Chương VII: Tìm kiếm - II

Tìm kiếm – Phần II

- Nội dung
  - Các dạng cây đặc biệt sử dụng trong tìm kiếm
    - Cây tìm kiếm đa nhánh
    - Cây 2:3
    - Cây nhị phân tìm kiếm tối ưu
  - Cấu trúc Bảng băm (Hash Table)
  - Tìm kiếm xâu mẫu (Pattern Matching)

Cây 2-3

- Cây tìm kiếm đặc biệt
  - Một nút nhánh có 2 hoặc 3 con
  - Tất cả các đường đi từ nút gốc tới nút lá đều có độ dài bằng nhau
  - Cây có một nút là trường hợp đặc biệt của cây 2-3
- Cấu trúc của các nút trong cây 2-3
  - Chỉ có nút lá chứa các giá trị (Các phần tử), các nút lá chứa các giá trị tăng dần (xét từ trái sang phải)
  - Các nút nhánh chứa thông tin về đường đi hỗ trợ cho việc tìm kiếm các giá trị

Cây 2-3

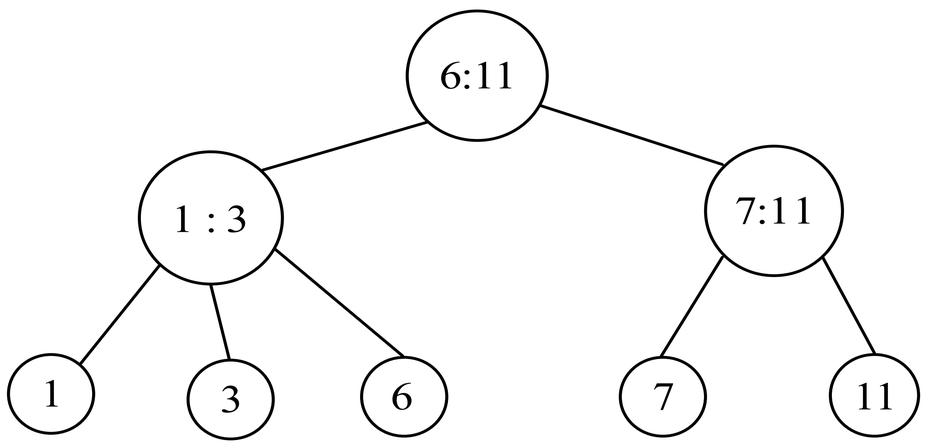
- Quy cách của nút lá trong cây 2-3

TYPE	VALUE

- Quy cách của nút nhánh của cây

TYPE	LPTR	LDATA	MPTR	MDATA	RPTR

Cây 2-3 – Ví dụ



Tìm kiếm trên cây 2-3

Function SEARCH-2-3(T,X)

1. p:= T;
2. while TYPE(p) =1 do begin

if X <= LDATA(p) then p:= LPTR(p);

else if X <= MDATA(p) then p:= MPTR(p);

else if RPTR(p) < > NULL then p:= RPTR(p)

else SEARCH-2-3 := NULL;

end

3. {Tìm được đến 1 nút lá}

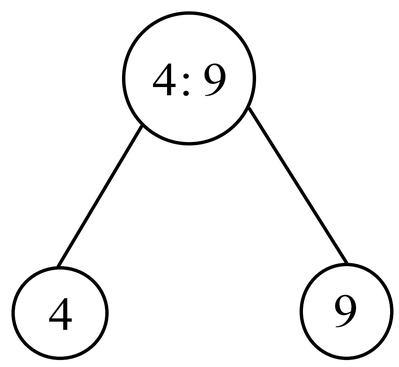
if VALUE(p) = X then SEARCH-2-3 := p; else SEARCH-2-3 := NULL;

4. return

Cây 2-3 : Bổ sung

- Bổ sung phải xét 4 trường hợp
  - Cây ban đầu rỗng
  - Cây ban đầu chỉ có 1 nút: Sau khi bổ sung, cây có thêm một nút nhánh

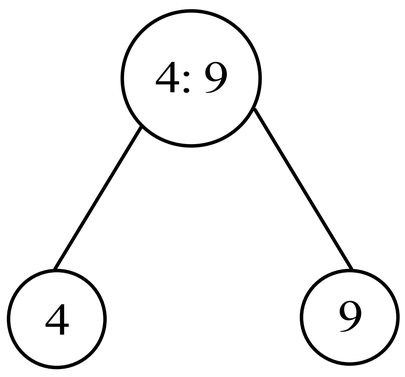
Cây ban đầu rỗng, bổ sung 4



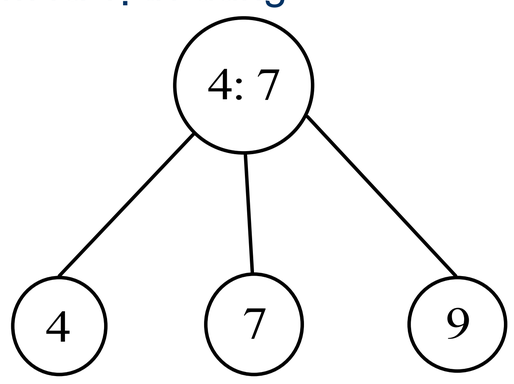
Cây ban đầu có 1 nút, bổ sung thêm 9

Cây 2-3 : Bổ sung

  - Cây ban đầu có nhiều nút, nút mới được bổ sung thành con của một nút hiện đang có 2 con: So sánh giá trị của nút mới với 2 nút con để tìm ra vị trí đúng



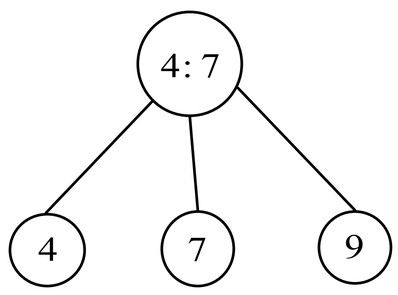
Trước khi bổ sung

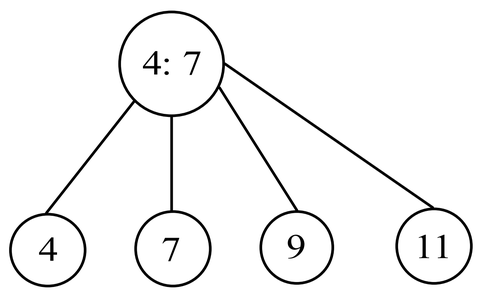


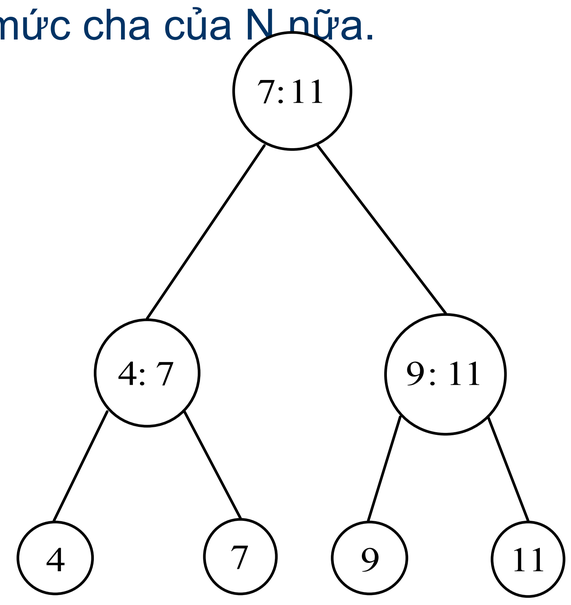
Sau khi bổ sung 7

Cây 2-3: Bổ sung

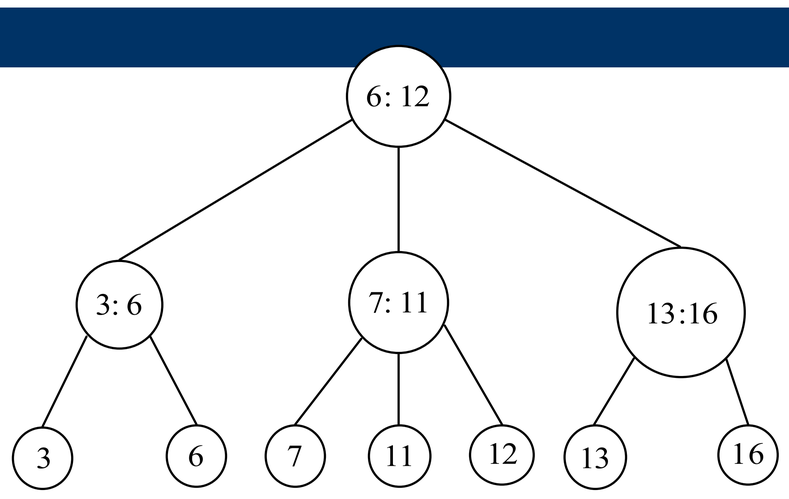
- Nút mới được bổ sung làm con của một nút N đã có 3 con: Tạo một nút nhánh mới N2 – sẽ là nút anh em bên phải của N, lấy 2 con cực phải của N làm con của N2. Việc tách có thể sẽ diễn ra ở các mức cha của N nữa.





