**Chương 7**

**Sự đánh đổi giữa bộ nhớ và thời gian**

Việc đánh đổi giữa bộ nhớ và thời gian trong thiết kế thuật toán là một vấn đề thiết thực cho cả lý thuyết và thực hành trong máy tính. Xem xét một ví dụ, tính giá trị của một hàm tại nhiều điểm trong miền xác định của nó. Nếu thời gian là vấn đề cần quan tâm, có thể tính trước giá trị của hàm và lưu trữ chúng trong một bảng. Đây là những điều con người phải làm trước khi dùng đến máy tính điện tử, việc làm như thế, nó tăng hiệu quả về mặt thời gian để giải một số bài toán. Chúng ta gọi phương pháp này là tăng cường đầu vào (input enhancement), và xem xét các thuật toán sau đây dựa trên ý tưởng tăng cường đầu vào.

- Phương pháp đếm để sắp xếp   
- Giải thuật Boyer –Moore, một giải thuật đơn giản để so khớp chuỗi do Horspool đề xuất

Những kỹ thuật khác khai thác về vấn đề đánh đổi giữa bộ nhớ và thời gian thì đơn giản là sử dụng bộ nhớ-thêm (extra space) nhằm truy cập nhanh và linh hoạt hơn. Kỹ thuật này gọi là tiền cấu trúc (prestructing), nó được minh họa bằng cách:

* Băm
* Lập chỉ mục với B-cây

Một giải thuật nữa liên quan đến thời gian và bộ nhớ là lập trình động (dynamic programming), sẽ được trình bày trong chương sau.

Hai ý kiến ​​cuối cùng về sự tương tác giữa thời gian và bộ nhớ trong thiết kế thuật toán cần phải được thực hiện. Thứ nhất, hai nguồn tài nguyên thời gian và bộ nhớ không phải cạnh tranh với nhau trong tất cả các tình huống thiết kế. Trong thực tế, chúng có thể sắp xếp để mang lại một giải pháp giảm thiểu cả thời gian thực hiện và bộ nhớ cần sử dụng. Xem xét , một ví dụ, bài toán người du lịch giải bằng đồ thị, thời gian thực hiện cho bài toán này bằng tìm kiếm theo chiều sâu và tìm kiếm theo chiều rộng thì tùy thuộc vào cấu trúc dữ liệu được dùng để biểu diễn đồ thị. Nếu dùng ma trận kề thì độ phức tạp thời gian là *O*(*n*2), dùng danh sách liền kề là *O*(*n*+*m*) với *n* và *m* là số đỉnh và số cạnh tương ứng. Như vậy sử dụng cấu trúc dữ liệu ảnh hưởng đến thời gian của thuật toán. Thứ hai, không thể thảo luận về đánh đổi giữa bộ nhớ và thời gian mà không đề cập đến vấn đề nén dữ liệu. Tuy nhiên, trong quá trình nén dữ liệu, giảm kích thước là mục tiêu chứ không phải là một kỹ thuật để giải quyết một vấn đề khác. Vấn đề này sẽ thảo luận trong chương sau, chỉ là một thuật toán nén dữ liệu,.

**7.1 Sắp xếp bằng cách đếm**

Phần này tìm hiểu hai thuật giải Horspool và Boyer–Moore, để so khớp một chuỗi với một văn bản, sử dụng kỹ thuật tăng cường đầu vào.

**7.1.1** **Sắp xếp bằng cách đếm khi so sánh**

Ví dụ đầu tiên áp dụng kỹ thuật tăng cường đầu vào, là thảo luận về ứng dụng của kỹ thuật này đối với bài toán sắp xếp. Ý tưởng khá rõ ràng là đếm, với mỗi phần tử của mảng được sắp xếp, đếm số các phần tử trong mảng mà nhỏ hơn so với phần tử này, ghi lại kết quả vào một bảng. Những con số này sẽ chỉ ra vị trí của các phần tử trong mảng được sắp xếp: ví dụ, nếu một phần tử nào đó đếm được là 10, thì nó phải ở vị trí thứ 11 (với chỉ số 10, nếu chúng ta bắt đầu đếm với 0) trong mảng được sắp xếp. Do đó, có thể sắp xếp mảng bằng cách sao chép các phần tử của của mảng cũ vào các vị trí thích hợp của chúng trong một mảng mới. Thuật toán này được gọi là sắp xếp bằng cách đếm khi so sánh (comparision-counting sort) như sau:

**THUẬT TOÁN** *ComparisonCountingSort* (A[0..n−1])

//sắp xếp một mảng bằng cách đếm

//đầu vào: mảng A[0..n−1] cần xếp thứ tự

//đầu ra: danh sách S[0..n−1] gồm những phần tử trong A đã được xếp thứ tự tăng

**for** i ← 0 **to** n−1 **do** count[i] ← 0

**for** i ← 0 **to** n−2 **do**

**for** j ← i+1 **to** n−1 **do**

**if** A[i] < A[j]

count[j] ← count[j]+1

**else** count[i] ← count[i]+1

**for** i ← 0 **to** n−1 **do** S[count[i]] ← A[i]

**return** S

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Mảng *A*[0..5] |  | **62** | **31** | **84** | **96** | **19** | **47** |
|  |  |  |  |  |  |  |  |
| Khởi đầu | count [ ] | 0 | 0 | 0 | 0 | 0 | 0 |
| sau khi *i* = 0 | count [ ] | 3 | 0 | 1 | 1 | 0 | 0 |
| sau khi *i* = 1 | count [ ] |  | 1 | 2 | 2 | 0 | 1 |
| sau khi *i* = 2 | count [ ] |  |  | 4 | 3 | 0 | 1 |
| sau khi *i* = 3 | count [ ] |  |  |  | 5 | 0 | 1 |
| sau khi *i* = 4 | count [ ] |  |  |  |  | 0 | 2 |
| Trạng thái cuối | count [ ] | 3 | 1 | 4 | 5 | 0 | 2 |
|  |  |  |  |  |  |  |  |
| Mảng *S*[0..5] |  | **19** | **31** | **47** | **62** | **84** | **96** |

**Hình 7.1:** Một ví dụ sắp xếp bằng cách đếm

Trong hình 7.1, mô tả quá trình thực hiện thuật toán *ComparisonCountingSort*, nó sắp xếp mảng *A*[0..5] thành mảng *S*[0..5] có thứ tự. Mỗi phần tử của *A* được so sánh với mọi phần tử còn lại trong *A*, với mỗi lần so sánh thì giá trị phần tử thứ *i* hay thứ *j* trong danh sách *Count*, được tăng lên 1 (khởi đầu thuật toán, giá trị của mọi phần tử trong *Count* được gán giá trị 0). Cuối thuật toán, giá trị của các phần tử trong *Count*[*i*] (dòng: Trạng thái cuối trong hình) chỉ ra vị trí của phần tử *A*[*i*] trong danh sách *S*.

Độ phức tạp theo thời gian *O*(*n*) của thuật toán này là gì? Nó phải là bậc hai bởi vì các thuật toán xem xét tất cả các cặp khác nhau của một mảng *n* phần tử. Theo thuật toán, số lần hoạt động chính của nó, là so sánh *A*[*i*] với *A*[*j*], được đặt trong hai vòng lặp for lồng nhau, vòng for ngoài lặp theo biến *i* từ 0 đến *n*-2 lần. Với mỗi giá trị của *i* thì vòng for trong lặp từ *i*+1 đến *n*-1 lần. Do vậy, *O*(*n*) được tính:

*O*(*n*) =

Thuật toán đã thực hiện một số phép so sánh để lựa chọn sắp xếp và sử dụng một lượng tuyến tính bộ nhớ-thêm. Về mặt hiệu quả, thuật toán làm tối thiểu số lần phải di chuyển những phần tử của mảng trong quá trình sắp xếp để đặt chúng vào đúng vị trí cần sắp xếp.

**7.1.2 Sắp xếp bằng cách đếm phân phối**

Thuật toán sắp xếp bằng cách đếm phân phối được sử dụng trong trường hợp đặc biệt, khi mà mảng *A*[0..*n*] cần sắp xếp bao gồm những phần tử là những số nguyên có giá trị nằm giữa số nguyên *l* và số nguyên *u*. Khi đó, ta có một tập *V* gồm những giá trị khác nhau của *A*, phần tử đầu của *A* là *l* và phần tử cuối là *u*. Tập *F* ghi số lần xuất hiện (tần xuất) của từng phần tử của *V* trong *A*, *F* chính là tập các biến đếm, dựa vào các biến đếm, ta có thể biết một phần tử trong *A* sẽ xếp từ vị trí nào đến vị trí nào trong mảng *S* đã được sắp xếp. Xét ví dụ sau:

**Ví dụ:** Sắp xếpmảng*A*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 13 | 11 | 12 | 13 | 12 | 12 |

Ta có: *V* = { 11, 12, 13},

*F* = { 1, 4, 6 }.

Mảng *A* được sắp xếp thành mảng *S* như sau:

Phần tử trong *F* là 1, vì số 11 xuất hiện một lần trong *A*, số 11 được xếp ở vị trí 0 trong *S.*

Phần tử trong *F* là 3, vì số 12 xuất hiện ba lần trong *A*, số 12 được xếp ba vị trí liên tiếp 1, 2 và 3 trong *S*, tương tự số 13 được xếp hai vị trí liên tiếp 4 và 5 trong *S*.

