MỤC LỤC

Phần 1 – PHẦN MỞ ĐẦU

Chương 1 – GIỚI THIỆU

1.1. Về phương pháp phân tích thiết kế hướng đối tượng....1
1.2. Giới thiệu môn học Cấu trúc dữ liệu (CTDL) và giải thuật....1
1.3. Cách tiếp cận trong quá trình tìm hiểu các lớp CTDL....4
1.3.1. Các bước trong quá trình phân tích thiết kế hướng đối tượng....4
1.3.2. Quá trình xây dựng các lớp CTDL....5
1.4. Một số định nghĩa cơ bản....6
1.4.1. Định nghĩa kiểu dữ liệu....6
1.4.2. Kiểu nguyên tố và các kiểu có cấu trúc....6
1.4.3. Chuỗi nối tiếp và danh sách....6
1.4.4. Các kiểu dữ liệu trừu tượng....7
1.5. Một số nguyên tắc và phương pháp để học tốt môn CTDL và giải thuật....8
1.5.1. Cách tiếp cận và phương hướng suy nghĩ tích cực....8
1.5.2. Các nguyên tắc....9
1.5.3. Phong cách lập trình (*style of programming*) và các kỹ năng:....10
1.6. Giới thiệu về ngôn ngữ giả:....14

Phần 2 – CÁC CẤU TRÚC DỮ LIỆU

Chương 2 – NGĂN XẾP

2.1. Định nghĩa ngăn xếp....17
2.2. Đặc tả ngăn xếp....18
2.3. Các phương án hiện thực ngăn xếp....22
2.4. Hiện thực ngăn xếp....22
2.4.1. Hiện thực ngăn xếp liên tục....22
2.4.2. Hiện thực ngăn xếp liên kết....25
2.4.3. Ngăn xếp liên kết với sự an toàn....29
2.4.4. Đặc tả ngăn xếp liên kết đã hiệu chỉnh....34

Chương 3 – HÀNG ĐỢI

3.1. Định nghĩa hàng 37
3.2. Đặc tả hàng 38
3.3. Các phương án hiện thực hàng 41
3.3.1. Các phương án hiện thực hàng liên tục 41
3.3.2. Phương án hiện thực hàng liên kết 45
3.4. Hiện thực hàng 46
3.4.1. Hiện thực hàng liên tục 46
3.4.2. Hiện thực hàng liên kết 48
3.4.3. Hàng liên kết mở rộng 50

Chương 4 – DANH SÁCH

4.1. Định nghĩa danh sách 51
4.2. Đặc tả các phương thức cho danh sách 51
4.3. Hiện thực danh sách 54
4.3.1. Hiện thực danh sách liên tục 54
4.3.2. Hiện thực danh sách liên kết đơn giản 56
4.3.3. Lưu lại vị trí hiện tại 61
4.3.4. Danh sách liên kết kép 63
4.4. So sánh các cách hiện thực của danh sách 66
4.5. Danh sách liên kết trong mảng liên tục 67
4.5.1. Phương pháp 67
4.5.2. Các tác vụ quản lý vùng nhớ 70
4.5.3. Các tác vụ khác 73
4.5.4. Các biến thể của danh sách liên kết trong mảng liên tục 74

Chương 5 – CHUỖI KÝ TỰ

5.1. Chuỗi ký tự trong C và trong C++ 75
5.2. Đặc tả của lớp String 77
5.2.1. Các phép so sánh 77
5.2.2. Một số *constructor* tiện dụng 77
5.3. Hiện thực lớp String 79
5.4. Các tác vụ trên String 81
5.5. Các giải thuật tìm một chuỗi con trong một chuỗi 83
5.5.1. Giải thuật Brute-Force 83
5.5.2. Giải thuật Knuth-Morris-Pratt 85

Chương 6 – ĐỆ QUY

6.1. Giới thiệu về đệ quy 91
6.1.1. Cơ cấu ngăn xếp cho các lần gọi hàm 91
6.1.2. Cây biểu diễn các lần gọi hàm 92
6.1.3. Giai thừa: Một định nghĩa đệ quy 93
6.1.4. Chia để trị: Bài toán Tháp Hà Nội 95
6.2. Các nguyên tắc của đệ quy 100
6.2.1. Thiết kế giải thuật đệ quy 100
6.2.2. Cách thực hiện của đệ quy 102
6.2.3. Đệ quy đuôi 104
6.2.4. Phân tích một số trường hợp nên và không nên dùng đệ quy 106
6.2.5. Các nhận xét 110
6.3. Phương pháp quay lui (*backtracking*) 112
6.3.1. Lời giải cho bài toán tám con hậu 112
6.3.2. Ví dụ với bốn con Hậu 114
6.3.3. Phương pháp quay lui (*Backtracking*) 115
6.3.4. Phác thảo chung cho chương trình đặt các con hậu lên bàn cờ 115
6.3.5. Tinh chế: Cấu trúc dữ liệu đầu tiên và các phương thức 118
6.3.6. Xem xét lại và tinh chế 120
6.3.7. Phân tích về phương pháp quay lui 124
6.4. Các chương trình có cấu trúc cây: dự đoán trước trong các trò chơi 127
6.4.1. Các cây trò chơi 127
6.4.2. Phương pháp Minimax 128
6.4.3. Phát triển giải thuật 130
6.4.4. Tinh chế 131
6.4.5. Tic-Tac-Toe 132

Chương 7 – TÌM KIẾM

7.1. Giới thiệu 137
7.1.1. Khóa 137
7.1.2. Phân tích 137
7.1.3. Tìm kiếm nội và tìm kiếm ngoại 137
7.1.4. Lớp Record và lớp Key 138
7.1.5. Thông số 139
7.2. Tìm kiếm tuần tự 139

7.2.1. Giải thuật và hàm ... 139
7.2.2. Phân tích ... 140
7.3. Tìm kiếm nhị phân ... 141
7.3.1. Danh sách có thứ tự ... 142
7.3.2. Xây dựng giải thuật ... 143
7.3.3. Phiên bản thứ nhất ... 143
7.3.4. Nhận biết sớm phần tử có chứa khóa đích ... 145
7.4. Cây so sánh ... 147

Chương 8 – SẮP XẾP

8.1. Giới thiệu ... 149
8.2. Sắp xếp kiểu chèn (*Insertion Sort*) ... 150
8.2.1. Chèn phần tử vào danh sách đã có thứ tự ... 150
8.2.2. Sắp xếp kiểu chèn cho danh sách liên tục ... 151
8.2.3. Sắp xếp kiểu chèn cho danh sách liên kết ... 153
8.3. Sắp xếp kiểu chọn (*Selection Sort*) ... 155
8.3.1. Giải thuật ... 155
8.3.2. Sắp xếp chọn trên danh sách liên tục ... 156
8.4. Shell_sort ... 158
8.5. Các phương pháp sắp xếp theo kiểu chia để trị ... 160
8.5.1. Ý tưởng cơ bản ... 160
8.5.2. Ví dụ ... 161
8.6. Merge_sort cho danh sách liên kết ... 164
8.7. Quick_sort cho danh sách liên tục ... 167
8.7.1. Các hàm ... 167
8.7.2. Phân hoạch danh sách ... 168
8.8. Heap và Heap_sort ... 170
8.8.1. Định nghĩa heap nhị phân ... 171
8.8.2. Phát triển giải thuật Heap_sort ... 172
8.9. Radix Sort ... 176
8.9.1. Ý tưởng ... 177
8.9.2. Hiện thực ... 177
8.9.3. Phân tích phương pháp radix_sort ... 181

Chương 9 – CÂY NHỊ PHÂN

9.1. Các khái niệm cơ bản về cây ... 183

9.2. Cây nhị phân 185
9.2.1. Các định nghĩa 185
9.2.2. Duyệt cây nhị phân 187
9.2.3. Hiện thực liên kết của cây nhị phân 193
9.3. Cây nhị phân tìm kiếm 197
9.3.1. Các danh sách có thứ tự và các cách hiện thực 198
9.3.2. Tìm kiếm trên cây 199
9.3.3. Thêm phần tử vào cây nhị phân tìm kiếm 203
9.3.4. Sắp thứ tự theo cây 206
9.3.5. Loại phần tử trong cây nhị phân tìm kiếm 207
9.4. Xây dựng một cây nhị phân tìm kiếm 210
9.4.1. Thiết kế giải thuật 212
9.4.2. Các khai báo và hàm main 213
9.4.3. Thêm một nút 214
9.4.4. Hoàn tất công việc 215
9.4.5. Đánh giá 217
9.5. Cân bằng chiều cao: Cây AVL 218
9.5.1. Định nghĩa 218
9.5.2. Thêm một nút 222
9.5.3. Loại một nút 230
9.5.4. Chiều cao của cây AVL 234

Chương 10 – CÂY NHIỀU NHÁNH

10.1. Vườn cây, cây, và cây nhị phân 237
10.1.1. Các tên gọi cho cây 237
10.1.2. Cây có thứ tự 239
10.1.3. Rừng và vườn 241
10.1.4. Sự tương ứng hình thức 243
10.1.5. Phép quay 244
10.1.6. Tổng kết 244
10.2. Cây từ điển tìm kiếm: Trie 245
10.2.1. Tries 245
10.2.2. Tìm kiếm một khóa 245
10.2.3. Giải thuật C++ 246
10.2.4. Tìm kiếm trong cây Trie 247
10.2.5. Thêm phần tử vào Trie 247
10.2.6. Loại phần tử trong Trie 248

10.2.7. Truy xuất Trie 248
10.3. Tìm kiếm ngoài: B-tree 249
10.3.1. Thời gian truy xuất 249
10.3.2. Cây tìm kiếm nhiều nhánh 250
10.3.3. Cây nhiều nhánh cân bằng 250
10.3.4. Thêm phần tử vào B-tree 251
10.3.5. Giải thuật C++: tìm kiếm và thêm vào 253
10.3.6. Loại phần tử trong B-tree 263
10.4. Cây đỏ-đen 271
10.4.1. Dẫn nhập 271
10.4.2. Định nghĩa và phân tích 272
10.4.3. Đặc tả cây đỏ đen 274
10.4.4. Thêm phần tử 276
10.4.5. Phương thức thêm vào. Hiện thực 279
10.4.6. Loại một nút 282

Chương 11 – HÀNG ƯU TIÊN

11.1. Định nghĩa hàng ưu tiên 283
11.2. Các phương án hiện thực hàng ưu tiên 283
11.3. Hiện thực các tác vụ cơ bản trên heap nhị phân 284
11.3.1. Tác vụ thêm phần tử 284
11.3.2. Tác vụ loại phần tử 286
11.4. Các tác vụ khác trên heap nhị phân 287
11.4.1. Tác vụ tìm phần tử lớn nhất 287
11.4.2. Tác vụ tăng giảm độ ưu tiên 287
11.4.3. Tác vụ loại một phần tử không ở đầu hàng 288
11.5. Một số phương án khác của heap 288
11.5.1. d-heaps 288
11.5.2. Heap lệch trái (*Leftist heap*) 289
11.5.3. Skew heap 295
11.5.4. Hàng nhị thức (*Binomial Queue*) 295

Chương 12 – BẢNG VÀ TRUY XUẤT THÔNG TIN

12.1. Dẫn nhập: phá vỡ rào cản lgn 305
12.2. Các bảng chữ nhật 306
12.2.1. Thứ tự ưu tiên hàng và thứ tự ưu tiên cột 306

12.2.2. Đánh chỉ số cho bảng chữ nhật 307
12.2.3. Biến thể: mảng truy xuất 308
12.3. Các bảng với nhiều hình dạng khác nhau 308
12.3.1. Các bảng tam giác 309
12.3.2. Các bảng lồi lõm 310
12.3.3. Các bảng chuyển đổi 311
12.4. Bảng: Một kiểu dữ liệu trừu tượng mới 313
12.4.1. Các hàm 313
12.4.2. Một kiểu dữ liệu trừu tượng 314
12.4.3. Hiện thực 315
12.4.4. So sánh giữa danh sách và bảng 315
12.5. Bảng băm 317
12.5.1. Các bảng thưa 317
12.5.2. Lựa chọn hàm băm 318
12.5.3. Phác thảo giải thuật cho các thao tác dữ liệu trong bảng băm 321
12.5.4. Ví dụ trong C++ 322
12.5.5. Giải quyết đụng độ bằng phương pháp địa chỉ mở 323
12.5.6. Giải quyết đụng độ bằng phương pháp nối kết 323
12.6. Phân tích bảng băm 331
12.6.1. Điều ngạc nhiên về ngày sinh 331
12.6.2. Đếm số lần thử 332
12.6.3. Phân tích phương pháp nối kết 332
12.6.4. Phân tích phương pháp địa chỉ mở 333
12.6.5. Các so sánh lý thuyết 334
12.6.6. Các so sánh thực nghiệm 335
12.7. Kết luận: so sánh các phương pháp 336

Chương 13 – ĐỒ THỊ

13.1. Nền tảng toán học 339
13.1.1. Các định nghĩa và ví dụ 339
13.1.2. Đồ thị vô hướng 340
13.1.3. Đồ thị có hướng 341
13.2. Biểu diễn bằng máy tính 341
13.2.1. Biểu diễn của tập hợp 342
13.2.2. Danh sách kề 344
13.2.3. Các thông tin khác trong đồ thị 346
13.3. Duyệt đồ thị 346

13.3.1. Các phương pháp ... 346
13.3.2. Giải thuật duyệt theo chiều sâu ... 347
13.3.3. Giải thuật duyệt theo chiều rộng ... 348
13.4. Sắp thứ tự topo ... 349
13.4.1. Đặt vấn đề ... 349
13.4.2. Giải thuật duyệt theo chiều sâu ... 350
13.4.3. Giải thuật duyệt theo chiều rộng ... 352
13.5. Giải thuật Greedy: Tìm đường đi ngắn nhất ... 353
13.5.1. Đặt vấn đề ... 353
13.5.2. Phương pháp ... 354
13.5.3. Ví dụ ... 356
13.5.4. Hiện thực ... 356
13.6. Cây phủ tối tiểu ... 357
13.6.1. Đặt vấn đề ... 357
13.6.2. Phương pháp ... 359
13.6.3. Hiện thực ... 361
13.6.4. Kiểm tra giải thuật Prim ... 362
13.7. Sử dụng đồ thị như là cấu trúc dữ liệu ... 364

Phần 3 – CÁC ỨNG DỤNG CỦA CÁC LỚP CTDL

Chương 14 – ỨNG DỤNG CỦA NGĂN XẾP

14.1. Đảo ngược dữ liệu ... 365
14.2. Phân tích biên dịch (parsing) dữ liệu ... 366
14.3. Trì hoãn công việc ... 368
14.3.1. Ứng dụng tính trị của biểu thức *postfix* ... 368
14.3.2. Ứng dụng chuyển đổi biểu thức dạng *infix* thành dạng *postfix* ... 371
14.4. Giải thuật quay lui (*backtracking*) ... 372
14.4.1. Ứng dụng trong bài toán tìm đích (*goal seeking*). ... 372
14.4.2. Bài toán mã đi tuần và bài toán tám con hậu ... 375

Chương 15 – ỨNG DỤNG CỦA HÀNG ĐỢI

15.1. Các dịch vụ ... 377
15.2. Phân loại ... 377
15.3. Phương pháp sắp thứ tự Radix Sort ... 377

15.4. Tính trị cho biểu thức *prefix* 378
15.5. Ứng dụng phép tính trên đa thức 378
15.5.1. Mục đích của ứng dụng 378
15.5.2. Chương trình 378
15.5.3. Cấu trúc dữ liệu của đa thức 381
15.5.4. Đọc và ghi các đa thức 384
15.5.5. Phép cộng đa thức 385
15.5.6. Hoàn tất chương trình 386

***Chương 16* – ỨNG DỤNG XỬ LÝ VĂN BẢN**

16.1. Các đặc tả 387
16.2. Hiện thực 388
16.2.1. Chương trình chính 388
16.2.2. Đặc tả lớp Editor 389
16.2.3. Nhận lệnh từ người sử dụng 390
16.2.4. Thực hiện lệnh 390
16.2.5. Đọc và ghi tập tin 392
16.2.6. Chèn một hàng 393
16.2.7. Tìm một chuỗi ký tự 393
16.2.8. Biến đổi chuỗi ký tự 394

***Chương 17* – ỨNG DỤNG SINH CÁC HOÁN VỊ**

17.1. Ý tưởng 395
17.2. Tinh chế 396
17.3. Thủ tục chung 396
17.4. Tối ưu hóa cấu trúc dữ liệu để tăng tốc độ cho chương trình sinh các hoán vị 397
17.5. Chương trình 398

***Chương 18* – ỨNG DỤNG DANH SÁCH LIÊN KẾT VÀ BẢNG BĂM**

18.1. Giới thiệu về chương trình Game_Of_Life 401
18.2. Các ví dụ 401
18.3. Giải thuật 402
18.4. Chương trình chính cho Game_Of_Life 403

18.4.1. Phiên bản thứ nhất cho lớp Life 404
18.4.2. Phiên bản thứ hai với CTDL mới cho Life 407

Phần 1 – PHẦN MỞ ĐẦU

Chương 1 – GIỚI THIỆU

1.1. Về phương pháp phân tích thiết kế hướng đối tượng

Thông thường phần quan trọng nhất của quá trình phân tích thiết kế là chia vấn đề thành nhiều vấn đề nhỏ dễ hiểu và chi tiết hơn. Nếu chúng vẫn còn khó hiểu, chúng lại được chia nhỏ hơn nữa. Trong bất kỳ một tổ chức nào, người quản lý cao nhất cũng không thể quan tâm đến mọi chi tiết cũng như mọi hoạt động. Họ cần tập trung vào mục tiêu và các nhiệm vụ chính, họ chia bớt trách nhiệm cho những người cộng sự dưới quyền của họ. Việc lập trình trong máy tính cũng tương tự. Ngay cả khi dự án đủ nhỏ cho một người thực hiện từ đầu tới cuối, việc chia nhỏ công việc cũng rất quan trọng. Phương pháp phân tích thiết kế hướng đối tượng dựa trên quan điểm này. Cái khó nhất là định ra các lớp sao cho mỗi lớp sau này sẽ cung cấp các đối tượng có các hành vi đúng như chúng ta mong đợi. Việc lập trình giải quyết bài toán lớn của chúng ta sẽ được tập trung vào những giải thuật lớn. Chương trình khi đó được xem như một kịch bản, trong đó các đối tượng sẽ được gọi để thực hiện các hành vi của mình vào những lúc cần thiết. Chúng ta không còn phải lo bị mất phương hướng vì những chi tiết vụn vặt khi cần phải phác thảo một kịch bản đúng đắn, một khi chúng ta đã tin tưởng hoàn toàn vào khả năng hoàn thành nhiệm vụ của các lớp mà chúng ta đã giao phó.

Các lớp do người lập trình định nghĩa đóng vai trò trung tâm trong việc hiện thực giải thuật.

1.2. Giới thiệu môn học Cấu trúc dữ liệu (CTDL) và giải thuật

Theo quan điểm của phân tích thiết kế hướng đối tượng, mỗi lớp sẽ được xây dựng với một số chức năng nào đó và các đối tượng của nó sẽ tham gia vào hoạt động của chương trình. Điểm mạnh của hướng đối tượng là tính đóng kín và tính sử dụng lại của các lớp. Mỗi phần mềm biên dịch cho một ngôn ngữ lập trình nào đó đều chứa rất nhiều thư viện các lớp như vậy. Chúng ta thử điểm qua một số lớp mà người lập trình thường hay sử dụng: các lớp có nhiệm vụ đọc/ ghi để trao đổi dữ liệu với các thiết bị ngoại vi như đĩa, máy in, bàn phím,...; các lớp đồ họa cung cấp các chức năng vẽ, tô màu cơ bản; các lớp điều khiển cho phép xử lý việc giao tiếp với người sử dụng thông qua bàn phím, chuột, màn hình; các lớp phục vụ các giao dịch truyền nhận thông tin qua mạng;...Các lớp CTDL mà chúng ta sắp bàn đến cũng không là một trường hợp ngoại lệ. Có thể chia tất cả các lớp này thành hai nhóm chính:

- Các lớp dịch vụ.
- Các lớp có khả năng lưu trữ và xử lý lượng dữ liệu lớn.

Nhóm thứ hai muốn nói đến các lớp CTDL (CTDL). Vậy có gì giống và khác nhau giữa các lớp CTDL và các lớp khác?

- Điểm giống nhau giữa các lớp CTDL và các lớp khác: mỗi lớp đều phải thực hiện một số chức năng thông qua các hành vi của các đối tượng của nó. Một khi chúng ta đã xây dựng xong một lớp CTDL nào đó, chúng ta hoàn toàn tin tưởng rằng nó sẽ hoàn thành xuất sắc những nhiệm vụ mà chúng ta đã thiết kế và đã giao phó cho nó. Điều này rất khác biệt so với những tài liệu viết về CTDL theo quan điểm hướng thủ tục trước đây: việc xử lý dữ liệu không hề có tính đóng kín và tính sử dụng lại. Tuy về mặt thực thi thì các chương trình như thế có khả năng chạy nhanh hơn, nhưng chúng bộc lộ rất nhiều nhược điểm: thời gian phát triển giải thuật chính rất chậm gây khó khăn nhiều cho người lập trình, chương trình thiếu tính trong sáng, rất khó sửa lỗi và phát triển.
- Đặc trưng riêng của các lớp CTDL: Nhiệm vụ chính của các lớp CTDL là nắm giữ dữ liệu sao cho có thể đáp ứng mỗi khi được chương trình yêu cầu trả về một dữ liệu cụ thể nào đó mà chương trình cần đến. Những thao tác cơ bản đối với một CTDL thường là: thêm dữ liệu mới, xóa bỏ dữ liệu đã có, tìm kiếm, truy xuất.

Ngoài các thao tác dữ liệu cơ bản, các CTDL khác nhau sẽ khác nhau về các thao tác bổ sung khác. Chính vì điều này mà khi thiết kế những giải thuật để giải quyết các bài toán lớn, người ta sẽ lựa chọn CTDL nào là thích hợp nhất.

Chúng ta thử xem xét một ví dụ thật đơn giản sau đây.

Giả sử chúng ta cần viết một chương trình nhận vào một dãy các con số, và in chúng ra theo thứ tự ngược với thứ tự nhập vào ban đầu.

Để giải quyết bài toán này, nếu chúng ta nghĩ đến việc phải khai báo các biến để lưu các giá trị nhập vào như thế nào, và sau đó là thứ tự in ra sao để đáp ứng yêu cầu bài toán, thì dường như là chúng ta đã quên áp dụng nguyên tắc lập trình hướng đối tượng: chúng ta đã phải bận tâm đến những việc quá chi tiết. Đây chỉ là một ví dụ vô cùng đơn giản, nhưng nó có thể minh họa cho vai trò của CTDL. Nếu chúng ta nhớ rằng, việc tổ chức và lưu dữ liệu như thế nào là một việc quá chi tiết và tỉ mỉ không nên thực hiện vào lúc này, thì đó chính là lúc chúng ta đã bước đầu hiểu được vai trò của các lớp CTDL.

Môn CTDL và giải thuật sẽ giúp chúng ta hiểu rõ về các lớp CTDL có sẵn trong các phần mềm. Hơn thế nữa, trong khi học cách xây dựng các lớp CTDL từ đơn giản đến phức tạp, chúng ta sẽ nắm được các phương pháp cũng như các kỹ năng thông qua một số nguyên tắc chung. Từ đó, ngoài khả năng hiểu rõ để có thể lựa chọn một cách đúng đắn nhất những CTDL có sẵn, chúng ta còn có khả năng xây dựng những lớp CTDL phức tạp hơn, tinh tế và thích hợp hơn trong mỗi bài toán mà chúng ta cần giải quyết. Khả năng thừa kế các CTDL có sẵn để phát triển thêm các tính năng mong muốn cũng là một điều đáng lưu ý.

Với ví dụ trên, những ai đã từng tiếp xúc ít nhiều với việc lập trình đều không xa lạ với khái niệm "ngăn xếp". Đây là một CTDL đơn giản nhất nhưng lại rất thông dụng, và dĩ nhiên chúng ta sẽ có dịp học kỹ hơn về nó. Ở đây chúng ta muốn mượn nó để minh họa, và cũng nhằm giúp cho người đọc làm quen với một phương pháp tiếp cận hoàn toàn nhất quán trong suốt giáo trình này.
Giả sử CTDL ngăn xếp của chúng ta đã được giao cho một nhiệm vụ là cất giữ những dữ liệu và trả về khi có yêu cầu, theo một quy định bất di bất dịch là dữ liệu đưa vào sau phải được lấy ra trước. Bằng cách sử dụng CTDL ngăn xếp, chương trình trở nên hết sức đơn giản và được trình bày bằng ngôn ngữ giả như sau:

```
Lặp cho đến khi nhập đủ các con số mong muốn
{
  Nhập 1 con số.
  Cất vào ngăn xếp con số vừa nhập.
}
Lặp trong khi mà ngăn xếp vẫn còn dữ liệu
{
  Lấy từ ngăn xếp ra một con số.
  In số vừa lấy được.
}
```

Chúng ta sẽ có dịp gặp nhiều bài toán phức tạp hơn mà cũng cần sử dụng đến đặc tính này của ngăn xếp. Tính đóng kín của các lớp giúp cho chương trình vô cùng trong sáng. Đoạn chương trình trên không hề cho chúng ta thấy ngăn xếp đã làm việc với các dữ liệu được đưa vào như thế nào, đó là nhiệm vụ mà chúng ta đã giao phó cho nó và chúng ta hoàn toàn yên tâm về điều này. Bằng cách này, khi đã có những CTDL thích hợp, người lập trình có thể dễ dàng giải quyết các bài toán lớn. Họ có thể yên tâm tập trung vào những điểm mấu chốt để xây dựng, tinh chế giải thuật và kiểm lỗi.

Trên đây chúng ta chỉ vừa mới giới thiệu về phần CTDL nằm trong nội dung của môn học "CTDL và giải thuật". Vậy giải thuật là gì? Đứng trên quan điểm thiết kế và lập trình hướng đối tượng, chúng ta đã hiểu vai trò của các lớp. Vậy khi đã có các lớp rồi thì người ta cần xây dựng kịch bản cho các đối tượng hoạt động nhằm giải quyết bài toán chính. Chúng ta cần một cấu trúc chương trình để tạo ra kịch bản đó: việc gì làm trước, việc gì làm sau; việc gì chỉ làm trong những tình huống đặc biệt nào đó; việc gì cần làm lặp lại nhiều lần. Chúng ta nhắc đến giải thuật chính là quay về với khái niệm của "lập trình thủ tục" trước kia. Ngoài ra, chúng ta cũng cần đến giải thuật khi cần hiện thực cho mỗi lớp: xong phần đặc tả các phương thức - phương tiện giao tiếp của lớp với bên ngoài - chúng ta cần đến khái niệm "lập trình thủ tục" để giải quyết phần hiện thực bên trong của

các phương thức này. Đó là việc chúng ta phải xử lý những dữ liệu bên trong của chúng như thế nào mới có thể hoàn thành được chức năng mà phương thức phải đảm nhiệm.

Như vậy, về phần giải thuật trong môn học này, chủ yếu chúng ta sẽ tìm hiểu các giải thuật mà các phương thức của các lớp CTDL dùng đến, một số giải thuật sắp xếp tìm kiếm, và các giải thuật trong các ứng dụng minh họa việc sử dụng các lớp CTDL để giải quyết một số bài toán đó.

Trong giáo trình này, ý tưởng về các giải thuật sẽ được trình bày cặn kẽ, phần chương trình dùng ngôn ngữ C++ hoặc ngôn ngữ giả theo quy ước ở cuối chương này. Phần đánh giá giải thuật chỉ nêu những kết quả đã được chứng minh và kiểm nghiệm, sinh viên có thể tìm hiểu kỹ hơn trong các sách tham khảo.

1.3. Cách tiếp cận trong quá trình tìm hiểu các lớp CTDL

1.3.1. Các bước trong quá trình phân tích thiết kế hướng đối tượng

Quá trình phân tích thiết kế hướng đối tượng khi giải quyết một bài toán gồm các bước như sau:

1. Định ra các lớp với các chức năng mà chúng ta mong đợi. Công việc này cũng giống như công việc phân công công việc cho các nhân viên cùng tham gia một dự án.
2. Giải quyết bài toán bằng cách lựa chọn các giải thuật chính. Đó là việc tạo ra một môi trường để các đối tượng của các lớp nêu trên tương tác lẫn nhau. Giải thuật chính được xem như một kịch bản dẫn dắt các đối tượng thực hiện các hành vi của chúng vào những thời điểm cần thiết.
3. Hiện thực cho mỗi lớp.

Ý tưởng chính ở đây nằm ở bước thứ hai, dẫu cho các lớp chưa được hiện thực, chúng ta hoàn toàn có thể sử dụng chúng sau khi đã biết rõ những chức năng mà mỗi lớp sẽ phải hoàn thành. Trung thành với quan điểm này của hướng đối tượng, chúng ta cũng sẽ nêu ra đây phương pháp tiếp cận mà chúng ta sẽ sử dụng một cách hoàn toàn nhất quán trong việc nghiên cứu và xây dựng các lớp CTDL.

Ứng dụng trong chương 18 về chương trình Game Of Life là một dẫn chứng về các bước phân tích thiết kế trong quá trình xây dựng nên một chương trình. Sinh viên có thể tham khảo ngay phần này. Riêng phần 18.4.2 phiên bản thứ hai của chương trình sinh viên chỉ có thể tham khảo sau khi đọc qua chương 4 về danh sách và chương 12 về bảng băm.

1.3.2. Quá trình xây dựng các lớp CTDL

Chúng ta sẽ lần lượt xây dựng từ các lớp CTDL đơn giản cho đến các lớp CTDL phức tạp hơn. Tuy nhiên, quá trình thiết kế và hiện thực cho mọi lớp CTDL đều tuân theo đúng các bước sau đây:

1. Xuất phát từ một mô hình toán học hay dựa vào một nhu cầu thực tế nào đó, chúng ta định ra các chức năng của lớp CTDL chúng ta cần có. Bước này giống bước thứ nhất ở trên, vì lớp CTDL, cũng như các lớp khác, sẽ cung cấp cho chúng ta các đối tượng để hoạt động trong chương trình chính. Và như vậy, những nhiệm vụ mà chúng ta sẽ giao cho nó sẽ được chỉ ra một cách rõ ràng và chính xác ở bước kế tiếp sau đây.

2. Đặc tả đầy đủ cách thức giao tiếp giữa lớp CTDL đang được thiết kế với môi trường ngoài (các chương trình sẽ sử dụng nó). Phần giao tiếp này được mô tả thông qua định nghĩa các phương thức của lớp. Mỗi phương thức là một hành vi của đối tượng CTDL sau này, phần đặc tả gồm các yếu tố sau:
   - Kiểu của kết quả mà phương thức trả về.
   - Các thông số vào / ra.
   - Các điều kiện ban đầu trước khi phương thức được gọi (*precondition*).
   - Các kết quả mà phương thức làm được (*postcondition*).
   - Các lớp, hàm có sử dụng trong phương thức (*uses*).

   Thông qua phần đặc tả này, các CTDL hoàn toàn có thể được sử dụng trong việc xây dựng nên những giải thuật lớn trong các bài toán lớn. Phần đặc tả này có thể được xem như những cam kết mà không bao giờ được quyền thay đổi. Có như vậy các chương trình có sử dụng CTDL mới không bị thay đổi một khi đã sử dụng chúng.

3. Tìm hiểu các phương án hiện thực cho lớp CTDL. Chúng ta cũng nên nhớ rằng, có rất nhiều cách hiện thực khác nhau cho cùng một đặc tả của một lớp CTDL. Về mặt hiệu quả, có những phương án gần như giống nhau, nhưng cũng có những phương án khác nhau rất xa. Điều này phụ thuộc rất nhiều vào cách tổ chức dữ liệu bên trong bản thân của lớp CTDL, vào các giải thuật liên quan đến việc xử lý dữ liệu của các phương thức.

4. Chọn phương án và hiện thực lớp. Trong bước này bao gồm cả việc kiểm tra để hoàn tất lớp CTDL như là một lớp để bổ sung vào thư viện, người lập trình có thể sử dụng chúng trong nhiều chương trình sau này. Công việc ở bước này kể cũng khá vất vả, vì chúng ta sẽ phải kiểm tra thật kỹ lưỡng, trước khi đưa sản phẩm ra như những đóng gói, mà người khác có thể hoàn toàn yên tâm khi sử dụng chúng.

Để có được những sản phẩm hoàn hảo thực hiện đúng những điều đã cam kết, bước thứ hai trên đây được xem là bước quan trọng nhất. Và để có được một định nghĩa và một đặc tả đầy đủ và chính xác nhất cho một CTDL mới nào đó, bước thứ hai phải được thực hiện hoàn toàn độc lập với hai bước sau nó. Đây là nguyên tắc vô cùng quan trọng mà chúng ta sẽ phải tuân thủ một cách triệt để. Vì trong trường hợp ngược lại, việc xem xét sớm các chi tiết cụ thể sẽ làm cho chúng ta dễ có cái nhìn phiến diện, điều này dễ dẫn đến những đặc tả mang nhiều sơ suất.

1.4. Một số định nghĩa cơ bản

Chúng ta bắt đầu bằng định nghĩa của một kiểu dữ liệu (*type*):

1.4.1. Định nghĩa kiểu dữ liệu

Định nghĩa: Một kiểu dữ liệu là một tập hợp, các phần tử của tập hợp này được gọi là các trị của kiểu dữ liệu.

Chúng ta có thể gọi một kiểu số nguyên là một tập các số nguyên, kiểu số thực là một tập các số thực, hoặc kiểu ký tự là một tập các ký hiệu mà chúng ta mong muốn sử dụng trong các giải thuật của chúng ta.

Lưu ý rằng chúng ta đã có thể chỉ ra sự phân biệt giữa một kiểu dữ liệu trừu tượng và cách hiện thực của nó. Chẳng hạn, kiểu int trong C++ không phải là tập của tất cả các số nguyên, nó chỉ chứa các số nguyên được biểu diễn thực sự bởi một máy tính xác định, số nguyên lớn nhất trong tập phụ thuộc vào số bit người ta dành để biểu diễn nó (thường là một từ gồm 2 bytes tức 16 bits). Tương tự, kiểu float và `double` trong C++ biểu diễn một tập các số thực có dấu chấm động nào đó, và đó chỉ là một tập con của tập tất cả các số thực.

1.4.2. Kiểu nguyên tố và các kiểu có cấu trúc

Các kiểu như `int, float, char` được gọi là các kiểu nguyên tố (*atomic*) vì chúng ta xem các trị của chúng chỉ là một thực thể đơn, chúng ta không có mong muốn chia nhỏ chúng. Tuy nhiên, các ngôn ngữ máy tính thường cung cấp các công cụ cho phép chúng ta xây dựng các kiểu dữ liệu mới gọi là các kiểu có cấu trúc (*structured types*). Chẳng hạn như một struct trong C++ có thể chứa nhiều kiểu nguyên tố khác nhau, trong đó không loại trừ một kiểu có cấu trúc khác làm thành phần. Trị của một kiểu có cấu trúc cho chúng ta biết nó được tạo ra bởi các phần tử nào.

1.4.3. Chuỗi nối tiếp và danh sách

Định nghĩa: Một chuỗi nối tiếp (*sequence*) kích thước 0 là một chuỗi rỗng. Một chuỗi nối tiếp kích thước $n \geq 1$ các phần tử của tập T là một cặp có thứ tự (Sn-1, t), trong đó Sn-1 là một chuỗi nối tiếp kích thước $n - 1$ các phần tử của tập T, và t là một phần tử của tập T.

Từ định nghĩa này chúng ta có thể tạo nên một chuỗi nối tiếp dài tùy ý, bắt đầu từ một chuỗi nối tiếp rỗng và thêm mỗi lần một phần tử của tập T.

Chúng ta phân biệt hai từ: nối tiếp (*sequential*) ngụ ý các phần tử thuộc một chuỗi nối tiếp về mặt luận lý, còn từ liên tục (*contiguous*) ngụ ý các phần tử nằm kề nhau trong bộ nhớ. Trong định nghĩa trên đây chúng ta chỉ dùng từ nối tiếp mà thôi, chúng ta chưa hề quan tâm về mặt vật lý.

Từ định nghĩa chuỗi nối tiếp hữu hạn cho phép chúng ta định nghĩa một danh sách (*list*):

Định nghĩa: Một danh sách các phần tử thuộc kiểu T là một chuỗi nối tiếp hữu hạn các phần tử kiểu T.

1.4.4. Các kiểu dữ liệu trừu tượng

Định nghĩa: CTDL (*Data Structure*) là một sự kết hợp của các kiểu dữ liệu nguyên tố, và/ hoặc các kiểu dữ liệu có cấu trúc, và/ hoặc các CTDL khác vào một tập, cùng các quy tắc về các mối quan hệ giữa chúng.

Trong định nghĩa này, cấu trúc có nghĩa là tập các quy tắc kết nối các dữ liệu với nhau. Mặt khác, đứng trên quan điểm của hướng đối tượng, chúng ta sẽ xây dựng mỗi CTDL như là một lớp mà ngoài khả năng chứa dữ liệu, nó còn có các hành vi đặc trưng riêng, đó chính là các thao tác cho phép cập nhập, truy xuất các giá trị dữ liệu cho từng đối tượng. Nhờ đó, chúng ta có được một khái niệm mới: kiểu dữ liệu trừu tượng (*abstract data type*), thường viết tắt là ADT.

Nguyên tắc quan trọng ở đây là một định nghĩa của bất kỳ một kiểu dữ liệu trừu tượng nào cũng gồm hai phần: phần thứ nhất mô tả cách mà các phần tử trong kiểu liên quan đến nhau, phần thứ hai là sự liệt kê các thao tác có thể thực hiện trên các phần tử của kiểu dữ liệu trừu tượng đó.

Lưu ý rằng khi định nghĩa cho một kiểu dữ liệu trừu tượng chúng ta hoàn toàn không quan tâm đến cách hiện thực của nó. Một định nghĩa cho một kiểu dữ liệu trừu tượng phụ thuộc vào những nhiệm vụ mà chúng ta trông đợi nó phải thực hiện được. Dưới đây là một số vấn đề chúng ta thường hay xem xét:

- Có quan tâm đến thứ tự thêm vào của các phần tử hay không?
- Việc truy xuất phần tử phụ thuộc thứ tự thêm vào của các phần tử, hay có thể truy xuất phần tử bất kỳ dựa vào khóa cho trước?
- Việc tìm kiếm phần tử theo khóa, nếu được phép, là hoàn toàn như nhau đối với bất kỳ khóa nào, hay phụ thuộc vào thứ tự khi thêm vào, hay phụ thuộc vào tần suất mà khóa được truy xuất?
- ...

Một đặc tả cho một kiểu dữ liệu trừu tượng hoàn toàn có thể có nhiều cách hiện thực khác nhau. Mỗi cách hiện thực mang lại tính khả thi và tính hiệu quả khác nhau. Điều này phụ thuộc vào yêu cầu về thời gian và không gian của bài toán. Nhưng cần nhấn mạnh rằng, mọi cách hiện thực của một kiểu dữ liệu trừu tượng đều luôn trung thành với đặc tả ban đầu về các chức năng của nó.

Nhiệm vụ của chúng ta trong việc hiện thực CTDL trong C++ là bắt đầu từ những khái niệm, thường là định nghĩa của một ADT, sau đó tinh chế dần để có được hiện thực bằng một lớp trong C++. Các phương thức của lớp trong C++ tương ứng một cách tự nhiên với các thao tác dữ liệu trên ADT, trong khi những thành phần dữ liệu của lớp trong C++ tương ứng với CTDL vật lý mà chúng ta chọn để biểu diễn ADT.

1.5. Một số nguyên tắc và phương pháp để học tốt môn CTDL và giải thuật

1.5.1. Cách tiếp cận và phương hướng suy nghĩ tích cực

Mỗi CTDL đều được trình bày theo đúng các bước sau đây:

- Định nghĩa lớp.
- Đặc tả lớp.
- Phân tích các phương án hiện thực.
- Hiện thực lớp.
-

Lưu ý rằng, sự trừu tượng và đặc tả dữ liệu phải luôn đi trước sự lựa chọn cách thức tổ chức lưu trữ dữ liệu và cách hiện thực chúng.

Trong phần định nghĩa và đặc tả lớp, để có thể hiểu sâu vấn đề và cảm thấy hứng thú hơn, sinh viên nên tự xem mình là một trong những người tham gia vào công việc thiết kế. Vì chức năng của lớp hoàn toàn phụ thuộc vào quan điểm của người thiết kế. Nếu chúng ta giới hạn cho mỗi lớp CTDL một số chức năng thao tác dữ liệu cơ bản nhất, chúng ta có một thư viện gọn nhẹ. Ngược lại, thư viện sẽ rất lớn, nhưng người lập trình có thể gọi thực hiện bất cứ công việc nào mà họ muốn từ những phương thức đã có sẵn của mỗi lớp. Thư viện các lớp CTDL trong VC++ là một minh họa cho thấy mỗi lớp CTDL có sẵn rất nhiều phương thức đáp ứng được nhu cầu của nhiều người dùng khác nhau.

Các phương thức được đặc tả kỹ càng cho mỗi lớp trong giáo trình này cũng chỉ là để minh họa. Sinh viên có thể tự ý chọn lựa để đặc tả một số phương thức bổ sung khác theo ý muốn.

Trước khi tìm hiểu các phương án hiện thực được trình bày trong giáo trình dành cho mỗi lớp CTDL, sinh viên cũng nên tự phác họa theo suy nghĩ của riêng

bản thân mình. Với cách chủ động như vậy, sinh viên sẽ dễ dàng nhìn ra các ưu nhược điểm trong từng phương án. Và nếu có được những ý tưởng hoàn toàn mới mẻ so với những gì được trình bày trong giáo trình, sinh viên sẽ tự tin hơn khi cần giải quyết các bài toán. Những CTDL nhằm phục vụ cho các bài toán lớn đôi khi được hình thành từ sự ghép nối của một số CTDL đơn giản. Chính sự ghép nối này làm nảy sinh vô vàn phương án khác nhau mà chúng ta phải chọn lựa thật thận trọng, để bảo đảm tính khả thi và hiệu quả của chương trình. Một khi gặp một bài toán cần giải quyết, nếu sinh viên biết chọn cho mình những phương án ghép nối các CTDL đơn giản, biết cách sử dụng lại những gì đã có trong thư viện, và biết cách làm thế nào để hiện thực bổ sung những gì thuộc về những ý tưởng mới mẻ vừa nảy sinh, xem như sinh viên đã học tốt môn CTDL và giải thuật.

So với nhiều giáo trình khác, giáo trình này tách riêng phần ứng dụng các CTDL nhằm làm gọn nhẹ hơn cho phần II là phần chỉ trình bày về các CTDL. Như vậy sẽ thuận tiện hơn cho sinh viên trong việc tìm hiểu những phần căn bản hay là tra cứu mở rộng kiến thức. Hơn nữa, có nhiều ứng dụng liên quan đến nhiều CTDL khác nhau.

1.5.2. Các nguyên tắc

1. Trước khi hiện thực bất kỳ một lớp CTDL nào, chúng ta cần chắc chắn rằng chúng ta đã định nghĩa CTDL và đặc tả các tác vụ cho nó một cách thật đầy đủ. Có như vậy mới bảo đảm được rằng:
   - Chúng ta đã hiểu về nó một cách chính xác.
   - Trong khi hiện thực chúng ta không phải quay lại sửa đổi các đặc tả của nó, vì việc sửa đổi này có thể làm cho chúng ta mất phương hướng, CTDL sẽ không còn đúng như những ý tưởng ban đầu mà chúng ta đã dự định cho nó.
   - Các chương trình ứng dụng không cần phải được chỉnh sửa một khi đã sử dụng CTDL này.
   - Nếu chúng ta cung cấp nhiều hiện thực khác nhau cho cùng một CTDL, thì khi đổi sang sử dụng một hiện thực khác, chương trình ứng dụng không đòi hỏi phải được chỉnh sửa lại. Các hiện thực khác nhau của cùng một CTDL luôn có cùng một giao diện thống nhất.

2. Mỗi phương thức của lớp luôn có năm phần mô tả (kiểu trả về, thông số vào/ra, *precondition*, *postcondition*, *uses*)

   Đây là yêu cầu chung trong việc lập tài liệu cho một hàm. Các CTDL của chúng ta sẽ được sử dụng trong rất nhiều ứng dụng khác nhau. Do đó chúng ta cần xây dựng sao cho người lập trình bớt được mọi công sức có thể. Lời khuyên ở đây là: phần *precondition* chỉ nhằm giải thích ý nghĩa các thông số

vào, chứ không nên ràng buộc những trị hợp lệ mà thông số vào phải thoả. Nhiệm vụ trong phần hiện thực của phương thức là chúng ta phải lường hết mọi khả năng có thể có của thông số vào và giải quyết thỏa đáng từng trường hợp.

> *Chúng ta xem các CTDL cũng như các dịch vụ, chúng được viết một lần và được sử dụng trong rất nhiều ứng dụng khác nhau. Do đó CTDL cần được xây dựng sao cho người sử dụng bớt được công sức mọi lúc có thể.*
>
> *Các phương thức public của các CTDL nên được hiện thực không có precondition.*

3. Đảm bảo tính đóng kín (*encapsulation*) của lớp CTDL. Dữ liệu có tính đóng kín khi chúng chỉ có thể được truy xuất bởi các phương thức của lớp.

Ưu điểm của việc sử dụng một lớp có tính đóng kín là khi chúng ta đặc tả và hiện thực các phương thức, chúng ta không phải lo lắng đến các giá trị không hợp lệ của các dữ liệu đang được lưu trong đối tượng của lớp.

> *Các thành phần dữ liệu của CTDL nên được khai báo private.*

4. Ngoại trừ các *constructor* có chủ đích, mỗi đối tượng của CTDL luôn phải được khởi tạo là một đối tượng rỗng và chỉ được sửa đổi bởi chính các phương thức của lớp. Với các phương thức đã được hiện thực và kiểm tra kỹ lưỡng, chúng ta luôn an tâm rằng các đối tượng CTDL luôn chứa những dữ liệu hợp lệ. Điều này giúp chúng luôn hoàn thành nhiệm vụ được giao, và đó cũng là nguyên tắc của hướng đối tượng.

1.5.3. Phong cách lập trình (*style of programming*) và các kỹ năng:

1. Vấn đề xử lý lỗi:

Việc xử lý lỗi cung cấp một mức độ an toàn cần thiết mà chúng ta nên thực hiện mọi lúc có thể trong CTDL của chúng ta. Có vài cách khác nhau trong việc xử lý lỗi. Chẳng hạn chúng ta có thể in ra thông báo hoặc ngưng chương trình khi gặp lỗi. Hoặc thay vào đó, chúng ta dành việc xử lý lỗi lại cho người lập trình sử dụng CTDL của chúng ta bằng cách trả về các mã lỗi thông qua trị trả về của các phương thức. Cách cuối cùng này hay hơn vì nó cung cấp khả năng lựa chọn cho người lập trình. Có những tình huống mà người lập trình thấy cần thiết phải ngưng ngay chương trình, nhưng cũng có những tình huống lỗi có thể bỏ qua để chương trình tiếp tục chạy. Bằng cách này, người lập trình khi sử dụng các CTDL hoàn toàn có thể chủ động đối

phó với mọi tình huống. Hơn nữa, các CTDL của chúng ta sẽ được xây dựng như là các thư viện dùng chung cho rất nhiều chương trình.

> *Khi sử dụng một phương thức của một lớp CTDL, người lập trình cần phải xem xét lại mã lỗi mà phương thức trả về để xử lý lỗi khi cần thiết.*
>
> *Các lớp CTDL cần phải được thiết kế sao cho có thể cho phép người lập trình chọn lựa cách thức xử lý lỗi theo ý muốn.*

Chúng ta sẽ dùng khai báo `ErrorCode` như một kiểu dữ liệu kiểu liệt kê tập các trị tương ứng các tình huống có thể xảy ra khi một phương thức của một lớp được gọi: thành công hay thất bại, tràn bộ nhớ, trị thông số không hợp lệ,... Chúng ta sẽ cố gắng thiết kế một cách thật nhất quán: hầu hết các phương thức của các lớp trả về kiểu `ErrorCode` này.

> *Sự nhất quán bao giờ cũng tạo ra thói quen rất tốt trong phong cách lập trình. Điều này tiết kiệm rất nhiều công sức và thời gian của người lập trình.*

2. Cách truyền nhận dữ liệu giữa đối tượng CTDL với chương trình sử dụng

Các giao tiếp truyền nhận dữ liệu khác giữa chương trình sử dụng và các lớp CTDL dĩ nhiên cũng thông qua danh sách các thông số. Trong phương thức của lớp CTDL sẽ không có việc chờ nhận dữ liệu trực tiếp từ bàn phím. Chúng ta nên dành cho người lập trình quyền chuyển hướng dòng nhập xuất dữ liệu với các thiết bị bên ngoài như bàn phím, màn hình, tập tin, máy in,...

3. Vấn đề kiểu của dữ liệu được lưu trong CTDL.

Mỗi lớp CTDL mà chúng ta xây dựng đều có tính tổng quát cao, nó có thể chấp nhận bất kỳ một kiểu dữ liệu nào cho dữ liệu được lưu trong nó. Trong C++ từ khóa `template` cho phép chúng ta làm điều này. Các kiểu dữ liệu này thường được yêu cầu phải có sẵn một số thao tác cần thiết như so sánh, nhập, xuất,...

4. Các khai báo bên trong một lớp CTDL.

Lớp CTDL cung cấp các thao tác dữ liệu thông qua các phương thức được khai báo là `public`.
Tất cả những thuộc tính và các hàm còn lại luôn được khai báo `private` hoặc `protected`.
Các thuộc tính của một lớp CTDL có thể được phân làm hai loại:

- Thuộc tính bắt buộc phải có để lưu dữ liệu.

- Thuộc tính mà đối tượng cần có để tự quản lý, trong số này có thuộc tính được bổ sung chỉ để đẩy nhanh tốc độ của các thao tác dữ liệu.

Các hàm được che dấu bên trong lớp được gọi là các hàm phụ trợ (auxilary function), chúng chỉ được sử dụng bởi chính các phương thức của lớp CTDL đó mà thôi.

Việc mở rộng thêm các tác vụ cho một lớp có sẵn có thể theo một trong hai cách:

- Bổ sung thêm phương thức mới.
- Xây dựng lớp thừa kế.

5. Một số hướng dẫn cần thiết trong việc thử nghiệm chương trình.

✓ Bộ chương trình thử (*driver*): Đây là đoạn chương trình thường được viết trong hàm `main` và chứa một thực đơn (*menu*) cho phép thử mọi phương thức của lớp CTDL đang được xây dựng.
Chúng ta sẽ viết, thử nghiệm, và hoàn chỉnh nhiều lớp CTDL khác nhau. Do đó ngay từ đầu chúng ta nên xây dựng một driver sao cho tổng quát, khi cần thử với một CTDL nào đó chỉ cần chỉnh sửa lại đôi chút mà thôi.
Trong driver chúng ta nên chuẩn hóa việc đọc ghi tập tin, xử lý các thao tác đọc từ bàn phím và xuất ra màn hình. Phần còn lại là một menu cho phép người sử dụng chạy chương trình chọn các chức năng như tạo đối tượng CTDL mới, gọi các thao tác thêm, xóa, tìm kiếm, truy xuất,... trên CTDL đó.

✓ Các mẩu tạm (*stub*): đây là một mẹo nhỏ nhưng rất hữu ích. Để dịch và chạy thử một vài phần nhỏ đã viết, những phần chưa viết của chương trình sẽ được tạo như những mẩu nhỏ và chỉ cần hợp cú pháp (Xem ứng dụng tính toán các đa thức trong chương 15).
Ví dụ: Trong đoạn chương trình nào đó chúng ta đang muốn chạy thử mà có sử dụng lớp A, hàm B,..., chúng ta sẽ tạm khai báo các *stub*:

```
class A
{
}; // Một lớp chưa có thuộc tính vì chúng ta chưa quyết định nên chọn
      kiểu      thuộc tính như thế nào.
void B()
{
} // Một hàm với thân hàm còn rỗng mà chúng ta hẹn sẽ viết sau.
```

Nếu một hàm đã có định nghĩa thì chỉ cần trả về sao cho hợp lệ:

```
int C()
{
    return 1;
} // chỉ cần cẩn thận trong trường hợp nếu như giá trị trả về lại được
  dùng trong một biểu thức luận lý để xét điều kiện lặp vòng thì có
  khả năng vòng lặp không được thực hiện hoặc lặp vô tận.
```

- ✓ Cách thức theo dõi một chương trình đang chạy hoặc nhu cầu khảo sát cách làm việc của một trình biên dịch nào đó:

  Ví dụ gợi ý:

```
void D()
{
    count << "\n Hàm D đang được gọi \n";
}
```

  Trong C++ các hàm *constructor* và *destructor* được trình biên dịch gọi khi một đối tượng vừa được tạo ra hoặc sắp bị hủy. Vậy nếu có thắc mắc về thứ tự gọi các hàm này của một lớp thừa kế từ lớp khác, chúng ta có thể dùng cách tương tự để viết trong *constructor* và *destructor* của từng lớp cha, con.

  Nếu chúng ta có thắc mắc về cách ứng xử của trình biên dịch khi gọi các hàm này hay các hàm được định nghĩa đè (*overloaded*, *overwriten*) trong trường hợp các lớp thừa kế lẫn nhau, hoặc một số trường hợp khác nào đó, thì đây là cách hay nhất để chúng ta tự kiểm nghiệm lấy.

  Phần lớn các giải thuật được nghiên cứu trước hết chỉ dựa trên ý tưởng (biểu diễn bằng ngôn ngữ giả và độc lập với mọi ngôn ngữ lập trình). Tuy nhiên khi hiện thực chúng ta thường gặp vướng mắc ở chỗ mỗi ngôn ngữ lập trình có một số đặc điểm khác nhau, và ngay cả khi dùng chung một ngôn ngữ, các trình biên dịch khác nhau (khác hãng sản xuất hay khác phiên bản) đôi khi cũng ứng xử khác nhau. Điều đó gây rất nhiều khó khăn và lãng phí thời gian của nhiều sinh viên.

  Chỉ cần lấy một ví dụ đơn giản, đó là việc đọc ghi file, việc thường xuyên phải cần đến khi muốn thử nghiệm một giải thuật nào đó. Các vòng lặp thường nhầm lẫn ở điều kiện kết thúc file trong ngôn ngữ C++, mà điều này hoàn toàn phụ thuộc vào việc xử lý con trỏ file của trình biên dịch. Ngay một phần mềm như Visual C++ hiện tại cũng chứa cùng lúc trong thư viện không biết bao nhiêu lớp phục vụ cho việc khai báo và đọc ghi file. Chúng ta chỉ có thể sử dụng một trong các thư viện đó một cách chính xác sau khi đã tìm hiểu thật kỹ! Một ví dụ khác cũng hay gây những lỗi mất rất nhiều thời gian, đó là việc so sánh các trị: NULL, '0', '\0', 0, … mà nếu không khảo sát kỹ chúng ta sẽ bị trả giá bởi sự chủ quan cho rằng mình đã hiểu đúng quy ước của trình biên dịch.

Việc tìm đọc tài liệu kèm theo trình biên dịch là một việc làm cần thiết, nó cho chúng ta sự hiểu biết đầy đủ và chính xác. Nhưng để rút ngắn thời gian thì gợi ý trên đây cũng là một lời khuyên quý báu. Không gì nhanh và chính xác bằng cách tìm câu trả lời trong thử nghiệm. Việc sửa đổi chương trình như thế nào để có được các *stub* thỏa những nhu cầu cần thử nghiệm là tùy thuộc vào sự tích cực, say mê và sáng tạo của sinh viên.

- ✓ Gỡ rối chương trình (*debug*)
  Đây là khả năng theo vết chương trình ở những đoạn mà người lập trình còn nghi ngờ có lỗi. Bất cứ người lập trình nào cũng có lúc cần phải chạy debug. Vì vậy tốt hơn hết là ngay từ đầu sinh viên nên tìm hiểu kỹ các khả năng của công cụ debug của trình biên dịch mà mình sử dụng (cho phép theo dõi trị các biến, lịch sử các lần gọi hàm,...).

  Một gợi ý trong phần này là sinh viên cần biết cách cô lập từng phần của chương trình đã viết bằng cách dùng ký hiệu dành cho phần chú thích (*comment*) để khóa bớt những phần chưa đến lượt kiểm tra. Hoặc khi lỗi do trình biên dịch báo có vẻ mơ hồ, thì cách cô lập bằng cách giới hạn dần đoạn chương trình đang dịch thử sẽ giúp chúng ta sớm xác định được phạm vi có lỗi. Cũng có thể làm ngược lại, chỉ dịch thử và chỉnh sửa từng đoạn chương trình nhỏ, cho đến khi hết lỗi mới nới dần phạm vi chương trình để dịch tiếp.

1.6. Giới thiệu về ngôn ngữ giả:

Phần lớn chương trình được trình bày trong giáo trình này đều sử dụng ngôn ngữ C++, sau khi ý tưởng về giải thuật đã được giải thích cặn kẽ. Phần còn lại có một số giải thuật được trình bày bằng ngôn ngữ giả.

Ngôn ngữ giả, hay còn gọi là mã giả (*pseudocode*), là một cách biểu diễn độc lập với mọi ngôn ngữ lập trình, nó không ràng buộc sinh viên vào những cú pháp nghiêm ngặt cũng như cách gọi sao cho chính xác các từ khóa, các hàm có trong thư viện một trình biên dịch nào đó. Nhờ đó sinh viên có thể tập trung vào ý tưởng lớn của giải thuật.

Các quy định về mã giả được sử dụng trong giáo trình này:

- ➢ Biểu diễn sự tuần tự của các lệnh chương trình: các lệnh được thực thi tuần tự lệnh này sang lệnh khác sẽ có cùng khoảng cách canh lề như nhau và được đánh số thứ tự tăng dần, luôn bắt đầu từ 1.

- Cấu trúc khối lồng nhau: một khối nằm trong một khối khác sẽ có khoảng cách canh lề lớn hơn.

  Trong giáo trình này, chỉ những phần được trình bày bằng mã giả mới có số thứ tự ở đầu mỗi dòng lệnh.

  Ví dụ:

```
1.
   1.
   2.
     1. // Đây là dòng lệnh có số thứ tự là 1.2.1
     2.
   3.
2.
   1.
3.
   1.
   2.
```

- Sự rẽ nhánh: chúng ta sử dụng các từ khóa:
  - `if` <biểu thức luận lý>

    …

    `endif`
  - `if` <biểu thức luận lý>

    …

    `else`

    …

    `endif`
  - `case`

    `case1:` …

    `case2:` …

    `case3:` …

    `else:` …

    `endcase`

- Sự lặp vòng:
  - `loop` <biểu thức luận lý>

    …

    `endloop` // lặp trong khi biểu thức luận lý còn đúng.
  - `repeat`

    …

    `until` <biểu thức luận lý> // lặp cho đến khi biểu thức luận lý đúng.

- Khai báo hàm:
  <kiểu trả về> tên hàm (danh sách thông số)
  trong đó danh sách thông số: `val/ ref` <tên kiểu> tên thông số, val/ ref <tên kiểu> tên thông số,...
  `val`: dành cho tham trị; `ref`: dành cho tham biến.

- Khai báo cấu trúc, lớp:
  `struct` tên kiểu dữ liệu cấu trúc
  `end struct`

  class tên kiểu dữ liệu cấu trúc
  `end class`

- Khai báo phương thức của lớp:
  <kiểu trả về> tên lớp::tên hàm (danh sách thông số);

- Khai báo biến:
  <tên kiểu> tên biến

<u>Một chút lưu ý về cách trình bày trong giáo trình</u>:

Do các đoạn chương trình sử dụng *font* chữ `Courier New`, nên các tên biến, tên lớp, tên đối tượng, tên các hàm khi được nhắc đến cũng dùng *font* chữ này. Các từ tiếng Anh khác được in nghiêng. Đặc biệt những phần có liên quan chặt chẽ đến những đặc thù của ngôn ngữ lập trình C++ thường dùng kích cỡ chữ nhỏ hơn, để phân biệt với những phần quan trọng khác khi nói về ý tưởng và giải thuật, và đó mới là mục đích chính của môn học này.
Có một số từ hay đoạn được in đậm hay gạch dưới nhằm giúp sinh viên đọc dễ dàng hơn.

Phần 2 – CÁC CẤU TRÚC DỮ LIỆU

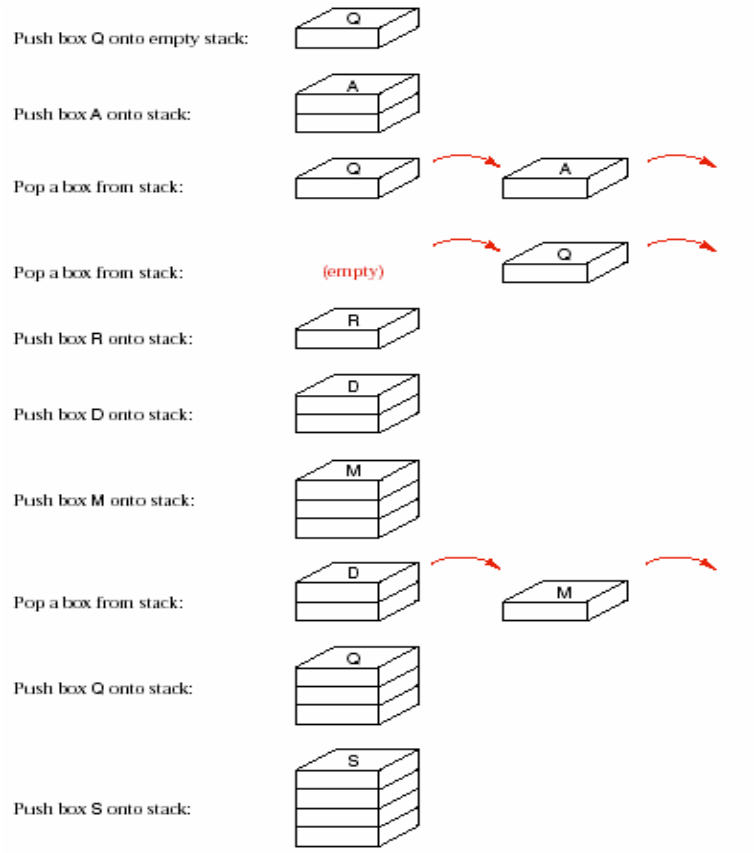
Chương 2 – NGĂN XẾP

Chúng ta sẽ tìm hiểu một CTDL đơn giản nhất, đó là ngăn xếp. Một cách nhất quán như phần giới thiệu môn học đã trình bày, mỗi CTDL đều được xây dựng theo đúng trình tự:

- Định nghĩa.
- Đặc tả.
- Phân tích các phương án hiện thực.
- Hiện thực.

2.1. Định nghĩa ngăn xếp

Với định nghĩa danh sách trong chương mở đầu, chúng ta hiểu rằng trong danh sách, mỗi phần tử, ngoại trừ phần tử cuối, đều có duy nhất một phần tử đứng sau nó. Ngăn xếp là một trường hợp của danh sách, được sử dụng trong các ứng dụng có liên quan đến sự đảo ngược. Trong CTDL ngăn xếp, việc thêm hay lấy dữ liệu chỉ được thực hiện tại một đầu. Dữ liệu thêm vào trước sẽ lấy ra sau, tính chất này còn được gọi là vào trước ra sau (*First In Last Out* - FILO).
Đầu thêm hay lấy dữ liệu của ngăn xếp còn gọi là đỉnh (*top*) của ngăn xếp.



Hình 2.1- Thêm phần tử vào và lấy phần tử ra khỏi ngăn xếp.

Vậy chúng ta có định nghĩa của ngăn xếp dưới đây, không khác gì đối với định nghĩa danh sách, ngoại trừ cách thức mà ngăn xếp cho phép thay đổi hoặc truy xuất đến các phần tử của nó.

Định nghĩa: Một ngăn xếp các phần tử kiểu T là một chuỗi nối tiếp các phần tử của T, kèm các tác vụ sau:

1. Tạo một đối tượng ngăn xếp rỗng.

2. Đẩy (`push`) một phần tử mới vào ngăn xếp, giả sử ngăn xếp chưa đầy (phần tử dữ liệu mới luôn được thêm tại đỉnh).

3. Lấy (`pop`) một phần tử ra khỏi ngăn xếp, giả sử ngăn xếp chưa rỗng (phần tử bị loại là phần tử đang nằm tại đỉnh).

4. Xem phần tử tại đỉnh ngăn xếp (`top`).

Lưu ý rằng định nghĩa này không quan tâm đến cách hiện thực của kiểu dữ liệu trừu tượng ngăn xếp. Chúng ta sẽ tìm hiểu một vài cách hiện thực khác nhau của ngăn xếp và tất cả chúng đều phù hợp với định nghĩa này.

2.2. Đặc tả ngăn xếp

Ngoài các tác vụ chính trên, các phương thức khác có thể bổ sung tuỳ vào nhu cầu mà chúng ta thấy cần thiết:

+ `empty`: cho biết ngăn xếp có rỗng hay không.
+ `full`: cho biết ngăn xếp có đầy hay chưa.
+ `clear`: xóa sạch tất cả dữ liệu và làm cho ngăn xếp trở nên rỗng.

Chúng ta lưu ý rằng, khi thiết kế các phương thức cho mỗi lớp CTDL, ngoài một số phương thức chính để thêm vào hay lấy dữ liệu ra, chúng ta có thể bổ sung thêm nhiều phương thức khác. Việc thêm dựa vào quan niệm của mỗi người về sự tiện dụng của lớp CTDL đó. Nhưng điều đặc biệt quan trọng ở đây là các phương thức đó không thể mâu thuẫn với định nghĩa ban đầu cũng như các chức năng mà chúng ta đã định ra cho lớp. Chẳng hạn, trong trường hợp ngăn xếp của chúng ta, để bảo đảm quy luật "Vào trước ra sau" thì trật tự của các phần tử trong ngăn xếp là rất quan trọng. Chúng ta không thể cho chúng một phương thức có thể thay đổi trật tự của các phần tử đang có, hoặc phương thức lấy một phần tử tại một vị trí bất kỳ nào đó của ngăn xếp.

Trên đây là những phương thức liên quan đến các thao tác dữ liệu trên ngăn xếp.

Đối với bất kỳ lớp CTDL nào, chúng ta cũng không thể không xem xét đến hai phương thức cực kỳ quan trọng: đó chính là hai hàm dựng lớp và hủy lớp: ***constructor*** và ***destructor***. Trong C++ các hàm *constructor* và *destructor* được

trình biên dịch gọi khi một đối tượng vừa được tạo ra hoặc sắp bị hủy. Chúng ta cần bảo đảm cho mỗi đối tượng CTDL được tạo ra có trạng thái ban đầu là hợp lệ. Sự hợp lệ này sẽ tiếp tục được duy trì bởi các phương thức thao tác dữ liệu bên trên.

Trạng thái ban đầu hợp lệ là trạng thái rỗng không chứa dữ liệu nào hoặc trạng thái đã chứa một số dữ liệu theo như mong muốn của người lập trình sử dụng CTDL. Như vậy, mỗi lớp CTDL luôn có một *constructor* mặc định (không có thông số) để tạo đối tượng rỗng, các *constructor* có thông số khác chúng ta có thể thiết kế bổ sung nếu thấy hợp lý và cần thiết.

Một đối tượng CTDL khi bị hủy phải đảm bảo không để lại rác trong bộ nhớ. Chúng ta đã biết rằng, với các biến cấp phát tĩnh, trình biên dịch sẽ tự lấy lại những vùng nhớ đã cấp phát cho chúng. Với các biến cấp phát động thì ngược lại, vùng nhớ phải được chương trình giải phóng khi không còn sử dụng đến. Như vậy, đối với mỗi phương án hiện thực cụ thể cho mỗi lớp CTDL mà có sử dụng vùng nhớ cấp phát động, chúng ta sẽ xây dựng *destructor* cho nó để lo việc giải phóng vùng nhớ trước khi đối tượng bị hủy.

Trong C++, *constructor* có cùng tên với lớp và không có kiểu trả về. *Constructor* của một lớp được gọi một cách tự động khi một đối tượng của lớp đó được khai báo.

Đặc tả *constructor* cho lớp ngăn xếp, mà chúng ta đặt tên là lớp `Stack`, như sau:

```
template <class Entry>
Stack<Entry>::Stack();
```
pre: không có.
post: đối tượng ngăn xếp vừa được tạo ra là rỗng.
uses: không có.

Để đặc tả tiếp cho các phương thức khác, chúng ta chọn ra các trị của `ErrorCode` đủ để sử dụng cho lớp `Stack` là:

`success, overflow, underflow`

Các thông số dành cho các phương thức dưới đây được thiết kế tùy vào chức năng và nhu cầu của từng phương thức.

Phương thức loại một phần tử ra khỏi ngăn xếp:

```
template <class Entry>
ErrorCode Stack<Entry>::pop();
```
pre: không có.
post: nếu ngăn xếp không rỗng, phần tử tại đỉnh ngăn xếp được lấy đi, `ErrorCode` trả về là `success`; nếu ngăn xếp rỗng, `ErrorCode` trả về là `underflow`, ngăn xếp không đổi.
uses: không có.

Phương thức thêm một phần tử dữ liệu vào ngăn xếp:

```
template <class Entry>
ErrorCode Stack<Entry>::push(const Entry &item);
```
pre: không có.
post: nếu ngăn xếp không đầy, `item` được thêm vào trên đỉnh ngăn xếp, `ErrorCode` trả về là `success`; nếu ngăn xếp đầy, `ErrorCode` trả về là `overflow`, ngăn xếp không đổi.
uses: không có.

Lưu ý rằng `item` trong thông số của `push` đóng vai trò là tham trị nên được khai báo là tham chiếu hằng (*const reference*). Trong khi đó trong phương thức `top, item` là tham biến.

Hai phương thức `top` và `empty` được khai báo **`const`** vì chúng không làm thay đổi ngăn xếp.

```
template <class Entry>
ErrorCode Stack<Entry>:: top(Entry &item) const;
```
pre: không có
post: nếu ngăn xếp không rỗng, phần tử tại đỉnh ngăn xếp được chép vào `item`, `ErrorCode` trả về là `success`; nếu ngăn xếp rỗng, `ErrorCode` trả về là `underflow`; cả hai trường hợp ngăn xếp đều không đổi.
uses: không có.

```
template <class Entry>
bool Stack<Entry>::empty() const;
```
pre: không có
post: nếu ngăn xếp rỗng, hàm trả về `true`; nếu ngăn xếp không rỗng, hàm trả về `false`, ngăn xếp không đổi.
uses: không có.

Sinh viên có thể đặc tả tương tự cho phương thức **`full, clear`**, hay các phương thức bổ sung khác.
Từ nay về sau, chúng ta quy ước rằng nếu hai phần *precondition* hoặc *uses* không có thì chúng ta không cần phải ghi ra.

Chúng ta có phần giao tiếp mà lớp `Stack` dành cho người lập trình sử dụng như sau:

```
template<class Entry>
class Stack {
public:
   Stack();
   bool empty() const;
   ErrorCode pop();
   ErrorCode top(Entry &item) const;
   ErrorCode push(const Entry &item);
};
```

Với một đặc tả như vậy chúng ta đã hoàn toàn có thể sử dụng lớp Stack trong các ứng dụng. Sinh viên nên tiếp tục xem đến phần trình bày các ứng dụng về ngăn xếp trong chương 14. Dưới đây là chương trình minh họa việc sử dụng ngăn

xếp thông qua các đặc tả trên. Chương trình giải quyết bài toán in các số theo thứ tự ngược với thứ tự nhập vào đã được trình bày trong phần mở đầu.

Ví dụ:

Chương trình sẽ đọc vào một số nguyên n và n số thực kế đó. Mỗi số thực nhập vào sẽ được lưu vào ngăn xếp. Cuối cùng các số được lấy từ ngăn xếp và in ra.

```
#include   <Stack> //Sử dụng lớp Stack.
int main()
/*
pre:  Người sử dụng nhập vào một số nguyên n và n số thực.
post:  Các số sẽ được in ra theo thứ tự ngược.
uses:  lớp Stack và các phương thức của Stack.
*/
{
   int n;
   double item;
   Stack<double> numbers;
   cout <<"Type in an integer n followed by n decimal numbers."<< endl;
   cout << " The numbers will be printed in reverse order." << endl;
   cin  >> n;

   for (int i = 0; i < n; i++) {
      cin >> item;
      numbers.push(item);
   }

   cout << endl << endl;
   while (!numbers.empty()) {
      numbers.top(item)
      cout << item << " ";
      numbers.pop();
   }
   cout << endl;
}
```

Che dấu thông tin: khi sử dụng lớp `Stack` chúng ta không cần biết nó được lưu trữ trong bộ nhớ như thế nào và các phương thức của nó hiện thực ra sao. Đây là vấn đề che dấu thông tin (*information hiding*).

Một CTDL có thể có nhiều cách hiện thực khác nhau, nhưng mọi cách hiện thực đều có chung phần đặc tả các giao tiếp đối với bên ngoài. Nhờ đó mà các chương trình ứng dụng giữ được sự độc lập với các hiện thực khác nhau của cùng một lớp CTDL. Khi cần thay đổi hiện thực của CTDL mà ứng dụng đang sử dụng, chúng ta không cần chỉnh sửa mã nguồn của ứng dụng.

Tính khả thi và hiệu quả của ứng dụng: Tuy ứng dụng cần phải độc lập với hiện thực của cấu trúc dữ liệu, nhưng việc chọn cách hiện thực nào ảnh hưởng đến tính khả thi và hiệu quả của ứng dụng. Chúng ta cần hiểu các ưu nhược điểm của mỗi cách hiện thực của cấu trúc dữ liệu để lựa chọn cho phù hợp với tính chất của ứng dụng.

Tính trong sáng của chương trình: Ưu điểm khác của che dấu thông tin là tính trong sáng của chương trình. Những tên gọi quen thuộc dành cho các thao tác trên cấu trúc dữ liệu giúp chúng ta hình dung rõ ràng giải thuật của chương trình. Chẳng hạn với thao tác trên ngăn xếp, người ta thường quen dùng các từ: `push` – đẩy vào ngăn xếp, `pop` – lấy ra khỏi ngăn xếp.

Thiết kế từ trên xuống: Sự tách rời giữa việc sử dụng cấu trúc dữ liệu và cách hiện thực của nó còn giúp chúng ta thực hiện tốt hơn quá trình thiết kế từ trên xuống (*top-down design*) cả cho cấu trúc dữ liệu và cả cho chương trình ứng dụng.

2.3. Các phương án hiện thực ngăn xếp

Trong phần này chúng ta sẽ tìm hiểu các phương án hiện thực cho lớp ngăn xếp. Các ưu nhược điểm của các cách hiện thực khác nhau đối với một đặc tả CTDL thường liên quan đến những vấn đề sau đây:

- Cho phép hay không cho phép lưu trữ và thao tác với lượng dữ liệu lớn.
- Tốc độ xử lý của các phương thức.

Vì ngăn xếp là một trường hợp đặc biệt của danh sách, nên đã đến lúc chúng ta bàn đến cách lưu trữ các phần tử trong một danh sách. Có hai phương án lưu trữ chính:

- Các phần tử nằm kế nhau trong bộ nhớ mà chúng ta hay dùng từ liên tục (*contiguous*) để nói đến.
- Các phần tử không nằm kế nhau trong bộ nhớ mà chúng ta dùng công cụ là các biến kiểu con trỏ (*pointer*) để quản lý, trường hợp này chúng ta gọi là danh sách liên kết (*linked list*).

Hiện thực liên tục đơn giản nhưng bị hạn chế ở chỗ kích thước cần được biết trước. Nếu dùng mảng lớn quá sẽ bị lãng phí, nhưng nhỏ quá dễ bị đầy. Hiện thực liên kết linh động hơn, nó chỉ bị đầy khi bộ nhớ thực sự không còn chỗ trống nữa.

2.4. Hiện thực ngăn xếp

2.4.1. Hiện thực ngăn xếp liên tục

Để hiện thực lớp ngăn xếp liên tục (*contiguous stack*), chúng ta dùng một mảng (*array* trong C++) để chứa các phần tử của ngăn xếp và một thuộc tính `count` cho biết số phần tử hiện có trong ngăn xếp.

```
const int max = 10;    // Dùng số nhỏ để kiểm tra chương trình.
template <class Entry>
class Stack {
public:
   Stack();
```

```
   bool empty() const;
   ErrorCode pop();
   ErrorCode top(Entry &item) const;
   ErrorCode push(const Entry &item);
private:
   int count;
   Entry entry[max];
};
```

Push, pop, và các phương thức khác

Ý tưởng chung của các phương thức này như sau:

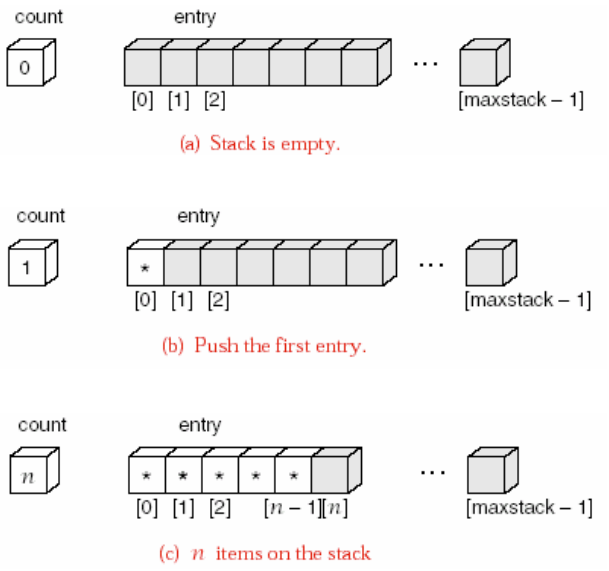
- Việc thêm dữ liệu mới chỉ thực hiện được khi ngăn xếp còn chỗ trống.
- Việc loại phần tử khỏi ngăn xếp hoặc xem phần tử trên đỉnh ngăn xếp chỉ thực hiện được khi ngăn xếp không rỗng.
- Do `count` là số phần tử hiện có trong ngăn xếp và chỉ số của *array* trong C++ được bắt đầu từ 0, nên `count-1` chính là chỉ số của phần tử tại đỉnh ngăn xếp khi cần xem hoặc cần loại bỏ khỏi ngăn xếp.
- Khi cần thêm phần tử mới, `count` là chỉ số chỉ đến vị trí còn trống ngay trên đỉnh ngăn xếp, cũng là chỉ số của phần tử mới nếu được thêm vào.
- Khi ngăn xếp được thêm hoặc lấy phần tử thì thuộc tính `count` luôn phải được cập nhật lại.
- *Constructor* tạo đối tượng ngăn xếp rỗng bằng cách gán thuộc tính `count` bằng 0. Lưu ý rằng chúng ta không cần quan tâm đến trị của những phần tử nằm từ vị trí `count` cho đến hết mảng (`max -1`), các vị trí từ 0 đến `count-1` mới thực sự chứa những dữ liệu đã được đưa vào ngăn xếp.

```
template <class Entry>
ErrorCode Stack<Entry>::push(const Entry &item)
/*
post: nếu ngăn xếp không đầy, item được thêm vào trên đỉnh ngăn xếp, ErrorCode trả về là
      success; nếu ngăn xếp đầy, ErrorCode trả về là overflow, ngăn xếp không đổi.
*/
{
   ErrorCode outcome = success;
   if (count >= max)
      outcome = overflow;
   else
      entry[count++] = item;
   return outcome;
}

template <class Entry>
ErrorCode Stack<Entry>::pop()
/*
post: nếu ngăn xếp không rỗng, phần tử tại đỉnh ngăn xếp được lấy đi, ErrorCode trả về là
      success; nếu ngăn xếp rỗng, ErrorCode trả về là underflow, ngăn xếp không đổi.
*/
{
   ErrorCode outcome = success;
   if (count == 0)
      outcome = underflow;
```

```
   else --count;
   return outcome;
}
```

```
template <class Entry>
ErrorCode Stack<Entry>::top(Entry &item) const
/*
post: nếu ngăn xếp không rỗng, phần tử tại đỉnh ngăn xếp được chép vào item, ErrorCode trả
      về là success; nếu ngăn xếp rỗng, ErrorCode trả về là underflow; cả hai trường hợp
      ngăn xếp  đều không đổi.
*/
{
   ErrorCode outcome = success;
   if (count == 0)
      outcome = underflow;
   else
      item = entry[count - 1];
   return outcome;
}
```

```
template <class Entry>
bool Stack<Entry>::empty() const
/*
post: nếu ngăn xếp rỗng, hàm trả về true; nếu ngăn xếp không rỗng, hàm trả về false, ngăn
      xếp không đổi.
*/
{
   bool outcome = true;
   if (count > 0) outcome = false;
   return outcome;
}
```



Hình 2.2- Biểu diễn của dữ liệu trong ngăn xếp liên tục.

Constructor sẽ khởi tạo một ngăn xếp rỗng.

```
template <class Entry>
Stack<Entry>::Stack()
/*
post: ngăn xếp được khởi tạo rỗng.
*/
{
   count = 0;
}
```

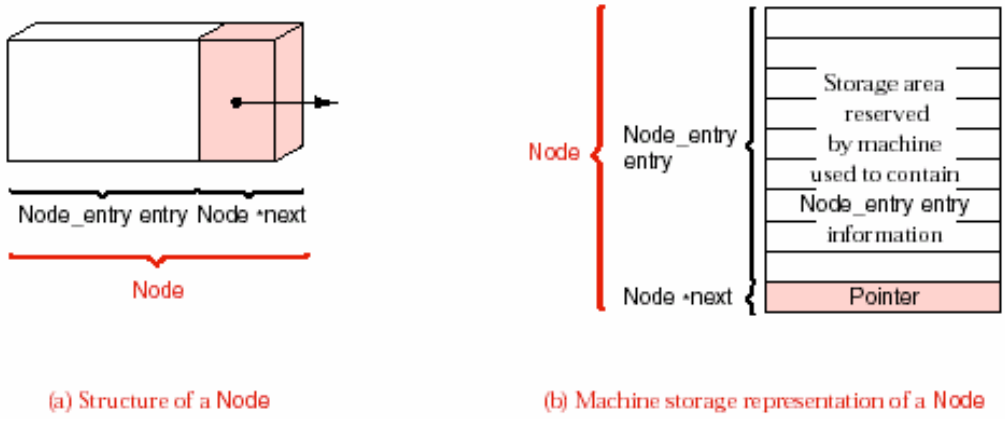
2.4.2. Hiện thực ngăn xếp liên kết

Cấu trúc liên kết được tạo bởi các phần tử , mỗi phần tử chứa hai phần: một chứa thông tin chính là dữ liệu của phần tử, một chứa con trỏ tham chiếu đến phần tử kế, và được khai báo trong C++ như sau:

```
template <class Entry>
struct Node {
//  data members
   Entry entry;
   Node<Entry> *next;
//  constructors
   Node();
   Node(Entry item, Node<Entry> *add_on = NULL);
};
```

Và đây là hình ảnh của một phần tử trong cấu trúc liên kết:

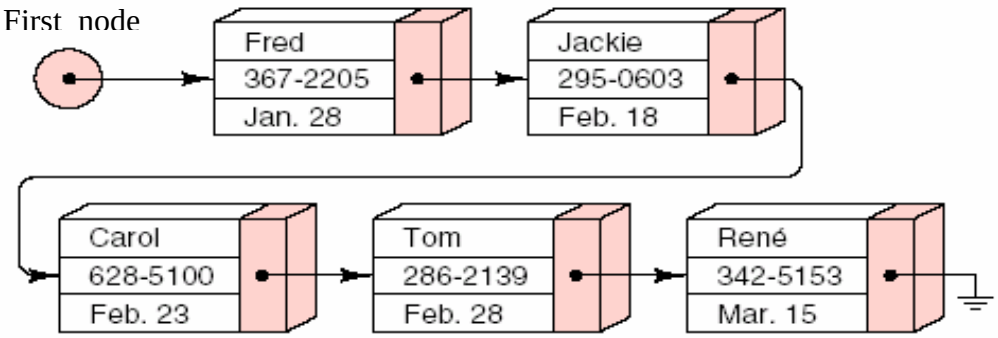
Hình dưới đây biểu diễn một cấu trúc liên kết có con trỏ chỉ đến phần tử đầu là `First_node`.



Hình 2.3- Cấu trúc `Node` chứa con trỏ

Vấn đề đặt ra là chúng ta nên chọn phần tử đầu hay phần tử cuối của cấu trúc liên kết làm đỉnh của ngăn xếp. Thoạt nhìn, dường như việc thêm một `node` mới vào cuối cấu trúc liên kết là dễ hơn (tương tự như đối với ngăn xếp liên tục). Nhưng việc loại một phần tử ở cuối là khó bởi vì việc xác định phần tử ngay trước

phần tử bị loại không thể thực hiện nhanh chóng. Lý do là các con trỏ trong cấu trúc liên kết chỉ theo một chiều. Khi loại đi một phần tử ở cuối cấu trúc liên kết, chúng ta phải bắt đầu từ đầu, lần theo các con trỏ, mới xác định được phần tử cuối. Do đó, cách tốt nhất là việc thêm vào hay loại phần tử đều được thực hiện ở phần tử đầu của cấu trúc liên kết. Đỉnh của ngăn xếp liên kết được chọn là phần tử đầu của cấu trúc liên kết.



Hình 2.4- Cấu trúc liên kết

Mỗi cấu trúc liên kết cần một thành phần con trỏ chỉ đến phần tử đầu tiên. Đối với ngăn xếp liên kết, thành phần này luôn chỉ đến đỉnh của ngăn xếp. Do mỗi phần tử trong cấu trúc liên kết có tham chiếu đến phần tử kế nên từ phần tử đầu tiên chúng ta có thể đến mọi phần tử trong ngăn xếp liên kết bằng cách lần theo các tham chiếu này. Từ đó, thông tin duy nhất cần thiết để có thể truy xuất dữ liệu trong ngăn xếp liên kết là địa chỉ của phần tử tại đỉnh. Phần tử tại đáy ngăn xếp luôn có tham chiếu là `NULL`.

```
template <class Entry>
class Stack {
public:
   Stack();
   bool empty() const;
   ErrorCode push(const Entry &item);
   ErrorCode pop();
   ErrorCode top(Entry &item) const;
protected:
   Node<Entry> *top_node;
};
```

Do lớp `Stack` giờ đây chỉ chứa một phần tử dữ liệu, chúng ta có thể cho rằng việc dùng lớp có thể không cần thiết, thay vào đó chúng ta dùng luôn một biến để chứa địa chỉ của đỉnh ngăn xếp. Tuy nhiên, ở đây có bốn lý do để sử dụng lớp `Stack`.

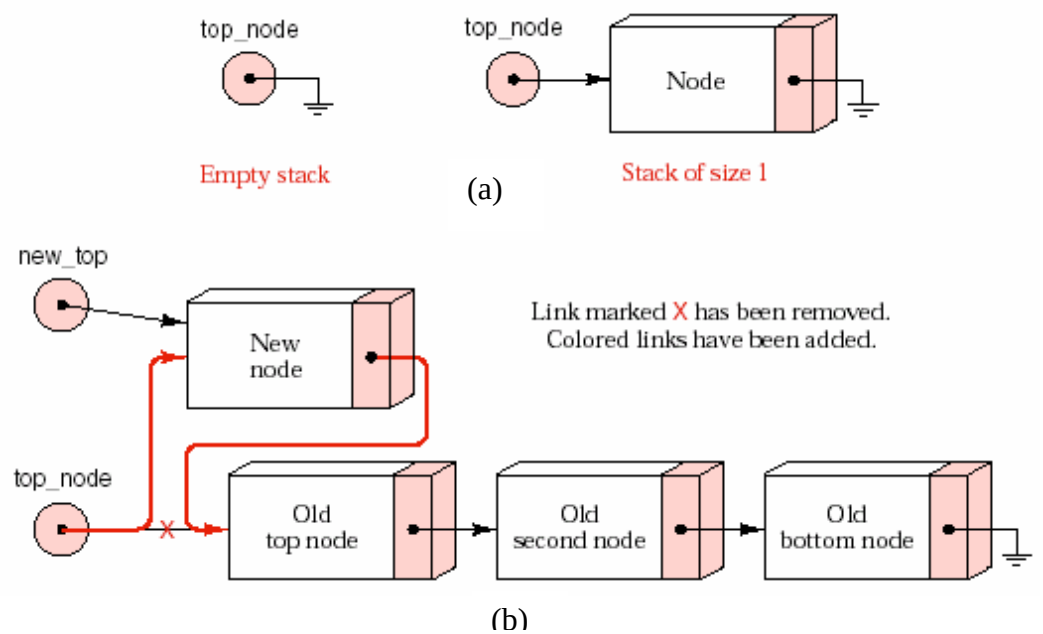
- Lý do quan trọng nhất là sự duy trì tính đóng kín: nếu chúng ta không sử dụng lớp ngăn xếp, chúng ta không thể xây dựng các phương thức cho ngăn xếp.

- Lý do thứ hai là để duy trì sự khác biệt luận lý giữa lớp ngăn xếp, mà bản thân được tạo bởi các phần tử là các `node`, với `top` của ngăn xếp là một con trỏ tham chiếu đến chỉ một `node`. Việc chúng ta chỉ cần nắm giữ `top` của ngăn xếp, là có thể tìm đến mọi phần tử khác của ngăn xếp tuy là hiển nhiên, nhưng không thích đáng với cấu trúc luận lý này.
- Lý do thứ ba là để duy trì tính nhất quán với các cấu trúc dữ liệu khác cũng như các cách hiện thực khác nhau của một cấu trúc dữ liệu: một cấu trúc dữ liệu bao gồm các dữ liệu và một tập các thao tác.
- Cuối cùng, việc xem ngăn xếp như một con trỏ đến đỉnh của nó không được phù hợp với các kiểu dữ liệu. Thông thường, các kiểu dữ liệu phải có khả năng hỗ trợ trong việc *debug* chương trình bằng cách cho phép trình biên dịch thực hiện việc kiểm tra kiểu một cách tốt nhất.

Chúng ta hãy bắt đầu bằng một ngăn xếp rỗng, top_node == NULL, và xem xét việc thêm phần tử đầu tiên vào ngăn xếp. Chúng ta cần tạo một node mới chứa bản sao của thông số item nhận vào bở phương thức push. Node này được truy xuất bởi biến con trỏ new_top. Sau đó địa chỉ chứa trong new_top sẽ được chép vào top_node của Stack (hình 2.5a):

```
Node *new_top = new Node<Entry>(item);
top_node = new_top;
```

Chú ý rằng ở đây, *constructor* khi tạo một `node` mới đã gán `next` của nó bằng `NULL`, và chúng ta hoàn toàn an tâm vì không bao giờ có con trỏ mang trị ngẫu nhiên.



Hình 2.5- Thêm một phần tử vào ngăn xếp liên kết.

Nếu trung thành với nguyên tắc "Không bao giờ để một biến con trỏ mang trị ngẫu nhiên", chúng ta sẽ giảm được gánh nặng đáng kể trong công sức lập trình vì không phải mất quá nhiều thì giờ và đau đầu do những lỗi mà nó gây ra.

Để tiếp tục, xem như chúng ta đã có một ngăn xếp không rỗng. Để đưa thêm phần tử vào ngăn xếp, chúng ta cần thêm một `node` vào ngăn xếp. Trước hết chúng ta cần tạo một `node` mới được tham chiếu bởi con trỏ `new_top`, `node` này phải có dữ liệu là `item` và liên kết `next` tham chiếu đến `top` cũ của ngăn xếp. Sau đó chúng ta sẽ thay đổi `top_node` của ngăn xếp tham chiếu đến `node` mới này (hình 2.5b). Thứ tự của hai phép gán này rất quan trọng: nếu chúng ta làm theo thứ tự ngược lại, việc thay đổi `top_node` sớm sẽ làm mất khả năng truy xuất các phần tử đã có của ngăn xếp. Chúng ta có phương thức `push` như sau:

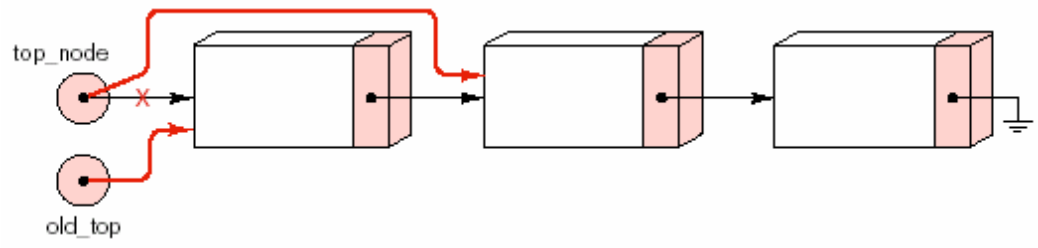
```
template <class Entry>
ErrorCode Stack<Entry>::push(const Entry &item)
/*
post: nếu ngăn xếp không đầy, item được thêm vào trên đỉnh ngăn xếp, ErrorCode trả về là
      success; nếu ngăn xếp đầy, ErrorCode trả về là overflow, ngăn xếp không đổi.
*/
{
   Node *new_top = new Node<Entry>(item, top_node);
   if (new_top == NULL) return overflow;
   top_node = new_top;
   return success;
}
```

Khi thay đổi các tham chiếu (các biến con trỏ), thứ tự các phép gán luôn cần được xem xét một cách kỹ lưỡng.

.

Phương thức `push` trả về `ErrorCode` là `overflow` trong trường hợp bộ nhớ động không tìm được chỗ trống để cấp phát cho phần tử mới, toán tử `new` trả về trị `NULL`.

Việc lấy một phần tử ra khỏi ngăn xếp thực sự đơn giản:



Hình 2.6- Lấy một phần tử ra khỏi ngăn xếp liên kết.

```
template <class Entry>
ErrorCode Stack<Entry>::pop()
/*
post: nếu ngăn xếp không rỗng, phần tử tại đỉnh ngăn xếp được lấy đi, ErrorCode trả về là
      success; nếu ngăn xếp rỗng, ErrorCode trả về là underflow, ngăn xếp không đổi.
*/
{
   Node *old_top = top_node;
   if (top_node == NULL) return underflow;
   top_node = old_top->next;
   delete old_top;
   return success;
}
```

Lưu ý rằng trong phương thức pop, chỉ cần gán top_node của ngăn xếp tham chiếu đến phần tử thứ hai trong ngăn xếp thì phần tử thứ nhất xem như đã được loại khỏi ngăn xếp. Tuy nhiên, nếu không thực hiện việc giải phóng phần tử trên đỉnh ngăn xếp, chương trình sẽ gây ra rác. Trong ứng dụng nhỏ, phương thức pop vẫn chạy tốt. Nhưng nếu ứng dụng lớn gọi phương thức này rất nhiều lần, số lượng rác sẽ lớn lên đáng kể dẫn đến không đủ vùng nhớ để chương trình chạy tiếp.

Khi một cấu trúc dữ liệu được hiện thực, nó phải được xử lý tốt trong mọi trường hợp để có thể được sử dụng trong nhiều ứng dụng khác nhau.

2.4.3. Ngăn xếp liên kết với sự an toàn

Khi sử dụng các phương thức mà chúng ta vừa xây dựng cho ngăn xếp liên kết, người lập trình có thể vô tình gây nên rác hoặc phá vỡ tính đóng kín của các đối tượng ngăn xếp. Trong phần này chúng ta sẽ xem xét chi tiết về các nguy cơ làm mất đi tính an toàn và tìm hiểu thêm ba phương thức mà C++ cung cấp để khắc phục vấn đề này, đó là các tác vụ hủy đối tượng (***destructor***), tạo đối tượng bằng cách sao chép từ đối tượng khác (***copy constructor***) và phép gán được định nghĩa lại (***overloaded assignment***). Hai tác vụ đầu không được gọi tường minh bởi người lập trình, chúng sẽ được trình biên dịch gọi lúc cần thiết; riêng tác vụ thứ ba được gọi bởi người lập trình khi cần gán hai đối tượng. Như vậy, việc bổ sung nhằm bảo đảm tính an toàn cho lớp Stack không làm thay đổi vẻ bề ngoài của Stack đối với người sử dụng.

2.4.3.1. *Hàm hủy đối tượng (Destructor)*

Giả sử như người lập trình viết một vòng lặp đơn giản trong đó khai báo một đối tượng ngăn xếp có tên là small và đưa dữ liệu vào. Chẳng hạn chúng ta xem xét đoạn lệnh sau:

```
for (int i=0; i < 1000000; i++) {
   Stack<Entry> small;
   small.push(some_data);
}
```

Trong mỗi lần lặp, đối tượng `small` được tạo ra, dữ liệu thêm vào thuộc vùng bộ nhớ cấp phát động, sau đó đối tượng `small` không còn tồn tại khi ra khỏi tầm vực hoạt động của nó (*scope*). Giả sử chương trình sử dụng ngăn xếp liên kết được hiện thực như hình 2.4. Ngay khi đối tượng `small` không còn tồn tại , dữ liệu trong ngăn xếp trở thành rác, vì bản thân đối tượng `small` chỉ chứa con trỏ `top_node`, vùng nhớ mà con trỏ này chiếm sẽ được trả về cho hệ thống, còn các dữ liệu mà con trỏ này tham chiếu đến thuộc vùng nhớ cấp phát động vẫn chưa được trả về hệ thống. Vòng lặp trên được thực hiện hàng triệu lần, và rác sẽ bị tích lũy rất nhiều. Trong trường hợp này không thể buộc tội người lập trình: do vòng lặp sẽ chẳng gây ra vấn đề gì nếu người lập trình sử dụng hiện thực ngăn xếp liên tục, mọi vùng nhớ dành cho dữ liệu trong ngăn xếp liên tục đều được giải phóng khi ngăn xếp ra khỏi tầm vực.

Một điều chắc chắn rằng khi hiện thực ngăn xếp liên kết, chúng ta cần phải cảnh báo người sử dụng không được để một đối tượng ngăn xếp không rỗng ra khỏi tầm vực, hoặc chúng ta phải làm rỗng ngăn xếp trước khi nó ra khỏi tầm vực.

Ngôn ngữ C++ cung cấp cho lớp phương thức ***destructor*** để giải quyết vấn đề này. Đối với mọi lớp, *destructor* là một phương thức đặc biệt được thực thi cho đối tượng của lớp ngay trước khi đối tượng ra khỏi tầm vực. Người sử dụng không cần phải gọi *destructor* một cách tường minh và thậm chí cũng không cần biết đến sự tồn tại của nó. Đối với người sử dụng, một lớp có *destructor* có thể được thay thế một cách đơn giản bởi một lớp mà không có nó.

Destructor thường được sử dụng để giải phóng các đối tượng cấp phát động mà chúng có thể tạo nên rác. Trong trường hợp của chúng ta, chúng ta nên bổ sung thêm *destructor* cho lớp ngăn xếp liên kết. Sau hiệu chỉnh này, người sử dụng sẽ không thể gây ra rác khi để một đối tượng ngăn xếp không rỗng ra khỏi tầm vực.

Destructor được khai báo như một phương thức của lớp, không có thông số và không có trị trả về. Tên của *destructor* là tên lớp có thêm dấu ~ phía trước.

```
Stack::~Stack();
```

Do phương thức `pop` được lập trình để loại một phần tử ra khỏi ngăn xếp, chúng ta có thể hiện thực *destructor* cho ngăn xếp bằng cách lặp nhiều lần việc gọi `pop`:

```
template <class Entry>
Stack::~Stack() //  Destructor
/*
post: ngăn xếp đã được làm rỗng.
*/
{
   while (!empty())
      pop();
}
```

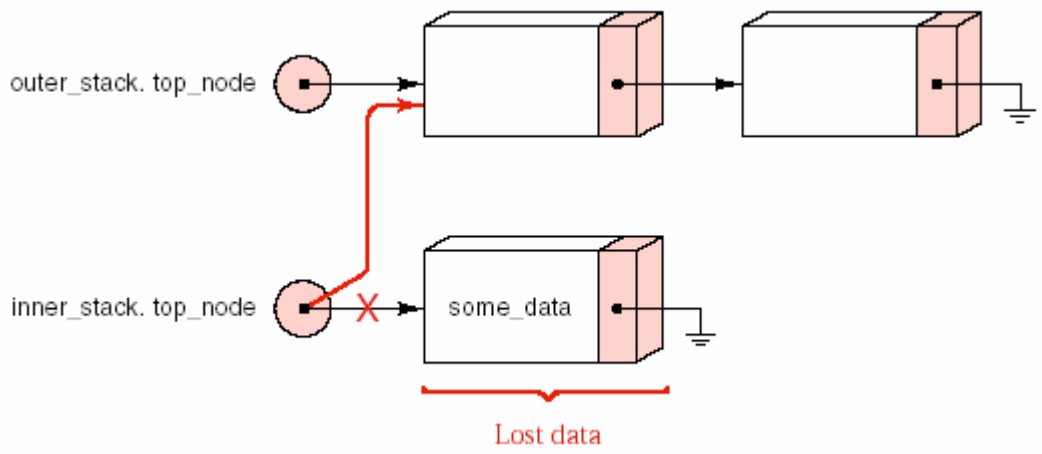
Đối với mọi cấu trúc liên kết chúng ta cần viết destructor để dọn dẹp các đối tượng trước khi chúng ra khỏi tầm vực.

2.4.3.2. *Định nghĩa lại phép gán*

Ngay khi chúng ta đã bổ sung *destructor* cho ngăn xếp liên kết, người sử dụng cũng có thể tạo rác khi viết vòng lặp tựa như ví dụ sau.

```
Stack<Entry> outer_stack;
for (int i=0; i < 1000000; i++) {
   Stack<Entry> inner_stack;
   inner_stack.push(some_data);
   inner_stack = outer_stack;
}
```

Đầu tiên là đối tượng outer_stack được tạo ra, sau đó vòng lặp được thực hiện. Mỗi lần lặp là một lần tạo một đối tượng inner_stack, đưa dữ liệu vào inner_stack, gán outer_stack vào inner_stack. Lệnh gán này gây ra một vấn đề nghiêm trọng cho hiện thực ngăn xếp liên kết của chúng ta. Thông thường, C++ thực hiện phép gán các đối tượng bằng cách chép các thuộc tính của các đối tượng. Do đó, outer_stack.top_node được ghi đè lên inner_stack.top_node, làm cho dữ liệu cũ tham chiếu bởi inner_stack.top_node trở thành rác. Cũng giống như phần trước, nếu hiện thực ngăn xếp liên tục được sử dụng thì không có vấn đề gì xảy ra. Như vậy, lỗi là do hiện thực ngăn xếp liên kết còn thiếu sót.



Hình 2.7- Ứng dụng chép ngăn xếp.

Hình 2.7 cho thấy tác vụ gán không được thực hiện như mong muốn. Sau phép gán, cả hai ngăn xếp cùng chia xẻ các `node`. Cuối mỗi lần lặp, *destructor* của `inner_stack` sẽ giải phóng mọi dữ liệu của `outer_stack`. Việc giải phóng dữ liệu của `outer_stack` còn làm cho `outer_stack.top_node` trở thành tham chiếu treo, có nghĩa là tham chiếu đến vùng nhớ không xác định.

Vấn đề sinh ra bởi phép gán trên ngăn xếp liên kết là do nó chép các tham chiếu chứ không phải chép các trị. Phép gán trong trường hợp này được gọi là phép gán có ngữ nghĩa tham chiếu (*reference semantics*) . Ngược lại, khi phép gán thực sự chép dữ liệu trong CTDL chúng ta gọi là phép gán có ngữ nghĩa trị (*value semantics*). Trong hiện thực lớp ngăn xếp liên kết, hoặc chúng ta phải cảnh báo cho người sử dụng rằng phép gán chỉ là phép gán có ngữ nghĩa tham chiếu, hoặc chúng ta phải làm cho trình biên dịch C++ thực hiện phép gán có ngữ nghĩa trị.

Trong C++, chúng ta hiện thực một phương thức đặc biệt gọi là phép gán được định nghĩa lại (*overloaded assignment*) để định nghĩa lại cách thực hiện phép gán. Khi trình biên dịch C++ dịch một phép gán có dạng x = y, nó ưu tiên chọn phép gán được định nghĩa lại trước nếu như lớp x có định nghĩa. Chỉ khi không có phương thức này, trình biên dịch mới dịch phép gán như là phép gán từng bit đối với các thuộc tính của đối tượng (nếu thuộc tính là con trỏ thì phép gán trở thành phép gán có ngữ nghĩa tham chiếu). Chúng ta cần định nghĩa lại để phép gán cho ngăn xếp liên kết trở thành phép gán có ngữ nghĩa trị.

Có một vài cách để khai báo và hiện thực phép gán được định nghĩa lại. Cách đơn giản là khai báo như sau:

```
void Stack<Entry>::operator= ( const Stack &original );
```

Phép gán này có thể được gọi như sau:

```
x.operator = (y);
```

hoặc gọi theo cú pháp thường dùng:

```
x = y;
```

Phép gán định nghĩa lại cho ngăn xếp liên kết cần làm những việc sau:

- Chép dữ liệu từ ngăn xếp được truyền vào thông qua thông số.
- Giải phóng vùng nhớ chiếm giữ bởi dữ liệu của đối tượng ngăn xếp đang được thực hiện lệnh gán.
- Chuyển các dữ liệu vừa chép được cho đối tượng ngăn xếp được gán.

```
template <class Entry>
void Stack::operator = (const Stack<Entry> &original) //  Overload assignment
/*
post: đối tượng ngăn xếp được gán chứa các dữ liệu giống hệt ngăn xếp được truyền vào qua
      thông số.
*/
{
   Node<Entry> *new_top, *new_copy, *original_node = original.top_node;
   if (original_node == NULL) new_top = NULL;
   else {                          //  Tạo bản sao các node.
      new_copy = new_top = new Node<Entry>(original_node->entry);
```

```
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node<Entry>(original_node->entry);
         new_copy = new_copy->next;
      }
   }
   while (!empty())                // Làm rỗng ngăn xếp sẽ được gán.
      pop();
   top_node = new_top;             // Ngăn xếp được gán sẽ nắm giữ bản sao.
}
```

Lưu ý rằng trong phương thức trên chúng ta tạo ra một bản sao từ ngăn xếp `original` trước, rồi mới dọn dẹp ngăn xếp sẽ được gán bằng cách gọi phương thức `pop` nhiều lần. Nhờ vậy nếu trong ứng dụng có phép gán `x = x` thì dữ liệu cũng không bị sai.

Phép gán được định nghĩa lại như trên thỏa yêu cầu là phép gán có ngữ nghĩa trị, tuy nhiên để có thể sử dụng trong trường hợp: `first_stack=second_stack=third_stack=...`, phép gán phải trả về **`Stack&`** (tham chiếu đến lớp `Stack`).

2.4.3.3. *Copy constructor*

Trường hợp cuối cùng về sự thiếu an toàn của các cấu trúc liên kết là khi trình biên dịch cần chép một đối tượng. Chẳng hạn, một đối tượng cần được chép khi nó là **tham trị** gởi cho một hàm. Trong C++, tác vụ chép mặc nhiên là chép các thuộc tính thành phần của lớp. Cũng giống như minh họa trong hình 2.7, tác vụ chép mặc nhiên này sẽ dẫn đến việc các đối tượng cùng chia xẻ dữ liệu. Nói một cách khác, tác vụ chép mặc định có ngữ nghĩa tham chiếu khi đối tượng có thuộc tính kiểu con trỏ. Điều này làm cho người sử dụng có thể vô tình làm mất dữ liệu:

```
void destroy_the_stack (Stack<Entry> copy)
{
   …
}
int main()
{
   Stack<Entry> vital_data;
   destroy_the_stack(vital_data);
}
```

Hàm `destroy_the_stack` nhận một bản sao `copy` của `vital_data`. Bản sao này cùng chia xẻ dữ liệu với `vital_data`, do đó khi kết thúc hàm, *destructor* thực hiện trên bản sao `copy` sẽ làm mất luôn dữ liệu của `vital_data`.

C++ cho phép lớp có thêm phương thức ***copy constructor*** để tạo một đối tượng mới giống một đối tượng đã có. Nếu chúng ta hiện thực *copy constructor*

cho lớp Stack thì trình biên dịch C++ sẽ ưu tiên chọn tác vụ chép này thay cho tác vụ chép mặc định. Chúng ta cần hiện thực *copy constructor* để có được ngữ nghĩa trị.

Đối với mọi lớp, khai báo chuẩn cho *copy constructor* cũng giống như khai báo *constructor* nhưng có thêm thông số là tham chiếu hằng đến đối tượng của lớp:

```
Stack<Entry>::Stack ( const Stack<Entry> &original );
```

Do đối tượng gọi *copy constructor* là một đối tượng rỗng vừa được tạo mới nên chúng ta không phải lo dọn dẹp dữ liệu như trường hợp đối với phép gán được định nghĩa lại. Chúng ta chỉ việc chép node đầu tiên và sau đó dùng vòng lặp để chép tiếp các node còn lại.

```
template <class Entry>
Stack<Entry>::Stack(const Stack<Entry> &original) //  copy constructor
/*
post: đối tượng ngăn xếp vừa được tạo ra có dữ liệu giống với ngăn xếp original
*/
{
   Node<Entry> *new_copy, *original_node = original.top_node;
   if (original_node == NULL) top_node = NULL;
   else {                          //  Tạo bản sao cho các node.
      top_node = new_copy = new Node<Entry>(original_node->entry);
      while (original_node->next != NULL) {
         original_node = original_node->next;
         new_copy->next = new Node<Entry>(original_node->entry);
         new_copy = new_copy->next;
      }
  a(
}
```

Một cách tổng quát, đối với mọi lớp liên kết, hoặc chúng ta bổ sung *copy constructor*, hoặc chúng ta cảnh báo người sử dụng rằng việc chép đối tượng có ngữ nghĩa tham chiếu.

2.4.4. Đặc tả ngăn xếp liên kết đã hiệu chỉnh

Chúng ta kết thúc phần này bằng đặc tả đã được hiệu chỉnh dưới đây cho ngăn xếp liên kết. Phần đặc tả này có được mọi đặc tính an toàn mà chúng ta đã phân tích.

```
template <class Entry>
class Stack {
public:
//  Các phương thức chuẩn của lớp Stack:
   Stack();
   bool empty() const;
   ErrorCode push(const Entry &item);
   ErrorCode pop();
   ErrorCode top(Entry &item) const;
```

```
//  Các đặc tả đảm bảo tính an toàn cho cấu trúc liên kết:
   ~Stack();
   Stack(const Stack<Entry> &original);
   void operator =(const Stack<Entry> &original);
protected:
   Node<Entry> *top_node;
};
```

Trên đây là phần trình bày đầy đủ nhất về những yêu cầu cần có đối với ngăn xếp liên kết, nhưng nó cũng đúng với các cấu trúc liên kết khác. Trong các phần sau của giáo trình này sẽ không giải thích thêm về 3 tác vụ này nữa, sinh viên tự phải hoàn chỉnh khi hiện thực bất kỳ CTDL nào có thuộc tính kiểu con trỏ.

Chương 3 – HÀNG ĐỢI

3.1. Định nghĩa hàng

Trong các ứng dụng máy tính, chúng ta định nghĩa CTDL hàng là một danh sách trong đó việc thêm một phần tử vào được thực hiện ở một đầu của danh sách (cuối hàng), và việc lấy dữ liệu khỏi danh sách thực hiện ở đầu còn lại (đầu hàng). Chúng ta có thể hình dung CTDL hàng cũng giống như một hàng người lần lượt chờ mua vé, ai đến trước được phục vụ trước. Hàng còn được gọi là danh sách FIFO (*First In First Out*)

Hình 3.1- Hàng đợi

Các ứng dụng có sử dụng hàng còn phổ biến hơn các ứng dụng có sử dụng ngăn xếp, vì khi máy tính thực hiện các nhiệm vụ, cũng giống như các công việc trong cuộc sống, mỗi công việc đều cần phải đợi đến lượt của mình. Trong một hệ thống máy tính có thể có nhiều hàng đợi các công việc đang chờ đến lượt được in, được truy xuất đĩa hoặc được sử dụng CPU. Trong một chương trình đơn giản có thể có nhiều công việc được lưu vào hàng đợi, hoặc một công việc có thể khởi tạo một số công việc khác mà chúng cũng cần được lưu vào hàng để chờ đến lượt thực hiện.

Phần tử đầu hàng sẽ được phục vụ trước, thường phần tử này được gọi là **`front`**, hay **`head`** của hàng. Tương tự, phần tử cuối hàng, cũng là phần tử vừa được thêm vào hàng, được gọi là **`rear`** hay **`tail`** của hàng.

Định nghĩa: Một hàng các phần tử kiểu T là một chuỗi nối tiếp các phần tử của T, kèm các tác vụ sau:
1. Tạo mới một đối tượng hàng rỗng.
2. Thêm một phần tử mới vào hàng, giả sử hàng chưa đầy (phần tử dữ liệu mới luôn được thêm vào cuối hàng).
3. Loại một phần tử ra khỏi hàng, giả sử hàng chưa rỗng (phần tử bị loại là phần tử tại đầu hàng, thường là phần tử vừa được xử lý xong).
4. Xem phần tử tại đầu hàng (phần tử sắp được xử lý).

3.2. Đặc tả hàng

Để hoàn tất định nghĩa của cấu trúc dữ liệu trừu tượng hàng, chúng ta đặc tả mọi tác vụ mà hàng thực hiện. Các đặc tả này cũng tương tự như các đặc tả cho ngăn xếp, chúng ta đưa ra tên, kiểu trả về, danh sách thông số, *precondition*, *postcondition* và *uses* cho mỗi phương thức. **`Entry`** biểu diễn một kiểu tổng quát cho phần tử chứa trong hàng.

```
template <class Entry>
Queue<Entry>::Queue();
```
post: đối tượng hàng đã tồn tại và được khởi tạo là hàng rỗng.

```
template <class Entry>
ErrorCode Queue<Entry>::append(const Entry &item);
```
post: nếu hàng còn chỗ, `item` được thêm vào tại `rear`, `ErrorCode` trả về là `success`; ngược lại, `ErrorCode` trả về là `overflow`, hàng không đổi.

```
template <class Entry>
ErrorCode Queue<Entry>::serve();
```
post: nếu hàng không rỗng, phần tử tại `front` được lấy đi, `ErrorCode` trả về là `success`; ngược lại, `ErrorCode` trả về là `underflow`, hàng không đổi.

```
template <class Entry>
ErrorCode Queue<Entry>::retrieve(const Entry &item) const;
```
post: nếu hàng không rỗng, phần tử tại `front` được chép vào `item`, `ErrorCode` trả về là `success`; ngược lại, `ErrorCode` trả về là `underflow`; cả hai trường hợp hàng đều không đổi.

```
template <class Entry>
bool Queue<Entry>::empty() const;
```
post: hàm trả về `true` nếu hàng rỗng; ngược lại, hàm trả về `false`.

Từ **`append`** (thêm vào hàng) và **`serve`** (đã được phục vụ) được dùng cho các tác vụ cơ bản trên hàng để chỉ ra một cách rõ ràng công việc thực hiện đối với hàng,

và để tránh nhầm lẫn với những từ mà chúng ta sẽ dùng với các cấu trúc dữ liệu khác.

Chúng ta có lớp Queue như sau:

```
template <class Entry>
class Queue {
public:
   Queue();
   bool empty() const;
   ErrorCode append(const Entry &item);
   ErrorCode serve();
   ErrorCode retrieve(Entry &item) const;
};
```

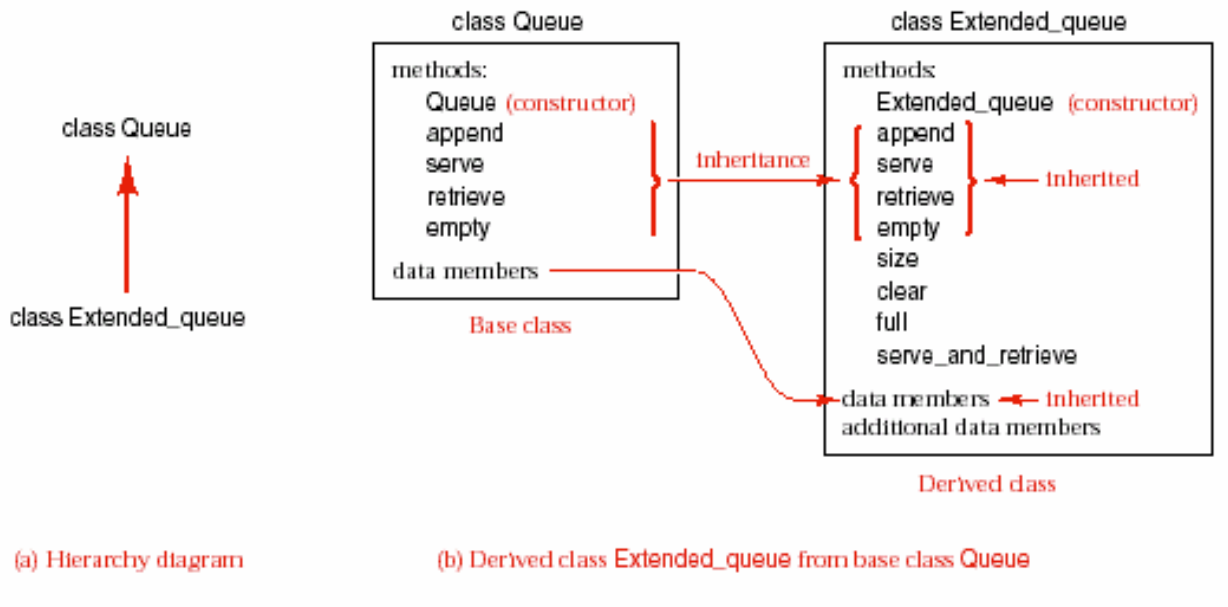
Ngoài các tác vụ cơ bản như **append**, **serve**, **retrieve**, và **empty** đôi khi chúng ta cần thêm một số tác vụ khác. Chẳng hạn như tác vụ **full** để kiểm tra xem hàng đã đầy hay chưa.

Có ba tác vụ rất tiện lợi đối với hàng: **clear** để dọn dẹp các phần tử trong một hàng có sẵn và làm cho hàng rỗng, **size** cho biết số phần tử hiện có trong hàng, cuối cùng là **serve_and_retrieve** gom hai tác vụ **serve** và **retrieve** làm một vì người sử dụng thường gọi hai tác vụ này một lúc.

Chúng ta có thể bổ sung các tác vụ trên vào lớp hàng đã có ở trên. Tuy nhiên, chúng ta có thể tạo lớp mới có thể sử dụng lại các phương thức và cách hiện thực của các lớp đã có. Trong trường hợp này chúng ta xây dựng lớp **Extended_Queue** để bổ sung các phương thức thêm vào các phương thức cơ bản của lớp Queue. Lớp Extended_Queue được gọi là lớp dẫn xuất từ lớp Queue.

Khái niệm dẫn xuất cung cấp một cách định nghĩa các lớp mới đơn giản bằng cách bổ sung thêm các phương thức vào một lớp có sẵn. Khả năng của lớp dẫn xuất sử dụng lại các thành phần của lớp cơ sở được gọi là sự thừa kế. Sự thừa kế (*inheritance*) là một trong các đặc tính cơ bản của lập trình hướng đối tượng.

Chúng ta minh họa mối quan hệ giữa lớp Queue và lớp dẫn xuất Extended_Queue bởi sơ đồ thừa kế (hình 3.2a). Mũi tên chỉ từ lớp dẫn xuất đến lớp cơ sở mà nó thừa kế. Hình 3.2b minh họa sự thừa kế các phương thức và các phương thức bổ sung.



Hình 3.2- Sự thừa kế và lớp dẫn xuất

Chúng ta có lớp `Extended_Queue`:

```
template <class Entry>
class Extended_Queue: public Queue {
public:
   bool full() const;
   int size() const;
   void clear();
   ErrorCode serve_and_retrieve(Entry &item);
};
```

Từ khóa **public** trong khai báo thừa kế có nghĩa là khả năng người sử dụng nhìn thấy đối với các thành phần mà lớp dẫn xuất có được qua sự thừa kế sẽ giống hệt như khả năng người sử dụng nhìn thấy chúng ở lớp cơ sở.

Đặc tả của các phương thức bổ sung:

```
template <class Entry>
bool Extended_Queue<Entry>::full() const;
```
`post`: trả về `true` nếu hàng đầy, ngược lại, trả về `false`. Hàng không đổi.

```
template <class Entry>
void Extended_Queue<Entry>::clear();
```
post: mọi phần tử trong hàng được loại khỏi hàng, hàng trở nên rỗng.

```
template <class Entry>
int Extended_Queue<Entry>::size() const;
```
post: trả về số phần tử hiện có của hàng. Hàng không đổi.

```
template <class Entry>
ErrorCode Extended_Queue<Entry>::serve_and_retrieve(const Entry &item);
```
post: nếu hàng không rỗng, phần tử tại `front` được chép vào `item` đồng thời được loại khỏi hàng, `ErrorCode` trả về là `success`; ngược lại, `ErrorCode` trả về là `underflow`, hàng không đổi.

Mối quan hệ giữa lớp `Extended_Queue` và lớp `Queue` thường được gọi là mối quan hệ ***is-a*** vì mỗi đối tượng thuộc lớp `Extended_Queue` cũng là một đối tượng thuộc lớp `Queue` mà có thêm một số đặc tính khác, đó là các phương thức `serve_and_retrieve, full, size` và `clear.`

3.3. Các phương án hiện thực hàng

3.3.1. Các phương án hiện thực hàng liên tục

3.3.1.1. Mô hình vật lý

Tương tự như chúng ta đã làm với ngăn xếp, chúng ta có thể tạo một hàng trong bộ nhớ máy tính bằng một dãy (kiểu dữ liệu `array`) để chứa các phần tử của hàng. Tuy nhiên, ở đây chúng ta cần phải nắm giữ được cả **`front`** và **`rear.`** Một cách đơn giản là chúng ta giữ `front` luôn là vị trí đầu của dãy. Lúc đó, để thêm mới một phần tử vào hàng, chúng ta tăng biến đếm biểu diễn `rear` y hệt như chúng ta thêm phần tử vào ngăn xếp. Để lấy một phần tử ra khỏi hàng, chúng ta phải trả một giá đắt cho việc di chuyển tất cả các phần tử hiện có trong hàng tới một bước để lấp đầy chỗ trống tại `front.` Mặc dù cách hiện thực này rất giống với hình ảnh hàng người sắp hàng đợi để được phục vụ, nhưng nó là một lựa chọn rất dở trong máy tính.

3.3.1.2. Hiện thực tuyến tính

Để việc xử lý hàng có hiệu quả, chúng ta dùng hai chỉ số để nắm giữ `front` và `rear` mà không di chuyển các phần tử. Muốn thêm một phần tử vào hàng, đơn giản chúng ta chỉ cần tăng `rear` lên một và thêm phần tử vào vị trí này. Khi lấy một phần tử ra khỏi hàng chúng ta lấy phần tử tại vị trí `front` và tăng `front` lên một. Tuy nhiên phương pháp này có một nhược điểm lớn, đó là `front` và `rear` luôn luôn tăng chứ không giảm. Ngay cả khi trong hàng không bao giờ có quá hai phần tử, hàng vẫn đòi hỏi một vùng nhớ không có giới hạn nếu như các tác vụ được gọi liên tục như sau:

```
append, append, serve, append, serve, append, serve, append, serve, append, ...
```

Vấn đề ở đây là khi các phần tử trong hàng dịch chuyển tới trong dãy thì các vị trí đầu của dãy sẽ không bao giờ được sử dụng đến. Chúng ta có thể hình dung

hàng lúc đó trông như một con rắn luôn trườn mình tới. Con rắn có lúc dài ra, có lúc ngắn lại, nhưng nếu cứ trườn tới mãi theo một hướng thì cũng phải đến lúc nó gặp điểm dừng của bộ nhớ.

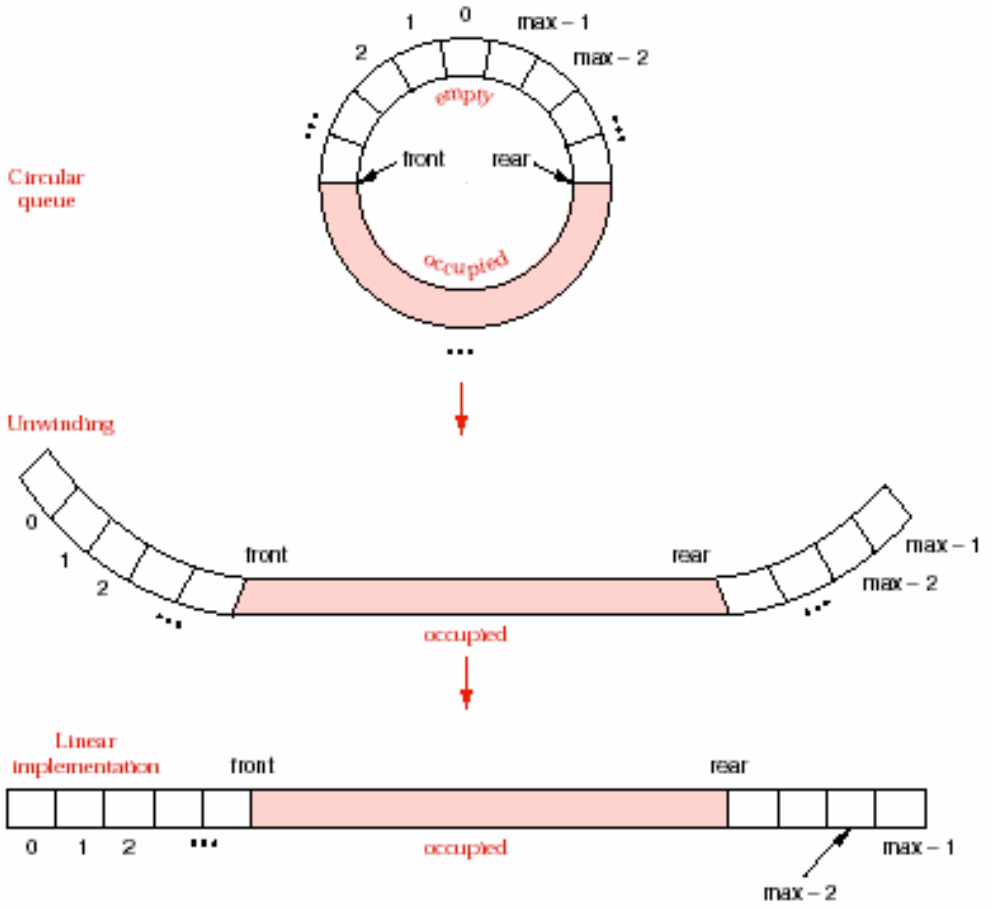
Tuy nhiên, cũng cần chú ý rằng trong các ứng dụng mà có lúc hàng trở nên rỗng (khi một loạt các yêu cầu đang đợi đã được giải quyết hết tại một thời điểm nào đó), thì tại thời điểm này hàng có thể được sắp xếp lại, `front` và `rear` được gán trở lại về đầu dãy. Trường hợp này cho thấy việc sử dụng một sơ đồ đơn giản gồm hai chỉ số và một bộ nhớ tuyến tính như vừa nêu là một cách hiện thực có hiệu quả cao.

3.3.1.3. Dãy vòng

Về ý niệm, chúng ta có thể khắc phục tính thiếu hiệu quả trong việc sử dụng bộ nhớ bằng cách hình dung dãy có dạng vòng thay vì tuyến tính. Khi phần tử được thêm vào hay lấy ra khỏi hàng, điểm đầu của hàng sẽ đuổi theo điểm cuối của hàng vòng theo dãy, và như vậy con rắn vẫn có thể trườn tới vô hạn nhưng vẫn bị nhốt trong một vòng có giới hạn. Tại các thời điểm khác nhau, hàng sẽ chiếm những phần khác nhau trong dãy vòng, nhưng chúng ta sẽ không bao giờ phải lo về sự vượt giới hạn bộ nhớ trừ khi dãy thật sự không còn phần tử trống, trường hợp này được xem như hàng đầy, `ErrorCode` sẽ nhận trị `overflow`.

Hiện thực của dãy vòng

Vấn đề tiếp theo của chúng ta là dùng một dãy tuyến tính để mô phỏng một dãy vòng. Các vị trí trong vòng tròn được đánh số từ 0 đến max-1, trong đó max là tổng số phần tử trong dãy vòng. Để hiện thực dãy vòng, chúng ta cũng sử dụng các phần tử được đánh số tương tự dãy tuyến tính. Sự thay đổi các chỉ số chỉ đơn giản là phép lấy phần dư trong số học: khi một chỉ số tăng vượt qua giới hạn `max`-1, nó được bắt đầu trở lại với trị 0. Điều này tương tự việc cộng thêm giờ trong đồng hồ mặt tròn, các giờ được đánh số từ 1 đến 12, nếu chúng ta cộng thêm 4 giờ vào 10 giờ chúng ta sẽ có 2 giờ.



Hình 3.3- Hàng trong dãy vòng

Dãy vòng trong C++

Trong C++, chúng ta có thể tăng chỉ số *i* trong một dãy vòng như sau:

```
i = ((i+1) == max) ? 0: (i+1);
```

hoặc

```
if ((i+1) == max)  i = 0; else i = i+1;
```

hoặc

```
i = (i+1) % max;
```

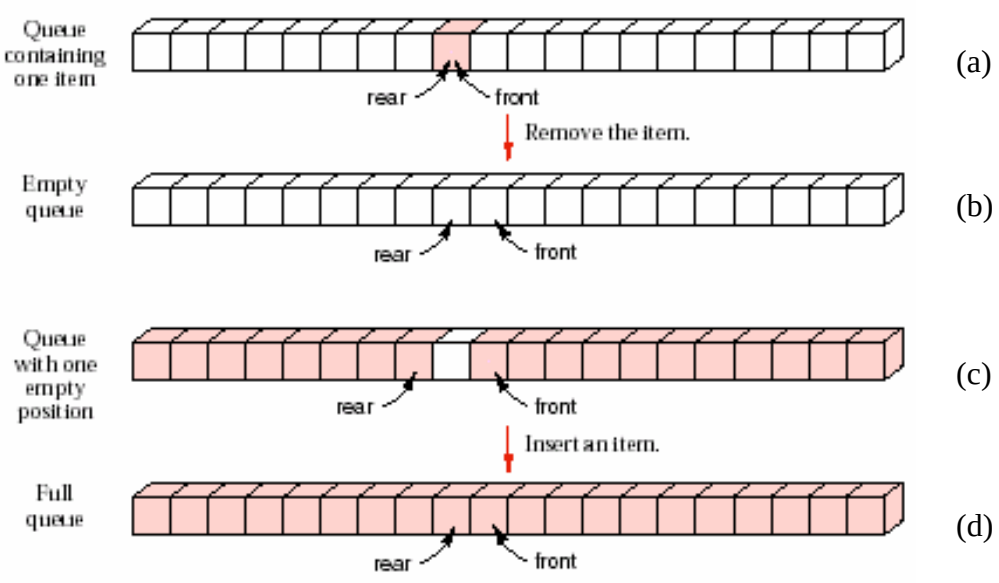
Các điều kiện biên

Trước khi viết những giải thuật thêm hoặc loại phần tử ra khỏi hàng, chúng ta hãy xem xét đến các điều kiện biên (*boundary conditions*), đó là các dấu hiệu cho biết hàng còn rỗng hay đã đầy.

Nếu trong hàng chỉ có một phần tử thì cả `front` và `rear` đều chỉ đến phần tử này (hình 3.4 a). Khi phần tử này được loại khỏi hàng, `front` sẽ tăng lên 1. Do đó hàng là rỗng khi `rear` chỉ vị trí ngay trước `front` (hình 3.4 b).

Do `rear` di chuyển về phía trước mỗi khi thêm phần tử mới, nên khi hàng sắp đầy và bằng cách di chuyển vòng thì `rear` cũng sẽ gần gặp `front` trở lại (hình 3.3 c). Lúc này khi phần tử cuối cùng được thêm vào làm cho hàng đầy thì `rear` cũng chỉ vị trí ngay trước `front` (hình 3.4 d).

Chúng ta gặp một khó khăn mới: vị trí tương đối của `front` và `rear` giống hệt nhau trong cả hai trường hợp hàng đầy và hàng rỗng.



Hình 3.4- Hình ảnh minh họa hàng rỗng và hàng đầy

Các cách giải quyết có thể

Có ít nhất 3 cách giải quyết cho vấn đề nêu trên. Cách thứ nhất là dành lại một vị trí trống khi hàng đầy, `rear` sẽ cách `front` một vị trí giữa. Cách thứ hai là sử dụng thêm một biến, chẳng hạn một biến cờ kiểu `bool` sẽ có trị `true` khi `rear` nhích đến sát `front` trong trường hợp hàng đầy (chúng ta có thể tùy ý chọn trường hợp hàng đầy hay rỗng), hay một biến đếm để đếm số phần tử hiện có trong hàng. Cách thứ ba là cho một hoặc cả hai chỉ số `front` và `rear` mang một trị đặc biệt nào đó để chỉ ra hàng rỗng, ví dụ như `rear` sẽ là -1 khi hàng rỗng.

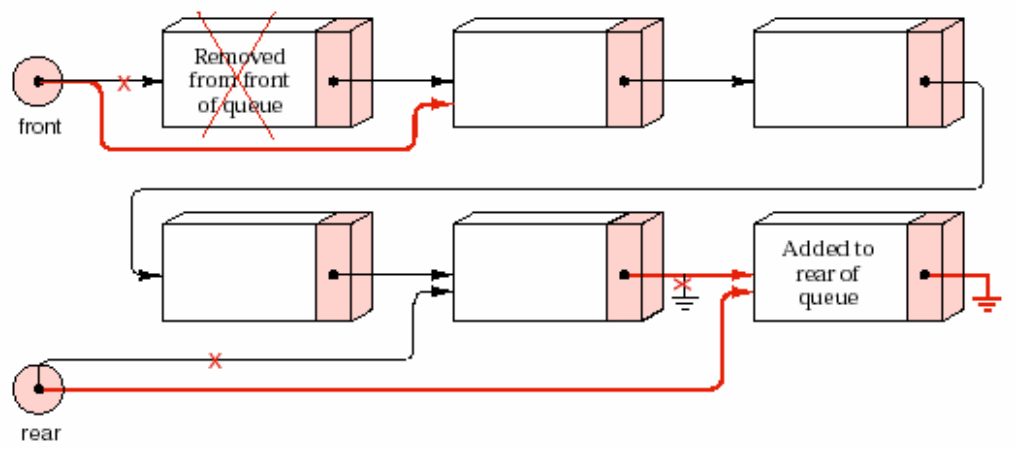
3.3.1.4. Tổng kết các cách hiện thực cho hàng liên tục

Để tổng kết những điều đã bàn về hàng, chúng ta liệt kê dưới đây tất cả các phương pháp mà chúng ta đã thảo luận về các cách hiện thực hàng.

- Mô hình vật lý: một dãy tuyến tính có `front` luôn chỉ vị trí đầu tiên trong hàng và mọi phần tử của hàng phải di chuyển tới một bước khi phần tử tại `front` được lấy đi. Đây là phương pháp rất dở trong máy tính nói chung.
- Một dãy tuyến tính có hai chỉ số `front` và `rear` luôn luôn tăng. Đây là phương pháp tốt nếu như hàng có thể được làm rỗng.
- Một dãy vòng có hai chỉ số `front,rear` và một vị trí để trống.
- Một dãy vòng có hai chỉ số `front,rear` và một cờ cho biết hàng đầy (hoặc rỗng).
- Một dãy vòng có hai chỉ số `front,rear` và một biến đếm số phần tử hiện có trong hàng.
- Một dãy vòng có hai chỉ số `front,rear` mà hai chỉ số này sẽ mang trị đặc biệt trong trường hợp hàng rỗng.

3.3.2. Phương án hiện thực hàng liên kết

Bằng cách sử dụng bộ nhớ liên tục, việc hiệc thực hàng khó hơn việc hiện thực ngăn xếp rất nhiều do chúng ta dùng vùng nhớ tuyến tính để giả lặp tổ chức vòng và gặp khó khăn trong việc phân biệt một hàng đầy với một hàng rỗng. Tuy nhiên, hiện thực hàng liên kết lại thực sự dễ dàng như hiện thực ngăn xếp liên kết. Chúng ta chỉ cần nắm giữ hai con trỏ, `front` và `rear` để tham chiếu đến phần tử đầu và phần tử cuối của hàng. Các tác vụ thêm hay loại phần tử trên hàng được minh họa trong hình 3.5.



Hình 3.5 Các tác vụ thêm và loại phần tử trên hàng liên kết

3.4. Hiện thực hàng

3.4.1. Hiện thực hàng liên tục

Hiện thực vòng cho hàng liên tục trong C++

Phần này trình bày các phương thức của cách hiện thực hàng bằng dãy vòng có biến đếm các phần tử. Chúng ta có định nghĩa lớp `Queue` như sau:

```
const int maxQueue = 10; //  Giá trị nhỏ chỉ để kiểm tra CTDL Queue.

template <class Entry>
class Queue {
public:
   Queue();
   bool empty() const;
   ErrorCode serve();
   ErrorCode append(const Entry &item);
   ErrorCode retrieve(Entry &item) const;
protected:
   int count;
   int front, rear;
   Entry entry[maxQueue];
};
```

Các dữ liệu thành phần trong lớp `Queue` được khai báo **`protected.`** Đối với người sử dụng sẽ không có gì thay đổi, nghĩa là chúng vẫn không được người sử dụng nhìn thấy và vẫn đảm bảo sự che dấu thông tin. Mục đích ở đây là khi chúng ta xây dựng lớp `Extended_Queue` dẫn xuất từ lớp `Queue` thì lớp dẫn xuất sẽ sử dụng được các dữ liệu thành phần này. Khi các dữ liệu thành phần của lớp cơ sở được khai báo là **`private`** thì lớp dẫn xuất cũng sẽ không nhìn thấy chúng.

```
template <class Entry>
Queue<Entry>::Queue()
/*
post: đối tượng hàng đã tồn tại và được khởi tạo là hàng rỗng.
*/
```

```
{
   count = 0;
   rear = maxQueue - 1;
   front = 0;
}
```

```
template <class Entry>
bool Queue<Entry>::empty() const
/*
post: hàm trả về true nếu hàng rỗng; ngược lại, hàm trả về false.
*/
{   return count == 0;
}
```

```
template <class Entry>
ErrorCode Queue<Entry>::append(const Entry &item)
/*
post: nếu hàng còn chỗ, item được thêm vào tại rear, ErrorCode trả về là success; ngược lại,
     ErrorCode trả về là overflow, hàng không đổi.
*/
{
   if (count >= maxQueue) return overflow;
   count++;
   rear = ((rear + 1) == maxQueue) ? 0 : (rear + 1);
   entry[rear] = item;
   return success;
}
template <class Entry>
ErrorCode Queue<Entry>::serve()
/*
post: nếu hàng không rỗng, phần tử tại front được lấy đi, ErrorCode trả về là success; ngược
      lại, ErrorCode trả về là underflow, hàng không đổi.
*/
{
   if (count <= 0) return underflow;
   count--;
   front = ((front + 1) == maxQueue) ? 0 : (front + 1);
   return success;
}
```

```
template <class Entry>
ErrorCode Queue<Entry>::retrieve(Entry &item) const
/*
post: nếu hàng không rỗng, phần tử tại front được chép vào item, ErrorCode trả về là
      success; ngược lại, ErrorCode trả về là underflow; cả hai trường hợp hàng đều không
      đổi.
*/
{
   if (count <= 0) return underflow;
   item = entry[front];
   return success;
}
```

Chúng ta dành phương thức empty lại như là bài tập. Phương thức size dưới đây rất đơn giản do lớp Extended_Queue sử dụng được thuộc tính count của lớp

Queue, nếu như count được khai báo là private thì phương thức size của lớp Extended_Queue phải sử dụng hàng loạt câu lệnh gọi các phương thức public của Queue như serve, retrieve, append mới thực hiện được. Các phương thức còn lại của Extended_Queue cũng tương tự, và chúng ta dành lại phần bài tập.

```
template <class Entry>
int Extended_Queue<Entry>::size() const
/*
post: trả về số phần tử hiện có của hàng. Hàng không đổi.
*/
{
   return count;
}
```

3.4.2. Hiện thực hàng liên kết

3.4.2.1. Các khai báo cơ bản

Với mọi hàng, chúng ta dùng kiểu Entry cho kiểu của dữ liệu lưu trong hàng. Đối với hiện thực liên kết, chúng ta khai báo các **node** tương tự như đã làm cho ngăn xếp liên kết trong chương 2. Chúng ta có đặc tả dưới đây:

```
template <class Entry>
class Queue {
public:
//  Các phương thức chuẩn của hàng
   Queue();
   bool empty() const;
   ErrorCode append(const Entry &item);
   ErrorCode serve();
   ErrorCode retrieve(Entry &item) const;
//  Các phương thức bảo đảm tính an toàn cho hàng liên kết
   ~Queue();
   Queue(const Queue<Entry> &original);
   void operator =(const Queue<Entry> &original);
protected:
   Node<Entry> *front, *rear;
};
```

Constructor thứ nhất khởi tạo một hàng rỗng:

```
template <class Entry>
Queue<Entry>::Queue()
/*
post: đối tượng hàng đã tồn tại và được khởi tạo là hàng rỗng.
*/
{
   front = rear = NULL;
}
```

Phương thức *append* thêm một phần tử vào đầu *rear* của hàng:

```
template <class Entry>
ErrorCode Queue<Entry>::append(const Entry &item)
/*
post: nếu hàng còn chỗ, item được thêm vào tại rear, ErrorCode trả về là success; ngược
      lại, ErrorCode trả về là overflow, hàng không đổi.
*/
{
   Node *new_rear = new Node<Entry> (item);
   if (new_rear == NULL) return overflow;
   if (rear == NULL) front = rear = new_rear;   // trường hợp đặc biệt: thêm vào hàng
                                                   đang rỗng.
   else {
      rear->next = new_rear;
      rear = new_rear;
   }
   return success;
}
```

Trường hợp hàng rỗng cần phân biệt với các trường hợp bình thường khác, do khi thêm một node vào một hàng rỗng cần phải gán cả front và rear tham chiếu đến node này, trong khi việc thêm một node vào một hàng không rỗng chỉ có rear là cần được gán lại.

Phương thức loại một phần tử ra khỏi hàng được viết như sau:

```
template <class Entry>
ErrorCode Queue::serve()
/*
post: nếu hàng không rỗng, phần tử tại front được lấy đi, ErrorCode trả về là success; ngược
      lại, ErrorCode trả về là underflow, hàng không đổi.
*/
{
   if (front == NULL) return underflow;
   Node *old_front = front;
   front = old_front->next;
   if (front == NULL) rear= NULL;// trường hợp đặc biệt: loại phần tử cuối cùng của hàng
   delete old_front;
   return success;
}
```

Một lần nữa trường hợp hàng rỗng cần được xem xét riêng. Khi phần tử được loại khỏi hàng không phải là phần tử cuối trong hàng, chỉ có front cần được gán lại, ngược lại cả front và rear cần phải gán về NULL.

Các phương thức khác của hàng liên kết được dành lại cho phần bài tập.

Nếu so sánh với hàng liên tục, chúng ta sẽ thấy rằng hàng liên kết dễ hiểu hơn cả về mặt khái niệm cả về cách hiện thực chương trình.

3.4.3. Hàng liên kết mở rộng

Hiện thực liên kết của lớp Queue cung cấp một lớp cơ sở cho các lớp khác. Định nghĩa dưới đây dành cho lớp dẫn xuất Extended_Queue hoàn toàn tương tự như hàng liên tục.

```
template <class Entry>
class Extended_queue: public Queue {
public:
   bool full() const;
   int size() const;
   void clear();
   ErrorCode serve_and_retrieve(Entry &item);
};
```

Mặc dù lớp Extended_Queue này có hiện thực liên kết, nhưng nó không cần các phương thức như *copy contructor*, *overloaded assignment*, hoặc *destructor*. Đối với một trong các phương thức này, trình biên dịch sẽ gọi các phương thức mặc định của lớp Extended_Queue. Phương thức mặc định của một lớp dẫn xuất sẽ gọi phương thức tương ứng của lớp cơ sở. Chẳng hạn, khi một đối tượng của lớp Extended_Queue chuẩn bị ra khỏi tầm vực, *destructor* mặc định của lớp Extended_Queue sẽ gọi *destructor* của lớp Queue: mọi node cấp phát động của Extended_Queue đều được giải phóng. Do lớp Extended_Queue của chúng ta không chứa thêm thuộc tính con trỏ nào ngoài các thuộc tính con trỏ thừa kế từ lớp Queue, *destructor* mà trình biên dịch gọi đã làm được tất cả những điều mà chúng ta mong muốn.

Các phương thức được khai báo cho lớp Extended_Queue cần được viết lại cho phù hợp với cấu trúc liên kết của hàng. Chẳng hạn, phương thức *size* cần sử dụng một con trỏ tạm window để duyệt hàng (nói cách khác, con trỏ window sẽ di chuyển dọc theo hàng và lần lượt chỉ đến từng node trong hàng).

```
template <class Entry>
int Extended_queue<Entry>::size() const
/*
post: trả về số phần tử hiện có của hàng. Hàng không đổi.
*/
{
   Node *window = front;
   int count = 0;
   while (window != NULL) {
      window = window->next;
      count++;
   }
   return count;
}
```

Các phương thức khác của Extended_Queue liên kết xem như bài tập.

Chương 4 – DANH SÁCH

Chúng ta đã làm quen với các danh sách hạn chế như ngăn xếp và hàng, trong đó việc thêm/ bớt dữ liệu chỉ thực hiện ở các đầu của danh sách. Trong chương này chúng ta tìm hiểu các danh sách thông thường hơn mà trong đó việc thêm, loại hoặc truy xuất phần tử có thể thực hiện tại bất kỳ vị trí nào trong danh sách.

4.1. Định nghĩa danh sách

Chúng ta bắt đầu bằng việc định nghĩa kiểu cấu trúc dữ liệu trừu tượng gọi là danh sách (*list*). Cũng giống như ngăn xếp và hàng, danh sách bao gồm một chuỗi nối tiếp các phần tử dữ liệu. Tuy nhiên, khác với ngăn xếp và hàng, danh sách cho phép thao tác trên mọi phần tử.

Định nghĩa: Danh sách các phần tử kiểu T là một chuỗi nối tiếp hữu hạn các phần tử kiểu T cùng các tác vụ sau:

1. Tạo một danh sách rỗng.
2. Xác định danh sách có rỗng hay không.
3. Xác định danh sách có đầy hay chưa.
4. Tìm số phần tử của danh sách.
5. Làm rỗng danh sách.
6. Thêm phần tử vào một vị trí nào đó của danh sách.
7. Loại phần tử tại một vị trí nào đó của danh sách.
8. Truy xuất phần tử tại một vị trí nào đó của danh sách.
9. Thay thế phần tử tại một vị trí nào đó của danh sách.
10. Duyệt danh sách, thực hiện một công việc cho trước trên mỗi phần tử.

Ngoài ra còn một số tác vụ khác có thể áp lên một chuỗi nối tiếp các phần tử. Chúng ta có thể xây dựng rất nhiều dạng khác nhau cho các kiểu cấu trúc dữ liệu trừu tượng tương tự bằng cách sử dụng các gói tác vụ khác nhau. Bất kỳ một trong các dạng này đều có thể được định nghĩa cho tên gọi CTDL danh sách. Tuy nhiên, chúng ta chỉ tập trung vào một danh sách cụ thể mà các tác vụ của nó có thể được xem như một khuôn mẫu để minh họa ý tưởng và các vấn đề cần giải quyết trên danh sách.

4.2. Đặc tả các phương thức cho danh sách

Khi bắt đầu tìm hiểu ngăn xếp, chúng ta nhấn mạnh việc che dấu thông tin bằng cách phân biệt giữa việc sử dụng ngăn xếp và việc lập trình cho các tác vụ trên ngăn xếp. Đối với hàng, chúng ta tiếp tục ý tưởng này và đã nhanh chóng tìm được rất nhiều cách hiện thực có thể có. Các danh sách thông dụng cho phép truy xuất và thay đổi bất kỳ phần tử nào. Do đó nguyên tắc che dấu thông tin đối

với danh sách càng quan trọng hơn nhiều so với ngăn xếp và hàng. Chúng ta hãy đặc tả cho các tác vụ trên danh sách:

Constructor cần có trước khi danh sách được sử dụng:

```
template <class Entry>
List<Entry>::List();
```
post: đối tượng danh sách rỗng đã được tạo.

Tác vụ thực hiện trên một danh sách đã có và làm rỗng danh sách:

```
template <class Entry>
void List<Entry>::clear();
```
post: Mọi phần tử của danh sách đã được giải phóng, danh sách trở nên rỗng.

Các tác vụ xác định trạng thái của danh sách:

```
template <class Entry>
bool List<Entry>::empty() const;
```
post: trả về *true* nếu danh sách rỗng, ngược lại trả về *false*. Danh sách không đổi.

```
template <class Entry>
bool List<Entry>::full() const;
```
post: trả về *true* nếu danh sách đầy, ngược lại trả về *false*. Danh sách không đổi.

```
template <class Entry>
int List<Entry>::size() const;
```
post: trả về số phần tử của danh sách. Danh sách không đổi.

Chúng ta xem xét tiếp các tác vụ truy xuất các phần tử của danh sách. Tương tự như đối với ngăn xếp và hàng, các tác vụ này sẽ trả về `ErrorCode` khi cần thiết.

Chúng ta dùng một số nguyên để chỉ vị trí (`position`) của phần tử trong danh sách. Vị trí ở đây được hiểu là thứ tự của phần tử trong danh sách. Các vị trí trong danh sách được đánh số 0, 1, 2, ...Việc xác định một phần tử trong danh sách thông qua vị trí rất giống với sự sử dụng chỉ số trong dãy, tuy nhiên vẫn có một số điểm khác nhau quan trọng. Nếu chúng ta thêm một phần tử vào một vị trí nào đó trong danh sách thì vị trí của tất cả các phần tử phía sau sẽ tăng lên 1. Nếu loại một phần tử thì vị trí các phần tử phía sau giảm 1. Vị trí của các phần tử trong danh sách được xác định không xét đến cách hiện thực. Đối với danh sách liên tục, hiện thực bằng dãy, vị trí phần tử rõ ràng là chỉ số của phần tử trong dãy. Nhưng chúng ta cũng vẫn thông qua vị trí để tìm các phần tử trong danh sách liên kết dù rằng danh sách liên kết không có chỉ số.

Chúng ta sẽ đặc tả chính xác các phương thức liên quan đến chỉ một phần tử của danh sách dưới đây.

template <class Entry>
ErrorCode List<Entry>**::insert(int position, const Entry &x);**
post: Nếu danh sách chưa đầy và `0 ≤ position ≤ n`, `n` là số phần tử hiện có của danh sách, phương thức trả về `success`: mọi phần tử từ `position` đến cuối danh sách sẽ có vị trí tăng lên 1, `x` được thêm vào tại `position`; ngược lại, danh sách không đổi, `ErrorCode` sẽ cho biết lỗi cụ thể.

