Chương 6

TÌM KIẾM

NỘI DUNG

6.1. Tìm kiếm tuần tự và tìm kiếm nhị phân

6.2. Cây nhị phân tìm kiếm

6.3. Cây AVL

6.4. Bảng băm

6.1. Tìm kiếm tuần tự và tìm kiếm nhị phân

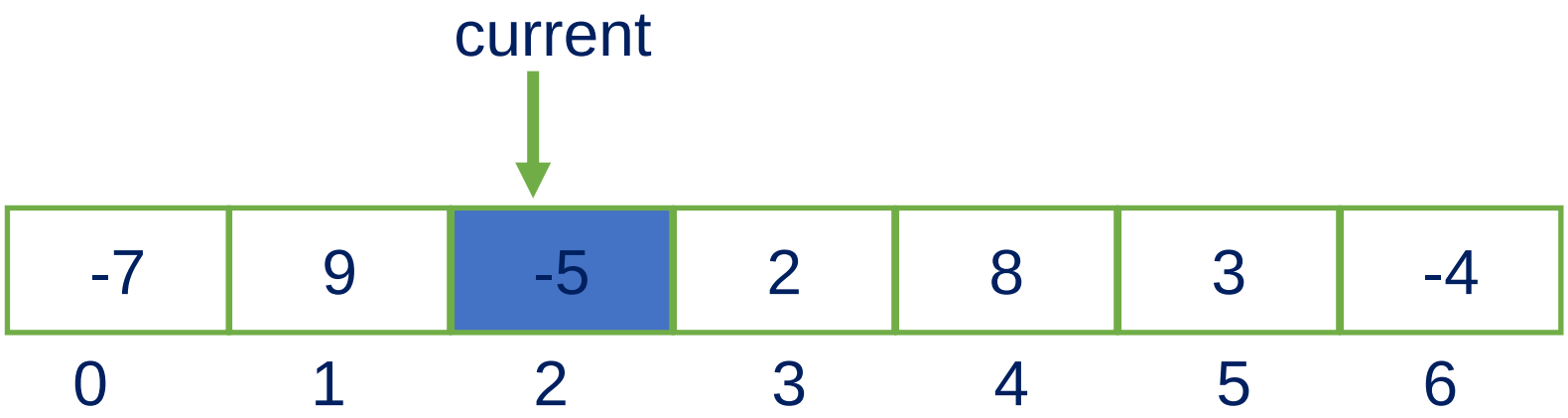
6.1.1. Tìm kiếm tuần tự (Linear Search or Sequential Search)

6.1.2. Tìm kiếm nhị phân

Bài toán tìm kiếm

- Cho danh sách ***a*** gồm *n* phần tử a_1, a_2, ..., a_n và một số *x*.
- *x* có mặt trong danh sách đã cho hay không?
- Nếu có, hãy đưa ra *vị trí xuất hiện của x trong dãy đã cho*
  - đưa ra chỉ số i sao cho $a_i = x$.

6.1.1. Tìm kiếm tuần tự



- Bắt đầu từ phần tử đầu tiên, duyệt qua từng phần tử cho đến khi tìm được đích hoặc kết luận không tìm được.
- Các số không cần sắp thứ tự
- Làm việc được với cả danh sách móc nối (Linked Lists)

Độ phức tạp: $O(n)$

Linear Search

```
int linearSearch(float a[], int size, int target)
{
    int i;
    for (i = 0; i < size; i++)
    {
        if (a[i] == target)
        {
            return i;
        }
    }
    return -1;
}
```

Phân tích thời gian tính

- Độ dài đầu vào là *n.*
- Đánh giá bởi số lần thực hiện

  (*) **(a[i] == target)**

  trong vòng lặp for.

- Nếu *a*[1] = *target* thì (*) phải thực hiện 1 lần.

  -> thời gian tính tốt nhất : $\Theta(1)$.

- Nếu *target* không có mặt trong dãy thì (*) phải thực hiện *n* lần.

  ->thời gian tính tồi nhất :$\Theta(n)$.

Phân tích thời gian tính

- Nếu *target* tìm thấy ở vị trí thứ *i* của dãy (*target* = *a*[*i*]) thì (*) phải thực hiện *i* lần (*i* = 1, 2, ..., *n*).
- => số lần trung bình phải thực hiện phép so sánh (*) là

$$[(1 + 2 + \ldots + n) + n] / (n+1)$$

$$= [n + n(n+1)/2]/(n+1)$$

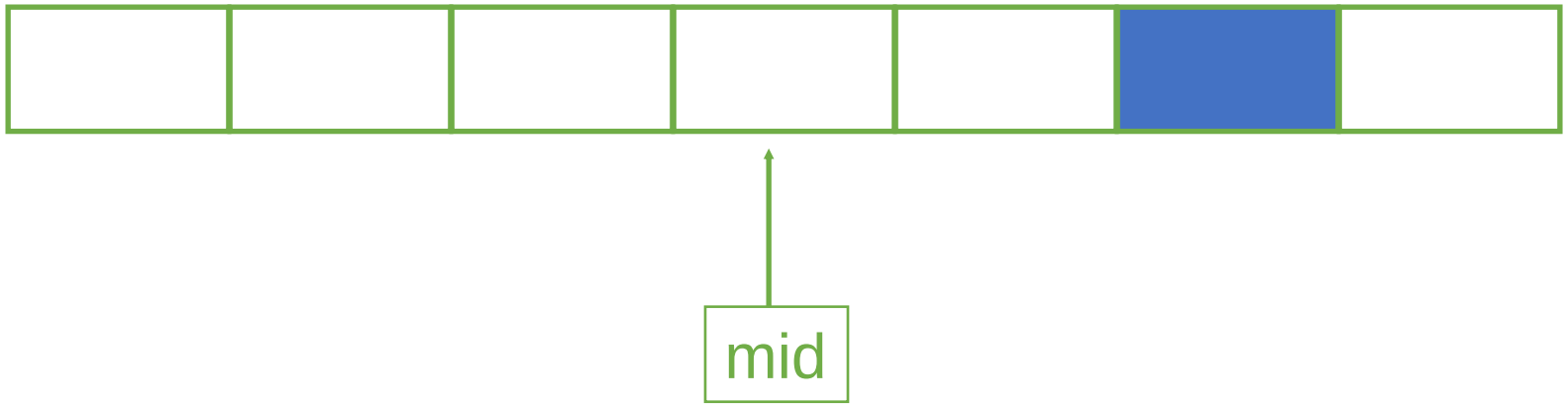
$$= (n^2 + 3n)/[2(n+1)] .$$

- Ta có:

$$n/4 \leq (n^2+3n)/[2(n+1)] \leq n$$

- →thời gian tính trung bình : $\Theta(n)$.

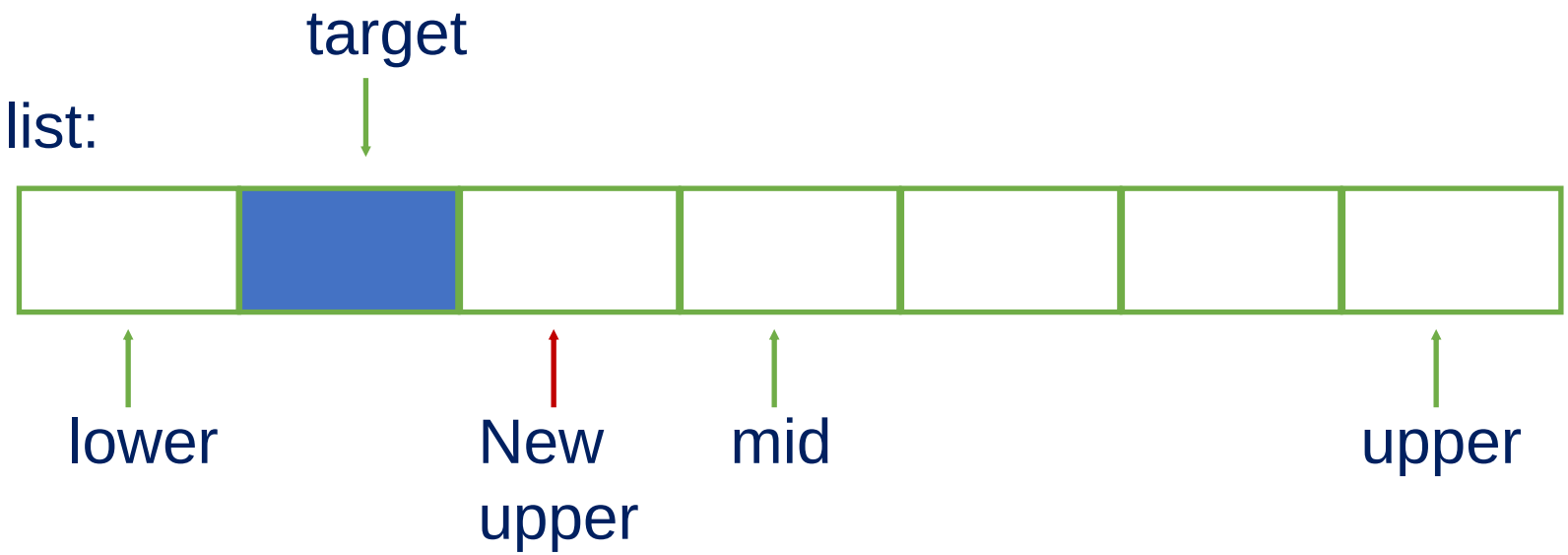
6.1.2. Tìm kiếm nhị phân- Binary Search



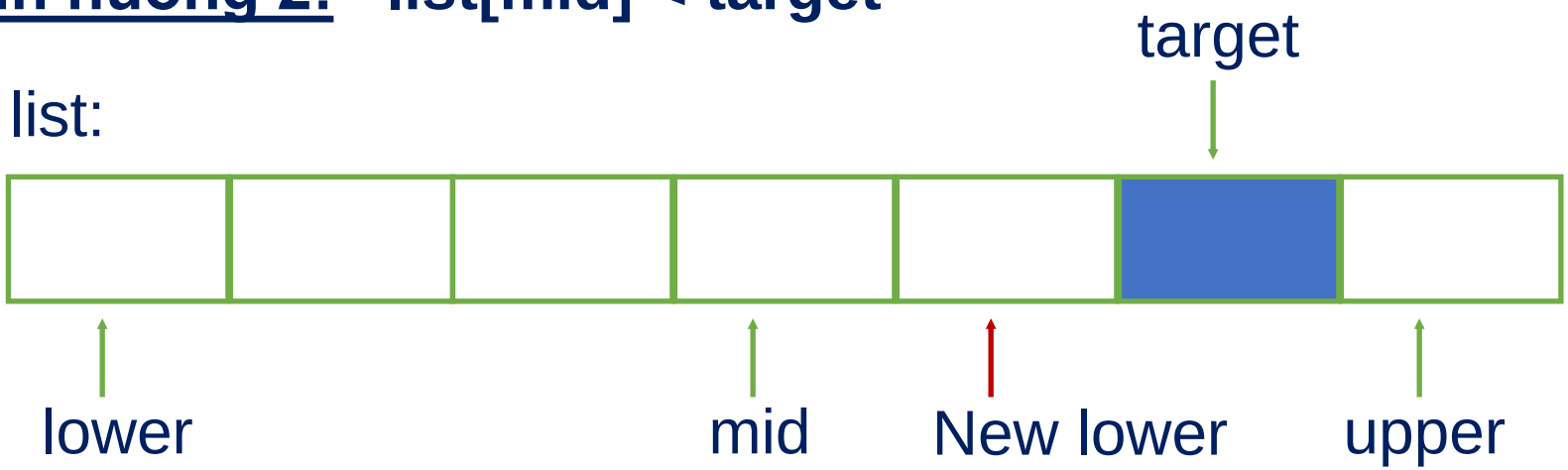
- **Điều kiện:**
  - Danh sách phải được sắp thứ tự.
  - Phải cho phép trực truy.

Độ phức tạp: $O(\log n)$

Tình huống 1: **target < list[mid]**



Tình huống 2: **list[mid] < target**



Cài đặt trên C

```
int binarySearch(float array[], int size, int target)
{
   int lower = 0, upper = size - 1, mid;

   while (lower <= upper) {
      mid = (upper + lower)/2;
      if (array[mid] > target)
      { upper = mid - 1; }
      else if (array[mid] < target)
      { lower = mid + 1; }
      else
      { return mid; }
   }
   return -1;
}
```

Đoạn cần khảo sát
có độ dài giảm đi một nửa
sau mỗi lần lặp

Độ phức tạp: $O(\log n)$

NỘI DUNG

6.1. Tìm kiếm tuần tự và tìm kiếm nhị phân

6.2. Cây nhị phân tìm kiếm

6.3. Cây AVL

6.4. Bảng băm

6.2. Cây nhị phân tìm kiếm

Binary Search Tree

6.2.1. Định nghĩa

6.2.2. Biểu diễn cây nhị phân tìm kiếm

6.2.3. Các phép toán

6.2.1. Định nghĩa cây nhị phân tìm kiếm
Binary Search Trees

Cây nhị phân có các tính chất sau:

- **Mỗi nút *x* (ngoài thông tin đi kèm) có các trường:**

key: **khoá (**thường giả thiết là khoá của các nút là khác nhau từng đôi, trái lại nếu có khoá trùng nhau thì cần chỉ rõ thứ tự của hai khoá trùng nhau).

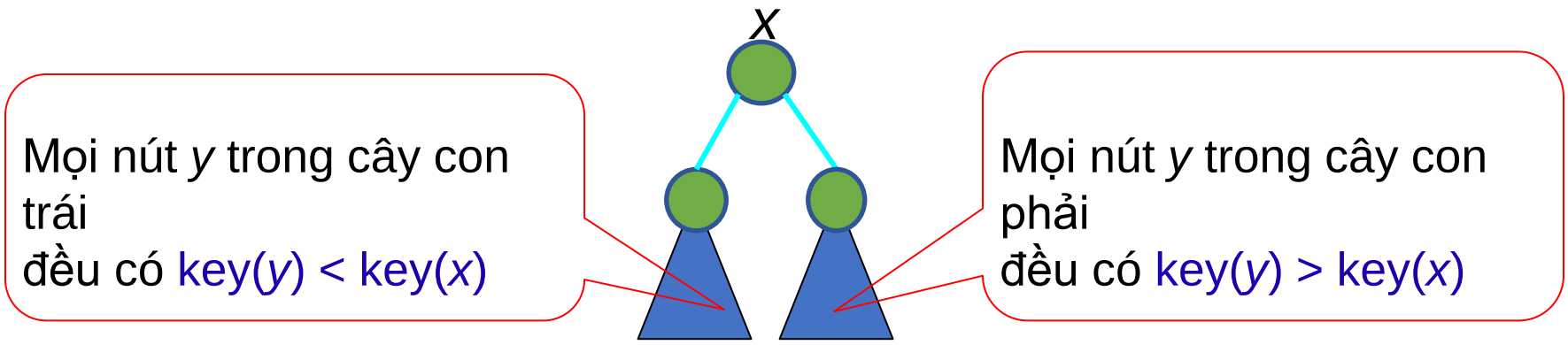
Tính chất BST

- **Tính chất BST (Binary-search-tree):**

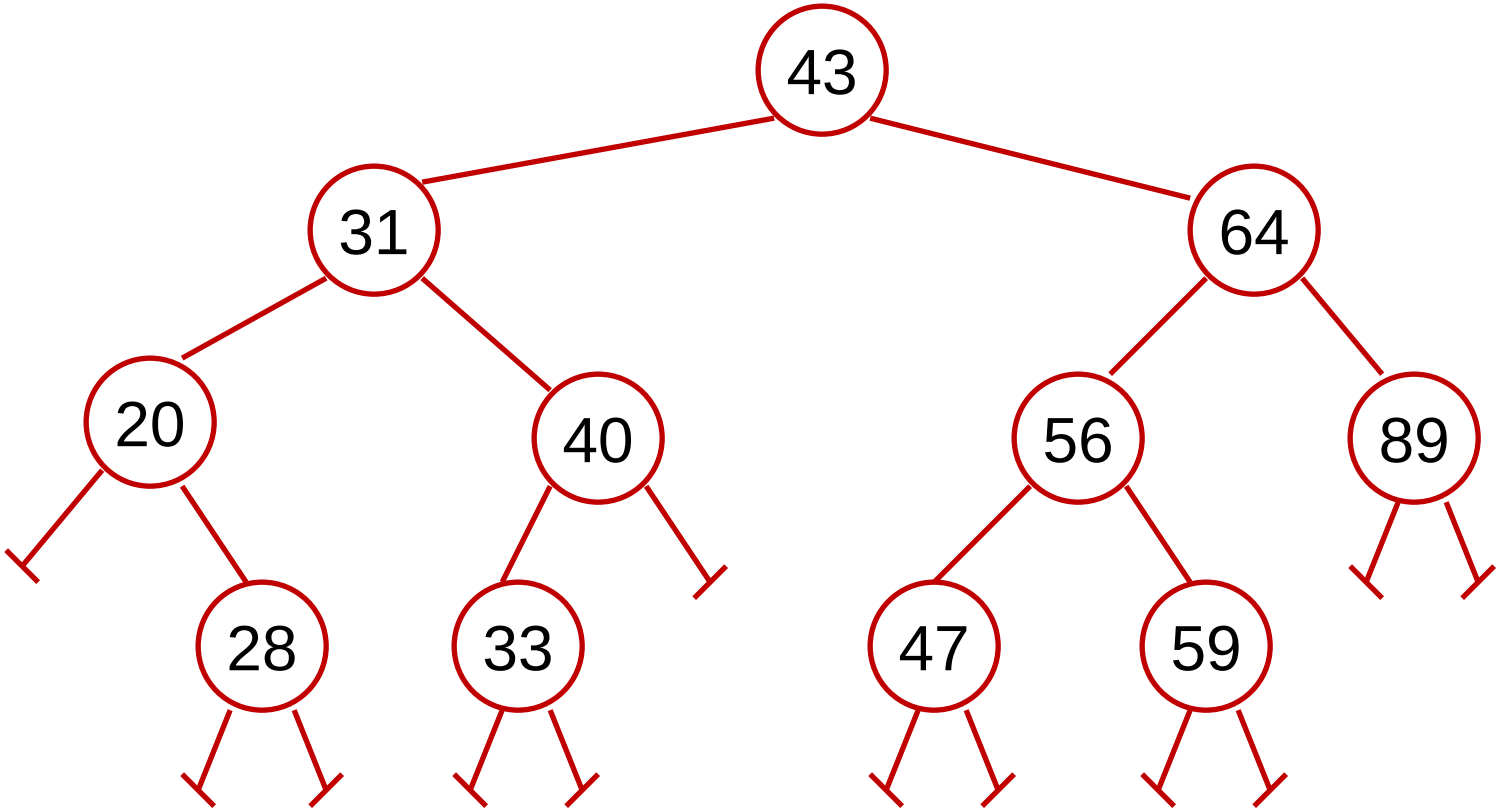
  Giả sử *x* là gốc của một cây con:

  - ∀ *y* thuộc cây con trái của *x*: *key*(*y*) < *key*(*x*).
  - ∀ *y* thuộc cây con phải của *x*: *key*(*y*) > *key*(*x*).

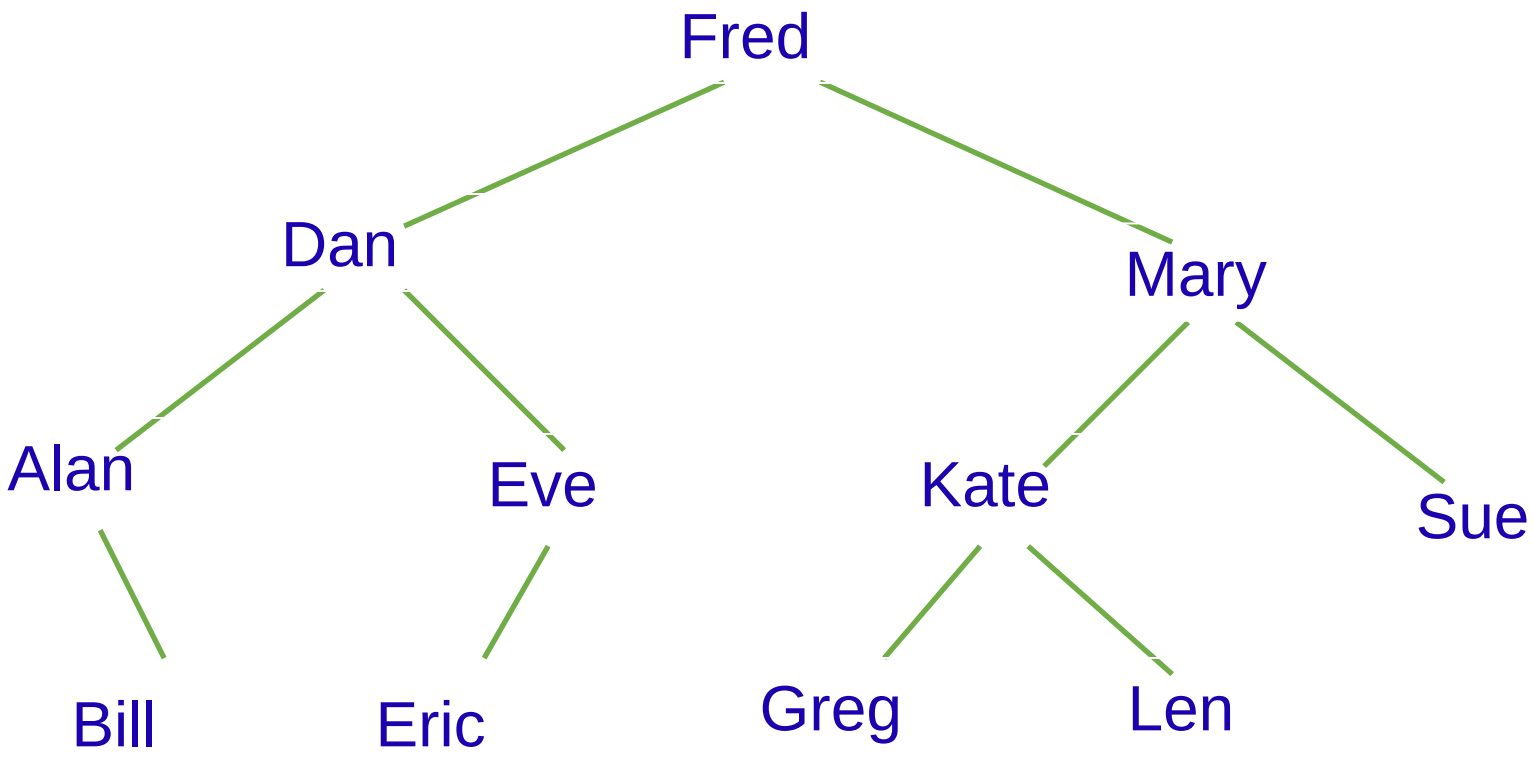
  (**Tất cả** các khoá của các nút trong **cây con trái (phải)** của *x* đều **nhỏ hơn (lớn hơn)** khoá của *x*.)



