



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



Cấu trúc dữ liệu và giải thuật

Nguyễn Khánh Phương

**Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn**

Nội dung khóa học

Chương 1. Các khái niệm cơ bản

Chương 2. Các cấu trúc dữ liệu cơ bản

Chương 3. Cây

Chương 4. Sắp xếp

Chương 5. Tìm kiếm



TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG



Chương 5. Tìm kiếm (Searching)

Nguyễn Khánh Phương

**Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn**

Bài toán tìm kiếm (Searching problem)

Cho danh sách A gồm n phần tử a_1, a_2, \dots, a_n và 1 số x .

Câu hỏi: x có mặt trong danh sách A hay không?

- Nếu x có mặt trong danh sách A, hãy đưa ra vị trí xuất hiện của x trong danh sách đã cho, nghĩa là đưa ra chỉ số i sao cho $a_i = x$

Nội dung

1. Tìm kiếm tuần tự
2. Tìm kiếm nhị phân
3. Cây nhị phân tìm kiếm
4. Bảng băm
5. Tìm kiếm xâu mẫu

Nội dung

1. Tìm kiếm tuần tự
2. Tìm kiếm nhị phân
3. Cây nhị phân tìm kiếm
4. Bảng băm
5. Tìm kiếm xâu mẫu

1. Tìm kiếm tuần tự (Linear Search/Sequential search)

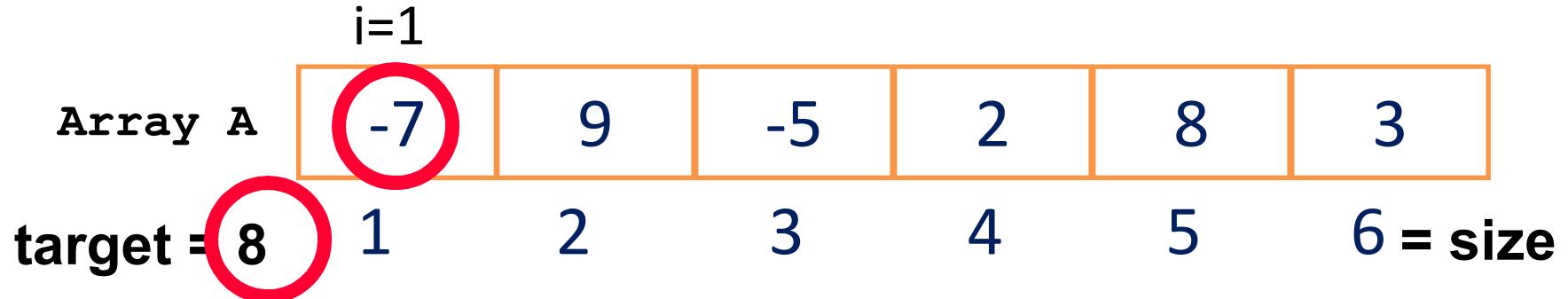
- Đầu vào:
 - Cho mảng A gồm n phần tử và giá trị tìm kiếm x .
 - Mảng A không cần thiết đã được sắp xếp
- Thuật toán: Bắt đầu từ phần tử đầu tiên, duyệt qua từng phần tử cho đến khi tìm được x hoặc toàn bộ các phần tử của mảng đã được duyệt hết
- Độ phức tạp: $O(n)$

A:	-7	9	-5	2	8	3
	1	2	3	4	5	6

Target $x = 8$:

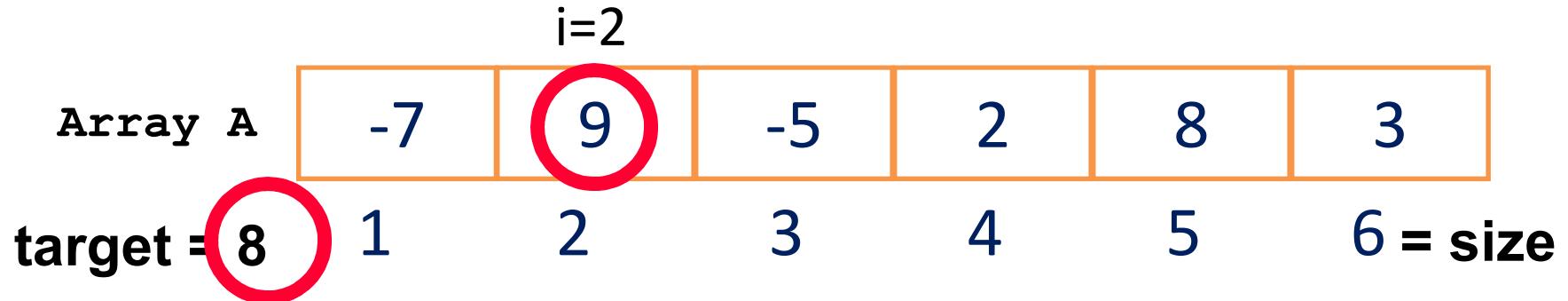
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



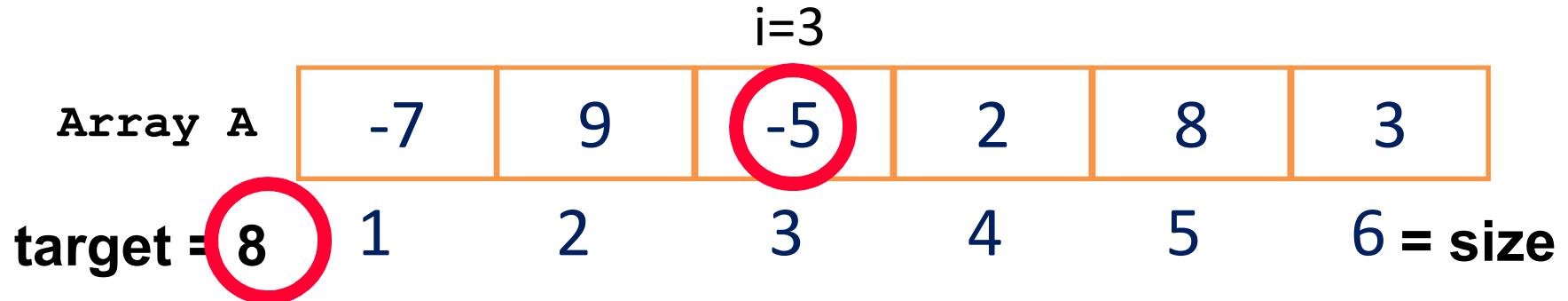
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



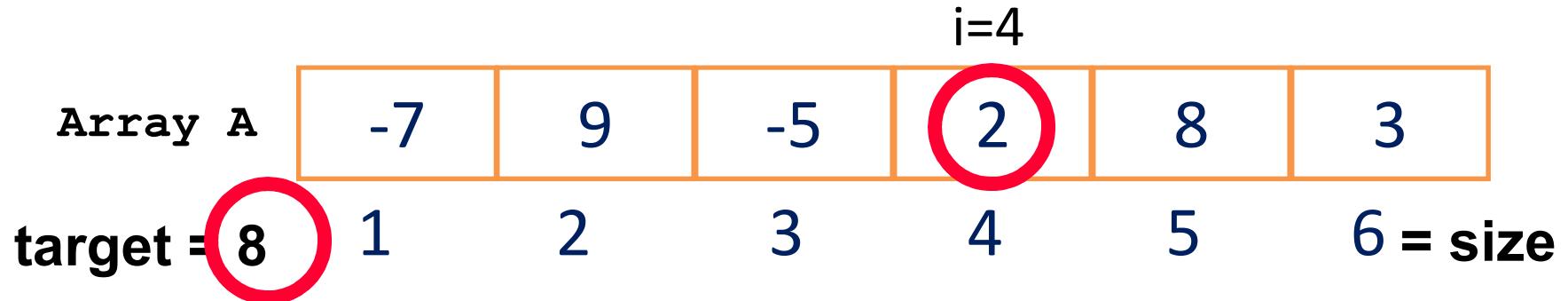
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



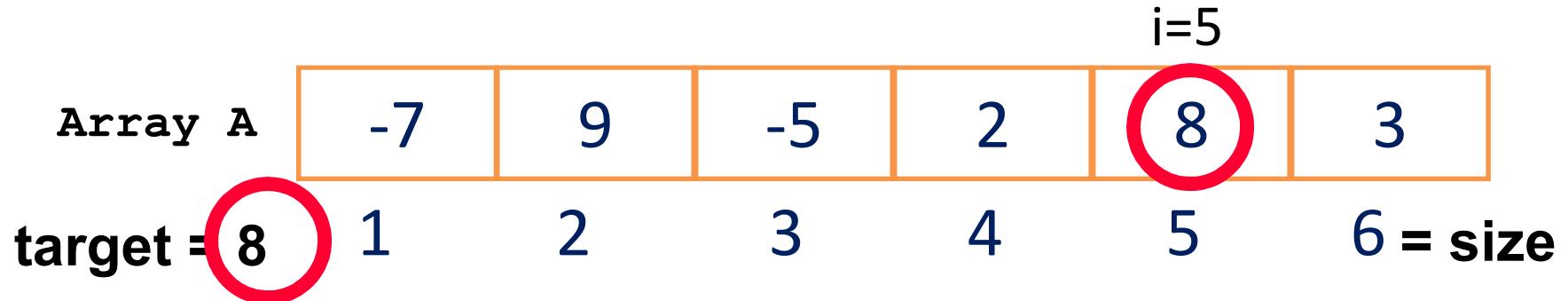
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



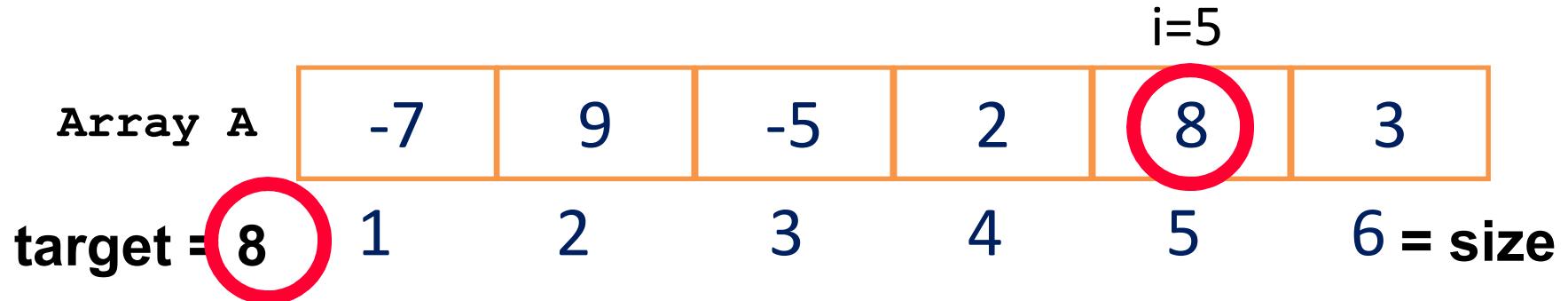
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



1. Tìm kiếm tuần tự (Linear Search/Sequential search)

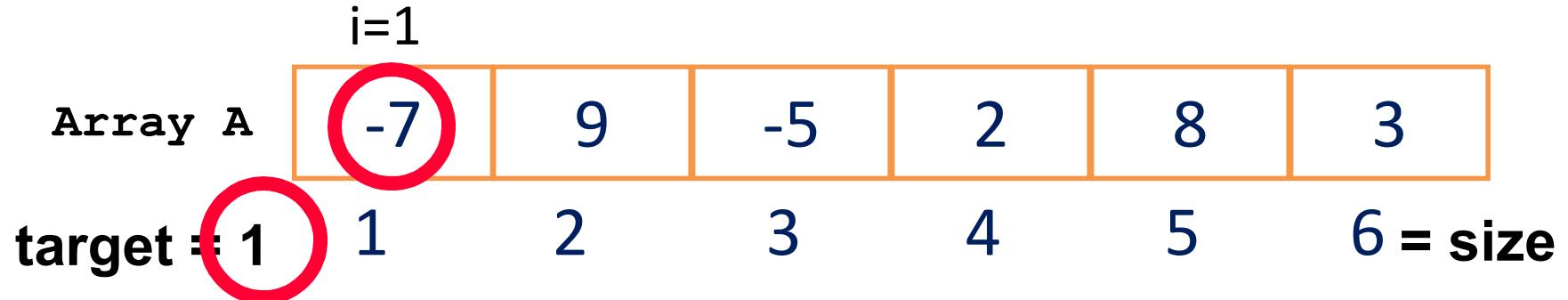
```
void linearSearch(int a[], int size, int target)
{    int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d", i);
    else
        printf("The target is NOT in the list");
}
```



The target is in the list @ index = 5

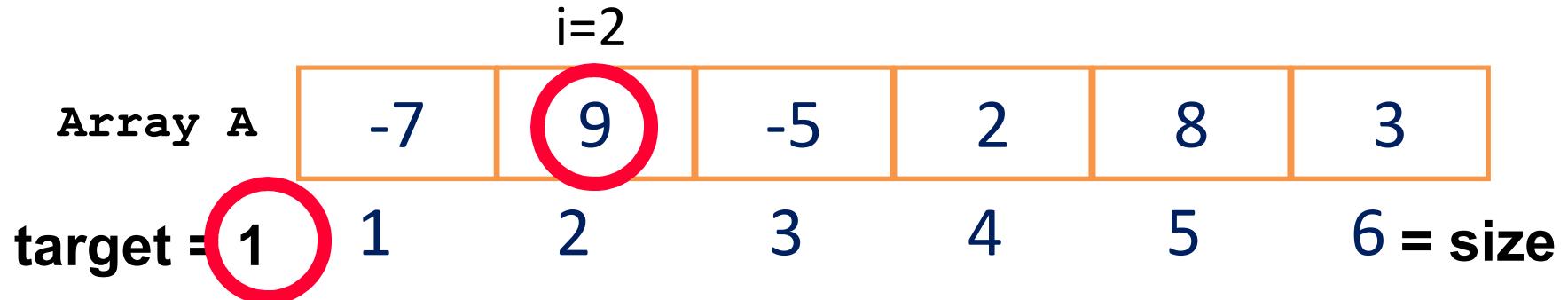
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



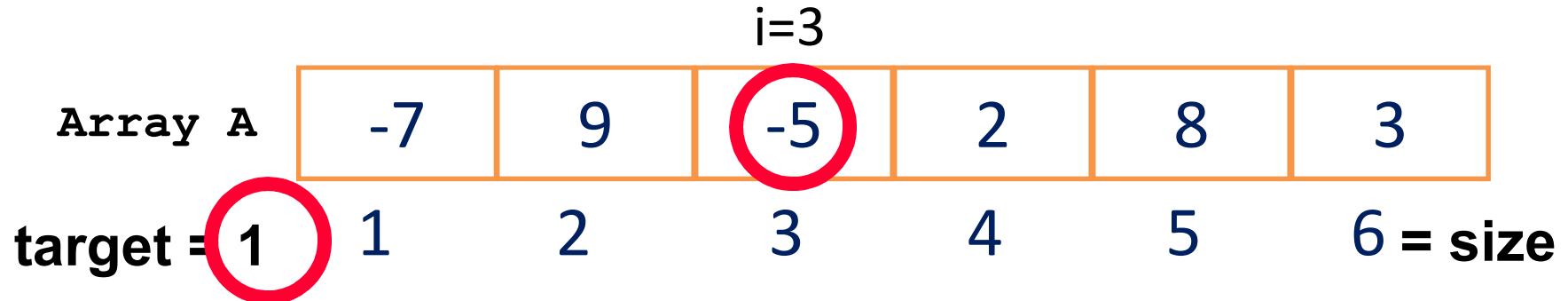
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



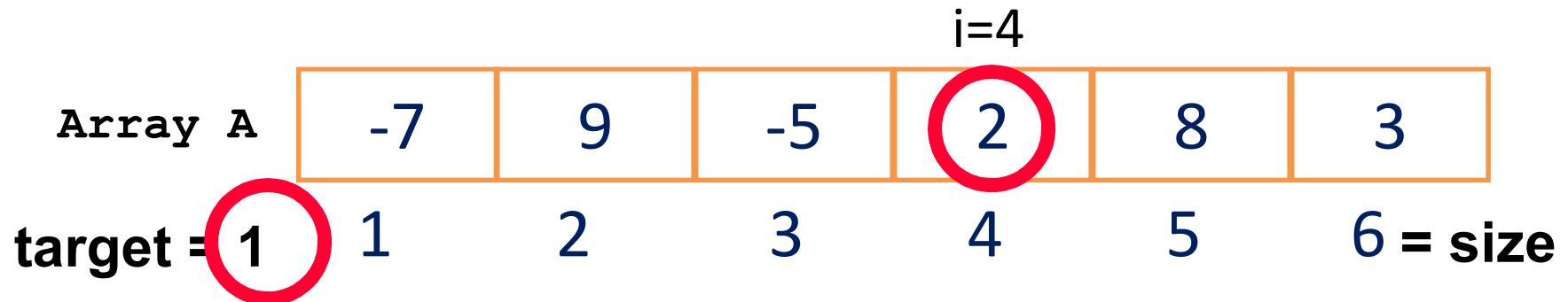
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



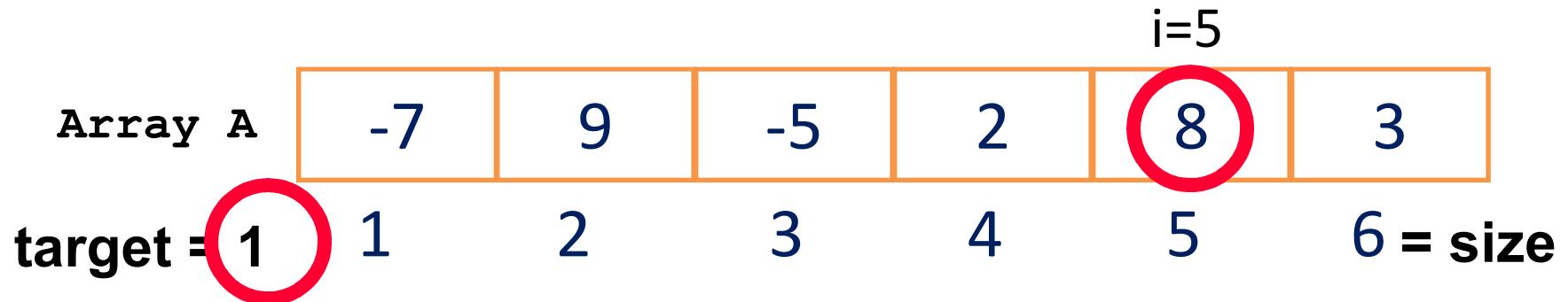
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



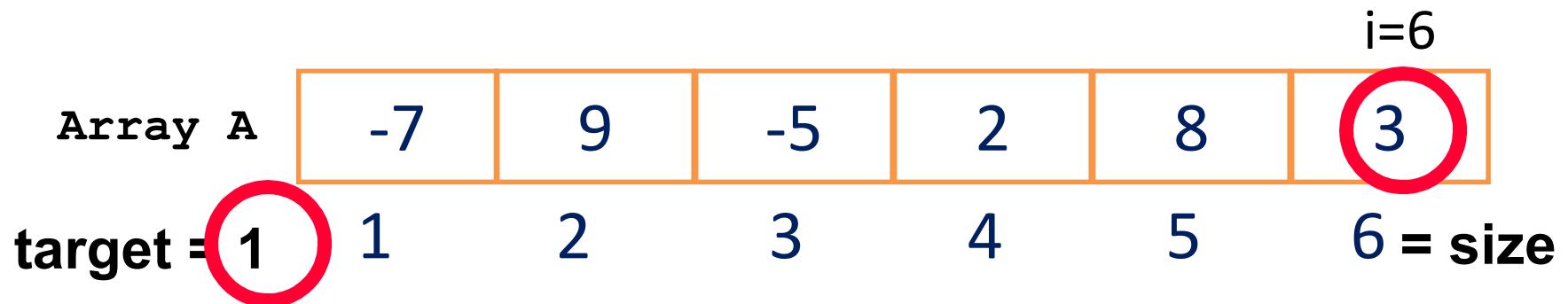
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



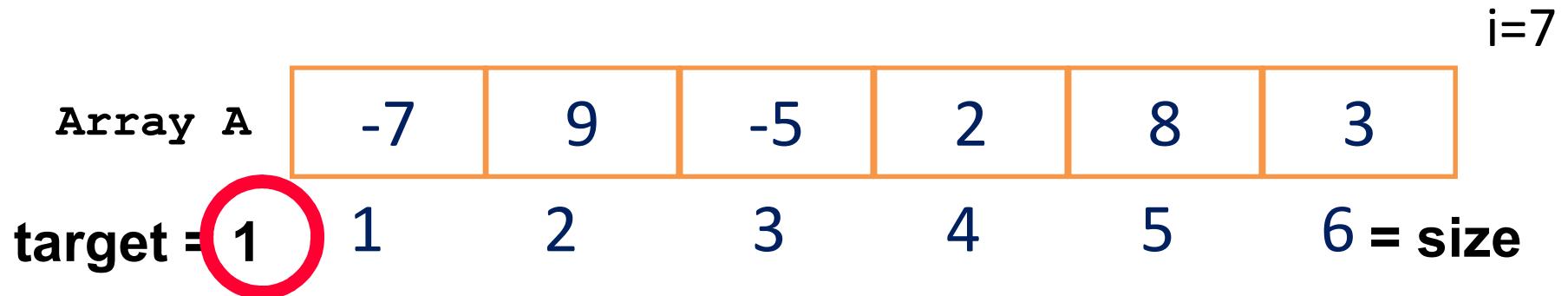
1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{   int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d",i);
    else
        printf("The target is NOT in the list");
}
```



1. Tìm kiếm tuần tự (Linear Search/Sequential search)

```
void linearSearch(int a[], int size, int target)
{    int i;
    for (i = 1; i <= size; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d", i);
else
    printf("The target is NOT in the list");
}
```



The target is NOT in the list

1. Tìm kiếm tuần tự: 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:

(*) $(a[i] == \text{target})$

trong vòng lặp for.

- Nếu phần tử đầu tiên $a[1] = \text{target}$ 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 $\Theta(n)$.

```
void linearSearch(int a[], int n, int target)
{
    int i;
    for (i = 1; i <= n; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d", i);
    else
        printf("The target is NOT in the list");
}
```

1. Tìm kiếm tuần tự: 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, ..., *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 $\Theta(n)$.

```
void linearSearch(int a[], int n, int target)
{
    int i;
    for (i = 1; i <= n; i++)
        if (a[i] == target) break;
    if (i <= size)
        printf("The target is in the list @ index = %d", i);
    else
        printf("The target is NOT in the list");
}
```

Nội dung

1. Tìm kiếm tuần tự
- 2. Tìm kiếm nhị phân**
3. Cây nhị phân tìm kiếm
4. Bảng băm
5. Tìm kiếm xâu mẫu

2. Tìm kiếm nhị phân (Binary Search)

- Đầu vào: Mảng A gồm n phần tử: $A[0], \dots, A[n-1]$ đã được sắp xếp theo **thứ tự tăng dần**; Giá trị **target** có kiểu dữ liệu giống như các phần tử của mảng A .
 - Đầu ra: chỉ số phần tử mảng có giá trị target nếu **target** tìm thấy trong A , -1 nếu **target** không có trong A
 - Thuật toán: lấy phần tử giữa mảng $A[middle]$ với $middle = n/2$
 - If $target < A[middle]$:
 - target không xuất hiện ở nửa bên phải dãy gồm các phần tử $A[middle+1] \dots A[n-1]$, vì vậy có thể bỏ qua dãy nửa bên phải trong quá trình tìm kiếm
 - Lặp lại thao tác này cho nửa bên trái gồm $A[0] \dots A[middle-1]$
 - Else If $target > A[middle]$:
 - target không xuất hiện ở nửa bên trái gồm các phần tử từ $A[0] \dots A[middle]$, vì vậy có thể bỏ qua nửa bên trái
 - Lặp lại thao tác này cho nửa bên phải gồm $A[middle+1] \dots A[n-1]$
 - Else If $target == A[middle]$, then giá trị target tìm thấy ở vị trí $middle$
- If toàn bộ mảng đã được tìm kiếm mà không thấy target, thì kết luận: target ₂₄ không có trong mảng

2. TÌM KIẾM NHỊ PHÂN (Binary Search)

Đầu vào: Mảng A gồm n phần tử: $A[0], \dots, A[n-1]$ đã được sắp xếp theo thứ tự tăng dần; Giá trị **target** có kiểu dữ liệu giống như các phần tử của

`/*Tìm target trong mảng A. Nếu target không có trong mảng A, thì hàm trả về giá trị -1. Ngược lại, hàm trả về vị trí index thỏa mãn $A[index]==target$.`

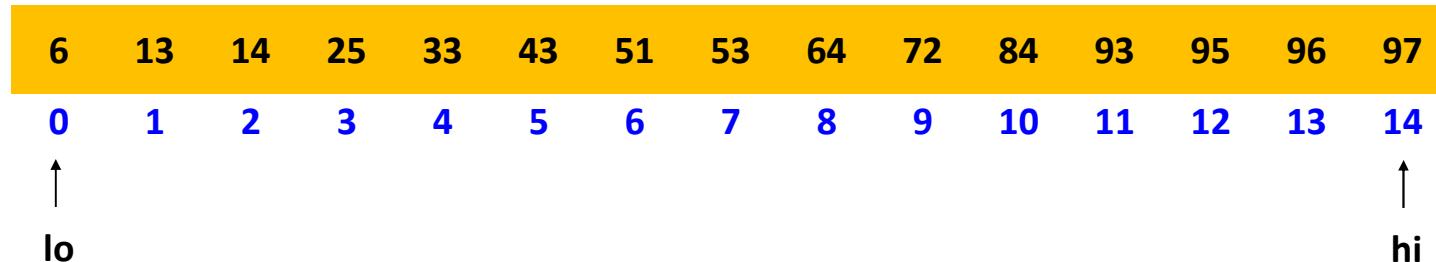
`Điều kiện của mảng: $A[low] \leq A[low+1] \leq \dots \leq A[last]$ */`

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]==target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

2. Tìm kiếm nhị phân (Binary Search)

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, key);
        else
            return binsearch(mid+1, high, A, key);
    }
    else return -1;
}
```

target=33

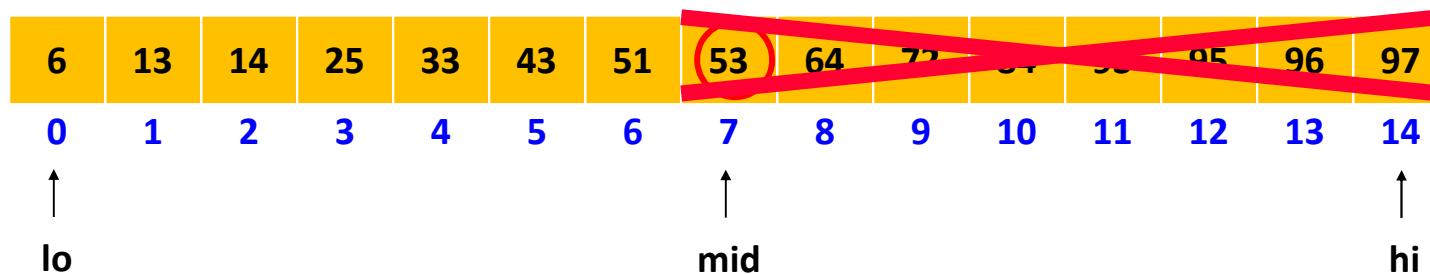


binsearch(0, 14, A, 33);

2. Tìm kiếm nhị phân (Binary Search)

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



binsearch(0, 14, A, 33);

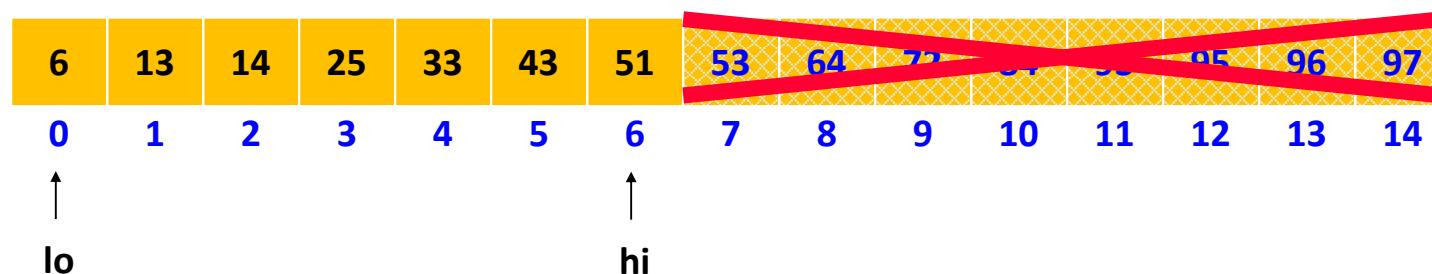
binsearch(0, 6, A, 33);

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

2. Tìm kiếm nhị phân (Binary Search)

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



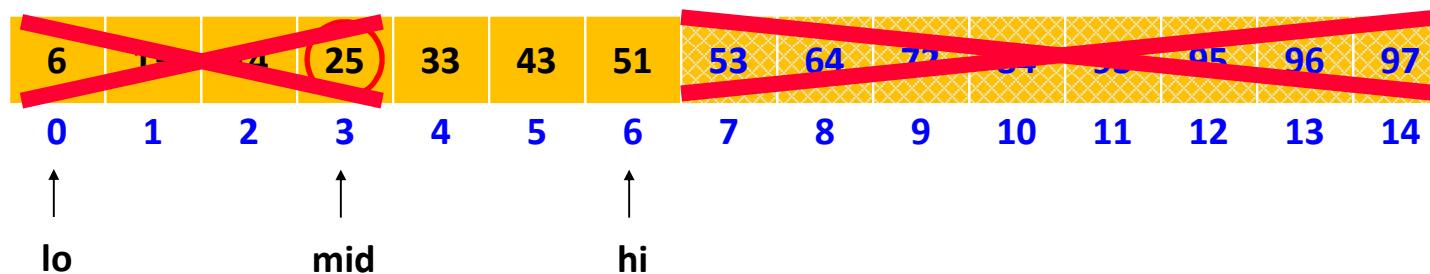
binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

2. Tìm kiếm nhị phân (Binary Search)

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



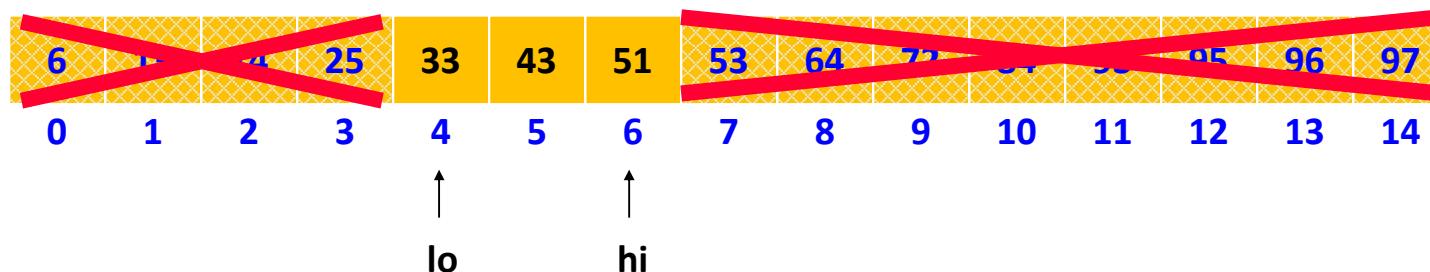
Đoạn cần khảo sát
có độ dài giảm đi một nửa
sau mỗi lần lặp

binsearch(0, 14, A, 33);
binsearch(0, 6, A, 33);
binsearch(4, 6, A, 33);

2. Tìm kiếm nhị phân (Binary Search)

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



binsearch(0, 14, A, 33);

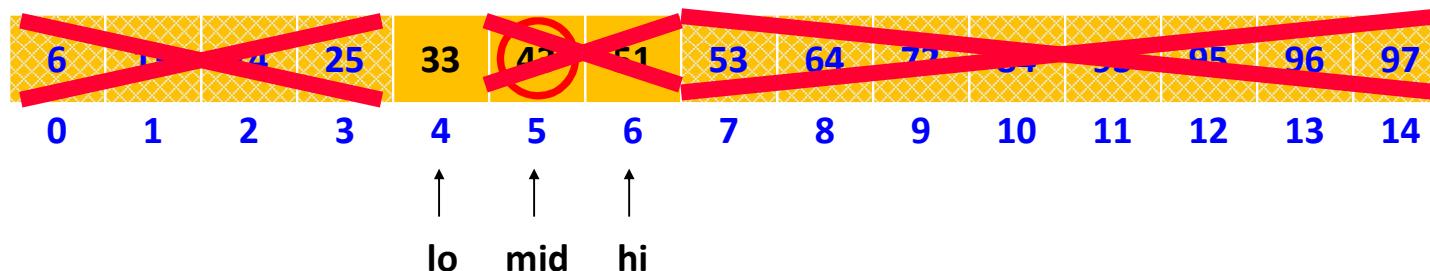
binsearch(0, 6, A, 33);

binsearch(4, 6, A, 33);

2. Tìm kiếm nhị phân (Binary Search)

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

binsearch(4, 6, A, 33);

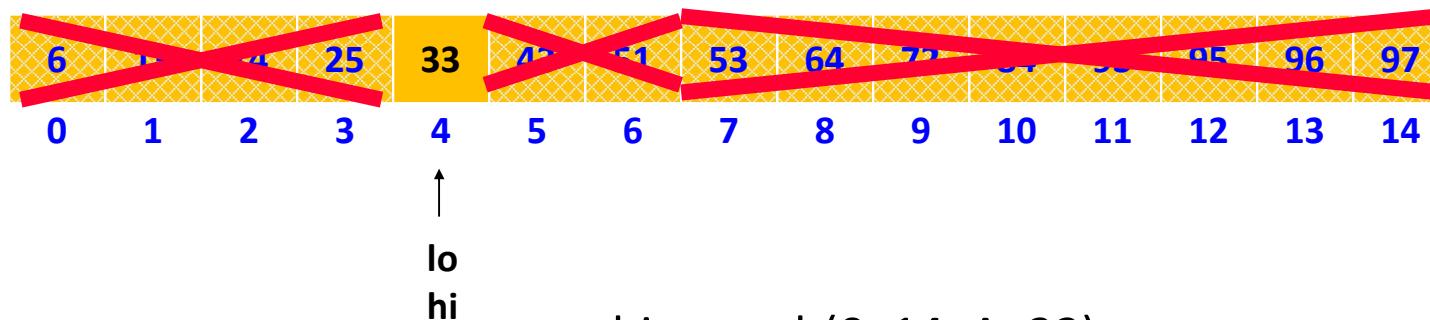
binsearch(4, 4, A, 33);

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

2. Tìm kiếm nhị phân (Binary Search)

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

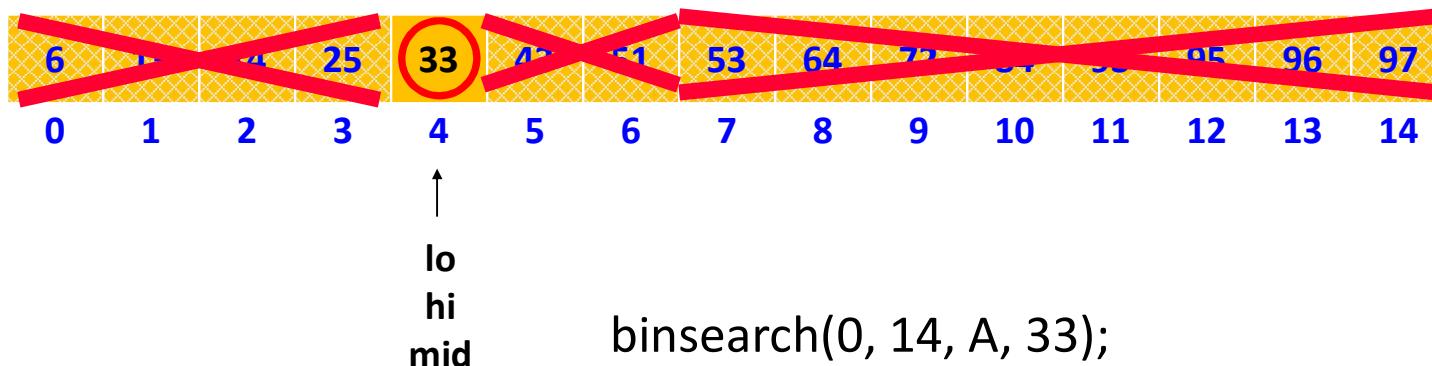
binsearch(4, 6, A, 33);

binsearch(4, 4, A, 33);

2. Tìm kiếm nhị phân (Binary Search)

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33



binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

binsearch(4, 6, A, 33);

binsearch(4, 4, A, 33);

2. TÌM KIẾM NHỊ PHÂN (Binary Search)

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]== target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

target=33
Giá trị 33 xuất hiện @index=4

target=31??

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑
lo
hi
mid

binsearch(0, 14, A, 33);

binsearch(0, 6, A, 33);

binsearch(4, 6, A, 33);

binsearch(4, 4, A, 33);

2. Tìm kiếm nhị phân (Binary Search)

- Đầu vào: Mảng A gồm n phần tử: $A[0], \dots, A[n-1]$ đã được sắp xếp theo thứ tự tăng dần; Giá trị **target** có kiểu dữ liệu giống như các phần tử của mảng A .
- Đầu ra: chỉ số phần tử mảng có giá trị target nếu **target** tìm thấy trong A , -1 nếu **target** không có trong A

```
int binsearch(int low, int high, int A[], int target)
{
    if (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid]==target) return mid;
        else if (target < A[mid])
            return binsearch(low, mid-1, A, target);
        else
            return binsearch(mid+1, high, A, target);
    }
    else return -1;
}
```

→ Để tìm kiếm giá trị target trong mảng A , ta gọi: $\text{binsearch}(0, n-1, A, \text{target})$;
Complexity: $O(\log_2 n)$

Dãy cấp 3: ví dụ minh họa ứng dụng Binary Search

Bộ gồm ba số nguyên (a_1, a_2, a_3) được gọi là một bộ cấp số cộng tăng nếu như:

$$a_2 - a_1 = a_3 - a_2 \text{ và } a_2 - a_1 > 0.$$

Bộ ba (a_1, a_2, a_3) được gọi là đi trước bộ ba (b_1, b_2, b_3) trong thứ tự từ điển nếu như một trong ba điều kiện sau đây được thực hiện:

- 1) $a_1 < b_1$;
- 2) $a_1 = b_1$ và $a_2 < b_2$;
- 3) $a_1 = b_1$, $a_2 = b_2$ và $a_3 < b_3$.

- Dãy số nguyên c_1, c_2, \dots, c_n ($|c_i| < 2^{31}$, $i = 1, 2, \dots, n$) được gọi là **dãy cấp 3** nếu như có thể tìm được ba số hạng của nó để lập thành một bộ cấp số cộng tăng.

Ví dụ: Dãy 3, 1, 5, 2, -7, 0, -1 là một dãy cấp 3, vì nó chứa bộ cấp số cộng tăng, chẳng hạn (1, 3, 5) hay (-7, -1, 5). Bộ ba (-7, -1, 5) là bộ cấp số cộng tăng đầu tiên theo thứ tự từ điển trong số các bộ cấp số cộng tăng của dãy đã cho.