Phương thức `insert` chấp nhận `position` bằng `n` vì nó chấp nhận thêm phần tử mới ngay sau phần tử cuối. Tuy nhiên, các phương thức sau chỉ chấp nhận `position<n`, vì chúng chỉ thực hiện trên những phần tử đã có sẵn.

template <class Entry>
ErrorCode List<Entry>**::remove(int position, Entry &x);**
post: Nếu `0 ≤ position < n`, `n` là số phần tử hiện có của danh sách, phương thức trả về `success`: phần tử tại `position` được loại khỏi danh sách, trị của nó được chép vào `x`, các phần tử phía sau giảm vị trí bớt 1; ngược lại, danh sách không đổi, `ErrorCode` sẽ cho biết lỗi cụ thể.

template <class Entry>
ErrorCode List<Entry>**::retrieve(int position, Entry &x) const;**
post: Nếu 0 ≤ `position < n`, `n` là số phần tử hiện có của danh sách, phương thức trả về `success`: phần tử tại `position` được chép vào `x`, danh sách không đổi; ngược lại, `ErrorCode` sẽ cho biết lỗi cụ thể. Cả hai trường hợp danh sách đều không đổi.

template <class Entry>
ErrorCode List<Entry>**::replace(int position, const Entry &x);**
post: Nếu 0 ≤ `position < n`, `n` là số phần tử hiện có của danh sách, phương thức trả về `success`: phần tử tại `position` được thay thế bởi `x`; ngược lại, danh sách không đổi, `ErrorCode` sẽ cho biết lỗi cụ thể.

Phương thức duyệt danh sách để thực hiện một nhiệm vụ nào đó cho từng phần tử của danh sách thường tỏ ra có lợi, đặc biệt cho mục đích kiểm tra. Người sử dụng gọi phương thức này khi muốn thực hiện một công việc gì đó trên từng phần tử của danh sách. Chẳng hạn, người sử dụng có hai hàm

`void` **`update`**`(List_Entry &x)` và `void` **`modify`**`(List_Entry &x),`

và một đối tượng `the_list` của lớp `List`, có thể sử dụng lệnh

`the_list.`**`traverse`**`(update)` hoặc `the_list.`**`traverse`**`(modify)`

để thực hiện một trong hai hàm trên lên mỗi phần tử của danh sách. Nếu người sử dụng muốn in mọi phần tử của danh sách thì gọi như sau:

```
the_list.traverse(print)
```

với `void print(Entry &x)` là một hàm dùng để in một phần tử của danh sách.

Khi gọi phương thức `traverse`, người sử dụng gởi tên của hàm làm thông số. Trong C++, tên của hàm mà không có cặp dấu ngoặc chính là con trỏ chỉ đến hàm. Thông số hình thức **`visit`** dưới đây của phương thức **`traverse`** cần được khai báo như một con trỏ chỉ đến hàm. Ngoài ra, khai báo con trỏ hàm làm thông số phải có kiểu trả về là `void` và có thông số tham chiếu đến `Entry`.

```
template <class Entry>
void List<Entry>::traverse(void(*visit)(Entry &x));
```
post: Công việc đặc tả bởi hàm `*visit` được thực hiện lần lượt trên từng phần tử của danh sách, bắt đầu từ phần tử thứ 0.

Cũng giống như mọi thông số khác, `visit` chỉ là tên hình thức và chỉ được gán bởi một con trỏ thực sự khi `traverse` bắt đầu thực thi. Biểu diễn **`*visit`** thay mặt cho hàm sẽ được sử dụng để xử lý cho từng phần tử của danh sách khi `traverse` thực thi.

Trong phần kế tiếp chúng ta sẽ hiện thực các phương thức này.

4.3. Hiện thực danh sách

Chúng ta đã đặc tả đầy đủ các tác vụ mong muốn đối với danh sách. Phần này sẽ hiện thực chi tiết chúng trong C++. Ngăn xếp và hàng đã được hiện thực cả hai dạng liên tục và liên kết. Chúng ta cũng sẽ làm tương tự cho danh sách.

4.3.1. Hiện thực danh sách liên tục

Trong hiện thực danh sách liên tục (*contiguous list*), các phần tử của danh sách có kiểu là `Entry` được chứa trong dãy kích thước là `max_list`. Cũng giống như hiện thực ngăn xếp liên tục, ở đây chúng ta cần một biến `count` đếm số phần tử hiện có trong danh sách. Sau đây là định nghĩa lớp `List` có hai thuộc tính thành phần và tất cả các phương thức mà chúng ta đã đặc tả.

```
template <class Entry>
class List {
public:
//  Các phương thức của kiểu dữ liệu trừu tượng danh sách
   List();
   int size() const;
   bool full() const;
   bool empty() const;
   void clear();
```

```
   void traverse(void (*visit)(Entry &));
   ErrorCode retrieve(int position, Entry &x) const;
   ErrorCode replace(int position, const Entry &x);
   ErrorCode remove(int position, Entry &x);
   ErrorCode insert(int position, const Entry &x);

protected:
// Các thuộc tính cho hiện thực danh sách liên tục
   int count;
   Entry entry[max_list];
};
```

Hầu hết các phương thức (`List, clear, empty, full, size, retrieve`) rất dễ hiện thực.

```
template <class Entry>
int List<Entry>::size() const
/*
post: trả về số phần tử của danh sách. Danh sách không đổi.
*/
{
   return count;
}
```

Chúng ta dành các phương thức đơn giản khác lại cho phần bài tập. Ở đây chúng ta sẽ tập trung vào các phương thức truy xuất dữ liệu. Khi thêm một phần tử mới, các phần tử trong dãy phải được di chuyển để nhường chỗ.

```
template <class Entry>
ErrorCode List<Entry>::insert(int position, const Entry &x)
/*
post: Nếu danh sách chưa đầy và 0 ≤ position ≤ n, n là số phần tử hiện có của danh sách,
      phương thức trả về success: mọi phần tử từ position đến cuối danh sách sẽ có vị trí
      tăng lên 1, x được thêm vào tại position; ngược lại, danh sách không đổi, ErrorCode sẽ
      cho biết lỗi cụ thể.
*/
{
   if (full())
      return overflow;
   if (position < 0 || position > count)
      return range_error;
   for (int i = count - 1; i >= position; i--)
      entry[i + 1] = entry[i];
   entry[position] = x;
   count++;
   return success;
}
```

Có bao nhiêu công việc mà hàm trên cần phải làm? Nếu phần tử mới được thêm vào cuối danh sách thì hàm chỉ phải thực hiện một số không đổi các lệnh. Trong trường hợp ngược lại, nếu phần tử được thêm vào đầu danh sách, hàm sẽ phải dịch chuyển một số phần tử lớn nhất để tạo chỗ trống, nếu danh sách đã

khá dài thì công việc cần làm rất nhiều. Xét bình quân, nếu chúng ta giả sử mọi vị trí trong danh sách đều có khả năng thêm phần tử mới như nhau, hàm trên sẽ phải dịch chuyển một nửa số phần tử trong danh sách. Chúng ta nói rằng số việc cần làm trong hàm tỉ lệ với chiều dài n của danh sách.

Tương tự, việc loại phần tử trong danh sách cũng cần phải dịch chuyển các phần tử để lấp chỗ trống và việc loại này cũng tốn thời gian tỉ lệ với chiều dài n của danh sách.

Khác với hai trường hợp trên, hầu hết các phương thức còn lại không cần thực hiện vòng lặp nào và thời gian thực hiện là hằng số. Tóm lại,

Trong xử lý danh sách liên tục có n phần tử:

- **`insert`** và **`remove`** cần thời gian tỉ lệ với n.
- **`List`**, **`clear`**, **`empty`**, **`full`**, **`size`**, **`replace`** và **`retrieve`** thực hiện trong thời gian không đổi.

Chúng ta chưa kể ra đây phương thức **`traverse`** vì thời gian thực hiện còn phụ thuộc vào thông số hàm visit. Riêng `traverse` thì ít nhất cũng cần thời gian tỉ lệ với n do phải có vòng lặp để duyệt qua hết các phần tử của danh sách. Tuy nhiên, với cùng một hàm `visit` thì `traverse` cần thời gian tỉ lệ với n.

```
template <class Entry>
void List<Entry>::traverse(void (*visit)(Entry &))
/*
post: Công việc đặc tả bởi hàm *visit được thực hiện lần lượt trên từng phần tử của danh sách,
      bắt đầu từ phần tử thứ 0.
*/
{
   for (int i = 0; i < count; i++)
      (*visit)(entry[i]);
}
```

4.3.2. Hiện thực danh sách liên kết đơn giản

4.3.2.1. Các khai báo

Để hiện thực danh sách liên kết (*linked list*), chúng ta bắt đầu với khai báo `Node`. `Node` dưới đây cũng tương tự như trong ngăn xếp liên kết và hàng liên kết.

```
template <class Entry>
struct Node {
//  Các thuộc tính
   Entry entry;
   Node<Entry> *next;
//  constructors
   Node();
   Node(Entry item, Node<Entry> *link = NULL);
};
template <class Entry>
```

```
class List {
public:
//  Các phương thức của danh sách liên kết (cũng giống như của danh sách liên tục)
//  Các phương thức bảo đảm tính an toàn cho CTDL có chứa thuộc tính con trỏ.
   ~List();
   List(const List<Entry> &copy);
   void operator =(const List<Entry> &copy);
protected:
//  Các thuộc tính cho hiện thực liên kết của danh sách
   int count;
   Node<Entry> *head; // Con trỏ chỉ phần tử đầu của danh sách.

//  The following auxiliary function is used to locate list positions
   Node<Entry> *set_position(int position) const;
};
```

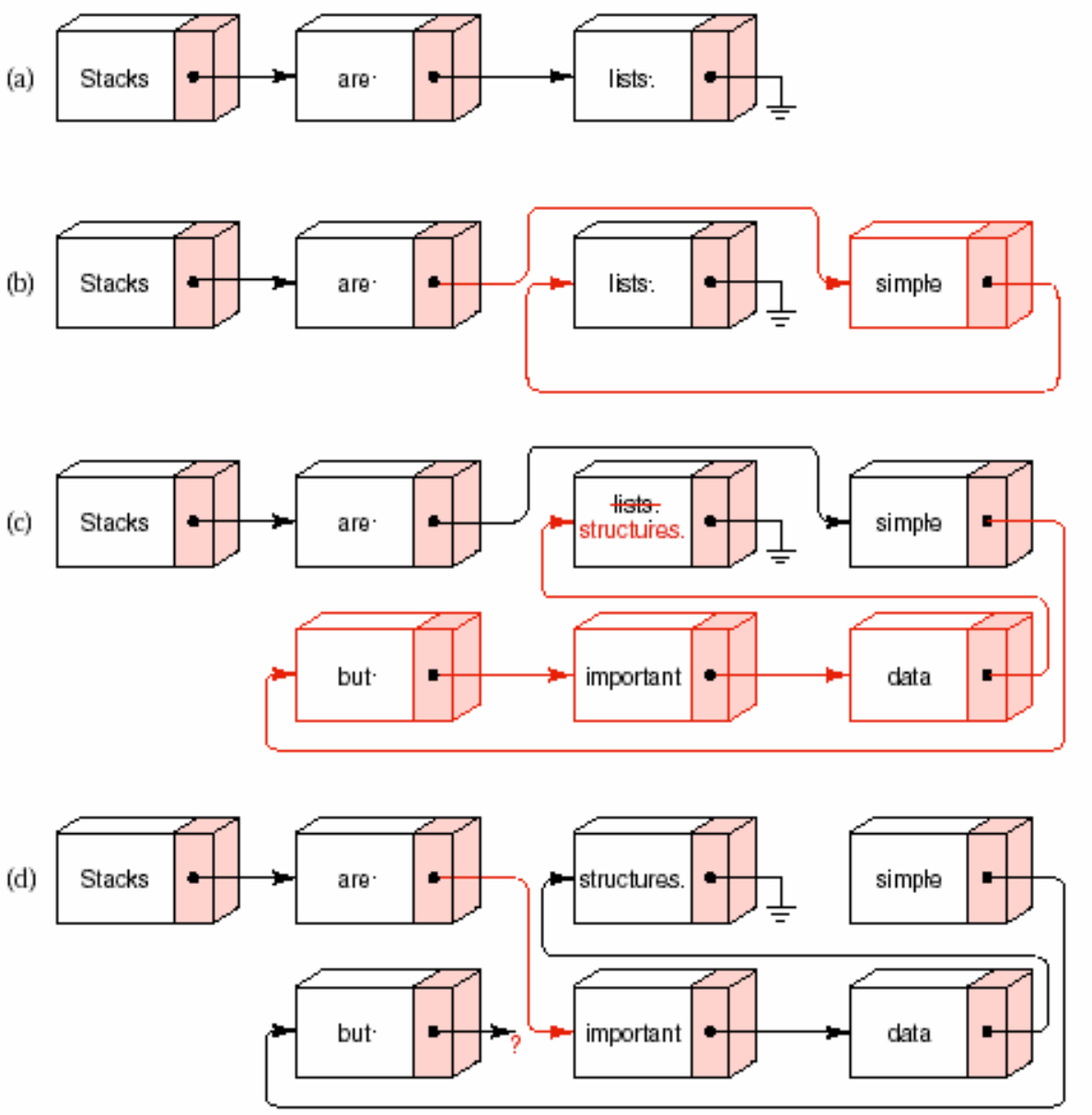
Trong định nghĩa trên chúng ta không liệt kê lại các phương thức của danh sách liên kết vì chúng cũng tương tự như đối với danh sách liên tục. Trong phần `protected` chúng ta có bổ sung phương thức `set_position` mà chúng ta sẽ thấy ích lợi của nó trong khi hiện thực các phương thức `public` khác.

4.3.2.2. Ví dụ

Hình 4.1 minh họa việc thêm bớt dữ liệu trong danh sách qua một ví dụ sửa đổi văn bản. Mỗi phần tử trong danh sách chứa một từ và một tham chiếu đến phần tử kế. Hình a là danh sách chứa câu ban đầu là "*Stacks are lists*" . Nếu chúng ta thêm từ "*simple*" trước từ "*lists*" chúng ta có danh sách như hình b. Tiếp theo chúng ta quyết định thay thế từ "*lists*" bởi từ "*structures*" và thêm ba từ "*but important data*" thì có hình c. Cuối cùng chúng ta lại quyết định bỏ đi các từ "*simple but*" để có được câu cuối cùng "*Stacks are important data structures*".

4.3.2.3. Tìm đến một vị trí trong danh sách

Chúng ta thiết kế một hàm **`set_position`** để được gọi trong một vài phương thức. Hàm này nhận thông số là **`position`** (một số nguyên chỉ vị trí phần tử trong danh sách) và trả về con trỏ tham chiếu đến phần tử tương ứng trong danh sách.



Hình 4.1- Các thao tác trên danh sách liên kết.

Nếu người sử dụng nhìn thấy được `set_position` thì họ sẽ có thể truy xuất đến mọi phần tử trong danh sách. Vì vậy, để duy trì tính đóng kín của dữ liệu, chúng ta sẽ không cho phép người sử dụng nhìn thấy hàm `set_position`. Bằng cách khai báo `protected` chúng ta bảo đảm rằng hàm này chỉ được gọi trong các phương thức khác của danh sách.

Cách dễ nhất để xây dựng hàm `set_position` là bắt đầu duyệt từ đầu của danh sách cho đến phần tử mà chúng ta muốn tìm.

```
template <class Entry>
Node<Entry> *List<Entry>::set_position(int position) const
/*
```

```
Pre:  position phải hợp lệ; 0 <= position < count.
Post: trả về địa chỉ của phần tử tại position.
*/
{
   Node<Entry> *q = head;
   for (int i = 0; i < position; i++) q = q->next;
   return q;
}
```

Do chúng ta nắm được chính xác các phương thức nào cần gọi đến `set_position`, trong hàm này chúng ta không cần kiểm tra lỗi. Thay vào đó chúng ta bảo đảm bằng `precondition` cho nó. Có nghĩa là các phương thức trước khi gọi `set_position` sẽ kiểm tra trước và chỉ gọi khi điều kiện hợp lệ. Việc kiểm tra sẽ không phải lặp lại trong hàm này, chương trình sẽ hiệu quả hơn.

Nếu mọi phần tử được truy xuất với xác suất ngang nhau thì trung bình hàm `set_position` sẽ phải duyệt qua một nửa số phần tử trong danh sách để đến được vị trí cần thiết. Thời gian này tỉ lệ với chiều dài n của danh sách.

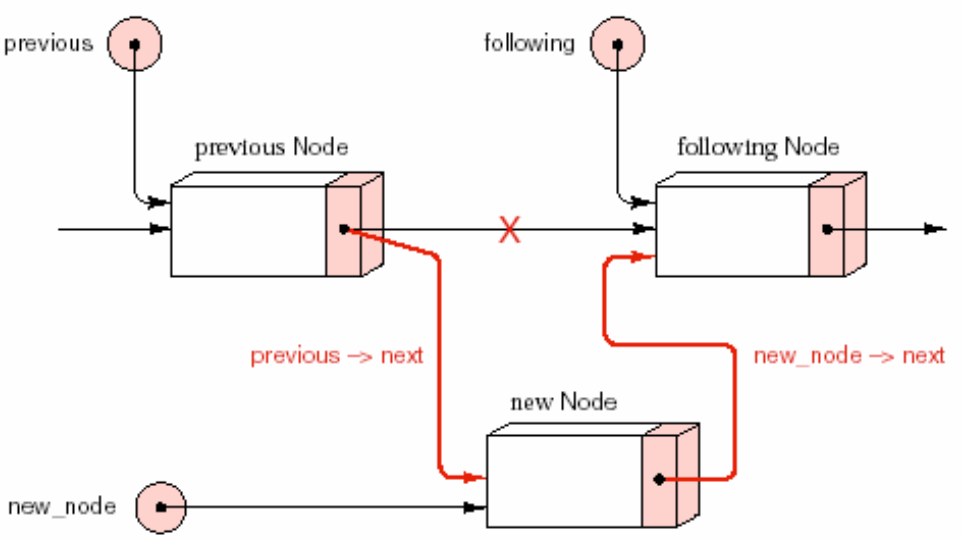
4.3.2.4. Thêm phần tử vào danh sách

Tiếp theo chúng ta sẽ xem xét vấn đề thêm một phần tử mới vào danh sách. Nếu chúng ta có một phần tử mới và chúng ta muốn chèn phần tử này vào một vị trí nào đó trong danh sách, ngoại trừ vị trí đầu danh sách, như hình 4.2, chúng ta cần có hai con trỏ **`previous`** và **`following`** chỉ đến hai phần tử trước và sau vị trí cần chèn. Nếu con trỏ `new_node` đang chỉ phần tử mới cần chèn thì các lệnh gán sau sẽ chèn được phần tử mới vào danh sách:

```
new_node->next = following;
previous->next = new_node;
```

Trong phương thức `insert` dưới đây phép gán `new_node->next= following` được thực hiện thông qua *constructor* có nhận thông số thứ hai là `following`.

Việc thêm phần tử vào đầu danh sách cần được xử lý riêng, do trường hợp này không có phần tử nào nằm trước phần tử mới nên chúng ta không sử dụng con trỏ `previous`, thay vào đó thuộc tính `head` chỉ đến phần tử đầu của danh sách phải được gán lại.



Hình 4.2- Thêm phần tử vào danh sách liên kết.

```
template <class Entry>
ErrorCode List<Entry>::insert(int position, const Entry &x)
/*
post: Nếu danh sách chưa đầy và 0 ≤ position ≤ n, n là số phần tử hiện có của danh sách,
      phương thức trả về success: mọi phần tử từ position đến cuối danh sách sẽ có vị trí tăng
      lên 1, x được thêm vào tại position; ngược lại, danh sách không đổi, ErrorCode sẽ cho
      biết lỗi cụ thể.
*/
{
   if (position < 0 || position > count)
      return range_error;
   Node<Entry> *new_node, *previous, *following;
   if (position == 0) // Trường hợp đặc biệt: phần tử mới thêm vào đầu danh sách.
      following = head;
   else {             // Trường hợp tổng quát.
      previous = set_position(position - 1); // Tìm phần tử phía trước vị trí cần thêm
                                             phần tử mới.
      following = previous->next;
   }

   new_node = new Node<Entry>(x, following);
   if (new_node == NULL)
      return overflow;
   if (position == 0) // Trường hợp đặc biệt: phần tử mới thêm vào đầu danh sách.
      head = new_node;
   else               // Trường hợp tổng quát.
      previous->next = new_node;
   count++;
   return success;
}
```

Ngoài lệnh gọi hàm `set_position`, các lệnh còn lại trong `insert` không phụ thuộc vào chiều dài n của danh sách. Do đó `insert`, cũng giống như `set_position`, sẽ có thời gian thực hiện tỉ lệ với chiều dài n của danh sách.

4.3.2.5. Các tác vụ khác

Các phương thức còn lại của danh sách liên kết xem như bài tập. Việc tìm kiếm một phần tử nào đó trong các phương thức luôn phải gọi hàm `set_position`. Hầu hết các phương thức này cũng giống như `insert`, sử dụng các lệnh chiếm thời gian không đổi, ngoại trừ lúc gọi hàm `set_position`. Chỉ có phương thức `clear` và `traverse` là phải duyệt qua các phần tử của danh sách. Chúng ta có kết luận như sau:

Trong việc xử lý một danh sách liên kết có n phần tử:

- **`insert`**, **`remove`**, **`retrieve`** và **`replace`** cần thời gian tỉ lệ với n.
- **`List`**, **`empty`**, **`full`** và **`size`** thực hiện với thời gian không đổi.

Một lần nữa, chúng ta chưa kể đến phương thức **`traverse`** ở đây, vì thời gian nó cần còn phụ thuộc vào thông số `visit`. Tuy nhiên, cũng như phần trước, với cùng một hàm visit thì `traverse` cần thời gian tỉ lệ với n.

4.3.3. Lưu lại vị trí hiện tại

Đa số các ứng dụng truy xuất các phần tử của danh sách theo thứ tự các phần tử. Nhiều ứng dụng khác truy xuất cùng một phần tử nhiều lần, thực hiện các tác vụ truy xuất hoặc thay thế trước khi chuyển qua phần tử khác. Đối với tất cả các ứng dụng này, cách hiện thực danh sách hiện tại của chúng ta tỏ ra không hiệu quả, do mỗi lần truy xuất một phần tử, hàm set_position đều phải tìm từ đầu danh sách đến phần tử mong muốn. Nếu chúng ta có thể nhớ lại phần tử vừa được truy xuất trong danh sách, và tác vụ mà ứng dụng yêu cầu tiếp theo cũng xem xét phần tử này hoặc phần tử kế thì việc tìm kiếm bắt đầu từ vị trí được nhớ này nhanh hơn rất nhiều.

Tuy nhiên, không phải việc nhớ lại vị trí vừa được truy xuất này luôn có hiệu lực đối với mọi ứng dụng. Chẳng hạn với ứng dụng truy xuất các phần tử trong danh sách theo thứ tự ngược, mọi truy xuất đều phải bắt đầu từ đầu danh sách do các tham chiếu trong các phần tử chỉ có một chiều.

Chúng ta dùng thuộc tính **`current_position`** để lưu vị trí vừa nói trên. Thuộc tính này sẽ được `set_position` sử dụng cũng như sẽ cập nhật lại mỗi khi hàm này được gọi. Điều cần lưu ý là `set_position` được gọi trong các phương thức khác của danh sách, trong đó có một số phương thức được đặc tả là `const` có nghĩa là không được làm thay đổi danh sách, trong khi đó `current_position` phải được thay đổi. Như vậy, chúng ta sẽ dùng từ

khóa **mutable** của C++ nhưng lưu ý rằng không phải từ khóa này luôn được cung cấp bởi mọi trình biên dịch C++. Khi một thuộc tính của một lớp được khai báo là `mutable` thì nó có thể được thay đổi ngay cả trong các hàm được khai báo là `const`.

Định nghĩa danh sách mới như sau:

```
template <class Entry>
class List {
public:
//   Các phương thức của danh sách liên kết (cũng giống như của danh sách liên tục)
//   Các phương thức bảo đảm tính an toàn cho CTDL có chứa thuộc tính con trỏ.

protected:
//   Các thuộc tính cho hiện thực liên kết của danh sách có lưu vị trí hiện tại.
   int count;
   mutable int current_position;
   Node<Entry> *head;
   mutable Node<Entry> *current;

//   Hàm phụ trợ để tìm một phần tử.
   void set_position(int position) const;
};
```

Hai thuộc tính được thêm vào **current_position** và **current** đều được khai báo `protected`, do đó đối với người sử dụng lớp `List` vẫn không có gì thay đổi so với định nghĩa cũ.

Hàm `set_position` được viết lại như sau:

```
template <class Entry>
void List<Entry>::set_position(int position) const
/*
pre:  position hợp lệ: 0 <= position < count.
post: Thuộc tính current chứa địa chỉ phần tử được tìm thấy tại position,
      current_position được cập nhật tương ứng.
*/
{
   if (position < current_position) { // Trường hợp phải tìm từ đầu danh sách
      current_position = 0;
      current = head;
   }
   for ( ; current_position != position; current_position++)
      current = current->next;
}
```

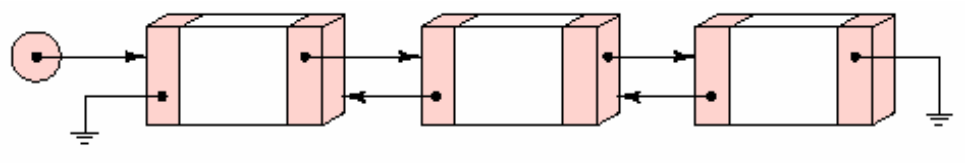
Nếu một phần tử trong danh sách được truy xuất lập lại nhiều lần thì các lệnh trong `if` cũng như trong vòng `for` của hàm trên đều không phải thực hiện, hàm sẽ không hề chiếm thời gian chạy. Nếu phần tử kế được truy xuất, các lệnh trong vòng `for` chỉ chạy một lần, hàm vẫn thực hiện rất nhanh. Trong trường hợp xấu

nhất, nếu cần phải bắt đầu từ đầu danh sách, hàm cũng sẽ làm việc giống như cách chúng ta đã hiện thực trước đây.

4.3.4. Danh sách liên kết kép

Một vài ứng dụng thường xuyên yêu cầu dịch chuyển tới và lui trên danh sách. Trong phần trước chúng ta đã giải quyết việc dịch chuyển theo một chiều trong quá trình duyệt danh sách. Nhưng việc lập trình hơi khó khăn và thời gian chạy chương trình phụ thuộc vào danh sách, nhất là khi danh sách có nhiều phần tử.

Để khắc phục vấn đề này, có nhiều chiến lược khác nhằm giải quyết việc tìm phần tử nằm trước phần tử hiện tại trong danh sách. Trong phần này chúng ta tìm hiểu một chiến lược đơn giản nhất nhưng cũng linh động và phù hợp trong rất nhiều trường hợp.



Hình 4.3- Danh sách liên kết kép.

4.3.4.1. Các khai báo cho danh sách liên kết kép

Như hình 4.3 minh họa, ý tưởng ở đây là việc lưu cả hai con trỏ chỉ hai hướng ngược nhau trong cùng một node của danh sách. Bằng cách dịch chuyển theo tham chiếu thích hợp chúng ta có thể duyệt danh sách theo hướng mong muốn. CTDL này được gọi là danh sách liên kết kép (*doubly linked list*).

```
template <class Node_entry>
struct Node {
//  Các thuộc tính
   Node_entry entry;
   Node<Node_entry> *next;
   Node<Node_entry> *back;
//  constructors
   Node();
   Node(Node_entry, Node<Node_entry> *link_back = NULL,
                    Node<Node_entry> *link_next = NULL);
};
```

Constructor cho Node của danh sách liên kết kép gần giống *constructor* cho Node của danh sách liên kết đơn. Dưới đây là đặc tả cho lớp danh sách liên kết kép:

```
template <class Entry>
class List {
public:
// Các phương thức thông thường của danh sách.
// Các phương thức bảo đảm tính an toàn cho CTDL có thuộc tính con trỏ.
protected:
// Các thuộc tính
   int count;
   mutable int current_position;
   mutable Node<Entry> *current;

// Hàm phụ trợ để tìm đến một phần tử trong danh sách
   void set_position(int position) const;
};
```

Trong cách hiện thực này chúng ta chỉ cần giữ một con trỏ tham chiếu đến một phần tử nào đó trong danh sách là chúng ta có thể duyệt danh sách theo hướng này hoặc hướng kia. Như vậy, chúng ta dùng luôn con trỏ **`current`** chỉ đến phần tử hiện tại để làm nhiệm vụ này, và chúng ta không cần giữ con trỏ chỉ đến đầu hoặc cuối danh sách.

4.3.4.2. Các tác vụ trên danh sách liên kết kép

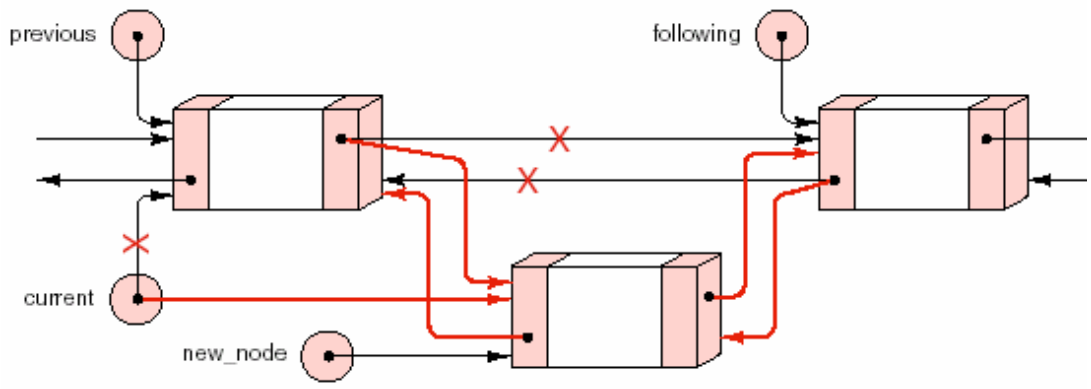
Trong danh sách liên kết kép, việc duyệt danh sách theo cả hai hướng để tìm một phần tử, việc thêm hoặc loại phần tử tại vị trí nào đó có thể được thực hiện dễ dàng. Một vài tác vụ có thay đổi chút ít so với danh sách liên kết đơn, như `insert` và `delete`, do phải cập nhật đầy đủ cả hai con trỏ theo hai hướng của danh sách .

Trước hết, để tìm một vị trí nào đó, chúng ta chỉ cần quyết định nên duyệt theo hướng nào trong danh sách bắt đầu từ con trỏ `current`.

```
template <class Entry>
void List<Entry>::set_position(int position) const
/*
pre:  position hợp lệ: 0 <= position < count.
post: thuộc tính current chứa địa chỉ của phần tử tại position.
*/
{
   if (current_position <= position)
      for ( ; current_position != position; current_position++)
         current = current->next;
   else
      for ( ; current_position != position; current_position--)
         current = current->back;
}
```

Với hàm này chúng ta viết tác vụ `insert` sau đây. Hình 4.4 minh họa các con trỏ cần phải cập nhật. Chúng ta cũng đặc biệt chú ý hai trường hợp hơi đặc biệt,

đó là khi thêm phần tử vào một trong hai đầu của danh sách hoặc thêm vào một danh sách rỗng.



Hình 4.4- Thêm phần tử và danh sách liên kết kép.

```
template <class Entry>
ErrorCode List<Entry>::insert(int position, const Entry &x)
/*
post: Nếu danh sách chưa đầy và 0 ≤ position ≤ n, n là số phần tử hiện có của danh sách,
      phương thức trả về success: mọi phần tử từ position đến cuối danh sách sẽ có vị trí
      tăng lên 1, x được thêm vào tại position; ngược lại, danh sách không đổi, ErrorCode sẽ
      cho biết lỗi cụ thể.
*/
{
   Node<Entry> *new_node, *following, *preceding;
   if (position < 0 || position > count) return range_error;
   if (position == 0) {
      if (count == 0) following = NULL;    // Trường hợp đặc biệt: danh sách đang
                                              rỗng.
      else {
         set_position(0);
         following = current;
      }
      preceding = NULL;                    // Trường hợp đặc biệt: thêm phần tử mới
                                              vào đầu danh sách, không có phần tử
                                              đứng trước.
   }
   else {
      set_position(position - 1);
      preceding = current;
      following = preceding->next;         // Trường hợp tổng quát: thêm phần tử
                                              vào giữa, nhưng gộp cả trường hợp
                                              thêm vào cuối (position = n)
                                              following sẽ nhận trị NULL.
   }
   new_node = new Node<Entry>(x, preceding, following);

   if (new_node == NULL) return overflow;
   if (preceding != NULL) preceding->next = new_node;
   if (following != NULL) following->back = new_node;
   current = new_node;
```

```
   current_position = position;
   count++;
   return success;
}
```

Giá phải trả đối với danh sách liên kết kép là vùng nhớ cho tham chiếu thứ hai trong mỗi `Node`. Với phần lớn ứng dụng, do vùng **`entry`** cần chứa thông tin trong `Node` lớn hơn nhiều vùng nhớ dành cho các con trỏ, nên tổng dung lượng bộ nhớ tăng không đáng kể.

4.4. So sánh các cách hiện thực của danh sách

Chúng ta đã xem xét một vài giải thuật xử lý cho danh sách liên kết và một vài biến thể về cấu trúc và cách hiện thực. Trong phần này chúng ta sẽ phân tích các ưu nhược điểm của danh sách liên kết và danh sách liên tục.

Ưu điểm lớn nhất của danh sách liên kết trong bộ nhớ động là tính mềm dẻo. Không có vấn đề tràn bộ nhớ trừ khi bộ nhớ máy tính thực sự đã được sử dụng hết. Đặc biệt khi một `entry` chứa thông tin quá lớn, chúng ta khó có thể xác định tổng dung lượng vùng nhớ như thế nào cho vừa đủ để khai báo một dãy, trong khi chúng ta cũng cần phải xét đến phần bộ nhớ còn lại sao cho đủ để dành cho các biến khác. Trong bộ nhớ động, chúng ta không cần phải quyết định điều này trước khi chương trình chạy.

Trong danh sách liên kết, việc thêm hay loại phần tử có thể thực hiện nhanh hơn so với trong danh sách liên tục. Đối với các CTDL lớn, việc thay đổi một vài con trỏ nhanh hơn rất nhiều so với việc chép dữ liệu sang chỗ khác.

Nhược điểm đầu tiên của danh sách liên kết là tốn vùng nhớ cho các con trỏ. Trong phần lớn hệ thống, một con trỏ chiếm vùng nhớ bằng vùng nhớ của một số nguyên. Như vậy một danh sách liên kết các số nguyên sẽ đòi hỏi vùng nhớ gấp đôi một danh sách liên tục các số nguyên.

Trong nhiều ứng dụng thực tiễn, một `node` trong danh sách thường lớn, dữ liệu thường chứa hàng trăm *bytes*, việc sử dụng danh sách liên kết chỉ tốn thêm khoản một phần trăm vùng nhớ. Thực ra, danh sách liên kết tiết kiệm vùng nhớ hơn nhiều, nếu xét đến những vùng nhớ được khai báo dự trữ sẵn cho việc thêm phần tử trong danh sách liên tục mà chưa được dùng đến. Nếu mỗi `entry` chiếm 100 *bytes* thì vùng nhớ liên tục chỉ tiết kiệm khi số phần tử sử dụng thực sự trong dãy lên đến 99 *bytes*.

Nhược điểm chính của danh sách liên kết là nó không thích hợp với việc truy xuất ngẫu nhiên. Trong vùng nhớ liên tục, việc truy xuất đến bất kỳ vị trí nào cũng rất nhanh và không khác gì so với những vị trí khác. Trong danh sách liên kết, có thể phải duyệt cả chặng đường dài mới đến được phần tử mong muốn. Việc

truy xuất một node trong vùng nhớ liên kết cũng chiếm hơn một chút thời gian vì trước hết phải có được con trỏ và sau đó mới đến được địa chỉ cần tìm, tuy nhiên điều này thường không quan trọng. Ngoài ra, các tác vụ xử lý trong danh sách liên kết thường phải lập trình công phu hơn.

Danh sách liên tục, nói chung, thường được chọn khi:

- Mỗi entry rất nhỏ.
- Kích thước của danh sách được biết trước khi lập trình.
- Ít có nhu cầu thêm hoặc loại phần tử trừ trường hợp phần tử cuối danh sách.
- Việc truy xuất ngẫu nhiên thường xảy ra.

Danh sách liên kết tỏ ra ưu thế khi:

- Mỗi entry lớn.
- Kích thước của danh sách không được biết trước khi ứng dụng chạy.
- Có yêu cầu về tính linh hoạt: thêm, loại phần tử hoặc tổ chức lại các phần tử.

Để chọn lựa CTDL với cách hiện thực thích hợp, người lập trình cần xem xét các tác vụ nào sẽ được thực hiện trên cấu trúc đó, tác vụ nào trong số đó là quan trọng nhất. Việc truy xuất là cục bộ nếu một phần tử được truy xuất, nó có thể được truy xuất lần nữa. Và nếu các phần tử thường được truy xuất theo thứ tự, thì nên nhớ lại vị trí phần tử vừa được truy xuất như là một thuộc tính của danh sách. Còn nếu việc truy xuất theo hai hướng của danh sách là cần thiết thì nên chọn cách hiện thực danh sách liên kết kép.

4.5. Danh sách liên kết trong mảng liên tục

Một vài ngôn ngữ tuy xưa nhưng rất phổ biến như Fortran, Cobol và Basic không cung cấp khả năng sử dụng bộ nhớ động hoặc con trỏ. Nếu cần sử dụng các ngôn ngữ này để giải quyết các bài toán mà trong đó các tác vụ trên danh sách liên kết (DSLK) tỏ ra có ưu thế hơn hẳn trên danh sách liên tục (việc thay đổi một vài con trỏ dễ dàng và nhanh chóng hơn nhiều việc phải chép lại một số lượng lớn dữ liệu), chúng ta vẫn có thể sử dụng mảng liên tục để mô phỏng DSLK. Trong phần này chúng ta sẽ tìm hiểu một hiện thực của DSLK mà không cần con trỏ. Hay nói cách khác, chúng ta không dùng con trỏ chứa địa chỉ, mà sẽ dùng con trỏ là một số nguyên, và DSLK sẽ được hiện thực trong một mảng liên tục.

4.5.1. Phương pháp

Ý tưởng chính ở đây bắt đầu từ một mảng liên tục dùng để chứa các phần tử của một DSLK. Chúng ta xem mảng này như một vùng nhớ chưa sử dụng và chúng ta sẽ tự phân phối lấy. Chúng ta sẽ xây dựng một số hàm để quản lý mảng

này: nhận biết vùng nào trong mảng chưa được sử dụng, nối kết các phần tử trong mảng theo một thứ tự mong muốn.

Một đặc điểm của DSLK mà chúng ta phải bỏ qua trong phần này là việc định vị bộ nhớ động, ngay từ đầu chúng ta phải xác định kích thước cần thiết cho mảng. Mọi ưu điểm còn lại khác của DSLK đều được giữ nguyên, như tính mềm dẻo trong việc tổ chức lại vùng nhớ cho các phần tử có kích thước lớn, hoặc tính dễ dàng và hiệu quả trong việc thêm hay bớt bất cứ phần tử nào trong danh sách.

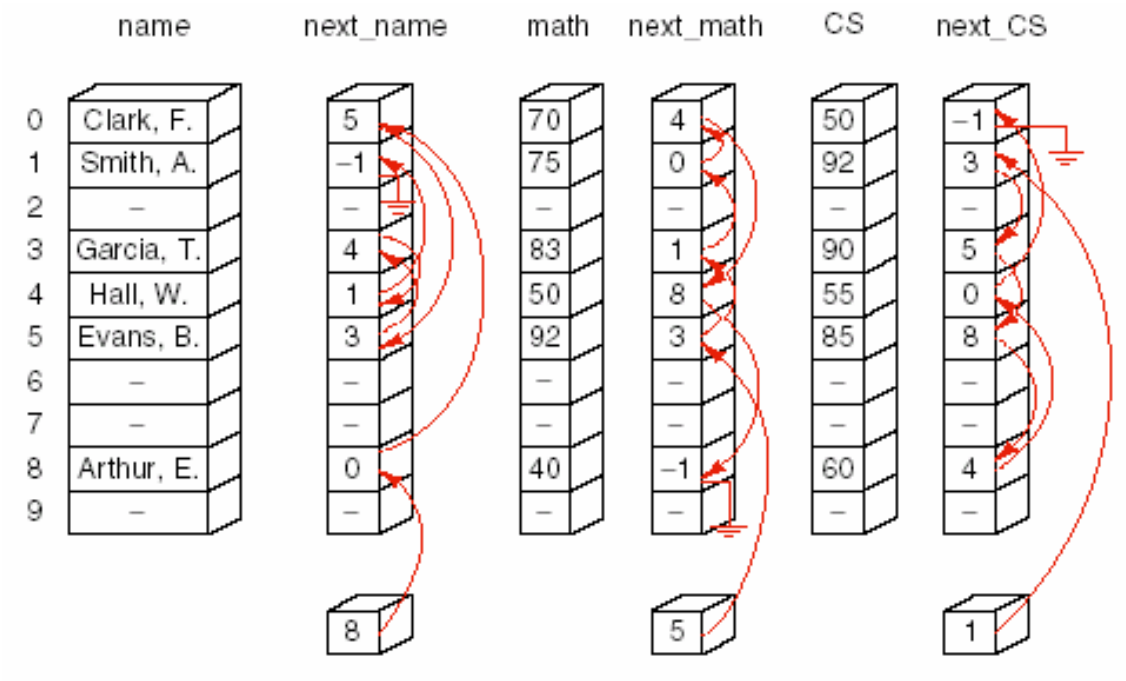
Hiện thực DSLK trong mảng liên tục cũng tỏ ra rất hiệu quả trong những ngôn ngữ tựa C++ có cung cấp con trỏ và cách định vị bộ nhớ động. Các ứng dụng dưới đây được xem là thích hợp khi sử dụng DSLK trong mảng liên tục :

- Số phần tử tối đa trong danh sách được biết trước.
- Các tham chiếu thường xuyên được tổ chức lại, nhưng việc thêm hoặc loại phần tử tương đối ít xảy ra.
- Cùng dữ liệu nhưng có khi cần xử lý như DSLK có khi lại cần xử lý như danh sách liên tục.

Hình 4.5 là một ví dụ minh họa cho những ứng dụng như vậy. Đây là một phần dữ liệu chứa các thông tin về sinh viên. Mã sinh viên được gán cho các sinh viên theo thứ tự nhập trường, tên và điểm số không theo một thứ tự đặc biệt nào. Thông tin về sinh viên có thể được tìm thấy nhanh chóng dựa vào mã sinh viên do số này được dùng như chỉ số để tìm trong mảng liên tục. Tuy nhiên, thỉnh thoảng chúng ta cần in danh sách sinh viên có thứ tự theo tên, và điều này có thể làm được bằng cách lần theo **các tham chiếu được lưu trong mảng `next_name`**. Tương tự, các điểm số cũng có thể sắp thứ tự nhờ các tham chiếu trong các mảng tương ứng.

Để thấy được cách hiện thực này của DSLK làm việc như thế nào, chúng ta hãy duyệt DSLK `next_name` trong phần đầu của hình 4.5. Đầu vào của danh sách chứa trị là 8, có nghĩa là phần tử trong vị trí 8, Arthur, E., là phần tử đầu tiên của danh sách. Vị trí 8 của `next_name` chứa trị 0, có nghĩa là tên ở vị trí 0, Clark, F., là phần tử thứ hai. Vị trí 0 của `next_name` chứ trị 5, vậy Evans, B., là phần tử kế tiếp. Vị trí 5 chỉ đến vị trí 3 (Garcia, T.), vị trí 3 lại chỉ ví trí 4 (Hall, W.), và vị trí 4 chỉ vị trí 1 (Smith, A.). Tại vị trí 1, `next_name` chứa trị -1, có nghĩa là vị trí 1 là phần tử cuối cùng của danh sách.

Tương tự, mảng `next_math` biểu diễn một DSLK, cho phép truy xuất mảng `math` theo thứ tự giảm dần. Phần tử đầu tiên tại vị trí 5, kế đến là 3, 1, 0, 4, 8. Thứ tự các phần tử xuất hiện trong DSLK biểu diễn bởi `next_CS` là 1, 3, 5, 8, 4, 0.



Hình 4.5- DSLK trong mảng liên tục.

Như ví dụ trong hình 4.5, hiện thực của DSLK trong mảng liên tục có được tính linh hoạt của DSLK đối với những sự thay đổi. Ngoài ra nó còn có khả năng chia sẻ thông tin (chẳng hạn tên sinh viên) giữa các DSLK khác nhau. Hiện thực này cũng còn có ưu điểm của danh sách liên tục là có thể truy xuất ngẫu nhiên các phần tử nhờ cách sử dụng chỉ số truy xuất trực tiếp.

Trong hiện thực của DSLK trong mảng liên tục, các con trỏ trở thành các chỉ số tương đối so với điểm bắt đầu của danh sách. Các tham chiếu của danh sách chứa trong một mảng, mỗi phần tử của mảng chứa một số nguyên chỉ đến vị trí của phần tử kế của danh sách trong mảng chứa dữ liệu. Để phân biệt với các con trỏ (*pointer*) của DSLK trong bộ nhớ động, chúng ta sẽ dùng từ **chỉ số** (*index*) để gọi các tham chiếu này.

Chúng ta cần khai báo hai mảng liên tục cho mỗi DSLK, **`entry[ ]`** để chứa dữ liệu, và **`next_node[ ]`** để chứa chỉ số chỉ đến phần tử kế. Đối với phần lớn các ứng dụng, **`entry`** là một mảng mà mỗi phần tử là một cấu trúc, hoặc một vài mảng trong trường hợp ngôn ngữ lập trình không cung cấp kiểu cấu trúc. Cả hai mảng **`entry`** và **`next_node`** cần đánh chỉ số từ 0 đến **`max_list-1`**, **`max_list`** là một hằng số biết trước.

Do chúng ta dùng chỉ số bắt đầu từ 0, chúng ta sẽ dùng trị đặc biệt –1 để biểu diễn danh sách đã kết thúc, tương tự như trị `NULL` của con trỏ trong bộ nhớ động.

4.5.2. Các tác vụ quản lý vùng nhớ

Nhiệm vụ đầu tiên của chúng ta là nắm được các vùng nhớ còn trống để viết một số hàm tìm một vị trí mới hay trả một vị trí không sử dụng nữa về lại vùng nhớ trống. Khác với phần 4.3.2, toàn bộ vùng nhớ mà chúng ta sẽ dùng ở đây là một mảng liên tục gọi là **workspace**, các phần tử của nó tương ứng với các phần tử trong DSLK (hình 4.6). Chúng ta cũng sẽ gọi các phần tử trong workspace là node và sẽ khai báo **Node** để chứa dữ liệu. Mỗi **Node** là một cấu trúc gồm hai phần: **entry** kiểu **Entry** chứa dữ liệu, và **next** kiểu **index**. Kiểu index được hiện thực bằng số nguyên, có các trị biểu diễn vị trí các phần tử trong mảng liên tục, và như vậy nó thay thế kiểu con trỏ như trong các DSLK trước đây.

Các vị trí trống trong workspace có hai dạng:

- Các node chưa được sử dụng tới.
- Các node đã từng được sử dụng nhưng đã được giải phóng.

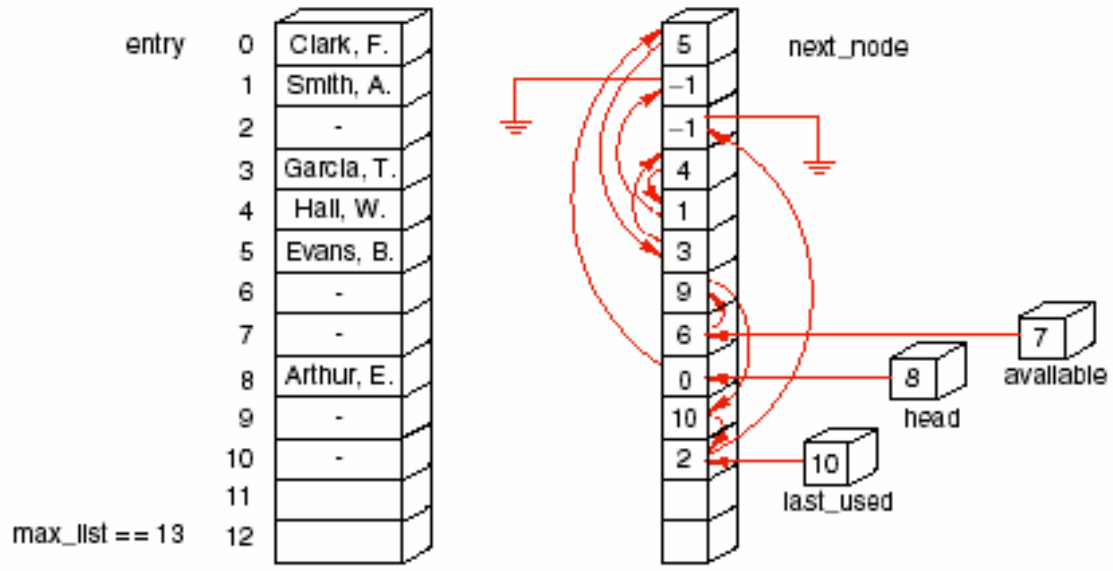
Chúng ta sẽ bắt đầu sử dụng từ đầu mảng liên tục và dùng chỉ số **last_used** chứa vị trí cuối vừa mới sử dụng trong mảng. Các vị trí trong mảng có chỉ số lớn hơn last_used là các vị trí chưa hề được sử dụng.

Các node đang chứa dữ liệu sẽ thuộc một DSLK có đầu vào là **head**. Head chứa vị trí của phần tử đầu của DSLK trong mảng. Các node kế tiếp trong DSLK này được truy xuất thông qua các chỉ số trong thành phần **next** của các **node**, biểu diễn bởi các mũi tên bên trái của **next_node** trong hình 4.6**.** Node cuối cùng của DSLK có chỉ số là –1. Chúng ta đọc được danh sách có các tên sắp theo thứ tự *alphabet* bắt đầu từ head = 8, Arthur, E. nằm đầu danh sách, rồi đến các tên tại các vị trí 0, 5, 3, 4, 1 của mảng; Smith, A. là tên nằm cuối danh sách.

Đối với những node đã từng sử dụng và đã được giải phóng, chúng ta sẽ dùng một dạng của cấu trúc liên kết để nối kết chúng lại với nhau và để có thể truy xuất đến, từ node này đến node kế. Do ngăn xếp liên kết là một dạng đơn giản nhất của cấu trúc liên kết, chúng ta sẽ dùng ngăn xếp liên kết cho trường hợp này. Ngăn xếp liên kết cũng được nối kết nhờ chỉ số **next** trong mỗi node, **available** là một chỉ số chứa **top** của ngăn xếp.

Các mũi tên bên phải của **next_node** trong hình 4.6, với đầu vào là **available**, chỉ các node trong một ngăn xếp bao gồm các node đã từng được sử dụng và đã được giải phóng. Chú ý rằng chỉ số trong các vùng **next** của ngăn xếp liên kết này chính xác là các số ≤ **last_used**, đó cũng là các vị trí trong mảng hiện tại không có dữ liệu. Bắt đầu từ **available** = 7, rồi đến 6, 9, 10, 2. Còn các vị trí từ **last_used**+1 trở đi là các vị trí chưa hề có dữ liệu.

Khi có một node bị loại khỏi DSLK chứa dữ liệu (chẳng hạn loại tên một sinh viên ra khỏi danh sách), vị trí của nó trong mảng được giải phóng và được push vào ngăn xếp liên kết. Ngược lại, khi cần thêm một node mới vào DSLK, chúng ta tìm vị trí trống bằng cách pop một phần tử ra khỏi ngăn xếp liên kết: trong hình 4.6 thì node mới sẽ được thêm vào vị trí 7 của mảng, còn **available** được cập nhật lại là 6. Chỉ khi cần thêm một node mới vào DSLK mà **available**=-1 (ngăn xếp liên kết rỗng), chúng ta mới dùng đến một node mới chưa hề sử dụng đến, đó là vị trí **last_used+1**, **last_used** được cập nhật lại. Khi **last_used** đạt **max_list-1**, mà **available=-1**, thì workspace đầy, danh sách của chúng ta không cho phép thêm node mới nữa.



Hình 4.6- DSLK trong mảng liên tục và ngăn xếp liên kết chứa các vùng có thể sử dụng.

Khi đối tượng **List** được khởi tạo, **available** và **last_used** phải được gán -1: **available=-1** chỉ ra rằng ngăn xếp chứa các vùng nhớ đã từng được sử dụng và đã được giải phóng là rỗng, **last_used=-1** chỉ rằng chưa có vùng nhớ nào trong mảng đã được sử dụng.

Chúng ta có khai báo DSLK trong mảng liên tục như sau:

```
typedef int index;
const int max_list = 7; //  giá trị nhỏ dành cho việc kiểm tra CTDL.
template <class Entry>
class Node {
public:
   Entry entry;
   index next;
};

template <class Entry>
class List {
```

```
public:
//  Methods of the list ADT
   List();
   int size() const;
   bool full() const;
   bool empty() const;
   void clear();
   void traverse(void (*visit)(Entry &));
   Error_code retrieve(int position, Entry &x) const;
   Error_code replace(int position, const Entry &x);
   Error_code remove(int position, Entry &x);
   Error_code insert(int position, const Entry &x);

protected:
//  Các thuộc tính
   Node<Entry> workspace[max_list];
   index available, last_used, head;
   int count;

//  Các hàm phụ trợ
   index new_node();
   void delete_node(index n);
   int current_position(index n) const;
   index set_position(int position) const;
};
```

Các phương thức `public` trên được đặc tả hoàn toàn giống với các hiện thực khác của `List` trước đây. Điều này có nghĩa là hiện thực mới của chúng ta có thể thay thế bất kỳ một hiện thực nào khác của CTDL trừu tượng `List` trong các ứng dụng. Một số hàm phụ trợ `protected` được bổ sung để quản lý các `node` trong `workspace`. Chúng được sử dụng để xây dựng các phương thức `public` nhưng không được nhìn thấy bởi người sử dụng. Các hàm **`new_node`** và **`delete_node`** thay thế cho các tác vụ **`new`** và **`delete`** của C++. Chẳng hạn, **`new_node`** trả về chỉ số của một vùng đang trống của `workspace` (để thêm phần tử mới cho danh sách).

```
template <class Entry>
index List<Entry>::new_node()
/*
post: Trả về chỉ số của phần tử đầu tiên có thể sử dụng trong workspace; các thuộc tính
      available, last_used, và workspace được cập nhật nếu cần thiết .
      Nếu workspace thực sự đầy, trả về -1.
*/

{
   index new_index;

   if (available != -1) {                    // ngăn xếp chưa rỗng.
      new_index = available;                 // pop từ ngăn xếp.
      available = workspace[available].next; // cập nhật cho ngăn xếp.
   } else if (last_used<max_list - 1){// ngăn xếp rỗng và workspace chưa đầy.
      new_index = ++last_used;
   } else return -1;
```

```
   workspace[new_index].next = -1;
   return new_index;
}
```

```
template <class Entry>
void List<Entry>::delete_node(index old_index)
/*
pre:  Danh sách có một phần tử lưu tại chỉ số old_index.
post: Vùng nhớ có chỉ số old_index được đẩy vào ngăn xếp các chỗ trống có thể sử dụng lại;
      các thuộc tính available, last_used, và workspace được cập nhật nếu cần thiết.
*/
{
   index previous;
   if (old_index == head) head = workspace[old_index].next;
   else {
      previous = set_position(current_position(old_index) - 1);
      workspace[previous].next = workspace[old_index].next;
   }
   workspace[old_index].next = available;       // đẩy vào ngăn xếp.
   available = old_index;
}
```

Cả hai hàm này thực ra là thực hiện việc push và pop ngăn xếp. Nếu muốn chúng ta có thể viết các hàm xử lý ngăn xếp riêng rồi mới viết hàm sử dụng chúng.

Các hàm phụ trợ protected khác là **set_position** và **current_position**. Cũng giống như các hiện thực trước, **set_position** nhận vị trí (thứ tự) của phần tử trong danh sách và trả về chỉ số phần tử đó trong workspace. Hàm **current_position** nhận chỉ số phần tử trong workspace và trả về vị trí (thứ tự) của phần tử trong danh sách. Hiện thực của chúng được xem như bài tập.

```
index List<Entry>::set_position(int position) const;
pre: position là vị trí hợp lý trong danh sách; 0 ≤ position < count.
post: trả về chỉ số trong workspace của node tại vị trí position trong danh sách..
```

```
int List<Entry>::current_position(index n) const;
post: trả về vị trí trong danh sách của node tại chỉ số n trong workspace, hoặc –1 nếu không có
      node này.
```

4.5.3. Các tác vụ khác

Chúng ta hiện thực các phương thức xử lý cho DSLK trong mảng liên tục bằng cách thay đổi các phương thức đã có của DSLK trong phần 4.3.2. Phần lớn việc hiện thực này được dành lại xem như bài tập, ở đây chúng ta sẽ viết hàm duyệt danh sách và thêm phần tử mới vào danh sách.

```
template <class Entry>
void List<Entry>::traverse(void (*visit)(Entry &))
```

```
/*
post: Công việc cần làm bởi hàm *visit được thực hiện trên từng phần tử của danh sách, bắt
      đầu tử phần tử thứ 0.
*/
{
   for (index n = head; n != -1; n = workspace[n].next)
      (*visit)(workspace[n].entry);
}
```

So sánh phương thức này với phương thức tương ứng đối với DSLK dùng con trỏ và bộ nhớ động trong phần 4.3.2, chúng ta dễ dàng nhận thấy mỗi dòng lệnh là một sự chuyển đổi đơn giản một dòng lệnh của hiện thực trước. Bằng cách chuyển đổi tương tự chúng ta cũng có được phương thức thêm một phần tử mới vào DSLK trong mảng liên tục.

```
template <class Entry>
Error_code List<Entry>::insert(int position, const Entry &x)
/*
post: Nếu danh sách chưa đầy và 0 <= position <= n, với n là số phần tử hiện có trong
      danh sách, phương thức sẽ thực hiện thành công việc chèn x vào tại position và các
      phần tử phía sau bị đẩy lùi thứ tự bởi 1 đơn vị. Ngược lại, phương thức trả về mã lỗi thích
      hợp.
*/
{
   index new_index, previous, following;

   if (position < 0 || position > count) return range_error;

   if (position > 0) {                                  } Tìm phần tử trước và sau vị trí
      previous = set_position(position - 1);            } cần thêm phần tử mới.
      following = workspace[previous].next;             }
   }
   else following = head;
   if ((new_index = new_node()) == -1) return overflow; // Tìm vùng trống.
   workspace[new_index].entry = x;              // Cập nhật dữ liệu vào vùng trống.
   workspace[new_index].next = following;

   if (position == 0)
      head = new_index; // Trường hợp đặc biệt: thêm                    }
                                vào đầu DSLK.                           } Thêm phần tử mới
   else                                                                 } vào DSLK.
      workspace[previous].next = new_index;                             }
   count++;
   return success;
}
```

4.5.4. Các biến thể của danh sách liên kết trong mảng liên tục

Mảng liên tục cùng các chỉ số không những được dùng cho DSLK, chúng còn có hiệu quả tương tự đối với DSLK kép hoặc với vài biến thể khác. Đối với DSLK kép, khả năng áp dụng phép tính số học cho các chỉ số cho phép một hiện thực mà trong đó các tham chiếu tới và lui chỉ cần chứa trong một vùng chỉ số đơn (sử dụng cả trị âm lẫn trị dương cho các chỉ số).

Chương 5 – CHUỖI KÝ TỰ

Trong phần này chúng ta sẽ hiện thực một lớp biểu diễn một chuỗi nối tiếp các ký tự. Ví dụ ta có các chuỗi ký tự: "Đây là một chuỗi ký tự", "Tên?" trong đó cặp dấu " " không phải là bộ phận của chuỗi ký tự. Một chuỗi ký tự rỗng được ký hiệu "". Chuỗi ký tự cũng là một danh sách các ký tự. Tuy nhiên, các tác vụ trên chuỗi ký tự có hơi đặc biệt và khác với các tác vụ trên một danh sách trừu tượng mà chúng ta đã định nghĩa, chúng ta sẽ không dẫn xuất lớp chuỗi ký tự từ một lớp `List` nào trước đây.

Trong các tác vụ thao tác trên chuỗi ký tự, tác vụ tìm kiếm là khó khăn nhất. Chúng ta sẽ tìm hiểu hai giải thuật tìm kiếm vào cuối chương này. Trong phần đầu, chúng ta đặc biệt quan tâm đến việc khắc phục tính thiếu an toàn của chuỗi ký tự trong ngôn ngữ C mà đa số người lập trình đã từng sử dụng. Do đó phần trình bày tiếp theo đây liên quan chặt chẽ đến ngôn ngữ C và C++.

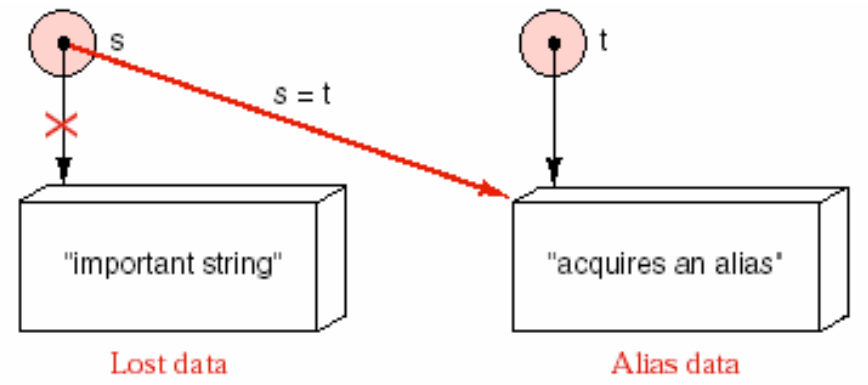
5.1. Chuỗi ký tự trong C và trong C++

Ngôn ngữ `C++` cung cấp hai cách hiện thực chuỗi ký tự. Cách nguyên thủy là hiện thực `string` của `C`. Giống như những phần khác, hiện thực `string` của ngôn ngữ `C` có thể chạy trong mọi hiện thực của `C++`. Chúng ta sẽ gọi các đối tượng `string` cung cấp bởi `C` là **`C-String.`** `C-String` thể hiện cả các điểm mạnh và cả các điểm yếu của ngôn ngữ `C`: chúng rất phổ biến, rất hiệu quả nhưng cũng rất hay bị dùng sai. `C-String` liên quan đến một loạt các tập quán mà chúng ta sẽ xem lại dưới đây.

Một `C-String` có kiểu **`char*`**. Do đó, một `C-String` tham chiếu đến một địa chỉ trong bộ nhớ; địa chỉ này là điểm bắt đầu của tập các bytes chứa các ký tự trong chuỗi ký tự. Vùng nhớ chiếm bởi một chuỗi ký tự phải được kết thúc bằng một ký tự đặc biệt **`'\0'`**. Trình biên dịch không thể kiểm tra giúp quy định này, sự thiếu sót sẽ gây lỗi thời gian chạy. Nói cách khác, `C-String` không có tính đóng kín và thiếu an toàn.

Tập tin chuẩn `<cstring>` chứa thư viện các hàm xử lý `C-String`. Trong các trình biên dịch `C++` cũ, tập tin này thường có tên là `<string.h>`. Các hàm thư viện này rất tiện lợi, hiệu quả và chứa hầu hết các tác vụ trên chuỗi ký tự mà chúng ta cần. Giả sử s và t là các `C-String`. Tác vụ `strlen(s)` trả về chiều dài của s, `strcmp(s,t)` so sánh từng ký tự của s và t, và `strstr(s,t)` trả về con trỏ tham chiếu đến vị trí bắt đầu của t trong s. Ngoài ra, trong `C++` tác vụ xuất `<<` được định nghĩa lại cho `C-String`, nhờ vậy, lệnh đơn giản `<< s` sẽ in chuỗi ký tự s.

Mặc dù hiện thực `C-String` có nhiều ưu điểm tuyệt vời, nhưng nó cũng có những nhược điểm nghiêm trọng. Thực vậy, nó có những vấn đề mà chúng ta đã gặp phải khi nghiên cứu CTDL ngăn xếp liên kết trong chương 2 cũng như các CTDL có chứa thuộc tính con trỏ nói chung. Thật dễ dàng khi người sử dụng có thể tạo bí danh cho chuỗi ký tự, cũng như gây nên rác. Trong hình 5.1, chúng ta thấy rõ phép gán s = t dẫn đến cả hai vấn đề trên.



Hình 5.1- Sự thiếu an toàn của `C-String`.

Một vấn đề khác cũng thường nảy sinh trong các ứng dụng có sử dụng `C-String`. Một `C-String` chưa khởi tạo cần được gán `NULL`. Tuy nhiên, rất nhiều hàm thư viện của `C-String` sẽ gặp sự cố trong thời gian chạy khi gặp đối tượng `C-String` là NULL. Chẳng hạn, lệnh

```
char* x = NULL;
cout << strlen(x);
```

được một số trình biên dịch chấp nhận, nhưng với nhiều hiện thực khác của thư viện `C-String`, thì gặp lỗi trong thời gian chạy. Do đó, người sử dụng phải kiểm tra kỹ lưỡng điều kiện trước khi gọi các hàm thư viện.

Trong `C++`, việc đóng gói `string` vào một lớp có tính đóng kín và an toàn được thực hiện dễ dàng. Thư viện chuẩn STL có lớp `String` an toàn chứa trong tập tin `<string>`. Thư viện này hiện thực lớp có tên `std::String` vừa tiện lợi, an toàn vừa hiệu quả.

Trong phần này chúng ta sẽ tự xây dựng một lớp `String` để có dịp hiểu kỹ về cách tạo nên một CTDL có tính đóng kín và an toàn cao. Chúng ta sẽ không phải viết lại toàn bộ mà chỉ sử dụng lại thư viện đã có `C-String`.

5.2. Đặc tả của lớp `String`

Để tạo một hiện thực lớp `String` an toàn, chúng ta đóng gói `C-String` như một thuộc tính thành phần của nó và để thuận tiện hơn, chúng ta thêm một thuộc tính chiều dài cho chuỗi ký tự. Do thuộc tính char* là một con trỏ, chúng ta cần thêm các tác vụ gán định nghĩa lại (*overloaded assignment*), *copy constructor*, *destructor*, để lớp `String` của chúng ta tránh được các vấn đề bí danh, tạo rác, hoặc việc sử dụng đối tượng mà chưa được khởi tạo.

5.2.1. Các phép so sánh

Với một số ứng dụng, sẽ hết sức thuận tiện nếu chúng ta bổ sung thêm các tác vụ so sánh <, >, <=, >=, ==, != để so sánh từng cặp đối tượng `String` theo từng ký tự. Vì thế, lớp `String` của chúng ta sẽ chứa các tác vụ so sánh được định nghĩa lại (*overloaded comparison operators*).

5.2.2. Một số *constructor* tiện dụng

Tạo đối tượng `String` từ một `C-String`

Chúng ta sẽ xây dựng *constructor* với thông số `char*` cho lớp `String`. *Constructor* này cung cấp một cách chuyển đổi thuận tiện một đối tượng `C-String` sang đối tượng `String`. Việc chuyển đổi thông qua cách gọi tường minh như sau:

```
String s(“some_string”);
```

Trong lệnh này, đối tượng `String` s được tạo ra chứa dữ liệu là `“some_string”.`

Constructor này đôi khi còn được gọi một cách không tường minh bởi trình biên dịch mỗi khi chương trình cần đến sự ép kiểu (*type cast*) từ kiểu `char*` sang `String`. Lấy ví dụ,

```
String s;
s = “some_string”;
```

Để chạy lệnh thứ hai, trình biên dịch `C++` trước hết gọi *constructor* của chúng ta để chuyển `“some_string”` thành một đối tượng `String` tạm. Sau đó phép gán định nghĩa lại của `String` được gọi để chép đối tượng tạm này vào s. Cuối cùng *destructor* cho đối tượng tạm được thực hiện.

Tạo đối tượng `String` từ một danh sách các ký tự

Tương tự, chúng ta cũng nên có *constructor* để chuyển một danh sách các ký tự sang một đối tượng `String`. Chẳng hạn, khi đọc một chuỗi ký tự từ người sử dụng, chúng ta nên đọc từng ký tự vào một danh sách các ký tự do chưa biết trước

chiều dài của nó. Sau đó chúng ta sẽ chuyển đổi danh sách này sang một đối tượng `String`.

Chuyển từ một đối tượng `String` sang một `C-String`

Cuối cùng, nếu có thể chuyển đổi ngược từ một đối tượng `String` sang một đối tượng `C-String` thì sẽ rất có lợi cho những trường hợp `string` cần được xem là `char*`. Đó là những lúc chúng ta cần sử dụng lại các hàm thư viện của `C-String` cho các đối tượng `String`. Phương thức này sẽ được gọi là `c_str()` và phải trả về `const char*` là một con trỏ tham chiếu đến dữ liệu biểu diễn `String`. Phương thức `c_str()` có thể được gọi như sau:

```
String s = “some_String”;
const char* new_s = s.c_str();
```

Điều quan trọng ở đây là `c_str()` trả về một `C-String` như là các ký tự hằng. Chúng ta có thể thấy được sự cần thiết này nếu chúng ta xem xét đến vùng nhớ chiếm bởi chuỗi ký tự `new_s`. Vùng nhớ này rõ ràng là thuộc đối tượng của lớp `String`. Chúng ta thấy rằng lớp `String` nên chịu trách nhiệm về vùng nhớ này, vì điều đó cho phép chúng ta hiện thực hàm chuyển đổi một cách hiệu quả, đồng thời tránh được cho người sử dụng khỏi phải chịu trách nhiệm về việc quên xóa một `C-String` đã được chuyển đổi từ một đối tượng `String`. Do đó, chúng ta khai báo `c_str()` trả về `const char*` để người sử dụng không thể sử dụng con trỏ trả về này mà thay đổi các ký tự dữ liệu được tham chiếu đến, sự thay đổi này chỉ thuộc quyền của lớp `String` mà thôi.

Với một số ít đặc tính được mô tả trên chúng ta có được một cách xử lý chuỗi ký tự vô cùng linh hoạt, hiệu quả và an toàn. Lớp `String` của chúng ta là một ADT đóng kín hoàn toàn, nhưng nó cung cấp một giao diện thật đầy đủ.

Chúng ta có đặc tả lớp `String` như sau:

```
class String {
public:
   String();
   ~String();
   String (const String &copy);   //  copy constructor
   String (const char * copy);    //  Chuyển đổi từ C-string
   String (List<char> &copy);     //  Chuyển đổi từ List các ký tự

   void operator =(const String &copy);
   const char *c_str() const;     //  Chuyển đổi sang C-string

protected:
   char *entries;
   int length;
};
```

```
bool operator ==(const String &first, const String &second);
bool operator >(const String &first, const String &second);
bool operator <(const String &first, const String &second);
bool operator >=(const String &first, const String &second);
bool operator <=(const String &first, const String &second);
bool operator !=(const String &first, const String &second);
```

5.3. Hiện thực lớp **String**

Các *constructor* chuyển đổi C-String và danh sách các ký tự sang đối tượng String:

```
String::String (const char *in_string)
/*
pre:  Con trỏ in_string tham chiếu đến một C-string.
post: Đối tượng String được khởi tạo từ chuỗi ký tự C-string in_string, và nó nắm giữ
      một bản sao của in_string, chuỗi ký tự trong in_string không thay đổi.
*/
{
   length = strlen(in_string);
   entries = new char[length + 1];
   strcpy(entries, in_string);
}
```

```
String::String (List<char> in_list)
/*
post: Đối tượng String được khởi tạo từ danh sách các ký tự trong đối tượng List, và nó nắm
      giữ một bản sao khác, đối tượng in_list không thay đổi.
*/
{
   length = in_list.size();
   entries = new char[length + 1];
   for (int i = 0; i < length; i++) in_list.retrieve(i,entries[i]);
   entries[length] = '\0';
}
```

Chúng ta chọn cách hiện thực phương thức chuyển đổi đối tượng String sang const char* như sau:

```
const char*String::c_str() const
/*
post: trả về con trỏ chỉ ký tự đầu tiên của chuỗi ký tự trong đối tượng String. Lưu ý rằng ở đây
      có việc chia sẻ cùng một chuỗi ký tự.
*/
{
   return (const char *) entries;
}
```

Cách hiện thực này cũng không hoàn toàn thích đáng do nó cho phép truy xuất dữ liệu bên trong của đối tượng String. Tuy nhiên chúng ta sẽ thấy những

cách giải quyết khác cũng gặp một số vấn đề. Cách giải quyết này còn có được ưu điểm là tính hiệu quả.

Phương thức `c_str()` trả về con trỏ chỉ đến mảng các ký tự chỉ có thể đọc chứ không thể sửa đổi do chúng ta đã ép kiểu sang `const char*`. Tuy nhiên người lập trình có thể ép kiểu ngược trở lại và gán vào một con trỏ khác làm phá vỡ tính đóng kín của dữ liệu của chúng ta. Một vấn đề nghiêm trọng hơn chính là bí danh được tạo bởi phương thức này. Chúng ta thấy rằng người lập trình nên sử dụng con trỏ trả về ngay sau khi vừa gọi phương thức, nếu không những gì xảy ra sẽ không lường trước được. Lấy ví dụ sau:

```
String s = "abc";
const char *new_string = s.c_str();
s = "def";
cout << new_string;
```

Lệnh `s = "def"` đã làm thay đổi dữ liệu mà `new_string` chỉ đến.

Một chiến lược khác cho phương thức `c_str()` có thể là định vị vùng nhớ động mới để chép dữ liệu của đối tượng `String` sang. Cách hiện thực này rõ ràng là kém hiệu quả hơn, đặc biệt đối với `String` dài. Ngoài ra nó còn có một nhược điểm nghiêm trọng, đó là khả năng tạo rác. `String` mà `c_str()` trả về không còn chia sẻ dữ liệu với đối tượng `String` nữa, và như vậy người lập trình phải nhớ `delete` nó khi không còn sử dụng. Chẳng hạn, nếu chỉ việc in ra như dưới đây thì trong bộ nhớ đã để lại rác do cách hiện thực vừa nêu.

```
String s = "Some very long string";
cout << s.c_str();
```

Tóm lại, tuy chúng ta vẫn giữ phương án đầu tiên cho phương thức `c_str()`, nhưng người lập trình không nên sử dụng phương thức này vì nó phá vỡ tính đóng kín của đối tượng `String`, trừ khi muốn sử dụng lại các hàm thư viện của `C-String` và đã hiểu thật rõ về bản chất của sự việc.

Cuối cùng, chúng ta xem xét các tác vụ so sánh được định nghĩa lại. Hiện thực sau đây của phép so sánh bằng được định nghĩa lại thật ngắn gọn và hiệu quả nhờ phương thức `c_str()`.

```
bool operator ==(const String &first, const String &second)
/*
post: Trả về true nếu đối tượng first giống đối tượng second. Ngược lại trả về false.
*/
{
   return strcmp(first.c_str(), second.c_str()) == 0;
}
```

Các tác vụ so sánh định nghĩa lại khác có hiện thực hầu như tương tự.

5.4. Các tác vụ trên String

Chúng ta sẽ phát triển một số tác vụ làm việc trên các đối tượng String. Trong nhiều trường hợp, các hàm của C-String có thể được gọi trực tiếp cho các đối tượng String đã chuyển đổi:

```
String s = "some_string";
cout << s.c_str() << endl;
cout << strlen(s.c_str()) << endl;
```

Đối với những hàm không thay đổi các thông số String như strcpy, chúng ta sẽ viết các phiên bản định nghĩa lại có thông số là đối tượng String thay vì char*. Như chúng ta đã biết, trong C++, một hàm được gọi là có định nghĩa lại nếu hai hoặc ba phiên bản khác nhau của nó có trong cùng một chương trình. Chúng ta đã có các *constructor* và các tác vụ gán định nghĩa lại. Khi một hàm được định nghĩa lại, chúng phải có các thông số khác nhau. Căn cứ vào các thông số được gởi khi gọi hàm, trình biên dịch biết được cần phải sử dụng phiên bản nào.

Phiên bản định nghĩa lại cho strcat có khai báo như sau:

```
void strcat(String &add_to, const String &add_on)
```

Người sử dụng có thể gọi strcat(s,t) để nối chuỗi ký tự t vào chuỗi ký tự s. s là một String, t có thể là String hoặc C-String. Nếu t là C-String thì trước hết *constructor* có thông số char* sẽ thực hiện để chuyển t thành một đối tượng String cho hợp kiểu thông số mà strcat yêu cầu.

```
void strcat(String &add_to, const String &add_on)
/*
post: String add_on được nối vào sau String add_to.
*/
{
   const char *cfirst = add_to.c_str();
   const char *csecond = add_on.c_str();
   char *copy = new char[strlen(cfirst) + strlen(csecond) + 1];
   strcpy(copy, cfirst);
   strcat(copy, csecond);
   add_to = copy;
   delete []copy;
}
```

Trong phương thức trên có gọi strcat với hai thông số là char* và const char*, tại đây trình biên dịch sẽ gọi đúng hàm thư viện của C-String chứ không phải gọi đệ quy chính phương thức này.

Do add_to là đối tượng String, lệnh add_to = copy trước hết gọi *constructor* để chuyển copy kiểu char* sang đối tượng String, sau đó mới gọi phép gán định nghĩa lại của lớp String. Nói cách khác, điều này dẫn đến việc

chép chuỗi ký tự hai lần. Để tránh điều này chúng ta hãy thử thay đổi dòng lệnh. Chẳng hạn, một cách đơn giản chúng ta khai báo `strcat` là một hàm *friend* của lớp `String`. Khi đó chúng ta có thể truy cập đến thuộc tính `entries` của lớp `String: add_to.entries = copy.`

Chúng ta cần hàm để đọc các đối tượng `String`. Chúng ta có thể thực hiện tương tự như đối với `C-String`, tác vụ << sẽ được định nghĩa lại để nhận thông số là một `String`. Tuy nhiên, chúng ta cũng có thể dùng cách khác để xây dựng hàm `read_in` như sau:

```
String read_in(istream &input)
/*
post: Trả về một đối tượng String đọc từ thông số istream (ký tự kết thúc chuỗi ký tự được
      quy ước là ký tự xuống hàng hoặc kết thúc tập tin)
*/
{
   List<char> temp;
   int size = 0;
   char c;
   while ((c = input.peek()) != EOF && (c = input.get()) != '\n')
      temp.insert(size++, c);
   String answer(temp);
   return answer;
}
```

Hàm trên sử dụng một đối tượng `temp` để gom các ký tự từ thông số `input`, sau đó gọi *constructor* để chuyển đổi `temp` này thành đối tượng `String`. Ký tự kết thúc chuỗi ký tự là ký tự xuống hàng hoặc ký tự kết thúc tập tin.

Một phiên bản được đề nghị khác cho hàm `read_in` là thêm thông số thứ hai để chỉ ra ký tự kết thúc chuỗi ký tự mong muốn:

```
String read_in(istream &input, int &terminator);

post: Trả về một đối tượng String đọc từ thông số istream (ký tự kết thúc chuỗi ký tự được quy ước
      là ký tự xuống hàng hoặc kết thúc tập tin, ký tự này cũng được trả về thông qua tham biến
      terminator.)
```

Tương tự chúng ta có phương thức để in một đối tượng `String`:

```
void write(String &s)
/*
post: Đối tượng String s được in ra cout.
*/
{
   if (strlen(s.c_str())>0)
      cout << s.c_str() << endl;
}
```

Trong các phần tiếp theo chúng ta sẽ sử dụng các hàm thư viện cho lớp `String` như sau:

```
void strcpy(String &copy, const String &original);
```
post: Hàm chép **String original** sang **String copy**.

```
void strncpy(String &copy, const String &original, int n);
```
post: Hàm chép nhiều nhất là n ký tự từ **String original** sang **String copy**.

```
int strstr(const String &text, const String &target);
```
post: Nếu **String target** là chuỗi con (*subtring*) của **String text**, hàm trả về vị trí xuất hiện đầu tiên của `target` trong `text`; ngược lại, hàm trả về -1.

Các hiện thực của các hàm này theo cách sử dụng lại thư viện `C-String` được xem như bài tập.

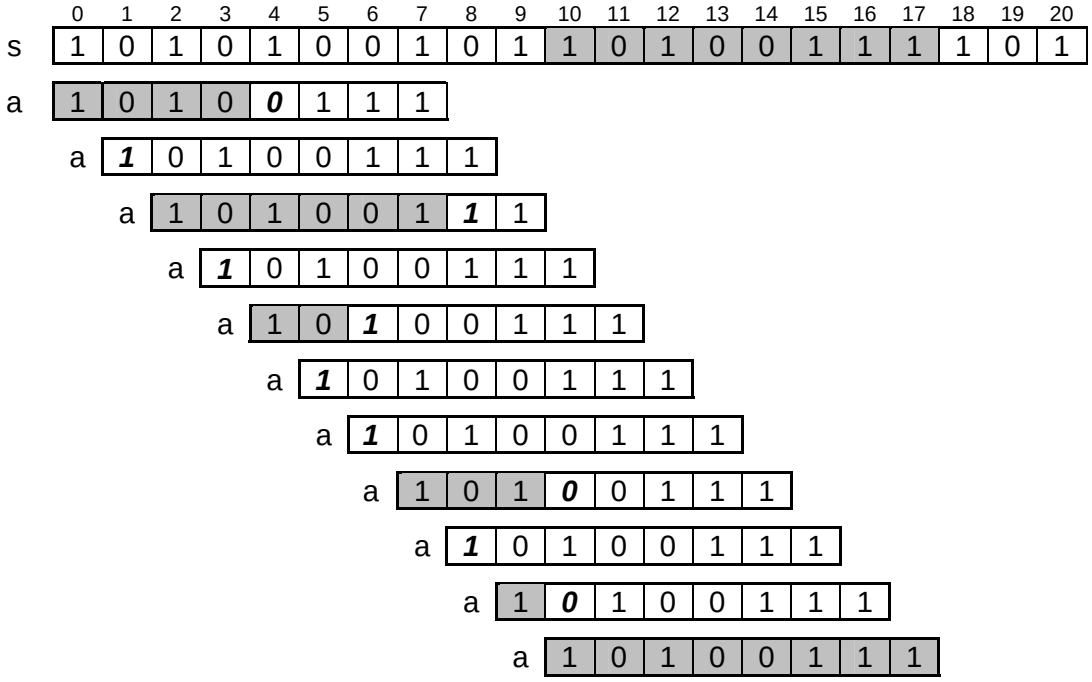
5.5. Các giải thuật tìm một chuỗi con trong một chuỗi

Phần sau đây chúng ta sẽ tìm hiểu lại cách hiện thực của một vài hàm thư viện của `C-String`. Các phép xử lý cơ bản trên chuỗi ký tự bao gồm: tìm một chuỗi con trong một chuỗi, thay thế một chuỗi con bằng một chuỗi khác, chèn một chuỗi con vào một chuỗi, loại một chuỗi con trong một chuỗi,... Trong đó phép tìm một chuỗi con trong một chuỗi có thể xem là phép cơ bản nhất, những phép còn lại có thể được thực hiện dễ dàng sau khi đã xác định được vị trí của chuỗi con. Chúng ta sẽ tìm hiểu hai giải thuật tìm chuỗi con trong một chuỗi cho trước.

5.5.1. Giải thuật Brute-Force

Ý tưởng giải thuật này vô cùng đơn giản, đó là thử so trùng chuỗi con tại mọi vị trí bắt đầu trong chuỗi đã cho (Hình 5.2). Giả sử chúng ta cần tìm vị trí của chuỗi a trong chuỗi s. Các vị trí bắt đầu so trùng a trên s là 0, 1, 2, ...Mỗi lần so trùng, chúng ta lần lượt so sánh từng cặp ký tự của a và s từ trái sang phải. Khi gặp hai ký tự khác nhau, chúng ta lại phải bắt đầu so trùng từ đầu chuỗi a với vị trí mới. Vị trí bắt đầu so trùng trên s lần thứ i sẽ là vị trí bắt đầu so trùng trên s lần thứ i-1 cộng thêm 1. Các ký tự in nghiêng trong hình vẽ bên dưới là vị trí thất bại trong một lần so trùng, phần có nền xám bên trái chúng là những ký tự so trùng đã thành công. Một lần so trùng nào đó mà chúng ta đã duyệt qua được hết chiều dài của a xem như đã tìm thấy a trong s và giải thuật dừng.

Cho i là chỉ số chạy trên s và j là chỉ số chạy trên a, j luôn được gán về 0 khi bắt đầu một lần so trùng. Khi gặp thất bại trong một lần so trùng nào đó thì cả i và j đều đã tiến được j bước so với lúc bắt đầu so trùng. Do đó để bắt đầu so trùng cho lần sau, i cần lùi về j-1 bước.



Hình 5.2- Minh họa giải thuật Brute-Force

```
// Giải thuật Brute-Force
int strstr(const String &s, const String &a);
/*
post: Nếu chuỗi a là chuỗi con của chuỗi  s, hàm trả về vị trí xuất hiện đầu tiên của a trong
s; ngược lại, hàm trả về -1.
*/
{
   int   i = 0, // Chỉ số chạy trên s.
         j = 0, // Chỉ số chạy trên a.
         ls = s.strlen(); // Số ký tự của s.
         la = a.strlen(), // Số ký tự của a.
   const char* pa = a.c_str(); //Địa chỉ ký tự đầu tiên của a.
   const char* ps = s.c_str(); //Địa chỉ ký tự đầu tiên của s.
   do {
      if (pa[j] == ps[i]){
         i++;
         j++;
      };
      else {
         i = i - (j - 1); // Lùi về cho lần so trùng kế tiếp.
         j = 0;
      }
   } while ((j<la) && (i<ls));
   if (j>=la) return i - la;
   else return -1;
}
```

Trường hợp xấu nhất của giải thuật Brute-Force là chuỗi con a trùng với phần cuối cùng của chuỗi s. Khi đó chúng ta đã phải lập lại `ls-la+1` lần so trùng, với

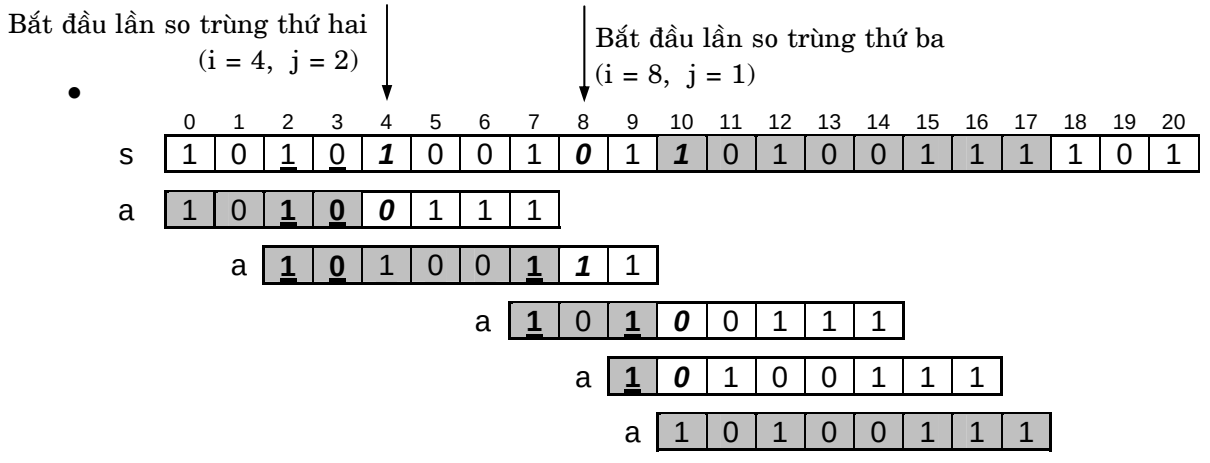
`ls` và `la` là chiều dài của chuỗi s và chuỗi a. Mỗi lần so trùng đã phải so sánh `la` ký tự. Số lần so sánh tối đa là `la*(ls-la+1)` ≈ `la*ls`.

5.5.2. Giải thuật Knuth-Morris-Pratt

Giải thuật này do Knuth, Morris và Pratt đưa ra, còn gọi là giải thuật KMP-Search.

Trong ví dụ trên chúng ta thấy giải thuật Brute-Force phải so trùng đến lần thứ 11 mới phát hiện được vị trí cần tìm. Giải thuật `KMP-Search` dưới đây tiết kiệm được một số lần so trùng và chỉ phải so trùng đến lần thứ 5. Hơn thế nữa, chỉ số i chạy trên s cũng không bao giờ phải lùi lại. Để có được điều này, chúng ta hãy cố gắng rút ra nhận xét từ hình 5.3 bên dưới. Trong lần so trùng thứ nhất, khi i=4 thì $a_j \neq s_i$, khi đó a sẽ được dịch chuyển về phía phải sao cho đoạn đầu của a trùng khớp với đoạn cuối của a trong phần đã được duyệt qua (chỉ tính phần màu xám). Trong hình vẽ là hai ký tự 1 và 0 có gạch dưới. Lần so trùng kế tiếp chính là từ vị trí này, và những lần so trùng trung gian giữa hai lần này có thể bỏ qua. Điều này có thể lý giải như sau: nếu phần đầu của a trùng với phần cuối của a thì nó cũng trùng với phần tương ứng của s bên trên, do phần cuối của a vừa mới được so trùng thành công với phần tương ứng của s. Được như vậy thì i mới hoàn toàn không phải lùi lại. Trong lần so trùng mới, chính s_i này sẽ được so sánh với a_j, với j sẽ được tính toán thích hợp mà chúng ta sẽ bàn đến sau. Trong ví dụ chúng ta thấy j = 2, lần so sánh đầu tiên của lần so trùng thứ hai là so sánh giữa s_4 và a_2.

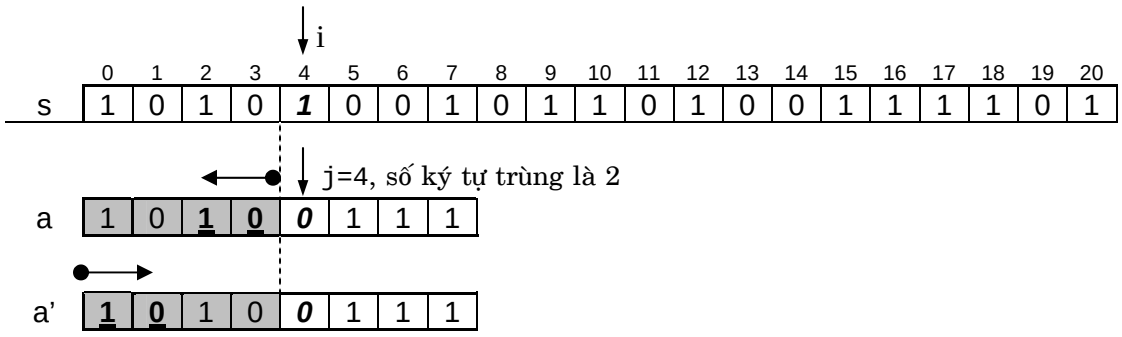
Tương tự, khi lần so trùng thứ hai thất bại tại s_8, chuỗi con a sẽ được dịch chuyển rất xa, tiết kiệm được rất nhiều lần so trùng. Chúng ta dễ dàng kiểm chứng, với những vị trí trung gian khác, phần đầu của a không trùng với phần cuối (chỉ tính phần màu xám) của a, nên cũng không thể trùng với phần tương ứng trên s, có thực hiện so trùng cũng sẽ thất bại mà thôi.



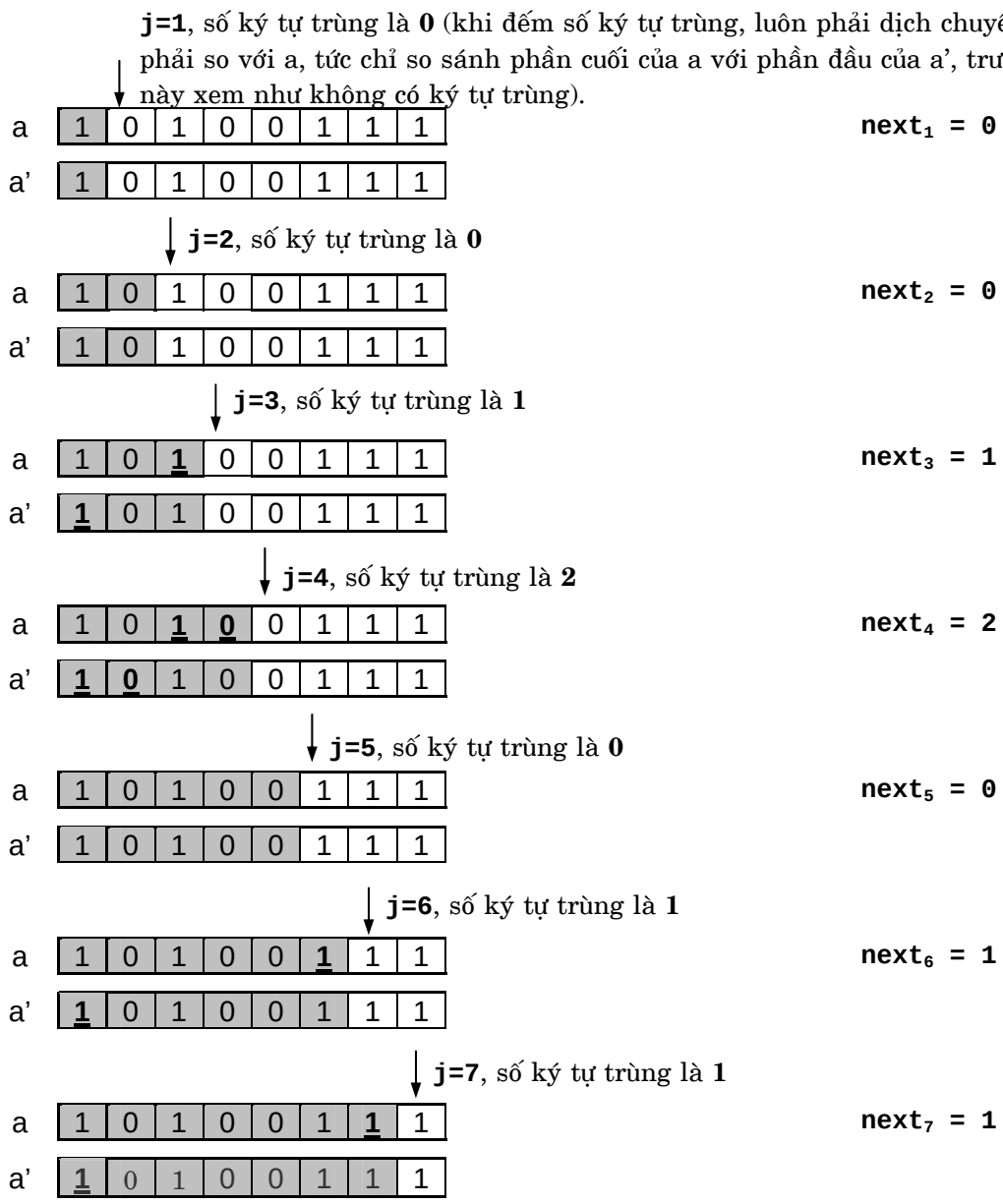
Hình 5.3- Minh họa giải thuật Knuth-Morris-Pratt

Hình vẽ dưới đây giúp chúng ta hiểu được cách tính chỉ số j thích hợp cho đầu mỗi lần so trùng (trong khi i không lùi về mà giữ nguyên để tiếp tục tiến tới).

Trích từ hình vẽ trên, chúng ta có được kết quả sau đây.
Xét vị trí `i = 4, j = 4`, do so sánh s_i với a_j thất bại, chúng ta đang muốn biết phần cuối của `a` kể từ điểm này trở về trước (tức chỉ tính phần màu xám) và phần đầu của `a` trùng được bao nhiêu ký tự. Gọi `a' = a`. Chúng ta sẽ nhìn quét từ cuối phần màu xám của `a` và từ đầu của `a'`, chúng ta sẽ biết được có bao nhiêu ký tự trùng. Đó là hai ký tự `1` và `0` được gạch dưới.



Như vậy, điều này hoàn toàn không còn phụ thuộc vào s nữa. Chúng ta có thể tính số ký tự trùng theo j dựa trên a và a'. Đồng thời ta thấy số ký tự trùng này cũng là chỉ số mà j phải lùi về cho lần so trùng kế tiếp a_j với s_i, i không đổi. Chúng ta bắt đầu với j = 1 và xem hình 5.4 sau đây.



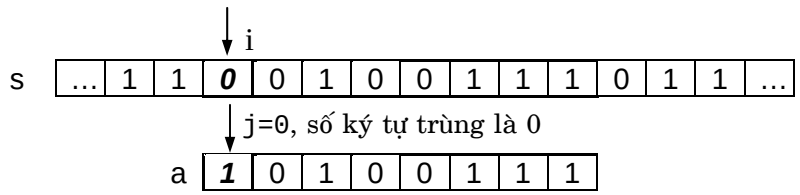
Hình 5.4- Minh họa giải thuật Knuth-Morris-Pratt

Giả sử chúng ta đã tạo được danh sách **next** mà phần tử thứ j chứa trị mà j phải lùi về khi đang so sánh a_j với s_i mà thất bại ($a_j \neq s_i$), để bắt đầu lần so trùng kế tiếp (i giữ nguyên không đổi). Hình 5.4 cho thấy $next_1$ luôn bằng 0 với mọi a. Chúng ta có giải thuật `KMP-Search` như dưới đây.

Lần so trùng thứ nhất luôn bắt đầu từ ký tự đầu của s và a, nên hai chỉ số i và j đều là 0.

- Trường hợp dễ hiểu nhất là trong khi mà $a_j=s_i$ thì i và j đều được nhích tới. Điều kiện dừng của vòng lặp hoàn toàn như giải thuật Brute-Force trên, có nghĩa là j đi được hết chiều dài của a (tìm thấy a trong s), hoặc i đi quá chiều dài của s (việc tìm kết thúc thất bại).

- Trường hợp $a_j \neq s_i$ (với $j \neq 0$) trong một lần so trùng nào đó thì như đã nói ở trên, chỉ việc cho j lùi về vị trí đã được chứa trong phần tử thứ j trong danh sách next. Nhờ vậy trong lần lặp kế tiếp sẽ tiếp tục so sánh a_j này với s_i mà i không đổi.
- Riêng trường hợp đặc biệt, với j = 0 mà $a_j \neq s_i$, ta xem hình dưới đây



Bất cứ một lần so sánh s_i nào đó với a_0 mà thất bại thì chuỗi a cũng phải dịch chuyển sang phải một bước, để lần so sánh kế tiếp (cũng là lần so trùng mới) có thể so sánh $\mathbf{a_0}$ với $\mathbf{s_{i+1}}$. Như vậy ta chỉ cần tăng i và giữ nguyên j mà thôi.

```
// Giải thuật Knuth- Morris – Pratt

int strstr(const String &s, const String &a);
/*
post: Nếu a là chuỗi con của s, hàm trả về vị trí xuất hiện đầu tiên của a trong s;
      ngược lại, hàm trả về -1.
*/
{
   List<int> next;
   int   i = 0, // Chỉ số chạy trên s.
         j = 0, // Chỉ số chạy trên a.
         ls = s.strlen(); // Số ký tự của s.
         la = a.strlen(), // Số ký tự của a.
   const char* pa = a.c_str(); //Địa chỉ ký tự đầu tiên của a.
   const char* ps = s.c_str(); //Địa chỉ ký tự đầu tiên của s.
   InitNext(a, next);    // Khởi gán các phần tử next1, next2,…,nextla-1.
                         // Không sử dụng next0.
   do {
      if (pa[j]==ps[i]){// Vẫn còn ký tự trùng trong một lần so trùng
            i++;         // nào đó, i và j được quyền nhích tới.
            j++;
      }
      else
         if (j == 0)     // Đây là trường hợp đặc biệt, phải dịch a sang phải
            i++;         // một bước, có nghĩa là cho i nhích tới.
         else
            next.retrieve(j, j); // Cho j lùi về trị đã chứa trong nextj.
   } while ((j<la) && (i<ls));
   if (j>=la) return i – la;
   else return –1;
}
```

Sau đây chúng ta sẽ viết hàm InitNext gán các trị cho các phần tử của next, tức là tìm số phần tử trùng theo hình vẽ 5.4. Có một điều khá thú vị trong giải thuật này, đó chính là hàm tạo danh sách next lại sử dụng ngay chính danh sách này. Chúng ta thấy rằng để tìm số phần tử trùng như đã nói, chúng ta cần dịch chuyển a′ về bên phải so với a, mà việc dịch chuyển a′ trên a cũng hoàn toàn giống như việc dịch chuyển a trên s trong khi đi tìm a trong s.

```
// Hàm phụ trợ gán các phần tử cho danh sách next.

void InitNext(const String &a, List<int> &next);
/*
post: Gán các trị cho các phần tử của next dựa trên chuỗi ký tự a.
*/
{
   int   i = 1, // Chỉ số chạy trên a.
         j = 0, // Chỉ số chạy trên a'.
         la = a.strlen(), // Số ký tự của a (cũng là của a').
   const char* pa = a.c_str(); //Địa chỉ ký tự đầu tiên của a (cũng là của a').
   next.clear();
   next.insert(1, 0); // Luôn đúng với mọi a.
   do {
      if (pa[j]==pa[i]){    // Vẫn còn ký tự trùng trong một lần so trùng
      i++;                 // nào đó, i và j được quyền nhích tới.
         j++;                 // Từ vị trí i trên a trở về trước, j xem như đã
         next.insert(i, j);// quét được số phần tử trùng của a' so với a.
      }
      else
         if (j == 0){    // Trường hợp đặc biệt, phải dịch a sang phải
             i++;         // một bước, có nghĩa là cho i nhích tới.
            next.insert(i, j);
         };
         else
            next.retrieve(j, j); // Cho j lùi về trị đã chứa trong next_j.
   } while (i<la); // i=la là đã gán xong la phần tử của next,
                   // không sử dụng next_0.
}
```

Hàm tạo next được chép lại từ giải thuật KMP-Search trên, chỉ có vài điểm bổ sung như sau: với i chạy trên a và j chạy trên a′, và a′ luôn phải dịch phải so với a, chúng ta khởi gán i=1 và j=0.

Do i tăng đến đâu là chúng ta xem như đã so trùng xong phần cuối của a (kể từ vị trí i này trở về trước) với phần đầu của a′, nên $next_i$ đã được xác định. Trong quá trình so trùng, trong khi mà a_i vẫn còn bằng a'_j, i và j đều nhích tới. Vì vậy, chúng ta dễ thấy rằng j chính là số phần tử đã trùng được của a' so với a, chúng ta có phép gán $next_i$=j.

Khi $a_i \neq a'_j$, chúng ta sử dụng ý tưởng của KMP-Search là cho j lùi về $next_j$. Vấn đề còn lại cần kiểm chứng là giá trị của $next_j$ phải có trước khi nó được sử dụng. Do chúng ta đã gán vào $\mathbf{next_i}$ và đã sử dụng $\mathbf{next_j}$, mà i luôn luôn đi trước j, nên chúng ta hoàn toàn yên tâm về điều này.

Cuối cùng, chỉ còn một điều nhỏ mà chúng ta cần xem xét. Đó là trường hợp có nhiều phương án cho số ký tự trùng nhau. Chẳng hạn với a là "10101010111…" và j=8, số ký tự trùng khi dịch a'=a về bên phải so với a là:

```
                      ↓ Vị trí j đang xét
a        10101010111...          Số ký tự trùng là 6
a'         101010 10111...

a        1010101 0111...
a'           1010 1010111...     Số ký tự trùng là 4

a        101010 10111...
a'             10 101010111...   Số ký tự trùng là 2
```

Sinh viên hãy tự suy nghĩ xem cách chọn phương án nào là đúng đắn nhất và kiểm tra lại các đoạn chương trình trên xem chúng có cần phải được sửa đổi gì hay không.

Ngoài ra, giải thuật KMP-Search còn có thể cải tiến một điểm nhỏ, đó là trước khi gán $next_i$=j trong InitNext, chúng ta kiểm tra nếu pa_j=pa_i thì sẽ gán $next_i$=$next_j$. Do khi so trùng pa_i mà thất bại thì có lùi về pa_{nexti}=paj cũng sẽ thất bại, chúng ta nên lùi hẳn về pa_{nextj}.

Số lần so sánh tối đa trong KMP-Search là ls+la.

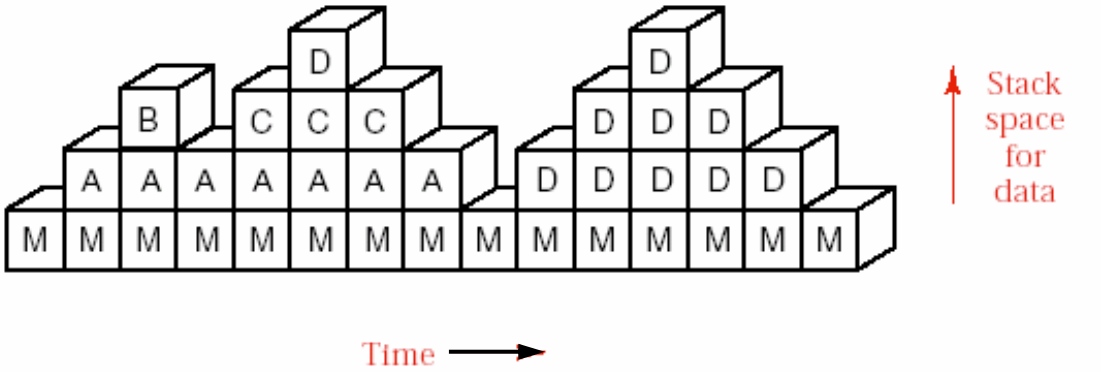
Chương 6 – ĐỆ QUY

Chương này trình bày về đệ quy (*recursion*) – một phương pháp mà trong đó để giải một bài toán, người ta giải các trường hợp nhỏ hơn của nó. Chúng ta cần tìm hiểu một vài ứng dụng và chương trình mẫu để thấy được một số trong rất nhiều dạng bài toán mà việc sử dụng đệ quy để giải rất có lợi. Một số ví dụ đơn giản, một số khác thực sự phức tạp. Chúng ta cũng sẽ phân tích xem đệ quy thường được hiện thực trong máy tính như thế nào, khi nào nên dùng đệ quy và khi nào nên tránh.

6.1. Giới thiệu về đệ quy

6.1.1. Cơ cấu ngăn xếp cho các lần gọi hàm

Khi một hàm gọi một hàm khác, thì tất cả các trạng thái mà hàm gọi đang có cần được khôi phục lại sau khi hàm được gọi kết thúc, để hàm này tiếp tục thực hiện công việc dở dang của mình. Trạng thái đó gồm có: điểm quay về (dòng lệnh kế sau lệnh gọi hàm); các trị trong các thanh ghi, vì các thanh ghi trong bộ xử lý sẽ được hàm được gọi sử dụng đến; các trị trong các biến cục bộ và các tham trị của nó. Như vậy mỗi hàm cần có một vùng nhớ dành riêng cho nó. Vùng nhớ này phải được tồn tại trong suốt thời gian kể từ khi hàm thực hiện cho đến khi nó kết thúc công việc.



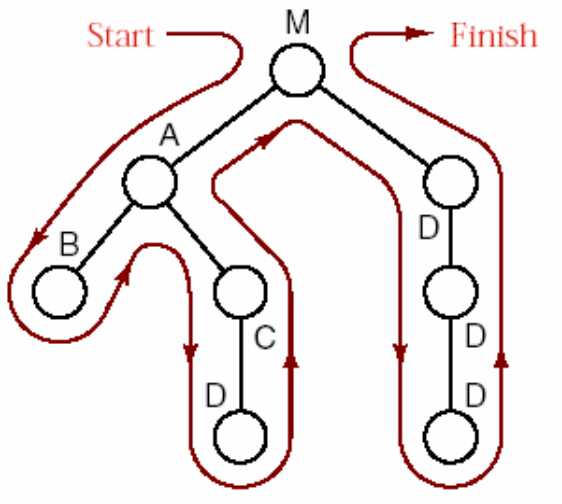
Hình 6.1- Cơ cấu ngăn xếp cho các lần gọi hàm

Giả sử chúng ta có ba hàm A, B, C, mà A gọi B, B gọi C. B sẽ không kết thúc trước khi C kết thúc. Tương tự, A khởi sự công việc đầu tiên nhưng lại kết thúc cuối cùng. Sự diễn tiến của các hoạt động của các hàm xảy ra theo tính chất vào sau ra trước (*Last In First Out –LIFO).* Nếu xét đến nhiệm vụ của máy tính trong việc tổ chức các vùng nhớ tạm dành cho các hàm này sử dụng, chúng ta thấy rằng các vùng nhớ này cũng phải nằm trong một danh sách có cùng tính chất trên, có nghĩa là ngăn xếp. Vì thế, ngăn xếp đóng một vai trò chủ chốt liên quan đến các hàm trong hệ thống máy tính. Trong hình 6.1, M biểu diễn chương trình chính, A, B, C là các hàm trên.

Hình 6.1 biểu diễn một dãy các vùng nhớ tạm cho các hàm, mỗi cột là hình ảnh của ngăn xếp tại một thời điểm, các thay đổi của ngăn xếp có thể được nhìn thấy bằng cách đọc từ trái sang phải. Hình ảnh này cũng cho chúng ta thấy rằng không có sự khác nhau trong cách đưa một vùng nhớ tạm vào ngăn xếp giữa hai trường hợp: một hàm gọi một hàm khác và một hàm gọi chính nó. **Đệ quy** là tên gọi trường hợp một hàm gọi chính nó, hay trường hợp các hàm lần lượt gọi nhau mà trong đó có một hàm gọi trở lại hàm đầu tiên. Theo cách nhìn của cơ cấu ngăn xếp, sự gọi hàm đệ quy không có gì khác với sự gọi hàm không đệ quy.

6.1.2. Cây biểu diễn các lần gọi hàm

Sơ đồ cây (*tree diagram*) có thể làm rõ hơn mối liên quan giữa ngăn xếp và việc gọi hàm. Sơ đồ cây hình 6.2 tương đương với cơ cấu ngăn xếp ở hình 6.1.



Hình 6.2- Cây biểu diễn các lần gọi hàm.

Chúng ta bắt đầu từ gốc của cây, tương ứng với chương trình chính. (Các thuật ngữ dùng cho các thành phần của cây có thể tham khảo trong chương 9) Mỗi vòng tròn gọi là nút của cây, tương ứng với một lần gọi hàm. Các nút ngay dưới gốc cây biểu diễn các hàm được gọi trực tiếp từ chương trình chính. Mỗi hàm trong số trên có thể gọi hàm khác, các hàm này lại được biểu diễn bởi các nút ở sâu hơn. Bằng cách này cây sẽ lớn lên như hình 6.2 và chúng ta gọi cây này là cây biểu diễn các lần gọi hàm.

Để theo vết các lần gọi hàm, chúng ta bắt đầu từ gốc của cây và di chuyển qua hết cây theo mũi tên trong hình 6.2. Cách đi này được gọi là **phép duyệt cây** (*traversal*). Khi đi xuống và gặp một nút, đó là lúc gọi hàm. Sau khi duyệt qua hết phần cây bên dưới, chúng ta gặp trở lại nút này, đó là lúc kết thúc hàm được gọi. Các nút lá biểu diễn các hàm không gọi một hàm nào khác.

Chúng ta đặc biệt chú ý đến đệ quy, do đó thông thường chúng ta chỉ vẽ một phần của cây biểu diễn sự gọi đệ quy, và chúng ta gọi là **cây đệ quy** (*recursion tree*). Trong sơ đồ cây chúng ta cũng lưu ý một điều là không có sự khác nhau giữa cách gọi đệ quy với cách gọi hàm khác. Sự đệ quy đơn giản chỉ là sự xuất hiện của các nút khác nhau trong cây có quan hệ nút trước – nút sau với nhau mà có cùng tên. Điểm thứ hai cần lưu ý rằng, chính vì cây biểu diễn **các lần gọi hàm**, nên trong chương trình, nếu một lệnh gọi hàm chỉ xuất hiện một lần nhưng lại nằm trong vòng lặp, thì nút biểu diễn hàm sẽ **xuất hiện nhiều lần** trong cây, mỗi lần tương ứng một lần gọi hàm. Tương tự, nếu lệnh gọi hàm nằm trong phần rẽ nhánh của một điều kiện mà điều kiện này không xảy ra thì nút biểu diễn hàm sẽ **không xuất hiện** trong cây.

Cơ cấu ngăn xếp ở hình 6.1 cho thấy nhu cầu về vùng nhớ của đệ quy. Nếu một hàm gọi đệ quy chính nó vài lần thì bản sao của các biến khai báo trong hàm được tạo ra cho mỗi lần gọi đệ quy. Trong cách hiện thực thông thường của đệ quy, chúng được giữ trong ngăn xếp. Chú ý rằng **tổng dung lượng vùng nhớ** cần cho ngăn xếp này **tỉ lệ với chiều cao** của cây đệ quy chứ **không phụ thuộc vào tổng số nút** trong cây. Điều này có nghĩa rằng, tổng dung lượng vùng nhớ cần thiết để hiện thực một hàm đệ quy phụ thuộc vào độ sâu của đệ quy, không phụ thuộc vào số lần mà hàm được gọi.

Hai hình ảnh trên cho chúng ta thấy mối liên quan mật thiết giữa một biểu diễn cây và ngăn xếp:

Trong quá trình duyệt qua bất kỳ một cây nào, các nút được thêm vào hay lấy đi đúng theo kiểu của ngăn xếp. Trái lại, cho trước một ngăn xếp, có thể vẽ một cây để mô tả quá trình thay đổi của ngăn xếp.

Chúng ta hãy tìm hiểu một vài ví dụ đơn giản về đệ quy. Sau đó chúng ta sẽ xem xét đệ quy thường được hiện thực trong máy tính như thế nào.

6.1.3. Giai thừa: Một định nghĩa đệ quy

Trong toán học. giai thừa của một số nguyên thường được định nghĩa bởi công thức:

$$n! = n \times (n-1) \times \ldots \times 1.$$

Hoặc định nghĩa sau:

$$n! = \begin{cases} 1 & \text{nếu } n=0 \\ n \times (n-1)! & \text{nếu } n>0. \end{cases}$$

Giả sử chúng ta cần tính 4!. Theo định nghĩa chúng ta có:

$$
\begin{aligned}
4! &= 4 \times 3! \\
&= 4 \times (3 \times 2!) \\
&= 4 \times (3 \times (2 \times 1!)) \\
&= 4 \times (3 \times (2 \times (1 \times 0!))) \\
&= 4 \times (3 \times (2 \times (1 \times 1))) \\
&= 4 \times (3 \times (2 \times 1)) \\
&= 4 \times (3 \times 2) \\
&= 4 \times 6 \\
&= 24
\end{aligned}
$$

Việc tính toán trên minh họa bản chất của cách mà đệ quy thực hiện. Để có được câu trả lời cho một bài toán lớn, phương pháp chung là giảm bài toán lớn thành một hoặc nhiều bài toán con có bản chất tương tự mà kích thước nhỏ hơn. Sau đó cũng **chính phương pháp chung này** lại được sử dụng cho những bài toán con, cứ như thế đệ quy sẽ tiếp tục cho đến khi kích thước của bài toán con đã giảm đến một kích thước nhỏ nhất nào đó của một vài trường hợp cơ bản, mà lời giải của chúng có thể có được một cách trực tiếp không cần đến đệ quy nữa. Nói cách khác:

Mọi quá trình đệ quy gồm có hai phần:

- Một vài trường hợp cơ bản nhỏ nhất có thể được giải quyết mà không cần đệ quy.
- Một phương pháp chung có thể giảm một trường hợp thành một hoặc nhiều trường hợp nhỏ hơn, và nhờ đó việc giảm nhỏ vấn đề có thể tiến triển cho đến kết quả cuối cùng là các trường hợp cơ bản.

C++, cũng như các ngôn ngữ máy tính hiện đại khác, cho phép đệ quy dễ dàng. Việc tính giai thừa trong C++ trở thành một hàm sau đây.

```
int factorial(int n)
/*
pre:  n là một số không âm.
post: trả về trị của n giai thừa.
*/
{
   if (n == 0)
      return 1;
   else
      return n * factorial(n - 1);
}
```

Như chúng ta thấy, định nghĩa đệ quy và lời giải đệ quy của một bài toán đều có thể rất ngắn gọn và đẹp đẽ. Tuy nhiên việc tính toán chi tiết có thể đòi hỏi phải giữ lại rất nhiều phép tính từng phần trước khi có được kết quả đầy đủ.

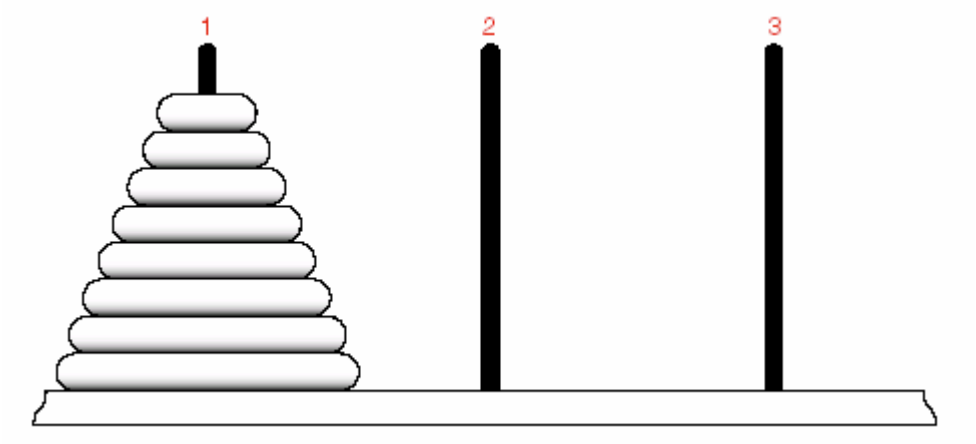
Máy tính có thể dễ dàng nhớ các tính toán từng phần bằng một ngăn xếp. Con người thì khó làm được như vậy, con người khó có thể nhớ một dãy dài các kết quả tính toán từng phần để rồi sau đó quay lại hoàn tất chúng. Do đó, khi sử dụng đệ quy, cách chúng ta suy nghĩ có khác với các cách lập trình khác. Chúng ta phải xem xét vấn đề bằng một cách nhìn tổng thể và dành những việc tính toán chi tiết lại cho máy tính.

Chúng ta phải đặc tả trong giải thuật của chúng ta một cách chính xác các bước tổng quát của việc giảm một bài toán lớn thành nhiều trường hợp nhỏ hơn; chúng ta phải xác định điều kiện dừng (các trường hợp nhỏ nhất) và cách giải của chúng. Ngoại trừ một số ít ví dụ nhỏ và đơn giản, chúng ta không nên cố gắng hiểu giải thuật đệ quy bằng cách biến đổi từ bài toán ban đầu cho đến tận bước kết thúc, hoặc lần theo vết của các công việc mà máy tính sẽ làm. Làm như thế, chúng ta sẽ nhanh chóng lẫn lộn bởi các công việc bị trì hoãn lại và chúng ta sẽ bị mất phương hướng.

6.1.4. Chia để trị: Bài toán Tháp Hà Nội

6.1.4.1. Bài toán

Vào thế kỷ thứ 19 ở châu Âu xuất hiện một trò chơi được gọi là Tháp Hà Nội. Người ta kể rằng trò chơi này biểu diễn một nhiệm vụ ở một ngôi đền của Ấn Độ giáo. Vào cái ngày mà thế giới mới được tạo nên, các vị linh mục được giao cho 3 cái tháp bằng kim cương, tại tháp thứ nhất có để 64 cái đĩa bằng vàng. Các linh mục này phải di chuyển các đĩa từ tháp thứ nhất sang tháp thứ ba sao cho mỗi lần chỉ di chuyển 1 đĩa và không có đĩa lớn nằm trên đĩa nhỏ. Người ta bảo rằng khi công việc hoàn tất thì đến ngày tận thế.



Hình 6.3- Bài toán tháp Hà nội

Nhiệm vụ của chúng ta là viết một chương trình in ra các bước di chuyển các đĩa giúp cho các nhà linh mục, chúng ta gọi dòng lệnh sau

```
move(64, 1, 3, 2)
```

có nghĩa là: chuyển 64 đĩa từ tháp thứ nhất sang tháp thứ ba, sử dụng tháp thứ hai làm nơi để tạm.

6.1.4.2. Lời giải

Ý tưởng để đến với lời giải ở đây là, sự tập trung chú ý của chúng ta không phải là vào bước đầu tiên di chuyển cái đĩa trên cùng, mà là vào bước khó nhất: di chuyển cái đĩa dưới cùng. Đĩa lớn nhất dưới cùng này sẽ phải có vị trí ở dưới cùng tại tháp thứ ba theo yêu cầu bài toán. Không có cách nào khác để chạm được đến đĩa cuối cùng trước khi 63 đĩa nằm trên đã được chuyển đi. Đồng thời 63 đĩa này phải được đặt tại tháp thứ hai để tháp thứ ba trống.

Chúng ta đã có được một bước nhỏ để tiến đến lời giải, đây là một bước rất nhỏ vì chúng ta còn phải tìm cách di chuyển 63 đĩa. Tuy nhiên đây lại là một bước rất quan trọng, vì việc di chuyển 63 đĩa đã có cùng bản chất với bài toán ban đầu, vì không có lý do gì ngăn cản việc chúng ta di chuyển 63 đĩa này theo cùng một cách tương tự.

```
move(63,1,2,3);// Chuyển 63 đĩa từ tháp 1 sang tháp 2 (tháp 3 dùng làm nơi để tạm).
cout << "Chuyển đĩa thứ 64 từ tháp 1 sang tháp 3."  << endl;
move(63,2,3,1);// Chuyển 63 đĩa từ tháp 2 sang tháp 3 (tháp 1 dùng làm nơi để tạm).
```

Cách suy nghĩ như trên chính là ý tưởng của đệ quy. Chúng ta đã mô tả các bước chủ chốt được thực hiện như thế nào, và các công việc còn lại của bài toán cũng sẽ được thực hiện một cách tương tự. Đây cũng là ý tưởng của việc chia để trị: để giải quyết một bài toán, chúng ta chia công việc ra thành nhiều phần nhỏ hơn, mỗi phần lại được chia nhỏ hơn nữa, cho đến khi việc giải chúng trở nên dễ dàng hơn bài toán ban đầu rất nhiều.

6.1.4.3. Tinh chế

Để viết được giải thuật, chúng ta cần biết tại mỗi bước, tháp nào được dùng để chứa tạm các đĩa. Chúng ta có đặc tả sau đây cho hàm:

void move(int count, int start, int finish, int temp);
pre: Có ít nhất là **count** đĩa tại tháp **start**. Đĩa trên cùng của tháp **temp** và tháp **finish** lớn hơn bất kỳ đĩa nào trong **count** đĩa trên cùng tại tháp **start**.
post: **count** đĩa trên cùng tại tháp **start** đã được chuyển sang tháp **finish**; tháp **temp** được dùng làm nơi để tạm sẽ trở lại trạng thái ban đầu.

Giả sử rằng bài toán của chúng ta sẽ dừng sau một số bước hữu hạn (mặc dầu đó có thể là ngày tận thế!), và như vậy phải có cách nào đó để việc đệ quy dừng lại. Một điều kiện dừng hiển nhiên là khi không còn đĩa cần di chuyển nữa. Chúng ta có thể viết chương trình sau:

```
const int disks = 64;  // Cần sửa hằng số này thật nhỏ để chạy thử chương trình.

void move(int count, int start, int finish, int temp);
/*
pre: Không có.
post: Chương trình mô phỏng bài toán Tháp Hà Nội kết thúc.
*/
main()
{
   move(disks, 1, 3, 2);
}
```

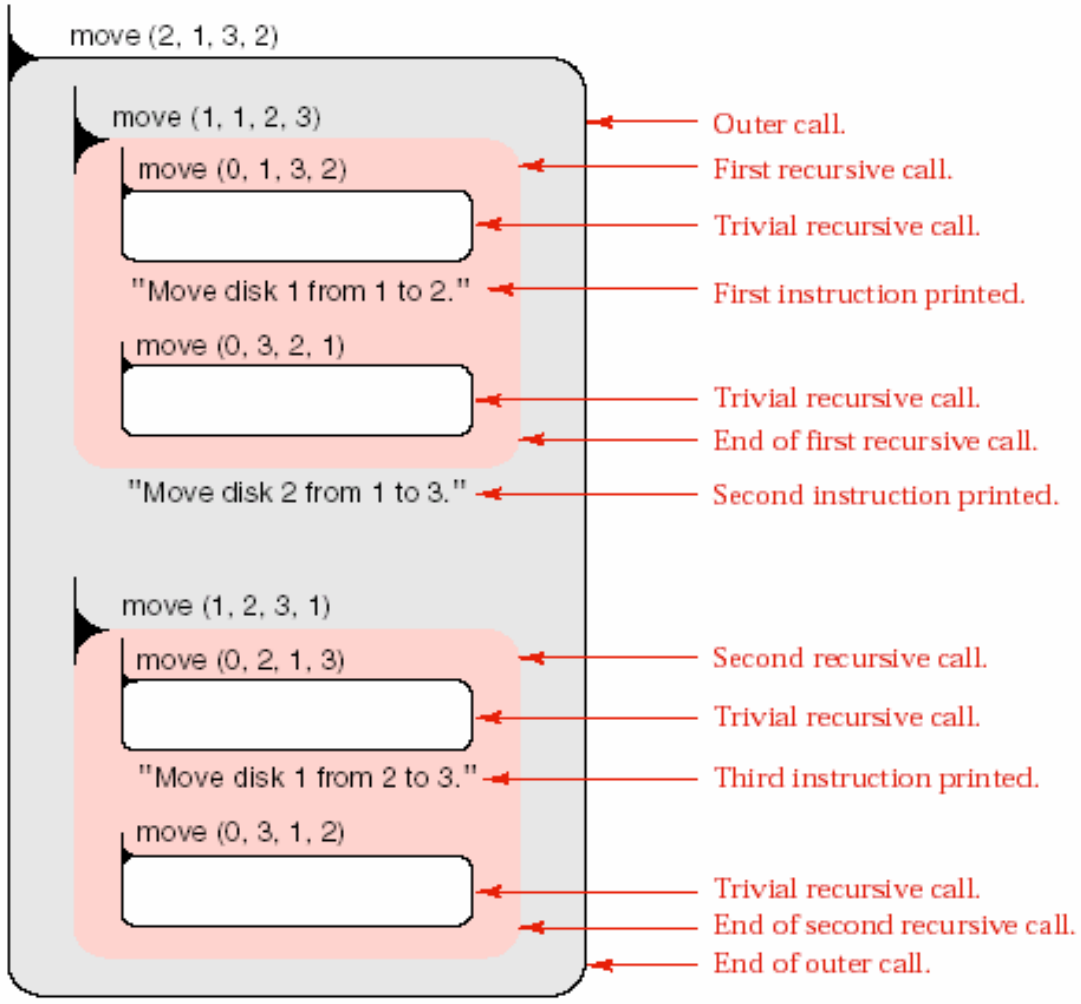
Hàm đệ quy như sau:

```
void move(int count, int start, int finish, int temp)
{
   if (count > 0) {
      move(count - 1, start, temp, finish);
      cout << "Move disk " << count << " from " << start
           << " to " << finish << "." << endl;
      move(count - 1, temp, finish, start);
   }
}
```

6.1.4.4. Theo vết của chương trình

Công cụ hữu ích của chúng ta trong việc tìm hiểu một hàm đệ quy là hình ảnh thể hiện các bước thực hiện của nó trên một ví dụ thật nhỏ. Các lần gọi hàm trong hình 6.4 là cho trường hợp số đĩa bằng 2. Mỗi khối trong sơ đồ biểu diễn những gì diễn ra trong một lần gọi hàm. Lần gọi ngoài cùng `move(2,1,3,2)` (do chương trình chính gọi) có ba dòng lệnh sau:

```
move(1,1,2,3);// Chuyển 1 đĩa từ tháp 1 sang tháp 2 (tháp 3 dùng làm nơi để tạm).
cout << " Chuyển đĩa thứ 2 từ tháp 1 sang tháp 3."  << endl;
move(1,2,3,1);// Chuyển 1 đĩa từ tháp 2 sang tháp 3 (tháp 1 dùng làm nơi để tạm).
```



Hình 6.4- Theo vết của chương trình Tháp Hà Nội với số đĩa là 2.

Dòng lệnh thứ nhất và dòng lệnh thứ ba gọi đệ quy. Dòng lệnh **`move(1,1,2,3)`** bắt đầu gọi hàm `move` thực hiện trở lại dòng lệnh đầu tiên, nhưng với các thông số mới. Dòng lệnh này sẽ thực hiện đúng ba lệnh sau:

`move(0,1,3,2);//` Chuyển 0 đĩa (gọi đệ quy lần nữa, biểu diễn bởi khối nhỏ bên / / trong).

cout << "**Chuyển đĩa 1 từ tháp 1 sang tháp 2**" << endl;

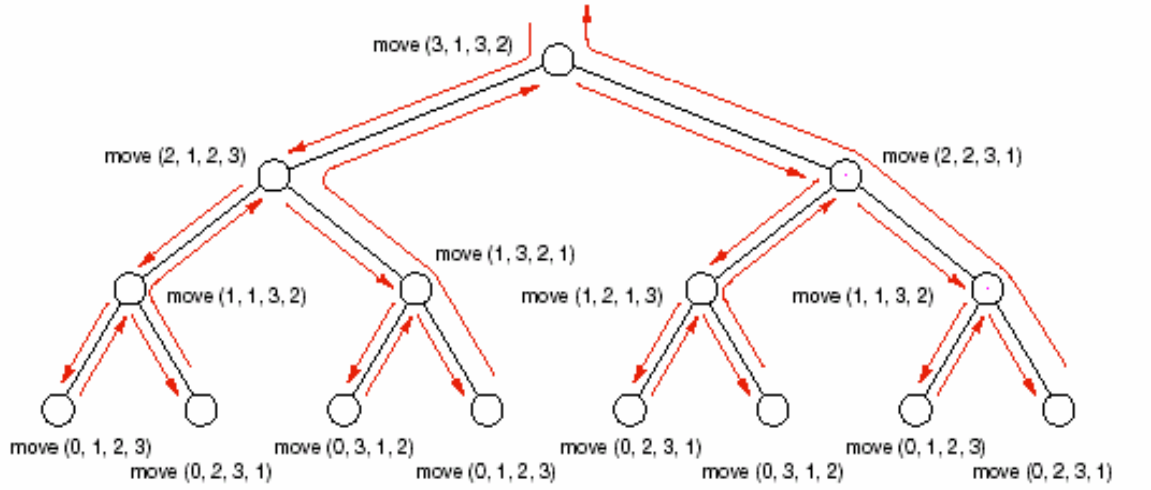
`move(0,3,2,1);//` Chuyển 0 đĩa (gọi đệ quy lần nữa, biểu diễn bởi khối nhỏ bên / / trong).

Sau khi khối biểu diễn lần gọi đệ quy này kết thúc, dòng lệnh hiển thị "**Chuyển đĩa thứ 2 từ tháp 1 sang tháp 3**" thực hiện. Sau đó là khối biểu diễn lần gọi đệ quy **`move(1,2,3,1)`**.

Chúng ta thấy rằng hai lần gọi đệ quy bên trong khối **`move(1,1,2,3)`** có số đĩa là `0` nên không phải thực hiện điều gì, hình biểu diễn là một khối rỗng. Giữa hai lần này là hiểu thị "**Chuyển đĩa 1 từ tháp 1 sang tháp 2.**" Tương tự cho các dòng lệnh bên trong **`move(1,2,3,1)`**, chúng ta hiểu được cách mà đệ quy hiện thực.

Chúng ta sẽ xem xét thêm một công cụ khác có tính hiển thị cao hơn trong việc biểu diễn sự đệ quy bằng cách lần theo vết của chương trình vừa rồi.

6.1.4.5. Phân tích



Hình 6.5- Cây đệ quy cho trường hợp 3 đĩa

Hình 6.5 là cây đệ quy cho bài toán Tháp Hà Nội với 3 đĩa.

Lưu ý rằng chương trình của chúng ta cho bài toán Tháp Hà Nội không chỉ sinh ra một lời giải đầy đủ cho bài toán mà còn sinh ra một lời giải tốt nhất có thể có, và đây cũng là lời giải duy nhất được tìm thấy trừ khi chúng ta chấp nhận lời giải với một dãy dài lê thê các bước dư thừa và bất lợi như sau:

Chuyển đĩa 1 từ tháp 1 sang tháp 2.
Chuyển đĩa 1 từ tháp 2 sang tháp 3.
Chuyển đĩa 1 từ tháp 3 sang tháp 1. . . .

Để chứng minh tính duy nhất của một lời giải không thể giản lược hơn được nữa, chúng ta chú ý rằng, tại mỗi bước, nhiệm vụ cần làm được tổng kết lại là cần di chuyển một số đĩa nhất định nào đó từ một tháp này sang một tháp khác. Không có cách nào khác ngoài cách là trước hết **phải di chuyển toàn bộ số đĩa bên trên**, trừ đĩa cuối cùng nằm dưới, sau đó có thể thực hiện một số bước dư thừa nào đó, tiếp theo là **di chuyển chính đĩa cuối cùng**, rồi lại có thể thực hiện một số bước dư thừa nào đó, để cuối cùng là **di chuyển toàn bộ số đĩa cũ về lại trên đĩa dưới cùng này**. Như vậy, nếu loại đi tất cả các việc làm dư thừa thì những việc còn lại chính là cốt lõi của giải thuật đệ quy của chúng ta.

Tiếp theo, chúng ta sẽ tính xem đệ quy được gọi liên tiếp bao nhiêu lần trước khi có sự quay về. Lần đầu đệ quy có `count=64`, mỗi lần đệ quy `count` được giảm đi `1`. Vậy nếu chúng ta gọi đệ quy với `count = 0`, lần đệ quy này không thực hiện gì, chúng ta có tổng độ sâu của đệ quy là `64`. Điều này có nghĩa rằng, nếu chúng ta vẽ cây đệ quy cho chương trình, thì cây sẽ có `64` mức không kể mức của

các mức lá. Ngoại trừ các nút lá, các nút khác đều gọi đệ quy hai lần trong mỗi nút, như vậy tổng số nút tại mỗi mức chính xác bằng hai lần tổng số nút ở mức cao hơn.

Từ cách suy nghĩ trên về cây đệ quy (ngay cả khi cây quá lớn không thể vẽ được), chúng ta có thể dễ dàng tính ra số lần di chuyển cần làm (mỗi lần di chuyển một đĩa) để di chuyển hết 64 đĩa theo yêu cầu bài toán. Mỗi nút trong cây sẽ in một lời hướng dẫn tương ứng một lần chuyển một đĩa, trừ các nút lá. Tổng số nút gốc và nút trung gian là:

$$1 + 2 + 4 + \ldots + 2^{63} = 2^0 + 2^1 + 2^2 + \ldots + 2^{63} = 2^{64} - 1.$$

nên số lần di chuyển đĩa cần thực hiện tất cả là $2^{64} - 1$. Chúng ta có thể ước chừng con số này lớn như thế nào bằng cách so sánh với

$$10^3 = 1000 \approx 1024 = 2^{10},$$

ta có tổng số lần di chuyển đĩa bằng $2^{64} = 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1.6 \times 10^{19}$

Mỗi năm có khoảng 3.2×10^7 giây. Giả sử mỗi lần di chuyển một đĩa được thực hiện mất 1 giây, thì toàn bộ công việc của các linh mục sẽ phải thực hiện mất 5×10^{11} năm. Các nhà thiên văn học ước đoán tuổi thọ của vũ trụ sẽ nhỏ hơn 20 tỉ năm, như vậy, theo truyền thuyết của bài toán này thì thế giới còn kéo dài hơn cả việc tính toán đó đến 25 lần!

Không có một máy tính nào có thể chạy được chương trình Tháp Hà Nội, do không đủ thời gian, nhưng rõ ràng không phải là do vấn đề không gian. Không gian ở đây chỉ đòi hỏi 64 lần gọi đệ quy.

6.2. Các nguyên tắc của đệ quy

6.2.1. Thiết kế giải thuật đệ quy

Đệ quy là một công cụ cho phép người lập trình tập trung vào bước chính yếu của giải thuật mà không phải lo lắng tại thời điểm khởi đầu về cách kết nối bước chính yếu này với các bước khác. Khi cần giải quyết một vấn đề, bước tiếp cận đầu tiên nên làm thường là xem xét một vài ví dụ đơn giản, và chỉ sau khi đã hiểu được chúng một cách kỹ lưỡng, chúng ta mới thử cố gắng xây dựng một phương pháp tổng quát hơn. Một vài điểm quan trọng trong việc thiết kế một giải thuật đệ quy được liệt kê sau đây:

Tìm bước chính yếu. Hãy bắt đầu bằng câu hỏi "*Bài toán này có thể <u>được chia nhỏ</u> như thế nào?*" hoặc "*Bước chính yếu <u>trong giai đoạn giữa</u> sẽ được thực hiện*

như thế nào?". Nên đảm bảo rằng câu trả lời của bạn đơn giản nhưng có tính tổng quát. Không nên đi từ điểm khởi đầu hay điểm kết thúc của bài toán lớn, hoặc sa vào quá nhiều trường hợp đặc biệt (do chúng chỉ phù hợp với các bài toán nhỏ). Khi đã có được một bước nhỏ và đơn giản để hướng tới lời giải, hãy tự hỏi rằng những khúc mắc còn lại của bài toán có thể được giải quyết bằng cách tương tự hay không, để sửa lại phương pháp của bạn cho tổng quát hơn, nếu cần thiết.
Ngoại trừ những định nghĩa toán học thể hiện sự đệ quy quá rõ ràng, một điều thú vị mà chúng ta sẽ lần lượt gặp trong những chương sau là, khi những bài toán cần được giải quyết trên những cấu trúc dữ liệu mà định nghĩa mang tính chất đệ quy như danh sách, chuỗi ký tự biểu diễu biểu thức số học, cây, hay đồ thị,... thì giải pháp hướng tới một giải thuật đệ quy là rất dễ nhìn thấy.

Tìm điều kiện dừng. Điều kiện dừng chỉ ra rằng bài toán hoặc một phần nào đó của bài toán đã được giải quyết. Điều kiện dừng thường là trường hợp nhỏ, đặc biệt, có thể được giải quyết một cách dễ dàng không cần đệ quy.

Phác thảo giải thuật. Kết hợp điều kiện dừng với bước chính yếu của bài toán, sử dụng lệnh **if** để chọn lựa giữa chúng. Đến đây thì chúng ta có thể viết hàm đệ quy, trong đó mô tả cách mà bước chính yếu được tiến hành cho đến khi gặp được điều kiện dừng. Mỗi lần gọi đệ quy hoặc là phải giải quyết một phần của bài toán khi gặp một trong các điều kiện dừng, hoặc là phải giảm kích thước bài toán hướng dần đến điều kiện dừng.

Kiểm tra sự kết thúc. Kế tiếp, và cũng là điều tối quan trọng, là phải chắc chắn việc gọi dệ quy sẽ không bị lặp vô tận. Bắt đầu từ một trường hợp chung, qua một số bước hữu hạn, chúng ta cần kiểm tra liệu điều kiện dừng có khả năng xảy ra để quá trình đệ quy kết thúc hay không. Trong bất kỳ một giải thuật nào, khi một lần gọi hàm không phải làm gì, nó thường quay về một cách êm thấm. Đối với giải thuật đệ quy, điều này rất thường xảy ra, do việc gọi hàm mà không phải làm gì thường là một điều kiện dừng. Do đó, cần lưu ý rằng việc gọi hàm mà không làm gì thường không phải là một lỗi trong trường hợp của hàm đệ quy.

Kiểm tra lại mọi trường hợp đặc biệt
Cuối cùng chúng ta cũng cần bảo đảm rằng giải thuật của chúng ta luôn đáp ứng mọi trường hợp đặc biệt.

Vẽ cây đệ quy. Công cụ chính để phân tích các giải thuật đệ quy là cây đệ quy. Như chúng ta đã thấy trong bài toán Tháp Hà Nội, chiều cao của cây đệ quy liên quan mật thiết đến tổng dung lượng bộ nhớ mà chương trình cần đến, và kích thước tổng cộng của cây phản ánh số lần thực hiện bước chính yếu và cũng là tổng thời gian chạy chương trình. Thông thường chúng ta nên vẽ cây đệ quy cho

một hoặc hai trường hợp đơn giản của bài toán của chúng ta vì nó sẽ chỉ dẫn cho chúng ta nhiều điều.

6.2.2. Cách thực hiện của đệ quy

Câu hỏi về cách hiện thực của một chương trình đệ quy trong máy tính cần được tách rời khỏi câu hỏi về sử dụng đệ quy để thiết kế giải thuật.

Trong giai đoạn thiết kế, chúng ta nên sử dụng mọi phương pháp giải quyết vấn đề mà chúng tỏ ra thích hợp với bài toán, đệ quy là một trong các công cụ hiệu quả và linh hoạt này.

Trong giai đoạn hiện thực, chúng ta cần tìm xem phương pháp nào trong số các phương pháp sẽ là tốt nhất so với từng tình huống.

Có ít nhất hai cách để hiện thực đệ quy trong hệ thống máy tính. Quan điểm chính của chúng ta khi xem xét hai cách hiện thực khác nhau dưới đây là, cho dù có sự hạn chế về không gian và thời gian, chúng cũng nên được tách riêng ra khỏi quá trình thiết kế giải thuật. Các loại thiết bị tính toán khác nhau trong tương lai có thể dẫn đến những khả năng và những hạn chế khác nhau. Chúng ta sẽ tìm hiểu hai cách hiện thực đa xử lý và đơn xử lý của đệ quy dưới đây.

6.2.2.1. Hiện thực đa xử lý: sự đồng thời

Có lẽ rằng cách suy nghĩ tự nhiên về quá trình hiện thực của đệ quy là các hàm không chiếm những phần riêng trong cùng một máy tính, mà chúng sẽ được thực hiện trên những máy khác nhau. Bằng cách này, khi một hàm cần gọi một hàm khác, nó khởi động chiếc máy tương ứng, và khi máy này kết thúc công việc, nó sẽ trả về chiếc máy ban đầu kết quả tính được để chiếc máy ban đầu có thể tiếp tục công việc. Nếu một hàm gọi đệ quy chính nó hai lần, đơn giản nó chỉ cần khởi động hai chiếc máy khác để thực hiện cũng những dòng lệnh y như những dòng lệnh mà nó đang thực hiện. Khi hai máy này hoàn tất công việc chúng trả kết quả về cho máy gọi chúng. Nếu chúng cần gọi đệ quy, dĩ nhiên chúng cũng khởi động những chiếc máy khác nữa.

Thông thường bộ xử lý trung ương là thành phần đắt nhất trong hệ thống máy tính, nên bất kỳ một ý nghĩ nào về một hệ thống có nhiều hơn một bộ xử lý cũng cần phải xem xét đến sự lãng phí. Nhưng rất có thể trong tương lai chúng ta sẽ thấy những hệ thống máy tính lớn chứa hàng trăm, nếu không là hàng ngàn, các bộ vi xử lý tương tự trong các thành phần của nó. Khi đó thì việc thực hiện đệ quy bằng nhiều bộ xử lý song song sẽ trở nên bình thường.

Với đa xử lý, những người lập trình sẽ không còn xem xét các giải thuật chỉ như một chuỗi tuyến tính các hành động, thay vào đó, cần phải nhận ra một số phần của giải thuật có thể thực hiện song song. Cách xử lý này còn được gọi là xử

lý đồng thời (*concurrent*). Việc nghiên cứu về xử lý đồng thời và các phương pháp kết nối giữa chúng hiện tại là một đề tài nghiên cứu trong khoa học máy tính, một điều chắc chắn là nó sẽ cải tiến cách mà các giải thuật sẽ được mô tả và hiện thực trong nhiều năm tới.

6.2.2.2. Hiện thực đơn xử lý: vấn đề vùng nhớ

Để xem xét làm cách nào mà đệ quy có thể được thực hiện trong một hệ thống chỉ có một bộ xử lý, chúng ta nhớ lại cơ cấu ngăn xếp của các lần gọi hàm đã được giới thiệu ở đầu chương này. Một hàm khi được gọi cần phải **có một vùng nhớ riêng** để chứa các biến cục bộ và các tham trị của nó, kể cả các trị trong các thanh ghi và địa chỉ quay về khi nó chuẩn bị gọi một hàm khác. Sau khi hàm kết thúc, nó sẽ không còn cần đến bất cứ thứ gì trong vùng nhớ dành riêng cho nó nữa. **Thực sự là không có sự khác nhau giữa việc gọi một hàm đệ quy và việc gọi một hàm không đệ quy**. Khi một hàm chưa kết thúc, vùng nhớ của nó là bất khả xâm phạm. Một lần gọi hàm đệ quy cũng là một lần gọi hàm riêng biệt. Chúng ta cần chú ý rằng hai lần gọi đệ quy là hoàn toàn khác nhau, để **chúng ta không trộn lẫn vùng nhớ của chúng khi chúng chưa kết thúc**. Đối với những hàm đệ quy, những thông tin lưu trữ dành cho lần gọi ngoài cần được giữ cho đến khi nó kết thúc, như vậy một lần gọi bên trong phải sử dụng một vùng khác làm vùng nhớ của riêng nó.

Đối với một hàm không đệ quy, vùng nhớ có thể là một vùng cố định và được dành cho lâu dài, do chúng ta biết rằng một lần gọi hàm sẽ được trả về trước khi hàm có thể lại được gọi lần nữa, và sau khi lần gọi trước được trả về, các thông tin trong vùng nhớ của nó không còn cần thiết nữa. Vùng nhớ lâu dài được dành sẵn cho các hàm không đệ quy có thể gây lãng phí rất lớn, do những khi hàm không được yêu cầu thực hiện, vùng nhớ đó không thể được sử dụng vào mục đích khác. Đó cũng là cách quản lý vùng nhớ dành cho các hàm của các phiên bản cũ của các ngôn ngữ như FORTRAN và COBOL, và chính điều này cũng là lý do mà các ngôn ngữ này không cho phép đệ quy.

6.2.2.3. Nhu cầu về thời gian và không gian của một quá trình đệ quy

Chúng ta hãy xem lại cây biểu diễn các lần gọi hàm: trong quá trình duyệt cây, các nút được thêm vào hay lấy đi đúng theo kiểu của ngăn xếp. Quá trình này được minh họa trong hình 6.1.

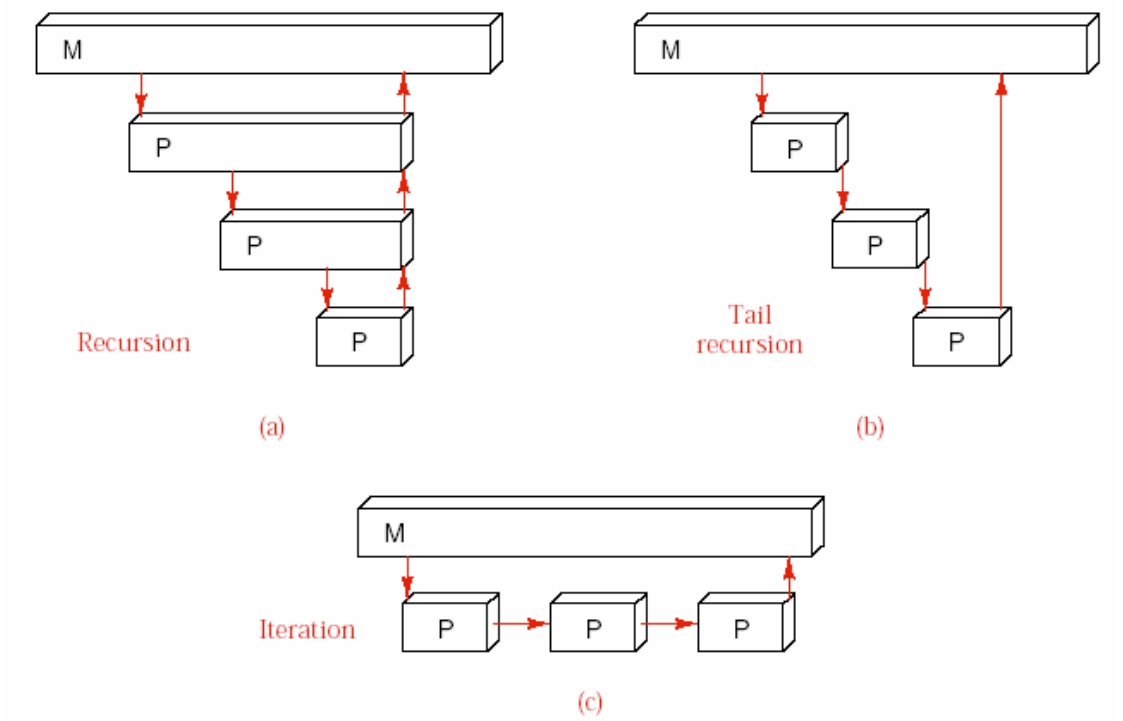
Từ hình này, chúng ta có thể kết luận ngay rằng tổng dung lượng vùng nhớ cần để hiện thực đệ quy tỉ lệ thuận với chiều cao của cây đệ quy. Những người lập trình không tìm hiểu kỹ về đệ quy thỉnh thoảng vẫn nhầm lẫn rằng không gian cần phải có liên quan đến tổng số nút trong cây. Thời gian chạy chương trình liên quan đến số lần gọi hàm, đó là tổng số nút trong cây; nhưng dung lượng vùng nhớ tại một thời điểm chỉ là tổng các vùng nhớ dành cho các nút nằm trên đường đi từ nút tương ứng với hàm đang thực thi ngược về gốc của cây. Không gian cần

thiết được phản ánh bởi chiều cao của cây. Một cây đệ quy có nhiều nút nhưng không cao thể hiện một quá trình đệ quy mà nó thực hiện được rất nhiều công việc trên một vùng nhớ không lớn.

6.2.3. Đệ quy đuôi

Chúng ta hãy xét đến trường hợp hành động cuối cùng trong một hàm là việc gọi đệ quy chính nó. Hãy xem xét ngăn xếp dành cho quá trình đệ quy, như chúng ta thấy, các thông tin cần để khôi phục lại trạng thái cho lần đệ quy ngoài sẽ được lưu lại ngay trước khi lần đệ quy trong được gọi. Tuy nhiên khi lần đệ quy trong thực hiện xong thì lần đệ quy ngoài cũng không còn việc gì phải làm nữa, do việc gọi đệ quy là hành động cuối cùng của hàm nên đây cũng là lúc mà hàm đệ quy ngoài kết thúc. Và như vậy việc lưu lại những thông tin dùng để khôi phục trạng thái cũ của lần đệ quy ngoài trở nên hoàn toàn vô ích. Mọi việc cần làm ở đây chỉ là gán các trị cần thiết cho các biến và quay ngay trở về đầu hàm, các biến được gán trị y như là chính hàm đệ quy bên trong nhận được qua danh sách thông số vậy. Chúng ta tổng kết nguyên tắc này như sau:

*Nếu dòng lệnh **sẽ được chạy cuối cùng** trong một hàm là gọi đệ quy chính nó, thì việc gọi đệ quy này có thể được loại bỏ bằng cách gán lại các thông số gọi theo các giá trị như là đệ quy vẫn được gọi, và sau đó lập lại toàn bộ hàm.*



Hình 6.6 – Đệ quy đuôi

Quá trình thay đổi này được minh họa trong hình 6.6. Hình 6.6a thể hiện vùng nhớ được sử dụng bởi chương trình gọi M và một số bản sao của hàm đệ quy P, mỗi hàm một vùng nhớ riêng. Các mũi tên xuống thể hiện sự gọi hàm. Mỗi sự gọi từ P đến chính nó cũng là hành động cuối trong hàm, việc duy trì vùng nhớ cho hàm trong khi chờ đợi sự trả về từ hàm được gọi là không cần thiết. Cách biến đổi như trên sẽ giảm kích thước vùng nhớ đáng kể (hình 6.6b). Cuối cùng, hình 6.6c biểu diễn các lần gọi hàm P như một dạng lặp lại trong cùng một mức của sơ đồ.

