NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 2
CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

NGUYỄN HỮU ĐIỂN

THUẬT TOÁN VÀ LẬP TRÌNH

QUYỂN 2
CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

LỜI NÓI ĐẦU

Những năm trước khi lập trình VieTeX tôi toàn dùng C/C++ thu thập tài liệu nhiều nhưng không có thời gian để viết lại. Nay muốn viết lại thì sức khỏe không ổn định. Tôi đã cố gắng gom lại thành các tập lập trình theo chủ đề. Nội dung mỗi thuật toán bắt đầu từ lý thuyết đến lập trình bằng C/C++.

Cuốn sách viết ra không dành riêng cho các bạn học tin học, mà các bạ học toán, thầy cô giáo, các bạn thích tìm hiểu về thuật toán. Cũng như tôi bắt đầu có biết gì về lập trình đâu, tự học và chăm chỉ là thành công thôi. Tôi dùng trình biên dịch Dev-C++: https://www.bloodshed.net/
Hiện nay Dev-C++ cải tiến rất nhiều và chạy tốt với môi trường unicode. Những ví dụ trong tài liệu các bạn chép thẳng vào soạn thảo và biên dịch không cần cấu hình trình biên dịch.

Tôi đã làm các quyển sách:

1. Thuật toán và số học.
2. Thuật toán và dữ liệu.
3. Thuật toán xắp xếp
4. Thuật toán tìm kiếm
5. Thuật toán đồ thị,
6. Thuật toán quay lui
7. Thuật toán chia để trị
8. Thuật toán động
9. Thuật toán tham
10. Thuật toán nén
11. Một số đề thi Olympic Tin học.

Cuốn sách dành cho học sinh phổ thông yêu toán, học sinh khá giỏi môn toán, các thầy cô giáo, sinh viên đại học ngành toán, ngành tin học và những người yêu thích Toán - Tin. Trong biên soạn không thể tránh khỏi sai sót và nhầm lẫn mong bạn đọc cho ý kiến.

Hà Nội, ngày 25 tháng 2 năm 2022

Nguyễn Hữu Điển

NHỮNG KÝ HIỆU

Trong cuốn sách này ta dùng những kí hiệu với các ý nghĩa xác định trong bảng dưới đây:

$\mathbb{N}$	tập hợp số tự nhiên
$\mathbb{N}^*$	tập hợp số tự nhiên khác 0
$\mathbb{Z}$	tập hợp số nguyên
$\mathbb{Q}$	tập hợp số hữu tỉ
$\mathbb{R}$	tập hợp số thực
$\mathbb{C}$	tập hợp số phức
$\equiv$	dấu đồng dư
∞	dương vô cùng (tương đương với $+\infty$)
$-\infty$	âm vô cùng
$\emptyset$	tập hợp rỗng
C_m^k	tổ hợp chập k của m phần tử
$\vdots$	phép chia hết
$\not\vdots$	không chia hết
$UCLN$	ước số chung lớn nhất
$BCNN$	bội số chung nhỏ nhất
deg	bậc của đa thức
IMO	International Mathematics Olympiad
APMO	Asian Pacific Mathematics Olympiad

NỘI DUNG

Lời nói đầu iii
Những kí hiệu iv
Mục lục iv
Danh sách hình vii
Danh sách bảng ix
Chương 2. Cấu trúc dữ liệu và thuật toán **153**
2.1. Giới thiệu 154
2.2. Danh sách, ngăn xếp và hàng đợi 157
2.2.1. Danh sách 157
2.2.2. Ngăn xếp 159
2.2.3. Hàng đợi 160
2.2.4. Hàng đợi hai đầu 161
2.3. Thực hiện cụ thể cấu trúc trên 161
2.3.1. Danh sách 161
2.3.2. Ngăn xếp 162
2.3.3. Hàng đợi 164
2.4. Thể hiện cấu trúc động 168
2.4.1. Đưa vào một phần tử 169
2.4.2. Cuộn qua một danh sách 169
2.4.3. Đưa vào sau một phần tử được chỉ định bởi một chỉ dẫn đã cho 171
2.4.4. Đưa vào phía trước một phần tử được chỉ định bởi một thư mục 171
2.4.5. Xóa theo khóa mặc định và con trỏ lên đầu danh sách ... 172
2.4.6. Xóa một phần tử được chỉ định bởi một thư mục .. 173
2.5. Cây nhị phân 178
2.5.1. Tìm kiếm bằng chìa khóa 183
2.5.2. Thêm vào một đỉnh mới 184
2.5.3. Xóa một đỉnh bằng từ khóa đã cho 184

2.5.4. Thu thập thông tin 189
2.5.5. Bài tập 191
2.6. Cây cân đối 192
2.6.1. Vòng xoay. Cây đỏ và đen 196
2.6.2. B-Cây 198
2.7. Bảng băm (H-bảng) 201
2.7.1. Hàm băm (H-hàm số) 202
2.7.2. Sự xung đột 203
2.7.3. Hàm băm cổ điển 205
2.7.4. Đối phó với xung đột 214
2.7.5. Triển khai bảng băm 218
2.8. Câu hỏi và bài tập 226

DANH SÁCH CÁC HÌNH

2.1 Ngăn xếp 159
2.2 Hàng đợi 160
2.3 Danh sách chia bộ nhớ 162
2.4 Danh sách chia bộ nhớ 165
2.5 Hàng đợi thể hiện quả mảng chu kỳ 165
2.6 Danh sách liên kết động 168
2.7 Đưa vào đầu và dịch chuyển con trỏ 170
2.8 Đưa vào sau phần tử chỉ ra từ bảng chỉ dẫn 172
2.9 Loại trừ khỏi danh sách. 173
2.10 Biểu diễn trực quan thông qua các tập hợp lồng nhau. . . 179
2.11 Trình bày trong ngoặc đơn. 179
2.12 Biểu diễn bằng đồ thị liên thông xoay chiều có định hướng (xem Chương 5). 180
2.13 Biểu diễn bằng một đồ thị liên thông xoay chiều không có hướng (xem Chương 5), với một trong các đỉnh được chọn làm gốc: trong trường hợp này là đỉnh A. 180
2.14 Biểu diễn bằng phép dời hình từng bước. 180
2.15 Cây nhị phân 181
2.16 Cây 181
2.17 Trình bày biểu thức $a + ((c * d) * (e \breve{} b))$ 181
2.18 Trình bày động của một cây nhị phân trong bộ nhớ. . . . 182
2.19 Cây tìm kiếm nhị phân. 183
2.20 Xóa kết nối đầu bằng từ khóa 5. 185
2.21 Xóa kết nối đầu bằng từ khóa 10. 185
2.22 Trường hợp xấu nhất với cây nhị phân. 192
2.23 Cây Fibonacci của hàng 0. 193
2.24 Cây Fibonacci của hàng 1. 193
2.25 Cây Fibonacci của hàng 2. 194

2.26 Cây Fibonacci của hàng 3. 194
2.27 Cây Fibonacci của hàng 4. 194
2.28 Cây Fibonacci của hàng 5. 195
2.29 Xoay phải (A), trái (B), xoay trái-phải (C) và xoay phải-trái (D) quay. 197
2.30 Ví dụ về cây đỏ-đen. 198
2.31 2-3-4 cây. 199
2.32 B-cây hàng 5. 200
2.33 Hàm băm khớp với địa chỉ của từng khóa trong bảng băm. 203
2.34 χ^2 so sánh các hàm băm trên PIN, $n = 1031$, $m = 1031000$. 207
2.35 Bao gồm tuần tự của một phần tử. Cho phép xung đột với thử nghiệm tuyến tính với bước 1. 215
2.36 Giải quyết xung đột bằng cách phân bổ bộ nhớ bổ sung. . 216
2.37 Kết nối động: bảng băm, sau khi bao gồm các phần tử 234, 235, 567, 123, 534, 647. 217

DANH SÁCH CÁC BẢNG

Danh sách chương trình

2.1 Ngăn xếp (202stack2.c) 163
2.2 Hàng đợi (203queue.c) 166
2.3 Thao tác trên danh sách (204list.c) 173
2.4 Thao tác trên cây nhị phân (205bintree.c) 185
2.5 Bảng băm (206hash.c) 218
2.6 Bảng băm tập hợp (206hashset.c) 222

CHƯƠNG 2
CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

2.1. Giới thiệu **154**
2.2. Danh sách, ngăn xếp và hàng đợi **157**
2.2.1. Danh sách 157
2.2.2. Ngăn xếp 159
2.2.3. Hàng đợi 160
2.2.4. Hàng đợi hai đầu 161
2.3. Thực hiện cụ thể cấu trúc trên **161**
2.3.1. Danh sách 161
2.3.2. Ngăn xếp 162
2.3.3. Hàng đợi 164
2.4. Thể hiện cấu trúc động **168**
2.4.1. Đưa vào một phần tử 169
2.4.2. Cuộn qua một danh sách 169
2.4.3. Đưa vào sau một phần tử được chỉ định bởi một chỉ dẫn đã cho 171
2.4.4. Đưa vào phía trước một phần tử được chỉ định bởi một thư mục 171
2.4.5. Xóa theo khóa mặc định và con trỏ lên đầu danh sách 172
2.4.6. Xóa một phần tử được chỉ định bởi một thư mục 173
2.5. Cây nhị phân **178**
2.5.1. Tìm kiếm bằng chìa khóa 183
2.5.2. Thêm vào một đỉnh mới 184
2.5.3. Xóa một đỉnh bằng từ khóa đã cho 184
2.5.4. Thu thập thông tin 189
2.5.5. Bài tập 191
2.6. Cây cân đối **192**
2.6.1. Vòng xoay. Cây đỏ và đen 196
2.6.2. B-Cây 198
2.7. Bảng băm (H-bảng) **201**
2.7.1. Hàm băm (H-hàm số) 202

2.7.2. Sự xung đột 203
2.7.3. Hàm băm cổ điển 205
2.7.4. Đối phó với xung đột 214
2.7.5. Triển khai bảng băm 218
2.8. Câu hỏi và bài tập 226

"Một chương trình không có vòng lặp và một biến có cấu trúc không đáng để viết."

Epigram trong lập trình

2.1. Giới thiệu

Điều quan trọng nhất đối với giải pháp hiệu quả của mỗi bài toán là sự lựa chọn của một thuật toán thích hợp. Tùy thuộc vào sự lựa chọn này liệu vấn đề sẽ được giải quyết chính xác hay xấp xỉ, một phần hay toàn bộ, thời gian tính toán và nguồn lực cần thiết để giải quyết nó, v.v. Đổi lại, hiệu quả của từng thuật toán cụ thể cũng phụ thuộc vào cách nó sẽ được thực hiện. Sự phức tạp của nó liên quan chặt chẽ đến cách thức mà các đối tượng trong bài toán sẽ được trình bày và tổ chức, nghĩa là với sự lựa chọn *cấu trúc dữ liệu* thích hợp. Để hiểu tại sao lại như vậy, chỉ cần suy nghĩ sâu hơn về công việc của một chương trình máy tính là đủ - ngoài thuật toán đã chọn, phần lớn thời gian tính toán được dành cho các quy trình liên quan đến truy cập, lưu trữ và xử lý dữ liệu.

Một trong những ví dụ đơn giản nhất về cấu trúc dữ liệu là *các biến* kiểu đơn giản (số nguyên/số thực, ký hiệu). Chúng được hỗ trợ bởi hầu hết các ngôn ngữ lập trình và được sử dụng trong hầu hết các chương trình. Cấu trúc của chúng là không đổi - chúng chỉ có thể chấp nhận các giá trị được chấp nhận cho kiểu của chúng và chiếm một lượng bộ nhớ không đổi (nghĩa là không thể thay đổi trong quá trình hoạt động của chương trình).

Một cấu trúc dữ liệu cơ bản khác là *một mảng*: một số cố định các biến cùng kiểu được lưu trữ tuần tự trong bộ nhớ máy tính. Việc truy cập vào từng biến này (tức là vào các phần tử của mảng) được thực hiện bởi sự kết hợp giữa tên của mảng và chỉ số của biến tương

ứng. Ưu điểm chính của việc làm việc với mảng là quyền truy cập vào phần tử thứ k là trực tiếp (tuy nhiên, không giống như các biến đơn giản, có địa chỉ gián tiếp: địa chỉ của đầu cộng với phần bù được tính toán). Mặt khác, kích thước của mảng là cố định - điều này có nghĩa là thứ nhất, bộ nhớ được cấp phát khi khai báo mảng vẫn bị chiếm trong suốt chương trình và thứ hai, nếu kích thước của mảng không đủ cho nhu cầu của chương trình, phóng đại chỉ có thể được thực hiện bằng cách sao chép các phần tử vào một mảng lớn.

Các biến và mảng đơn giản được định nghĩa là *cấu trúc dữ liệu tĩnh*. Hầu hết các ngôn ngữ lập trình cũng cung cấp khả năng cấp phát bộ nhớ động. Với nó, bộ nhớ được dự trữ khi cần thiết và được giải phóng khi không còn cần thiết. Việc sao lưu và giải phóng diễn ra trong quá trình hoạt động của chương trình, dẫn đến việc sử dụng bộ nhớ linh hoạt và hiệu quả, cũng như khả năng mô hình hóa các cấu trúc dữ liệu phức tạp hơn có logic riêng và "tuổi thọ" thuật toán của riêng chúng.

Nói chung, việc biểu diễn và mô hình hóa các đối tượng thực trong chương trình máy tính, cũng như các hoạt động được thực hiện trên chúng, có thể được chia thành hai bài toán con:

1) *Định nghĩa cấu trúc dữ liệu trừu tượng* (ADS): đây là cách mà các đối tượng thực sẽ được mô hình hóa thành các đối tượng toán học, cũng như xác định tập hợp các phép toán cho phép trên chúng.

2) *Hiện thực hóa các cấu trúc dữ liệu trừu tượng*: Đây là cách mà các đối tượng toán học đã xác định sẽ được biểu diễn trong bộ nhớ máy tính (thông qua các kiểu đơn giản hoặc dưới dạng kết hợp các triển khai có sẵn của ASD khác), cũng như cách thức mà chúng sẽ được hiện thực hóa hoạt động với chúng.

Nói cách khác, mục đích của việc xác định cấu trúc dữ liệu trừu tượng là xác định những gì có thể được thực hiện với các đối tượng, và việc thực hiện là cách thực hiện.

Chúng ta sẽ minh họa những gì đã được nói cho đến nay bằng một ví dụ cụ thể. Hãy trở thành một bộ bài 52 lá. Để biểu diễn nó, chúng ta có thể sử dụng một cấu trúc dữ liệu trừu tượng - *một tập hợp* có các phần tử là quân bài. Chúng ta xem xét các hoạt động sau:

- *loại trừ* bất kỳ phần tử nào khỏi bộ (loại bỏ bất kỳ thẻ nào khỏi

bộ bài).
- *Đưa vào* một phần tử trong tập hợp (thêm một quân bài).
- *kiểm tra* xem một phần tử có thuộc bộ hay không (kiểm tra xem một thẻ có trong bộ bài hay không).

Việc thực hiện bộ này cũng rất mơ hồ. Ví dụ: mỗi phần tử của nó (bộ bài) có thể được biểu diễn bằng hai ký hiệu - cho loại quân bài ('2', ..., '9', 'T', 'J', 'Q', 'K', ' A ') và cho quân bài vẽ (' S ',' C ',' D ',' H '). Có thể biểu diễn khác - trên mỗi thẻ để so sánh một số tự nhiên duy nhất giữa 0 và 51 (số thẻ khác nhau là 52). Ngoài ra, chúng ta phải chọn một đại diện của tập hợp và thực hiện các phép toán được phép trên nó. Chúng ta sẽ hiển thị một khả năng hiện thực hóa (chúng ta giả định rằng các thẻ được đánh dấu bằng các số nguyên từ 0 đến 51):

Chúng ta sẽ sử dụng một mảng `int cards[52]`. Giá trị của thẻ `cards[i]` sẽ là 1 nếu thẻ có số i nằm trong bộ bài. Nếu không giá trị của các thẻ `cards[i]` sẽ là 0. Như vậy, để thêm thẻ có số k, chỉ cần thực hiện các thẻ gán `[k] = 1` là đủ; . Phép toán loại trừ một phần tử tùy ý được thực hiện như sau: chúng ta tìm một số k cho thẻ nào `cards[k] == 1` và gán thẻ `cards[k] = 0`. Việc kiểm tra xem một quân bài có số k có trong bộ bài hay không đưa vào một phép thử duy nhất về giá trị của `cards[k]`.

Vấn đề thực hiện là tối quan trọng trong việc giải quyết bài toán cụ thể. Trong ví dụ về bộ bài, việc thực hiện một tập hợp bằng cách sử dụng một mảng `cards[52]` là thích hợp, vì các quân bài chỉ có 52. Tuy nhiên, có thể các phần tử của tập hợp đó lớn hơn nhiều. Ví dụ, hãy đánh số các sinh viên trong một trường đại học - mỗi sinh viên được gán một số tự nhiên duy nhất. Nếu chúng ta sử dụng một chuyển đổi tương tự như trong ví dụ bản đồ để biểu diễn số lượng sinh viên trong một nhóm hành chính (giả sử rằng không có hơn 30.000 sinh viên tại trường đại học), sẽ có một sự lãng phí bộ nhớ lớn: toàn bộ một mảng `int students[30000]` , trong khi số học sinh trong một nhóm thực sự không quá 30. Chúng ta có thể tiếp cận theo cách khác: nhập một biến số nguyên `unsigned n`, biểu thị số học sinh trong nhóm và một mảng `unsigned students[n]` sao cho vị trí thứ i ($0 \leq i < n$) để ghi số sinh viên.

Nói chung, việc lựa chọn cách triển khai không chỉ phụ thuộc

vào dung lượng bộ nhớ sẽ được sử dụng, mà còn phụ thuộc vào độ phức tạp tính toán của mỗi hoạt động. Trong chương này, chúng ta sẽ làm quen với các cấu trúc dữ liệu trừu tượng chính, cũng như hai cách tiếp cận quan trọng để triển khai chúng - tuần tự (tĩnh, như chúng ta đã áp dụng trong các ví dụ được liệt kê ở trên) và kết nối (động).

2.2. Danh sách, ngăn xếp và hàng đợi

2.2.1. Danh sách

Định nghĩa 2.1. Một *danh sách tuyến tính* (hay chỉ một danh sách) được gọi là một dãy gồm $n (n \geq 0)$ phần tử $x_1, x_2, ..., x_n$, được sắp xếp tuần tự (tuyến tính). Tại $n = 0$, danh sách được gọi là *rỗng*. Nếu $n > 0$, thì x_1 được gọi là *phần tử đầu tiên* trong danh sách và x_n - *phần tử cuối cùng*.

Mối quan hệ duy nhất giữa các phần tử trong cấu trúc này được xác định bởi các điều kiện: Với mỗi i $(1 \leq i \leq n)$ phần tử thứ i, x_i đứng trước phần tử x_{i-1} và theo sau là phần tử x_{i+1}.

Trên thực tế, từ "tuyến tính" là thừa: danh sách phi tuyến tính có tên đặc biệt - cột, cây, v.v., và không được coi là danh sách. Chúng ta sử dụng nó để tuân thủ thuật ngữ được chấp nhận chung, nhưng rất tiếc là sai.

Trong thực tế, bởi "phần tử" trong một danh sách, chúng ta có nghĩa là một cấu trúc dữ liệu (trường) tùy ý. Thông thường, các phần tử của danh sách là các kiểu dữ liệu đơn giản và có cấu trúc giống hệt nhau.

Hãy biểu thị kiểu dữ liệu mà mỗi phần tử chứa dữ liệu và xem xét ví dụ sau: Chúng ta muốn xây dựng một danh sách sinh viên trong một trường đại học. Mỗi mục trong danh sách có thể chứa các thông tin sau:

Cấu trúc dữ liệu

```
struct data {
short age; / *Tuổi của sinh viên (nhỏ hơn 128)* /
char sex; / *Gới tính sinh viên, 'm' - nam;'f' - nữ* /
unsigned fn; / *Số báo danh sinh viên* /
```

```
char *name; / *Tên sinh viên* /
};
```

Hãy xem xét các hoạt động chính có thể được thực hiện trên một danh sách tuyến tính:

- Có được quyền truy cập vào phần tử thứ k trong danh sách (và có thể *thay đổ*i giá trị của bất kỳ trường nào của nó).
- *Đưa vào* (chèn) một mục trong danh sách (trước hoặc sau một mục, cũng như trong danh sách trống)
- *Loại trừ* (xóa) mục thứ k khỏi danh sách.
- *Tìm* tất cả các mục trong danh sách có chứa một giá trị (trong một trường hoặc tập hợp các trường nhất định).

Nhiều hoạt động khác có thể được định nghĩa - sắp xếp, hợp nhất, lật danh sách, v.v. Các hoạt động này mang tính thuật toán cao hơn và sẽ được thảo luận ở phần sau của cuốn sách, và trong chương này, chúng ta sẽ giới hạn bản thân chỉ xem xét các nguyên tắc cơ bản trong việc xác định và triển khai cấu trúc dữ liệu.

Nói chung, thao tác tìm kiếm có thể được thực hiện theo một tiêu chí khác: ví dụ, trong danh sách sinh viên được xác định ở trên, chúng ta có thể quan tâm đến tất cả sinh viên dưới 22 tuổi hoặc tất cả sinh viên có số giảng viên là số nguyên tố.