Thuật toán sắp xếp bằng cách đếm phân phối *DistributionCountingSort* như sau:

**THUẬT TOÁN** *DistributionCountingSort* (*A*[0..*n*−1], *l*, *u*)

// sắp xếp mảng số nguyên hữu hạn bằng đếm phân phối

// đầu vào: Mảng *A*[0..*n* −1] gồm những số nguyên giữa *l* và *u* (*l* ≤ *u*)

// đầu ra: Mảng *S*[0..*n* −1] gồm những phần tử trong *A* đã được sắp xếp

**for** *j* ←0 **to** *u* − *l* **do** *F*[*j*] ←0 // khởi động tập *F*

**for** *i* ←0 **to** *n*−1 **do** *F*[A[*i*] − *l*] ← *F*[*A*[*i*] − *l*] + 1 // đếm số lần xuất hiện

**for** *j* ←1 **to** *u* − *l* **do** *F*[*j*] ← *F*[*j*−1] + *F*[*j*] // sử dụng biến đếm trong *F*

**for** *i* ← *n*−1 **downto** 0 **do**

*j* ← *A*[*i*]−*l*

*S*[*F*[*j*] − 1] ← *A*[*i*]

*F*[*j*] ← *F*[*j*] − 1

**return** *S*

Mảng *A* trong ví dụ này sau khi thực hiện ba vòng for đầu trong thuật toán, thì đếm được số lần xuất hiện của ba phần tử 11, 12 và 13 trong *A* lần lượt là 1, 4 và 6 và những giá trị đếm được, được ghi vào trong tập *F*. Vòng for cuối cùng cho kết quả đã xắp xếp trong mảng *S*. Quá trình thực hiện này như hình 7.2.

*F* [0..2] *S* [0..5]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A[5] = 12 | 1 | **4** | 6 |  |  |  |  | 12 |  |  |
| A[4] = 12 | 1 | 3 | 6 |  |  |  | 12 |  |  |  |
| A[3] = 13 | 1 | 2 | **6** |  |  |  |  |  |  | 13 |
| A[2] = 12 | 1 | **2** | 5 |  |  | 12 |  |  |  |  |
| A[1] = 11 | **1** | 1 | 5 |  | 11 |  |  |  |  |  |
| A[0] = 13 | 0 | 1 | **5** |  |  |  |  |  | 13 |  |

**Hình 7.2:** Sắpxếp bằng đếm phân phối, giá trị phân phối giảm được in đậm

Dễ thấy thuật toán đếm phân phối có độ phức tạp về thời gian là tuyến tính, nhưng đây là trường hợp đặc biệt vì mảng cần sắp xếp biết trước giá trị *l*, *u* và tập *V*.

**Bài tập 7.1**

1. Có thể hoán chuyển giá trị số của hai biến *u* và *v* cho nhau mà không cần dùng bất kỳ một lưu trữ nào không? (có nghĩa là trong bộ nhới chỉ có biến *u* và *v*, ngoài ra không có khai báo thêm bất kỳ một biến nào khác). Nếu có, nêu cách thực hiện.
2. Thuật toán *ComparisionCountingSort* có thể thực hiện trong trường hợp mảng có các phần tử có cùng giá trị không?
3. Lần vết thuật toán *ComparisionCountingSort* với mảng đầu vào: *A*[*b*, *c*, *d*, *c*, *b*, *a*, *a*, *b*], cho kết quả lần vết như hình 7.2
4. Thiết kế thuật toán chỉ có một dòng for để sắp xếp một mảng *A* gồm những số nguyên liên tiếp khác nhau từ 1 đến *n*, vào một mảng S. Ví dụ: *n* =5, thì *A* có thể là: *A*[4, 3, 1, 5, 2] và *S*[1, 2, 3, 4, 5].
5. Bài toán sắp xếp các băng màu đỏ, trắng và xanh thành lá cờ của nước Hà lan (lá cờ Hà lan có 3 giải màu, trên cùng là màu đỏ, giữa là màu trắng và cuối cùng là màu xanh), được gọi là bài toán lá cờ Hà lan. Cho trước một mảng *A* gồm *n* giải màu, gồm có các màu: trắng, đỏ và xanh. Trong đó các màu được mã hóa thành các ký tự R, W và B tương ứng với màu đỏ, trắng và xanh, nghĩa là mảng *A* có *n* ký tự bao gồm các ký tự R, W và B. Thiết kế thuật toán có độ phức tạp tuyến tính để sắp xếp *A* sao cho những ký tự trong A có thự tự như: những ký tự R đứng trước những ký tự W và sau cùng là những ký tự B.

**7.2 Tăng cường đầu vào trong so khớp chuỗi**

 Bộ nhớ lưu trữ có thể tiết kiệm thời gian tính toán. Bảng có thể có chức năng đánh giá chung để giảm các cuộc gọi đệ quy. Kỹ thuật khác là lưu trữ các tính toán được xử trước rồi ghi vào một mảng được sử dụng trong các tính toán cuối cùng, điều này được gọi là tăng cường đầu vào. Xem xét một thuật toán nổi tiếng về so khớp chuỗi, dựa trên ý tưởng tăng cường đầu vào là thuật toán Boyer-Moore. Trước tiên, tìm hiểu thuật toán Horspool là một thuật toán đơn giản hơn.

Nhớ lại bài toán so khớp một chuỗi với một văn bản. Xét mẫu *P*[0 ... *m*-1] có *m* kí tự được tìm kiếm trong một văn bản *T* [0 ... *n*-1] có *n* kí tự. Các thuật toán brute force trong trường hợp xấu nhất cần *m* (*n*-*m* +1) lần so sánh, vì vậy chi phí là *O*(*n*×*m*). Nhưng trung bình chỉ là một vài so sánh được thực hiện trước khi dịch chuyển mẫu *P*, vì vậy chi phí là *O*(*n*). Với một văn bản bằng ngôn ngữ tự nhiên, thì chi phí trung bình cho việc so khớp mẫu là *O*(*n*+*m*). Bây giờ hãy xem xét những thuật toán cũng đạt được chi phí này.

**7.2.1 Thuật toán Horspool**

Coi ví dụ, tìm kiếm mẫu BARBER trong một văn bản *T*:

*T*[0] … *c* … *T*[*n*-1]

BARBER

Bắt đầu với ký tự R cuối cùng của mẫu , dịch chuyển mẫu từ phải sang trái và so sánh các cặp ký tự tương ứng giữa mẫu và văn bản. Nếu tất cả các ký tự của mẫu trùng với văn bản, thì hiển nhiên mẫu đã tìm thấy trong văn bản. Sau đó tìm kiếm có thể dừng lại hay tiếp tục, nếu muốn tìm những chuỗi khác trong văn bản trùng với mẫu. Nếu không có sự trùng khớp kí tự của mẫu với ký tự trong văn bản, phải dịch chuyển mẫu sang bên phải. Rõ ràng, mong muốn thực hiện một dịch chuyển lớn nhất có thể mà không sợ bị bỏ qua một chuỗi của văn bản phù hợp với mẫu. Bằng cách nhìn vào các ký tự trong văn bản, thuật toán Horspool xác định cần dịch chuyển mẫu bao nhiêu vị trí sang phải để ký tự cuối cùng của mẫu trùng với ký tự trong văn bản.

  Coi *c* là ký tự trong văn bản, cần so sánh *c* với một ký tự của mẫu khi mẫu đang được so khớp với văn bản (mẫu đang được đặt sát dưới từng dòng của văn bản, đồng thời dịch chuyển dọc theo văn bản từ trái sang phải để so khớp) có có bốn trường hợp để xem xét:

**Trường hợp 1**

*c* không tồn tại trong mẫu, thì ta dịch chuyển toàn bộ mẫu sang bên phải của ký tự *c*. Trong hình minh họa sau thì *c* là ký tự *S*.

*T*[0] ... S ... *T*[*n*-1]

|

BARBER

BARBER

**Trường hợp 2**

Ký tự cuối cùng của mẫu không trùng với *c* là ký tự A trong ví dụ, thì chuyển dịch mẫu sang bên phải nhiều nhất là *m*-1 lần (ví dụ sau dịch chuyển 3 lần), để tìm có ký tự nào trong mẫu trùng hợp với *c* hay không.

*T*[0] ... A ... *T*[*n*-1]

|

LEADER

LEADER

**Trường hợp 3**

Nếu *c* là R, ký tự cuối cùng trong mẫu cũng là R, nhưng không có R trong *m*-1 ký tự còn lại của mẫu, tình huống này tương tự như trường hợp 1 và mẫu được chuyển dịch sang phải *m* lần:

*T*[0] ... MER ... *T*[*n*-1]

|

LEADER

LEADER

**Trường hợp 4**

Cuối cùng, nếu *c* là R trùng với ký tự cuối cùng trong mẫu và còn có một *c* khác trong *m*-1 ký tự còn lại của mẫu, tình hình cũng tương tự như trong trường hợp 2, khi đó cần dịch chuyển mẫu sang phải một số lần để gióng hàng *c* của văn bản với ký tự bắt gặp đầu tiên trong mẫu trùng với *c*.

*T*[0] ... AR ... *T*[*n*-1]

 |

REORDER

REORDER

Những ví dụ trên cho thấy việc so sánh ký tự từ phải sang trái có thể dẫn đến mẫu bị dịch chuyển nhiều hơn một vị trí. Trong khi chỉ được phép dịch chuyển mẫu một vị trí ở thuật toán brute-force. Tuy nhiên, nếu như một thuật toán phải kiểm tra tất cả các ký tự của mẫu trên tất cả các thử nghiệm, nó sẽ tốn chi phí thời gian. Do vậy, ý tưởng của việc tăng cường đầu vào làm cho việc so sánh lặp đi lặp lại là không cần thiết, giảm chi phí thời gian. Có thể tính trước kích thước cần dịch chuyển và lưu trữ chúng vào một bảng. Bảng sẽ được lập chỉ mục cho tất cả các loại ký tự có trong văn bản, mà trong quá trình dịch chuyển mẫu sẽ gặp chúng. Kể cả văn bản ngôn ngữ tự nhiên, nó có khoảng trống, dấu chấm câu và ký tự đặc biệt khác. Như vậy cần xây dựng một bảng *T*, nó sẽ chỉ ra các kích thước *t*(*c*) cần dịch chuyển.