Trường hợp đặc biệt chúng ta vừa nêu trên là vô cùng quan trọng vì nó cũng thường xuyên xảy ra. Chúng ta gọi đó là trường hợp đệ quy đuôi (*tail recursion*). Chúng ta nên cẩn thận rằng trong đệ quy đuôi, việc gọi đệ quy là **hành động cuối trong hàm**, chứ không phải là **dòng lệnh cuối được viết trong hàm**. Trong chương trình có khi chúng ta thấy đệ quy đuôi xuất hiện trong lệnh **`switch`** hoặc lệnh **`if`** trong hàm mà sau đó còn có thể có nhiều dòng lệnh khác nữa.

Đối với phần lớn các trình biên dịch, chỉ có một sự khác nhau nhỏ giữa thời gian chạy trong hai trường hợp: trường hợp đệ quy đuôi và trường hợp nó đã được thay thế bằng vòng lệnh lặp. Tuy nhiên, nếu không gian được xem là quan trọng, thì việc loại đệ quy đuôi là rất cần thiết. Đệ quy đuôi thường được thay bởi vòng lặp **`while`** hoặc **`do while`**.

Trong giải thuật chia để trị của bài toán Tháp Hà Nội, lần gọi đệ quy trên không phải là đệ quy đuôi, lần gọi sau đó mới là đệ quy đuôi. Hàm sau đây đã được loại đệ quy đuôi:

```
void move(int count, int start, int finish, int temp)
/*    move: phiên bản lặp.
pre:  count là số đĩa cần di chuyển.
post: count đĩa đã được chuyển từ start sang finish dùng temp làm nơi chứa tạm.
*/
{
   int swap;
   while (count > 0) {  //  Thay lệnh if trong đệ quy bằng vòng lặp.
      move(count - 1, start, temp, finish);// lần gọi đệ quy đầu không phải
                                           // đệ quy đuôi.
      cout << "Move disk " << count << " from " << start
           << " to " << finish << "." << endl;
      count--; // Thay đổi các thông số cho tương đương với việc gọi đệ quy đuôi.
      swap = start;
      start = temp;
      temp = swap;
   }
}
```

Thật ra chúng ta có thể nghĩ ngay đến phương án này khi mới bắt đầu giải bài toán. Nhưng chúng ta đã xem xét nó từ một cách nhìn khác, bây giờ chúng ta sẽ lý giải lại các dòng lệnh trên một cách tự nhiên hơn. Chúng ta sẽ thấy rằng hai tháp **start** và **temp** không có gì khác nhau, do chúng cùng được sử dụng để làm nơi chứa tạm trong khi chúng ta chuyển dần các đĩa về tháp **finish**.
Để chuyển một số đĩa từ **start** về **finish**, chúng ta chuyển tất cả đĩa trong số đó, trừ cái cuối cùng, sang tháp còn lại là **temp**. Sau đó chuyển đĩa cuối sang **finish.**

Tiếp tục lặp lại việc vừa rồi, chúng ta lại cần chuyển tất cả các đĩa từ **temp,** trừ cái cuối cùng, sang tháp còn lại là **start**, để có thể chuyển đĩa cuối cùng sang **finish**. Lần thực hiện thứ hai này sử dụng lại các dòng lệnh trong chương trình bằng cách hoán đổi **start** với **temp**. Cứ như thế, sau mỗi lần hoán đổi **start** với **temp**, công việc được lặp lại y như nhau, kết quả của mỗi lần lặp là chúng ta có được thêm một đĩa mới trên **finish**.

6.2.4. Phân tích một số trường hợp nên và không nên dùng đệ quy

6.2.4.1. Giai thừa

Chúng ta hãy xem xét hai hàm tính giai thừa sau đây. Đây là hàm đệ quy:

```
int factorial(int n)
/*    factorial: phiên bản đệ quy.
pre:  n là một số không âm.
post: trả về trị của n giai thừa.
*/
{
   if (n == 0) return 1;
   else        return n * factorial(n - 1);
}
```

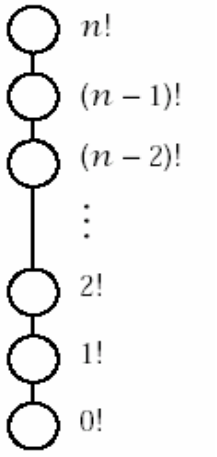
Và đây là hàm không đệ quy:

```
int factorial(int n)
/*    factorial: phiên bản không đệ quy.
pre:  n là một số không âm.
post: trả về trị của n giai thừa.
*/
{
   int count, product = 1;
   for (count = 1; count <= n; count++)
      product *= count;
   return product;
}
```

Chương trình nào trên đây sử dụng ít vùng nhớ hơn? Với cái nhìn đầu tiên, dường như chương trình đệ quy chiếm ít vùng nhớ hơn, do nó không có biến cục bộ, còn chương trình không đệ quy có đến hai biến cục bộ. Tuy nhiên, chương trình đệ quy cần một ngăn xếp để chứa n con số

n, n-1, n-2, ..., 2, 1

là những thông số để gọi đệ quy (hình 6.7), và theo cách đệ quy của mình, nó cũng phải nhân các số lại với nhau theo một thứ tự không khác gì so với chương trình không đệ quy. Tiến trình thực hiện của chương trình đệ quy cho n = 5 như sau:




```
factorial(5) =5*factorial(4)
 =5*(4*factorial(3))
 =5*(4*(3*factorial(2)))
 =5*(4*(3*(2*factorial(1))))
 =5*(4*(3*(2*(1*factorial(0)))))
 =5*(4*(3*(2*(1*1))))
 =5*(4*(3*(2*1)))
 =5*(4*(3*2))
 =5*(4*6)
 =5*24
 =120
```

Hình 6.7 –
Cây đệ quy tính giai thừa

Như vậy chương trình đệ quy chiếm nhiều vùng nhớ hơn chương trình không đệ quy, đồng thời nó cũng chiếm nhiều thời gian hơn do chúng vừa phải cất và lấy các trị từ ngăn xếp vừa phải thực hiện việc tính toán.

6.2.4.2. Các số *Fibonacci*

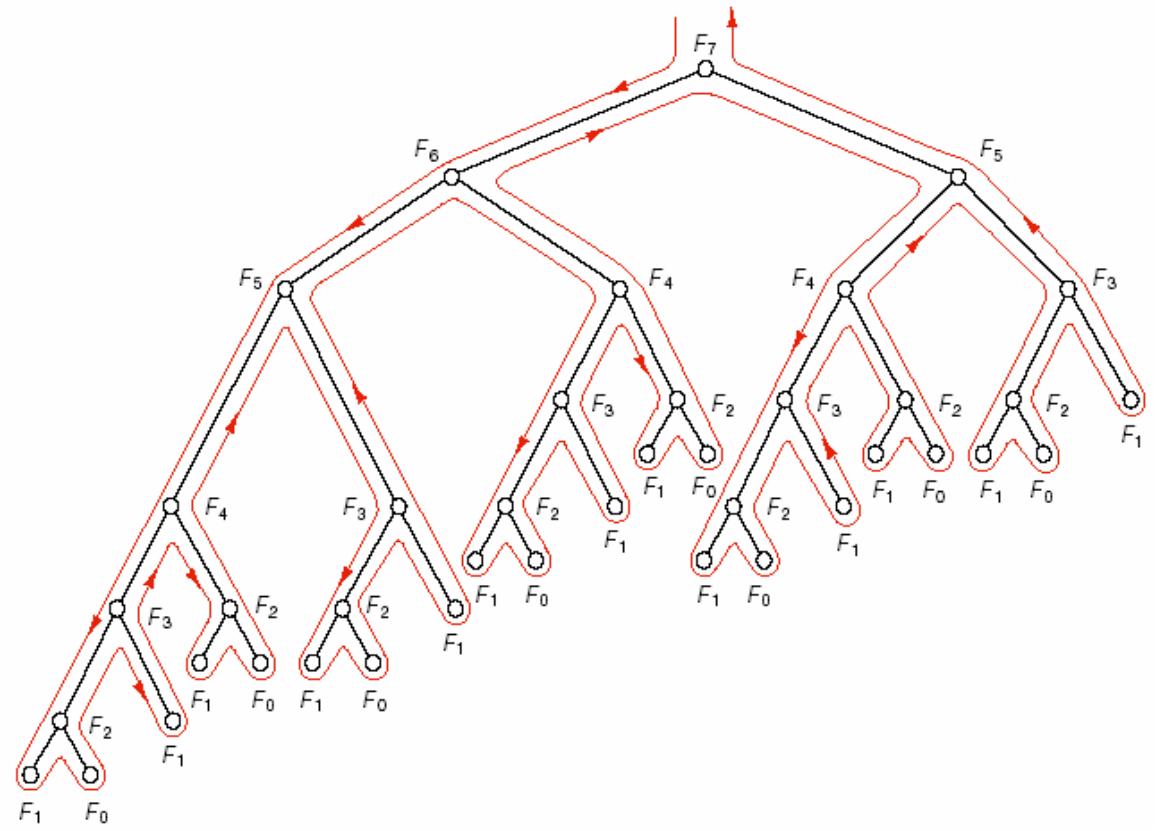
Một ví dụ còn lãng phí hơn chương trình tính giai thừa là việc tính các số *Fibonacci*. Các số này được định nghĩa như sau:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ nếu } n \geq 2.$$

Chương trình đệ quy tính các số *Fibonacci* rất giống với định nghĩa:

```
int fibonacci(int n)
/*    fibonacci: phiên bản đệ quy.
pre:  n là một số không âm.
post: trả về số Fibonacci thứ n.
*/
{
   if (n <= 0)  return 0;
   else if (n == 1)  return 1;
   else              return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Thực tế, chương trình này trông rất đẹp mắt, do nó có dạng chia để trị: kết quả có được bằng cách tính toán hai trường hợp nhỏ hơn. Tuy nhiên, chúng ta sẽ thấy rằng đây hoàn toàn không phải là trường hợp "chia để trị", mà là "chia làm cho phức tạp thêm".



Hình 6.8- Cây đệ quy tính F_7.

Để xem xét giải thuật này, chúng ta thử tính F_7, minh họa trong hình 6.8. Trước hết hàm cần tính F_6 và F_5. Để có F_6, phải có F_5 và F_4, và cứ như thế tiếp tục. Nhưng sau khi F_5 được tính để có được F_6, thì F_5 sẽ không được giữ lại. Như vậy để tính F_7 sau đó, F_5 lại phải được tính lại. Cây đệ quy đã cho chúng ta thấy rất rõ rằng chương trình đệ quy phải lập đi lập lại nhiều phép tính một cách không cần thiết.Tổng thời gian để hàm đệ quy tính được F_n là một hàm mũ của n.

Cũng giống như việc tính giai thừa, chúng ta có thể có được một chương trình đơn giản bằng cách giữ lại ba biến, đó là trị của số *Fibonacci* mới nhất và hai số *Fibonacci* kế trước:

```
int fibonacci(int n)
/*    fibonacci: phiên bản không đệ quy.
pre:  n là một số không âm.
post: trả về số Fibonacci thứ n.
*/
{
```

```
   int current;         //  số Fibonacci hiện tại Fi
   int last_value;      //  Fi-1
   int last_but_one;    //  Fi-2
   if (n <= 0) return 0;
   else if (n == 1) return 1;
   else {
      last_but_one = 0;
      last_value = 1;
      for (int i = 2; i <= n; i++) {
         current = last_but_one + last_value;
         last_but_one = last_value;
         last_value = current;
      }
      return current;
   }
}
```

Chương trình không đệ quy này có thời gian chạy tỉ lệ với n.

6.2.4.3. So sánh giữa đệ quy và không đệ quy

Đâu là điều khác nhau cơ bản giữa chương trình vừa rồi với chương trình đệ quy? Để trả lời câu hỏi này, chúng ta hãy xem xét lại cây đệ quy. Việc phân tích cây đệ quy sẽ đem lại nhiều thông tin hữu ích giúp chúng ta biết được khi nào thì nên sử dụng đệ quy và khi nào thì không.

Nếu một hàm gọi đệ quy chính nó chỉ có một lần thì cây đệ quy sẽ có dạng rất đơn giản: đó là một chuỗi các mắc xích, có nghĩa là, mỗi nút chỉ có duy nhất một con. Nút con này tương ứng với một lần gọi đệ quy. Đối với hàm giai thừa, đó chỉ đơn giản là một danh sách các yêu cầu việc tính toàn các số giai thừa từ (n-1)! cho đến 1!. Bằng cách đọc cây đệ quy từ dưới lên trên thay vì từ trên xuống dưới, chúng ta có ngay chương trình không đệ quy từ một chương trình đệ quy. Khi một cây suy giảm thành một danh sách, việc chuyển từ chương trình đệ quy thành chương trình không đệ quy thường dễ dàng, và kết quả có được thường tiết kiệm cả không gian lẫn thời gian.

Lưu ý rằng một hàm gọi đệ quy chính bản thân nó có thể có nhiều dạng khác nhau. Dòng gọi đệ quy hoặc là chỉ xuất hiện một lần trong một vòng lặp nhưng thực sự lại được gọi nhiều lần, hoặc là xuất hiện hai lần trong lệnh rẽ nhánh **`if`**, **`else`** nhưng thực sự chỉ được thực hiện có một lần.

Cây đệ quy tính các số *Fibonacci* không phải là một chuỗi các mắc xích. Nó chứa một số rất lớn các nút biểu diễn những công việc được lặp lại. Khi chương trình đệ quy chạy, nó tạo một ngăn xếp để sử dụng trong khi duyệt qua cây. Tuy nhiên, các kết quả lưu vào ngăn xếp khi lấy ra chỉ được sử dụng có một lần và sẽ bị mất đi mà không thể sử dụng lại (vì đỉnh ngăn xếp sau khi được truy xuất cần được loại bỏ mới có thể truy xuất tiếp những phần tử khác trong ngăn xếp), và như vậy một công việc nào đó có thể phải được thực hiện nhiều lần.

Trong những trường hợp như vậy, tốt hơn hết là thay ngăn xếp bằng một cấu trúc dữ liệu khác, một cấu trúc dữ liệu mà cho phép truy nhập vào nhiều vị trí khác nhau thay vì chỉ ở đỉnh như ngăn xếp. Trong ví dụ đơn giản về các số *Fibonacci*, chúng ta chỉ cần thêm hai biến tạm để chứa hai trị cần cho việc tính số mới.

Cuối cùng, khác với việc một chương trình đệ quy tự tạo cho mình một ngăn xếp riêng, bằng cách tạo một ngăn xếp tường minh, chúng ta luôn có thể chuyển mọi chương trình đệ quy thành chương trình không đệ quy. Chương trình không đệ quy thường phức tạp và khó hiểu hơn. Nếu một chương trình đệ quy có thể chạy được với một không gian và thời gian cho phép, thì chúng ta không nên khử đệ quy trừ trường hợp ngôn ngữ lập trình mà chúng ta sử dụng không có khả năng đệ quy.

6.2.4.4. So sánh giữa *Fibonacci* và Tháp Hà Nội: kích thước của lời giải

Hàm đệ quy tính các số *Fibonacci* và hàm đệ quy giải bài toán Tháp Hà Nội đều có dạng chia để trị rất giống nhau. Mỗi hàm đều gọi đệ quy chính nó hai lần cho các trường hợp nhỏ hơn. Tuy nhiên, vì sao chương trình Tháp Hà Nội lại vô cùng hiệu quả trong khi chương trình tính các số *Fibonacci* lại hoàn toàn ngược lại? Câu trả lời liên quan đến kích thước của lời giải. Để tính một số *Fibonacci*, rõ ràng kết quả mà chúng ta cần chỉ có mỗi một số, và chúng ta mong muốn việc tính toán sẽ hoàn tất qua một số ít các bước, như là các dòng lệnh trong chương trình không đệ quy. Trong khi đó, chương trình đệ quy *Fibonacci* lại thực hiện quá nhiều bước. Trong chương trình Tháp Hà Nội, ngược lại, kích thước của lời giải là số các lời chỉ dẫn cần in ra cho các linh mục và là một hàm mũ của tổng số đĩa.

6.2.5. Các nhận xét

Để đi đến kết luận về giải pháp lựa chọn cho một chương trình đệ quy hay không đệ quy, điểm bắt đầu tốt nhất cũng là xem xét cây đệ quy. Nếu cây đệ quy có dạng đơn giản, chương trình không đệ quy sẽ tốt hơn. Nếu cây chứa nhiều công việc được lặp lại mà các cấu trúc dữ liệu khác thích hợp hơn là ngăn xếp, thì đệ quy cũng không còn cần thiết nữa. Nếu cây đệ quy thực sự "rậm rạp", mà trong đó số công việc lặp lại không đáng kể, thì chương trình đệ quy là giải pháp tốt nhất.

Ngăn xếp được sử dụng khi đệ quy (do chương trình đệ quy tự tạo lấy) được xem như một danh sách chứa các công việc cần trì hoãn của chương trình. Nếu danh sách này có thể được tạo trước, thì chúng ta nên viết chương trình không đệ quy, ngược lại, chúng ta sẽ viết chương trình đệ quy. Đệ quy như một cách tiếp cận từ trên xuống khi cần giải quyết vấn đề, nó chia bài toán thành những bài toán nhỏ hơn, hoặc chọn ra bước chủ yếu và trì hoãn các bước còn lại. Chương trình không đệ quy gần với cách tiếp cận từ dưới lên, nó bắt đầu từ những cái đã biết và từng bước xây dựng nên lời giải.

Một chương trình đệ quy luôn có thể được thay thế bởi một chương trình không đệ quy có sử dụng ngăn xếp. Điều ngược lại cũng luôn đúng: một chương trình không đệ quy có sử dụng ngăn xếp có thể được thay bởi chương trình đệ quy không có ngăn xếp. Do đó, không những người lập trình thường phải tự hỏi có nên khử đệ quy hay không, mà đôi khi chính họ lại cần đặt câu hỏi ngược lại, có nên chuyển thành đệ quy một chương trình không đệ quy có sử dụng ngăn xếp hay không. Điều thứ hai này có thể dẫn đến một chương trình gần với bản chất tự nhiên của bài toán hơn và do đó dễ hiểu hơn. Đó cũng là một cách để cải tiến cách tiếp cận bài toán cũng như các kết quả đạt được.

Có một số lời khuyên trong việc sử dụng đệ quy, đó là chúng ta không nên dùng đệ quy khi câu trả lời cho bất kỳ câu hỏi nào dưới đây đều là không:

- Bản thân giải thuật hoặc cấu trúc dữ liệu có tính chất đệ quy một cách tự nhiên?
- Lời giải đệ quy ngắn gọn và dễ hiểu hơn?
- Lời giải đệ quy đòi hỏi một không gian và thời gian chấp nhận được?

Các bước gợi ý trong việc khử đệ quy đuôi

1. Sử dụng một biến để thay thế cho việc gọi đệ quy trở lại.
2. Sử dụng một vòng lặp với điều kiện kết thúc giống như điều kiện dừng của đệ quy.
3. Đặt tất cả các lệnh vốn cần thực hiện trong lần gọi đệ quy đuôi vào trong vòng lặp.
4. Thay lệnh gọi đệ quy bằng một phép gán.
5. Dùng các lệnh gán để gán các trị như là các thông số mà hàm đệ quy lẽ ra nhận được.
6. Trả về trị cho biến đã định nghĩa ở bước 1.

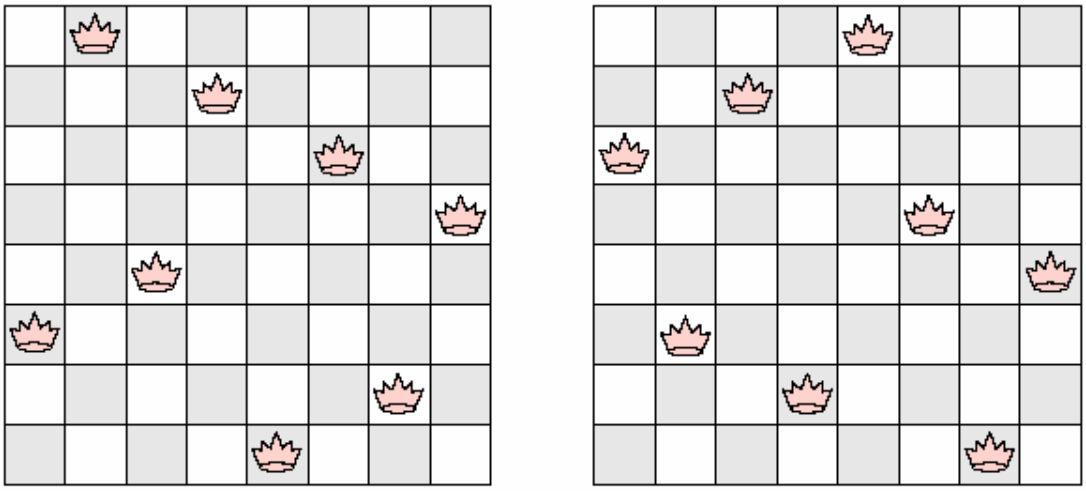
Các bước gợi ý trong việc khử đệ quy một cách tổng quát

Chúng ta có thể tạo một ngăn xếp để chứa các bản ghi. Lệnh goi đệ quy và lệnh trả về từ hàm đệ quy có thể dược thay thế như sau. Mỗi lệnh gọi đệ quy có thể được thay bởi:

1. Đưa vào ngăn xếp một bản ghi chứa các biến cục bộ, các thông số và vị trí dòng lệnh ngay sau lệnh gọi đệ quy.
2. Gán mọi thông số về các trị mới thích hợp.
3. Trở về thực hiện dòng lệnh đầu tiên trong giải thuật đệ quy.
4. Mỗi lệnh trả về của hàm đệ quy được thay bởi:
5. Lấy từ ngăn xếp để khôi phục mọi biến cục bộ và thông số.
6. Bắt đầu thực hiện dòng lệnh tại vị trí mà trước đó đã được cất trong ngăn xếp.

6.3. Phương pháp quay lui (*backtracking*)

Như một ứng dụng khá phức tạp về đệ quy, chúng ta hãy xem xét một câu đố rất phổ biến về việc làm cách nào để đặt tám con hậu trên một bàn cờ có tám hàng và tám cột sao cho chúng không thể nhìn thấy nhau. Theo luật của bàn cờ thì một con hậu có thể nhìn thấy những con cờ khác nằm trên hàng, hoặc cột, hoặc hai đường chéo có chứa nó.



Hình 6.9- Hai cấu hình thỏa điều kiện của bài toán tám con hậu.

Làm cách nào để giải câu đố này, điều này còn khá mơ hồ. Ngay cả nhà toán học nổi tiếng Gauss C. F. vẫn chưa tìm ra được một lời giải đầy đủ khi ông xem xét câu đố này vào năm 1850. Điều thường xảy ra đối với các câu đố dường như là không có một cách nào có thể đưa ra các lời giải có được sự phân tích đầy đủ, mà chỉ có những lời giải được phát hiện một cách tình cờ do sự may mắn của một số lần áp dụng phương pháp thử sai, hoặc sau khi đã thực hiện một số lượng khổng lồ các phép tính. Hình 6.9 cho chúng ta hai phương án thỏa yêu cầu câu đố và cũng cho chúng ta tin rằng câu đố có lời giải.

Trong phần này, chúng ta sẽ phát triển hai chương trình để giải bài toán tám con hậu, đồng thời chúng ta cũng sẽ thấy được rằng, việc lựa chọn các cấu trúc dữ liệu có thể ảnh hưởng lên một chương trình đệ quy như thế nào.

6.3.1. Lời giải cho bài toán tám con hậu

Bất kỳ ai khi thử tìm lời giải cho bài toán tám con hậu thường ngay lập tức bị cuốn hút vào việc tìm cách đặt những con hậu lên bàn cờ, có thể là ngẫu nhiên, có thể theo một trật tự luận lý nào đó, nhưng dù cách nào đi nữa thì điều chắc chắn xảy ra là con hậu được đặt sau sẽ không bao giờ được nhìn thấy các con hậu đã được đặt trước đó. Bằng cách này, nếu may mắn, một người có thể đặt được cả

tám con hậu thỏa yêu cầu bài toán, và đó là một lời giải. Nếu không may, một hoặc nhiều con hậu sẽ phải được lấy đi để đặt lại vào những chỗ khác và việc tìm lời giải lại được tiếp tục.

Chúng ta sẽ viết một hàm đệ quy **solve_from** để giải bài toán này. Công việc cần làm tại một thời điểm (một trạng thái nào đó của bàn cờ mà số hậu chưa đủ) là:

- Đặt thêm một con hậu vào một vị trí hợp lệ.
- Gọi đệ quy để xử lý tương tự với số hậu cần xử lý tiếp đã giảm 1.

Lớp **Queens** dưới đây sẽ có một số phương thức cần thiết mà chúng ta sẽ bàn đến chi tiết sau. Ở đây chúng chỉ cần biết rằng đối tượng **configuration** của nó sẽ chứa một trạng thái của bàn cờ. Khi nó làm thông số cho lần đệ quy bên trong, nó đã có được thêm một con hậu hợp lệ. Trong lần gọi **solve_from** đầu tiên từ chương trình chính, số hậu cần giải quyết là 8 và bàn cờ chưa có hậu nào.

```
solve_from(Queens configuration)
{
   1. if configuration Queens đã có tám con hậu
      1. print configuration
   2. else
      1. for mỗi ô của bàn cờ mà chưa bị nhìn thấy bởi con hậu nào {
         1. Đặt một con hậu lên ô p của configuration;
         2. solve_from(configuration);
         3. Lấy con hậu ra khỏi ô p của configuration;
         }
}
```

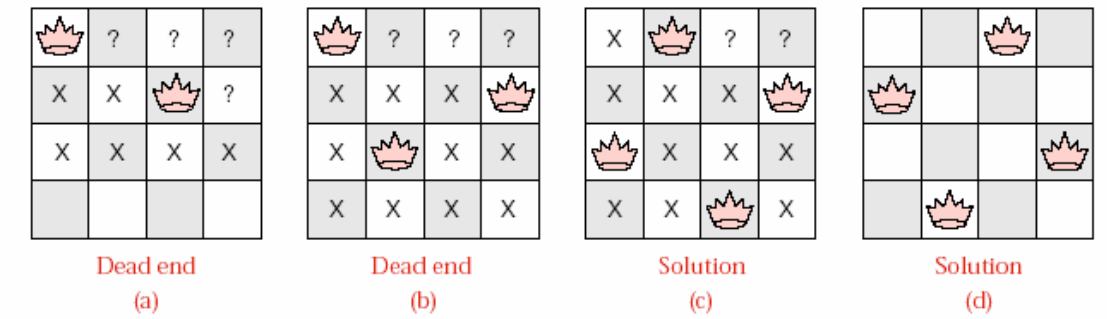
Rõ ràng là sau mỗi bước đệ quy, kích thước bài toán giảm dần. Chúng ta có điểm dừng của đệ quy là khi cả 8 con hậu đều đã tìm được vị trí thích hợp (lệnh rẽ nhánh **if**), hoặc khi không còn tìm được vị trí nào hợp lệ cho con hậu cần đặt tiếp nữa (trường hợp vòng lặp **for** đã quét hết các vị trí hợp lệ còn lại).

Việc đặt một con hậu ở một ô p chỉ là một bước thử nghiệm, chúng ta vẫn còn chưa thay đổi vị trí của nó trong khi mà chúng ta còn có thể tiếp tục đặt những con hậu khác cho đến khi đạt được cả 8 con. Và cũng giống như khi người ta làm bằng tay, khi không còn vị trí nào có thể đặt tiếp hậu, hàm **solve_from** cũng phải lấy đi những con hậu đã đặt (dòng lệnh `2.1.3`) để tiếp tục thử với những vị trí hợp lệ khác (vòng lặp `for` sẽ lần lượt chuyển sang các vị trí này). Rõ ràng là nếu không thể tiếp tục đặt thêm một con hậu nào đó nữa thì việc quay lui là để thay đổi lần thử nghiệm vừa rồi sang một phương án khác để tiếp tục tìm lời giải. Tuy nhiên, giải thuật quay lui của chúng ta còn có một đặc điểm nữa là khi đã đạt được lời giải cho cả 8 con hậu, thì chương trình cũng vẫn quay lui, và việc quay lui này là để tìm thêm nhiều lời giải khác. Tóm lại, khi một lần gọi đệ quy bên trong

kết thúc, chương trình của chúng ta sẽ luôn lùi một bước để khảo sát tiếp các khả năng khác còn lại, và giải thuật sẽ cho đáp án là tất cả các lời giải của bài toán.

6.3.2. Ví dụ với bốn con Hậu

Chúng ta sẽ xét xem giải thuật trên được thực hiện như thế nào cho một trường hợp đơn giản, đó là bài toán đặt bốn con hậu lên bàn cờ 4x4, hình 6.10.



Hình 6.10 – Lời giải cho bài toán bốn con hậu

Chúng ta cần phải đặt mỗi con hậu lên một hàng của bàn cờ. Chúng ta luôn bắt đầu thử từ ô cực trái còn hợp lệ của hàng. Ở hàng trên cùng chúng ta chọn ô ở góc trái (hình 6.10a). Các dấu hỏi đánh dấu những lựa chọn hợp lệ khác mà chúng ta chưa thử đến. Trước khi thử các ô này, chúng ta chuyển sang hàng thứ hai. Hai ô đầu tiên đã bị nhìn thấy bởi con hậu ở hàng 1, chúng ta đánh dấu bằng dấu chéo. Ô thứ 3 và thứ 4 còn tự do, chùng ta tiếp tục thử với ô thứ 3 và đáng dấu hỏi cho ô thứ 4. Tiếp theo chúng ta chuyển xuống hàng thứ 3, nhưng cả bốn ô ở hàng này đều đã bị nhìn thấy bởi một trong hai con hậu ở hai hàng trên. Xem như chúng ta đã gặp điểm chết.

Khi gặp một điểm chết, chúng ta cần phải quay ngược lại và bỏ đi sự lựa chọn mới nhất để thử một khả năng khác. Trường hợp này thể hiện trong hình 6.10b), con hậu ở hàng thứ nhất không đổi nhưng con hậu ở hàng thứ hai đã được thử với vị trí còn lại là ô thứ 4 (ô thứ 3 vừa thử thất bại được đánh dấu chéo). Sau đó chúng ta thấy ở hàng thứ ba chỉ có ô thứ 2 là tự do, nhưng khi tiếp tục sang hàng thứ 4 thì cũng không còn ô nào tự do. Lúc này chúng ta lại gặp một điểm chết mới, chúng ta lại cần phải quay ngược lại.

Tại điểm này, quay ngược lên hàng trên, chúng ta thấy hàng thứ ba không còn khả năng lựa chọn nào, ngược lên hàng thứ hai cũng vậy. Do đó chúng ta phải quay ngược lên đến hàng thứ nhất, ô thử mới trong hàng này là ô thứ 2 (hình 6.10c). Khi đi xuống, chúng ta thấy hàng thứ hai chỉ có một khả năng lựa chọn, đó là ô thứ 4. Xuống hàng thứ ba chỉ có ô thứ 1 là tự do. Cuối cùng, ở hàng thứ tư có một ô tự do là ô 3. Tuy nhiên đây chỉ là một trường hợp thỏa yêu cầu

bài toán, mà chưa phải là một lời giải trọn vẹn cho bài toán đặt bốn con hậu lên bàn cờ 4x4.

Nếu muốn tìm mọi lời giải, chúng ta có thể tiếp tục bằng cách tương tự: quay ngược lại lần lựa chọn mới nhất để thử với khả năng tiếp theo. Trong hình 6.10c không còn lựa chọn nào khác ở hàng thứ tư, hàng thứ ba và hàng thứ hai. Do đó chúng ta ngược lên đến hàng thứ nhất, thử ô thứ 3. Lựa chọn này dẫn đến một lời giải duy nhất ở hình 6.10d.

Cuối cùng, khi thử với ô 4 ở hàng thứ nhất, chúng ta cũng không thu thêm một kết quả nào. Thực ra, cấu hình với con hậu ở ô 3 và con hậu ở ô 4 của hàng thứ nhất chính là hai hình ảnh ngược của ô 2 và ô 1 tương ứng. Nếu chúng ta đi từ trái sang phải trên hình 6.10c, chúng ta có được cấu hình ở hình 6.10d.

6.3.3. Phương pháp quay lui (*Backtracking*)

Phương pháp trên được gọi là giải thuật quay lui. Trong đó, việc tìm một lời giải đầy đủ được thực hiện bằng cách xây dựng các lời giải riêng phần sao cho chúng luôn thỏa những điều kiện của bài toán. Giải thuật được áp dụng để kéo dài một lời giải riêng phần cho thành một lời giải trọn vẹn. Tuy nhiên, tại một bước nào đó, nếu có sự vi phạm điều kiện của bài toán, giải thuật sẽ quay ngược trở lại, bỏ đi sự lựa chọn mới nhất để thử với một khả năng cho phép khác.

Giải thuật quay lui tỏ ra hiệu quả trong những trường hợp mà ban đầu tưởng như có rất nhiều khả năng lựa chọn, nhưng sau đó chỉ một số ít khả năng là còn sót lại sau tiến trình kiểm tra xa hơn. Trong các bài toán xếp thời khóa biểu, chẳng hạn trong việc tổ chức các vòng đấu thể thao, sự lựa chọn thời gian cho một số trận đấu ban đầu thường là rất dễ, nhưng càng về sau, các ràng buộc sẽ làm giảm đáng kể các khả năng có thể.

Phần tiếp theo đây chúng ta sẽ tìm hiểu cách hiện thực cụ thể cho bài toán con hậu cũng như một số chương trình liên quan đến các trò chơi, để thấy được ý tưởng đệ quy và giải thuật quay lui là cốt lõi của những bài toán ở dạng này."

6.3.4. Phác thảo chung cho chương trình đặt các con hậu lên bàn cờ

6.3.4.1. Chương trình chính

Mặc dù chúng ta còn phải xác định rất nhiều chi tiết về cấu trúc dữ liệu để chứa các vị trí của các con hậu trên bàn cờ, nhưng ở đây chúng ta vẫn có thể viết trước chương trình chính để gọi hàm đệ quy mà chúng ta đã phác thảo.

Đầu tiên các thông tin về chương trình sẽ được in ra. Do việc kiểm tra chương trình với những bài toán nhỏ hơn là có ích, chẳng hạn với bài toán chỉ có bốn con hậu, chúng ta sẽ cho phép người sử dụng nhập vào số con hậu theo ý muốn. Đây cũng là kích thước của bàn cờ (**board-size**). Hằng số **max_board** sẽ được khai báo trong file **queens.h**.

```
int main()
/*
pre:  Người sử dụng cần cho biết kích thước của bàn cờ.
post: Mọi lời giải cho bài toán các con hậu được in ra.
uses: lớp Queens và hàm đệ quy solve_from.
*/
{
   int board_size;
   print_information();
   cout << "What is the size of the board? " << flush;
   cin  >> board_size;
   if (board_size < 0 || board_size > max_board)
      cout << "The number must be between 0 and " << max_board << endl;


   else {
      Queens configuration(board_size);  // Bàn cờ chưa có hậu nào.
      solve_from(configuration);
   }
}
```

6.3.4.2. Lớp Queens

Định nghĩa biến **Queens configuration(board_size)** dùng một *constructor* có thông số của lớp **Queens** để tạo một bàn cờ có kích thước theo sự lựa chọn của người sử dụng và khởi tạo một đối tượng **Queens** rỗng có tên là **configuration**. Đối tượng Queens rỗng này được gởi cho hàm đệ quy của chúng ta, trong đó các con hậu sẽ được đặt lần lượt lên bàn cờ.

Phác thảo trong phần 6.3.1 cho thấy lớp Queens cần các phương thức như in một trạng thái, thêm một con hậu vào một ô trên bàn cờ, lấy con hậu này đi, kiểm tra xem một ô nào đó có tự do hay không. Ngoài ra, để hiện thực hàm **solve_from**, lớp Queens cũng cần chứa dữ liệu là **board_size** để chứa kích thước bàn cờ cũng như thuộc tính **count** để đếm số con hậu đã được đặt lên bàn cờ.

Sau khi bắt đầu xây dựng một cấu hình, chúng ta sẽ tìm ô kế tiếp bằng cách nào? Ngay khi một con hậu vừa được đặt trong một hàng nào đó, không ai lại đi mất thì giờ vào việc tìm một vị trí khác cho con hậu mới cũng trên cùng hàng đó, do nó chắc chắn sẽ bị nhìn thấy bởi con hậu vừa đặt xong. Không thể có nhiều hơn một con hậu trong cùng một hàng. Mục đích của chúng ta là đặt cho được số con hậu được yêu cầu lên bàn cờ (board_size), và ở đây cũng chỉ có **board_size**

hàng. Do đó mỗi hàng phải có chính xác chỉ một con hậu. Đây là nguyên tắc tổ chim câu (*pigeonhole principle*): nếu chúng ta có n chú chim và n cái tổ và không cho phép nhiều hơn một con trong một tổ thì chúng ta chỉ có thể đặt mỗi con vào một tổ và phải dùng hết các tổ. Chúng ta có thể tiến hành bằng cách đặt các con hậu vào bàn cờ, mỗi lần cho một hàng, bắt đầu từ hàng số 0, như vậy **count** không chỉ là để đếm số hậu đã được đặt mà còn là chỉ số của hàng sẽ được đặt hậu kế tiếp.

Các đặc tả cho các phương thức của lớp **Queens** như sau:

bool unguarded(int col) const;
post: trả về true nếu ô thuộc hàng **count (**hàng đang được xử lý kế tiếp) và cột **col** không bị nhìn thấy bởi một con hậu nào khác; ngược lại trả về false.

void insert(int col);
pre: Ô tại hàng **count** và cột **col** không bị nhìn thấy bởi bất kỳ con hậu nào.
post: Một con hậu vừa được đặt vào ô tại hàng **count** và cột **col**, **count** tăng thêm 1.

void remove(int col);
pre: Ô tại hàng **count-1** và cột **col** đang có một con hậu.
post: Con hậu trên được lấy đi, **count** giảm đi 1.

bool is_solved() const;
post: trả về true nếu số hậu đã đặt vào bàn cờ bằng với kích thước bàn cờ **board_size**; ngược lại, trả về false.

6.3.4.3. Hàm đệ quy solve_from

Với các đặc tả trên hàm **solve_from** đã phác thảo trong phần trước đã được cụ thể hóa như sau. Tham số **configuration** được gởi tham chiếu do bên trong hàm có thay đổi nó.

```
void solve_from(Queens &configuration)
/*
pre:  Bàn cờ đã chứa được count hậu hợp lệ tại hàng 0 đến hàng count -1.
post: n con hậu đã được đặt hợp lệ lên bàn cờ.
uses: lớp Queens và các hàm solve_from, recursively.
*/


{
   if (configuration.is_solved()) configuration.print();
   else
      for (int col = 0; col < configuration.board_size; col++)
         if (configuration.unguarded(col)) {
            configuration.insert(col);
            solve_from(configuration);// Gọi đệ quy tiếp để thêm các con hậu còn lại.
            configuration.remove(col);
         }
}
```

6.3.5. Tinh chế: Cấu trúc dữ liệu đầu tiên và các phương thức

Một cách hiển nhiên để hiện thực cấu hình Queens là lưu bàn cờ như một mảng hai chiều, mỗi phần tử biểu diễn việc có hay không một con hậu. Vậy mảng hai chiều là lựa chọn đầu tiên của chúng ta cho cấu trúc dữ liệu. Tập tin queens.h chứa định nghĩa sau:

```
const int max_board = 30;

class Queens {
public:
   Queens(int size);
   bool is_solved() const;
   void print() const;
   bool unguarded(int col) const;
   void insert(int col);
   void remove(int col);
   int  board_size; // Kích thước của bàn cờ bằng số hậu cần đặt.
private:
   int  count;//Chứa số hậu đã đặt được và cũng là chỉ số của hàng sẽ được đặt tiếp hậu.
   bool queen_square[max_board][max_board];
};
```

Với cấu trúc dữ liệu này, phương thức thêm một con hậu dễ dàng như sau:

```
void Queens::insert(int col)
/*
pre: Ô tại hàng count và cột col không bị nhìn thấy bởi bất kỳ con hậu nào.
post: Một con hậu vừa được đặt vào ô tại hàng count và cột col, count tăng thêm 1.
*/
{
   queen_square[count++][col] = true;
}
```

Các phương thức is_solved, remove, print cũng rất dễ và chúng ta xem như bài tập.

Để khởi tạo cấu hình Queens, chúng ta cần *constructor* có thông số để đặt kích thước cho bàn cờ:

```
Queens::Queens(int size)
/*
post: Bàn cờ được khởi tạo chưa có hậu nào.
*/
{
   board_size = size;
   count = 0;
   for (int row = 0; row < board_size; row++)
      for (int col = 0; col < board_size; col++)
         queen_square[row][col] = false;
}
```

Thuộc tính `count` khởi gán là 0 vì chưa có con hậu nào được đặt lên bàn cờ. *Constructor* này được thực hiện khi chúng ta vừa khai báo một đối tượng `Queens` trong chương trình chính.

Cuối cùng, chúng ta cần viết phương thức kiểm tra một ô tại một cột nào đó trên hàng đầu tiên chưa có hậu (xét từ trên xuống) có bị nhìn thấy bởi các con hậu đã có trên bàn cờ hay không. Chúng ta cần xét cột hiện tại và hai đường chéo đi qua ô này. Việc xét cột thật dễ dàng, còn việc xét đường chéo cần một số tính toán về chỉ số. Chúng ta hãy xem hình 6.11 cho trường hợp bàn cờ 4x4.

Chúng ta có thể gọi tên cho bốn hướng của hai đường chéo như sau: đường chéo trái-dưới (*lower-left*) hướng xuống dưới về bên trái, đường chéo phải-dưới (*lower-right*), đường chéo trái-trên (*upper-left*), và đường chéo phải trên (*upper-right*).

Trước tiên, chúng ta hãy xem xét đường chéo trái-trên ở hình 6.11c. Nếu chúng ta bắt đầu từ ô `[row][col]`, các ô thuộc đường chéo trái-trên có toạ độ `[row-i][col-i]` với `i` là số nguyên dương. Đường chéo trái-trên này phải kết thúc khi gặp cạnh trên của bàn cờ `(row-i==0)` hoặc cạnh trái của bàn cờ `(col-i==0)`. Chúng ta có thể dùng vòng lặp tăng `i` từ 1 cho đến khi `row-i<0` hoặc `col-i<0`.

Chúng ta có thể làm tương tự cho ba đường chéo còn lại. Tuy nhiên, khi kiểm tra một ô có bị nhìn thấy bởi các con hậu hay không thì chúng ta không cần kiểm tra hai đường chéo dưới của ô này vì theo giải thuật các hàng dưới vẫn chưa có hậu.

```
bool Queens::unguarded(int col) const
/*
post: trả về true nếu ô thuộc hàng count (hàng đang được xử lý kế tiếp) và cột col không bị
      nhìn thấy bởi một con hậu nào khác; ngược lại trả về false.
*/


{
   int i;
   bool ok = true; // sẽ được gán lại false nếu chúng ta tìm thấy hậu trên cùng cột hoặc
                                                          đường chéo.
   for (i = 0; ok && i < count; i++)
      ok = !queen_square[i][col];                // kiểm tra phần trên của cột.
   for (i = 1; ok && count - i >= 0 && col - i >= 0; i++)
      ok = !queen_square[count - i][col - i];  // kiểm tra phần trên bên trái của
                                                          đường chéo.
   for (i = 1; ok && count - i >= 0 && col + i < board_size; i++)
      ok = !queen_square[count - i][col + i];  // kiểm tra phần trên bên phải của
                                                          đường chéo.
   return ok;
}
```

6.3.6. Xem xét lại và tinh chế

Chương trình mà chúng ta vừa hoàn tất đáp ứng hoàn toàn cho bài toán tám con hậu. Kết quả chạy chương trình cho chúng ta 92 lời giải khác nhau. Tuy nhiên, với bàn cờ có kích thước lớn hơn, thời gian cần để chạy chương trình rất lớn. Bảng sau đây cho chúng ta một vài ví dụ:

Kích thước	8	9	10	11	12	13
Số lời giải	92	352	724	2680	14200	73712
Thời gian (*second*)	0.05	0.21	1.17	6.62	39.11	243.05
Thời gian cho một lời giải (*ms.*)	0.54	0.6	1.62	2.47	2.75	3.30

Như chúng ta thấy, số lượng lời giải tăng rất nhanh theo kích thước của bàn cờ, và thời gian tăng còn nhanh hơn rất nhiều, do thời gian cho một lời giải cũng tăng theo kích thước bàn cờ. Nếu muốn giải cho các bàn cờ kích thước lớn, chúng ta cần tìm một chương trình hiệu quả hơn.

Chúng ta hãy tìm xem vì sao chương trình của chúng ta chạy quá lâu như vậy. **Việc gọi đệ quy và quay lui rõ ràng là chiếm nhiều thời gian, nhưng thời gian này lại phản ánh đúng phương pháp cơ bản mà chúng ta dùng để giải bài toán.** Đó chính là bản chất của giải thuật và nó được lý giải bởi một số lượng lớn lời giải. Một số vòng lặp trong phương thức `unguarded` cũng đòi hỏi một lượng thời gian đáng kể. Chúng ta hãy thử xét xem có thể bỏ bớt một vài vòng lặp được chăng. Có cách nào để xét một ô có bị nhìn thấy bởi các con hậu hay không mà không phải xét hết các ô trên cùng cột và hai đường chéo bắc ngang?

Có một cách để thực hiện điều này, đó là cách thay đổi dữ liệu mà chúng ta lưu giữ trong mảng. Thay vì lưu thông tin các ô nào đã có các con hậu, chúng ta có thể dùng mảng để nắm giữ tất cả **các ô đã bị các con hậu nhìn thấy**. Từ đó, chúng ta sẽ dễ dàng hơn trong việc kiểm tra xem một ô có bị các con hậu nhìn thấy hay không. Một sự thay đổi nhỏ có thể giúp ích cho việc quay lui, bởi vì một ô có thể có nhiều hơn một con hậu nhìn thấy. Với mỗi ô, chúng ta có thể lưu **một số để đếm số con hậu nhìn thấy nó**. Khi một con hậu được thêm vào, chúng ta tăng biến đếm này thêm 1 cho tất cả các ô cùng hàng, cùng cột và trên hai đường chéo của nó. Ngược lại, khi lấy đi một con hậu, các biến đếm tương ứng này cũng cần giảm bớt 1.

Việc lập trình theo phương án này được dành lại như bài tập. Chúng ta nhận xét thêm rằng, tuy phương án mới này có chạy nhanh hơn chương trình đầu tiên nhưng vẫn còn một số vòng lặp để cập nhật lại các biến đếm vừa nêu. Suy nghĩ

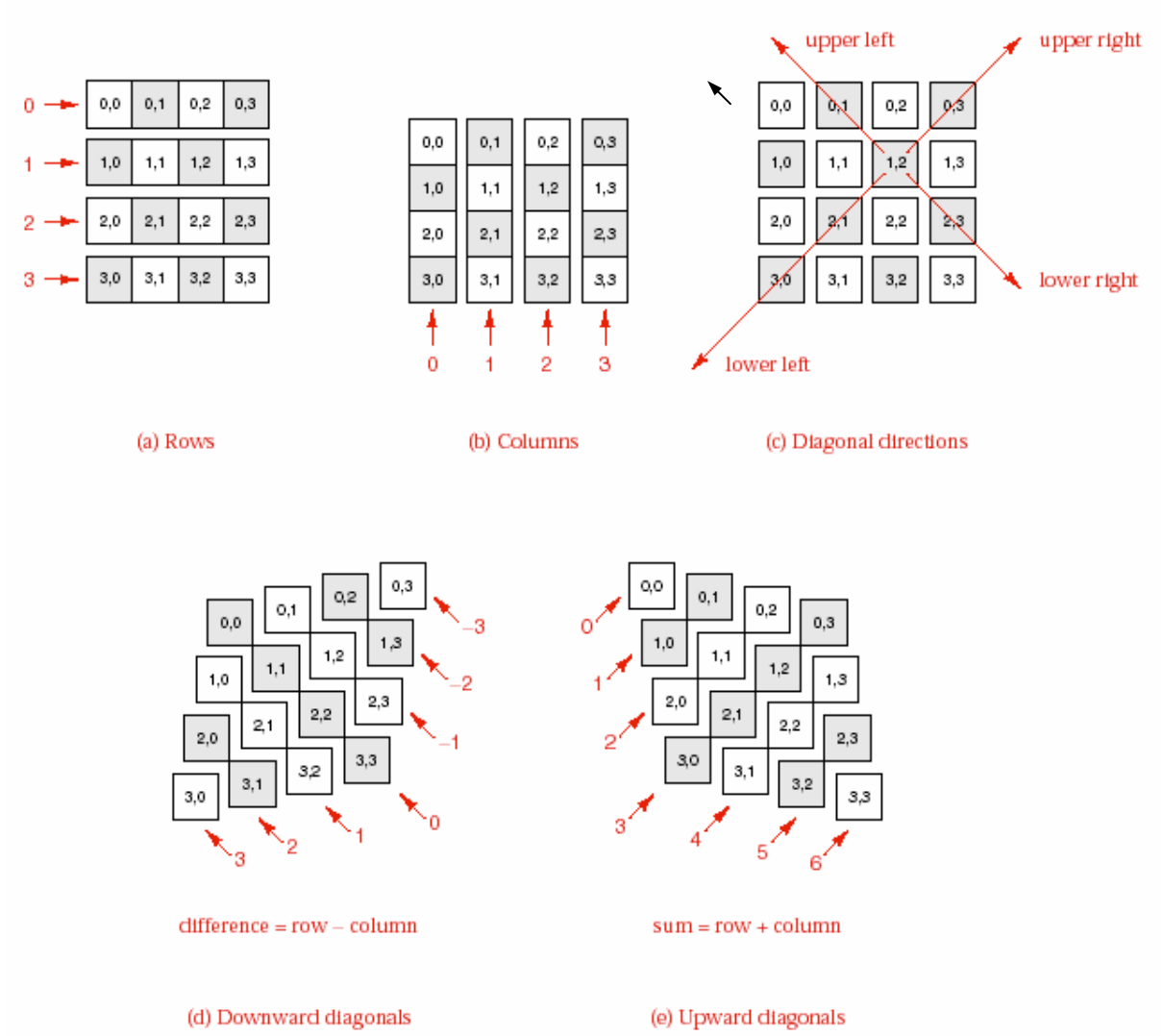
thêm một chút nữa, chúng ta sẽ thấy rằng chúng ta có thể loại bỏ hoàn toàn các vòng lặp này.

Ý tưởng chính ở đây là việc nhận ra rằng mỗi hàng, mỗi cột, mỗi đường chéo trên bàn cờ đều chỉ có thể chứa nhiều nhất là một con hậu. (Nguyên tắc tổ chim câu cho thấy rằng, trong một lời giải, mọi hàng và mọi cột đều có con hậu, nhưng không phải mọi đường chéo đều có con hậu, do số đường chéo nhiều hơn số hàng và số cột.)

Từ đó, chúng ta có thể nắm giữ **các ô chưa bị các con hậu nhìn thấy** bằng cách sử dụng 3 mảng có các phần tử kiểu `bool`: **`col_free`**, **`upward_free`**, và **`downward_free`**, trong đó các đường chéo từ dưới lên và trái sang phải được gọi là *upward*, các đường chéo từ trên xuống và trái sang phải được gọi là *downward* (hình 6.11d và e). Do chúng ta đặt các con hậu lên bàn cờ mỗi lần tại một hàng, bắt đầu từ hàng 0, chúng ta không cần một mảng để biết được hàng nào còn trống.

Cuối cùng, để in một cấu hình, chúng ta cần **biết số thứ tự của cột có chứa con hậu trong mỗi hàng**, chúng ta sẽ dùng một mảng các số nguyên, mỗi phần tử dành cho một hàng và chứa số của cột chứa con hậu trong hàng đó.

Cho đến bây giờ, chúng ta đã có thể giải quyết trọn vẹn bài toán mà không cần đến mảng hai chiều biểu diễn bàn cờ như phương án đầu tiên nữa, và chúng ta cũng đã có thể loại mọi vòng lặp trừ các vòng lặp khởi tạo các trị ban đầu cho các mảng. Nhờ vậy, thời gian cần thiết để chạy chương trình mới này phản ánh một cách chặt chẽ số bước cần khảo sát trong phương pháp quay lui.



Hình 6.11 – Các chỉ số của các ô trong bàn cờ

Chúng ta sẽ đánh số cho các ô trong một đường chéo như thế nào? Chỉ số của đường chéo *upward* dài nhất trong mảng hai chiều như sau:

```
[board_size-1][0],[board_size-2][1], ...,[0][board_size-1]
```

Đặc tính chung của các chỉ số này là tổng của hàng và cột luôn là (**board_size-1**). Điều này gợi ý rằng, như hình 6.11e, bất kỳ một đường chéo *upward* nào cũng có tổng của hàng và cột của mọi ô đều là một hằngsố. Tổng này bắt đầu từ **0** cho đường chéo *upward* có chiều dài là 1 tại góc trái trên cùng của mảng, cho đến(**2xboard_size-2**) cho đường chéo *upward* có chiều dài là 1 tại góc phải dưới cùng của mảng. Do đó chúng ta có thể đánh số cho các đường chéo *upward* từ **0** đến (**2xboard_size-2**), và như vậy, ô ở hàng **i** và cột **j** sẽ thuộc đường chéo *upward* có số thứ tự là **i+j**.

Bằng cách tương tự, như hình 6.11d, các đường chéo *downward* có hiệu giữa hàng và cột là một hằng số, từ (**-board_size+1**) ñeán (**board_size-1**). Các đường chéo *downward* sẽ được đánh số từ **0** đến (**2xboard_size-1**), một ô tại hàng **i** và cột **j** thuộc đường chéo *downward* có số thứ tự (**i-j+board_size-1**).

Sau khi đã có các chọn lựa trên chúng ta có định nghĩa mới cho lớp Queens như sau:

```
class Queens {
public:
   Queens(int size);
   bool is_solved() const;
   void print() const;
   bool unguarded(int col) const;
   void insert(int col);
   void remove(int col);
   int board_size;
private:
   int  count;
   bool col_free[max_board];
   bool upward_free[2 * max_board - 1];
   bool downward_free[2 * max_board - 1];
   int  queen_in_row[max_board];  // số thứ tự của cột chứa hậu trong mỗi hàng.
};
```

Chúng ta sẽ hoàn tất chương trình qua việc hiện thực các phương thức cho lớp mới. Đầu tiên là *constructor* khởi gán tất cả các trị cần thiết cho các mảng.

```
Queens::Queens(int size)
/*
post: The Queens object is set up as an empty
      configuration on a chessboard with size squares in each row and column.
*/
{
   board_size = size;
   count = 0;
   for (int i = 0; i < board_size; i++) col_free[i] = true;
   for (int j = 0; j < (2*board_size -1); j++) upward_free[j] = true;
   for (int k = 0; k < (2*board_size -1); k++) downward_free[k] = true;
}
```

Phương thức insert chỉ cần cập nhật cột và hai đường chéo đi ngang qua ô tại [count][col] là đã bị nhìn thấy bởi con hậu mới thêm vào, các trị này cũng có thể là false sẵn trước đó do chúng đã bị các con hậu trước đó nhìn thấy.

```
void Queens::insert(int col)
/*
Pre:  The square in the first unoccupied row (row count) and column col
      is not guarded by any queen.
Post: A queen has been inserted into the square at row count and column
      col; count has been incremented by 1.
*/
```

```
{
   queen_in_row[count] = col;
   col_free[col] = false;
   upward_free[count + col] = false;
   downward_free[count - col + board_size - 1] = false;
   count++;
}
```

Cuối cùng phương thức unguarded chỉ cần kiểm tra cột và hai đường chéo đi ngang qua ô tại [count][col] có bị các con hậu nhìn thấy hay chưa.

```
bool Queens::unguarded(int col) const
/*
Post: Returns true or false according as the square in the first
      unoccupied row (row count) and column col is not guarded by any queen.
*/
{
   return  col_free[col]
           && upward_free[count + col]
           && downward_free[count - col + board_size - 1];
}
```

Chúng ta thấy rằng phương thức trên đơn giản hơn trong phương án đầu tiên của nó rất nhiều. Các phương thức còn lại is_solved, remove, và print xem như bài tập. Bảng sau đây cho các con số nhận được từ chương trình cuối cùng này.

Kích thước	8	9	10	11	12	13
Số lời giải	92	352	724	2680	14200	73712
Thời gian (seconds)	0.01	0.05	0.22	1.06	5.94	34.44
Thời gian cho một lời giải (ms.)	0.11	0.14	0.30	0.39	0.42	0.47

Với trường hợp tám con hậu, chương trình mới chạy nhanh gấp 5 lần chương trình cũ. Khi kích thước bàn cờ tăng lên tỉ lệ này còn cao hơn nữa, như trường hợp 13 con hậu, chương trình mới chạy nhanh gấp 7 lần chương trình cũ.

6.3.7. Phân tích về phương pháp quay lui

Chúng ta sẽ tổng kết phần này qua việc phỏng đoán tổng số các công việc mà chương trình của chúng ta phải làm.

6.3.7.1. Tính hiệu quả của phương pháp quay lui

Chúng ta bắt đầu bằng việc tính xem việc quay lui đã nhớ được bao nhiêu công việc so với tất cả các công việc có thể có. Xét trường hợp bàn cờ 8x8, nếu chúng ta tiếp cận bài toán một cách đơn giản bằng cách viết một chương trình tìm tất cả các phương án để xếp sao cho đủ tám con hậu lên bàn cờ rồi sau đó mới loại các

trường hợp không hợp lệ, với mỗi cấu hình là một sự lựa chọn 8 vị trí trong 64 vị trí, chúng ta có số cấu hình cần khảo sát lên đến:

$$\begin{bmatrix} 64 \\ 8 \end{bmatrix} = 4{,}426{,}165{,}368.$$

Nếu chúng ta nhìn thấy được rằng mỗi hàng chỉ có thể có một con hậu thì số cấu hình cần thử giảm ngay xuống:

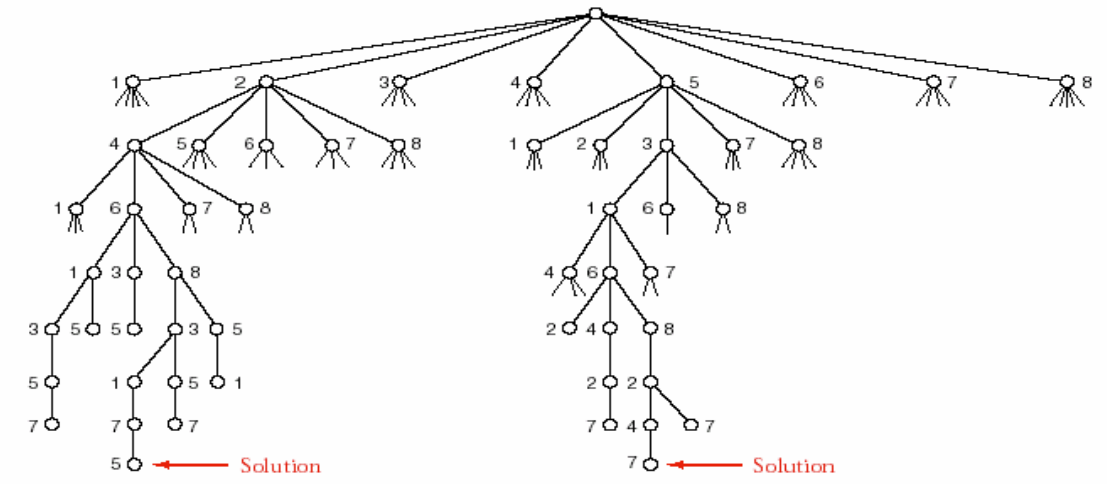
$$8^8 = 16{,}777{,}216.$$

Con số này vẫn còn rất lớn. Chúng ta tiếp tục cho rằng mỗi cột chỉ có thể có một con hậu, số lựa chọn các cột còn tự do trên mỗi hàng sẽ là 8, 7, ..., 1. Tổng số lựa chọn trên 8 hàng hoàn toàn có thể được xử lý bởi máy tính:

$$8! = 40{,}320.$$

và số trường hợp mà chương trình của chúng ta phải xem xét còn nhỏ hơn, do những ô ở hàng đang xét mà thuộc cùng đường chéo với các con hậu ở những hàng trên thì đã bị bỏ qua ngay lập tức.

Cách hoạt động của quá trình quay lui cho thấy tính hiệu quả của nó như sau: các vị trí đã được cho rằng không chấp nhận được sẽ ngăn cản sự khảo sát tiếp các đường đi vô ích sau đó.



Hình 6.12 – Một phần của cây đệ quy cho bài toán tám con hậu

Một cách khác để biểu diễn hành vi của việc quay lui là việc xem xét cây đệ quy của hàm đệ quy solve_from, hình 6.12 thể hiện một phần của cây này. Hai lời giải có trong hình tương ứng với hai lời giải trong hình 6.9. Mỗi nút trong cây có thể có tối đa là tám nút con tương ứng tám lần gọi đệ quy hàm `solve_from` với

tám trị hợp lệ của `new_col`. Tuy nhiên, ngay cả tại các mức gần với nút gốc, phần lớn các nhánh này đều được xác định sớm là không thỏa, và hiển nhiên rằng cứ mỗi nút cha không thỏa điều kiện của bài toán thì trong cây cũng không xuất hiện tiếp các nút con của nó. Phương pháp quay lui là một công cụ vô cùng hiệu quả để thu giảm một cây đệ quy về một kích thước có thể xử lý được.

6.3.7.2. Các cận dưới

Với bài toán n con hậu, tổng số công việc cần được thực hiệc bởi phương pháp quay lui tăng rất nhanh theo n. Chúng ta thử hình dung tốc độ tăng trưởng này nhanh đến cỡ nào. Khi đặt một con hậu vào một hàng trên bàn cờ, nó sẽ loại trừ nhiều nhất là ba vị trí ở hàng dưới (một vị trí cùng cột và hai vị trí chéo). Đối với hàng thứ nhất, việc quay lui sẽ khảo sát tất cả n vị trí. Ở hàng thứ hai, có ít nhất n-3 vị trí cần khảo sát; hàng thứ ba là n-6, và cứ thế. Vì vậy, để đặt các con hậu lên n/4 hàng đầu tiên, việc quay lui cần khảo sát ít nhất số vị trí:

$$n(n-3)(n-6)...(n-3n/4)$$

Tích này > $(n/4)^{n/4}$ do có tất cả n/4 thừa số và thừa số cuối cùng là n/4, các thừa số khác đều lớn hơn n/4. Để hình dung con số này tăng nhanh theo n như thế nào, chúng ta nhớ lại rằng bài toán tháp Hà Nội cần có 2^n bước cho n đĩa, mà $(n/4)^{n/4}$ còn tăng nhanh hơn 2^n nhiều khi n tăng. Để thấy được điều này, chúng ta có:

$$\frac{\log((n/4)^{n/4})}{\log(2^n)} = \frac{\log(n/4)}{4\log(2)}$$

Tỉ lệ này tăng không giới hạn khi n tăng. Chúng ta nói 2^n tăng theo hàm mũ, còn $(n/4)^{n/4}$ còn tăng nhanh hơn rất nhiều. Vậy với n lớn, chương trình áp dụng phương pháp quay lui cho bài toán tháp Hà Nội trên cũng sẽ chạy rất chậm.

6.3.7.3. Số lời giải

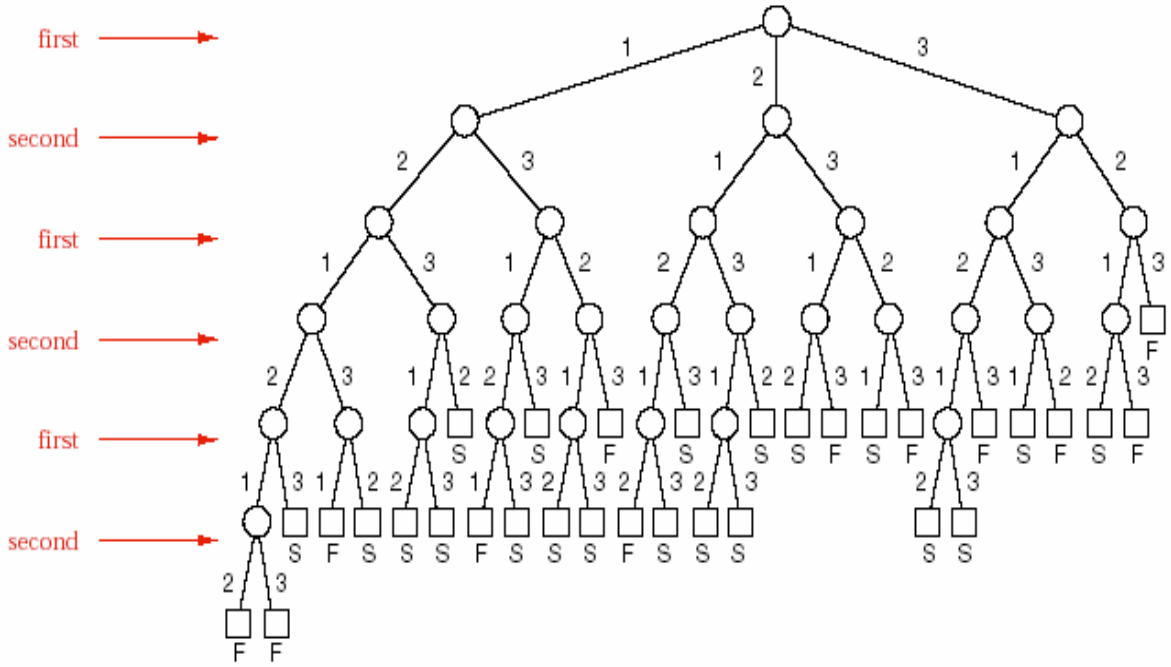
Chúng ta vẫn chưa chứng minh rằng việc in mọi lời giải cho bài toán n con hậu với n lớn là không thể thực hiện được bằng máy tính, mà chỉ mới chỉ ra rằng phương pháp quay lui là không thể làm được. Có thể còn một số giải thuật thông minh hơn nào đó có thể in các lời giải một cách nhanh chóng hơn giải thuật quay lui. Tuy nhiên, điều chúng ta quan tâm ở đây không phải là khả năng của máy tính mà là số lời giải thực sự của bài toán n con hậu. Điều đã có thể chứng minh được là số lời giải của bài toán này không thể được giới hạn bởi bất kỳ một đa thức bậc n nào. Thậm chí số lời giải này dường như còn không thể bị giới hạn bởi một biểu thức hàm mũ k^n, với k là một hằng số, nhưng việc chứng minh điều này vẫn còn là một bài toán chưa có lời giải.

6.4. Các chương trình có cấu trúc cây: dự đoán trước trong các trò chơi

Trong các trò chơi trí tuệ, con người có thể dự đoán trước một số bước. Trong phần này chúng ta phát triển một giải thuật cho máy tính để tham gia các trò chơi có khảo sát trước một số bước đi. Chúng ta sẽ trình bày giải thuật qua hình ảnh của một cây và sử dụng đệ quy để lập trình cho cấu trúc này.

6.4.1. Các cây trò chơi

Chúng ta có thể vẽ ra các bước di chuyển có thể có qua hình ảnh của một cây trò chơi, trong đó gốc cây là trạng thái ban đầu, các cành xuất phát từ gốc là các bước đi hợp lệ của người chơi thứ nhất. Ở mức kế tiếp, các cành lại biểu diễn các bước đi hợp lệ của người chơi thứ hai tương ứng với mỗi cách đi của người thứ nhất, và cứ như thế. Nếu cho rằng mức của gốc là 0 thì các cành xuất phát từ các nút có mức chẵn biểu diễn bước đi của người thứ nhất, các cành xuất phát từ các nút có mức lẻ biểu diễn bước đi của người thứ hai.



Hình 6.13 – Cây cho trò chơi số tám

Hình 6.13 là cây trò chơi đầy đủ của trò chơi Số tám. Trong trò chơi này, người chơi thứ nhất sẽ lựa chọn một trong các số 1, 2, hoặc 3. Mỗi bước sau đó, mỗi người chơi sẽ được quyền chọn một trong ba số 1, 2, hoặc 3, nhưng không được trùng với số mà người kia vừa chọn xong. Các cành trong cây mang con số do người chơi chọn. Tổng các số được chọn sẽ được cộng tích lũy dần. Nếu đến lượt mình, người chơi chọn được con số làm cho tổng này đúng bằng tám thì sẽ là người thắng cuộc; nếu tổng vượt quá tám thì người còn lại sẽ thắng. Trò chơi này

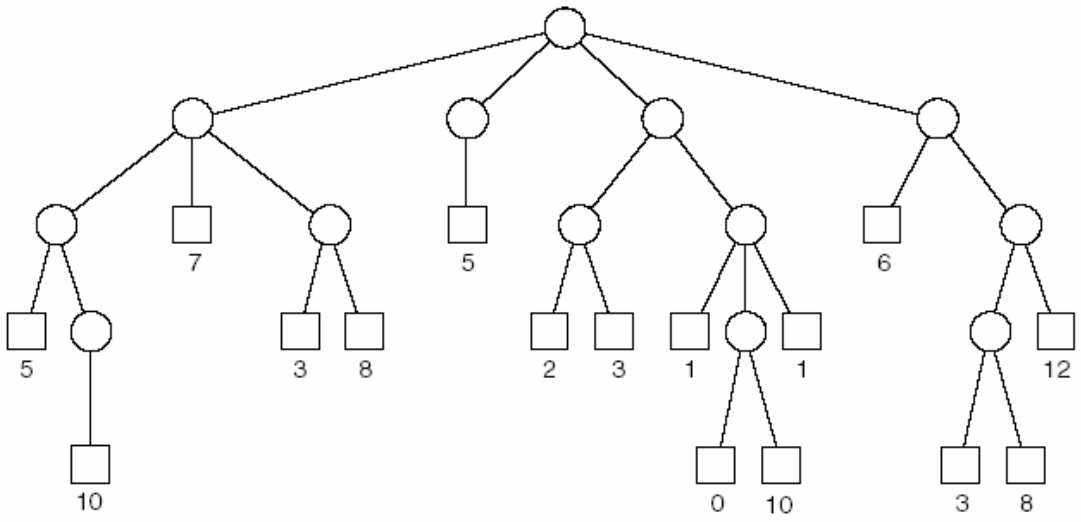
không có khả năng hòa. Trong sơ đồ, F có nghĩa là người thứ nhất thắng cuộc, và S là người thứ hai thắng cuộc.

Ngay một trò chơi tầm thường như trò chơi Số tám trên đây cũng đã sinh ra một cây có kích thước không nhỏ. Các trò chơi thực sự hấp dẫn tựa như cờ vua còn có cây trò chơi lớn hơn rất nhiều, và chúng ta cũng không hy vọng gì vào việc có thể khảo sát hết các cành của nó. Như vậy, một chương trình chạy trong một thời gian cho phép chỉ có thể khảo sát một vài mức dưới một nút hiện tại của cây. Con người khi chơi những trò chơi này cũng không thể nhìn thấy được mọi khả năng phát triển cây cho đến khi trò chơi kết thúc, nhưng họ có thể có một số lựa chọn thông minh, bởi vì, theo kinh nghiệm, thông thường một người có thể nhận biết ngay một vài tình huống này là tốt hơn so với các tình huống khác, mặc dù họ cũng không bảo đảm được là sẽ thắng.

Đối với mọi trò chơi hấp dẫn mà chúng ta muốn chơi bằng máy tính, chúng ta cần một hàm gọi là hàm lượng giá để kiểm tra một tình huống hiện tại về mức độ thuận lợi của nó.

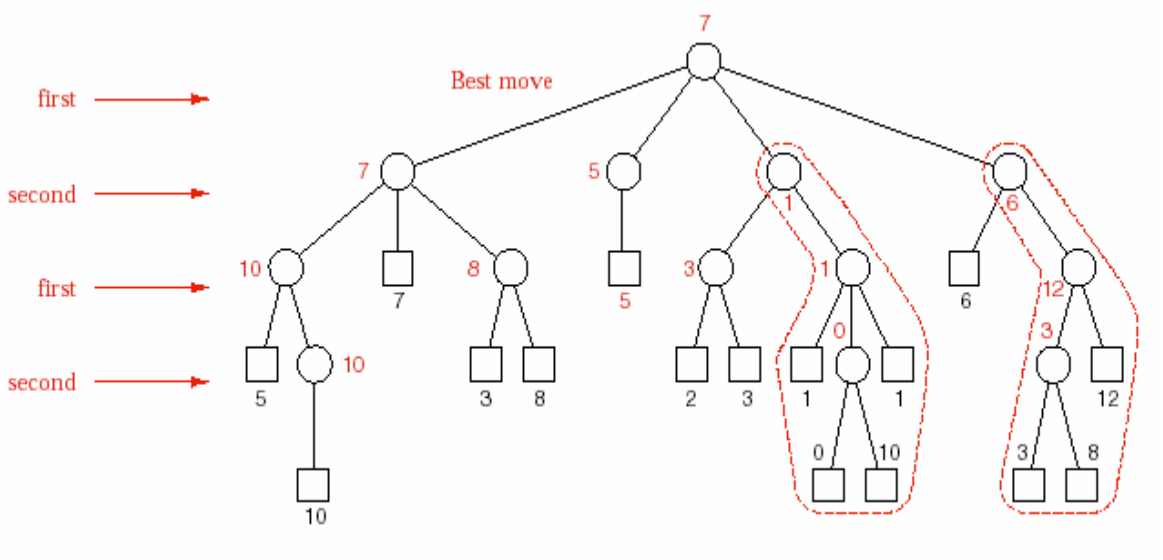
6.4.2. Phương pháp Minimax

Hình 6.14 là một cây con biểu diễn một phần của một cây trò chơi với mục đích minh họa cho bất kỳ trò chơi nào. Gốc cây con này biểu diễn vị trí hiện tại mà chúng ta đang muốn nhìn trước các bước đi. Chúng ta chỉ cần hàm lượng giá tại các nút lá của cây con này (đó là những vị trí mà chúng ta sẽ không nhìn tới trước xa hơn nữa), và từ các thông tin này, chúng ta phải chọn ra một cách đi cho vị trí hiện tại. Các nút lá trong cây được vẽ bằng hình vuông để phân biệt với các nút khác. Hình 6.14 chứa các trị cho các nút lá.



Hình 6.14 – Cây trò chơi với các trị được gán ở các nút lá

Bước đi mà chúng ta chọn sẽ là một trong các cành xuất phát từ gốc cây. Chúng ta dùng hàm lượng giá theo cách nhìn của người chơi sẽ thực hiện bước đi này. Giả sử gọi đó là người thứ nhất, người này sẽ chọn con số lớn nhất có thể. Tại bước kế, người chơi thứ hai lại chọn con số nhỏ nhất có thể, và cứ thế tiếp tục. Do hàm lượng giá được tính theo các tiêu chí dành cho người thứ nhất, nên giá trị nhỏ nhất của nó tương ứng tình huống xấu nhất của người thứ nhất. Giá trị này luôn được người thứ hai chọn vì theo khuynh hướng tình huống xấu nhất đối với người thứ nhất chính là tình huống tốt nhất của người thứ hai.Bằng cách đi ngược từ các nút lá lên, chúng ta có thể gán các trị cho mọi nút trong cây. Chúng ta hãy thực hiện việc này cho một phần của cây trong hình 6.14, bắt đầu từ cành bên trái nhất của cây. Nút đầu tiên chưa có nhãn là nút tròn nằm ngay trên nút hình vuông mang số 10. Do không có sự lựa chọn nào khác từ nút này nên nó cũng có số 10. Nút cha của nó có hai nút con mang số 5 và 10. Nút cha này thuộc mức thứ ba trong cây nên biểu diễn bước đi của người thứ nhất, đó là người ưu tiên chọn số lớn. Vậy người này đã chọn 10 nên nút cha này cũng sẽ mang trị 10.



Hình 6.15 – Lượng giá theo phương pháp Minimax cho cây trò chơi

Chúng ta tiếp tục chuyển lên mức trên. Ở đây nút cha có ba nút con, con thứ nhất từ trái sang là 10, con thứ hai là 7, con thứ ba là số lớn được chọn ra từ 3 và 8 nên là 8. Đây là mức chơi của người thứ hai, do đó trị nhỏ nhất trong ba trị trên được chọn, đó là 7. Cứ như thế, nếu tiếp tục chúng ta sẽ có được cây ở hình 6.15. Trị của vị trí hiện tại này là 7, và người chơi hiện tại, là người chơi thứ nhất theo hình vẽ, sẽ chọn cành bên trái nhất của vị trí hiện tại này.

Trong việc đánh giá cây trò chơi này, chúng ta đã luân phiên chọn số nhỏ nhất (*minima*) và số lớn nhất (*maxima*). Do đó quá trình này còn gọi là phương pháp *minimax*.

6.4.3. Phát triển giải thuật

Chúng ta sẽ xem xét bằng cách nào mà phương pháp *minimax* có thể được lồng trong một giải thuật hình thức để dự đoán trước trong các chương trình trò chơi. Chúng ta sẽ viết một giải thuật tổng quát để có thể sử dụng trong bất kỳ một trò chơi nào có hai người chơi.

Chương trình sẽ cần truy xuất đến thông tin về một trò chơi cụ thể nào đó mà chúng ta muốn chơi. Chúng ta giả sử rằng thông tin này được tập hợp trong hiện thực của hai lớp gọi là **`Move`** và **`Board`**. Một đối tượng của lớp **`Move`** biểu diễn **một bước đi của trò chơi**, và một đối tượng của lớp **`Board`** biểu diễn **một tình huống của trò chơi**. Sau này chúng ta sẽ hiện thực các phiên bản của hai lớp trên đây cho trò chơi *Tic-Tac-Toe*.

Đối với lớp **`Move`**, chúng ta chỉ cần phương thức *constructor:* một *constructor* để tạo đối tượng `Move` theo ý của người chơi, và một *constructor* khác để tạo một đối tượng `Move` rỗng. Chúng ta cũng giả sử rằng các đối tượng của lớp `Move` và lớp `Board` đều có thể được gởi bằng tham trị cho hàm cũng như có thể được sử dụng phép gán một cách an toàn, do đó chúng ta cần viết đầy đủ toán tử gán định nghĩa lại (*overloaded assignment operator*) và *copy constructor* cho cả hai lớp.

Đối với lớp **`Board`**, các phương thức cần có là: khởi tạo đối tượng, kiểm tra trò chơi đã kết thúc hay chưa, tiến hành một bước đi nhận được qua tham trị, đánh giá một tình huống, và cung cấp một danh sách các nước đi hợp lệ hiện tại.

Phương thức **`legal_moves`**, trả về các nước đi hợp lệ hiện tại, sẽ cần một danh sách các thông số để tính toán các kết quả. Chúng ta cần một sự lựa chọn giữa các hiện thực của cấu trúc dữ liệu list để chứa các cách đi này. Trong việc nhìn tới trước, thứ tự giữa các cách đi mà chúng ta sẽ khảo sát là không quan trọng, do đó chúng có thể được lưu trong bất kỳ dạng nào của list. Để đơn giản cho việc lập trình, chúng ta sẽ sử dụng ngăn xếp. Phần tử của ngăn xếp sẽ là các đối tượng `Move`. Chúng ta cần định nghĩa:

```
typedef Move Stack_entry;
```

Chúng ta cần thêm hai phương thức để hỗ trợ trong việc lựa chọn nước đi tốt nhất trong số các nước đi hợp lệ. Phương thức đầu tiên, gọi là **`better`**, sử dụng hai thông số là hai số nguyên và trả về một kết quả khác 0 nếu người chơi chọn nước đi theo thông số thứ nhất, hoặc bằng 0 nếu người chơi chọn nước đi theo thông số thứ hai. Phương thức thứ hai, gọi là **`worst_case`**, trả về một hằng số được xác định trước có trị thấp hơn tất cả các trị mà hàm lượng giá tính được trong quá trình nhìn trước.

Chúng ta có định nghĩa của lớp Board như sau:

```
class Board {
public:
   Board();//constructor khởi tạo trạng thái ban đầu thích hợp đối với mỗi trò chơi.
   int done()  const;            //  kiểm tra xem trò chơi đã kết thúc hay chưa.
   void play(Move try_it);
   int evaluate()  const;
   int legal_moves(Stack &moves)  const;
   int worst_case()  const;
   int better(int value, int old_value) const; // chọn nước đi tốt nhất.
   void print()  const;
   void instructions()  const;
/*  Các phương thức, hàm và dữ liệu bổ sung tùy từng trò chơi cụ thể */
};
```

Đối tượng Board cần lưu một tình huống của trò chơi và người mà sắp thực hiện bước đi.

Trước khi viết hàm nhìn trước để đánh giá cây trò chơi, chúng ta cần chọn ra số bước mà giải thuật nhìn trước sẽ phải khảo sát. Đối với một trò chơi tương đối phức tạp, chúng ta cần định ra độ sâu (**depth**) sẽ được nhìn trước, các mức bên dưới nữa sẽ không được xét đến. Một điều kiện dừng khác của việc nhìn trước chính là khi trò chơi kết thúc, đó là lúc mà phương thức **done** của Board trả về true. Nhiệm vụ chính của việc nhìn trước trong cây có thể được mô tả bởi giải thuật đệ quy sau đây:

Algorithm look_ahead (thông số là một đối tượng **Board**);
1. **if** đệ quy dừng (độ sâu **depth** == 0 hoặc game.done())
    1. **return** trị lượng giá của tình huống
2. **else**
    1. **for** mỗi nước đi **Move** hợp lệ
        1. tạo một đối tượng **Board** mới bằng cách thực hiện nước đi **Move**.
        2. gọi đệ quy **look_ahead** tương ứng với sự lựa chọn tốt nhất của người chơi kế tiếp;
    2. Chọn cách đi tốt nhất cho người chơi trong số các cách đi tìm được trong vòng lặp trên;
3. **return** đối tượng **Move** tương ứng và trị;

6.4.4. Tinh chế

Chương trình cụ thể của giải thuật trên như sau:

```
int look_ahead(const Board &game, int depth, Move &recommended)
/*
pre:   đối tượng Board biểu diễn một tình huống hợp lệ của trò chơi.
post:  các nước đi của trò chơi được nhìn trước với độ sâu là depth, nước đi tốt nhất được chỉ ra
       trong tham biến recommended.
uses:  các lớp Stack, Board, và Move, cùng với hàm look_ahead một cách đệ quy.
*/
```

```
{
   if (game.done() || depth == 0)
      return game.evaluate();
   else {
      Stack moves;
      game.legal_moves(moves);
      int value, best_value = game.worst_case();

      while (!moves.empty()) {
         Move try_it, reply;
         moves.top(try_it);
         Board new_game = game;
         new_game.play(try_it);
         value = look_ahead(new_game, depth - 1, reply);
         if (game.better(value, best_value)) { //  nước đi thử try_it vừa rồi hiện
                                                   là tốt nhất.
            best_value = value;
            recommended = try_it;
         }
         moves.pop();
      }
      return best_value;
   }
}
```

Tham biến **recommended** sẽ nhận về một nước đi được tiến cử (trừ khi đệ quy rơi vào điểm dừng, đó là khi trò chơi kết thúc hoặc độ sâu của việc nhìn trước **depth** bằng 0). Do chúng ta không muốn đối tượng **game** bị thay đổi trong hàm, đồng thời để tránh việc chép lại mất thời gian, nó được gởi cho hàm bằng tham chiếu hằng. Chúng ta cũng lưu ý rằng việc khai báo tham chiếu hằng này chỉ hợp lệ khi đối tượng **game** trong hàm chỉ thực hiện các phương thức đã được khai báo const trong định nghĩa của lớp **Board**.

6.4.5. Tic-Tac-Toe

Như trên đã nói, hàm đệ quy **look_ahead** cùng hai lớp **Board** và **Move** trên đây tuy rất đơn giản, nhưng nó lại là cốt lõi trong các chương trình biểu diễn trò chơi có hai đối thủ. Chúng ta sẽ triển khai phác thảo này trong trò chơi *tic-tac-toe* rất quen thuộc. Để làm được điều này, cả hai lớp sẽ chứa thêm một ít dữ liệu khác so với hiện thực hình thức ban đầu.

Việc viết chương trình chính hoàn tất cùng hàm look_ahead cho trò chơi *tic-tac-toe* được dành lại như bài tập. Chương trình có thể chứa thêm nhiều chức năng như cho phép người chơi với máy, đưa ra các phân tích đầy đủ cho mỗi tình huống, cung cấp chức năng cho hai người chơi với nhau, đánh giá các bước đi của hai đối thủ,...

Chúng ta biểu diễn lưới trò chơi *tic-tac-toe* bằng một mảng 3x3 các số nguyên, ô có trị 0 là ô trống, trị 1 và 2 biểu diễn nước đi của người thứ nhất và thứ hai tương ứng.

Trong đối tượng **Move**, chúng ta chứa tọa độ các ô trên lưới. Một nước đi hợp lệ chứa tọa độ có các trị từ 0 đến 2. Chúng ta không cần đến tính đóng kín của đối tượng **Move** vì nó chỉ như một vật chứa để chứa các giá trị.

```
//  lớp move cho trò chơi tic-tac-toe
class Move {
public:
   Move();
   Move(int r, int c);
   int row;
   int col;
};
```

```
Move::Move()
/*
post: đối tượng Move được khởi tạo bởi trị mặc định không hợp lệ.
*/
{
   row = 3;
   col = 3;
}
```

```
Move::Move(int r, int c)
/*
post: đối tượng Move được khởi tạo bởi toạ độ cho trước.
*/
{
   row = r;
   col = c;
}
```

Lớp **Board** cần một *constructor* để khởi tạo trò chơi, phương thức **print** và **instruction** (in các thông tin cho người chơi), phương thức **done**, **play** và **legal_moves** (hiện thực các quy tắc chơi), và các phương thức **evaluate**, **better**, và **worst_case** (phán đoán điểm cho các nước đi khác nhau). Chúng ta còn có thể bổ sung hàm phụ trợ **the_winner** cho biết rằng trò chơi đã có người thắng chưa và người thắng là ai.

Lớp **Board** cần chứa thông tin về trạng thái hiện tại của trò chơi trong mảng 3x3 và tổng số bước đi đã thực hiện. Chúng ta có lớp **Board** như sau:

```
class Board {
public:
   Board();
   bool done() const;
   void print() const;
   void instructions() const;
   bool better(int value, int old_value) const;
   void play(Move try_it);
   int worst_case() const;
```

```
   int evaluate() const;
   int legal_moves(Stack &moves) const;
private:
   int squares[3][3];
   int moves_done;
   int the_winner() const;
};
```

Constructor đơn giản chỉ làm một việc là khởi tạo tất cả các ô của mảng là 0 để chỉ rằng cả hai người chơi đều chưa đi nước nào.

```
Board::Board()
/*
post: đối tượng Board được khởi tạo rỗng tương ứng trạng thái ban đầu của trò chơi.
*/
{
   for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
         squares[i][j] = 0;
   moves_done = 0;
}
```

Chúng ta dành các phương thức in các thông tin cho người chơi như là bài tập. Thay vào đó, chúng ta tập trung vào các phương thức liên quan đến các quy tắc của trò chơi. Để thực hiện một nước đi, chúng ta chỉ cần gán lại trị cho một ô và tăng biến đếm **moves_done** lên 1. Dựa vào biến đếm này chúng ta còn biết được đến lượt người nào đi.

```
void Board::play(Move try_it)
/*
post: nước đi try_it được thực hiện
*/
{
   squares[try_it.row][try_it.col] = moves_done % 2 + 1;
   moves_done++;
}
```

Hàm phụ trợ **the_winner** trả về một số khác không nếu đã có người thắng.

```
int Board::the_winner() const
/*
post: trả về 0 nếu chưa ai thắng; 1 nếu người thứ nhất thắng; 2 nếu người thứ hai thắng.
*/
{
   int i;
   for (i = 0; i < 3; i++)
      if (squares[i][0] && squares[i][0] == squares[i][1]
                        && squares[i][0] == squares[i][2])
            return squares[i][0];

   for (i = 0; i < 3; i++)
      if (squares[0][i] && squares[0][i] == squares[1][i]
                        && squares[0][i] == squares[2][i])
            return squares[0][i];
```

```
   if (squares[0][0] && squares[0][0] == squares[1][1]
                     && squares[0][0] == squares[2][2])
            return squares[0][0];

   if (squares[0][2] && squares[0][2] == squares[1][1]
                     && squares[2][0] == squares[0][2])
            return squares[0][2];
   return 0;
}
```

Trò chơi kết thúc sau 9 nước đi hoặc khi có người thắng. (Chương trình của chúng ta không phát hiện sớm khả năng hòa trước khi kết thúc 9 nước đi).

```
bool Board::done() const
/*
post: trả về true nếu trò chơi đã phân thắng bại hoặc cả 9 ô trên bàn cờ đã được đánh dấu,
      ngược lại trả về false.
*/
{
   return moves_done == 9 || the_winner() > 0;
}
```

Trong trò chơi đơn giản này, nước đi hợp lệ là những ô mang trị 0.

```
int Board::legal_moves(Stack &moves) const
/*
post: Thông số moves chứa các nước đi hợp lệ.
*/
{
   int count = 0;
   while (!moves.empty()) moves.pop();
   for (int i = 0; i < 3; i++)
      for (int j = 0; j < 3; j++)
         if (squares[i][j] == 0) {
            Move can_play(i,j);
            moves.push(can_play);
            count++;
         }
   return count;
}
```

Chúng ta chuyển sang các phương thức có thể cho những sự đánh giá về một tình huống của trò chơi hoặc của một nước đi có thể xảy ra. Chúng ta sẽ bắt đầu đánh giá một tình huống của trò chơi là 0 trong trường hợp chưa có người nào thắng. Nếu một trong hai người thắng, chúng ta sẽ đánh giá tình huống dựa vào quy tắc: càng thắng nhanh thì càng hay. Điều này cũng đồng nghĩa với việc càng thua nhanh thì càng dở. Nếu moves_done là số nước đi cả hai người chơi đã thực hiện thì (10-moves_done) càng lớn khi số nước đã đi càng nhỏ, tương ứng sự đánh giá cao khi người thứ nhất sớm thắng. Ngược lại, nếu người thứ hai thắng thì chúng ta dùng trị (moves_done-10), số nước đi càng nhỏ thì trị này là một số càng âm, tương ứng việc thua nhanh chóng của người thứ nhất là rất dở.

Dĩ nhiên rằng, cách đánh giá này chỉ được áp lên điểm cuối của việc nhìn trước (ở độ sâu mong muốn hoặc khi trò chơi đã phân thắng bại). Trong phần lớn các tình huống, nếu chúng ta nhìn trước càng xa thì bản chất chưa tinh của cách đánh giá sẽ đỡ gây hậu quả xấu hơn.

```
int Board::evaluate() const
/*
post: trả về 0 khi chưa có người thắng cuộc; hoặc 1 số dương từ 1 đến 9 trong trường hợp người
      thứ nhất thắng, ngược lại là một số âm từ –1 đến –9 trong trường hợp người thứ hai
      thắng.
*/
{
   int winner = the_winner();
   if (winner == 1) return 10 - moves_done;
   else if (winner == 2) return moves_done - 10;
   else return 0;
}
```

Phương thức **worst_case** có thể chỉ đơn giản trả về một trong hai trị 10 hoặc -10, do **evaluate** luôn trả một trị nằm giữa -9 và 9. Từ đó, phương thức so sánh **better** chỉ cần so sánh một cặp số nguyên có trị nằm giữa -10 và 10. Các phương thức này xem như bài tập.

Giờ thì chúng ta đã phác thảo xong phần lớn chương trình trò chơi *tic-tac-toe*. Một chương trình lấy độ sâu cho việc nhìn trước là 9 sẽ chơi tốt do chúng ta luôn có thể nhìn trước đến một tình huống mà sự đánh giá về nó là chính xác. Một chương trình nhìn trước với một độ sâu không lớn có thể mắc phải sai lầm, do nó có thể kết thúc việc nhìn trước bởi tập các tình huống có trị đánh giá bằng 0 một cách nhầm lẫn.

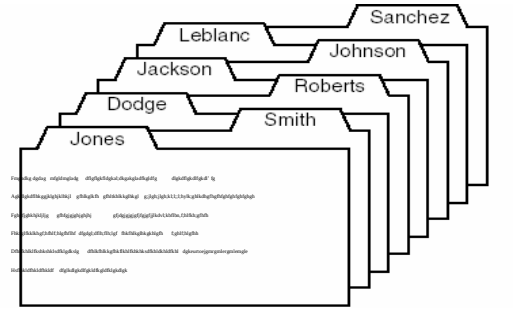
Chương 7 – TÌM KIẾM

Chương này giới thiệu bài toán tìm kiếm một phần tử trong một danh sách. Phần trình bày tập trung chủ yếu vào hai giải thuật: tìm kiếm tuần tự và tìm kiếm nhị phân.

7.1. Giới thiệu

7.1.1. Khóa

Trong bài toán tìm kiếm, dựa vào một phần thông tin được gọi là khoá (*key*), chúng ta phải tìm một mẫu tin (*record*) chứa các thông tin khác liên quan với khoá này. Có thể có nhiều mẫu tin hoặc không có mẫu tin nào chứa khoá cần tìm.



Hình 7.1. Mẫu tin và khoá.

7.1.2. Phân tích

Tìm kiếm thông thường là tác vụ tốn nhiều thời gian trong một chương trình. Vì thế việc tổ chức cấu trúc dữ liệu và giải thuật cho việc tìm kiếm có thể có những ảnh hưởng lớn đến hiệu suất hoạt động của chương trình. Ơ đây, thông số đo chủ yếu là số lần so sánh khoá cần tìm với các mẩu tin khác.

7.1.3. Tìm kiếm nội và tìm kiếm ngoại

Bài toán tìm kiếm bao gồm hai nhóm: tìm kiếm nội và tìm kiếm ngoại. Nếu lượng dữ liệu lớn phải lưu trên thiết bị lưu trữ ngoài như đĩa hay băng từ thì bài toán được gọi là tìm kiếm ngoại. Ngược lại nếu toàn bộ dữ liệu được lưu trữ trên bộ nhớ chính thì được gọi là tìm kiếm nội. Ơ đây ta quan tâm chủ yếu đến tìm kiếm nội.

Giải thuật tìm kiếm trên các cấu trúc liên kết hoàn toàn phụ thuộc vào cách tổ chức đặc trưng của chúng. Danh sách liên kết đơn là cấu trúc liên kết đơn giản nhất, việc tìm kiếm chỉ có thể duyệt tuần tự qua từng phần tử mà thôi. Đối với các cấu trúc liên kết khác, chúng ta sẽ có dịp tìm hiểu các chiến lược tìm kiếm khác nhau khi gặp từng cấu trúc cụ thể, chẳng hạn như cây nhị phân tìm kiếm, cây `B-tree`, hàng ưu tiên,.... Có một cấu trúc dữ liệu khá đặc biệt đối với việc tìm kiếm, đó là bảng băm. Ý tưởng cơ bản và đặc biệt nhất của bảng băm làm cho nó

khác với các cấu trúc dữ liệu khác ở chỗ, trong bảng băm không có khái niệm duyệt qua các phần tử trước khi đến được phần tử mong muốn. Chúng ta cũng sẽ được học về bảng băm trong chương 12.

Chương này chỉ trình bày những ý tưởng cơ bản và đơn giản nhất của việc tìm kiếm. Trong đó, giả sử rằng khi cần truy xuất một phần tử bất kỳ nào đó chúng ta có thể nhảy ngay đến vị trí của nó trong danh sách với thời gian là hằng số. Điều này chỉ có thể đạt được khi **các phần tử được lưu trong danh sách liên tục**. Và như vậy, trong chương này các giải thuật tìm kiếm rõ ràng chỉ phụ thuộc vào số lần so sánh khóa, chứ không phụ thuộc vào thời gian di chuyển qua các phần tử.

Cách hiện thực của các phương thức bổ sung cũng như các giải thuật tìm kiếm dưới đây hoàn toàn sử dụng các phương thức có sẵn của lớp `List` trong chương 4**.** Chúng ta nên có một số nhận xét như sau. Thứ nhất, cách sử dụng các phương thức có sẵn của lớp `List` không ngăn cấm chúng ta việc sử dụng hiện thực danh sách liên kết thay vì danh sách liên tục. Đối với danh sách liên kết cần phải chi phí trong khi truy xuất phần tử tại vị trí `position` nào đó (điều này vẫn còn điểm khác nhau giữa hai phương án của danh sách liên kết có hoặc không có lưu lại thuộc tính `current_position`). Đối với danh sách liên tục, có thể trực tiếp truy xuất một phần tử thông qua một số nguyên chỉ vị trí của nó, thay vì gọi phương thức có sẵn `retrieve`.

7.1.4. Lớp `Record` và lớp `Key`

Chúng ta có một số quy ước như sau. Các phần tử trong danh sách đang được tìm kiếm thoả các tiêu chuẩn tối thiểu sau:

- Mỗi mẫu tin có một khoá đi kèm.
- Các khóa có thể được so sánh với nhau bằng các toán tử so sánh.
- Một mẩu tin có thể được chuyển đổi tự động thành một khóa. Do đó có thể so sánh các mẫu tin với nhau hoặc so sánh mẫu tin với khoá thông qua việc việc chuyển đổi mẫu tin về khóa liên quan đến nó.

Chúng ta sẽ cài đặt các chương trình tìm kiếm làm việc với các đối tượng thuộc lớp **`Record`** thoả các điều kiện trên. Ngoài ra còn có một lớp **`Key`** (có thể trùng với `Record`) và một tác vụ để chuyển đổi một `Record` thành một `Key`. Tác vụ đó có thể được cài đặt theo một trong hai cách sau:

- Một phương thức của lớp `Record` có khai báo là **`operator Key() const`**;
- Một *constructor* của lớp `Key` có khai báo là **`Key(const Record&);`**

Nếu `Record` và `Key` là giống nhau thì không cần tác vụ này.

Trên lớp Key chúng ta cần phải định nghĩa các phép so sánh ==, !=, <, >, <=, >= mọi cặp đối tượng thuộc lớp Key. Do mọi Record đều có thể được chuyển đổi thành Key nhờ trình biên dịch bằng một trong các tác vụ trên, các tác vụ so sánh Key đều có thể được sử dụng để so sánh hai Record hay so sánh một Record với một Key.

```
// Khai báo cho lớp Key
class Key{
   public:
      //  Các constructor và các phương thức.
   private:
      //  Các thuộc tính của Key.
};

//  Khai báo các tác vụ so sánh cho khoá.
  bool operator ==(const Key &x, const Key &y);
  bool operator > (const Key &x, const Key &y);
  bool operator < (const Key &x, const Key &y);
  bool operator >=(const Key &x, const Key &y);
  bool operator <=(const Key &x, const Key &y);
  bool operator !=(const Key &x, const Key &y);

// Khai báo cho lớp Record
class Record{
  public:
    operator Key(); //  Chuyển đổi từ Record sang Key.
     //  Các constructor và các phương thức của Record.
private:
  // Các thuộc tính của Record
};
```

7.1.5. Thông số

Các hàm tìm kiếm sẽ nhận hai tham trị. Tham trị thứ nhất là danh sách cần tìm, tham trị thứ hai là phần tử cần tìm. Thông số thứ hai được gọi là đích của phép tìm kiếm. Trị trả về của hàm có kiểu là ErrorCode cho biết việc tìm kiếm có thành công hay không. Nếu tìm thấy thì tham biến **position** chứa vị trí tìm thấy phần tử liên quan đến khóa cần tìm trong danh sách.

7.2. Tìm kiếm tuần tự

7.2.1. Giải thuật và hàm

Phương pháp đơn giản nhất để tìm kiếm là xuất phát từ một đầu của danh sách và tìm dọc theo danh sách cho đến khi gặp được phần tử cần tìm hay đến khi hết danh sách. Đây là giải thuật được sử dụng trong hàm sau.

```
ErrorCode sequential_search(const List<Record> &the_list,
                            const Key &target, int &position)
/*
post: Nếu có phần tử trong danh sách có khóa trùng với target, hàm trả về success và
      tham biến position chứa vị trí của phần tử được tìm thấy trongt danh sách. Ngược lại
      hàm trả về not_present và position không có nghĩa.
*/
{
   int s = the_list.size();
   for (position = 0; position < s; position++) {
      Record data;
      the_list.retrieve(position, data);
      if (data == target) return success;
   }
   return not_present;
}
```

Vòng lặp `for` trong hàm này duyệt danh sách cho đến khi gặp phần tử cần tìm hoặc đến khi hết danh sách. Nếu gặp phần tử cần tìm thì giải thuật kết thúc ngay lập tức và `position` chứa vị trí phần tử tìm được, ngược lại nếu không tìm thấy thì hàm trả về `not_present` và `position` chứ vị trí không hợp lệ.

7.2.2. Phân tích

Sau đây chúng ta sẽ đánh giá khối lượng công việc mà giải thuật tìm kiếm tuần tự thực hiện để làm cơ sở so sánh với các phương pháp khác sau này.

Giả sử giải thuật tìm kiếm tuần tự được thực thi trên một danh sách dài. Các lệnh ngoài vòng `for` được thực hiện một lần, do đó không ảnh hưởng nhiều đến thời gian chạy giải thuật. Trong mỗi lần lặp, một khoá của một mẫu tin được so sánh với khoá đích. Ngoài ra còn có một số tác vụ khác cũng được thực hiện một lần cho mỗi lần lặp.

Như vậy các tác vụ mà ta cần quan tâm có liên hệ trực tiếp với số lần so sánh khoá. Những cách lập trình khác nhau của cùng một giải thuật có thể cho ra các số lượng công việc khác nhau nhưng đều cho cùng một số lần so sánh. Khi chiều dài của danh sách thay đổi thì số lượng công việc cũng thay đổi theo một cách tương ứng.

Ơ đây chúng ta sẽ tìm hiểu sự phụ thuộc của số lần so sánh khoá vào độ dài của danh sách. Đây là thông tin hữu ích nhất trong việc tìm hiểu giải thuật tìm kiếm này, nó không phụ thuộc vào cách thức lập trình cụ thể cũng như loại máy tính cụ thể đang được sử dụng. Việc phân tích các giải thuật tìm kiếm được dựa trên giả thiết căn bản là: khối lượng công việc mà một giải thuật tìm kiếm thực hiện (hay thời gian chạy của giải thuật) được phản ảnh bởi tổng số lần so sánh khoá mà giải thuật thực hiện.

Chúng ta hãy tìm số lần so sánh khoá mà giải thuật tìm kiếm tuần tự cần khi nó chạy trên một danh sách gồm n phần tử. Do giải thuật tìm kiếm tuần tự lần lượt so sánh khoá đích với từng khoá của các phần tử trong danh sách nên tổng số lần so sánh phụ thuộc vào vị trí của đích trong danh sách. Nếu đích là phần tử đầu tiên của danh sách thì chỉ cần một lần so sánh. Nếu đích là phần tử cuối cùng của danh sách thì cần n lần so sánh. Nếu phép tìm kiếm không thành công (không có phần tử đích trong danh sách) thì số lần so sánh cũng là n. Như vậy, trong trường hợp tốt nhất, giải thuật tìm kiếm tuần tự chỉ cần một lần so sánh, còn trong trường hợp xấu nhất thì nó cần n lần so sánh.

Trong phần lớn các trường hợp, chúng ta không biết chính xác đặc điểm của các danh sách cần tìm kiếm, do đó chúng ta thường không áp dụng được các kết quả về thời gian chạy "tốt nhất" và "xấu nhất" trên kia. Trong các trường hợp này, chúng ta thường sử dụng **thời gian chạy trung bình**. Ở đây **trung bình** có nghĩa là chúng ta xét mỗi khả năng một lần và lấy kết quả trung bình của chúng. Tức là chúng ta giả sử các trường hợp cần tìm xảy ra với xác suất như nhau. Lưu ý rằng trong thực tế không phải lúc nào giả thiết này cũng phù hợp.

Chúng ta có số lần so sánh trung bình của giải thuật tìm kiếm tuần tự (trường hợp thành công) như sau.

$$\frac{1+2+3+...+n}{n} = \frac{1}{2}(n+1)$$

7.3. Tìm kiếm nhị phân

Giải thuật tìm kiếm tuần tự có thể được cài đặt dễ dàng và khá hiệu quả với những danh sách ngắn nhưng với những danh sách dài thì giải thuật chạy rất chậm. Với các danh sách dài, có nhiều phương pháp hữu hiệu hơn để giải quyết bài toán tìm kiếm, nhưng với điều kiện là các khoá của danh sách đã được sắp xếp sẵn.

Một trong những phương pháp tốt nhất để tìm kiếm trên **một danh sách mà các khoá đã được sắp xếp là tìm kiếm nhị phân**. Trong phương pháp này, chúng ta so sánh khoá đích với khoá của phần tử ở giữa của danh sách. Tuỳ thuộc vào khoá đích nằm trước hay sau khoá ở giữa mà chúng ta tiếp tục quá trình tìm kiếm trong nửa đầu hay nửa sau của danh sách. Với cách này, tại mỗi bước chúng ta giảm kích thước của danh sách cần tìm đi một nửa. Một danh sách chứa khoảng một triệu phần tử sẽ được xử lý trong khoảng hai mươi lần so sánh.

7.3.1. Danh sách có thứ tự

Sau đây chúng ta định nghĩa một kiểu dữ liệu trừu tượng cho một danh sách có thứ tự.

Định nghĩa: Danh sách có thứ tự (*ordered list*) là danh sách trong đó mỗi phần tử có chứa một khoá sao cho các khoá này đã được sắp thứ tự. Tức là nếu phần tử *i* đứng trước phần tử *j* trong danh sách thì khoá của *i* nhỏ hơn hay bằng khoá của *j*.

Để tìm kiếm nhị phân, danh sách cần phải có thứ tự. Chúng ta cài đặt danh sách có thứ tự là một lớp được thừa kế từ lớp `List` đã có và viết lại các phương thức **`insert`** và **`replace`**.

```
class Ordered_list:public List<Record>{
public:
   Ordered_list();
   ErrorCode insert(const Record &data);
   ErrorCode replace(int position, const Record &data);
};
```

Phương thức **`insert`** trên chèn một phần tử vào đúng vị trí của nó trong danh sách dựa vào thứ tự của các khoá. Nếu danh sách chứa nhiều khoá trùng với khoá của phần tử đang thêm vào thì khoá mới sẽ là phần tử đầu tiên trong số các phần tử có khoá trùng nhau.

```
ErrorCode Ordered_list::insert(const Record &data)
/*
post: Nếu danh sách chưa đầy, phần tử mới data được chèn vào vị trí ngay sau phần tử lớn
      nhất trong số các phần tử nhỏ hơn nó, phương thức trả về success, ngược lại phương thức
      trả về overflow.
*/
{
   int s = size();
   int position;
   for (position = 0; position < s; position++) { // Tìm vị trí thích hợp.
      Record list_data;
      retrieve(position, list_data);
      if (data >= list_data) break;
   }
   return insert(position, data); // Gọi phương thức đã có của lớp List.
}
```

Phương thức `replace` cũng cần kiểm tra tính hợp lệ của phần tử được thay thế sao cho danh sách vẫn đảm bảo thứ tự.

7.3.2. Xây dựng giải thuật

Để đảm bảo rằng giải thuật được xây dựng sẽ cho ra kết quả đúng đắn, chúng ta cần mô tả rõ ràng về ý nghĩa của các biến sử dụng trong chương trình và các điều kiện cần phải thoả trước và sau mỗi vòng lặp, đồng thời vòng lặp cũng phải được đảm bảo rằng sẽ dừng đúng.

Giải thuật tìm kiếm nhị phân sẽ sử dụng hai chỉ số, **top** và **bottom**, để giới hạn phần danh sách mà chúng ta đang tiến hành tìm kiếm. Tại mỗi bước, giải thuật giảm kích thước của phần này đi khoảng một nửa. Để tiện theo dõi tiến trình của giải thuật, chúng ta cần xác nhận một điều rằng, trước mỗi lần lặp có một điều kiện luôn đúng: khoá đích, nếu có trong danh sách, phải luôn nằm trong khoảng từ bottom đến top, có kể cả hai vị trí này. Điều kiện này lúc đầu được bảo đảm bằng cách đặt **bottom** bằng **0** và **top** là **the_list.size()-1**.

Trước tiên , giải thuật tìm vị trí phần tử ở giữa `bottom` và `top` theo công thức

$$mid = \frac{(bottom + top)}{2}$$

Kế đó giải thuật so sánh khoá đích với khoá của phần tử tại vị trí `mid` và thay đổi `top` hoặc `bottom` dựa theo kết quả của phép so sánh này.

Chúng ta lưu ý rằng giải thuật nên kết thúc khi $top \leq bottom$; tức là khi phần danh sách cần tìm còn không quá một phần tử (giả sử rằng giải thuật đã không chấm dứt sớm hơn trước đó trong trường hợp khoá đích đã được tìm thấy).

Cuối cùng, để chắc chắn rằng giải thuật dừng, số phần tử cần tìm của danh sách `(top – bottom + 1)` phải giảm sau mỗi lần lặp của giải thuật.

7.3.3. Phiên bản thứ nhất

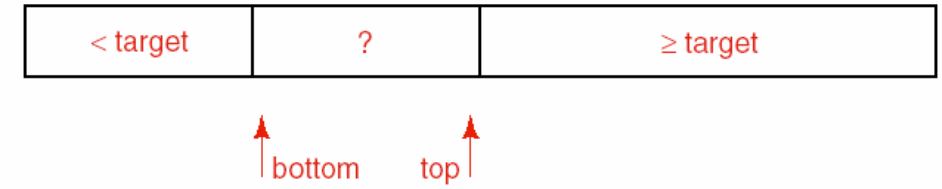
Cách cài đặt đơn giản nhất của giải thuật là cứ tiếp tục chia đôi danh sách, bất kể khoá đích có được tìm thấy hay chưa, cho tới khi danh sách còn lại có chiều dài là 1.

Hàm sau đây được viết đệ qui.

```
Error_code recursive_binary_1(const Ordered_list &the_list, const Key
                              &target, int bottom, int top, int &position)
/*
pre:  Các chỉ số bottom và top chỉ ra dãy các phần tử trong danh sách phục vụ cho việc tìm
      kiếm target.
post: Nếu phần tử có khóa trùng với target được tìm thấy thì trả về success, position chỉ
      vị trí tìm thấy. Ngược lại phương thức trả về not_present, position không có nghĩa.
uses: recursive_binary_1 và các phương thức của lớp List và Record.
```

```
*/
{
   Record data;
   if (bottom < top) {               // List có nhiều hơn 1 phần tử.
      int mid = (bottom + top) / 2;
      the_list.retrieve(mid, data);
      if (data < target)                    // Cần loại bỏ một nửa số phần tử bên phải.
         return recursive_binary_1(the_list, target, mid + 1, top, position);
      else                                  // Cần loại bỏ một nửa số phần tử bên trái.
         return recursive_binary_1(the_list, target, bottom, mid, position);
   }
   else if (top < bottom)
      return not_present;            // List rỗng.
   else {                            // List có chính xác 1 phần tử.
      position = bottom;
      the_list.retrieve(bottom, data);
      if (data == target) return success;
      else return not_present;
   }
}
```

Sự phân chia của danh sách trong quá trình tìm kiếm có thể được minh hoạ như sau:



Lưu ý rằng trong sơ đồ này phần đầu tiên chỉ chứa các phần tử nhỏ hơn khoá đích còn phần cuối có thể chứa các phần tử lớn hơn hoặc bằng khoá đích. Bằng cách này, khi phần giữa của danh sách chỉ còn một phần tử mà lại là phần tử chứa khóa đích thì phần tử này luôn là phần tử đầu tiên nếu có nhiều phần tử có khoá trùng với nó trong danh sách.

Nếu danh sách là rỗng thì hàm trên thất bại, ngược lại nó tính giá trị của **mid**. Vì **mid** được tính là trung bình của **top** và **bottom** nên nó nằm giữa **top** và **bottom,** và do đó nó là chỉ số hợp lệ của một phần tử của danh sách.

Biểu thức chia nguyên luôn làm tròn xuống, nên chúng ta có

$$bottom \leq mid < top$$

Sau khi quá trình đệ qui kết thúc, giải thuật phải kiểm tra xem khoá đích đã được tìm thấy hay chưa vì quá trình đệ qui không thực hiện phép kiểm tra này.

Để chuyển hàm trên về dạng hàm tìm kiếm chuẩn mà chúng ta định ra ở trên chúng ta định nghĩa hàm sau:

```
Error_code run_recursive_binary_1(const Ordered_list &the_list,
                                  const Key &target, int &position)
{
  return recursive_binary_1(the_list, target, 0, the_list.size() - 1,
                                                          position);
}
```

Vì phép đệ qui được sử dụng trong hàm trên là đệ qui đuôi (*tail recursion*) nên có thể chuyển thành vòng lặp một cách dễ dàng. Đồng thời chúng ta có thể làm cho các thông số của hàm trở nên thống nhất với các hàm tìm kiếm khác.

```
ErrorCode binary_search_1 (const Ordered_list &the_list,
                           const Key &target, int &position)
/*
post: Nếu phần tử có khóa trùng với target được tìm thấy thì trả về success, position chỉ vị trí
      tìm thấy. Ngược lại phương thức trả về not_present, position không có nghĩa.
uses: Các phương thức của lớp List và Record.
*/
{
   Record data;
   int bottom = 0, top = the_list.size() - 1;

   while (bottom < top) {
      int mid = (bottom + top) / 2;
      the_list.retrieve(mid, data);
      if (data < target)
         bottom = mid + 1;
      else
         top = mid;
   }
   if (top < bottom) return not_present;
   else {
      position = bottom;
      the_list.retrieve(bottom, data);
      if (data == target) return success;
      else return not_present;
   }
}
```

7.3.4. Nhận biết sớm phần tử có chứa khóa đích

Tuy binary_search_1 là một dạng đơn giản của giải thuật tìm kiếm nhị phân, nhưng nó thực hiện thừa một số lần so sánh vì nó không nhận ra trường hợp phần tử khoá được tìm thấy sớm hơn. Vì thế chúng ta có thể cải tiến giải thuật như sau.

```
Error_code recursive_binary_2(const Ordered_list &the_list, const Key
                          &target,int bottom, int top, int &position)
/*
pre:  Các chỉ số bottom và top chỉ ra dãy các phần tử trong danh sách phục vụ cho việc tìm
      kiếm target.
```

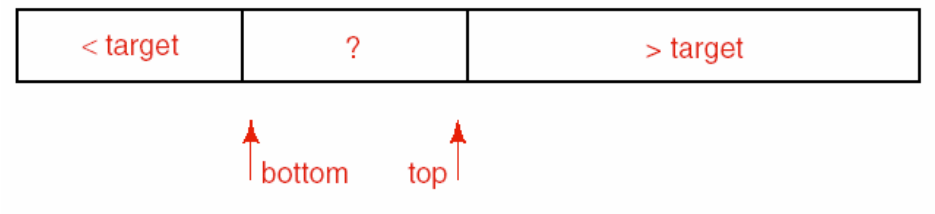
```
post: Nếu phần tử có khóa trùng với target được tìm thấy thì trả về success, position chỉ
      vị trí tìm thấy. Ngược lại phương thức trả về not_present, position không có nghĩa.
uses: recursive_binary_2 và các phương thức của lớp List và Record.
*/
{
   Record data;
   if (bottom <= top) {
      int mid = (bottom + top) / 2;
      the_list.retrieve(mid, data);
      if (data == target) {
         position = mid;
         return success;
      }

      else if (data < target)
         return recursive_binary_2(the_list, target, mid + 1, top, position);
      else
         return recursive_binary_2(the_list, target, bottom, mid - 1,
                                                                position);
   }
   else return not_present;
}
```

```
Error_code run_recursive_binary_2(const Ordered_list &the_list,
                                  const Key &target, int &position)
{
  return recursive_binary_2(the_list, target, 0, the_list.size() - 1,
                                                      position);
}
```

Chúng ta có thể chuyển hàm này thành dạng không đệ qui như sau.

```
Error_code binary_search_2(const Ordered_list &the_list,
                           const Key &target, int &position)
/*
post: Nếu phần tử có khóa trùng với target được tìm thấy thì trả về success, position chỉ
      vị trí tìm thấy. Ngược lại phương thức trả về not_present, position không có nghĩa.
uses: Các phương thức của lớp List và Record.
*/
{
   Record data;
   int bottom = 0, top = the_list.size() - 1;
   while (bottom <= top) {
      position = (bottom + top) / 2;
      the_list.retrieve(position, data);
      if (data == target) return success;
      if (data < target) bottom = position + 1;
      else top = position - 1;
   }
   return not_present;
}
```

Các hoạt động này có thể được minh họa như sau:

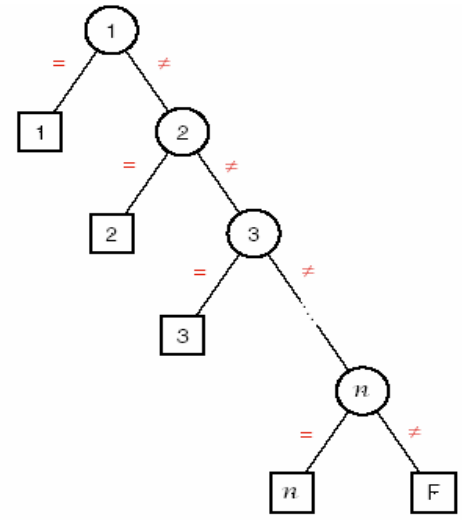


Sơ đồ này là đối xứng theo nghĩa phần thứ nhất chỉ chứa các phần tử nhỏ hơn khoá, phần thứ ba chỉ chứa các phần tử lớn hơn khoá. Khi đó, nếu như khoá xuất hiện ở nhiều vị trí trong danh sách thì giải thuật có thể trả về bất kỳ vị trí nào trong số đó. Đây cũng là nhược điểm của cách cải tiến để nhận ra sớm phần tử cần tìm này, vì trong một số ứng dụng, vị trí tương đối giữa phần tử được tìm thấy so với các phần tử có khóa trùng với nó rất quan trọng.

7.4. Cây so sánh

Cây so sánh (*comparison tree*) của một giải thuật tìm kiếm, còn gọi là cây quyết định (*decision tree*) hay cây tìm kiếm (*search tree*), là một cây có được bằng cách lần theo vết các hành vi của giải thuật. Mỗi nút của cây biểu diễn một phép so sánh. Tên của nút là chỉ số của phần tử có khóa đang được so sánh với khóa đích. Các nhánh xuất phát từ mỗi nút là các kết quả có thể có của phép so sánh tại nút đó và được đặt tên tương ứng. Khi giải thuật kết thúc chúng ta có một nút lá. Nếu giải thuật thất bại thì nút lá này được đặt tên là F, ngược lại tên của nút là chỉ số của phần tử có khoá trùng với khoá đích.

Cây so sánh cho giải thuật tìm kiếm tuần tự rất đơn giản:

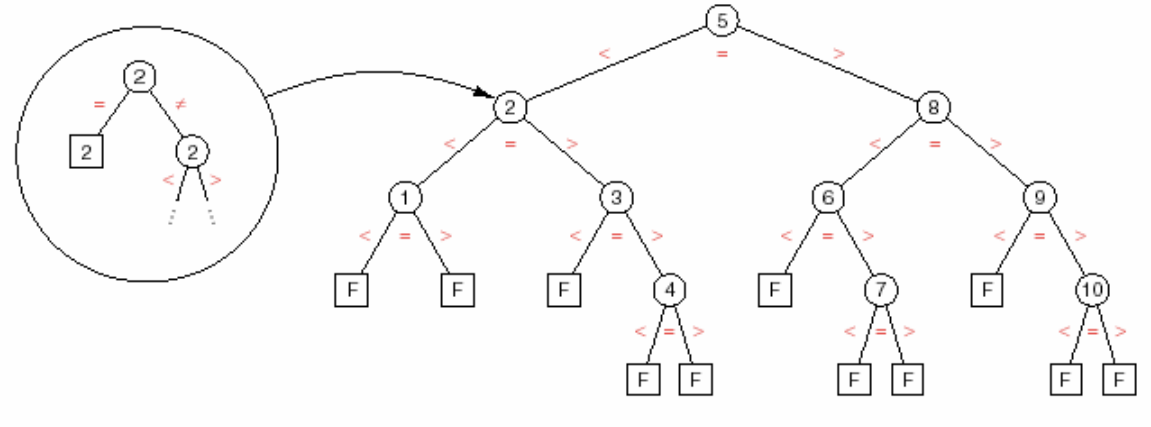


Hình 7.2- Cây so sánh cho tìm kiếm tuần tự

Số lần so sánh mà một giải thuật tìm kiếm thực hiện khi tiến hành một phép tìm kiếm cụ thể là số nút trung gian mà giải thuật đi qua kể từ gốc (nút trên cùng của cây) để đi đến nút lá cần thiết.

Hình dạng của cây so sánh cho tìm kiếm nhị phân:

Giải thuật tìm kiếm tuần tự cần nhiều phép so sánh hơn giải thuật tìm kiếm nhị phân. Chúng ta dễ dàng nhận thấy điều này qua hình dạng các cây so sánh của chúng. Giải thuật tìm kiếm tuần tự có cây so sánh hẹp và cao trong khi cây so sánh của giải thuật tìm kiếm nhị phân rộng và thấp hơn nhiều. Hình dạng cây này giúp ta hiểu được tại sao số lần so sánh trong phép tìm kiếm nhị phân là ít hơn so với tìm kiếm tuần tự.



Hình 7.3- Cây so sánh cho tìm kiếm nhị phân.

Chương 8 – SẮP XẾP

Chương này giới thiệu một số phương pháp sắp xếp cho cả danh sách liên tục và danh sách liên kết.

8.1. Giới thiệu

Để truy xuất thông tin nhanh chóng và chính xác, người ta thường sắp xếp thông tin theo một trật tự hợp lý nào đó. Có một số cấu trúc dữ liệu mà định nghĩa của chúng đã bao hàm trật tự của các phần tử, khi đó mỗi phần tử khi thêm vào đều phải đảm bảo trật tự này. Trong chương này chúng ta sẽ tìm hiểu việc sắp xếp các danh sách chưa có thứ tự trở nên có thứ tự.

Vì sắp xếp có vai trò quan trọng nên có rất nhiều phương pháp được đưa ra để giải quyết bài toán này. Các phương pháp này khác nhau ở nhiều điểm, trong đó điểm quan trọng nhất là dữ liệu cần sắp xếp nằm toàn bộ trong bộ nhớ chính (tương ứng các giải thuật sắp xếp nội) hay có một phần nằm trong thiết bị lưu trữ ngoài (tương ứng các giải thuật sắp xếp ngoại). Trong chương này chúng ta chỉ xem xét một số giải thuật sắp xếp nội.

Chúng ta sẽ sử dụng các lớp đã có ở chương 4 và chương 7. Ngoài ra, trong trường hợp khi có nhiều phần tử khác nhau có cùng khóa thì các giải thuật sắp xếp khác nhau sẽ cho ra những thứ tự khác nhau giữa chúng, và đôi khi sự khác nhau này cũng có ảnh hưởng đến các ứng dụng.

Trong các giải thuật tìm kiếm, khối lượng công việc phải thực hiện có liên quan chặt chẽ với số lần so sánh các khóa. Trong các giải thuật sắp xếp thì điều này cũng đúng. Ngoài ra, các giải thuật sắp xếp còn phải di chuyển các phần tử. Công việc này cũng chiếm nhiều thời gian, đặc biệt khi các phần tử có kích thước lớn được lưu trữ trong danh sách liên tục.

Để có thể đạt được hiệu quả cao, các giải thuật sắp xếp thường phải tận dụng các đặc điểm riêng của từng loại cấu trúc dữ liệu. Chúng ta sẽ viết các giải thuật sắp xếp dưới dạng các phương thức của lớp `List`.

```
template <class Record>
class Sortable_list:public List<Record> {
public:  //  Khai báo của các phương thức sắp xếp được thêm vào đây
private: //  Các hàm phụ trợ.
};
```

Chúng ta có thể sử dụng bất kỳ dạng hiện thực nào của lớp `List` trong chương 4. Các phần tử dữ liệu trong `Sortable_list` có kiểu là `Record`. Như đã giới thiệu trong chương 7, `Record` có các tính chất sau đây:

- Mỗi mẫu tin có một khoá đi kèm.
- Các khóa có thể được so sánh với nhau bằng các toán tử so sánh.
- Một mẩu tin có thể được chuyển đổi tự động thành một khóa. Do đó có thể so sánh các mẫu tin với nhau hoặc so sánh mẫu tin với khoá thông qua việc chuyển đổi mẫu tin về khóa liên quan đến nó.

8.2. Sắp xếp kiểu chèn (*Insertion Sort*)

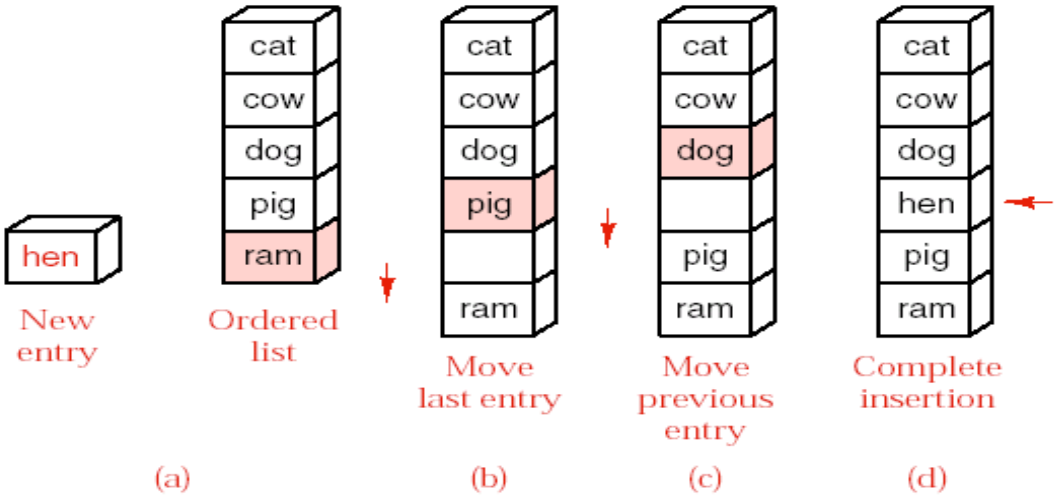
Phương pháp sắp xếp chen vào dựa trên ý tưởng chèn phần tử vào danh sách đã có thứ tự.

8.2.1. Chèn phần tử vào danh sách đã có thứ tự

Định nghĩa danh sách có thứ tự đã được trình bày trong chương 7.

Với các danh sách có thứ tự, một số tác vụ chỉ sử dụng khóa của phần tử chứ không sử dụng vị trí của phần tử:

- `retrieve`: truy xuất một phần tử có khóa cho trước.
- `insert`: chèn một phần tử có khóa cho trước sao cho danh sách vẫn còn thứ tự, vị trí của phần tử mới được xác định bởi khóa của nó.



Hình 8.1 – Chèn phần tử vào danh sách đã có thứ tự.

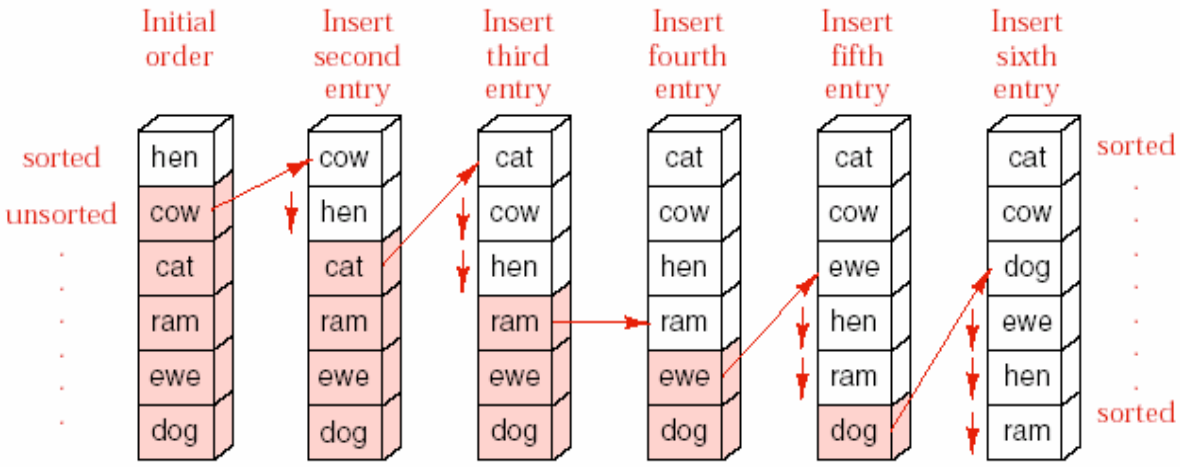
Phép thêm vào và phép truy xuất có thể không cho kết quả duy nhất trong trường hợp có nhiều phần tử trùng khóa. Phép truy xuất phần tử dựa trên khóa chính là phép tìm kiếm đã được trình bày trong chương 7.

Để thêm phần tử mới vào danh sách liên tục đã có thứ tự, các phần tử sẽ đứng sau nó phải được dịch chuyển để tạo chỗ trống. Chúng ta cần thực hiện phép

tìm kiếm để tìm vị trí chen vào. Vì danh sách đã có thứ tự nên ta có thể sử dụng phép tìm kiếm nhị phân. Tuy nhiên, do thời gian cần cho việc di chuyển các phần tử lớn hơn nhiều so với thời gian tìm kiếm, nên việc tiết kiệm thời gian tìm kiếm cũng không cải thiện thời gian chạy toàn bộ giải thuật được bao nhiêu. Nếu việc tìm kiếm tuần tự từ cuối danh sách có thứ tự được thực hiện đồng thời với việc di chuyển phần tử, thì chi phí cho một lần chen một phần tử mới là tối thiểu.

8.2.2. Sắp xếp kiểu chèn cho danh sách liên tục

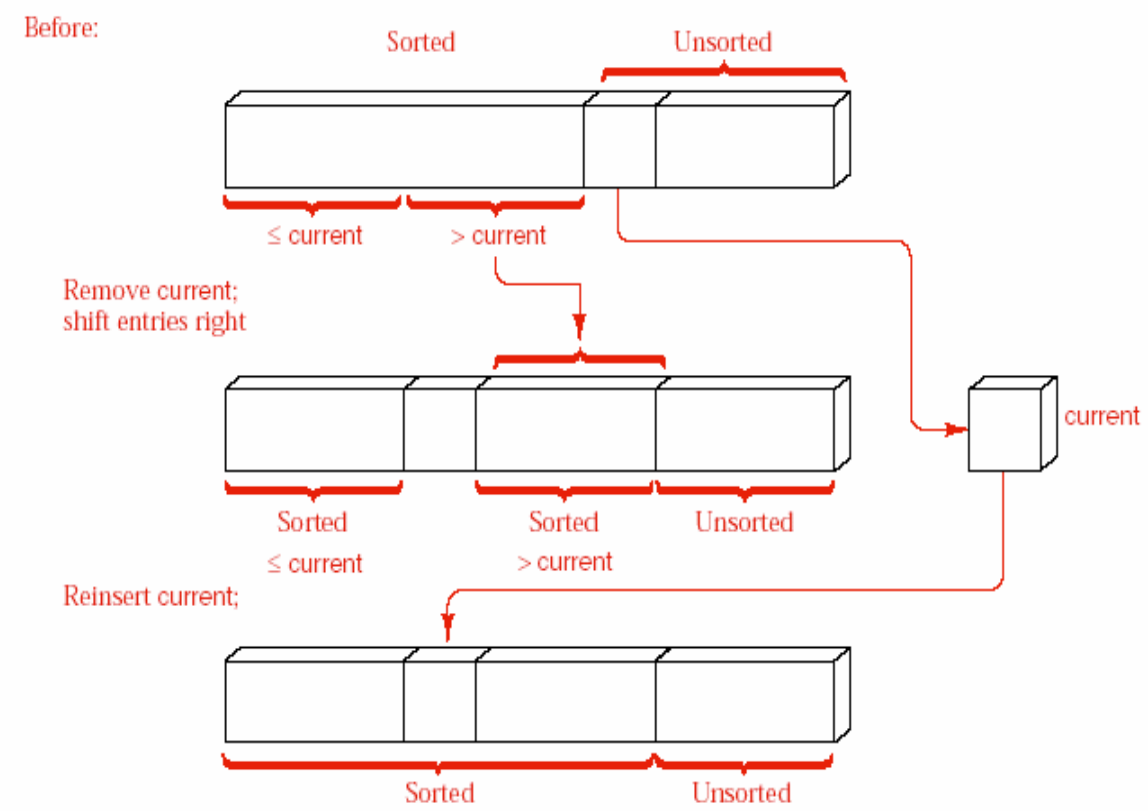
Tác vụ thêm một phần tử vào danh sách có thứ tự là cơ sở của phép sắp xếp kiểu chèn. Để sắp xếp một danh sách chưa có thứ tự, chúng ta lần lượt lấy ra từng phần tử của nó và dùng tác vụ trên để đưa vào một danh sách lúc đầu là rỗng.



Hình 8.2- Ví dụ về sắp xếp kiểu chèn cho danh sách liên tục.

Phương pháp này được minh họa trong hình 8.2. Hình này chỉ ra các bước cần thiết để sắp xếp một danh sách có 6 từ. Nhìn hình vẽ chúng ta thấy, phần danh sách đã có thứ tự gồm các phần tử từ chỉ số `sorted` trở lên trên, các phần tử từ chỉ số `unsorted` trở xuống dưới là chưa có thứ tự. Bước đầu tiên, từ "hen" được xem là đã có thứ tự do danh sách có một phần tử đương nhiên là danh sách có thứ tự. Tại mỗi bước, phần tử đầu tiên trong phần danh sách bên dưới được lấy ra và chèn vào vị trí thích hợp trong phần danh sách đã có thứ tự bên trên. Để có chỗ chèn phần tử này, một số phần tử khác trong phần danh sách đã có thứ tự được di chuyển về phía cuối danh sách.

Trong phương thức dưới đây, **`first_unsorted`** là chỉ số phần tử đầu tiên trong phần danh sách chưa có thứ tự, và **`current`** là biến tạm nắm giữ phần tử này cho đến khi tìm được chỗ trống để chèn vào.



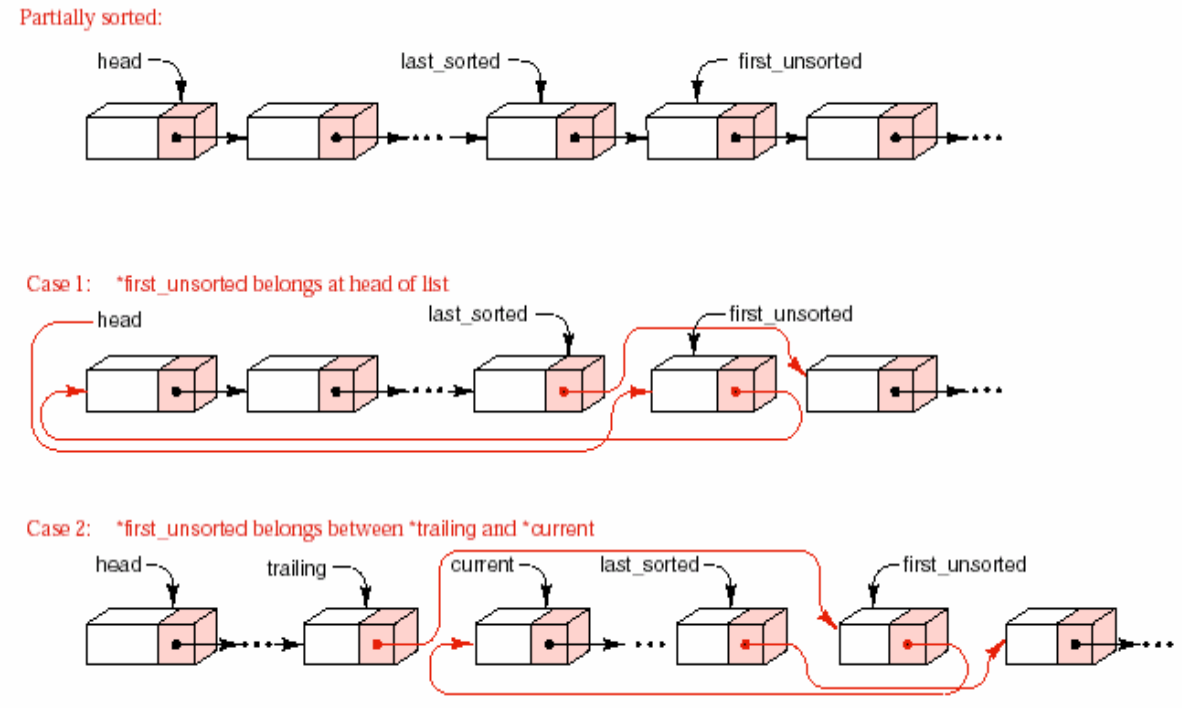
Hình 8.3- Bước chính của giải thuật sắp xếp kiểu chèn.

```
// Dành cho danh sách liên tục  trong chương 4.

template <class Record>
void Sortable_list<Record>::insertion_sort()
/*
post: Các phần tử trong danh sách đã được sắp xếp theo thứ tự không giảm.
uses: Các phương thức của lớp Record.
*/
{
   int first_unsorted;//Chỉ số phần tử đầu tiên trong phần danh sách chưa có thứ tự.
   int position;  // Chỉ số dùng cho việc di chuyển các phần tử về phía sau.
   Record current;// Nắm giữ phần tử đang được tìm chỗ chèn vào phần danh sách đã có thứ
                                                                          tự.
   for (first_unsorted = 1; first_unsorted < count; first_unsorted++)
      if (entry[first_unsorted] < entry[first_unsorted - 1]) {
         position = first_unsorted;
         current = entry[first_unsorted];
         do {    // Di chuyển dần các phần tử về phía sau để tìm chỗ trống thích hợp.
            entry[position] = entry[position - 1];
            position--;
         } while (position > 0 && entry[position - 1] > current);
         entry[position] = current;
      }
}
```

Vì danh sách có một phần tử xem như đã có thứ tự nên vòng lặp trên biến `first_unsorted` bắt đầu với phần tử thứ hai. Nếu phần tử này đã ở đúng vị trí thì không cần phải tiến hành thao tác nào. Ngược lại, phần tử được đưa vào biến **`current`**, vòng lặp **`do...while`** đẩy các phần tử lùi về sau một vị trí cho đến khi tìm được vị trí đúng cho **`current`**. Trường hợp vị trí đúng của `current` là đầu dãy cần được nhận biết riêng bởi điều kiện thoát khỏi vòng lặp là **`position==0`**.

8.2.3. Sắp xếp kiểu chèn cho danh sách liên kết

Với danh sách liên kết, chúng ta không cần di chuyển các phần tử, do đó cũng không cần bắt đầu tìm kiếm từ phần tử cuối của phần danh sách đã có thứ tự. Hình 8.4 minh họa giải thuật này. Con trỏ **`last_sorted`** chỉ phần tử cuối cùng của phần danh sách đã có thứ tự, **`last_sorted->next`** chỉ phần tử đầu tiên của phần danh sách chưa có thứ tự. Ta dùng biến **`first_unsorted`** để chỉ vào phần tử này và biến **`current`** để tìm vị trí thích hợp cho việc chèn phần tử **`*first_unsorted`** vào. Nếu vị trí đúng của **`*first_unsorted`** là đầu danh sách thì nó được chèn vào vị trí này. Ngược lại, **`current`** được di chuyển về phía cuối danh sách cho đến khi có (**`current->entry >= first_unsorted->entry`**) và **`*first_unsorted`** được thêm vào ngay trước **`*current`**. Để có thể thực hiện việc thêm vào trước **`current`**, chúng ta cần một con trỏ **`trailing`** luôn đứng trước **`current`** một vị trí.



Hình 8.4- Sắp xếp kiểu chèn cho danh sách liên kết.

Như chúng ta đã biết, phần tử cầm canh (*sentinel*) là một phần tử được thêm vào một đầu của danh sách để đảm bảo rằng vòng lặp luôn kết thúc mà không cần phải thực hiện bổ sung một phép kiểm tra nào. Vì chúng ta đã có

`last_sorted->next == first_unsorted`,

nên phần tử **`*first_unsorted`** đóng luôn vai trò của phần tử cầm canh trong khi **`current`** tiến dần về phía cuối phần danh sách đã có thứ tự. Nhờ đó, điều kiện dừng của vòng lặp di chuyển **current** luôn được đảm bảo.

Ngoài ra, danh sách rỗng hay danh sách có một phần tử là đương nhiên có thứ tự, nên ta có thể kiểm tra trước dễ dàng.

Mặc dù cơ chế hiện thực của sắp xếp kiểu chèn cho danh sách liên kết và cho danh sách liên tục có nhiều điểm khác nhau nhưng về ý tưởng thì hai phiên bản này rất giống nhau. Điểm khác biệt lớn nhất là trong danh sách liên kết việc tìm kiếm được thực hiện từ đầu danh sách trong khi trong danh sách liên tục việc tìm kiếm được thực hiện theo chiều ngược lại.

```
// Dành cho danh sách liên kết trong chương 4.
template <class Record>
void Sortable_list<Record>::insertion_sort()
/*
post: Các phần tử trong danh sách đã được sắp xếp theo thứ tự không giảm.
uses: Các phương thức của lớp Record.
*/
{
   Node <Record>  *first_unsorted,
                  *last_sorted,
                  *current,
                  *trailing;
   if (head != NULL) {          // Loại trường hợp danh sách rỗng và
      last_sorted =     head;   // trường hợp danh sách chỉ có 1 phần tử.

      while (last_sorted->next != NULL) {
         first_unsorted = last_sorted->next;
         if (first_unsorted->entry < head->entry) {
            //  *first_unsorted được chèn vào đầu danh sách.
            last_sorted->next = first_unsorted->next;
            first_unsorted->next = head;
            head = first_unsorted;
         }
         else {
            //  Tìm vị trí thích hợp.
            trailing = head;
            current = trailing->next;
            while (first_unsorted->entry > current->entry) {
               trailing = current;
               current = trailing->next;
            }
            // *first_unsorted được chèn vào giữa *trailing và *current.
```

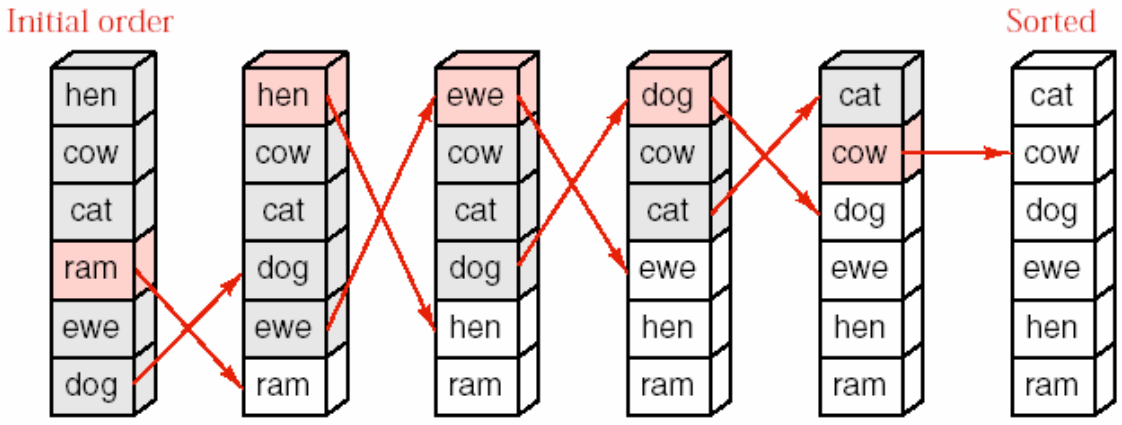
```
            if (first_unsorted == current)
               last_sorted = first_unsorted;        // vị trí đang có đã đúng.
            else {
               last_sorted->next = first_unsorted->next;
               first_unsorted->next = current;
               trailing->next = first_unsorted;
            }
         }
      }
   }
}
```

Thời gian cần thiết để sắp xếp danh sách dùng giải thuật sắp xếp kiểu chèn tỉ lệ với bình phương số phần tử của danh sách.

8.3. Sắp xếp kiểu chọn (*Selection Sort*)

Sắp xếp kiểu chèn có một nhược điểm lớn. Sau khi một số phần tử đã được sắp xếp vào phần đầu của danh sách, việc sắp xếp một phần tử phía sau đôi khi đòi hỏi phải di chuyển phần lớn các phần tử đã có thứ tự này. Mỗi lần di chuyển, các phần tử chỉ được di chuyển một vị trí, do đó nếu một phần tử cần di chuyển 20 vị trí để đến được vị trí đúng cuối cùng của nó thì nó cần được di chuyển 20 lần. Nếu kích thước của mỗi phần tử là nhỏ hoặc chúng ta sử dụng danh sách liên kết thì việc di chuyển này không cần nhiều thời gian lắm. Nhưng nếu kích thước mỗi phần tử lớn và danh sách là liên tục thì thời gian di chuyển các phần tử sẽ rất lớn. Như vậy, nếu như mỗi phần tử, khi cần phải di chuyển, được di chuyển ngay đến vị trí đúng sau cùng của nó thì giải thuật sẽ chạy hiệu quả hơn nhiều. Sau đây chúng ta trình bày một giải thuật để đạt được điều đó.

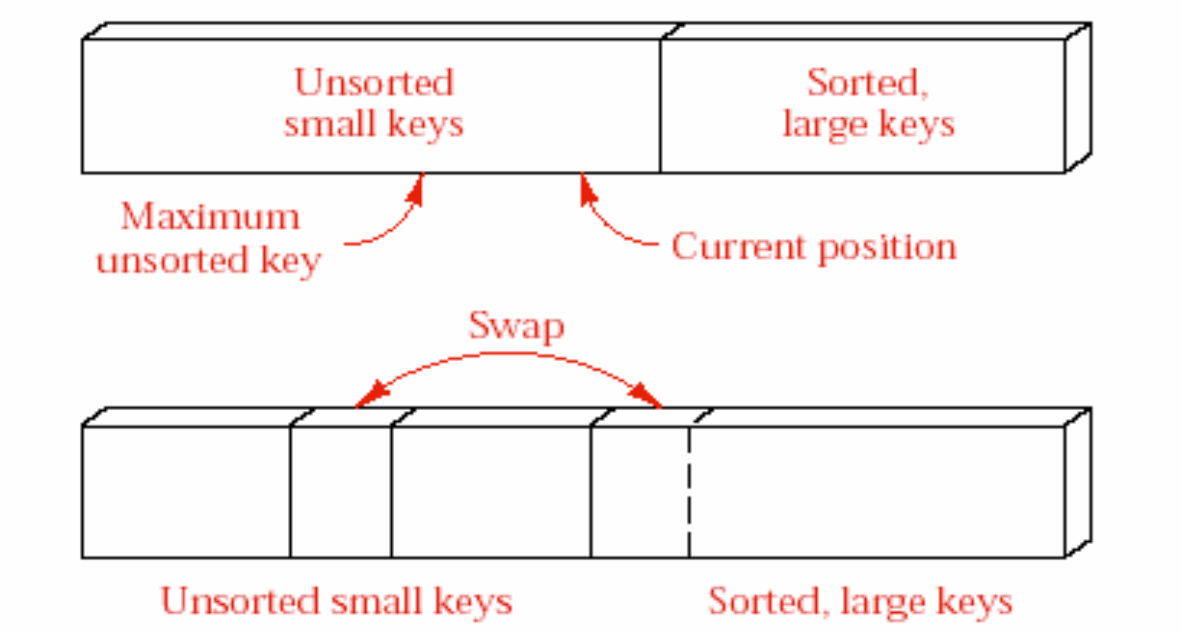
8.3.1. Giải thuật



Hình 8.5- Ví dụ sắp xếp kiểu chọn.

Hình 8.5 trình bày một ví dụ sắp xếp 6 từ theo thứ tự của bảng chữ cái. Tại bước đầu tiên, chúng ta tìm phần tử sẽ đứng tại vị trí cuối cùng trong danh sách có thứ tự và tráo đổi vị trí với phần tử cuối cùng hiện tại. Trong các bước kế tiếp, chúng ta tiếp tục lặp lại công việc trên với phần còn lại của danh sách không kể các phần tử đã được chọn trong các bước trước. Khi phần danh sách chưa được sắp xếp chỉ còn lại một phần tử thì giải thuật kết thúc.

Bước tổng quát trong sắp xếp kiểu chọn được minh họa trong hình 8.6. Các phần tử có khóa lớn đã được sắp theo thứ tự và đặt ở phần cuối danh sách. Các phần tử có khóa nhỏ hơn chưa được sắp xếp. Chúng ta tìm trong số những phần tử chưa được sắp xếp để lấy ra phần tử có khóa lớn nhất và đổi chỗ nó về cuối các phần tử này. Bằng cách này, tại mỗi bước, một phần tử được đưa về đúng vị trí cuối cùng của nó.



Hình 8.6- Một bước trong sắp xếp kiểu chọn.

8.3.2. Sắp xếp chọn trên danh sách liên tục

Sắp xếp chọn giảm tối đa việc di chuyển dữ liệu do mỗi bước đều có ít nhất một phần tử được đặt vào đúng vị trí cuối cùng của nó. Vì vậy sắp xếp kiểu chọn thích hợp cho các danh sách liên tục có các phần tử có kích thước lớn. Nếu các phần tử có kích thước nhỏ hay danh sách có hiện thực là liên kết thì sắp xếp kiểu chèn thường nhanh hơn sắp xếp kiểu chọn. Do đó chúng ta chỉ xem xét sắp xếp kiểu chọn cho danh sách liên tục. Giải thuật sau đây sử dụng hàm phụ trợ **`max_key`** của `Sortable_list` để tìm phần tử lớn nhất.

```
// Dành cho danh sách liên tục trong chương 4.

template <class Record>
void Sortable_list<Record>::selection_sort()
/*
post: Các phần tử trong danh sách đã được sắp xếp theo thứ tự không giảm.
uses: max_key, swap.
*/
{
   for (int position = count - 1; position > 0; position--) {
      int max = max_key(0, position);
      swap(max, position);
   }
}
```

Lưu ý rằng khi n-1 phần tử đã đứng vào vị trí đúng (n là số phần tử của danh sách) thì phần tử còn lại đương nhiên có vị trí đúng. Do đó vòng lặp kết thúc tại `position==1`.