Từ bây giờ, khi chúng ta nói về các phần tử của một cấu trúc, để thuận tiện, chúng ta sẽ giả sử rằng cùng với dữ liệu mà nó chứa, mỗi phần tử nhất thiết sẽ có một trường, trường này được gọi là khóa của phần tử. Việc tìm kiếm một mục trong danh sách sẽ chỉ được thực hiện bằng khóa này.

Các hoạt động được xác định ở trên không thể được thực hiện theo cách mà chúng đều có hiệu quả cùng một lúc (ví dụ: tất cả chúng đều có độ phức tạp không đổi). Ngoài ra, các trường hợp $k = 1$ và $k = n$ đặc biệt hơn - có thể truy cập chúng với độ phức tạp ít hơn so với truy cập vào các mục khác trong danh sách. Hơn nữa, thường chỉ cần thực hiện các thao tác trong danh sách trên phần tử đầu tiên và/hoặc phần tử cuối cùng, điều này dẫn đến việc định nghĩa các cấu trúc dữ liệu cụ thể:

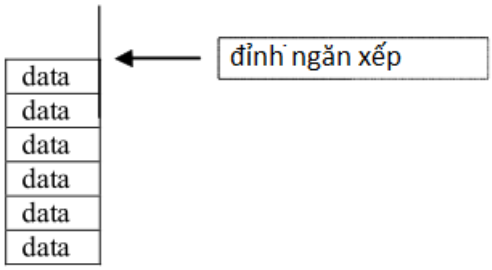
Bài tập:

▷ **2.1.** Tại sao không thể thực hiện tất cả bốn hoạt động cơ bản một

cách hiệu quả như nhau, ví dụ, có độ phức tạp liên tục?

2.2.2. Ngăn xếp

Ngăn xếp là một danh sách tuyến tính trong đó tất cả các hoạt động (bật, tắt, v.v.) chỉ được thực hiện ở một đầu, tức là chỉ trên phần tử đầu tiên hoặc chỉ trên phần tử cuối cùng. Người ta chấp nhận rằng phần tử mà các hoạt động được thực hiện được gọi là phần trên cùng của ngăn xếp (xem Hình 2.1).



Hình 2.1. Ngăn xếp

Các phép toán trừu tượng cổ điển trên ngăn xếp S như sau:

Thao tác trên ngăn xếp

```
void init(S);   // khởi tạo ngăn xếp trống
void push(S, data x); // thêm một phần tử mới vào đầu ngăn xếp
data pop(S);    // lấy vật phẩm từ trên cùng của ngăn xếp
int isEmpty(S); // kiểm tra xem ngăn xếp có trống không
```

Các hoạt động bổ sung của ngăn xếp thường được thực hiện: lấy một phần tử từ trên cùng của ngăn xếp mà không cần tắt nó; thêm/bớt k phần tử, v.v.

Nhiều ví dụ về ngăn xếp có thể được đưa ra. Từ stack dịch sang tiếng Việt là đống. Sự tương đồng là hiển nhiên - nếu chúng ta nhìn vào một đống báo (hoặc sách) trong một chiếc hộp, chúng ta có thể dễ dàng lấy tờ báo từ trên cùng hoặc thêm một tờ mới ở trên cùng. Người ta cũng nói rằng ngăn xếp tuân theo kỷ luật LIFO (Last-In-First-Out Vào sau, ra trước).

2.2.3. Hàng đợi

Hàng đợi là một danh sách tuyến tính trong đó thao tác bật chỉ có thể được thực hiện ở một đầu của danh sách và hoạt động tắt - ở đầu kia (Hình 2.2).

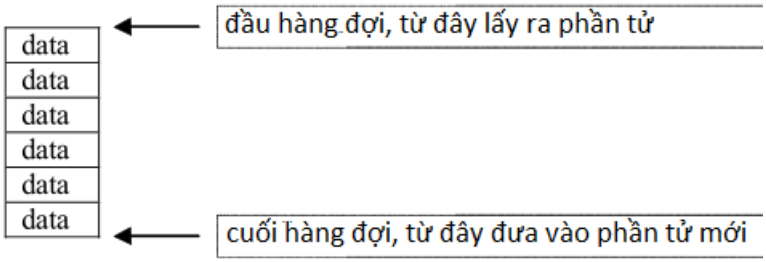
Các thao tác cổ điển trên hàng đợi Q như sau:

Thao tác trên hàng đợi

```
void init(Q); //khởi tạo hàng đợi trống
void put(Q, data x); // thêm một phần tử (ở cuối hàng đợi)
data get(Q);//truy xuất phần tử đầu tiên của hàng đợi
                    //(trả về phần tử đầu tiên, sau đó xóa nó)
int isEmpty(Q);//kiểm tra xem hàng đợi có trống không
```

Do đó, các phần tử của hàng đợi tuân theo kỷ luật FIFO (First-In-First-Out: vào trước, ra trước) - ngược lại ngăn xếp. Và ở đây chúng ta có một phép tương tự với ý nghĩa của từ xếp hàng được sử dụng trong thực tế - ví dụ: trong hàng đợi mua vé (trừ khi chúng ta đủ đậm), chúng ta xếp hàng ở phía sau và chờ đến lượt của mình, và khi chúng ta ra phía trước, chúng ta được phục vụ và rời khỏi hàng đợi .

Có một số loại hàng đợi thú vị và quan trọng - ví dụ: hàng đợi ưu tiên (xem Bài tập 2.6.)



Hình 2.2. Hàng đợi

Ngăn xếp và đuôi có một số ứng dụng. Trong hầu hết các ngôn ngữ lập trình, phần bộ nhớ của lập trình viên được sử dụng ngầm cho ngăn xếp chương trình, việc tính toán các biểu thức được thực hiện dễ dàng thông qua ngăn xếp, v.v. Các lệnh chờ được thực thi

bởi bộ xử lý tạo thành một hàng đợi; có một hàng đợi cho các tài liệu được gửi đến máy in để in; các hệ thống xử lý đơn mô phỏng việc thực thi song song các tác vụ, sử dụng hàng đợi, v.v.

2.2.4. Hàng đợi hai đầu

Có một phiên bản khác, ít được sử dụng hơn, của danh sách, là bản tóm tắt của một ngăn xếp và một hàng đợi cùng một lúc - đây là một *hàng đợi hai đầu* (DEQue, viết tắt của Double-Ended-Queue). Như tên gọi của nó, trong cấu trúc này, các phần tử có thể được bật và tắt ở cả hai đầu. Về phần mình, hàng đợi hai đầu cũng có một số biến thể.

Có thể chỉ bật ở một bên và tắt - trên cả hai (hàng đợi hai đầu có lối vào hạn chế) hoặc ngược lại: bật ở cả hai bên và tắt - ở một (hàng đợi hai đầu có lối ra hạn chế). Nhìn chung, các bộ bài không được sử dụng rộng rãi, mặc dù trong một số trường hợp đặc biệt (ví dụ, trong các hệ thống đa xử lý) sẽ rất tiện lợi khi sử dụng cấu trúc cho các bài toán khác nhau (tính toán giá trị của biểu thức, v.v.) [Shishkov-1995].

2.3. Thực hiện cụ thể cấu trúc trên

2.3.1. Danh sách

Cách đơn giản nhất để thực hiện một danh sách là lưu trữ tuần tự các phần tử của nó trong bộ nhớ của máy tính (Hình 2.3). Trong trường hợp này, địa chỉ của phần tử thứ k, $addr(k)$ được xác định bằng công thức

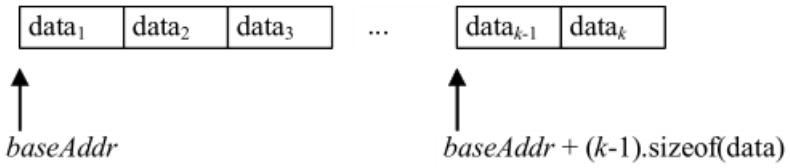
$$addr(k) = addr(k-1) + sizeof(dliu),$$

trong đó $sizeof(data)$ là bộ nhớ cần thiết để lưu một mục trong danh sách. Trong kiểu triển khai này, quyền truy cập vào phần tử thứ k là trực tiếp. Vì địa chỉ của phần tử ban đầu đã được biết (đây là địa chỉ cơ sở $baseAddr$), nên địa chỉ của phần tử thứ k có thể được xác định bằng công thức, bất kể địa chỉ của phần tử đứng trước:

$$addr(k) = baseAddr + (k-1)sizeof(data).$$

Tuy nhiên, sự thuận tiện của việc truy cập trực tiếp vào các phần tử được trả bằng cách bật và tắt không hiệu quả: để đưa vào một phần

tử sau chữ thứ k, chúng ta phải di chuyển tất cả các phần tử từ $k+1$ đến n theo một vị trí sang phải. Tương tự, để loại trừ phần tử thứ k, chúng ta phải chuyển tất cả các phần tử từ $k+1$ đến n theo một vị trí sang trái. Tìm kiếm khóa cũng chậm: trung bình $\frac{n}{2}$ so sánh được thực hiện (điều này cũng đúng với việc chèn và xóa), tức là độ phức tạp trung bình là $\Theta(n)$.



Hình 2.3. Danh sách chia bộ nhớ

2.3.2. Ngăn xếp

Chúng ta sẽ triển khai các cấu trúc ngăn xếp và hàng đợi một cách tĩnh thông qua một mảng. Trong trường hợp ngăn xếp, chúng ta sẽ nhập một chỉ mục duy nhất sẽ cho biết địa chỉ ở trên cùng của ngăn xếp. Do đó, trong hoạt động đưa vào, phần tử sẽ được ghi vào địa chỉ được chỉ định bởi chỉ mục, sau đó chỉ mục sẽ được tăng lên một. Theo đó, khi tắt, chỉ số đầu tiên sẽ giảm đi một, sau đó phần tử sẽ được trích xuất. Đây là cách triển khai C trông như thế nào khi sử dụng mảng stack[] của loại data và một biến top trỏ đến đầu ngăn xếp:

Khung ngăn xếp

```
#include <stdio.h>
typedef int data;
data stack[10];
int top;

void init (void) {  top = 0;  }
void push(data i)  {  stack[top++] = i;  }
data pop(void) { return stack[--top];  }
int empty(void) { return(top == 0); }
```

```
int main(void) {
  /* ... */
  return 0;
}
```

Những bất lợi của việc thực hiện ở trên là một số. Đầu tiên chúng ta khai báo một mảng có 10 phần tử. Nếu đã có 10 phần tử trong ngăn xếp (index top tương ứng bằng 10) và chúng ta muốn đưa vào một phần tử mới, sẽ xảy ra tràn và chương trình sẽ không đăng ký phần tử đó. Việc ghi lại trên bộ nhớ "ngoại lai" này có thể dẫn đến kết quả không mong đợi (và thường là thảm khốc). Tương tự, khi cố gắng loại trừ một mục khỏi một ngăn xếp trống, chỉ mục top sẽ trở thành một số âm và hàm pop() sẽ trả về một giá trị tùy ý (thực sự trả về giá trị của địa chỉ trong bộ nhớ ngay trước phần tử đầu tiên của mảng stack). Ngoài ra, một sự bất tiện liên tục được tạo ra bởi hằng số 10 được sử dụng trong chương trình. Ví dụ: nếu chúng ta quyết định thay đổi số phần tử tối đa thành 20, chúng ta sẽ phải xem lại toàn bộ mã chương trình, điều này một mặt đòi hỏi thời gian và mặt khác - là một hoạt động không an toàn, bởi vì chúng ta có thể dễ dàng bỏ lỡ một cái gì đó (ví dụ: , đến từ 10–1, v.v.).

Để giải quyết những thiếu sót này, chúng ta sẽ thực hiện các sửa đổi sau - chúng ta sẽ xác định macro MAX, đặt số lượng phần tử tối đa trong ngăn xếp. Khi một mục mới được đưa vào, nó sẽ được kiểm tra xem có phải là top nó đã không trở nên lớn hơn MAX và nếu vậy, thì có một phần tràn. Một thay đổi khác đã được thực hiện trong chương trình bên dưới - trước mỗi lần cố gắng tắt một mục, hãy kiểm tra xem ngăn xếp có trống không.

Ví dụ

Chương trình 2.1. Ngăn xếp (202stack2.c)

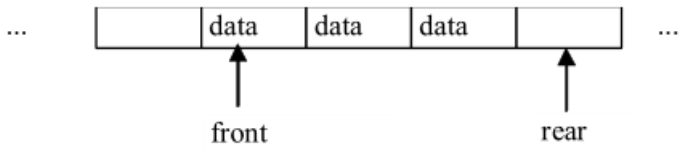
```
1 #include <stdio.h>
2 #define MAX 10
3 typedef int data;
4 data stack[MAX];
5 int top;

7 void init(void) { top = 0; }
```

```
9  void push(data i)
10 { if (MAX == top)
11     fprintf(stderr, "Them vao ngan xep \n");
12   else
13     stack[top++] = i;
14 }

16 data pop(void)
17 { if (0 == top) {
18     fprintf(stderr, "Ngan Xep troong \n");
19     return 0;
20   }
21   else
22     return stack[--top];
23 }

25 int empty(void) { return (0 == top); }

27 int main(void) {
28   data p;
29    init ();

31 //Đưa vào một số nguyên và đưa  vào ngăn xếp, số 0 kết thúc
32   scanf("%d", &p);
33   while (0 != p) {
34     push(p);
35     scanf("%d", &p);
36   };

38 //Lấy ra lần lượt các phần tử của ngăn xếp và in ra.
39 //như vậy lấy ra ngược với dãy đưa vào.
40   while (!empty()) printf("%d ", pop());
41   printf("\n");
42   return 0;
43 }
```

2.3.3. Hàng đợi

Việc thực hiện hàng đợi phức tạp hơn một chút. Để đặt trước bộ nhớ tuần tự, chúng ta sẽ lại sử dụng một mảng, nhưng các chỉ mục

sẽ là hai: front , cho biết phần đầu của hàng đợi và rear , cho biết vị trí sau khi kết thúc (Hình 2.4).

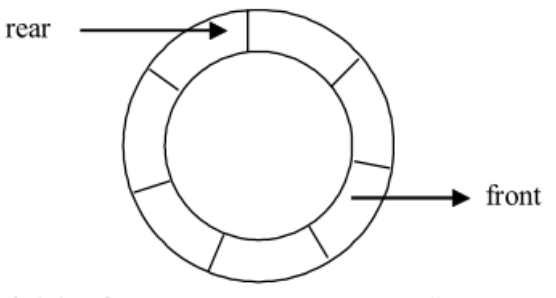


Hình 2.4. Danh sách chia bộ nhớ

Việc bật và tắt các phần tử được thực hiện như sau:

- *Đưa vào phần tử i*: Viết *i* vào vị trí được chỉ ra bởi chỉ số rear và tăng nó lên một.
- *Loại trừ một mục*: Trả mục đó về phía trước front , sau đó tăng chỉ mục front với một.

Có một số vấn đề cần giải quyết khi triển khai bật và tắt. Vấn đề chính liên quan đến thực tế là hàng đợi "di chuyển" trong bộ nhớ: Nếu chúng ta bật và tắt các mục liên tục, thì chỉ mục front và rear chúng sẽ tăng liên tục và nhanh chóng vượt quá bộ nhớ được cấp phát ban đầu cho hàng đợi. Vì vậy, nếu chúng ta sử dụng mảng Queue [MAX] (với một số phần tử cố định MAX), kích thước của nó sẽ nhanh chóng không đủ, mặc dù trong thực tế, hàng đợi sẽ chứa ít hơn MAX thành phần. Do đó, nếu một trong hai chỉ mục front hoặc rear trở thành bằng với MAX , nó sẽ được gán giá trị bằng 0. Do đó chúng ta đạt được "tính chu kỳ" của mảng queue[] (xem Hình 2.5).



Hình 2.5. Hàng đợi thể hiện quả mảng chu kỳ

Tại thời điểm bắt đầu, các chỉ số phía trước và phía sau trỏ đến phần tử không. Hơn nữa, nếu một lúc nào đó chúng trở lại bằng nhau (trỏ đến cùng một ô), điều này có nghĩa là một trong hai điều:

- Nếu bình đẳng thu được sau *không bao gồm* một phần tử (chỉ mục front Has to rear), Thì hàng đợi vẫn trống sau thao tác.
- Nếu đẳng thức nhận được sau *inclusive* của một phần tử (rear Đã đạt tới front), Thì hàng đợi đã đầy - nó chứa phần tử MAX và không thể thêm được nữa.

Đối với trạng thái của hàng đợi (cho dù nó đầy hay trống), chúng ta sẽ sử dụng một biến bổ sung rỗng, bằng 1 khi hàng đợi trống và bằng 0 - nếu không. Biến rỗng được khởi tạo bằng 1 (hàng đợi trống ở đầu) và nó nhận giá trị 0 ngay sau lần đưa vào đầu tiên một phần tử. Hơn nữa, biến này chỉ được sửa đổi trong hai trường hợp đặc biệt được liệt kê ở trên. Sự nhận biết đầy đủ của Si như sau:

Chương trình 2.2. Hàng đợi (203queue.c)

```
1  #include <stdio.h>
2  #define MAX 10

4  typedef int data;
5  data queue[MAX];
6  int front, rear, empty;

8  void init(void) { front = rear = 0; empty = 1; }

10 void put(data i)
11 { if (front == rear && !empty) {
12 //Kiểm tra đầy chưa*/
13 //Đầy rồi thì các chỉ số bằng nhau, hàng đợi khác rỗng
14   fprintf(stderr, "Hang doi rong! \n");
15     return;
16   }
17   queue[rear++] = i;
18   if (rear >= MAX) rear = 0;
19   empty = 0;
20 }

22 data get(void)
23 { data x;
```

```
24    if (empty) { // Kiểm tra hàng đợi rỗng
25      fprintf(stderr, "Hang doi rong! \n");
26      return 0;
27    }
28    x = queue[front++];
29    if (front >= MAX) front = 0;
30    if (front == rear) empty = 1;
31    return x;
32  }

34  int main(void) {
35    data p;
36    int i;
37     init();
38    for (i = 0; i < 2 * MAX; i++) {
39      put(i);
40      p = get();
41      printf("%d ", p);
42    }

44    printf("\n");

46  //Đưa phần tử tiếp theo vào
47    for (i = 0; i < MAX + 1; i++) put(i);

49  //Lấy các phần tử ra, khi có lỗi là hàng đợi rỗng
50    for (i = 0; i < MAX + 1; i++) get();
51    return 0;
52  }
```

Bài tập

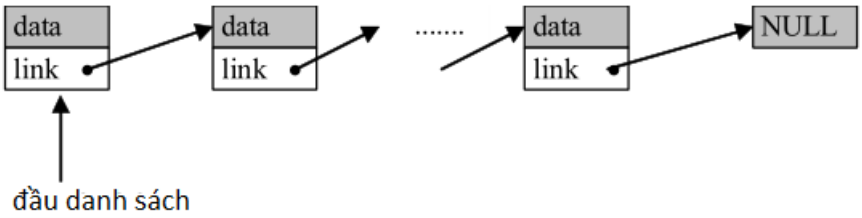
▷ **2.2.** Thiết kế lại các chương trình trên để các hàm không in thông báo lỗi, nhưng trả về giá trị Boolean nếu chúng thành công.

▷ **2.3.** Để thực hiện phiên bản "bán tĩnh" của các chương trình trên: nếu ngăn xếp / hàng đợi đã đầy, để tự động mở rộng mảng.

▷ **2.4.** Để thực hiện các chức năng chính để làm việc với Dec.

2.4. Thể hiện cấu trúc động

Với cấu trúc dữ liệu động, các phần tử không nhất thiết phải nằm trong các địa chỉ bộ nhớ liên tiếp. Trong trường hợp chung, chúng ta chỉ có một con trỏ đến đầu (phần tử đầu tiên) của danh sách và địa chỉ của tất cả các phần tử khác không thể truy cập trực tiếp. Sau đó chúng được truy cập như thế nào? Một trường khác được thêm vào mỗi phần tử (ngoài các trường chứa dữ liệu): một con trỏ đến phần tử tiếp theo (Hình 2.6).



Hình 2.6. Danh sách liên kết động

Loại danh sách tuyến tính này được gọi là danh sách liên kết đơn tuyến tính: liên kết đơn bởi vì chúng ta có một con trỏ chỉ đến phần tử tiếp theo.

Để đến phần tử thứ k của danh sách, chúng ta phải đi qua tất cả các phần tử đứng trước thứ $k-1$ một cách tuần tự. Cấu trúc của các phần tử như sau:

Cấu trúc động

```
typedef int data;
typedef int keyType;
struct list {
keyType key;
data info;
struct list *next;
};
```

Mỗi phần tử có một định danh: key (field keyType key). ìnfo chứa dữ liệu bổ sung cho mỗi phần tử. Con trỏ đến phần tử tiếp theo là một con trỏ đến cùng cấu trúc `struct list *next;` - Ngôn ngữ này tự

cho phép định nghĩa đệ quy cấu trúc như vậy (nhưng chỉ khi chúng ta sử dụng con trỏ tới một cấu trúc; khai báo tĩnh kiểu `struct` list next; Trong phần thân của list chưa đủ). Mục cuối cùng trong danh sách liên kết động có giá trị là NULL : gán một giá trị như vậy cho thư mục *next có nghĩa là không còn mục nào trong danh sách.

Có các cách triển khai khác có thể có của danh sách tuyến tính [Nakov-1998] [Wirth-1980]. Ví dụ: trong danh sách liên kết đôi, hai con trỏ được giữ cho mỗi phần tử: phần tử trước và phần tử tiếp theo. Nếu trong danh sách liên kết đơn có thêm một con trỏ từ phần tử cuối cùng đến phần tử đầu tiên, thì danh sách đó được gọi là tuần hoàn và các phần tử khác.