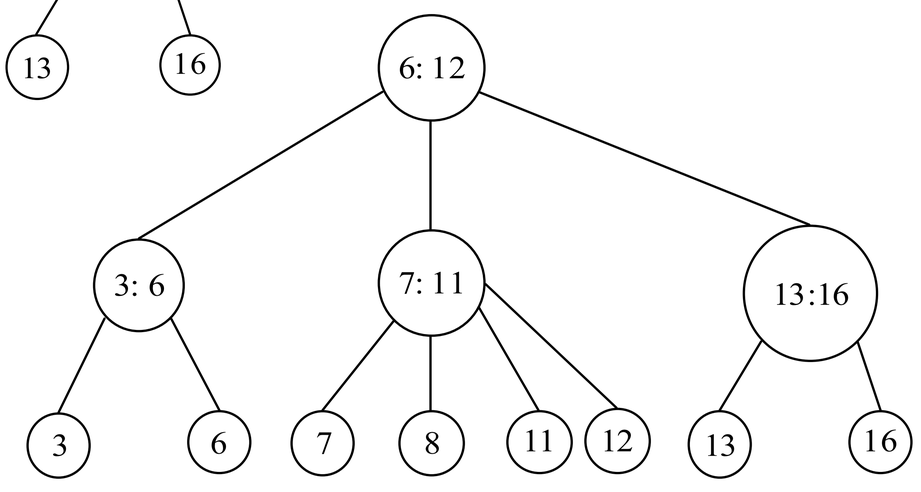


Cây 2-3 : Bổ sung

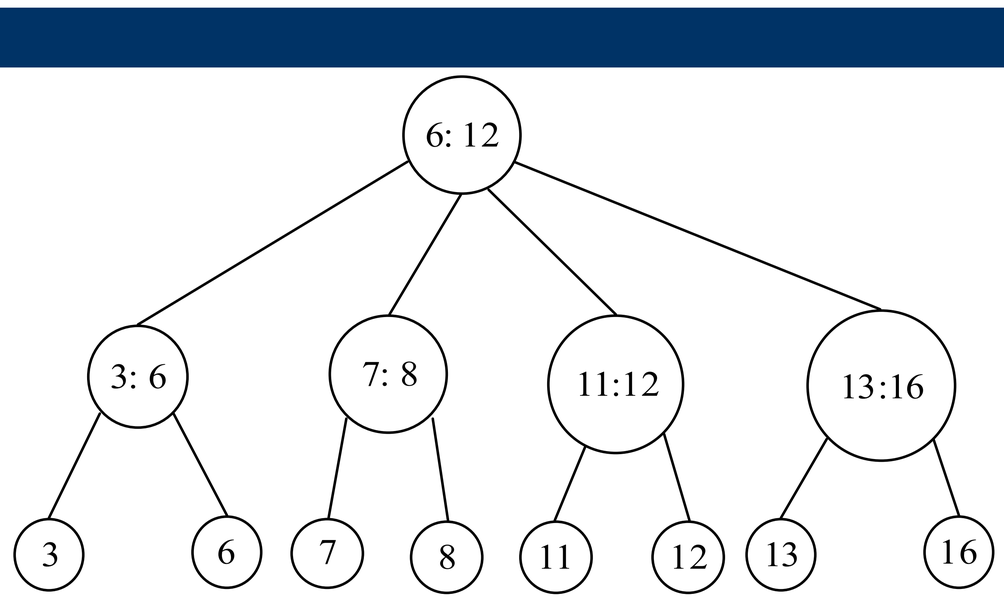


Trước khi bổ sung

Ngay sau khi bổ sung 8

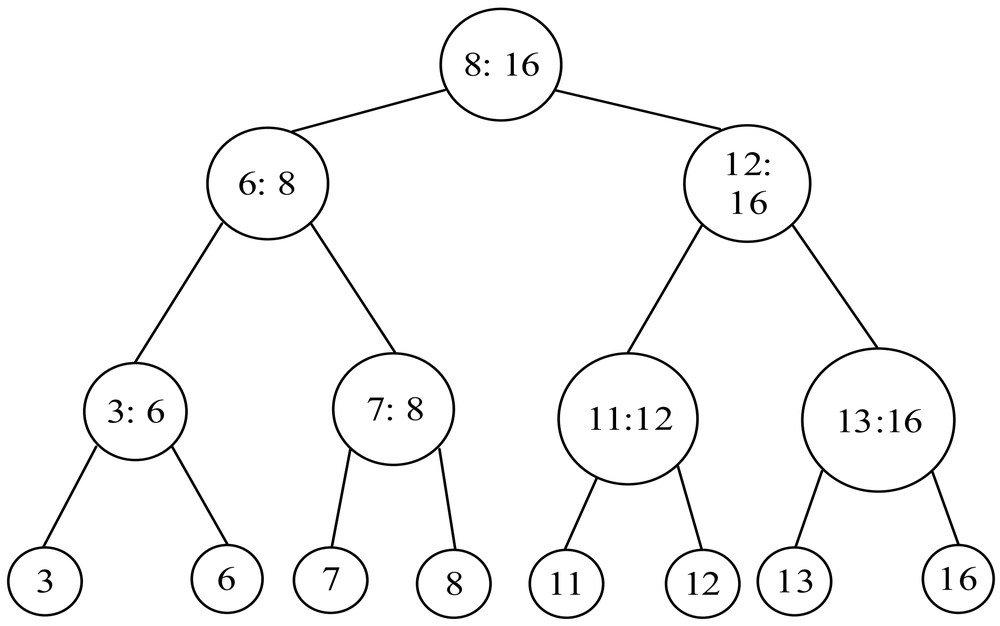


Cây 2-3: Bổ sung



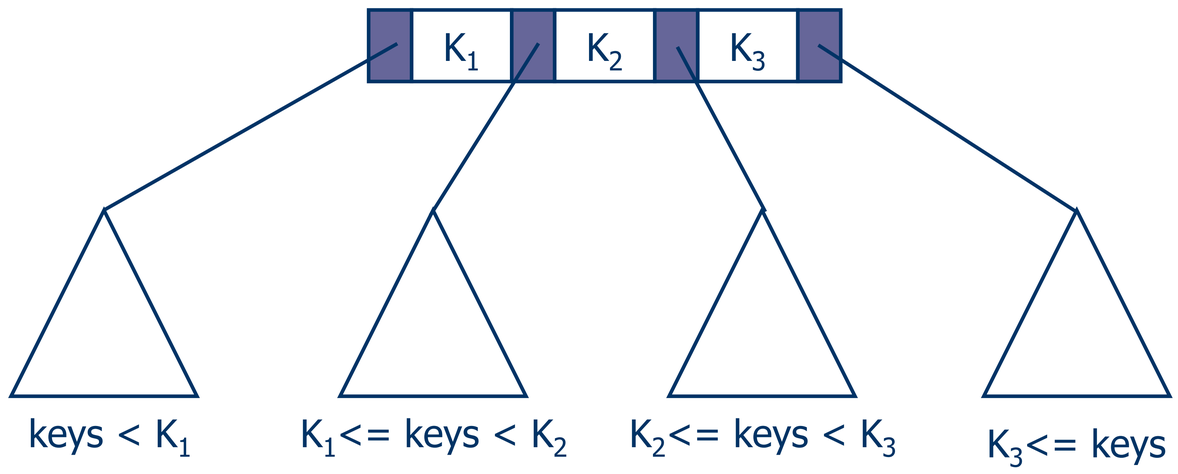
Sau khi tách nút lần 1

Cây 2-3: Bổ sung



Sau khi tách nút lần 2

Các dạng cây khác trong tìm kiếm



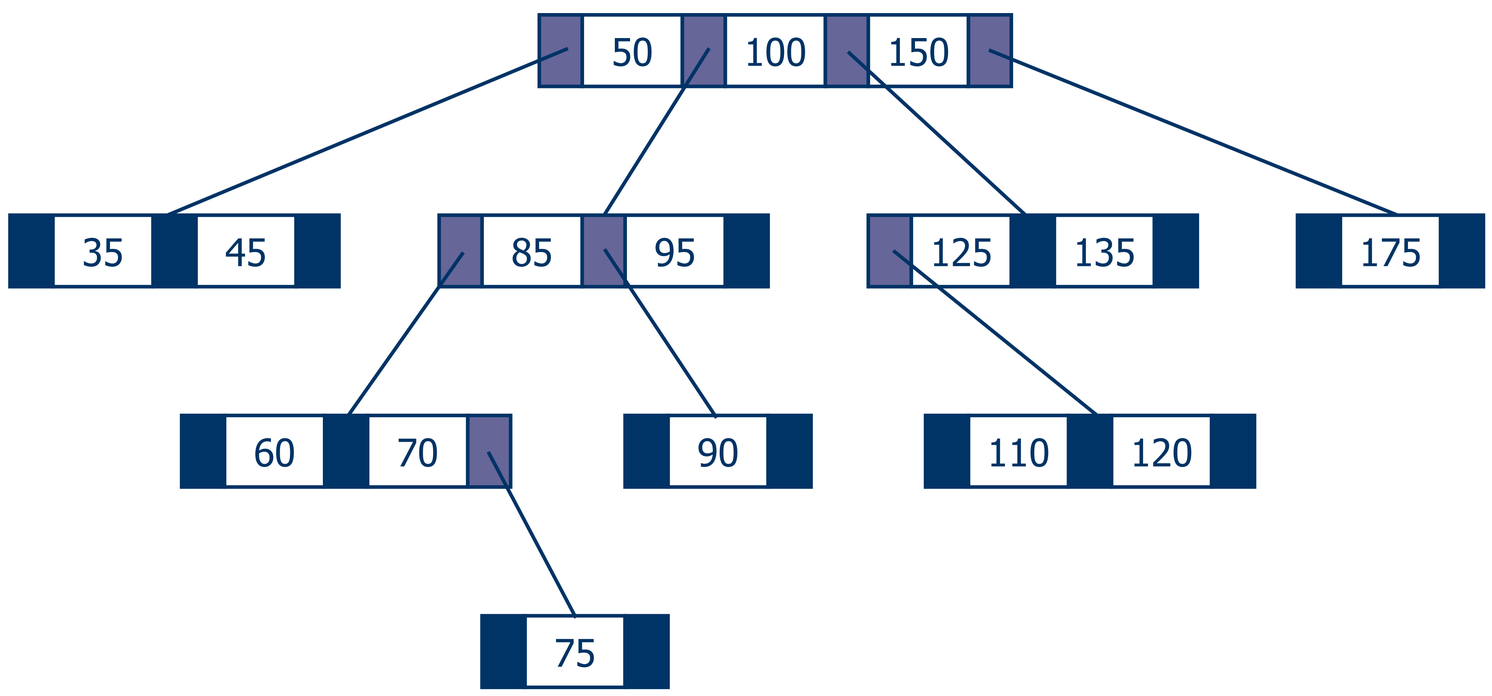
Một cây dạng cây tìm kiếm đa nhánh

Các dạng cây khác trong tìm kiếm

- Cây tìm kiếm đa nhánh(Multi-way Tree)
  - Là một cây có bậc bất kỳ nhưng có tính chất thứ tự tương tự như cây nhị phân
  - Mỗi nút trong cây có chứa m-1 khóa và m con trỏ trỏ đến các cây con
  - Các giá trị xuất hiện trong một cây con được trỏ bởi con trỏ p
    - Nhỏ hơn giá trị khóa bên phải của p
    - Lớn hơn hoặc bằng gía trị khóa bên trái p

Các dạng cây khác trong tìm kiếm

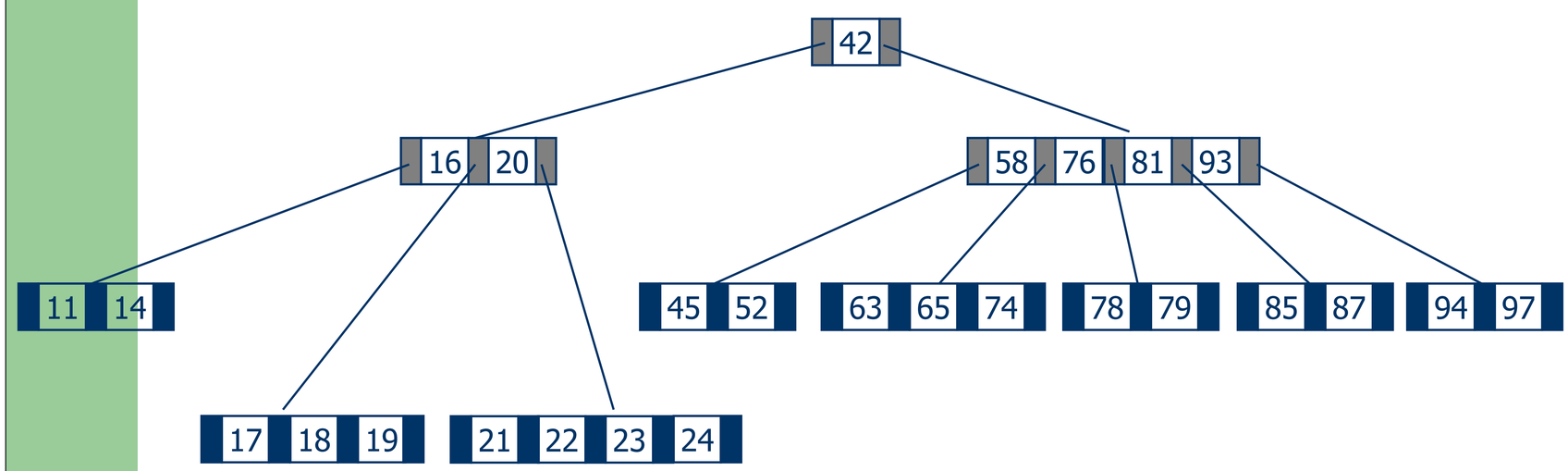
- Ví dụ cây tìm kiếm đa nhánh



Các dạng cây khác trong tìm kiếm

- Cây B – Cây tìm kiếm đa nhánh cân bằng
  - Một cây tìm kiếm đa nhánh cân bằng bậc m có các đặc trưng sau
    - Gốc của cây là một nút lá hoặc có ít nhất 2 con
    - Tất cả các nút nhánh của cây (trừ nút gốc) có từ m/2 đến m con
    - Các nút lá có từ m/2 -1 đến m-1 giá trị khóa trong đó.
    - Đường đi từ nút gốc tới một nút lá bất kỳ đều có độ dài như nhau

Các dạng cây khác trong tìm kiếm



B- Tree với m = 5

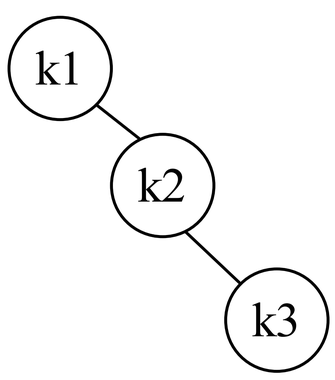
Cây nhị phân tìm kiếm tối ưu

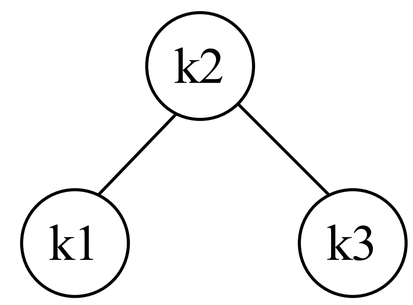
- Cây nhị phân tìm kiếm tối ưu:
  - Là cây nhị phân tìm kiếm có tính đến trường hợp các khóa khác nhau trong một tập có xác suất xuất hiện khác nhau
  - Khóa xuất hiện nhiều thì tìm nhanh hơn ⇔ đường đi từ đỉnh đến vị trí của khóa có độ dài ngắn hơn
  - Khái niệm: Giá trị của cây T

$$C(T) = \sum_{i=1}^{n} p_i * h_i$$

Cây nhị phân tìm kiếm tối ưu

- Ví dụ: Cây nhị phân tìm kiếm ứng với 3 khóa $k_1 < k_2 < k_3$ với xác suất $p_1 = 1/7$; $p_2 = 2/7$, $p_3 = 4/7$



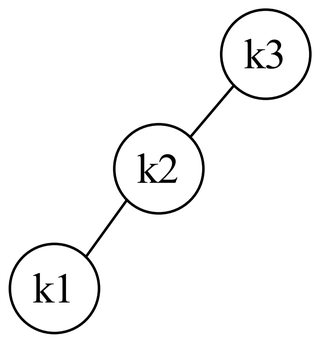


C = 1 * 1/7 + 2*2/7 + 3*4/7 = 17/7

C= 1*2/7 + 2 * 1/7 + 2*4/7 =12/7

Cây nhị phân tìm kiếm tối ưu

- Cây nhị phân tìm kiếm tối ưu: Là cây nhị phân tìm kiếm ứng với dãy khóa $k_1 < k_2 < \ldots < k_n$ có xác suất xuất hiện lần lượt là $p_1, p_2, \ldots, p_n$ mà cây đó có giá trị nhỏ nhất
- Ví dụ: Cây nhị phân tìm kiếm tối ưu ứng với 3 khóa $k_1 < k_2 < k_3$ với xác suất $p_1 = 1/7$; $p_2 = 2/7$, $p_3 = 4/7$



Cây nhị phân tìm kiếm tối ưu

- Bài toán xây dựng cây tối ưu
  - Đầu vào: Dãy khóa $k_1 < k_2 < \ldots < k_n$ có xác suất xuất hiện lần lượt là $p_1, p_2, \ldots, p_n$
  - Đầu ra: Xác định cây nhị phân tìm kiếm tối ưu xác lập được trên n nút tương ứng với n khóa đã cho

Cây nhị phân tìm kiếm tối ưu

- Nhận xét
  - Cây T là cây nhị phân tìm kiếm tối ưu gồm n khóa , k_r là gốc của cây

  →Cây $T_{1,r-1}$ gồm r-1 khóa đầu tiên, cây $T_{r+1,n}$ gồm n-r khóa cuối cùng đều phải là cây nhị phân tìm kiếm tối ưu

  →Muốn dựng được T, cần phải dựng từ hai cây con của nó

Cây nhị phân tìm kiếm tối ưu

- Tính giá trị của một cây $T_{i,j}$ dựa vào giá trị các cây con
- $T_{i,j}$ là cây tạo dựng được từ các khóa $k_i < k_{i+1} < \ldots < k_j$