Ví dụ 1: *khoá là số nguyên*



Ví dụ 2: *khoá là xâu ký tự*



6.2. Cây nhị phân tìm kiếm

Binary Search Tree

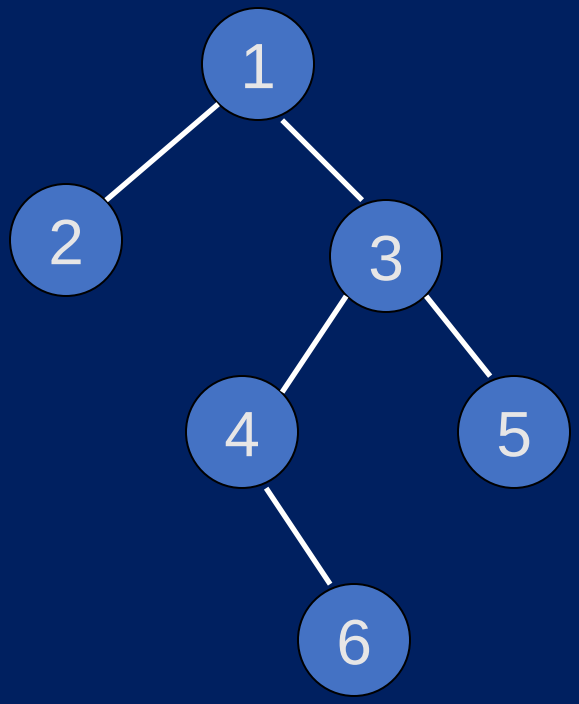
6.2.1. Định nghĩa

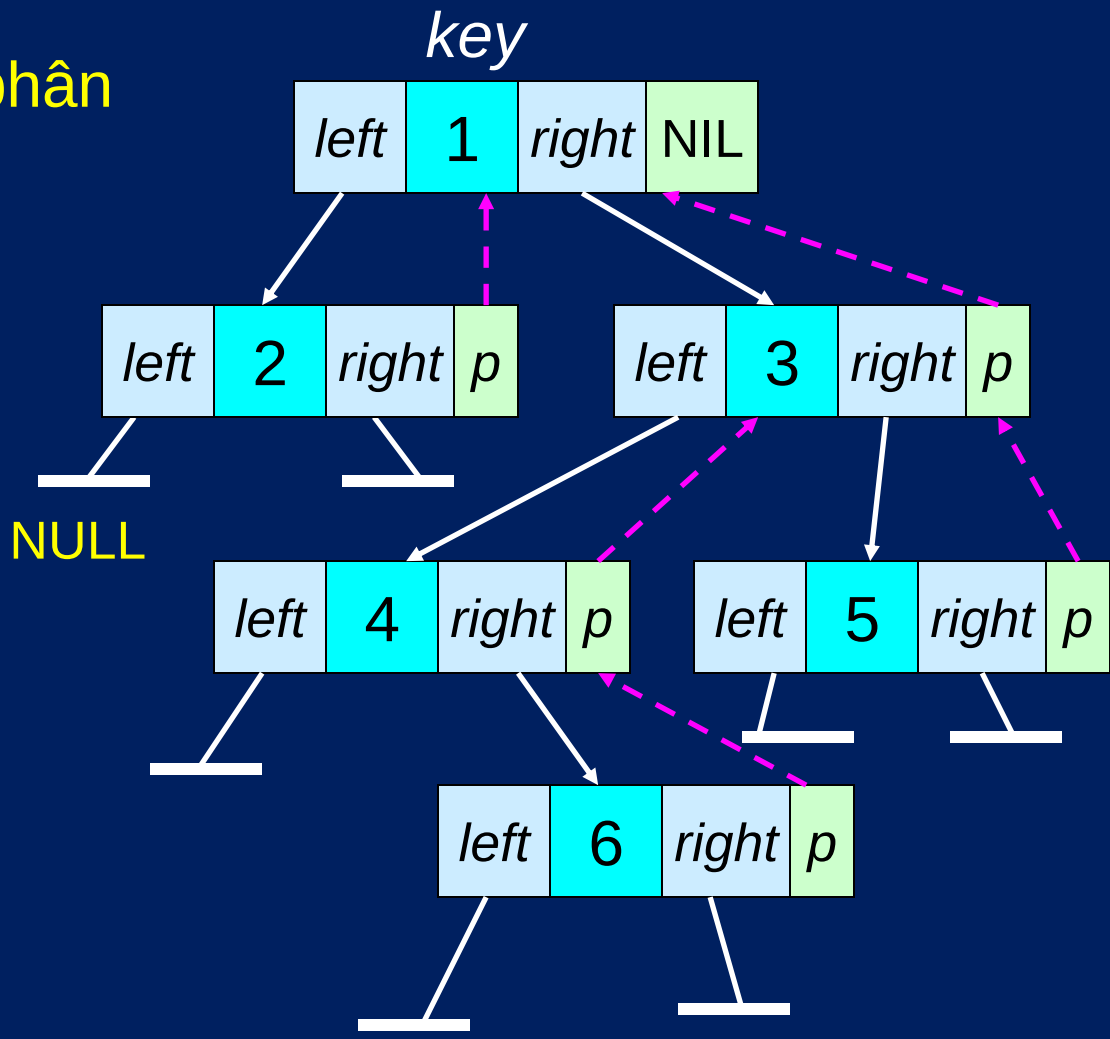
6.2.2. Biểu diễn cây nhị phân tìm kiếm

6.2.3. Các phép toán

6.2.2. Biểu diễn BST

Sử dụng cấu trúc cây nhị phân





Binary Search Tree Node

***Ví dụ* 1: Khoá là số nguyên**

```
struct TreeNodeRec
{
    int     key;

    struct TreeNodeRec*  leftPtr;
    struct TreeNodeRec*  rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

Binary Search Tree Node

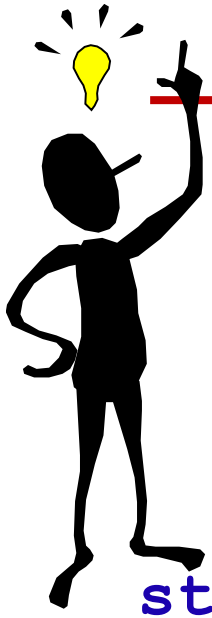
***Ví dụ 2*: Khoá là xâu ký tự**

```
#define MAXLEN 15

struct TreeNodeRec
{
    char    key[MAXLEN];

    struct TreeNodeRec*  leftPtr;
    struct TreeNodeRec*  rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

Chú ý:

```
#define MAXLEN 15

struct TreeNodeRec
{
    char    key[MAXLEN];

    struct TreeNodeRec*   leftPtr;
    struct TreeNodeRec*   rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

độ dài lớn nhất của xâu là cố định



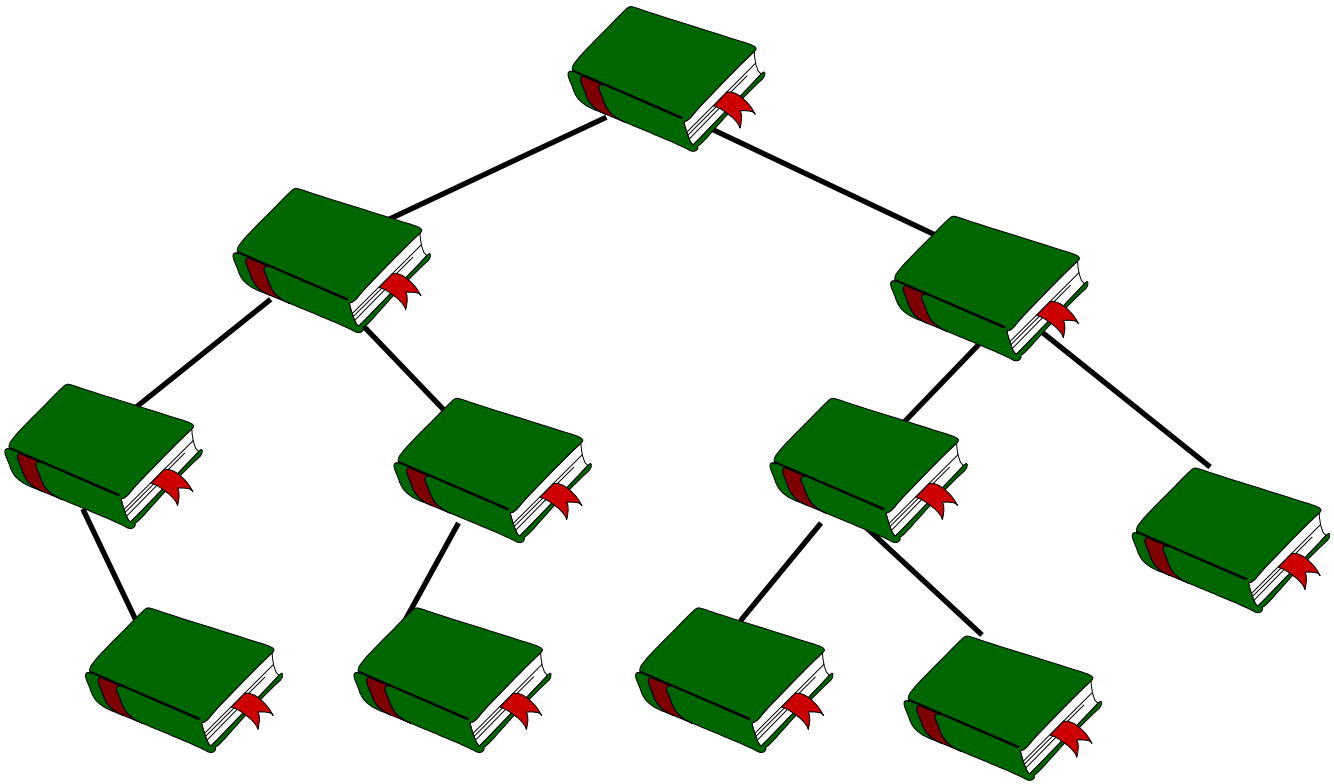
Chú ý:

- *Cho phép dùng xâu độ dài tuỳ ý.*
- *Cần cấp phát bộ nhớ trước khi dùng.*
- *Sử dụng* **`strcmp`** *để so sánh hai xâu.*

```
struct TreeNodeRec
{
    char*  key;

    struct TreeNodeRec*  leftPtr;
    struct TreeNodeRec*  rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

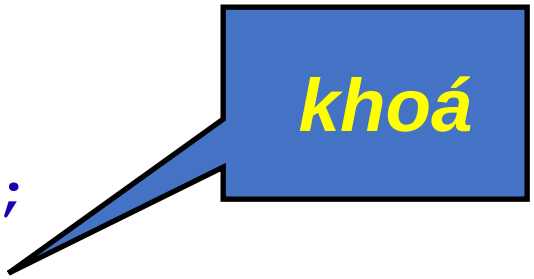
Ví dụ 4. Thông tin đi kèm:

Tổ chức lưu trữ thông tin về các cuốn sách hỗ trợ các thao tác tra cứu của bạn đọc



Giả sử thông tin về sách được ghi nhận trong các bản ghi có cấu trúc:

```
struct BookRec
{
    char*  author;
    char*  title;
    char*  publisher;

    /* etc.: other book information. */
};

typedef struct BookRec Book;
```



***Ví dụ* 4:** Khi đó ta có thể mô tả Binary Search Tree Node như sau

```
struct TreeNodeRec
{
   Book   info;

   struct TreeNodeRec*  leftPtr;
   struct TreeNodeRec*  rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

6.2. Cây nhị phân tìm kiếm

Binary Search Tree

6.2.1. Định nghĩa

6.2.2. Biểu diễn cây nhị phân tìm kiếm

6.2.3. Các phép toán

Tree Node

Trong phần tiếp theo ta sử dụng mô tả nút sau đây:

```
struct TreeNodeRec
{
    float  key;

    struct TreeNodeRec*  leftPtr;
    struct TreeNodeRec*  rightPtr;
};

typedef struct TreeNodeRec TreeNode;
```

Các phép toán cơ bản

- **makeTreeNode(value)**
- **search(nodePtr, k)**
- **find_min(nodePtr)**
- **find_max:**
- **Successor(x):**
- **Predcessor(x):**
- **insert(nodePtr, float item)**
- **delete(nodePtr, item)**

- **Các hàm cơ bản đối với cây nhị phân :**
  - **void printInorder(nodePtr);**
  - **void printPreorder(nodePtr);**
  - **void printPostorder(nodePtr); . . .**

Mô tả trên C

```
struct TreeNodeRec
{
   float                key;
   struct TreeNodeRec*  leftPtr;
   struct TreeNodeRec*  rightPtr;
};

typedef struct TreeNodeRec TreeNode;

TreeNode* makeTreeNode(float value);
TreeNode* insert(TreeNode* nodePtr, float item);
TreeNode* search(TreeNode* nodePtr, float item);
void printInorder(const TreeNode* nodePtr);
void printPreorder(const TreeNode* nodePtr);
void printPostorder(const TreeNode* nodePtr);
```

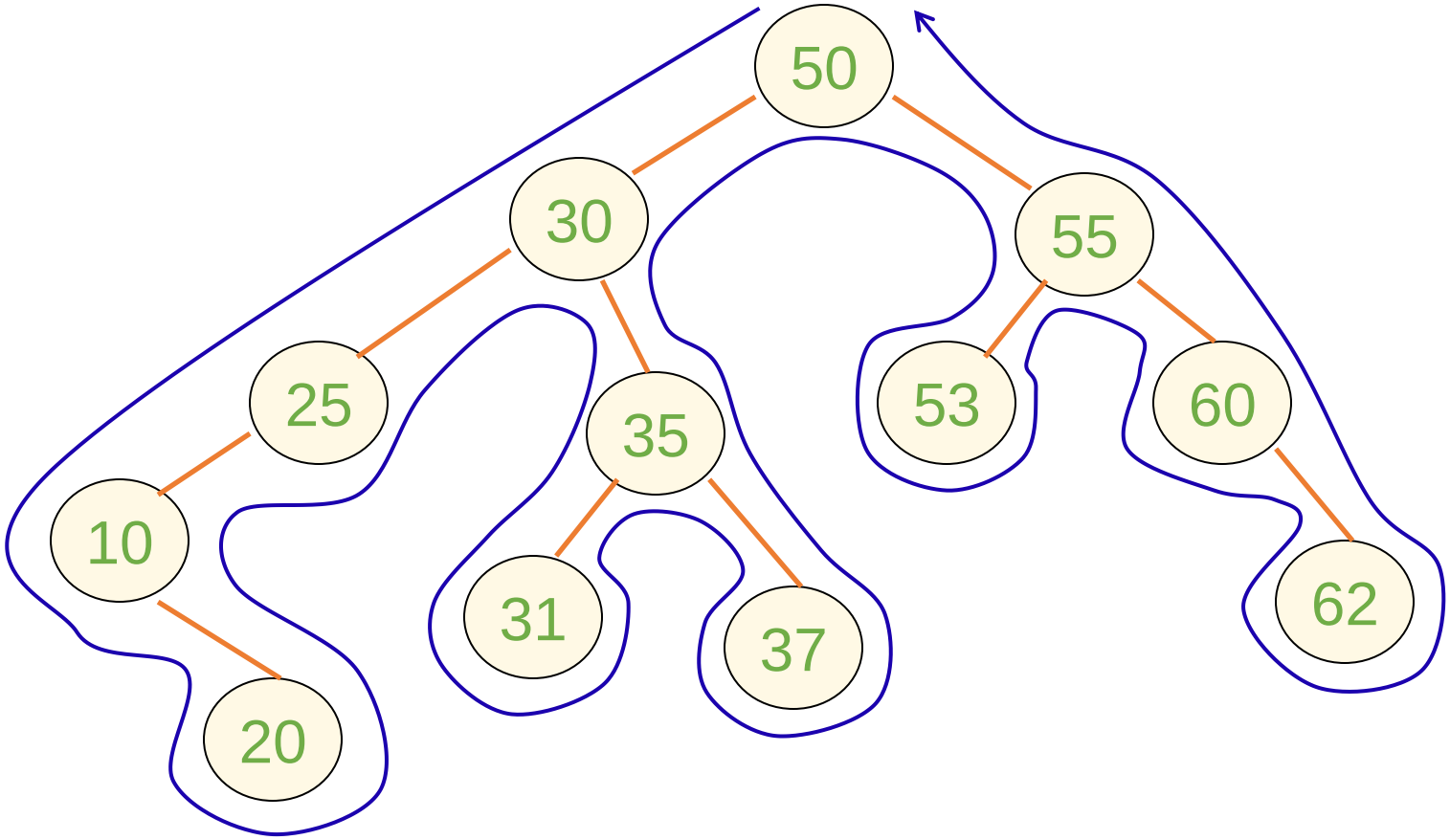
makeTreeNode

```
TreeNode* makeTreeNode(float value) {
    TreeNode* newNodePtr = NULL;

    newNodePtr = (TreeNode*)malloc(sizeof(TreeNode));

    if (newNodePtr == NULL)
    {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    else
    {
        newNodePtr->key = value;
        newNodePtr->leftPtr = NULL;
        newNodePtr->rightPtr = NULL;
    }
    return newNodePtr;
}
```

find_min và find_max