Để sử dụng danh sách liên kết, chúng ta khai báo:

```
struct list *L;
```

và khởi tạo L = NULL. Thao tác đầu tiên chúng ta sẽ xem xét là đưa một mục vào đầu (trước mục đầu tiên) của danh sách.

2.4.1. Đưa vào một phần tử

Ba hoạt động đơn giản được thực hiện liên tiếp (xem Hình 2.7):

1) Cấp phát bộ nhớ cho một phần tử mới `struct` list:

```
struct list *temp;
temp = (struct list *) malloc (sizeof (*temp));
```

2) Phần tử tạm thời mới cho biết phần đầu của danh sách:

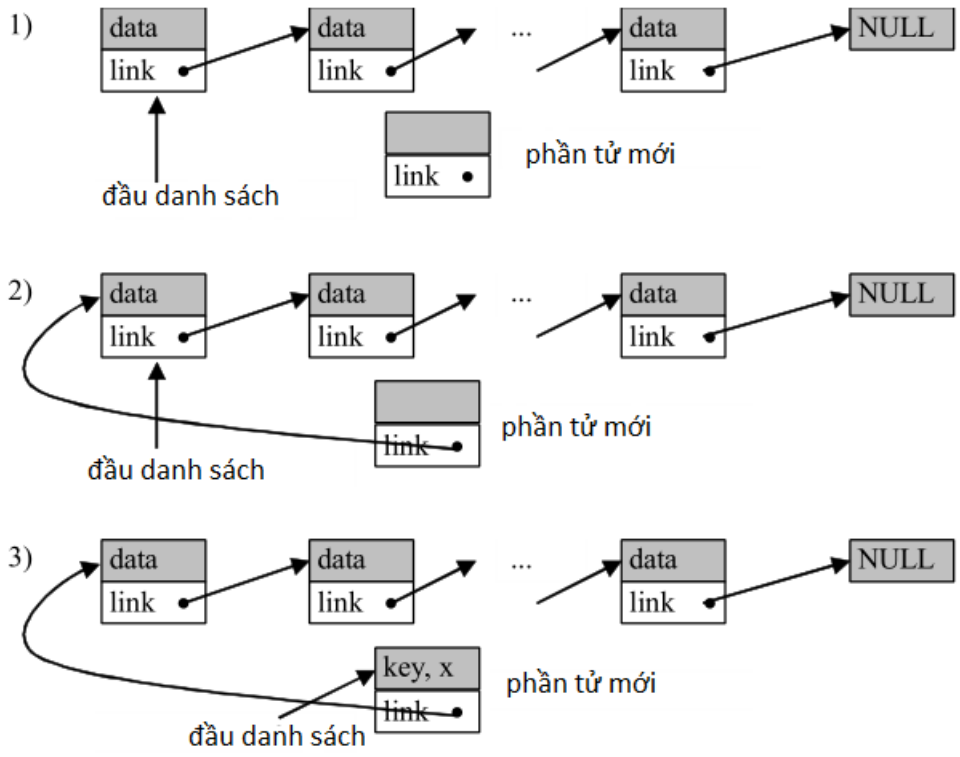
```
temp-> tiếp theo = L;
```

3) Phần đầu của danh sách được thiết lập trong phần tử mới và trường dữ liệu và khóa đã đặt được gán cho nó:

```
L = temp;
L-> key = key;
L-> info = x;
```

2.4.2. Cuộn qua một danh sách

Với thao tác này, có thể xây dựng một danh sách toàn bộ. Nó có thể được thu thập thông tin như sau: chúng ta bắt đầu từ chỉ mục đến đầu danh sách L và chúng ta sẽ in khóa của các phần tử khi thu thập thông tin:



Hình 2.7. Đưa vào đầu và dịch chuyển con trỏ

```
while (L != NULL) {
printf("%d ", L->key);
L = L->next; //chuyển sang mục tiếp theo
}
```

Thu thập thông tin tương tự được sử dụng khi tìm kiếm một phần tử bằng một khóa nhất định:

Tìm kiếm

```
struct list* search(struct list *L, keyType key)
{ while (L != NULL) {
if (L->key == key) return L;
L = L->next;
}
return NULL;
}
```

Đặc điểm khi được đưa vào đầu danh sách là thứ tự của các phần tử trong đó đối lập với thứ tự nhận của chúng. Để có thể làm việc đầy đủ với các danh sách, chúng ta sẽ cần thêm ba thao tác nữa, mỗi thao tác chúng ta sẽ xem xét riêng:

2.4.3. Đưa vào sau một phần tử được chỉ định bởi một chỉ dẫn đã cho

Chúng ta định nghĩa hàm bao gồm là
`void insertAfter (struct list ** L, keyType key, data x).`
Con trỏ kép L chỉ ra một mục trong danh sách (đặc biệt đây có thể là mục đầu tiên), sau đó là một mục có khóa key và trường dữ liệu verb!data! . Thuật toán chèn như sau:

1) Một phần tử trống được tạo key và giá trị x:

Chèn phần tử

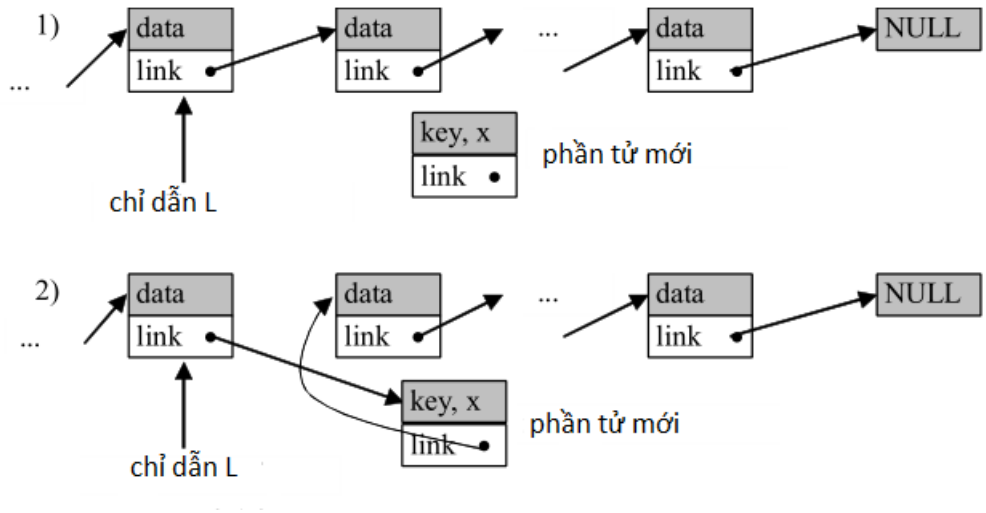
```
struct list *temp;
temp = (struct list *) malloc(sizeof(*temp));
temp->key = key;
temp->info = x;
```

2) Chuyển hướng con trỏ: Phần tử mới sẽ trỏ đến phần tử sau L và L sẽ trỏ đến phần tử mới (Hình 2.8):

```
temp->next = (*L)->next;
(*L)->next = temp;
```

2.4.4. Đưa vào phía trước một phần tử được chỉ định bởi một thư mục

Hàm kích hoạt là
`void insertBefore(struct list **L,keyType key,data x).`
Việc đưa vào trước một phần tử được chỉ ra bởi L là phức tạp bởi thực tế là chúng ta không có quyền truy cập trực tiếp vào phần tử trước đó (để thay đổi con trỏ của nó next). Thủ thuật sau được sử dụng để giải quyết vấn đề: phần tử mới được chèn vào sau L theo cách đã mô tả ở trên, sau đó các giá trị tương ứng của các trường



Hình 2.8. Đưa vào sau phần tử chỉ ra từ bảng chỉ dẫn

của nó được trao đổi với giá trị của L.

```
struct list  *temp;
temp = (struct list  *)  malloc(sizeof(*temp));
*temp = **L;
(*L)->next = temp;
(*L)->key = key;
(*L)->info = x;
```

2.4.5. Xóa theo khóa mặc định và con trỏ lên đầu danh sách

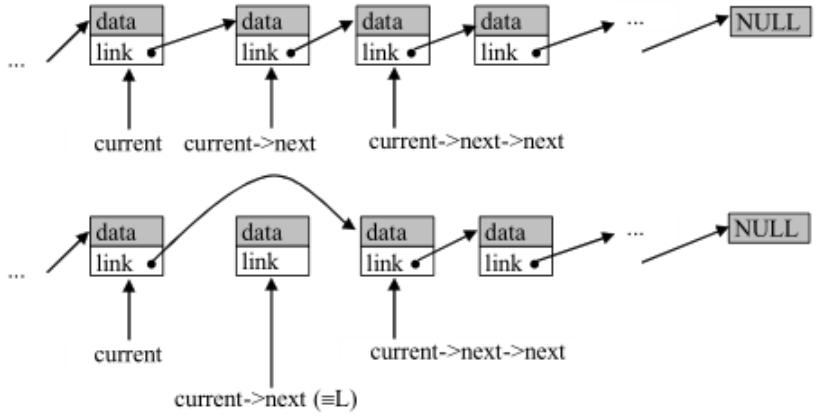
Chúng ta định nghĩa hàm xóa là
`void deleteNode(struct list **L, keyType key)`
Bước đầu tiên là tìm yếu tố chính. Việc tìm kiếm được thực hiện theo cách mà tại thời điểm chúng ta tìm thấy mục cần xóa, chúng ta cũng có một con trỏ đến mục tiền nhiệm của nó:

```
(1)  struct list  *current = *L;
       while (current->next != NULL && current->next->key != key)
       current  =  current->next;
```

Do đó, phần tử xóa được chỉ định bởi current-> next. Quá trình xóa

được thực hiện như sau: Phần tử tiền nhiệm của phần tử xóa được đặt để trỏ đến phần tử kế nhiệm của nó, tức là phần tử bị "bỏ qua", sau đó bộ nhớ mà nó chiếm được giải phóng - Hình 2.9:

```
(2) save = current->next;
    current->next = current->next->next;
    free(save);
```



Hình 2.9. Loại trừ khỏi danh sách.

Khi loại trừ một mục khỏi danh sách, cần lưu ý một điều nữa - khi tìm kiếm (1), nếu đến cuối danh sách, nghĩa là không có mục nào có khóa đã cho và thông báo lỗi được hiển thị.

2.4.6. Xóa một phần tử được chỉ định bởi một thư mục

Thao tác xóa phần tử được chỉ ra bởi con trỏ L cũng tương tự như thao tác xóa bằng khóa - cần tìm phần tử trước của phần tử được chỉ thị bởi L và con trỏ của nó tới phần tử tiếp theo để được chuyển hướng một cách thích hợp. Do đó, độ phức tạp của việc xóa theo khóa, cũng như theo con trỏ tới một phần tử, là tuyến tính. Để đạt được độ phức tạp không đổi khi xóa theo thư mục, cần sử dụng danh sách tuyến tính hai liên kết (xem bài toán 2.4.). Sau đây là cách triển khai động đầy đủ của một danh sách được liên kết:

Chương trình 2.3. Thao tác trên danh sách (204list.c)

```
#include <stdio.h>
#include <stdlib.h>
typedef int data;
typedef int keyType;

struct list {
keyType key;
data info;
struct list *next;
};

/* đưa vào một phần tử ở đầu danh sách được liên kết*/
void insertBegin(struct list **L, keyType key, data x)
{
  struct list *temp;
  temp = (struct list *) malloc(sizeof(*temp));
  if (NULL == temp) {
    fprintf(stderr, " Không đủ bộ nhớ cho phần tử mới!\n");
    return;
  }
  temp->next = *L;
  (*L) = temp;
  (*L)->key = key;
  (*L)->info = x;
}

/* Đưa vào sau phần tử */
void insertAfter(struct list **L, keyType key, data x)
{ struct list *temp;
  if (NULL == *L) { /*nếu danh sách rỗng => trường hợp đặc biệt*/
    insertBegin(L, key, x);
    return;
}

temp = (struct list *) malloc(sizeof(*temp));
/*Tạo ra phần tử mới */
if (NULL == temp) {
  fprintf(stderr, " Không đủ bộ nhớ cho phần tử mới!\n");
  return;
```

```
    }
    temp->key = key;
    temp->info = x;
    temp->next = (*L)->next;
    (*L)->next = temp;
}

/* Đưa vào trước một phần tử */
void insertBefore(struct list **L, keyType key, data x)
{ struct list *temp;
if (NULL == *L) { /*phần tử phải được chèn trước mục đầu tiên*/
   insertBegin(L, key, x);
   return;
}

temp = (struct list *) malloc(sizeof(*temp));
/* Tạo ra phần tử mới */
if (NULL == temp) {
   fprintf(stderr, "Không đủ bộ nhớ cho phần tử mới!\n");
   return;
}
   *temp = **L;
   (*L)->next = temp;
   (*L)->key = key;
   (*L)->info = x;
}

/*xóa một phần tử khỏi danh sách*/
void deleteNode(struct list **L, keyType key)
{ struct list *current = *L;
struct list *save;
if ((*L)->key == key) { /* mục đầu tiên phải được xóa */
current = (*L)->next;
free(*L);
(*L) = current;
return;
}
/*tìm phần tử cần xóa*/
while (current->next != NULL && current->next->key != key) {
current = current->next;
```

```
}
if (NULL == current->next) {
fprintf(stderr, "Lỗi: Không tìm thấy mục cần xóa! \n");
return;
}
else {
save = current->next;
current->next = current->next->next;
free(save);
}
}
/* in các phần tử của một danh sách được liên kết */
void print(struct list *L)
{ while (NULL != L) {
printf("%d(%d) ", L->key, L->info);
L = L->next;
}
printf("\n");
}
/* tìm kiếm một mục quan trọng trong danh sách được liên kết */
struct list* search(struct list *L, keyType key)
{ while (L != NULL) {
if (L->key == key) return L;
L = L->next;
}
return NULL;
}

int main() {
struct list *L = NULL;
int i, edata;
insertBegin(&L, 0, 42);
for (i = 1; i < 6; i++) {
edata = rand() % 100;
printf("Chèn trước: %d(%d)\n", i, edata);
insertBefore(&L, i, edata);
}

for (i = 6; i < 10; i++) {
edata = rand() % 100;
```

```
printf("Chèn sau: %d(%d)\n", i, edata);
insertAfter(&L, i, edata);
}
print(L);
deleteNode(&L, 9); print(L);
deleteNode(&L, 0); print(L);
deleteNode(&L, 3); print(L);
deleteNode(&L, 5); print(L);
deleteNode(&L, 5);
return 0;
}
```

Kết quả thực hiện chương trình:

```
Chèn trước: 1 (46)
Chèn trước: 2 (30)
Chèn trước: 3 (82)
Chèn trước: 4 (90)
Chèn trước: 5 (56)
Chèn sau: 6 (17)
Chèn sau: 7 (95)
Chèn sau: 8 (15)
Chèn sau: 9 (48)
5 (56) 9 (48) 8 (15) 7 (95) 6 (17) 4 (90) 3 (82) 2 (30) 1 (46) 0 (42)
5 (56) 8 (15) 7 (95) 6 (17) 4 (90) 3 (82) 2 (30) 1 (46) 0 (42)
5 (56) 8 (15) 7 (95) 6 (17) 4 (90) 3 (82) 2 (30) 1 (46)
5 (56) 8 (15) 7 (95) 6 (17) 4 (90) 2 (30) 1 (46)
8 (15) 7 (95) 6 (17) 4 (90) 2 (30) 1 (46)
Lỗi: Không tìm thấy phần tử xóa!
```

Bài tập

▷ **2.5.** Làm lại các chương trình trên để các hàm không in ra thông báo lỗi, nhưng trả về giá trị Boolean nếu chúng thành công.

▷ **2.6.** Cấu trúc dữ liệu ngăn xếp, hàng đợi, bộ bài, v.v. chúng có thể được coi là danh sách tuyến tính đặc biệt và cũng có thể được thực hiện động. Chúng ta cho phép người đọc, bằng cách sử dụng triển khai danh sách không liên kết tuyến tính ở trên, cố gắng sửa đổi danh sách này thành triển khai ngăn xếp và hàng đợi động. Điều

quan trọng là độ phức tạp của các hoạt động cơ bản (bật, tắt) vẫn không đổi.

▷ **2.7.** Để thực hiện một chiến lược kết hợp để trình bày một ngăn xếp trong bộ nhớ, sử dụng một danh sách các mảng. Ban đầu, nó bắt đầu với một mảng duy nhất và khi nó đầy, một khối mới được cấp phát, khối này được kết nối với một con trỏ tới khối trước đó. Khi mảng được đưa vào cuối cùng bị làm trống, nó sẽ bị loại trừ khỏi danh sách. Ưu điểm và nhược điểm của cách tiếp cận được đề xuất là gì? Bạn có giới thiệu nó cho một hàng đợi không? Và cho bộ bài?

2.5. Cây nhị phân

Việc sử dụng danh sách tuyến tính cực kỳ kém hiệu quả đối với các tác vụ trong đó hoạt động tìm kiếm được thực hiện rất thường xuyên so với các tác vụ khác liên quan đến việc thay đổi danh sách: bao gồm, xóa, sắp xếp lại, v.v.

Như chúng ta đã thấy, trong việc triển khai động của một danh sách tuyến tính, việc bao gồm và loại trừ một phần tử là các phép toán nhanh (độ phức tạp của chúng là không đổi - $\Theta(1)$), trái ngược với tìm kiếm khóa, có độ phức tạp là $\Theta(n)$.

Trong thực tế tĩnh, sự phức tạp của bao gồm là $\Theta(1)$, và loại trừ, cũng như nhu cầu - $\Theta(n)$. Có thể lưu trữ các mục trong danh sách, được sắp xếp theo khóa của chúng (chuyển đổi tĩnh). Điều này sẽ làm giảm hiệu quả của hoạt động bao gồm (các phần tử sẽ cần được sắp xếp lại), nhưng sẽ cho phép áp dụng tìm kiếm nhị phân cổ điển (được thảo luận trong Chương 4 - xem 4.3.) Và giảm độ phức tạp của tìm kiếm xuống $\Theta(log_2 n)$. được gọi là hàng đợi ưu tiên, xem bài toán 2.6.). Không thể áp dụng tìm kiếm nhị phân trong chuyển đổi động vì không có quyền truy cập trực tiếp vào phần tử thứ k trong danh sách.

Chúng ta sẽ xem xét cấu trúc dữ liệu *dạng cây* kết hợp lợi ích của chuyển đổi động với hiệu suất tìm kiếm tĩnh. Trong cấu trúc này, ba phép toán bật, tắt và chìa khóa trao tay trong trường hợp ở giữa có độ phức tạp theo lôgarit.

Định nghĩa 2.2. Tập hợp T là *một cây gốc* (sau đây chỉ gọi là cây) nếu

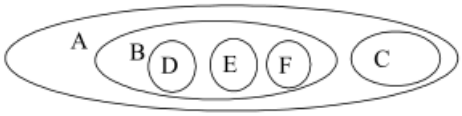
nó là một tập hợp rỗng hoặc nếu các điều kiện sau được thỏa mãn đồng thời:

1) Nó chứa một phần tử duy nhất t, được gọi là *gốc* của cây, mà chúng ta sẽ ký hiệu là $root(T)$.

2) Các phần tử khác (trừ gốc) được chia thành $m(m \geq 0)$ tập hợp rỗng không giao nhau $T_1, T_2, ..., T_m$, mỗi tập là *một cây*.

Các tập $T_1, T_2, ..., T_m$ được gọi là con của T. Các gốc T_1 của T_1, T_2 của $T_2, ..., t_m$ của T_m được gọi là *con trực tiếp* của t. Các đỉnh khác của $T_1, T_2, ..., T_m$ là người *thừa kế gián tiếp* của t.

Kết quả là, mỗi phần tử (sau đây, các phần tử của T sẽ được gọi là *ngọn cây*) là gốc của một cây con. Số người thừa kế trực tiếp của một đỉnh là mức độ của nó. Nếu bậc của đỉnh bằng 0, nó được gọi là *lá*.



Hình 2.10. Biểu diễn trực quan thông qua các tập hợp lồng nhau.

(A (B(D)(E)(F)) (C))

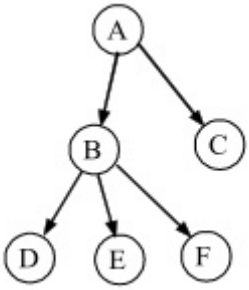
Hình 2.11. Trình bày trong ngoặc đơn.

Trong các Hình 2.10, 2.11, 2.12, 2.13 và **??** các cách trình bày trực quan khác nhau được thể hiện. Xét cây $T = \{A, B, C, D, E, F\}$ được thể hiện trong các hình sau: ở đây đỉnh A là gốc và $T_1 = \{C\}$ và $T_2 = \{B, D, E, F\}$ là các cây con của nó (đỉnh C là một lá). Trong cây $T_2 = \{B, D, E, F\}$ gốc là ngọn B, và các cây con là $\{D\}, \{E\}, \{F\}$ (D, E và F là lá).