Yêu cầu: Hãy kiểm tra xem dãy số nguyên cho trước có phải là dãy cấp 3 hay không. Nếu câu trả lời là khẳng định hãy đưa ra bộ cấp số cộng tăng đầu tiên trong thứ tự từ điển

Dãy cấp 3: Thuật toán trực tiếp

- Sắp xếp dãy $a[1..n]$ theo thứ tự không giảm
- Duyệt tất cả các bộ ba $a[i], a[j], a[k]$ với $1 \leq i < j < k \leq n$

```
for(i = 1; i < n-1; i++) {  
    for(j = i+1; j < n; j++) {  
        long key = 2*a[j] - a[i];  
        for (k=j+1; k<=n; k++)  
            if (a[k]==key) {  
                printf("YES; bộ 3 đầu tiên %li %li %li",a[i],a[j],a[k]);  
                halt;  
            }  
    }  
}
```

$a[j] - a[i] = a[k] - a[j]$

 $2 * a[j] - a[i] = a[k]$

Thời gian tính: $O(n^3)$

Dãy cấp 3: Thuật toán nhanh hơn

- Sắp xếp dãy $a[1..n]$ theo thứ tự không giảm
- Với mỗi bộ

$$(a[i], a[j]), i=1, \dots, n-2; j=i+1, \dots, n-1,$$

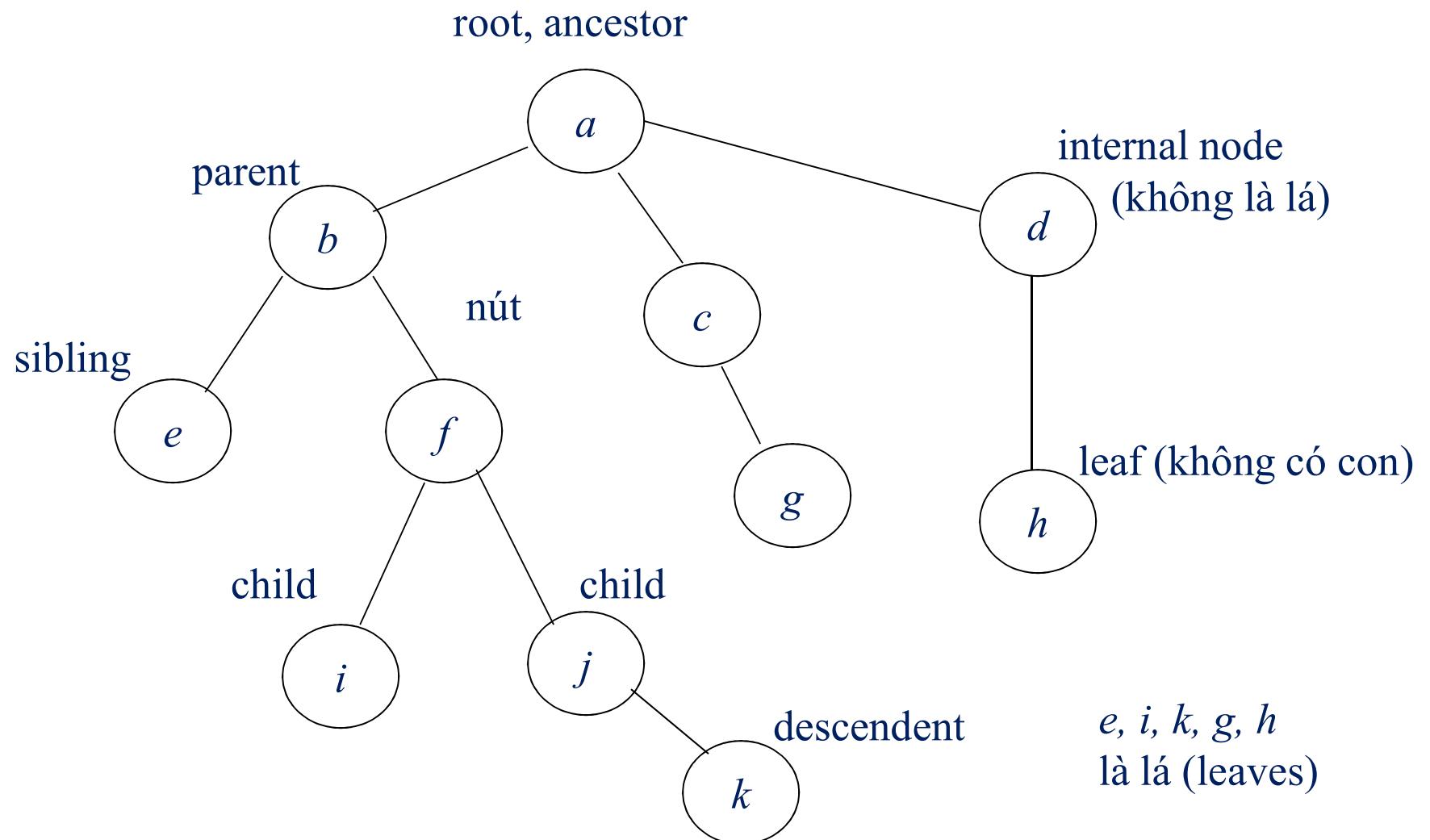
Ta gọi binary search trên dãy gồm các phần tử $a[j+1] \dots a[n]$ với giá trị tìm kiếm target $= 2*a[j]-a[i]$. Nếu tìm thấy target xuất hiện ở vị trí $a[k]$ thì $(a[i], a[j], a[k])$ là một bộ ba thỏa mãn điều kiện đầu bài.

- Bộ ba đầu tiên tìm được sẽ là bộ ba cần đưa ra. Nếu không tìm được bộ ba nào thì dãy đã cho không là dãy cấp 3.
- Thời gian tính: ? Bao nhiêu lần gọi binarysearch ?

Nội dung

1. Tìm kiếm tuần tự
2. Tìm kiếm nhị phân
- 3. Cây nhị phân tìm kiếm**
4. Bảng băm
5. Tìm kiếm xâu mẫu

Các thuật ngữ với cây có gốc



3. Cây nhị phân tìm kiếm

3.1. Định nghĩa

3.2. Biểu diễn cây nhị phân tìm kiếm

3.3. Các phép toán

3. Cây nhị phân tìm kiếm

3.1. Định nghĩa

3.2. Biểu diễn cây nhị phân tìm kiếm

3.3. Các phép toán

Đặt vấn đề

Ta xây dựng cấu trúc để biểu diễn các tập biến động (*dynamic sets*)

- Các phần tử có khoá (*key*) và thông tin đi kèm (*satellite data*)
- Tập động cần hỗ trợ các truy vấn (*queries*) như:
 - **Search(S, k)**: Tìm phần tử có khoá k trong tập S
 - **Minimum(S), Maximum(S)**: Tìm phần tử có khoá nhỏ nhất, lớn nhất trong tập S
 - **Predecessor(S, x), Successor(S, x)**: Tìm phần tử kế cận trước, kế cận sau của x trong tập S

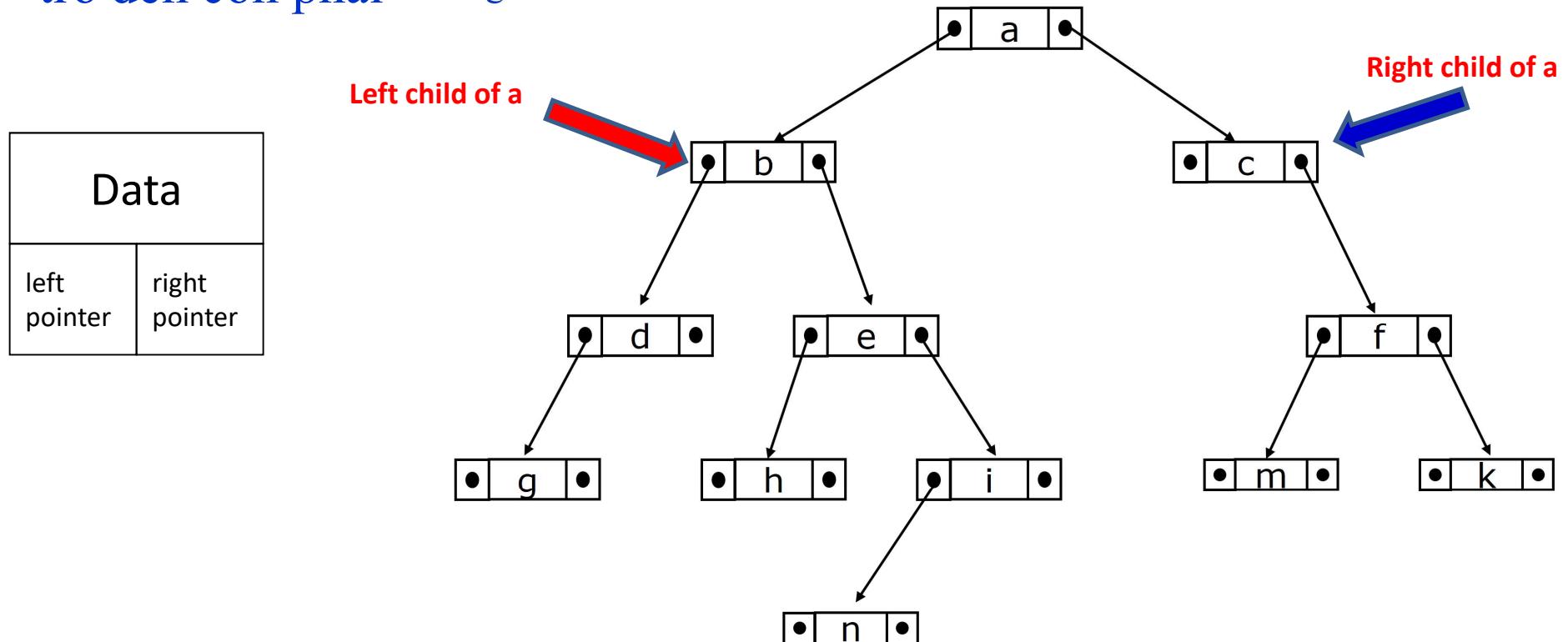
đồng thời cũng hỗ trợ các thao tác biến đổi như:

- **Insert(S, x)**: *Bổ sung (Chèn) x vào S*
- **Delete(S, x)**: *Loại bỏ (Xoá) x khỏi S*

Cây nhị phân tìm kiếm là 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.

Cây nhị phân (Binary tree)

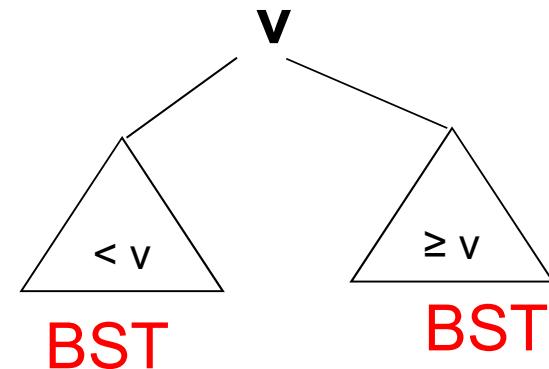
- Cây nhị phân là 1 cây mà không node nào của nó có quá 2 con.
- ➔ Mỗi nút trên cây hoặc: (1) không có con nào, (2) chỉ có con trái, (3) chỉ có con phải, (4) có cả con trái và con phải
- Mỗi nút sẽ bao gồm: dữ liệu (data), con trỏ trỏ đến con trái và con trỏ trỏ đến con phải



Cây nhị phân tìm kiếm (Binary search tree)

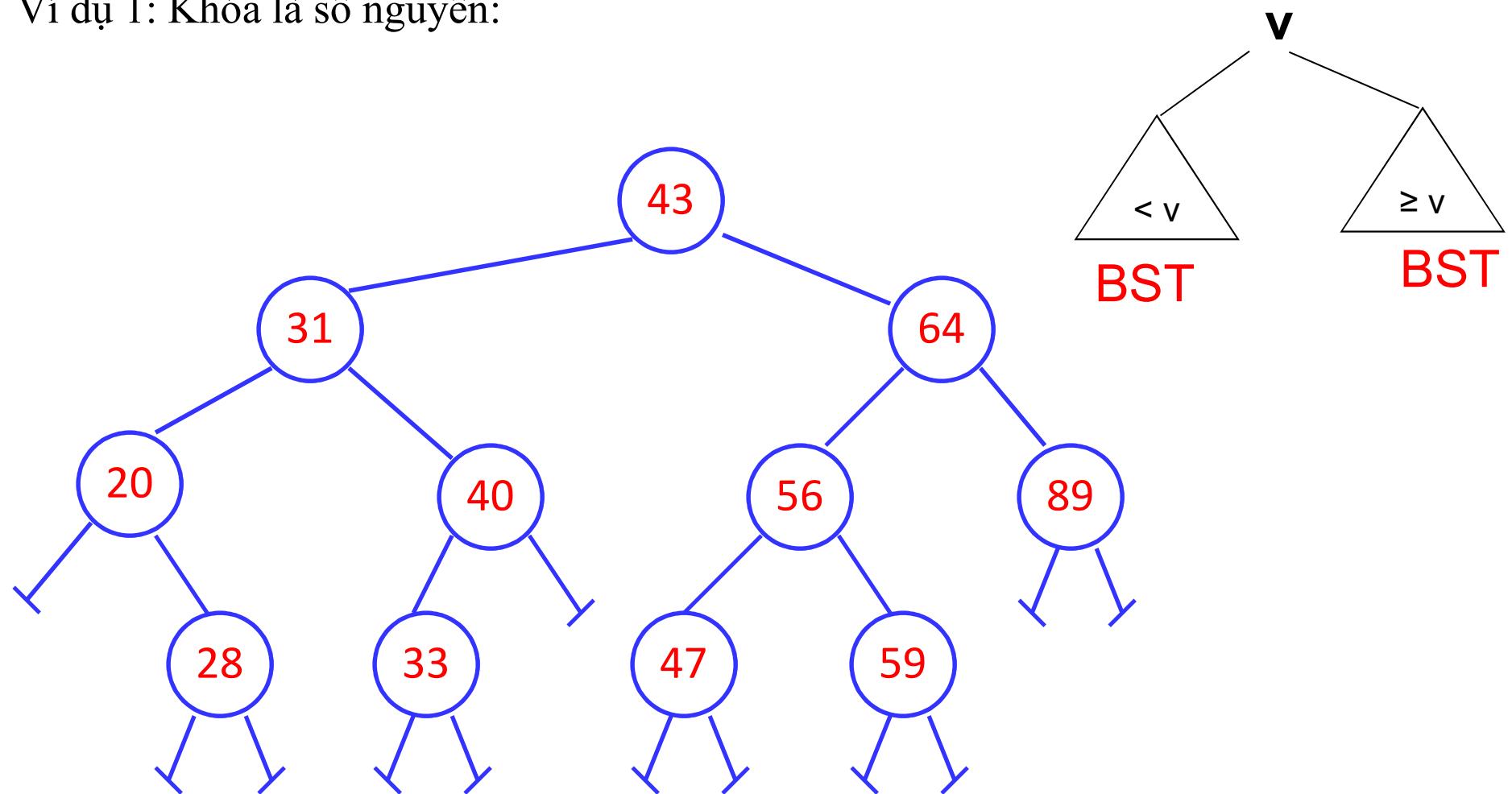
Cây nhị phân tìm kiếm (viết tắt là BST) có các tính chất sau:

- 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 tuỳ 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).
- Mỗi nút có một **khoá duy nhất**
- Tất cả các khoá trong **cây con trái** của nút đều **nhỏ hơn** khoá của nút
- Tất cả các khoá trong **cây con phải** của nút đều **lớn hơn hoặc bằng** khoá của nút



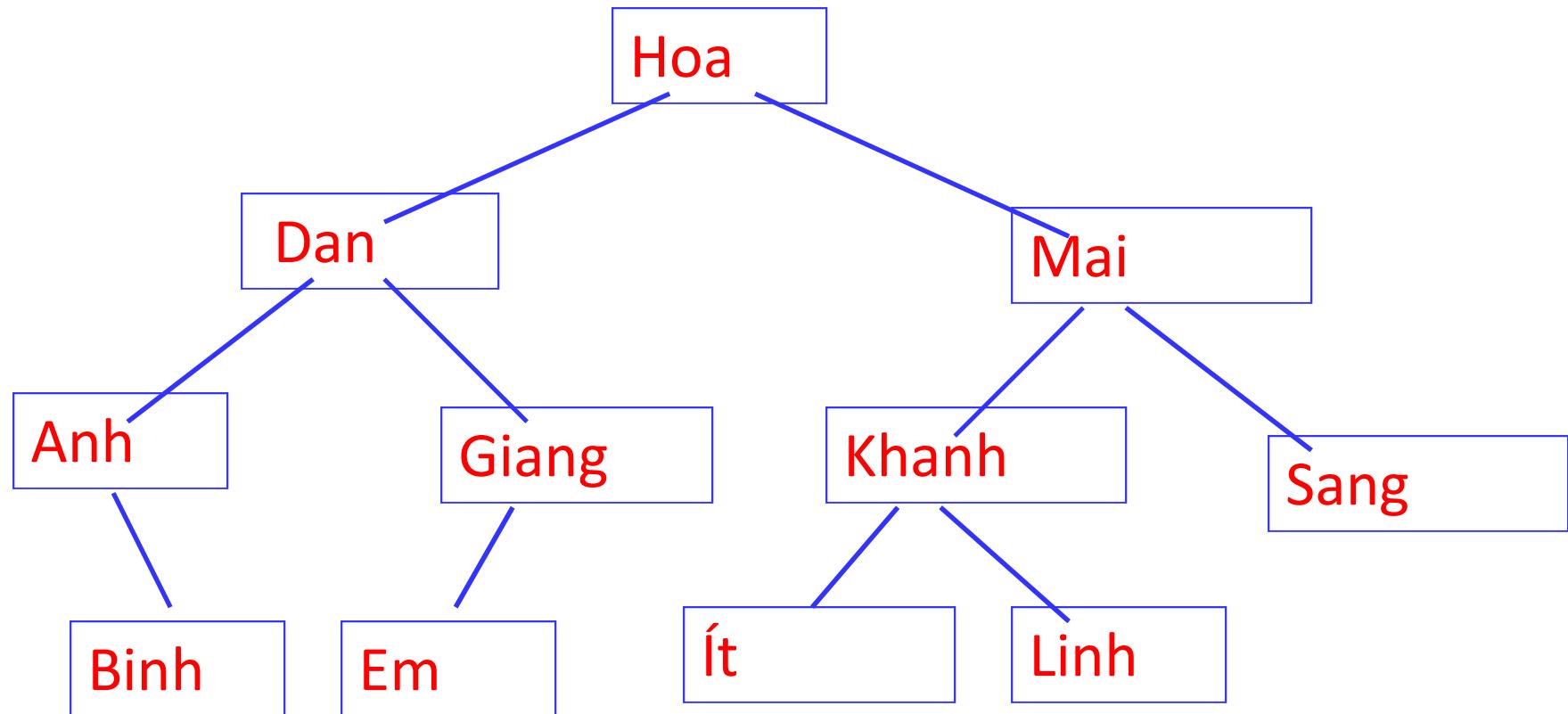
Cây nhị phân tìm kiếm (Binary search tree)

Ví dụ 1: Khóa là số nguyên:



Cây nhị phân tìm kiếm (Binary search tree)

Ví dụ 2: Cây nhị phân tìm kiếm với khóa là xâu kí tự



3. Cây nhị phân tìm kiếm

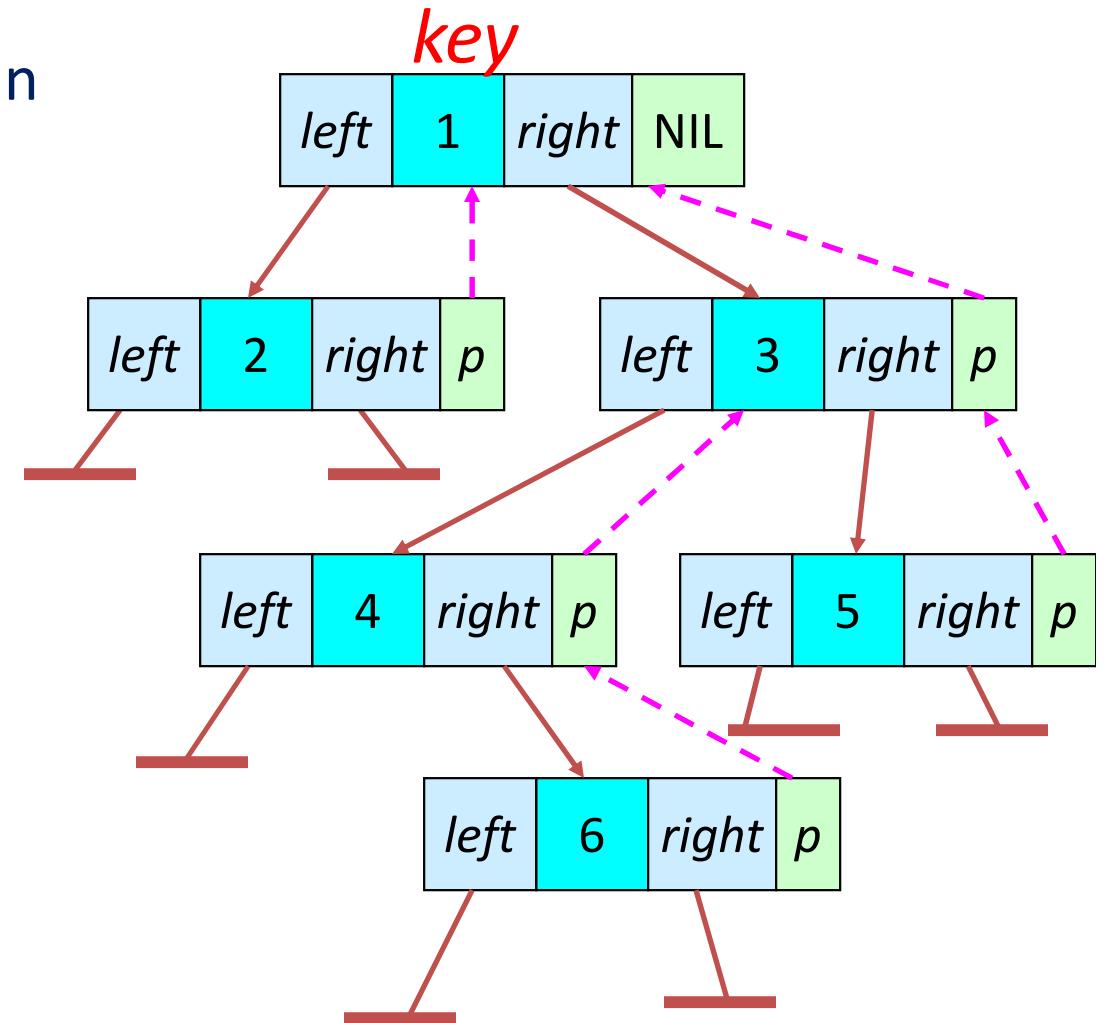
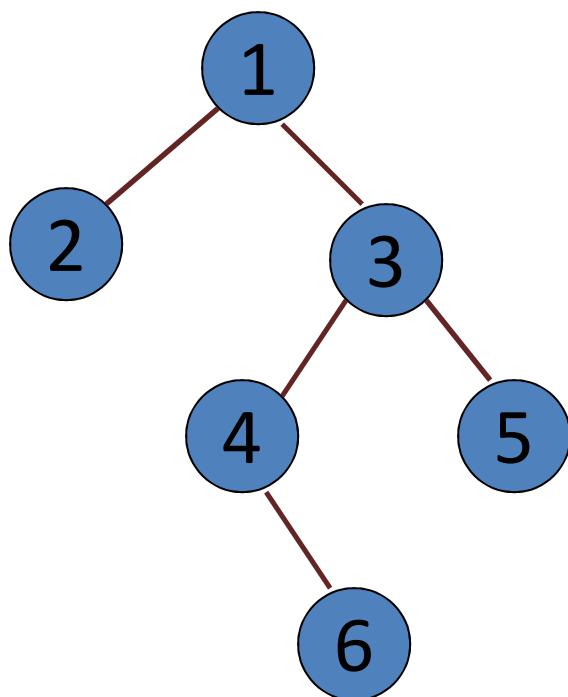
3.1. Định nghĩa

3.2. Biểu diễn cây nhị phân tìm kiếm

3.3. Các phép toán

3.2. Biểu diễn BST

Sử dụng cấu trúc cây nhị phân
(Binary Tree Structure)



3.2. Biểu diễn BST

Biểu diễn liên kết:

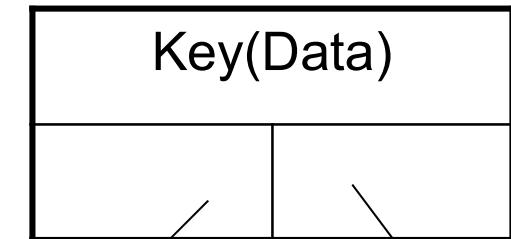
- Mỗi node của cây được biểu diễn như là 1 đối tượng có cùng kiểu dữ liệu.
- Không gian yêu cầu bởi n node cây nhị phân tìm kiếm
 $= n * (\text{không gian yêu cầu của 1 node})$

```
typedef ... elementType; //bất cứ kiểu phần tử nào
```

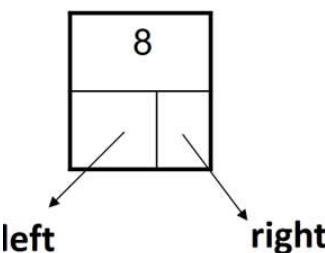
```
typedef struct TreeNode {  
    elementType data;  
    struct TreeNode *left, *right;  
};
```

Ví dụ 1: nếu khóa (dữ liệu) của nút là số nguyên

```
typedef struct TreeNode {  
    int data;  
    struct TreeNode *left, *right;  
};
```



Con trái Con phải



left right

Cây BST

```
typedef ... elementType; //bất cứ kiểu phân tử nào
typedef struct TreeNode {
    elementType data;
    struct TreeNode *left, *right;
};
```

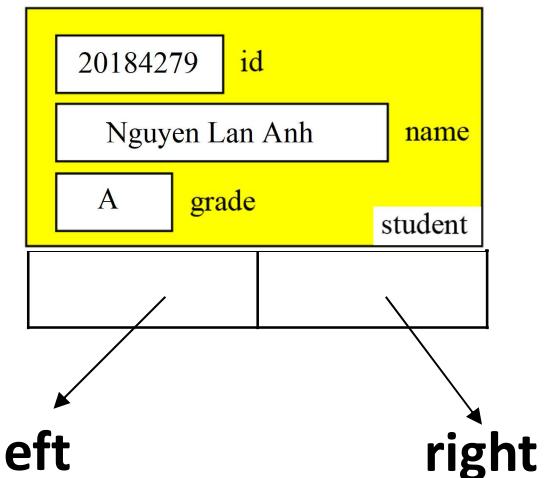
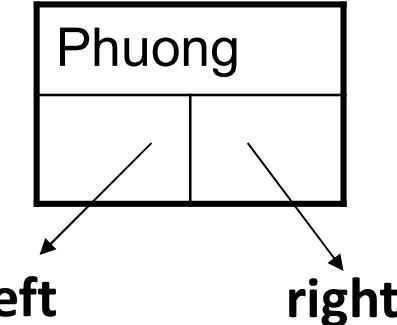
Ví dụ 2: nếu khóa (dữ liệu) mỗi nút là 1 xâu kí tự

```
typedef struct TreeNode {
    char *data;
    struct TreeNode *left, *right;
};
```

Ví dụ 3: nếu dữ liệu mỗi nút là 1 bản ghi lưu trữ 3 thông tin: mã sinh viên, họ tên sinh viên, điểm môn cấu trúc dữ liệu và thuật toán (\rightarrow mã sinh viên sẽ là khóa của nút)

```
typedef struct {
    char *id, *name;
    char grade;
} student;

typedef struct TreeNode {
    student data;
    struct TreeNode *left, *right;
};
```



3. Cây nhị phân tìm kiếm

3.1. Định nghĩa

3.2. Biểu diễn cây nhị phân tìm kiếm

3.3. Các phép toán

3.3. Các phép toán

1. Tìm kiếm (Search)
2. Tìm cực đại, cực tiểu (Findmax, Findmin)
3. Kế cận trước, kế cận sau (Predecessor, Successor)
4. Chèn (Insert)
5. Xóa (Delete)

Để mô tả các phép toán ở trên, ta sẽ sử dụng cây BST có khóa kiểu int:

```
typedef struct TreeNode
{
    int key;
    struct TreeNode *left, *right;
};
```

Tạo một nút mới (create_node)

- **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

```
TreeNode *create_node(int NewKey)
{
    TreeNode *N = (TreeNode*)malloc(sizeof(TreeNode));
    if (N == NULL)
    {
        printf("ERROR: Out of memory\n");
        exit(1);
    }
    else
    {
        N->key = NewKey;
        N->left = NULL;
        N->right = NULL;
    }
    return N;
}
```

Tạo một nút mới (create_node)

```
TreeNode *create_node(int NewKey)
{
    TreeNode *N = (TreeNode*) malloc(sizeof(TreeNode));
    if (N == NULL)
    {
        printf("ERROR: Out of memory\n");
        exit(1);
    }
    else
    {
        N->key = NewKey;
        N->left = NULL;
        N->right = NULL;
    }
    return N;
}
```

3.3. Các phép toán cơ bản: Tìm kiếm (Search)

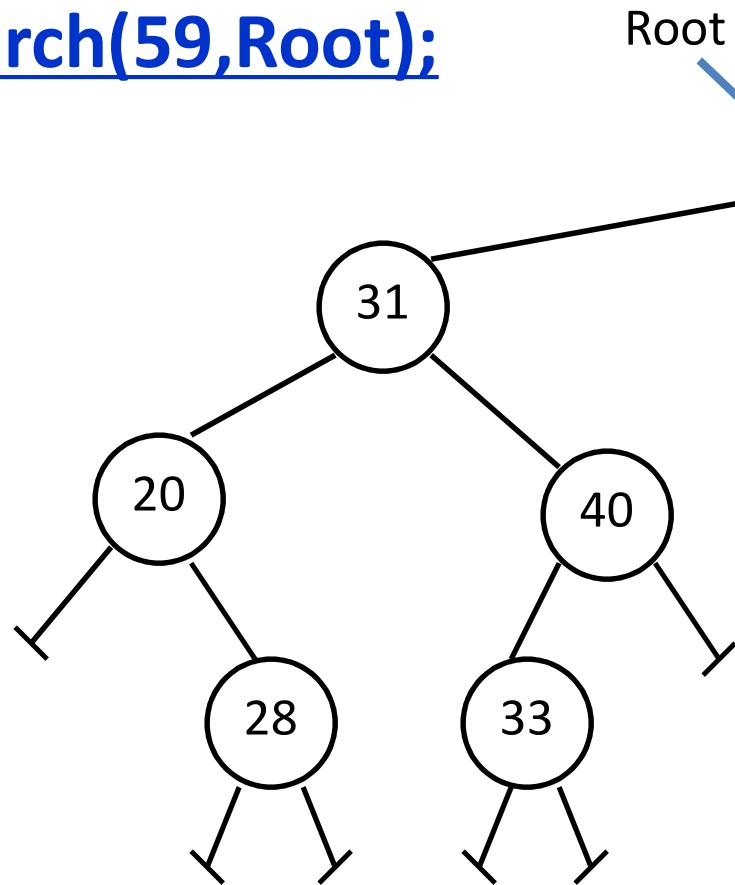
- Nếu khoá cần tìm **nhỏ hơn** khoá của nút hiện tại thì tiếp tục tìm kiếm ở **cây con trái**.
- Trái lại, nếu khoá cần tìm là **lớn hơn** khoá của nút hiện tại, thì tiếp tục tìm kiếm ở **cây con phải**.
- Kết quả cần đưa ra:
 - nếu tìm thấy (nghĩa là khoá cần tìm là **bằng** khoá của nút hiện tại),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

```
TreeNode* Search(int target,TreeNode* Root) {  
    if (Root == NULL) return NULL; // không tìm thấy  
    else if (target == Root->key) /* tìm thấy target */  
        return Root;  
    else if (target < Root->key)  
        return Search(target,Root->left);/*tiếp tục tìm kiếm ở  
cây con trái*/  
    else  
        return Search(target,Root->right); /*tiếp tục tìm kiếm ở  
cây con phải */  
}
```

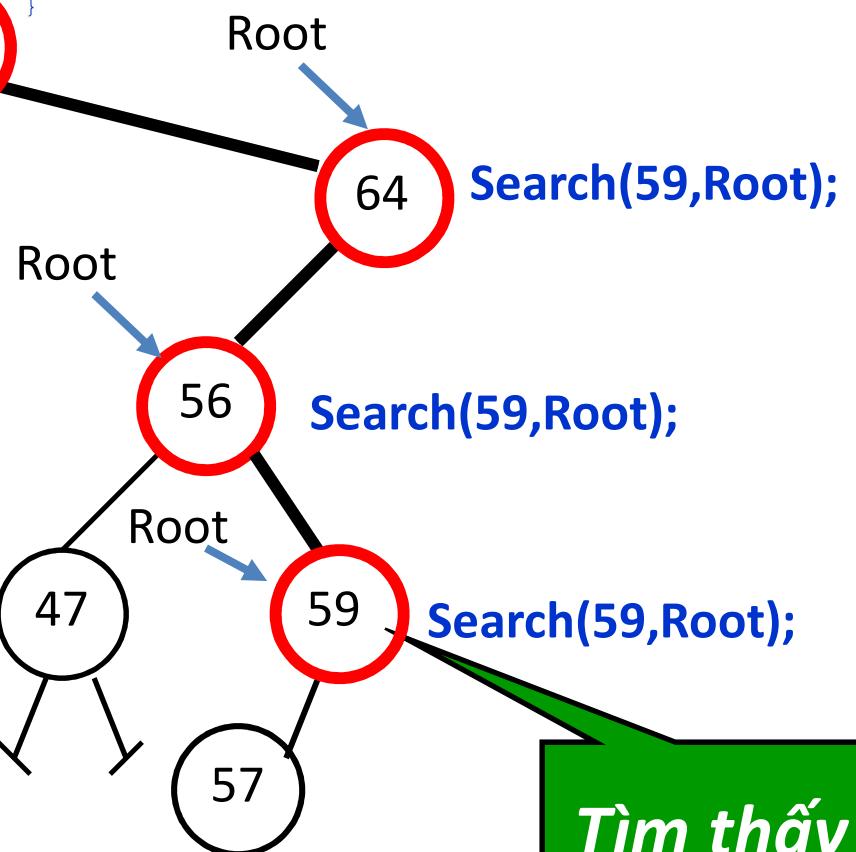
Thời gian tính: $O(h)$,
trong đó h là độ cao của BST

Ví dụ 1: tìm key 59

Search(59,Root);

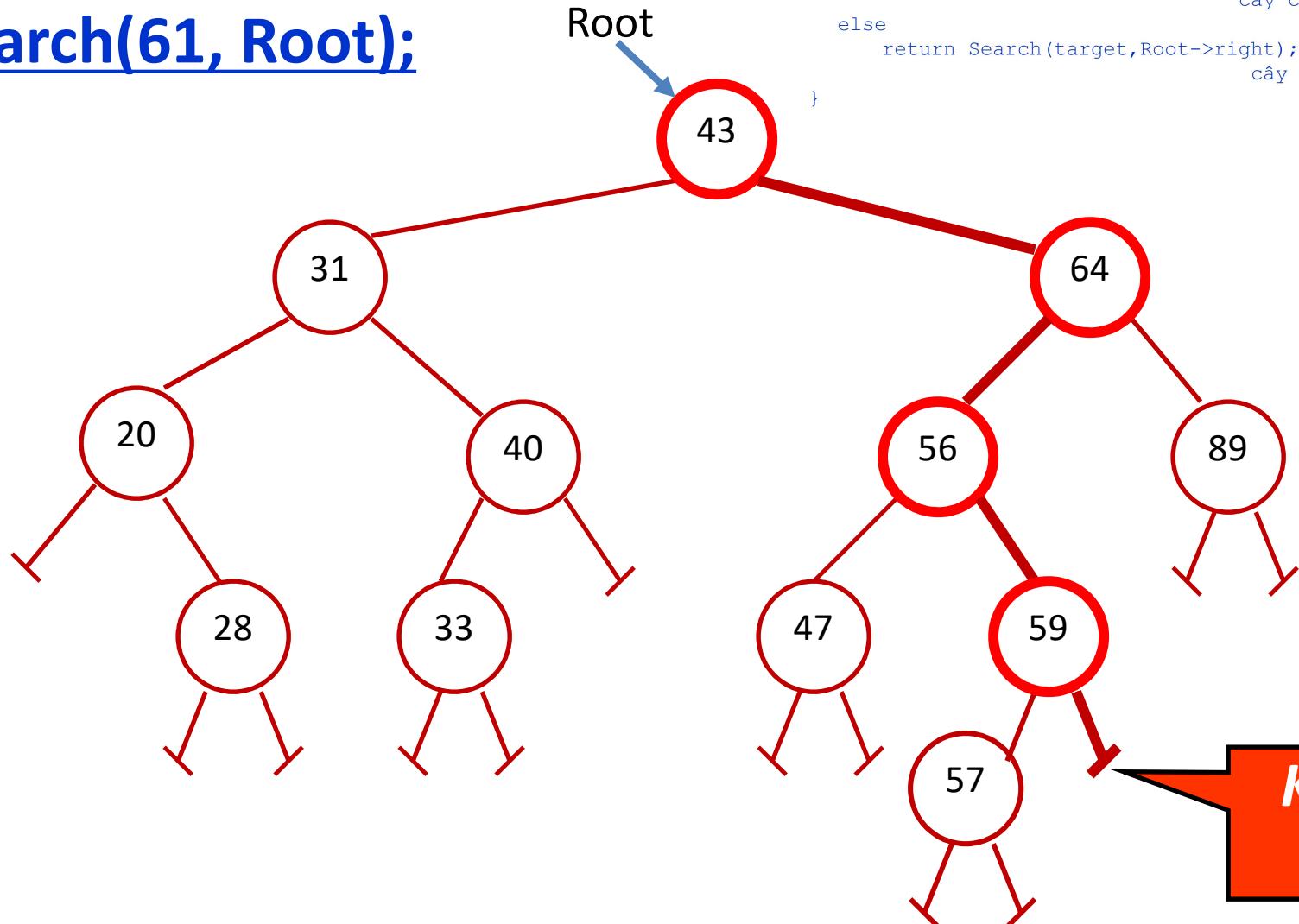


```
TreeNode* Search(int target,TreeNode* Root){  
    if (Root == NULL) return NULL; // không tìm thấy  
    else if (target == Root->key) /* tìm thấy target */  
        return Root;  
    else if (target < Root->key)  
        return Search(target,Root->left);/*tiếp tục tìm kiếm ở  
cây con trái*/  
    else  
        return Search(target,Root->right); /*tiếp tục tìm kiếm ở  
cây con phải */  
}
```



Ví dụ 2: tìm key 61

Search(61, Root);



```
TreeNode* Search(int target,TreeNode* Root){  
    if (Root == NULL) return NULL; // không tìm thấy  
    else if (target == Root->key) /* tìm thấy target */  
        return Root;  
    else if (target < Root->key)  
        return Search(target,Root->left);/*tiếp tục tìm kiếm ở  
cây con trái*/  
    else  
        return Search(target,Root->right); /*tiếp tục tìm kiếm ở  
cây con phải */  
}
```

**Không
thấy**

3.3. Các phép toán cơ bản: findMin, findMax

- Để tìm phần tử nhỏ nhất trên BST trả bởi root, ta đi theo con trái cho đến khi gặp NULL

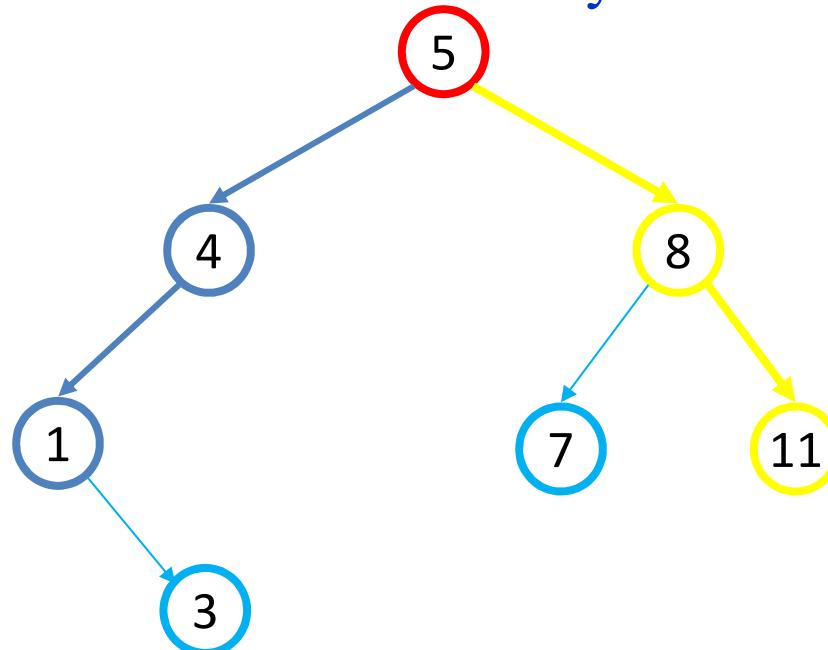
`TreeNode* find_min (TreeNode *root)`

hàm trả về nút có khoá nhỏ nhất trên cây BST có gốc trả bởi root

- Để tìm phần tử lớn nhất trên BST trả bởi root, ta đi theo con phải cho đến khi gặp NULL

`TreeNode* find_max (TreeNode *root)`

hàm trả về nút có khoá lớn nhất trên cây BST có gốc trả bởi root



3.3. Các phép toán cơ bản: findMin, findMax

```
TreeNode* find_min(TreeNode* root)
{ /* luôn đi theo con trái */
    if (root == NULL) return (NULL);
    else
        if (root->left == NULL) return (root);
        else return (find_min(root->left));
}
```

```
TreeNode* find_max(TreeNode* root)
/* luôn đi theo con phải */
{
    if (root != NULL)
        while (root->right != NULL) root = root->right;
    return root;
}
```

3.3. Các phép toán cơ bản

Duyệt cây theo chiều sâu:

- Preorder (Duyệt theo thứ tự trước)

```
void printPreorder(TreeNode *Root)
```

- Inorder (Duyệt theo thứ tự giữa)

```
void printInorder(TreeNode *Root)
```

- Postorder (Duyệt theo thứ tự sau)

```
void printPostorder(TreeNode *Root)
```

Duyệt theo thứ tự trước (Preorder)

```
void printPreorder(TreeNode *Root)
{
    visit the node
    traverse left sub-tree
    traverse right sub-tree
}
```



```
void printPreorder(TreeNode *Root)
{
    if (Root != NULL) {
        printf(" %d", Root->key);
        printPreorder(Root->left);
        printPreorder(Root->right);
    }
}
```

Duyệt theo thứ tự sau (Postorder)

```
void printPostorder(TreeNode *Root)
{
    traverse left sub-tree
    traverse right sub-tree
    visit the node
}
```



đầu tiên,
trở tới nút gốc

```
void printPostorder(TreeNode *Root)
{
    if (Root != NULL) {
        printPostorder(Root->left) ;
        printPostorder(Root->right) ;
        printf(" %d", Root->key) ;
    }
}
```

Duyệt theo thứ tự giữa (Inorder)

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

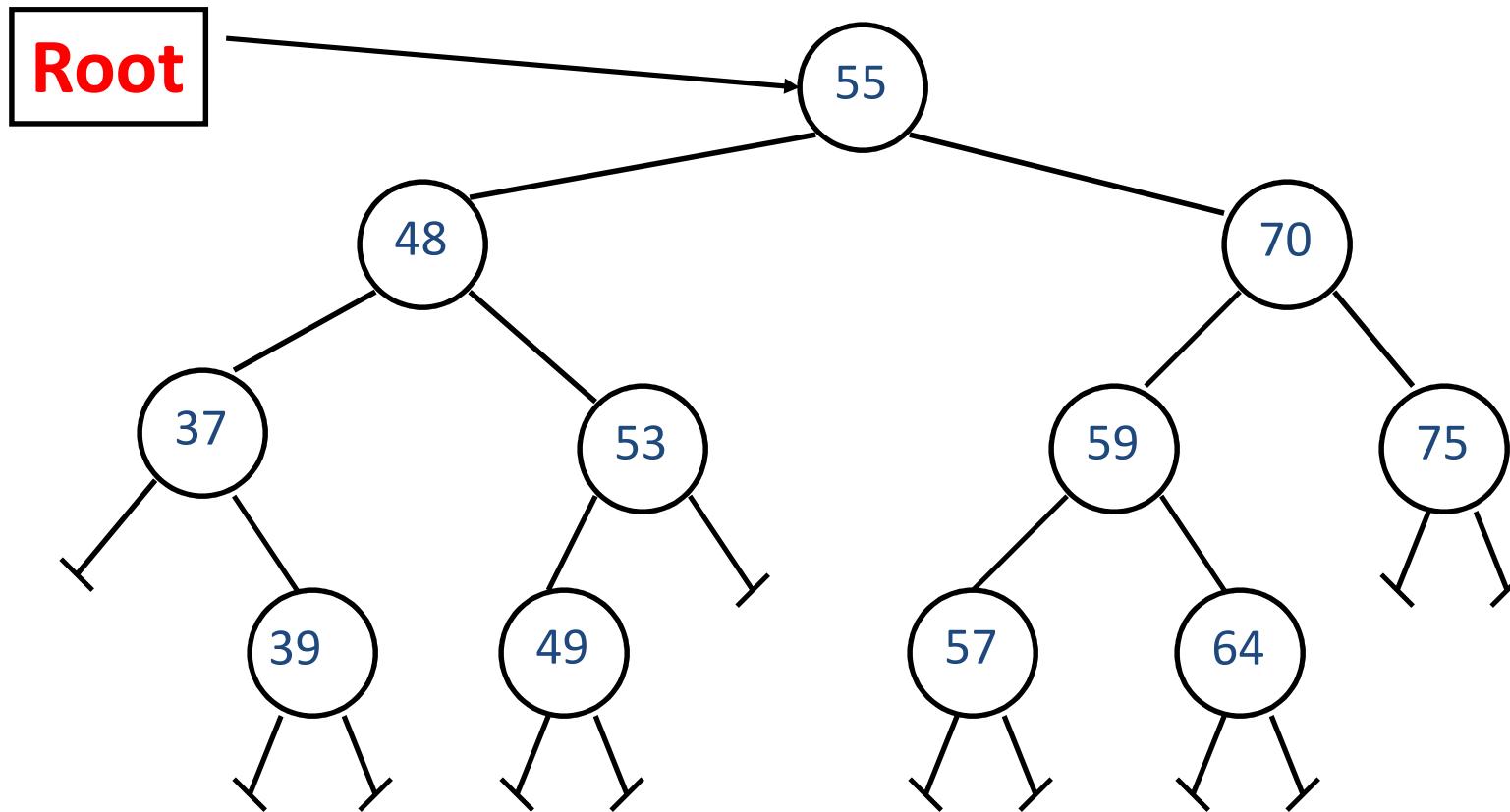
```
void printInorder(TreeNode *Root)
{
    traverse left sub-tree
    visit the node
    traverse right sub-tree
}
```



đầu tiên,
trả tới nút gốc

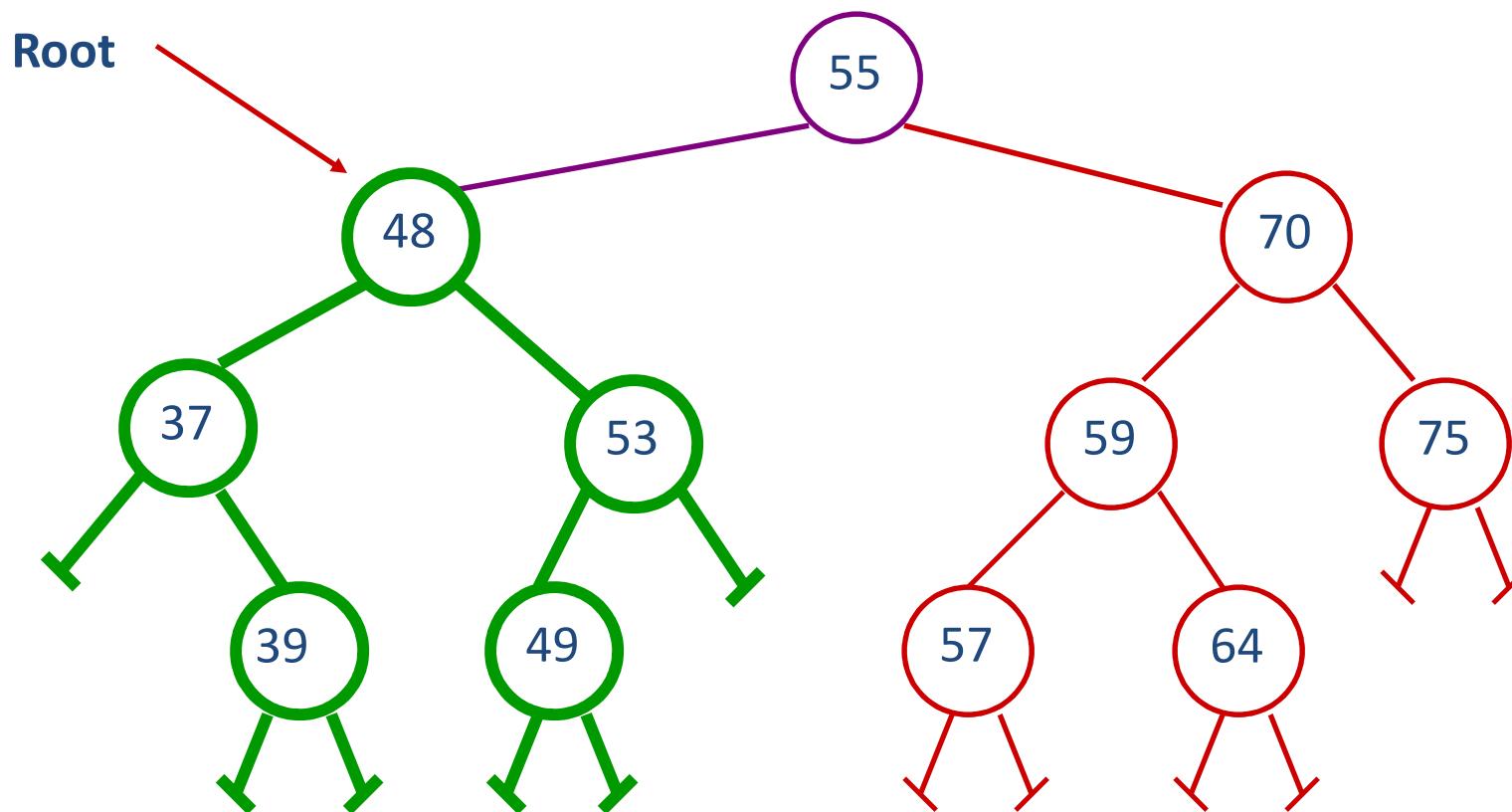
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Duyệt theo thứ tự giữa (Inorder)



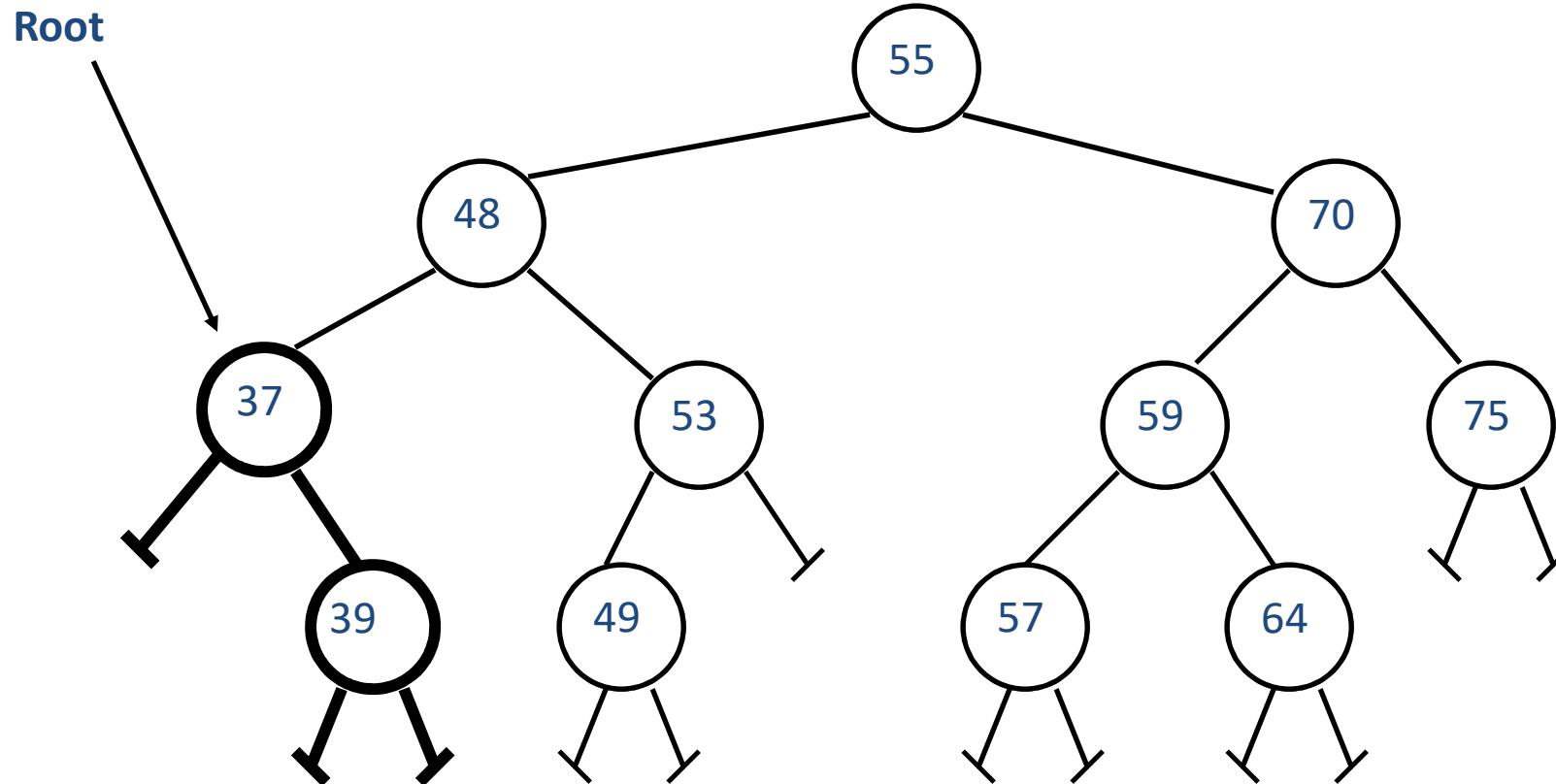
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Duyệt theo thứ tự giữa (Inorder)



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

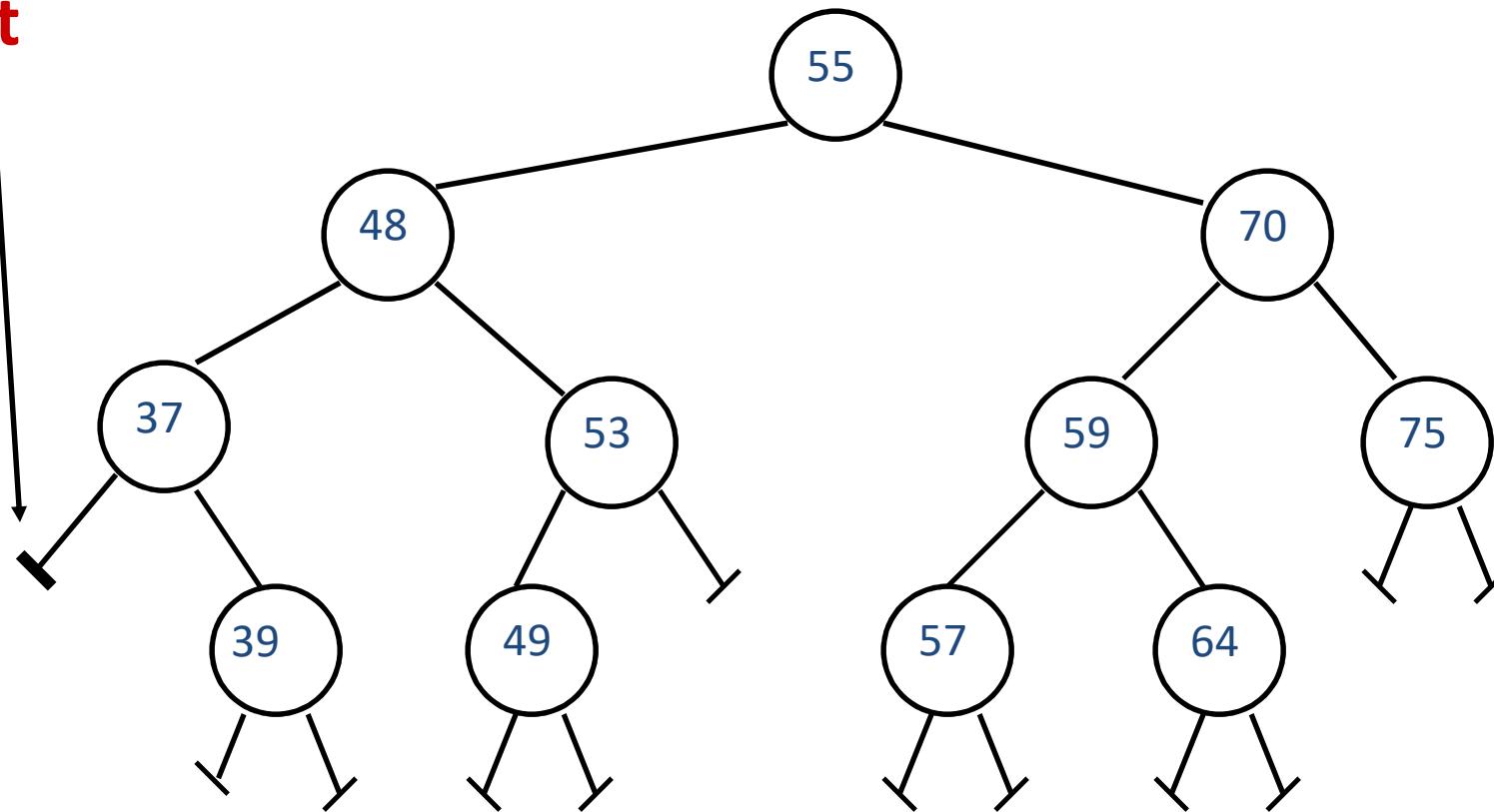
Duyệt theo thứ tự giữa (Inorder)



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Duyệt theo thứ tự giữa (Inorder)

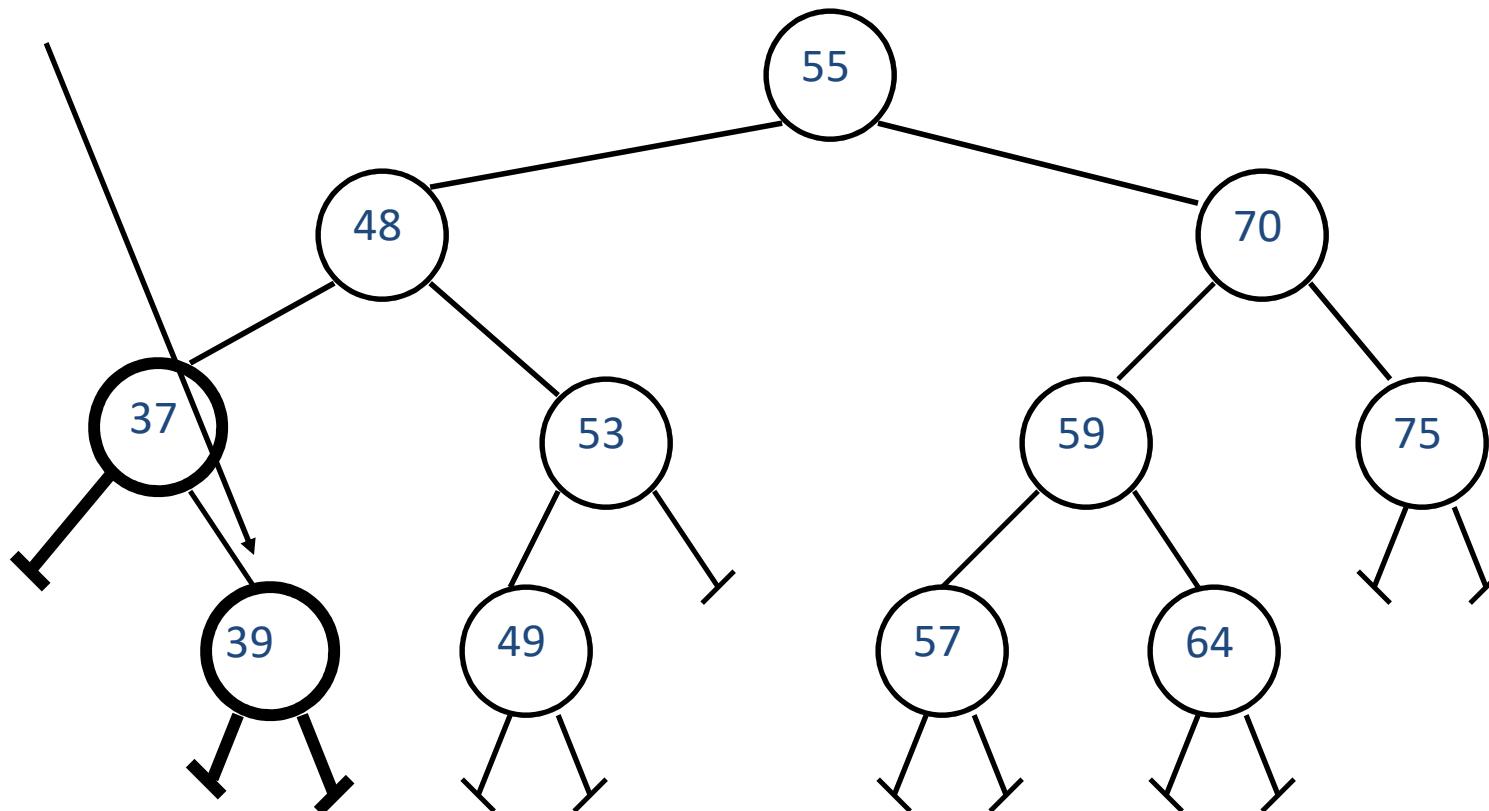
Root



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

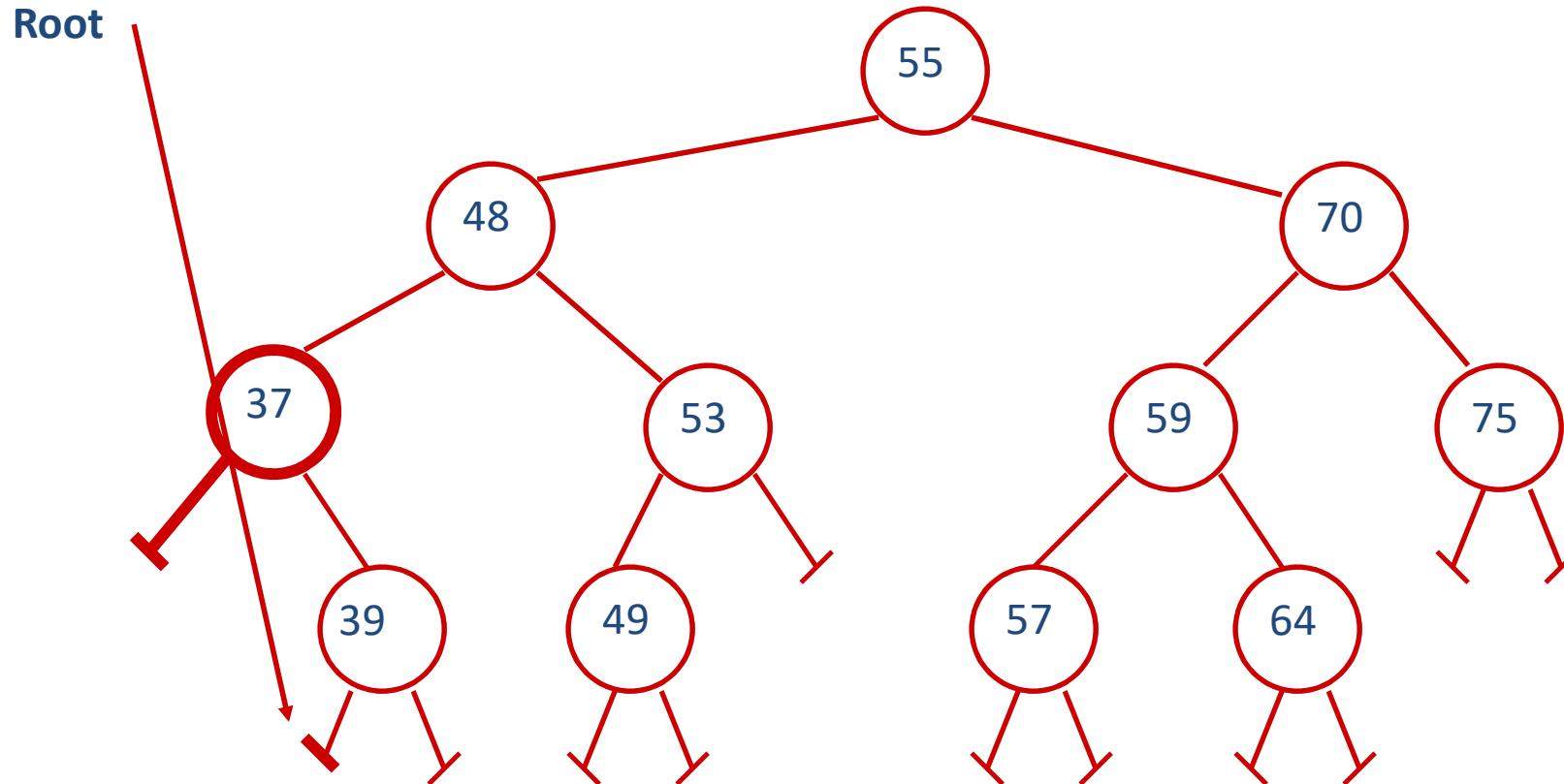
Inorder: 37,

Root



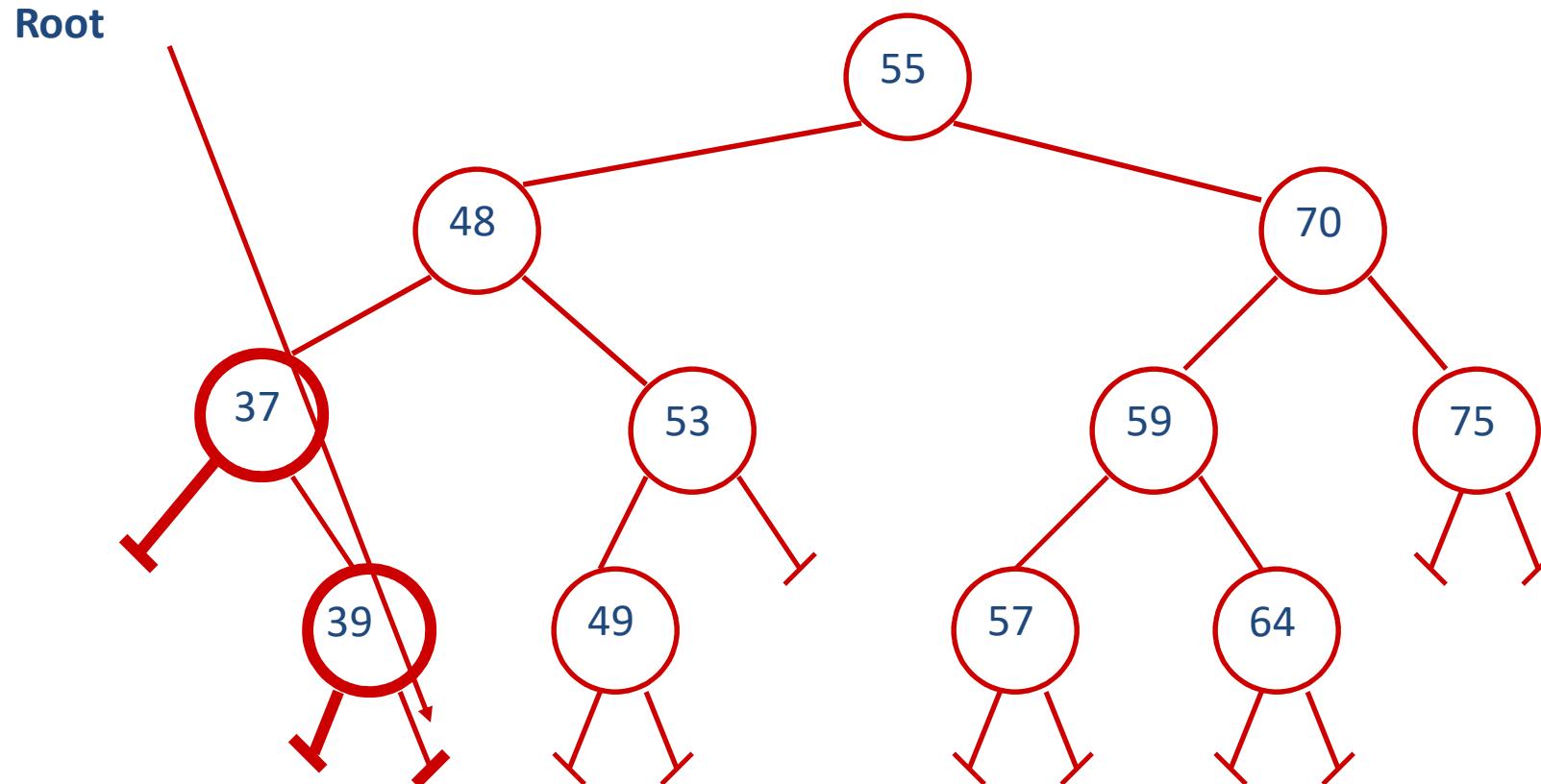
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37,



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

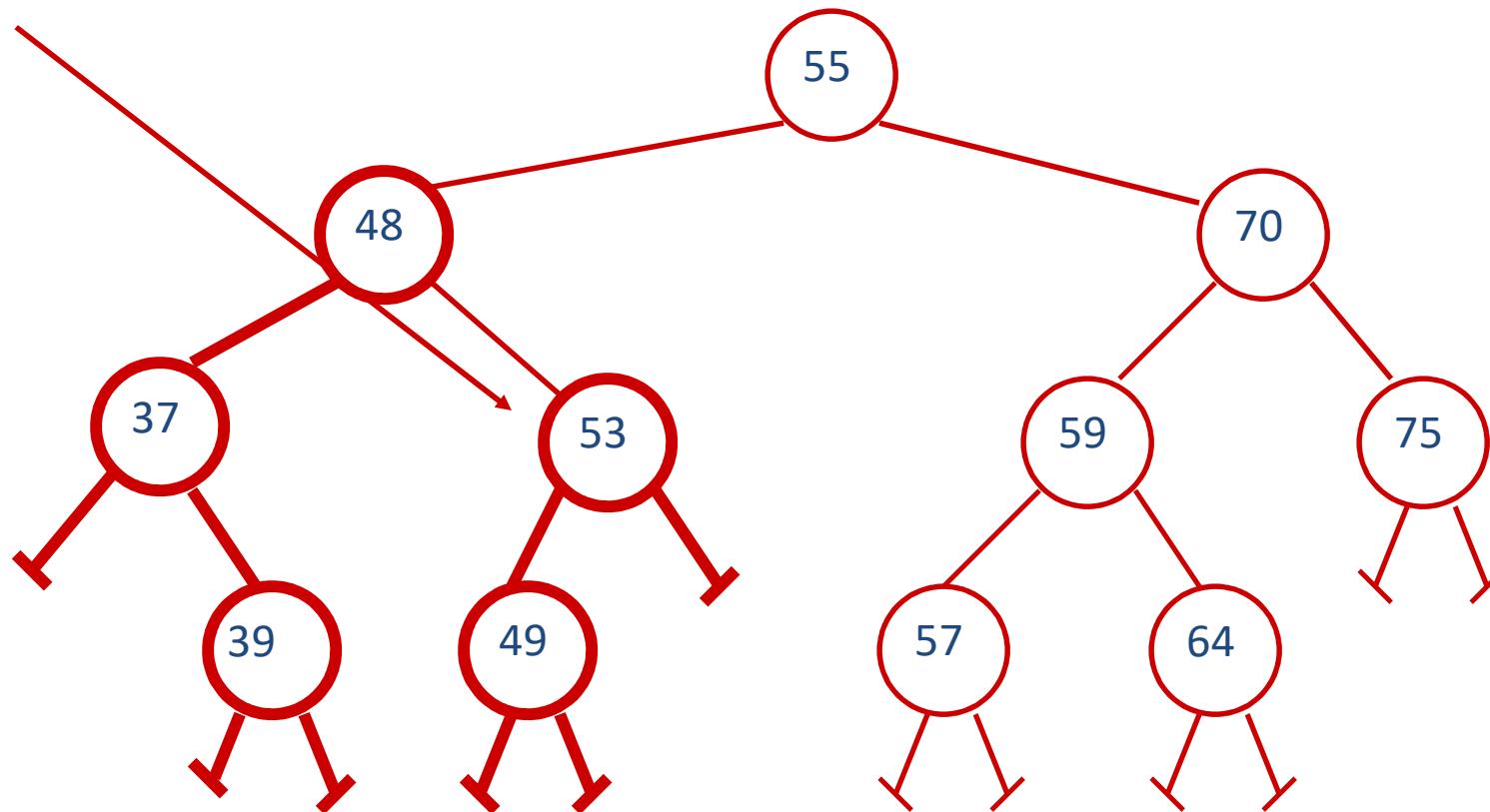
Inorder: 37,39



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48

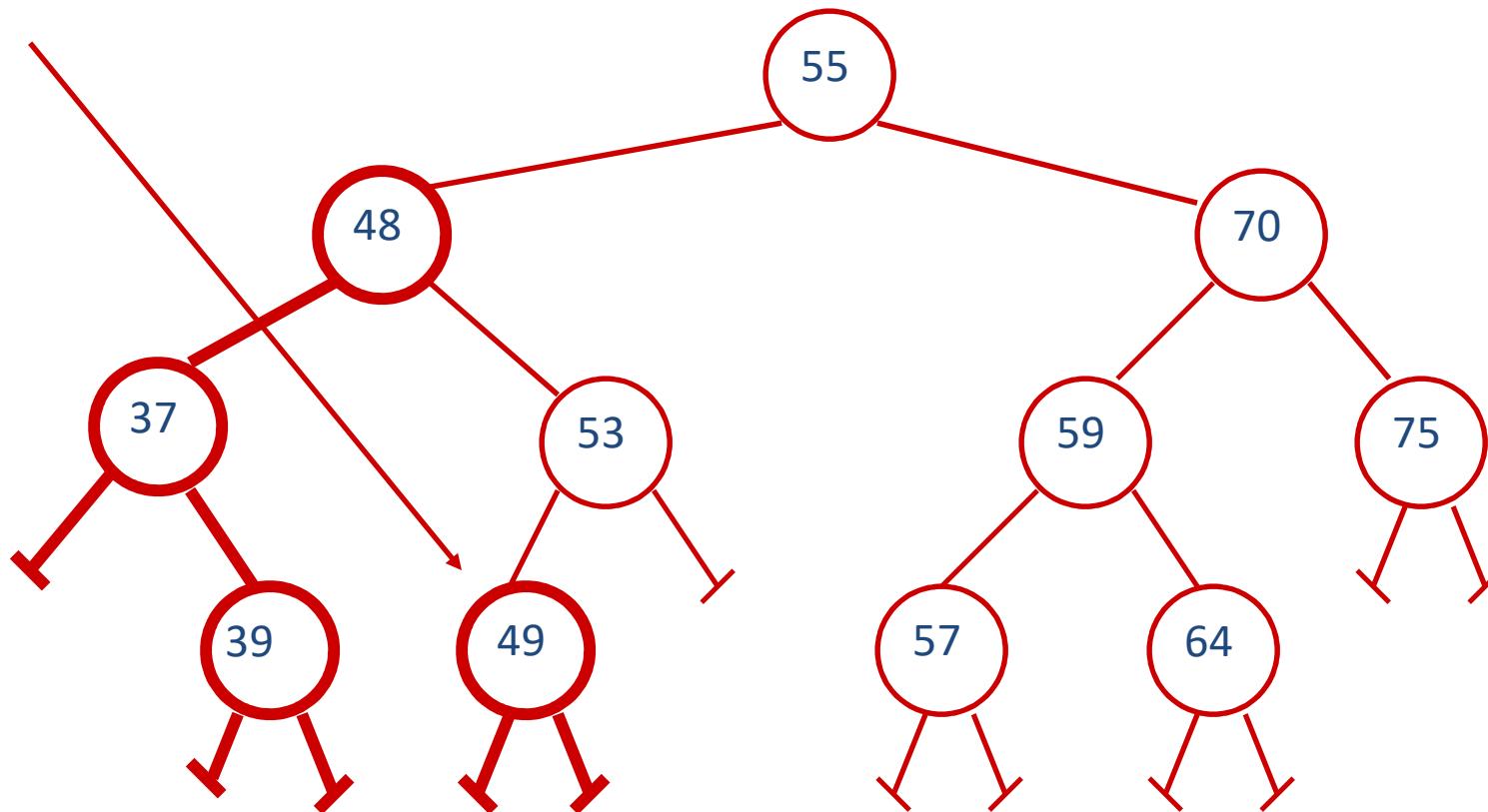
Root



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

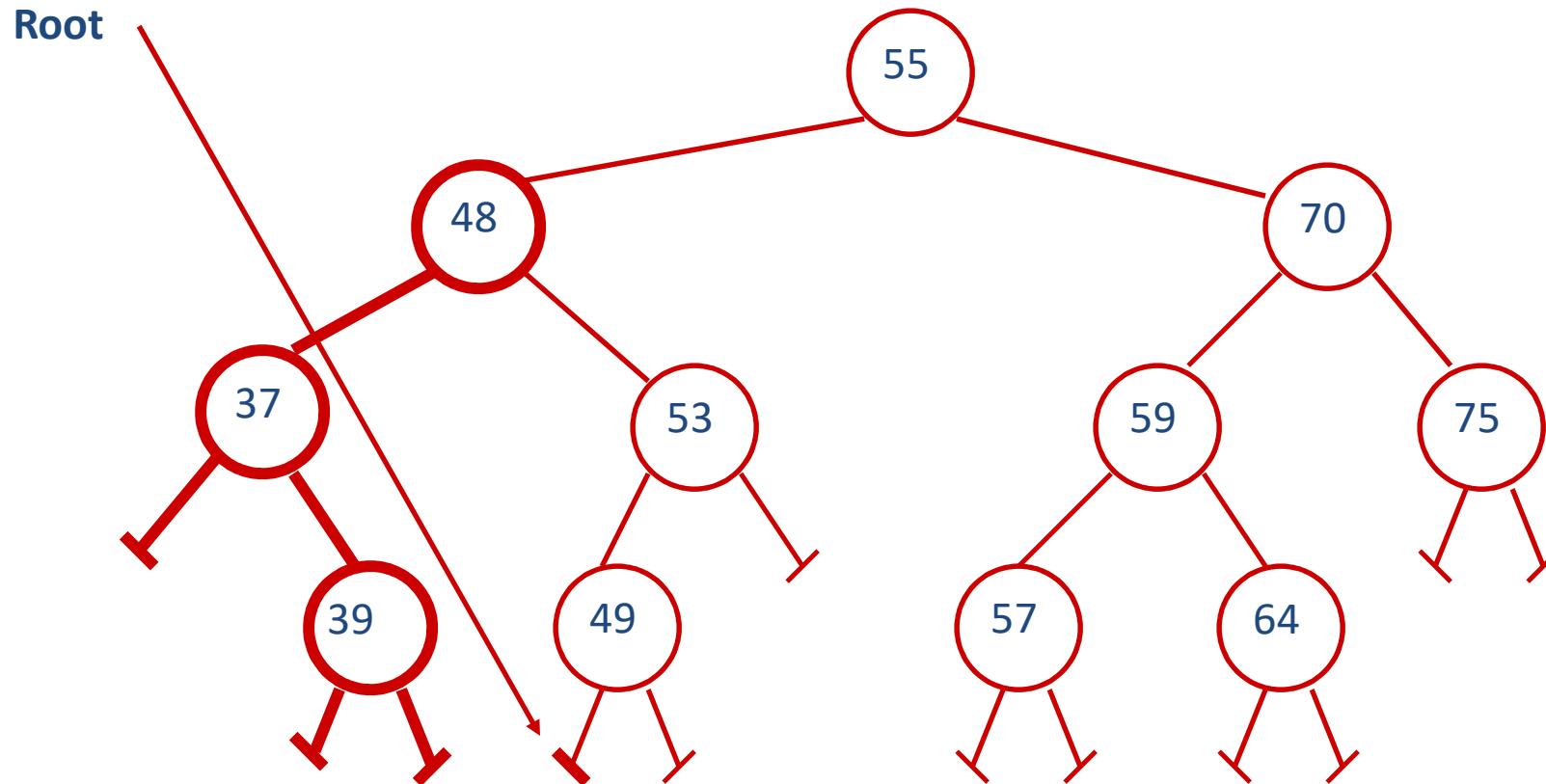
Inorder: 37, 39, 48

Root



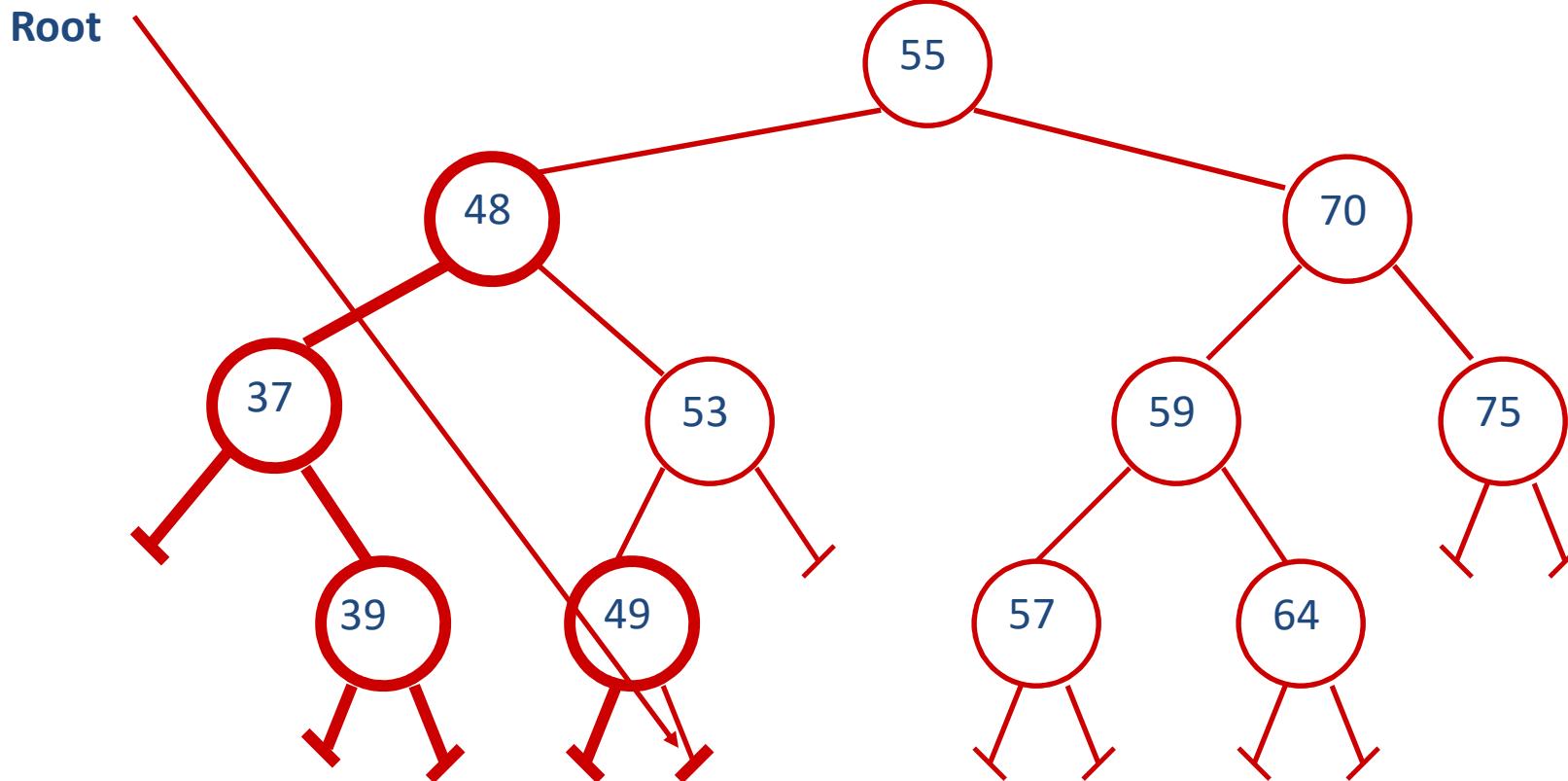
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

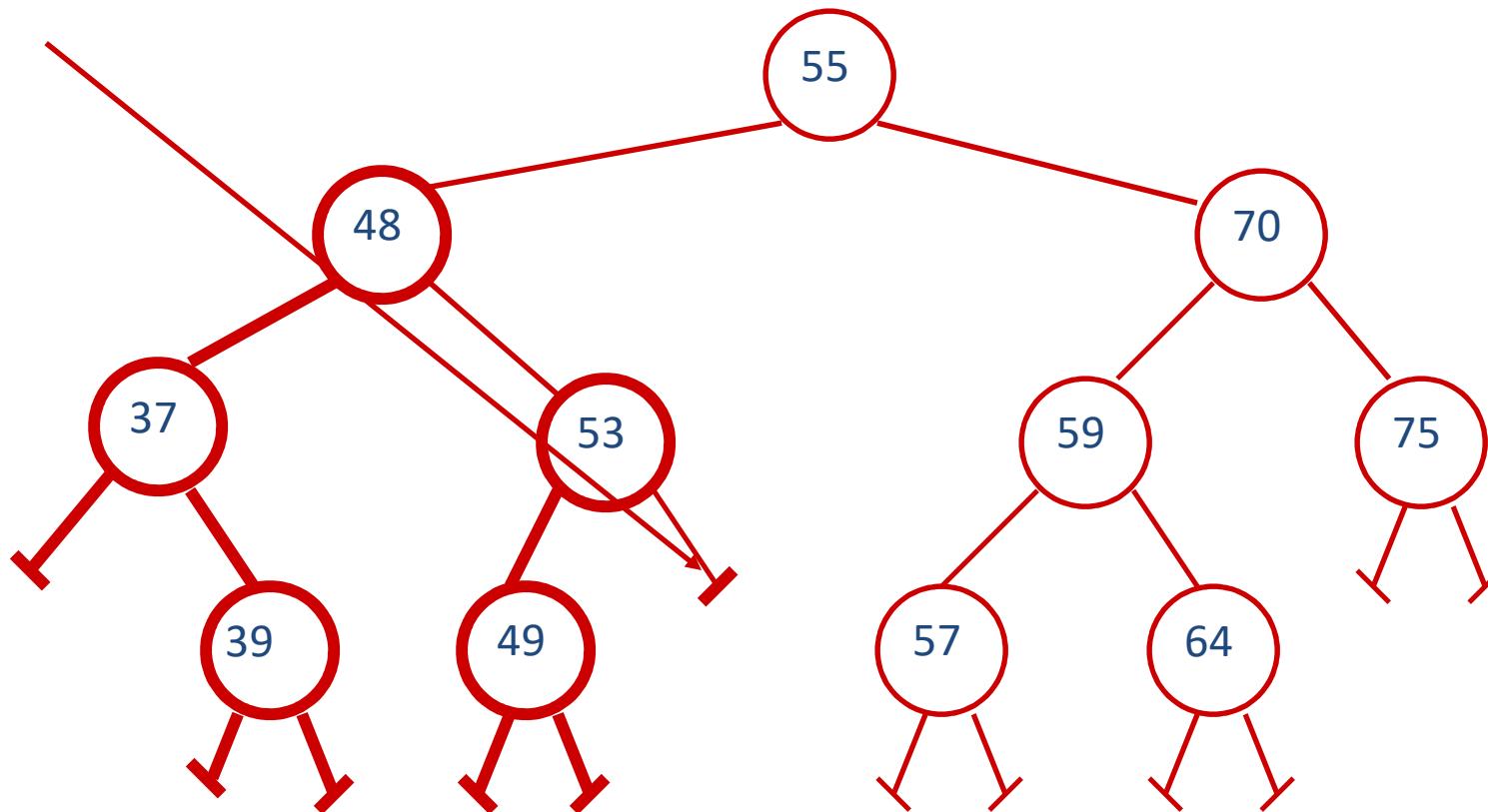
Inorder: 37, 39, 48, 49



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

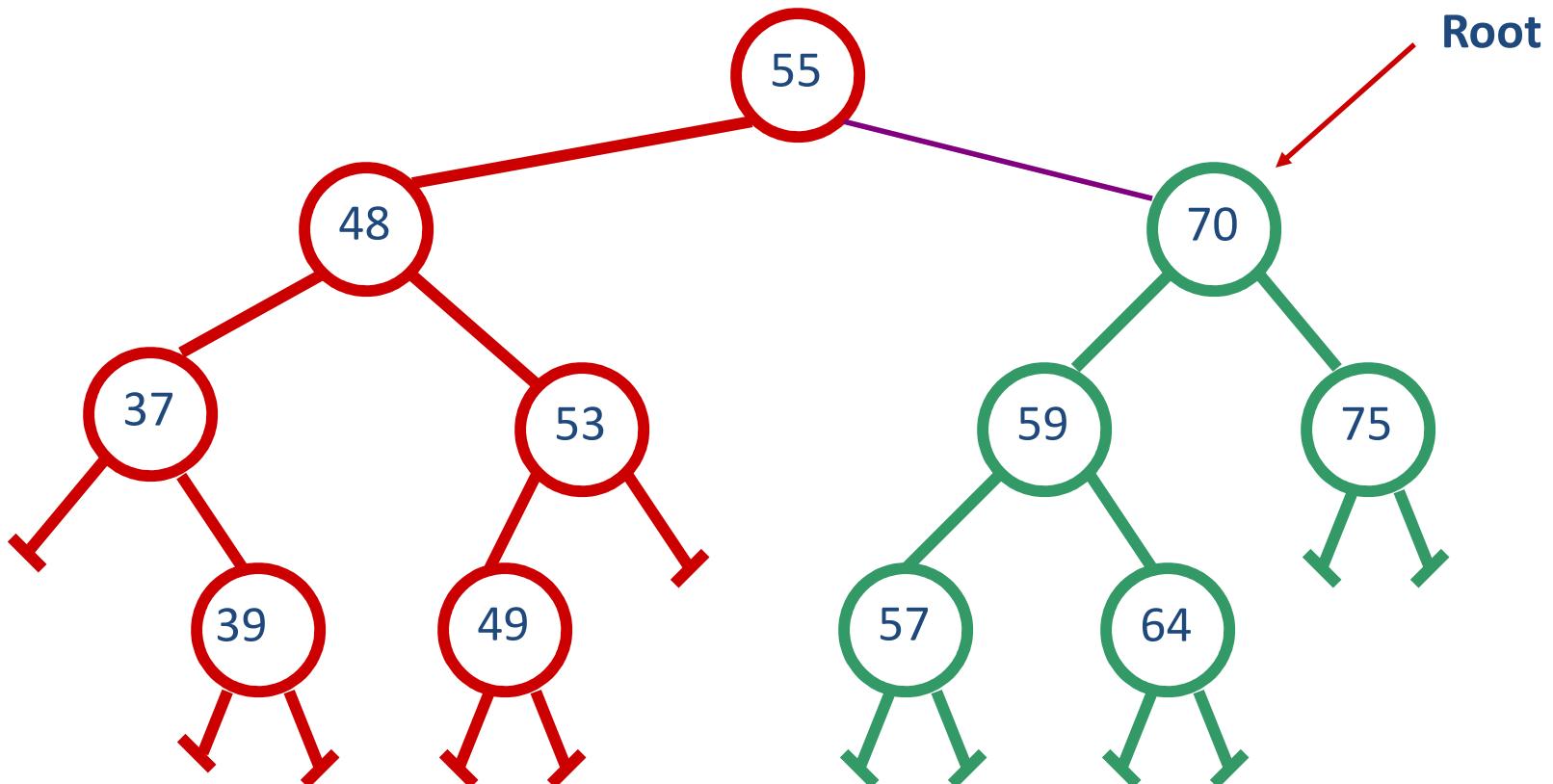
Inorder: 37, 39, 48, 49, 53

Root



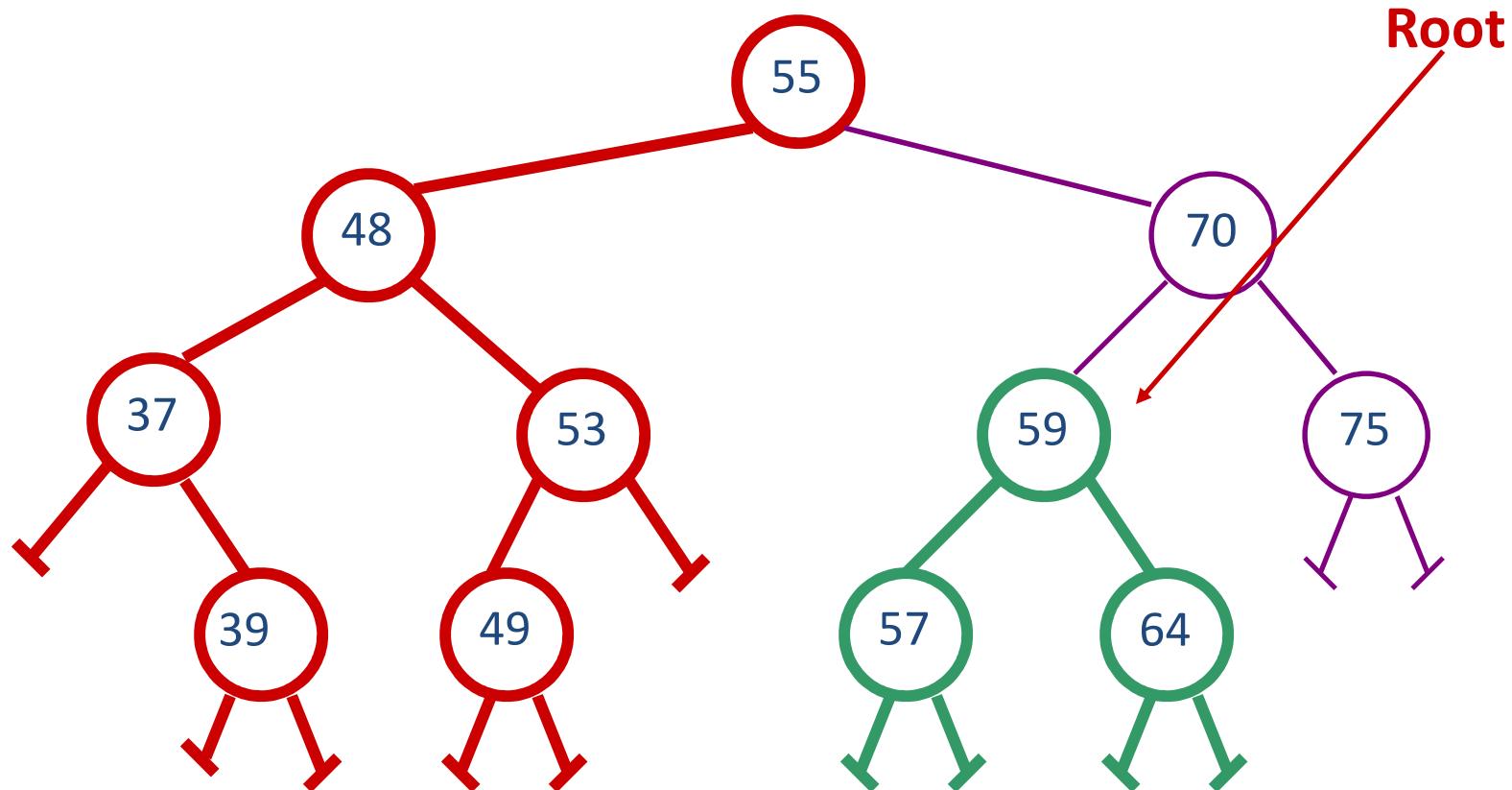
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48, 49, 53, 55



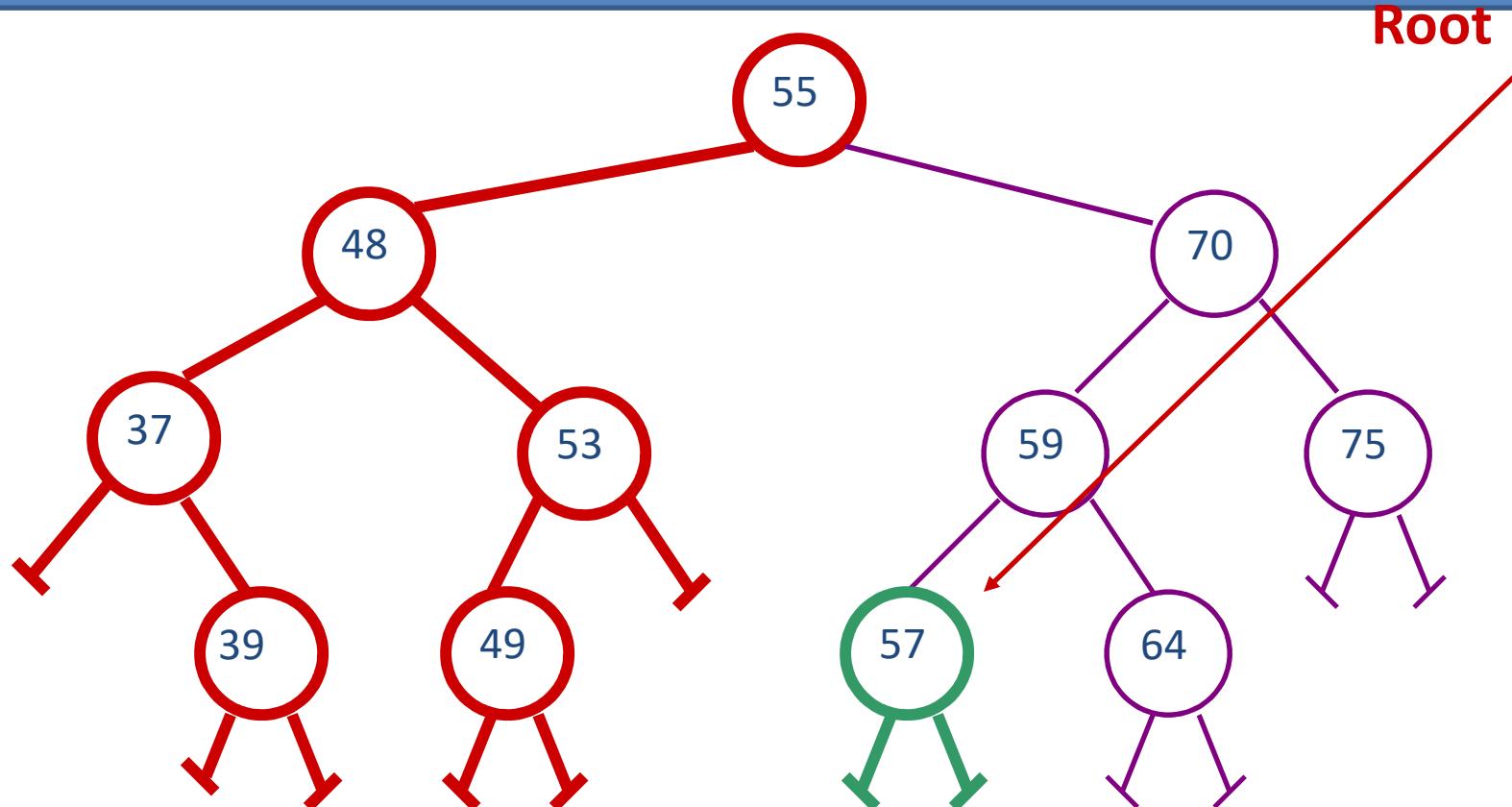
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48, 49, 53, 55



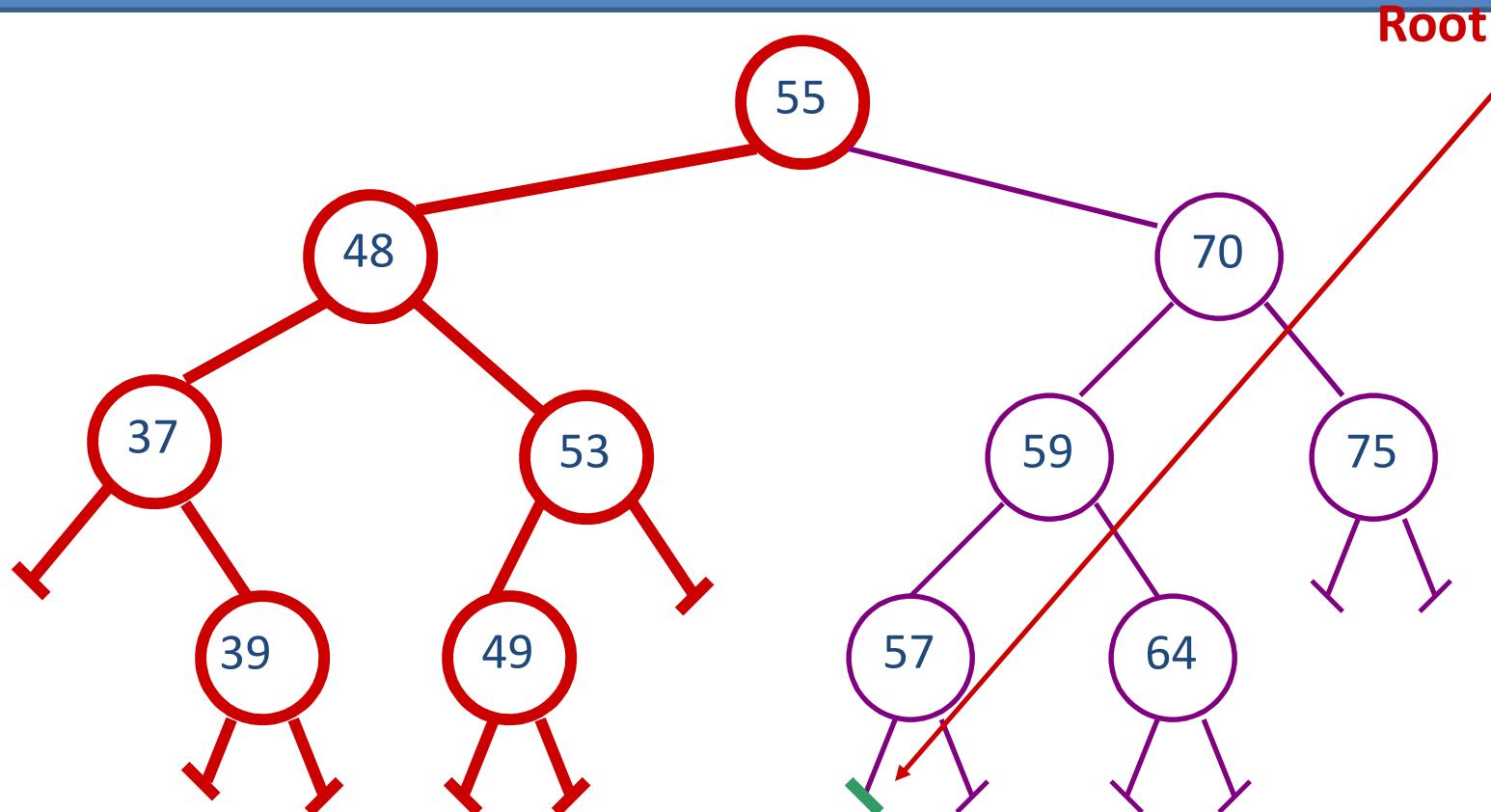
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48, 49, 53, 55



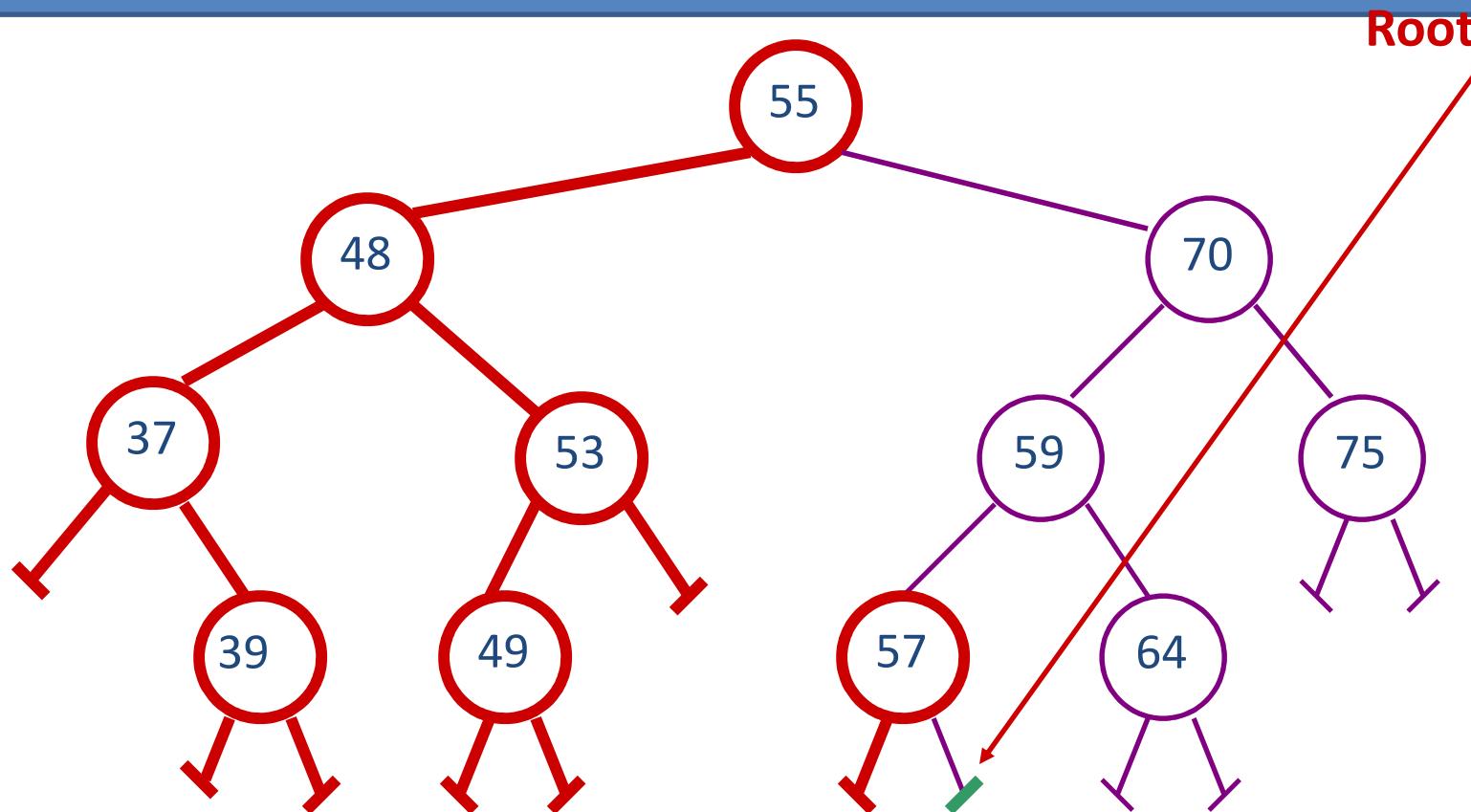
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48, 49, 53, 55



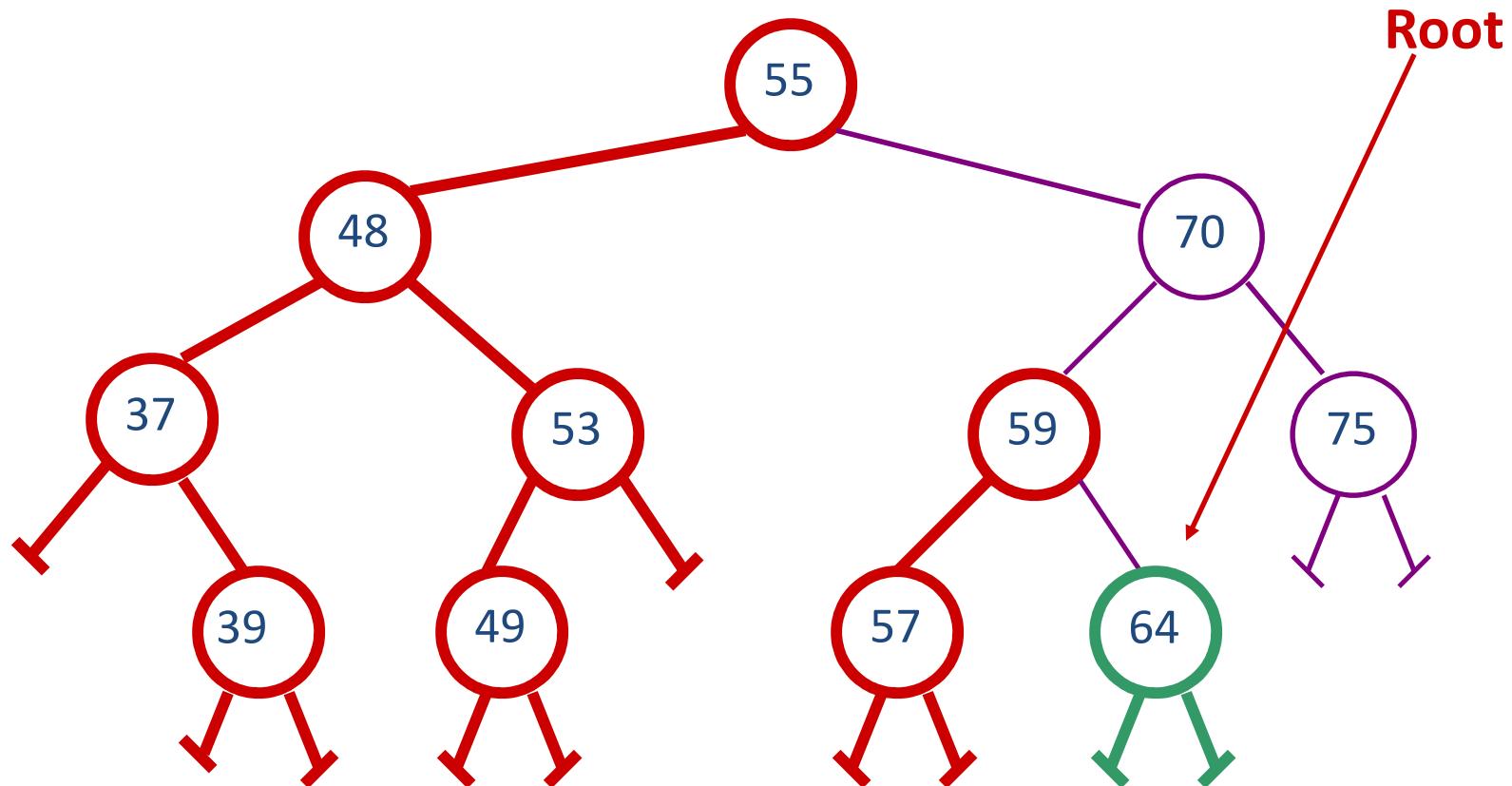
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57



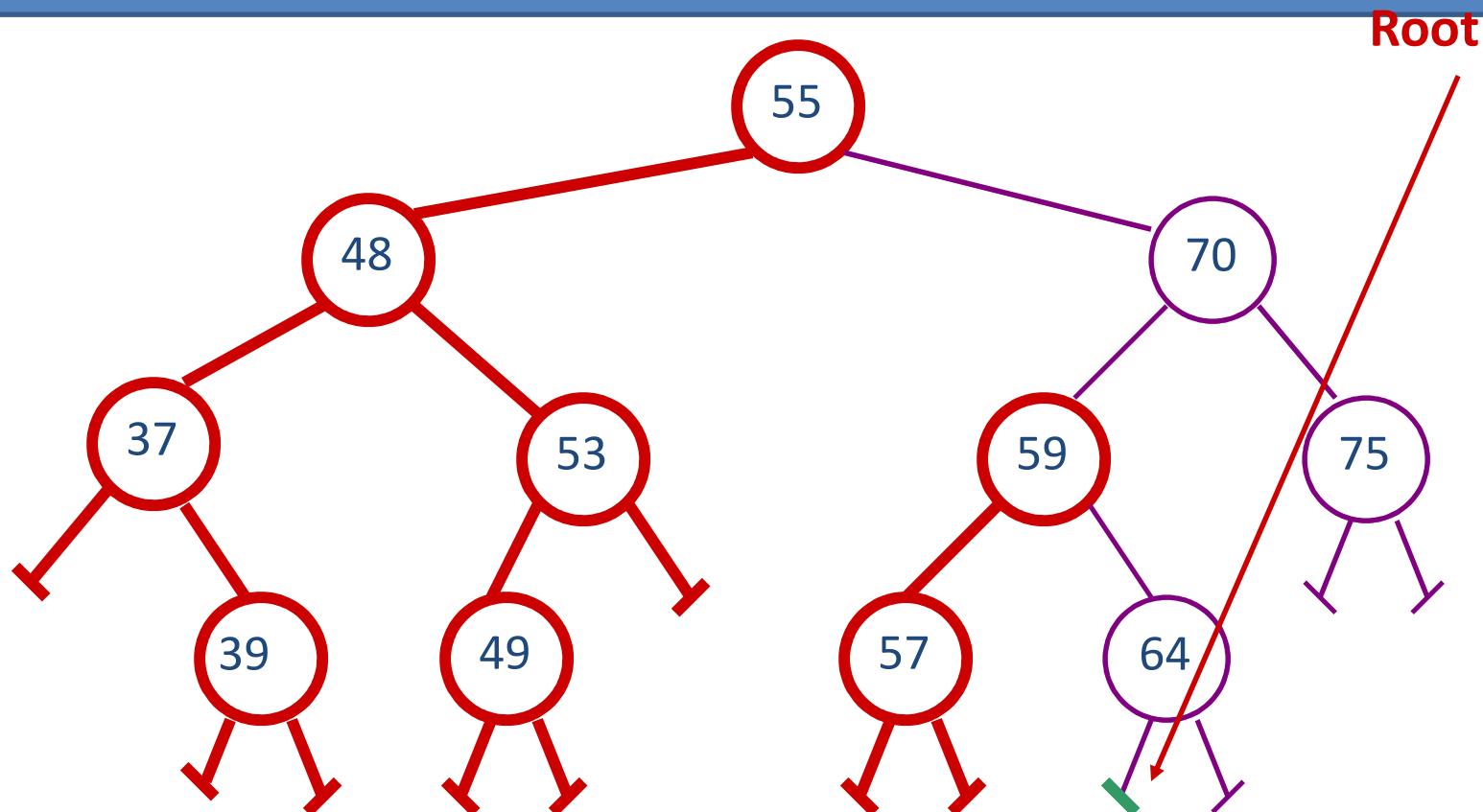
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59



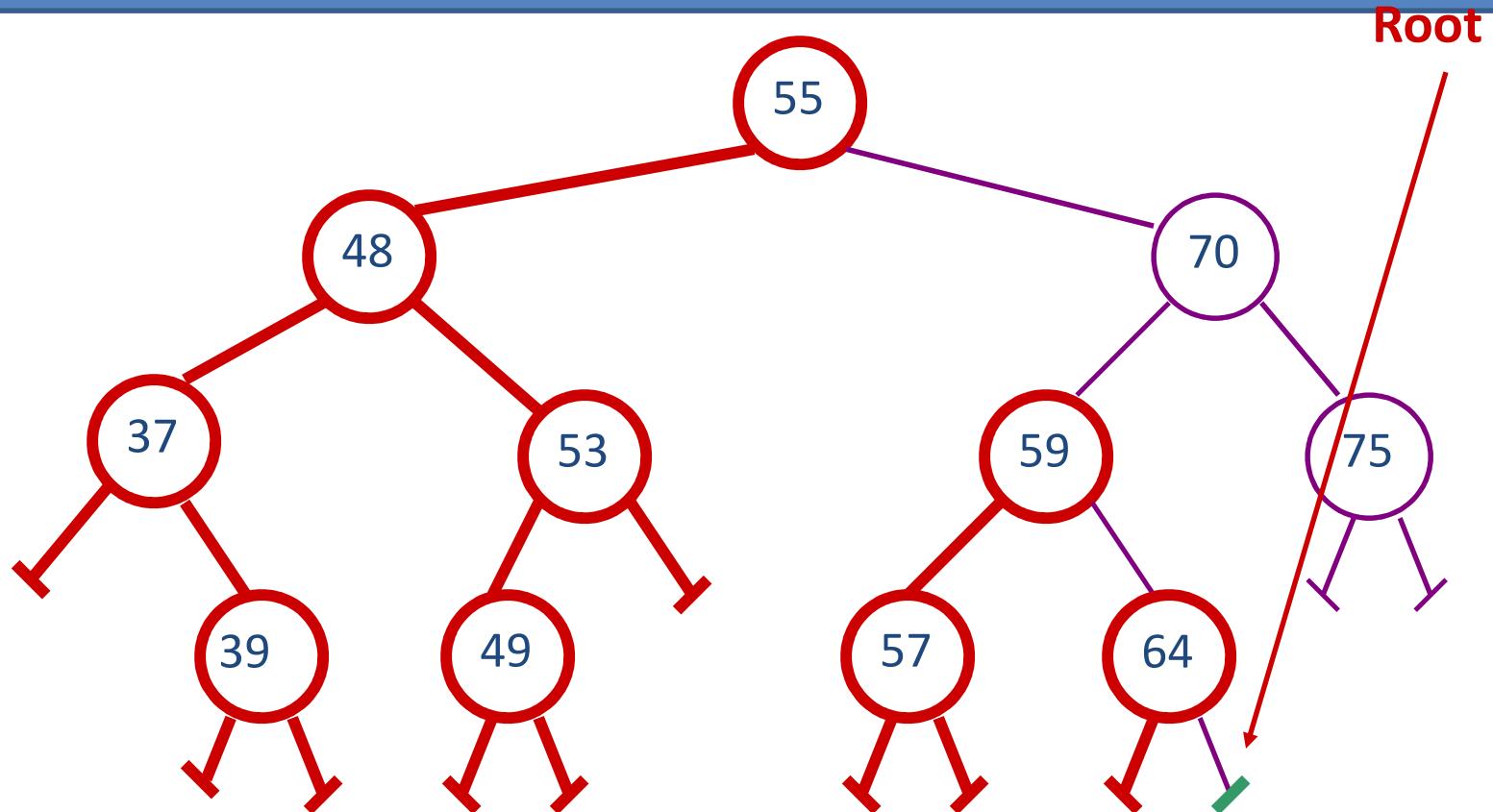
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59



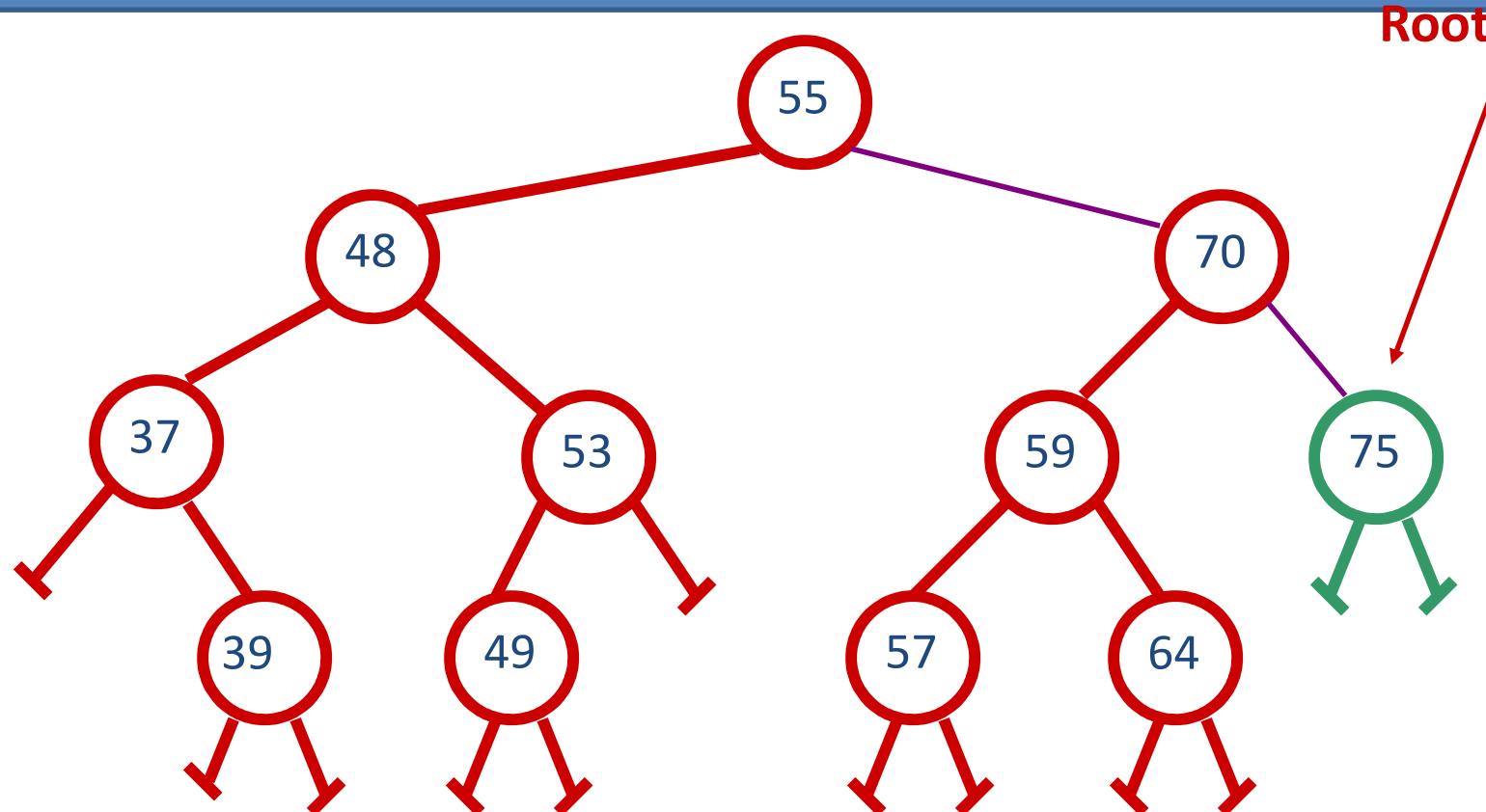
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59, 64



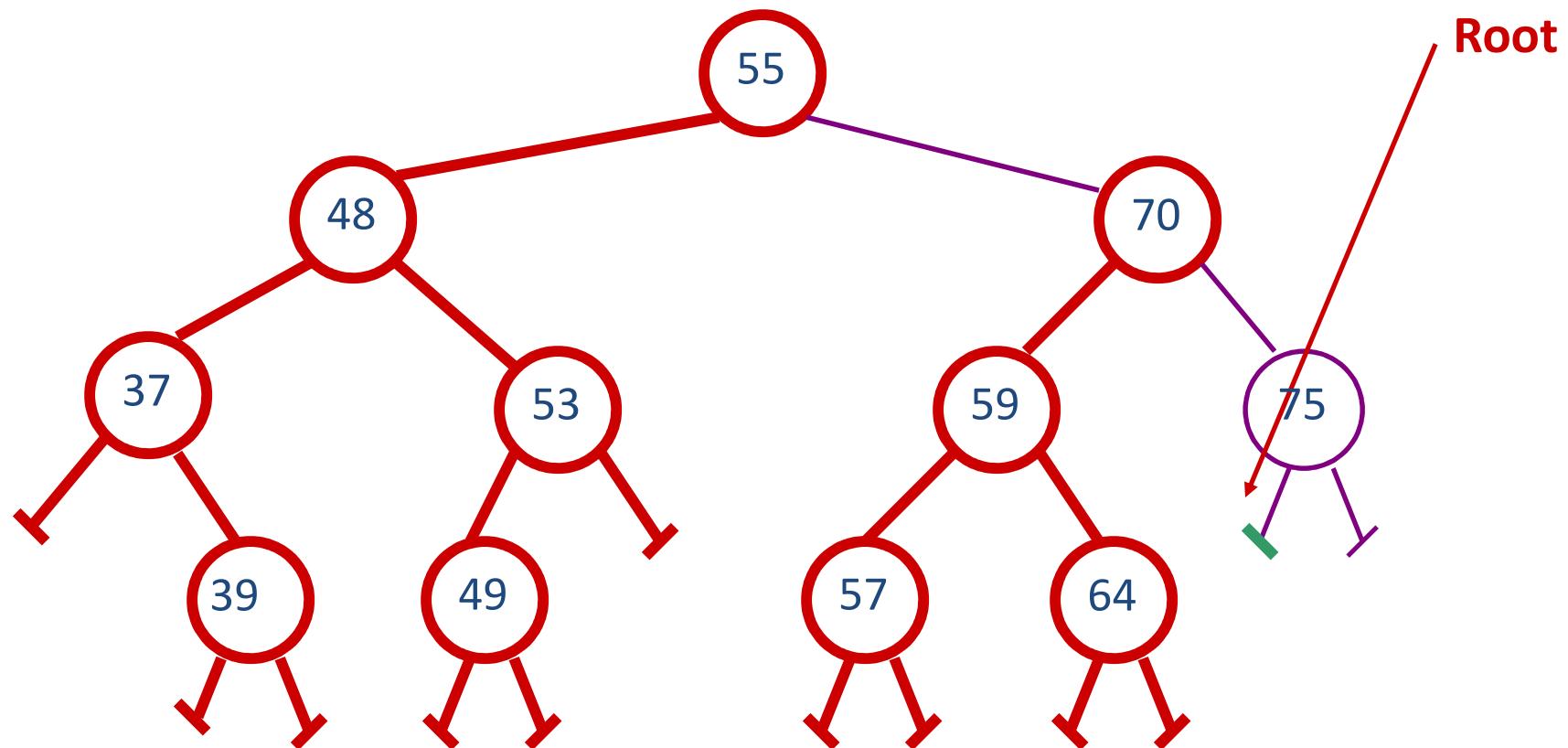
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59, 64, 70



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

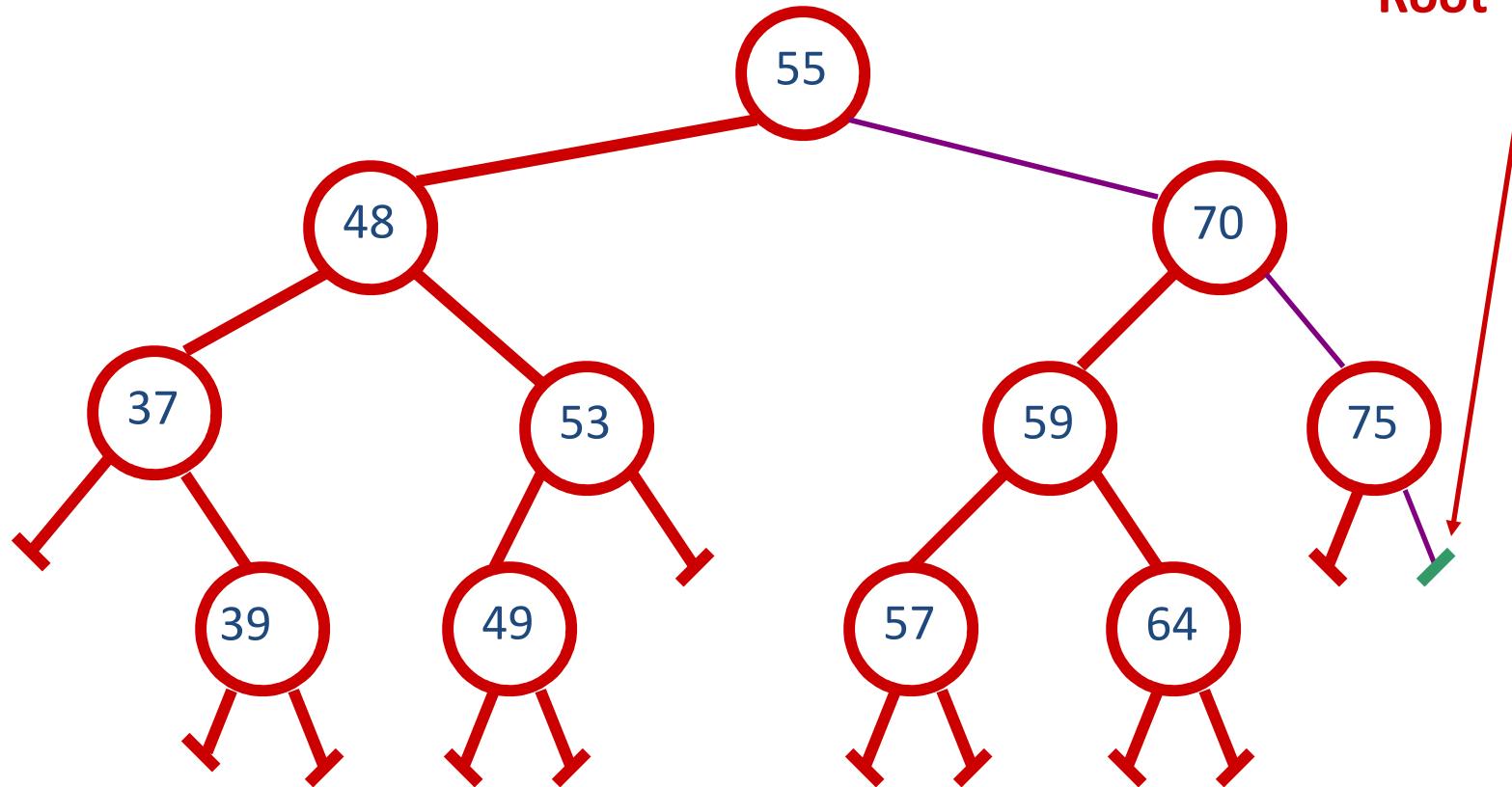
Inorder: 37, 39, 48, 49, 53, 55, 57, 59, 64, 70



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

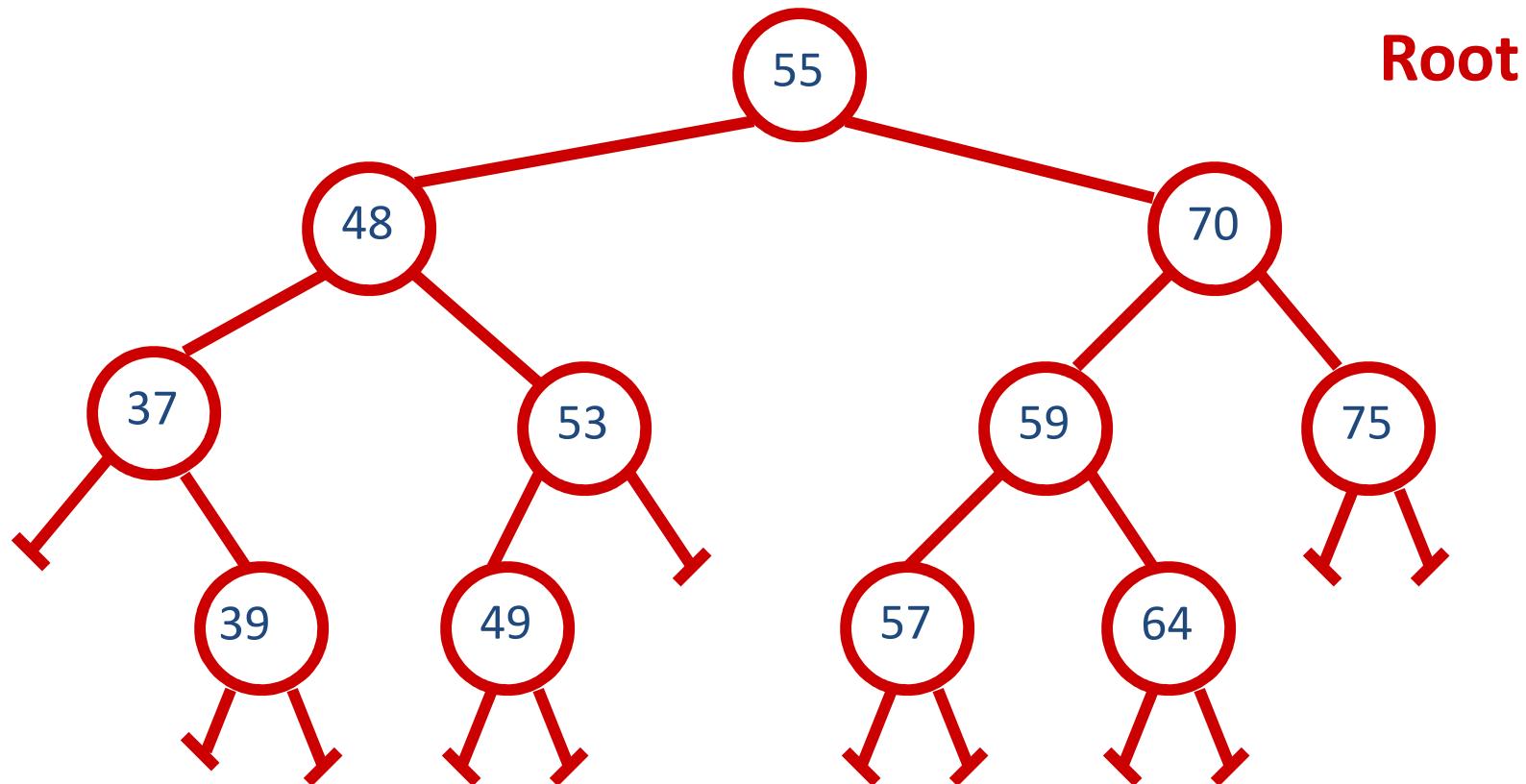
Inorder: 37, 39, 48, 49, 53, 55, 57, 59, 64, 70, 75

Root



```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Inorder: 37, 39, 48, 49, 53, 55, 57, 59, 64, 70, 75



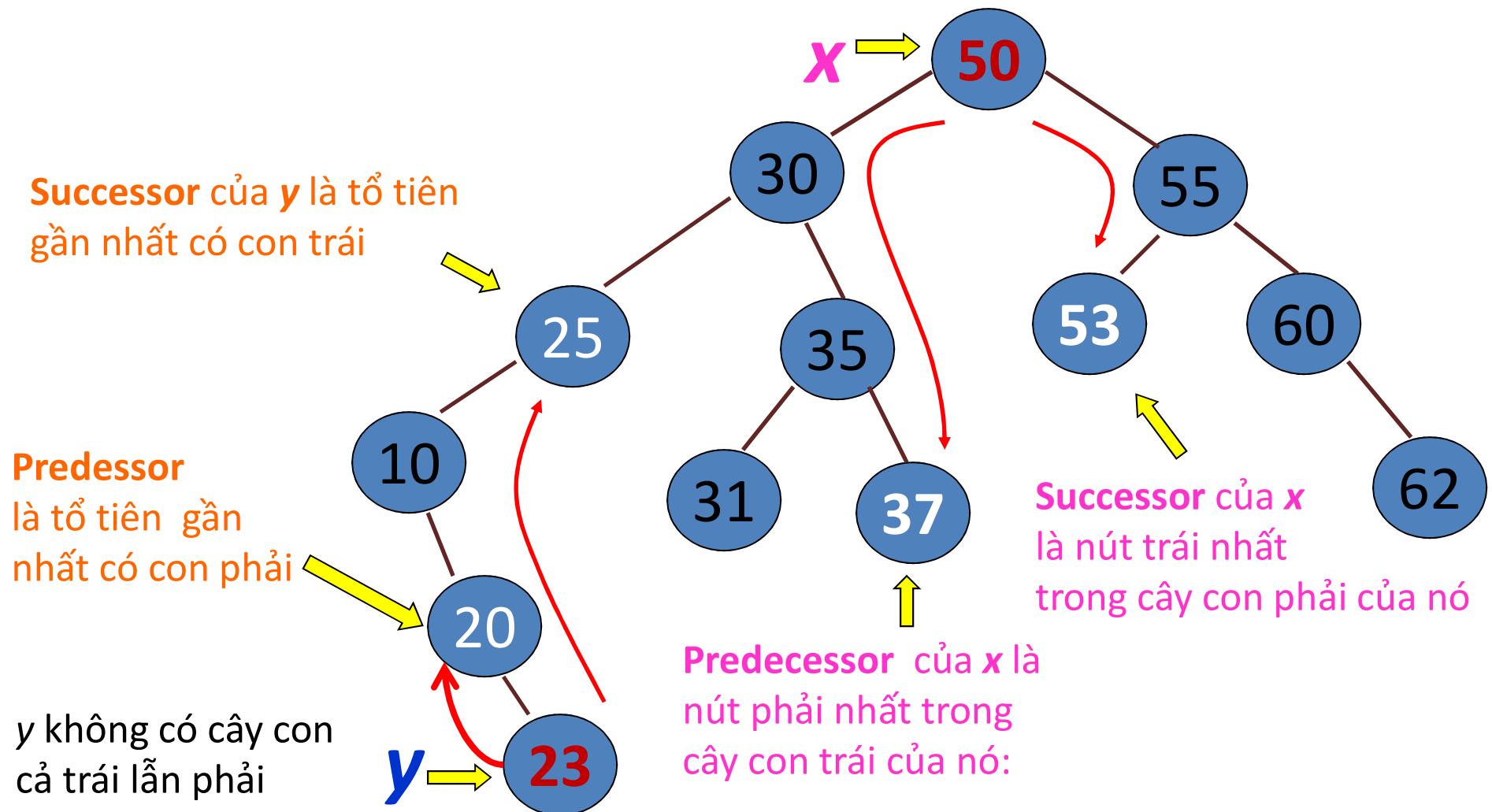
```
void printInorder(TreeNode *Root)
{
    if (Root != NULL) {
        printInorder(Root->left);
        printf(" %d", Root->key);
        printInorder(Root->right);
    }
}
```

Kế cận trước (Predecessor) và Kế cận sau (Successor)

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

Kế cận trước (Predecessor) và Kế cận sau (Successor)

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



Tìm kế cận sau của x

Successor (TreeNode *x) : Trả lại nút kế cận sau của nút x

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 find-min(x->right): $y = \text{find-min}(x->\text{right})$
 - 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 của x

Predecessor (TreeNode x) : Trả lại nút kế cận trước của nút x

Tương tự như tìm kế cận sau, có hai tình huống

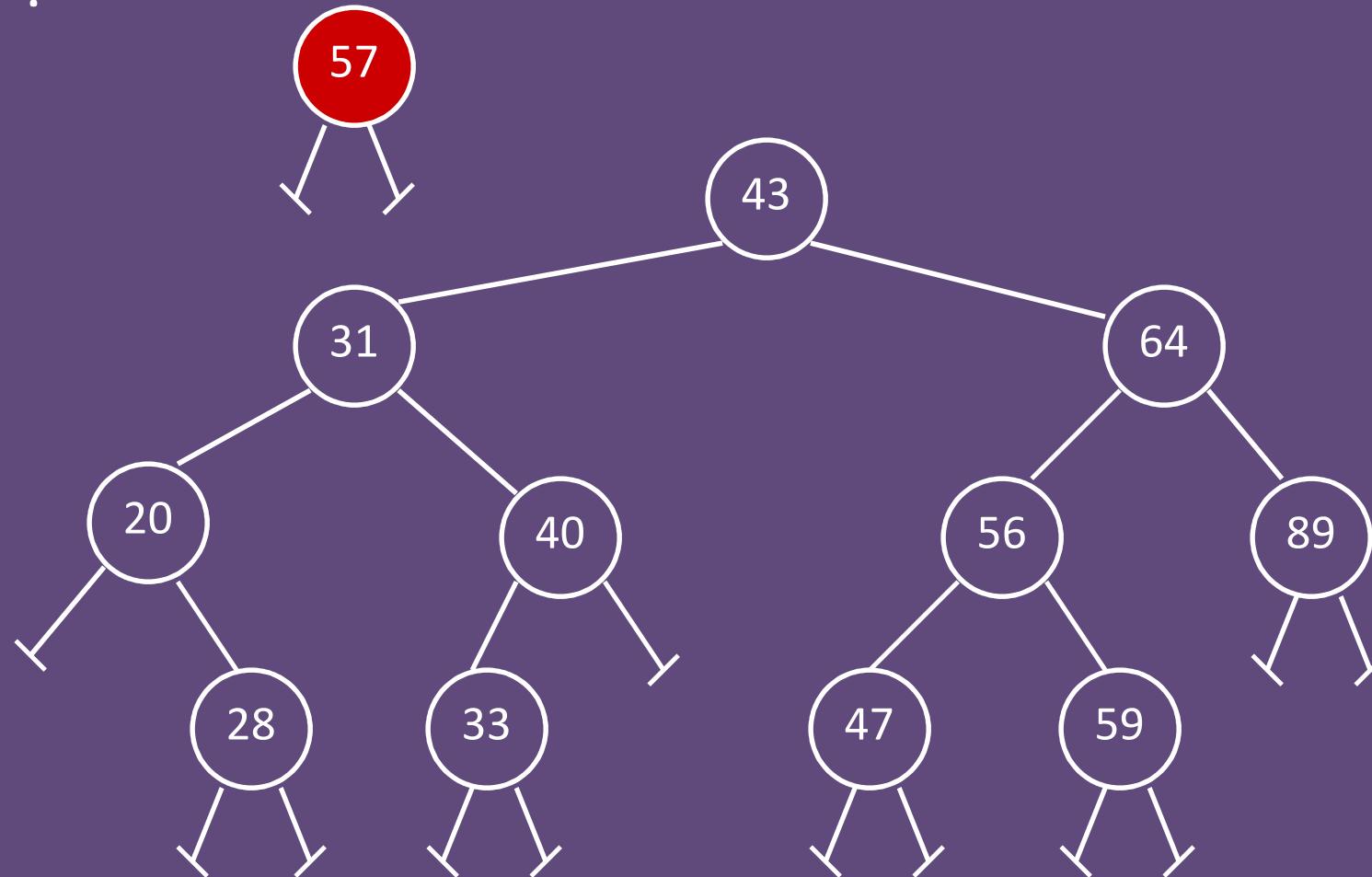
- Nếu x có con trái thì kế cận trước của x sẽ là nút y với khoá $key[y]$ lớn nhất trong cây con trái của x (nói cách khác y là nút phải nhất nhất trong cây con trái của x):

y = find_max(x->left)

- 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 .

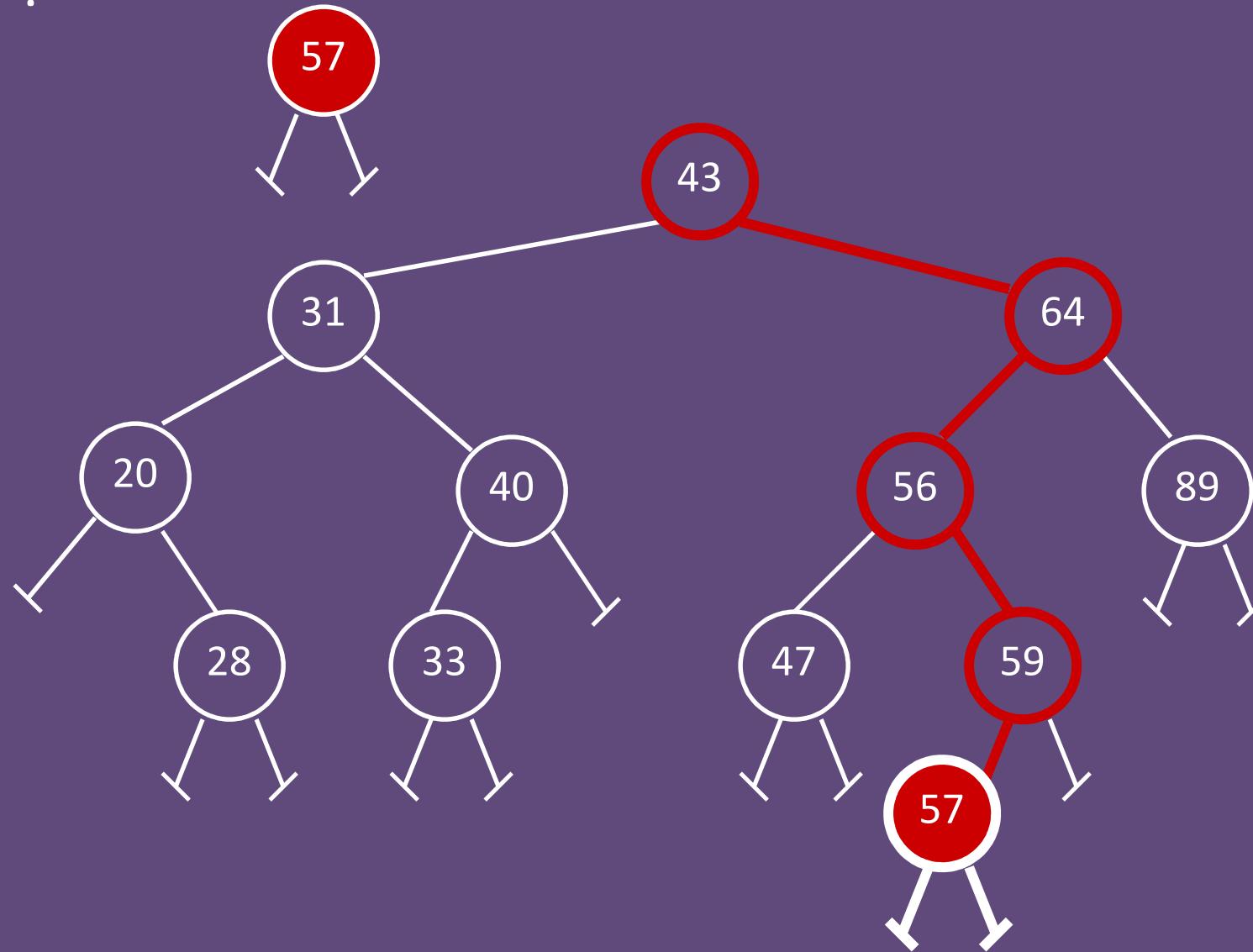
3.3. Các phép toán cơ bản: Chèn (Insert)

Ví dụ 1



3.3. Các phép toán cơ bản: Chèn (Insert)

Ví dụ 1



3.3. Các phép toán cơ bản: Chèn (Insert)

TreeNode* insert(int x, TreeNode* root)

- Thông số đầu vào:
 - Khóa của phần tử cần bổ sung;
 - con trỏ đến nút đang xét (thoạt tiên là nút gốc).
- Thông số đầu ra:
 - Con trỏ trỏ đến nút mới được bổ sung vào cây
- Thuật toán:

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).

3.3. Các phép toán cơ bản: Chèn (Insert)

```
TreeNode* insert(int x, TreeNode* root)
{
    if (root == NULL)
        root = create_node(x);
    else if (x < root->key)
        root->left = insert(x, root->left);
    else if (x > root->key)
        root->right = insert(x, root->right);

    return root;
}
```

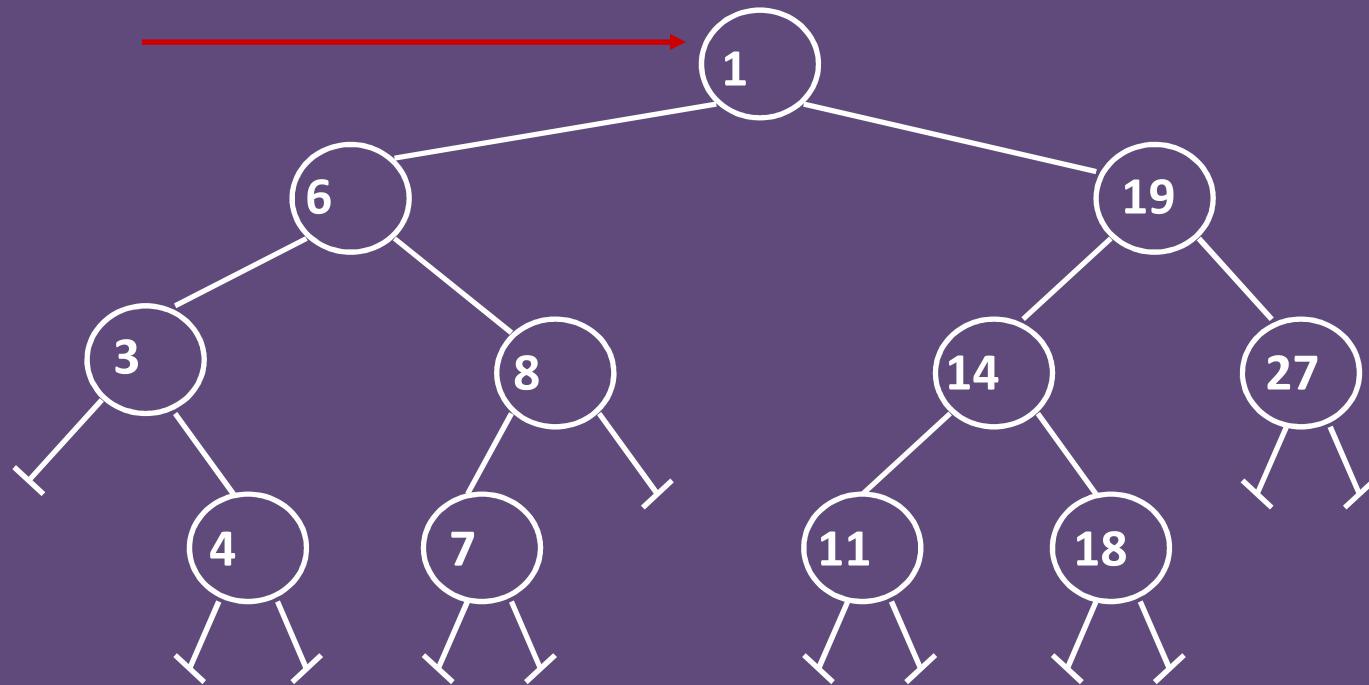
- Thuật toán:

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).

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

Insert 9

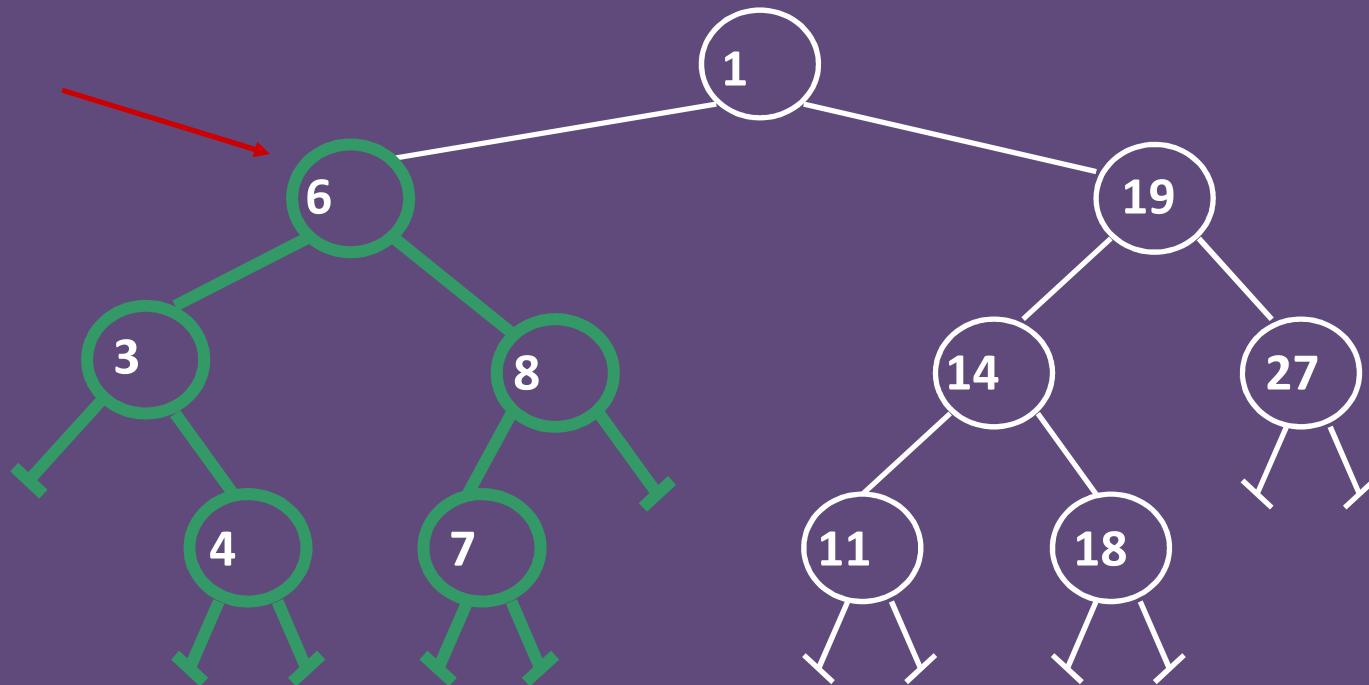
root



```
TreeNode* insert(int x, TreeNode* root)
{
    if (root == NULL)
        root = create_node(x);
    else if (x < root->key)
        root->left = insert(x, root->left);
    else if (x > root->key)
        root->right = insert(x, root->right);
    return root;
}
```

Insert 9

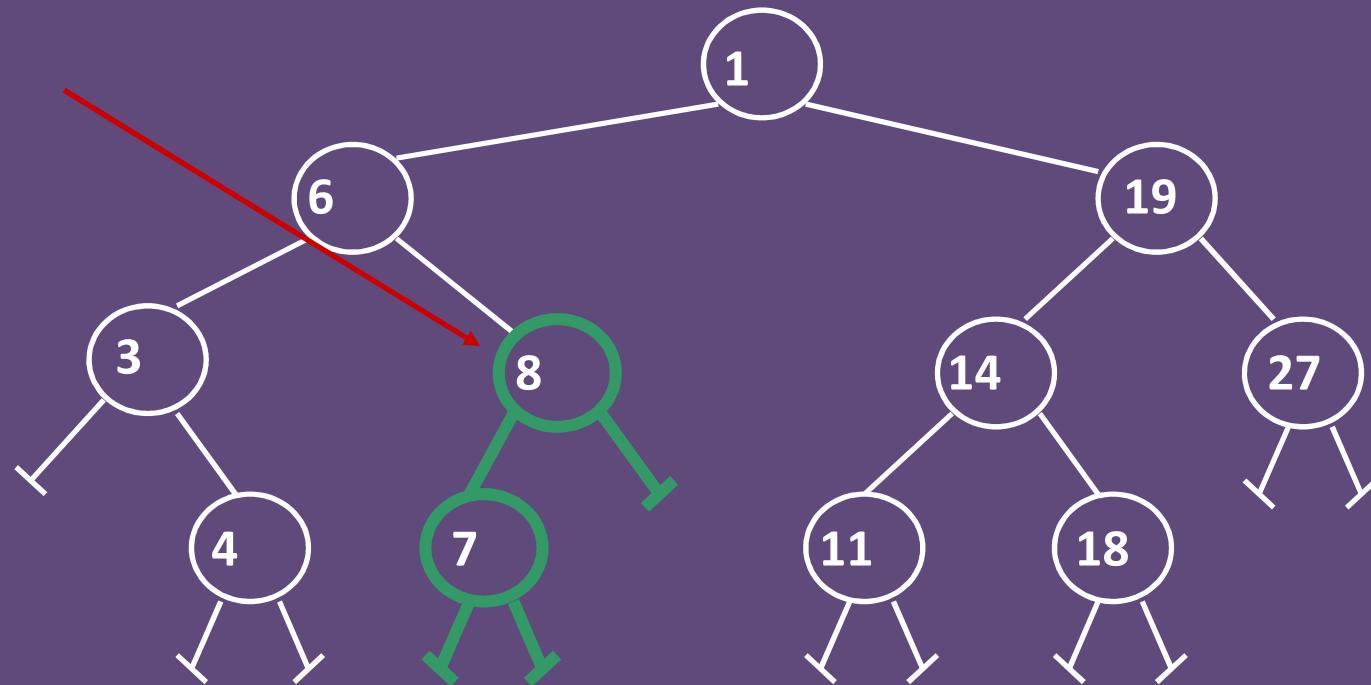
root



```
TreeNode* insert(int x, TreeNode* root)
{
    if (root == NULL)
        root = create_node(x);
    else if (x < root->key)
        root->left = insert(x, root->left);
    else if (x > root->key)
        root->right = insert(x, root->right);
    return root;
}
```

Insert 9

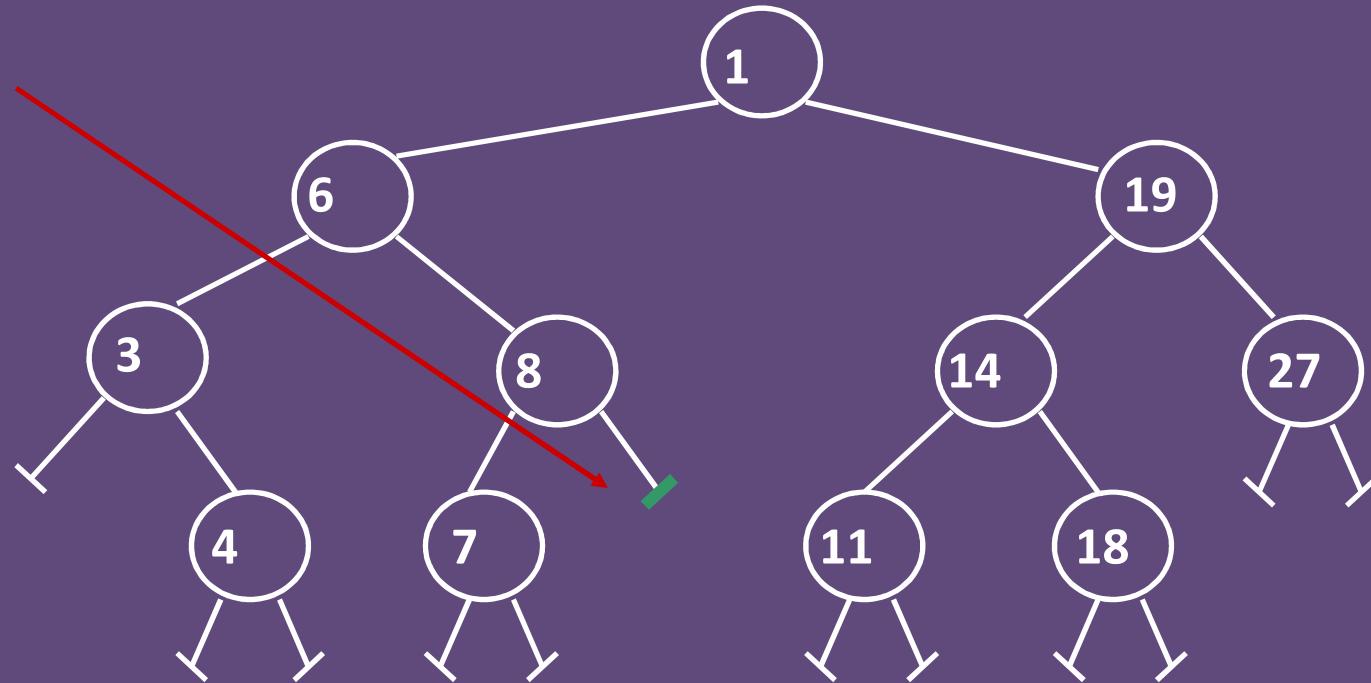
root



```
TreeNode* insert(int x, TreeNode* root)
{
    if (root == NULL)
        root = create_node(x);
    else if (x < root->key)
        root->left = insert(x, root->left);
    else if (x > root->key)
        root->right = insert(x, root->right);
    return root;
}
```

Insert 9

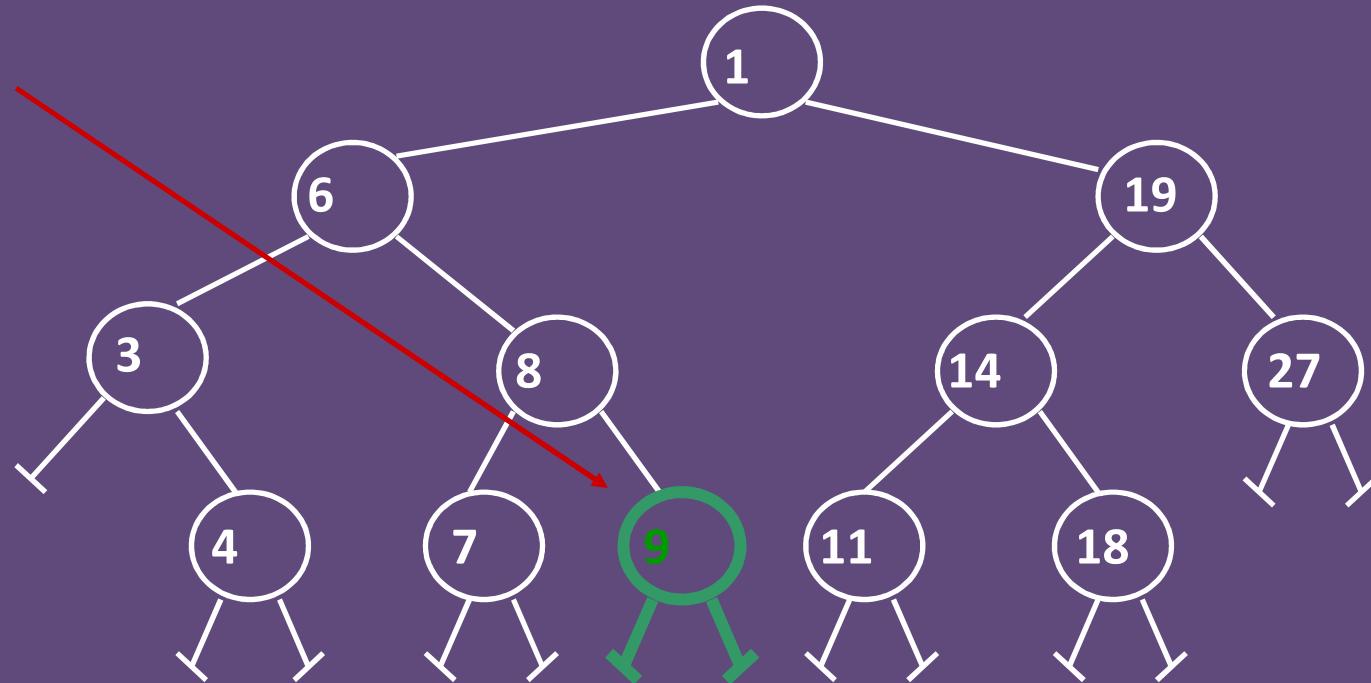
root



```
TreeNode* insert(int x, TreeNode* root)
{
    if (root == NULL)
        root = create_node(x);
    else if (x < root->key)
        root->left = insert(x, root->left);
    else if (x > root->key)
        root->right = insert(x, root->right);
    return root;
}
```

Insert 9

root



```
TreeNode* insert(int x, TreeNode* root)
{
    if (root == NULL)
        root = create_node(x);
    else if (x < root->key)
        root->left = insert(x, root->left);
    else if (x > root->key)
        root->right = insert(x, root->right);
    return root;
}
```

3.3. Các phép toán cơ bản: xóa (delete) 1 node khỏi BST

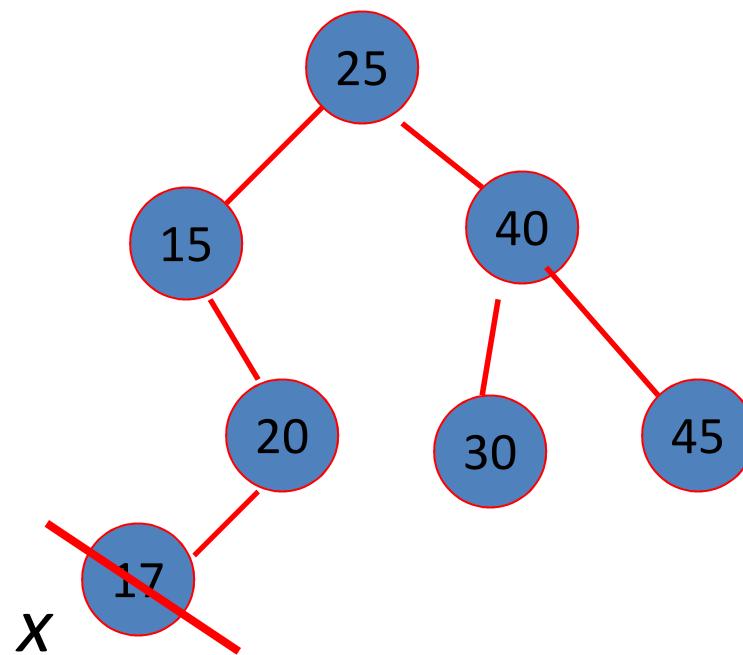
`TreeNode* delete(int x, TreeNode * root)`: xóa nút có khóa bằng x trên cây BST có gốc trả bởi root; hàm trả về gốc của cây BST sau khi xóa nút

- Khi loại bỏ một nút, cần phải đảm bảo cây thu được vẫn là cây nhị phân tìm kiếm. 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

3.3. Các phép toán cơ bản: xóa (delete) 1 node khỏi BST

Tình huống 1: Nút cần xoá là lá

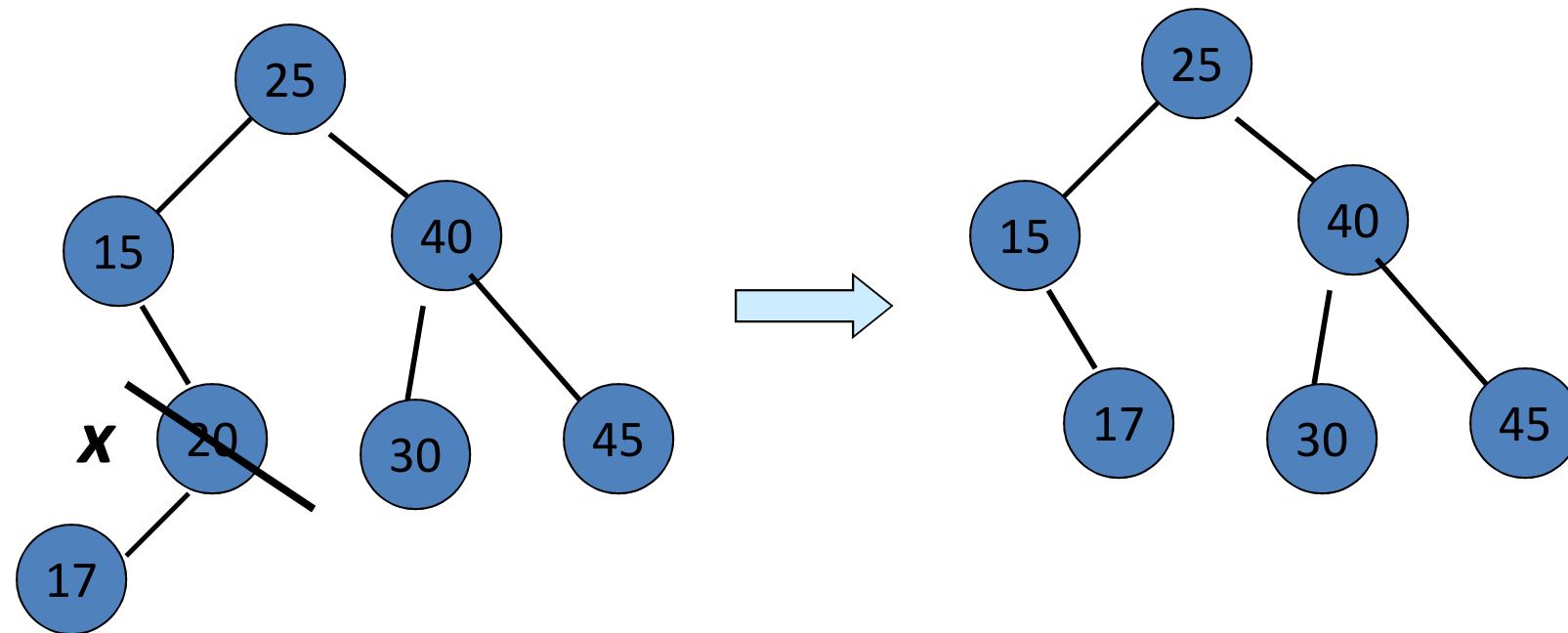
Thao tác: chỉ cần thiết lập con trỏ con của node cha về NULL.



3.3. Các phép toán cơ bản: xóa (delete) 1 node khỏi BST

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

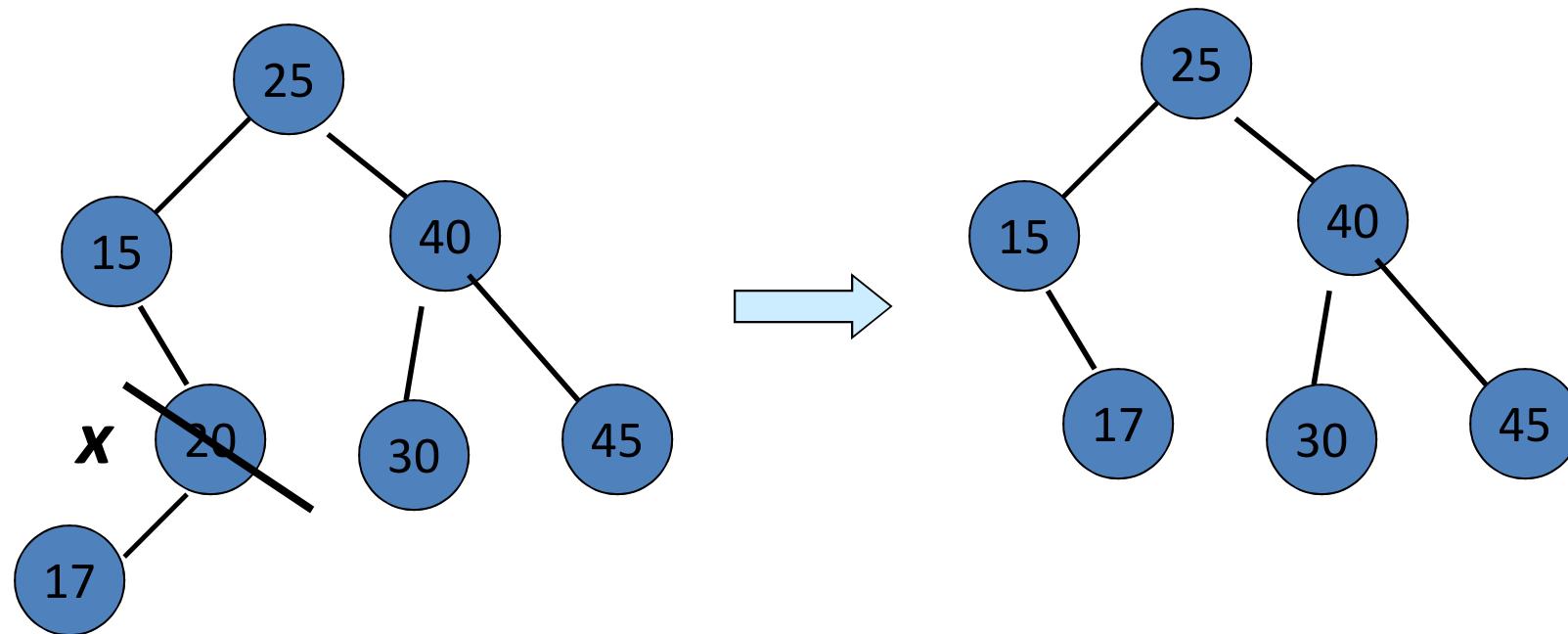
Thao tác: gắn cây con trái của x vào cha của x



3.3. Các phép toán cơ bản: xóa (delete) 1 node khỏi BST

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

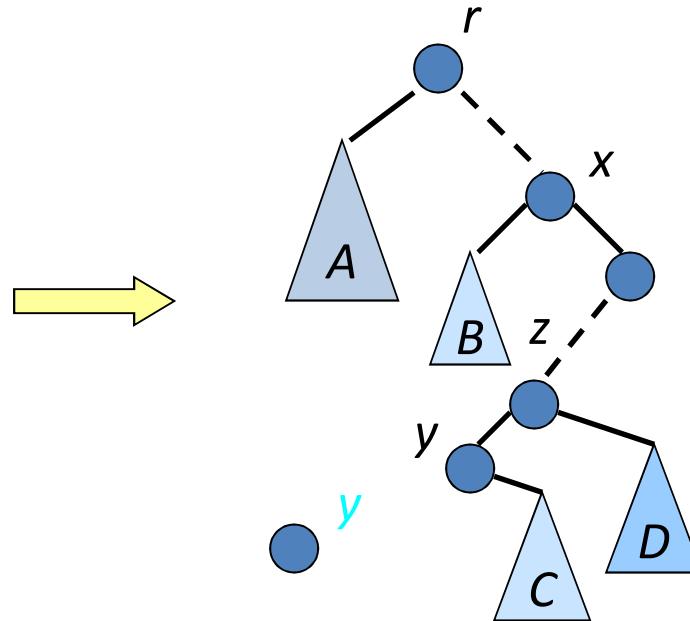
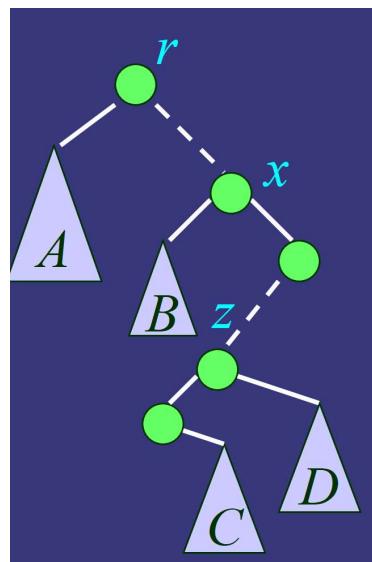
Thao tác: gắn cây con trái của x vào cha của x



3.3. Các phép toán cơ bản: xóa (delete) 1 node khỏi BST

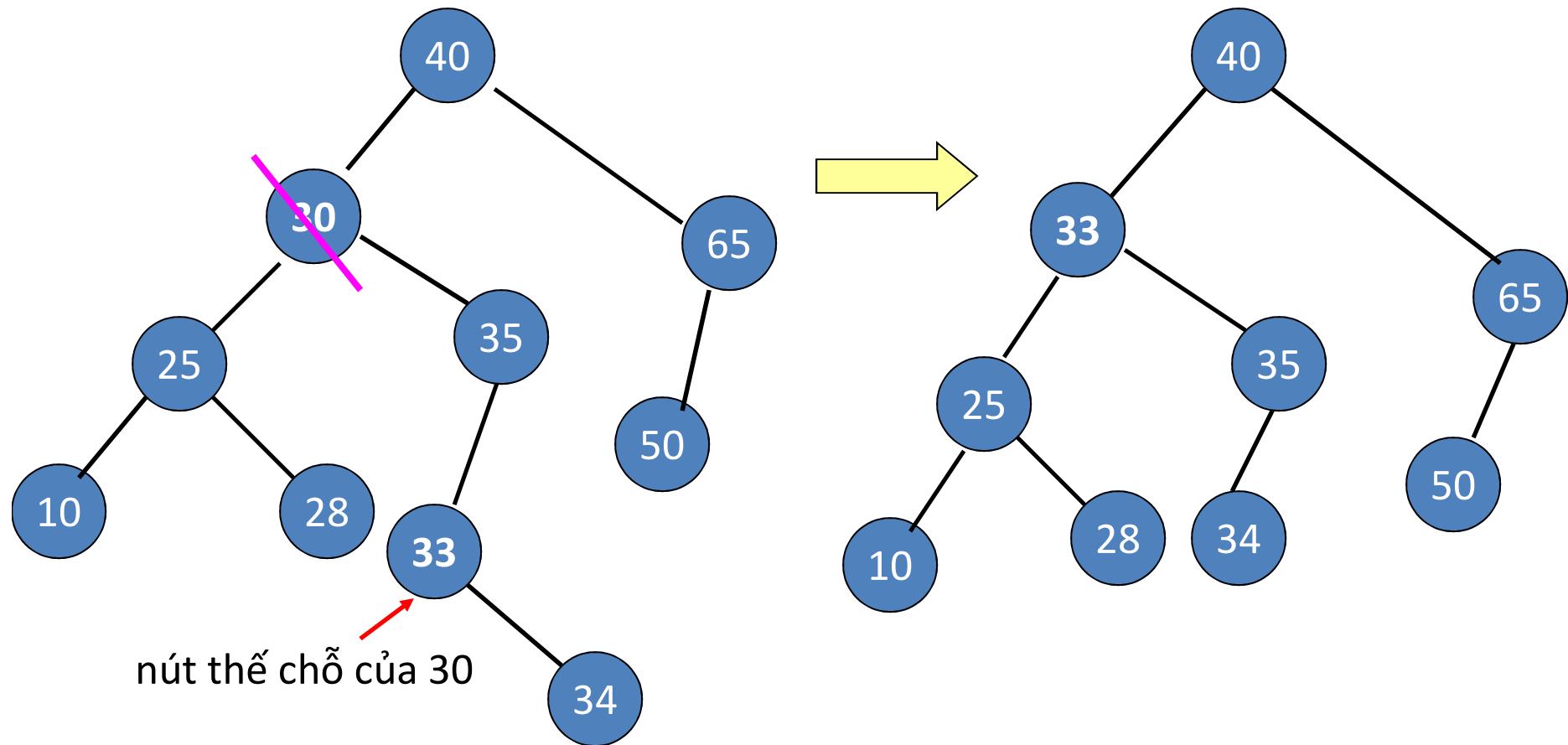
Tình huống 4: nút cần xóa x có hai con

- Thao tác:**
1. Chọn nút y để thế vào chỗ của x , nút y sẽ là nút trái nhất của cây con phải 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.



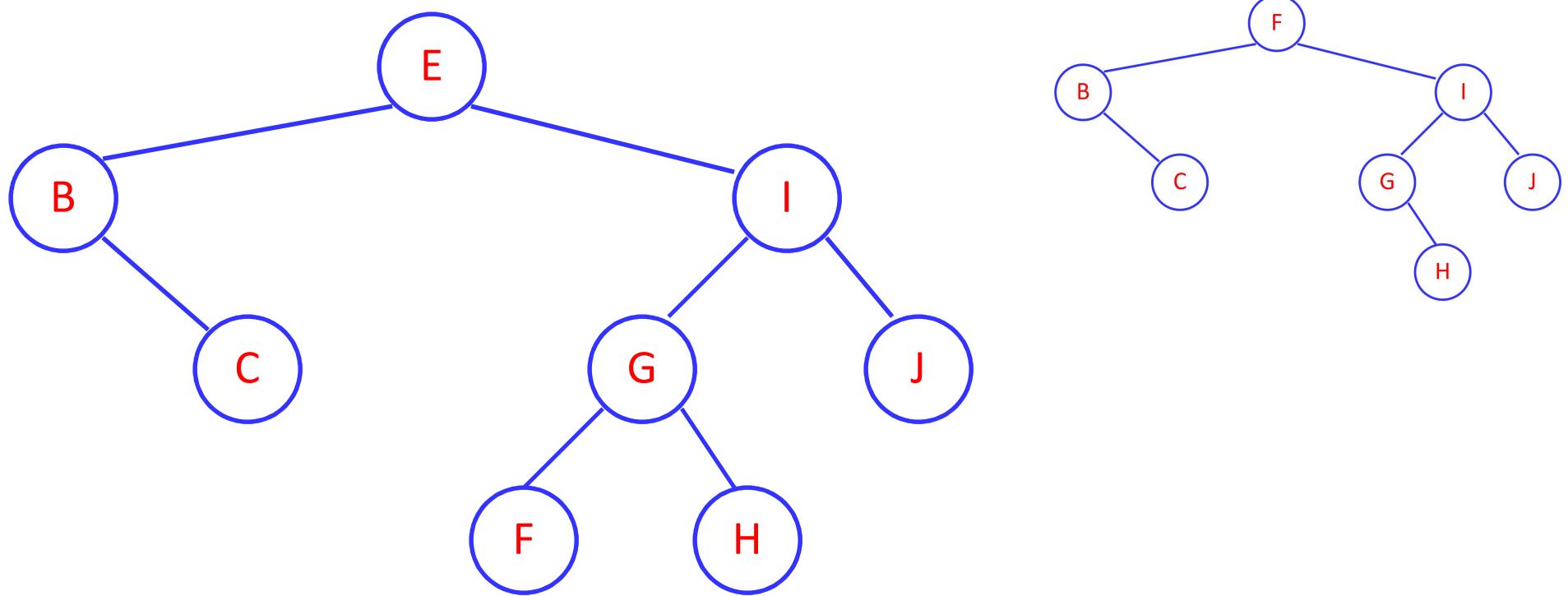
Ví dụ 1: Tinh huống 4

10, 25, 28, 30, 33, 34, 35, 40, 50, 65 \rightarrow 10, 25, 28, 33, 34, 35, 40, 50, 65



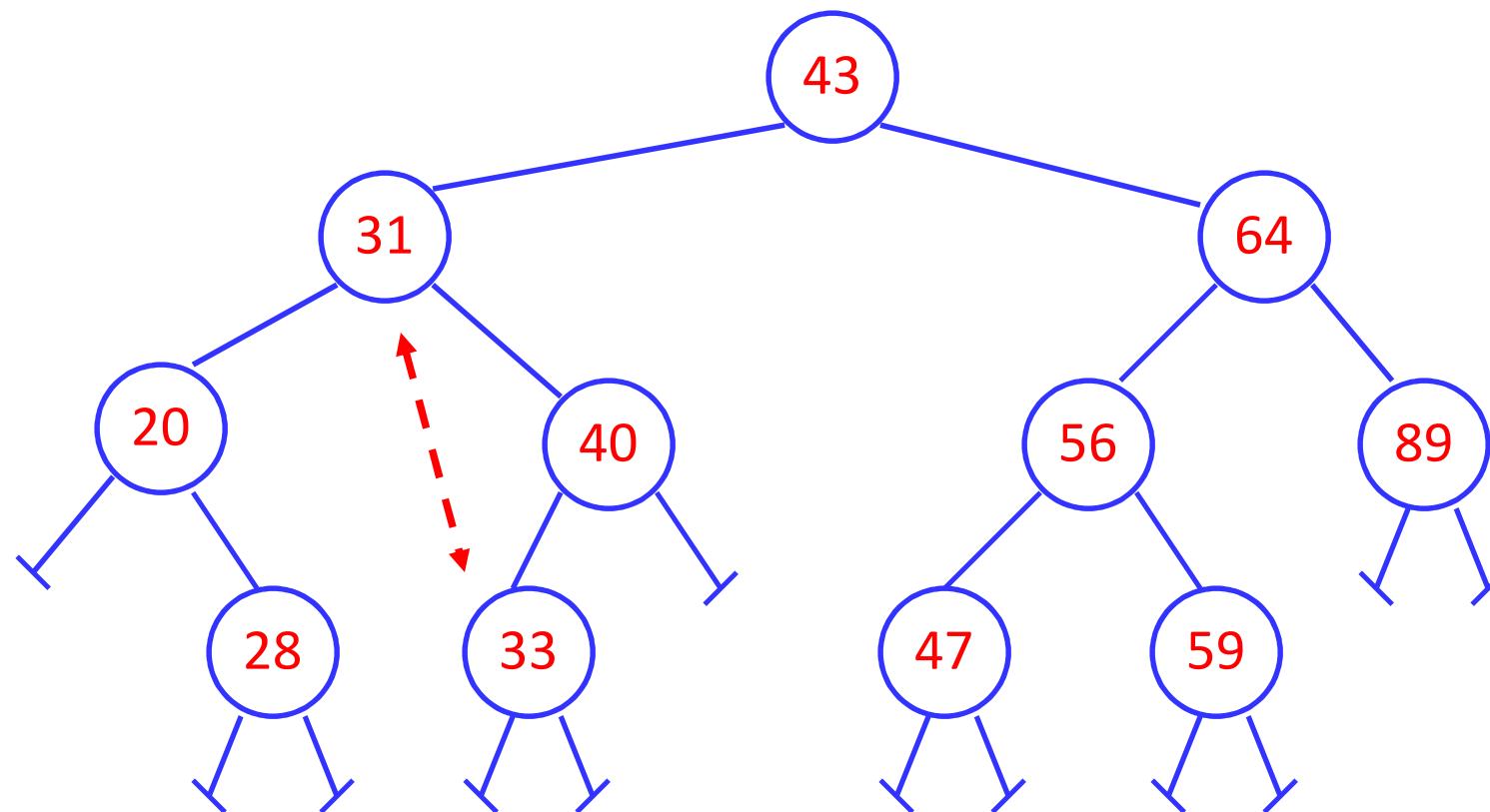
Ví dụ 2: Tình huống 4

Xóa node E



Ví dụ 3: Tình huống 4

Xóa node 31



Xoá phần tử có key = x

```
TreeNode* delete(int x, TreeNode *root) {
    TreeNode *tmp;
    if (root == NULL) printf("Not found\n");
    else if (x < root->key) root->left = delete(x, root->left); /* đi bên trái */
    else if (x > root->key) root->right = delete(x, root->right); /* đi bên phải */
    else /* tìm được phần tử cần xoá */
        if (root->left != NULL && root->right != NULL)
        {
            /* Tình huống 4: phần tử thê chõ là phần tử min ở cây con phải */
            tmp = find_min(root->right);
            root->key = tmp->key;
            root->right = delete(root->key, root->right);
        }
        else /*TH1,2,3: có 1 con hoặc không có con */
        {
            tmp = root;
            if (root->left== NULL) /* chỉ có con phải hoặc không có con */
                root = root->right;
            else if (root->right == NULL) /* chỉ có con trái */
                root = root->left;
            free(tmp);
        }
    return (root);
}
```

Tình huống 1: Nút cần xoá là lá

Thao tác: chỉ cần thiết lập con trỏ con của node cha về NULL.

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

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

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.

Sắp xếp

Sắp xếp dãy sau nhờ sử dụng BST

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

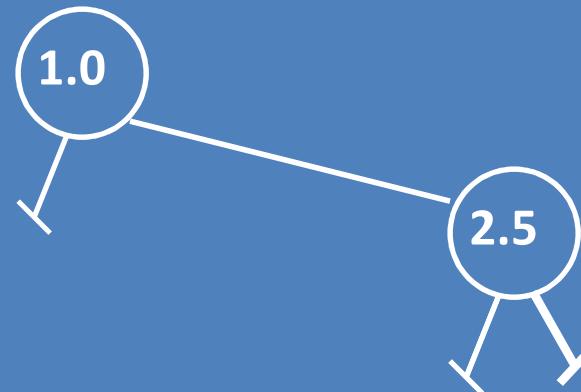
Sort



Sắp xếp dãy sau nhờ sử dụng BST

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

Sort



Sắp xếp dãy sau nhờ sử dụng BST

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

Sort



Sắp xếp dãy sau nhờ sử dụng BST

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

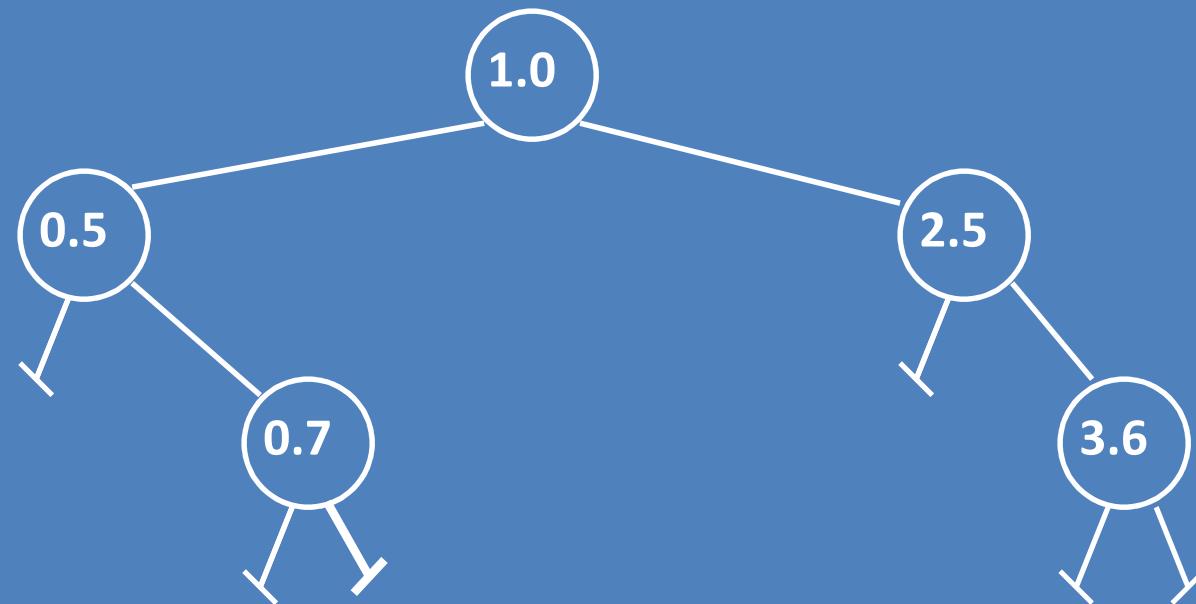
Sort



Sắp xếp dãy sau nhờ sử dụng BST

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

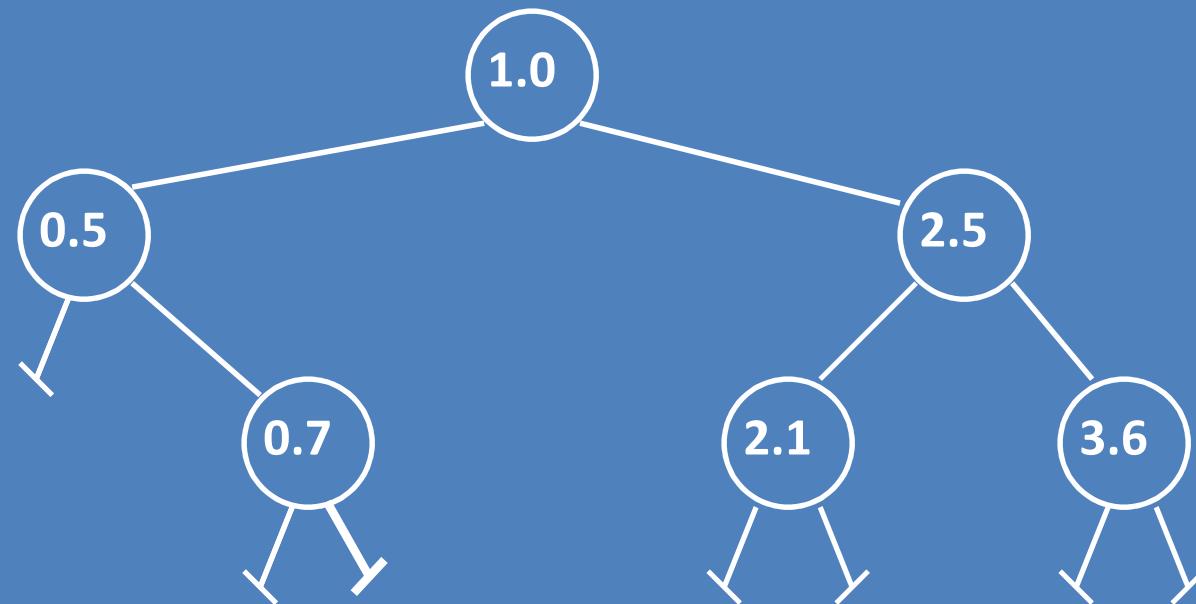
Sort



Sắp xếp dãy sau nhờ sử dụng BST

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

Sort

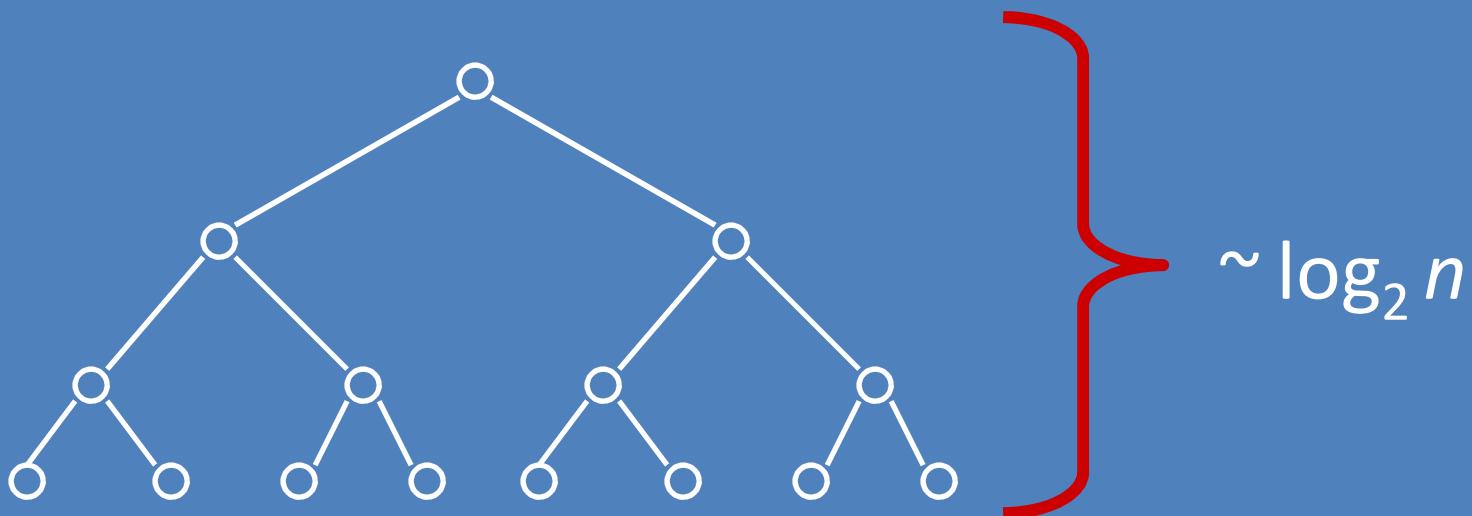


Sắp xếp dãy sau nhờ sử dụng BST

1.0	2.5	0.5	0.7	3.6	2.1
-----	-----	-----	-----	-----	-----

Sorting: Phân tích hiệu quả

- Tình huống trung bình: $O(n \log_2 n)$

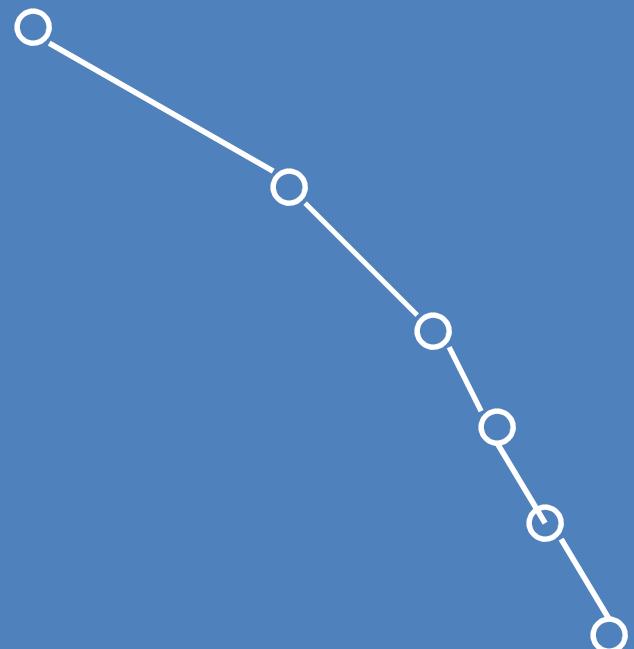


- Chèn phần tử thứ $(i+1)$ tốn quãng $\log_2(i)$ phép so sánh

Sorting: Phân tích hiệu quả

- Tình huống tồi nhất: $O(n^2)$

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
$$h = O(\log_2 n)$$
- Từ đó suy ra độ phức tạp trung bình của các thao tác với BST là:

Insertion	$O(\log_2 n)$
Deletion	$O(\log_2 n)$
Find Min	$O(\log_2 n)$
Find Max	$O(\log_2 n)$
BST Sort	$O(n \log_2 n)$

Độ phức tạp của các thao tác với BST

- Như đã biết, cây nhị phân có n nút có độ cao tối thiểu là $\log_2 n$. 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 $O(\log_2 n)$:
 - 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 \[Tarjan\]](#))

Nội dung

1. Tìm kiếm tuần tự
2. Tìm kiếm nhị phân
3. Cây nhị phân tìm kiếm
- 4. Bảng băm**
5. Tìm kiếm xâu mẫu

4. Bảng băm

4.1. Đặt vấn đề

4.2. Địa chỉ trực tiếp

4.3. Hàm băm

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

4. Bảng băm

4.1. Đặt vấn đề

4.2. Địa chỉ trực tiếp

4.3. Hàm băm

Địa chỉ trực tiếp (Direct Addressing)

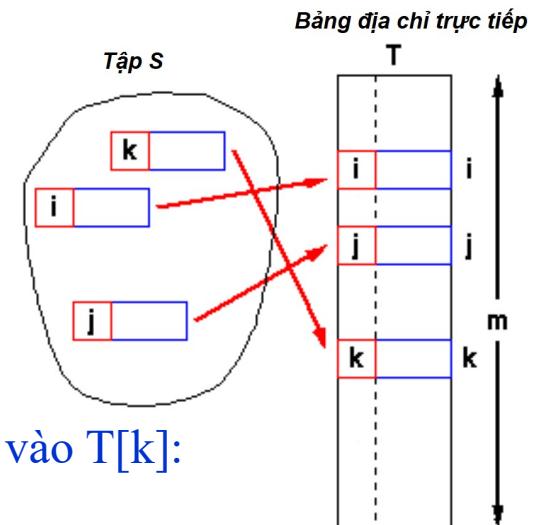
Giả sử tập S gồm n phần tử, trường khóa **key** của mỗi phần tử x :

- là các số trong khoảng từ 0 đến $m-1$ ($m \geq n$)
- các khoá là khác nhau từng đôi

Ta có thể lưu trữ n phần tử này trong một 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.



Searching: Nếu cần tìm kiếm phần tử có khóa $= k$, ta chỉ cần truy cập vào $T[k]$:

- Nếu $T[k]$ khác $\text{NULL} \rightarrow$ return phần tử chứa trong $T[k]$
 - Nếu $T[k] = \text{NULL} \rightarrow$ return NULL ;
- ➔ Tốn $O(1)$

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)

$T[key[x]] = \text{NULL}$

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

Ví dụ: Dictionary (từ điển)

- Ta cần tìm cấu trúc dữ liệu hỗ trợ hiệu quả hai thao tác được thực hiện thường xuyên trên từ điển:
 - Thêm một phần tử mới có khóa x vào từ điển insert (T, x)
 - Tìm kiếm một phần tử có khóa x trong từ điển search (T, x)

	Insert	Search
direct addressing	$O(1)$	$O(1)$
ordered array	$O(N)$	$O(\lg N)$
unordered array	$O(1)$	$O(N)$
ordered list	$O(N)$	$O(N)$
balance search tree	$O(\lg N)$	$O(\lg N)$

Hạn chế của phương pháp địa chỉ trực tiếp

- Tốn bộ nhớ khi: số phần tử $n \ll m$ (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*)

4. Bảng băm

4.1. Đặt vấn đề

4.2. Địa chỉ trực tiếp

4.3. Hàm băm

4.3. 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 .

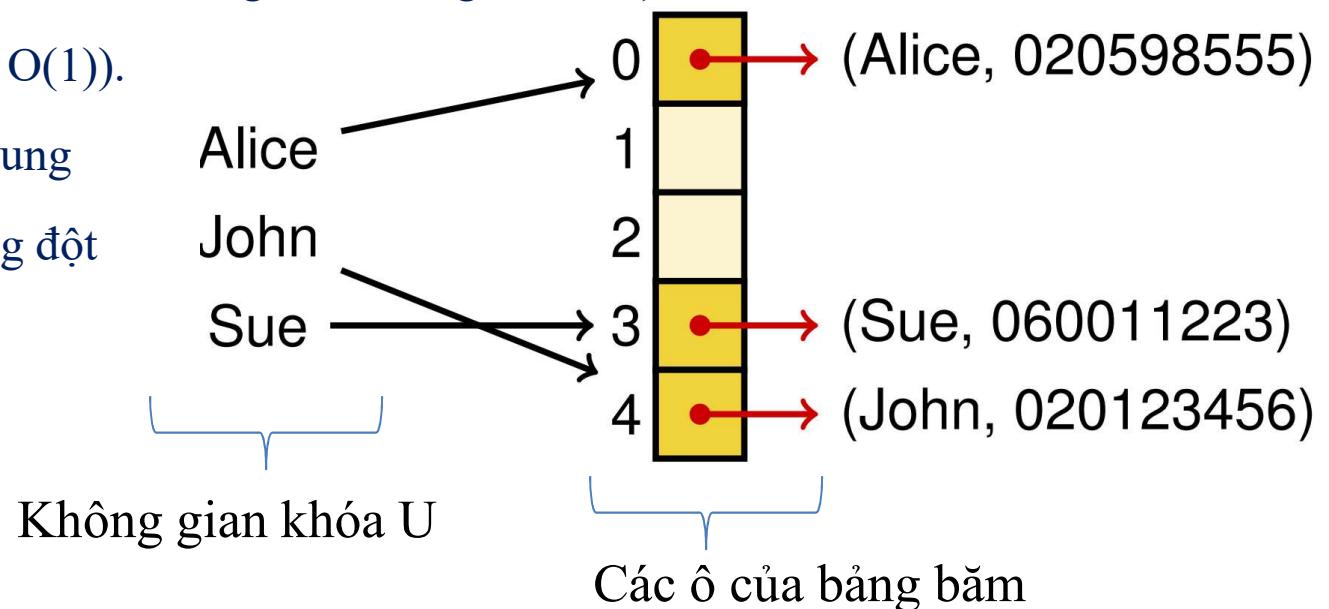
Ví dụ: Số điện thoại có kích thước $N = 5$ lưu trữ (họ tên, số điện thoại) với họ tên là khoá. Hàm băm $h(k) = (\text{độ dài của xâu biểu diễn tên người k trong danh bạ}) \bmod 5$.

Lý tưởng: Tìm Search(k) (tốn $O(1)$).

Nảy sinh vấn đề: nếu cần bổ sung

Thêm Joe vào danh bạ \rightarrow xung đột

với Sue đã có trong danh bạ?



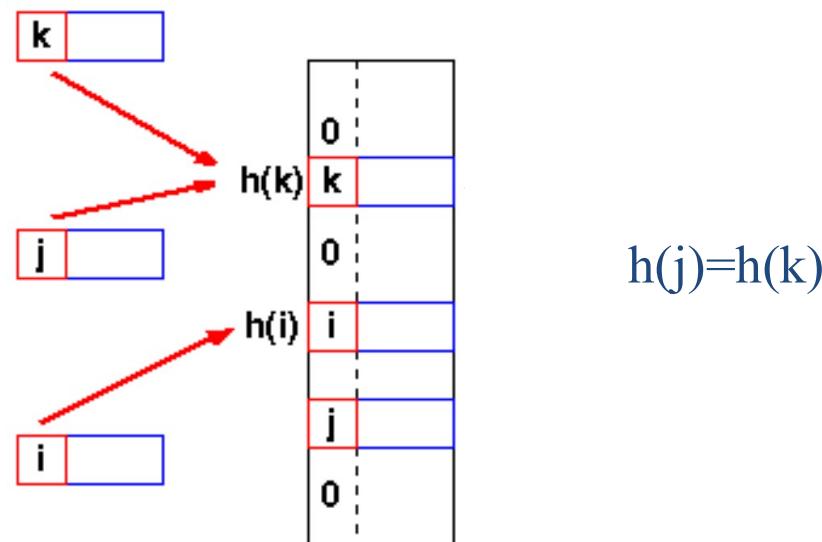
4.3. Hàm băm

- 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 .

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



4.3. Hàm băm: Giải quyết xung đột

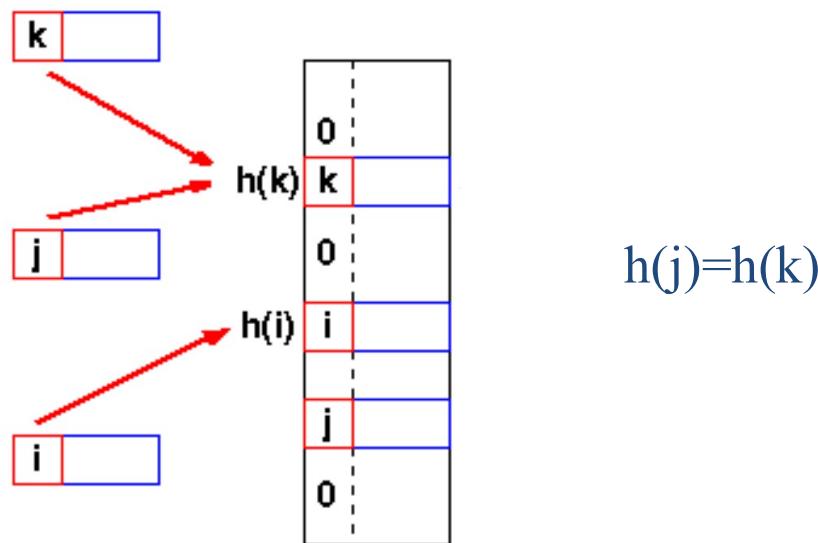
Ta cần giải quyết xung đột như thế nào?

- Cách giải quyết 1:

Tạo chuỗi (chaining)

- Cách giải quyết 2:

Phương pháp địa chỉ mở (open addressing)



Giải quyết xung đột: (1) Tạo chuỗi (Chaining)

- Theo phương pháp này, ta sẽ tạo danh sách mốc nối để chứa các phần tử được gắn với cùng một vị trí trong bảng.
- Ta cần thực hiện bổ sung phần tử như thế nào?

(Như bổ sung vào danh sách mốc nối)

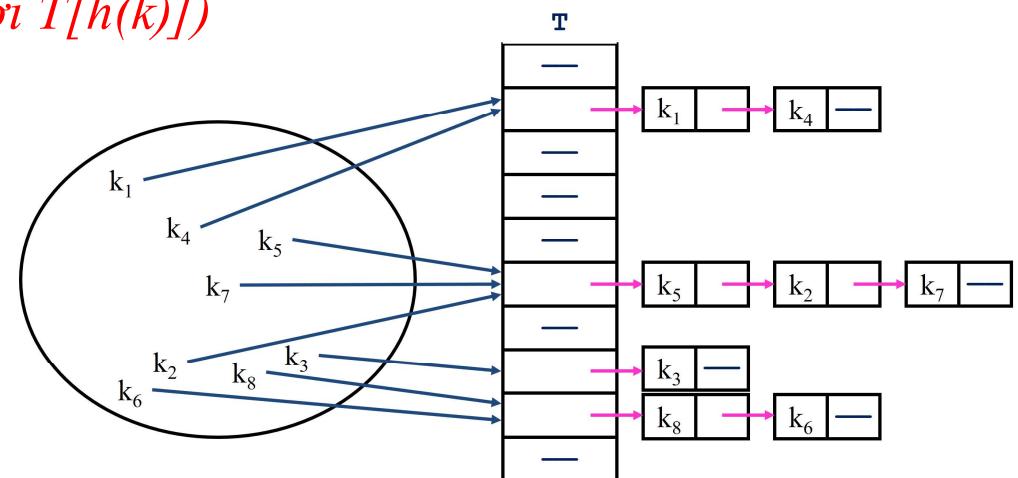
Ví dụ: $h(k) = k \bmod m$. Nếu ô $T[h(k)]$ trong bảng băm đã bận, ta thêm phần tử mới này vào đầu danh sách mốc nối tại ô $T[h(k)]$

- Ta cần thực hiện loại bỏ phần tử như thế nào? Có cần sử dụng danh sách mốc nối đôi để thực hiện xoá một cách hiệu quả không?

(Không! Vì thông thường chuỗi có độ dài không lớn)

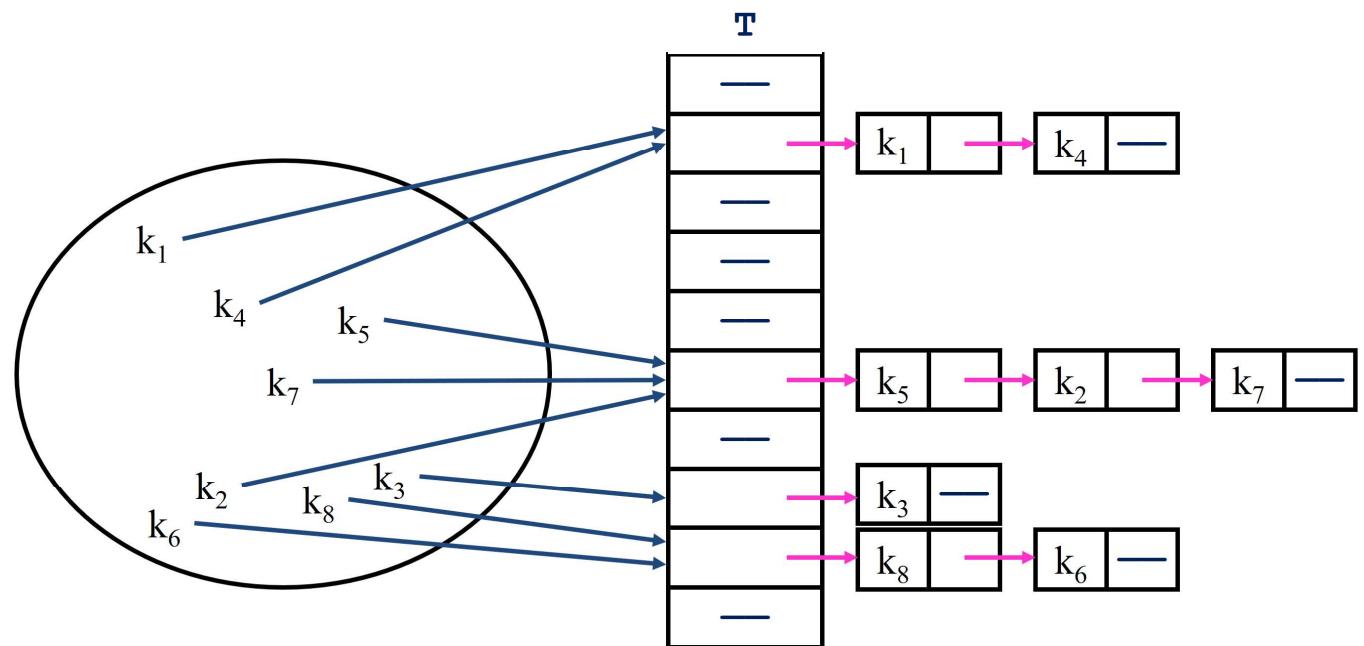
- Thực hiện tìm kiếm phần tử với khoá k cho trước như thế nào?

(Tìm kiếm trên danh sách mốc nối trả bởi $T[h(k)]$)



Tạo chuỗi (Chaining): Các thao tác

- CHAINED-HASH-INSERT (T, x)
chèn x vào đầu danh sách mốc 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 $T[h(key[x])]$



Lợi ích của phương pháp chuỗi

- Nếu số lượng mẫu tin lớn: tiết kiệm vùng nhớ.
- Giải quyết đụng độ: đơn giản là đẩy vào cùng một danh sách liên kết.
- Bảng hash nhỏ hơn nhiều so với số lượng mẫu tin.
- Xóa một phần tử là đơn giản và nhanh chóng.
- Độ phức tạp khi tìm kiếm:
 - Nếu có n khóa, và bảng hash có kích thước $m \rightarrow$ Độ dài trung bình của danh sách liên kết là n/m . (Ta định nghĩa nhân tử nạp (*load factor*) $\alpha = n/m =$ Số lượng khoá trung bình trên một ô)
 - 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. 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 $O(1+\alpha)$

Giải quyết xung đột: (2) 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 dây các phép dò thử giống như dây 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

Giải quyết xung đột: (2) Địa chỉ mở (Open Addressing)

Xét chi tiết một số kỹ thuật:

1. Dò tuyến tính (Linear Probing)

Tăng chỉ số lên một: $h(k, i) = (h'(k)+i) \% m$ với $0 \leq i \leq m-1$

2. Dò bậc hai (Quadratic Probing)

Tăng chỉ số lên theo bình phương: $h(k, i) = (h'(k) + i^2)\% m$ với $0 \leq i \leq m-1$

3. Hàm băm kép (Double Hashing)

Dò tuyển tính (Linear Probing)

Nếu vị trí hiện tại đã bận, ta dò kiểm vị trí tiếp theo trong bảng:

$$h(k, i) = (h'(k) + i) \% m \text{ với } 0 \leq i \leq m-1$$

Đầu tiên thử $h(k, 0) = h'(k)$, nếu vị trí này bận, thử tiếp $h(k, 1) \dots$ cho đến khi tìm được ô trống.

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

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

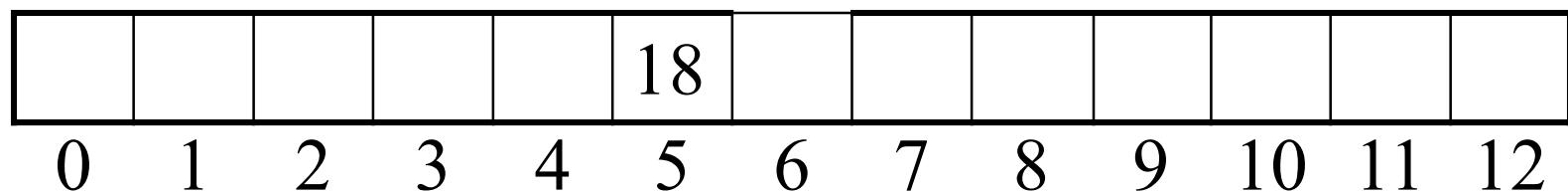
- **Ưu điểm:** Đòi hỏi bộ nhớ ít hơn phương pháp tạo chuỗi (không có móc nối)
- **Hạn chế:** Đòi hỏi nhiều thời gian hơn tạo chuỗi (nếu đường dò kiểm là dài)
- Thực hiện xoá bằng cách đánh dấu xoá (đánh dấu ô đã bị xoá)

Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5,

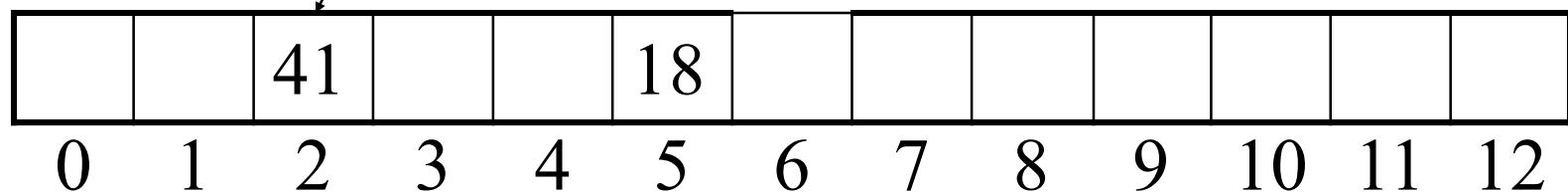


Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

$h(k)$: 5, 2,

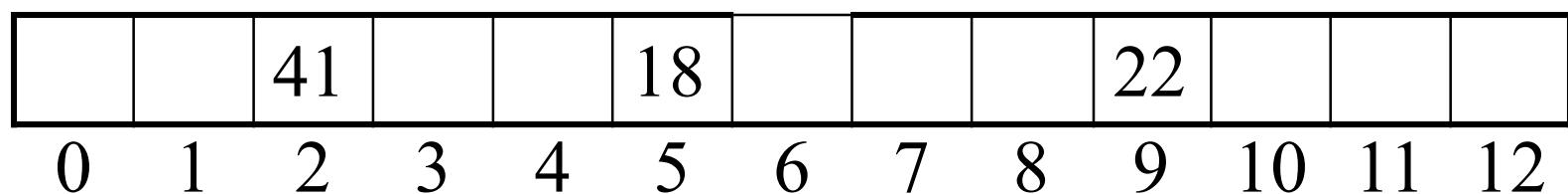


Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

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

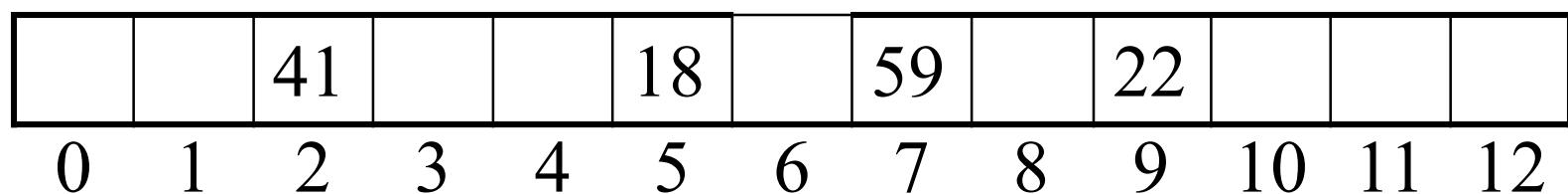


Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

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

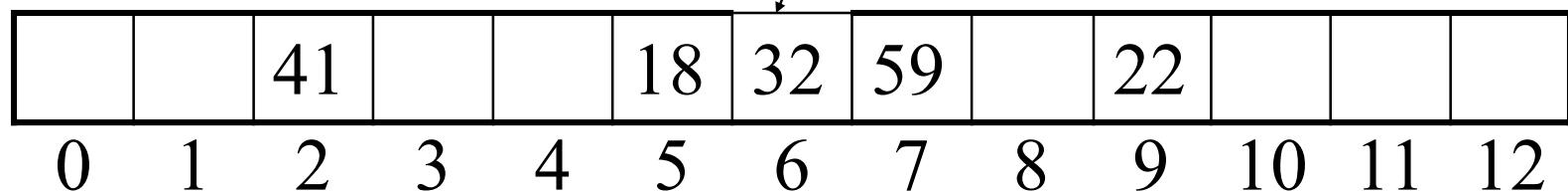


Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

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

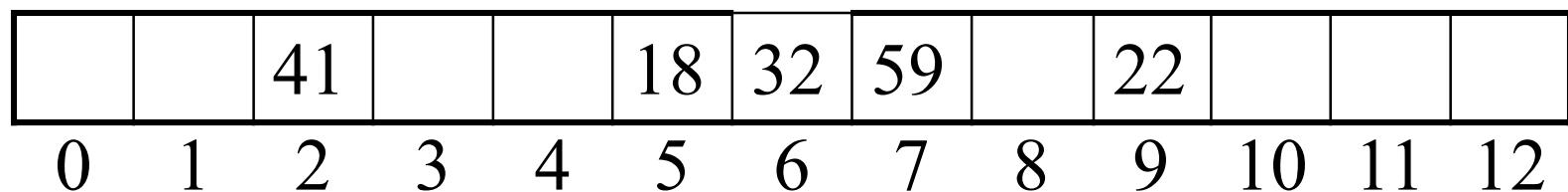


Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

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

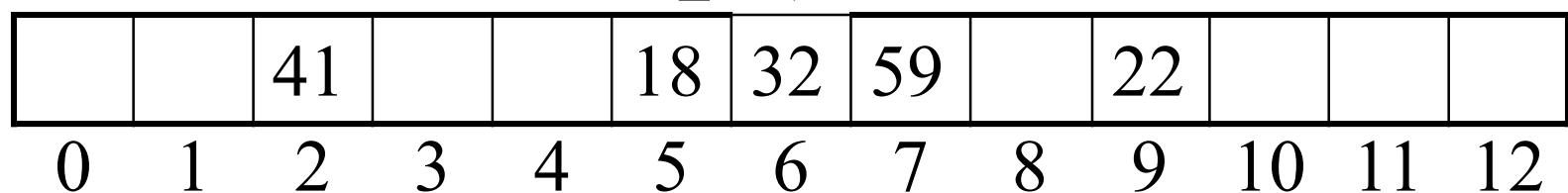


Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

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

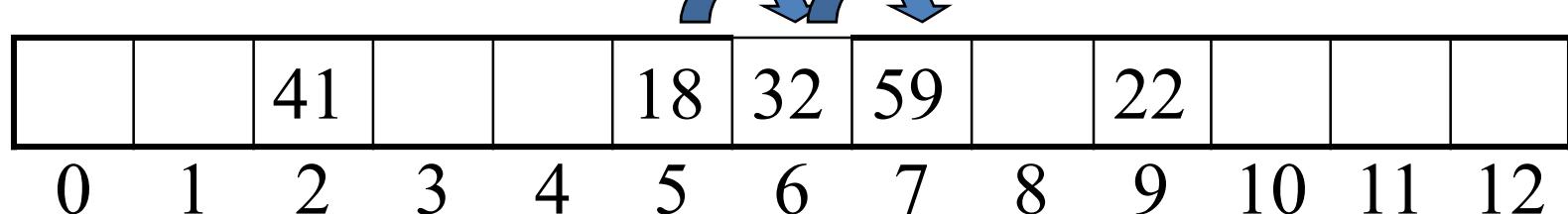


Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

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

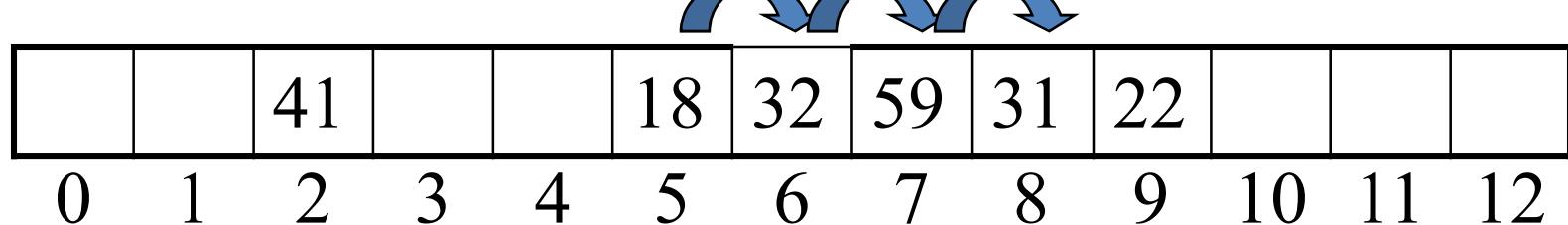


Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

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

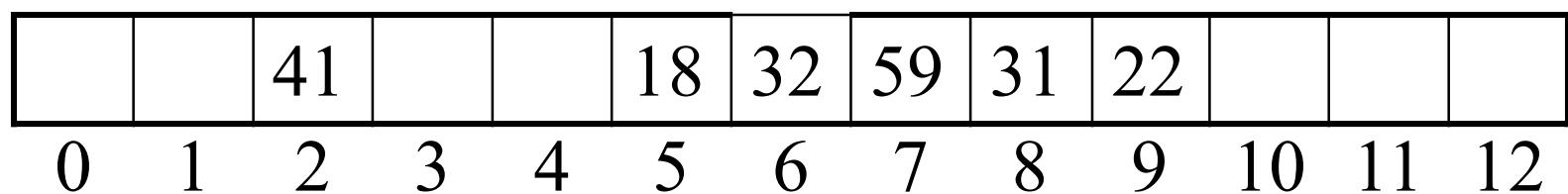


Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

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

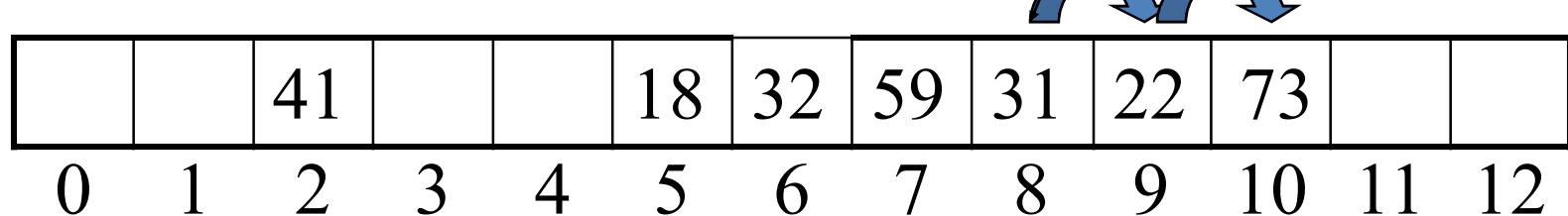


Linear Probing

Sử dụng hàm băm: $h'(k) = k \% 13$

Chèn: 18, 41, 22, 59, 32, 31, 73

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



Giải quyết xung đột: (2) Địa chỉ mở (Open Addressing)

Xét chi tiết một số kỹ thuật:

1. Dò tuyến tính (Linear Probing)

Tăng chỉ số lên một: $h(k, i) = (h'(k) + i) \% m$ với $0 \leq i \leq m-1$

2. Dò bậc hai (Quadratic Probing)

Tăng chỉ số lên theo bình phương: $h(k, i) = (h'(k) + i^2) \% m$ với $0 \leq i \leq m-1$

3. Hàm băm kép (Double Hashing)

$$h(k, i) = (h_1(k) + i h_2(k)) \% m$$

với $h_1(k)$ và $h_2(k)$ là hai hàm băm

Hàm băm $h_2(k)$ được coi là step function

Giải quyết xung đột: (2) Địa chỉ mở (Open Addressing): dò bậc 2

Tăng chỉ số lên theo bình phương: $h(k, i) = (h'(k) + i^2) \% m$ với $0 \leq i \leq m-1$

$$h'(k) = k \bmod m$$

Lần băm 0: $h'(k) \bmod m$

Lần băm 1: $(h'(k) + 1) \bmod m$

Lần băm 2: $(h'(k) + 4) \bmod m$

Lần băm 3: $(h'(k) + 9) \bmod m$

...

Lần băm i: $(h'(k) + i^2) \bmod m$

Giải quyết xung đột: (2) Địa chỉ mở (Open Addressing): dò bậc 2

Sử dụng hàm băm: $h'(k) = k \% 7$

Chèn: 76, 40, 48, 5, 55, 47, 80

$h'(k)$: 6, 5, 6, 5, 6, 5, 8

48		5	55		40	76
0	1	2	3	4	5	6

Giải quyết xung đột: (2) Địa chỉ mở (Open Addressing): dò bậc 2

Nếu m là số nguyên tố và $\alpha < \frac{1}{2}$

với $\alpha = \langle \text{số lượng phần tử trong bảng băm} \rangle / \langle \text{kích thước bảng băm} \rangle$,
thì dò bậc 2 sẽ tìm được ô trống trong bảng băm sau nhiều nhất $m/2$ lần băm, vì
mỗi lần băm sẽ cho một giá trị băm khác nhau:

- Chứng minh rằng $0 \leq i, j \leq m/2$ và $i \neq j$:
 $(h(x) + i^2) \bmod m \neq (h(x) + j^2) \bmod m$

Chứng minh bằng phản chứng: giả sử tìm được $i \neq j$ sao cho:

$$(h(x) + i^2) \bmod m = (h(x) + j^2) \bmod m$$

$$\Rightarrow i^2 \bmod m = j^2 \bmod m$$

$$\Rightarrow (i^2 - j^2) \bmod m = 0$$

$$\Rightarrow [(i+j)(i-j)] \bmod m = 0$$

Vì m là số nguyên tố nên $(i-j)=0$ hoặc $(i+j)=0$, điều này không thể xảy ra vì giả thiết rằng $0 \leq i, j$ và $i \neq j$.

Kết luận: Với $\alpha < \frac{1}{2}$, dò bậc hai luôn tìm ra vị trí rỗng trong bảng băm, với $\alpha \geq \frac{1}{2}$, dò bậc hai có khả năng không tìm ra vị trí rỗng trong bảng băm

Giải quyết xung đột: (2) Địa chỉ mở (Open Addressing)

Xét chi tiết một số kỹ thuật:

1. Dò tuyến tính (Linear Probing)

Tăng chỉ số lên một: $h(k, i) = (h'(k) + i) \% m$ với $0 \leq i \leq m-1$

2. Dò bậc hai (Quadratic Probing)

Tăng chỉ số lên theo bình phương: $h(k, i) = (h'(k) + i^2) \% m$ với $0 \leq i \leq m-1$

3. Hàm băm kép (Double Hashing)

$$h(k, i) = (h_1(k) + i h_2(k)) \% m$$

với $h_1(k)$ và $h_2(k)$ là hai hàm băm

Hàm băm $h_2(k)$ được coi là step function

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);
    step = h2(k);
    while (table[probe] is occupied)
        probe = (probe + step) % 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: $h(k,i) = (h_1(k) + i h_2(k)) \% m$

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

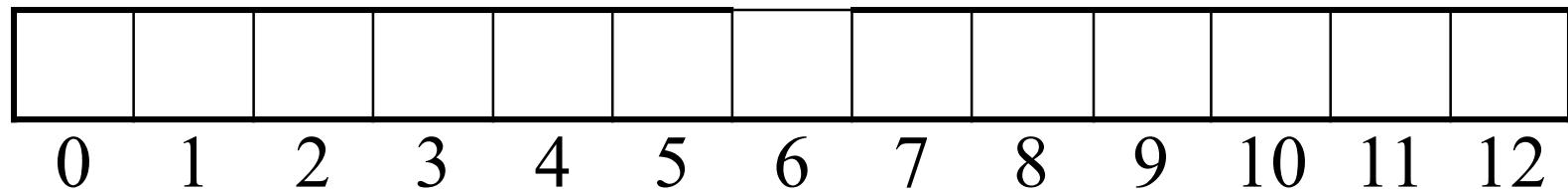
step = $h_2(k) = 8 - (k \% 8)$

probe = (probe+step) % 13;

Chèn: 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: $h(k,i) = (h_1(k) + i h_2(k)) \% m$

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

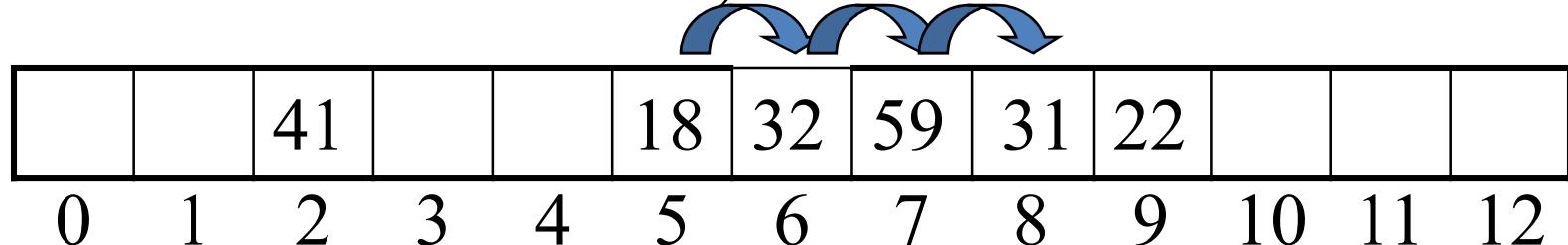
step = $h_2(k) = 8 - (k \% 8)$

probe = (probe+step) % 13;

Chèn: 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: $h(k,i) = (h_1(k) + i h_2(k)) \% m$

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

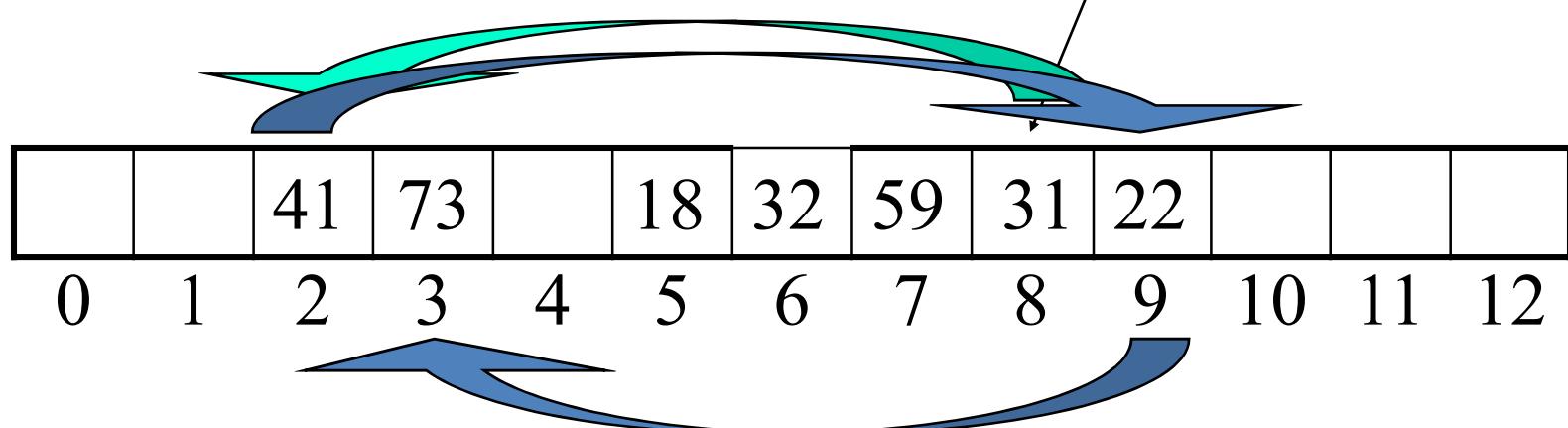
step = $h_2(k) = 8 - (k \% 8)$

probe = (probe+step) % 13;

Chèn: 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

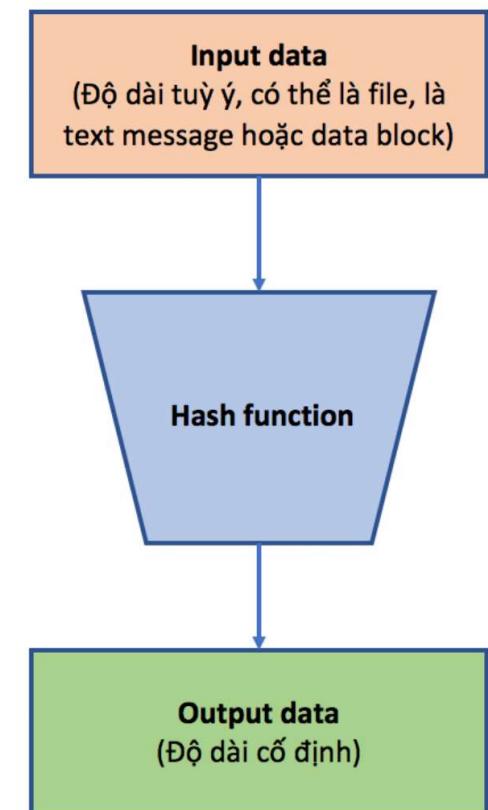
Giả sử rằng thực hiện điều kiện *simple uniform hashing*: Mỗi khoá trong bảng băm là đồng khả năng được gán với một ô bất kỳ. Khi đó ta có:

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

$$\alpha = \langle \text{số lượng phần tử trong bảng băm} \rangle / \langle \text{kích thước bảng băm} \rangle$$

Một số ứng dụng của hàm băm

- Hàm băm (hash function) là hàm nhận một dữ liệu đầu vào, từ dữ liệu đó tạo ra một giá trị đầu ra (hay còn gọi là giá trị băm - hash value) tương ứng. Giá trị đầu vào có thể có độ dài tùy ý nhưng giá trị băm thì luôn có độ dài cố định, nhờ vậy có thể đơn giản hóa việc tìm kiếm dữ liệu.



Một số ứng dụng của hàm băm

- **Một hàm băm tốt phải thỏa mãn các điều kiện sau:**
 - Tính toán nhanh
 - Các khoá được phân bổ đều trong bảng.
 - Ít xảy ra đụng độ, tức là khả năng để các giá trị đầu vào khác nhau cho ra cùng một giá trị băm là rất thấp
 - Xử lý được các loại khóa có kiểu dữ liệu khác nhau
 - Không thể đảo ngược: Đảm bảo không có phương pháp khả thi để tính toán được dữ liệu vào nào đó để cho ra giá trị băm mong muốn:
tức là nếu nhận được đầu ra y , ta không thể tính được đầu vào x sao cho $h(x) = y$

Một số ứng dụng của hàm băm

Hàm băm được ứng dụng chính vào một số công việc sau:

- Xây dựng cấu trúc dữ liệu để tối ưu việc tìm kiếm: Hashtable
- Lưu và kiểm tra password

Ví dụ trên máy tính của chúng ta lưu trữ $h(PW)$, chứ không lưu trữ PW. Khi người dùng nhập PW0, tính $h(PW0)$ và so sánh với $h(PW)$ → cần đảm bảo tính chất “Không thể đảo ngược”

- Kiểm tra tính toán vẹn của dữ liệu
- Sinh mã OTP
- Chữ ký điện tử

Một số ứng dụng của hàm băm

Ví dụ Hàm băm MD5 (MD là viết tắt của Message Digest):

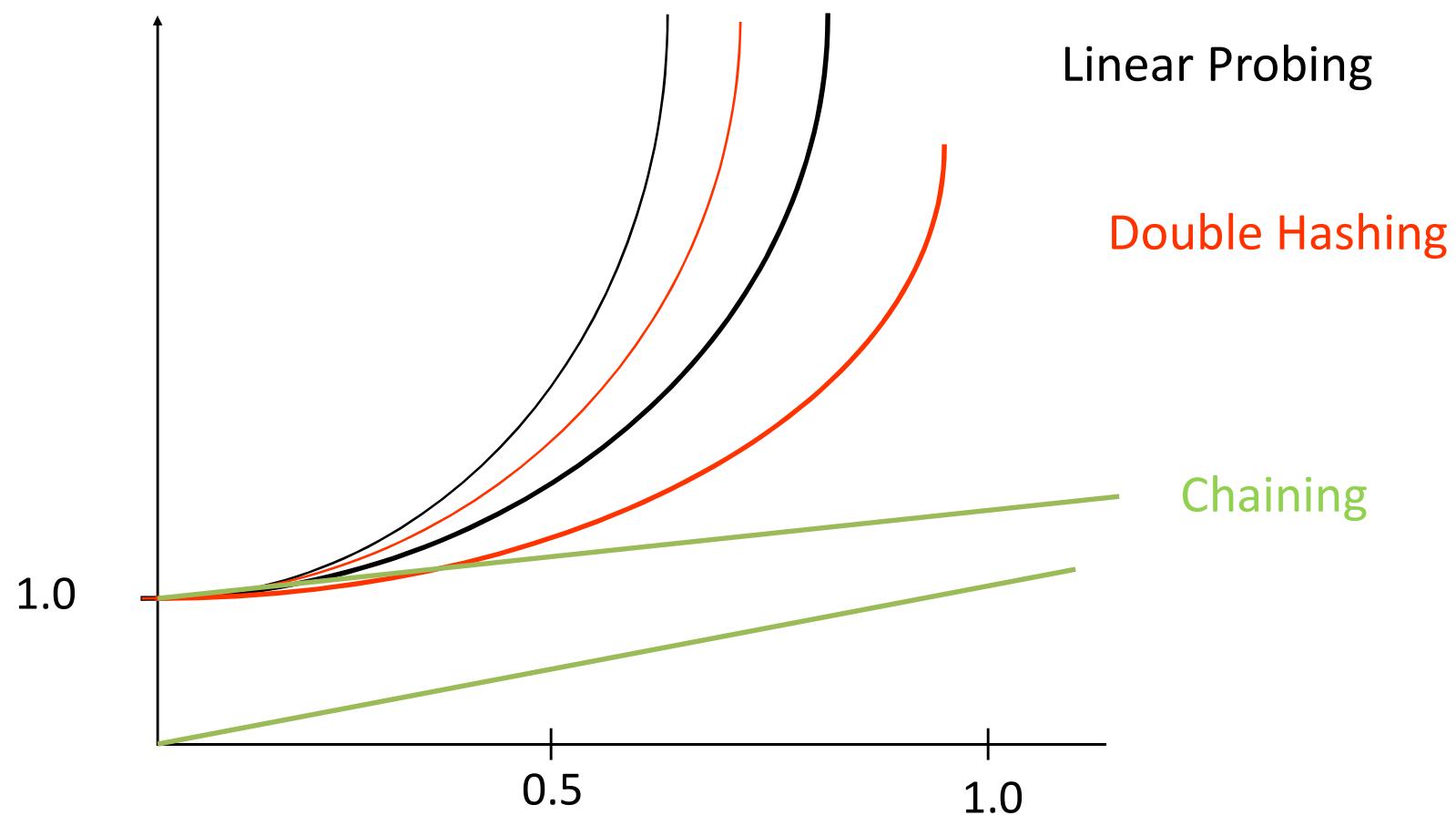
- Là hàm băm 128-bit (dùng 128 bit để lưu giá trị băm), giá trị băm được biểu diễn bằng chuỗi 32 ký tự mã 16.
- MD5 được sử dụng rộng rãi để kiểm tra tính toán vẹn khi truyền file.

Ví dụ: quá trình truyền file trên mạng

- file server cung cấp giá trị băm bằng MD5 cho file (gọi là mã checksum) trước khi truyền,
- người dùng sau khi download file xuống có thể dùng tool tính lại mã checksum của file đó và so sánh với mã checksum mà server cung cấp. Nếu 2 mã là giống nhau thì có nghĩa là file download xuống giống với file gốc, không bị chỉnh sửa, giả mạo hay attach virus.

Tuy nhiên vào năm 2004, “đụng độ” trong thuật toán MD5 đã được phát hiện ra. Một nghiên cứu cho thấy chỉ mất vài tiếng đồng hồ sử dụng một mạng lưới gồm nhiều máy tính có thể cho ra được giá trị input để cho ra được giá trị băm mong muốn. Chính vì vậy MD5 bây giờ không còn được khuyến khích sử dụng đối với những hệ thống hoặc chức năng yêu cầu security cao.

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 khóa vào các ô của bảng băm
 - Hạn chế tối đa tình huống các khóa gần giống nhau bị phân bố vào cùng ô của bảng băm
 - Ví dụ: khóa là các xâu kí tự gần giống nhau như *stop, tops, pots* cần có giá trị băm khác nhau để phân bố vào các ô khác nhau trong bảng băm
 - 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 khóa k vào bảng băm có m ô nhò sử dụng ô xác định bởi phần dư của phép chia k cho m

- *Ưu điểm: nhanh, tính giá trị băm chỉ cần 1 phép toán*
- *Điều gì xảy ra nếu m là luỹ 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 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)

Key	m	
	97	100
16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

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 , $0 < A < 1$:

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

- Nhược điểm: chậm hơn phương pháp chia
- Ưu điểm: giá trị của m không quan trọng, thường chọn $m = 2^p$
- Chọn A không quá gần với 0 hoặc 1

Nội dung

1. Tìm kiếm tuần tự
2. Tìm kiếm nhị phân
3. Cây nhị phân tìm kiếm
4. Bảng băm
- 5. Tìm kiếm xâu mẫu**

5. Tìm kiếm xâu mẫu

5.1. Phát biểu bài toán

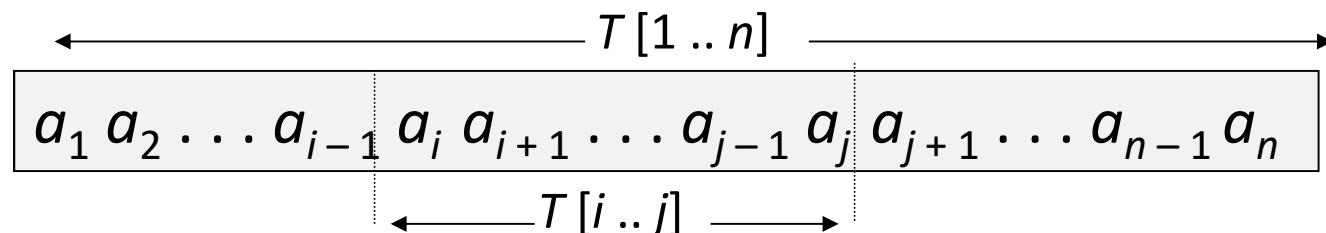
5.2. Thuật toán trực tiếp

5.3. Thuật toán Boyer-Moore

5.4. Thuật toán Knuth-Morris-Pratt (KMP)

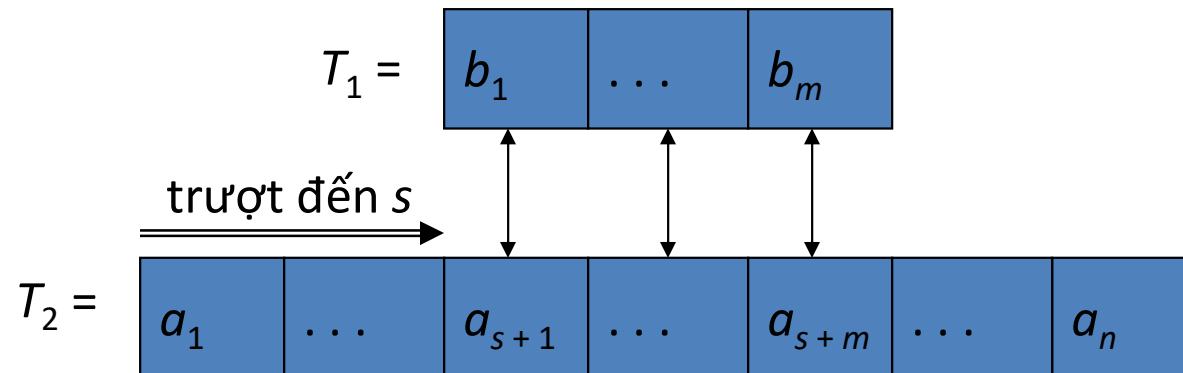
Xâu (Strings)

- Xâu là dãy ký hiệu lấy từ bảng ký hiệu (alphabet) Σ
- Ký hiệu $T[i..j]$ là xâu con của xâu T bắt đầu từ vị trí i và kết thúc ở vị trí j



Trượt/đẩy (Shifts)

- Giả sử T_1 và T_2 là 2 xâu, trong đó $|T_1| = m$ và $|T_2| = n$
- Ta nói T_1 xuất hiện nhò trượt (đẩy) đến s trong T_2 nếu
 $T_1[1 .. m] = T_2[s + 1 .. s + m]$



và s được gọi là **vị trí khớp** của T_1 trong T_2
ngược lại s được gọi là **vị trí không khớp**

Bài toán tìm kiếm xâu mẫu (The String Matching Problem)

- Cho xâu T độ dài n
 - T được gọi là văn bản
- Cho xâu P độ dài m
 - P được gọi là xâu mẫu (*pattern*)
- **Bài toán: Tìm tất cả các vị trí khớp của P trong T**
- **Ứng dụng:**
 - trong thu thập thông tin (information retrieval)
 - trong soạn thảo văn bản (text editing)
 - trong tính toán sinh học (computational biology)
 - ...

Ví dụ

T = 000010001010001 và xâu mẫu P = 0001, các vị trí khớp là:

➤ s = 1

➤ T = 0**0001**0001010001

➤ P = 0001

➤ s = 5

➤ T = 0**0001**000**1**010001

➤ P = 0001

➤ s = 11

➤ T = 0**0001**000101**0001**

➤ P = 0001

5. Tìm kiếm xâu mẫu

5.1. Phát biểu bài toán

5.2. Thuật toán trực tiếp

5.3. Thuật toán Boyer-Moore

5.4. Thuật toán Knuth-Morris-Pratt (KMP)

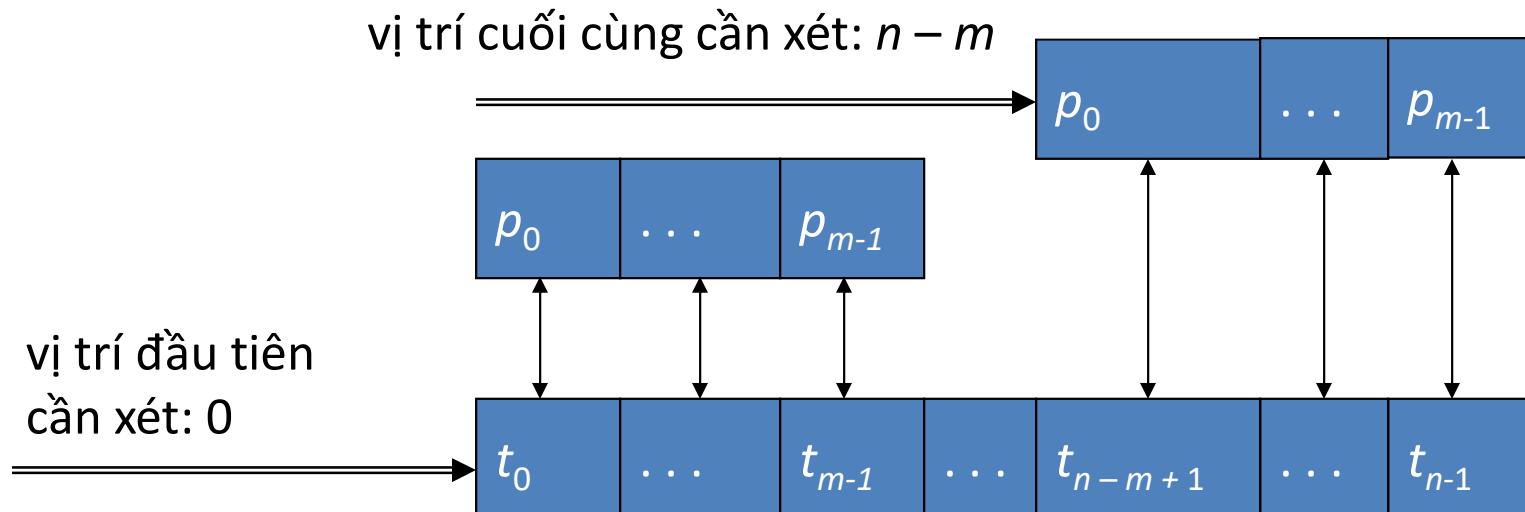
5.2. Thuật toán trực tiếp (Naïve algorithm)

▪ Bài toán:

- Cho xâu T độ dài n : $t_0 \ t_1 \ \dots \ t_{n-1}$
- Cho xâu mẫu P độ dài m : $p_0 \ p_1 \ \dots \ p_{m-1}$

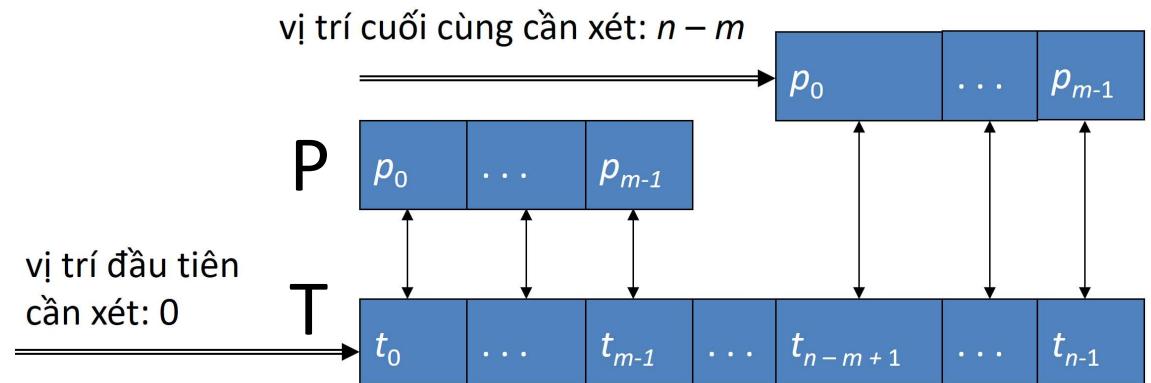
Tìm tất cả các vị trí khớp của P trong T

- **Ý tưởng:** Dịch chuyển từng vị trí $s = 0, 1, \dots, n-m$, với mỗi vị trí kiểm tra xem xâu mẫu có xuất hiện ở vị trí đó hay không.



5.2. Thuật toán trực tiếp (Naïve algorithm)

- Ý tưởng: Dịch chuyển từng vị trí $s = 0, 1, \dots, n-m$, với mỗi vị trí kiểm tra xem xâu mẫu có xuất hiện ở vị trí đó hay không.



