CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

TÌM KIẾM

Nội dung

- Giới thiệu bài toán tìm kiếm
- Tìm kiếm tuần tự
- Tìm kiếm nhị phân
- Cây nhị phân tìm kiếm
- Cây nhị phân tìm kiếm cân bằng
- Tìm kiếm xâu mẫu
- Ánh xạ và bảng băm

GIỚI THIỆU BÀI TOÁN TÌM KIẾM

- Cần tìm kiếm 1 phần tử nào đó trong một tập dữ liệu
- Bài toán tìm kiếm xuất hiện rất phổ biến trong các bài toán tính toán cũng như các phần mềm ứng dụng
- Tập dữ liệu cần được lưu trữ một cách có cấu trúc để việc tìm kiếm được nhanh chóng và hiệu quả

TÌM KIẾM TUẦN TỰ

- Tập dữ liệu được lưu trữ một cách tuyến tính và không có thông tin gì thêm
- Duyệt lần lượt các phần tử của tập dữ liệu và so sánh với khoá đầu vào

```
sequentialSearch(X[], int L, int R,
    int Y) {
  for(i = L; i <= R; i++)
     if(X[i] = Y) return i;
  return -1;
}
```

TÌM KIẾM NHỊ PHÂN

- Tập dữ liệu được lưu trữ một cách tuyến tính theo thứ tự không giảm của khoá
- Chia để trị
  - Chia dãy cần tìm thành 2 nửa bằng nhau
  - So sánh khoá đầu vào với phần tử ở giữa và quyết định tiếp tục tìm kiếm nửa bên trái hoặc nửa bên phải tuỳ thuộc vào kết quả so sánh

```
binarySearch(X[], int L, int R,
    int Y) {
  if(L = R){
    if(X[L] = Y) return L;
    return -1;
  }
  int mid = (L+R)/2;
  if(X[mid] = Y) return mid;
  if(X[mid] < Y)
   return binarySearch(X,mid+1,R,Y);
  return binarySearch(X,L,mid-1,Y);
}
```

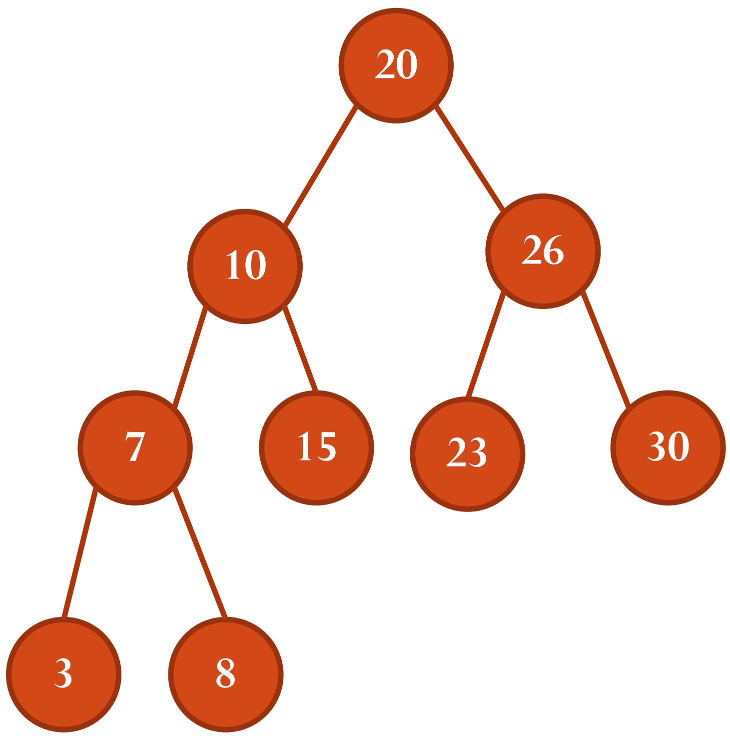
TÌM KIẾM NHỊ PHÂN

- Ví dụ ứng dụng: Cho dãy a_1, a_2, …, a_N (các phần tử đôi một khác nhau) và 1 giá trị b. Hãy đếm xem trong dãy có bao nhiêu cặp (a_i, a_j) sao cho $a_i + a_j = b$ ($i < j$).

CÂY NHỊ PHÂN TÌM KIẾM
Binary Search Tree - BST

- Cấu trúc dữ liệu lưu trữ các đối tượng dưới dạng cây nhị phân
  - Khoá của mỗi nút lớn hơn khoá của các nút của cây con trái và nhỏ hơn hoặc bằng khoá của các nút của cây con phải

```
struct Node{
  int key;
  Node* leftChild;
  Node* rightChild;
};
Node* root;
```



CÂY NHỊ PHÂN TÌM KIẾM

- Các thao tác
  - `Node* makeNode(int v)`: tạo ra một nút có khoá v và trả về con trỏ trỏ đến nút tạo được
  - `Node* insert(Node* r, int v)`: tạo một nút mới có khoá v và chèn vào BST có gốc là r
  - `Node* search(Node* r, int v)`: tìm và trả về con trỏ trỏ đến nút có khoá là v trên BST có gốc r
  - `Node* findMin(Node* r)`: trả về con trỏ trỏ đến nút có khoá nhỏ nhất
  - `Node* del(Node* r, int v)`: loại bỏ nút có khoá v ra khỏi cây nhị phân tìm kiếm gốc được trỏ bởi r

CÂY NHỊ PHÂN TÌM KIẾM

```
Node* makeNode(int v) {
  Node* p = new Node;
  p->key = v;
  p->leftChild = NULL;
  p->rightChild = NULL;
  return p;
}
```

CÂY NHỊ PHÂN TÌM KIẾM

```
Node* insert(Node* r, int v) {
  if(r == NULL)
    r = makeNode(v);
  else if(r->key > v)
    r->leftChild = insert(r->leftChild,v);
  else if(r->key <= v)
    r->rightChild = insert(r->rightChild,v);
  return r;
}
```