Ví dụ như: cho mẫu là BARBER và văn bản gồm các ký tự từ A … Z và khoảng trắng được ký hiệu là gạch nối “\_” . Đầu tiên những ký tự *c* không có trong mẫu thì *t*(*c*) được gán giá trị 6 là chiều dài của mẫu, các ký tự có trong mẫu là B, E, R và A có giá trị *t*(*c*) lần lượt là 1, 2, 3 và 4. Kết quả như bảng 7.2a sau. Trong bảng này, dòng đầu của bảng là những ký tự có trong văn bản, dòng thứ hai chỉ ra số lần phải dịch chuyển mẫu khi ký tự cuối cùng của mẫu gióng hàng với ký tự của văn bản tương ứng trên dòng đầu của bảng:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Kí tự *c* | A | B | C | D | E | F | … | R | … | Z | \_ |
| *t*(*c*) | 4 | 2 | 6 | 6 | 1 | 6 | 6 | 3 | 6 | 6 | 6 |

**Bảng7.2a: chỉ ra những bước cần dịch chuyển mẫu dọc theo văn bản**

Sau đây là một thuật toán đơn giản ShiftTable để tính toán kích thước dịch chuyển mẫu được ghi trong bảng *Table*, giống như bảng 7.2a

**Thuật toán** *ShiftTable* (*P*[0*..m*-1])

// đầu vào: mẫu *P*[0..*m*-1] và một bảng gồm *l* các ký tự khác nhau có trong văn bản *T*,

// kể cả những ký tự đặc biệt như khoảng trắng, dấu chấm câu …

// đầu ra: bảng *Table* [0.. *l* - 1] ghi kích thước dịch chuyển mẫu

**for** i ← 0 to *l* - 1 **do** *Table* [*i*] ← *m*

**for** *j* ← 0 to *m* - 2 **do** *Table* [*P*[*j*]] ← *m* - 1- *j*

**return** *Table*

**Các bước giải của thuật toán Horspool**.

**Bước 1**: Đối với một mẫu có chiều dài *m* và một văn bản , xây dựng bảng chuyển dịch *Table*

như mô tả ở trên.  
**Bước 2**: Gióng hàng mẫu vào đầu của văn bản.

**Bước 3**: Lặp lại sau đây cho đến khi hoặc là một chuỗi con trong văn bản trùng với mẫu được

tìm thấy hoặc mẫu đã dịch chuyển ra ngoài ký tự cuối cùng của văn bản. Bắt đầu với ký tự cuối cùng trong mẫu, so sánh các ký tự tương ứng trong mẫu với văn bản cho đến khi nào tất cả *m* ký tự được so khớp (sau đó dừng lại) hoặc gặp một cặp không trùng nhau. Trong trường hợp này truy xuất giá trị *t*(*c*) của bảng *Table* ở cột mà *c* là ký tự của văn bản đang được gióng hàng với ký tự cuối cùng của mẫu, và dịch chuyển mẫu sang bên phải dọc theo văn bản với kích thước là *t*(*c*).

**Thuật toán Horspool so khớp một chuỗi được viết như sau:**

**Thuật toán**  *HorspoolMatching* (*P*[0*..m*-1], *T*[0*..n*-1])

// đầu vào: mẫu *P*[0*..m*-1], văn bản *T*[0*..n*-1], mà *T*[0] =0 là vị trí của ký tự đầu tiên trong

// văn bản và *T*[*n*-1] là vị trí của ký tự cuối cùng trong văn bản, (kể cả những ký tự đặc

// biệt của văn bản như khoảng trắng, dấu chấm câu …), văn bản *n* ký tự, mẫu *m* ký tự

// đầu ra: vị trí của ký tự đầu tiên của mẫu trùng với văn bản hay không có chuỗi nào của

// văn bản trùng với mẫu

*ShiftTable*(*P*[0*..m*-1])

*i* ← *m*-1 // vị trí cuối bên phải của mẫu

**while** *i* ≤ *n*-1 **do**

*k* ← 0 // số ký tự phù hợp

**while** *k* ≤ *m*-1 **and** *P*[*m* - 1- *k*] = *T* [*i - k*] **do** *k* ← *k*+1

**if** *k* = *m* **then** **return**  *i – m* + 1

**else** *i* ← *i* + *Table*[*T*[i]] **//** *i* làvị trí của văn bản mà ký tự cuối cùng của mẫu dịch

**//** chuyển đến để gióng hang với văn bản

**return** -1

**Ví dụ**: cho văn bản và mẫu sau:

Văn bản: JIM\_SAW\_ME\_IN\_A\_BARBERSHOP

Mẫu: BARBER

Quá trình thực hiện thuật toán, bảng *Table* trong thủ tục *ShiftTable* giống như bảng 7.2a, mẫu dịch chuyển 6 lần dọc theo văn bản để tìm được chuỗi con trong văn bản khớp với mẫu, như trong hình 7.2

JIM\_SAW\_ME\_IN\_A\_BARBERSHOP

BARBER

BARBER

BARBER

BARBER

BARBER

BARBER

**Hình 7.2: ví dụ thực hiện thuật toán Horspool**

Nếu coi bảng *Table* trong thuật toán *ShiftTable* được tính toán trước theo ý tưởng tăng cường đầu vào thì chi phí chính trong thuật toán Horspool là 2 vòng lặp While. Do vậy, trường hợp xấu nhất chi phí là *O*(*n*×*m*). Chiều dài *m* của mẫu thì khá nhỏ so với chiều dài *n* nên chi phí có thể được coi như là *O*(*n*). Rõ rang thuật toán Horspool có hiệu quả hơn thuật toán bute-force.

**7.2.2 Thuật toán Boyer-Moore**

Nếu ký tự bên phải nhất của mẫu không trùng khớp với ký tự *c* tương ứng trong văn bản, thuật toán Boyer-Moore thực hiện tương tự như thuật toán Horspool. Cụ thể là, nó dịch chuyển mẫu sang bên phải một số ký tự như trong bảng 7.2a ở trên.

Thuật toán Boyer-Moore có điểm khác với thuật toán Horspool là nó mang tính chất tổng hợp hơn, nghĩa là: không phải chỉ một ký tự cuối cùng của mẫu được so khớp với ký tự tương ứng *c* trong văn bản, mà xét đến *k* ký tự cuối cùng của mẫu đã được so trùng khớp với văn bản với (0<*k* <*m*). Do vậy kích thước dịch chuyển mẫu phải được tính toán lại, nó cũng dẫn đến thuật toán Boyer-Moore thực hiện nhanh hơn thuật toán Horspool. Kích thước dịch chuyển mẫu được phân chia thành hai trường hợp khác nhau là: Dịch chuyển ký hiệu-yếu (bad-symbol shift) và dịch chuyển hậu tố-mạnh (good-suffix shift). Xem xét các trường hợp sau:

**Trường hợp 1**

Khi *k* ký tự cuối cùng của mẫu được so trùng khớp với văn bản, nhưng ký tự tiếp theo *p*[*m*-*k*+1] của mẫu không trùng khớp với ký tự *c* của văn bản như hình sau:

*T*[0] … *c* *T*[*i* - *k*+1] … *T*[i] ... *T*[*n*-1]

| |

*p*[0] … *p*[*m*-*k*+1] *p*[*m*-*k*] … *p*[*m*-1]

Trong trường hợp này, thuật toán Boyer-Moore xác định kích thước dịch chuyển theo công thức *t*1(*c*) – *k*, mà *t*1(*c*) là kích thước cần dịch chuyển như thuật toán Horspool, giống như bảng 7.2a, *k* là số ký tự của mẫu đã so trùng khớp với văn bản. Đây là kiểu dịch chuyển loại 1 của thuật toán Boyer-Moore, và được gọi là kiểu dịch-chuyển ký hiệu-yếu (bad-symbol shift). Mẫu được dịch chuyển như sau:

*T*[0] … c *T*[*i* - *k*+1] … *T*[i] ... *T*[*n*-1]

| |

*p*[0] … *p*[*m*-*k*+1] *p*[*m*-*k*] … *p*[*m*-1] mẫu

*p*[0] … *p*[*m*-1] mẫu đã dịch chuyển

Ví dụ như, so khớp mẫu BARBER với văn bản, hai ký tự cuối E và R trùng với ký tự trong văn bản (*k*=2), nhưng ký tự tiếp theo là B thì không trùng với ký tự S của văn bản, lúc đó mẫu được dịch chuyển sang bên phải dọc theo mẫu một số vị trí là *t*1(*S*) – 2 = 6 – 2 = 4 vị trí:

*T*[0] … SER ... *T*[*n*-1]

BARBER

BARBER

Một dịch chuyển tương tự như trên, nhưng ký tự S của văn bản là ký tự A, ký tự này có trong mẫu thì *t*1(*A*) – 2 = 4 – 2 = 2 vị trí:

*T*[0] … AER ... *T*[*n*-1]

BARBER

BARBER

Nếu *t*1(*c*) – *k* ≤0, thì số vị trí dịch chuyển chỉ là 1, giống như thuật toán brute-force, vì không thể dịch chuyển một số âm vị trí. Do vậy công thức tổng quát cho dịch chuyển *d*1 vị trí là:

*d*1 = { *t*1(*c*) – *k* , 1}

**Trường hợp 2**

Đây là kiểu dịch chuyển loại 2, được gọi là dịch chuyển hậu tố-mạnh (good-suffix shift). Trường hợp này là *k* ký tự cuối cùng của mẫu được so đúng khớp với văn bản (*k*> 0), ta nói rằng hậu tố của mẫu có kích thước *k*, được so trùng khớp với văn bản, ký hiệu là suff(*k*) và nó cũng được gọi là chuỗi hậu tố của mẫu. Có hai trường hợp cần xem xét trong trường hợp 2 này:

Thứ nhất, trong mẫu có một chuỗi con khác giống như suff(*k*), thì mẫu được dịch chuyển với kích thước *d*2 từ kí tự bên phải của chuỗi con đến ký tự bên phải của chuỗi hậu tố. Ví dụ như: xét mẫu ABCBAB, nếu *k*= 1, nghĩa là suff(1) = B là ký tự phải nhất của mẫu, ngoài ra trong mẫu còn ký tự B thứ 2 là chuỗi con trùng với suff(1), tính từ phải sang trái. Do vậy kích thước dịch chuyển *d*2=2. Tương tự, cùng với mẫu này, nếu *k*= 2, suff(2) = AB, chuỗi con là 2 ký tự đầu tiên của chuỗi tính từ trái sang phải và *d*2=4. Hình sau, mô tả ví dụ này, chuỗi gạch dưới là suff(*k*), ký tự in nghiêng là chuỗi con giống như suff(*k*).