```
void NaiveSM(char *P, int m, char *T, int n) {  
    int i, j;  
    /* Searching */  
    for (s = 0; s <= n - m; ++s) {  
        for (i = 0; i < m && P[i] == T[i + s]; ++i);  
        if (i==m) print "s là vị trí khớp"; //pattern xuất hiện tại vị trí s  
    }  
}
```

Thời gian tính trong tình huống tồi nhất = $O((n-m+1)m)$

5.2. Thuật toán trực tiếp (Naïve algorithm)

```
#include <bits/stdc++.h>
using namespace std;

void NaiveSM(char* pat, int m, char* txt, int n)
{
    for (int s = 0; s <= n - m; s++)
    {
        int i;
        /* For current index s, check for pattern match */
        for (i = 0; i < m && pat[i]==txt[i+s]; i++);
        if (i == m) // if pat[0...M-1] = txt[i, i+1, ...i+M-1]
            cout << "Pattern xuat hien tai vi tri " << s << endl; //s la vi tri khop
    }
}

int main()
{
    char txt[] = "AABAACAAADAABAAABAA";
    char pat[] = "AABA";
    int m = strlen(pat);
    int n = strlen(txt);
    NaiveSM(pat,m, txt,n);
    return 0;
}
```

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A

A A B A	A A B A
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	A A B A

```
Pattern xuat hien tai vi tri 0
Pattern xuat hien tai vi tri 9
Pattern xuat hien tai vi tri 13
```

5.2. Thuật toán trực tiếp

- Ví dụ: $T = 000010001010001$ và $P = 0001$.

	$T = 000010001010001$	
$s=0$	$P = 0001$	\Rightarrow không khớp
$s=1$	$P = 0001$	\Rightarrow khớp
$s=2$	$P = 0001$	\Rightarrow không khớp
$s=3$	$P = 0001$	\Rightarrow không khớp
$s=4$	$P = 0001$	\Rightarrow không khớp
$s=5$	$P = 0001$	\Rightarrow khớp
$s=6$	$P = 0001$	\Rightarrow không khớp
...		

5. Tìm kiếm xâu mẫu

5.1. Phát biểu bài toán

5.2. Thuật toán trực tiếp

5.3. Thuật toán Boyer-Moore

5.4. Thuật toán Knuth-Morris-Pratt (KMP)

5.3. Thuật toán Boyer-Moore

- Làm việc tốt khi P dài và Σ lớn
- Chúng ta sẽ xét sự khớp nhau bằng cách duyệt từ phải qua trái.
- Trước hết hãy quan sát lại thuật toán trực tiếp làm việc theo thứ tự duyệt này...



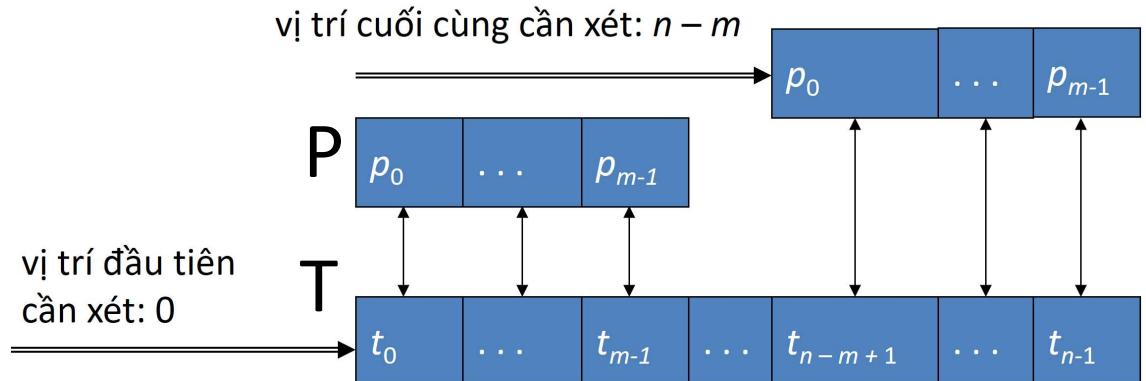
Robert-Stephen Boyer



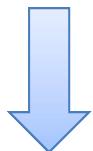
J. Strother Moore

Thuật toán trực tiếp cải biên

- Trước hết hãy quan sát lại thuật toán trực tiếp làm việc theo thứ tự duyệt từ trái sang phải:



```
void NaiveSM(char *P, int m, char *T, int n) {
    int i, j;
    /* Searching */
    for (s = 0; s <= n - m; ++s) {
        for (i = 0; i < m && P[i] == T[i + s]; ++i);
        if (i==m) print "s là vị trí khớp"; //pattern xuất hiện tại vị trí s
    }
}
```

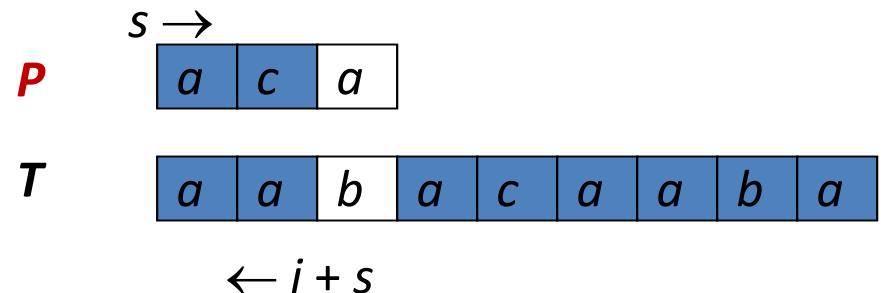


```
for (s = 0; s <= n - m; ++s) {
    for (i = m-1; i>=0 && P[i]==T[i+s];i--) ;
    if (i < 0) print "s là vị trí khớp";
}
```

So sánh từng kí tự từ trái sang phải

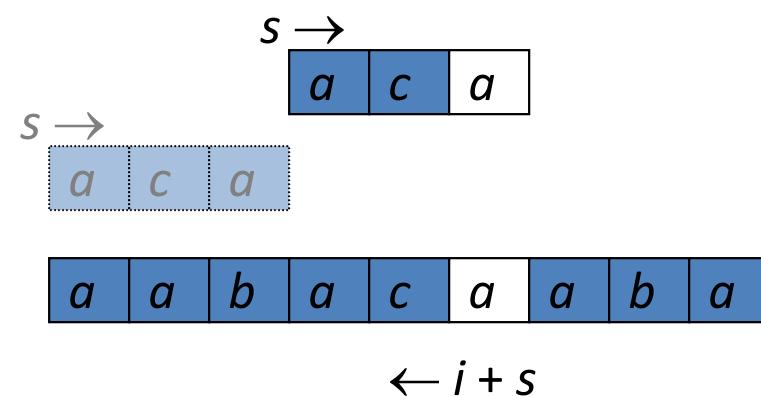
So sánh từng kí tự từ phải sang trái

Xét ví dụ



b không xuất hiện ở bất cứ đâu trong P vì thế có thể di chuyển P bỏ qua b

Bỏ qua được một đoạn



Xét ví dụ khác

$s \rightarrow$

P	a	c	b	a	b
----------	---	---	---	---	---

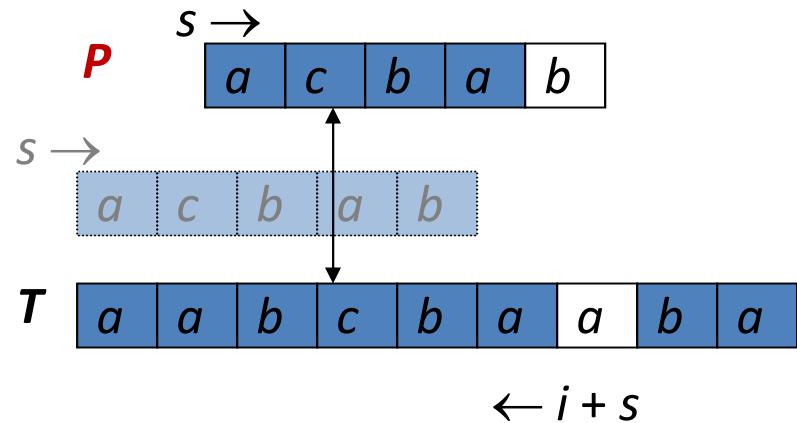
T	a	a	b	c	b	a	a	b	a
----------	---	---	---	---	---	---	---	---	---

$\leftarrow i + s$

c xuất hiện ở vị trí 2 trong P

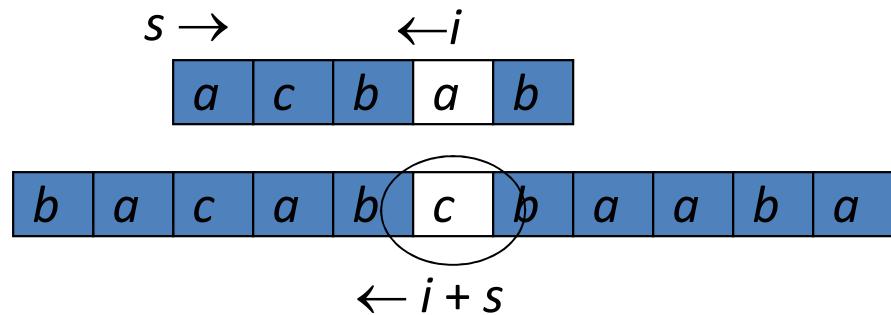
vì thế có thể di chuyển P sao cho các c được đóng thẳng

Bỏ qua được một đoạn



Ký tự tồi

Giả sử $P[i] \neq T[i + s]$



- Ta gọi $T[i + s]$ là ký tự tồi (**bad character**)
- Sử dụng ký tự tồi ta có thể tránh được việc dóng hàng không đúng

Hàm Last

- $T[i + s]$ xuất hiện ở vị trí nào trong xâu mẫu P ?

Ta xác định hàm **last** như sau:

Nếu c xuất hiện trong xâu mẫu P thì đặt

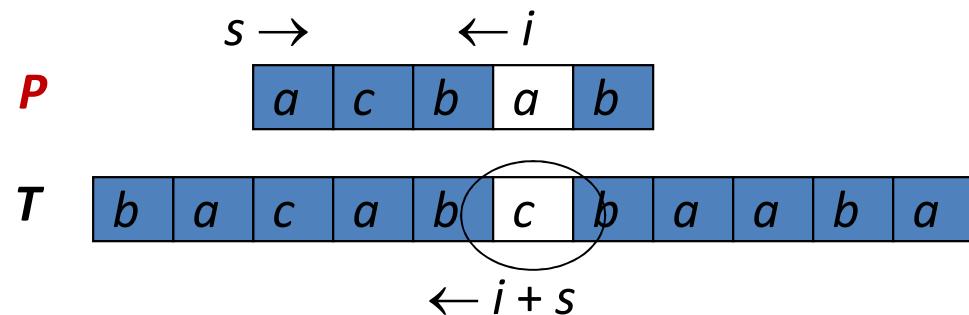
$\text{last}(c) = \text{chỉ số lớn nhất (bên phải nhất) của vị trí xuất hiện của } c \text{ trong } P$

Trái lại đặt

$\text{last}(c) = 0$

Tăng vị trí dịch chuyển

Giả sử ký tự tồi $c = T[i+s]$ được đóng ở vị trí i của P :

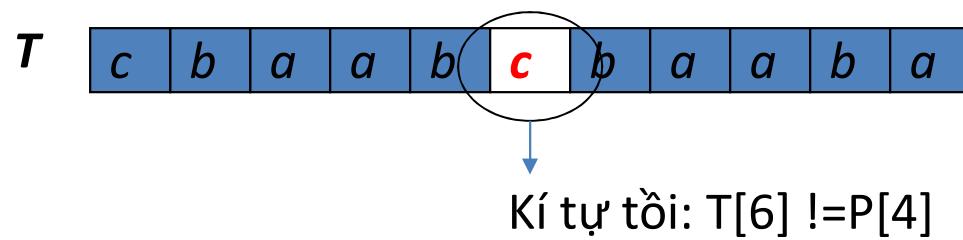
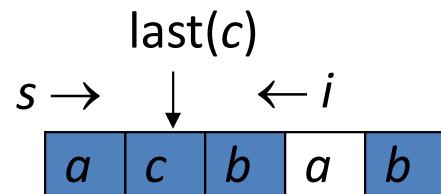


Khi đó chỉ có thể xảy ra 3 khả năng sau:

- 1) Ký tự tồi c có mặt trong xâu mẫu P và $\text{last}(c) < i$
- 2) Ký tự tồi có mặt trong xâu mẫu P và $\text{last}(c) > i$
- 3) Ký tự tồi không có mặt trong xâu mẫu P (tức là $\text{last}(c) = 0$)

Tình huống 1

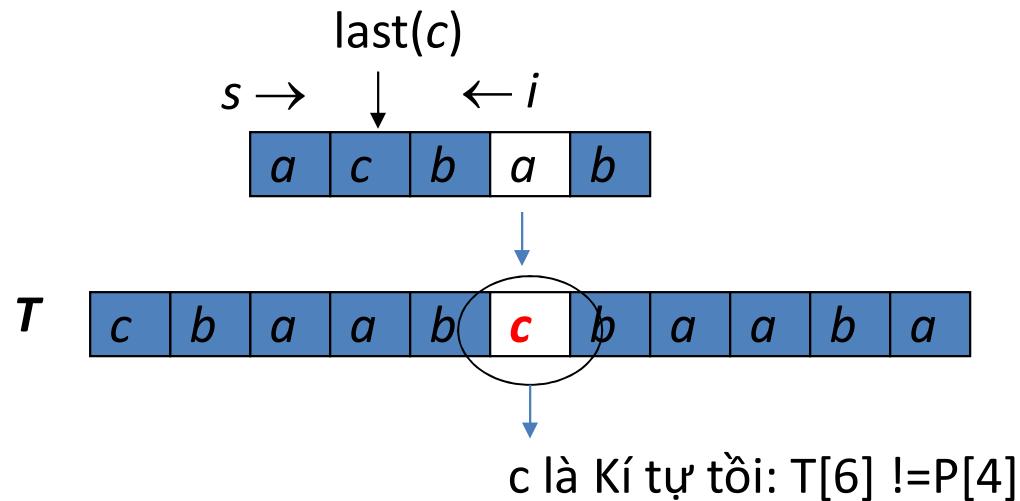
- Ký tự tồi có mặt trong xâu mẫu P và $\text{last}(c) < i$



Khi đó có thể đẩy đến: $s \leftarrow s + (i - \text{last}(c))$

Tình huống 1

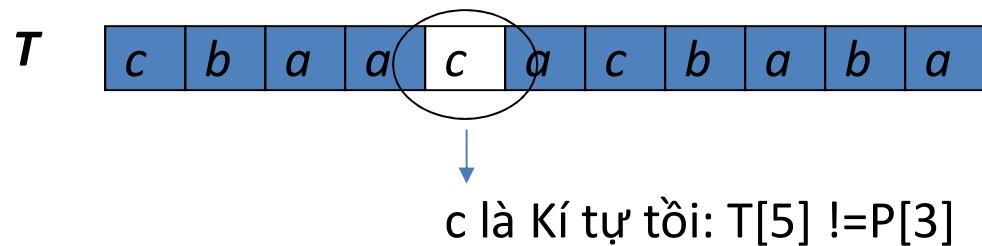
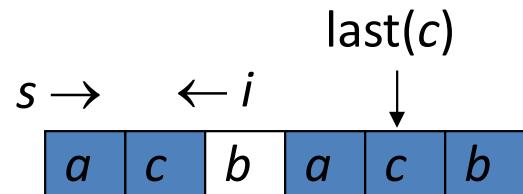
- Ký tự tồi có mặt trong xâu mẫu P và $\text{last}(c) < i$



Khi đó có thể đẩy đến: $s \leftarrow s + (i - \text{last}(c))$

Tình huống 2

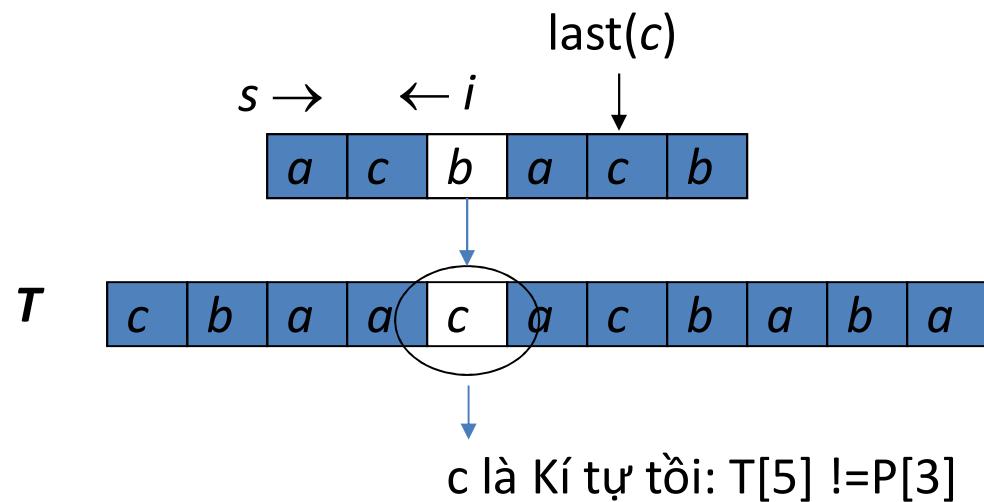
- Ký tự tồi có mặt trong P và $\text{last}(c) > i$



Khi đó: $s \leftarrow s + 1$

Tình huống 2

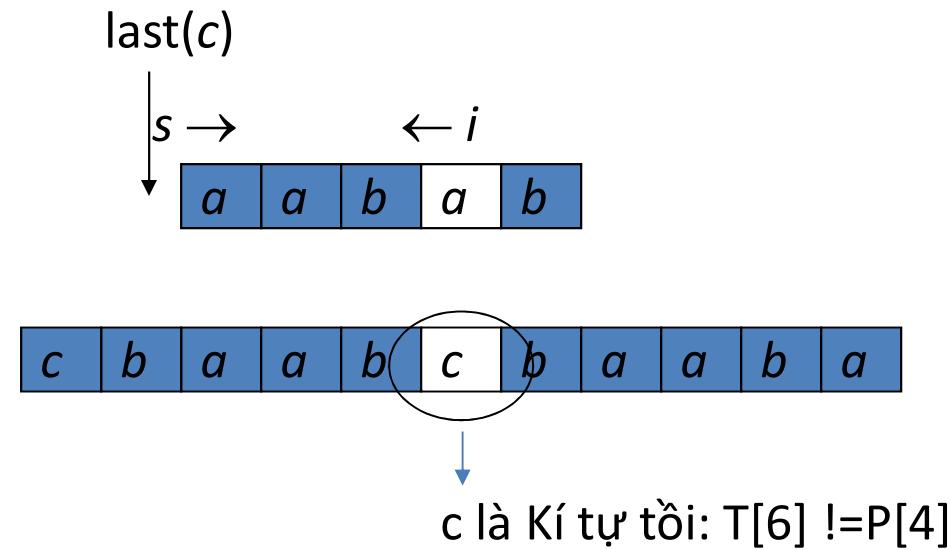
- Ký tự tồi có mặt trong P và $\text{last}(c) > i$



Khi đó: $s \leftarrow s + 1$

Tình huống 3

- Ký tự tồi không có mặt trong xâu mẫu P và vì thế $\text{last}(c) = 0$

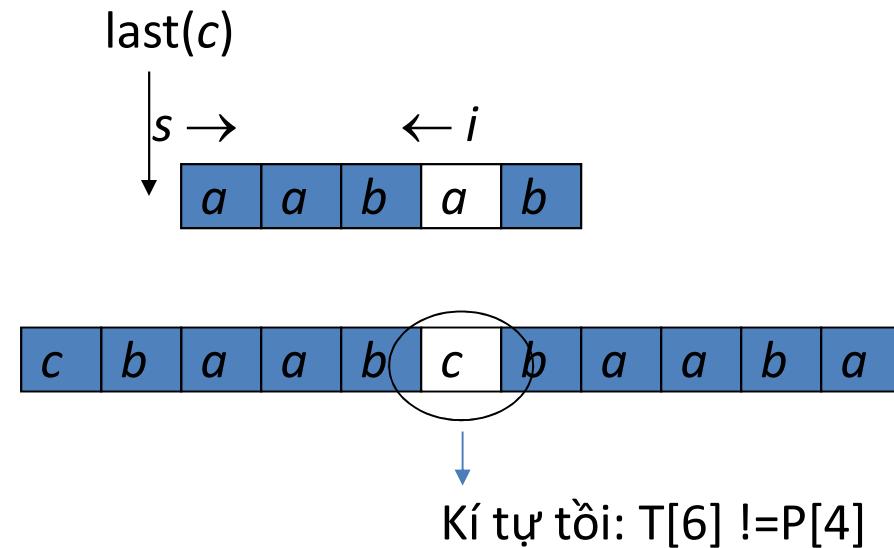


Khi đó: $s \leftarrow s + (i - \text{last}(c))$

hay $s \leftarrow s + i$

Tình huống 3

- Ký tự t_0 không có mặt trong xâu mẫu P và vì thế $\text{last}(c) = 0$



Khi đó: $s \leftarrow s + (i - \text{last}(c))$

hay $s \leftarrow s + i$

Boyer-Moore Algorithm

```
s ← 0
while s ≤ n – m do
    i ← m -1
    while i >= 0 and T[ i + s ] = P[ i ] do
        i ← i – 1
    if i < 0 then
        print s “là vị trí khớp”
        k ← (s + m < n)? m-last[T[s + m]] : 1;
        s ← s + k
    else //T[i+s] là kí tự tồi:
        k ← last( T[ i + s ] )
        s ← s + max( i – k , 1)
```

Boyer-Moore Algorithm

```
#include <bits/stdc++.h>
using namespace std;

#define NO_OF_CHARS 256
void Last(char *pat, int m, int last[NO_OF_CHARS])
{
    int i;
    //Khoi tao:
    for (i = 0; i < NO_OF_CHARS; i++) last[i] = -1;

    //Tinh ham last:
    for (i = 0; i < m; i++) last[(int) pat[i]] = i;
}
```

Boyer-Moore Algorithm

```
void BoyerMoore(char *pat, int m, char *txt, int n)
{
    int last[NO_OF_CHARS];
    Last(pat, m, last);
    int s = 0;
    while (s <= (n - m)) {
        int i = m - 1;
        while(i >= 0 && pat[i] == txt[s + i]) i--;
        if (i < 0) {
            cout << "pattern xuat hien tai vi tri " << s << endl;
            /* Shift the pattern so that the next character in text aligns with the last
               occurrence of it in pattern. The condition s+m < n is necessary for
               the case when pattern occurs at the end of text */
            int k = (s + m < n)? m - last[txt[s + m]] : 1;
            s += k;
        }
        else { //txt[i+s] la ki tutoi:
            int k = last[txt[s+i]];
            s += max(i - k, 1);
        }
    }
}

int main()
{
    char txt[] = "AABAACAAADAAABAAABAA";
    char pat[] = "AABA";
    int m = strlen(pat); int n = strlen(txt);
//    NaiveSM(pat, m, txt, n);
    BoyerMoore(pat, m, txt, n);
    return 0;
}
```

Thời gian tính

- Việc tính hàm **last** đòi hỏi thời gian $O(m + |\Sigma|)$
- Tình huống tồi nhất không khác gì thuật toán trực tiếp, nghĩa là đòi hỏi thời gian $O((n-m+1)m + |\Sigma|)$

Ví dụ tình huống tồi nhất xảy ra khi

- Pattern: ba^{m-1}
- Text: a^n
- Thuật toán làm việc kém hiệu quả đối với bảng Σ nhỏ

5. Tìm kiếm xâu mẫu

5.1. Phát biểu bài toán

5.2. Thuật toán trực tiếp

5.3. Thuật toán Boyer-Moore

5.4. Thuật toán Knuth-Morris-Pratt (KMP)

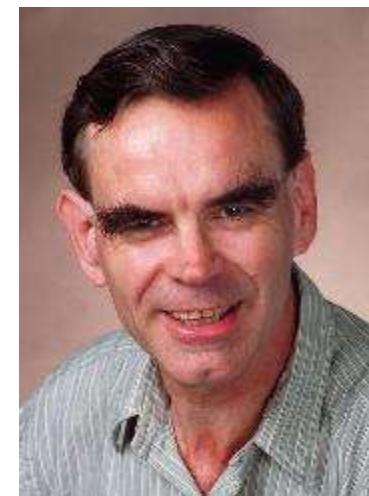
5.4.Thuật toán Knuth-Morris-Pratt



Donald Knuth



James Morris



Vaughan Pratt

Ý tưởng chính là:

- ★ Sử dụng hàm bô trợ (*prefix function*).
- ★ Đạt được thời gian tính $O(n+m)$!

5.4. Thuật toán KMP

- Xâu W được gọi là **tiền tố (prefix)** của xâu X nếu $X = WY$ với một xâu Y nào đó, ký hiệu là $W \sqsubset X$.

Ví dụ $W = \text{ab}$ là prefix của $X = \text{abefac}$ trong đó $Y = \text{efac}$.

Xâu rỗng ϵ là prefix của mọi xâu.

- Xâu W được gọi là **hậu tố (suffix)** của xâu X nếu $X = YW$ với một xâu Y nào đó, ký hiệu là $W \sqsupset X$.

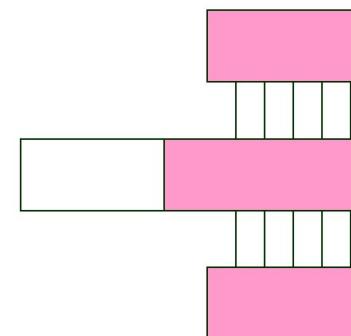
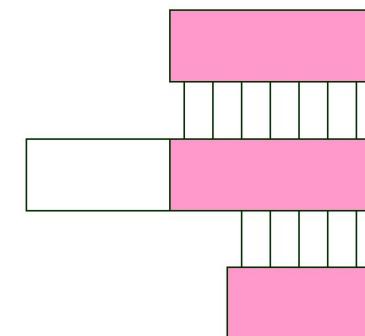
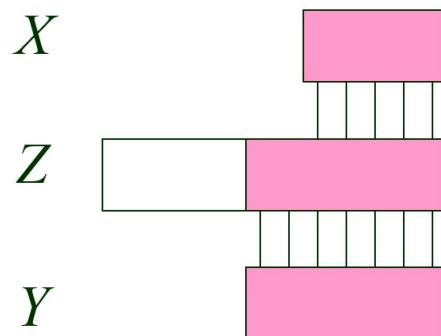
Ví dụ: $W = \text{cdaa}$ là suffix của $X = \text{acbecdaa}$ trong đó $Y = \text{acbe}$

Xâu rỗng ϵ là prefix của mọi xâu.

5.4. Thuật toán KMP

Bố đề Giả sử $X \sqsupset Z$ và $Y \sqsupset Z$.

- a) nếu $|X| \leq |Y|$, thì $X \sqsupset Y$;
- b) nếu $|X| \geq |Y|$, thì $Y \sqsupset X$;
- c) nếu $|X| = |Y|$, thì $X = Y$.

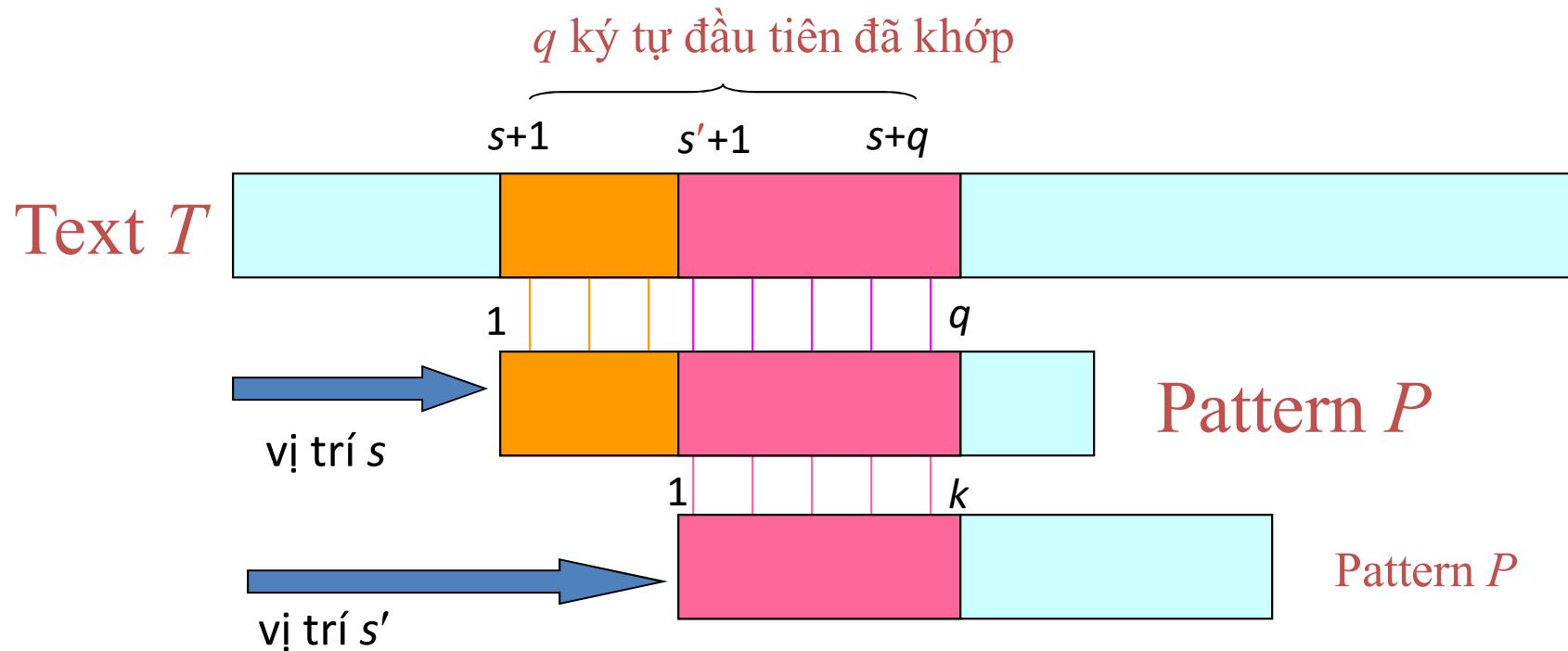


a)

b)

c)

Dịch chuyển tối thiểu



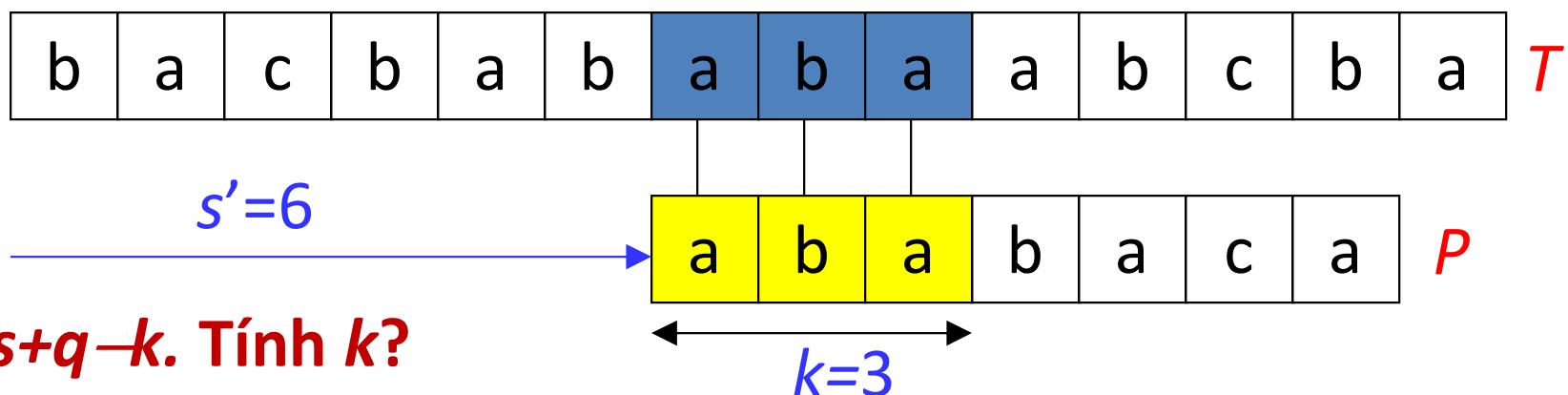
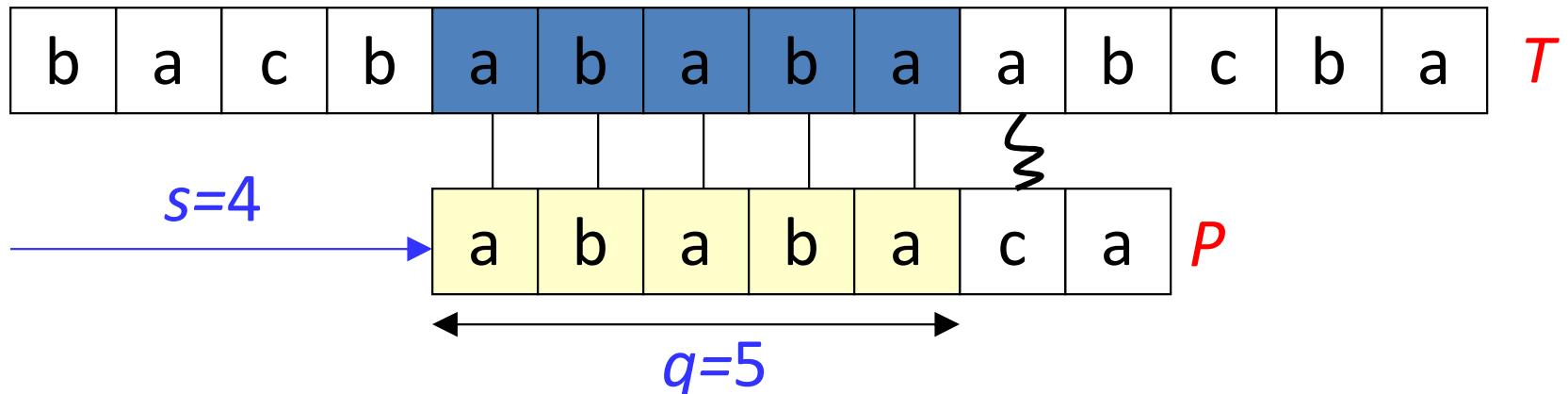
Vấn đề đặt ra là:

Biết rằng prefix $P[1..q]$ của xâu mẫu là khớp với đoạn $T[(s+1)..(s+q)]$, tìm giá trị nhỏ nhất $s' > s$ sao cho:

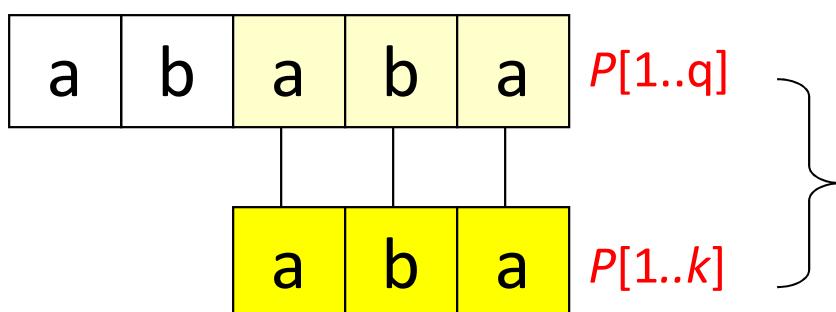
$$P[1..k] = T[(s'+1)..(s'+k)], \text{ trong đó } s'+k=s+q.$$

Khi đó, tại vị trí s' , không cần thiết so sánh k ký tự đầu của P với các ký tự tương ứng của T , bởi vì ta biết chắc rằng chúng là khớp nhau.

Prefix Function: Ví dụ



$s'=s+q-k$. Tính k ?



So sánh xâu mẫu với chính nó; prefix dài nhất của P đồng thời là suffix của $P[1..5]$ là $P[1..3]$; do đó $\pi[5]=3$

Hàm tiền tố (Prefix Function)

Prefix function: $\pi[q]$ là độ dài của prefix dài nhất của $P[1..m]$ đồng thời là suffix **thực sự** của $P[1..q]$.

$$\pi[q] = \max\{ k: k < q \text{ và } P[1..k] \text{ là suffix của } P[1..q] \}$$

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

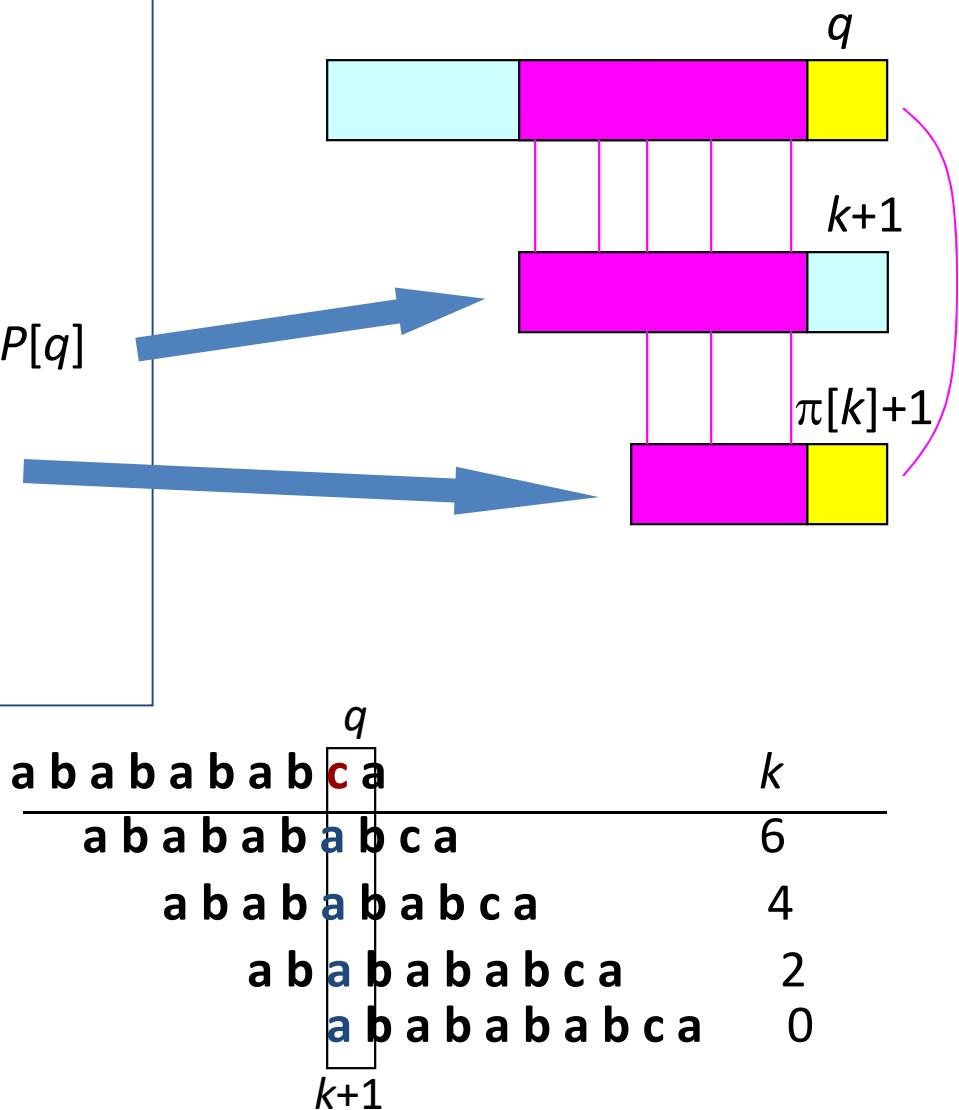
Tính giá trị của Prefix Function

```

Compute-Prefix-Function( $P$ )
 $m \leftarrow \text{length}[P]$ 
 $\pi[1] \leftarrow 0$ 
 $k \leftarrow 0$ 
for  $q \leftarrow 2$  to  $m$ 
    do while  $k > 0$  and  $P[k] \neq P[q]$ 
        do  $k \leftarrow \pi[k]$ 
        if  $P[k+1] = P[q]$ 
            then  $k \leftarrow k + 1$ 
     $\pi[q] \leftarrow k$ 
return  $\pi$ 

```

Ví dụ. $q = 9$ và $k = 6$
 $p[k+1] = a \neq c = p[q]$



Thời gian tính

Compute-Prefix-Function(P)

$m \leftarrow \text{length}[P]$

$\pi[1] \leftarrow 0$

$k \leftarrow 0$

for $q \leftarrow 2$ **to** m

do while $k > 0$ **and** $P[k+1] \neq P[q]$

do $k \leftarrow \pi[k]$ // decrease k by at least 1

if $P[k+1] = P[q]$

then $k \leftarrow k+1$ // $\leq m - 1$ increments, each by 1

$\pi[q] \leftarrow k$

return π

số lần decrements \leq số lần increments,
như vậy, tổng cộng dòng 7 thực hiện nhiều nhất $m - 1$ lần.

Thời gian tính $\Theta(m)$.

KMP Algorithm

```
KMP-Matcher( $T, P$ ) //  $n = |T|$  and  $m = |P|$ 
     $\pi \leftarrow \text{Compute-Prefix-Function}(P)$  //  $\Theta(m)$  time.
     $q \leftarrow 0$ 
    for  $i \leftarrow 1$  to  $n$ 
        do while  $q > 0$  and  $P[q+1] \neq T[i]$ 
            do  $q \leftarrow \pi[q]$ 
        if  $P[q+1] = T[i]$ 
            then  $q \leftarrow q+1$  //  $\leq n$  total increments
        if  $q = m$ 
            then print “Pattern xuất hiện ở vị trí”  $i - m$ 
             $q \leftarrow \pi[q]$ 
    } //  $\Theta(n)$  time
```

Thời gian tổng cộng: $\Theta(m+n)$.

Ví dụ: Thực hiện thuật toán KMP

i	1 2 3 4 5 6 7 8 9 10 11
$P[1..i]$	a b a b b a b a b a a
$\pi[i]$	0 0 1 2 0 1 2 3 4 3 1

abab**abbababbaababbabaa**
abab**bababaa**

đẩy đi $q - \pi[q] = 4 - 2$

ab**ababbabab**baba**bababababaa**
ababbab**babaa**

abababbababba**ababbababaa**
ababbababaa

đẩy đi $9 - 4 = 5$

đẩy đi $1 - 0 = 1$

abababb**ababba**ababbababaa
ababba**babaa**



đẩy đi $6 - 1 = 5$

abababbababba**a**ababbababaa
ababababaa