```
template <class Record>
// Dành cho danh sách liên tục trong chương 4.

int Sortable_list<Record>::max_key(int low, int high)
/*
pre:  low và high là hai vị trí hợp lệ trong danh sách và low <= high.
post: trả về vị trí phần tử lớn nhất nằm trong khoảng chỉ số từ low đến high.
uses: lớp Record.
*/
{
   int largest, current;
   largest = low;
   for (current = low + 1; current <= high; current++)
      if (entry[largest] < entry[current])
         largest = current;
   return largest;
}

template <class Record>
void Sortable_list<Record>::swap(int low, int high)
/*
pre:  low và high là hai vị trí hợp lệ trong danh sách Sortable_list.
post: Phần tử tại low hoán đổi với phần tử tại high.
uses: Hiện thực danh sách liên tục trong chương 4.
*/
{
   Record temp;
   temp = entry[low];
   entry[low] = entry[high];
   entry[high] = temp;
}
```

Ưu điểm chính của sắp xếp kiểu chọn liên quan đến việc di chuyển dữ liệu. Nếu một phần tử đã ở đúng vị trí của nó thì sẽ không bị di chuyển nữa. Khi hai

phần tử nào đó được đổi chỗ thì ít nhất một trong hai phần tử sẽ được đưa vào đúng vị trí cuối cùng của phần tử trong danh sách.

8.4. Shell_sort

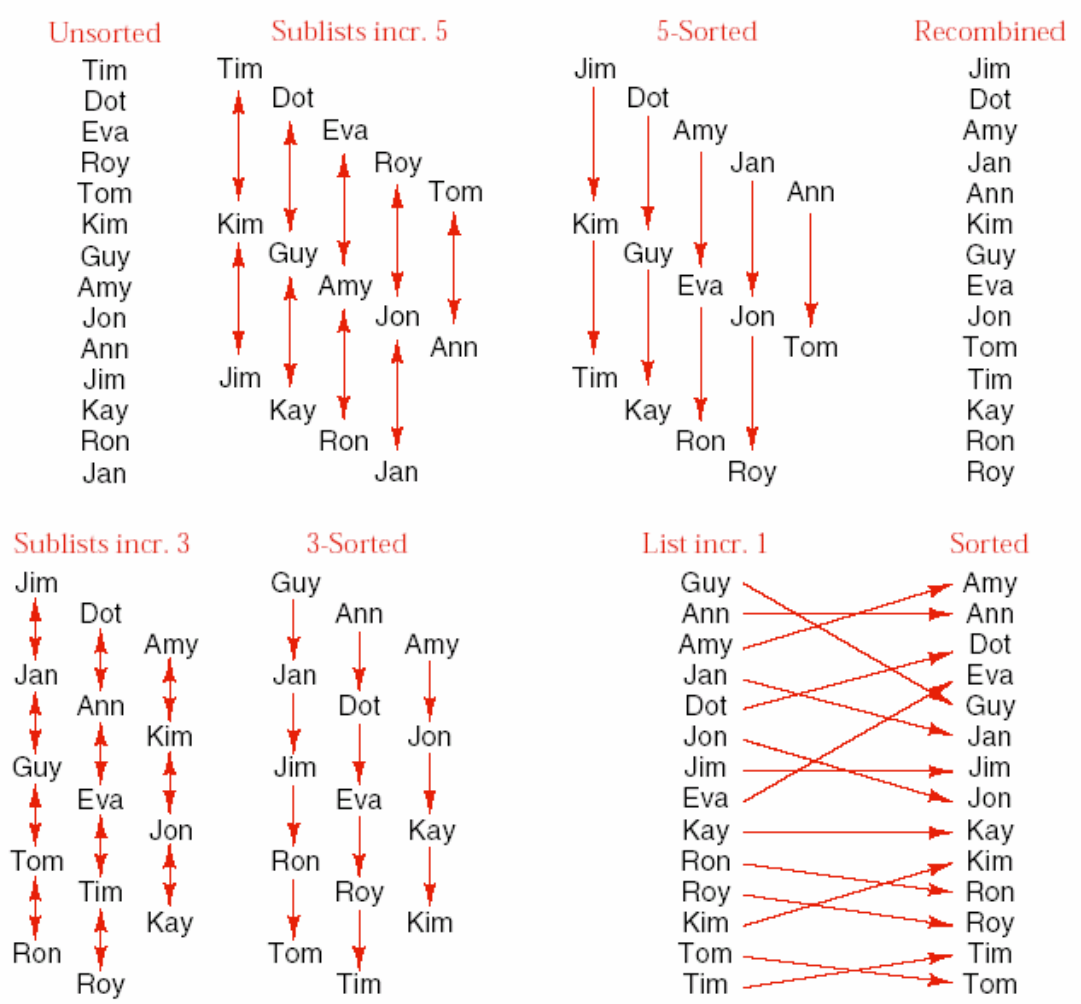
Như chúng ta thấy, nguyên tắc hoạt động của sắp xếp kiểu chèn và sắp xếp kiểu chọn là ngược nhau. Sắp xếp kiểu chọn thực hiện việc di chuyển phần tử rất hiệu quả nhưng lại thực hiện nhiều phép so sánh thừa. Trong trường hợp tốt nhất có thể xảy ra, sắp xếp kiểu chèn thực hiện rất ít các phép so sánh nhưng lại thực hiện rất nhiều phép di chuyển dữ liệu thừa. Sau đây chúng ta xem xét một phương pháp trong đó nhược điểm của mỗi phương pháp trên sẽ được tránh càng nhiều càng tốt.

Lý do khiến giải thuật sắp xếp kiểu chèn chỉ di chuyển các phần tử mỗi lần được một vị trí là vì nó chỉ so sánh các phần tử gần nhau. Nếu chúng ta thay đổi giải thuật này sao cho nó so sánh các phần tử ở xa nhau trước thì khi có sự đổi chỗ, một phần tử sẽ di chuyển xa hơn. Dần dần, khoảng cách này được giảm dần đến 1 để đảm bảo rằng toàn bộ danh sách được sắp xếp. Đây cũng là tư tưởng của giải thuật Shell sort, được D.L. Shell đề xuất và hiện thực vào năm 1959. Phương pháp này đôi khi còn được gọi là phương pháp sắp xếp giảm độ tăng (*diminishing-increment sort*).

Ơ đây chúng ta xem một ví dụ khi sắp xếp các tên. Lúc đầu ta sắp xếp các tên ở cách nhau 5 vị trí, sau đó giảm xuống 3 và cuối cùng còn 1.

Mặc dù chúng ta phải duyệt danh sách nhiều lần, nhưng trong những lần duyệt trước các phần tử đã được di chuyển đến gần vị trí cuối cùng của chúng. Nhờ vậy những lần duyệt sau, các phần tử nhanh chóng được di chuyển về vị trí đúng sau cùng của chúng.

Các khoảng cách 5, 3, 1 được chọn ngẫu nhiên. Tuy nhiên, không nên chọn các bước di chuyển mà chúng lại là bội số của nhau. Vì khi đó thì các phần tử đã được so sánh với nhau ở bước trước sẽ được so sánh trở lại ở bước sau, mà như vậy vị trí của chúng sẽ không được cải thiện. Đã có một số nghiên cứu về `Shell_sort`, nhưng chưa ai có thể chỉ ra các khoảng cách di chuyển nào là tốt nhất. Tuy nhiên cũng có một số gợi ý về cách chọn các khoảng cách di chuyển. Nếu các khoảng di chuyển được chọn gần nhau thì sẽ phải duyệt danh sách nhiều lần hơn nhưng mỗi lần duyệt lại nhanh hơn. Ngược lại, nếu khoảng cách di chuyển giảm nhanh thì có ít lần duyệt hơn và mỗi lần duyệt sẽ tốn nhiều thời gian hơn. Điều quan trọng nhất là khoảng di chuyển cuối cùng phải là 1.



Hình 8.7 – Ví dụ về `Shell_Sort`.

```
template <class Record>
void Sortable_list<Record>::shell_sort()
/*
post: Các phần tử trong Sortable_list đã được sắp theo thứ tự khóa không giảm.
uses: Hàm sort_interval
*/
{
   int increment = count; //  Khoảng cách giữa các phần tử trong mỗi danh sách con.
   int start;
   do {
      increment = increment / 3 + 1; // Giảm khoảng cách qua mỗi lần lặp.
      for (start = 0; start < increment; start++)
         sort_interval(start, increment);// Biến thể của giải thuật sắp xếp kiểu chèn.
   } while (increment > 1);
}
```

Hàm `sort_interval` là một biến thể của giải thuật sắp xếp kiểu chèn, với thông số increment là khoảng cách của hai phần tử kế nhau trong danh sách cần được sắp thứ tự. Tuy nhiên có một điều cần lưu ý là việc sắp xếp trong từng danh sách con không nhất thiết phải dùng phương pháp chen vào.

Tại bước cuối cùng, khoảng di chuyển là 1 nên Shell_sort về bản chất vẫn là sắp xếp kiểu chèn. Vì vậy tính đúng đắn của Shell_sort cũng tương tự như sắp xếp kiểu chèn. Về mặt hiệu quả, chúng ta hy vọng rằng các bước tiền xử lý sẽ giúp cho quá trình xử lý nhanh hơn.

Việc phân tích Shell_sort là đặc biệt khó. Cho đến nay, người ta chỉ mới có thể ước lượng được số lần so sánh và số phép gán cần thiết cho giải thuật trong những trường hợp đặc biệt.

8.5. Các phương pháp sắp xếp theo kiểu chia để trị

8.5.1. Ý tưởng cơ bản

Từ những giải thuật đã được trình bày và từ kinh nghiệm thực tế ta rút ra kết luận rằng sắp xếp danh sách dài thì khó hơn là sắp xếp danh sách ngắn. Nếu chiều dài danh sách tăng gấp đôi thì công việc sắp xếp thông thường tăng hơn gấp đôi (với sắp xếp kiểu chèn và sắp xếp kiểu chọn, khối lượng công việc tăng lên khoảng bốn lần). Do đó, nếu chúng ta có thể chia một danh sách ra thành hai phần có kích thước xấp xỉ nhau và thực hiện việc sắp xếp mỗi phần một cách riêng rẽ thì khối lượng công việc cần thiết cho việc sắp xếp sẽ giảm đi đáng kể. Ví dụ việc sắp xếp các phiếu trong thư viện sẽ nhanh hơn nếu các phiếu được chia thành từng nhóm có chung chữ cái đầu và từng nhóm được tiến hành sắp xếp riêng rẽ.

Ơ đây chúng ta vận dụng ý tưởng chia một bài toán thành nhiều bài toán tương tự như bài toán ban đầu nhưng nhỏ hơn và giải quyết các bài toán nhỏ này. Sau đó chúng ta tổng hợp lại để có lời giải cho toàn bộ bài toán ban đầu. Phương pháp này được gọi là “chia để trị” (*divide-and-conquer*).

Để sắp xếp các danh sách con, chúng ta lại áp dụng chiến lược chia để trị để tiếp tục chia nhỏ từng danh sách con. Quá trình này dĩ nhiên không bị lặp vô tận. Khi ta có các danh sách con với kích thước là 1 phần tử thì quá trình dừng.

Chúng ta có thể thể hiện ý tưởng trên trong đoạn mã giả sau đây.

```
void Sortable_list::sort()
{
   if (danh sách có nhiều hơn 1 phần tử)
   {
      Phân hoạch danh sách thành hai danh sách con lowlist, highlist;
      lowlist.sort();
      highlist.sort();
      Kết nối hai danh sách con lowlist và highlist;
   }
}
```

Vấn đề còn lại cần xem xét là cách phân hoạch (*partition*) danh sách ban đầu và cách kết nối (*combine*) hai danh sách đã có thứ tự cho thành một danh sách có thứ tự. Có hai phương pháp dưới đây, mỗi phương pháp sẽ làm việc tốt ứng với một số trường hợp riêng.

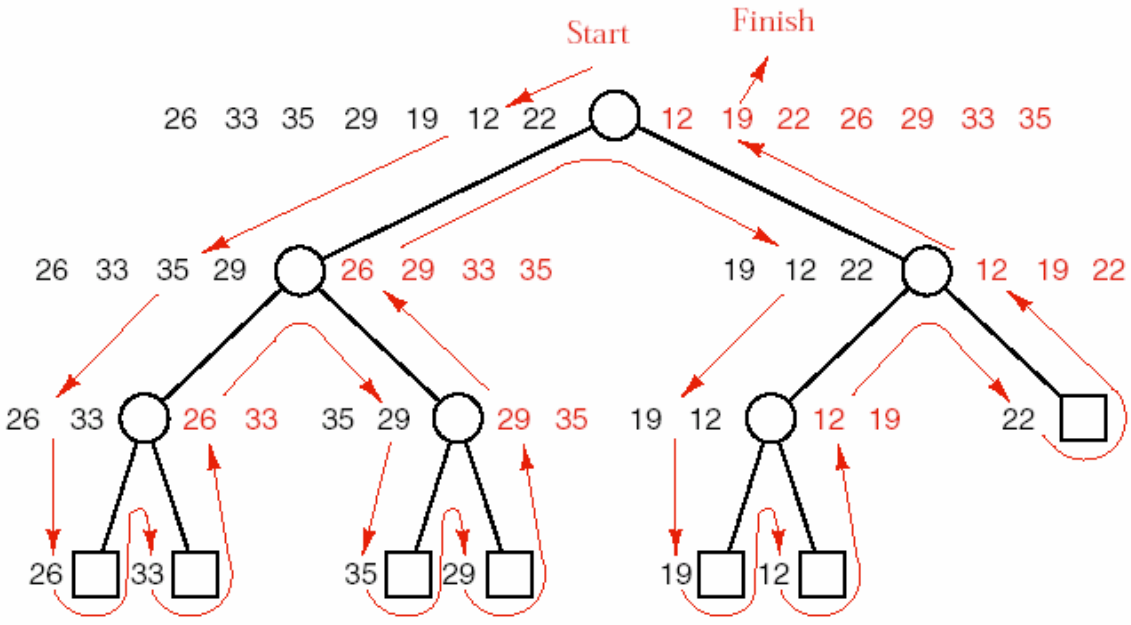
- **`Merge_sort`**: theo phương pháp này hai danh sách con chỉ cần có kích thước gần bằng nhau. Sau khi sắp xếp xong thì chúng được trộn lại sao cho danh sách cuối cùng có thứ tự.

- **`Quick_sort`**: theo phương pháp này, hai danh sách con được chia sao cho bước kết nối lại trở nên đơn giản. Phương pháp này được C. A. R. Hoare đưa ra. Để phân hoạch danh sách, chúng ta sẽ chọn một phần tử từ trong danh sách với hi vọng rằng có khoảng một nửa số phần tử đứng trước và khoảng một nửa số phần tử đứng sau phần tử được chọn trong danh sách có thứ tự sau cùng. Phần tử này được gọi là phần tử trụ (`pivot`). Sau đó chúng ta chia các phần tử theo qui tắc: các phần tử có khoá nhỏ hơn khoá của phần tử trụ được chia vào danh sách thứ nhất, các phần tử có khoá lớn hơn khoá của phần tử trụ được chia vào danh sách thứ hai. Khi hai danh sách này đã được sắp xếp thì chúng ta chỉ cần nối chúng lại với nhau.

8.5.2. Ví dụ

Trước khi xem xét chi tiết của các giải thuật, chúng ta sẽ thực hiện việc sắp xếp một danh sách cụ thể có 7 số như sau:

26 33 35 29 19 12 22



Hình 8.8- Cây đệ quy cho `Merge_sort` với 7 số.

8.5.2.1. Ví dụ cho Merge_sort

Bước đầu tiên là chia danh sách thành hai phần. Khi số phần tử của danh sách là lẻ thì chúng ta sẽ qui ước danh sách con bên trái sẽ dài hơn danh sách con bên phải một phần tử. Theo qui ước này, chúng ta có hai danh sách con

26 33 35 29 và 19 12 22

Ta xem xét danh sách con thứ nhất trước. Danh sách này cũng được chia thành hai phần

26 33 và 35 29

với mỗi nửa này, chúng ta lại áp dụng phương pháp trên để được các danh sách con có chiều dài là 1. Các danh sách con chiều dài 1 phần tử này không cần phải sắp xếp. Cuối cùng chúng ta cần phải trộn các danh sách con này để được một danh sách có thứ tự. 26 và 33 tạo thành danh sách 26 33; 35 và 29 được trộn thành 29 35. Kế tiếp, hai danh sách này này được trộn thành 26 29 33 35.

Tương tự như vậy, với nửa thứ hai của danh sách ban đầu ta được

12 19 22

Cuối cùng, trộn hai phần này ta được

12 19 22 26 29 33 35

8.5.2.2. Ví dụ cho Quick_sort

Chúng ta sử dụng ví dụ này cho `Quick_sort`.

Để sử dụng `Quick_sort`, trước tiên chúng ta phải xác định phần tử trụ. Phần tử này có thể là phần tử bất kỳ nào của danh sách, tuy nhiên, để cho thống nhất chúng ta sẽ chọn phần tử đầu tiên. Trong các ứng dụng thực tế thường người ta có những cách xác định phần tử trụ khác tốt hơn.

Theo ví dụ này, phần tử trụ đầu tiên là 26. Do đó hai danh sách con được tạo ra là:

19 12 22 và 33 35 29

Hai danh sách này lần lượt chứa các số nhỏ hơn và lớn hơn phần tử trụ. Ơ đây thứ tự của các phần tử trong hai danh sách con không đổi so với danh sách ban đầu nhưng đây không phải là điều bắt buộc.

Chúng ta tiếp tục sắp xếp các chuỗi con. Với chuỗi con thứ nhất, chúng ta chọn phần tử trụ là 19, do đó được hai danh sách con là 12 và 22. Hai danh sách này

chỉ có một phần tử nên đương nhiên có thứ tự. Cuối cùng, gom hai danh sách con và phần tử trụ lại ta có danh sách đã sắp xếp

12 19 22

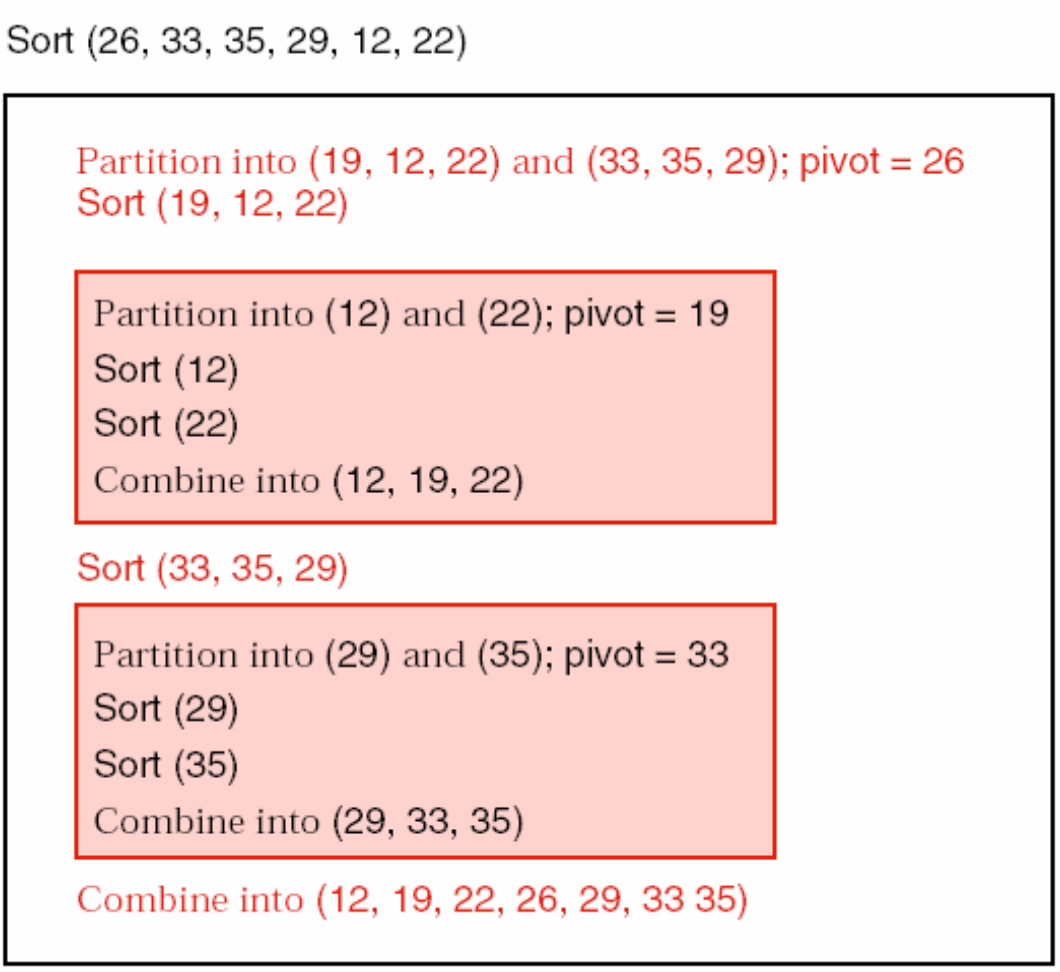
Áp dụng phương pháp trên cho phần thứ hai của danh sách, ta được danh sách cuối cùng là

29 33 35

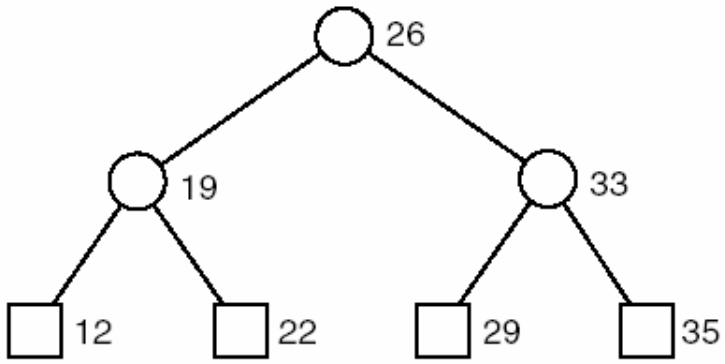
Gom hai danh sách con đã sắp xếp này và phần tử trụ đầu tiên ta được danh sách có thứ tự sau cùng:

12 19 22 26 29 33 35

Các bước của giải thuật được minh hoạ bởi hình sau.



Hình 8.9- Các bước thực thi của `Quick_sort`



Hình 8.10- Cây đệ quy cho `Quick_sort` với 7 phần tử.

8.6. `Merge_sort` cho danh sách liên kết

Sau đây là các hàm để thực hiện các phép sắp xếp nói trên. Với `Merge_sort`, chúng ta viết một phiên bản cho danh sách liên kết còn với `Quick_sort` thì chúng ta viết một phiên bản cho danh sách liên tục. Sinh viên có thể tự phân tích xem liệu cách làm ngược lại có khả thi và có hiệu quả hay không (`Merge_sort` cho danh sách liên tục và `Quick_sort` cho danh sách liên kết).

`Merge_sort` còn là một phương pháp rất tốt cho việc sắp xếp ngoại, tức là sắp xếp các dữ liệu nằm trên bộ nhớ ngoài có tốc độ truy xuất khá chậm và không có khả năng truy xuất ngẫu nhiên.

Sắp xếp danh sách liên kết có nghĩa là thay đổi các mối liên kết trong danh sách và tránh việc tạo mới hay xoá đi các phần tử. Cụ thể hơn, chương trình `Merge_sort` sẽ gọi một hàm đệ qui để xử lý từng tập con các phần tử của danh sách liên kết.

```
// Dành cho danh sách liên kết trong chương 4.

template <class Record>
void Sortable_list<Record>::merge_sort()
/*
post: Các phần tử trong danh sách đã được sắp theo thứ tự không giảm.
uses: recursive_merge_sort.
*/
{
   recursive_merge_sort(head);
}
```

Sau đây là hàm `recursive_merge_sort` được viết dưới dạng đệ qui.

```
template <class Record>
void Sortable_list<Record>::recursive_merge_sort(Node<Record> *&sub_list)
/*
post: Các phần tử trong danh sách tham chiếu bởi sub_list đã được sắp theo thứ tự không
      giảm. Tham biến con trỏ sub_list được cập nhật chứa địa chỉ phần tử đầu tiên và cũng
      là phần tử nhỏ nhất trong danh sách.
uses: Các hàm divide_from, merge, và recursive_merge_sort.
*/
{
   if (sub_list != NULL && sub_list->next != NULL) {
      Node<Record> *second_half = divide_from(sub_list);
      recursive_merge_sort(sub_list);
      recursive_merge_sort(second_half);
      sub_list = merge(sub_list, second_half);
   }
}
```

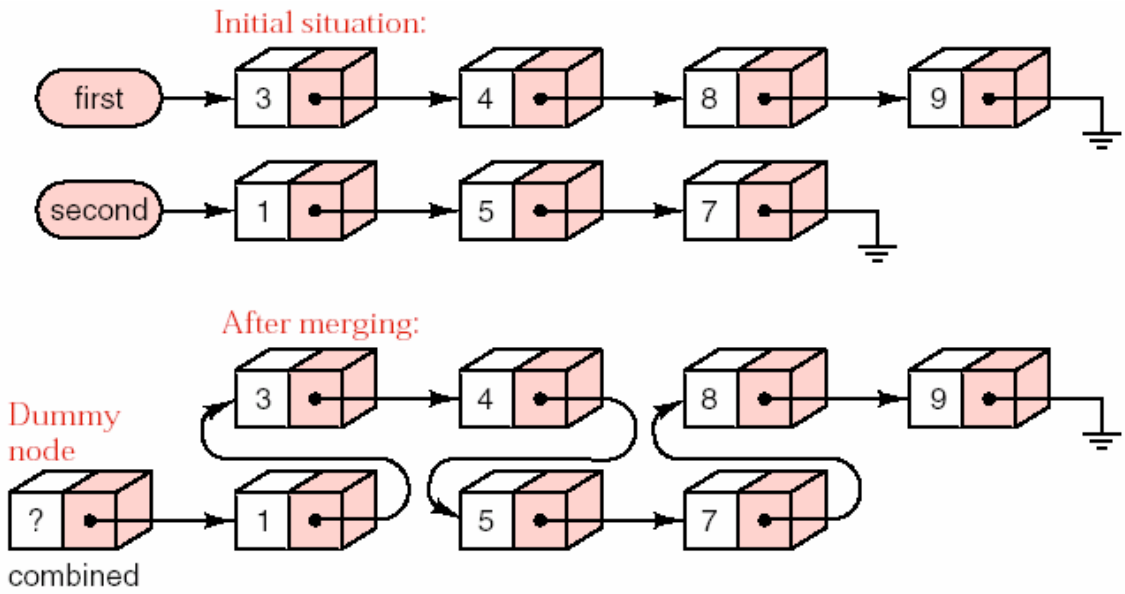
Hàm đầu tiên mà hàm **`recursive_merge_sort`** sử dụng là **`divide_from`**. Hàm này nhận vào danh sách được tham chiếu bởi **`sub_list`** và tách nó thành hai nửa bằng cách thay liên kết ở giữa danh sách bằng NULL và trả về con trỏ đến phần tử đầu tiên của phần còn lại của danh sách ban đầu. Bằng cách cho con trỏ **`midpoint`** tiến một bước và con trỏ **`position`** tiến hai bước trong mỗi lần lặp, khi **`position`** đến cuối danh sách thì **`midpoint`** dừng ngay giữa danh sách.

```
// Dành cho danh sách liên kết trong chương 4.

template <class Record>
Node<Record> *Sortable_list<Record>::divide_from(Node<Record> *sub_list)
/*
post: Số phần tử trong danh sách trỏ bởi sub_list giảm một nửa. Địa chỉ phần tử đầu trong
      danh sách các phần tử còn lại được trả về. Nếu danh sách ban đầu có số phần tử lẻ thì
      danh sách thứ nhất nhiều hơn danh sách thứ hai 1 phần tử.
*/
{
   Node<Record> *position,
                *midpoint,
                *second_half;

   if ((midpoint = sub_list) == NULL) return NULL;  //  Danh sách ban đầu rỗng.
   position = midpoint->next;
   while (position != NULL) {                //  Tìm phần tử nằm giữa danh sách.
      position = position->next;
      if (position != NULL) {
         midpoint = midpoint->next;
         position = position->next;
      }
   }
   second_half = midpoint->next;
   midpoint->next = NULL;
   return second_half;
}
```

Hàm thứ hai

```
Node<Record> *merge(Node<Record> *first, Node<Record> *second)
```

trộn hai danh sách có thứ tự không giảm trỏ bởi **`first`** và **`second`** thành một danh sách có thứ tự không giảm và trả về con trỏ đến phần tử có khoá nhỏ nhất (cũng là phần tử đầu tiên của danh sách kết quả). Hàm này duyệt đồng thời hai danh sách, so sánh một cặp khoá lấy từ hai phần tử, mỗi phần tử thuộc một trong hai danh sách nói trên, và đưa phần tử thích hợp (nhỏ hơn hoặc bằng) vào trong danh sách kết quả. Trường hợp đầu và cuối của danh sách cần phải được xử lý riêng biệt. Khi một trong hai danh sách hết trước, chúng ta chỉ cần nối phần còn lại của danh sách kia vào cuối danh sách kết quả. Cần nhắc lại rằng, đối với danh sách liên kết, cách xử lý cho phần tử đầu tiên không giống với cách xử lý cho các phần tử từ vị trí thứ hai trở đi, do có ảnh hưởng đến con trỏ đầu danh sách. Cách dễ dàng nhất là chúng ta tạo một nút tạm thời gọi là **`combined`**. Nút này được đặt ở đầu danh sách và không chứa dữ liệu thực. Với cấu trúc này, các phần tử có thể được đưa vào danh sách mà không cần phải phân biệt đâu là phần tử đầu tiên thực sự. Cuối cùng, giá trị trả về của hàm là con trỏ `next` của nút **`combined`**. Nút `combined` còn được gọi là nút giả vì nó không chứa dữ liệu thật sự mà chỉ được dùng để đơn giản hoá việc xử lý các con trỏ, nó sẽ không còn tồn tại khi hết phạm vi của phương thức `merge`.



Hình 8.11- Trôn hai danh sách đã có thứ tự.

```
// Dành cho danh sách liên kết trong chương 4.

template <class Record>
Node<Record> *Sortable_list<Record>::merge(Node<Record> *first,
                                           Node<Record> *second)
/*
pre:  first và second trỏ đến hai danh sách có thứ tự.
post: Phương thức trả về con trỏ trỏ đến danh sách các phần tử đã có thứ tự, đó là các phần
tử của hai danh sách ban đầu được trộn lại. Hai danh sách ban đầu không còn phần tử.
uses: Các phương thức của lớp Records.
*/
{
   Node<Record> *last_sorted;
   Node<Record> combined;    // Phần tử giả.
   last_sorted = &combined; // Danh sách kết quả nhận dần các phần tử từ first và
                            // second, theo thứ tự từ phần tử nhỏ đến phần tử lớn.
   while (first != NULL && second != NULL) {
      if (first->entry <= second->entry) {
         last_sorted->next = first;
         last_sorted = first;
         first = first->next;
      }
      else {
         last_sorted->next = second;
         last_sorted = second;
         second = second->next;
      }
   }
//  Nối phần còn lại của danh sách chưa hết.
   if (first == NULL)
      last_sorted->next = second;
   else
      last_sorted->next = first;
   return combined.next;
}
```

8.7. Quick_sort cho danh sách liên tục

8.7.1. Các hàm

Giải thuật Quick_sort cho danh sách liên tục cần đến một giải thuật phân hoạch danh sách thông qua việc sử dụng phần tử trụ. Giải thuật này đổi chỗ các phần tử sao cho các phần tử có khoá nhỏ hơn khoá phần tử trụ sẽ được đứng trước, kế đến là các phần tử có khoá trùng với khoá của phần tử trụ, và cuối cùng là các phần tử có khoá lớn hơn khoá của phần tử trụ. Chúng ta dùng biến **pivot_position** để lưu lại vị trí của phần tử trụ trong danh sách đã được phân hoạch.

Do các danh sách con, kết quả của phép phân hoạch, được lưu trong cùng một danh sách và theo đúng vị trí tương đối giữa chúng, nên việc gom chúng lại thành một danh sách là hoàn toàn không cần thiết và chương trình không cần phải làm thêm bất cứ điều gì.

Để có thể gọi đệ qui hàm sắp xếp cho các danh sách con, các giới hạn trên và dưới của danh sách phải là các tham số cho hàm sắp xếp. Tuy nhiên, do qui ước của chúng ta là các phương thức sắp xếp không nhận tham số, chúng ta sẽ dùng một phương thức không có tham số để gọi hàm sắp xếp đệ qui có tham số.

```
// Dành cho danh sách liên tục trong chương 4.
template <class Record>
void Sortable_list<Record>::quick_sort()
/*
post: Các phần tử trong danh sách đã được sắp theo thứ tự không giảm.
uses: recursive_quick_sort.
*/
{
   recursive_quick_sort(0, count - 1);
}
```

Hàm đệ qui thực hiện việc sắp xếp:

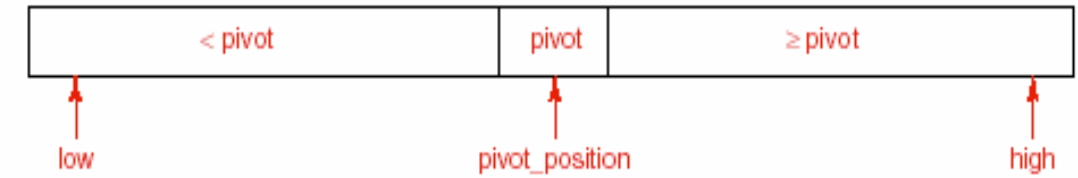
```
// Dành cho danh sách liên tục trong chương 4.
template <class Record>
void Sortable_list<Record>::recursive_quick_sort(int low, int high)
/*
pre: low và high là các vị trí hợp lệ trong Sortable_list.
post: Các phần tử trong danh sách từ chỉ số low đến chỉ số high đã được sắp theo thứ tự không
      giảm.
uses: recursive_quick_sort, partition.
*/
{
   int pivot_position;
   if (low < high) {                      //    Danh sách có nhiều hơn một phần tử.
      pivot_position = partition(low, high);
      recursive_quick_sort(low, pivot_position - 1);
      recursive_quick_sort(pivot_position + 1, high);
   }
}
```

8.7.2. Phân hoạch danh sách

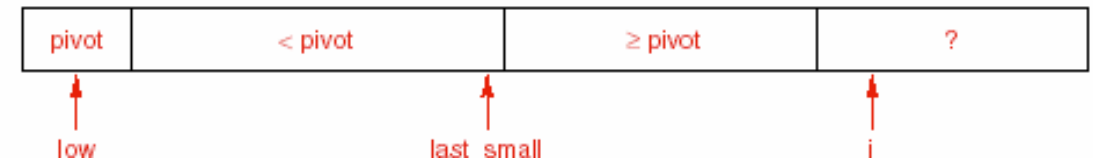
Có nhiều giải thuật để phân hoạch danh sách. Phương pháp chúng ta sử dụng ở đây rất đơn giản nhưng hiệu quả. Nó thực hiện số lần so sánh khoá nhỏ nhất có thể được.

8.7.2.1. Phát triển giải thuật

Cho giá trị của phần tử trụ, chúng ta phải bố trí lại các phần tử và tính chỉ số **pivot_position** sao cho phần tử trụ nằm tại **pivot_position**, các phần tử nhỏ hơn nằm phía bên trái và các phần tử lớn hơn nằm phía bên phải phần tử trụ. Để có thể xử lý trường hợp có nhiều hơn một phần tử có khoá đúng bằng khoá của phần tử trụ, chúng ta qui ước rằng các phần tử bên trái có khoá nhỏ hơn khoá của phần tử trụ một cách nghiêm ngặt trong khi các phần tử bên phải có khoá lớn hơn hoặc bằng khoá của phần tử trụ như trong sơ đồ sau đây.

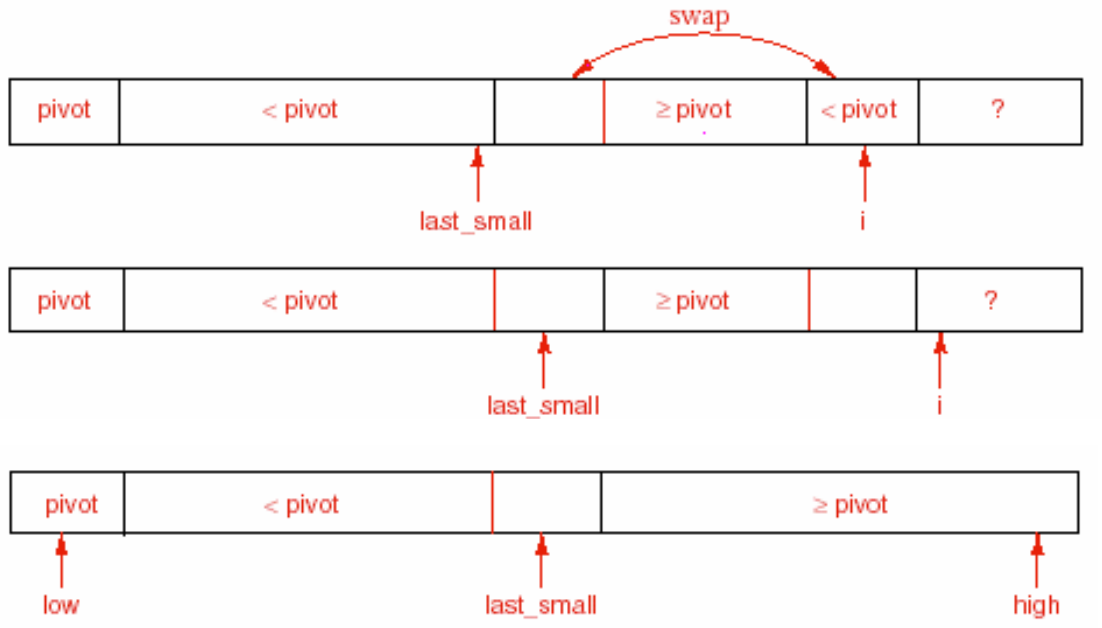


Để có được các phần tử như thế này, chúng ta phải so sánh từng phần tử với phần tử trụ bằng cách sử dụng một vòng **for** với biến chỉ số là **i**. Ngoài ra, chúng ta còn sử dụng biến **last_small** sao cho các phần tử từ **last_small** trở về trước có khoá nhỏ hơn khoá của phần tử trụ. Giả sử ban đầu phần tử trụ nằm tại đầu



danh sách. Tạm thời chúng ta sẽ giữ nguyên vị trí của nó ở đó. Khi chương trình đang ở trong thân vòng lặp, danh sách và các biến có quan hệ với nhau như sau:

Khi chương trình kiểm tra phần tử tại vị trí **i** thì có hai trường hợp xảy ra. Nếu phần tử đó vẫn lớn hơn hay bằng phần tử trụ thì biến **i** sẽ tiếp tục tăng lên và danh sách vẫn bảo toàn tính chất cần thiết. Ngược lại, chúng ta sẽ tăng **last_small** và hoán đổi hai phần tử tại **last_small** (mới) và phần tử tại **i**. Khi đó, tính chất của danh sách vẫn được duy trì. Cuối cùng, khi **i** đã đi hết chiều dài của danh sách, chúng ta chỉ cần đổi chỗ phần tử trụ từ vị trí **low** sang vị trí **last_small** là sẽ có kết quả cần thiết.



8.7.2.2. Chọn phần tử trụ

Phần tử trụ không nhất thiết phải là phần tử đầu tiên của danh sách. Chúng ta có thể chọn bất cứ phần tử nào của danh sách để làm phần tử trụ. Thực tế thì phần tử đầu tiên thường không phải là phần tử trụ tốt. Vì nếu danh sách đã có thứ tự thì không có phần tử nào nhỏ hơn phần tử trụ, do đó một trong hai danh sách con sẽ rỗng và trong trường hợp này Quick_sort trở thành "Slow_sort". Vì vậy, một lựa chọn tốt hơn cho phần tử trụ là phần tử gần giữa của danh sách với hi vọng rằng phần tử này sẽ chia hai danh sách có kích thước gần như nhau.

8.7.2.3. Viết chương trình

Với những lựa chọn này, chúng ta có được phương thức sau.

```
template <class Record>
int Sortable_list<Record>::partition(int low, int high)
/*
pre: low và high là các vị trí hợp lệ trong Sortable_list, low <= high.
post: Phần tử nằm giữa hai chỉ số low và high được chọn làm pivot. Các phần tử trong danh
      sách từ chỉ số low đến chỉ số high đã được phân hoạch như sau: các phần tử nhỏ hơn
      pivot đứng trước pivot, các phần tử còn lại đứng sau pivot. Hàm trả về vị trí của
      pivot.
uses: các phương thức của lớp Record và hàm swap(int i, int j) hoán vị hai phần tử tại vị
      trí i và j trong danh sách.
*/
{
   Record pivot;
   int i, last_small;
   swap(low, (low + high) / 2);
   pivot = entry[low];
   last_small = low;
   for (i = low + 1; i <= high; i++)
            // Đầu mỗi lần lặp, chúng ta có các điều kiện sau:
            //      If low < j <= last_small then entry[j].key < pivot.
            //      If last_small < j < i then entry[j].key >= pivot.
      if (entry[i] < pivot) {
         last_small = last_small + 1;
         swap(last_small, i);
      }
   swap(low, last_small);// Trả pivot về đúng vị trí thích hợp
   return last_small;
}
```

8.8. Heap và Heap_sort

Quick_sort có một nhược điểm là độ phức tạp trong trường hợp xấu nhất rất lớn. Thông thường thì Quick_sort chạy rất nhanh nhưng cũng có những trường hợp dữ liệu vào khiến cho Quick_sort chạy rất chậm. Trong phần này chúng ta xem xét một phương pháp sắp xếp khác khắc phục được nhược điểm này. Giải thuật này có tên là Heap_sort. Heap_sort cần O(n log n) phép so sánh và di chuyển phần tử để sắp xếp một danh sách liên tục có n phần tử, ngay cả trong trường hợp xấu nhất. Như vậy trong trường hợp xấu nhất, Heap_sort có giới hạn

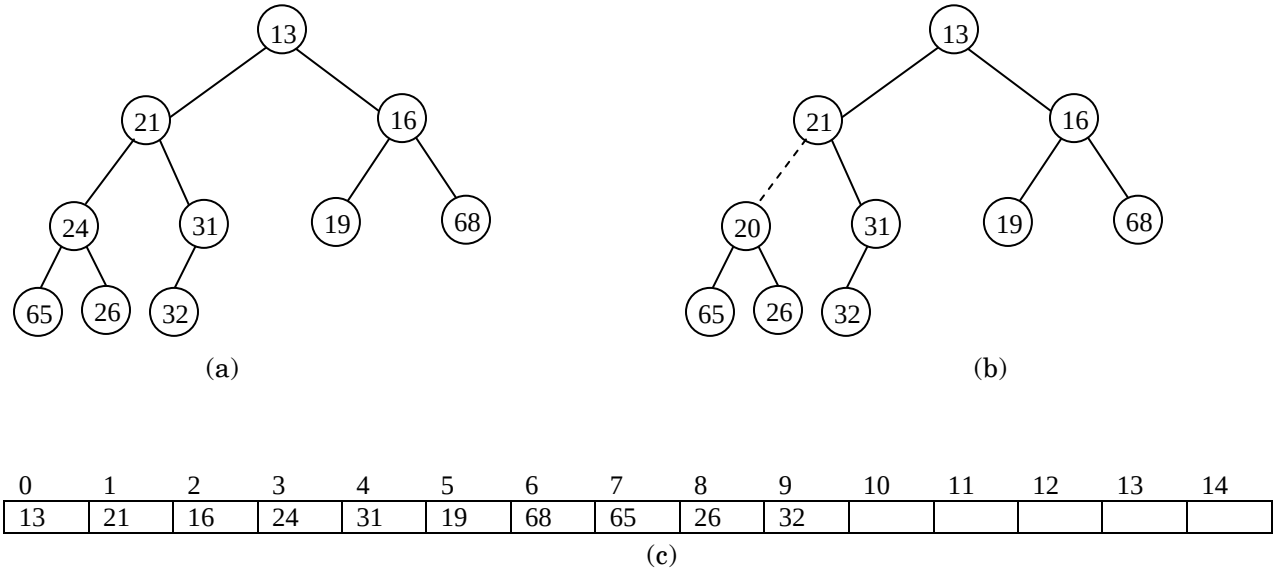
về thời gian chạy tốt hơn so với `Quick_sort`. Ngoài ra nó cũng tốt hơn `Merge_sort` trên danh sách liên tục về mặt sử dụng không gian.

Giải thuật `Heap_sort` cũng như một số hiện thực của hàng ưu tiên trong chương 11 đều dựa trên cùng một khái niệm `heap` như nhau. Đó là một cấu trúc cây tương tự như cấu trúc cấp bậc trong một tổ chức. Chúng ta thường biểu diễn cấu trúc tổ chức của một công ty nào đó bằng một cấu trúc cây. Khi giám đốc công ty nghỉ việc thì một trong hai phó giám đốc (người tốt hơn, theo một số tiêu chí nào đó) sẽ được chọn để thế chỗ và như vậy tiếp tục trống một vị trí khác. Quá trình này được lặp lại từ chỗ cao nhất trong cấu trúc cho đến chỗ thấp nhất.
Chúng ta làm quen với định nghĩa `heap` nhị phân dưới đây.

8.8.1. Định nghĩa heap nhị phân

Định nghĩa: Một **heap nhị phân** là một cấu trúc cây nhị phân với hai tính chất sau:
1. Cây đầy đủ hoặc gần như đầy đủ.
2. Khóa tại mỗi nút đều **nhỏ hơn hoặc bằng** khóa của các nút trong hai cây con của nó.

Một cây nhị phân đầy đủ hoặc gần như đầy đủ chiều cao h sẽ có từ 2^{h-1} đến 2^h-1 nút. Do đó chiều cao của nó sẽ là O(logN).
Một điều quan trọng nhất ở đây là do tính chất cây đầy đủ hoặc gần như đầy đủ nên `heap` có thể được hiện thực bằng mảng liên tục các phần tử mà không cần dùng đến con trỏ. Nếu phần tử đầu tiên của mảng có chỉ số là 0 thì, một phần tử tại vị trí i sẽ có con trái tại 2i+1 và con phải tại 2i+2, và cha của nó tại $\lfloor i-1/2 \rfloor$.



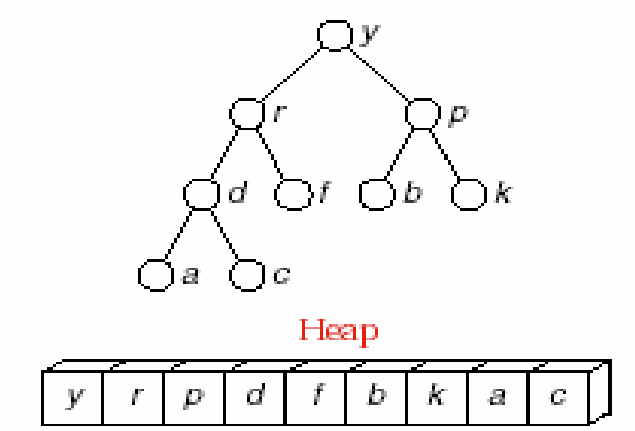
Hình 8.12 (a) Cây nhị phân gần như đầy đủ biểu diễn một `heap`.

(b) Không thỏa điều kiện của heap tại nét đứt rời.
(c) Hiện thực heap ở hình a trong một mảng liên tục.

Lưu ý rằng có khi chúng ta gọi heap được định nghĩa như trên là một min-heap, để phân biệt với trường hợp max-heap. Trong min-heap thì phần tử tại gốc là phần tử bé nhất. Max-heap sẽ sửa điều kiện thứ hai thành "Khóa tại mỗi nút đều **lớn hơn hoặc bằng** khóa của các nút trong hai cây con của nó", do đó phần tử tại gốc sẽ là phần tử lớn nhất.

Max-heap được sử dụng trong giải thuật Heap_sort.

Khi đọc xong về hàng ưu tiên, sinh viên có thể thấy rằng có thể sử dụng ngay hàng ưu tiên (nếu đã có sẵn) để phục vụ cho việc sắp xếp theo đúng ý tưởng của Heap_sort. Tuy nhiên, nếu chỉ với mục đích hiện thực một giải thuật sắp thứ tự thật hiệu quả trên một danh sách sách liên tục, thì giải thuật sắp được trình bày dưới đây đơn giản và tiết kiệm không gian hơn rất nhiều, do nó đã cố gắng chỉ sửng dụng chính phần bộ nhớ chứa các phần tử cần sắp xếp, chứ không đòi hỏi thêm vùng nhớ nào đáng kể.



Hình 8.13- Max-heap biểu diễn dưới dạng cây và dưới dạng danh sách liên tục.

Lưu ý rằng heap không phải là một danh sách có thứ tự. Tuy phần tử đầu tiên là phần tử lớn nhất của danh sách nhưng tại các vị trí k > 1 khác thì không có một thứ tự bắt buộc giữa các phần tử k và k+1. Ngoài ra, từ heap ở đây không có liên hệ gì với từ heap dùng trong việc quản lý bộ nhớ động.

8.8.2. Phát triển giải thuật Heap_sort

8.8.2.1. Phương pháp

Giải thuật Heap_sort bao gồm hai giai đoạn. Đầu tiên, các phần tử trong danh sách được sắp xếp để thoả yêu cầu của một heap. Kế đến chúng ta lập lại nhiều lần việc lấy ra phần tử đầu tiên của heap và chọn một phần tử khác lên thay thế nó.

Trong giai đoạn thứ hai, chúng ta nhận thấy rằng gốc của cây (cũng là phần tử đầu tiên của danh sách hay là đỉnh của heap) là phần tử có khóa lớn nhất. Vị trí đúng cuối cùng của phần tử này là ở cuối danh sách. Như vậy chúng ta di chuyển phần tử này về cuối danh sách, phần tử tại vị trí cuối danh sách thì được chép vào phần tử tạm **current** để nhường chỗ. Kế đến chúng ta giảm biến **last_unsorted** là ranh giới giữa phần danh sách chưa được sắp xếp với các phần tử đã được đưa về vị trí đúng (Quá trình sắp xếp tiếp theo sẽ không quan tâm đến các phần tử nằm sau **last_unsorted).** Lúc này, vị trí đầu danh sách chưa sắp xếp được xem như trống, các phần tử còn lại trong danh sách đều đang thỏa điều kiện của heap. Việc cần làm chính là tìm chỗ thích hợp để đặt phần tử đang chứa tạm trong current vào danh sách này mà vẫn duy trì tính chất của heap, trước khi chọn ra phần tử lớn nhất kế tiếp. Hàm phụ trợ **insert_heap** sẽ làm điều này.

Như vậy, theo giải thuật này, Heap_sort cần truy xuất ngẫu nhiên đến các phần tử trong danh sách. Heap_sort chỉ thích hợp với danh sách liên tục.

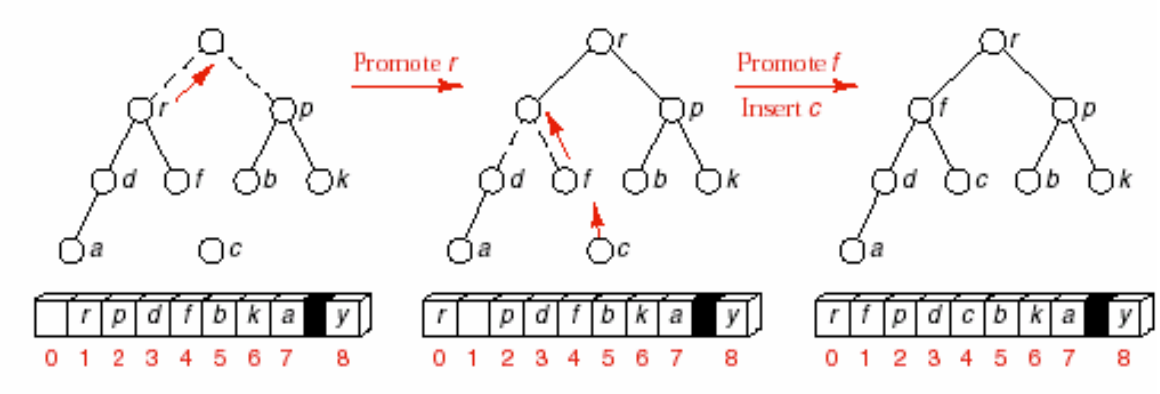
8.8.2.2. Chương trình chính

Sau đây là hàm thực hiện giải thuật Heap_sort.

```
// Dành cho danh sách liên tục trong chương 4.

template <class Record>
void Sortable_list<Record>::heap_sort()
/*
post: Các phần tử trong danh sách đã được sắp xếp theo thứ tự không giảm.
uses: build_heap và insert_heap.
*/
{
   Record current;
   int last_unsorted; // Các phần tử nằm sau last_unsorted đã có thứ tự
   build_heap();      // Giai đoạn 1: tạo heap từ danh sách các phần tử.
   for (last_unsorted = count - 1; last_unsorted > 0; last_unsorted--)
{
      current = entry[last_unsorted];
      entry[last_unsorted] = entry[0];// Mỗi lần lặp chọn được một phần tử lớn nhất.
      insert_heap(current, 0, last_unsorted - 1);  // Khôi phục heap.
   }
}
```

8.8.2.3. Ví dụ

Trước khi viết các hàm để tạo heap (**build_heap**) và đưa phần tử vào heap (**insert_heap**) chúng ta xem xét một số bước đầu tiên của quá trình sắp xếp heap trên hình.

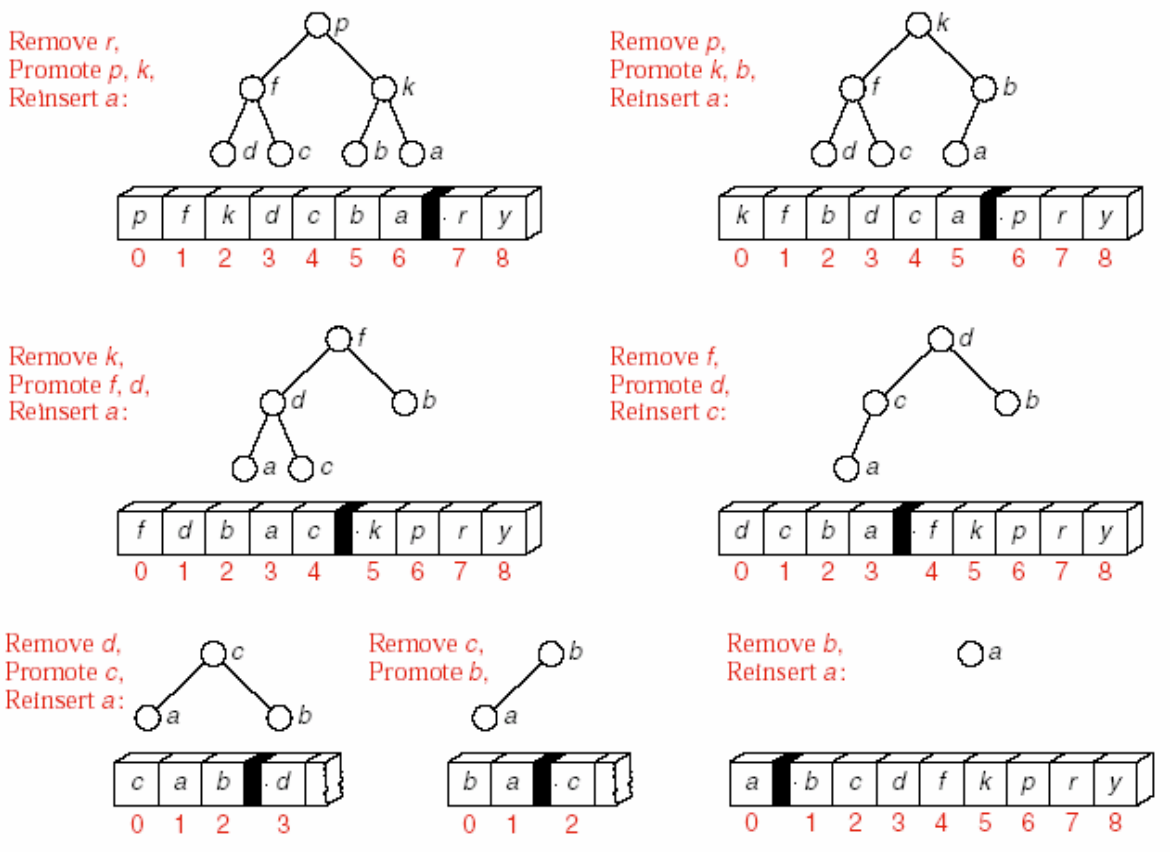


Hình 8.14 – Bước thứ nhất của `Heapsort`.

Trong bước đầu tiên, khoá lớn nhất, y, được di chuyển từ đầu đến cuối danh sách. Hình vẽ đầu tiên cho thấy kết quả của việc di chuyển, trong đó y được tách ra khỏi cây và phần tử cuối cùng trước đây, c, được đưa vào phần tử tạm current. Để tổ chức lại cây, chúng ta xem xét hai phần tử tại gốc của hai cây con. Mỗi phần tử này lớn hơn tất cả các phần tử khác trong cây con tương ứng. Do đó chúng ta chọn phần tử lớn nhất trong ba phần tử, hai phần tử gốc của hai cây con và bản thân c, làm phần tử gốc mới của toàn bộ cây. Trong ví dụ này, chúng ta sẽ đưa phần tử r lên gốc và lặp lại quá trình cho hai cây con của r. Tới đây, chúng ta sẽ dùng f để thế chỗ cho r. Tại f, vì f không có nút con cho nên c sẽ thế chỗ f và quá trình dừng.

Chúng ta thấy rằng điều kiện dừng khi tìm thấy chỗ trống thích hợp cho c thỏa tính chất của `heap` là: một chỗ trống mà không có nút con, hoặc một chỗ trống mà cả hai nút con đều <= c.

Tiếp theo, chúng ta lại có thể tách phần tử lớn nhất trong `heap` (phần tử đầu tiên) và lặp lại toàn bộ quá trình.



Hình 8. 15 – Theo vết của `Heap_sort`

8.8.2.4. Hàm insert_heap

Từ những điều trình bày ở trên ta có hàm `insert_heap` như sau.

```
// Dành cho danh sách liên tục trong chương 4.

template <class Record>
void Sortable_list<Record>::insert_heap(const Record &current, int low, int
high)
/*
pre:  Các phần tử từ chỉ số low + 1 đến high thỏa điều kiện của heap. low xem như vị trí
      còn trống (phần tử tại low sẽ bị bỏ đi).
post: Phần tử trong current thay thế cho phần tử tại low, và tất cả các phần tử từ low đến
      high được tái sắp xếp để thỏa điều kiện của heap.
uses: Lớp Record
*/
{
   int large;
   large = 2 * low + 1; //  large là vị trí con trái của low.

   while (large <= high) {
      if (large < high && entry[large] < entry[large + 1])
         large++;//large là vị trí của con có khóa lớn nhất trong 2 con của phần tử tại low.
```

```
      if (current >= entry[large])
         break;    //  low chính là vị trí có thể đặt current vào..
      else {       //  Dời phần tử con lên lấp chỗ trống.
         entry[low] = entry[large];
         low = large;
         large = 2 * low + 1;
      }
   }
   entry[low] = current;
}
```

8.8.2.5. Xây dựng heap ban đầu

Việc còn lại mà chúng ta phải làm là xây dựng heap ban đầu từ danh sách có thứ tự bất kỳ. Trước tiên, ta lưu ý rằng cây nhị phân có một số nút tự động thoả điều kiện của heap do chúng không có nút con. Đó chính là các nút lá trong cây. Do đó chúng ta không cần sắp xếp các nút lá của cây, tức là nửa sau của danh sách. Nếu chúng ta bắt đầu từ điểm giữa của danh sách và duyệt về phía đầu danh sách thì có thể sử dụng hàm insert_heap để đưa lần lượt từng phần tử vào trong heap, với lưu ý sử dụng các thông số low và high thích hợp.

```
// Dành cho danh sách liên tục trong chương 4.

template <class Record>
void Sortable_list<Record>::build_heap()
/*
post: Các phần tử trong danh sách được sắp xếp để thỏa điều kiện của heap.
uses: insert_heap.
*/
{
   int low;    //  Các phần tử từ low+1 đến cuối danh sách thỏa điều kiện của heap.
   for (low = count / 2 - 1; low >= 0; low--) {
      Record current = entry[low];
      insert_heap(current, low, count - 1);
   }
}
```

8.9. Radix Sort

Cuối cùng chúng ta xem xét một giải thuật sắp thứ tự hơi đặc biệt một chút, đó là giải thuật thường được dùng cho các phần tử có khóa là các chuỗi ký tự. Giải thuật radix_sort vốn được đưa ra trong những ngày đầu của lịch sử máy tính để sử dụng cho các thẻ đục lỗ, nhưng đã được phát triển thành một phương pháp sắp thứ tự rất hiệu quả cho các cấu trúc dữ liệu có liên kết . Ý tưởng được trình bày dưới đây cũng được xem như một ứng dụng khá thú vị của hiện thực liên kết của CTDL hàng đợi.

8.9.1. Ý tưởng

Ý tưởng của giải thuật là xét từng ký tự một và chia danh sách thành nhiều danh sách con, số danh sách con phụ thuộc vào số ký tự khác nhau có trong khóa. Giả sử các khóa là các từ gồm các chữ cái, thì chúng ta chia danh sách cần sắp thứ tự ra 26 danh sách con tại mỗi bước và phân phối các phần tử vào các danh sách con này tương ứng với một trong các ký tự có trong khóa.

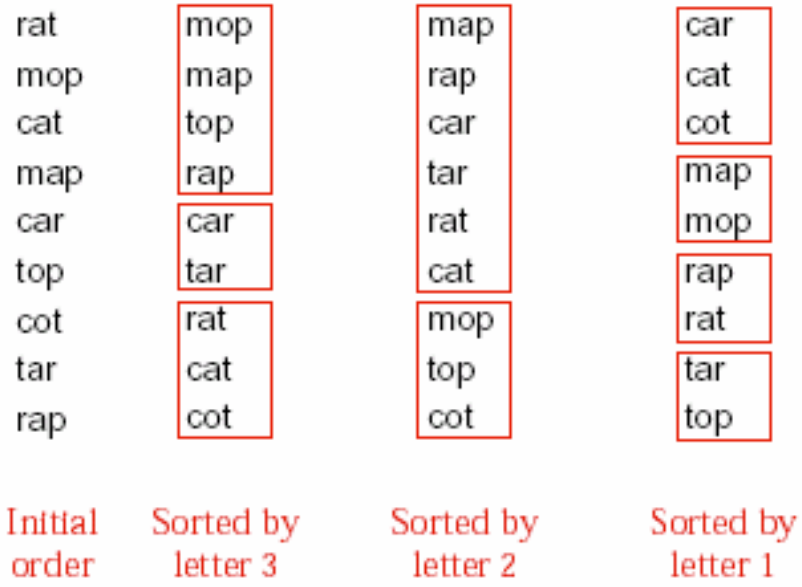
Để sắp thứ tự các từ, trước tiên người ta thường chia chúng thành 26 danh sách con theo ký tự đầu tiên, sau đó lại chia mỗi danh sách con thành 26 danh sách con theo ký tự thứ hai, và cứ thế tiếp tục. Như vậy số danh sách con sẽ trở nên quá lớn, chúng ta khó nắm giữ được. Ý tưởng dưới đây nhằm tránh số lượng danh sách con bùng nổ quá lớn. Thay vì lần lượt xét các ký tự từ trái qua phải, chúng ta sẽ xét theo thứ tự từ phải qua trái. Trước tiên nên chia các phần tử vào các danh sách con theo vị trí của ký tự cuối của từ dài nhất. Sau đó các danh sách con này lại được nối lại thành một danh sách, thứ tự các từ trong mỗi danh sách con giữ nguyên, thứ tự nối các danh sách con tuân theo thứ tự *alphabet* của các ký tự tại vị trí vừa xét. Danh sách này lại được chia theo vị trí kế trước vị trí vừa rồi, rồi lại được nối lại. Cứ thế tiếp tục cho đến khi danh sách được chia theo vị trí đầu tiên của các chuỗi ký tự và được nối lại, chúng ta sẽ có một danh sách các từ có thứ tự theo *alphabet*.

Quá trình này được minh họa qua việc sắp thứ tự 9 từ có chiều dài tối đa ba ký tự trong hình 8.16. Cột bên trái của hình vẽ là thứ tự ban đầu của các từ. Chúng được chia thành 3 danh sách tương ứng với 3 ký tự khác nhau ở vị trí cuối cùng, kết quả ở cột thứ hai của hình vẽ, mỗi hình khối biểu diễn một danh sách con. Thứ tự giữa các từ trong một danh sách con không thay đổi so với thứ tự giữa chúng trong danh sách lớn khi chưa được chia. Tiếp theo, các danh sách con được nối lại và được chia thành 2 danh sách con tương ứng 2 ký tự khác nhau ở vị trí kế cuối (vị trí thứ hai của từ) như cột thứ ba trong hình. Cuối cùng chúng được nối lại và chia thành 4 danh sách con tương ứng 4 ký tự khác nhau ở vị trí đầu của các từ. Khi các danh sách con được nối lại thì chúng ta có một danh sách đã có thứ tự.

8.9.2. Hiện thực

Chúng ta sẽ hiện thực phương pháp này trong C++ cho danh sách các mẫu tin có khóa là các chuỗi ký tự. Sau mỗi lần phân hoạch thành các danh sách con, chúng được nối lại thành một danh sách để sau đó lại được phân hoạch tiếp tương ứng với vị trí kế trước trong khóa. Chúng ta sử dụng các hàng đợi để chứa các danh sách con, do trong giải thuật, khi phân hoạch, các phần tử luôn được thêm vào cuối các danh sách con và khi nối lại thì các phần tử lại được lấy ra từ đầu các danh sách con (FIFO).

Nếu chúng ta dùng các CTDL hàng và danh sách tổng quát có sẵn để xử lý, sẽ có một số thao tác di chuyển các phần tử không cần thiết. Ngược lại, nếu sử dụng cấu trúc liên kết, có thể nối các hàng liên kết thành một danh sách kiên kết bằng cách nối `rear` của hàng này vào `front` của hàng kia, thì chương trình sẽ hiệu quả hơn rất nhiều. Quá trình này được minh họa trong hình 8.17. Chương trình `radix_sort` như vậy cần có thêm lớp dẫn xuất từ lớp `Queue` có bổ sung phương thức để nối các hàng lại với nhau, phần này có thể xem như bài tập. Phần minh họa tiếp theo chúng ta chỉ sử dụng lớp `Queue` đơn giản có sẵn.



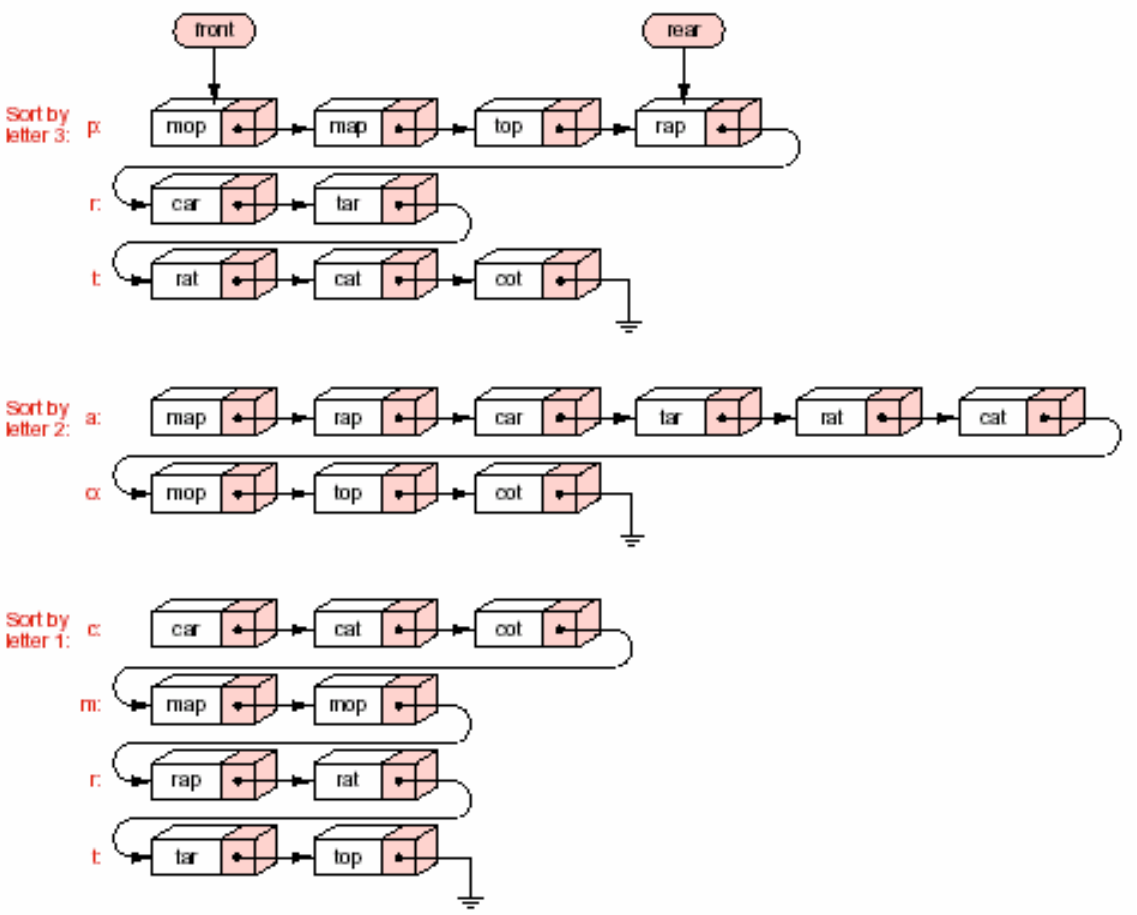
Hình 8.16 – Tiến trình của `radix_sort`.

Chúng ta sẽ dùng một mảng có 28 hàng đợi để chứa 28 danh sách con. Vị trí 0 tương ứng ký tự trống (khoảng trắng không có ký tự), các vị trí từ 1 đến 26 tương ứng các ký tự chữ cái (không phân biệt chữ hoa chữ thường), còn vị trí 27 tương ứng mọi ký tự còn lại (nếu có) xuất hiện trong khóa. Việc xử lý sẽ được lặp lại từ ký tự cuối đến ký tự đầu trong khóa, mỗi lần lặp chúng ta sẽ duyệt qua danh sách liên kết để thêm từng phần tử vào cuối mỗi danh sách con tương ứng. Sau khi danh sách đã được phân hoạch, chúng ta nối các danh sách con lại thành một danh sách. Cuối vòng lặp danh sách đã có thứ tự.

Chúng ta sẽ hiện thực **`radix_sort`** như là một phương thức của **`Sortable_list`**.

```
// Có thể sử dụng cho bất kỳ hiện thực nào của List.

template <class Record>
class Sortable_list: public List<Record> {
public:    //    Các phương thức sắp thứ tự.
   void radix_sort();

private:   //    Các hàm phụ trợ.
   void rethread(Queue queues[]);
};
```

Ở đây, lớp cơ sở `List` có thể là bất kỳ hiện thực nào của `List` trong chương 4. Hàm phụ trợ **`rethread`** sẽ được sử dụng để nối các hàng đợi.



Hình 8.17 – Radix sort liên kết

Chúng ta sẽ sử dụng các phương thức của lớp `Record`, `char` **`key_letter`**`(int` **`position`**`)` trả về ký tự tại **`position`** của khóa, hoặc trả về khoảng trắng nếu chiều dài của khóa nhỏ hơn `position`. Định nghĩa `Record` như sau:

```
class Record {
public:
   char key_letter(int position) const;
   Record();                    //  constructor mặc định.
   operator Key() const;        //  trả về khóa của phần tử.
//    Các phương thức và các thuộc tính khác của lớp.
};
```

8.9.2.1. Phương pháp sắp thứ tự radix_sort

```
const int max_chars = 28;
template <class Record>
void Sortable_list<Record>::radix_sort()
/*
post: Các phần tử của danh sách đã được sắp theo thứ tự alphabet.
uses: Các phương thức của các lớp List, Queue, và Record;
      functions position and rethread.
*/
{
   Record data;
   Queue queues[max_chars];
   for (int position = key_size - 1; position >= 0; position--) {
      //    Lặp từ vị trí cuối đến vị trí đầu các chuỗi ký tự.
      while (remove(0, data) == success) {
         int queue_number = alphabetic_order(data.key_letter(position));
         queues[queue_number].append(data); // Đưa vào hàng thích hợp.
      }
      rethread(queues);// Nối các danh sách con trong các hàng thành danh sách.
   }
}
```

Hàm này sử dụng hai hàm phụ trợ: **alphabetic_order** để xác định hàng đợi tương ứng với một ký tự cho trước, và Sortable_list::**rethread()** để nối các hàng. Chúng ta có thể dùng bất kỳ hiện thực nào của hàng tương thích với đặc tả Queue trừu tượng trong chương 3.

8.9.2.2. Chọn một queue

Hàm **alphabetic_order** lấy thứ tự của một ký tự cho trước trong bảng chữ cái, với quy ước rằng tất cả các ký tự không phải là ký tự chữ cái sẽ có cùng thứ tự là 27, khoảng trắng có thứ tự 0. Hàm không phân biệt chữ hoa và chữ thường.

```
int alphabetic_order(char c)
/*
post: Trả về thứ tự của ký tự trong  c  theo thứ tự alphabet, hoặc trả về 0 nếu c là khoảng
      trắng.
*/
{
   if (c == ' ') return 0;
   if ('a' <= c && c <= 'z') return c - 'a' + 1;
   if ('A' <= c && c <= 'Z') return c - 'A' + 1;
   return 27;
}
```

8.9.2.3. Nối các hàng

Hàm **rethread** nối 28 hàng lại thành một **Sortable_list**. Hàm cũng làm rỗng tất cả các hàng này để chúng có thể được sử dụng trong lần lặp sau của giải thuật. Hàm này có thể được viết lại theo cách phụ thuộc vào hiện thực của các hàng để giải thuật radix_sort có thể chạy nhanh hơn, chúng ta xem như bài tập.

```
template <class Record>
void Sortable_list<Record>::rethread(Queue queues[])
/*
post: Mọi hàng được nối lại thành một danh sách, các hàng trở thành rỗng.
uses: Các phương thức của các lớp List và Queue.
*/
{
   Record data;
   for (int i = 0; i < max_chars; i++)
      while (!queues[i].empty()) {
         queues[i].retrieve(data);
         insert(size(), data);
         queues[i].serve();
      }
}
```

8.9.3. Phân tích phương pháp radix_sort

Chú ý rằng thời gian để chạy radix_sort là θ(nk), n là số phần tử cần sắp thứ tự và k là số ký tự có trong khóa. Thời gian cần cho các phương pháp sắp thứ tự khác của chúng ta phụ thuộc vào n và không phụ thuộc trực tiếp vào chiều dài của khóa. Thời gian tốt nhất là đối với merge_sort: nlgn + O(n).

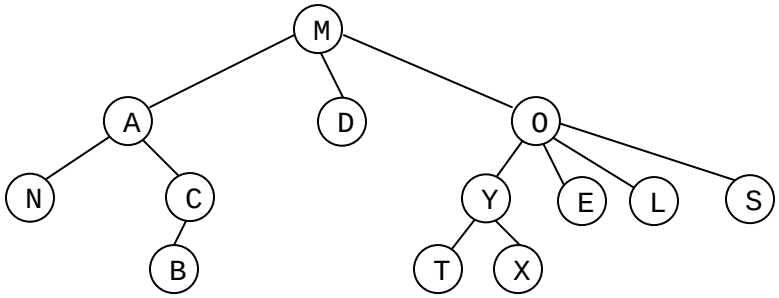
Hiệu quả tương đối của các phương pháp có liên quan đến kích thước nk và n lgn, có nghĩa là, k và lgn. Nếu các khóa có chiều dài lớn và số phần tử cần sắp thứ tự không lớn (k lớn và lg n tương đối nhỏ), thì các phương pháp khác, tựa như merge_sort, sẽ hiệu quả hơn radix_sort; ngược lại nếu k nhỏ (các khóa ngắn) và số phần tử cần sắp thứ tự nhiều, thì radix_sort sẽ nhanh hơn mọi phương pháp sắp thứ tự mà chúng ta đã biết.

Chương 9 – CÂY NHỊ PHÂN

So với hiện thực liên tục của các cấu trúc dữ liệu, các danh sách liên kết có những ưu điểm lớn về tính mềm dẻo. Nhưng chúng cũng có một điểm yếu, đó là sự tuần tự, chúng được tổ chức theo cách mà việc di chuyển trên chúng chỉ có thể qua từng phần tử một. Trong chương này chúng ta khắc phục nhược điểm này bằng cách sử dụng các cấu trúc dữ liệu cây chứa con trỏ. Cây được dùng trong rất nhiều ứng dụng, đặc biệt trong việc truy xuất dữ liệu.

9.1. Các khái niệm cơ bản về cây

Một **cây** (*tree*) - hình 9.1- gồm một tập hữu hạn các **nút** (*node*) và một tập hữu hạn các **cành** (*branch*) nối giữa các nút. Cành đi vào nút gọi là **cành vào** (*indegree*), cành đi ra khỏi nút gọi là **cành ra** (*outdegree*). Số cành ra từ một nút gọi là **bậc** (*degree*) của nút đó. Nếu cây không rỗng thì phải có một nút gọi là **nút gốc** (*root*), **nút này không có cành vào**. Cây trong hình 9.1 có M là nút gốc. Các nút còn lại, mỗi nút phải có **chính xác một cành vào**. Tất cả các nút đều có thể có 0, 1, hoặc nhiều hơn số cành ra.



(a)

```
M
-   A
-   -   N
-   -   C
-   -   -   B
-   D

-   O
-   -   Y
-   -   -   T
-   -   -   X
-   -   E
-   -   L
-   -   S
```

(b)

M (A (N C (B)) D O (Y (T X) E L S))

(c)

Hình 9.1 – Các cách biểu diễn của cây

Nút lá (*leaf*) được định nghĩa như là nút của cây mà số cành ra bằng 0. Các nút không phải nút gốc hoặc nút lá thì được gọi là **nút trung gian** hay **nút trong** (*internal node*). Nút có số cành ra khác 0 có thể gọi là **nút cha** (*parent*) của các nút mà cành ra của nó đi vào, các nút này cũng được gọi là các **nút con** (*child*) của nó. Các nút cùng cha được gọi là các **nút anh em** (*sibling*) với nhau. Nút trên nút cha có thể gọi là **nút ông** (*grandparent,* trong một số bài toán chúng ta cũng cần gọi tên như vậy để trình bày giải thuật).

Theo hình 9.1, các nút lá gồm: N, B, D, T, X, E, L, S; các nút trung gian gồm: A, C, O, Y. Nút Y là cha của hai nút T và X. T và X là con của Y, và là nút anh em với nhau.

Đường đi (*path*) từ nút n_1 đến nút n_k được định nghĩa là một dãy các nút n_1, n_2, ..., n_k sao cho n_i là nút cha của nút n_{i+1} với $1 \leq i < k$. **Chiều dài** (*length*) **đường đi** này là số cành trên nó, đó là k-1. Mỗi nút có đường đi chiều dài bằng 0 đến chính nó. Trong một cây, từ nút gốc đến mỗi nút còn lại chỉ có duy nhất một đường đi.

Đối với mỗi nút n_i, **độ sâu** (*depth*) hay còn gọi là **mức** (*level*) của nó chính là chiều dài đường đi duy nhất từ nút gốc đến nó cộng 1. Nút gốc có mức bằng 1. **Chiều cao** (*height*) **của nút** n_i là chiều dài của đường đi dài nhất từ nó đến một nút lá. Mọi nút lá có chiều cao bằng 1. **Chiều cao của cây** bằng chiều cao của nút gốc. **Độ sâu của cây** bằng độ sâu của nút lá sâu nhất, nó luôn bằng chiều cao của cây.

Nếu giữa nút n_1 và nút n_2 có một đường đi, thì n_1 đươc gọi là **nút trước** (*ancestor*) của n_2 và n_2 là **nút sau** (*descendant*) của n_1.

M là nút trước của nút B. M là nút gốc, có mức là 1. Đường đi từ M đến B là: M, A, C, B, có chiều dài là 3. B có mức là 4.

B là nút lá, có chiều cao là 1. Chiều cao của C là 2, của A là 3, và của M là 4 chính bằng chiều cao của cây.

Một cây có thể được chia thành nhiều cây con (*subtree*). Một cây con là bất kỳ một cấu trúc cây bên dưới của nút gốc. Nút đầu tiên của cây con là nút gốc của nó và đôi khi người ta dùng tên của nút này để gọi cho cây con. Cây con gốc A (hay gọi tắt là cây con A) gồm các nút A, N, C, B. Một cây con cũng có thể chia thành nhiều cây con khác. Khái niệm cây con dẫn đến định nghĩa đệ quy cho cây như sau:

Định nghĩa: Một cây là tập các nút mà
- là tập rỗng, hoặc
- có một nút gọi là nút gốc có không hoặc nhiều cây con, các cây con cũng là cây

Các cách biểu diễn cây

Thông thường có 3 cách biểu diễn cây: biểu diễn bằng đồ thị – hình 9.1a, biểu diễn bằng cách canh lề – hình 9.1b, và biểu diễn bằng biểu thức có dấu ngoặc – hình 9.1c.

9.2. Cây nhị phân

9.2.1. Các định nghĩa

Định nghĩa: Một cây nhị phân hoặc là một cây rỗng, hoặc bao gồm một nút gọi là nút gốc (*root*) và hai cây nhị phân được gọi là cây con bên trái và cây con bên phải của nút gốc.

Lưu ý rằng định nghĩa này là định nghĩa toán học cho một cấu trúc cây. Để đặc tả cây nhị phân như một kiểu dữ liệu trừu tượng, chúng ta cần chỉ ra các tác vụ có thể thực hiện trên cây nhị phân. Các phương thức cơ bản của một cây nhị phân tổng quát chúng ta bàn đến có thể là tạo cây, giải phóng cây, kiểm tra cây rỗng, duyệt cây,...

Định nghĩa này không quan tâm đến cách hiện thực của cây nhị phân trong bộ nhớ. Chúng ta sẽ thấy ngay rằng một biểu diễn liên kết là tự nhiên và dễ sử dụng, nhưng các hiện thực khác như mảng liên tục cũng có thể thích hợp. Định nghĩa này cũng không quan tâm đến các khóa hoặc cách mà chúng được sắp thứ tự. Cây nhị phân được dùng cho nhiều mục đích khác hơn là chỉ có tìm kiếm truy xuất, do đó chúng ta cần giữ một định nghĩa tổng quát.

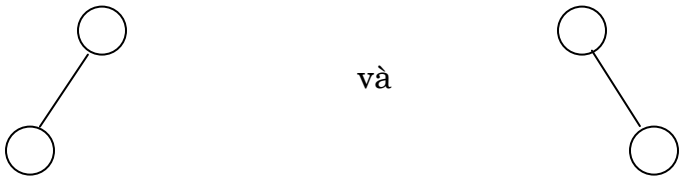
Trước khi xem xét xa hơn về các đặc tính chung của cây nhị phân, chúng ta hãy quay về định nghĩa tổng quát và nhìn xem bản chất đệ quy của nó thể hiện như thế nào trong cấu trúc của một cây nhị phân nhỏ.

Trường hợp thứ nhất, một trường hợp cơ bản không liên quan đến đệ quy, đó là một cây nhị phân rỗng.

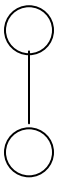
Cách duy nhất để xây dựng một cây nhị phân có một nút là cho nút đó là gốc và cho hai cây con trái và phải là hai cây rỗng.

Với cây có hai nút, một trong hai sẽ là gốc và nút còn lại sẽ thuộc cây con. Hoặc cây con trái hoặc cây con phải là cây rỗng, và cây còn lại chứa chính xác chỉ

một nút. Như vậy có hai cây nhị phân khác nhau có hai nút. Hai cây nhị phân có hai nút có thể được vẽ như sau:



và đây là hai cây khác nhau. Chúng ta sẽ không bao giờ vẽ bất kỳ một phần nào của một cây nhị phân như sau:

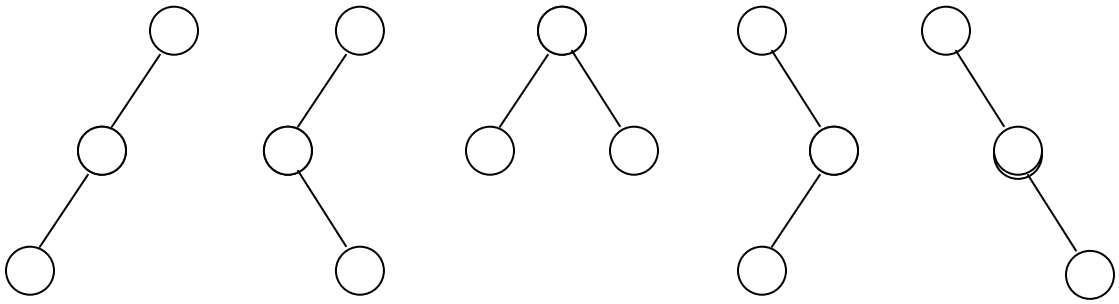


do chúng ta sẽ không thể nói được nút bên dưới là con trái hay con phải của nút trên.

Đối với trường hợp cây nhị phân có ba nút, một trong chúng sẽ là gốc, và hai nút còn lại có thể được chia giữa cây con trái và cây con phải theo một trong các cách sau:

2 + 0 1 + 1 0 + 2

Do có thể có hai cây nhị phân có hai nút và chỉ có một cây rỗng, trường hợp thứ nhất trên cho ra hai cây nhị phân. Trường hợp thứ ba, tương tự, cho thêm hai cây khác. Trường hợp giữa, cây con trái và cây con phải mỗi cây chỉ có một nút, và chỉ có duy nhất một cây nhị phân có một nút nên trường hợp này chỉ có một cây nhị phân. Tất cả chúng ta có năm cây nhị phân có ba nút:



Hình 9.2- Các cây nhị phân có ba nút

Các bước để xây dựng cây này là một điển hình cho các trường hợp lớn hơn. Chúng ta bắt đầu từ gốc của cây và xem các nút còn lại như là các cách phân chia giữa cây con trái và cây con phải. Cây con trái và cây con phải lúc này sẽ là các trường hợp nhỏ hơn mà chúng ta đã biết.

Gọi N là số nút của cây nhị phân, H là chiều cao của cây thì,

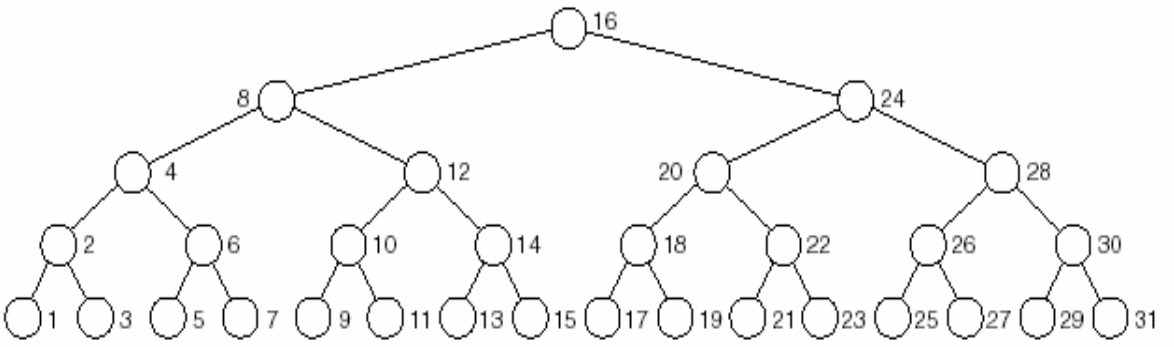
$$H_{max} = N,\ H_{min} = \lfloor \log_2 N \rfloor + 1$$
$$N_{min} = H,\ N_{max} = 2^H - 1$$

Khoảng cách từ một nút đến nút gốc xác định chi phí cần để định vị nó. Chẳng hạn một nút có độ sâu là 5 thì chúng ta phải đi từ nút gốc và qua 5 cành trên đường đi từ gốc đến nó để tìm đến nó. Do đó, nếu cây càng thấp thì việc tìm đến các nút sẽ càng nhanh. Điều này dẫn đến tính chất cân bằng của cây nhị phân. Hệ số cân bằng của cây (*balance factor*) là sự chênh lệch giữa chiều cao của hai cây con trái và phải của nó:

$$B = H_L - H_R$$

Một **cây cân bằng** khi hệ số này bằng 0 và các cây con của nó cũng cân bằng. Một cây nhị phân cân bằng với chiều cao cho trước sẽ có số nút là lớn nhất có thể. Ngược lại, với số nút cho trước cây nhị phân cân bằng có chiều cao nhỏ nhất. Thông thường điều này rất khó xảy ra nên định nghĩa có thể nới lỏng hơn với các trị B = –1, 0, hoặc 1 thay vì chỉ là 0. Chúng ta sẽ học kỹ hơn về cây cân bằng AVL trong phần sau.

Một **cây nhị phân đầy đủ** (*complete tree*) là cây có được số nút tối đa với chiều cao của nó. Đó cũng chính là cây có B=0 với mọi nút. Thuật ngữ **cây nhị phân gần như đầy đủ** cũng được dùng cho trường hợp cây có được chiều cao tối thiểu của nó và mọi nút ở mức lớn nhất dồn hết về bên trái.
Hình 9.3 biểu diễn cây nhị phân đầy đủ có 31 nút. Giả sử loại đi các nút 19, 21, 23, 25, 27, 29, 31 ta có một cây nhị phân gần như đầy đủ.



Hình 9.3 – Cây nhị phân đầy đủ với 31 nút.

9.2.2. Duyệt cây nhị phân

Một trong các tác vụ quan trọng nhất được thực hiện trên cây nhị phân là duyệt cây (*traversal*). **Một phép duyệt cây là một sự di chuyển qua khắp các nút của cây theo một thứ tự định trước, mỗi nút chỉ được xử lý một**

lần duy nhất. Cũng như phép duyệt trên các cấu trúc dữ liệu khác, hành động mà chúng ta cần làm khi ghé qua một nút sẽ phụ thuộc vào ứng dụng.

Đối với các danh sách, các nút nằm theo một thứ tự tự nhiên từ nút đầu đến nút cuối, và phép duyệt cũng theo thứ tự này. Tuy nhiên, đối với các cây, có rất nhiều thứ tự khác nhau để duyệt qua các nút.

Có 2 cách tiếp cận chính khi duyệt cây: duyệt theo chiều sâu và duyệt theo chiều rộng.

Duyệt theo chiều sâu (*defth-first traversal*): mọi nút sau của một nút con được duyệt trước khi sang một nút con khác.

Duyệt theo chiều rộng (*breadth-first traversal*): mọi nút trong cùng một mức được duyệt trước khi sang mức khác.

9.2.2.1. Duyệt theo chiều sâu

Tại một nút cho trước, có ba việc mà chúng ta muốn làm: ghé nút này, duyệt cây con bên trái, duyệt cây con bên phải. Sự khác nhau giữa các phương án duyệt là chúng ta quyết định ghé nút đó trước hoặc sau khi duyệt hai cây con, hoặc giữa khi duyệt hai cây con.

Nếu chúng ta gọi công việc ghé một nút là V, duyệt cây con trái là L, duyệt cây con phải là R, thì có đến sáu cách kết hợp giữa chúng:

VLR LVR LRV VRL RVL RLV.

Các thứ tự duyệt cây chuẩn

Theo quy ước chuẩn, sáu cách duyệt trên giảm xuống chỉ còn ba bởi chúng ta chỉ xem xét các cách mà trong đó cây con trái được duyệt trước cây con phải. Ba cách còn lại rõ ràng là tương tự vì chúng chính là những thứ tự ngược của ba cách chuẩn. Các cách chuẩn này được đặt tên như sau:

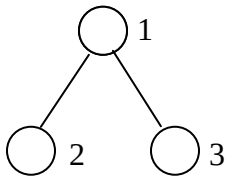
VLR	LVR	LRV
preorder	*inorder*	*postorder*

Các tên này được chọn tương ứng với bước mà nút đã cho được ghé đến. Trong phép duyệt *preorder*, nút được ghé trước các cây con; trong phép duyệt *inorder*, nó được ghé đến giữa khi duyệt hai cây con; và trong phép duyệt *postorder*, gốc của cây được ghé sau hai cây con của nó.

Phép duyệt *inorder* đôi khi còn được gọi là phép duyệt đối xứng (*symmetric order*), và *postorder* được gọi là *endorder*.

Các ví dụ đơn giản

Trong ví dụ thứ nhất, chúng ta hãy xét cây nhị phân sau:

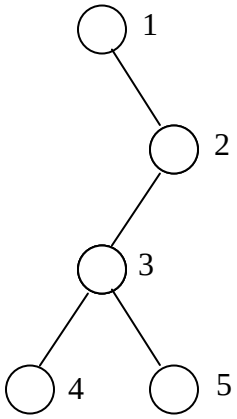


Với phép duyệt *preorder*, gốc cây mang nhãn 1 được ghé đầu tiên, sau đó phép duyệt di chuyển sang cây con trái. Cây con trái chỉ chứa một nút có nhãn là 2, nút này được duyệt thứ hai. Sau đó phép duyệt chuyển sang cây con phải của nút gốc, cuối cùng là nút mang nhãn 3 được ghé. Vậy phép duyệt preorder sẽ ghé các nút theo thứ tự 1, 2, 3.

Trước khi gốc của cây được ghé theo thứ tự *inorder*, chúng ta phải duyệt cây con trái của nó trước. Do đó nút mang nhãn 2 được ghé đầu tiên. Đó là nút duy nhất trong cây con trái. Sau đó phép duyệt chuyển đến nút gốc mang nhãn 1, và cuối cùng duyệt qua cây con phải. Vậy phép duyệt inorder sẽ ghé các nút theo thứ tự 2, 1, 3.

Với phép duyệt *postorder*, chúng ta phải duyệt các hai cây con trái và phải trước khi ghé nút gốc. Trước tiên chúng ta đi đến cây con bên trái chỉ có một nút mang nhãn 2, và nó được ghé đầu tiên. Tiếp theo, chúng ta duyệt qua cây con phải, ghé nút 3, và cuối cùng chúng ta ghé nút 1. Phép duyệt *postorder* duyệt các nút theo thứ tự 2, 3, 1.

Ví dụ thứ hai phức tạp hơn, chúng ta hãy xem xét cây nhị phân dưới đây:

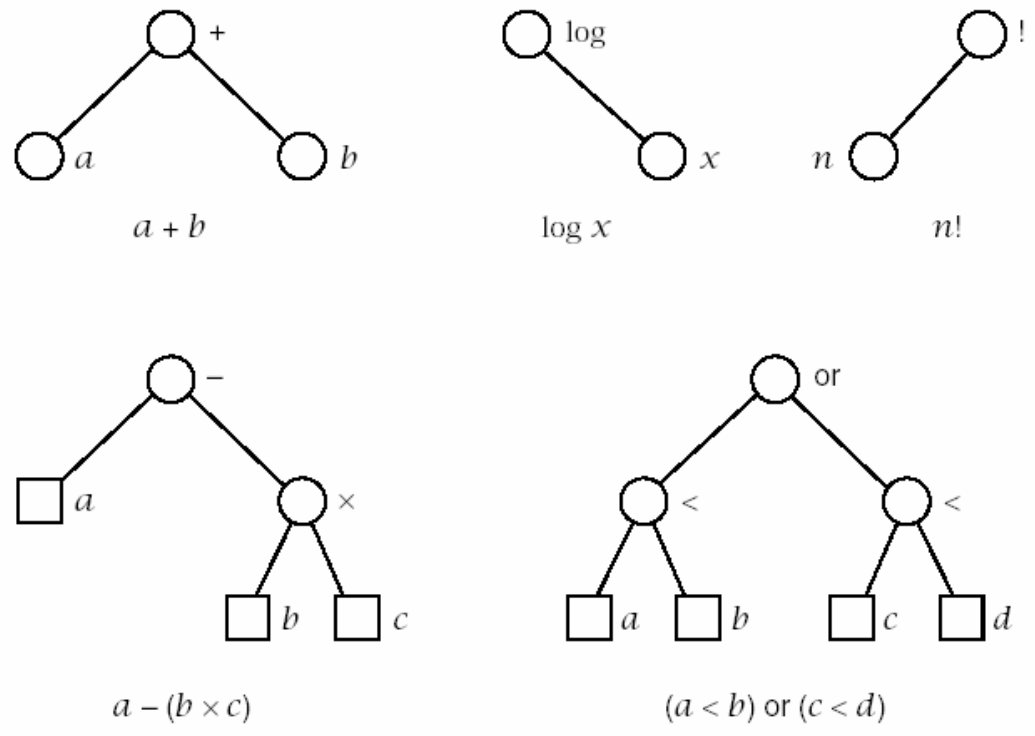


Tương tự cách làm trên chúng ta có phép duyệt *preorder* sẽ ghé các nút theo thứ tự 1, 2, 3, 4, 5. Phép duyệt *inorder* sẽ ghé các nút theo thứ tự 1, 4, 3, 5, 2. Phép duyệt *postorder* sẽ ghé các nút theo thứ tự 4, 5, 3, 2, 1.

Cây biểu thức

Cách chọn các tên *preorder*, *inorder*, và *postorder* cho ba phép duyệt cây trên không phải là tình cờ, nó liên quan chặt chẽ đến một trong những ứng dụng, đó là các cây biểu thức.

Một cây biểu thức (*expression tree*) được tạo nên từ các toán hạng đơn giản và các toán tử (số học hoặc luận lý) của biểu thức bằng cách thay thế các toán hạng đơn giản bằng các nút lá của một cây nhị phân và các toán tử bằng các nút bên trong cây. Đối với mỗi toán tử hai ngôi, cây con trái chứa mọi toán hạng và mọi toán tử thuộc toán hạng bên trái của toán tử đó, và cây con phải chứa mọi toán hạng và mọi toán tử thuộc toán hạng bên phải của nó.

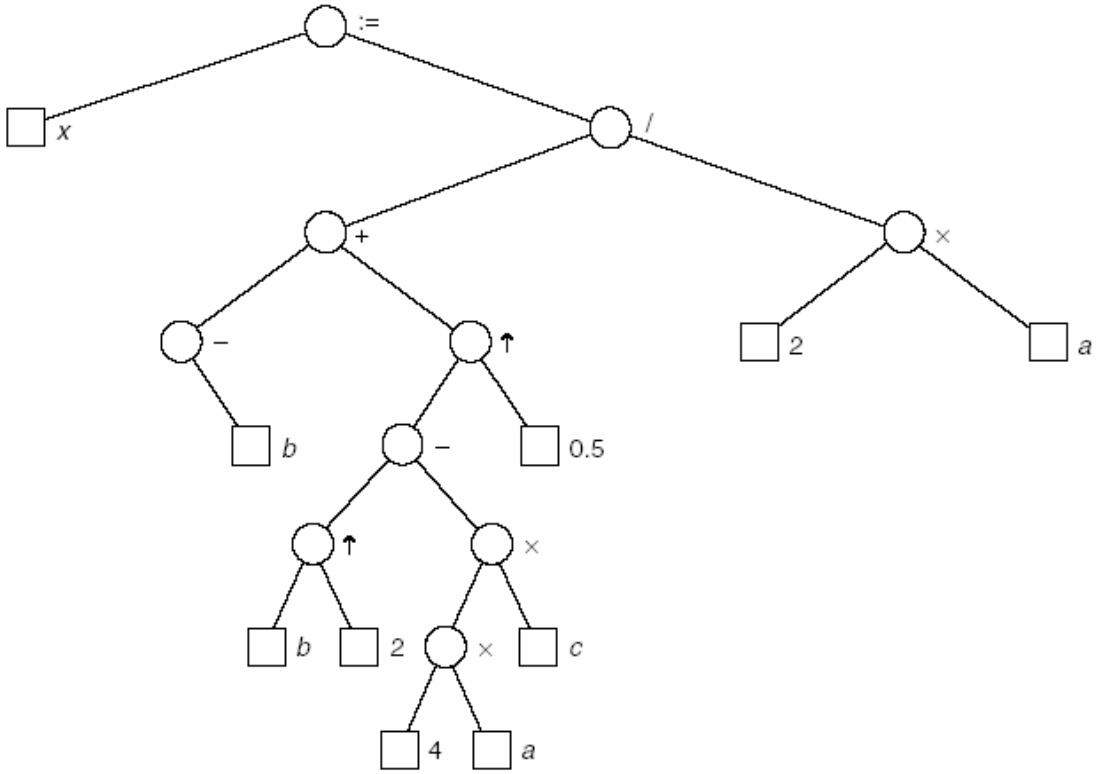


Hình 9.4 – Cây biểu thức

Đối với toán tử một ngôi, một trong hai cây con sẽ rỗng. Chúng ta thường viết một vài toán tử một ngôi phía bên trái của toán hạng của chúng, chẳng hạn dấu trừ (phép lấy số âm) hoặc các hàm chuẩn như log() và cos(). Các toán tử một ngôi khác được viết bên phải của toán hạng, chẳng hạn hàm giai thừa ()! hoặc hàm bình phương $()^2$. Đôi khi cả hai phía đều hợp lệ, như phép lấy đạo hàm có thể viết d/dx phía bên trái, hoặc ()' phía bên phải, hoặc toán tử tăng ++ có ảnh hưởng

khác nhau khi nằm bên trái hoặc nằm bên phải. Nếu toán tử được ghi bên trái, thì trong cây biểu thức nó sẽ có cây con trái rỗng, như vậy toán hạng sẽ xuất hiện bên phải của nó trong cây. Ngược lại, nếu toán tử xuất hiện bên phải, thì cây con phải của nó sẽ rỗng, và toán hạng sẽ là cây con trái của nó.

Một số cây biểu thức của một vài biểu thức đơn giản được minh họa trong hình 9.4. Hình 9.5 biểu diễn một công thức bậc hai phức tạp hơn. Ba thứ tự duyệt cây chuẩn cho cây biểu thức này liệt kê trong hình 9.6.



$$x := (-b + (b \uparrow 2 - 4 \times a \times c) \uparrow 0.5)/(2 \times a)$$

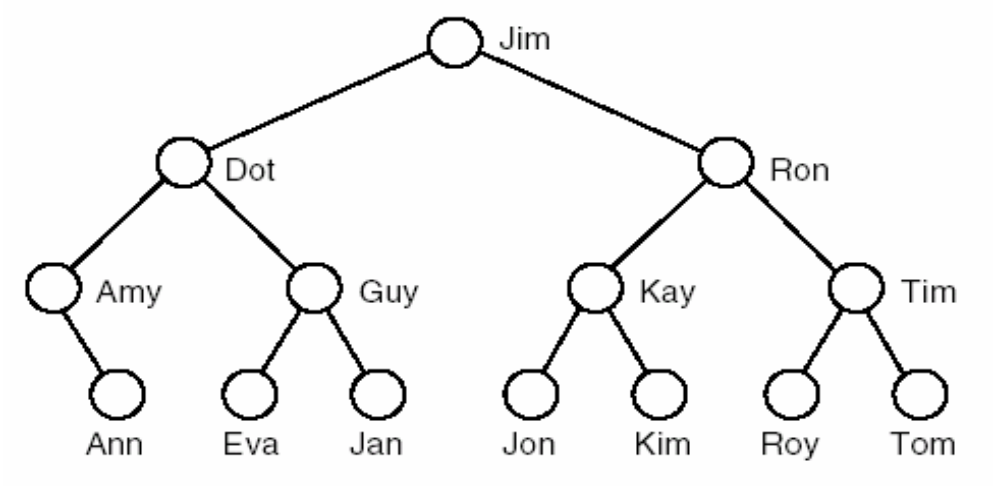
Hình 9.5 – Cây biểu thức cho công thức bậc hai.

Các tên của các phép duyệt liên quan đến các dạng Balan của biểu thức: duyệt cây biểu thức theo *preorder* là dạng *prefix*, trong đó mỗi toán tử nằm trước các toán hạng của nó; duyệt cây biểu thức theo *inorder* là dạng *infix* (cách viết biểu thức quen thuộc của chúng ta); duyệt cây biểu thức theo *postorder* là dạng *postfix*, mọi toán hạng nằm trước toán tử của chúng. Như vậy các cây con trái và cây con phải của mỗi nút luôn là các toán hạng của nó, và vị trí tương đối của một toán tử so với các toán hạng của nó trong ba dạng Balan hoàn toàn giống với thứ tự tương đối của các lần ghé các thành phần này theo một trong ba phép duyệt cây biểu thức.

Expression:	$a + b$	$\log x$	$n!$	$a - (b \times c)$	$(a < b)$ or $(c < d)$
Preorder :	$+\ a\ b$	$\log\ x$	$!\ n$	$-\ a \times b\ c$	or $<\ a\ b <\ c\ d$
Inorder :	$a + b$	$\log\ x$	$n\ !$	$a - b \times c$	$a < b$ or $c < d$
Postorder :	$a\ b\ +$	$x\ \log$	$n\ !$	$a\ b\ c \times -$	$a\ b <\ c\ d <$ or

Hình 9.6 – Các thứ tư duyệt cho cây biểu thức

Cây so sánh



Hình 9.7 – Cây so sánh để tìm nhị phân

Chúng ta hãy xem lại ví dụ trong hình 9.7 và ghi lại kết quả của ba phép duyệt cây chuẩn như sau:

preorder: Jim Dot Amy Ann Guy Eva Jan Ron Kay Jon Kim Tim Roy Tom
inorder: Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom
postorder:Ann Amy Eva Jan Guy Dot Jon Kim Kay Roy Tom Tim Ron Jim

Phép duyệt *inorder* cho các tên có thứ tự theo alphabet. Cách tạo một cây so sánh như hình 9.7 như sau: di chuyển sang trái khi khóa của nút cần thêm nhỏ hơn khóa của nút đang xét, ngược lại thì di chuyển sang phải. Như vậy cây nhị phân trên đã được xây dựng sao cho mọi nút trong cây con trái của mỗi nút có thứ tự nhỏ hơn thứ tự của nó, và mọi nút trong cây con phải có thứ tự lớn hơn nó. Do đối với mỗi nút, phép duyệt *inorder* sẽ duyệt qua các nút trong cây con trái trước, rồi đến chính nó, và cuối cùng là các nút trong cây con phải, nên chúng ta có được các nút theo thứ tự.

Trong phần sau chúng ta sẽ tìm hiểu các cây nhị phân với đặc tính trên, chúng còn được gọi là các cây nhị phân tìm kiếm (*binary search tree*), do chúng rất có ích và hiệu quả cho yêu cầu tìm kiếm.

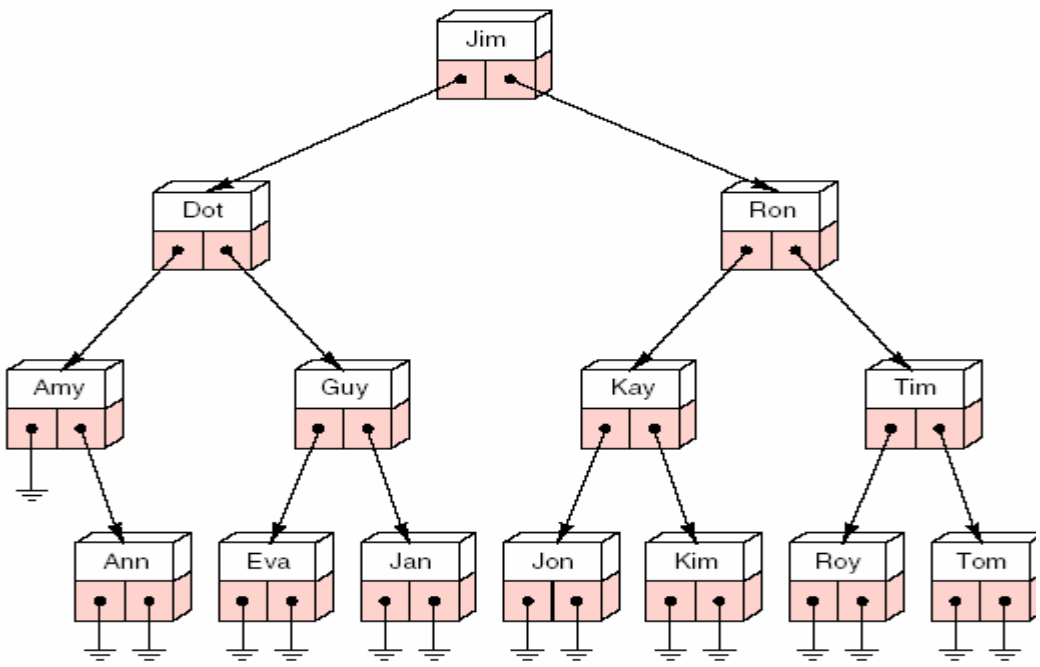
9.2.2.2. Duyệt theo chiều rộng

Thứ tự duyệt cây theo chiều rộng là thứ tự duyệt hết mức này đến mức kia, có thể từ mức cao đến mức thấp hoặc ngược lại. Trong mỗi mức có thể duyệt từ trái sang phải hoặc từ phải sang trái. Ví dụ cây trong hình 9.7 nếu duyệt theo chiều rộng từ mức thấp đến mức cao, trong mỗi mức duyệt từ trái sang phải, ta có: Jim, Dot, Ron, Amy, Guy, Kay, Tim, Ann, Eva, Jan, Jon, Kim, Roy, Tom.

9.2.3. Hiện thực liên kết của cây nhị phân

Chúng ta hãy xem xét cách biểu diễn của các nút để xây dựng nên cây.

9.2.3.1. Cấu trúc cơ bản cho một nút trong cây nhị phân



Hình 9.8 – Cây nhị phân liên kết

Mỗi nút của một cây nhị phân (cũng là gốc của một cây con nào đó) có hai cây con trái và phải. Các cây con này có thể được xác định thông qua các con trỏ chỉ đến các nút gốc của nó. Chúng ta có đặc tả sau:

```
template <class Entry>
struct Binary_node {
// Các thành phần.
   Entry data;
   Binary_node<Entry> *left;
   Binary_node<Entry> *right;
```

```
// constructors:
   Binary_node();
   Binary_node(const Entry &x);
};
```

Binary_node chứa hai *constructor* đều khởi gán các thuộc tính con trỏ là NULL mỗi khi đối tượng được tạo ra.

Trong hình 9.8, chúng ta thấy những tham chiếu NULL, tuy nhiên chúng ta có thể quy ước rằng các cây con rỗng và các cành đến nó có thể bỏ qua không cần hiển thị khi vẽ cây.

9.2.3.2. Đặc tả cây nhị phân

Một cây nhị phân có một hiện thực tự nhiên trong vùng nhớ liên kết. Cũng như các cấu trúc liên kết, chúng ta sẽ cấp phát động các nút, nối kết chúng lại với nhau. Chúng ta chỉ cần một con trỏ chỉ đến nút gốc của cây.

```
template <class Entry>
class Binary_tree {
public:
   Binary_tree();
   bool empty() const;
   void preorder(void (*visit)(Entry &));
   void inorder(void (*visit)(Entry &));
   void postorder(void (*visit)(Entry &));

   int size() const;
   void clear();
   int height() const;
   void insert(const Entry &);

   Binary_tree (const Binary_tree<Entry> &original);
   Binary_tree & operator =(const Binary_tree<Entry> &original);
   ~Binary_tree();

protected:
   //  Các hàm đệ quy phụ trợ:
   void recursive_inorder(Binary_node<Entry>*sub_root,
                                          void (*visit)(Entry &))
   void recursive_preorder(Binary_node<Entry>*sub_root,
                                          void (*visit)(Entry &))
   void recursive_postorder(Binary_node<Entry>*sub_root,
                                          void (*visit)(Entry &))

   Binary_node<Entry> *root;
};
```

Với con trỏ root, có thể dễ dàng nhận ra một cây nhị phân rỗng bởi biểu thức

```
root == NULL;
```

và khi tạo một cây nhị phân mới chúng ta chỉ cần gán root bằng NULL.

```
template <class Entry>
Binary_tree<Entry>::Binary_tree()
/*
post: Cây nhị phân rỗng được tạo ra.
*/
{
   root = NULL;
}
```

Phương thức empty kiểm tra xem một cây nhị phân có rỗng hay không:

```
template <class Entry>
bool Binary_tree<Entry>::empty() const
/*
post: Trả về true nếu cậy rỗng, ngược lại trả về false.
*/
{
   return root == NULL;
}
```

9.2.3.3. Duyệt cây

Bây giờ chúng ta sẽ xây dựng các phương thức duyệt một cây nhị phân liên kết theo cả ba phép duyệt cơ bản. Cũng như trước kia, chúng ta sẽ giả sử như chúng ta đã có hàm visit để thực hiện một công việc mong muốn nào đó cho mỗi nút của cây. Và như các hàm duyệt cho những cấu trúc dữ liệu khác, con trỏ hàm visit sẽ là một thông số hình thức của các hàm duyệt cây.

Trong các hàm duyệt cây, chúng ta cần ghé đến nút gốc và duyệt các cây con của nó. Đệ quy sẽ làm cho việc duyệt các cây con trở nên hết sức dễ dàng. Các cây con được tìm thấy nhờ các con trỏ trong nút gốc, do đó các con trỏ này cần được chuyển cho các lần gọi đệ quy. Mỗi phương thức duyệt cần gọi hàm đệ quy có một thông số con trỏ. Chẳng hạn, phương thức duyệt *inorder* được viết như sau:

```
template <class Entry>
void Binary_tree<Entry>::inorder(void (*visit)(Entry &))
/*
post: Cây được duyệt theo thứ tự inorder
uses: Hàm recursive_inorder
*/
{
   recursive_inorder(root, visit);
}
```

Một cách tổng quát, chúng ta nhận thấy một cách tổng quát rằng bất kỳ phương thức nào của Binary_tree mà bản chất là một quá trình đệ quy cũng được hiện thực bằng cách gọi một hàm đệ quy phụ trợ có thông số là gốc của cây. Hàm duyệt *inorder* phụ trợ được hiện thực bằng cách gọi đệ quy đơn giản như sau:

```
template <class Entry>
void Binary_tree<Entry>::recursive_inorder(Binary_node<Entry>*sub_root, void
                                           (*visit)(Entry &))
/*
pre:  sub_root hoặc là NULL hoặc chỉ đến gốc của một cây con.
post: Cây con được duyệt theo thứ tự inorder.
uses: Hàm recursive_inorder được gọi đệ quy.
*/
{
   if (sub_root != NULL) {
      recursive_inorder(sub_root->left, visit);
      (*visit)(sub_root->data);
      recursive_inorder(sub_root->right, visit);
   }
}
```

Các phương thức duyệt khác cũng được xây dựng một cách tương tự bằng cách gọi các hàm đệ quy phụ trợ. các hàm đệ quy phụ trợ có hiện thực như sau:

```
template <class Entry>
void Binary_tree<Entry>::recursive_preorder(Binary_node<Entry> *sub_root,
                                      void (*visit)(Entry &))
/*
pre:  sub_root hoặc là NULL hoặc chỉ đến gốc của một cây con.
post: Cây con được duyệt theo thứ tự preorder.
uses: Hàm recursive_inorder được gọi đệ quy.
*/
{
   if (sub_root != NULL) {
      (*visit)(sub_root->data);
      recursive_preorder(sub_root->left, visit);
      recursive_preorder(sub_root->right, visit);
   }
}
```

```
template <class Entry>
void Binary_tree<Entry>::recursive_postorder(Binary_node<Entry> *sub_root,
                                    void (*visit)(Entry &))
/*
pre:  sub_root hoặc là NULL hoặc chỉ đến gốc của một cây con.
post: Cây con được duyệt theo thứ tự postorder.
uses: Hàm recursive_inorder được gọi đệ quy.
*/
{
   if (sub_root != NULL) {
      recursive_postorder(sub_root->left, visit);
      recursive_postorder(sub_root->right, visit);
      (*visit)(sub_root->data);
   }
}
```

Chương trình duyệt cây theo chiều rộng luôn phải sử dụng đến CTDL hàng đợi. Nếu duyệt theo thứ tự từ mức thấp đến mức cao, mỗi mức duyệt từ trái sang phải, trước tiên nút gốc được đưa vào hàng đợi. Công việc được lặp cho đến khi

hàng đợi rỗng: lấy một nút ra khỏi hàng đợi, xử lý cho nó, đưa các nút con của nó vào hàng đợi (theo đúng thứ tự từ trái sang phải). Các biến thể khác của phép duyệt cây theo chiều rộng cũng vô cùng đơn giản, sinh viên có thể tự suy nghĩ thêm.

Chúng ta để phần hiện thực các phương thức của cây nhị phân như **`height`**, **`size`**, và **`clear`** như là bài tập. Các phương thức này cũng được hiện thực dễ dàng bằng cách gọi các hàm đệ quy phụ trợ. Trong phần bài tập chúng ta cũng sẽ viết phương thức **`insert`** để thêm các phần tử vào cây nhị phân, phương thức này cần để tạo một cây nhị phân, sau đó, kết hợp với các phương thức nêu trên, chúng ta sẽ kiểm tra lớp `Binary_tree` mà chúng ta xây dựng được.

Trong phần sau của chương này, chúng ta sẽ xây dựng các lớp dẫn xuất từ cây nhị phân có nhiều đặc tính và hữu ích hơn (các lớp dẫn xuất này sẽ có các phương thức thêm hoặc loại phần tử trong cây thích hợp với đặc tính của từng loại cây). Còn hiện tại thì chúng ta không nên thêm những phương thức như vậy vào cây nhị phân cơ bản.

Mặc dù lớp `Binary_tree` của chúng ta xuất hiện chỉ như là một lớp vỏ mà các phương thức của nó đều đẩy các công việc cần làm đến cho các hàm phụ trợ, bản thân nó lại mang một ý nghĩa quan trọng. Lớp này tập trung vào nó nhiều hàm khác nhau và cung cấp một giao diện thuận tiện tương tự các kiểu dữ liệu trừu tượng khác. Hơn nữa, chính lớp mới có thể cung cấp tính đóng kín: không có nó thì các dữ liệu trong cây không được bảo vệ một cách an toàn và dễ dàng bị thâm nhập và sửa đổi ngoài ý muốn. Cuối cùng, chúng ta có thể thấy lớp `Binary_tree` còn làm một lớp cơ sở cho các lớp khác dẫn xuất từ nó hữu ích hơn.

9.3. Cây nhị phân tìm kiếm

Chúng ta hãy xem xét vấn đề tìm kiếm một khóa trong một danh sách liên kết. Không có cách nào khác ngoài cách di chuyển trên danh sách mỗi lần một phần tử, và do đó việc tìm kiếm trên danh sách liên kết luôn là tìm tuần tự. Việc tìm kiếm sẽ trở nên nhanh hơn nhiều nếu chúng ta sử dụng danh sách liên tục và tìm nhị phân. Tuy nhiên, danh sách liên tục lại không phù hợp với sự biến động dữ liệu. Giả sử chúng ta cũng cần thay đổi danh sách thường xuyên, thêm các phần tử mới hoặc loại các phần tử hiện có. Như vậy danh sách liên tục sẽ chậm hơn nhiều so với danh sách liên kết, do việc thêm và loại phần tử trong danh sách liên tục mỗi lần đều đòi hỏi phải di chuyển nhiều phần tử sang các vị trí khác. Trong danh sách liên kết chỉ cần thay đổi một vài con trỏ mà thôi.

Vấn đề chủ chốt trong phần này chính là:

Liệu chúng ta có thể tìm một hiện thực cho các danh sách có thứ tự mà trong đó chúng ta có thể tìm kiếm, hoặc thêm bớt phần tử đều rất nhanh?

Cây nhị phân cho một lời giải tốt cho vấn đề này. Bằng cách đặt các `entry` của một danh sách có thứ tự vào trong các nút của một cây nhị phân, chúng ta sẽ thấy rằng chúng ta có thể tìm một khóa cho trước qua O(log n) bước, giống như tìm nhị phân, đồng thời chúng ta cũng có giải thuật thêm và loại phần tử trong O(log n) thời gian.

Định nghĩa: Một cây nhị phân tìm kiếm (*binary search tree* -BST) là một cây hoặc rỗng hoặc trong đó mỗi nút có một khóa (nằm trong phần dữ liệu của nó) và thỏa các điều kiện sau:
1. Khóa của nút gốc lớn hơn khóa của bất kỳ nút nào trong cây con trái của nó.
2. Khóa của nút gốc nhỏ hơn khóa của bất kỳ nút nào trong cây con phải của nó.
3. Cây con trái và cây con phải của gốc cũng là các cây nhị phân tìm kiếm.

Hai đặc tính đầu tiên mô tả thứ tự liên quan đến khóa của nút gốc, đặc tính thứ ba mở rộng chúng đến mọi nút trong cây; do đó chúng ta có thể tiếp tục sử dụng cấu trúc đệ quy của cây nhị phân. Chúng ta đã viết định nghĩa này theo cách mà nó bảo đảm rằng không có hai phần tử trong một cây nhị phân tìm kiếm có cùng khóa, do các khóa trong cây con trái chính xác là nhỏ hơn khóa của gốc, và các khóa của cây con phải cũng chính xác là lớn hơn khóa của gốc. Chúng ta có thể thay đổi định nghĩa để cho phép các phần tử trùng khóa. Tuy nhiên trong phần này chúng ta có thể giả sử rằng:

Không có hai phần tử trong một cây nhị phân tìm kiếm có trùng khóa.

Các cây nhị phân trong hình 9.7 và 9.8 là các cây nhị phân tìm kiếm, do quyết định di chuyển sang trái hoặc phải tại mỗi nút dựa trên cách so sánh các khóa trong định nghĩa của một cây tìm kiếm.

9.3.1. Các danh sách có thứ tự và các cách hiện thực

Đã đến lúc bắt đầu xây dựng các phương thức C++ để xử lý cho cây nhị phân tìm kiếm, chúng ta nên lưu ý rằng có ít nhất là ba quan điểm khác nhau dưới đây:

- Chúng ta có thể xem cây nhị phân tìm kiếm như một kiểu dữ liệu trừu tượng mới với định nghĩa và các phương thức của nó;
- Do cây nhị phân tìm kiếm là một dạng đặc biệt của cây nhị phân, chúng ta có thể xem các phương thức của nó như các dạng đặc biệt của các phương thức của cây nhị phân;
- Do các phần tử trong cây nhị phân tìm kiếm có chứa các khóa, và do chúng được gán dữ liệu để truy xuất thông tin theo cách tương tự như các danh sách có thứ tự, chúng ta có thể nghiên cứu cây nhị phân tìm kiếm như là một hiện thực mới của kiểu dữ liệu trừu tượng danh sách có thứ tự (*ordered list ADT*).

Trong thực tế, đôi khi các lập trình viên chỉ tập trung vào một trong ba quan điểm trên, và chúng ta cũng sẽ như thế. Chúng ta sẽ đặc tả lớp cây nhị phân tìm kiếm dẫn xuất từ cây nhị phân. Như vậy, lớp cây nhị phân của chúng ta lại biểu diễn cho một kiểu dữ liệu trừu tượng khác. Tuy nhiên, lớp mới sẽ thừa kế các phương thức của lớp cây nhị phân trước kia. Bằng cách này, sự sử dụng lớp thừa kế nhấn mạnh vào hai quan điểm trên. Quan điểm thứ ba thường được nhìn thấy trong các ứng dụng của cây nhị phân tìm kiếm. Chương trình của người sử dụng có thể dùng lớp của chúng ta để giải quyết các bài toán sắp thứ tự và tìm kiếm liên quan đến danh sách có thứ tự.

Chúng ta đã đưa ra những khai báo C++ cho phép xử lý cho cây nhị phân. Chúng ta sẽ sử dụng hiện thực này của cây nhị phân làm cơ sở cho lớp cây nhị phân tìm kiếm.

```
template <class Record>
class Search_tree: public Binary_tree<Record> {
public:
   Error_code insert(const Record &new_data);
   Error_code remove(const Record &old_data);
   Error_code tree_search(Record &target) const;
 private: //  Các hàm đệ quy phụ trợ.
};
```

Do lớp cây nhị phân tìm kiếm thừa kế từ lớp nhị phân, chúng ta có thể dùng lại các phương thức đã định nghĩa trên cây nhị phân tổng quát cho cây nhị phân tìm kiếm. Các phương thức này là `constructor, destructor, clear, empty, size, height`, và các phương thức duyệt `preorder, inorder, postorder`. Để thêm vào các phương thức này, một cây nhị phân tìm kiếm cần thêm các phương thức chuyên biệt hóa như `insert, remove,` và `tree_search.`

9.3.2. Tìm kiếm trên cây

Phương thức mới quan trọng đầu tiên của cây nhị phân tìm kiếm là: tìm một phần tử với một khóa cho trước trong cây nhị phân tìm kiếm liên kết. Đặc tả của phương thức như sau:

```
Error_code Search_tree<Record> :: tree_search (Record &target) const;
```
post: Nếu có một phần tử có khóa trùng với khóa trong **target**, thì **target** được chép đè bởi phần tử này, phương thức trả về `success`; ngược lại phương thức trả về `not_present.`

Ở đây chúng ta dùng lớp `Record` như đã mô tả trong chương **7.** Ngoài thuộc tính thuộc lớp `Key` dành cho khóa, trong `Record` có thể còn nhiều thành phần dữ liệu khác. Trong các ứng dụng, phương thức này thường được gọi với thông số `target` chỉ chứa trị của thành phần khóa. Nếu tìm thấy khóa cần tìm, phương thức sẽ bổ sung các dữ liệu đầy đủ vào các thành phần khác còn lại của `Record`.

9.3.2.1. Chiến lược

Để tìm một khóa, trước tiên chúng ta so sánh nó với khóa của nút gốc trong cây. Nếu so trùng, giải thuật dừng. Ngược lại, chúng ta đi sang cây con trái hoặc cây con phải và lặp lại việc tìm kiếm trong cây con này.

Ví dụ, chúng ta cần tìm tên Kim trong cây nhị phân tìm kiếm hình 9.7 và 9.8. Chúng ta so sánh Kim với phần tử tại nút gốc, Jim. Do Kim lớn hơn Jim theo thứ tự *alphabet*, chúng ta đi sang phải và tiếp tục so sánh Kim với Ron. Do Kim nhỏ hơn Jon, chúng ta di chuyển sang trái, so sánh Kim với Kay. Chúng ta lại di chuyển sang phải và gặp được phần tử cần tìm.

Đây rõ ràng là một quá trình đệ quy, cho nên chúng ta sẽ hiện thực phương thức này bằng cách gọi một hàm đệ quy phụ trợ. Liệu điều kiện dừng của việc tìm kiếm đệ quy là gì? Rõ ràng là, nếu chúng ta tìm thấy phần tử cần tìm, hàm sẽ kết thúc thành công. Nếu không, chúng ta sẽ cứ tiếp tục tìm cho đến khi gặp một cây rỗng, trong trường hợp này việc tìm kiếm thất bại.

Hàm đệ quy tìm kiếm phụ trợ sẽ trả về một con trỏ chỉ đến phần tử được tìm thấy. Mặc dù con trỏ này có thể được sử dụng để truy xuất đến dữ liệu lưu trong đối tượng cây, nhưng chỉ có các hàm là những phương thức của cây mới có thể gọi hàm tìm kiếm phụ trợ này (vì chỉ có chúng mới có thể gởi thuộc tính `root` của cây làm thông số). Như vậy, việc trả về con trỏ đến một nút sẽ không vi phạm đến tính đóng kín của cây khi nhìn từ ứng dụng bên ngoài. Chúng ta có đặc tả sau đây của hàm tìm kiếm phụ trợ.

```
Binary_node<Record> *Search_tree<Record> :: search_for_node
             (Binary_node<Record> *sub_root, const Record &target) const;
```
pre: `sub_root` hoặc là `NULL` hoặc chỉ đến một cây con của lớp `Search_tree`.
post:Nếu khóa của `target` không có trong cây con `sub_tree`, hàm trả về `NULL`; ngược lại, hàm trả về con trỏ đến nút chứa `target`.

9.3.2.2. Phiên bản đệ quy

Cách đơn giản nhất để viết hàm tìm kiếm trên là dùng đệ quy:

```
template <class Record>
Binary_node<Record> *Search_tree<Record>::search_for_node(
            Binary_node<Record>* sub_root, const Record &target) const
{
   if (sub_root == NULL || sub_root->data == target) return sub_root;
   else if (sub_root->data < target)
             return search_for_node(sub_root->right, target);
        else return search_for_node(sub_root->left, target);
}
```

9.3.2.3. Khử đệ quy

Đệ quy xuất hiện trong hàm trên chỉ là đệ quy đuôi, đó là lệnh cuối cùng được thực hiện trong hàm. Bằng cách sử dụng vòng lặp, đệ quy đuôi luôn có thể được thay thế bởi sự lặp lại nhiều lần. Trong trường hợp này chúng ta cần viết vòng lặp thế cho lệnh `if` đầu tiên, và thay đổi thông số `sub_root` để nó di chuyển xuống các cành của cây.

```
template <class Record>
Binary_node<Record> *Search_tree<Record>::search_for_node(
            Binary_node<Record> *sub_root, const Record &target) const
{  while (sub_root != NULL && sub_root->data != target)
      if (sub_root->data < target)
           sub_root = sub_root->right;
      else sub_root = sub_root->left;
   return sub_root;
}
```

9.3.2.4. Phương thức `tree_search`

Phương thức `tree_search` đơn giản chỉ gọi hàm phụ trợ `search_for_node` để tìm nút chứa khóa trùng với khóa cần tìm trong cây tìm kiếm nhị phân. Sau đó nó trích dữ liệu cần thiết và trả về `Error_code` tương ứng.

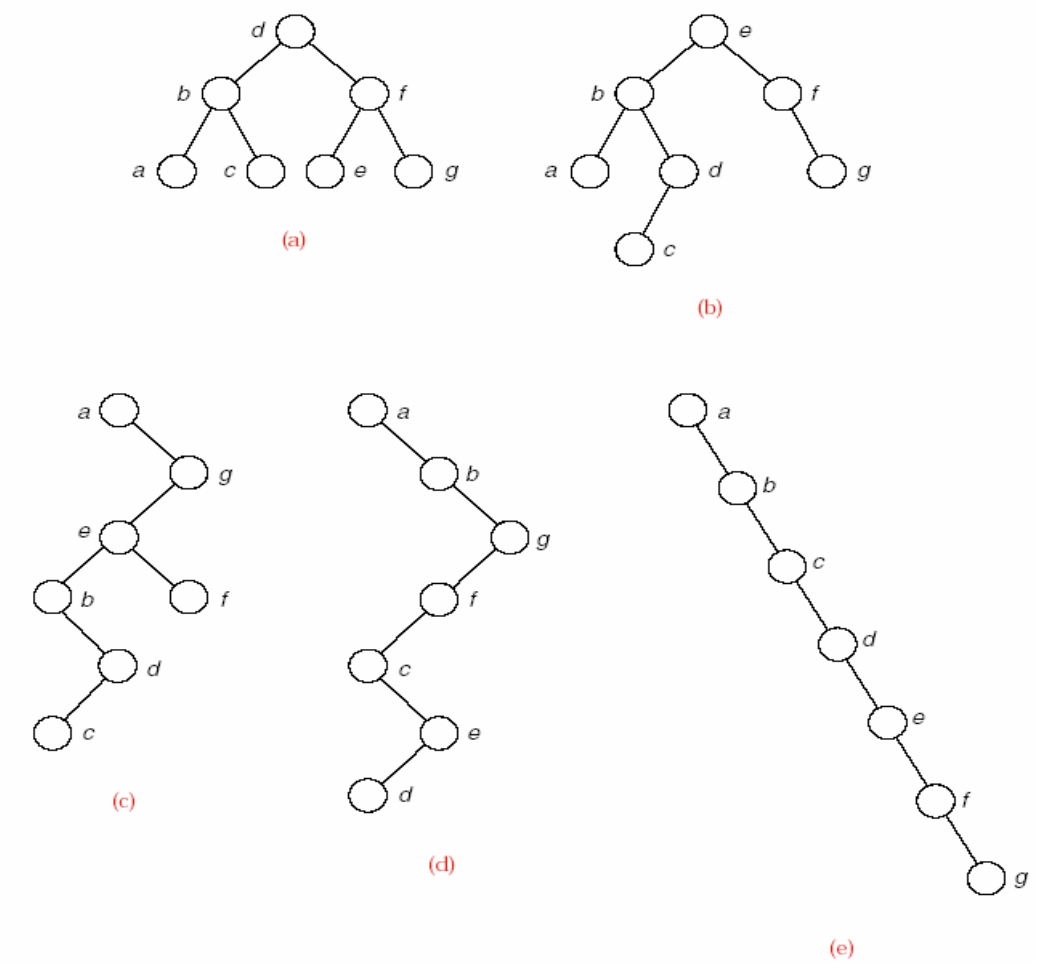
```
template <class Record>
Error_code Search_tree<Record>::tree_search(Record &target) const
/*
post: Nếu tìm thấy khóa cần tìm trong target, phương thức sẽ bổ sung các dữ liệu đầy đủ vào
      các thành phần khác còn lại của target và trả về success. Ngược lại trả về
      not_present. Cả hai trường hợp cây đều không thay đổi.
Uses: Hàm search_for_node
*/
{
   Error_code result = success;
   Binary_node<Record> *found = search_for_node(root, target);
   if (found == NULL)
      result = not_present;
   else
      target = found->data;
   return result;
}
```

9.3.2.5. Hành vi của giải thuật

Chúng ta thấy rằng `tree_search` dựa trên cơ sở của tìm nhị phân. Nếu chúng ta thực hiện tìm nhị phân trên một danh sách có thứ tự, chúng ta thấy rằng tìm nhị phân thực hiện các phép so sánh hoàn toàn giống như `tree_search`. Chúng ta cũng đã biết tìm nhị phân thực hiện O(log n) lần so sánh đối với danh sách có chiều dài n. Điều này thực sự tốt so với các phương pháp tìm kiếm khác, do log n tăng rất chậm khi n tăng.

Cây trong hình 9.9a là cây tốt nhất đối với việc tìm kiếm. Cây càng "rậm rạp" càng tốt: nó có chiều cao nhỏ nhất đối với số nút cho trước. Số nút nằm giữa nút gốc và nút cần tìm, kể cả nút cần tìm, là số lần so sánh cần thực hiện khi tìm kiếm. Vì vậy, cây càng rậm rạp thì số lần so sánh này càng nhỏ.

Không phải chúng ta luôn có thể dự đoán trước hình dạng của một cây nhị phân tìm kiếm trước khi cây được tạo ra, và cây ở hình (b) là một cây điển hình thường có nhất so với cây ở hình (a). Trong cây này, việc tìm phần tử c cần bốn lần so sánh, còn hình (a) chỉ cần ba lần so sánh. Tuy nhiên, cây ở hình (b) vẫn còn tương đối rậm rạp và việc tìm kiếm trên nó chỉ dở hơn một ít so với cây tối ưu trong hình (a).



Hình 9.9 – Một vài cây nhị phân tìm kiếm có các khóa giống nhau

Trong hình (c), cây đã trở nên suy thoái, và việc tìm phần tử c cần đến 6 lần so sánh. Hình (d) và (e) các cây đã trở thành chuỗi các mắc xích. Khi tìm trên các chuỗi mắc xích như vậy, `tree_search` không thể làm được gì khác hơn là duyệt từ phần tử này sang phần tử kia. Nói cách khác, `tree_search` khi thực hiện trên chuỗi các mắc xích như vậy đã suy thoái thành tìm tuần tự. Trong trường hợp xấu

nhất này, với một cây có n nút, tree_search có thể cần đến n lần so sánh để tìm một phần tử.

Trong thực tế, nếu các nút được thêm vào một cây nhị phân tìm kiếm treo một thứ tự ngẫu nhiên, thì rất hiếm khi cây trở nên suy thoái thành các dạng như ở hình (d) hoặc (e). Thay vào đó, cây sẽ có hình dạng gần giống với hình (a) hoặc (b). Do đó, hầu như là tree_search luôn thực hiện gần giống với tìm nhị phân. Đối với cây nhị phân tìm kiếm ngẫu nhiên, sự thực hiện tree_search chỉ chậm hơn 39% so với sự tìm kiếm tối ưu với lg n lần so sánh các khóa, và như vậy nó cũng tốt hơn rất nhiều so với tìm tuần tự có n lần so sánh.

9.3.3. Thêm phần tử vào cây nhị phân tìm kiếm

9.3.3.1. Đặt vấn đề

Tác vụ quan trọng tiếp theo đối với chúng ta là thêm một phần tử mới vào cây nhị phân tìm kiếm sao cho các khóa trong cây vẫn giữ đúng thứ tự; có nghĩa là, cây kết quả vẫn thỏa định nghĩa của một cây nhị phân tìm kiếm. Đặc tả tác vụ này như sau:

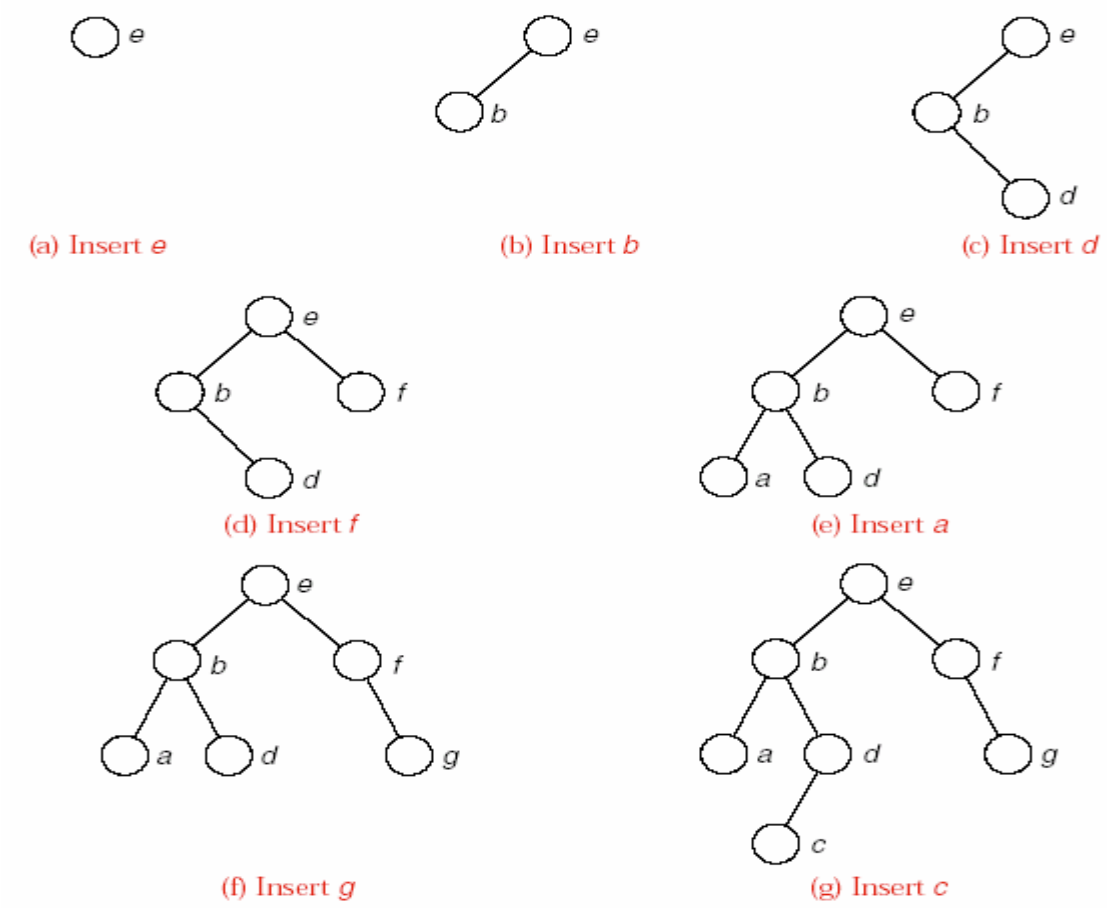
```
Error_code Search_tree<Record>::insert(const Record &new_data);
```
post: Nếu bản ghi có khóa trùng với khóa của new_data đã có trong cây thì Search_tree trả về duplicate_error. Ngược lại, new_data được thêm vào cây sao cho cây vẫn giữ được các đặc tính của một cây nhị phân tìm kiếm, phương thức trả về success.

9.3.3.2. Các ví dụ

Trước khi viết phương thức này, chúng ta hãy xem một vài ví dụ. Hình 9.10 minh họa những gì xảy ra khi chúng ta thêm các khóa e, b, d, f, a, g, c vào một cây rỗng theo đúng thứ tự này.

Khi phần tử đầu tiên e được thêm vào, nó trở thành gốc của cây như hình 9.10a. Khi thêm b, do b nhỏ hơn e, b được thêm vào cây con bên trái của e như hình (b). Tiếp theo, chúng ta thêm d, do d nhỏ hơn e, chúng ta đi qua trái, so sánh d với b, chúng ta đi qua phải. Khi thêm f, chúng ta qua phải của e như hình (d). Để thêm a, chúng ta qua trái của e, rồi qua trái của b, do a là khóa nhỏ nhất trong các khóa cần thêm vào. Tương tự, khóa g là khóa lớn nhất trong các khóa cần thêm, chúng ta đi sang phải liên tục trong khi còn có thể, như hình (f). Cuối cùng, việc thêm c, so sánh với e, rẽ sang trái, so sánh với b, rẽ phải, và so sánh với d, rẽ trái. Chúng ta có được cây ở hình (g).

Hoàn toàn có thể có một thứ tự thêm vào khác cũng tạo ra một cây nhị phân tìm kiếm tương tự. Chẳng hạn, cây ở hình 9.10 có thể được tạo ra khi các khóa



Hình 9.10 – Thêm phần tử vào cây nhị phân tìm kiếm

được thêm theo thứ tự e, f, g, b, a, d, c hoặc e, b, d, c, a, f, g hoặc một số thứ tự khác.

Có một trường hợp thật đặc biệt. Giả sử các khóa được thêm vào một cây rỗng theo đúng thứ tự tự nhiên a, b, ..., g, thì cây nhị phân tìm kiếm được tạo ra sẽ là một chuỗi các mắc xích, như hình 9.9e. Chuỗi mắc xích như vậy rất kém hiệu quả đối với việc tìm kiếm. Chúng ta có kết luận sau:

Nếu các khóa được thêm vào một cây nhị phân tìm kiếm rỗng theo thứ tự tự nhiên của chúng, thì phương thức `insert` *sẽ sinh ra một cây suy thoái về một chuỗi mắc xích kém hiệu quả. Phương thức* `insert` *không nên dùng với các khóa đã có thứ tự.*

Kết quả trên cũng đúng trong trường hợp các khóa có thứ tự ngược hoặc gần như có thứ tự.

9.3.3.3. Phương pháp

Từ ví dụ trên đến phương thức `insert` tổng quát của chúng ta chỉ có một bước nhỏ.

Trong trường hợp thứ nhất, thêm một nút vào một cây rỗng rất dễ. Chúng ta chỉ cần cho con trỏ `root` chỉ đến nút này. Nếu cây không rỗng, chúng ta cần so sánh khóa của nút cần thêm với khóa của nút gốc. Nếu nhỏ hơn, nút mới cần thêm vào cây con trái, nếu lớn hơn, nút mới cần thêm vào cây con phải. Nếu hai khóa bằng nhau thì phương thức trả về `duplicate_error`.

Lưu ý rằng chúng ta vừa mô tả việc thêm vào bằng cách sử dụng đệ quy. Sau khi chúng ta so sánh khóa, chúng ta sẽ thêm nút mới vào cho cây con trái hoặc cây con phải theo đúng phương pháp mà chúng ta sử dụng cho nút gốc.

9.3.3.4. Hàm đệ quy

Giờ chúng ta đã có thể viết phương thức `insert`, phương thức này sẽ gọi hàm đệ quy phụ trợ với thông số `root`.

```
template <class Record>
Error_code Search_tree<Record>::insert(const Record &new_data)
{
   return search_and_insert(root, new_data);
}
```

Lưu ý rằng hàm phụ trợ cần thay đổi `sub_root`, đó là trường hợp việc thêm nút mới thành công. Do đó, thông số `sub_root` phải là tham chiếu.

```
template <class Record>
Error_code Search_tree<Record>::search_and_insert(
           Binary_node<Record> *&sub_root, const Record &new_data)
{
   if (sub_root == NULL) {
      sub_root = new Binary_node<Record>(new_data);
      return success;
   }
   else if (new_data < sub_root->data)
      return search_and_insert(sub_root->left, new_data);
   else if (new_data > sub_root->data)
      return search_and_insert(sub_root->right, new_data);
   else return duplicate_error;
}
```

Chúng ta đã quy ước cây nhị phân tìm kiếm sẽ không có hai phần tử trùng khóa, do đó hàm `search_and_insert` từ chối mọi phần tử có trùng khóa.

Sự sử dụng đệ quy trong phương thức `insert` thật ra không phải là bản chất, vì đây là đệ quy đuôi. Cách hiện thực không đệ quy được xem như bài tập.

Xét về tính hiệu quả, `insert` cũng thực hiện cùng một số lần so sánh các khóa như `tree_search` đã làm khi tìm một khóa đã thêm vào trước đó. Phương thức `insert` còn làm thêm một việc là thay đổi một con trỏ, nhưng không hề thực hiện việc di chuyển các phần tử hoặc bất cứ việc gì khác chiếm nhiều thời gian. Vì thế, hiệu quả của `insert` cũng giống như `tree_search`:

Phương thức `insert` có thể thêm một nút mới vào một cây nhị phân tìm kiếm ngẫu nhiên có n nút trong O(log n) bước. Có thể xảy ra, nhưng cực kỳ hiếm, một cây ngẫu nhiên trở nên suy thoái và làm cho việc thêm vào cần đến n bước. Nếu các khóa được thêm vào một cây rỗng mà đã có thứ tự thì trường hợp suy thoái này sẽ xảy ra.

9.3.4. Sắp thứ tự theo cây

Khi duyệt một cây nhị phân tìm kiếm theo *inorder* chúng ta sẽ có được các khóa theo đúng thứ tự của chúng. Lý do là vì tất cả các khóa bên trái của một khóa đều nhỏ hơn chính nó, và các khóa bên phải của nó đều lớn hơn nó. Bằng đệ quy, điều này cũng tiếp tục đúng với các cây con cho đến khi cây con chỉ còn là một nút. Vậy phép duyệt *inorder* luôn cho các khóa có thứ tự.

9.3.4.1. Thủ tục sắp thứ tự

Điều quan sát được trên là cơ sở cho một thủ tục sắp thứ tự thú vị được gọi là *treesort*. Chúng ta chỉ cần dùng phương thức `insert` để xây dựng một cây nhị phân tìm kiếm từ các phần tử cần sắp thứ tự, sau đó dùng phép duyệt *inorder* chúng ta sẽ có các phần tử có thứ tự.

9.3.4.2. So sánh với *quicksort*

Chúng ta sẽ xem thử số lần so sánh khóa của *treesort* là bao nhiêu. Nút đầu tiên là gốc của cây, không cần phải so sánh khóa. Với hai nút tiếp theo, khóa của chúng trước tiên cần so sánh với khóa của gốc để sau đó rẽ trái hoặc phải. *Quicksort* cũng tương tự, trong đó, ở bước thứ nhất mỗi khóa cần so sánh với phần tử *pivot* để được đặt vào danh sách con bên trái hoặc bên phải. Trong *treesort*, khi mỗi nút được thêm, nó sẽ dần đi tới vị trí cuối cùng của nó trong cấu trúc liên kết. Khi nút thứ hai trở thành nút gốc của cây con trái hoặc cây con phải, mọi nút thuộc một trong hai cây con này sẽ được so sánh với nút gốc của nó. Tương tự, trong *quicksort* mọi khóa trong một danh sách con được so sánh với phần tử *pivot* của nó. Tiếp tục theo cách tương tự, chúng ta có được nhận xét sau:

Treesort có cùng số lần so sánh các khóa với quicksort.

Như chúng ta đã biết, *quicksort* là một phương pháp rất tốt. Xét trung bình, trong các phương pháp mà chúng ta đã học, chỉ có *mergesort* là có số lần so sánh

các khóa ít nhất. Do đó chúng ta có thể hy vọng rằng *treesort* cũng là một phương pháp tốt nếu xét về số lần so sánh khóa. Từ phần 8.8.4 chúng ta có thể kết luận:

Trong trường hợp trung bình, trong một danh sách có thứ tự ngẫu nhiên có n phần tử, *treesort* thực hiện

$$2n \ln n + O(n) \approx 1.39 \lg n + O(n)$$

số lần so sánh.

Treesort còn có một ưu điểm so với *quicksort*. *Quicksort* cần truy xuất mọi phần tử trong suốt quá trình sắp thứ tự. Với *treesort*, khi bắt đầu quá trình, các phần tử không cần phải có sẵn một lúc, mà chúng được thêm vào cây từng phần tử một. Do đó *treesort* thích hợp với các ứng dụng mà trong đó các phần tử được nhận vào mỗi lúc một phần tử. **Ưu điểm lớn của *treesort* là cây nhị phân tìm kiếm vừa cho phép thêm hoặc loại phần tử đi sau đó, vừa cho phép tìm kiếm theo thời gian *logarit***. Trong khi tất cả các phương pháp sắp thứ tự trước kia của chúng ta, với hiện thực danh sách liên tục thì việc thêm hoặc loại phần tử rất khó, còn với danh sách liên kết, thì việc tìm kiếm chỉ có thể là tuần tự.

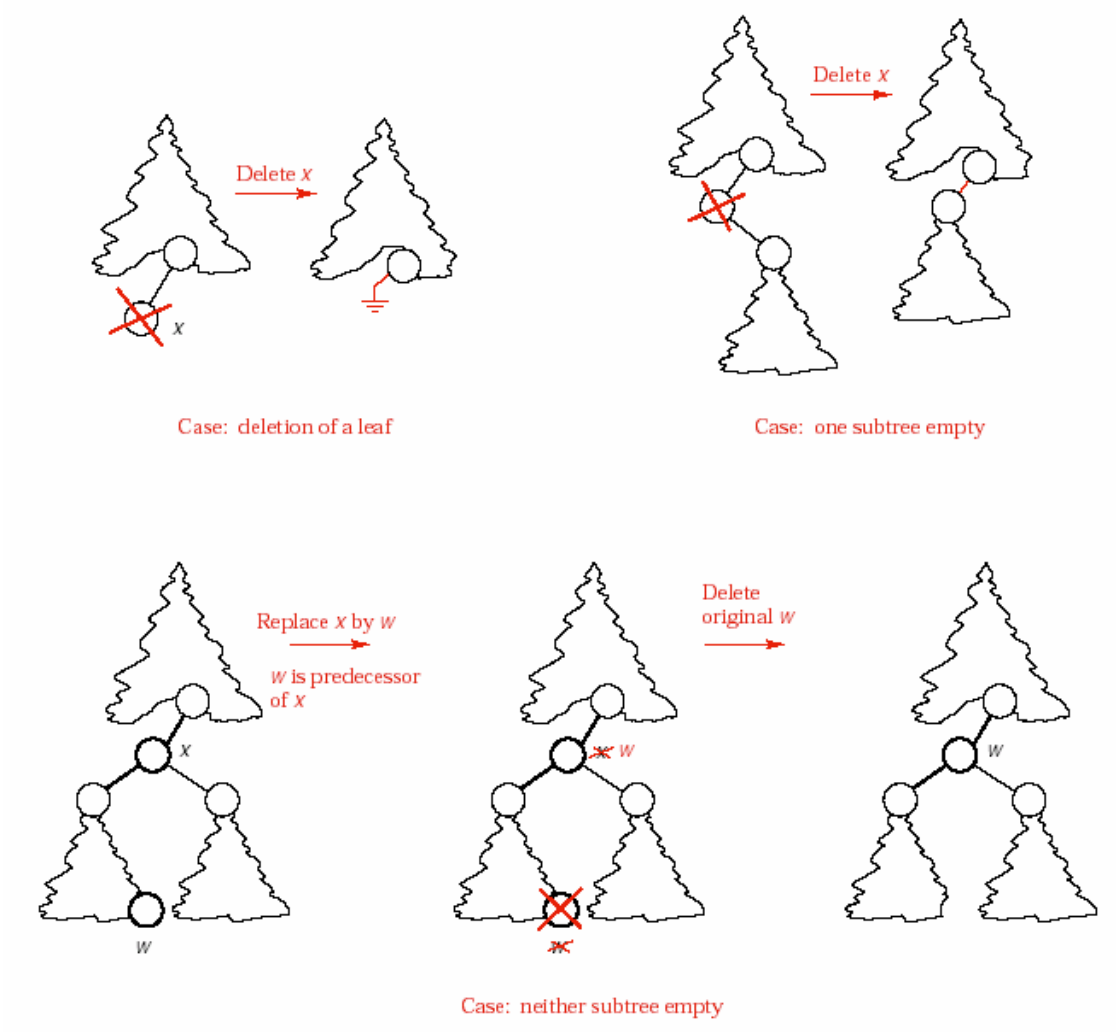
Nhược điểm chính của *treesort* được xem xét như sau. Chúng ta biết rằng *quicksort* có hiệu quả rất thấp trong trường hợp xấu nhất của nó, nhưng nếu phần tử *pivot* được chọn tốt thì trường hợp này cũng rất hiếm khi xảy ra. Khi chúng ta chọn phần tử đầu của mỗi danh sách con làm *pivot,* trường hợp xấu nhất là khi các khóa đã có thứ tự. Tương tự, nếu các khóa đã có thứ tự thì *treesort* sẽ trở nên rất dở, cây tìm kiếm sẽ suy thoái về một chuỗi các mắc xích. *Treesort* không bao giờ nên dùng với các khóa đã có thứ tự, hoặc gần như có thứ tự.

9.3.5. Loại phần tử trong cây nhị phân tìm kiếm

Khi xem xét về *treesort*, chúng ta đã nhắc đến khả năng thay đổi trong cây nhị phần tìm kiếm là một ưu điểm. Chúng ta cũng đã có một giải thuật thêm một nút vào một cây nhị phân tìm kiếm, và nó có thể được sử dụng cho cả trường hợp cập nhật lại cây cũng như trường hợp xây dựng cây từ đầu. Nhưng chúng ta chưa đề cập đến cách loại một phần tử ra khỏi cây. Nếu nút cần loại là một nút lá, thì công việc rất dễ: chỉ cần sửa tham chiếu đến nút cần loại thành `NULL` (sau khi đã giải phóng nút đó). Công việc cũng vẫn dễ dàng khi nút cần loại chỉ có một cây con khác rỗng: tham chiếu từ nút cha của nút cần loại được chỉ đến cây con khác rỗng đó.

Khi nút cần loại có đến hai cây con khác rỗng, vấn đề trở nên phức tạp hơn nhiều. Cây con nào sẽ được tham chiếu từ nút cha? Đối với cây con còn lại cần phải làm như thế nào? Hình 9.11 minh họa trường hợp này. Trước tiên, chúng ta

cần tìm nút ngay kế trước nút cần loại trong phép duyệt *inorder* (còn gọi là nút cực phải của cây con trái) bằng cách đi xuống nút con trái của nó và sau đó đi về bên phải liên tiếp nhiều lần cho đến khi không thể đi được nữa. Nút cực phải của cây con trái này sẽ không có nút con bên phải, cho nên nó có thể được loại đi một cách dễ dàng. Như vậy dữ liệu của nút cần loại sẽ được chép đè bởi dữ liệu của nút này, và nút này sẽ được loại đi. Bằng cách này cây vẫn còn giữ được đặc tính của cây nhị phân tìm kiếm, do giữa nút cần loại và nút ngay kế trước nó trong phép duyệt *inorder* không còn nút nào khác, và thứ tự duyệt *inorder* vẫn không bị xáo trộn. (Cũng có thể làm tương tự khi chọn để loại nút ngay kế sau của nút cần loại - nút cực trái của cây con phải - sau khi chép dữ liệu của nút này lên dữ liệu của nút cần loại).



Hình 9.11 – Loại một phần tử ra khỏi cây nhị phân tìm kiếm

Chúng ta bắt đầu bằng một hàm phụ trợ sẽ loại đi một nút nào đó trong cây nhị phân tìm kiếm. Hàm này có thông số là địa chỉ của nút cần loại. Thông số này phải là tham biến để việc thay đổi nó làm thay đổi thực sự con trỏ được gởi làm thông số. Ngoài ra, mục đích của hàm là cập nhật lại cây nên trong chương trình gọi, thông số thực sự phải là một

trong các tham chiếu đến chính một nút của cây, chứ không phải chỉ là một bản sao của nó. Nói một cách khác, nếu nút con trái của nút x cần bị loại thì hàm sẽ được gọi như sau

```
remove_root(x->left),
```

nếu chính root cần bị loại thì hàm sẽ gọi

```
remove_root(root).
```

Cách gọi sau đây không đúng do khi y thay đổi, x->left không hề thay đổi:

```
y = x->left; remove_root(y);
```

Hàm phụ trợ remove_root được hiện thực như sau:

```
template <class Record>
Error_code Search_tree<Record>::remove_root(Binary_node<Record>
                                                        *&sub_root)
/*
pre: sub_root là NULL, hoặc là địa chỉ của nút gốc của một cây con mà nút gốc này cần được
     loại khỏi cây nhị phân tìm kiếm.
post: Nếu sub_root là NULL, hàm trả về not_present. Ngược lại, gốc của cây con này sẽ
     được loại sao cho cây còn lại vẫn là cây nhị phân tìm kiếm. Thông số sub_root được gán
     lại gốc mới của cây con, hàm trả về success.
*/
{
   if (sub_root == NULL) return not_present;
   Binary_node<Record> *to_delete = sub_root;  // Nhớ lại nút cần loại.
   if (sub_root->right == NULL)
      sub_root = sub_root->left;
   else if (sub_root->left == NULL)
      sub_root = sub_root->right;
   else {                          // Cả 2 cây con đều rỗng.
      to_delete = sub_root->left;  // Về bên trái để đi tìm nút đứng ngay trước nút cần
                                   loại trong thứ tự duyệt inorder..