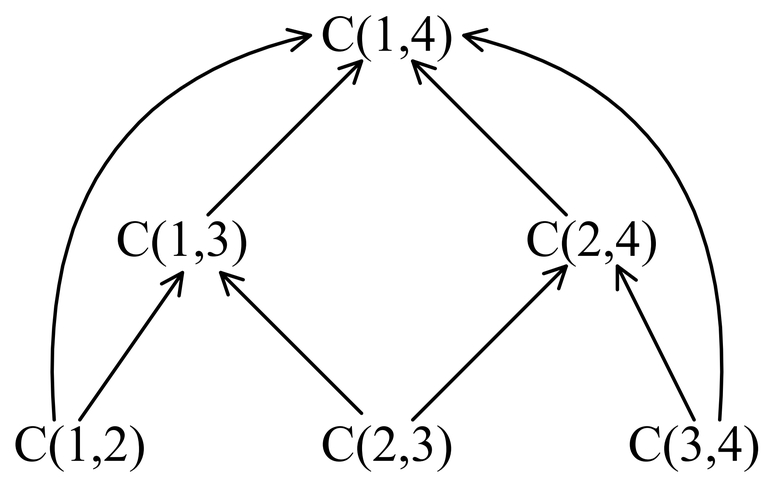
$$C_{i,j} = p_{i,j} + \min[(C_{i,r-1} + C_{r+1,j})] \; (i \le r \le j)$$

$$p_{i,j} = \sum_{k=i}^{j} p_k$$

- r trong công thức cho ta xác định được khóa nào là gốc của cây

Cây nhị phân tìm kiếm tối ưu

- Xác định cây tối ưu với 4 nút, cần phải thực hiện tính toán các giá trị theo sơ đồ sau



Cây nhị phân tìm kiếm tối ưu

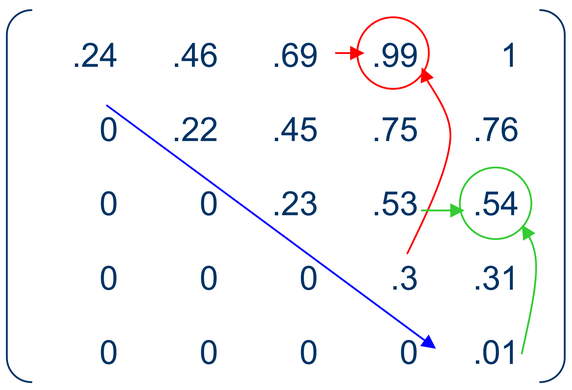
- Ví dụ: Dãy bao gồm 5 khóa, với xác suất như sau

i	1	2	3	4	5
p_i	0.24	0.22	0.23	0.3	0.01

Cây nhị phân tìm kiếm tối ưu

- Các giá trị $p_{i,j}$ (i<= j) được xác định và thể hiện trong ma trận sau

P[i,j]=



Cây nhị phân tìm kiếm tối ưu

- Kết quả các giá trị của các cây tối ưu

C[i,j]=

.24	.68	1.16	1.99	2
0	.22	.67	1.27	1.3
0	0	.23	.76	.78
0	0	0	.3	.32
0	0	0	0	.01

Cây nhị phân tìm kiếm tối ưu

P[i,j]=

.24	.46	.69	.99	1
0	.22	.45	.75	.76
0	0	.23	.53	.54
0	0	0	.3	.31
0	0	0	0	.01

C[i,j]=

.24	.68			
0	.22	.67		
0	0	.23		
0	0	0	.3	.32
0	0	0	0	.01

C[1,2]=P[1,2]+min { r=1, C[2,2] ← .22 ; r=2, C[1,1] ← .24 } r=1 → 1, 2

C[2,3]=P[2,3]+min { r=2, C[3,3] ← .23 ; r=3, C[2,2] ← .22 } r=3 → 3, 2

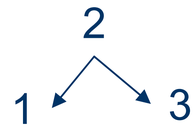
……

C[4,5]=P[4,5]+min { r=4, C[5,5] ← .01 ; r=5, C[4,4] ← .3 } r=4 → 4, 5

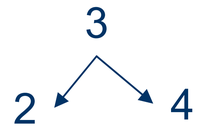
Cây nhị phân tìm kiếm tối ưu

$$P[i,j]=\begin{pmatrix} .24 & .46 & .69 & .99 & 1 \\ 0 & .22 & .45 & .75 & .76 \\ 0 & 0 & .23 & .53 & .54 \\ 0 & 0 & 0 & .3 & .31 \\ 0 & 0 & 0 & 0 & .01 \end{pmatrix} \quad C[i,j]=\begin{pmatrix} .24 & .68 & 1.16 & & \\ 0 & .22 & .67 & 1.27 & \\ 0 & 0 & .23 & .76 & \\ 0 & 0 & 0 & .3 & .32 \\ 0 & 0 & 0 & 0 & .01 \end{pmatrix}$$

$C[1,3]=P[1,3]+\min$ { r=1, C[2,3] ←.67; r=2, C[1,1]+C[3,3] ←.47; r=3, C[1,2] ←.68 } r=2



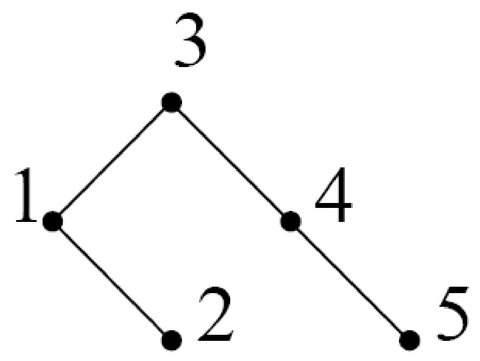
$C[2,4]=P[2,4]+\min$ { r=2, C[3,4] ←.76; r=3, C[2,2]+C[4,4] ←.52; r=4, C[2,3] ←.67 } r=3



Cây nhị phân tìm kiếm tối ưu

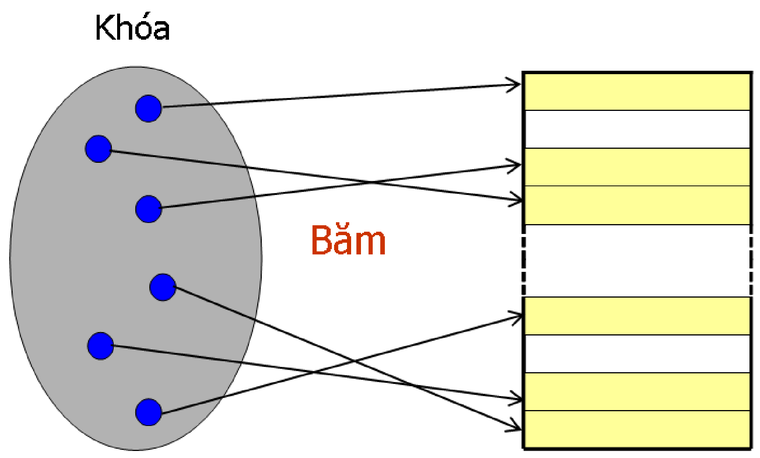
$C[1,5]=P[1,5]+\min$ { r=1, C[2,5] ←1.3; r=2, C[1,1]+C[3,5] ← 1.02; r=3, C[1,2]+C[4,5] ←1; r=4, C[1,3]+C[5,5] ← 1.17; r=5, C[1,4] ←1.99 } r=3

Cây kết quả



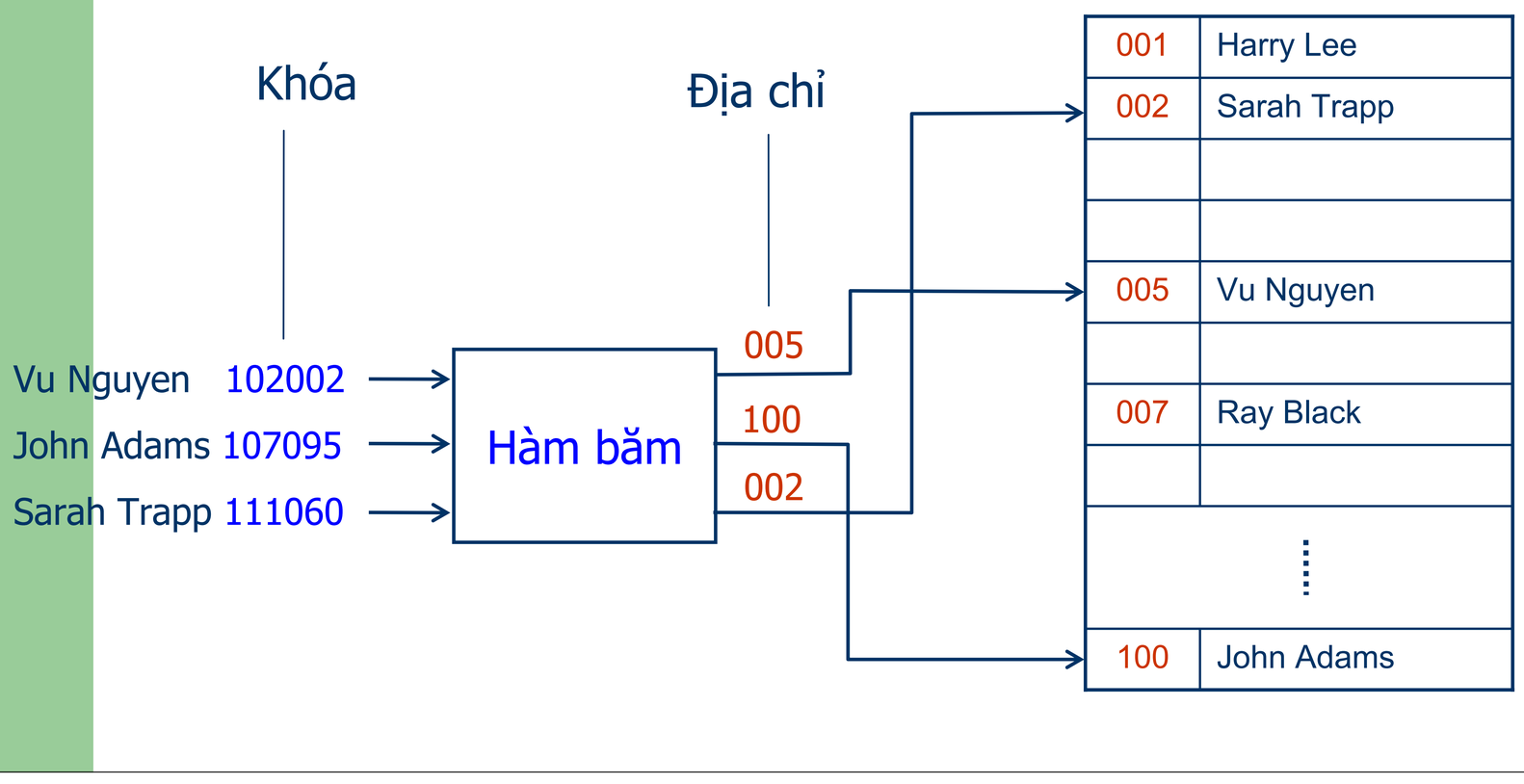
Tìm kiếm dựa trên bảng băm

- Tìm kiếm không dựa trên so sánh giá trị khóa mà dựa vào bản thân giá trị khóa
- Sử dụng một qui tắc biến đổi tham chiếu một giá trị khóa sang một địa chỉ (tương đối) lưu trữ phần tử dữ liệu