CÂY NHỊ PHÂN TÌM KIẾM

```
Node* search(Node* r, int v) {
  if(r == NULL)
    return NULL;
  if(r->key == v)
    return r;
  else if(r->key > v)
    return search(r->leftChild, v);
  return search(r->rightChild, v);
}
```

CÂY NHỊ PHÂN TÌM KIẾM

```
Node* findMin(Node* r) {
  if(r == NULL)
    return NULL;
  Node* lmin = findMin(r->leftChild);
  if(lmin != NULL) return lmin;
  return r;
}
```

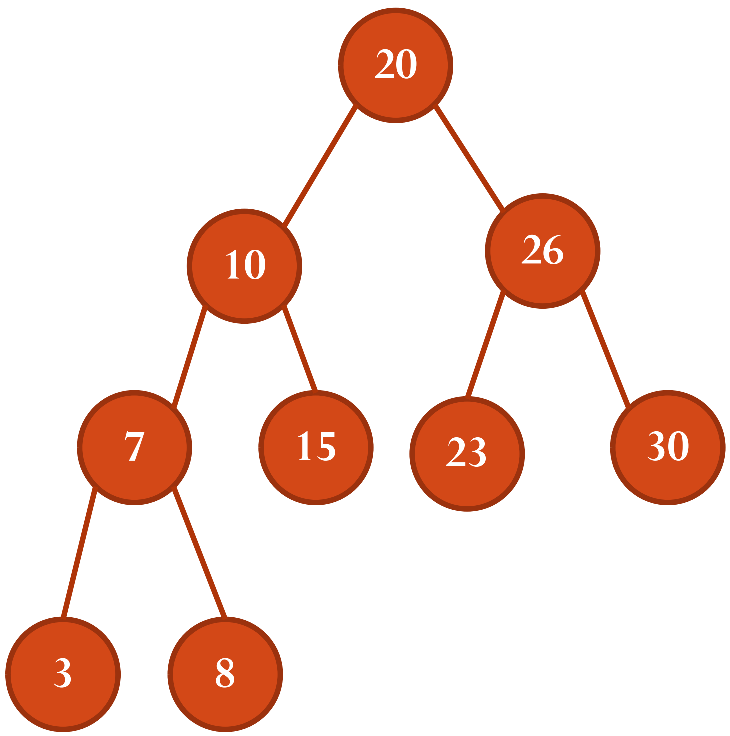
CÂY NHỊ PHÂN TÌM KIẾM

```
Node* del(Node* r, int v) {
  if(r == NULL)
    return NULL;
  else if(v < r->key) r->leftChild = del(r->leftChild, v);
  else if(v > r->key) r->rightChild = del(r->rightChild, v);
  else{
    if(r->leftChild != NULL && r->rightChild != NULL){
      Node* tmp = findMin(r->rightChild);
      r->key = tmp->key;
      r->rightChild = del(r->rightChild, tmp->key);
    }else{
      Node* tmp = r;
      if(r->leftChild == NULL) r = r->rightChild;
      else r = r->leftChild;
      delete tmp;
    }
  }
  return r;
}
```

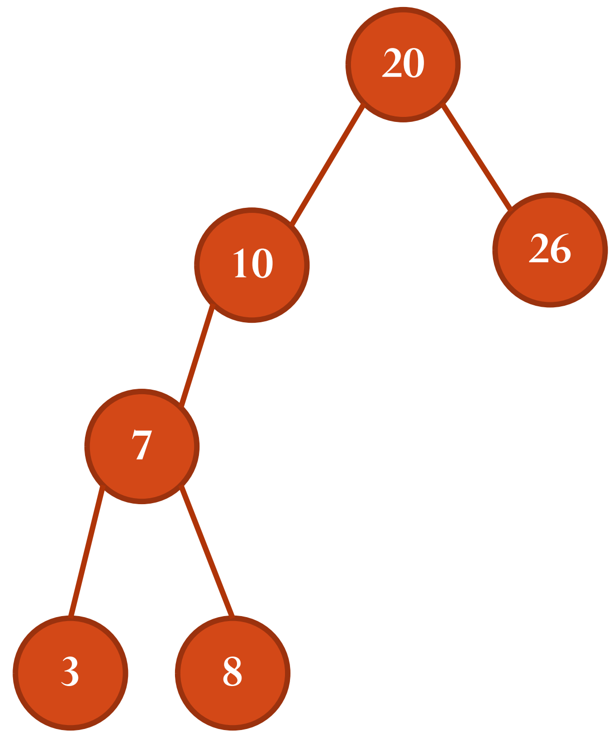
CÂY NHỊ PHÂN TÌM KIẾM

- Cây nhị phân tìm kiếm cân bằng (AVL)
  - Là một BST
  - Chênh lệch độ cao của nút con trái và con phải của mỗi nút không quá 1
  - Độ cao của cây logN (N là số nút)
  - Mỗi thao tác thêm, loại bỏ nút trên cây AVL cần bảo tồn tính cân bằng của cây

