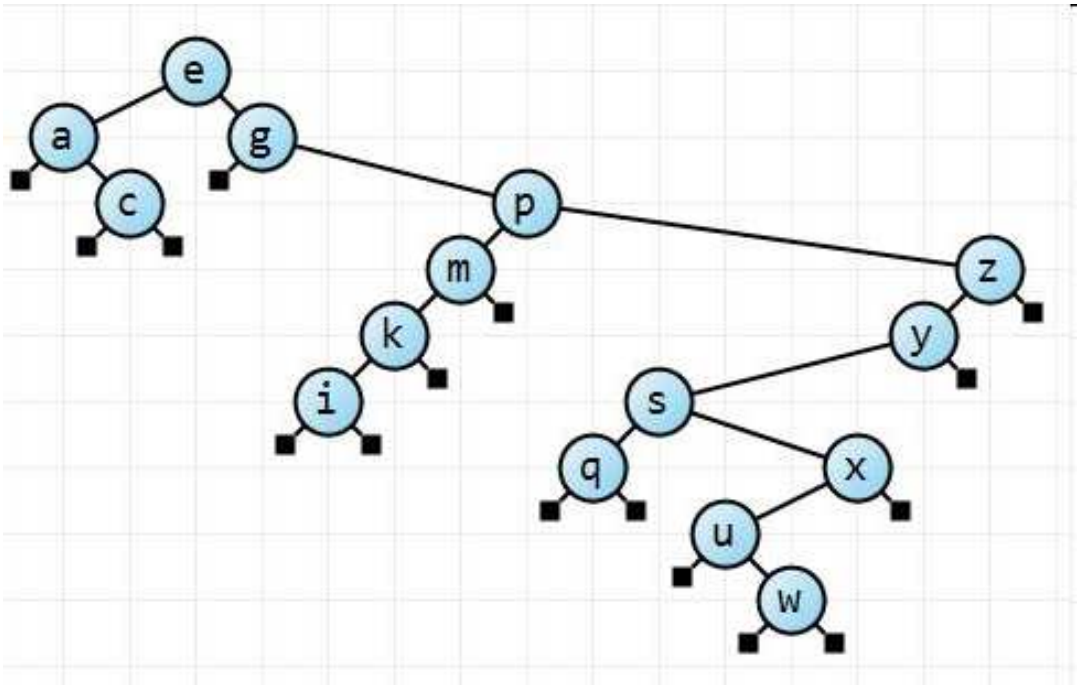
Các cấu trúc dữ liệu nâng cao

(Advanced Data Structures)

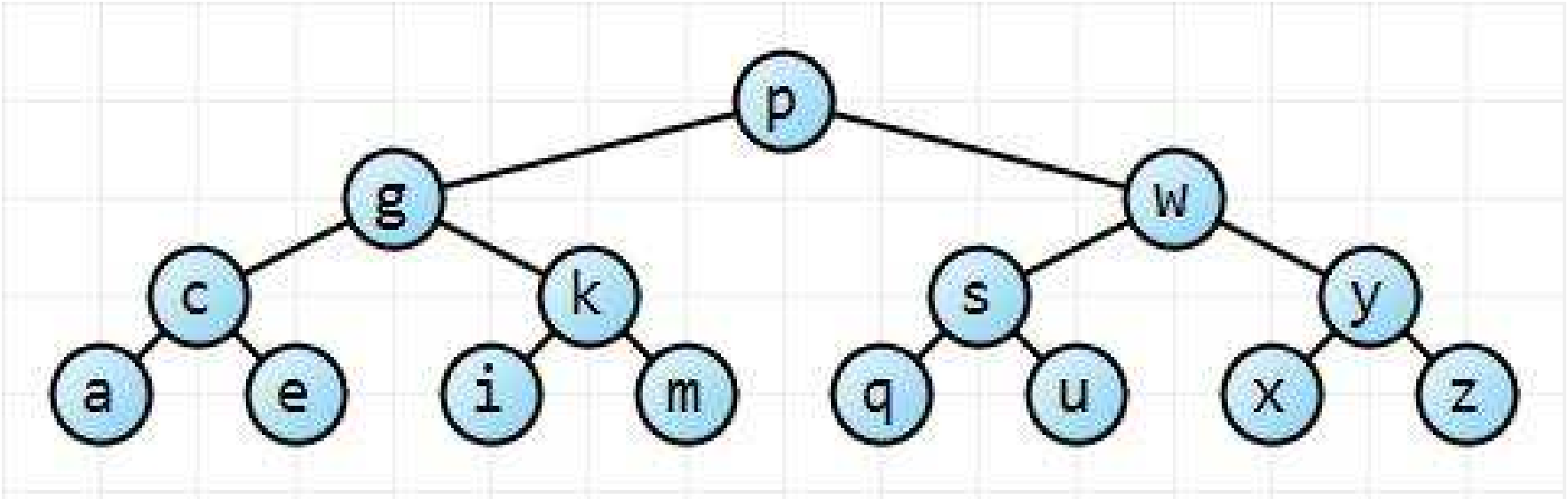
Cây nhị phân tìm kiếm cân bằng (1)

- Cây BST có thể bị lệch
  - Vì sao cây BST trở nên bị lệch ?
  - Chi phí tìm kiếm trên cây bị lệch ?



Một cây BST không cân bằng

Cây nhị phân tìm kiếm cân bằng (2)



Cây cân bằng → chiều cao và chi phí tìm kiếm tối ưu $O(\log_2 N)$

Cây nhị phân tìm kiếm cân bằng (3)

Cần có phương pháp để duy trì tính cân bằng cho cây BST

Cây nhị phân tìm kiếm cân bằng (4)

- Các loại cây BST cân bằng
  - Cây AVL
  - Cây Đỏ - Đen (Red – Black tree)
  - Cây AA

Cây AVL (1)

- Định nghĩa
- Cài đặt cấu trúc dữ liệu
- Mất cân bằng khi thêm/xóa node
- Các thuật toán điều chỉnh cây
- Đánh giá/so sánh

E. M. Landis

G. M. Adelson-Velskii

Cây AVL (2)

- Cấu trúc cây AVL do 2 tác giả người Liên xô: G. M. Adelson-Velskii và E. M. Landis công bố năm 1962

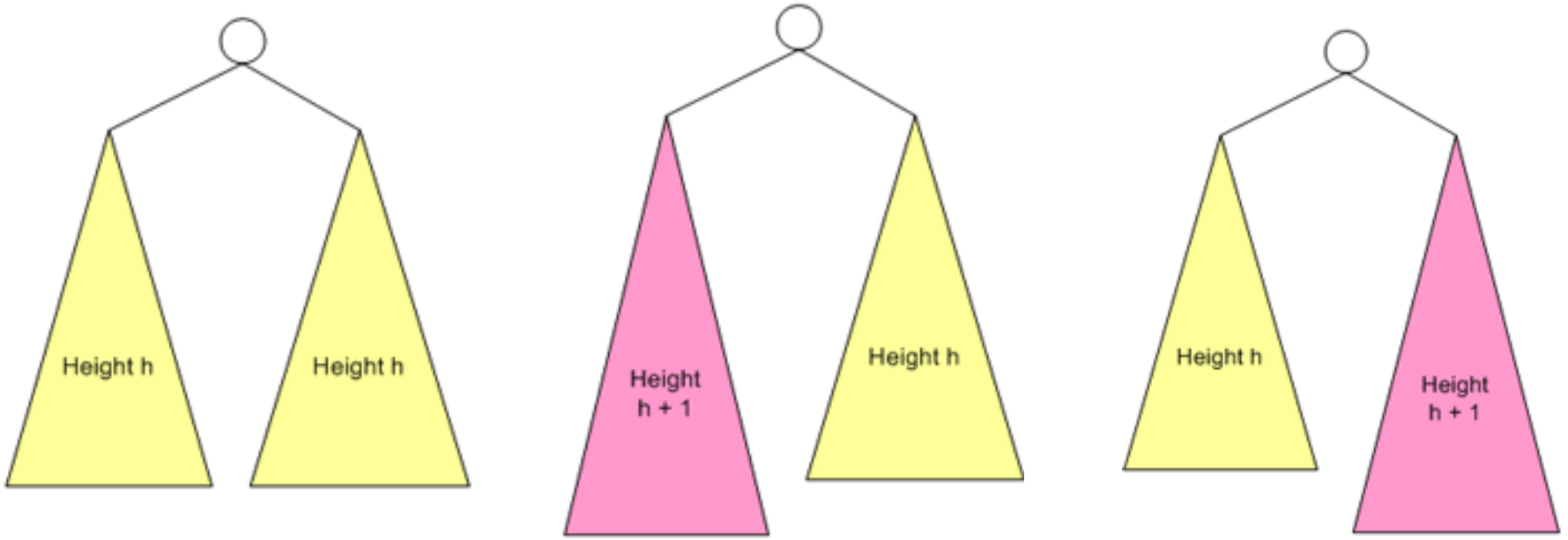
- Đây là mô hình cây tự cân bằng đầu tiên được đề xuất *(self-adjusting, height-balanced binary search tree)*

Định nghĩa cây AVL (1)

- Cây AVL:
  - Là một cây nhị phân tìm kiếm (BST)
  - Mỗi nút p của cây đều thỏa: chiều cao của cây con bên trái (`p->left`) và chiều cao của cây con bên phải (`p->right`) chênh lệch nhau không quá 1

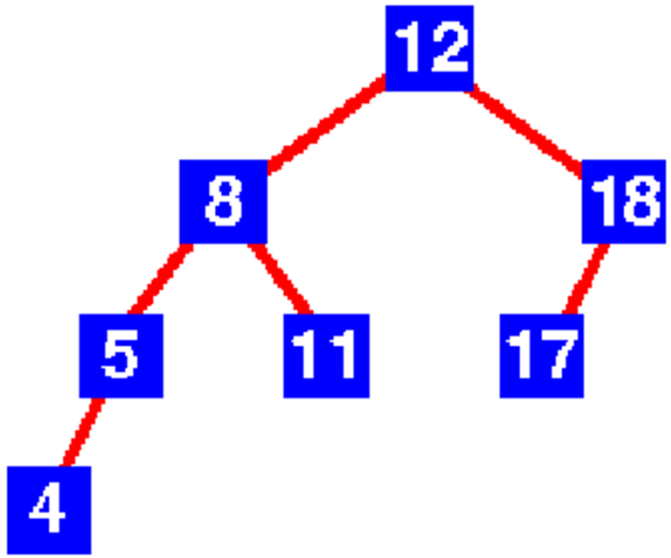
$$\forall p \in T_{AVL}: abs(h_{p->left} - h_{p->right}) \leq 1$$

Định nghĩa cây AVL (2)