      Binary_node<Record> *parent = sub_root;
      while (to_delete->right != NULL) {     // to_delete sẽ đến được nút
         parent = to_delete;                 // cần tìm và parent sẽ là
         to_delete = to_delete->right;       // nút cha của nó.
      }
      sub_root->data = to_delete->data;      // Chép đè lên dữ liệu cần loại.

      if (parent == sub_root)                // Trường hợp đặc biệt: nút con
           sub_root->left = to_delete->left; // trái của nút cần loại cũng
                                             // chính là nút đứng ngay trước
                                             // nó trong thứ tự duyệt inorder.
      else parent->right = to_delete->left;
   }
   delete to_delete; // Loại phần tử cực phải của cây con trái của phần tử cần loại.
   return success;
}
```

Chúng ta cần phải cẩn thận phân biệt giữa trường hợp nút ngay trước nút cần loại trong thứ tự duyệt *inorder* là chính nút con trái của nó với trường hợp chúng ta cần phải di chuyển về bên phải để tìm. Trường hợp thứ nhất là trường hợp đặc biệt, nút con trái của nút cần loại có cây con bên phải rỗng. Trường hợp thứ hai là trường hợp tổng quát hơn, nhưng cũng cần lưu ý là chúng ta đi tìm nút có cây con phải là rỗng chứ không phải tìm một cây con rỗng.

Phương thức **remove** dưới đây nhận thông số là dữ liệu của nút cần loại chứ không phải con trỏ chỉ đến nó. Để loại một nút, việc đầu tiên cần làm là đi tìm nó trong cây. Chúng ta kết hợp việc tìm đệ quy trong cây với việc loại bỏ như sau:

```
template <class Record>
Error_code Search_tree<Record>::remove(const Record &target)
/*
post: Nếu tìm được dữ liệu có khóa trùng với khóa trong target thì dữ liệu đó sẽ bị loại khỏi
      cây sao cho cây vẫn là cây nhị phân tìm kiếm, phương thức trả về success. Ngược lại,
      phương thức trả về not_present.
uses: Hàm search_and_destroy
*/
{
   return search_and_destroy(root, target);
}
```

Như thường lệ, phương thức trên gọi một hàm đệ quy phụ trợ có thông số là con trỏ root.

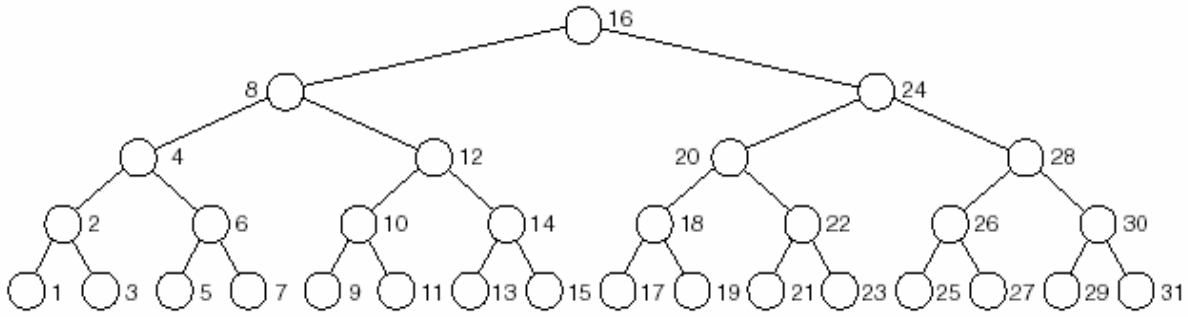
```
template <class Record>
Error_code Search_tree<Record>::search_and_destroy(
           Binary_node<Record>* &sub_root, const Record &target)
/*
pre: sub_root là NULL hoặc là địa chỉ của gốc của một cây con của cây nhị phân tìm kiếm.
post: Nếu khóa trong target không có trong cây con sub_root, hàm trả về not_present.
     Ngược lại, nút có chứa dữ liệu tìm thấy sẽ được loại sao cho tính chất cây nhị phân tìm kiếm
     vẫn đượcbảo toàn, hàm trả về success.
uses: Hàm search_and_destroy (một cách đệ quy) và hàm remove_root.
*/
{
   if (sub_root == NULL || sub_root->data == target)
      return remove_root(sub_root);
   else if (target < sub_root->data)
      return search_and_destroy(sub_root->left, target);
   else
      return search_and_destroy(sub_root->right, target);
}
```

9.4. Xây dựng một cây nhị phân tìm kiếm

Giả sử chúng ta có một danh sách các dữ liệu đã có thứ tự, hoặc có thể là một file các bản ghi có các khóa đã có thứ tự. Nếu chúng ta muốn sử dụng các dữ liệu

này để tìm kiếm thông tin, hoặc thực hiện một số thay đổi nào đó, chúng ta có thể tạo một cây nhị phân tìm kiếm từ danh sách hoặc file này.

Chúng ta có thể bắt đầu từ một cây rỗng và đơn giản sử dụng giải thuật thêm vào cây để thêm từng phần tử. Tuy nhiên do các phần tử đã có thứ tự, cây tìm kiếm của chúng ta sẽ trở thành một chuỗi các mắc xích rất dài, và việc sử dụng nó trở nên rất chậm chạp với tốc độ của tìm tuần tự chứ không phải là tìm nhị phân. Thay vào đó chúng ta mong muốn rằng các phần tử sẽ được xây dựng thành một cây càng rậm rạp càng tốt, có nghĩa là cây không bị cao quá, để giảm thời gian tạo cây cũng như thời gian tìm kiếm. Chẳng hạn, khi số phần tử n bằng 31, chúng ta muốn cây sẽ có được dạng như hình 9.12. Đây là cây có được sự cân bằng tốt nhất giữa các nhánh, và được gọi là cây nhị phân đầy đủ.



Hình 9.12 – Cây nhị phân đầy đủ với 31 nút.

Trong hình 9.12, các phần tử được đánh số theo thứ tự mà giá trị của chúng tăng dần. Đây cũng là thứ tự tự duyệt cây *inorder*, và cũng là thứ tự mà chúng sẽ được thêm vào cây theo giải thuật của chúng ta. Chúng ta cũng sẽ dùng các con số này như là nhãn của các nút trong cây. Nếu chúng ta xem xét kỹ sơ đồ trên, chúng ta có thể nhận thấy một đặc tính quan trọng của các nhãn. Các nhãn của các nút lá chỉ toàn số lẻ. Các nhãn của các nút ở mức trên các nút lá một bậc là 2, 6, 10, 14, 18, 22, 26, 30. Các số này đều gấp đôi một số lẻ, có nghĩa chúng đều là số chẵn, và chúng đều không chia hết cho 4. Trên mức cao hơn một bậc các nút có nhãn 4, 12, 20 và 28, đây là những con số chia hết cho 4, nhưng không chia hết cho 8. Cuối cùng, các nút ngay dưới nút gốc có nhãn là 8 và 24. và nút gốc có nhãn là 16.

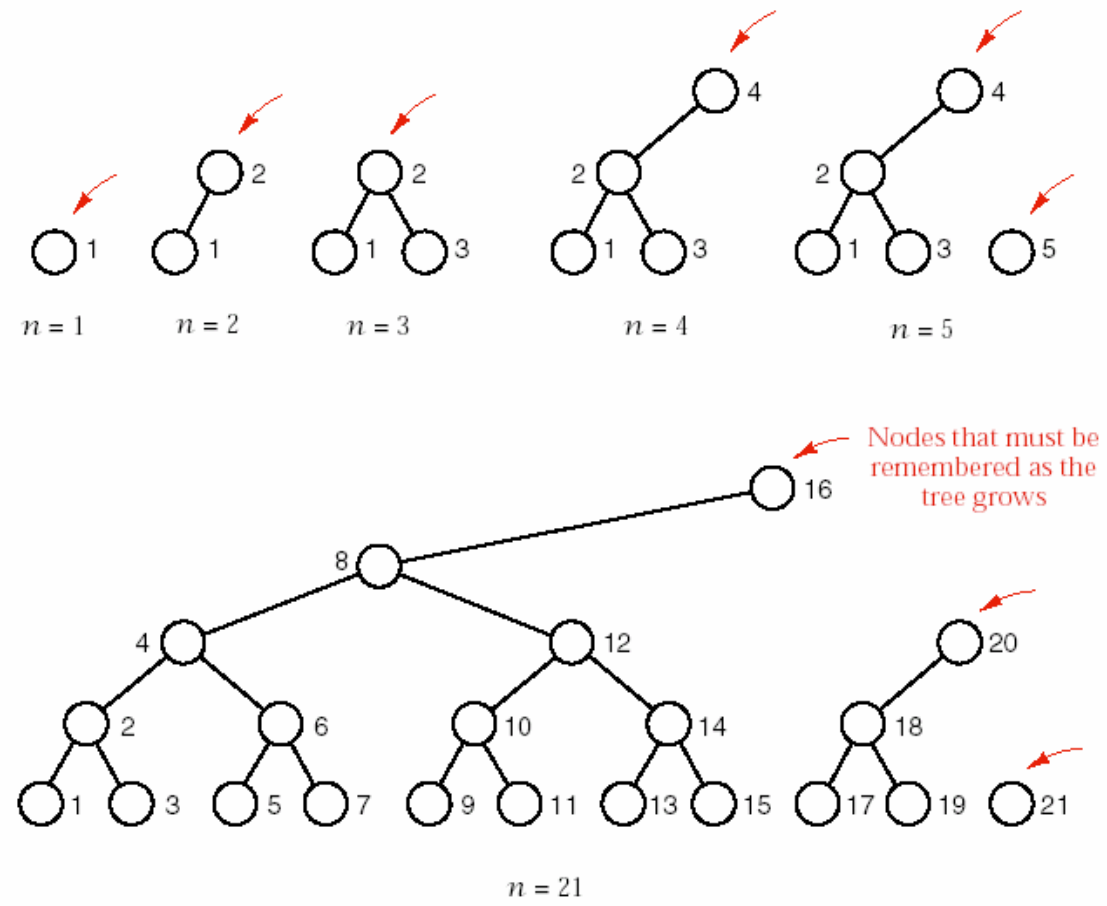
Nếu các nút của một cây nhị phân đầy đủ có các nhãn theo thứ tự duyệt inorder, bắt đầu từ 1, thì nhãn của mỗi nút là một số có số lần chia chẵn cho 2 bằng với hiệu giữa mức của nó với mức của các nút lá.

Như vậy, nếu cho mức của các nút lá là 1, khi thêm một nút mới, dựa vào nhãn của nó chúng ta sẽ tính được mức tương ứng.

Giả sử chúng ta không biết trước số nút sẽ tạo cây. Điều này được giải thích rằng, khi các nút đến từ một file hoặc một danh sách liên kết, chúng ta không có cách gì thuận tiện để đếm trước số nút cả. Điều giả thiết này cũng còn một ưu điểm là chúng ta không cần phải lo lắng về việc số nút có phải là một số lũy thừa của 2 trừ đi 1 hay không. Trong trường hợp số nút không phải là số lũy thừa của 2 trừ đi 1, cây được tạo ra sẽ không phải là một cây đầy đủ và đối xứng hoàn toàn như hình 9.12. Với giả thiết cây sẽ là một cây đầy đủ, chúng ta sẽ đưa dần các nút vào cây, cho đến khi mọi phần tử đã được đưa vào cây, chúng ta sẽ xác định cách để kết thúc việc tạo cây.

9.4.1. Thiết kế giải thuật

Khi nhận phần tử thứ nhất có nhãn là 1, chúng ta sẽ tạo một nút lá có các con trỏ left và right đều là `NULL`. Nút số 2 nằm trên nút 1, như hình 9.13. Do nút 2 nắm giữ nút 1, bằng cách nào đó chúng ta cần nhớ địa chỉ nút 1 cho đến khi có nút 2. Nút số 3 lại là nút lá, nhưng nó là nút con phải của nút 2, vậy chúng ta cần nhớ lại địa chỉ nút 2



Hình 9.13 – Tạo các nút đầu tiên cho một cây nhị phân tìm kiếm

Làm như vậy liệu chúng ta có cần phải nắm giữ một danh sách các con trỏ đến tất cả các nút đã được đưa vào cây, để sau đó chúng mới được gắn vào con trỏ

left hoặc right của cha chúng khi cha chúng xuất hiện sau hay không? Câu trả lời là không. Do khi nút 2 được thêm vào, mọi mối nối với nút 1 đã hoàn tất. Nút 2 cần được nhớ cho đến khi nút 4 được thêm vào, để tạo liên kết trái của nút 4. Tương tự, nút 4 cần được nhớ cho đến khi nút 8 được thêm vào. Trong hình 9.13, các mũi tên chỉ các nút cần được nhớ lại khi cây đang lớn lên.

Chúng ta thấy rằng để thực hiện các mối nối sau đó, đối với mỗi mức chúng ta chỉ cần nhớ lại duy nhất một con trỏ chỉ đến một nút, đó chính là nút cuối cùng trong mức đó. Chúng ta nắm giữ các con trỏ này trong một danh sách gọi là **last_node**, và danh sách này cũng sẽ rất nhỏ. Lấy ví dụ, một cây có 20 mức có thể chứa đến 2^{20}-1 > 1,000,000 nút, nhưng chỉ cần 20 phần tử trong last_node.

Khi một nút được thêm vào, chúng ta có thể gán con trỏ right của nó là NULL, có thể chỉ là tạm thời, vì nút con phải của nó (nếu có) sẽ được thêm vào sau. Con trỏ trái của một nút mới sẽ là NULL nếu đó là nút lá. Ngược lại, nút con trái của nó chính là nút ở ngay mức thấp hơn mức của nó mà địa chỉ đang chứa trong last_node. Chúng ta cũng có thể xử lý cho các nút lá tương tự như các nút khác bằng cách, nếu cho mức của nút lá là 1, thì chúng ta sẽ cho phần tử đầu tiên, tại vị trí 0, của last_node luôn mang trị NULL. Các mức bên trên mức của nút lá sẽ là 2, 3, ...

9.4.2. Các khai báo và hàm main

Chúng ta có thể định nghĩa một lớp mới, gọi là **Buildable_tree**, dẫn xuất từ lớp Search_tree.

```
template <class Record>
class Buildable_tree: public Search_tree<Record> {
public:
   Error_code build_tree(const List<Record> &supply);
 private: //  Các hàm phụ trợ.
};
```

Bước đầu tiên của build_tree là nhận các phần tử. Để đơn giản chúng ta sẽ cho rằng các phần tử này được chứa trong một danh sách các Record gọi là **supply**. Tuy nhiên, chúng ta cũng có thể viết lại hàm để nhận các phần tử từ một hàng đợi, hoặc một tập tin, hoặc thậm chí từ một cây nhị phân tìm kiếm khác với mong muốn cân bằng lại cây này.

Khi nhận các phần tử mới để thêm vào cây, chúng ta sẽ cập nhật lại một biến count để biết được có bao nhiêu phần tử đã được thêm vào. Rõ ràng là trị của count còn được dùng để lấy dữ liệu từ danh sách supply. Quan trọng hơn nữa, trị của count còn xác định mức của nút đang được thêm vào cây, nó sẽ được gởi cho một hàm chuyên lo việc tính toán này.

Sau khi tất cả các phần tử từ supply đã được thêm vào cây nhị phân tìm kiếm mới, chúng ta cần tìm gốc của cây và nối tất cả các cây con phải còn rời rạc.

```
template <class Record>
Error_code Buildable_tree<Record>::build_tree
                                    (const List<Record>&supply)
/*
post: Nếu dữ liệu trong supply có thứ tự tăng dần, cây nhị phân tìm kiếm khá cân bằng sẽ
      được tạo ra từ các dữ liệu này, phương thức trả về success. Ngược lại, cây nhị phân tìm
      kiếm chỉ được tạo ra từ mảng các dữ liệu tăng dần dài nhất tính bắt đầu từ đầu danh sách
      supply, phương thức trả về fail.
uses: Các phương thức của lớp List và các hàm build_insert, connect_subtrees, và
      find_root
*/
{  Error_code ordered_data = success;      // Sẽ được gán lại là fail nếu dữ liệu
                                              không tăng dần.
   int count = 0;
   Record x, last_x;
   List<Binary_node<Record> *> last_node; // Chỉ đến các nút cuối cùng của mỗi mức
                                              trong cây.
   Binary_node<Record> *none = NULL;
   last_node.insert(0, none); // luôn là NULL (dành cho các nút lá trong cây).
   while (supply.retrieve(count, x) == success) {
      if (count > 0 && x <= last_x) {
         ordered_data = fail;
         break;
      }
      build_insert(++count, x, last_node);
      last_x = x;
   }
   root = find_root(last_node);
   connect_trees(last_node);
   return ordered_data;
}
```

9.4.3. Thêm một nút

Phần trên đã xem xét cách nối các liên kết trái của các nút, nhưng trong quá trình xây dựng cây, các liên kết phải của các nút có lúc vẫn còn là NULL và chúng cần được thay đổi sau đó. Khi một nút mới được thêm vào, nó có thể sẽ có một cây con phải không rỗng. Do nó là nút mới nhất và lớn nhất được đưa vào cây cho đến thời điểm hiện tại, các nút trong cây con phải của nó chưa có. Ngược lại, một nút mới được thêm cũng có thể là nút con phải của một nút nào đó đã được đưa vào trước. Đồng thời, nó có thể là con trái của một nút có khóa lớn hơn, trong trường hợp này thì nút cha của nó chưa có. Chúng ta có thể xác định từng trường hợp đang xảy ra nhờ vào danh sách last_node. Nếu mức **level** của một nút đang được thêm vào lớn hơn hoặc bằng 1, thì nút cha của nó có mức **level+1**. Chúng ta tìm phần tử thứ level+1 trong last_node. Nếu con trỏ right của nút này vẫn còn là NULL thì nút đang được thêm mới chính là nút con phải của nó.

Ngược lại, nếu nó đã có con phải thì nút đang được thêm mới phải là con trái của một nút nào đó sẽ được thêm vào sau. Xem hình 9.1.

Chúng ta có thể viết hàm thêm một nút mới vào cây như sau:

```
template <class Record>
void Buildable_tree<Record>::build_insert(int count,
                                  const Record &new_data,
                                  List<Binary_node<Record>*> &last_node)
/*
post: Một nút mới chứa new_data được thêm vào cây. Mức của nút này bằng số lần count chia
      chẵn cho 2 cộng thêm 1, nếu xem mức của nút lá bằng 1.
uses: Các phương thức của lớp List.
*/
{
   int level;
   for (level = 1; count % 2 == 0; level++)
      count /= 2;    // Sử dụng count để tính mức của nút kế.
   Binary_node<Record> *next_node = new Binary_node<Record>(new_data),
                       *parent;
   last_node.retrieve(level - 1, next_node->left); // Nút mới được tạo ra
                           // nhận con trái chính là nút có địa chỉ đang được lưu
                           // trong last_node tại phần tử tương ứng với  mức
                           // ngay dưới mức của nút mới.
   if (last_node.size() <= level)
      last_node.insert(level, next_node); // Nút mới là nút đầu tiên xuất hiện
                                          //  trong mức của nó.
   else
      last_node.replace(level, next_node);// Đã có nhiều nút cùng mức và nút
                                          //  mới này chính là nút xuất hiện
                                          // mới nhất trong các nút cùng mức
                                          //  nên cần lưu lại địa chỉ.
   if (last_node.retrieve(level + 1, parent)==success
                                           && parent->right==NULL)
                                          //  Đây là trường hợp nút cha của
                                          // nút mới đã tồn tại và nút cha này
                                          // sẽ nhận nó làm con phải.
      parent->right = next_node;
}
```

9.4.4. Hoàn tất công việc

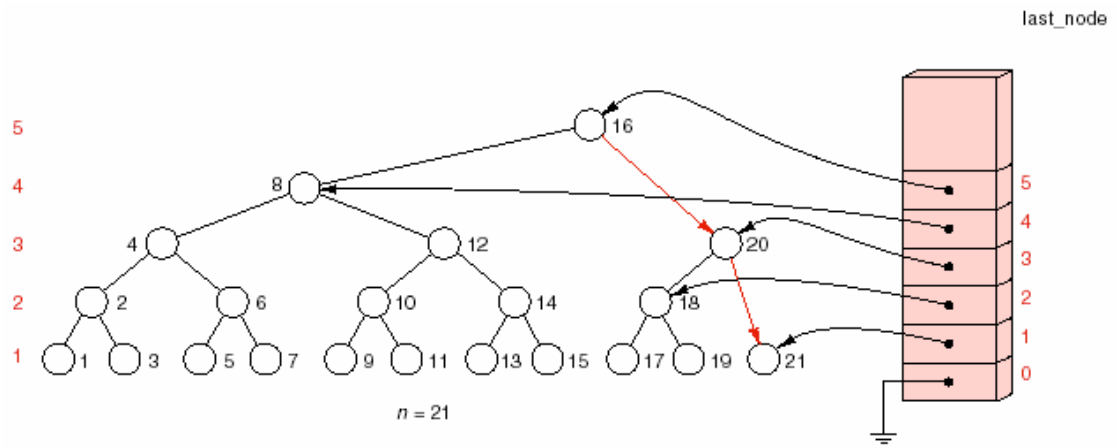
Tìm gốc của cây vừa được tạo là một việc dễ dàng: gốc chính là nút ở mức cao nhất trong cây, con trỏ chỉ đến nó chính là phần tử cuối cùng trong danh sách last_node. Cây có 21 nút như hình 9.13 có nút cao nhất là nút 16 ở mức 5, đó chính gốc của cây. Các con trỏ đến các nút cuối của mỗi mức được chứa trong last_node như hình vẽ 9.14.

Chúng ta có hàm như sau:

```
template <class Record>
Binary_node<Record> *Buildable_tree<Record>::find_root
                                (List<Binary_node<Record>*> &last_node)
/*
pre: Danh sách last_node chứa địa chỉ các nút cuối cùng của mỗi mức trong cây nhị phân tìm
     kiếm.
post: Trả về địa chỉ nút gốc của cây nhị phân tìm kiếm đã được tạo ra.
uses: Các phương thức của lớp List.
*/
{
   Binary_node<Record> *high_node;
   last_node.retrieve(last_node.size() - 1, high_node);
      // Tìm địa chỉ nút gốc của cây tại phần tử tương ứng với mức cao nhất trong cây.
   return high_node;
}
```

Cuối cùng, chúng ta cần xác định cách nối các cây con còn nằm ngoài. Chẳng hạn, với cây có n = 21, chúng ta cần nối ba thành phần ở hình 9.13 vào một cây duy nhất. Theo hình vẽ chúng ta thấy rằng một số nút trong các phần trên của cây có thể vẫn còn con trỏ right là NULL, trong khi các nút đã thêm vào cây bây giờ phải trở thành nút con phải của chúng.

Bất kỳ nút nào trong số các nút có con trỏ right vẫn còn là NULL, trừ nút lá, đều là một trong các nút nằm trong last_node. Với n=21, đó là các nút 16 và 20



Hình 9.14 – Hoàn tất cây nhị phân tìm kiếm.

tại các vị trí 5 và 3 tương ứng trong last_node trong hình 9.14.

Trong hàm sau đây chúng ta dùng con trỏ **high_node** để chỉ đến các nút có con trỏ right là NULL. Chúng ta cần xác định con trỏ **lower_node** chỉ đến nút con phải của **high_node**. Con trỏ lower_node có thể được xác định bởi nút cao nhất trong last_node mà không phải là nút con trái của high_node**.** Để xác định một

nút có phải là con trái của high_node hay không chúng ta so sánh khóa của nó với khóa của high_node.

```
template <class Record>
void Buildable_tree<Record>::connect_trees
                          (const List<Binary_node<Record>*> &last_node)
/*
pre: Danh sách last_node chứa địa chỉ các nút cuối cùng của mỗi mức trong cây nhị phân tìm
     kiếm. Cây nhị phân tìm kiếm được đã được tạo gần hoàn chỉnh.
post: Các liên kết cuối cùng trong cây được nối lại.
uses: Các phương thức của lớp List.
*/
{  Binary_node<Record> *high_node, //  from last_node with NULL right child
                       *low_node;  //  candidate for right child of high_node
   int high_level = last_node.size() - 1,
       low_level;
   while (high_level > 2) {        //  Nodes on levels 1 and 2 are already OK.
      last_node.retrieve(high_level, high_node);
      if (high_node->right != NULL)
         high_level--;             //  Search down for highest dangling node.
      else {                       //  Case: undefined right tree
         low_level = high_level;
         do {                      //  Find the highest entry not in the left
                                                              subtree.
            last_node.retrieve(--low_level, low_node);
         }while (low_node != NULL && low_node->data < high_node->data);
         high_node->right = low_node;
         high_level = low_level;
      }
   }
}
```

9.4.5. Đánh giá

Cây nhị phân tìm kiếm do giải thuật trên đây tạo ra không luôn là một cây cân bằng tốt nhất. Như chúng ta thấy, hình 9.14 là cây có n = 21 nút. Nếu nó có 31 nút, nó mới có sự cân bằng tốt nhất. Nếu nút thứ 32 được thêm vào thì nó sẽ trở thành gốc của cây, và tất cả 31 nút đã có sẽ thuộc cây con trái của nó. Trong trường hợp này, các nút lá nằm cách nút gốc 5 bước tìm kiếm. Như vậy cây có 32 nút thường sẽ phải cần số lần so sánh nhiều hơn số lần so sánh cần thiết là một. Với cây có 32 nút, nếu nút gốc được chọn một cách tối ưu, thì đa số các nút lá cần 4 bước tìm kiếm từ nút gốc, chỉ có một nút lá là cần 5 bước.

Một lần so sánh dôi ra trong tìm nhị phân không phải là một sự trả giá cao, và rõ ràng rằng cây được tạo ra bởi phương pháp trên đây của chúng ta sẽ không bao giờ có số mức nhiều hơn số mức tối ưu quá một đơn vị. Còn có nhiều phương pháp phức tạp hơn để tạo ra một cây nhị phân tìm kiếm đạt được sự cân bằng cao nhất có thể, nhưng một phương pháp đơn giản như trên đây cũng rất cần thiết, đặc biệt là phương pháp này **không cần biết trước số nút sẽ được thêm vào cây.**

Trong phần 9.5 chúng ta sẽ tìm hiểu về cây AVL, trong đó việc thêm hay loại phần tử luôn bảo đảm cây vẫn gần với trạng thái cân bằng. Tuy nhiên, đối với nhiều ứng dụng, giải thuật đơn giản mà chúng ta mô tả ở đây đã là thích hợp.

9.5. Cân bằng chiều cao: Cây AVL

Giải thuật trong phần 9.4 có thể được sử dụng để xây dựng một cây nhị phân tìm kiếm gần như cân bằng, hoặc để khôi phục sự cân bằng. Tuy nhiên, trong nhiều ứng dụng, việc thêm và loại phần tử trong cây xảy ra thường xuyên, và với một thứ tự không biết trước. Trong một vài ứng dụng loại này, điều quan trọng cần có là tối ưu thời gian tìm kiếm bằng cách luôn duy trì cây gần với tình trạng cân bằng. Từ năm 1962 hai nhà toán học người Nga, G. M. Adel'Son-Vel'Skil và E. M. Landis, đã mô tả một phương pháp nhằm đáp ứng yêu cầu này, và cây nhị phân tìm kiếm này được gọi là cây AVL.

Cây AVL đạt được mục đích là việc tìm kiếm, thêm vào, loại bỏ phần tử trong một cây n nút có được thời gian là O(log n), ngay cả trong trường hợp xấu nhất. Chiều cao của cây AVL n nút, mà chúng ta sẽ xem xét sau, sẽ không bao giờ vượt quá 1.44lg n, và như vậy ngay cả trong trường hợp xấu nhất, các hành vi trong cây AVL cũng không thể chậm hơn một cây nhị phân tìm kiếm ngẫu nhiên. Tuy nhiên, trong phần lớn các trường hợp, thời gian tìm kiếm thật sự thường rất gần với lg n, và như vậy hành vi của các cây AVL rất gần với hành vi của một cây nhị phân tìm kiếm cân bằng hoàn toàn lý tưởng.

9.5.1. Định nghĩa

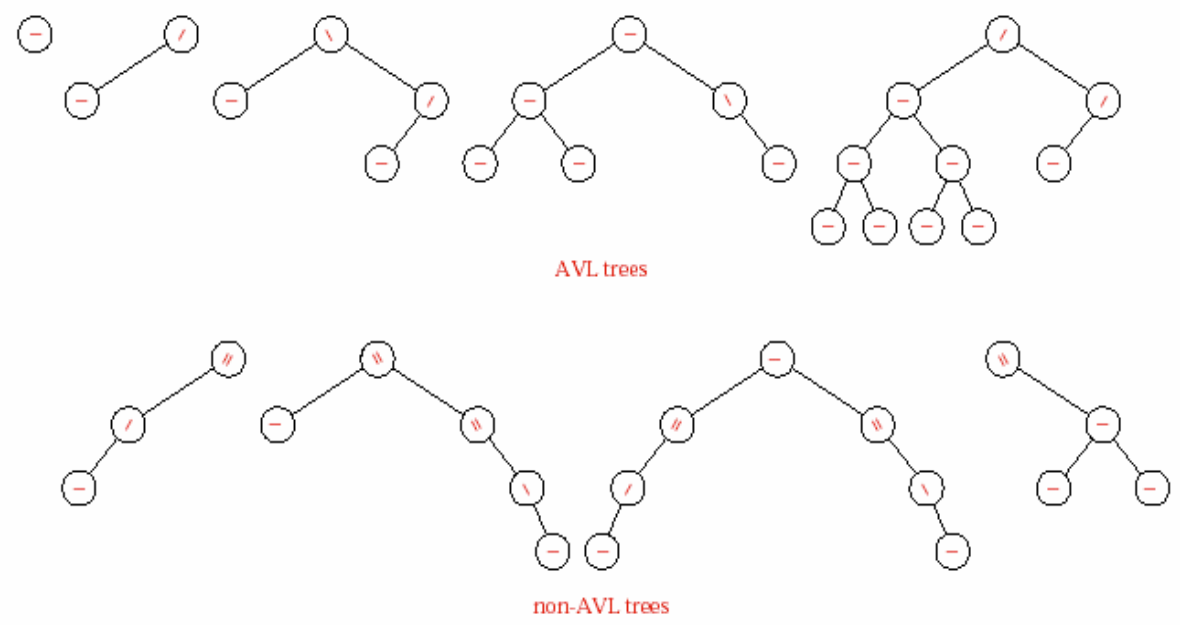
Trong một cây cân bằng hoàn toàn, các cây con trái và cây con phải của bất kỳ một nút nào cũng phải có cùng chiều cao. Mặc dù chúng ta không thể luôn luôn đạt được điều này, nhưng bằng cách xây dựng cây nhị phân tìm kiếm một cách cẩn thận, chúng ta luôn có thể bảo đảm rằng chiều cao của cây con trái và chiều cao của cây con phải của bất kỳ một nút nào đều hơn kém nhau không quá 1. Chúng ta có định nghĩa sau:

Định nghĩa: Cây AVL là một cây nhị phân tìm kiếm trong đó chiều cao cây con trái và chiều cao cây con phải của nút gốc hơn kém nhau không quá 1, và cả hai cây con trái và phải này đều là cây AVL.

Mỗi nút của cây AVL có thêm một thông số cân bằng mang trị `left-higher`, `equal-height`, hoặc `right-higher` tương ứng trường hợp cây con trái cao hơn, bằng, hoặc thấp hơn cây con phải.

Trong sơ đồ, chúng ta dùng '/' để chỉ nút có cây con trái cao hơn cây con phải, '-' chỉ nút cân bằng có hai cây con cao bằng nhau, và '\' chỉ nút có cây con phải cao hơn cây con trái. Hình 9.15 minh họa một vài cây AVL nhỏ và một số cây nhị phân không thỏa định nghĩa cây AVL.

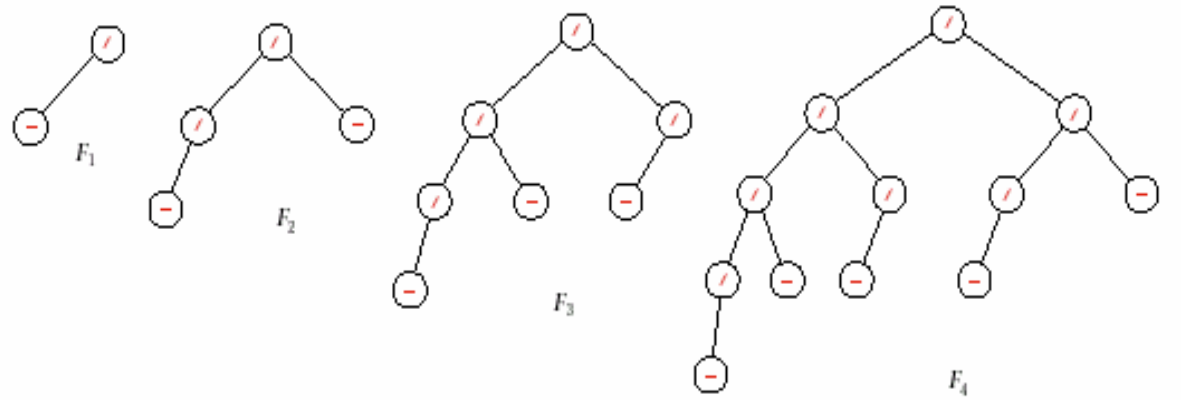
Lưu ý rằng, định nghĩa không yêu cầu mọi nút lá có cùng mức hoặc phải ở các mức kế nhau. Hình 9.16 minh họa một vài cây AVL rất không đối xứng, với các cây con trái cao hơn các cây con phải.



Hình 9.15 - Các ví dụ về cây AVL và các cây nhị phân khác.

Chúng ta dùng kiểu liệt kê để khai báo thông số cân bằng:

```
enum Balance_factor {left_higher,equal_height,right_higher};
```



Hình 9.16 – Một số cây AVL không đối xứng với cây con trái cao hơn cây con phải.

Thông số cân bằng phải có trong tất cả các nút của cây AVL, chúng ta cần bổ sung đặc tả một node như sau:

```
template <class Record>
struct AVL_node: public Binary_node<Record> {
   Balance_factor balance;
// constructors:
   AVL_node();
   AVL_node(const Record &x);
// Các hàm ảo được định nghĩa lại:
   void set_balance(Balance_factor b);
   Balance_factor get_balance() const;
};
```

Một điểm cần lưu ý trong đặc tả này là các con trỏ left và right của Binary_node có kiểu Binary_node*. Do đó các con trỏ mà AVL_node thừa kế cũng có kiểu này. Tuy nhiên, trong một cây AVL, rõ ràng là chúng ta cần các nút của cây AVL tham chiếu đến các nút khác của cây AVL. Sự không tương thích về kiểu của con trỏ này không phải là một vấn đề nghiêm trọng, vì một con trỏ đến một đối tượng của một lớp cơ sở cũng có thể chỉ đến một đối tượng của lớp dẫn xuất. Trong trường hợp của chúng ta, các con trỏ left và rigth của một AVL_node có thể chỉ đến các AVL_node khác một cách hợp lệ. Lợi ích của việc hiện thực AVL_node thừa kế từ Binary_node là sự sử dụng lại tất cả các phương thức của cây nhị phân và cây tìm kiếm. Tuy nhiên, chúng ta cần bảo đảm rằng khi thêm một nút mới vào cây AVL, chúng ta chỉ thêm đúng các nút AVL mà thôi.

Chúng ta sẽ sử dụng các phương thức get_balance và set_balance để xác định thông số cân bằng của AVL_node.

```
template <class Record>
void AVL_node<Record>::set_balance(Balance_factor b)
{
   balance = b;
}
```

```
template <class Record>
Balance_factor AVL_node<Record>::get_balance() const
{
   return balance;
}
```

Chúng ta sẽ gọi hai phương thức này thông qua con trỏ chỉ đến nút của cây, chẳng hạn left->get_balance(). Tuy nhiên, cách gọi này có thể gặp vấn đề đối với trình biên dịch C++ do left là con trỏ chỉ đến một Binary_node chứ không phải AVL_node. Trình biên dịch phải từ chối biểu thức left->get_balance() do nó không chắc rằng đó có là phương thức của đối tượng *left hay không. Chúng ta giải quyết vấn đề này bằng cách thêm vào các phiên bản giả (*dummy version*) của các phương thức get_balance() và set_balance() cho cấu trúc Binary_node. Các phương thức giả được thêm vào này chỉ để dành cho các hiện thực của cây AVL dẫn xuất.

Sau khi chúng ta bổ sung các phương thức giả cho cấu trúc Binary_node, trình biên dịch sẽ chấp nhận biểu thức left->set_balance(). Tuy nhiên, vẫn còn một vấn đề mà trình biên dịch vẫn không thể giải quyết được: nó nên sử dụng phương thức của AVL_node hay phương thức giả? Sự lựa chọn đúng chỉ có thể được thực hiện trong thời gian chạy của chương trình, khi kiểu của *left đã được biết. Một cách tương ứng, chúng ta phải khai báo các phiên bản set_balance và get_balance của Binary_node là các phương thức ảo (*virtual*). Điều này có nghĩa là sự lựa chọn phiên bản nào sẽ được thực hiện trong thời gian chạy của chương trình. Chẳng hạn, nếu

`set_balance()` được gọi như một phương thức của `AVL_node`, thì phiên bản của AVL sẽ được sử dụng, nếu nó được gọi như một phương thức của `Binary_node` thì phiên bản giả sẽ được sử dụng.

Dưới đây là đặc tả của `Binary_node` đã được sửa đổi:

```
template <class Entry>
struct Binary_node {
//    data members:
   Entry data;
   Binary_node<Entry> *left;
   Binary_node<Entry> *right;
//    constructors:
   Binary_node();
   Binary_node(const Entry &x);
//    virtual methods:
   virtual void set_balance(Balance_factor b);
   virtual Balance_factor get_balance() const;
};
```

```
template <class Entry>
void Binary_node<Entry>::set_balance(Balance_factor b)
{
}
```

```
template <class Entry>
Balance_factor Binary_node<Entry>::get_balance() const
{
   return equal_height;
}
```

Ngoài ra không có sự thay đổi nào khác trong các lớp và các phương thức của chúng ta, và mọi hàm xử lý cho các nút trước kia của chúng ta bây giờ đã có thể sử dụng cho các nút AVL.

Bây giờ chúng ta đã có thể đặc tả lớp cho cây AVL. Chúng ta chỉ cần định nghĩa lại các phương thức thêm và loại phần tử để có thể duy trì cấu trúc cây cân bằng. Các phương thức khác của cây nhị phân tìm kiếm đều có thể được thừa kế mà không cần thay đổi gì.

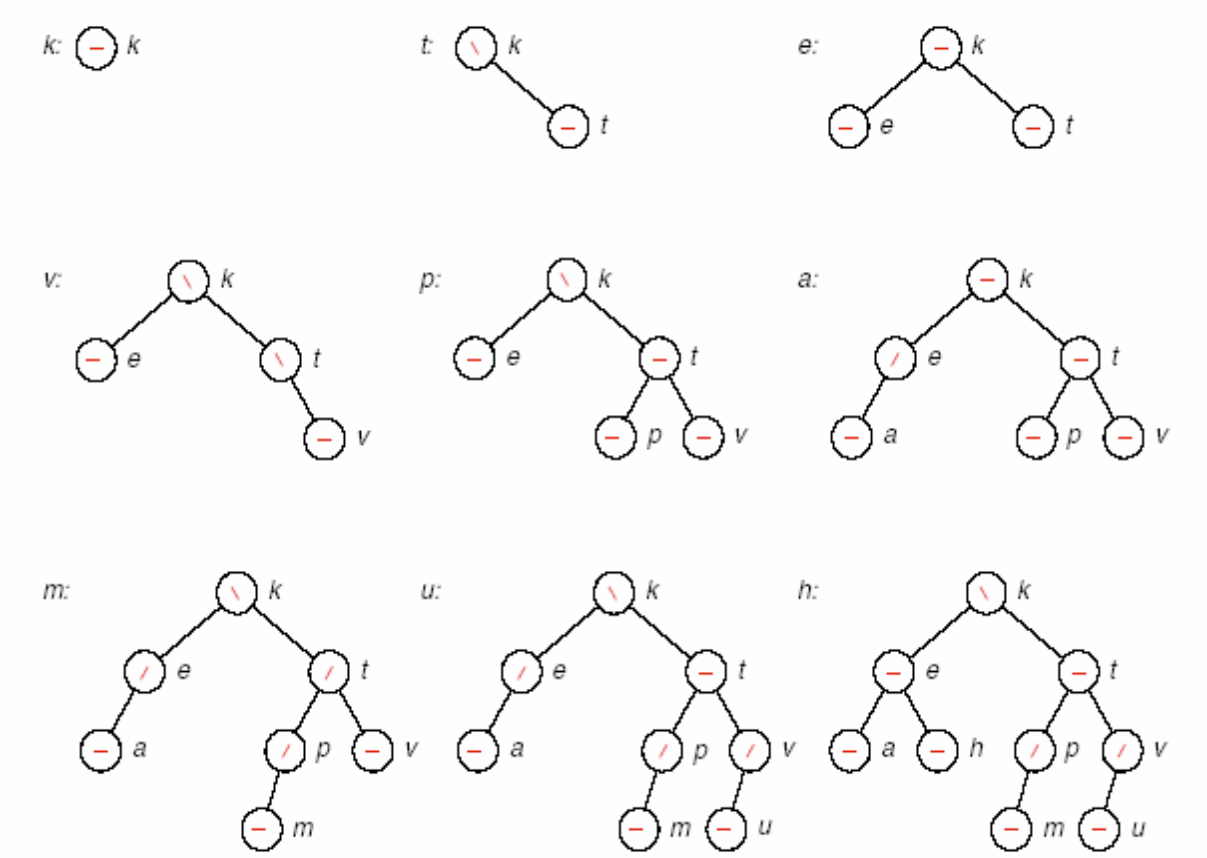
```
template <class Record>
class AVL_tree: public Search_tree<Record> {
public:
   Error_code insert(const Record &new_data);
   Error_code remove(const Record &old_data);
 private: //  Các hàm phụ trợ.
};
```

Thuộc tính dữ liệu lớp này thừa kế được là con trỏ root. Con trỏ này có kiểu `Binary_node<Record>*` và do đó, như chúng ta đã thấy, nó có thể chứa địa chỉ của một nút của cây nhị phân nguyên thủy hoặc địa chỉ của một nút của cây AVL. Chúng ta phải bảo đảm rằng phương thức `insert` được định nghĩa lại chỉ tạo ra các nút có kiểu `AVL_node`, làm như vậy mới bảo đảm rằng mọi nút được truy xuất thông qua con trỏ `root` của cây AVL đều là các nút AVL.

9.5.2. Thêm một nút

9.5.2.1. Dẫn nhập

Chúng ta hãy theo dõi sự lớn lên của cây AVL trong hình 9.17 qua việc thêm một số phần tử.



Hình 9.17 – Thêm nút vào cây AVL: các trường hợp đơn giản

Quá trình thêm ở hình 9.17 được tiến hành chính xác theo cùng một cách với quá trình thêm vào cây nhị phân tìm kiếm thông thường, ngoại trừ việc cập nhật lại thông số cân bằng. Tuy nhiên, chúng ta lưu ý rằng các thông số cân bằng chỉ có thể được xác định sau khi việc thêm vào đã được thực hiện. Khi v được thêm vào theo hình 9.17, thông số cân bằng trong gốc k thay đổi, nhưng nó không thay đổi khi p được thêm kế sau đó. Cả v và p đều được thêm vào cây con phải t của nút gốc, và chỉ sau khi việc thêm vào hoàn tất, thông số cân bằng tại nút gốc k mới có thể được xác định.

Chúng ta có thể thêm một nút mới vào một cây AVL bằng cách như sau. Trước hết theo đúng giải thuật thêm vào một cây nhị phân tìm kiếm bình thường : so sánh khóa của nút mới với khóa của nút gốc, và thêm nút mới vào cây con trái hoặc cây con phải thích hợp.

Trong quá trình lần tìm xuống các nhánh, nếu khóa cần thêm vào chưa có thì việc thêm nút mới cuối cùng sẽ được thực hiện tại một cây con rỗng. Từ cây con rỗng trở thành cây con có một nút, chiều cao nó đã tăng lên. Điều này có thể ảnh hưởng đến các nút trên đường đi từ nút cha của nó trở lên gốc. Vì vậy trong quá trình duyệt cây đi xuống để tìm vị trí thêm vào, chúng ta cần lưu lại vết để có thể lần ngược về để xử lý. Chúng ta có thể dùng ngăn xếp, hoặc đơn giản hơn là dùng hàm đệ quy.

Tham biến `taller` của hàm đệ quy được gán bằng `true` tại lần đệ quy cuối cùng, đó chính là lúc nút mới được tạo ra tại một cây con rỗng làm cho nó tăng chiều cao như đã nói ở trên. Những việc sau đây cần được thực hiện tại mỗi nút nằm trên đường đi từ nút cha của nút vừa được tạo ra cho đến nút gốc của cây. Trong khi mà tham biến `taller` trả lên còn là `true`, việc giải quyết cứ phải tiếp tục cho đến khi có một nút nhận được trị trả về của tham biến này là `false`. Một khi có một cây con nào đó báo lên rằng chiều cao của nó không bị tăng lên thì các nút trên của nó sẽ không bị ảnh hưởng gì, giải thuật kết thúc. Ngược lại, gọi nút đang được xử lý trong hàm đệ quy là `sub_root`. Sau khi gọi đệ quy xuống cây con và nhận được thông báo trả về rằng chiều cao cây con có tăng (`taller == true`), cần xét các trường hợp sau:

1. Cây con tăng chiều cao vốn là cây con thấp hơn cây con còn lại, thì chỉ có thông số cân bằng trong `sub_root` là cần thay đổi, cây có gốc là `sub_root` cũng không thay đổi chiều cao. Tham biến `taller` được gán về `false` bắt đầu từ đây.

2. Trước khi một cây con báo lên là tăng chiều cao thì hai cây con vốn cao bằng nhau. Trường hợp này cần thay đổi thông số cân bằng trong `sub_root`, đồng thời cây có gốc là `sub_root` cũng cao lên. Tham biến `taller` vẫn giữ nguyên là `true` để nút trên của `sub_root` giải quyết tiếp.

3. Cây con tăng chiều cao vốn là cây con cao hơn cây con còn lại. Khi đó `sub_root` sẽ có hai cây con có chiều cao chênh lệch là 2, không thỏa định nghĩa cây AVL. Đây là trường hợp chúng ta cần đến hàm `right_balance` hoặc `left_balance` để cân bằng lại cây có gốc là `sub_root` này. Chúng ta sẽ nghiên cứu nhiệm vụ của hai hàm này sau, nhưng tại đây cũng có thể nói trước rằng việc cân bằng lại luôn giải quyết được triệt để. Có nghĩa là cây có gốc là `sub_root` sẽ không bị tăng chiều cao, và như vậy, tham biến `taller` cũng được gán về `false` bắt đầu từ đây.

Với các quyết định trên, chúng ta có thể viết các phương thức và các hàm phụ trợ để thêm một nút mới vào cây AVL.

```
template <class Record>
Error_code AVL_tree<Record>::insert(const Record &new_data)
/*
post: Nếu khóa trong new_data đã có trong cây, phương thức trả về duplicate_error. Ngược
      lại, new_data được thêm vào cây sao cho cây vẫn thoả cây AVL, phương thức trả về
      success.
uses: Hàm avl_insert.
*/
{  bool taller;
   return avl_insert(root, new_data, taller);
}
```

```
template <class Record>
Error_code AVL_tree<Record>::avl_insert(Binary_node<Record> *&sub_root,
           const Record &new_data, bool &taller)
/*
pre: sub_root là NULL hoặc là gốc một cây con trong cây AVL.
post: Nếu khóa trong new_data đã có trong cây, phương thức trả về duplicate_error. Ngược
      lại, new_data được thêm vào cây sao cho cây vẫn thoả cây AVL, phương thức trả về
      success. Nếu cây con có tăng chiều cao, thông số taller được gán trị true, ngược lại
      nó được gán trị false.
uses: Các phương thức của AVL_node; hàm avl_insert một cách đệ quy, các hàm
      left_balance và right_balance.
*/
{  Error_code result = success;
   if (sub_root == NULL) {
      sub_root = new AVL_node<Record>(new_data);
      taller = true;
   }

   else if (new_data == sub_root->data) {
      result = duplicate_error;
      taller = false;
   }

   else if (new_data < sub_root->data) {    //  Thêm vào cây con trái.
      result = avl_insert(sub_root->left, new_data, taller);
      if (taller == true)
         switch (sub_root->get_balance()) { //  Change balance factors.

         case left_higher:
            left_balance(sub_root);
            taller = false;    // Cân bằng lại luôn làm cho cây không bị cao lên.
            break;

         case equal_height:
            sub_root->set_balance(left_higher); // Cây con có gốc sub_root
                                                // đã bị cao lên, taller vẫn là true.
            break;

         case right_higher:
            sub_root->set_balance(equal_height);
            taller = false; // Cây con có gốc sub_root không bị cao lên.
            break;
         }
   }
```

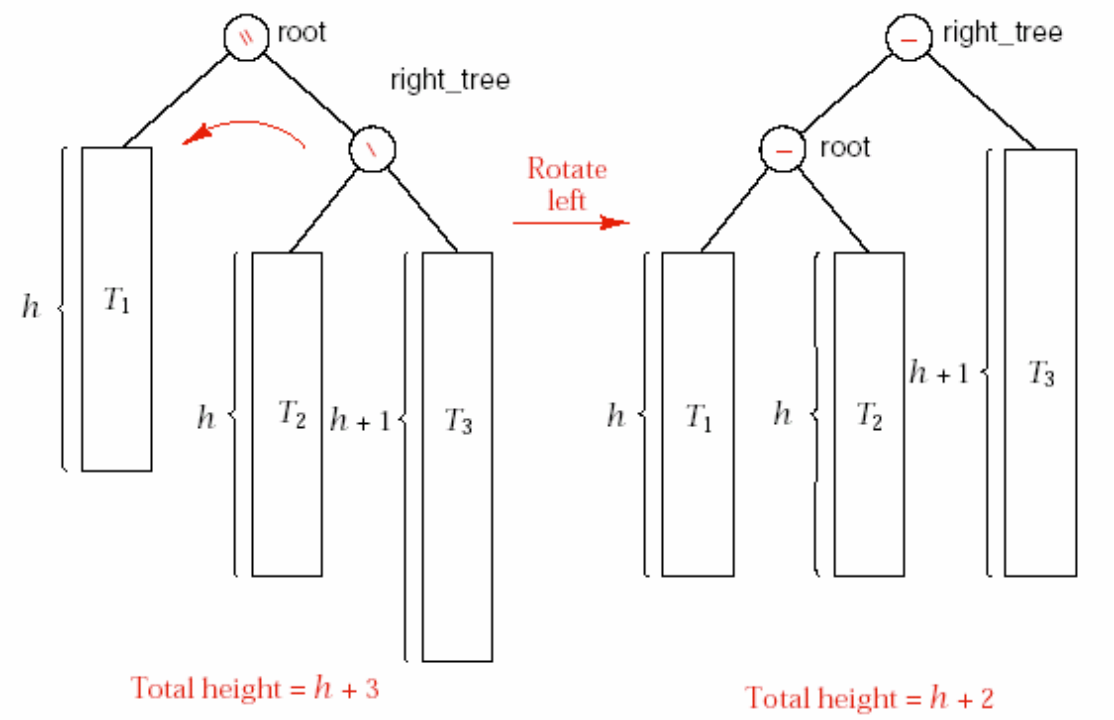
```
   else {                                        //  Thêm vào cây con phải.
      result = avl_insert(sub_root->right, new_data, taller);
      if (taller == true)
         switch (sub_root->get_balance()) {

         case left_higher:
            sub_root->set_balance(equal_height);
            taller = false;       // Cây con có gốc sub_root không bị cao lên.

            break;

         case equal_height:
            sub_root->set_balance(right_higher);// Cây con có gốc sub_root // đã
                                          bị cao lên, taller vẫn là true.
            break;

         case right_higher:
            right_balance(sub_root);
            taller = false;    // Cân bằng lại luôn làm cho cây không bị cao lên.
            break;
         }
   }
   return result;
}
```

9.5.2.2. Các phép quay

Chúng ta hãy xét trường hợp nút mới được thêm vào cây con cao hơn và chiều cao của nó tăng lên, chênh lệch chiều cao hai cây con trở thành 2, và cây không còn thoả điều kiện của cây AVL. Chúng ta cần tổ chức lại một phần của cây để khôi phục lại sự cân bằng. Hàm **right_balance** sẽ xem xét trường hợp cây có cây con phải cao hơn cây con trái 2 đơn vị. (Trường hợp ngược lại được giải quyết trong hàm **left_balance** mà chúng ta sẽ xem như bài tập). Cho **root** là gốc của cây và **right_tree** là gốc của cây con phải của nó.

Có ba trường hợp cần phải xem xét, phụ thuộc vào thông số cân bằng của gốc của right_tree.

***Trường hợp 1*: Cây con phải của right_tree cao hơn cây con trái của nó**

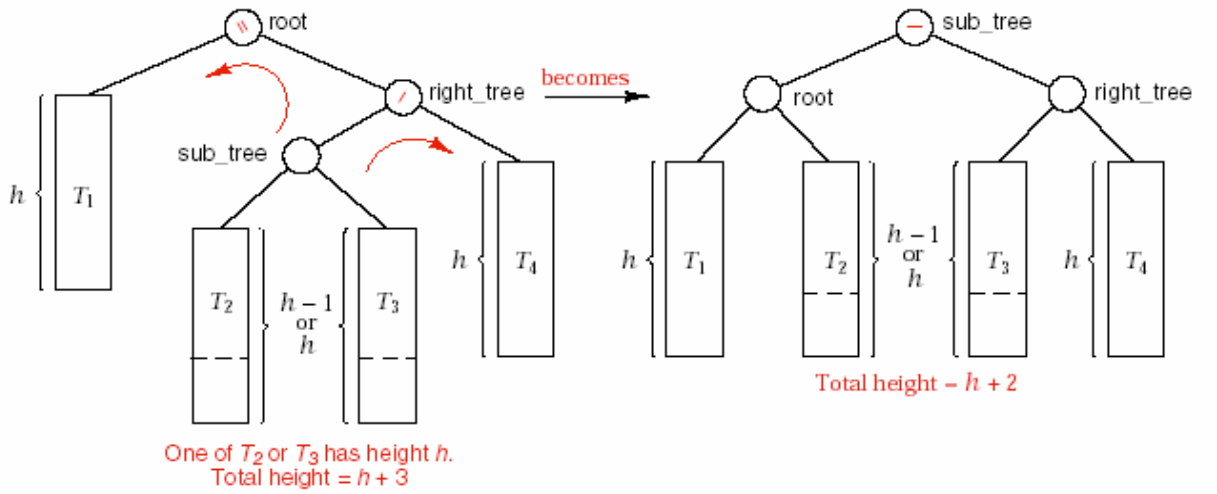


Hình 9.18 – Trường hợp 1: Khôi phục sự cân bằng bởi phép quay trái.

Trường hợp thứ nhất, khi thông số cân bằng trong nút gốc của `right_tree` là `right_higher`, xem hình 9.18, cần thực hiện phép quay trái (*left rotation*). Chúng ta cần quay nút gốc của **`right_tree`** hướng về **`root`**, hạ **`root`** xuống thành cây con trái. Cây con T2, vốn là cây con trái của `right_tree`, trở thành cây con phải của `root`. Do T2 gồm các nút có khóa nằm giữa khóa của `root` và khóa của nút gốc của `right_tree` nên cách thay đổi như vậy vẫn bảo đảm thứ tự các khóa của một cây nhị phân tìm kiếm. Phép quay trái được hiện thực trong hàm **rotate_left** dưới đây. Chúng ta đặc biệt lưu ý rằng, khi thực hiện theo một trật tự thích hợp, các bước gán tạo thành một sự quay vòng của các trị trong ba biến con trỏ. Sau phép quay, chiều cao của toàn bộ cây giảm 1, nhưng do nó vừa tăng như đã nói ở trên (và đây cũng chính là nguyên nhân gây nên sự mất cân bằng cần phải cân bằng lại bằng phép quay này) , nên chiều cao cuối cùng của cây xem như không đổi.

```
template <class Record>
void AVL_tree<Record>::rotate_left(Binary_node<Record> *&sub_root)
/*
pre:  sub_root là địa chỉ của gốc của một cây con trong cây AVL. Cây con này có cây con phải
      không rỗng.
post: sub_root được gán lại địa chỉ của nút vốn là con phải của nó, nút gốc của cây con ban
      đầu sẽ thành nút con trái của cây con mới.
*/
```

```
{
   if (sub_root == NULL || sub_root->right == NULL) // các trường hợp
                                                    // không thể xảy ra
      cout << "WARNING: program error detected in rotate_left" << endl;

else {
      Binary_node<Record> *right_tree = sub_root->right;
      sub_root->right = right_tree->left;
      right_tree->left = sub_root;
      sub_root = right_tree;
   }
}
```



Hình 9.19 – Trường hợp 2: Khôi phục sự cân bằng bởi phép quay kép.

***Trường hợp 2*: Cây con trái của `right_tree` cao hơn cây con phải của nó**

Trường hợp thứ hai, khi thông số trong `right_tree` là `left_higher`, phức tạp hơn. Theo hình vẽ 9.19, nút **`sub_tree`** cần di chuyển lên hai mức để thành nút gốc mới. Cây con phải T3 của **`sub_tree`** sẽ trở thành cây con trái của **`right_tree`**. Cây con trái T2 của **`sub_tree`** trở thành cây con phải của **`root`**. Quá trình này còn được gọi là phép quay kép (*double rotation*), do sự biến đổi cần qua hai bước. Thứ nhất là cây con có gốc là **`right_tree`** được quay sang phải để **`sub_tree`** trở thành gốc của cây này. Sau đó cây có gốc là **`root`** quay sang trái để **`sub_tree`** trở thành gốc.

Trong trường hợp thứ hai này, thông số cân bằng mới của **`root`** và **`right_tree`** phụ thuộc vào thông số cân bằng trước đó của **`sub_tree`**. Thông số cân bằng mới của **`sub_tree`** luôn là `equal_height`. Hình 9.19 minh họa các cây con của **`sub_tree`** có chiều cao bằng nhau, nhưng thực tế **`sub_tree`** có thể là left_higher hoặc right_higher. Các thông số cân bằng kết quả sẽ là:

old **sub_tree**	*new* **root**	*new* **right_tree**	*new* **sub_tree**
-	-	-	-
/	-	\	-
\	/	-	-

***Trường hợp 3*: Hai cây con của `right_tree` cao bằng nhau**

Cuối cùng, chúng ta xét trường hợp thứ ba, khi cả hai cây con của **`right_tree`** cao bằng nhau, nhưng thực tế trường hợp này không thể xảy ra.

Cũng nên nhắc lại rằng trường hợp cần giải quyết ở đây là do chúng ta vừa thêm một nút mới vào cây có gốc **`right_tree,`** làm cho cây này có chiều cao lớn hơn chiều cao cây con trái của **`root`** là 2 đơn vị. Nếu cây `right_tree` có hai cây con cao bằng nhau sau khi nhận thêm một nút mới vào cây con trái hoặc cây con phải của nó, thì **nó đã không thể tăng chiều cao**, do chiều cao của nó vẫn luôn bằng chiều cao của cây con vốn cao hơn cộng thêm 1. Vậy, trước khi thêm nút mới mà cây có gốc là `right_tree` đã cao hơn cây con trái của root 2 đơn vị là vô lý và sai giả thiết rằng cây vốn phải thỏa điều kiện của cây AVL.

9.5.2.3. Hàm right_balance

Chúng ta có hàm `right_balance` dưới đây. Các hàm `rotate_right` và `left_balance` cũng tương tự `rotate_left` và `right_balance`, chúng ta xem như bài tập.

```
template <class Record>
void AVL_tree<Record>::right_balance(Binary_node<Record> *&sub_root)
/*
pre:  sub_root chứa địa chỉ gốc của cây con trong AVL mà tại nút gốc này đang vi phạm điều
      kiện AVL: cây con phải cao hơn cây con trái 2 đơn vị.
post: Việc cân bằng lại giúp cho cây con có gốc sub_root đã thoả điều kiện cây AVL.
uses: các phương thức của AVL_node; các hàm rotate_right và rotate_left.
*/
{
   Binary_node<Record> *&right_tree = sub_root->right;

   switch (right_tree->get_balance()) {

   case right_higher: // Thực hiện 1 phép quay đơn sang trái.
      sub_root->set_balance(equal_height);
      right_tree->set_balance(equal_height);
      rotate_left(sub_root);
      break;

   case equal_height:                          // Trường hợp không thể xảy ra.
      cout << "WARNING:program error detected in right_balance" <<endl;
```

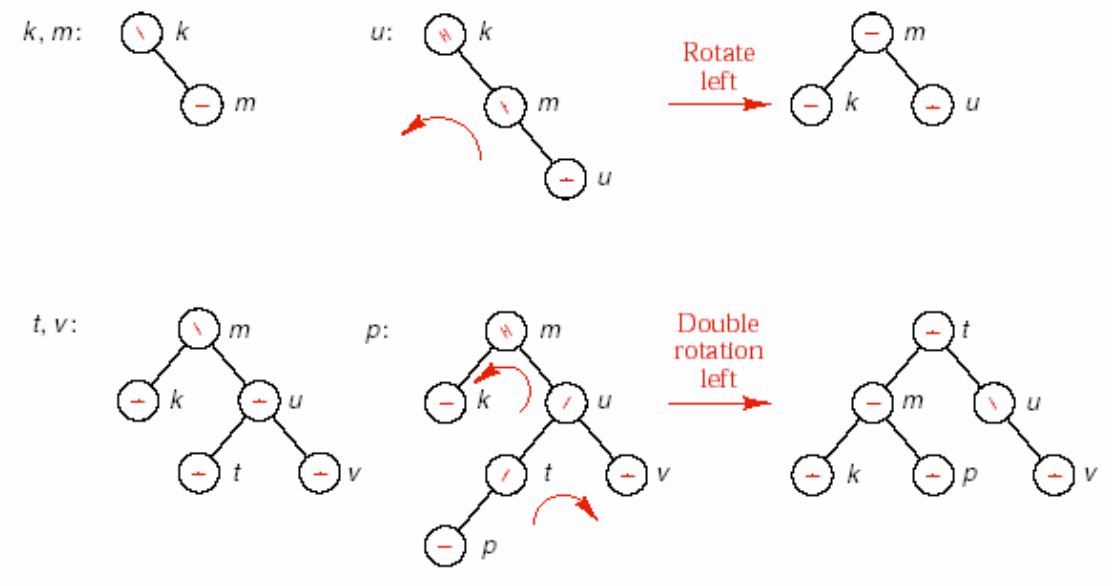
```
   case left_higher:   // Quay kép: quay đơn sang phải, rồi quay đơn sang trái.
      Binary_node<Record> *sub_tree = right_tree->left;
      switch (sub_tree->get_balance()) {

      case equal_height:
         sub_root->set_balance(equal_height);
         right_tree->set_balance(equal_height);
         break;

      case left_higher:
         sub_root->set_balance(equal_height);
         right_tree->set_balance(right_higher);
         break;

      case right_higher:
         sub_root->set_balance(left_higher);
         right_tree->set_balance(equal_height);
         break;
      }
      sub_tree->set_balance(equal_height);
      rotate_right(right_tree);
      rotate_left(sub_root);
      break;
   }
}
```

Hình 9.20 minh họa một ví dụ việc thêm vào cần có quay đơn và quay kép.

9.5.2.4. Hành vi của giải thuật

Số lần mà hàm `avl_insert` gọi đệ quy chính nó để thêm một nút mới có thể lớn bằng chiều cao của cây. Thoạt nhìn, dường như là mỗi lần gọi đệ quy đều dẫn đến một lần quay đơn hoặc quay kép cho cây con tương ứng, nhưng thực ra, nhiều nhất là chỉ có một phép quay (đơn hoặc kép) là được thực hiện. Để nhìn thấy điều này, chúng ta biết rằng các phép quay được thực hiện chỉ trong các hàm `right_balance` và `left_balance`, và các hàm này được gọi chỉ khi chiều cao của cây con có tăng lên. Tuy nhiên, khi các hàm này thực hiện xong, các phép quay đã làm cho chiều cao cây con trở về giá trị ban đầu, như vậy, đối với các lần gọi đệ quy trước đó chưa kết thúc, chiều cao cây con tương ứng sẽ không thay đổi, và sẽ không có một phép quay hay một sự thay đổi các thông số cân bằng nào nữa.

Phần lớn việc thêm vào cây AVL không dẫn đến phép quay. Ngay cả khi phép quay là cần thiết, nó thường xảy ra gần với nút lá vừa được thêm vào. Mặc dù giải thuật thêm vào một cây AVL là phức tạp, nhưng chúng ta có lý do để tin rằng thời gian chạy của nó không khác bao nhiêu so với việc thêm vào một cây nhị phân tìm kiếm bình thường có cùng chiều cao. Chúng ta còn có thể hy vọng rằng chiều cao của cây AVL nhỏ hơn rất nhiều chiều cao của cây nhị phân tìm kiếm bình thường, và như vậy cả hai việc thêm vào và loại bỏ nút sẽ hiệu quả hơn nhiều so với cây nhị phân tìm kiếm bình thường.



Hình 9.20 – Thêm nút vào cây AVL: các trường hợp cần có phép quay.

9.5.3. Loại một nút

Việc loại một nút x ra khỏi một cây AVL cũng theo ý tưởng tương tự, cũng bao gồm phép quay đơn và phép quay kép. Chúng ta sẽ phác thảo các bước của giải thuật dưới đây, hiện thực cụ thể của hàm xem như bài tập.

1. Chúng ta chỉ cần xem xét trường hợp nút x cần loại có nhiều nhất là một cây con, vì đối với trường hợp x có hai cây con, chúng ta sẽ tìm nút y là nút kế trước nó (hoặc nút kế sau nó) trong thứ tự duyệt *inorder* để loại thay cho nó. Từ nút con trái của x nếu di chuyển sang phải cho đến khi không thể di chuyển được nữa, chúng ta gặp một nút y không có con phải. Lúc đó chúng ta sẽ chép dữ liệu trong y vào x, không thay đổi thông số cân bằng cũng như con trái và con phải của x. Cuối cùng nút y sẽ được loại đi thay cho nút x theo các bước dưới đây.

2. Gọi nút cần loại là x. Loại nút x ra khỏi cây. Do x có nhiều nhất một con, việc loại x đơn giản chỉ là nối tham chiếu từ nút cha của x đến nút con của nó (hoặc `NULL` nếu x không có con nào). Chiều cao cây con có gốc là x giảm đi 1, và điều này **có thể ảnh hưởng đến các nút nằm trên đường đi từ x ngược về gốc của cây**. Vì vậy trong quá trình duyệt cây để xuống đến nút x, chúng ta cần lưu vết bằng cách sử dụng ngăn xếp để có thể xử lý lần ngược về. Để đơn giản, chúng ta có thể dùng **hàm đệ quy với tham biến `shorter`** kiểu `bool` cho biết chiều cao của cây con có bị giảm đi hay không. Các việc cần làm tại mỗi nút phụ thuộc vào trị của tham biến `shorter` mà con nó trả về sau khi việc gọi đệ quy xuống cây con kết thúc, vào thông số cân bằng của nó, và đôi khi phụ thuộc vào cả thông số cân bằng của nút con của nó.

3. Biến `bool shorter` được khởi gán trị true. Các bước sau đây sẽ được thực hiện tại mỗi nút nằm trên đường đi từ nút cha của x cho đến nút gốc của cây. Trong khi mà tham biến `shorter` trả lên còn là `true`, việc giải quyết cứ tiếp tục cho đến khi có một nút nhận được trị trả về của tham biến này là `false`. Một khi có một cây con nào đó báo rằng chiều cao của nó không bị giảm đi thì các nút trên của nó sẽ không bị ảnh hưởng gì cả. Ngược lại, gọi nút p là nút vừa gọi đệ quy xuống nút con xong và nhận thông báo `shorter` là `true`, cần xét các trường hợp sau:

   Trường hợp 1: Nút p hiện tại có thông số cân bằng là **`equal_height`**. Thông số này sẽ thay đổi tùy thuộc vào cây con trái hay cây con phải của nó đã bị ngắn đi. Biến `shorter` đổi thành **`false`**.

   Trường hợp 2: Nút p hiện tại có thông số cân bằng **không là `equal_height`**. **Cây con vốn cao hơn vừa bị ngắn đi**. Thông số cân bằng của p chỉ cần đổi về `equal_height`. Trường hợp này cây gốc p cũng bị giảm chiều cao nên tham biến `shorter` vẫn là `true` để trả lên trên cho cha của p giải quyết tiếp.

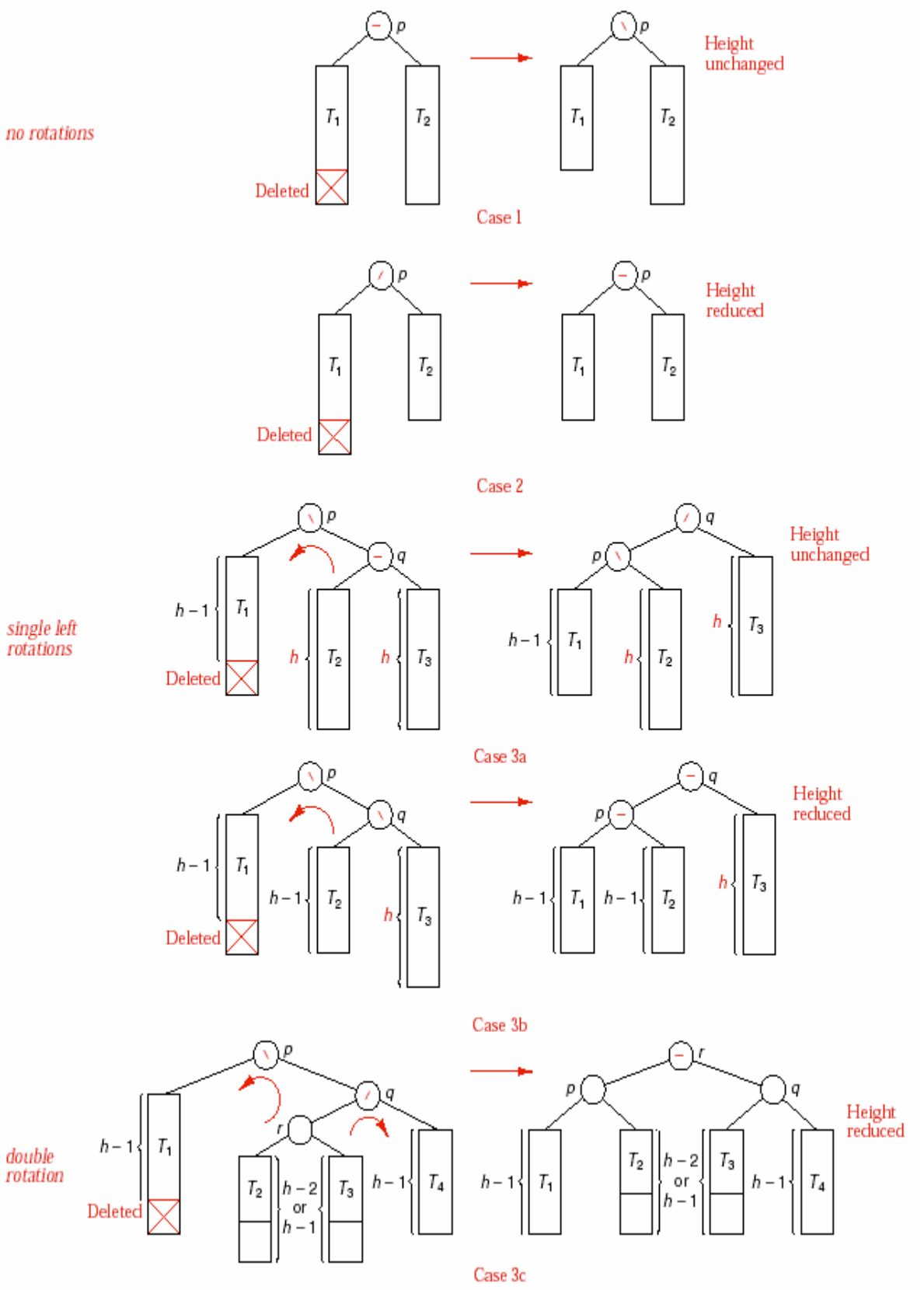
   Trường hợp 3: Nút p hiện tại có thông số cân bằng **không là `equal_height`. Cây con vốn thấp hơn vừa bị ngắn đi**. Như vậy tại nút p **vi phạm điều kiện của cây AVL**, và chúng ta thực hiện phép quay để khôi phục lại sự cân bằng như sau. Gọi q là gốc của cây con cao hơn của p, có 3 trường hợp tương ứng với thông số cân bằng trong q:

      Trường hợp 3a: **Thông số cân bằng của q là `equal_height`**. Thực hiện một phép **quay đơn** để thay đổi các thông số cân bằng của p và q, trạng thái cân bằng sẽ được khôi phục, `shorter` đổi thành **`false.`**

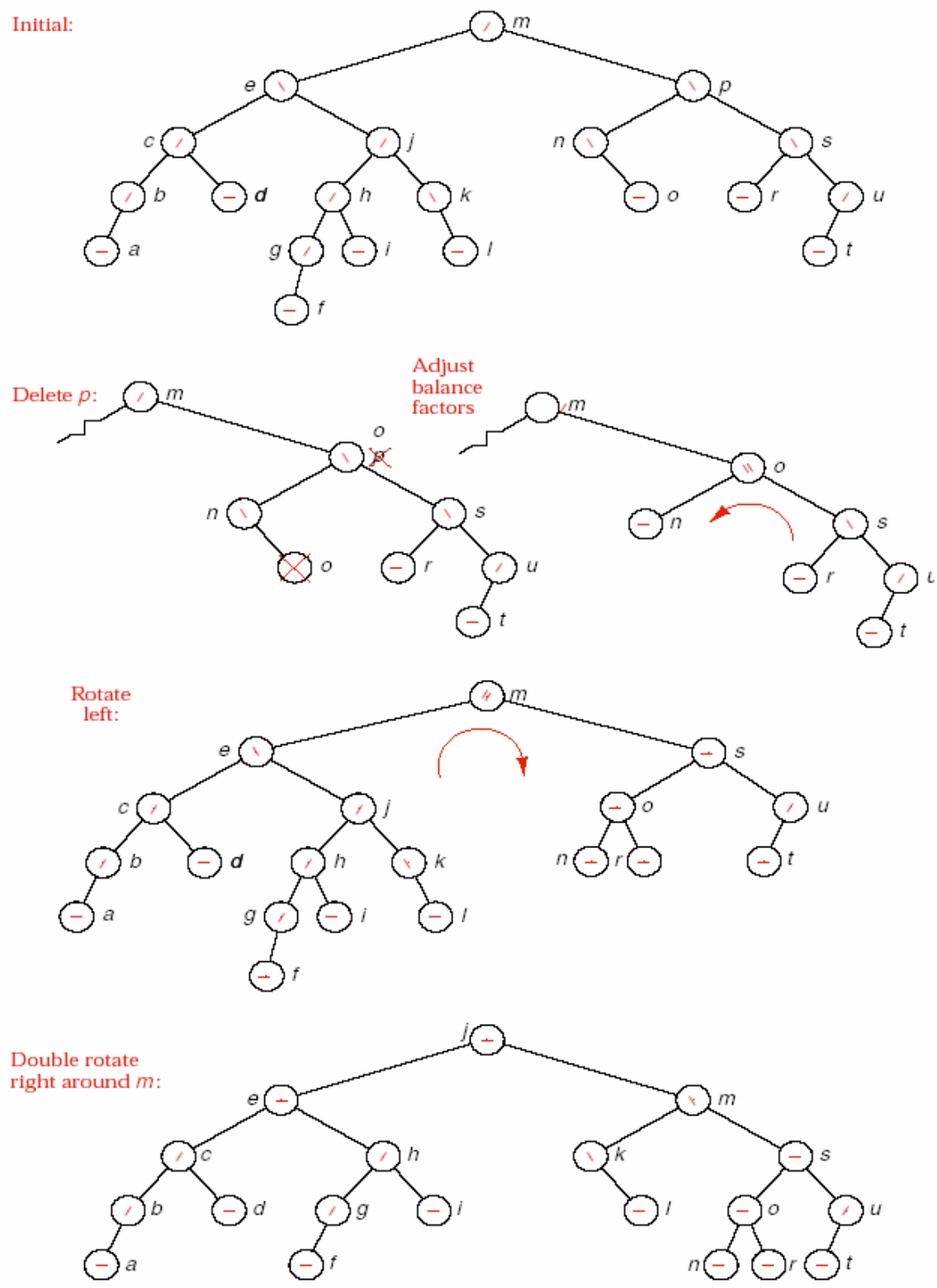
      Trường hợp 3b: **Thông số cân bằng của q giống với thông số cân bằng của p**. Thực hiện một phép **quay đơn** để chuyển các thông số *này thành `equal_height, shorter` vẫn là **`true.`***

      Trường hợp 3c: **Thông số cân bằng của q ngược với thông số cân bằng của p.** Thực hiện một phép **quay kép** (trước hết quay quanh q, sau đó quay quanh p), thông số cân bằng của gốc mới sẽ là `equal_height,` các thông số cân bằng của p và q được biến đổi thích hợp, `shorter` vẫn là **`true.`**

Trong các trường hợp 3a, b, c, chiều của các phép quay phụ thuộc vào cây con trái hay cây con phải bị ngắn đi. Hình 9.21 minh họa một vài trường hợp, và một ví dụ loại một nút được minh họa trong hình 9.22.



Hình 9.21 – Các trường hợp loại một nút ra khỏi cây AVL.



Hình 9.22 – Ví dụ loại một nút ra khỏi cây AVL.

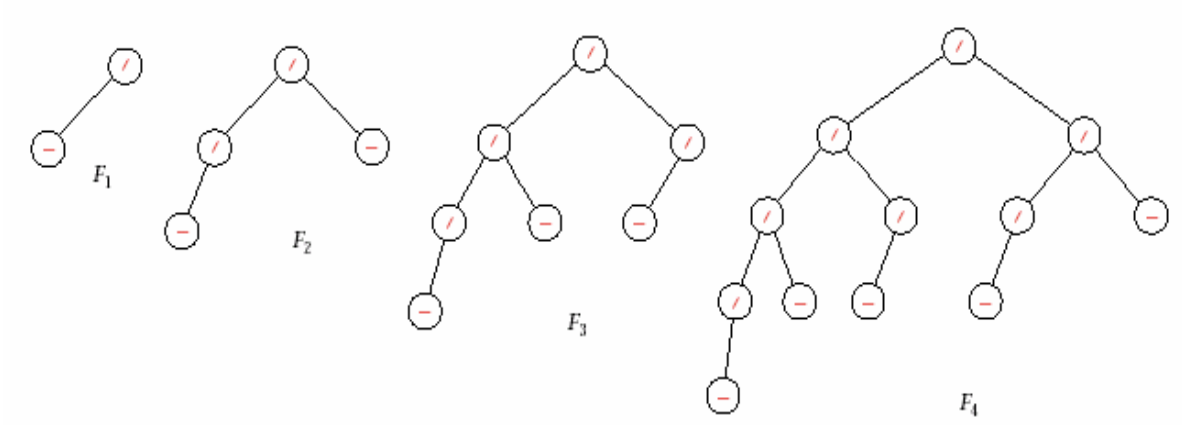
9.5.4. Chiều cao của cây AVL

Việc tìm chiều cao của một cây AVL trung bình là một việc rất khó, do đó việc xác định số bước trung bình cần thực hiện bởi các giải thuật trong phần này cũng không dễ. Cách dễ nhất cho chúng ta là xác định những gì sẽ xảy ra trong trường hợp xấu nhất, và các kết quả này sẽ cho chúng ta thấy rằng các hành vi của cây AVL trong trường hợp xấu nhất cũng không tệ hơn các hành vi của các cây ngẫu nhiên. Bằng chứng thực nghiệm cho thấy các hành vi trung bình của các cây AVL tốt hơn rất nhiều so với các cây ngẫu nhiên, hầu như nó còn tốt ngang với những gì sẽ có được từ một cây cân bằng hoàn toàn.

Để xác định chiều cao tối đa có thể có được của một cây AVL có n nút, chúng ta sẽ xem thử với một chiều cao h cho trước, cây AVL sẽ có số nút tối thiểu là bao nhiêu. Gọi F_h là một cây như vậy, thì cây con trái và cây con phải của nút gốc của F_h

sẽ là F_l và F_r. Một trong hai cây con này phải có chiều cao là h-1, giả sử cây F_l, và cây con còn lại có chiều cao h-1 hoặc h-2. Do F_h có số nút tối thiểu của một cây AVL có chiều cao h, F_l cũng phải có số nút tối thiểu của cây AVL cao h-1 (có nghĩa là F_l có dạng F_{h-1}), và F_r phải có chiều cao h-2 với số nút tối thiểu (F_r có dạng F_{h-2}).

Các cây được tạo bởi quy tắc trên, nghĩa là các cây thỏa điều kiện cây AVL nhưng càng thưa càng tốt, được gọi là các cây *Fibonacci*. Một số ít cây đầu tiên có thể được thấy trong hình 9.23.



Hình 9.23 – Các cây Fibonacci

Với ký hiệu |T| biểu diễn số nút trong cây T, chúng ta có công thức sau:

$$| F_h | = | F_{h-1} | + | F_{h-2} | + 1.$$

với $| F_0 | = 1$ và $| F_1 | = 2$. Bằng cách thêm 1 vào cả hai vế, chúng ta thấy con số $| F_h | + 1$ thỏa định nghĩa của các số *Fibonacci* bậc 2. Bằng cách tính các số *Fibonacci* chúng ta có

$$| F_h | + 1 \approx \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2}$$

Lấy *logarit* hai vế và chỉ giữ lại các số lớn chúng ta có

$$h \approx 1.44 \lg | F_h |.$$

Kết quả cho thấy một cây AVL n nút thưa nhất sẽ có chiều cao xấp xỉ 1.44 lg n. Một cây nhị phân cân bằng hoàn toàn có n nút có chiều cao bằng lg n, còn cây suy thoái sẽ có chiều cao là n. Thời gian chạy các giải thuật trên cây AVL được bảo đảm không vượt quá 44 phần trăm thời gian cần thiết đối với cây tối ưu. Trong thực tế, các cây AVL còn tốt hơn rất nhiều. Điều này có thể được chứng minh như sau, ngay cả đối với các cây *Fibonacci* - các cây AVL trong trường hợp xấu nhất – thời gian tìm kiếm trung bình chỉ có 4 phần trăm lớn hơn cây tối ưu. Phần lớn các cây AVL sẽ không thưa thớt như các cây *Fibonacci*, và do đó thời gian tìm kiếm trung bình trên các cây AVL trung bình rất gần với cây tối ưu. Thực nghiệm cho thấy số lần so sánh trung bình khoảng lg n + 0.25 khi n lớn.

Chương 10 – CÂY NHIỀU NHÁNH

Chương này tiếp tục nghiên cứu về các cấu trúc dữ liệu cây, tập trung vào các cây mà số nhánh tại mỗi nút nhiều hơn hai. Chúng ta bắt đầu từ việc trình bày các mối nối trong cây nhị phân. Kế tiếp chúng ta tìm hiểu về một lớp của cây gọi là *trie* được xem như từ điển chứa các từ. Sau đó chúng ta tìm hiểu đến cây *B-tree* có ý nghĩa rất lớn trong việc truy xuất thông tin trong các tập tin. Mỗi phần trong số này độc lập với các phần còn lại. Cuối cùng, chúng ta áp dụng ý tưởng của *B-tree* để có được một lớp khác của cây nhị phân tìm kiếm gọi là cây đỏ-đen (*red-black tree*).

10.1. Vườn cây, cây, và cây nhị phân

Như chúng ta đã thấy, cây nhị phân là một dạng cấu trúc dữ liệu đơn giản và hiệu quả. Tuy nhiên, với một số ứng dụng cần sử dụng cấu trúc dữ liệu cây mà trong đó số con của mỗi nút chưa biết trước, cây nhị phân với hạn chế mỗi nút chỉ có tối đa hai con không đáp ứng được. Phần này làm sáng tỏ một điều ngạc nhiên thú vị và hữu ích: cây nhị phân cung cấp một khả năng biểu diễn những cây khác bao quát hơn.

10.1.1. Các tên gọi cho cây

Trước khi mở rộng về các loại cây, chúng ta xét đến các định nghĩa. Trong toán học, khái niệm cây có một ý nghĩa rộng: đó là một tập bất kỳ các điểm (gọi là **đỉnh**), và tập bất kỳ các cặp nối hai đỉnh khác nhau (gọi là **cạnh** hoặc nhánh) sao cho luôn có một dãy liên tục các cạnh (**đường đi**) từ một đỉnh bất kỳ đến một đỉnh bất kỳ khác, và không có chu trình, nghĩa là không có đường đi nào bắt đầu từ một đỉnh nào đó lại quay về chính nó.

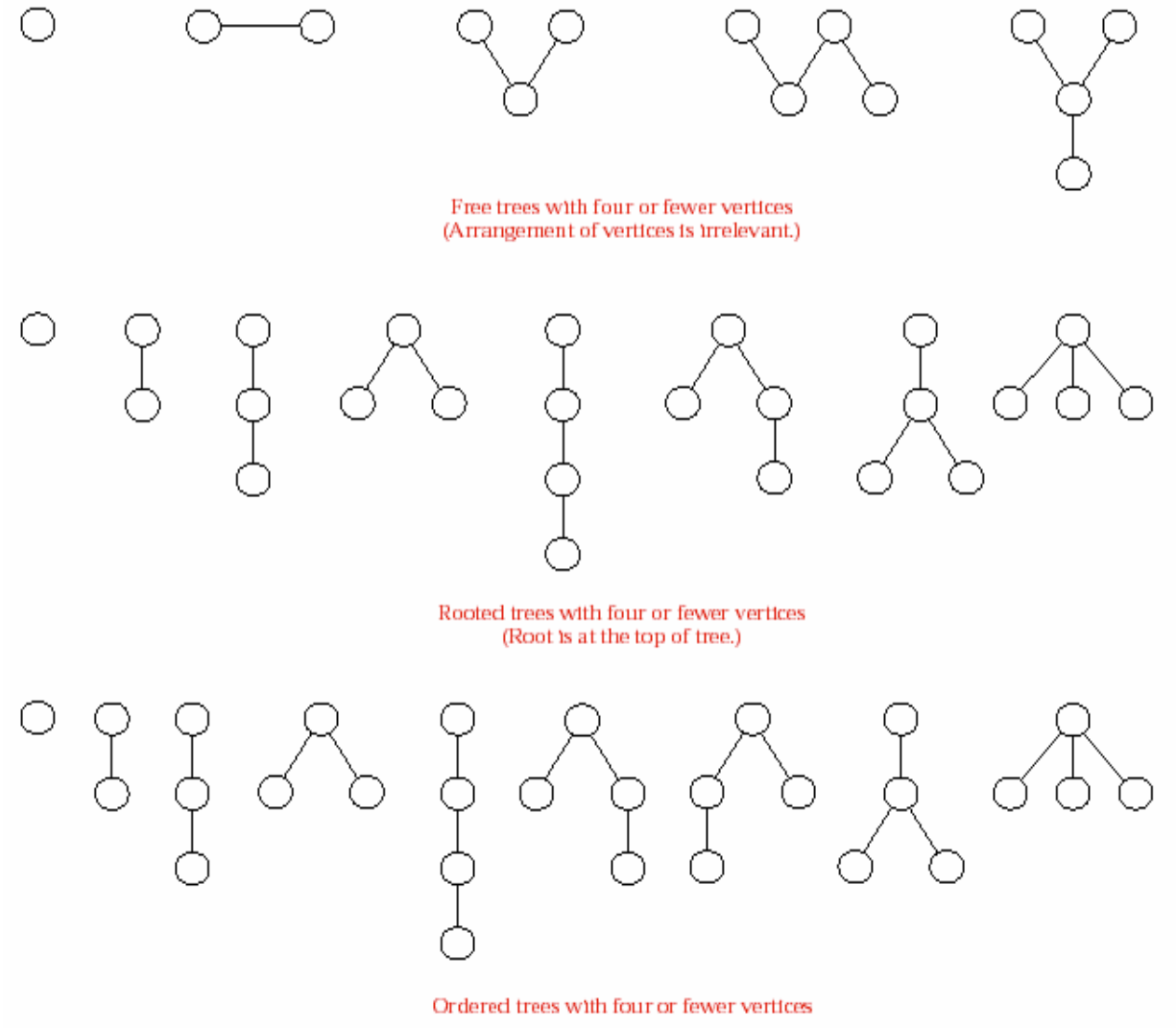
Đối với các ứng dụng trong máy tính, chúng ta thường không cần nghiên cứu cây một cách tổng quát như vậy, và khi cần làm việc với những cây này, để nhấn mạnh, chúng ta thường gọi chúng là các **cây tự do** (*free tree*). Các cây của chúng ta phần lớn luôn có một đỉnh đặc biệt, gọi là gốc của cây, và các cây dạng này chúng ta sẽ gọi là các **cây có gốc** (*rooted tree*).

Một cây có gốc có thể được vẽ theo cách thông thường của chúng ta là gốc nằm trên, các nút và nhánh khác quay xuống dưới, với các nút lá nằm dưới cùng. Mặc dù vậy, các cây có gốc vẫn chưa phải là tất cả các dạng cây mà chúng ta thường dùng. Trong một cây có gốc, thường không phân biệt trái hoặc phải, hoặc khi một nút có nhiều nút con, không thể nói rằng nút nào là nút con thứ nhất, thứ hai, v.v...Nếu không vì một lý do nào khác, sự thi hành tuần tự các lệnh thường buộc chặt một thứ tự lên các nút con của một nút. Chúng ta định nghĩa một cây có thứ

tự (*ordered tree*) là một cây có gốc trong đó các con của một nút được gán cho một thứ tự.

Lưu ý rằng các cây có thứ tự mà trong đó mỗi nút có không quá hai con vẫn chưa phải cùng một lớp với cây nhị phân. Nếu một nút trong cây nhị phân chỉ có một con, nó có thể nằm bên trái hoặc bên phải, lúc đó ta có hai cây nhị phân khác nhau, nhưng chúng cùng là một cây có thứ tự.

Như một nhận xét cuối cùng liên quan đến các định nghĩa, chúng ta hãy lưu ý rằng cây *2-tree* mà chúng ta đã nghiên cứu khi phân tích các giải thuật ở những chương trước là một cây có gốc (nhưng không nhất thiết phải là cây có thứ tự) với đặc tính là mỗi nút trong cây có 0 hoặc 2 nút con.



Hình 10.1 - Các dạng khác nhau của cây.

Hình 10.1 cho thấy rất nhiều dạng cây khác nhau với số nút nhỏ. Mỗi lớp cây kể từ cây đầu tiên có được bằng cách kết hợp các cây từ các lớp có trước theo nhiều cách khác nhau. Các cây nhị phân có thể có được từ các cây có thứ tự tương ứng, bằng cách phân biệt các nhánh trái và phải.

10.1.2. Cây có thứ tự

10.1.2.1. Hiện thực trong máy tính

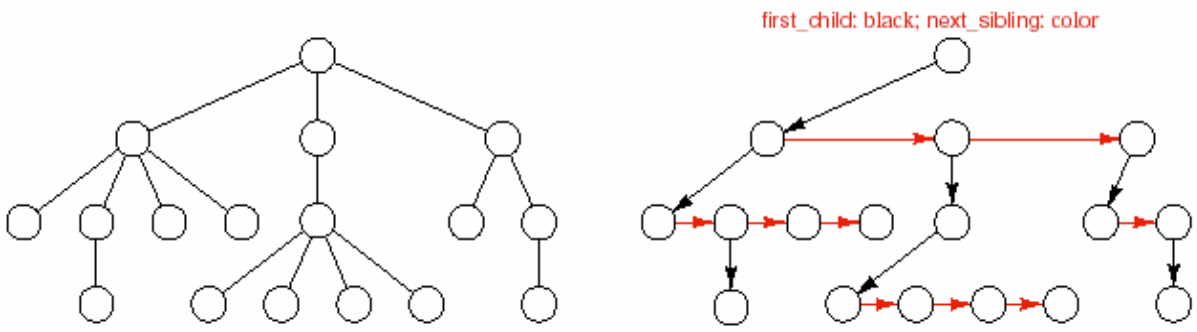
Nếu chúng ta muốn sử dụng một cây có thứ tự như một cấu trúc dữ liệu, một cách hiển nhiên để hiện thực trong bộ nhớ máy tính là mở rộng cách hiện thực chuẩn của một cây nhị phân, với số con trỏ thành viên trong mỗi nút tương ứng số cây con có thể có, thay vì chỉ có hai như đối với cây nhị phân. Chẳng hạn, trong một cây có một vài nút có đến mười cây con, chúng ta cần phải giữ đến mười con trỏ thành viên trong một nút. Nhưng như vậy sẽ dẫn đến việc cây phải chứa một số rất lớn các con trỏ chứa trị `NULL`. Chúng ta có thể tính được chính xác con số này. Nếu cây có n nút, mỗi nút có k con trỏ thành viên, thì sẽ có tất cả là n x k con trỏ. Mỗi nút có chính xác là một con trỏ tham chiếu đến nó, ngoại trừ nút gốc. Như vậy có n-1 con trỏ khác `NULL`. Tỉ lệ các con trỏ `NULL` sẽ là:

$$\frac{(n \times k) - (n - 1)}{n \times k} > 1 - \frac{1}{k}$$

Nếu một nút có thể có mười cây con, thì có hơn 90% con trỏ là `NULL`. Rõ ràng là phương pháp biểu diễn cây có thứ tự này hao tốn rất nhiều vùng nhớ. Lý do là vì, trong mỗi nút, chúng ta đã giữ một danh sách liên tục các con trỏ đến tất cả các con của nó, và các danh sách liên tục này chứa quá nhiều vùng nhớ chưa được sử dụng. Chúng ta cần tìm cách thay thế các danh sách liên tục này bởi các danh sách liên kết.

10.1.2.2. Hiện thực liên kết

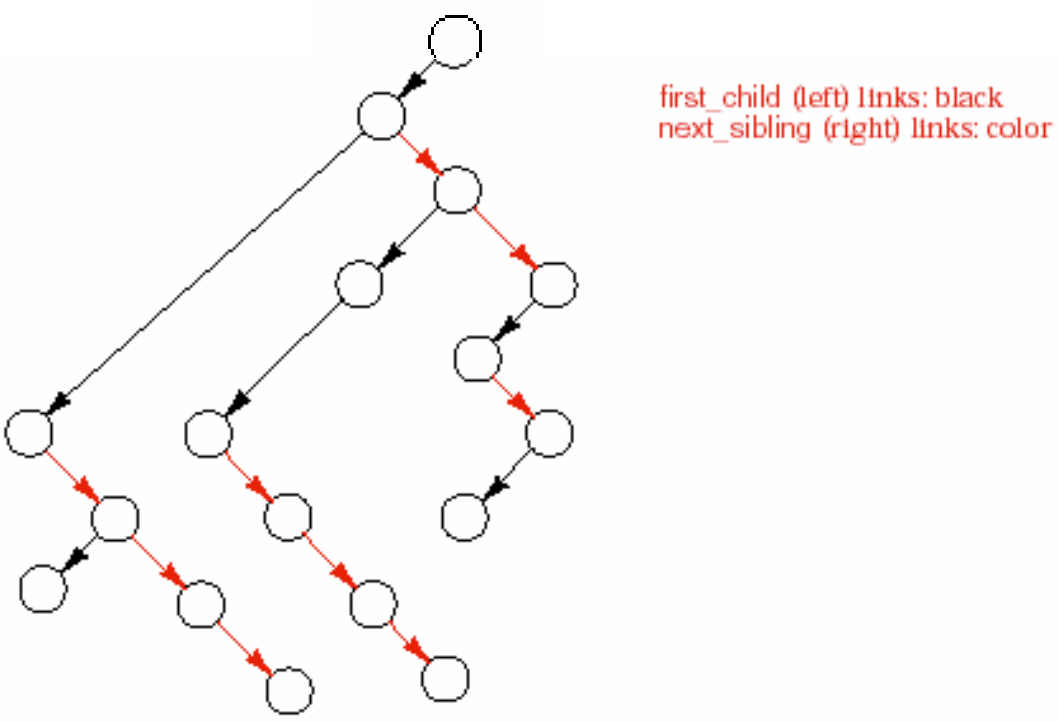
Để nắm các con của một nút trong một danh sách liên kết, chúng ta cần hai loại tham chiếu. Thứ nhất là tham chiếu từ nút cha đến nút con đầu tiên bên trái của nó, chúng ta sẽ gọi là **`first_child`**. Thứ hai, mỗi nút, ngoại trừ nút gốc, sẽ xuất hiện như một phần tử trong danh sách liên kết này, do đó nó cần thêm một tham chiếu đến nút kế trong danh sách, nghĩa là tham chiếu đến nút con kế tiếp cùng cha. Tham chiếu thứ hai này được gọi là **`next_sibling`**. Hiện thực này được minh họa trong hình 10.2.



Hình 10.2 – Hiện thực liên kết của cây có thứ tự

10.1.2.3. Sự tương ứng tự nhiên

Đối với mỗi nút của cây có thứ tự chúng ta đã định nghĩa hai tham chiếu **`first_child`** và **`next_sibling`**. Bằng cách sử dụng hai tham chiếu này chúng ta có được cấu trúc của một cây nhị phân, nghĩa là, hiện thực liên kết của một cây có thứ tự là một cây nhị phân liên kết. Nếu muốn, chúng ta có thể có được một hình ảnh dễ nhìn hơn cho cây nhị phân bằng cách sử dụng hiện thực liên kết của cây có thứ tự và quay theo chiều kim đồng hồ một góc nhỏ, sao cho các tham chiếu hướng xuống (**`first_child`**) hướng sang trái, và các tham chiếu nằm ngang (**`next_sibling`**) hướng sang phải. Đối với hình 10.2, chúng ta có được cây nhị phân ở hình 10.3.



Hình 10.3 – Hình đã được quay của hiện thực liên kết

10.1.2.4. Sự tương ứng ngược lại

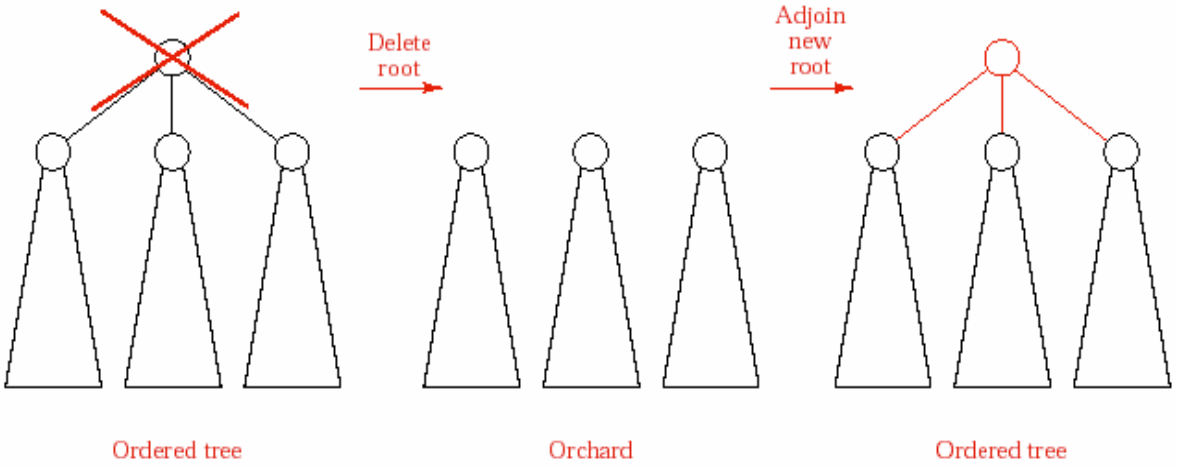
Giả sử như chúng ta làm ngược lại các bước của quá trình trên, bắt đầu từ một cây nhị phân và cố gắng khôi phục lại một cây có thứ tự. Điều quan sát đầu tiên chúng ta cần nhận thấy là không phải mọi cây nhị phân đều có thể có được từ một cây có thứ tự bởi quá trình trên: do tham chiếu **`next_sibling`** của nút gốc của cây có thứ tự luôn bằng `NULL` nên gốc của cây nhị phân tương ứng luôn có cây con bên phải rỗng. Để tìm hiểu sự tương ứng ngược lại này một cách cẩn thận, chúng ta cần phải xem xét một lớp cấu trúc dữ liệu khác qua một số định nghĩa mới dưới đây.

10.1.3. Rừng và vườn

Trong quá trình tìm hiểu về cây nhị phân chúng ta đã có kinh nghiệm về cách sử dụng đệ quy, đối với các lớp khác của cây chúng ta cũng sẽ tiếp tục làm như vậy. Sử dụng đệ quy có nghĩa là thu hẹp vấn đề thành vấn đề nhỏ hơn. Do đó chúng ta nên xem thử điều gì sẽ xảy ra nếu chúng ta lấy **một cây có gốc** hoặc **một cây có thứ tự** và cắt bỏ đi nút gốc. Những phần còn lại, nếu không rỗng, sẽ là **một tập các cây có gốc** hoặc **một tập có thứ tự các cây có thứ tự** tương ứng.

Thuật ngữ chuẩn để gọi một tập trừu tượng các cây đó là rừng (*forest*), nhưng khi chúng ta dùng thuật ngữ này, nói chung chúng ta thường hình dung đó là các **cây có gốc**. Cụm từ "rừng có thứ tự" (*ordered forest*) đôi khi còn được sử dụng để gọi **tập có thứ tự các cây có thứ tự**, do đó chúng ta sẽ đề cử một thuật ngữ có tính đặc tả tương tự cho **lớp các cây có thứ tự**, đó là thuật ngữ **vườn** (*orchard*).

Lưu ý rằng chúng ta không chỉ có được một **rừng** hoặc một **vườn** nhờ vào cách loại bỏ đi nút gốc của một **cây có gốc** hoặc một **cây có thứ tự**, chúng ta còn có thể tạo nên một **cây có gốc** hoặc một **cây có thứ tự** bằng cách bắt đầu từ một **rừng** hoặc một **vườn**, thêm một nút mới tại đỉnh, và nối các nhánh từ nút mới này đến gốc của tất cả các cây trong rừng hoặc vườn đó. Cách này được minh họa trong hình 10.4.



Hình 10.4 – Loại bỏ và thêm nút gốc.

Chúng ta sẽ sử dụng quá trình này để đưa ra một định nghĩa đệ quy mới cho các cây có thứ tự và các vườn. Trước hết, chúng ta hãy xem thử nên bắt đầu như thế nào. Chúng ta nhớ rằng một cây nhị phân có thể rỗng. Một rừng hay một vườn cũng có thể rỗng. Tuy nhiên một cây có gốc hay một cây có thứ tự không thể là cây rỗng, vì nó phải chứa ít nhất là một nút gốc. Nếu chúng ta muốn bắt đầu xây dựng cây và rừng, chúng ta có thể lưu ý rằng một cây với chỉ một nút có thể có được bằng cách thêm một gốc mới vào một rừng đang rỗng. Một khi chúng ta đã có cây này rồi thì chúng ta có thể tạo được một rừng gồm bao nhiêu cây một nút cũng được. Sau đó chúng ta có thể thêm gốc mới để tạo các cây có gốc chiều

cao là 1. Bằng cách này chúng ta có thể tiếp tục tạo nên các cây có gốc phù hợp với định nghĩa đệ quy sau:

> *Định nghĩa*: Một **cây có gốc** (*rooted tree*) bao gồm một nút đơn v, gọi là gốc (*root*) của cây, và một **rừng** F (*forest*) gồm các cây gọi là các cây con của nút gốc.

Một rừng *F* là một tập (có thể rỗng) các cây có gốc.

Một quá trình tạo tương tự cho các cây có thứ tự và vườn.

> *Định nghĩa*: Một **cây có thứ tự** *T* (*ordered tree*) bao gồm một nút đơn v, gọi là gốc (*root*) của cây,và một **vườn** *O* (*orchard*) gồm các cây được gọi là các cây con của gốc v.

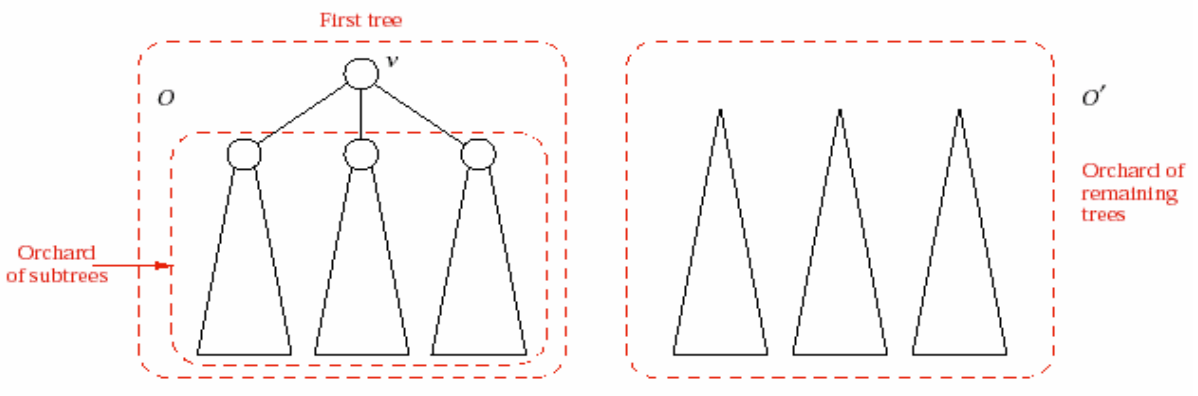
Chúng ta có thể biểu diễn cây có thứ tự bằng một cặp có thứ tự

$$T = \{v, O\}.$$

Một vườn *O* hoặc là một tập rỗng, hoặc gồm một cây có thứ tự *T*, gọi là cây thứ nhất (*first tree*) của vườn, và một vườn khác *O'* (chứa các cây còn lại của vườn). Chúng ta có thể biểu diễn vườn bằng một cặp có thứ tự

$$O = (T, O').$$

Lưu ý rằng thứ tự của các cây ẩn chứa trong định nghĩa của vườn. Một vườn không rỗng chứa cây thứ nhất và các cây còn lại tạo nên một vườn khác, vườn này lại có một cây thứ nhất và là cây thứ hai của vườn ban đầu. Tiếp tục đối với các vườn còn lại chúng ta có cây thứ ba, thứ tư, v.v...cho đến khi vườn cuối cùng là một vườn rỗng. Xem hình 10.5.



Hình 10.5 – Cấu trúc đệ quy của các cây có thứ tự và vườn.

10.1.4. Sự tương ứng hình thức

Bây giờ chúng ta có thể có một kết quả mang tính nguyên tắc cho phần này.

Định lý: *Cho S là một tập hữu hạn bất kỳ gồm các nút. Có một ánh xạ một-một f từ tập các vườn có tập nút là S đến tập các cây nhị phân có tập nút là S.*

Chứng minh định lý:

Chúng ta sẽ dùng những ký hiệu trong các định nghĩa để chứng minh định lý trên. Trước hết chúng ta cần một ký hiệu tương tự cho cây nhị phân. Một cây nhị phân B hoặc là một tập rỗng ∅ hoặc gồm một nút gốc ν và hai cây nhị phân B_1 và B_2. Ký hiệu cho một cây nhị phân không rỗng là một bộ ba

$$B = [\nu, B_1, B_2].$$

Chúng ta sẽ chứng minh định lý bằng phương pháp quy nạp toán học trên số nút trong *S*. Trường hợp thứ nhất được xét là một vườn rỗng ∅, tương ứng với một cây nhị phân rỗng.

$$f(\varnothing) = \varnothing.$$

Nếu vườn *O* không rỗng, nó được ký hiệu bằng một bộ hai

$$O = (T, O_2)$$

với T là một cây có thứ tự và O_2 là một vườn khác. Cây thứ tự T được ký hiệu bởi một cặp

$$T = \{\nu, O_1\}$$

với ν là một nút và O_1 là một vườn khác. Thay biểu thức T vào biểu thức O ta có

$$O = (\{\nu, O_1\}, O_2).$$

Theo giả thiết quy nạp, f là một ánh xạ một-một từ các vườn có ít nút hơn *S* đến các cây nhị phân, với O_1 và O_2 nhỏ hơn *O*, nên các cây nhị phân $f(O_1)$ và $f(O_2)$ được xác định bởi giả thiết quy nạp. Nếu chúng ta định nghĩa ánh xạ f từ một vườn đến một cây nhị phân bởi

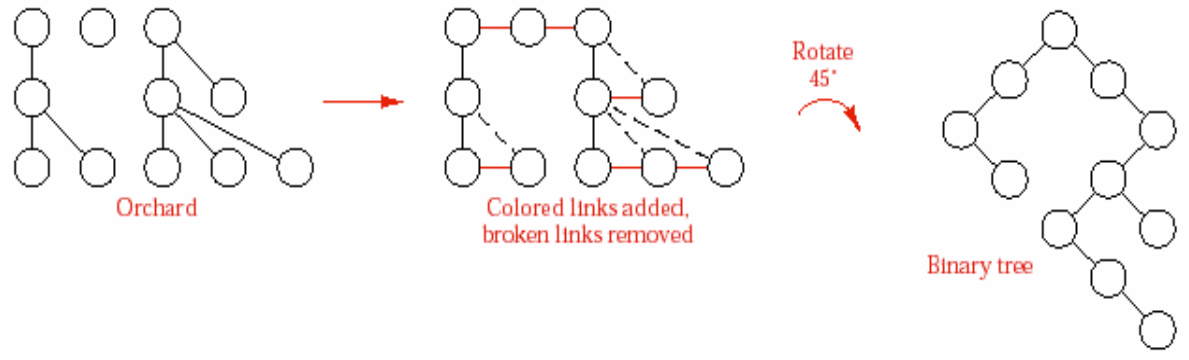
$$f(\{\nu, O_1\}, O_2) = [\nu, f(O_1), f(O_2)].$$

thì f là một sự tương ứng một-một giữa các vườn và các cây nhị phân có cùng số nút. Với bất kỳ cách thay thế nào cho các ký tự ν, O_1, và O_2 ở vế trái đều có chính xác một cách để thay thế cho chúng ở vế phải, và ngược lại.

10.1.5. Phép quay

Chúng ta có thể sử dụng dạng ký hiệu của sự tương ứng để hình dung phép biến đổi từ vườn sang cây nhị phân. Trong cây nhị phân [v, $f(O_1)$, $f(O_2)$] tham chiếu trái từ v đến nút gốc của cây nhị phân $f(O_1)$, đó là nút con thứ nhất của v trong cây có thứ tự {v, O_1}. Tham chiếu phải từ v đến nút vốn là gốc của cây có thứ tự kế tiếp về bên phải trong vườn. Có nghĩa là, "tham chiếu trái" trong cây nhị phân tương ứng với "con thứ nhất" trong cây có thứ tự, và "tham chiếu phải" tương ứng "em kế". Các quy tắc biến đổi trong hình như sau:

1. Vẽ vườn sao cho con thứ nhất của mỗi nút nằm ngay dưới nó, thay vì canh khoảng cách cho tất cả các con nằm đều bên dưới nút này.
2. Vẽ một tham chiếu thẳng đứng từ mỗi nút đến nút con thứ nhất của nó, và vẽ một tham chiếu nằm ngang từ mỗi nút đến em kế của nó.
3. Loại bỏ tất cả các tham chiếu khác còn lại.
4. Quay sơ đồ 45 độ theo chiều kim đồng hồ, sao cho các tham chiếu thẳng đứng trở thành các tham chiếu trái và các tham chiếu nằm ngang trở thành các tham chiếu phải.
5. Quá trình này được minh họa trong hình 10.6



Hình 10.6 – Chuyển đổi từ vườn sang cây nhị phân.

10.1.6. Tổng kết

Chúng ta đã xem xét ba cách biểu diễn sự tương ứng giữa các vườn và các cây nhị phân:

- Các tham chiếu **first_child** và **next_sibling**.
- Phép quay các sơ đồ.
- Sự tương đương ký hiệu một cách hình thức.

Nhiều người cho rằng cách thứ hai, quay các sơ đồ, là cách dễ nhớ và dễ hình dung nhất. Cách thứ nhất, tạo các tham chiếu, thường được dùng để viết các chương trình thực sự. Cuối cùng, cách thứ ba, sự tương đương ký hiệu một cách

hình thức, thường rất có ích trong việc chứng minh rất nhiều đặc tính của cây nhị phân và vườn.

10.2. Cây từ điển tìm kiếm: Trie

Trong các chương trước chúng ta đã thấy sự khác nhau trong việc tìm kiếm trong một danh sách và việc tra cứu trong một bảng. Chúng ta có thể áp dụng ý tưởng trong việc tra cứu bảng vào việc truy xuất thông tin trong một cây bằng cách sử dụng một khóa hoặc một phần của khóa. Thay vì tìm kiếm bằng cách so sánh các khóa, chúng ta có thể xem khóa như là một chuỗi các ký tự (chữ cái hoặc ký số), và sử dụng các ký tự này để xác định đường đi tại mỗi bước. Nếu các khóa của chúng ta chứa các chữ cái, chúng ta sẽ tạo một cây có 26 nhánh tương ứng 26 chữ cái là ký tự đầu tiên của các khóa. Mỗi cây con bên dưới lại có 26 nhánh tương ứng với ký tự thứ hai, và cứ thế tiếp tục ở các mức cao hơn. Tuy nhiên chúng ta cũng có thể tiến hành phân thành nhiều nhánh ở một số mức ban đầu, sau đó nếu cây trở nên quá lớn, chúng ta có thể dùng một vài cách thức khác nào đó để sắp thứ tự cho những mức còn lại.

10.2.1. Tries

Có một phương pháp là cắt tỉa bớt các nhánh không cần thiết trong cây. Đó là các nhánh không dẫn đến một khóa nào. Lấy ví dụ, trong tiếng Anh, không có các từ bắt đầu bởi 'bb', 'bc', 'bf', 'bg', ..., nhưng có các từ bắt đầu bởi 'ba', 'bd', 'be'. Do đó, mọi nhánh và nút cho các từ không tồn tại có thể được loại khỏi cây. Cây kết quả này được gọi là **`Trie`**. Từ này nguyên thủy được lấy từ *re**trie**val*, nhưng thường được đọc là "*try*".

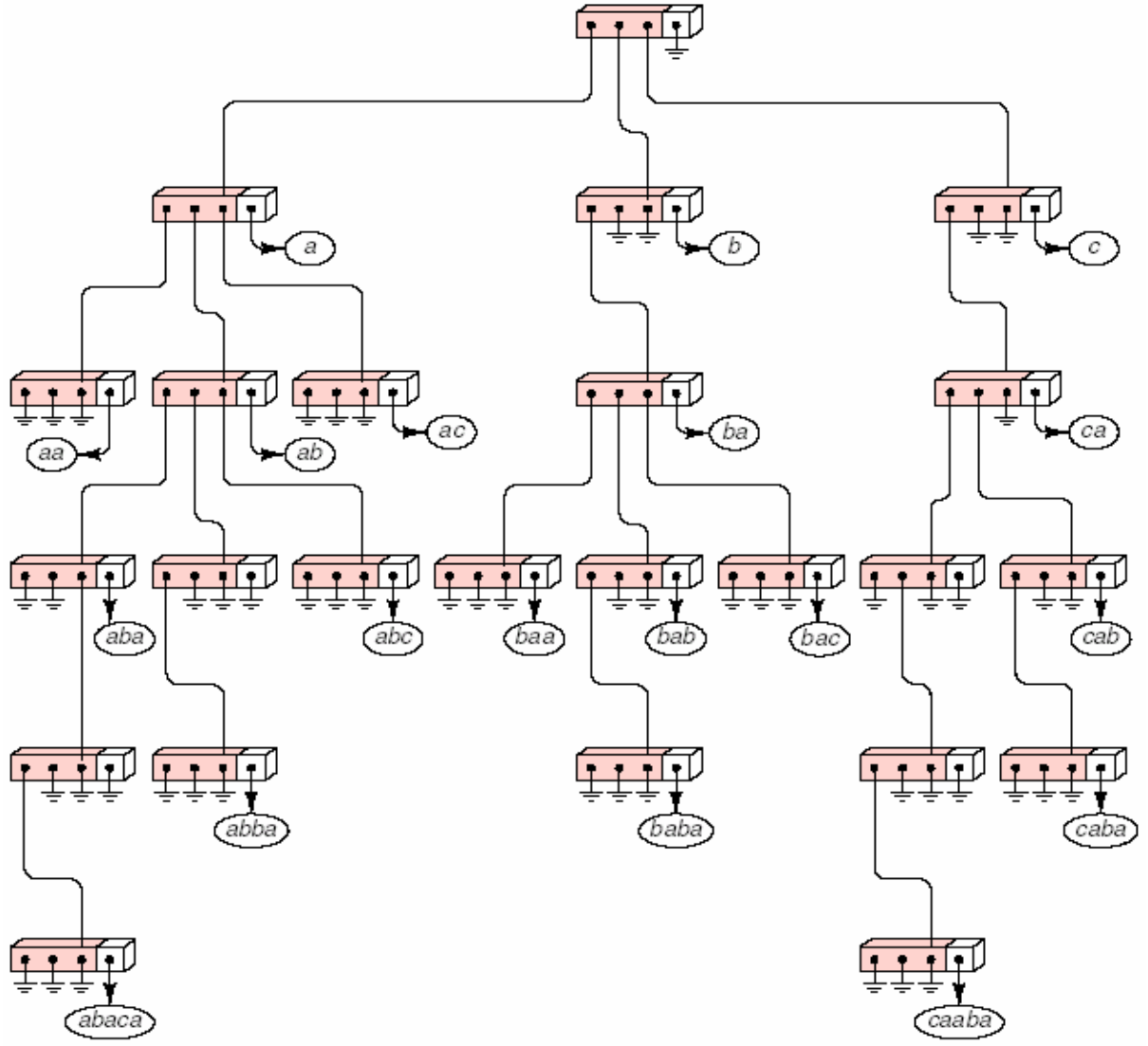
> *Định nghĩa*: Một cây `Trie` bậc m có thể được định nghĩa một cách hình thức là một cây rỗng hoặc gồm một chuỗi nối tiếp có thứ tự của m cây `Trie` bậc m.

10.2.2. Tìm kiếm một khóa

Giả sử các từ có 3 ký tự có nghĩa gồm các từ được lưu trong cây `Trie` ở hình 10.7. Việc tìm kiếm một khóa được bắt đầu từ nút gốc. Ký tự đầu tiên của khóa được dùng để xác định nhánh nào cần đi xuống. Nhánh cần đi rỗng có nghĩa là khóa cần tìm chưa có trong cây. Ngược lại, trên nhánh được chọn này, ký tự thứ hai lại được dùng để xác định nhánh nào trong mức kế tiếp cần đi xuống, và cứ thế tiếp tục. Khi chúng ta xét đến cuối từ, là chúng ta đã đến được nút có con trỏ tham chiếu đến thông tin cần tìm. Đối với nút tương ứng một từ không có nghĩa sẽ có con trỏ tham chiếu đến thông tin là `NULL`. Chẳng hạn, từ **a** là phần đầu của từ **aba**, từ này lại là phần đầu của từ **abaca**, nhưng chuỗi ký tự **abac** không phải là một từ có nghĩa, do đó nút biểu diễn **abac** có con trỏ tham chiếu thông tin là `NULL`.

10.2.3. Giải thuật C++

Chúng ta sẽ chuyển quá trình tìm kiếm vừa được mô tả trên thành một phương thức tìm kiếm các bản ghi có khóa là các chuỗi ký tự. Chúng ta sẽ sử dụng phương thức **`char key_letter(int position)`** trả về ký tự tại vị trí **`position`** trong khóa hoặc ký tự rỗng nếu khóa có chiều dài ngắn hơn **`position`**,



Hình 10.7 – Trie chứa các từ được cấu tạo từ a, b, c.

và hàm phụ trợ **`int alphabetic_order(char symbol)`** trả về thứ tự của **`symbol`** trong bảng chữ cái. Hàm này trả về 0 cho ký tự rỗng, 27 cho các ký tự không phải chữ cái. Trong hiện thực liên kết, cây *`Trie`* chứa một con trỏ đến nút gốc của nó.

```
class Trie {
public:  // Các phương thức cập nhật, tìm kiếm, truy xuất.

private:
   Trie_node *root;
};
```

Mỗi nút của Trie cần chứa một con trỏ chỉ đến một bản ghi và một mảng các con trỏ đến các nhánh. Số nhánh là 28 tương ứng kết quả trả về của **alphabetic_order**.

```
const int num_chars = 28;
struct Trie_node {
//    Các thuộc tính
   Record *data;
   Trie_node *branch[num_chars];
//    constructors
   Trie_node();
};
```

Constructor cho **Trie_node** đơn giản chỉ gán tất cả các con trỏ là NULL.

10.2.4. Tìm kiếm trong cây Trie

Phương thức sau tìm một bản ghi chứa khóa cho trước trong cây Trie.

```
Error_code Trie::trie_search(const Key &target, Record &x) const
/*
post: Nếu tìm thấy khóa target, bản ghi x chứa khóa sẽ được trả về, phương thức trả về
      success. Ngược lại phương thức trả về not_present.
uses: Các phương thức của lớp Key.
*/
{
   int position = 0;
   char next_char;
   Trie_node *location = root;
   while (location!=NULL&&(next_char=target.key_letter(position))!=' ')
   {
      location = location->branch[alphabetic_order(next_char)];
                 // Đi xuống dần các nhánh tương ứng với các ký tự trong target.
      position++;// Để xét ký tự kế tiếp của target.
   }

   if (location != NULL && location->data != NULL) {
      x = *(location->data);
      return success;
   }
   else
      return not_present;
}
```

Điều kiện kết thúc vòng lặp là con trỏ location bằng NULL (khóa cần tìm không có trong cây), hoặc ký tự kế là rỗng (đã xét hết chiều dài khóa cần tìm). Kết thúc vòng lặp, con trỏ **location** nếu khác NULL chính là con trỏ tham chiếu bản ghi chứa khóa cần tìm.

10.2.5. Thêm phần tử vào Trie

Thêm một phần tử vào cây Trie hoàn toàn tương tự như tìm kiếm: lần theo các nhánh để đi xuống cho đến khi gặp vị trí thích hợp, tạo bản ghi chứa dữ liệu

và cho con trỏ **data** chỉ đến. Nếu trên đường đi chúng ta gặp một nhánh NULL, chúng ta phải tạo thêm các nút mới để đưa vào cây sao cho có thể tạo được một đường đi đến nút tương ứng với khóa mới cần thêm vào.

```
Error_code Trie::insert(const Record &new_entry)
/*
post: Nếu khóa của new_entry đã có trong Trie, phương thức trả về duplicate_error.
      Ngược lại new_entry được thêm vào Trie, phương thức trả về success.
uses: các phương thức của các lớp Record và Trie_node.
*/
{
   Error_code result = success;
   if (root == NULL) root = new Trie_node; // Tạo một cây Trie rỗng.
   int position = 0;                // Vị trí ký tự đang xét trong new_entry.
   char next_char;
   Trie_node *location = root;      //  Đi dần xuống các nhánh trong Trie.
   while (location != NULL &&
          (next_char = new_entry.key_letter(position)) != ' ') {
      int next_position = alphabetic_order(next_char);
      if (location->branch[next_position] == NULL)
         location->branch[next_position] = new Trie_node;
      location = location->branch[next_position];
      position++;
   }
            // Không còn nhánh để đi tiếp hoặc đã xét hết các ký tự của new_entry.
   if (location->data != NULL) result = duplicate_error;
   else location->data = new Record(new_entry);
   return result;
}
```

10.2.6. Loại phần tử trong Trie

Cách thực hiện của việc thêm và tìm kiếm phần tử cũng được áp dụng cho việc loại một phần tử trong cây Trie. Chúng ta lần theo đường đi tương ứng với khóa cần loại, khi gặp nút này, chúng ta gán NULL cho con trỏ **data**. Tuy nhiên, nếu nút này có tất cả các thuộc tính đều là các con trỏ NULL (các cây con và con trỏ **data**), chúng ta cần xóa luôn chính nó. Và điều này cần phải được thực hiện cho tất cả các nút trên của nó trên đường đi từ nó ngược về nút gốc cho đến khi gặp một nút có ít nhất một thuộc tính thành viên khác NULL. Để làm được điều này, chúng ta có thể tạo một ngăn xếp chứa các con trỏ đến các nút trên đường đi từ nút gốc đến nút cần tìm để loại. Hoặc chúng ta có thể sử dụng đệ quy trong giải thuật loại phần tử nhằm tránh việc sử dụng ngăn xếp một cách tường minh. Cả hai cách này đều được xem như bài tập.

10.2.7. Truy xuất Trie

Số bước cần thực hiện để tìm kiếm trong cây Trie (hoặc thêm nút mới vào Trie) tỉ lệ với số ký tự tạo nên một khóa, không phụ thuộc vào *logarit* của số khóa như các cách tìm kiếm dựa trên các cây khác. Nếu số ký tự nhỏ so với *logarit* cơ số 2 của số khóa, cây Trie tỏ ra có ưu thế hơn cây nhị phân tìm kiếm

nhiều. Lấy ví dụ, các khóa gồm mọi khả năng của một chuỗi 5 ký tự, thì cây `Trie` có thể chứa đến $n = 26^5 = 11{,}881{,}376$ khóa với mỗi lần tìm kiếm tối đa là 5 lần lặp để đi xuống 5 mức, trong khi đó cây nhị phân tìm kiếm tốt nhất có thể thực hiện đến $\lg n \approx 23.5$ lần so sánh các khóa.

Tuy nhiên, trong nhiều ứng dụng có số ký tự trong một khóa lớn, và tập các khóa thực sự xuất hiện lại ít so với mọi khả năng có thể có của các khóa. Trong trường hợp này, số lần lặp cần có để tìm một khóa trong cây *Trie* có thể vượt xa số lần so sánh các khóa cần có trong cây nhị phân tìm kiếm.

Cuối cùng, lời giải tốt nhất có thể là sự kết hợp của nhiều phương pháp. Cây *Trie* có thể được sử dụng cho một ít ký tự đầu của các khóa, và sau đó một phương pháp khác có thể được sử dụng cho phần còn lại của khóa.

10.3. Tìm kiếm ngoài: B-tree

Từ trước đến nay, chúng ta đã giả sử rằng mọi cấu trúc dữ liệu đều được giữ trong bộ nhớ tốc độ cao; nghĩa là chúng ta đã chỉ xem xét việc truy xuất thông tin trong (*internal information retrieval*). Với một số ứng dụng, giả thiết này có thể chấp nhận được, nhưng với nhiều ứng dụng quan trọng khác thì không. Chúng ta hãy xem xét vấn đề truy xuất thông tin ngoài (*external information retrieval*), trong đó các bản ghi cần tìm kiếm và truy xuất được lưu trong các tập tin.

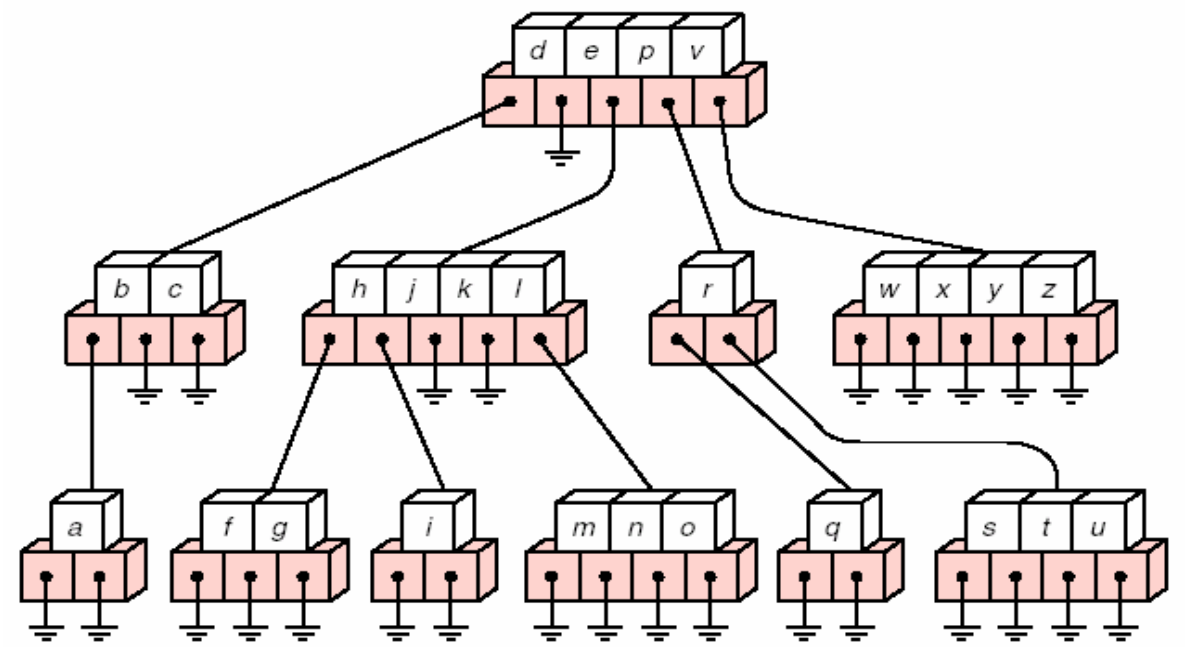
10.3.1. Thời gian truy xuất

Thời gian cần có để thâm nhập và truy xuất một từ trong bộ nhớ tốc độ cao nhiều nhất là một vài microgiây. Thời gian cần để định vị một bản ghi trong đĩa cứng được đo bằng miligiây, đối với đĩa mềm có thể vượt quá một giây. Như vậy thời gian cho một lần truy xuất ngoài lớn gấp hàng ngàn lần so với một lần truy xuất trong. Khi một bản ghi nằm trong đĩa, thực tế mỗi lần không phải chỉ đọc một từ, mà đọc một trang lớn (*page*) hay còn gọi là một khối (*block*) thông tin. Kích thước chuẩn của khối thường từ 256 đến 1024 ký tự hoặc từ.

Mục đích của chúng ta trong việc tìm kiếm ngoài là phải làm tối thiểu số lần truy xuất đĩa, do mỗi lần truy xuất chiếm thời gian đáng kể so với các tính toán bên trong bộ nhớ. Mỗi lần truy xuất đĩa, chúng ta có được một khối mà có thể chứa nhiều bản ghi. Bằng cách sử dụng các bản ghi này, chúng ta có thể chọn lựa giữa nhiều khả năng để quyết định khối nào sẽ được truy xuất kế tiếp. Nhờ đó mà toàn bộ dữ liệu không cần phải lưu đồng thời trong bộ nhớ. Khái niệm cây nhiều nhánh mà chúng ta sẽ xem xét dưới đây đặc biệt thích hợp đối với việc tìm kiếm ngoài.

10.3.2. Cây tìm kiếm nhiều nhánh

Cây nhị phân tìm kiếm được tổng quát hóa một cách trực tiếp đến cây tìm kiếm nhiều nhánh, trong đó, với một số nguyên m nào đó được gọi là bậc (*order*) của cây, mỗi nút có nhiều nhất m nút con. Nếu k ($k \le m$) là số con của một nút thì nút này chứa chính xác là k-1 khóa, và các khóa này phân hoạch tất cả các khóa của các cây con thành k tập con. Hình 10.8 cho thấy một cây tìm kiếm có 5 nhánh nằm xen kẽ các phần tử từ thứ 1 và đến thứ 4 trong mỗi nút, trong đó một vài nhánh có thể rỗng.



Hình 10.8 – Một cây tìm kiếm 5 nhánh (không phải cây B-tree)

10.3.3. Cây nhiều nhánh cân bằng

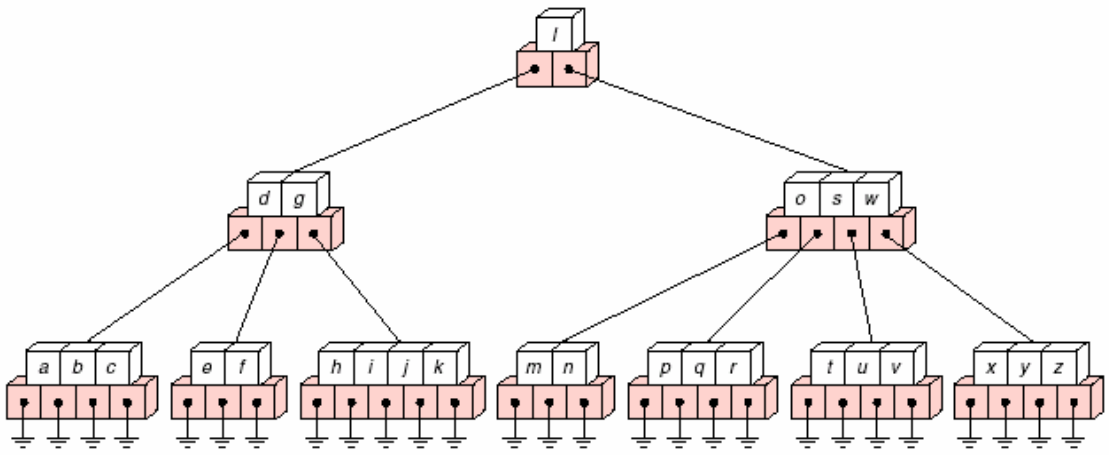
Giả sử mỗi lần đọc tập tin, chúng ta đọc lên được một khối chứa các khóa trong cùng một nút. Nhờ sự phân hoạch các khóa trong các cây con dựa trên các khóa này, chúng ta biết được nhánh nào chúng ta cần tiếp tục công việc tìm kiếm khóa cần tìm. Bằng cách này số lần đọc đĩa tối đa chính là chiều cao của cây. Và chi phí bộ nhớ cũng chỉ dành tối đa là cho các nút trên đường đi từ nút gốc đến nút có khóa cần tìm, chứ không phải toàn bộ dữ liệu lưu trong cây.

Mục đích của chúng ta sử dụng cây tìm kiếm nhiều nhánh để làm giảm việc truy xuất tập tin, do đó chúng ta mong muốn chiều cao của cây càng nhỏ càng tốt. Chúng ta có thể thực hiện điều này bằng cách cho rằng, thứ nhất, không có các cây con rỗng xuất hiện bên trên các nút lá (như vậy sự phân hoạch các khóa thành các tập con sẽ hiệu quả nhất); thứ hai, rằng mọi nút lá đều thuộc cùng một mức (để cho việc tìm kiếm được bảo đảm là sẽ kết thúc với cùng số lần truy xuất

tập tin); và, thứ ba, rằng mọi nút, ngoại trừ các nút lá có ít nhất một số nút con tối thiểu nào đó. Chúng ta đưa ra yêu cầu rằng, mọi nút, ngoại trừ các nút lá, có ít nhất là một nửa số con so với số con tối đa có thể có. Các điều kiện trên dẫn đến định nghĩa sau:

Định nghĩa: Một cây **B-tree bậc m** là một cây m nhánh, trong đó,
1. Mọi nút lá có cùng mức.
2. Mọi nút trung gian (không phải nút lá và nút gốc), có nhiều nhất m nút con khác rỗng, ít nhất là $\lceil m/2 \rceil$ nút con khác rỗng.
3. Số khóa trong mỗi nút trong nhỏ hơn số nút con khác rỗng 1 đơn vị, và các khóa này phân hoạch các khóa trong các cây con theo cách của cây tìm kiếm.
4. Nút gốc có nhiều nhất m nút con, và nếu nó không đồng thời là nút lá (trường hợp cây chỉ có 1 nút), thì nó có thể có ít nhất là 2 nút con.

Cây trong hình 10.8 không phải là cây B-tree, do một vài nút có các nút con rỗng, một vài nút có quá ít con, và các nút lá không cùng một mức. Hình 10.9 minh họa một cây B-tree có bậc là 5 với các khóa là các ký tự chữ cái. Trường hợp này mỗi nút trung gian có ít nhất 3 nút con (phân hoạch bởi 2 khóa).



Hình 10.9 – Cây B-tree bậc 5.

10.3.4. Thêm phần tử vào B-tree

Điều kiện mọi nút lá thuộc cùng mức nhấn mạnh hành vi đặc trưng của B-tree: Ngược với cây nhị phân tìm kiếm, B-tree không cho phép lớn lên tại các nút lá; thay vào đó, nó lớn lên tại gốc. Phương pháp chung để thêm phần tử vào nó như sau. Trước hết, thực hiện việc tìm kiếm để xem khóa cần thêm đã có trong cây hay chưa. Nếu chưa có, việc tìm kiếm sẽ kết thúc tại một nút lá. Khóa mới sẽ được thêm vào nút lá. Nếu nút lá vốn chưa đầy, việc thêm vào hoàn tất.

Khi nút lá cần thêm phần tử mới đã đầy, nút này sẽ được phân làm hai nút cạnh nhau trong cùng một mức, khóa chính giữa sẽ không thuộc nút nào trong hai nút này, nó được gởi ngược lên để thêm vào nút cha. Nhờ vậy, sau này, khi cần tìm kiếm, sự so sánh với khóa giữa này sẽ dẫn đường xuống tiếp cây con tương ứng bên trái hoặc bên phải. Quá trình phân đôi các nút có thể được lan truyền ngược về gốc. Quá trình này sẽ chấm dứt khi có một nút cha nào đó cần được thêm một khóa gởi từ dưới lên mà chưa đầy. Khi một khóa được thêm vào nút gốc đã đầy, nút gốc sẽ được phân làm hai và khóa nằm giữa cũng được gởi ngược lên, và nó sẽ trở thành một gốc mới. Đó chính là lúc duy nhất cây `B-tree` tăng trưởng chiều cao.

Quá trình này có thể được làm sáng tỏ bằng ví dụ thêm vào cây `B-tree` cấp 5 ở hình 10.10. Chúng ta sẽ lần lượt thêm các khóa

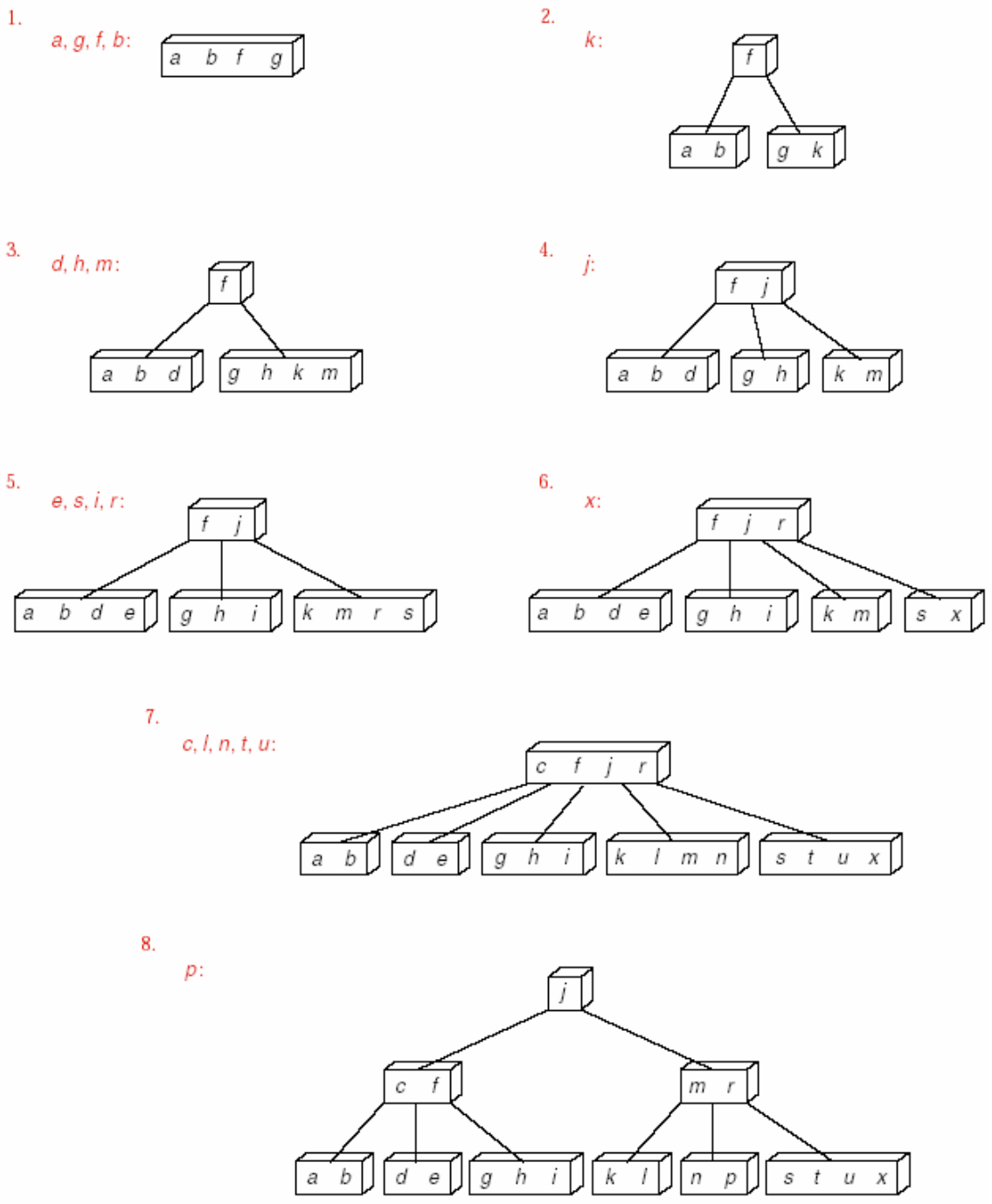
a g f b k d h m j e s i r x c l n t u p

vào một cây rỗng theo thứ tự này.

Bốn khóa đầu tiên sẽ được thêm vào chỉ một nút, như trong phần đầu của hình 10.10. Chúng được sắp thứ tự ngay khi được thêm vào. Tuy nhiên, đối với khóa thứ năm, **k**, nút này không còn chỗ. Nút này được phân làm hai nút mới, khóa nằm giữa, **f**, được chuyển lên trên và tạo nên nút mới, đó cũng là gốc mới. Do các nút sau khi phân chia chỉ chứa một nửa số khóa có thể có, ba khóa tiếp theo có thể được thêm vào mà không gặp khó khăn gì. Tuy nhiên, việc thêm vào đơn giản này cũng đòi hỏi việc tổ chức lại các khóa trong một nút. Để thêm **j**, một lần nữa lại cần phân chia một nút, và lần này khóa chuyển lên trên chính là **j**.

Một số lần thêm các khóa tiếp theo được thực hiện tương tự. Lần thêm cuối cùng, **p**, đặc biệt hơn. Việc thêm **p** vào trước tiên làm phân chia một nút vốn chứa **k, l, m, n**, và gởi khóa nằm giữa **m** lên trên cho nút cha chứa **c, f, j, r,** tuy nhiên, nút này đã đầy. Như vậy, nút này lại phân chia làm hai nút mới, và cuối cùng nút gốc mới chứa **j** được tạo ra.

Có hai điểm cần chú ý khi quan sát sự lớn lên có trật tự của `B-tree`. Thứ nhất, khi một nút được phân đôi, nó tạo ra hai nút mới, mỗi nút chỉ có một nửa số phần tử tối đa có thể có. Nhờ đó, những lần thêm tiếp theo có thể không cần phải phân chia nút lần nữa. Như vậy một lần phân chia nút là chuẩn bị cho một vài lần thêm đơn giản. Thứ hai, khóa được chuyển lên trên luôn là khóa nằm giữa chứ không phải chính khóa cần thêm vào. Do đó, nhiều lần thêm lập lại sẽ có chiều hướng cải thiện sự cân bằng cho cây, không phụ thuộc vào thứ tự các khóa được thêm vào.



Hình 10.10 – Sự lớn lên của cây B-tree.

10.3.5. Giải thuật C++: tìm kiếm và thêm vào

Để phát triển thành giải thuật C++ tìm kiếm và thêm vào một cây `B-tree`, chúng ta hãy bắt đầu với các khai báo cho cây. Để đơn giản chúng ta sẽ xây dựng cây `B-tree` trong bộ nhớ tốc độ cao, sử dụng các con trỏ chứa địa chỉ các nút trong cây. Trong phần lớn các ứng dụng, các con trỏ này có thể được thay thế bởi

địa chỉ của các khối hoặc trang trong đĩa, hoặc số thứ tự các bản ghi trong tập tin.

10.3.5.1. Các khai báo

Chúng ta sẽ cho người sử dụng tự do chọn lựa kiểu của bản ghi mà họ muốn lưu vào cây B-tree. Lớp **B-tree** của chúng ta, và lớp **node** tương ứng, sẽ có thông số template là lớp **Record**. Thông số template thứ hai sẽ là một số nguyên biểu diễn bậc của B-tree. Để có được một đối tượng *B*-tree, người sử dụng chỉ việc khai báo một cách đơn giản, chẳng hạn:**B-tree<int, 5> sample_tree;** sẽ khai báo **sample_tree** là một cây B-tree bậc 5 chứa các bản ghi là các số nguyên.

```
template <class Record, int order>
class B_tree {
public:  //  Các phương thức.

private: //  Thuộc tính:
   B_node<Record, order> *root;
         //  Các hàm phụ trợ.
};
```

Bên trong mỗi nút của B-tree chúng ta cần một danh sách các phần tử và một danh sách các con trỏ đến các nút con. Do cách danh sách này ngắn, để đơn giản, chúng ta dùng các mảng liên tục và một thuộc tính **count** để biểu diễn chúng.

