HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

Tìm kiếm tuần tự và tìm kiếm nhị phân

ONE LOVE. ONE FUTURE.

MỤC TIÊU

Sau bài học này, người học có thể:

1. Hiểu được khái niệm thuật toán tìm kiếm tuần tự và tìm kiếm nhị phân

2. Cài đặt được thuật toán

NỘI DUNG TIẾP THEO

1. **Tìm kiếm tuần tự**
    1.1. Bài toán tìm kiếm
    1.2. Mã cài đặt
    1.3. Phân tích thời gian tính

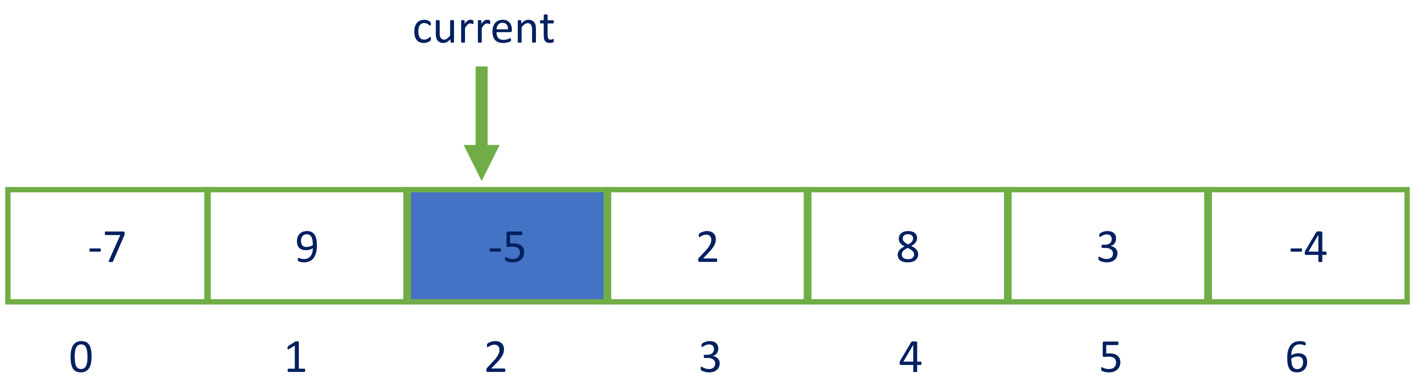
2. Tìm kiếm nhị phân

1. TÌM KIẾM TUẦN TỰ

- 1.1. 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 $a_i = x$.

1. TÌM KIẾM TUẦN TỰ

- 1.1. Bài toán tìm kiếm



- 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: $O(n)$

1. TÌM KIẾM TUẦN TỰ

- 1.2. Mã cài đặt

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

1. TÌM KIẾM TUẦN TỰ

- 1.3. Phân tích thời gian tính
  - Độ dài đầu vào: n
  - Đánh giá số lần thực hiện

    (*) **(a[i] == x)** trong vòng lặp for.
  - Nếu $a[1] = x$
    - (*) phải thực hiện 1 lần.

      → thời gian tính tốt nhất: $\Theta(1)$.
  - Nếu x không có mặt trong dãy
    - (*) phải thực hiện n lần.

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

1. TÌM KIẾM TUẦN TỰ

- 1.3. Phân tích thời gian tính
  - Nếu *x* tìm thấy ở vị trí thứ *i* của dãy (*x* = *a*[*i*])
    - (*) phải thực hiện *i* lần (*i* = 1, 2, ..., *n*).

      → số lần trung bình phải thực hiện phép so sánh (*):

$$
\begin{aligned}
&[(1 + 2 + \ldots + n) + n]/(n+1) \\
&= [n + n(n+1)/2]/(n+1) \\
&= (n^2 + 3n)/[2(n+1)].
\end{aligned}
$$

  - Ta có:

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

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

NỘI DUNG TIẾP THEO

1. Tìm kiếm tuần tự

2. Tìm kiếm nhị phân

  2.2. Ví dụ

  2.3. Cài đặt thuật toán bằng vòng lặp

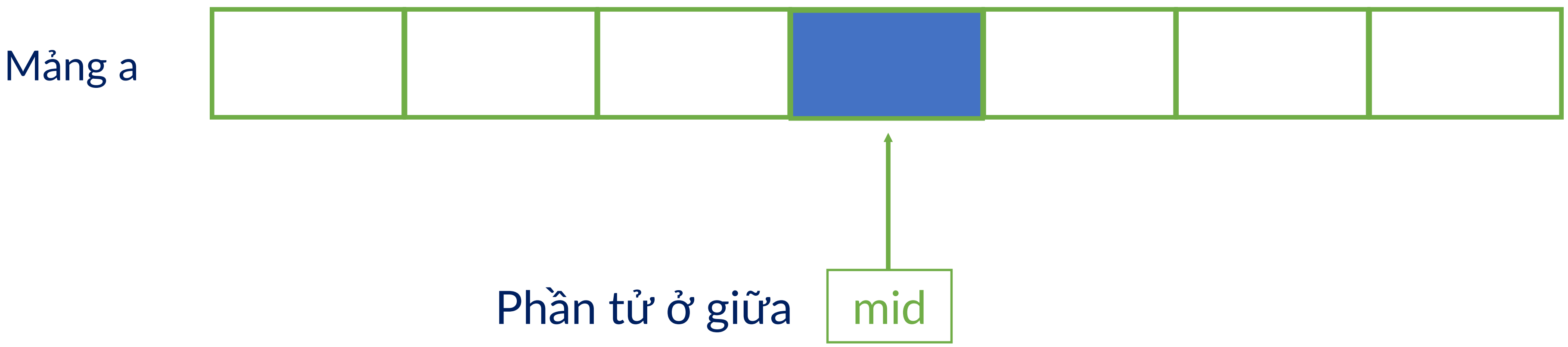
  2.4. Cài đặt thuật toán bằng đệ qui

2. TÌM KIẾM NHỊ PHÂN

- **2.1. Bài toán**
  - Cho mảng số $a[1..n]$ được sắp xếp theo thứ tự không giảm và số x.

    Cần tìm chỉ số i (1 ≤ i ≤ n) sao cho $a[i] = x$.
  - Nhận xét: x có trong mảng a thì hoặc là x

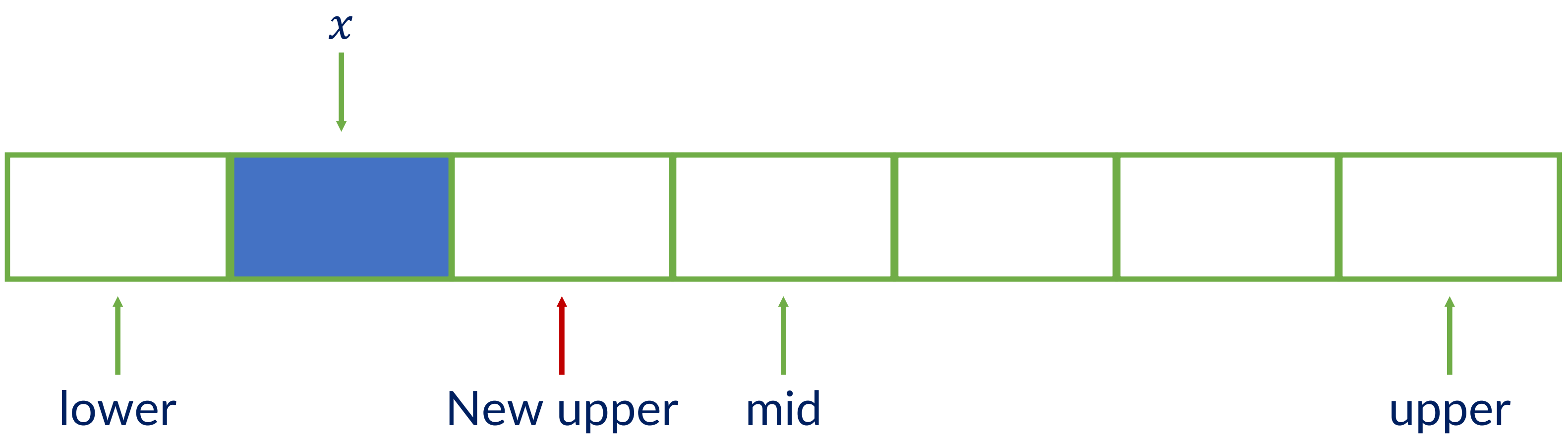
    (1) bằng phần tử nằm ở vị trí ở giữa mảng a



- **2.1. Bài toán**
  - Cho mảng số $a[1..n]$ được sắp xếp theo thứ tự không giảm và số x.

    Cần tìm chỉ số i (1 ≤ i ≤ n) sao cho $a[i] = x$.
  - Nhận xét: x có trong mảng a thì hoặc là x

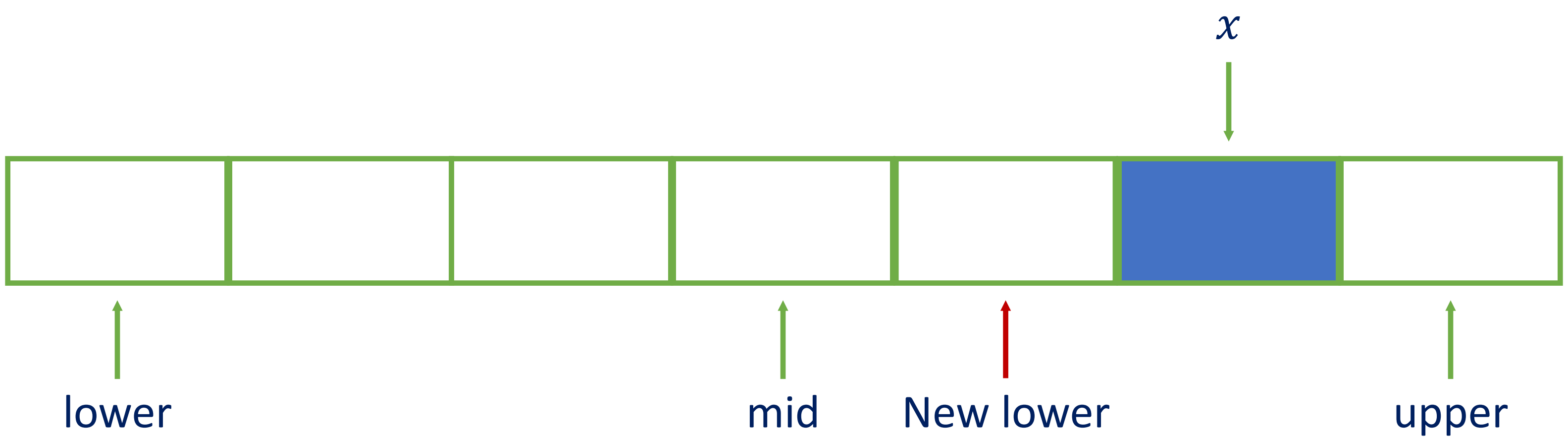
  (2) nằm ở nửa bên trái (L) của mảng a



- **2.1. Bài toán**
  - Cho mảng số $a[1..n]$ được sắp xếp theo thứ tự không giảm và số x.