CÂY NHỊ PHÂN TÌM KIẾM



Cây AVL



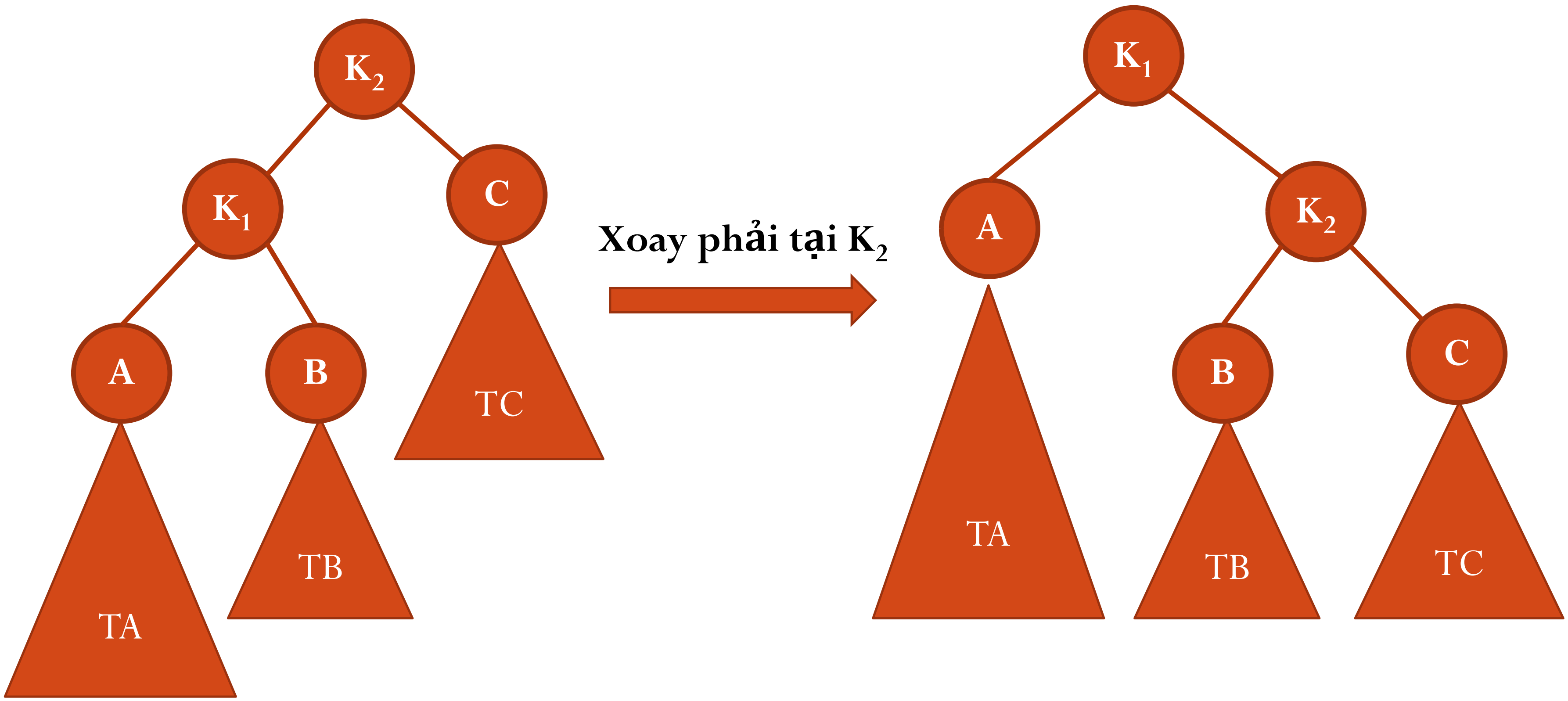
Cây BST nhưng không là AVL

CÂY NHỊ PHÂN TÌM KIẾM

- Mỗi thao tác loại bỏ hoặc thêm mới 1 nút trên AVL có thể làm mất tính cân bằng
  - Chênh lệch độ cao giữa 2 nút con của mỗi nút cùng lắm là 2 đơn vị
  - Thực hiện các phép xoay để khôi phục lại thuộc tính cân bằng