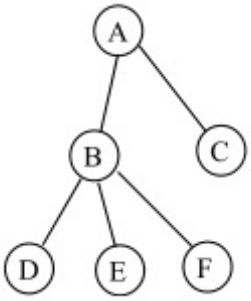
Định nghĩa 2.3. Đường đi trong cây là một dãy các đỉnh $t_1, t_2, ..., t_k$ không lặp lại và cứ hai đỉnh liên tiếp t_{i-1} và t_i $(2 \leq i \leq k)$ thì thỏa mãn đúng một trong hai đỉnh: t_i là sự kế thừa của t_{i-1}, hoặc t_{i-1} là sự kế vị của t_i. Số $k - 1$ được gọi là *độ dài* đường đi.

Mệnh đề 2.1. *Chỉ có một lối đi giữa hai ngọn cây.*

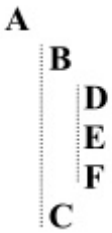
Định nghĩa 2.4. *Mức đỉnh* được gọi là độ dài của đường đi từ gốc đến ngọn. *Chiều cao* của cây được gọi là mức tối đa của đỉnh trong đó.



Hình 2.12. Biểu diễn bằng đồ thị liên thông xoay chiều có định hướng (xem Chương 5).



Hình 2.13. Biểu diễn bằng một đồ thị liên thông xoay chiều không có hướng (xem Chương 5), với một trong các đỉnh được chọn làm gốc: trong trường hợp này là đỉnh *A*.

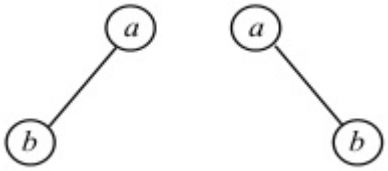


Hình 2.14. Biểu diễn bằng phép dời hình từng bước.

Định nghĩa 2.5. Nếu độ của mỗi ngọn cây nhỏ hơn hoặc bằng hai thì cây đó được gọi là *cây nhị phân*. Vì có nhiều nhất hai cây con trên mỗi đỉnh, nên một sắc lệnh của những người thừa kế (và của các cây con, tương ứng) thường được đưa ra nhiều nhất: *trái* và *phải.*

Cần lưu ý rằng, sau khi chúng ta đưa ra một quy định, về mặt cấu trúc, cây nhị phân khác về chất với cây và không phải là một tập con của chúng. Ví dụ, hai đối tượng trong Hình 2.15. chúng hoàn toàn không mô tả cùng một cây nhị phân, mặc dù là cây, chúng không thể phân biệt được và đại diện cho cây trong Hình 2.16.

Đến cuối đoạn này, chúng ta sẽ xem xét cây nhị phân. Thực hiện theo một số ví dụ:

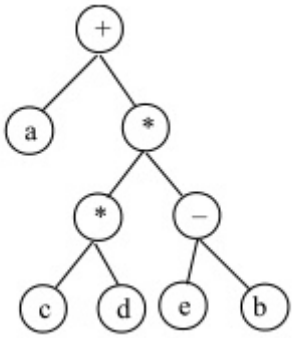


Hình 2.15. Cây nhị phân

Hình 2.16. Cây

- Giải đấu quần vợt, nơi nó được chơi với việc loại bỏ trực tiếp.
- Trình bày biểu thức số học với các giao dịch hai phần (xem Hình 2.17): Mỗi thao tác là một đỉnh với những người thừa kế cả hai toán hạng của nó.

Cây nhị phân được sử dụng rộng rãi trong trình biên dịch, cơ sở dữ liệu và nhiều hơn nữa.

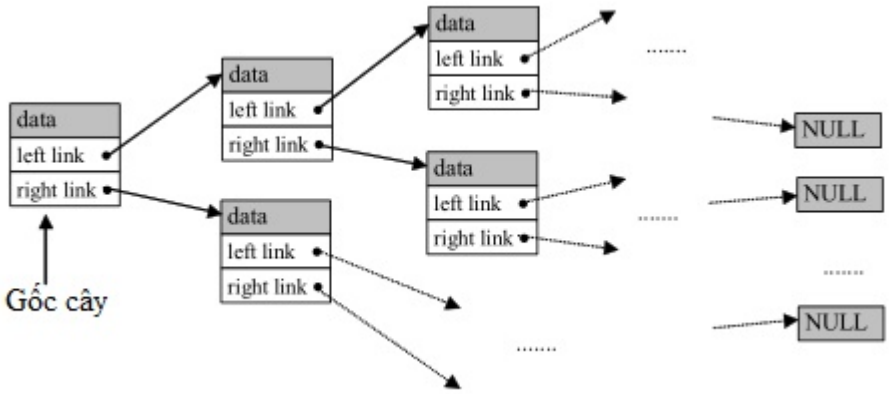


Hình 2.17. Trình bày biểu thức $a + ((c * d) * (e \breve{} b))$

Hãy là một cây nhị phân T, với các thông tin sau (Hình 2.18):

- Khóa khóa key(t) trên đầu trang.
- Trường data(t) với dữ liệu hàng đầu bổ sung.
- Hai thư mục left (t) và right(t): Ở bên trái và bên phải của T tương ứng của t

```
struct tree {
keyType key;
  data info;
  struct tree * left ;
  struct tree *right;
};
```



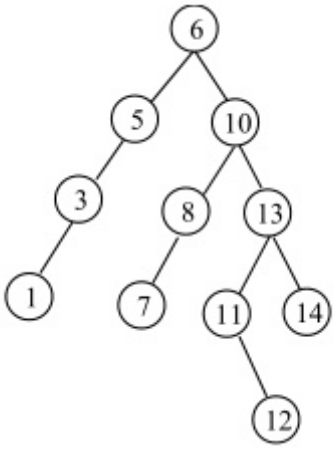
Hình 2.18. Trình bày động của một cây nhị phân trong bộ nhớ.

Chúng ta sẽ xác định ba thao tác cơ bản trên cây nhị phân:

- Thêm một đỉnh
  `void insertKey(keyType key, data x, struct tree **T);`
- Xóa một khóa chính
  `void deleteKey(keyType key, struct tree **T);`
- Tìm kiếm một đỉnh (và truy xuất dữ liệu từ nó) bằng khóa
  `struct tree *search(keyType key, struct tree *T);`

Nhiều loại cây nhị phân là *cây tìm kiếm nhị phân*. Với sự giúp đỡ của họ, chúng ta sẽ thực hiện các phép toán vừa liệt kê để độ phức tạp của chúng là logarit.

Định nghĩa 2.6. *Cây tìm kiếm nhị phân* (còn gọi là *cây nhị phân*) là một cây nhị phân cho mỗi đỉnh được thực thi: khóa của nó lớn hơn khóa của tất cả các phần tử trong cây con bên trái của nó và nhỏ hơn (hoặc bằng trong trường hợp cho phép trong cây. để thêm các phần tử có cùng khóa) từ các khóa của tất cả các phần tử ở bên phải.



Hình 2.19. Cây tìm kiếm nhị phân.

Hình 2.19 hiển thị một cây tìm kiếm nhị phân (các khóa phần tử là các số nguyên). Có thể thấy rằng khóa trên mỗi đỉnh có giá trị cao hơn các khóa trong cây con bên trái của nó và giá trị thấp hơn các khóa trong cây con bên phải của nó. Đặc biệt, điều kiện sau được đáp ứng cho những người thừa kế của mỗi đỉnh (điều này quan trọng trong việc thực hiện các hoạt động trong cây).

Tìm kiếm khóa trong cây nhị phân được thực hiện như sau:

2.5.1. Tìm kiếm bằng chìa khóa

Chúng ta bắt đầu tìm kiếm từ gốc, tức là t = root (T);

search(key, t):

1) Nếu key <key (t), ta tiếp tục tìm kiếm trong cây con bên trái của *t*, tức là ta thực hiện tìm kiếm đệ quy search(left (t), key);

2) Nếu key> key (t), ta tiếp tục tìm kiếm trong cây con bên phải của t, tức là ta thực hiện tìm kiếm đệ quy search(right(t), key);

3) Nếu key == key (t), chúng ta đã tìm thấy đỉnh cần thiết.

Trong thuật toán trên, chúng ta đã giả định rằng khóa chúng ta đang tìm kiếm nằm trong cây. Nếu bạn tìm kiếm một khóa không tồn tại, thì đến một bước nào đó, cây cần tìm sẽ trống (tức là con trỏ cha tương ứng cho biết NULL). Ví dụ, nếu trong cây của Hình 2.19 chúng ta đang tìm kiếm một phần tử có khóa 9, thuật toán sẽ thực

hiện liên tiếp các kiểm tra sau:
$9 > 6$ - chuyển đến gốc của cây con bên phải - trên cùng với khóa 10.
$9 < 10$ - chuyển đến gốc của cây con bên trái - trên cùng với khóa 8.
$9 > 8$ - chúng ta phải đi đến gốc của cây con bên phải, nhưng nó trỏ đến NULL, vì vậy phần tử được tìm kiếm bị thiếu.

2.5.2. Thêm vào một đỉnh mới

Việc đưa vào được thực hiện tương tự như tìm kiếm, mục tiêu ở đây là đến được một cây trống để có thể thêm một đỉnh mới vào đó.

```
insert (t, p):
```

1) Nếu `t == NULL` thì sau đó chúng ta đã tìm thấy nơi đưa đỉnh mới p và đưa nó vào: `t = p;`

2) Nếu `key(p) < key(t) insert (left (t), key);`

3) Nếu `key(p) > key t) insert (right (t), key);`

4) Nếu `key (p) == key (t)` thì phần tử cần chèn đã có trong cây. Trong trường hợp này, chúng ta có hai lựa chọn: chúng ta không thể làm gì, hoặc chúng ta có thể hiển thị thông báo lỗi (tức là chúng ta không cho phép các khóa trùng lặp).

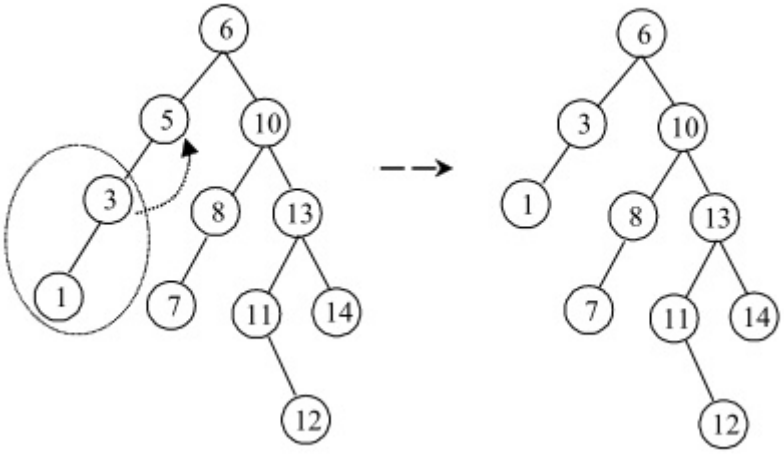
2.5.3. Xóa một đỉnh bằng từ khóa đã cho

Thao tác này phức tạp hơn một chút. Rõ ràng là trước khi xóa mẹo có khóa k khỏi cây, chúng ta cần tìm nó. Điều này được thực hiện theo thuật toán đã được mô tả để tìm một đỉnh. Sau đó, 3 tình huống có thể xảy ra:

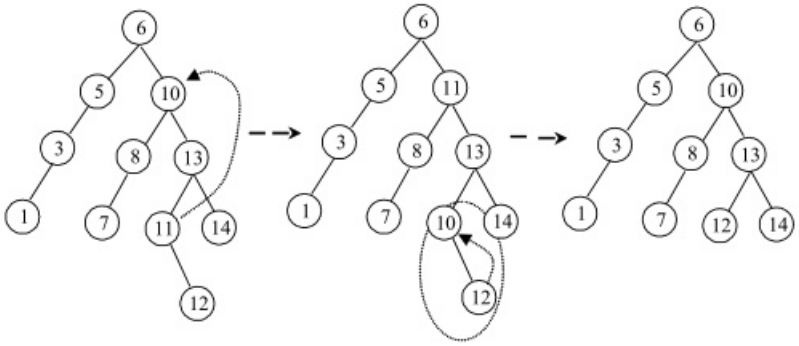
- Nếu đỉnh là một lá - vùng nhớ bị chiếm bởi nó được giải phóng và con trỏ của đỉnh trỏ đến nó bị thay đổi (nó được gán giá trị NULL).
- Nếu trên cùng chỉ có cây con bên trái hoặc chỉ cây con bên phải thì thay bằng gốc của cây con đó.
- Trường hợp phức tạp nhất là khi mũi xóa p có cả cây con trái và phải. Sau đó áp dụng cách sau: Tìm đỉnh có khóa nhỏ nhất trong cây con bên phải (ngoài cùng bên trái trong cây con bên phải) và đổi đỉnh này với p. Sau khi trao đổi p sẽ có nhiều nhất một cây con và bị loại trừ bởi một trong hai quy tắc trên. (Tất

nhiên, chúng ta có thể hoán đổi trái với phải và ngược lại: tìm phần tử có khóa lớn nhất trong cây con bên trái, hoán đổi nó với p, v.v.)

Hai trường hợp cuối cùng được minh họa trong Hình 2.20 và Hình 2.21.



Hình 2.20. Xóa kết nối đầu bằng từ khóa 5.



Hình 2.21. Xóa kết nối đầu bằng từ khóa 10.

Sau đây là cách triển khai đầy đủ các hoạt động được mô tả ở trên:

Chương trình 2.4. Thao tác trên cây nhị phân (205bintree.c)

```
#include <stdio.h>
#include <stdlib.h>

typedef char *data;
typedef int keyType;
struct tree {
  keyType key;
  data info;
  struct tree * left ;
  struct tree *right;
};

/*Tìm kiếm trong cây nhị phân*/
struct tree *search(keyType key, struct tree *T)
{ if (NULL == T)
    return NULL;
  else if (key < T->key)
    return search(key, T->left);
  else if (key > T->key)
    return search(key, T->right);
  else
    return T;
}

/*Đưa vào cây nhị phân*/
void insertKey(keyType key, data x, struct tree **T)
{ if (NULL == *T) {
  *T = (struct tree *) malloc(sizeof(**T));
  (*T)->key = key;
  (*T)->info = x;
  (*T)->left  = NULL;
  (*T)->right = NULL;
}
else if (key < (*T)->key)
  insertKey(key, x, &(*T)->left);
else if (key > (*T)->key)
  insertKey(key, x, &(*T)->right);
else
  fprintf(stderr, "Phần tử đã có trong cây!\n");
}
```

```
/*Loại trừ khỏi cây nhị phân*/
/* Tìm phần tử tối thiểu trong cây*/
struct tree *findMin(struct tree *T)
{ while (NULL != T->left) T = T->left;
  return T;
}
void deleteKey(keyType key, struct tree **T)
{ if (NULL == *T) {
  fprintf(stderr,"Đỉnh cần xóa không có!\n");
} else {
  if (key < (*T)->key)
    deleteKey(key, &(*T)->left);
  else if (key > (*T)->key)
    deleteKey(key, &(*T)->right);
  else /* phần tử loại trừ đã được tìm thấy*/
  if ((*T)->left && (*T)->right) {/*đỉnh có hai nhánh thừa kế*/
    /*có một đỉnh để trao đổi */
    struct tree *replace = findMin((*T)->right);
    (*T)->key = replace->key;
    (*T)->info = replace->info;
    deleteKey((*T)->key, &(*T)->right); /*Đỉnh loại trừ */
  }
  else { /* phần tử không có hoặc một cây con */
    struct tree *temp = *T;
    if ((*T)->left)
      *T = (*T)->left;
    else
      *T = (*T)->right;
    free(temp);
   }
  }
}

void printTree(struct tree *T)
{ if (NULL == T) return;
  printf("%d ", T->key);
  printTree(T->left);
  printTree(T->right);
}
```

```
int main() {
  struct tree *T = NULL, *result;
  int i;
  /*đưa vào đỉnh 10 với các khóa ngẫu nhiên*/
  for (i = 0; i < 10; i++) {
  int ikey = (rand() % 20) + 1;
  printf("Chèn phần tử bằng khóa %d \n", ikey);
  insertKey(ikey, "someinfo", &T);
}
  printf("Cây: ");
  printTree(T);
  printf("\n");
  /* tìm kiếm phần tử bằng khóa 5 */
  result = search(5, T);
  printf("đã tìm thấy: %s\n", result->info);

  /*xóa đỉnh 10 ngẫu nhiên khỏi cây */
for (i = 0; i < 10; i++) {
  int ikey = (rand() % 20) + 1;
  printf("Xóa phần tử bằng khóa %d \n", ikey);
  deleteKey(ikey, &T);
}
  printf("Cây: ");
  printTree(T);
  printf("\n");
  return 0;
}
```

Kết quả thực hiện chương trình:

```
Chèn phần tử bằng khóa 1
Chèn phần tử bằng khóa 1
Phần tử đã có trong cây!
Một phần tử có khóa 7 được chèn
Chèn phần tử bằng khóa 1
Phần tử đã có trong cây!
Một phần tử có khóa 8 được chèn
Chèn phần tử bằng khóa 5
Một phần tử có khóa 11 được chèn
Chèn phần tử bằng khóa 4
```

```
Chèn phần tử bằng khóa 15
Chèn phần tử bằng khóa 19
Cây: 1 7 5 4 8 11 15 19
Tìm thấy: someinfo
Xóa phần tử bằng khóa 6
Đỉnh cần xóa không có!
Xóa phần tử bằng khóa 9
Đỉnh cần xóa không có!
Xóa phần tử bằng khóa 3
Đỉnh cần xóa không có!
Xóa phần tử bằng khóa 14
Đỉnh cần xóa không có!
Xóa phần tử bằng khóa 12
Đỉnh cần xóa không có!
Xóa phần tử bằng khóa 1
Xóa phần tử bằng khóa 4
Xóa phần tử bằng khóa 17
Đỉnh cần xóa không có!
Xóa phần tử bằng khóa 14
Đỉnh cần xóa không có!
Xóa phần tử bằng khóa 16
Đỉnh cần xóa không có!
Cây: 7 5 8 11 15 19
```

2.5.4. Thu thập thông tin

Có thể thấy rằng trong chương trình trên chúng ta cũng đã sử dụng một hàm để in ra tất cả các đỉnh trên cây nhị phân printTree (`struct` tree * T). Hàm là đệ quy và dựa trên thuật toán sau: Phần gốc của cây được in, sau đó (đệ quy!) phần bên trái được in, và sau đó là cây con bên phải của T.

Không đưa ra định nghĩa chính thức (sẽ có một định nghĩa khi xem xét các cột trong Chương 5), dưới bò một cái cây, chúng ta sẽ hiểu việc kiểm tra tuần tự từng ngọn của nó, bắt đầu từ gốc.

Chức năng trên để in cây thực tế là rùa bò. Trong đó, gốc được kiểm tra đầu tiên, và chỉ sau đó đến ngọn của các cây con bên trái và bên phải. Thu thập thông tin này được gọi là *thu thập thông tin gốc-*

trái-phải hoặc preorder:CLD. Rõ ràng, sử dụng cùng một nguyên tắc, cây có thể được thu thập thông tin theo hai cách khác nhau đáng kể:

- LCD (Inorder): Được xem theo thứ tự: cây con bên trái, cây con gốc và cây con bên phải.
- LDC (Postorder): Các ngọn của cây con bên trái và bên phải được kiểm tra đầu tiên, và chỉ sau đó mới kiểm tra gốc.

Mỗi loại trong ba loại thu thập thông tin được sử dụng trong các bài toán và cách diễn giải khác nhau của cây nhị phân. Ví dụ, nếu chúng ta áp dụng một inorder cho cây tìm kiếm nhị phân và in khóa trên mỗi đỉnh mà chúng ta truy cập, chúng ta sẽ nhận được các khóa được sắp xếp theo thứ tự tăng dần.

Hãy xem xét cây nhị phân trong Hình 2.18. Hãy đi xung quanh nó và xem điều gì sẽ xảy ra:

- CLD: + a ** cd - eb
- LCD: a + c * d * e - b
- LDC: acd * eb - * +

Mỗi loại trong ba kiểu bò đều có kiểu "gương" riêng - chúng ta trao đổi thứ tự bò trên cây con bên trái và bên phải:

- CDL: + * - be * dca
- DCL: b - e * d * c + a
- DLC: be - dc ** a +

Như vậy, tổng số các kiểu duyệt sẽ trở thành 6: nhiều như hoán vị của các chữ cái L (bên trái), D (bên phải) và C (gốc). Có thể thấy rằng ký hiệu đại số tiêu chuẩn của biểu thức có được bằng cách thu thập dữ liệu kiểu LCD (tuy nhiên, không giữ lại các ưu tiên trong tính toán và để được sử dụng trong thực tế, phải được sửa đổi theo cách thích hợp - làm thế nào?), và trong CLD và LDC thu được bản ghi trường trực tiếp và đảo ngược tương ứng. Ký hiệu trường thẳng và đảo ngược rất hữu ích vì chúng có thể dễ dàng được sử dụng để tính toán một giá trị số mà không cần sử dụng dấu ngoặc đơn để thay đổi mức độ ưu tiên của các phép toán (khi so sánh giá trị của các biến trong một biểu thức, trong trường hợp này là a, c, d, e và f) [Wirth-1980] [Nakov-1998].

2.5.5. Bài tập

▷ **2.8.** Xây dựng cây tìm kiếm nhị phân có thứ tự bằng cách thêm tuần tự các đỉnh sau:
a) 7, 14, 28, 35, 65, 12, 18, 42, 81, 64, 61, 4, 13
b) 12, 7, 14, 81, 42, 18, 61, 4, 64, 35, 13, 28, 65
c) 4, 7, 12, 13, 14, 18, 28, 35, 42, 61, 64, 65, 81
e) 81, 65, 64, 61, 42, 35, 28, 18, 14, 13, 12, 7, 4
e) 28, 64, 13, 42, 7, 81, 61, 4, 12, 65, 35, 18, 14
So sánh các cây kết quả. Có thể rút ra kết luận gì?

▷ **2.9.** 2. Theo thứ tự này, xóa các ngọn $8, 13, 5$ và 6 của cây trong Hình2.19.

▷ **2.10.** Biểu thức $(A + B - C) * (D/F) - G * H$ đã cho. Tìm bản ghi tiếng Ba Lan ngược của anh ấy.

▷ **2.11.** Chứng minh rằng chỉ có một đường đi giữa hai đỉnh của cây.

▷ **2.12.** Phần nhỏ nhất của "cây" trong Hình 2.17 là gì. nên được cắt bớt để ví dụ đáp ứng Định nghĩa 2.2.

▷ **2.13.** Viết phiên bản lặp lại của hàm thu thập thông tin cây nhị phân.

▷ **2.14.** Viết một biến thể của hàm để duyệt cây nhị phân, hàm này sẽ in ra để có thể nhìn thấy cấu trúc của nó.

▷ **2.15.** Để chứng minh tính đúng đắn của thuật toán đã mô tả để xóa một đỉnh khỏi cây nhị phân có thứ tự.

▷ **2.16.** Thuật toán được mô tả ở trên để xóa một đỉnh khỏi cây nhị phân có thứ tự trong trường hợp hai người thừa kế đang tìm kiếm người thừa kế ngoài cùng bên trái của cây con bên phải. Triển khai một tùy chọn tìm kiếm cây kế thừa ngoài cùng bên phải cho cây con bên trái. Có khả năng nào khác không?

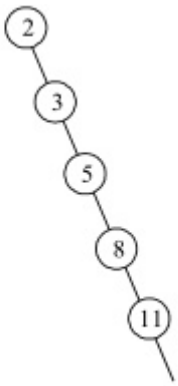
▷ **2.17.** Đề xuất và thực hiện một biểu diễn tĩnh của một cây nhị phân.

▷ **2.18.** Ở trên chúng ta đã thấy rằng cách chuẩn để biểu diễn một biểu thức trong cây nhị phân là viết các biến và hằng số trong danh sách và các phép toán trong các đỉnh. Lưu ý rằng chúng ta chỉ cho phép các phép toán hai đối số (chẳng hạn như `+`, `-`, `*`, `/`), với người thừa kế bên trái chứa đối số đầu tiên và người thừa kế bên phải là đối số thứ hai. Có thể cho phép các phép toán một ngôi (đối số đơn) như: `+` và `-` không? Điều gì đang thay đổi?

2.6. Cây cân đối

Phân tích kỹ hơn các phép toán với cây có thứ tự nhị phân cho thấy rằng có những trường hợp cây kết quả phân nhánh rất yếu và giống như một danh sách trong cấu trúc, và do đó về hiệu quả. Hãy để chúng ta, bắt đầu từ một cây trống, thực hiện một chuỗi gồm n phần tử. Kết quả thu được có thể cực kỳ "khó chịu" - ví dụ, nếu số lượng khóa của các phần tử được bao gồm tăng lên nghiêm ngặt (Hình **??**):

$$2, 3, 5, 8, 11, ...$$



Hình 2.22. Trường hợp xấu nhất với cây nhị phân.

Cây kết quả "suy biến" thành một danh sách liên kết, tìm kiếm mà, như chúng ta biết, có độ phức tạp tuyến tính. Thông thường trong thực tế, chúng ta cần tìm kiếm nhanh, bất kể loại đầu vào là gì, và trong những trường hợp như vậy, độ phức tạp tuyến tính

là một giải pháp không thể chấp nhận được trong trường hợp xấu nhất.

Chúng ta sẽ xem xét một cấu trúc dữ liệu "giữ" cây ở trạng thái trong đó các phép toán cơ bản (bật, tắt, tìm kiếm) có độ phức tạp logarit ngay cả trong trường hợp xấu nhất.

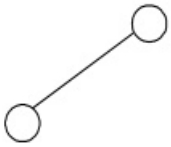
Định nghĩa 2.7. Cây nhị phân được gọi là *cân bằng* nếu sự khác biệt về chiều cao của các cây con bên trái và bên phải ở mỗi đỉnh nhiều nhất là một.

Định nghĩa 2.8. Một cây nhị phân có n đỉnh được gọi là *cân bằng hoàn toàn* nếu sự khác biệt về số lượng đỉnh của cây con bên trái và bên phải của mỗi đỉnh nhiều nhất là một.

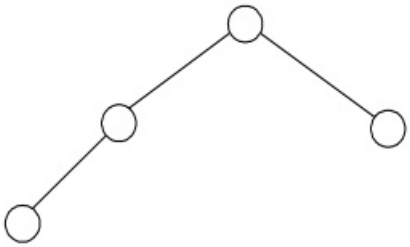
Định nghĩa 2.9. Định nghĩa 2.9. *Cây Fibonacci T_k có thứ tự k* được gọi là cây nhị phân trong đó:

- T_0 là cây rỗng - có chiều cao 0;
- T_1 là cây chứa một nút đơn - có chiều cao 1;
- Với $k \geq 2$, cây bao gồm một gốc, một cây Fibonacci T_{k-1} của hàng $k-1$ (cây con bên trái) và một cây Fibonacci T_{k-2} của hàng $k-2$ (cây con bên phải).

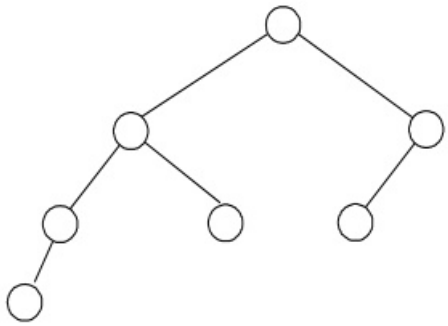
Hình 2.23. Cây Fibonacci của hàng 0.



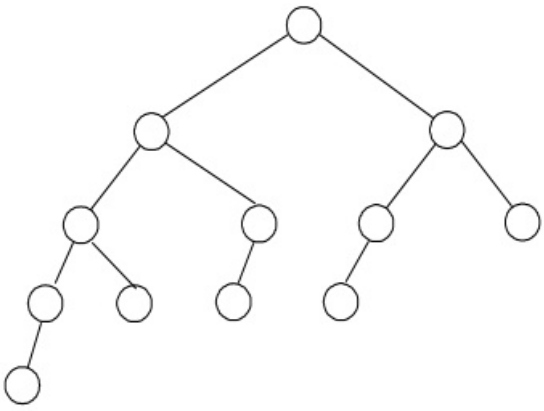
Hình 2.24. Cây Fibonacci của hàng 1.



Hình 2.25. Cây Fibonacci của hàng 2.



Hình 2.26. Cây Fibonacci của hàng 3.



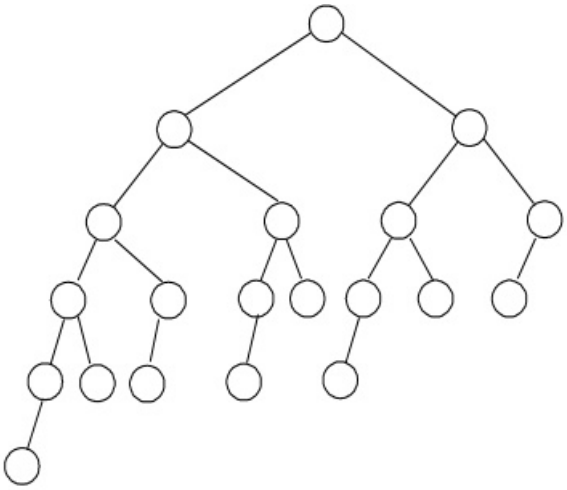
Hình 2.27. Cây Fibonacci của hàng 4.

Một vài cây Fibonacci đầu tiên được thể hiện trong Hình 2.23, 2.24, 2.25, 2.26, 2.27. và 2.28. Dễ dàng nhận thấy rằng cây Fibonacci là trường hợp xấu nhất mà chúng ta có thể mắc phải đối với cây cân bằng nhị phân. Thực tế này cũng được sử dụng trong định lý sau:

Định lý 2.1 (Adelson, Welsh và Landis)**.** *Cho là một cây nhị phân T với n đỉnh bên trong. Gọi h là chiều cao của T. Khi đó*

$$\log_2(n+1) < h < h1,4404\log_2(n+2) - 0,3277.$$

Kết quả cuối cùng thu được như thế nào? Rõ ràng, cây nhị phân có chiều cao h không thể có nhiều hơn 2^h đỉnh bên trong, tức là $n+1 \leq 2^h$, hoặc $h \geq \lceil \log_2(n+1) \rceil$.



Hình 2.28. Cây Fibonacci của hàng 5.

Để giới hạn h từ phía trên, chúng ta hãy ký hiệu T_h là một cây cân bằng, với tính chất sau: T_h có chiều cao h và có số đỉnh tối thiểu. Vì T_h là cân bằng, (không giới hạn cộng đồng) cây con bên trái của T_h là $h-1$ và cây con bên phải là $h-1$ hoặc $h-2$. Vì T_h là cây có số đỉnh tối thiểu nên cây con bên trái của gốc của T_h sẽ là T_{h-1} và bên phải: T_{h-2}. Trong sự tiếp tục quy nạp của việc xây dựng, chúng ta sẽ nhận được rằng T_h sẽ là cây Fibonacci bậc $h+1$ (Tại sao?). Do đó $n \geq F_h + 2$, và từ thuộc tính [Knuth-3/1968]:

$$F_{h+2} - 1 > \varphi^{h+2}/\sqrt{5} - 2.$$

phần khác của các bất đẳng thức thu được.

Bây giờ trở lại cây nhị phân để tìm kiếm. Khi liên tiếp thêm các phần tử mới, về cơ bản có hai cách tiếp cận để giữ cây tìm kiếm theo cách "cân bằng":

- tái cấu trúc cây nhị phân trong quá trình xây dựng.
- giảm cấp độ bằng cách sử dụng các cây có bậc cao hơn - hàng thứ k.

Hai cách tiếp cận này, cũng như một số giống cây cân đối cụ thể, sẽ được thảo luận trong hai đoạn tiếp theo.

Bài tập

▷ **2.19.** Xây dựng cây tìm kiếm nhị phân cân bằng hoàn hảo cho tập hợp sau các đỉnh $\{4, 7, 12, 13, 14, 18, 28, 35, 42, 61, 64, 65, 81\}$.

▷ **2.20.** Cây có cân đối không từ:
a) Hình 2.13.
b) Hình 2.18.
c) Hình 2.19.

▷ **2.21.** Cây có cân đối hoàn hảo từ:
a) Hình 2.13.
b) Hình 2.18.
c) Hình 2.19.

2.6.1. Vòng xoay. Cây đỏ và đen

Để duy trì chiều cao logarit của cây tìm kiếm nhị phân, các phép tái cấu trúc đặc biệt (phép quay) được xác định, được áp dụng bất cứ khi nào xảy ra "sự mất cân bằng". Đây là xoay trái và phải, xoay đôi và những thứ khác. (Hình 2.29).

Các thuộc tính quan trọng của các phép quay được hiển thị là:

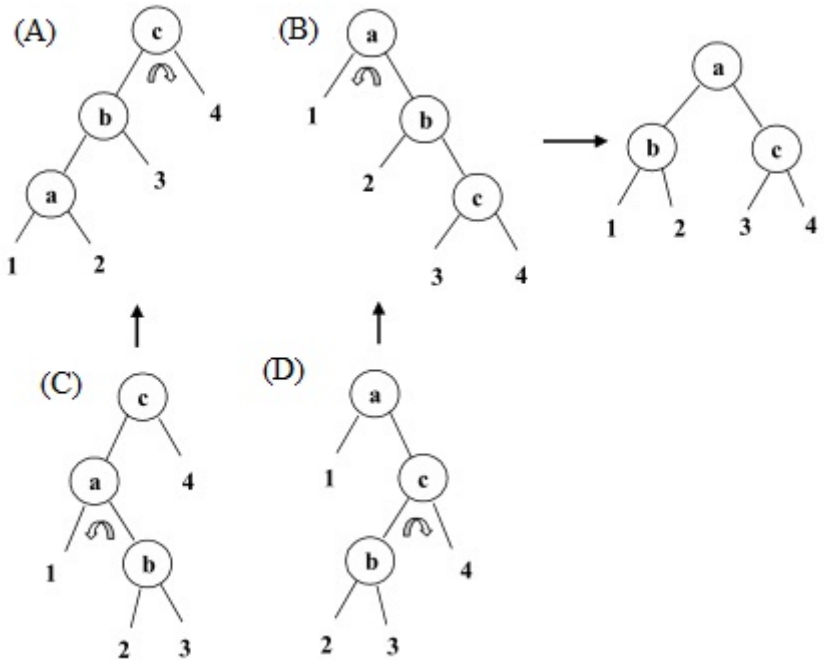
- việc thực hiện chúng vừa đơn giản vừa hiệu quả;
- bảo toàn kết quả thu thập thông tin cây kiểu LCD;
- đảm bảo cân bằng (nhưng không phải là cân bằng lý tưởng) của cây.

Việc lựa chọn phép quay nào để thực hiện trên các đỉnh khác nhau phải được làm rõ thêm liên quan đến các thuật toán cụ thể. Như một ví dụ, chúng ta sẽ xem xét một cấu trúc dữ liệu thú vị và phổ biến - cái gọi là Cây đỏ-đen.

Định nghĩa 2.10. *Cây đỏ-đen* được gọi là cây tìm kiếm nhị phân, trong đó mỗi đỉnh được đánh dấu đỏ hoặc đen và các đặc tính sau được bổ sung:

- Tất cả các lá (đỉnh NULL, chúng không thực sự có mặt) đều có màu đen.
- Nếu một đỉnh có màu đỏ thì hai đỉnh kế tiếp của nó có màu đen.

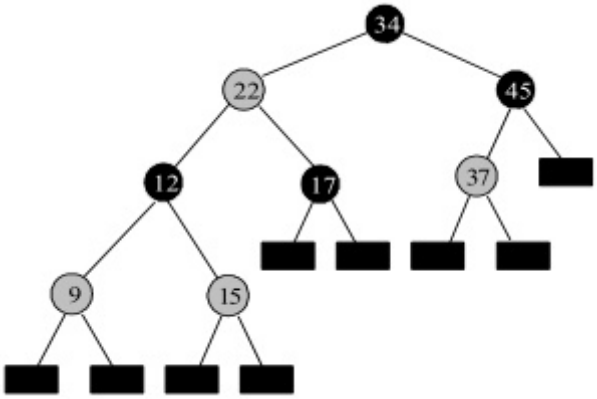
- Tất cả các đường đi từ đỉnh t bất kỳ đến bất kỳ lá nào từ cây con có gốc t đều chứa cùng một số đỉnh đen.



Hình 2.29. Xoay phải (A), trái (B), xoay trái-phải (C) và xoay phải-trái (D) quay.

Hệ quả trực tiếp của Định nghĩa 2.10 là tính chất hữu ích sau: chiều cao của cây đỏ đen với n đỉnh nhiều nhất là $2.log_2(n+1)$, tức là cây đỏ đen là cây tìm kiếm nhị phân xấp xỉ cân bằng. Tuy nhiên, chúng không cân bằng theo định nghĩa của chúng ta. Nhớ lại rằng một cây là cân bằng, mà sự khác biệt về chiều cao của cây con bên trái và bên phải của mỗi đỉnh cao nhất là 1. Trong màu đỏ-đen, sự khác biệt này nhiều nhất là hai lần. Đối với cây đỏ đen, có các thuật toán tương đối đơn giản cho các phép toán cơ bản (bật, tìm kiếm, tắt) với độ phức tạp $\Theta(\log_2 n)$, đó là lý do tại sao chúng được sử dụng rộng rãi nhất trong thực tế cây cân bằng [Shishkov-1995] [Cormen , Leiserson, Rivest-1997].

Có những loại cây nhị phân tìm kiếm cân bằng cổ điển khác, chẳng hạn như cây AVL (được đặt theo tên của Adelson-Velskii và



Hình 2.30. Ví dụ về cây đỏ-đen.

Landis), cũng đảm bảo độ phức tạp logarit cho các phép toán trên [Wirt-1980] [Knuth-3/1968]. Tuy nhiên, các thuật toán làm việc với cây AVL phức tạp hơn và hiệu quả của chúng trong thực tế thường kém hơn so với màu đỏ và đen, đó là lý do tại sao chúng ít được sử dụng hơn.

Bài tập

▷ **2.22.** Sử dụng Định nghĩa 2.10, Để chứng minh rằng mỗi cây rụng lá bằng cây đỏ - đen có đúng hai người thừa kế.

▷ **2.23.** Để xem xét các thao tác chính để làm việc với cây đỏ và đen có thể được thực hiện như thế nào trong mô tả đã cho về cấu trúc của chúng.

2.6.2. B-Cây

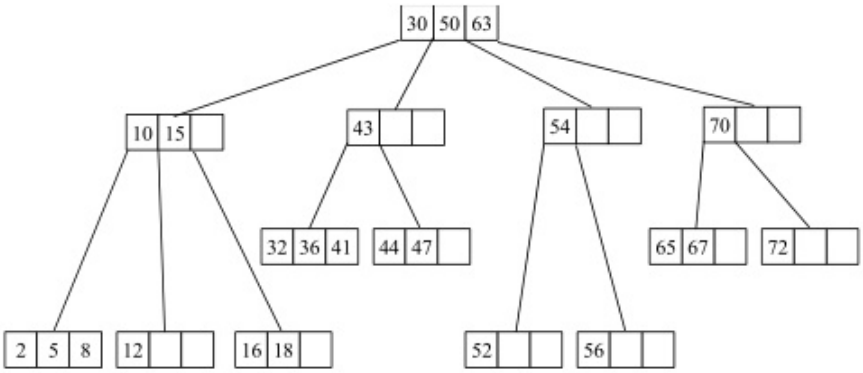
Ý tưởng xây dựng cây tìm kiếm (trong tài liệu cho đến nay chúng ta chỉ coi là cây nhị phân) có thể được mở rộng cho các cây ở mức độ cao hơn.

Định nghĩa 2.11. Chúng ta gọi *một đỉnh k* trong cây là một đỉnh chứa thông tin về $k-1$ phần tử (ở đây một phần tử có nghĩa là một khóa + dữ liệu bổ sung) với các khóa $t_1 < t_2 < ... < t_{k-1}$ và với k kế. - Cây con $T_1, T_2, ..., T_k$, trong đó đối với mỗi phím $t_i, i = 1, 2, ..., k-1$,

điều kiện sau được đáp ứng: tất cả các phím trong cây con T_i đều nhỏ hơn hơn t_i và tất cả các phím trong cây con T_{i+1} đều lớn hơn ti.

Định nghĩa 2.12. Cây 2-3-4 được gọi là cây T với các đặc tính sau:

- T là cây hoàn toàn cân đối.
- Mỗi khối chóp bên trong (không phải là chiếc lá) đều là khối chóp 2, hoặc khối 3, hoặc khối 4.
- Tất cả các đường đi từ gốc đến bất kỳ lá nào đều có cùng chiều dài.



Hình 2.31. 2-3-4 cây.

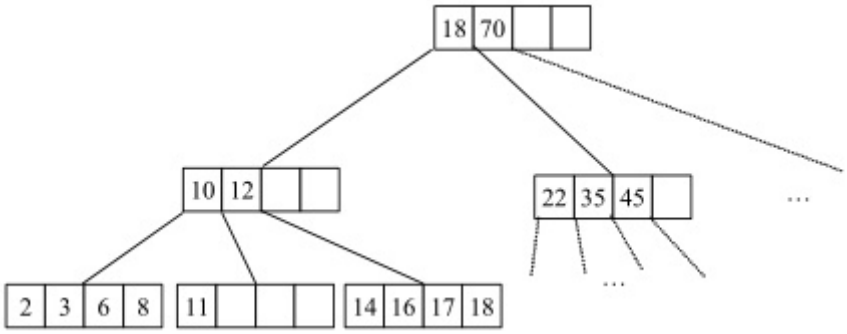
Có các thuật toán hiệu quả để làm việc với 2-3-4 cây, đảm bảo độ phức tạp của các phép toán cơ bản (thêm, tìm, xóa) bậc $\Theta(log_2 n)$. Cũng cần lưu ý rằng có một cách để chuyển đổi cây 2-3-4 thành cây nhị phân tương đương (cụ thể là cây đỏ-đen đã thảo luận ở trên), cho phép chúng được trình bày đơn giản hơn trong bộ nhớ và tồn tại các thuật toán đơn giản hơn. , đó là lý do tại sao trong thực tế, cây đỏ-đen thường được sử dụng thay vì cây 2-3-4 cây [Flamig-1993].

Chúng ta sẽ xem xét một bản tóm tắt quan trọng của 2-3-4 cây.

Định nghĩa 2.13. B-cây của hàng m được gọi là cây tìm kiếm với các thuộc tính sau:

- Mỗi đỉnh, trừ gốc và lá, là đỉnh k, với k là một số trong khoảng từ $[m/2]$ đến m.
- Tất cả các đường đi từ gốc đến bất kỳ lá nào đều có cùng độ dài.

Theo hệ quả của Định nghĩa 2.13. Cây B trở thành cây tìm kiếm hoàn toàn cân bằng. Đặc biệt, tại $m = 4$, chúng ta nhận được 2-3-4-cây.



Hình 2.32. B-cây hàng 5.

Chúng ta sẽ xem xét một số thuộc tính và ứng dụng quan trọng hơn của B-cây. Đối với độc giả tò mò, có rất nhiều nguồn với thông tin bổ sung về các thuật toán và việc thực hiện các hoạt động cơ bản trên B-wood [Wirt-1980] [Shishkov-1995] [Knuth-3/1968].

B-cây được sử dụng rộng rãi trong thực tế, đặc biệt là trong các hệ thống quản lý thông tin khối lượng lớn (ví dụ: cơ sở dữ liệu quan hệ). Được biết, việc truy cập bộ nhớ ngoài trong hệ thống máy tính thường được thực hiện theo khối. Giả sử rằng trên một số mô hình đĩa cứng, một khu vực vật lý chứa chính xác 4096 byte. Sau đó, việc đọc và viết sẽ diễn ra trong các khối có kích thước này. Tuy nhiên, truy cập vào các đơn vị đĩa cứng chậm hơn đáng kể so với truy cập vào RAM, do đó, để tối ưu hóa hệ thống, cần giảm thiểu số lượng các cung được chuyển giữa RAM và đĩa cứng. Giả sử chúng ta phải tổ chức một lượng lớn thông tin không vừa trong RAM, và cần phải tìm kiếm một phần tử của nó bằng một khóa cho trước. Chức năng tìm kiếm thực hiện những việc sau: nó đọc một khối và tìm khóa nó cần trong đó. Nếu anh ta tìm thấy nó - anh ta hoàn thành công việc, nếu không anh ta đọc khối khác và tìm kiếm lại. Quá trình kết thúc nếu mục được tìm thấy hoặc không tồn tại. Do đó, các khối được đọc phải mang thông tin đến mức cần đọc một số khối bổ sung tối thiểu để xác định xem liệu khóa được tìm kiếm có tồn tại hay không.

Ứng dụng hiệu quả của B-cây trong các hệ thống như vậy bắt nguồn từ đặc điểm quan trọng sau: mỗi đỉnh k của B-cây chứa từ $\lceil m/2 \rceil - 1$ đến $m-1$ phần tử. Với một giá trị được chọn đúng của m, hóa ra một đỉnh chiếm đúng một khối bộ nhớ (trong trường hợp của chúng ta là một khu vực vật lý). Bởi vì B-cây cân bằng hoàn toàn, chiều cao của chúng tối đa là $h = \log_{\lceil m/2 \rceil}(n+1)/2$), mà ở các giá trị lớn hơn của m là một con số đủ nhỏ, thậm chí là rất lớn n. Các thuật toán tìm kiếm, thêm và xóa một phần tử có độ phức tạp là bậc $\Theta(h)$ và thậm chí nhiều hơn h chúng truy cập không quá h với số đỉnh từ B-cây.

Cần lưu ý rằng truy cập bộ nhớ chặn là điển hình không chỉ đối với các thiết bị lưu trữ bên ngoài. Đây cũng là trường hợp của hệ thống bộ nhớ ảo. Hệ quản trị cơ sở dữ liệu quan hệ Oracle sử dụng một sửa đổi của B-cây cổ điển để tổ chức và lưu trữ các chỉ mục của nó. Hệ điều hành Windows của Microsoft cũng sử dụng B-cây đã sửa đổi, bao gồm cả việc tổ chức hệ thống tệp.

Bài tập

▷ **2.24.** Chứng minh rằng B-cây là cây tìm kiếm cân đối hoàn toàn.

▷ **2.25.** So sánh B-cây, cây đỏ đen và cây Fibonacci.

▷ **2.26.** Để xem xét các hoạt động cơ bản để làm việc với B-cây có thể được thực hiện như thế nào trong mô tả đã cho về cấu trúc của nó.

2.7. Bảng băm (H-bảng)

Bảng băm là một cấu trúc dữ liệu được đặc trưng bởi quyền truy cập trực tiếp vào các phần tử, bất kể kiểu của chúng. Độ phức tạp của các thao tác chính cơ bản (tìm kiếm, chèn, xóa và cập nhật) nói chung là không đổi, điều này làm cho nó cực kỳ hữu ích trong nhiều trường hợp. Khi định nghĩa bảng băm là một cấu trúc trừu tượng, chúng ta sẽ tự giới hạn trong ba phép toán cổ điển sau:

- `void` put(key, data) - đưa vào một phần tử
- data get(key); - tìm kiếm phần tử
- `void` remove(key) - Loại đi một phần tử

Như với danh sách duyệt và cây, bây giờ mỗi phần tử của bảng băm được đặc trưng bởi hai trường: khóa key và dữ liệu data. Khóa là

một định danh duy nhất: nếu hai phần tử có cùng một khóa, chúng được coi là giống hệt nhau.

Là một trường hợp đặc biệt của bảng băm, chúng ta có thể coi một mảng n phần tử trong đó chỉ số $i(i = 0, 1, ..., n - 1)$ là khóa của phần tử thứ i. Có thể thấy rằng hai phần tử có cùng khóa (chỉ mục) sẽ rơi vào cùng một phần tử của mảng. Ngoài ra, các thao tác đưa, xóa và tìm kiếm có độ phức tạp không đổi (do truy cập trực tiếp vào các phần tử của mảng). Trong thực tế, một mảng được triển khai khi thực hiện một bảng băm (tức là một cấu trúc có quyền truy cập trực tiếp đến các phần tử).

Quá trình mà khóa hằng số của một phần tử xác định địa chỉ của nó với độ phức tạp không đổi được gọi là băm.

2.7.1. Hàm băm (H-hàm số)

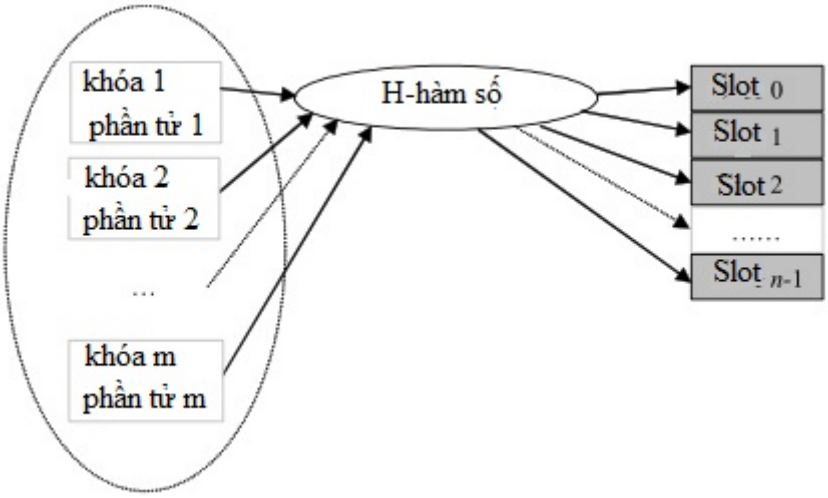
Chúng ta sẽ nhấn mạnh một lần nữa rằng khóa của phần tử có thể là bất kỳ cấu trúc nào xác định duy nhất nó. Ví dụ: nhân viên trong một công ty có thể tự nhận dạng mã PIN của họ (số tự nhiên gồm 10 chữ số, có thể có số 0 ở đầu. Trên thực tế, một nghiên cứu gần đây đã chỉ ra một số lỗi trong việc cấp mã PIN, bao gồm cả việc trùng lặp, khiến chúng trở thành khóa không đáng tin cậy. .) Hoặc tên (chuỗi ký hiệu, nhưng chỉ với điều kiện bổ sung là không có hai người trùng tên). Chúng ta sẽ trình bày chi tiết hơn về ví dụ đầu tiên: chúng ta giả định rằng khóa là một số có 10 chữ số. Chúng ta sẽ sử dụng một mảng a[] với n phần tử (số n được gọi là dung lượng bảng băm). Rõ ràng là nếu số lượng nhân viên nhỏ hơn hoặc bằng n, chúng ta sẽ có đủ bộ nhớ để ghi chúng vào mảng a[]. Tuy nhiên, để đạt được độ phức tạp không đổi của việc đưa vào, chúng ta phải có một hàm (hàm băm), hàm này trên khóa của mỗi phần tử để khớp duy nhất địa chỉ từ 0 đến $n - 1$. Nếu chúng ta coi các trường của một mảng là các vị trí (xem Hình 1), được đánh số từ 0 đến $n - 1$, trong đó một phần tử duy nhất có thể được đặt, thì cần có một hàm để xác định vị trí tương ứng bằng một khóa phần tử. .

Định nghĩa 2.14. Chúng ta xem xét một bảng băm với kích thước n và đặt U các giá trị cho phép đối với các khóa (vũ trụ). Sau đó, bằng

hàm băm, chúng ta sẽ hiểu hình ảnh:

$$h : U \to \{0, 1, ..., n-1\}$$

Ví dụ, đối với bất kỳ số k (PIN) có 10 chữ số nào, có thể lấy một số nguyên từ 0 đến $n-1$ bằng phép toán $k\%n$. Phần còn lại thu được bằng cách chia k cho dung lượng của bảng băm xác định một hàm băm khả thi. Trong trường hợp này, đặc điểm quan trọng thứ hai của bảng băm được chú ý: chúng ta có thể thấy mình ở trong tình huống mà các phần tử có các khóa khác nhau được ánh xạ tới cùng một địa chỉ từ mảng. Ví dụ, với $k_1 = 8004104369, k_2 = 8004102469$ và $n = 100$ ta có: $k_1\%n = k_2\%n = 69$, tức là các phần tử có hai khóa khác nhau nhận cùng một địa chỉ băm. Sự xuất hiện của một tình huống như vậy được gọi là xung đột và là một vấn đề lớn đối với bảng băm.



Hình 2.33. Hàm băm khớp với địa chỉ của từng khóa trong bảng băm.

2.7.2. Sự xung đột

Định nghĩa 2.15. Nếu cho hàm băm h và hai khóa $k_1, k_2 \in U$ thỏa mãn $h(k_1) = h(k_2)$, thì ta nói rằng các phần tử k_1 và k_2 *đang xung đột*. Chúng ta cũng sẽ nói rằng chúng *đồng nghĩa*.

Không khó để thấy rằng hàm băm xác định một *quan hệ tương đương*: tập hợp các từ đồng nghĩa của cùng một lớp và chỉ chúng tạo thành một *lớp tương đương*.

Nguyên lý (*nguyên lý Dirichlet, nguyên tắc ngăn kéo*): n ngăn kéo được đưa ra. Nếu chúng ta đặt $n + 1$ đối tượng vào chúng, thì sẽ có ít nhất hai đối tượng trong ít nhất một ngăn kéo.

Xung đột là không thể tránh khỏi - ngay cả khi chúng ta tạo một hàm băm "hoàn hảo", xung đột được đảm bảo sẽ xảy ra muộn nhất khi số lượng phần tử được đưa vào vượt quá khả năng của bảng băm (theo nguyên tắc Dirichlet). Trong thực tế, vụ xung đột sẽ xảy ra sớm hơn nhiều (như thể theo Định luật Murphy, nhưng thực tế là theo thống kê). Một ví dụ minh họa cho trường hợp thứ hai là "ví dụ về ngày tháng năm sinh": Xác suất để trong số 23 người có hai người sinh cùng ngày (tức là cùng ngày và tháng sinh) là bao nhiêu? Hóa ra xác suất này là một số trên $\frac{2}{1}$ một chút: nó được tính là $1 - \prod_{i=1}^{23} \frac{366 - i}{365} = 0,5063$. Phương trình trên thu được như thế nào? Hãy kí hiệu A là biến cố "*không có hai người trùng ngày sinh*". Khi đó xác suất yêu cầu sẽ bằng $1 - P(A)$. Để tính $P(A)$, chúng ta sẽ sử dụng định nghĩa cổ điển của xác suất: tỷ số giữa số sự kiện thuận lợi trên tổng số sự kiện cơ bản. Trong trường hợp của chúng ta, thuận lợi (liên quan đến A) là các sự kiện không có hai người (trong số 23 người) có cùng ngày sinh và họ có số lượng là $365, 364, 363...343$. Đồng thời, tất cả các loại cấu hình ngày tháng năm sinh đều được đưa ra bởi số 365^{23}. Do đó công thức trên dễ dàng làm theo. Như vậy, cơ hội chiến thắng sẽ nghiêng về phía chúng ta nếu đặt cược vào kèo cuối cùng trong bữa tiệc có đủ số lượng khách mời. Ở đây chúng ta đã giả định rằng một năm có 365 ngày, tức là nó không cao.

Như vậy, ngoài mong muốn chọn một hàm băm tốt, tức là dẫn đến một số xung đột tối thiểu, vấn đề giải quyết chúng là điều tối quan trọng. Trong hai điểm tiếp theo, chúng ta sẽ xem xét các hàm băm được sử dụng phổ biến nhất, cũng như các cách tiếp cận phổ biến nhất để đối phó với xung đột. Trong 2.5.3. chúng ta cũng sẽ đưa ra hai ví dụ triển khai bảng băm.

2.7.3. Hàm băm cổ điển

Việc lựa chọn một hàm băm có phù hợp hay không được xác định bởi hai điều: hàm băm không mất nhiều thời gian tính toán và phân phối các phần tử tương đối đồng đều trong các địa chỉ băm. Chúng ta sẽ giả định rằng khóa của các phần tử là số nguyên. Nếu không, chúng ta có thể so sánh chúng theo một số quy tắc: ví dụ: nếu các khóa là chuỗi, chúng ta có thể tính tổng các mã ASCII tương ứng. Do đó, đối với chuỗi "hello", chúng ta sẽ nhận được $104 + 101 + 108 + 108 + 111 = 532$. Nếu chúng ta chắc chắn rằng chuỗi chỉ bao gồm các chữ cái Latinh viết thường, thì chúng ta có thể trừ từ mã ASCII của mỗi chữ cái ASCII. mã của chữ cái 'a'. Điều này sẽ tránh bị tràn nếu chuỗi quá dài. Trong một số trường hợp, cách chuyển đổi từ khóa chuỗi thành số nguyên được mô tả dẫn đến kết quả rất kém. Ví dụ, đối với hàm băm ở trên, bất kỳ hoán vị nào của các ký tự sẽ cho cùng một mã băm; các chuỗi ngắn sẽ ở đầu bảng băm. Do đó, đôi khi số kết quả phụ thuộc vào vị trí của ký tự trong chuỗi là thích hợp. Nếu không có ràng buộc cộng đồng, chúng ta có thể giả định rằng chúng ta đang làm việc với các khóa số nguyên: mỗi chuỗi ký tự $a_1a_2...a_n$ có thể được coi là một số được viết trong hệ thống số gồm b chữ số (trong đó b là số các ký tự hợp lệ khác nhau).

Cho một bảng băm với dung lượng n và phần tử có khóa k. Dưới đây chúng ta sẽ liệt kê các hàm băm phổ biến nhất trong thực tế (trên một khóa số nguyên).

Lượng dư khi chia theo kích thước của bàn.

Ở trên chúng ta đã đề cập đến cách băm đơn giản và đồng thời khá hiệu quả này - khóa k được chia nguyên cho n và lấy phần dư của phép chia. Trong cách tiếp cận này, không phù hợp với dung lượng của bảng băm khi chọn số n, bậc của cặp: nếu $n = 2^p$, thì mã băm của một khóa k sẽ là p bit ít nhất của k. Ví dụ, cho $n = 2^4 = 16$ và $k = 173$. Biểu diễn nhị phân của 173 là 10101101. Phần còn lại 173%16 là 13, phần cuối cùng được viết trong hệ thống số nhị phân là 1101 - tức là chính xác là 4 bit thấp hơn của 173. (xem 1.1.6.)

Phép băm này sẽ hoạt động không tốt nếu phân phối khóa có một số lượng lớn các số với các bit thấp hơn phù hợp. Nói chung,

điều tốt là mã băm kết quả phụ thuộc vào tất cả các bit của khóa. Với mục đích này, một số nguyên tố thường được chọn cho dung lượng bảng băm.

Phép băm nhân đôi

Một phương pháp phổ biến khác là *băm nhân*. Một hằng số thực a, $0 < a < 1$. Đối với một khóa k cho trước, hàm băm có dạng:

$$h(k) = \lfloor n.\{k.a\} \rfloor$$

Ở đây chúng ta biểu thị bằng $\{k.a\}$ phần phân số của số thực, tức là $k.a - \lfloor k.a \rfloor$.

Mặc dù lựa chọn hằng số a (với ràng buộc $0 < a < 1$) là tùy ý, đối với một số giá trị, kết quả thực tế tốt hơn. Knuth [Knuth-3/1968] đề xuất sử dụng tỷ lệ vàng (xem 1.2.2.):

$$a = \frac{\sqrt{5}-1}{2} = 0,6180339887....$$

Hàm băm trên các bộ phận khóa

Trích số

Trong lược đồ này, chỉ các chữ số nằm ở các vị trí nhất định mới được trích xuất từ khóa (ví dụ, chúng ta có thể lấy các chữ số đầu tiên, thứ ba và thứ năm của số). Như vậy, với các khóa $123569, 425435, 546754, 676576$ ta sẽ nhận được địa chỉ băm lần lượt là $136, 453, 565, 667$. Có thể thấy đối với trường hợp cụ thể n phải là 1000 (vì khi giải nén chữ số $1, 3$ và 5 chúng ta có thể nhận được một số trong phạm vi $[0, 999]$). Phương pháp này hoạt động tốt khi các số không chứa nhiều chữ số lặp lại.

Gấp

Phương pháp này thường được sử dụng nhất khi các khóa có số lượng rất lớn. Có thể có sự đa dạng, nhưng nhìn chung tất cả đều dựa trên việc chia khóa thành các phần và thực hiện một số phép toán số học trên các phần kết quả. Ví dụ, số có thể được chia thành hai (hoặc ba hoặc nhiều phần) và tổng các số thu được có thể xác định địa chỉ băm.

Nâng giữa lên thành hình vuông

Lược đồ này dựa trên việc trích xuất các chữ số p ở giữa của khóa và bình phương chúng. Ví dụ: đối với khóa 125657134280980, ba chữ số trung bình là 134. Chúng ta bình phương chúng: $134.134 = 17956$. Nếu kết quả vượt quá n, một vài chữ số có nghĩa đầu tiên bị loại bỏ: ví dụ: nếu $n = 10001$, thì từ 17956 chữ số đầu tiên bị loại bỏ và địa chỉ băm kết quả là 7956. Lưu ý rằng phép toán cuối cùng không tương đương với việc tìm phần dư khi chia cho n.

So sánh một số hàm băm được xem xét

Để so sánh các phương pháp đã trình bày, chúng ta sẽ áp dụng kết quả của một thử nghiệm thực tế: Chúng ta xem xét một bảng băm với kích thước n và m số PIN ngẫu nhiên, chúng ta sẽ sử dụng làm khóa (được tạo theo cách chúng hợp lệ và ngày sinh là từ thế kỷ trước). Mục tiêu của chúng ta sẽ là so sánh sự phân bố của mã băm khi sử dụng từng hàm băm được thảo luận ở trên. Kết quả của thí nghiệm (tại $m = 1031000, n = 1031$) được trình bày trong Bảng **??**, Trong đó $\chi^2 = \frac{n}{m}\sum_{i=1}^{n}\left(f_i - \frac{m}{n}\right)^2$, và f_i là số khóa có mã băm, bằng i.

Hàm băm	χ^2
Phần dư khi chia cho 1031	729
Trích xuất các chữ số (số được tạo bởi 3 chữ số cuối cùng của mã PIN)	352
Gấp (chia mã PIN thành 1-3-3-3 và tổng các số thu được)	735
Bình phương số được tạo bởi ba chữ số trung bình của mã PIN.	233

Hình 2.34. χ^2 so sánh các hàm băm trên PIN, $n = 1031$, $m = 1031000$.

Đối với thống kê χ^2, người ta biết rằng nếu hàm băm là ngẫu nhiên (tức là nó được mong đợi hoạt động "tốt" như nhau trên bất kỳ bộ khóa nào, không chỉ đối với các số là mã PIN hợp lệ) và $m > cn$, thì χ^2 phải bằng $n \pm \sqrt{n}$ với xác suất $1 - 1/c$.

Có thể thấy rằng đối với ví dụ đã chọn (khóa PIN) và các giá trị $n = 1031$ và $m = 1031000$, phương pháp thứ ba và thứ nhất là hiệu quả nhất, phương pháp thứ hai và thứ tư dẫn đến băm không đồng

đều hơn.