    Cần tìm chỉ số i (1 ≤ i ≤ n) sao cho $a[i] = x$.
  - Nhận xét: x có trong mảng a thì hoặc là x

  (3) nằm ở nửa bên phải (R) của mảng a.



2. TÌM KIẾM NHỊ PHÂN

• 2.1. Bài toán

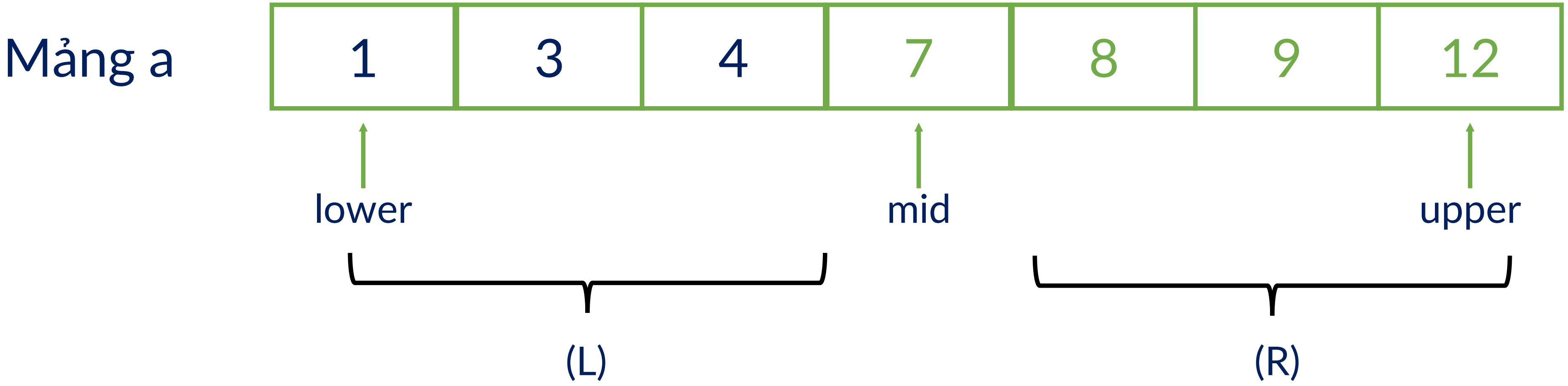
- Cho mảng số $a[1..n]$ được sắp xếp theo thứ tự không giảm và số x.

  Cần tìm chỉ số i (1 ≤ i ≤ n) sao cho $a[i] = x$.

- Nhận xét:
  - Tình huống (2) và (3) xảy ra chỉ khi x nhỏ hơn (lớn hơn) phần tử ở giữa của mảng a.)
  - → Lặp lại tìm x ở nửa (L) hoặc (R)

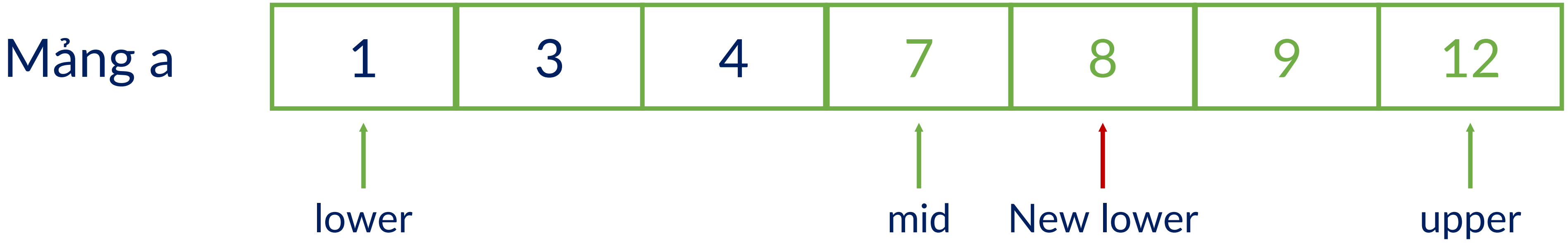
• 2.2. Ví dụ

- Cho mảng $a = \{1,3,4,7,8,9,12\}$ tìm số $x = 8$



$$\boxed{x\ != a[mid]}$$

- **2.2. Ví dụ**
  - Cho mảng $a = \{1,3,4,7,8,9,12\}$ tìm số $x = 8$



$\boxed{x > a[mid] \rightarrow \text{ Tìm ở nửa bên phải}}$

2. TÌM KIẾM NHỊ PHÂN

- **2.2. Ví dụ**
  - Cho mảng $a = \{1,3,4,7,8,9,12\}$ tìm số $x = 8$

Mảng a	1	3	4	7	8	9	12
					lower	mid	upper
					(L)		(R)

$x < a[mid] \rightarrow$ Tìm ở nửa bên trái

- **2.2. Ví dụ**
  - Cho mảng $a = \{1,3,4,7,8,9,12\}$ tìm số $x = 8$

Mảng a	1	3	4	7	8	9	12

lower mid upper

$\boxed{x = a[mid] \rightarrow \text{ Dừng và đưa ra kết quả}}$

2. TÌM KIẾM NHỊ PHÂN

- 2.3. Cài đặt thuật toán bằng vòng lặp

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

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

Độ phức tạp: $O(\log n)$

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

2. TÌM KIẾM NHỊ PHÂN

- 2.4. Cài đặt thuật toán bằng đệ qui

```
int binarySearch(float a[], int lower, int upper, float x)
{
    if (upper >= lower) {
        int mid = lower + (upper - lower) / 2;
        if (a[mid] == x)
            return mid;

        if (a[mid] > x)
            return binarySearch(a, lower, mid - 1, x);
        return binarySearch(a, mid + 1, upper, x);
    }
    return -1;
}
```

Độ phức tạp: O(log *n*)

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

2. TÌM KIẾM NHỊ PHÂN

- **Nhận xét**
  - Ưu điểm
    - Nhanh hơn tìm kiếm tuyến tính, đặc biệt đối với các mảng lớn.
    - Hiệu quả hơn các thuật toán tìm kiếm khác có độ phức tạp về thời gian tương tự (ví dụ: tìm kiếm nội suy hoặc tìm kiếm theo cấp số nhân).
    - Phù hợp để tìm kiếm các tập dữ liệu lớn được lưu trữ trong bộ nhớ ngoài, chẳng hạn như trên ổ cứng hoặc trên đám mây.
  - Nhược điểm
    - Mảng nên được sắp xếp.
    - Yêu cầu cấu trúc dữ liệu đang được tìm kiếm được lưu trữ ở các vị trí bộ nhớ liền kề.
    - Yêu cầu các phần tử của mảng phải có thể so sánh được, nghĩa là chúng phải có thứ tự.

Chương 8- Tìm kiếm

Bài 2. Cây nhị phân tìm kiếm

MỤC TIÊU

Sau bài học này, người học có thể:

1. Hiểu được khái niệm cây nhị phân tìm kiếm

2. Cài đặt được cây nhị phân tìm kiếm

NỘI DUNG TIẾP THEO

1. Định nghĩa

2. Biểu diễn cây nhị phân tìm kiếm

3. Các phép toán

1. ĐỊNH NGHĨA

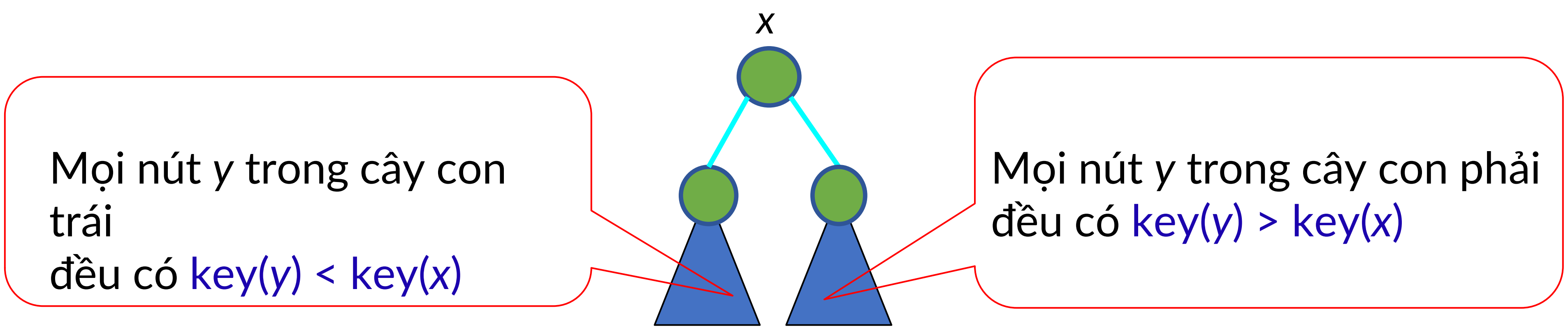
- Cây nhị phân có các tính chất sau:

  - ***Cây nhị phân tìm kiếm***: *cấu trúc dữ liệu quan trọng để biểu diễn tập động, trong đó tất cả các thao tác đều được thực hiện với thời gian O(h), trong đó h là chiều cao của cây.*