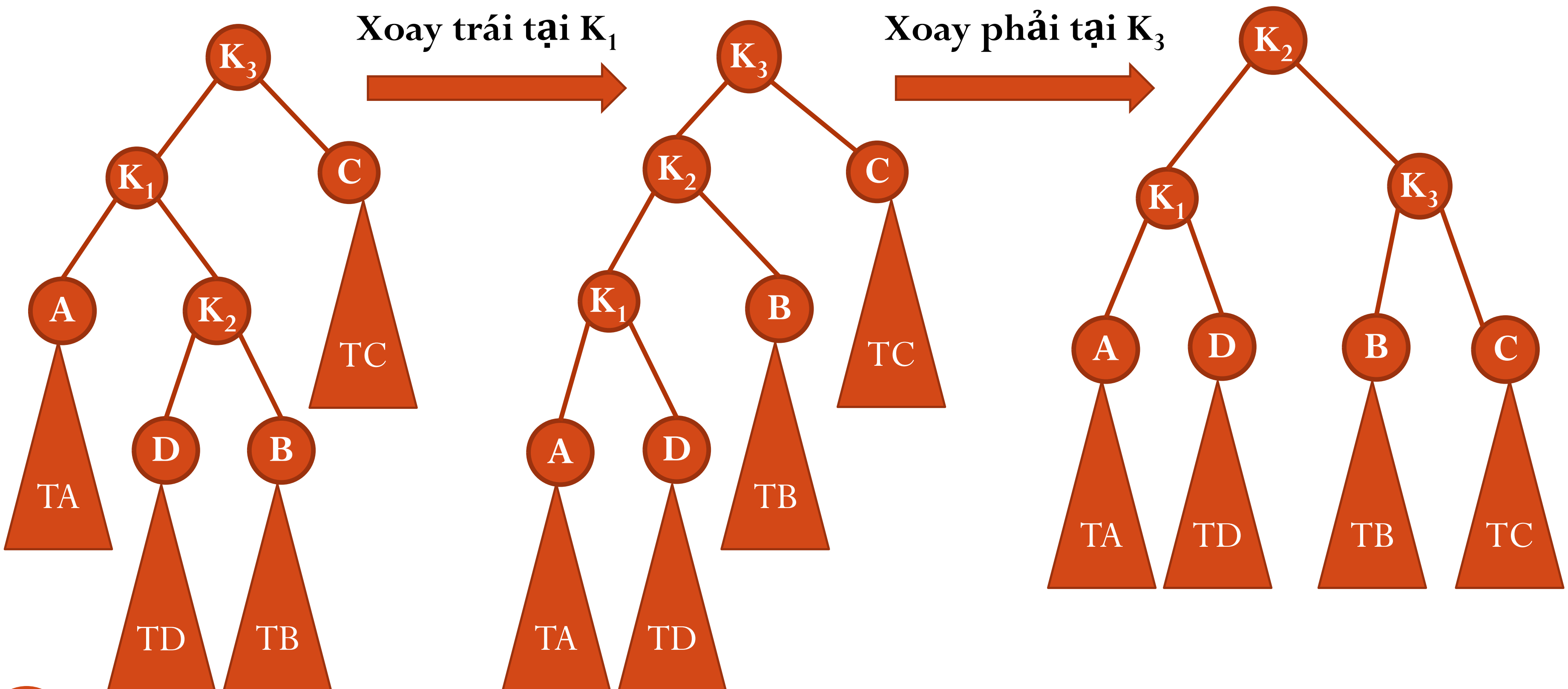
CÂY NHỊ PHÂN TÌM KIẾM

Trường hợp 1: chênh lệch độ cao của K_1 và C là 2, độ cao của B và C bằng nhau, độ cao của A hơn độ vào của B là 1 đơn vị



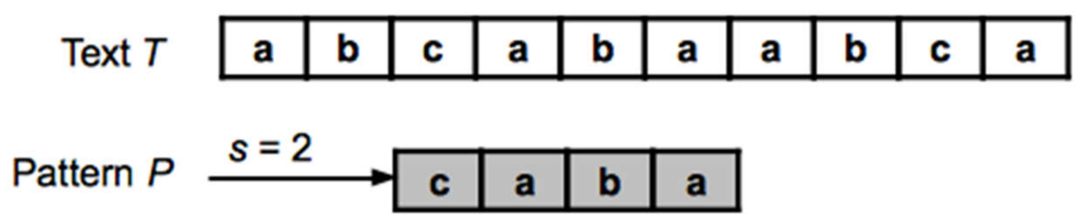
CÂY NHỊ PHÂN TÌM KIẾM

Trường hợp 2:



TÌM KIẾM XÂU MẪU

- Tìm sự xuất hiện của các xâu trong 1 văn bản cho trước
  - Cho văn bản T được biểu diễn bởi 1 mảng các ký tự $T[1..N]$ và 1 xâu (pattern) $P[1..M]$.
  - Cần tìm tất cả các vị trí xuất hiện của P trong T
  - P được gọi là xuất hiện trong T với độ lệch s nếu $P[i] = T[s+i]$ với mọi $i = 1,\ldots, M$



- Ứng dụng
  - Trình soạn thảo văn bản
  - Trích chọn thông tin
  - Xử lý chuỗi ADN

TÌM KIẾM XÂU MẪU

- Thuật toán trực tiếp
- Thuật toán Boyer Moore
- Thuật toán Rabin-Karp
- Thuật toán KMP (Knuth-Morris Pratt)

TÌM KIẾM XÂU MẪU

- Thuật toán trực tiếp
  - Xâu mẫu trượt từ trái qua phải của T
  - So khớp được thực hiện từ trái qua phải
  - Khi gặp trường hợp không khớp (mismatch) thì thực hiện dịch chuyển mẫu P ***một*** vị trí sang phải trên T

```
naiveSM(P, T) {
  foreach s = 0 . . N-M do
    i = 1;
    while i <= M && P[i] = T[i+s] do
      i = i + 1;
    endwhile
    if i > M then output(s);
  endfor
}
```

TÌM KIẾM XÂU MẪU

- Thuật toán Boyer Moore
  - Trượt xâu mẫu từ trái qua phải
  - Đối sánh: phải qua trái
  - Sử dụng thông tin tiền xử lý để bỏ qua càng nhiều ký tự càng tốt
  - Tiền xử lý xâu mẫu *P*
    - Last[*x*]: vị trí bên phải nhất xuất hiện ký tự *x* trong *P*
  - Khi tình trạng không khớp xảy ra với ký tự tồi là *x* (ký tự trên *T*), *P* sẽ được trượt 1 khoảng:

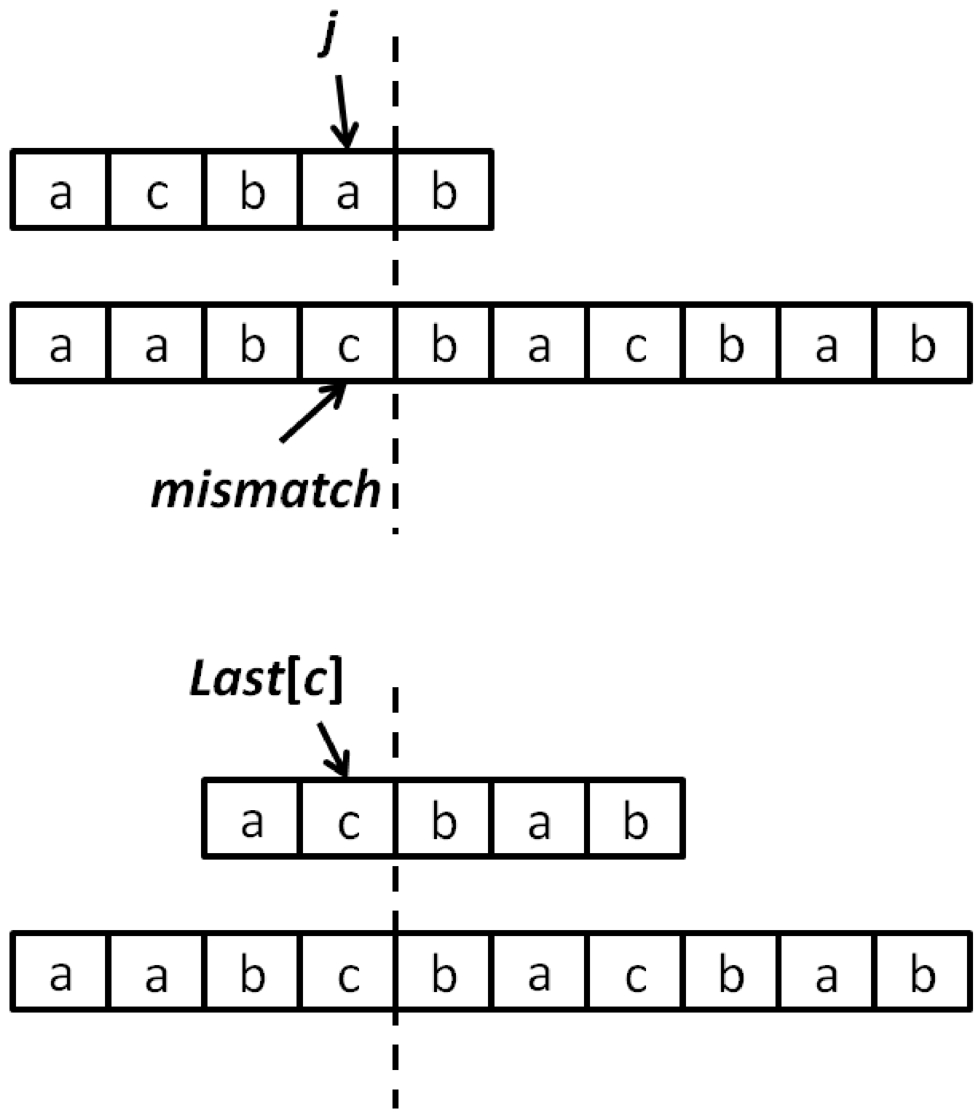
$$\max\{j - \text{Last}[x], 1\}$$

trong đó *j* là chỉ số hiện tại (xảy ra không khớp) trên *P* khi so khớp ký tự từ phải qua trái

```
computeLast(P) {
  for c = 0..255 do last[c] = 0;
  for i = m downto 1 do {
    if(last[P[i]] = 0) last[P[i]] = i;
  }
}
boyerMoore(P, T){
  s = 0;
  while(s <= N-M){
    j = M;
    while(j > 0 && T[j+s] = P[j])
      j = j-1;
    if(j = 0){
      output(s); s = s + 1;
    }else{
      k = last[T[j+s]];
      s = s + max(j-k,1);
    }
  }
}
```

TÌM KIẾM XÂU MẪU

- Thuật toán Boyer Moore
  - Last[a] = 4
  - Last[b] = 5
  - Last[c] = 2



TÌM KIẾM XÂU MẪU

- Thuật toán Rabin-Karp
  - Mỗi ký tự trong bảng chữ cái được biểu diễn bởi 1 số nguyên không âm nhỏ hơn d (d là độ dài của bảng chữ cái)
  - Đổi xâu $P[1..M]$ sang giá trị số nguyên dương

$$p = P[1]*d^{M-1} + P[2]*d^{M-2} + \ldots + P[M]*d^0$$

  - Đối sánh mẫu bằng cách so sánh 2 giá trị số nguyên dương tương ứng
  - Sử dụng lược đồ Horner để tăng tốc độ tính toán
  - Đổi xâu con $T[s+1 .. s+M]$ sang số

$$T_s = T[\mathrm{s}+1]*d^{M-1} + T[\mathrm{s}+2]*d^{M-2} + \ldots + T[\mathrm{s}+M]*d^0$$

  - T_{s+1} có thể được tính toán hiệu quả dựa vào T_s (được tính trước đó)

$$T_{s+1} = (T_s - T[s+1]*d^{M-1})*d + T[s+M+1]$$

TÌM KIẾM XÂU MẪU

- Thuật toán Rabin-Karp
  - Nhược điểm
    - Khi *M* lớn thì việc chuyển đổi xâu sang số mất thời gian đáng kể,
    - Có thể gây ra tràn số đối với kiểu dữ liệu cơ bản của ngôn ngữ lập trình
  - Cách giải quyết: thực hiện phép chia lấy đối với giá trị số dư cho Q
    - Khi 2 số dư khác nhau có nghĩa 2 giá trị số khác nhau và 2 xâu tương ứng cũng khác nhau
    - Khi 2 số dư bằng nhau, tiến hành đối sánh từng ký tự như cách truyền thống