```
template <class Record, int order>
struct B_node {
//  Các thuộc tính:
   int count;
   Record data[order - 1];
   B_node<Record, order> *branch[order];
//  constructor:
   B_node();
};
```

Thuộc tính **count** chứa số bản ghi hiện tại trong từng nút. Nếu **count** khác 0 thì nút có count+1 nút con khác rỗng. Nhánh **branch[0]** chỉ đến cây con chứa các bản ghi có các khóa nhỏ hơn khóa trong **data[0]**; với mỗi trị của **position** nằm giữa 1 và count-1, kể cả hai cận này, **branch[position]** chỉ đến cây con có các khóa nằm giữa hai khóa của **data[position-1]** và **data[position]**; và **branch[count]** chỉ đến cây con có các khóa lớn hơn khóa trong **data[count-1]**.

Constructor của **B_node** tạo một nút rỗng bằng cách gán **count** bằng 0.

10.3.5.2. Tìm kiếm

Như ví dụ đơn giản đầu tiên, chúng ta viết phương thức tìm kiếm trong một cây B-tree cho một bản ghi có khóa trùng với khóa của **target**. Trong phương thức tìm kiếm của chúng ta, như thường lệ, chúng ta sẽ giả thiết rằng các bản ghi này có thể được so sánh bởi các toán tử so sánh chuẩn. Cũng như việc tìm kiếm trong cây nhị phân tìm kiếm, chúng ta bắt đầu bằng cách gọi một hàm đệ quy phụ trợ.

```
template <class Record, int order>
Error_code B_tree<Record, order>::search_tree(Record &target)
/*
post: Nếu tìm thấy phần tử có khóa trùng với khóa trong target thì toàn bộ bản ghi phần tử
      này được chép vào target, phương thức trả về success. Ngược lại, phương thức trả về
      not_present .
uses: Hàm đệ quy phụ trợ recursive_search_tree
*/
{
   return recursive_search_tree(root, target);
}
```

Thông số vào cho hàm đệ quy phụ trợ **recursive_search_tree** là con trỏ đến gốc của cây con trong B-tree và bản ghi **target** chứa khóa cần tìm. Hàm sẽ trả về mã lỗi cho biết việc tìm kiếm kết thúc thành công hay không; nếu tìm thấy, **target** được cập nhật bởi bản ghi chứa khóa được tìm thấy trong cây.

Phương pháp chung để tìm kiếm bằng cách lần theo các con trỏ để đi xuống trong cây tương tự cách tìm kiếm trong cây nhị phân tìm kiếm. Tuy nhiên, trong một cây nhiều nhánh, chúng ta cần tốn nhiều công hơn trong việc xác định ra nhánh cần xuống tiếp theo trong mỗi nút. Việc này sẽ được thực hiện bởi một hàm phụ trợ khác của B-tree là **search_node**, hàm này tìm bản ghi có khóa trùng với khóa của **target** trong số các bản ghi có trong nút được tham chiếu bởi con trỏ **current**. Hàm **search_node** có sử dụng tham biến **position**, nếu tìm thấy, tham biến này sẽ nhận về chỉ số của bản ghi chứa khóa cần tìm trong nút tham chiếu bởi **current**; ngược lại nó chứa chỉ số của nhánh bên dưới tiếp theo cần tìm.

```
template <class Record, int order>
Error_code B_tree<Record, order>::recursive_search_tree
                      (B_node<Record, order> *current, Record &target)
/*
pre:  current là NULL hoặc chỉ đến gốc một cây con trong B_tree.
post: Nếu khóa trong target không tìm thấy, hàm trả về not_present. Ngược lại, target
      được cập nhật bởi bản ghi có chứa khóa tìm dược trong cây, hàm trả về success.
uses: Hàm phụ trợ recursive_search_tree một cách đệ quy và hàm search_node.
*/
{
   Error_code result = not_present;
```

```
   int position;
   if (current != NULL) {
      result = search_node(current, target, position);
      if (result == not_present)
         result=recursive_search_tree(current->branch[position],
                                                    target);
      else
         target = current->data[position];
   }
   return result;
}
```

Hàm trên được viết đệ quy để chứng tỏ sự tương tự giữa cấu trúc của nó với cấu trúc của hàm thêm phần tử trong phần tiếp theo dưới đây. Tuy nhiên, đây là đệ quy đuôi, và nó có thể được thay bởi cấu trúc lặp.

10.3.5.3. Tìm kiếm trong một nút

Hàm **search_node** dưới đây thực hiện việc tìm tuần tự. Hàm này cần xác định xem **target** đã có trong nút hiện tại hay chưa, nếu chưa, nó cần xác định nhánh nào trong số **count+1** nhánh là chứa **target**. Dưới đây là cách tìm tuần tự với biến tạm chạy từ 0 đến vị trí tìm thấy hoặc vừa vượt qua khóa của **target**.

```
template <class Record, int order>
Error_code B_tree<Record, order>::search_node
  (B_node<Record, order> *current, const Record &target, int &position)
/*
pre:  current chứa địa chỉ 1 nút trong B_tree.
post: Nếu khóa trong target được tìm thấy trong *current, thông số position sẽ chứa vị trí
      của phần tử target trong nút này, target được cập nhật lại, hàm trả về success.
      Ngược lại, hàm trả về not_present, position sẽ là chỉ số của nhánh con bên dưới cần
      tiếp tục việc tìm kiếm.
uses: Các phương thức của lớp Record.
*/
{
   position = 0;
   while (position < current->count && target >current->data[position])
      position++;     //  Tìm tuần tự.
   if (position < current->count && target == current->data[position])
      return success;
   else
      return not_present;
}
```

Đối với cây B-tree có các nút khá lớn, hàm trên cần được sửa đổi để sử dụng cách tìm nhị phân thay vì tìm tuần tự. Trong một vài ứng dụng, mỗi bản ghi của cây B-tree chứa rất nhiều dữ liệu, điều này làm cho bậc của cây trở nên tương đối nhỏ, và việc tìm tuần tự trong một nút là thích hợp. Trong nhiều ứng dụng khác, chỉ có các khóa là được chứa trong các nút, nên bậc của cây trở nên khá lớn, chúng ta cần dùng cách tìm nhị phân để tìm vị trí của một khóa trong một nút.

Một khả năng khác cũng có thể được xem xét, đó là việc sử dụng một cây nhị phân tìm kiếm thay cho mảng liên tục các phần tử trong mỗi nút của cây B-tree.

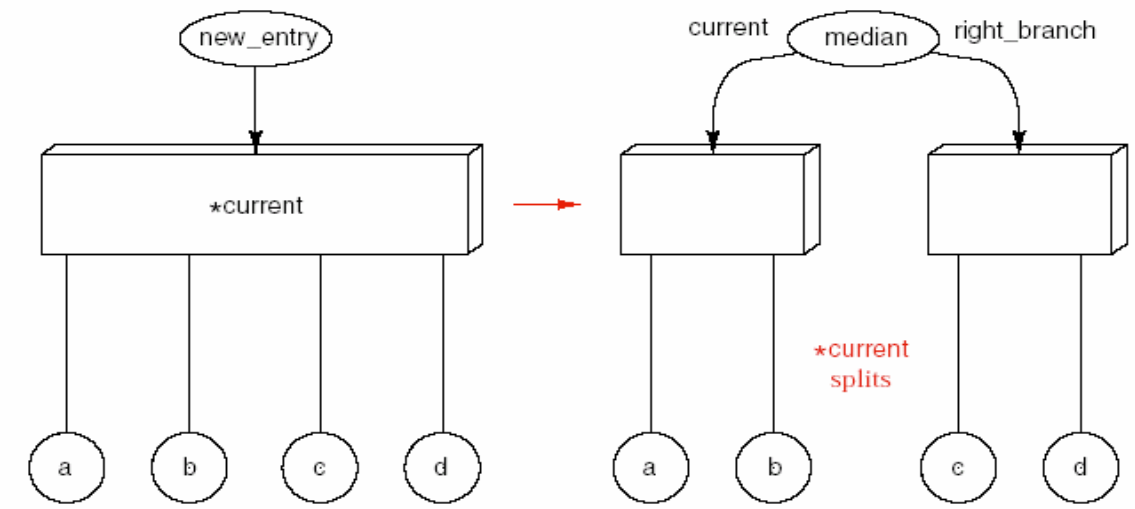
10.3.5.4. Thêm vào: phương thức insert và hàm đệ quy push_down

Việc thêm phần tử vào một cây B-tree có thể được xây dựng một cách tự nhiên như một hàm đệ quy. Đệ quy cho phép chúng ta giữ được vết của đường đi đến một nút trong cây, để khi quay về (khi các lần gọi đệ quy lần lượt kết thúc), chúng ta có thể thực hiện tiếp một số công việc cần thiết ở các nút thuộc mức trên theo thứ tự ngược với khi đi xuống. Nhờ vậy, chúng ta không cần sử dụng ngăn xếp một cách tường minh. Cách làm này hoàn toàn tương tự với cách mà chúng ta đã làm trong việc cân bằng lại khi thêm hoặc loại một nút trong cây cân bằng.

Như thường lệ, chúng ta cần biết chắc là khóa cần thêm chưa có trong cây. Phương thức thêm vào **insert** chỉ cần một thông số **new_entry** chứa bản ghi cần thêm. Tuy nhiên, hàm đệ quy **push_down** của chúng ta cần thêm ba tham biến bổ sung. Thông qua các tham biến này, một nút, sau khi gọi đệ quy xuống nút con của nó, sẽ biết được cần phải giải quyết những việc gì mà nút con của nó đã gởi gắm trở lại. Đó chính là khi một nút ở mức nào đó được phân đôi và quá trình này có thể sẽ phải lan truyền ngược về nút gốc của cây.

Hàm đệ quy **push_down** với thông số **new_entry** được gọi xuống cây con có gốc là current để thêm new_entry vào cây con này**.** Hàm push_down trả về duplicate_error nếu new_entry đã có trong cây; trả về success nếu việc thêm vào thành công và mọi chuyện đã được giải quyết triệt để trong cây con mà nó xử lý. Trong trường hợp có sự thêm new_entry vào cây con mà công việc còn chưa giải quyết triệt để (ngay tại nút *current có sự phân chia làm hai nút), hàm push_down sẽ trả về overflow để báo lên nút cha của cây con này giải quyết tiếp. Lúc đó, các tham biến sẽ có vai trò như sau. Do nút ***current** cần được phân đôi, chúng ta sẽ để **current** chỉ đến nút chứa một nửa số phần tử bên trái, và địa chỉ của nút mới chứa một nửa số phần tử bên phải sẽ được trả lên mức trên thông qua tham biến **right_branch**. Tham biến **median** được sử dụng để chứa bản ghi nằm giữa để trả lên mức trên.

Trường hợp có một nút được phân đôi được minh họa trong hình 10.11.



Hình 10.11- Hành vi của hàm push_down khi một nút được phân đôi.

Quá trình đệ quy được bắt đầu trong phương thức **insert** của B-tree. Trong trường hợp những việc cần giải quyết lan truyền lên đến tận nút gốc và lần gọi đệ quy ngoài cùng của hàm **push_down** trả về overflow, thì vẫn còn một bản ghi, **median**, cần được thêm vào cây. Một nút gốc mới cần được tạo ra để chứa bản ghi này, và chiều cao của cây B-tree tăng thêm 1. Đó là cách duy nhất để B-tree tăng chiều cao.

```
template <class Record, int order>
Error_code B_tree<Record, order>::insert(const Record &new_entry)
/*
post: Nếu khóa trong new_entry đã có trong B-tree, phương thức trả về duplicate_error.
      Ngược lại, new_entry được thêm vào cây sao cho cây vẫn thỏa điều kiện cây B-tree,
      phương thức trả về success.
uses: Các phương thức của B_node và hàm phụ trợ push_down.
*/
{
   Record median;
   B_node<Record, order> *right_branch, *new_root;
   Error_code result =push_down(root, new_entry, median, right_branch);
   if (result == overflow) {  //  Cây tăng chiều cao lên 1 đơn vị.
       //Một nút mới được tạo ra để làm gốc mới cho cây, gốc cũ của cây sẽ là gốc của cây con
         thuộc nhánh con đầu tiên của nút gốc.
      new_root = new B_node<Record, order>;
      new_root->count = 1;
      new_root->data[0] = median;
      new_root->branch[0] = root;
      new_root->branch[1] = right_branch;
      root = new_root;
      result = success;
   }
   return result;
}
```

10.3.5.5. Thêm đệ quy vào một cây con

Chúng ta hãy hiện thực hàm đệ quy **push_down**. Hàm này sử dụng con trỏ **current** tham chiếu đến gốc của cây con cần thực hiện việc tìm kiếm để thêm vào. Trong cây B-tree, bản ghi mới trước hết cần được thêm vào một nút lá. Chúng ta sẽ sử dụng điều kiện **current** == NULL để kết thúc đệ quy; nghĩa là, chúng ta sẽ tiếp tục di chuyển xuống theo cây trong khi tìm kiếm **new_entry** cho đến khi gặp phải một cây con rỗng. Do cây B-tree không lớn lên bằng cách thêm nút lá mới, chúng ta không thêm **new_entry** ngay lập tức, mà thay vào đó hàm sẽ trả về overflow, **new_entry** được gởi trả về thông qua tham biến **median** và sẽ được thêm vào một nút lá đã có ở mức trên. Việc cần làm tiếp theo cũng hoàn toàn giống với trường hợp tổng quát tại bất cứ nút nào trong cây mà chúng ta sẽ xem xét tiếp sau đây.

Khi một lần đệ quy trả về **overflow**, cũng có nghĩa là còn một bản ghi **median** vẫn chưa được thêm vào cây, và chúng ta sẽ thử thêm nó vào nút hiện tại. Nếu nút này còn chỗ trống, việc thêm sẽ hoàn tất, hàm trả về **success**. Điều này cũng làm cho các lần đệ quy trước đó sẽ lần lượt kết thúc mà không phải làm gì thêm. Ngược lại, nút ***current** được phân thành hai nút ***current** và ***right_branch,** và một bản ghi nằm giữa, **median** (có thể khác với bản ghi **median** từ lần đệ quy bên dưới trả về), được gởi ngược lên phía trên của cây, thông số trả về vẫn được giữ nguyên là **overflow**.

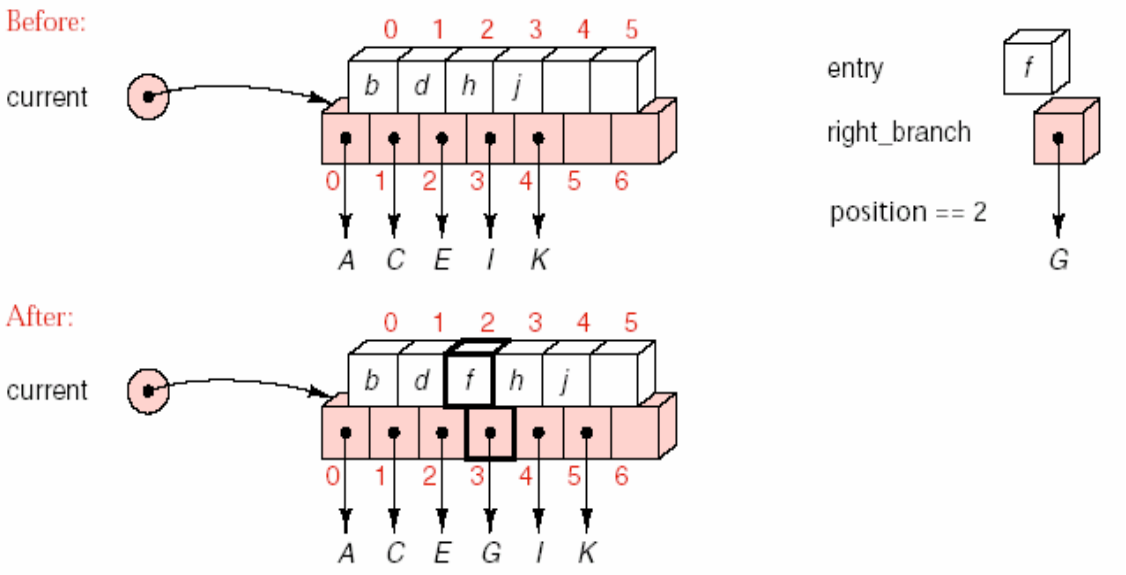
Push_down sử dụng ba hàm phụ trợ: **search_node** (giống như trong trường hợp tìm kiếm); **push_in** thêm bản ghi **median** vào nút ***current** với giả thiết rằng nút này còn chỗ trống; và **split** để chia đôi nút ***current** đã đầy thành hai nút mới, hai nút này sẽ là anh em trong cùng một mức trong cây B-tree.

```
template <class Record, int order>
Error_code B_tree<Record, order>::push_down
                                  (B_node<Record, order> *current,
                                   const Record &new_entry,
                                   Record &median,
                                   B_node<Record, order> *&right_branch)
/*
pre:  current là NULL hoặc chỉ đến một nút trong cây B_tree.
post: Nếu khóa trong new_entry đã có trong cây con có gốc current, hàm trả về
      duplicate_error. Ngược lại new_entry được chèn vào cây con, nếu diều này làm cho
      cây con cao lên, hàm trả về overflow và bản ghi median được tách ra để được chèn ở
      mức cao hơn trong cây B-tree, đồng thời right_branch chứa gốc của cây con bên phải
      bản ghi median này. Nếu cây con không cần cao lên thì hàm trả về success.
uses: Hàm push_down (một cách đệ quy), search_node, split_node, and push_in.
*/
{
   Error_code result;
   int position;
```

```
   if (current == NULL) { //  Do không thể chèn vào một cây con rỗng nên đệ quy // kết
                        thúc, việc cần làm sẽ được giải quyết ở mức trên sau đó.
      median = new_entry;
      right_branch = NULL;
      result = overflow;
   }
   else {   //   Search the current node.
      if (search_node(current, new_entry, position) == success)
         result = duplicate_error;
      else {
         Record extra_entry;
         B_node<Record, order> *extra_branch;
         result = push_down(current->branch[position], new_entry,
                            extra_entry, extra_branch);
         if (result == overflow) { // Cần giải quyết công việc nút con gởi lên.
         if (current->count < order - 1) {
               result = success;
               push_in(current, extra_entry, extra_branch, position);
            }

            else split_node(current, extra_entry, extra_branch,
                                 position, right_branch, median);
            //Bản ghi median và right_branch được cập nhật trong chính hàm này
         }
      }
   }
   return result;
}
```

10.3.5.6. Thêm một khóa vào một nút

Hàm phụ trợ kế tiếp, **push_in**, thêm bản ghi **entry** và con trỏ bên phải của nó là **right_branch** vào nút ***current**, giả sử rằng nút này còn chỗ trống để thêm vào. Hình 10.12 minh họa trường hợp này.



Hình 10.12- Hành vi của hàm `push in`.

```
template <class Record, int order>
void B_tree<Record, order>::push_in(B_node<Record, order> *current,
                                    const Record &entry, B_node<Record,
                                    order> *right_branch, int position)
/*
pre:  current chứa địa chỉ một nút trong B_tree. Nút *current chưa đầy và entry cần
      được chèn vào *current tại vị trí position, right_branch cần được cập nhật chính
      là cây con bên phải của entry trong *current.
post: entry và right_branch đã được chèn vào *current tại vị trí position.
*/
{
   for (int i = current->count; i > position; i--) {
                  // Di chuyển các phần tử cần thiết sang phải để nhường chỗ.
      current->data[i] = current->data[i - 1];
      current->branch[i + 1] = current->branch[i];
   }
   current->data[position] = entry;
   current->branch[position + 1] = right_branch;
   current->count++;
}
```

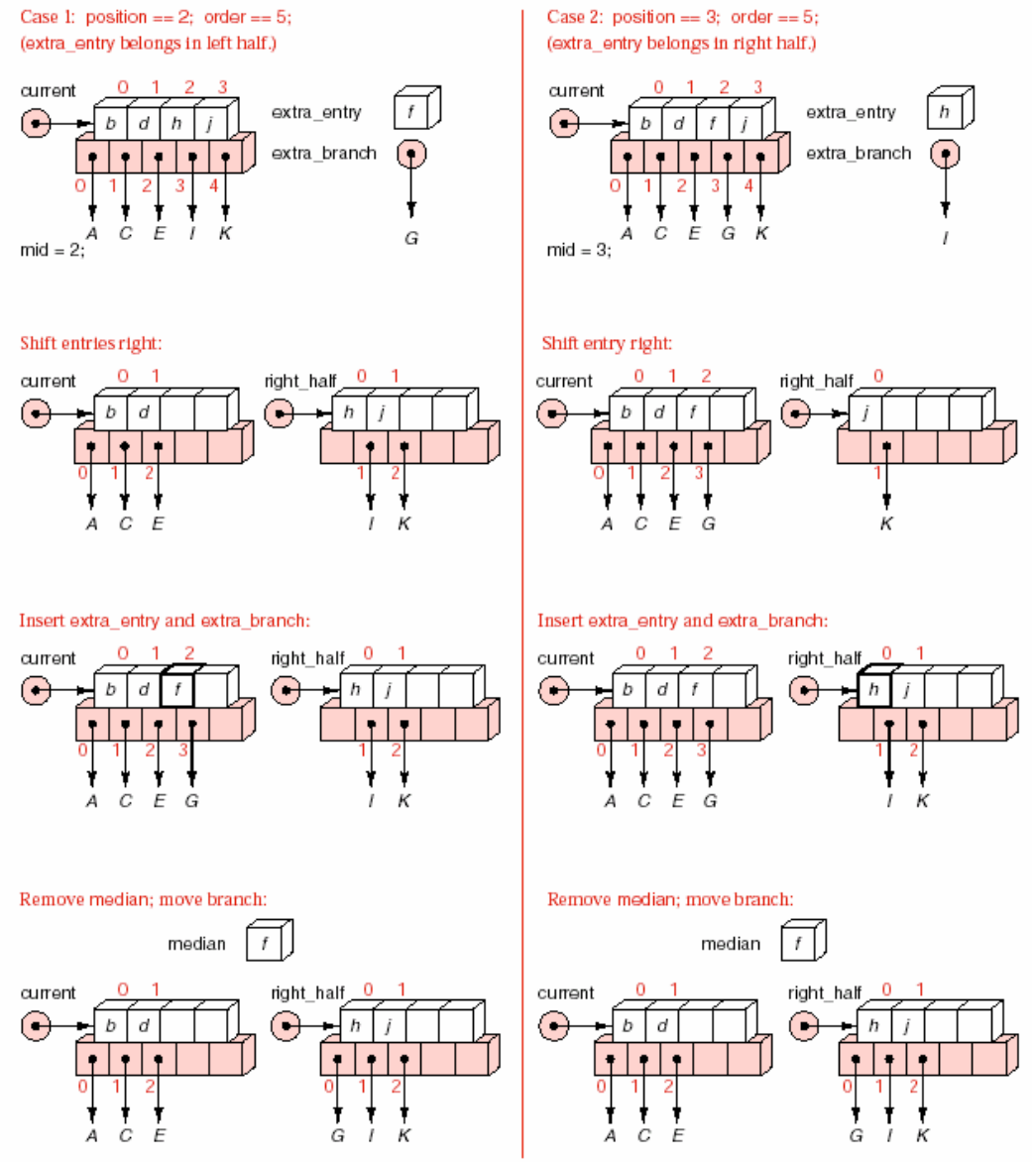
10.3.5.7. Phân đôi một nút đang đầy

Hàm phụ trợ cuối cùng cho phương thức thêm vào, **split_node**, được sử dụng khi cần thêm bản ghi **extra_entry** cùng con trỏ chỉ đến cây con **extra_branch** vào nút đã đầy ***current**. Hàm này tạo nút mới tham chiếu bởi **right_half** và chuyển một nửa số bản ghi bên phải của nút ***current** sang, gởi bản ghi nằm giữa lên phía trên của cây để nó có thể được thêm vào sau đó.

Dĩ nhiên là không thể thêm bản ghi **extra_entry** thẳng vào nút đã đầy: trước hết chúng ta cần xác định xem **extra_entry** sẽ thuộc nửa bên trái hay nửa bên phải số bản ghi sẵn có trong nút ***current**, sau đó di chuyển các bản ghi thích hợp, và cuối cùng sẽ thêm **extra_entry** vào bên tương ứng. Chúng ta sẽ chia đôi số phần tử trong nút ***current** sao cho bản ghi **median** là phần tử có khóa lớn nhất trong nửa số phần tử bên trái. Hình 10.13 minh họa điều này.

```
template <class Record, int order>
void B_tree<Record,order>::split_node
  (B_node<Record,order> *current,     // Nút cần được phân đôi.
   const Record &extra_entry,         // Phần tử mới cần chèn vào.
  B_node<Record,order>*extra_branch, // Cây con bên phải của extra_entry.
   int position,// Vị trí của extra_entry trong *current so với các phần tử đã có.
  B_node<Record,order>*&right_half,
                //Nút mới để chứa một nửa số phần tử từ *current.
   Record &median)//Phần tử giữa không nằm trong cả hai *current hoặc
  // *right_half mà sẽ được chuyển lên phía trên trong cây B_tree.
/*
pre:  current chứa địa chỉ một nút trong cây B_tree.
      Nút *current đã đầy, nhưng phần tử extra_entry cùng cây con bên phải của nó
      extra_branch cần được chèn vào vị rí position, 0 <=position <order.
```

```
post: Các phần tử đã có trong nút *current cùng với extra_entry và extra_branch (xem
      như đã được xếp vào đúng vị trí position) được phân phối vào nút *current và nút mới
      *right_half, ngoại trừ phần tử chính giữa trong số các phần tử này được đưa vào
      median.
uses: các phương thức của B_node, hàm push_in.
*/
{  right_half = new B_node<Record, order>;
   int mid = order/2;  //
   if (position <= mid){ // Trường hợp 1: extra_entry thuộc nửa bên trái.
      for (int i = mid; i < order - 1; i++){ // Chuyển các phần tử từ
             // *current sang *right_half trước rồi mới gọi hàm push_in để
             // chèn extra_entry và extra_branch vào *current sau.
         right_half->data[i - mid] = current->data[i];
         right_half->branch[i + 1 - mid] = current->branch[i + 1];
      }
      current->count = mid;
      right_half->count = order - 1 - mid;
      push_in(current, extra_entry, extra_branch, position);
   }
   else {          //  Trường hợp 2:  extra_entry thuộc nửa bên phải.
      mid++;       //  Tạm thời vẫn để phần tử cần chép vào median ở lại trong nửa bên
                                                                    trái.
      for (int i = mid; i < order - 1; i++) {// Chuyển các phần tử từ
             // *current sang *right_half trước rồi mới gọi hàm push_in để
             // chèn extra_entry và extra_branch vào *right_half sau.
         right_half->data[i - mid] = current->data[i];
         right_half->branch[i + 1 - mid] = current->branch[i + 1];
      }
      current->count = mid;
      right_half->count = order - 1 - mid;
      push_in(right_half, extra_entry, extra_branch, position - mid);
   }
      median = current->data[current->count -1];//Chép phần tử vào median
      right_half->branch[0] = current->branch[current->count];
      current->count--;
}
```



Hình 10.13 – Hành vi của hàm `split`.

10.3.6. Loại phần tử trong B-tree

10.3.6.1. Phương pháp

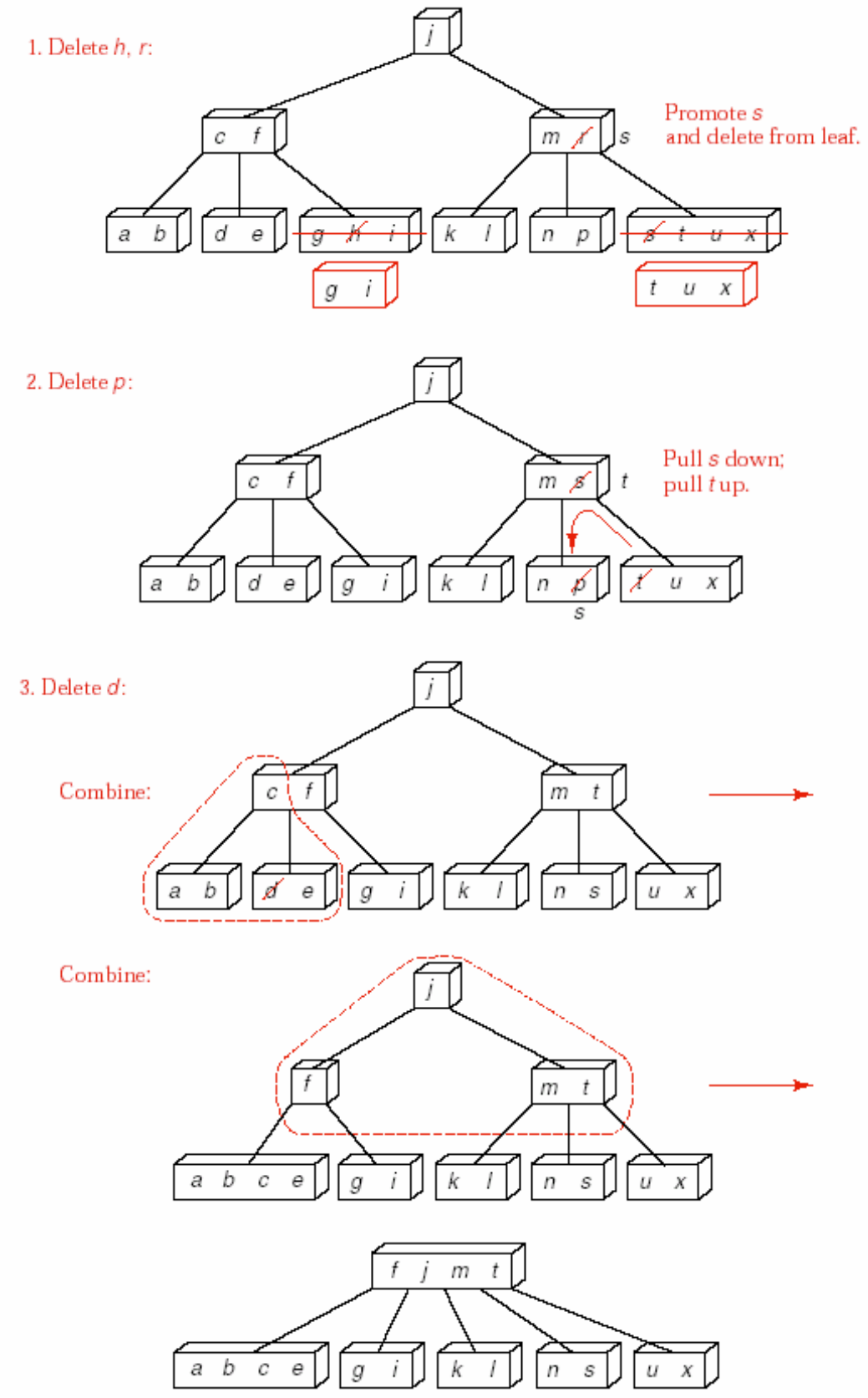
Đối với việc loại bỏ phần tử, chúng ta mong muốn rằng phần tử được loại bỏ thuộc một nút lá nào đó. Nếu phần tử này không thuộc nút lá, thì phần tử ngay kế trước nó (hoặc ngay kế sau nó) theo thứ tự tự nhiên của các khóa sẽ thuộc nút lá. Chúng ta sẽ đặt phần tử kế trước này (hoặc kế sau) thế vào chỗ của phần tử cần loại, sau đó loại vị trí của nó ra khỏi nút lá. Cách làm này rất giống với cách làm trong cây nhị phân tìm kiếm.

Nếu nút lá cần loại đi một phần tử có nhiều hơn số phần tử tối thiểu thì việc loại kết thúc. Ngược lại, nếu nút lá đang chứa số phần tử bằng số phần tử tối thiểu, thì trước hết chúng ta sẽ xem xét hai nút lá kế cận với nó và cùng một cha (hoặc chỉ một nút lá kế cận trong trường hợp nút lá đang xét nằm ở biên), nếu một trong hai có nhiều hơn số phần tử tối thiểu thì một phần tử trong số đó có thể di chuyển lên nút cha và phần tử trong nút cha sẽ di chuyển xuống nút lá đang thiếu phần tử (Chúng ta biết rằng cần phải di chuyển như vậy để **bảo đảm thứ tự giữa các phần tử**). Cuối cùng, nếu cả hai nút lá kế cận chỉ có số phần tử tối thiểu, thì nút lá đang thiếu cần kết hợp với một trong hai nút lá kế cận, **có lấy thêm một phần tử từ nút cha**, thành một nút lá mới (**Do số nút con giảm nên số phần tử trong nút cha cũng phải giảm**). Nút này sẽ chứa số phần tử không nhiều hơn số phần tử tối đa được phép. Nếu bước này làm cho nút cha còn lại số phần tử ít hơn số phần tử tối thiểu, thì việc giải quyết cũng tương tự, và quá trình này sẽ lan truyền ngược lên phía trên của cây. Quá trình lan truyền sẽ chấm dứt khi một nút cha nào đó khi cho đi một phần tử vẫn không trở nên thiếu hụt phần tử. Trong trường hợp đặc biệt, khi phần tử cuối cùng trong nút gốc bị lấy đi thì nút này cũng được giải phóng và cây sẽ giảm chiều cao.

10.3.6.2. Ví dụ

Quá trình loại bỏ trong cây `B-tree` bậc 5 sẵn có của chúng ta được minh họa trong hình 10.14. Lần loại thứ nhất không có vấn đề gì do **h** nằm trong nút lá đang có nhiều hơn số phần tử tối thiểu. Lần thứ hai, loại **r**, do **r** không thuộc nút lá, nên phần tử ngay kế sau **r** là **s** được chép đè lên **r**, và **s** được loại khỏi nút lá. Lần thứ ba, việc loại **p** làm cho nút chứa nó còn quá ít phần tử. Khóa **s** từ nút cha được chuyển xuống lấp đi sự thiếu hụt và vị trí của **s** được thế bởi **t**.

Việc loại **d** tiếp theo phức tạp hơn, nó làm cho nút còn lại quá ít phần tử, và cả hai nút kế cùng cha đều không thể sang bớt phần tử cho nó. Nút đang thiếu hụt này phải kết hợp với một trong hai nút kế, khi đó một phần tử nằm giữa chúng từ nút cha được đưa xuống (biểu diễn bởi nét rời trong sơ đồ thứ nhất của trường hợp này). Nút kết hợp được gồm các phần tử **a, b, c, e** theo sơ đồ thứ hai. Tuy nhiên, quá trình này làm cho nút cha chỉ còn lại một phần tử **f.** Ba nút phía trên của cây phải được kết hợp lại và cuối cùng chúng ta có cây như sơ đồ cuối của hình vẽ.



Hình 10.14 – Loại phần tử ra khỏi B-tree.

10.3.6.3. Hiện thực C++

Chúng ta có thể viết giải thuật loại phần tử với cấu trúc tổng thể tương tự như giải thuật thêm vào. Chúng ta sẽ sử dụng đệ quy, với một phương thức riêng để khởi động quá trình đệ quy. Thay cho việc đẩy một phần tử từ nút cha xuống trong khi gọi đệ quy xuống bên dưới, chúng ta sẽ cho hàm đệ quy trả về một nút đang thiếu phần tử thông qua tham biến. Lần gọi đệ quy bên trên sẽ phân tích điều gì đã xảy ra và thực hiện việc di chuyển các phần tử cần thiết. Khi phần tử cuối cùng trong nút gốc bị lấy đi, nút rỗng sẽ được giải phóng và chiều cao của cây giảm bớt 1.

```
template <class Record, int order>
Error_code B_tree<Record, order>::remove(const Record &target)
/*
post: Nếu khóa trong target được tìm thấy trong cây, phần tử chứa khóa này bị loại khỏi cây,
      phương thức trả về success. Ngược lại, phương thức trả về not_present.
uses: Hàm recursive_remove.
*/
{
   Error_code result;
   result = recursive_remove(root, target);
   if (root != NULL && root->count == 0) {  //  Cây giảm chiều cao.
      B_node<Record, order> *old_root = root;
      root = root->branch[0];
      delete old_root;
   }
   return result;
}
```

10.3.6.4. Loại đệ quy

Phần lớn công việc được thực hiện trong hàm đệ quy **recursive_remove**. Trước tiên nó tìm nút chứa **target**. Nếu **target** được tìm thấy và nút chứa nó không là nút lá, thì phần tử kế sau **target** sẽ được tìm và được chép đè lên nó. Phần tử kế sau này sẽ được loại bỏ. Việc loại phần tử trong nút lá được thực hiện một cách dễ dàng, và những phần công việc còn lại sẽ được tiếp tục nhờ đệ quy. Khi một lần gọi đệ quy được trả về, hàm sẽ kiểm tra xem nút tương ứng có còn đủ số phần tử hay không, nếu thiếu, nó di chuyển các phần tử để đáp ứng yêu cầu. Các hàm phụ trợ sẽ được sử dụng trong một số bước này.

```
template <class Record, int order>
Error_code B_tree<Record, order>::recursive_remove
                (B_node<Record, order> *current, const Record &target)
/*
pre:  current là NULL hoặc chứa địa chỉ nút gốc của một cây con trong B_tree.
post: Nếu khóa trong target được tìm thấy trong cây, phần tử chứa khóa này bị loại khỏi cây
      sao cho cây vẫn giữ tính chất cây B-tree, phương thức trả về success. Ngược lại,
      phương thức trả về not_present.
uses: Các hàm search_node, copy_in_predecessor,
      recursive_remove (một cách đệ quy), remove_data, và restore.
*/
```

```
{
   Error_code result;
   int position;
   if (current == NULL) result = not_present;
   else {
      if (search_node(current, target, position) == success){
                          // Khóa trong target được tìm thấy trong current.
         result = success;
         if (current->branch[position] !=NULL){//current không phải nút lá
            copy_in_predecessor(current, position);

            recursive_remove(current->branch[position],
                             current->data[position]);
         }
         else remove_data(current, position);// Loại phần tử trong nút lá.
      }
      else result =recursive_remove(current->branch[position], target);
      if (current->branch[position] != NULL)
         if (current->branch[position]->count < (order - 1) / 2)
            restore(current, position);
   }
   return result;
}
```

10.3.6.5. Các hàm phụ trợ

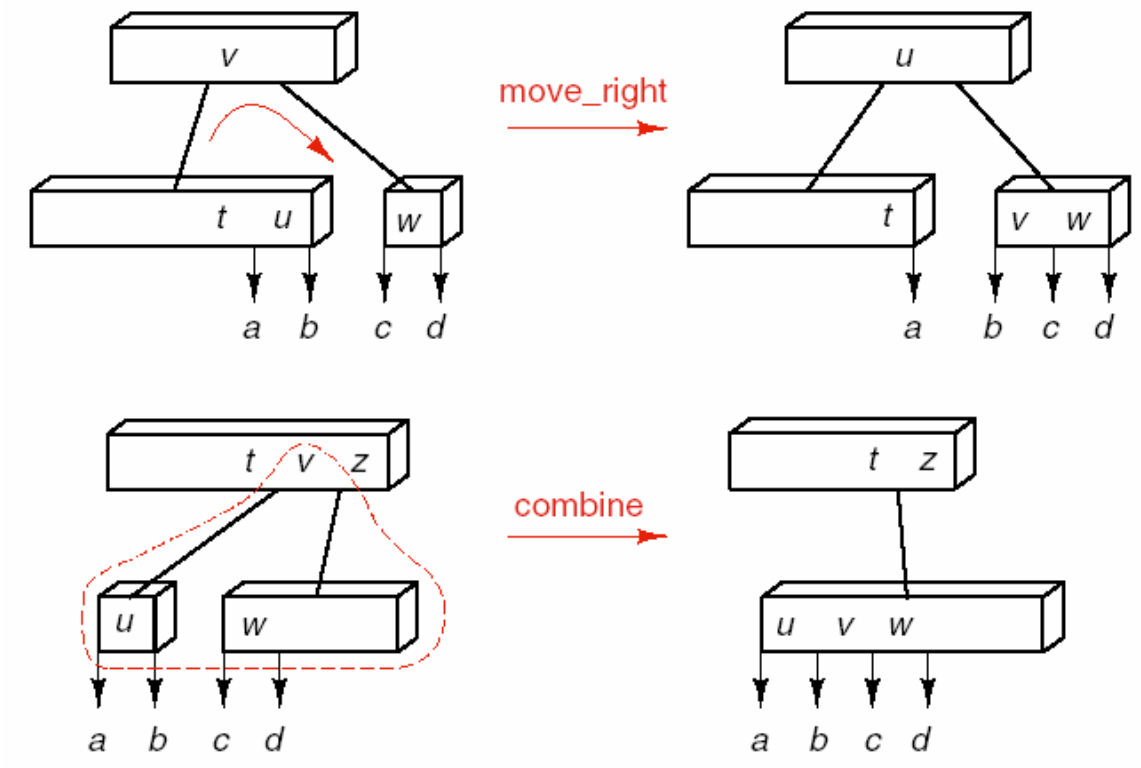
Giờ chúng ta đã có thể kết thúc quá trình loại bỏ trong một cây *B-tree* bằng cách viết các hàm phụ trợ cho nó. Hàm **remove_data** loại một phần tử và nhánh bên phải của nó khỏi một nút trong cây *B-tree*. Hàm này được gọi chỉ trong trường hợp khi một phần tử được loại khỏi một nút lá của cây.

```
template <class Record, int order>
void B_tree<Record, order>::remove_data(B_node<Record, order> *current,
                                        int position)
/*
pre:  current là địa chỉ nút lá có entry tại vị trí position.
post: entry được loại khỏi nút *current.
*/
{
   for (int i = position; i < current->count - 1; i++)
      current->data[i] = current->data[i + 1];
   current->count--;
}
```

Hàm **copy_in_predecessor** được gọi khi một phần tử cần được loại ra khỏi một nút không phải nút lá. Trong trường hợp này, phần tử ngay kế trước (theo thứ tự các khóa) sẽ được tìm bằng cách bắt đầu từ nhánh bên trái của nó và đi xuống theo các nhánh tận cùng bên phải của mỗi nút cho đến khi gặp nút lá. Phần tử tận cùng bên phải của nút lá này sẽ thay thế phần tử cần được loại.

```
template <class Record, int order>
void B_tree<Record, order>::copy_in_predecessor
                        (B_node<Record, order> *current, int position)
/*
pre:  current là địa chỉ nút không phải nút lá trong B-tree và có entry là phần tử cần loại
      tại position.
post: entry được thay thế bởi phần tử đứng ngay kế trước nó trong thứ tự tăng dần của khóa.
*/
{
   B_node<Record, order> *leaf = current->branch[position]; // Di chuyển qua
                        nhánh trái để tìm phần tử ngay kế trước entry.
   while (leaf->branch[leaf->count] != NULL)
      leaf = leaf->branch[leaf->count]; // Xuống phần tử cực phải của nhánh trái của
                                                              current.
   current->data[position] = leaf->data[leaf->count - 1];
}
```

Cuối cùng, chúng ta cần chỉ ra cách khôi phục lại số nút tối thiểu cho nút được tham chiếu bởi **`root->branch[position]`** nếu như lần đệ quy bên trong làm cho nó trở nên thiếu phần tử. Hàm chúng ta viết dưới đây hơi thiên về bên trái, nghĩa là, trước hết nó tìm nút anh em kề bên trái để xin bớt phần tử, và nó chỉ xét đến nút anh em kề bên phải khi nút kề bên trái không thừa phần tử. Các bước thực hiện được minh họa trong hình 10.15.



Hình 10.15 – Khôi phục lại số phần tử tối thiểu trong một nút.

```
template <class Record, int order>
void B_tree<Record, order>::restore(B_node<Record, order> *current,
                                    int position)
/*
pre:  current là địa chỉ một nút không phải nút lá trong B-tree;
      current->branch[position] là địa chỉ nút đang bị thiếu một phần tử.
post: Nút do current->branch[position] chỉ đến được khôi phục lại cho đủ số phần tử tối
      thiểu cần có.
uses: Các hàm move_left, move_right, combine.
*/
{  if (position == current->count)// Trường hợp không có nút anh em bên phải.
      if (current->branch[position - 1]->count > (order - 1) / 2)
         move_right(current, position - 1);
      else
         combine(current, position);
   else if (position == 0)        // Trường hợp không có nút anh em bên trái.
      if (current->branch[1]->count > (order - 1) / 2)
         move_left(current, 1);
      else
         combine(current, 1);
   else                           // Trường hợp có nút anh em cả hai bên.
      if (current->branch[position - 1]->count > (order - 1) / 2)
         move_right(current, position - 1);
      else if (current->branch[position + 1]->count > (order - 1) / 2)
         move_left(current, position + 1);
      else
         combine(current, position);
}
```

Các hành vi của ba hàm còn lại **move_left**, **move_right**, và **combine** được thể hiện rất rõ ràng trong hình 10.15.

```
template <class Record, int order>
void B_tree<Record, order>::move_left(B_node<Record, order> *current,
                                      int position)
/*
pre:  current là địa chỉ một nút trong B-tree, nhánh con tại position có nút gốc có số
      phần tử nhiều hơn số phần tử tối thiểu theo quy định, nhánh con tại position–1 có nút
      gốc đang thiếu 1 phần tử.
post: Một phần tử của *current di chuyển xuống nút gốc của nhánh con tại
      position–1, phần tử tại vị trí 0 trong nút gốc của nhánh tại position di chuyển lên
      *current (cây con tại vị trí 0 cũng như các phần tử còn lại trong nút gốc của nhánh này
      sẽ được dịch chuyển hợp lý).
*/
{  B_node<Record, order> *left_branch = current->branch[position - 1],
                        *right_branch = current->branch[position];
   left_branch->data[left_branch->count] = current->data[position - 1];
                           // Lấy một entry từ nút cha *current.
   left_branch->branch[++left_branch->count] = right_branch->branch[0];
   // Giải quyết cho cây con tại vị trí 0 trong nhánh con bên phải: số phần tử trong nhánh con
   // bên trái tăng thêm 1 nên số cây con cũng phải tăng thêm 1, đồng thời cách di chuyển này
   // vẫn bảo đảm thứ tự các khóa trong cây.
   current->data[position - 1] = right_branch->data[0];
                    // Nút cha *current lấy một entry từ nhánh con bên phải.
```

```
    right_branch->count--;
   for (int i = 0; i < right_branch->count; i++) {
                    // Dịch chuyển tất cả các phần tử về bên trái để lấp chỗ trống.
      right_branch->data[i]   = right_branch->data[i + 1];
      right_branch->branch[i] = right_branch->branch[i + 1];
   }
   right_branch->branch[right_branch->count] =
                              right_branch->branch[right_branch->count + 1];
}


template <class Record, int order>
void B_tree<Record, order>::move_right(B_node<Record, order> *current,
                                       int position)
/*
pre:  current là địa chỉ một nút trong B-tree, nhánh con tại position có nút gốc có số
      phần tử nhiều hơn số phần tử tối thiểu theo quy định, nhánh con tại position+1 có nút
      gốc đang thiếu 1 phần tử.
post: Một phần tử của *current di chuyển xuống nút gốc của nhánh con tại
      position+1. Phần tử có khóa lớn nhất (tại count-1) trong nút gốc của nhánh tại
      position di chuyển lên *current (cây con bên phải của nó được bố trí lại hợp lý).
*/
{
   B_node<Record, order> *right_branch = current->branch[position + 1],
                        *left_branch = current->branch[position];
   right_branch->branch[right_branch->count + 1] =
      right_branch->branch[right_branch->count];
   for (int i = right_branch->count ; i > 0; i--) {
      // Di chuyển sang phải để dành chỗ trống cho phần tử từ *current đưa xuống.
      right_branch->data[i] = right_branch->data[i - 1];
      right_branch->branch[i] = right_branch->branch[i - 1];
   }
   right_branch->count++;
   right_branch->data[0] = current->data[position];
                                  //  Nhận entry từ nút cha *current.
   right_branch->branch[0] = left_branch->branch[left_branch->count];
      // Bố trí lại cây con thừa ở nhánh trái do số phần tử giảm bớt 1 (thứ tự các khóa trong
                                                          B-tree vẫn bảo đảm).
   left_branch->count--;
   current->data[position] = left_branch->data[left_branch->count];
}
```

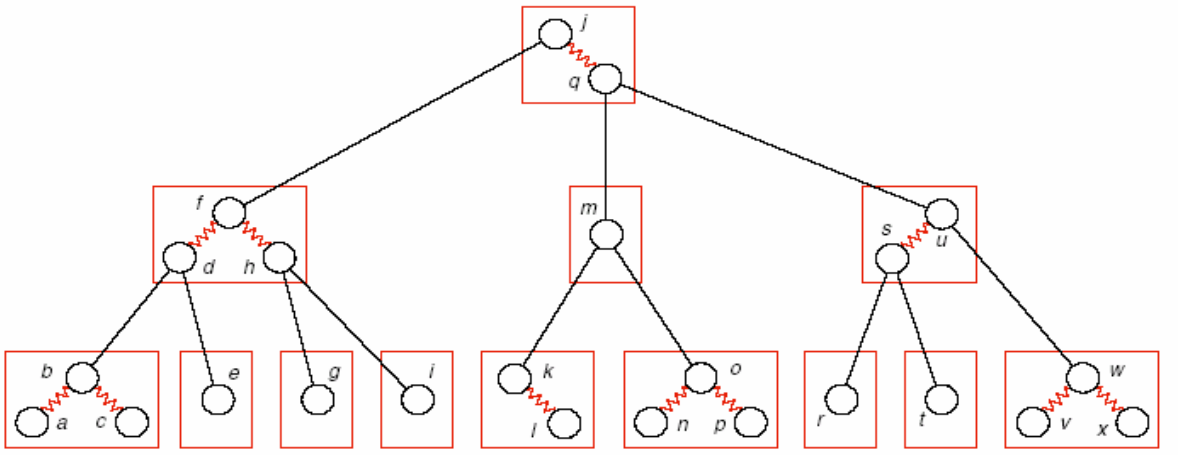
```
template <class Record, int order>
void B_tree<Record, order>::combine(B_node<Record, order> *current,
                                    int position)
/*
pre:  current chứa địa chỉ một nút trong B-tree có các nút gốc của hai nhánh con tại
      position và position – 1 cần ghép (lại do không đủ số phần tử để di chuyển qua lại
      sao cho cả 2 nút vẫn đủ số phần tử tối thiểu).
post: Hai nút gốc của hai nhánh tại position-1 và position được ghép lại (phần tử tại vị trí
      position-1 của *current cũng di chuyển xuống nút này để bảo đảm rằng số phần tử
      trong *current giảm bớt 1 khi số nhánh con giảm bớt 1).
*/


{
```

```
   int i;
   B_node<Record, order> *left_branch = current->branch[position - 1],
                        *right_branch = current->branch[position];
   left_branch->data[left_branch->count] = current->data[position - 1];
   left_branch->branch[++left_branch->count] = right_branch->branch[0];
   for (i = 0; i < right_branch->count; i++) {
      left_branch->data[left_branch->count] = right_branch->data[i];
      left_branch->branch[++left_branch->count] =
                                          right_branch->branch[i + 1];
   }
   current->count--;
   for (i = position - 1; i < current->count; i++) {
      current->data[i] = current->data[i + 1];
      current->branch[i + 1] = current->branch[i + 2];
   }
   delete right_branch;
}
```

10.4. Cây đỏ-đen

10.4.1. Dẫn nhập

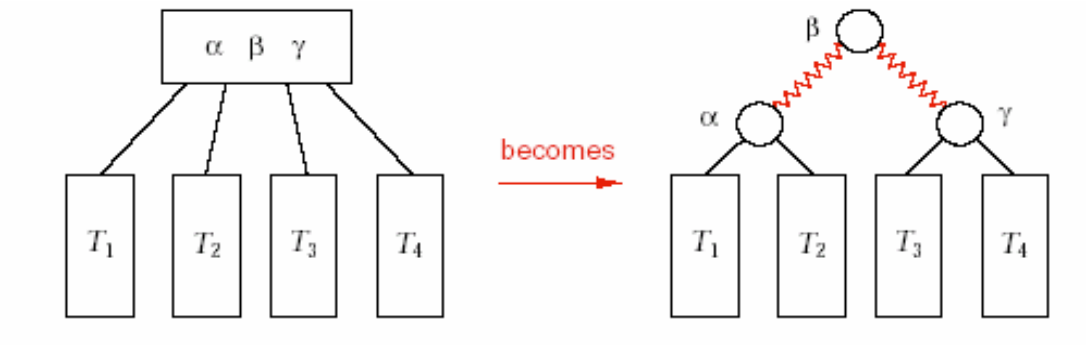
Trong phần trước, chúng ta đã sử dụng danh sách liên tục để chứa các phần tử của cây `B-tree`. Tuy nhiên, nói một các tổng quát, chúng ta có thể dùng bất kỳ cấu trúc có thứ tự nào để chứa các phần tử trong mỗi nút của `B-tree`. Một cây nhị phân tìm kiếm nhỏ là một lựa chọn tốt. Chúng ta chỉ cần chú ý phân biệt các con trỏ bên trong mỗi nút của cây `B-tree` (nối các nút của cây nhị phân tìm kiếm) với các con trỏ từ nút này đến nút khác của `B-tree`. Chúng ta hãy vẽ các tham chiếu bên trong một nút bằng các **đường xoắn màu đỏ** và những con trỏ giữa các nút trong cây `B-tree` bằng các **đường thẳng màu đen**. Xem hình 10.16.



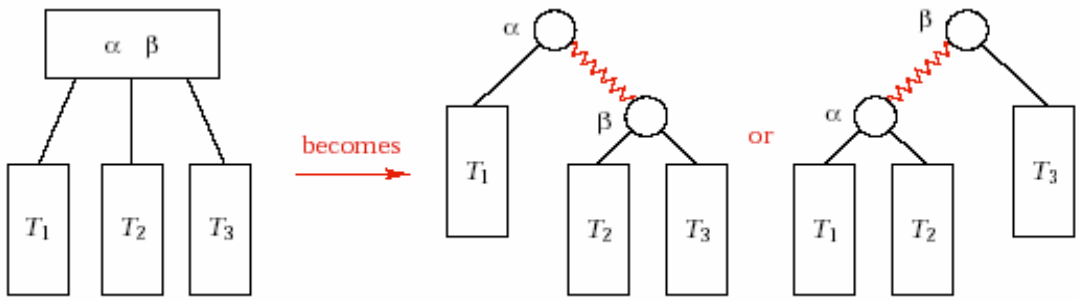
Hình 10.16 – Cây `B-tree` bậc 4 như một cây tìm kiếm nhị phân.

10.4.2. Định nghĩa và phân tích

Cấu trúc này đặc biệt có ích đối với cây `B-tree` bậc 4 (hình 10.16), trong đó mỗi nút của cây chứa một, hai hoặc ba phần tử. Trường hợp một nút có một phần tử thì tương tự như trong cây `B-tree` và cây nhị phân tìm kiếm. Trường hợp một nút có ba phần tử được biến đổi như sau:



Một nút có hai phần tử có thể có hai biểu diễn:



Nếu muốn, chúng ta chỉ cần sử dụng một trong hai cách biểu diễn trên, nhưng không có lý do gì để làm điều đó, chúng ta sẽ thấy rằng các giải thuật của chúng ta sẽ sinh ra cả hai cách biểu diễn này một cách tự nhiên. Như vậy chúng ta sẽ sử dụng cả hai cách biểu diễn cho các nút có hai phần tử trong cây `B-tree.`

Chúng ta có định nghĩa cơ bản cho phần này như sau: Một cây đỏ-đen (*red-black tree*) là một cây nhị phân tìm kiếm, với các tham chiếu có màu đỏ hoặc đen, có được từ một cây `B-tree` bậc bốn bằng cách vừa được mô tả trên.

Sau khi chuyển đổi một cây `B-tree` thành cây đỏ đen, chúng ta có thể sử dụng nó tương tự bất kỳ cây nhị phân tìm kiếm nào. Việc **tìm kiếm** và **duyệt** cây đỏ đen hoàn toàn giống như đối với cây nhị phân tìm kiếm; chúng ta chỉ đơn giản bỏ qua các màu của các tham chiếu. Tuy nhiên, việc **thêm và loại phần tử** đòi hỏi

nhiều công sức hơn để duy trì cấu trúc của một cây `B-tree`. Chúng ta hãy chuyển đổi các yêu cầu đối với cây `B-tree` thành các yêu cầu tương ứng đối với cây đỏ đen.

Trước hết, chúng ta hãy lưu ý một số điểm: chúng ta sẽ xem mỗi nút trong cây đỏ đen cũng **có màu như màu của tham chiếu đến nó**, như vậy chúng ta sẽ gọi các nút màu đỏ và các nút màu đen thay vì gọi các tham chiếu đỏ và các tham chiếu đen. Bằng cách này, chúng ta chỉ cần giữ thêm một thông tin cho mỗi nút để chỉ ra màu của nó.

Do nút gốc không có tham chiếu đến nó, **nó sẽ không có màu**. Để làm đơn giản một số giải thuật, **chúng ta quy ước rằng nút gốc có màu đen**. Tương tự, chúng ta sẽ xem tất cả **các cây rỗng** (tương ứng tham chiếu `NULL`) **có màu đen**.

Theo điều kiện thứ nhất trong định nghĩa của cây `B-tree`, mọi cây con rỗng phải thuộc cùng mức, nghĩa là mọi đường đi từ gốc đến mỗi cây con rỗng sẽ đi qua cùng một số **nút `B-tree`** như nhau. Mỗi **nút `B-tree`** trong cây đỏ đen luôn có một nút đen, do các con trỏ giữa các nút trong cây `B-tree` được biểu diễn bằng các đường thẳng màu đen. Đối với **nút `B-tree` có nhiều hơn một nút thì ngoài một nút đen, các nút còn lại phải có màu đỏ**. Từ đó chúng ta có "điều kiện đen" như sau:

> *Mọi đường đi từ gốc đến mỗi cây con rỗng đều đi qua cùng một số nút đen như nhau.*

Do cây `B-tree` thoả các đặc tính của cây tìm kiếm, nên cây đỏ đen cũng thỏa đặc tính này. Những phần còn lại trong định nghĩa đối với cây `B-tree` bậc 4 nói lên rằng mỗi nút chứa một, hai hoặc ba phần tử dữ liệu. Chúng ta cần một điều kiện trên cây đỏ đen để bảo đảm rằng khi các nút trong cây này được gom lại thành các **nút `B-tree`** thì mỗi **nút `B-tree`** có không quá ba nút. Nếu một **nút `B-tree`** có hai **nút của cây đỏ đen**, thì trong đó sẽ có một nút cha và một nút con, nếu một **nút `B-tree`** có ba **nút của cây đỏ đen**, thì trong đó sẽ có một nút cha và hai nút con (theo hình vẽ các cách biểu diễn trên). Nút cha trong cả hai trường hợp trên phải luôn có màu đen do tham chiếu đến nó chính là con trỏ giữa các **nút `B-tree`**. Như vậy, chúng ta thấy trong cây đỏ đen không thể có một nút đỏ mà có nút cha màu đỏ. Vậy "điều kiện đỏ" như sau đây sẽ bảo đảm rằng **mọi nút `B-tree` trong cây đỏ đen có không quá ba nút:**

Nếu một nút có màu đỏ, thì nút cha của nó phải tồn tại và có màu đen.

Do chúng ta quy ước nút gốc có màu đen nên điều kiện trên vẫn thỏa.

Chúng ta có thể tổng kết lại các điều phân tích trên đây thành một định nghĩa hình thức cho cây đỏ đen như sau (và chúng ta không cần nhắc đến cây `B-tree` nữa khi nói đến cây đỏ đen):

Định nghĩa: Một cây đỏ đen là một cây tìm kiếm nhị phân, trong đó mỗi nút có màu đỏ hoặc đen, thỏa các điều kiện sau:
1. Mọi đường đi từ nút gốc đến mỗi cây con rỗng (tham chiếu `NULL`) đều đi qua cùng một số nút đen như nhau.
2. Nếu một nút có màu đỏ thì nút cha của nó phải tồn tại và có màu đen.

Định nghĩa này dẫn đến một điều là trong cây đỏ đen không có một đường đi nào từ gốc đến một cây con rỗng có thể dài hơn gấp đôi một đường đi khác, bởi vì, theo điều kiện đen, số nút đen của tất cả các đường đi này phải bằng nhau, và theo điều kiện đỏ thì số nút đỏ phải nhỏ hơn hay bằng số nút đen. Do đó, ta có định lý sau:

Định lý: Chiều cao của một cây đỏ đen n nút không lớn hơn 2 lg n.

Thời gian tìm kiếm trong một cây đỏ đen có n nút là $O(\log n)$ trong mọi trường hợp. Chúng ta cũng sẽ thấy rằng thời gian thêm nút mới cũng là $O(\log n)$, nhưng trước hết chúng ta cần phát triển giải thuật trước.

Chúng ta nhớ lại trong phần 10.4, cây AVL, trong trường hợp xấu nhất, có chiều cao bằng 1.44 lg n, và trong trường hợp trung bình, có chiều cao thấp hơn. Sự khác nhau về chiều cao liên quan đến số nút của hai cây này là do cây đỏ đen không cân bằng tốt bằng cây AVL. Tuy nhiên, điều này không có nghĩa là các thao tác dữ liệu trên cây đỏ đen nhất thiết phải chậm hơn cây AVL, do cây AVL có thể cần đến nhiều phép quay để duy trì sự cân bằng hơn những gì mà cây đỏ đen cần đến.

10.4.3. Đặc tả cây đỏ đen

Để đặc tả một lớp C++ nhằm biểu diễn cho các đối tượng của cây đỏ đen, chúng ta cần khảo sát một vài tác vụ trên chúng. Chúng ta có thể hiện thực cây đỏ đen như là cây `B-tree` mà các nút của nó chứa các cây tìm kiếm thay vì các danh sách liên tục. Cách tiếp cận này buộc chúng ta phải viết lại nhiều phương thức và hàm phụ trợ đã có đối với cây `B-tree`, do phiên bản trước đây của `B-tree` liên quan chặt chẽ đến hiện thực liên tục của các phần tử trong mỗi nút.

Vì thế chúng ta sẽ đề xuất một hiện thực lớp cây đỏ đen thừa kế các đặc tính của lớp cây tìm kiếm trong phần 10.2.

Chúng ta bắt đầu bằng cách thêm thuộc tính màu vào mỗi nút của cây đỏ đen:

```
enum Color {red, black};

template <class Record>
struct RB_node: public Binary_node<Record> {
   Color color;
   RB_node(const Record &new_entry) { color = red; data = new_entry;
                                      left = right = NULL; }
   RB_node()                { color = red; left = right = NULL; }
   void set_color(Color c)  { color = c; }
   Color get_color() const  { return color; }
};
```

Để thuận tiện, chúng ta sử dụng các định nghĩa trong dòng (*inline definition*) cho các *contructor* và một số phương thức khác của **RB_node**. Cấu trúc **struct RB_node** rất giống với cấu trúc **struct AVL_node** dùng trong cây AVL trước kia trong phần 10.4: sự khác nhau duy nhất chỉ là thuộc tính màu thay cho thuộc tính cân bằng.

Để có thể gọi các phương thức **get_color** và **set_color** thông qua các con trỏ chỉ đến **Binary_node**, chúng ta cần bổ sung các hàm ảo tương ứng trong **struct Binary_node**, tương tự như chúng ta đã làm khi xây dựng cây AVL.

Cấu trúc của **Binary_node** đã sửa đổi như sau:

```
template <class Entry>
struct Binary_node {
   Entry data;
   Binary_node<Entry> *left;
   Binary_node<Entry> *right;
   virtual Color get_color() const { return red; }
   virtual void set_color(Color c) { }
   Binary_node()                   { left = right = NULL; }
   Binary_node(const Entry &x)     { data = x; left = right = NULL; }
};
```

Bằng cách sửa đổi như vậy, chúng ta đã có thể sử dụng lại mọi phương thức và hàm xử lý cho cây nhị phân tìm kiếm và các nút của nó. Việc tìm kiếm và duyệt trên cây đỏ đen tương tự như đối với cây nhị phân tìm kiếm.

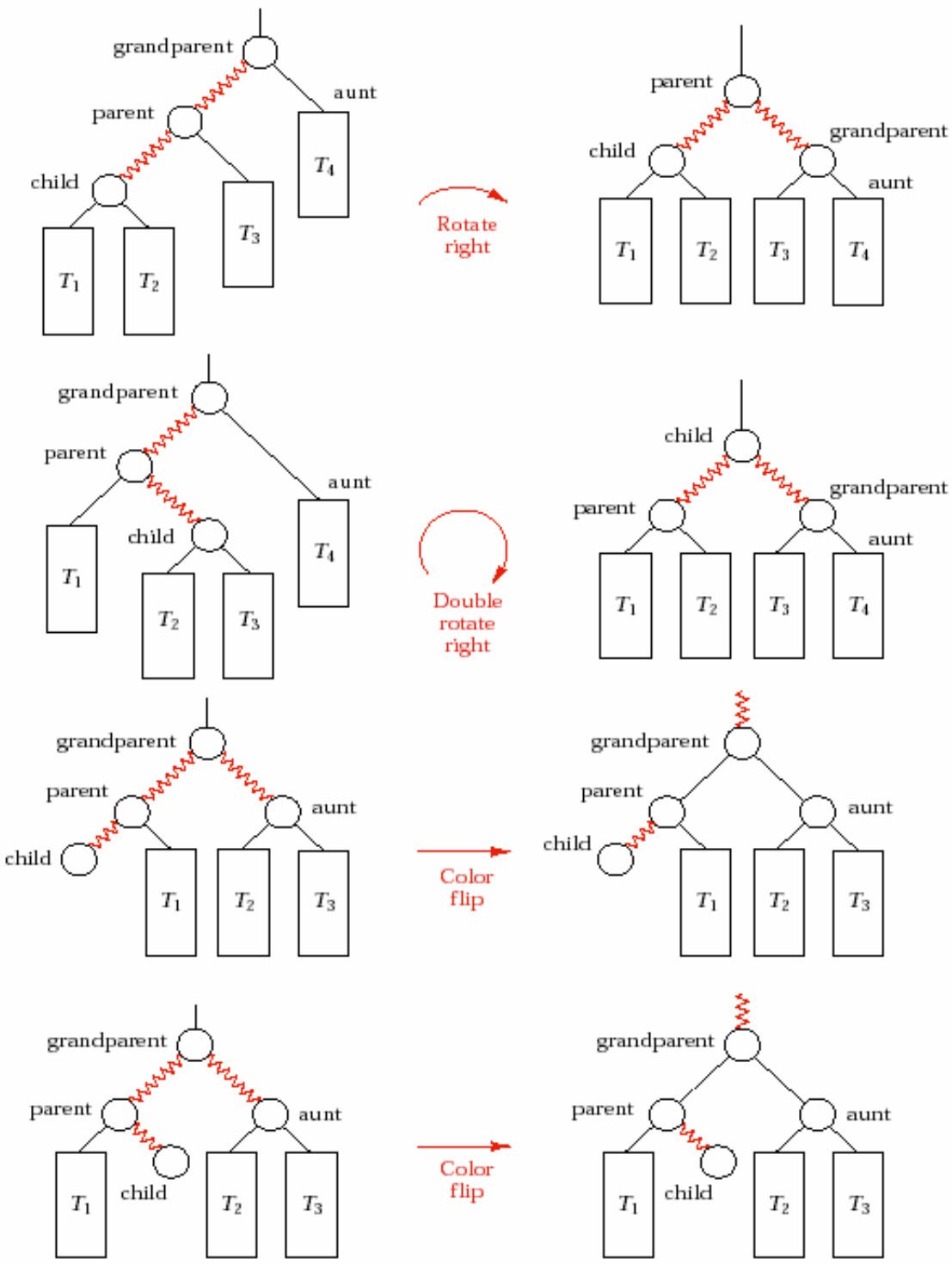
Chúng ta sẽ viết phương thức thêm phần tử vào cây đỏ đen sao cho nó vẫn giữ được tính chất của cây đỏ đen sau khi thêm vào.

```
template <class Record>
class RB_tree: public Search_tree<Record> {
```

```
public:
   Error_code insert(const Record & new_entry);
private:  //  Các hàm phụ trợ.
};
```

10.4.4. Thêm phần tử

Chúng ta hãy bắt đầu từ giải thuật đệ quy chuẩn đối với việc thêm vào một cây tìm kiếm nhị phân. Nghĩa là, chúng ta sẽ so sánh khóa mới của **target** với khóa của gốc, nếu cây không rỗng, và sau đó thêm đệ quy nút mới vào cây con trái hoặc cây con phải của gốc. Quá trình này kết thúc khi chúng ta gặp một cây con rỗng, tại đó chúng ta sẽ tạo một nút mới.



Hình 10.17 – Khôi phục các điều kiện đỏ và đen.

Nút mới này nên là màu đỏ hay là màu đen? Nếu cho nó màu đen, chúng ta đã làm tăng số nút đen trên đường đi từ gốc đến nó, và vi phạm điều kiện đen. Vậy nút mới phải có màu đỏ. Như **vậy khi thêm một nút mới vào cây đỏ đen thì**

nó luôn có màu đỏ. Nếu nút cha của nút mới này có màu đen, việc thêm vào kết thúc. Ngược lại, điều kiện đỏ bị vi phạm, và điều vi phạm này cần phải được giải quyết. Chúng ta sẽ phân tích các trường hợp và sẽ xử lý riêng rẽ cho chúng.

Giải thuật của chúng ta khá là đơn giản nếu chúng ta không xem xét các trường hợp này ngay lập tức mà trì hoãn chúng lại trong chừng mực có thể. Khi tạo một nút đỏ, chúng ta sẽ không cố gắng điều chỉnh lại cây ngay, thay vào đó, chúng ta chỉ đơn giản trả về một chỉ số trạng thái cho biết nút vừa xử lý xong có màu đỏ.

Khi một lần gọi đệ quy kết thúc, chỉ số trạng thái này được gởi ngược về lần đệ quy đã gọi nó, việc xử lý sẽ được thực hiện ở nút cha. Nếu nút cha có màu đen, các điều kiện của cây đỏ đen không bị vi phạm, quá trình xử lý kết thúc. Nếu nút cha có màu đỏ, chúng ta cũng sẽ không giải quyết ngay, mà một lần nữa lại gán chỉ số trạng thái cho biết vừa có hai nút màu đỏ. Lần đệ quy bên trên tại nút ông sẽ nhận được chỉ số trạng thái này, có kèm thêm thông tin cho biết hai nút màu đỏ vừa rồi thuộc cây con trái hay cây con phải của nó.

Sau khi các lần đệ quy trả về đến nút ông, việc xử lý sẽ được tiến hành tại đây. Chúng ta biết rằng nút ông luôn có màu đen, do trước khi xuất hiện nút con màu đỏ, các màu của nút cha và nút ông phải thỏa điều kiện của cây đỏ đen, mà nút cha đỏ thì nút ông phải đen. Quy ước nút gốc màu đen có lợi trong trường hợp này, vì nếu nút cha màu đỏ, thì nó không thể là nút gốc. Khi quá trình xử lý lan truyền về tận gốc thì nút ông phải luôn luôn tồn tại.

Cuối cùng, tại lần đệ quy của nút ông, chúng ta có thể biến đổi cây để khôi phục lại các điều kiện đỏ đen. Chúng ta chỉ cần xem xét trường hợp hai nút đỏ thuộc cây con trái của nút ông. Trường hợp ngược lại chỉ là đối xứng. Chúng ta cần phân biệt hai trường hợp tương ứng với màu của nút con còn lại của nút ông, nghĩa là nút chú của nút đỏ mới xuất hiện.

Trước tiên, giả sử nút chú có màu đen. Trường hợp này có gộp cả trường hợp nút chú rỗng, do quy ước cây con rỗng có màu đen. Các đặc tính của cây đỏ đen sẽ được khôi phục nhờ một phép quay đơn hay một phép quay kép qua phải, như hai phần đầu của hình 10.17. Trong cả hai sơ đồ này, phép quay, kéo theo sự thay đổi các màu nút tương ứng, sẽ loại được sự vi phạm điều kiện đỏ, đồng thời bảo toàn điều kiện đen do không làm thay đổi số nút đen trong bất kỳ đường đi nào từ gốc đến các nút lá.

Giờ chúng ta giả sử nút chú có màu đỏ, như hai phần bên dưới của hình 10.17. Việc biến đổi rất đơn giản: không có phép quay nào xảy ra, chỉ có sự thay đổi các màu. Nút cha và nút chú trở thành màu đen, nút ông trở thành màu đỏ. Điều kiện đỏ vẫn bảo đảm khi xét mối quan hệ giữa nút ông và nút cha, nút chú; giữa nút

cha và nút con màu đỏ mới xuất hiện bên dưới. Điều kiện đen không bị vi phạm do số nút đen trong các đường đi dọc theo cây không hề thay đổi. Tuy nhiên, khi nút ông vừa được biến đổi thành màu đỏ, điều kiện đỏ có thể bị vi phạm khi xét mối quan hệ với nút cha của nút này. Quá trình xử lý chưa thể chấm dứt. Mặc dù vậy, chúng ta có thể thấy rằng nút ông vừa được đổi sang màu đỏ hoàn toàn giống với trường hợp một nút đỏ mới xuất hiện lúc trước. Vậy chúng ta chỉ cần để lại cho các lần gọi đệ quy bên trên xử lý tại các nút bên trên nữa trong cây, bằng cách lại cho chỉ số trạng thái về lại trường hợp có một nút đỏ mới xuất hiện. Như vậy quá trình xử lý khi điều kiện đỏ bị vi phạm được lan truyền dọc lên phía trên của cây. Quá trình này có thể kết thúc tại một nút nào đó, hoặc có thể lan truyền lên tận gốc. Và khi nút gốc cần được đổi sang màu đỏ, thì chương trình ngoài cùng chỉ cần đổi nút này thành màu đen cho đúng quy ước. Điều này cũng không vi phạm điều kiện đen, do nó làm cho số nút đen trong tất cả các đường đi dọc theo cây tăng thêm một đơn vị. Và đây cũng chính là trường hợp duy nhất làm cho số nút đen trong các đường đi này tăng lên.

10.4.5. Phương thức thêm vào. Hiện thực

Chúng ta sẽ chuyển giải thuật trên thành chương trình C++. Cũng như mọi khi, phần lớn công việc đều được thực hiện bởi hàm đệ quy, phương thức **insert** chỉ cần đổi nút gốc thành màu đen khi cần và kiểm tra lỗi. Phần quan trọng nhất của giải thuật thêm phần tử vào cây đỏ đen là nắm giữ được chỉ số trạng thái mỗi khi có một lần đệ quy kết thúc. Chúng ta cần một kiểu liệt kê mới để phân biệt các trạng thái:

```
enum RB_code {okay, red_node, left_red, right_red, duplicate};
/*   Các giá trị trạng thái mà một lần đệ quy cần chuẩn bị trước khi kết thúc để trả về cho lần đệ
quy bên trên như sau (giả sử gọi nút đang được xử lý trong lần đệ quy hiện tại là *current):

okay:       Màu của *current không có sự thay đổi nào.

red_node:   Màu của *current vừa chuyển từ đen sang đỏ. Lần đệ quy bên trên khi nhận được
            thông tin này cần xem xét để xử lý.

right_red:  Màu của *current  và nút con phải của nó đều là đỏ, có sự vi phạm điều kiện đỏ.
            Lần đệ quy bên trên khi nhận được thông tin này cần thực hiện phép quay hoặc đổi
            màu cần thiết.

left_red:   Màu của *current  và nút con trái của nó đều là đỏ, có sự vi phạm điều kiện đỏ.
            Lần đệ quy bên trên khi nhận được thông tin này cần thực hiện phép quay hoặc đổi
            màu cần thiết.

duplicate:  Phần tử cần thêm vào cây đã có trong cây.
*/
```

```
template <class Record>
Error_code RB_tree<Record>::insert(const Record &new_entry)
```

```
/*
post: Nếu khóa trong new_entry đã có trong RB_tree, phương thức trả về
      duplicate_error. Ngược lại, phương thức trả về success và new_entry được thêm
      vào cây sao cho cây vẫn thỏa cây RB-tree.
uses: Các phương thức của struct RB_node và hàm đệ quy rb_insert.
*/
{
   RB_code status = rb_insert(root, new_entry);
   switch (status) {
      case red_node:
         root->set_color(black);
      case okay:
         return success;
      case duplicate:
         return duplicate_error;
      case right_red:
      case left_red:
         cout << "WARNING: Program error detected in RB_tree::insert" <<
              endl;
         return internal_error;
   }
}
```

Hàm đệ quy **rb_insert** thực hiện thực sự việc thêm phần tử mới vào cây: tìm kiếm trong cây theo cách thông thường, gặp cây con rỗng, thêm nút mới vào tại đây, các việc còn lại được thực hiện trên đường quay về của các lần gọi đệ quy. Hàm này có gọi **modify_left** hoặc **modify_right** để thực hiện các phép quay và đổi màu tương ứng với các trường hợp trong hình 10.17.

```
template <class Record>
RB_code RB_tree<Record>::rb_insert(Binary_node<Record> *&current,
                                   const Record &new_entry)
/*
pre:  current là NULL hoặc là địa chỉ của nút gốc của một cây con trong RB_tree.
post: Nếu khóa trong new_entry đã có trong RB_tree, phương thức trả về
      duplicate_error. Ngược lại, phương thức trả về success và new_entry được thêm
      vào cây con có gốc là current. Tính chất cây đỏ đen trong cây con này vẫn thỏa ngoại
      trừ màu tại nút gốc của nó và một trong hai nút con của nút gốc này. Trạng thái này sẽ
      được hàm modify_right hoặc modify_left điều chỉnh thích hợp (tương ứng các trị của
      RB_code) để trả về cho lần đệ quy bên trên của lần gọi rb_insert này.
uses: Các phương thức của lớp RB_node, các hàm rb_insert (một cách đệ quy),
      modify_left, và modify_right.
*/
{
   RB_code status,
           child_status;
   if (current == NULL) {
      current = new RB_node<Record>(new_entry);
      status = red_node;
   }

   else if (new_entry == current->data)
      return duplicate;
   else if (new_entry < current->data) {
```

```
      child_status = rb_insert(current->left, new_entry);
      status = modify_left(current, child_status);
   }
   else {
      child_status = rb_insert(current->right, new_entry);
      status = modify_right(current, child_status);
   }
   return status;
}
```

Hàm **modify_left** dựa vào trạng thái của các nút con để thực hiện phép quay hay sửa đổi màu tương ứng, đồng thời cập nhật lại và trả về chỉ số trạng thái cho **rb_insert**. Chính trong hàm này chúng ta quyết định trì hoãn công việc khôi phục các đặc tính của cây đỏ đen. Khi **modify_left** được gọi, chúng ta biết rằng việc thêm vào vừa được thực hiện trong cây con bên trái của nút hiện tại, biết được màu của nút hiện tại, và thông qua chỉ số trạng thái, chúng ta còn biết được trường hợp nào đã xảy ra ở cây con trái của nó. Bằng cách sử dụng các thông tin này, chúng ta có thể xác định chính xác những việc cần làm để khôi phục các đặc tính của cây đỏ đen.

```
template <class Record>
RB_code RB_tree<Record>::modify_left(Binary_node<Record> *&current,
                                     RB_code &child_status)
/*
pre:  Cây con bên trái của current vừa được thêm nút mới, trị trong child_status sẽ quyết
      định việc xử lý kế tiếp trong hàm này.
post: Việc đổi màu hoặc quay cần thiết đã được thực hiện, trạng thái thích hợp được trả về bởi
      hàm này.
uses: Các phương thức của struct RB_node, các hàm rotate_right,
      double_rotate_right, và flip_color.
*/
{
   RB_code status = okay;
   Binary_node<Record> *aunt = current->right;
   Color aunt_color = black;
   if (aunt != NULL) aunt_color = aunt->get_color();
   switch (child_status) {
   case okay:
      break;  //  Việc xử lý đã kết thúc và không cần lan truyền lên trên nữa.
   case red_node:
      if (current->get_color() == red)
         status = left_red;
      else
         status = okay;// current màu đen, nút con trái màu đỏ, đã thỏa cây
                                                                RB-tree.
      break;
   case left_red:
      if (aunt_color == black) status = rotate_right(current);
      else                     status = flip_color(current);
      break;
   case right_red:
      if (aunt_color == black) status = double_rotate_right(current);
      else                     status = flip_color(current);
```

```
      break;
   }
   return status;
}
```

Hàm phụ trợ **modify_right** cũng tương tự, nó xử lý cho các tình huống cây có dạng như những hình ảnh phản chiếu qua gương của các tình huống trong hình 10.17. Hàm đổi màu **flip_color** được xem như bài tập. Các hàm quay dựa trên cơ sở các hàm quay của cây AVL, có thêm việc gán lại các màu và chỉ số trạng thái thích hợp.

10.4.6. Loại một nút

Cũng như cây B-tree, việc loại bỏ một nút phức tạp hơn việc thêm vào, đối với cây đỏ đen, việc loại nút còn khó khăn hơn rất nhiều. Việc thêm vào tạo ra một nút mới màu đỏ dẫn đến nguy cơ vi phạm điều kiện đỏ, chúng ta cần xem xét một cách cẩn thận để giải quyết các vi phạm này. Việc loại một nút đỏ ra khỏi cây không khó lắm. Tuy nhiên, việc loại một nút đen khỏi cây dẫn đến nguy cơ vi phạm điều kiện đen, và nó đòi hỏi chúng ta phải xem xét rất nhiều trường hợp đặc biệt để khôi phục điều kiện đen cho cây.

Chương 11 – HÀNG ƯU TIÊN

Cấu trúc dữ liệu hàng đợi mà chúng ta đã xem xét trong chương 3 là theo đúng nguyên tắc FIFO. Tuy nhiên trong thực tế, có những trường hợp cần có sự linh động hơn. Chẳng hạn trong số các công việc cần xử lý, có một số ít công việc vô cùng quan trọng, chúng cần được xử lý càng sớm càng tốt ngay khi có thể. Hoặc trong trường hợp có nhiều tập tin cùng đang chờ được in, một số tập tin chỉ có 1 trang trong khi một vài tập tin khác thì rất dài. Nếu các tập tin 1 trang được in trước thì không ảnh hưởng đến thời gian chờ đợi của các tập tin khác bao nhiêu. Ngược lại, nếu cứ theo thứ tự FIFO, một số bản in chỉ có 1 trang lại phải chờ đợi quá lâu.

Hàng có xét thứ tự ưu tiên, hay gọi tắt là hàng ưu tiên (*priority queue*), là một cách giải quyết các trường hợp trên một cách thỏa đáng. Tùy vào ứng dụng, tiêu chí để xét độ ưu tiên do chúng ta quyết định. Trong chương này chúng ta sẽ đặc tả và hiện thực CTDL cho hàng ưu tiên này.

11.1. Định nghĩa hàng ưu tiên

Hàng ưu tiên có các phương thức gần giống như một hàng đợi thông dụng, chỉ khác về mặt chiến lược:

Định nghĩa: Một **hàng ưu tiên** các phần tử kiểu T gồm các phần tử của T, kèm các tác vụ sau:

1. Tạo mới một đối tượng hàng rỗng.
2. **`priority_append`**: Thêm một phần tử mới vào hàng, giả sử hàng chưa đầy (tùy vào độ ưu tiên của phần tử dữ liệu mới nó sẽ được đứng ở một vị trí thích hợp).
3. **`priority_serve`**: Loại một phần tử ra khỏi hàng, giả sử hàng chưa rỗng (phần tử bị loại là phần tử đến lượt được xem xét theo quy ước độ ưu tiên đã định).
4. **`priority_retrieve`**: Xem phần tử tại đầu hàng (phần tử sắp được xem xét).

11.2. Các phương án hiện thực hàng ưu tiên

Giả sử độ ưu tiên là sự kết hợp bởi độ ưu tiên theo tiêu chí mà chúng ta đã chọn cùng với thứ tự xuất hiện của công việc. Khi đưa vào hàng, mỗi công việc sẽ có một thông số để chứa độ ưu tiên này. Chúng ta quy ước rằng độ ưu tiên càng nhỏ thứ tự ưu tiên càng cao.

Chúng ta có thể dùng DSLK đơn để hiện thực hàng ưu tiên. Việc thêm vào luôn thực hiện ở đầu danh sách, việc lấy ra sẽ phải duyệt lần lượt hết danh sách để chọn phần tử có độ ưu tiên cao nhất. Ngược lại, nếu khi thêm vào luôn giữ

danh sách đúng thứ tự tăng dần của độ ưu tiên, khi lấy ra chỉ cần lấy phần tử đầu danh sách. Cả hai cách đều tốn O(1) cho một tác vụ và O(N) cho tác vụ còn lại.

Phương án thứ hai có thể hiện thực hàng ưu tiên bởi một cây nhị phân tìm kiếm (BST). Phương án này cần O(logN) cho mỗi tác vụ thêm hoặc loại phần tử. Tác vụ `priority_serve` sẽ luôn lấy ra phần tử cực trái của cây nhị phân, điều này sẽ làm cho cây mất cân bằng và có thể khắc phục bằng cách sử dụng cây AVL thay cho cây BST bình thường.

Tuy nhiên các phương án sử dụng cây trên đây hơi bị thừa. Việc quản lý cây quá phức tạp so với mục đích của chúng ta.

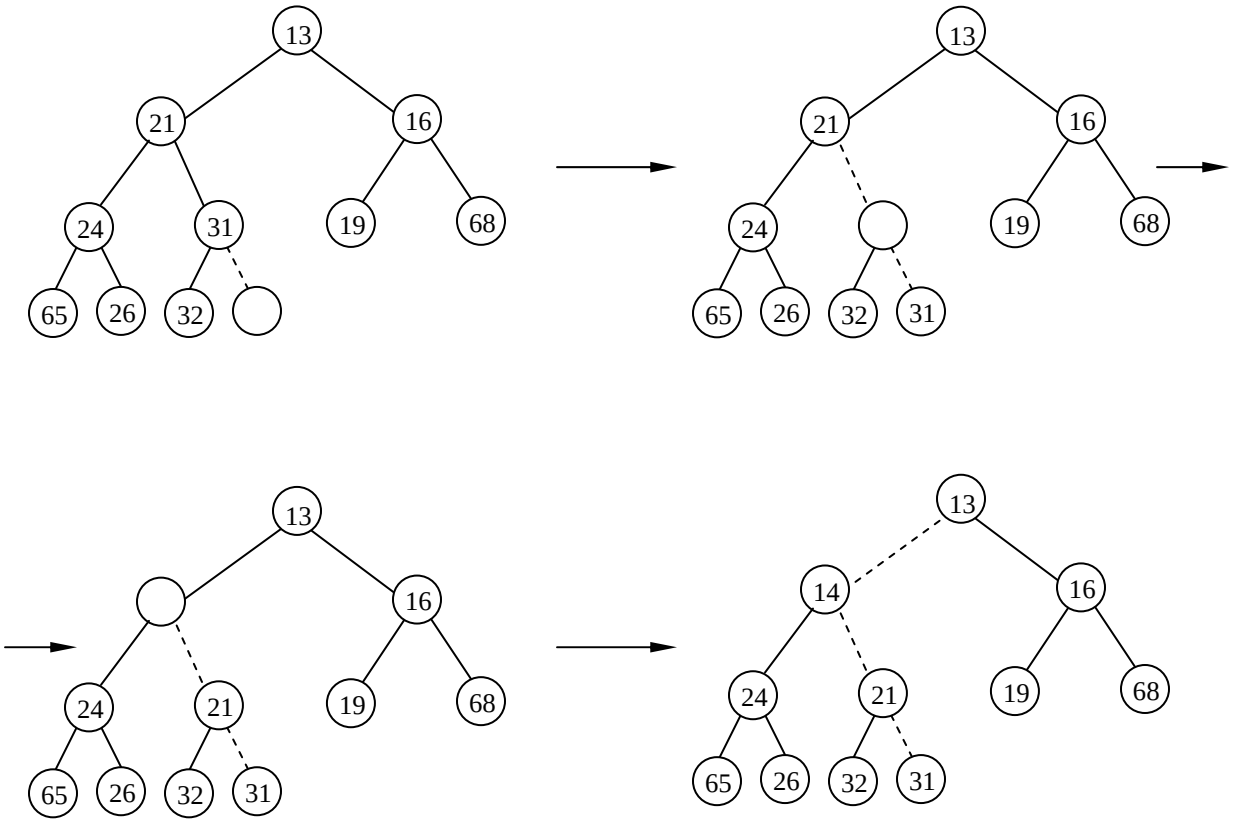
Cách hiện thực đơn giản và phổ biến cho hàng ưu tiên là `heap` nhị phân (*binary heap*), có khi còn được gọi tắt là `heap`. Và vì nó khá phổ biến nên nhiều lúc người ta chỉ gọi đơn giản là `heap`, chứ không còn gọi là hàng ưu tiên nữa. Định nghĩa `heap` nhị phân đã được trình bày trong chương 8. Trong chương này chúng ta sử dụng `heap` nhị phân là một `min-heap`.

Phần hiện thực một CTDL `heap` cụ thể được xem như bài tập. Phần tiếp sau đây trình bày các thao tác trên `heap` bằng mã giả, và để dễ dàng hình dung chúng ta cũng vẫn dùng hình ảnh của cây nhị phân để minh họa.

11.3. Hiện thực các tác vụ cơ bản trên heap nhị phân

11.3.1. Tác vụ thêm phần tử

Để thêm một phần tử mới vào heap, chúng ta tạo một chỗ trống ngay sau phần tử cuối của heap, điều này bảo đảm heap vẫn có cấu trúc cây nhị phân đầy đủ hoặc gần như đầy đủ. Nếu phần tử mới có thể đặt vào chỗ trống này mà không vi phạm điều kiện thứ hai của heap (bằng cách so sánh phần tử mới với nút cha của chỗ trống này) thì giải thuật kết thúc. Ngược lại, chúng ta lấy phần tử cha của chỗ trống này để lấp vào chỗ trống, lúc đó sẽ xuất hiện chỗ trống mới. Công việc lập lại tương tự cho đến khi tìm được vị trí thích hợp cho phần tử mới.
Hình 11.1 minh họa việc thêm phần tử 14 vào một `heap`.
Việc thêm phần tử mới tốn nhiều nhất là O(logN).

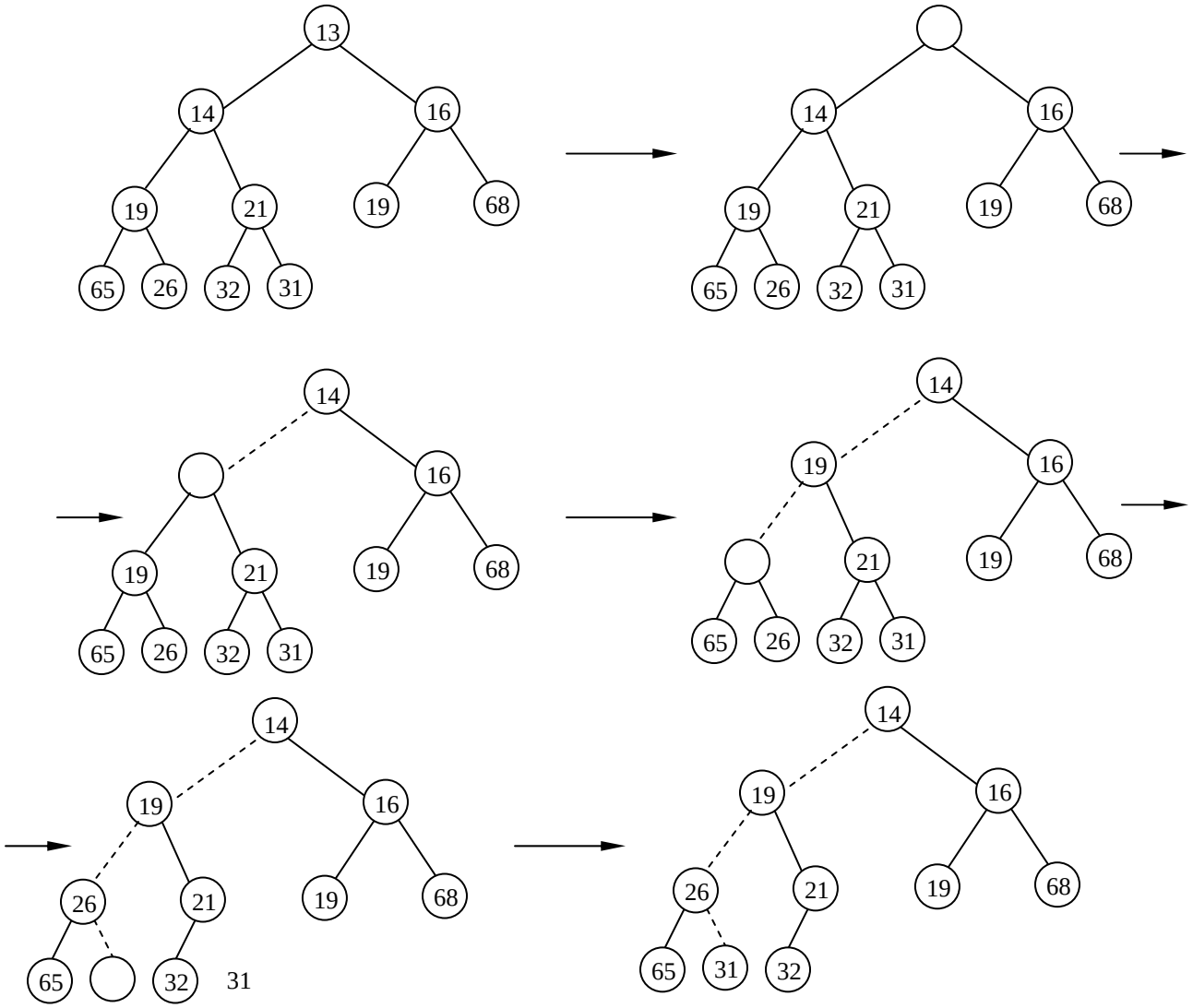


Hình 11.1. Thêm phần tử 14 vào heap

```
ErrorCode Priority_Queue::priority_append(Entry item)
/*
pre: đối tượng Priority_Queue có thuộc tính heap là mảng liên tục chứa các phần tử.
post: item được thêm vào hàng ưu tiên sao cho tính chất heap vẫn thoả.
*/
{
   1. if (full())
      1. return overflow;
   2. else
      1. current_position = size()
      2. loop ((tồn tại parent là cha của phần tử tại current_position) AND
                                                          (parent > item)
           // Lần lượt di chuyển các nút cha xuống nếu cha lớn hơn item để lấp chỗ trống
         1. heap[current_position] = parent
         2. Cho current_position là vị trí của parent
      3. endloop
      4. heap[current_position] = item
      5. // Cập nhật lại kích thước của heap.
      6. return success
   3. endif
}
```

11.3.2. Tác vụ loại phần tử

Việc loại phần tử cũng tương tự việc thêm vào. Phần tử lấy ra chính là phần tử tại gốc của cây vì đó là phần tử có độ ưu tiên cao nhất. Việc còn lại là giải quyết cho chỗ trống này. Trong hình 11.2 chúng ta sẽ thấy quá trình di chuyển của chỗ trống. Một trong hai phần tử con của nó sẽ di chuyển lấp đầy chỗ trống. Phần tử nhỏ nhất trong hai phần tử con được chọn để thỏa định nghĩa của heap. Cuối cùng, với chỗ trống không còn nút con thì được lấp đầy bởi phần tử cuối của heap vì chúng ta luôn biết rằng đây là cây nhị phân đầy đủ hoặc gần như đầy đủ, nó luôn chứa các phần tử có thể điền vào một mảng liên tục từ trái sang phải. Và thực sự chúng ta cũng hiện thực heap trong một mảng liên tục chứ không phải cấu trúc cây có con trỏ. Mọi thao tác với các chỉ số để định vị đến các phần tử cha, con, đều rất nhanh chóng. Chúng ta có thể chắc chắn rằng điều kiện thứ hai trong định nghĩa của heap cũng không bị vi phạm khi dời phần tử cuối bằng cách này. Chi phí trong việc loại phần tử là O(logN).



Hình 11.2. Loại một phần tử ra khỏi heap

```
ErrorCode Priority_Queue::priority_serve()
/*
pre: đối tượng Priority_Queue có thuộc tính heap là mảng liên tục chứa các phần tử.
post: phần tử nhỏ nhất trong hàng ưu tiên được lấy đi.
*/
{
   1. if (empty())
      1. return underflow;
   2. else
      1. current_position = 0
      2. loop (phần tử tại current_position có con)// Lần lượt di chuyển các nút
                                                    // con lên để lấp chỗ trống.
         1. child là phần tử nhỏ nhất trong hai con
         2. child được chép lên vị trí current_position
         3. Cho current_position là vị trí của child vừa được chép
      3. endloop
      4. heap[current_position] = heap[size()-1]
      5. // Cập nhật lại kích thước của heap.
      6. return success
   3. endif
}
```

11.4. Các tác vụ khác trên heap nhị phân

11.4.1. Tác vụ tìm phần tử lớn nhất

Tác vụ priority_retrieve truy xuất phần tử bé nhất trong min-heap có chi phí O(1). Đối với việc tìm ra phần tử lớn nhất trong min-heap khó khăn hơn. Tuy vậy, chúng ta cũng không phải dùng cách tìm tuyến tính trên toàn bộ heap, vì các phần tử trong heap luôn có một trật tự nhất định theo định nghĩa. Chúng ta thấy ngay rằng phần tử lớn nhất phải là một trong các nút lá, đó là một nửa bên phải của mảng liên tục chứa các phần tử của heap.

11.4.2. Tác vụ tăng giảm độ ưu tiên

Tại thời điểm khi mà các công việc được đưa vào hàng ưu tiên, mỗi công việc đều đã được xác định độ ưu tiên, và chỉ số này chính là khóa được xử lý bởi heap. Tuy nhiên, giả sử trong khi các công việc đang nằm trong hàng ưu tiên để chờ đến lượt được cung cấp dịch vụ, người điều hành muốn can thiệp vào thứ tự ưu tiên này vì một số lý do. Chẳng hạn có một công việc đang phải chờ quá lâu và có một yêu cầu đột xuất cần thúc đẩy nó được hoàn thành sớm hơn. Ngược lại người điều hành cũng có thể muốn điều chỉnh giảm độ ưu tiên một công việc nào khác. Những điều này rất thường xảy ra nếu chúng ta xét trong một tình huống rằng, trong chế độ phục vụ có phân chia thời gian, khi hết khoảng thời gian quy định, nếu công việc vẫn chưa thực hiện xong thì lại phải quay lại hàng đợi nằm chờ tiếp. Mỗi công việc thường phải đợi, được phục vụ, đợi, được phục vụ,... vài lần như thế mới kết thúc. Như vậy thì việc người điều hành được quyền can thiệp vào hàng đợi tùy vào những tình thế cụ thể là rất có lợi. Những công việc đôi khi do

tính thiếu hiệu quả, đã sử dụng tổng thời gian phục vụ quá lớn mà vẫn chưa kết thúc, thường bị giảm độ ưu tiên như một hình thức phạt.

Trước hết chúng ta cần bổ sung một vài CTDL và một vài hàm phụ trợ sao cho khi một công việc được đưa vào hàng ưu tiên chúng ta luôn nắm được vị trí của nó trong hàng ưu tiên, kể cả khi nó bị dịch chuyển do các thao tác của các tác vụ thêm, loại,...Tất nhiên những bổ sung này sẽ được đóng kín trong CTDL `Priority_Heap` của chúng ta. Sau đó, chúng ta có thể thiết kế hai tác vụ **`decrease_key`**`(position, delta)` và **`increase_key`**`(position, delta)` cho phép giảm hoặc tăng khóa của phần tử tại vị trí `position` trong `heap` một lượng `delta`. Khi giảm hoặc tăng như vậy, chúng ta chỉ cần xử lý dịch chuyển phần tử này lên hoặc xuống vị trí thích hợp trong `heap` so với giá trị mới của khóa, việc này rất dễ dàng và gần giống với những gì chúng ta đã làm trong hai tác vụ `priority_append` và `priority_serve`.

11.4.3. Tác vụ loại một phần tử không ở đầu hàng

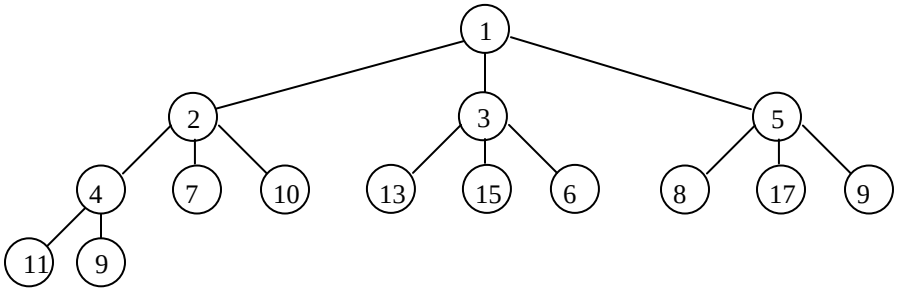
Chúng ta cũng có thể bổ sung thêm tác vụ loại hẳn một công việc đang đợi trong hàng (không phải là phần tử đang ở đầu hàng và có độ ưu tiên cao nhất) nhằm đáp ứng yêu cầu của một người sử dụng nào đó muốn ngưng không thực hiện công việc nữa. Tác vụ **`delete`**`(position)` đơn giản là gọi **`decrease_key`**`(position, delta)` với delta vô cùng lớn rồi gọi `priority_serve`.

11.5. Một số phương án khác của heap

Đặc tính chính yếu của heap là **trật tự giữa các phần tử cha – con**. Điều này đáp ứng mục đích của hàng ưu tiên là truy xuất nhanh chóng phần tử nhỏ nhất. `Heap` nhị phân khai thác tính năng của mảng liên tục tạo hiệu quả nhất định trong các thao tác trên hàng ưu tiên. Dưới đây là một vài phương án khác của `heap`, chúng khai thác các ưu điểm của các cách hiện thực khác nhau.

11.5.1. d-heaps

`Heap` nhị phân luôn phổ biến khi người ta cần dùng đến hàng ưu tiên. `d-heaps` hoàn toàn giống heap nhị phân ngoại trừ mỗi nút có d chứ không phải 2 con. d càng lớn càng lợi cho phép thêm vào, ngược lại, trong phép loại bỏ phần tử bé nhất, cần phải chi phí trong việc so sánh d phần tử con của một nút để lấy phần tử nhỏ nhất đẩy lên. Do đó `d-heap` thích hợp với các ứng dụng mà phép thêm vào được thực hiện thường xuyên. Ngoài ra còn phải tính đến chi phí trong việc định vị các nút cha, nút con trong mảng liên tục. Nếu d là lũy thừa của 2 thì các phép nhân, chia được thực hiện bởi phép dịch chuyển bit rất tiện lợi. Cuối cùng, tương tự `B-tree`, khi dữ liệu quá lớn không chứa đủ trong bộ nhớ thì `d-heap` cũng thích hợp với việc sử dụng thêm bộ nhớ ngoài.



Hình 11.3 . d-heap

Nhược điểm của các heap trên đây là việc tìm kiếm một phần tử bất kỳ hay việc trộn hai heap với nhau không thích hợp. Chúng ta sẽ xem xét một số cấu trúc phức tạp hơn nhưng rất thích hợp cho phép trộn.

11.5.2. Heap lệch trái (*Leftist heap*)

Việc sử dụng mảng liên tục như heap nhị phân thật không dễ để thực hiện phép trộn một cách hiệu quả, vì nó luôn đòi hỏi việc di chuyển các phần tử. Mọi CTDL thích hợp cho việc trộn đều dùng đến con trỏ. Nhược điểm của con trỏ là thời gian xác đinh vị trí các phần tử lâu hơn so với trong mảng liên tục. Heap lệch trái sẽ sử dụng cấu trúc liên kết với các con trỏ trái và phải tại mỗi nút để chứa địa chỉ của hai nút con, mục đích để khai thác điểm mạnh của phép trộn.

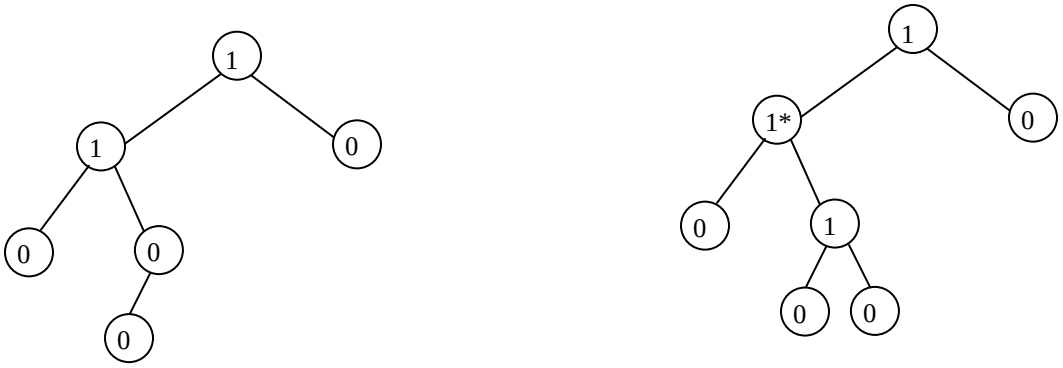
Heap lệch trái cũng giống với heap nhị phân ở cấu trúc nhị phân và trật tự giữa các phần tử cha – con. Chúng ta luôn nhớ rằng, trật tự giữa các phần tử cha – con là tính chất cơ bản nhất của mọi heap. Điểm khác ở đây là heap lệch trái không có sự cân bằng.

Chúng ta định nghĩa chiều dài đường đi đến NULL của một phần tử X (*null path length – Npl(X)*) là chiều dài của đường đi ngắn nhất từ X đến một nút lá. Npl của nút lá và nút bậc một bằng 0, Npl(NULL) = -1. Npl của bất kỳ nút nào bằng Npl của nút con có Npl nhỏ nhất cộng thêm 1.

Heap lệch trái có tính chất sau đây: tại mọi nút, Npl của nút con trái luôn lớn hơn hoặc bằng Npl của nút con phải. Tính chất này làm cho heap lệch trái mất cân bằng. Chúng ta gọi **đường đi trái** (**hoặc phải**) tại mỗi nút là đường đi đến nút dưới cực trái (cực phải) tương ứng của nút đó. Mọi nút trong heap lệch trái luôn có khuynh hướng có đường đi trái dài hơn đường đi phải, do đó đường đi phải tại nút gốc luôn là đường ngắn nhất trong các đường đi từ gốc đến các nút lá.

Có thể chứng minh bằng suy diễn rằng một cây heap lệch trái với r nút trên đường đi phải sẽ có ít nhất 2^r-1 nút. Từ đó cây heap lệch trái N nút sẽ có nhiều nhất $\lfloor \log(N+1) \rfloor$ nút trên đường đi phải. Ý tưởng chính của heap lệch trái là

mọi tác vụ đều được thực hiện trên đường đi phải để luôn được bảo đảm với chi phí `log(N)`.



(a) (b)

Hình 11.4. Npl tại mỗi nút.
(a)- Thoả điều kiện heap lệch trái.
(b)- Nút có dấu * vi phạm điều kiện heap lệch trái..

Các tác vụ trên heap lệch trái

Tác vụ cơ bản nhất của heap lệch trái là tác vụ trộn. Chúng ta sẽ thấy phép thêm vào và phép loại bỏ được thực hiện dễ dàng nhờ gọi phép trộn này.

Ngoài hai con trỏ trái và phải, mỗi nút trong heap lệch trái còn có thêm thông tin là Npl của nó.

Để trộn hai heap lệch trái H_1 và H_2:

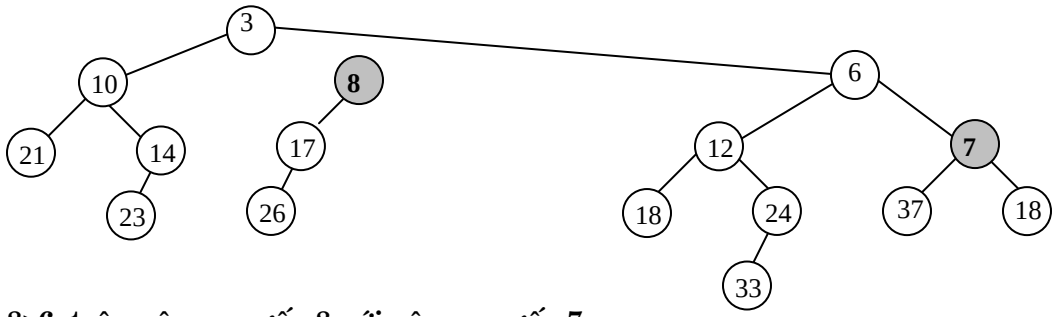
- Nếu một trong hai heap rỗng thì trả về heap còn lại.
- Ngược lại, so sánh dữ liệu tại hai nút gốc, thực hiện trộn heap có gốc lớn hơn với cây con phải của heap có gốc nhỏ hơn. Điều này bảo đảm gốc của heap kết quả luôn có trị nhỏ nhất trong tất cả các nút, vì heap kết quả có gốc chính là gốc của heap ban đầu có gốc nhỏ hơn.

Đây chính là quá trình đệ quy. Trước khi kết thúc một lần gọi đệ quy, chúng ta chỉ cần kiểm tra nút gốc này có thỏa điều kiện của heap lệch trái hay không, nếu cần chúng ta chỉ cần hoán vị hai con trái và phải của nó để có được Npl của nút con trái lớn hơn hoặc bằng Npl của nút con phải. Npl của nút gốc được cập nhật bằng Npl của nút con phải cộng thêm 1.

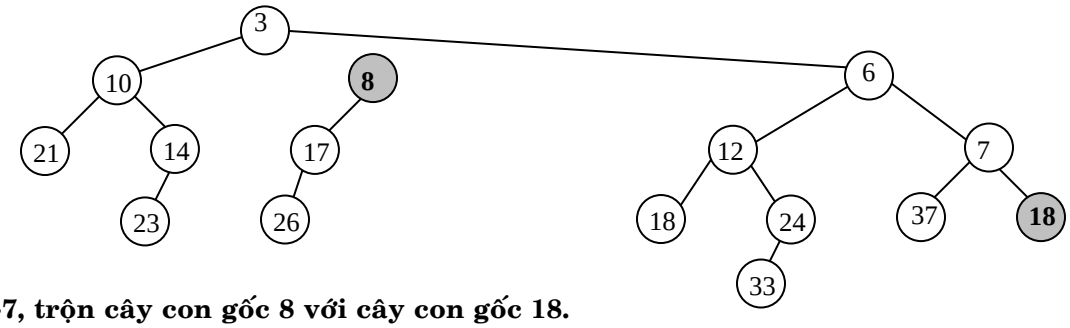
Hình 11.5 minh họa quá trình trộn hai heap lệch trái H_1 và H_2.

H_1 **H_2**

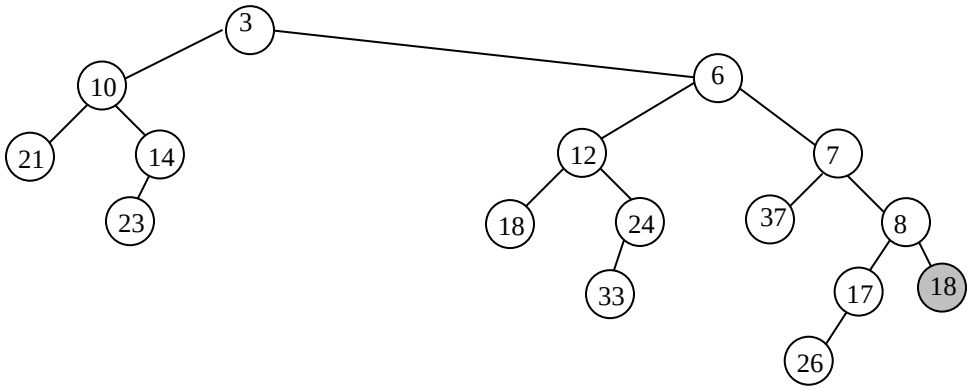
(a)- Do 6>3, trộn H_2 với cây con gốc 8.



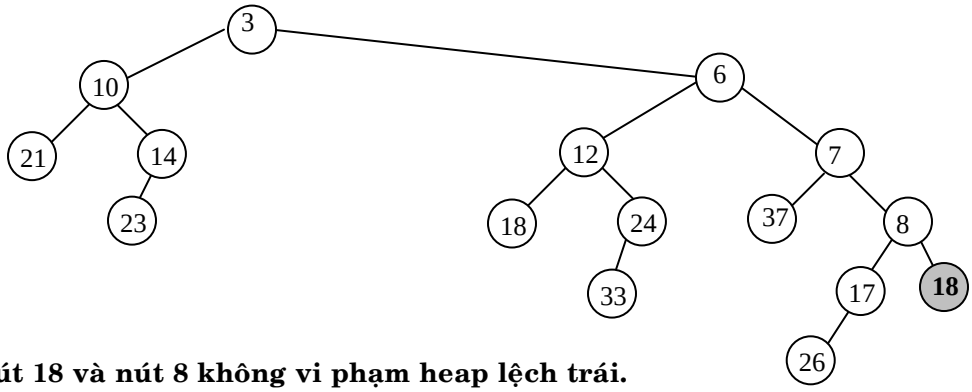
(b)- Do8>6, trộn cây con gốc 8 với cây con gốc 7.



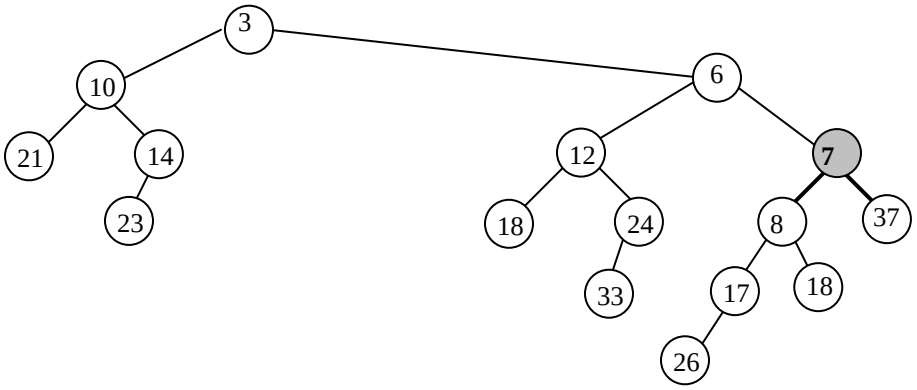
(c)-Do 8>7, trộn cây con gốc 8 với cây con gốc 18.



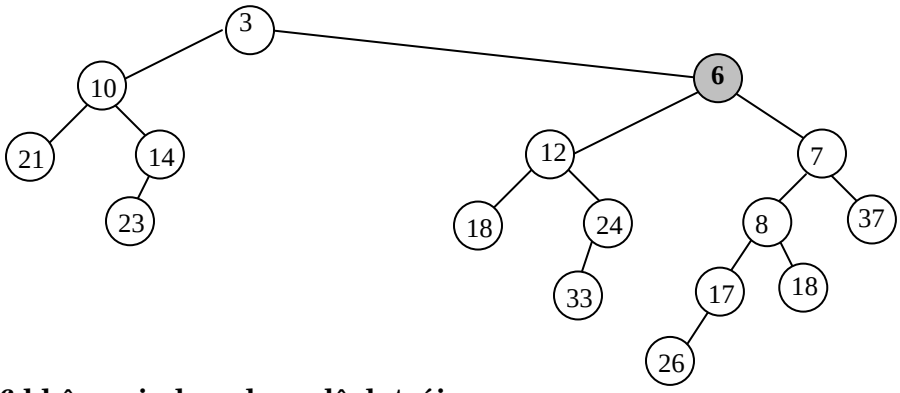
(d)- Do 18>8, trộn cây con gốc 18 với cây con phải của 8 (NULL)



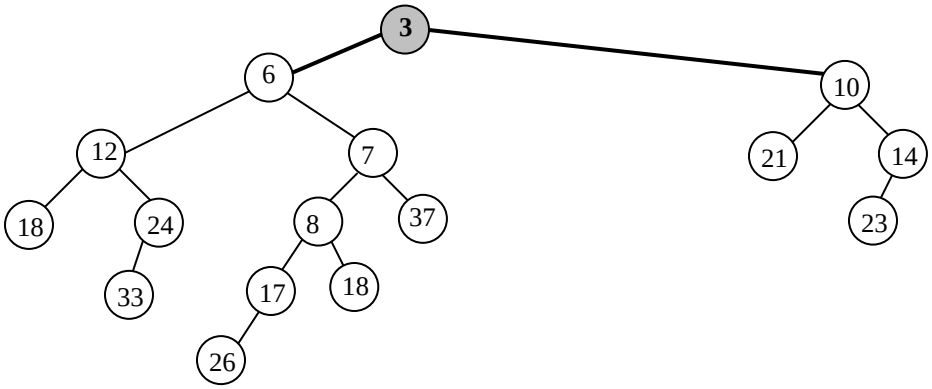
(e)- Tại nút 18 và nút 8 không vi phạm heap lệch trái.



(f)- Tại nút 7 vi phạm heap lệch trái, hoán vị hai cây con.



(g)- Tại nút 6 không vi phạm heap lệch trái.



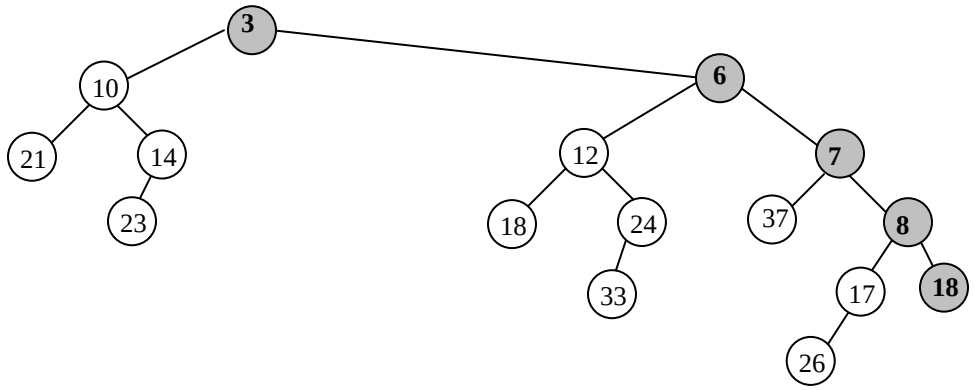
(h)- Tại nút 3 vi phạm heap lệch trái, hoán vị hai cây con.

Hình 11.5- Trộn hai heap lệch trái H_1 và H_2

```
//Phần mã giả cho khai báo và một số tác vụ cho LeftistHeap.

struct LeftistHeap_Node
   DataType          data
   LeftistHeap_Node* left
   LeftistHeap_Node* right
   int               Npl
end struct

class Leftist_Heap
   public:
      void merge(ref Leftist_Heap H1, ref Leftist_Heap H2)
   private:
      LeftistHeap_Node* recursive_merge(ref LeftistHeap_Node* p1,
                                        ref LeftistHeap_Node* p2)
      LeftistHeap_Node* aux_merge(ref LeftistHeap_Node* p1,
                                  ref LeftistHeap_Node* p2)
      LeftistHeap_Node* root
end class


void Leftist_Heap::merge(ref Leftist_Heap H1, ref Leftist_Heap H2)
/*
post: H1 là heap lệch trái là kết quả trộn hai heap H1 và H2, H2 rỗng.
*/
{
   1. recursive_merge(H1.root, H2.root);
}


LeftistHeap_Node* Leftist_Heap::recursive_merge(ref LeftistHeap_Node* p1,
                                                ref LeftistHeap_Node* p2)
/*
pre: p1 và p2 là đĩa chỉ nút gốc của hai heap lệch trái.
post: Trả về địa chỉ nút gốc của heap lệch trái là kết quả trộn hai heap ban đầu, p1 và p2 là
                                                                        NULL.
uses: Sử dụng hàm aux_merge.
*/
{
   LeftistHeap_Node* p;
   1. if (p1 == NULL)
      1. p = p2;
   2. if (p2 = NULL)
      1. p = p1;
   3. if (p1->data < p2->data)
      1. p = aux_merge(p1, p2);
   4. else
      1. p = aux_merge(p2, p1);
   5. p1 = NULL;
   6. p2 = NULL;
   7. return p;
}
```

```
LeftistHeap_Node* Leftist_Heap::aux_merge(ref LeftistHeap_Node* p1,
                                          ref LeftistHeap_Node* p2)
/*
pre: p1 và p2 là địa chỉ nút gốc của hai heap lệch trái.
post: Heap p2 được trộn với cây con phải của heap p1, p2 gán về NULL.
uses: Sử dụng hàm SwapChildren và recursive_merge.
*/
{
   1. if (p1->left == NULL) // Trường hợp này p1_>right đã là NULL do điều kiện
      1. p1->left = p2;     // của heap lệch trái.
   2. else
      1. p1->right = recursive_merge(p1->right, p2);
      2. if(p1->left->Npl < p1->right->Npl)
         1. SwapChildren(p1); // Tráo 2 cây con của p1.
      3. p1->Npl = p1->right->Npl + 1;
   3. return p1; // p2 đã được gán NULL trong recursive_merge.
}
```

Thời gian trộn hai heap lệch trái tỉ lệ với chiều dài của đường đi phải, và đó là O(logN). Giải thuật đệ quy trên có thể được khử đệ quy như sau. Bước thứ nhất thực hiện trộn đường đi phải của hai heap, đường đi phải của heap kết quả chính là thứ tự tăng dần của các nút trên đường đi phải của hai heap ban đầu (3, 6, 7, 8, 18). Kết quả cho thấy ở hình 11.6. Bước thứ hai đi lần ngược trên đường đi phải của cây kết quả để hoán vị các con trái và con phải khi cần thiết. Trường hợp chúng ta việc hoán vị cần thực hiện tại nút 7 và nút 3.



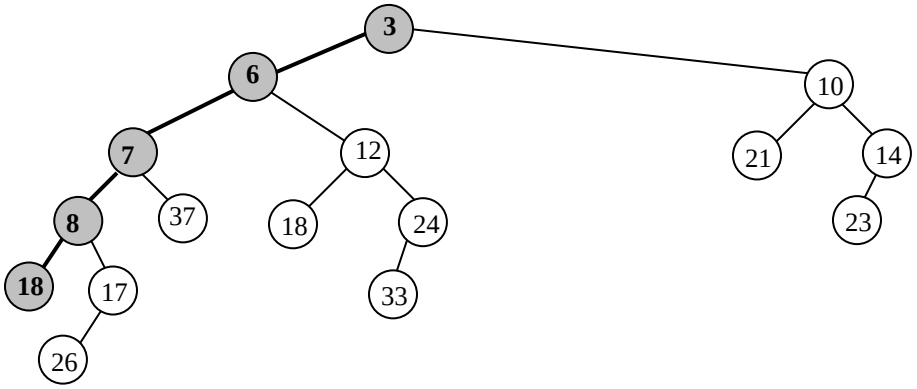
Hình 11.6. Kết quả sau bước thứ nhất của giải thuật trộn không đệ quy.

Phép thêm một phần tử mới chính là phép trộn heap lệch trái đã có với một heap lệch trái chỉ có duy nhất nút cần thêm vào. Phép loại phần tử nhỏ nhất của heap lệch trái là phép lấy đi phần tử tại gốc và trộn hai cây con của nó với nhau. Như vậy tất cả các tác vụ trên heap lệch trái đều có chi phí O(logN).

11.5.3. Skew heap

Skew heap giống như heap lệch trái nhưng không chứa thông tin về Npl và không có ràng buộc gì về chiều dài đường đi phải. Như vậy trong trường hợp xấu nhất các tác vụ chi phí đến O(N), khi cây nhị phân suy biến thành chuỗi mắc xích N nút về bên phải.

Tác vụ chính của skew heap cũng là phép trộn, trong đó việc hoán vị hai nhánh con luôn được làm. Tác vụ này có thể được hiện thực đệ quy hoặc không đệ quy. Kết quả việc hoán vị tại tất cả các nút sẽ là đường đi trái của heap cuối cùng sẽ chứa tất cả các nút trên hai đường đi phải của hai heap ban đầu theo đúng thứ tự tăng dần giữa chúng.



Hình 11.7. Kết quả trộn H1 và H2 theo cách của skew heap, hoán vị hai con trái và phải tại mọi nút

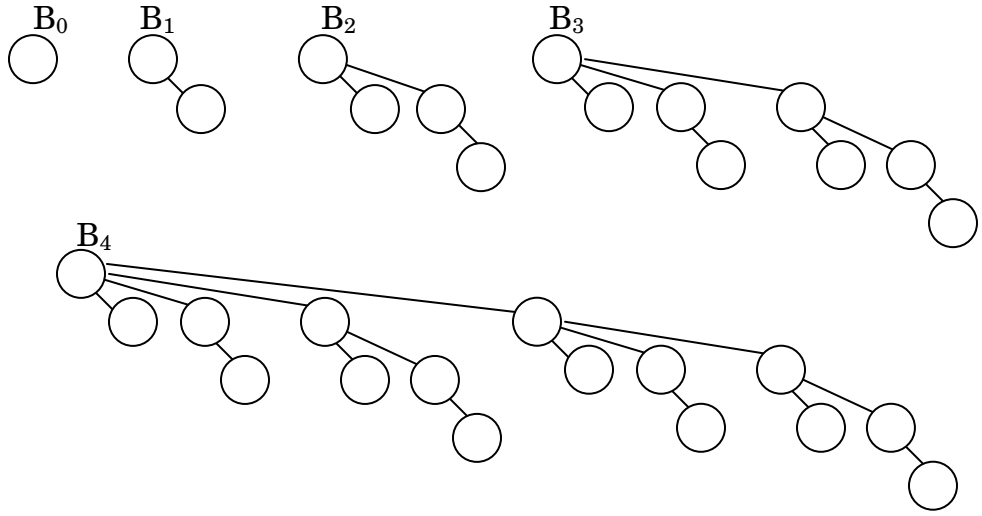
Như vậy tuy trường hợp xấu nhất của skew heap là O(log N), nhưng skew heap có ưu điểm là không phải chi phí để quản lý Npl tại mỗi nút và cũng không phải so sánh Npl để quyết định có hoán vị hai nút con tại mỗi nút hay không.

11.5.4. Hàng nhị thức (*Binomial Queue*)

Hàng nhị thức là một phương án hiện thực của hàng ưu tiên. Heap lệch trái và skew heap thực hiện O(log N) mỗi tác vụ đối với việc trộn, thêm và loại phần tử nhỏ nhất. Heap nhị phân thực hiện mỗi tác vụ thêm vào với thời gian trung bình là hằng số. Hàng nhị thức trong trường hợp xấu nhất thực hiện O(logN) cho mỗi tác vụ, nhưng việc thêm vào cũng có thời gian trung bình là hằng số.

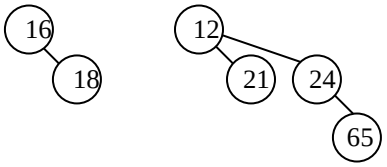
Khác với mọi hiện thực của hàng ưu tiên mà chúng ta đã xem xét, hàng nhị thức không phải là một cây có trật tự của heap, mà là một rừng các cây có trật tự của heap, trong đó không được phép có hai cây có cùng chiều cao. Theo quy ước, cây có chiều cao 0 là cây có 1 nút; cây có chiều cao k có được bằng cách nối một cây chiều cao k-1 vào nút gốc của một cây chiều cao k-1 khác. Hình 11.8 biểu diễn các cây có chiều cao lần lượt là 0, 1, 2, 3, 4. Từ hình vẽ chúng ta thấy, cây B_k bao gồm một nút gốc và các cây con B_0, B_1,..., B_{k-1}. Cây B_k có chính xác là 2^k nút, do đó

từ đây chúng ta sẽ gọi các cây này là **cây nhị thức**. Số nút ở mức d trong cây nhị thức là C^d_k. Nếu mọi cây nhị thức trong hàng nhị thức đều có trật tự của heap và không cho phép có nhiều hơn một cây nhị thức có cùng chiều cao thì hàng ưu tiên với kích thước bất kỳ đều có thể được biểu diễn bởi một hàng nhị thức như thế này. Chẳng hạn hàng ưu tiên có 13 nút sẽ gồm các cây nhị thức B_3, B_2, và B_0. Biểu diễn nhị phân của 13 chính là 1101, điều này hoàn toàn tương ứng với sự có mặt của B_3, B_2, và B_0, mà không có mặt của B_1.



Hình 11.8- Hình dạng các cây nhị thức với các chiều cao 0, 1, 2, 3, và 4 được quy định trong hàng nhị thức.

Hình 11.9 biểu diễn một hàng nhị thức có 6 nút.



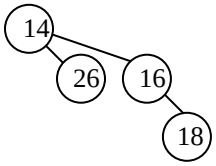
Hình 11.9- Hàng nhị thức có 6 nút.

Phần tử nhỏ nhất trong hàng nhị thức chính là một trong các nút gốc của các cây nhị thức có trong hàng. Do hàng nhị thức có nhiều nhất là log N cây nhị thức khác nhau, việc tìm kiếm chi phí `O(log N)`. Nếu chúng ta đánh dấu phần tử nhỏ nhất mỗi khi có sự thay đổi bởi một tác vụ nào thì chi phí này là `O(1)`.

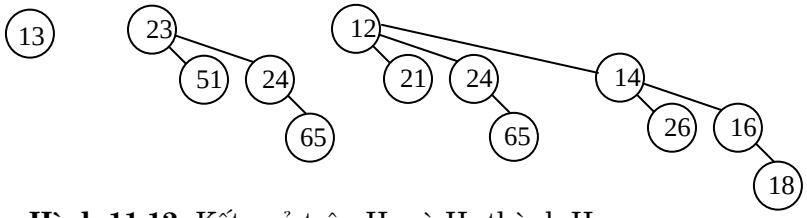
Phép trộn trong hàng nhị thức rất dễ dàng. Giả sử chúng ta cần trộn hai hàng nhị thức H_1 và H_2 trong hình 11.10. H_3 là hàng nhị thức kết quả. Trong H_1 và H_2 chỉ có một cây nhị thức B_0, vậy B_0 sẽ là một thành phần của H_3. Chúng ta kết hợp hai cây nhị thức B_1 trong H_1 và H_2 bằng cách cho cây nhị thức có gốc lớn hơn làm cây con của cây nhị thức còn lại. Với cây nhị thức B_2 vừa có được và hai cây nhị thức B_2 ban đầu trong H_1 và H_2, chúng ta để lại một cây trong H_3, và kết hợp

hai cây B_2 còn lại thành một cây B_3. Vì H_3 chưa có cây nhị thức B_3 nên B_3 cuối cùng này sẽ là thành phần của H_3.

Hình 11.10- Hai hàng nhị thức H_1 và H_2.



Hình 11.11 - Kết hợp hai cây nhị thức B_1 trong H_1 và H_2.

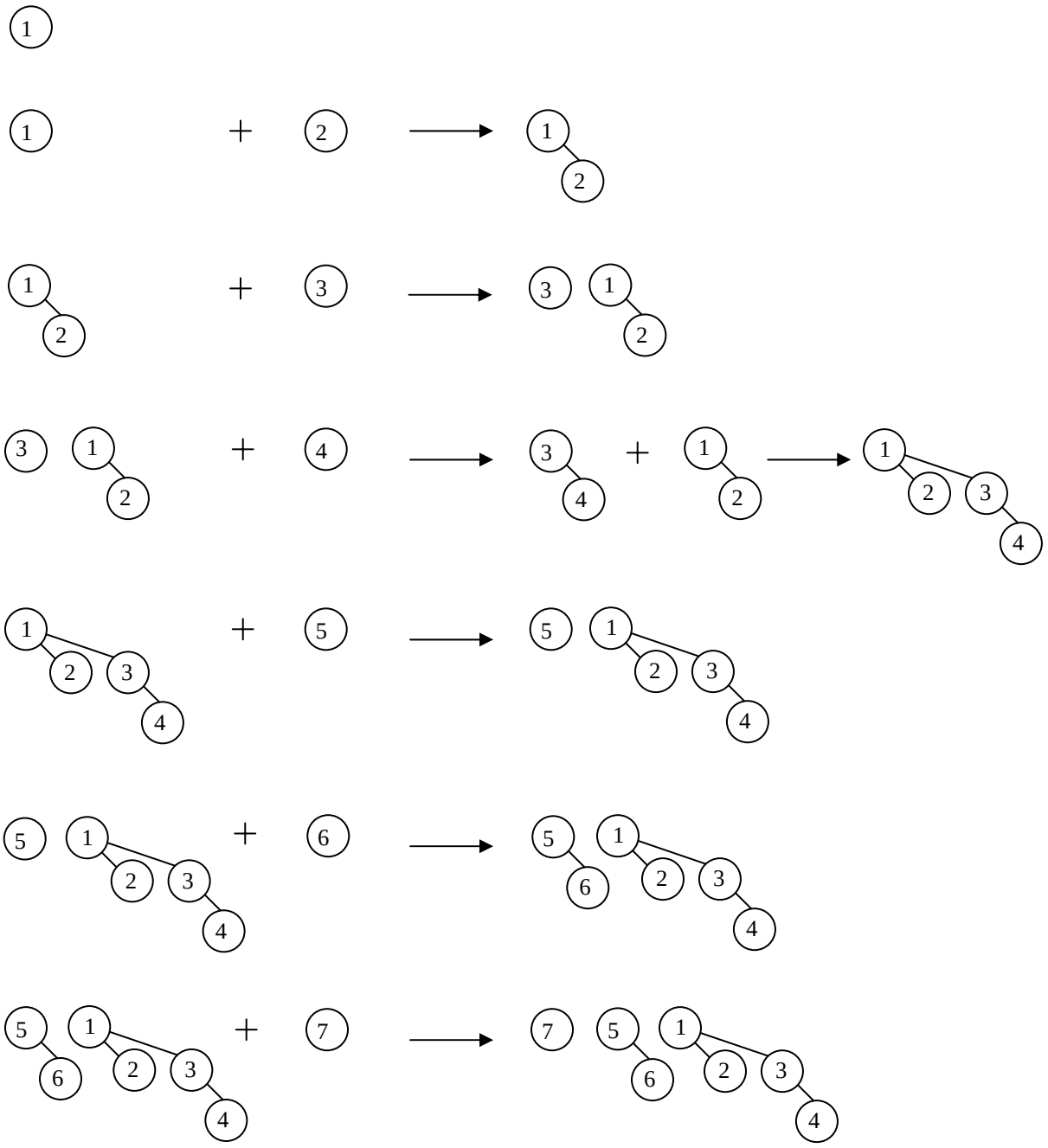


Hình 11.12- Kết quả trộn H_1 và H_2 thành H_3.

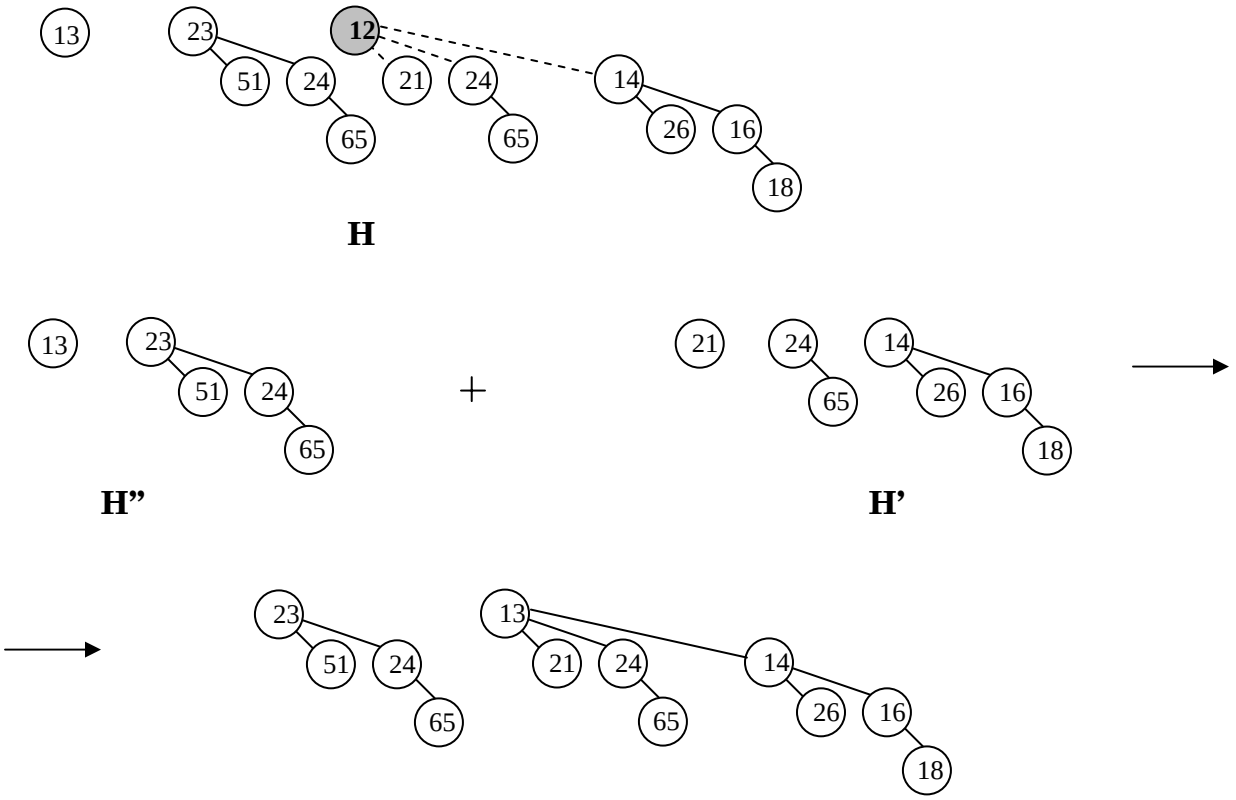
Việc kết hợp hai cây nhị thức tốn `O(1)`, với `O(log N)` cây nhị thức trong mỗi hàng nhị thức, việc trộn hai hàng nhị thức cũng tốn `O(log N)` trong trường hợp xấu nhất. Để tăng tính hiệu quả, chúng ta lưu các cây nhị thức trong mỗi hàng nhị thức theo thứ tự tăng dần của chiều cao các cây.

Việc thêm một phần tử mới vào hàng nhị thức tương tự như đối với heap lệch trái: trộn hàng nhị thức có duy nhất phần tử cần thêm với hàng nhị thức đã có.

Việc loại phần tử nhỏ nhất cũng đơn giản: tìm phần tử nhỏ nhất trong số các nút gốc của các cây nhị thức. Giả sử đó là cây B_k. Sau khi loại bỏ nút gốc của cây B_k, chúng ta còn lại các cây con của nó: B_0, B_1,... B_{k-1}. Gọi rừng gồm các cây con này là H', và H" là hàng nhị thức ban đầu không kể B_K. Việc cuối cùng cần làm là trộn H' và H". Chúng ta có thể tự kiểm tra rằng các phép thêm và loại này đều tốn `O(log N)` trong trường hợp xấu nhất.



Hình 11.13- Quá trình thêm các phần tử 1, 2,..., 7 vào hàng nhị thức.

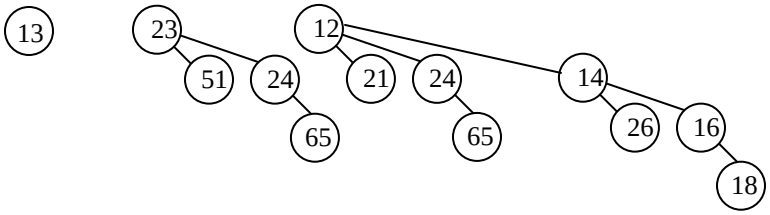


Hình 11.14- Quá trình loại phần tử nhỏ nhất trong hàng nhị thức H.

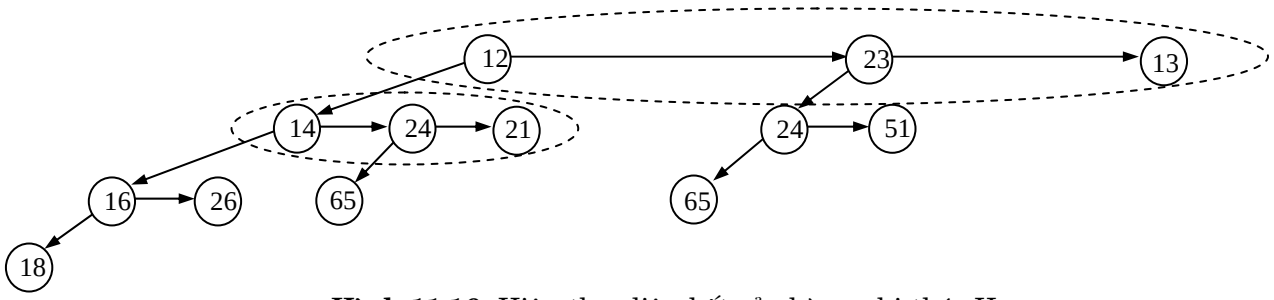
Hiện thực của hàng nhị thức

Việc tìm phần tử nhỏ nhất cần duyệt qua các gốc của các cây nhị thức trong hàng nhị thức (12, 23 và 13 trong hình 11.14). Chúng ta có thể dùng danh sách liên kết để chứa các nút gốc này. Danh sách sẽ có thứ tự theo chiều cao của các cây nhị thức để phục vụ cho phép trộn hai hàng nhị thức được dễ dàng.

Tương tự, các nút con của nút gốc của một cây nhị thức cũng được chứa trong một danh sách liên kết (14, 24 và 21 trong hình 11.14), để khi loại bỏ nút gốc (nút 12) thì phần còn lại cũng có cấu trúc tương tự như một hàng nhị thức mới, rất thuận lợi trong phép loại phần tử nhỏ nhất trong hàng nhị thức.



Hình 11.15- Hàng nhị thức H

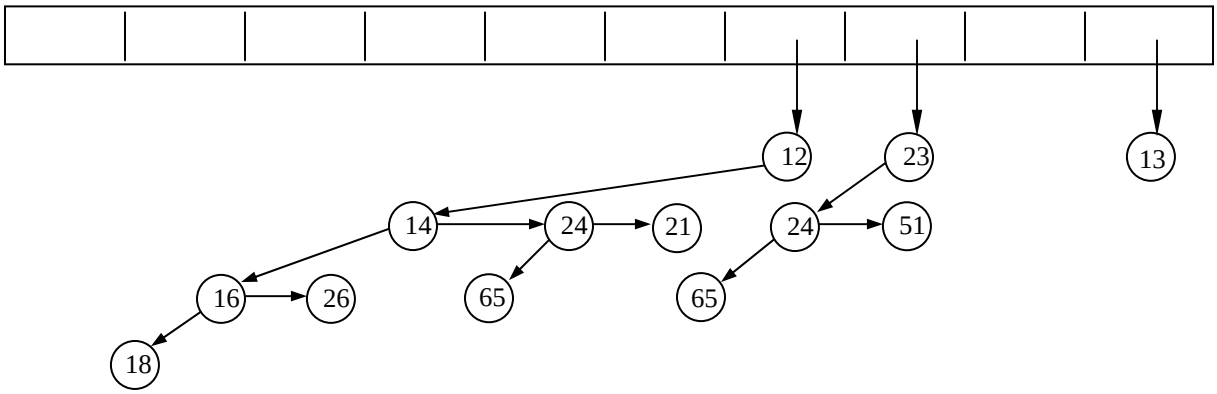


Hình 11.16- Hiện thực liên kết của hàng nhị thức H.

Trong hình vẽ trên chúng ta thấy hình *elipse* nét rời biểu diễn các danh sách liên kết các nút mà chúng ta thường phải duyệt qua. Hình *elipse* nét rời lớn chứa các gốc của các cây nhị thức trong hàng nhị thức, khi cần tìm phần tử nhỏ nhất trong hàng nhị thức chúng ta tìm trong danh sách này. Hình *elipse* nét rời nhỏ chứa các nút con của nút gốc trong một cây nhị thức.

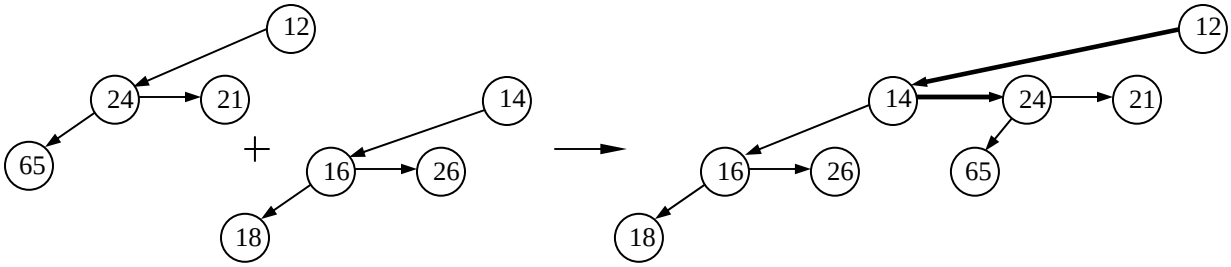
Trong quá trình hình thành hàng nhị thức trong một số thao tác dữ liệu, khi kết hợp hai cây nhị thức có cùng chiều cao (hình 11.18), chúng ta cần nối một trong hai cây thành cây con của cây còn lại, mà cây con mới này cũng chính là cây con có chiều cao lớn nhất so với các cây con đã có. Việc chèn cây con mới vào đầu của danh sách liên kết sẽ thuận tiện hơn, vì vậy chúng ta cho danh sách này có thứ tự giảm dần theo chiều cao của các cây con (hình 11.18).

Ngoài ra, mỗi nút trong cây nhị thức sẽ có một con trỏ cho phép truy xuất đến danh sách các con của nó. Tóm lại, hiện thực đơn giản và hiệu quả cho hàng nhị thức thật đơn giản như hình 11.18, đó là cấu trúc của một cây nhị phân, mỗi nút có con trỏ trái chỉ đến nút con đầu tiên của nó và con trỏ phải chỉ đến nút anh em của nó.



Hình 11.17- Gốc các cây nhị thức được chứa trong mảng liên tục.

Hình 11.17 là một phương án thay danh sách liên kết trên cùng bởi mảng liên tục. Chúng ta có thể dùng mảng liên tục cấp phát động để khắc phục nhược điểm do không biết trước chiều cao của cây nhị thức cao nhất trong hàng nhị thức. Việc dùng mảng liên tục cho phép tìm nhị phân phần tử nhỏ nhất, tuy nhiên sẽ là lãng phí lớn khi hàng nhị thức gồm quá ít cây nhị thức với nhiều chiều cao khác nhau.



Hình 11.18- Kết hợp hai cây nhị thức B_2 thành một cây nhị thức B_3.

Dưới đây là phần mã giả cho các khai báo cấu trúc của cây nhị thức và hàng nhị thức. Các tác vụ kết hợp hai cây nhị thức, trộn hai hàng nhị thức, và loại phần tử nhỏ nhất trong hàng nhị thức cũng được trình bày bằng mã giả.