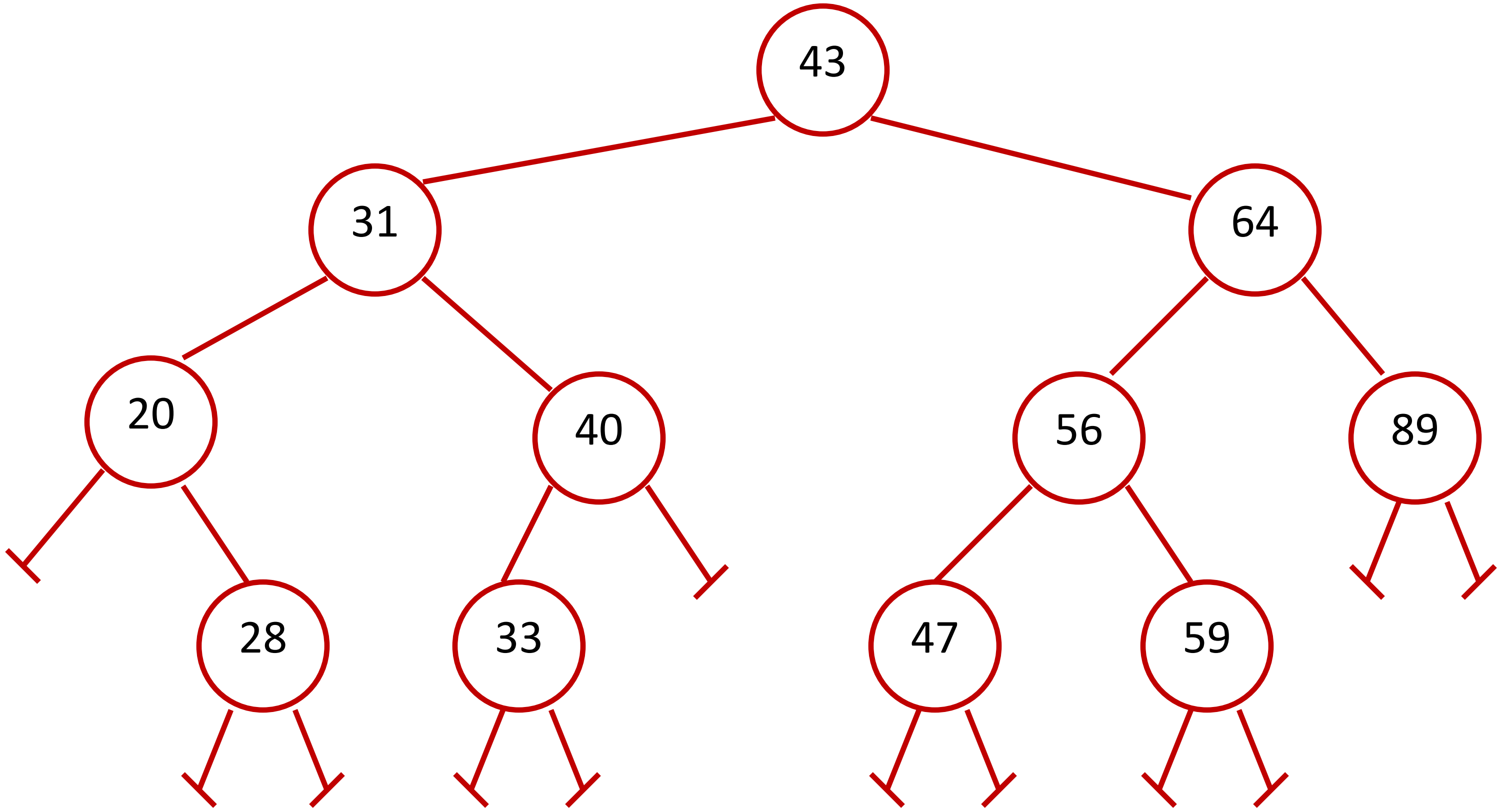
- **Mỗi nút x (ngoài thông tin đi kèm) có các trường:**

left	*key*	*right*	*parent*

- Tính chất BST :
  - 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.)

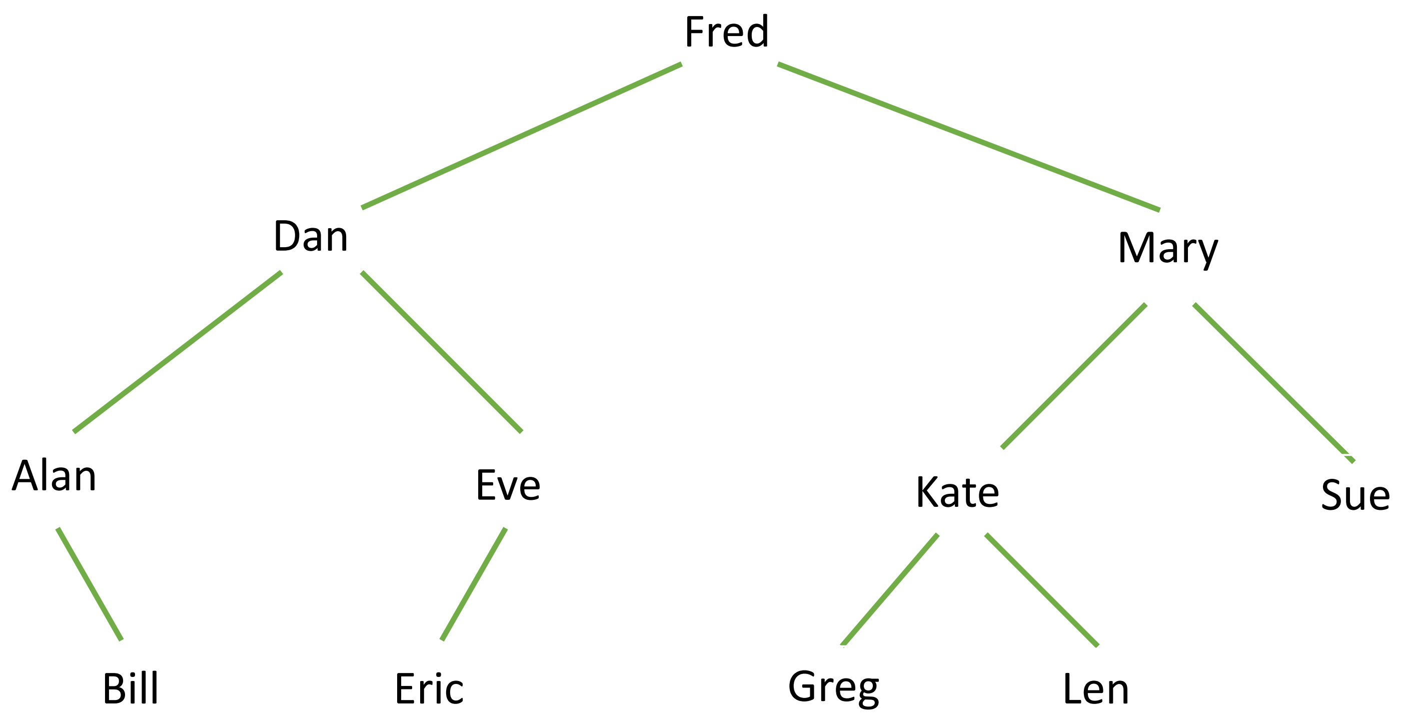


Ví dụ 1:



Khoá là số nguyên

- Ví dụ 2:



Khoá là xâu ký tự

NỘI DUNG TIẾP THEO

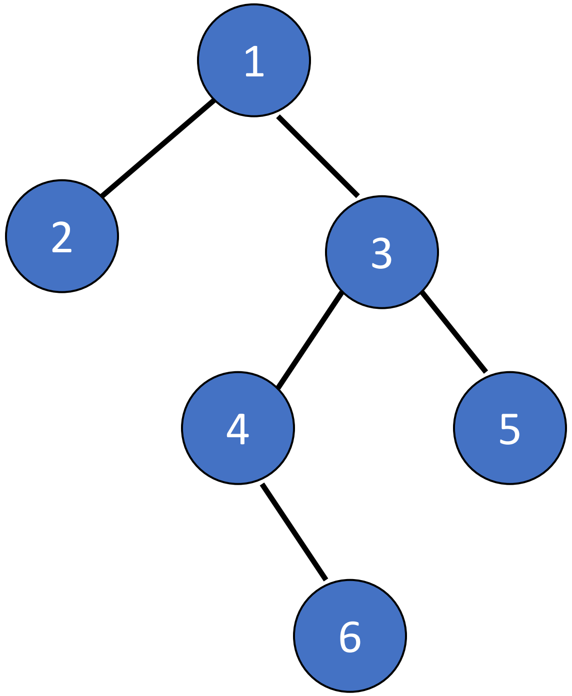
1. Định nghĩa

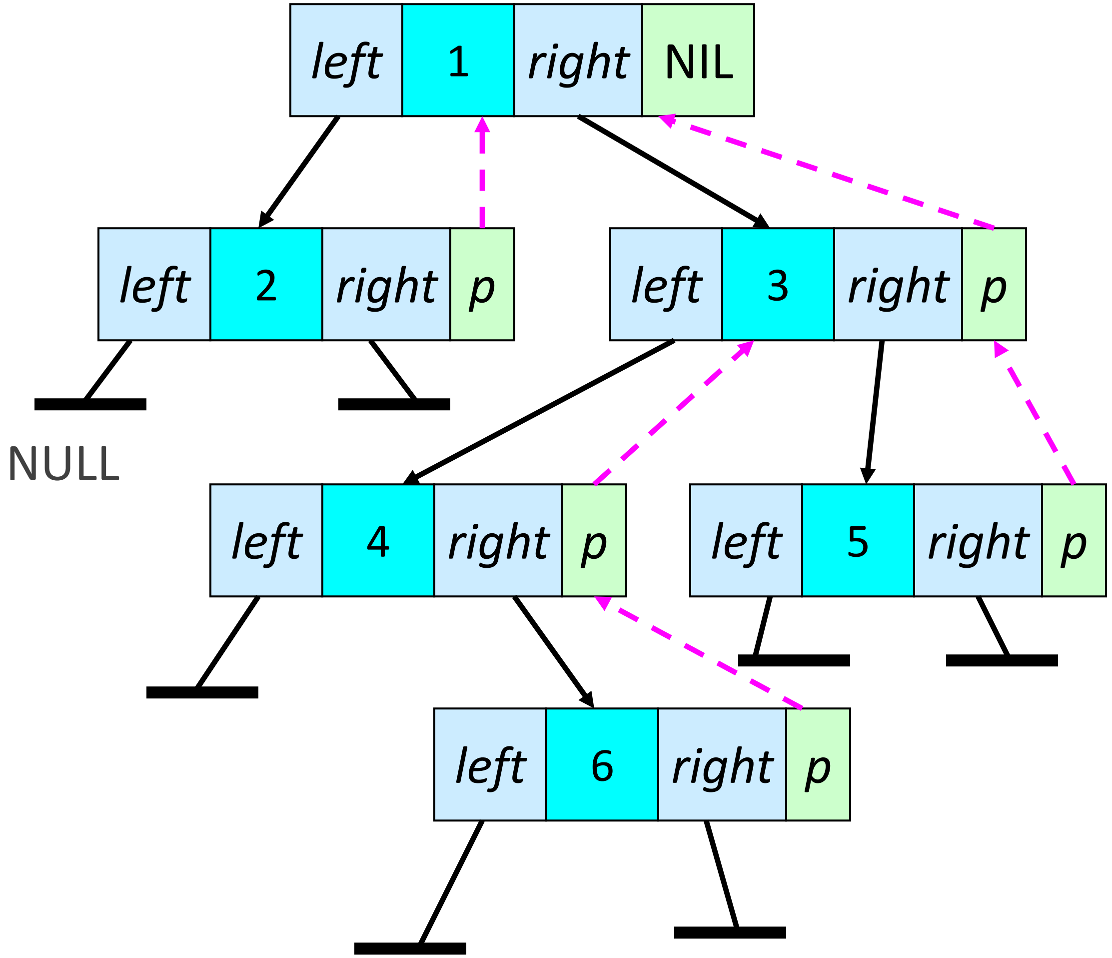
2. **Biểu diễn cây nhị phân tìm kiếm**

3. Các phép toán

2. BIỂU DIỄN CÂY NHỊ PHÂN TÌM KIẾM

- Sử dụng cấu trúc cây nhị phân





Ví dụ 1: Khoá là số nguyên

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

typedef struct TreeNodeRec TreeNode;
```

2. BIỂU DIỄN CÂY NHỊ PHÂN TÌM KIẾM

Ví dụ 2: Khóa là xâu ký tự

```
#define MAXLEN 15

struct TreeNodeRec
{
   char   key[MAXLEN];

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

typedef struct TreeNodeRec TreeNode;
```

NỘI DUNG TIẾP THEO

1. Định nghĩa

2. Biểu diễn cây nhị phân tìm kiếm

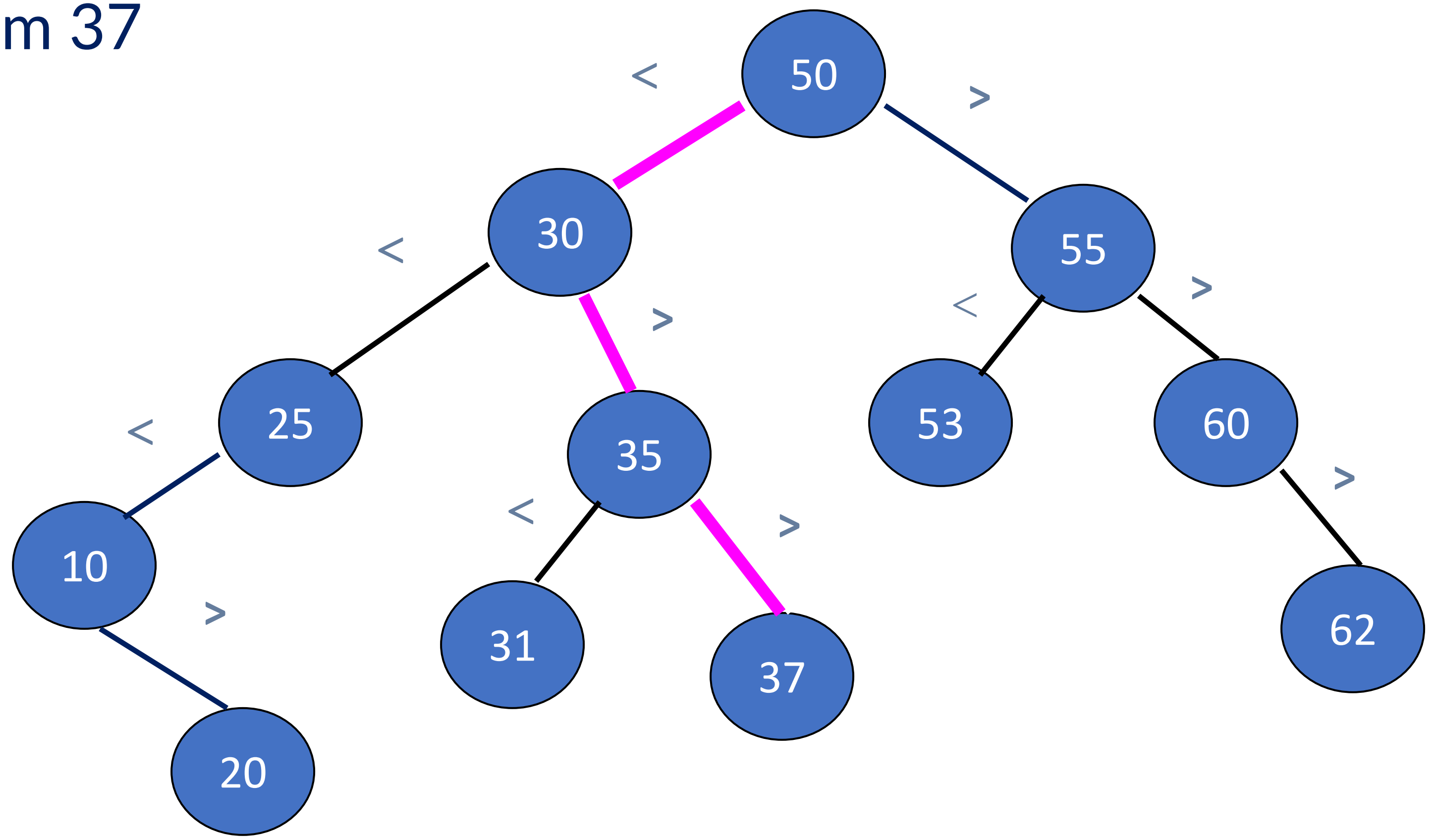
3. Các phép toán
   - 3.1. Tìm kiếm trên BST
   - 3.2. Tìm phần tử nhỏ nhất, lớn nhất
   - 3.3. Kế cận trước và Kế cận sau
   - 3.4. Bổ sung 1 nút vào BST
   - 3.5. Loại bỏ 1 nút trên BST
   - 3.6. Duyệt theo thứ tự giữa
   - 3.7. Độ phức tạp trung bình của thao tác với BST

- 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;
```