Mỗi khóa chỉ có duy nhất 1 địa chỉ

Tìm kiếm dựa trên bảng băm



Tìm kiếm dựa trên bảng băm

- Hàm băm h(k) thường có độ phức tạp là O(1)
  - h: U → {0, 1, .., m-1}
  - Một phần tử có khóa k sẽ được tham chiếu vào một ô được đánh chỉ số là h(k) có giá trị từ 0-> m-1 trong bảng băm kích thước m
  - Khi sử dụng bảng băm để tìm kiếm, thay vì quan tâm đến |U| giá trị, chúng ta chỉ quan tâm đến m ô trong bảng
- Đụng độ
  - Hiện tượng xảy ra khi hai hay nhiều khóa khác nhau sau khi băm cho cùng một giá trị địa chỉ tương đối
  - Hai phương pháp giải quyết đụng độ
    - Phương pháp móc xích
    - Phương pháp địa chỉ mở

Hàm băm

- Hàm băm tốt là một hàm băm đơn giản và có thể tính toán được trong thời gian ngắn
- Mục tiêu của hàm băm là phân bổ một tập các giá trị khóa một cách ngẫu nhiên và đều trên một tập các ô trong bảng

Hàm băm

- h(k) = k mod m
  - m = 12 and k=100 → h(k) = 4
  - Cần phải tránh sử dung một số giá trị cho m
    - m không nên là một số dạng 2^p
  - Thông thường, m được chọn là một số nguyên tố không quá gần với một giá trị 2^P
  - Ví dụ: n=2000, ta chấp nhận kiểm tra 3 phần tử khi thực hiện việc tìm kiếm, ta có thể chọn m = 701

    vì 701 là một số nguyên tố gần với 2000/3

    h(k)=k mod 701

Hàm băm

- h(k) = ⌊m*(k*A mod 1)⌋
  - A là một giá trị nằm trong khoảng 0-1. Theo Knuth đề xuất

    $$A \approx \left(\sqrt{5}-1\right)/2 = 0.6180339887...$$
  - Nhân k với A, lấy phần sau dấu phẩy
  - Nhân phần sau dấu phẩy đó với m , rồi lấy phần nguyên

- Ví dụ :k=123456,m=10000,A=0.618

  h(k)=floor(10000*(123456*0.618… mod 1))
  =floor(10000*(76300.004151..mod 1))
  =floor(10000*0.0041151…..)=41.

Hàm băm

- h(k) = số tạo bởi một số chữ số ở giữa của bình phương của khóa
- Ví dụ: k = 9452
  - 9452 * 9452 = 89340304 → 3403
- Nếu khóa lớn, có thể chỉ dùng một phần của khóa khi tính bình phương

  379452: 379 * 379 = 143641 → 364
  121267: 121 * 121 = 014641 → 464
  045128: 045 * 045 = 002025 → 202

Hàm băm

- Sử dụng phương pháp phân đoạn
  - Khóa được chia thành nhiều đoạn, thường có độ dài bằng độ dài địa chỉ
  - Áp dụng một số kỹ thuật trên các đoạn để xác định địa chỉ
    - Ví dụ: Khóa = 123|456|789

kỹ thuật tách

123 + 456 + 789 = 1368
⇒ 368

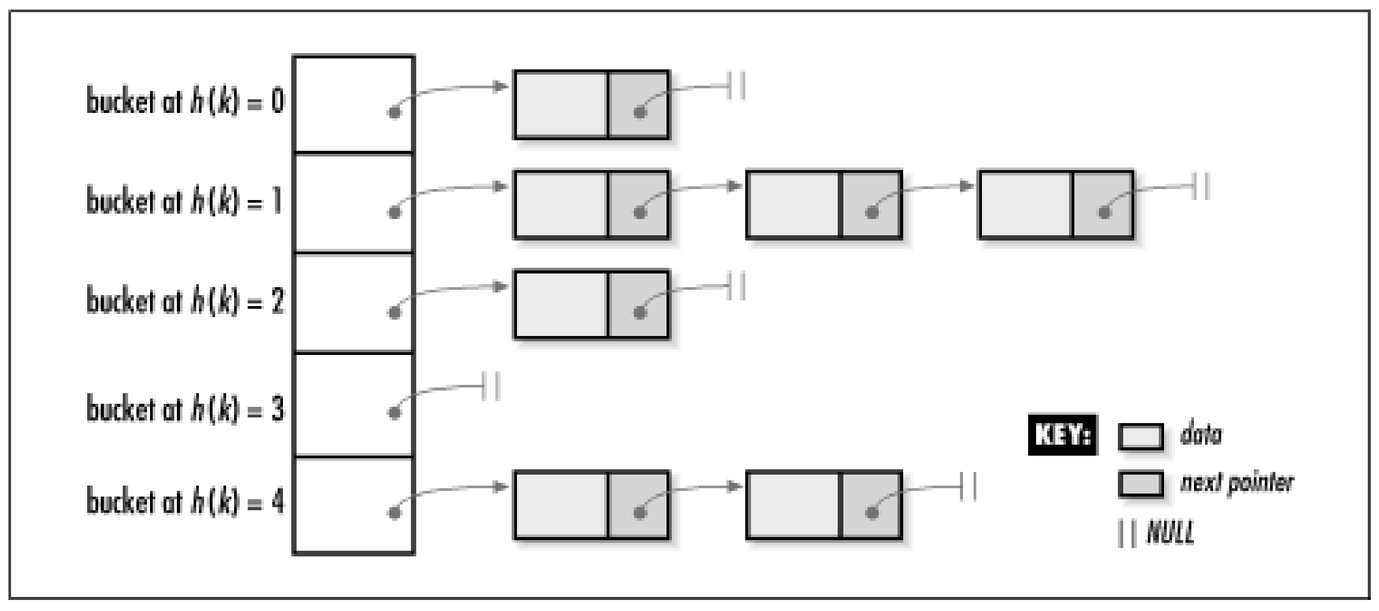
kỹ thuật gấp

321 + 456 + 987 = 1764
⇒ 764

Giải quyết đụng độ

- Các kỹ thuật giải quyết đụng độ
  - Phương pháp móc xích: Mỗi phần tử trong bảng băm là một danh sách móc nối chứa các phần tử
  - Phương pháp địa chỉ mở : Tìm trong bảng băm theo một qui tắc nào đó để xác định một ô trống lưu phần tử mới nếu có đụng độ xảy ra
    - Thử tuyến tính
    - Băm lại