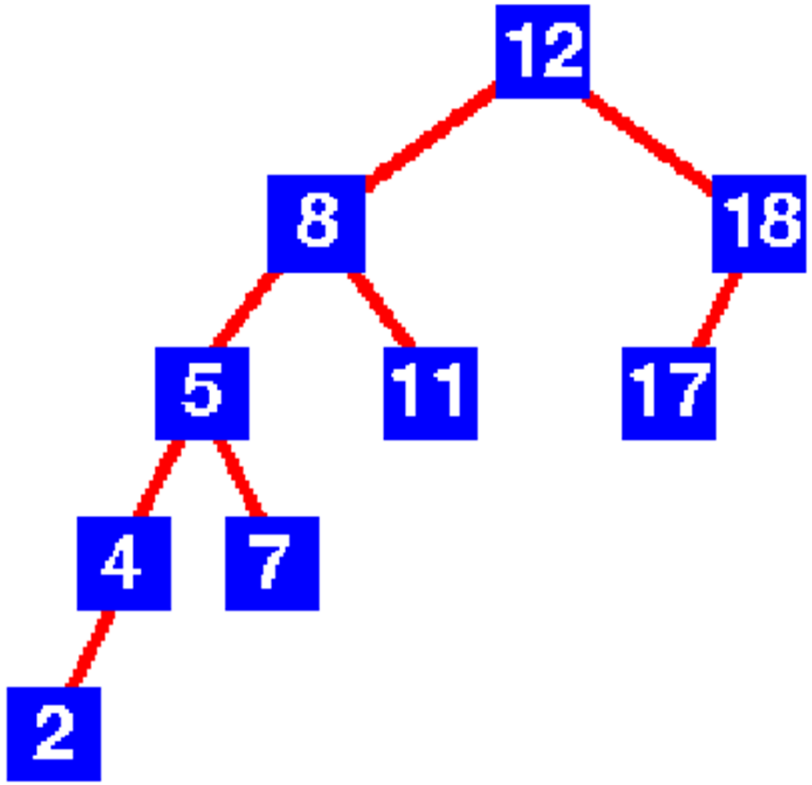


Chiều cao 2 cây con left, right chênh lệch không quá 1

Định nghĩa cây AVL (3)



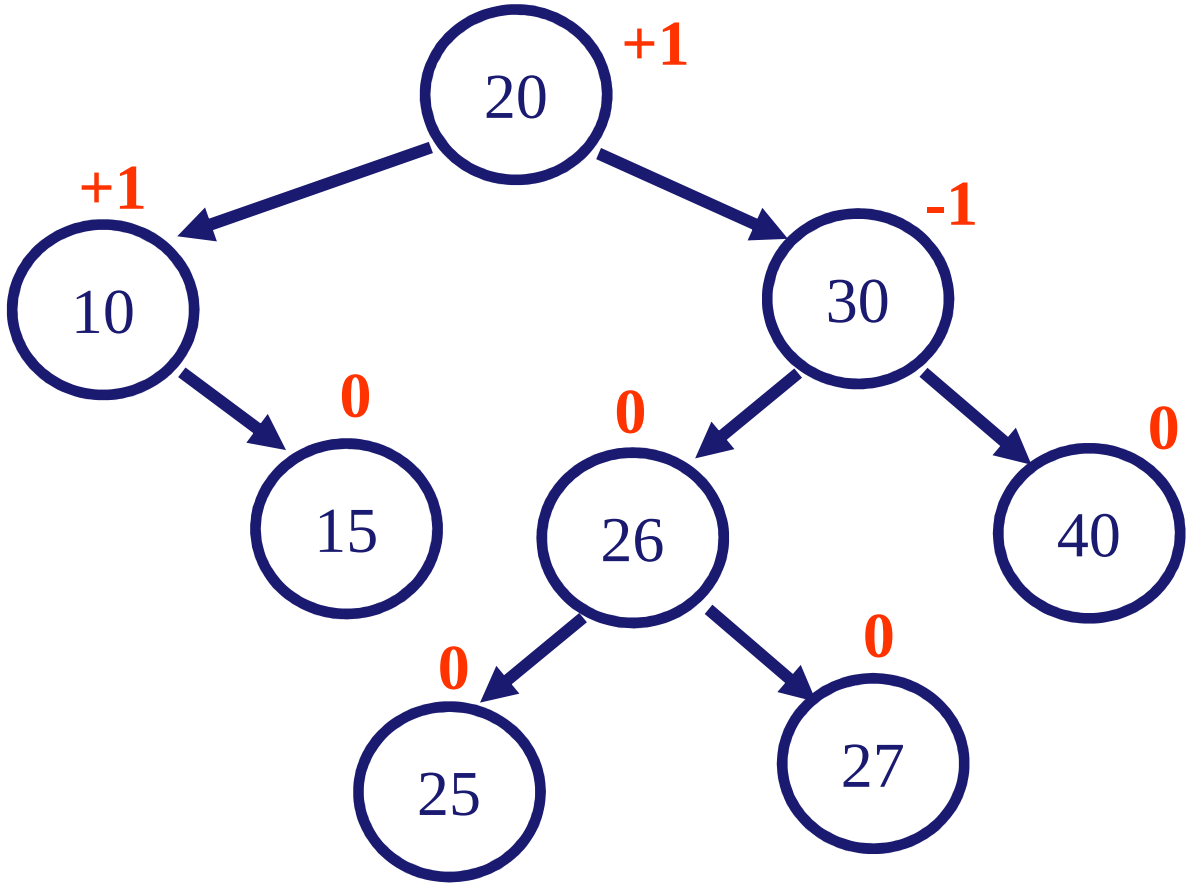
Cây AVL ?



Cài đặt cấu trúc dữ liệu (1)

- Cấu trúc node, tree tương tự như BST
- Thêm vào mỗi node một field *balance*, diễn tả trạng thái cân bằng của node đó:
  - ***balance* = *-1***: node lệch trái (cây con trái cao hơn cây con phải)
  - ***balance* = *0***: node cân bằng (cây con trái cao bằng cây con phải)
  - ***balance* = *+1***: node lệch phải (cây con phải cao hơn cây con trái)

Cài đặt cấu trúc dữ liệu (2)



Hệ số cân bằng của các node trong cây AVL

Cài đặt cấu trúc dữ liệu (3)

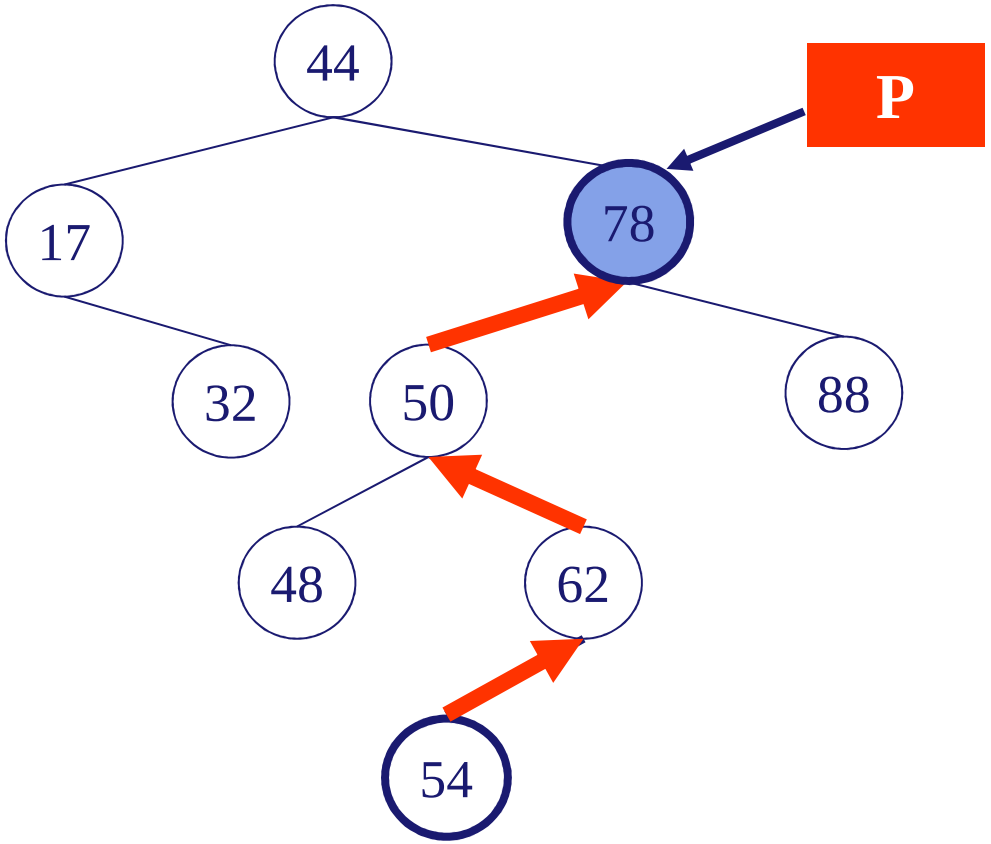
```
template <class T> class AVLNode {
   public:
        T               key;          // key of node
        char            balance;      // balance status of node
        BSTNode         *left;        // pointer to left child
        BSTNode         *right;       // pointer to right child

        BSTNode() {   }
        BSTNode(T aKey)
        {
                key = aKey;
                balance = 0;
                left = right = NULL;
        }
}; // end class
```

Mất cân bằng khi thêm/xóa node (1)

- [Insert – Thêm 1 phần tử vào cây]: có thể làm cây mất cân bằng.
  - Duyệt từ node vừa thêm ngược về node gốc
  - Nếu tìm thấy node P bị mất cân bằng thì tiến hành xoay cây tại nút P (chỉ cần điều chỉnh 1 lần duy nhất)

Mất cân bằng khi thêm/xóa node (2)

Thêm phần tử 54 làm cây mất cân bằng tại node P

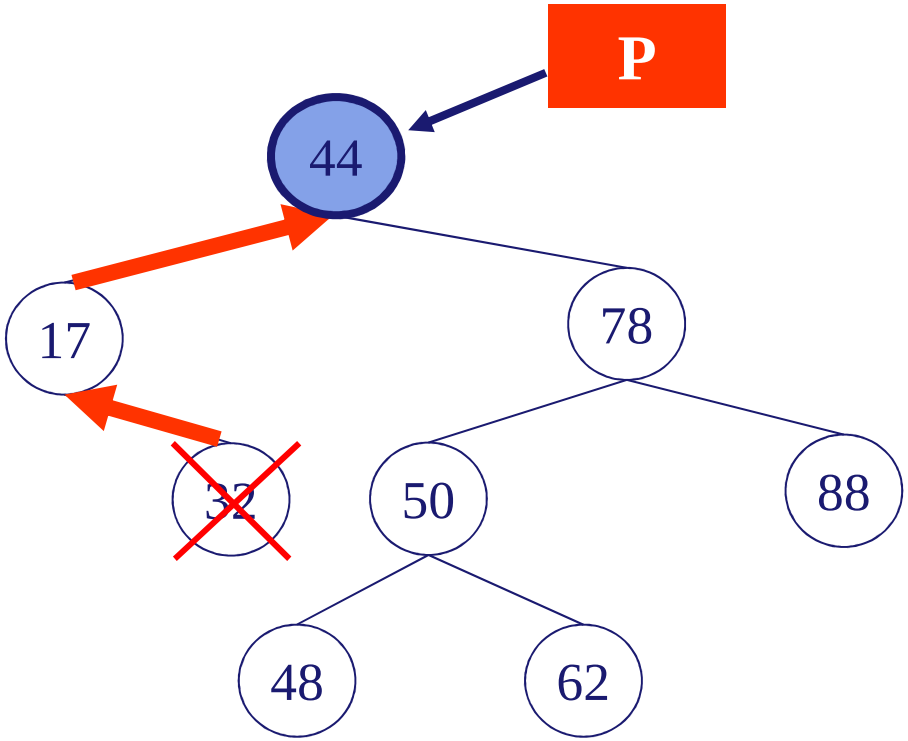


Mất cân bằng khi thêm/xóa node (3)