```
//Phần mã giả cho khai báo và một số tác vụ cho hàng nhị thức.

struct Binomial_Node
   DataType          data
   Binomial_Node* leftChild
   Binomial_Node* nextSibling
end struct

struct Binomial_Tree
   Binomial_Tree CombineTrees(ref Binomial_Tree T1, ref Binomial_Tree T2)
   Binomial_Node* root
   int notEmpty() // Trả về 1 nếu cây không rỗng, ngược lại trả về 0.
end struct

class Binomial_Queue
   public:
      Binomial_Queue(Binomial_Node* p, int k)// k là số nút trong hàng nhị thức.
      int empty
      Binomial_Queue merge(ref Binomial_Queue H1, ref Binomial_Queue H2)
   protected:
      int count // Tổng số nút trong tất cả các cây nhị thức.
      Binomial_Tree Trees[MAX]
end class

Binomial_Tree Binomial_Tree::CombineTrees(ref Binomial_Tree T1,
                                          ref Binomial_Tree T2);
/*
pre: T1 và T2 khác rỗng.
post: T1 chứa kết quả kết hợp hai cây T1 và T2, T2 rỗng. Trả về cây T1.
*/
{
   1. if (T1.root->data > T2.root->data)
      1. Tráo T1 và T2 cho nhau
   2. T2.root->nextSibling = T1.root->leftChild // Chèn cây T2 vào đầu danh
                                                // sách các cây con của gốc của T1
   3. T1.root->leftChild = T2.root // Cập nhật nút con trái cho nút gốc của T1
   4. T2.root  = NULL
   5. return T1
}
```

```
Binomial_Queue Binomial_Queue::merge(ref Binomial_Queue H1,
                                     ref Binomial_Queue H2)
/*
post: H1 chứa kết quả trộn hai hàng nhị thức H1 và H2, H2 rỗng. Trả về hàng H1.
uses: hàm Binomial_Tree::notEmpty() trả về 1 nếu cây không rỗng, ngược lại trả về 0.
*/
{
   Binomial_Tree T1, T2, carry
   int i = 0
   1. H1.count = H1.count + H2.count
   2. loop (H2 chưa rỗng hoặc carry không rỗng)
      1. T1 = H1.Trees[i]
      2. T2 = H2.Trees[i]
      3. case (T1.notEmpty() + 2*T2.notEmpty() + 4*carry.notEmpty())
         1. case 0: // Không có cây nào
         2. case 1: // Chỉ có cây T1
            1. break
         3. case 2: // Chỉ có cây T2
            1. H1.Trees[i] = T2
            2. H2.Trees[i].root = NULL
            3. break
         4. case 4: // Chỉ có cây carry
            1. H1.Trees[i] = carry
            2. carry.root = NULL
            3. break
         5. case 3: // Có cây T1 và T2
            1. carry = combineTrees(T1, T2)
            2. H1.Trees[i].root = NULL
            3. H2.Trees[i].root = NULL
            4. break
         6. case 5: // Có cây T1 và carry
            1. carry = combineTrees(T1, carry)
            2. H1.Trees[i].root = NULL
            3. break
         7. case 6:// Có cây T2 và carry
            1. carry = combineTrees(T2, carry)
            2. H2.Trees[i].root = NULL
            3. break
         8. case 7:// Có cả 3 cây T1, T2, và carry
            1. H1.Trees[i] = carry
            2. carry = combineTrees(T1, T2)
            3. H2.Trees[i].root = NULL
            4. break
      4. endcase
      5. i = i + 1
   3. endloop
   4. return H1
}
```

```
Binomial_Queue::Binomial_Queue(Binomial_Node* p, int k)
/*
pre: p là địa chỉ nút đầu tiên của một cấu trúc liên kết các cây nhị thức B_k,B_k-1,...,B_0.
post: Hàng nhị thức mới được tạo thành từ các cây này.
*/
{
   1. count = (1 << (k+1)) –1
   2. loop (k >= 0)
      1. Trees[k] = p
      2. p = p->nextSibling
      3. Trees[k]->nextSibling = NULL// Cắt rời các cây con để đưa vào mảng liên
                                        // tục các cây nhị thức cho hàng nhị thức mới.
      4. k = k – 1
   3. endloop
}
```

Tóm lại, trong chương này chúng ta đã xem xét một số cách hiện thực của hàng ưu tiên. Heap nhị phân vừa đơn giản vừa hiệu quả vì không sử dụng con trỏ, nó chỉ hơi tốn nhiều vùng nhớ chưa sử dụng đến mà thôi. Chúng ta đã nghiên cứu thêm tác vụ trộn và bổ sung thêm ba phương án khác của hàng ưu tiên. Heap lệch trái là một ví dụ khá hay về giải thuật đệ quy. Skew heap giống heap lệch trái nhưng đơn giản hơn, với hy vọng rằng các ứng dụng có tính ngẫu nhiên thì các trường hợp xấu nhất sẽ không thường xuyên xảy ra. Cuối cùng hàng nhị thức cho thấy một ý tưởng đơn giản mà lại giới hạn được chi phí cho giải thuật khá tốt.

Chương 12 – BẢNG VÀ TRUY XUẤT THÔNG TIN

Chương này tiếp tục nghiên cứu về cách tìm kiếm truy xuất thông tin đã đề cập ở chương 7, nhưng tập trung vào các bảng thay vì các danh sách. Chúng ta bắt đầu từ các bảng hình chữ nhật thông thường, sau đó là các dạng bảng khác và cuối cùng là bảng băm.

12.1. Dẫn nhập: phá vỡ rào cản lgn

Trong chương 7 chúng ta đã thấy rằng, bằng cách so sánh khóa, trung bình việc tìm kiếm trong n phần tử không thể có ít hơn lg n lần so sánh. Nhưng kết quả này chỉ nói đến việc tìm kiếm bằng cách so sánh các khóa. Bằng một vài phương pháp khác, chúng ta có thể tổ chức các dữ liệu sao cho vị trí của một phần tử có thể được xác định nhanh hơn.

Thật vậy, chúng ta thường làm thế. Nếu chúng ta có 500 phần tử khác nhau có các khóa từ 0 đến 499, thì chúng ta sẽ không bao giờ nghĩ đến việc tìm kiếm tuần tự hoặc tìm kiếm nhị phân để xác định vị trí một phần tử. Đơn giản chúng ta chỉ lưu các phần tử này trong một mảng kích thước là 500, và sử dụng chỉ số n để xác định phần tử có khóa là n bằng cách tra cứu bình thường đối với một bảng.

Việc tra cứu trong bảng cũng như việc tìm kiếm có chung một mục đích, đó là truy xuất thông tin. Chúng ta bắt đầu từ một khóa và mong muốn tìm một phần tử chứa khóa này

Trong chương này chúng ta tìm hiểu cách hiện thực và truy xuất các bảng trong vùng nhớ liên tục, bắt đầu từ các bảng hình chữ nhật thông thường, sau đó đến các bảng có một số vị trí hạn chế như các bảng tam giác, bảng lồi lõm. Sau đó chúng ta chuyển sang các vấn đề mang tính tổng quát hơn, với mục đích tìm hiểu cách sử dụng các mảng truy xuất và các bảng băm để truy xuất thông tin.

Chúng ta sẽ thấy rằng, tuỳ theo hình dạng của bảng, chúng ta cần có một số bước để truy xuất một phần tử, tuy vậy, thời gian cần thiết vẫn là 0(1) - có nghĩa là, thời gian có giới hạn là một hằng số và độc lập với kích thước của bảng- và do đó việc tra cứu bảng có thể đạt hiệu quả hơn nhiều so với bất kỳ phương pháp tìm kiếm nào.

Các phần tử của các bảng mà chúng ta xem xét được đánh chỉ số bằng một mảng các số nguyên, tương tự cách đánh chỉ số của mảng. Chúng ta sẽ hiện thực các bảng được định nghĩa trừu tượng bằng các mảng. Để phân biệt giữa khái niệm trừu tượng và các hiện thực của nó, chúng ta có một quy ước sau:

Chỉ số xác định một phần tử của một bảng định nghĩa trừu tượng được bao bởi cặp dấu ngoặc đơn, còn chỉ số của một phần tử trong mảng được bao bởi cặp dấu ngoặc vuông.

Ví dụ, T(1,2,3) là phần tử của bảng T được đánh chỉ số bởi dãy số 1, 2, 3, và A[1][2][3] tương ứng phần tử với chỉ số trong mảng A của C++.

12.2. Các bảng chữ nhật

Do tầm quan trọng của các bảng chữ nhật, hầu hết các ngôn ngữ lập trình cấp cao đều cung cấp mảng hai chiều để chứa và truy xuất chúng, và nói chung người lập trình không cần phải bận tâm đến cách hiện thực chi tiết của nó. Tuy nhiên, bộ nhớ máy tính thường có tổ chức cơ bản là một mảng liên tục (như một mảng tuyến tính có phần tử này nằm kế phần tử kia), đối với mỗi truy xuất đến bảng chữ nhật, máy cần phải có một số tính toán để chuyển đổi một vị trí trong hình chữ nhật sang một vị trí trong mảng tuyến tính. Chúng ta hãy xem xét điều này một cách chi tiết hơn.

12.2.1. Thứ tự ưu tiên hàng và thứ tự ưu tiên cột

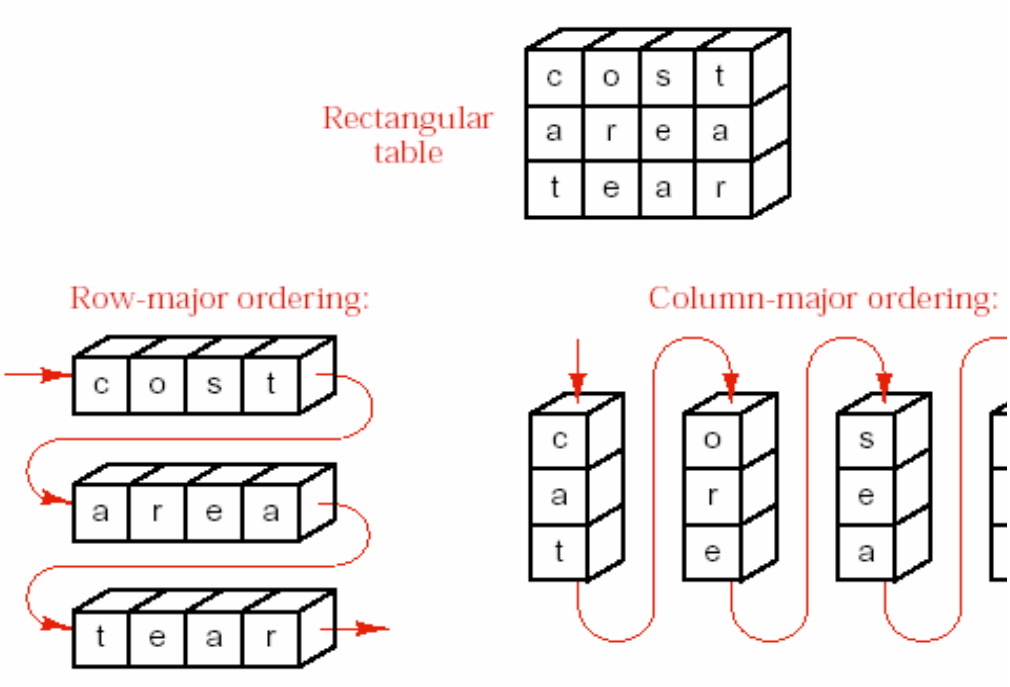
Cách tự nhiên để đọc một bảng chữ nhật là đọc các phần tử ở hàng thứ nhất trước, từ trái sang phải, sau đó đến các phần tử hàng thứ hai, và cứ thế tiếp tục cho đến khi hàng cuối đã được đọc xong. Đây cũng là thứ tự mà đa số các trình biên dịch lưu trữ bảng chữ nhật, và được gọi là thứ tự ưu tiên hàng (*row-major ordering*). Chẳng hạn, một bảng trừu tượng có hàng được đánh số là từ 1 đến 2, và cột được đánh số từ 5 đến 7, thì thứ tự của các phần tử theo thứ tự ưu tiên hàng như sau:

(1,5) (1,6) (1,7) (2,5) (2,6) (2,7)

Đây cũng là thứ tự được dùng trong C++ và hầu hết các ngôn ngữ lập trình cấp cao để lưu trữ các phần tử của một mảng hai chiều. FORTRAN chuẩn lại sử dụng thứ tự ưu tiên cột, trong đó các phần tử của cột thứ nhất được lưu trước, rồi đến cột thứ hai,v.v...Ví dụ thứ tự ưu tiên cột như sau:

(1,5) (2,5) (1,6) (2,6) (1,7) (2,7)

Hình 12.1 minh họa các thứ tự ưu tiên cho một bảng có 3 hàng và 4 cột.



Hình 12.1 – Biểu diễn nối tiếp cho mảng chữ nhật

12.2.2. Đánh chỉ số cho bảng chữ nhật

Một cách tổng quát, trình biên dịch có thể bắt đầu từ chỉ số (i,j) để tính vị trí trong một mảng nối tiếp mà một phần tử tương ứng trong bảng được lưu trữ. Chúng ta sẽ đưa ra công thức tính toán sau đây. Để đơn giản chúng ta chỉ sử dụng thứ tự ưu tiên hàng cùng với giả thiết là hàng được đánh số từ 0 đến m-1, và cột từ 0 đến n-1. Trường hợp các hàng và các cột được đánh số không phải từ 0 được xem như bài tập. Số phần tử của bảng sẽ là mn, và đó cũng là số phần tử trong hiện thực liên tục trong mảng. Chúng ta đánh số các phần tử trong mảng từ 0 đến mn –1. Để có công thức tính vị trí của phần tử (i,j) trong mảng, trước hết chúng ta quan sát một vài trường hợp đặc biệt. Phần tử (0,0) nằm tại vị trí 0, các phần tử thuộc hàng đầu tiên trong bảng rất dễ tìm thấy: (0,j) nằm tại vị trí j. Phần tử đầu của hàng thứ hai (1,0) nằm ngay sau phần tử (0,n-1), đó là vị trí n. Tiếp theo, chúng ta thấy phần tử (1,j) nằm tại vị trí n+j. Các phần tử của hàng kế tiếp cũng sẽ nằm sau số phần tử của hai hàng trước đó (2n phần tử). Do đó phần tử (2,j) nằm tại vị trí 2n+j. Một cách tổng quát, các phần tử thuộc hàng i có n i phần tử phía trước, nên công thức chung là:

Phần tử (i,j) trong bảng chữ nhật nằm tại vị trí n i + j trong mảng nối tiếp.

Công thức này cho biết vị trí trong mảng nối tiếp mà một phần tử trong bảng chữ nhật được lưu trữ, và được gọi là hàm chỉ số (*index function*).

12.2.3. Biến thể: mảng truy xuất

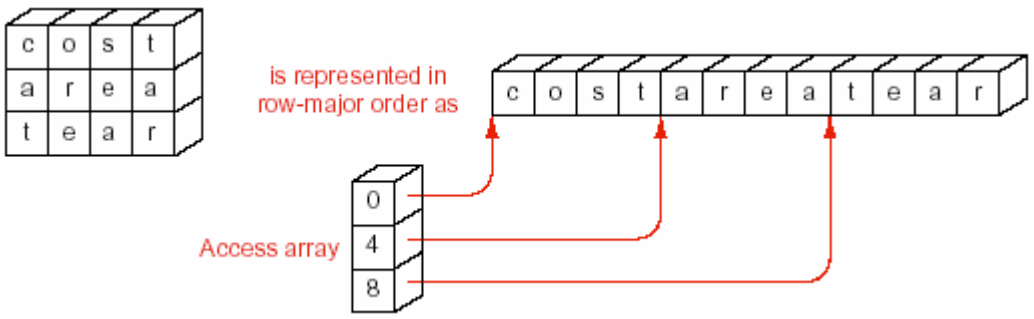
Việc tính toán cho các hàm chỉ số của các bảng chữ nhật thật ra không khó lắm, các trình biên dịch của hầu hết các ngôn ngữ cấp cao sẽ dịch hàm này sang ngôn ngữ máy thành một số bước tính toán cần thiết. Tuy nhiên, trên các máy tính nhỏ, phép nhân thường thực hiện rất chậm, một phương pháp khác có thể được sử dụng để tránh phép nhân.

Phương pháp này lưu một mảng phụ trợ chứa một phần của bảng nhân với thừa số là n:

0, n, 2n, 3n, ..., (m-1)n.

Lưu ý rằng mảng này nhỏ hơn bảng chữ nhật rất nhiều, nên nó có thể được lưu thường trực trong bộ nhớ. Các phần tử của nó chỉ phải tính một lần (và chúng có thể được tính chỉ bằng phép cộng). Khi gặp một yêu cầu tham chiếu đến bảng chữ nhật, trình biên dịch có thể tìm vị trí của phần tử (i,j) bằng cách lấy phần tử thứ i trong mảng phụ trợ cộng thêm j để đến vị trí cần có.

Mảng phụ trợ này cung cấp cho chúng ta một ví dụ đầu tiên về một mảng truy xuất (*access mảng*) (Hình 12.2). Nói chung, một mảng truy xuất là một mảng phụ trợ được sử dụng để tìm một dữ liệu được lưu trữ đâu đó. Mảng truy xuất có khi còn được gọi là *vector* truy xuất (*access vector*).



Hình 12.2 – Mảng truy xuất cho bảng chữ nhật

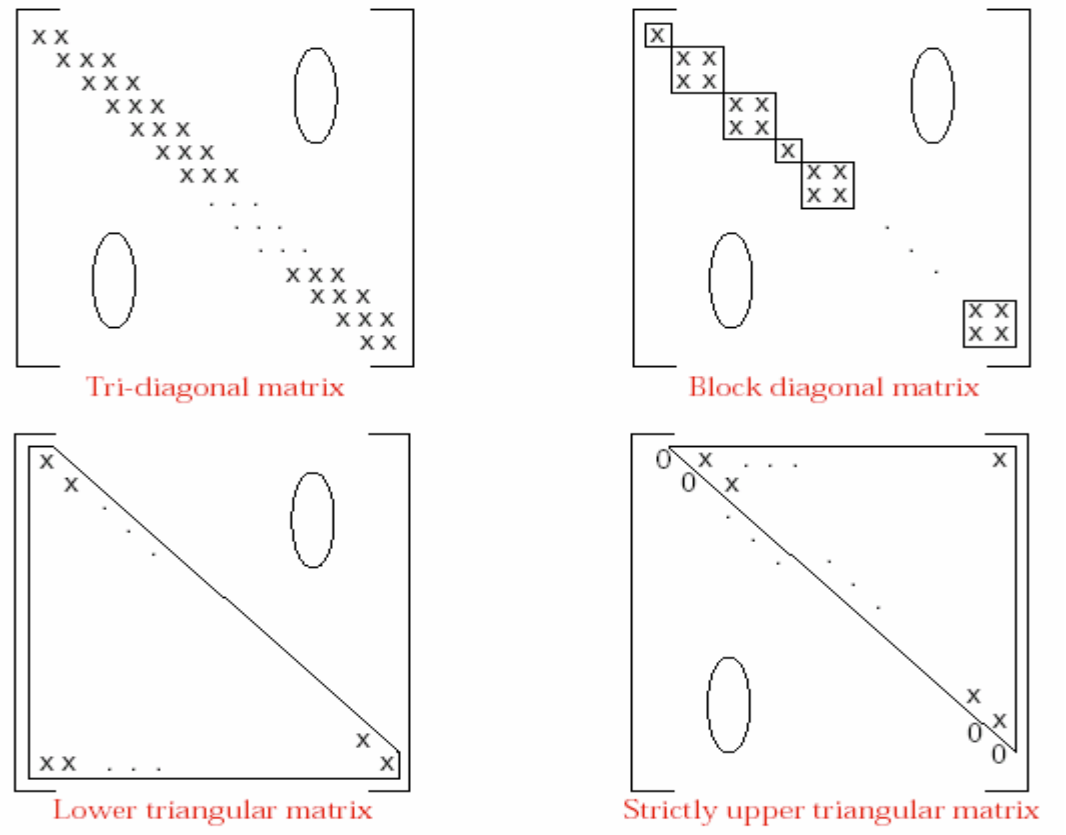
12.3. Các bảng với nhiều hình dạng khác nhau

Thông tin thường lưu trong một bảng chữ nhật có thể không cần đến mọi vị trí trong hình chữ nhật đó. Nếu chúng ta định nghĩa ma trận là một bảng chữ nhật gồm các con số, thì thường là một vài vị trí trong ma trận đó mang trị 0. Một vài ví dụ như thế được minh họa trong hình 12.3. Ngay cả khi các phần tử trong một bảng không phải là những con số, các vị trí được sử dụng thực sự cũng có thể không phải là tất cả hình chữ nhật, và như vậy có thể có cách hiện thực khác hay hơn thay vì sử dụng một bảng chữ nhật với nhiều chỗ trống. Trong phần này, chúng ta tìm hiểu các cách hiện thực các bảng với nhiều hình dạng khác nhau,

những cách này sẽ không đòi hỏi vùng nhớ cho những phần tử vắng mặt như trong bảng chữ nhật.

12.3.1. Các bảng tam giác

Chúng ta hãy xem xét cách biểu diễn bảng tam giác dưới như trong hình vẽ 12.3. Một bảng như vậy chỉ cần các chỉ số (i,j) với i≥j. Chúng ta có thể hiện thực một bảng tam giác trong một mảng liên tục bằng cách trượt mỗi hàng ra sau hàng nằm ngay trên nó, như cách biểu diễn ở hình 12.4.

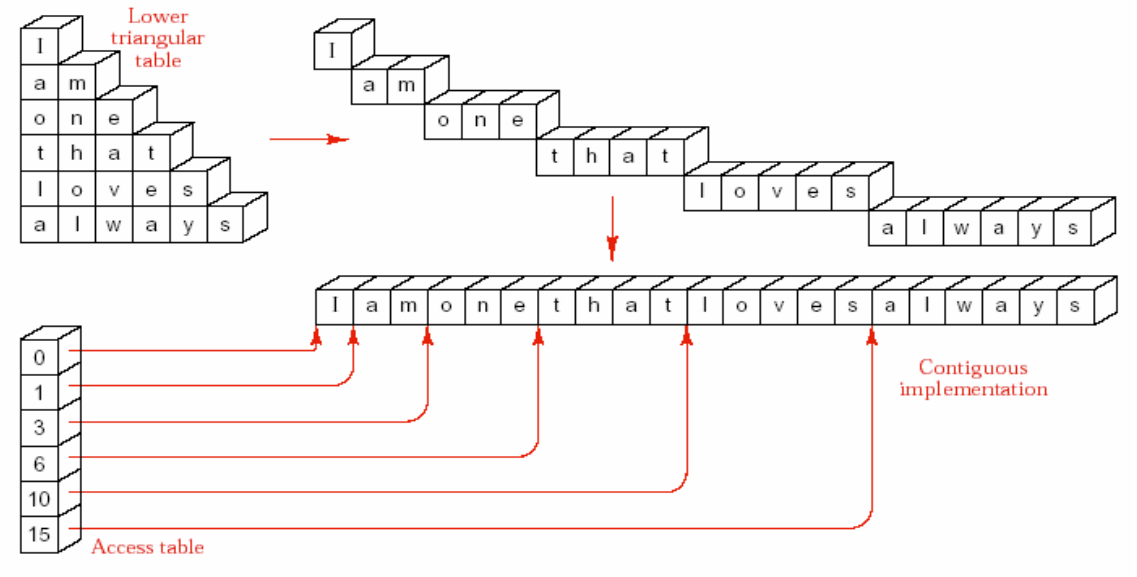


Hình 12.3 – Các bảng với nhiều dạng khác nhau.

Để xây dựng hàm chỉ số mô tả cách ánh xạ này, chúng ta cũng giả sử rằng các hàng và các cột đều được đánh số bắt đầu từ 0. Để tìm vị trí của phần tử (i,j) trong mảng liên tục chúng ta cần tìm vị trí bắt đầu của hàng i, sau đó để tìm cột j chúng ta chỉ việc cộng thêm j vào điểm bắt đầu của hàng i. Nếu các phần tử của mảng liên tục cũng được đánh số bắt đầu từ 0, thì chỉ số của điểm bắt đầu của hàng thứ i cũng chính là số phần tử nằm ở các hàng trên hàng i. Rõ ràng là trên hàng thứ 0 có 0 phần tử, và chỉ có một phần tử của hàng 0 là xuất hiện trước hàng 1. Đối với hàng 2, có 1 + 2 = 3 phần tử đi trước, và trong trường hợp tổng quát chúng ta thấy số phần tử có trước hàng i chính xác là:

$$1 + 2 + \ldots + i = \frac{1}{2} i(i + 1).$$

Vậy phần tử (i,j) trong bảng tam giác tương ứng phần tử $\frac{1}{2}$ i(i + 1) + j của mảng liên tục.

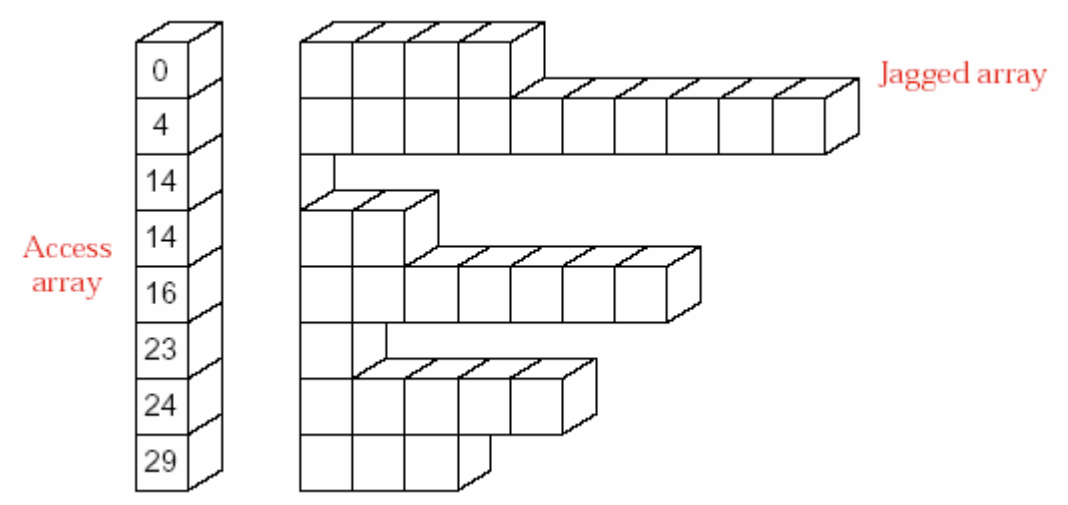


Hình 12.4 – Hiện thực liên tục của bảng tam giác.

Cũng như chúng ta đã làm cho các bảng chữ nhật, chúng ta cũng tránh mọi phép nhân và chia bằng cách tạo một mảng truy xuất chứa các phần tử tương ứng với các chỉ số của các hàng trong bảng tam giác. Vị trí i trong mảng truy xuất mang trị $\frac{1}{2}$i (i + 1). Mảng truy xuất được tính toán chỉ một lần khi bắt đầu chương trình, và được sử dụng lặp lại cho mỗi truy xuất đến bảng tam giác. Chú ý rằng ngay cả việc tính toán ban đầu cũng không cần đến phép nhân hoặc chia mà chí có phép cộng theo thứ tự sau mà thôi:

0, 1, 1+2, (1 + 2) + 3, . . .

12.3.2. Các bảng lồi lõm



Hình 12.5 – Mảng truy xuất cho bảng lồi lõm.

Trong cả hai ví dụ đã đề cập trước chúng ta đã xem xét một bảng được tạo từ các hàng của nó. Trong các bảng chữ nhật thông thường, tất cả các hàng đều có cùng chiều dài; trong bảng tam giác, chiều dài mỗi hàng có thể được tính dựa vào một công thức đơn giản. Bây giờ chúng ta hãy xem xét đến trường hợp của các bảng lồi lõm tựa như hình 12.5, không có một mối quan hệ có thể đoán trước nào giữa vị trí của một hàng và chiều dài của nó.

Một điều hiển nhiên được nhìn thấy từ sơ đồ rằng, tuy chúng ta không thể xây dựng một hàm thứ tự nào để ánh xạ một bảng lồi lõm sang vùng nhớ liên tục, nhưng việc sử dụng một mảng truy xuất cũng dễ dàng như các ví dụ trước, và các phần tử của bảng lồi lõm có thể được truy xuất một cách nhanh chóng. Để tạo mảng truy xuất, chúng ta phải xây dựng bảng lồi lõm theo thứ tự vốn có của nó, bắt đầu từ hàng đầu tiên. Phần tử 0 của mảng truy xuất, cũng như trước kia, là bắt đầu của mảng liên tục. Sau khi mỗi hàng của bảng lồi lõm được xây dựng xong, chỉ số của vị trí đầu tiên chưa được sử dụng tới của vùng nhớ liên tục chính là trị của phần tử kế tiếp trong mảng truy xuất và được sử dụng để bắt đầu xây dựng hàng kế của bảng lồi lõm.

12.3.3. Các bảng chuyển đổi

Tiếp theo, chúng ta hãy xem xét một ví dụ minh họa việc sử dụng nhiều mảng truy xuất để tham chiếu cùng lúc đến một bảng các phần tử qua một vài khóa khác nhau.

Chúng ta xem xét nhiệm vụ của một công ty điện thoại trong việc truy xuất đến các phần tử về các khách hàng của họ. Để in danh mục điện thoại, các phần tử cần sắp thứ tự tên khách hàng theo *alphabet*. Tuy nhiên, để xử lý các cuộc gọi đường dài, các phần tử lại cần có thứ tự theo số điện thoại. Ngoài ra, để tiến hành bảo trì định kỳ, danh sách các khách hàng sắp thứ tự theo địa chỉ sẽ có ích cho các nhân viên bảo trì. Như vậy, công ty điện thoại cần phải lưu cả ba, hoặc nhiều hơn, danh sách các khách hàng theo các thứ tự khác nhau. Bằng cách này, không những tốn kém nhiều vùng lưu trữ mà còn có khả năng thông tin bị sai lệch do không được cập nhật đồng thời.

Chúng ta có thể tránh được việc phải lưu nhiều lần cùng một tập các phần tử bằng cách sử dụng các mảng truy xuất, và chúng ta có thể tìm các phần tử theo bất kỳ một khóa nào một cách nhanh chóng chẳng khác gì chúng đã được sắp thứ tự theo khóa đó. Chúng ta sẽ tạo một mảng truy xuất cho tên các khách hàng. Phần tử đầu tiên của mảng này chứa vị trí của khách hàng đứng đầu danh sách theo alphabet. Phần tử thứ hai chứa vị trí khách hàng thứ hai, và cứ thế. Trong mảng truy xuất thứ hai, phần tử đầu tiên chứa vị trí của khách hàng có số điện thoại nhỏ nhất. Tương tự, mảng truy xuất thứ ba có các phần tử chứa vị trí của các khách hàng theo thứ tự địa chỉ của họ. (Hình 12.6)

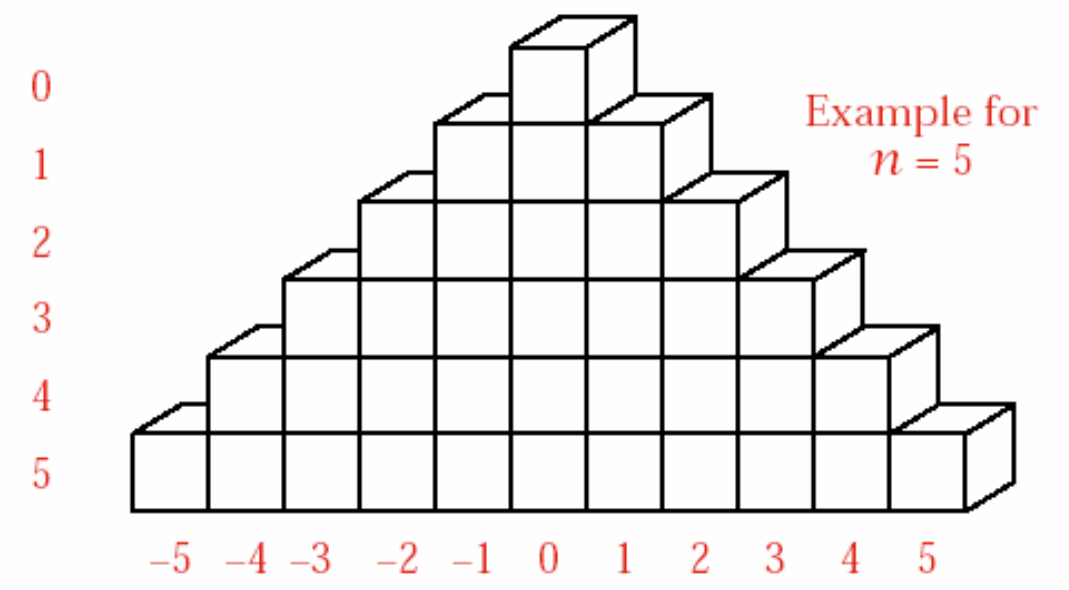
Index	*Name*	*Address*	*Phone*
1	Hill, Thomas M.	High Towers #317	2829478
2	Baker, John S.	17 King Street	2884285
3	Roberts, L. B.	53 Ash Street	4372296
4	King, Barbara	High Towers #802	2863386
5	Hill, Thomas M.	39 King Street	2495723
6	Byers, Carolyn	118 Maple Street	4394231
7	Moody, C. L.	High Towers #210	2822214

Access Tables

Name	*Address*	*Phone*
2	3	5
6	7	7
1	1	1
5	4	4
4	2	2
7	5	3
3	6	6

Hình 12.6 – Mảng truy xuất cho nhiều khóa: bảng chuyển đổi

Chúng ta lưu ý rằng trong phương pháp này các thành phần dữ liệu được xem như là khóa đều được xử lý cùng một cách. Không có lý do gì buộc các phần tử phải có thứ tự vật lý ưu tiên theo khóa này mà không theo khóa khác. Các phần tử có thể được lưu trữ theo một thứ tự tùy ý, có thể nói đó là thứ tự mà chúng được nhập vào hệ thống. Cũng không có sự khác nhau giữa việc các phần tử được lưu trong một danh sách liên tục là mảng (mà các phần tử của các mảng truy xuất chứa các chỉ số của mảng này) hay các phần tử đang thuộc một danh sách liên kết (các phần tử của các mảng truy xuất chứa các địa chỉ đến từng phần tử riêng). Trong mọi trường hợp, chính các mảng truy xuất được sử dụng để truy xuất thông tin, và, cũng giống như các mảng liên tục thông thường, chúng có thể được sử dụng trong việc tra cứu các bảng, hoặc tìm kiếm nhị phân, hoặc với bất kỳ mục đích nào khác thích hợp với cách hiện thực liên tục.



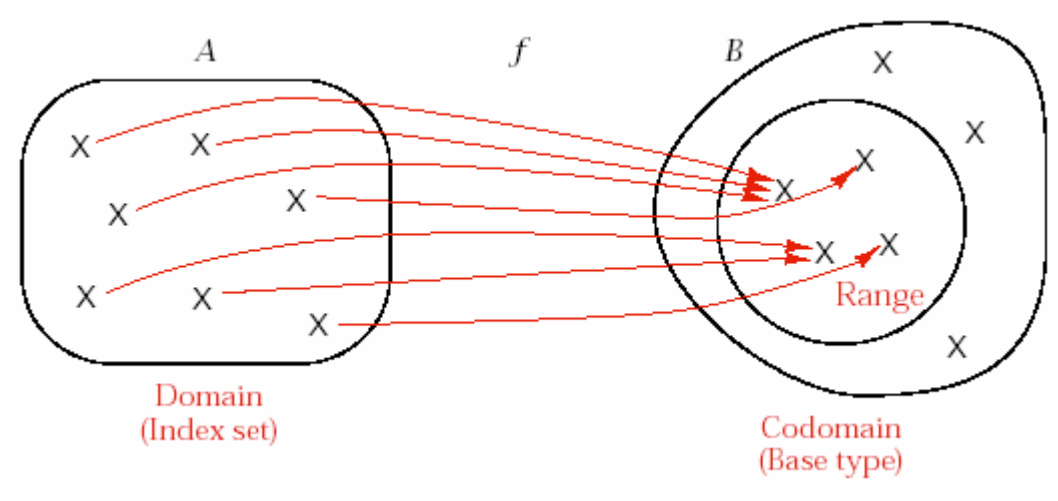
Hình 12.7 – Ví dụ về bảng tam giác đối xứng qua 0.

12.4. Bảng: Một kiểu dữ liệu trừu tượng mới

Từ đầu chương này chúng ta đã biết đến một số hàm chỉ số được dùng để tìm kiếm các phần tử trong các bảng, sau đó chúng ta cũng đã gặp các mảng truy xuất là các mảng được dùng với cùng một mục đích như các hàm chỉ số. Có một sự giống nhau rất lớn giữa các hàm với việc tra cứu bảng: với một hàm, chúng ta bắt đầu bằng một thông số để tính một giá trị tương ứng; với một bảng, chúng ta bắt đầu bằng một chỉ số để truy xuất một giá trị dữ liệu tương ứng được lưu trong bảng. Chúng ta hãy sử dụng sự tương tự này để xây dựng một định nghĩa hình thức cho thuật ngữ **bảng**.

12.4.1. Các hàm

Trong toán học, một hàm được định nghĩa dựa trên hai tập hợp và sự tương ứng từ các phần tử của tập thứ nhất đến các phần tử của tập thứ hai. Nếu f là một hàm từ tập A sang tập B, thì f gán cho mỗi phần tử của A một phần tử duy nhất của B. Tập A được gọi là *domain* của f, còn tập B được gọi là *codomain* của f. Tập con của B chỉ chứa các phần tử là các trị của f được gọi là *range* của f. Định nghĩa này được minh họa trong hình 12.8.



Hình 12.8 – *Domain*, *codomain* và *range* của một hàm

Việc truy xuất bảng bắt đầu bằng một chỉ số và bảng được sử dụng để tra cứu một trị tương ứng. Đối với một bảng chúng ta gọi *domain* là tập chỉ số (*index set*), và *codomain* là kiểu cơ sở (*base type*) hoặc kiểu trị (*value type*). Lấy ví dụ, chúng ta có một khai báo mảng như sau:

```
double array[n];
```

thì tập chỉ số là tập các số nguyên từ 0 đến n-1, và kiểu cơ sở là tập tất cả các số thực. Lấy ví dụ thứ hai, chúng ta hãy xét một bảng tam giác có m hàng, mỗi phần tử có kiểu `item`. Kiểu cơ sở sẽ là kiểu `item` và tập chỉ số là tập các cặp số nguyên

$$\{(i,j) \mid 0 \le j \le i < m\}$$

12.4.2. Một kiểu dữ liệu trừu tượng

Chúng ta đang đi đến một định nghĩa cho bảng như một kiểu dữ liệu trừu tượng mới, đồng thời trong các chương trước chúng ta đã biết rằng để hoàn tất một định nghĩa cho một cấu trúc dữ liệu, chúng ta cần phải đặc tả các tác vụ đi kèm.

Định nghĩa: Một bảng với tập chỉ số I và kiểu cơ sở T là một hàm từ I đến T kèm các tác vụ sau:
1. *Access* (truy xuất bảng): Xác định trị của hàm theo bất kỳ một chỉ số trong I.
2. *Assignment* (ghi bảng): Sửa đổi hàm bằng cách thay đổi trị của nó tại một chỉ số nào đó trong I thành một trị mới được chỉ ra trong phép gán.

Hai tác vụ này là tất cả những gì được cung cấp bởi các mảng trong C++ hoặc một vài ngôn ngữ khác, nhưng đó không phải là lý do để có thể ngăn cản chúng ta thêm một số tác vụ khác cho một bảng trừu tượng. Nếu so sánh với định nghĩa

của một danh sách (*list*), chúng ta đã có các tác vụ như thêm phần tử, xóa phần tử cũng như truy xuất hoặc cập nhật lại. Vậy chúng ta có thể làm tương tự đối với bảng.

Các tác vụ bổ sung cho bảng:
1. *Creation* (Tạo): Tạo một hàm từ I vào T.
2. *Clearing* (Dọn dẹp): Loại bỏ mọi phần tử trong tập chỉ số I, *domain* sẽ là một tập rỗng.
3. *Insertion* (Thêm): Thêm một phần tử x vào tập chỉ số I và xác định một trị tương ứng của hàm tại x.
4. *Deletion* (Xóa): Loại bỏ một phần tử x trong tập chỉ số I và hạn chế chỉ cho hàm xác định trên tập chỉ số còn lại.

12.4.3. Hiện thực

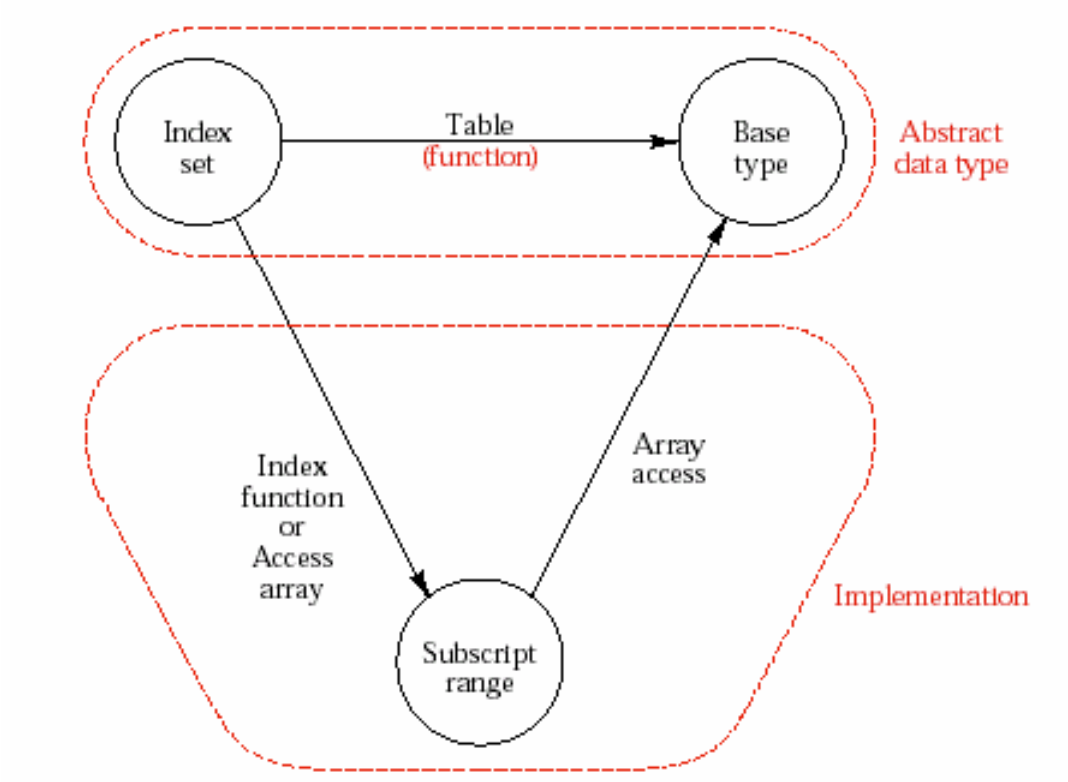
Định nghĩa trên chỉ mới là định nghĩa của một kiểu dữ liệu trừu tượng mà chưa nói gì đến cách hiện thực. Nó cũng không hề nhắc đến các hàm chỉ số hay các mảng truy xuất. Chúng ta hãy xem hình minh họa trong hình 12.9. Phần trên của hình này cho chúng ta thấy một sự trừu tượng trong định nghĩa, truy xuất bảng đơn giản chỉ là một ánh xạ từ một tập chỉ số sang một kiểu cơ sở. Phần dưới của hình là ý tưởng tổng quát của phần hiện thực. Một hàm chỉ số hoặc một mảng truy xuất nhận thông số từ một tập chỉ số theo một dạng đã được đặc tả nào đó. Chẳng hạn (i,j) trong bảng 2 chiều hoặc (i,j,k) trong bảng 3 chiều với i, j, k đã có miền xác định đã định. Kết quả của hàm chỉ số hoặc mảng truy xuất sẽ là một trong các trị trong miền các chỉ số, chẳng hạn tập con của tập các số nguyên. Miền trị này có thể được sử dụng trực tiếp như chỉ số cho mảng và được cung cấp bởi ngôn ngữ lập trình.

Đến đây xem như chúng ta đã giới thiệu xong một kiểu cấu trúc dữ liệu mới, đó là bảng. Tùy từng mục đích của các ứng dụng, bảng có thể có nhiều phiên bản khác nhau. Phần định nghĩa chi tiết hơn cho các phiên bản này cũng như các cách hiện thực của chúng được xem như bài tập. Phần tiếp theo đây trình bày sự giống và khác nhau giữa danh sách và bảng. Sau đó chúng ta sẽ tiếp tục làm quen với một cấu trúc dữ liệu khá đặc biệt và rất phổ biến, đó là bảng băm. Cấu trúc dữ liệu bảng băm cũng xuất phát từ ý tưởng sử dụng bảng như phần này đã giới thiệu.

12.4.4. So sánh giữa danh sách và bảng

Chúng ta hãy so sánh hai kiểu dữ liệu trừu tượng danh sách và bảng. Nền tảng toán học cơ bản của danh sách là **một chuỗi nối tiếp các phần tử**, còn đối với bảng, đó là **tập hợp và hàm**. Chuỗi nối tiếp có một trật tự ngầm trong đó, đó là phần tử đầu tiên, phần tử thứ hai, v.v..., còn tập hợp và hàm không có thứ tự.

Việc truy xuất thông tin trong một danh sách thường liên quan đến việc tìm kiếm, nhưng việc truy xuất thông tin trong bảng đòi hỏi những phương pháp khác, đó là các phương pháp có thể đi thẳng đến phần tử mong muốn. Thời gian cần thiết để tìm kiếm trong danh sách nói chung phụ thuộc vào n là số phần tử trong danh sách và ít nhất là bằng lg n, nhưng thời gian để truy xuất bảng thường không phụ thuộc vào số phần tử trong bảng, và thường là O(1). Vì lý do này, trong nhiều ứng dụng, việc truy xuất bảng thực sự nhanh hơn việc tìm kiếm trong một danh sách.



Hình 12.9 – Hiện thực của bảng

Mặt khác, **duyệt là một tác vụ tự nhiên đối với một danh sách, nhưng đối với bảng thì không**. Việc di chuyển xuyên suốt một danh sách để thực hiện một tác vụ nào đó lên từng phần tử của nó nói chung là dễ dàng. Điều này đối với bảng không dễ dàng chút nào, đặc biệt trong trường hợp có yêu cầu trước về một trật tự nào đó của các phần tử được duyệt.

Cuối cùng, chúng ta cần làm rõ sự khác nhau giữa bảng và mảng. Nói chung, chúng ta dùng từ bảng như là chúng ta đã định nghĩa trong phần vừa rồi và giới hạn từ mảng chỉ với nghĩa như là một phương tiện dùng để lập trình của C++ và phần lớn các ngôn ngữ cấp cao, các mảng này thường được sử dụng để hiện thực cả hai: bảng và danh sách liên tục.

12.5. Bảng băm

Khi giới thiệu tổng quát về bảng cũng như cách sử dụng hàm chỉ số và mảng truy xuất, chúng ta cần nhận ra một điều rằng, thông số cho hàm chỉ số hoặc mảng truy xuất phần nào phản ánh vị trí, hay nói rõ hơn, đó là trật tự của phần tử cần truy xuất trong bảng. Chẳng hạn trật tự theo chỉ số hàng và cột trong bảng (i,j), hay trường hợp danh sách các khách hàng sử dụng điện thoại: tên của các khách hàng có thứ tự theo *alphabet*. Bảng băm mà chúng ta sẽ nghiên cứu tiếp theo mang một đặc điểm hoàn toàn khác. Việc truy xuất bảng bắt đầu từ giá trị của khóa trong phần tử dữ liệu, và thông thường khóa này không liên quan đến trật tự trong hàng hoặc cột của bảng để có thể sử dụng một hàm chỉ số đơn giản cho ra vị trí của nó trong bảng như ở phần trên đã giới thiệu.

12.5.1. Các bảng thưa

12.5.1.1. Các hàm chỉ số

Điều chúng ta có thể làm là xây dựng sự tương ứng một – một giữa các khóa và các chỉ số mà chúng ta sử dụng để truy xuất bảng. So với các phần trước, hàm chỉ số mà chúng ta xây dựng ở đây sẽ phức tạp hơn, vì có khi chúng ta cần đến sự biến đổi của các khóa, chẳng hạn từ các chữ cái sang các số nguyên. Theo nguyên tắc, điều này luôn có thể làm được.

Khó khăn thực sự chỉ là khi số các khóa có thể có vượt ra ngoài không gian của bảng. Lấy ví dụ, nếu các khóa là các từ có 8 ký tự, thì có thể có đến $26^8 \approx 2 \times 10^{11}$ khóa khác nhau, và đây cũng là con số lớn hơn rất nhiều dung lượng cho phép của một bộ nhớ tốc độ cao. Tuy nhiên trong thực tế, chỉ có một số không lớn các khóa này là thực sự xuất hiện. Điều đó có nghĩa là bảng chứa sẽ rất thưa thớt. Chúng ta có thể xem bảng được đánh chỉ số bằng một tập rất lớn, nhưng chỉ có một số tương đối ít vị trí là thực sự có phần tử.

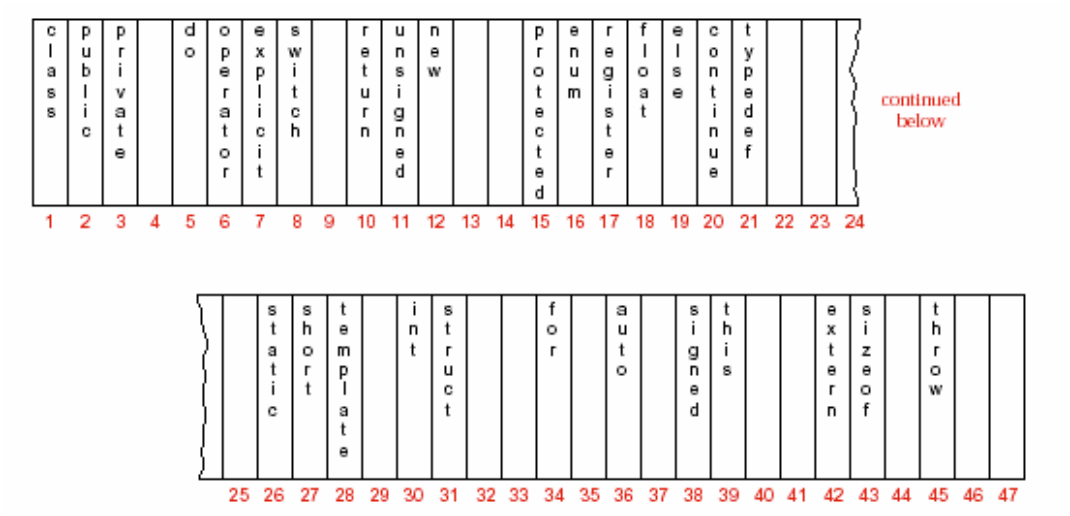
12.5.1.2. Khái niệm băm

Nhằm tránh một bảng quá thưa thớt có nhiều vị trí không bao giờ được dùng đến, chúng ta làm quen với khái niệm băm. Ý tưởng của bảng băm (hình 12.10) là cho phép ánh xạ một tập các khóa khác nhau vào các vị trí trong một mảng với kích thước cho phép. Gọi kích thước mảng này là `hash_size`, mỗi khóa sẽ được ánh xạ vào một chỉ số trong khoảng [0, `hash_size`-1]. Ánh xạ này được gọi là hàm băm (*hash function*). Một cách lý tưởng, hàm này cần có cách tính đơn giản và phân bổ các khóa sao cho hai khóa khác nhau luôn vào hai vị trí khác nhau. Nhưng do kích thước mảng là giới hạn và miền trị của các khóa là rất lớn, điều này là không thể được. Chúng ta chỉ có thể hy vọng rằng một hàm băm tốt thì sẽ phân bổ được các khóa vào các chỉ số một cách khá đồng đều và tránh được hiện tượng gom tụ.

Hàm băm nói chung luôn ánh xạ một vài khóa khác nhau vào cùng một chỉ số. Nếu phần tử cần tìm đang nằm tại chỉ số được ánh xạ đến, vấn đề của chúng ta xem như đã được giải quyết; ngược lại, chúng ta cần sử dụng một phương pháp nào đó để giải quyết đụng độ. Việc đụng độ (*collision*) xảy ra khi hai phần tử cần được chứa trong cùng một vị trí của bảng.

Trên đây là ý tưởng cơ bản của việc sử dụng bảng băm. Có ba vấn đề chúng ta cần xem xét khi sử dụng phương pháp băm:

- Tìm hàm băm tốt.
- Xác định phương pháp giải quyết đụng độ.
- Xác định kích thước bảng băm.



Hình 12.10 – Bảng băm

12.5.2. Lựa chọn hàm băm

Hai tiêu chí cơ bản để chọn lựa một hàm băm là:

- Hàm băm cần được tính toán dễ dàng và nhanh chóng.
- Việc phân phối các khóa có thể xuất hiện rải đều trên bảng băm.

Nếu chúng ta biết trước chính xác những khóa nào sẽ xuất hiện, thì chúng ta có thể xây dựng một hàm băm thật hiệu quả, nhưng nói chung chúng ta thường không biết trước điều này.

Chúng ta cần lưu ý rằng một hàm băm không hề có tính ngẫu nhiên. Khi tính nhiều lần cho cùng một khóa, một hàm băm phải cho cùng một trị, có như vậy thì khóa mới có thể được truy xuất sau khi được lưu trữ.

12.5.2.1. Chia lấy phần dư (*modular arithmetic*)

Trước hết chúng ta hãy xem xét một trường hợp thật đơn giản. Nếu các khóa là các số nguyên, hàm băm đơn giản và phổ biến được dùng là phép chia cho hash_size để lấy phần dư, vì như vậy chúng ta sẽ có các chỉ số thuộc [0, hash_size -1]. Tuy nhiên cũng cần lưu ý những trường hợp các khóa tập trung vào một số giá trị đặc biệt nào đó. Chẳng hạn nếu hash_size = 10, mà phần lớn các khóa lại có con số ở hàng đơn vị là 0. Sự phân tán các khóa phụ thuộc nhiều vào phép chia lấy phần dư, đó chính là kích thước của bảng băm. Nếu kích thước đó là một bội số của các số nguyên nhỏ như 2 hoặc 10, thì rất nhiều khóa sẽ cho cùng chỉ số như nhau, trong khi đó có một số chỉ số rất ít được sử dụng đến. Cách chọn phép chia lấy phần dư tốt nhất thường là chia cho một số nguyên tố (nhưng không phải là luôn luôn), kết quả sẽ rải đều các khóa trong bảng băm hơn. Như vậy, thay vì chọn bảng băm kích thước 1000, chúng ta nên chọn kích thước 997 hoặc 1009; cách chọn $2^{10} = 1024$ là một cách chọn rất dở.

Thông thường, các khóa là các chuỗi ký tự. Một cách tự nhiên, người ta thường lấy một số nguyên bằng với tổng của các mã ASCII của các ký tự trong khóa làm đại diện cho nó. Hàm băm với cách viết của C chuẩn sau đây thật đơn giản và tính cũng rất nhanh:

```
index Hash(const char *Key, int hash_size)
{
     unsigned int HashVal = 0;
     while (*Key != '\0')
     {
          HashVal += *Key;
          Key++;
     }
             return HashVal % hash_size;
}
```

Tuy nhiên, nếu hash_size lớn, hàm sẽ không phân bổ các khóa tốt. Lấy ví dụ với hash_size =10007 (một số nguyên tố). Giả sử các khóa có chiều dài 8 ký tự hoặc ít hơn. Mỗi ký tự có mã ASCII <=127. Giá trị của hàm băm chỉ có thể từ 0 đến 127 x 8 = 1016.

Một cải tiến khác của hàm băm như sau: với giả thiết rằng các khóa đều có ít nhất 3 ký tự, số 27 được dùng vì đó là số ký tự trong bảng chữ cái tiếng Anh (tính cả khoảng trắng).

```
index Hash(const char *Key, int hash_size)
{
   return (Key[0] + 27*Key[1] + 27*27*Key[2]) %hash_size;
}
```

Hàm này chỉ quan tâm 3 ký tự đầu của các khóa, nhưng nếu chúng là ngẫu nhiên và hash_size là 10007 như trên, thì sự phân bổ khá đồng đều. Điều không may ở đây là các từ trong tiếng Anh không phải là một sự ghép các ký tự một cách ngẫu nhiên. Mặc dù có đến $26^3 = 17576$ khả năng ghép 3 ký tự, thực tế trong từ điển cho thấy chỉ có 2851 khả năng xảy ra. Ngay cả khi không có sự đụng độ xảy ra giữa từng cặp trong các khả năng này, thì cũng chỉ có 28% vị trí trong bảng là được sử dụng.

Thêm một cải tiến khác như sau đây:

```
index Hash(const char *Key, int hash_size)
{
     unsigned int HashVal = 0;
     while (*Key != '\0')
     {
          HashVal = (HashVal << 5 ) + *Key);
          Key++;
     }
             return HashVal % hash_size;
}
```

Hàm này quan tâm đến mọi ký tự trong khóa và nói chung có thể phân bổ các khóa đồng đều trong một bảng kích thước tương đối lớn. Trị của hàm được tính $\sum_{i=0}^{KeySize-1}$ Key[KeySize-i-1].32^i. Đây là đa thức với hệ số là 32 và sử dụng công thức Horner. Ví dụ, để tính $h_k = k_1 + 27k_2 + 27^2k_3$, người ta tính $h_k = ((k_3)*27 + k_2)*27 + k_1$. Việc dùng số 32 thay số 27 là vì với 32 thì không cần làm phép nhân mà chỉ đơn giản là phép dịch chuyển bit ($32 = 2^5$), và thực tế là dùng phép XOR.

Hàm trên đây chưa phải là hàm tốt nhất khi xét đến tiêu chí phân bổ đồng đều, nhưng nó cho phép việc tính toán được thực hiện rất nhanh chóng. Nếu khóa quá dài thì nó cũng lộ nhược điểm là phải tính quá lâu. Hơn nữa quá trình dịch bit sẽ làm mất đi tác dụng của các ký tự đã được xét trước. Thực tế khắc phục điều này bằng cách không sử dụng tất cả các ký tự có trong khóa.

12.5.2.2. Cắt xén (*truncation*)

Phương pháp cắt xén bỏ qua một phần của khóa, phần còn lại được xem như chỉ số (các dữ liệu không phải số thì lấy theo bảng mã của chúng). Ví dụ, nếu khóa là một số nguyên 8 ký số và bảng băm có 1000 vị trí, thì việc lấy từ vị trí thứ nhất, thứ hai và thứ năm kể từ phải sang sẽ là hàm băm. Có nghĩa là khóa 21**2**96**87**6 có chỉ số là 976. Cắt xén là một phương pháp cực nhanh, nhưng nó thường không phân phối các khóa đều khắp bảng băm.

12.5.2.3. Xáo trộn (*folding*)

Ý tưởng xáo trộn (*folding*) dưới đây giúp cho các bộ phận của khóa đều có thể tham gia vào việc xác định kết quả cuối cùng của hàm băm. Từ băm ở đây có nghĩa là kết quả sinh ra có phần giống với khóa ban đầu. Ngoài ra, sự xáo trộn cho phép chúng ta hy vọng rằng mọi khuôn mẫu hoặc sự lặp lại có thể xuất hiện trong các khóa (hậu quả của tính thiếu ngẫu nhiên của dữ liệu trong thực tế) sẽ bị triệt tiêu. Có như vậy thì các kết quả mới được phân phối theo cùng một quy luật như nhau mà không có sự trùng lặp của từng nhóm kết quả và chúng ta tránh được hiện tượng gom tụ. Ở đây chúng ta thấy rằng thuật ngữ "băm" mang tính mô tả rõ nhất. Tuy nhiên trong một số tài liệu khác người ta dùng các các từ mang tính kỹ thuật hơn như "bộ nhớ phân tán" (*scatter-storage*) hoặc "phép biến đổi khóa" (*key-transformation*).

Phương pháp xáo trộn chia khóa làm nhiều phần và kết nối các phần này lại theo một cách thích hợp (thường sử dụng phép cộng hoặc phép nhân). Lấy ví dụ, một số nguyên 8 ký số có thể được chia làm 3 nhóm gồm 3, 3, và 2 ký số, các nhóm này được cộng lại với nhau, sau đó có thể được cắt xén bớt nếu cần thiết để cho ra các chỉ số phù hợp kích thước bảng băm. Khóa 21296876 sẽ được băm thành 212 + 968 + 76 = 1256, cắt ngắn còn 256. Do mọi dữ liệu trong khóa đều có ảnh hưởng đến kết quả hàm băm nên phương pháp này làm cho các khóa rải đều trên bảng băm hơn là phương pháp cắt xén nêu trên.

Tóm lại, chúng ta đã xem xét một số phương pháp mà chúng ta có thể kết hợp lại theo nhiều cách khác nhau để xây dựng hàm băm. Lấy phần dư thường là một cách tốt để kết thúc việc tính toán của một hàm băm, do nó vừa có thể đạt được sự rải đều các khóa trong bảng băm vừa bảo đảm kết quả nhận được luôn nằm trong miền các chỉ số cho phép.

12.5.3. Phác thảo giải thuật cho các thao tác dữ liệu trong bảng băm

Trước hết, chúng ta cần khai báo một mảng để chứa bảng băm. Sau đó, các vị trí trong mảng cần được khởi tạo là trống. Giá trị khởi tạo phụ thuộc vào ứng dụng, thông thường chúng ta cho các vị trí trống này chứa một giá trị đặc biệt nào đó. Chẳng hạn, với các khóa là các chữ cái, một trị chứa toàn ký tự trống có thể biểu diễn một vị trí trống.

Để thêm một phần tử vào bảng băm, cần tính hàm băm cho khóa của nó. Nếu vị trí tìm thấy còn trống, phần tử sẽ được thêm vào; nếu đã có phần tử tại vị trí này và khóa của nó trùng với khóa của phần tử cần thêm thì việc thêm sẽ không được thực hiện; trường hợp cuối cùng, nếu tại vị trí tìm thấy đã có một phần tử nhưng của một khóa khác, chúng ta sẽ áp dụng một phương pháp giải quyết đụng độ nào đó để tìm đến một vị trí khác cho việc thêm phần tử mới của chúng ta.

Việc truy xuất một phần tử với khóa cho trước được làm tương tự. Trước tiên, hàm băm được tính cho khóa cho trước. Nếu phần tử cần tìm đang nằm tại vị trí được chỉ bởi hàm băm, thì việc truy xuất sẽ được thực hiện thành công; ngược lại, trong khi mà vị trí đang xét không trống và mọi vị trí chưa được xét đến, cần tiến hành các bước tương tự như các bước đã được sử dụng khi giải quyết đụng độ trong quá trình thêm vào. Nếu trong khi tìm kiếm gặp một vị trí trống, hoặc khi mọi vị trí đã được xét đến, thì có thể kết luận việc tìm kiếm thất bại: không có phần tử với khóa cần tìm trong bảng băm.

12.5.4. Ví dụ trong C++

Như một ví dụ đơn giản, chúng ta sẽ viết một hàm băm trong C++ để chuyển đổi một khóa gồm 8 ký tự chữ cái sang một số nguyên trong miền

```
0 . . hash_size – 1.
```

Chúng ta có một lớp **Key** với các phương thức như sau:

```
class Key: public String{
public:
    char key_letter(int position) const;
    void make_blank();
    //  Các constructor và các phương thức khác.
};
```

Để giảm công sức lập trình khi hiện thực lớp, chúng ta chọn cách thừa kế các phương thức của lớp **String** trong chương 5. Chúng ta sẽ đỡ phải viết lại các tác vụ so sánh. Phương thức **key_letter(int position)** trả về ký tự tại vị trí **position** trong khóa, hoặc trả về khoảng trắng nếu khóa có chiều dài nhỏ hơn n. Phương thức **make_blank** tạo một khóa trống.

```
int hash(const Key &target)
/*
post: Hàm băm trên target trả về trị trong miền 0 .. hash_size-1.
uses: Các phương thức của lớp Key.
*/
{
   int value = 0;
   for (int position = 0; position < 8; position++)
      value = 4 * value + target.key_letter(position);
   return value % hash_size;
}
```

Hàm băm trên đơn giản chỉ cộng dồn các mã của mỗi ký tự trong khóa sau khi đã nhân với 4. Chúng ta không thể lý giải được rằng phương pháp này là tốt hơn (hoặc xấu hơn) một vài phương pháp khác. Chúng ta cũng có thể lấy mã của ký tự trừ đi một số nào đó, rồi nhân từng cặp với nhau, hoặc bỏ qua một vài ký tự nào

đó. Đôi khi một ứng dụng sẽ cho thấy một hàm băm này là tốt hơn một hàm băm khác, đôi khi cần phải có sự khảo sát bằng thực nghiệm mới chỉ ra được hàm nào là tốt hơn.

12.5.5. Giải quyết đụng độ bằng phương pháp địa chỉ mở

Có hai nhóm phương pháp giải quyết đụng độ: nhóm phương pháp địa chỉ mở và nhóm phương pháp nối kết. Nhóm phương pháp địa chỉ mở chỉ sử dụng các mảng cấp phát tĩnh. Nhóm phương pháp nối kết có sử dụng những vùng nhớ cấp phát động được quản lý bởi các con trỏ nối kết.

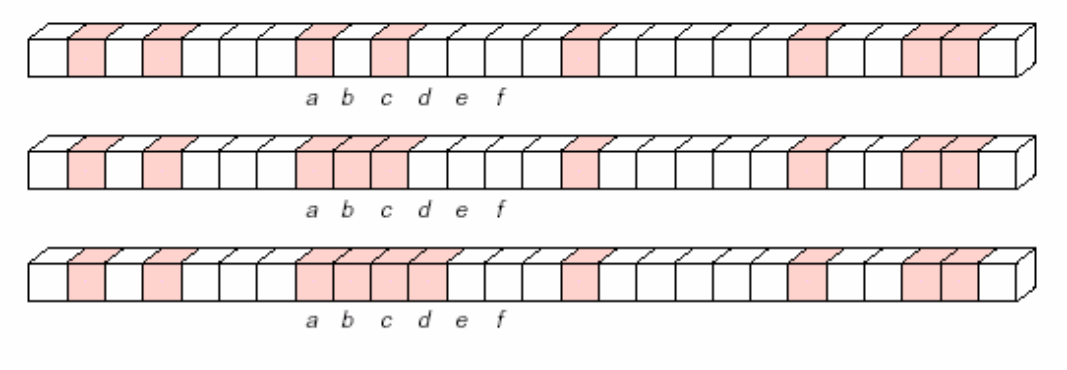
Dưới đây là các phương pháp dùng địa chỉ mở.

12.5.5.1. Thử tuyến tính

Phương pháp đơn giản nhất để giải quyết đụng độ là bắt đầu từ vị trí trả về từ hàm băm có xảy ra đụng độ, việc tìm kiếm sẽ tiếp tục một cách tuần tự ở các vị trí kế trong bảng cho đến khi gặp khóa mong muốn hoặc một vị trí trống. Phương pháp này được gọi là phương pháp thử tuyến tính (*linear probing*). Bảng được xem như một mảng vòng, khi việc tìm kiếm đạt đến vị trí cuối của bảng thì sẽ quay về vị trí đầu của bảng.

Hiện tượng gom tụ

Nhược điểm chính của phương pháp thử tuyến tính là khi có khoảng một nửa số vị trí trong bảng đã chứa dữ liệu, khuynh hướng gom tụ sẽ xuất hiện; nghĩa là, các phần tử sẽ nằm trong các chuỗi liên tục các vị trí, giữa các chuỗi này là những lỗ hổng. Việc tìm kiếm tuần tự một vị trí trống trong bảng sẽ ngày càng lâu hơn. Chúng ta hãy xem ví dụ ở hình 12.11. Giả sử mảng có n vị trí thì xác suất mà hàm băm chọn một vị trí nào đó là 1/n. Ban đầu việc phân phối được thực hiện khá đều trong bảng (phần trên của hình). Giả sử cần thêm dữ liệu mới mà hàm băm trả về vị trí b thì dữ liệu được thêm vào tại đây, nhưng nếu hàm băm trả về vị trí a mà vị trí này đã có dữ liệu, việc thêm vào sẽ được thực hiện tại b. Như vậy xác suất để vị trí b nhận dữ liệu là 2/n. Tại bước tiếp theo, khi dữ liệu cần thêm vào một trong các vị trí a, b, c, hoặc d thì chỗ trống thực sự để thêm vào chỉ là d, như vậy xác suất dữ liệu thêm vào d là 4/n. Sau đó, xác suất dữ liệu thêm vào vị trí e lại là 5/n. Và cứ như thế, khi dữ liệu càng được thêm vào nhiều thì chuỗi liên tục các vị trí đã có dữ liệu bắt đầu từ a ngày càng dài ra. Như vậy cách thực hiện của bảng băm bắt đầu suy thoái dần tới sự tìm kiếm tuần tự.



Hình 12.11 – Hiện tượng gom tụ trong bảng băm.

Hiện tượng gom tụ chính là nguyên nhân của tính thiếu ổn định. Nếu một số ít các khóa ngẫu nhiên nằm kế nhau, thì sau đó các khóa khác bỗng trở nên kết dính với chúng, có nghĩa là vị trí chứa chúng phụ thuộc lẫn nhau, và sự phân phối dần dần trở nên thiếu cân bằng.

12.5.5.2. Hàm gia tăng

Để tránh hiện tượng gom tụ, chúng ta phải sử dụng phương pháp phức tạp hơn để chọn ra chuỗi các vị trí cần xem xét đến để thêm một dữ liệu mới nào đó khi có xảy ra đụng độ. Có nhiều cách để thực hiện. Ý tưởng chung là sử dụng **một hoặc một vài hàm gia tăng để xác định khoảng cách từ vị trí vừa đụng độ đến một vị trí mới**. Cần lưu ý rằng kết quả của hàm gia tăng không được phép trả về trị 0.

Hàm gia tăng có thể phụ thuộc vào khóa, hoặc vào số lần đã thử, sao cho có thể tránh được hiện tượng gom tụ.

Trường hợp thứ nhất, **khi hàm gia tăng phụ thuộc vào khóa**, chúng ta có khái niệm băm lại. Đó là cách sử dụng một hàm băm thứ hai. Kết quả của hàm băm này là số vị trí cần di chuyển kể từ vị trí đã bị đụng độ trước đó. Nếu vị trí này lại đụng độ, chúng ta lại dùng một hàm băm khác nữa để tìm đến vị trí thứ ba, và cứ thế. Cũng có khi từ kết quả tính của hàm băm thứ hai người ta dùng luôn số này để di chuyển giữa hai lần thử kế tiếp.

Trong trường hợp thứ hai, **hàm gia tăng phụ thuộc vào số lần đã thử**, có thể kể ra đây phương pháp thử bậc hai.

Thử bậc hai

Nếu có sự đụng độ tại địa chỉ băm được h, phương pháp thử bậc hai (*quadratic probing*) thử các vị trí kế tiếp là `h+1, h+4, h+9`, ... trong bảng, có nghĩa là các vị trí $h + i^2$, với i là lần thử. Nói cách khác, hàm gia tăng là i^2.

Phương pháp thử tuyến tính đã nêu trên cũng có thể xem như một trường hợp sử dụng hàm gia tăng là i.

Phương pháp thử bậc hai thực sự có làm giảm hiện tượng gom tụ, nhưng thực tế thường nó không thể thử hết mọi vị trí trong bảng. Đối với một vài giá trị của hash_size, hàm i^2 sẽ thử một số tương đối ít các vị trí trong bảng. Lấy ví dụ, khi hash_size là một bội số lớn của 2, chỉ khoảng một phần sáu số các vị trí trong bảng băm là được thử. Khi hash_size là một số nguyên tố, thử bậc hai sẽ đạt được một nửa số vị trí trong bảng băm.

Để chứng minh điều trên, giả sử rằng hash_size là một số nguyên tố. Giả sử chúng ta cùng đạt một vị trí khi thử lần thứ i và lần thứ i + j với j là một số nguyên > 0. Giả sử j là một số nguyên nhỏ nhất theo điều kiện trên. Giá trị tính được bởi hàm băm lần thứ i và lần thứ i + j khác nhau bởi một bội số của hash_size. Nói cách khác,

$$h + i^2 \equiv h + (i + j)^2 \pmod{hash\_size}$$

Biến đổi biểu thức trên ta có:

$$j^2 + 2ij = j(j + 2i) \equiv 0 \pmod{hash\_size}.$$

Biểu thức này có nghĩa là j(j + 2i) chia hết cho hash_size. Một tích chia hết cho một số nguyên tố chỉ khi một trong các thừa số của tích đó chia hết cho số nguyên tố đó. Vậy hoặc j chia hết cho hash_size, hoặc j+2i chia hết cho hash_size. Trong trường hợp thứ nhất, chúng ta đã phải thử j=hash_size lần trước khi gặp lại vị trí đã thử với i (chúng ta nhớ rằng j là số nhỏ nhất theo giả thiết). Tuy nhiên trường hợp thứ hai sẽ xảy ra sớm hơn, khi j=hash_size –2i, hoặc khi biểu thức tăng thêm hash_size nếu biểu thức này âm. Do đó tổng số vị trí khác nhau được thử sẽ là

$$(hash\_size + 1) / 2.$$

Khi đã thử với số lần như trên chúng ta có thể xem như bảng đã đầy.

Chú ý rằng phương pháp thử bậc hai có thể được thực hiện mà không cần phép nhân: Sau lần thử thứ nhất tại vị trí h, biến tăng được gán là 1. Tại mỗi lần thử thành công, biến tăng sẽ tăng thêm 2 sau khi nó đã được thêm vào vị trí trước đó.

Do $1 + 3 + 5 + \ldots + (2i - 1) = i^2$

đối với mọi i≥1 , lần thử i sẽ tìm tại vị trí h +1 +. . . +(2i-1)=h+i^2, theo như mong muốn.

12.5.5.3. Thử ngẫu nhiên

Phương pháp cuối cùng là sử dụng số ngẫu nhiên được sinh ra để làm biến gia tăng. Chúng ta chỉ được dùng một bộ sinh số ngẫu nhiên để từ một số bắt đầu cho trước nó luôn luôn sinh ra cùng một chuỗi các số ngẫu nhiên kế tiếp. Đây là một phương pháp rất tốt để tránh hiện tượng gom tụ, nhưng nó có thể chậm hơn các phương pháp khác.

12.5.5.4. Giải thuật C++

Để kết thúc việc nghiên cứu về phương pháp địa chỉ mở, chúng ta có một ví dụ C++ với các khóa là các ký tự chữ cái. Chúng ta giả sử rằng lớp Key và lớp Record có các đặc tính mà chúng ta vừa sử dụng trong hai phần cuối. Lớp Key có phương thức **key_letter(int position)** để trả về ký tự tại **position**, lớp Record có phương thức để lấy một khóa của một phần tử.

Bảng băm của chúng ta sẽ có khai báo như sau:

```
const int hash_size = 997;   //  Số nguyên tố
class Hash_table {
public:
   Hash_table();
   void clear();
   Error_code insert(const Record &new_entry);
   Error_code retrieve(const Key &target, Record &found) const;
private:
   Record table[hash_size];
};
```

Bảng băm sẽ được khởi tạo sao cho tất cả các phần tử trong mảng đều chứa khóa đặc biệt gồm 8 khoảng trắng. Đây là nhiệm vụ của *constructor*:

```
Hash_table:: Hash_table();
// post: Bảng băm được tạo và được khởi tạo là rỗng.
```

Phương thức clear cần để loại tất cả các dữ liệu hiện có trong bảng băm:

```
void Hash_table::clear();
// post: Bảng băm đã được dọn dẹp và trở thành bảng băm rỗng.
```

Mặc dù chúng ta đã bắt đầu đặc tả các phương thức của bảng băm, chúng ta sẽ không tiếp tục phát triển thành một gói tổng quát và đầy đủ. Do việc chọn một hàm băm tốt phụ thuộc nhiều vào loại của khóa sẽ được sử dụng, các phương thức

của bảng băm thường phụ thuộc mạnh mẽ vào từng ứng dụng riêng, một gói tổng quát cho một bảng băm là không có lợi.

Để minh họa cách viết các hàm tiếp theo, chúng ta sẽ sử dụng phương pháp thử bậc hai để giải quyết đụng độ. Chúng ta đã chứng minh rằng số lần thử tối đa có thể thực hiện theo phương pháp này là `(hash_size+1) / 2`, nên sẽ dùng biến đếm **`probe_count`** để kiểm tra giới hạn này.

Với những quy ước như trên, chúng ta có phương thức thêm một phần tử **`new_entry`** vào bảng băm như sau:

```
Error_code Hash_table::insert(const Record &new_entry)
/*
post: Nếu bảng băm đầy, phương thức trả về overflow.
      Nếu bảng băm đã chứa phần tử có khóa trùng khóa trong new_entry thì ph7ơng thức trả
      về duplicate_error. Ngược lại, phần tử new_entry đươc thêm vào bảng băm và
      phương thức trả về success.
uses: Các phương thức của các lớp Key và Record, hàm hash.
*/
{
   Error_code result = success;
   int probe_count,     //  Đếm số lần thử để phát hiện bảng đầy.
       increment,       //  Số gia tăng bở phép thử bậc hai.
       probe;           //  Vị trí thử hiện thời.
   Key null;            //  Giá trị NULL của khóa dùng cho phép so sánh.
   null.make_blank();

   probe = hash(new_entry);
   probe_count = 0;
   increment = 1;

   while (table[probe] != null                 // Vị trí thử có trống hay không?
     && table[probe] != new_entry              // Khóa đã có trong bảng băm?
     && probe_count < (hash_size + 1) / 2) {// Bảng đầy hay chưa?
      probe_count++;
      probe = (probe + increment) % hash_size;
      increment += 2;           // Tính lại độ dời cho lần thử kế tiếp.
   }
   if (table[probe]==null) table[probe]= new_entry;
   else if (table[probe] == new_entry) result = duplicate_error;
   else result =  overflow;
   return result;
}
```

Phương thức để truy xuất một phần tử với một khóa cho trước có dạng tương tự, chúng ta dành lại như bài tập. Đặc tả của nó như sau:

```
Error_code Hash_table:: retrieve(const Key &target, Record &found) const;
//post: Nếu một phần tử trong bảng băm có khóa giống target, thì found sẽ được gán trị của
        phần tử đó, phương thức trả về success. Ngược lại, trả về not_present.
```

12.5.5.5. Loại bỏ một phần tử

Cho đến bây giờ, chúng ta vẫn chưa nói gì đến việc loại một phần tử trong bảng băm. Thoạt nhìn, dường như đó là một việc dễ dàng, chỉ cần gán lại cho vị trí cần loại một trị đặc biệt của khóa để chỉ ra rằng đó là vị trí trống. Tuy nhiên cách này không thể áp dụng được. Lý do là một vị trí trống được xem như một dấu hiệu để kết thúc quá trình tìm kiếm một khóa. Giả sử như trước khi loại, đã xảy ra một hoặc hai lần đụng độ và một phần tử nào đó lẽ ra phải được thêm vào tại vị trí đang xét lại phải dời đến một vị trí đâu đó trong bảng. Nếu bây giờ chúng ta cần truy xuất đến phần tử này thì chỗ trống mới được tạo ra sẽ kết thúc việc tìm kiếm, và chúng ta không thể tìm thấy nó, mặc dù nó vẫn tồn tại trong bảng.

Một phương pháp để ngăn ngừa tình huống trên là sử dụng một khóa đặc biệt để đặt vào các vị trí cần loại đi phần tử. Khóa đặc biệt này chỉ ra rằng đó là một vị trí trống có thể thêm phần tử mới vào, nhưng nó không được dùng để kết thúc quá trình tìm kiếm một khóa nào khác trong bảng. Tuy nhiên cách sử dụng khóa đặc biệt này sẽ làm cho giải thuật phức tạp hơn và chậm hơn. Còn một số phương pháp khác có thể áp dụng trong việc loại bỏ một phần tử khỏi bảng băm. Tuy nhiên chúng ta cần nhớ rằng phương pháp loại bỏ nào cũng phải tương thích với chiến lược thêm và tìm kiếm phần tử, để hai tác vụ này luôn hoạt động một cách chính xác. Danh sách liên kết trong mảng liên tục trong phần 4.5 cũng thường được sử dụng làm bảng băm, và cũng thuộc nhóm phương pháp địa chỉ mở để giải quyết đụng độ. Các phần tử có cùng giá trị hàm băm sẽ được nối kết trong cùng một danh sách liên kết. Và trong bảng băm có nhiều danh sách liên kết như vậy.

12.5.6. Giải quyết đụng độ bằng phương pháp nối kết

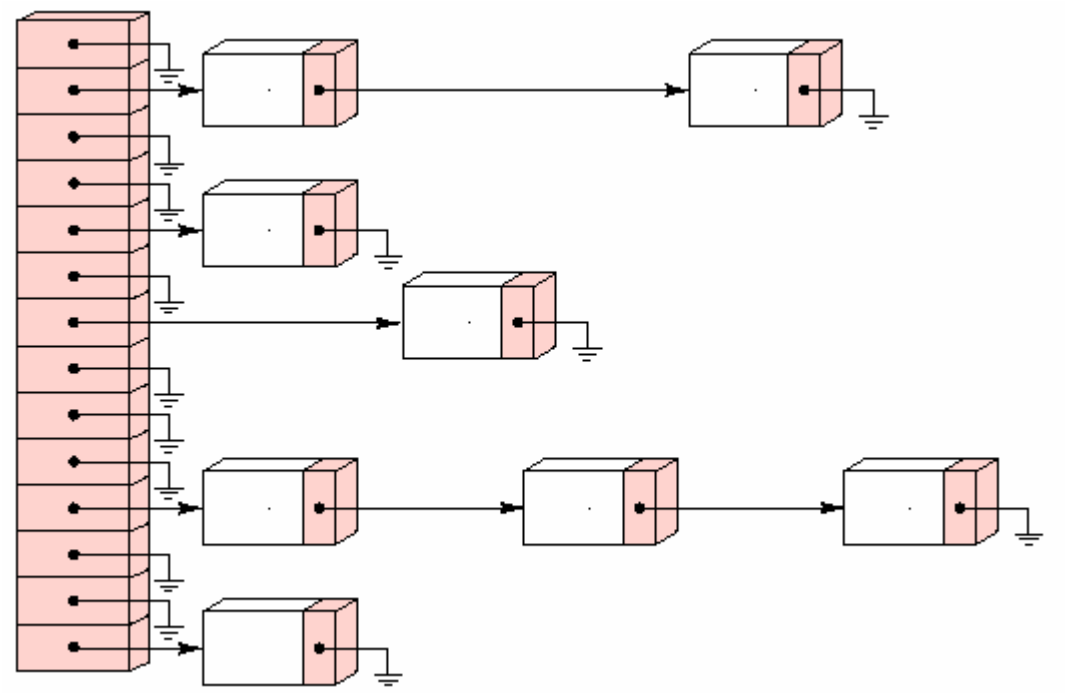
Cho đến bây giờ chúng ta vẫn cứ ngầm hiểu rằng chúng ta chỉ sử dụng vùng nhớ liên tục để chứa bảng băm. Thật vậy, vùng nhớ liên tục là cách chọn tự nhiên để hiện thực bảng băm, do chúng ta cần truy xuất một vị trí ngẫu nhiên trong bảng một cách nhanh chóng, mà vùng nhớ liên kết thì không hỗ trợ việc truy xuất ngẫu nhiên. Tuy nhiên, điều đó không có nghĩa là vùng nhớ liên kết không thể được dùng để chứa các phần tử. Chúng ta có thể dùng bảng băm là một mảng các danh sách liên kết. Chúng ta hãy xem hình 12.12.

Người ta thường quen gọi các danh sách liên kết từ bảng băm là các chuỗi mắc xích nối kết (*chain*) nên phương pháp giải quyết đụng độ này còn được gọi là phương pháp nối kết (*chaining*).

12.5.6.1. Ưu điểm của phương pháp nối kết

Ưu điểm thứ nhất và cũng là ưu điểm quan trọng nhất của phương pháp này là nó có thể tiết kiệm vùng nhớ khi bản thân các phần tử khá lớn. Do bảng băm là một mảng, chúng ta cần khai báo trước một số lượng phần tử khá lớn để tránh

hiện tượng tràn. Nếu để các phần tử nằm trong bảng băm thì khi chưa có nhiều dữ liệu, có quá nhiều vị trí để trống, trong khi chương trình của chúng ta có thể cần nhiều vùng nhớ cho những biến khác nữa. Ngược lại, nếu bảng băm chỉ chứa các con trỏ mà mỗi con trỏ chỉ cần chiếm số byte bằng số byte của một từ thì kích thước bảng băm giảm đáng kể.



Hình 12.12 – Bảng băm nối kết

Ưu điểm chính thứ hai của việc lưu các danh sách liên kết kèm với bảng băm là nó cho phép xử lý đụng độ một cách đơn giản và hiệu quả. Với một hàm băm tốt, chỉ có một số ít khóa là trùng địa chỉ băm, và như vậy các danh sách liên kết đều ngắn và các khóa đều có thể được tìm kiếm nhanh chóng. Gom tụ không còn là vấn đề phải quan tâm bởi vì các khóa có các địa chỉ băm khác nhau luôn nằm trong các danh sách khác nhau.

Ưu điểm thứ ba là kích thước bảng băm không nhất thiết phải lớn hơn số phần tử có thể có, chúng ta không còn phải lo vấn đề tràn. Khi số phần tử nhiều hơn kích thước bảng băm thì chỉ có nghĩa rằng, chắc chắn là có một vài danh sách liên kết nào đó có nhiều hơn một phần tử. Ngay cả khi số phần tử nhiều gấp vài lần kích thước bảng băm thì chiều dài trung bình của mỗi danh sách liên kết vẫn nhỏ và việc tìm kiếm tuần tự trên từng danh sách vẫn còn hiệu quả.

Cuối cùng, việc loại một phần tử trở thành một công việc dễ dàng và nhanh chóng đối với bảng băm theo phương pháp nối kết. Việc loại bỏ này được tiến

hành hoàn toàn giống với việc loại một phần tử ra khỏi một danh sách liên kết đơn.

12.5.6.2. Nhược điểm của phương pháp nối kết

Các ưu điểm của bảng băm theo phương pháp nối kết thực sự là rất có lợi. Chúng ta nên tin rằng phương pháp nối kết luôn là phương pháp tốt hơn so với phương pháp địa chỉ mở. Tuy vậy, chúng ta hãy xét đến một nhược điểm quan trọng của nó: mọi mối liên kết đều chiếm vùng nhớ. Nếu phần tử có kích thước lớn thì kích thước của các con trỏ sẽ không đáng kể, ngược lại sẽ là điều không hay.

Giả sử rằng mỗi mối liên kết chiếm một từ (một *word* chiếm 2 hoặc 4 *bytes*) và mỗi phần tử cũng chỉ chiếm một từ. Những ứng dụng như vậy cũng tương đối phổ biến, trong đó chúng ta sử dụng bảng băm chỉ để trả lời một vài câu hỏi yes-no về các khóa. Giả sử chúng ta dùng bảng băm theo phương pháp nối kết và khai báo một mảng nhỏ để chứa bảng băm với n là số phần tử của mảng mà cũng là số phần tử sẽ có. Chúng ta sẽ phải sử dụng 3n từ trong bộ nhớ: n cho bảng băm, n cho các khóa, và n cho các mối liên kết để tìm đến phần tử kế trong các danh sách liên kết. Do bảng băm gần như đầy nên đụng độ sẽ xảy ra nhiều hơn, một số danh sách liên kết sẽ có vài phần tử. Việc tìm kiếm sẽ chậm. Mặt khác, giả sử như chúng ta dùng phương pháp địa chỉ mở. Cũng với 3n từ của bộ nhớ, nếu chúng ta chứa trực tiếp các phần tử trong bảng băm có kích thước 3n này thì chỉ có một phần ba bảng là có dữ liệu, như vậy số đụng độ cũng sẽ tương đối ít và việc tìm một phần tử sẽ nhanh hơn rất nhiều.

12.5.6.3. Các giải thuật trong C++

Bảng băm theo phương pháp nối kết trong C++ có định nghĩa đơn giản như sau:

```
class Hash_table {
public:
   //  Specify methods here.
private:
   List<Record> table[hash_size];
};
```

Ở đây lớp `List` có thể là bất kỳ một hiện thực liên kết tổng quát nào của một danh sách đã học trong chương 4**.** Để được nhất quán, các phương thức của bảng băm nối kết sẽ chứa mọi phương thức của hiện thực bảng băm trước kia của chúng ta. *Constructor* của bảng băm chỉ đơn giản gọi các *constructor* cho từng danh sách của mảng. Trong khi đó việc dọn dẹp xóa sạch các phần tử trong bảng băm nối kết lại là một việc hoàn toàn khác, chúng ta cần dọn dẹp từng danh sách tại mỗi vị trí trong bảng băm. Việc này có thể được thực hiện nhờ phương thức **`clear()`** của `List`.

Chúng ta còn có thể sử dụng các phương thức trong đóng gói `List` để truy xuất bảng băm. Bản thân hàm băm không khác so với bảng băm theo phương pháp địa chỉ mở. Để truy xuất dữ liệu, chúng ta có thể đơn giản sử dụng phiên bản liên kết của hàm **`sequential_search`** trong phần 7.2. Cốt lõi của phương thức **`Hash_table::retrieve`** là

```
sequential_search( table[hash(target)], target, position);
```

Chi tiết của việc chuyển đổi hàm này thành một hàm đầy đủ được xem như bài tập. Tương tự, cốt lõi của việc thêm dữ liệu vào bảng băm là

```
table[hash(new_entry)].insert( 0, new_entry );
```

Ở đây chúng ta chọn cách khi thêm phần tử mới vào thì nó sẽ được đứng tại vị trí đầu của danh sách liên kết, do đây là cách dễ nhất. Như chúng ta đã thấy, cả hai việc thêm vào và truy xuất phần tử này đều đơn giản hơn là phương pháp địa chỉ mở, do việc giải quyết đụng độ không còn là vấn đề nữa.

Việc loại phần tử ra khỏi bảng băm nối kết cũng đơn giản hơn rất nhiều so với bảng băm địa chỉ mở. Để loại một phần tử với một khóa cho trước, chúng ta chỉ cần tìm tuần tự phần tử đó trong danh sách liên kết có chứa nó và loại nó ra khỏi danh sách này. Đặc tả của phương thức loại bỏ như sau:

```
Error_code Hash_table::remove(const Key &target, Record &x);
```
post: Nếu bảng có chứa phần tử có khóa bằng `target`, thì phần tử này được chép vào `x` và được loại khỏi bảng băm, phương thức trả về `success`. Ngược lại phương thức trả về `not_present`.

Hiện thực của phương thức này cũng được dành lại như bài tập.

12.6. Phân tích bảng băm

12.6.1. Điều ngạc nhiên về ngày sinh

Khả năng xảy ra đụng độ trong việc băm có liên quan đến một chuyện vui khá nổi tiếng trong toán học: nếu chọn một cách ngẫu nhiên từng người để đưa vào một căn phòng thì sẽ được bao nhiêu người trước khi có thể xảy ra việc hai người trong số đó có cùng một ngày sinh. Do mỗi năm có `365` ngày, nhiều người đoán rằng câu trả lời phải lên đến con số hàng trăm, nhưng lời giải thực ra là chỉ có `23` người.

Chúng ta có thể định ra xác suất cho câu hỏi này bằng cách trả lời theo hướng ngược lại: Với m người được chọn một cách ngẫu nhiên để đưa vào phòng, xác suất để hai người có cùng ngày sinh là bao nhiêu? Chúng ta hãy bắt đầu với bất kỳ

người nào và đáng dấu loại ngày sinh của họ trên lịch. Xác suất để người thứ hai có ngày sinh khác với người đã chọn là `364/365`. Tiếp tục đánh dấu loại ngày sinh của người này chúng ta có xác suất để người thứ ba có ngày sinh khác sẽ là `363/365`. Tiếp tục tiến hành theo cách này, chúng ta sẽ thấy nếu m-1 người đầu tiên có các ngày sinh khác nhau, thì xác suất người thứ m có ngày sinh khác nữa là `(365 – m + 1) / 365`. Do các ngày sinh của những người khác nhau là độc lập nhau, các xác suất này sẽ được nhân với nhau, và chúng ta có được xác suất để m người có các ngày sinh không trùng nhau là

$$\frac{364}{365} \text{ x } \frac{363}{365} \text{ x } \frac{362}{365} \text{ x } \ldots \text{ x } \frac{365\text{-m+1}}{365}$$

Biểu thức này sẽ nhỏ hơn `0.5` khi $m \geq 23$.

Khi xét đến bảng băm, điều đáng ngạc nhiên về các ngày sinh trên đây cho chúng ta biết rằng với bất kỳ một kích thước nào thì hầu như sự đụng độ cũng chắc chắn sẽ xảy ra. Vì thế, cách tiếp cận của chúng ta không nên chỉ dừng lại ở việc làm giảm số lần đụng độ, mà còn phải xử lý chúng càng hiệu quả càng tốt.

12.6.2. Đếm số lần thử

Cũng như những phương pháp truy xuất thông tin khác, chúng ta muốn biết số lần so sánh trung bình của các khóa trong cả hai trường hợp tìm kiếm thành công và không thành công đối với một khóa cho trước. Chúng ta sẽ dùng từ thử (*probe*) cho việc xem xét một phần tử và so sánh khóa của nó với khóa cần tìm.

Số lần thử cần thiết phụ thuộc vào mức độ đầy của bảng. Do đó (cũng như các phương pháp tìm kiếm), chúng ta gọi n là số phần tử trong một bảng và t (cũng là `hash_size`) là số vị trí trong mảng chứa bảng băm. **Hệ số tải** (*load factor*) của bảng sẽ là $\lambda = n/t$; $\lambda = 0$ có nghĩa là bảng rỗng; $\lambda = 0.5$ là bảng chứa một nửa số phần tử. Đối với bảng địa chỉ mở, λ không bao giờ có thể vượt quá 1, nhưng đối với bảng nối kết sẽ không có giới hạn cho λ. Chúng ta sẽ xem xét riêng từng bảng trên.

12.6.3. Phân tích phương pháp nối kết

Với một bảng nối kết chúng ta đi trực tiếp đến một trong các danh sách liên kết trước khi thực hiện bất kỳ một phép thử nào. Giả sử như danh sách có chứa khóa cần tìm có k phần tử. Chú ý rằng k có thể bằng 0.

Nếu việc tìm kiếm không thành công, thì khóa cần tìm sẽ phải được so sánh với tất cả k khóa của k phần tử tương ứng. Do các phần tử được phân phối một cách như nhau trên tất cả t danh sách (xác suất xuất hiện bằng nhau trên mọi

danh sách), số phần tử được mong đợi trong danh sách đang được tìm kiếm là $\lambda = n/t$. Do đó số lần thử trung bình của một lần tìm kiếm không thành công là λ.

Bây giờ chúng ta hãy giả sử là việc tìm kiếm sẽ thành sông. Từ phân tích của việc tìm tuần tự, chúng ta đã biết rằng số lần so sánh trung bình là $\frac{1}{2}(k+1)$, với k là chiều dài của danh sách chứa phần tử cần tìm. Nhưng chiều dài mong đợi của danh sách này không lớn hơn λ, và chúng ta biết trước là nó chứa ít nhất một phần tử (phần tử cần tìm). Ngoại trừ phần tử cần tìm, $n-1$ phần tử còn lại được phân phối như nhau trên tất cả t danh sách; vậy số phần tử mong đợi trên danh sách có chứa phần tử cần tìm là $1+(n-1)/t$. Không kể các bảng có kích thước nhỏ, chúng ta lấy xấp xỉ $(n-1)/t$ bằng $n/t=\lambda$. Vậy số lần thử trung bình cho một lần tìm kiếm thành công gần với

$$\frac{1}{2}(k+1) \approx \frac{1}{2}(1 + \lambda + 1) = 1 + \frac{1}{2}\lambda.$$

Tóm lại, việc truy xuất một bảng băm nối kết có hệ số tải λ trung bình cần đến $1 + \frac{1}{2}\lambda$ lần thử cho một lần tìm kiếm thành công và λ lần thử cho một lần tìm kiếm không thành công.

12.6.4. Phân tích phương pháp địa chỉ mở

Để phân tích số lần thử trong bảng băm địa chỉ mở, trước hết chúng ta bỏ qua vấn đề gom tụ với giả thiết rằng không chỉ lần thử đầu tiên là ngẫu nhiên mà ngay cả sau khi xảy ra đụng độ, lần thử kế tiếp cũng ngẫu nhiên trên khắp các vị trí còn lại của bảng. Nói cách khác, giả sử rằng bảng băm có kích thước rất lớn sao cho mọi lần thử có thể được xem là độc lập nhau. Kết quả tính được như sau:

Việc truy xuất từ bảng băm địa chỉ mở, với phép thử ngẫu nhiên và hệ số tải λ, có số lần thử trung bình xấp xỉ bằng

$$\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$$

trong trường hợp tìm kiếm thành công và $1/(1-\lambda)$ trong trường hợp tìm kiếm không thành công.

Trong trường hợp bảng băm địa chỉ mở với phép thử tuyến tính, lưu ý rằng giả thiết các lần thử độc lập nhau là không thể chấp nhận. Kết quả tính được như sau:

Việc truy xuất bảng băm địa chỉ mở với phép thử tuyến tính và hệ số tải λ cần số lần thử trung bình xấp xỉ bằng

$$\frac{1}{2}\left[1 + \frac{1}{1 - \lambda}\right]$$

trong trường hợp thành công và

$$\frac{1}{2}\left[1 + \frac{1}{(1 - \lambda)^2}\right]$$

trong trường hợp không thành công.

12.6.5. Các so sánh lý thuyết

Load factor	0.10	0.50	0.80	0.90	0.99	2.00
Successful search, expected number of probes:						
Chaining	1.05	1.25	1.40	1.45	1.50	2.00
Open, Random probes	1.05	1.4	2.0	2.6	4.6	—
Open, Linear probes	1.06	1.5	3.0	5.5	50.5	—
Unsuccessful search, expected number of probes:						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Random probes	1.1	2.0	5.0	10.0	100.	—
Open, Linear probes	1.12	2.5	13.	50.	5000.	—

Hình 12.13 – So sánh lý thuyết các phương pháp băm

Hình 12.13 cho thấy các giá trị của các biểu thức trên với các trị khác nhau của hệ số tải λ.

Chúng ta có thể thấy được một vài kết luận từ bảng này. Trước hết, rõ ràng là bảng băm nối kết cần ít lần thử hơn bảng băm địa chỉ mở. Mặt khác, việc duyệt các danh sách liên kết thường chậm hơn là truy xuất mảng, điều này làm giảm ưu điểm của bảng băm nối kết, nhất là trong những trường hợp mà việc so sánh khóa có thể được thực hiện rất nhanh. Phương pháp nối kết trở nên hợp lý khi các phần tử có kích thước lớn và thời gian cần để so sánh các khóa là nhiều. Ngoài ra, nó còn tỏ ra có lợi thế khi việc tìm kiếm không thành công thường xảy ra, do những lúc quá trình tìm kiếm gặp được một danh sách rỗng hoặc một danh sách thật ngắn và có thể kết thúc nhanh với rất ít lần so sánh khóa.

Đối với việc tìm kiếm thành công trong bảng băm địa chỉ mở, phương pháp thử tuyến tính đơn giản không làm chậm quá trình tìm kiếm đi nhiều so với các phương pháp giải quyết đụng độ phức tạp khác, ít nhất là cho đến khi bảng gần

như đầy. Tuy nhiên, đối với việc tìm kiếm không thành công, hiện tượng gom tụ nhanh chóng làm cho phép thử tuyến tính suy thoái thành quá trình tìm kiếm tuần tự. Vì thế, chúng ta có thể kết luận rằng nếu việc tìm kiếm thường thành công, và hệ số tải vừa phải, thì phép thử tuyến tính sẽ đáp ứng được; còn trong những trường hợp khác, nên sử dụng những phương pháp giải quyết đụng độ khác như phương pháp thử bậc hai chẳng hạn.

12.6.6. Các so sánh thực nghiệm

Một điều quan trọng cần nhớ là các tính toán trong hình 12.13 chỉ là các con số xấp xỉ, và trong thực tế không có gì là hoàn toàn ngẫu nhiên, do đó chúng ta luôn biết rằng sẽ có một vài điều khác nhau giữa các kết quả lý thuyết và việc tính toán thực sự. Vì vậy, để so sánh, hình 12.14 cho thấy kết quả của việc nghiên cứu bằng thực nghiệm với 900 khóa lấy ngẫu nhiên giữa 0 và 1.

Nếu so sánh các con số trong hai bảng 12.15 và 12.16, chúng ta thấy rằng kết quả thực nghiệm trên bảng băm nối kết gần giống với kết quả lý thuyết. Các kết quả của phép thử bậc hai lại gần giống với kết quả lý thuyết của việc thử ngẫu nhiên; sự khác nhau có thể được giải thích dễ dàng là vì thử bậc hai chưa thật sự ngẫu nhiên. Đối với thử tuyến tính, các kết quả tương tự khi bảng còn tương đối trống, nhưng khi bảng gần như đầy thì các con số xấp xỉ được tính bằng lý thuyết khác nhiều so với thực nghiệm. Đó là hậu quả của các giả thiết đã được đơn giản hóa trong toán học.

Load factor	0.1	0.5	0.8	0.9	0.99	2.0
Successful search, average number of probes:						
Chaining	1.04	1.2	1.4	1.4	1.5	2.0
Open, Quadratic probes	1.04	1.5	2.1	2.7	5.2	—
Open, Linear probes	1.05	1.6	3.4	6.2	21.3	—
Unsuccessful search, average number of probes:						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Quadratic probes	1.13	2.2	5.2	11.9	126.	—
Open, Linear probes	1.13	2.7	15.4	59.8	430.	—

Hình 12.14 – So sánh thực nghiệm các phương pháp băm.

So sánh với các phương pháp truy xuất thông tin khác, điều quan trọng cần lưu ý về tất cả những con số này là chúng chỉ phụ thuộc vào hệ số tải, mà không phụ thuộc vào số phần tử thực sự có trong bảng. Việc truy xuất từ bảng băm có 20,000 phần tử trong 40,000 vị trí có thể có của bảng, xét trung bình, không chậm hơn

việc tìm kiếm trong 20 phần tử trong 40 vị trí có thể có. Với việc tìm tuần tự, một danh sách có kích thước lớn gấp 1000 lần sẽ làm cho quá trình tìm lâu hơn 1000 lần. Với tìm kiếm nhị phân, tỉ lệ này giảm xuống 10 (chính xác hơn là lg1000), nhưng thời gian tìm kiếm vẫn luôn phụ thuộc vào kích thước của danh sách, điều này không có ở bảng băm.

Chúng ta có thể tổng kết những khảo sát về việc truy xuất từ n phần tử như sau:

- Tìm tuần tự là $\theta(n)$.
- Tìm nhị phân là $\theta(\log n)$.
- Truy xuất bảng băm là $\theta(1)$.

Cuối cùng, chúng ta nhấn mạnh về tầm quan trọng của việc lựa chọn một hàm băm tốt, một hàm băm thực hiện tính toán nhanh và rải đều các khóa trong bảng. Nếu hàm băm không tốt thì băm sẽ suy thoái về tìm kiếm tuần tự.

12.7. Kết luận: so sánh các phương pháp

Trong chương này và chương 7, chúng ta đã xem xét bốn phương pháp khác nhau để truy xuất thông tin:

- Tìm tuần tự,
- Tìm nhị phân,
- Tra cứu bảng, và
- Băm.

Nếu được hỏi rằng phương pháp nào là tốt nhất, trước hết chúng ta cần chọn ra các tiêu chí để đánh giá. Các tiêu chí gồm các yêu cầu của ứng dụng, và các mối quan tâm khác có ảnh hưởng lên sự chọn lựa cấu trúc dữ liệu, do hai phương pháp đầu chỉ có thể áp dụng với các danh sách còn hai phương pháp sau chỉ dành cho các bảng. Trong nhiều ứng dụng, chúng ta cũng được tự do trong việc chọn lựa giữa danh sách và bảng.

Về mặt tốc độ cũng như tính thuận lợi, việc tra cứu theo thứ tự trong các bảng chắc chắn là tốt nhất, nhưng có nhiều ứng dụng mà điều này lại không áp dụng được do tập các khóa khá thưa thớt hoặc đối với chúng danh sách tỏ ra ưu thế hơn. Một điều không thích ứng nữa là khi việc thêm và loại phần tử xảy ra thường xuyên, trong vùng nhớ liên tục các tác vụ này đòi hỏi phải di chuyển một số lớn dữ liệu.

Trong ba phương pháp còn lại, phương pháp nào là tốt nhất phụ thuộc vào tiêu chí khác như dạng của dữ liệu.

Tìm tuần tự là phương pháp mềm dẻo nhất trong các phương pháp. Dữ liệu có thể được lưu theo bất kỳ thứ tự nào, trong hiện thực liên tục hoặc liên kết. Tìm nhị phân đòi hỏi nhiều hơn, các khóa phải lưu theo thứ tự và dữ liệu phải lưu trong vùng nhớ cho phép truy xuất ngẫu nhiên (vùng nhớ liên tục). Băm còn đòi hỏi nhiều hơn nữa, tuy thứ tự khác thường của các khóa vẫn đáp ứng được việc truy xuất từ bảng băm, nhưng nói chung nó không có lợi cho bất kỳ một mục đích nào khác. Nếu như dữ liệu cần phải luôn sẵn sàng cho một sự khảo sát thì một thứ tự nào đó là cần thiết, và như vậy bảng băm không đáp ứng.

Cuối cùng là vấn đề liên quan đến việc tìm kiếm không thành công. Tìm tuần tự và băm khi không thành công thì xem như không có kết quả gì. Trong khi đó, nếu thất bại thì tìm nhị phân sẽ cho biết dữ liệu có khóa gần với khóa cần tìm, và như vậy nó có thể cung cấp thông tin hữu ích. Trong chương 9 và 10 chúng ta đã nghiên cứu các phương pháp lưu trữ dữ liệu dựa trên cơ sở cây, có kết hợp tính hiệu quả của tìm nhị phân với sự mềm dẻo của các cấu trúc liên kết.

Chương 13 – ĐỒ THỊ

Chương này trình bày về các cấu trúc toán học quan trọng được gọi là đồ thị. Đồ thị thường được ứng dụng trong rất nhiều lĩnh vực: điều tra xã hội, hóa học, địa lý, kỹ thuật điện,.... Chúng ta sẽ tìm hiểu các phương pháp biểu điễn đồ thị bằng các cấu trúc dữ liệu và xây dựng một số giải thuật tiêu biểu liên quan đến đồ thị.

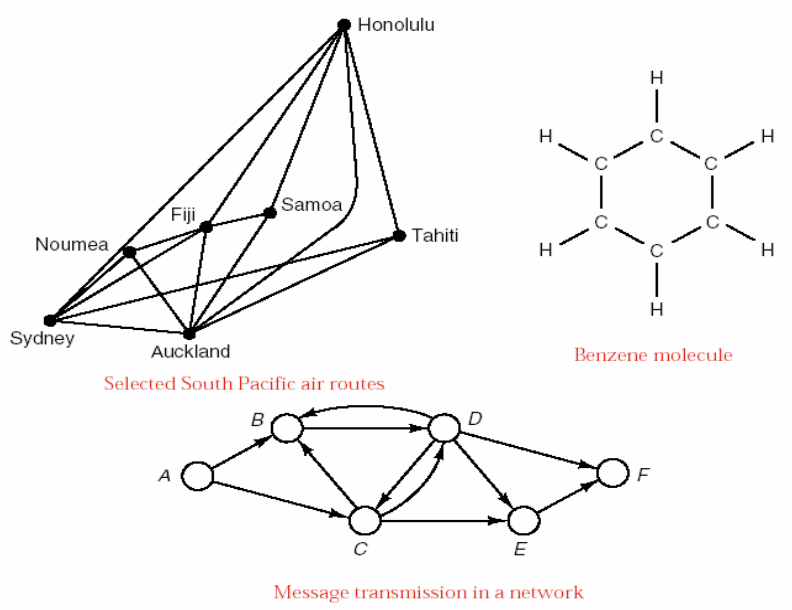
13.1. Nền tảng toán học

13.1.1. Các định nghĩa và ví dụ

Một đồ thị (*graph*) G gồm một tập V chứa các đỉnh của đồ thị, và tập E chứa các cặp đỉnh khác nhau từ V. Các cặp đỉnh này được gọi là các cạnh của G. Nếu $e = (\nu, \mu)$ là một cạnh có hai đỉnh ν và μ, thì chúng ta gọi ν và μ nằm trên e, và e nối với ν và μ. Nếu các cặp đỉnh không có thứ tự, G được gọi là đồ thị vô hướng (*undirected graph*), ngược lại, G được gọi là đồ thị có hướng (*directed graph*). Thông thường đồ thị có hướng được gọi tắt là *digraph*, còn từ *graph* thường mang nghĩa là đồ thị vô hướng. Cách tự nhiên để vẽ đồ thị là biểu diễn các đỉnh bằng các điểm hoặc vòng tròn, và các cạnh bằng các đường thẳng hoặc các cung nối các đỉnh. Đối với đồ thị có hướng thì các đường thẳng hay các cung cần có mũi tên chỉ hướng. Hình 13.1 minh họa một số ví dụ về đồ thị.

Đồ thị thứ nhất trong hình 13.1 có các thành phố là các đỉnh, và các tuyến bay là các cạnh. Trong đồ thị thứ hai, các nguyên tử *hydro* và *carbon* là các đỉnh, các liên kết hóa học là các cạnh. Hình thứ ba là một đồ thị có hướng cho biết khả năng truyền nhận dữ liệu trên mạng, các nút của mạng (A, B, ..., F) là các đỉnh và các đường nối các nút là có hướng. Đôi khi cách chọn tập đỉnh và tập cạnh cho đồ thị phụ thuộc vào giải thuật mà chúng ta dùng để giải bài toán, chẳng hạn bài toán liên quan đến quy trình công việc, bài toán xếp thời khóa biểu,...

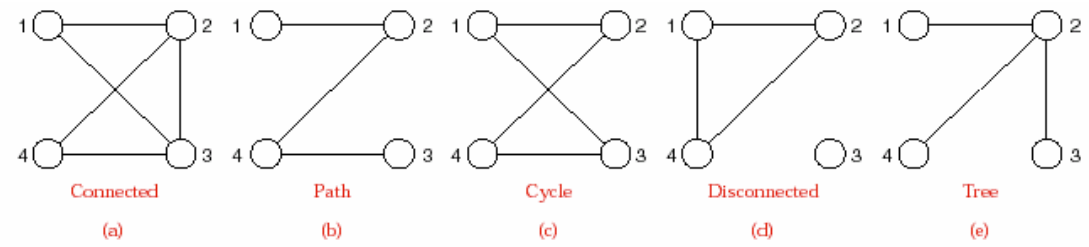
Đồ thị được sử dụng để mô hình hóa rất nhiều dạng quá trình cũng như cấu trúc khác nhau. Đồ thị có thể biểu diễn mạng giao thông giữa các thành phố, hoặc các thành phần của một mạch in điện tử và các đường nối giữa chúng, hoặc cấu trúc của một phân tử gồm các nguyên tử và các liên kết hóa học. Những người dân trong một thành phố cũng có thể được biểu diễn bởi các đỉnh của đồ thị mà các cạnh là các mối quan hệ giữa họ. Nhân viên trong một công ty có thể được biểu diễn trong một đồ thị có hướng mà các cạnh có hướng cho biết mối quan hệ của họ với những người quản lý. Những người này cũng có thể có những mối quan hệ "cùng làm việc" biểu diễn bởi các cạnh không hướng trong một đồ thị vô hướng.



Hình 13.1 – Các ví dụ về đồ thị

13.1.2. Đồ thị vô hướng

Một vài dạng của đồ thị vô hướng được minh họa trong hình 13.2. Hai đỉnh trong một đồ thị vô hướng được gọi là kề nhau (*adjacent*) nếu tồn tại một cạnh nối từ đỉnh này đến đỉnh kia. Trong đồ thị vô hướng trong hình 13.2 a, đỉnh 1 và 2 là kề nhau, đỉnh 3 và 4 là kề nhau, nhưng đỉnh 1 và đỉnh 4 không kề nhau. Một đường đi (*path*) là một dãy các đỉnh khác nhau, trong đó mỗi đỉnh kề với đỉnh kế tiếp. Hình (b) cho thấy một đường đi. Một chu trình (*cycle*) là một đường đi chứa ít nhất ba đỉnh sao cho đỉnh cuối cùng kề với đỉnh đầu tiên. Hình (c) là một chu trình. Một đồ thị được gọi là liên thông (*connected*) nếu luôn có một đường đi từ một đỉnh bất kỳ đến một đỉnh bất kỳ nào khác. Hình (a), (b), và (c) là các đồ thị liên thông. Hình (d) không phải là đồ thị liên thông. Nếu một đồ thị là không liên thông, chúng ta xem mỗi tập con lớn nhất các đỉnh liên thông nhau như một thành phần liên thông. Ví dụ, đồ thị không liên thông ở hình (d) có hai thành phần liên thông: một thành phần chứa các đỉnh 1,2 và 4; một thành phần chỉ có đỉnh 3.

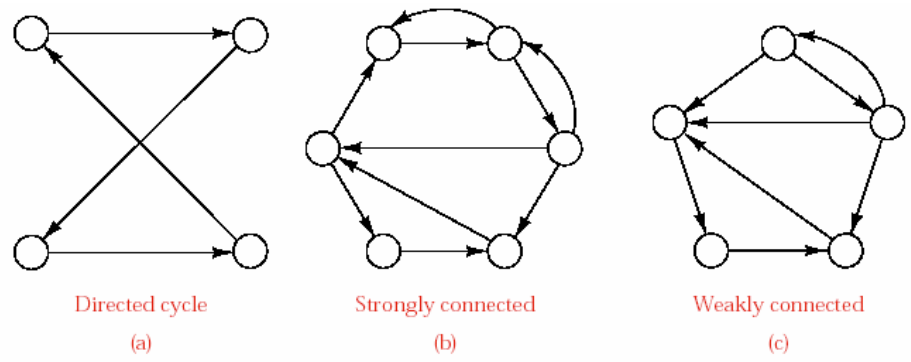


Hình 13.2 – Các dạng của đồ thị vô hướng

Phần (e) là một đồ thị liên thông không có chu trình. Chúng ta có thể nhận thấy đồ thị cuối cùng này thực sự là một cây, và chúng ta dùng đặc tính này để định nghĩa: Một cây tự do (*free tree*) được định nghĩa là một đồ thị vô hướng liên thông không có chu trình.

13.1.3. Đồ thị có hướng

Đối với các đồ thị có hướng, chúng ta có thể có những định nghĩa tương tự. Chúng ta yêu cầu mọi cạnh trong một đường đi hoặc một chu trình đều có cùng hướng, như vậy việc lần theo một đường đi hoặc một chu trình có nghĩa là phải di chuyển theo hướng chỉ bởi các mũi tên. Những đường đi (hay chu trình) như vậy được gọi là đường đi có hướng (hay chu trình có hướng). Một đồ thị có hướng được gọi là liên thông mạnh (*strongly connected*) nếu nó luôn có một đường đi có hướng từ một đỉnh bất kỳ đến một đỉnh bất kỳ nào khác. Trong một đồ thị có hướng không liên thông mạnh, nếu bỏ qua chiều của các cạnh mà chúng ta có được một đồ thị vô hướng liên thông thì đồ thị có hướng ban đầu được gọi là đồ thị liên thông yếu (*weakly connected*). Hình 13.3 minh họa một chu trình có hướng, một đồ thị có hướng liên thông mạnh và một đồ thị có hướng liên thông yếu.



Hình 13.3 – Các ví dụ về đồ thị có hướng

Các đồ thị có hướng trong phần (b) và (c) hình 13.3 có các cặp đỉnh có các cạnh có hướng theo cả hai chiều giữa chúng. Các cạnh có hướng là các cặp có thứ tự và các cặp có thứ tự (ν, μ) và (μ,ν) là khác nhau nếu $\nu \neq \mu$. Trong đồ thị vô hướng, chỉ có thể có nhiều nhất một cạnh nối hai đỉnh khác nhau. Tương tự, do các đỉnh trên một cạnh theo định nghĩa là phải khác nhau, không thể có một cạnh nối một đỉnh với chính nó. Tuy nhiên, cũng có những trường hợp mở rộng định nghĩa, người ta cho phép nhiều cạnh nối một cặp đỉnh, và một cạnh nối một đỉnh với chính nó.

13.2. Biểu diễn bằng máy tính

Nếu chúng ta chuẩn bị viết chương trình để giải quyết một bài toán có liên quan đến đồ thị, trước hết chúng ta phải tìm cách để biểu diễn cấu trúc toán học của đồ thị như là một dạng nào đó của cấu trúc dữ liệu. Có nhiều phương pháp

được dùng phổ biến, về cơ bản chúng khác nhau trong việc lựa chọn kiểu dữ liệu trừu tượng để biểu diễn đồ thị, cũng như nhiều cách hiện thực khác nhau cho mỗi kiểu dữ liệu trừu tượng. Nói cách khác, chúng ta bắt đầu từ một định nghĩa toán học, đó là đồ thị, sau đó chúng ta tìm hiểu cách mô tả nó như một kiểu dữ liệu trừu tượng (tập hợp, bảng, hay danh sách đều có thể dùng được), và cuối cùng chúng ta lựa chọn cách hiện thực cho kiểu dữ liệu trừu tượng mà chúng ta chọn.

13.2.1. Biểu diễn của tập hợp

Đồ thị được định nghĩa bằng một tập hợp, như vậy một cách hết sức tự nhiên là dùng tập hợp để xác định cách biểu diễn nó như là dữ liệu. Trước tiên, chúng ta có một tập các đỉnh, và thứ hai, chúng ta có các cạnh như là tập các cặp đỉnh. Thay vì thử biểu diễn tập các cặp đỉnh này một cách trực tiếp, chúng ta chia nó ra thành nhiều phần nhỏ bằng cách xem xét tập các cạnh liên quan đến từng đỉnh riêng rẽ. Nói một cách khác, chúng ta có thể biết được tất cả các cạnh trong đồ thị bằng cách nắm giữ tập E_v các cạnh có chứa v đối với mỗi đỉnh v trong đồ thị, hoặc, một cách tương đương, tập A_v gồm tất cả các đỉnh kề với v. Thật vậy, chúng ta có thể dùng ý tưởng này để đưa ra một định nghĩa mới tương đương cho đồ thị:

Định nghĩa: Một đồ thị có hướng G bao gồm tập V, gọi là các đỉnh của G, và, đối với mọi $v \in V$, có một tập con A_v, gọi là tập các đỉnh kề của v.

Từ các tập con A_v chúng ta có thể tái tạo lại các cạnh như là các cặp có thứ tự theo quy tắc sau: cặp (v, w) là một cạnh nếu và chỉ nếu $w \in A_v$. Xử lý cho tập các đỉnh dễ hơn là tập các cạnh. Ngoài ra, định nghĩa mới này thích hợp với cả đồ thị có hướng và đồ thị vô hướng. Một đồ thị là vô hướng khi nó thỏa tính chất đối xứng sau: $w \in A_v$ kéo theo $v \in A_w$ với mọi v, $w \in V$. Tính chất này có thể được phát biểu lại như sau: Một cạnh không có hướng giữa v và w có thể được xem như hai cạnh có hướng, một từ v đến w và một từ w đến v.

13.2.1.1. Hiện thực các tập hợp

Có nhiều cách để hiện thực tập các đỉnh trong cấu trúc dữ liệu và giải thuật. Cách thứ nhất là biểu diễn tập các đỉnh như là một danh sách các phần tử của nó, chúng ta sẽ tìm hiểu phương pháp này sau. Cách thứ hai, thường gọi là chuỗi các bit (*bit string*), lưu một trị *Boolean* cho mỗi phần tử của tập hợp để chỉ ra rằng nó có hay không có trong tập hợp. Để đơn giản, chúng ta sẽ xem các phần tử có thể có của tập hợp được đánh chỉ số từ 0 đến `max_set-1`, với `max_set` là số phần tử tối đa cho phép. Điều này có thể được hiện thực một cách dễ dàng bằng cách sử dụng thư viện chuẩn (*Standard Template Library- STL*)

`std::bitset<max_set>`, hoặc lớp có sử dụng `template` cho kích thước tập hợp của chúng ta như sau:

```
template <int max_set>
struct Set {
  bool is_element[max_set];
};
```

Đây chỉ là một cách hiện thực đơn giản nhất của khái niệm tập hợp. Sinh viên có thể thấy rằng không có gì ngăn cản chúng ta đặc tả và hiện thực một CTDL tập hợp với các phương thức hội, giao, hiệu, xét thành viên của nó,..., một cách hoàn chỉnh nếu như cần sử dụng tập hợp trong những bài toán lớn nào đó.

Giờ chúng ta đã có thể đặc tả cách biểu diễn thứ nhất cho đồ thị của chúng ta:

```
// Tương ứng hình 13.4-b
template <int max_size>
class Digraph {
   int count;   //  Số đỉnh của đồ thị, nhiều nhất là max_size
   Set<max_size> neighbors[max_size];
};
```

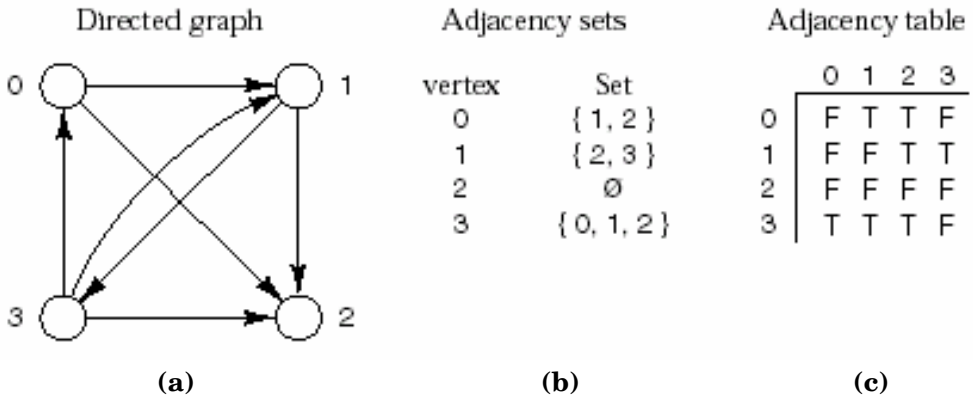
Trong cách hiện thực này, các đỉnh được đặt tên bằng các số nguyên từ 0 đến `count-1`. Nếu ν là một số nguyên thì phần tử `neighbors[ν]` của mảng là một tập các đỉnh kề với đỉnh ν.

13.2.1.2. Bảng kề

Trong cách hiện thực trên đây, cấu trúc `Set` được hiện thực như một mảng các phần tử kiểu `bool`. Mỗi phần tử chỉ ra rằng đỉnh tương ứng có là thành phần của tập hợp hay không. Nếu chúng ta thay thế tập các đỉnh kề này bằng một mảng, chúng ta sẽ thấy rằng mảng `neighbors` trong định nghĩa của lớp `Graph` có thể được biến đổi thành mảng các mảng (mảng hai chiều) như sau đây, và chúng ta gọi là bảng kề (*adjacency table*):

```
// Tương ứng hình 13.4-c
template <int max_size>
class Digraph {
   int count;   //  Số đỉnh của đồ thị, nhiều nhất là max_size.
   bool adjacency[max_size][max_size];
};
```

Bảng kề chứa các thông tin một cách tự nhiên như sau: `adjacency[v][w]` là `true` nếu và chỉ nếu đỉnh v là đỉnh kề của w. Nếu là đồ thị có hướng, `adjacency[v][w]` cho biết cạnh từ v đến w có trong đồ thị hay không. Nếu đồ thị vô hướng, bảng kề phải đối xứng, nghĩa là `adjacency[v][w]= adjacency[v][w]` với mọi v và w. Biểu diễn đồ thị bởi tập các đỉnh kề và bởi bảng kề được minh họa trong hình 13.4.



Hình 13.4 – Tập các đỉnh kề và bảng kề.

13.2.2. Danh sách kề

Một cách khác để biểu diễn một tập hợp là dùng danh sách các phần tử. Chúng ta có một danh sách các đỉnh, và, đối với mỗi đỉnh, có một danh sách các đỉnh kề. Chúng ta có thể xem xét cách hiện thực cho đồ thị bằng danh sách liên tục hoặc danh sách liên kết đơn. Tuy nhiên, đối với nhiều ứng dụng, người ta thường sử dụng các hiện thực khác của danh sách phức tạp hơn như cây nhị phân tìm kiếm, cây nhiều nhánh tìm kiếm, hoặc là `heap`. Lưu ý rằng, bằng cách đặt tên các đỉnh theo các chỉ số trong các cách hiện thực trước đây, chúng ta cũng có được cách hiện thực cho tập các đỉnh như là một danh sách liên tục.

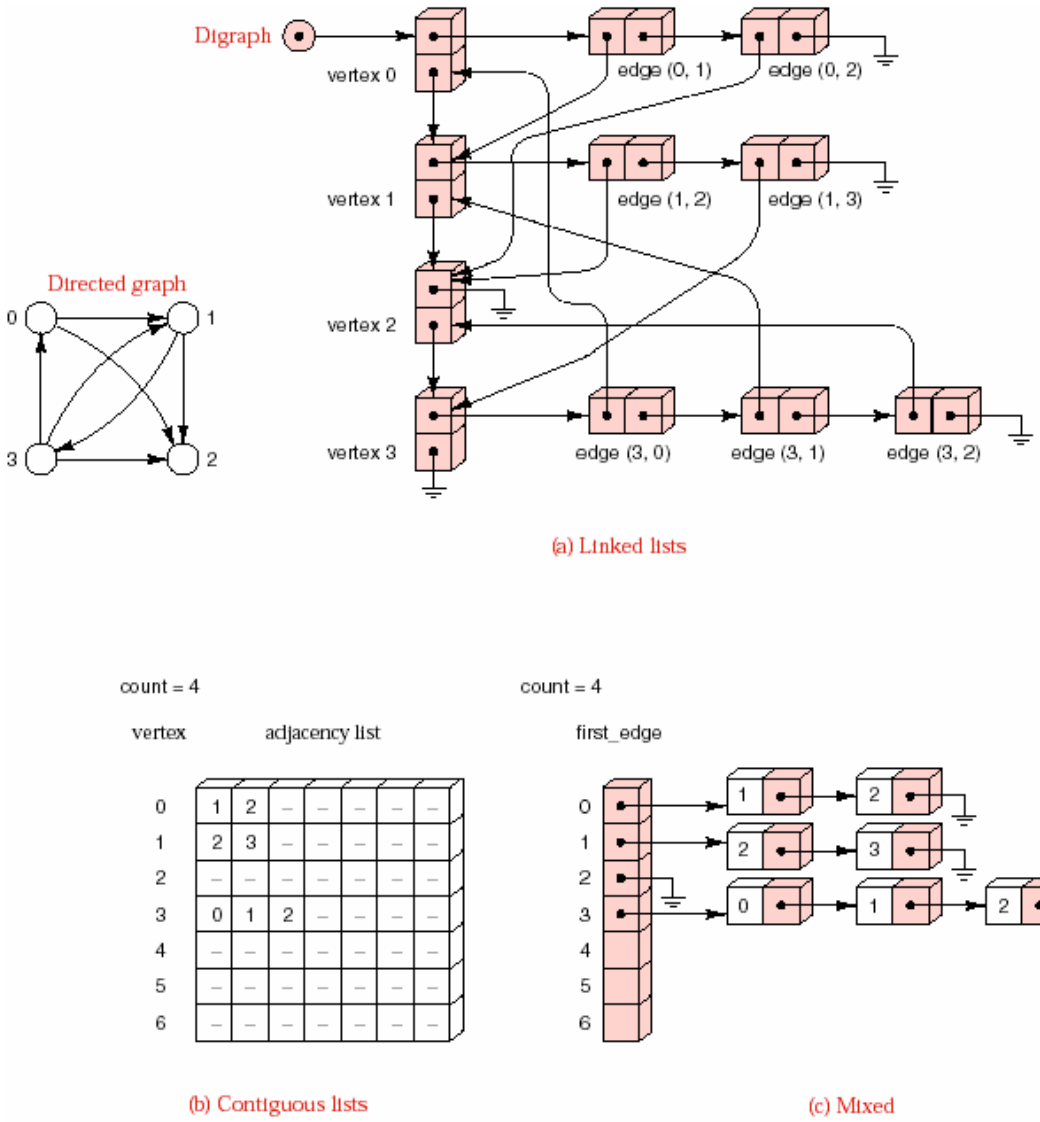
13.2.2.1. Hiện thực dựa trên cơ sở là danh sách

Chúng ta có được hiện thực của đồ thị dựa trên cơ sở là danh sách bằng cách thay thế các tập hợp đỉnh kề trước kia bằng các danh sách. Hiện thực này có thể sử dụng hoặc danh sách liên tục hoặc danh sách liên kết. Phần (b) và (c) của hình 13.5 minh họa hai cách hiện thực này.

```
// Tổng quát cho cả danh sách liên tục lẫn liên kết (hình 13.5-b và c).

typedef int Vertex;
template <int max_size>
class Digraph {
   int count;   //  Số đỉnh của đồ thị, nhiều nhất là max_size.
   List<Vertex> neighbors[max_size];
};
```

13.2.2.2. Hiện thực liên kết

Bằng cách sử dụng các đối tượng liên kết cho cả các đỉnh và cho cả các danh sách kề, đồ thị sẽ có được tính linh hoạt cao nhất. Hiện thực này được minh họa trong hình 13.5-a và có các định nghĩa như sau:



Hình 13.5 – Hiện thực đồ thị bằng các danh sách

```
class Edge;
class Vertex {
   Edge *first_edge;     //  Chỉ đến phần tử đầu của DSLK các đỉnh kề.
   Vertex *next_vertex; //  Chỉ đến phần tử kế trong DSLK các đỉnh có trong đồ thị.
};

class Edge {
   Vertex *end_point;   //  Chỉ đến một đỉnh kề với đỉnh mà danh sách này thuộc về.
   Edge *next_edge;     //  Chỉ đến phần tử biểu diễn đỉnh kề kế tiếp trong danh sách các
                                  đỉnh kề với một đỉnh mà danh sách này thuộc về.
};
```

```
class Digraph {
   Vertex *first_vertex;//  Chỉ đến phần tử đầu tiên trong danh sách các đỉnh của đồ thị.
};
```

13.2.3. Các thông tin khác trong đồ thị

Nhiều ứng dụng về đồ thị không những cần những thông tin về các đỉnh kề của một đỉnh mà còn cần thêm một số thông tin khác liên quan đến các đỉnh cũng như các cạnh. Trong hiện thực liên kết, các thông tin này có thể được lưu như các thuộc tính bổ sung bên trong các bản ghi tương ứng, và trong hiện thực liên tục, chúng có thể được lưu trong các mảng các phần tử bên trong các bản ghi. Lấy ví dụ trường hợp mạng các máy tính, nó được định nghĩa như một đồ thị trong đó mỗi cạnh có thêm thông tin là tải trọng của đường truyền từ máy này qua máy khác. Đối với nhiều giải thuật trên mạng, cách biểu diễn tốt nhất là dùng bảng kề, trong đó các phần tử sẽ chứa tải trọng thay vì một trị kiểu `bool`. Chúng ta sẽ quay lại vấn đề này sau trong chương này.

13.3. Duyệt đồ thị

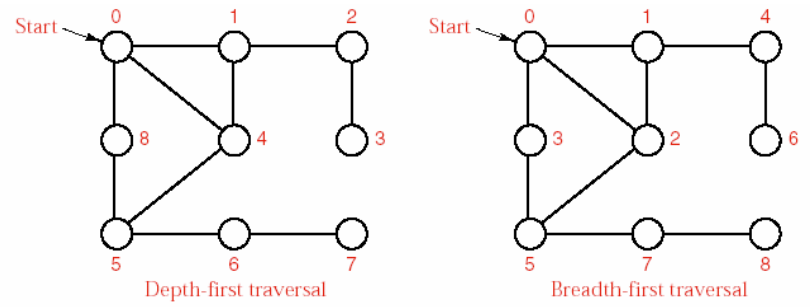
13.3.1. Các phương pháp

Trong nhiều bài toán, chúng ta mong muốn được khảo sát các đỉnh trong đồ thị theo một thứ tự nào đó. Tựa như đối với cây nhị phân chúng ta đã phát triển một vài phương pháp duyệt qua các phần tử một cách có hệ thống. Khi duyệt cây, chúng ta thường bắt đầu từ nút gốc. Trong đồ thị, thường không có đỉnh nào là đỉnh đặc biệt, nên việc duyệt qua đồ thị có thể bắt đầu từ một đỉnh bất kỳ nào đó. Tuy có nhiều thứ tự khác nhau để duyệt qua các đỉnh của đồ thị, có hai phương pháp được xem là đặc biệt quan trọng.

Phương pháp duyệt theo chiều sâu (*depth-first traversal*) trên một đồ thị gần giống với phép duyệt *preorder* cho một cây có thứ tự. Giả sử như phép duyệt vừa duyệt xong đỉnh v, và gọi w_1, w_2,...,w_k là các đỉnh kề với v, thì w_1 là đỉnh được duyệt kế tiếp, trong khi các đỉnh w_2,...,w_k sẽ nằm đợi. Sau khi duyệt qua đỉnh w_1 chúng ta sẽ duyệt qua tất cả các đỉnh kề với w_1, trước khi quay lại với w_2,...,w_k.

Phương pháp duyệt theo chiều rộng (*breadth-first traversal*) trên một đồ thị gần giống với phép duyệt theo mức (*level by level*) cho một cây có thứ tự. Nếu phép duyệt vừa duyệt xong đỉnh v, thì tất cả các đỉnh kề với v sẽ được duyệt tiếp sau đó, trong khi các đỉnh kề với các đỉnh này sẽ được đặt vào một danh sách chờ, chúng sẽ được duyệt tới chỉ sau khi tất cả các đỉnh kề với v đã được duyệt xong.

Hình 13.6 minh họa hai phương pháp duyệt trên, các con số tại các đỉnh biểu diễn thứ tự mà chúng được duyệt đến.



Hình 13.6 - Duyệt đồ thị

13.3.2. Giải thuật duyệt theo chiều sâu

Phương pháp duyệt theo chiều sâu thường được xây dựng như một giải thuật đệ quy. Các công việc cần làm khi gặp một đỉnh v là:

```
visit(v);
for (mỗi đỉnh w kề với đỉnh v)
    traverse(w);
```

Tuy nhiên, trong phép duyệt đồ thị, có hai điểm khó khăn mà trong phép duyệt cây không có. Thứ nhất, đồ thị có thể chứa chu trình, và giải thuật của chúng ta có thể gặp lại một đỉnh lần thứ hai. Để ngăn chặn đệ quy vô tận, chúng ta dùng một mảng các phần tử kiểu `bool` **`visited`**, `visited[v]` sẽ là `true` khi v vừa được duyệt xong, và chúng ta luôn xét trị của `visited[w]` trước khi xử lý cho w, nếu trị này đã là `true` thì w không cần xử lý nữa. Điều khó khăn thứ hai là, đồ thị có thể không liên thông, và giải thuật duyệt có thể không đạt được đến tất cả các đỉnh của đồ thị nếu chỉ bắt đầu đi từ một đỉnh. Do đó chúng ta cần thực hiện một vòng lặp để có thể bắt đầu từ mọi đỉnh trong đồ thị, nhờ vậy chúng ta sẽ không bỏ sót một đỉnh nào. Với những phân tích trên, chúng ta có phác thảo của giải thuật duyệt đồ thị theo chiều sâu dưới đây. Chi tiết hơn cho giải thuật còn phụ thuộc vào cách chọn lựa hiện thực của đồ thị và các đỉnh, và chúng ta để lại cho các chương trình ứng dụng.

```
template <int max_size>
void Digraph<max_size>::depth_first(void (*visit)(Vertex &)) const
/*
post: Hàm *visit được thực hiện tại mỗi đỉnh của đồ thị một lần, theo thứ tự duyệt theo chiều
      sâu.
uses: Hàm traverse thực hiện duyệt theo chiều sâu.
*/
```

```
{
   bool visited[max_size];
   Vertex v;
   for (all v in G) visited[v] = false;
   for (all v in G) if (!visited[v])
      traverse(v, visited, visit);
}
```

Việc đệ quy được thực hiện trong hàm phụ trợ **traverse**. Do hàm này cần truy nhập vào cấu trúc bên trong của đồ thị, nó phải là hàm thành viên của lớp Digraph. Ngoài ra, do traverse là một hàm phụ trợ và chỉ được sử dụng trong phương thức depth_first, nó nên được khai báo private bên trong lớp.

```
template <int max_size>
void Digraph<max_size>::traverse(Vertex &v, bool visited[],
                                 void (*visit)(Vertex &)) const
/*
pre:  v là một đỉnh của đồ thị Digraph.
post: Duyệt theo chiều sâu, hàm *visit sẽ được thực hiện tại v và tại tất cả các đỉnh có thể
      đến được từ v.
uses: Hàm traverse một cách đệ quy.
*/
{  Vertex w;
   visited[v] = true;
   (*visit)(v);
   for (all w adjacent to v)
     if (!visited[w])
        traverse(w, visited, visit);
}
```

13.3.3. Giải thuật duyệt theo chiều rộng

Do sử dụng đệ quy và lập trình với ngăn xếp về bản chất là tương đương, chúng ta có thể xây dựng giải thuật duyệt theo chiều sâu bằng cách sử dụng ngăn xếp. Khi một đỉnh đang được duyệt thì các đỉnh kề của nó được đẩy vào ngăn xếp, khi một đỉnh vừa được duyệt xong thì đỉnh kế tiếp cần duyệt là đỉnh được lấy ra từ ngăn xếp. Giải thuật duyệt theo chiều rộng cũng tương tự như giải thuật vừa được đề cập đến trong việc duyệt theo chiều sâu, tuy nhiên hàng đợi cần được sử dụng thay cho ngăn xếp.

```
template <int max_size>
void Digraph<max_size>::breadth_first(void (*visit)(Vertex &)) const
/*
post: Hàm *visit được thực hiện tại mỗi đỉnh của đồ thị một lần, theo thứ tự duyệt theo chiều
      rộng.
uses: Các phương thức của lớp Queue.
*/
{  Queue q;
   bool visited[max_size];
   Vertex v, w, x;
   for (all v in G) visited[v] = false;
```

```
   for (all v in G)
     if (!visited[v]) {
        q.append(v);
        while (!q.empty()){
           q.retrieve(w);
           if (!visited[w]) {
              visited[w] = true;
              (*visit)(w);
              for (all x adjacent to w)
                 q.append(x);
           }
           q.serve();
        }
     }
}
```

13.4. Sắp thứ tự topo

13.4.1. Đặt vấn đề

Nếu G là một đồ thị có hướng không có chu trình, thì thứ tự topo (*topological order*) của G là một cách liệt kê tuần tự mọi đỉnh trong G sao cho, với mọi ν, $\mu \in G$, nếu có một cạnh từ ν đến μ, thì ν nằm trước μ.

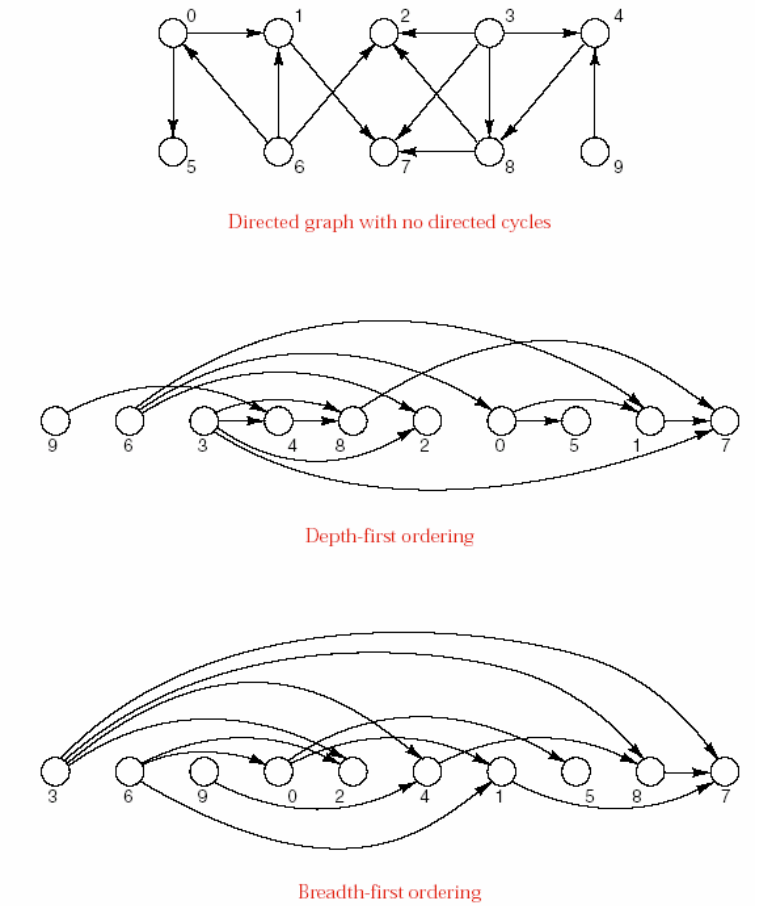
Trong suốt phần này, chúng sẽ chỉ xem xét các đồ thị có hướng không có chu trình. Thuật ngữ *acyclic* có nghĩa là một đồ thị không có chu trình. Các đồ thị như vậy xuất hiện trong rất nhiều bài toán. Như một ví dụ đầu tiên về thứ tự topo, chúng ta hãy xem xét các môn học trong một trường đại học như là các đỉnh của đồ thị, trong đó một cạnh nối từ môn này đến môn kia có nghĩa là môn thứ nhất là môn tiên quyết của môn thứ hai. Như vậy thứ tự topo sẽ liệt kê tất cả các môn sao cho mọi môn tiên quyết của một môn sẽ nằm trước môn đó. Ví dụ thứ hai là từ điển các thuật ngữ kỹ thuật. Các từ trong từ điển được sắp thứ tự sao cho không có từ nào được sử dụng trong một định nghĩa của từ khác trước khi chính nó được định nghĩa. Tương tự, các tác giả của các sách sử dụng thứ tự topo cho các đề mục trong sách. Hai thứ tự topo khác nhau của một đồ thị có hướng được minh họa trong hình 13.7.

Chúng ta sẽ xây dựng hàm để sinh ra thứ tự topo cho các đỉnh của một đồ thị không có chu trình theo hai cách: sử dụng phép duyệt theo chiều sâu và phép duyệt theo chiều rộng. Cả hai phương pháp được dùng cho một đối tượng của lớp Digraph sử dụng hiện thực dựa trên cơ sở là danh sách. Chúng ta có đặc tả lớp như sau:

```
typedef int Vertex;

template <int graph_size>
class Digraph {
public:
   Digraph();
   void read();
   void write();
```

```
//  Các phương thức sắp thứ tự topo.
   void depth_sort(List<Vertex> &topological_order);
   void breadth_sort(List<Vertex> &topological_order);

private:
   int count;
   List <Vertex> neighbors[graph_size];
   void recursive_depth_sort(Vertex v, bool visited[],
                             List<Vertex> &topological_order);
};
```

Hàm phụ trợ **recursive_depth_sort** được sử dụng bởi phương thức **depth_sort**. Cả hai phương pháp sắp thứ tự đều sẽ tạo ra một danh sách các đỉnh của đồ thị theo thứ tự topo tương ứng.



Hình 13.7 – Các thứ tự topo của một đồ thị có hướng

13.4.2. Giải thuật duyệt theo chiều sâu

Trong thứ tự topo, mỗi đỉnh phải xuất hiện trước mọi đỉnh kề của nó trong đồ thị. Giải thuật duyệt theo chiều sâu này đặt dần các đỉnh vào mảng thứ tự topo từ phải sang trái. Bắt đầu từ một đỉnh chưa từng được duyệt đến, chúng ta cần gọi đệ quy để đến được các đỉnh mà không còn đỉnh kề, các đỉnh này sẽ được đặt vào mảng thứ tự topo ở các vị trí cuối mảng. Tính từ phải sang trái trong mảng thứ tự topo này, khi các đỉnh kề của một đỉnh đã được duyệt xong, thì chính đỉnh đó

có thể có mặt trong mảng. Đó chính là lúc các lần gọi đệ quy bên trong lùi về lần gọi đệ quy bên ngoài. Phương pháp này là một cách hiện thực trực tiếp của thủ tục duyệt theo chiều sâu một cách tổng quát đã được trình bày ở trên. Điểm khác biệt là việc xử lý tại mỗi đỉnh (ghi vào mảng thứ tự topo) chỉ được thực hiện sau khi các đỉnh kề của nó đã được xử lý.

```
template <int graph_size>
void Digraph<graph_size>::depth_sort(List<Vertex> &topological_order)
/*
post: Các đỉnh của một đồ thị có hướng không có chu trình được xếp theo thứ tự topo tương ứng
      cách duyệt đồ thị theo chiều sâu.
uses: Các phương thức của lớp List, hàm đệ quy recursive_depth_sort.
*/
{
   bool visited[graph_size];
   Vertex v;
   for (v = 0; v < count; v++) visited[v] = false;
   topological_order.clear();
   for (v = 0; v < count; v++)
      if (!visited[v]) // Thêm các đỉnh kề của v và sau đó là v vào mảng tứ tự topo từ
                                                                    phải sang
                                                                    trái.
         recursive_depth_sort(v, visited, topological_order);
}
```

Hàm phụ trợ **recursive_depth_sort** thực hiện việc đệ quy, dựa trên phác thảo của hàm **traverse** tổng quát, trước hết đặt tất cả các đỉnh sau của một đỉnh v vào các vị trí của chúng trong thứ tự topo, sau đó mới đặt v vào.

```
template <int graph_size>
void Digraph<graph_size>::recursive_depth_sort(Vertex v, bool *visited,
                          List<Vertex> &topological_order)
/*
pre:  Đỉnh v chưa có trong mảng thứ tự topo.
post: Thêm các đỉnh kề của v và sau đó là v vào mảng tứ tự topo từ phải sang trái.
uses: Các phương thức của lớp List và hàm đệ quy recursive_depth_sort.
*/
{  visited[v] = true;
   int degree = neighbors[v].size();
   for (int i = 0; i < degree; i++) {
      Vertex w;
      neighbors[v].retrieve(i, w);  // Một đỉnh kề của v.
      if (!visited[w])              // Duyệt tiếp xuống đỉnh w.
         recursive_depth_sort(w, visited, topological_order);
   }
   topological_order.insert(0, v);    // Đặt v vào mảng thứ tự topo.
}
```

Do giải thuật này duyệt qua mỗi đỉnh của đồ thị chính xác một lần và xem xét mỗi cạnh cũng một lần, đồng thời nó không hề thực hiện việc tìm kiếm nào, nên thời gian chạy là O(n+e), với n là số đỉnh và e là số cạnh của đồ thị.

13.4.3. Giải thuật duyệt theo chiều rộng

Trong thứ tự topo theo chiều rộng của một đồ thị có hướng không có chu trình, chúng ta bắt đầu bằng cách tìm các đỉnh có thể là các đỉnh đầu tiên trong thứ tự topo và sau đó chúng ta áp dụng nguyên tắc rằng, mọi đỉnh cần phải xuất hiện trước tất cả các đỉnh sau của nó trong thứ tự topo. Giải thuật này sẽ lần lượt đặt các đỉnh của đồ thị vào mảng thứ tự topo từ trái sang phải.

Các đỉnh có thể là đỉnh đầu tiên chính là các đỉnh không là đỉnh sau của bất kỳ đỉnh nào trong đồ thị. Để tìm được chúng, chúng ta tạo một mảng **predecessor_count**, mỗi phần tử tại chỉ số v chứa số đỉnh đứng ngay trước đỉnh v. Các đỉnh cần tìm chính là các đỉnh mà không có đỉnh nào đứng ngay trước nó, trị trong phần tử tương ứng của mảng bằng 0. Các đỉnh như vậy đã sẵn sàng được đặt vào mảng thứ tư topo. Như vậy chúng ta sẽ khởi động quá trình duyệt theo chiều rộng bằng cách đặt các đỉnh này vào một hàng các đỉnh sẽ được xử lý. Khi một đỉnh cần được xử lý, nó sẽ được lấy ra từ hàng và đặt vào vị trí kế tiếp trong mảng topo, lúc này, việc xem xét nó xem như kết thúc và được đánh dấu bằng cách cho các phần tử trong mảng **predecessor_count** tương ứng với các đỉnh là đỉnh sau của nó giảm đi 1. Khi một phần tử nào trong mảng này đạt được trị 0, thì cũng có nghĩa là đỉnh tương ứng với nó có các đỉnh trước đã được duyệt xong, và đỉnh này đã sẵn sàng được duyệt nên được đưa vào hàng đợi.

```
template <int graph_size>
void Digraph<graph_size>::breadth_sort(List<Vertex> &topological_order)
/*
post: Các đỉnh của một đồ thị có hướng không có chu trình được xếp theo thứ tự topo tương ứng
      cách duyệt đồ thị theo chiều rộng.
uses: Các phương thức của các lớp List, Queue.
*/
{  topological_order.clear();
   Vertex v, w;
   int predecessor_count[graph_size];
   for (v = 0; v < count; v++) predecessor_count[v] = 0;
   for (v = 0; v < count; v++)
      for (int i = 0; i < neighbors[v].size(); i++) { // Cập nhật số đỉnh đứng
                                                      trước cho mỗi đỉnh.
         neighbors[v].retrieve(i, w);
         predecessor_count[w]++;
      }
   Queue<Vertex> ready_to_process;
   for (v = 0; v < count; v++)
      if (predecessor_count[v] == 0)
         ready_to_process.append(v);

   while (!ready_to_process.empty()) {
      ready_to_process.retrieve(v);
      topological_order.insert(topological_order.size(), v);
      for (int j = 0; j < neighbors[v].size(); j++) { // Giảm số đỉnh đứng trước
         neighbors[v].retrieve(j, w);             // của mỗi đỉnh kề của v đi 1
         predecessor_count[w]--;
```

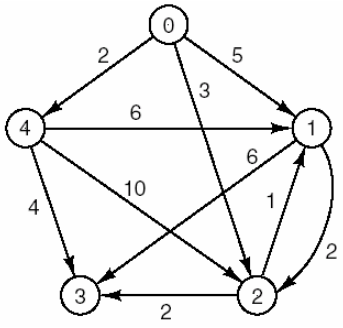
```
            if (predecessor_count[w] == 0)
               ready_to_process.append(w);
         }
         ready_to_process.serve();
      }
   }
```

Giải thuật này cần đến một trong các hiện thực của lớp `Queue`. `Queue` có thể có hiện thực theo bất kỳ cách nào đã được mô tả trong chương 3. Do các phần tử trong `Queue` là các đỉnh. Cũng như duyệt theo chiều sâu, thời gian cần cho hàm `breadth_first` là O(n+e), với n là số đỉnh và e là số cạnh của đồ thị.

13.5. Giải thuật Greedy: Tìm đường đi ngắn nhất

13.5.1. Đặt vấn đề

Như một ứng dụng khác của đồ thị, chúng ta xem xét một bài toán hơi phức tạp sau đây. Chúng ta có một đồ thị có hướng G, trong đó mỗi cạnh được gán một con số không âm gọi là tải trọng (*weight*). Bài toán của chúng ta là tìm một đường đi từ một đỉnh v đến một đỉnh w sao cho tổng tải trọng trên đường đi là nhỏ nhất. Chúng ta gọi đường đi như vậy là đường đi ngắn nhất (*shortest path*), mặc dù tải trọng có thể biểu diễn cho giá cả, thời gian, hoặc một vài đại lượng nào khác thay vì khoảng cách. Chúng ta có thể xem G như một bản đồ các tuyến bay, chẳng hạn, mỗi đỉnh của đồ thị biểu diễn một thành phố và tải trọng trên mỗi cạnh biểu diễn chi phí bay từ thành phố này sang thành phố kia. Bài toán của chúng ta là tìm lộ trình bay từ thành phố v đến thành phố w sao cho tổng chi phí là nhỏ nhất. Chúng ta hãy xem xét đồ thị ở hình 13.8. Đường ngắn nhất từ đỉnh 0 đến đỉnh 1 đi ngang qua đỉnh 2 có tổng tải trọng là 4, so với tải trọng là 5 đối với cạnh nối trực tiếp từ 0 sang 1, và tổng tải trọng là 8 nếu đi ngang qua đỉnh 4.



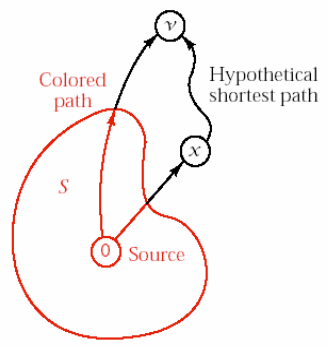
Hình 13.8 – Đồ thị có hướng với các tải trọng

Có thể dễ dàng giải bài toán một cách tổng quát như sau: bắt đầu từ một đỉnh, gọi là đỉnh nguồn, tìm đường đi ngắn nhất đến mọi đỉnh còn lại, thay vì chỉ tìm đường đến một đỉnh đích. Chúng ta cần tải trọng phải là những số không âm.

13.5.2. Phương pháp

Giải thuật sẽ được thực hiện bằng cách nắm giữ một tập S các đỉnh mà đường đi ngắn nhất từ đỉnh nguồn đến chúng đã được biết. Mới đầu, đỉnh nguồn là đỉnh duy nhất trong S. Tại mỗi bước, chúng ta thêm vào S các đỉnh còn lại mà đường đi ngắn nhất từ nguồn đến chúng vừa được tìm thấy. Bài toán bây giờ trở thành bài toán xác định đỉnh nào để thêm vào S tại mỗi bước. Chúng ta hãy xem những đỉnh đã có trong S như đã được tô một màu nào đó, và các cạnh nằm trong các đường đi ngắn nhất từ đỉnh nguồn đến các đỉnh có màu cũng được tô màu.

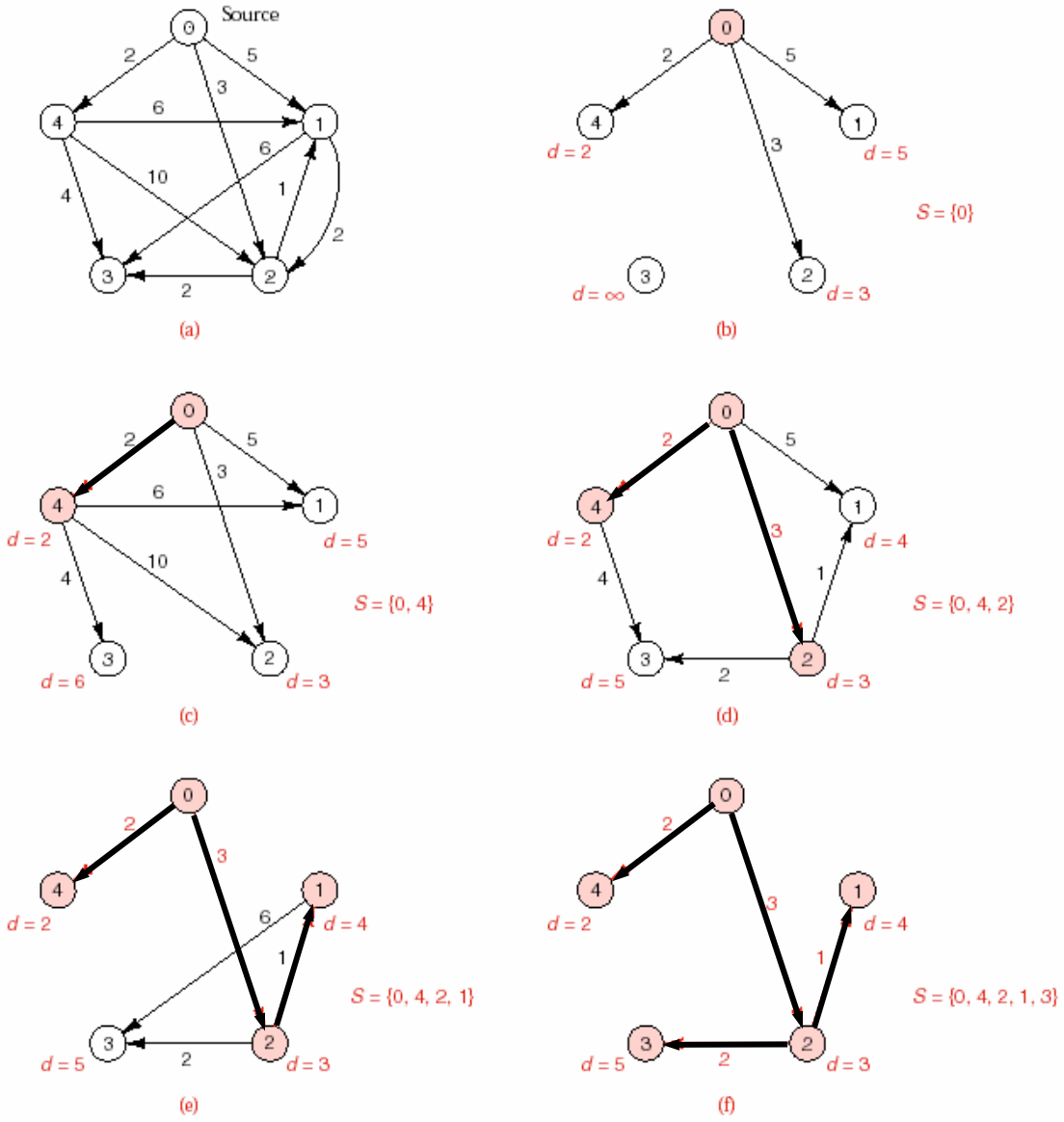
Chúng ta sẽ nắm giữ một mảng `distance` cho biết rằng, đối với mỗi đỉnh v, khoảng cách từ đỉnh nguồn dọc theo đường đi có các cạnh đã có màu, có thể trừ cạnh cuối cùng. Nghĩa là, nếu v thuộc *S*, thì `distance[v]` chứa khoảng cách ngắn nhất đến v, và mọi cạnh nằm trên đường đi tương ứng đều có màu. Nếu v không thuộc S, thì `distance[v]` chứa chiều dài của đường đi từ đỉnh nguồn đến một đỉnh w nào đó cộng với tải trọng của cạnh nối từ w đến v, và mọi cạnh nằm trên đường đi này, trừ cạnh cuối, đều có màu. Mảng `distance` được khởi tạo bằng cách gán từng `distance[v]` với trị của tải trọng của cạnh nối từ đỉnh nguồn đến v nếu tồn tại cạnh này, ngược lại nó được gán bằng vô cực.



Hình 13.9 – Tìm một đường đi ngắn

Để xác định đỉnh được thêm vào S tại mỗi bước, chúng ta áp dụng một "tiêu chí tham lam" (*greedy criterion*) trong việc chọn ra một đỉnh v có khoảng cách nhỏ nhất từ trong mảng `distance` sao cho v chưa có trong *S*. Chúng ta cần chứng minh rằng, đối với đỉnh v này, khoảng cách chứa trong mảng `distance` thực sự là chiều dài của đường đi ngắn nhất từ đỉnh nguồn đến v. Giả sử có một đường đi ngắn hơn từ nguồn đến v, như hình 13.9. Đường đi này đi ngang một đỉnh *x* nào đó chưa thuộc S rồi mới đến v (có thể lại qua một số đỉnh khác có nằm trong *S* trước khi gặp v). Nhưng nếu đường đi này ngắn hơn đường đi đã được tô màu đến v, thì đoạn đường ban đầu trong nó từ nguồn đến *x* còn ngắn hơn nữa, nghĩa là `distance[x] < distance[v]`, và như vậy tiêu chí tham lam đã phải chọn *x* thay vì v là đỉnh kế tiếp được thêm vào S.

Khi thêm v và S, chúng ta sẽ tô màu v và tô màu luôn đường đi ngắn nhất từ đỉnh nguồn đến v (mọi cạnh trong nó trừ cạnh cuối thực sự đã được tô màu trước



Hình 12.10 - Ví dụ về các đường đi ngắn nhất

đó). Kế tiếp, chúng ta cần cập nhật lại các phần tử của mảng `distance` bằng cách xem xét đối với mỗi đỉnh w nằm ngoài *S*, đường đi từ nguồn qua v rồi đến w có ngắn hơn khoảng cách từ nguồn đến w đã được ghi nhận trước đó hay không. Nếu điều này xảy ra có nghĩa là chúng ta vừa phát hiện được một đường đi mới cho đỉnh w có đi ngang qua v ngắn hơn cách đi đã xác định trước đó, và như vậy chúng ta cần cập nhật lại `distance[w]` bằng `distance[v]` cộng với tải trọng của cạnh nối từ v đến *w*.

13.5.3. Ví dụ

Trước khi viết một hàm cho phương pháp này, chúng ta hãy xem qua ví dụ ở hình 13.10. Đối với đồ thị có hướng ở hình (a), trạng thái ban đầu được chỉ ra ở hình (b): tập S (các đỉnh đã được tô màu) chỉ gồm có nguồn là đỉnh 0, các phần tử của mảng `distance` chứa các con số được đặt cạnh mỗi đỉnh còn lại. Khoảng cách đến đỉnh 4 ngắn nhất, nên 4 được thêm vào S như ở hình (c), và `distance[3]` được cập nhật thành 6. Do khoảng cách đến 1 và 2 ngang qua 4 lớn hơn khoảng cách đã chứa trong mảng `distance`, nên các khoảng cách này trong `distance` được giữ không đổi. Đỉnh kế tiếp gần nhất đối với nguồn là đỉnh 2, nó được thêm vào S như hình (d), khoảng cách đến các đỉnh 1 và 3 được cập nhật lại ngang qua đỉnh này. Hai bước cuối cùng, trong hình (e) và (f), đỉnh 1 và 3 được thêm vào và các đường đi ngắn nhất từ đỉnh nguồn đến các đỉnh còn lại được chỉ ra trong sơ đồ cuối.

13.5.4. Hiện thực

Để hiện thực giải thuật tìm đường ngắn nhất trên, chúng ta cần chọn cách hiện thực cho đồ thị có hướng. Việc dùng bảng kề cho phép truy xuất ngẫu nhiên đến mọi đỉnh của đồ thị. Hơn nữa, chúng ta có thể sử dụng bảng vừa để chứa các tải trọng vừa để chứa thông tin về các đỉnh kề. Trong đặc tả dưới đây của đồ thị có hướng, chúng ta thêm thông số `template` cho phép người sử dụng chọn lựa kiểu của tải trọng theo ý muốn. Lấy ví dụ, người sử dụng khi dùng lớp `Digraph` để mô hình hóa mạng các tuyến bay, họ có thể chứa giá vé là một số nguyên hay một số thực.

```
template <class Weight, int graph_size>
class Digraph {
public:
    //  Thêm constructor và các phương thức nhập và xuất đồ thị.
   void set_distances(Vertex source, Weight distance[]) const;
protected:
   int count;
   Weight adjacency[graph_size][graph_size];
};
```

Thuộc tính **`count`** chứa số đỉnh của một đối tượng đồ thị. Trong các ứng dụng, chúng ta cần bổ sung các phương thức để nhập hay xuất các thông tin của một đối tượng đồ thị, nhưng do chúng không cần thiết trong việc hiện thực phương thức **`distance`** để tìm các đường đi ngắn nhất, chúng ta xem chúng như là bài tập.

Chúng ta sẽ giả sử rằng lớp `Weight` đã có các tác vụ so sánh. Ngoài ra, người sử dụng sẽ phải khai báo trị lớn nhất có thể có của `Weight`, gọi là **`infinity`**. Chẳng hạn, chương trình của người sử dụng với tải trọng là số nguyên có thể sử dụng thư viện chuẩn ANSI C++`<limits>` với định nghĩa toàn cục như sau:

```
const Weight infinity = numeric_limits<int>::max();
```

Chúng ta sẽ đặt trị của infinity vào các phần tử của mảng distance tương ứng với các cạnh từ không tồn tại nguồn đến mỗi đỉnh. Phương thức **set_distance** sẽ tìm các đường đi ngắn nhất và các khoảng cách ngắn nhất này sẽ được trả về qua tham biến distance[].

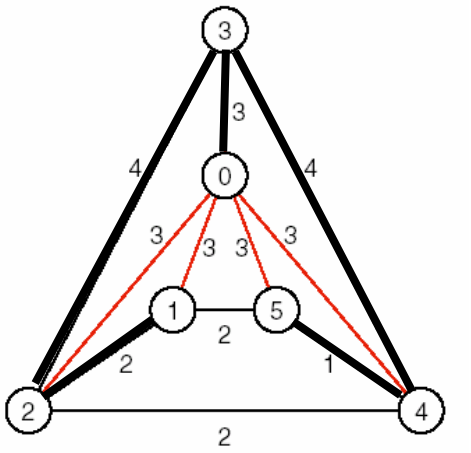
```
template <class Weight, int graph_size>
void Digraph<Weight, graph_size>::set_distances(Vertex source,
                                    Weight distance[]) const
/*
post: Mảng distance chứa đường đi có tải trọng ngắn nhất từ đỉnh nguồn đến mỗi đỉnh trong
      đồ thị.
*/
{  Vertex v, w;
   bool found[graph_size];   //  Biểu diễn các đỉnh trong S.
   for (v = 0; v < count; v++) {
      found[v] = false;
      distance[v] = adjacency[source][v];
   }
   found[source]=true;// Khởi tạo bằng cách bỏ đỉnh nguồn vào S.
   distance[source] = 0;
   for(int i = 0; i < count; i++){ // Mỗi lần lặp bỏ thêm một đỉnh vào S.
      Weight min = infinity;
      for (w = 0; w < count; w++) if (!found[w])
         if (distance[w] < min) {
            v = w;
            min = distance[w];
         }
      found[v] = true;
      for (w = 0; w < count; w++) if (!found[w])
         if (min + adjacency[v][w] < distance[w])
             distance[w] = min + adjacency[v][w];
   }
}
```

Để ước đoán thời gian cần để chạy hàm này, chúng ta thấy rằng vòng lặp chính thực hiện n-1 lần, trong đó n là số đỉnh, và bên trong vòng lặp chính có hai vòng lặp khác, mỗi vòng lặp này thực hiện n-1 lần. Vậy các vòng lặp thực hiện $(n-1)^2$ lần. Các lệnh bên ngoài vòng lặp chỉ hết O(n), nên thời gian chạy của hàm là $O(n^2)$.

13.6. Cây phủ tối tiểu

13.6.1. Đặt vấn đề

Giải thuật tìm đường đi ngắn nhất của phần trên có thể được áp dụng với mạng hay đồ thị có hướng cũng như mạng hay đồ thị không có hướng. Ví dụ, hình 13.11 minh họa ứng dụng tìm các đường đi ngắn nhất từ đỉnh nguồn 0 đến các đỉnh khác trong mạng.

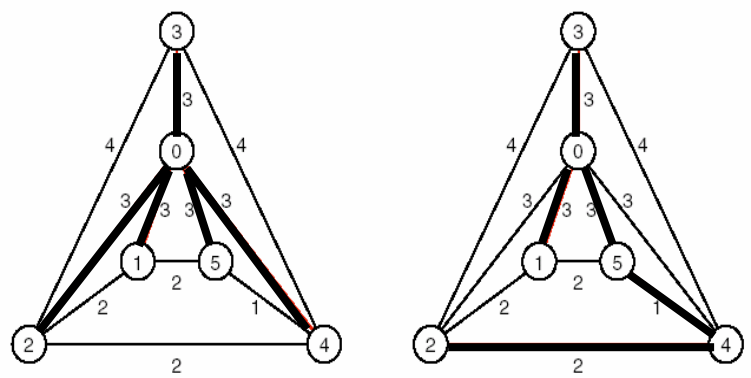


Hình 13.11 – Tìm đường đi ngắn nhất trong một mạng

Nếu mạng dựa trên cơ sở là một đồ thị liên thông G, thì các đường đi ngắn nhất từ một đỉnh nguồn nào đó sẽ nối nguồn này với tất cả các đỉnh khác trong G. Từ đó, như trong hình 13.11, nếu chúng ta kết hợp các đường đi ngắn nhất tính được lại với nhau, chúng ta có một cây nối tất cả các đỉnh của G. Nói cách khác, đó là cây được tạo bởi tất cả các đỉnh và một số cạnh của đồ thị G. Chúng ta gọi những cây như vậy là cây phủ (*spanning tree*) của G. Cũng như phần trước, chúng ta có thể xem một mạng trên một đồ thị G như là một bản đồ các tuyến bay, với mỗi đỉnh biểu diễn một thành phố và tải trọng trên một cạnh là giá vé bay từ thành phố này sang thành phố kia. Một cây phủ của G biểu diễn một tập các đường bay cho phép các hành khách hoàn tất một chuyến du lịch qua khắp các thành phố. Dĩ nhiên rằng, hành khách cần phải thực hiện một số tuyến bay nào đó một vài lần mới hoàn tất được chuyến du lịch. Điều bất tiện này được bù đắp bởi chi phí thấp. Nếu chúng ta hình dung mạng trong hình 13.11 như một hệ thống điều khiển tập trung, thì đỉnh nguồn tương ứng với sân bay trung tâm, và các đường đi từ đỉnh này là những hành trình bay. Một điều quan trọng đối với một sân bay theo hệ thống điều khiển tập trung là giảm tối đa chi phí bằng cách chọn lựa một hệ thống các đường bay mà tổng chi phí nhỏ nhất.

Định nghĩa: Một cây phủ tối tiểu (*minimal spanning tree*) của một mạng liên thông là cây phủ mà tổng các tải trọng trên các cạnh của nó là nhỏ nhất.

Mặc dù việc so sánh hai cây phủ trong hình 13.12 là không khó khăn, nhưng cũng khó mà biết được còn có cây phủ nào có trị nhỏ hơn nữa hay không. Bài toán của chúng ta là xây dựng một phương pháp xác định một cây phủ tối tiểu của một mạng liên thông.



Hình 13.12 – Hai cây phủ trong một mạng

13.6.2. Phương pháp

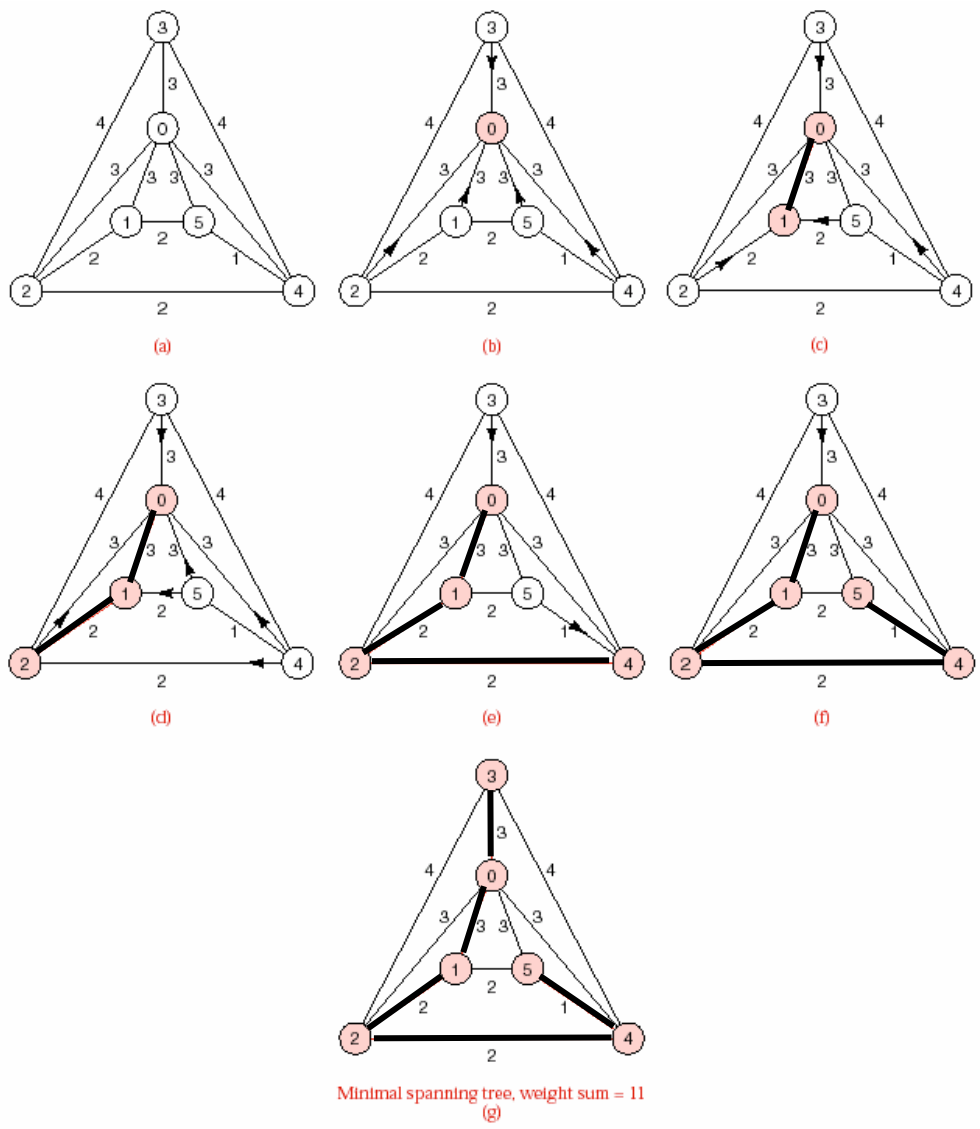
Chúng ta đã biết giải thuật tìm cây phủ trong một đồ thị liên thông, do giải thuật tìm đường ngắn nhất đã có. Chúng ta có thể thay đổi chút ít trong giải thuật tìm đường ngắn nhất để có được phương pháp tìm cây phủ tối tiểu mà R. C. Prim đã đưa ra lần đầu vào năm 1957.

Trước hết chúng ta chọn ra một đỉnh bắt đầu, gọi là nguồn, và trong khi tiến hành phương pháp, chúng ta nắm giữ một tập X các đỉnh mà đường đi từ chúng đến đỉnh nguồn thuộc về cây phủ tối tiểu mà chúng ta đã tìm thấy. Chúng ta cũng cần nắm một tập Y gồm các cạnh nối các đỉnh trong X mà thuộc cây đang được xây dựng. Như vậy, chúng ta có thể hình dung rằng các đỉnh trong *X* và các cạnh trong Y đã tạo ra một phần của cây mà chúng ta cần tìm, cây này sẽ lớn lên thành cây phủ tối tiểu cuối cùng. Lúc khởi đầu, đỉnh nguồn là đỉnh duy nhất trong X, và Y là tập rỗng. Tại mỗi bước, chúng ta thêm một đỉnh vào X: đỉnh này được chọn sao cho nó có một cạnh nối với một đỉnh nào đó đã có trong X có tải trọng nhỏ nhất, so với các tải trọng của tất cả các cạnh khác nối các đỉnh còn nằm ngoài X với các đỉnh đã có trong X. Cạnh tối tiểu này sẽ được đưa vào Y.

Việc chứng minh giải thuật Prim đem lại cây phủ tối tiểu không thực dễ dàng, chúng ta tạm hoãn việc này lại sau. Tuy nhiên, chúng ta có thể hiểu về việc chọn lựa một đỉnh mới để thêm vào X và một cạnh mới để thêm vào Y . Tiêu chí Prim cho chúng ta một cách dễ dàng nhất để thực hiện việc kết nối này, và như vậy, theo tiêu chí tham lam, chúng ta nên sử dụng nó.

Khi hiện thực giải thuật Prim, chúng ta cần nắm giữ một danh sách các đỉnh thuộc X như là các phần tử của một mảng kiểu `bool`. Cách nắm giữ các cạnh trong Y sẽ dễ dàng nếu như chúng ta bắt chước cách lưu trữ các cạnh trong một đồ thị.

Chúng ta sẽ cần đến một mảng phụ **`neighbor`** để biết thêm rằng, đối với mỗi đỉnh v, đỉnh nào trong X có cạnh đến v có tải trọng nhỏ nhất. Các tải trọng nhỏ nhất này cũng chứa trong mảng **`distance`** tương ứng. Nếu một đỉnh v không có một cạnh nào nối được với một đỉnh nào đó trong X, trị của phần tử tương ứng của nó trong `distance` sẽ là **`infinity`**.



Hình 13.13 – Ví dụ về giải thuật Prim

Mảng `neighbor` được khởi tạo bằng cách gán `neighbor[v]` đến đỉnh nguồn cho tất cả các đỉnh v, và `distance` được khởi tạo bằng cách gán `distance[v]` bởi tải trọng của cạnh nối từ đỉnh nguồn đến v hoặc `infinity` nếu cạnh này không tồn tại.

Để xác định đỉnh nào sẽ được thêm vào tập X tại mỗi bước, chúng ta chọn đỉnh v trong số các đỉnh chưa có trong X mà trị tương ứng trong mảng `distance` là nhỏ nhất. Sau đó chúng ta cần cập nhật lại các mảng để phản ánh đúng sự thay đổi mà chúng ta đã làm đối với tập X như sau: với mỗi w chưa có trong X, nếu có

một cạnh nối v và w, chúng ta xem thử tải trọng của cạnh này có nhỏ hơn `distance[w]` hay không, nếu quả thực như vậy thì `distance[w]` cần được cập nhật lại bằng trị của tải trọng này, và `neighbor[w]` sẽ là v.

Lấy ví dụ, chúng ta hãy xem xét mạng trong hình (a) của hình 13.13. Trạng thái ban đầu trong hình (b): Tập X (các đỉnh được tô màu) chỉ gồm đỉnh nguồn 0, và đối với mỗi đỉnh w, đỉnh được lưu trong `neighbor[w]` được chỉ bởi các mũi tên từ w. Trị của `distance[w]` là tải trọng của cạnh tương ứng. Khoảng cách từ nguồn đến đỉnh 1 là một trong những trị nhỏ nhất, nên 1 được thêm vào X như hình (c), và trong các phần tử của các mảng `neighbor` và `distance` chỉ có các phần tử tương ứng với đỉnh 2 và đỉnh 5 là được cập nhật lại. Đỉnh kế tiếp gần với các đỉnh trong X nhất là đỉnh 2, nó được thêm vào X như hình (d), trong này cũng đã chỉ ra các trị của các mảng đã được cập nhật lại. Các bước cuối cùng được minh họa trong hình (e), (f) và (g).

13.6.3. Hiện thực

Để hiện thực giải thuật Prim, chúng ta cần chọn một lớp để biểu diễn cho mạng. Sự tương tự của giải thuật này so với giải thuật tìm đường ngắn nhất trong phần trước giúp chúng ta quyết định thiết kế lớp `Network` dẫn xuất từ lớp `Digraph`.

```
template <class Weight, int graph_size>
class Network: public Digraph<Weight, graph_size> {
public:
   Network();
   void read();   //  Định nghĩa lại để nhập thông tin về mạng.
   void make_empty(int size = 0);
   void add_edge(Vertex v, Vertex w, Weight x);
   void minimal_spanning(Vertex source,
                         Network<Weight, graph_size> &tree) const;
};
```

Chúng ta sẽ viết lại phương thức nhập **`read`** để đảm bảo rằng tải trọng của cạnh (v, w) luôn trùng với tải trọng của cạnh (w, v), với mọi v và w, vì đây là một mạng vô hướng. Phương thức mới **`make_empty(int size)`** tạo một mạng có **`size`** đỉnh và không có cạnh. Phương thức khác, **`add_edge`**, thêm một cạnh có một tải trọng cho trước vào mạng. Chúng ta cũng đã giả sử rằng lớp `Weight` đã có đầy đủ các toán tử so sánh. Ngoài ra, người sử dụng cần khai báo trị lớn nhất có thể của `Weight` gọi là **`infinity`**. Phương thức **`minimal_spanning`** mà chúng ta sẽ viết ở đây sẽ tìm cây phủ tối tiểu và trả về qua tham biến **`tree`**. Tuy phương thức chỉ có thể tìm cây phủ khi thực hiện trên một mạng liên thông, nó cũng có thể tìm một cây phủ cho một thành phần liên thông có chứa đỉnh **`source`** trong mạng.

```
template <class Weight, int graph_size>
void Network<Weight, graph_size>::minimal_spanning(Vertex source,
                     Network<Weight, graph_size> &tree) const
/*
post: Xác định cây phủ tối tiểu trong thành phần liên thông có chứa đỉnh source của mạng.
*/
{
   tree.make_empty(count);
   bool component[graph_size]; // Các đỉnh trong tập X.
   Vertex neighbor[graph_size];// Phần tử thứ i chứa đỉnh trước của nó sao cho khoảng
                               // cách giữa chúng nhỏ nhất so với các khoảng cách từ
                               // các đỉnh trước khác đã có trong tập X đến nó.
   Weight distance[graph_size];// Các khoảng cách nhỏ nhất tương ứng với từng phần tử
                               // trong mảng neighbor trên.
   Vertex w;

   for (w = 0; w < count; w++) {
      component[w] = false;
      distance[w] = adjacency[source][w];
      neighbor[w] = source;
   }
   component[source] = true;    //  Tập X chỉ có duy nhất đỉnh nguồn.
   for (int i = 1; i < count; i++) {
      Vertex v;                 //  Mỗi lần lặp bổ sung thêm một đỉnh vào tập X.
      Weight min = infinity;
      for (w = 0; w < count; w++) if (!component[w])
         if (distance[w] < min) {
            v = w;
            min = distance[w];
         }

      if (min < infinity) {
         component[v] = true;
         tree.add_edge(v, neighbor[v], distance[v]);
         for (w = 0; w < count; w++) if (!component[w])
            if (adjacency[v][w] < distance[w]) {
               distance[w] = adjacency[v][w];
               neighbor[w] = v;
            }
      }
      else break;      //  Xong một thành phần liên thông trong đồ thị không liên thông.
   }
}
```

Vòng lặp chính trong hàm trên thực hiện n-1 lần, với n là số đỉnh, và trong vòng lặp chính còn có hai vòng lặp khác, mỗi vòng lặp này thực hiện n-1 lần. Vậy các vòng lặp thực hiện $(n-1)^2$ lần. Các lệnh bên ngoài vòng lặp chỉ hết O(n), nên thời gian chạy của hàm là $O(n^2)$.

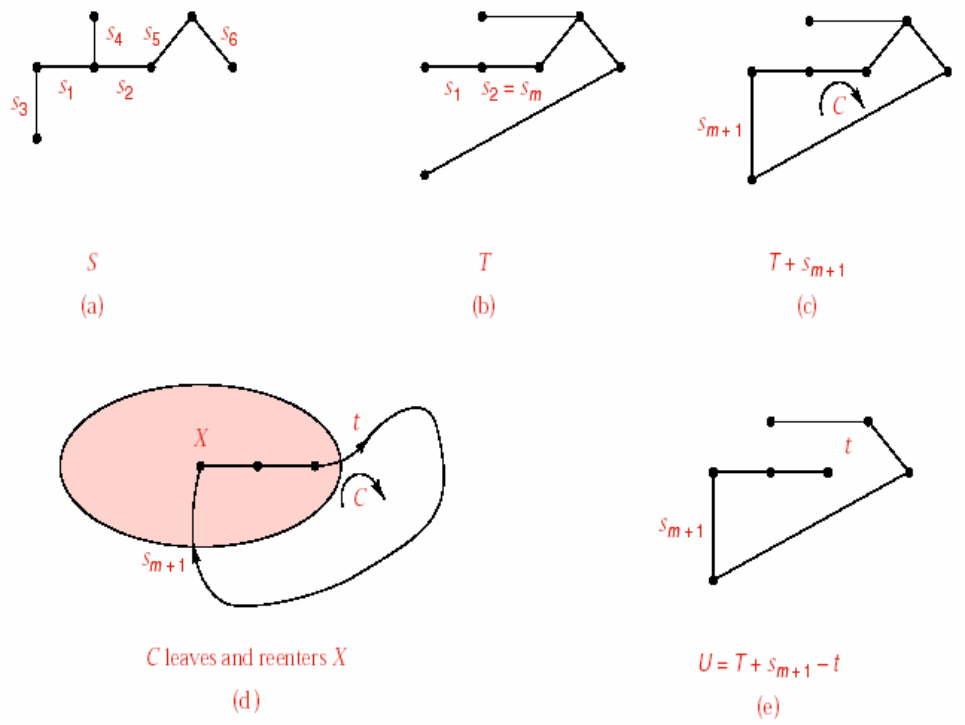
13.6.4. Kiểm tra giải thuật Prim

Chúng ta cần chứng minh rằng, đối với đồ thị liên thông G, cây phủ S sinh ra bởi giải thuật Prim phải có tổng tải trọng trên các cạnh nhỏ hơn so với bất kỳ

một cây phủ nào khác của G. Giải thuật Prim xác định một chuỗi các cạnh s_1, s_2, ..., s_n tạo ra cây phủ S. Như hình 13.14, s_1 là cạnh thứ nhất được thêm vào tập Y trong giải thuật Prim, s_2 là cạnh thứ hai được thêm vào, và cứ thế.

Để chứng minh S là một cây phủ tối tiểu, chúng ta chứng minh rằng nếu m là một số nguyên, $0 \le m \le n$, thì sẽ có một cây phủ tối tiểu chứa cạnh s_i với $i \le m$. Chúng ta sẽ chứng minh quy nạp trên m. Trường hợp cơ bản, khi m = 0, rõ ràng là đúng, vì bất kỳ cây phủ tối tiểu nào cũng đều phải chứa một tập rỗng các cạnh. Ngoài ra, khi chúng ta hoàn tất việc quy nạp, trường hợp cuối cùng với m = n chỉ ra rằng có một cây phủ tối tiểu chứa tất cả các cạnh của S, và chính là S. (Lưu ý rằng nếu thêm bất kỳ một cạnh nào vào một cây phủ thì cũng tạo ra một chu trình, nên bất kỳ cây phủ nào chứa mọi cạnh của S đều phải chính là S). Nói cách khác, khi chúng ta hoàn tất việc quy nạp, chúng ta đã chứng minh được rằng S chính là cây phủ tối tiểu.

Như vậy chúng ta cần trình bày các bước quy nạp bằng cách chứng minh rằng nếu $m < n$ và T là một cây phủ tối tiểu chứa các cạnh s_i với $i \le m$, thì phải có một cây phủ tối tiểu U cũng chứa các cạnh trên và thêm một cạnh s_{m+1}. Nếu s_{m+1} đã có trong T, chúng ta có thể đơn giản cho U = T, như vậy chúng ta có thể giả sử rằng s_{m+1} không là một cạnh trong T. Chúng ta hãy xem hình (b) của hình 13.14.



Hình 13.14 – Kiểm tra giải thuật Prim

Chúng ta hãy gọi X là tập các đỉnh trong S thuộc các cạnh s_1, s_2, ..., s_m và R là tập các đỉnh còn lại trong S. Trong giải thuật Prim, cạnh s_{m+1} nối một đỉnh trong

X với một đỉnh trong R, và s_{m+1} là một trong các cạnh có tải trọng nhỏ nhất nối giữa hai tập này. Chúng ta hãy xét ảnh hưởng của việc thêm cạnh s_{m+1} này vào T, như minh họa trong hình (c). Việc thêm vào này phải tạo ra một chu trình C, do mạng liên thông T rõ ràng là có chứa một đường đi có nhiều cạnh nối hai đầu của cạnh s_{m+1}. Chu trình C phải có chứa một cạnh $t \neq s_{m+1}$ nối tập X với tập R, do nếu chúng ta di dọc theo đường đi khép kín C chúng ta phải đi vào tập X một số lần bằng với số lần chúng ta đi ra khỏi nó. Chúng ta hãy xem hình (d). Giải thuật Prim bảo đảm rằng tải trọng của s_{m+1} nhỏ hơn hoặc bằng tải trọng của t. Do đó, cây phủ mới U trong hình (e), có được từ T bằng cách loại đi t và thêm vào s_{m+1}, sẽ có tổng tải trọng không lớn hơn tải trọng của T. Như vậy chúng ta đã có được U là một cây phủ tối tiểu của G, mà U chứa các cạnh s_1, s_2, ..., s_m, s_{m+1}. Điều này hoàn tất được quá trình quy nạp của chúng ta.

13.7. Sử dụng đồ thị như là cấu trúc dữ liệu

Trong chương này, chúng ta đã tìm hiểu chỉ một ít ứng dụng của đồ thị, nhưng chúng ta đã bắt đầu thâm nhập vào những vấn đề sâu sắc của các giải thuật về đồ thị. Nhiều giải thuật trong số đó, đồ thị xuất hiện như các cấu trúc toán học và đã nắm bắt được các đặc trưng thiết yếu của bài toán, thay vì chỉ là những công cụ tính toán cho ra được những lời giải của chúng. Lưu ý rằng trong chương này chúng ta đã nói về các đồ thị như là các cấu trúc toán học, chứ không như các cấu trúc dữ liệu, do chúng ta đã sử dụng chúng để đặc tả các vấn đề trong toán học, và để viết các giải thuật, chúng ta đã hiện thực các đồ thị trong các cấu trúc dữ liệu như danh sách hoặc bảng. Tuy vậy, rõ ràng là đồ thị tự bản thân nó có thể được xem như các cấu trúc dữ liệu - các cấu trúc dữ liệu mà có chứa các mối quan hệ giữa các dữ liệu phức tạp hơn những gì đã được mô tả trong một danh sách hoặc một cây. Do tính tổng quát và mềm dẻo, đồ thị là cấu trúc dữ liệu rất hiệu quả và đã tỏ rõ những giá trị của nó trong những ứng dụng cấp tiến như thiết kế hệ quản trị cơ sở dữ liệu chẳng hạn. Tất nhiên, một công cụ mạnh như vậy càng nên được sử dụng mọi khi cần thiết, nhưng việc sử dụng nó cần phải được kết hợp với việc xem xét một cách cẩn thận để sức mạnh của nó không làm cho cho chúng ta bị rối. Cách an toàn nhất trong việc sử dụng một công cụ mạnh là dựa trên sự chính quy; nghĩa là, chúng ta chỉ sử dụng công cụ mạnh trong những phương pháp đã được định nghĩa một cách cẩn thận và dễ hiểu. Do tính tổng quát của đồ thị, việc tuân thủ nguyên tắc vừa nêu ra trong việc sử dụng nó không phải luôn dễ dàng.

Phần 3 – CÁC ỨNG DỤNG CỦA CÁC LỚP CTDL

Chương 14 – ỨNG DỤNG CỦA NGĂN XẾP

Dựa trên tính chất của các giải thuật, các ứng dụng của ngăn xếp có thể được chia làm bốn nhóm như sau: đảo ngược dữ liệu, phân tích biên dịch dữ liệu, trì hoãn công việc và các giải thuật quay lui. Một điều đáng chú ý ở đây là khi xem xét các ứng dụng, chúng ta không bao giờ quan tâm đến cấu trúc chi tiết của ngăn xếp. Chúng ta luôn sử dụng ngăn xếp như một cấu trúc dữ liệu trừu tượng với các chức năng mà chúng ta đã định nghĩa cho nó.

14.1. Đảo ngược dữ liệu

Trong phần trình bày về ngăn xếp chúng ta đã được làm quen với một ví dụ xuất các phần tử theo thứ tự ngược với thứ tự nhập vào. Ở đây chúng ta tiếp tục tham khảo thêm ứng dụng đổi một số thập phân sang một số nhị phân.

Ứng dụng đổi số thập phân sang số nhị phân

Giải thuật dưới đây chuyển đổi số thập phân `decNum` sang một số nhị phân.

```
1  loop (decNum > 0)
   1  digit = decNum % 2
   2  xuất (digit)
   3  decNum = decNum / 2
2  endloop
```

Tuy nhiên các ký số được xuất ra sẽ là thứ tự ngược của kết quả mà chúng ta mong muốn. Chẳng hạn số 19 lẽ ra phải được đổi thành 10011 chứ không phải là 11001. Thực là dễ dàng nếu chúng ta sử dụng ngăn xếp để khắc phục điều này.

```
void DecimalToBinary (val int decNum)
post: số nhị phân tương đương với số thập phân decNum sẽ được xuất ra.
uses: sử dụng lớp Stack để đảo ngược thứ tự các số 1 và số 0.
{
   1. Stack<int> reverse; // Khởi tạo ngăn xếp để chứa các ký số 0 và 1.
   2. loop (decNum > 0)
      1. digit = decNum % 2
      2. reverse.push(digit)
      3. decNum = decNum / 2
   3. endloop
   4. loop (!reverse.empty())
      1. reverse.top(digit)
      2. reverse.pop()
      3. xuất(digit)
   5  endloop
}
```

Một điều dễ nhận thấy là nếu chúng ta dùng một mảng liên tục (`array` trong C++) để chứa các số `digit` rồi tìm cách in theo thứ tự đảo lại, chúng ta sẽ phải tiêu tốn sức lực vào việc quản lý các biến chỉ số chạy trên mảng. Đó là điều nên tránh. Việc tuân thủ lời khuyên này giúp chúng ta có thói quen tốt khi đụng phải những bài toán lớn hơn: chúng ta có thể tập trung vào giải quyết những vấn đề chính của bài toán.

14.2. Phân tích biên dịch (parsing) dữ liệu

Việc phân tích dữ liệu thường bao gồm phân tích từ vựng và phân tích cú pháp. Chẳng hạn, để chuyển đổi một chương trình nguồn được viết bởi một ngôn ngữ nào đó thành ngôn ngữ máy, trình biên dịch cần tách chương trình ấy ra thành các từ khóa, các danh hiệu, các ký hiệu, sau đó tiến hành kiểm tra tính hợp lệ về từ vựng, về cú pháp. Trong việc kiểm tra cú pháp thì việc kiểm tra cấu trúc khối lồng nhau một cách hợp lệ là một trong những điều có thể được thực hiện dễ dàng nhờ ngăn xếp.

Ứng dụng kiểm tra tính hợp lệ của các cấu trúc khối lồng nhau

Để kiểm tra tính hợp lệ của các cấu trúc khối lồng nhau, chúng ta cần kiểm tra các cặp dấu ngoặc như [], {}, () phải tuân theo một thứ tự đóng mở hợp lệ, có nghĩa là mỗi khối cần phải nằm gọn trong một khối khác, nếu có.

Lý do sử dụng ngăn xếp được giải thích như sau: theo thứ tự xuất hiện, một dấu ngoặc mở xuất hiện sau cần phải có dấu ngoặc đóng tương ứng xuất hiện trước. Ví dụ [...(...)...] là hợp lệ, [...(...]...) là không hợp lệ. Điều này rõ ràng liên quan đến nguyên tắc FILO của ngăn xếp. Mỗi cấu trúc khối sẽ được chúng ta biết đến

khi bắt đầu gặp dấu ngoặc mở của nó, và chúng ta sẽ chờ cho đến khi nào gặp dấu ngoặc đóng tương ứng của nó thì xem như chúng ta đã duyệt qua cấu trúc đó. Các dấu ngoặc mở mà chúng ta gặp, chúng ta sẽ lần lượt lưu vào ngăn xếp, nếu đoạn chương trình hợp lệ, thì chúng ta cứ yên tâm rằng các dấu ngoặc đóng tương ứng của chúng sẽ xuất hiện theo đúng thứ tự ngược lại. Như vậy, mỗi khi gặp một dấu ngoặc đóng, việc cần làm là lấy từ ngăn xếp ra một dấu ngoặc mở và so trùng.

Văn bản cần kiểm tra thường là một biểu thức tính toán hay một đoạn chương trình.

Giải thuật: Đọc đoạn văn bản từng ký tự một. Mỗi dấu ngoặc mở (, [, { được xem như một dấu ngoặc chưa so trùng và được lưu vào ngăn xếp cho đến khi gặp một dấu ngoặc đóng),], } so trùng tương ứng. Mỗi dấu ngoặc đóng cần phải so trùng được với dấu ngoặc mở vừa được lưu cuối cùng, và như vậy dấu ngoặc mở này sẽ được lấy ra khỏi ngăn xếp và bỏ đi. Như vậy việc kiểm tra sẽ được lặp cho đến khi gặp một dấu ngoặc đóng mà không so trùng được với dấu ngoặc mở vừa lưu trữ (lỗi các khối không lồng nhau) hoặc đến khi hết văn bản cần kiểm tra. Trường hợp dấu ngoặc đóng xuất hiện mà ngăn xếp rỗng là trường hợp văn bản bị lỗi thừa dấu ngoặc đóng (tính đến vị trí đang xét); ngược lại, khi đọc hết đoạn văn bản, nếu ngăn xếp không rỗng thì do lỗi thừa dấu ngoặc mở.

Chương trình có thể mở rộng hơn đối với nhiều cặp dấu ngoặc khác nhau, hoặc cho cả trường hợp đặc biệt về đoạn chú thích trong một chương trình C (/* ...phần trong này dĩ nhiên không cần kiểm tra tính hợp lệ của các cặp dấu ngoặc ...*/)

```
int main()
/*
post: Chương trình sẽ báo cho người sử dụng khi đoạn văn bản cần phân tích gặp lỗi.
uses: lớp Stack.
*/
{
   Stack<char> openings;
   char symbol;
   bool is_matched = true;
   while (is_matched && (symbol = cin.get()) != '\n') {
      if (symbol == '{' || symbol == '(' || symbol == '[')
         openings.push(symbol);
      if (symbol == '}' || symbol == ')' || symbol == ']') {
         if (openings.empty()) {
            cout << "Unmatched closing bracket " << symbol
                 << " detected." << endl;
            is_matched = false;
         }
         else {
            char match;
            openings.top(match);
            openings.pop();
            is_matched = (symbol == '}' && match == '{')
                         || (symbol == ')' && match == '(')
                         || (symbol == ']' && match == '[');
```

```
            if (!is_matched)
               cout << "Bad match " << match << symbol << endl;
         }
      }
   }
   if (!openings.empty())
      cout << "Unmatched opening bracket(s) detected." << endl;
}
```

14.3. Trì hoãn công việc

Khi sử dụng ngăn xếp để đảo ngược dữ liệu, toàn bộ dữ liệu cần được duyệt xong, chúng ta mới bắt đầu lấy dữ liệu từ ngăn xếp. Nhóm ứng dụng liên quan đến việc trì hoãn công việc thường chỉ cần trì hoãn việc xử lý dữ liệu trong một thời gian nhất định nào đó mà thôi.

Có nhiều giải thuật mà dữ liệu cần xử lý có thể xuất hiện bất cứ lúc nào, chúng sẽ được lưu giữ lại để chương trình lần lượt giải quyết. Trong trường hợp dữ liệu cần được xử lý theo đúng thứ tự mà chúng xuất hiện, chúng ta sẽ dùng hàng đợi làm nơi lưu trữ dữ liệu. Ngược lại, nếu thứ tự xử lý dữ liệu ngược với thứ tự mà chúng xuất hiện, chúng ta sẽ dùng ngăn xếp do nguyên tắc FILO của nó.

14.3.1. Ứng dụng tính trị của biểu thức *postfix*

Chúng ta sẽ xem xét ví dụ về cách tính trị của một biểu thức ở dạng Balan ngược (*reverse Polish calculator*- còn gọi là *postfix*). Trong biểu thức này toán tử luôn đứng sau toán hạng của nó. Trong quá trình duyệt biểu thức, khi gặp các toán hạng chúng ta phải hoãn việc tính toán cho đến khi gặp toán tử tương ứng của chúng, do đó chúng sẽ được đẩy vào ngăn xếp. Khi gặp toán tử, các toán hạng được lấy ra khỏi ngăn xếp, phép tính được thực hiện và kết quả lại được đẩy vào ngăn xếp (do kết quả này có thể lại là toán hạng của một phép tính khác mà toán tử của nó chưa xuất hiện). Thứ tự FILO được nhìn thấy ở chỗ: toán tử của những toán hạng xuất hiện trước luôn đứng sau toán tử của những toán hạng xuất hiện sau. Chẳng hạn, với 8 5 2 - + (tương đương 8 + (5-2)), số 8 xuất hiện trước số 2, nhưng phép trừ của (5 - 2) lại có trước phép cộng.

Việc phân tích một biểu thức liên quan đến việc xử lý chuỗi để tách ra các toán hạng cũng như các toán tử. Do phần tiếp theo đây chỉ chú trọng đến ý tưởng sử dụng ngăn xếp trong giải thuật, nên chương trình sẽ nhận biết các thành phần của biểu thức một cách dễ dàng thông qua việc cho phép người sử dụng lần lượt nhập chúng. Việc phân tích biểu thức có thể được xem như bài tập khi sinh viên kết hợp với các kiến thức khác có liên quan đến việc xử lý chuỗi ký tự.

Trong chương trình, người sử dụng nhập dấu ? để báo trước sẽ nhập một toán hạng, toán hạng này sẽ được chương trình lưu vào ngăn xếp. Khi các dấu +, -, *, / được nhập, chương trình sẽ lấy các toán hạng từ ngăn xếp, tính và đưa kết quả

vào ngăn xếp; dấu = yêu cầu hiển thị phần tử tại đỉnh ngăn xếp (nhưng không lấy ra khỏi ngăn xếp), đó là kết quả của một phép tính mới nhất vừa được thực hiện.

Giả sử a, b, c, d biểu diễn các giá trị số. Dòng nhập **? a ? b ? c - = * ? d + =** được thực hiện như sau:

? a: đẩy a vào ngăn xếp;
? b: đẩy b vào ngăn xếp
? c: đẩy c vào ngăn xếp
- : lấy c và b ra khỏi ngăn xếp, đẩy b-c vào ngăn xếp
= : in giá trị b-c
* : lấy 2 toán hạng từ ngăn xếp là trị (b-c) và a, tính a * (b-c), đưa kết quả vào ngăn xếp.
? d: đẩy d vào ngăn xếp.
+ : lấy 2 toán hạng từ ngăn xếp là d và trị (a * (b-c)), tính (a * (b-c)) + d, đưa kết quả vào ngăn xếp.
= : in kết quả (a * (b-c)) + d

Ưu điểm của cách tính Balan ngược là mọi biểu thức phức tạp đều có thể được biểu diễn không cần cặp dấu ngoặc ().

Cách biểu diễn Balan ngược rất tiện lợi trong các trình biên dịch cũng như các phép tính toán.

Hàm phụ trợ **get_command** nhận lệnh từ người sử dụng, kiểm tra hợp lệ và chuyển thành chữ thường bởi **tolower()** trong thư viện cctype.

```
int main()
/*
post: chương trình thực hiện tính toán trị của biểu thức số học dạng postfix do người sử dụng
      nhập vào.
uses: lớp Stack và các hàm introduction, instructions, do_command,  get_command.
*/

{
   Stack<double> stored_numbers;
   introduction(); // Giới thiệu về chương trình.
   instructions(); // Xuất các hướng dẫn sử dụng chương trình.
   while (do_command(get_command(), stored_numbers));
}
```

```
char get_command()
/*
post: trả về một  trong những ký tự hợp lệ do người sử dụng gõ vào (?, =, +, -, *, /, q).
*/
{
```

```
   char command;
   bool waiting = true;
   cout << "Select command and press <Enter>:";

   while (waiting) {
      cin >> command;
      command = tolower(command);
      if (command == '?' || command == '=' || command == '+' ||
          command == '-' || command == '*' || command == '/' ||
          command == 'q' ) waiting = false;
      else {
         cout << "Please enter a valid command:"   << endl
              << "[?]push to stack   [=]print top" << endl
              << "[+] [-] [*] [/]   are arithmetic operations" << endl
              << "[Q]uit." << endl;
      }
   }
   return command;
}
```

Ngăn xếp stored_numbers làm thông số cho hàm do_command được khai báo là tham chiếu do nó cần phải thay đổi khi hàm được gọi.

```
bool do_command(char command, Stack<double> &numbers)
/*
pre:  command chứa ký hiệu của phép tính số học (+, - *, /) hoặc các ký tự đã quy định (q, =, ?).
post: Việc xử lý tùy thuộc thông số  command.  Hàm trả về true, ngoại trừ trường hợp kết thúc
      viêc tính toán khi command là 'q'.
uses: lớp Stack.
*/
{
   double p, q;
   switch (command) {
   case '?':
      cout << "Enter a real number: " << flush;
      cin >> p;
      if (numbers.push(p) == overflow)
         cout << "Warning: Stack full, lost number" << endl;
      break;

   case '=':
      if (numbers.top(p) == underflow)
         cout << "Stack empty" << endl;
      else
         cout << p << endl;
      break;

   case '+':
      if (numbers.top(p) == underflow)
         cout << "Stack empty" << endl;
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }

         else {
```

```
            numbers.pop();
            if (numbers.push(q + p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;

   //   Add options for further user commands.

    case 'q':
      cout << "Calculation finished.\n";
      return false;
   }
   return true;
}
```

14.3.2. Ứng dụng chuyển đổi biểu thức dạng *infix* thành dạng *postfix*

Ngược với biểu thức dạng *postfix, b*iểu thức dạng *infix* cho phép có các dấu ngoặc đóng mở quy ước về độ ưu tiên của các phép tính. Chúng ta có độ ưu tiên từ cao xuống thấp theo thứ tự sau đây:

Độ ưu tiên 2 (cao nhất): các phép tính * và /
Độ ưu tiên 1 : các phép tính + và -
Độ ưu tiên 0 : dấu (,)

Khi duyệt biểu thức *infix* từ trái sang phải, các toán hạng trong biểu thức *infix* đều được đưa ngay vào biểu thức *postfix*, các toán tử cần được hoãn lại nên được đưa vào ngăn xếp. Trong biểu thức *postfix*, toán tử nào gặp trước sẽ được tính toán trước. Do đó, từ biểu thức *infix,* trước khi một toán tử nào đó cần được đưa vào ngăn xếp thì phải lấy từ đỉnh ngăn xếp tất cả các toán tử có độ ưu tiên cao hơn nó để đặt vào biểu thức *postfix* trước. Riêng trường hợp độ ưu tiên của toán tử đang cần đưa vào ngăn xếp bằng với độ ưu tiên của toán tử đang ở đỉnh ngăn xếp, thì toán tử ở đỉnh ngăn xếp cũng được lấy ra trước nếu các toán tử này là các toán tử kết hợp trái (Việc tính toán xử lý từ trái sang phải).

Chúng ta quan sát ba ví dụ sau đây:

a + b * c được chuyển thành a b c * + (1)
a * b **+** c được chuyển thành a b * c **+** (2)
a + b **-** c được chuyển thành a b + c **-** (3)

Ví dụ 1, dấu + vẫn ở trong ngăn xếp, và dấu * được đẩy vào ngăn xếp.
Ví dụ 2, dấu * được lấy ra khỏi ngăn xếp trước khi đưa dấu + vào.
Ví dụ 3, dấu + được lấy ra khỏi ngăn xếp trước khi đưa dấu - vào.

Trong trường hợp có dấu ngoặc, dấu mở '(' cần được lưu trong ngăn xếp cho đến khi gặp dấu đóng ')' tương ứng. Chúng ta quy ước độ ưu tiên của dấu ngoặc mở thấp nhất là hợp lý. Mọi toán tử xuất hiện sau dấu ngoặc mở đều không thể làm cho dấu này được lấy ra khỏi ngăn xếp, trừ khi dấu ngoặc đóng tương ứng được duyệt đến. Khi gặp dấu đóng ')', xem như kết thúc mọi việc tính toán trongcặp dấu (), mọi toán tử còn nằm trên dấu mở '(' trong ngăn xếp đều được lấy ra để đưa vào biểu thức dạng *postfix*. Do các dấu ngoặc không bao giờ xuất hiện trong biểu thức *postfix,* dấu '(' được đặt vào ngăn xếp để chờ dấu ')' tương ứng, khi nó được lấy ra thì sẽ bị vất bỏ. Dấu ')' để giải quyết cho dấu '(', và cũng sẽ bị vất đi.

Các toán tử +, -, *, / đều là các toán tử kết hợp từ trái sang phải. Biểu thức a - b - c (được hiểu là `(a-b)-c`) được chuyển đổi thành a b - c - (không phải a b c - -). Với một số toán tử kết hợp từ phải sang trái, chẳng hạn phép tính lũy thừa thì 2^2^3 = 2^(2^3)= 2^8=256, không phải (2^2)^3= 4^3=64, thì các xử lý trên cần được sửa đổi cho hợp lý. Chương trình hoàn chỉnh chuyển đổi biểu thức dang *infix* sang biểu thức dạng *postfix*, cũng như việc xử lý đặc biệt cho trường hợp toán tử kết hợp phải được xem như bài tập.

14.4. Giải thuật quay lui (*backtracking*)

Ngăn xếp còn được sử dụng trong các giải thuật quay lui nhằm lưu lại các thông tin đã từng duyệt qua để có thể quay ngược trở lại. Chúng ta sẽ xem xét các ví dụ sau đây.

14.4.1. Ứng dụng trong bài toán tìm đích (*goal seeking*).

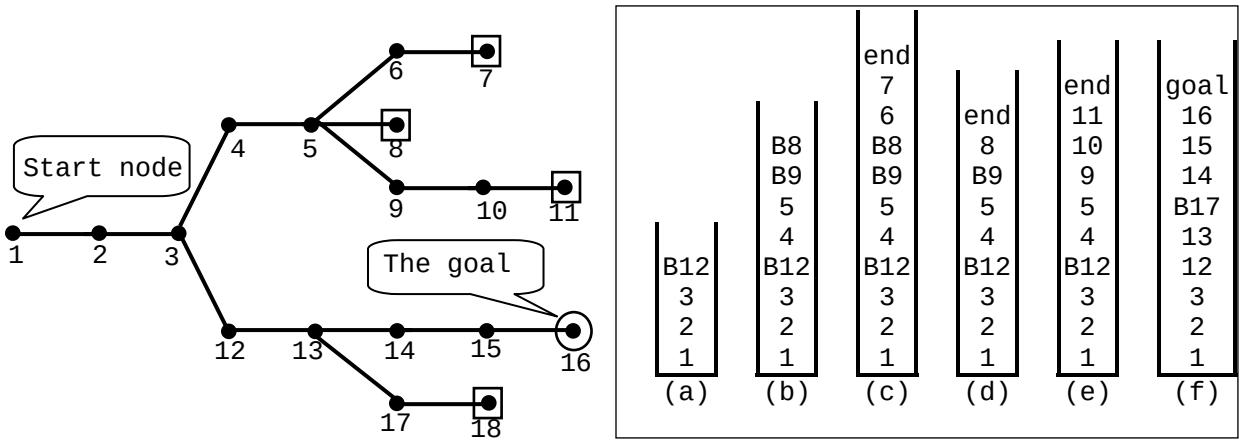
Hình 14.1 minh họa cho bài toán tìm đích. Chúng ta có một nút bắt đầu và một nút gọi là đích đến. Để đơn giản, chúng ta xét đồ thị không có chu trình và chỉ có duy nhất một đường đi từ nơi bắt đầu đến đích. Nhìn hình vẽ chúng ta có thể nhận ra ngay đường đi này. Tuy nhiên máy tính cần một giải thuật thích hợp để tìm ra được con đường này.

Chúng ta bắt đầu từ nút 1, sang nút 2 và nút 3. Tại nút 3 có 2 ngã rẽ, giả sử chúng ta đi theo đường trên, đến nút 4 và nút 5. Tại nút 5 chúng ta lại đi theo đường trên đến nút 6 và nút 7. Đến đây chúng ta không còn đường đi tiếp và cũng chưa tìm được đích cần đến, chúng ta phải quay trở lại nút 5 để chọn lối đi khác. Tại nút 8 chúng ta lại phải quay lại nút 5 để đi sang nút 9,.... Bằng cách này, từ nút 13, khi chúng ta tìm được nút 16 thì chúng ta không cần phải quay lui để thử với nút 17, 18 nữa. Giải thuật kết thúc khi tìm thấy đích đến.

Giải thuật của chúng ta cần lưu các nút để quay lại. So sánh nút 3 và nút 5, chúng ta thấy rằng trên đường đi chúng ta gặp nút 3 trước, nhưng diểm quay về để thử trước lại là nút 5. Do đó cấu trúc dữ liệu thích hợp chính là ngăn xếp với

nguyên tắc FILO của nó. Ngoài ra nếu chúng ta lưu nút 3 và nút 5 thì có sự bất tiện ở chỗ là khi quay về, thông tin lấy từ ngăn xếp không cho chúng ta biết các nhánh nào đã được duyệt qua và các nhánh nào cần tiếp tục duyệt. Do đó, tại nút 3, trước khi đi sang 4, chúng ta lưu nút 12, tại nút 5, trước khi đi sang 6 chúng ta lưu nút 8 và nút 9,...

Với giải thuật trên chúng ta có thể tìm đến đích một cách dễ dàng. Tuy nhiên, nếu bài toán yêu cầu in ra các nút trên đường đi từ nút bắt đầu đến đích thì chúng ta chưa làm được. Như vậy chúng ta cũng cần phải lưu cả các nút trên đường mà chúng ta đã đi qua. Những nút nằm trên những đoạn đường không dẫn đến đích sẽ được dỡ bỏ khỏi ngăn xếp khi chúng ta quay lui. Ở đây chúng ta gặp phải một vấn đề cũng tương đối phổ biến trong một số bài toán khác, đó là những gì chúng ta bỏ vào ngăn xếp không có cùng mục đích. Có hai nhóm thông tin khác nhau: một là các nút nằm trên đường đang đi qua, hai là các nút nằm trên các nhánh rẽ khác mà chúng ta sẽ lần lượt thử tiếp khi gặp thất bại trên con đường đang đi. Trong những trường hợp như vậy, việc giải quyết rất là dễ dàng: chúng ta dùng cách đánh dấu để phân biệt từng trường hợp, khi lấy ra khỏi ngăn xếp, căn cứ vào cách đánh dấu này chúng ta sẽ biết phải xử lý như thế nào cho thích hợp (Việc duyệt cây theo thứ tự LRN trong chương 10 nếu dùng ngăn xếp cũng là một ví dụ). Trong hình 14.1, ký tự B trong ngăn xếp cho biết đó là những nút dành cho việc quay lui (*Backtracking*) để thử với nhánh khác. Vậy khi gặp điểm cuối của một con đường không dẫn đến đích, chúng ta dỡ bỏ khỏi ngăn xếp các nút cho đến khi gặp một nút có ký tự 'B', bỏ lại nút này vào ngăn xếp (không còn ký tự 'B'), và đi tiếp các nút kế tiếp theo nút này. Cuối cùng khi gặp đích, con đường được tìm thấy chính là các nút đang lưu trong ngăn xếp mà không có ký tự 'B' ở đầu.



Hình 14.1- Ví dụ và ngăn xếp minh họa quá trình *backtracking*.

```
Algorithm seek_goal(val graph_type graph, val node_type start,
                                                  val node_type goal)
/*
pre: graph chứa đồ thị không có chu trình, trong đó có một nút start và một nút goal.
post: nếu đường đi từ start đến goal được tìm thấy sẽ được in ra theo thứ tự từ goal ngược
     về start
*/
{
   1. Stack<node_type> nodes. // node_type là kiểu dữ liệu thích hợp.
   2. node_type current_node = start
   3. boolean fail = FALSE
   4. loop ((current_node chưa là goal) AND (!fail))
      1. nodes.push(current_node)
      2. if (current_node là điểm rẽ nhánh)
         1. next_node = nút rẽ vào 1 nhánh
         2. loop (còn nhánh chưa đưa vào ngăn xếp) // đưa các nhánh
                                                   // còn lại vào ngăn xếp.
            1. nodes.push(nút rẽ vào 1 nhánh, có kèm 'B')
         3. endloop
      3. else
         1. if (tồn tại nút kế)
            1. next_node = nút kế
         2. else
            1. repeat
               1. if (nodes.empty())
                  1. fail = TRUE
               2. else
                  1. nodes.top(next_node)
                  2. nodes.pop()
               3. endif
            2. until ((fail) OR (next_node có 'B'))
         3. endif
      4. endif
      5. current_node = next_node
   5. endloop
   6. if (!fail)
      1. print("The path to your goal is:")
      2. print(current_node)  // chính là goal
      3. loop (!nodes.empty())
         1. nodes.top(current_node)
         2. nodes.pop()
         3. if (current_node không có ký tự 'B')
            1. print(current_node)
         4. endif
      4. endloop
   7. else
      1. print("Path not found!")
   8. endif
}
```

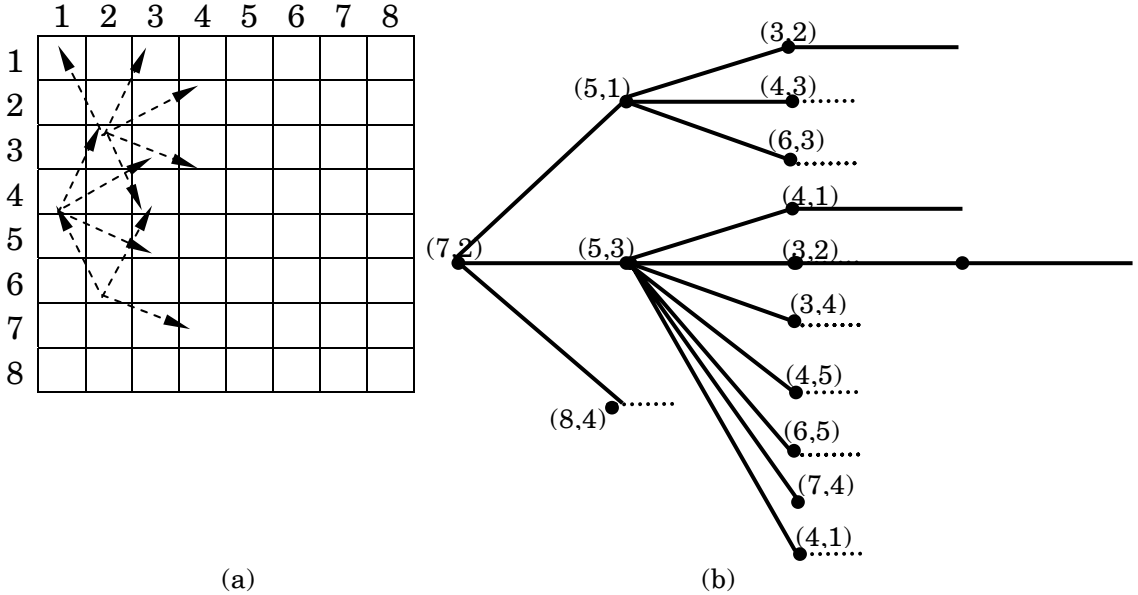
Trên đây là mã giả cho bài toán tìm đích, cấu trúc dữ liệu để chứa đồ thị chúng ta sẽ được tìm hiểu sau trong chương 13.

14.4.2. Bài toán mã đi tuần và bài toán tám con hậu

Ví dụ tiếp theo liên quan đến bài toán mã đi tuần. Thực ra bài toán tám con hậu được trình bày trong chương 6 cũng hoàn toàn tương tự. Sinh viên có thể tham khảo ý tưởng được trình bày dưới đây để giải bài toán tám con hậu sử dụng ngăn xếp thay vì đệ quy.

Với bàn cờ 64 ô, bài toán mã đi tuần yêu cầu chúng ta chỉ ra đường đi cho con mã bắt đầu từ một ô nào đó, lần lượt đi qua tất cả các ô, mà không có ô nào lập lại hơn một lần.

Từ một vị trí, con mã có tối đa là 8 vị trí chung quanh có thể đi được. Giải thuật quay lui rất gần với hướng suy nghĩ tự nhiên của chúng ta. Trong các khả năng có thể, chúng ta thường chọn ngẫu nhiên một khả năng để đi. Và với vị trí mới chúng ta cũng làm điều tương tự. Mỗi ô đi qua chúng ta đánh dấu lại. Trong quá trình thử này, nếu có lúc không còn khả năng lựa chọn nào khác vì các ô nằm trong khả năng đi tiếp đều đã được đánh dấu, chúng ta cần phải lùi lại để thử những khả năng khác. Thay vì biểu diễn đường đi cho con mã trên bàn cờ (hình 14.2a), chúng ta vẽ lại các khả năng đi và thấy chúng không khác gì so với bài toán tìm đích (hình 14.2b). Và như vậy, chúng ta thấy cách sử dụng ngăn xếp trong những bài toán dạng này hoàn toàn đơn giản và tương tự. Phương pháp quay lui trong trường hợp này đôi khi còn được gọi là phương pháp thử sai (*trial and error method*).



Hình 14.2- Bài toán mã đi tuần.

Hình trên đây minh họa bài toán mã đi tuần với điểm bắt đầu là (7,2). Đích đến không cụ thể như bài toán trên, mà đường đi cần tìm chính là đường đi nào trong đồ thị này có đủ 64 nút. Bài toán này phụ thuộc vào số ô của bàn cờ và vị

trí bắt đầu của con mã, khả năng có lời giải và có bao nhiêu lời giải chúng ta không xem xét ở đây. Chúng ta chỉ quan tâm đến cách sử dụng ngăn xếp trong những bài toán tương tự. Bản chất những bài này đều có cùng một đặc trưng của bài toán tìm đích trên một đồ thị không có chu trình. Đích đến có thể là nhiều hơn một (tương ứng với nhiều lời giải được tìm thấy) như trong bài toán tám con hậu mà cách giải đệ quy được trình bày trong chương 6. Sự khác nhau cơ bản giữa chúng chỉ là một vài kỹ thuật nho nhỏ dùng để chuyển từ dữ liệu ban đầu của bài toán sang dạng đồ thị biểu diễn các khả năng di chuyển trong quá trình tìm đến đích mà thôi.

Chương 15 – ỨNG DỤNG CỦA HÀNG ĐỢI

CTDL hàng đợi được sử dụng rất rộng rãi trong các chương trình máy tính. Đặc biệt là trong công việc của hệ điều hành khi cần xử lý các công việc một cách tuần tự. Hàng đợi trong chương 3 là một khái niệm FIFO đơn giản. Trong thực tế, người ta thường rất hay sử dụng các hàng đợi ưu tiên được trình bày trong chương 11. Chương này minh họa một số ứng dụng đơn giản của hàng đợi.

15.1. Các dịch vụ

Chúng ta có thể viết một chương trình mô phỏng việc cung cấp các dịch vụ. Chẳng hạn tại quầy bán vé các tuyến bay, có nhiều người đang đến và đang sắp hàng chờ để mua vé. Có khả năng chỉ có một nhân viên bán vé, hoặc có nhiều nhân viên bán vé đồng thời. Sinh viên hãy xem đây như là một gợi ý để viết thành một ứng dụng cho CTDL hàng đợi. Những điều thường được quan tâm là:

- Thời gian chờ đợi trung bình (*queue time*) của một khách hàng từ lúc đến cho đến lúc được bắt đầu được phục vụ.
- Thời gian phục vụ trung bình (*service time*) mà một dịch vụ được thực hiện.
- Thời gian đáp ứng trung bình (*response time*) của một khách hàng từ lúc đến cho đến lúc rời khỏi quầy (chính bằng tổng hai thời gian trên).
- Tần suất đến của khách hàng.

Dựa vào những điều trên người ta có thể điều chỉnh các kế hoạch phục vụ cho thích hợp hơn.

15.2. Phân loại

Một ví dụ về phân loại là việc sử dụng nhiều hàng đợi cùng một lúc. Tùy theo đặc điểm của các yêu cầu công việc, mỗi hàng đợi chỉ nhận các công việc cùng đặc điểm mà thôi. Như vậy đầu ra của mỗi hàng đợi sẽ là những công việc có chung đặc điểm. Việc sử dụng hàng đợi ở đây giúp ta phân loại được công việc, đồng thời trong các công việc cùng loại, chúng vẫn giữ nguyên thứ tự ban đầu giữa chúng. Hình ảnh dễ thấy nhất chính là ví dụ trên với mỗi hàng người đợi sẽ được mua vé đi cùng một tuyến bay nào đó (một ô cửa chỉ bán cho một tuyến bay nhất định). Một ví dụ khác về sự phân loại các bưu kiện tại một trung tâm chuyển phát: các bưu kiện sẽ được phân loại vào các hàng đợi dựa vào thể tích, trọng lượng, nơi đến,...., mà thứ tự giữa chúng trong một hàng đợi là không thay đổi.

15.3. Phương pháp sắp thứ tự Radix Sort

Ứng dụng hàng liên kết trong phương pháp sắp thứ tự Radix Sort được trình bày trong chương 8.

15.4. Tính trị cho biểu thức *prefix*

Để tính trị cho biểu thức dạng *prefix* người ta dùng hàng đợi. Việc tính được thực hiện lặp lại nhiều lần, mỗi lần luôn xử lý cho biểu thức từ trái sang phải như dưới đây:

- + * 9 **+ 2 8** * **+ 4 8** 6 3

- + *** 9 10** *** 12 6** 3

- **+ 90 72** 3

- 162 3

159

Mỗi lần duyệt biểu thức, chúng ta thay thế mọi toán tử mà có đủ hai toán hạng đứng ngay sau bằng kết quả của phép tính cho toán tử đó. Do thứ tự duyệt qua biểu thức luôn là từ trái sang phải, nên chúng ta có thể lưu biểu thức vào hàng đợi và sử dụng các phương thức của hàng đợi để lấy từng phần tử cũng như đưa từng phần tử vào hàng. Hiện thực chương trình được xem như bài tập cho sinh viên.

15.5. Ứng dụng phép tính trên đa thức

Đây là một ứng dụng có sử dụng CTDL ngăn xếp và hàng đợi. Trong ứng dụng này chúng ta có dịp nhìn thấy công dụng của các CTDL đã được thiết kế hoàn chỉnh. Có nhiều bài toán có thể sử dụng các CTDL hoàn chỉnh để phát triển thêm các lớp thừa kế rất tiện lợi. Ngoài ra, phương pháp phát triển dần thành một chương trình hoàn chỉnh được trình bày dưới đây cũng là một tham khảo rất tốt cho sinh viên khi làm quen với các kỹ năng thiết kế và lập trình.

15.5.1. Mục đích của ứng dụng

Trong phần 14.3 chúng ta đã viết chương trình mô phỏng một máy tính đơn giản thực hiện các phép cộng, trừ, nhân, chia và một số phép tính khác tương tự. Phần này sẽ mô phỏng một máy tính tương tự thực hiện cách tính Balan ngược cho các phép tính trên đa thức. Ý tưởng chính của giải thuật là sử dụng ngăn xếp để lưu các toán hạng. Còn hàng đợi sẽ được sử dụng để mô phỏng đa thức.

15.5.2. Chương trình

Chúng ta sẽ hiện thực một lớp đa thức (`Polynomial`) để sử dụng trong chương trình. Việc hiện thực chương trình sẽ trở nên rất đơn giản. Chương trình chính

cần khai báo một ngăn xếp để chứa các đa thức, nhận các yêu cầu tính và thực hiện.

```
int main()
/*
post: Chương trình thực hiện các phép tính toán số học cho các đa thức do người sử dụng nhập
      vào.
uses: Các lớp Stack, Polynomial và các hàm introduction, instructions,
      do_command, get_command.
*/
{
   Stack<Polynomial> stored_polynomials;
   introduction();
   instructions();
   while (do_command(get_command(), stored_polynomials));
}
```

Chương trình này hầu như tương tự với chương trình chính ở phần 14.3, hàm phụ trợ `get_command` tương tự hàm đã có.

15.5.2.1. Các phương thức của lớp `Polynomial`

Tương tự phần 14.3, thay vì nhập một con số, dấu ? chờ người sử dụng nhập vào một đa thức.

Phần lớn các việc cần làm trong chương trình này là việc hiện thực các phương thức của `Polynomial`. Chẳng hạn chúng ta cần phương thức `equals_sum` để cộng hai đa thức. Cho p, q, r là các đối tượng đa thức, `p.equals_sum(q,r)` sẽ thay p bởi đa thức tổng của hai đa thức q và r. Tương tự chúng ta có các phương thức `equals_difference`, `equals_product`, và `equals_quotient` để thực hiện các phép tính số học khác trên đa thức.

Ngoài ra, lớp `Polynomial` còn hai phương thức không thông số là `print` và `read` để xuất và đọc đa thức.

15.5.2.2. Thực hiện các yêu cầu

Hàm `do_command` nhận đối tượng ngăn xếp là tham biến do ngăn xếp cần được biến đổi trong hàm.

```
bool do_command(char command, Stack<Polynomial> &stored_polynomials)
/*
pre:  command chứa ký tự hợp lệ biểu diễn yêu cầu của người sử dụng.
post: Yêu cầu của người sử dụng được thực hiện đối với các đa thức trong ngăn xếp, hàm trả về
      true ngoại trừ trường hợp command='q' thì hàm trả về false.
uses: Các lớp Stack và Polynomial.
*/
```

```
{
   Polynomial p, q, r;
   switch (command) {

   case '?':
      p.read();
      if (stored_polynomials.push(p) == overflow)
         cout << "Warning: Stack full, lost polynomial" << endl;
      break;

   case '=':
      if (stored_polynomials.empty())
         cout << "Stack empty" << endl;
      else {
         stored_polynomials.top(p);
         p.print();
      }
      break;

   case '+':
      if (stored_polynomials.empty())
         cout << "Stack empty" << endl;
      else {
         stored_polynomials.top(p);
         stored_polynomials.pop();
         if (stored_polynomials.empty()) {
            cout << "Stack has just one polynomial" << endl;
            stored_polynomials.push(p);
         }
         else {
            stored_polynomials.top(q);
            stored_polynomials.pop();
            r.equals_sum(q, p);
            if (stored_polynomials.push(r) == overflow)
               cout << "Warning: Stack full, lost polynomial" << endl;
         }
      }
      break;
   //    Cần bổ sung các dòng lệnh để thực hiện các phép tính còn lại.

    case 'q':
      cout << "Calculation finished." << endl;
      return false;
   }
   return true;
}
```

15.5.2.3. Các mẩu chương trình và công việc kiểm tra

Cách làm sau đây minh họa ý tưởng sử dụng các mẩu tạm trong chương trình như đã trình bày trong phần 1.5.3 (các kỹ năng lập trình).

Hiện tại chúng ta chỉ phát triển chương trình vừa đủ để có thể dịch, chỉnh sửa lỗi, và kiểm tra tính đúng đắn của những phần đã viết.

Để dịch thử chương trình, chúng ta cần tạo các mẩu cho mọi phần tử còn thiếu của chương trình. Phần tử còn thiếu ở đây là lớp `Polynomial`. Do chúng ta còn chưa quyết định sẽ hiện thực các đối tượng đa thức như thế nào, chúng ta hãy chạy chương trình như một máy tính Balan ngược dành cho các số thực. Thay vào chỗ cần khai báo dữ liệu cho lớp `Polynomial`, chúng ta khai báo số thực.

```
class Polynomial {
public:
   void read();
   void print();
   void equals_sum(Polynomial p, Polynomial q);
   void equals_difference(Polynomial p, Polynomial q);
   void equals_product(Polynomial p, Polynomial q);
   ErrorCode equals_quotient(Polynomial p, Polynomial q);
private:
   double value; // mẩu tạm dùng để thử chương trình.
};
```

Phương thức `equals_quotient` cần kiểm tra phép chia 0 nên cần trả về `ErrorCode`. Hàm sau đây là một điển hình cho mẩu phương thức dùng để thử chương trình.

```
void Polynomial::equals_sum(Polynomial p, Polynomial q)
{
   value = p.value + q.value; // Chỉ viết tạm, sau sẽ viết lại đúng cho đa thức.
}
```

Việc tạo ra một bộ khung chương trình tại thời điểm này cho phép chúng ta kiểm tra xem ngăn xếp và các gói tiện ích (chứa các hàm phụ trợ) đã được kết nối một cách đúng đắn trong chương trình hay chưa. Chương trình, cùng các mẩu thử của nó, phải thực thi một cách chính xác cả với hiện thực ngăn xếp liên tục lẫn ngăn xếp liên kết.

15.5.3. Cấu trúc dữ liệu của đa thức

Chúng ta hãy quay lại nhiệm vụ chính của chúng ta là chọn lựa cách biểu diễn đa thức và viết các phương thức xử lý cho chúng. Các đa thức có dạng sau

$$3x^5 - 2x^3 + x^2 + 4$$

Thông tin quan trọng trong một đa thức là các hệ số và các số mũ của x, còn bản thân x chỉ là một biến. Chúng ta xem đa thức được tạo thành từ các số hạng (`term`), mỗi số hạng chứa một hệ số và một số mũ. Trong máy tính có thể xem đa thức là một danh sách các cặp gồm hệ số và số mũ. Chúng ta dùng `struct` để khai báo số hạng

```
struct Term {
   int degree;
   double coefficient;
   Term (int exponent = 0, double scalar = 0);
};
```

```
Term::Term(int exponent, double scalar)
/*
post: Term được khởi tạo bởi hệ số và số mũ nhận được, nếu không có thông số truyền vào thì
      hai số này được gán mặc định là 0.
*/
{
   degree = exponent;
   coefficient = scalar;
}
```

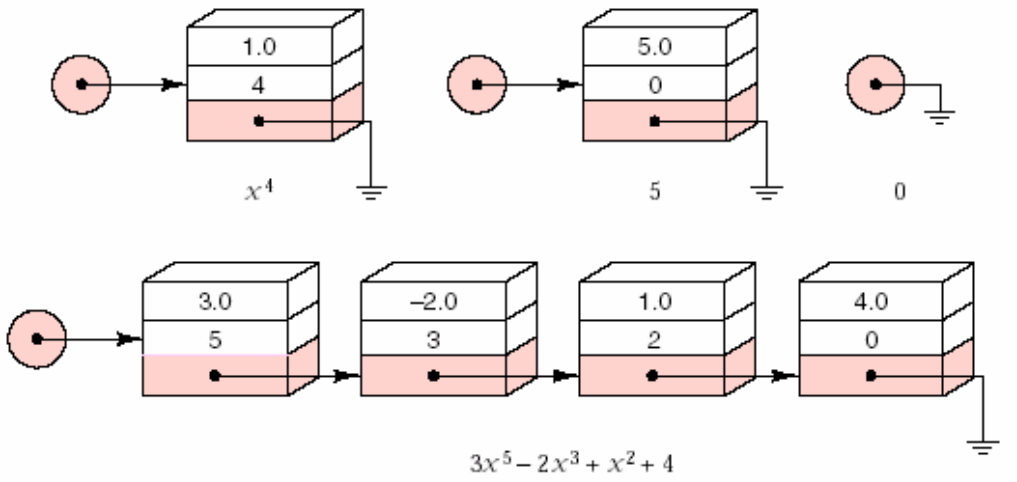
Chúng ta chưa lưu tâm về thứ tự của các số hạng trong đa thức, tuy nhiên nếu cho phép các số hạng có một thứ tự tùy ý thì chúng ta sẽ gặp khó khăn trong việc nhận ra các đa thức bằng nhau cũng như trong việc tính toán trên các đa thức. Chúng ta chọn phương án buộc các số hạng trong một đa thức phải có thứ tự giảm dần theo số mũ, đồng thời cũng không cho phép hai số hạng có cùng số mũ hoặc số hạng có hệ số bằng không. Với sự ràng buộc này, khi thực hiện một phép tính số học nào đó trên hai đa thức, chúng ta thường lần lượt xử lý cho từng cặp số hạng của hai đa thức từ trái sang phải. Các số hạng của đa thức kết quả cũng được ghi vào đa thức theo thứ tự này. Một đa thức được biểu diễn bằng một danh sách các `Term`. Như vậy, các phép tính số học có thể xem đa thức như là một `Queue`, hay chính xác hơn, như là `Extended_queue`, vì chúng ta thường xuyên cần đến các phương thức `clear` và `serve_and_retrieve`.

Chúng ta thử hỏi nên dùng hàng liên tục hay hàng liên kết. Nếu biết trước bậc của đa thức và các đa thức ít có hệ số bằng không thì chúng ta nên dùng hàng liên tục. Ngược lại, nếu không biết trước bậc, hoặc đa thức có rất ít hệ số khác không thì hàng liên kết thích hợp hơn. Hình 15.1 minh họa đa thức hiện thực bằng hàng liên kết.

Mỗi phần tử trong hàng chứa một số hạng của đa thức và chỉ có những số hạng có hệ số khác không mới được lưu vào hàng. Đa thức bằng 0 biểu diễn bởi hàng rỗng.

Với cấu trúc dữ liệu được chọn cho đa thức như trên chúng ta xây dựng lớp `Polynomial` là lớp dẫn xuất từ lớp `Extended_Queue`, chúng ta chỉ việc bổ sung các phương thức riêng của đa thức.

Để hiện thực cụ thể cho lớp dẫn xuất `Polynomial`, chúng ta cần đặt câu hỏi: một đa thức có hẳn là một `Extended_Queue` hay không?



Hình 15.1- Biểu diễn đa thức bởi một hàng liên kết các số hạng

Một `Extended_Queue` cung cấp các phương thức như `serve` chẳng hạn, phương thức này không nhất thiết và cũng không nên là phương thức của `Polynomial`. Sẽ là một điều không hay khi chúng ta cho phép người sử dụng gọi phương thức `serve` để loại đi một phần tử của `Polynomial`. Đa thức nên là một đối tượng đóng kín. Vì vậy một `Polynomial` không hẳn là một `Extended_Queue`. Mặc dù rất có lợi khi sử dụng lại các thuộc tính và các phương thức từ `Extended_Queue` cho `Polynomial`, nhưng chúng ta không thể sử dụng phép thừa kế đơn giản, vì quan hệ của hai lớp này không phải là quan hệ "*is-a*".

Quan hệ "*is-a*" là dạng thừa kế `public` trong C++. Nếu khai báo thừa kế `public` thì người sử dụng có thể xem một đa thức cũng là một hàng đợi. C++ cung cấp một dạng thừa kế thứ hai, gọi là thừa kế riêng (*private inheritance*), đây chính là điều chúng ta mong muốn. Sự thừa kế riêng cho phép lớp dẫn xuất được hiện thực bằng cách sử dụng lại lớp cơ sở, nhưng người sử dụng không gọi được các phương thức của lớp cơ sở thông qua đối tượng của lớp dẫn xuất. Quan hệ này còn được gọi là quan hệ "*is implemented of*". Chúng ta sẽ định nghĩa lớp `Polynomial` thừa kế riêng từ lớp `Extended_Queue`: các thuộc tính và các phương thức của `Extended_Queue` có thể được sử dụng trong hiện thực của lớp `Polynomial`, nhưng chúng không được nhìn thấy bởi người sử dụng.

```
class Polynomial: private Extended_queue<Term> {  //  Thừa kế riêng.

public:
   void read();
   void print() const;
   void equals_sum(Polynomial p, Polynomial q);
   void equals_difference(Polynomial p, Polynomial q);
   void equals_product(Polynomial p, Polynomial q);
   Error_code equals_quotient(Polynomial p, Polynomial q);
   int degree() const;
private:
   void mult_term(Polynomial p, Term t);
};
```

Định nghĩa trên bổ sung thêm các phương thức như `degree` trả về bậc của đa thức và `mult_term` để nhân một đa thức với một `term`.

15.5.4. Đọc và ghi các đa thức

Việc in đa thức đơn giản chỉ là duyệt qua các phần tử của hàng và in dữ liệu trong mỗi phần tử. Phương thức dưới đây in đa thức với một số xử lý cho các trường hợp đặc biệt như loại bỏ dấu + (nếu có) ở đầu đa thức, không in hệ số hoặc số mũ nếu bằng 1 và không in x^0.

```
void Polynomial::print() const
/*
post: Đa thức được in ra cout.
*/
{
   Node *print_node = front;
   bool first_term = true;

   while (print_node != NULL) {
      Term &print_term = print_node->entry;
      if (first_term) {                    // Không in dấu '+' đầu đa thức.
         first_term = false;
         if (print_term.coefficient < 0) cout << "- ";
      }
      else if (print_term.coefficient < 0) cout << " - ";
      else cout << " + ";
      double r = (print_term.coefficient >= 0)
                 ? print_term.coefficient : -(print_term.coefficient);
      if (r != 1) cout << r;
      if (print_term.degree > 1) cout << " X^" << print_term.degree;
      if (print_term.degree == 1) cout << " X";
      if (r == 1 && print_term.degree == 0) cout << " 1";
      print_node = print_node->next;
   }
   if (first_term)
      cout << "0";  //  Print 0 for an empty Polynomial.
   cout << endl;
}
```

Phương thức đọc đa thức thực hiện việc kiểm tra để các số hạng nhập vào thỏa điều kiện số mũ giảm dần và một số ràng buộc trong việc biểu diễn đa thức như đã nêu trên. Việc nhập kết thúc khi hệ số và số mũ đều nhận trị 0.

```
void Polynomial::read()
/*
post: Đa thức được đọc từ cin.
*/

{
   clear();
   double coefficient;
   int last_exponent, exponent;
   bool first_term = true;
```

```
   cout << "Enter the coefficients and exponents for the polynomial, "
        << "one pair per line.  Exponents must be in descending order."
         << endl
        << "Enter a coefficient of 0 or an exponent of 0 to terminate." <<
         endl;

   do {
      cout << "coefficient? " << flush;
      cin  >> coefficient;

      if (coefficient != 0.0) {
         cout << "exponent? " << flush;
         cin  >> exponent;
         if ((!first_term && exponent >= last_exponent) || exponent < 0) {
            exponent = 0;

            cout << "Bad exponent: Polynomial terminates without its last
                     term."
                 << endl;
         }
         else {
            Term new_term(exponent, coefficient);
            append(new_term);
            first_term = false;
         }
         last_exponent = exponent;
      }
   } while (coefficient != 0.0 && exponent != 0);
}
```

15.5.5. Phép cộng đa thức

Phần này chúng ta chỉ hiện thực phép cộng đa thức, các phép tính khác xem như bài tập.

Do các số hạng trong cả hai đa thức có số mũ giảm dần nên phép cộng rất đơn giản. Chúng ta chỉ cần duyệt qua một lần và đồng thời đối với cả hai đa thức. Nếu gặp hai số hạng của hai đa thức có số mũ bằng nhau, chúng ta cộng hai hệ số và kết quả đưa vào đa thức tổng, ngược lại, số hạng của đa thức nào có số mũ cao hơn được đưa vào tổng, phép duyệt chỉ dịch chuyển tới một bước trên đa thức này. Chúng ta chỉ phải lưu ý đến trường hợp tổng hai hệ số của hai số hạng bằng 0 thì sẽ không có phần tử mới được đưa vào đa thức tổng. Ngoài ra, vì phương thức này sẽ phá hủy các đa thức toán hạng được đưa vào nên chúng được gởi bằng tham trị.

```
void Polynomial::equals_sum(Polynomial p, Polynomial q)
/*
post: Đối tượng đa thức sẽ có trị bằng tổng hai đa thức nhận vào từ thông số.
*/
{
   clear();
   while (!p.empty() || !q.empty()) {
      Term p_term, q_term;
      if (p.degree() > q.degree()) {
         p.serve_and_retrieve(p_term);
         append(p_term);
      }
```

```
      else if (q.degree() > p.degree()) {
         q.serve_and_retrieve(q_term);
         append(q_term);
      }
      else {
         p.serve_and_retrieve(p_term);
         q.serve_and_retrieve(q_term);


         if (p_term.coefficient + q_term.coefficient != 0) {
            Term answer_term(p_term.degree,
                             p_term.coefficient + q_term.coefficient);
            append(answer_term);
         }
      }
   }
}
```

Phương thức trên bắt đầu bằng việc dọn dẹp mọi số hạng trong đa thức sẽ chứa kết quả của phép cộng. Phương thức `degree` được gọi để trả về bậc của đa thức, nếu đa thức rỗng, `degree` sẽ trả về -1.

15.5.6. Hoàn tất chương trình

15.5.6.1. Các phương thức còn thiếu

Phép trừ hai đa thức hoàn toàn tương tự phép cộng. Đối với phép nhân, trước hết chúng ta phải viết hàm nhân một đa thức với một số hạng, sau đó kết hợp với phương thức cộng hai đa thức để hoàn tất phép nhân. Phép chia đa thức phức tạp hơn rất nhiều, phép chia đa thức cho một kết quả là đa thức thương và một là đa thức số dư.

Chương 16 – ỨNG DỤNG XỬ LÝ VĂN BẢN

Phần này minh họa một ứng dụng có sử dụng cả lớp `List` và `String`. Đó là một chương trình xử lý văn bản, tuy chỉ có một vài lệnh đơn giản, nhưng nó cũng minh họa được những ý tưởng cơ bản để xây dựng những chương trình xử lý văn bản lớn và tinh tế hơn.

16.1. Các đặc tả

Chương trình xử lý văn bản của chúng ta cho phép đọc một tập tin từ đĩa vào bộ nhớ mà chúng ta gọi là vùng đệm (*buffer*). Vùng đệm này được hiện thực như một đối tượng của lớp **`Editor`**. Mỗi dòng văn bản trong đối tượng `Editor` là một `String`. Do đó lớp `Editor` sẽ được thừa kế từ lớp `List` các `String`. Các lệnh xử lý văn bản được chia làm hai nhóm: nhóm các tác vụ của `List` sẽ xử lý cho các dòng văn bản, và nhóm các tác vụ của `String` sẽ xử lý cho các ký tự trong mỗi dòng văn bản.

Tại mỗi thời điểm, người sử dụng có thể nhập hoặc các ký tự để chèn vào văn bản, hoặc các lệnh xử lý cho phần văn bản đã có. Chương trình xử lý văn bản cần biết bỏ qua những ký tự nhập không hợp lệ, nhận biết các lệnh, hoặc hỏi lại người sử dụng trước khi thực hiện các lệnh quan trọng (chẳng hạn như xóa toàn bộ vùng đệm).

Chương trình xử lý văn bản có các lệnh dưới đây. Mỗi lệnh sẽ được người sử dụng nhập vào khi có dấu nhắc '??' và có thể nhập chữ hoa hoặc chữ thường.

'R' (*Read*) Đọc tập tin văn bản vào vùng đệm. Tên tập tin văn bản đã được chỉ ra khi chạy chương trình. Nội dung có sẵn trong vùng đệm được xóa sạch. Dòng đầu tiên của văn bản được xem là dòng hiện tại.

'W'(*Write*) Ghi nội dung trong vùng đệm vào tập tin văn bản có tên đã được chỉ ra khi chạy chương trình. Vùng đệm cũng như dòng hiện tại đều không đổi.

'I' (*Insert*) Thêm một dòng mới. Người sử dụng có thể nhập số thứ tự của dòng mới sẽ được thêm vào.

'D' (*Delete*) Xóa dòng hiện tại và chuyển đến dòng kế.

'F' (*Find*) Bắt đầu từ dòng hiện tại, tìm dòng đầu tiên có chứa chuỗi ký tự do người sử dụng yêu cầu.

'L' (*Length*) Cho biết số ký tự có trong dòng hiện tại và số dòng có trong vùng đệm.

'C' (*Change*) Đổi một chuỗi ký tự sang một chuỗi ký tự khác. Chỉ đổi trong dòng hiện tại.

'Q' (*Quit*) Thoát khỏi chương trình.

'H' (*Help*) In giải thích về các lệnh. Có thể dùng '?' thay cho 'H'.

'N' (*Next*) Chuyển sang dòng kế trong vùng đệm.
'P' (*Previous*) Trở về dòng trước trong vùng đệm.
'B' (*Beginning*) Chuyển đến dòng đầu tiên trong vùng đệm.
'E' (*End*) Chuyển đến dòng cuối trong vùng đệm.
'G' (*Go*) Chuyển đến dòng có số thứ tự do người sử dụng yêu cầu.
'S' (*Subtitute*) Thay dòng hiện tại bởi dòng do người sử dụng nhập vào. Chương trình sẽ in dòng sẽ bị thay thế để kiểm tra lại và hỏi người sử dụng nhập dòng mới.
'V' (*View*) Xem toàn bộ nội dung trong vùng đệm.

16.2. Hiện thực

16.2.1. Chương trình chính

Nhiệm vụ đầu tiên của chương trình chính là sử dụng các thông số nhập vào từ dòng lệnh để mở tập tin đọc và tập tin ghi. Cách sử dụng chương trình:

```
edit  infile  outfile
```

trong đó `infile` và `outfile` là tên tập tin đọc và tên tập tin ghi tương ứng. Khi các tập tin đã mở thành công, chương trình khai báo một đối tượng `Editor` gọi là `buffer`, lặp lại việc chạy phương thức `get_command` của `Editor` để đọc các lệnh rồi xử lý các lệnh này.

```
int main(int argc, char *argv[]) // count, values of command-line arguments
/*
pre:  Thông số của dòng lệnh là tên tập tin đọc và tập tin ghi.
post: Chương trình đọc nội dung từ tập tin đọc, cho phép soạn thảo, chỉnh sửa văn bản, và ghi
      vào tập tin ghi.
uses: Các phương thức của lớp Editor.
*/
{
   if (argc != 3) {
      cout << "Usage:\n\t edit  inputfile  outputfile" << endl;
      exit (1);
   }
   ifstream file_in(argv[1]);              // Khai báo và mở tập tin đọc.
   if (file_in == 0) {
      cout << "Can't open input file " << argv[1] << endl;
      exit (1);
   }
   ofstream file_out(argv[2]);             // Khai báo và mở tập tin ghi.
   if (file_out == 0) {
      cout << "Can't open output file " << argv[2] << endl;
      exit (1);
   }
   Editor buffer(&file_in, &file_out);
   while (buffer.get_command())
      buffer.run_command();
}
```

16.2.2. Đặc tả lớp Editor

Lớp Editor cần chứa một List các đối tượng String, và cho phép các tác vụ di chuyển theo cả hai hướng của List một cách hiệu quả. Chúng ta cũng không biết trước vùng đệm sẽ phải lớn bao nhiêu, do đó chúng ta sẽ khai báo lớp Editor dẫn xuất từ hiện thực danh sách liên kết kép (*doubly linked list*). Lớp dẫn xuất này cần bổ sung thêm hai phương thức get_command và run_command mà chương trình chính sẽ gọi. Ngoài ra lớp Editor còn cần thêm thuộc tính để chứa lệnh từ người sử dụng (user_command) và các tham chiếu đến dòng nhập và xuất (infile và outfile).

```
class Editor:public List<String> {
public:
   Editor(ifstream *file_in, ofstream *file_out);
   bool get_command();
   void run_command();
private:
   ifstream *infile;
   ofstream *outfile;
   char user_command;

//  Các hàm phụ trợ
   Error_code next_line();
   Error_code previous_line();
   Error_code goto_line();
   Error_code insert_line();
   Error_code substitute_line();
   Error_code change_line();
   void read_file();
   void write_file();
   void find_string();
};
```

Trong đặc tả trên chúng ta còn thấy một số hàm phụ trợ để hiện thực các lệnh xử lý văn bản khác nhau.

Constructor thực hiện nối dòng nhập và dòng xuất với đối tượng của lớp Editor.

```
Editor::Editor(ifstream *file_in, ofstream *file_out)
/*
post: Khởi tạo đối tượng Editor với trị cho hai thuộc tính infile, outfile.
*/
{
   infile = file_in;
   outfile = file_out;
}
```

16.2.3. Nhận lệnh từ người sử dụng

Do chương trình xử lý văn bản phải biết bỏ qua những ký tự nhập không hợp lệ, nên các lệnh nhập vào phải được kiểm tra kỹ lưỡng. Chương trình dùng hàm `tolower` chuyển ký tự hoa thành ký tự thường có trong thư viện `<cctype>`, cho phép người sử dụng có thể nhập chữ hoa hoặc chữ thường. `Get_command` sẽ in dòng hiện tại, hiện dấu nhắc chờ lệnh, đổi lệnh sang ký tự thường.

```
bool Editor::get_command()
/*
post: Gán trị cho thuộc tính user_command; trả về true trừ khi người sử dụng gõ 'q'
uses: Hàm tolower của thư viện C..
*/
{
   if (current != NULL)
      cout << current_position << " : "
           << current->entry.c_str() << "\n??" << flush;
   else
      cout << "File is empty. \n??" << flush;

   cin >> user_command;// Bỏ qua các khoảng trắng và nhận lệnh của người sử dụng
   user_command = tolower(user_command);
   while (cin.get() != '\n');          //  Bỏ qua phím "enter"
   if (user_command == 'q')
      return false;
   else
      return true;
}
```

16.2.4. Thực hiện lệnh

Phương thức `run_command` chứa lệnh `switch` để chọn các hàm khác nhau tương ứng với các lệnh cần thực hiện. Một vài lệnh trong số này (tựa như `remove`) là các phương thức của `List`. Những lệnh khác dựa trên các tác vụ xử lý của `List` nhưng có bổ sung xử lý những yêu cầu của người sử dụng.

```
void Editor::run_command()
/*
post: Lệnh trong user_command được thực hiện.
uses: Các phương thức và các hàm phụ trợ của các lớp Editor,
      String, vá các hàm xử lý chuỗi ký tự.
*/
{
   String temp_string;
   switch (user_command) {
   case 'b':
      if (empty())
         cout << " Warning: empty buffer " << endl;
      else
         while (previous_line() == success);
      break;
   case 'c':
```

```
      if (empty())
         cout << " Warning: Empty file" << endl;
      else if (change_line() != success)
         cout << " Error: Substitution failed " << endl;
      break;

   case 'd':
      if (remove(current_position, temp_string) != success)
         cout << " Error: Deletion failed " << endl;
      break;

   case 'e':
      if (empty())
         cout << " Warning: empty buffer " << endl;
      else
         while (next_line() == success)
            ;
      break;

   case 'f':
      if (empty())
         cout << " Warning: Empty file" << endl;
      else
         find_string();
      break;

   case 'g':
      if (goto_line() != success)
         cout << " Warning: No such line" << endl;
      break;

   case '?':
   case 'h':
      cout  << "Valid commands are: b(egin) c(hange) d(el) e(nd)"
            << endl
            << "f(ind) g(o) h(elp) i(nsert) l(ength) n(ext) p(rior) "
            << endl
            << "q(uit) r(ead) s(ubstitute) v(iew) w(rite) " << endl;

   case 'i':
      if (insert_line() != success)
         cout << " Error: Insertion failed " << endl;
      break;

   case 'l':
      cout << "There are " << size() << " lines in the file." << endl;
      if (!empty())
         cout << "Current line length is "
              << strlen((current->entry).c_str()) << endl;
      break;

   case 'n':
      if (next_line() != success)
         cout << " Warning: at end of buffer" << endl;
      break;
   case 'p':
      if (previous_line() != success)
         cout << " Warning: at start of buffer" << endl;
      break;
```

```
   case 'r':
      read_file();
      break;

   case 's':
      if (substitute_line() != success)
         cout << " Error: Substitution failed " << endl;
      break;

   case 'v':
      traverse(write);
      break;

   case 'w':
      if (empty())
         cout << " Warning: Empty file" << endl;
      else
         write_file();
      break;

   default :
      cout << "Press h or ? for help or enter a valid command: ";
   }
}
```

16.2.5. Đọc và ghi tập tin

Tập tin sẽ được đọc và ghi đè lên vùng đệm. Nếu vùng đệm không rỗng, cần có thông báo hỏi lại người sử dụng trước khi thực hiện lệnh.

```
void Editor::read_file()
/*
pre:  Nếu Editor không rỗng thì người sử dụng sẽ trả lời có chép đè nội dung mới lên hay
      không.
post: Đọc tập tin đọc vào Editor. Nội dung cũ nếu có trong Editor sẽ bị chép đè.
uses: Các phương thức và các hàm của String và Editor.
*/

{
   bool proceed = true;
   if (!empty()) {
      cout << "Buffer is not empty; the read will destroy it." << endl;
      cout << " OK to proceed? " << endl;
      if (proceed = user_says_yes()) clear();
   }

  int line_number = 0, terminal_char;
  while (proceed) {
      String in_string = read_in(*infile, terminal_char);
      if (terminal_char == EOF) {
         proceed = false;
         if (strlen(in_string.c_str()) > 0)
              insert(line_number , in_string);
      }
      else insert(line_number++, in_string);
   }
}
```

16.2.6. Chèn một hàng

Để chèn một dòng mới, trước hết chúng ta đọc một chuỗi ký tự nhờ phương thức `read_in` của lớp `String` trong phần 5.3. Sau đó chuỗi ký tự được chèn vào `List` nhờ phương thức insert của `List`. Chúng ta không cần kiểm tra vùng đệm đã đầy hay chưa do các tác vụ của `List` đã chịu trách nhiệm về việc này.

```
Error_code Editor::insert_line()
/*
post: Mộ dòng do người sử dụng gõ vào sẽ được chèn vào vị trí theo yêu cầu.
uses: Các phương thức và các hàm của String và Editor.
*/
{
   int line_number;
   cout << " Insert what line number? " << flush;
   cin  >> line_number;
   while (cin.get() != '\n');
   cout << " What is the new line to insert? " << flush;
   String to_insert = read_in(cin);
   return insert(line_number, to_insert);
}
```

16.2.7. Tìm một chuỗi ký tự

Đây là một công việc tương đối khó. Việc tìm một chuỗi ký tự do người sử dụng yêu cầu được thực hiện trên toàn vùng đệm. Hàm `strstr` của `String` sẽ tìm chuỗi ký tự được yêu cầu trong dòng hiện tại trước, nếu không có sẽ tìm tiếp ở các dòng kế tiếp trong vùng đệm. Nếu tìm thấy, dòng có chuỗi ký tự cần tìm sẽ được hiển thị và trở thành dòng hiện tại, dấu '^' sẽ chỉ ra vị trí chuỗi ký tự được tìm thấy.

```
void Editor::find_string()
/*
pre:  Editor đang chứa văn bản.
post: Chuỗi ký tự được yêu cầu sẽ được tìm từ dòng hiện tại trở đi. Nếu tìm thấy thì hiện dòng
      chứa chuỗi ký tự đó, đồng thời chỉ ra chuỗi ký tự.
uses: Các phương thức và các hàm của String và Editor.
*/
{
   int index;
   cout << "Enter string to search for:" << endl;
   String search_string = read_in(cin);
   while ((index = strstr(current->entry, search_string)) == -1)
      if (next_line() != success) break;
   if (index == -1) cout << "String was not found.";
   else {
      cout << (current->entry).c_str() << endl;
      for (int i = 0; i < index; i++)
         cout << " ";
      for (int j = 0; j < strlen(search_string.c_str()); j++)
         cout << "^";
   }
   cout << endl;
}
```

16.2.8. Biến đổi chuỗi ký tự

Hàm `change_line` nhận một chuỗi ký tự cần thay thế từ người sử dụng. Khi chuỗi ký tự này được tìm thấy trong dòng hiện tại, người sử dụng sẽ được yêu cầu nhập chuỗi ký tự để thay thế. Cuối cùng là một loạt các tác vụ của `String` và `C-String` được thực hiện để loại chuỗi ký tự cũ và thế chuỗi ký tự mới vào.

```
Error_code Editor::change_line()
/*
pre:  Editor đang chứa văn bản.
post: Nếu chuỗi ký tự được yêu cầu được tìm thấy trong dòng hiện tại thì sẽ được thay thế bởi
      chuỗi ký tự khác (cũng do người sử dụng gõ vào), rả về success; ngược lại trả về fail.
uses: Các phương thức và các hàm của String và Editor.
*/
{
   Error_code result = success;
   cout << " What text segment do you want to replace? " << flush;
   String old_text = read_in(cin);
   cout << " What new text segment do you want to add in? " << flush;
   String new_text = read_in(cin);

   int index = strstr(current->entry, old_text);
   if (index == -1) result = fail;
   else {
      String new_line;
      strncpy(new_line, current->entry, index);
      strcat(new_line, new_text);
      const char *old_line = (current->entry).c_str();
            strcat(new_line, (String)(old_line + index +                    +
                                                  strlen(old_text.c_str())));
      current->entry = new_line;
   }
   return result;
}
```

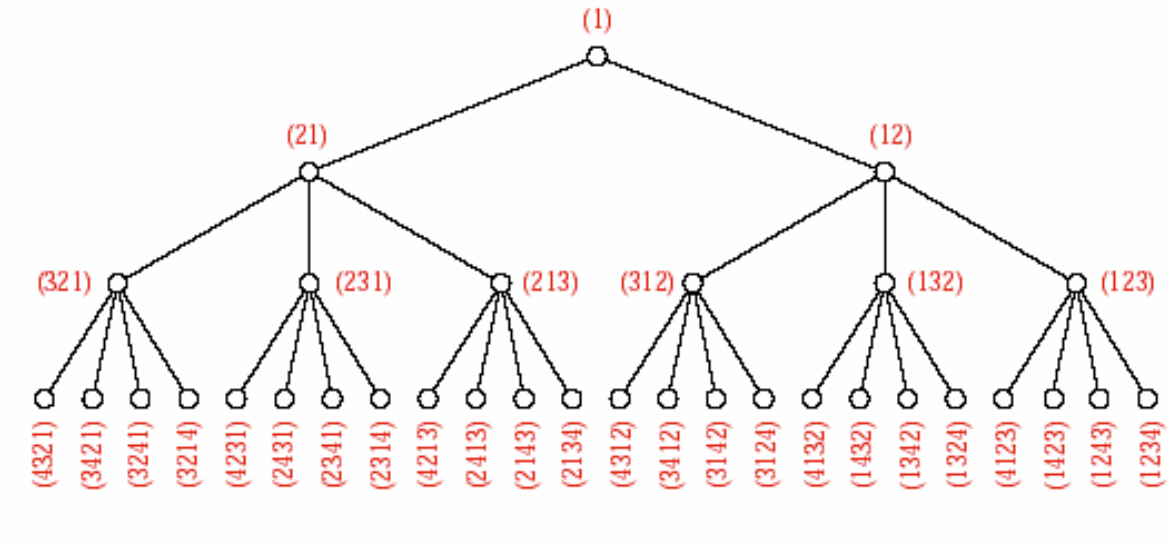
Lệnh `strcat(new_line, (String)(old_line+index+strlen(old_text.c_str()))` tìm con trỏ tạm chỉ vị trí bắt đầu phần còn lại ngay sau chuỗi ký tự được thay thế trong dòng cũ, con trỏ này sẽ được chuyển đổi thành đối tượng `String` và nối với `new_line`.

Chương 17 – ỨNG DỤNG SINH CÁC HOÁN VỊ

Ứng dụng này minh họa sự sử dụng cả hai loại danh sách: danh sách tổng quát và DSLK trong mảng liên tục. Ứng dụng này sẽ sinh ra n! cách hoán vị của n đối tượng một cách hiệu quả nhất. Chúng ta gọi các hoán vị của n đối tượng khác nhau là tất cả các phương án thiết lập chúng theo mọi thứ tự có thể có.

Chúng ta có thể chọn bất kỳ đối tượng nào trong n đối tượng đặt tại vị trí đầu tiên, sau đó có thể chọn bất kỳ trong n-1 đối tượng còn lại đặt tại vị trí thứ hai, và cứ thế tiếp tục. Các chọn lựa này độc lập nhau nên tổng số cách chọn lựa là:

```
n x (n-1) x (n-2) x ... x 2 x 1 = n!
```



Hình 17.1- Sinh các hoán vị cho {1, 2, 3, 4}

17.1. Ý tưởng

Chúng ta xác định các hoán vị qua các nút trên cây. Đầu tiên chỉ có 1 ở gốc cây. Chúng ta có hai hoán vị của {1, 2} bằng cách ghi 2 bên trái, sau đó bên phải của 1. Tương tự, sáu hoán vị của {1, 2, 3} có được từ (2, 1) và (1, 2) bằng cách thêm 3 vào cả ba vị trí có thể (trái, giữa, phải). Như vậy các hoán vị của {1, 2, ..., k} có được như sau:

Đối với mỗi hoán vị của {1, 2, ..., k-1} chúng ta đưa các phần tử vào một list. Sau đó chèn k vào mọi vị trí có thể trong list đó để có được k hoán vị khác nhau của {1, 2, ..., k}.

Giải thuật này minh họa việc sử dụng đệ quy để hoàn thành các công việc đã tạm hoãn. Chúng ta sẽ viết một hàm, đầu tiên là thêm 1 vào một danh sách rỗng,

sau đó gọi đệ quy để thêm lần lượt các số khác từ 2 đến n vào danh sách. Lần gọi đệ quy đầu tiên sẽ thêm 2 vào danh sách chỉ chứa có 1, giả sử thêm 2 bên trái của 1, và trì hoãn các khả năng thêm khác (như là thêm 2 bên phải của 1), để gọi đệ quy tiếp. Cuối cùng, lần gọi đệ quy thứ n sẽ thêm n vào danh sách. Bằng cách này, bắt đầu từ một cấu trúc cây, chúng ta phát triển một giải thuật trong đó cây này trở thành một cây đệ quy.

17.2. Tinh chế

Chúng ta sẽ phát triển giải thuật trên cụ thể hơn. Hàm thêm các số từ 1 đến n để sinh `n!` hoán vị sẽ được gọi như sau:

```
permute(1, n)
```

Giả sử đã có k-1 số đã được thêm vào danh sách, hàm sau sẽ thêm các số còn lại từ k đến n vào danh sách:

```
void permute(int k,int n)
/*
pre:  Từ 1 đến k - 1 đã có trong danh sách các hoán vị;
post: Chèn các số nguyên từ k đến n vào danh sách các hoán vị.
*/
{
   for //  với mỗi vị trí trong k vị trí có thể trong danh sách.
   {
       //  Chèn k vào vị trí này.
       if (k == n) process_permutation;
       else permute(k + 1, n);
       //  Lấy k ra khỏi vị trí vừa chèn.
   }
}
```

Khi có được một hoán vị đầy đủ của `{1, 2, ..., n}`, chúng ta có thể in kết quả, hoặc gởi kết quả như là thông số vào cho một bài toán nào khác, đó là nhiệm vụ của hàm `process_permutation`.

17.3. Thủ tục chung

Để chuyển giải thuật thành chương trình C++, chúng ta có các tên biến như sau: danh sách các số nguyên **`permutation`** chứa hoán vị của các số; **`new_entry`**, thay cho k, là số nguyên sẽ được thêm vào; và **`degree`**, thay cho n, là số các phần tử cần hoán vị.

```
void permute(int new_entry, int degree, List<int> &permutation)
/*
pre:  permutation chứa 1 hoán vị với các phần tử từ 1 đến new_entry - 1.
post: Mọi hoán vị của degree phần tử được tạo nên từ hoán vị đã có và sẽ được xử lý trong hàm
      process_permutation.
uses: hàm permute một cách đệ quy, hàm process_permutation, và các hàm của List.
*/
{
   for (int current = 0; current < permutation.size() + 1; current++) {
      permutation.insert(current, new_entry);
      if (new_entry == degree)
         process_permutation(permutation);
      else
         permute(new_entry + 1, degree, permutation);
      permutation.remove(current, new_entry);
   }
}
```

Hàm trên đây có thể sử dụng với bất kỳ hiện thực nào của danh sách mà chúng ta đã làm quen (DSLK sử dụng con trỏ, danh sách liên tục, DSLK trong mảng liên tục). Việc xây dựng ứng dụng đầy đủ để sinh các hoán vị xem như bài tập. Tuy nhiên chúng ta sẽ thấy rằng ưu điểm của DSLK trong mảng liên tục rất thích hợp đối với bài toán này trong phần tiếp theo dưới đây.o3

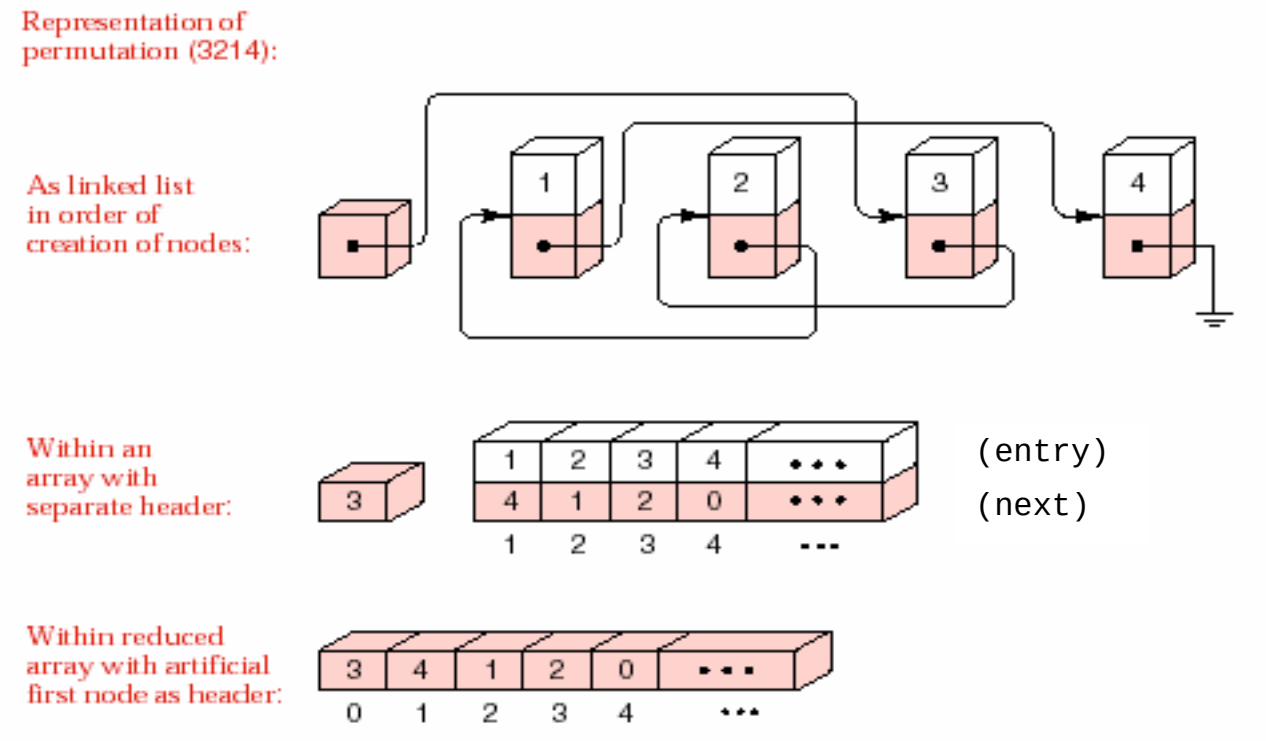
17.4. Tối ưu hóa cấu trúc dữ liệu để tăng tốc độ cho chương trình sinh các hoán vị

n! tăng rất nhanh khi n tăng. Do đó số hoán vị có được sẽ rất lớn. Ứng dụng trên là một trong các ứng dụng mà sự tối ưu hóa để tăng tốc độ chương trình rất đáng để trả giá, ngay cả khi ảnh hưởng đến tính dễ đọc của chương trình. Chúng ta sẽ dùng DSLK trong mảng liên tục có kèm một chút cải tiến cho bài toán trên.

Chúng ta hãy xem xét một vài cách tổ chức dữ liệu theo hướng làm tăng tốc độ chương trình càng nhanh càng tốt. Chúng ta sử dụng một danh sách để chứa các số cần hoán vị. Mỗi lần gọi đệ quy đều phải cập nhật các phần tử trong danh sách. Do chúng ta phải thường xuyên thêm và loại phần tử của danh sách, DSLK tỏ ra thích hợp hơn danh sách liên tục. Mặt khác, do tổng số phần tử trong danh sách không bao giờ vượt quá n, chúng ta nên sử dụng DSLK trong mảng liên tục thay vì DSLK trong bộ nhớ động.

Hình 17.2 minh họa cách tổ chức cấu trúc dữ liệu. Hình trên cùng là DSLK cho hoán vị (3, 2, 1, 4). Hình bên dưới biểu diễn hoán vị này trong DSLK trong mảng liên tục. Đặc biệt trong hình này, chúng ta nhận thấy, trị của phần tử được thêm vào cũng chính bằng chỉ số của phần tử trong array, nên việc lưu các trị này không cần thiết nữa. (Chúng ta chú ý rằng, trong giải thuật đệ quy, các số được thêm vào danh sách theo thứ tự tăng dần, nên mỗi phần tử sẽ chiếm vị trí trong mảng đúng bằng trị của nó; các hoán vị khác nhau của các phần tử này được phân

biệt bởi thứ tự của chúng trong danh sách, đó chính là sự khác nhau về cách nối các tham chiếu). Cuối cùng chỉ còn các tham chiếu là cần lưu trong mảng (hình dưới cùng trong hình 17.2). `Node 0` dùng để chứa đầu vào của DSLK trong mảng liên tục. Trong chương trình dưới đây chúng ta viết lại các công việc thêm và loại phần tử trong danh sách thay vì gọi các phương thức của danh sách để tăng hiệu quả tối đa.



Hình 17.2 – Cấu trúc dữ liệu chứa hoán vị

17.5. Chương trình

Chúng ta có hàm `permute` đã được cải tiến:

```
void permute(int new_entry, int degree, int *permutation)
/*
pre:  permutation chứa 1 hoán vị với các phần tử từ 1 đến new_entry - 1.
post: Mọi hoán vị của degree phần tử được tạo nên từ hoán vị đã có và sẽ được xử lý trong hàm
      process_permutation.
uses: hàm permute một cách đệ quy, hàm process_permutation.
*/
{
   int current = 0;


   do {
      permutation[new_entry] = permutation[current];
      permutation[current] = new_entry;
```

```
      if (new_entry == degree)
         process_permutation(permutation);
      else
         permute(new_entry + 1, degree, permutation);
      permutation[current] = permutation[new_entry];
      current = permutation[current];
   } while (current != 0);
}
```

Chương trình chính thực hiện các khai báo và khởi tạo:

```
main()
/*
pre:  Người sử dụng nhập vào degree là số phần tử cần hoán vị.
post: Mọi hoán vị của degree phần tử được in ra.
*/
{
   int degree;
   int permutation[max_degree + 1];

   cout << "Number of elements to permute? ";
   cin  >> degree;

   if (degree < 1 || degree > max_degree)
      cout << "Number must be between 1 and " << max_degree << endl;
   else {
      permutation[0] = 0;
      permute(1, degree, permutation);
   }
}
```

Danh sách **permutation** làm thông số cho hàm **process_permutation** chứa cách nối kết các phần tử trong một hoán vị, chúng ta có thể in các số nguyên của một cách hoán vị như sau:

```
void process_permutation(int *permutation)
/*
pre:  permutation trong cấu trúc liên kết bởi các chỉ số.
post: permutation được in ra.
*/
{
   int current = 0;
   while (permutation[current] != 0) {
      cout << permutation[current] << " ";
      current = permutation[current];
   }
   cout << endl;
}
```

Chương 18 – ỨNG DỤNG DANH SÁCH LIÊN KẾT VÀ BẢNG BĂM

Đây là một ứng dụng có sử dụng CTDL danh sách và bảng băm. Thông qua ứng dụng này sinh viên có dịp nâng cao kỹ năng thiết kế hướng đối tượng, giải quyết bài toán từ ngoài vào trong. Ngoài ra, đây cũng là một ví dụ rất hay về việc sử dụng một CTDL đúng đắn không những đáp ứng được yêu cầu bài toán mà còn làm tăng hiệu quả của chương trình lên rất nhiều.

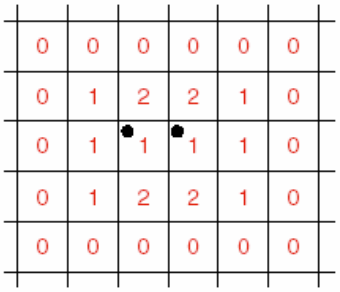
18.1. Giới thiệu về chương trình `Game_Of_Life`

`Game_Of_Life` là một chương trình giả lập một sự tiến triển của sự sống, không phải là một trò chơi với người sử dụng. Trên một lưới chữ nhật không có giới hạn, mỗi ô hoặc là ô trống hoặc đang có một tế bào chiếm giữ. Ô có tế bào được gọi là ô sống, ngược lại là ô chết. Mỗi thời điểm ổn định của toàn bộ lưới chúng ta gọi là một trạng thái. Để chuyển sang trạng thái mới, một ô sẽ thay đổi tình trạng sống hay chết tùy thuộc vào số ô sống chung quanh nó trong trạng thái cũ theo các quy tắc sau:

1. Một ô có tám ô kế cận.
2. Một ô đang sống mà không có hoặc chỉ có 1 ô kế cận sống thì ô đó sẽ chết do đơn độc.
3. Một ô đang sống mà có từ 4 ô kế cận trở lên sống thì ô đó cũng sẽ chết do quá đông.
4. Một ô đang sống mà có 2 hoặc 3 ô kế cận sống thì nó sẽ sống tiếp trong trạng thái sau.
5. Một ô đang chết trở thành sống trong trạng thái sau nếu nó có chính xác 3 ô kế cận sống.
6. Sự chuyển trạng thái của các ô là đồng thời, có nghĩa là căn cứ vào số ô kế cận sống hay chết trong một trạng thái để quyết định sự sống chết của các ô ở trạng thái sau.

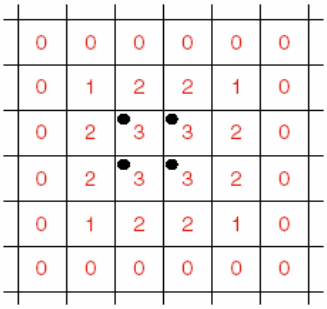
18.2. Các ví dụ

Chúng ta gọi một đối tượng lưới chứa các ô sống và chết như vậy là một cấu hình. Trong hình 18.1, con số ở mỗi ô biểu diễn số ô sống chung quanh nó, theo quy tắc thì cấu hình này sẽ không còn ô nào sống ở trạng thái sau. Trong khi đó cấu hình ở hình 18.2 sẽ bền vững và không bao giờ thay đổi.

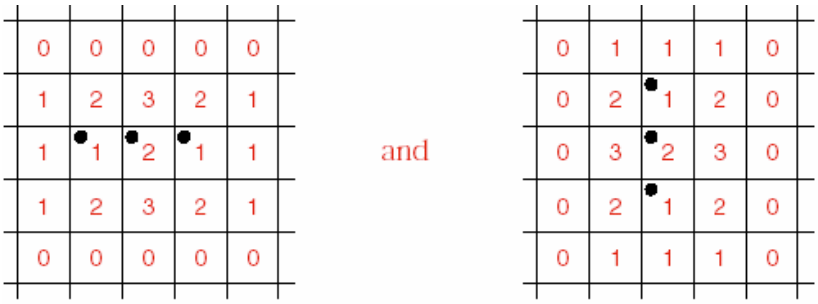


Hình 18.1- Một trang thái của Game of Life

Với một trạng thái khởi đầu nào đó, chúng ta khó lường trước được điều gì sẽ xảy ra. Một vài cấu hình đơn giản ban đầu có thể biến đổi qua nhiều bước để thành các cấu hình phức tạp hơn nhiều, hoặc chết dần một cách chậm chạp, hoặc sẽ đạt đến sự bền vững, hoặc chỉ còn là sự chuyển đổi lặp lại giữa một vài trạng thái.



Hình 18.2 – Cấu hình có trạng thái bền vững



Hình 18.3 – Hai cấu hình này luân phiên thay đổi nhau.

18.3. Giải thuật

Mục đích của chúng ta là viết một chương trình hiển thị các trạng thái liên tiếp nhau của một cấu hình từ một trạng thái ban đầu nào đó.

Giải thuật:

- Khởi tạo một cấu hình ban đầu có một số ô sống.
- In cấu hình đã khởi tạo.
- Trong khi người sử dụng vẫn còn muốn xem sự biến đổi của các trạng thái:
  - Cập nhật trạng thái mới dựa vào các quy tắc của chương trình.
  - In cấu hình.

Chúng ta sẽ xây dựng lớp **Life** mà đối tượng của nó sẽ có tên là **configuration**. Đối tượng này cần 3 phương thức: **initialize()** để khởi tạo, **print()** để in trạng thái hiện tại và **update()** để cập nhật trạng thái mới.

18.4. Chương trình chính cho Game_Of_Life

```
#include "utility.h"
#include "life.h"

int main()  //  Chương trình Game_Of_Life.
/*
pre:  Người sử dụng cho biết trạng thái ban đầu của cấu hình.
post: Chương trình in các trạng thái thay đổi của cấu hình cho đến khi người sử dụng muốn
      ngưng chương trình. Cách thức thay đổi trạng thái tuân theo các quy tắc của trò chơi.
uses: Lớp Life với các phương thức initialize(), print(), update().
      Các hàm phụ trợ instructions(),  user_says_yes().
*/

{
   Life configuration;
   instructions();
   configuration.initialize();
   configuration.print();
   cout << "Continue viewing new generations? " << endl;
   while (user_says_yes()) {
      configuration.update();
      configuration.print();
      cout << "Continue viewing new generations? " << endl;
   }
}
```

Với chương trình Life này chúng ta cần hiện thực những phần sau:

- Lớp **Life**.
- Phương thức **initialize()** khởi tạo cấu hình của **Life.**
- Phương thức **print()** hiển thị cấu hình của **Life**.
- Phương thức **update()** cập nhật đối tượng **Life** chứa cấu hình ở trạng thái mới.
- Hàm **user_says_yes()** để hỏi người sử dụng có tiếp tục xem trạng thái kế tiếp hay không.
- Hàm **instruction()** hiển thị hướng dẫn sử dụng chương trình.

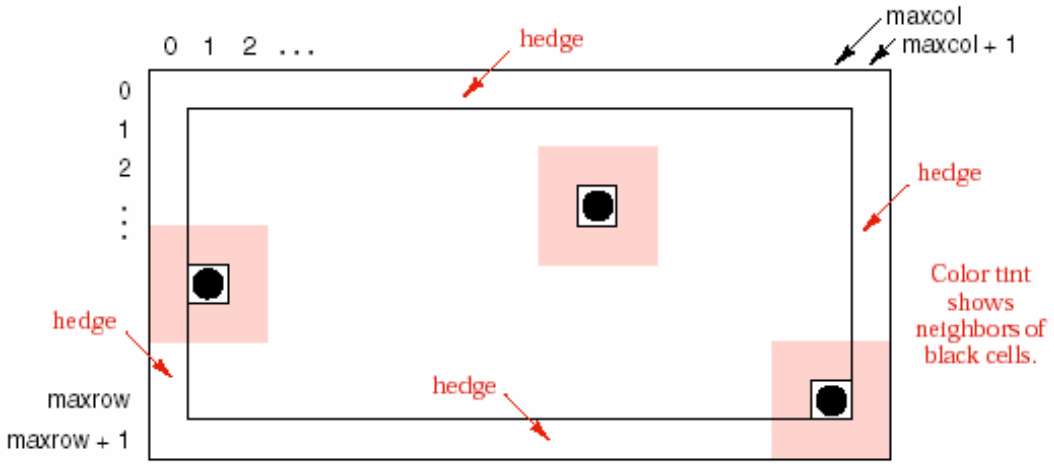
Với cách phác thảo này chúng ta có thể chuyển sang giai đoạn kế, đó là chọn lựa cách tổ chức dữ liệu để hiện thực lớp Life.

18.4.1. Phiên bản thứ nhất cho lớp `Life`

Trong phiên bản thứ nhất này, chúng ta chưa sử dụng một lớp CTDL có sẵn nào, mà chỉ suy nghĩ đơn giản rằng đối tượng `Life` cần một mảng hai chiều các số nguyên để biểu diễn lưới các ô. Trị 1 biểu diễn ô sống và trị 0 biểu diễn ô chết. Kích thước mảng lấy thêm bốn biên dự trữ để việc đếm số ô sống kế cận được thực hiện dễ dàng cho cả các ô nằm trên cạnh biên hay góc. Tất nhiên với cách chọn lựa này chúng ta đã phải lơ qua một đòi hỏi của chương trình: đó là lưới chữ nhật phải không có giới hạn.

Ngoài các phương thức `public`, lớp `Life` cần thêm một hàm phụ trợ **`neighbor_count`** để tính các ô sống kế cận của một ô cho trước.

```
const int maxrow = 20, maxcol = 60; // Kích thước để thử chương trình
class Life {
public:
   void initialize();
   void print();
   void update();
private:
   int grid[maxrow + 2][maxcol + 2];// Dự trữ thêm 4 biên như hình vẽ dưới đây
   int neighbor_count(int row, int col);
};
```



Hình 18.4 – Lưới các ô của `Life` có dự trữ bốn biên

Dưới đây là hàm **`neighbor_count`** được gọi bởi phương thức **`update`**.

```
int Life::neighbor_count(int row, int col)
/*
pre:  Đối tượng Life chứa trạng thái các ô sống, chết. row và col là tọa độ hợp lệ của một ô.
post: Trả về số ô đang sống chung quanh ô tại tọa độ row, col.
*/
{
   int i, j;
   int count = 0;
```

```
   for (i = row - 1; i <= row + 1; i++) // Quét tất cả 9 ô, kể cả tại (row, col)
      for (j = col - 1; j <= col + 1; j++)
         count += grid[i][j];  //  Nếu ô (i,j) sống thì có trị 1 và được cộng vào count
   count -= grid[row][col]; //  Trừ đi bản thân ô đang được xét
   return count;
}
```

Trong phương thức **update** dưới đây chúng ta cần một mảng tạm **new_grid** để lưu trạng thái mới vừa tính được.

```
void Life::update()
/*
pre:  Đối tượng Life đang chứa một trạng thái hiện tại.
post: Đối tượng Life chứa trạng thái mới.
*/
{
   int row, col;
   int new_grid[maxrow + 2][maxcol + 2];

   for (row = 1; row <= maxrow; row++)
      for (col = 1; col <= maxcol; col++)
         switch (neighbor_count(row, col)) {
         case 2:
            new_grid[row][col] = grid[row][col];  // giữ nguyên tình trạng cũ
            break;

         case 3:
            new_grid[row][col] = 1;                 // ô sẽ sống
            break;
         default:
            new_grid[row][col] = 0;                 // ô sẽ chết
         }

   for (row = 1; row <= maxrow; row++)
      for (col = 1; col <= maxcol; col++)
         grid[row][col] = new_grid[row][col];
}
```

Phương thức **initialize** nhận thông tin từ người sử dụng về các ô sống ở trạng thái ban đầu.

```
void Life::initialize()
/*
post: Đối tượng Life đang chứa trạng thái ban đầu mà người sử dụng mong muốn.
*/
{
   int row, col;

   for (row = 0; row <= maxrow+1; row++)
      for (col = 0; col <= maxcol+1; col++)
         grid[row][col] = 0;
   cout << "List the coordinates for living cells." << endl;
   cout << "Terminate the list with the special pair -1 -1" << endl;
   cin >> row >> col;
```

```
   while (row != -1 || col != -1) {
      if (row >= 1 && row <= maxrow)
         if (col >= 1 && col <= maxcol)
            grid[row][col] = 1;
         else
            cout << "Column " << col << " is out of range." << endl;
      else
         cout << "Row " << row << " is out of range." << endl;
      cin >> row >> col;
   }
}
```

```
void Life::print()
/*
pre:  Đối tượng Life đang chứa một trạng thái.
post: Các ô sống được in cho người sử dụng xem.
*/

{
   int row, col;
   cout << "\nThe current Life configuration is:" <<endl;
   for (row = 1; row <= maxrow; row++) {
      for (col = 1; col <= maxcol; col++)
         if (grid[row][col] == 1) cout << '*';
         else cout << ' ';
      cout << endl;
   }
   cout << endl;
}
```

Các hàm phụ trợ

Các hàm phụ trợ dưới đây có thể xem là khuôn mẫu và có thể được sửa đổi đôi chút để dùng cho các ứng dụng khác.

```
void instructions()
/*
post: In hướng dẫn sử dụng chương trình Game_Of_Life.
*/

{
   cout << "Welcome to Conway's game of Life." << endl;
   cout << "This game uses a grid of size "
        << maxrow << " by " << maxcol << " in which" << endl;
   cout << "each cell can either be occupied by an organism or not." << endl;
   cout << "The occupied cells change from generation to generation" << endl;
   cout << "according to the number of neighboring cells which are alive."
        << endl;
}
```

```
bool user_says_yes()
{
   int c;
   bool initial_response = true;
```

```
   do {  // Lặp cho đến khi người sử dụng gõ một ký tự hợp lệ.
      if (initial_response)
         cout << " (y,n)? " << flush;

      else
         cout << "Respond with either y or n: " << flush;

      do { //  Bỏ qua các khoảng trắng.
         c = cin.get();
      } while (c == '\n' || c ==' ' || c == '\t');
      initial_response = false;
   } while (c != 'y' && c != 'Y' && c != 'n' && c != 'N');
   return (c == 'y' || c == 'Y');
}
```

18.4.2. Phiên bản thứ hai với CTDL mới cho Life

Phiên bản trên giải quyết được bài toán Game_Of_Life nhưng với hạn chế là lưới các ô có kích thước giới hạn. Yêu cầu của bài toán là tấm lưới chứa các ô của Life là không có giới hạn. Chúng ta có thể khai báo lớp Life chứa một mảng thật lớn như sau:

```
class Life {
public:
   //  Các phương thức.
private:
   bool map[int][int];
   //  Các thuộc tính khác và các hàm phụ trợ.
};
```

nhưng cho dù nó có lớn mấy đi nữa thì cũng vẫn có giới hạn, đồng thời các giải thuật phải quét hết tất cả các ô trong lưới là hoàn toàn phí phạm. Điều không hợp lý ở đây là tại mỗi thời điểm chỉ có một số giới hạn các ô của Life là sống, tốt hơn hết chúng ta nên nhìn các ô sống này như là một ma trận thưa. Và chúng ta sẽ dùng các cấu trúc liên kết thích hợp.

18.4.2.1. Lựa chọn giải thuật

Chúng ta sẽ thấy, các công việc cần xử lý trên dữ liệu góp phần quyết định cấu trúc của dữ liệu.

Khi cần biết trạng thái của một ô đang sống hay chết, nếu chúng ta dùng phương pháp tra cứu của bảng băm thì giải thuật hiệu quả hơn rất nhiều: nếu ô có trong bảng thì có nghĩa là nó đang sống, ngược lại là nó đang chết. Việc duyệt danh sách để xác nhận sự có mặt của một phần tử hay không không hiệu quả bằng phương pháp băm như chúng ta đã biết. Đối với bất kỳ một ô nào có trong

cấu hình, chúng ta có thể xác định số ô sống chung quanh nó bằng cách tra cứu trạng thái của chúng.

Trong hiện thực mới của chúng ta cho phương thức **`update`**, chúng ta sẽ duyệt qua tất cả các ô có khả năng thay đổi trạng thái, xác định số ô sống chung quanh mỗi ô nhờ sử dụng bảng, và chọn ra những ô sẽ thực sự sống trong trạng thái kế.

18.4.2.2. Đặc tả cấu trúc dữ liệu

Tuy rằng bảng băm chứa tất cả các ô đang sống, nhưng nó chỉ tiện trong việc tra cứu trạng thái của từng ô mà thôi. Chúng ta cũng sẽ cần duyệt qua các ô sống trong cấu hình đó. Việc duyệt một bảng băm thường không hiệu quả. Do đó, ngoài bảng băm, chúng ta cần một danh sách các ô sống như là thành phần dữ liệu thứ hai của một cấu hình `Life`. Các đối tượng được lưu trong danh sách và bảng băm của cấu hình `Life` cùng chứa thông tin về các ô sống, nhưng chúng ta có hai cách truy cập khác nhau. Điều này phục vụ đắc lực cho giải thuật của bài toán như đã phân tích ở trên. Chúng ta sẽ biểu diễn các ô bằng các thể hiện của một cấu trúc gọi là **`Cell`**: mỗi ô cần một cặp tọa độ.

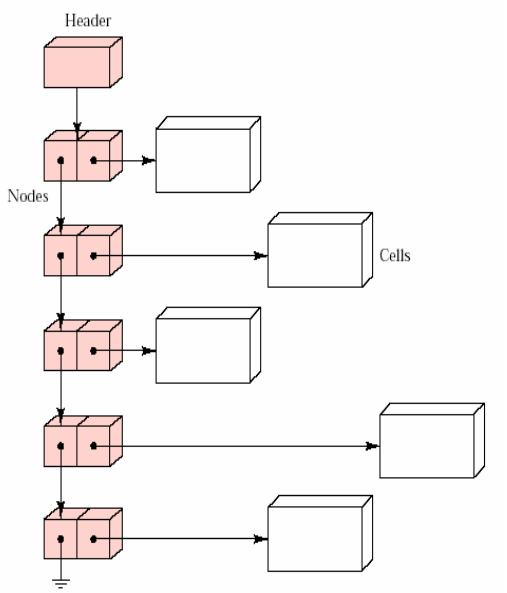
```
struct Cell {
   Cell() { row = col = 0; }   //  Các constructor
   Cell(int x, int y) { row = x;  col = y; }
   int row, col;
}
```

Khi cấu hình `Life` nới rộng, các ô ở bìa ngoài của nó sẽ xuất hiện dần dần. Như vậy một ô mới sẽ xuất hiện nhờ vào việc cấp phát động vùng nhớ, và nó sẽ chỉ được truy xuất đến thông qua con trỏ. Chúng ta sẽ dùng một `List` mà mỗi phần tử chứa con trỏ đến một ô (hình 18.5). Mỗi phần tử của `List` gồm hai con trỏ: một chỉ đến một ô đang sống và một chỉ đến phần tử kế trong `List`.

Cho trước một con trỏ chỉ một ô đang sống, chúng ta có thể xác định các tọa độ của ô đó bằng cách lần theo con trỏ rồi lấy hai thành phần `row` và `col` của nó. Như vậy, chúng ta có thể lưu các con trỏ chỉ đến các ô như là các bản ghi trong bảng băm; các toạ độ `row` và `col` của các ô, được xác định bởi con trỏ, sẽ là các khóa tương ứng.

Chúng ta cần lựa chọn giữa bảng băm địa chỉ mở và bảng băm nối kết. Các phần tử sẽ chứa trong bảng băm chỉ có kích thước nhỏ: mỗi phần tử chỉ cần chứa một con trỏ đến một ô đang sống. Như vậy, với bảng băm nối kết, kích thước của mỗi bản ghi sẽ tăng 100% do phải chứa thêm các con trỏ liên kết trong các danh sách liên kết. Tuy nhiên, bản thân bảng băm nối kết sẽ có kích thước rất nhỏ mà vẫn có thể chứa số bản ghi lớn gấp nhiều lần kích thước chính nó. Với bảng băm

địa chỉ mở, các bản ghi sẽ nhỏ hơn vì chỉ chứa địa chỉ các ô đang sống, nhưng cần phải dự trữ nhiều vị trí trống để tránh hiện tượng tràn xảy ra và để quá trình tìm kiếm không bị kéo dài quá lâu khi đụng độ thường xuyên xảy ra.



Hình 18.5 – Danh sách liên kết gián tiếp.

Để tăng tính linh hoạt, chúng ta quyết định sẽ dùng bảng băm nối kết có định nghĩa như sau:

```
class Hash_table {
public:
   Error_code insert(Cell *new_entry);
   bool retrieve(int row, int col) const;
private:
   List<Cell *> table[hash_size]; // Dùng danh sách liên kết.
};
```

Ở đây, chúng ta chỉ đặc tả hai phương thức: **`insert`** và **`retrieve`**. Việc truy xuất bảng là để biết bảng có chứa con trỏ chỉ đến một ô có tọa độ cho trước hay không. Do đó phương thức **`retrieve`** cần hai thông số chứa tọa độ `row` và `col` và trả về một trị `bool`. Chúng ta dành việc hiện thực hai phương thức này như là bài tập vì chúng rất tương tự với những gì chúng ta đã thảo luận về bảng băm nối kết trong chương 12.

Chúng ta lưu ý rằng `Hash_table` cần có những phương thức *constructor* và *destructor* của nó. Chẳng hạn, *destructor* của `Hash_table` cần gọi *destructor* của `List` cho từng phần tử của mảng **`table`**.

18.4.2.3. Lớp Life

Với các quyết định trên, chúng ta sẽ gút lại cách biểu diễn và những điều cần lưu ý cho lớp `Life`. Để cho việc thay đổi cấu hình được dễ dàng chúng ta sẽ lưu các thành phần dữ liệu một cách gián tiếp qua các con trỏ. Như vậy lớp `Life` cần có *constructor* và *destructor* để định vị cũng như giải phóng các vùng nhớ cấp phát động cho các cấu trúc này.

```
class Life {
public:
   Life();
   void initialize();
   void print();
   void update();
   ~Life();

private:
   List<Cell *> *living;
   Hash_table *is_living;
   bool retrieve(int row, int col) const;
   Error_code insert(int row, int col);
   int neighbor_count(int row, int col) const;
};
```

Các hàm phụ trợ **`retrieve`** và **`neighbor_count`** xác định trạng thái của một ô bằng cách truy xuất bảng băm. Hàm phụ trợ khác, **`insert`**, khởi tạo một đối tượng **`Cell`** cấp phát động và chèn nó vào bảng băm cũng như danh sách các ô trong đối tượng **`Life`**.

18.4.2.4. Các phương thức của Life

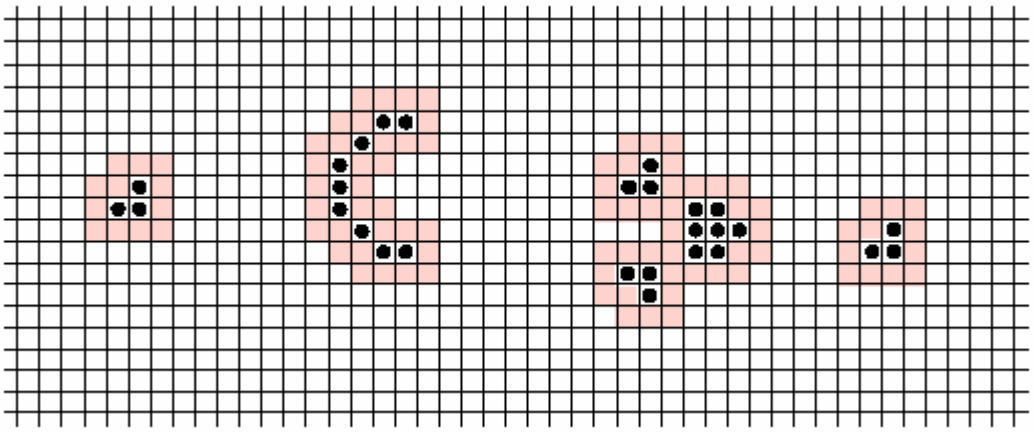
Chúng ta sẽ viết một vài phương thức và hàm của `Life` để minh họa cách xử lý các ô, các danh sách và những gì diễn ra trong bảng băm. Các hàm còn lại xem như bài tập.

Cập nhật cấu hình

Phương thức **`update`** có nhiệm vụ xác định cấu hình kế tiếp của `Life` từ một cấu hình cho trước. Trong phiên bản trước, chúng ta làm điều này bằng cách xét mọi ô có trong lưới chứa cấu hình **`grid`**, tính các ô kế cận chung quanh cho mỗi ô để xác định trạng thái kế tiếp của nó. Các thông tin này được chứa trong biến cục bộ **`new_grid`** và sau đó được chép vào **`grid`**.

Chúng ta sẽ lặp lại những công việc này ngoại trừ việc phải xét mọi ô có thể có trong cấu hình do đây là một lưới không có giới hạn. Thay vào đó, chúng ta nên giới hạn tầm nhìn của chúng ta chỉ trong các ô có khả năng sẽ sống trong trạng thái kế. Đó có thể là các ô nào? Rõ ràng đó chính là các ô đang sống trong trạng thái hiện tại, chúng có thể chết đi nhưng cũng có thể tiếp tục sống trong

trạng thái kế. Ngoài ra, một số ô đang chết cũng có thể trở nên sống trong trạng thái sau, nhưng đó chỉ là những ô đang chết nằm kề những ô đang sống (các ô có màu xám trong hình 9.18). Những ô xa hơn nữa không có khả năng sống dậy do chúng không có ô nào kế cận đang sống.



Hình 18.6 – Cấu hình Life với các ô chết viền chung quanh.

Trong phương thức **update**, biến cục bộ `Life` **new_configuration** dùng để chứa cấu hình mới: chúng ta thực hiện vòng lặp cho tất cả những ô đang sống và những ô kế cận của chúng, đối với mỗi ô như vậy, trước hết cần xác định xem nó đã được thêm vào **new_configuration** hay chưa, chúng ta cần phải cẩn thận để tránh việc thêm bị lặp lại hai lần cho một ô. Nếu quả thật ô đang xét chưa có trong **new_configuration**, chúng ta dùng hàm **neighbor_count** để quyết định việc có thêm nó vào cấu hình mới hay không.

Ở cuối phương thức, chúng ta cần hoán đổi các thành phần **List** và **Hash_table** giữa cấu hình hiện tại và cấu hình mới **new_configuration**. Sự hoán đổi này làm cho đối tượng **Life** có được cấu hình đã được cập nhật, ngoài ra, nó còn bảo đảm rằng các ô, danh sách, và bảng băm trong cấu hình cũ của đối tượng **Life** sẽ được giải phóng khỏi bộ nhớ cấp phát động (việc này được thực hiện do trình biên dịch tự động gọi *destructor* cho đối tượng cục bộ **new_configuration**).

```
void Life::update()
/*
post: Đối tượng Life chứa cấu hình ở trạng thái kế.
uses: Lớp Hash_table và lớp Life và các hàm phụ trợ.
*/
{
   Life new_configuration;
   Cell *old_cell;
```

```
    for (int i = 0; i < living->size(); i++) {
      living->retrieve(i, old_cell);      //  Lấy một ô đang sống.
      for (int row_add = -1; row_add < 2; row_add ++)
         for (int col_add = -1; col_add < 2; col_add++) {
            int new_row = old_cell->row + row_add,
                new_col = old_cell->col + col_add;

//  new_row, new_col là toạ độ của ô đang xét hoặc ô kế cận của nó.
            if (!new_configuration.retrieve(new_row, new_col))
               switch (neighbor_count(new_row, new_col)) {
               case 3:  //  Số ô sống kế cận là 3 thì ô đang xét trở thành sống.
                  new_configuration.insert(new_row, new_col);
                  break;

               case 2:  //  Số ô sống kế cận là 2 thì ô đang xét giữ nguyên trạng thái.
                  if (retrieve(new_row, new_col))
                     new_configuration.insert(new_row, new_col);
                  break;

               default: //  Ô sẽ chết.
                  break;
               }
         }
   }
// Tráo dữ liệu trong configuration với dữ liệu trong new_configuration.
                  new_configuration sẽ được dọn dẹp bởi destructor của nó.
   List<Cell *> *temp_list = living;
   living = new_configuration.living;
   new_configuration.living = temp_list;
   Hash_table *temp_hash = is_living;
   is_living = new_configuration.is_living;
   new_configuration.is_living = temp_hash;
}
```

In cấu hình

Để in được tất cả các ô đang sống, chúng ta có thể liệt kê lần lượt mỗi dòng một ô với tọa độ `row, col` của nó. Ngược lại nếu muốn biểu diễn đúng vị trí `row, col` trên lưới, chúng ta nhận thấy rằng chúng ta không thể hiển thị nhiều hơn là một mẩu nhỏ của một cấu hình `Life` không có giới hạn lên màn hình. Do đó chúng ta sẽ in một cửa sổ chữ nhật để hiển thị trạng thái của 20 x 80 các vị trí ở trung tâm của cấu hình `Life`.

Đối với mỗi ô trong cửa sổ, chúng ta truy xuất trạng thái của nó từ bảng băm và in một ký tự trắng hoặc khác trắng tương ứng với trạng thái chết hoặc sống của nó.

```
void Life::print()
/*
post: In một trạng thái của đối tượng Life.
uses: Life::retrieve.
*/
```

```
{
   int row, col;
   cout << endl << "The current Life configuration is:" << endl;

   for (row = 0; row < 20; row++) {
      for (col = 0; col < 80; col++)
         if (retrieve(row, col)) cout << '*';
         else cout << ' ';
      cout << endl;
   }
   cout << endl;
}
```

Tạo và thêm các ô mới

Phương thức **`insert`** tạo một ô mới với tọa độ cho trước và thêm nó vào bảng băm **`is_living`** và vào danh sách **`living`**.

```
ErrorCode Life::insert(int row, int col)
/*
pre:  Ô có tọa độ (row,col) không thuộc về đối tượng Life (ô đang chết)
post: Ô có tọa độ (row,col) được bổ sung vào đối tượng Life (ô trở nên sống). Nếu việc
      thêm vào List hoặc Hash_table không thành công thì lỗi sẽ được trả về.
uses: Các lớp List, Hash_table, và cấu trúc Cell.
*/
{
   Error_code outcome;
   Cell *new_cell = new Cell(row, col);

   int index = living->size();
   outcome = living->insert(index, new_cell);
   if (outcome == success)
      outcome = is_living->insert(new_cell);
   if (outcome != success)
      cout << " Warning: new Cell insertion failed" << endl;
   return outcome;
}
```

Constructor và *destructor* cho các đối tượng `Life`

Chúng ta cần cung cấp *constructor* và *destructor* cho lớp `Life` của chúng ta để định vị và giải phóng các thành phần cấp phát động của nó. *Constructor* cần thực hiện toán tử new cho các thuộc tính con trỏ.

```
Life::Life()
/*
post: Các thuộc tính thành phần của đối tượng Life được cấp phát động và được khởi tạo.
uses: Các lớp Hash_table, List.
*/
{
   living = new List<Cell *>;
   is_living = new Hash_table;
}
```

Destructor cần giải phóng mọi phần tử được cấp phát động bởi bất kỳ một phương thức nào đó của lớp `Life`. Bên cạnh hai con trỏ **`living`** và **`is_living`** được khởi tạo nhờ *constructor* còn có các đối tượng **`Cell`** mà chúng tham chiếu đến đã được cấp phát động trong phương thức **`insert`**. *Destructor* cần giải phóng các đối tượng **`Cell`** này trước khi giải phóng **`living`** và **`is_living`**.

```
Life::~Life()
/*
post: Các thuộc tính cấp phát động của đối tượng Life và các đối tượng do chúng tham chiếu
      được giải phóng.
uses: Các lớp Hash_table, List.
*/
{
   Cell *old_cell;
   for (int i = 0; i < living->size(); i++) {
      living->retrieve(i, old_cell);
      delete old_cell;
   }
   delete is_living;          //  Gọi destructor của Hash_table.
   delete living;             //  Gọi destructor của List.
}
```

Đối với những cấu trúc có nhiều liên kết bằng con trỏ như thế này, chúng ta luôn phải đặt vấn đề về khả năng tạo rác do những sơ suất của chúng ta. Thông thường thì *destructor* của một lớp luôn làm tốt nhiệm vụ của nó, nhưng nó chỉ dọn dẹp những gì thuộc đối tượng của nó, chứ không biết đến những gì mà đối tượng của nó tham chiếu đến. Nếu chúng ta chủ quan, việc dọn dẹp không diễn ra đúng như chúng ta tưởng. Khi chạy, chương trình có thể là quên dọn dẹp, cũng có thể là dọn dẹp nhiều hơn một lần đối với một vùng nhớ được cấp phát động. Đây cũng là một vấn đề khá lý thú mà sinh viên nên tự suy nghĩ thêm.

Hàm băm

Hàm băm ở đây có hơi khác với những gì chúng ta đã gặp trong những phần trước, thông số của nó có đến hai thành phần (`row` và `col`), nhờ vậy chúng ta có thể dễ dàng sử dụng một vài dạng của phép trộn. Trước khi quyết định nên làm như thế nào, chúng ta hãy xét trường hợp một mảng chữ nhật nhỏ có ánh xạ một một dưới đây chính là một hàm chỉ số. Có chính xác là **`maxrow`** phần tử trong mỗi hàng, các chỉ số i, j được ánh xạ đến `i + maxrow*j` để đặt mảng chữ nhật vào một chuỗi các vùng nhớ liên tục, hàng này kế tiếp hàng kia.

Chúng ta nên dùng cách ánh xạ tương tự cho hàm băm của chúng ta và sẽ thay thế `maxrow` bằng một số thích hợp, chẳng hạn như một số nguyên tố, để việc phân phối được rải đều và giảm sự đụng độ.

```
const int factor = 101;

int hash(int row, int col)
/*
post: Trả về giá trị hàm băm từ 0 đến hash_size - 1 tương ứng với tọa độ (row, col).
*/
{
   int value;
   value = row + factor * col;
   value %= hash_size;
   if (value < 0) return value + hash_size;
   else return value;
}
```

Các chương trình con khác

Các phương thức còn lại cuûa `Life` như `initialize`, `retrieve`, và `neighbor_count` đều được xem xét tương tự như các hàm vừa rồi hoặc như các hàm tương ứng trong phiên bản thứ nhất. Chúng ta dành chúng lại như bài tập.