3.1. Tìm kiếm trên BST

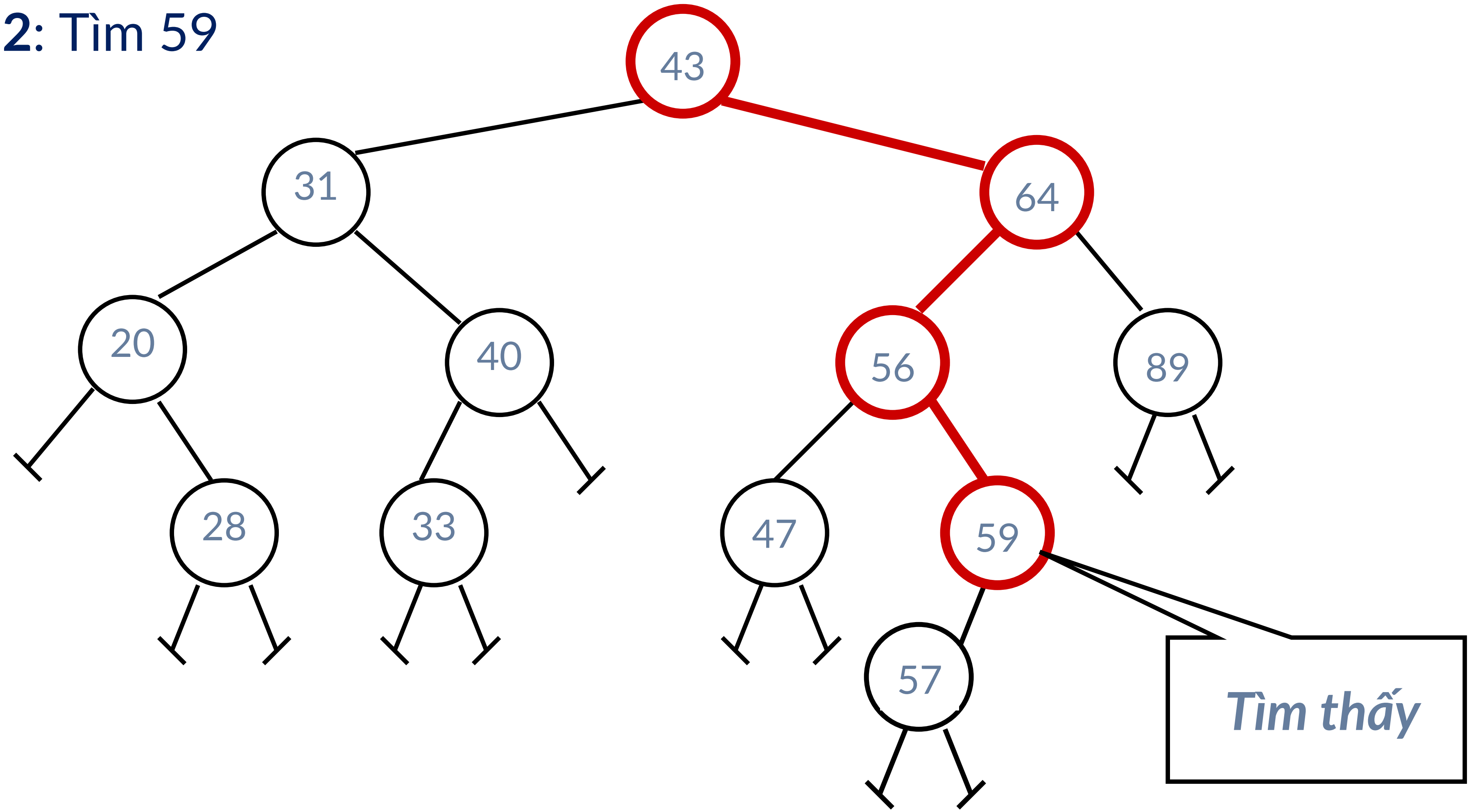
Ví dụ 1: Tìm 37



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

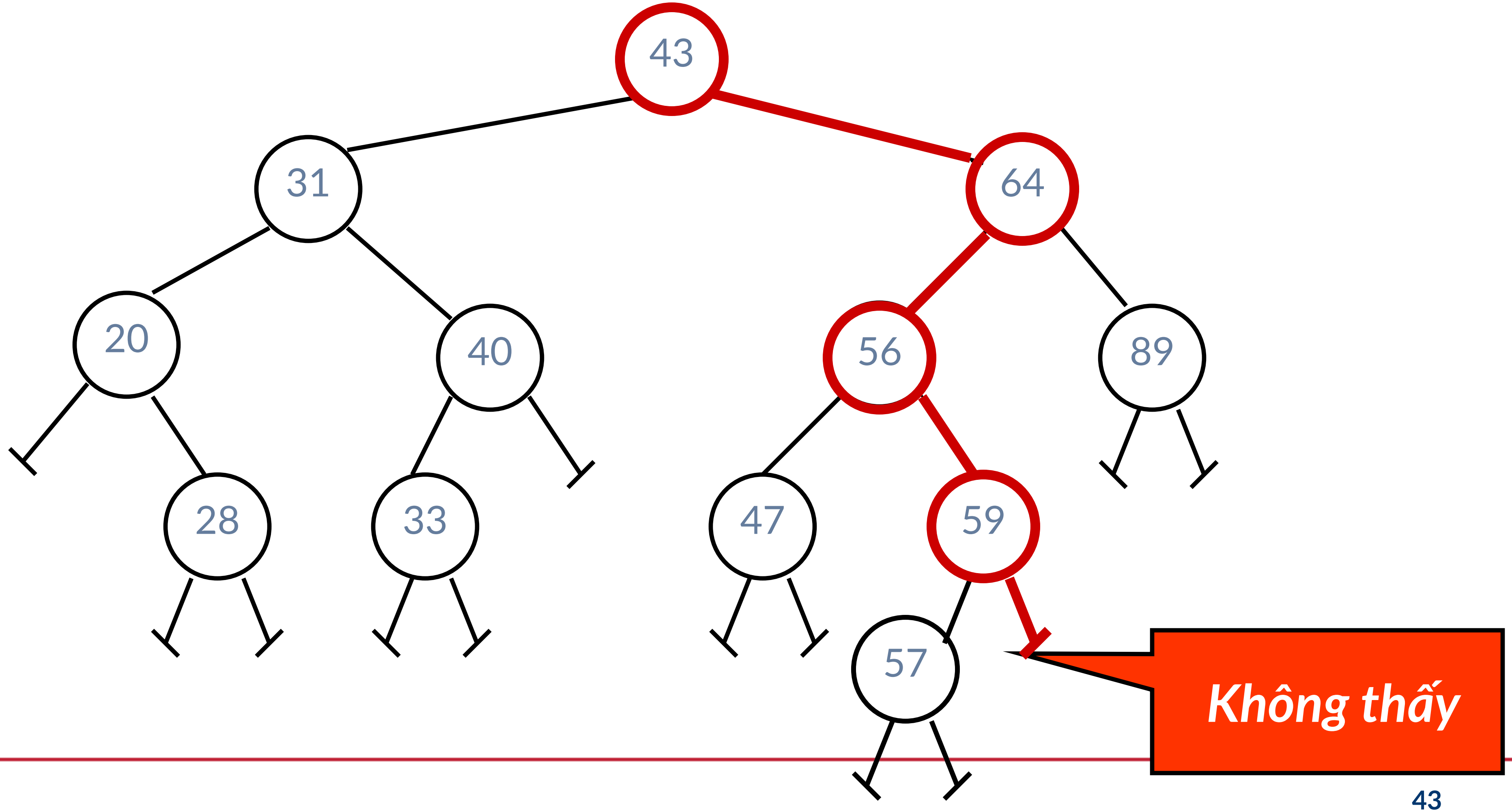
3.1. Tìm kiếm trên BST

Ví dụ 2: Tìm 59



3.1. Tìm kiếm trên BST

Ví dụ 3: Tìm 61



3.1. Tìm kiếm trên BST

```
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;
}
```

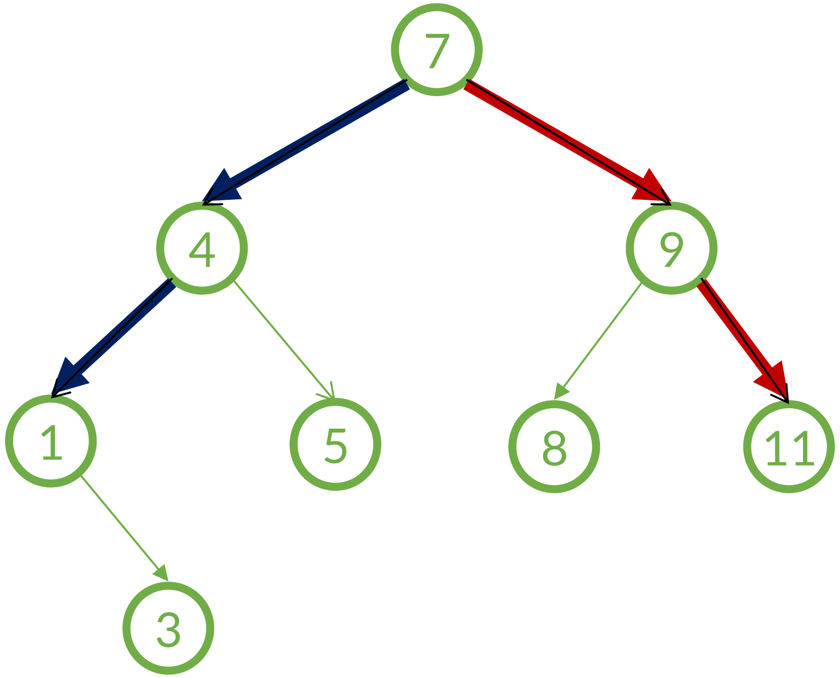
Thời gian tính: $O(h)$
h là độ cao của BST

3.1. Tìm kiếm trên BST

```
/* … Ví dụ đoạn chương trình gọi đến hàm search…  */

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

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

/* … các lệnh khác … */
```

3.2. Tìm phần tử nhỏ nhất, lớn nhất

```
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);
}
```

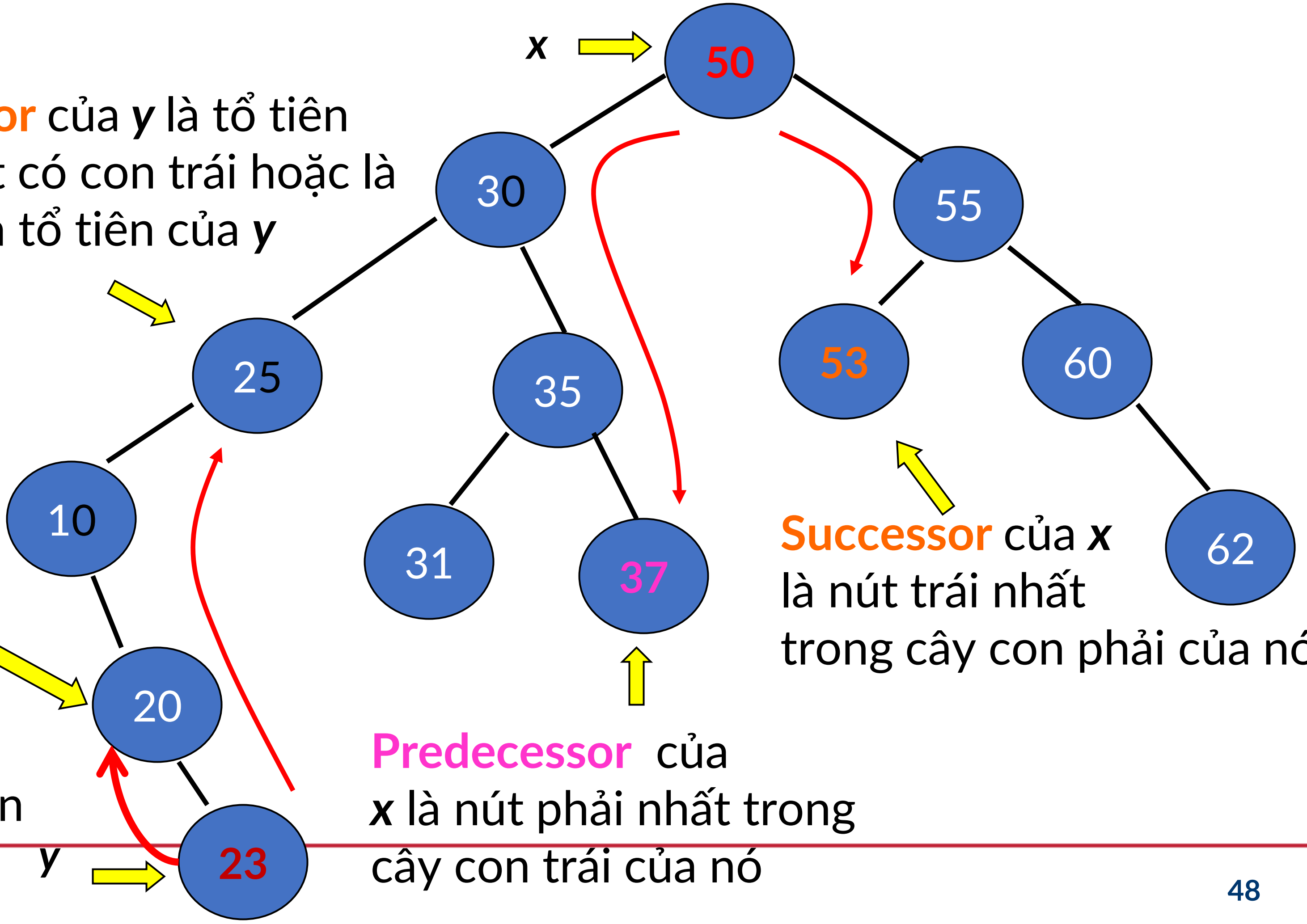


3.3. Kế cận trước và Kế cận sau

- **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.