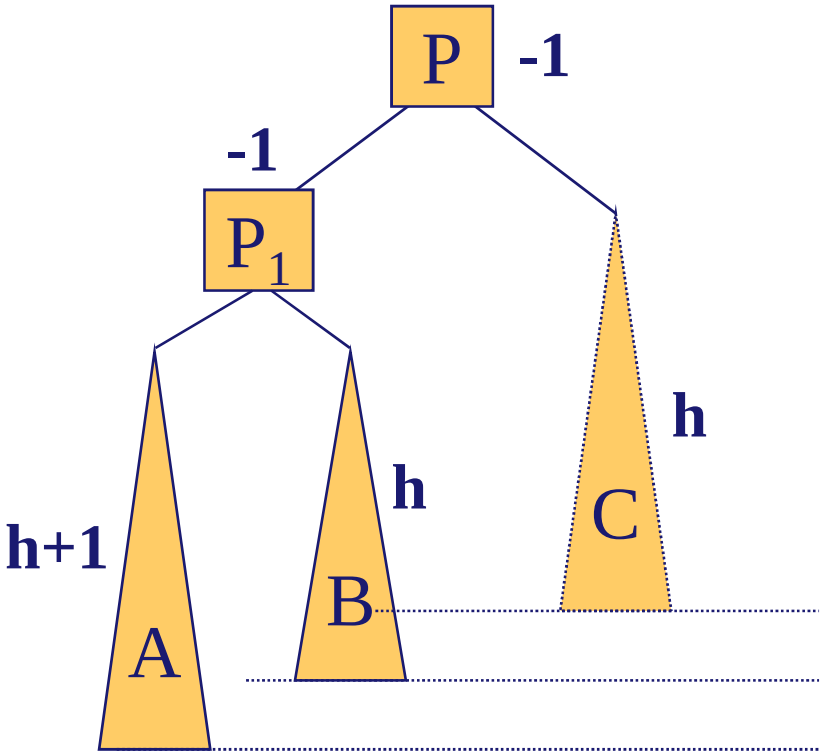
- [Delete – Xóa 1 phần tử]: có thể làm cây mất cân bằng.
  - Duyệt từ node vừa xóa ngược về node gốc
  - Nếu tìm thấy node P bị mất cân bằng thì tiến hành xoay cây tại node P
  - *Lưu ý:* Thao tác điều chỉnh có thể làm cho những node phía trên của node P bị mất cân bằng → cần điều chỉnh cho đến khi không còn node nào bị mất cân bằng nữa (lùi dần về node gốc)

Mất cân bằng khi thêm/xóa node (4)

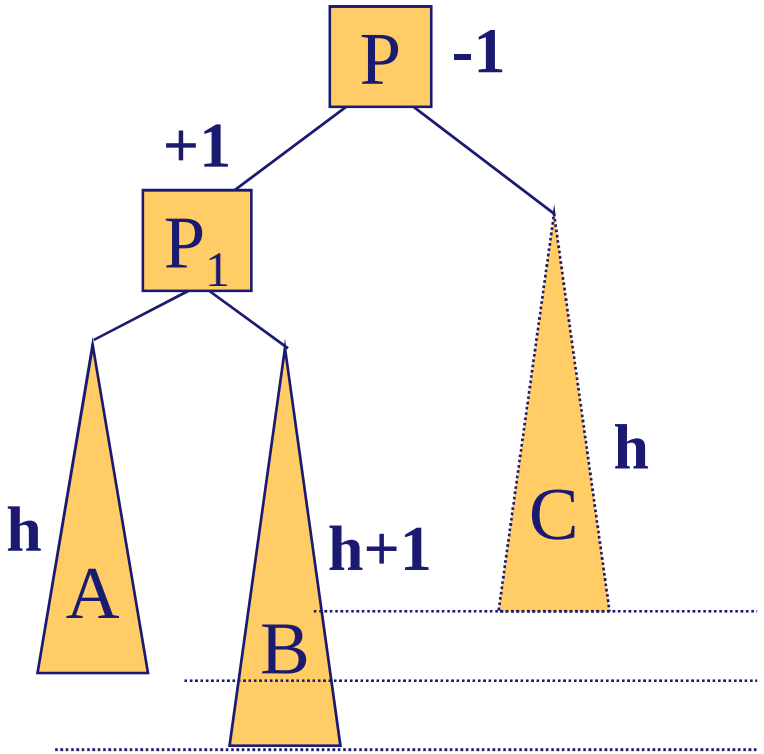
Xóa phần tử 32 làm cây mất cân bằng tại node P



Các thuật toán điều chỉnh cây (1)



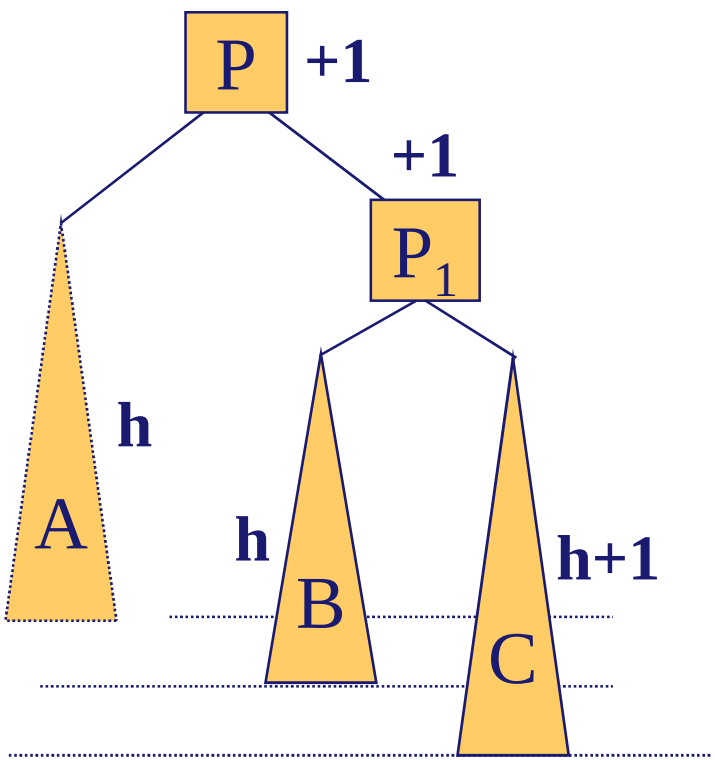
(a1)



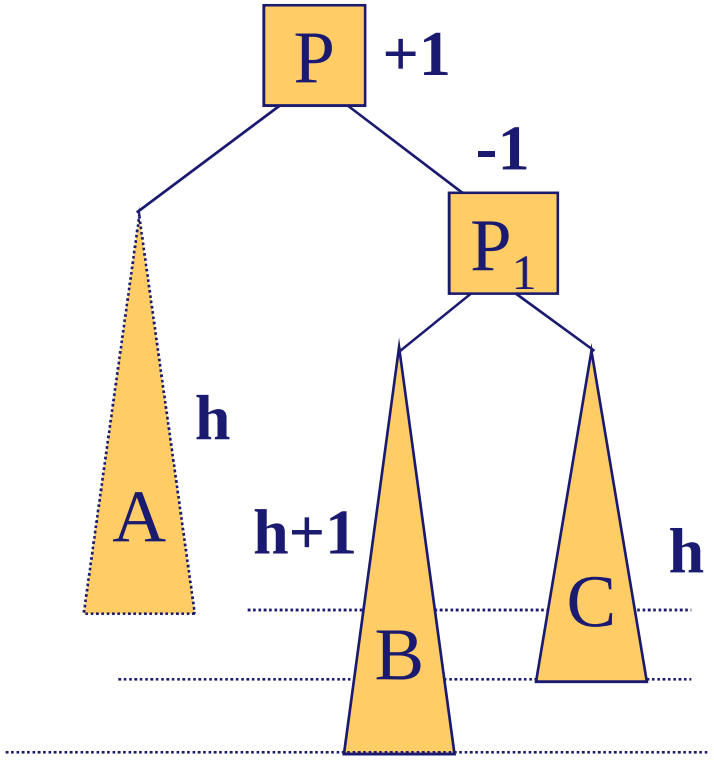
(b1)

Hai trường hợp cây bị mất cân bằng ở nhánh trái

Các thuật toán điều chỉnh cây (2)



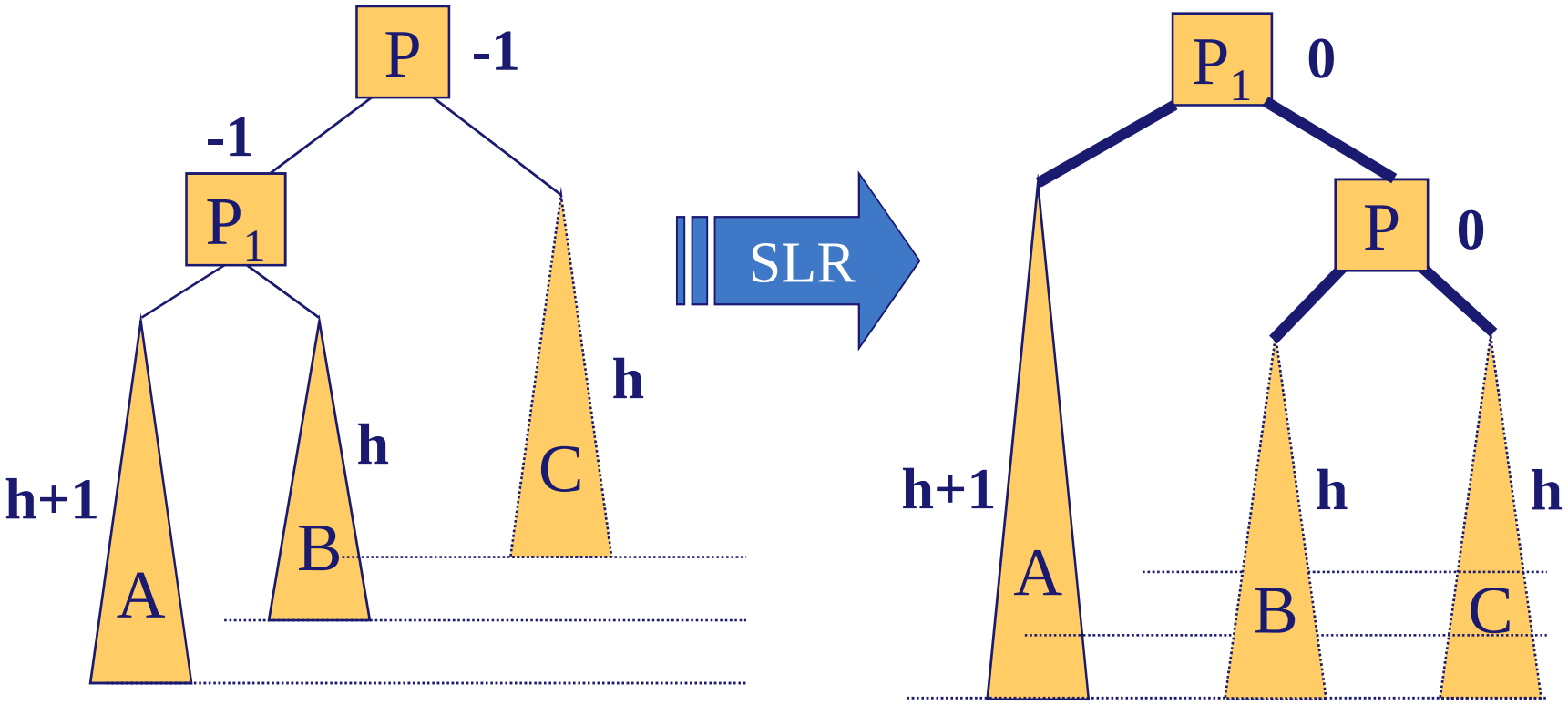
(a2)



(b2)

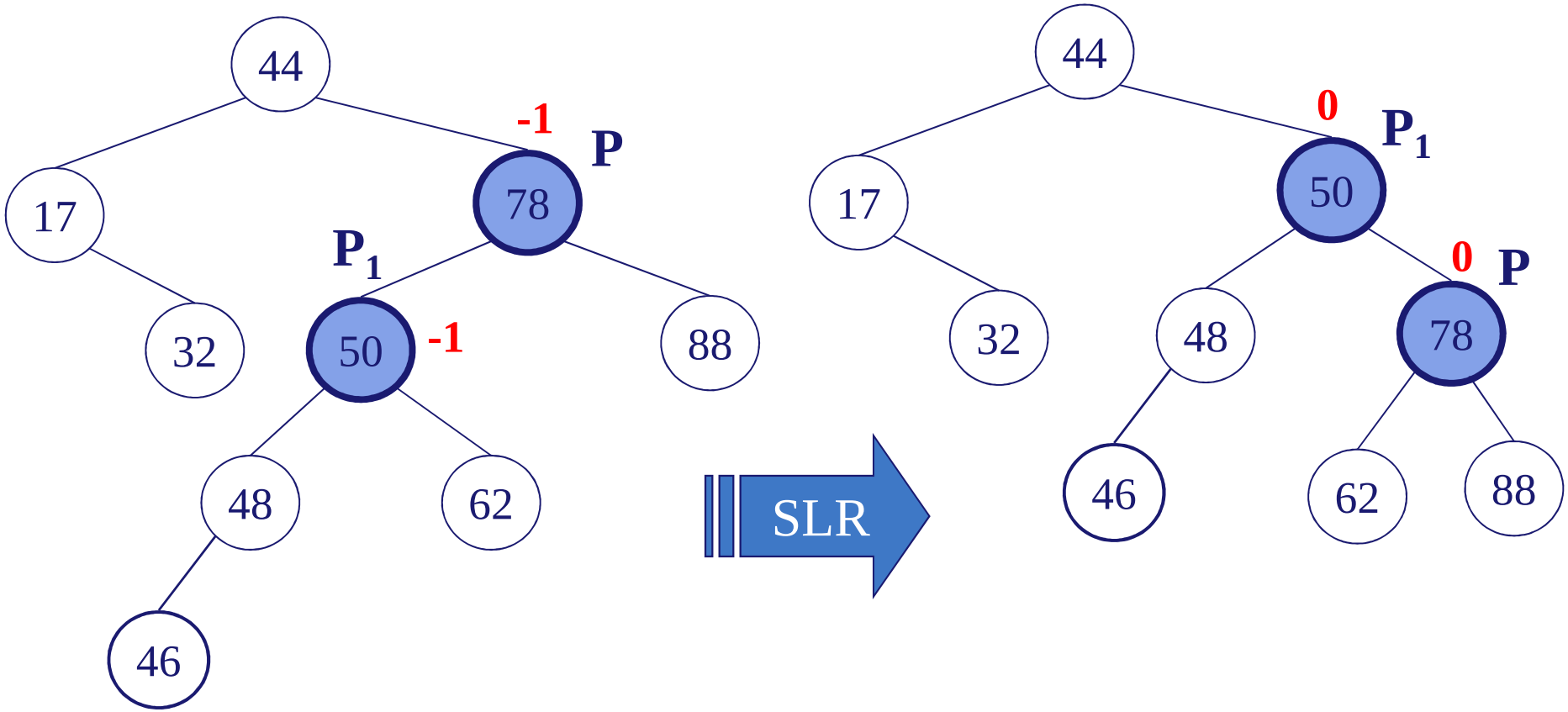
Hai trường hợp cây bị mất cân bằng ở nhánh phải

Các thuật toán điều chỉnh cây (3)



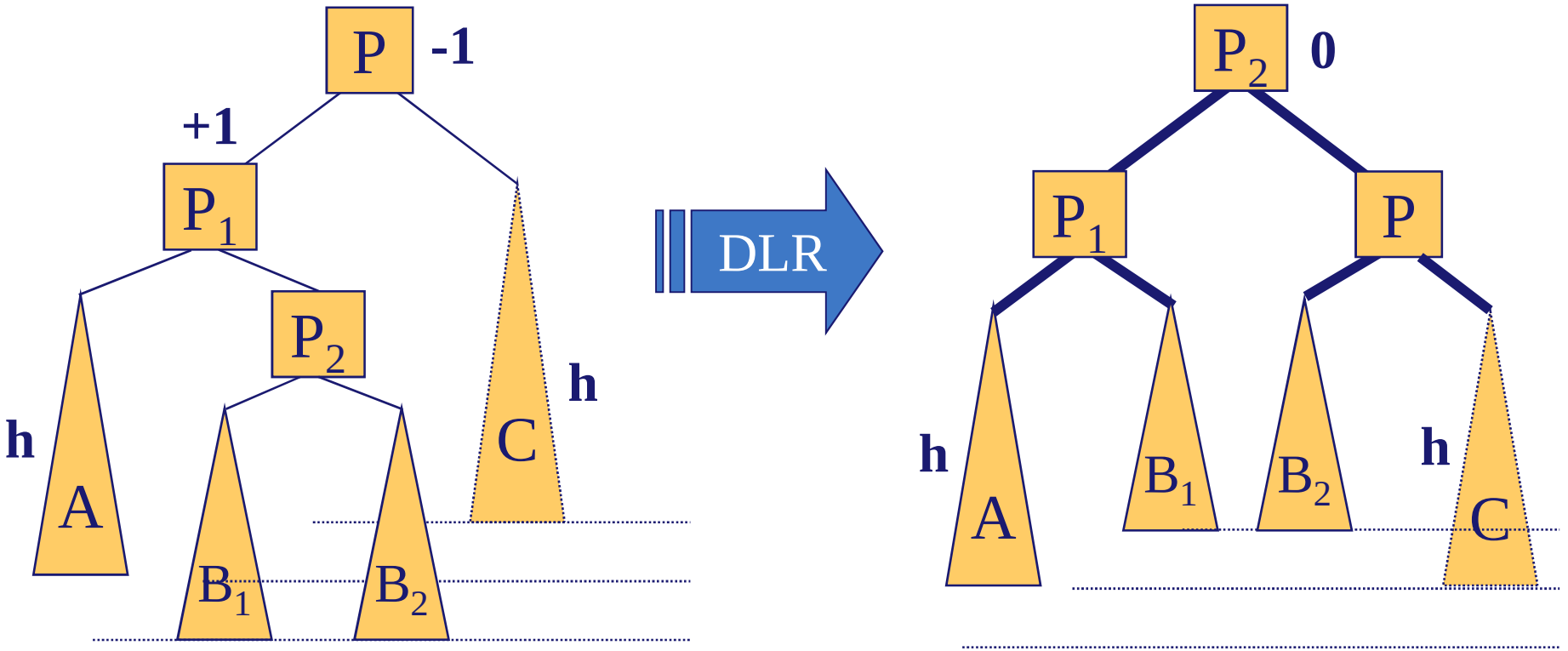
Trường hợp (a1): áp dụng phép xoay đơn Trái - Phải
(SLR – Single Left-to-Right)

Các thuật toán điều chỉnh cây (4)



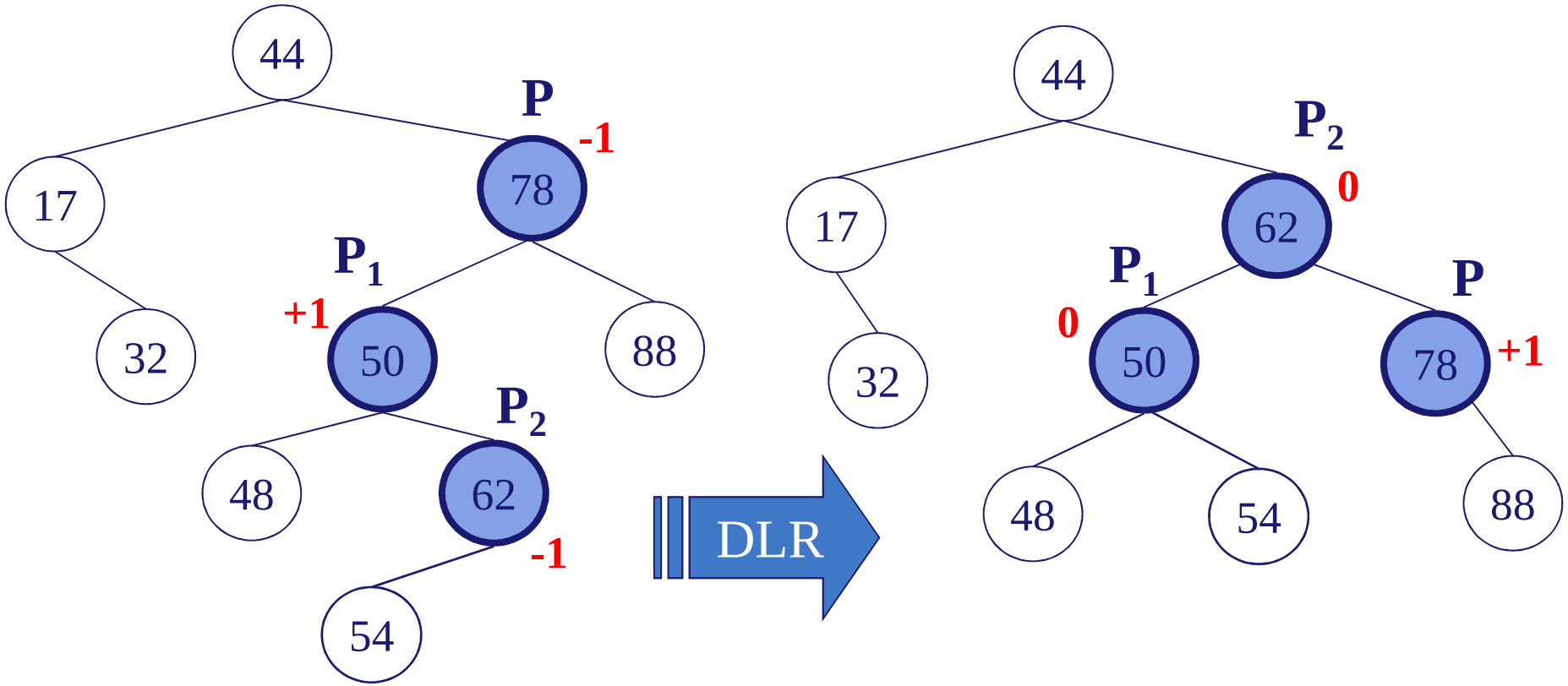
Ví dụ: điều chỉnh cây bằng thao tác xoay đơn SLR

Các thuật toán điều chỉnh cây (5)



Trường hợp (b1): áp dụng phép xoay kép Trái - Phải
(DLR – Double Left–to-Right)

Các thuật toán điều chỉnh cây (6)



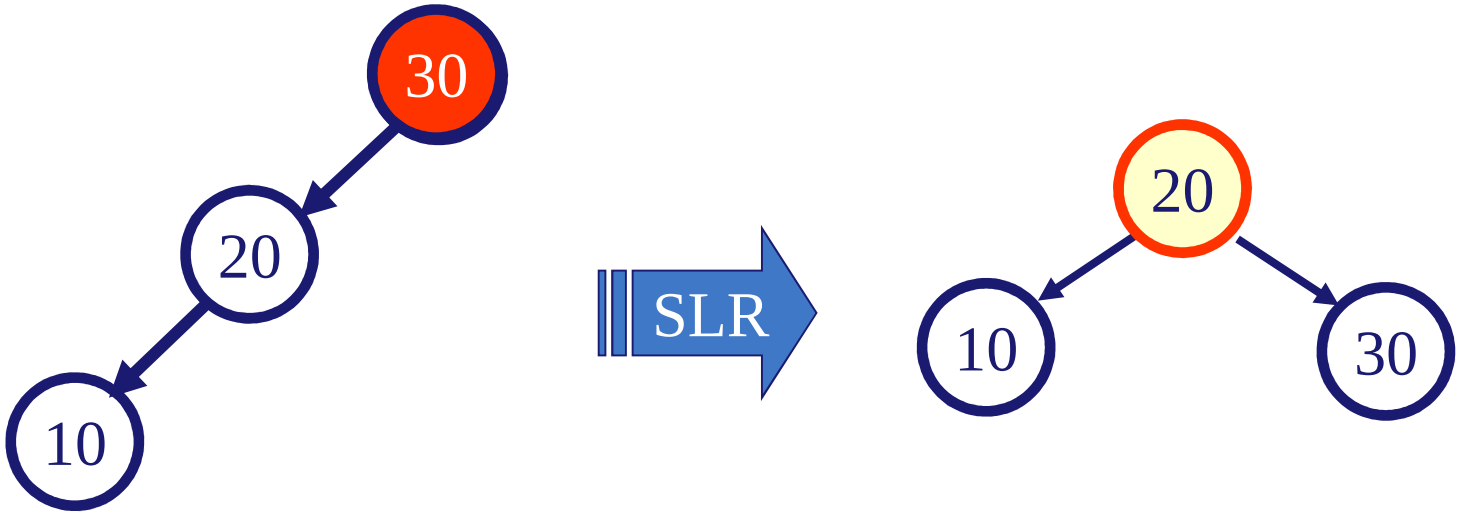
Ví dụ: thao tác xoay kép DLR

Các thuật toán điều chỉnh cây (7)

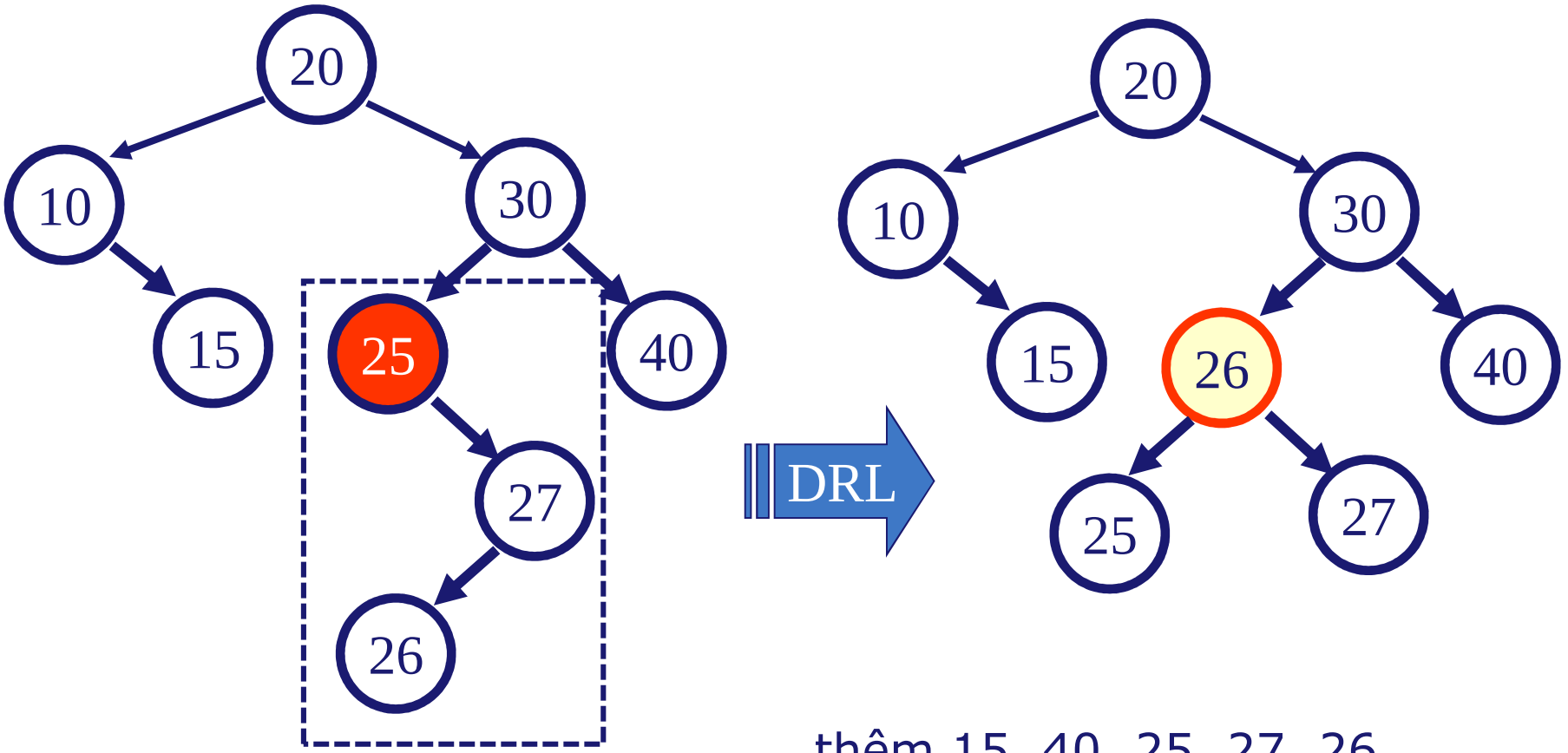
- Đối với trường hợp (a2) và (b2)
  - Xử lý tương tự như (a1) và (b1), đối xứng qua trục đứng
- Trường hợp (a2)
  - Áp dụng phép xoay SRL – Single Right-to-Left
- Trường hợp (b2)
  - Áp dụng phép xoay DRL – Double Right-to-Left

Ví dụ tạo cây AVL (1)

- Tạo cây AVL với các khóa lần lượt là: 30, 20, 10,…

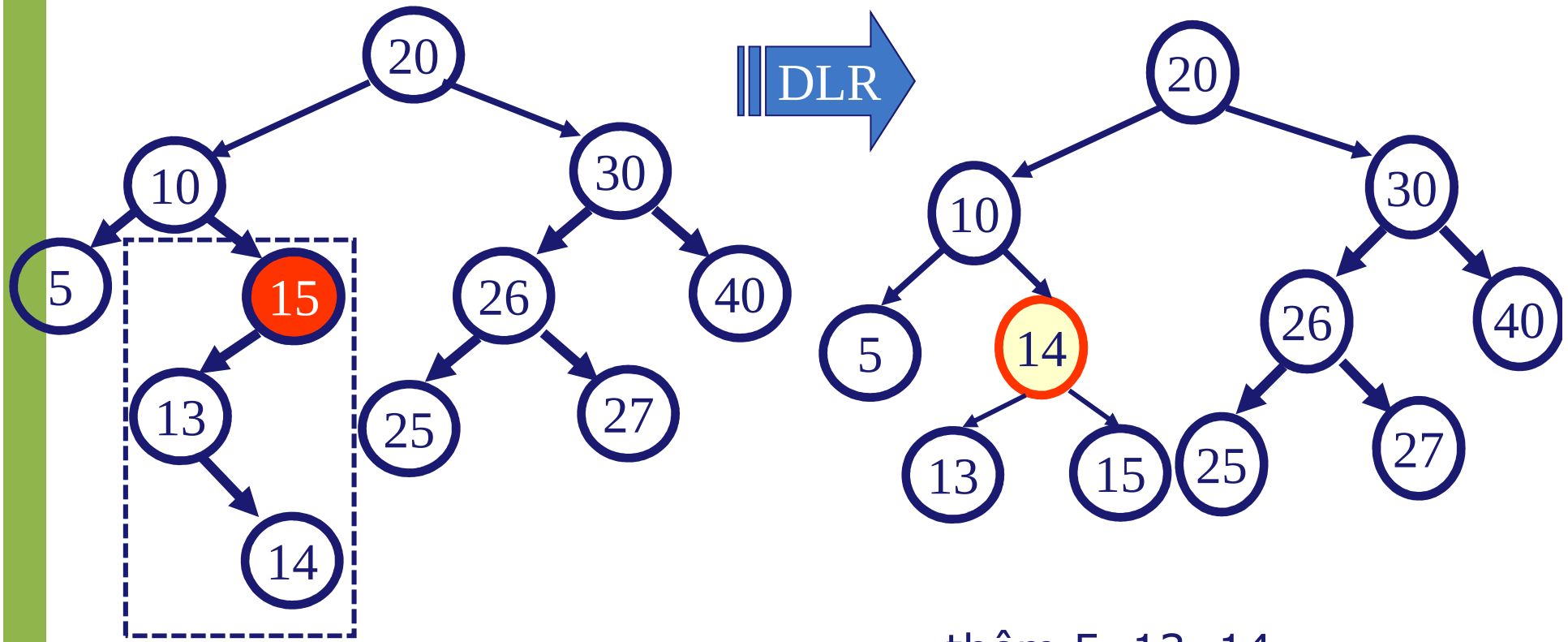


Ví dụ tạo cây AVL (2)



…thêm 15, 40, 25, 27, 26

Ví dụ tạo cây AVL (3)



… thêm 5, 13, 14

Đánh giá/so sánh

- Độ cao của cây: $h_{AVL} < 1.44*\log_2(N+1)$

  Cây AVL có độ cao nhiều hơn không quá 44% so với độ cao của 1 cây nhị phân tối ưu.

- Chi phí tìm kiếm $O(\log_2 N)$

- Chi phí thêm phần tử $O(\log_2 N)$
  - Tìm kiếm: $O(\log_2 N)$
  - Điều chỉnh cây: $O(\log_2 N)$

- Chi phí xóa phần tử $O(\log_2 N)$
  - Tìm kiếm: $O(\log_2 N)$
  - Điều chỉnh cây: $O(\log_2 N)$

Các cấu trúc dữ liệu nâng cao

(Advanced Data Structures)

Bảng băm – Hash Table

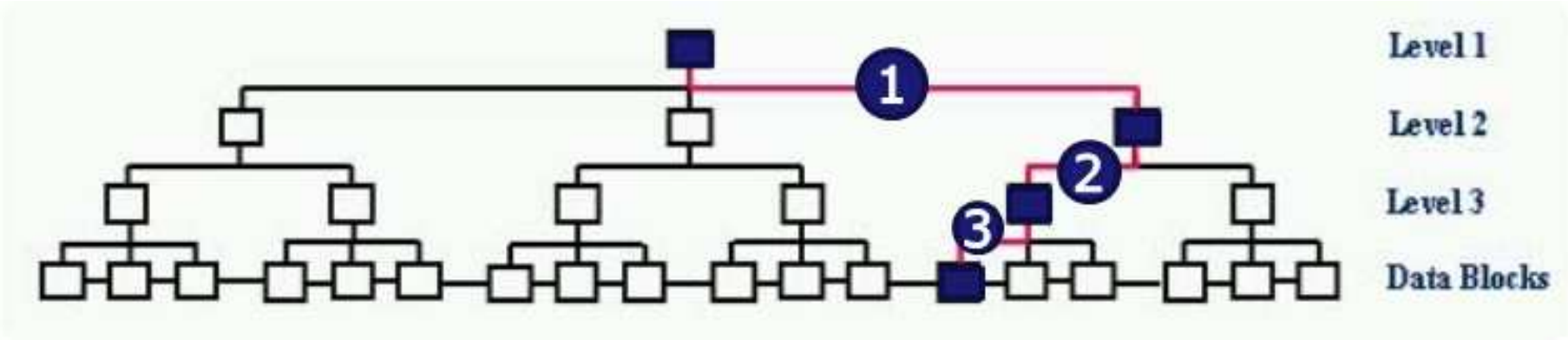
- Giới thiệu
- Direct-address table
- Bảng băm
- Khai báo cấu trúc Hash Table
- Xung đột địa chỉ
- Hàm băm
- Các phương pháp xử lý xung đột

Giới thiệu (1)

- Bài toán:
  - Cho một tập các khóa (key).
  - Nhu cầu chủ yếu là tìm kiếm (thêm, xóa ít khi xảy ra)
  - Cách tổ chức lưu trữ và tìm kiếm với chi phí thấp ?

Giới thiệu (3)

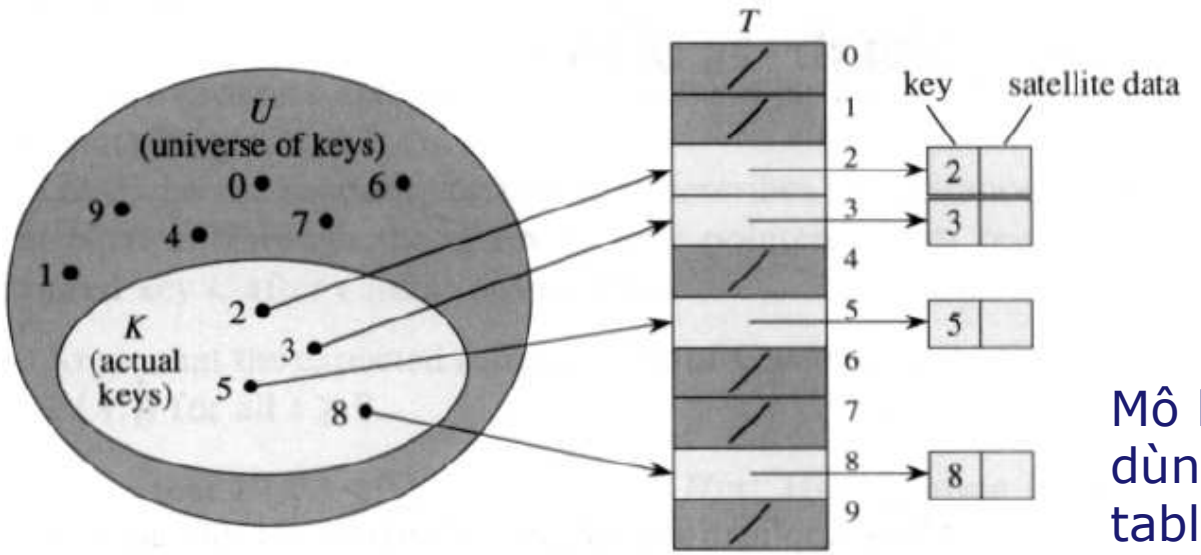
- Các cấu trúc dữ liệu đã biết:
  - Mảng, Danh sách liên kết, BST,… tìm kiếm bằng cách so sánh lần lượt các phần tử → thời gian tìm kiếm không nhanh và phụ thuộc N (số phần tử)



Cây bậc 3 → chi phí tìm kiếm $O(\log_3 N)$

Direct-address table (1)

- Giả sử có một tập khoá U:
  - Kích thước không quá lớn
  - Các giá trị khoá phân biệt
  - VD. U = {0, 1, 2, …, 9}



Mô hình minh họa dùng direct-address table T[m] để lưu trữ các khoá trong tập U

Direct-address table (2)

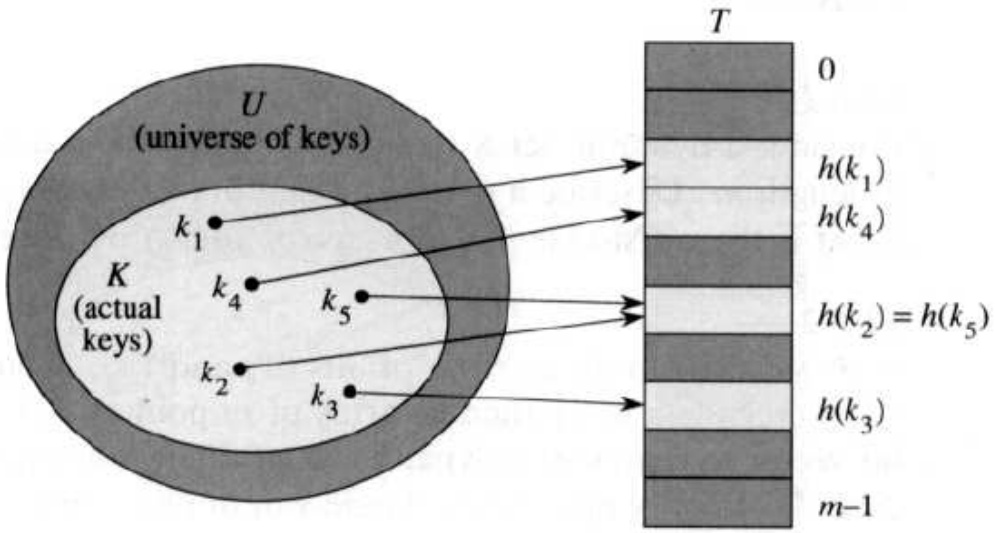
- Direct-address table:
  - Một mảng T[m] (T[0],…,T[m-1]) để chứa các khoá trong tập U
  - |T| = |U|
  - Mỗi vị trí T[k] (slot) sẽ chứa:
    - Khóa k, hay
    - NULL nếu khoá k không có trong tập hợp
- Lưu ý:
  - U (Universe of keys): tập các giá trị khóa
  - K (Actual keys): tập các khoá thực sự được dùng
- Chi phí thao tác: O(1)

Direct-address table (3)

- Các giới hạn của direct-address table:
  - Kích thước tập U quá lớn → không thể tạo bảng T với số slot tương ứng với |U|
  - Kích thước của tập K quá nhỏ so với U → rất nhiều slot bị bỏ trống

Bảng băm (1)

- Khi tập khóa K nhỏ hơn nhiều (VD) so với tập U → ta chỉ dùng mảng T[m] với kích thước vừa đủ cho tập K
  - m = Θ(|K|)
- Do đó, không thể áp dụng ánh xạ trực tiếp T[k] ← k được nữa



Thay vì ánh xạ trực tiếp T[k] ← k, ta dùng hàm băm h để ánh xạ T[h(k)] ← k

Bảng băm (2)

- *Hàm băm* h: dùng để ánh xạ các khoá của tập U vào những slot của *bảng băm* T[0..m-1]

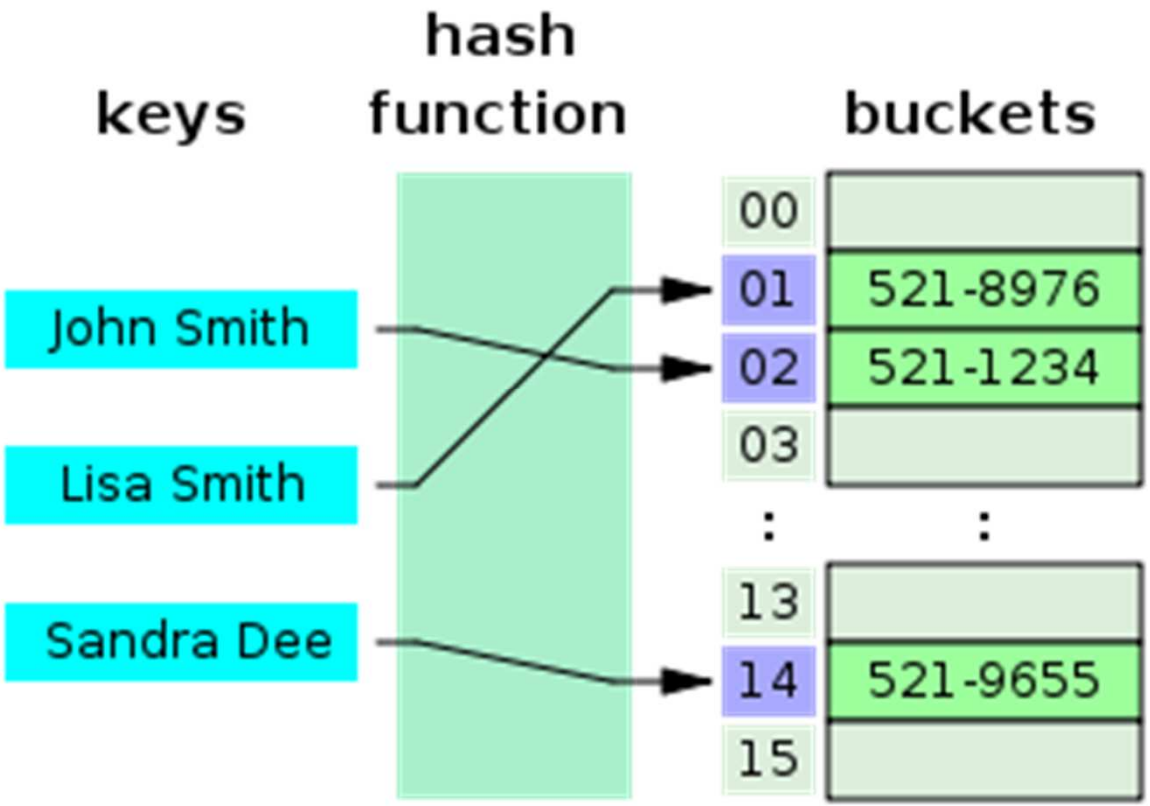
$$h : U \rightarrow \{0, 1, \ldots, m-1\} \ .$$

- h(k): giá trị băm (hash value) của khoá k

Bảng băm (3)

- Định nghĩa bảng băm:
  - Bảng băm là một cấu trúc dữ liệu, lưu trữ các khóa trong bảng T (danh sách đặc); sử dụng một *hàm băm (hash function)* để ánh xạ khoá (key) với một địa chỉ lưu trữ
  - *Hàm băm* có tác dụng biến đổi khoá thành *chỉ số địa chỉ (index) – tương ứng với khoá*
- Bảng băm là cấu trúc rất phù hợp để cài đặt cho bài toán "từ điển (dictionary)"
  - Dictionary: dạng bài toán chỉ chủ yếu sử dụng thao tác *chèn thêm (Insert)* và *tìm kiếm (Search)*

Bảng băm (4)



Hàm băm – biến đổi khoá thành địa chỉ index

Bảng băm (5)

- Các tính chất:
  - Cấu trúc lưu trữ dùng trong Hash table thường là danh sách đặc: mảng hay file
  - Thao tác cơ bản được cung cấp bởi Hash table là *tìm kiếm (lookup)*
  - Chi phí trung bình là O(1)
  - Chi phí tìm kiếm xấu nhất (ít gặp) có thể là O(n)

Khai báo cấu trúc Hash Table

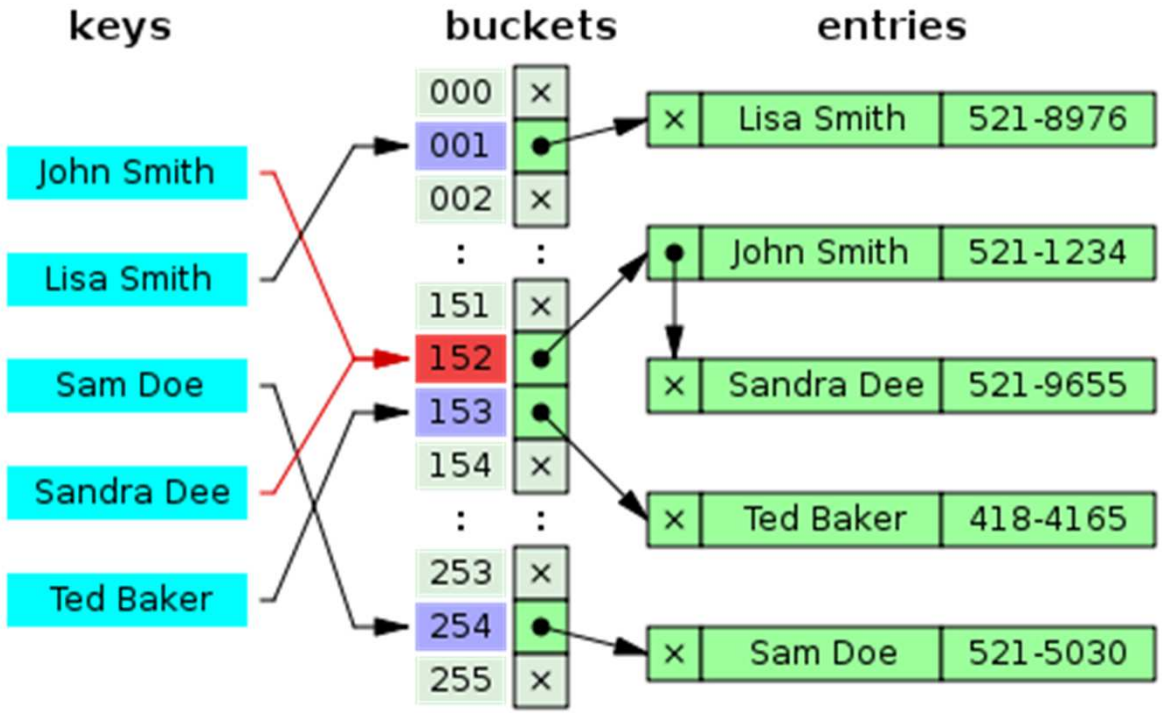
```
template <class T> class HASH_TABLE {
   private:
       T            *items;  // array of hash items
       int          maxSize; // maximum size of hash table

   public:
       HASH_TABLE(int size);      // create hash table with
                                  // 'size' items
       HASH_TABLE(const HASH_TABLE &aHashTable);
       ~HASH_TABLE ();            // destructor

       // operations
       bool   insert(T newItem);
       bool   remove(T key);
       bool   retrieve(T key, T &item);
}; // end class
```

Xung đột địa chỉ (1)

- Một cách lý tưởng, hàm băm sẽ ánh xạ mỗi khoá vào một slot riêng biệt của bảng T
- Tuy nhiên, điều này trong thực tế khó đạt được, vì:
  - m << |U|
  - Các khoá là không biết trước

Xung đột địa chỉ (2)

- Hầu hết cấu trúc bảng băm trong thực tế đầu chấp nhận *một tỉ lệ nhỏ các khoá đụng độ* và xây dựng phương án giải quyết sự đụng độ đó



Minh họa sự đụng độ và phương án giải quyết "chaining (móc xích)"

Hàm băm (1)

- Thành phần quan trọng nhất của bảng băm là "hàm băm"
- Nhiệm vụ của hàm băm là biến đổi khóa k của phần tử thành địa chỉ trong bảng băm.
  - Khóa có thể là dạng số hay dạng chuỗi
- Phương án xử lý chính của hàm băm là xem các khoá như là các số nguyên
  - Khóa là chuỗi "key" → xử lý với 3 thành phần 107 (k), 101 (e), 121 (y)

Hàm băm (2)

- Một hàm băm tốt là yếu tố tiên quyết để tạo ra bảng băm hiệu quả
- Các yêu cầu cơ bản đối với hàm băm:
  - Tính toán nhanh, dễ dàng
  - Các khóa được phân bố đều trong bảng
  - Ít xảy ra đụng độ

Hàm băm (3)

- Các phương pháp xây dựng hàm băm:
  - Cắt bỏ (truncation)
  - Gấp (folding)
  - Áp dụng các phép tính toán
    - Phép chia modular
    - Phương pháp nhân

Hàm băm (4)

- Xây dựng hàm băm – phương pháp chia:
  - h(k) = k mod m
  - VD. h(k) = k mod 11
- Chọn m như thế nào ?
  - m không được là lũy thừa của 2. Nếu m = 2^p thì h(k) = k mod m chính là p bit thấp của k
  - m không nên là lũy thừa của 10, vì khi đó, hash value sẽ không sử dụng tất cả chữ số thành phần của k
  - Nên chọn m là số nguyên tố nhưng không quá gần với giá trị 2^n

Hàm băm (5)

- Xây dựng hàm băm – phương pháp nhân:
  - h(k) = ⌊ m * (k*A mod 1) ⌋
  - Trong đó:

    0 < A < 1

    (k*A mod 1) là phần thập phân của k*A

    ⌊ x ⌋ là floor(x)
  - Ở phương pháp này, giá trị m không quan trọng, ta thường chọn m = 2^p
  - Knuth đã phân tích và đưa ra một giá trị A tối ưu:

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$$

Hàm băm (6)

- Ví dụ phương pháp nhân:
  - Giả sử ta có k = 123456; m = 10000; A như trên

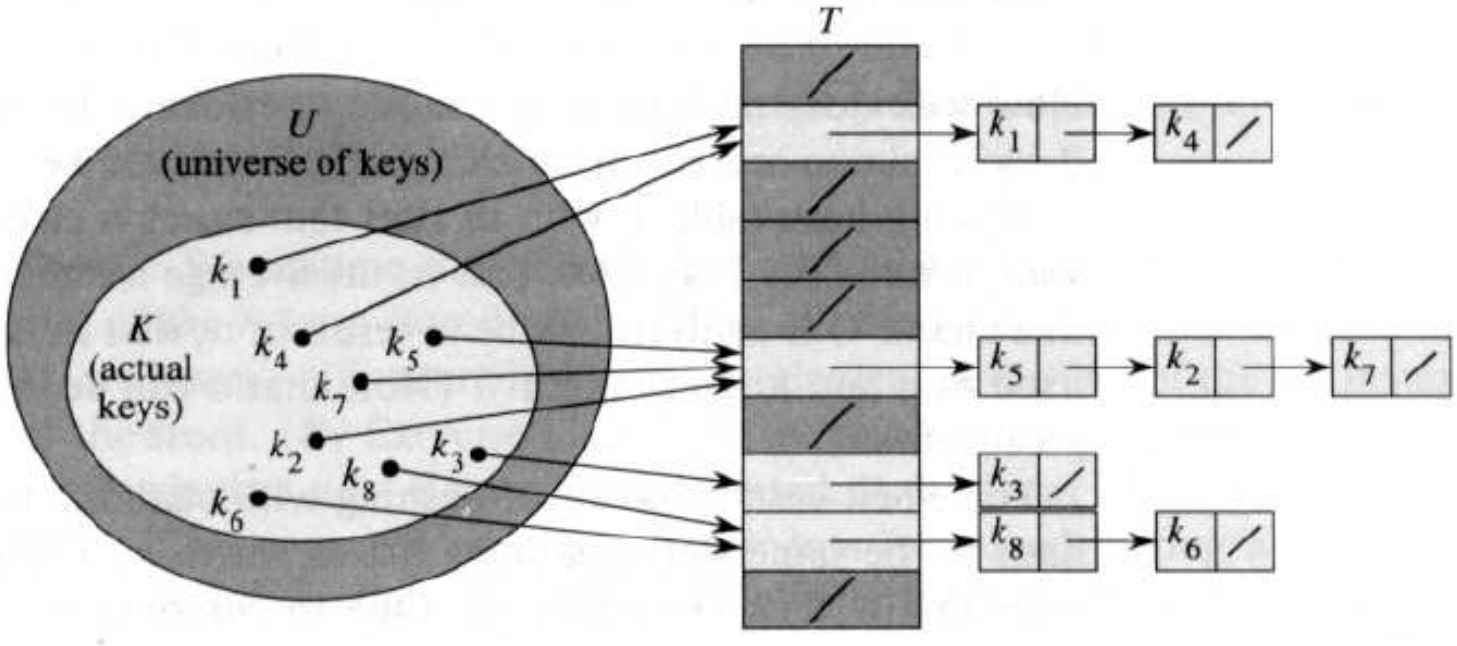
$$
\begin{aligned}
h(k) &= \lfloor 10000 \cdot (123456 \cdot 0.61803\ldots \bmod 1) \rfloor \\
&= \lfloor 10000 \cdot (76300.0041151\ldots \bmod 1) \rfloor \\
&= \lfloor 10000 \cdot 0.0041151\ldots \rfloor \\
&= \lfloor 41.151\ldots \rfloor \\
&= 41 .
\end{aligned}
$$

Các phương pháp xử lý xung đột

- Phương pháp nối kết (Separate chaining)
- Phương pháp địa chỉ mở (Open addressing)

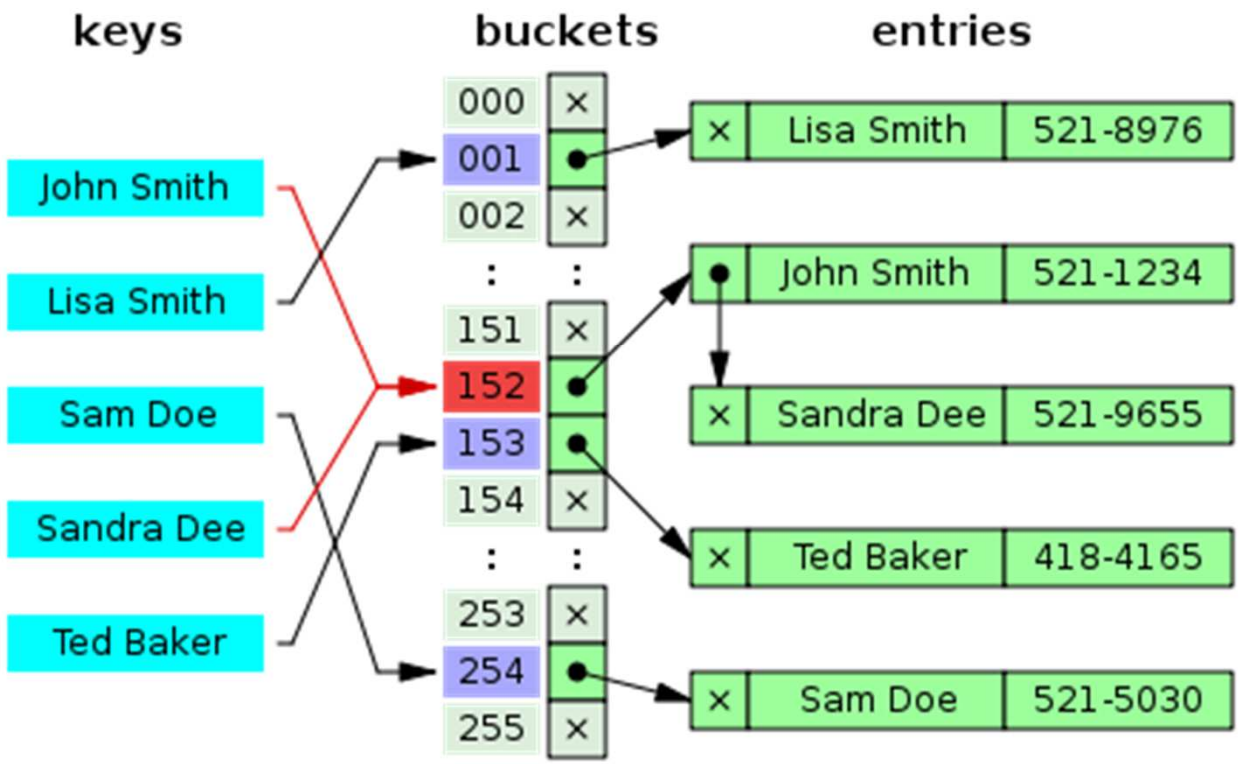
Phương pháp nối kết (1)

- Đưa tất cả các khóa đụng độ vào một slot, lưu thành một linked-list



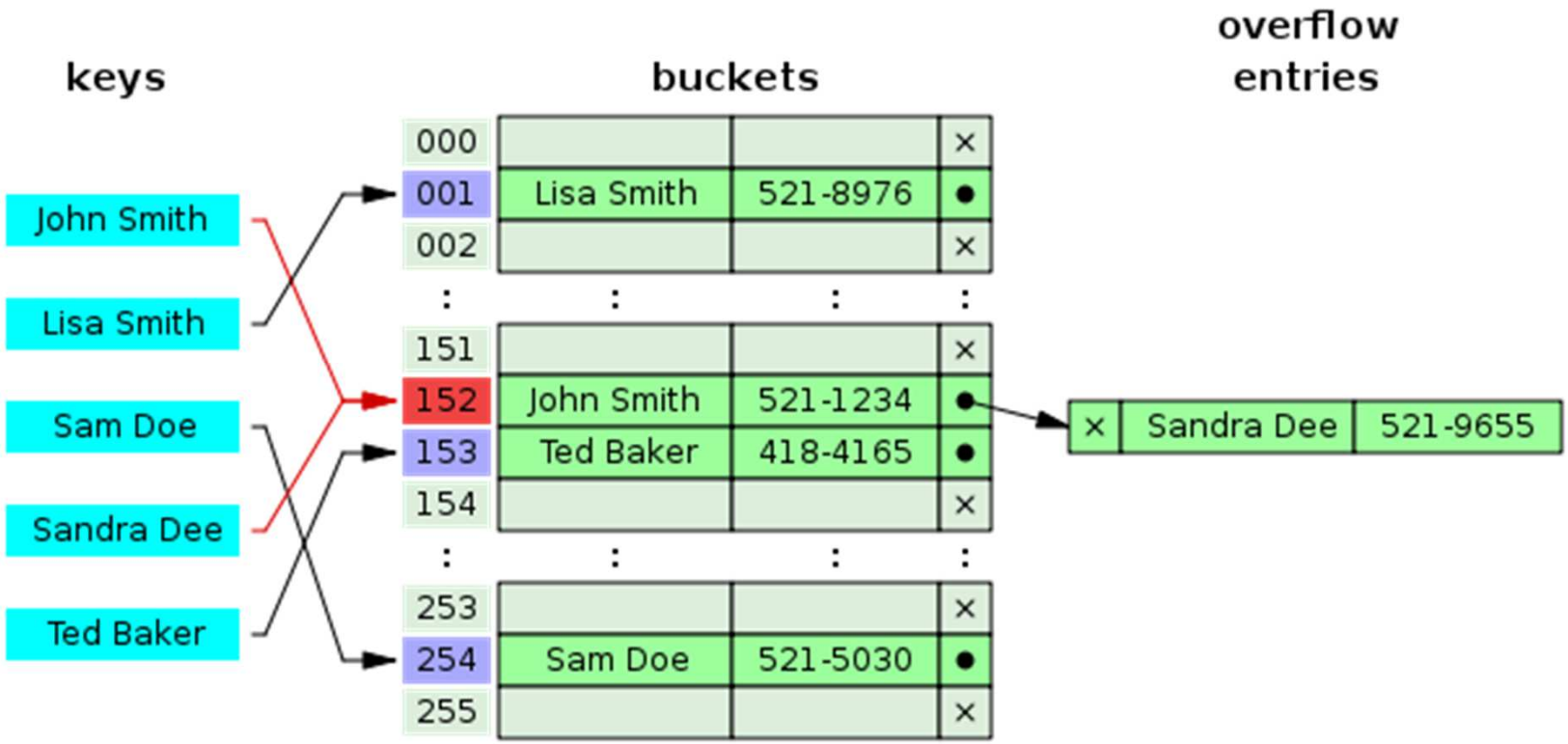
Mô hình cách xử lý đụng độ bằng phương pháp chaining

Phương pháp nối kết (2)



Phương pháp chaining – bảng T chỉ lưu con trỏ của linked-list

Phương pháp nối kết (3)



Phương pháp chaining –
bảng T lưu phần tử đầu tiên + con trỏ của linked-list

Phương pháp nối kết (4)

- Chi phí các thao tác:
  - Insert: chi phí xấu nhất là O(1)
  - Search và Delete: chi phí trung bình là $\Theta(1+\alpha)$

    $\alpha = n/m$ (load factor: số phần tử trung bình lưu trữ trong một slot)

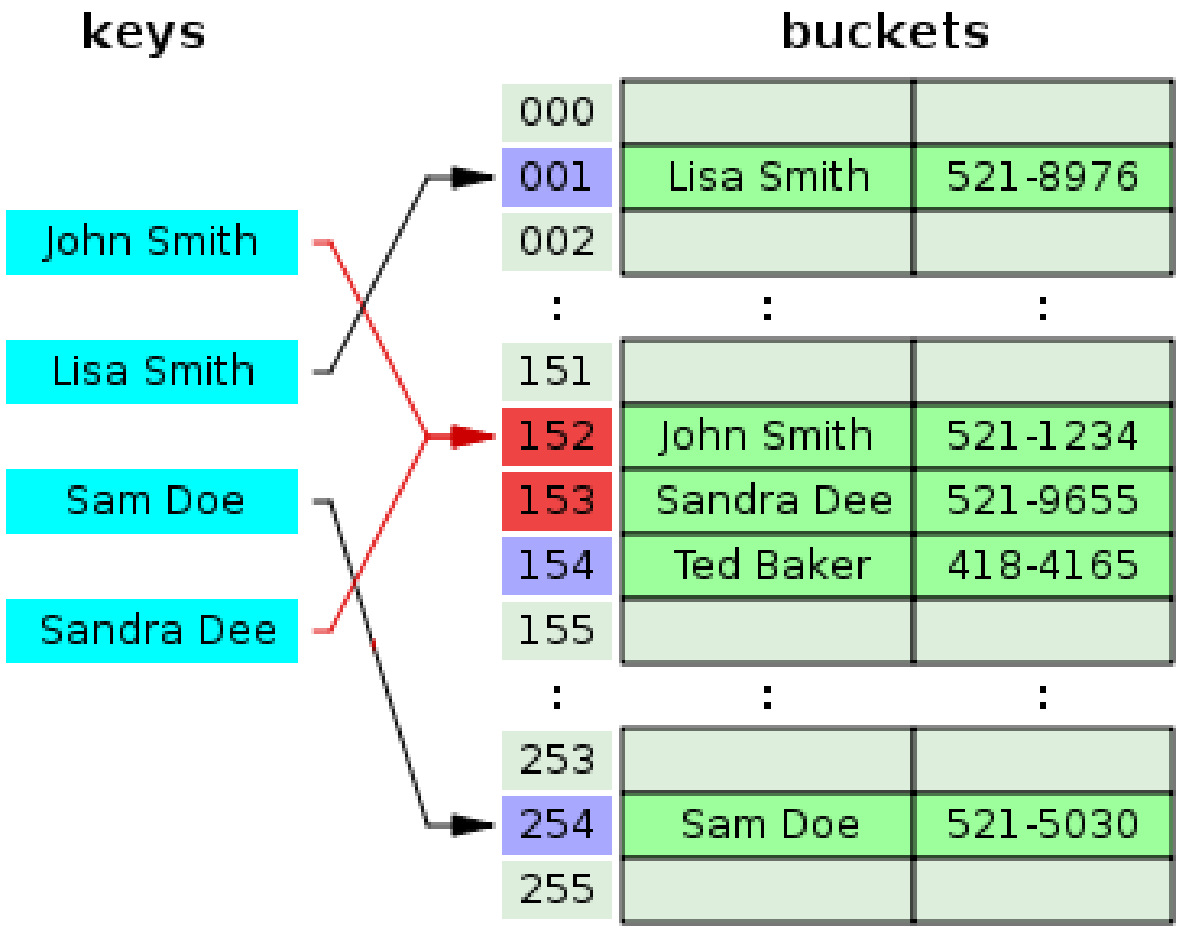
- Các cấu trúc dữ liệu khác:
  - Ngoài linked-list, ta có thể áp dụng các cấu trúc khác hiệu quả hơn (khi tìm kiếm) như: cây cân bằng (AVL, Red-Black, AA), hay mảng cấp phát động,…

Phương pháp địa chỉ mở (1)

- Các phần tử chỉ lưu trong bảng T, không dùng thêm bộ nhớ mở rộng như phương pháp nối kết
- Thuật toán cơ bản để thêm khóa k:

```
HASH-INSERT(T, k)
1  i ← 0
2  repeat j ← h(k, i)
3         if T[j] = NIL
4            then T[j] ← k
5                 return j
6            else i ← i + 1
7    until i = m
8  error "hash table overflow"
```

Phương pháp địa chỉ mở (2)



Phương pháp Open addressing – Linear probing

Phương pháp địa chỉ mở (3)

- Thuật toán cơ bản để tìm khóa k:

```
HASH-SEARCH(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = k
4             then return j
5          i ← i + 1
6     until  T[j] = NIL or i = m
7  return NIL
```

Phương pháp địa chỉ mở (4)

- Tên gọi "open addressing" mang ý nghĩa là địa chỉ (address) của phần tử không phải chỉ được xác định bằng "duy nhất" hash value của phần tử đó, mà còn có sự can thiệp của phép "dò tìm (probing)"

- Có 3 phương pháp dò tìm phổ biến:
  - Phương pháp dò tuần tự (Linear probing)
  - Phương pháp dò bậc 2 (Quadratic probing)
  - Phương pháp băm kép (Double hashing)

Linear probing

- Mô tả:

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

  - i: thứ tự của lần thử (i = 0, 1, 2,…)
  - h(k): hàm băm
  - m: số slot của bảng băm
  - h(k, i): địa chỉ của khóa ***k*** tại lần thử thứ ***i***

Quadratic probing

- Mô tả:

$$h(k, i) = (h(k) + i^2) \bmod m$$

  - i: thứ tự của lần thử (i = 0,1,2,…)
  - h(k): hàm băm
  - m: số slot của bảng băm
  - h(k, i): địa chỉ của khóa ***k*** tại lần thử thứ ***i***

Double hashing

- Mô tả:

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

  - i: thứ tự của lần thử (i = 0,1,2,…)
  - h(k) và h’ (k) : hàm băm
  - m: số slot của bảng băm
  - h(k, i): địa chỉ của khóa ***k*** tại lần thử thứ ***i***

Thảo luận

- Hãy so sánh các ưu, khuyết điểm của phương pháp chaining và open addressing

Ví dụ

- Bài tập:
  - Có 1 bảng băm T, chiều dài m = 11; hàm băm h(k) = k mod m
  - Cho một dãy phần tử theo thứ tự như sau:

    10, 22, 31, 4, 15, 28, 17, 88, 59
  - Hãy trình bày kết quả khi thêm các phần tử trên vào bảng băm, với lần lượt từng phương pháp xử lý đụng độ:
    - Nối kết (Chaining)
    - Dò tuần tự (Linear probing)
    - Dò bậc 2 (Quadratic probing)
    - Băm kép (Double hashing), với h’(k) = 1 + (k mod (m – 1))