TÌM KIẾM XÂU MẪU

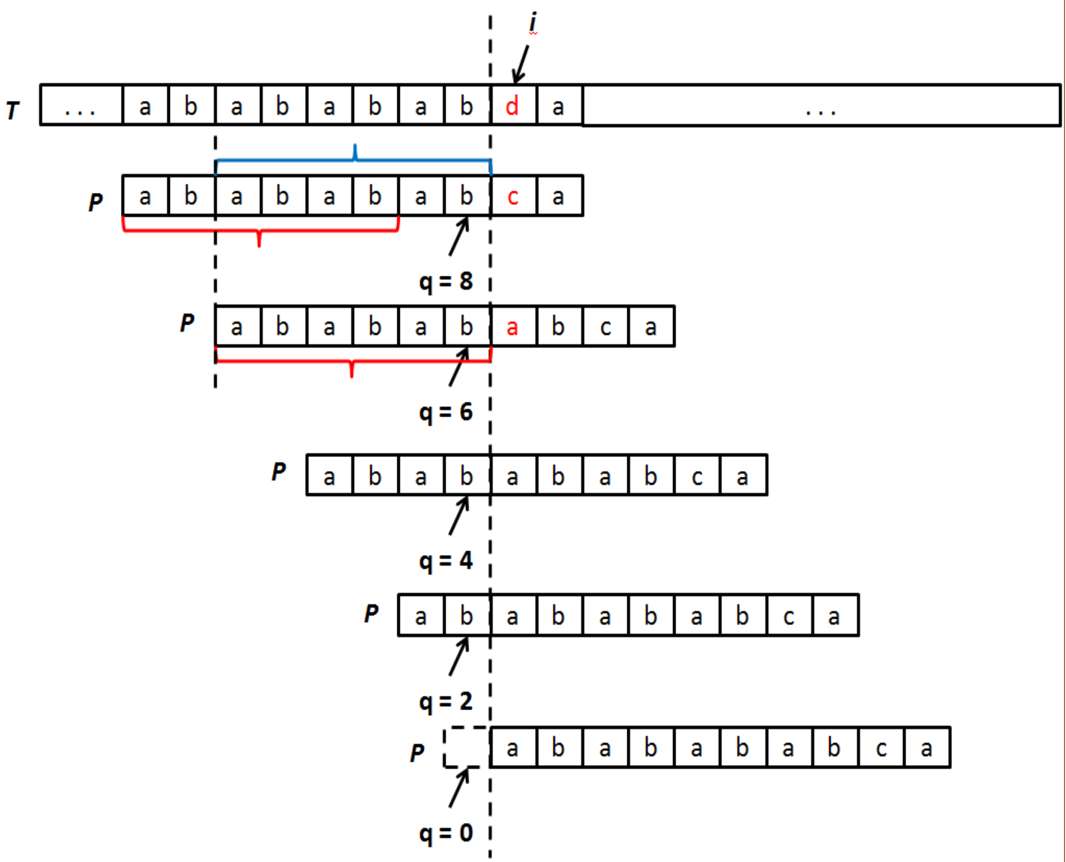
- Thuật toán KMP (Knuth-Morris Pratt)
  - Đối sánh: từ trái qua phải
  - Trượt: từ trái qua phải
  - π[q]: độ dài của tiền tố dài nhất cũng đồng thời là hậu tố ***ngặt*** của xâu P[1..q]

q	1	2	3	4	5	6	7	8	9	10
$P[q]$	a	b	a	b	a	b	a	b	c	a
$\pi[q]$	0	0	1	2	3	4	5	6	0	1

```
computePI(P){
  pi[1] = 0;
  k = 0;
  for q = 2..M do {
    while(k > 0 && P[k+1] != P[q])
      k = pi[k];
    if P[k+1] = P[q] then
      k = k + 1;
    pi[q] = k;
  }
}
```

TÌM KIẾM XÂU MẪU

- Thuật toán KMP (Knuth-Morris Pratt)



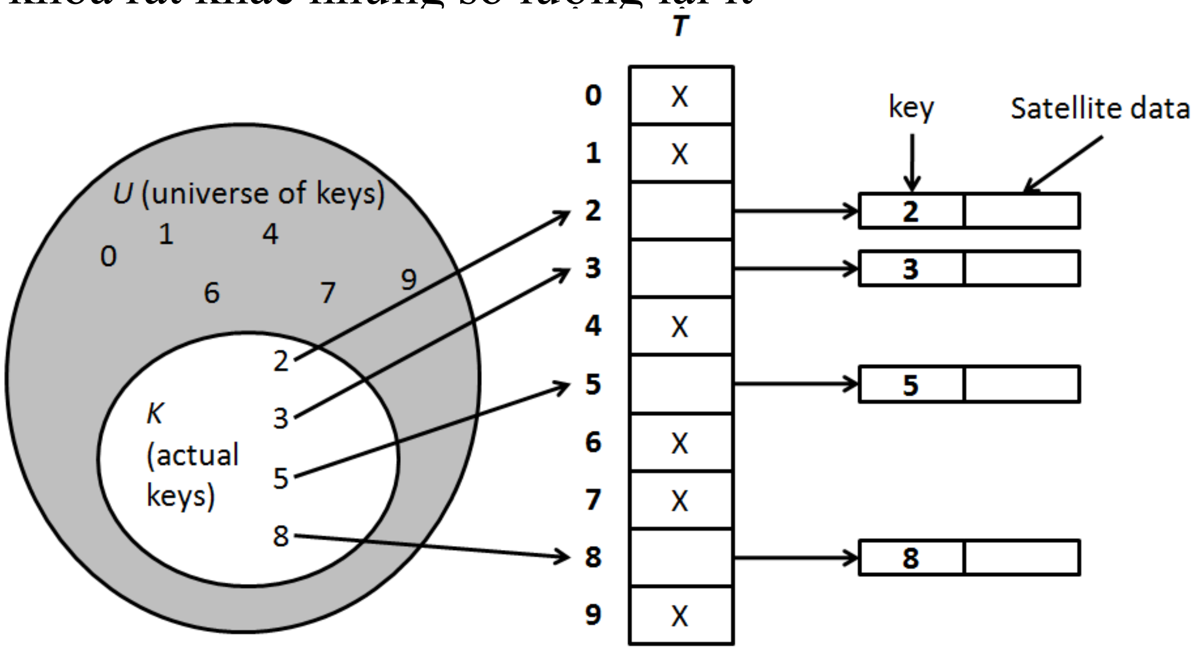

```
kmp(P, T){
  q = 0;
  for i = 1..N do {
    while q > 0 && P[q+1] != T[i]
      q = pi[q];
    if P[q+1] = T[i]
      q = q + 1;
    if(q = M){
      output(i-M+1);
      q = pi[q];
    }
  }
}
```

ÁNH XẠ VÀ BẢNG BĂM

- Từ điển: cấu trúc dữ liệu để ánh xạ một đối tượng với 1 đối tượng khác
  - put(*k*,*v*): thiết lập ánh xạ giữa khóa *k* và giá trị *v*
  - get(*k*): trả về giá trị tương ứng với khóa *k*
- Để cài đặt từ điển có thể dùng
  - Cây nhị phân tìm kiếm
  - Bảng băm

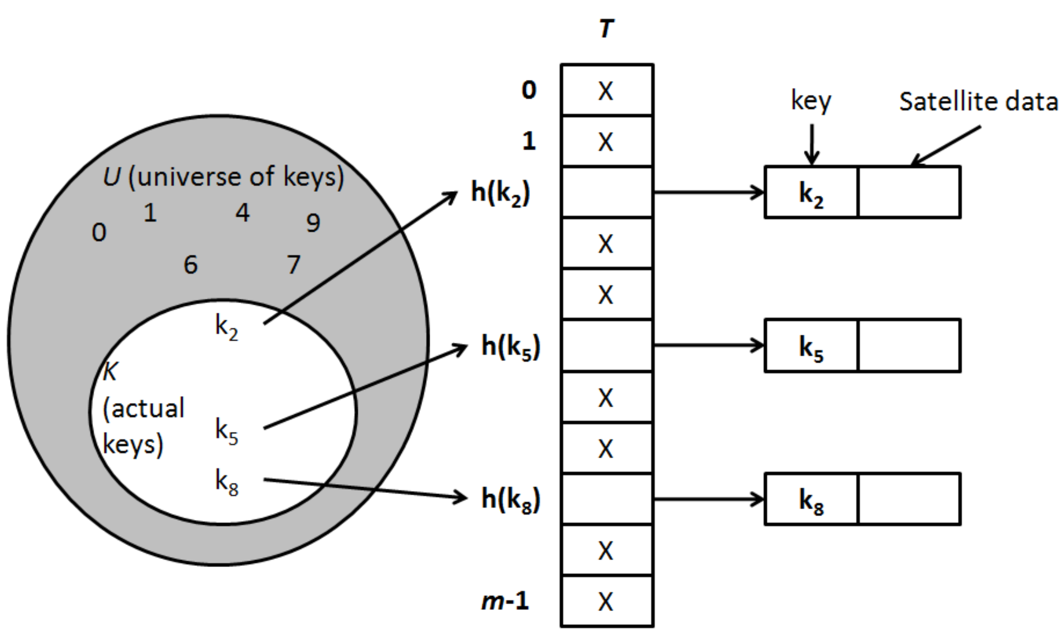
ÁNH XẠ VÀ BẢNG BĂM

- Phương pháp địa chỉ trực tiếp
  - Giá trị của khóa *k* sẽ là địa chỉ trực tiếp trong bảng nơi lưu trữ giá trị tương ứng với *k* trong từ điển
  - Ưu điểm: nhanh, đơn giản
  - Nhược điểm: Hiệu quả sử dụng bộ nhớ kém khi không gian khóa sử dụng có giá trị khóa rất khác nhưng số lượng lại ít



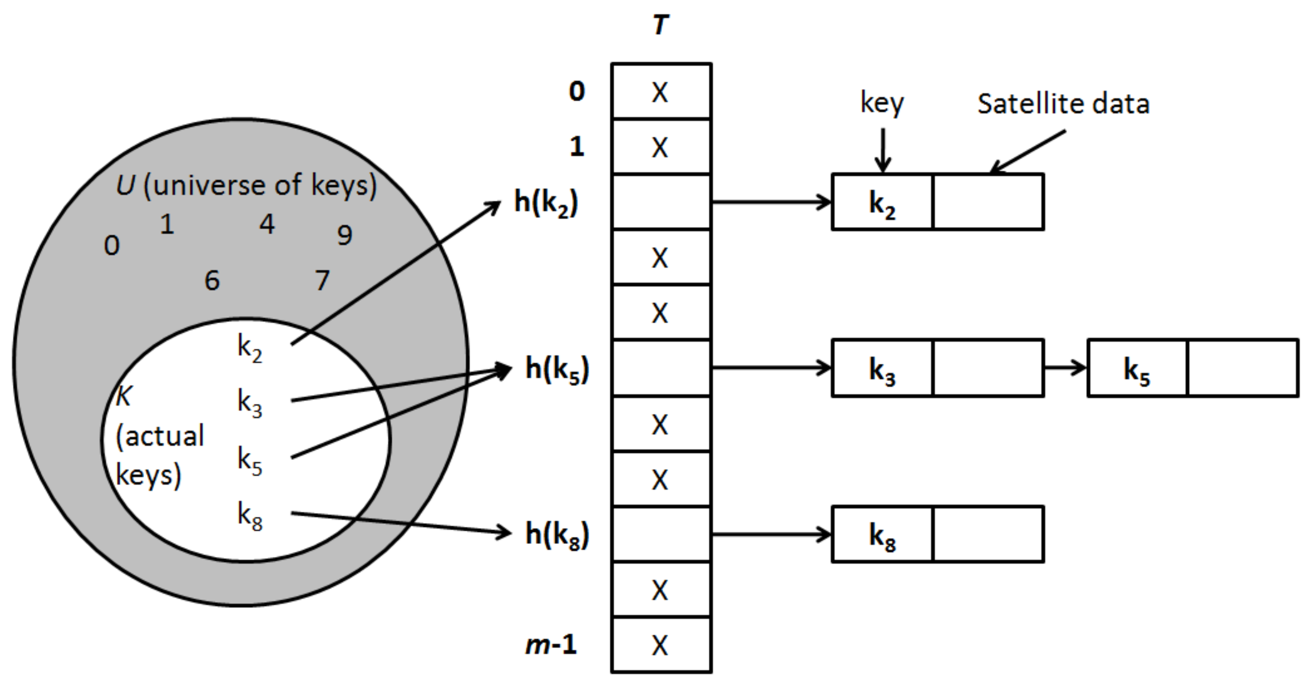
ÁNH XẠ VÀ BẢNG BĂM

- Phương pháp hàm băm
  - Với mỗi khóa *k*, hàm *h*(*k*) sẽ đưa ra địa chỉ trong bảng nơi lưu trữ giá trị tương ứng với *k*
  - Hàm *h*(*k*) phải đơn giản và tính toán phải hiệu quả



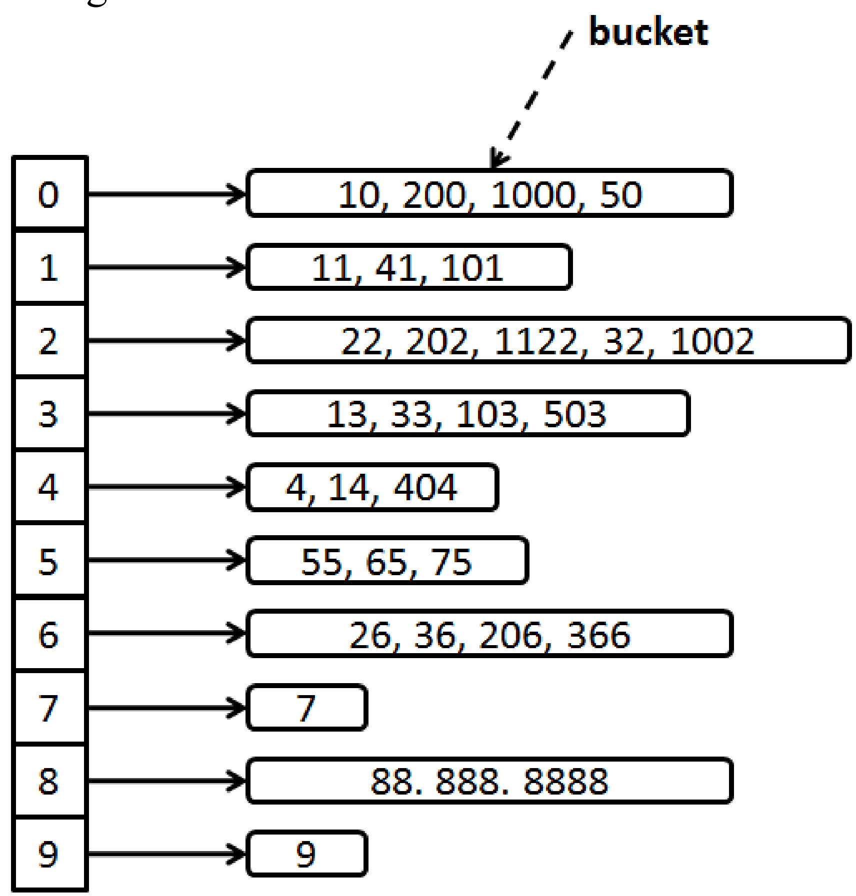
ÁNH XẠ VÀ BẢNG BĂM

- Phương pháp hàm băm
  - Xung đột: hai khoá khác nhau cho hai giá trị hàm băm bằng nhau
  - Giải pháp:
    - Nhóm chuỗi (Chaining): Các đối tượng cho cùng giá trị hàm băm sẽ được nhóm theo chuỗi (danh sách liên kết, cây hoặc bảng băm cấp 2)
    - Phương pháp địa chỉ mở (Open Addressing)



ÁNH XẠ VÀ BẢNG BĂM

- Hàm băm phổ biến: hàm chia lấy dư (mod)
  - Giả thiết khóa là các số nguyên
  - $h(k) = k \bmod m$ trong đó m là số phần tử của bảng lưu trữ



ÁNH XẠ VÀ BẢNG BĂM

- Phương pháp địa chỉ mở (Open Addressing)
  - Các cặp (khoá, giá trị) được lưu trữ ngay trong bảng có m vị trí
  - Thao tác put(k, v) và get(k) sẽ cần dò (probe) bảng lưu trữ
    - Thao tác put(k, v): dò để tìm ra vị trí còn trống để lưu (k, v)
    - Thao tác get(k): dò để tìm ra vị trí trong bảng lưu trữ khoá k
    - Thứ tự dò: $h(k, 0)$, $h(k, 1)$, $h(k, 2)$, …, $h(k, m\text{-}1)$
    - Có các phương pháp dò
      - Dò tuyến tính: $h(k, i) = (h_1(k) + i) \bmod m$ trong đó h_1 là hàm băm thông thường
      - Dò quadratic: $h(k, i) = (h_1(k) + c_1 i + c_2 i^2) \bmod m$ trong đó h_1 là hàm băm thông thường
      - Băm kép: $h(k, i) = (h_1(k) + i * \ h_2(k) \bmod m$ trong đó h_1 và h_2 là hàm băm thông thường

ÁNH XẠ VÀ BẢNG BĂM

- Phương pháp địa chỉ mở (Open Addressing)
  - Thao tác put(*k*, *v*)

```
put(k, v)
{
  // T: bảng lưu trữ
  x.key = k; x.value = v;
  i = 0;
  while(i < m) {
    j = h(k,i);
    if(T[j] = NULL) {
      T[j] = x; return j;
    }
    i = i + 1;
  }
  error("Hash table overflow");
}
```

ÁNH XẠ VÀ BẢNG BĂM

- Phương pháp địa chỉ mở (Open Addressing)
  - Thao tác get(*k*)

```
get(k)
{
  // T: bảng lưu trữ
  i = 0;
  while(i < m) {
    j = h(k,i);
    if(T[j].key = k) {
      return T[j];
    }
    i = i + 1;
  }
  return NULL;
}
```

ÁNH XẠ VÀ BẢNG BĂM

- Bài tập: một bảng lưu trữ được cấp phát m phần tử, áp dụng phương pháp địa chỉ mở với hàm dò $h(k, i)$ có dạng

$$h(k, i) = (k \bmod m + i) \bmod m$$

- Ban đầu bảng trống rỗng, hãy vẽ trạng thái bảng khi chèn liên tiếp các khoá 7, 8, 6, 17, 4, 28 vào bảng trong trường hợp $m = 10$