array )

19 entry.isActive = false;

20

21 // Copy table over

22 currentSize = 0;

23 for( auto & entry : oldArray )

24 if( entry.isActive )

25 insert( std::move( entry.element ) );

26 }

Hình 5.44 Mã băm lại và mở rộng cho bảng băm cuckoo

1 template <int count>

2 class StringHashFamily

3 {

4 public:

5 StringHashFamily( ) : MULTIPLIERS( count )

6 {

7 generateNewFunctions( );

8 }

9

10 int getNumberOfFunctions( ) const

11 {

12 return count;

13 }

14

15 void generateNewFunctions( )

16 {

17 for( auto & mult : MULTIPLIERS )

18 mult = r.nextInt( );

19 }

20

21 size\_t hash( const string & x, int which ) const

22 {

23 const int multiplier = MULTIPLIERS[ which ];

24 size\_t hashVal = 0;

25

26 for( auto ch : x )

27 hashVal = multiplier \* hashVal + ch;

28

29 return hashVal;

30 }

31

32 private:

33 vector<int> MULTIPLIERS;

34 UniformRandom r;

35 };

Hình 5.45 Băm chuỗi thông thường để băm chim cu gáy; các hàm băm này không đáp ứng đầy đủ các yêu cầu cần thiết cho việc băm cuckoo nhưng cung cấp hiệu suất tốt nếu bảng không được tải nhiều và sử dụng quy trình chèn thay thế trong Hình 5.43.

Lợi ích của việc băm cuckoo bao gồm thời gian tra cứu và xóa liên tục trong trường hợp xấu nhất, tránh xóa lười và dữ liệu bổ sung, và khả năng xảy ra song song.

Tuy nhiên, băm cuckoo cực kỳ nhạy cảm với việc lựa chọn các hàm băm; inven-tors của bảng băm cuckoo báo cáo rằng nhiều hàm băm tiêu chuẩn mà họ đã cố gắng thực hiện kém trong các thử nghiệm. Hơn nữa, mặc dù thời gian chèn dự kiến ​​là thời gian không đổi miễn là hệ số tải dưới 1, nhưng giới hạn đã được hiển thị cho chi phí chèn dự kiến ​​cho phép băm cổ điển cuckoo với hai bảng riêng biệt (cả hai đều có hệ số tải λ) là khoảng 1 / (1 - (4λ2) 1/3), suy giảm nhanh chóng khi hệ số tải gần bằng 2 (bản thân công thức không có ý nghĩa khi λ bằng hoặc vượt quá). Sử dụng hệ số tải thấp hơn hoặc nhiều hơn hai hàm băm có vẻ hợp lý thay thế.

5.7.3 Hàm băm lò cò

Hàm băm lò cò là một thuật toán mới cố gắng cải thiện thuật toán thăm dò tuyến tính cổ điển. Nhớ lại rằng trong thăm dò tuyến tính, các ô được thử theo thứ tự tuần tự, bắt đầu từ vị trí băm. Do phân cụm sơ cấp và thứ cấp, trình tự này có thể dài trung bình khi bảng được tải và do đó nhiều cải tiến như thăm dò bậc hai, băm kép, v.v., đã được đề xuất để giảm số lượng va chạm. Tuy nhiên, trên một số kiến ​​trúc hiện đại, vị trí được tạo ra bằng cách thăm dò các ô liền kề là một yếu tố quan trọng hơn so với các đầu dò bổ sung, và thăm dò tuyến tính vẫn có thể thực tế hoặc thậm chí là lựa chọn tốt nhất.

Đặt MAX\_DIST là giới hạn đã chọn trên chuỗi thăm dò tối đa. Điều này có nghĩa là mục x phải được tìm thấy ở đâu đó trong các vị trí MAX\_DIST được liệt kê trong hash (x), hash (x) + 1, ..., hash (x) + (MAX\_DIST - 1). Để xử lý nhanh chóng việc loại bỏ, chúng tôi duy trì thông tin cho mỗi vị trí x, liệu mục ở vị trí thay thế có bị chiếm bởi một phần tử băm đến vị trí x hay không.

Ví dụ, hình 5.46 cho thấy một bảng băm nhảy lò cò khá đông đúc, sử dụng MAX\_DIST = 4. Mảng bit cho vị trí 6 cho thấy chỉ có vị trí 6 có một mục (C) với giá trị băm 6: Chỉ bit đầu tiên của Hop [6] được thiết lập. Hop [7] có tập hợp hai bit đầu tiên, cho biết rằng vị trí 7 và 8 (A và D) được sử dụng với các mục có giá trị băm là 7. Và Hop [8] chỉ có bộ bit thứ ba, cho biết rằng mục ở vị trí 10 (E) có giá trị băm 8. Nếu MAX\_DIST không quá 32, mảng Hop về cơ bản là một mảng các số nguyên 32 bit, vì vậy yêu cầu về không gian bổ sung là không đáng kể. Nếu Hop [pos] chứa tất cả 1s cho một số pos, thì nỗ lực chèn một mục có giá trị băm là pos rõ ràng sẽ không thành công, vì bây giờ sẽ có MAX\_DIST + 1 mục đang cố gắng nằm trong MAX\_DIST vị trí của pos — một điều bất khả thi.

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1000 |
| 10 | E | 0000 |
| 11 | G | 1000 |
| 12 | F | 1000 |
| 13 |  | 0000 |
| 14 |  | 0000 |
| ... | | |

A: 7

B: 9

C: 6

D: 7

F: 12

G: 1

Hình 5.46 Bảng băm lò cò. Các bước nhảy cho biết vị trí nào trong khối được chiếm bởi các ô chứa giá trị băm này. Do đó Hop [8] = 0010 chỉ ra rằng chỉ có vị trí 10 hiện chứa các mục có giá trị băm là 8, trong khi các vị trí 8, 9 và 11 thì không.

Vì vậy, chúng tôi thử Hop [12], nơi chúng tôi cần vị trí đầu tiên là 1, chính là nó. Như vậy F có thể di chuyển xuống dưới. Hai bước này được thể hiện trong Hình 5.48. Tuy nhiên, đó không phải là vấn đề với thuật toán của chúng tôi; thay vào đó, sẽ không có cách nào để đặt tất cả C, I, A, D, E, B, H và F (nếu giá trị băm của F là 9); tất cả các mục này sẽ có giá trị băm từ 6 đến 9, và do đó cần phải được đặt ở bảy vị trí từ 6 đến 12. Nhưng đó sẽ là tám mục trong bảy vị trí - một điều không thể. Tuy nhiên, vì đây không phải là trường hợp của ví dụ của chúng tôi và chúng tôi đã loại bỏ một mục khỏi vị trí 12, giờ chúng tôi có thể tiếp tục. Hình 5.49 cho thấy sự di chuyển còn lại khỏi vị trí 9 và vị trí tiếp theo của I.

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1000 |
| 10 | E | 0000 |
| 11 | G | 1000 |
| 12 | F | 1000 |
| 13 |  | 0000 |
| 14 |  | 0000 |
| ... | | |

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1000 |
| 10 | E | 0000 |
| 11 |  | 0010 |
| 12 | F | 1000 |
| 13 | G | 0000 |
| 14 |  | 0000 |
| ... | | |

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1010 |
| 10 | E | 0000 |
| 11 | H | 0010 |
| 12 | F | 1000 |
| 13 | G | 0000 |
| 14 |  | 0000 |
| ... | | |

A: 7

B: 9

C: 6

D: 7

E: 8

F: 12

G: 11

H: 9

Hình 5.47 Bảng băm lò cò. Cố gắng chèn H. Thăm dò tuyến tính gợi ý vị trí 13, nhưng vị trí đó quá xa, vì vậy chúng tôi loại bỏ G khỏi vị trí 11 và ﬁ một vị trí gần hơn.

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1010 |
| 10 | E | 0000 |
| 11 | H | 0010 |
| 12 | F | 1000 |
| 13 | G | 0000 |
| 14 |  | 0000 |
| ... | | |

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1010 |
| 10 | E | 0000 |
| 11 | H | 0001 |
| 12 | F | 1000 |
| 13 |  | 0000 |
| 14 | G | 0000 |
| ... | | |

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1010 |
| 10 | E | 0000 |
| 11 | H | 0001 |
| 12 |  | 0100 |
| 13 | F | 0000 |
| 14 | G | 0000 |
| ... | | |

A: 7

B: 9

C: 6

D: 7

E: 8

F: 12

G: 11

H: 9

I: 6

Hình 5.47 Bảng băm lò cò. Cố gắng chèn H. Thăm dò tuyến tính gợi ý vị trí 13, nhưng vị trí đó quá xa, vì vậy chúng tôi loại bỏ G khỏi vị trí 11 và ﬁ một vị trí gần hơn.

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1010 |
| 10 | E | 0000 |
| 11 | H | 0010 |
| 12 | F | 1000 |
| 13 | G | 0000 |
| 14 |  | 0000 |
| ... | | |

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1010 |
| 10 | E | 0000 |
| 11 | H | 0001 |
| 12 | F | 1000 |
| 13 |  | 0000 |
| 14 | G | 0000 |
| ... | | |

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1010 |
| 10 | E | 0000 |
| 11 | H | 0001 |
| 12 |  | 0100 |
| 13 | F | 0000 |
| 14 | G | 0000 |
| ... | | |

A: 7

B: 9

C: 6

D: 7

E: 8

F: 12

G: 11

H: 9

I: 6

Hình 5.48 Bảng băm Hopscotch. Cố gắng chèn I. Thăm dò tuyến tính cho thấy vị trí 14, nhưng điều đó quá xa; tham khảo ý kiến Hop [11], chúng tôi thấy rằng G có thể di chuyển xuống, bỏ ngỏ vị trí 13. Tư vấn Hop [10] không đưa ra đề xuất nào. Hop [11] cũng không giúp được gì (tại sao?), Vì vậy Hop [12] đề nghị di chuyển F.

Hopscotch băm là một thuật toán tương đối mới, nhưng kết quả thử nghiệm ban đầu rất đáng mừng, đặc biệt là đối với các ứng dụng sử dụng nhiều bộ xử lý và yêu cầu sự song song và đồng thời đáng kể. Vẫn còn phải xem nếu băm chim cu gáy hoặc băm lò cò nổi lên như một sự thay thế thực tế cho các sơ đồ thăm dò chuỗi và tuyến tính / bậc hai riêng biệt cổ điển.

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | B | 1010 |
| 10 | E | 0000 |
| 11 | H | 0001 |
| 12 |  | 0100 |
| 13 | F | 0000 |
| 14 | G | 0000 |
| ... | | |

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1000 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 |  | 0011 |
| 10 | E | 0000 |
| 11 | H | 0001 |
| 12 | B | 0100 |
| 13 | F | 0000 |
| 14 | G | 0000 |
| ... | | |

|  |  |  |
| --- | --- | --- |
|  | Item | Hop |
| ... | | |
| 6 | C | 1001 |
| 7 | A | 1100 |
| 8 | D | 0010 |
| 9 | I | 0011 |
| 10 | E | 0000 |
| 11 | H | 0001 |
| 12 | B | 0100 |
| 13 | F | 0000 |
| 14 | G | 0000 |
| ... | | |

A: 7

B: 9

C: 6

D: 7

E: 8

F: 12

G: 11

H: 9

I: 6

Hình 5.49 Bảng băm Hopscotch. Việc chèn I tiếp tục: Tiếp theo, B bị loại ra, và cuối cùng, chúng ta có một vị trí đủ gần với giá trị băm và có thể chèn I.

5.8 Băm chung

Mặc dù bảng băm rất hiệu quả và có chi phí trung bình không đổi cho mỗi hoạt động, giả sử các hệ số tải phù hợp, phân tích và hiệu suất của chúng phụ thuộc vào hàm băm có hai thuộc tính cơ bản:

1. Hàm băm phải có thể tính toán được trong thời gian không đổi (tức là không phụ thuộc vào số lượng mục trong bảng băm).

2. Hàm băm phải phân phối các mục của nó một cách đồng nhất giữa các vị trí mảng.

Đặc biệt, nếu hàm băm kém, thì tất cả các cược sẽ tắt và chi phí cho mỗi hoạt động có thể là tuyến tính. Trong phần này, chúng ta thảo luận về các hàm băm phổ quát, cho phép chúng ta chọn hàm băm một cách ngẫu nhiên theo cách mà điều kiện 2 ở trên được thỏa mãn. Như trong Phần 5.7, chúng tôi sử dụng M để đại diện cho TableSize. Mặc dù động lực mạnh mẽ cho việc sử dụng các hàm băm phổ quát là cung cấp cơ sở lý thuyết cho các giả định được sử dụng trong các phân tích bảng băm cổ điển, nhưng các hàm này cũng có thể được sử dụng trong các ứng dụng yêu cầu mức độ mạnh mẽ, trong trường hợp xấu nhất (hoặc thậm chí xuống cấp đáng kể) hiệu suất, có lẽ dựa trên đầu vào do kẻ phá hoại hoặc tin tặc tạo ra, đơn giản là không thể chấp nhận được.

Như trong Phần 5.7, chúng tôi sử dụng M để đại diện cho TableSize.

Định nghĩa 5.1

Họ H của các hàm băm là phổ quát, nếu với bất kỳ x = y nào, thì số hàm băm h trong H mà h (x) = h (y) là nhiều nhất | H | / M.

Lưu ý rằng định nghĩa này áp dụng cho từng cặp mục, thay vì được tính trung bình cho tất cả các cặp mục. Định nghĩa ở trên có nghĩa là nếu chúng ta chọn ngẫu nhiên một hàm băm từ một họ phổ quát H, thì xác suất xảy ra va chạm giữa bất kỳ hai mục khác biệt nào nhiều nhất là 1 / M và khi thêm vào một bảng với N mục, xác suất của một va chạm tại điểm ban đầu tối đa là N / M, hoặc hệ số tải trọng.

Việc sử dụng một hàm băm phổ quát để băm chuỗi riêng biệt hoặc băm nhảy lò cò sẽ có thể làm thay đổi được các giả định được sử dụng trong phân tích về cấu trúc đo lường. Để phân tích hoạt động, mỗi bước phải có xác suất va chạm là N / M độc lập, với một mục x khác nhau là đối tượng của hàm băm. Chúng tôi có thể chính thức hóa yêu cầu độc lập này theo định nghĩa sau.

Định nghĩa 5.2

Họ H của các hàm băm là k-phổ quát, nếu với bất kỳ x1 = y1, x2 = y2, ..., xk = yk, thì số hàm băm h trong H mà h (x1) = h (y1), h (x2) = h (y2), ..., và h (xk) = h (yk) tối đa là | H | / M.

Với khái niệm này, chúng tôi thấy rằng việc phân tích băm cuckoo yêu cầu hàm băm O (logN) -universal (sau nhiều lần loại bỏ, chúng tôi từ bỏ và xử lý lại). Trong phần này, chúng ta chỉ xem xét các hàm băm phổ quát.

Để thiết kế một hàm băm phổ quát đơn giản, trước tiên chúng ta sẽ giả sử rằng chúng ta đang ánh xạ các số nguyên rất lớn thành các số nguyên nhỏ hơn trong khoảng từ 0 đến M - 1. Cho p là một số nguyên tố lớn hơn khóa đầu vào lớn nhất.

Họ phổ quát H của chúng ta sẽ bao gồm tập hợp các hàm sau, trong đó a và b được chọn ngẫu nhiên:

H = {Ha, b (x) = ((ax + b) mod p) mod M, trong đó 1 ≤ a ≤ p - 1,0 ≤ b ≤ p - 1}

Ví dụ, trong họ này, ba trong số các lựa chọn ngẫu nhiên có thể có của (a, b) mang lại ba hàm băm khác nhau:

H3,7(x) = ((3x + 7) mod p) mod M H4,1(x) = ((4x + 1) mod p) mod M H8,0(x) = ((8x) mod p) mod M

Quan sát rằng có p (p - 1) hàm băm có thể được chọn.

Định lý 5.4

Họ băm H = {Ha, b (x) = ((ax + b) mod p) mod M, trong đó 1 ≤ a ≤ p - 1, 0 ≤ b ≤ p - 1} là phổ quát.

Việc thực hiện hàm băm này dường như yêu cầu hai hoạt động mod: một mod p và mod thứ hai M. Hình 5.50 cho thấy một cách triển khai đơn giản trong C ++, giả sử rằng M có nghĩa không nhỏ hơn 231 - 1. Bởi vì các phép tính bây giờ phải chính xác như cụ thể, và do đó quá nhiều không còn được chấp nhận, chúng tôi khuyến khích các phép tính dài dài, có ít nhất 64 bit.

Tuy nhiên, chúng ta được phép chọn bất kỳ số nguyên tố p nào, miễn là nó lớn hơn M. Do đó, nên chọn một số nguyên tố thuận lợi nhất cho các phép tính. Một số nguyên tố như vậy là p = 231 - 1. Các số nguyên tố dạng này được gọi là số nguyên tố Mersenne; các số nguyên tố Mersenne khác bao gồm 25 - 1, 261 - 1 và 289 - 1. Cũng như phép nhân với một số nguyên tố Mersenne chẳng hạn như 31 có thể được thực hiện bằng một phép dịch chuyển bit và một phép trừ, một phép toán mod liên quan đến một số nguyên tố Mersenne cũng có thể được thực hiện bằng một chút thay đổi và một bổ sung:

Giả sử r ≡ y (mod p). Nếu chúng ta chia y cho (p + 1) thì y = q (p + 1) + r, trong đó q và r tương ứng là thương và dư. Do đó, r ≡ q (p + 1) + r (mod p).

Và vì (p + 1) ≡ 1 (mod p), ta thu được r ≡ q + r (mod p).

1 int universalHash( int x, int A, int B, int P, int M )

2 {

3 return static\_cast<int>( ( ( static\_cast<long long>( A ) \* x ) + B ) % P ) % M;

4 }

Hình 5.50 Cách thực hiện đơn giản của băm phổ quát

1 const int DIGS = 31;

2 const int mersennep = (1<<DIGS) - 1;

3

4 int universalHash( int x, int A, int B, int M )

5 {

6 long long hashVal = static\_cast<long long>( A ) \* x + B;

7

8 hashVal = ( ( hashVal >> DIGS ) + ( hashVal & mersennep ) );

9 if( hashVal >= mersennep )

10 hashVal -= mersennep; 11

12 return static\_cast<int>( hashVal ) % M;

13 }

Hình 5.51 Cách thực hiện đơn giản của băm phổ quát

Hình 5.51 thực hiện ý tưởng này, được gọi là thủ thuật Carter-Wegman. Trên dòng 8, phép chuyển bit tính thương số và phép bit-và tính phần dư khi chia cho (p 1); các phép toán bitwise này hoạt động vì (p 1) là lũy thừa chính xác của hai. Vì phần còn lại có thể lớn gần bằng p, nên tổng kết quả có thể lớn hơn p, vì vậy chúng tôi chia tỷ lệ xuống ở dòng 9 và 10.

5.9 Hàm băm mở rộng

Chủ đề cuối cùng của chúng ta trong chương này đề cập đến trường hợp lượng dữ liệu quá lớn không thể chứa trong bộ nhớ chính.

Một giải pháp thay thế thông minh, được gọi là băm có thể mở rộng, cho phép thực hiện tìm kiếm trong hai lần truy cập đĩa. Phần chèn cũng yêu cầu ít quyền truy cập đĩa.

WerecallfromChapter4thataB-treehasdepthO (log N) .AsMincreases, độ sâu của cây B giảm. Theo lý thuyết, chúng ta có thể chọn M lớn đến mức độ sâu của cây B sẽ là 1. Sau đó, bất kỳ tìm kiếm nào sau lần tìm kiếm đầu tiên sẽ chiếm một quyền truy cập đĩa, vì có lẽ nút gốc có thể được lưu trong bộ nhớ chính. Vấn đề với chiến lược này là hệ số phân nhánh quá cao nên cần phải xử lý đáng kể để xác định dữ liệu nằm trong lá nào

|  |  |  |  |
| --- | --- | --- | --- |
| 00 | 01 | 10 | 11 |

(2)

111000

111001

(2)

100000

101000

101100

101110

(2)

010100

011000

(2)

000100

001000

001010

001011

Hình 5.52 Hàm băm có thể mở rộng: dữ liệu gốc

Hiện tại, chúng ta hãy giả sử rằng dữ liệu của chúng ta bao gồm một số số nguyên 6 bit. Hình 5.52 cho thấy một lược đồ băm có thể mở rộng cho những dữ liệu này. Gốc của “cây” chứa bốn con trỏ được xác định bởi hai bit đầu của dữ liệu. Mỗi lá có tối đa M = 4 phần tử. Điều xảy ra là trong mỗi lá, hai bit đầu tiên giống hệt nhau; điều này được chỉ ra bởi số trong ngoặc đơn. Nói một cách chính thức hơn, D sẽ đại diện cho số bit được sử dụng bởi thư mục gốc, đôi khi được gọi là thư mục. Do đó, số lượng mục nhập trong thư mục là 2D. dL là số bit đầu tiên mà tất cả các phần tử của một số lá L có điểm chung. dL sẽ phụ thuộc vào lá cụ thể, và dL ≤ D.

Giả sử rằng chúng ta muốn chèn khóa 100100. Điều này sẽ đi vào lá thứ ba, nhưng vì lá thứ ba đã đầy nên không còn chỗ. Do đó, chúng tôi chia lá này thành hai lá, bây giờ được xác định bởi ba bit đầu tiên. Điều này đòi hỏi phải tăng kích thước thư mục lên 3. Những thay đổi này được trình bày lại trong Hình 5.53.

Lưu ý rằng tất cả các lá không liên quan đến sự phân chia hiện được trỏ đến bởi hai mục nhập thư mục liền kề. Vì vậy, mặc dù toàn bộ thư mục được viết lại, không có lá nào khác thực sự được truy cập.

Nếu khóa 000000 bây giờ được thêm vào, thì lá đầu tiên được tách ra, tạo ra hai lá với dL = 3. Vì D = 3, thay đổi duy nhất cần thiết trong thư mục là cập nhật các con trỏ 000 và 001. Xem Hình 5.54.

Chiến lược rất đơn giản này cung cấp thời gian truy cập nhanh chóng cho các hoạt động chèn và tìm kiếm trên cơ sở dữ liệu lớn. Có một vài chi tiết quan trọng mà chúng tôi chưa xem xét.

Đầu tiên, có thể yêu cầu một số phân tách thư mục nếu các phần tử trong một lá đồng ý trong nhiều hơn D + 1 bit đầu. Ví dụ, bắt đầu từ ví dụ ban đầu, với D = 2, nếu 111010, 111011 và ﬁ nally 111100 được chèn, kích thước thư mục phải được tăng lên 4 để phân biệt giữa các khóa ve. Đây là một chi tiết dễ chăm sóc, nhưng không được quên. Thứ hai, có khả năng trùng lặp các khóa; Nếu có nhiều hơn M bản sao, thì thuật toán này hoàn toàn không hoạt động. Trong trường hợp này, một số sắp xếp khác cần được thực hiện.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

(2)

111000

111001

(3)

101000

101100

101110

(3)

100000

100100

(2)

010100

011000

(2)

000100

001000

001010

001011

Hình 5.53 Hàm băm có thể mở rộng: sau khi chèn 100100 và chia thư mục

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

(2)

111000

111001

(3)

101000

101100

101110

(3)

100000

100100

(2)

010100

011000

(3)

001000

001010

001011

(3)

000000

000100

Hình 5.54 Băm mở rộng: sau khi chèn 000000 và tách lá

Những khả năng này cho thấy rằng điều quan trọng là các bit phải khá ngẫu nhiên. Điều này có thể được thực hiện bằng cách băm các khóa thành một số nguyên dài hợp lý — do đó có tên.

Số lượng lá dự kiến là (N / M) log e. Như vậy lá đầy đủ trung bình là ln2 = 0,69. Điều này cũng giống như đối với cây B, điều này không hoàn toàn đáng ngạc nhiên, vì đối với cả hai cấu trúc dữ liệu, các nút mới được tạo khi mục nhập thứ (M + 1) được thêm vào.

Kết quả đáng ngạc nhiên hơn là kích thước dự kiến của thư mục (nói cách khác, 2D) là O (N1 + 1 / M / M). Nếu M rất nhỏ, thì thư mục có thể lớn quá mức. Trong trường hợp này, chúng ta có thể có các lá chứa con trỏ đến các bản ghi thay vì các bản ghi thực, do đó làm tăng giá trị của M. Điều này thêm quyền truy cập đĩa thứ hai vào mỗi thao tác tìm kiếm để duy trì một thư mục nhỏ hơn. Nếu thư mục quá lớn không thể chứa được trong bộ nhớ chính, thì dù sao đi nữa cũng cần có quyền truy cập đĩa thứ hai.

Tóm lại

Bảng băm có thể được sử dụng để thực hiện chèn và chứa các hoạt động trong thời gian trung bình không đổi. Điều đặc biệt quan trọng là phải chú ý đến các chi tiết như hệ số tải khi sử dụng bảng băm, vì nếu không thì giới hạn thời gian không hợp lệ. Cũng cần lưu ý chọn hàm băm cẩn thận khi khóa không phải là một chuỗi ngắn hoặc số nguyên.

Đối với băm chuỗi riêng biệt, hệ số tải phải gần bằng 1, mặc dù perfor-mancedoesnotsigni ﬁ cantlydegradeunlesstheloadfactorbecomesverylarge. Đối với băm chuỗi, hệ số tải không được vượt quá 0,5, trừ khi điều này hoàn toàn không thể tránh khỏi. Nếu sử dụng phương pháp thăm dò tuyến tính, hiệu suất sẽ suy giảm nhanh chóng khi hệ số tải tiến tới 1. Có thể thực hiện băm lại để cho phép bảng phát triển (và thu nhỏ), do đó duy trì hệ số tải hợp lý. Điều này quan trọng nếu không gian chật hẹp và không thể chỉ khai báo một bảng băm lớn.

Cây tìm kiếm nhị phân cũng có thể được sử dụng để thực hiện các hoạt động chèn và chứa. Mặc dù giới hạn thời gian trung bình kết quả là O (logN), cây tìm kiếm nhị phân cũng hỗ trợ các quy trình yêu cầu thứ tự và do đó mạnh mẽ hơn. Sử dụng bảng băm, không thể tìm phần tử nhỏ nhất. Không thể tìm kiếm một cách rõ ràng một chuỗi trừ khi biết chuỗi chính xác. Một cây tìm kiếm nhị phân có thể nhanh chóng tìm ra tất cả các mục trong một phạm vi nhất định; điều này không được hỗ trợ bởi bảng băm. Hơn nữa, giới hạn O (logN) không nhất thiết phải nhiều hơn O (1), đặc biệt vì cây tìm kiếm không yêu cầu phép nhân hoặc phép chia.

Mặt khác, trường hợp xấu nhất đối với việc băm thường là do lỗi triển khai, trong khi đầu vào được sắp xếp có thể làm cho cây nhị phân hoạt động kém. Cây tìm kiếm cân bằng khá tốn kém để thực hiện, vì vậy nếu không yêu cầu thông tin sắp xếp và có bất kỳ nghi ngờ nào về việc đầu vào có thể được sắp xếp, thì băm là cấu trúc dữ liệu được lựa chọn.

Bảng băm hữu ích cho bất kỳ bài toán lý thuyết đồ thị nào trong đó các nút có tên thực thay vì số. Ở đây, khi đầu vào được đọc, các đỉnh được gán các số nguyên từ 1 trở đi theo thứ tự xuất hiện. Một lần nữa, đầu vào có thể có nhiều nhóm mục nhập được sắp xếp theo thứ tự bảng chữ cái. Ví dụ, các đỉnh có thể là máy tính. Sau đó, nếu một cài đặt cụ thể liệt kê các máy tính của nó là ibm1, ibm2, ibm3, ..., có thể có một ảnh hưởng đáng kể đến hiệu suất nếu cây tìm kiếm được sử dụng.

Tuy nhiên, một công dụng khác của băm là trong các trình kiểm tra chính tả trực tuyến. Nếu việc phát hiện lỗi chính tả (thay vì sửa) là quan trọng, thì toàn bộ từ điển có thể được băm sẵn và các từ có thể được kiểm tra trong thời gian liên tục. Bảng băm rất thích hợp cho việc này, bởi vì việc sắp xếp theo thứ tự bảng chữ cái không quan trọng; việc in ra các lỗi chính tả theo thứ tự chúng xuất hiện trong tài liệu chắc chắn có thể chấp nhận được.

Bảng băm thường được sử dụng để triển khai bộ nhớ đệm, cả trong phần mềm (ví dụ: bộ đệm trong trình duyệt Internet của bạn) và trong phần cứng (ví dụ: bộ nhớ đệm trong máy tính hiện đại). Chúng cũng được sử dụng trong việc triển khai phần cứng của bộ định tuyến.

CHƯƠNG 4. TREES (cây)

Đối với một lượng lớn đầu vào, thời gian truy cập tuyến tính của danh sách được liên kết là cấm. Trong chương này, chúng ta xem xét một cấu trúc dữ liệu đơn giản mà thời gian chạy trung bình của hầu hết các oper-ations là O(logN). Chúng tôi cũng phác thảo một sửa đổi khái niệm đơn giản để cấu trúc dữ liệu này đảm bảo thời gian trên bị ràng buộc trong trường hợp xấu nhất và thảo luận về một modifica-tion thứ hai mà về cơ bản cung cấp cho một O (logN) thời gian chạy cho mỗi hoạt động cho một chuỗi dài các hướng dẫn.

Cấu trúc dữ liệu mà chúng ta đang đề cập đến được gọi là cây tìm kiếm nhị phân. Cây tìm kiếm nhị phân là cơ sở để thực hiện hai lớp tập hợp thư viện, tập hợp và ánh xạ, được sử dụng trong nhiều ứng dụng. Cây cối nói chung là những yếu tố trừu tượng rất hữu ích trong khoa học máy tính, vì vậy chúng ta sẽ thảo luận về việc sử dụng chúng trong các ứng dụng khác, tổng quát hơn. Trong chương này, chúng ta sẽ ...

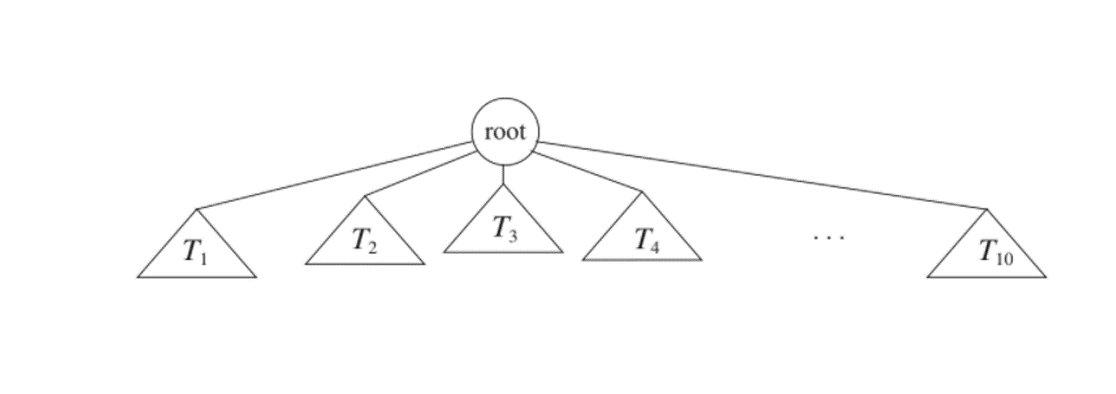
* Xem cách cây được sử dụng để triển khai hệ thống tệp của một số hệ điều hành phổ biến.
* Xem cách cây có thể được sử dụng để đánh giá các biểu thức số học.
* Trình bày cách sử dụng cây để hỗ trợ hoạt động tìm kiếm trong thời gian trung bình O (logN) và cách tinh chỉnh những ý tưởng này để có được giới hạn trong trường hợp xấu nhất O (logN). Chúng tôi cũng sẽ xem cách triển khai các hoạt động này khi dữ liệu được lưu trữ trên đĩa.
* Thảo luận và sử dụng các lớp tập hợp và bản đồ.

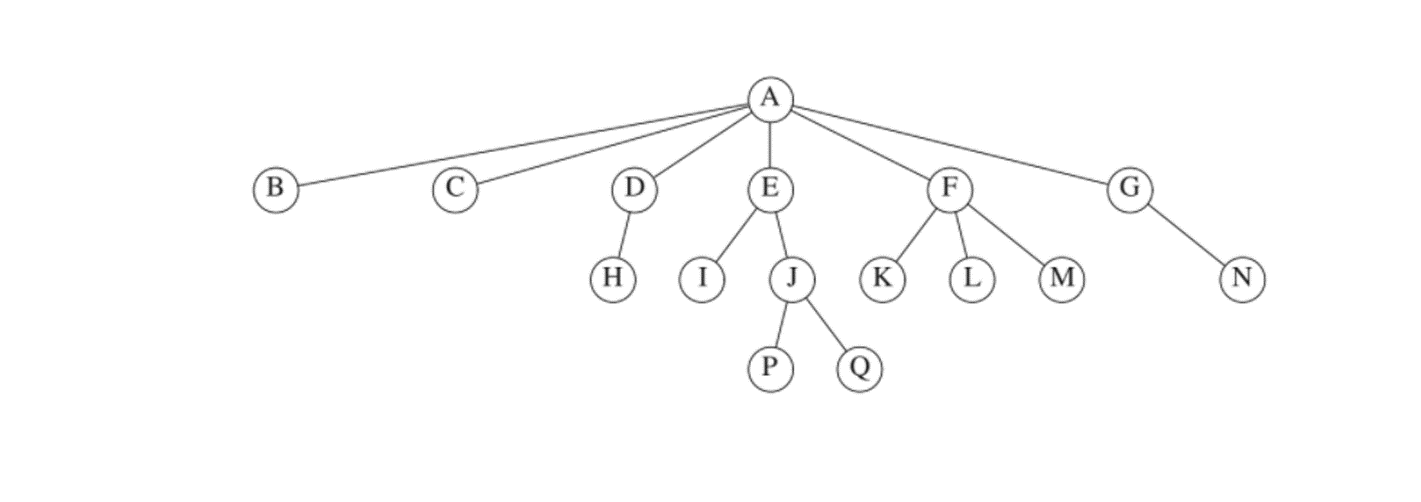
**4.1 Preliminaries** (Sơ bộ)

Một cây có thể được định nghĩa theo nhiều cách. Một cách tự nhiên để xác định cây là đệ quy. Cây là một tập hợp các nút. Bộ sưu tập có thể trống; nếu không, một cây bao gồm một nút không bị hạn chế, r, được gọi là gốc, và không hoặc nhiều cây khác (con) T1, T2, ..., Tk, mỗi cây có gốc được nối với nhau bằng một cạnh có hướng từ r.

Gốc của mỗi cây con được cho là con của r, và r là gốc của mỗi gốc cây con. Hình 4.1 cho thấy một cây điển hình sử dụng định nghĩa đệ quy.

Từ định nghĩa đệ quy, chúng ta thấy rằng một cây là tập hợp của N nút, một trong số đó là gốc và N - 1 cạnh. Có N - 1 cạnh xuất phát từ thực tế là mỗi cạnh nối một số nút với nút cha của nó và mọi nút ngoại trừ gốc đều có một nút cha (xem Hình 4.2).

**Hình 4.1:** Cây chung (Generic tree)

**Hình 4.2:** Cấu tạo của một cây (A tree)

Trong cây của Hình 4.2, gốc là A. Nút F có A là cha và K, L và M là con. Mỗi nút có thể có một số nút con tùy ý, có thể bằng không. Các nút không có con được gọi là các **lá**; các lá ở cây trên là B, C, H, I, P, Q, K, L, M, N. Có cùng cha mẹ là **anh chị em ruột**; do đó, K, L và M đều là anh em ruột. Quan hệ **ông bà** và **cháu** có thể được định nghĩa theo cách tương tự.

**Đường** dẫn từ nút n1 đến nk được định nghĩa là một chuỗi các nút n1, n2, ..., nk sao cho ni là cha của ni + 1 với 1 ≤ i <k. **Độ dài** của đường dẫn này là số cạnh trên đường đi, cụ thể là k - 1. Có một đường đi có độ dài bằng 0 từ mọi nút đến chính nó. Chú ý rằng trong cây có chính xác một đường dẫn từ gốc đến mỗi nút.

Đối với bất kỳ nút ni nào, **độ sâu** của ni là độ dài của đường đi duy nhất từ gốc đến ni. Như vậy, rễ ở độ sâu 0. **Chiều cao** của ni là độ dài của con đường dài nhất từ ni đến lá. Như vậy tất cả các lá đều cao bằng 0. Chiều cao của cây bằng chiều cao của rễ. Đối với cây trong Hình 4.2, E ở độ sâu 1 và chiều cao 2; F ở độ sâu 1 và độ cao 1; chiều cao của cây là 3. Chiều sâu của cây bằng chiều sâu của lá sâu nhất; điều này luôn luôn bằng chiều cao của cây.

Nếu có một đường đi từ n1 đến n2, thì n1 là **tổ tiên** của n2 và n2 là **hậu duệ** của n1. Nếu n1 = n2, thì n1 là **tổ tiên** của n2 và n2 là **hậu duệ** của n1.

**4.1.1 Implementation of trees** (Thực hiện cây)

Một cách để triển khai cây là có trong mỗi nút, bên cạnh dữ liệu của nó, một liên kết đến mỗi nút con của nút. Tuy nhiên, vì số lượng nút con trên mỗi nút có thể khác nhau rất nhiều và không được biết trước, nên có thể không thực hiện được việc tạo liên kết trực tiếp các nút con trong dữ liệu

1 struct TreeNode

2 {

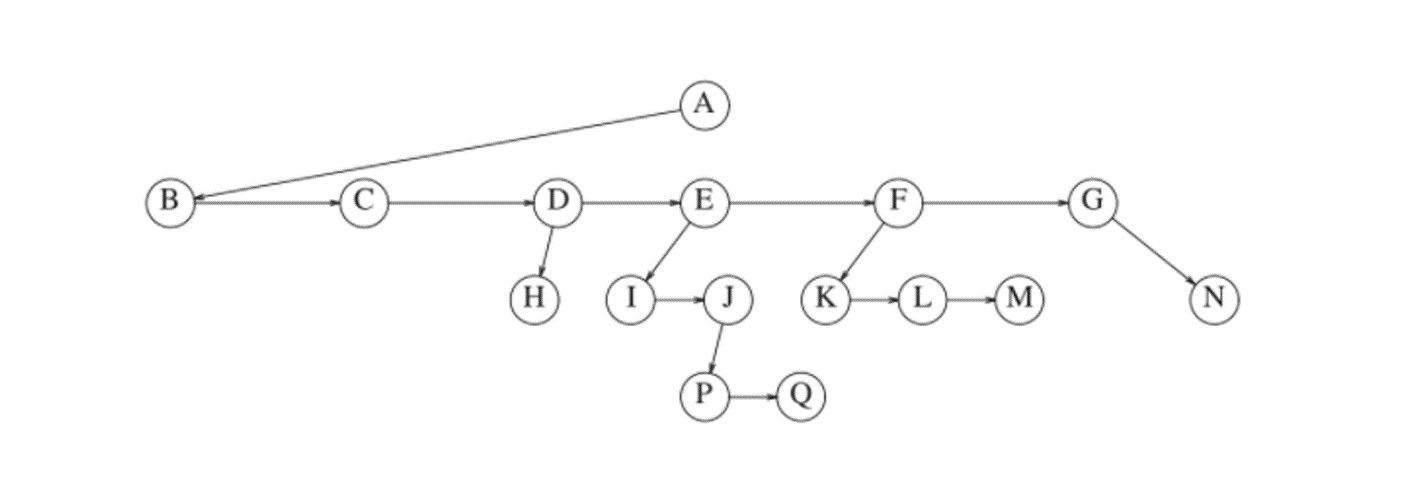
3 0bject element;

4 TreeNode \*firstChild;

5 TreeNode \*nextSibling;

6 };

**Hình 4.3:** Khai báo nút cho cây (Node declarations for trees)

**Hình 4.4:** Biểu diễn con đầu tiên / anh chị em kế tiếp của cây được thể hiện trong Hình 4.2

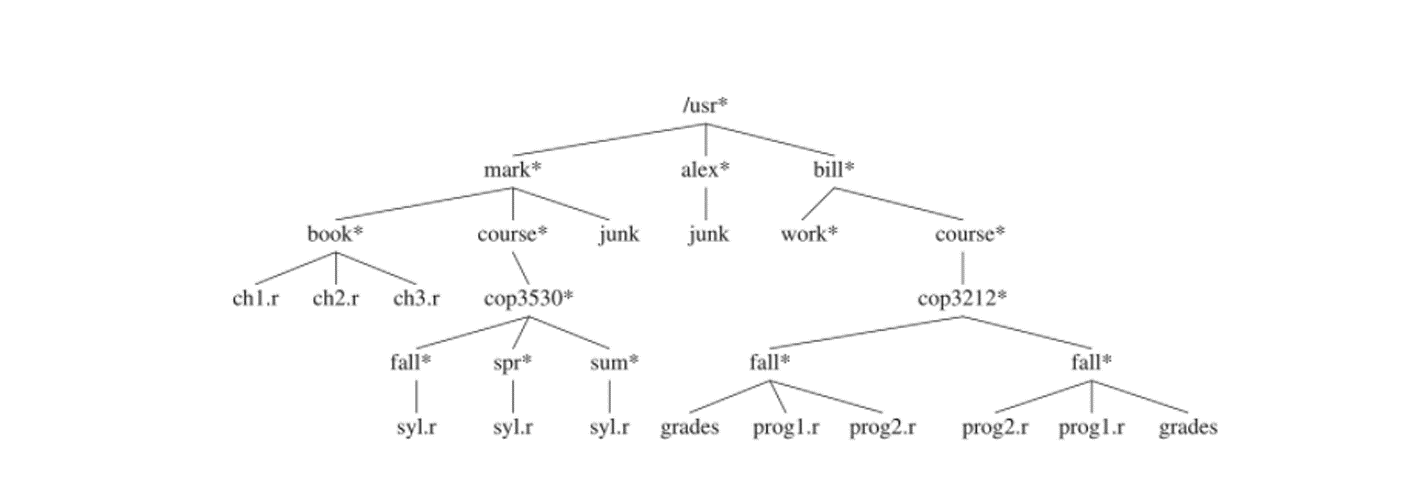
Cấu trúc, bởi vì sẽ có quá nhiều không gian lãng phí. Giải pháp rất đơn giản: Giữ các nút con của mỗi nút trong danh sách liên kết các nút cây. Khai báo trong Hình 4.3 là điển hình.

Hình 4.4 cho thấy một cây có thể được biểu diễn như thế nào trong việc triển khai này. Các mũi tên ngang hướng xuống dưới là liên kết FirstChild. Các mũi tên đi từ trái sang phải là các liên kết nextSibling. Các liên kết rỗng không được vẽ, vì có quá nhiều.

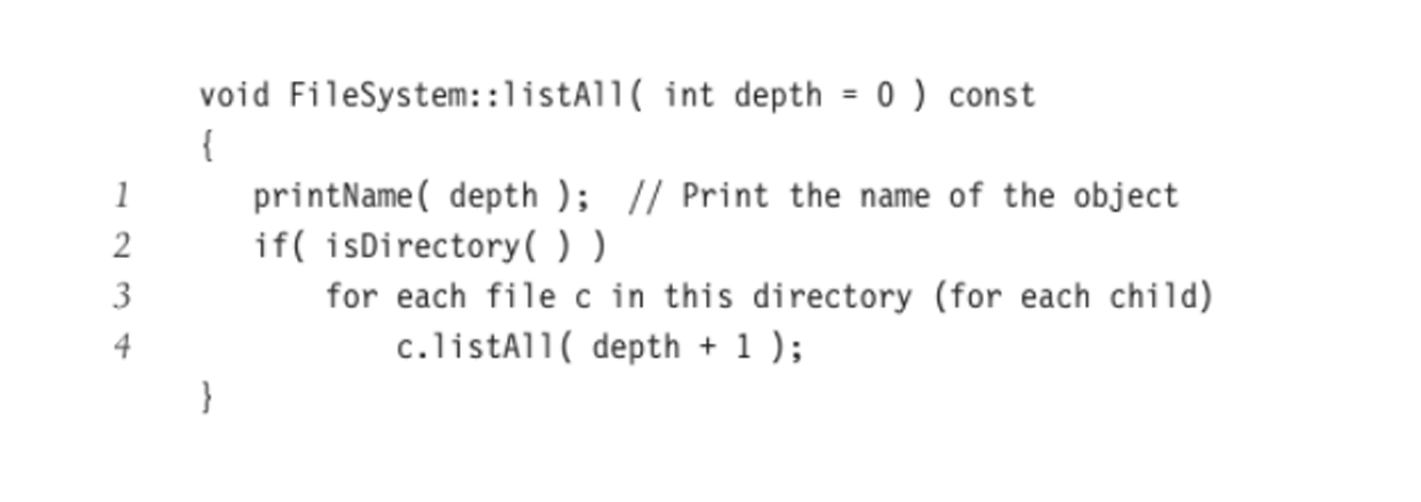
Trong cây của Hình 4.4, nút E có cả liên kết đến anh chị em (F) và liên kết đến con (I), trong khi một số nút không có.

**4.1.2 Tree Traversals with an Application** (Cây Traversals với một ứng dụng)

Có rất nhiều ứng dụng cho cây. Một trong những cách sử dụng phổ biến là cấu trúc thư mục trong nhiều hệ điều hành phổ biến, bao gồm UNIX và DOS. Hình 4.5 là một thư mục điển hình trong hệ thống tệp UNIX.

 Thư mục gốc của thư mục này là / usr. (Dấu hoa thị bên cạnh tên cho biết bản thân / usr là một thư mục.) / Usr có ba phần tử con là mark, alex và bill, là các thư mục của chúng. Do đó, / usr chứa ba thư mục và không có tệp thông thường. Tên tệp / usr / mark / book / ch1.r có được bằng cách theo sau con ngoài cùng bên trái ba lần. Mỗi / sau cái đầu tiên chỉ một cạnh; kết quả là tên đường dẫn đầy đủ. Hệ thống tệp phân cấp này rất phổ biến vì nó cho phép người dùng tổ chức dữ liệu của họ một cách hợp lý. Hơn nữa, hai tệp trong các thư mục khác nhau có thể dùng chung tên, bởi vì chúng phải có các đường dẫn khác nhau từ gốc và do đó có các **tên đường dẫn** khác nhau. Một thư mục trong hệ thống tệp UNIX chỉ là một tệp có danh sách tất cả các phần tử con của nó, do đó, các thư mục được cấu trúc gần như chính xác theo.

**Hình 4.5:** Thư mục UNIX (UNIX directory)

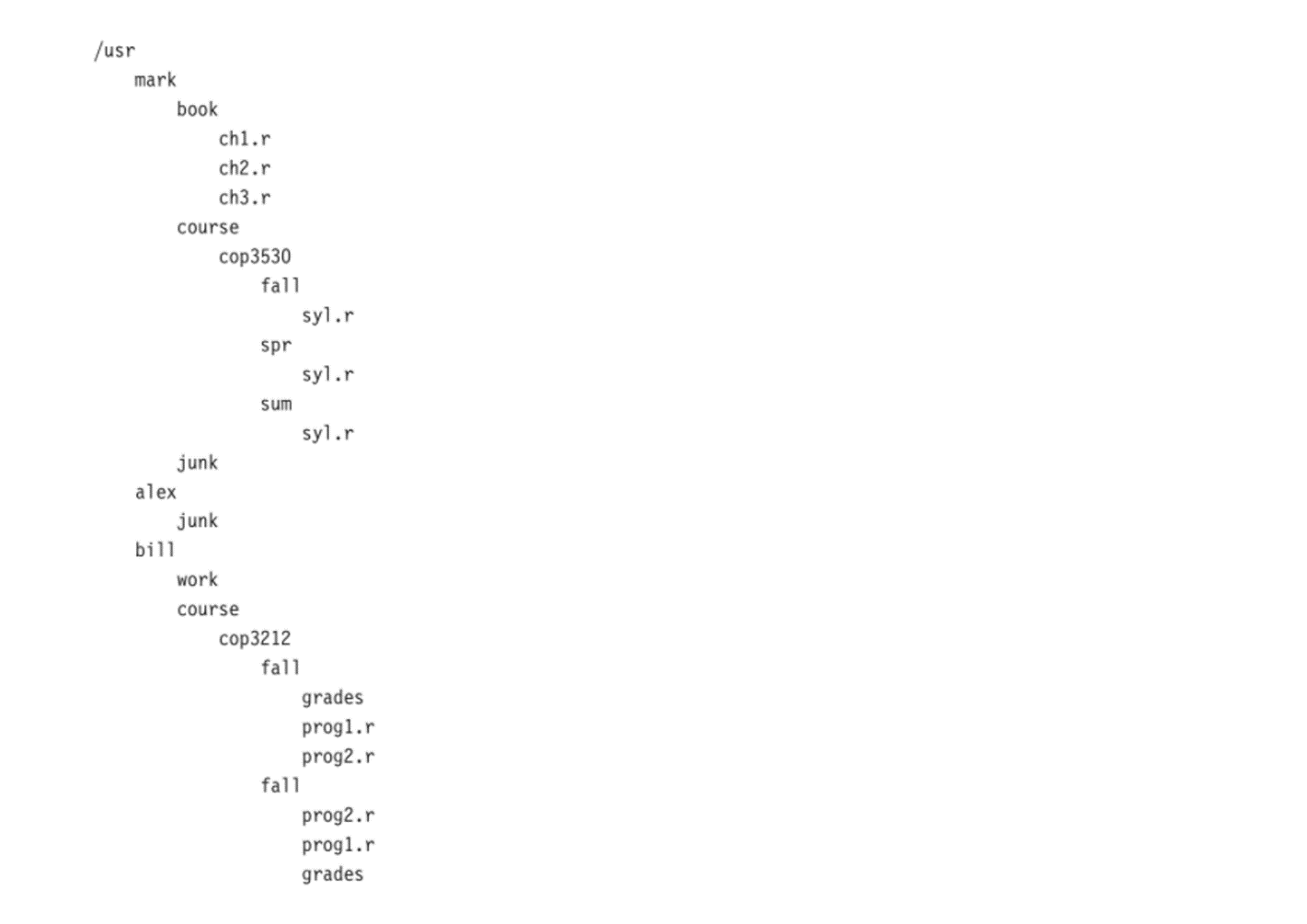
**Hình 4.6** Mã giả để liệt kê một thư mục trong hệ thống tệp thuê bao

Với khai báo kiểu ở trên.1 Thật vậy, trên một số phiên bản UNIX, nếu lệnh com-mand thông thường để in tệp được áp dụng cho một thư mục, thì tên của các tệp trong thư mục có thể được nhìn thấy trong đầu ra (cùng với các thông tin không phải ASCII).

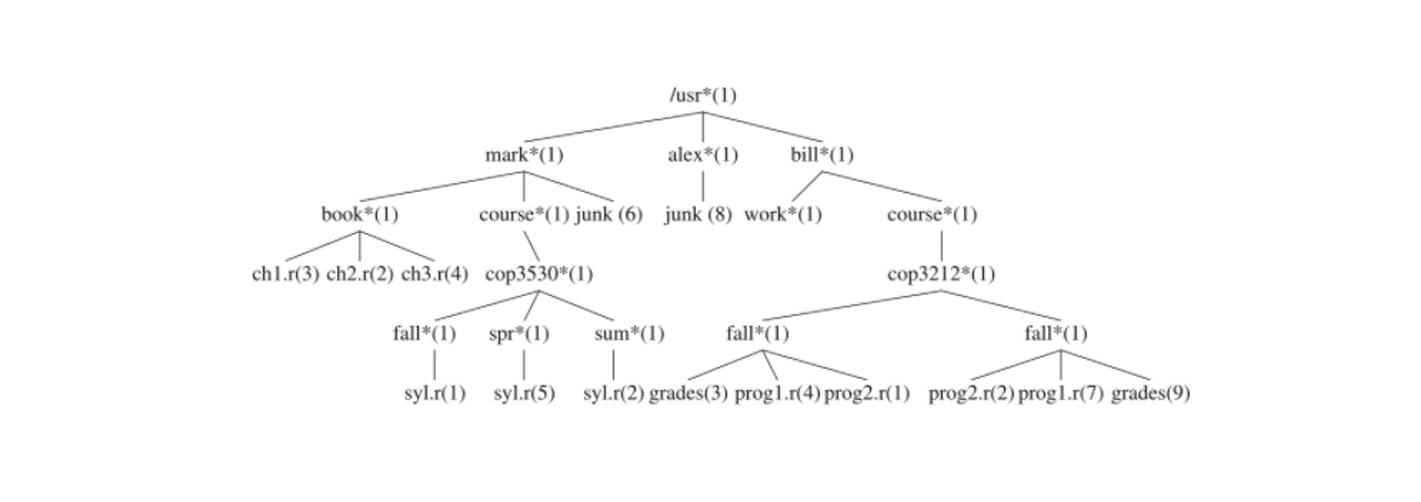
Giả sử chúng ta muốn liệt kê tên của tất cả các tệp trong thư mục. Định dạng đầu ra của chúng tôi sẽ là các tệp có độ sâu sẽ có tên của chúng được thụt vào trong các tab di. Thuật toán của chúng tôi được đưa ra trong Hình 4.6 dưới dạng mã giả.

Danh sách hàm đệ quy Tất cả cần được bắt đầu với độ sâu 0 để biểu thị không thụt lề cho gốc. Độ sâu này là một biến số ghi sổ kế toán nội bộ và hầu như không phải là một tham số mà một thói quen gọi điện cần phải biết. Do đó, giá trị mặc định của 0 được cung cấp cho độ sâu.

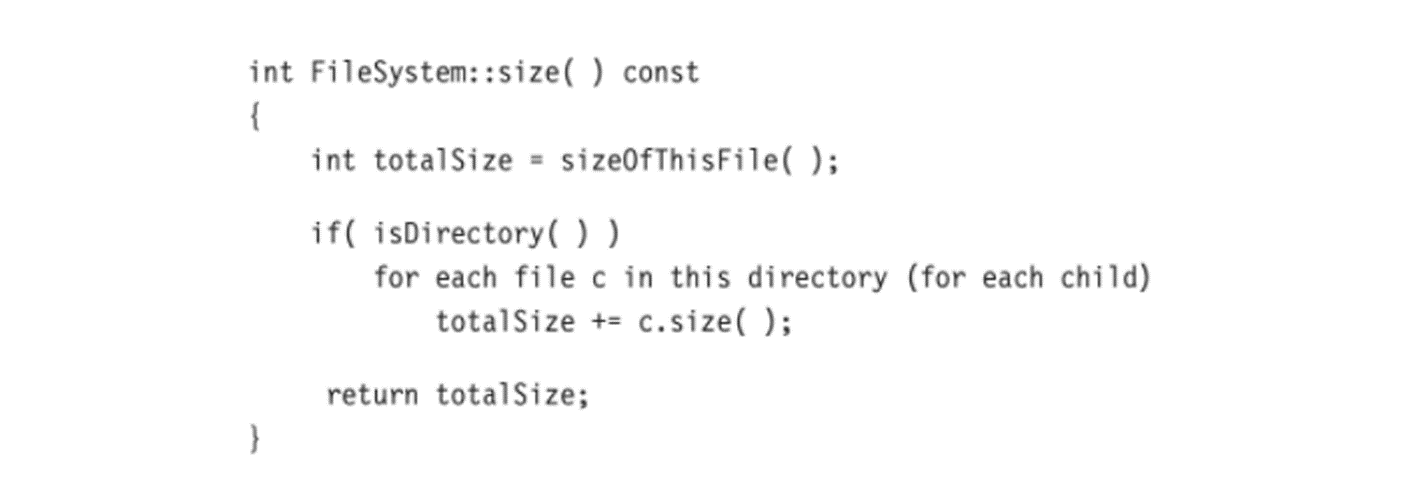
Logic của thuật toán rất đơn giản để tuân theo. Tên của đối tượng tệp được in ra với số lượng tab thích hợp. Nếu mục nhập là một thư mục, thì chúng tôi xử lý tất cả các con một cách đệ quy, từng cái một. Những trẻ này sâu hơn một cấp, và do đó cần phải được thụt vào một khoảng trống. Kết quả là trong Hình 4.7.

**Hình 4.7:** Danh sách thư mục (đặt hàng trước)

node. Hơn nữa, dòng 4 có thể được thực thi nhiều nhất một lần cho mỗi nút con của mỗi nút. Nhưng số con ít hơn số nút đúng một con. Cuối cùng, vòng lặp for lặp lại một lần mỗi lần thực hiện dòng 4 cộng với một lần mỗi khi vòng lặp kết thúc. Do đó, tổng số lượng công việc là không đổi trên mỗi nút. Nếu có N tên tệp được xuất, thì thời gian chạy là O (N).

Vì các thư mục chính là tệp, chúng cũng có kích thước. Giả sử chúng ta muốn tính tổng số khối được sử dụng bởi tất cả các tệp trong cây. Cách tự nhiên nhất để làm điều này là tìm số khối chứa trong các thư mục con / usr / mark (30), / usr / alex (9) và / usr / bill (32). Tổng số khối sau đó là tổng số trong

**Hình 4.8:** Thư mục UNIX với kích thước tệp thu được thông qua trình duyệt bưu điện



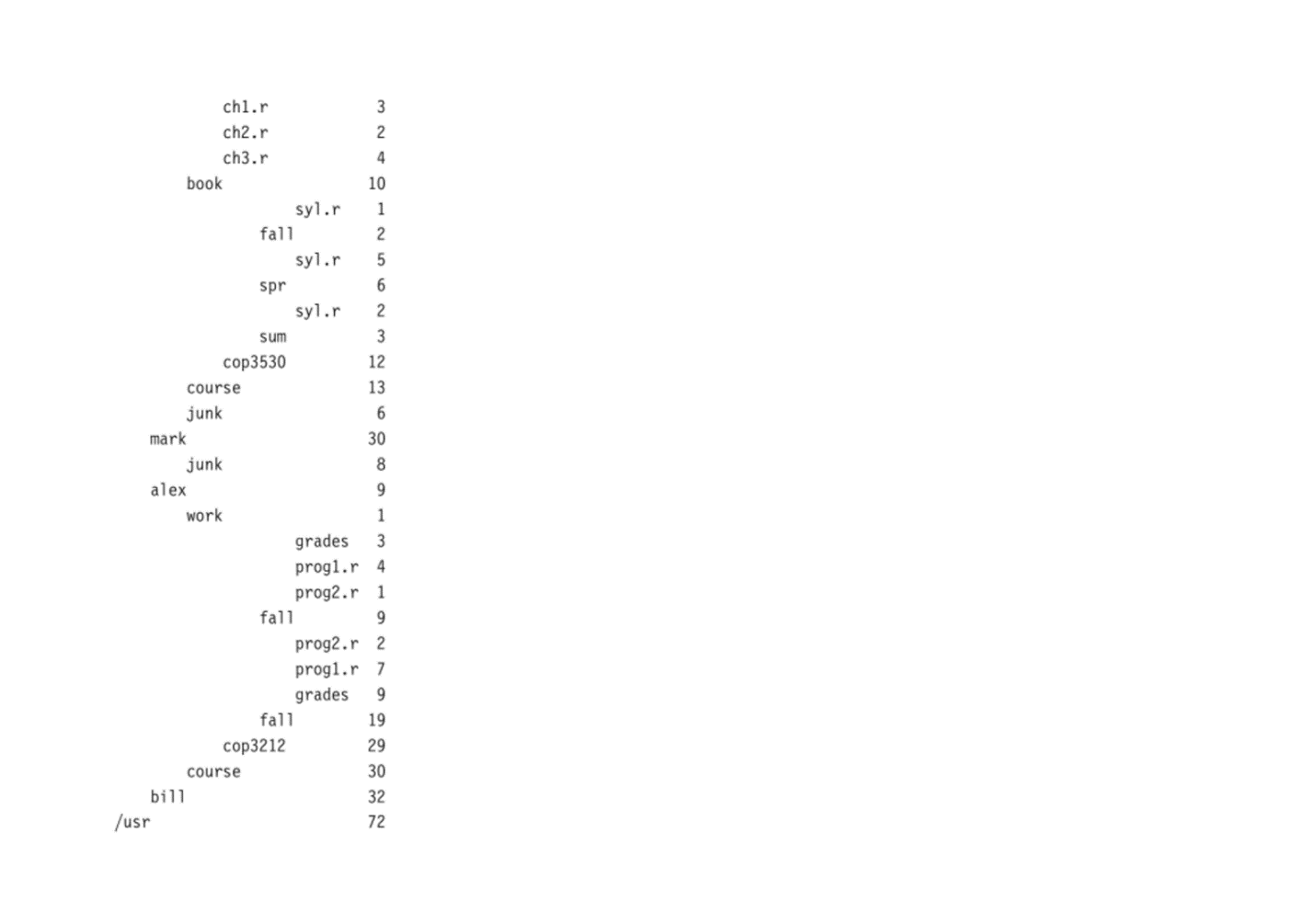
**Hình 4.9:** Mã giả để tính toán kích thước của một thư mục

Thư mục con (71) cộng với một khối được sử dụng bởi / usr, tổng cộng là 72. Kích thước phương pháp mã giả trong Hình 4.9 thực hiện chiến lược này.

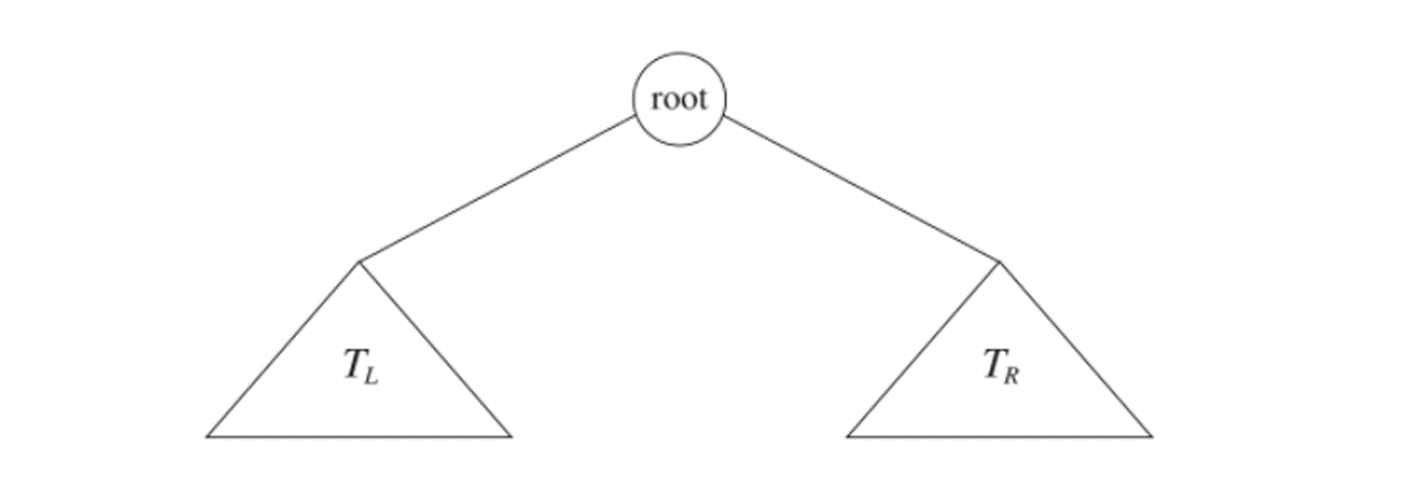
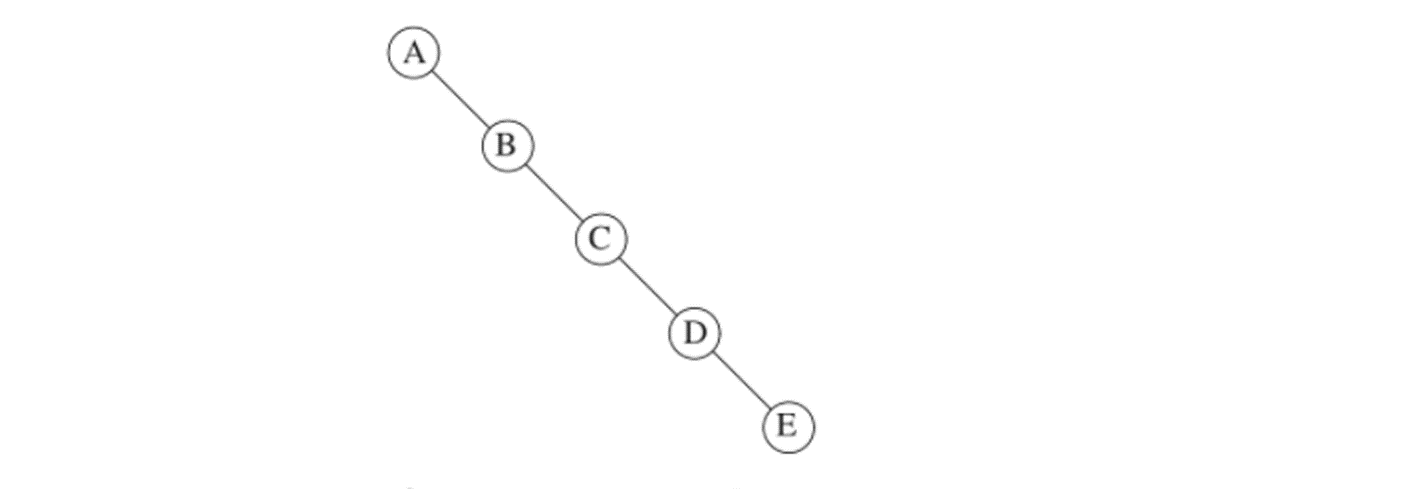
**4.2 Binary Trees** (Cây nhị phân)

Cây nhị phân là một cây trong đó không có nút nào có thể có nhiều hơn hai nút con.

Hình 4.11 cho thấy rằng một cây nhị phân bao gồm một gốc và hai cây con, TL và TR, cả hai đều có thể trống.

 Một thuộc tính của cây nhị phân đôi khi rất quan trọng là độ sâu của cây nhị phân trung bình nhỏ hơn đáng kể so với N. Một phân tích cho thấy độ sâu trung bình là O (√N) và đó là đối với một loại cây nhị phân đặc biệt, cụ thể là cây tìm kiếm nhị phân, giá trị trung bình của độ sâu là O (logN). Thật không may, độ sâu có thể lớn đến N - 1, như ví dụ trong Hình 4.12 cho thấy.

**Hình 4.10:** Dấu vết của hàm kích thước

**Hình 4.11:** Cây nhị phân chung

**Hình 4.12:** Cây nhị phân trường hợp xấu nhất

**4.2.1 Implementation** (Thực hiện)

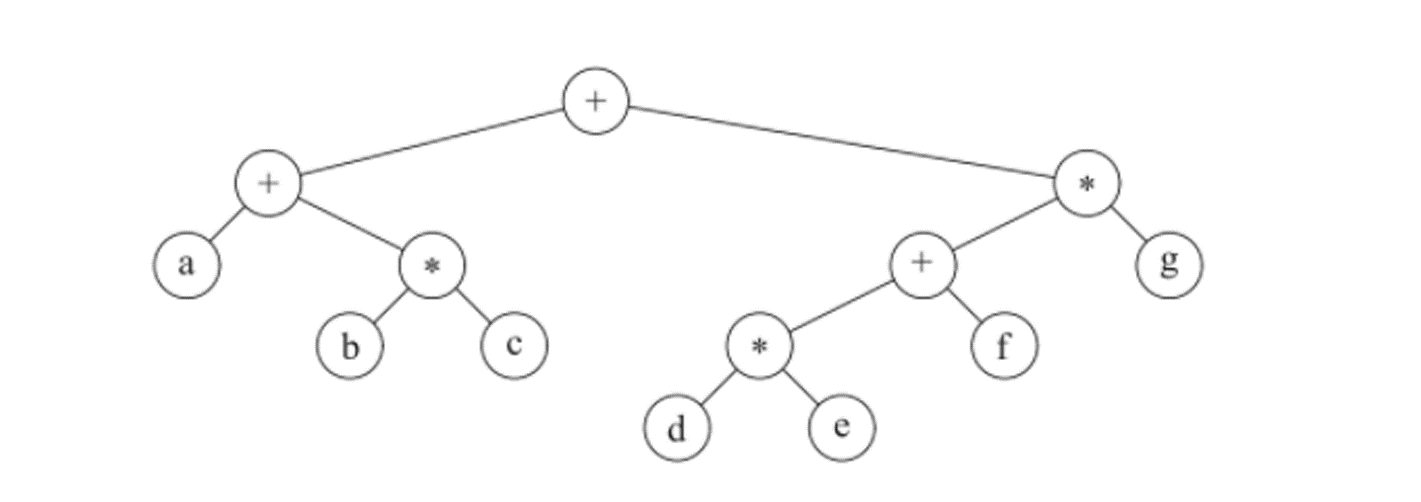
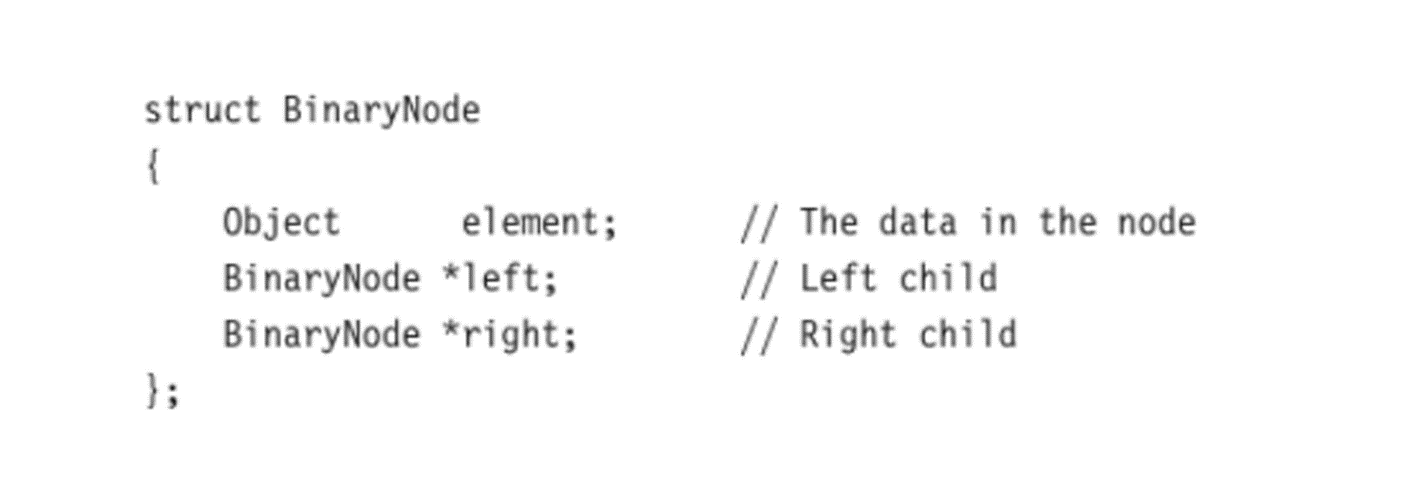
Bởi vì một nút cây nhị phân có nhiều nhất hai nút con, chúng ta có thể giữ các liên kết trực tiếp đến chúng. Việc khai báo các nút cây có cấu trúc tương tự như đối với danh sách liên kết kép, trong đó một nút là cấu trúc bao gồm thông tin phần tử cộng với hai con trỏ (trái và phải) đến các nút khác (xem Hình 4.13).

Chúng ta có thể vẽ cây nhị phân bằng cách sử dụng các hộp hình chữ nhật thường dùng cho danh sách được liên kết, nhưng cây thường được vẽ dưới dạng các vòng tròn nối với nhau bằng các đường, vì chúng thực sự là đồ thị. Chúng tôi cũng không vẽ rõ ràng các liên kết nullptr khi tham chiếu đến cây, bởi vì mọi cây nhị phân có N nút sẽ yêu cầu N + 1 liên kết nullptr.

Cây nhị phân có nhiều công dụng quan trọng không gắn với việc tìm kiếm. Một trong những ứng dụng chính của cây nhị phân là trong lĩnh vực thiết kế trình biên dịch, bây giờ chúng ta sẽ khám phá.

**4.2.2 An Example: Expression Trees** (Ví dụ: Cây biểu hiện)

Hình 4.14 cho thấy một ví dụ về cây biểu thức. Các lá của cây biểu thức là các toán hạng, chẳng hạn như hằng số hoặc tên biến và các nút khác chứa các toán tử. Cây cụ thể này thường là nhị phân, vì tất cả các toán tử đều là nhị phân, và mặc dù đây là trường hợp đơn giản nhất, các nút có thể có nhiều hơn hai con. Cũng có thể một nút chỉ có một nút con, như trường hợp của toán tử trừ một bậc. Chúng ta có thể đánh giá một cây biểu thức, T, bằng cách áp dụng toán tử ở gốc cho các giá trị

**Hình 4.13:** Lớp nút cây nhị phân (mã giả)

**Hình 4.14:** Cây biểu thức cho (a + b \* c) + ((d \* e + f) \* g)

Thu được bằng cách đánh giá đệ quy các cây con trái và phải. Trong ví dụ của chúng ta, cây con bên trái đánh giá là a + (b \* c) và cây con bên phải đánh giá là ((d \* e) + f) \* g. Do đó, toàn bộ cây đại diện cho (a + (b \* c)) + (((d \* e) + f) \* g).

Chúng ta có thể tạo ra một biểu thức infix (quá nhiều dấu ngoặc đơn) bằng cách tạo ra một cách đệ quy một biểu thức bên trái được đặt trong ngoặc đơn, sau đó in ra toán tử ở gốc và cuối cùng tạo ra một biểu thức bên phải được đặt trong ngoặc đơn. Chiến lược chung này (trái, nút, phải) được biết đến như là một phương án nhỏ hơn; nó dễ nhớ vì kiểu biểu thức mà nó tạo ra.

Một chiến lược duyệt thay thế là in ra đệ quy cây con bên trái, cây con bên phải và sau đó là toán tử. Nếu chúng ta áp dụng chiến lược này cho cây của chúng ta ở trên, kết quả đầu ra là abc \* + de \* f + g \* +, có thể dễ dàng xem là biểu diễn hậu tố của Phần 3.6.3. Chiến lược truyền tải này thường được biết đến như một chiến lược trình tự sau. Chúng tôi đã thấy điều này

chiến lược đi ngang trước đó trong Phần 4.1.

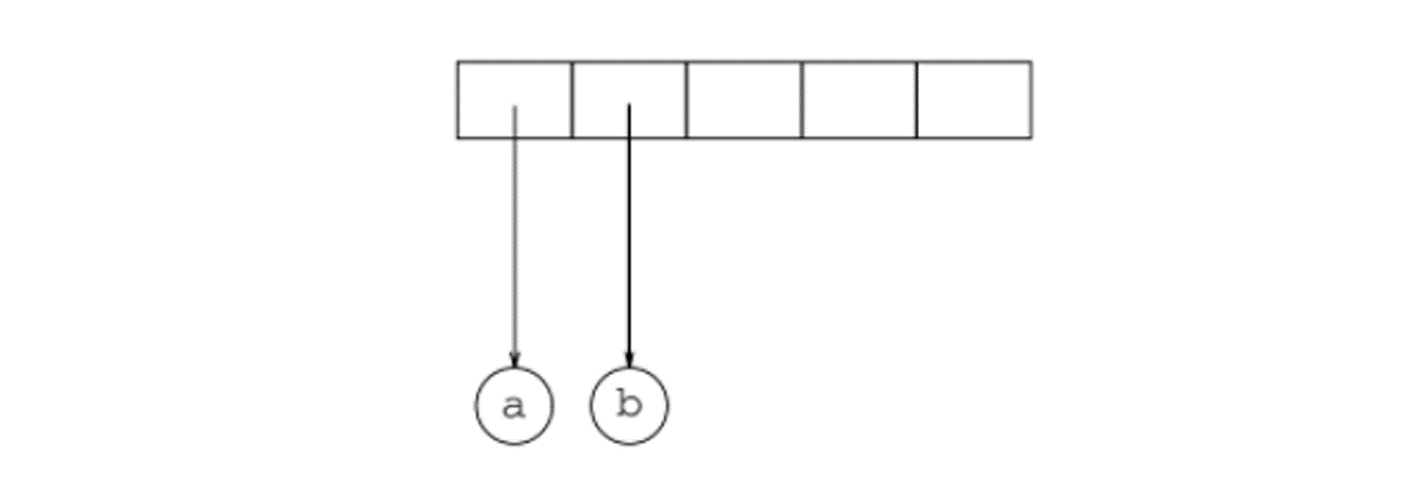
Chiến lược duyệt thứ ba là in ra toán tử đầu tiên và sau đó in ra đệ quy các cây con trái và phải. Biểu thức kết quả, ++ a \* b c \* + \* d e f g, là ký hiệu tiền tố kém hữu ích hơn và chiến lược truyền tải là một phương thức truyền tải có thứ tự trước, chúng ta cũng đã thấy trước đó trong Phần 4.1. Chúng ta sẽ quay trở lại các chiến lược truyền tải này ở phần sau của chương.

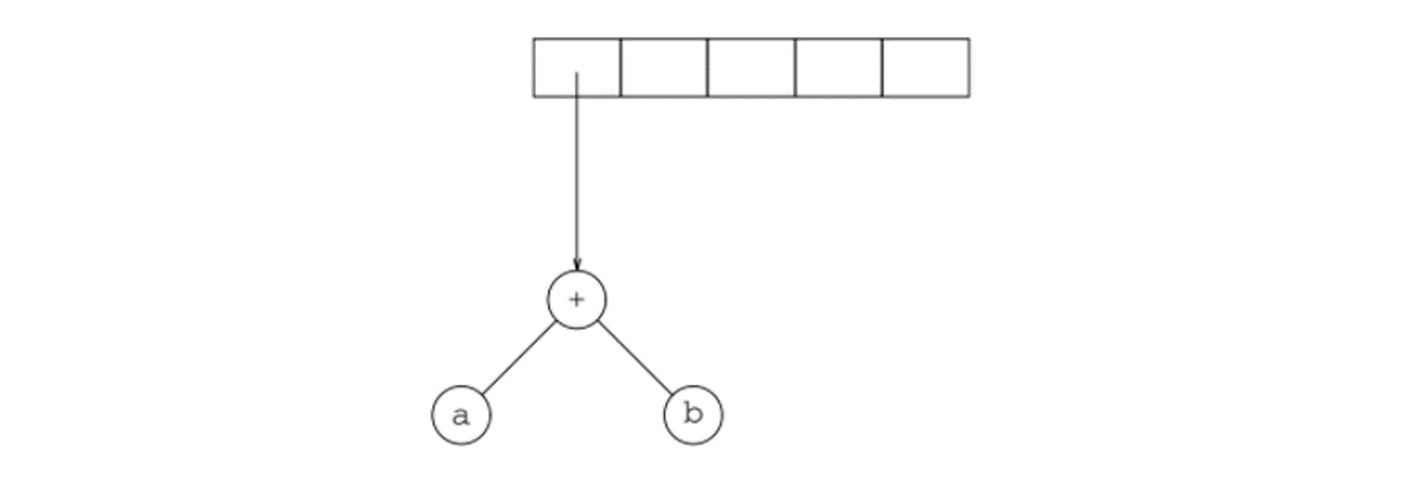
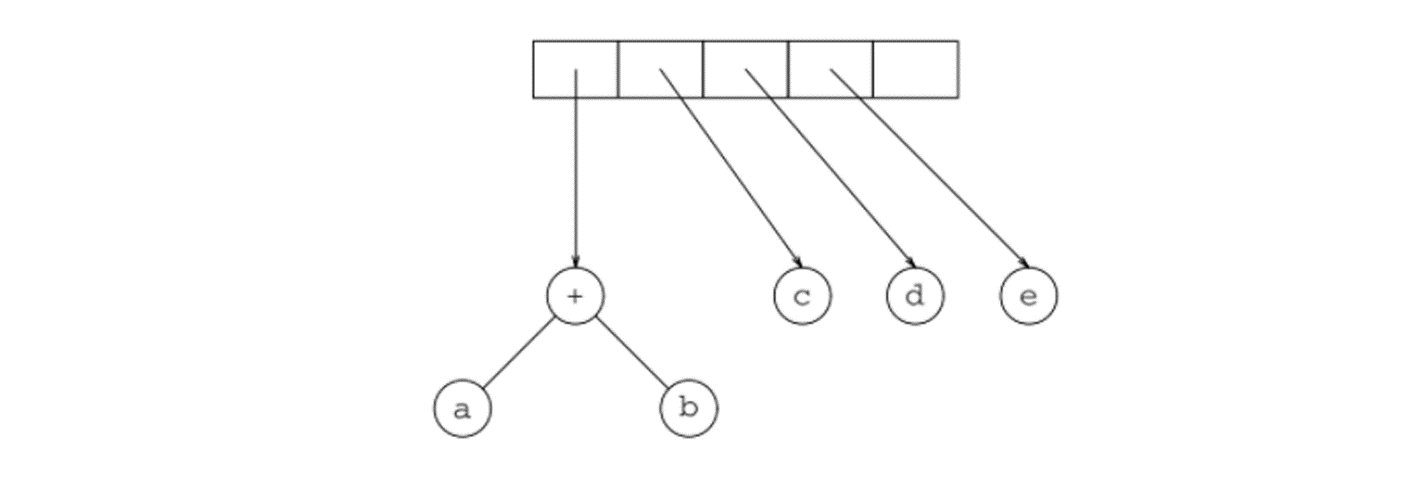
***Xây dựng cây biểu thức***

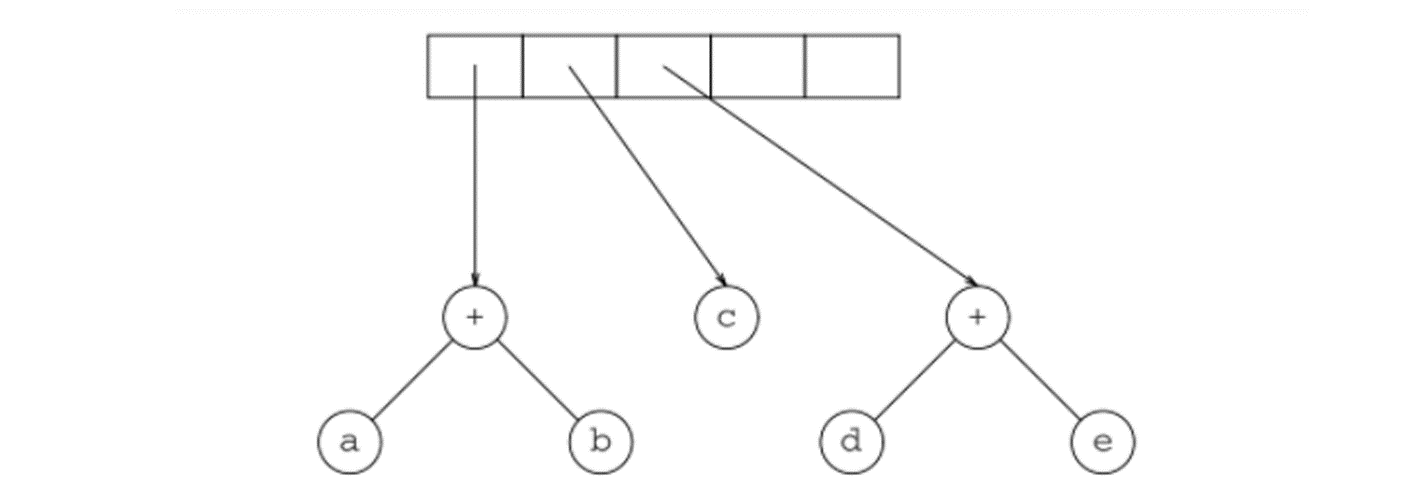
Bây giờ chúng ta đưa ra một thuật toán để chuyển đổi một biểu thức hậu tố thành một cây biểu thức. Vì chúng ta đã có một thuật toán để chuyển đổi infix thành postfix, chúng ta có thể tạo cây biểu thức từ hai loại đầu vào phổ biến. Phương pháp mà chúng tôi mô tả rất giống với thuật toán đánh giá hậu tố của Phần 3.6.3. Chúng tôi đọc biểu tượng của chúng tôi từng biểu tượng một. Nếu biểu tượng là một toán hạng, chúng ta tạo cây một nút và đẩy một con trỏ đến nó vào một ngăn xếp. Nếu biểu tượng là một toán tử, chúng tôi bật (con trỏ) đến hai cây T1 và T2 từ ngăn xếp (T1 được xuất hiện trước) và tạo thành một cây mới có gốc là toán tử và có con trái và phải trỏ tới T2 và T1 tương ứng. . Một con trỏ đến cây mới này sau đó được đẩy lên ngăn xếp.

Ví dụ, giả sử đầu vào là

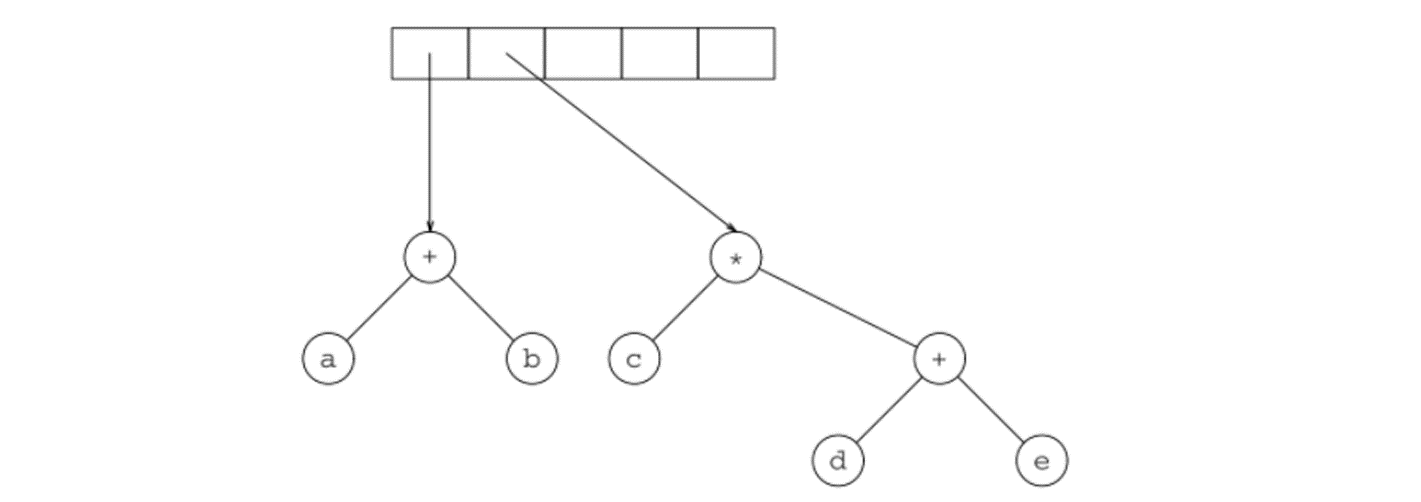
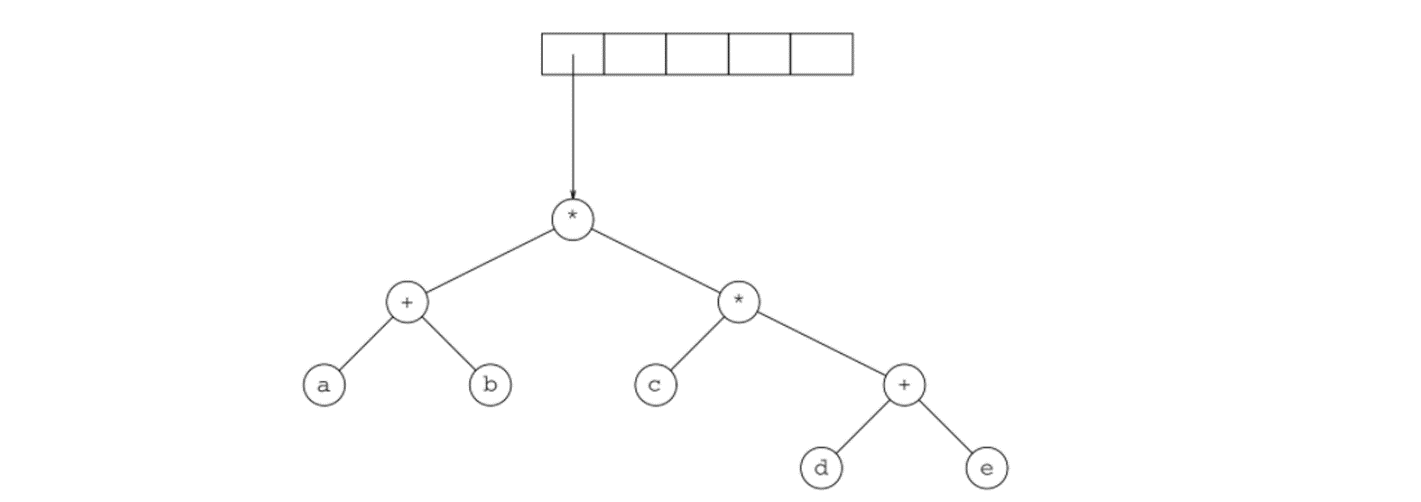
a b + c d e + \* \*

 Hai biểu tượng đầu tiên là các toán hạng, vì vậy chúng ta tạo cây một nút và đẩy các con trỏ tới chúng vào một ngăn xếp.

Tiếp theo, dấu + được đọc, vì vậy hai con trỏ đến cây được xuất hiện, một cây mới được hình thành và một con trỏ tới nó được đẩy lên ngăn xếp.Tiếp theo, c, d và e được đọc và đối với mỗi cây một nút được tạo và một con trỏ đến cây tương ứng được đẩy lên ngăn xếp. 

****Bây giờ dấu + được đọc, vì vậy hai cây được hợp nhất.

Tiếp tục, dấu \* được đọc, vì vậy chúng ta bật hai con trỏ cây và tạo một cây mới với dấu \* là gốc.

Cuối cùng, ký hiệu cuối cùng được đọc, hai cây được hợp nhất và một con trỏ đến cây cuối cùng được để lại trên ngăn xếp.

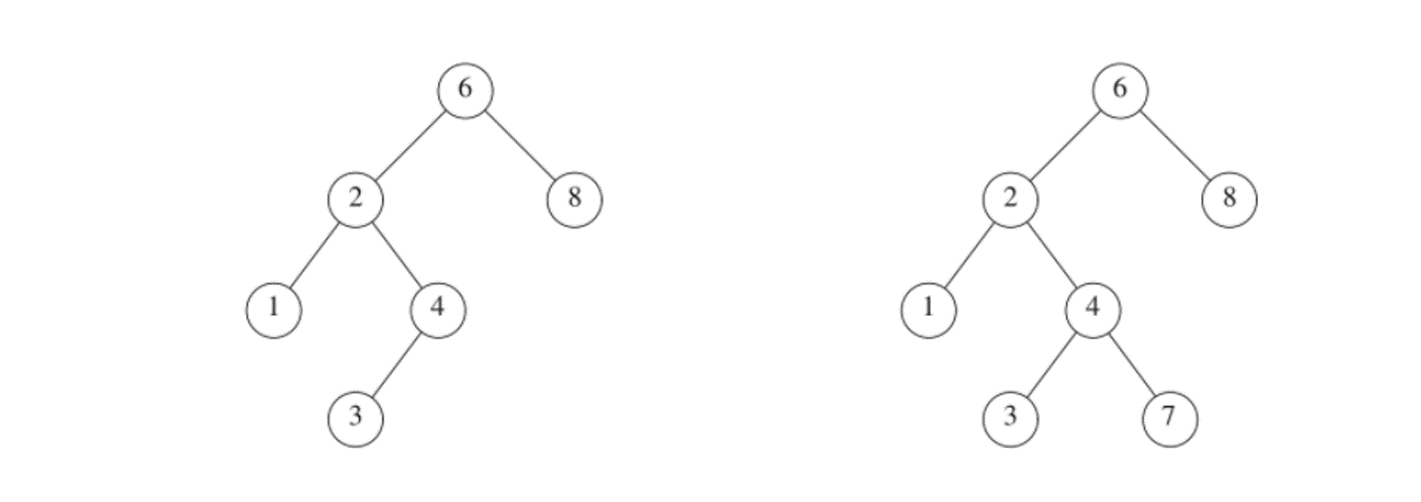
**4.3 The Search Tree ADT – Binary Search Trees** (Cây tìm kiếm ADT - Cây tìm kiếm nhị phân)

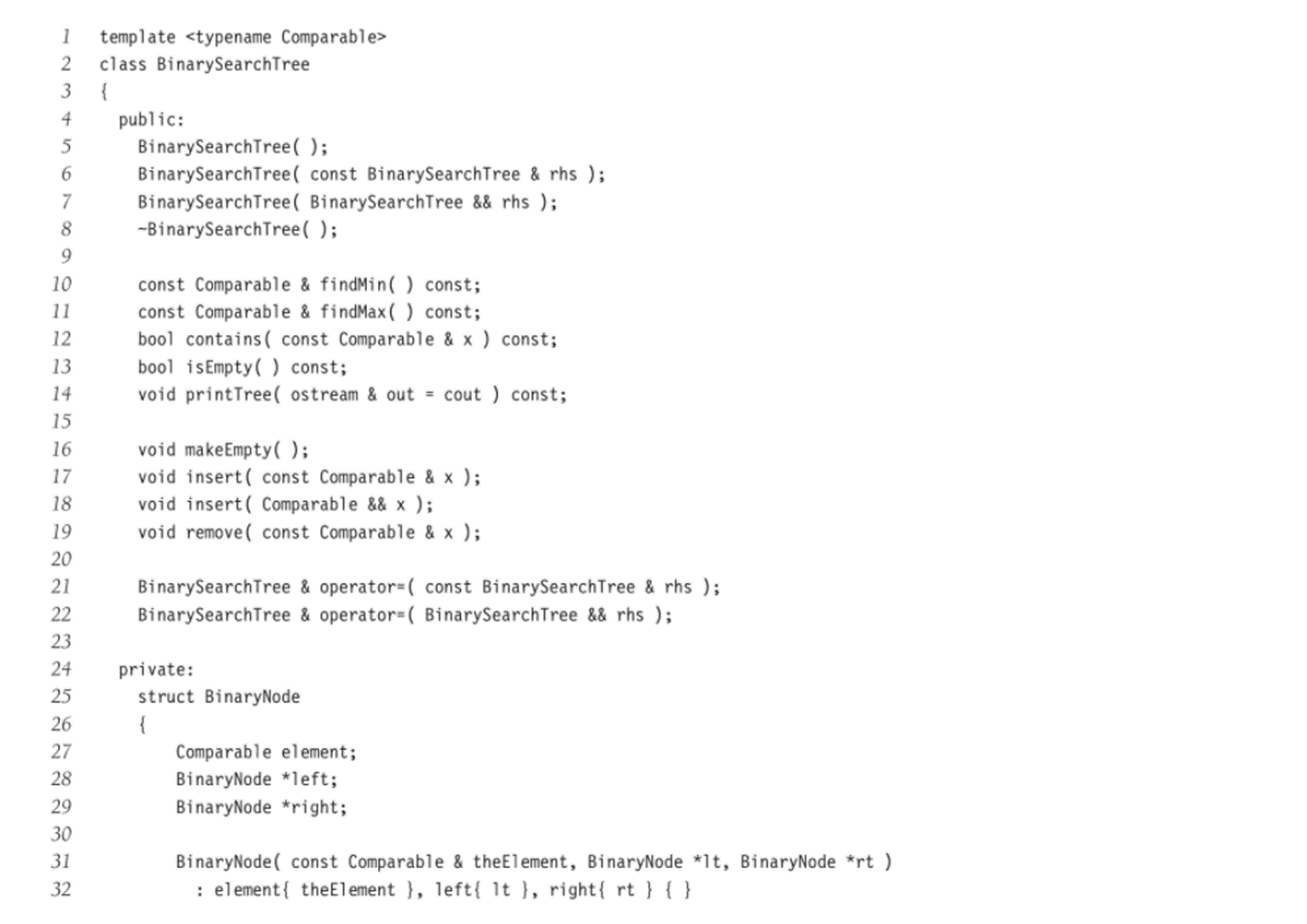
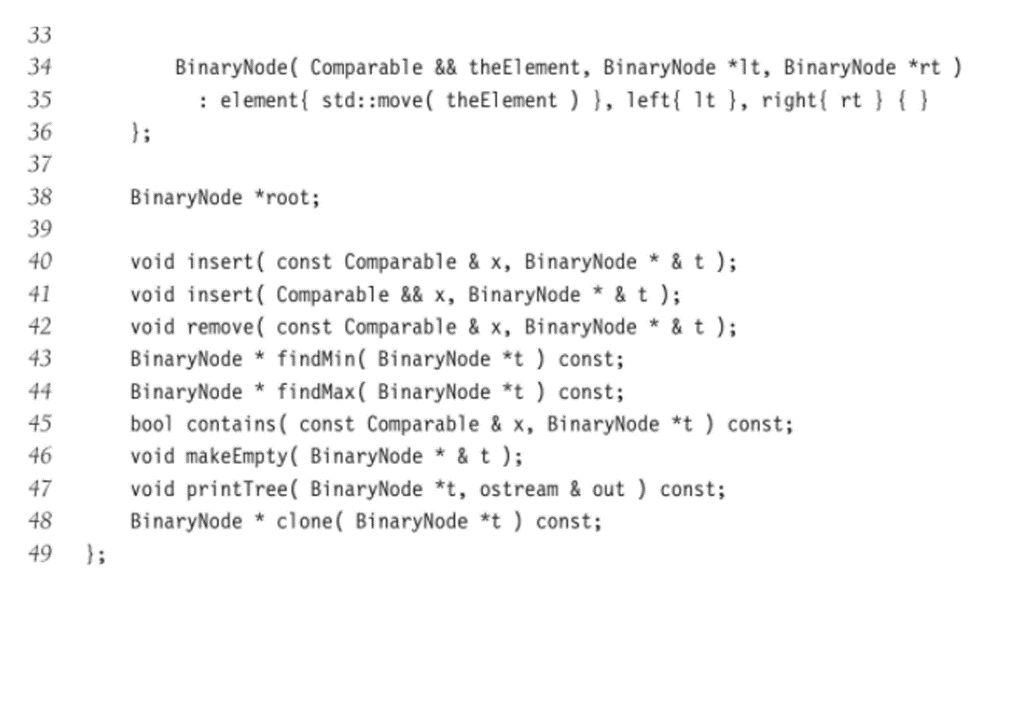
Một ứng dụng quan trọng của cây nhị phân là việc sử dụng chúng trong việc tìm kiếm. Giả sử rằng mỗi nút trong cây lưu trữ một mục. Trong các ví dụ của chúng tôi, chúng tôi sẽ giả định, vì đơn giản, đây là các số nguyên, mặc dù các mục phức tạp tùy ý có thể dễ dàng xử lý trong C ++. Chúng tôi cũng sẽ giả định rằng tất cả các mục là khác biệt và chúng tôi sẽ xử lý các mục trùng lặp sau.

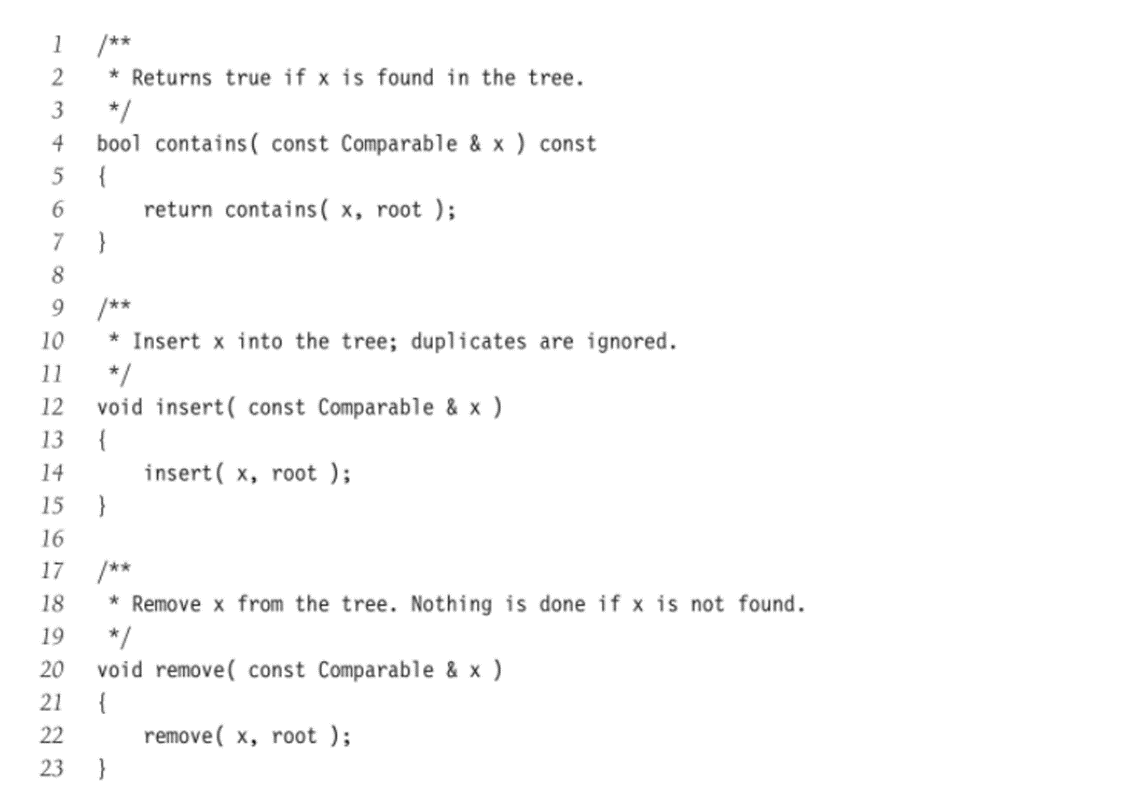
Thuộc tính biến cây nhị phân thành cây tìm kiếm nhị phân là đối với mọi nút, X, trong cây, giá trị của tất cả các mục trong cây con bên trái của nó nhỏ hơn mục trong X và giá trị của tất cả các mục trong cây con bên phải của nó lớn hơn mục trong X. Lưu ý rằng điều này ngụ ý rằng tất cả các phần tử trong cây có thể được sắp xếp theo một số cách nhất quán. Trong hình 4.15, cây bên trái là cây tìm kiếm nhị phân, nhưng cây bên phải thì không. Cây bên phải có một nút với mục 7 trong cây con bên trái của một nút có mục 6 (tình cờ là gốc).

Hình 4.16 cho thấy giao diện của mẫu lớp BinarySearchTree. Có sev- những điều đáng lưu ý. Tìm kiếm dựa trên toán tử <phải được xác định cho kiểu Có thể so sánh cụ thể. Cụ thể, mục x khớp với y nếu cả x <y và y <x đều sai.

Thành viên dữ liệu là một con trỏ tới nút gốc; con trỏ này là nullptr cho cây trống. Các hàm thành viên công khai sử dụng kỹ thuật chung để gọi các hàm đệ quy riêng. Ví dụ về cách thực hiện điều này đối với hàm chứa, chèn và loại bỏ được hiển thị trong Hình 4.17.

**Hình 4.15:** Hai cây nhị phân (chỉ cây bên trái là cây tìm kiếm)

**Hình 4.6:** Bộ xương lớp cây tìm kiếm nhị phân

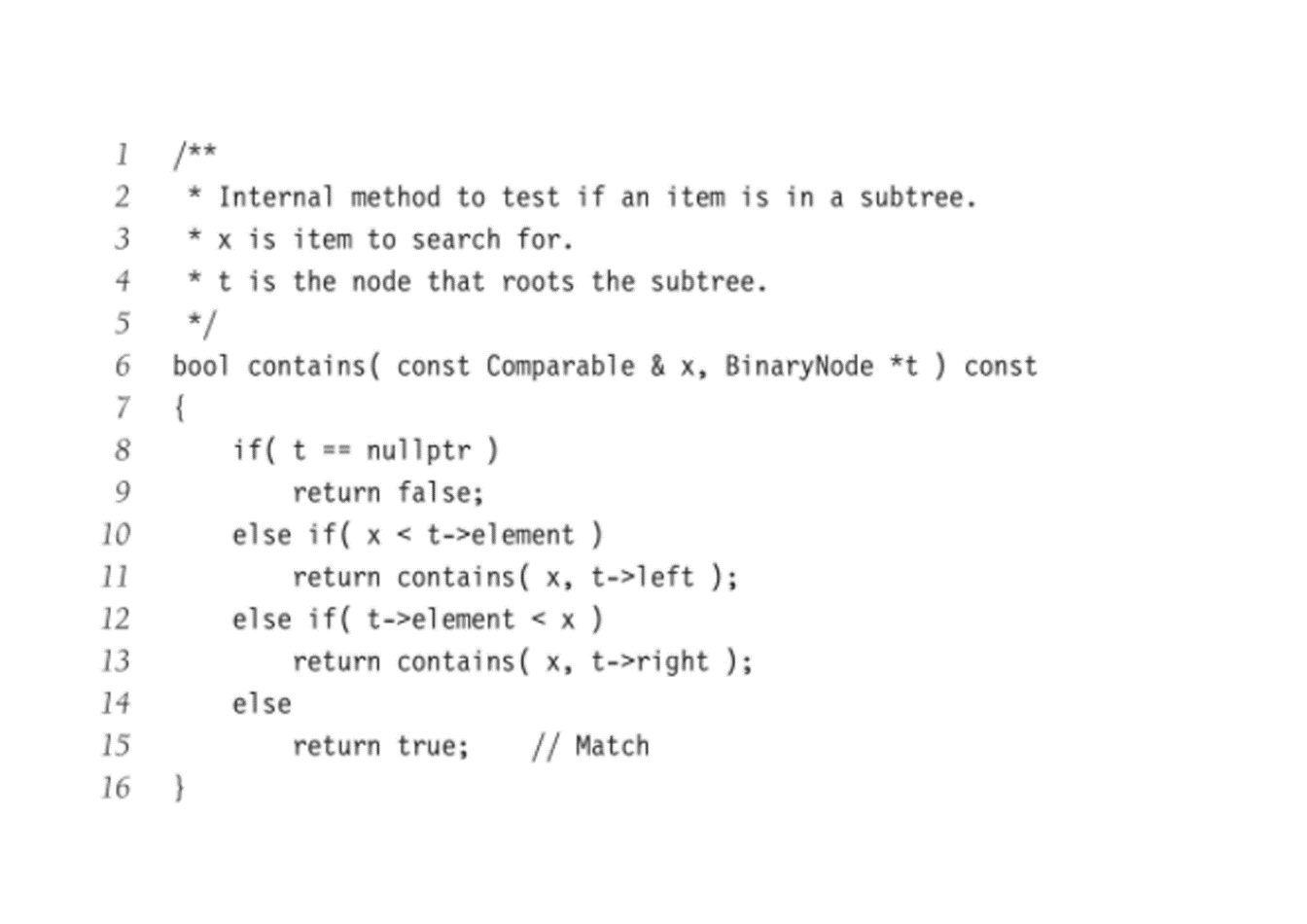
**Hình 4.17:** Minh họa về hàm thành viên công cộng gọi hàm thành viên đệ quy riêng

Một số hàm thành viên riêng sử dụng kỹ thuật truyền một biến con trỏ bằng cách gọi tham chiếu. Điều này cho phép các hàm thành viên công cộng chuyển một con trỏ tới gốc tới các hàm thành viên đệ quy riêng tư. Sau đó, các hàm đệ quy có thể thay đổi giá trị của gốc để gốc trỏ đến một nút khác. Chúng tôi sẽ mô tả kỹ thuật chi tiết hơn khi chúng tôi kiểm tra mã để chèn.

Bây giờ chúng ta có thể mô tả một số phương pháp riêng tư.

**4.3.1 contanis**

Thao tác này yêu cầu trả về true nếu có một nút trong cây T có mục X, hoặc false nếu không có nút đó. Cấu trúc của cây làm cho điều này trở nên đơn giản. Nếu T trống, thì chúng ta chỉ có thể trả về false. Ngược lại, nếu mục được lưu trữ tại T là X, chúng ta có thể trả về true. Nếu không, chúng ta thực hiện một cuộc gọi đệ quy trên cây con của T, trái hoặc phải, tùy thuộc vào mối quan hệ của X với mục được lưu trữ trong T. Đoạn mã trong Hình 4.18 là một triển khai của chiến lược này.

**Hình 4.18:** Chứa hoạt động cho cây tìm kiếm nhị phân

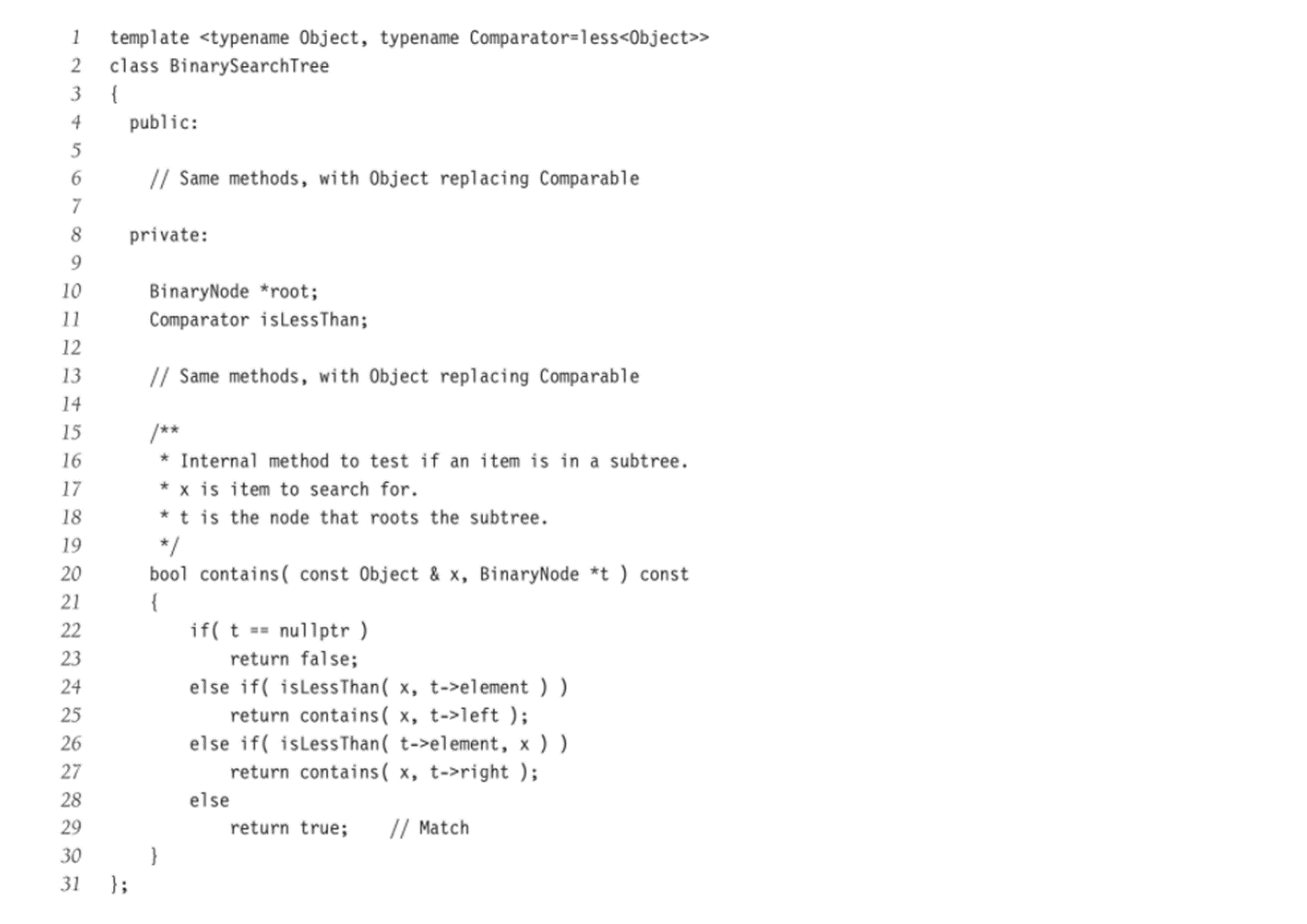
Lưu ý thứ tự của các bài kiểm tra. Điều quan trọng là việc kiểm tra cây trống phải được thực hiện trước, vì nếu không, chúng tôi sẽ tạo ra lỗi thời gian chạy khi cố gắng truy cập vào thành viên dữ liệu thông qua con trỏ nullptr. Các bài kiểm tra còn lại được sắp xếp với trường hợp ít có khả năng xảy ra nhất sau cùng. Cũng lưu ý rằng cả hai lệnh gọi đệ quy thực sự là đệ quy đuôi và có thể dễ dàng loại bỏ bằng vòng lặp while. Việc sử dụng đệ quy đuôi là hợp lý ở đây vì tính đơn giản của biểu thức thuật toán bù cho sự giảm tốc độ và lượng không gian ngăn xếp được sử dụng dự kiến chỉ là O (log N).

Hình 4.19 cho thấy những thay đổi nhỏ cần thiết để sử dụng một đối tượng chức năng hơn là yêu cầu các mục phải có thể So sánh được. Điều này bắt chước các thành ngữ trong Phần 1.6.4.

**4.3.2 findMin và findMax**

Các quy trình private này trả về một con trỏ tới nút chứa các phần tử nhỏ nhất và lớn nhất trong cây, tương ứng. Để thực hiện findMin, hãy bắt đầu từ gốc và đi sang trái miễn là có con bên trái. Điểm dừng là phần tử nhỏ nhất. Quy trình findMax cũng giống như vậy, ngoại trừ việc phân nhánh đến đúng con.

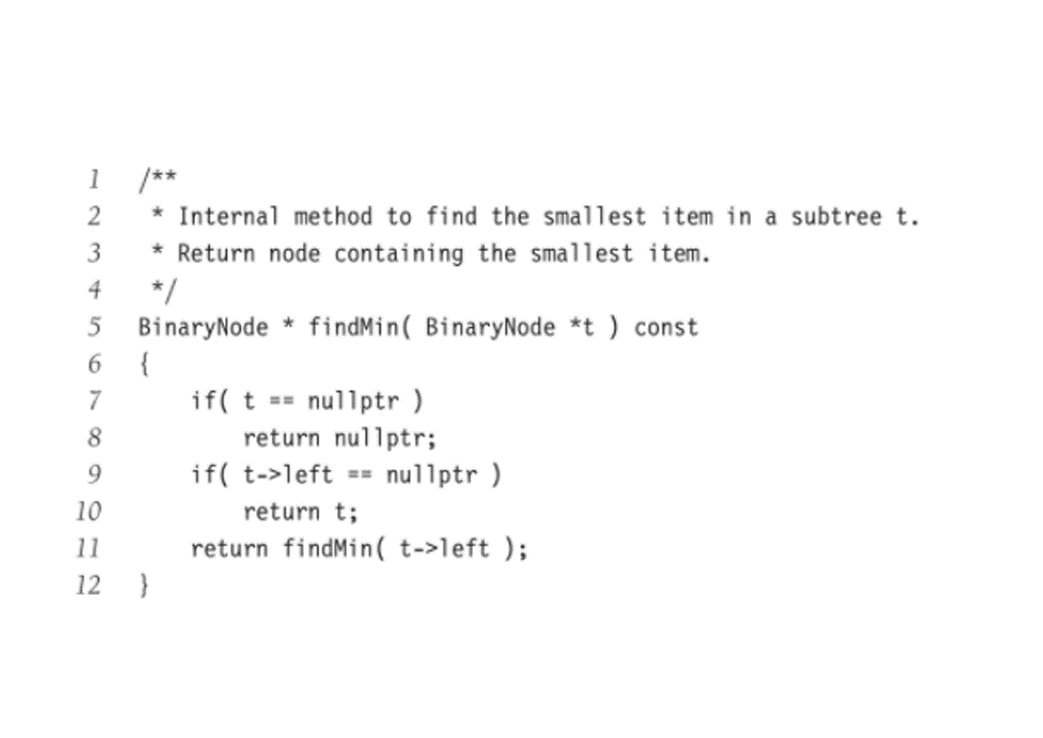
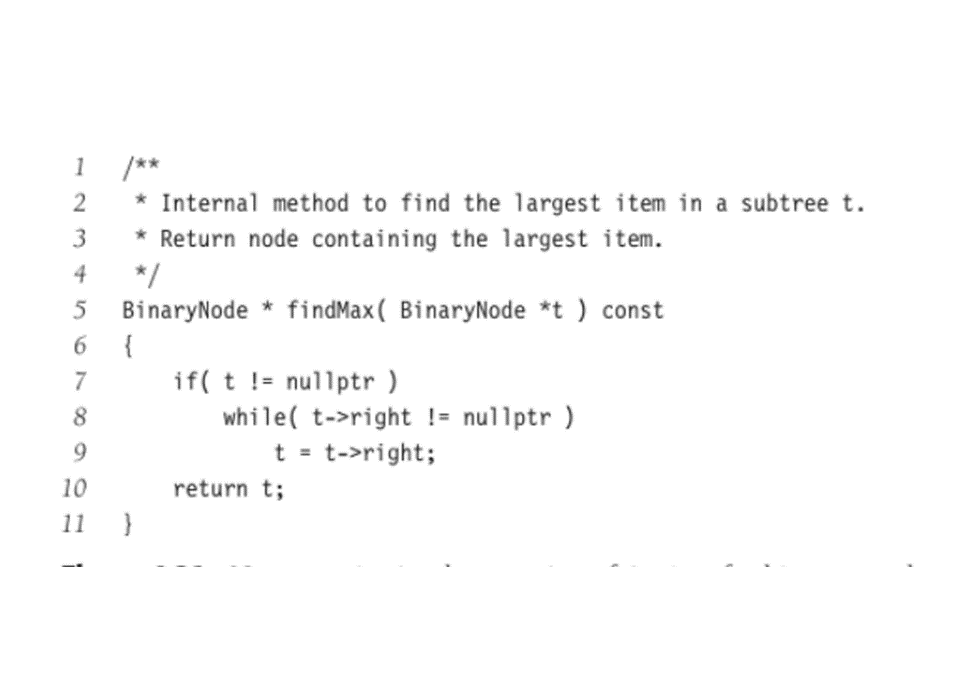
Nhiều lập trình viên không bận tâm đến việc sử dụng đệ quy. Chúng tôi sẽ mã hóa các quy trình theo cả hai cách bằng cách thực hiện đệ quy findMin và findMax không đệ quy (xem Hình 4.20 và 4.21).

 Chú ý cách chúng tôi xử lý cẩn thận trường hợp thoái hóa của cây trống. Mặc dù điều này luôn quan trọng phải làm, nó đặc biệt quan trọng trong các chương trình đệ quy. Cũng lưu ý rằng việc thay đổi t trong findMax là an toàn, vì chúng tôi chỉ làm việc với bản sao của một con trỏ. Tuy nhiên, hãy luôn cực kỳ cẩn thận vì một câu lệnh như t-> right = t-> right-> right sẽ tạo ra các thay đổi.

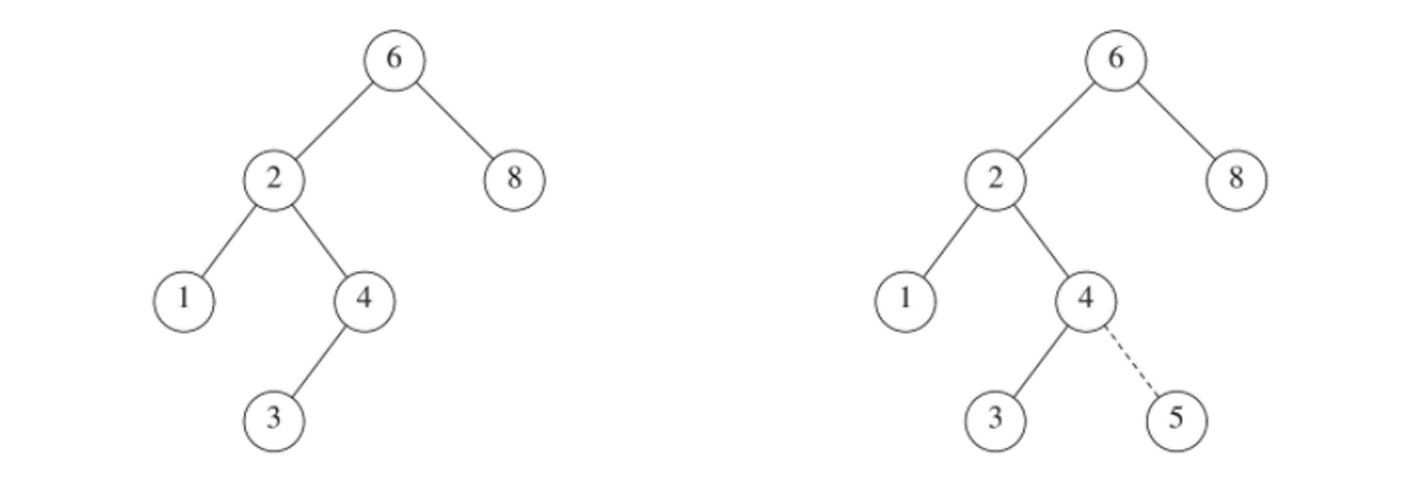
**Hình 4.19:** Minh họa việc sử dụng một đối tượng hàm để triển khai cây tìm kiếm nhị phân

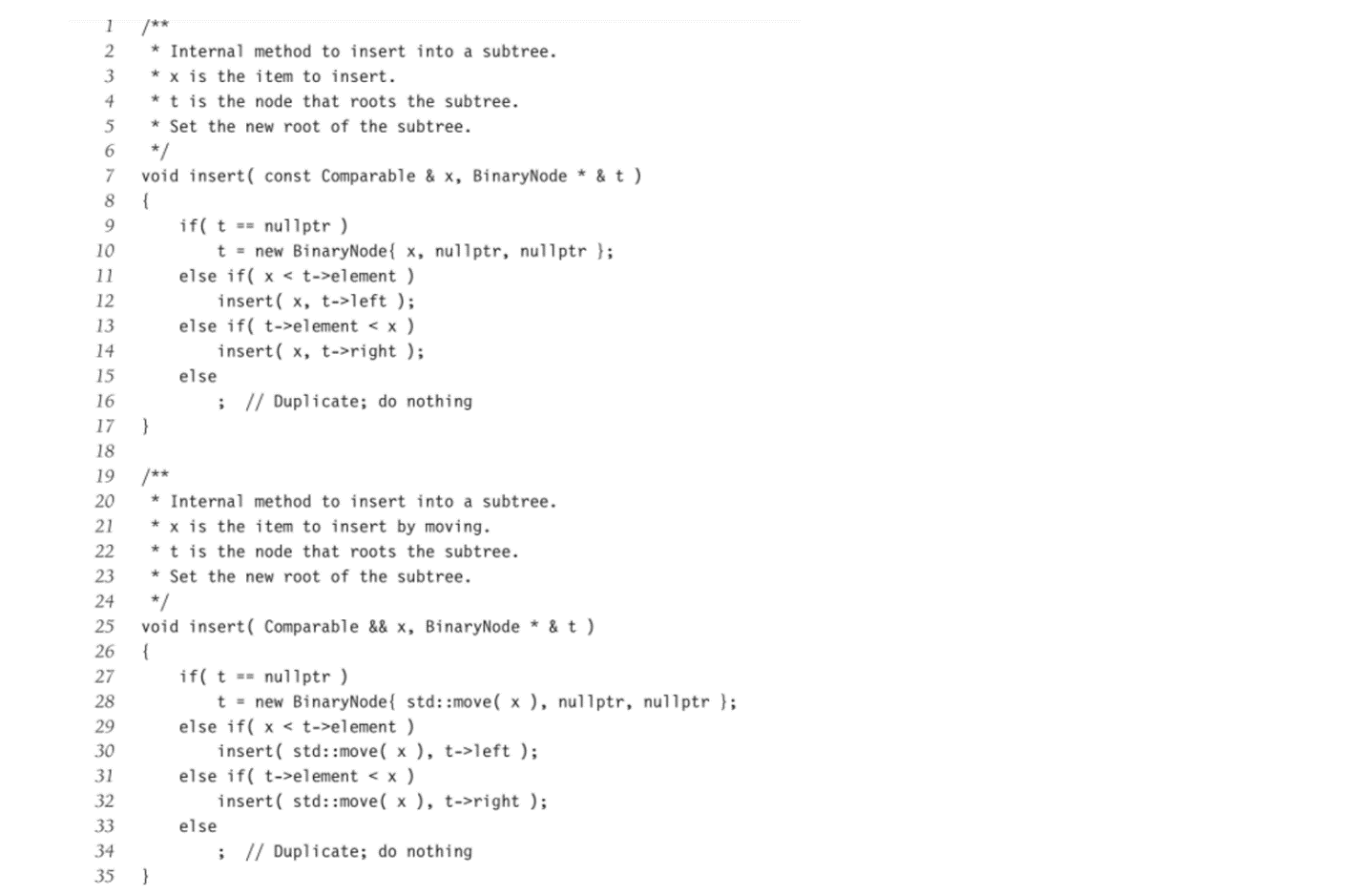
**4.3.3 insert**

Quy trình chèn rất đơn giản về mặt khái niệm. Để chèn X vào cây T, hãy tiến hành xuống cây như bạn làm với hàm chứa. Nếu X được tìm thấy, không làm gì cả. Nếu không, hãy chèn X vào vị trí cuối cùng trên đường đi ngang. Hình 4.22 cho thấy điều gì xảy ra. Để chèn 5, chúng tôi duyệt cây như thể một chứa đang xảy ra. Tại nút có mục 4, chúng ta cần phải đi sang phải, nhưng không có cây con, vì vậy 5 không có trong cây và đây là vị trí chính xác để đặt 5.

Các bản sao có thể được xử lý bằng cách giữ một trường bổ sung trong bản ghi nút cho biết tần suất xuất hiện. Điều này tạo thêm một số không gian cho toàn bộ cây nhưng tốt hơn là đặt các bản sao vào cây (có xu hướng làm cho cây rất sâu). Tất nhiên,  
**Hình 4.20:** Triển khai đệ quy findMin cho cây tìm kiếm nhị phân

**Hình 4.21:** Triển khai không đệ quy của tìm Max cho cây tìm kiếm nhị phân

**Hình 4.22:** Cây tìm kiếm nhị phân trước và sau khi chèn 5



**Hình 4.23:** Chèn vào cây tìm kiếm nhị phân

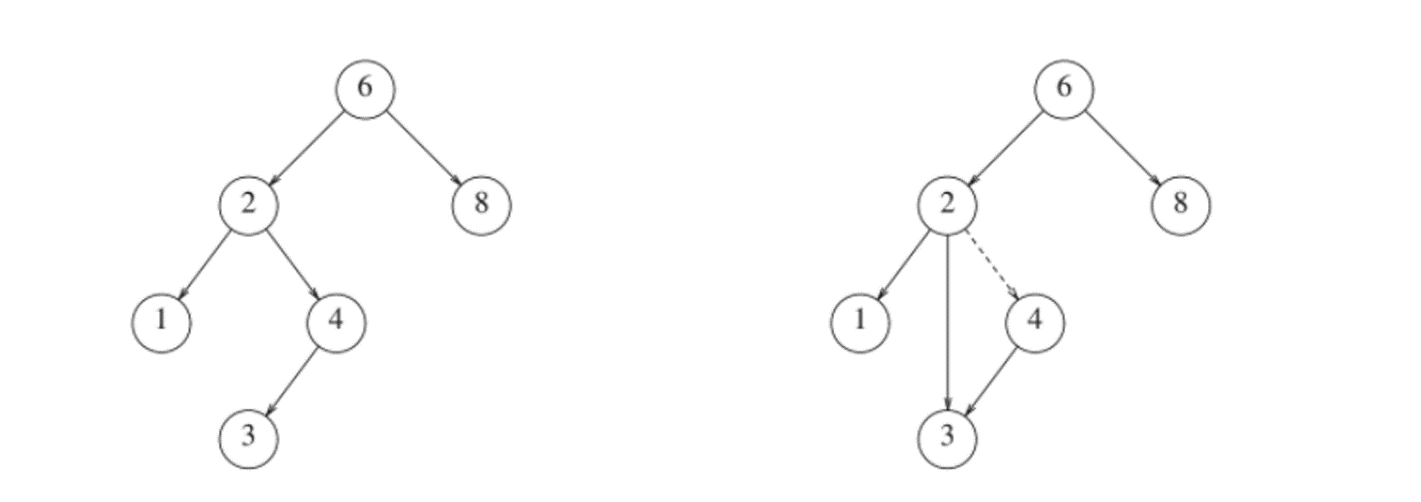
Sẽ được insert (x, p-> left) hoặc insert (x, p-> right). Dù bằng cách nào, t bây giờ là một tham chiếu đến p-> left hoặc p-> right, nghĩa là p-> left hoặc p-> right sẽ được thay đổi để trỏ đến nút mới. Nói chung, một thao tác mượt mà.

**4.3.4 remove**

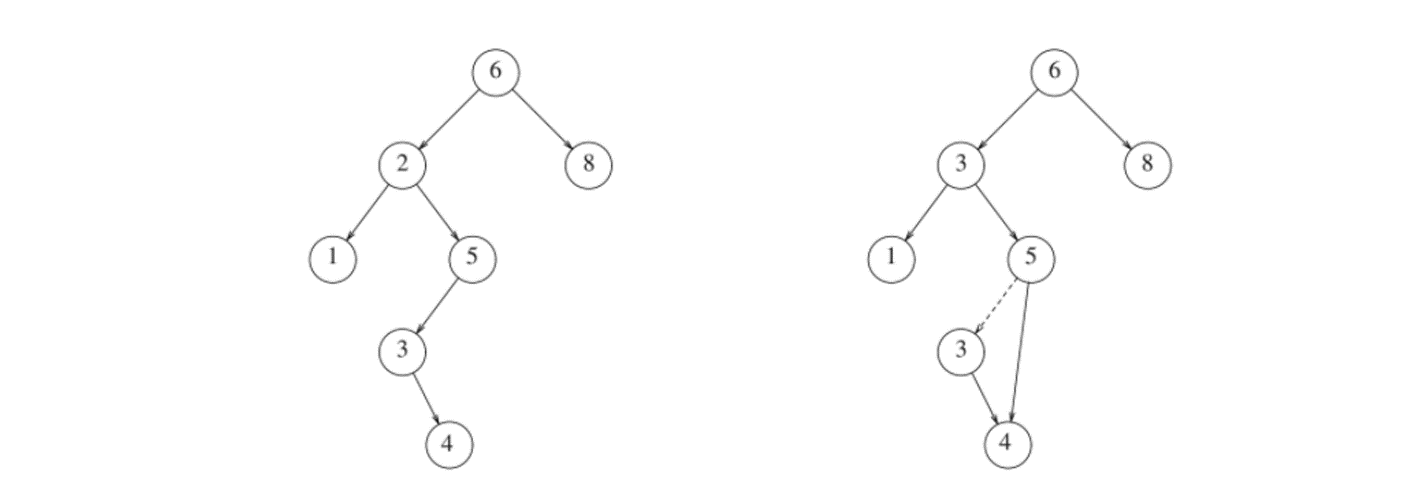
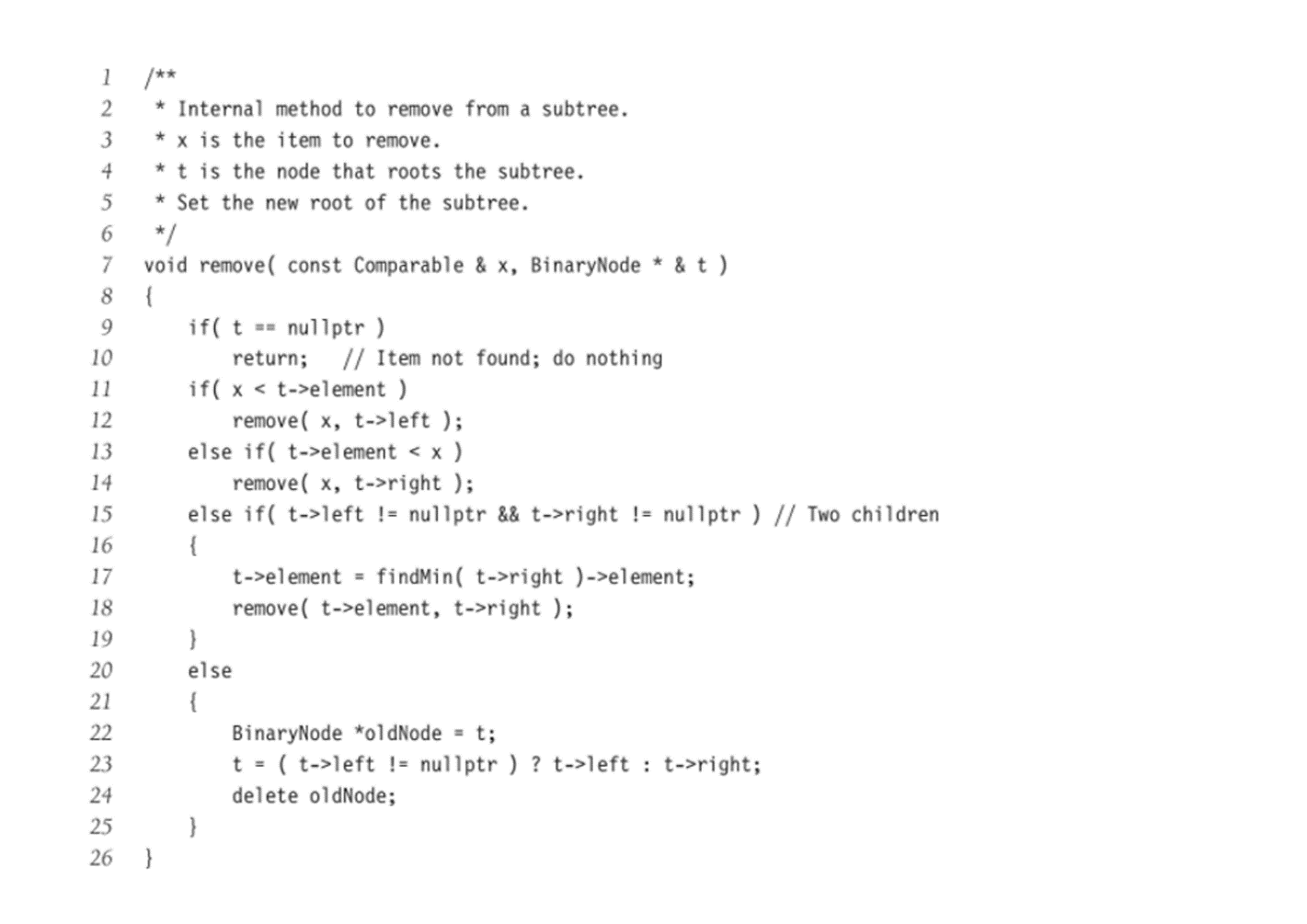
Như thường thấy với nhiều cấu trúc dữ liệu, thao tác khó nhất là xóa. Khi chúng tôi nhận thấy nút bị xóa, chúng tôi cần xem xét một số khả năng.

Nếu nút là một lá, nó có thể bị xóa ngay lập tức. Nếu nút có một nút con, nút có thể bị xóa sau khi nút cha của nó điều chỉnh một liên kết để bỏ qua nút (chúng tôi sẽ vẽ các hướng liên kết một cách rõ ràng cho rõ ràng). Xem Hình 4.24.

Đoạn mã trong Hình 4.26 thực hiện xóa. Nó không hiệu quả bởi vì nó thực hiện hai lần chuyển xuống cây để tìm và xóa nút nhỏ nhất trong cây con bên phải khi nút này được chấp thuận. Thật dễ dàng để loại bỏ sự kém hiệu quả này bằng cách viết một phương thức removeMin đặc biệt và chúng tôi chỉ để lại nó vì sự đơn giản.

Nếu số lần xóa dự kiến là ít, thì một chiến lược phổ biến được sử dụng là xóa lười: Khi một phần tử bị xóa, nó sẽ được để lại trong cây và chỉ được đánh dấu là bị xóa. Điều này đặc biệt phổ biến nếu có các mục trùng lặp, vì khi đó thành phần dữ liệu lưu giữ tần suất xuất hiện có thể bị giảm đi. Nếu số lượng nút thực trong cây bằng với số nút “bị xóa”, thì độ sâu của cây chỉ được mong đợi tăng lên bởi một hằng số nhỏ (tại sao?),

**Hình 4.24:** Xóa một nút (4) với một nút con, trước và sau

  
**Hình 4.25:** Xóa một nút (2) có hai nút con, trước và sau.

**Hình 4.26:** Quy trình xóa cây tìm kiếm nhị phân

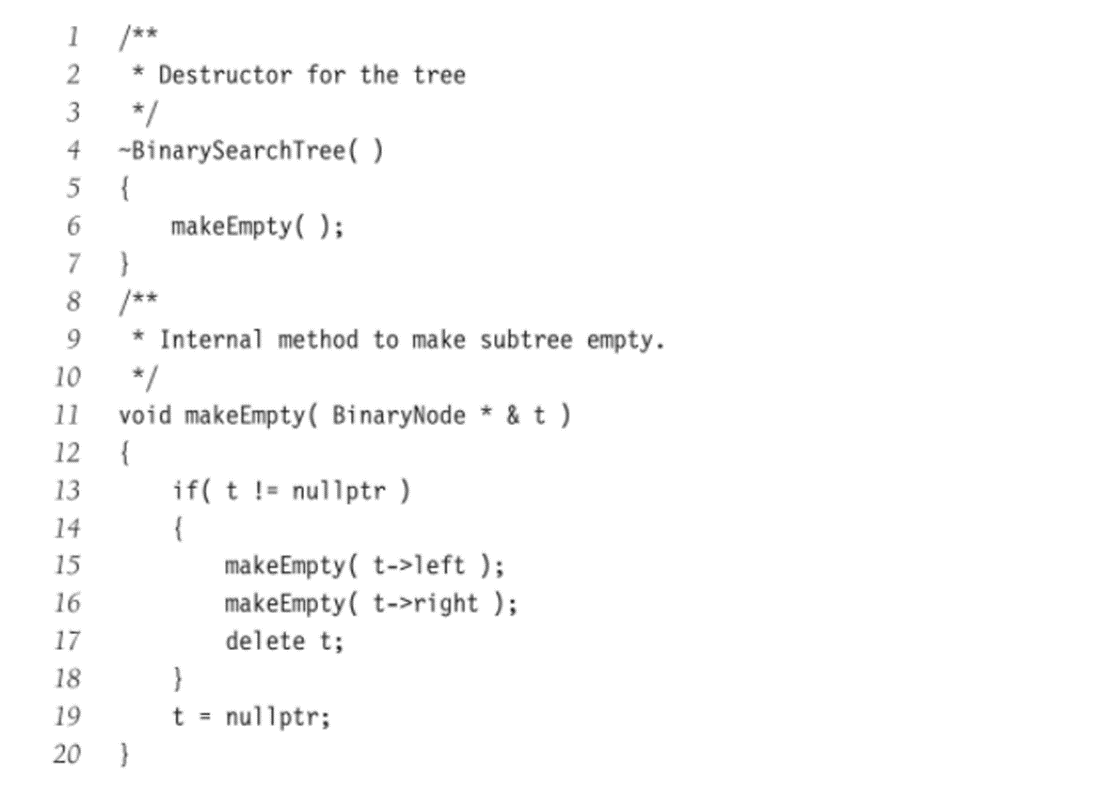
**4.3.5 Destrutor and Copy Constructor** (Bộ tạo hủy và sao chép)

Như thường lệ, hàm hủy gọi makeEmpty. Công khai makeEmpty (không hiển thị) chỉ đơn giản gọi phiên bản đệ quy riêng tư. Như trong Hình 4.27, sau khi xử lý đệ quy các con của t, một cuộc gọi để xóa được thực hiện cho t. Do đó, tất cả các nút được lấy lại một cách đệ quy. Lưu ý rằng ở cuối, t, và do đó gốc, được thay đổi thành điểm nullptr. Phương thức tạo bản sao, được hiển thị trong Hình 4.28, tuân theo thủ tục thông thường, đầu tiên khởi tạo root thành nullptr và sau đó tạo một bản sao của rhs. Chúng tôi sử dụng một hàm đệ quy rất mượt có tên là clone để thực hiện tất cả các công việc bẩn thỉu.

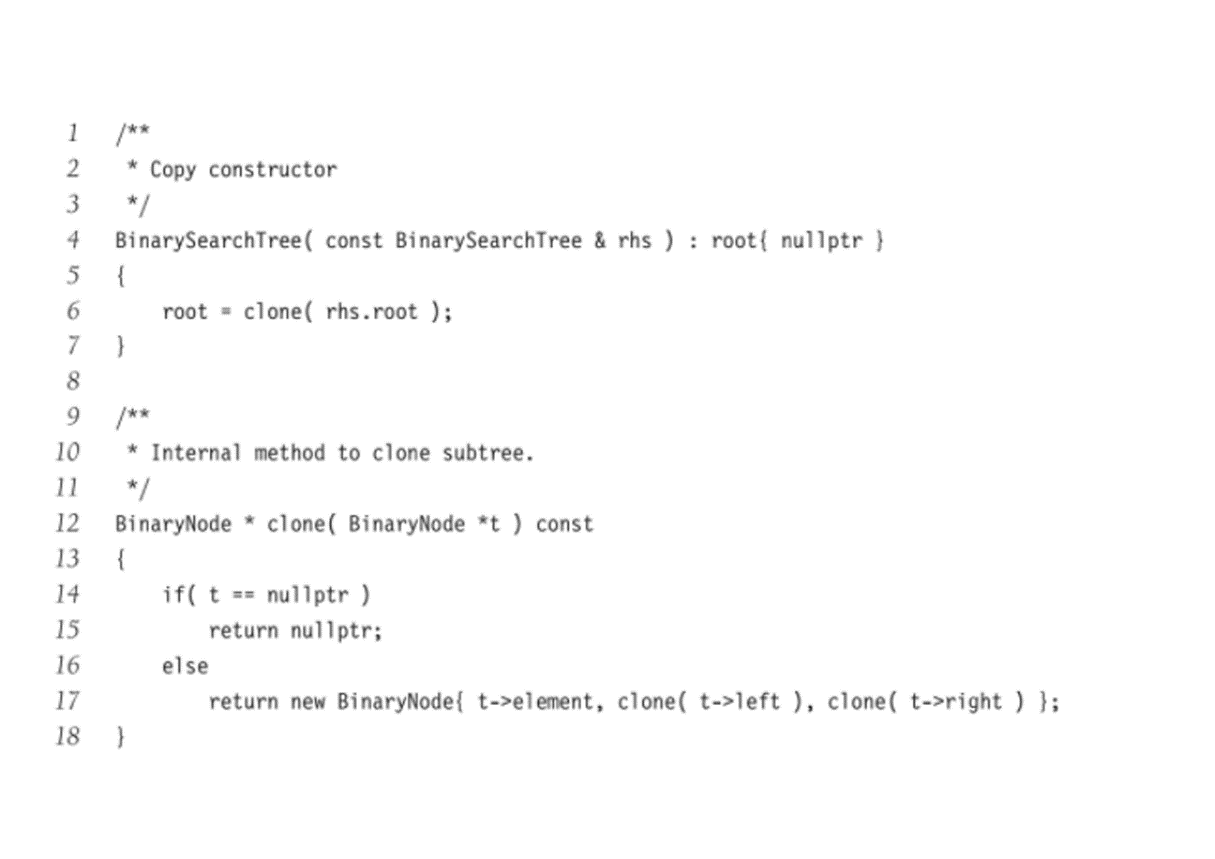
**4.3.6 Average-Case Analysis** (Phân tích trường hợp trung bình)

Theo trực giác, chúng tôi hy vọng rằng tất cả các hoạt động được mô tả trong phần này, ngoại trừ makeEmpty và sao chép, sẽ mất thời gian O (logN), bởi vì trong thời gian liên tục, chúng tôi giảm một cấp trong cây, do đó hoạt động trên một cây mà bây giờ gần một nửa lớn. Thật vậy, thời gian chạy của tất cả các hoạt động (ngoại trừ makeEmpty và sao chép) là O (d), trong đó d là độ sâu của nút chứa mục được truy cập (trong trường hợp loại bỏ, đây có thể là nút thay thế trong hai- trường hợp con).

Trong phần này, chúng tôi chứng minh rằng độ sâu trung bình trên tất cả các nút trong cây là O (logN) trên giả thiết rằng tất cả các chuỗi chèn đều có khả năng như nhau.

Tổng độ sâu của tất cả các nút trong cây được gọi là chiều dài đường dẫn nội bộ. Bây giờ chúng ta sẽ tính toán độ dài đường dẫn nội bộ trung bình của cây tìm kiếm nhị phân, trong đó giá trị trung bình được lấy trên tất cả các chuỗi chèn có thể có vào cây tìm kiếm nhị phân.

**Hình 4.27:** Hàm hủy và hàm thành viên makeEmpty đệ quy

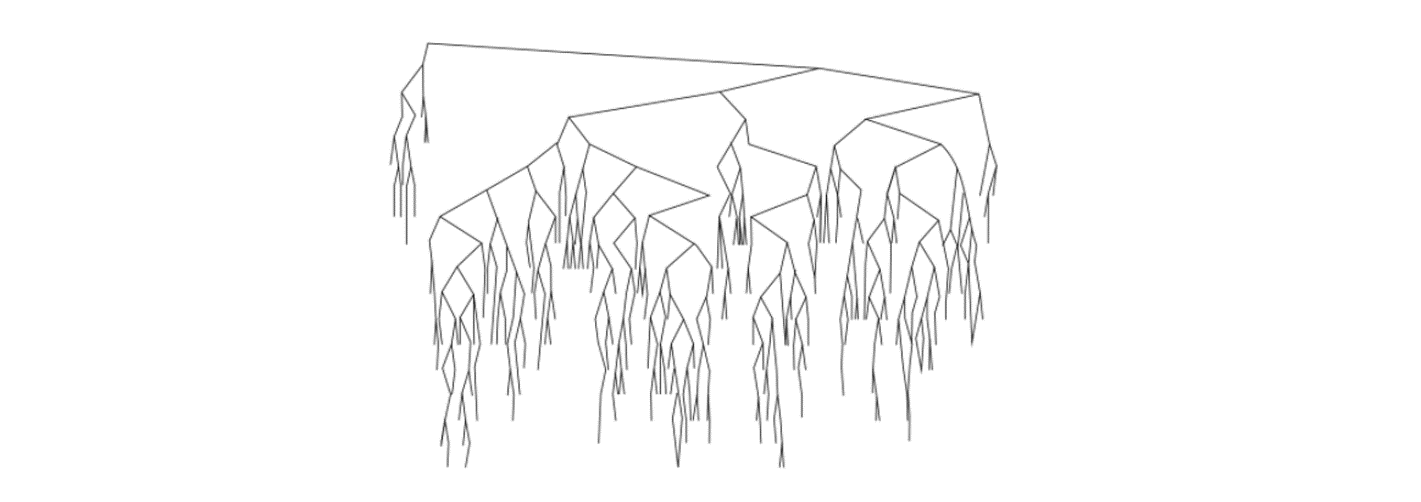
**Hình 4.28:** Sao chép hàm tạo và hàm thành viên sao chép đệ quy

Gọi D (N) là độ dài đường đi trong của một số cây T gồm N nút. D (1) = 0. Một cây N nút bao gồm một cây con bên trái thứ i và một cây con bên phải (N - i - 1), cộng với một gốc ở độ sâu 0 cho 0 ≤ i <N. D (i ) là độ dài đường dẫn bên trong của cây con bên trái đối với gốc của nó. Trong cây chính, tất cả các nút này sâu hơn một cấp. Tương tự đối với cây con bên phải. Do đó, chúng tôi nhận được sự lặp lại

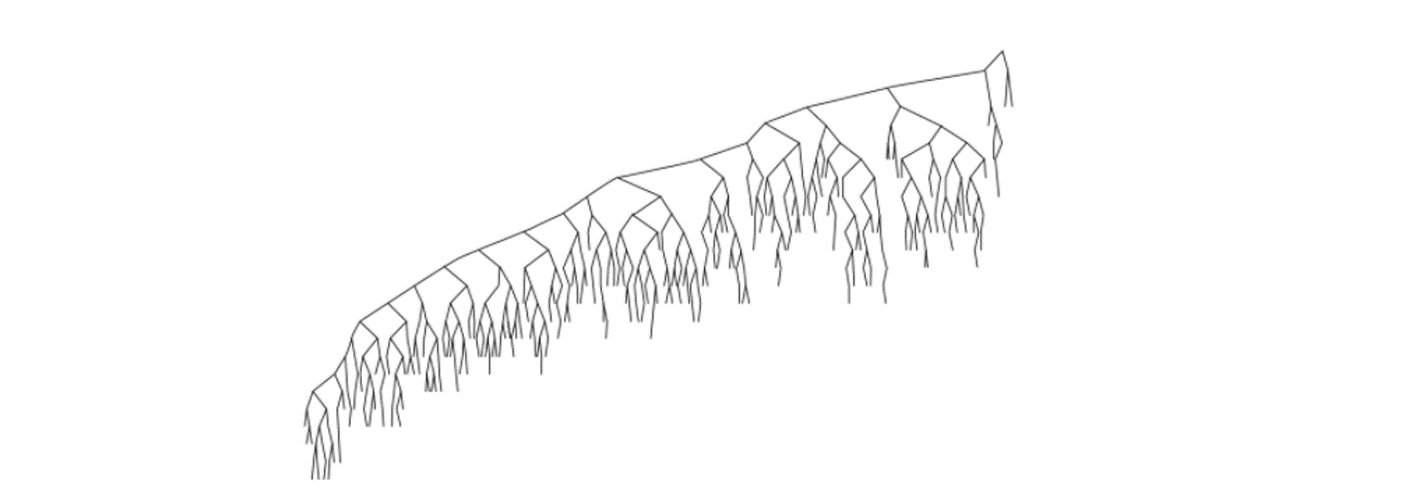
D(N) = D(i) + D(N − i − 1) + N – 1

Nếu tất cả các kích thước cây con đều có khả năng như nhau, điều này đúng với cây tìm kiếm nhị phân (vì kích thước cây con chỉ phụ thuộc vào thứ hạng tương đối của phần tử đầu tiên được chèn vào cây), nhưng không phải cây nhị phân, thì giá trị trung bình của cả D (i ) và D (N - i - 1) là (1 / N) N − 1 j = 0 D (j). Điều này mang lại

Sự tái diễn này sẽ được gặp và giải quyết trong Chương 7, thu được giá trị trung bình là D (N) = O (N logN). Do đó, độ sâu dự kiến của bất kỳ nút nào là O (logN). Ví dụ, cây 500 nút được tạo ngẫu nhiên trong Hình 4.29 có các nút ở độ sâu dự kiến 9,98.

**Hình 4.29** Cây tìm kiếm nhị phân được tạo ngẫu nhiên

nó dường như chỉ là một tính mới về mặt lý thuyết. Người ta đã chỉ ra rằng nếu chúng ta xen kẽ các lần chèn và lần xóa (N2), thì cây sẽ có độ sâu dự kiến là (√N). Sau một phần tư triệu cặp chèn / loại bỏ ngẫu nhiên, cây hơi nặng bên phải trong Hình 4.29 trông hoàn toàn không cân đối (độ sâu trung bình = 12,51) trong Hình 4.30.

Chúng ta có thể cố gắng loại bỏ vấn đề bằng cách chọn ngẫu nhiên giữa phần tử nhỏ nhất trong cây con bên phải và phần tử lớn nhất ở bên trái khi thay thế phần tử đã xóa. Điều này dường như loại bỏ sự thiên vị và sẽ giữ cho cây cân bằng, nhưng không ai có

**Hình 4.30** Cây tìm kiếm nhị phân sau khi chèn / xóa các cặp (N2)

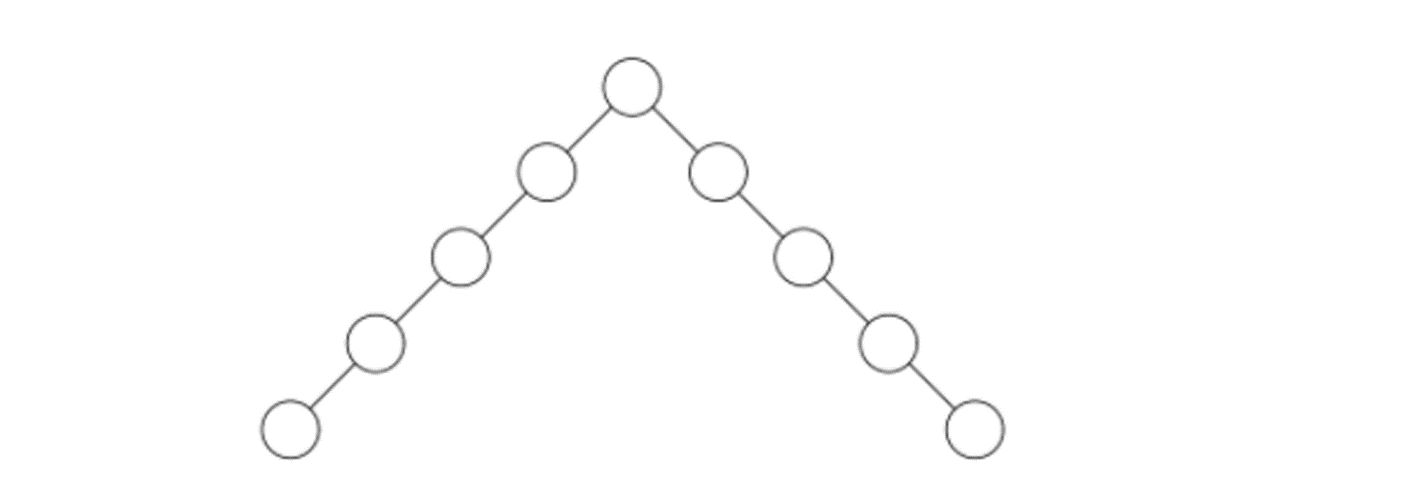
thực tế đã chứng minh điều này. Trong bất kỳ trường hợp nào, hiện tượng này hầu như chỉ là một tính mới về mặt lý thuyết, bởi vì hiệu ứng này hoàn toàn không xuất hiện đối với các cây nhỏ, và người lạ vẫn còn, nếu sử dụng cặp chèn / loại bỏ o (N2), thì cây dường như đạt được. thăng bằng!

Điểm chính của cuộc thảo luận này là việc quyết định “trung bình” nghĩa là gì là vô cùng khó khăn và có thể yêu cầu các giả định có thể có hoặc có thể không hợp lệ. Trong trường hợp không xóa hoặc khi xóa lười được sử dụng, chúng ta có thể kết luận rằng thời gian chạy trung bình của các hoạt động trên là O (logN). Ngoại trừ những trường hợp kỳ lạ như đã thảo luận ở trên, kết quả này rất phù hợp với hành vi quan sát được.

Phương pháp thứ hai là bỏ qua điều kiện cân bằng và cho phép cây có độ sâu tùy ý, nhưng sau mỗi hoạt động, quy tắc tái cấu trúc được áp dụng có xu hướng làm cho hoạt động trong tương lai hiệu quả. Các loại cấu trúc dữ liệu này thường được phân loại là tự điều chỉnh. Trong trường hợp cây tìm kiếm nhị phân, chúng ta không còn có thể đảm bảo ràng buộc O (logN) trên bất kỳ thao tác đơn lẻ nào nhưng có thể chỉ ra rằng bất kỳ chuỗi thao tác M nào cũng chiếm tổng thời gian là O (M logN) trong trường hợp xấu nhất. Điều này nói chung là đủ bảo vệ chống lại trường hợp xấu nhất. Cấu trúc dữ liệu chúng ta sẽ thảo luận được biết đến như một cây splay; phân tích của nó khá phức tạp và được thảo luận trong Chương 11.

**4.4 AVL Trees** (Cây AVL)

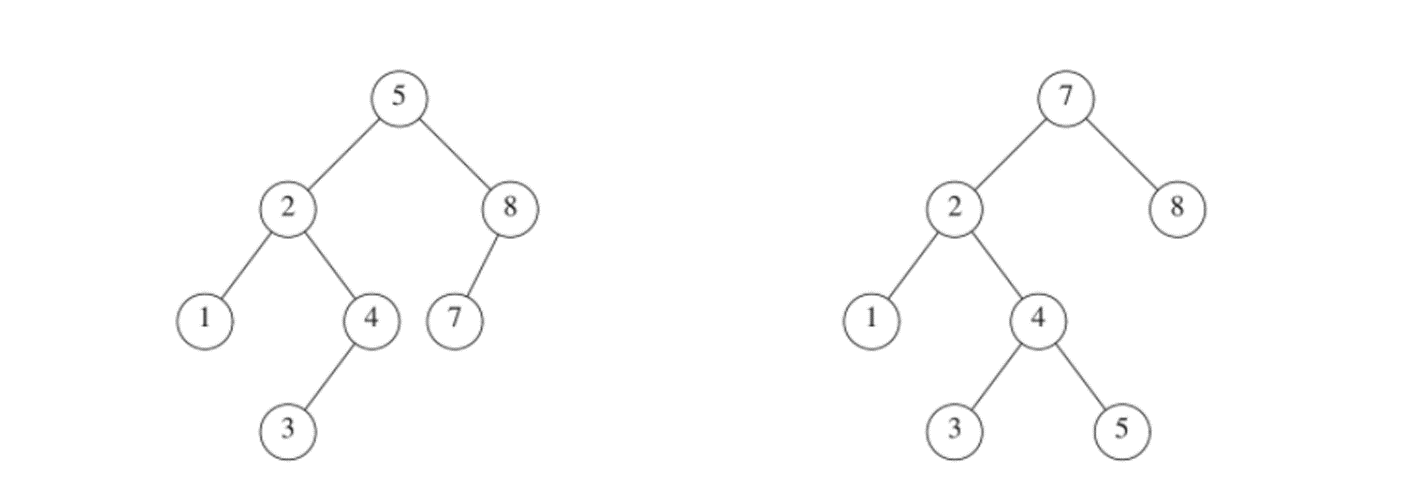
Cây AVL (Adelson-Velskii và Landis) là cây tìm kiếm nhị phân với điều kiện cân bằng. Điều kiện cân bằng phải dễ duy trì và đảm bảo độ sâu của cây là O (logN). Ý tưởng đơn giản nhất là yêu cầu các cây con bên trái và bên phải có cùng chiều cao. Như hình 4.31 cho thấy, ý tưởng này không buộc cây phải cạn.



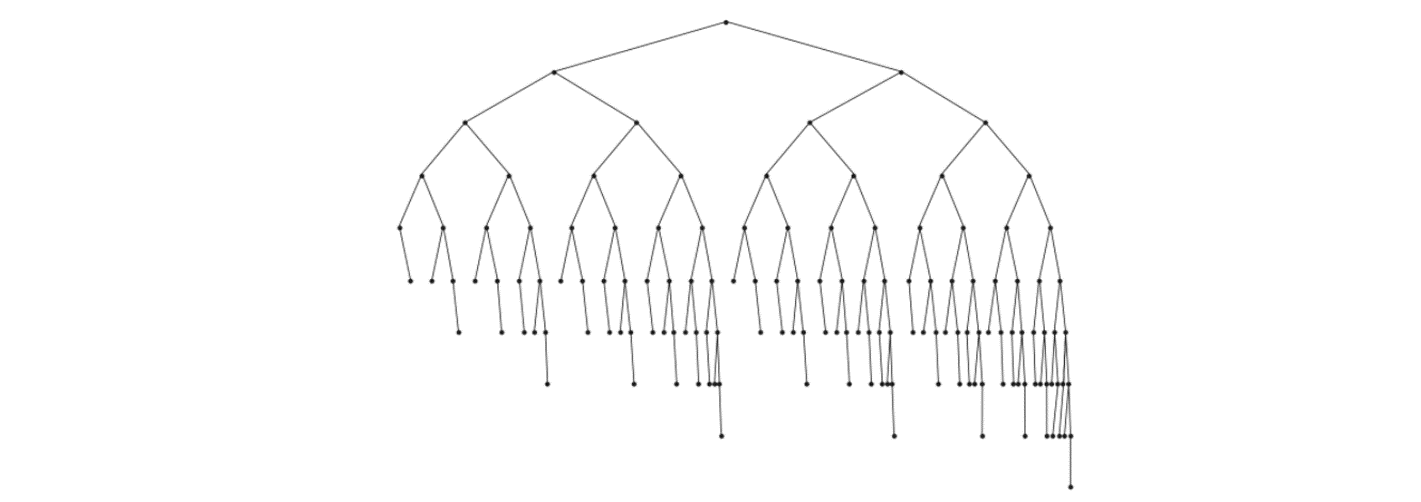
**Hình 4.31** Một cây nhị phân xấu. Yêu cầu sự cân bằng ở gốc là không đủ.

Một điều kiện cân bằng khác sẽ nhấn mạnh rằng mọi nút phải có cây con trái và phải có cùng chiều cao. Nếu chiều cao của cây con trống được xác định là −1 (như thường lệ), thì chỉ những cây cân bằng hoàn hảo gồm 2k - 1 nút mới thỏa mãn tiêu chí này. Do đó, mặc dù điều này đảm bảo cho cây có độ sâu nhỏ, nhưng điều kiện cân bằng quá cứng nhắc không thể hữu ích và cần được nới lỏng.

Cây AVL giống hệt cây tìm kiếm nhị phân, ngoại trừ mỗi nút trong cây, chiều cao của cây con bên trái và bên phải có thể khác nhau nhiều nhất là 1 (Chiều cao của cây trống được định nghĩa là −1.) Trong hình 4.32 cây bên trái là cây AVL nhưng cây bên phải thì không. Thông tin về độ cao được giữ cho mỗi nút (trong cấu trúc nút). Có thể chỉ ra rằng chiều cao của cây AVL nhiều nhất là khoảng 1,44 log (N + 2) - 1,328, nhưng trên thực tế, nó chỉ nhỉnh hơn một chút so với logN. Ví dụ, cây AVL có chiều cao 9 với ít nút nhất (143) được thể hiện trong Hình 4.33Cây con bên phải là cây AVL có kích thước tối thiểu là 8 chiều cao. Điều này cho chúng ta biết rằng số lượng nút tối thiểu, S (h), trong cây AVL có chiều cao h được cho bởi S (h) = S (h - 1) + S (h - 2) + 1. Với h = 0 , S (h) = 1. Với h = 1, S (h) = 2. Hàm S (h) có liên quan chặt chẽ với các số Fibonacci, từ đó giới hạn xác nhận ở trên đối với chiều cao của cây AVL theo sau.

Sau khi chèn, chỉ các nút nằm trên đường dẫn từ điểm chèn đến gốc mới có thể bị thay đổi số dư của chúng vì chỉ những nút đó mới bị thay đổi cây con. Khi chúng tôi đi theo đường dẫn đến gốc và cập nhật thông tin cân bằng, chúng tôi có thể tìm thấy một nút có số dư mới vi phạm điều kiện AVL. Chúng tôi sẽ chỉ ra cách cân bằng lại cây tại nút đầu tiên (tức là sâu nhất) như vậy và chúng tôi sẽ chứng minh rằng sự cân bằng lại này đảm bảo rằng toàn bộ cây thỏa mãn thuộc tính AVL.

**Hình 4.32** Hai cây tìm kiếm nhị phân. Chỉ có cây bên trái là AVL.

**Hình 4.33** Cây AVL nhỏ nhất có chiều cao 9

Chúng ta hãy gọi nút phải được cân bằng lại α. Vì bất kỳ nút nào đều có nhiều nhất hai chil-dren và sự mất cân bằng độ cao yêu cầu độ cao của hai cây con của α khác nhau hai, nên dễ dàng nhận thấy rằng vi phạm có thể xảy ra trong bốn trường hợp:

1. Một phần chèn vào cây con bên trái của con trái của α

2. Một phần chèn vào cây con bên phải của con trái của α

3. Một phần chèn vào cây con bên trái của con bên phải của α

4. Một phần chèn vào cây con bên phải con bên phải của α

Trường hợp 1 và 4 là đối xứng ảnh qua gương đối với α, cũng như trường hợp 2 và 3. Do đó, theo lý thuyết, có hai trường hợp cơ bản. Từ góc độ lập trình, tất nhiên, vẫn có bốn trường hợp.

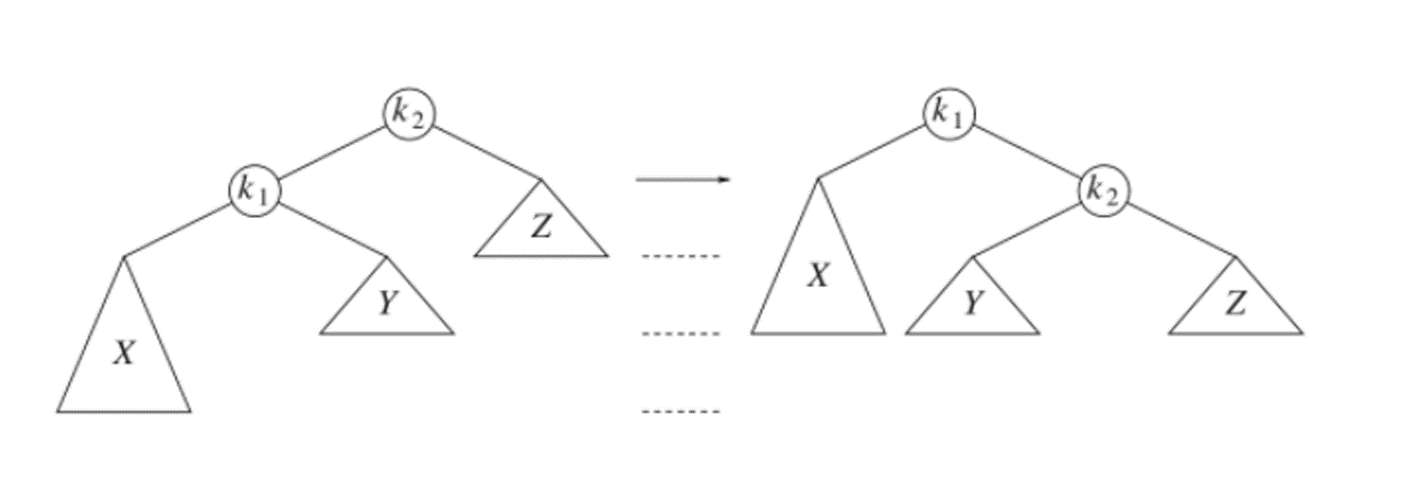
Trường hợp đầu tiên, trong đó việc chèn xảy ra ở “bên ngoài” (tức là trái – trái hoặc phải – phải), được cố định bằng một lần xoay cây. Trường hợp thứ hai, trong đó việc chèn xảy ra ở “bên trong” (tức là trái – phải hoặc phải – trái) được xử lý bằng cách quay kép phức tạp hơn một chút. Đây là các phép toán cơ bản trên cây mà chúng ta sẽ thấy được sử dụng nhiều lần trong các thuật toán cây cân bằng. Phần còn lại của phần này mô tả các phép quay này, chứng minh rằng chúng đủ để duy trì sự cân bằng và đưa ra cách triển khai bình thường của cây AVL. Chương 12 mô tả các phương pháp cây cân bằng khác với hướng thực hiện cẩn thận hơn.

**4.41 Single Rotation** (Xoay đơn)

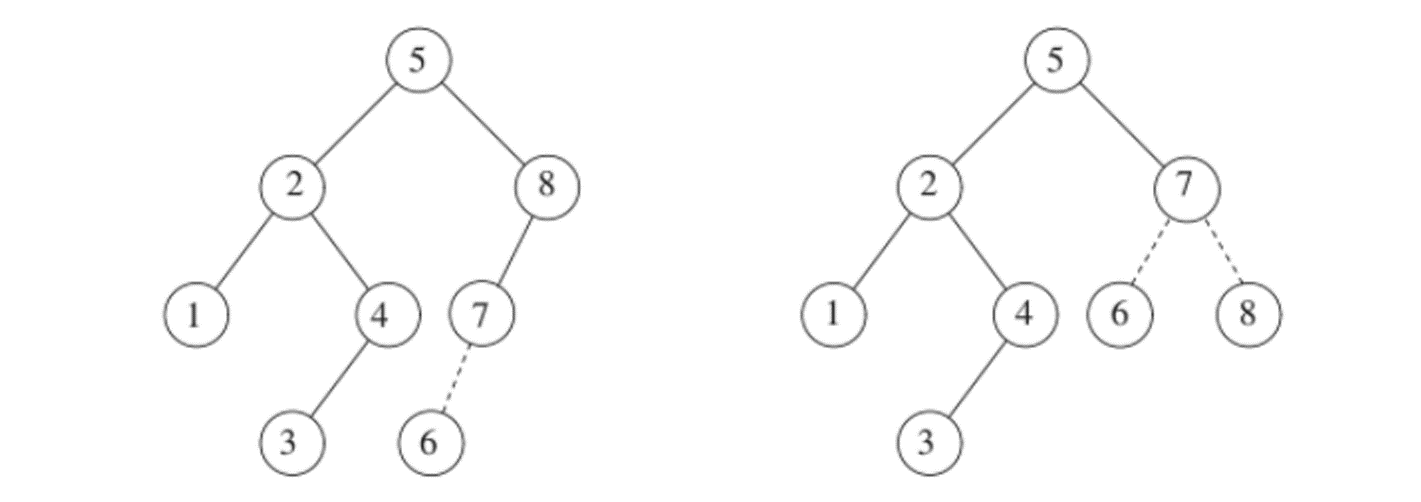
Hình 4.34 cho thấy một vòng quay duy nhất sửa chữa trường hợp 1. Hình trước ở bên trái và hình sau ở bên phải. Hãy cùng chúng tôi phân tích kỹ những gì đang diễn ra. Nút k2 vi phạm thuộc tính số dư AVL vì cây con bên trái của nó sâu hơn hai mức so với cây con bên phải (các đường đứt nét ở giữa biểu đồ đánh dấu các mức). Tình huống được mô tả là trường hợp duy nhất có thể xảy ra 1 kịch bản cho phép k2 thỏa mãn thuộc tính AVL trước khi chèn nhưng lại vi phạm sau đó. Cây con X đã phát triển đến một mức phụ, khiến nó sâu hơn chính xác hai mức so với Z. Y không thể ở cùng mức với X mới vì khi đó k2 sẽ mất cân bằng trước khi chèn và Y không thể ở cùng mức với Z vì khi đó k1 sẽ là nút đầu tiên trên đường dẫn tới gốc vi phạm điều kiện cân bằng AVL.

Kết quả của công việc này, chỉ yêu cầu một vài thay đổi con trỏ, chúng ta có một cây tìm kiếm nhị phân khác là cây AVL. Điều này xảy ra bởi vì X tăng một cấp, Y vẫn ở cùng một mức và Z giảm xuống một cấp. k2 và k1 không chỉ thỏa mãn các yêu cầu của AVL mà chúng còn có các cây con có cùng chiều cao. Hơn nữa, chiều cao mới của toàn bộ cây con hoàn toàn giống với chiều cao của cây con ban đầu trước khi chèn khiến X lớn lên. Do đó, không cần cập nhật thêm độ cao trên đường dẫn đến gốc, và do đó không cần quay thêm nữa. Hình 4.35 cho thấy sau khi chèn nút 6 vào cây AVL ban đầu ở bên trái, nút 8 trở nên không cân bằng. Do đó, chúng tôi thực hiện một vòng quay duy nhất giữa 7 và 8, thu được cây ở bên phải.

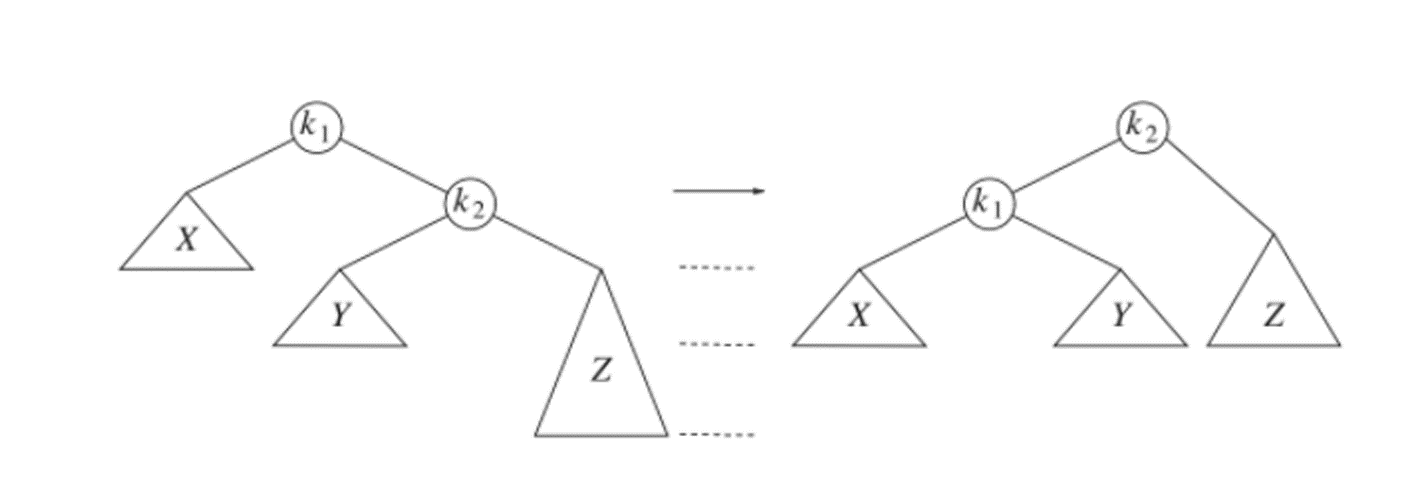
Như chúng ta đã đề cập trước đó, trường hợp 4 đại diện cho một trường hợp đối xứng. Hình 4.36 cho thấy cách áp dụng một phép quay. Hãy để chúng tôi làm việc qua một ví dụ khá dài. Giả sử chúng ta bắt đầu với một cây AVL trống ban đầu và chèn các mục 3, 2, 1, và sau đó là 4 đến 7 theo thứ tự tuần tự. Sự cố đầu tiên xảy

 ra khi đến lúc chèn mục 1 vì AVL

**Hình 4.34** Xoay đơn để sửa trường hợp 1

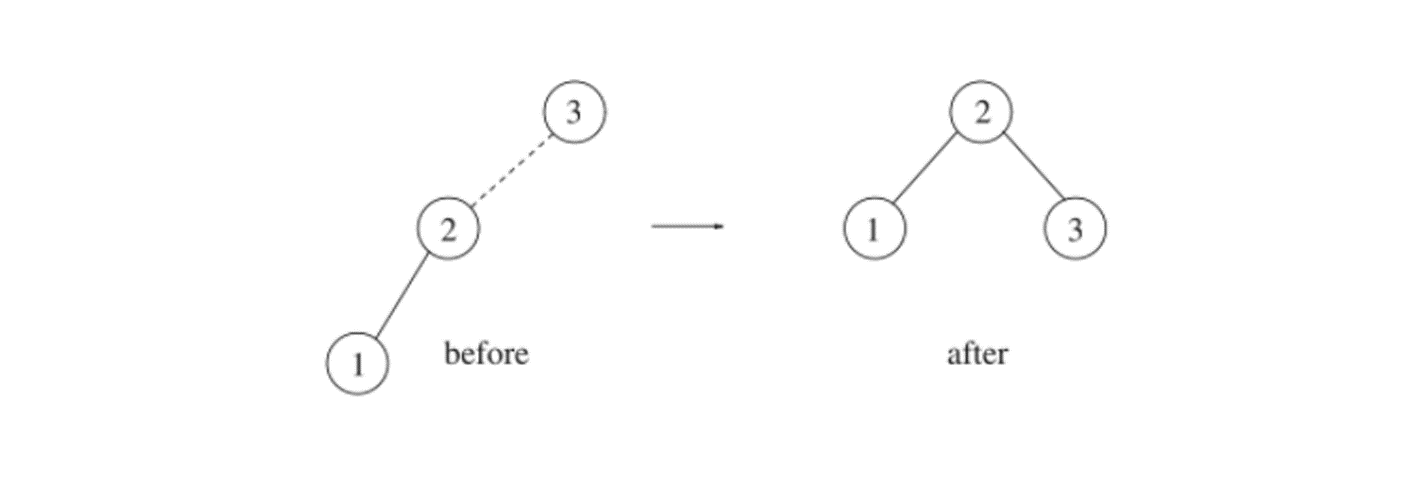


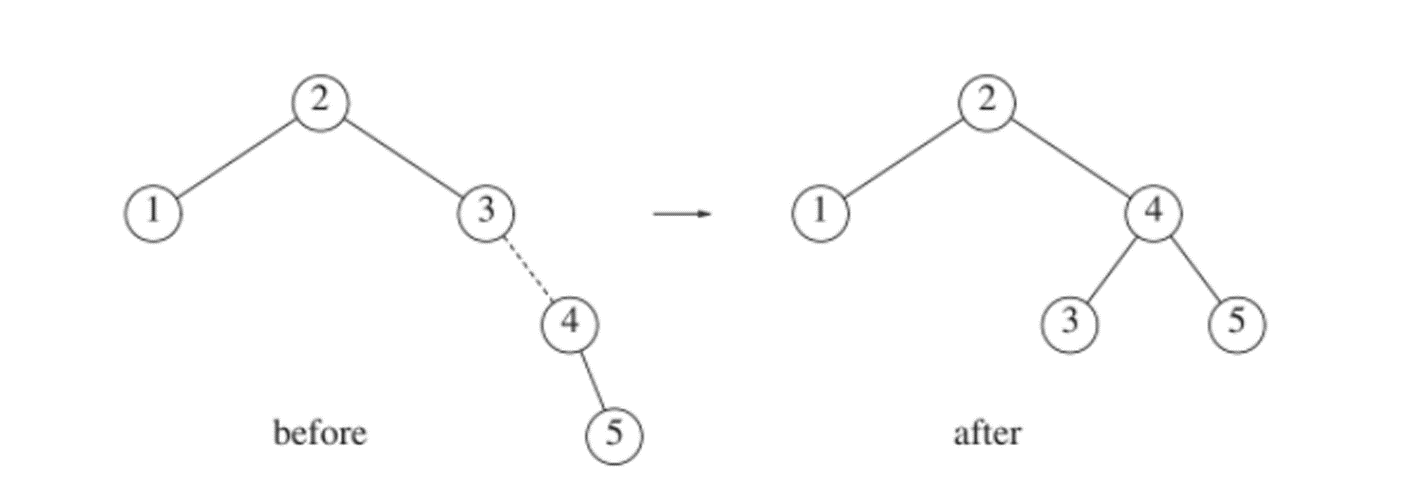
**Hình 4.35** Thuộc tính AVL bị phá hủy bằng cách chèn 6, sau đó được cố định bằng một vòng quay

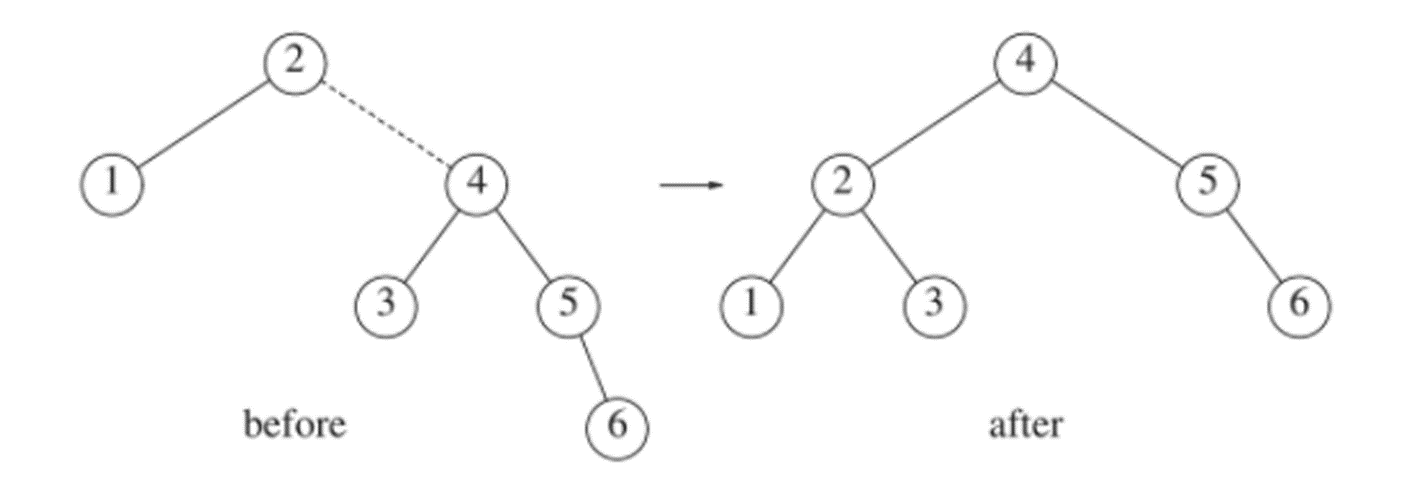


**Hình 4.36** Trường hợp sửa lỗi xoay đơn

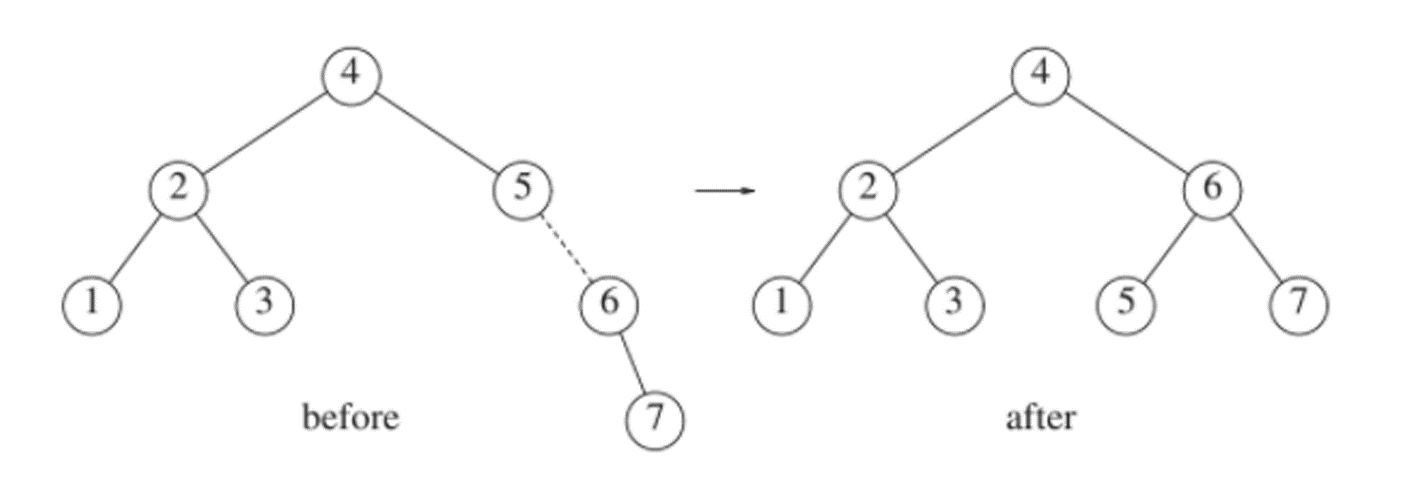
tài sản bị xâm phạm tận gốc. Chúng tôi thực hiện một phép quay giữa gốc và con trái của nó để khắc phục sự cố. Đây là những cây trước và sau:



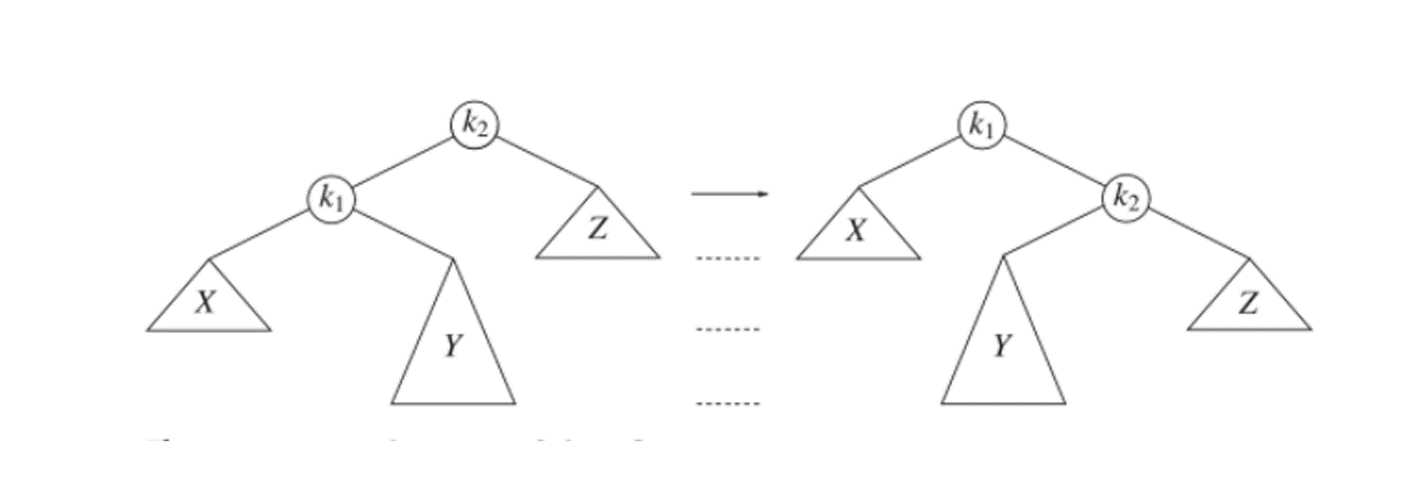
Một đường đứt nét nối hai nút là đối tượng của phép quay. Tiếp theo, chúng ta chèn 4, điều này không gây ra vấn đề gì, nhưng việc chèn 5 tạo ra vi phạm tại nút 3 được cố định bằng một lần quay. Bên cạnh sự thay đổi cục bộ gây ra bởi phép quay, người lập trình phải nhớ rằng phần còn lại của cây phải được thông báo về sự thay đổi này. Ở đây, điều này có nghĩa là con bên phải của 2 phải được đặt lại thành liên kết thành 4 thay vì 3. Việc quên làm như vậy rất dễ dàng và sẽ phá hủy cây (4 sẽ không thể truy cập được).

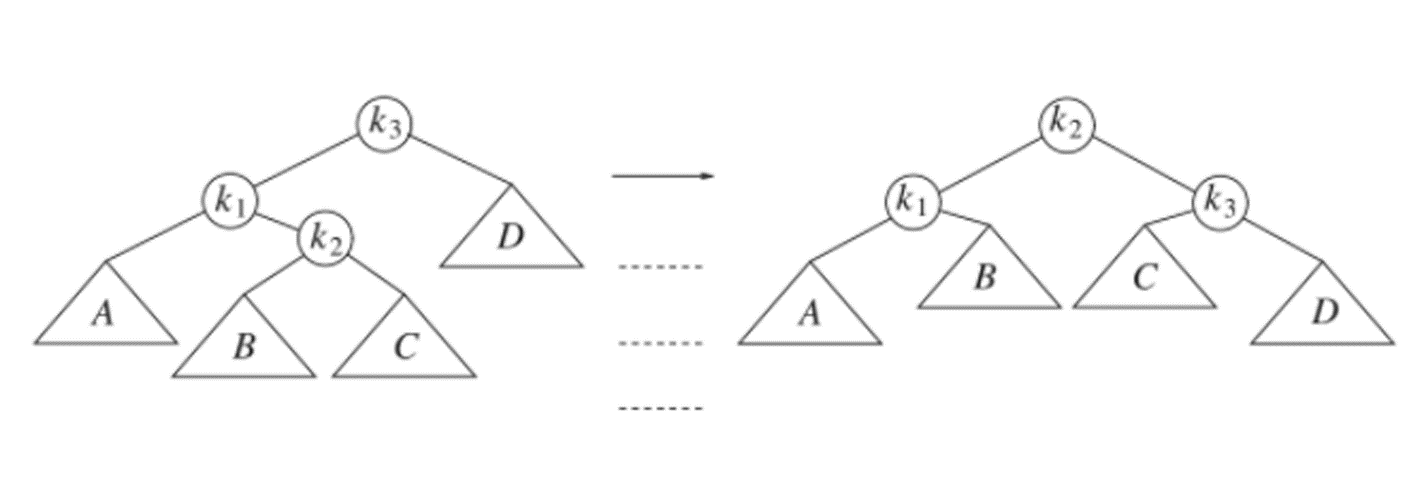
Tiếp theo, chúng tôi chèn 6. Điều này gây ra vấn đề cân bằng ở gốc, vì cây con bên trái của nó có chiều cao là 0 và cây con bên phải của nó sẽ có chiều cao 2. Do đó, chúng tôi thực hiện một phép quay ở gốc giữa 2 và 4.

Phép quay được thực hiện bằng cách biến 2 trở thành cây con bên trái ban đầu của 4 và 4 thành cây con bên phải mới của 2. Mọi mục trong cây con này phải nằm giữa 2 và 4, do đó, phép biến đổi này có ý nghĩa. Mục tiếp theo chúng tôi chèn là 7, gây ra một vòng quay khác:

**4.4.2 Double Rotation** (Xoay đôi)

Thuật toán được mô tả ở trên có một vấn đề: Như Hình 4.37 cho thấy, nó không hoạt động đối với trường hợp 2 hoặc 3. Vấn đề là cây con Y quá sâu và một phép quay không làm cho nó bớt sâu hơn. Phép quay kép giải quyết vấn đề được thể hiện trong Hình 4.38.

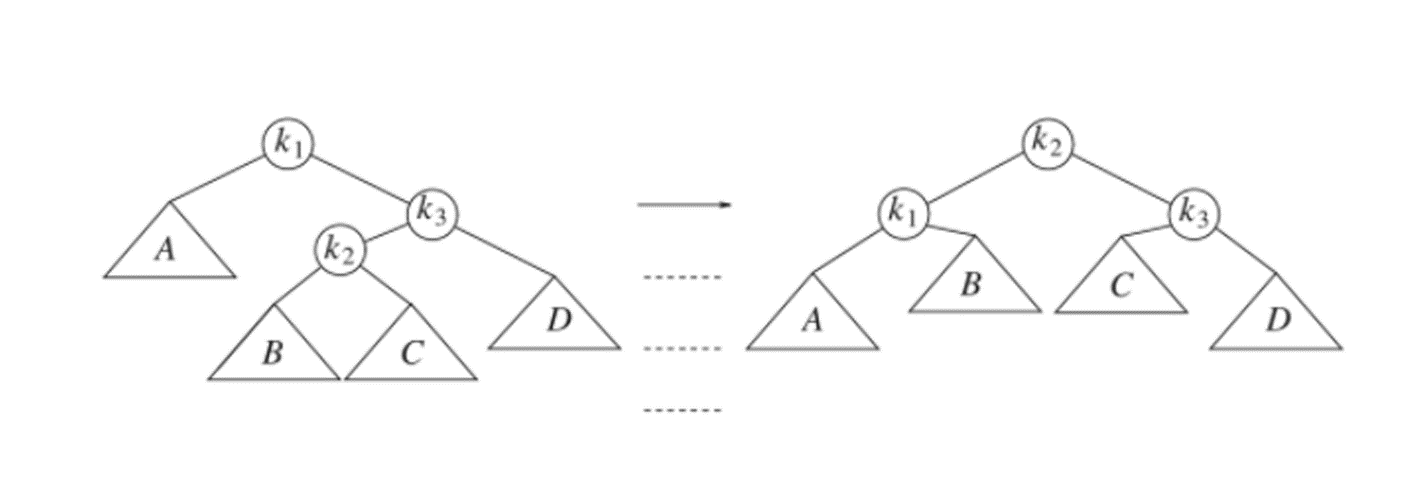
 Thực tế là cây con Y trong Hình 4.37 đã có một mục được chèn vào nó đảm bảo rằng nó không phải là mục. Do đó, chúng ta có thể giả định rằng nó có một gốc và hai cây con.

**Hình 4.37** Xoay đơn không khắc phục được trường hợp 2

**Hình 4.38** Xoay đôi trái-phải để sửa trường hợp 2

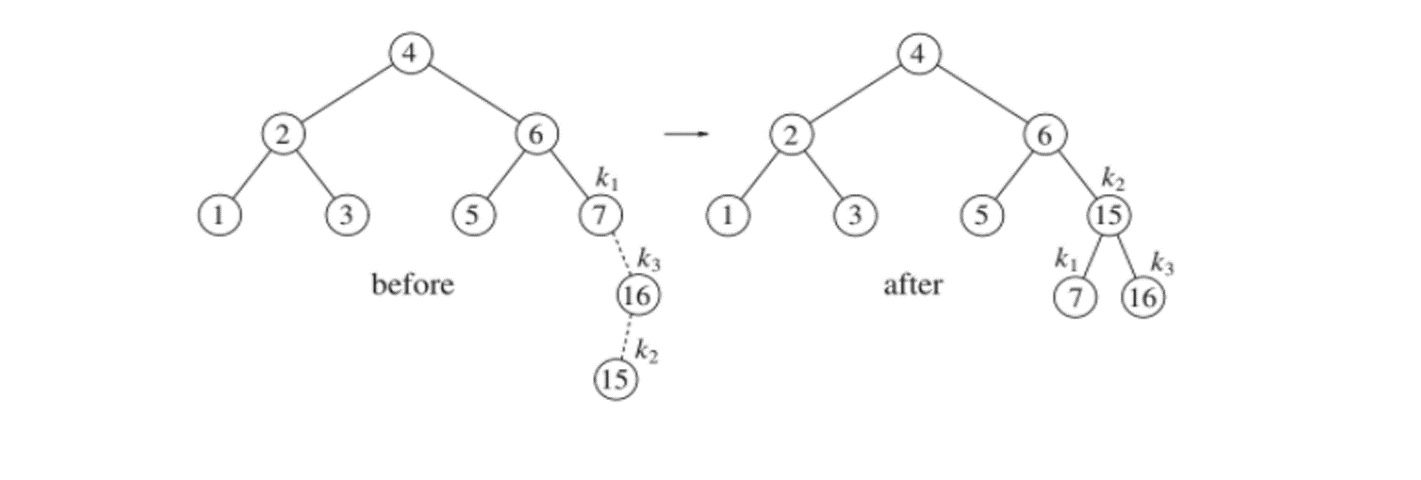
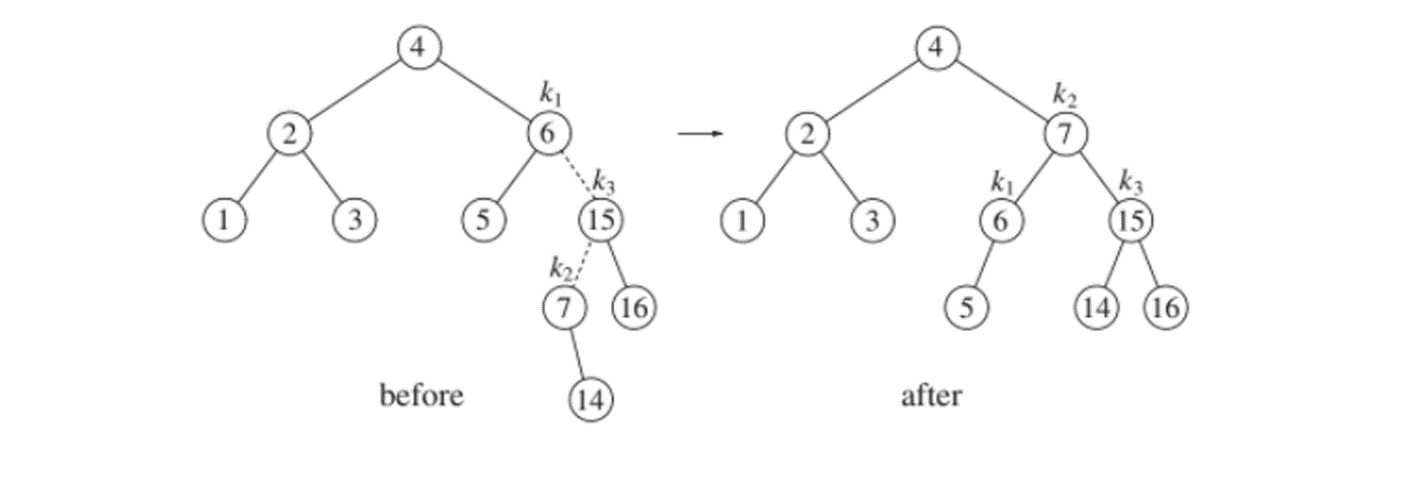
cây có thể được xem như bốn cây con được kết nối bởi ba nút. Như sơ đồ gợi ý, chính xác một trong số cây B hoặc C sâu hơn hai cấp so với D (trừ khi tất cả đều trống), nhưng chúng ta không thể chắc chắn cái nào. Hóa ra không thành vấn đề; trong Hình 4.38, cả B và C đều được vẽ ở mức 1 \* (1/2) bên dưới D.

Để cân bằng lại, chúng ta thấy rằng chúng ta không thể để k3 làm gốc, và phép quay giữa k3 và k1 được thể hiện trong Hình 4.37 không hoạt động, vì vậy cách thay thế duy nhất là đặt k2 làm gốc mới. Điều này buộc k1 là con trái của k2 và k3 là con phải của nó và nó cũng hoàn toàn xác định vị trí kết quả của bốn cây con. Dễ dàng nhận thấy rằng cây kết quả thỏa mãn thuộc tính cây AVL, và như trường hợp với phép quay đơn, nó khôi phục chiều cao về mức trước khi chèn, do đó đảm bảo rằng tất cả việc tái cân bằng và cập nhật chiều cao đã hoàn tất. Hình 4.39 cho thấy rằng trường hợp đối xứng 3 cũng có thể được sửa

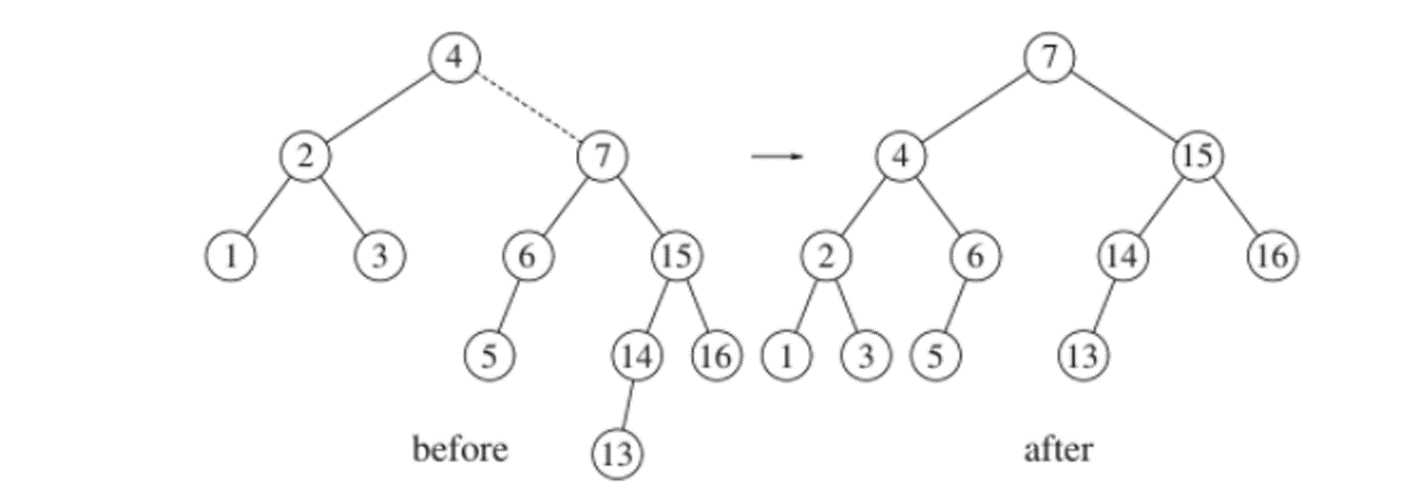
**Hình 4.39** Xoay đôi phải-trái để sửa trường hợp 3

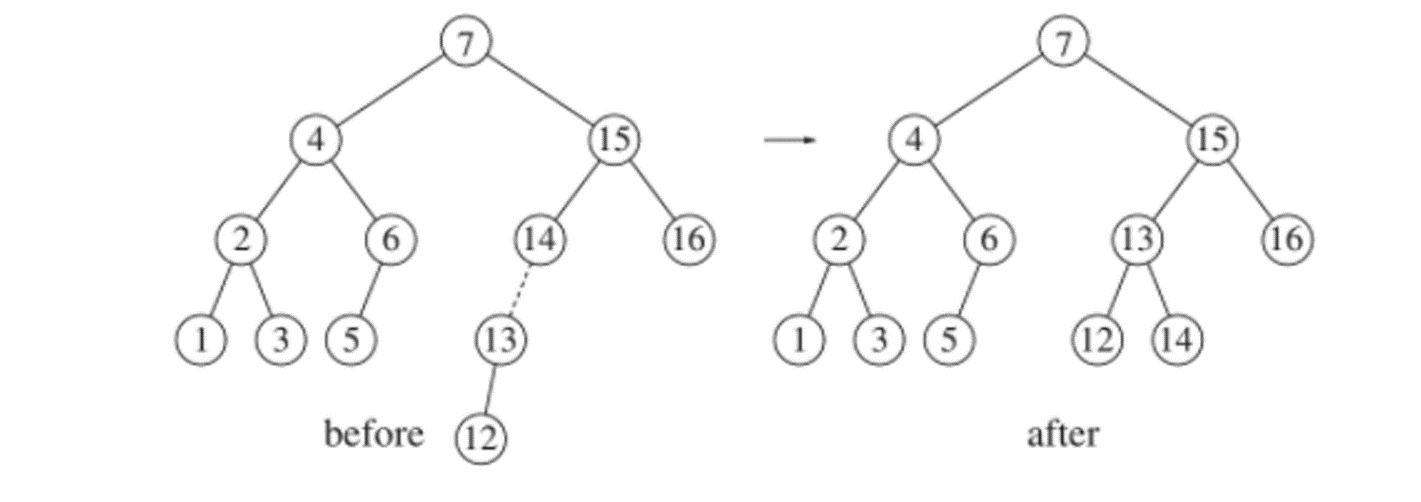
bằng một vòng quay kép. Trong cả hai trường hợp, hiệu ứng giống như xoay vòng giữa con và cháu của α, sau đó giữa α và con mới của nó.

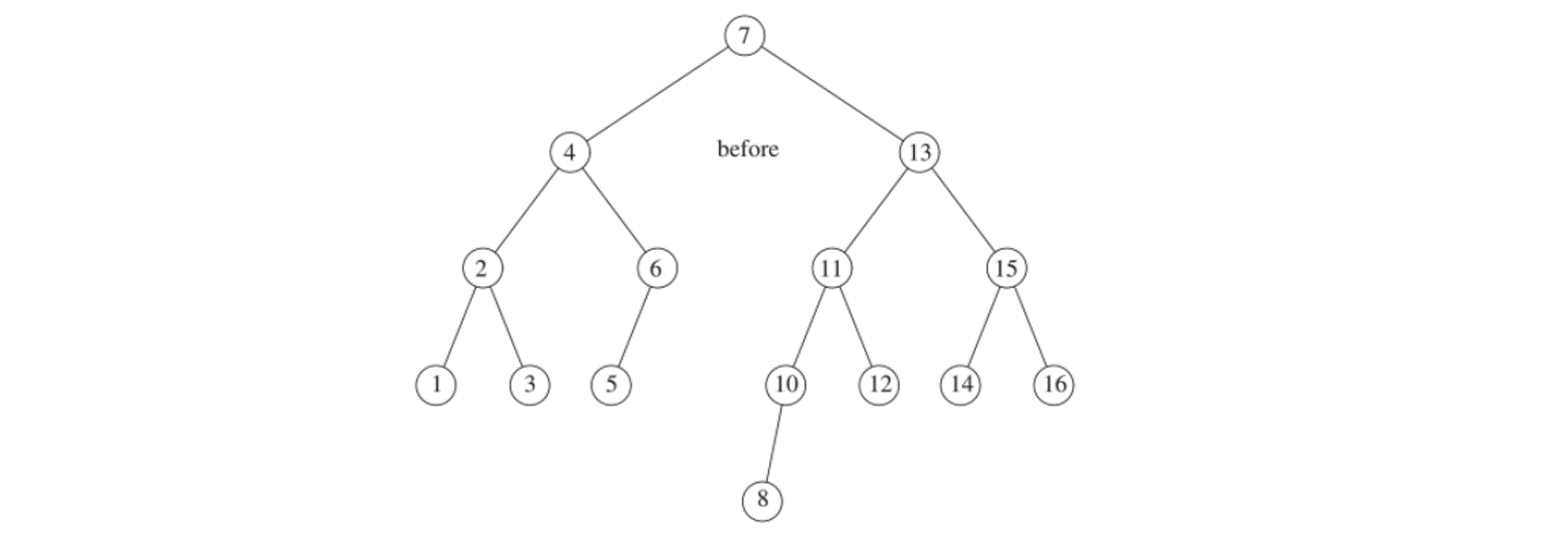
Chúng ta sẽ tiếp tục ví dụ trước bằng cách chèn 10 đến 16 theo thứ tự ngược lại, tiếp theo là 8 và sau đó là 9. Chèn 16 rất dễ dàng, vì nó không phá hủy thuộc tính cân bằng, nhưng việc chèn 15 gây ra sự mất cân bằng chiều cao ở nút 7. Đây là trường hợp 3, được giải quyết bằng cách quay đôi phải-trái. Trong ví dụ của chúng ta, phép quay kép phải-trái sẽ liên quan đến 7, 16 và 15. Trong trường hợp này, k1 là nút có mục 7, k3 là nút có mục 16 và k2 là nút có mục 15. Các cây con A , B, C và D trống.

Tiếp theo, chúng tôi chèn 14, cũng yêu cầu quay đôi. Ở đây, phép quay kép sẽ khôi phục cây một lần nữa là phép quay kép phải-trái sẽ liên quan đến 6, 15 và 7. Trong trường hợp này, k1 là nút với mục 6, k2 là nút với mục 7 và k3 là nút có mục 15. Cây con A là cây bắt nguồn từ nút có mục 5; cây con B là cây con trống ban đầu là cây con bên trái của nút có mục 7, cây con C là cây có gốc tại nút có mục 14 và cuối cùng, cây con D là cây có gốc tại nút có mục 16.

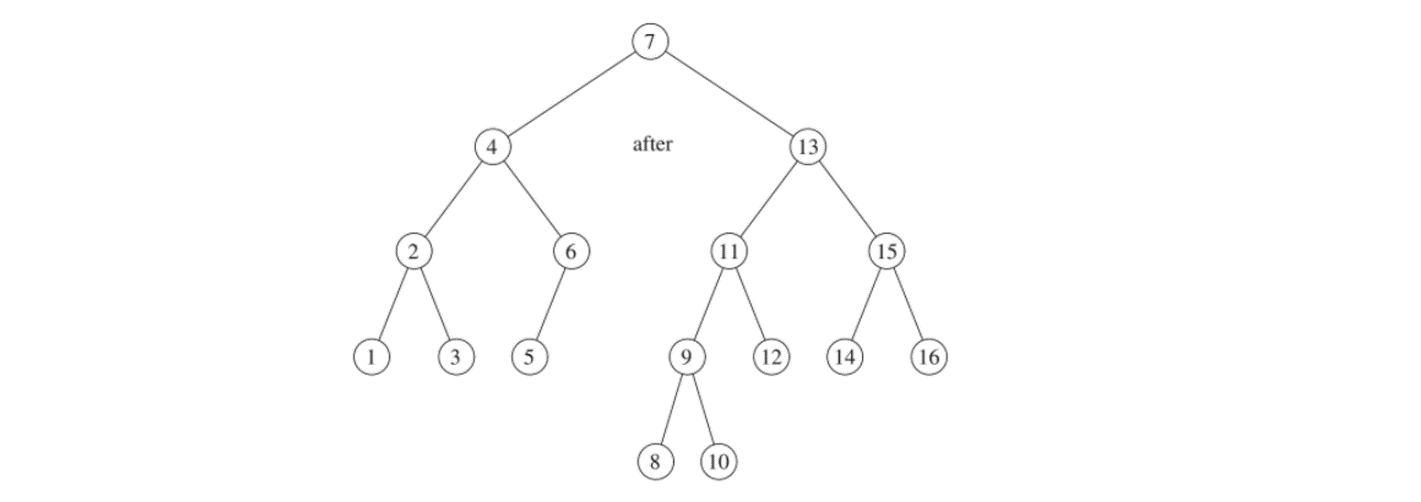
Nếu 13 bây giờ được chèn vào, có sự mất cân bằng ở gốc. Vì 13 không nằm giữa 4 và 7, chúng ta biết rằng phép quay đơn sẽ hoạt động.

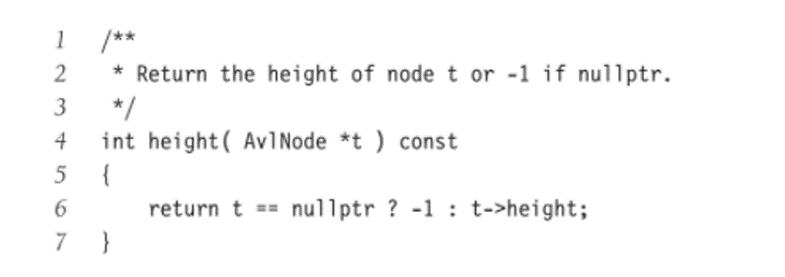
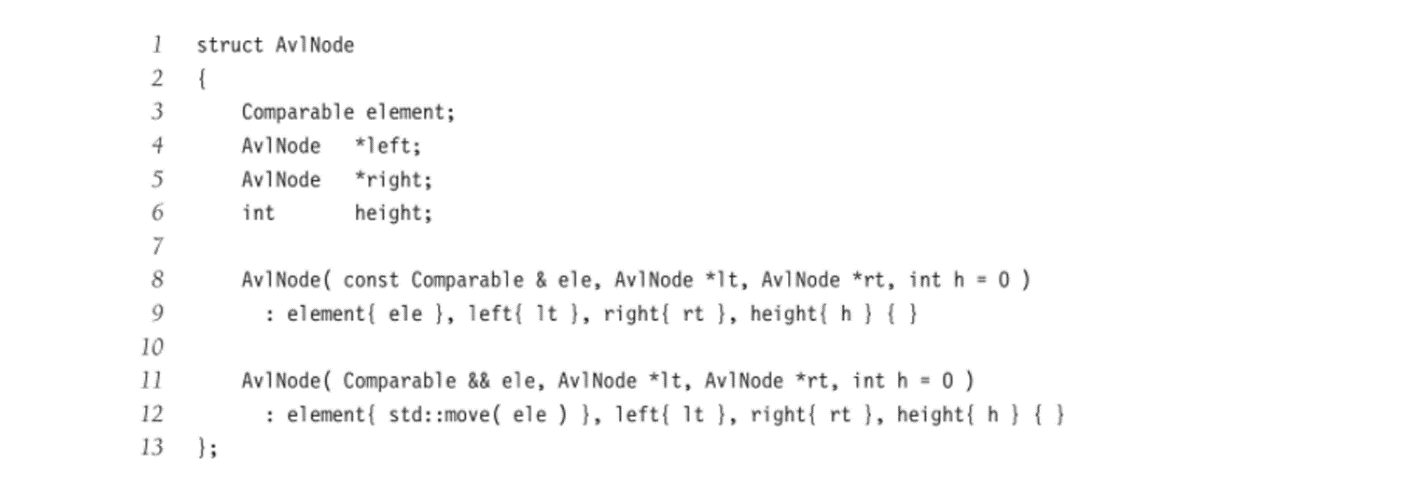


Việc chèn 12 cũng sẽ yêu cầu một vòng quay:

Để chèn 11, cần thực hiện một phép xoay đơn và điều này cũng đúng cho lần chèn tiếp theo của 10. Chúng tôi chèn 8 mà không cần xoay, tạo ra một cây cân bằng gần như hoàn hảo:

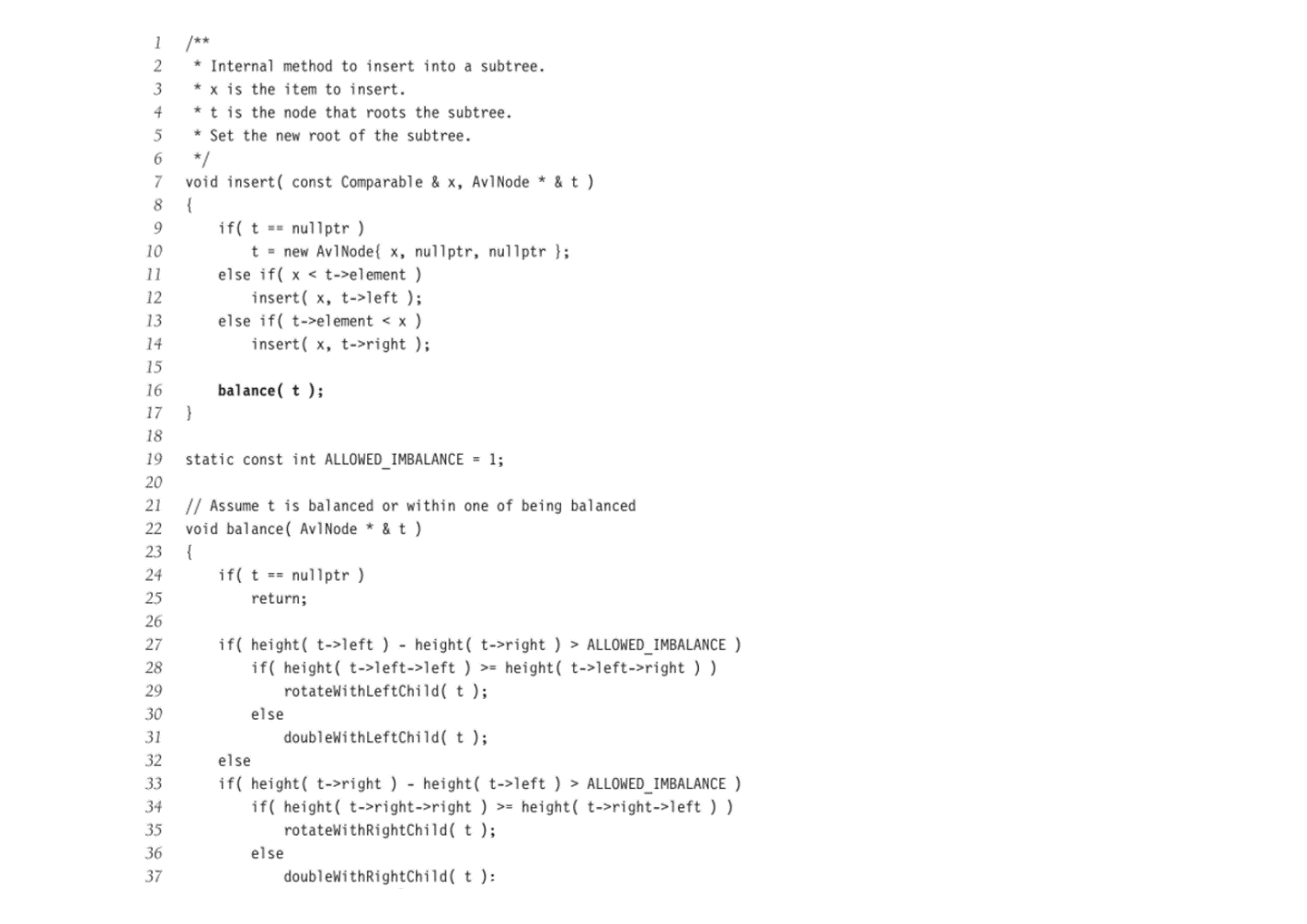
Cuối cùng, chúng tôi sẽ chèn 9 để hiển thị trường hợp đối xứng của phép quay kép. Lưu ý rằng 9 làm cho nút chứa 10 trở nên không cân bằng. Vì 9 nằm trong khoảng từ 10 đến 8 (là con của 10 trên đường dẫn đến 9), cần thực hiện một phép quay kép, tạo ra cây sau:

Hãy để chúng tôi tóm tắt những gì xảy ra. Các chi tiết lập trình khá đơn giản ngoại trừ một số trường hợp. Để chèn một nút mới với mục X vào cây AVL T, chúng tôi chèn đệ quy X vào cây con thích hợp của T (chúng ta hãy gọi đây là TLR). Nếu chiều cao của TLR không thay đổi, thì chúng ta đã hoàn tất. Ngược lại, nếu sự mất cân bằng chiều cao xuất hiện trong T, chúng tôi thực hiện xoay đơn hoặc xoay đôi thích hợp tùy thuộc vào X và các mục trong T và TLR, cập nhật chiều cao (tạo kết nối từ phần còn lại của cây ở trên) và chúng tôi đã hoàn tất. Vì một vòng quay luôn là đủ, một phiên bản không đệ quy được mã hóa cẩn thận thường nhanh hơn phiên bản đệ quy, nhưng trên các trình biên dịch hiện đại, sự khác biệt không còn đáng kể như trước đây.

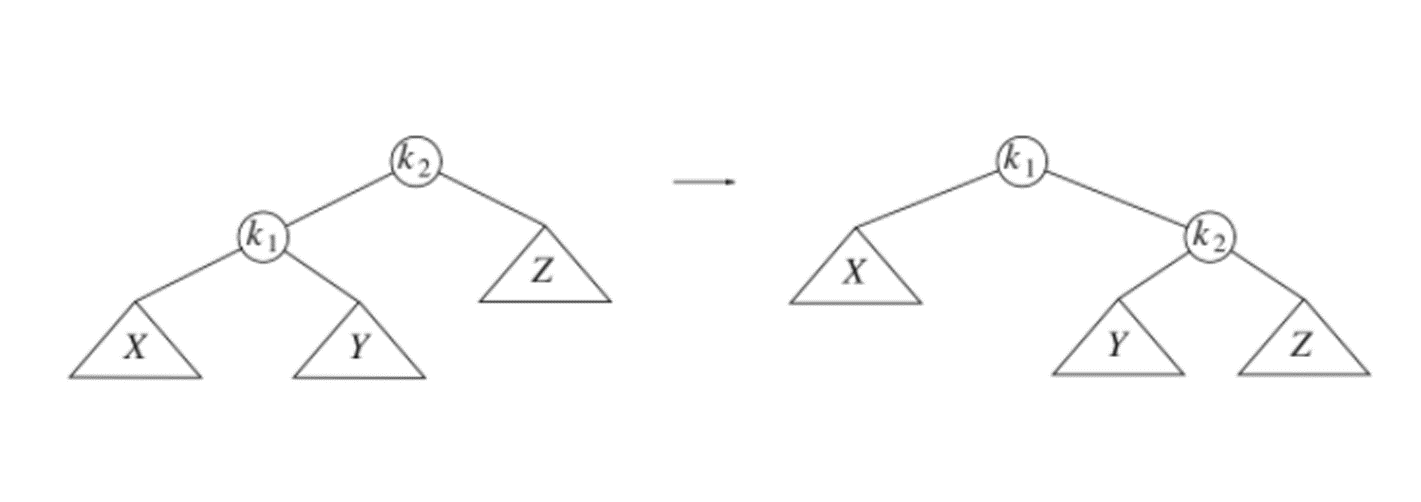
**Hình 4.40** Khai báo nút cho cây AVL

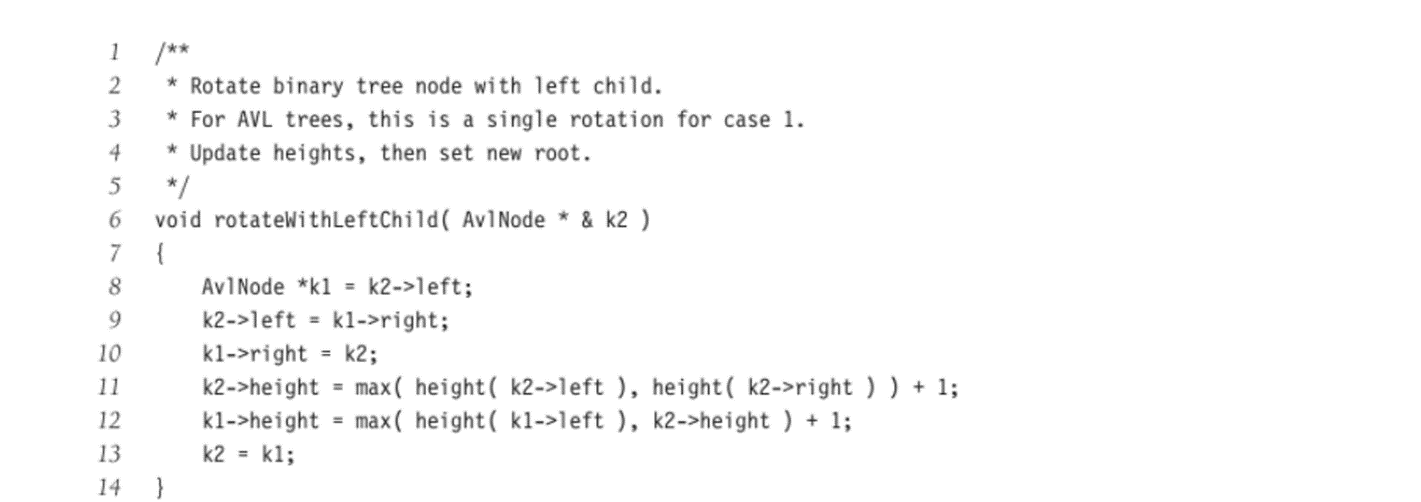
**Hình 4.41** Hàm tính toán chiều cao của nút AVL

Với tất cả những điều này, chúng tôi đã sẵn sàng để viết các quy trình AVL. Chúng tôi hiển thị một số mã ở đây; phần còn lại là trực tuyến. Đầu tiên, chúng ta cần lớp AvlNode. Điều này được cho trong Hình 4.40. Chúng ta cũng cần một hàm nhanh để trả về chiều cao của một nút. Chức năng này là cần thiết để xử lý trường hợp khó chịu của một con trỏ nullptr. Điều này được thể hiện trong Hình 4.41

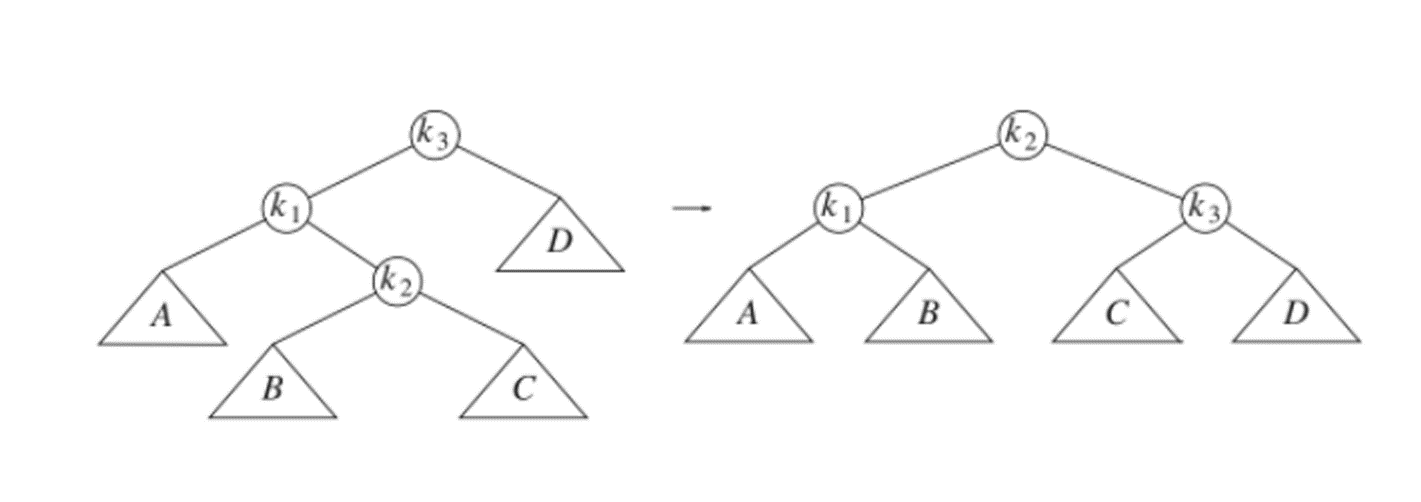
Vì việc xóa trong cây tìm kiếm nhị phân có phần phức tạp hơn việc chèn, nên người ta có thể cho rằng việc xóa trong cây AVL cũng phức tạp hơn. Trong một thế giới hoàn hảo, người ta có thể hy vọng rằng quy trình xóa trong Hình 4.26 có thể dễ dàng được sửa đổi bằng cách thay đổi dòng cuối cùng để trả về sau khi gọi phương thức số dư, như đã thực hiện đối với việc chèn.

**Hình 4.42** Chèn vào cây AVL

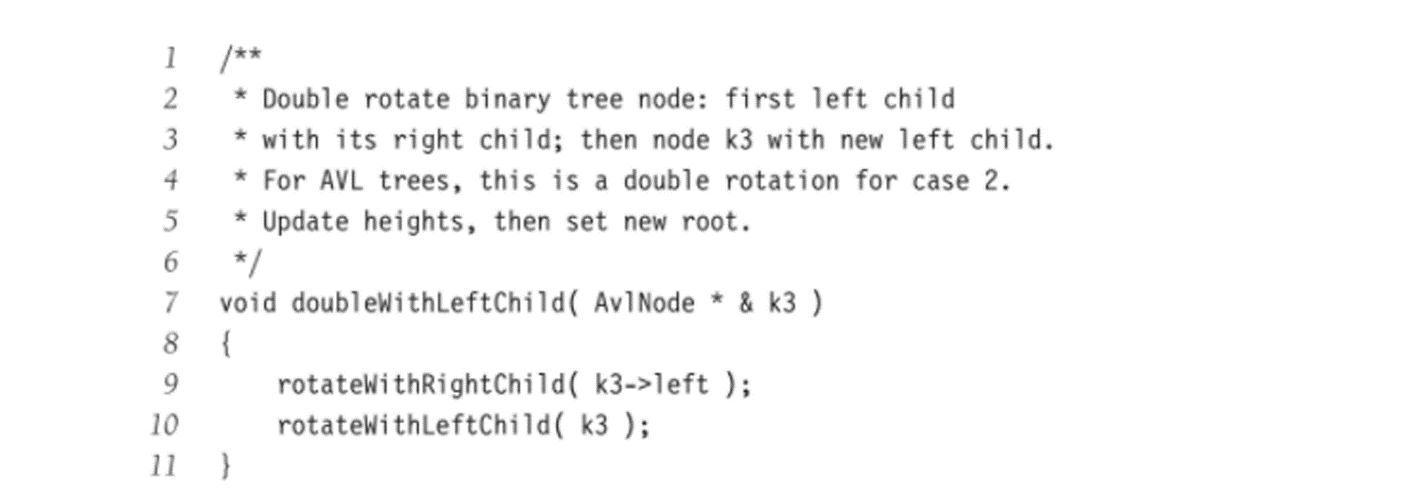
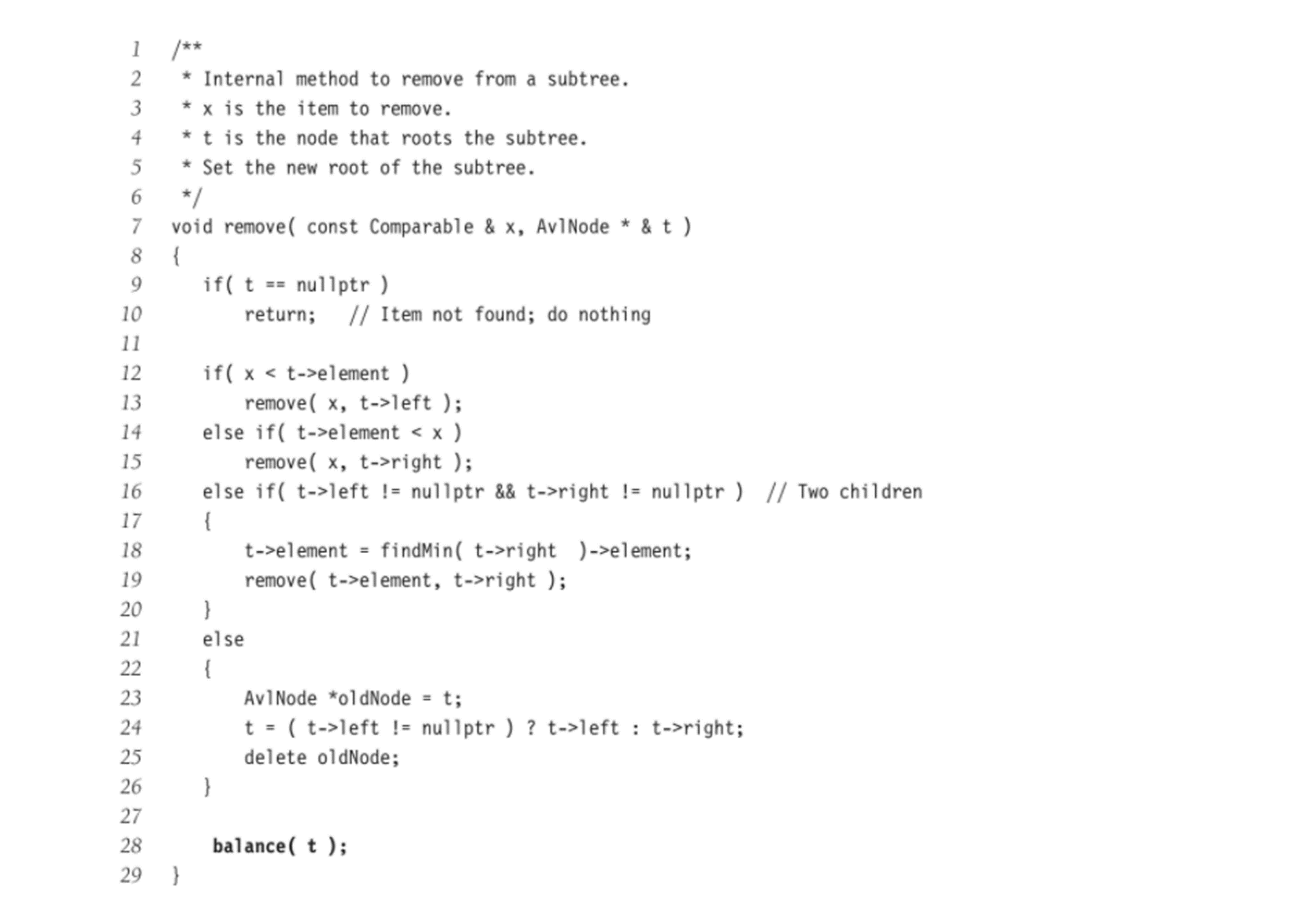


**Hình 4.43** Vòng quay đơn

**Hình 4.44** Quy trình thực hiện một vòng quay



**Hình 4.45** Xoay đôi

**Hình 4.46** Quy trình thực hiện quay kép

**Hình 4.47** Xóa trong cây AVL

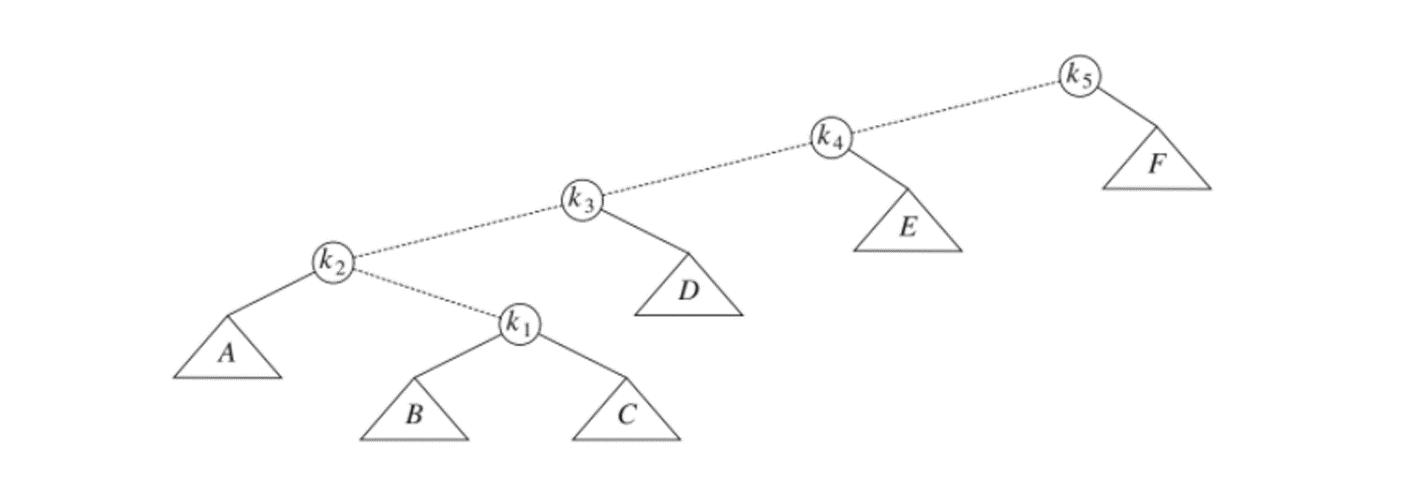
**4.5 Splay Trees** (Phát cây)

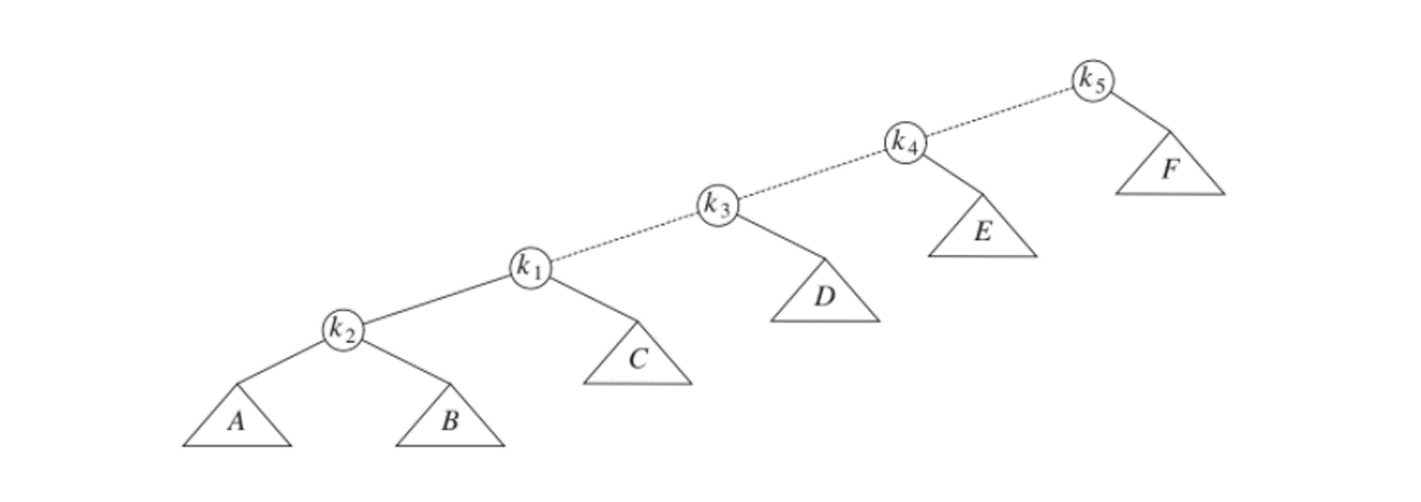
Bây giờ chúng ta mô tả một cấu trúc dữ liệu tương đối đơn giản được gọi là cây splay mà cho rằng M bất kỳ phép toán nào liên tiếp trên cây bắt đầu từ một cây trống sẽ mất nhiều nhất là O (M logN) thời gian. Mặc dù đảm bảo này không loại trừ khả năng bất kỳ hoạt động đơn lẻ nào có thể mất (N) thời gian và do đó ràng buộc không mạnh bằng ràng buộc trong trường hợp xấu nhất O (logN) trên mỗi hoạt động, nhưng hiệu quả thực là như nhau: Không có trình tự đầu vào xấu. Nói chung, khi một chuỗi M hoạt động có tổng thời gian chạy trong trường hợp xấu nhất là O (Mf (N)), chúng ta nói rằng thời gian chạy phân bổ là O (f (N)). Do đó, một cây splay có chi phí khấu hao mỗi lần hoạt động là O (logN). Trong một chuỗi dài các hoạt động, một số có thể mất nhiều hơn, một số ít hơn.

Cây ghép dựa trên thực tế là thời gian trong trường hợp xấu nhất O (N) trên mỗi thao tác đối với cây tìm kiếm nhị phân không phải là xấu, miễn là nó xảy ra tương đối thường xuyên. Bất kỳ một truy cập nào, ngay cả khi mất (N), vẫn có khả năng cực kỳ nhanh. Vấn đề với cây tìm kiếm nhị phân là có thể xảy ra một chuỗi toàn bộ các truy cập không hợp lệ. Thời gian chạy tích lũy sau đó trở nên đáng chú ý. Cấu trúc dữ liệu cây tìm kiếm với thời gian trong trường hợp xấu nhất là O (N), nhưng đảm bảo tối đa là O (M logN) cho bất kỳ hoạt động liên tiếp nào của M, chắc chắn là thỏa đáng, vì không có chuỗi nào xấu.

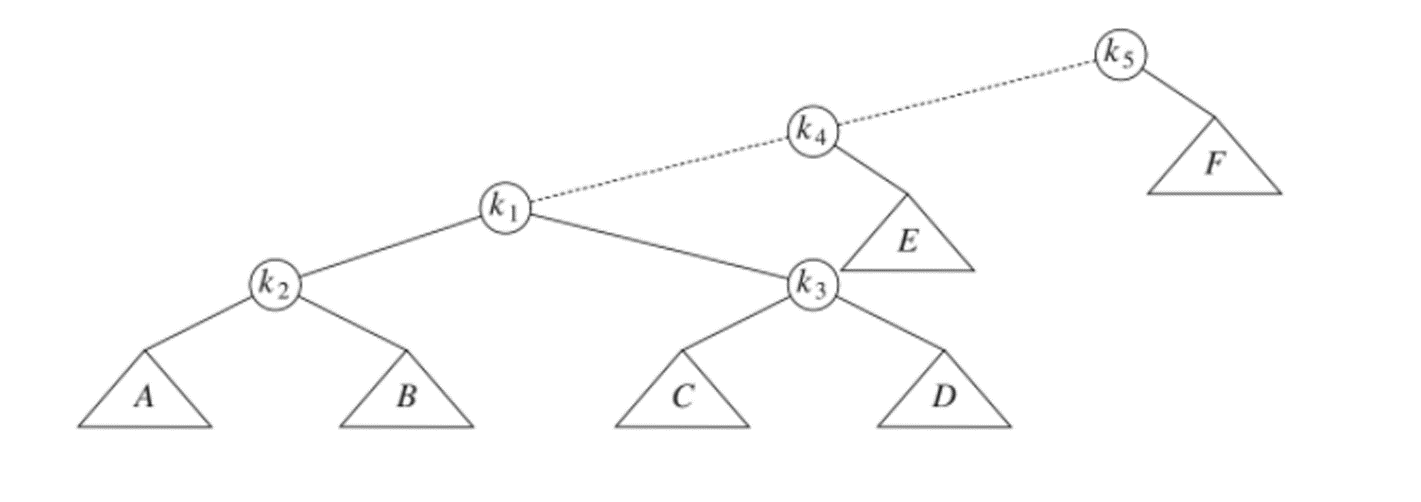
Ý tưởng cơ bản của cây splay là sau khi một nút được truy cập, nó được đẩy về gốc bởi một loạt các phép quay cây AVL. Lưu ý rằng nếu một nút nằm sâu, có nhiều nút trên đường dẫn cũng tương đối sâu và bằng cách tái cấu trúc, chúng tôi có thể làm cho việc truy cập trong tương lai rẻ hơn trên tất cả các nút này. Do đó, nếu nút quá sâu, thì chúng ta muốn việc tái cấu trúc này có tác dụng phụ là cân bằng cây (ở một mức độ nào đó).

**4.5.1 A Simple Idea (That Does Not Work)** - Một ý tưởng đơn giản (Điều đó không hiệu quả)

Một cách để thực hiện tái cấu trúc được mô tả ở trên là thực hiện các phép quay đơn lẻ, từ dưới lên. Điều này có nghĩa là chúng tôi xoay mọi nút trên đường dẫn truy cập với cha của nó. Ví dụ, hãy xem xét điều gì xảy ra sau một lần truy cập (một tìm thấy) trên k1 trong cây sau:

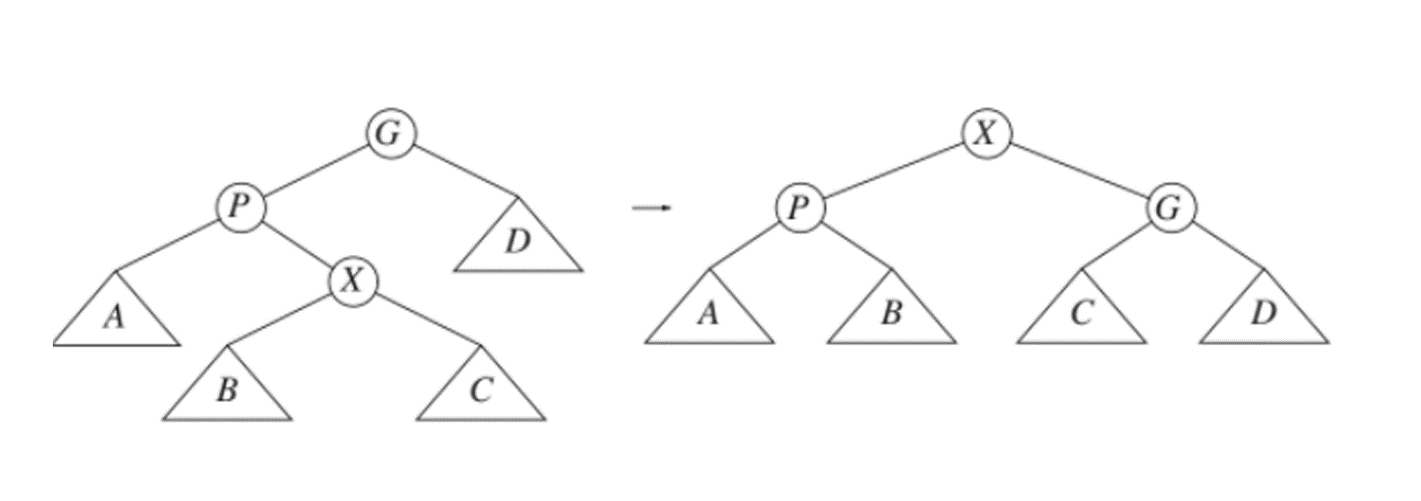
Đường dẫn truy cập bị gạch ngang. Đầu tiên, chúng ta sẽ thực hiện một phép quay giữa k1 và cha của nó, thu được cây sau:

Sau đó, chúng ta xoay giữa k1 và k3, thu được cây tiếp theo:

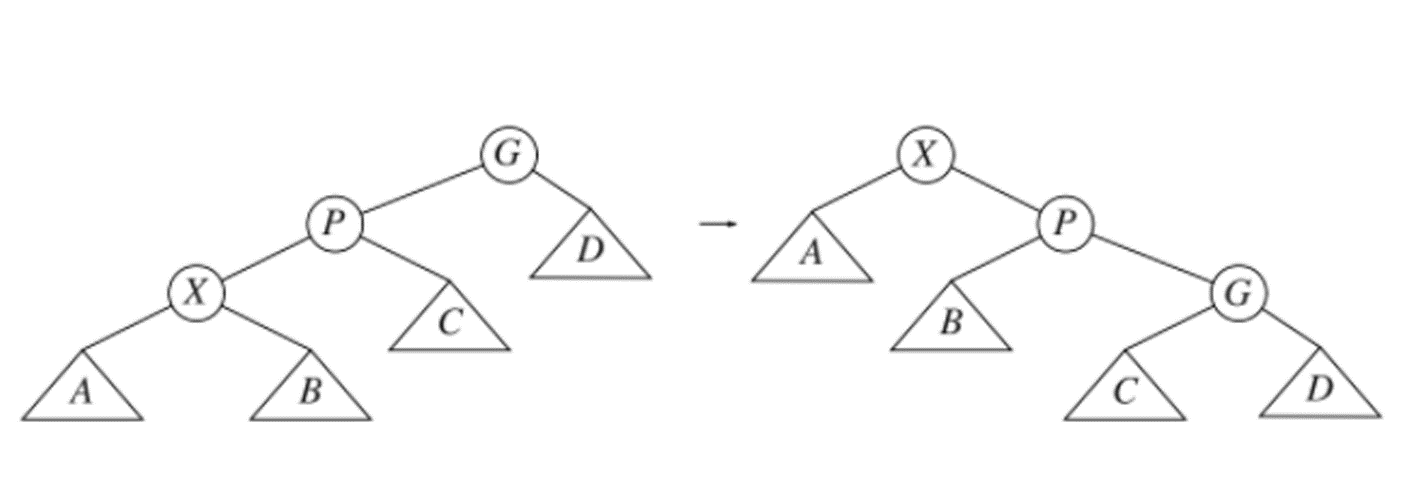
Sau đó, hai phép quay khác được thực hiện cho đến khi chúng ta đến gốc:

Các phép quay này có tác dụng đẩy k1 về tận gốc, để việc truy cập sau này trên k1 được dễ dàng (trong một thời gian). Thật không may, nó đã đẩy một nút khác (k3) xuống sâu gần như k1 trước đây. Một quyền truy cập trên nút đó sau đó sẽ đẩy một nút khác vào sâu, v.v. Mặc dù chiến lược này làm cho các truy cập trong tương lai của k1 rẻ hơn, nhưng nó không cải thiện đáng kể tình hình đối với các nút khác trên đường truy cập (ban đầu). Nó chỉ ra rằng có thể chứng minh rằng sử dụng chiến lược này, có một chuỗi M hoạt động đòi hỏi (M · N) thời gian, vì vậy ý ​​tưởng này không đủ tốt. Cách đơn giản nhất để chỉ ra điều này là xem xét cây được tạo thành bằng cách chèn các phím 1, 2, 3, ..., N vào một cây trống ban đầu (làm ví dụ này). Điều này tạo ra một cây chỉ bao gồm các con còn lại. Tuy nhiên, điều này không hẳn là xấu, vì thời gian để xây dựng cây này là tổng số O (N). Phần tồi tệ là việc truy cập vào nút với khóa 1 mất N đơn vị thời gian, trong đó mỗi nút trên đường dẫn truy cập được tính là một đơn vị. Sau khi các phép quay hoàn tất, một lần truy cập vào nút với phím 2 mất N đơn vị thời gian, phím 3 mất N - 1 đơn vị, v.v. Tổng để truy cập tất cả các khóa theo thứ tự là N + N i = 2 i = (N2). Sau khi chúng được truy cập, cây sẽ trở lại trạng thái ban đầu và chúng ta có thể lặp lại trình tự.

**4.5.2 Spaying** (Đánh đòn)

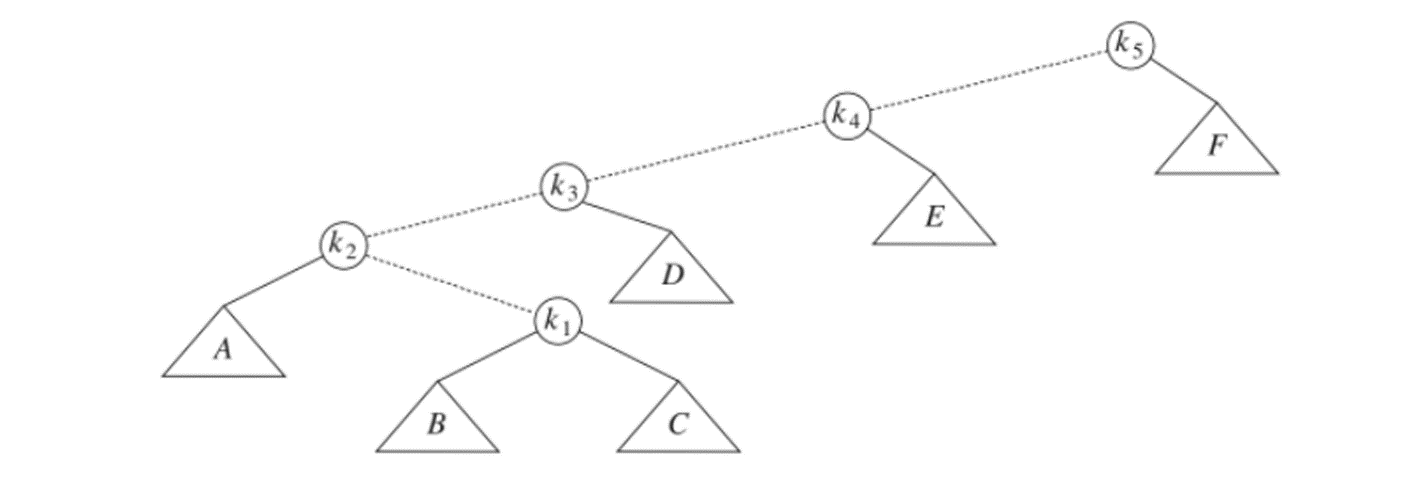
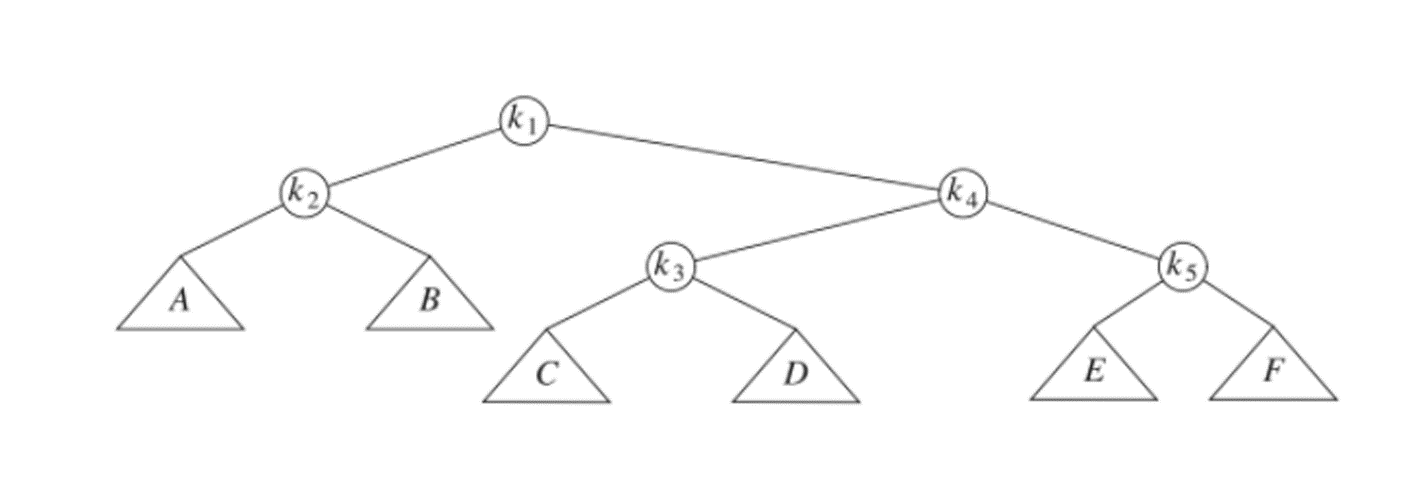
Chiến lược ghép tương tự như ý tưởng xoay ở trên, ngoại trừ việc chúng tôi chọn lọc hơn một chút về cách thực hiện các phép quay. Chúng tôi sẽ vẫn xoay từ dưới lên dọc theo quyền truy cập

**Hình 4.48** Zig-zag

**Hình 4.49** Zig-zag

Con đường. Gọi X là một nút (không phải nút gốc) trên đường dẫn truy cập mà chúng ta đang xoay. Nếu gốc của X là gốc của cây, chúng ta chỉ xoay X và gốc. Đây là vòng quay cuối cùng dọc theo đường dẫn vào. Nếu không, X có cả cha mẹ (P) và ông bà (G), và có hai trường hợp, cộng với các đối xứng, cần xem xét. Trường hợp đầu tiên là trường hợp zig-zag (xem Hình 4.48). Ở đây X là con phải và P là con trái (hoặc ngược lại). Nếu đúng như vậy, chúng tôi thực hiện quay kép, giống hệt như quay kép AVL. Nếu không, chúng ta có trường hợp zig-zig: X và P đều là con trái (hoặc, trong trường hợp đối xứng, cả hai đều là con phải). Trong trường hợp đó, chúng ta biến đổi cây bên trái của Hình 4.49 thành cây bên phải.

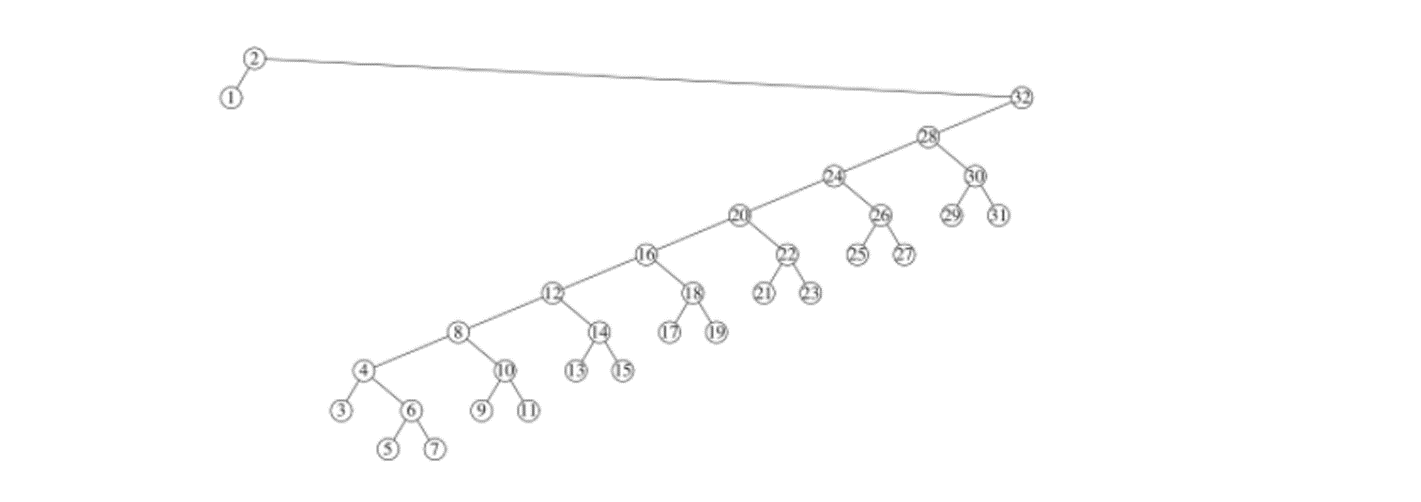
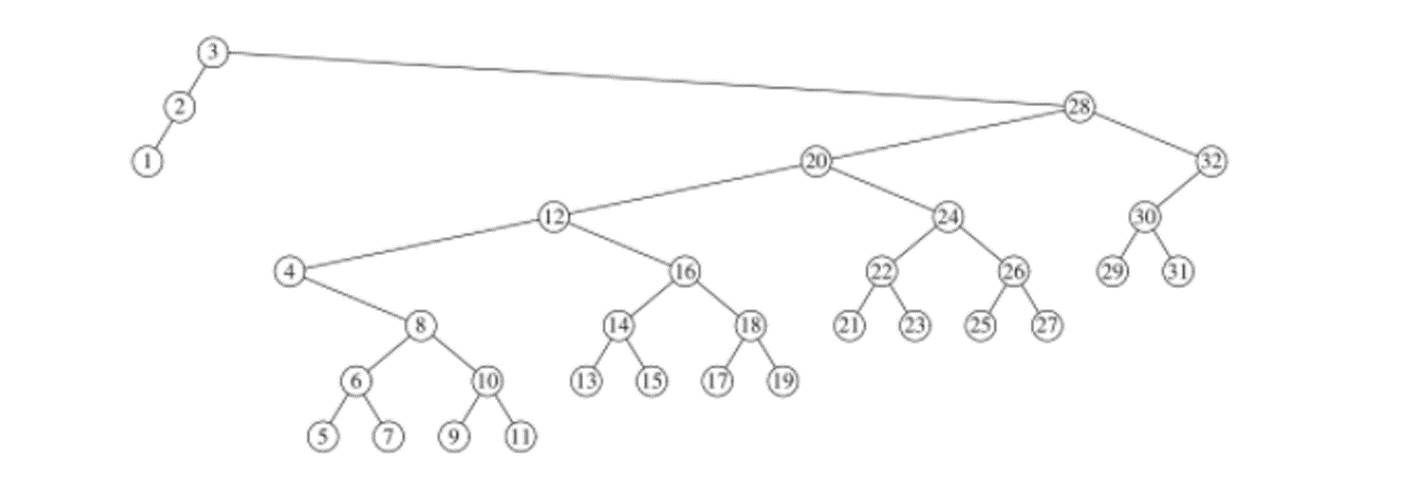
Ví dụ, hãy xem xét cây từ ví dụ cuối cùng, với một chứa trên k1:

Bước splay đầu tiên ở k1 và rõ ràng là zig-zag, vì vậy chúng tôi thực hiện phép quay kép AVL tiêu chuẩn sử dụng k1, k2 và k3. Cây kết quả như sau:

Mặc dù khó có thể nhìn thấy từ các ví dụ nhỏ, việc ghép nối không chỉ di chuyển nút được truy cập về gốc mà còn có tác dụng giảm khoảng một nửa độ sâu của hầu hết các nút trên đường dẫn truy cập (một số nút nông bị đẩy xuống ở nhiều nhất hai cấp).

Để thấy sự khác biệt mà phép xếp chồng tạo ra so với phép quay đơn giản, hãy xem xét lại hiệu quả của việc chèn các mục 1, 2, 3, ..., N vào một cây trống ban đầu. Điều này chiếm tổng O (N), như trước đây, và tạo ra cây giống như các phép quay đơn giản. Hình 4.50 cho thấy kết quả của việc hiển thị tại nút với mục 1. Sự khác biệt là sau khi truy cập vào nút với mục 1, chiếm N đơn vị, truy cập vào nút với mục 2 sẽ chỉ mất khoảng N / 2 đơn vị. của N đơn vị; không còn các nút khá sâu như trước.

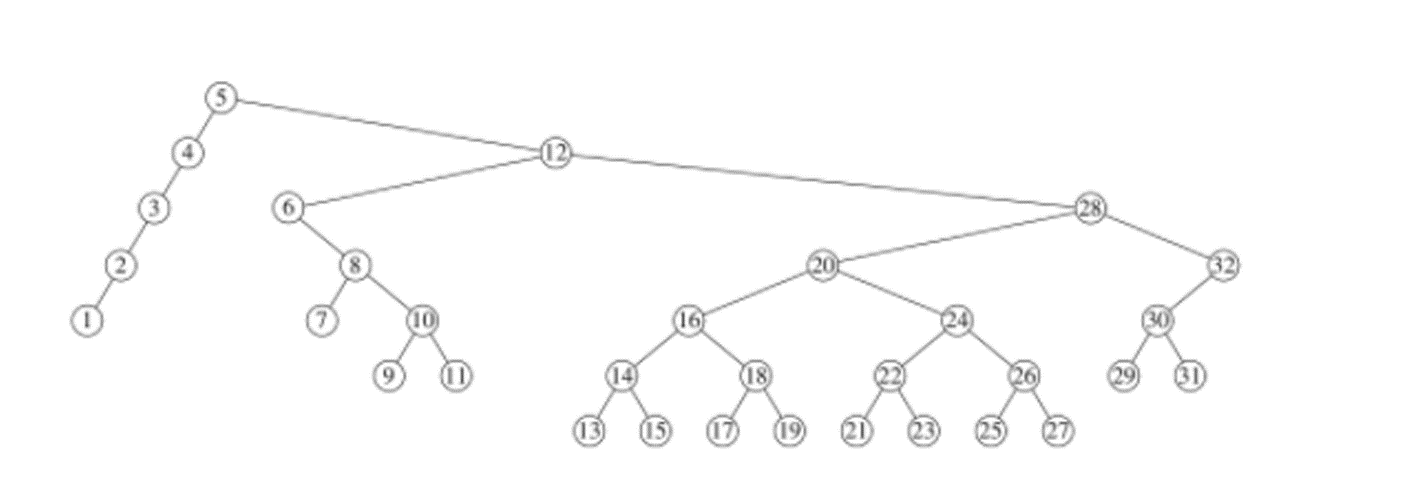
Những số liệu này làm nổi bật tính chất cơ bản và quan trọng của cây gỗ. Khi đường dẫn truy cập dài, do đó dẫn đến thời gian tìm kiếm lâu hơn bình thường, các phép quay có xu hướng tốt cho các hoạt động sau này. Khi quyền truy cập rẻ, các vòng quay sẽ không tốt và có thể xấu. Trường hợp cực đoan là cây ban đầu được hình thành bởi các phép chèn. Tất cả các thao tác chèn đều là các hoạt động thời gian liên tục dẫn đến một cây ban đầu xấu. Vào thời điểm đó, chúng tôi có một cây rất xấu, nhưng chúng tôi đã chạy trước kế hoạch và được đền bù ít hơn tổng thời gian chạy. Sau đó, một vài lần truy cập thực sự khủng khiếp để lại một cái cây gần như cân bằng, nhưng cái giá phải trả là chúng ta phải trả lại một phần thời gian đã tiết kiệm được. Định lý chính mà chúng ta sẽ chứng minh trong Chương 11 là chúng ta không bao giờ bị tụt hậu với tốc độ O (logN) trên mỗi phép toán: Chúng ta luôn đúng tiến độ, mặc dù đôi khi có những hoạt động không tốt.