```
TreeNode*  find_min(TreeNode * T) {
 /* luôn đi theo con trái */
if (T == NULL) return(NULL);
else
     if (T->leftPtr == NULL) return(T);
     else return(find_min(T->leftPtr));
}
TreeNode*  find_max(TreeNode* T) {
/* luôn đi theo con phải */
if (T != NULL)
     while (T->rightPtr  !=  NULL)
              T = T->rightPtr;
return(T);
}
```

Duyệt BST theo thứ tự giữa



Đưa ra dãy khoá được sắp xếp:

10, 20, 25, 30, 31, 35, 37, 50, 53, 55, 60, 62

Kế cận trước và Kế cận sau

- Kế cận sau (Successor) của nút *x* là nút *y* sao cho *key*[*y*] là khoá nhỏ nhất còn lớn hơn *key*[*x*].
- Kế cận sau của nút với khoá lớn nhất là NULL.

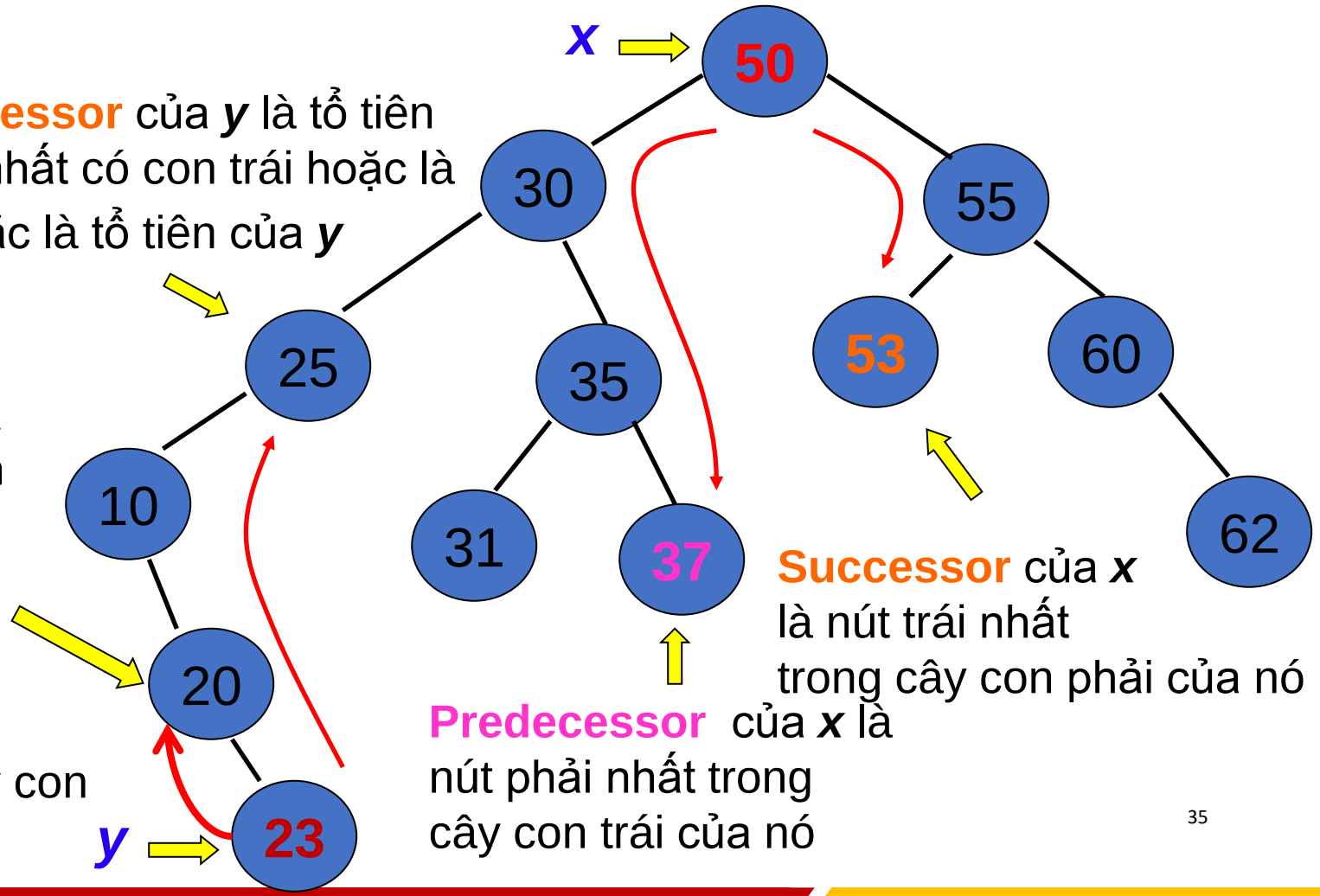
- Kế cận trước (Predcessor) của nút *x* là nút *y* sao cho *key*[*y*] là khoá lớn nhất còn nhỏ hơn *key*[*x*].
- Kế cận trước của nút với khoá nhỏ nhất là NULL.

- Việc tìm kiếm kế cận sau/trước được thực hiện mà không cần thực hiện so sánh khoá.

Kế cận sau và Kế cận trước

Successor và Predecessor

10, 20, 23, 25, 30, 31, 35, 37, 50, 53, 55, 60, 62



Successor của ***y*** là tổ tiên gần nhất có con trái hoặc là ***y*** hoặc là tổ tiên của ***y***

Predessor của ***y*** là tổ tiên gần nhất có con phải hoặc là ***y*** hoặc là tổ tiên của ***y***

y không có cây con cả trái lẫn phải

Successor của ***x*** là nút trái nhất trong cây con phải của nó

Predecessor của ***x*** là nút phải nhất trong cây con trái của nó

Tìm kế cận sau

- Có 2 tình huống

1) Nếu *x* có con phải thì kế cận sau của *x* sẽ là nút *y* với khoá key[*y*] nhỏ nhất trong cây con phải của *x*

   (nói cách khác *y* là nút trái nhất trong cây con phải của *x*).

   - Để tìm *y* có thể dùng find-min(x->rightPtr): ***y*= find-min(x->rightPtr)**
   - hoặc bắt đầu từ gốc của cây con phải luôn đi theo con trái đến khi gặp nút không có con trái chính là nút *y* cần tìm.

Tìm kế cận sau

- Có 2 tình huống

2) Nếu *x* không có con phải thì kế cận sau của *x* là tổ tiên gần nhất có con trái hoặc là ***x*** hoặc là tổ tiên của ***x.***

Để tìm kế cận sau:

- Bắt đầu từ *x* cần di chuyển lên trên (theo con trỏ parent) cho đến khi gặp nút *y* có con trái đầu tiên thì dừng: *y* là kế cận sau của *x*.

- Nếu không thể di chuyển tiếp được lên trên (tức là đã đến gốc) thì *x* là nút lớn nhất (và vì thế *x* không có kế cận sau).

Tìm kế cận trước

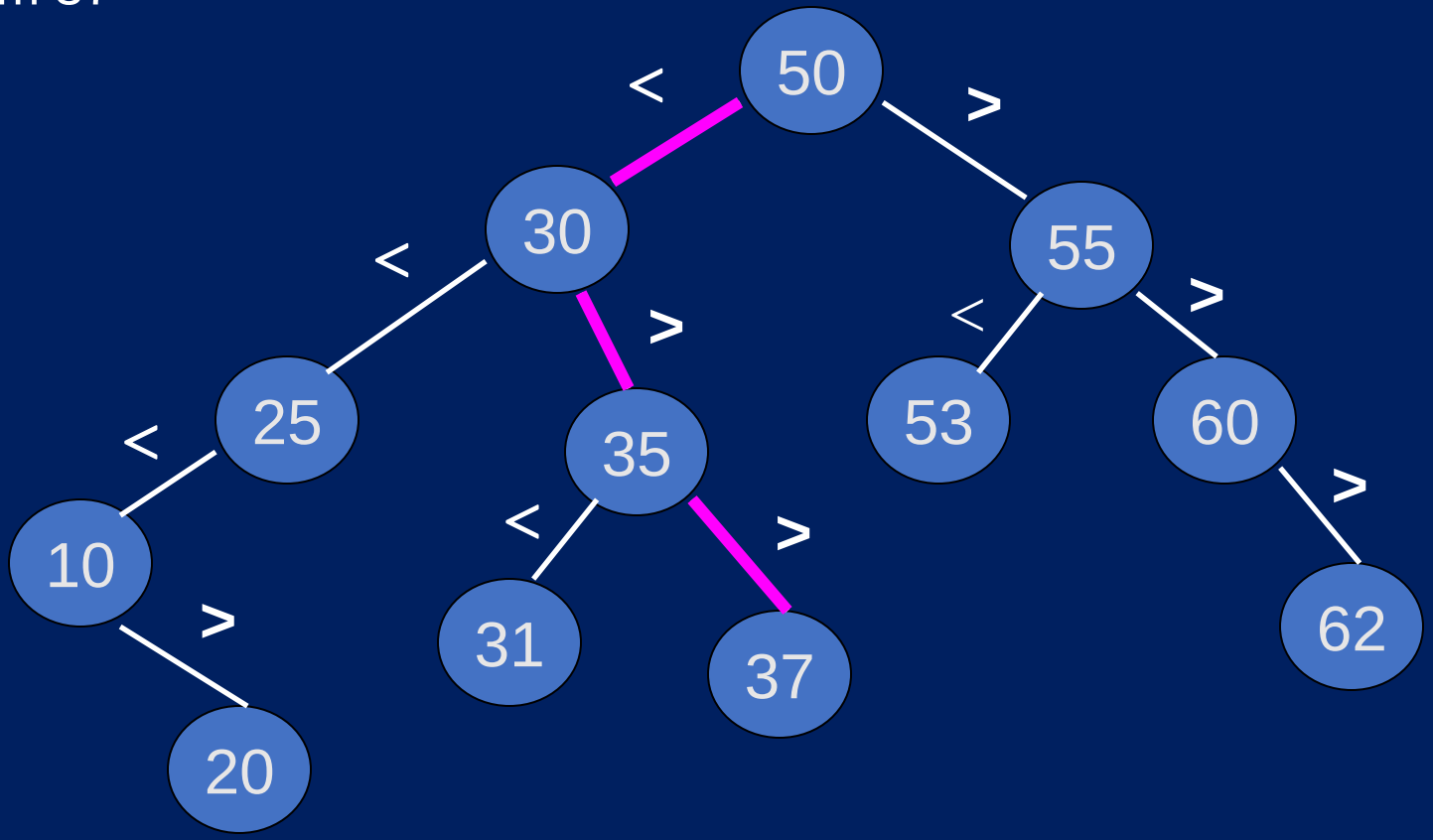
- Tương tự như tìm kế cận sau, có 2 tình huống
- Nếu *x* có con trái thì kế cận trước của *x* sẽ là nút *y* với khoá key[*y*] lớn nhất trong cây con trái của *x* (nói cách khác *y* là nút phải nhất nhất trong cây con trái của *x*):

  y = find_max(x->leftPtr)

- Nếu *x* không có con trái thì kế cận trước của *x* là tổ tiên gần nhất có con phải hoặc là ***x*** hoặc là tổ tiên của ***x.***

Ví dụ: Tìm kiếm trên BST

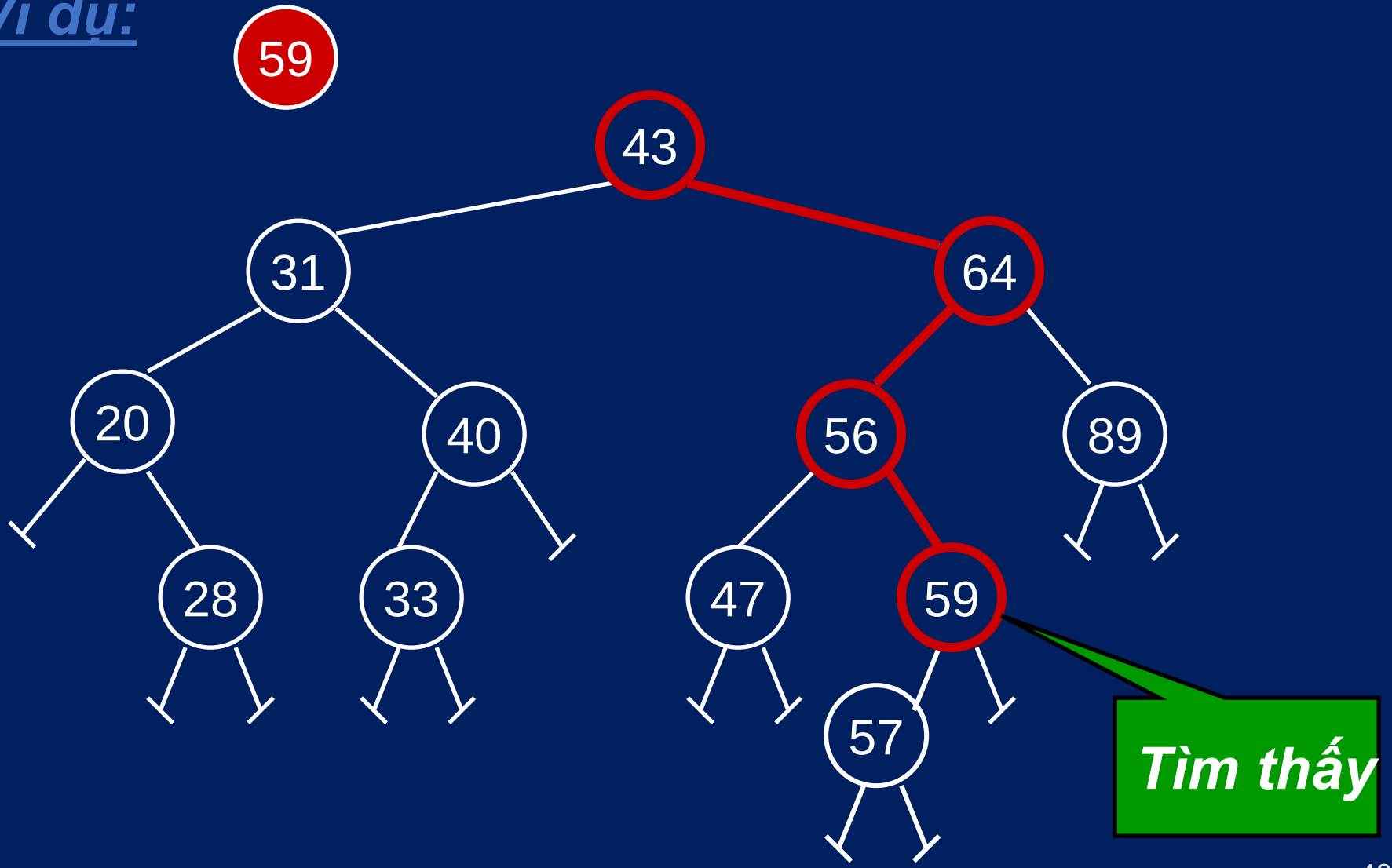
Tìm 37



Thời gian tìm: $O(h)$ trong đó h là chiều cao của cây

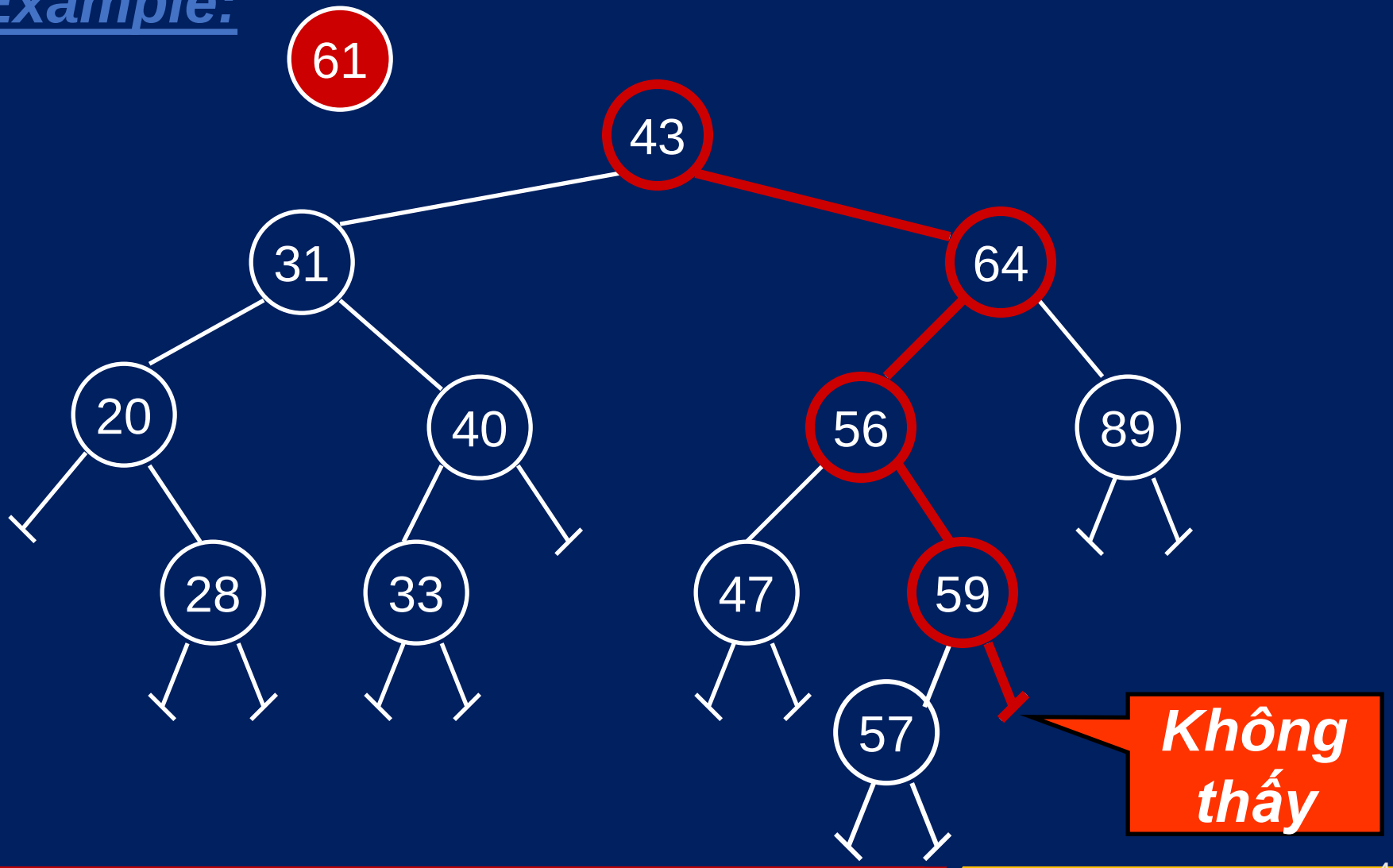
Tìm kiếm (Search)

Ví dụ:



Search

Example:



Cài đặt trên C

```
TreeNode* search(TreeNode* nodePtr, float target)
{
  if (nodePtr != NULL)
  {
    if (target < nodePtr->key)
    {
      nodePtr = search(nodePtr->leftPtr, target);
    }
    else if (target > nodePtr->key)
    {
      nodePtr = search(nodePtr->rightPtr, target);
    }
  }
  return nodePtr;
}
```

Thời gian tính: $O(h)$,
trong đó h là độ cao của BST

Ví dụ: Đoạn chương trình chứa lệnh gọi đến Search

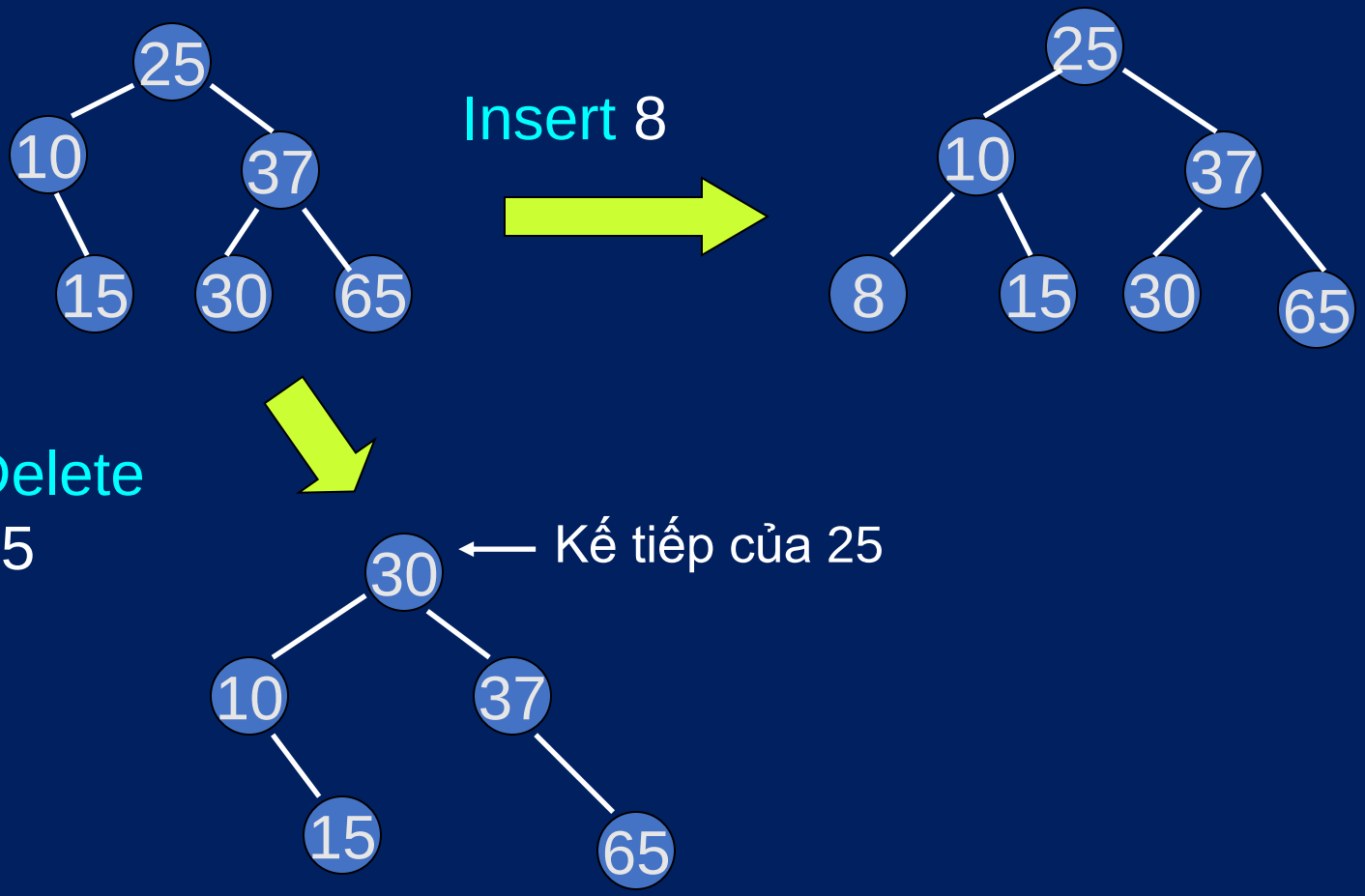
```
/* … một số lệnh …  */

printf("Enter target ");
scanf("%f", &item);

if (search(rootPtr, item) == NULL)
{
  printf("Không tìm thấy\n");
}
else
{
  printf("Tìm thấy\n");
}

/* … các lệnh khác … */
```

Inorder

- Duyệt theo thứ tự giữa của BST luôn cho dãy các khoá được sắp xếp.

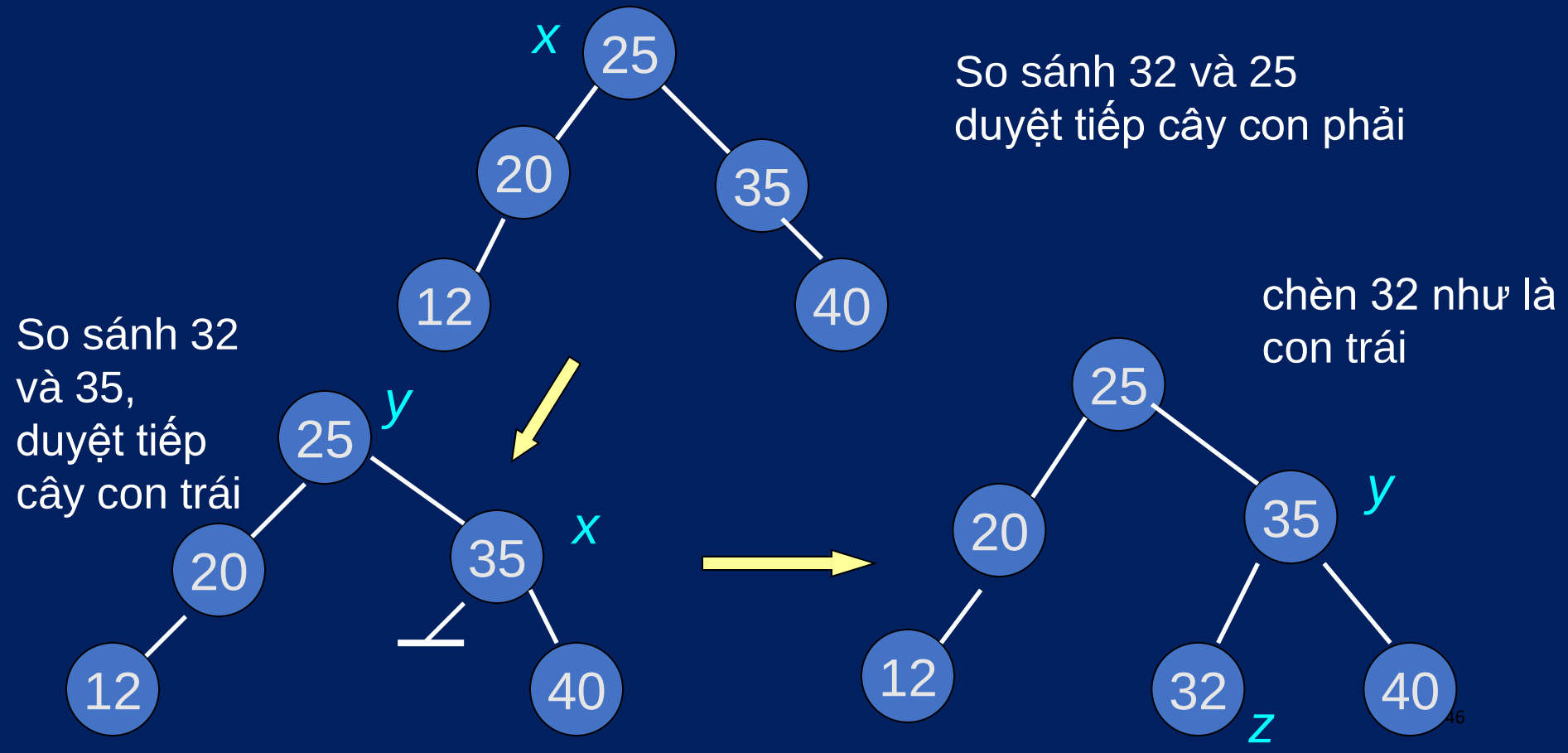
```
void printInorder(TreeNode* nodePtr)
{
     if (nodePtr != NULL)
     {
       printInorder(nodePtr->leftPtr);
       printf(" %f", nodePtr->key);
       printInorder(nodePtr->rightPtr);
     }
}
```

Chèn và xoá trên BST



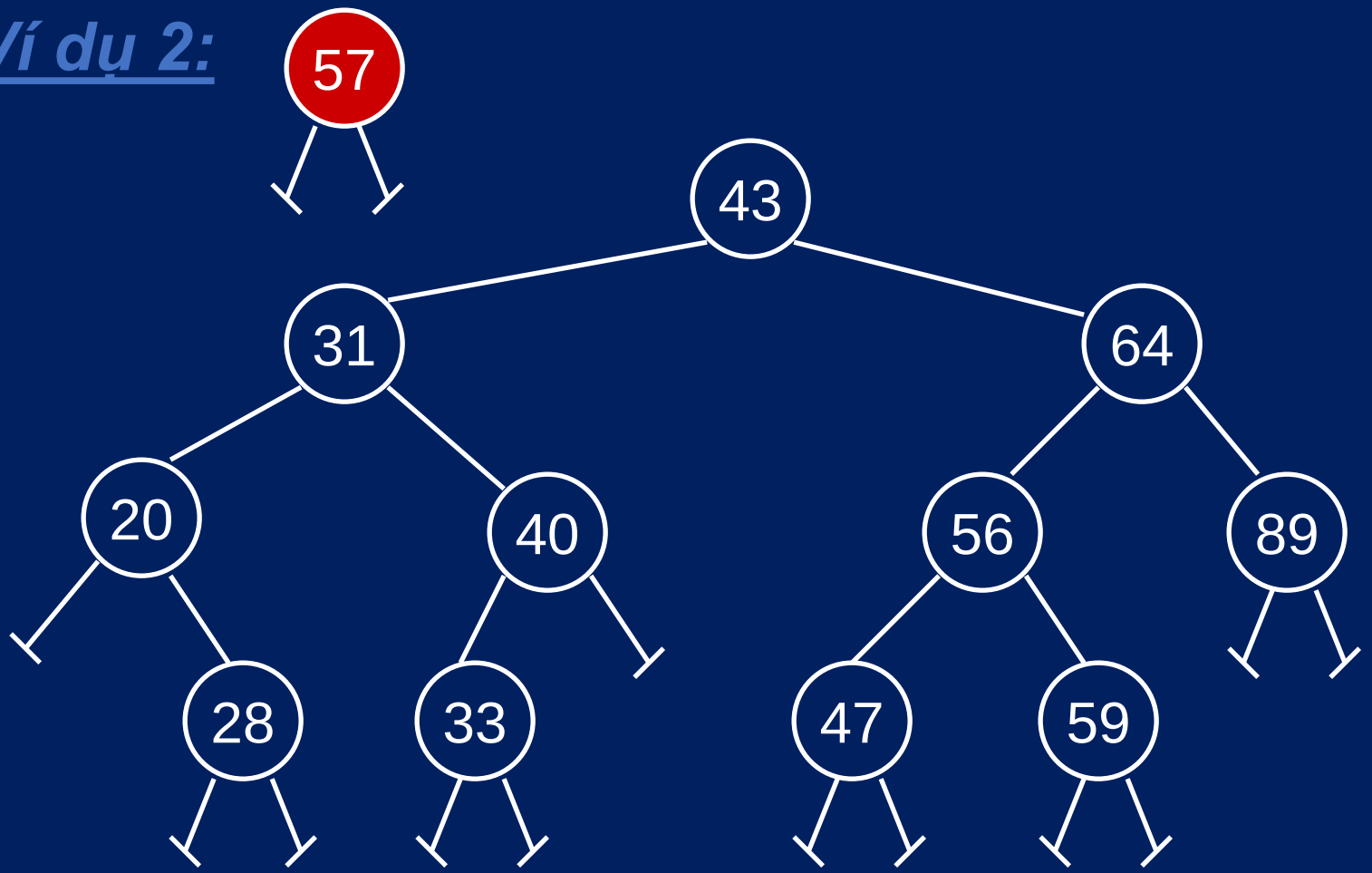
Bổ sung (Insertion)

Ví dụ 1: bổ sung $z = 32$



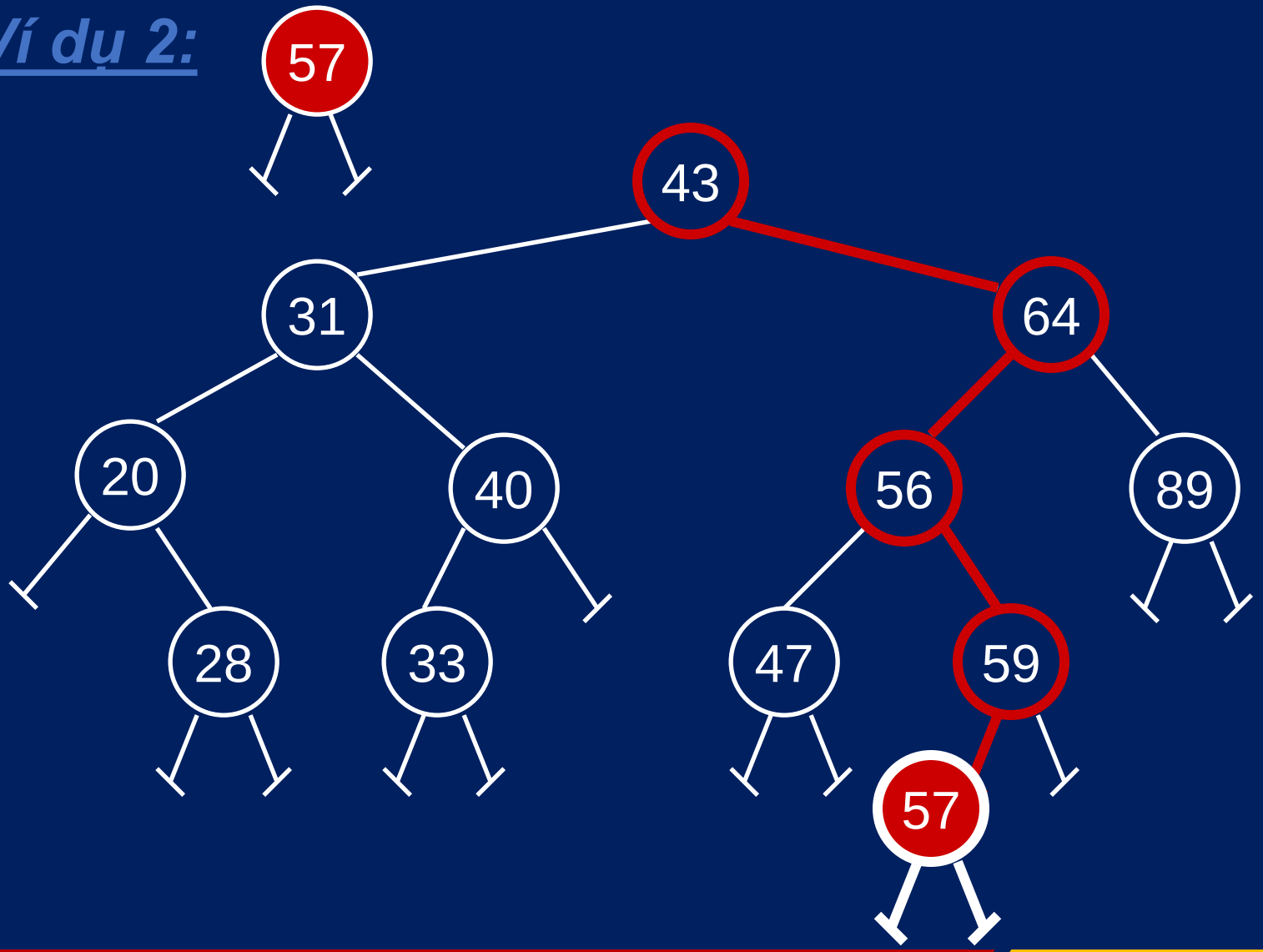
Insert

Ví dụ 2:



Insert

Ví dụ 2:



Thuật toán bổ sung (Insert)

- Tạo nút mới chứa phần tử cần chèn.
- Tìm cha của nút mới.
- Gắn nút mới như là lá.

Cài đặt trên C