Hàm băm trên chuỗi

Chuỗi ký tự là kiểu dữ liệu được băm phổ biến nhất. Đồng thời, việc tìm kiếm một hàm băm tốt là một vấn đề nghiêm trọng. Dưới đây là một số hàm băm được sử dụng phổ biến nhất cho các chuỗi ký tự, không có nghĩa là đã đầy đủ và không làm phiền người đọc về các chi tiết không cần thiết. Hoạt động của các hàm sẽ được giải thích tổng hợp trên cơ sở các đoạn chương trình cụ thể. Trước đó, chúng ta sẽ đưa ra sơ đồ chung nhất:

```
result = khởi tạo ();
while (c = next_character ()) {
result = kết hợp (result, c);
result = internal_modification (result);
}
result = add_modification (result);
```

Băm phụ gia

Đây là cách đơn giản nhất, phổ biến nhất (cổ điển), nhưng tiếc là cách băm kém hiệu quả nhất. Các mã ASCII của các ký hiệu được tính tổng và tổng được coi là một mô-đun của kích thước mảng.

```
unsigned long hashFunction(const char *key, unsigned long size)
{ unsigned long result = 0;
while (*key)
result += (unsigned char) *key++;
return result % size;
}
```

Thông thường độ dài chuỗi được bao gồm trong số lượng để nó có thể ảnh hưởng rõ ràng đến mã băm:

```
unsigned long hashFunction(const char *key, unsigned long size)
{ unsigned long result = strlen (key);
while (*key)
result += (unsigned char) *key++;
return result % size;
}
```

Lưu ý rằng độ dài của chuỗi ký tự có thể được truyền dưới dạng tham số, điều này giúp tiết kiệm quyền truy cập đắt tiền vào hàm

strlen(). Hoặc chỉ đơn giản là có được sự khác biệt giữa vị trí hiện tại trước và sau khi duyệt chuỗi (tuy nhiên, điều này không phải lúc nào cũng khả thi với các hàm băm tiếp theo, vì chúng thực hiện các thao tác result phức tạp hơn trong phần thân của vòng lặp):

```
unsigned long hashFunction(const char *key, unsigned long size)
{ const char *saveKey = key;
  unsigned long result = 0;
  while (*key)
     result  += (unsigned char) *key++;
  result  += saveKey - key;
  return result % size;
}
```

Kích thước của mảng thường được chọn là một số nguyên tố. Tuy nhiên, đôi khi một mảng có kích thước 2 được sử dụng, trong đó hàng cuối cùng có thể được đơn giản hóa thành:

```
return result & (size - 1);
```

hoặc tốt hơn:

```
hash = (hash ^ (hash>>10) ^ (hash>>20)) & mask;
```

Kỹ thuật này cũng hợp lệ cho các hàm bên dưới, có chứa dòng cuối cùng như vậy.

Băm xoay vòng

Không có bộ sưu tập ở đây, chỉ có hoạt động hàng loạt. Kích thước của mảng phải là một số nguyên tố.

```
unsigned long hashFunction(const char *key, unsigned long size)
{ unsigned long result = strlen(key);
  while (*key)
     result  = (result  << 4) ^ (result  >> 8) ^ ((unsigned char) *key+
     +);
  return result % size;
}
```

Băm từng cái một

Hàm là một biến thể của hàm trên nhưng với nhiều thao tác hơn trên kết quả biến, được thực hiện riêng biệt.

```
unsigned long hashFunction(const char *key, unsigned long size)
```

```
{ unsigned long result = 0;
  while (*key) {
     result  += (unsigned char) *key++;
     result  += result  << 10;
     result  ^= result  >> 6;
}
   result  += result  << 3;
   result  ^= result  >> 11;
   result  += result  << 15;
  return result % size;
}
```

Băm Pearson

Một mảng bổ sung tab[] được sử dụng ở đây, chứa hoán vị các số từ 0 đến 255. Lưu ý rằng mã băm kết quả là một byte: 0 đến 255. Có thể nhận được mã lớn hơn nếu hàm được gọi nhiều lần với các mảng khác nhau tab[], trong đó mỗi lệnh gọi sẽ cho một byte kết quả.

```
unsigned char hashFunction(const char *key, unsigned long size,
    const unsigned char tab[])
{ unsigned long result = strlen(key);
  while (*key)
     result  = tab[result  ^ ((unsigned char) *key++)];
  return result;
}
```

Băm CRC

Điều này yêu cầu các giá trị trong mảng được tạo bởi *Linear Feedback Shift Register*. Không đi vào quá nhiều chi tiết, chúng ta sẽ đề cập rằng cái sau là một thiết bị tạo ra một chuỗi nhị phân giả ngẫu nhiên thỏa mãn một số điều kiện nhất định.

```
unsigned long hashFunction(const char *key, unsigned long size,
     const unsigned long tab[])
{ unsigned long result = strlen(key);
  while (*key)
     result  = (result<<8) ^ tab[(result>>24) ^ ((unsigned char) *key
    ++)];
  return result % size;
```

```
}
```

Tổng quát CRC băm

Tương tự như trên, nhưng tab[] có thể chứa các giá trị tùy ý.

Băm phổ quát

Tuy nhiên, dù chúng ta chọn hàm băm nào thì trong trường hợp xấu nhất, nó sẽ gây ra $\Theta(n)$ xung đột trong đó n là kích thước của bảng băm. Tất nhiên, điều này sẽ được bù đắp bằng các cuộc gọi lặp đi lặp lại đến hàm, vì trường hợp xấu sẽ hiếm khi xảy ra.

Tuy nhiên, chúng ta có thể làm gì để bảo vệ mình khỏi những xung đột thường xuyên hơn? Câu trả lời là: để đảm bảo sự phân bố *đồng đều* của tập các giá trị cho phép của các khóa trên tập các chỉ số của mảng. Sau đó, xác suất so sánh cùng một chỉ mục trong một mảng cho hai chuỗi khác nhau, tức là một xung đột, sẽ là $1/n$.

Nhưng làm thế nào để đạt được nó? Một khả năng là kết hợp một số hàm băm để phân phối đồng đều các phần tử. Được rồi, nhưng có bao nhiêu tính năng như vậy để chọn? Gọi số giá trị cho phép khác nhau của khóa là m ($m = |U|$), và số hàm ta cần - f. Khi đó chúng ta sẽ muốn $f/m = 1/n$. nghĩa là, chúng ta sẽ cần m/n của một số hàm khác nhau, mỗi hàm phân phối đều các khóa trên các chỉ số của mảng.

Định nghĩa 2.16. Tập hợp $H = \cup_a \{h_a\}$ của các hàm băm được gọi là *tập hợp phổ quát của các hàm băm* nếu:

- $|H| = m/n$ - chứa đúng m/n trong hàm số h_a
- $P(h_a(x) = h_a(y)) = /n$ - xác suất xung đột là $1/n$

Một cách khả thi để có được một tập hợp phổ quát các hàm băm là chọn m/n hàm được định nghĩa là (a là một tham số khác nhau cho mỗi hàm):

$$h_a(k) = \left(\sum_{i=0}^{r} a_i k_i \right) \bmod n$$

ở đâu:
$k = (k_0, k_1, ..., k_r)$ đang phá khóa k thành $r+1$ phần (ví dụ: byte)
$a = (a_0, a_1, ..., a_r)$, một vectơ có các thành phần được chọn ngẫu nhiên từ $0, 1, ..., n-1$.

Sau đây là một ví dụ thực hiện. Mảng tab[] chứa bao nhiêu phần tử bằng số bit tối đa trong chuỗi đầu vào, được chọn ngẫu nhiên từ $\{0, 1, ..., n-1\}$. Chúng ta rời các chi tiết để người đọc.

```
unsigned long hashFunction(const char *key, unsigned long size,
    const unsigned long tab[MAXBITS])
{ unsigned char k;
  unsigned i;
  unsigned long result;
  unsigned long l3 = (result = strlen(key)) << 3;
  for (i = 0; i < l3; i += 8)
  {
    k = (unsigned char) key[i >> 3];
    if (k&0x01) result ^= tab[i+0];
    if (k&0x02) result ^= tab[i+1];
    if (k&0x04) result ^= tab[i+2];
    if (k&0x08) result ^= tab[i+3];
    if (k&0x10) result ^= tab[i+4];
    if (k&0x20) result ^= tab[i+5];
    if (k&0x40) result ^= tab[i+6];
    if (k&0x80) result ^= tab[i+7];
  }
  return result % size;
}
```

Hàm băm Zobrist

Ở đây, bảng tab[][] là hai chiều và các giá trị lại được chọn ngẫu nhiên từ $\{0, 1, .., n-1\}$. Lựa chọn cẩn thận có thể dẫn đến băm phổ quát.

```
    unsigned long hashFunction(const char *key, unsigned long size
    ,
const unsigned long tab[MAXBYTES][256])
{ unsigned i;
  unsigned long result = strlen(key);
  for (i = 0; i < len; i++)
      result ^= tab[i][*key++]
  return result % size;
}
```

Bài tập

▷ **2.27.** 1. So sánh các hàm băm được thảo luận ở trên với:

- độ phức tạp tính toán
- tính đồng nhất của phân phối chính

▷ **2.28.** Những vấn đề nào mà hàm băm gặp phải trên các bộ phận quan trọng?

▷ **2.29.** Bạn thấy ưu và nhược điểm nào đối với mỗi hàm băm được xem xét cho chuỗi ký tự?

▷ **2.30.** Có phải một hàm băm phân phối các phần tử đồng đều hơn luôn luôn tốt hơn?

▷ **2.31.** Để so sánh hàm băm phổ quát và hàm băm của Zobrist.

▷ **2.32.** Tập hợp m/n được cho bởi số hàm băm được xác định là (a là một tham số khác nhau đối với mỗi hàm):

$$h_a(k) = \left(\sum_{i=1}^{r} a_i k_i \right) \bmod n,$$

ở đâu:
$k = (k_0, k_1, ..., k_r)$ đang chia khóa x thành $r + 1$ phần (ví dụ: byte)

$a = (a_0, a_1, ..., a_r)$, một vectơ có các thành phần được chọn ngẫu nhiên từ $\{0, 1, .., n - 1\}$.

n - kích thước của bảng băm

m - lũy thừa của tập các giá trị cho phép của khóa.

Dựa trên Định nghĩa **??**, Để chứng minh rằng đây là một tập hợp phổ quát của các hàm băm.

▷ **2.33.** Bổ sung Bảng **??** với băm nhân.

▷ **2.34.** Để xem xét có thể áp dụng sự kết hợp (hoặc sửa đổi) nào của các phương pháp được xem xét để thu được kết quả tốt hơn nữa (trên một bộ mã PIN được chọn ngẫu nhiên, n và m)? χ^2 có thể được mong đợi là $n \pm \sqrt{n}$ nếu một hàm băm được biên dịch hoạt động tối ưu cho hàm băm mã PIN cụ thể không?

▷ **2.35.** Lặp lại thí nghiệm đã mô tả cho các giá trị khác nhau của n và m. Các kết quả trên đã được xác nhận chưa? Khi tạo dữ liệu thử nghiệm, có thể sử dụng thuật toán để kiểm tra tính đúng đắn của số PIN từ Bài toán 1.203.

▷ **2.36.** Bạn có đồng ý với phương pháp lấy dữ liệu cho bài kiểm tra điền dữ liệu từ Hình **??** (tạo với một hàm cho các số giả ngẫu nhiên, nhưng với phân phối nào ...)? Điều này có ảnh hưởng gì đến kết quả cuối cùng? Bạn có thể đề xuất một giải pháp cho vấn đề?

2.7.4. Đối phó với xung đột

Chúng ta sẽ minh họa các sơ đồ quản lý xung đột khác nhau bằng một ví dụ cụ thể. Hãy đưa ra một bảng băm có dung lượng $n = 10$. Chúng ta sẽ sử dụng một hàm băm đơn giản dựa trên việc trích xuất các chữ số - chữ số đầu tiên của khóa sẽ là địa chỉ băm của chúng ta. Trong ví dụ, các mục chúng ta muốn đưa vào là các khóa $234, 235, 567, 123, 534$ và 647.

Hàm băm đã đóng

Kiểm tra tuyến tính

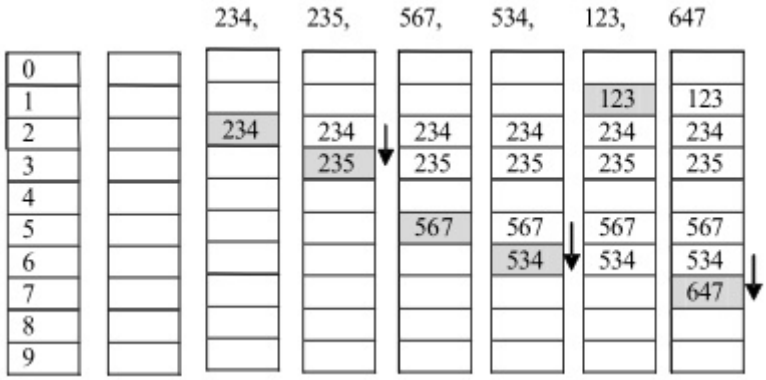
Kiểm tra tuyến tính, kiểm tra bậc hai và băm kép là một phần của sơ đồ giải quyết xung đột tổng quát hơn được gọi là *băm đóng*. Với nó, chúng ta có một nơi chính để lưu trữ dữ liệu (n "slot", trái ngược với việc triển khai có một phần bổ sung cho các xung đột). Khi xung đột xảy ra, chúng ta cố gắng thay đổi địa chỉ băm kết quả một cách tuần tự cho đến khi chúng ta đạt được vị trí trống. Sự thay đổi được thực hiện theo một sơ đồ xác định trước.

Ví dụ, có thể tăng địa chỉ của một số tự nhiên s ($0 < s < n$). Nếu địa chỉ trở nên bằng n, chúng ta tiếp tục tìm kiếm ở đầu bảng băm, lấy phần dư modulo n. Để có thể thu thập thông tin tất cả các địa chỉ của bảng băm ở độ phóng đại như vậy, n và s phải nguyên tố lẫn nhau, tức là $GNP(n, s) = 1$.

Rõ ràng, cách tiếp cận này không thể bao gồm nhiều phần tử hơn dung lượng của bảng băm (xem Hình 2.35) và khi nó đầy, cần phải mở rộng nó (cấp phát bộ nhớ cho một mảng lớn hơn và giải phóng phần bị chiếm từ mảng bộ nhớ hiện tại). Chúng ta sẽ xem điều này được triển khai như thế nào trong thực tế trong lần triển khai thứ

hai của bảng băm bên dưới (xem 2.5.3).

Chèn phần tử:



Hình 2.35. Bao gồm tuần tự của một phần tử. Cho phép xung đột với thử nghiệm tuyến tính với bước 1.

Khi tìm kiếm một phần tử theo khóa, hãy tiến hành theo cách tương tự - đầu tiên địa chỉ băm được tính bằng hàm băm và trong trường hợp không tìm thấy khóa được yêu cầu, kiểm tra tuyến tính (thay đổi địa chỉ) được thực hiện cho đến khi đạt được yêu cầu . phần tử. Sau đó giả định rằng khóa nằm trong bảng băm.

Trường hợp $s = 1$ là phổ biến: trong trường hợp va chạm, người ta di chuyển đến địa chỉ tiếp theo, v.v. với bước 1. Trong trường hợp này, cũng như trong trường hợp tổng quát hơn, có thể gặp các khóa mà việc băm đã tỏ ra không hiệu quả. Ví dụ: khi các phần tử tạo thành nhóm với mã băm gần nhau (cái gọi là "cụm"). Một loạt các va chạm mới (thứ cấp) sau đó phát sinh trong quá trình giải quyết va chạm (xem Hình 2.35). Dưới đây chúng ta sẽ xem xét hai kiểu băm đóng quan trọng khác.

Kiểm tra bậc hai

Trong *thử nghiệm bậc hai*, như tên cho thấy, một bước của loại $c_1 i + c_2 i^2, c_2 \neq 0$. Độ lệch phụ thuộc bậc hai vào số sê-ri của mẫu i. Phương pháp này hoạt động hiệu quả hơn đáng kể so với kiểm tra tuyến tính, mặc dù nó cũng cho thấy những tác động tiêu cực trong trường hợp một cụm khóa (tích lũy thứ cấp của các khóa). Tuy nhiên, vấn đề chính của nó là nó *không* đảm bảo thu thập dữ

liệu toàn bộ bảng, tức là anh ta có thể không tìm thấy chỗ trống, mặc dù có một chỗ. [Cormen, Leiserson, Rivest-1997].

Băm kép

Hàm băm kép tương tự như hai phương pháp được xem xét, sử dụng hai hàm băm:

$$h(k,i) = h_1(k) + i.h_2(k)$$

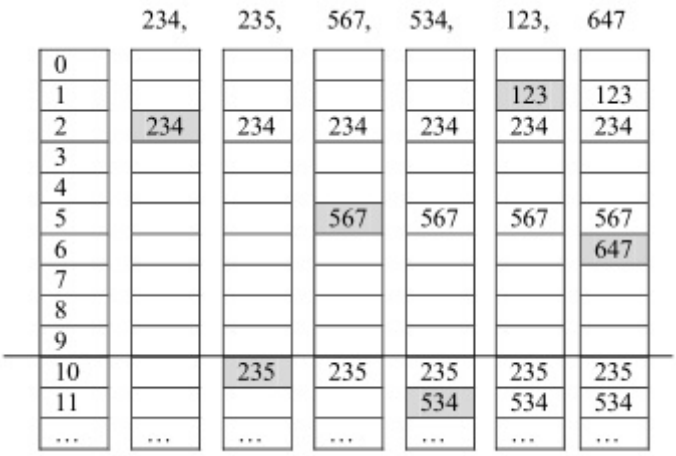
Đây i là số mẫu sau lần xung đột đầu tiên. Hàm băm thứ hai chỉ được sử dụng trong trường hợp kết quả của lần đầu tiên dẫn đến xung đột. Cả khóa gốc của phần tử và địa chỉ băm thu được khi áp dụng hàm băm đầu tiên đều có thể được sử dụng làm khóa để băm lại.

Mở băm

Bộ nhớ bổ sung cho các xung đột

Như tên cho thấy, lược đồ này cung cấp thêm bộ nhớ để xử lý xung đột. Mỗi phần tử xung đột nằm trong không gian trống đầu tiên trong bộ nhớ phụ (xem Hình 2.36).

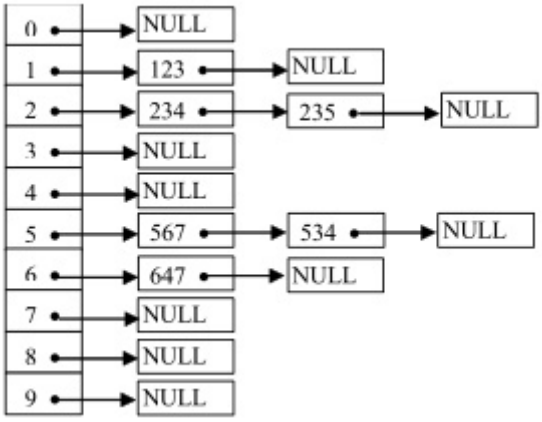
Chèn phần tử:



Hình 2.36. Giải quyết xung đột bằng cách phân bổ bộ nhớ bổ sung.

Khi tìm kiếm trong bảng băm, địa chỉ thu được từ hàm băm sẽ được kiểm tra đầu tiên và nếu phần tử có khóa được tìm kiếm không có ở đó, thì toàn bộ phần bổ sung sẽ được tìm kiếm.

Danh sách tràn



Hình 2.37. Kết nối động: bảng băm, sau khi bao gồm các phần tử 234, 235, 567, 123, 534, 647.

Cũng có thể sử dụng bộ nhớ được cấp phát động bên ngoài: Các phần tử có địa chỉ băm phù hợp nằm trong một số cấu trúc dữ liệu chuẩn. Cấu trúc được sử dụng phổ biến nhất nhưng không nhất thiết là hiệu quả nhất cho mục đích này là danh sách liên kết. Cách tiếp cận này để xử lý các xung đột thường được gọi là danh sách tràn: mỗi vị trí trong bảng băm là một con trỏ đến danh sách liên kết động chỉ chứa các thành viên của lớp từ đồng nghĩa tương ứng. Khi một phần tử mới được đưa vào, nó sẽ được thêm vào đầu danh sách nằm trên vị trí được xác định bởi hàm băm (xem Hình 2.37). Do đó, việc cẩn thận khi va chạm đương nhiên bị loại bỏ.

Tương tự, khi tìm kiếm, danh sách được xác định bởi hàm băm sẽ được xem. Độ phức tạp của thao tác tìm kiếm sau đó phụ thuộc vào số lượng phần tử mà phần tử đã cho va chạm với nhau, tức là vào độ dài của danh sách liên kết động tương ứng.

Các phần tử va chạm có thể được lưu trữ trong một cấu trúc trong đó tìm kiếm ít mang tính thuật toán hơn so với cấu trúc của danh sách tuyến tính - ví dụ: cây tìm kiếm nhị phân. Do đó, mỗi trường trong bảng băm sẽ là một con trỏ tới gốc của một cây nhị phân riêng biệt.

Bài tập

▷ **2.37.** Để so sánh băm mở và đóng: ưu điểm và nhược điểm. Khi nào mỗi người trong số họ nên được ưu tiên?

▷ **2.38.** Tại sao kiểm tra bậc hai không thể đảm bảo truyền toàn bộ mảng? Điều này có đúng với tất cả các hàm bình phương không?

2.7.5. Triển khai bảng băm

Chúng ta sẽ xem xét hai cách triển khai khác nhau của một bảng băm. Đầu tiên, khóa của bảng băm là số nguyên (điều này quan trọng trong việc thực hiện hàm băm, sẽ theo nguyên tắc chia cho phần dư). Chúng ta sẽ giải quyết các vụ va chạm bằng cách sử dụng một danh sách được liên kết động. Điều này sẽ minh họa cấu trúc dữ liệu hiệu quả có thể được xây dựng nhanh chóng và dễ dàng như thế nào (lưu ý rằng loại trừ việc triển khai danh sách được liên kết, mã bổ sung cho bảng băm rất ngắn).

Dung lượng của bảng băm N được đặt ở đầu chương trình dưới dạng macro. Chi tiết triển khai có thể được tìm thấy dưới dạng nhận xét trong mã nguồn.