Giải quyết đụng độ -Phương pháp móc xích

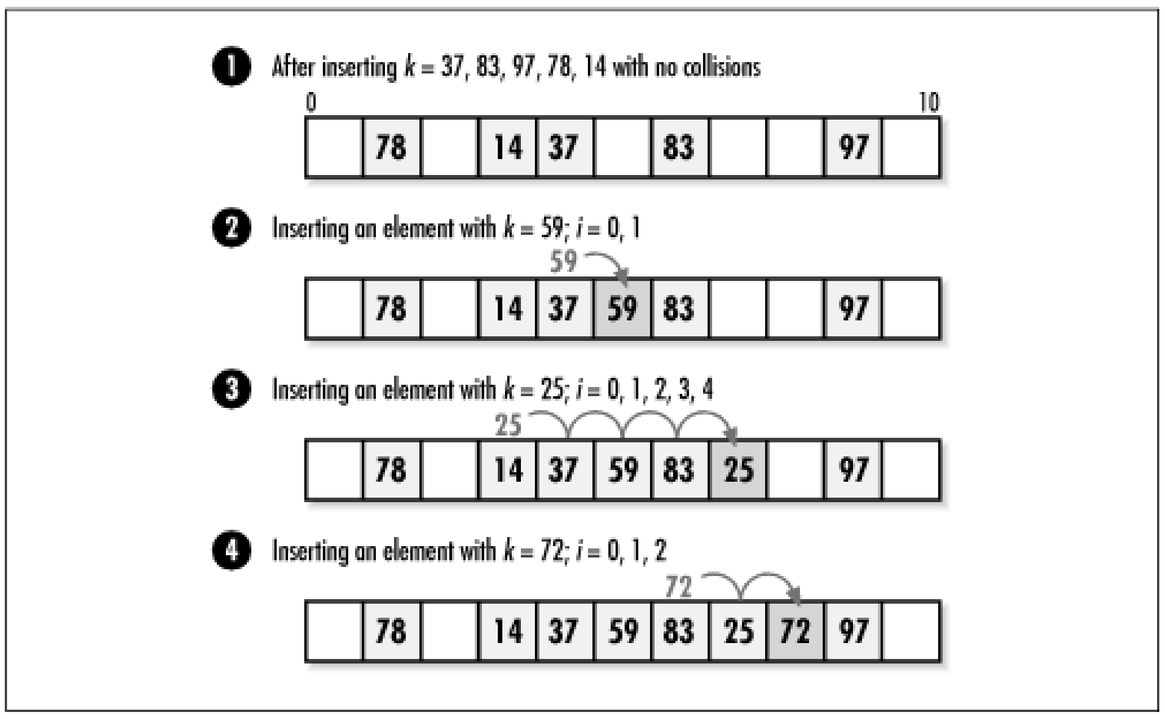


Giải quyết đụng độ - Phương pháp địa chỉ mở

- Thử tuyến tính (Linear Probing)
  - Nếu có một bản ghi mà địa chỉ băm tương ứng với giá trị khóa đã bị chiếm, tìm một ví trí trống gần nhất để lưu trữ nó bằng cách duyệt tuần tự các vị trí kế tiếp
  - H(k,i) = (h'(k) + i) mod m
    - i nhận giá trị từ 0 đến m-1 , thể hiện số lần phải dịch chuyển để xác định vị trí kế tiếp

Giải quyết đụng độ - Phương pháp địa chỉ mở

- h(k, i) = (k mod 11 + i) mod 11



Giải quyết đụng độ - Phương pháp địa chỉ mở

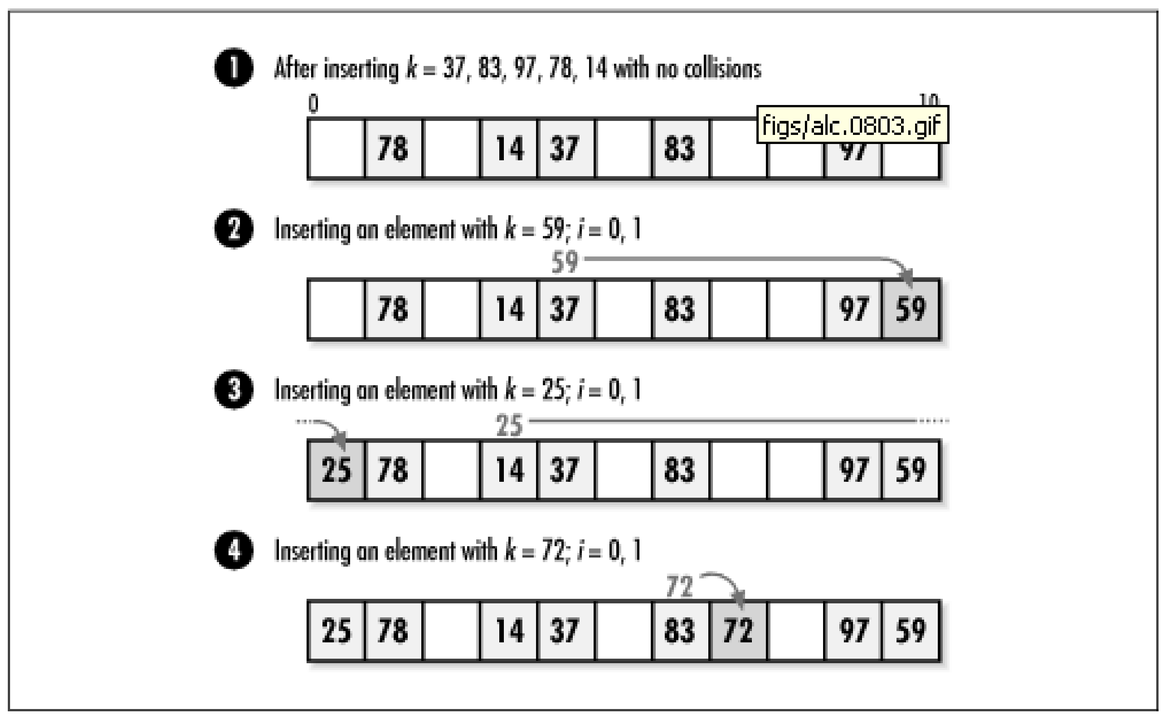
- Băm lại (Double Hashing)

$h(k,i) = (h_1(k) + i*h_2(k)) \bmod m \ (i = 0, 1, \ldots, m-1)$

- Thông thường m được chọn là số nguyên tố
- $h_1 = k \bmod m$; $h_2 = 1 + k \bmod m'$ với $m' < m$ và m' rất gần với m

Giải quyết đụng độ - Phương pháp địa chỉ mở

- $h(k,i) = (k \bmod 11 + i\,(1+k \bmod 9)) \bmod 11$



Tìm kiếm xâu mẫu

- Bài toán
  - Xâu
    - Một dãy ký tự lấy từ một bảng chữ cái
    - Ký hiệu T[i,j] là một xâu con của T bắt đầu từ vị trí i kết thúc tại vị trí j
  - Thao tác đẩy
    - Cho 2 xâu T_1, T_2 có độ dài m, n với m <= n
    - T_1 xuất hiện nhờ đẩy đến vị trí s trong T_2 nếu $T_1[0, m] = T_2[s, s+m]$
  - Vị trí khớp
    - Cho 2 xâu T_1, T_2 có độ dài m, n với m <= n
    - Nếu T_1 xuất hiện nhờ đẩy đến vị trí s trong T_2 thì s được gọi là vị trí khớp của T_1 trong T_2

Tìm kiếm xâu mẫu

- Bài toán
  - Tìm kiếm xâu mẫu là tìm tất cả các vị trí khớp của một xâu mẫu P trong một văn bản T
    - P có độ dài m, T có độ dài n
    - T: “the rain in spain stays mainly on the plain”
    - P: “ain ”
- Ứng dụng
  - Trong tìm kiếm thông tin
  - Soạn thảo văn bản
  - Xử lý dữ liệu sinh học (ADN)

Giải thuật đơn giản (Brute-Force alg.)

- Ý tưởng:
  - So khớp mẫu với văn bản bằng cách so khớp lần lượt từng ký tự trong mẫu từ trái sang phải
  - Khi có một vị trí không khớp được xác định, đẩy toàn bộ mẫu sang phải 1 vị trí và tiếp tục so khớp với văn bản theo cách thức trên
- Độ phức tạp của giải thuật trong trường hợp xấu nhất O(m*n)

```
Algorithm Naive(T, P)
    for  s = 0 to n-m do
    begin
        j = 0;
        while ( j< m && T[s+j] = P[j]) do j++;
        if j = m then output(s) ;
    end
```

Giải thuật đơn giản (Brute-Force alg.)

T	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	
P	0	0	0	1												
		0	0	0	1											Khớp
			0	0	0	1										
				0	0	0	1									
					0	0	0	1								Khớp
							...									
								...								
										0	0	0	1			
											0	0	0	1		Khớp

Giải thuật đơn giản

- Trường hợp tồi nhất
  - T = aaaaaaaaa.....aaaab (n ký tự)
  - P = aa..aab (m-1 ký tự a và 1 ký tự b)
  - Vị trí không khớp luôn được xác định sau m lần so khớp
  - Sự không khớp xảy ra n-m lần
  - Vị trí khớp xác định tại n-m+1
  - Số các phép so sánh m*(n-m+1)

Giải thuật Knuth-Morris-Pratt (KMP)

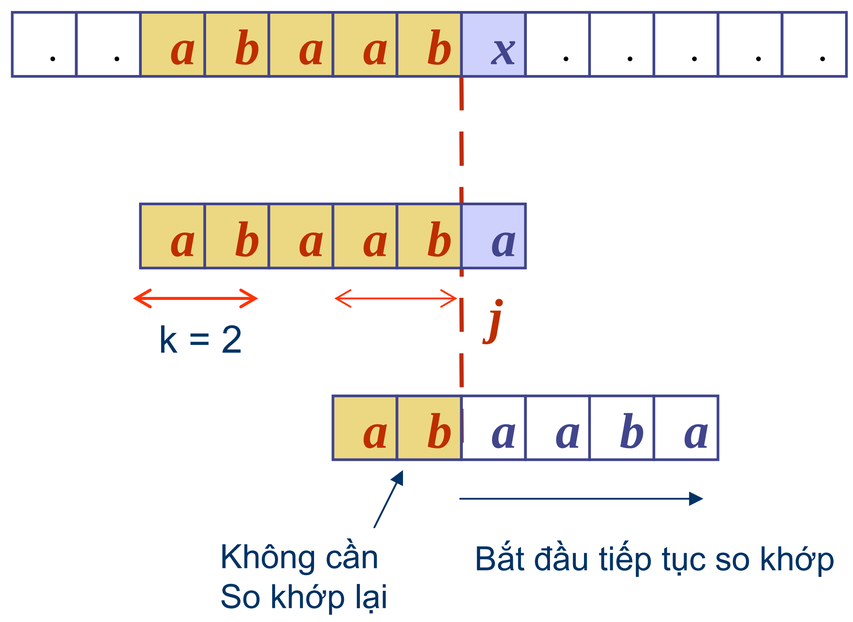
- Giải thuật KMP so khớp mẫu với văn bản từ trái sang phải theo cơ chế giống giải thuật đơn giản
- Giải thuật KMP xác định phép đẩy thông minh hơn giải thuật cơ bản.