3.3. Kế cận trước và Kế cận sau

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*

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

y không có cây con cả trái lẫn phải

3.3. Kế cận trước và Kế cận sau

- **Tìm kế cận sau (2 tình huống)**

a. 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*

(*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 →nút *y* cần tìm.

3.3. Kế cận trước và Kế cận sau

- **Tìm kế cận sau (2 tình huống)**

b. 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. 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).

3.3. Kế cận trước và Kế cận sau

- **Tìm kế cận trước (2 tình huống)**

a. 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

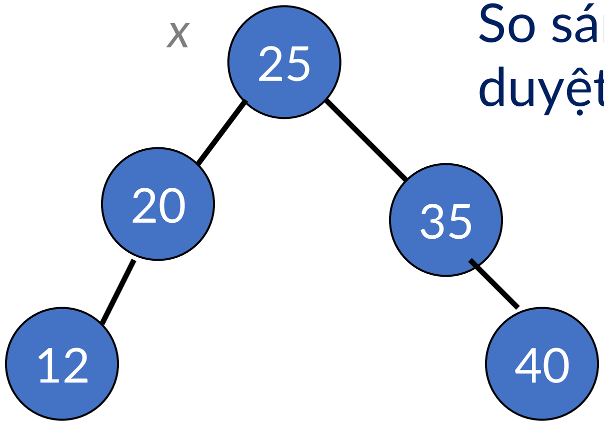
- (y là nút phải nhất nhất trong cây con trái của x):

y = find_max(x->leftPtr)

b. 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.