Chương trình 2.5. Bảng băm (206hash.c)

```
#include <stdio.h>
#include <stdlib.h>
#define N 211
typedef int data;
typedef long keyType;
#define NOT_EXIST (-1) /*trả về từ get () khi thiếu phần tử*/
struct list {
  keyType key;
  data info;
  struct list *next;
};
struct list *hashTable[N];

/* bao gồm một phần tử ở đầu danh sách được liên kết*/
void insertBegin(struct list **L, keyType key, data x)
{
  struct list *temp;
```

```
    temp = (struct list *) malloc(sizeof(*temp));
    if (NULL == temp) {
      fprintf(stderr, "Không đủ bộ nhớ cho mục mới!\n");
      return;
    }
    temp->next = *L;
    (*L) = temp;
    (*L)->key = key;
    (*L)->info = x;
}

/* xóa một phần tử khỏi danh sách */
void deleteNode(struct list **L, keyType key)
{ struct list *current = *L;
  struct list *save;
  if ((*L)->key == key) { /*phần tử đầu tiên phải được xóa */
    current = (*L)->next;
    free(*L);
    (*L) = current;
    return;
  }

  /* tìm phần tử cần xóa */
while (current->next != NULL && current->next->key != key)
  current = current->next;
if (NULL == current->next) {
  fprintf(stderr, "Lỗi: Không tìm thấy phần tử cần xóa! \n");
  return;
}
else {
  save = current->next;
  current->next = current->next->next;
  free(save);
  }
}

/*tìm kiếm một phần tử trong danh sách được liên kết*/
struct list* search(struct list *L, keyType key)
{ while (L != NULL) {
    if (L->key == key) return L;
```

```
    L = L->next;
  }
  return NULL;
}

unsigned hashFunction(keyType key)
{ return(key % N); }

void initHashTable(void)
{ unsigned i;
  for (i = 0; i < N; i++) hashTable[i] = NULL; }

void put(keyType key, data x)
{ int place = hashFunction(key);
  insertBegin(&hashTable[place], key, x); }

data get(keyType key)
{ int place = hashFunction(key);
  struct list *l = search(hashTable[place], key);
  return (NULL != l) ? l->info : NOT_EXIST; }

int main() {
  initHashTable();
  put(1234, 100); /* -> trong slot 179 */
  put(1774, 120); /* ->trong slot 86 */
  put(86, 180); /* ->trong slot 86 -> xung đột */
  printf("In dữ liệu cho phần tử có khóa 86: %d \n", get(86));
  printf("In dữ liệu cho phần tử có khóa 1234: %d \n", get(1234));
  printf("In dữ liệu cho phần tử có khóa 1774: %d \n", get(1774));
  printf("In dữ liệu cho phần tử có khóa 1773: %d \n", get(1773));
  return 0;
}
```

Kết quả thực hiện chương trình:

```
In dữ liệu cho phần tử có khóa 86: 180
In dữ liệu cho phần tử có khóa 1234: 100
In dữ liệu cho phần tử có khóa 1774: 120
In dữ liệu cho phần tử có khóa 1773: -1
```

Ví dụ thứ hai, chúng ta sẽ xem xét một bài toán thực tế cụ thể được giải quyết hiệu quả bằng cách sử dụng bảng băm. Trong phần

triển khai này, chúng ta sẽ tập trung vào một số vấn đề cụ thể hơn: băm chuỗi ký tự, mở rộng bảng băm khi mảng được cấp phát trước không đủ và hơn thế nữa.

Bài toán: Một văn bản (danh sách các từ) được đưa ra, cách nhau bởi một khoảng trắng. Tìm tần suất xuất hiện của mỗi từ trong văn bản.

Bài toán được mô tả thường phát sinh như một phần của các vấn đề thực tế phức tạp hơn. Ví dụ, phân tích một tài liệu từ một công cụ tìm kiếm (ví dụ: Google), trước khi xử lý thêm, hầu như nhất thiết phải bao gồm việc tìm số lần xuất hiện của mỗi từ trong tài liệu.

Để giải quyết vấn đề một cách hiệu quả với bảng băm, một số điều cơ bản cần được xem xét:

- Khóa bảng băm sẽ là một chuỗi ký tự - đây là một từ trong tài liệu. Ngoài ra, tần suất xuất hiện của từ sẽ được giữ nguyên. Đối với một số tác vụ, có thể không có dữ liệu bổ sung. Khi đó cấu trúc được gọi là tập hợp băm.
- Các xung đột sẽ được giải quyết bằng thử nghiệm tuyến tính, kích thước của bảng băm n sẽ là lũy thừa của 2. Lưu ý rằng trong triển khai đề xuất, vị trí của biểu tượng là quan trọng, tức là *abc* và *bca* sẽ có các mã băm khác nhau. Khi xem xét một hàm băm dựa trên phần dư khi chia cho một số nguyên tố trong 2.5.1, chúng ta đã chỉ ra bằng một ví dụ cụ thể rằng đây không phải là một cách tiếp cận hiệu quả, vì mã băm không phụ thuộc vào tất cả các bit của khóa. Tuy nhiên, trong trường hợp này, đây không phải là vấn đề nghiêm trọng, vì các khóa trên thực tế là các chuỗi ký tự và dự kiến sẽ được chuyển đổi thành một số trước khi hoạt động phần còn lại của mô-đun để đảm bảo phân phối đồng đều hơn ở các bit thấp hơn. Ưu điểm của việc sử dụng bảng băm có kích thước này là khả năng sử dụng thao tác nhanh hơn - bitwise "và", thay vì chia số nguyên với phần dư. Bước $s (s > 2)$ mà chúng ta sẽ thực hiện phép thử sẽ là một số nguyên tố: do đó chúng ta đảm bảo rằng thuộc tính n và s thoả mãn rằng chúng là nguyên tố cùng nhau.
- Nếu kích thước được xác định trước cho bảng băm chứng tỏ không đủ, chúng ta sẽ cấp phát bộ nhớ cho nhiều phần tử hơn:

chúng ta sẽ tăng gấp đôi kích thước của nó và do đó giữ thuộc tính dung lượng của nó là lũy thừa 2.

Chương trình 2.6. Bảng băm tập hợp (206hashset.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define S 107 /*bước phóng đại xung đột*/
#define MAX_FILL_LEVEL 0.8 /* Mức lấp đầy tối đa */
unsigned long n = 32; /* kích thước ban đầu của bảng băm */

struct singleWord {
  char *word; /* từ khóa - chuỗi ký tự*/
  unsigned long freq; /* tần suất xuất hiện của từ */
} *ht;
unsigned long count;

/* Hàm băm chuỗi ký tự */
unsigned long hashFunction(const char *key)
{ unsigned long result = 0;
  while (*key)
    result += result + (unsigned char) *key++;
  return result & (n - 1); }

/* Khởi tạo bảng băm */
void initHashtable(void)
{ unsigned long i;
  count = 0;
  ht = (struct singleWord *) malloc(sizeof(*ht)*n);
  for (i = 0; i < n; i++)
    ht[i].word = NULL;
}

/* Tìm kiếm trong bảng băm: trả về 1 nếu thành công và 0 - nếu khô
    ng */
/* Khi thành công: * ind chứa chỉ số của phần tử được tìm thấy */
/* Không thành công: vị trí trống, nơi có thể lắp vào */
char get(const char *key, unsigned long *ind)
{ unsigned long k;
  *ind = hashFunction(key);
  k = *ind;
```

```
  do {
    if (NULL == ht[*ind].word) return 0;
    if (0 == strcmp(key, ht[*ind].word)) return 1;
    *ind = (*ind + S) & (n - 1);
  } while (*ind != k);
  return 0;
}

/* Mở rộng bảng băm*/
void resize(void)
{ unsigned long ind, hashInd;
  struct singleWord *oldHashTable;

  /*  1. Lưu thư mục vào bảng băm */
  oldHashTable = ht;

  /* 2. Mở rộng gấp đôi */
  n <<= 1;

  /* 3. Phân bổ bộ nhớ cho bảng băm mới*/
  ht = (struct singleWord *) malloc(sizeof(*ht)*n);
  for (ind = 0; ind < n; ind++)
    ht[ind].word = NULL;

/*4. Di chuyển các bản ghi sang bảng băm mới*/
for (ind = 0; ind < (n >> 1); ind++) {
  if (oldHashTable[ind].word != NULL) {
    /* Di chuyển bản ghi đến vị trí mới*/
    if (!get(oldHashTable[ind].word, &hashInd))
      ht[hashInd] = oldHashTable[ind];
    else
      printf("Lỗi phần mở rộng bảng băm!\n");
    }
  }

  /* 5. Giải phóng bộ nhớ cũ */
  free(oldHashTable);
}

/* Thêm một phần tử vào bảng băm */
```

```
void put(char *key)
{ unsigned long ind;
if (!get(key, &ind)) { /* Từ không có trong bảng băm */
ht[ind].word = strdup(key);
ht[ind].freq = 1;
if (++count > ((unsigned long) n * MAX_FILL_LEVEL)) resize();
}
else ht[ind].freq++; }

/* In bảng băm */
void printAll(void)
{ unsigned long ind;
  for (ind = 0; ind < n; ind++)
    if (ht[ind].word != NULL)
      printf("%s %ld \n", ht[ind].word, ht[ind].freq);
}

/* Hủy bảng băm */
void destroy(void)
{ unsigned long ind;
  for (ind = 0; ind < n; ind++)
    if (ht[ind].word != NULL) free(ht[ind].word);
  free(ht);
}

int main() {
  unsigned long find;
  initHashtable();
  put("reload");
  put("crush tour");
  put("room service");
  put("load");
  put("reload");
  put("reload");

  printAll();

  if (get("reload", &find))
    printf("Tần suất từ 'reload': %d \n", ht[find].freq);
  else
```

```
    printf("Thiếu từ 'reload'!");

  if (get("download", &find))
    printf("Tần suất từ 'download': %d \n", ht[find].freq);
  else
    printf("Thiếu từ 'download'!");
  destroy();
  return 0;
}
```

Kết quả thực hiện chương trình:

```
load 1
reload 3
crush tour 1
room service 1
Tần suất của từ 'reload': 3
Thiếu từ 'download'!
```

Bài tập

▷ **2.39.** Để triển khai một hàm để loại trừ một phần tử khỏi bảng băm bằng một khóa. Độ phức tạp phải không đổi. Lưu ý rằng không đủ nếu chỉ tìm phần tử có khóa được chỉ định và xóa nó, như trong quá trình triển khai với danh sách tràn: nghĩa là điều này sẽ phá vỡ chuỗi các phần tử có cùng mã băm. của chuỗi các từ đồng nghĩa.

▷ **2.40.** Có thể sửa đổi thuật toán mở rộng bảng băm từ 2.4.3 không. (thực hiện nguyên tắc kiểm tra tuyến tính) để không cần phải di chuyển từng phần tử (làm nóng lại) mà chỉ cần sao chép bộ nhớ?

▷ **2.41.** Thực hiện một phương pháp để "thu nhỏ" bảng băm: giảm kích thước của nó khi số lượng phần tử của nó giảm. Điều gì có thể là tiêu chí để giảm kích thước của bảng băm? Nên tăng hoặc giảm tốc sau khi đầm nén? Tại sao?

▷ **2.42.** Trong chương trình đề xuất ở trên, không được phép điền bảng băm trên 80%: khi đạt đến mức này, nó sẽ tự động mở rộng. Các nghiên cứu chỉ ra rằng với mức độ lấp đầy như vậy và một hàm băm được chọn đúng cách, số lượng mẫu trung bình nhỏ hơn 2. Để

đánh giá thực nghiệm mối quan hệ giữa số lượng mẫu trung bình và mức độ lấp đầy của bảng băm. Số lượng mẫu trung bình tại thời điểm điền đầy là bao nhiêu: 85%, 90%, 93%, 95%, 97%, 98%, 99%, 100%?

▷ **2.43.** Trong cách triển khai được đề xuất ở trên, giá trị của hàm băm có tính đến vị trí của các ký hiệu:

```
result += result + (unsigned char) * key ++;
```

Để so sánh với hàm băm cổ điển cho các chuỗi ký tự:

```
result = result + (unsigned char) * key ++;
```

Ước tính số lượng mẫu trung bình cần thiết để phát hiện một phần tử có khóa hoặc xác định rằng phần tử đó bị thiếu trong mỗi một trong hai hàm băm.

▷ **2.44.** Trong cách triển khai được đề xuất ở trên, chúng ta đã sử dụng băm đóng với các mẫu tuyến tính. Thử nghiệm với các chiến lược quản lý xung đột tiêu chuẩn khác như lấy mẫu bậc hai và băm kép với các chức năng được chọn phù hợp. Số lượng mẫu trung bình giảm hay tăng? Tỷ lệ giữa số lượng mẫu trung bình trong các chiến lược quản lý xung đột khác nhau có phụ thuộc vào mức độ hoàn thành của bảng băm không? Và kích thước của nó? Lớp 2 có phải là lựa chọn tốt?

▷ **2.45.** Để kiểm tra việc triển khai được đề xuất ở trên trên các dữ liệu đầu vào khác nhau. Số lượng mẫu trung bình có thay đổi trong quá trình tìm kiếm không?

2.8. Câu hỏi và bài tập

▷ **2.46.** *Danh sách liên kết đôi*
Để biên dịch triển khai động của danh sách tuyến tính hai liên kết. Một danh sách tuyến tính liên kết đôi được định nghĩa một cách đệ quy như sau:

```
typedef int data;
typedef int keyType;

struct list {
keyType key;
```

```
data info;
struct list *prev;
struct list *next;
};
```

Mỗi phần tử trong danh sách liên kết đôi có hai con trỏ - đến phần tử tiếp theo (như trong danh sách liên kết đơn) và đến phần tử trước (con trỏ trước prev).

Hãy hiện thực các thao tác cơ bản để làm việc với danh sách tuyến tính - bao gồm một phần tử (trước và sau một phần tử được chỉ định bởi con trỏ), cũng như để thực hiện (với độ phức tạp không đổi), thao tác xóa một phần tử được chỉ ra bởi một con trỏ.

▷ **2.47.** *Danh sách tuần hoàn*
Danh sách liên kết đơn theo chu kỳ được gọi là danh sách liên kết đơn tuyến tính, trong đó con trỏ đến mục tiếp theo bên cạnh mục cuối cùng trỏ đến mục đầu tiên trong danh sách.

Hãy thiết lập triển khai động của danh sách liên kết đơn theo chu kỳ và triển khai các hoạt động cơ bản để làm việc với danh sách đó.

▷ **2.48.** *Hàng đợi ưu tiên*
Hàng đợi ưu tiên tìm thấy nhiều ứng dụng (bộ nhớ đệm, v.v.). Mỗi phần tử, ngoài dữ liệu mà nó chứa, được đặc trưng bởi một số nguyên - "mức độ ưu tiên" của nó. Khi một phần tử được đưa vào hàng đợi, điều này xảy ra không phải ở cuối mà ngay sau phần tử cuối cùng có mức ưu tiên cao hơn phần tử của nó (nếu các phần tử bằng nhau, thứ tự theo thứ tự nhận). Do đó, tất cả các phần tử có mức độ ưu tiên cao hơn nằm trước nó và các phần tử có mức độ ưu tiên thấp hơn - sau nó trong hàng đợi. Việc loại trừ khỏi hàng đợi được thực hiện ngay từ đầu - luôn có phần tử có mức ưu tiên cao nhất.

Viết chương trình mô phỏng hàng đợi ưu tiên và thực hiện các thao tác bật và tắt cơ bản của bản ghi độ phức tạp (log 2 n) [Nakov-1998] [Wirth-1980].

Lưu ý: Cấu trúc dữ liệu kim tự tháp được thảo luận trong 3.1.9. "Phân loại kim tự tháp của Williams." Ngoài việc triển khai hiệu quả hàng đợi ưu tiên, kim tự tháp còn được sử dụng để giảm độ phức

tạp trong nhiều thuật toán khác - thuật toán Dijkstra (xem 5.4.2.), V.v.

▷ **2.49.** *Hàng đợi "nhóm"*
Một loại đuôi ít được biết đến, thường thấy trong cuộc sống hàng ngày, được gọi là đuôi. hàng đợi "đội". Ví dụ, hàng đợi hình thành trước ghế học sinh là hàng đợi "đội". Khi một mục mới tham gia vào hàng đợi, trước tiên nó sẽ tìm kiếm "người quen" của mình trong đó và nếu được tìm thấy, sẽ tham gia ngay sau họ. Nếu anh ta không tìm thấy nó, nguyên tố mới sẽ không còn may mắn và sẽ xếp hàng. Loại trừ một phần tử khỏi hàng đợi là cách tiêu chuẩn - chỉ từ đầu hàng đợi. Viết chương trình mô phỏng hàng đợi "nhóm" cho một số nhóm phần tử nhất định (các phần tử thuộc một nhóm được coi là "đã biết").

▷ **2.50.** *Đảo ngược bản ghi trường*
Viết chương trình tính giá trị của một biểu thức được viết bằng ký hiệu Ba Lan ngược. Để giải quyết vấn đề sử dụng một ngăn xếp [Nakov-1998].

▷ **2.51.** *Khôi phục biểu thức*
Viết chương trình khôi phục ký hiệu đại số cổ điển của biểu thức có dấu ngoặc từ một biểu thức số học đã cho được viết bằng ký hiệu trường ngược. Để giải quyết vấn đề theo hai cách: có và không sử dụng gỗ nhị phân [Nakov-1998].

▷ **2.52.** *Phân tích cú pháp nhanh*
Viết chương trình đọc mã nguồn của chương trình C và kiểm tra xem (các) cặp `{ }`; `{ i } ; [ i ] ; /* i */` tham gia một cách đối xứng trong đó.

Gợi ý: Sử dụng ngăn xếp.

▷ **2.53.** *Mô phỏng đệ quy*
Khi các cuộc gọi đệ quy được thực hiện, dữ liệu cục bộ cho mỗi cuộc gọi kế tiếp được lưu trữ trong bộ nhớ ngăn xếp (ở cấp trình biên dịch / hệ điều hành, quá trình này như sau: có một phần bộ nhớ được gọi là ngăn xếp chương trình và một thanh ghi con trỏ lên trên cùng của ngăn xếp).

Mỗi hàm đệ quy có thể được thay thế bằng một biến thể lặp tương đương. Điều này có thể được thực hiện bằng một trong những cách sau:

- Tất cả các tham số của hàm đệ quy được xuất dưới dạng biến toàn cục và được sửa đổi cho phù hợp. Cách tiếp cận này liên quan đến việc sửa đổi logic của mã chương trình và không phải lúc nào cũng có thể thực hiện được. (Tại sao?)
- Mô phỏng tổ chức ngăn xếp chương trình: Sử dụng ngăn xếp của riêng bạn, ngăn xếp này ghi lại các giá trị cuối cùng của tất cả các biến liên quan đến hàm. Đến lượt nó, hàm hoạt động với tập biến cuối cùng được lưu trữ trong ngăn xếp.

Để thực hành kỹ thuật thứ hai, hãy giải quyết các tác vụ sau bằng cách sử dụng ngăn xếp:

- Nâng một số thực lên lũy thừa.
- Tìm số Fibonacci thứ n. Để duy trì tính logic và tính không hiệu quả của việc thực hiện đệ quy trực tiếp, xem 1.2.2.
- Bài toán của Hanoi Towers (xem 7.8.).

▷ **2.54.** *Đa thức*
Để viết chương trình làm việc với đa thức với hệ số thực và nghiệm nguyên có dạng:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

được trình bày thông qua một danh sách liên kết. Mô-đun để làm việc với đa thức phải bao gồm các phép toán cộng, trừ, nhân, chia đa thức, cũng như tìm giá trị của đa thức cho một giá trị nhất định của biến chưa biết x[Nakov-1998].

▷ **2.55.** *Sắp xếp theo cây*
Trong 2.3. chúng ta đã chỉ ra cách có thể lấy được các khóa của cây tìm kiếm nhị phân, được sắp xếp theo thứ tự tăng dần. Điều gì cần thay đổi khi thu thập dữ liệu để chuỗi khóa được sắp xếp theo thứ tự giảm dần?

▷ **2.56.** *Số lượng*
Viết chương trình tìm tổng các mức ở tất cả các đỉnh trong cây nhị phân.

▷ **2.57.** *Gần như một chiếc lá*
Viết chương trình in tất cả các ngọn của cây nhị phân chỉ có các lá là cây thừa kế.

▷ **2.58.** *Phục hồi cây*
Kết quả của hai lần thu thập thông tin cây nhị phân giữa {LCD, LDC và CLD} được đưa ra. Để tìm lần thu thập thông tin thứ ba [Nakov-1998].

▷ **2.59.** *Xây dựng một cái cây cân bằng hoàn hảo*
Viết chương trình xây dựng một cây cân bằng hoàn hảo trên một cây chứa các khóa giống nhau.

Gợi ý: Một danh sách tuyến tính có thể được xây dựng trong đó các phần tử tham gia theo thứ tự tăng dần, và sau đó được sử dụng để xây dựng một cây cân bằng hoàn hảo [Nakov-1998].

▷ **2.60.** *Kiểm tra cây cân đối hoàn hảo*
Viết hàm kiểm tra xem cây tìm kiếm nhị phân có cân bằng hoàn hảo hay không.

▷ **2.61.** *Tìm kiếm trong không gian*
Một cây tìm kiếm nhị phân và hai số nguyên t_1 và t_2 được đưa ra. Tìm tất cả các đỉnh x của cây có khóa nằm trong khoảng $[t_1, t_2]$, tức là $t_1 \leq key[x] \leq t_2$.