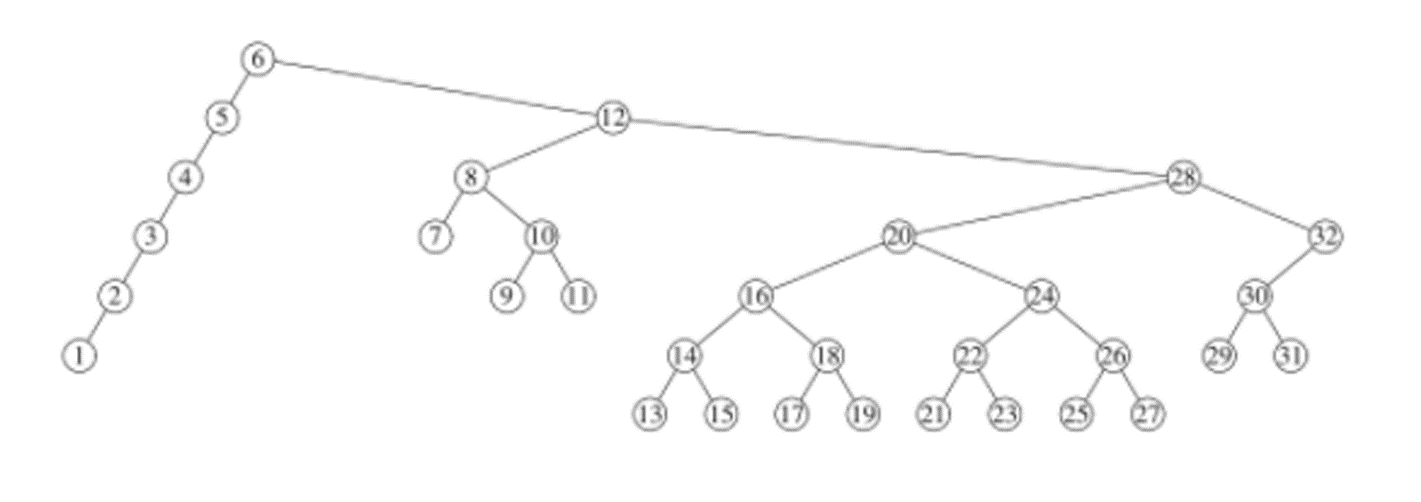
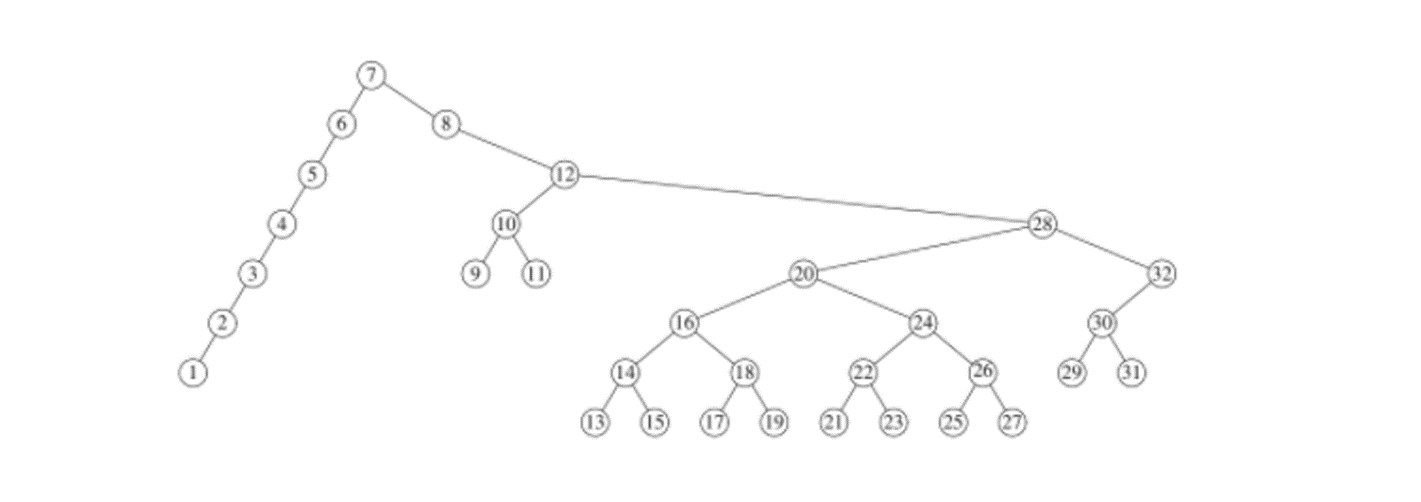
**Hình 4.52** Kết quả hiển thị cây trước đó tại nút 2

**Hình 4.53** Kết quả hiển thị cây trước đó tại nút 3

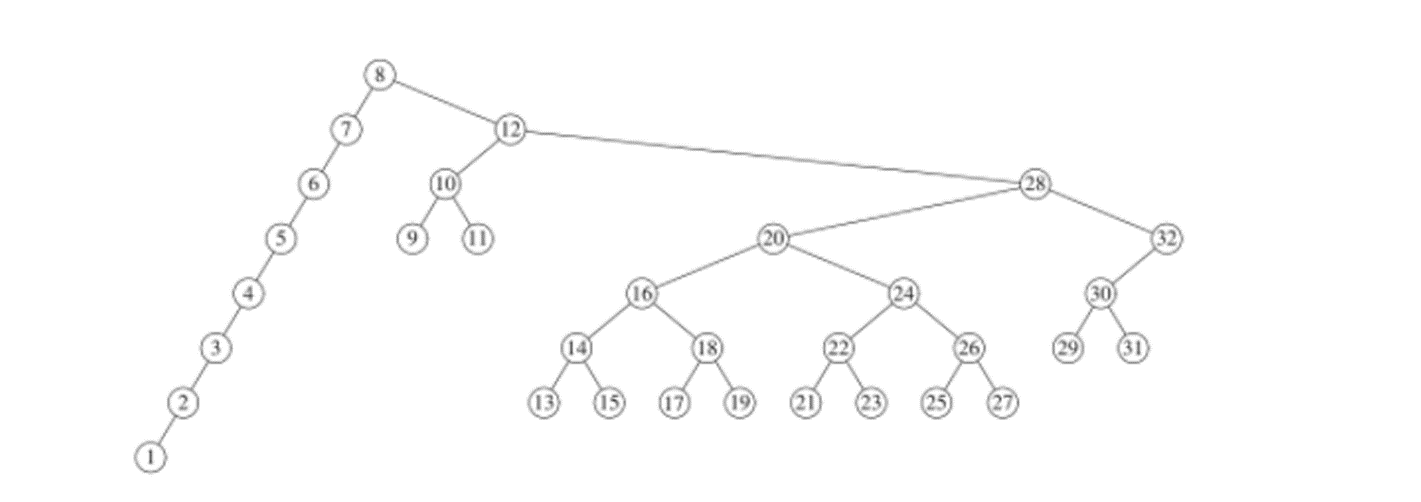
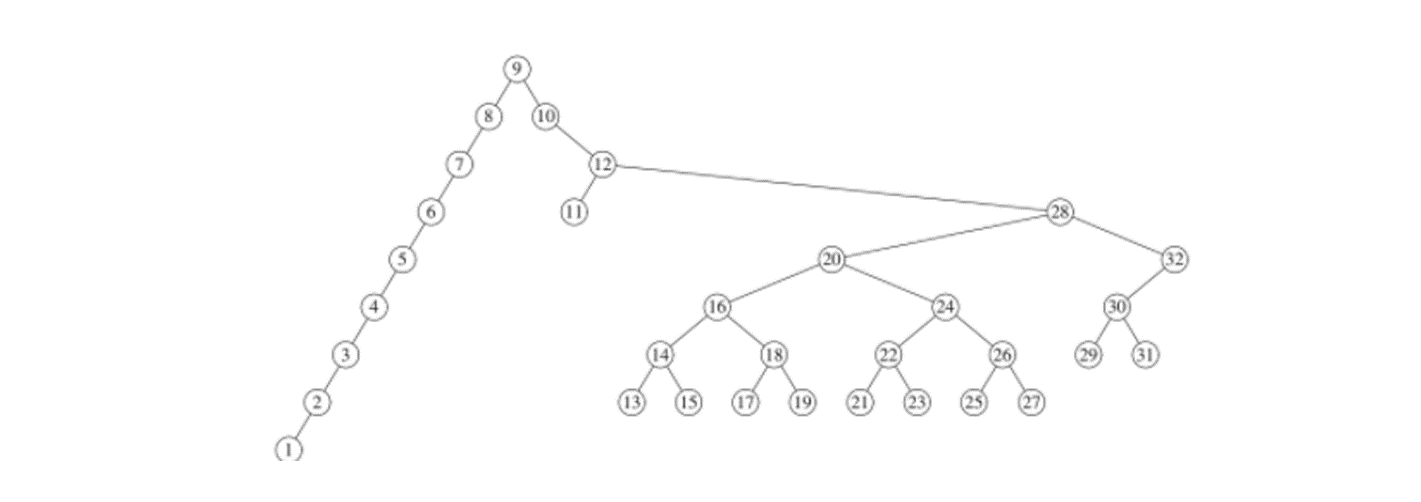
**Hình 4.54** Kết quả hiển thị cây trước đó tại nút 4

Chúng ta có thể thực hiện xóa bằng cách truy cập vào nút cần xóa. Điều này đặt nút ở gốc. Nếu nó bị xóa, chúng ta nhận được hai cây con TL và TR (trái và phải). Nếu chúng ta tìm thấy phần tử lớn nhất trong TL (điều này thật dễ dàng), thì phần tử này sẽ được xoay đến gốc của TL, và lúc này TL sẽ có một gốc không có con bên phải. Chúng ta có thể hoàn thành việc xóa bằng cách biến TR trở thành con phù hợp.

**Hình 4.55** Kết quả hiển thị cây trước đó tại nút 5

 **Hình 4.56** Kết quả hiển thị cây trước đó tại nút 6

**Hình 4.57** Kết quả hiển thị cây trước đó tại nút 7

**Hình 4.58** Kết quả hiển thị cây trước đó tại nút 8

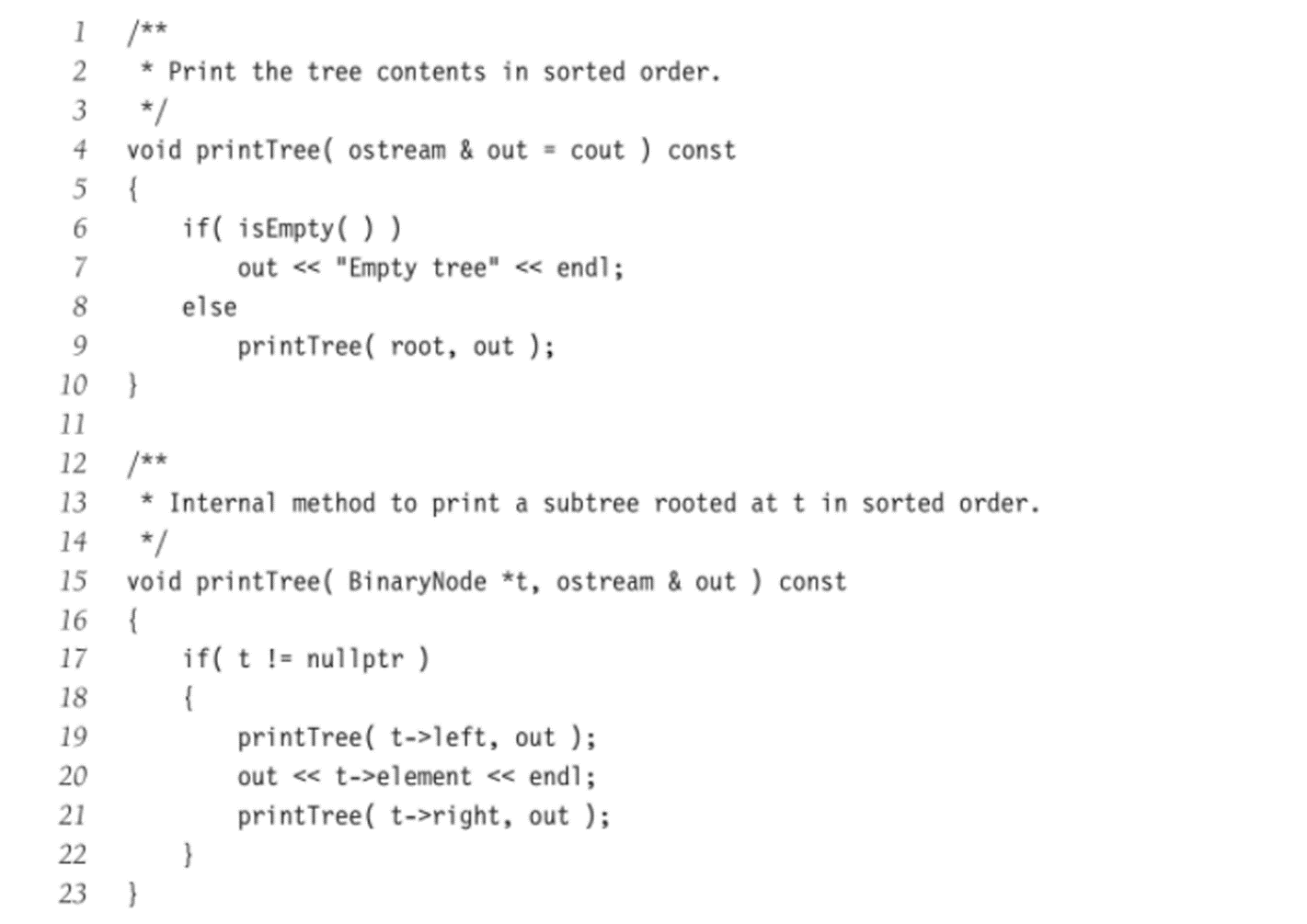
**Hình 4.59** Kết quả hiển thị cây trước đó tại nút 9

Việc phân tích cây cành rất khó, vì nó phải tính đến cấu trúc luôn thay đổi của cây. Mặt khác, cây splay được lập trình đơn giản hơn nhiều so với hầu hết các cây tìm kiếm cân bằng, vì có ít trường hợp cần xem xét hơn và không có thông tin cân bằng để duy trì. Một số bằng chứng thực nghiệm cho thấy rằng điều này chuyển thành mã nhanh hơn trong thực tế, mặc dù trường hợp này vẫn chưa hoàn thiện. Cuối cùng, chúng tôi chỉ ra rằng có một số biến thể của cây ghép có thể hoạt động tốt hơn trong thực tế. Một biến thể được mã hóa hoàn toàn trong Chương 12.

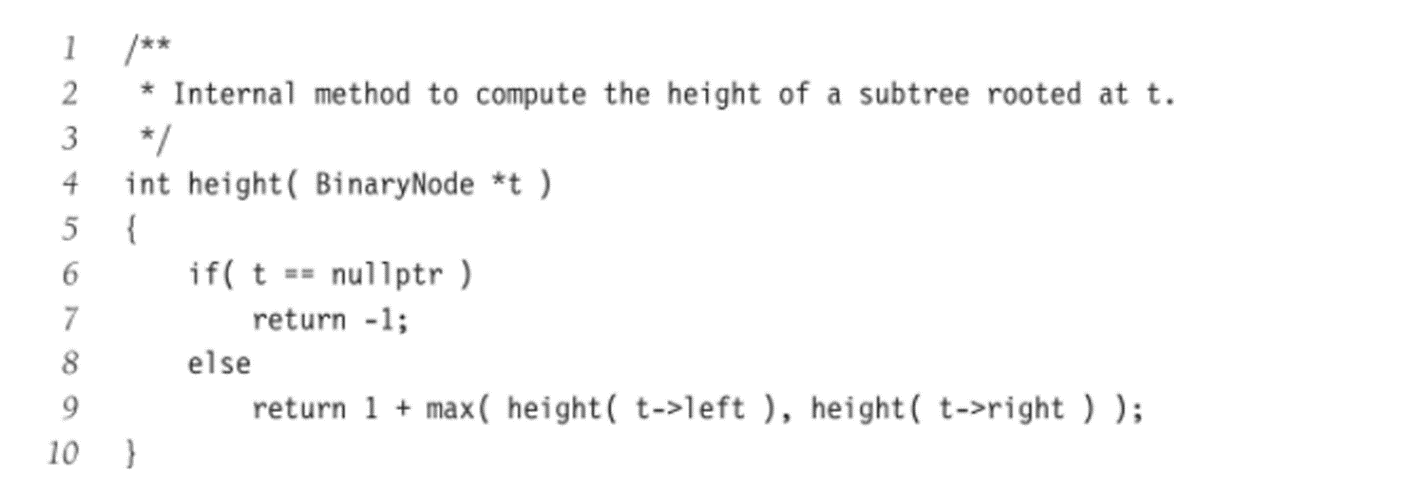
**4.6 Travers cây (Đã xem lại)** - Tree Traversals (Revisited)

Bởi vì thông tin thứ tự trong cây tìm kiếm nhị phân, rất đơn giản để liệt kê tất cả các mục theo thứ tự được sắp xếp. Hàm đệ quy trong Hình 4.60 hoạt động thực sự.

Thuyết phục bản thân rằng chức năng này hoạt động. Như chúng ta đã thấy trước đây, loại quy trình này khi áp dụng cho cây được biết đến như là một đường đi ngang qua không gian (điều này có ý nghĩa, vì nó liệt kê các mục theo thứ tự). Chiến lược chung của một trình duyệt nhỏ hơn là xử lý cây con bên trái trước, sau đó thực hiện xử lý tại nút hiện tại, và cuối cùng xử lý cây con bên phải. Phần thú vị về thuật toán này, ngoài tính đơn giản của nó, là tổng thời gian chạy là O (N). Điều này là do có công việc liên tục được thực hiện tại mọi nút trong cây. Mỗi nút được truy cập một lần và công việc được thực hiện tại mỗi nút đang kiểm tra nullptr, thiết lập hai lệnh gọi hàm và thực hiện một câu lệnh đầu ra. Vì có công việc liên tục trên mỗi nút và N nút, thời gian chạy là O (N).

 O (N), bởi vì công việc liên tục được thực hiện tại mỗi nút.

**Hình 4.60** Quy trình in cây tìm kiếm nhị phân theo thứ tự

**Hình 4.61** Quy trình tính toán chiều cao của cây bằng cách sử dụng trình tự chuyển hướng sau

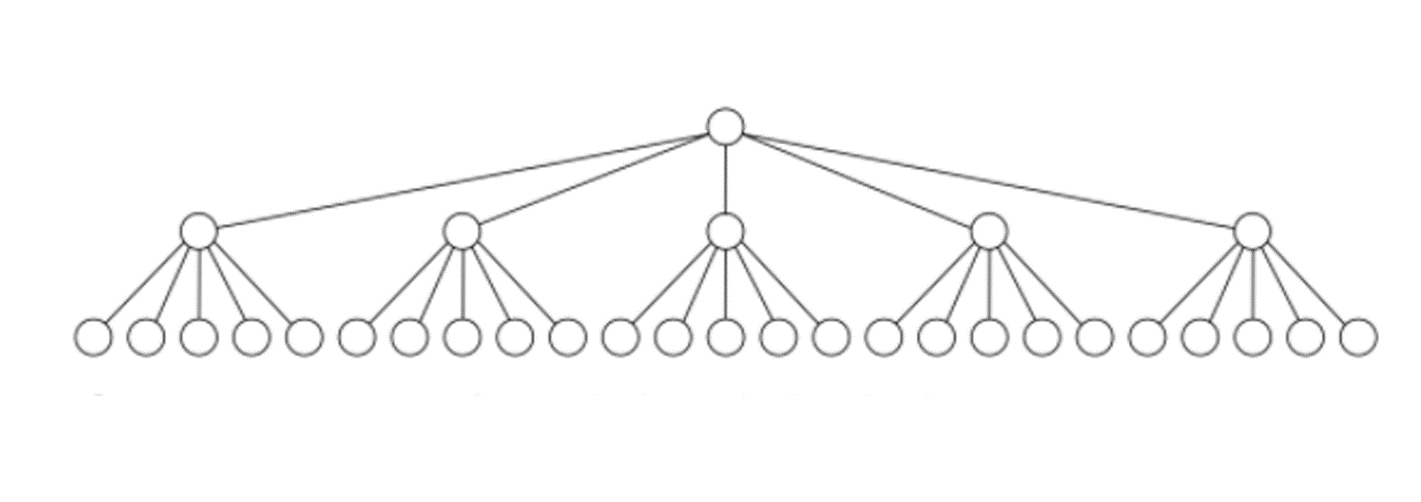
Sơ đồ truyền tải phổ biến thứ ba mà chúng tôi đã thấy là đặt hàng trước truyền tải. Ở đây, nút được xử lý trước phần con. Điều này có thể hữu ích, chẳng hạn, nếu bạn muốn gắn nhãn mỗi nút với độ sâu của nó.

Ý tưởng chung trong tất cả các quy trình này là bạn xử lý trường hợp nullptr trước rồi mới đến phần còn lại. Chú ý thiếu các biến ngoại lai. Các quy trình này chỉ truyền con trỏ đến nút gốc của cây con và không khai báo hoặc chuyển bất kỳ biến phụ nào. Mã càng nhỏ gọn thì khả năng xuất hiện lỗi ngớ ngẩn càng ít. Phương thức thứ tư, ít được sử dụng hơn, (mà chúng ta chưa thấy) là phương thức truyền bậc. Trong một trình duyệt theo thứ tự mức, tất cả các nút ở độ sâu d được xử lý trước bất kỳ nút nào ở độ sâu d + 1. Truyền ngang theo thứ tự mức khác với các lần truyền khác ở chỗ nó không được thực hiện một cách đệ quy; một hàng đợi được sử dụng, thay vì ngăn xếp đệ quy ngụ ý.

**4.7 B-Cây** (B-Trees)

Cho đến nay, chúng tôi đã giả định rằng chúng tôi có thể lưu trữ toàn bộ cấu trúc dữ liệu trong bộ nhớ chính của máy tính. Tuy nhiên, giả sử rằng chúng ta có nhiều dữ liệu hơn mức có thể chứa trong bộ nhớ chính và do đó, cấu trúc dữ liệu phải nằm trên đĩa. Khi điều này xảy ra, luật chơi sẽ thay đổi, vì mô hình Big-Oh không còn ý nghĩa nữa.

Vấn đề là một phân tích Big-Oh giả định rằng tất cả các hoạt động đều bình đẳng. Tuy nhiên, điều này không đúng, đặc biệt khi có liên quan đến I / O đĩa. Máy tính hiện đại thực hiện hàng tỷ lệnh mỗi giây. Điều đó là khá nhanh, chủ yếu là vì tốc độ phụ thuộc phần lớn vào các đặc tính điện. Mặt khác, đĩa là cơ học. Tốc độ của nó phụ thuộc phần lớn vào thời gian quay đĩa và di chuyển đầu đĩa. Nhiều đĩa quay với tốc độ 7.200 vòng / phút. Như vậy, trong 1 phút, nó tạo ra 7.200 vòng quay; do đó, một vòng quay xảy ra trong 1/120 giây, hoặc 8,3 ms.

 Cây tìm kiếm nhị phân không cân bằng là một thảm họa. Trong trường hợp xấu nhất, nó có độ sâu tuyến tính và do đó có thể yêu cầu 10.000.000 lượt truy cập đĩa. Trung bình, một tìm kiếm thành công sẽ yêu cầu truy cập đĩa 1,38 logN và kể từ log 10000000 ≈ 24, một tìm kiếm trung bình sẽ yêu cầu 32 lần truy cập đĩa, hoặc 5 giây. Trong một cây được xây dựng ngẫu nhiên điển hình, chúng tôi mong đợi rằng một vài nút sâu hơn ba lần; chúng sẽ yêu cầu khoảng 100 quyền truy cập đĩa, hoặc 16 giây. Một cây AVL có phần tốt hơn. Trường hợp xấu nhất là 1.44 logN khó có thể xảy ra, và trường hợp điển hình là rất gần với logN. Vì vậy, một cây AVL sẽ sử dụng trung bình khoảng 25 lần truy cập đĩa, yêu cầu 4 giây.

Hình 4.62 Cây 5 ary gồm 31 nút chỉ có ba mức

Chúng tôi muốn giảm số lượng truy cập đĩa xuống một hằng số rất nhỏ, chẳng hạn như ba hoặc bốn. Chúng tôi sẵn sàng viết những đoạn mã phức tạp để làm điều này, bởi vì các lệnh máy về cơ bản là miễn phí, miễn là chúng tôi không vô lý một cách lố bịch. Có lẽ phải rõ ràng rằng cây tìm kiếm nhị phân sẽ không hoạt động, vì cây AVL điển hình gần với chiều cao tối ưu. Chúng tôi không thể đi xuống dưới logN bằng cách sử dụng cây tìm kiếm nhị phân. Giải pháp rất đơn giản trực quan: Nếu chúng ta có nhiều nhánh hơn, chúng ta có ít chiều cao hơn. Do đó, trong khi một cây nhị phân hoàn hảo gồm 31 nút có năm mức, thì cây 5-ary gồm 31 nút chỉ có ba mức, như trong Hình 4.62. Cây tìm kiếm M-ary cho phép phân nhánh M-way. Khi phân nhánh tăng, độ sâu giảm. Trong khi một cây nhị phân hoàn chỉnh có chiều cao xấp xỉ log2 N, một cây M-ary hoàn chỉnh có chiều cao xấp xỉ logM N.

Chúng ta có thể tạo cây tìm kiếm M-ary theo cách giống như cây tìm kiếm nhị phân. Trong cây tìm kiếm nhị phân, chúng ta cần một khóa để quyết định chọn nhánh nào trong hai nhánh. Trong cây tìm kiếm M-ary, chúng ta cần M-1 phím để quyết định chọn nhánh nào. Để làm cho lược đồ này hiệu quả trong trường hợp xấu nhất, chúng ta cần đảm bảo rằng cây tìm kiếm M-ary được cân bằng theo một cách nào đóCó nghĩa là, chúng tôi không muốn cây tìm kiếm M-ary suy biến thành cây tìm kiếm nhị phân, bởi vì khi đó chúng tôi sẽ bị mắc kẹt với các truy cập logN

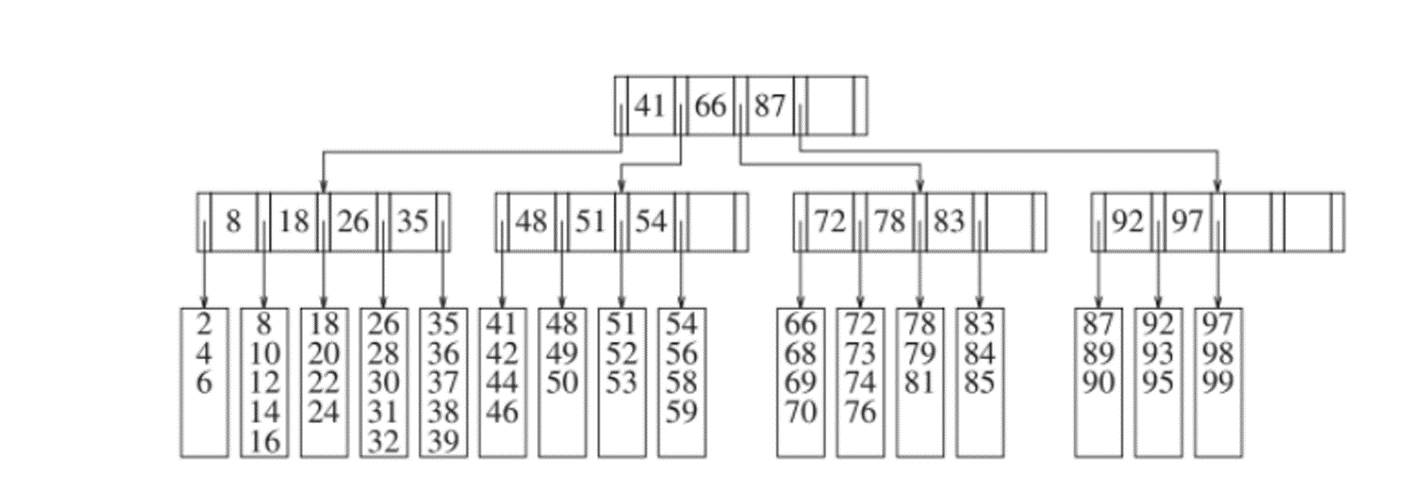
Cây B bậc M là cây M có các chất sau:

**1**. Các mục dữ liệu được lưu trữ tại các lá.

**2**. Các nút không phải ở trang chính lưu trữ tối đa M - 1 khóa để hướng dẫn tìm kiếm; khóa i đại diện cho khóa nhỏ nhất trong cây con i + 1.

**3**. Gốc là một chiếc lá hoặc có từ hai đến M con.

**4**. Tất cả các nút không phải lá (trừ nút gốc) có từ M / 2 đến M con.

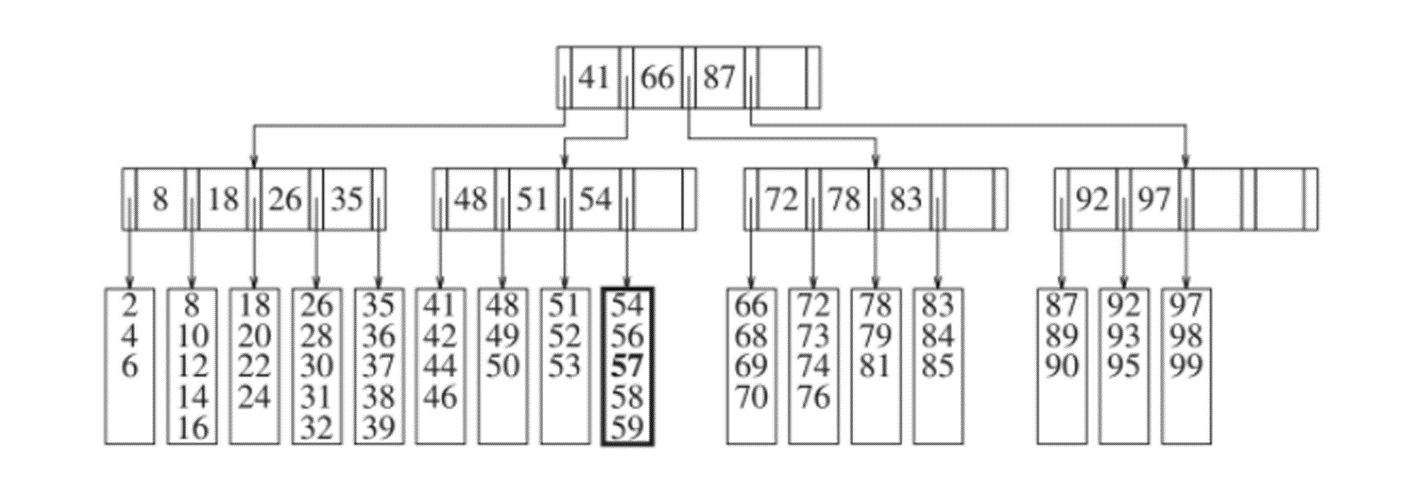
****. Tất cả các lá ở cùng độ sâu và có giữa các mục dữ liệu L / 2 và L, đối với một số L (việc xác định L được mô tả ngay sau đây).

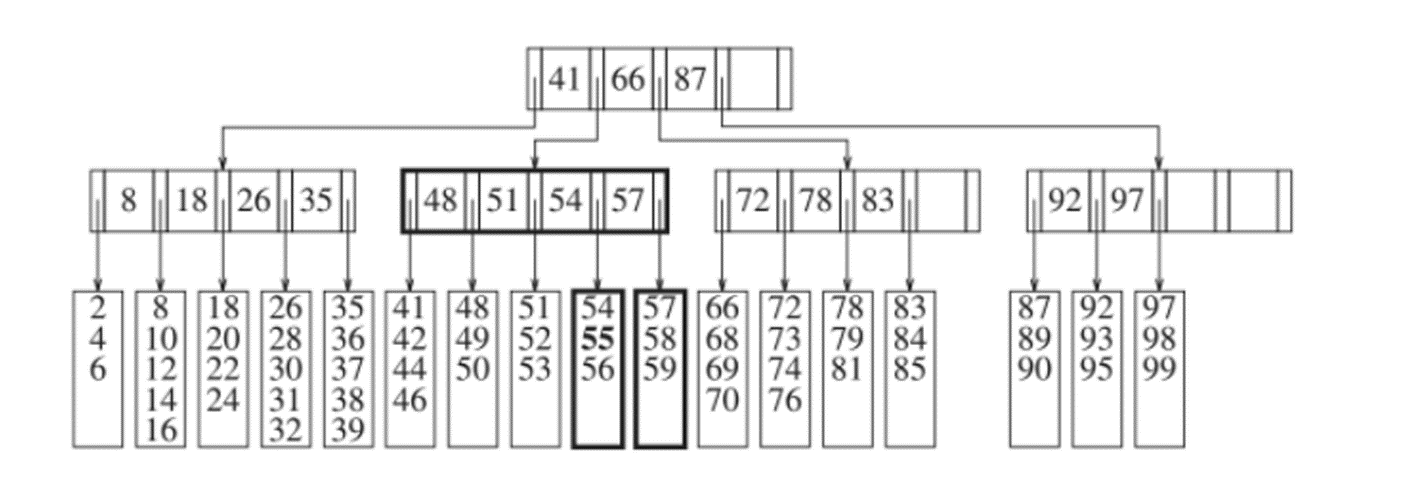
**Hình 4.63** Cây B của đơn hàng 5

Ví dụ về cây B bậc 5 được thể hiện trong Hình 4.63. Lưu ý rằng tất cả các nút không phải ở lá đều có từ ba đến năm nút con (và do đó từ hai đến bốn khóa); gốc có thể chỉ có hai con. Ở đây, chúng ta có L = 5. Điều xảy ra rằng L và M là các điểm phù hợp trong ví dụ này, nhưng điều này là không cần thiết. Vì L là 5, mỗi lá có từ ba đến năm mục dữ liệu. Việc yêu cầu các nút phải đầy một nửa đảm bảo rằng cây B không bị suy biến thành cây nhị phân đơn giản. Mặc dù có nhiều định nghĩa khác nhau về cây B thay đổi cấu trúc này, chủ yếu là theo những cách nhỏ, nhưng định nghĩa này là một trong những dạng phổ biến.

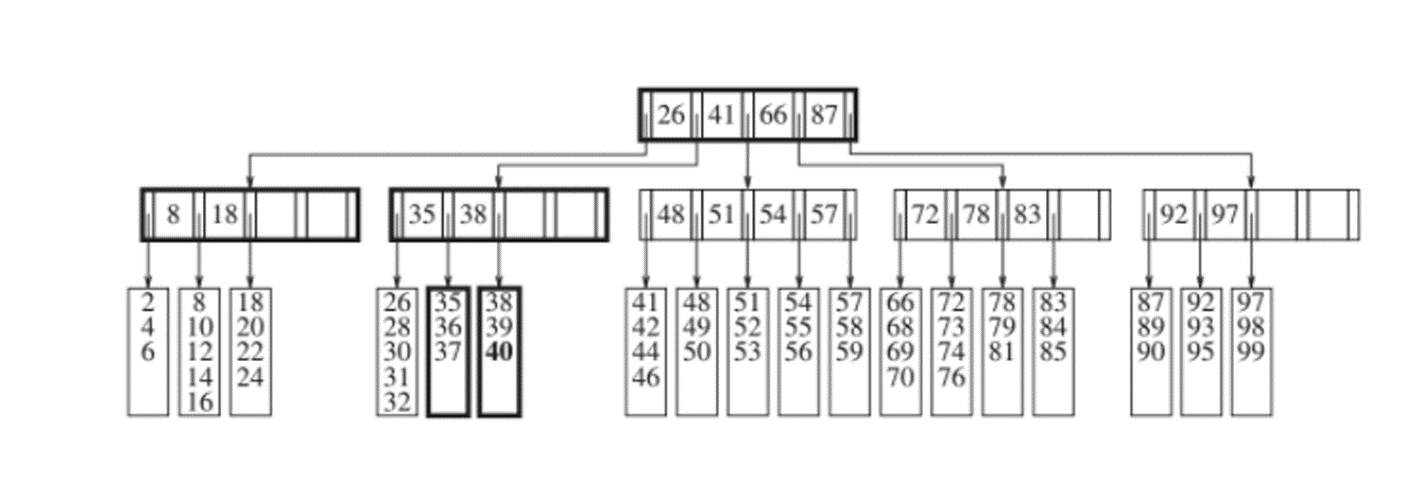
Mỗi nút đại diện cho một khối đĩa, vì vậy chúng tôi chọn M và L trên cơ sở kích thước của các mục đang được lưu trữ. Ví dụ, giả sử một khối chứa 8.192 byte. Trong ví dụ Florida của chúng tôi, mỗi khóa sử dụng 32 byte. Trong một cây B có thứ tự M, chúng ta sẽ có M-1 khóa, với tổng số 32M - 32 byte, cộng với M nhánh. Vì mỗi nhánh về cơ bản là một số của khối đĩa khác, chúng ta có thể giả định rằng một nhánh là 4 byte. Do đó, các nhánh sử dụng 4M byte. Do đó, tổng yêu cầu bộ nhớ cho một nút không phải là 36M-32. Giá trị lớn nhất của M mà giá trị này không lớn hơn 8,192 là 228. Vì vậy, chúng tôi sẽ chọn M = 228. Vì mỗi bản ghi dữ liệu là 256 byte, chúng tôi sẽ có thể phù hợp với 32 bản ghi trong một khối. Vì vậy, chúng tôi sẽ chọn L = 32. Chúng tôi đảm bảo rằng mỗi lá có từ 16 đến 32 bản ghi dữ liệu và mỗi nút bên trong (trừ nút gốc) phân nhánh theo ít nhất 114 cách. Vì có 10.000.000 bản ghi, nên nhiều nhất là 625.000 lá.

Tất nhiên, điều đó tương đối không đau, bởi vì chiếc lá chưa đầy. Giả sử bây giờ chúng ta muốn chèn 55. Hình 4.64 cho thấy một vấn đề: Lá 55 muốn đi đã đầy. Giải pháp rất đơn giản: Vì hiện tại chúng tôi có L +1 mục, chúng tôi chia chúng thành hai lá, cả hai đều được đảm bảo có số lượng bản ghi dữ liệu tối thiểu cần thiết. Chúng tôi tạo thành hai lá với ba mục mỗi loại. Cần có hai quyền truy cập đĩa để ghi các lá này, và cần có quyền truy cập đĩa thứ ba để cập nhật phụ huynh. Lưu ý rằng trong cha, cả khóa và nhánh đều thay đổi, nhưng chúng làm như vậy theo cách có kiểm soát và dễ dàng tính toán được. Cây B kết quả được thể hiện trong Hình 4.65. Mặc dù việc chia tách các nút tốn nhiều thời gian vì nó

**Hình 4.64** Cây B sau khi chèn 57 vào cây trong Hình 4.63

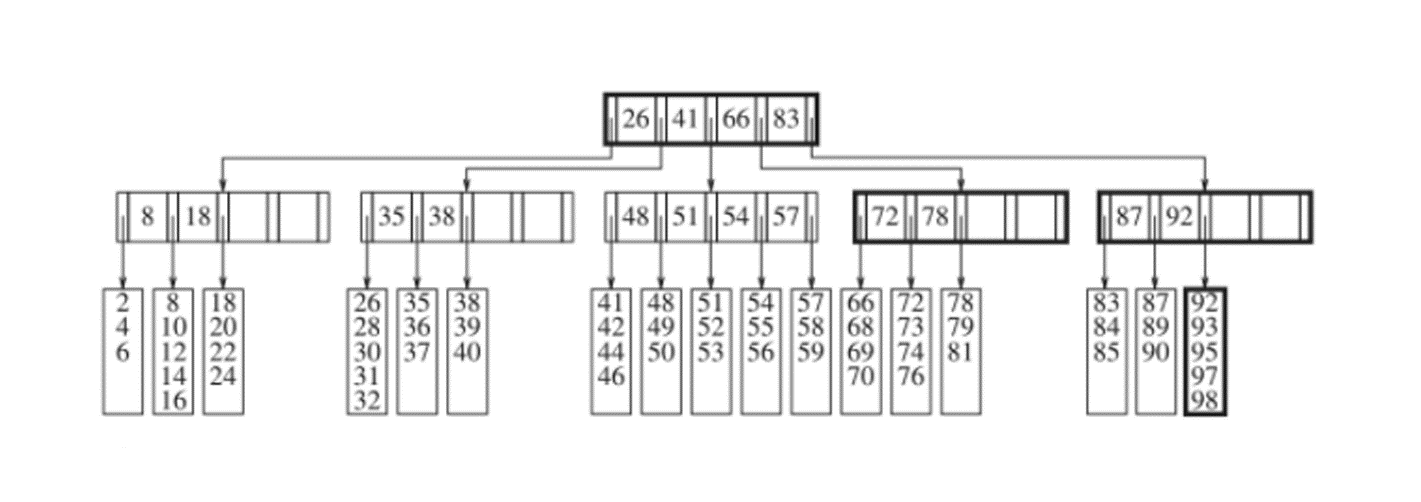
Việc tách nút trong ví dụ trước đã hoạt động vì nút cha không có phần bổ sung đầy đủ của nút con. Nhưng điều gì sẽ xảy ra nếu nó xảy ra? Ví dụ, giả sử rằng chúng ta chèn 40 vào cây B trong Hình 4.65. Chúng ta phải tách lá chứa các phím từ 35 đến 39, và bây giờ là 40, thành hai lá. Nhưng làm điều này sẽ cung cấp cho cha mẹ sáu đứa con, và nó chỉ được phép có năm. Giải pháp là tách cha mẹ. Kết quả của việc này được thể hiện trong Hình 4.66. Khi khóa chính được tách ra, chúng ta phải cập nhật giá trị của các khóa và cũng là khóa chính của khóa gốc, do đó phát sinh thêm hai lần ghi đĩa (do đó, việc chèn này tốn năm lần ghi đĩa). Tuy nhiên, một lần nữa, các khóa thay đổi theo cách rất được kiểm soát, mặc dù mã chắc chắn không đơn giản vì nhiều trường hợp.

**Hình 4.65** Chèn 55 vào cây B trong hình 4.64 gây ra hiện tượng tách thành hai lá

**Hình 4.66** Chèn 40 vào cây B trong Hình 4.65 gây ra sự phân tách thành hai lá và sau đó là sự phân tách của nút cha

Khi một nút không phải được tách ra, như trường hợp ở đây, nút cha của nó nhận được một nút con. Điều gì sẽ xảy ra nếu cha mẹ đã đạt đến giới hạn của con cái? Sau đó, chúng tôi tiếp tục tách các nút trên cây cho đến khi chúng tôi tìm thấy một nút cha không cần phải tách hoặc chúng tôi đạt được gốc. Nếu chúng ta tách gốc, thì chúng ta có hai gốc. Rõ ràng, điều này là không thể chấp nhận được, nhưng chúng ta có thể tạo ra một gốc mới có gốc tách như hai con của nó. Đây là lý do tại sao gốc được cấp quyền miễn trừ đặc biệt cho hai đứa trẻ. Đây cũng là cách duy nhất để cây B tăng chiều cao. Không cần phải nói, tách toàn bộ đến tận gốc là một sự kiện đặc biệt hiếm. Điều này là do một cây có bốn cấp chỉ ra rằng gốc đã được tách ba lần trong toàn bộ chuỗi chèn (giả sử không có sự xóa nào xảy ra). Trên thực tế, việc tách bất kỳ nút không phải là nút nào cũng khá hiếm.

Chúng tôi có thể thực hiện xóa bằng cách tìm mục cần xóa và sau đó xóa nó. Vấn đề là nếu lá của nó có số lượng mục dữ liệu tối thiểu, thì bây giờ nó dưới mức tối thiểu. Chúng ta có thể khắc phục tình trạng này bằng cách áp dụng một mục lân cận, nếu hàng xóm đó không phải là chính nó ở mức tối thiểu. Nếu có thì ta có thể kết hợp với hàng xóm để tạo thành một lá đầy đủ. Thật không may, điều này có nghĩa là cha mẹ đã mất một đứa trẻ. Nếu điều này gây ra

cha mẹ giảm xuống dưới mức tối thiểu của nó, sau đó nó tuân theo cùng một chiến lược. Quá trình này có thể thấm vào tận gốc rễ. Gốc không thể có chỉ một con (và ngay cả khi điều này được cho phép, nó sẽ là ngớ ngẩn). Nếu một gốc còn lại một con là kết quả của quá trình nhận nuôi, thì chúng ta loại bỏ gốc và đặt con của nó làm gốc mới của cây. Đây là cách duy nhất để cây B giảm chiều cao. Ví dụ, giả sử chúng ta muốn loại bỏ 99 khỏi cây B trong Hình 4.66. Vì lá chỉ có hai mục và người hàng xóm của nó đã có tối thiểu ba mục, chúng tôi kết hợp các mục thành một lá mới gồm năm mục. Kết quả là cha mẹ chỉ có hai con. Tuy nhiên, nó có thể nhận con nuôi từ một người hàng xóm, vì người hàng xóm có bốn đứa con. Kết quả là cả hai có ba người con. Kết quả như hình 4.67.

**Hình 4.67** Cây B sau khi xóa 99 khỏi cây B trong Hình 4.66

**4.8 Bộ và Bản đồ trong Thư viện Chuẩn** (Sets and Maps in the Standard Library)

Các vùng chứa STL được thảo luận trong Chương 3 — cụ thể là vectơ và danh sách — không hiệu quả cho việc tìm kiếm. Do đó, STL cung cấp hai vùng chứa bổ sung, tập hợp và ánh xạ, đảm bảo chi phí logarit cho các hoạt động cơ bản như chèn, xóa và tìm kiếm.

**4.8.1 Bộ** (Sets)

Tập hợp là một vùng chứa có thứ tự không cho phép trùng lặp. Nhiều thành ngữ được sử dụng để truy cập các mục trong vectơ và danh sách cũng hoạt động cho một tập hợp. Cụ thể, lồng trong tập hợp là các loại trình lặp và const\_iterator cho phép truyền qua tập hợp và một số phương thức từ vectơ và danh sách được đặt tên giống hệt nhau trong tập hợp, bao gồm bắt đầu, kết thúc, kích thước và trống. Mẫu hàm in được mô tả trong Hình 3.6 sẽ hoạt động nếu được thông qua một bộ.

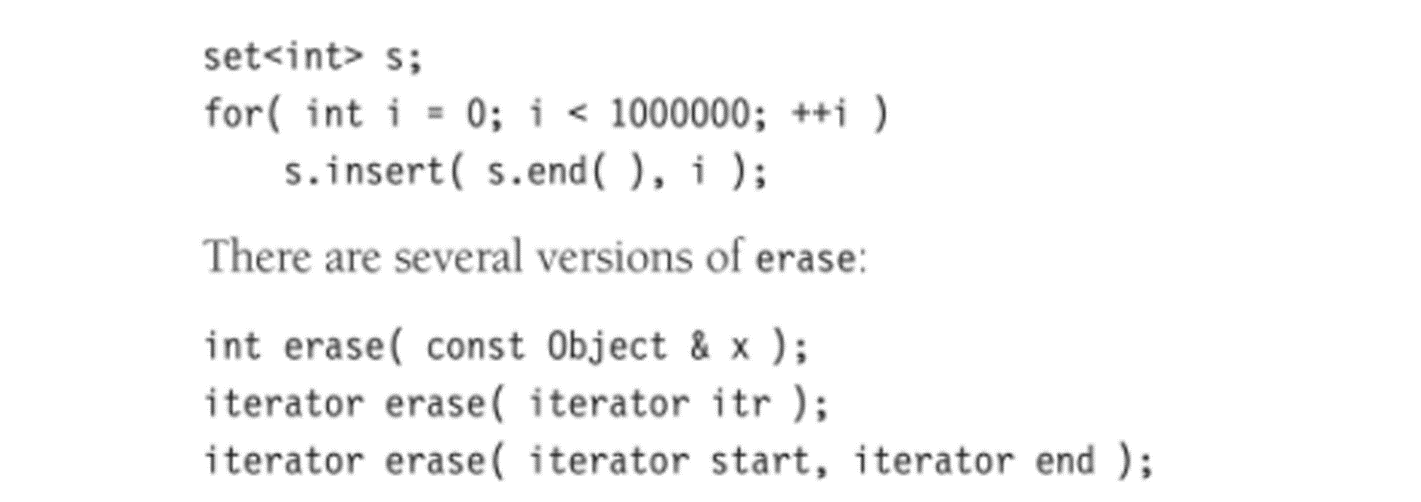
Các hoạt động duy nhất được yêu cầu bởi tập hợp là khả năng chèn, loại bỏ và thực hiện tìm kiếm cơ bản (hiệu quả).

Quy trình chèn được đặt tên phù hợp. Tuy nhiên, vì một tập hợp không cho phép trùng lặp, nên việc chèn có thể bị lỗi. Do đó, chúng tôi muốn kiểu trả về có thể chỉ ra điều này bằng một biến Boolean. Tuy nhiên, insert có kiểu trả về phức tạp hơn bool. Điều này là do insert cũng trả về một trình lặp đại diện cho vị trí x khi insert trả về. Trình vòng lặp này đại diện cho mục mới được chèn hoặc mục hiện có khiến việc chèn không thành công và nó rất hữu ích, vì biết vị trí của mục có thể làm cho việc xóa nó hiệu quả hơn bằng cách tránh tìm kiếm và truy cập trực tiếp vào nút chứa mục.

STL định nghĩa một mẫu lớp được gọi là cặp ít hơn một cấu trúc với các thành viên đầu tiên và thứ hai để truy cập hai mục trong cặp. Có hai quy trình chèn khác nhau:

pair<iterator,bool> insert( const Object & x );

pair<iterator,bool> insert( iterator hint, const Object & x );

Chèn một tham số hoạt động như mô tả ở trên. Chèn hai tham số cho phép đặc tả một gợi ý, đại diện cho vị trí mà x sẽ đến. Nếu gợi ý chính xác, việc chèn nhanh, thường là O (1). Nếu không, việc chèn được thực hiện bằng cách sử dụng thuật toán chèn thông thường và thực hiện tương tự với phép chèn một tham số. Ví dụ: mã sau có thể nhanh hơn bằng cách sử dụng chèn hai tham số thay vì chèn một tham số:

Thao tác xóa một tham số đầu tiên sẽ xóa x (nếu được tìm thấy) và trả về số lượng mục thực sự bị xóa, rõ ràng là 0 hoặc 1. Thao tác xóa một tham số thứ hai hoạt động giống như trong vector và danh sách. Nó loại bỏ đối tượng tại vị trí được cung cấp bởi trình vòng lặp, trả về một trình vòng lặp đại diện cho phần tử theo sau nó ngay trước lệnh gọi xóa và làm mất hiệu lực itr, trở nên cũ. Thao tác xóa hai tham số hoạt động giống như trong vectơ hoặc danh sách, xóa tất cả các mục bắt đầu từ đầu, cho đến nhưng không bao gồm mục ở cuối.

Đối với tìm kiếm, thay vì một quy trình chứa trả về một biến Boolean, tập hợp cung cấp một quy trình tìm kiếm trả về một trình lặp đại diện cho vị trí của mục (hoặc dấu đánh dấu nếu tìm kiếm không thành công). Điều này cung cấp nhiều thông tin hơn, không tốn thời gian chạy. Chữ ký của tìm là

iterator find( const Object & x ) const;

Theo mặc định, việc sắp xếp sử dụng đối tượng hàm less <Object>, đối tượng này được thực hiện bằng cách gọi toán tử <cho đối tượng. Một thứ tự thay thế có thể được chỉ định bằng cách cài đặt mẫu tập hợp với một loại đối tượng hàm. Ví dụ, chúng ta có thể tạo một tập hợp lưu trữ các đối tượng chuỗi, bỏ qua sự phân biệt chữ hoa chữ thường bằng cách sử dụng đối tượng hàm CaseInsensitiveCompare được mã hóa trong Hình 1.25. Trong đoạn mã sau, tập hợp s có kích thước 1:

set<string,CaseInsensitiveCompare> s;

s.insert( "Hello" ); s.insert( "HeLLo" );

cout << "The size is: " << s.size( ) << endl;

**4.8.2 Bản đồ** (Maps)

Bản đồ được sử dụng để lưu trữ một tập hợp các mục nhập có thứ tự bao gồm các khóa và giá trị của chúng. Các khóa phải là duy nhất, nhưng một số khóa có thể ánh xạ đến các giá trị giống nhau. Vì vậy, các giá trị không cần phải là duy nhất. Các phím trong bản đồ được duy trì theo thứ tự được sắp xếp hợp lý.

Bản đồ hoạt động giống như một tập hợp được khởi tạo với một cặp, có chức năng so sánh chỉ tham chiếu đến khóa.5 Do đó, nó hỗ trợ bắt đầu, kết thúc, kích thước và trống, nhưng trình vòng lặp nằm dưới là một cặp khóa-giá trị. Nói cách khác, đối với một trình lặp itr, \* itr thuộc loại cặp <KeyType, ValueType>. Bản đồ cũng hỗ trợ chèn, tìm và xóa. Để chèn, người ta phải cung cấp một cặp đối tượng <KeyType, ValueType>. Mặc dù tìm kiếm chỉ yêu cầu một khóa, trình lặp nó trả về một cặp tham chiếu. Chỉ sử dụng các thao tác này thường không đáng giá vì hành lý cú pháp có thể tốn kém.

May mắn thay, bản đồ có một thao tác bổ sung quan trọng mang lại cú pháp đơn giản. Toán tử lập chỉ mục mảng được nạp chồng cho các bản đồ như sau:

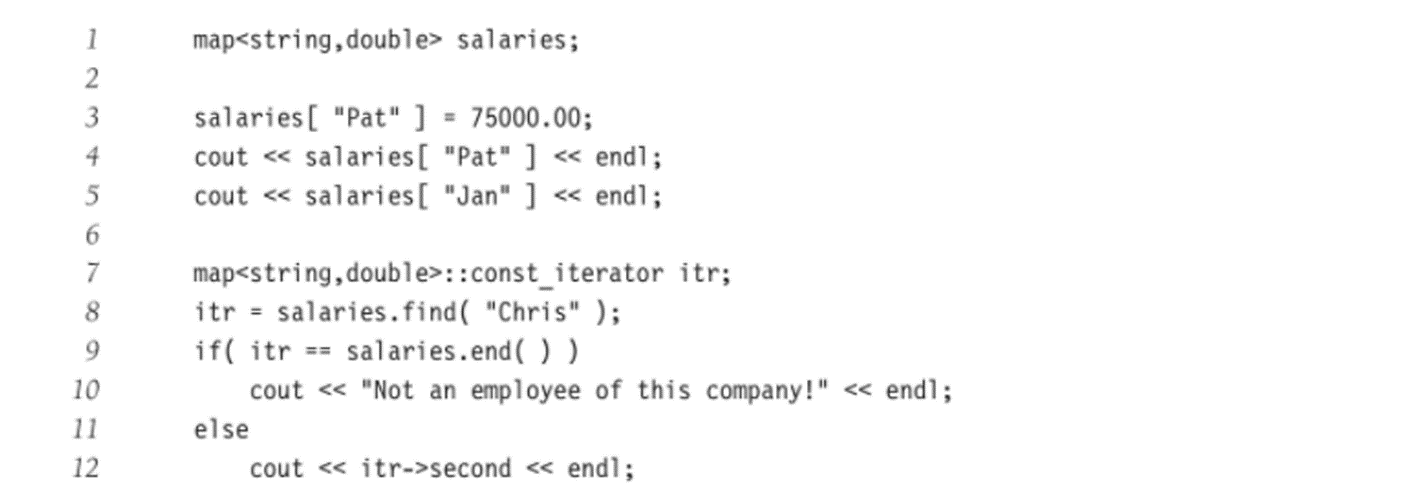
ValueType & operator [] (const KeyType & key);

Ngữ nghĩa của toán tử [] như sau. Nếu khóa có trong bản đồ, một tham chiếu đến giá trị tương ứng sẽ được trả về. Nếu khóa không có trong bản đồ, thì nó sẽ được chèn một giá trị mặc định vào bản đồ và sau đó một tham chiếu đến giá trị mặc định đã chèn sẽ được trả về. Giá trị mặc định nhận được bằng cách áp dụng một hàm tạo tham số bằng không hoặc bằng 0 cho các kiểu nguyên thủy. Những ngữ nghĩa này không cho phép một phiên bản truy cập của toán tử [], vì vậy toán tử [] không thể được sử dụng trên một bản đồ là hằng số. Ví dụ, nếu một bản đồ được truyền bằng tham chiếu không đổi, bên trong quy trình, toán tử [] không thể sử dụng được.

Đoạn mã trong Hình 4.68 minh họa hai kỹ thuật để truy cập các mục trong bản đồ. Đầu tiên, hãy quan sát rằng tại dòng 3, phía bên trái gọi toán tử [], do đó chèn "Pat" và một giá trị kép 0 vào bản đồ, và trả về một tham chiếu đến kép đó. Sau đó, nhiệm vụ thay đổi gấp đôi bên trong bản đồ thành 75000. Dòng 4 xuất ra 75000. Thật không may, dòng 5 chèn "Jan" và mức lương 0,0 vào bản đồ và sau đó in nó. Điều này có thể có hoặc có thể không phải là điều thích hợp để làm, tùy thuộc vào ứng dụng. Nếu điều quan trọng là phải phân biệt giữa các mục có trong bản đồ và những mục không có trong bản đồ hoặc nếu điều quan trọng là không chèn vào bản đồ (vì nó là bất biến), thì một cách tiếp cận thay thế được hiển thị ở dòng 7 đến dòng 12 có thể là đã sử dụng. Ở đó chúng tôi thấy một cuộc gọi để tìm. Nếu khóa không được tìm thấy, trình vòng lặp là dấu đánh dấu và có thể được kiểm tra. Nếu khóa được tìm thấy, chúng ta có thể truy cập mục thứ hai trong cặp được tham chiếu bởi trình vòng lặp, là giá trị được liên kết với khóa. Chúng tôi cũng có thể gán cho itr-> second nếu, thay vì một const\_iterator, itr là một trình lặp.

**4.8.3 Thực hiện bộ và bản đồ** (Implementation of set and map)

C ++ yêu cầu bộ và bản đồ đó hỗ trợ các thao tác chèn, xóa và tìm cơ bản trong trường hợp xấu nhất theo lôgarit. Do đó, việc triển khai cơ bản là một sự cân bằng

**Hình 4.68** Truy cập các giá trị trong bản đồ

cây tìm kiếm nhị phân. Thông thường, một cây AVL không được sử dụng; thay vào đó, các cây đỏ-đen từ trên xuống, được thảo luận trong Phần 12.2, thường được sử dụng.

Một vấn đề quan trọng trong việc triển khai set và map đang cung cấp hỗ trợ cho các lớp trình vòng lặp. Tất nhiên, bên trong, trình lặp duy trì một con trỏ đến nút "hiện tại" trong lần lặp. Phần cứng đang tiến đến nút tiếp theo một cách hiệu quả. Có một số giải pháp khả thi, một số giải pháp được liệt kê ở đây:

**1**. Khi trình lặp được xây dựng, hãy để mỗi trình lặp lưu trữ dưới dạng dữ liệu của nó một mảng chứa các mục đã đặt. Điều này không hiệu quả: Nó khiến bạn không thể triển khai hiệu quả bất kỳ quy trình nào trả về trình vòng lặp sau khi sửa đổi tập hợp, chẳng hạn như một số phiên bản xóa và chèn.

**2**. Yêu cầu trình lặp duy trì một ngăn xếp lưu trữ các nút trên đường dẫn đến nút hiện tại. Với thông tin này, người ta có thể suy ra nút tiếp theo trong lần lặp, đó là nút trong cây con bên phải của nút hiện tại có chứa mục tối thiểu hoặc mục gần nhất

tổ tiên chứa nút hiện tại trong cây con bên trái của nó. Điều này làm cho trình vòng lặp hơi lớn và làm cho mã trình vòng lặp vụng về.

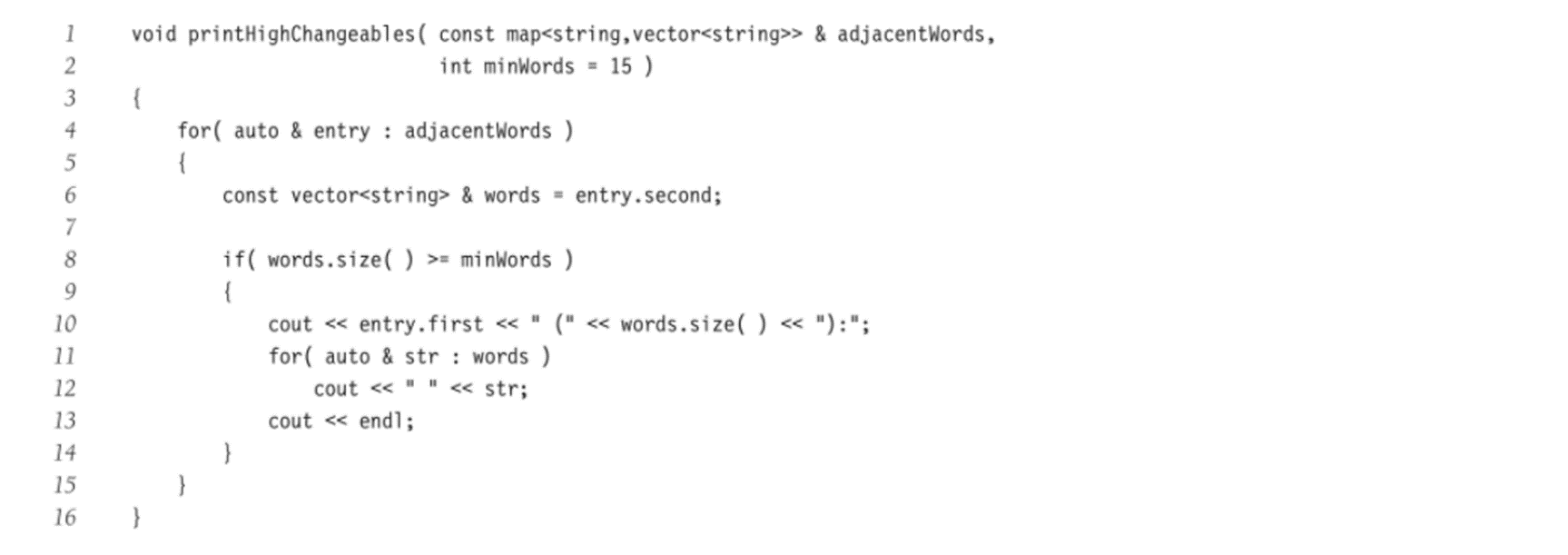
**3**. Có mỗi nút trong cây tìm kiếm lưu trữ cha của nó ngoài các nút con. Trình lặp không lớn bằng, nhưng hiện tại mỗi nút cần thêm bộ nhớ và mã để lặp vẫn còn vụng về.

**4**. Yêu cầu mỗi nút duy trì các liên kết bổ sung: một đến nút nhỏ hơn tiếp theo và một đến nút lớn hơn tiếp theo. Điều này tốn không gian, nhưng việc lặp lại rất đơn giản để thực hiện và dễ dàng duy trì các liên kết này.

**5**. Chỉ duy trì các liên kết bổ sung cho các nút có nullptr liên kết trái hoặc phải bằng cách sử dụng các biến Boolean bổ sung để cho phép các quy trình cho biết liệu một liên kết bên trái đang được sử dụng làm cây tìm kiếm nhị phân tiêu chuẩn liên kết bên trái hay một liên kết đến nút nhỏ hơn tiếp theo , và tương tự cho liên kết bên phải (Bài tập 4.49). Ý tưởng này được gọi là cây luồng và được sử dụng trong nhiều triển khai STL.

**4.8.4 Một ví dụ sử dụng một số bản đồ** (An Example That Uses Several Maps)

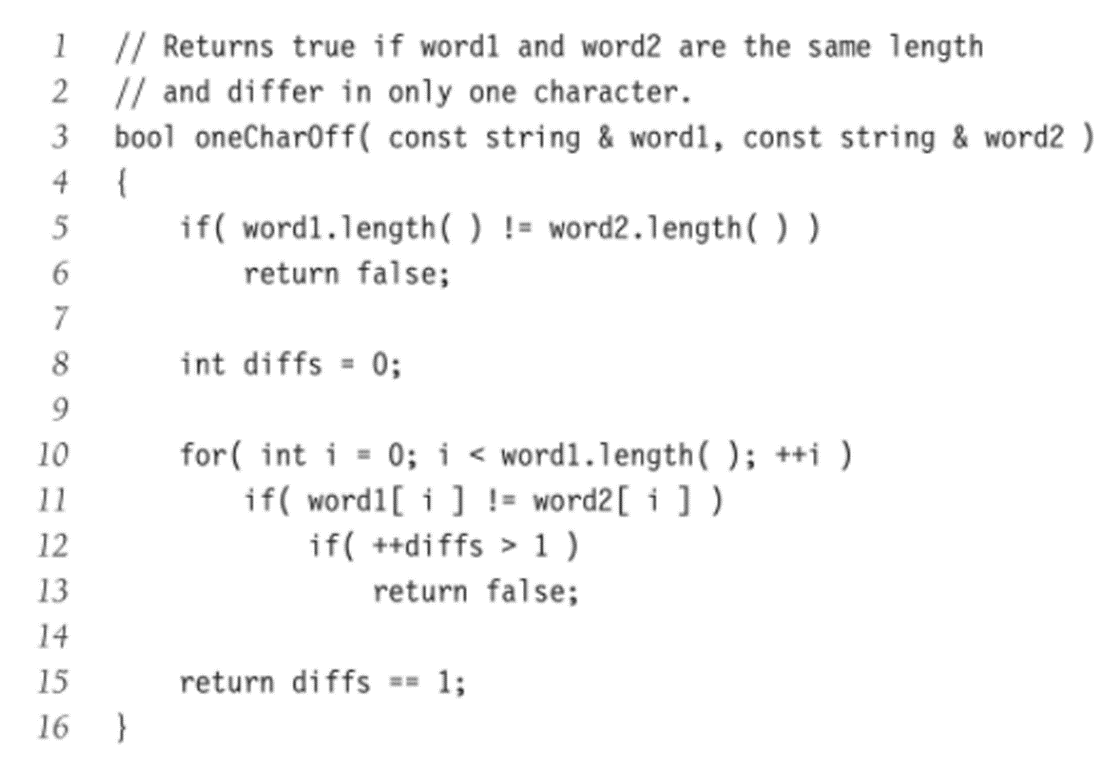
Nhiều từ tương tự với các từ khác. Ví dụ, bằng cách thay đổi chữ cái đầu tiên, từ wine có thể trở thành dine, fine, line, mine, nine, pine, hoặc wine. Bằng cách thay đổi chữ cái thứ ba, rượu có thể trở thành rộng, vợ, lau hoặc dây, trong số những người khác. Bằng cách thay đổi chữ cái thứ tư, rượu vang có thể trở thành gió, cánh, nháy mắt, hoặc chiến thắng, trong số những người khác. Điều này cho ra 15 từ khác nhau có thể thu được bằng cách chỉ thay đổi một chữ cái trong rượu vang. Trên thực tế, có hơn 20 từ khác nhau, một số từ khó hiểu hơn. Chúng tôi muốn viết một chương trình để tìm tất cả các từ có thể thay đổi thành ít nhất 15 từ khác bằng một phép thay thế một ký tự. Chúng tôi giả định rằng chúng tôi có một từ điển bao gồm khoảng 89.000 từ khác nhau với độ dài khác nhau. Hầu hết các từ có từ 6 đến 11 ký tự. Bản phân phối bao gồm 8.205 từ sáu chữ cái, 11.989 từ bảy chữ cái, 13.672 từ tám chữ cái, 13.014 từ chín chữ cái, 11.297 từ mười chữ cái và 8.617 từ mười một chữ cái. (Trên thực tế, những từ có thể thay đổi nhiều nhất là những từ có ba, bốn và năm chữ cái, nhưng những từ dài hơn là những từ tốn thời gian để kiểm tra.)

 Chiến lược đơn giản nhất là sử dụng một bản đồ trong đó các khóa là các từ và các giá trị là các vectơ chứa các từ có thể được thay đổi từ khóa bằng cách thay thế một ký tự. Quy trình trong Hình 4.69 cho thấy cách bản đồ được tạo đều (chúng tôi vẫn chưa viết mã cho phần đó) có thể được sử dụng như thế nào để in các câu trả lời được yêu cầu. Đoạn mã sử dụng một dải ô cho vòng lặp để đi qua bản đồ và xem các mục nhập là các cặp bao gồm một từ và một vectơ từ. Các tham chiếu không đổi ở dòng 4 và 6 được sử dụng để thay thế các biểu thức phức tạp và tránh tạo các bản sao không cần thiết.

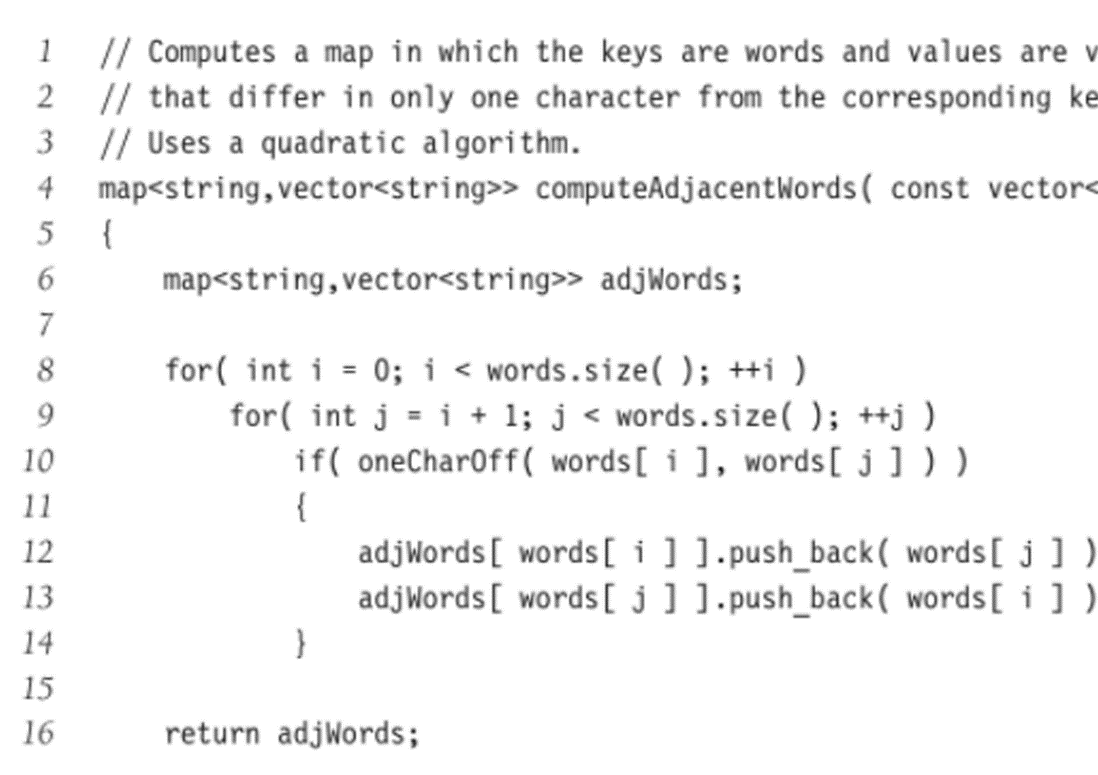
**Hình 4.69** Cho một bản đồ chứa các từ làm khóa và vectơ các từ chỉ khác nhau một ký tự làm giá trị, xuất ra các từ có từ min AdWords trở lên có thể có được bằng cách thay thế một ký tự

Vấn đề chính là làm thế nào để xây dựng bản đồ từ một mảng chứa 89.000 từ. Quy trình trong Hình 4.70 là một hàm đơn giản để kiểm tra xem hai từ có giống nhau không ngoại trừ sự thay thế một ký tự. Chúng tôi có thể sử dụng quy trình để cung cấp thuật toán đơn giản nhất cho việc xây dựng bản đồ, đó là một phép thử bạo lực của tất cả các cặp từ. Thuật toán này được thể hiện trong hình 4.71.

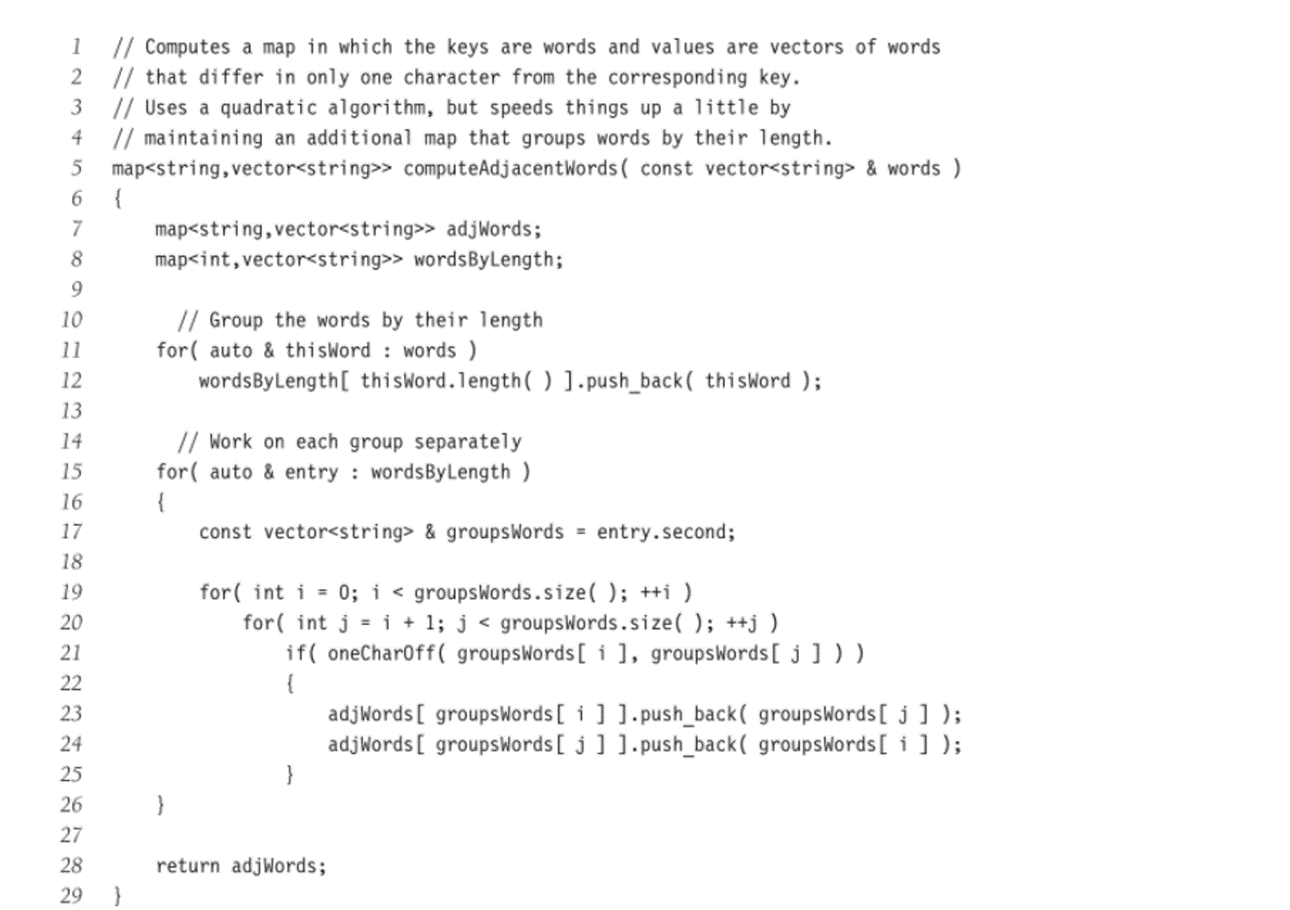
Vấn đề với thuật toán này là nó chậm và mất 97 giây trên com-puter của chúng tôi. Một cải tiến rõ ràng là tránh so sánh các từ có độ dài khác nhau. Chúng ta có thể làm điều này bằng cách nhóm các từ theo độ dài của chúng, và sau đó chạy thuật toán trước đó trên từng nhóm riêng biệt.

 Để làm điều này, chúng ta có thể sử dụng bản đồ thứ hai! Ở đây, khóa là một số nguyên đại diện cho độ dài từ và giá trị là tập hợp của tất cả các từ có độ dài đó. Chúng ta có thể sử dụng một vectơ để lưu trữ từng bộ sưu tập và áp dụng thành ngữ tương tự. Mã được hiển thị trong Hình 4.72. Dòng 8 hiển thị khai báo cho bản đồ thứ hai, dòng 11 và 12 điền vào bản đồ, sau đó một vòng lặp bổ sung được sử dụng để lặp lại từng nhóm từ. So với thuật toán đầu tiên, thuật toán thứ hai chỉ khó viết mã hơn một chút và chạy trong 18 giây, tức là nhanh hơn khoảng sáu lần.

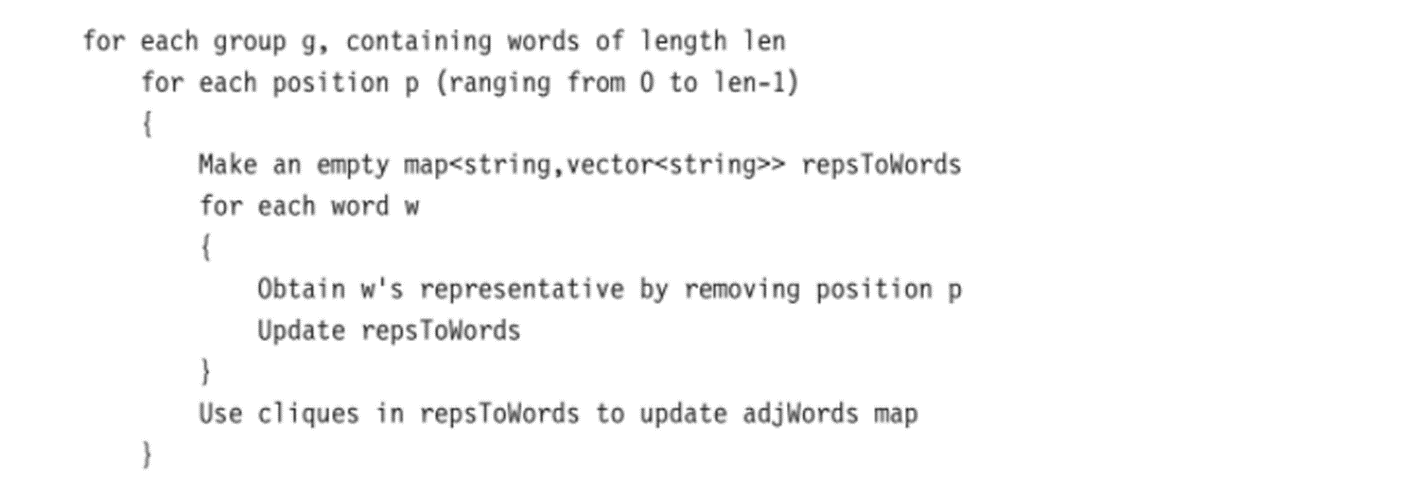
**Hình 4.70** Quy trình kiểm tra xem hai từ chỉ khác nhau một ký tự hay không



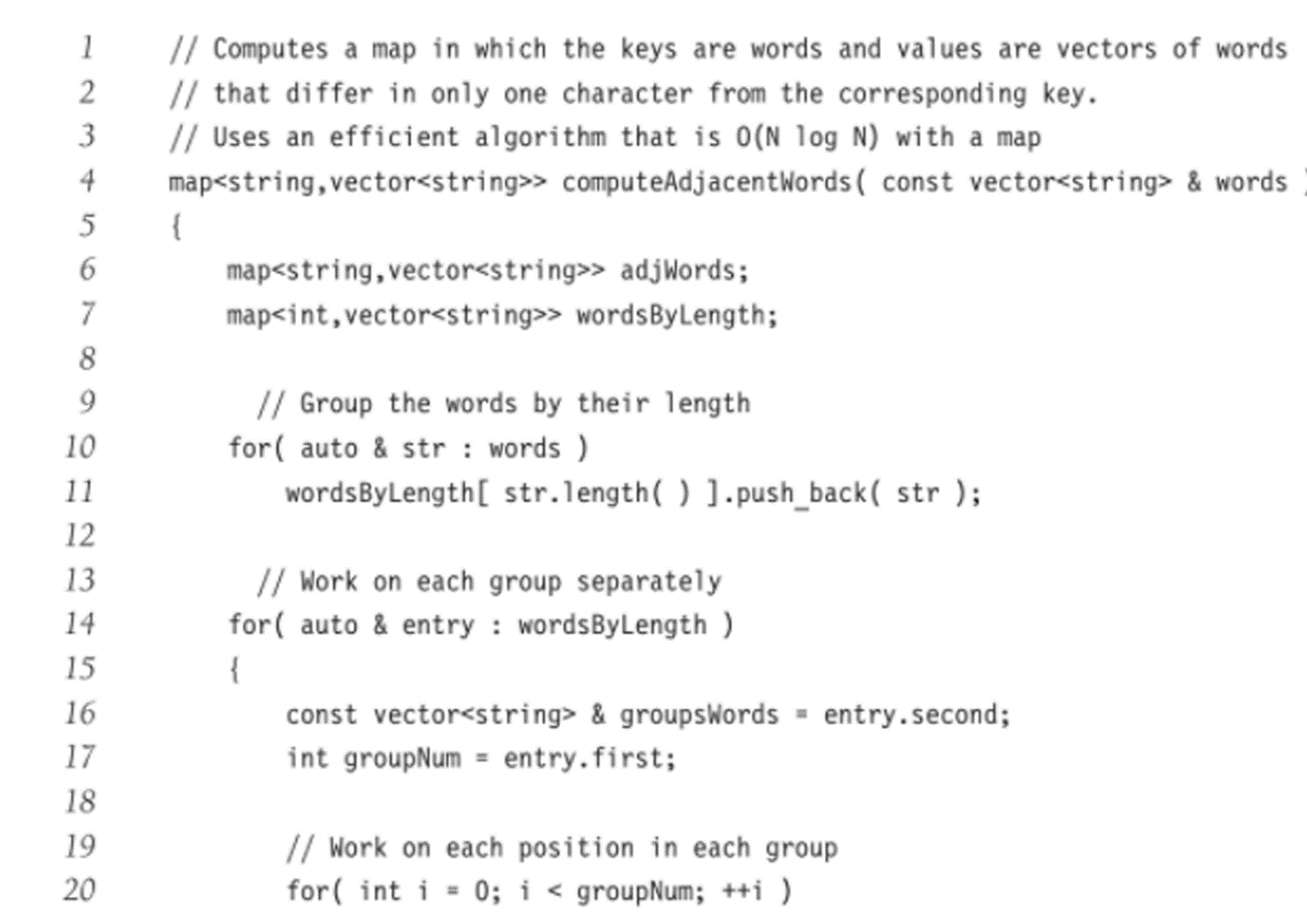
**Hình 4.71** Chức năng tính toán một bản đồ chứa các từ làm khóa và vectơ các từ chỉ khác nhau một ký tự làm giá trị. Phiên bản này chạy trong 1,5 phút trên một từ điển 89.000 từ.

**Hình 4.72** Hàm tính toán một bản đồ chứa các từ làm khóa và vectơ các từ chỉ khác nhau một ký tự làm giá trị. Nó chia các từ thành các nhóm theo độ dài từ. Phiên bản này chạy trong 18 giây trên một từ điển 89.000 từ.

Thuật toán thứ ba của chúng tôi phức tạp hơn và sử dụng các bản đồ bổ sung! Như trước đây, chúng tôi nhóm các từ theo độ dài từ, và sau đó làm việc trên từng nhóm riêng biệt. Để xem thuật toán này hoạt động như thế nào, giả sử chúng ta đang làm việc với các từ có độ dài 4. Trước tiên, chúng ta muốn tìm các cặp từ, chẳng hạn như rượu và số chín, giống hệt nhau ngoại trừ chữ cái đầu tiên. Một cách thực hiện như sau: Đối với mỗi từ có độ dài 4, loại bỏ ký tự đầu tiên, để lại đại diện từ ba ký tự. Tạo một bản đồ trong đó khóa là đại diện và giá trị là vectơ của tất cả các từ có đại diện đó. Ví dụ: khi xem xét ký tự đầu tiên của nhóm từ bốn chữ cái, đại diện "ine" tương ứng với "ăn tối", "tốt", "rượu", "chín", "của tôi", "cây nho", "cây thông", "hàng". Đại diện "oot" tương ứng với "boot", "foot", "hoot", "loot", "muội", "zoot". Mỗi vectơ riêng lẻ là một giá trị trong bản đồ mới nhất này tạo thành một cụm từ trong đó bất kỳ từ nào cũng có thể được thay đổi thành bất kỳ từ nào khác bằng cách thay thế một ký tự, vì vậy sau khi bản đồ mới nhất này được xây dựng, bạn có thể dễ dàng xem qua và thêm các mục vào bản đồ gốc đang được tính toán. Sau đó chúng tôi sẽ tiếp tục ký tự thứ hai của nhóm từ bốn chữ cái, với một bản đồ mới, rồi đến ký tự thứ ba, và cuối cùng là ký tự thứ tư.

Đại cương là

**Hình 4.73** chứa cách triển khai thuật toán này. Thời gian chạy cải thiện thành hai giây. Điều thú vị là mặc dù việc sử dụng các bản đồ bổ sung làm cho thuật toán nhanh hơn và cú pháp tương đối rõ ràng, nhưng mã không sử dụng thực tế là các khóa của bản đồ được duy trì theo thứ tự được sắp xếp.

**Hình 4.73** Hàm tính toán một bản đồ chứa các từ làm khóa và vectơ các từ chỉ khác nhau một ký tự làm giá trị. Phiên bản này chạy trong 2 giây trên một từ điển 89.000 từ.

**Hình 4.73** (tiếp theo)

Do đó, có thể cấu trúc dữ liệu hỗ trợ các hoạt động bản đồ nhưng không đảm bảo thứ tự được sắp xếp có thể hoạt động tốt hơn, vì nó đang được yêu cầu thực hiện ít hơn. Chương 5 khám phá khả năng này và thảo luận về những ý tưởng đằng sau việc triển khai bản đồ thay thế mà C ++ 11 thêm vào Thư viện Chuẩn, được gọi là bản đồ không có thứ tự. Một bản đồ không có thứ tự giảm thời gian thực hiện từ 2 giây xuống 1,5 giây.

**Tóm lược** (Summary)

Chúng tôi đã thấy việc sử dụng cây trong hệ điều hành, thiết kế trình biên dịch và tìm kiếm. Cây biểu thức là một ví dụ nhỏ về cấu trúc tổng quát hơn được gọi là cây phân tích cú pháp, là cấu trúc dữ liệu trung tâm trong thiết kế trình biên dịch. Cây phân tích cú pháp không phải là cây nhị phân, nhưng là phần mở rộng tương đối đơn giản của cây biểu thức (mặc dù các thuật toán để xây dựng chúng không hoàn toàn đơn giản).

Cây tìm kiếm có tầm quan trọng lớn trong thiết kế thuật toán. Chúng hỗ trợ hầu hết các hoạt động hữu ích và chi phí trung bình lôgarit là rất nhỏ. Các phép nhập không đệ quy của cây tìm kiếm có phần nhanh hơn, nhưng các phiên bản đệ quy đẹp hơn, thanh lịch hơn và dễ hiểu và dễ gỡ lỗi hơn. Vấn đề với cây tìm kiếm là hiệu suất của chúng phụ thuộc nhiều vào đầu vào là ngẫu nhiên. Nếu không đúng như vậy, thời gian chạy sẽ tăng lên đáng kể, đến mức cây tìm kiếm trở thành danh sách liên kết đắt đỏ.

Chúng tôi đã thấy một số cách để giải quyết vấn đề này. Cây AVL hoạt động bằng cách nhấn mạnh rằng tất cả các cây con bên trái và bên phải của tất cả các nút khác nhau về độ cao nhiều nhất là một. Điều này đảm bảo rằng cây không thể bị sâu quá. Các thao tác không thay đổi cây, như thao tác chèn, đều có thể sử dụng mã cây tìm kiếm nhị phân tiêu chuẩn. Các thao tác làm thay đổi cây phải phục hồi cây. Điều này có thể hơi phức tạp, đặc biệt là trong trường hợp xóa. Chúng tôi đã chỉ ra cách khôi phục cây sau khi chèn trong thời gian O (logN).

Chúng tôi cũng đã kiểm tra cây splay. Các nút trong cây lan có thể sâu tùy ý, nhưng sau mỗi lần tiếp cận, cây sẽ được điều chỉnh theo cách hơi bí ẩn. Hiệu quả thực là bất kỳ chuỗi M hoạt động nào cũng mất O (M logN) thời gian, giống như một cây cân bằng sẽ mất.

Cây B là cây M cách cân bằng (trái ngược với cây 2 chiều hoặc nhị phân), rất thích hợp cho đĩa; một trường hợp đặc biệt là cây 2–3 (M = 3), là một cách khác để thực hiện cây tìm kiếm cân bằng.

Trong thực tế, thời gian chạy của tất cả các lược đồ cây cân bằng, mặc dù nhanh hơn một chút để tìm kiếm, nhưng lại kém hơn (bởi hệ số không đổi) đối với việc chèn và xóa so với cây tìm kiếm nhị phân đơn giản, nhưng điều này nói chung có thể chấp nhận được về mặt bảo vệ được đưa ra so với đầu vào trường hợp xấu nhất dễ dàng thu được. Chương 12 thảo luận về một số cấu trúc dữ liệu cây tìm kiếm bổ sung và cung cấp các triển khai chi tiết.