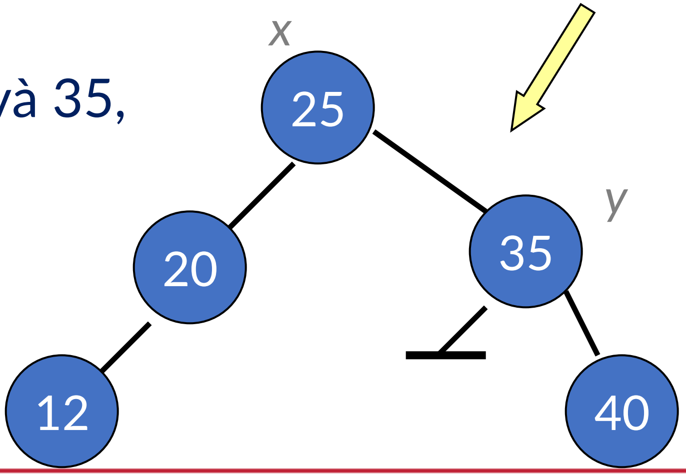
3.4. Bổ sung 1 nút vào BST

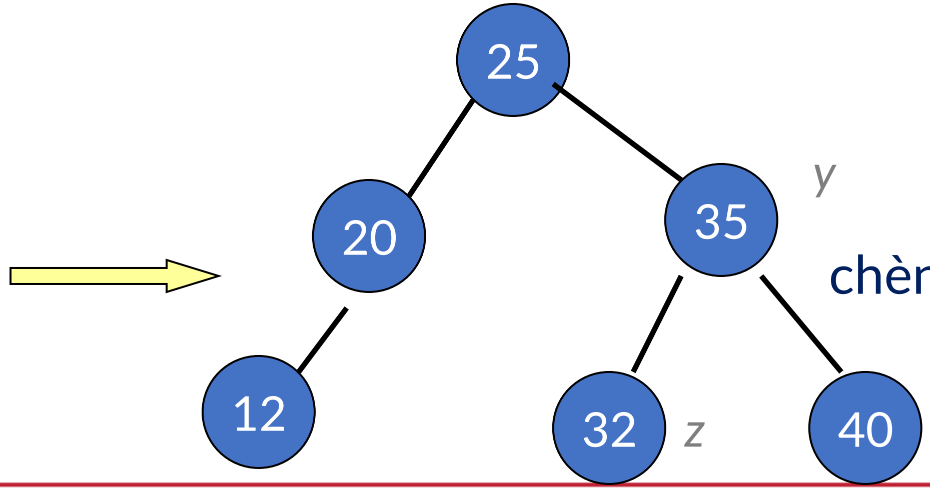
Ví dụ 1: Bổ sung z = 32



So sánh 32 và 25
duyệt tiếp cây con phải

So sánh 32 và 35,
duyệt tiếp
cây con trái

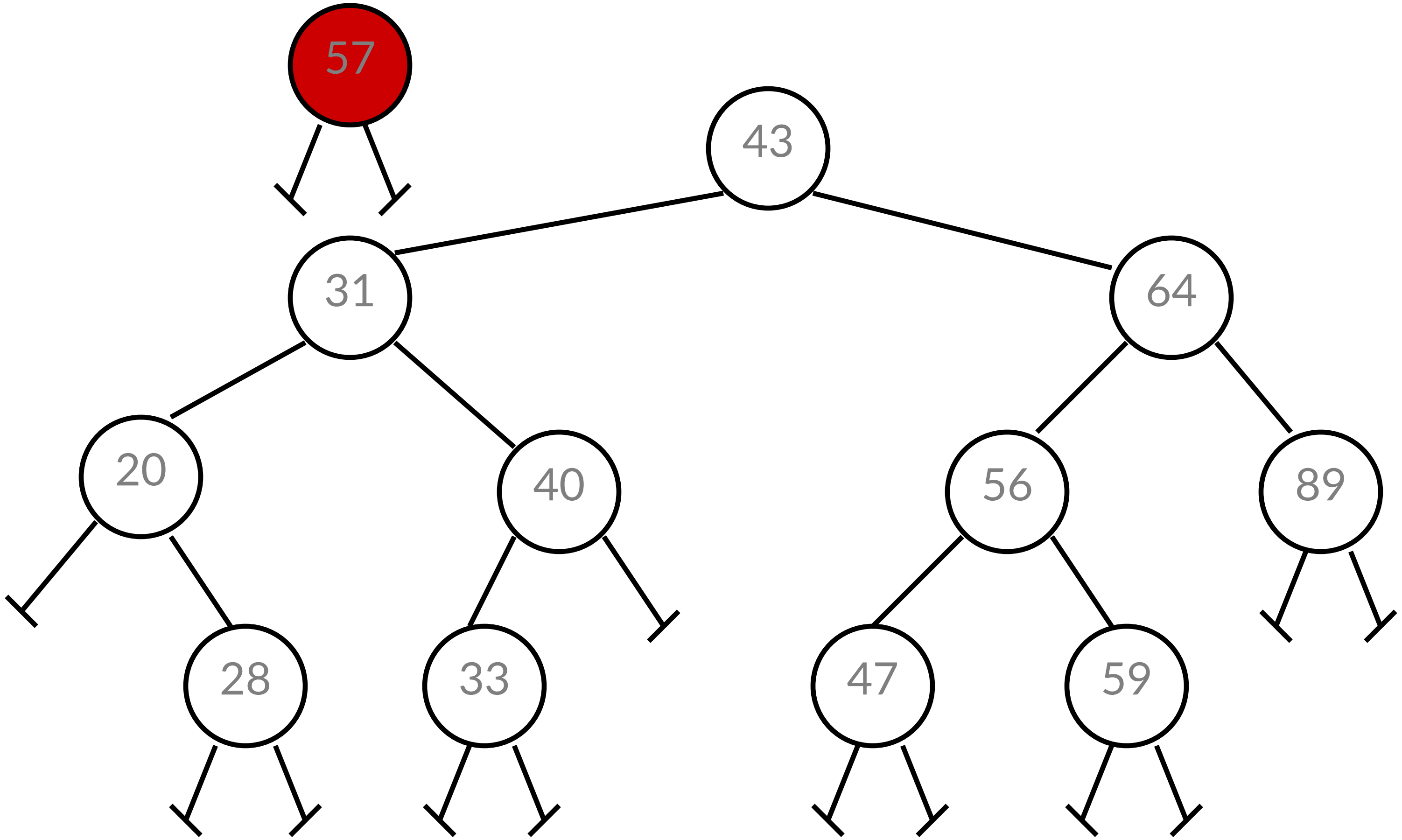




chèn 32 như là con trái

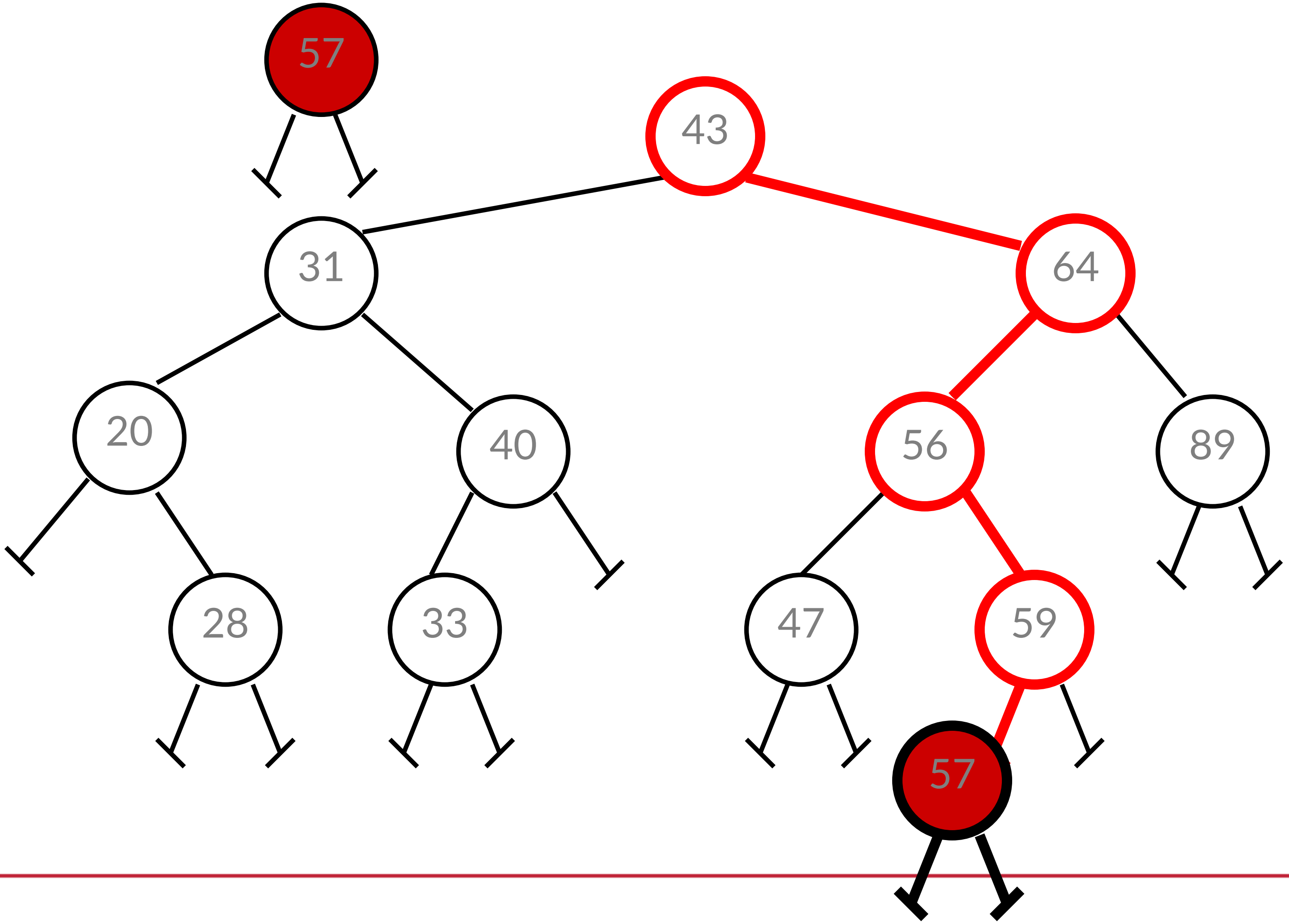
3.4. Bổ sung 1 nút vào BST

Ví dụ 2



3.4. Bổ sung 1 nút vào BST

Ví dụ 2



3.4. Bổ sung 1 nút vào BST

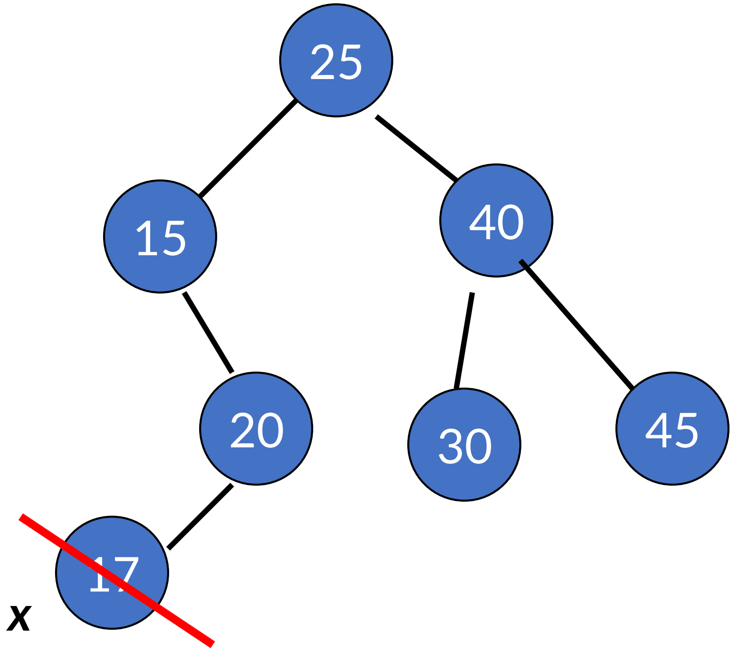
- **Cài đặt**

```
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: $O(h)$,
h là độ cao của BST

3.5. Loại bỏ 1 nút trên BST

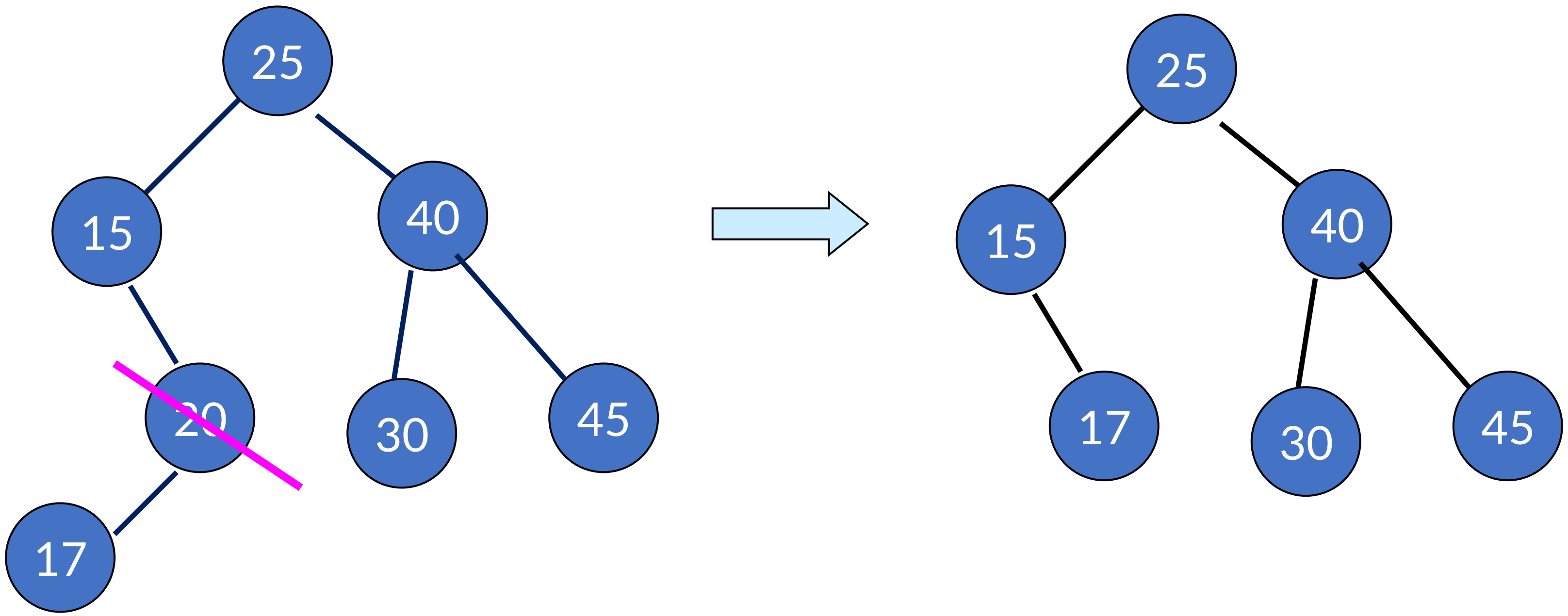
Tình huống 1:	Nút cần xoá ***x*** là lá (leaf
Thao tác:	Chữa lại nút cha của ***x*** có con rỗng.



3.5. Loại bỏ 1 nút trên BST

Tình huống 2: Nút cần xoá *x* có con trái mà không có con phải

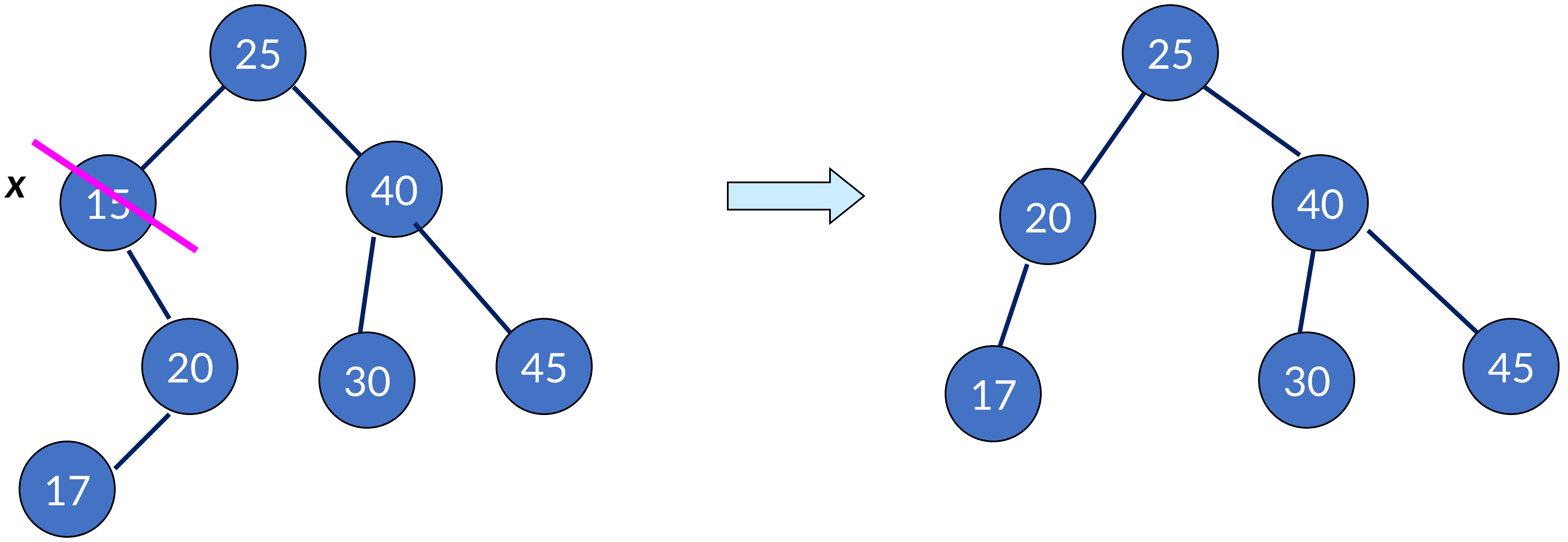
Thao tác: Gắn cây con trái của *x* vào cha



3.5. Loại bỏ 1 nút trên BST