|  |  |  |
| --- | --- | --- |
| *k* | Mẫu | *d*2 |
| 1  2 | ABC*B*AB  *AB*CBAB | 2  4 |

Ngược lại không có chuỗi con của suff(*k*) trong mẫu, thì mẫu được dịch chuyển với kích thước là *m* bằng chiều dài của mẫu. Ví dụ như sau, chuỗi DBCBAB dịch chuyển 6 vị trí ( *c* ≠ *C*):

*T*[0] … *c*BAB ... *T*[*n*-1]

DBCBAB

DBCBAB mẫu được dịch 6 vị trí

Nhưng không phải dịch chuyển mẫu như ví dụ trên là đúng, vì có thể bỏ xót chuỗi trong văn bản trùng khớp với mẫu. Coi ví dụ như sau:

*T*[0] … *c*B*ABCBAB* ... *T*[*n*-1]

ABCBAB

ABCBAB mẫu dịch chuyển, bỏ xót chuỗi in nghiêng

Thứ hai, do dịch chuyển có thể bỏ xót chuỗi trùng khớp với văn bản, nên thuật toán dùng kỹ thuật tiền tố và hậu tố của mẫu để loại bỏ sai xót này. Nếu một mẫu có suff(*k*), *k*>0 và tồn tại một tiền tố của mẫu là chuỗi con của suff(*k*) và chuỗi con này ở vị trí bên trái nhất của suff(*k*). Ví dụ như: coi mẫu ABCBAB có suff(k) = BAB và tiền tố của mẫu là chuỗi con AB trùng với hai vị trí cuối cùng của suff(*k*). Khi đó kích thước dịch chuyển *d*2 bằng số vị trí tính từ ký tự phải nhất của tiền tố đến ký tự phải nhất của mẫu. Bảng 7.2b được gọi là bảng hậu-tố mạnh (good-suffix table) mô tả kích thước dịch chuyển của thuật toán Boyer-Moore trong trường hợp hai. Trong bảng này, ký tự gạch dưới là suff(*k*), ký tự in nghiêng là tiền tố, ký tự in đậm là chuỗi con của suff(*k*), đồng thời nó tổng hợp các trường hợp phải dịch chuyển mẫu trong trường hợp hai, (*k*=1 hay *k*=2 không có tiền tố).

|  |  |  |  |
| --- | --- | --- | --- |
| *k* | Mẫu | *d*2 | Lý do |
| 1  2 | ABC**B**AB  **AB**CBAB | 2  4 | có chuỗi con khác giống suff(*k*)  có chuỗi con khác giống suff(*k*) |
| 3  4  5 | *AB*CBAB  *AB*CBAB  *AB*CBAB | 4  4  4 | suff(k) có tiền tố  suff(k) có tiền tố  suff(k) có tiền tố |

**Bảng 7.2b: Bảng hậu-tố mạnh ghi kích thước dịch chuyển mẫu**

**Thuật toán Boyer-Moore**

**Bước 1**: Cho trước một mẫu và bảng bao gồm các ký tự có trong mẫu cũng như trong văn bản,

xây dựng bảng dịch chuyển ký tự-yếu như đã trình bày ở trên.

**Bước 2**: Xây dựng bảng dịch chuyển hậu-tố-mạnh cũng đã trình bày ở trên cho mẫu.

**Bước 3**: Gióng hàng mẫu vào đầu văn bản.

**Bước 4**: Lặp lại bước sau cho dến khi đã tìm được một chuỗi trong văn bản trùng khớp với mẫu

hay mẫu đã dịch chuyển ra ngoài văn bản. Khởi đầu so khớp ký tự cuối cùng của mẫu với ký tự tương ứng trong văn bản cho tới khi các ký tự của mẫu đã trùng khớp với văn bản hoặc gặp một ký tự của mẫu không trùng khớp với ký tự *c* của văn bản sau khi đã có *k* ký tự của mẫu trùng với văn bản (*k*>0). Trong trường hợp sau, truy tìm giá trị *t*1(*c*) trong bảng đã xây dựng trong bước 1. Nếu *k*>0, cũng truy xuất giá trị *d*2 trong bảng đã xây dựng ở bước 2. Dịch chuyển mẫu sang phải một số vị trí *d* theo công thức:

*d* = *d*1 nếu *k* = 0,

*d* = max{ *d*1, *d*2} nếu *k*> 0,

với *d*1= max{ *t*1(*c*) − k, 1}.

**Ví dụ**: Cho mẫu BAOBAD, văn bản tiếng Anh gồm các ký tự và khoảng trắng ký hiệu gạch nối. Bảng dịch chuyển ký tự-yếu được xây dựng như sau:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *c* | A | B | C | D | … | O | … | Z | \_ |
| *t*1(*c*) | 1 | 2 | 6 | 6 | 6 | 3 | 6 | 6 | 6 |

Và bảng dịch chuyển hậu-tố-mạnh cũng được xây dựng như dưới đây:

(ký tự gạch dưới là suff(*k*), in đậm là chuỗi con trùng với suff(*k*), in nghiêng là tiền tố)

|  |  |  |
| --- | --- | --- |
| ***k*** | **Mẫu** | ***d*2** |
| 1  2  3  4  5 | BAO**B**AB  *B*AOBAB  *B*AOBAB  *B*AOBAB  *B*AOBAB | 2  5  5  5  5 |

Hình 7.3 mô tả các bước dịch chuyển của mẫu cho ví dụ này. Trường hợp xấu nhất chi phí cho thuật toán Boyer-Morre là tuyến tính.

BESS\_KNEW\_ABOUT\_BAOBABS

| ||| |||||||

BAOBAB ||| ||||||| // *k* = 0, *t*1(*K*)- 0 = 6, *d*1= max(6-0, 1)=6

(*k* = 0, không cần kiểm tra *d*2), *d* = 6

||| |||||||

BAOBAB ||||||| // *k* = 2, *t*1( \_ ) = 6, *d*1= max(6-2, 1) = 4,

*d*2 = 5, *d* = 5

|||||||

BAOBAB||||| // *k* = 1, *t*1( \_ ) = 6, *d*1= max(6-1, 1) = 5, *d*2 = 2,

*d* = 5

||||||

BAOBAB // trùng khớp

**Hình 7.3: Một ví dụ so khớp chuỗi với thuật toán Boyer-Moore, mỗi lần mẫu**

**dịch chuyển *d* vị trí**

**Bài tập 7.2**

1. Áp dụng thuật toán Horspool tìm kiếm mẫu BAOBAD trong đoạn văn bản sau:

BESS\_KNEW\_ABOUT\_BAOBASS.

1. Cho một gen TCCTATTCTT và một dãy DNA:

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT

1. Xây dựng bảng dịch chuyển cho gen này
2. Xác định kích thước mỗi lần dịch chuyển gen dọc theo dãy DNA theo thuật toán Horspool
3. Tìm tổng số ký tự phải so sánh trong thuật toán Horspool để tìm kiếm những mẫu sau đây trong một văn bản gồm một ngàn ký tự số không.
4. 00001
5. 10000
6. 01010
7. Cho một văn bản *T* có chiều dài *n*. Tìm hai mẫu khác nhau có chiều dài *m* với (*n* ≥ *m*), để mà một mẫu cho chi phí tốt nhất, còn mẫu kia có chi phí xấu nhất, khi tìm kiếm hai mẫu này trên *T*( *T* tự chọn) bằng thuật toán Horspool.
8. Đưa ra một ví dụ cho thấy thuật toán Horspool phải thực hiện số lần so sánh nhiều hơn thuật toán brute-force trên cùng một mẫu với cùng một văn bản.
9. Cho văn bản có 1000 ký tự số không. Có bao nhiêu kí tự phải so sánh khi dùng thuật toán Boyer-Moore để tìm kiếm những mẫu sau đây:
10. 00001
11. 10000
12. 01010
13. Hiện thực giải thuật Horspool, Boyer-Moore và brute-force trong một ngôn ngữ tùy chọn. Thực hiện tìm kiếm một số mẫu trên cả ba thuật giải này để đưa ra sự so sánh chí phí giữa những thuật toán đã hiện thực.

**7.3 Băm**

Trong phần này, chúng ta xem xét một cách rất hiệu quả để thực hiện thao tác từ điển. Nhớ lại rằng một từ điển là một kiểu dữ liệu trừu tượng, cụ thể là, một tập với các thao tác tìm kiếm (tra cứu), chèn và xóa được xác định trên các phần tử của nó. Các phần tử của tập này có thể có thể là: số, ký tự chữ cái, chuỗi ký tự, v… v. Trong thực tế, các trường hợp là các mẩu tin (hồ sơ học sinh trong một trường học, hồ sơ công dân trong một cơ quan hành chính, hồ sơ về sách trong thư viện …).

Thông thường, hồ sơ bao gồm một số lĩnh vực. Ví dụ, hồ sơ sinh viên có thể chứa các mục mã số, tên sinh viên, ngày sinh, giới tính, địa chỉ, v..v. Trong số các mục này có ít nhất một mục được gọi khóa, ví dụ như mã số được sử dụng như khóa để xác định các mục còn lại. Trong phần trình bày dưới đây, giả sử rằng chúng ta phải thực hiện một từ điển của *n* hồ sơ với các khóa *K*1, *K*2, ..., *K*n.

Băm dựa trên ý tưởng về phân phối các khóa trong một mảng một chiều *H* [0 .. *m*-1] được gọi là một bảng băm. Việc phân phối được thực hiện bằng cách cho mỗi khóa nhận một giá trị từ hàm đã được tính toán trước gọi là hàm băm *h*. Giá trị này là một số nguyên giữa 0 và *m*-1, cũng được gọi là địa chỉ băm.

Ví dụ, nếu khóa là những số nguyên dương, một hàm băm có thể có dạng *h*(*K*) = *K* mod *m*, hiển nhiên, phần dư của phép chia cho *m* luôn có giá trị từ 0 đến *m*-1. Nếu khóa là những chữ cái, ta có thể gán một giá trị là *ord*(*K*) cho một chữ cái là vị trí của nó trong bảng chữ cái. Cuối cùng, nếu *K* là chuỗi ký tự *c*0*c*1 ... *cs*-1, ta có thể sử dụng như một lựa chọn dễ dàng như.

((*ci*)) mod *m*. Một lựa chọn tốt hơn để tính *h*(*K*) như sau:

*h* ← 0; **for** *i* ← 0 **to** *s* – 1 **do** *h* ← (*h* × *C* + *ord*(*ci*)) mod *m*,

trong đó *C* là một hằng số lớn hơn *ord*(*ci*)

Tổng quát, một hàm băm cần phải đáp ứng yêu cầu:

- Kích thước của bảng băm không nên quá lớn so với số lượng các khóa, nhưng nó phải đủ để

không gây ảnh hưởng hiệu quả về thời gian thực hiện.

- Một hàm băm phải dễ dàng tính toán các giá trị của nó.

- Một hàm băm tốt cần phải phân phối khóa trong các cột của bảng băm một cách đồng đều.

Rõ ràng, nếu ta chọn kích thước m cho bảng băm nhỏ hơn so với số lượng n khóa, khi đó có sự xung đột xảy ra, có thể hai hay nhiều khóa có chung một giá trị trong bảng băm. Coi hình 7.4, hai khóa *K*1 và *K*2 có chung giá trị trong bảng băm. Nhưng xung đột như thế cũng cần xét đến, ​​ngay cả khi m là lớn hơn đáng kể so với *n*. Trong thực tế, trong trường hợp xấu nhất, tất cả các khóa có thể được băm vào cùng một cột của bảng băm. Tuy nhiên, với một kích thước bảng băm được lựa chọn thích hợp và một hàm băm tốt, tình trạng xung đột ít xảy ra. Nhưng, mỗi chương trình băm phải có một cơ chế giải quyết xung đột. Cơ chế này là khác nhau trong hai kỹ thuật băm: băm mở (open hashing), còn gọi là chuỗi riêng biệt (separate chaining) và băm đóng (close hashing), còn gọi là địa chỉ mở (open addressing).

0 *b m* - 1

**Hình 7.4** Sự xung đột khi hai khóa nhận một giá trị băm

*Ki*

*Kj*

**Băm mở**

Trong băm mở, khóa được lưu trong danh sách liên kết gắn liền với các cột của bảng băm. Mỗi danh sách chứa tất cả các khóa đã được băm. Xem ví dụ, danh sách của các từ sau đây:

A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED.

Như một hàm băm, ta sử dụng các chức năng đơn giản cho chuỗi ở trên, tức là, bổ sung thêm các vị trí các chữ cái của một từ trong bảng chữ cái và tính tổng các số dư sau khi chia cho 13. Bắt đầu với một bảng trống rỗng. Khóa đầu tiên là từ A, nó ở vị trí 1 trong bảng chữ cái nên giá trị băm của nó là *h*(A) = 1 mod 13 = 1. Khóa thứ hai là từ FOOL, lần lượt F ở vị trí 6, O vị trí 15 và L vị trí 12 trong bảng chữ cái. Do vậy, *h*(FOOL)=(6 +15 +15 +12) mod 13 = 9, nên FOOL được lưu vào cột thứ 9 trong bảng băm. Một cách tương tự như vậy, kết quả cuối cùng được đưa ra trong hình 7.5, lưu ý có xung đột giữa khóa ARE và SOON, vì *h*(ARE) = (1 +18 +5) mod 13 = 11 và *h*(SOON) = (19 +15 +15 14) mod 13 = 11.

Làm thế nào để tìm kiếm trong từ điển được cài đặt là một bảng danh sách liên kết? Vấn đề này thì đơn giản khi tìm kiếm một khóa. Ví dụ muốn tìm kiếm khóa KID trong bảng băm của hình 7.5, trước tiên chúng ta tính giá trị của hàm băm cho khóa: *h*(KID) = 11. Bởi vì cột thứ 11 trong bảng băm không rỗng, và nó có hai từ ARE và SOON, so sánh khóa KID với hai từ này, rõ ràng là không trùng khớp với từ nào, kết thúc tìm kiếm với kết quả không tìm thấy khóa KID.

Khóa A FOOL AND HIS MONEY ARE SOON PARTED

Địa chỉ băm 1 9 6 10 7 11 11 12

A AND MONEY FOOL HIS ARE PARTED

SOON

**Hình 7.5:** Ví dụ hàm băm với danh sách liên kết và những con trỏ

0 1 2 3 4 5 6 7 8 9 10 11 12

Nhìn chung, hiệu quả của việc tìm kiếm phụ thuộc vào độ dài của danh sách liên kết, trong đó, lần lượt, phụ thuộc vào từ điển và bảng kích thước bảng băm, cũng như chất lượng  
khóa của hàm băm.

Hàm băm phân phối *n* khóa một cách không đều làm tăng số lượng xung đột, và do đó tăng chi phí tìm kiếm. Nếu nó phân phối *n* khóa một cách đồng đều trong bảng băm có *m* cột, thì *α* = *n*/*m*, được gọi là hệ số tải (load factor) của bảng băm, nó đóng một vai trò quan trọng trong việc tính hiệu quả của băm. Đặc biệt, số lượng trung bình của con trỏ (là những mũi tên trong hình 7.5) được kiểm tra trong các tìm kiếm thành công *S*, và tìm kiếm không thành công *U* là:

*S* ≈ 1+*α*/2 và *U* = *α*,

Thông thường, ta muốn *α* gần với 1. Nó quá nhỏ sẽ có nhiều cột rỗng trong danh sách và như vậy không hiệu quả về bộ nhớ, nó quá lớn thì thời gian tìm kiếm lâu hơn, vì tốn thời gian so sánh với các khóa trong một cột. Nhưng nếu chúng ta có *α* gần với 1, thì việc tìm kiếm nhanh hơn, trên mức trung bình vì có thể chỉ cần một vài so sánh. Thực ra, ngoài việc so sánh, cần phải dành nhiều thời gian để tính giá trị của hàm băm cho một khóa tìm kiếm, nhưng nó hoạt động với thời gian là hằng số, độc lập với *n* và *m*. Lưu ý rằng có được hiệu quả đáng kể này không chỉ là do kết quả của kỹ thuật băm mà còn tốn chi phí sử dụng bộ nhớ.

Hai thao tác trong từ điển là chèn và xóa thì gần giống như thao tác tìm kiếm. Chèn thường được thực hiện ở cuối danh sách. Xóa được thực hiện bằng cách tìm kiếm một khóa và sau đó gỡ bỏ nó ra khỏi danh sách. Do đó, hiệu quả của thao tác này là giống hệt với tìm kiếm, và chúng đều là *O*(1) trong trường hợp trung bình nếu số lượng *n* khóa gần bằng kích cỡ *m* của bảng băm.

**Băm đóng**

Trong băm đóng, tất cả các khóa được lưu trữ trong bảng băm mà không sử dụng danh sách liên kết. (điều này hàm ý rằng kích thước *m* của bảng phải ít nhất là bằng số lượng *n* của các khóa.) Chiến lược khác nhau có thể được dùng để giải quyết xung đột. Đơn giản nhất được gọi là thăm dò tuyến tính (linear probing).

Nếu cột của bảng băm rỗng, thì khóa mới được lưu ở đây, nếu không rỗng thì lưu vào cột rỗng gần nhất, và cứ tiếp tục như thế cho đến khi các khóa đã được lưu vào bảng băm. Lưu ý rằng nếu đã đến cuối bảng băm, việc lưu khóa được chuyển đến đầu bảng, nghĩa là, nó được xử lý như là một mảng vòng tròn.

Ví dụ: dùng lại các khóa như trong hình 7.5. Các khóa lần lượt được lưu vào bảng băm như sau:

Dòng 1 của bảng băm, lưu khóa A ở cột 1, vì cột này rỗng, dòng 2 lưu khóa FOOL vào cột 9 vì cột này cũng rỗng, một cách tương tự các dòng 3, 4, 5 và 6, các khóa AND, HIS, MONEY và ARE lần lượt được ghi vào cột 6, 10, 7 và 11. Dòng 7 khóa SOON cần ghi vào cột 11, nhưng cột 11 đã có khóa ARE, nên nó được ghi vào vột rỗng gần nó nhất, đó là cột 12. Vì đã đi đến cột cuối cùng của bảng băm, nên trở lại đầu bảng, dòng 8, khóa PARTED được ghi vào cột 0. Dòng 8 của bảng băm là kết quả cuối cùng. Quá trình này đã xây dựng được bảng băm bằng chiến lược thăm dò tuyến tính. Hình 7.6 minh họa cho ví dụ này.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Khóa | A | FOOL | AND | HIS | MONEY | ARE | SOON | PARTED |
| Địa chỉ băm | 1 | 9 | 6 | 10 | 7 | 11 | 11 | 12 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|  | A |  |  |  |  |  |  |  |  |  |  |  |
|  | A |  |  |  |  |  |  |  | FOOL |  |  |  |
|  | A |  |  |  |  | AND |  |  | FOOL |  |  |  |
|  | A |  |  |  |  | AND |  |  | FOOL | HIS |  |  |
|  | A |  |  |  |  | AND | MONEY |  | FOOL | HIS |  |  |
|  | A |  |  |  |  | AND | MONEY |  | FOOL | HIS | ARE |  |
|  | A |  |  |  |  | AND | MONEY |  | FOOL | HIS | ARE | SOON |
| PARTED | A |  |  |  |  | AND | MONEY |  | FOOL | HIS | ARE | SOON |

**Hình 7.6:** bảng băm được xây dựng bằng thăm dò tuyến tính

Đi tìm một khóa *K*, ta tính *h*(*K*) mà *h* là hàm băm như đã nói trước đây, nếu cột *h*(*K*) của bảng băm là rỗng thì kết luận không tìm thấy. Nếu cột chứa khóa *K*, thì dừng vì đã tìm thấy *K*, ngược lại tìm ở cột tiếp theo, cứ như vậy cho đến khi hoặc là tìm thấy *K* hoặc là cột rỗng thì kết quả là không tìm thấy. Ví dụ, tìm khóa KID trong bảng băm của hình 7.6, ta có *h*(LIT) =(12+9+20) mod 13 = 2. Cột 2 rỗng, kết qủa là không tìm thấy. Nếu tìm khóa KID thì *h*(KID)=(11+9+4) mod 13=11, ta phải so sánh KID với ARE, SOON, PARTED và A trước khi có kết luận là không tìm thấy.

Phân tích bài toán thăm dò tuyến tính khó hơn nhiều so với bài toán băm mở. Thời gian trung bình cho việc phải truy cập vào bảng băm với hệ số tải *α* trong các tìm kiếm thành công *S* và không thành công *U* là:

S ≈ (1+ ) và *U* ≈ ( 1+

Giá trị của *S* và *U* tăng khi giá trị *α* tăng như bảng sau đây:

|  |  |  |
| --- | --- | --- |
| *α* | (1+ ) | ( 1+ |
| 50%  75%  90% | 1.5  2.5  5.5 | 2.5  8.5  50.5 |

Hiệu suất của thăm dò tuyến tính giảm đi do hiện tượng phân nhóm. Một nhóm trong thăm dò tuyến tính là một chuỗi các cột liên tiếp nhau đã dùng để lưu các khóa. Ví dụ dòng cuối của bảng băm trong hình 7.6 có hai nhóm. Việc xuất hiện những nhóm trong bảng băm làm cho các thao tác từ điển kém hiệu quả, một nhóm tăng lên về kích thước, xác xuất một phần tử mới gia nhập nhóm tăng lên. Ngoài ra hai nhóm có thể kết thành một nhóm lớn hơn, khi có một phần tử được chèn vào làm nối kết chúng lại, khiến số lượng nhóm cũng như kích thước của nhóm tăng.

Một số phương pháp giải quyết xung đột đã được đề xuất. Một trong chúng là phương pháp băm kép. Phương pháp này, sử dụng một giá trị gia tăng *s*(*k*) là một hằng số. Khi có xung đột tại vị trí *l* = *h*(*K*) của bảng băm, thì vị trí cho khóa cần ghi vào bảng băm tránh được xung đột được tính theo:

(*l* + *s*(*K*)) mod *m*, (*l* + 2*s*(*K*)) mod *m*, (*l* + 3*s*(*K*)) mod *m*, ... (\*)

Ví dụ: cho *m* = 7, chọn *s*(*k*) = 1, ghi các các khóa J, G, K, N và Q vào bảng băm.

Ta có: *h*(*J*) = 10 mod *m* = 3,

tương tự *h*(*G*) = 0, *h*(*K*) = 4, *h*(*N*) = 0 và *h*(*Q*) = 3.

Khởi đầu bảng băm rỗng, lần lượt J đưa vào vị trí 3, G vào vị trí 0, K vào vị trí 4 trong bảng băm. Tiếp tục đưa N vào vị trí 0, vị trí 0 đã bị chiếm chỗ bởi khóa G, có xung đột tại vị trí 0 (*l* = 0) của bảng băm nên dùng phương pháp băm kép để tính vị trí khác cho khóa N.

Ta có: (*l* + *s*(*k*)) mod *m* = (0 + 1) mod 7 = 1, nên ghi N vào vị trí 1 của bảng băm.

Tương tự khóa Q bị xung đột tại vị trí 3 (*l* = 3) nên ghi Q vào vị trí:

(*l* + *s*(*k*)) mod *m* = (3 + 1) mod 7 =4, nhưng tại vị trí 4 lại có xung đột với khóa K, nên phải tính lại. Vì (*l* + 2*s*(*k*)) mod m = (3 + 2) mod 7 = 5, vị trí 5 còn trống nên ghi Q vào vị trí 5 của bảng băm. Kết quả như hình 7.7.

0 1 2 3 4 5 6 . . .

G N J K Q . . .

**Hình 7.7:** Ví dụ dùng phương pháp băm kép

Để đảm bảo rằng tất cả các vị trí trong bảng được tính bằng biểu thức (\*), giải quyết được xung đột thì các giá trị gia tăng *s*(*k*) và kích thước *m* của bảng phải là số nguyên tố. Những hàm thường được dùng trong trường hợp này là *s*(*k*) = *m* – 2 - *k* mod (*m* - 2) và *s*(*k*) = 8 - ( *k* mod 8) cho bảng nhỏ, với bảng lớn thì *s*(*k*) = *k* mod 97+ 1.

Phân tích bài toán băm kép là khá khó. Một số kết quả và kinh nghiệm thực tế thì phương pháp này với một hàm băm tốt thì băm kép là một giải pháp vượt trội trong thăm dò tuyến tính. Nhưng hiệu quả của nó cũng bị suy giảm khi các cột của bảng gần như không rỗng. Một giải pháp tự nhiên trong tình hình như thế là băm lại: bảng hiện tại được quét, và tất cả các khóa của nó được di chuyển vào một bảng lớn hơn.

So sánh hiệu quả khi dùng phương pháp băm với phương pháp cây tìm kiếm cân bằng trong thực hiện các thao tác từ điển.

* *Hiệu quả của phương pháp*: Với băm, tìm kiếm, chèn, và xóa có thể được thực hiện với chi phí thời gian trung bình là *O*(1), trường hợp xấu nhất là *O*(*n*). Đối với cây tìm kiếm cân bằng, hiệu quả thời gian trung bình là O(log*n*) cho cả trường hợp trung bình và xấu nhất.
* *Duy trì thứ tự của các khóa*: Khác với cây tìm kiếm cân bằng, băm không cần sự thứ tự của khóa. Điều này làm cho băm không phù hợp cho các ứng dụng cần để lặp qua các khóa theo thứ tự hoặc yêu cầu truy vấn như: đếm số lượng các khóa giữa cận trên và cận dưới.

Phương pháp băm được những nhà nghiên cứu của công ty IBM đề xuất từ năm 1950, băm đã cho tìm thấy nhiều ứng dụng quan trọng. Đặc biệt, nó đã trở thành một kỹ thuật chuẩn để lưu trữ bảng ký hiệu, bảng những ký hiệu của một chương trình máy tính được tạo ra trong quá trình biên dịch. Băm là khá thuận tiện cho các ứng dụng của AI như đánh cờ. Với một số sửa đổi gọi là băm mở rộng, nó cũng đã được chứng minh là hữu ích để lưu trữ từ điển rất lớn trên đĩa. Trong phần tiếp theo, thảo luận phương pháp dùng B - cây, một phương pháp hữu hiệu để lưu trữ từ điển lớn.

**Bài tập 7.3**

1. Cho các khóa {30, 20, 56, 75, 31, 19}và hàm băm *h*(*K*) mod 11
2. Xây dựng bảng băm mở
3. Tìm khóa nào trong bảng băm ở câu a, tốn chi phí so sánh lớn nhất, giải thích
4. Tìm số lượng khóa trung bình phải so sánh khi tìm một khóa trong bảng băm
5. Cho các khóa {30, 20, 56, 75, 31, 19}và hàm băm *h*(*K*) mod 11
6. Xây dựng bảng băm đóng
7. Tìm khóa nào trong bảng băm ở câu a, tốn chi phí so sánh lớn nhất, giải thích
8. Tìm số lượng khóa trung bình phải so sánh khi tìm một khóa trong bảng băm
9. Tìm xác xuất của tất cả *n* khóa được băm vào cùng một cột của bảng băm có kích thước *m*, giả sử hàm băm đã phân phối đồng đều các khóa vào các cột của bảng băm.
10. Nếu tất cả các khóa được băm vào cùng một địa chỉ, thì chi phí tìm kiếm một khóa là bao nhiêu?
11. Điền vào bảng sau với chi phí trung bình ( các khóa được phân phối đồng đều vào bảng băm) cho việc cài đặt từ điển.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | mảng không có thứ tự | mảng có thứ tự | cây nhị phân tìm kiếm | băm mở | thăm dò tuyến tính |
| Tìm kiếm |  |  |  |  |  |
| Chèn |  |  |  |  |  |
| Xóa |  |  |  |  |  |

1. Hiện thực một chương trình từ điển Anh-Việt với các thao tác tìm kiếm, chèn bằng phương pháp băm kép. Trong đó:
2. Thao tác tìm kiếm: thiết kế màn hình để người dùng có thể nhập vào từ khóa cần tìm hay chọn từ danh sách các khóa trên màn hình được hiển thị theo thứ tự từ điển, kết quả hiển thị từ khóa tìm được và nghĩa tiếng Việt kèm theo.
3. Thao tác chèn, người soạn thảo từ điển, nhập từ tiếng Anh mới và soạn thảo ngữ nghĩa tiếng Việt tương ứng.

**7.4 B- Cây**

Ý tưởng của việc sử dụng bộ nhớ thêm (extra space) để truy cập nhanh hơn trên một tập dữ liệu nếu các dữ liệu có số lượng mẩu tin rất lớn mà cần phải được lưu trữ trên đĩa. Chỉ mục là phương pháp tốt trong việc tổ chức tập dữ liệu như thế, nó cung cấp một số thông tin về vị trí của mẩu tin. Đối với các tập dữ liệu có cấu trúc (như trái ngược với "cấu trúc" dữ liệu như văn bản, hình ảnh, âm thanh và video), Kiểu chỉ mục quan trọng nhất là B-cây, được giới thiệu bởi R. Bayer và E. McGreight. Nó mở rộng cây tìm kiếm bằng cách cho phép có nhiều hơn một khóa trong cùng một nút của cây.

Trong B-cây, được xem xét ở đây, tất cả các mẩu tin dữ liệu (hay những khóa mẩu tin) được lưu trữ tại mức lá, theo thứ tự tăng dần của các khóa. Các nút cha được sử dụng để lập chỉ mục. Cụ thể, mỗi nút cha chứa *n*-1 khóa có thứ tự *K*1 < ... < *Kn*-1. Các khóa được xen vào với *n* con trỏ chỉ đến nút con, tất cả các khóa trong cây con *T*0 có kích thước nhỏ hơn *K*1, tất cả các khóa trong cây con *T*1 lớn hơn hoặc bằng *K*1 và nhỏ hơn so với *K*2, với *K*1 bằng khóa nhỏ nhất trong *T*1, v …v, thông qua các cây con cuối *Tn*-1 có các khóa lớn hơn hoặc bằng *Kn*-1 với *Kn*-1 bằng khóa nhỏ nhất trong *Tn*-1 (xem Hình 7.8).

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *P*0 | *K*1 | *P*1 | … | *Pi*-1 | *Ki* | *Pi* | … | *Pn*-2 | *Kn*-1 | *Pn*-1 |

*T*0

*T*1

*Ti-*1

*Ti*

*Tn-*2

*Tn-*1

**Hình 7.8:** Nút cha của B-cây

Ngoài ra, một B-cây có bậc *m* ≥ 2 phải thỏa mãn những tính chất sau:

- Gốc hoặc nó cũng là nút lá hoặc nó có từ 2 đến *m* con.  
- Mỗi nút, trừ nút gốc và nút lá, có từ ⌈*m*/2⌉ đến *m* con (và do đó giữa ⌈*m*/2⌉-1 và *m*-1 khóa).  
- Tất cả các lá ở cùng cấp.

- Mỗi nút khác nút gốc có từ ⌈*m*/2⌉-1 khóa đến *m*-1 khóa

- Nút gốc có từ 1 đến *m*-1 khóa

**Hình 7.9**: Một B-cây bậc 5

13

2 4

5 8

16 20 24 28

6 7

14 15

17 19

22 23

25 27

9 11 12

28 32 37 40

Một ví dụ về một B-cây bậc 5 như hình 7.9. Trong hình này, các nút gốc có í nhất 3 con và nhiều nhất là 5 con. Do vậy, mỗi nút ( trừ nút gốc) có ít nhất là 2 khóa. Số con nhiều nhất của mỗi nút là 5, nên số khóa nhiều nhất trong mỗi nút là 4.

Tìm kiếm trong một B-cây tương tự như tìm kiếm trong cây nhị phân tìm kiếm. Bắt đầu từ gốc, đi theo một chuỗi các con trỏ đến các lá có thể chứa khóa cần tìm. Sau đó, tìm kiếm các khóa cần tìm trong các khóa của lá. Lưu ý rằng kể từ khi khóa được lưu có thứ tự, tại cả những nút cha và những nút lá, chúng ta có thể sử dụng tìm kiếm nhị phân, nếu số lượng các khóa tại một nút là đủ lớn.

Không quan tâm đến số lượng khóa cần so sánh, cần quan tâm đến ứng dụng điển hình của cấu trúc dữ liệu này. Khi được sử dụng để lưu trữ một tập tin dữ liệu lớn trên một đĩa, các nút của một B-cây thông thường tương ứng với những trang đĩa từ. Kể từ thời gian cần thiết để truy cập vào một trang đĩa thường là lớn hơn vài bậc so với thời gian cần thiết để so sánh các khóa trong bộ nhớ máy tính. Số lượng truy cập đĩa ảnh hưởng khá lớn đến hiệu quả.

Bao nhiêu nút của B-cây cần phải truy cập để tìm kiếm một mẩu tin với một khóa cho trước. Số nút cần truy cập bằng chiều cao của cây cộng với 1. Để ước tính chiều cao, chúng ta hãy tìm số nhỏ nhất của các khóa trên B-cây có bậc *m* và chiều cao là số dương *h*. Các gốc của cây sẽ có ít nhất một khóa. Mức 1 sẽ có ít nhất hai nút, mỗi nút có ít nhất ⌈*m*/2⌉-1 khóa, nên tổng số khóa tối thiểu là 2(⌈*m*/2⌉-1). Mức 2 sẽ có ít nhất 2⌈*m*/2⌉ khóa (con của các nút ở mức 1), mỗi nút có ít nhất ⌈*m*/2⌉-1 khóa, nên tổng số khóa tối thiểu là 2⌈*m*/2⌉(⌈*m*/2⌉-1). Tổng quát, các nút ở mức *i* với 1 ≤ *i* ≤ *h*-1, sẽ có ít nhất 2⌈*m*/2⌉*i*-1(⌈*m*/2⌉-1) khóa. Cuối cùng, mức *h* là mức lá, sẽ có ít nhất 2⌈*m*/2⌉*h*-1 nút, mỗi nút có ít nhất một khóa. Như vậy, đối với B-cây bậc *m* có *n* nút và chiều cao *h*> 0, ta có bất đẳng thức sau:

*n* ≥1 + *i*-1 (⌈*m*/2⌉-1)+2⌈*m*/2⌉*h*-1.

Sau một số lần đơn giản, bất đẳng thức sẽ là:

*n* ≥ 4 ⌈*m*/2⌉*h*-1-1,

và tìm được cận trên chiều cao *h* của B-cây bậc *m* có *n* nút:

*h* ≤ ⌊ log⌈m/2⌉ ⌋ + 1

Bất đẳng thức trên cho thấy chi phí tìm kiếm trong B-cây là *O*(log*n*), điều quan trọng không chỉ là hiệu quả về thời gian, nhưng là cho thấy số lượng cần truy cập đĩa, khi B-cây phải lưu trữ trong bộ nhớ ngoài. Bảng dưới đây ghi một vài giá trị ước lượng về bậc *m* và cận trên *h* của B-cây cho một tập tin có 100 triệu mẩu tin:

|  |  |  |  |
| --- | --- | --- | --- |
| Bậc *m*  Cận trên của *h* | 50  6 | 100  5 | 250  4 |

Coi một B-cây bậc 2, chứa một triệu khóa, cây này chính là cây nhị phân tìm kiếm cân bằng. Chi phí tìm kiếm một khóa trên cây này, như đã biết là:

log2106 [≈](http://vi.wikipedia.org/w/index.php?title=Equals_sign&action=edit&redlink=1) log2220 = 20,

trong khi đó: một B-cây bậc 100, cũng chứa một triệu khóa, thì chi phí tìm kiếm một khóa trên cây sẽ là:

log100106 = log1001003 = 3.

Điều này cho thấy, với số lượng khóa khá lớn, cây được lưu trữ trên bộ nhớ ngoài thì số lần truy cập đĩa trên B-cây giảm khá nhiều so với cây nhị phân tìm kiếm cân bằng có cùng số lượng khóa với B-cây.

Thao tác chèn và xóa thì không đơn giản bằng tìm kiếm, nhưng cả hai cũng có thể được thực hiện với cho phí về thời gian là O(log*n*). Phần sau đây trình bày thao tác chèn và xóa một khóa trên B-cây.

**7.4.1 Thao tác chèn**

Khi chèn một khóa, đầu tiên phải tìm kiếm nó trong B-cây. Đi từ gốc đến các nút lá, nếu không tìm thấy và có một vị trí trống phù hợp trên nút lá thì khóa được chèn tại đây (lưu ý thứ tự các khóa trong nút lá sau khi chèn này phải được bảo đảm). Trường hợp sau khi chèn, nút lá bị tràn (số lượng khóa đã vượt qui định cho nút lá trong B-cây), nút này phải tách thành 2 nút bằng cách: đẩy một khóa ở giữa lên nút cha, một nửa số khóa bên phải được dời sang nút mới. Trường hợp này có thể tách nút hàng loạt, khi mà nút cha được thêm một khóa từ nút con đẩy lên, có thể chúng cũng bị tràn, nên phải tách nút này và cứ thực hiện tách nút như thế, cho đến khi các nút ở mức trên không bị tràn, sau đó phải chỉnh lại con trỏ chỉ đến các nút đã tách cho phù hợp. Việc chèn phải tách nút gây ra B-cây tăng chiều cao.

Ví dụ chèn một danh sách các khóa: 3,14,7,1,8,5,11,17,14,6,23,12,20,26,4,16,18,24,25 và 19 vào một B-cây bậc 5. (Nhắc lại: với B-cây bậc 5 thì một nút có nhiều nhất là 5 con và 4 khóa, tất cả các nút trừ nút gốc có ít nhất 2 khóa). Quá trình thực hiện như hình 7.10.

(b)Chèn 5, 11 và 17 không phải tách nút, tiếp tục chèn khóa 13, phải tách nút

7

1 3 5

8 11 14 17

7 13

1 3 5

8 11

14 17

(a)Khởi đầu cây rỗng, bốn khóa đầu tiên 3, 14, 7 và 1 được chèn vào một nút, tiếp theo chèn khóa 8, nút tràn, tách nút

7

1 3

8 14

1 3 7 14

(c) Chèn khóa 6, 23, 12 và 20, không tách nút, chèn tiếp khóa 26, tách nút

8 11 12

7 13

1 3 5 6

14 17 20 23

1 3 5 6

7 13 20

14 17

20 23

8 11 12

(d) Chèn khóa 4, nút tràn, tách nút

8 11 12

14 17

23 26

4 7 13 20

1 3

5 6

(e) Chèn khóa 16, 18, 24 và 25 không tách nút

8 11 12

4 7 13 20

1 3

5 6

14 16 17 18

23 24 25 26

Hình 7.10 ví dụ chèn vào B-cây bậc 5

(f) Cuối cùng chèn khóa 19 tách nút lá và nút cha, cây tăng chiều cao

17 20

13

14 16

18 19

4 7

8 11 12

23 24 25 26

1 3

5 6

**7.4.2 Thao tác xóa**

Thao tác xóa, ngược lại với thao thác chèn, thay vì tách nút, thì có thể phải gộp hai nút lại. Nói một cách khác là sau khi xóa một khóa, phải cân bằng lại cây để bảo đảm tính chất của B-cây theo đúng như bậc của cây lúc chưa xóa. Dùng lại B-cây là hình cuối trong hình 7.10 để thực hiện thao tác xóa. Quá trình được thực hiện như hình 7.11

11 12

17 24

14 16

19 23

25 26

13

4 7

1 3

5 6

Hình 7.11 ví dụ xóa trên B-cây bậc 5

(d) Cuối cùng, xóa khóa 5. Trường hợp này gây ra gộp nút hàng loạt, khóa 5 ở nút lá (gọi là nút L), khi xóa 5, nút L chỉ còn một khóa, nó không thỏa tính chất B-cây, đồng thời không thể mượn khóa của nút anh em trái hoặc phải của L (nút anh em chỉ có số lượng khóa tối thiểu). Khi này, sau khi xóa 5 phải gộp L vào với nút anh em bên trái thành một nút (gọi là M), M thỏa tính chất về số lượng khóa, nhưng gây ra nút cha của L (gọi là P) thiếu đi một con nên nó dư một khóa, khóa dư là cha của L sẽ đưa xuống M. Nhưng nút P chỉ còn một khóa là 7, không thể chấp nhận được. Do vậy, phải gộp P với nút anh em. Trong ví dụ này thì phải gộp P với nút anh em thành một nút có 3 khóa vì nút cha của nó lúc này chỉ có một khóa và chỉ có một con, nên phải gộp nút cha xuống nút con thành một nút có 4 khóa thỏa tính chất B-cây. Điều này làm cây bị co lại về chiều cao.

11 12

17 24

14 16

19 23

25 26

13

7

1 3 4 6

11 12

1 3 4 6

14 16

19 23

25 26

7 13 17 24

(c)Tiếp theo, xóa khóa 18. Khóa này ở nút lá (gọi là nút L), nhưng khi xóa khóa này, L chỉ còn một khóa khộng thỏa tính chất của cây bậc 5. Nếu nút anh em (gọi là nút N) ở ngay bên trái hoặc bên phải có số khóa lớn hơn số khóa tối thiểu của một nút, thì mượn một khóa của N. Nếu N ở bên trái thì mượn khóa bên phải nhất của N, ngược lại N bên phải thì mượn khóa trái nhất của N. Khóa mượn này được đưa lên thay thế cho khóa ở nút cha của N, khóa của nút cha bị thay thế đưa xuống L. Trong ví dụ này, mượn khóa 24 thay cho khóa 23, khóa 23 đưa xuống nút có khóa 18 bị xóa.

17 23

14 16

18 19

24 25 26

13

4 7

1 3

5 6

11 12

(b)Tiếp theo, xóa 20. Vì 20 không thuộc nút lá, khóa tiếp theo của 20 (theo thứ tự tăng dần) trong cây là khóa 23, khi này khóa 23 đang ở nút lá được đẩy lên nút cha thay thế cho 20, nút lá này còn lại 3 khóa, thỏa tính chất của B-cây bậc 5, nên không cần gộp nút.

(a)Xóa khóa 8: tìm thấy 8 ở nút lá, xóa nó, số khóa còn lại là 2, nên không cần gộp nút

13

17 20

14 16

18 19

23 24 25 26

4 7

1 3

5 6

11 12

**Một ví dụ khác cho thao tác xóa**

Cho một B-cây bậc 5, hãy xóa khóa 3, qui trình thực hiện như hình 7.12

10

3 6

13 18 21

11 12

14 16

19 20

24 26

1 2

4 5

7 9

10

6

13 18 21

11 12

19 20

24 26

7 9

1 2 4 5

14 16

Hình 7.12 Một ví dụ khác xóa trên B-cây bậc 5

13

11 12

19 20

24 26

7 9

1 2 4 5

14 16

18 21

6 10

(c)Nhưng bây giờ nút chỉ có khóa 6 cần phải bổ sung thêm khóa, nên nó mượn khóa 13 của nút anh em bên phải đưa lên thay thế khóa 10 của nút cha, đồng thời đẩy khóa 10 xuống kết với khóa 6. Chú ý rằng nút chứa khóa 11 và 12 phải trở thành con bên phải của khóa 10.

(b)Nút chứa khóa 5 cần mượn một khóa từ nút anh em trái hay phải, nhưng chúng đều có số lượng khóa tối thiểu, nên phải gộp nút chứa khóa 5 với nút bên trái nó

(a)Tìm đến khóa 4 ở nút lá, đẩy nó lên thay thế cho khóa 3, khi này nút lá chỉ còn một khóa 5

10

19 20

24 26

4 6

13 18 21

5

11 12

1 2

7 9

14 16

Một B-cây không phải luôn gắn liền với việc lập chỉ mục của một tập tin lớn, và nó có thể được coi là một loại cây tìm kiếm trong lớp cây tìm kiếm. Như ví dụ thao tác chèn ở trên, một B-cây có thể được xây dựng bằng cách chèn liên tiếp các mẩu tin vào cây ban đầu trống rỗng. Khi tất cả các khóa đều nằm trong các nút lá và các cấp trên của cây được tổ chức như một B-cây bao gồm một chỉ số. Toàn bộ cấu trúc như vậy được gọi là một B+-cây.

**Bài tập 7.4**

1. a- Chứng minh:

1⌈*m*/2⌉*i*-1(⌈*m*/2⌉ - 1) + 2⌈*m*/2⌉ *h*-1 = 4⌈*m*/2⌉ *h*-1 – 1.

b- Cho *n* ≥ 4⌈*m*/2⌉ *h*-1 – 1, chứng minh:

*h* ≤ ⌊ log⌈*m*/2⌉ ⌋ + 1

(nhắc lại: *n* là số nút trong một B-cây bậc *m* có chiều cao là *h*)

1. B-cây có bậc *m* nhỏ nhất là bao nhiêu, để số lần truy xuất đĩa khi tìm một khóa trong tập tin có 100 triệu mẩu tin không vượt quá 3 lần. Giả sử rằng nút gốc của B-cây đã lưu trữ trong bộ nhớ chính.
2. Coi hình cuối của hình 7.12, vẽ lại hình này khi:
3. Chèn khóa có giá trị 3
4. Chèn khóa có giá trị 22 và 23
5. Chỉ ra lối đi từ gốc của một B-cây đến khóa có giá trị lớn nhất
6. Coi hình cuối của hình 7.12, vẽ lại hình này khi xóa
7. Xóa khóa có giá trị 13
8. Xóa khóa có giá trị 9
9. Hiện thực chương trình để thực hiện thao tác tạo một B-cây bậc *m* và xóa một khóa bất kỳ.

**TỔNG KẾT**

* Bộ nhớ và thời gian được đánh đổi trong thiết kế thuật toán là một vấn đề thiết thực cả lý thuyết cũng như thực hành trong máy tính. Như một kỹ thuật thiết kế thuật toán, bộ nhớ cần được sử dụng để đánh đổi nhắm đến thời gian có được chi phí thấp.
* *Tăng cường đầu* vào là một trong hai kỹ thuật chính của việc đánh đổi bộ nhớ với thời gian thực hiện trong thiết kế thuật toán. Ý tưởng của nó là tiền xử lý toàn bộ hay một phần đầu vào của bài toán, và lưu trữ các thông tin thu được để hỗ trợ cho quá trình giải bài toán được nhanh hơn. Thuật toán sắp xếp bằng đếm phân phối và một vài thuật toán so khớp chuỗi là những thuật toán dựa trên kỹ thuật này.
* *Đếm phân phối* là một phương pháp đặc biệt để sắp xếp một danh sách các phần tử từ một tập nhỏ các giá trị có trong danh sách.
* *Thuật toán Horspool* so khớp chuỗi có thể được coi như một phiên bản đơn giản của thuật toán Boyer-Moore. Cả hai thuật toán này được dựa trên ý tưởng của việc tăng cường đầu vào và so sánh những ký tự của một mẫu từ phải sang trái với văn bản. Cả hai thuật toán cùng sử dụng bảng dịch chuyển ký hiệu-yếu, riêng thuật toán Boyer-Moore còn dung them bảng dịch chuyển thứ hai, gọi là bảng dịch chuyển hậu tố- mạnh.
* *Tiền cấu trúc* là kỹ thuật thứ hai nhắm đến khai phá vấn đề đánh đổi giữa thời gian và bộ nhớ, dùng bộ nhớ thêm để truy xuất dữ liệu nhanh hơn hay linh hoạt hơn. Băm và B+-cây là những ví dụ quan trọng của kỹ thuật tiền cấu trúc.
* *Băm* là một phương pháp rất hiệu quả để thực hiện thao tác từ điển. Nó được dựa trên ý tưởng là ánh xạ các khóa vào bảng một chiều. Vì kích thước của bảng bị giới hạn, phương pháp băm đã đưa ra cơ chế giải quyết xung đột, khi hai khóa băm vào chung một cột của bảng. Hai phương pháp của băm là: băm mở (các khóa được lưu trong danh sách liên kết bên ngoài của bảng băm) và băm đóng (các khóa được lưu trong bảng). Cả hai cho phép tìm kiếm, chèn, và xóa với chi phí thời gian trung bình là O(1).
* *B-cây* là cây tìm kiếm cân bằng, nó là trường hợp tổng quát của cây 2-3 bằng cách cho phép một nút có nhiều khóa. Ứng dụng chính của nó, được gọi là B+-cây, để lưu những chỉ mục của dữ liệu được lưu trên một đĩa. Bằng cách lựa chọn bậc thích hợp cho cây, có thể thực hiện các thao tác tìm kiếm, chèn và xóa trên cây chỉ tốn vài lần truy cập đĩa, ngay cả đối với các tập tin rất lớn.