```
TreeNode* insert(TreeNode* nodePtr, float item)
{
   if (nodePtr == NULL)
   {
     nodePtr = makeTreeNode(item);
   }
   else if (item < nodePtr->key)
   {
     nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
   }
   else if (item > nodePtr->key)
   {
     nodePtr->rightPtr = insert(nodePtr->rightPtr, item);
   }
   return nodePtr;
}
```

Thời gian tính: $O(h)$,
trong đó h là độ cao của BST

Hàm gọi đến Insert

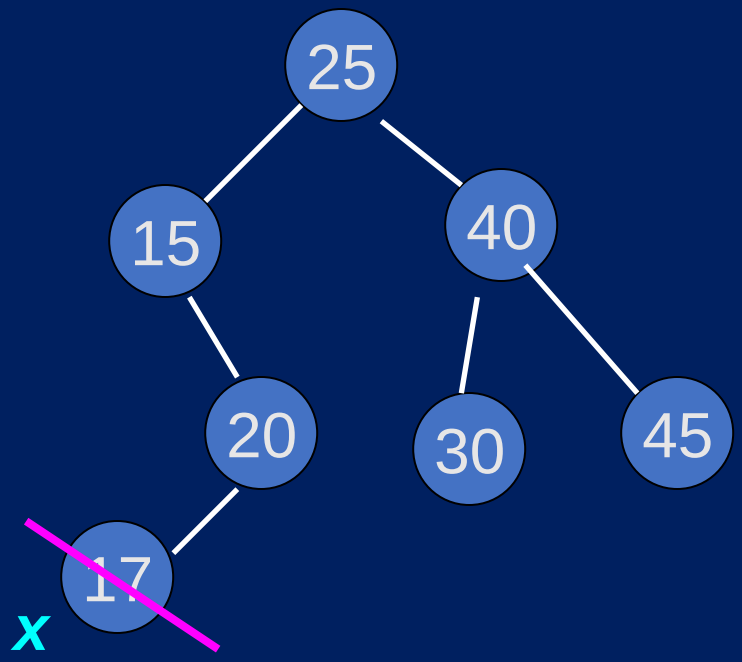
```
/* … còn các lệnh khác ở đây …  */

printf("Enter number of items ");
scanf("%d", &n);

for (i = 0; i < n; i++) {
   scanf("%f", &item);
   rootPtr = insert(rootPtr, item);
}

/* … và còn những lệnh tiếp tục … */
```

Thao tác loại bỏ - delete

- Khi loại bỏ một nút, cần phải đảm bảo cây thu được vẫn là cây nhị phân
- Vì thế, khi xoá cần phải xét cẩn thận các con của nó.
- Có bốn tình huống cần xét:
  - Tình huống 1: Nút cần xoá là lá
  - Tình huống 2: Nút cần xoá chỉ có con trái
  - Tình huống 3: Nút cần xoá chỉ có con phải
  - Tình huống 4: Nút cần xoá có hai con

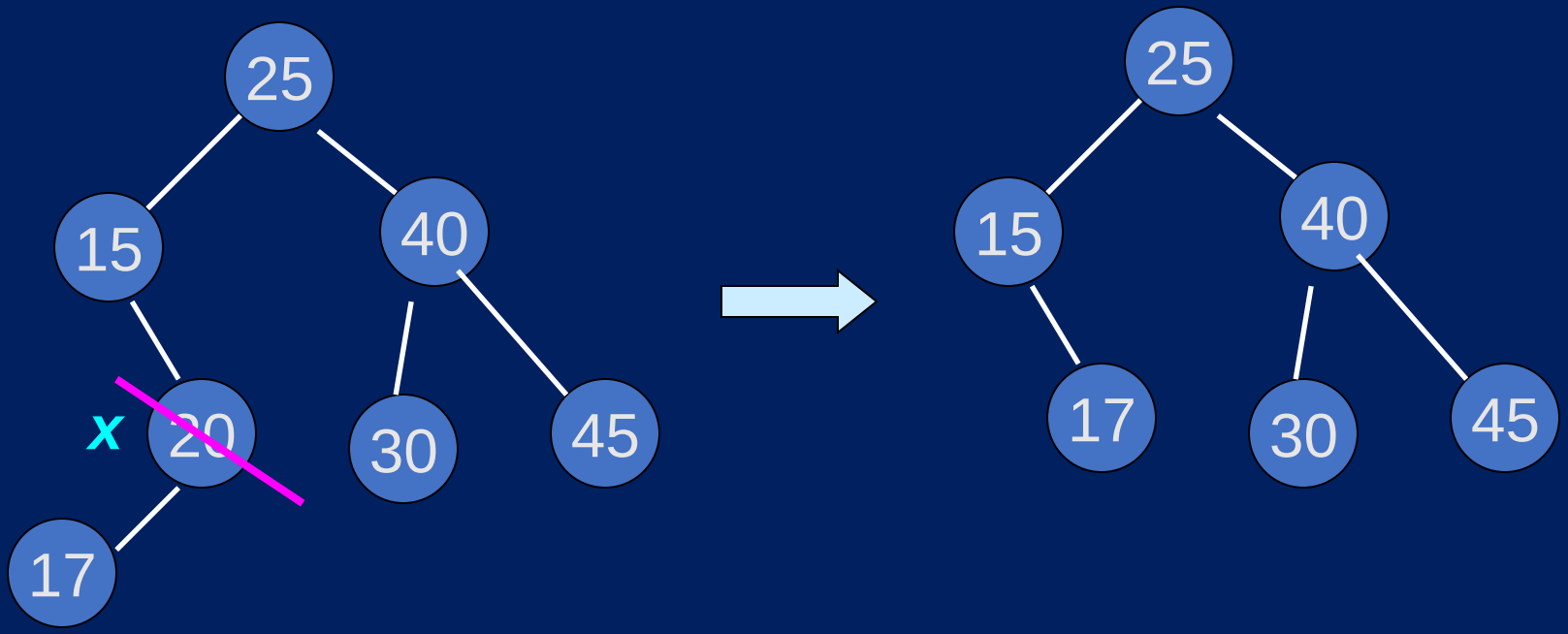
Deletion: Tình huống 1

Tình huống 1: Nút cần xoá ***x*** là lá (leaf node)
Thao tác: chữa lại nút cha của ***x*** có con rỗng.



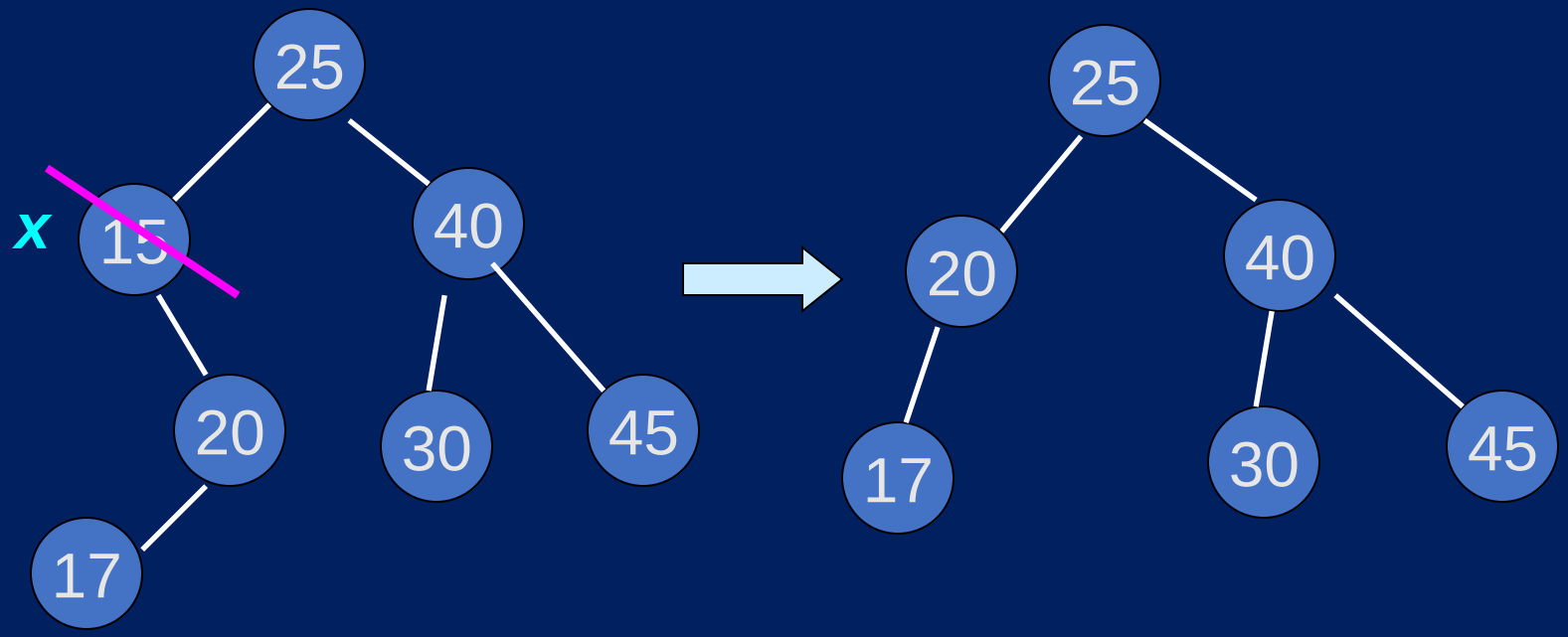
Deletion: Tình huống 2

Tình huống 2: nút cần xoá *x* có con trái mà không có con p
Thao tác: gắn cây con trái của *x* vào cha



Deletion: Tình huống 3

Tình huống 3: nút cần xoá *x* có con phải mà không có con t
Thao tác: gắn cây con phải của *x* vào cha

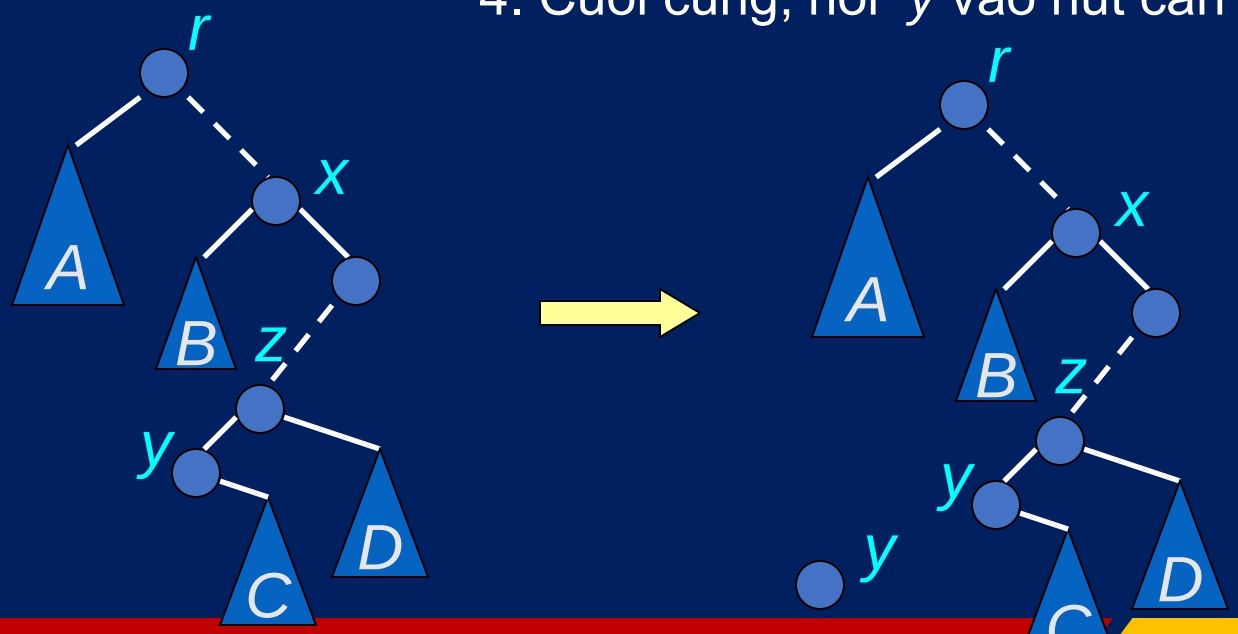


Deletion: Tình huống 4

Tình huống 4: nút *x* có hai con

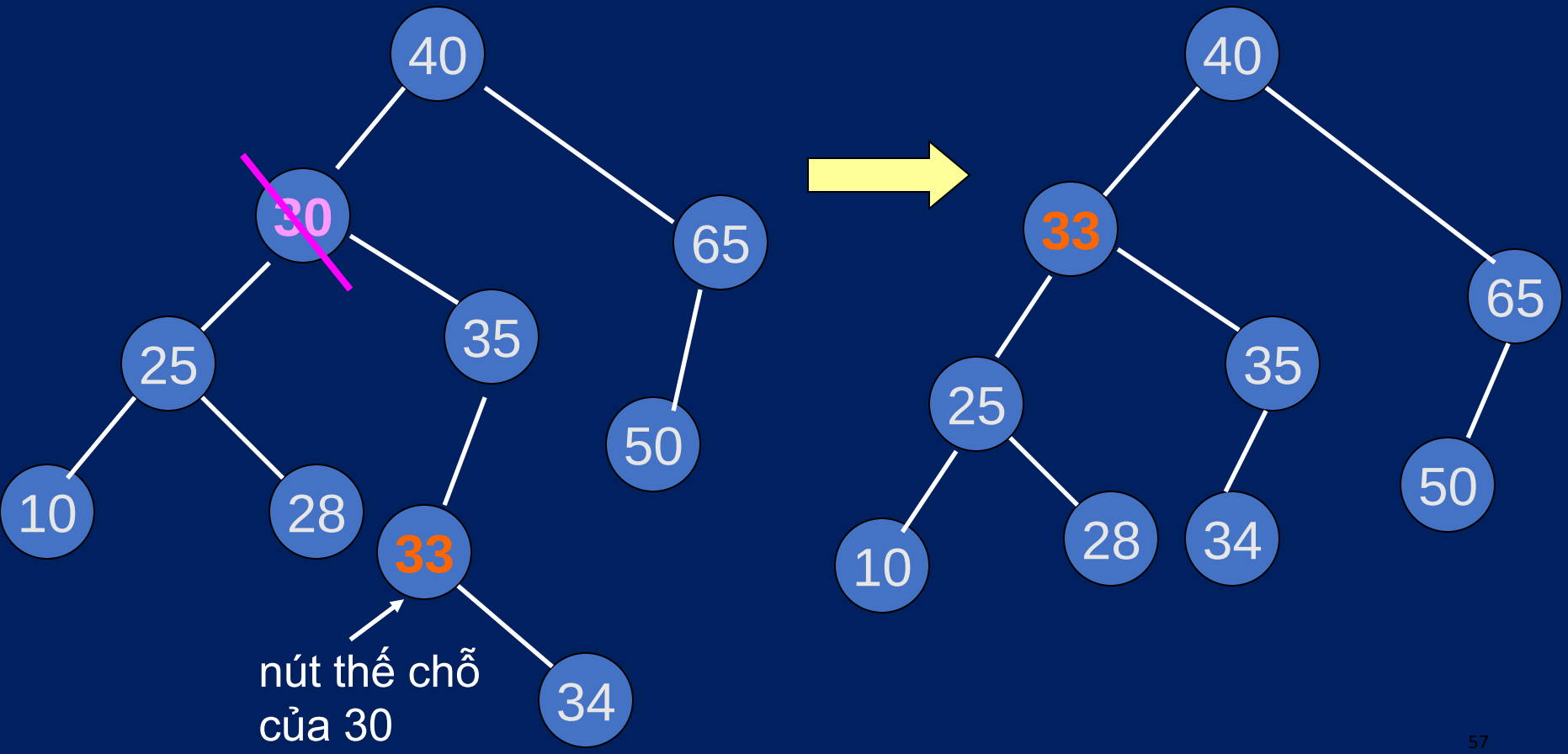
Thao tác: 1. Chọn nút *y* để thế vào chỗ của *x*, nút y sẽ là successor của *x*. *y* là giá trị nhỏ nhất còn lớn hơn *x*.

2. Gỡ nút *y* khỏi cây.
3. Nối con phải của *y* vào cha của *y*.
4. Cuối cùng, nối *y* vào nút cần xoá.



Ví dụ: Tình huống 4

10, 25, 28, **30**, **33**, 34, 35, 40, 50, 65 → 10, 25, 28, **33**, 34, 35, 40, 50, 65



Xoá phần tử có *key = x*