Tình huống 3: nút cần xoá x có con phải mà không có con trái

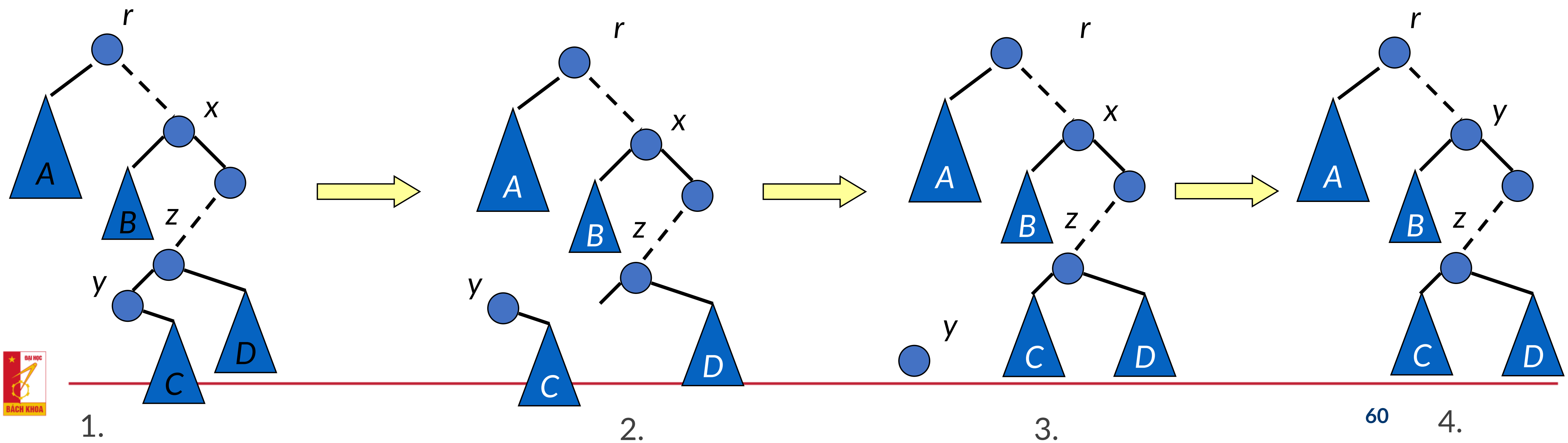
Thao tác: gắn cây con phải của x vào cha



3.5. Loại bỏ 1 nút trên BST

Tình huống 4: nút *x* có 2 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.

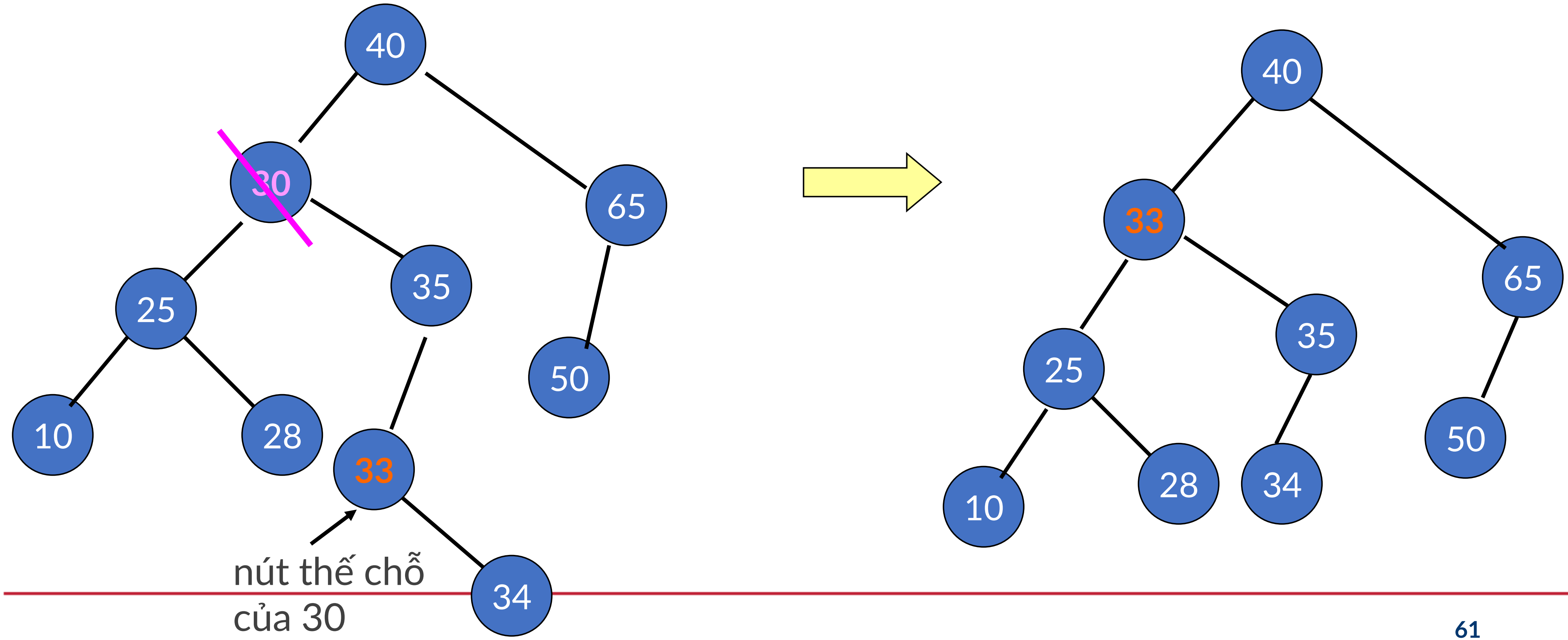


3. CÁC PHÉP TOÁN

3.5. Loại bỏ 1 nút trên BST

Ví dụ tình huống 4:

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



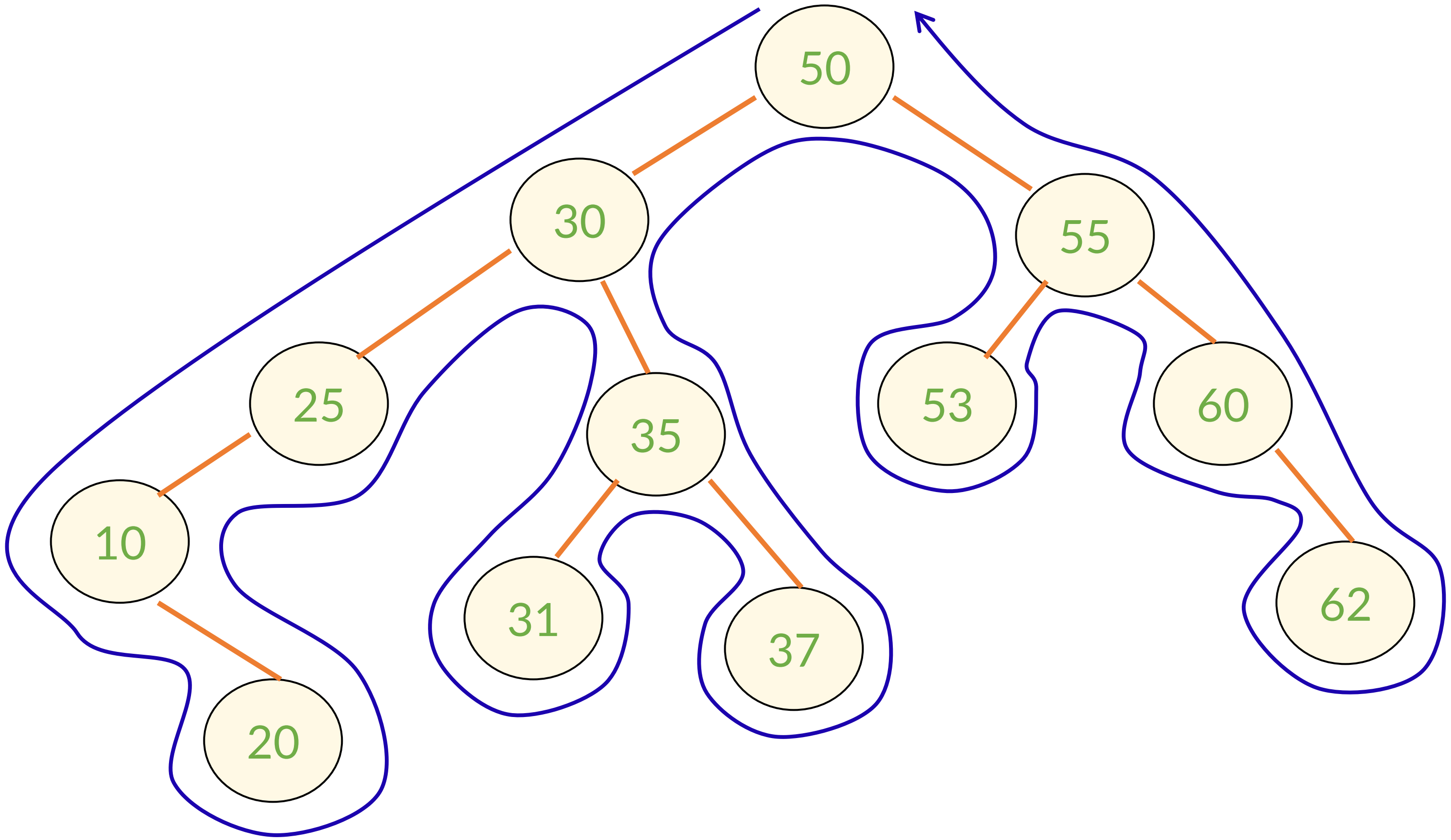
3. CÁC PHÉP TOÁN

3.5. Loại bỏ 1 nút trên BST

Cài đặt xóa 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);
}
```

3.6. Duyệt theo thứ tự giữa



Đưa ra dãy khoá được sắp xếp: 10, 20, 25, 30, 31, 35, 37, 50, 53, 55, 60, 62

3.7. Độ phức tạp trung bình của thao tác với BST

- Độ cao trung bình của BST là:

$$h = O(\log n).$$

→độ phức tạp trung bình của các thao tác với BST :

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

HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !