

Chương 6

TÌM KIẾM

Bài giảng của
PGS.TS. NGUYỄN ĐỨC NGHĨA

Khoa học Máy tính
Đại học Bách khoa Hà nội

NỘI DUNG

6.1. Tìm kiếm tuần tự và tìm kiếm nhị phân

6.2. Cây nhị phân tìm kiếm

6.3. 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

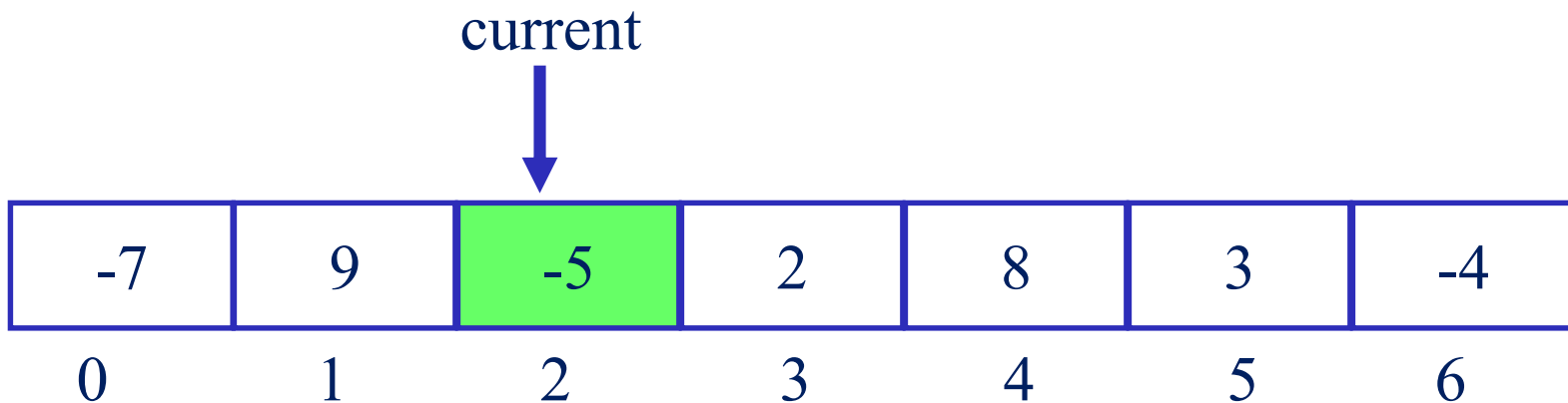
Bài toán tìm kiếm

Cho danh sách a gồm n phần tử a_1, a_2, \dots, a_n và một số x .

Hỏi x có mặt trong danh sách đã cho hay không?

Nếu câu trả lời là khẳng định, hãy đưa ra vị trí xuất hiện của x trong dãy đã cho, nghĩa là 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

- Cần đánh giá thời gian tính tốt nhất, tồi nhất, trung bình của thuật toán với độ dài đầu vào là n . Rõ ràng thời gian tính của thuật toán có thể đánh giá bởi số lần thực hiện phép so sánh

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

trong vòng lặp for.

- Nếu $a[1] = target$ thì phép so sánh (*) phải thực hiện 1 lần. Do đó thời gian tính tốt nhất của thuật toán là $\Theta(1)$.
- Nếu $target$ không có mặt trong dãy đã cho, thì phép so sánh (*) phải thực hiện n lần. Vì thế thời gian tính tồi nhất của thuật toán là $\Theta(n)$.

Phân tích thời gian tính

- Cuối cùng, ta tính thời gian tính trung bình của thuật toán. Nếu *target* tìm thấy ở vị trí thứ *i* của dãy ($target = a[i]$) thì phép so sánh (*) phải thực hiện *i* lần ($i = 1, 2, \dots, n$), còn nếu *target* không có mặt trong dãy đã cho thì phép so sánh (*) phải thực hiện *n* lần. Từ đó suy ra số lần trung bình phải thực hiện phép so sánh (*) là

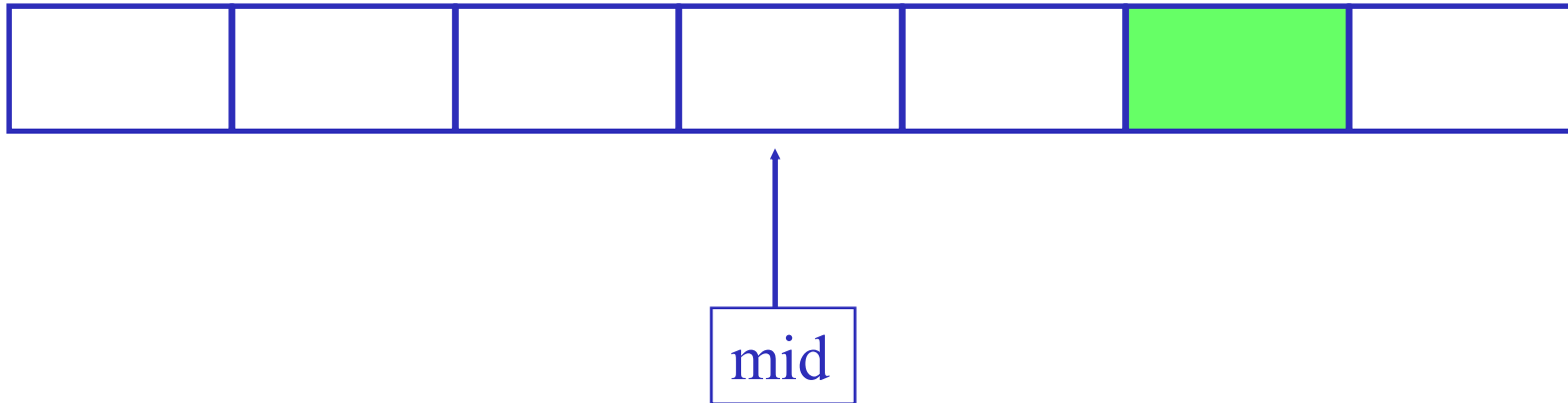
$$\begin{aligned} & [(1 + 2 + \dots + n) + n] / (n+1) \\ &= [n + n(n+1)/2] / (n+1) \\ &= (n^2 + 3n) / [2(n+1)] . \end{aligned}$$

- Ta có:

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

- Vì vậy, thời gian tính trung bình của thuật toán là $\Theta(n)$.

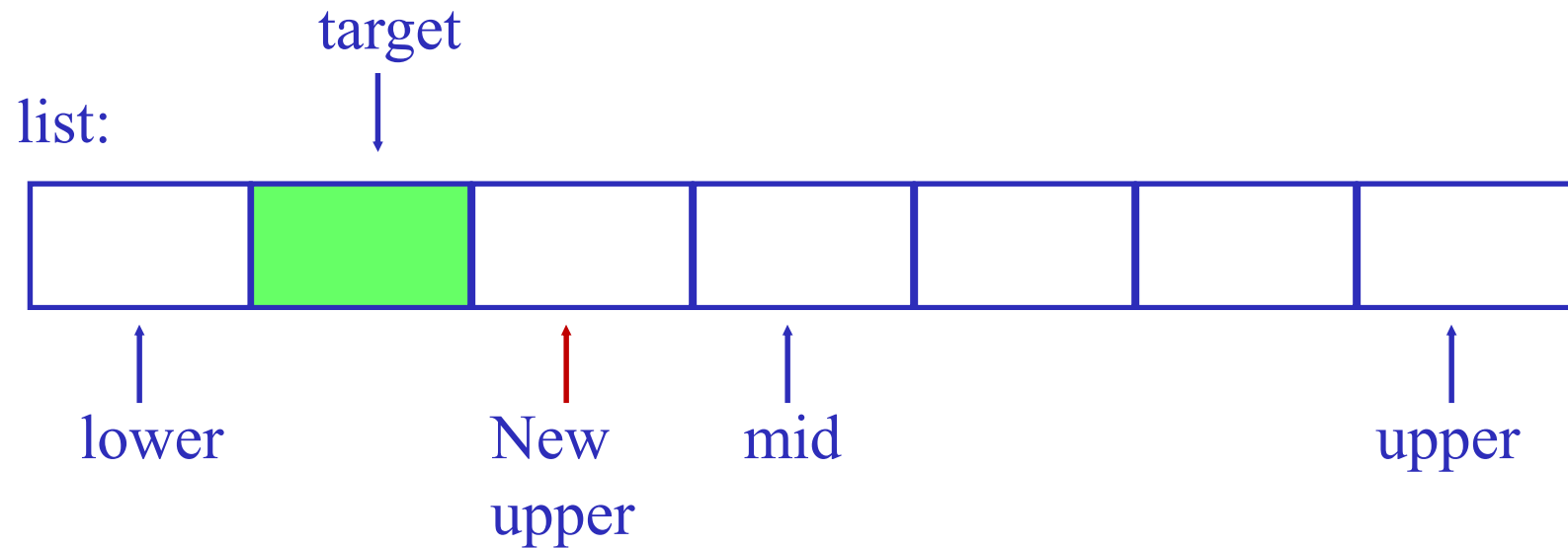
6.1.2. Tìm kiếm nhị phân- Binary Search



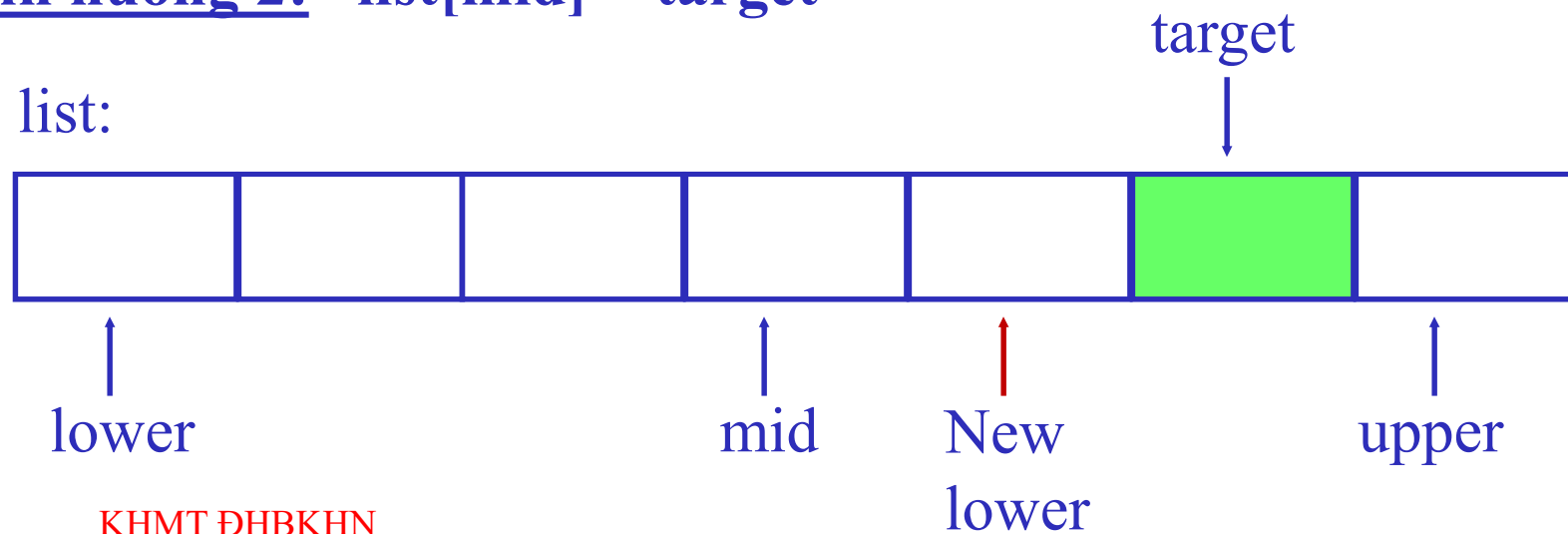
- **Điều kiện:**
 - Danh sách phải được sắp thứ tự.
 - Phải cho phép truy cập trực tiếp.

Độ phức tạp: $O(\log n)$

Tình huống 1: $\text{target} < \text{list}[\text{mid}]$



Tình huống 2: $\text{list}[\text{mid}] < \text{target}$



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. 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 tìm kiếm (sẽ viết tắt là BST) là cây nhị phân có các tính chất sau:

- **Mỗi nút x (ngoài thông tin đi kèm) có các trường:**
 - *left* : con trỏ đến con trái
 - *right*: con trỏ đến con phải,
 - *parent*: con trỏ đến cha (trường này là tùy chọn), và
 - *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).

<i>left</i>	<i>key</i>	<i>right</i>	<i>parent</i>
-------------	------------	--------------	---------------

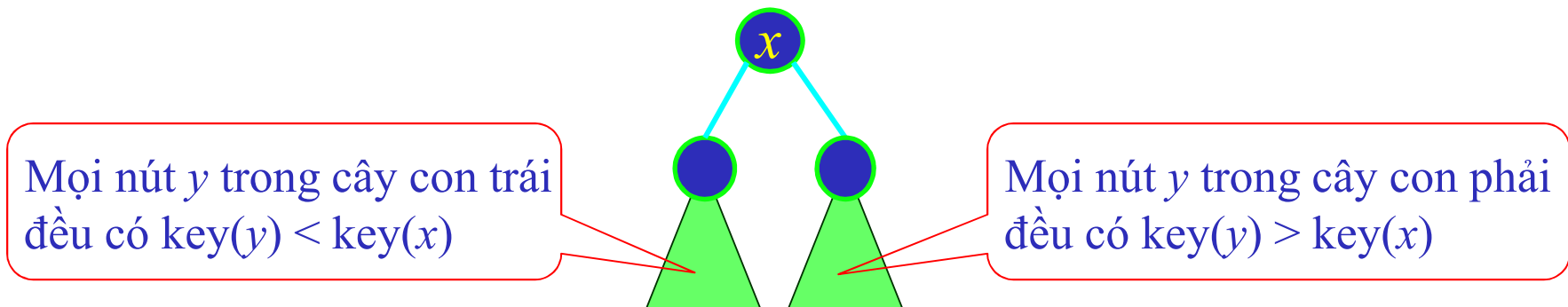
Tính chất BST

– Tính chất BST (Binary-search-tree property):

Giả sử x là gốc của một cây con:

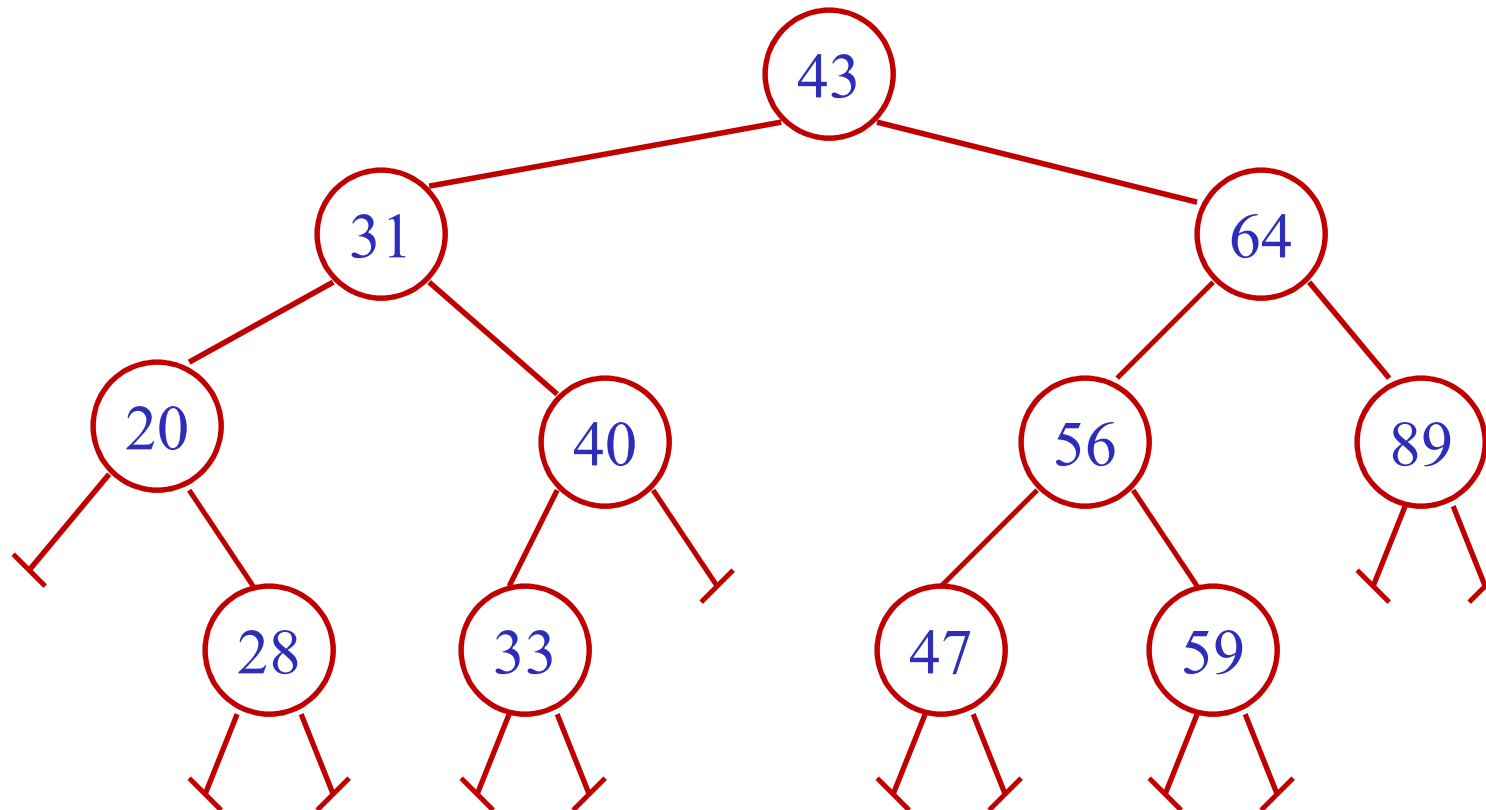
- $\forall y$ thuộc cây con trái của x : $key(y) < key(x)$.
- $\forall y$ thuộc cây con phải của x : $key(y) > key(x)$.

(**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



Các phép toán

- Tìm kiếm (Search)
- Tìm cực tiểu, cực đại (Minimum, Maximum)
- Kế cận sau, Kế cận trước (Predecessor, Successor)
- Chèn (Insert)
- Xoá (Delete)

6.2. Cây nhị phân tìm kiếm

Binary Search Tree

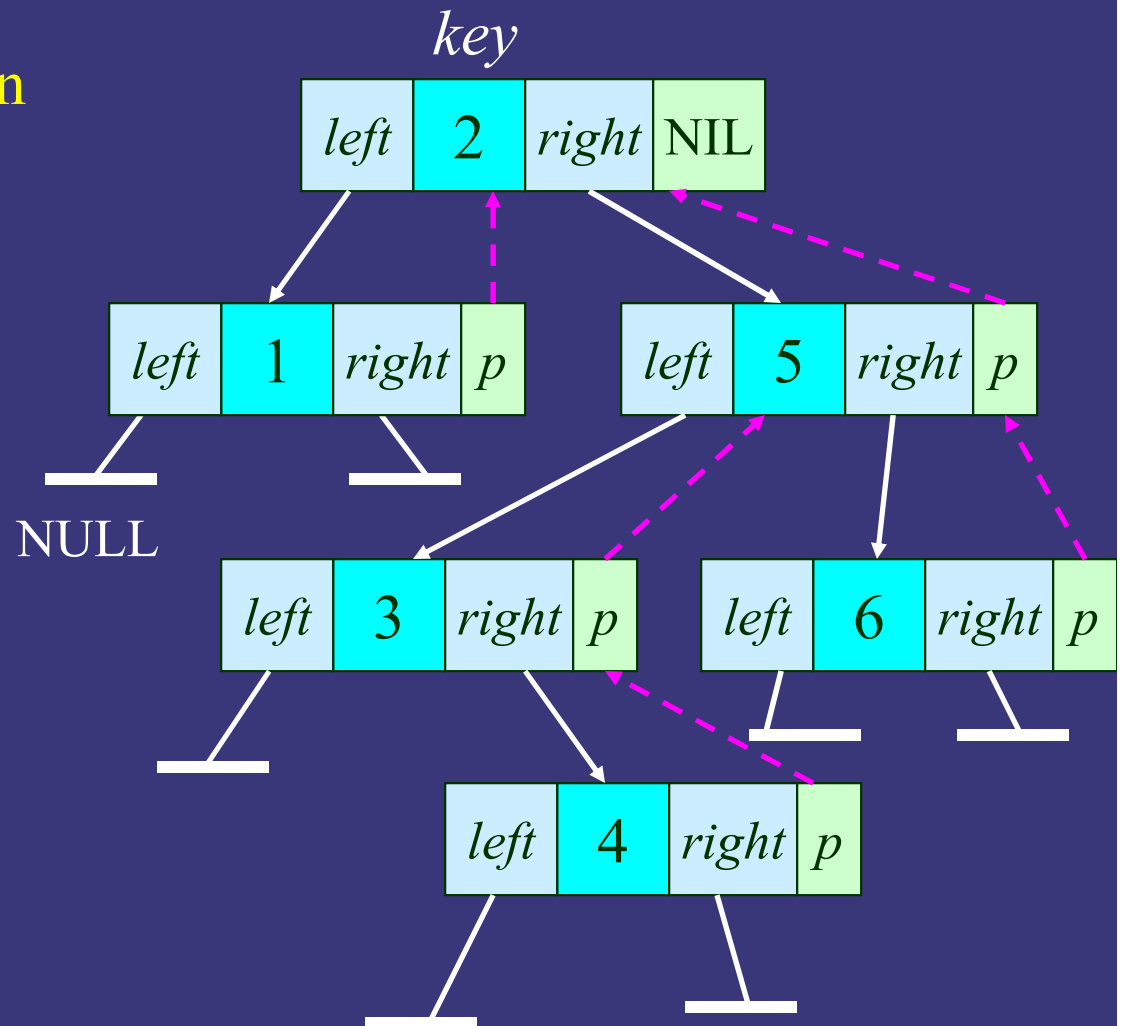
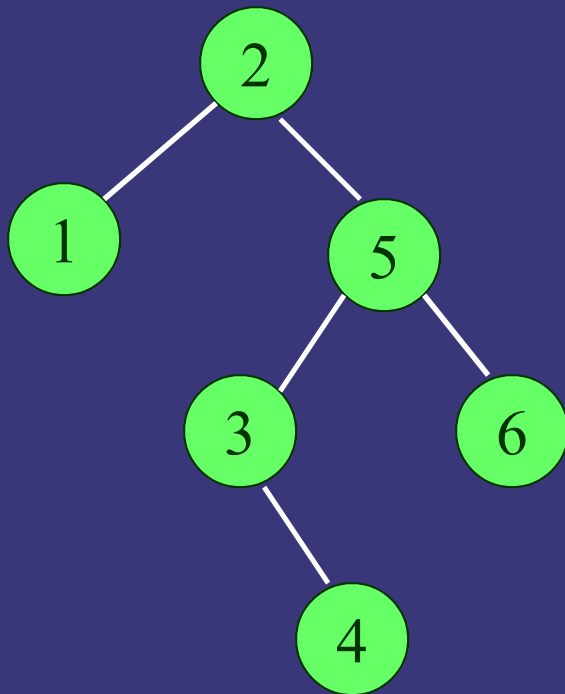
6.2.1. Định nghĩa

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 Tree Structure)



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 chuỗi ký tự

```
#define MAXLEN 15
```

```
struct TreeNodeRec
```

```
{
```

```
    char    key[MAXLEN] ;
```

```
    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)** - Tạo một nút với khoá cho bởi value
- **search(nodePtr, k)** - Tìm kiếm nút có giá trị khoá bằng k trên BST trả bởi nodePtr
- **find_min(nodePtr)**: Trả lại nút có khoá nhỏ nhất trên BST trả bởi nodePtr
- **find_max**: Trả lại nút có khoá lớn nhất trên BST trả bởi nodePtr
- **Successor(x)**: Trả lại nút kế cận sau của nút x
- **Predcessor(x)**: Trả lại nút kế cận trước của nút x
- **insert(nodePtr, float item)** - Chèn một nút với khoá cho bởi item vào BST trả bởi nodePtr
- **delete(nodePtr, item)** - Xóa nút có giá trị khoá bằng item trên BST trả bởi nodePtr
- Ngoài ra có thể sử dụng các hàm cơ bản đối với cây nhị phân như:
 - 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);
```

Tạo một nút mới (MakeNode)

- **Đầu vào:** phần tử cần chèn
- **Các bước:**
 - cấp phát bộ nhớ cho nút mới
 - kiểm tra lỗi cấp phát
 - nếu cấp phát được thì đưa phần tử vào nút mới
 - đặt con trái và phải là NULL
- **Đầu ra:** con trỏ tới (địa chỉ của) nút mới

makeTreeNode

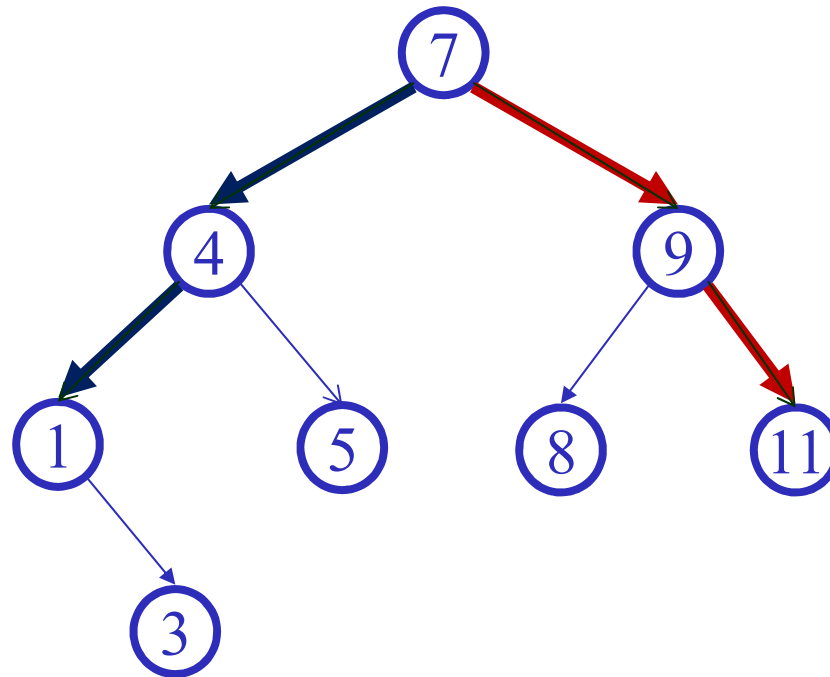
```
TreeNode* makeTreeNode(float value) {
    TreeNode* newNodePtr = NULL;

    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;
}
```

Tìm phần tử nhỏ nhất, lớn nhất: findMin, findMax

- Để tìm phần tử nhỏ nhất trên BST, ta đi theo con trái cho đến khi gặp NULL.
- Để tìm phần tử lớn nhất trên BST, ta đi theo con phải cho đến khi gặp NULL.



Tìm phần tử nhỏ nhất, lớn nhất:
findMin, findMax

find-min(T)

1. **while** *left*[T] \neq *NIL*
2. **do** T \leftarrow *left*[T]
3. **return** T

find-max(T)

1. **while** *right*[T] \neq *NIL*
2. **do** T \leftarrow *right*[T]
3. **return** T

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);
```

```
}
```

Kế cận trước và Kế cận sau

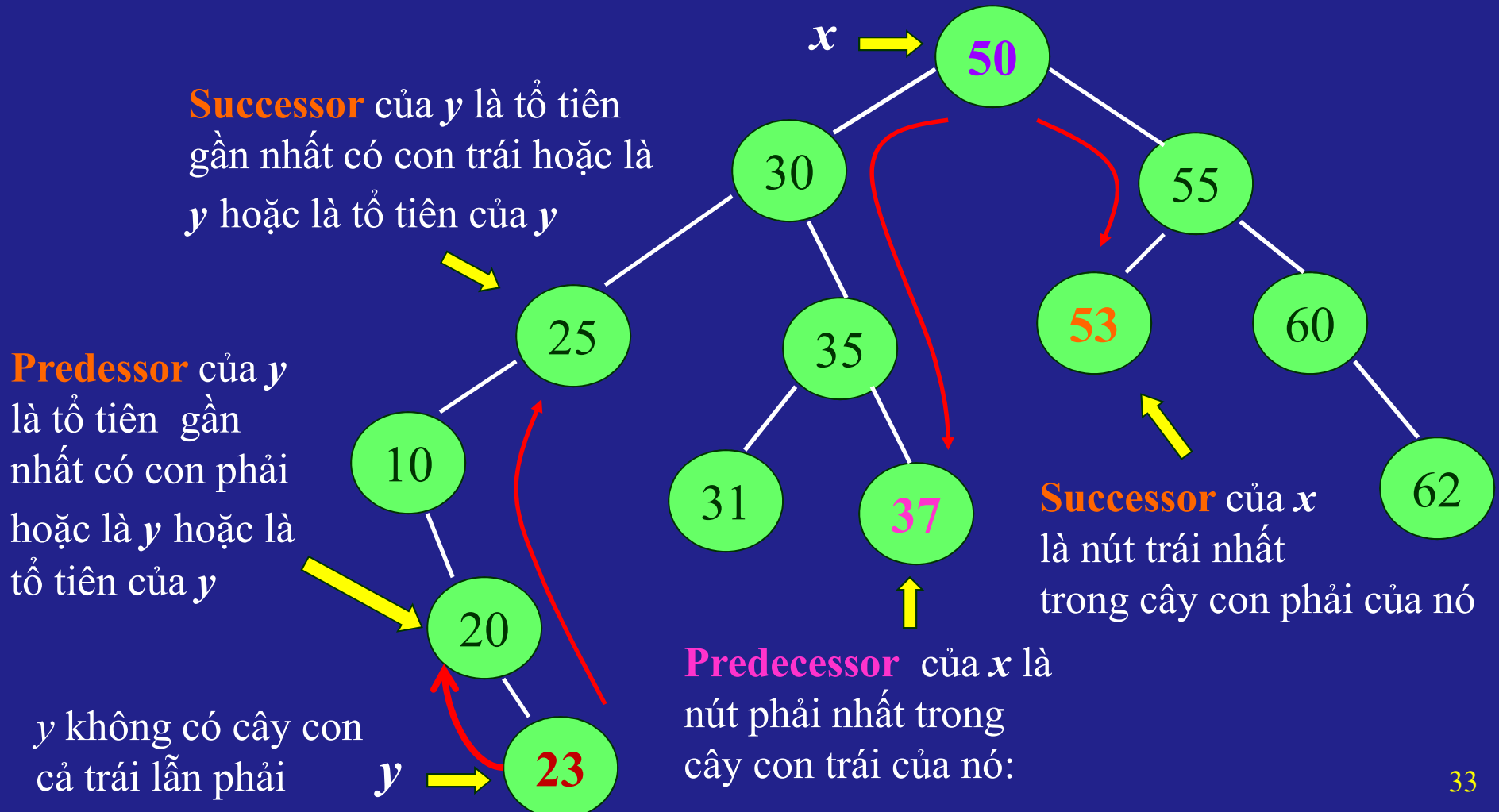
Predecessor and Successor

- 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 NIL.
- Kế cận trước (Predecessor) 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 NIL.
- 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



Tìm kế cận sau

- Có hai 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 $\text{find-min}(x \rightarrow \text{rightPtr})$: **$y = \text{find-min}(x \rightarrow \text{rightPtr})$**
 - 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.
- 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 hai tình huống
- Nếu x có con trái thì kế cận trước của x sẽ là nút y với khoá $\text{key}[y]$ lớn nhất trong cây con trái của x (nói cách khác y là nút phải nhất nhất trong cây con trái của x):

$y = \text{find_max}(x \rightarrow \text{leftPtr})$

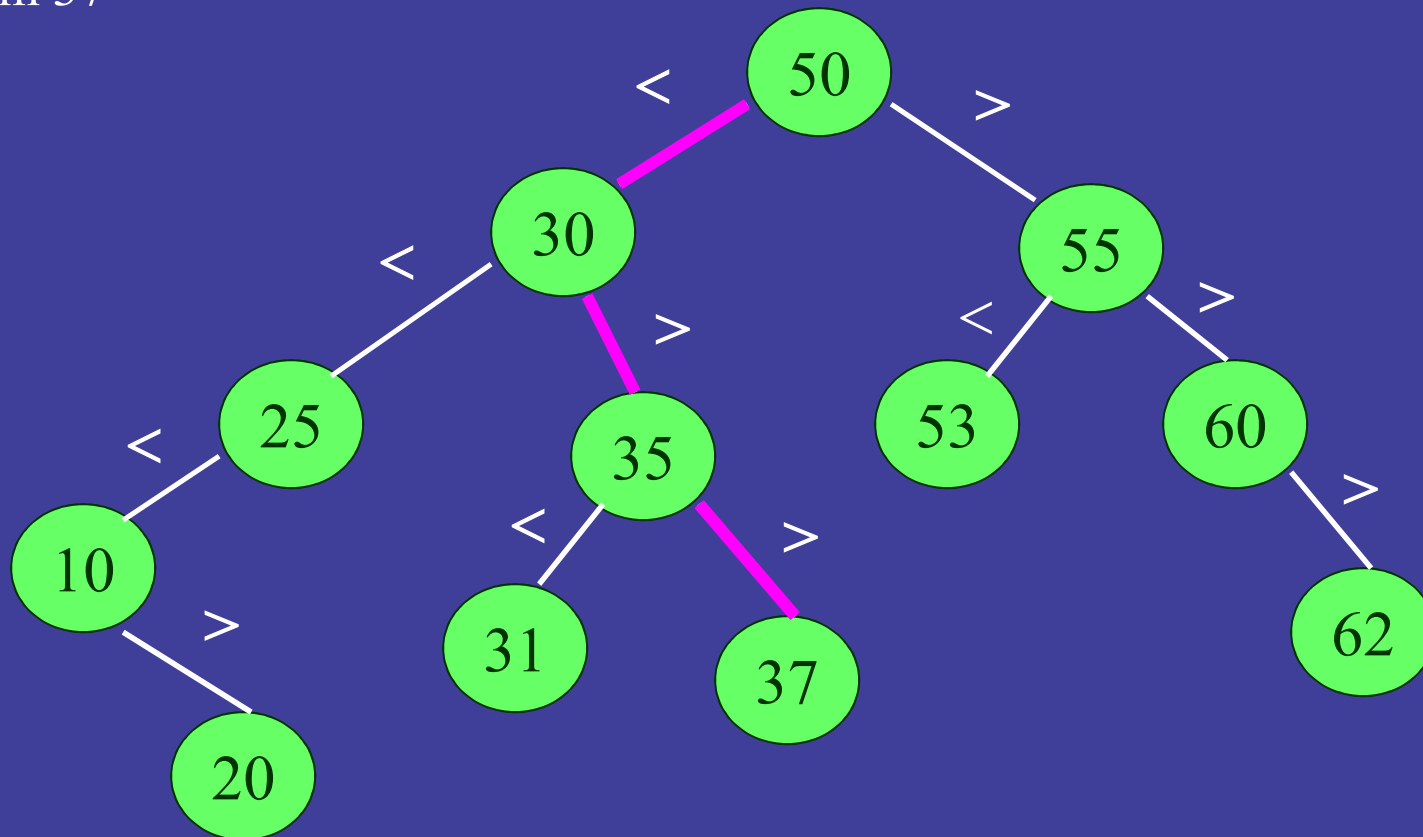
- Nếu x không có con trái thì kế cận trước của x là **tổ tiên gần nhất có con phải hoặc là x hoặc là tổ tiên của x .**

Thuật toán tìm kiếm trên BST

- Nếu khoá cần tìm **nhỏ hơn** khoá của nút hiện tại thì tiếp tục tìm kiếm ở **cây con trái**.
- Trái lại, nếu khoá cần tìm là **lớn hơn** khoá của nút hiện tại, thì tiếp tục tìm kiếm ở **cây con phải**.
- Kết quả cần đưa ra:
 - nếu tìm thấy (nghĩa là khoá cần tìm là bằng khoá của nút hiện tại), thì trả lại con trỏ đến nút chứa khoá cần tìm.
 - ngược lại, trả lại con trỏ NULL .

Ví dụ: Tìm kiếm trên BST

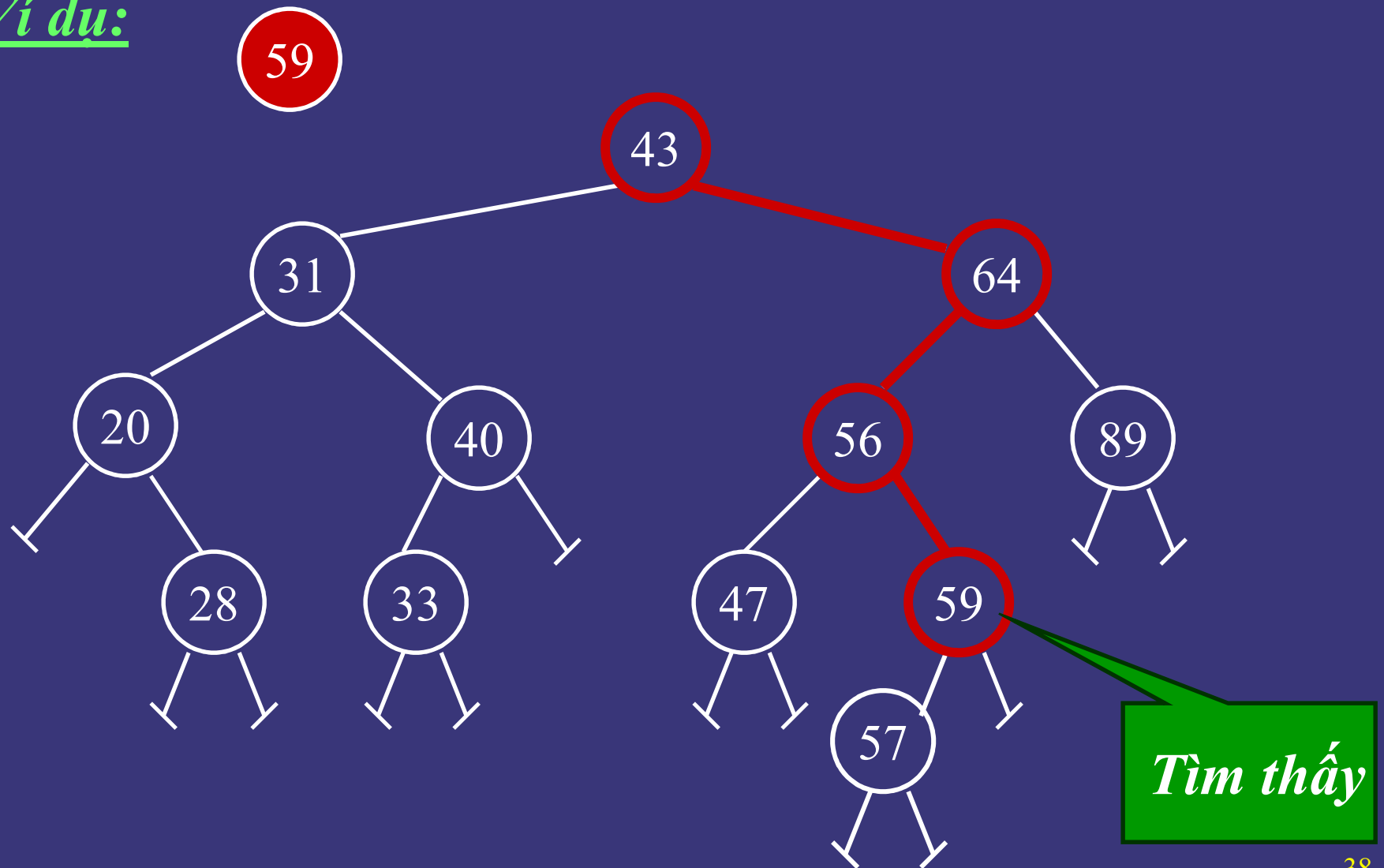
Tìm 37



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

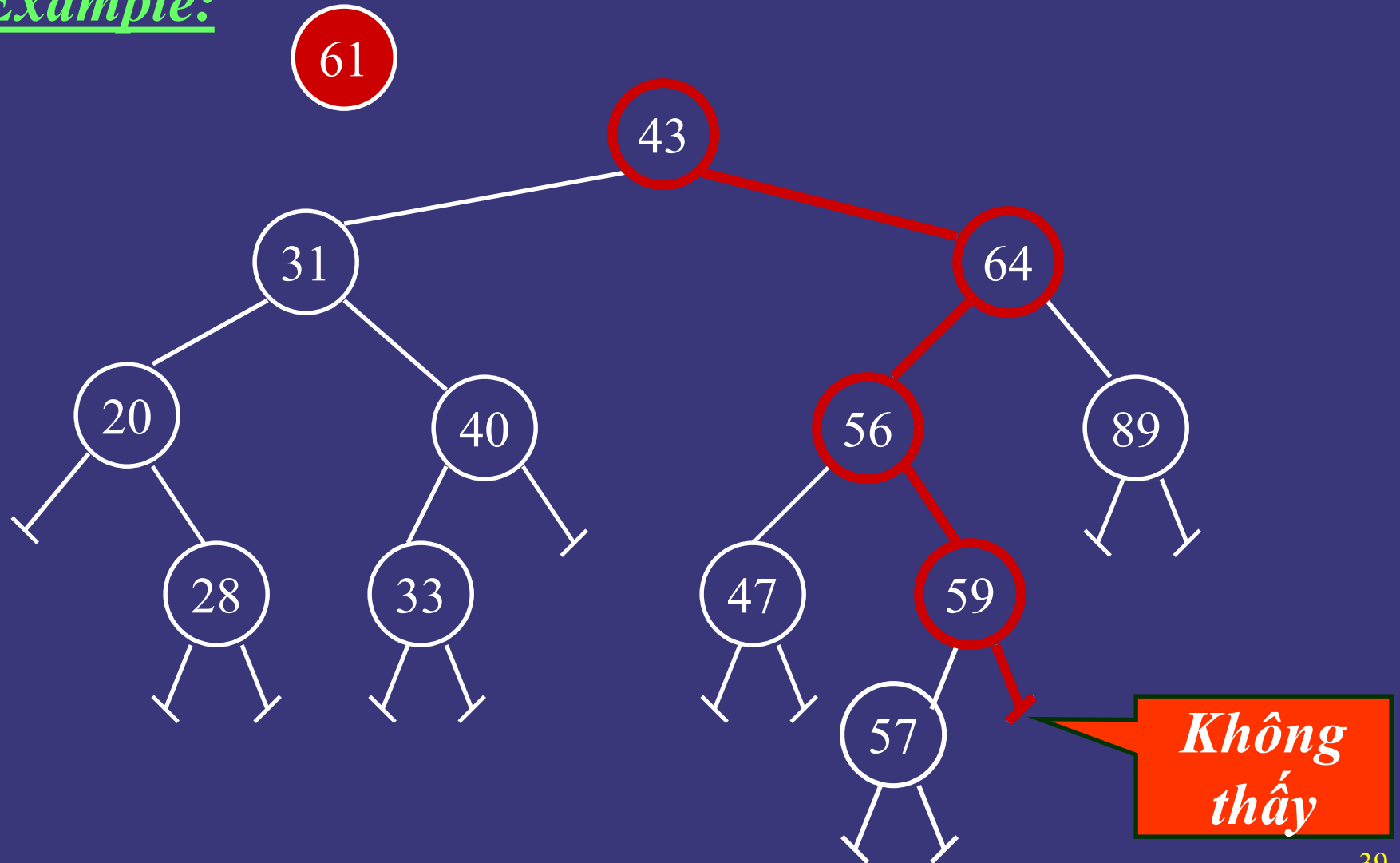
Tìm kiếm (Search)

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 ... */

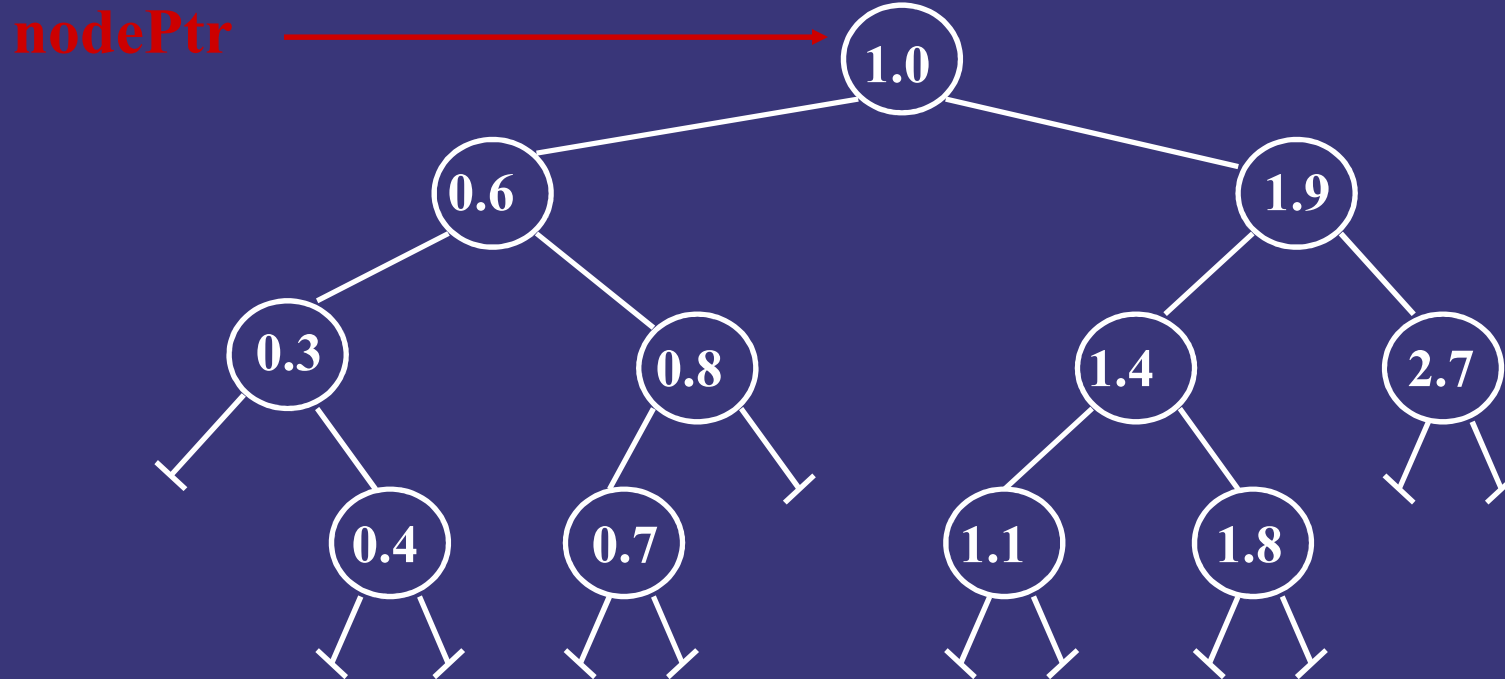
printf("Enter target ");
scanf("%f", &item);

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

/* ... các lệnh khác ... */
```

Ví dụ: Search

Find 0.7



```
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;  
}
```

Search

Find 0.7

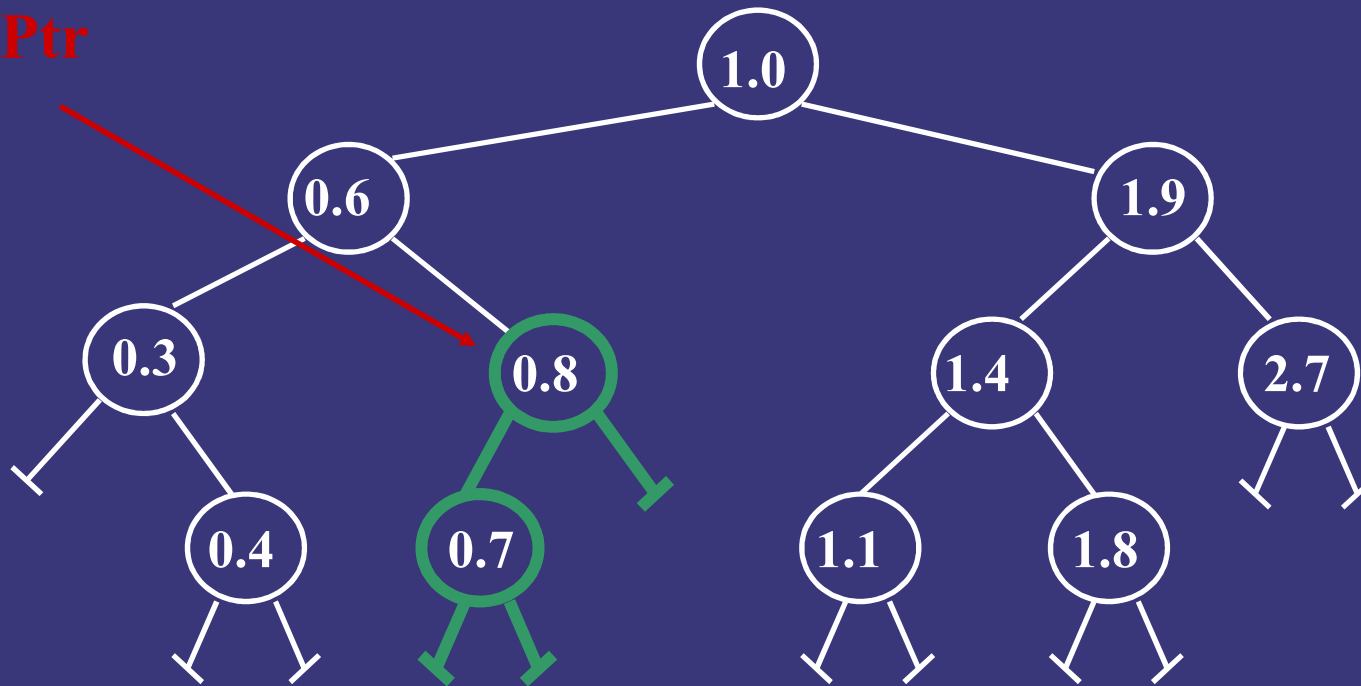


```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
    }
    return nodePtr;
}
```

Search

Find 0.7

nodePtr



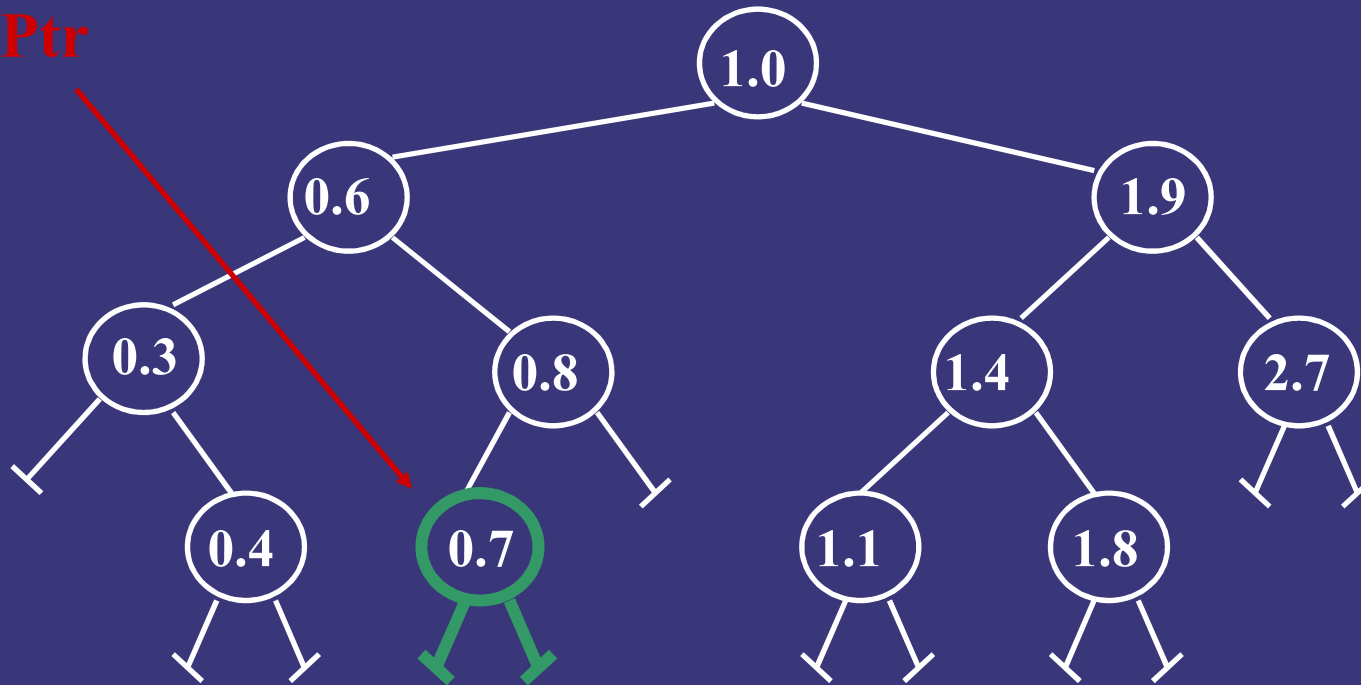
```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
        }
    return nodePtr;
}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

Search

Find 0.7



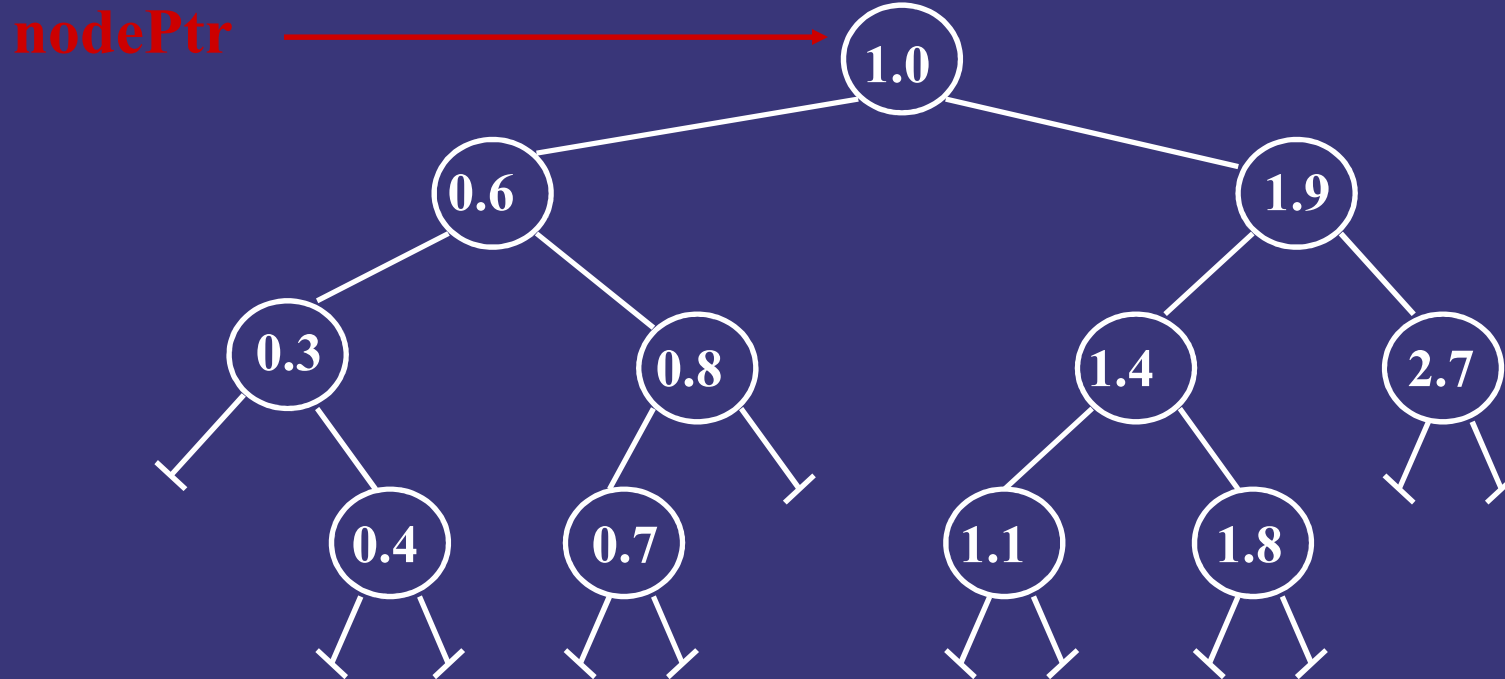
nodePtr



```
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;  
}
```

Search

Find 0.5



```
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;  
}
```

Search

Find 0.5

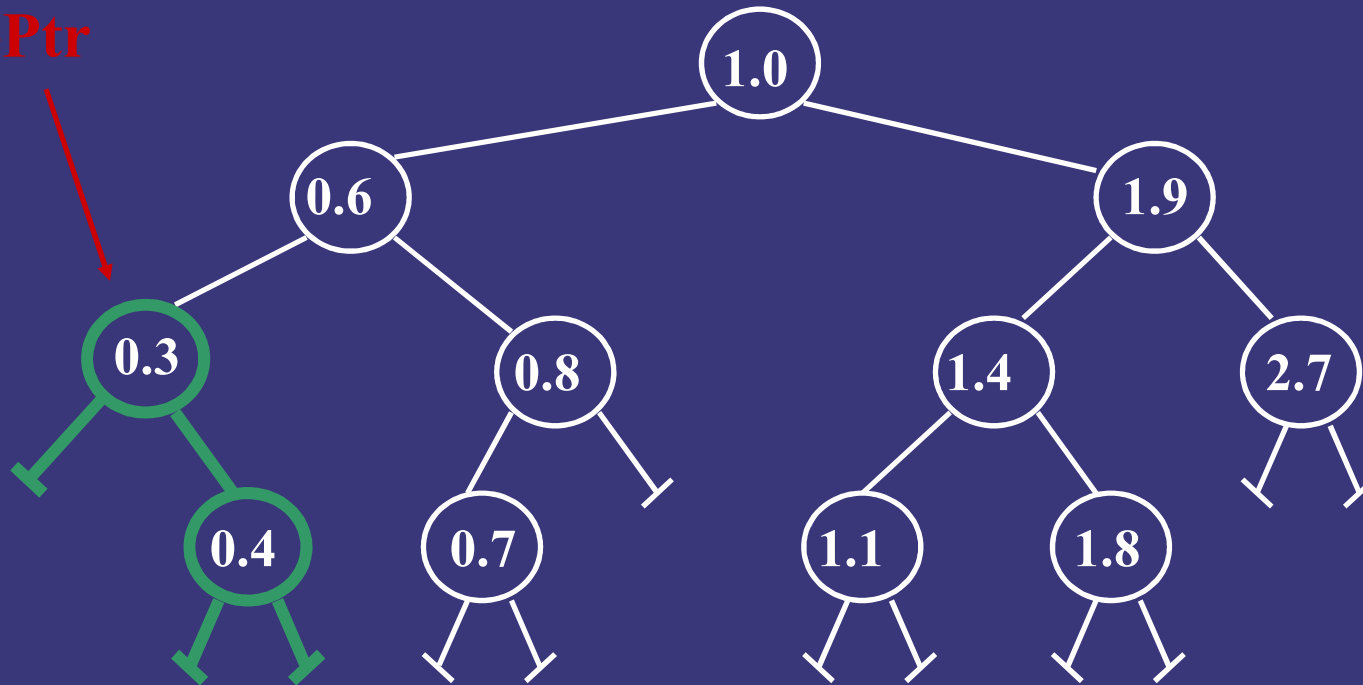


```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
    }
    return nodePtr;
}
```

Search

Find 0.5

nodePtr



```
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;
}
```


Search

Find 0.5



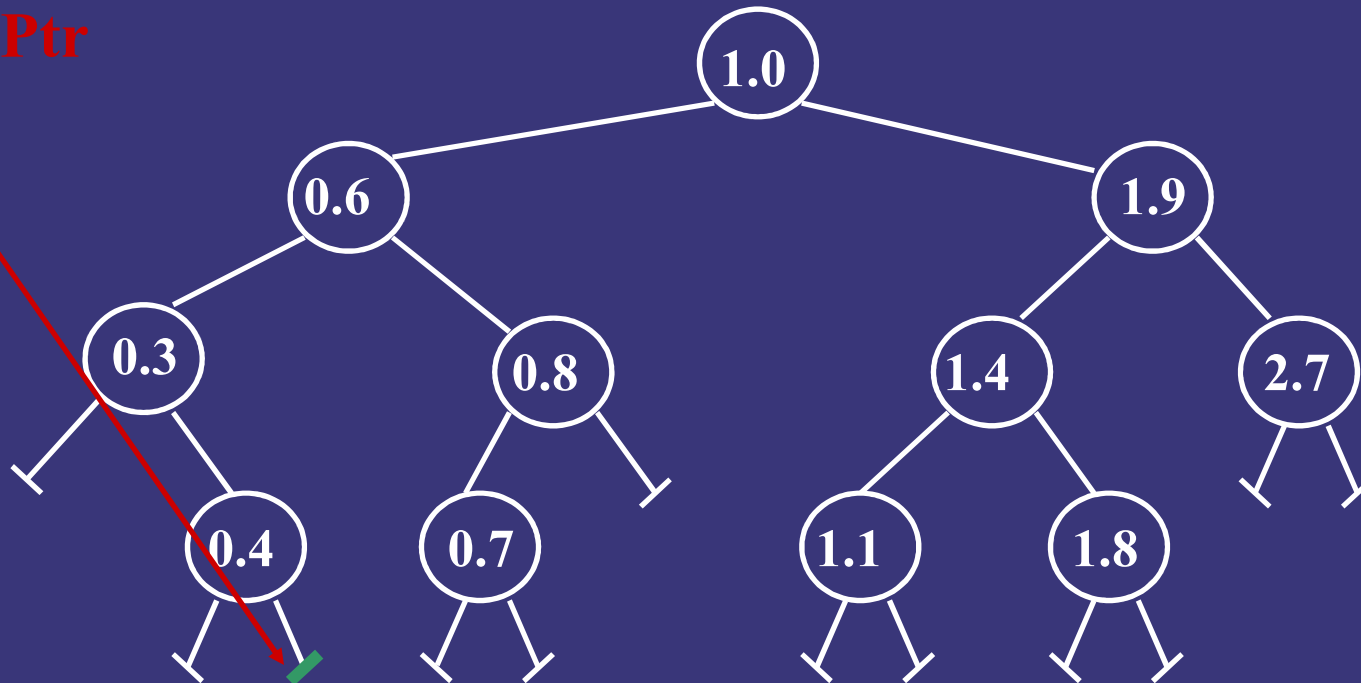
```
TreeNode* search(TreeNode* nodePtr, float target){
    if (nodePtr != NULL){
        if (target < nodePtr->key)
            nodePtr = search(nodePtr->leftPtr, target);
        else if (target > nodePtr->key)
            nodePtr = search(nodePtr->rightPtr, target);
    }
    return nodePtr;
}
```

Search

Find 0.5



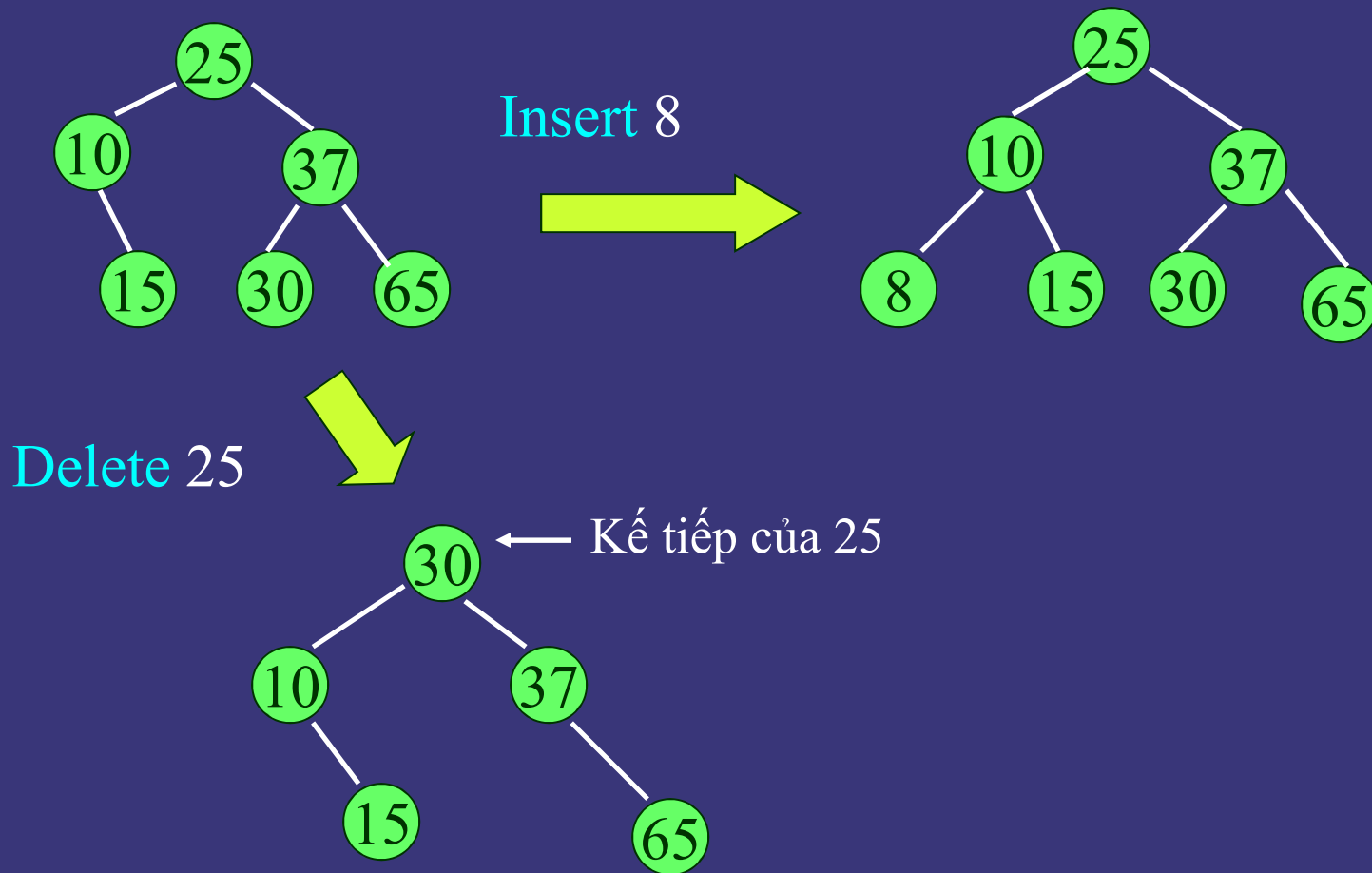
nodePtr



```
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;  
}
```

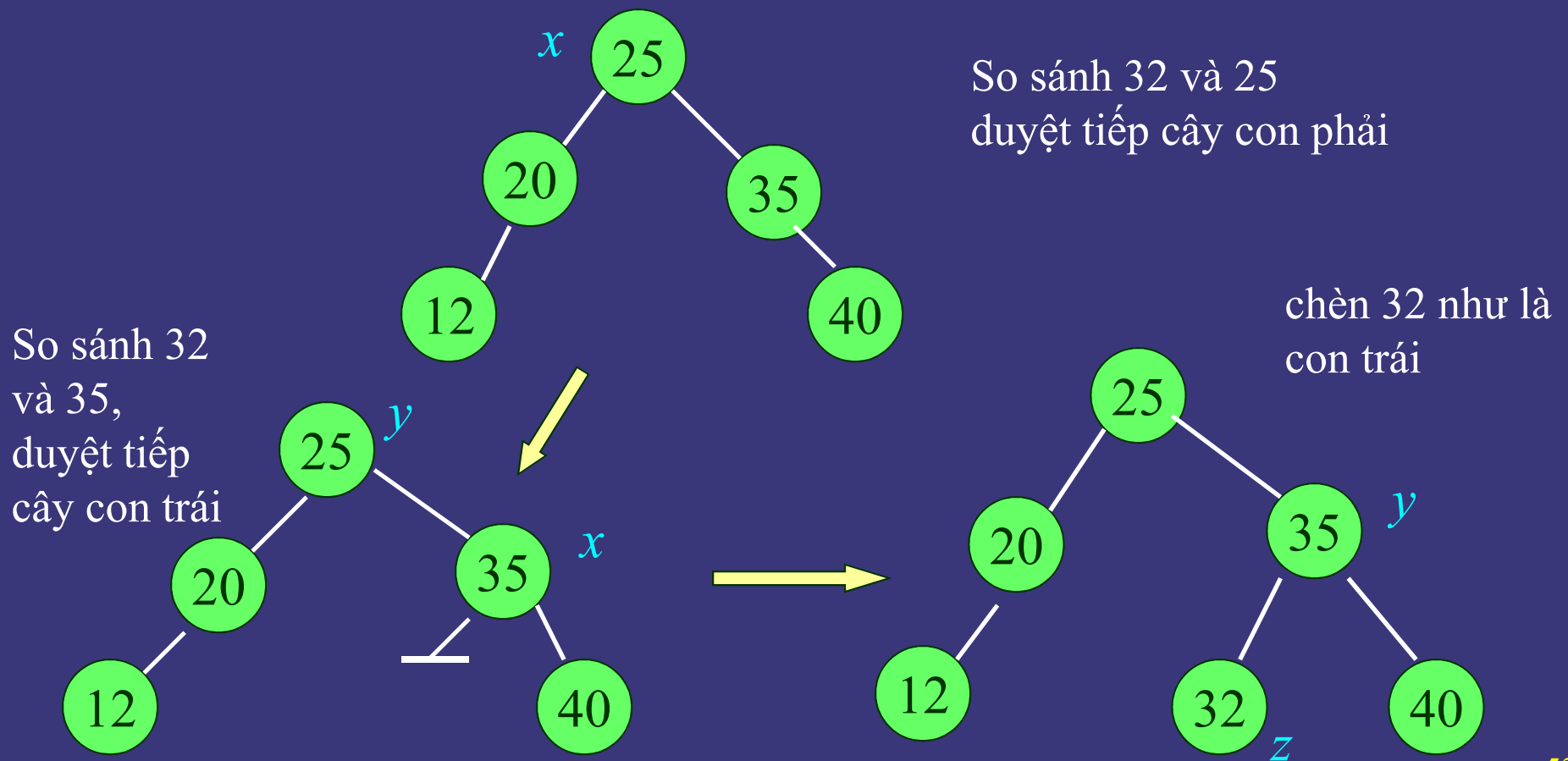
Chèn và Xoá trên BST

Chèn và xoá trên BST



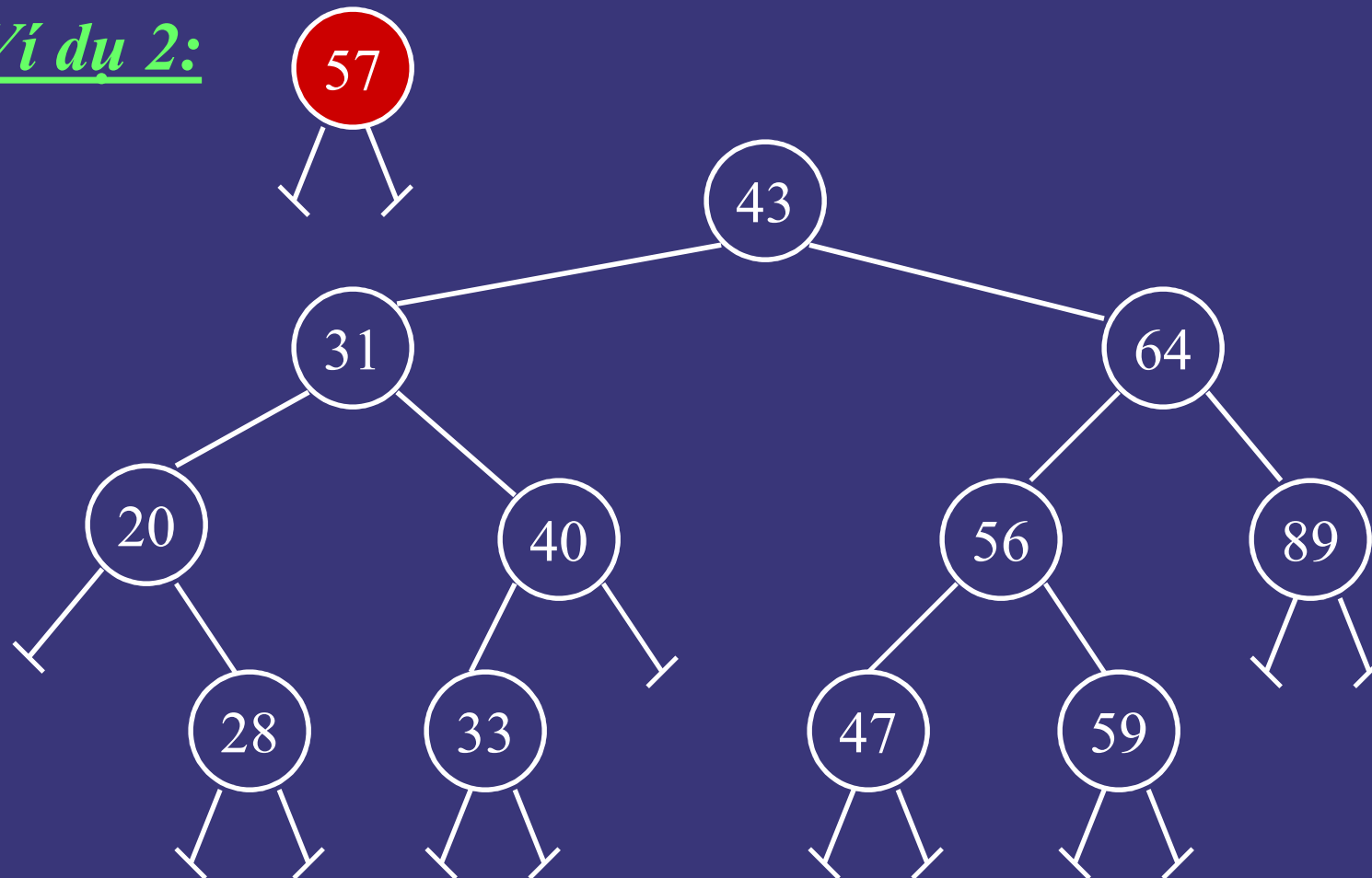
Bổ sung (Insertion)

Ví dụ 1: bổ sung $z = 32$



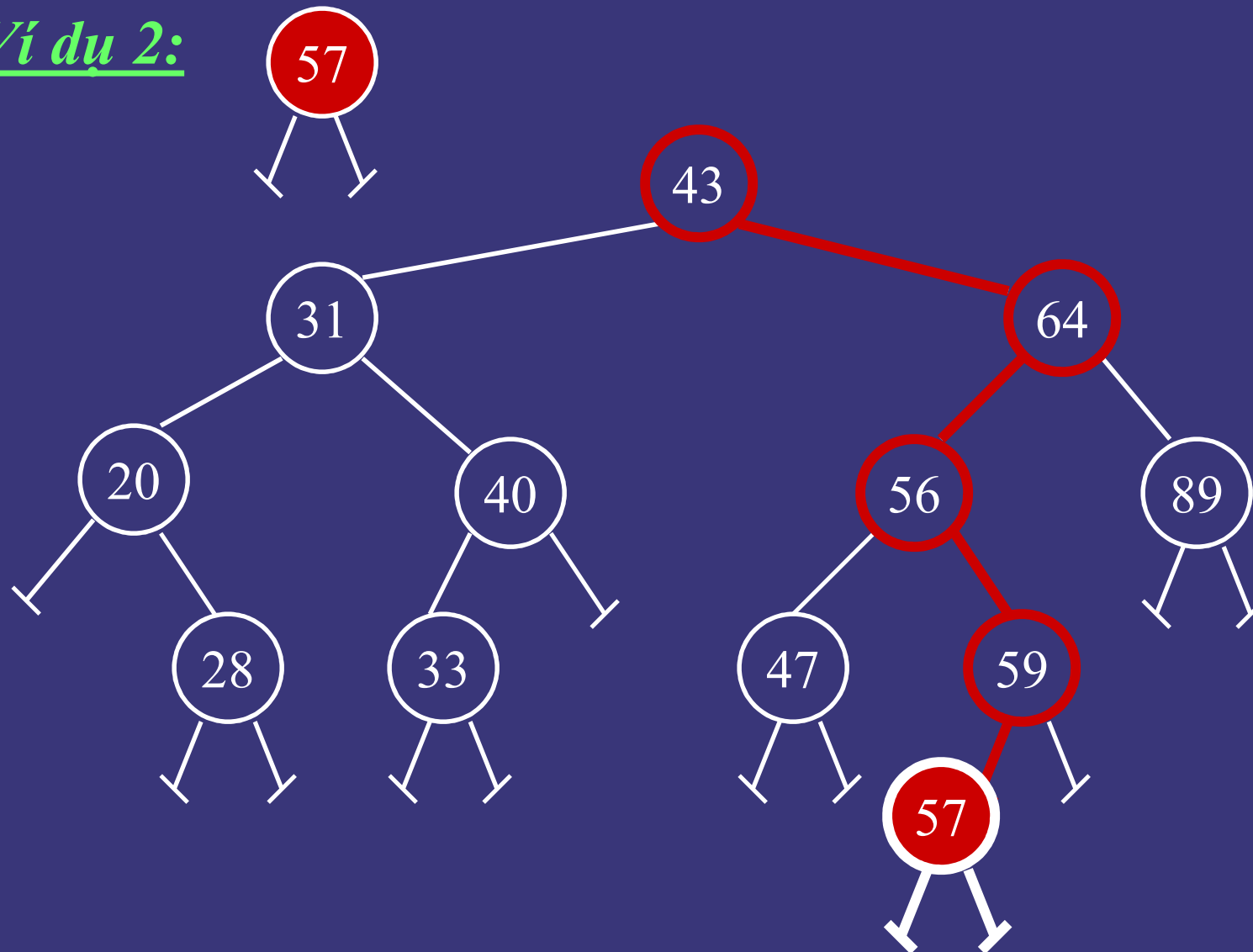
Insert

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.

Insert: Cài đặt đệ qui

- Thông số đầu vào:
 - con trỏ đến nút đang xét (thoạt tiên là nút gốc).
 - phần tử cần bổ sung.
- Nếu nút hiện thời là NULL
 - Tạo nút mới và trả lại nó.

trái lại, nếu khoá của phần tử bổ sung là nhỏ hơn (lớn hơn) khoá của nút hiện thời, thì tiếp tục quá trình với nút hiện thời là nút con trái (con phải).

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 ... */

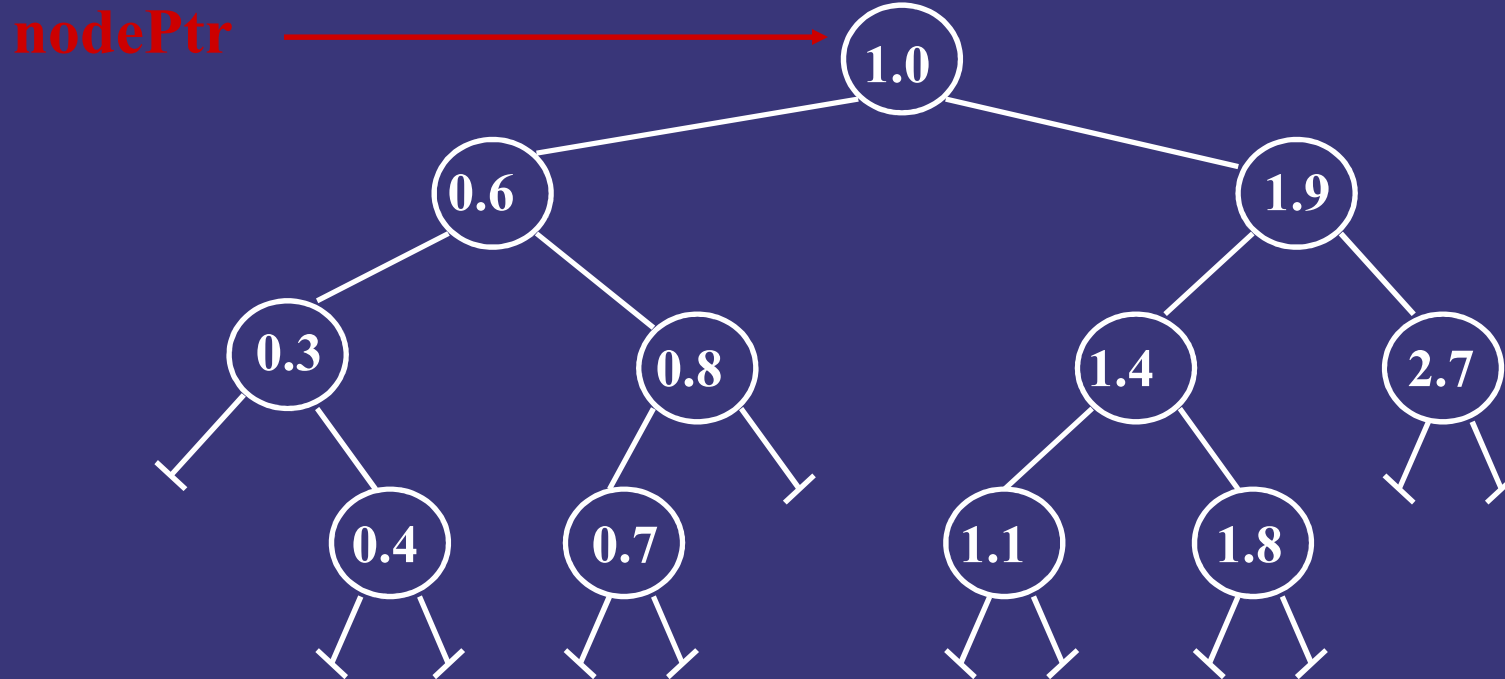
printf("Enter number of items ");
scanf("%d", &n);

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

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

Insert

Insert 0.9

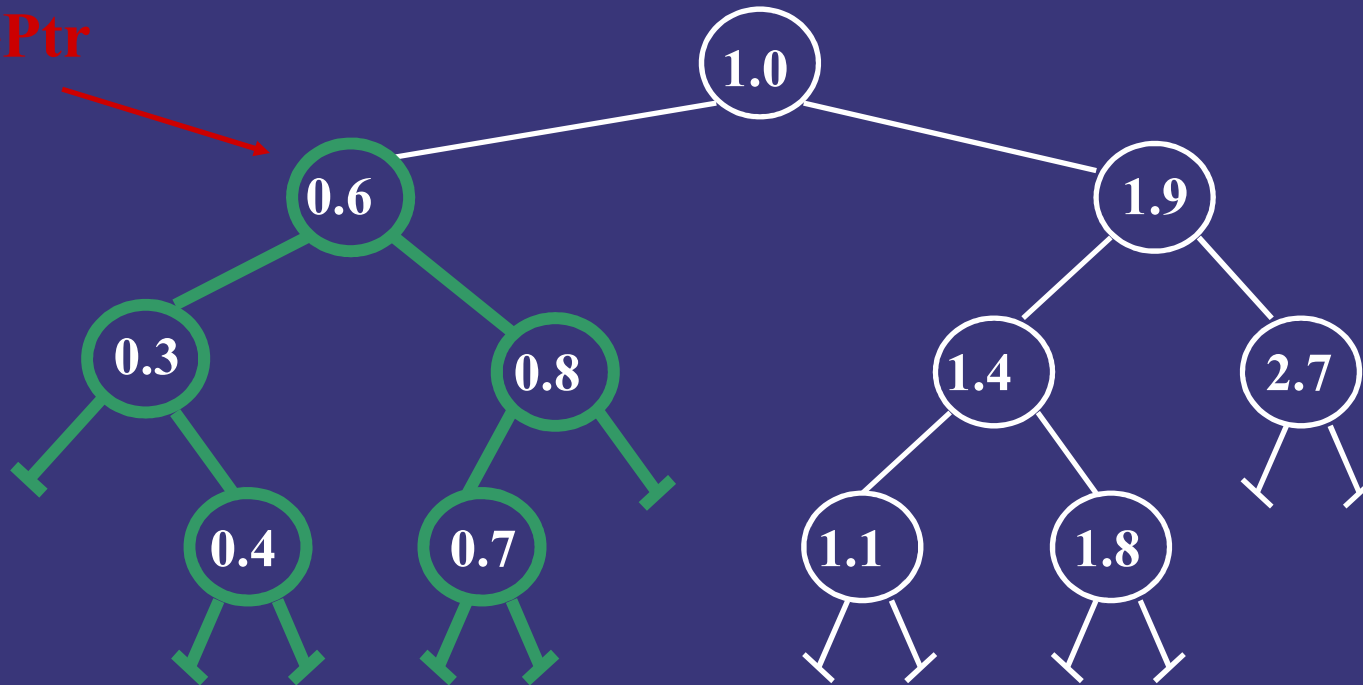


```
TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);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

Insert

Insert 0.9

nodePtr

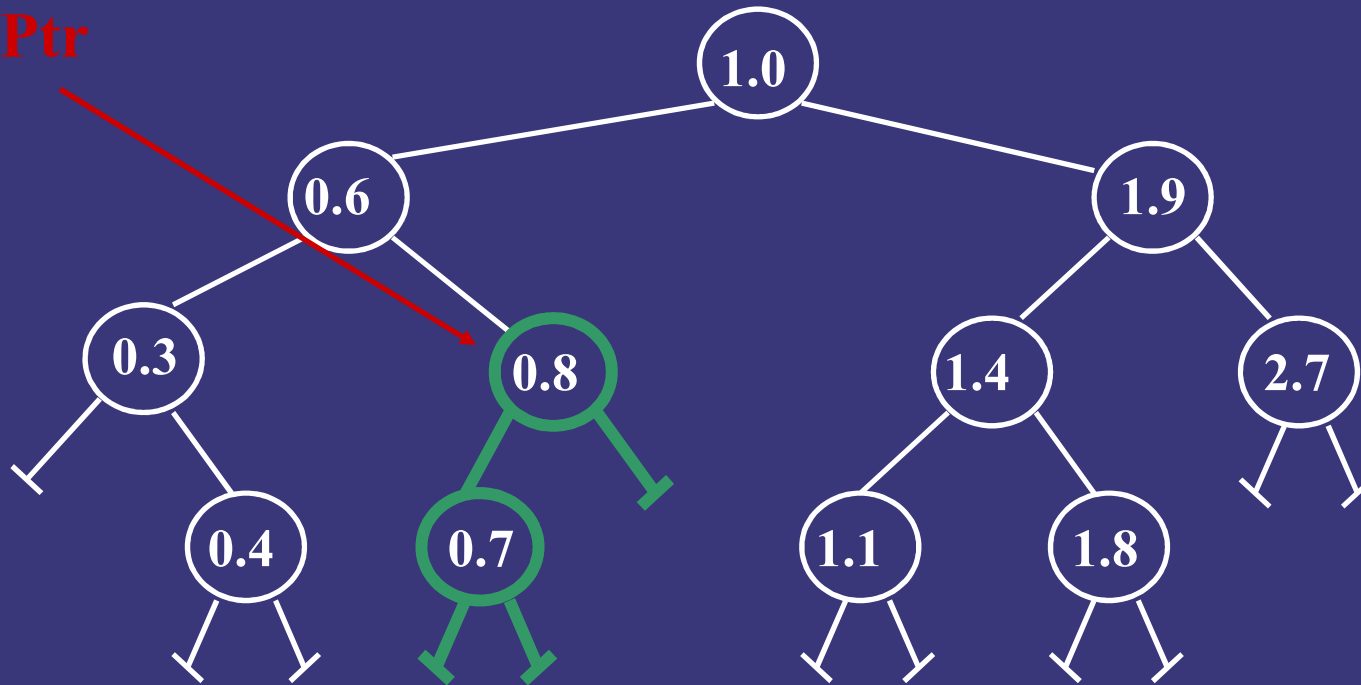


```
TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);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

Insert

Insert 0.9

nodePtr

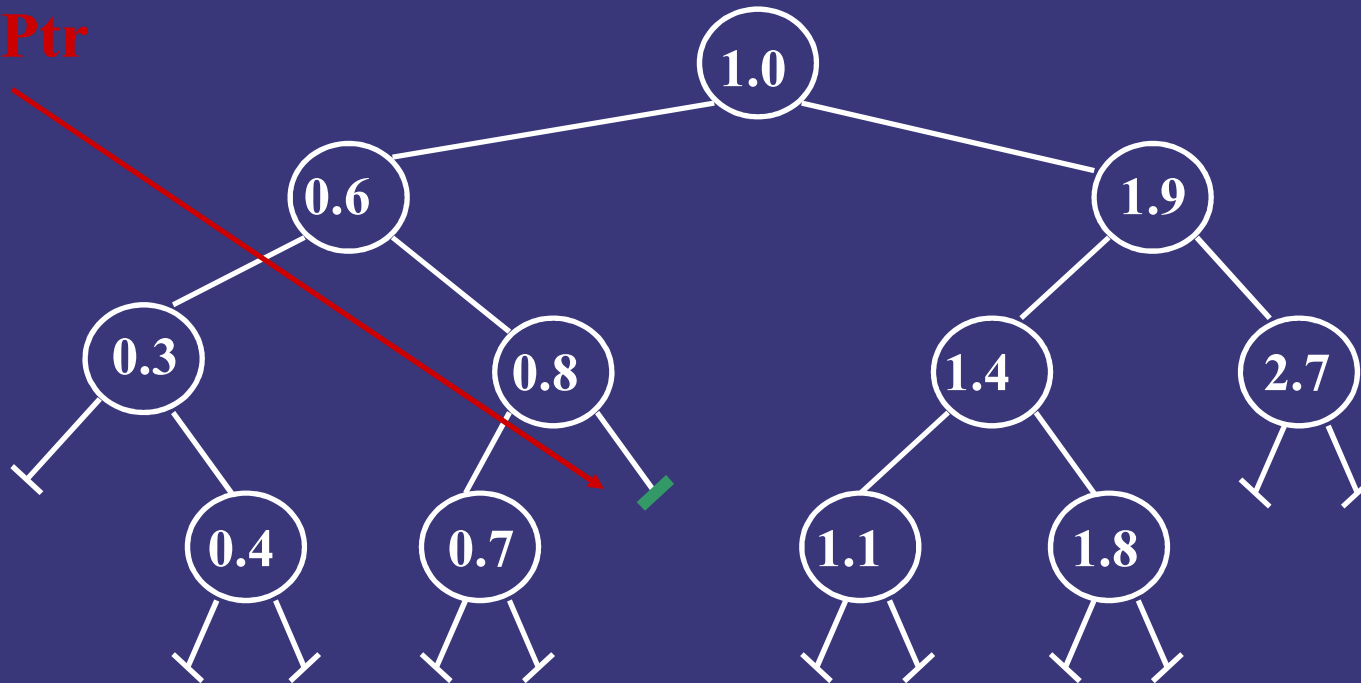


```
TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);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

Insert

Insert 0.9

nodePtr

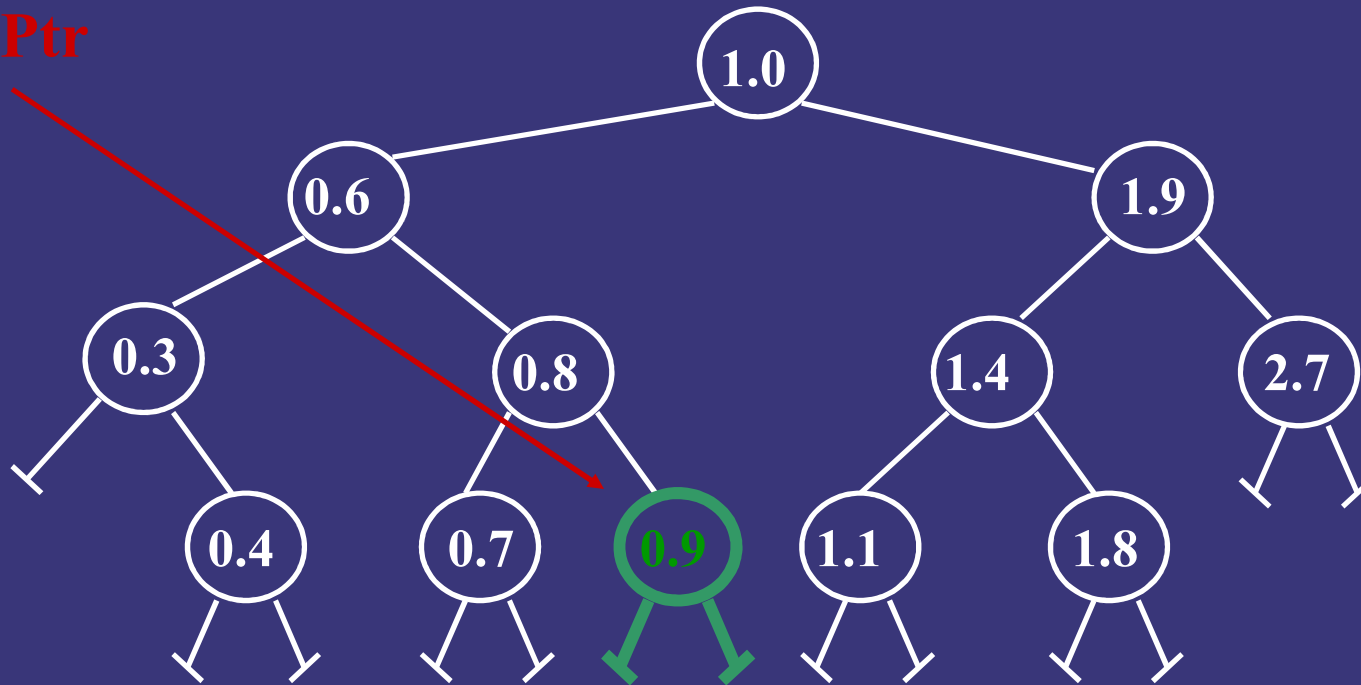


```
TreeNode* insert(TreeNode* nodePtr, float item)
{
    if (nodePtr == NULL)
        nodePtr = makeTreeNode(item);
    else if (item < nodePtr->key)
        nodePtr->leftPtr = insert(nodePtr->leftPtr, item);
    else if (item > nodePtr->key)
        nodePtr->rightPtr = insert(nodePtr->rightPtr, item);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

Insert

Insert 0.9

nodePtr



```
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;
}
```

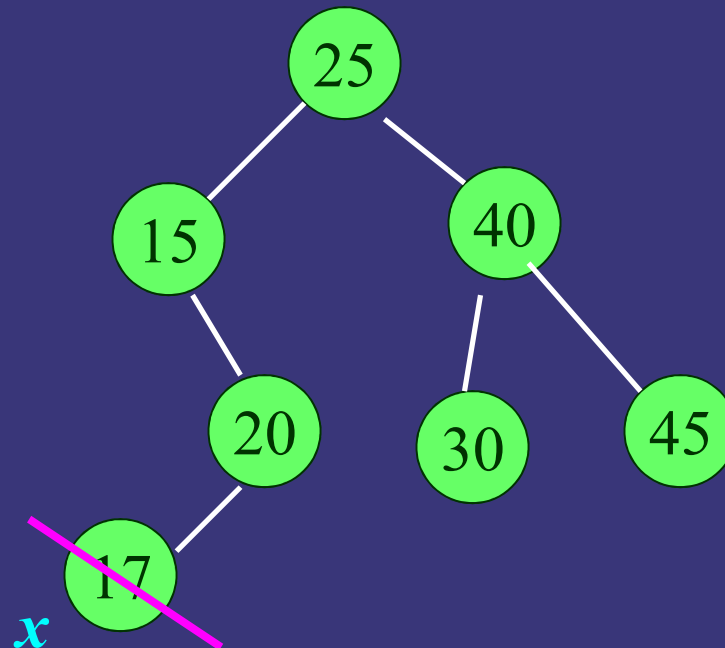

Thao tác loại bỏ - delete

- 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)

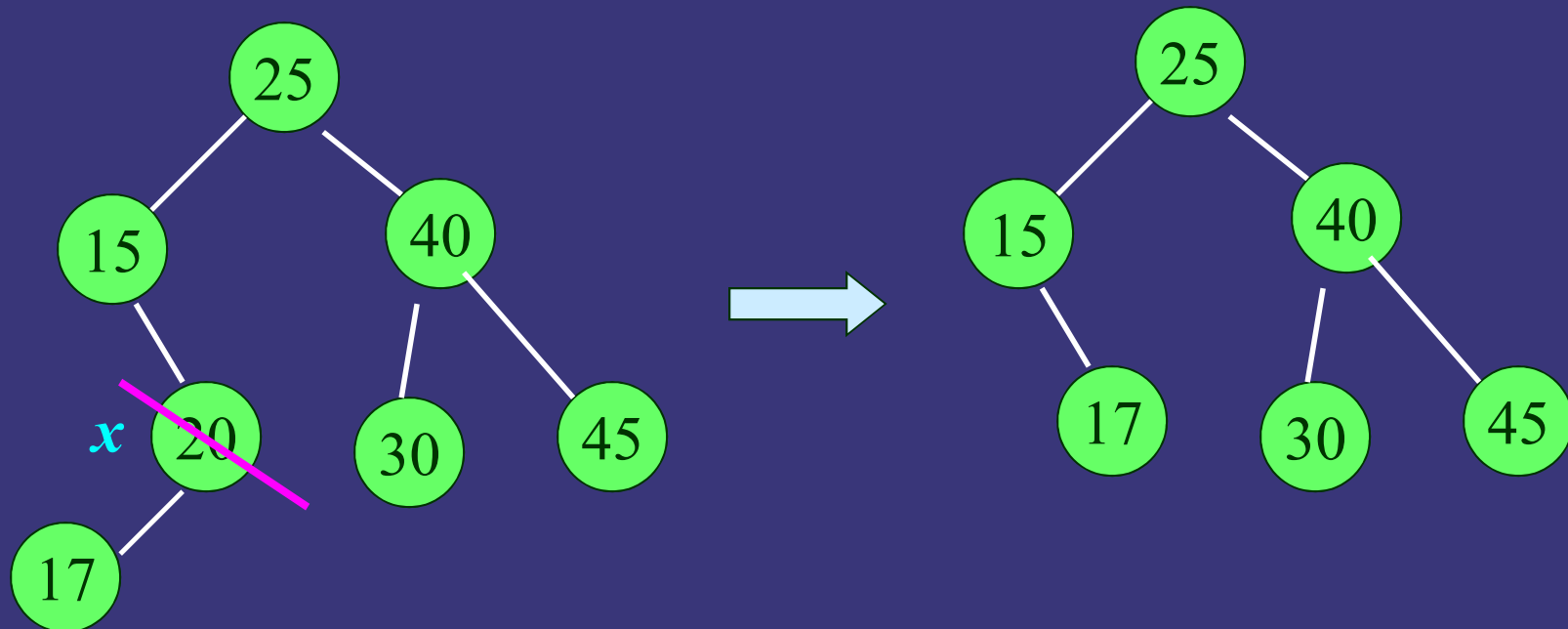
Thao tác: chữa lại nút cha của x có con rỗng.



Deletion: Tình huống 2

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

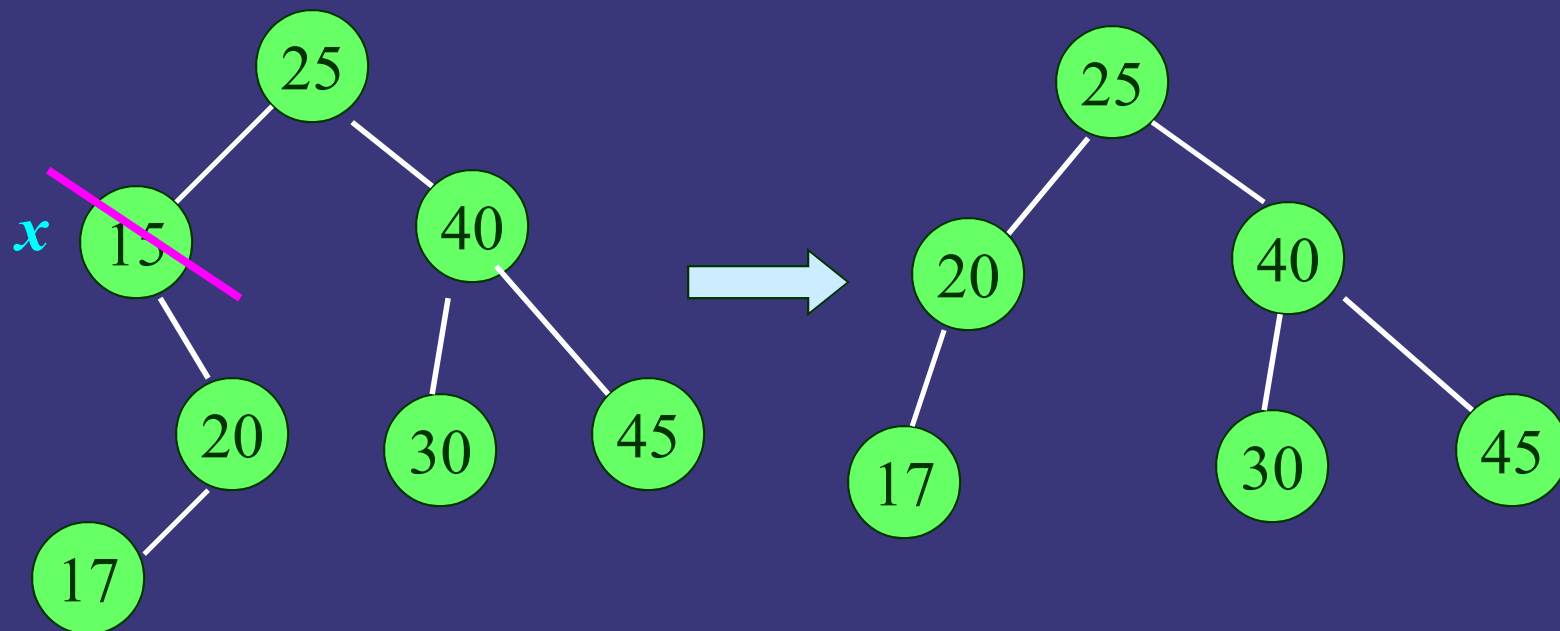
Thao tác: gắn cây con trái của x vào cha



Deletion: Tình huống 3

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

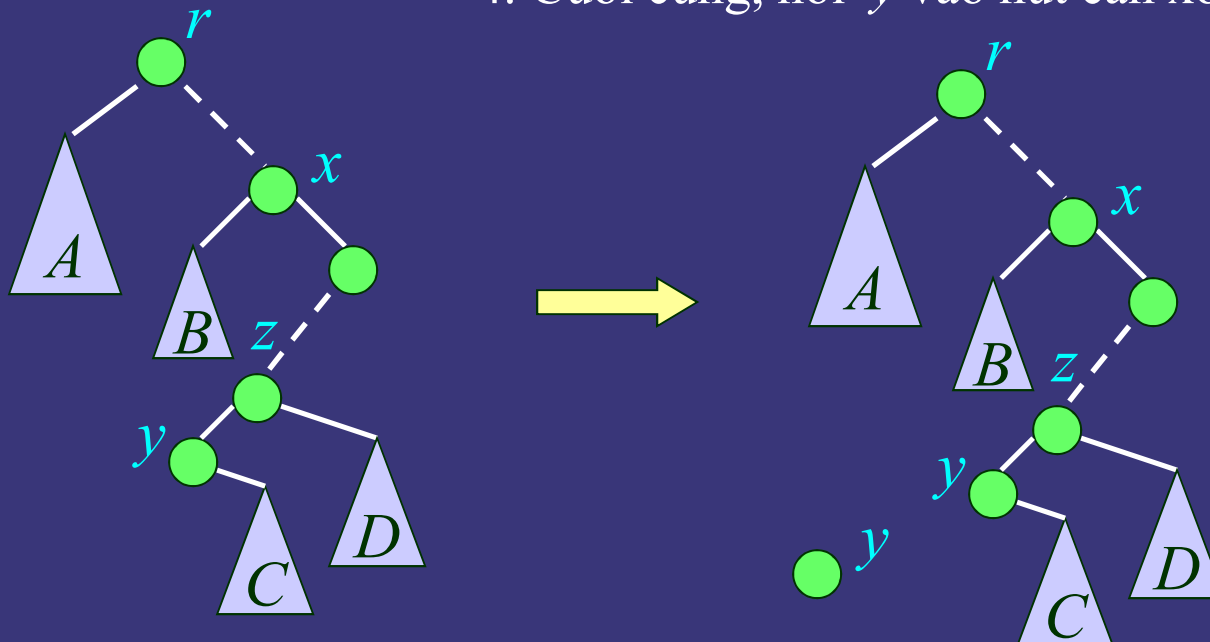
Thao tác: gán cây con phải của x vào cha



Deletion: Tình huống 4

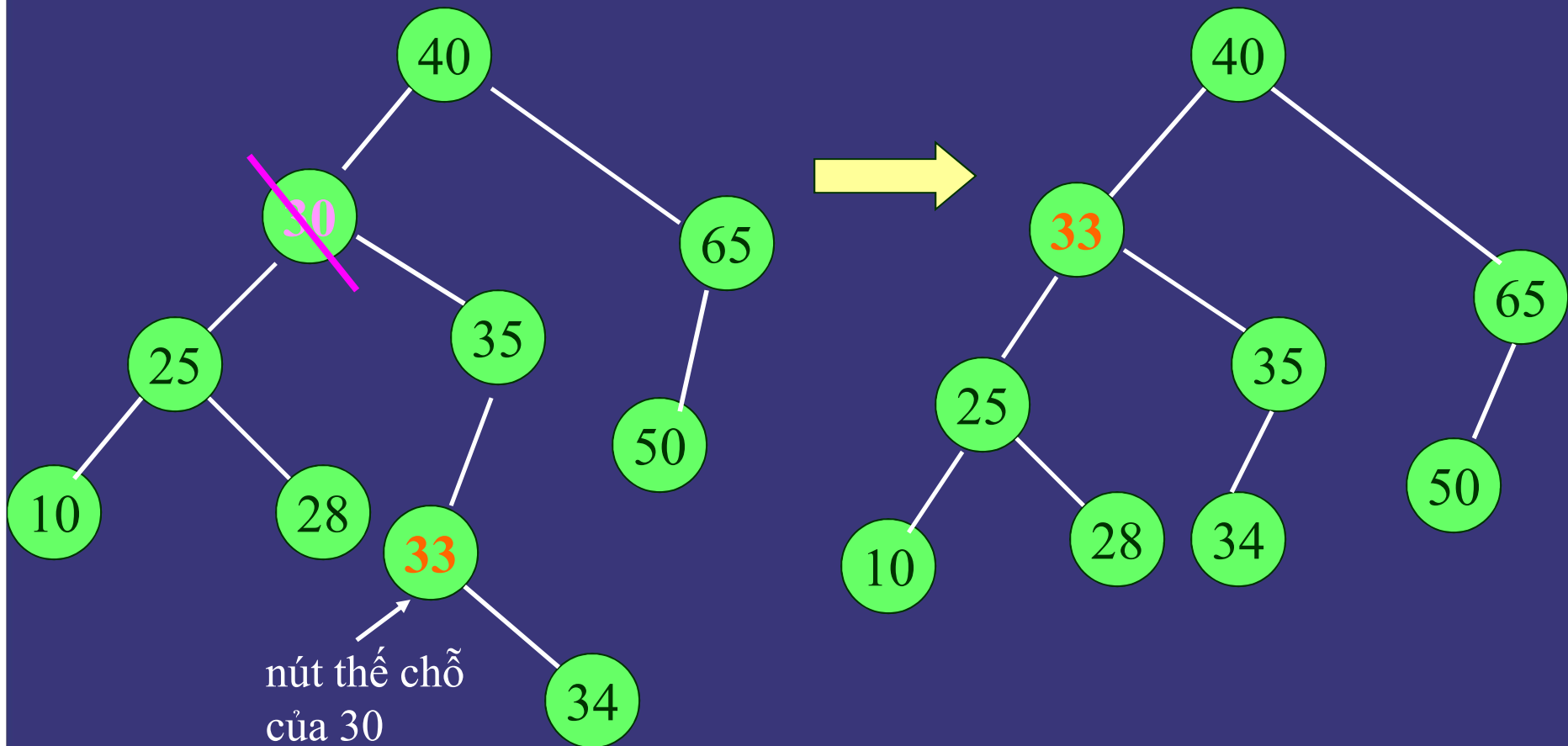
Tình huống 4: nút x có hai con

- 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 \longrightarrow 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->leftPtr == 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);
}
```

Độ phức tạp trung bình của các thao tác với BST

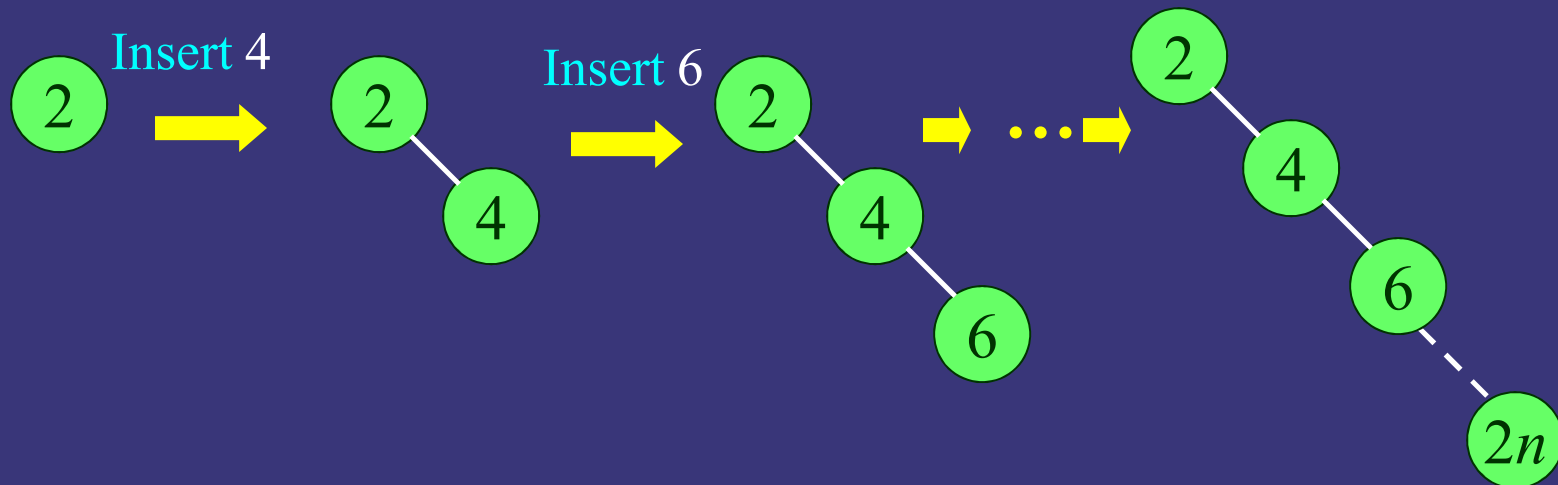
- Người ta chỉ ra được rằng độ cao trung bình của BST là
$$h = O(\log n).$$
- Từ đó suy ra độ phức tạp trung bình của các thao tác với BST là:

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

Thời gian tính tối 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 $h = O(\log n)$?

Độ phức tạp của các thao tác với BST

- Như đã biết, cây nhị phân có n nút có độ cao tối thiểu là $\log n$. Do đó, tình huống tốt nhất xảy ra khi ta dựng được BST là cây nhị phân đầy đủ (là cây có độ cao thấp nhất có thể được).
- Có hai cách tiếp cận nhằm đảm bảo độ cao của cây là $O(\log n)$:
 - Luôn giữ cho cây là cân bằng tại mọi thời điểm (AVL Trees)
 - thỉnh thoảng lại kiểm tra lại xem cây có "quá mất cân bằng" hay không và nếu điều đó xảy ra thì ta cần tìm cách cân bằng lại nó (Splay Trees [Tarjan])

6.3. Bảng băm

6.3.1. Đặt vấn đề

6.3.2. Địa chỉ trực tiếp

6.3.3. Hàm băm

6.3.1. 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.
- Trong mục này ta sẽ chỉ xét khoá là các số nguyên dương (có thể rất lớn)

Ứng dụng

- Xây dựng chương trình dịch của ngôn ngữ lập trình (Compiler): Ta cần thiết lập bảng ký hiệu trong đó khoá của các phần tử là dãy ký tự tương ứng với các từ định danh (identifiers) trong ngôn ngữ lập trình.
- Bảng băm là cấu trúc dữ liệu hiệu quả để cài đặt các từ điển (dictionaries).
- Mặc dù trong tình huống xấu nhất việc tìm kiếm đòi hỏi thời gian $O(n)$ giống như danh sách móc nối, nhưng 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 $O(1)$.
- Bảng băm 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 $O(1)$.

6.3. Bảng băm

6.3.1. Đặt vấn đề

6.3.2. Địa chỉ trực tiếp

6.3.3. Hàm băm

Địa chỉ trực tiếp

Direct Addressing

- **Giả thiết rằng:**

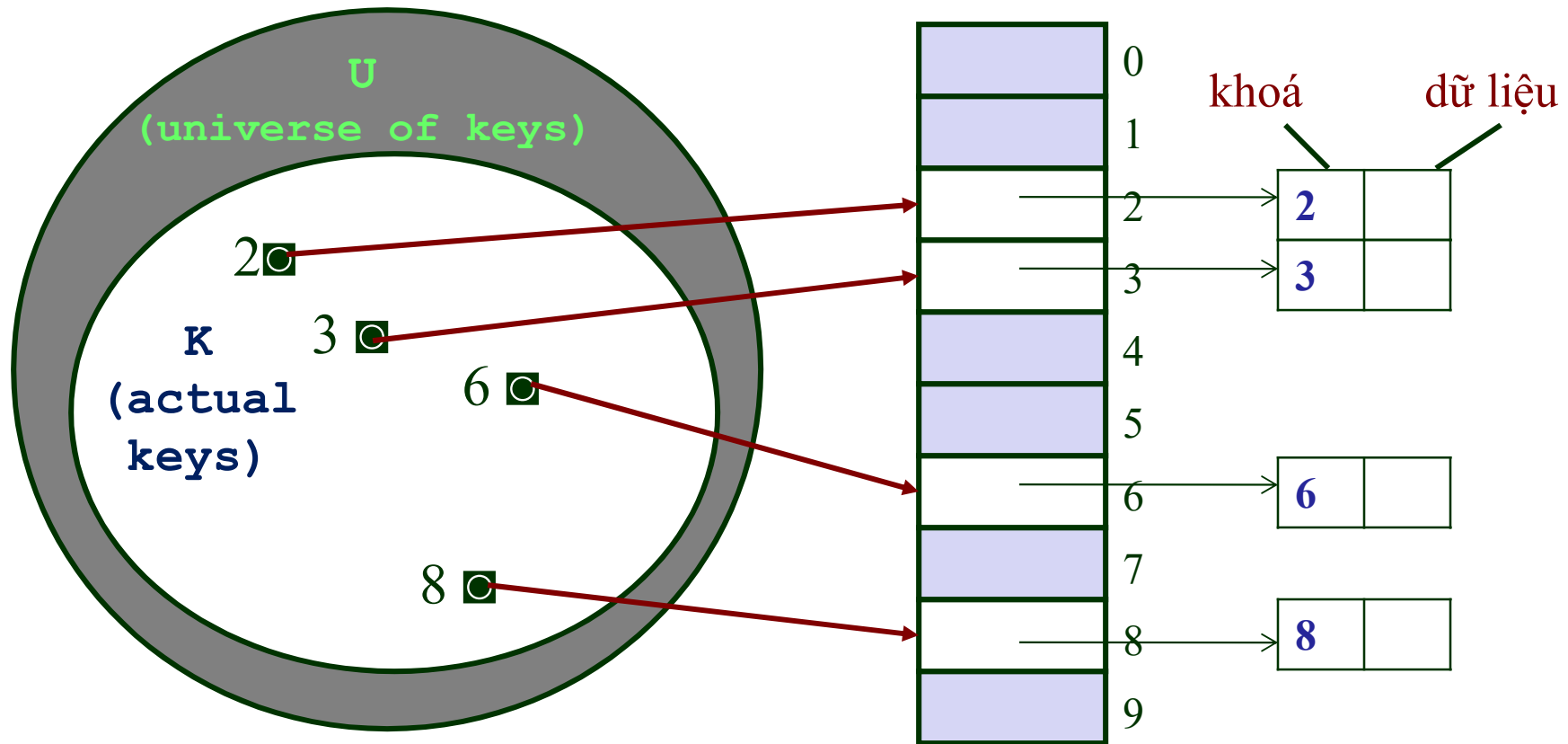
- Các khoá là các số trong khoảng từ 0 đến $m-1$
- Các khoá là khác nhau từng đôi

- **Ý tưởng:**

- Thiết lập mảng $T[0..m-1]$ trong đó
 - $T[i] = x$ nếu $x \in T$ và $\text{key}[x] = i$
 - $T[i] = \text{NULL}$ 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 $U = \{0, 1, \dots, 9\}$ tương ứng với một chỉ số trong bảng. Tập $K = \{2, 3, 5, 8\}$ 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:
- $\text{DIRECT-ADDRESS-SEARCH}(T, k)$
return $T[k]$
- $\text{DIRECT-ADDRESS-INSERT}(T, x)$
 $T[\text{key}[x]] = x$
- $\text{DIRECT-ADDRESS-DELETE}(T, x)$
 $T[\text{key}[x]] = \text{NULL}$
- 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*)

6.3. Bảng băm

6.3.1. Đặt vấn đề

6.3.2. Địa chỉ trực tiếp

6.3.3. Hàm băm

Hàm băm

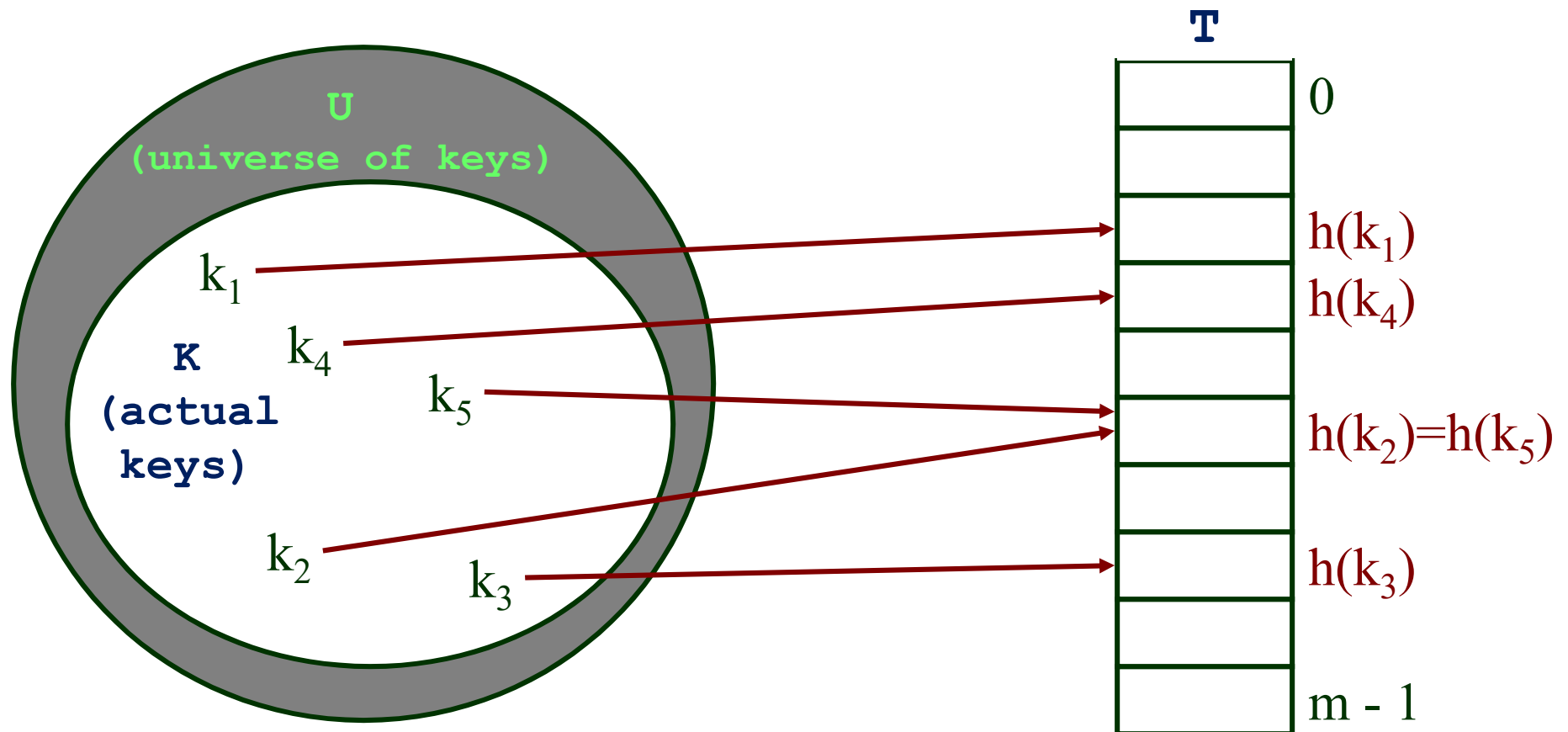
- Trong phương pháp địa chỉ trực tiếp, phần tử với khoá k được cất giữ ở ô k .
- Với bảng băm phần tử với khoá k được cất giữ ở ô $h(k)$, trong đó ta sử dụng hàm băm h để xác định ô cất giữ phần tử này từ khoá của nó (k).
- **Định nghĩa.** Hàm băm h là ánh xạ từ không gian khoá U vào các ô của bảng băm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

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

- 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:

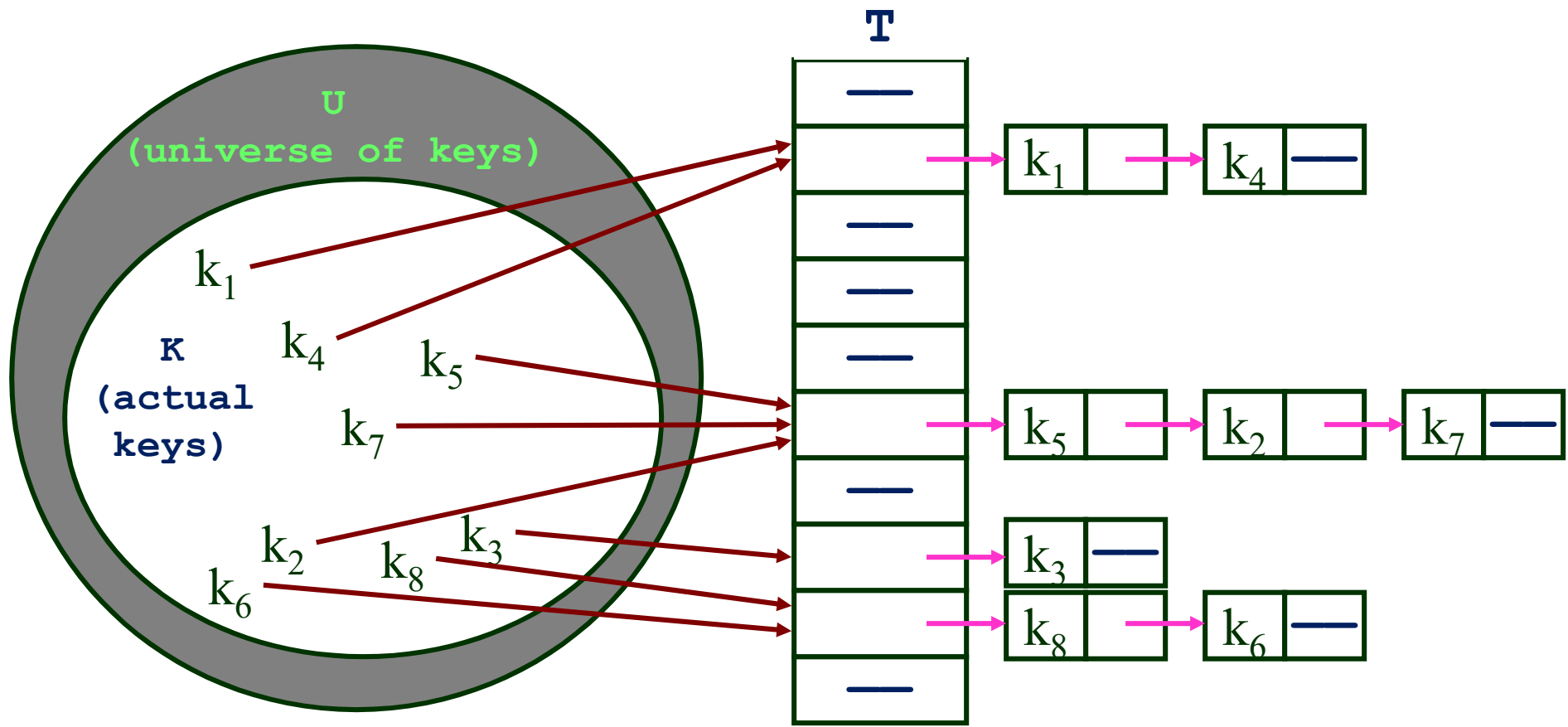
Tạo chuỗi (chaining)

- Cách giải quyết 2:

Phương pháp địa chỉ mở (open addressing)

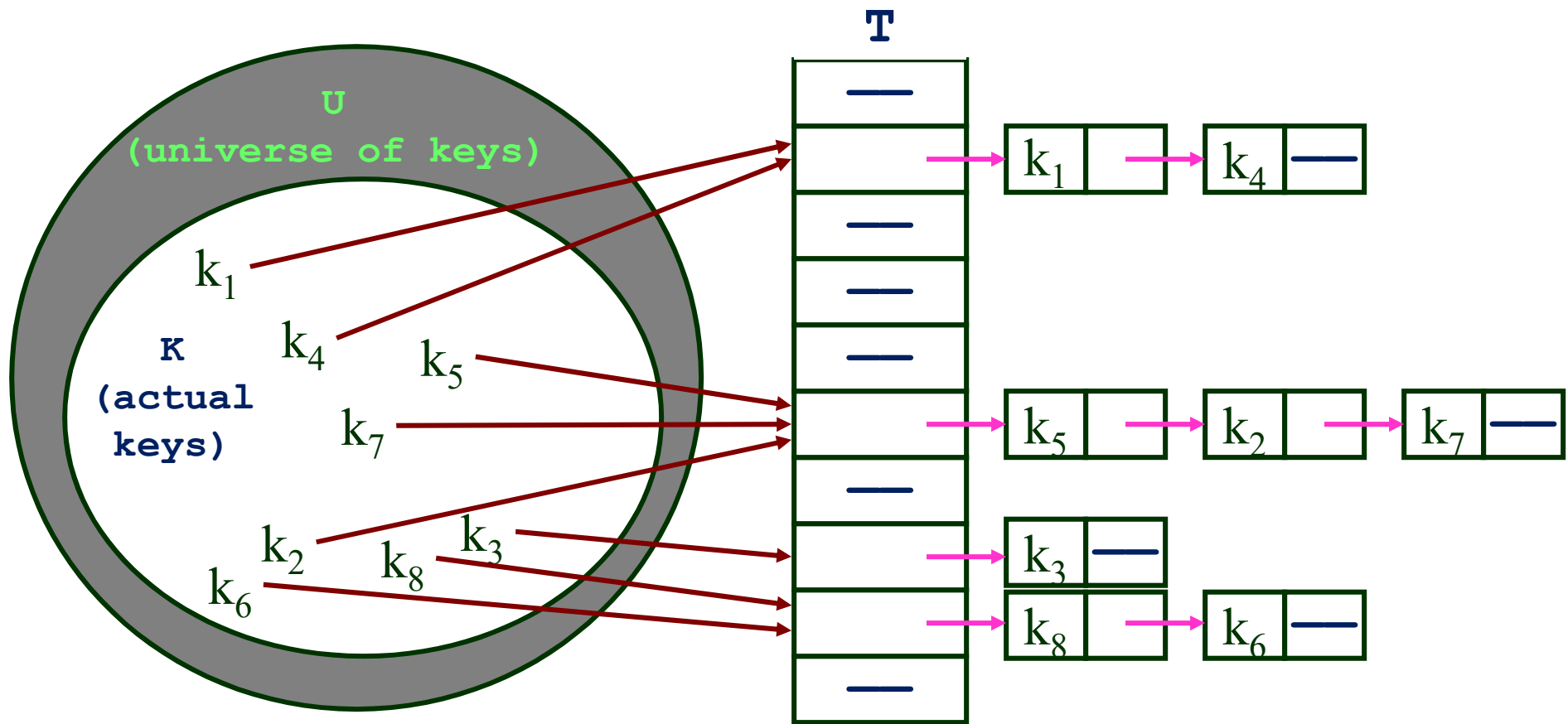
Tạo chuỗi (Chaining)

- 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.



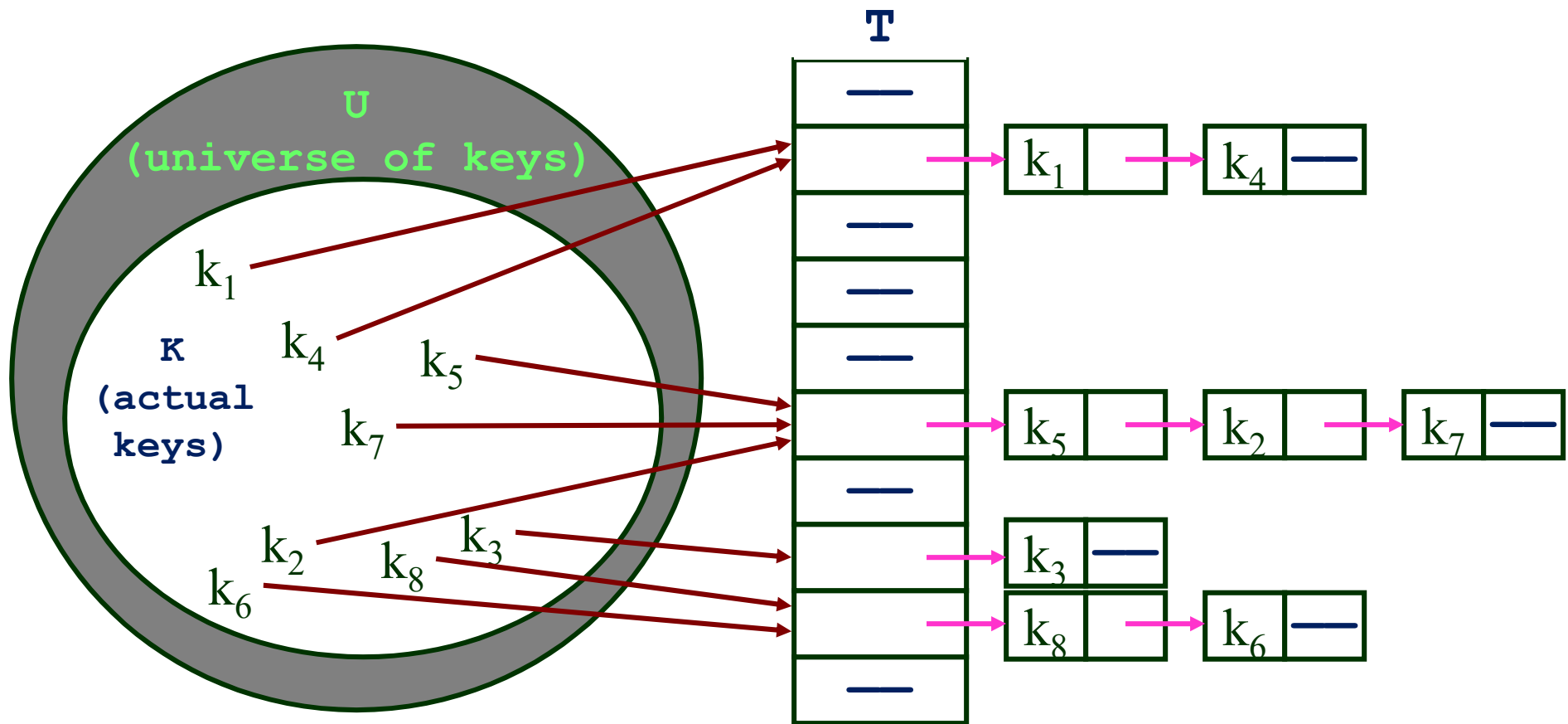
Tạo chuỗi (Chaining)

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



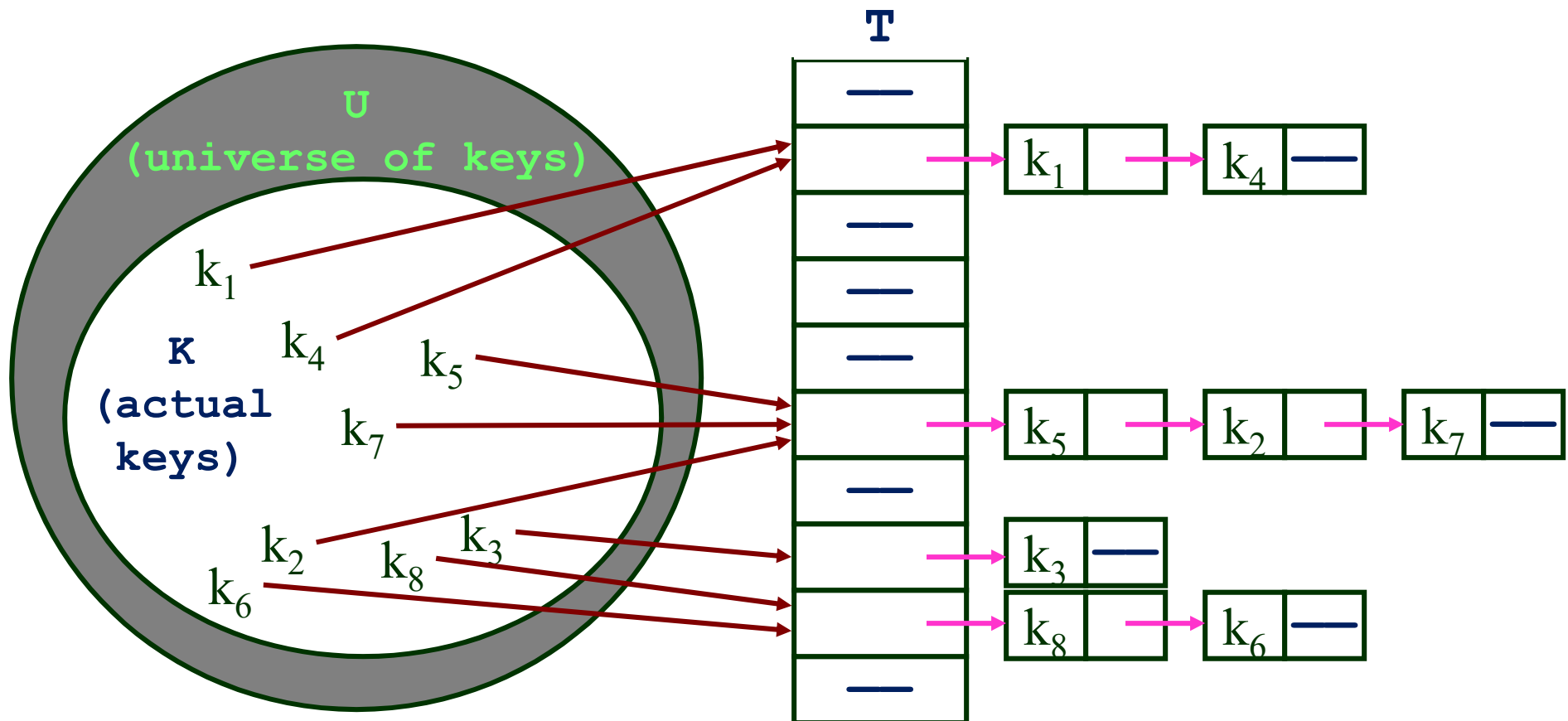
Tạo chuỗi (Chaining)

- Ta cần thực hiện loại bỏ phần tử như thế nào? Có cần sử dụng danh sách nối đôi để thực hiện xóa 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 trở 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 kiểm 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] = k;  
}
```

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

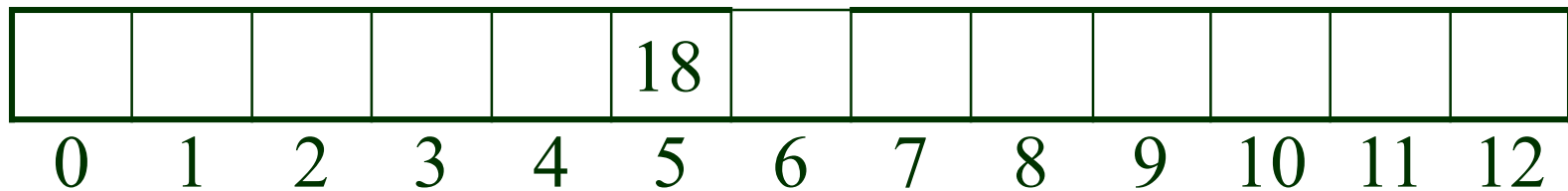
- **Ưu điểm:** Đơn giản bộ nhớ ít hơn phương pháp tạo chuỗi (không có móc nối)
- **Hạn chế:** Đơn giản nhiều thời gian hơn tạo chuỗi (nếu đường dò kiểm là dài)
- Thực hiện xóa bằng cách đánh dấu xóa (đánh dấu ô đã bị xóa)

Linear Probing

Sử dụng hàm băm: $h(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5,

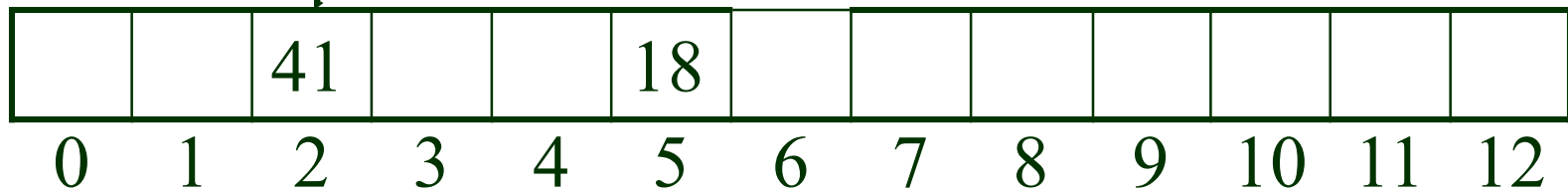


Linear Probing

$$h(k) = k \% 13$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2,




		41			18							
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing

$$h(k) = k \% 13$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2, 9,



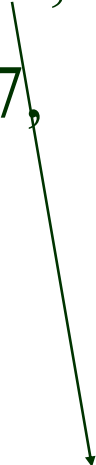
		41			18				22			
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing

$$h(k) = k \% 13$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2, 9, 7,



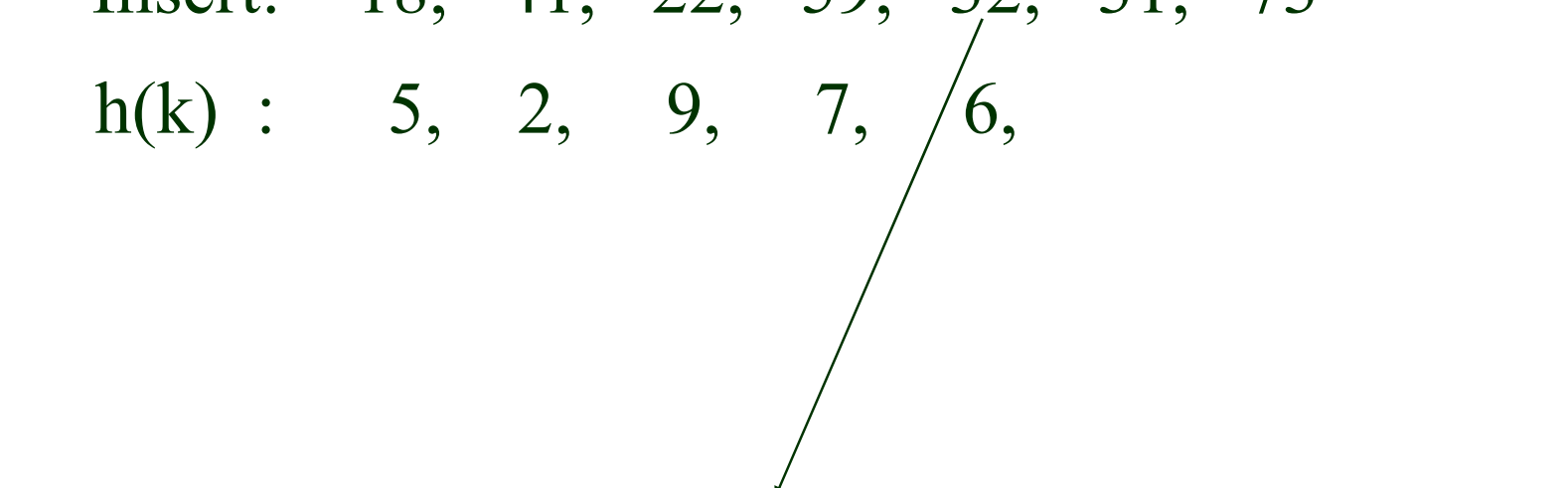
		41			18		59		22			
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing

$$h(k) = k \% 13$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2, 9, 7, 6,



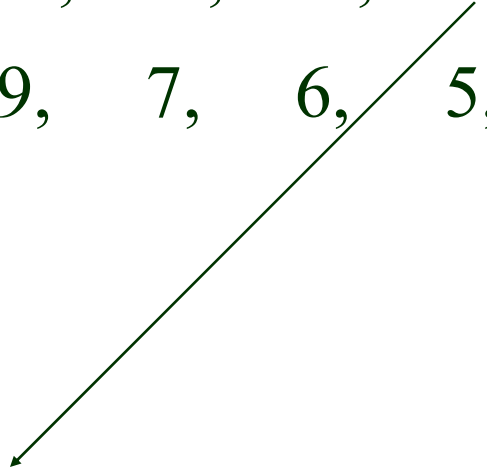
		41			18	32	59		22			
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing

$$h(k) = k \% 13$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2, 9, 7, 6, 5,



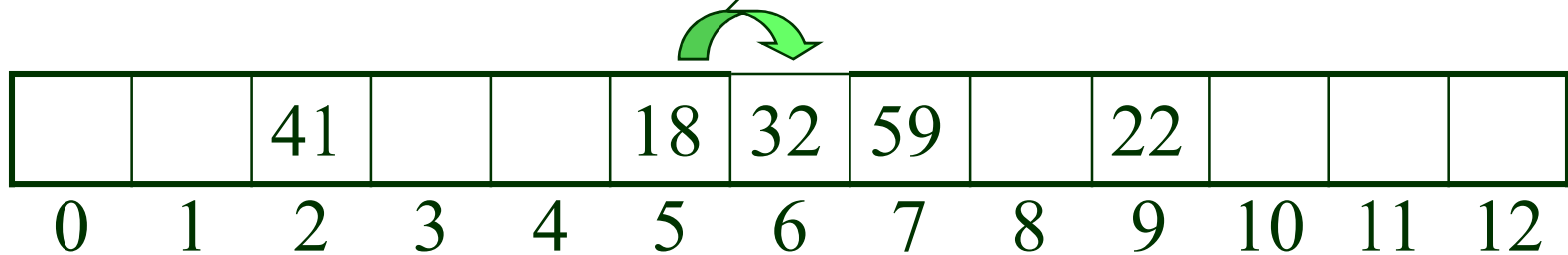
		41			18	32	59		22			
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing

$$h(k) = k \% 13$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2, 9, 7, 6, 5,

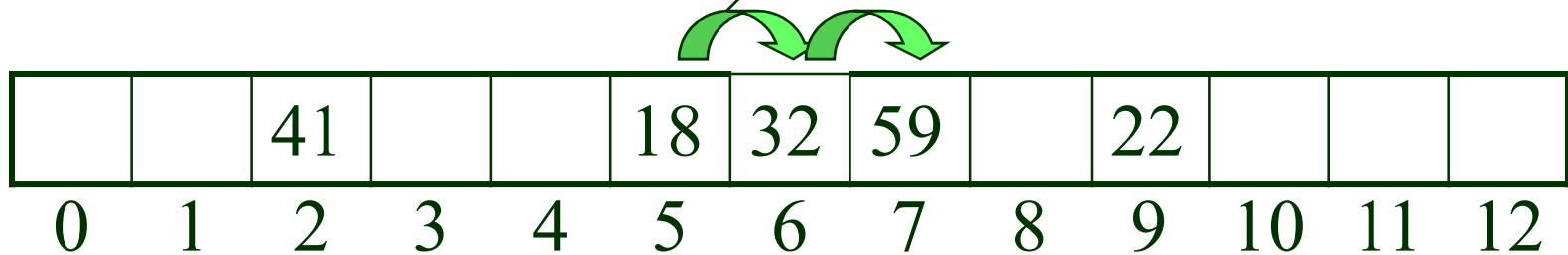


Linear Probing

$$h(k) = k \% 13$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2, 9, 7, 6, 5,

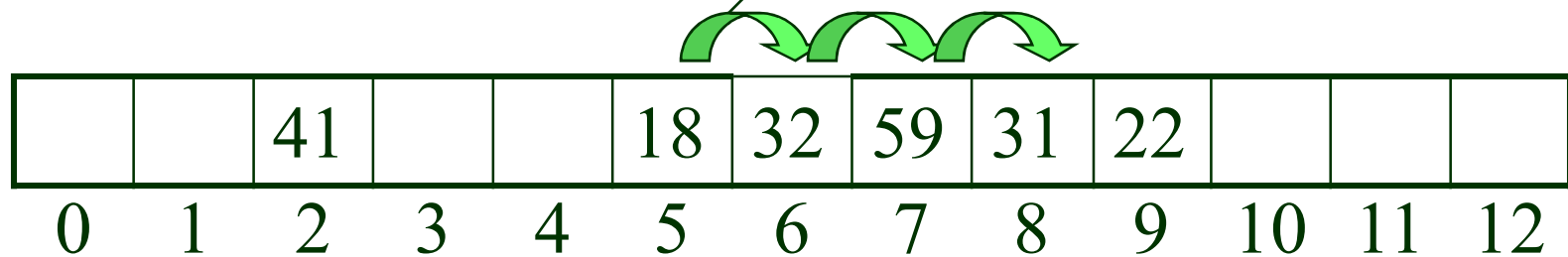


Linear Probing

$$h(k) = k \% 13$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2, 9, 7, 6, 5,

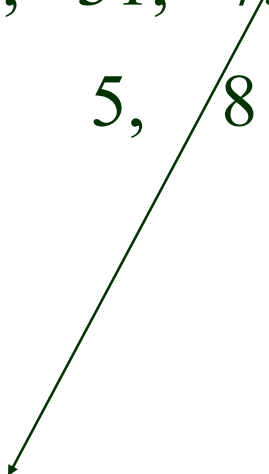


Linear Probing

$$h(k) = k \% 13$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2, 9, 7, 6, 5, 8



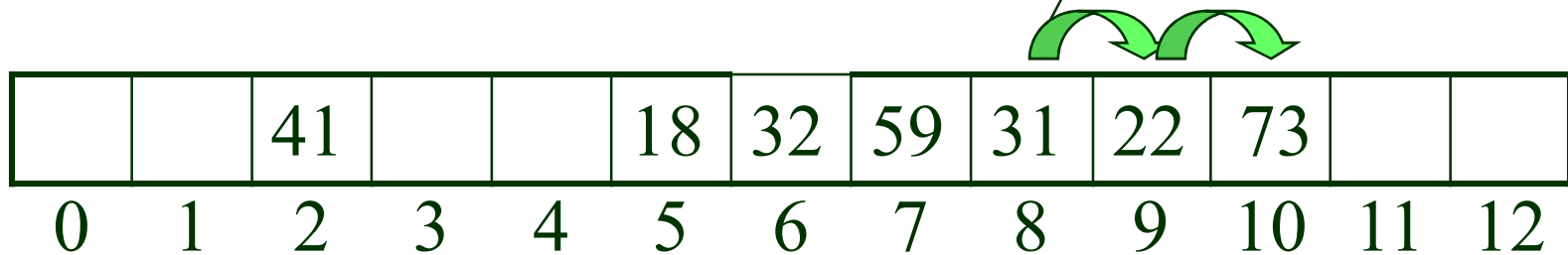
		41			18	32	59	31	22			
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing

$$h(k) = k \% 13$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2, 9, 7, 6, 5, 8



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

Double Hashing

$$h_1(k) = k \% 13$$

$$h_2(k) = 8 - k \% 8$$

$$\text{probe} = (\text{probe} + \text{offset}) \% m;$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h_1(k)$: 5, 2, 9, 7, 6, 5, 8

$h_2(k)$: 6, 7, 2, 5, 8, 1, 7

0	1	2	3	4	5	6	7	8	9	10	11	12

Double Hashing

$$h_1(k) = k \% 13$$

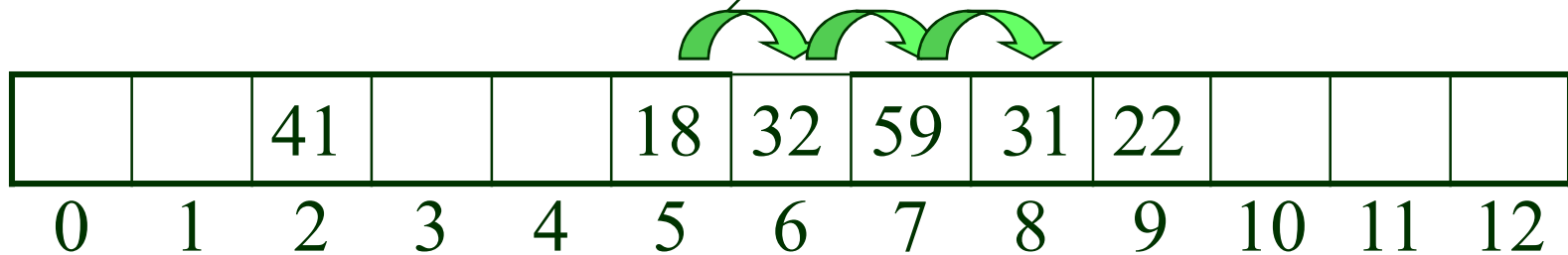
$$h_2(k) = 8 - k \% 8$$

$$\text{probe} = (\text{probe} + \text{offset}) \% m;$$

Insert: 18, 41, 22, 59, 32, 31, 73

$h_1(k)$: 5, 2, 9, 7, 6, 5, 8

$h_2(k)$: 6, 7, 2, 5, 8, 1, 7



Double Hashing

$$h_1(k) = k \% 13$$

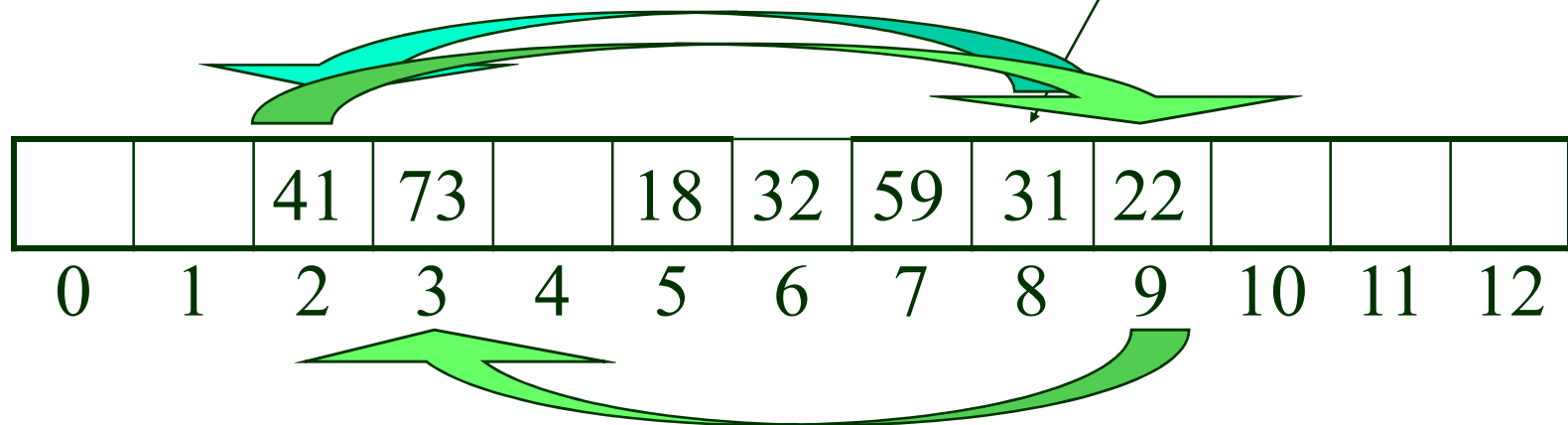
$$h_2(k) = 8 - k \% 8$$

```
probe = (probe+offset) % m;
```

Insert: 18, 41, 22, 59, 32, 31, 73

$h_1(k) : 5, 2, 9, 7, 6, 5, 8$

$h_2(k) : 6, 7, 2, 5, 8, 1, 7$

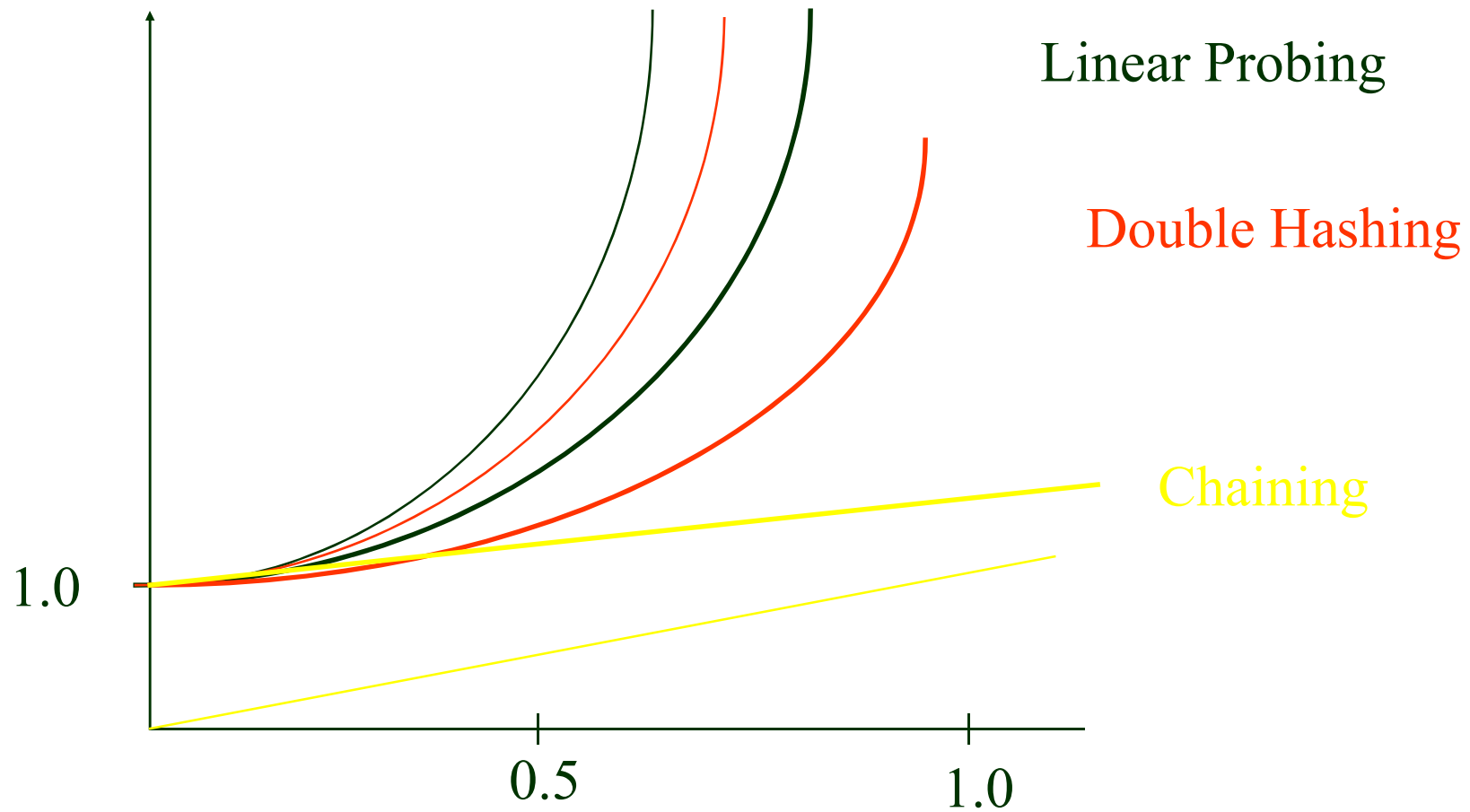


Kết quả lý thuyết: Số phép thử trung bình

	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	$\frac{1}{(1 - \alpha)}$	$\frac{1}{\alpha} \ln \frac{1}{(1 - \alpha)}$

$$\alpha = \text{<số lượng phần tử trong bảng>/<kích thước bảng>}$$

Expected Probes



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: gán 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 lũy thừa của 2 (chẳng hạn 2^p)?*
- Ans: khi đó $h(k)$ chính là p bit cuối của k
- *Điều gì xảy ra nếu m là lũy 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 lũy 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 , $0 < A < 1$:

➤
$$h(k) = \lfloor m \underbrace{(kA - \lfloor kA \rfloor)}_{\text{Phần thập phân của } kA} \rfloor$$

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?