```
TreeNode* delete(TreeNode * T, float x) {
  TreeNode  tmp;
  if (T == NULL) printf("Not found\n");
  else if (x < T->key) /* đi bên trái */
      T->leftPtr = delete(T->leftPtr, x);
  else if (x > T->key) /* đi bên phải */
      T->rightPtr = delete(T->rightPtr,x);
  else /* tìm được phần tử cần xoá */
  if (T->leftPtr && T->rightPtr) {
    /* Tình huống 4: phần tử thế chỗ là
         phần tử min ở cây con phải */
      tmp = find_min(T->right);
      T->key = tmp->key;
      T->rightPtr = delete(T->rightPtr, T-
>key);
  }
  else
{ /* có 1 con hoặc không có con */
      tmp = T;
      if (T->left Ptr== NULL)
        /* chỉ có con phải
             hoặc không có con */
          T = T->rightPtr;
      else
        if (T->rightPtr == NULL)
          /* chỉ có con trái */
          T = T->leftPtr;
    free(tmp);
   }
   return(T);
}
```

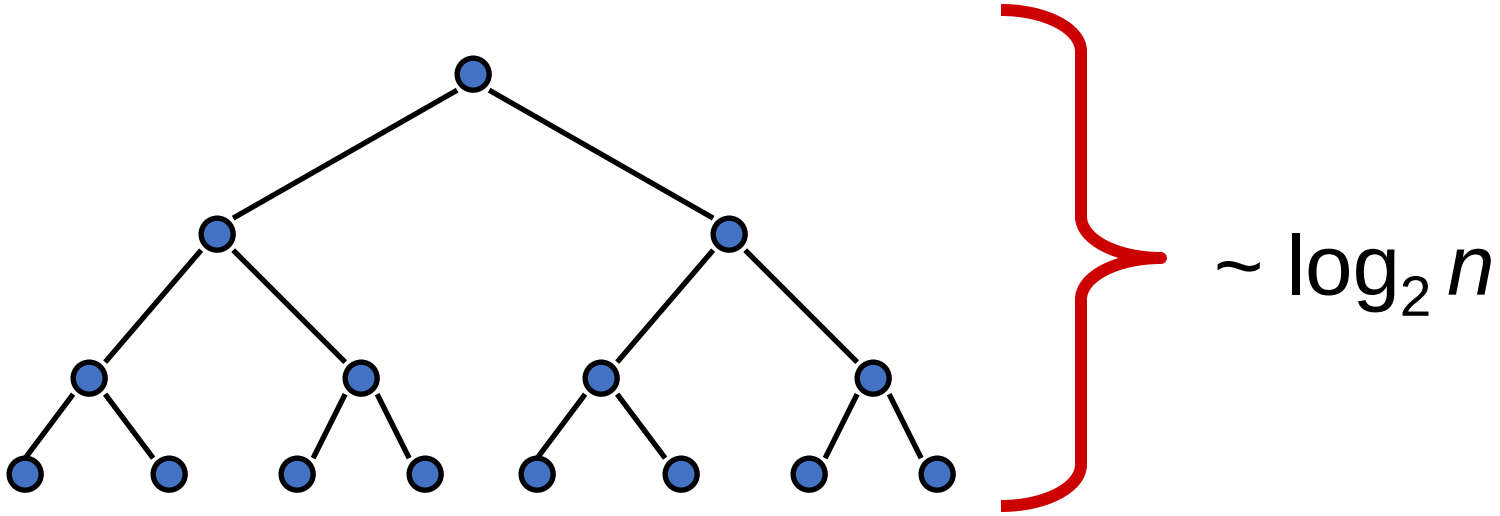
Sắp xếp nhờ sử dụng BST

Để sắp xếp dãy phần tử ta có thể thực hiện như sau:

- Bổ sung (insert) các phần tử vào cây nhị phân tìm kiếm.
- Duyệt BST theo thứ tự giữa để đưa ra dãy được sắp xếp.

Sorting: Phân tích hiệu quả

- Tình huống trung bình: O(*n* log *n*)



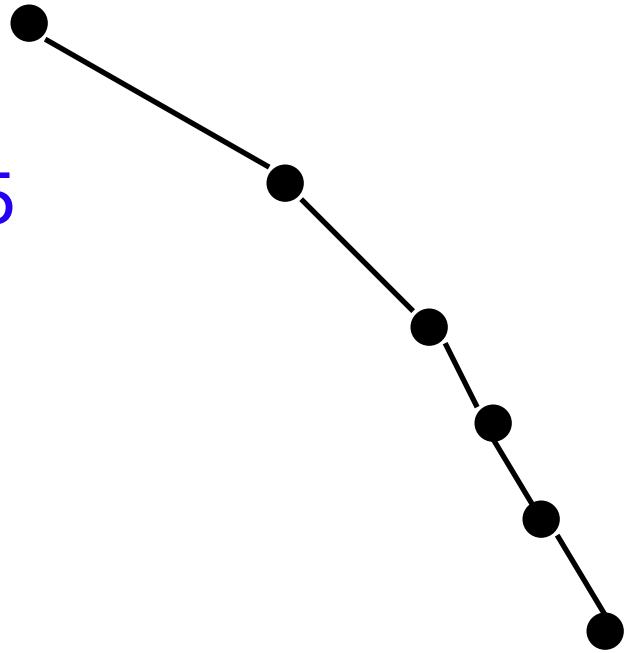
- Chèn phần tử thứ (*i*+1) tốn quãng $\log_2(i)$ phép so sánh

Sorting: Phân tích hiệu quả

- Tình huống tồi nhất: $O(n^2)$

Ví dụ:

Bổ sung dãy: 1, 3, 7, 9, 11, 15



- Bổ sung phần tử thứ (i+1) tốn quãng i phép so sánh

Độ phức tạp trung bình của các thao tác với BST

- Người ta chỉ ra được rằng độ cao trung bình của BST là

  $h = O(\log n)$.

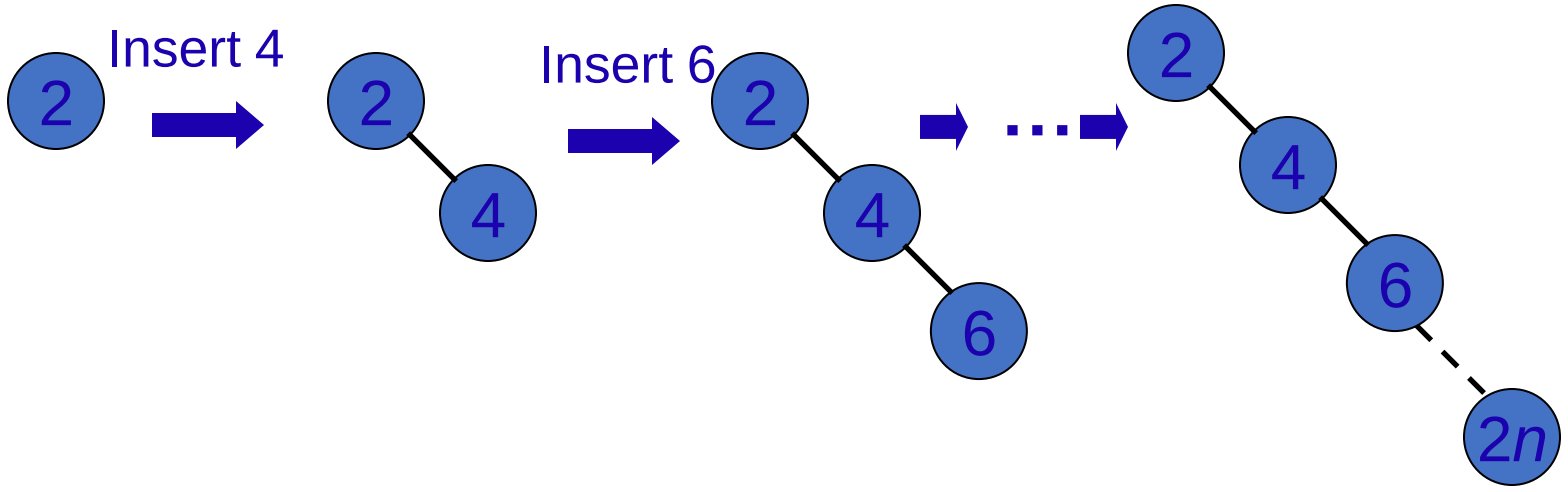
- Từ đó suy ra độ phức tạp trung bình của các thao tác với BST là:

Insertion	**……**	**$O(\log n)$**
Deletion	**……**	**$O(\log n)$**
Find Min	**……**	**$O(\log n)$**
Find Max	**……**	**$O(\log n)$**
BST Sort	**……**	**$O(n \log n)$**

Thời gian tính tồi nhất

Tất cả các phép toán cơ bản (Search, Successor, Predecessor, Minimum, Maximum, Insert, Delete) trên BST với chiều cao h đều có thời gian tính là $O(h)$.

Trong tình huống tồi nhất $h = n$:



Vấn đề đặt ra là: Có cách nào đảm bảo $h = O(\log n)$?

NỘI DUNG

6.1. Tìm kiếm tuần tự và tìm kiếm nhị phân

6.2. Cây nhị phân tìm kiếm

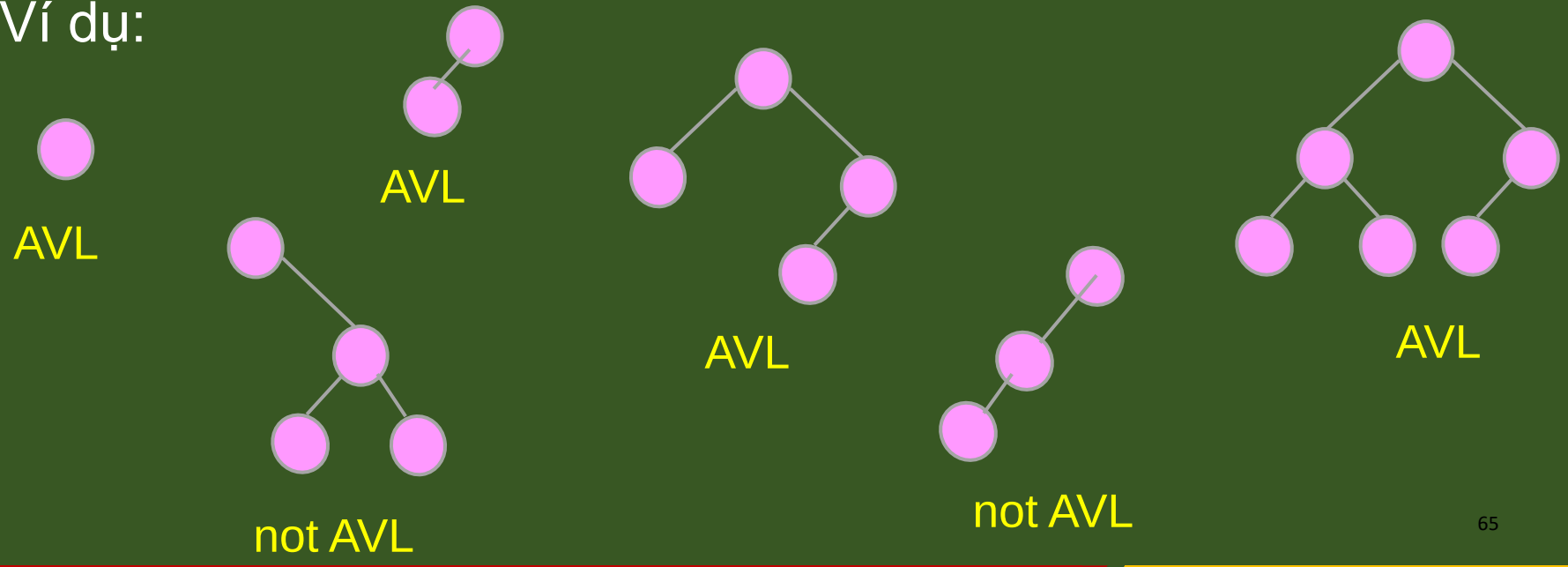
6.3. Cây AVL

6.4. Bảng băm

6.3. Cây AVL

Định nghĩa. ***Cây AVL*** là cây nhị phân tìm kiếm thoả mãn ***tính chất AVL*** sau đây: chiều cao của cây con trái và cây con phải của gốc chỉ sai khác nhau không quá 1 và cả cây con trái và cây con phải đều là cây AVL.

Ví dụ:

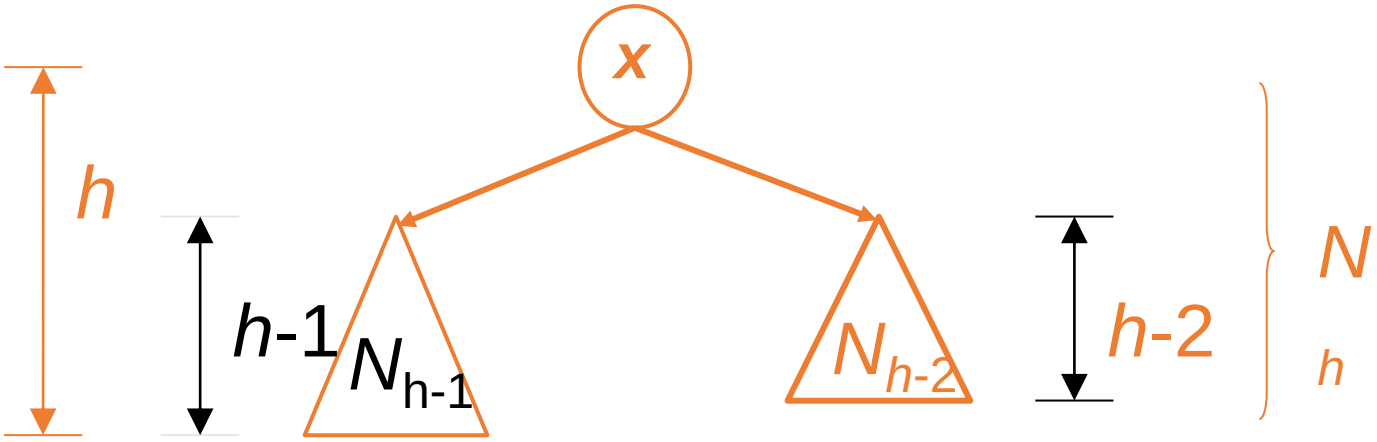


Tính chất AVL

Tính chất AVL (tính chất cân bằng):

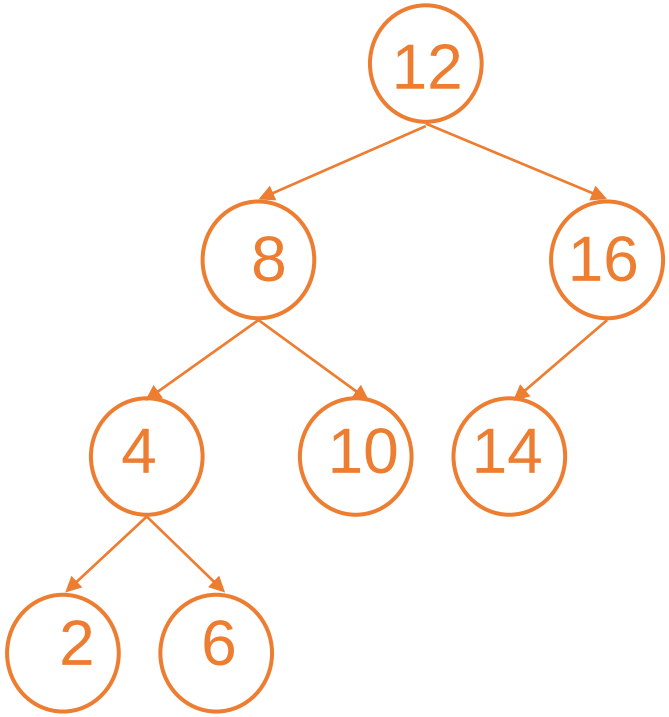
Chênh lệch giữa độ cao của cây con trái và độ cao của cây con phải của một nút bất kỳ trong cây là không quá 1.

Chú ý: Cây nhị phân đầy đủ và hoàn chỉnh có tính chất này, nhưng cây có tính chất AVL không nhất thiết phải là đầy đủ hay hoàn chỉnh

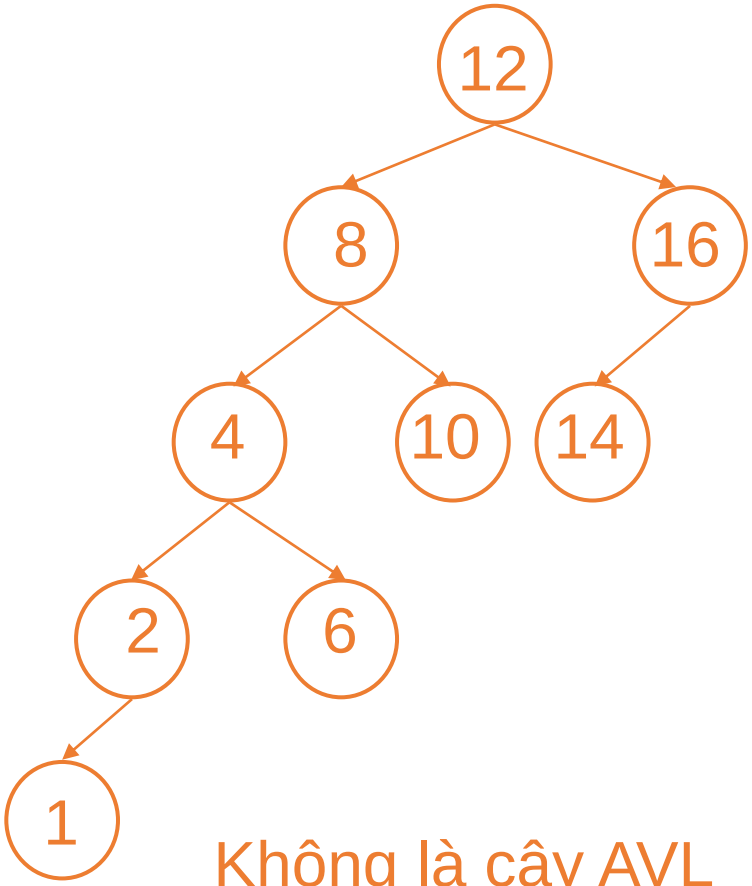


Số nút của cây: $N_h = N_{h-1} + N_{h-2} + 1$

Ví dụ: Cây AVL



Cây AVL



Không là cây AVL
(nút 8 và 12 không thoả mãn t/c AVL)

AVL Trees

- **Cây AVL** (Adelson-Velskii và Landis, 1962):
  - là cây BST được giữ sao cho luôn ở trạng thái cân bằng (cân đối).
  - *Ý chủ đạo:* Nếu việc bổ sung hay loại bỏ dẫn đến mất tính cân bằng của cây thì cần tiến hành khôi phục ngay lập tức
  - Các thao tác: tìm kiếm, bổ sung, loại bỏ đều có thể thực hiện với cây AVL có n nút trong thời gian $O(\log n)$ (cả trong tình huống trung bình lẫn tồi nhất!)

Kiểm tra tính AVL

- **Định nghĩa.** Chiều cao (height) của nút *x,* ký hiệu là *h*(*x*):
  - *h*(*x*) = 1, nếu *x* là lá. *h*(*x*) = 0, nếu *x* = NULL.
  - $h(x) = \max\{h_{left}(x), h_{right}(x)\}+1$, nếu *x* là nút trong, trong đó $h_{left}(x)$ ($h_{right}(x)$) là chiều cao của cây con trái (phải) của *x*.

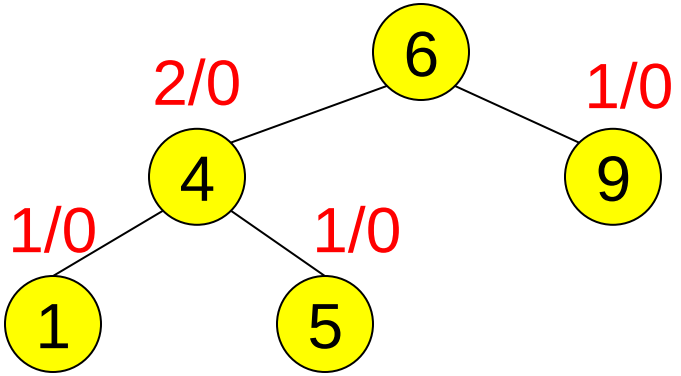
- **Định nghĩa.** Hệ số cân bằng (*balance factor*) của nút *x* , ký hiệu là bal(*x*): hiệu giữa chiều cao của cây con phải và cây con trái của *x*.

- Như vậy, cây nhị phân tìm kiếm là cây ***AVL*** nếu như hệ số cân bằng của mỗi nút của nó là 0, 1

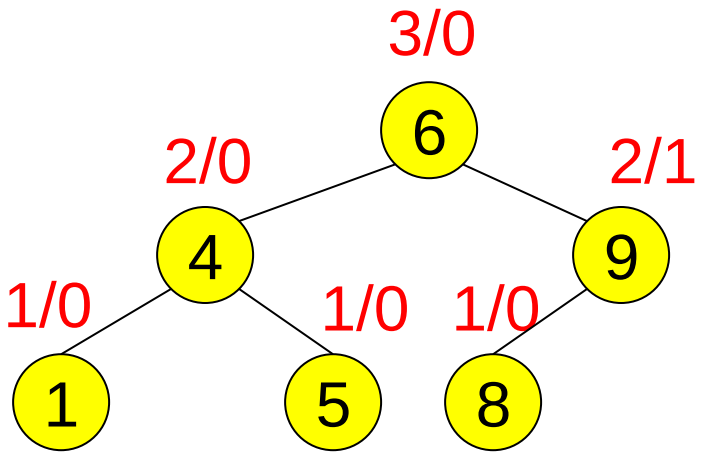
Ví dụ: Chiều cao của nút

Tree A
(AVL)

h=3 bal=2-1=1



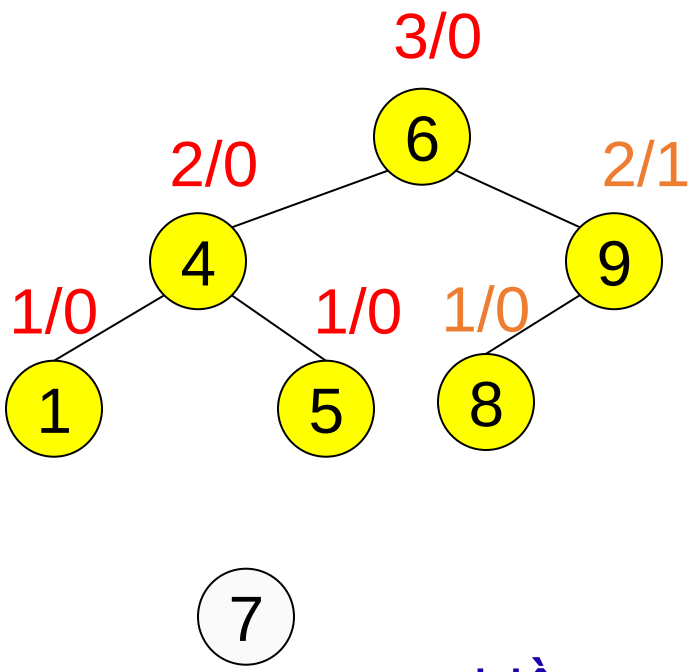
Tree B
(AVL)



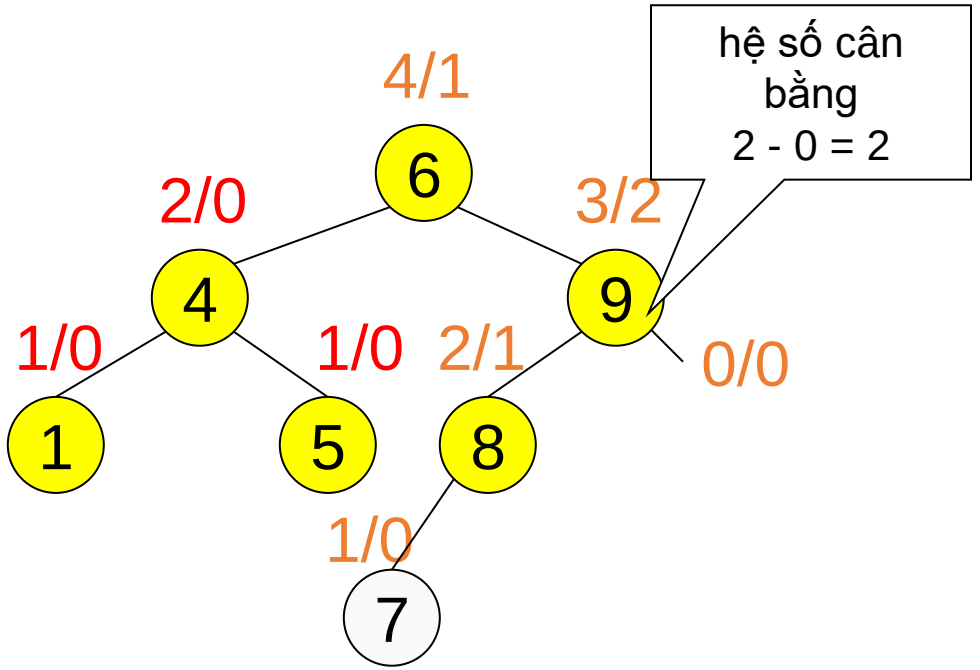
chiều cao của nút: $h(x)$
hệ số cân bằng = $h_{left}(x)$-$h_{right}(x)$
chiều cao của nút rỗng = 0

Chiều cao của nút sau khi chèn thêm nút 7

Tree A (AVL)



Tree B (not AVL)

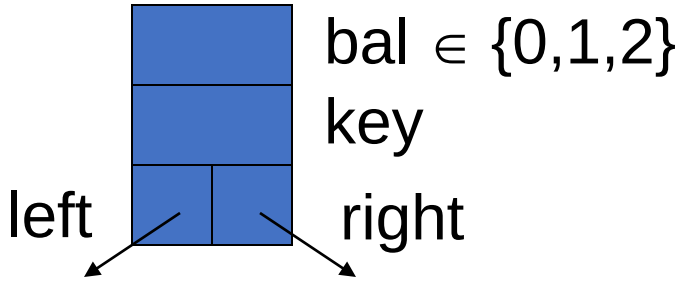


chiều cao của nút: $h(x)$

Hệ số cân bằng = $h_{left}(x)$-$h_{right}(x)$

chiều cao của nút rỗng = 0

Biểu diễn cây AVL




```
struct TreeNode
{   int                 bal;
    float               key;
    struct TreeNode*    left;
    struct TreeNode*    right;
};

typedef struct TreeNode AvlTree;
```

Khôi phục tính cân bằng của cây

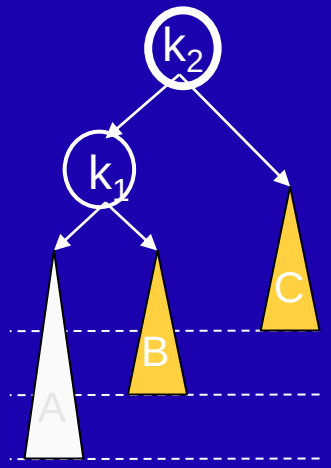
- Trước khi thực hiện thao tác: cây là cân bằng.
- Sau khi thực hiện thao tác bổ sung hoặc loại bỏ (insertion or deletion operation): cây có thể trở thành mất cân bằng.

  ➔ cần xác định cây con mất cân bằng.
- Chiều cao của một cây con bất kỳ chỉ có thể tăng hoặc giảm nhiều nhất là 1.

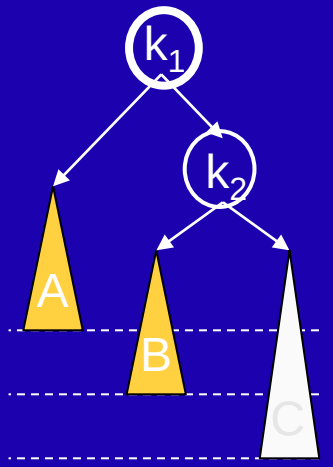
  ➔ nếu xảy ra mất cân bằng thì chênh lệch chiều cao giữa hai cây con chỉ có thể là 2. ➔ Có 4 tình huống.

Các tình huống mất cân bằng (1)

Tình huống 1: Cây con trái cao hơn cây con phải nguyên do bởi cây con trái của con trái.



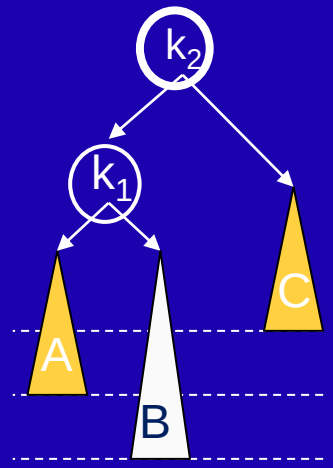
Tình huống 2: Cây con phải cao hơn cây con trái nguyên do bởi cây con phải của con phải.



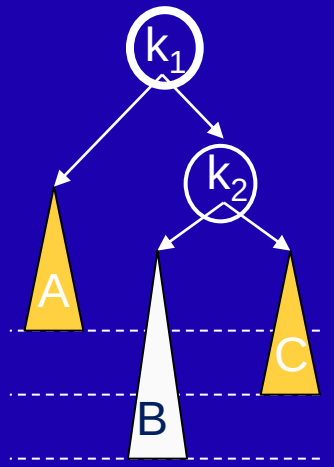
Khôi phục cân bằng nhờ sử dụng phép quay đơn: Quay phải hoặc quay trái (Single rotation: right rotation or left rotation)

Các tình huống mất cân bằng (2)

Tình huống 3: Cây con trái cao hơn cây con phải nguyên do bởi cây con phải của con trái.

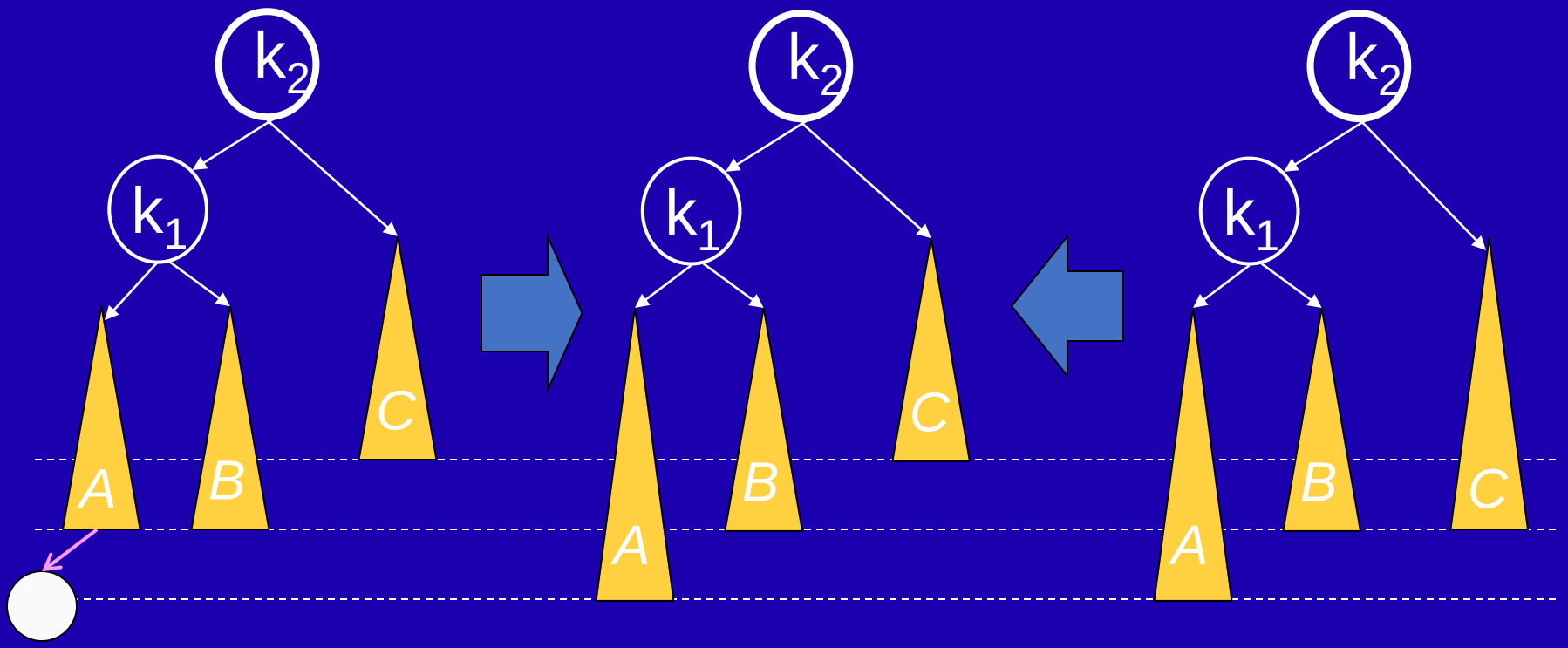


Tình huống 4: Cây con phải cao hơn cây con trái nguyên do bởi cây con trái của con phải



Khôi phục cân bằng nhờ sử dụng phép quay kép: Quay phải rồi quay trái hoặc quay trái rồi quay phải (Double rotation: right-left rotation or left-right rotation)

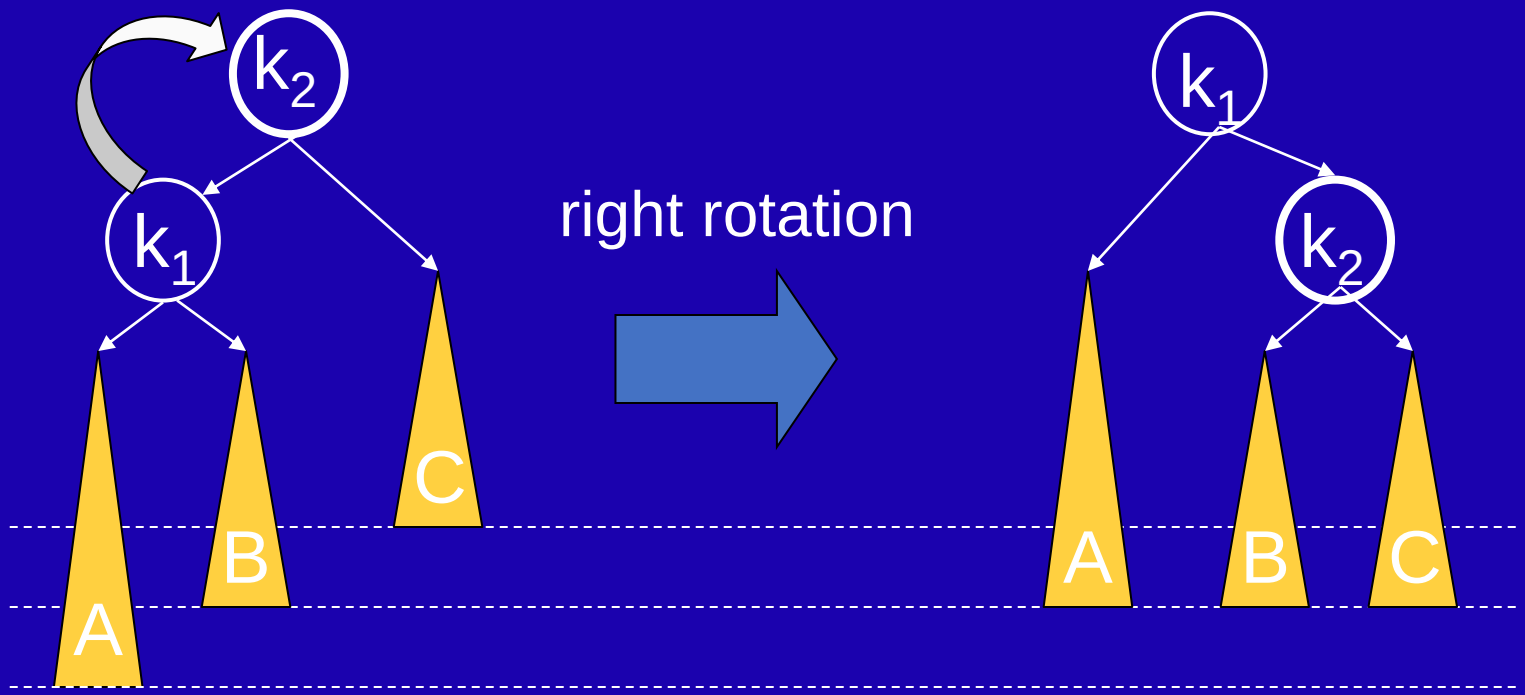
Thao tác bổ sung hoặc loại bỏ có thể dẫn đến tình huống 1



- **Insertion** – Chiều cao của cây con *A* có thể tăng thêm 1 sau khi bổ sung.

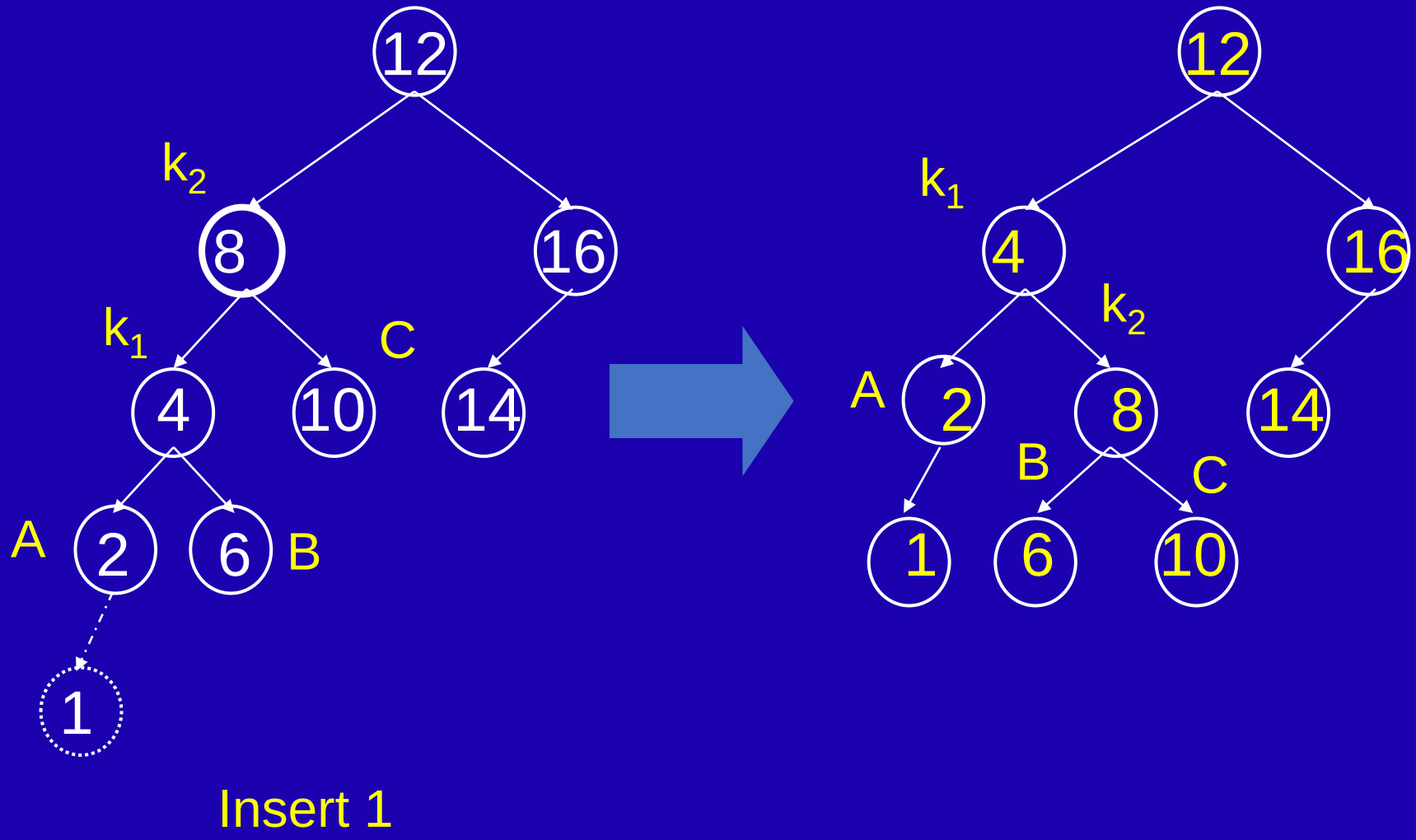
- **Deletion** – Chiều cao của cây con *C* có thể giảm đi 1 sau khi loại bỏ

Xử lý tình huống 1:
Thực hiện phép quay phải (right rotation)



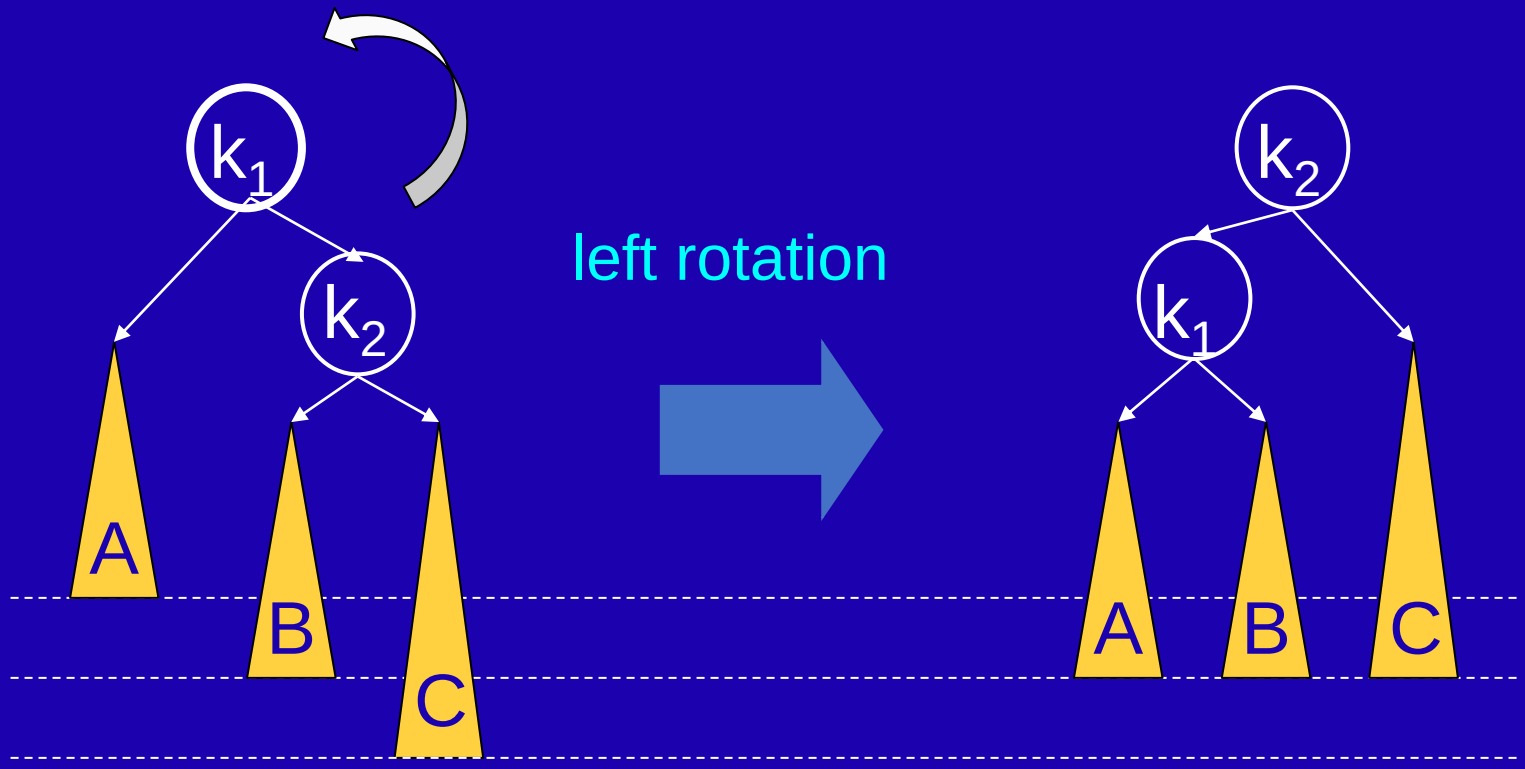
- Thời gian *O*(1) và khôi phục được cân bằng
- **Insertion** – Chiều cao của cây con *A* giảm đi 1. Sau khi quay cây có độ cao như trước khi thực hiện bổ sung.
- **Deletion** – Chiều cao của cây con *C* tăng thêm 1. Sau khi quay chiều cao của cây giảm đi 1 so với trước khi loại bỏ.

Ví dụ: Tình huống 1 xảy ra khi thực hiện bổ sung



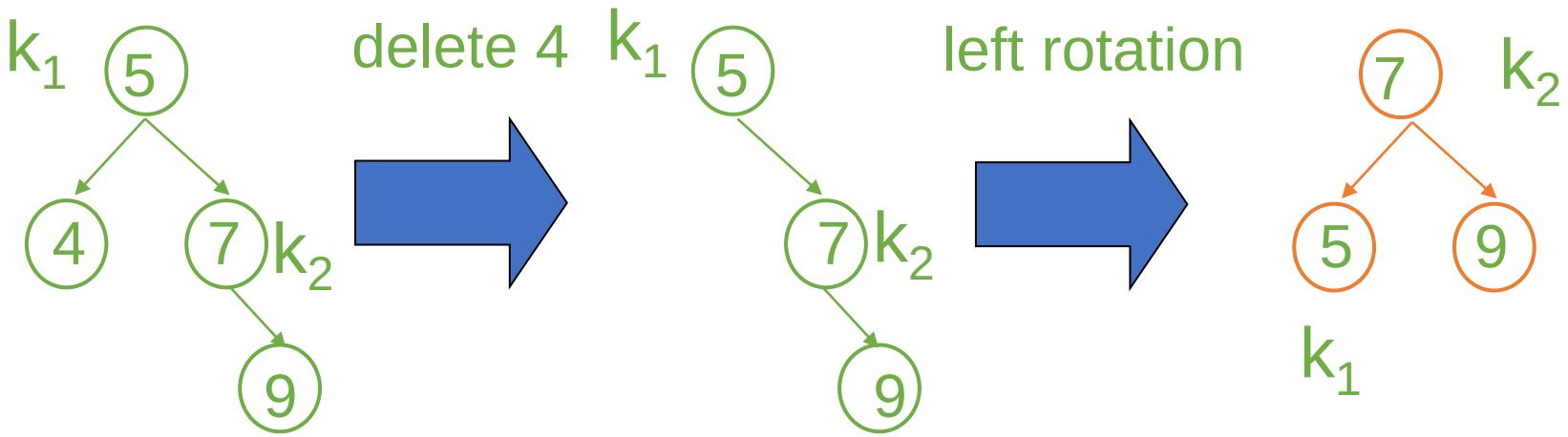
Xử lý tình huống 2:

Thực hiện phép quay trái (left rotation)

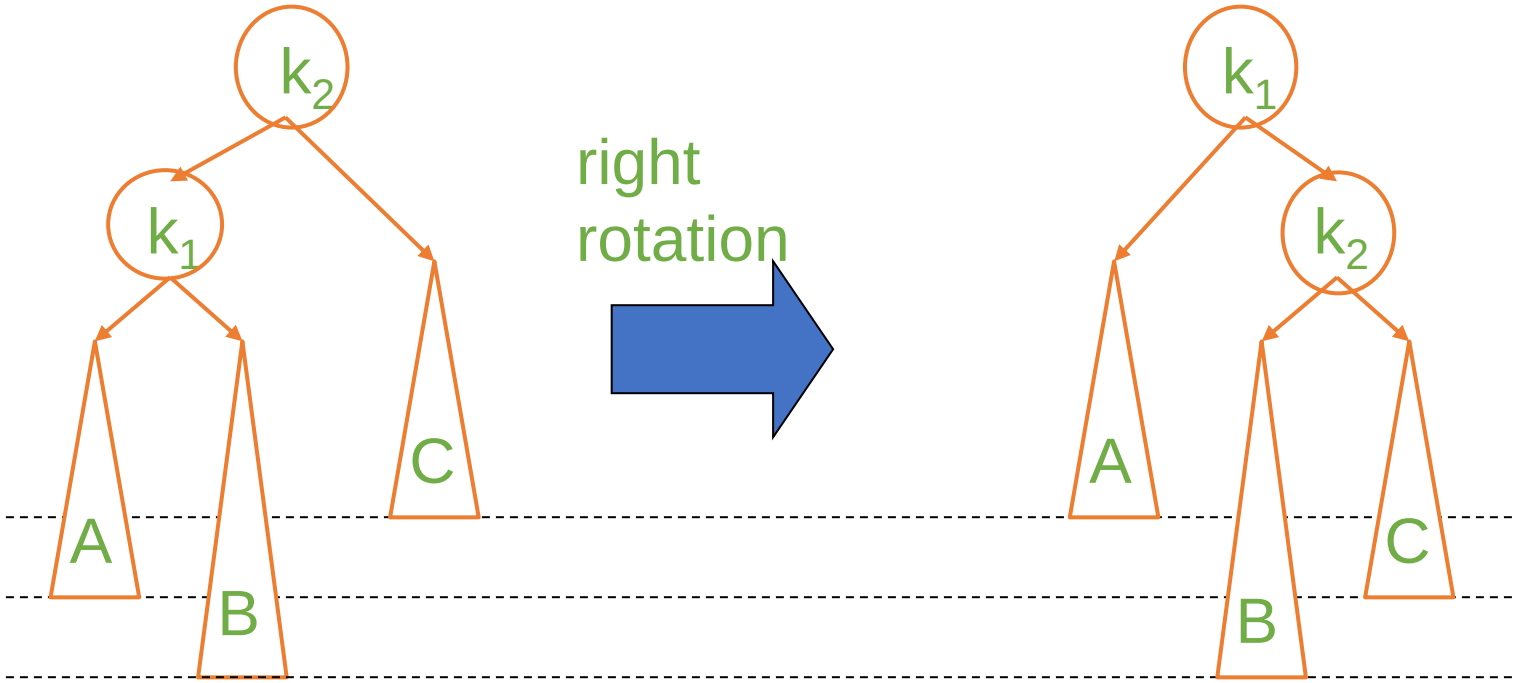


Phân tích giống tình huống 1

Ví dụ: Tình huống 2 xảy ra sau khi thực hiện loại bỏ

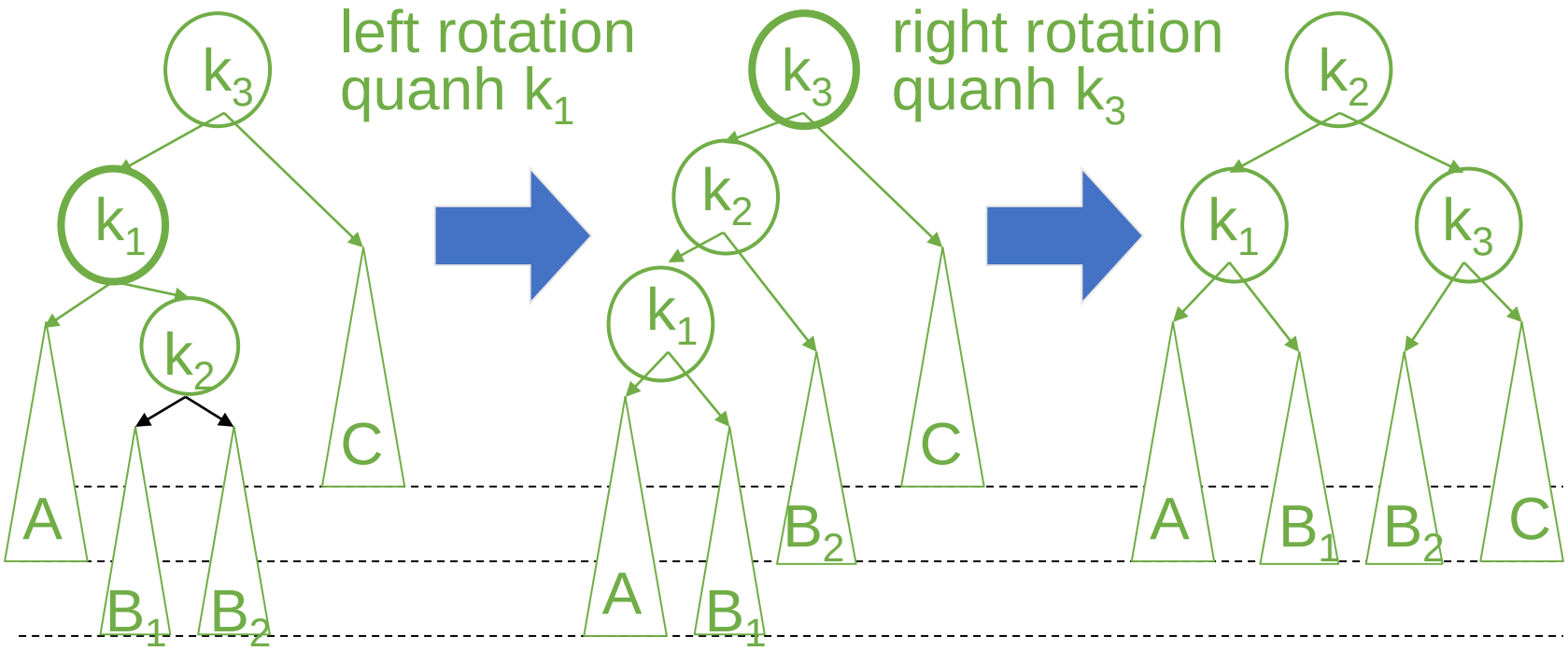


Xử lý tình huống 3 – thử quay đơn …

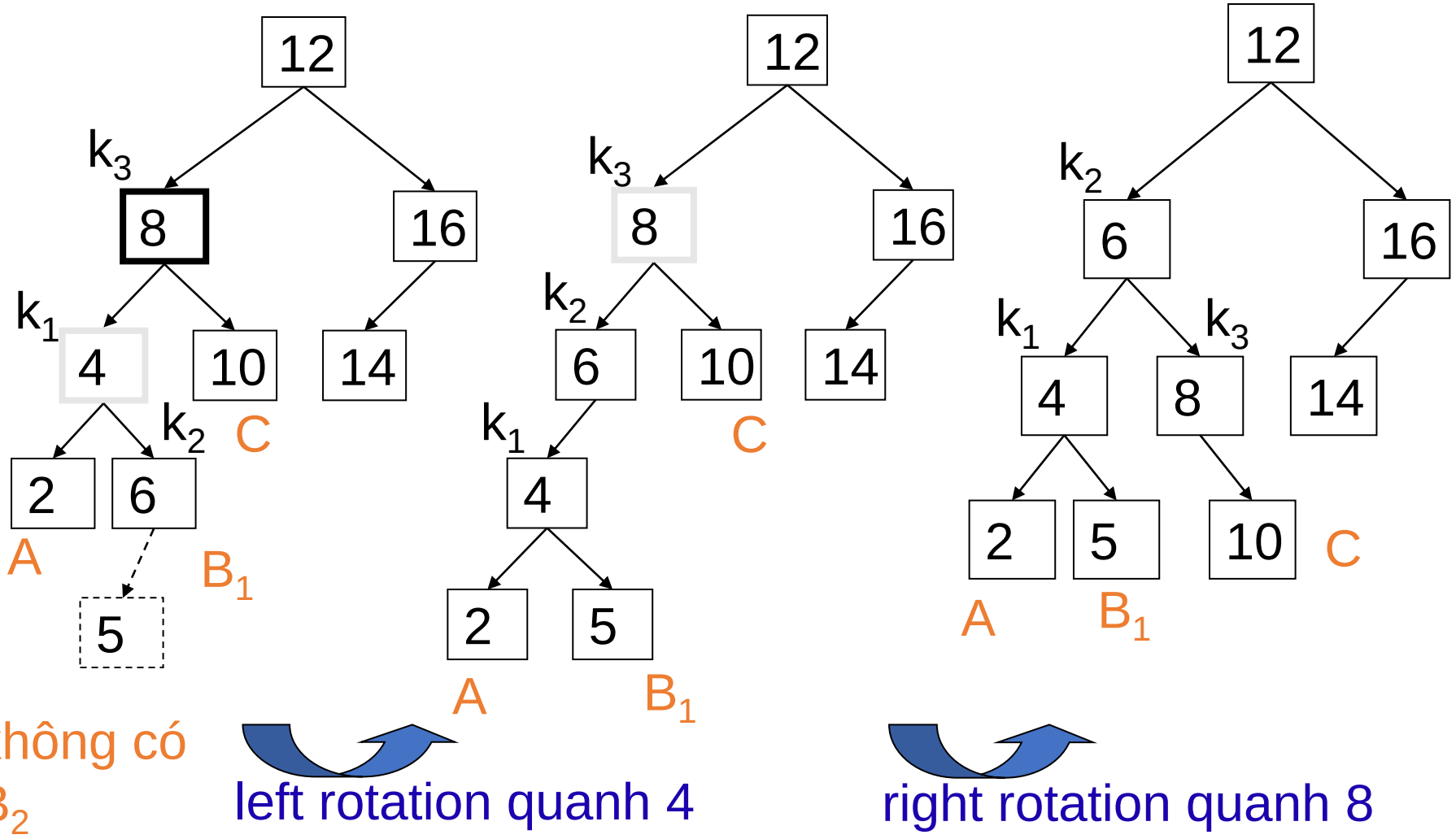


- Một phép quay là chưa đủ khôi phục cân bằng
- Phải thực hiện loạt phép quay quanh cây con gốc tại k_1

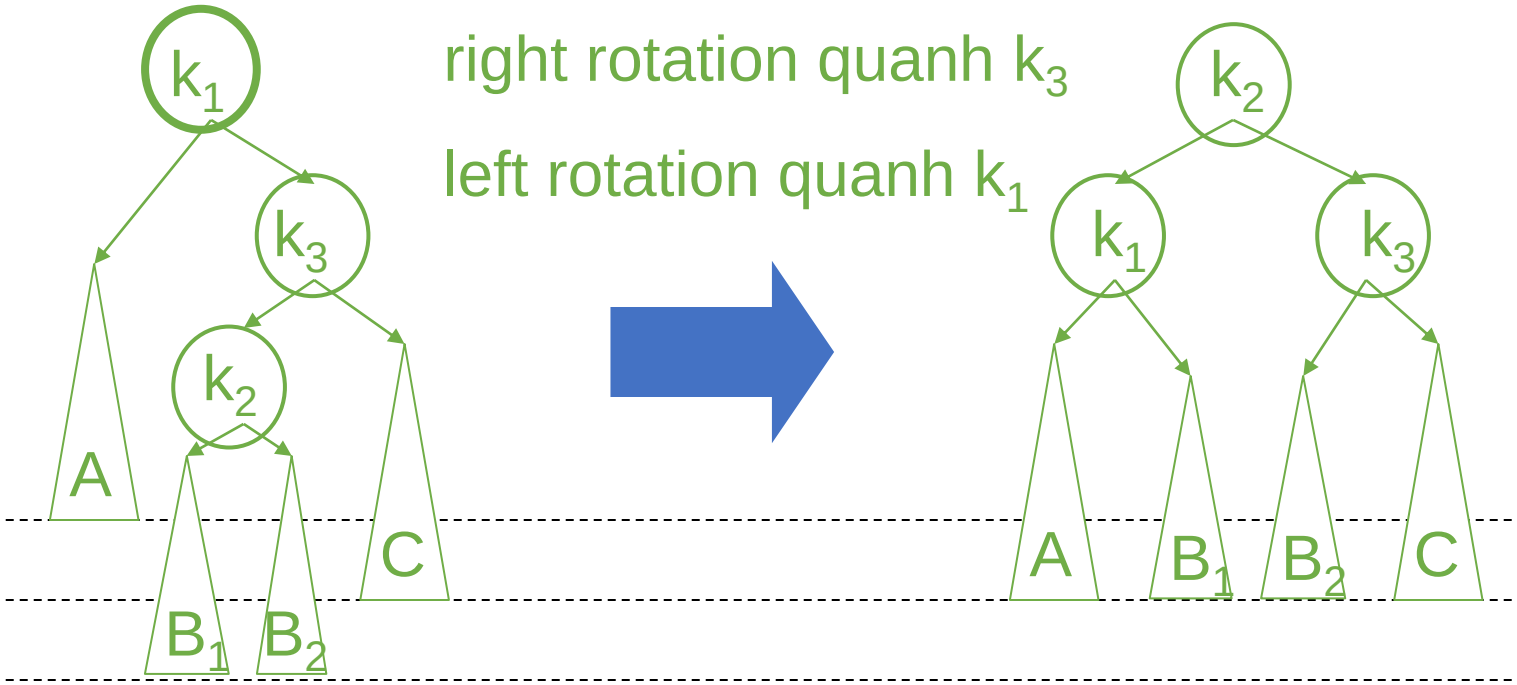
Xử lý tình huống 3:
Thực hiện phép quay kép, quay trái rồi quay phải



Ví dụ: Tình huống 3 sau khi thực hiện bổ sung

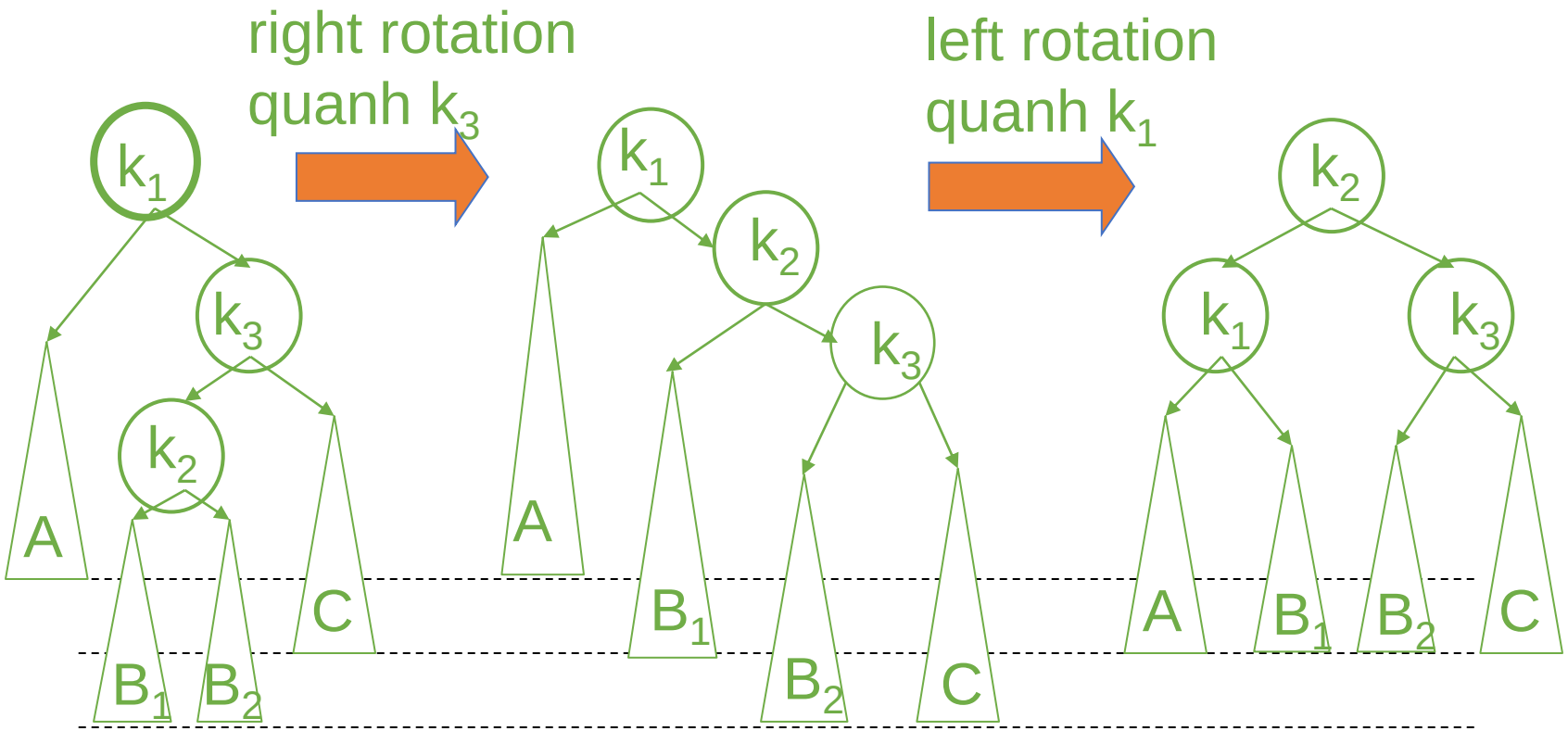


Xử lý tình huống 4:
Thực hiện phép quay kép, quay phải rồi quay trái



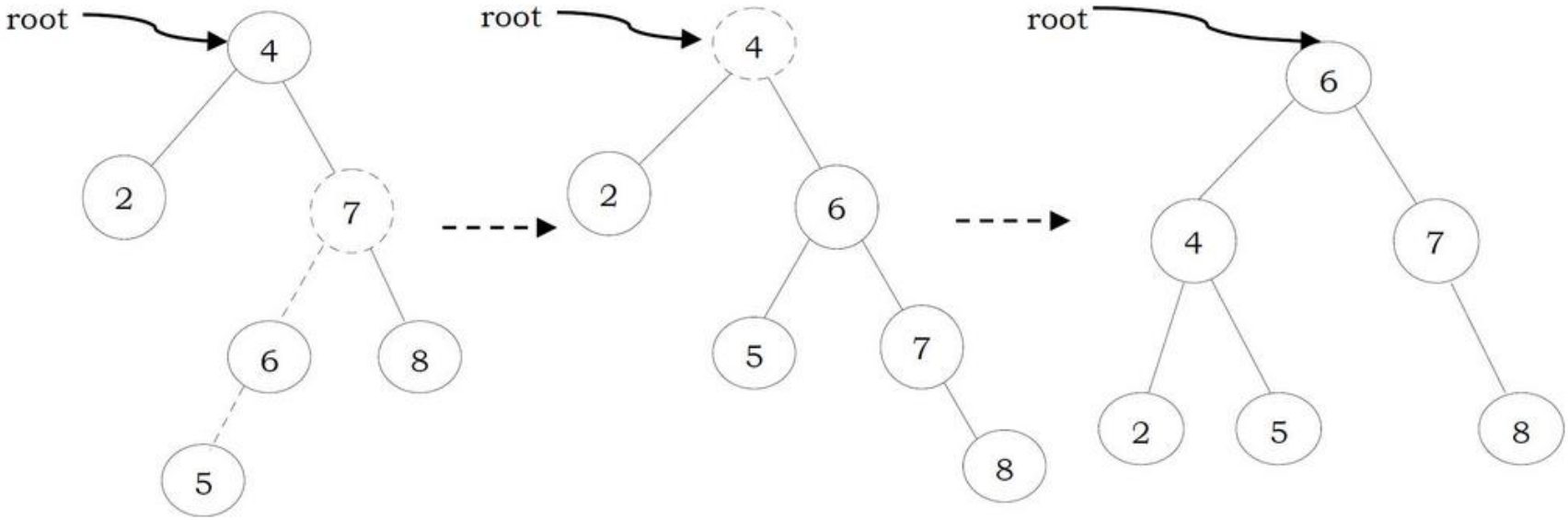
Phân tích tương tự như tình huống 3

Xử lý tình huống 4:
Thực hiện phép quay kép, quay phải rồi quay trái



Phân tích tương tự như tình huống 3

Ví dụ



Thuật toán bổ sung

- Chèn nút mới được thực hiện giống như trong cây BST.
- Lần theo đường đi từ nút mới bổ sung về gốc, với mỗi nút *x* kiểm tra $| h_{left}(x) - h_{right}(x) | \leq 1$.
- Nếu đúng, thì tiếp tục với parent(*x*). Nếu trái lại, cần khôi phục cân bằng nhờ thực hiện phép quay đơn hoặc phép quay kép.
- **Chú ý:** Khi một nút *x* khôi phục được tính cân bằng, thì đồng thời cũng khôi phục được tính cân bằng của tất cả các tổ tiên của nó. Nghĩa là việc khôi phục chỉ phải tiến hành với không quá một nút vi phạm.
- khôi phục cân bằng của 1 nút đòi hỏi thời gian $O(1)$, ➔ thao tác bổ sung đòi hỏi thời gian $O(h)$.

Thuật toán loại bỏ (Deletion)

- Thực hiện loại bỏ nút *x* giống như đối với cây BST.
- Tiếp đến duyệt từ nút lá mới đến gốc.
- Với mỗi nút *x* trên đường đi, kiểm tra tính cân bằng của nó.
  - Nếu *x* là cân bằng thì tiếp tục với parent(*x*).
  - Trái lại, khôi phục cân bằng của nút *x* nhờ thực hiện các phép quay.
  - tiếp tục quá trình cho đến khi gặp gốc của cây, bởi vì việc khôi phục cân bằng của nút *x* không đảm bảo khôi phục được tính cân bằng của tổ tiên của nó.

Thuật toán loại bỏ (Deletion)

- Dễ thấy việc khôi phục cân bằng của một nút đòi hỏi thời gian $O(1)$,
  còn đường đi tới gốc có độ dài không quá h,
  nên thao tác loại bỏ đòi hỏi thời gian $O(h)$.

- **Chú ý:** Do cài đặt thao tác xoá đối với cây AVL là khá phức tạp, nên nếu việc loại bỏ không phải làm thường xuyên thì có thể sử dụng ý tưởng "*lazy deletion*": chỉ đánh dấu xoá chứ không cần thực sự xoá!

Ưu điểm và nhược điểm của cây AVL

- Ưu điểm của cây AVL:
  1. Thời gian tìm kiếm là $O(\log n)$ và cây AVL luôn là cân bằng.
  2. Thao tác Bổ sung và Loại bỏ cũng chỉ tốn thời gian $O(\log n)$
  3. Việc khôi phục cân bằng không quá tốn kém.

- Nhược điểm của việc sử dụng cây AVL:
  1. Khó cài đặt và gỡ rối; cần thêm bộ nhớ cho yếu tố cân bằng.
  2. Thời gian là nhanh hơn tiệm cận nhưng tốn thời gian khôi phục cân bằng.
  3. Việc tìm kiếm với dữ liệu lớn thường thực hiện trên cơ sở dữ liệu trên đĩa và khi đó người ta sử dụng các cấu trúc khác (ví dụ, B-trees).
  4. Có thể chấp nhận thời gian $O(n)$ đối với vài thao tác riêng lẻ, nếu như tổng thời gian thực hiện nhiều thao tác khác lại là nhanh (ví dụ, sử dụng cây dẹt - Splay trees).

6.4. Bảng băm

6.4.1. Đặt vấn đề

6.4.2. Địa chỉ trực tiếp

6.4.3. Hàm băm

6.4.1. Đặt vấn đề

- Cho bảng *T* và bản ghi *x*, với **khoá** và **dữ liệu** đi kèm, ta cần hỗ trợ các thao tác sau:
  - Insert (*T*, *x*)
  - Delete (*T*, *x*)
  - Search(*T*, *x*)
- Ta muốn thực hiện các thao tác này một cách nhanh chóng mà không phải thực hiện việc sắp xếp các bản ghi.

Bảng băm

- (hash table) là cách tiếp cận giải quyết vấn đề đặt ra.
- Cấu trúc dữ liệu hiệu quả để cài đặt các từ điển (dictionaries).
- Có thể xem như sự mở rộng của mảng thông thường. Việc địa chỉ hoá trực tiếp trong mảng cho phép truy nhập đến phần tử bất kỳ trong thời gian $O(1)$.

Bảng băm

- Trong tình huống xấu nhất việc tìm kiếm đòi hỏi thời gian $O(n)$ giống như danh sách móc nối,
- Trên thực tế bảng băm làm việc hiệu quả hơn nhiều.
- Với một số giả thiết khá hợp lý, việc tìm kiếm phần tử trong bảng băm đòi hỏi thời gian $O(1)$.

- Trong mục này ta sẽ chỉ xét khoá là các số nguyên dương (có thể rất lớn)

6.4. Bảng băm

6.4.1. Đặt vấn đề

6.4.2. Địa chỉ trực tiếp

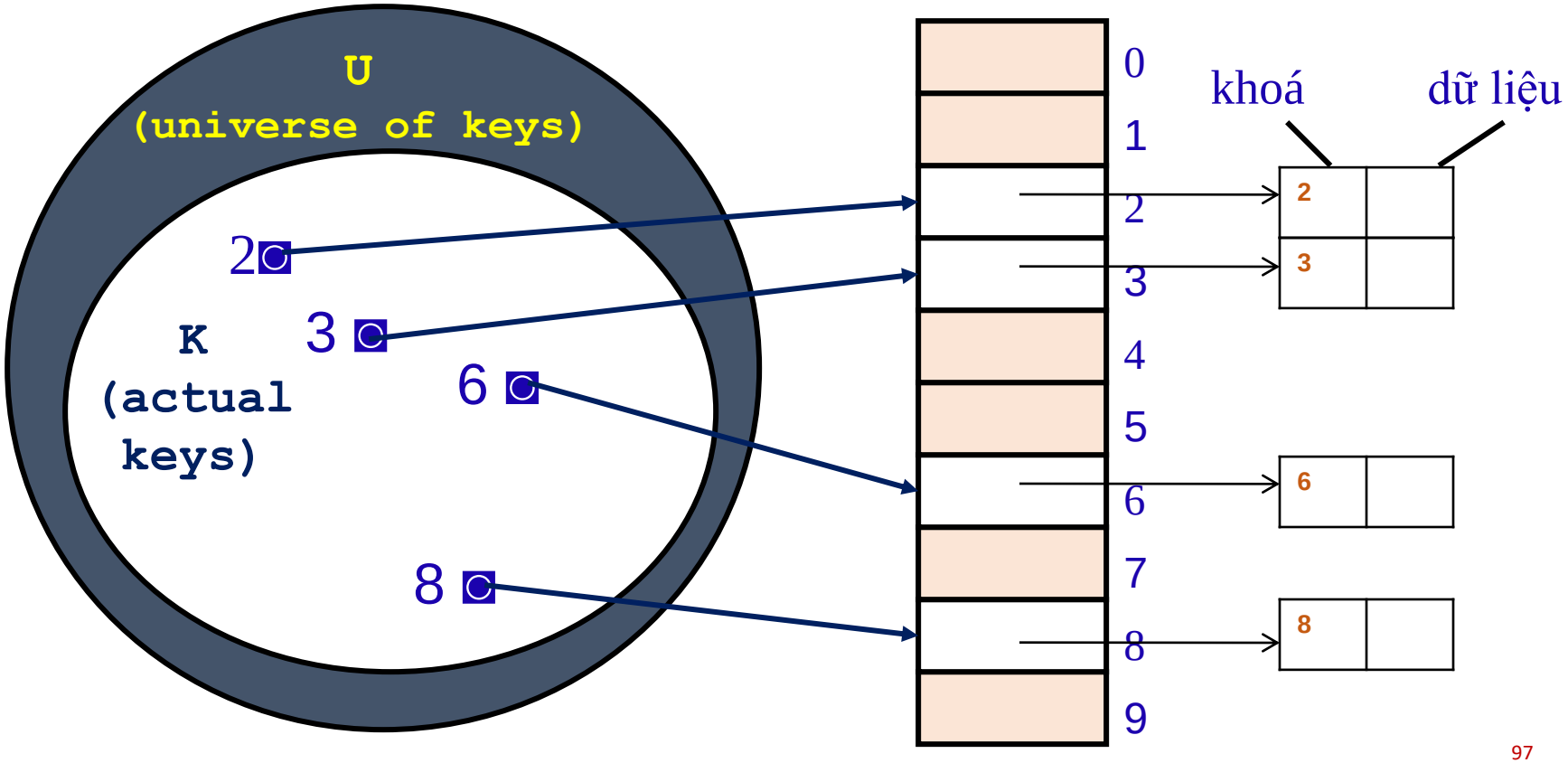
6.4.3. Hàm băm

Địa chỉ trực tiếp
Direct Addressing

- **Giả thiết rằng:**
  - Các khoá là các số trong khoảng từ 0 đến m-1
  - Các khoá là khác nhau từng đôi
- **Ý tưởng:**
  - Thiết lập mảng T[0..m-1] trong đó
    - T[i] = x nếu $x \in T$ và key[x] = i
    - T[i] = NULL nếu trái lại
  - T được gọi là bảng địa chỉ trực tiếp (*direct-address table*), các phần tử trong bảng T sẽ được gọi là các ô.

Ví dụ

Tạo bảng địa chỉ trực tiếp T. Mỗi khoá trong tập U = {0,1, . . . , 9} tương ứng với một chỉ số trong bảng. Tập K = {2, 3, 6, 8} gồm các khoá thực có xác định các ô trong bảng chứa con trỏ đến các phần tử. Các ô khác (được tô màu) chứa con trỏ NULL.



Các phép toán

- Các phép toán được cài đặt một cách trực tiếp:

- DIRECT-ADDRESS-SEARCH(T,k)

  return $T[k]$

- DIRECT-ADDRESS-INSERT(T,x)

  $T[key[x]] = x$

- DIRECT-ADDRESS-DELETE(T,x)

  $T[key[x]]$ = NULL

- **Thời gian thực hiện mỗi phép toán đều là $O(1)$.**

Hạn chế của phương pháp địa chỉ trực tiếp

The Problem With Direct Addressing

- Phương pháp địa chỉ trực tiếp làm việc tốt nếu như biên độ *m* của các khoá là tương đối nhỏ.
- Nếu các khoá là các số nguyên 32-bit thì sao?
  - Vấn đề 1: bảng địa chỉ trực tiếp sẽ phải có 2^{32} (hơn 4 tỷ) phần tử
  - Vấn đề 2: ngay cả khi bộ nhớ không là vấn đề, thì thời gian khởi tạo các phần tử là NULL cũng là rất tốn kém
- Cách giải quyết: Ánh xạ khoá vào khoảng biến đổi nhỏ hơn 0..*m*-1
- Ánh xạ này được gọi là hàm băm (*hash function*)

6.4. Bảng băm

6.4.1. Đặt vấn đề

6.4.2. Địa chỉ trực tiếp

6.4.3. Hàm băm

Hàm băm

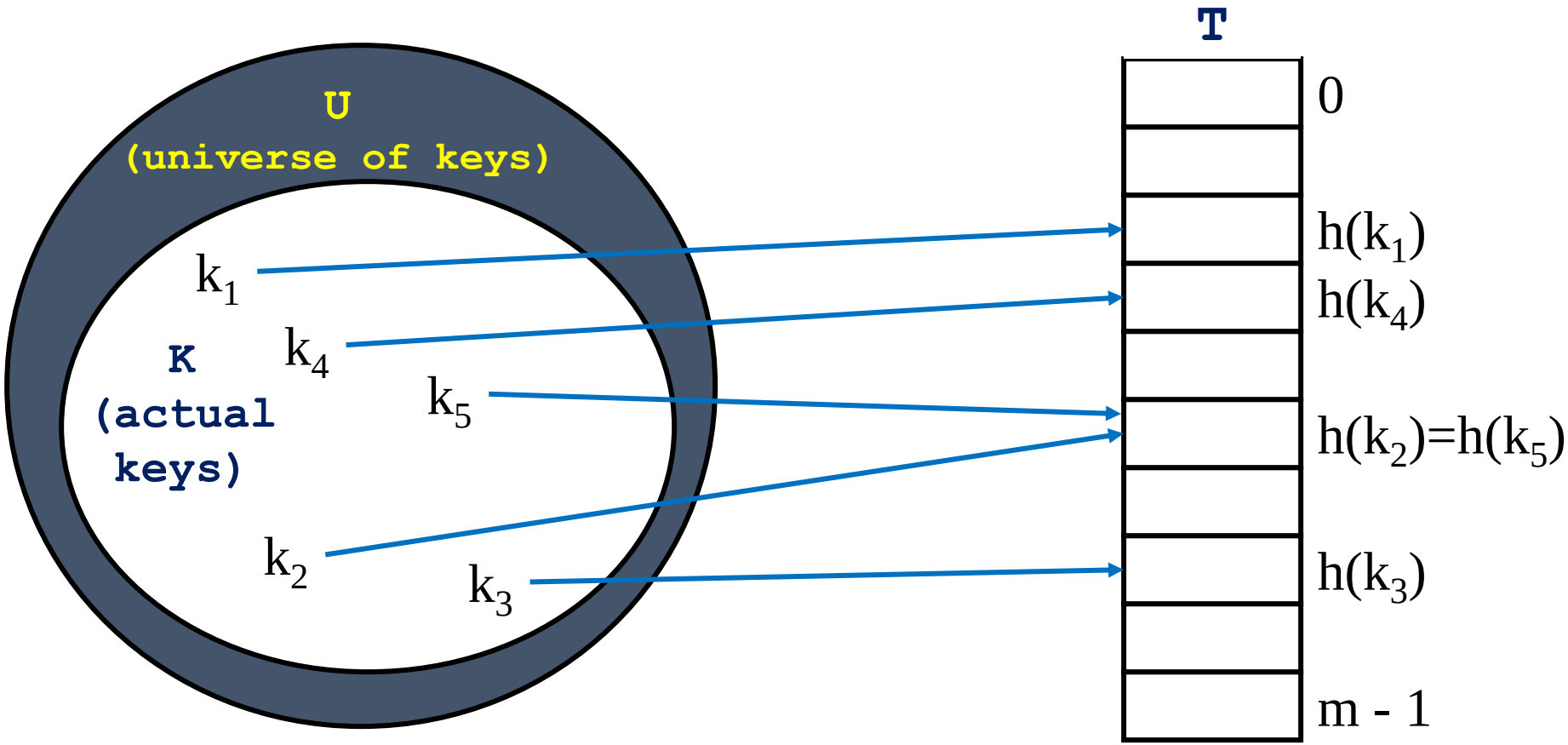
- Trong phương pháp địa chỉ trực tiếp, phần tử với khoá *k* được cất giữ ở ô *k*.
- Với bảng băm phần tử với khoá *k* được cất giữ ở ô *h*(*k*), trong đó ta sử dụng hàm băm *h* để xác định ô cất giữ phần tử này từ khoá của nó (*k*).
- ***Định nghĩa.*** Hàm băm *h* là ánh xạ từ không gian khoá *U* vào các ô của bảng băm T[0..m-1]:

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

- Ta sẽ nói rằng phần tử với khoá *k* được *gắn vào* ô *h*(*k*) và nói *h*(*k*) là *giá trị băm* của khoá *k*.

Hash Functions

- Vấn đề nảy sinh lại là xung đột (*collision*), khi nhiều khoá được đặt tương ứng với cùng một ô trong bảng địa chỉ T.



Giải quyết xung đột

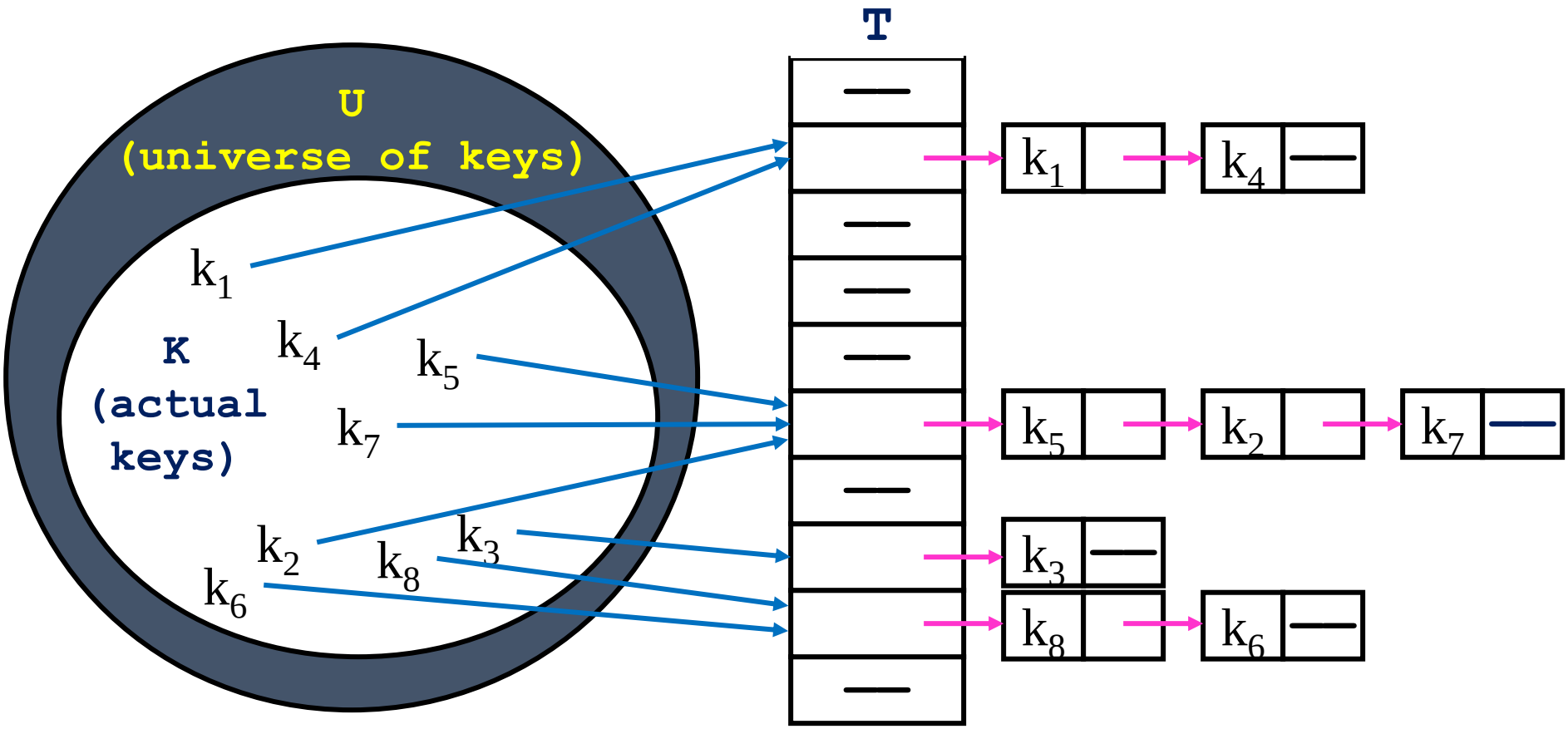
- *Ta cần giải quyết xung đột như thế nào?*
- Cách giải quyết 1:

  Tạo chuỗi (chaining)
- Cách giải quyết 2:

  Phương pháp địa chỉ mở (open addressing)

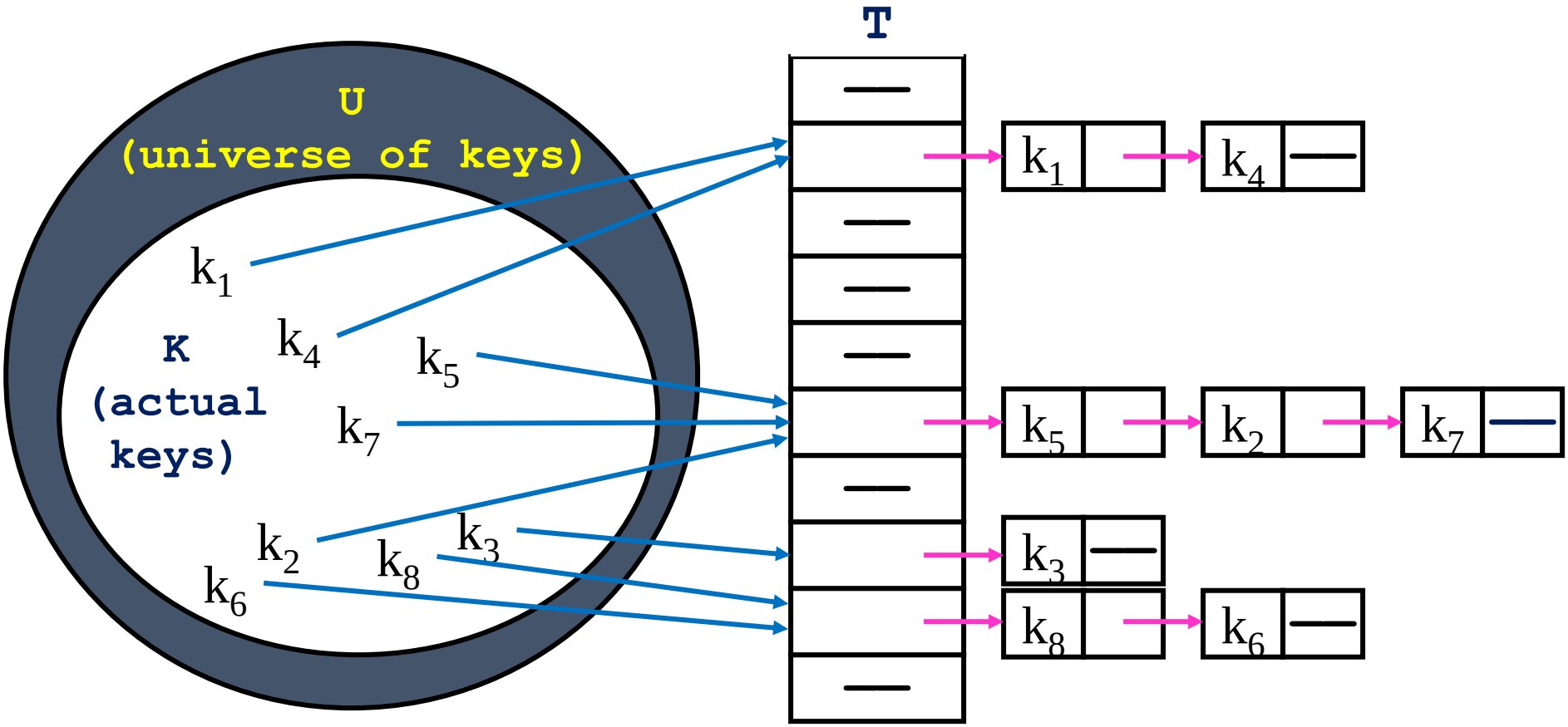
Tạo chuỗi (Chaining)

- Theo phương pháp này, ta sẽ tạo danh sách móc nối để chứa các phần tử được gắn với cùng một vị trí trong bảng.



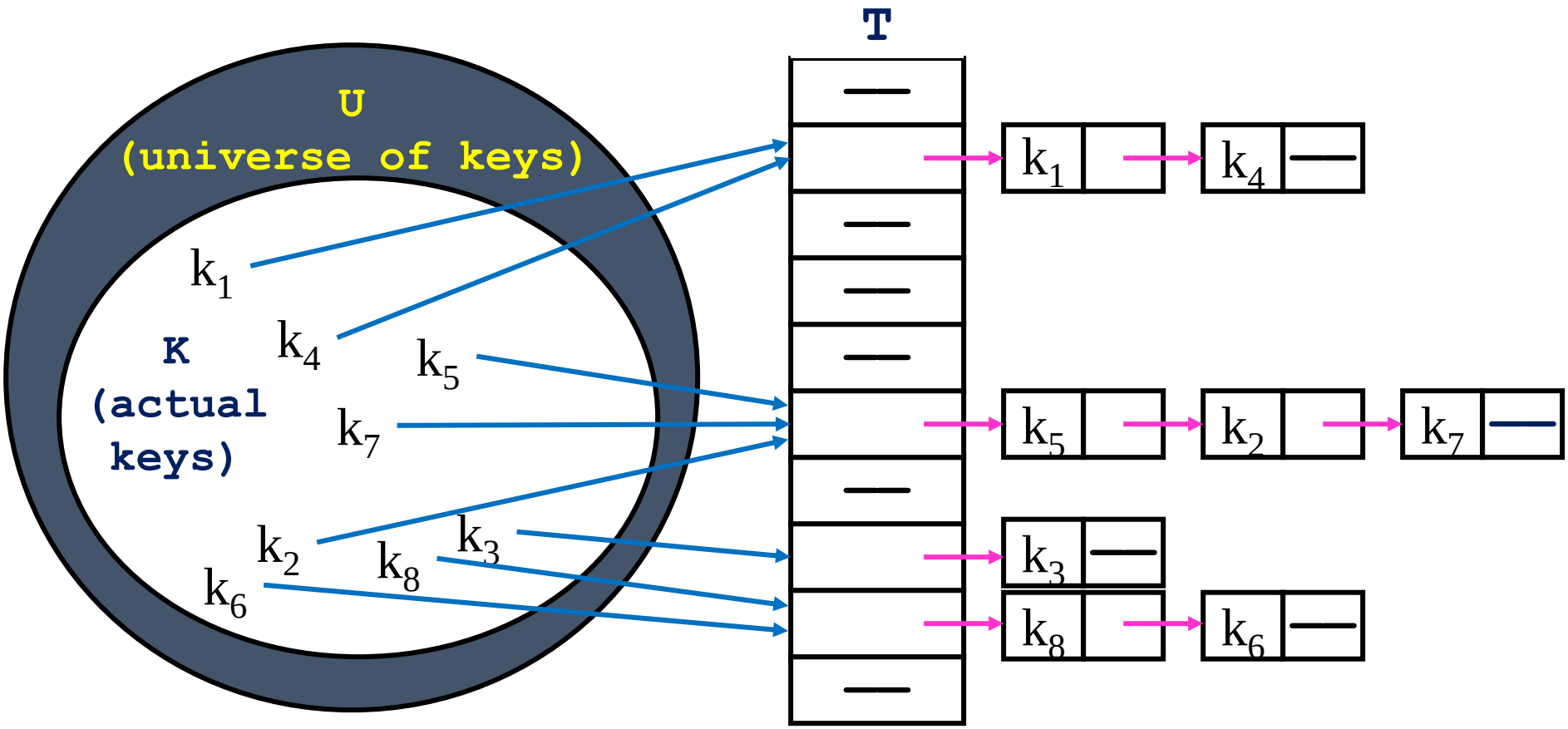
Tạo chuỗi (Chaining)

- *Ta cần thực hiện bổ sung phần tử như thế nào?*
- (Như bổ sung vào danh sách móc nối)



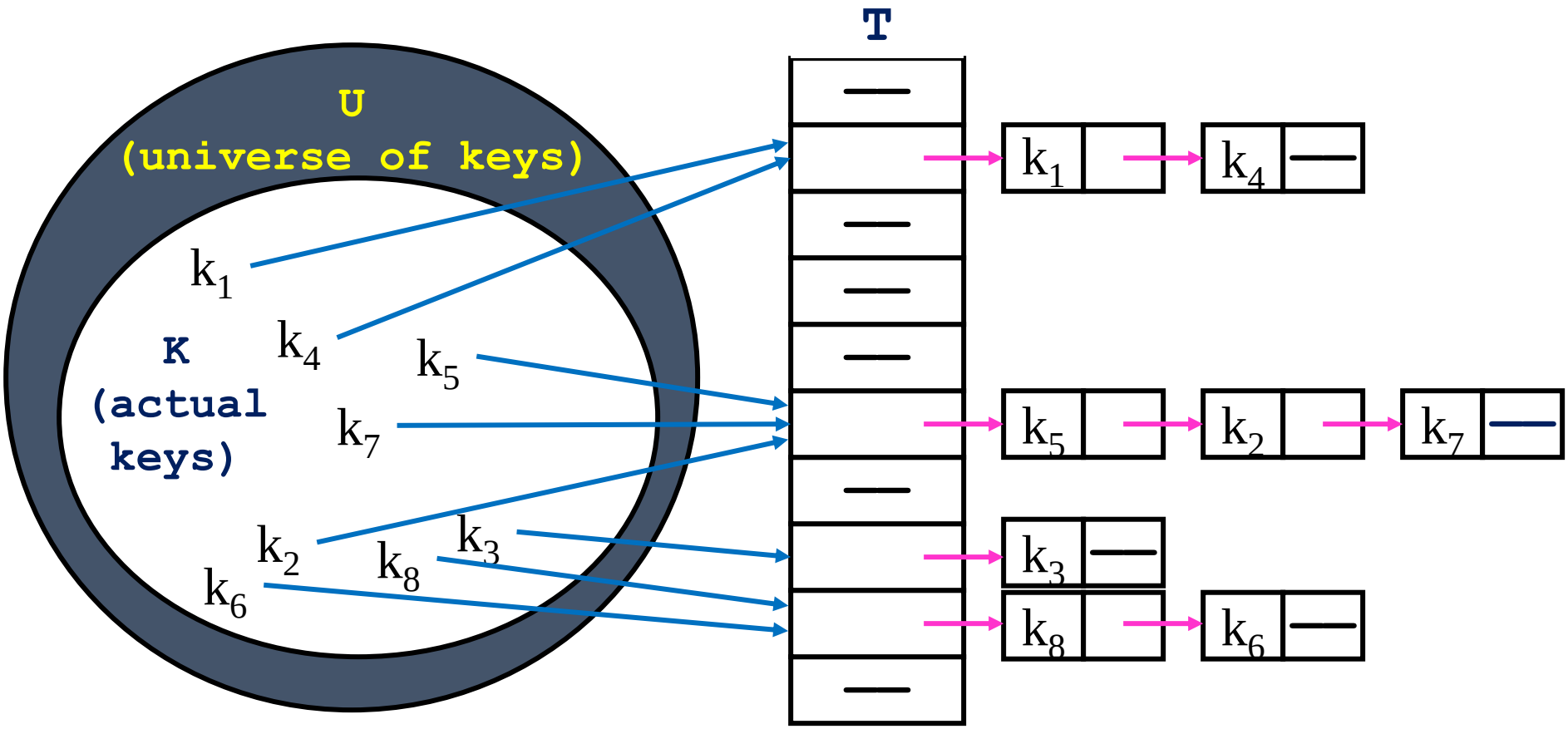
Tạo chuỗi (Chaining)

- *Ta cần thực hiện loại bỏ phần tử như thế nào? Có cần sử dụng danh sách nối đôi để thực hiện xoá một cách hiệu quả không?*
- *(Không ! Vì thông thường chuỗi có độ dài không lớn)*



Tạo chuỗi (Chaining)

- *Thực hiện tìm kiếm phần tử với khoá cho trước như thế nào?*
- *Tìm kiếm trên danh sách móc nối trỏ bởi T[h(k)]*



Các thao tác

- CHAINED-HASH-INSERT(*T, x*)

  chèn *x vào đầu danh sách móc nối T*[*h*(*key*[*x*])]
- CHAINED-HASH-SEARCH(*T, k*)

  tìm phần tử với khoá *k trong danh sách T*[*h*(*k*)]
- CHAINED-HASH-DELETE(*T, x*)

  xoá *x khỏi danh sách T* [*h*(*key*[*x*])]

Phân tích phương pháp chuỗi

- Giả sử rằng thực hiện điều kiện *simple uniform hashing*: Mỗi khoá trong bảng là đồng khả năng được gán với một ô bất kỳ.
- Cho n khoá và m ô trong bảng, ta định nghĩa nhân tử nạp (*load factor*) $\alpha = n/m$ = Số lượng khoá trung bình trên một ô
- Khi đó có thể chứng minh được rằng
- *Chi phí trung bình để phát hiện một khoá không có trong bảng là* $O(1+\alpha)$
- *Chi phí trung bình để phát hiện một khoá có trong bảng là* $O(1+\alpha)$
- Do đó chi phí tìm kiếm là $O(1+\alpha)$

Phân tích tạo chuỗi

- Như vậy chi phí của thao tác tìm kiếm là $O(1 + \alpha)$
- *Nếu số lượng khoá n là tỷ lệ với số lượng ô trong bảng thì α có giá trị là bao nhiêu?*
- Trả lời: $\alpha = O(1)$
  - Nói cách khác, ta có thể đảm bảo chi phí tìm kiếm mong đợi là hằng số nếu ta đảm bảo α là hằng số

Địa chỉ mở
Open Addressing

- **Ý tưởng cơ bản:**
  - Khi bổ sung (Insert): nếu ô là đã bận, thì ta tìm kiếm ô khác, ..., cho đến khi tìm được ô rỗng (phương pháp dò thử - *probing*).
  - Để tìm kiếm (search), ta sẽ tìm dọc theo dãy các phép dò thử giống như dãy dò thử khi thực hiện chèn phần tử vào bảng.
    - Nếu tìm được phần tử với khoá đã cho thì trả lại nó,
    - Nếu tìm được con trỏ NULL,thì phần tử cần tìm không có trong bảng
- Ý tưởng này áp dụng tốt khi ta làm việc với tập cố định (chỉ có bổ sung mà không có xoá)
  - Ví dụ: khi kiểm lỗi chính tả (spell checking)
- Bảng có kích thước không cần lớn hơn *n* quá nhiều

Địa chỉ mở

Xét chi tiết 2 kỹ thuật:

- Dò tuyến tính (Linear Probing)
- Hàm băm kép (Double Hashing)

Dò tuyến tính (Linear Probing)

Nếu vị trí hiện tại đã bận, ta dò kiếm vị trí tiếp theo trong bảng.

```
LinearProbingInsert(k)  {
   if (table is full) error;
   probe = h(k);
   while (table[probe] is occupied)
            probe = (probe+1) % m; //m – kích thước bảng
   table[probe] = m;
}
```

Di chuyển dọc theo bảng cho đến khi tìm được vị trí rỗng

- **Ưu điểm:** Đòi hỏi bộ nhớ ít hơn phương pháp tạo chuỗi (không có móc nối)
- **Hạn chế:** Đòi hỏi nhiều thời gian hơn tạo chuỗi (nếu đường dò kiếm là dài)
- Thực hiện xoá bằng cách đánh dấu xoá (đánh dấu ô đã bị xoá)

Double Hashing

Ý tưởng: Nếu vị trí hiện tại là bận, tìm vị trí khác trong bảng nhờ sử dụng hai hàm băm

```
DoubleHashInsert(k)  {
    if (table is full) error;
    probe = h1(k);
    offset = h2(k);
    while (table[probe] is occupied)
              probe = (probe+offset) % m; //m – kích thước
    bảng
    table[probe] = k;
}
```

- Dễ thấy: Nếu m là nguyên tố, thì ta sẽ dò thử tất cả các vị trí
- Ưu (nhược) điểm được phân tích tương tự như dò tuyến tính
- Ngoài ra, các khoá được rải đều hơn là dò tuyến tính

Kết quả lý thuyết:

Số phép thử trung bình

	Không tìm được	Tìm được
Chaining	$1+\alpha$	$1+\frac{\alpha}{2}$
Linear Probing	$\frac{1}{2}+\frac{1}{2(1-\alpha)^2}$	$\frac{1}{2}+\frac{1}{2(1-\alpha)}$
Double Hashing	$\frac{1}{(1-\alpha)}$	$\frac{1}{\alpha}\ln\frac{1}{(1-\alpha)}$

α = <số lượng phần tử trong bảng>/<kích thước bảng>

Chọn hàm băm

- Rõ ràng việc chọn hàm băm tốt sẽ có ý nghĩa quyết định
  - *Thời gian tính của hàm băm là bao nhiêu?*
  - *Thời gian tìm kiếm sẽ như thế nào?*
- Một số yêu cầu đối với hàm băm:
  - Phải phân bố đều các khoá vào các ô
  - Không phụ thuộc vào khuôn mẫu trong dữ liệu

Hash Functions:
Phương pháp chia (The Division Method)

- $h(k) = k \bmod m$
  - nghĩa là: gắn k vào bảng có m ô nhờ sử dụng ô xác định bởi phần dư của phép chia k cho m
- *Điều gì xảy ra nếu m là luỹ thừa của* 2 (*chẳng hạn* 2^p)*?*
- Ans: khi đó $h(k)$ chính là p bít cuối của k
- *Điều gì xảy ra nếu m là luỹ thừa* 10 (*chẳng hạn* 10^p)*?*
- Ans: khi đó $h(k)$ chỉ phụ thuộc vào p chữ số cuối của k

- Vì thế, thông thường người ta chọn kích thước bảng m là số nguyên tố không quá gần với luỹ thừa của 2 (hoặc 10)

Hash Functions:
Phương pháp nhân (The Multiplication Method)

- Phương pháp nhân để xây dựng hàm băm được tiến hành theo hai bước. Đầu tiên ta nhân k với một hằng số A, $0 < A < 1$ và lấy phần thập phân của kA. Sau đó, ta nhân giá trị này với m rồi lấy phần nguyên của kết quả:
  - Chọn hằng số A, $0 < A < 1$:
  - $h(k) = \lfloor m\ (kA - \lfloor kA \rfloor) \rfloor$

Phần thập phân của kA

- Chọn $m = 2^p$
- Chọn A không quá gần với 0 hoặc 1
- Knuth: Hãy chọn $A = (\sqrt{5} - 1)/2$

QUESTIONS?