Giải thuật Knuth-Morris-Pratt (KMP)

- Giả sử có xâu P có độ dài m
  - Một xâu con P[i..j] của P là một phần trong P chứa các ký tự trong khoảng vị trí từ i đến j
  - Một prefix của P là một xâu con có dạng P[0..i]
  - Một suffix của P là một xâu con có dạng P[i..m-1]
  - Ví dụ: P = abacade
    - P’ = aba là một prefix của P
    - P’’ = ade là một suffix của P

Giải thuật Knuth-Morris-Pratt (KMP)

- Khi có sự không khớp xảy ra tại vị trí j, xem xét xâu p’ gồm j-1 ký tự đã khớp trước
  - Nếu có một prefix có độ dài k trong p’ cũng xuất hiện là suffix thực sự trong p’ thì đẩy để khớp vị trí j trong văn bản với vị trí k trong mẫu
  - Nếu không tồn tại prefix có tính chất trên, đẩy để khớp vị trí j trong văn bản với vị trí 0 trong mẫu



Giải thuật Knuth-Morris-Pratt (KMP)

- Khi thực hiện KMP cần tiền xử lý xâu mẫu P để xác định sự xuất hiện của các prefix trong mẫu
- Xác định hàm Prefix - p(j) : là hàm xác định độ dài của prefix dài nhất của P[0..j] mà prefix này đồng thời là suffix của P[1..j]
- Giải thuật KMP dựa trên giải thuật cơ bản , tuy nhiên nếu có một vị trí không khớp được xác định tại P[j] thì sẽ đẩy một khoảng $p(j-1)$

Giải thuật Knuth-Morris-Pratt (KMP)

Algorithm ***prefixFunction***(***P***)

{ m là độ dài của P}

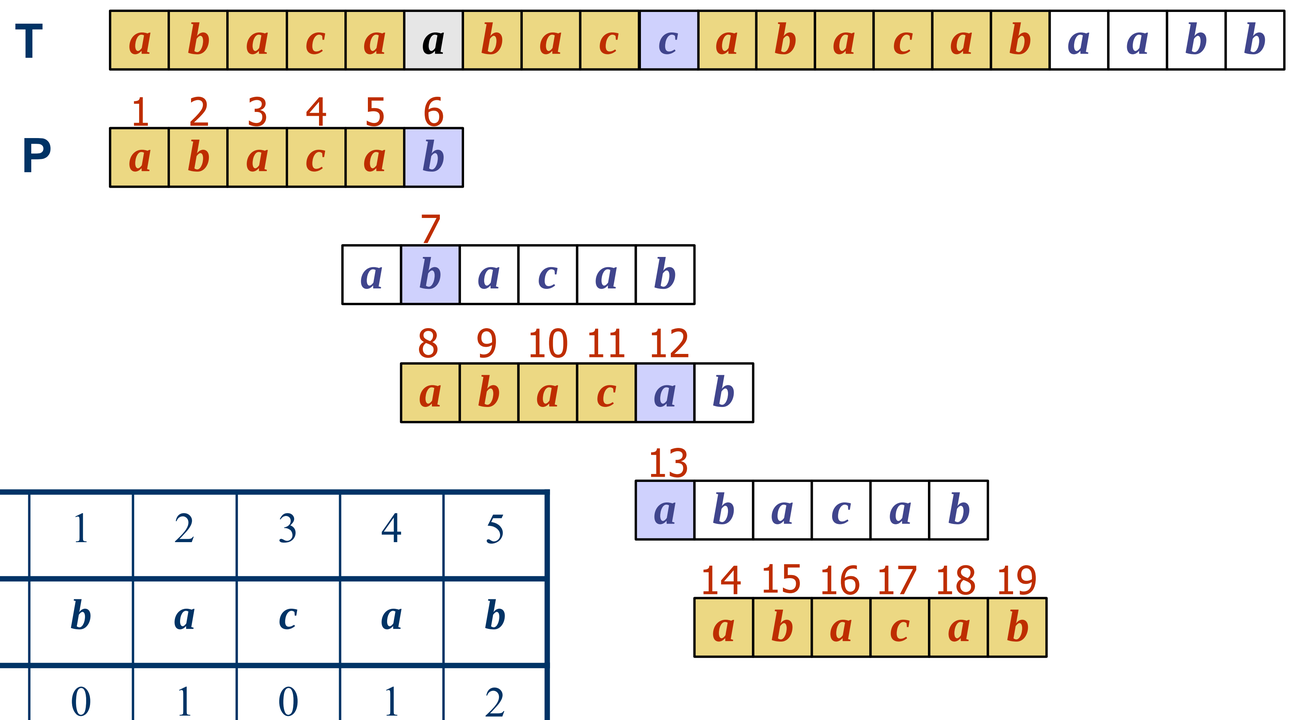
```
p[0] = 0; i = 1; j = 0;
while i < m do
begin
    while ( j > 0 && P[i] != P[j]) do
            j = p[j];
    if (P[i] = P[j]) then  j = j + 1;
    p[i] = j;
    i = i + 1;
end.
```

i	0	1	2	3	4	5
***P*[*i*]**	***a***	***b***	***a***	***a***	***b***	***a***
***p*(*i*)**	0	0	1	1	2	3

Giải thuật Knuth-Morris-Pratt (KMP)

```
Algorithm KMP(T, P)
    p ← prefixFunction(P);
    j = 0; i = 0 ;
    while ( i < n)  do
    begin
        while ( j >= 0 && T[i] != P[j]) do
                j = p[j];
        if (T[i] = P[j]) then  j = j + 1;
        if (j = m ) then output(i-m);
        i = i + 1;
    end.
```

Giải thuật Knuth-Morris-Pratt (KMP)



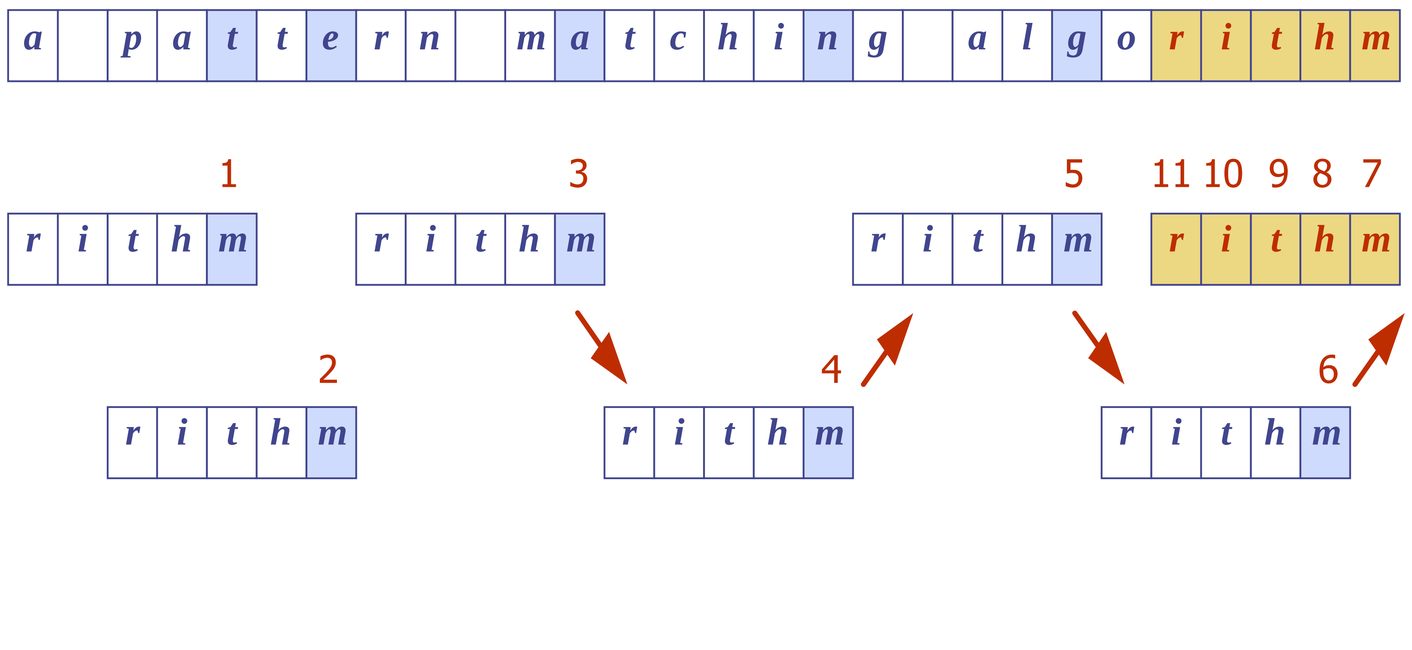
j	0	1	2	3	4	5
P[*j*]	*a*	*b*	*a*	*c*	*a*	*b*
p(*j*)	0	0	1	0	1	2

Giải thuật Boyer-Moore

- Giải thuật Boyer-Moore's thực hiện dựa trên hai điểm sau:
  - So sánh P với một xâu con của T từ phải sang trái
  - Khi có một ký tự T[i] = c không khớp với P[j] có thể xác định một bước đẩy sang phải dựa trên hai kỹ thuật
    - Sử dụng ký tự tồi (Bad- character heuristic)
    - Sử dụng hậu tố tốt (Good suffix heuristic)

Giải thuật Boyer-Moore

Ví dụ: Giải thuật Boyer-Moore dựa trên kỹ thuật sử dụng ký tự tồi



Giải thuật Boyer-Moore

- Khi có một vị trí không khớp xác định tại $T[i] = c$ (khác với P[j])
  - Nếu P chứa ký tự c, đẩy P sao cho vị trí xuất hiện cuối cùng của c trong P khớp với T[i]
  - Nếu không , đẩy cho $P[0]$ khớp với $T[i + 1]$

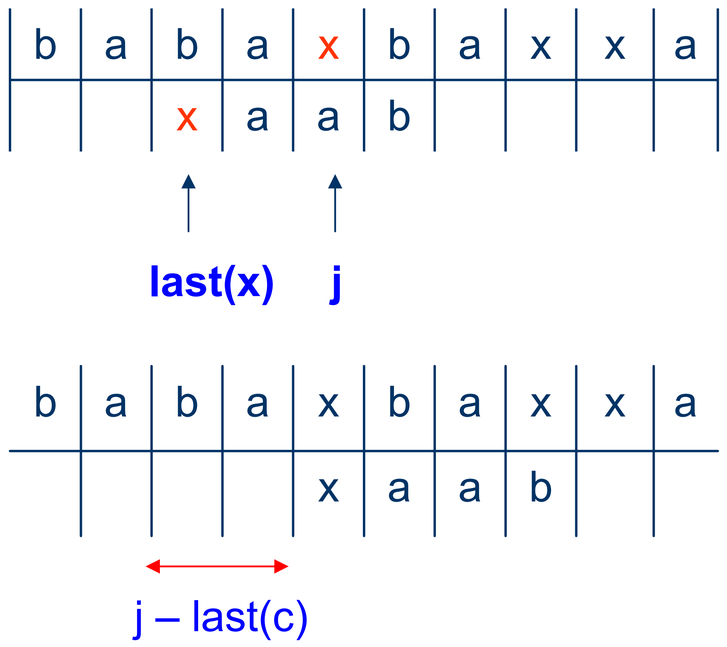
Giải thuật Boyer-Moore
Bad- character heuristic

- Tiền xử lý - Xác định hàm Last-Occurrence (Vị trí cuối cùng)
  - Tiền xử lý mẫu P và bảng chữ cái A để xác định hàm last là hàm ánh xạ từ văn bản A sang một tập các số nguyên.
  - Với từng ký tự c trong bảng chữ cái A
    - last (c) = chỉ số lớn nhất i (0 <= i <= m) sao cho P[i] = c
    - last (c) = -1 nếu c không xuất hiện trong P
  - Ví dụ: A= {**a, b, c, d**} ; **P** = **acabab**

c	***a***	***b***	***c***	***d***
***last*(*c*)**	4	5	1	–1

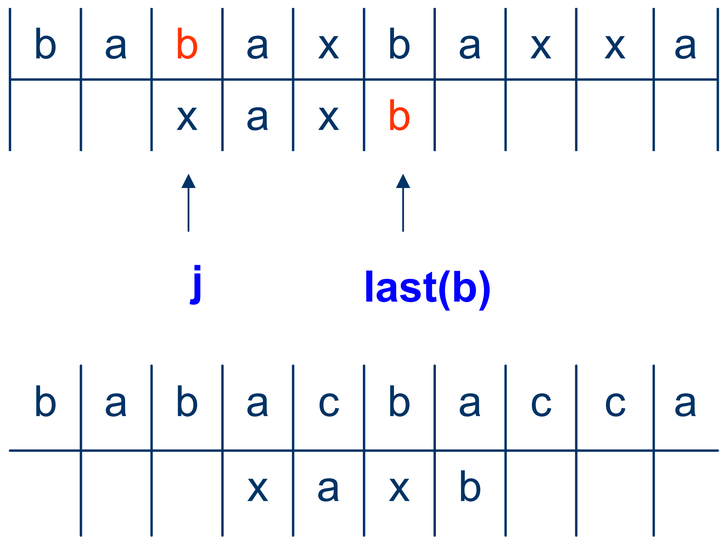
Giải thuật Boyer-Moore

- Xác định vị trí để đẩy
  - **Trường hợp 1**: T[i] = c là ký tự không khớp, c xuất hiện trong P và last(c) < j
  - Khi đó đẩy một bước s = s + (j- last(c))



Giải thuật Boyer-Moore

- Xác định vị trí để đẩy
  - **Trường hợp 2**: T[i] = c là ký tự không khớp, c xuất hiện trong P và last(c) > j
  - Khi đó đẩy một bước s = s + 1



Giải thuật Boyer-Moore

- Xác định vị trí để đẩy
  - **Trường hợp 3**: T[i] = c là ký tự không khớp, c không xuất hiện trong P; last(c) = -1
  - Khi đó đẩy sao cho P[0] trùng với T[i+1]

    s= s + (j – last(c))

b	a	b	a	x	b	a	x	x	a
		x	a	x	a				

j

b	a	b	a	c	b	a	c	c	a
						x	a	x	a

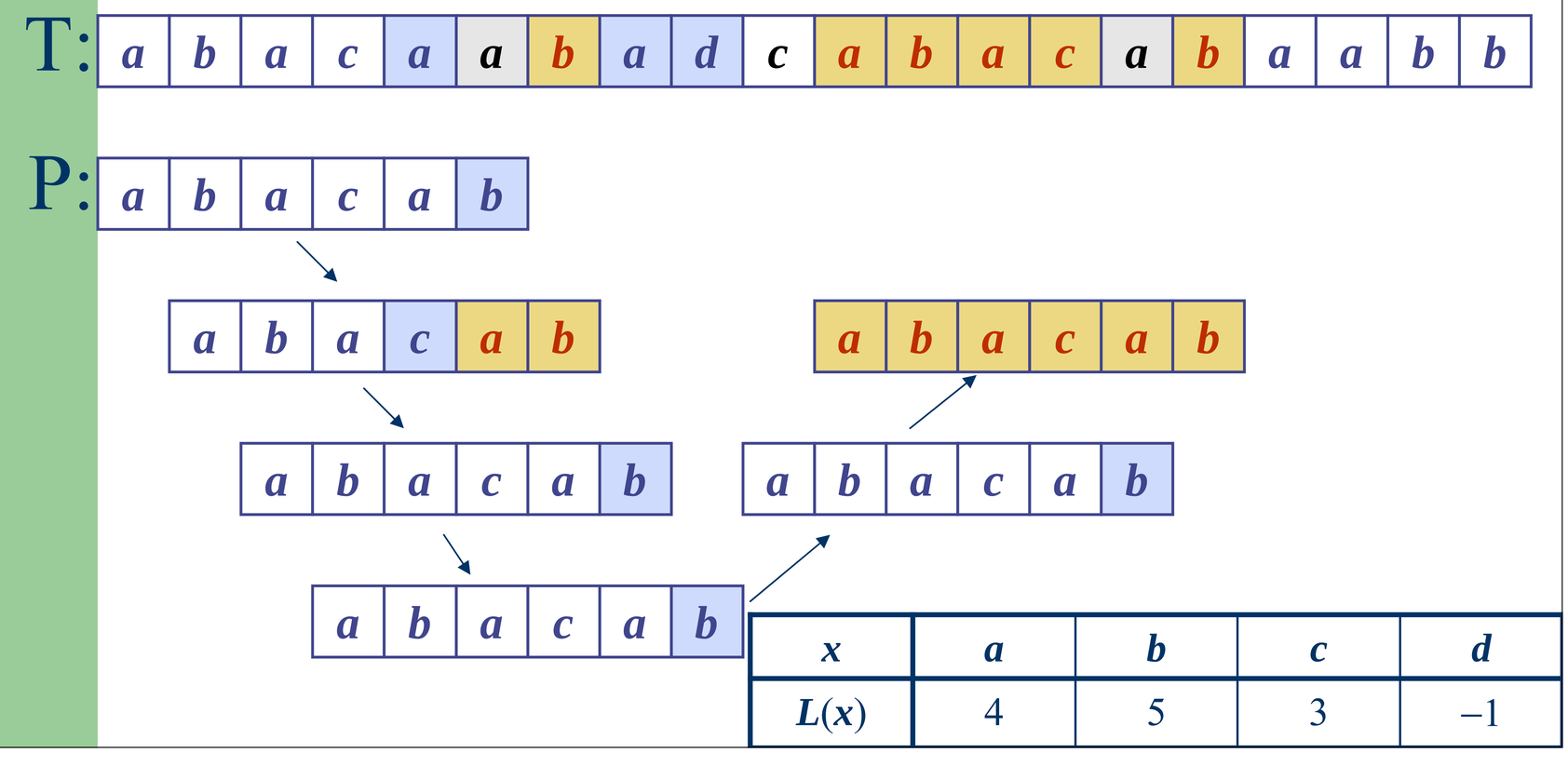
Giải thuật Boyer-Moore

Algorithm ***BoyerMooreMatch***(***T, P, A***)

```
L = lastOccurence(P, A)
s = 0 ;
while s <= n- m do
begin
    j = m-1;
    while j >= 0 and T[j+s] = P[j] do j = j – 1;
    if ( j = -1 ) then begin output(s); s = s + 1; end;
    else begin
        k = L[T[j+s]];
        s = s + max(j-k, 1);
    end;

end.
```

Giải thuật Boyer-Moore



x	a	b	c	d
L(x)	4	5	3	–1

Trường hợp xấu nhất

- T: "aaaaa…a"
- P: "baaaaa“
- Trong trường hợp xấu nhất giải thuật có độ phức tạp là O(m*n + |A|)