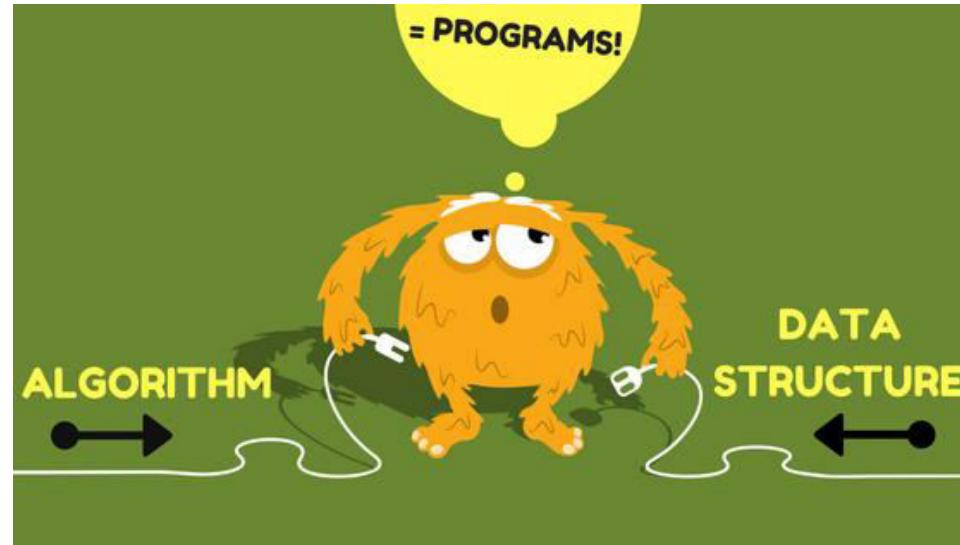




CẤU TRÚC DỮ LIỆU & GIẢI THUẬT

(DATA STRUCTURES & ALGORITHMS)



Khoa Công nghệ thông tin
Bộ môn Công nghệ phần mềm

GIỚI THIỆU MÔN HỌC

- ❖ Tên học phần: CẤU TRÚC DỮ LIỆU
- ❖ Số tín chỉ - số tiết: 3 - 45 tiết
- ❖ Loại học phần: bắt buộc
- ❖ Đối tượng: SV ĐH chính quy ngành CNTT/ATTT

NỘI DUNG MÔN HỌC

- ❖ CHƯƠNG 1: TỔNG QUAN
- ❖ CHƯƠNG 2: DANH SÁCH LIÊN KẾT
- ❖ CHƯƠNG 3: CÂY
- ❖ CHƯƠNG 4: BẢNG BĂM VÀ TÌM KIẾM DỮ LIỆU

NHIỆM VỤ CỦA SINH VIÊN

Tham dự giờ học lý thuyết trên lớp.

- ❖ Làm các bài tập, tiểu luận theo yêu cầu của giảng viên.
- ❖ Dự kiểm tra tại lớp và thi cuối học phần.
- ❖ **Đánh giá học phần:**
 - Đánh giá quá trình: 30%
 - Điểm thái độ học tập: 10%
 - Điểm tiểu luận (bài tập, kiểm tra tại lớp): 20%
 - Điểm thi kết thúc học phần: 70% (tự luận)

TÀI LIỆU THAM KHẢO

- ❖ Bộ slide bài giảng CTDL> – Khoa CNTT – ĐH CNTP TPHCM
- ❖ Bài giảng CTDL> – Khoa CNTT – ĐH CNTP TPHCM

- ❖ Algorithm - Robert Sedgewick and Kevin Wayne- Princeton University
- ❖ Data Structures & Algorithms in Java - Robert Lafore
- ❖ Introduction to Algorithms - Thomas, Charles,...
- ❖ A Laboratory Course in C++ Data Structure - James Roberge,...

THÔNG TIN GIẢNG VIÊN GIẢNG DẠY

- ❖ Giảng viên giảng dạy:
- ❖ Email:
- ❖ Mã lớp trên kênh Google Classroom:



CẤU TRÚC DỮ LIỆU & GIẢI THUẬT (DATA STRUCTURES & ALGORITHMS)

Chương 1

TỔNG QUAN CTDL>

Khoa Công nghệ thông tin
Bộ môn Công nghệ phần
mềm

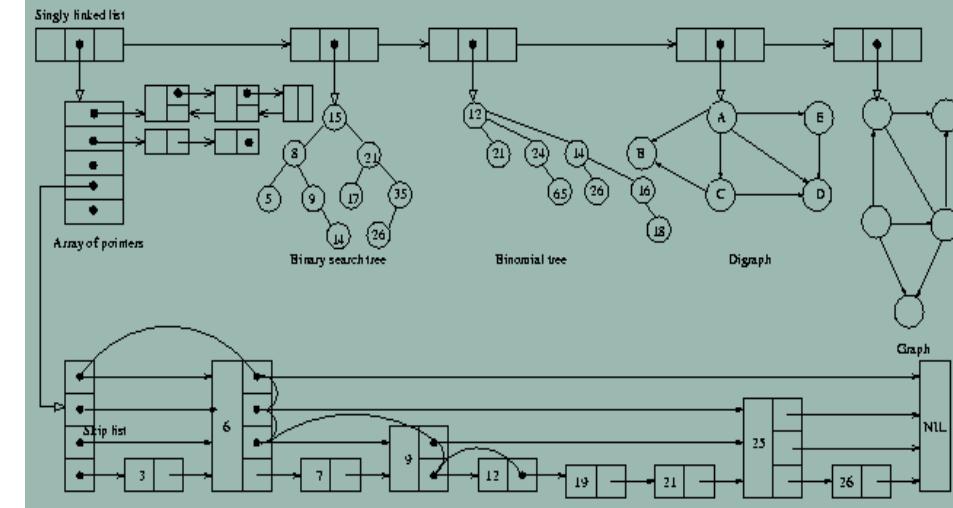
NỘI DUNG

1. Vai trò của CTDL trong tin học
2. Kiểu dữ liệu
3. Giải thuật

Algorithms + Data Structures = Programs

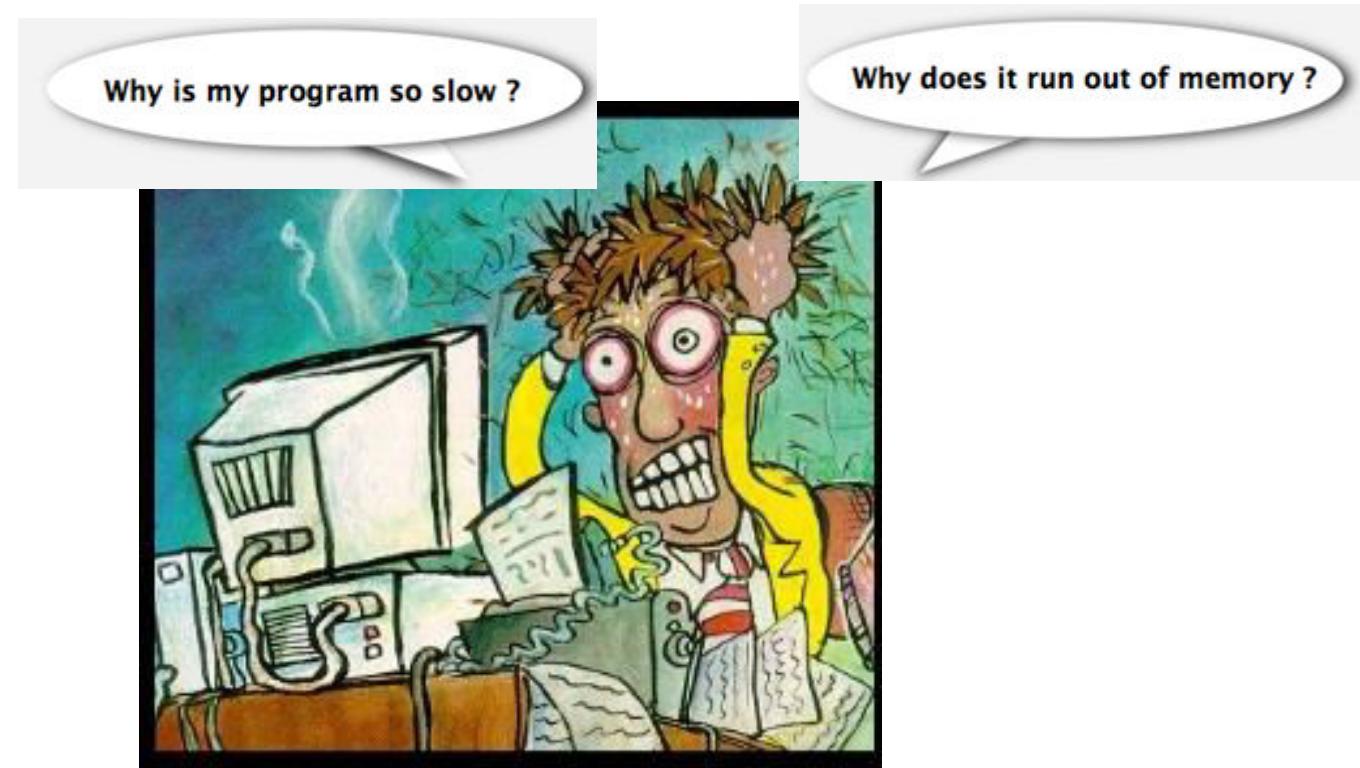
Niklaus Wirth (cha đẻ NNLT Pascal) - https://en.wikipedia.org/wiki/Niklaus_Wirth

```
while (j <> nul) and (l < 77,  
), end; i := l^.next; write(mass[i]); if r <> nul then r^.prev,  
more than = a > b, else r := rel then p=p^.next;  
v, beres integer); k := k+1, begin begin for l=2  
table (var l:art mas,  
else j^.next^.val := T;  
r <> nul then r^.prev,  
into i do inc(t+1, draw),  
t then first^.prev, x:=0,  
to do begin D := mass[k];  
L := L+1  
begin  
if r <> rel then p=p^.next, x:=0,  
<For J=1 to N>  
else first := p^.next; For l=0 to 2 do  
y],  
ory := r to do  
mass[i] := 1,  
p=p^.next, k := k+1, end;  
r = p^.prev, begin  
(End.)
```



Vấn đề quan trọng của thực thi chương trình: Thuật toán hay CTDL?

Chương trình của tôi có giải được một bài toán thực tế với dữ lớn không?



1. Vai trò của CSDL trong tin học

- ❖ Ví dụ:

Giả sử một lớp học phụ đạo có 5 học sinh, mỗi học sinh có điểm 3 môn toán, lý, hóa. Giả sử đã có dữ liệu điểm của các học sinh. Hãy viết chương trình xuất điểm các hs thành 1 bảng có 5 dòng, 3 cột.

Sinh viên hãy đề xuất các cách giải có thể có cho bài toán



Các cách giải quyết bài toán

❖ Cách 1: dùng mảng 1 chiều lưu điểm các học sinh

```
void main()
{
    // giả sử đã nhập điểm cho các hs.
    int diem[15] = {4,6,2,7,8,9,4,6,9,10,9,4,6,4,8};

    // phần xuất điểm
    for(int i=0; i<15; i++)
    {
        printf("%5d", diem[i]);
        if(i%3==2)
            printf("\n");
    }
}
```

muốn biết điểm môn k của hs n thì sao?
→ $\text{diem} [(n-1)*3+(k-1)]$ → công thức phức tạp

Nếu số lượng hs và điểm môn học nhiều hơn thì
gặp những khó khăn gì khi xử lý?

Các cách giải quyết bài toán (tt)

❖ Cách 2: dùng mảng 2 chiều có 5 dòng 3 cột để lưu điểm học sinh

```
void main()
{
    // giả sử đã nhập điểm cho các hs.
    int diem[5][3] = {4,6,2,7,8,9,4,6,9,10,9,4,6,4,8};
    // phần xuất điểm
    for(int i=0; i<5; i++)
    {
        for(int j=0; j<3; j++)
            printf("%5d", diem[i][j]);
        printf("\n");
    }
}
```

- * Muốn biết điểm môn k của hs n thì sao?
→ $\text{diem}[n-1][k-1]$
- * Cách 2 tốt hơn cách 1 thế nào?
- * Nếu lượng hs và môn học tăng lên nhiều và cần thêm xóa hs, điểm thì sẽ gặp khó khăn gì?

Các cách giải quyết bài toán (tt)

❖ Cách 3: dùng mảng 1 chiều có kiểu phần tử dạng cấu trúc DiemHS

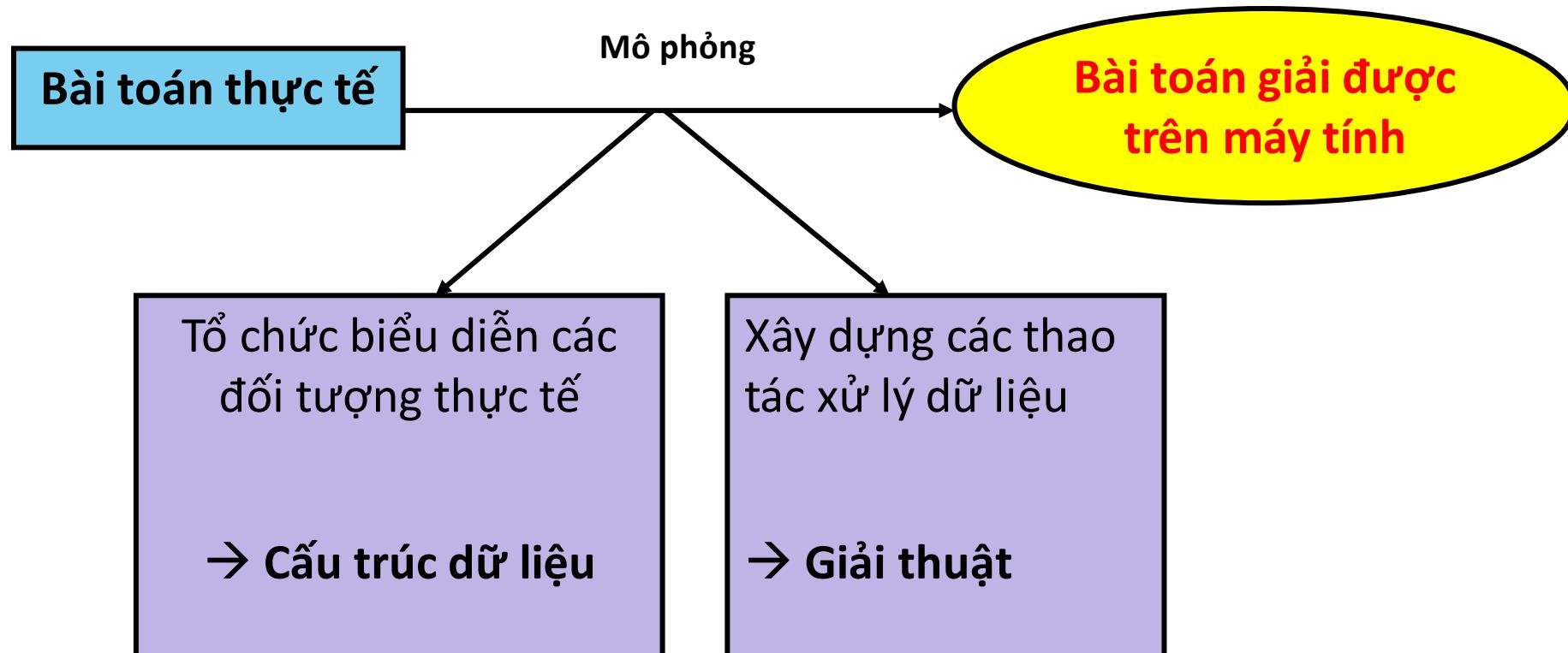
```
struct DiemHS
{
    int toan, ly, hoa;
};
void main()
{
    DiemHS diem[5];
    //giả sử đã nhập dữ liệu cho mảng diem rồi.
    //xuất điểm
    for(int i=0; i<5;i++)
        printf("%5d, %5d, %5d \n", diem[i].toan, diem[i].ly, diem[i].hoa);
}
```

muốn biết điểm môn *toan* của hs *n* thì sao?
→ diem [n-1].toan
Những ưu điểm của cách 3 so với cách 1&2?

Nhận xét

- ❖ Cả 3 phương án đều giải quyết trọn vẹn bài toán. Nhưng phương án 1 biểu diễn dữ liệu không tự nhiên và thao tác trên đối tượng phức tạp.
- ❖ Phương án 2 và 3 cung cấp một CTDL và các thao tác xử lý hợp lý. Do đó tùy theo trường hợp mà ta chọn phương án 2 hoặc 3.
- ❖ Chọn phương án 2 nếu muốn dễ dàng xử lý dữ liệu. Hạn chế thêm/xóa dữ liệu.
- ❖ Chọn phương án 3 nếu muốn nhấn mạnh việc mô phỏng ý nghĩa đối tượng, thao tác chọn học sinh, môn học dễ dàng

1. Vai trò của CSDL trong tin học



1 Vai trò của CSDL trong tin học (tt)

❖ Tổ chức biểu diễn các đối tượng thực tế.

Nghĩa là cần xây dựng cấu trúc thích hợp nhất sao cho phản ánh chính xác các đối tượng thực tế và có thể thao tác xử lý dễ dàng → lựa chọn **xây dựng CSDL**

❖ Xây dựng các thao tác xử lý dữ liệu

Từ những yêu cầu thực tế của bài toán, ta cần **tìm giải thuật** để xác định các thao tác máy tính tác động lên dữ liệu để cho ra kết quả mong muốn.

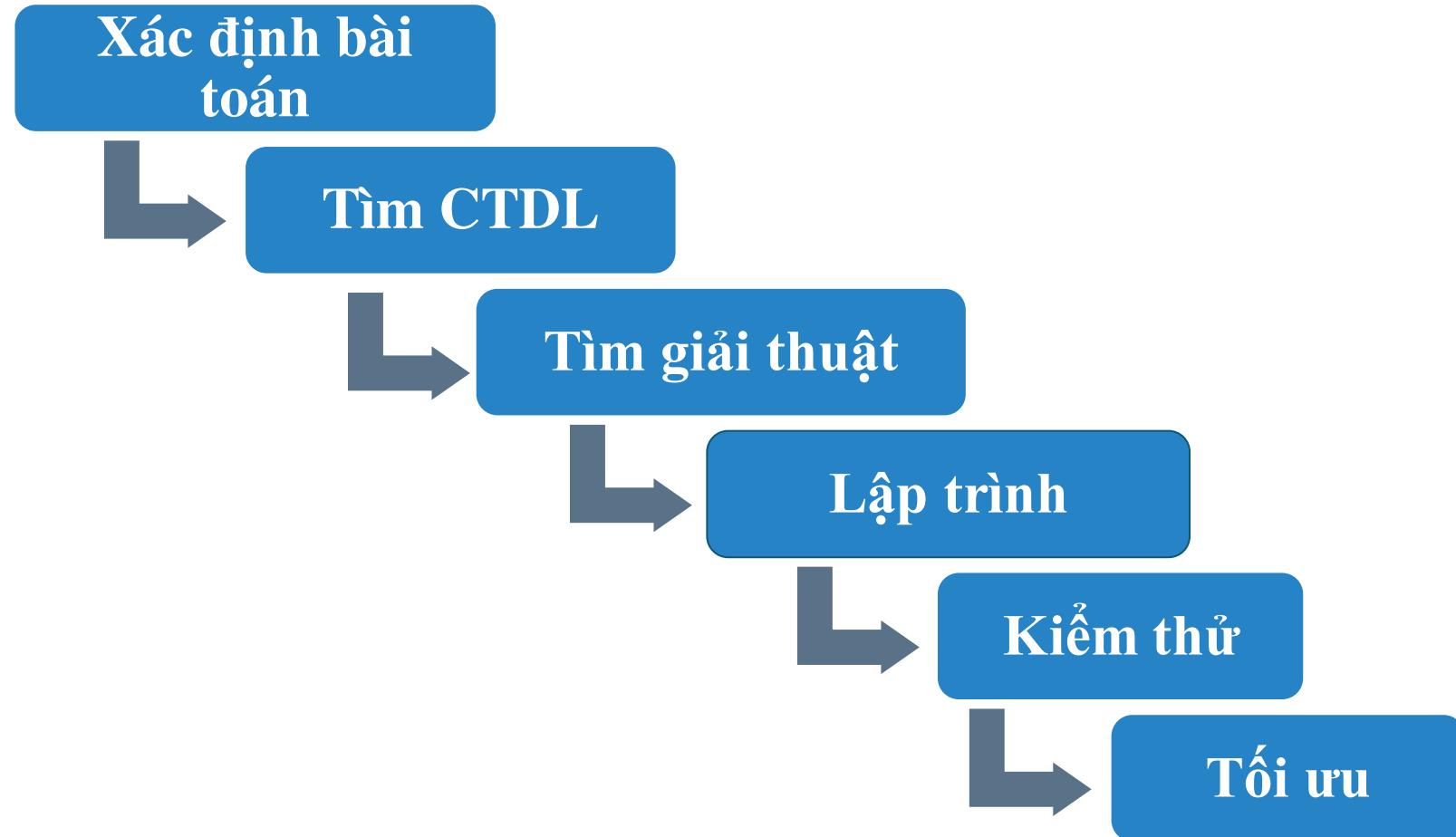
❖ Lưu ý:

Trong một số trường hợp ta chọn một giải thuật để giải quyết bài toán có thể chưa nhận được kết quả tối ưu, nhưng vì nếu dùng cần tối ưu thì khó thực hiện và chi phí thực hiện rất lớn → tìm các giải thuật xấp xỉ.

Kết luận

- ❖ Hai công việc trên (tổ chức biểu diễn các đối tượng thực tế, xây dựng các thao tác xử lý dữ liệu) không tồn tại độc lập mà có mối quan hệ chẽ với nhau. Giải thuật phản ảnh các phép xử lý, còn đối tượng xử lý của giải thuật là dữ liệu.
- ❖ Để xây dựng được giải thuật phù hợp cần phải biết nó tác động lên loại dữ liệu nào, và khi chọn lựa CTDL cũng cần phải hiểu rõ những thao tác nào sẽ tác động đến nó.

Qui trình giải một bài toán trong tin học



2. Kiểu dữ liệu

❖ Định nghĩa:

Một kiểu dữ liệu T được xác định bởi 1 bộ

$$T = \langle V, O \rangle$$

trong đó:

- V: tập giá trị hợp lệ mà 1 đối tượng T có thể lưu trữ.
- O: tập các thao tác xử lý trên đối tượng T

❖ Kiểu dữ liệu gồm các dạng:

- Các kiểu dữ liệu cơ bản: int, float, char...
- Các kiểu dữ liệu có cấu trúc cơ bản: mảng, danh sách liên kết
- Các kiểu dữ liệu có cấu trúc: do người tự định nghĩa (struct, class,...)

Ví dụ

❖ Kiểu dữ liệu Số Nguyên = <V, O>

- V= -32768 ... 32767 // miền giá trị
- O = {+,-,* ,/,%, <, > ==,...} // phép toán trên dữ liệu số nguyên

❖ Kiểu dữ liệu cấu trúc DiemHS = <V, O>

- V : giá trị điểm các môn
- O: nhập/xuất dữ liệu, tính trung bình, tìm max, min, sắp xếp, tìm kiếm...

1.3. Một số tiêu chí chọn lựa CTDL

❖ CTDL phải phản ánh đúng thực tế

Đây là tiêu chí quan trọng nhất quyết định tính đúng đắn của bài toán, cần phải khảo sát kỹ lưỡng, dự trù đầy đủ các tình huống xảy ra.

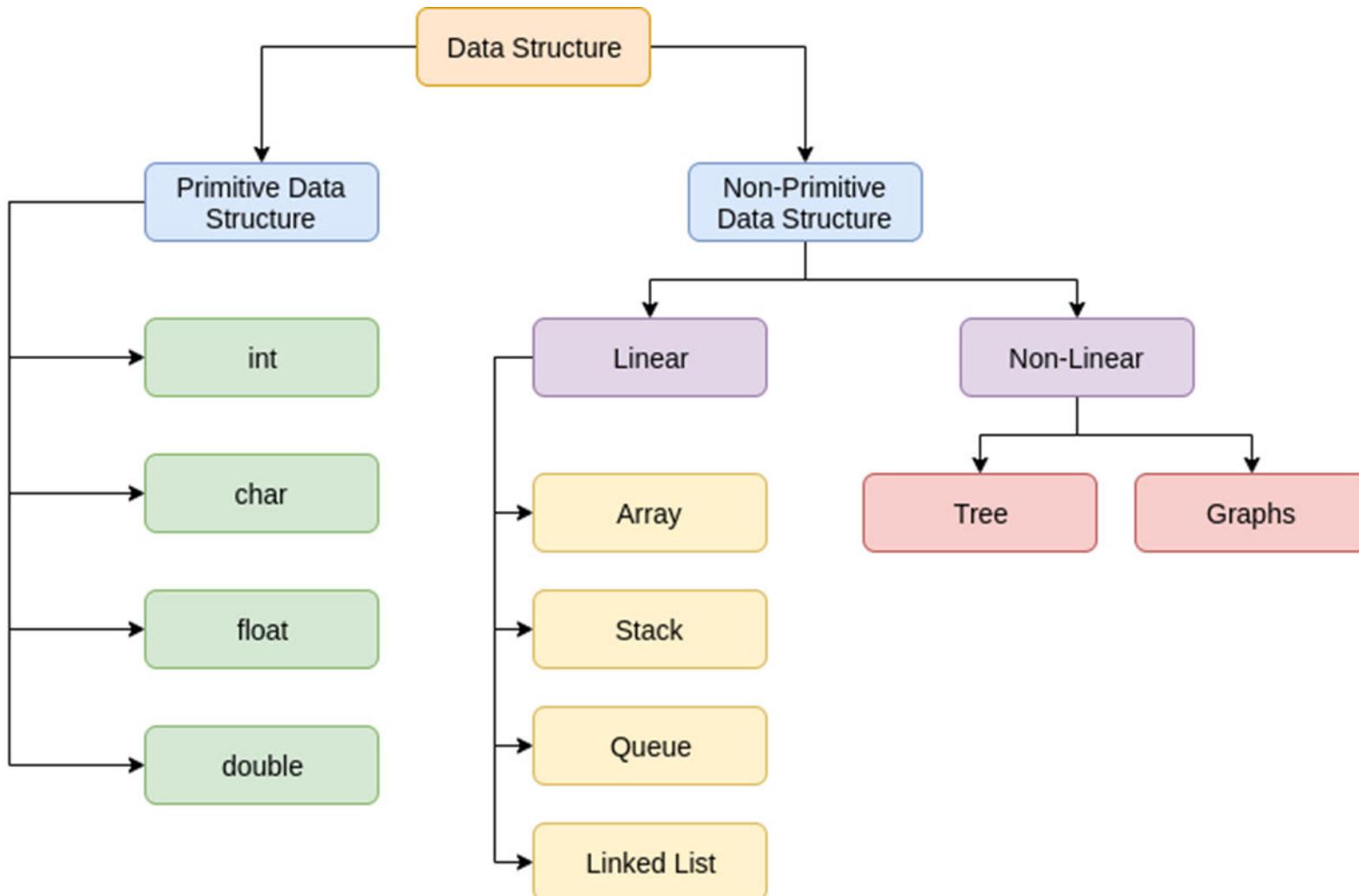
❖ CTDL phải phù hợp với các thao tác xử lý

Tiêu chí này giúp cho việc cài đặt thực hiện các yêu cầu bài toán được đơn giản hơn.

❖ CTDL phải tiết kiệm tài nguyên của hệ thống

Việc sử dụng tiết kiệm tài nguyên sẽ làm tăng tính hiệu quả cũng như số lượng các bài toán cài trên máy.

Phân loại CTDL



3. Giải thuật (Thuật toán)

Một dãy hữu hạn các chỉ thị có thể thi hành để đạt mục tiêu đề ra

- ❖ Ví dụ: Tìm giá trị lớn nhất của dãy a số nguyên, có n phần tử
- ❖ Giải thuật:
 - B1. Đặt giá trị max tạm thời bằng số nguyên đầu tiên trong dãy.
 - B2. So sánh số nguyên tiếp sau với giá trị max tạm thời. Nếu nó lớn hơn giá trị max tạm thời thì đặt max tạm thời bằng số nguyên đó.
 - B3. Lặp lại B2 nếu còn các số nguyên trong dãy.
 - B4. Dừng khi không còn số nguyên nào nữa trong dãy. Max tạm thời chính là số nguyên lớn nhất của dãy.

3.1. Tính chất của giải thuật

1. Tính đúng

Cho kết quả **chính xác**

2. Tính dừng

Dừng sau **hữu hạn số bước** thực hiện

3. Tính tổng quát

Có thể **áp dụng** cho các bài toán **tương tự**

4. Tính khách quan

Trong cùng điều kiện, **kết quả như nhau**

5. Tính hiệu quả

Tài nguyên chiếm dụng (CPU + Memory)



3.2. Mô tả giải thuật

1. Ngôn ngữ tự nhiên (natural language)
2. Lưu đồ (flow chart)
3. Mã giả (pseudo code)
4. Ngôn ngữ lập trình (programing language)

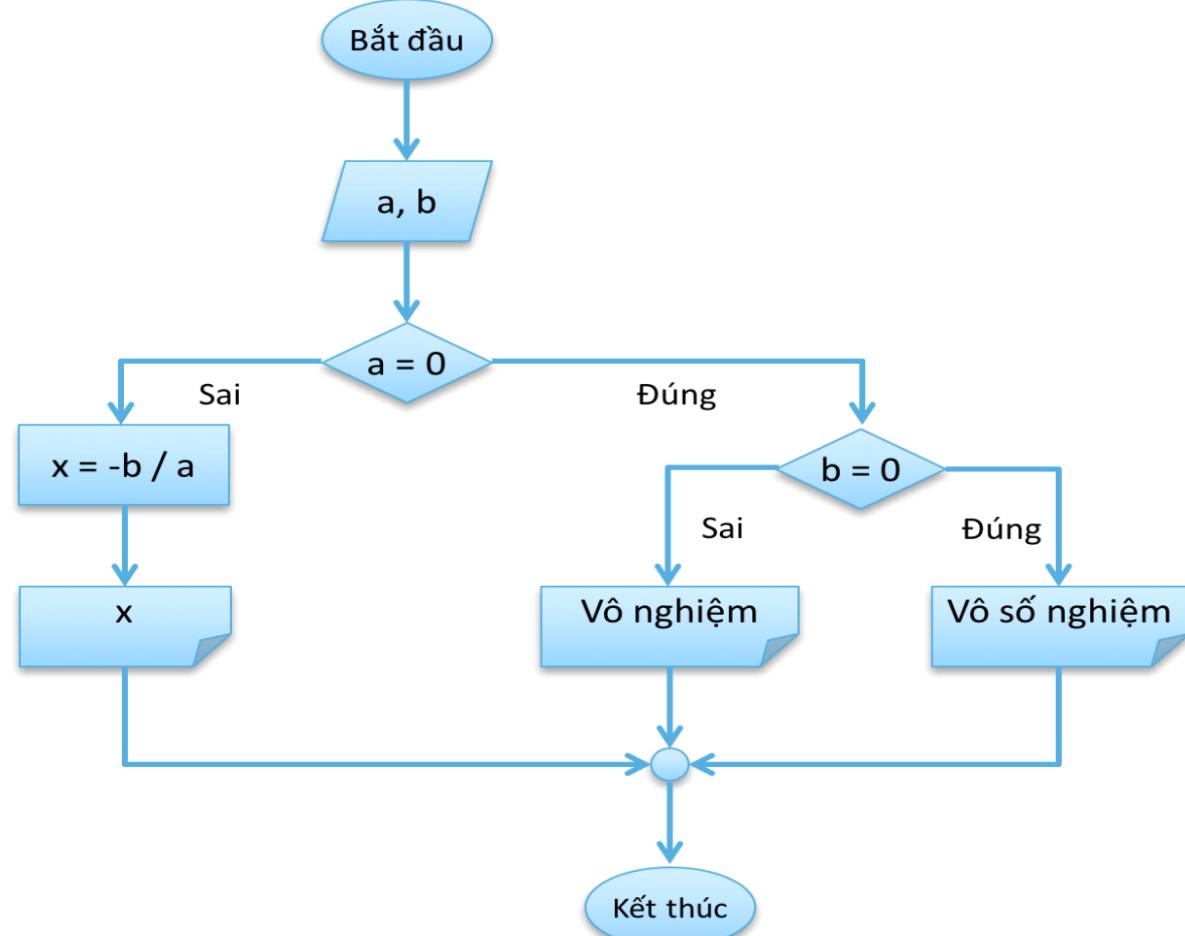
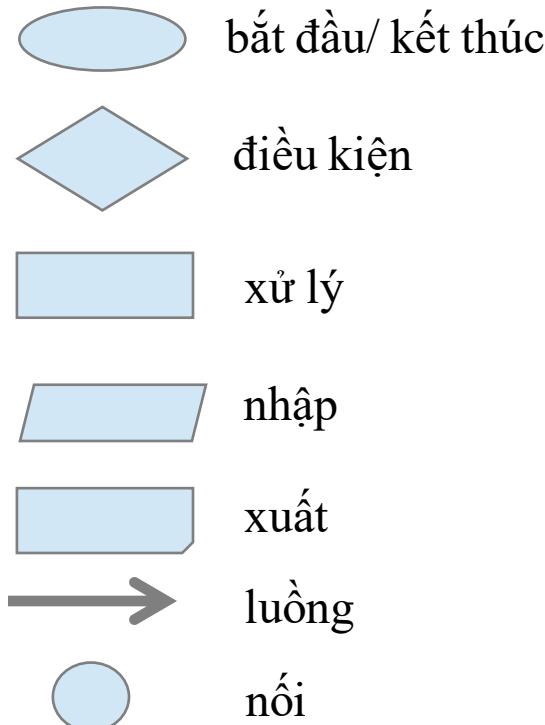
Mô tả thuật toán

- ❖ Giải và biện luận phương trình bậc nhất: $ax + b = 0$
- ❖ Mô tả bằng ngôn ngữ tự nhiên

- Nếu $a = 0$ thì
 - Nếu $b = 0$ thì PT có vô số nghiệm
 - Ngược lại PT vô nghiệm
- Ngược lại
 - PT có nghiệm $x = -b/a$

Mô tả thuật toán

- ❖ Giải và biện luận phương trình bậc nhất: $ax + b = 0$
- ❖ Mô tả bằng lưu đồ



Mô tả thuật toán

- ❖ Giải và biện luận phương trình bậc nhất: $ax + b = 0$
- ❖ Mô tả bằng mã giả (NN tự nhiên kết hợp NN lập trình)

```
if a = 0 then  
    if b = 0 then  
        write "PT có vô số nghiệm"  
    else  
        write "PT vô nghiệm"  
    endif  
else  
    x = -b/a  
    write "Nghiem x = " + x  
endif
```





CẤU TRÚC DỮ LIỆU & GIẢI THUẬT (DATA STRUCTURES & ALGORITHMS)

DANH SÁCH LIÊN KẾT (LINKED LIST)

Khoa Công nghệ thông tin
Bộ môn Công nghệ phần mềm

NỘI DUNG

I. ĐẶT VĂN ĐỀ

1. XÉT VÍ DỤ
2. BIẾN TĨNH
3. BIẾN ĐỘNG
4. CON TRỎ
5. TÍNH CHẤT MẢNG MỘT CHIỀU
6. TÍCH CHẤT DANH SÁCH LIÊN KẾT

II. DANH SÁCH LIÊN KẾT

1. ĐỊNH NGHĨA DSLK
2. CÁC LOẠI DSLK
3. DANH SÁCH LIÊN KẾT ĐƠN
4. KHAI BÁO DSLK ĐƠN
5. CÁC THAO TÁC CƠ BẢN TRÊN DSLK

MỤC TIÊU

- ❖ Nắm vững khái niệm về cấu trúc dữ liệu tĩnh và động.
- ❖ Nắm vững cách tổ chức dữ liệu động bằng danh sách liên kết và minh họa được các thao tác xử lý trên danh sách liên kết đơn.
- ❖ Cài đặt minh họa được các thao tác của danh sách đơn bằng ngôn ngữ C/ C++.

I. ĐẶT VÂN ĐỀ

1. Xét ví dụ

- ❖ Cho mảng một chiều (cấu trúc dữ liệu tĩnh).

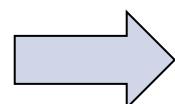
3	1	7	8	2	6	
---	---	---	---	---	---	--

- ❖ Chèn $x=4$ vào vị trí 3?

3	1	7	4	8	2	6
---	---	---	---	---	---	---

- ❖ Xoá phần tử có giá trị $x = 7$

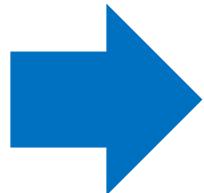
3	1	7	4	8	2	6
---	---	---	---	---	---	---



Độ phức tạp của chèn/xóa trên mảng 1 chiều là $O(n)$

Vấn đề cấu trúc dữ liệu tĩnh

- ❖ Vấn đề phức tạp khi chèn/ xóa?
- ❖ Vấn đề giới hạn kích thước vùng nhớ tối đa?
- ❖ Vấn đề giải phóng vùng nhớ khi không cần dùng đến?
- ❖ Vấn đề vùng nhớ không liên tục?



DÙNG CẤU TRÚC DỮ LIỆU ĐỘNG

2. Biến tĩnh (static)

- ❖ **Khai báo:**

<kiểu dữ liệu> tên biến;

- ❖ Ví dụ: `int a; float y; char s[20];`

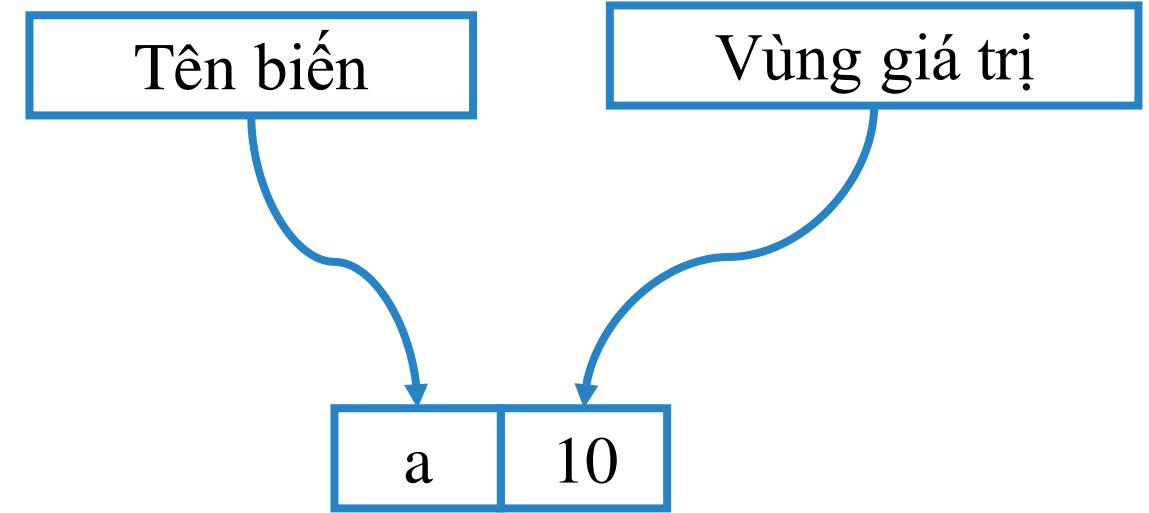
`a = 10; //gán giá trị cho a`

- ❖ **Đặc điểm:**

- ❖ Tồn tại trong phạm vi khai báo (tầm vực).

- ❖ Được cấp phát vùng nhớ trong vùng giá trị dữ liệu.

- ❖ Kích thước cố định.



3. Biến động (dynamic)

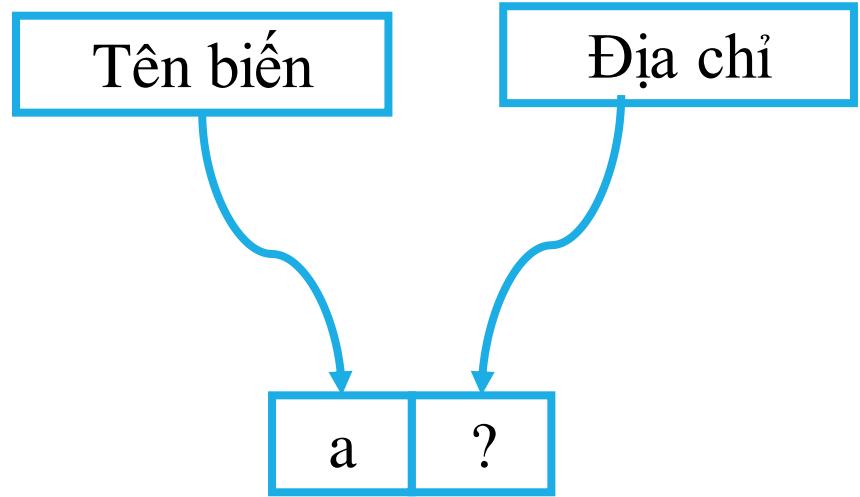
❖ Khai báo:

<kiểu dữ liệu> *tên biến;

❖ Ví dụ: int *a; float *y;

❖ Đặc điểm:

- ❖ Chứa địa chỉ của một đối tượng dữ liệu
- ❖ Được cấp phát hoặc giải phóng bộ nhớ tùy thuộc vào người lập trình
- ❖ Kích thước có thể thay đổi



Biến động

❖ Thao tác:

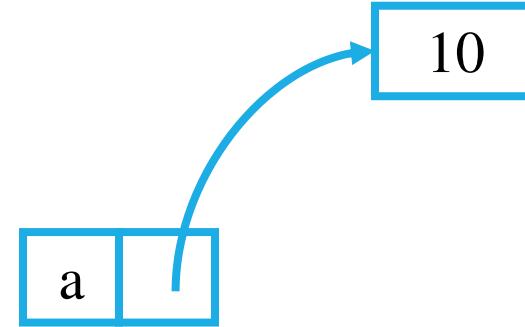
- ❖ Cấp phát bộ nhớ: `new`
- ❖ Giải phóng bộ nhớ: `delete`

❖ Ví dụ:

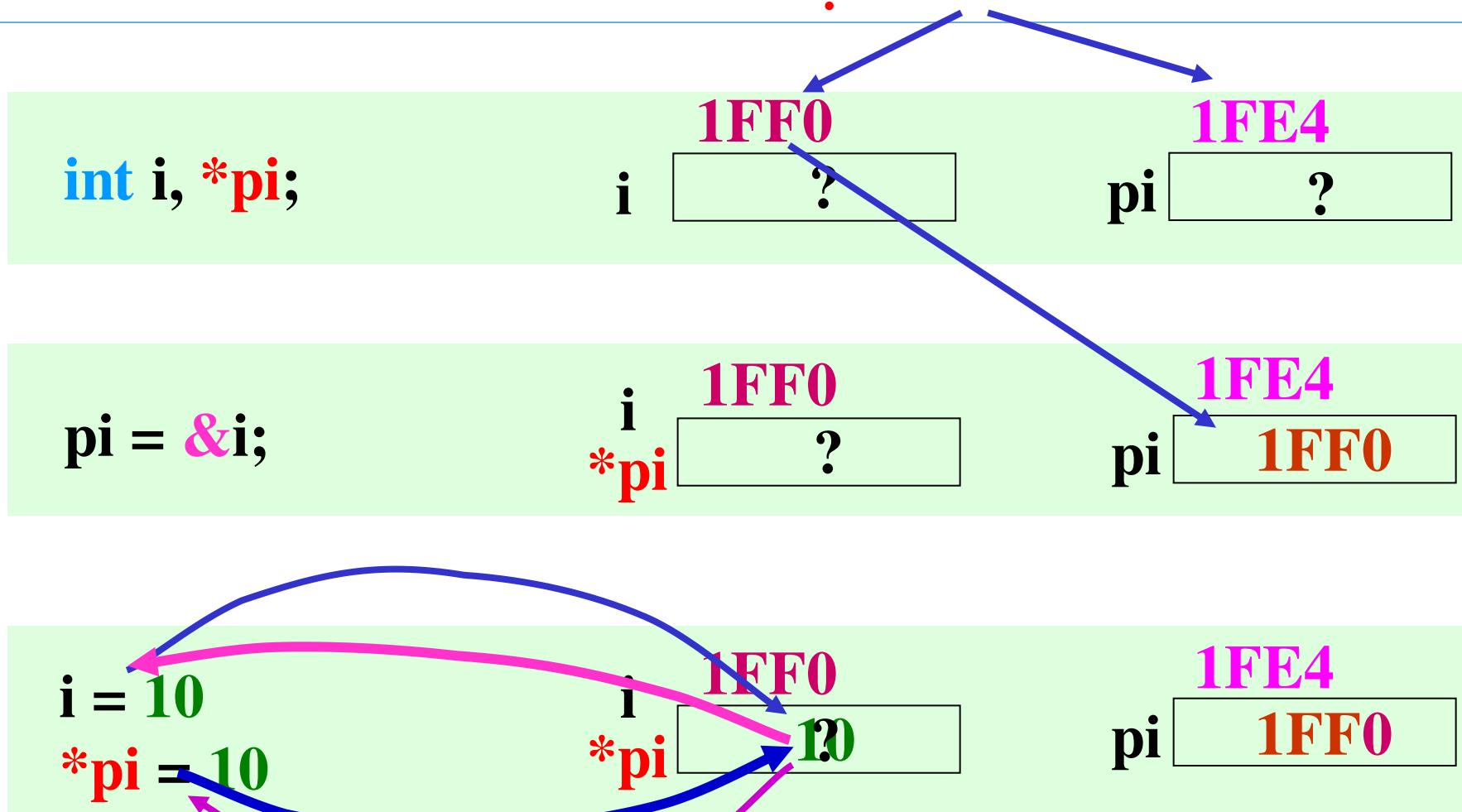
```
int *a; //Khai báo
```

```
a = new int; // Cấp phát
```

```
*a = 10; // ... Các thao tác (...)  
delete a; // Giải phóng
```

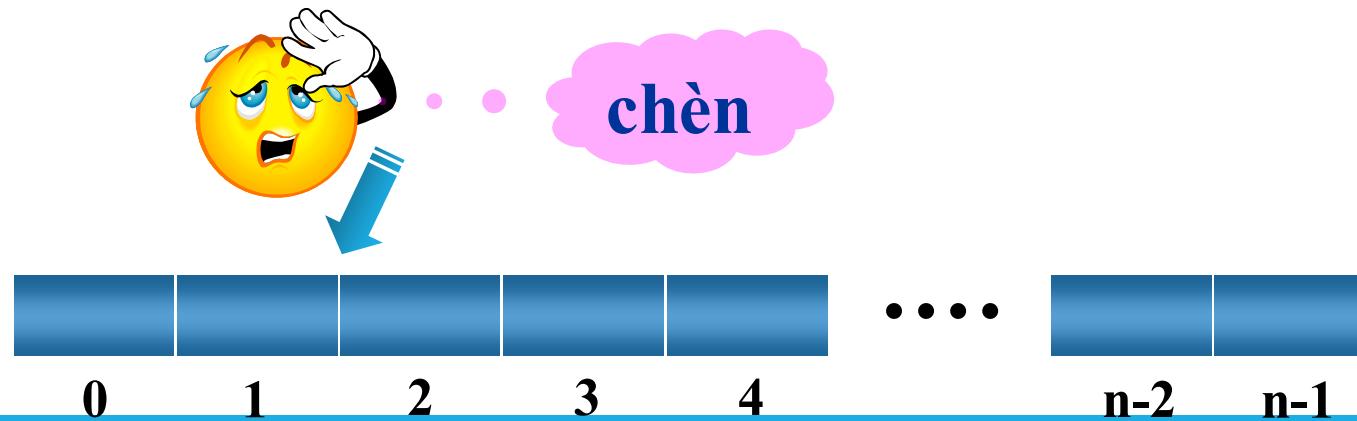


4. Con trỏ - Pointer



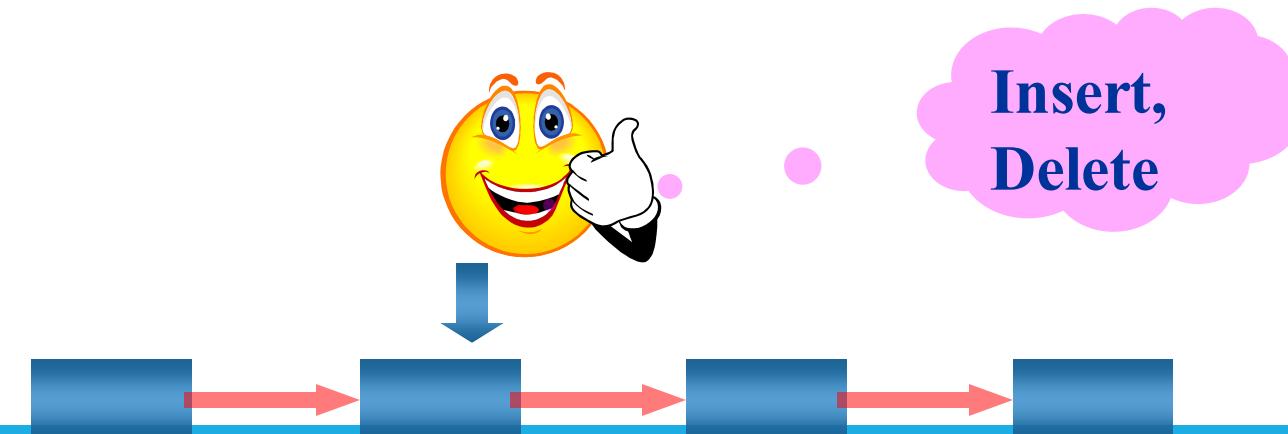
5. Tính chất mảng một chiều

- ❖ Có kích thước cố định (*cấp phát tĩnh*).
- ❖ Thêm/xóa phần tử có độ phức tạp cao.
- ❖ Các phần tử tuần tự theo chỉ số $0 \Rightarrow n-1$.
- ❖ Truy cập ngẫu nhiên.



6. Tính chất danh sách liên kết (DSLK)

- ❖ Cấp phát động lúc chạy chương trình.
- ❖ Các phần tử nằm rải rác ở nhiều nơi trong bộ nhớ.
- ❖ Kích thước danh sách chỉ bị giới hạn do RAM.
- ❖ Thao tác thêm/xóa đơn giản.



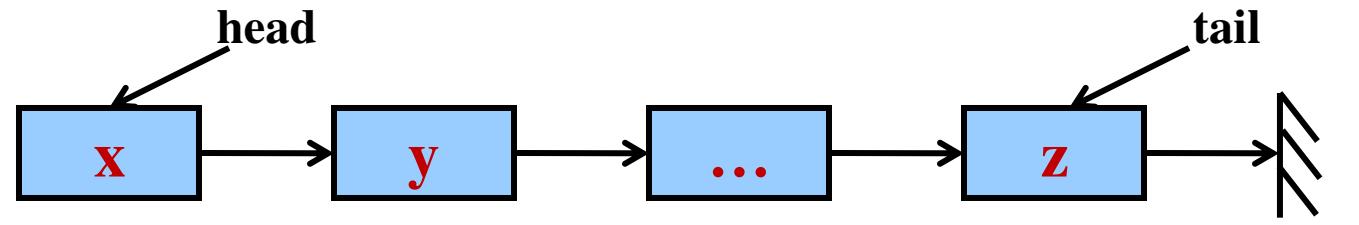
II. DANH SÁCH LIÊN KẾT – LINKED LIST

2. Định nghĩa DSLK

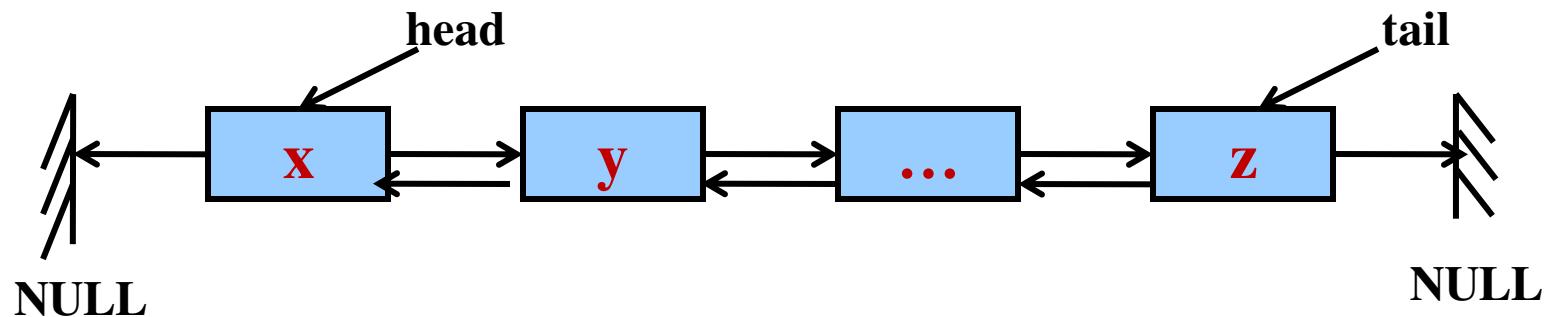
- ❖ Cho T là một **kiểu được định nghĩa trước**, **kiểu danh sách T_x gồm các phần tử thuộc kiểu T** được định nghĩa là: $T_x = \langle V_x, O_x \rangle$.
- ❖ $V_x = \{$ tập hợp có thứ tự các phần tử kiểu T được móc nối với nhau theo trình tự tuyến tính $\}$.
- ❖ $O_x = \{$ các phương thức thiết lập trên danh sách T_x (như tạo danh sách; hủy danh sách; tìm phần tử trong danh sách; hủy 1 phần tử...) $\}$

2. Các loại danh sách liên kết

- Danh sách liên kết đơn



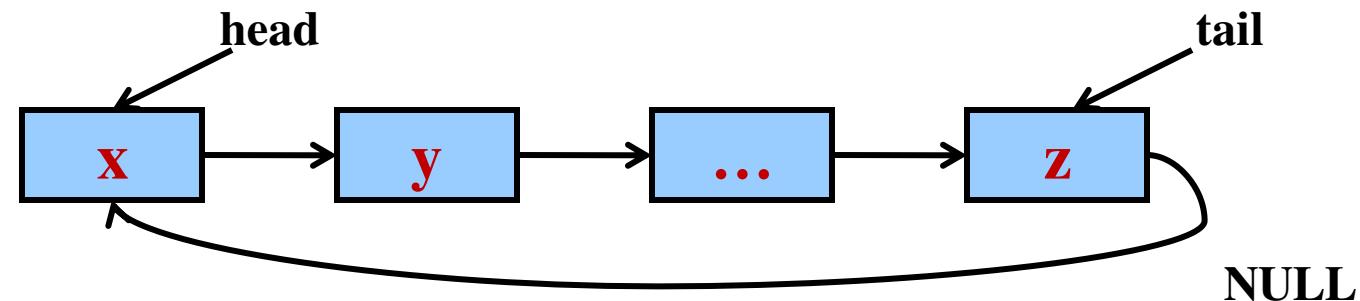
- Danh sách liên kết đôi



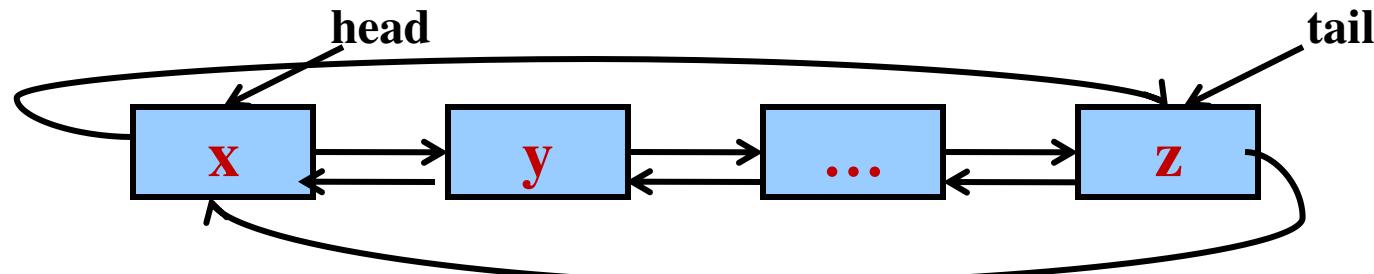
2. Các loại danh sách liên kết (tt)

❖ Danh sách liên kết vòng:

❖ Vòng đơn

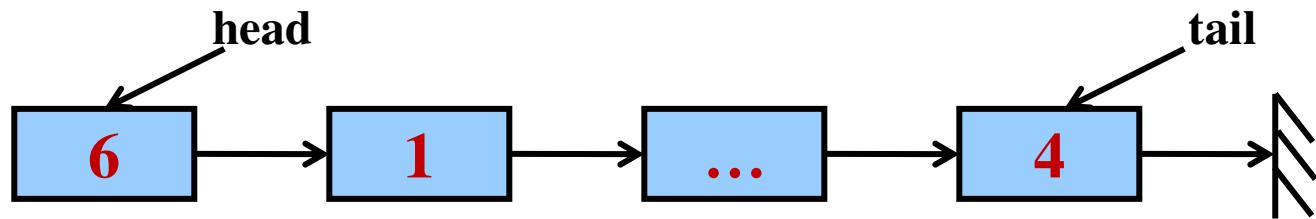


❖ Vòng đôi



3. Định nghĩa DSLK đơn (Single List)

Danh sách liên kết đơn là chuỗi các phần tử (**Node**), được tổ chức theo thứ tự tuyến tính. Mỗi phần tử liên kết với 1 phần tử đứng liền kề sau trong danh sách.



Mỗi phần tử (**Node**) gồm 2 thành phần:

NULL

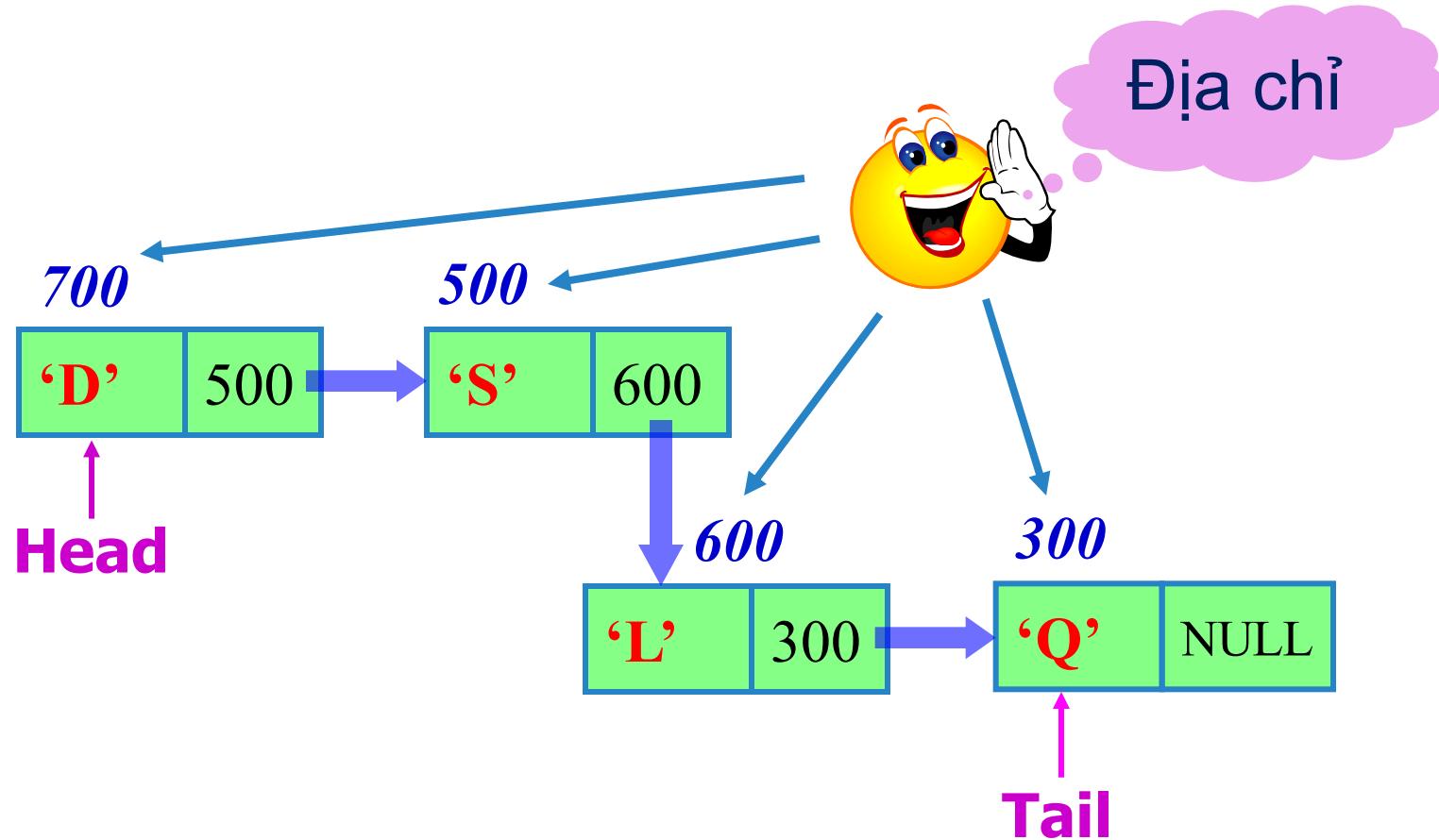
- **Thành phần dữ liệu (Data):** Lưu trữ thông tin phần tử (**Information**).



- **Thành phần liên kết (Link):** Lưu địa chỉ phần tử đứng kế sau (**Next**) trong danh sách, hoặc lưu trữ giá trị **NULL** nếu là phần tử cuối danh sách.

Minh Họa DSLK đơn

Ví dụ:



3.1. Khai báo Node và SList

Giả thiết cho dữ liệu lưu trữ trong List là số nguyên

❖ Khai báo Node của DSLK đơn:

```
struct Node // (Node: chứa thông tin một phần tử của dslk)
{
    <Data> info; // Data là kiểu dữ liệu của dữ liệu
                  // bài học giả thiết data là số int
    Node* next;
};
```

❖ Khai báo Danh sách liên kết đơn (SList): chỉ cần lưu địa chỉ 2 phần tử head và tail

```
struct SList // SList-Single List: dslk đơn
{
    Node *head;
    Node *tail;
};
```

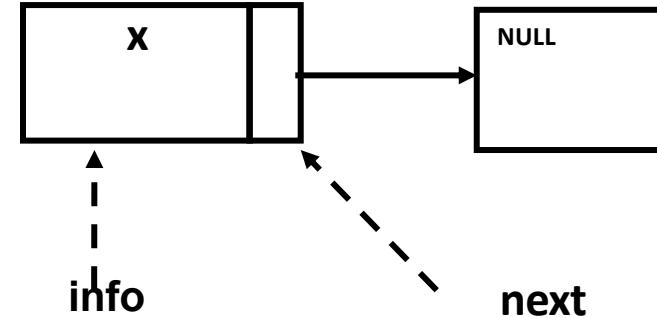
Ví dụ

- ❖ Khai báo DSLK đơn chứa các loại dữ liệu sau:
 1. Số thực.
 2. Phân số.
 3. Sinh viên (mã sv, họ tên, ngày sinh (kiểu Date), địa chỉ).
 4. Lớp (mã lớp, sỉ số, danh sách sinh viên (kiểu sv ở câu 3)).

3.2. Các thao tác cơ bản trên SList

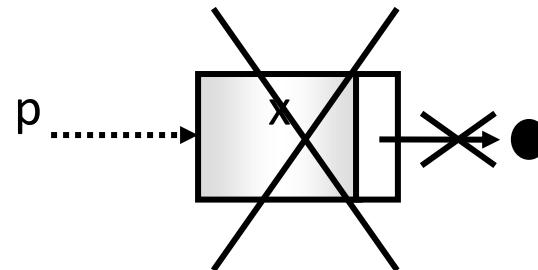
1. Hàm tạo Node

```
Node *createNode(int x)
{
    Node *p = new Node;
    if(p==NULL)
    {
        printf("Không đủ bộ nhớ để cấp phát !");
        return NULL;
    }
    p->info=x;
    p->next=NULL;
    return p;
}
```



2. Hàm xóa Node

```
void deleteNode(Node * &p)
{
    if (p==NULL)
        return;
    p->next=NULL;
    delete p;
}
```



3. Khởi tạo danh sách liên kết đơn

```
void createSList(SList &l, int n)
{
    int x;
    Node *q;
    l.head = l.tail = NULL;
    for(int i=1;i<=n;i++)
    {
        printf("Nhập phần tử thứ %d là:",i);
        scanf("%d", &x);
        q=createNode(x);
        if(q==NULL)
        {
            printf("Không đủ bộ nhớ cấp phát!");
            getch();
            return;
        } //tiếp theo ở khung kê
```

```
//nối p vào danh sách l
if(l.head==NULL)
    l.head=l.tail=q;
else
{
    l.tail->next=q;
    l.tail=q;
}
} //endfor
} //end-func
```

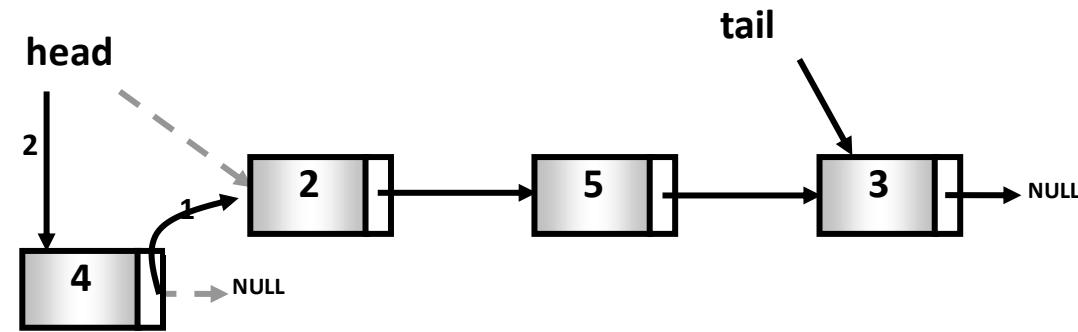
4. Tìm kiếm x trên dslk

```
Node *searchSList(SList l,int x) // duyệt list bằng while
{
    Node *p;
    p=l.head;
    while(p)
    {
        if(p->info==x) return p;
        p=p->next;
    }
    return NULL;
}
```

```
Node *searchSList(SList l,int x) //duyệt list bằng for
{
    for(Node *p=l.head;p!=NULL, p=p->next)
        if(p->info==x)
            return p;
    return NULL;
}
```

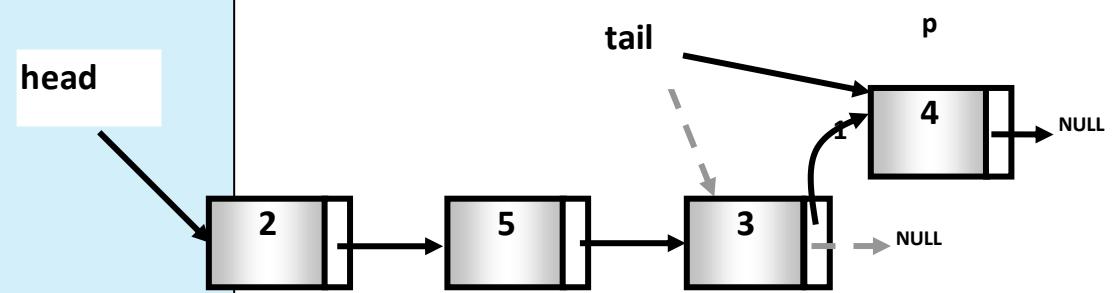
5.Thêm node vào đầu ds lk

```
void addHeadSList(SList &l, Node* p)
{
    if(p==NULL)
        return; //p rỗng thì không thêm
    else
        if(l.head==NULL) //ds rỗng
            l.head = l.tail = p;
        else
        {
            p->next = l.head;
            l.head = p;
        }
}
```



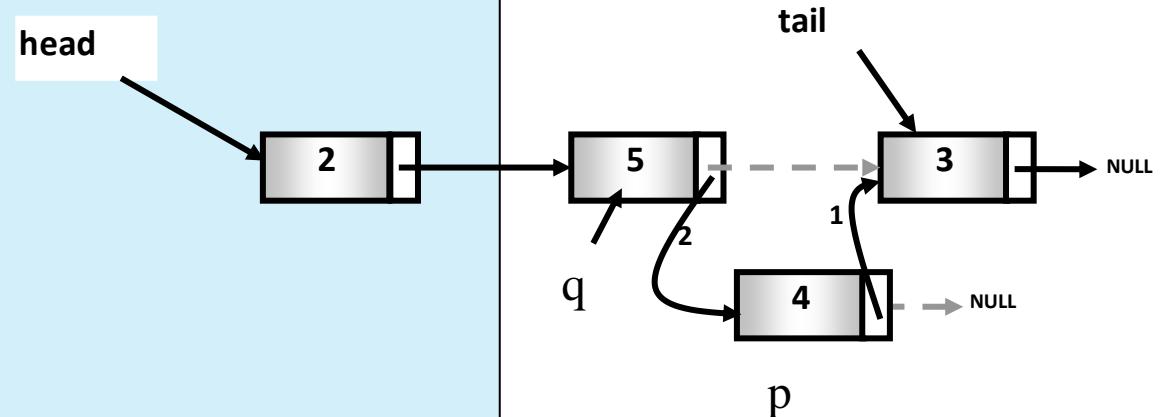
6.Thêm node vào cuối ds lk (tt)

```
void addTailSLList(SList &l, Node* p)
{
    if(p==NULL)
        return;
    else
        if(l.head==NULL)
            l.head = l.tail = p;
        else
        {
            l.tail->next = p;
            l.tail = p;
        }
}
```



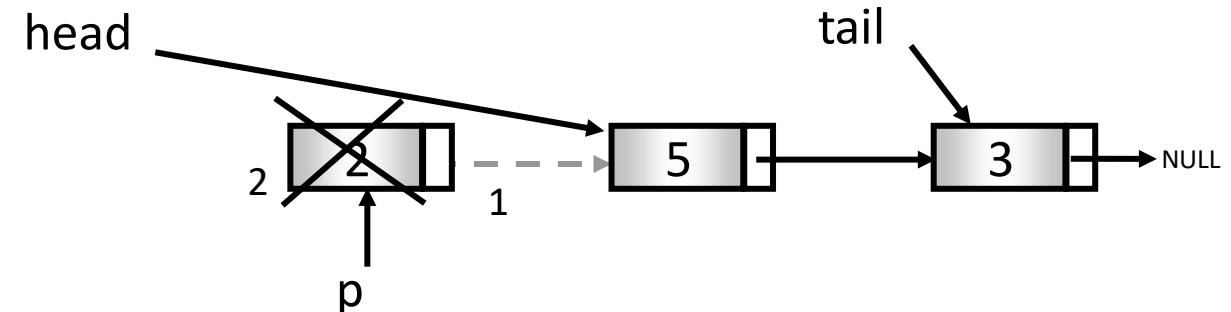
7. Thêm node p vào sau node q của dslk

```
void addAfterNodeSList (SList &l, Node* q, Node* p)
{
    if(p==NULL || q==NULL)
        return;
    else
        if(l.head==NULL) //
            l.head = l.tail = p;
        else
            if(q==l.tail)
                addTailSList(l,p);
            else
            {
                p->next = q->next;
                q->next = p;
            }
}
```



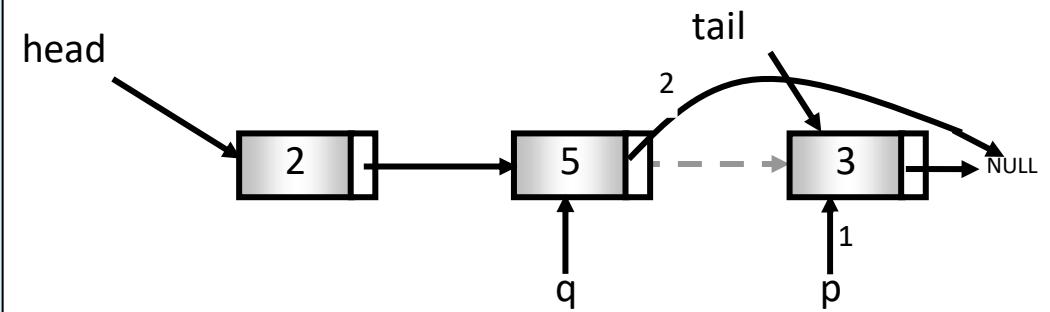
8. Xóa node đầu trong ds lk

```
void deleteHeadSList (SList &l)
{
    if(l.head==NULL)
        return ;
    else
    {
        Node* p=l.head;
        l.head = p->next;
        p->next = NULL;
        delete p;
    }
}
```



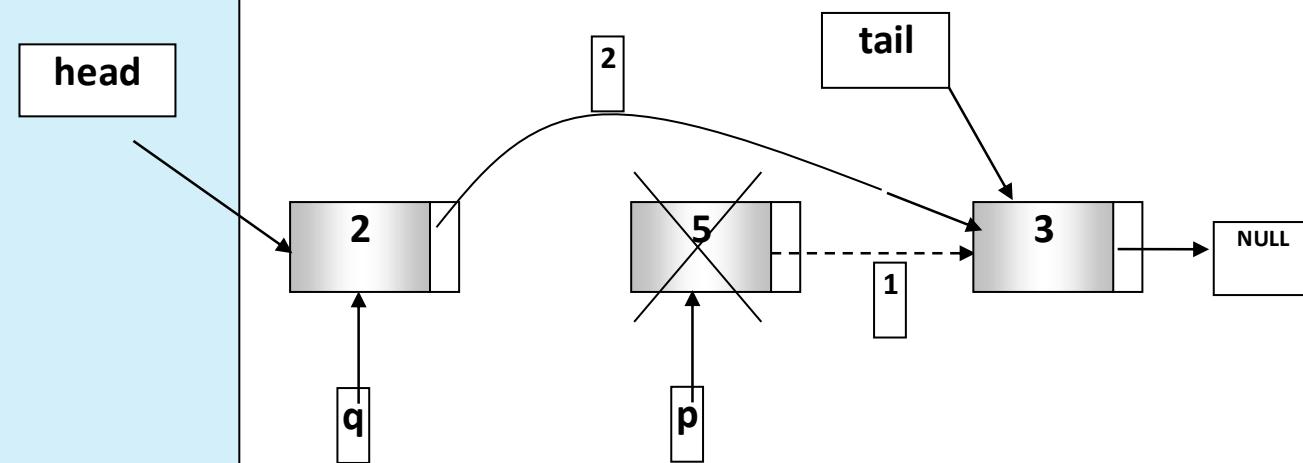
9. Xóa node cuối trong ds lk

```
void deleteTailList (SList &l)
{
    if(l.head==NULL)
        return;
    else
    {
        Node* p = l.tail;
        //tim node nam ngay truoc tail
        SNode *q = l.head;
        while(q->next!=l.tail)
            q=q->next;
        //cat node tail roi l và xoa
        l.tail = q;
        l.tail->next = NULL;
        delete p;
    }
}
```



10. Xóa node nằm sau node q ds lk (tt)

```
void deleteMidList (SList &l, Node* q)
{
    if(l.head==NULL || q==NULL || q==l.tail)
        return ;
    else
    {
        if(q->next == l.tail)
            deleteTailList(l);
        else
        {
            Node* p=q->next;
            q->next = p->next;
            p->next = NULL;
            delete p;
        }
    }
}
```



11. Sắp xếp tăng ds lk theo giải thuật InterchangeSort

```
void interchangeSortSList (SList l)
{
    for (Node* p = l.head; p!=l.tail; p = p->next)
        for (Node* q = p->next; q!=NULL; q=q->next)
            if (p->info > q->info)
                swap (p->info, q->info)
}
```

Bài tập

Cài đặt các hàm sau:

1. Xuất thông tin các phần tử trong dslk
2. Thêm node vào dslk (thêm ở đầu/giữa/cuối)
3. Xóa node trong dslk (xóa ở đầu/cuối/giữa)
4. Sắp xếp dslk tăng/giảm
5. Đếm số phần tử của dslk.
6. Tính tổng các phần tử của dslk
7. Xuất các phần tử chẵn/lẻ
8. Xuất các phần tử ở vị trí chẵn/lẻ
9. Tìm kiếm x trên dslk
10. Đếm trên dslk I có bao nhiêu x.

Bài tập (tt)

Cài đặt các hàm sau:

11. Đếm số phần tử trên dslk l
12. Tìm phần tử max/min trên dslk.
13. Tìm số chẵn nhỏ nhất trên dslk
14. Tìm số lẻ lớn nhất trên dslk.
15. Tìm số chẵn lớn nhất trong dslk, nếu dslk không có số chẵn thì trả về -1.
16. Tìm số lẻ nhỏ nhất trong dslk, nếu dslk không có số lẻ thì trả về 0.
17. Xóa phần tử max/min của dslk
18. Thêm x vào dslk sao cho x là phần tử cực đại (lớn hơn các phần tử kề nó)
19. Tìm dãy con dài nhất của dslk
20. Kiểm tra dslk có tăng dần/giảm dần không?





CẤU TRÚC DỮ LIỆU & GIẢI THUẬT (DATA STRUCTURES & ALGORITHMS)

NGĂN XẾP (STACK) & HÀNG ĐỢI (QUEUE)

Khoa Công nghệ thông tin
Bộ môn Công nghệ phần mềm

NỘI DUNG

I. STACK

1. KHÁI NIỆM
2. KHAI BÁO
3. CÁC THAO TÁC CƠ BẢN
4. CÁC BÀI TOÁN ỨNG DỤNG

II. QUEUE

1. KHÁI NIỆM
2. KHAI BÁO
3. CÁC THAO TÁC CƠ BẢN
4. CÁC BÀI TOÁN ỨNG DỤNG

I. Ngăn xếp - STACK

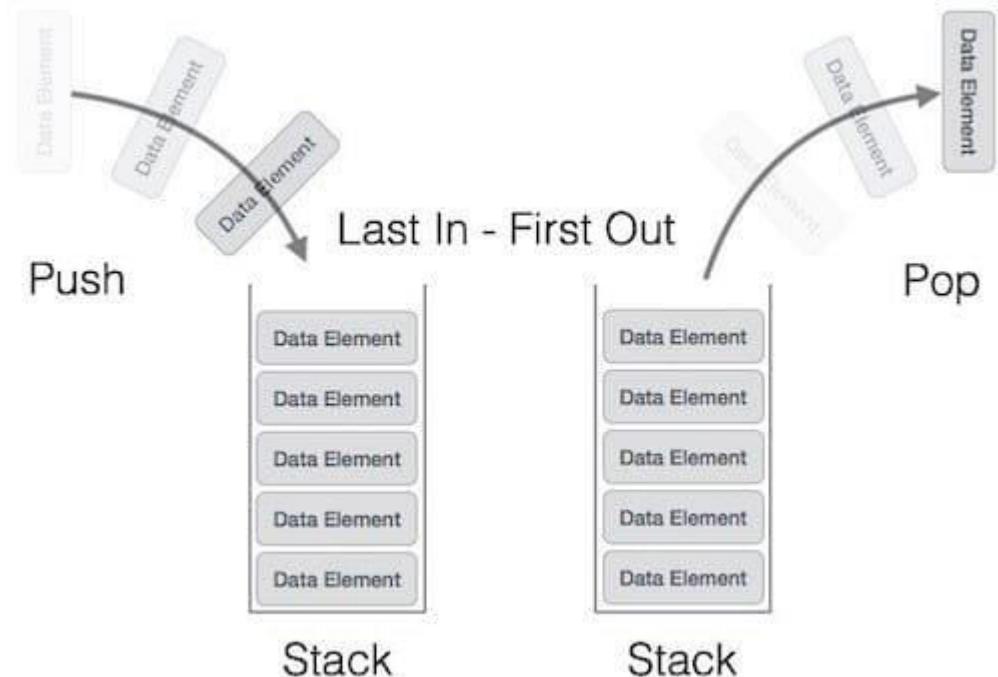
❖ Stack là một cấu trúc dữ liệu tuyến tính, tuân theo một thứ tự cụ thể trong đó các hoạt động được thực hiện theo nguyên tắc: vào sau ra trước (**Last In First Out – LIFO**).

❖ Có hai cách để triển khai và xây dựng

một stack đó là:

❖ Sử dụng mảng (phức tạp khi thêm/xóa phần tử).

❖ Sử dụng danh sách liên kết

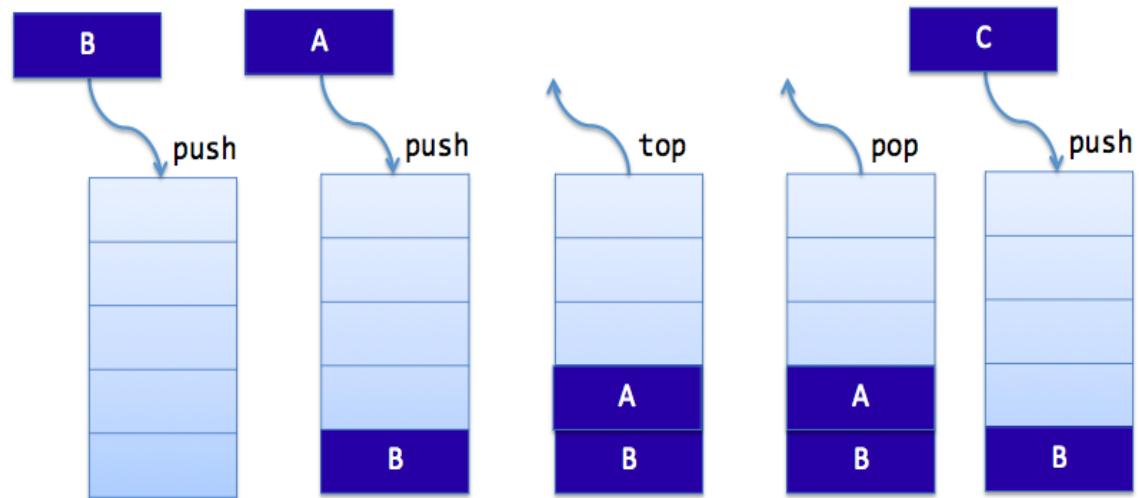


Ứng dụng của Stack

- ❖ Quản lý quá trình các lệnh gọi chức năng trong hệ điều hành.
- ❖ Tính năng undo (hoàn lại) ở nhiều nơi như chỉnh sửa, photoshop, MS office.
- ❖ Tính năng chuyển tiếp và lùi trong trình duyệt web.
- ❖ Được sử dụng trong nhiều thuật toán như Tower of Hanoi, duyệt cây, bài toán nhịp cổ phiếu, bài toán biểu đồ.
- ❖ Các ứng dụng khác có thể là Backtracking, Knight tour problem, N queen problem và sudoku solver
- ❖ Trong các thuật toán đồ thị như Sắp xếp theo cấu trúc liên kết và các thành phần được kết nối mạnh mẽ.
- ❖ Chuyển đổi Infix to Postfix
- ❖ Khử đệ quy đuôi

Thao tác cơ bản trên Stack

- ❖ initStack: khởi tạo Stack.
- ❖ isEmpty: kiểm tra Stack rỗng?
- ❖ isFull: kiểm tra Stack đầy?
- ❖ push: thêm 1 phần tử vào Stack.
- ❖ pop: lấy ra 1 phần tử khỏi Stack.
- ❖ top: xem phần tử trên đỉnh stack



Một số cài đặt cơ bản xây dựng Stack dùng DSLK đơn

Khai báo Stack

```
struct SNode
{ <Data> info; //giả sử Stack chứa số int
  SNode *next;
};

struct Stack
{
  SNode* top; //node đầu của ds lk
};
```

Lấy 1 phần tử ra khỏi Stack: pop

```
SNode *popS(Stack &s)
{
  SNode *p=NULL;
  if(s.top!=NULL)
  {
    p=s.top;
    s.top=s.top->next;
    p->next=NULL;
  }
  return p;
}
```

Thêm 1 phần tử vào Stack : push

```
void pushS(Stack &s, SNode *p)
{
  if(p!=NULL)
    if(s.top==NULL) s.top=p;
    else
    {
      p->next=s.top;
      s.top=p;
    }
}
```

Bài tập Stack

Bài 1. SV cài đặt các thao tác còn lại của Stack

Viết chương trình thực hiện:

- a. Nhập n phần tử đưa vào stack.
- b. Xuất n các giá trị lần lượt lấy ra từ stack.
- c. Cho stack rỗng. Thực hiện n lần nhóm thao tác sau: đưa vào stack 2 phần tử và lấy trong stack 1 phần tử. Hãy xuất dãy các phần tử trong stack và dãy các phần tử được lấy ra.

Bài 2. Dùng Stack hỗ trợ chuyển đổi cơ số (từ hệ 10 sang hệ 2, 8, 16)

→ gợi ý ở các slide tiếp theo

Bài tập 2. Đổi cơ số

❖ Các hệ thống số

Name	Base	Symbols
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Binary	2	0, 1
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
Octal	8	0, 1, 2, 3, 4, 5, 6, 7

System	Base	Value	Conversion Formula	Decimal Value
Decimal	10	104	$1 \times 10^2 + 0 \times 10^1 + 4 \times 10^0$	$100 + 0 + 4 = 104$
Binary	2	111	$1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	$4 + 2 + 1 = 7$
Octal	8	104	$1 \times 8^2 + 0 \times 8^1 + 4 \times 8^0$	$64 + 0 + 4 = 68$
Hexadecimal	16	FEC	$F \times 16^2 + E \times 16^1 + C \times 8^0$	$15 \times 256 + 14 \times 16 + 12 \times 1 = 3840 + 224 + 12 = 4076$

Tổng quát

$$a_k b^k + a_{k-1} b^{k-1} + \dots + a_1 b^1 + a_0 b^0$$

Bài tập 2. Đổi cơ số (10 sang 2, 8, 16)

- ❖ Input: n – decimal number, b – base of number
- ❖ Output: m – number (other base)

Giải thuật

```
S = {}  
while (n > 0) {  
    r = n % b  
    n = n / b  
    push (S, r)  
}  
while (!isEmpty(S)) {  
    x = pop (S)  
    printf("%d", x)  
}
```

$$25_{10} = 11001_2$$

Stack

n	n / 2	r = n % 2
25	12	1
12	6	0
6	3	0
3	1	1
1	0	1

Bài tập (tt)

Bài tập 3. Tìm dấu ngoặc đơn trùng lặp (dư thừa) trong một biểu thức

Đưa ra một biểu thức cân bằng có thể chứa dấu ngoặc mở và đóng, hãy kiểm tra xem biểu thức có chứa bất kỳ dấu ngoặc đơn trùng lặp nào hay không

Ví dụ:

- ❖ Input: $((x+y))+z$
- ❖ Output: Trùng lặp () được tìm thấy trong biểu thức con $((x + y))$
- ❖ Input: $(x+y)$
- ❖ Output: Không thấy trùng lặp ()
- ❖ Input: $((x+y)+((z)))$
- ❖ Output: Trùng lặp () được tìm thấy trong biểu thức con $((z))$

Gợi ý bài tập 3

Chúng ta có thể sử dụng một ngăn xếp để giải quyết vấn đề này. Ý tưởng là đi qua từng ký tự:

- ❖ Nếu ký tự hiện tại trong biểu thức không phải là một dấu ngoặc đóng, thì ta đẩy ký tự vào ngăn xếp.
- ❖ Nếu ký tự hiện tại trong biểu thức là một dấu ngoặc đóng, ta sẽ kiểm tra xem phần tử trên cùng trong ngăn xếp có phải là một dấu ngoặc mở hay không. Nếu nó đang mở ngoặc đơn, thì biểu thức con kết thúc ở ký tự hiện tại có dạng ((exp)), nếu không, ta tiếp tục xuất các ký tự từ ngăn xếp cho đến khi tìm thấy kết quả phù hợp cho hiện tại.

Bài tập (tt)

Bài tập 4. Kiểm tra xem biểu thức đã cho có phải là biểu thức cân bằng hay không

Cho một chuỗi chứa dấu ngoặc nhọn mở và đóng, hãy kiểm tra xem nó có đại diện cho một biểu thức cân bằng hay không?

❖ Ví dụ:

- ❖ $\{[\{\} \}]\ [(), \{\} \}, [] \{} ()$ là các biểu thức cân bằng.
- ❖ $\{()\} [, \{()\}$ không cân bằng.

❖ **Gợi ý: Chúng ta có thể sử dụng một ngăn xếp để giải quyết vấn đề này. Chúng ta duyệt qua biểu thức đã cho và**

- ❖ Nếu ký tự hiện tại trong biểu thức là một dấu ngoặc mở, chúng ta đẩy nó vào ngăn xếp.
- ❖ Nếu ký tự hiện tại trong biểu thức là một dấu ngoặc nhọn đóng, chúng ta lấy một ký tự từ ngăn xếp và chúng ta trả về `false` nếu ký tự được lấy không giống với ký tự hiện tại hoặc nó không ghép nối với ký tự hiện tại của biểu thức. Ngoài ra, nếu ngăn xếp được tìm thấy là trống, điều đó có nghĩa là số lượng dấu ngoặc nhọn mở ít hơn số lượng dấu ngoặc nhọn đóng tại điểm đó, do đó biểu thức không thể được cân bằng.

Bài tập 5. Ký pháp Ba Lan (SV đọc thêm – dành làm bài tập lớn)

Ký pháp Ba Lan: chuyển biểu thức trung tố → hậu tố, tính giá trị biểu thức hậu tố.

- ❖ Thuật toán Ba Lan ngược (Reverse Polish Notation – RPN)
- ❖ Định nghĩa RPN: Biểu thức toán học trong đó các toán tử được viết sau toán hạng và không dùng dấu ngoặc.
- ❖ Phát minh bởi Jan Lukasiewics một nhà khoa học Ba Lan vào những năm 1950.

Ký pháp Ba Lan (tt)

Infix : toán tử viết giữa toán hạng

Postfix (RPN) : toán tử viết sau toán hạng

Prefix : toán tử viết trước toán hạng

Ví dụ:

INFIX

A + B

A * B + C

A * (B + C)

A - (B - (C - D))

A - B - C - D

RPN (POSTFIX)

A B +

A B * C +

A B C + *

A B C D - - -

A B - C - D -

PREFIX

+ A B

+ * A B C

*** A + B C**

- A - B - C D

- - - A B C D

Ký pháp Ba Lan (tt)

Kỹ thuật gạch dưới để tính giá trị của biểu thức hậu tố

1. Duyệt từ trái sang phải của biểu thức cho đến khi gặp toán tử.
2. Gạch dưới 2 toán hạng ngay trước toán tử và kết hợp chúng bằng toán tử trên.
3. Lặp đi lặp lại cho đến hết biểu thức.

Ví dụ: $2 * ((3 + 4) - (5 - 6))$

→ 2 3 4 + 5 6 - - *

→ 2 3 4 + 5 6 - - *

→ 2 7 5 6 - - *

→ 2 7 5 6 - - *

→ 2 7 -1 - - *

→ 2 7 -1 - - * → 2 8 * → 2 8 * → 16

Bài 5.1. Chuyển infix thành postfix (trung tố → hậu tố)

1. Khởi tạo Stack rỗng (chứa các phép toán).
2. Lặp cho đến khi kết thúc biểu thức:
 - Đọc 01 phần tử của biểu thức
(01 phần tử có thể là hằng, biến, phép toán, ')' hay '(').
 - Nếu phần tử là:
 - 2.1 ')' : đưa vào Stack.
 - 2.2 ')' : lấy các phần tử của Stack ra **cho đến khi lấy xong '('**.
 - 2.3 Một phép toán: **^ + - * /**
 - Nếu **Stack rỗng**: đưa vào Stack.
 - Nếu Stack khác rỗng và **phép toán có độ ưu tiên cao hơn phần tử ở đầu Stack**: đưa vào Stack.
 - Nếu Stack khác rỗng và **phép toán có độ ưu tiên thấp hơn hoặc bằng phần tử ở đầu Stack**:
 - Lấy phần tử ở đỉnh Stack ra;
 - Sau đó lặp lại việc so sánh với phần tử ở đỉnh Stack.
 - 2.4 Hằng hoặc biến: đưa vào postfix.
 - 3. Lấy hết tất cả các phần tử của Stack ra.

Độ ưu tiên:

^	cao nhất
* , /	cao nhì
+ , -	cao ba

Ví dụ:

- Push (
- Display A
- Push +
- Display B
- Push *
- Display C
- Read)
- Pop *, Display *,
Pop +, Display +, Pop (
- Push /
- Push (
- Display D
- Push -
- Push (
- Display E
- Push -
- Display F
- Read)
- Pop -, Display -, Pop (
- Read)
- Pop -, Display -, Pop (
- Pop /, Display /

(A+B*C) / (D- (E-F))

PostFix

A

AB

ABC

ABC*

ABC*+

ABC*+D

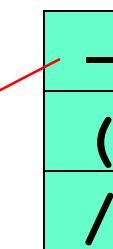
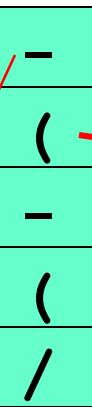
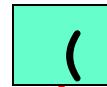
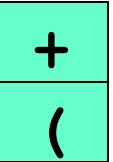
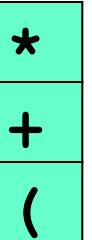
ABC*+DE

ABC*+DEF

ABC*+DEF-

ABC*+DEF--

ABC*+DEF---



—

Minh họa các bước thực hiện

Giải thuật chuyển biểu thức trung tố sang hậu tố

Input: P là biểu thức trung tố ban đầu.

Output: Q là biểu thức kết quả dạng hậu tố

```
S = {}  
- push(S, '(');  
- Thêm ')' vào P  
while (chưa đi hết biểu thức P) {  
    1. đọc 1 kí tự x trong P (trái sang phải)  
    2. if (x là toán hạng)  
        Thêm x vào Q  
    3. if (x là dấu ngoặc mở)  
        push(S, x)  
    4. if (x là toán tử)  
        4.1. while(độ ưu tiên (top) >= độ ưu tiên(x))  
            4.1.1. w = pop(S)  
            4.1.2. Thêm w vào Q  
        4.2 push(x)  
    5. if (x là dấu ngoặc đóng)  
        5.1. while( chưa gặp ngoặc mở)  
            5.1.1. w = pop(S)  
            5.1.2. Thêm w vào Q  
        5.2. pop() // loại '(' ra khỏi stack  
}
```

Chuyển biểu thức trung tố sang hậu tố

$(2 * 3 + 8 / 2) * (5 - 1)$

Đọc	Xử lý	Stack	postfix
(Đẩy vào Stack	(
2	Bỏ vào postfix	(2
*	Do ở đỉnh Stack chứa '(' nên bỏ dấu '*' vào Stack.	(*	2
3	Bỏ vào postfix	(*	2 3
+	Do '+' có độ ưu tiên thấp hơn '*' ở đỉnh Stack nên ta lấy '*' ra và bỏ vào postfix. Do ở đỉnh Stack chứa '(' nên bỏ dấu '+' vào Stack.	(+	2 3 *
8	Bỏ vào postfix	(+	2 3 * 8
/	Do '/' có độ ưu tiên cao hơn '+' ở đỉnh Stack nên đưa '/' vào Stack.	(+ /	2 3 * 8
2	Bỏ vào postfix	(+ /	2 3 * 8 2

Chuyển biểu thức trung tố sang hậu tố (tt)

(2 * 3 + 8 / 2) * (5 – 1)

Đọc	Xử lý	Stack	postfix
)	Lấy trong Stack ra cho đến khi gặp ngoặc ‘(’		2 3 * 8 2 / +
*	Đưa vào Stack	*	2 3 * 8 2 / +
(Đưa vào Stack	* (2 3 * 8 2 / +
5	Bỏ vào postfix	* (2 3 * 8 2 / + 5
-	Do trên đỉnh Stack chứa ‘(’ nên bỏ dấu ‘-’ vào Stack.	* (-	2 3 * 8 2 / + 5
1	Bỏ vào postfix	* (-	2 3 * 8 2 / + 5 1
)	Lấy trong Stack ra cho đến khi gặp ngoặc mở ‘(’	*	2 3 * 8 2 / + 5 1 -
	Lấy những phần tử còn lại trong Stack và bỏ vào Stack.		2 3 * 8 2 / + 5 1 - *

Bài tập

Chuyển biểu thức P sang hậu tố Q.

Sau đó tính giá trị Q.

❖ $P = (4 * 5 - (1 + 9) / 2) \rightarrow Q ?$

❖ $P = (A + B) * (C - (D + A)) \rightarrow Q?$

❖ $P = A + (B * C - D / E ^ F * G) * H \rightarrow Q?$

Bài 5.2.Tính giá trị biểu thức hậu tố

1. Khởi tạo Stack rỗng (*chứa hằng hoặc biến*).
2. Lặp cho đến khi kết thúc biểu thức:
 - Đọc 01 phần tử của biểu thức (*hằng, biến, phép toán*).
 - Nếu phần tử là hằng hay biến: đưa vào Stack.
 - Ngược lại:
 - Lấy ra 2 phần tử của Stack ra.
 - Áp dụng phép toán cho 2 phần tử vừa lấy ra.
 - Đưa kết quả vừa tính được vào lại Stack.
3. Giá trị của biểu thức chính là phần tử cuối cùng của Stack.

$$2 * ((3 + 4) - (5 - 6))$$

Ví dụ:

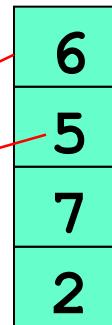
2 3 4 + 5 6 - - *

- Push 2
- Push 3
- Push 4
- Read +
- ➤ Pop 4, Pop 3,
- Push 7
- Push 5
- Push 6
- Read -
- ➤ Pop 6, Pop 5,
- Push -1
- Read -
- ➤ Pop -1, Pop 7,
- Push 8
- Read *
- ➤ Pop 8, Pop 2,
- Push 16
-

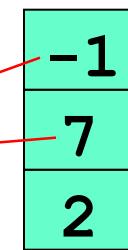
$$3 + 4 = 7$$



$$5 - 6 = -1$$



$$7 - -1 = 8$$



$$2 * 8 = 16$$



16

Bài 5.2.Tính giá trị biểu thức hậu tố

Input: P là biểu thức trung tố ban đầu.

Output: Q là biểu thức kết quả dạng hậu tố

```
S = {}  
while (đọc 1 ký tự x trong Q khác rỗng) {  
    0. x là một ký tự đang đọc  
    1. if (x là toán hạng)  
        push(S, x);  
  
    2. else if (x là toán tử) {  
        2.1. op2 = pop()  
        2.2. op1 = pop()  
        2.3. result = calc( x, op1, op2)  
        2.4 push(result)  
    }  
}  
return pop()
```

Tính giá trị biểu thức hậu tố: $2\ 3\ * \ 8\ 2\ / \ +\ 5\ 1\ -\ *$

Đọc	Xử lý	Stack	Output
2,3	Đưa vào Stack	2, 3	
*	$2*3$	6	
8	Đưa vào Stack	6, 8	
2	Đưa vào Stack	6, 8, 2	
/	Lấy 8/2	6, 4	
+	Lấy 6 + 4	10	
5	Đưa vào Stack	10, 5	
1	Đưa vào Stack	10, 5, 1	
-	Lấy 5 - 1	10, 4	
*	Lấy 10 * 4	40	40

Bài tập

- ❖ Tính giá trị các biểu thức sau:

$$4 \ 5 * 1 \ 9 + 2 / -\backslash$$

- ❖ Chuyển biểu thức sau về dạng hậu tố và tính giá trị biểu thức

$$3 * (2 + 6 * 2 / 3 - 1) - 2 * 3 / 2 + 1$$

II. HÀNG ĐỢI - QUEUE

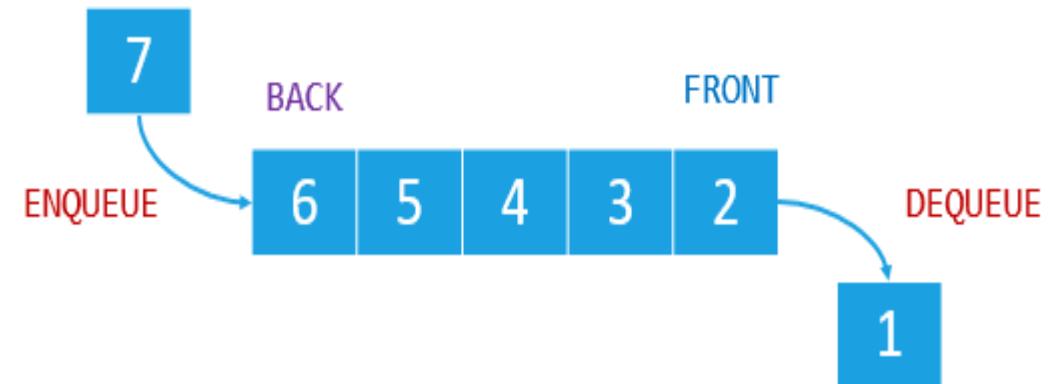
❖ Queue là một cấu trúc dữ liệu tuyến tính, tuân theo một thứ tự cụ thể trong đó các hoạt động được thực hiện theo nguyên tắc: vào trước ra trước (**First In First Out – FIFO**).

❖ Có hai cách để triển khai và xây dựng

một Queue đó là:

❖ Sử dụng mảng (phức tạp khi thêm/xóa phần tử).

❖ Sử dụng danh sách liên kết

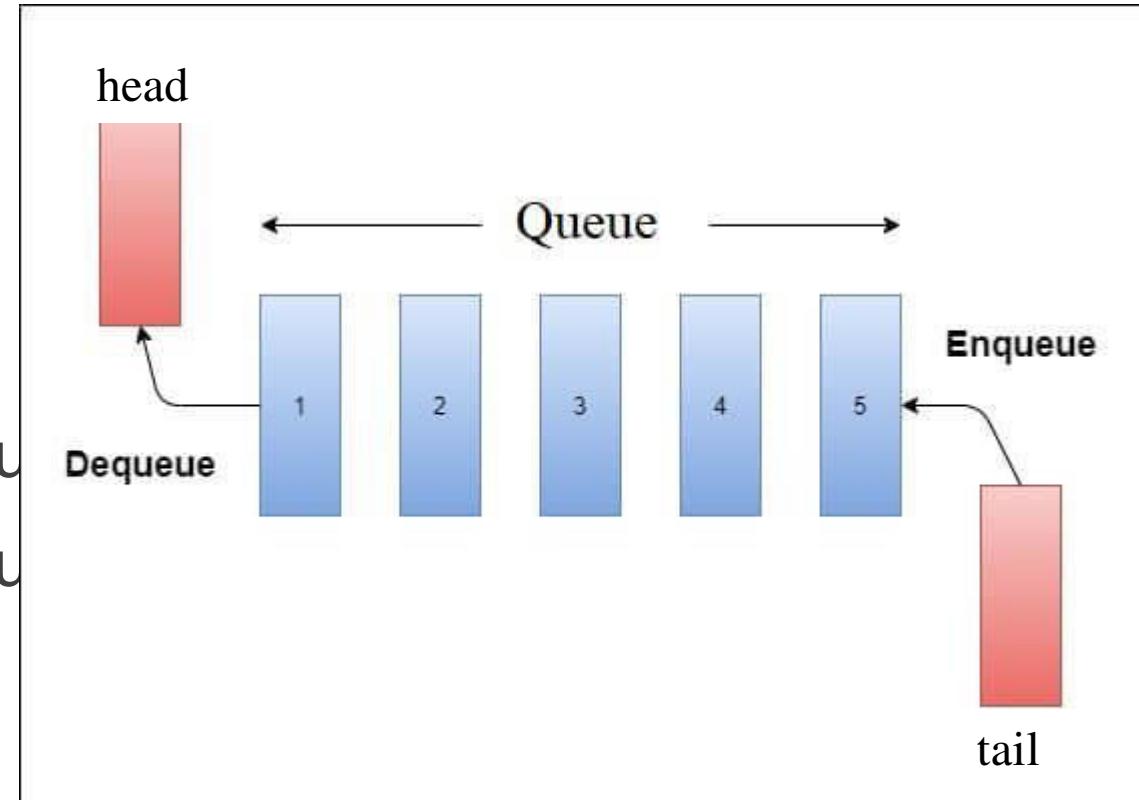


Ứng dụng của Queue

- ❖ Queue được sử dụng bất cứ khi nào chúng ta cần quản lý bất kỳ nhóm đối tượng nào theo thứ tự mà đối tượng đầu tiên đến, sẽ đưa ra trước, những đối tượng khác chờ đến lượt.
- ❖ Queue được ứng dụng các trường hợp sau:
 - ❖ Cung cấp các yêu cầu trên một tài nguyên được chia sẻ duy nhất, như máy in, lập lịch tác vụ CPU, ...
 - ❖ Trong thực tế, hệ thống điện thoại Call Center sử dụng Queue để xếp những người đang gọi cho theo thứ tự họ gọi đến.
 - ❖ ...

Thao tác cơ bản trên Queue

- ❖ initQueue: khởi tạo Queue.
- ❖ isEmpty: kiểm tra Queue rỗng?
- ❖ isFull: kiểm tra Queue đầy?
- ❖ Enqueue → push: thêm 1 phần tử vào Queue
- ❖ Dequeue → pop: lấy ra 1 phần tử khỏi Queue
- ❖ getHead: truy xuất phần tử đầu Queue
- ❖ getTail: truy xuất phần tử cuối Queue



Một số cài đặt cơ bản xây dựng Queue dùng DSLK đơn

Khai báo Queue

```
struct QNode
{ <Data> info; //giả sử Queue chứa số int
  QNode *next;
};

struct Queue
{
  QNode *head, *tail;
};
```

Lấy 1 phần tử ra khỏi Queue: pop (lấy phần tử ở đầu)

```
QNode *popQ(Queue &qu)
{
  QNode *p=NULL;
  if(qu.head!=NULL)
  {
    p=qu.head;
    qu.head=qu.head->next;
    p->next=NULL;
  }
  return p;
}
```

Thêm 1 phần tử vào Queue : push (thêm phần tử ở cuối)

```
void pushQ(Queue &qu, QNode *p)
{
  if(p!=NULL)
    if(qu.head==NULL) qu.head=p;
    else
    {
      qu.tail->next=p;
      qu.tail=p;
    }
}
```

Bài tập

Bài 1. SV tự cài đặt các hàm còn lại của Queue theo khai báo trên.

Bài 2. Trong phòng đào tạo trường HUFI, có 3 máy tính (client) cùng kết nối và dùng chung một máy in. Máy in thiết lập dãy các lệnh chờ in cho các file được chọn in từ client theo nguyên tắc của Queue. Giả sử có 3 mảng 1 chiều chứa dãy các file của 3 client.

$$A = \{a_1, a_2, a_3, \dots, a_n\}; B = \{b_1, b_2, \dots, b_n\}; C = \{c_1, c_2, \dots, c_n\}$$

Hãy viết chương trình minh họa việc xếp lịch chờ in và thứ tự in các tập tin từ 3 máy client:

- Tạo mảng 1 chiều D chứa dãy các tập tin chờ in từ 3 máy client.
- Chọn ngẫu nhiên các giá trị lần lượt của 3 mảng đưa vào D. → 3 client chọn in, queue xếp thứ tự trước sau dựa vào file nào của máy nào được chọn in trước
- Xuất các giá trị lấy ra từ D → thứ tự các file được in của 3 client

Bài tập (tt)

Bài 3. An có một cái điện thoại, màn hình điện thoại của An chỉ hiển thị được tối đa k tin nhắn theo thứ tự từ trên xuống. Màn hình của An hiện thị như sau:

- Không hiện thị 2 tin nhắn của cùng một số điện thoại (SĐT) trên cùng một khung hình, nếu SĐT A gửi tin nhắn đến mà trên màn hình đã có tin nhắn của SĐT A thì màn hình không thay đổi gì.
- Khi SĐT A gửi tin nhắn đến mà trên màn hình chưa có tin nhắn của SĐT A thì:
 - Nếu màn hình chưa đủ k tin nhắn thì tin nhắn của SĐT A sẽ được chèn vào đầu màn hình.
 - Nếu màn hình đã có đủ k tin nhắn thì màn hình sẽ đẩy tin nhắn dưới cùng ra và sau đó chèn tin nhắn của SĐT A vào đầu màn hình.

Cho dãy b là dãy các SĐT sẽ gửi tin nhắn cho An. Hỏi sau khi nhận được tin nhắn cuối cùng thì màn hình của An đang hiển thị tin nhắn của các SĐT nào, xuất ra theo thứ tự từ trên xuống dưới của màn hình.

Ví dụ: input: $b = 1, 2, 1, 3, 4; k=3$. \rightarrow output:

4
3
2



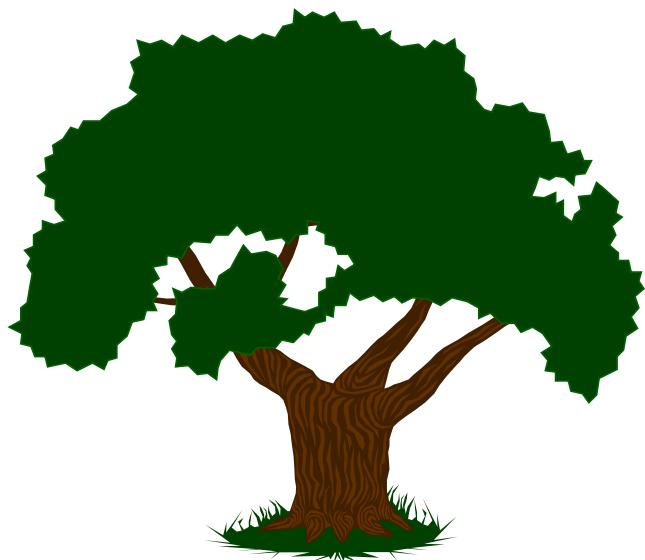


CẤU TRÚC DỮ LIỆU & GIẢI THUẬT (DATA STRUCTURES & ALGORITHMS)

Chương 3

CÂY

Khoa Công nghệ thông tin
Bộ môn Công nghệ phần mềm



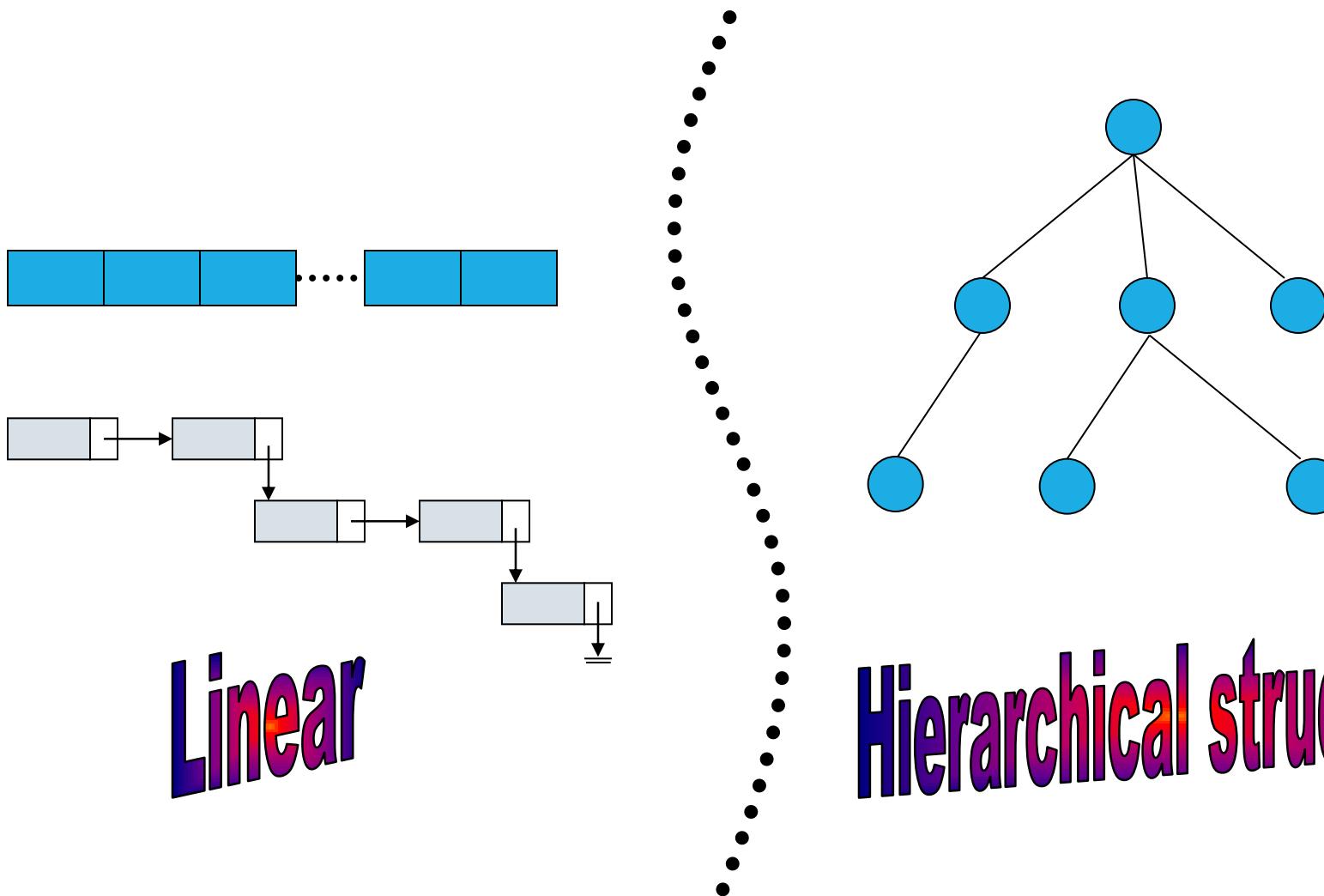
Nội dung

- Cấu trúc cây
- Cây nhị phân
- Cây nhị phân tìm kiếm
- Duyệt cây
- Cây AVL
- Cây đỏ đen
- Cây B-Tree

Nội dung

- **Cấu trúc cây**
- Cây nhị phân
- Cây nhị phân tìm kiếm
- Duyệt cây
- Cây AVL
- Cây đỏ đen
- Cây B-Tree

Cấu trúc dữ liệu



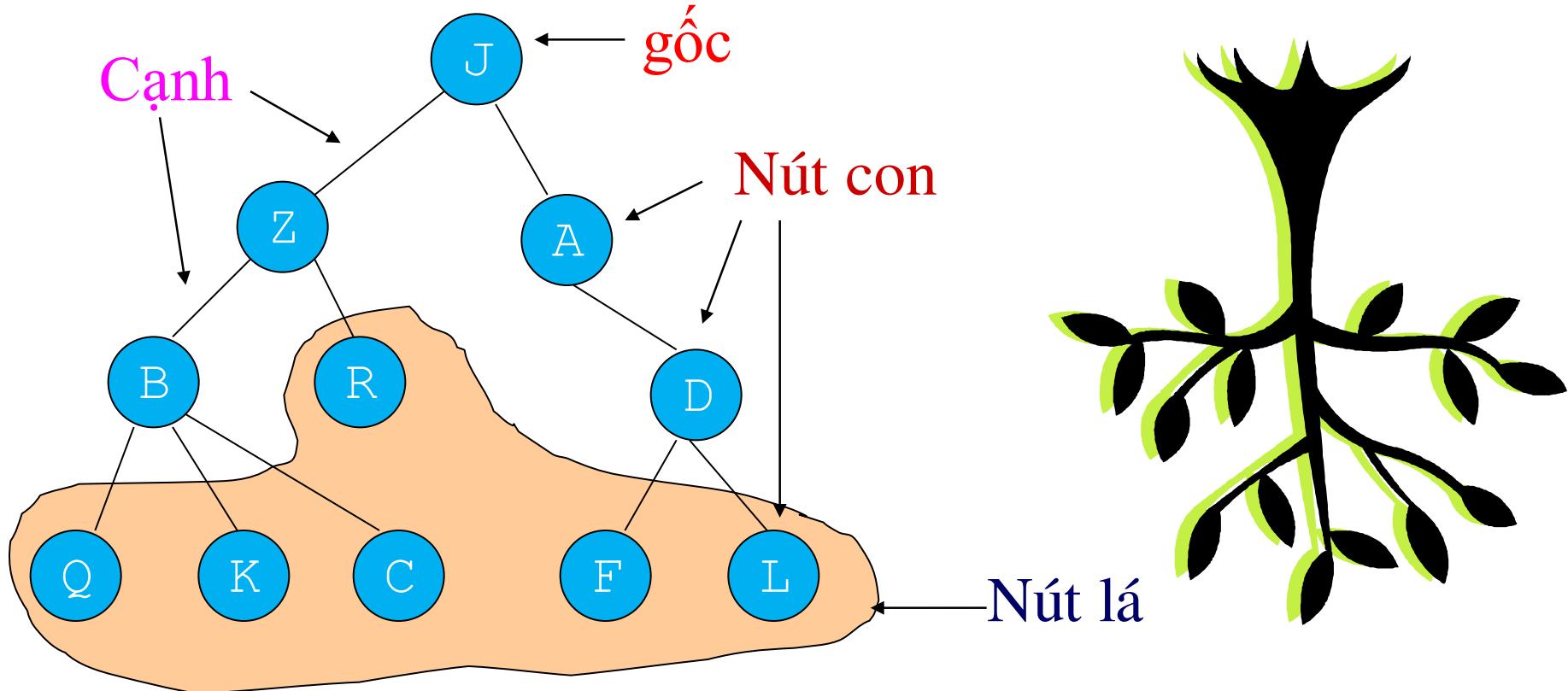
Cấu trúc cây

- Tập hợp các nút và cạnh nối các nút đó.
- Có một nút gọi là gốc.
- Quan hệ one-to-many giữa các nút.
- Có duy nhất một đường đi từ gốc đến nút con.
- Các loại cây:
 - **Nhi phân**: mỗi nút có $\{0, 1, 2\}$ nút con
 - **Tam phân**: mỗi nút có $\{0, 1, 2, 3\}$ nút con
 - **n-phân**: mỗi nút có $\{0, 1, \dots, n\}$ nút con

Cấu trúc cây



Khái niệm



Khái niệm

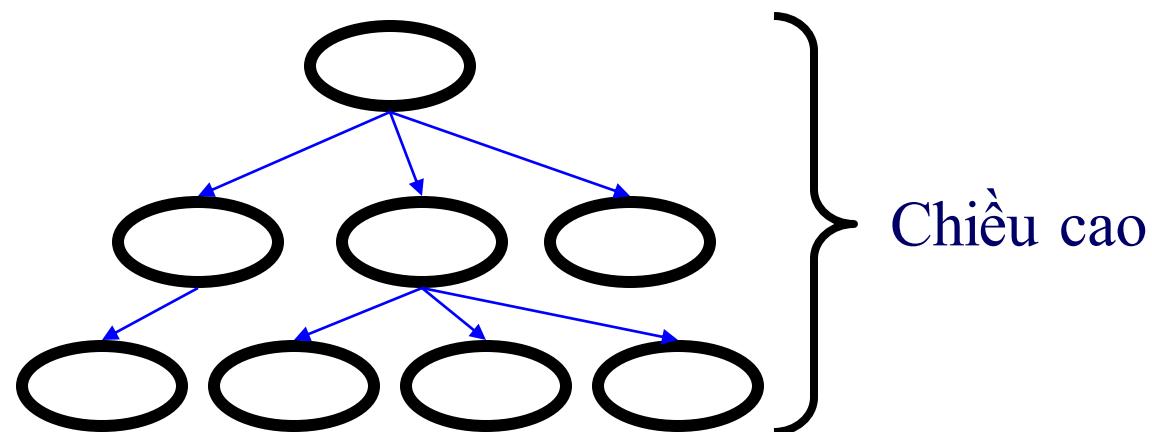
Thuật ngữ

- **Nút gốc**: không có nút cha
- **Nút lá**: không có nút con
- **Nút trong**: không phải nút lá và nút gốc
- **Chiều cao**: khoảng cách từ gốc đến lá của nhánh cao nhất

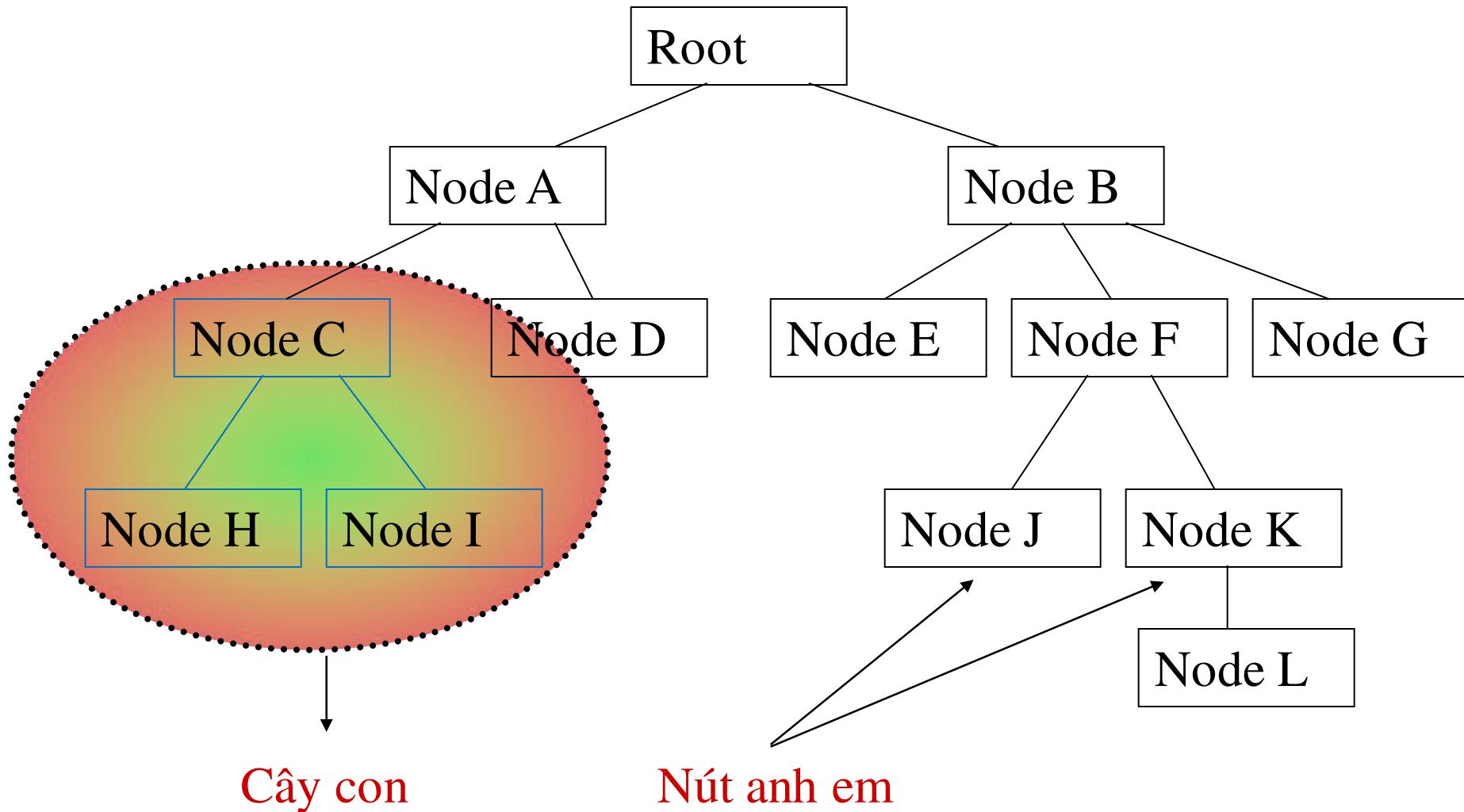
Nút gốc

Nút trong

Nút lá



Khái niệm



Nội dung

- Cấu trúc cây
- **Cây nhị phân**
- Cây nhị phân tìm kiếm
- Duyệt cây
- Cây AVL
- Cây đỏ đen
- Cây B-Tree

Cây nhị phân

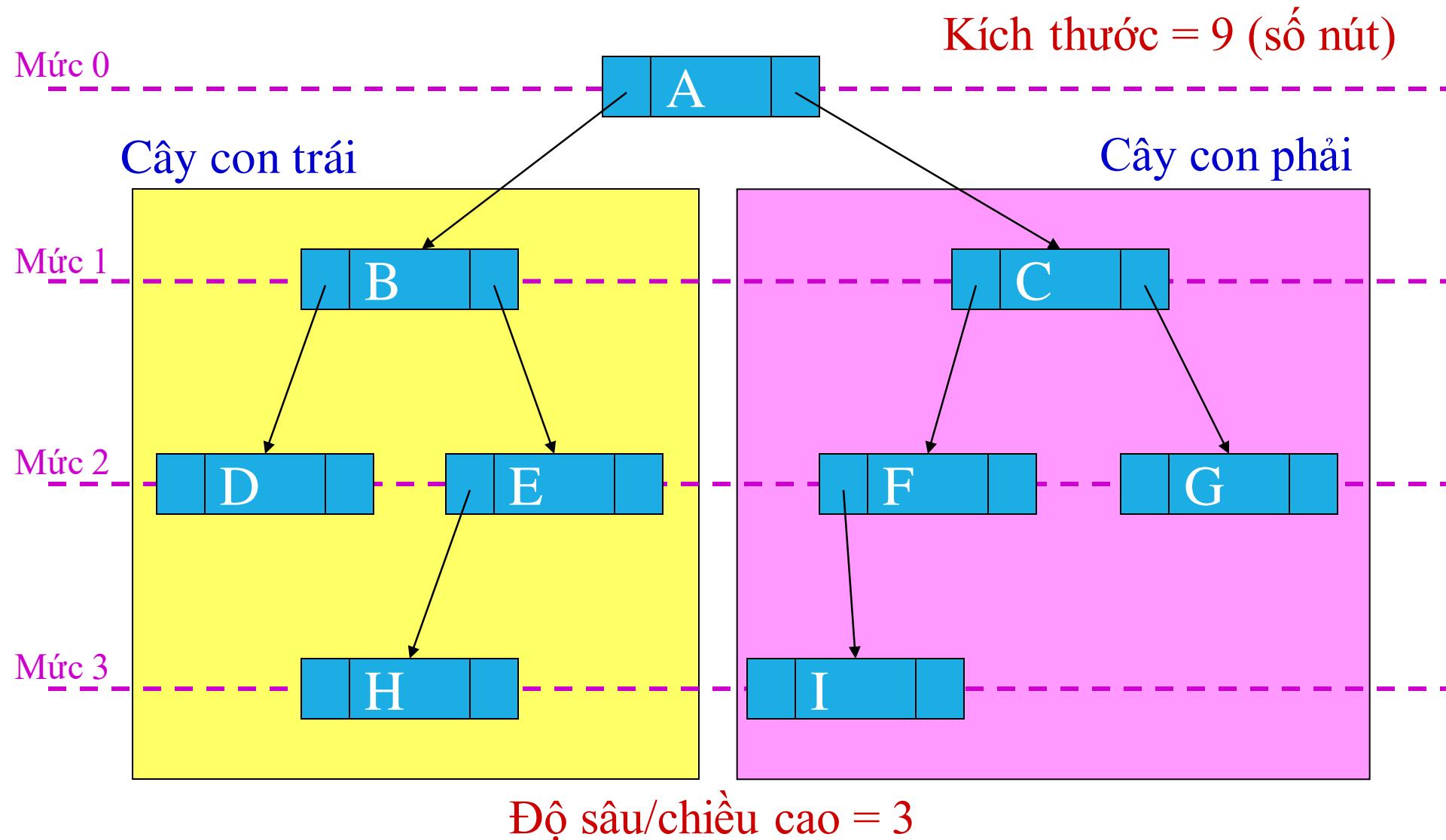
- Cây nhị phân là một đồ thị có hướng, mỗi nút có tối đa 2 nút con, hay *Cây nhị phân là cây có bậc bằng hai (bậc của mỗi nút tối đa bằng 2)*.
- Cấu trúc cây đơn giản nhất
- Tại mỗi nút gồm các 3 thành phần
 - Phần **Data**: chứa giá trị, thông tin của nút
 - Phần **Left**: Liên kết đến nút con trái (nếu có)
 - Phần **Right**: Liên kết đến nút con phải (nếu có)



Cây nhị phân

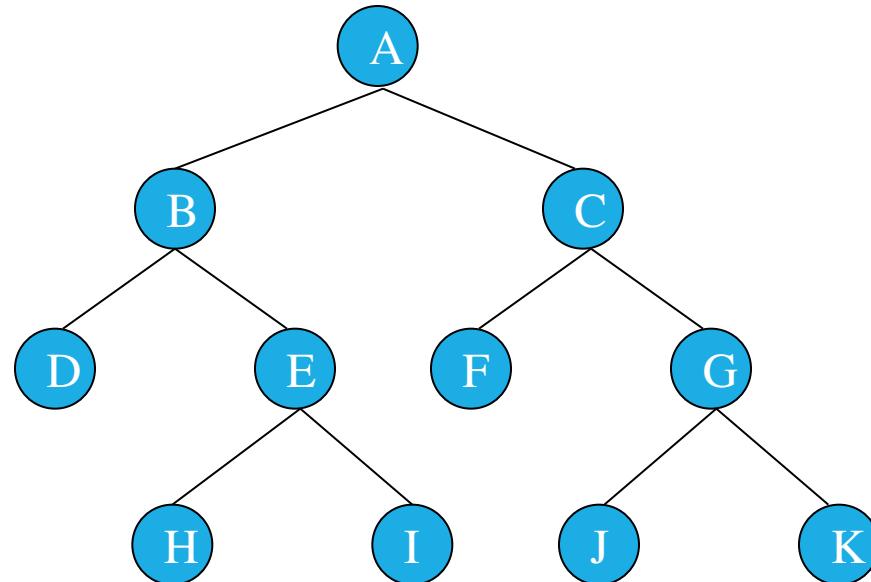
- Cây nhị phân có thể rỗng (không có nút nào)
- Cây nhị phân khác rỗng có 1 nút gốc
 - Có duy nhất 1 đường đi từ gốc đến 1 nút con.
 - Nút không có nút con bên trái và con bên phải là nút lá.

Cây nhị phân



Cây nhị phân

- Cây nhị phân đúng:
 - Nút gốc và nút trung gian có đúng 2 con
- Cây nhị phân đúng có n nút lá thì số nút trên cây $2^* n - 1$.



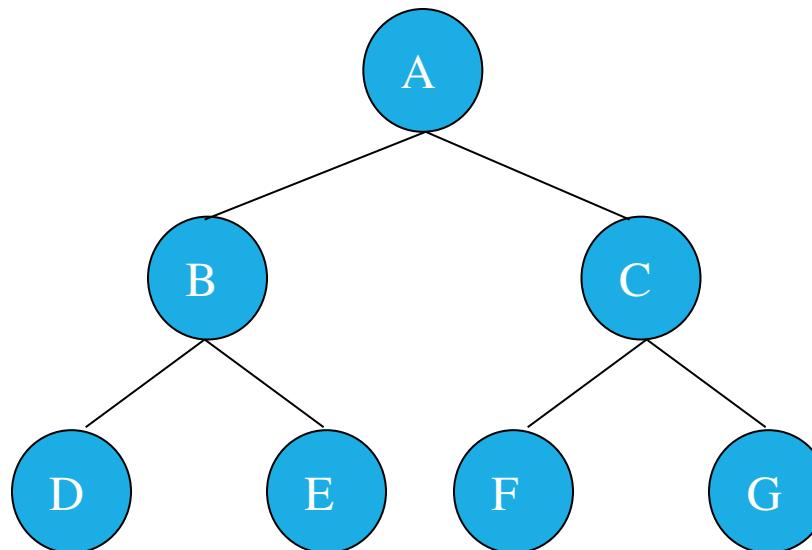
Cây nhị phân

- Cây nhị phân đầy đủ với chiều sâu d
 - Phải là cây nhị phân đúng
 - Tất cả nút lá có chiều sâu d

Số nút = $(2^{d+1} - 1)$

Số nút trung gian = ?

Biết số nút tính d của cây
nhị phân đầy đủ



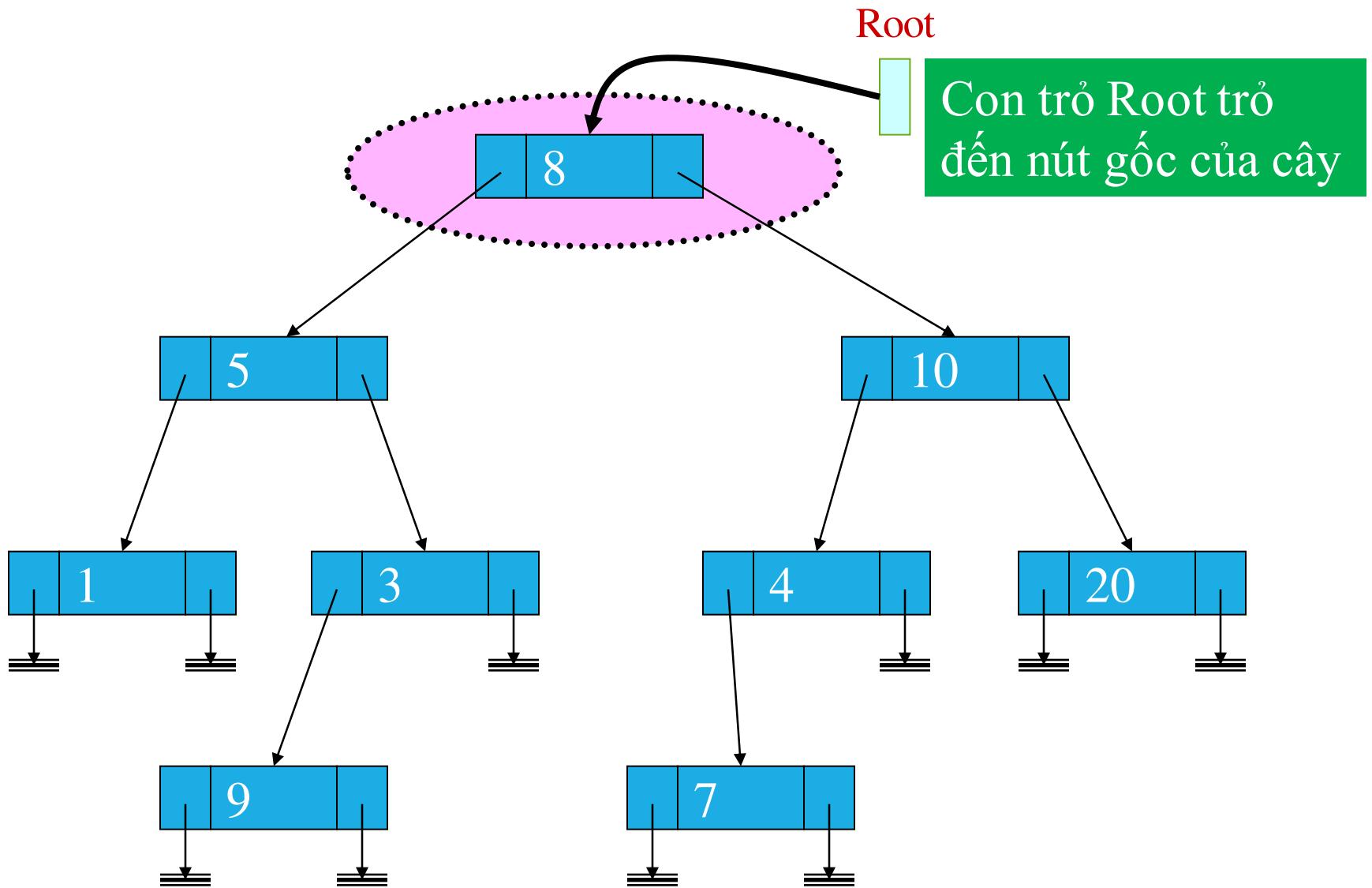
Cấu trúc cây nhị phân

- Cấu trúc của cây nhị phân.

```
typedef int ItemType;  
struct TNode  
{ //Cấu trúc của một nút  
    ItemType Info;  
    TNode* Left;  
    TNode* Right;  
};  
struct BTree  
{ //Cấu trúc của một cây  
    TNode* Root;  
};
```

The diagram illustrates the structure of a binary tree node and its root pointer. It consists of two main parts: a green box representing the `TNode` structure and a blue vertical bar representing the `BTree` structure. Inside the green box, there are three members: `ItemType Info`, `TNode* Left`, and `TNode* Right`. Three red arrows point from the text labels to their corresponding members: the arrow from "Chứa thông tin của nút" points to `Info`; the arrow from "Trỏ đến nút con trái" points to `Left`; and the arrow from "Trỏ đến nút con phải" points to `Right`. Outside the green box, a single red arrow points from the text label "Con trỏ đến nút gốc của cây" to the `Root` pointer in the `BTree` structure.

Sơ đồ cây nhị phân



Các thao tác cơ bản

- Khởi tạo cây
- Tạo nút mới.
- Thêm nút con trái T
- Thêm nút con phải T
- Thêm nút có giá trị x
- Duyệt cây
- Xóa con trái T
- Xóa con phải T
- Xóa nút có giá trị x
- Xóa cây.

Khởi tạo cây nhị phân rỗng

```
void initBTree(BTree &bt)
{
    bt.Root = NULL;
}
```

Tạo nút mới

```
TNode* createTNode(ItemType x)
```

```
{
```

```
    TNode* p = new TNode;
```

```
    if(!p) return NULL;
```

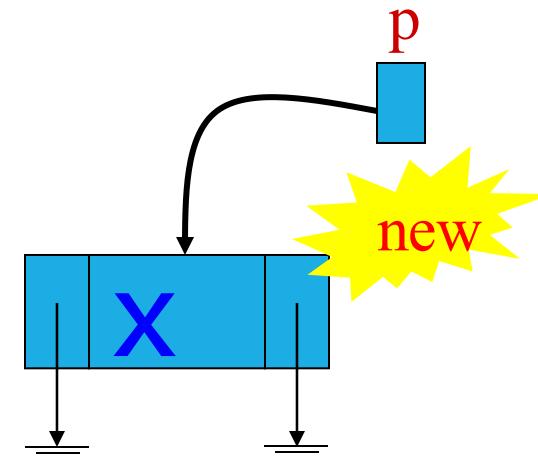
```
    p->Info = x;
```

```
    p->Left = NULL;
```

```
    p->Right = NULL;
```

```
    return p;
```

```
}
```



Thêm nút con bên trái của node T

```
int insertTNodeLeft(TNode* T, ItemType x)
{
    if(T == NULL)
        return 0; //Không tồn tại nút T
    if(T->Left != NULL)
        return 0; //Đã tồn tại nút con trái
    TNode* p = createTNode(x);
    T->Left = p;
    return 1;
}
```

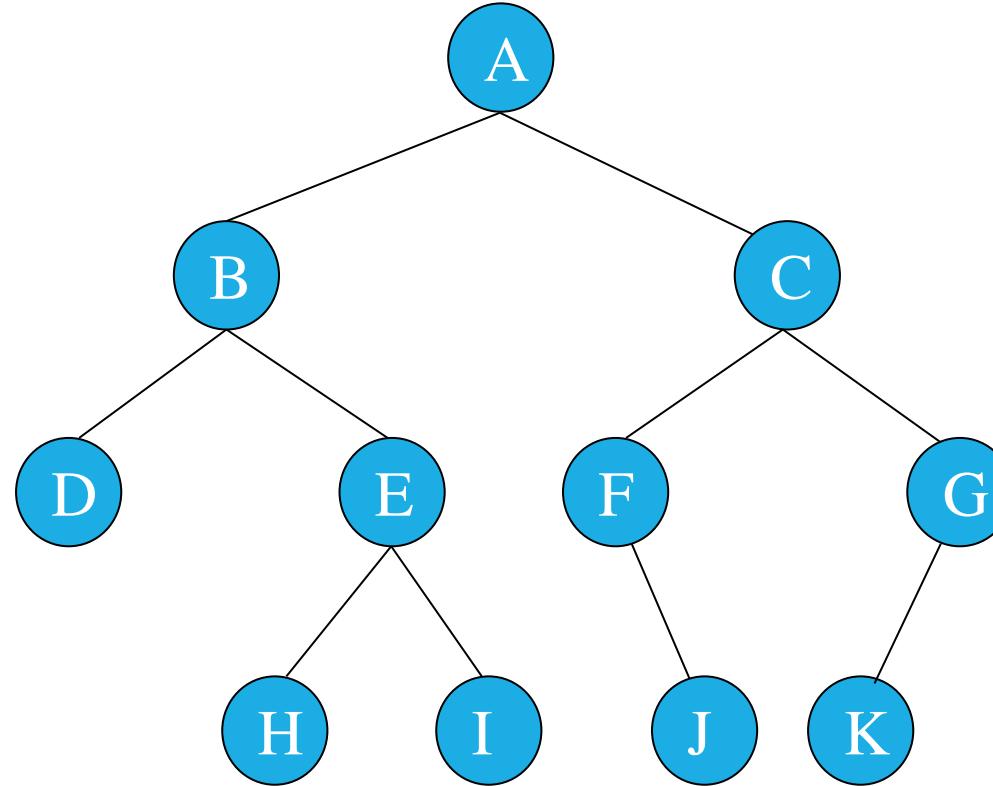
Thêm nút con bên phải của node T

```
int insertTNodeRight(TNode* T, ItemType x)
{
    if(T == NULL)
        return 0; //Không tồn tại nút T
    if(T->Right != NULL)
        return 0; //Đã tồn tại nút con phải
    TNode* p = createTNode(x);
    T->Right = p;
    return 1;
}
```

Duyệt cây

- Do cây là cấu trúc không tuyến tính
- Có 6 cách duyệt cây nhị phân:
 - Duyệt theo thứ tự trước **PreOrder**: **traverseNLR**, **traverseNRL**
 - Duyệt theo thứ tự giữa **InOrder**: **traverseLNR**, **traverseRNL**
 - Duyệt theo thứ tự sau **PostOrder**: **traverseLRN**, **traverseRLN**

Duyệt cây



PreOrder (traverseNLR, traverseNRL)

InOrder (traverseLNR, traverseRNL)

PostOrder (traverseLRN, traverseRLN)

Duyệt Node Left Right

```
void traverseNLR(TNode* root)
{
    if(root == NULL) return;
    <xử lý gốc>;
    traverseNLR(root→Left);
    traverseNLR(root→Right);
}
```

Duyệt Node Left Right

```
void traverseNRL(TNode* root)
{
    if(root == NULL) return;
    <xử lý gốc>;
    traverseNRL(root→Right);
    traverseNRL(root→Left);
}
```

Duyệt Left Node Right

```
void traverseLNR(TNode* root)
{
    if(root == NULL) return;
    traverseLNR(root→Left);
    <xử lý gốc>;
    traverseLNR(root→Right);
}
```

Duyệt Left Node Right

```
void traverseRNL(TNode* root)
{
    if(root == NULL) return;
    traverseRNL(root→Right);
    <xử lý gốc>;
    traverseRNL(root→Left);
}
```

Duyệt Left Right Node

```
void traverseLRN(TNode* root)
{
    if(root == NULL) return;
    traverseLRN(root→Left);
    traverseLRN(root→Right);
    <xử lý gốc>;
}
```

Duyệt Left Right Node

```
void traverseRLN(TNode* root)
{
    if(root == NULL) return;
    traverseRLN(root→Right);
    traverseRLN(root→Left);
    <xử lý gốc>;
}
```

Xóa nút con trái của node T

```
int deleteTNodeLeft(TNode* T)
{//Nút này phải là nút lá
    if(T == NULL) return 0;
    TNode* p = T → Left;
    if(p == NULL) return 0;
    if(p→Left != NULL || p→Right != NULL)
        return 0;
    delete p;
    return 1;
}
```

Xóa nút con phải của node T

```
int deleteTNodeRight(TNode* T)
{//Nút này phải là nút lá
    if(T == NULL) return 0;
    TNode* p = T→Right;
    if(p == NULL) return 0;
    if(p→Left != NULL || p→Right != NULL)
        return 0;
    delete p;
    return 1;
}
```

Tìm nút có khóa x

```
TNode* findTNode(TNode* root, ItemType x)
{
    if(!root) return NULL;
    if(root->Info == x) return root;
    TNode* p = findTNode(root->Left, x);
    if(p) return p;
    return findTNode(root->Right, x);
}
```

Xóa cây

```
int deleteTree(TNode* &root)
{
    if(!root) return 0;
    deleteTree(root→Left);
    deleteTree(root→Right);
    delete root;
    return 1;
}
```

Các thao tác mở rộng

- Đếm số nút trên cây.
- Đếm số nút lá trên cây.
- Đếm số nút trong.
- Xác định độ sâu/chiều cao của cây.
- Đếm số nút có giá trị bằng x.
- Tìm giá trị nhỏ nhất/lớn nhất trên cây.
- Tính tổng các giá trị trên cây.
- Xuất ra màn hình các nút ở mức thứ k.
- Đếm các nút lá ở mức k

Đếm số nút trên cây

```
int countTNode(TNode* root)
{
    if(!root) return 0;
    int cnl = countTNode(root→Left);
    int cnr = countTNode(root→Right);
    return (1 + cnl + cnr);
}
```

Đếm số nút lá

```
int countTNodeLeaf(TNode* root)
{
    if(!root) return 0;
    int cnl = countTNodeLeaf(root→Left);
    int cnr = countTNodeLeaf(root→Right);
    if(!root→Left && !root→Right)
        return (1 + cnl + cnr);
    return (cnl + cnr);
}
```

Đếm số nút không phải nút lá

```
int countTNodeNoLeaf(TNode* root)
{
    if(!root) return 0;
    int cnl=countTNodeNoLeaf(root→Left);
    int cnr=countTNodeNoLeaf(root→Right);
    if(root→Left || root→Right)
        return (1 + cnl + cnr);
    return (cnl + cnr);
}
```

Đếm số nút trong (*nút trung gian*)

```
int countTNodeMedium(TNode* root)
{
    int cn = countTNode(root);
    if(cn <= 2) return 0;
    int cnl = countTNodeLeaf(root);
    return (cn - cnl - 1); //trừ nút gốc
}
```

Đếm số nút trong (*nút trung gian*)

```
int countTNodeMedium(TNode* root)
{
    int n = countTNodeNoLeaf(root);
    if(n > 0)
        return (n - 1); //trừ nút gốc
    return 0;
}
```

Tính tổng

```
int sumTNode(TNode* root)
{
    if(!root) return 0;
    int suml = sumTNode(root→Left);
    int sumr = sumTNode(root→Right);
    return (root→Info + suml + sumr);
}
```

Tính chiều cao của cây

```
int highBTree(TNode* root)
{
    if(!root) return 0;
    int hl = highBTree(root→Left);
    int hr = highBTree(root→Right);
    if(hl > hr)
        return (1 + hl);
    else
        return (1 + hr);
}
```

Đếm số nút có giá trị bằng x

```
int countTNodeX(TNode* root, int x)
{
    if(!root) return 0;
    int nlx = countTNodeX(root→Left, x);
    int nrx = countTNodeX(root→Right, x);
    if(root→Info == x)
        return (1 + nlx + nrx);
    return (nlx + nrx);
}
```

Tìm giá trị lớn nhất trên cây nhị phân

```
int Max(int a, int b) { return (a>b) ? a : b; }
```

```
int maxTNode(TNode* root)
```

```
{
```

```
    if(!root→Left && !root→Right)
```

```
        return (root→Info);
```

```
    int maxl = maxTNode(root→Left);
```

```
    int maxr = maxTNode(root→Right);
```

```
    return Max(root→Info, Max(maxl, maxr));
```

```
}
```

Tìm giá trị lớn nhất trên cây nhị phân

```
int maxTNode(TNode* root) {  
    if(!root->Left && !root->Right)  
        return (root->Info);  
    int maxl = maxTNode(root->Left);  
    int maxr = maxTNode(root->Right);  
    int max = root->Info;  
    if(max < maxl) max = maxl;  
    if(max < maxr) max = maxr;  
    return max;  
}
```

Xuất ra màn hình các nút ở mức k

```
void showTNodeLevelK(TNode* root, int k)
{
    if(!root) return;
    if(k == 0) //đến tầng cần tìm
        printf("%4d", root→Info);
    k--;
    //mức k giảm dần về 0
    showTNodeLevelK(root→Left, k);
    showTNodeLevelK(root→Right, k);
}
```

Đếm các nút lá ở mức k

```
int countTNodeLeafLevelK(TNode* root, int k)
{
    if(!root) return 0;
    if(k==0 && !root→Left && !root→Right)
        return (1);
    k--; //mức k giảm dần về 0
    int nl=countTNodeLeafLevelK(root→Left,k);
    int nr=countTNodeLeafLevelK(root→Right,k);
    return (nl + nr);
}
```

Nội dung

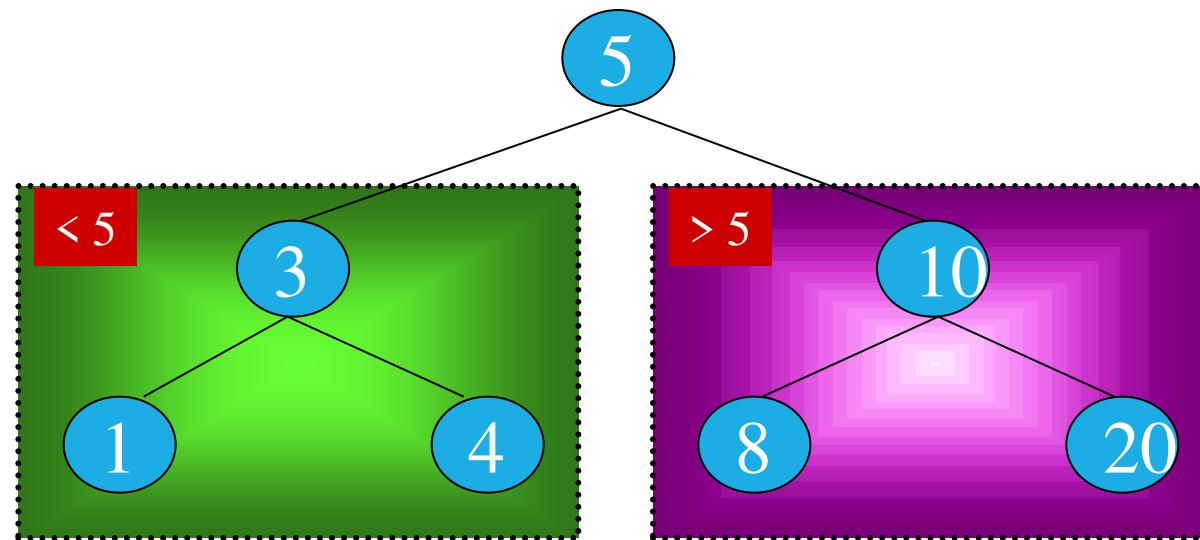
- Cấu trúc cây
- Cây nhị phân
- **Cây nhị phân tìm kiếm**
- Duyệt cây
- Cây AVL
- Cây đỏ đen
- Cây B-Tree

Demo

<https://www.cs.usfca.edu/~galles/visualization/BST.html>

Khái niệm

- BST (Binary Search Tree) là cây nhị phân mà mỗi nút thoả:
 - Giá trị của tất cả nút con trái < nút gốc
 - Giá trị của tất cả nút con phải > nút gốc



CẤU TRÚC CÂY NHỊ PHÂN TÌM KIẾM

- Cấu trúc dữ liệu của cây nhị phân tìm kiếm.

```
typedef int ItemType;
struct TNode
{ //Cấu trúc của một nút
    ItemType Info;
    TNode* Left;
    TNode* Right;
};
struct BSTree
{ //Cấu trúc của một cây
    TNode* Root;
};
```

The diagram illustrates the structure of a binary search tree. It shows two parts: a node structure and a tree structure. The node structure is defined by the `TNode` type, which contains an `ItemType Info` field and two pointers, `Left` and `Right`, both of type `TNode*`. The tree structure is defined by the `BSTree` type, which contains a single pointer `Root` of type `TNode*`. Red arrows point from the labels to their corresponding fields in the code. The `Info` field points to the `ItemType Info` field in the `TNode` struct. The `Left` pointer points to the `Left` pointer in the `TNode` struct. The `Right` pointer points to the `Right` pointer in the `TNode` struct. The `Root` pointer points to the `Root` pointer in the `BSTree` struct.

Chứa thông tin của nút

Trỏ đến nút con trái

Trỏ đến nút con phải

Con trỏ đến nút gốc của cây

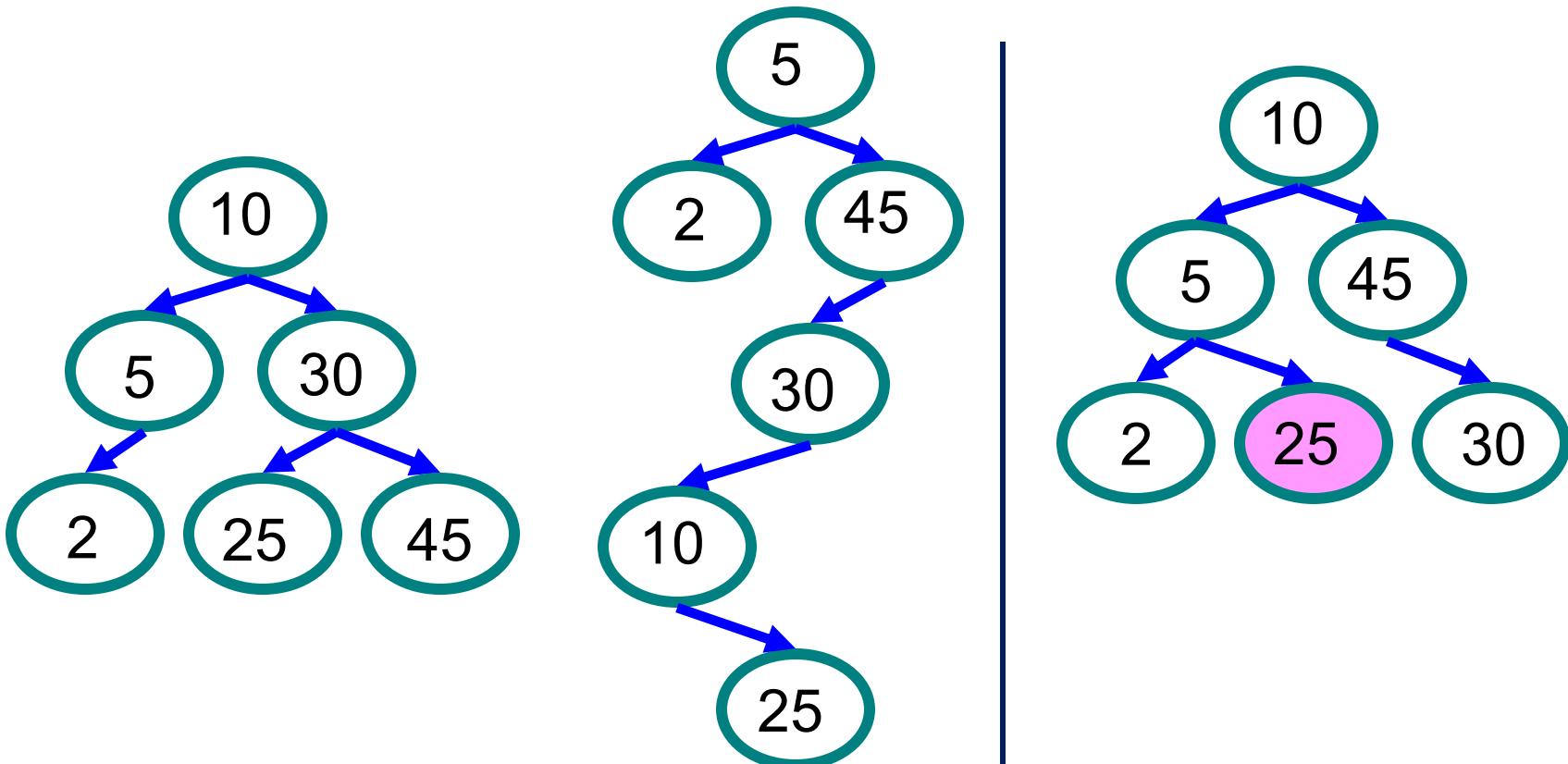
Cây nhị phân tìm kiếm

- Xây dựng cây BST
 - Tìm kiếm/ thêm
 - Xóa
- Luôn duy trì tính chất
 - Giá trị nhỏ hơn ở bên cây con trái
 - Giá trị lớn hơn ở bên cây con phải

Tìm kiếm

- Xuất phát từ gốc
 - Nếu gốc = NULL => không tìm thấy
 - Nếu khóa x = khóa nút gốc => tìm thấy
 - Ngược lại nếu khóa x < khóa nút gốc =>
 Tìm trên cây bên trái
 - Ngược lại => tìm trên cây bên phải

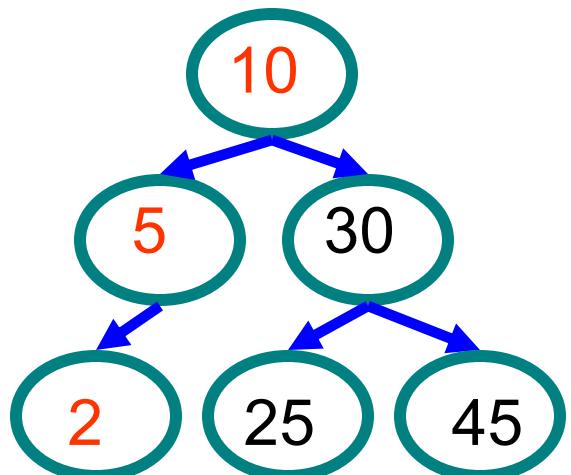
Ví dụ



Binary search
trees

Non-binary
search tree

Ví dụ tìm $x = 2$



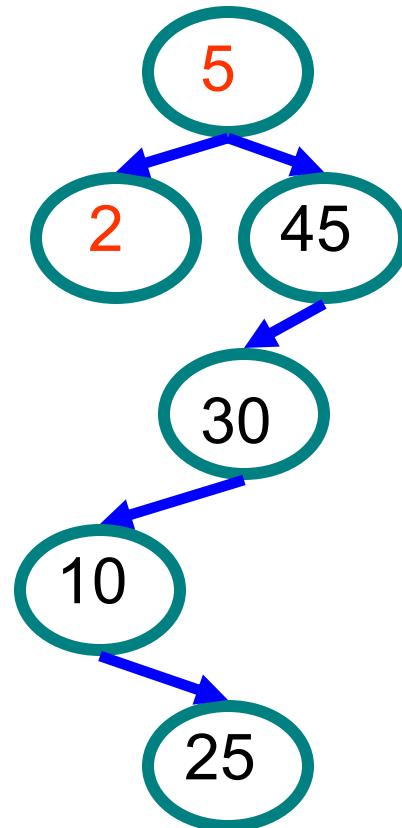
$10 > 2$, Left

$5 > 2$, Left

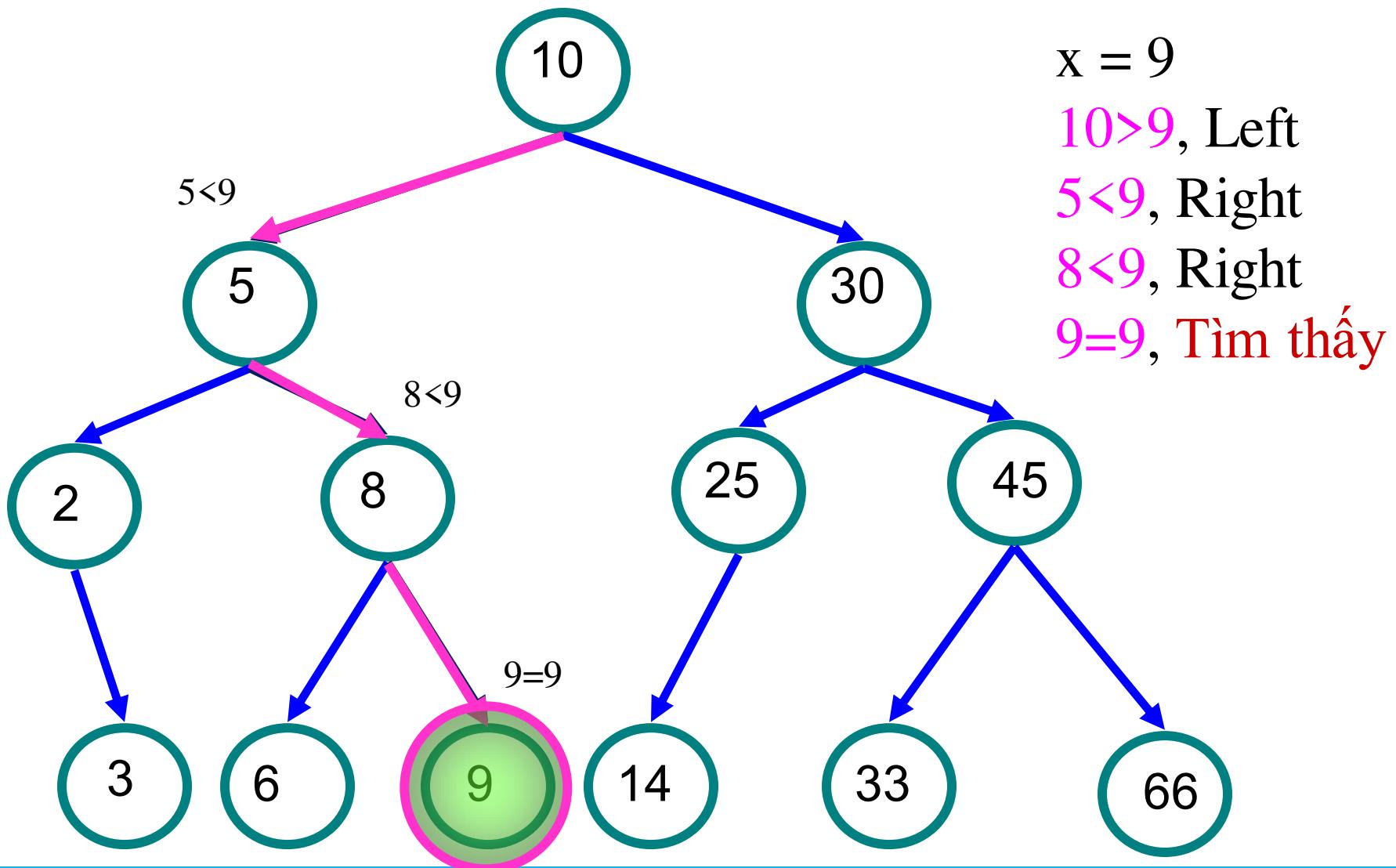
$2 = 2$, Tìm thấy

$5 > 2$, Left

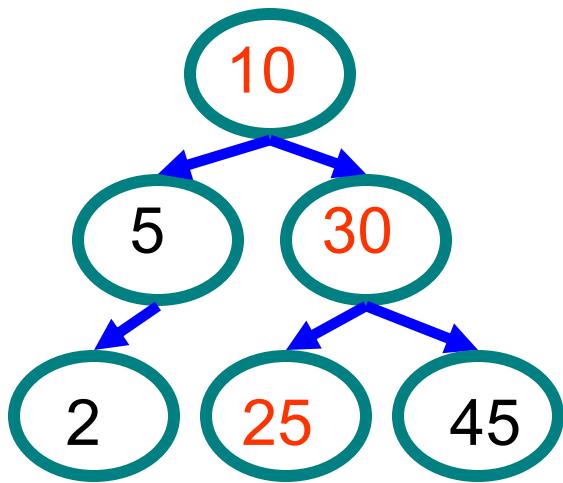
$2 = 2$, Tìm thấy



Ví dụ tìm $x=9$



Ví dụ tìm $x = 25$



$10 < 25$, Right

$30 > 25$, Left

$25 = 25$, Tìm thấy

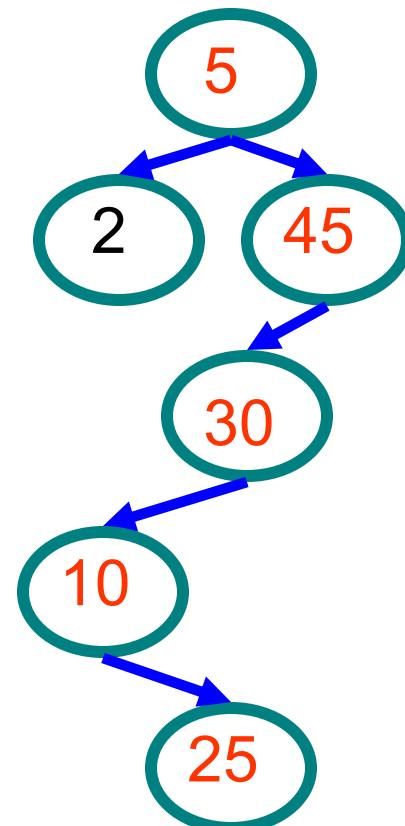
$5 < 25$, Right

$45 > 25$, Left

$30 > 25$, Left

$10 < 25$, Right

$25 = 25$, Tìm thấy



Tìm giá trị x

```
TNode* findTNodeX(TNode* root, ItemType x)
{ // Dùng đệ quy
    if(!root) return NULL;
    if(root→Info == x)
        return root;
    if(root→Info > x)
        return findTNodeX(root→Left, x);
    else
        return findTNodeX(root→Right, x);
}
```

Tìm giá trị x

```
TNode* findTNodeX(TNode* root, ItemType x)
{
    //Không dùng đệ quy
    TNode* p = root;
    while( p && p->Info != x )
    {
        if( p->Info > x )          p = p->Left;
        else                        p = p->Right;
    }
    return p;
}
```

Tìm giá trị lớn nhất trên cây NPTK

TNode* maxTNodeBSTree(TNode* root)

{//Hàm tìm nút có giá trị lớn nhất trên cây (là nút bên phải nhất)

 TNode* p=root;

 while(p→Right != NULL)

 p = p→Right;

 return (p);

}

Duyệt cây

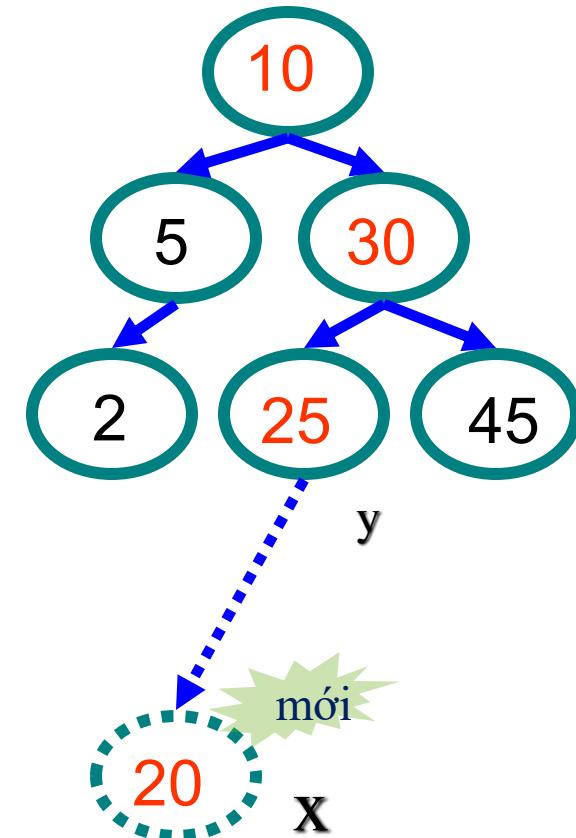
- Lưu ý: Cây nhị phân tìm kiếm duyệt theo traverseLNR thì thứ tự khóa tăng dần.
- Ví dụ: Đếm số nút lớn hơn x

Đếm số phần tử lớn hơn x

```
int countGreaterThanX(TNode* root, ItemType x)
{
    if(!root)  return 0;
    int nlx = countGreaterThanX(root→Left, x);
    int nrx = countGreaterThanX(root→Right, x);
    if( root→Info > x )
        return (1 + nlx + nrx);
    return (nlx + nrx);
}
```

Thêm x vào cây

- Thực hiện tìm kiếm giá trị x
- Tìm đến cuối nút y(nếu x không tồn tại trong cây)
- Nếu $x < y$, thêm nút lá x bên trái của y
- Nếu $x > y$, thêm nút lá x bên phải của y



Thêm x vào cây

```
int insertTNode(TNode* &root, TNode* p)
{//Ham chen 1 nut vao cay NPTK
    if(p == NULL) return 0; //Thêm không thành công
    if(root == NULL) { //Cây đang rỗng
        root = p;
        return 1; //Thêm thành công
    }
    if(root→Info == p→Info)
        return 0; //Bị trùng khóa nên không thêm nữa
    if(p→Info < root→Info)
        insertTNode(root→Left, p); //thêm vào nhánh trái
    else
        insertTNode(root→Right, p); //thêm vào nhánh phải
    return 1; //Thêm thành công
}
```

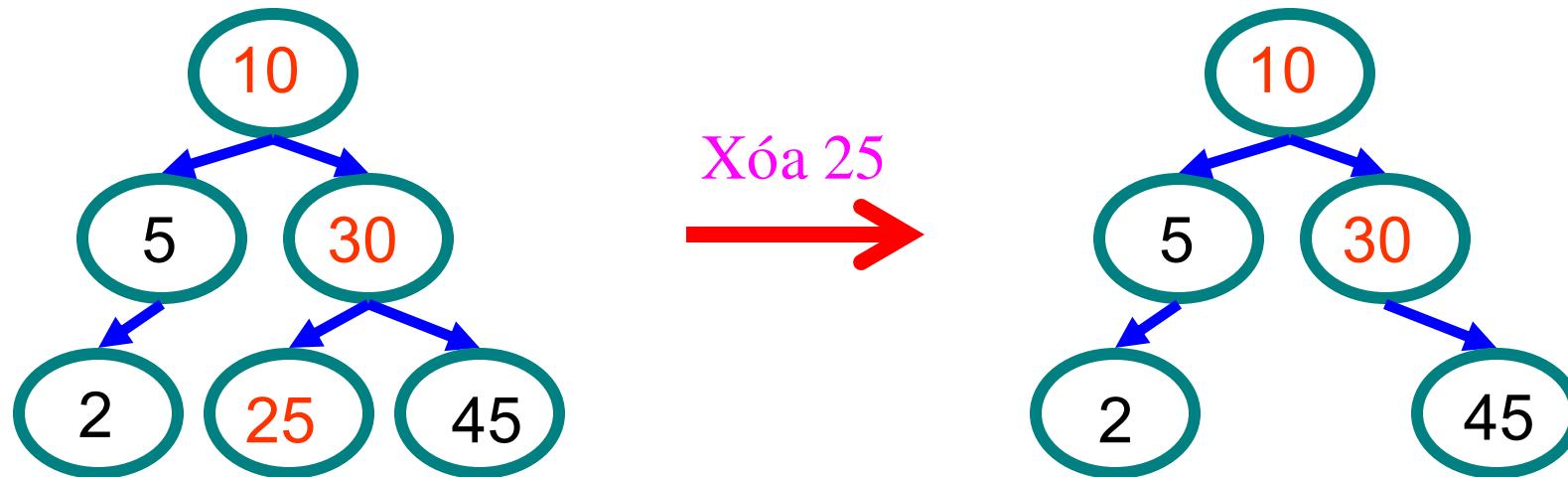
Xóa phần tử khỏi cây

Xóa nhưng phải đảm bảo vẫn là cây BST

- Thực hiện tìm nút có giá trị x.
- Nếu nút là nút lá, xóa nút.
- Ngược lại
 - Thay thế nút bằng một trong hai nút sau.
 - Y là nút lớn nhất của cây con bên trái
 - Z là nút nhỏ nhất của cây con bên phải
 - Chọn nút Y hoặc Z để thế chỗ.
 - Giải phóng nút có giá trị x.

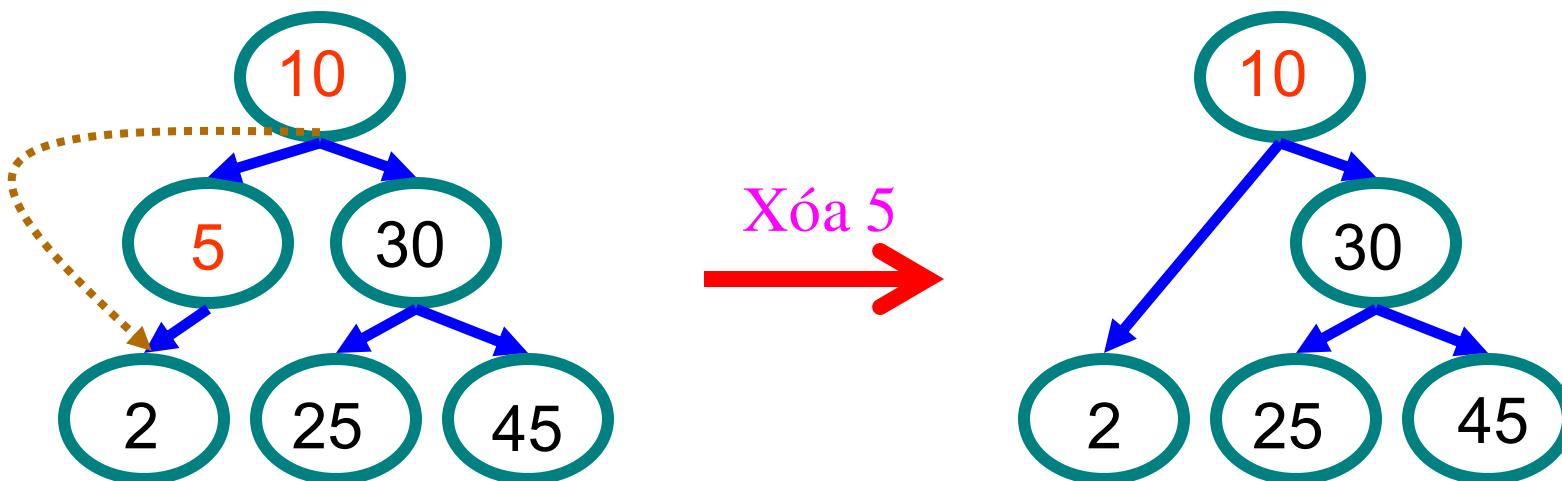
Ví dụ xóa $x = 25$

- Trường hợp 1: nút p là nút lá, xóa bình thường



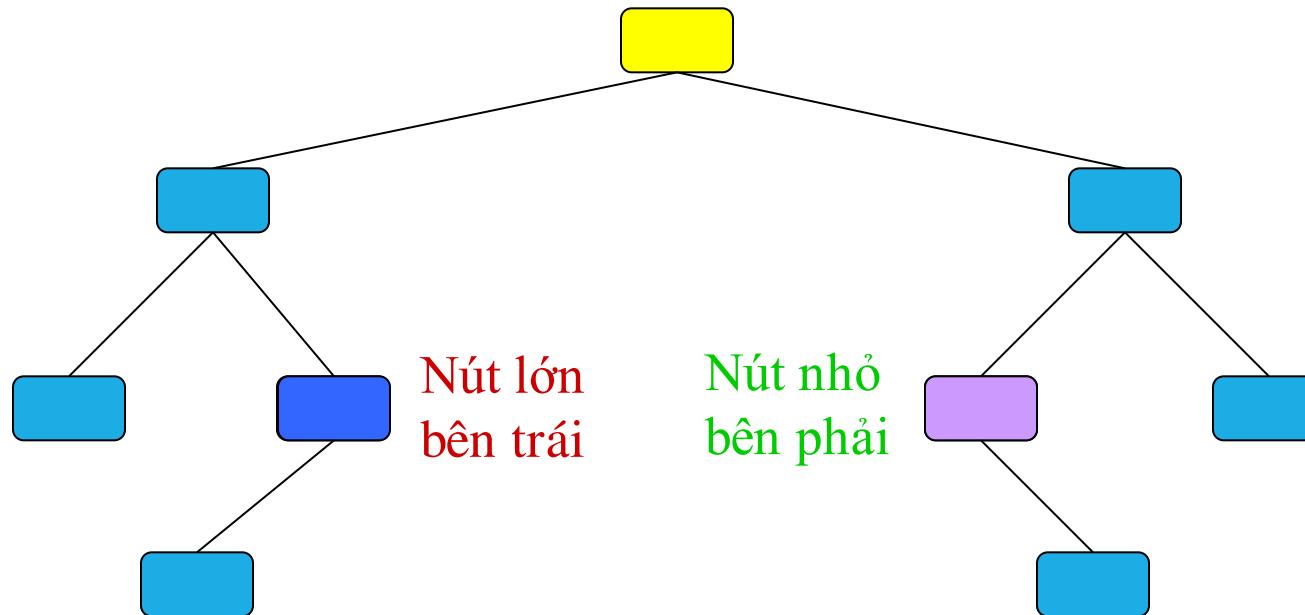
Ví dụ xóa $x = 5$

- Trường hợp 2: p chỉ có 1 cây con, cho nút cha của p trả tới nút con duy nhất của nó, rồi hủy p



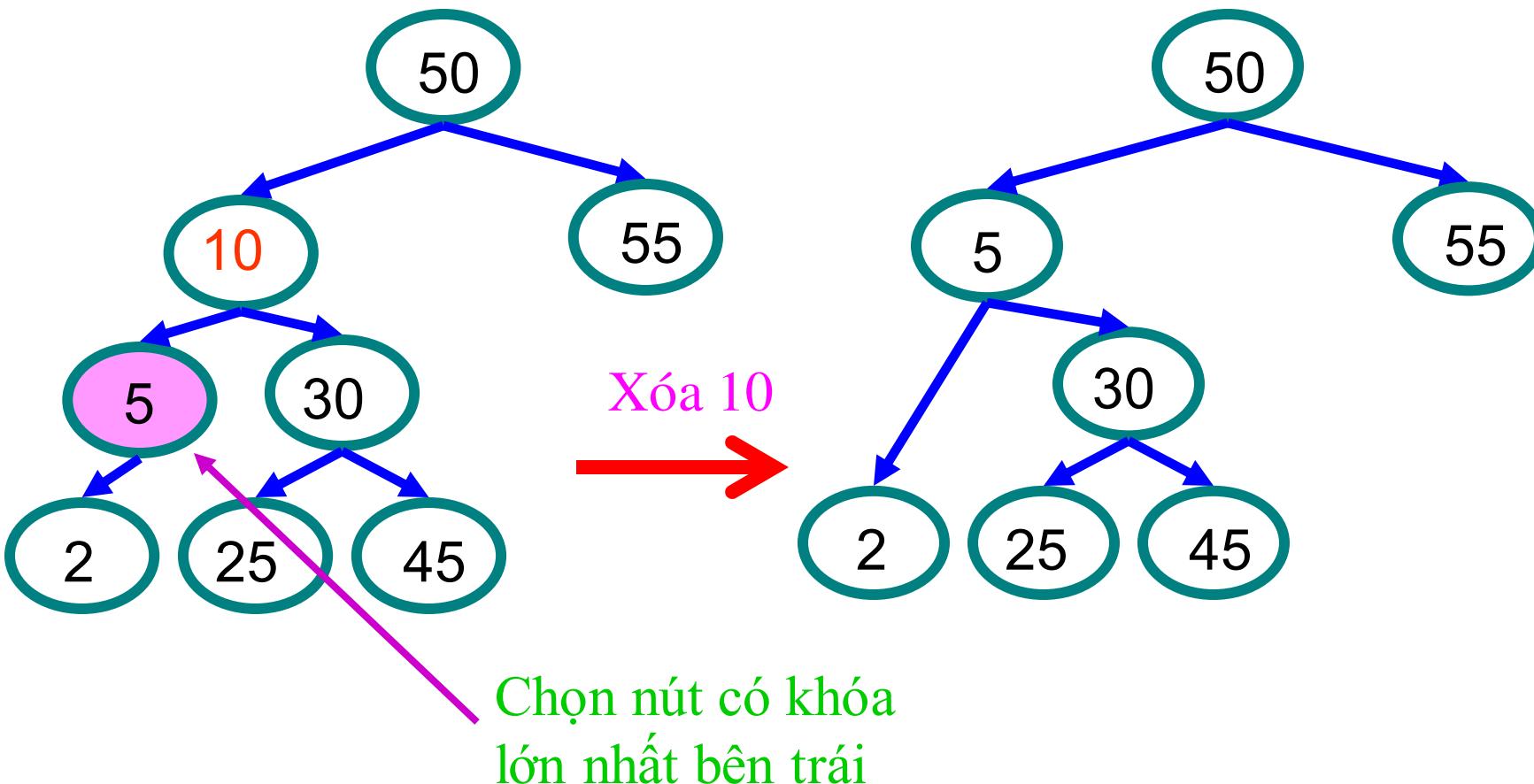
Ví dụ

- **Trường hợp 3:** nút p có 2 cây con, chọn nút thay thế theo 1 trong 2 cách như sau:
 - Nút lớn nhất trong cây con bên trái
 - Nút nhỏ nhất trong cây con bên phải



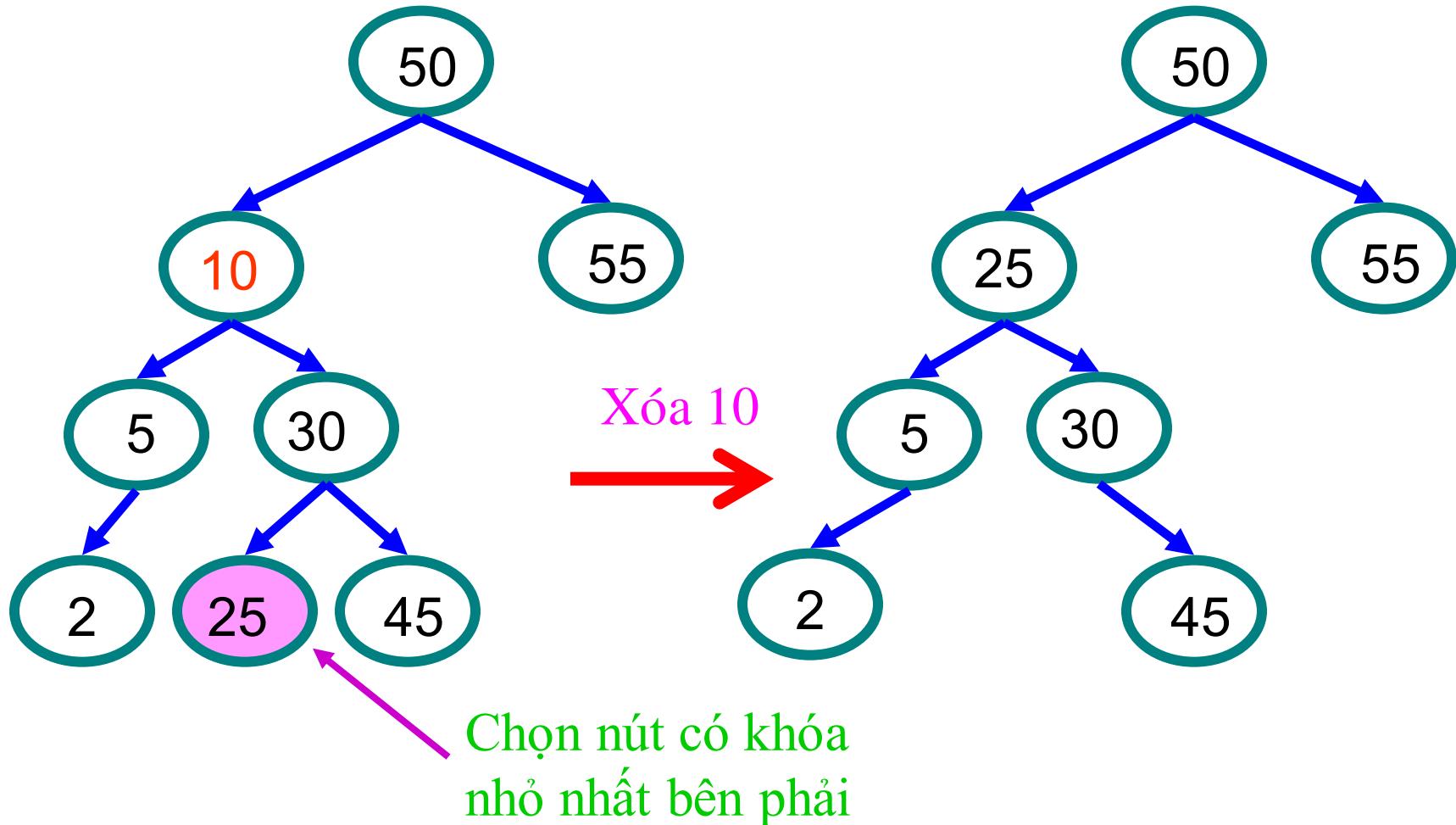
Ví dụ

- Xóa nút 10: cách 1

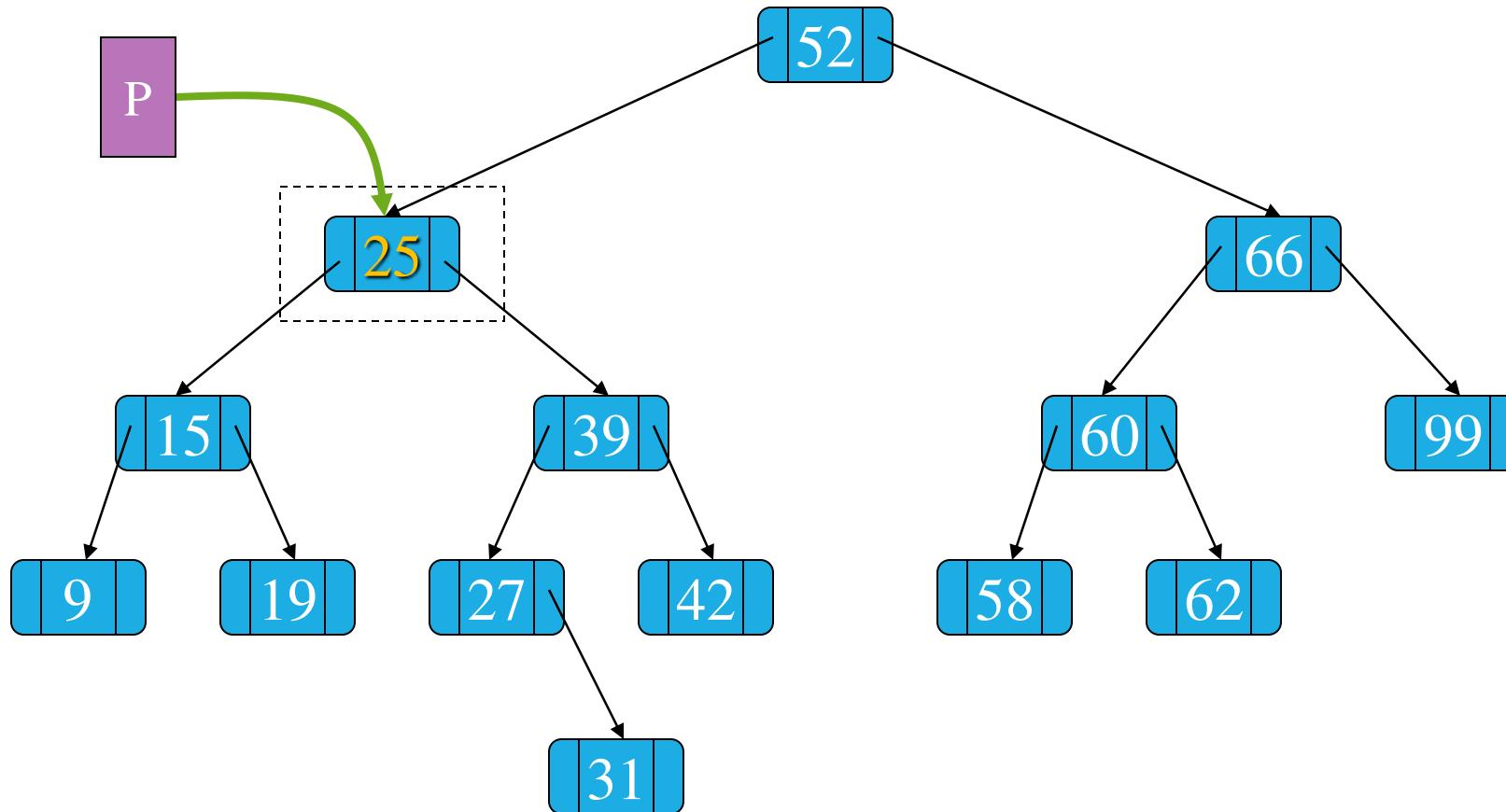


Ví dụ

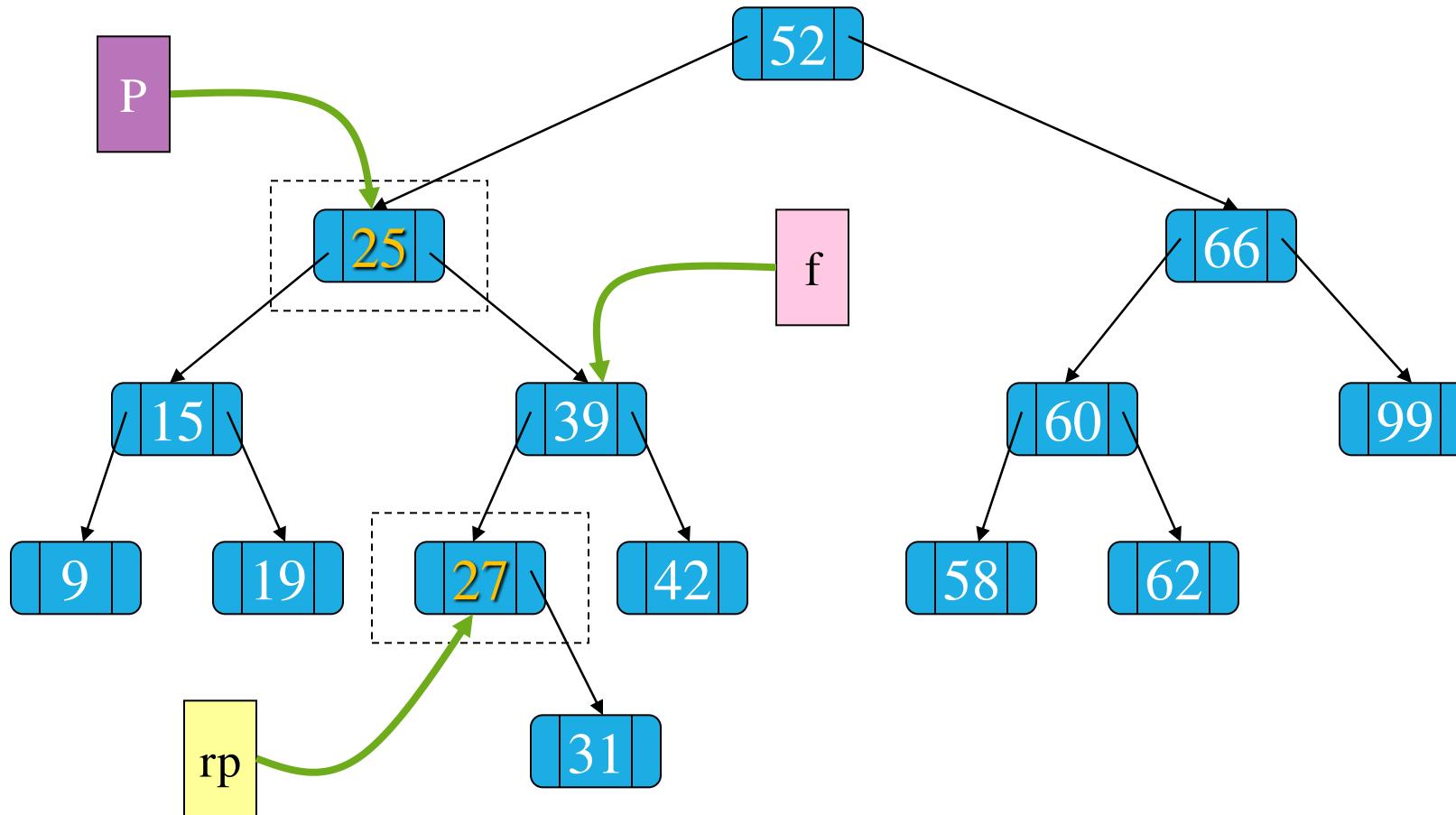
- Xóa nút 10: cách 2



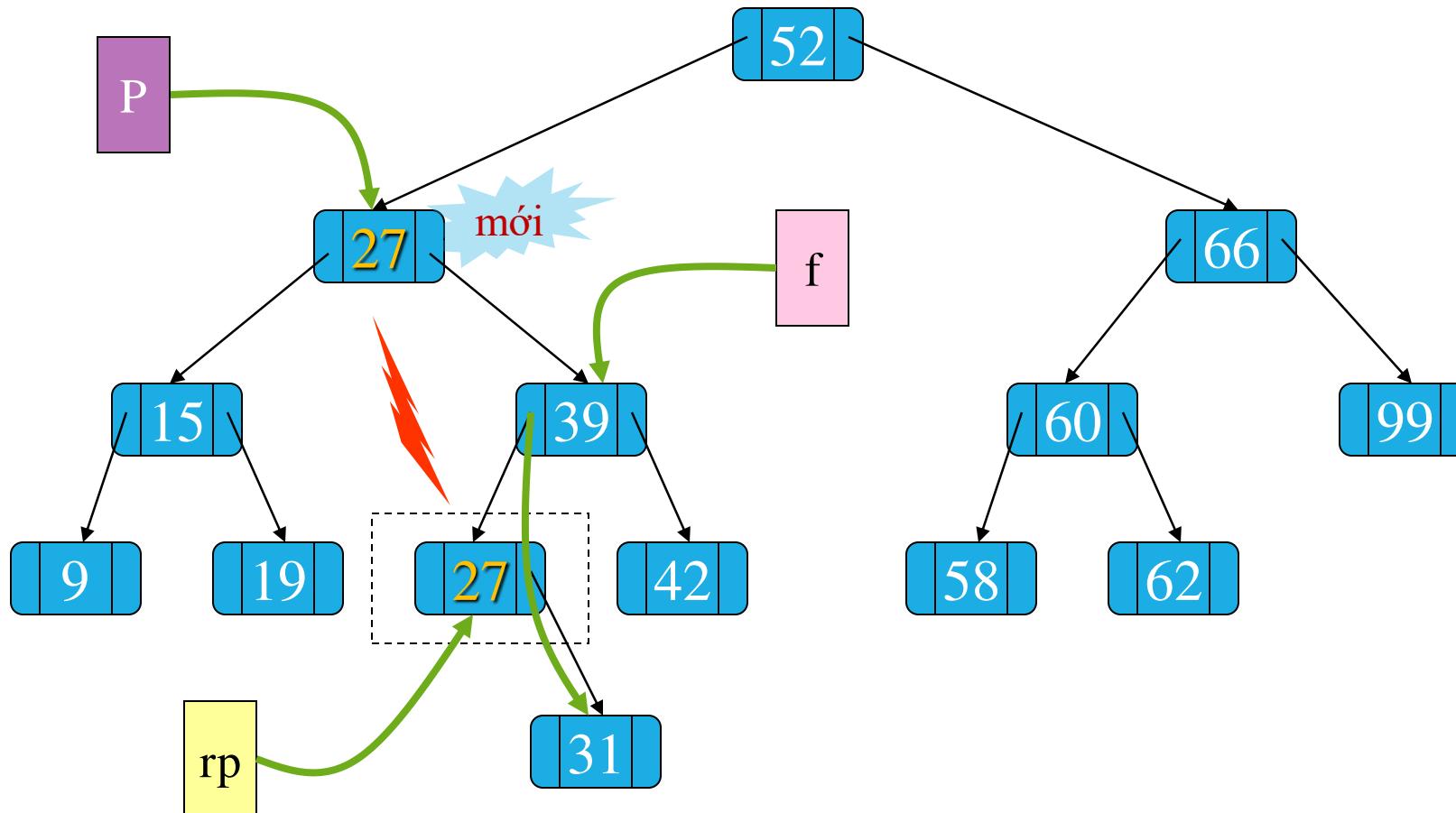
Ví dụ xóa $x = 25$



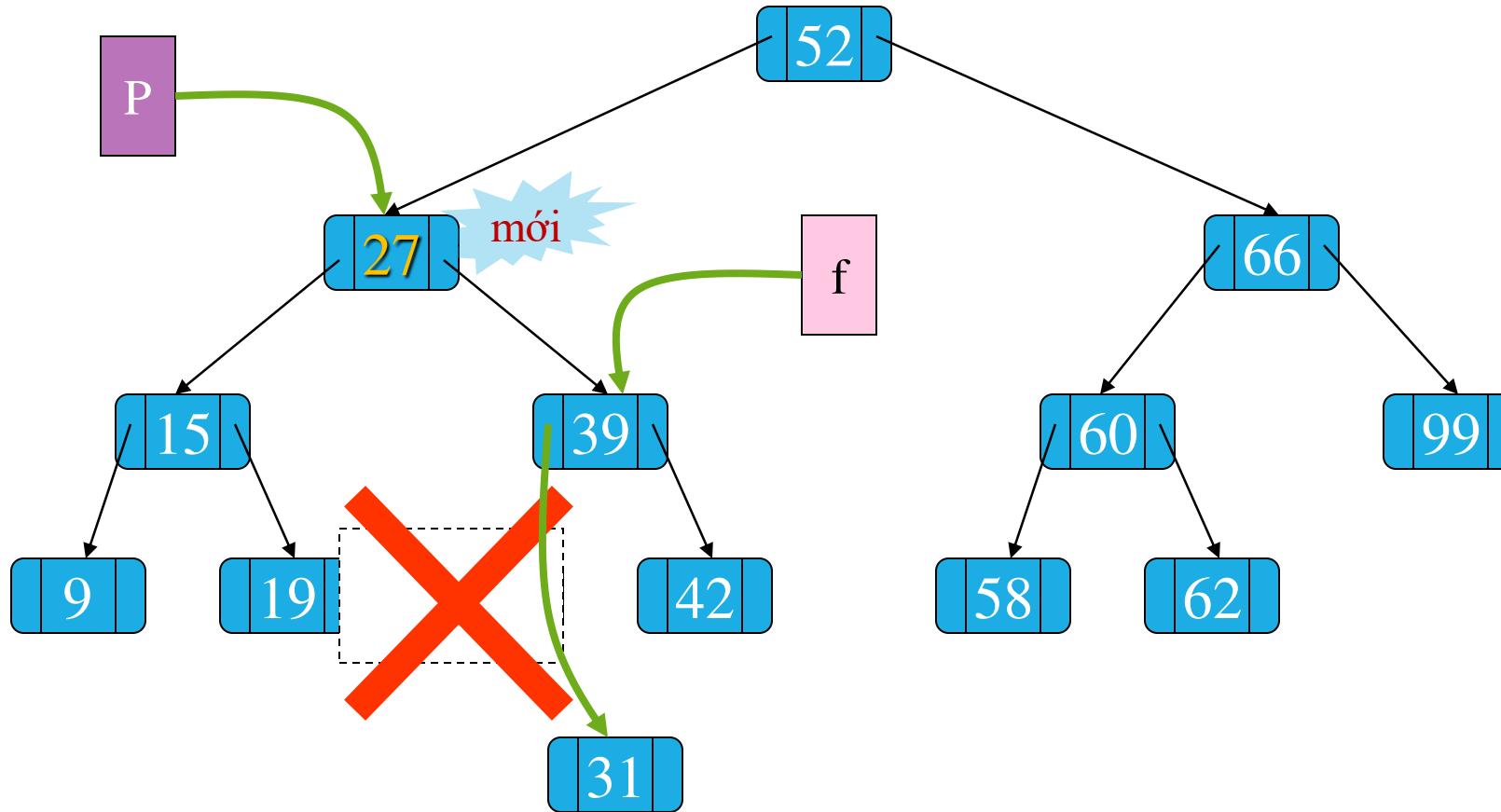
Ví dụ xóa $x = 25$



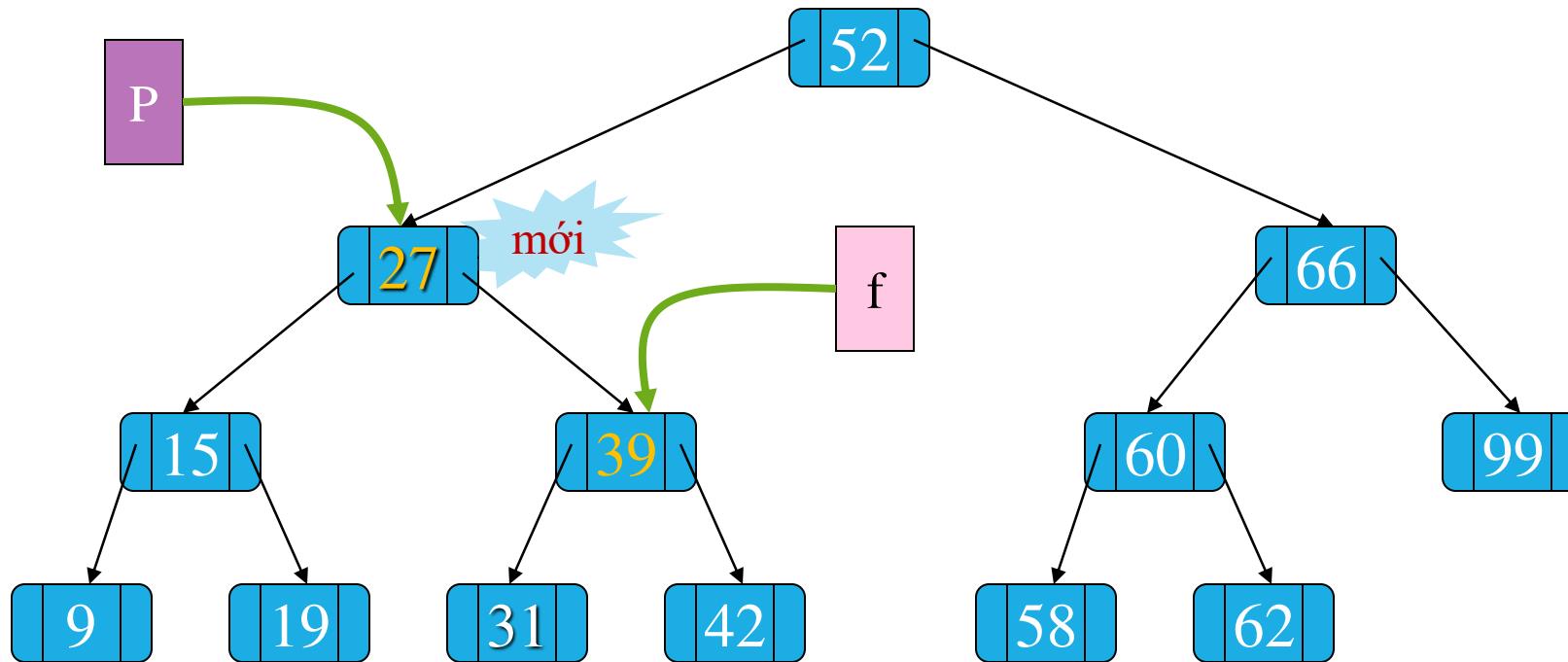
Ví dụ xóa $x = 25$



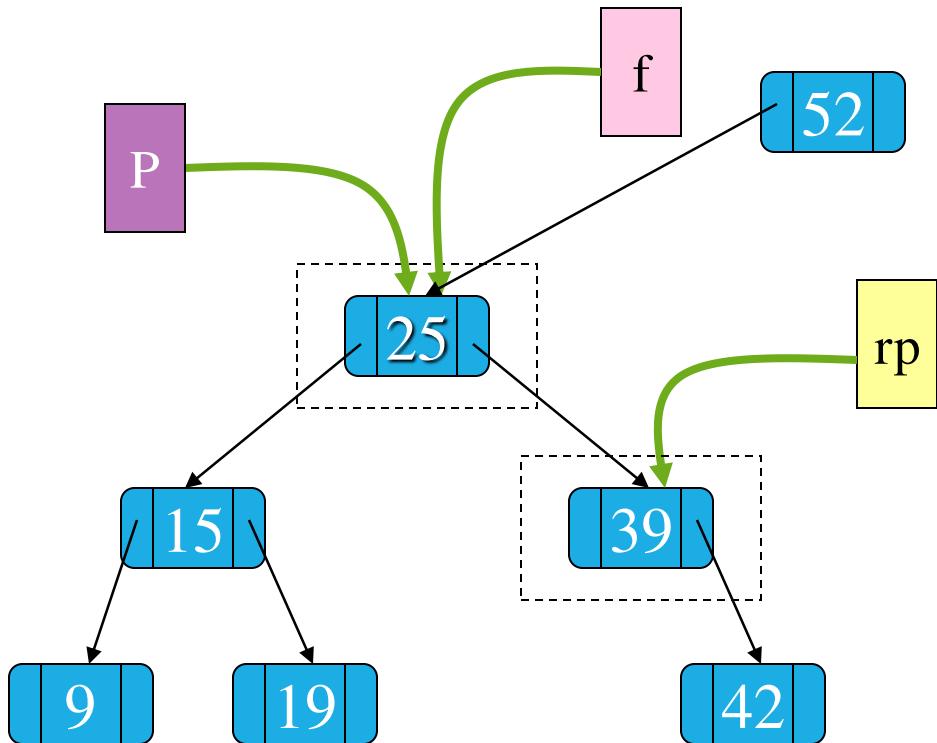
Ví dụ xóa $x = 25$



Ví dụ xóa $x = 25$

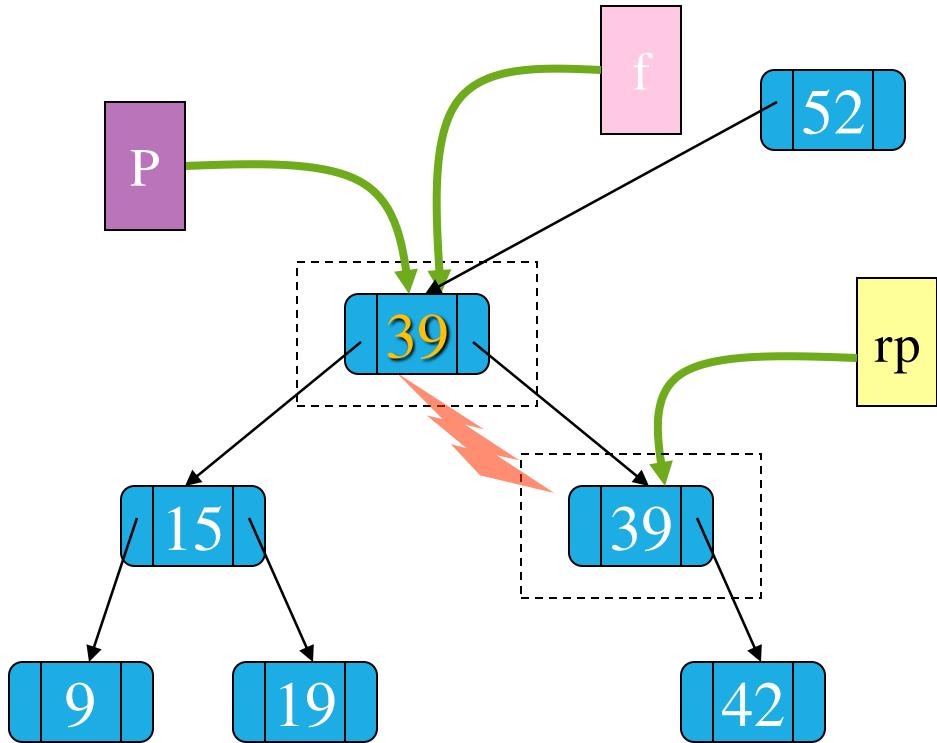


Ví dụ xóa $x = 25$



Trường hợp đặc biệt:
 $f == p$
Nút thế mạng rp là
nút con phải của nút
p cần xóa

Ví dụ xóa $x = 25$

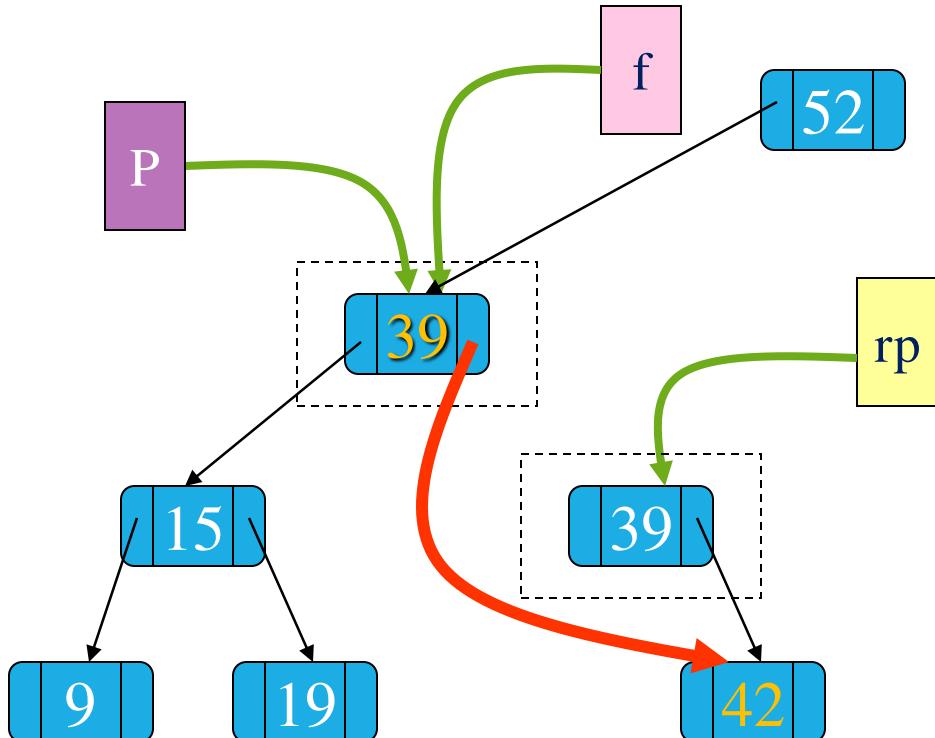


Trường hợp đặc biệt:
 $f == p$

Nút thế mạng rp là
nút con phải của nút
p cần xóa

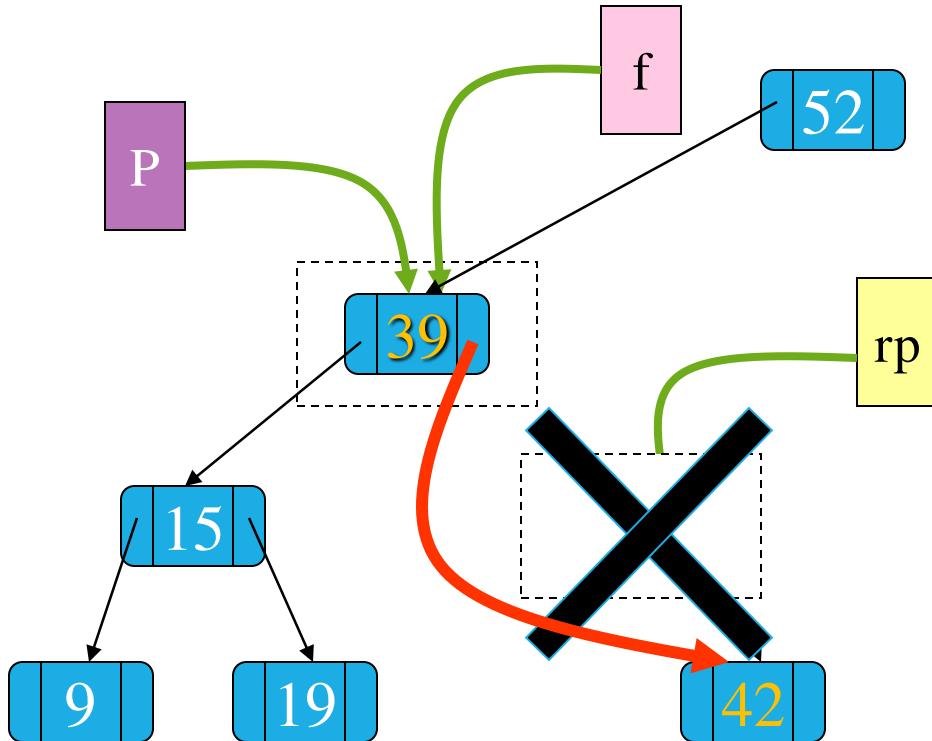
- Đưa giá trị của nút
rp lên nút p

Ví dụ xóa $x = 25$



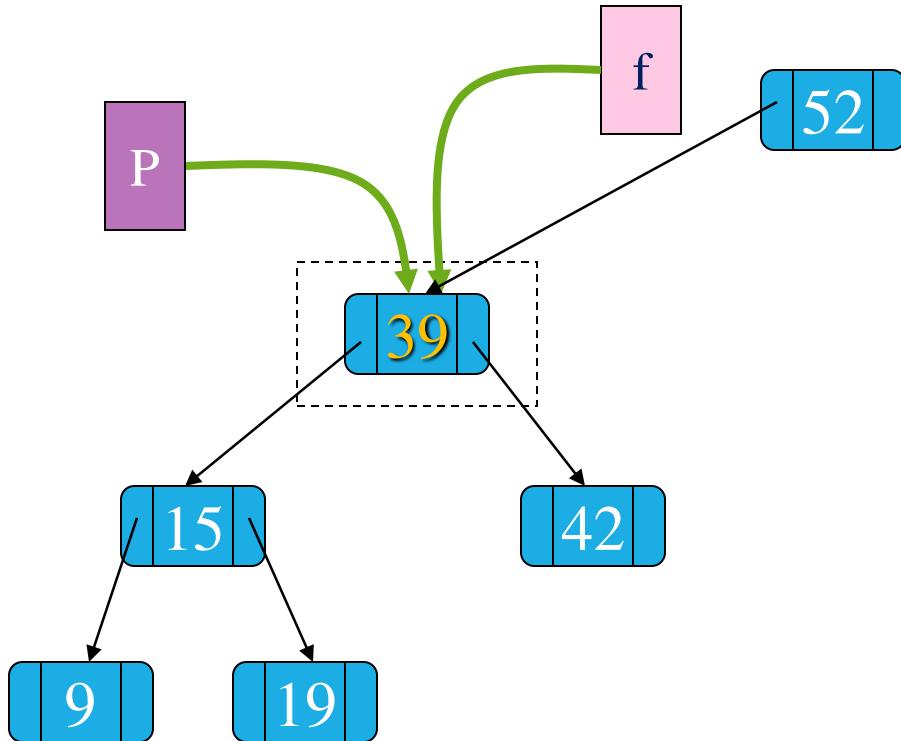
Trường hợp đặc biệt:
 $f == p$
Nút thế mạng rp là
nút con phải của nút
 p cần xóa
- Chuyển liên kết
phải của p đến liên
kết phải của rp

Ví dụ xóa $x = 25$



Trường hợp đặc biệt:
 $f == p$
Nút thế mạng rp là
nút con phải của nút
p cần xóa
- xóa nút rp

Ví dụ xóa $x = 25$



Trường hợp đặc biệt:
 $f == p$

Nút thế mạng rp là
nút con phải của nút
p cần xóa

- Sau khi xóa

Giải thuật

Nếu $T = \text{NULL}$ \Rightarrow thoát

Nếu $T \rightarrow \text{Info} > x \Rightarrow \text{Xoa}(T \rightarrow \text{Left}, x)$

Nếu $T \rightarrow \text{Info} < x \Rightarrow \text{Xoa}(T \rightarrow \text{Right}, x)$

Nếu $T \rightarrow \text{Info} = x$

$p = T$

Nếu T có 1 nút con thì T trở đến nút con đó

Giải thuật

Ngược lại có 2 con

Gọi $f = p$ và $rp = p \rightarrow Right$;

Tìm nút rp : $rp \rightarrow Left = NULL$ và nút f là nút cha nút rp

Thay đổi giá trị nội dung của p và rp

Nếu $f = p$ (trường hợp đặc biệt) thì:

$f \rightarrow Right = rp \rightarrow Right$;

Ngược lại:

$f \rightarrow Left = rp \rightarrow Right$;

Xóa rp ; // xóa nút thế mạng rp

Xóa nút có giá trị là x

```
int deleteTNode(TNode* &root, ItemType x)
{
    if(!root) return 0;
    if(root->Info > x) //tìm bên trái
        return deleteTNode(root->Left, x);
    else if(root->Info < x) //tìm bên phải
        return deleteTNode(root->Right, x);
    else
    {
        TNode* p = root;
```

Xóa nút có giá trị là x

```
if(!root→Left)//khi cay con khong co nhanh trai  
    root = root→Right;  
else if(!root→Right)//khi cay con khong co nhanh phai  
    root = root→Left;  
else  
{//khi cay con co ca 2 nhanh, chon min cua nhanh phai de the mang  
    TNode* p = root;  
    TNode* rp = findTNodeReplace(p);  
    deleteTNode(rp);  
}  
}  
}
```

Tìm nút thế mạng cho nút bị xóa

```
TNode* findTNodeReplace(TNode* &p)
{//Ham tim nut rp nho nhat tren cay con phai de the mang cho nut p
    TNode* f = p,* rp = p→Right;
    while(rp→Left != NULL) {
        f = rp; //Luu cha cua rp
        rp = rp→Left; //rp qua ben trai
    }
    p→Info = rp→Info; //tim duoc phan tu the mang cho p la rp
    if(f == p) //neu cha cua rp la p
        f→Right = rp→Right;
    else
        f→Left = rp→Right;
    return rp; //Tra ve nut rp la nut the mang cho p
}
```

Bài tập

- Tìm phần tử max âm trên cây.
- Đếm có bao nhiêu phần tử chẵn trên cây.

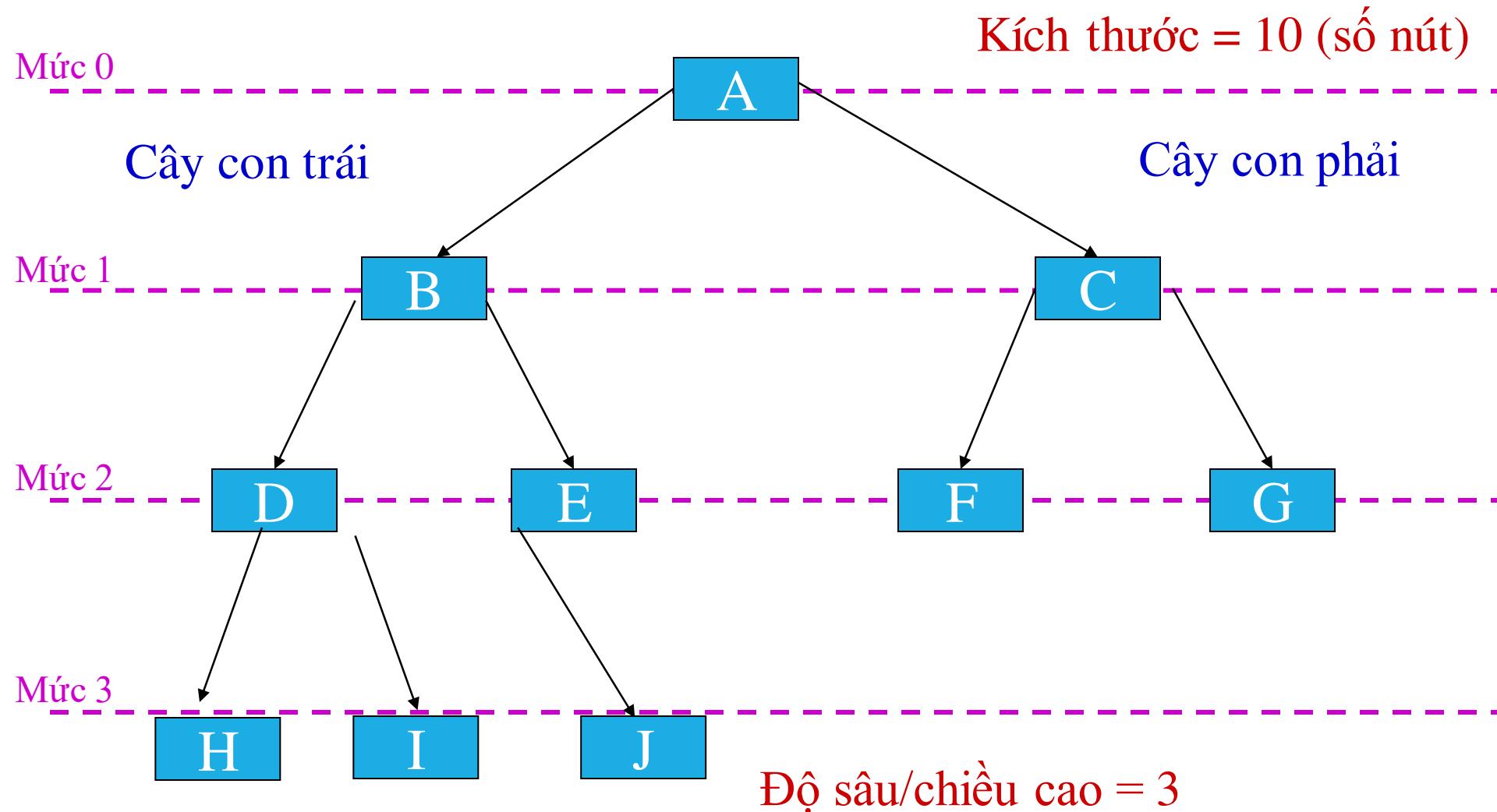
Nội dung

- Cấu trúc cây
- Cây nhị phân
- Cây nhị phân tìm kiếm
- **Duyệt cây**
- Cây AVL
- Cây đỏ đen
- Cây B-Tree

Duyệt cây theo chiều rộng

- ❖ Duyệt cây theo chiều rộng là cách duyệt cây theo một mức hay độ sâu nhất định trước khi duyệt tới mức tiếp theo sâu hơn.

Cây nhị phân



Duyệt cây theo chiều rộng

❖ Các bước thực hiện duyệt cây nhị phân:

- Đầu tiên ta duyệt node gốc (mức 0) và in ra dữ liệu là A
- Tiếp theo ta duyệt tới mức 1 và duyệt theo thứ tự từ trái sang phải, dữ liệu in ra là B, C
- Tiếp tục duyệt mức 2 và in ra dữ liệu theo chiều từ trái sang phải D, E, F, G
- Khi kết thúc mức 2 ta duyệt tới mức cuối cùng và dữ liệu in ra là H, I, J

Duyệt cây theo chiều rộng

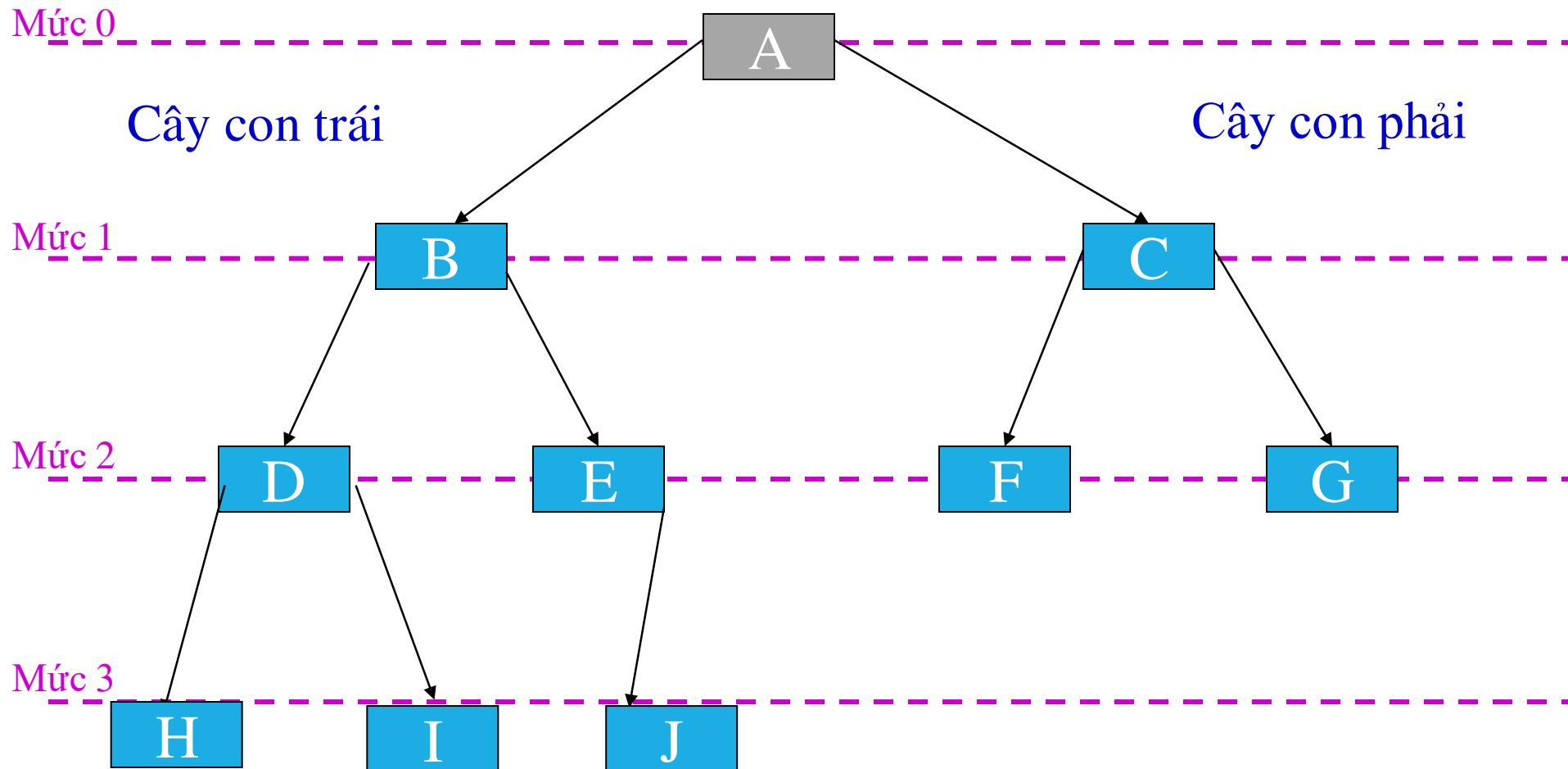
❖ Qui ước:

- Màu đỏ đã duyệt -----
- Màu xám đang duyệt
- Màu xanh chưa duyệt tới

Duyệt cây theo chiều rộng

- ❖ Bước 1: Duyệt node gốc, đánh dấu node đó là đang được duyệt (node khám phá).
- ❖ Enqueue node gốc (A) vào hàng đợi bf_queue

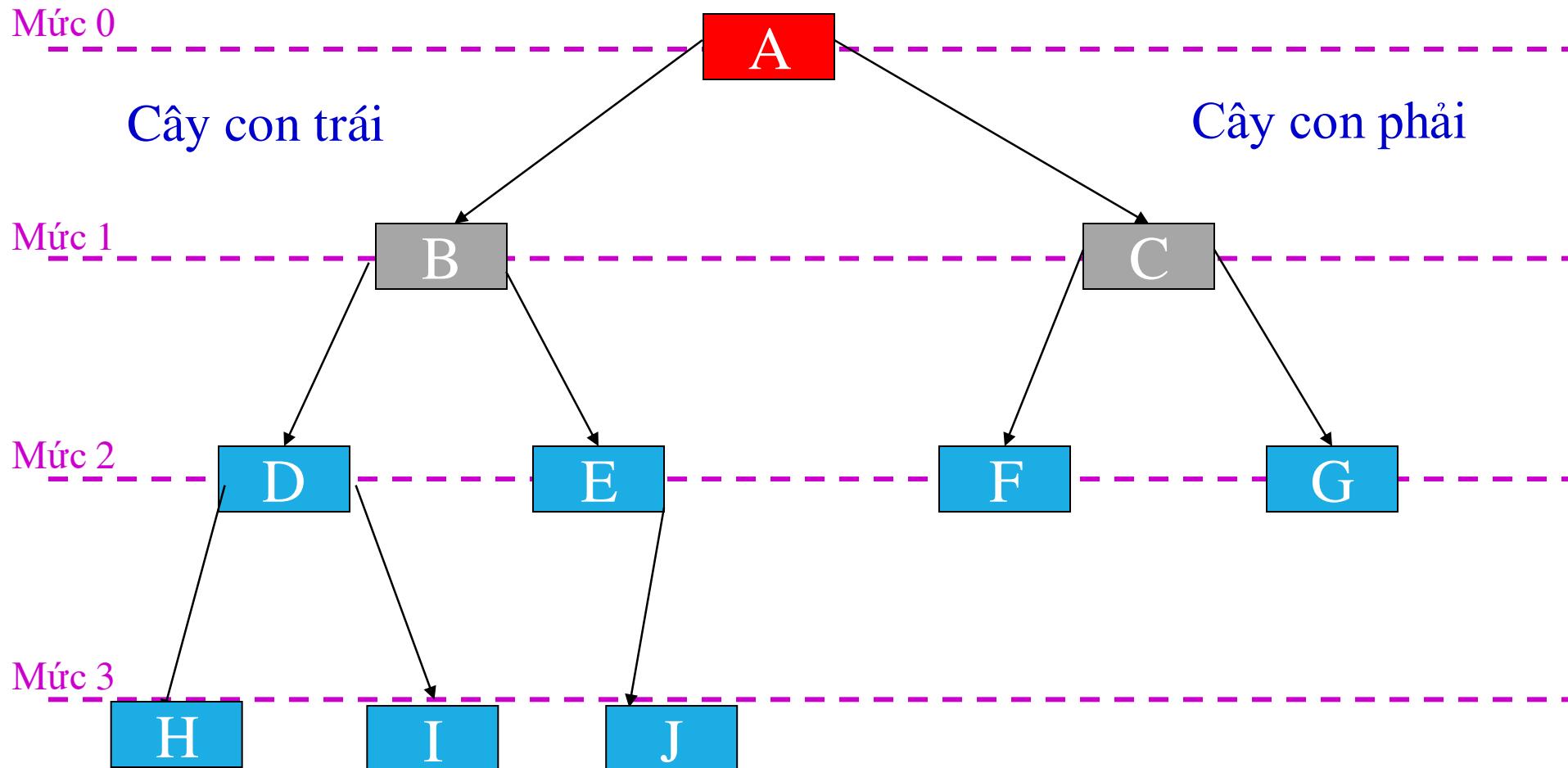
Duyệt cây theo chiều rộng



Duyệt cây theo chiều rộng

- ❖ Bước 2: Kiểm tra queue, queue không rỗng, lấy (dequeue) phần tử đầu tiên (A) ra khỏi queue.
- ❖ Phần tử này chuyển về trạng thái đã duyệt. Enqueue các con (B, C) của phần tử vừa dequeue (A) vào bf_queue. In ra dữ liệu của node A

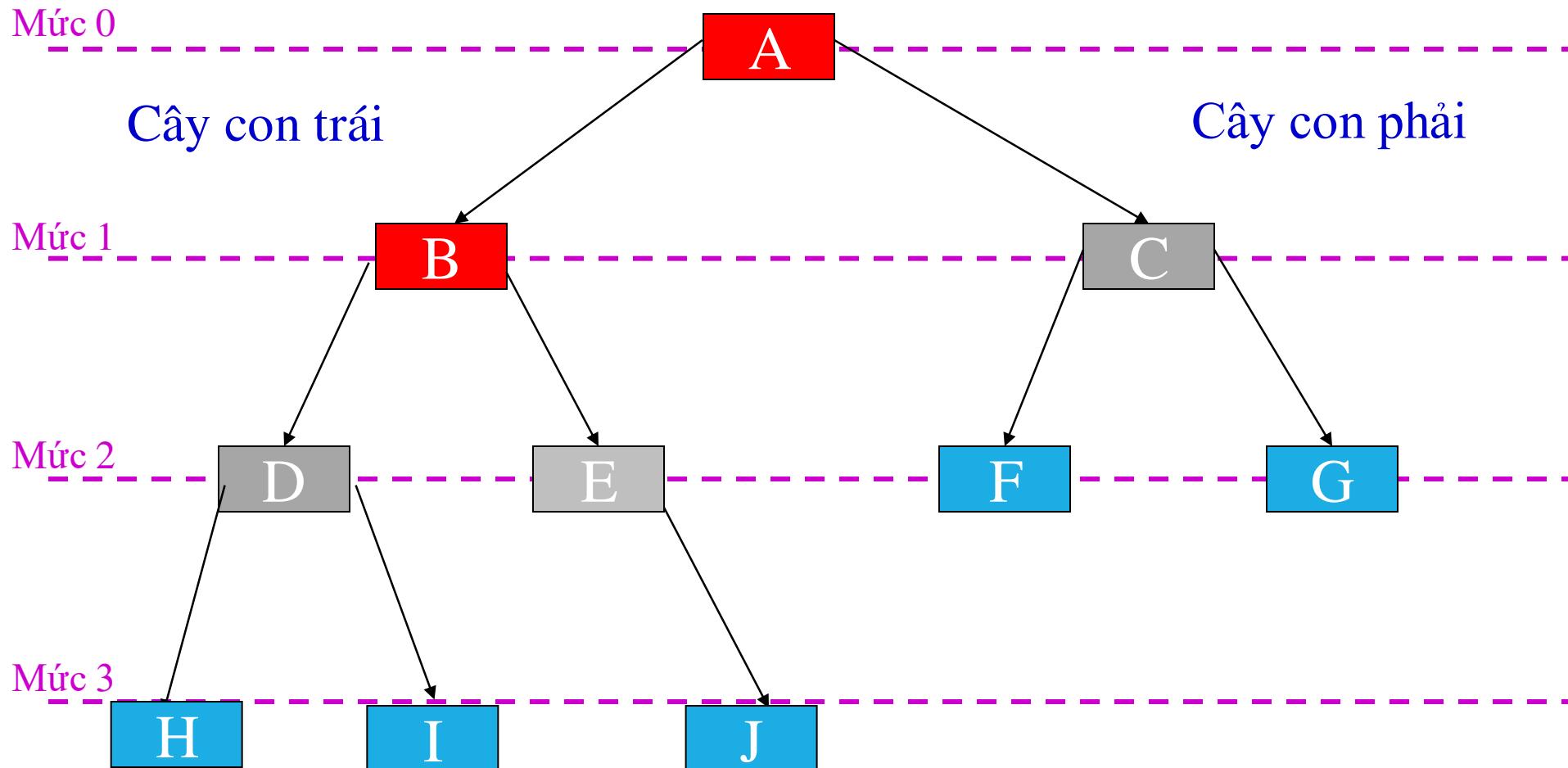
Duyệt cây theo chiều rộng



Duyệt cây theo chiều rộng

- ❖ Bước 3: Kiểm tra queue, queue không rỗng, lấy phần tử đầu tiên (B) ra khỏi queue.
- ❖ Phần tử này chuyển sang trạng thái đã duyệt. Enqueue các con (D, E) của phần tử vừa được dequeue (B) vào bf_queue.
- ❖ In ra dữ liệu node B.

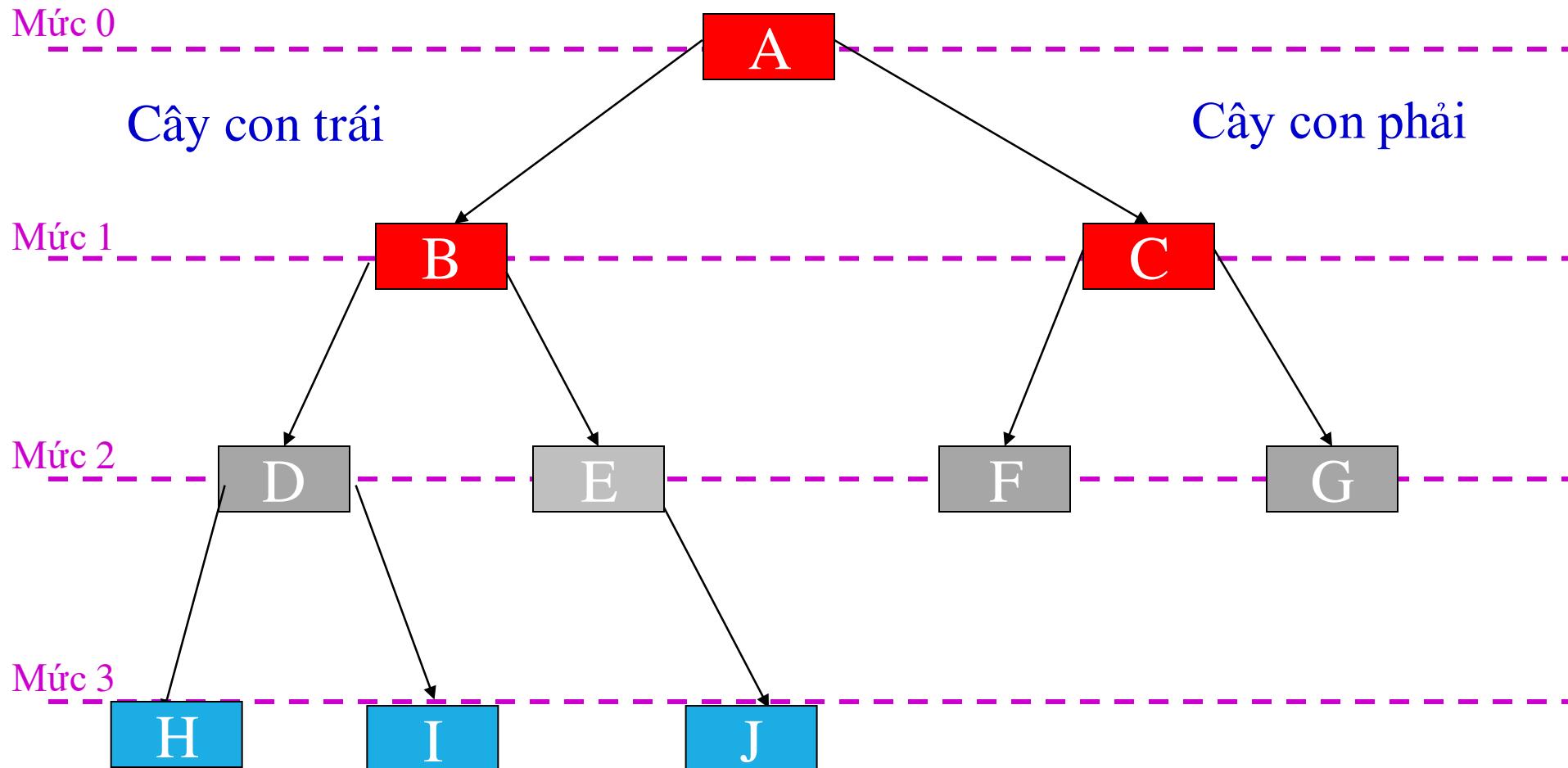
Duyệt cây theo chiều rộng



Duyệt cây theo chiều rộng

- ❖ Bước 4: Kiểm tra queue, queue không rỗng, lấy phần tử đầu tiên (C) ra khỏi queue. Phần tử này chuyển sang trạng thái đã duyệt.
- ❖ Enqueue các con (F, G) của phần tử vừa được dequeue (C) vào bf_queue.
- ❖ In ra dữ liệu node C

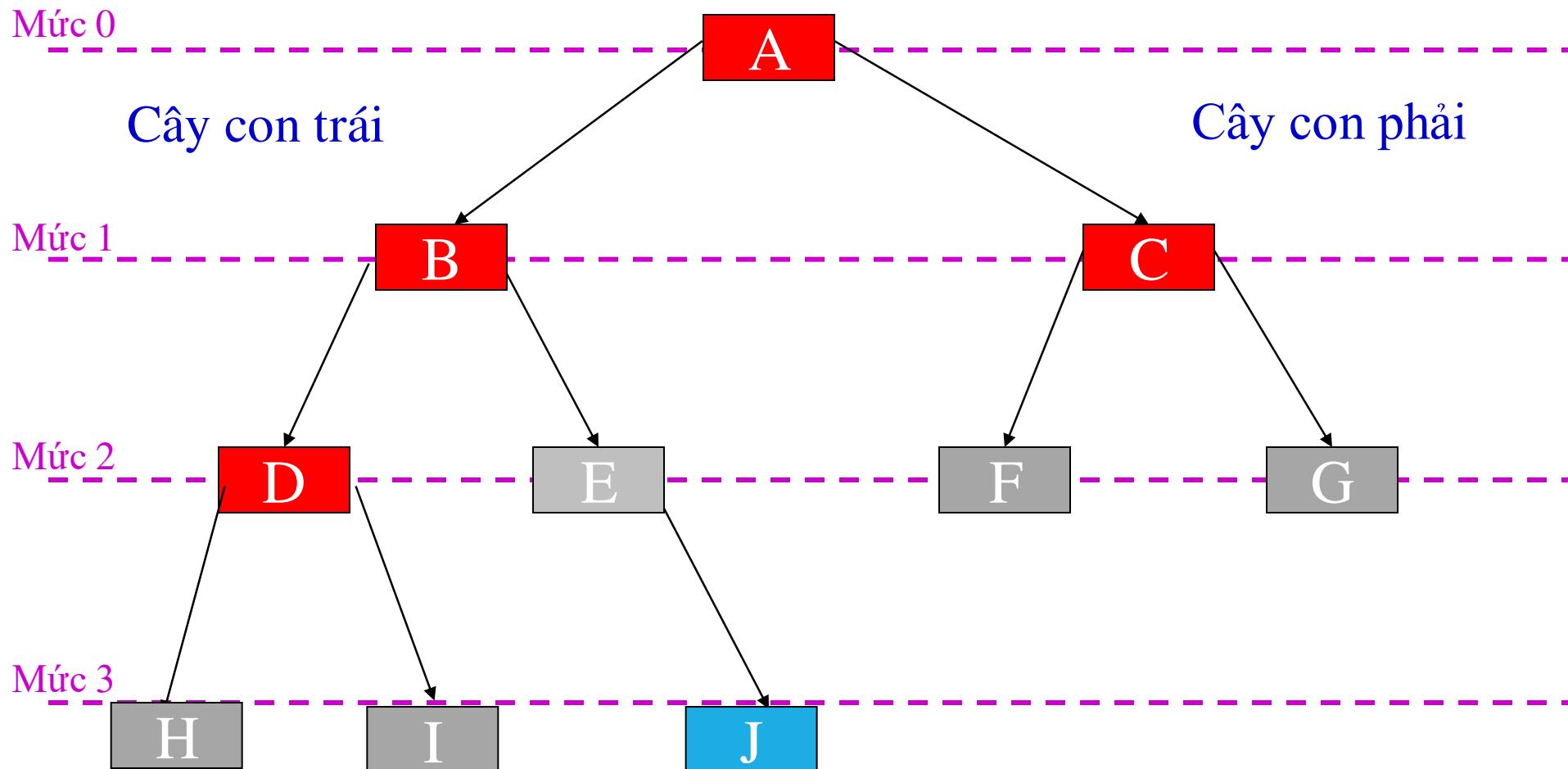
Duyệt cây theo chiều rộng



Duyệt cây theo chiều rộng

- ❖ Bước 5: Kiểm tra queue, queue không rỗng, lấy phần tử đầu tiên (D) ra khỏi queue. Phần tử này chuyển sang trạng thái đã duyệt.
- ❖ Enqueue các con (H, I) của phần tử vừa được dequeue (D) vào bf_queue.
- ❖ In ra dữ liệu node D.

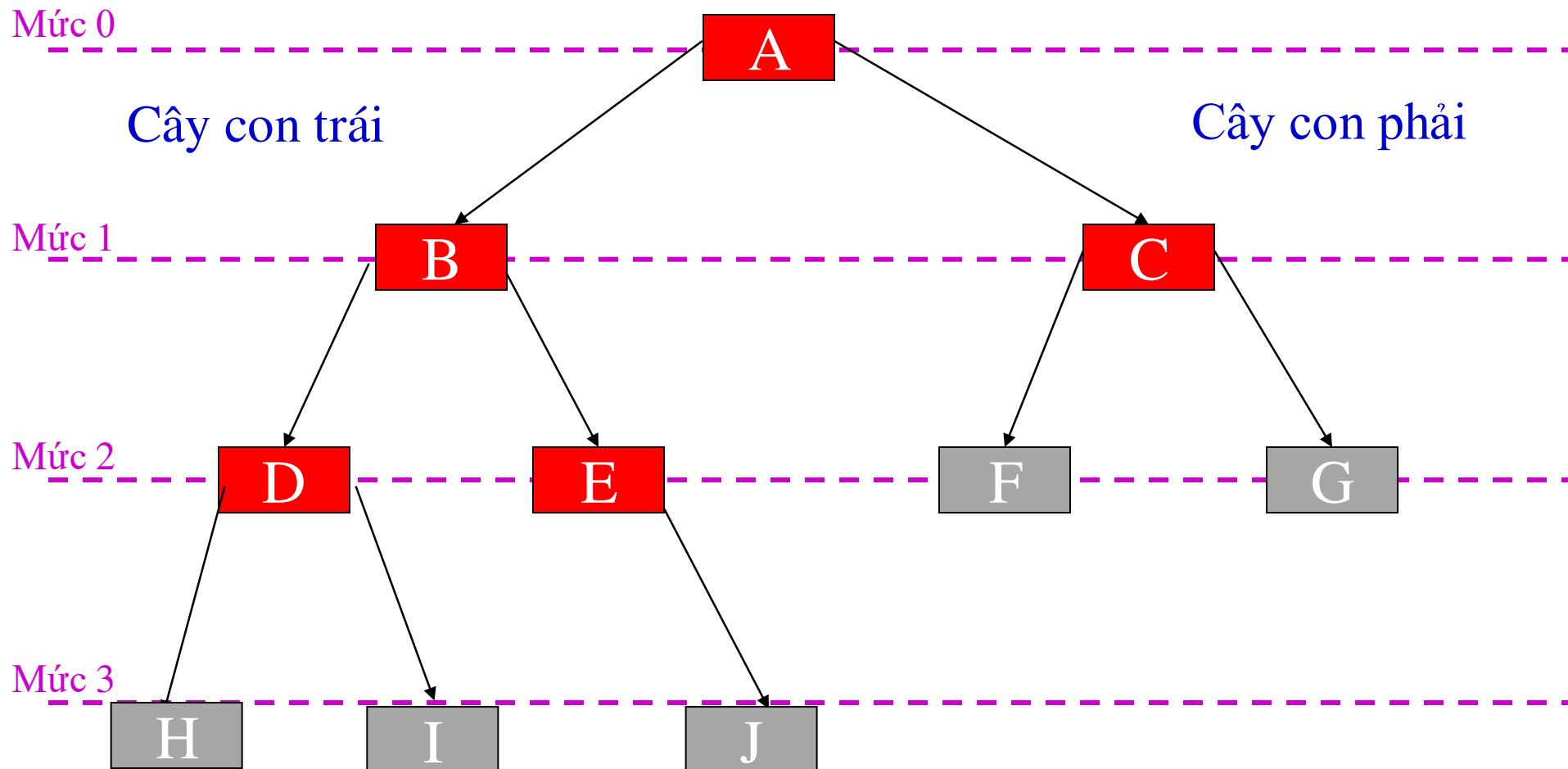
Duyệt cây theo chiều rộng



Duyệt cây theo chiều rộng

- ❖ Bước 6: Kiểm tra queue, queue không rỗng, lấy phần tử đầu tiên (E) ra khỏi queue. Phần tử này chuyển sang trạng thái đã duyệt.
- ❖ Enqueue các con (J) của phần tử vừa được dequeue (E) vào bf_queue.
- ❖ In ra dữ liệu node E.

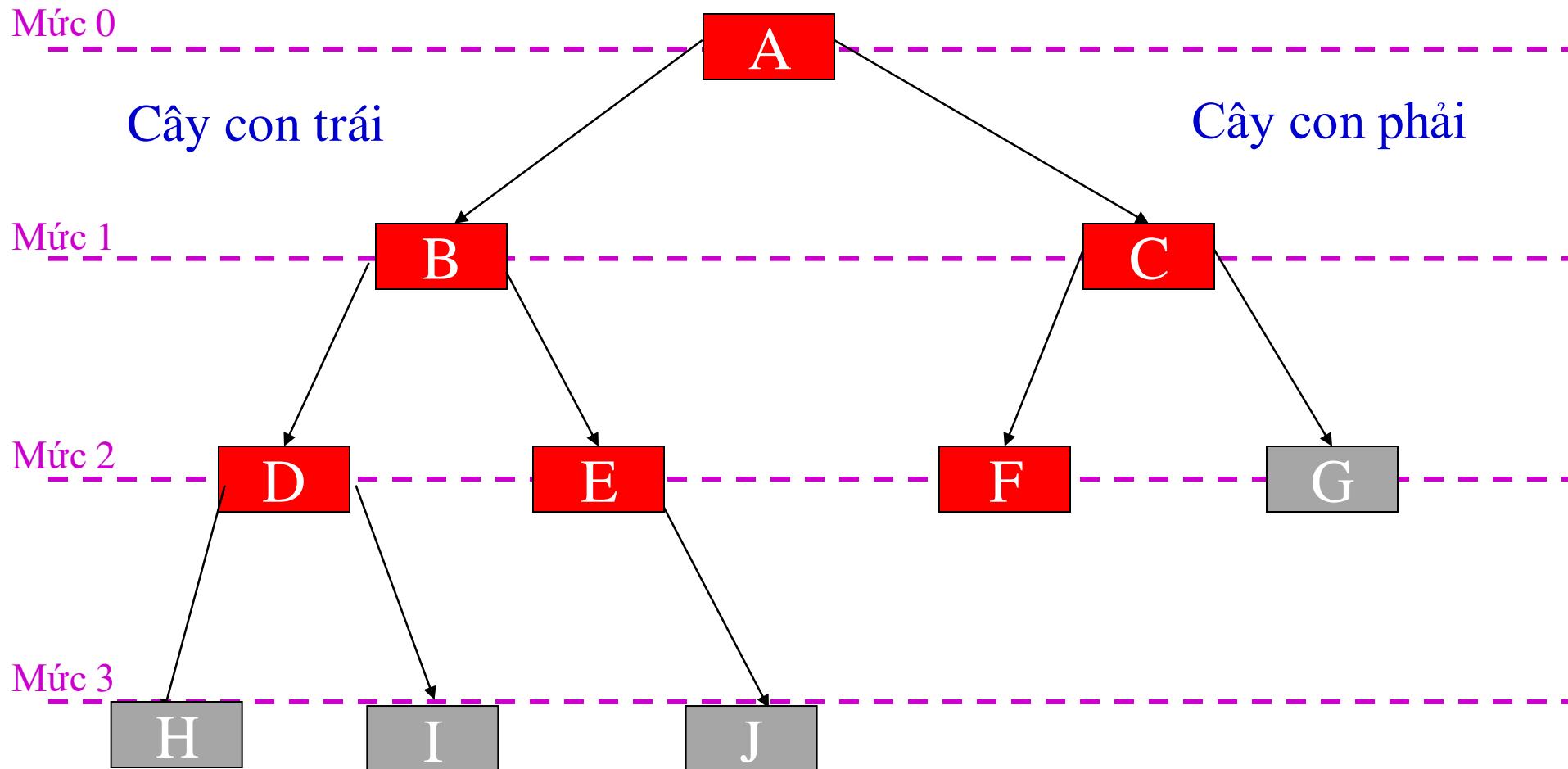
Duyệt cây theo chiều rộng



Duyệt cây theo chiều rộng

- ❖ Bước 7: Kiểm tra queue, queue không rỗng, lấy phần tử đầu tiên (F) ra khỏi queue. Phần tử này chuyển sang trạng thái đã duyệt.
- ❖ F là node lá, không có con nên không cần enqueue.
- ❖ In ra dữ liệu node F

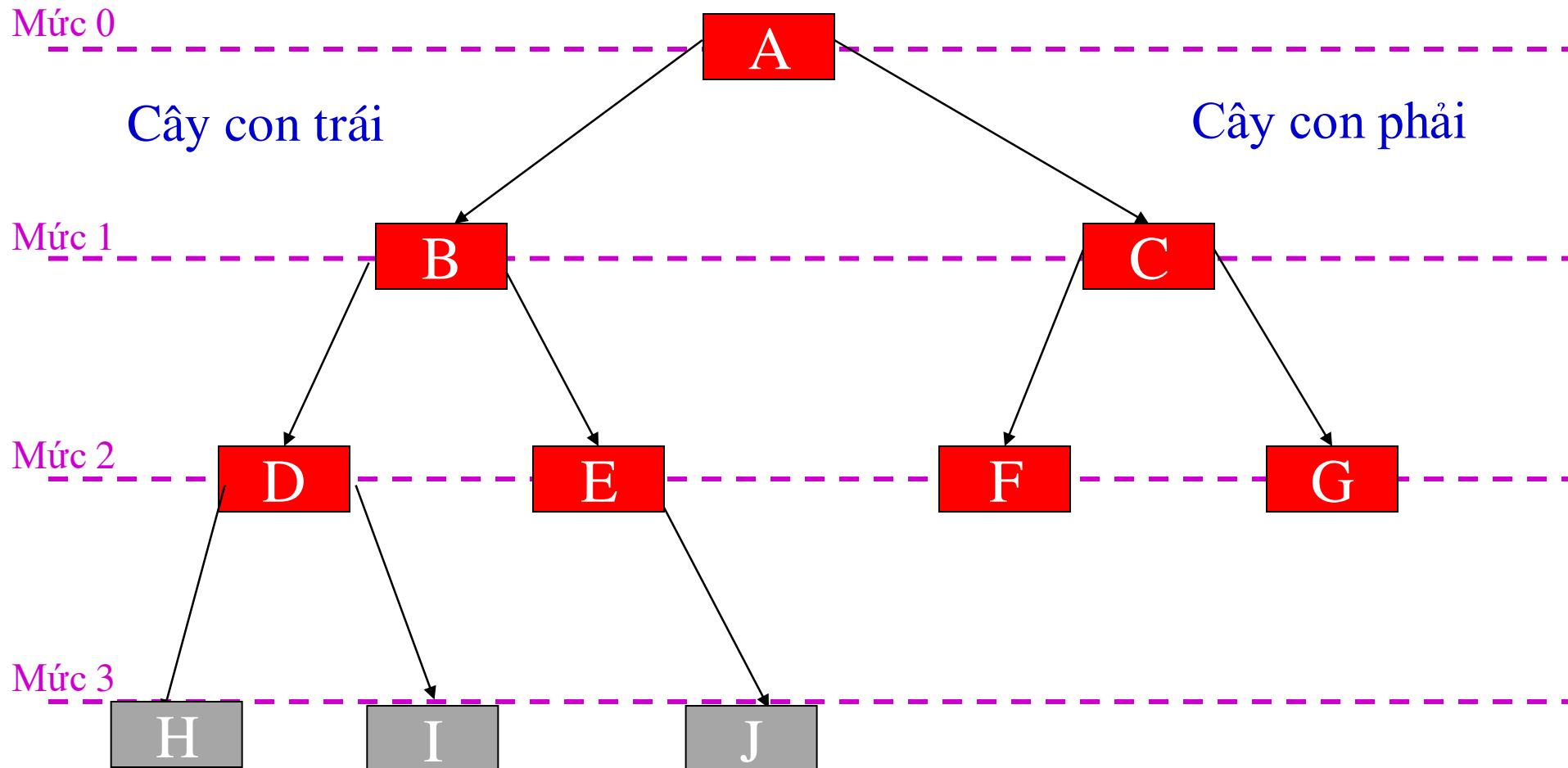
Duyệt cây theo chiều rộng



Duyệt cây theo chiều rộng

- ❖ Bước 8: Kiểm tra queue, queue không rỗng, lấy phần tử đầu tiên (G) ra khỏi queue. Phần tử này chuyển sang trạng thái đã duyệt.
- ❖ G là node lá, không có con nên không cần enqueue.
- ❖ In ra dữ liệu node G.

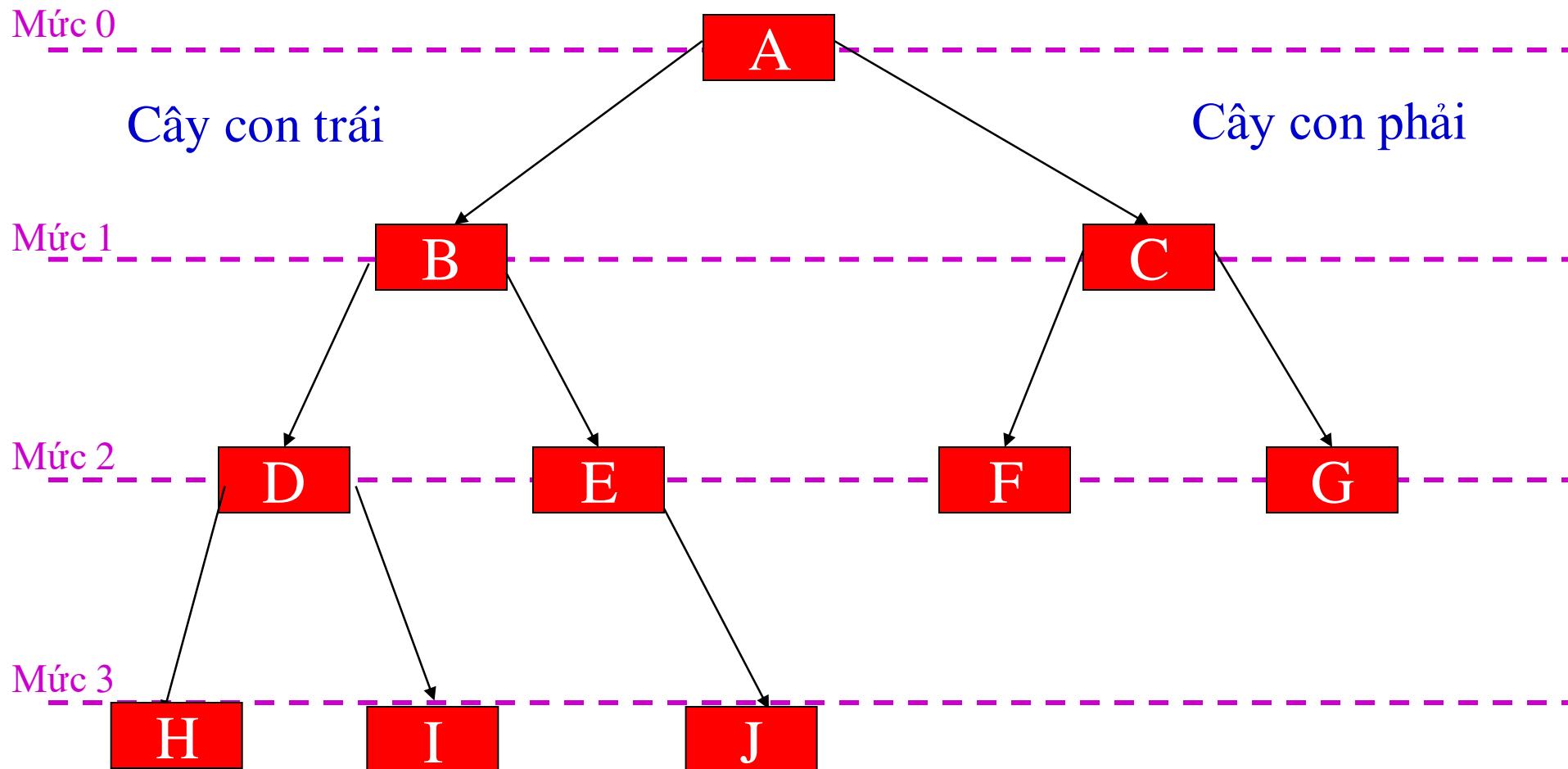
Duyệt cây theo chiều rộng



Duyệt cây theo chiều rộng

- ❖ Thực hiện các bước tương tự như bước 8 cho tới khi hàng đợi rỗng thì ta dừng việc duyệt lại (điều kiện để dừng việc duyệt chính là khi hàng đợi rỗng).
- ❖ Toàn bộ cây đã được duyệt và dữ liệu in ra theo thứ tự: A, B, C, D, E, F, G, H.

Duyệt cây theo chiều rộng



Duyệt cây theo chiều rộng

```
// Function to print Nodes in a binary tree in Level order  
void LevelOrder(Node *root) {  
    if(root == NULL) return;  
    queue<Node*> Q;  
    Q.push(root);  
    //while there is at least one discovered node
```

Duyệt cây theo chiều rộng

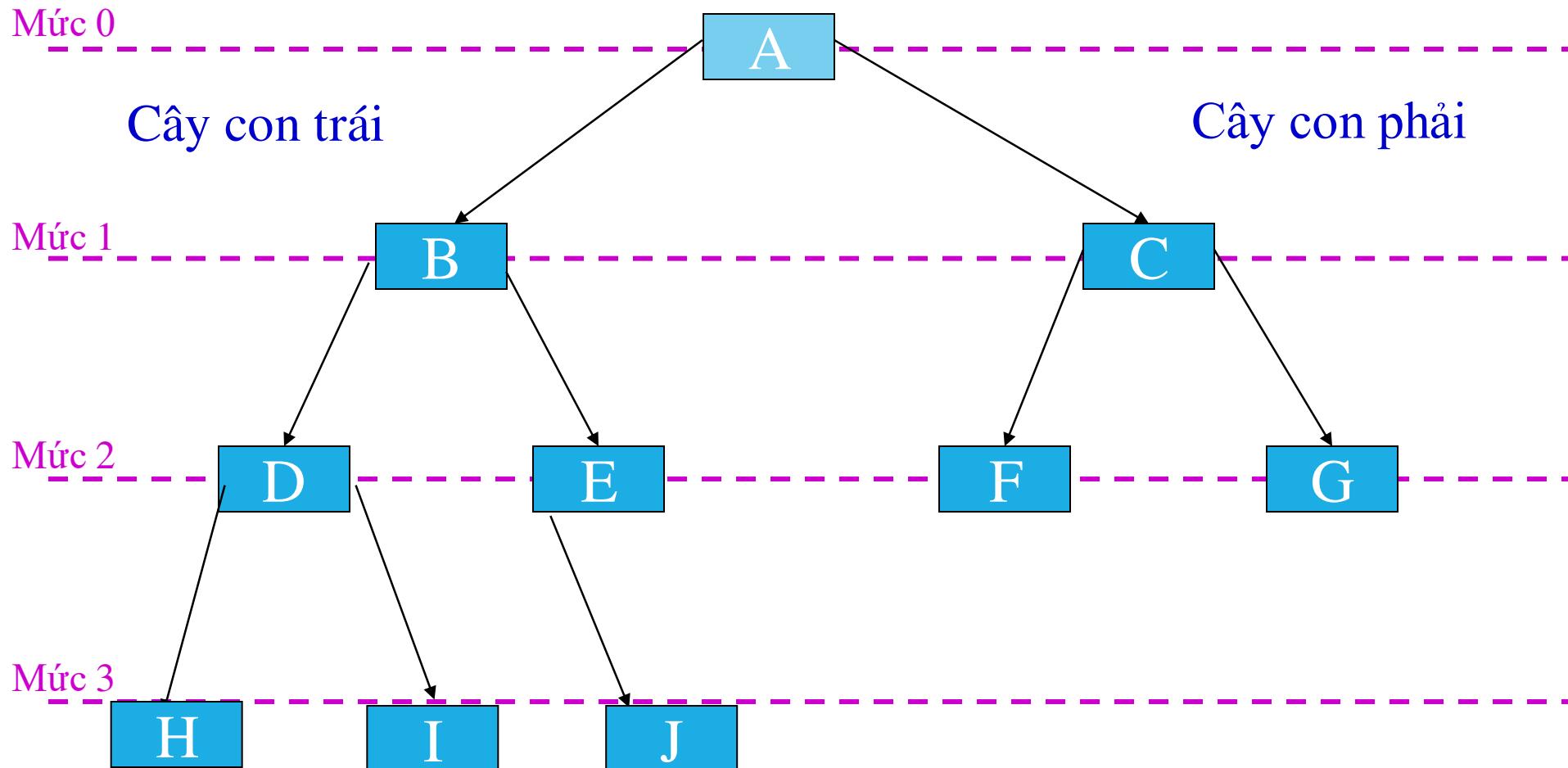
```
while(!Q.empty()) {  
    Node* current = Q.front();  
    Q.pop(); // removing the element at front  
    cout<<current->data<< " ";  
    if(current->left != NULL)  
        Q.push(current->left);  
    if(current->right != NULL)  
        Q.push(current->right);  
}  
}
```

Duyệt cây theo chiều sâu

- ❖ Có 3 cách duyệt theo chiều sâu cây nhị phân là:
Preorder, Inorder, Postorder.

Cách duyệt	Thứ tự		
Preorder	<gốc>	<cây con trái>	<cây con phải>
Inorder	<cây con trái>	<gốc>	<cây con phải>
Postorder	<cây con trái>	<cây con phải>	<gốc>

Duyệt cây theo chiều rộng



Duyệt cây theo chiều sâu

Các bước để duyệt cây với Preorder như sau:

- ❖ Bước 1: Duyệt node gốc.
- ❖ Bước 2: Duyệt cây con bên trái (duyệt tất cả các node của cây con bên trái)
- ❖ Bước 3: Duyệt cây con bên phải (duyệt tất cả các node của cây con bên phải).

Kết quả duyệt: A, B, D, H, I, E, J, C, F, G

Duyệt cây theo chiều sâu

```
//Function to visit nodes in Preorder
void PreOrder(struct Node *root) {
    // base condition for recursion
    // if tree/sub-tree is empty, return and exit
    if(root == NULL) return;
    printf("%c ", root->data); // Print data
    PreOrder(root->left);      // Visit left subtree
    PreOrder(root->right);     // Visit right subtree
}
```

Duyệt cây theo chiều sâu

Các bước để duyệt cây với InOrder:

Bước 1: Duyệt con bên trái.

Bước 2: Duyệt node gốc.

Bước 3: Duyệt con bên phải.

Duyệt cây theo chiều sâu

```
//Function to visit nodes in Inorder
void InOrder(Node *root) {
    if(root == NULL) return;
    InOrder(root->left); //Visit left subtree
    printf("%c ", root->data); //Print data
    InOrder(root->right); //Visit right subtree
}
```

Duyệt cây theo chiều sâu

Các bước để duyệt cây với PostOrder:

Bước 1: Duyệt con bên trái.

Bước 2: Duyệt con bên phải.

Bước 3: Duyệt node gốc.

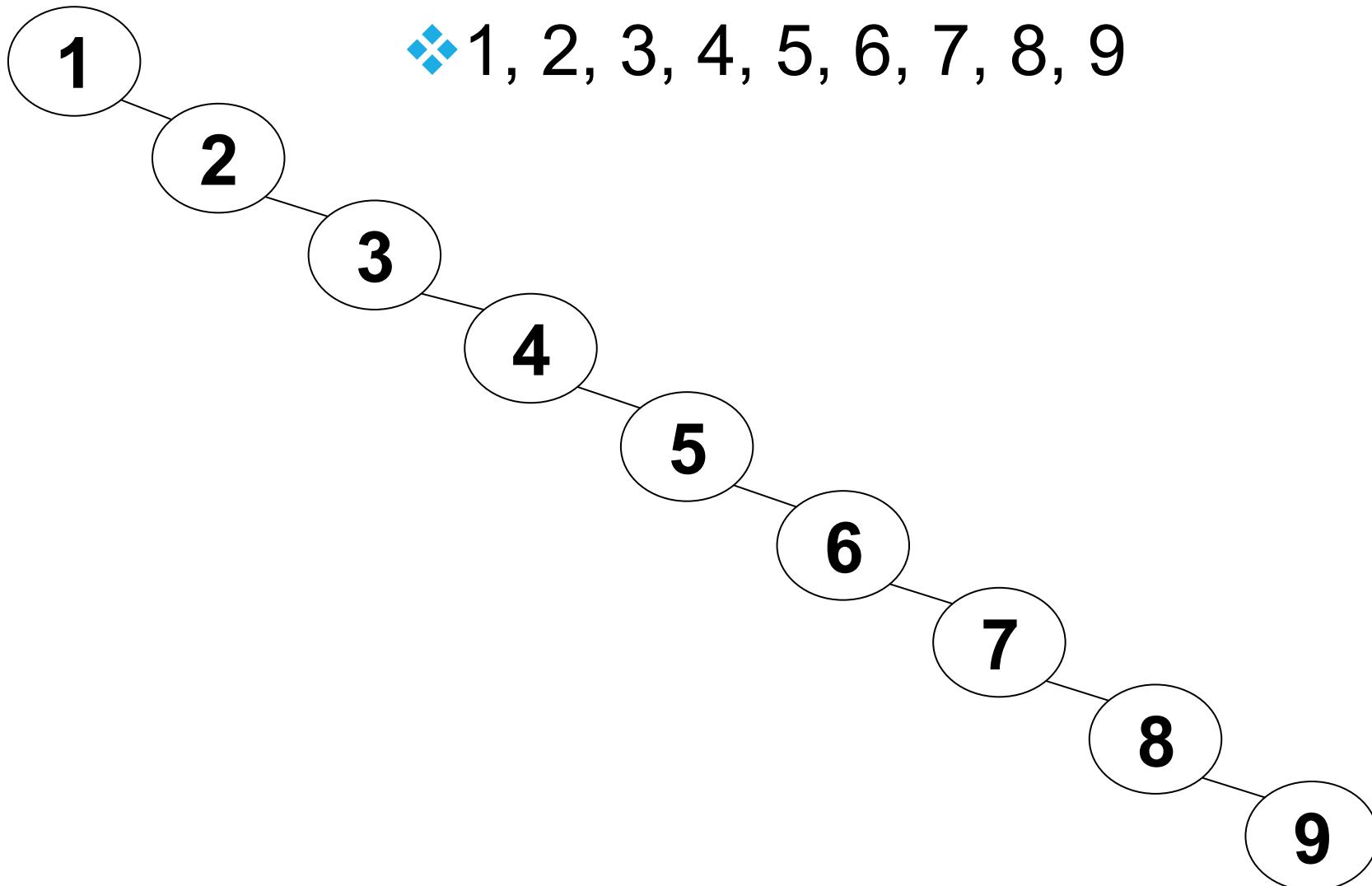
Duyệt cây theo chiều sâu

```
//Function to visit nodes in PostOrder
void PostOrder(Node *root) {
    if(root == NULL) return;
    PostOrder(root->left); // Visit left subtree
    PostOrder(root->right); // Visit right subtree
    printf("%c ", root->data);// Print data
}
```

Nội dung

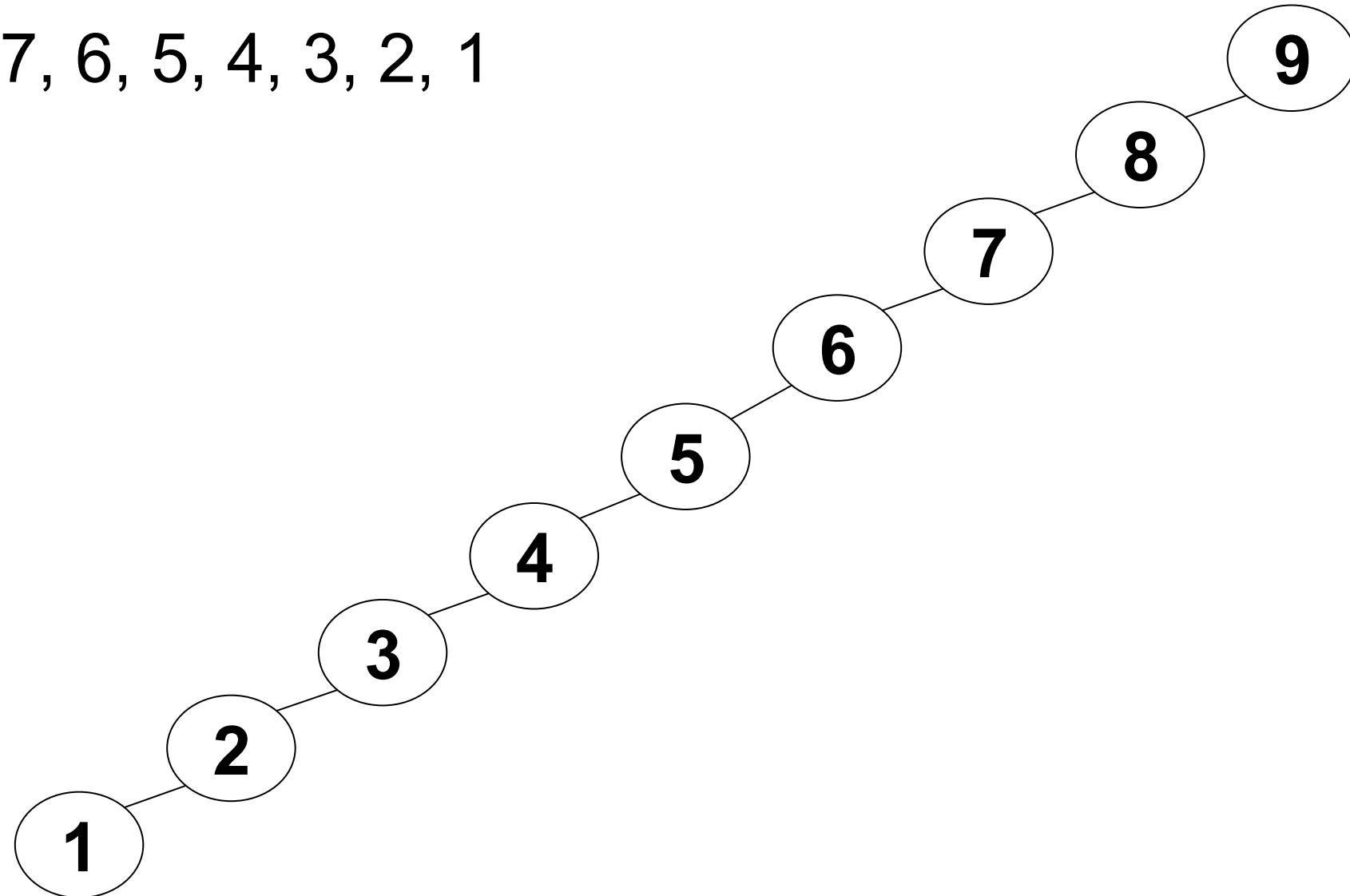
- Cấu trúc cây
- Cây nhị phân
- Cây nhị phân tìm kiếm
- Duyệt cây
- **Cây AVL**
- Cây đỏ đen
- Cây B-Tree

Đánh giá tìm kiếm



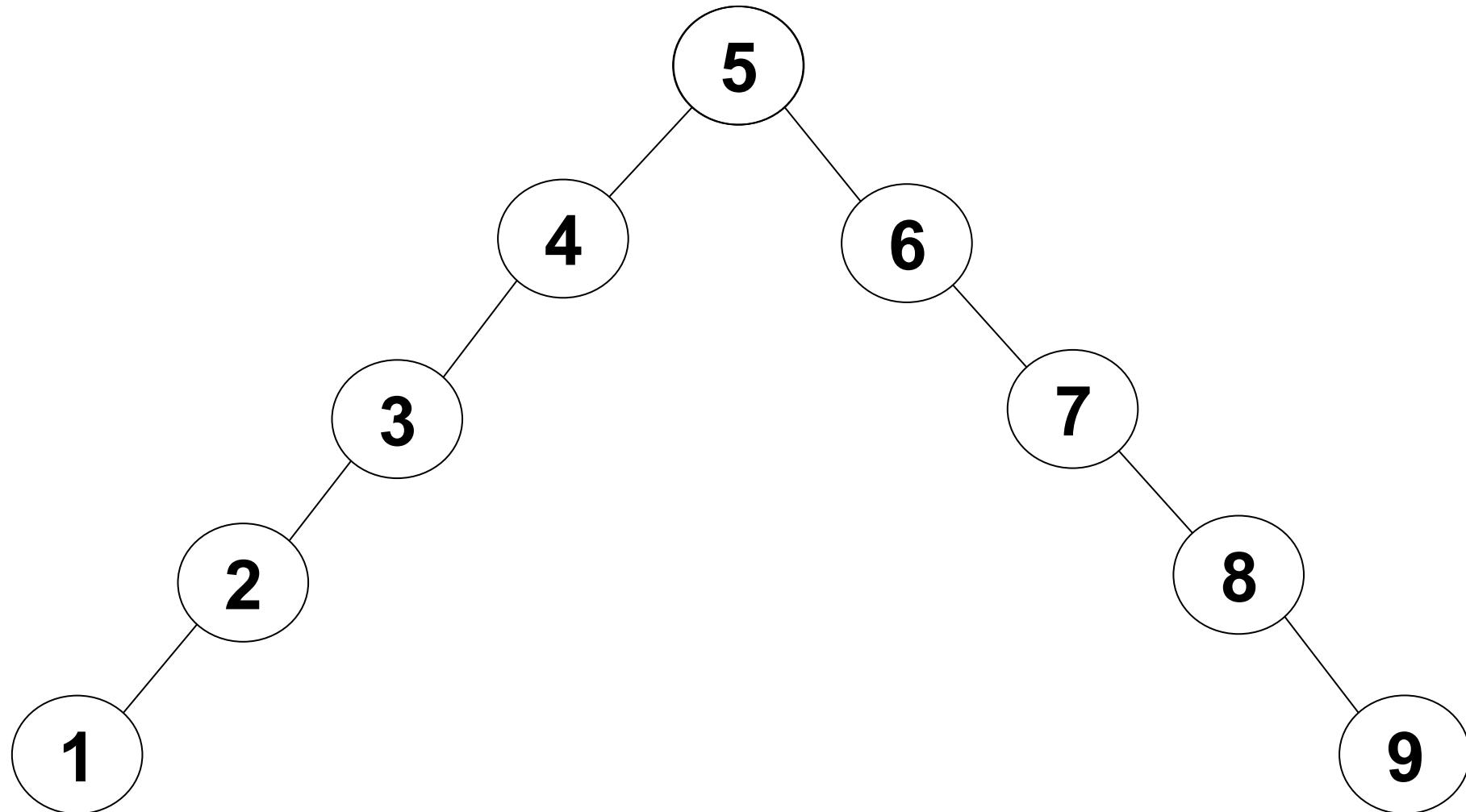
Đánh giá tìm kiếm

- ❖ 9, 8, 7, 6, 5, 4, 3, 2, 1



Đánh giá tìm kiếm

- ❖ 5, 4, 3, 2, 1, 6, 7, 8, 9



AVL Tree - Giới thiệu

- ❖ Phương pháp thêm trên Cây NPTK có thể có những biến dạng mất cân đối nghiêm trọng
 - ❖ Chi phí cho việc tìm kiếm trong trường hợp xấu nhất đạt tới n
 - ❖ VD: 1 triệu nút \Rightarrow chi phí tìm kiếm = 1.000.000 nút
- ❖ Nếu có một cây tìm kiếm nhị phân cân bằng hoàn toàn, chi phí cho việc tìm kiếm chỉ xấp xỉ $\log_2 n$
 - ❖ VD: 1 triệu nút \Rightarrow chi phí tìm kiếm = $\log_2 1.000.000 \approx 20$ nút
- ❖ G.M. Adelson - Velsky và E.M. Landis đã đề xuất một tiêu chuẩn cân bằng (gọi là cân bằng **AVL**)
 - ❖ Cây AVL có chiều cao **$O(\log_2(n))$**

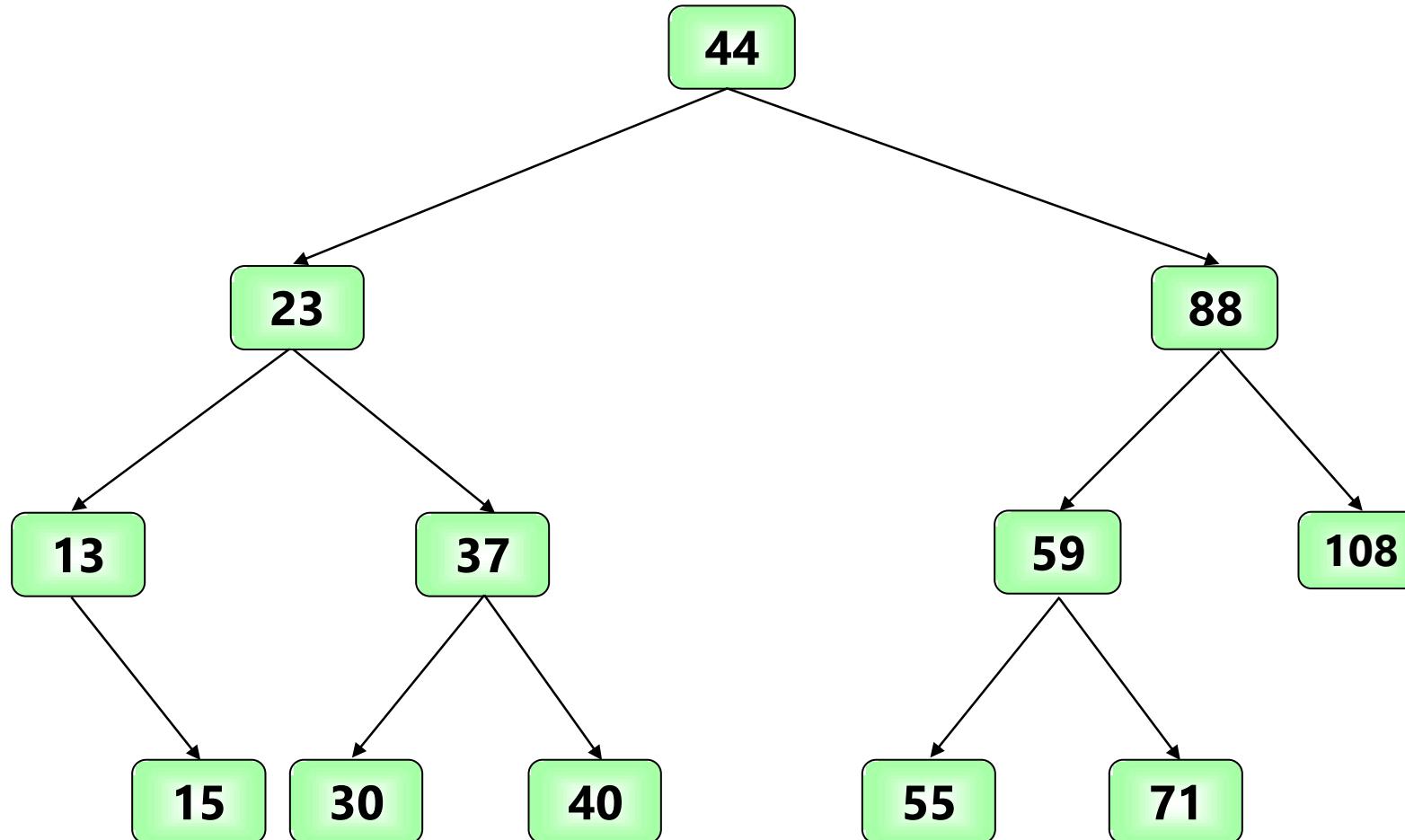
Demo

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

AVL Tree - Định nghĩa

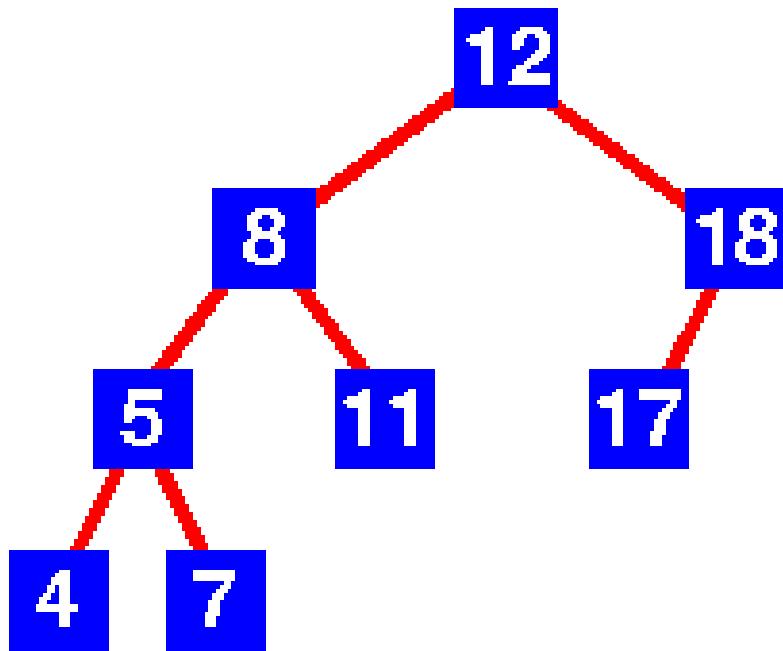
- Cây nhị phân tìm kiếm cân bằng là cây mà **tại mỗi nút** của nó **độ cao** của cây con trái và của cây con phải **chênh lệch nhau không quá một**.

Ví dụ:

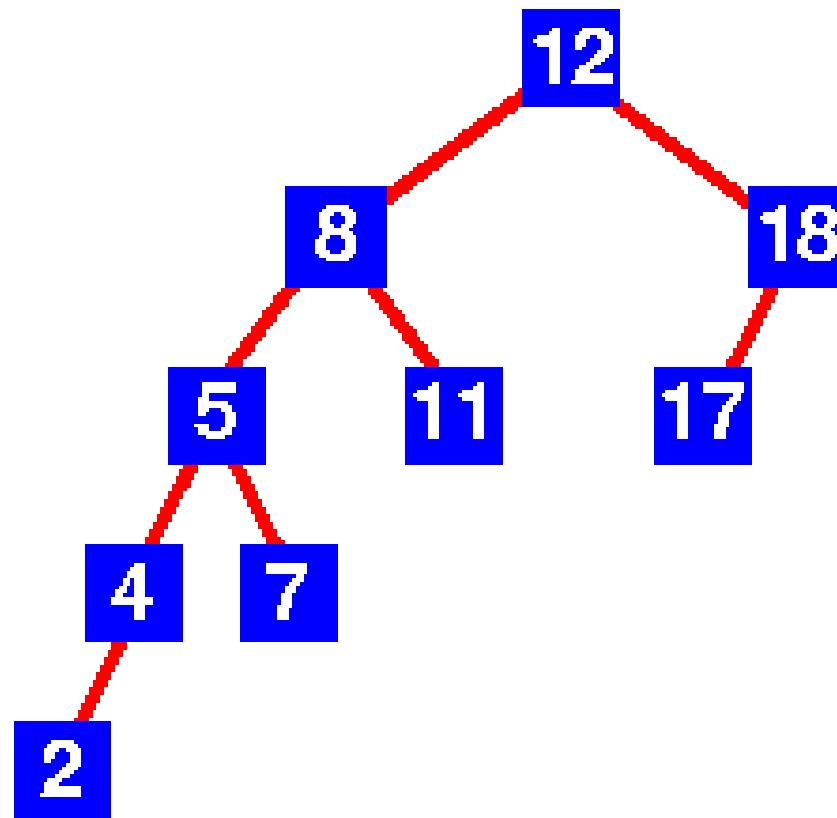


AVL Tree – Ví dụ

AVL Tree ?



AVL Tree?



Tổ chức dữ liệu

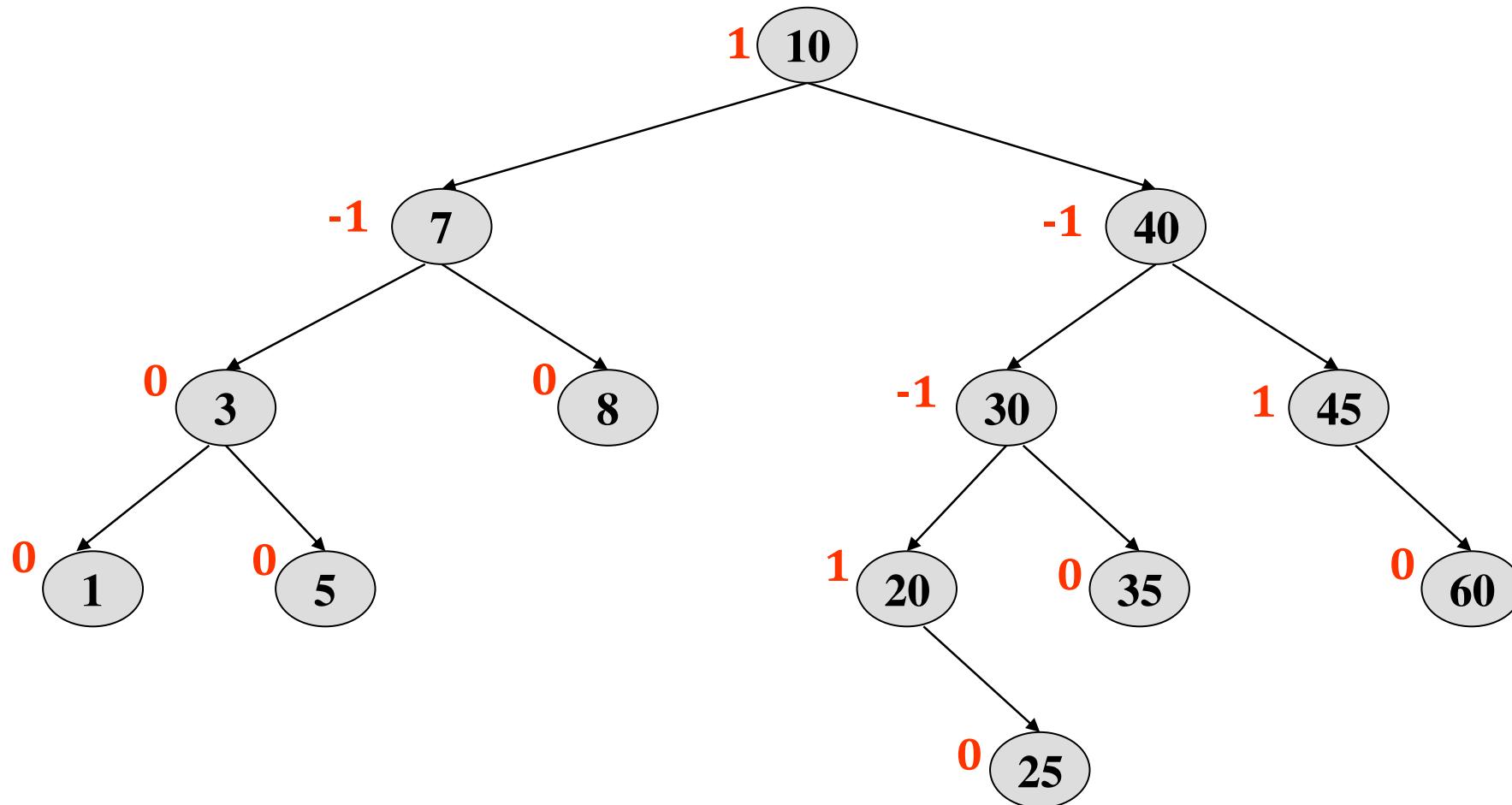
❖ Chỉ số cân bằng của một nút:

- ❖ Định nghĩa: Chỉ số cân bằng của một nút là **hiệu** của chiều cao cây con phải và cây con trái của nó.
- ❖ Đối với một cây cân bằng, chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một trong ba giá trị: -1, 0, và 1:

❖ Các giá trị hợp lệ:

- **CSCB(p)=0** \Leftrightarrow Độ cao cây trái (p) = Độ cao cây phải (p)
- **CSCB(p)=1** \Leftrightarrow Độ cao cây trái (p) < Độ cao cây phải (p)
- **CSCB(p)=-1** \Leftrightarrow Độ cao cây trái (p) > Độ cao cây phải (p)

Ví dụ - Chỉ số cân bằng của nút



- Hãy cho biết chỉ số cân bằng của mỗi nút trên cây?
- Đây có phải là cây AVL không?

Tổ chức dữ liệu (tt)

```
#define LH -1          //cây con trái cao hơn
#define EH 0           //cây con trái bằng cây con phải
#define RH 1           //cây con phải cao hơn
typedef <Kiểu dữ Liệu của nút> ItemType;
struct AVLNode
{
    int balFactor; //chỉ số cân bằng của cây
    ItemType Info;
    AVLNode* Left;
    AVLNode* Right;
};
struct AVLTree
{
    AVLNode* Root;
};
```

Cấp phát nút mới cho cây AVL

```
AVLNode* createAVLNode(ItemType x) {  
    AVLNode* p = new AVLNode;  
    if(p == NULL) {  
        printf("\nKhong du bo nho cap phat nut moi!");  
        getch();  
        return NULL;  
    }  
    p->balFactor = 0;  
    p->Info = x;  
    p->Left = NULL;  
    p->Right = NULL;  
    return p;  
}
```

Khởi tạo cây AVL

```
void initAVLTree(AVLTree &avl)
{
    avl.Root = NULL;
}
```

AVL Tree – Biểu diễn

- ❖ Các thao tác đặc trưng của cây AVL:
 - ❖ Thêm một phần tử vào cây AVL
 - ❖ Xóa một phần tử trên cây AVL
 - ❖ Cân bằng lại một cây vừa bị mất cân bằng (**Rotation**)
- ❖ Trường hợp **thêm** một phần tử trên cây AVL được thực hiện giống như thêm trên cây NPTK, tuy nhiên sau khi thêm phải cân bằng lại cây.
- ❖ Trường hợp **xóa** một phần tử trên cây AVL được thực hiện giống như hủy trên cây NPTK và cũng phải cân bằng lại cây.
- ❖ Việc **cân bằng lại** một cây sẽ phải thực hiện sao cho chỉ ảnh hưởng tối thiểu đến cây nhằm giảm thiểu chi phí cân bằng

Các trường hợp mất cân bằng

- ❖ Không khảo sát tính cân bằng của 1 cây nhị phân bất kỳ mà chỉ quan tâm đến khả năng mất cân bằng xảy ra khi **thêm** hoặc **xóa** một nút trên cây AVL.
- ❖ Các trường hợp mất cân bằng:
 - Sau khi thêm (xóa) cây con **trái lệch trái** (left of left)
 - Sau khi thêm (xóa) cây con **trái lệch phải** (right of left)
 - Sau khi thêm (xóa) cây con **phải lệch phải** (right of right)
 - Sau khi thêm (xóa) cây con **phải lệch trái** (left of right)

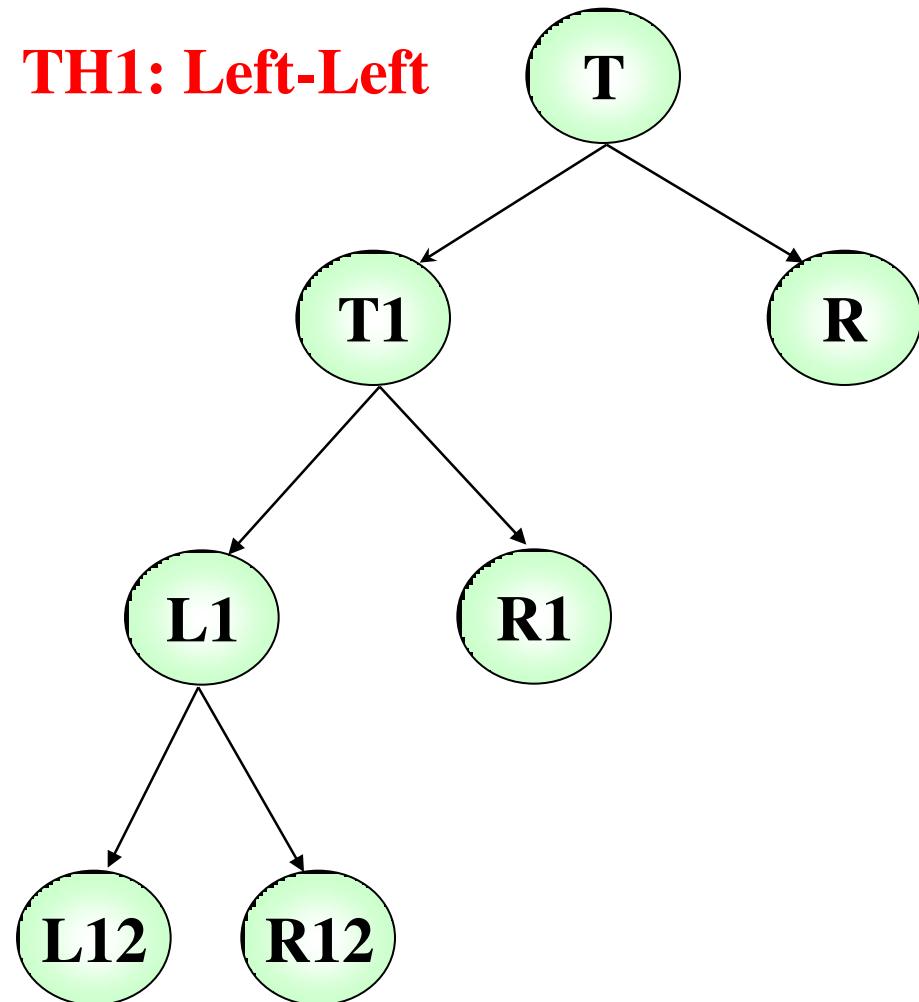
Các thao tác trên cây cân bằng

- ❖ Cây có khả năng mất cân bằng khi thay đổi chiều cao:
 - Thêm bên trái → Lệch nhánh trái
 - Thêm bên phải → Lệch nhánh phải
 - Hủy bên phải → Lệch nhánh trái
 - Hủy bên trái → Lệch nhánh phải
- ❖ **Cân bằng lại cây:** tìm cách bố trí lại cây sao cho chiều cao 2 cây con cân đối:
 - Kéo nhánh cao bù cho nhánh thấp
 - Phải bảo đảm cây vẫn là cây AVL.

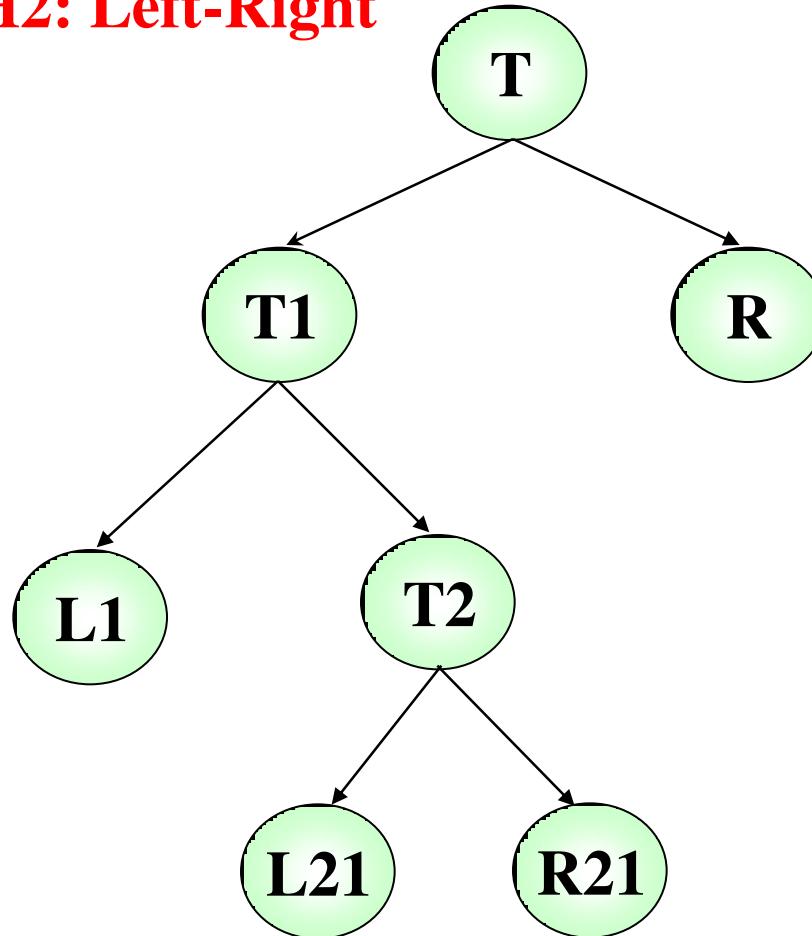
Các trường hợp mất cân bằng do lệch trái

➤ Cây mất cân bằng tại nút T

TH1: Left-Left



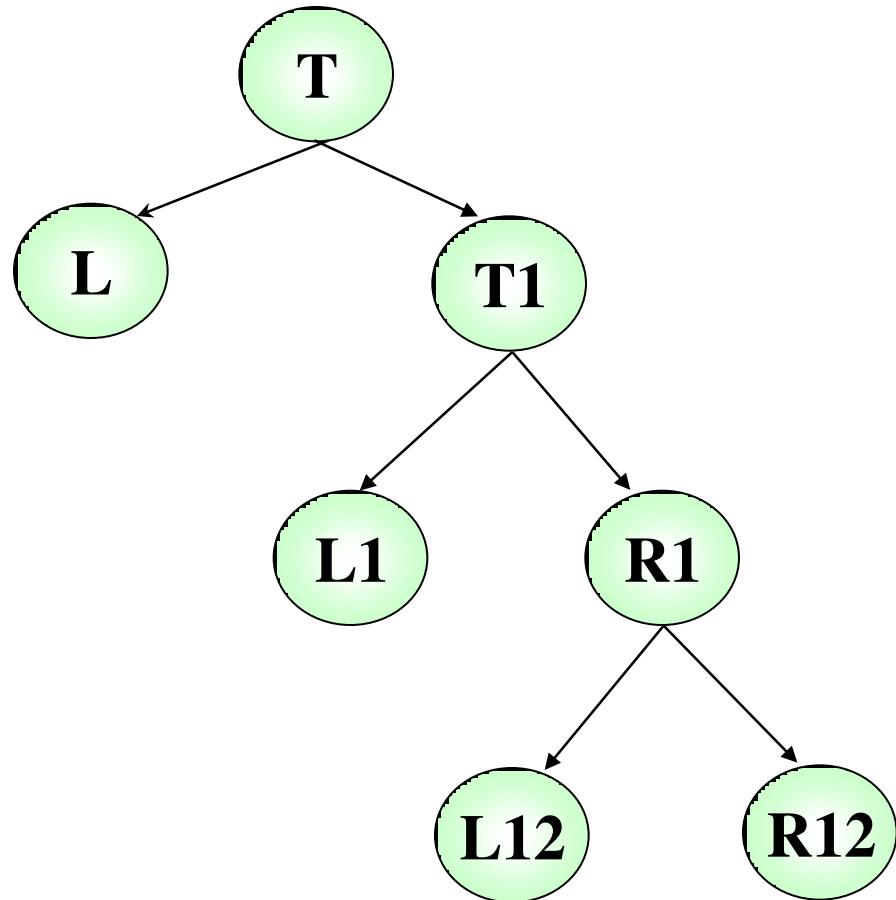
TH2: Left-Right



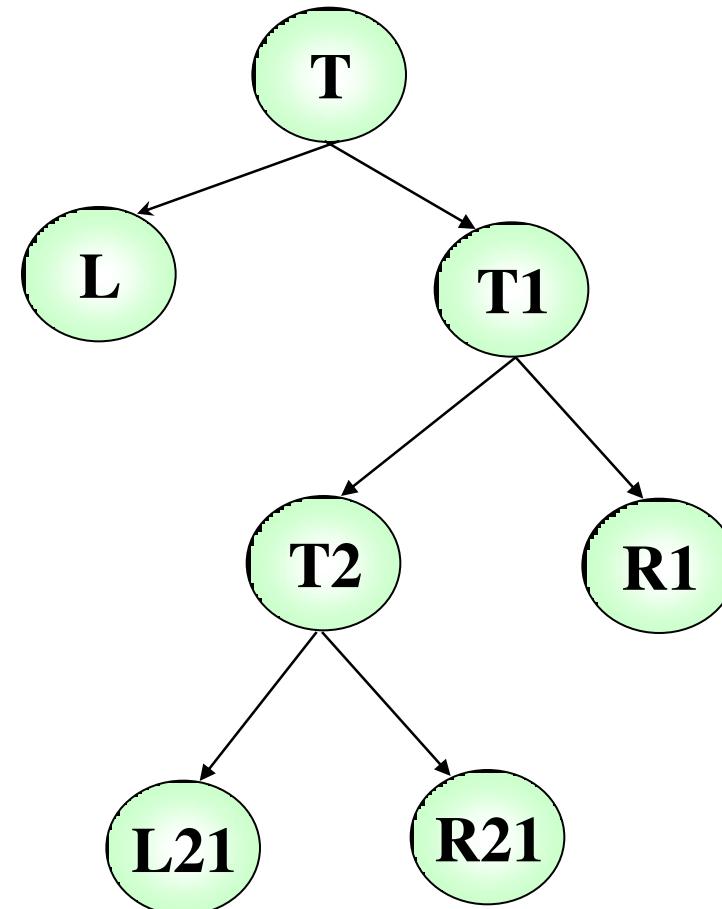
Các trường hợp mất cân bằng do lệch phải

➤ Cây mất cân bằng tại nút T

TH3: Right-Right



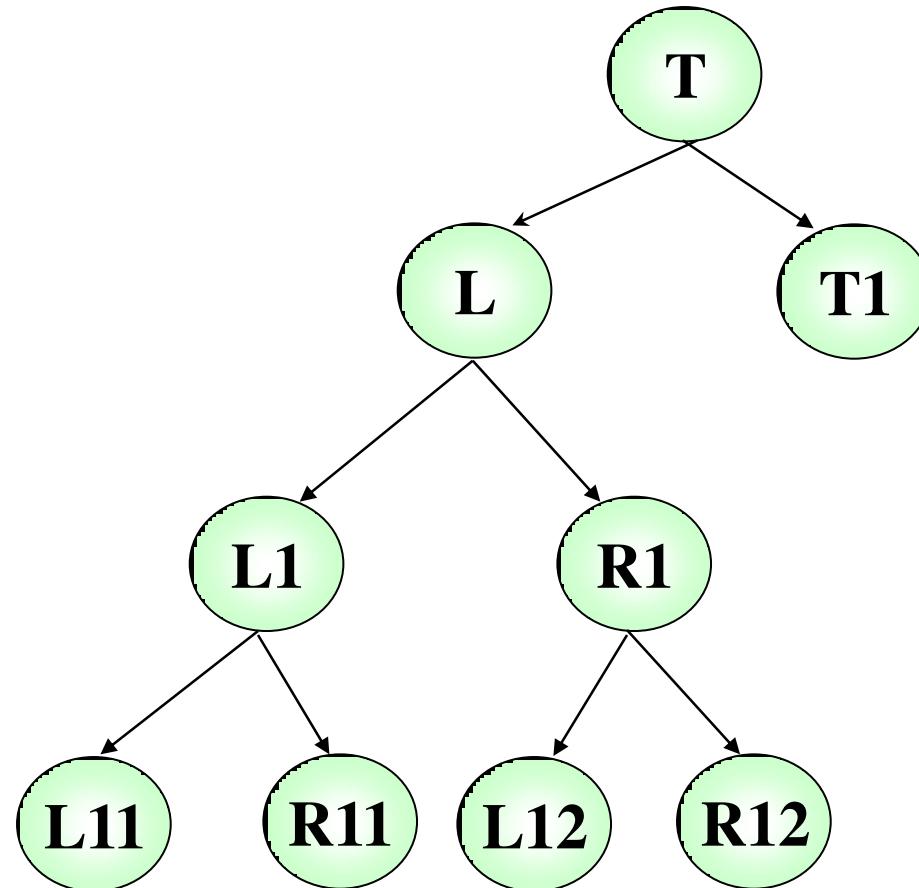
TH4: Right-Left



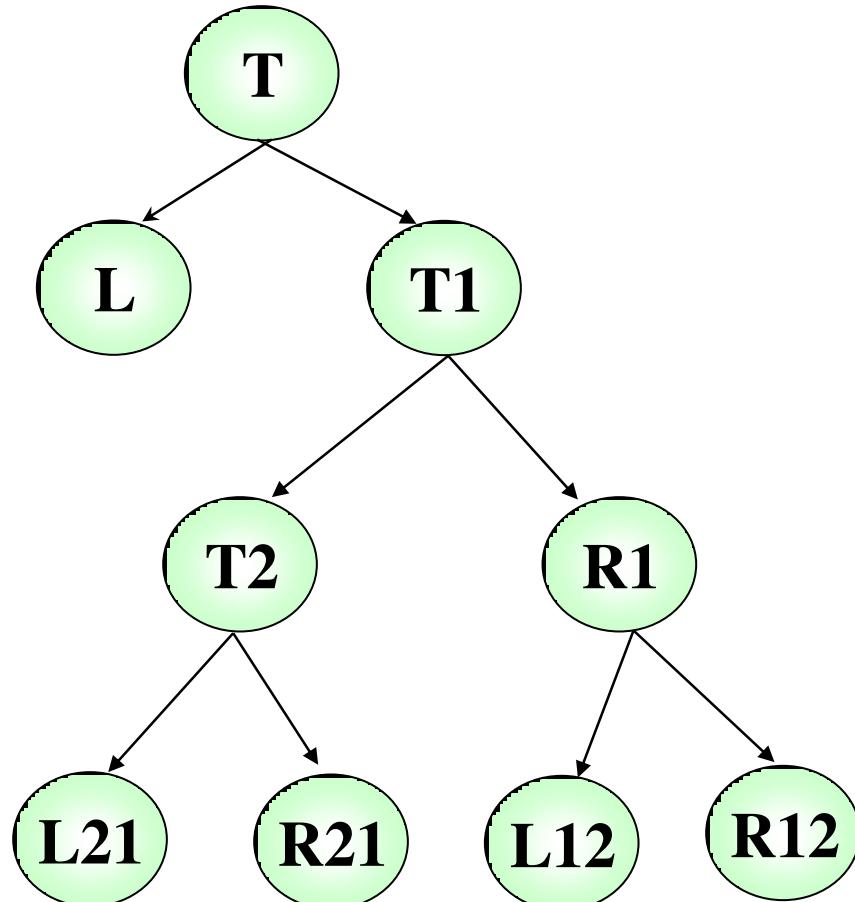
Các trường hợp mất cân bằng do xóa

➤ Cây mất cân bằng tại nút T

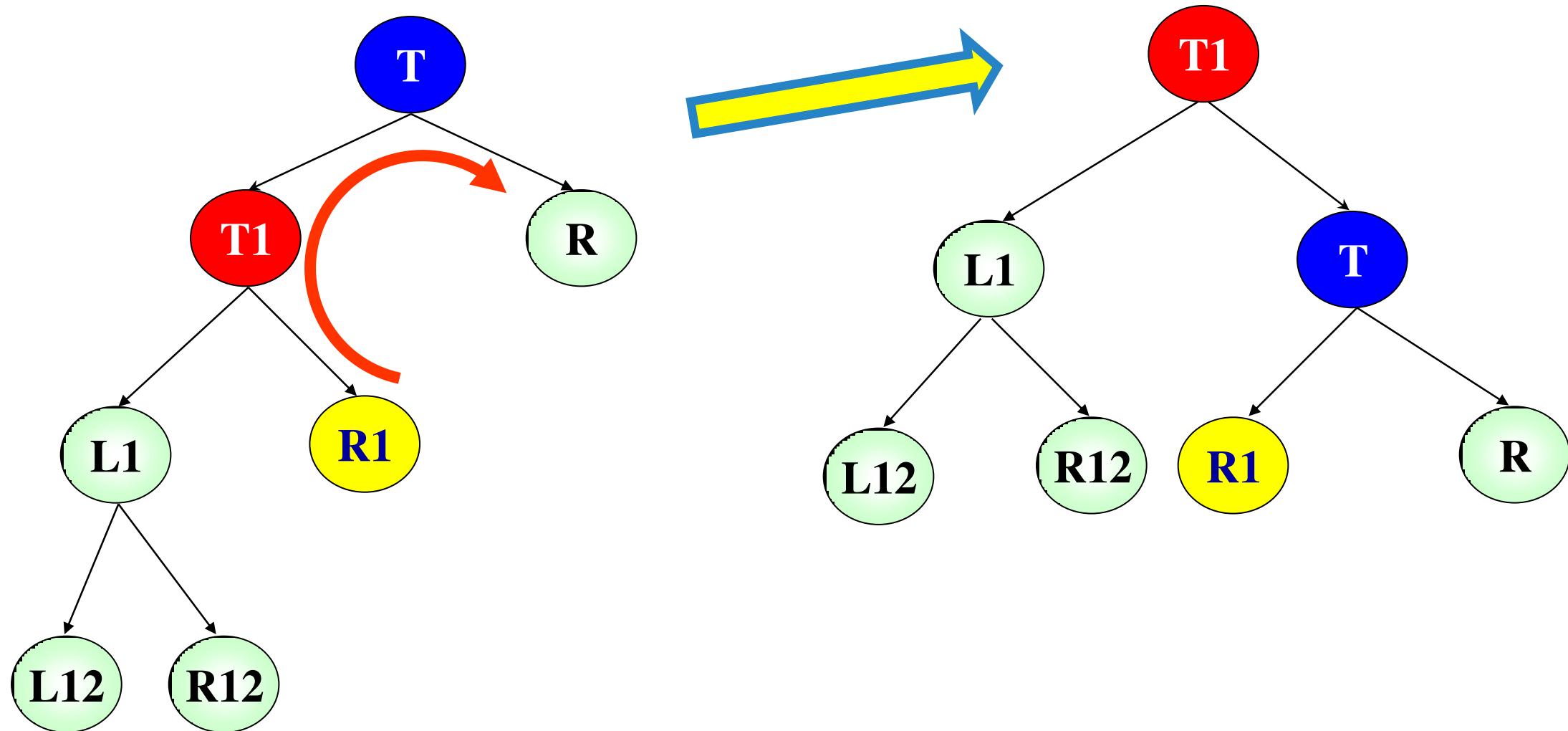
TH5: Left-Balance



TH6: Right-Balance



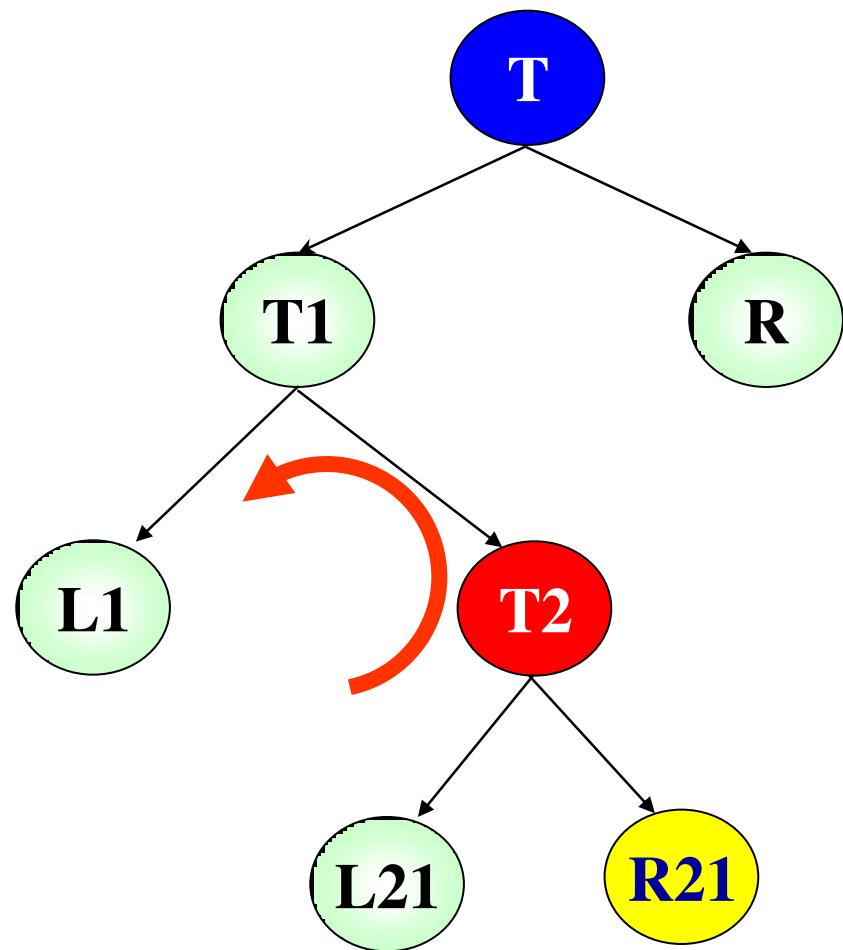
Cân bằng lại trường hợp 1



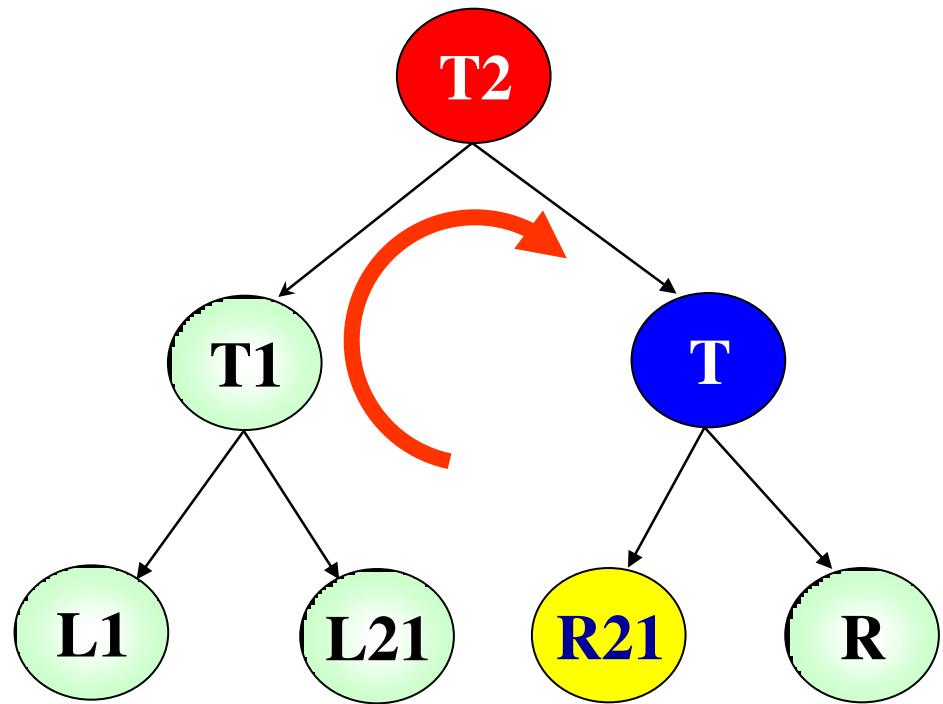
Cài đặt cân bằng lại cho trường hợp 1

```
void rotateLL(AVLTree &T) {  
    AVLNode *T1 = T→Left;  
    T→Left = T1→Right;  
    T1→Right = T;  
    switch(T1→balFactor) {  
        case LH:   T→balFactor = EH;  
                    T1→balFactor = EH; break;  
        case EH:   T→balFactor = LH;  
                    T1→balFactor = RH; break;  
    }  
    T = T1;  
}
```

Cân bằng lại trường hợp 2



Xoay trái tại nút T_1

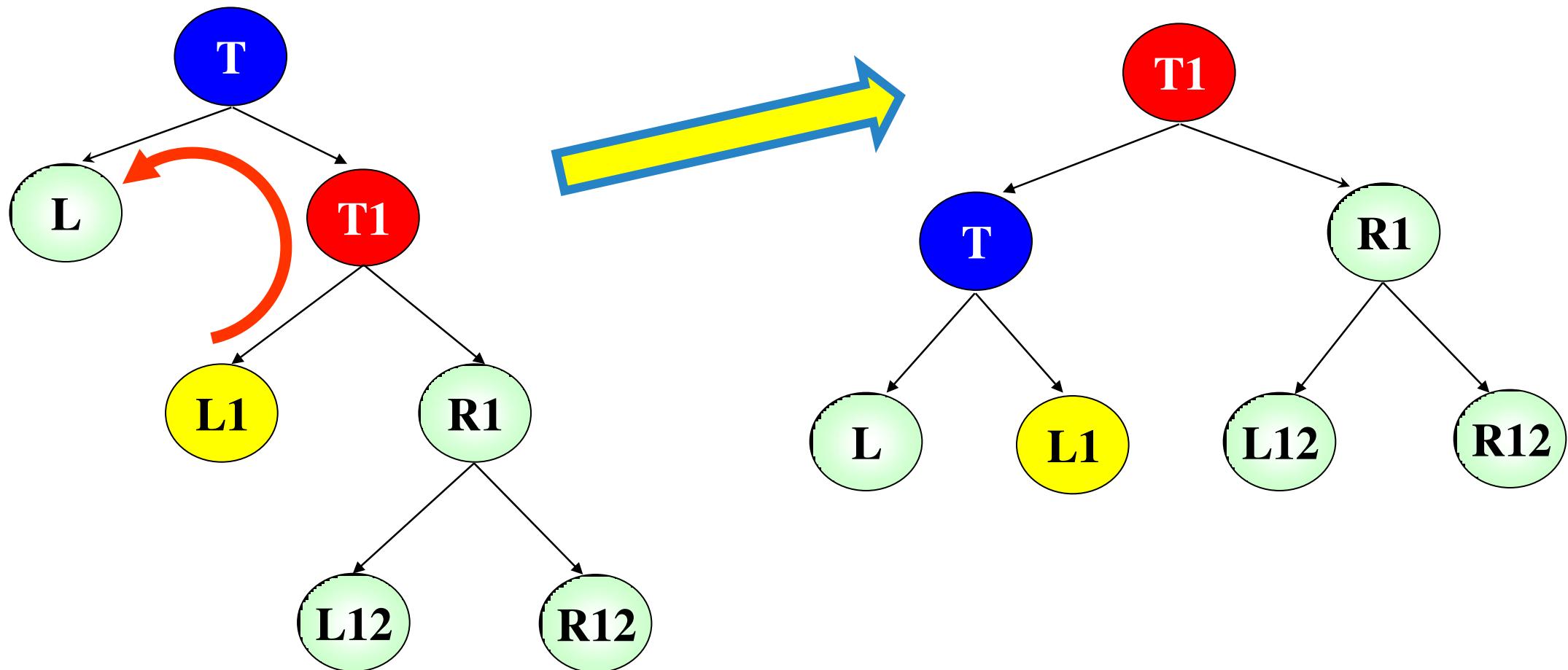


Xoay phải tại nút T_2

Cài đặt cân bằng lại cho trường hợp 2

```
void rotateLR(AVLTree &T) {  
    AVLNode *T1 = T→Left;  
    AVLNode *T2 = T1→Right;  
    T1→Right = T2→Left;    T2→Left = T1;  
    T→Left = T2→Right;    T2→Right = T;  
    switch(T2→balFactor) {  
        case LH:   T→balFactor = RH;  
                    T1→balFactor = EH; break;  
        case EH:   T→balFactor = EH;  
                    T1→balFactor = EH; break;  
        case RH:   T→balFactor = EH;  
                    T1→balFactor = LH; break;  
    }  
    T2→balFactor = EH; T = T2;  
}
```

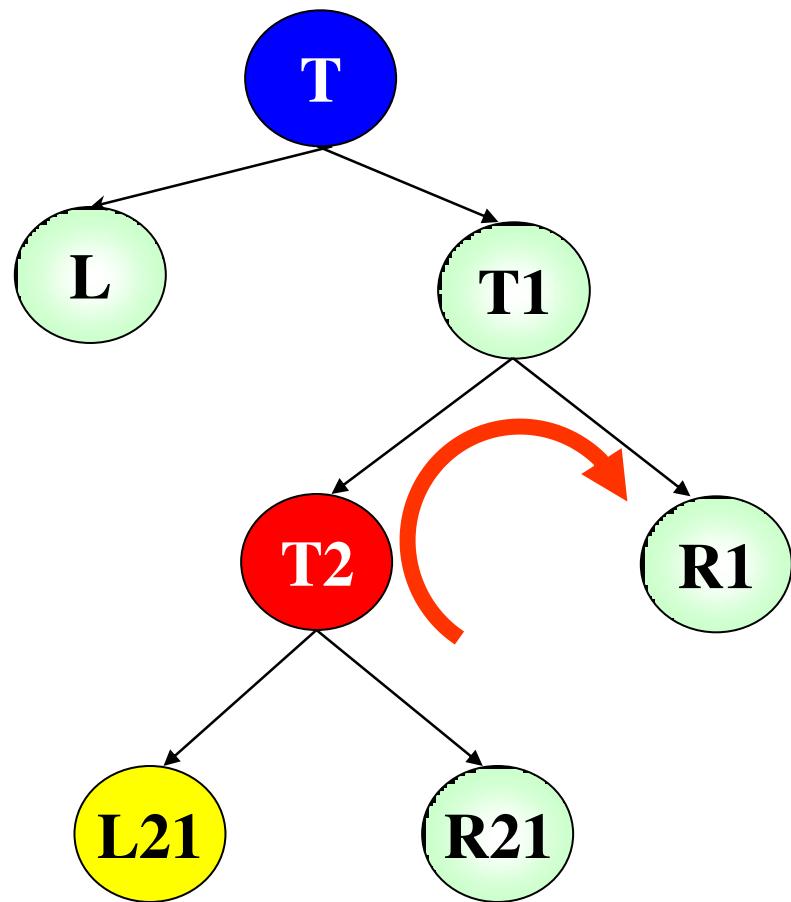
Cân bằng lại trường hợp 3



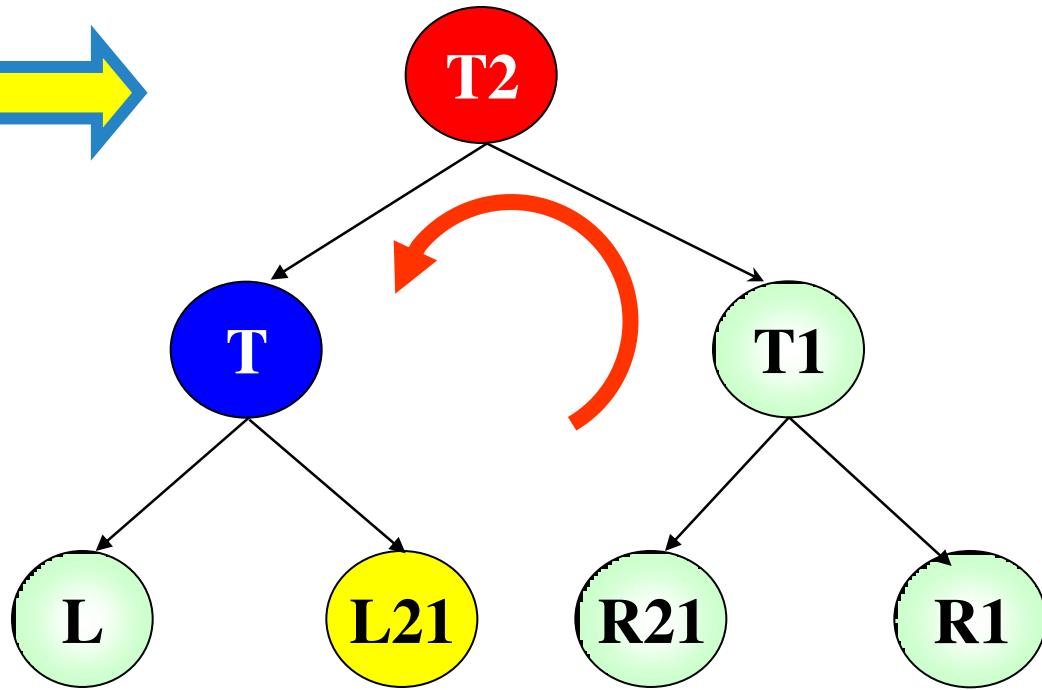
Cài đặt cân bằng lại cho trường hợp 3

```
void rotateRR(AVLTree &T) {  
    AVLNode *T1 = T→Right;  
    T→Right = T1→Left; T1→Left = T;  
    switch(T1→balFactor) {  
        case RH: T→balFactor = EH;  
                    T1→balFactor = EH; break;  
        case EH: T→balFactor = RH;  
                    T1→balFactor = LH; break;  
    }  
    T = T1;  
}
```

Cân bằng lại trường hợp 4



Xoay phải tại nút T_1

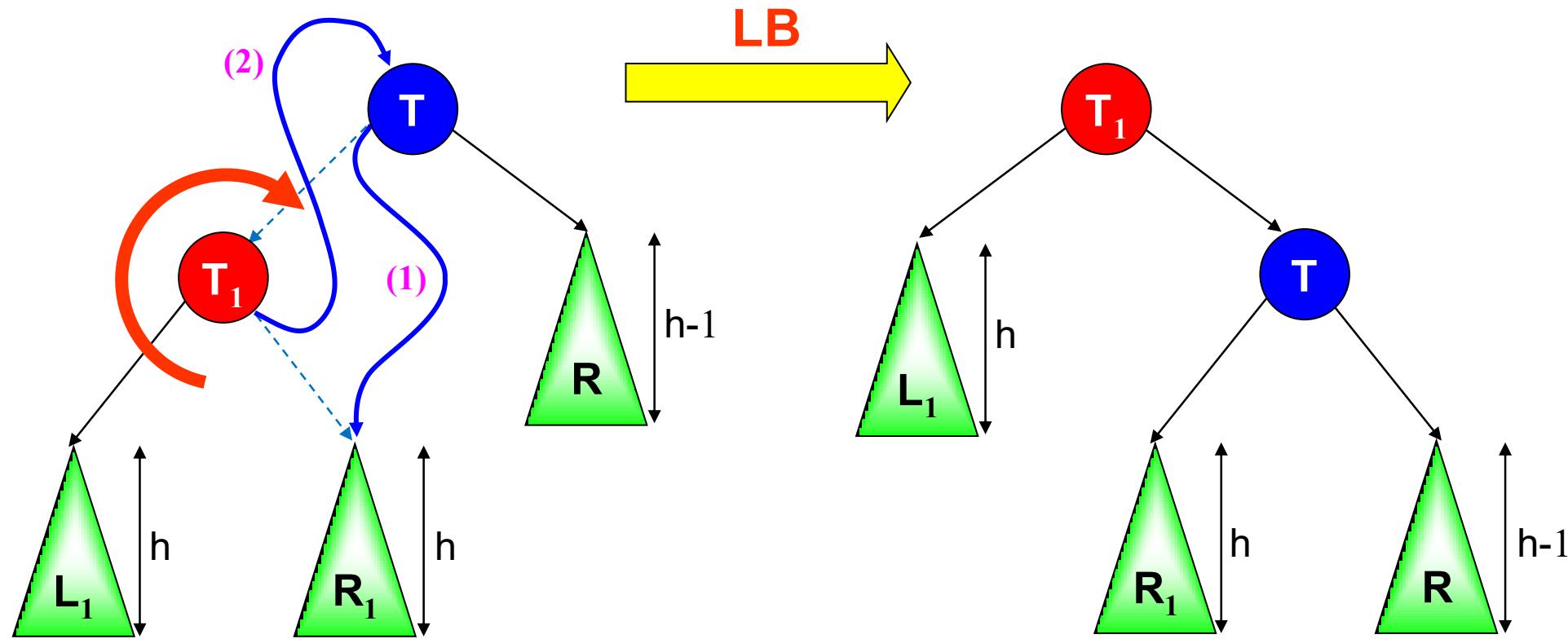


Xoay trái tại nút T_2

Cài đặt cân bằng lại cho trường hợp 4

```
void rotateRL(AVLTree &T) {  
    AVLNode *T1 = T→Right;  
    AVLNode *T2 = T1→Left;  
    T1→Left = T2→Right; T2→Right = T1;  
    T→Right = T2→Left; T2→Left = T;  
    switch(T2→balFactor) {  
        case RH: T→balFactor = LH;  
                  T1→balFactor = EH; break;  
        case EH: T→balFactor = EH;  
                  T1→balFactor = EH; break;  
        case LH: T→balFactor = EH;  
                  T1→balFactor = RH; break;  
    }  
    T2→balFactor = EH; T = T2;  
}
```

Cân bằng lại trường hợp 5

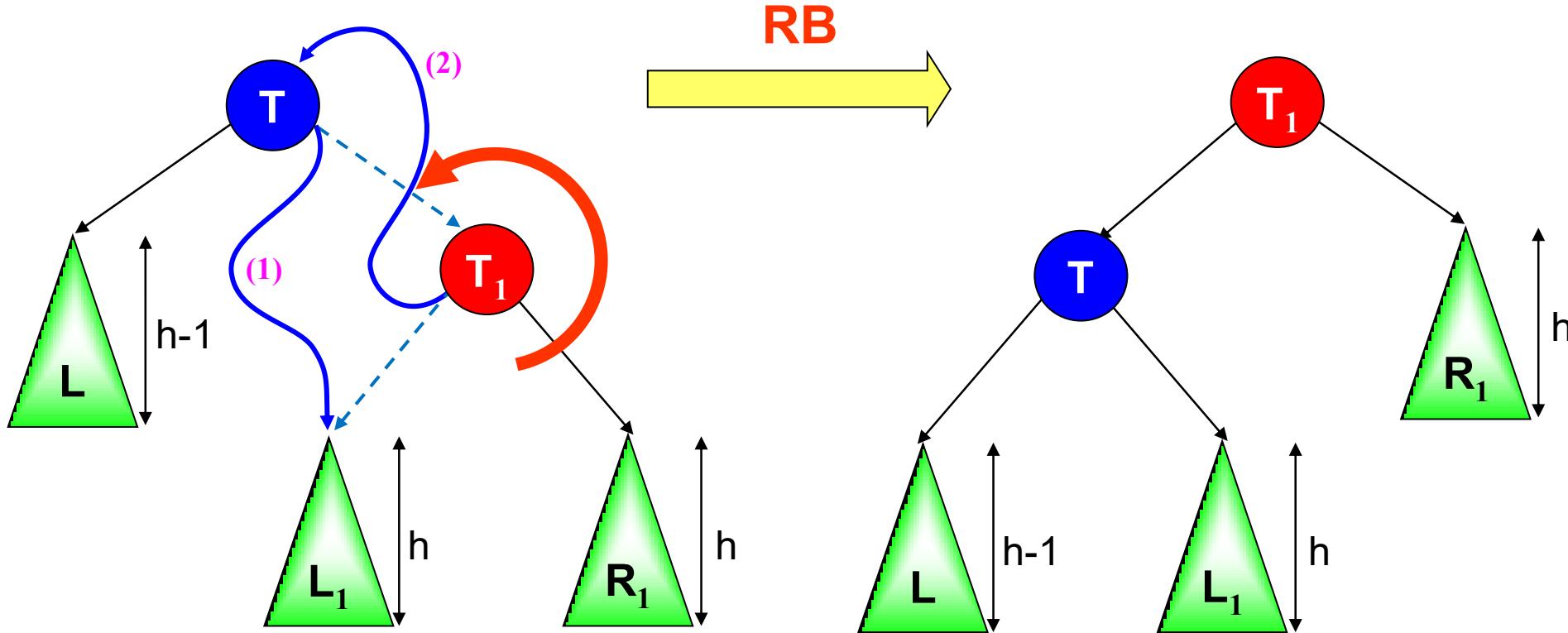


$T \rightarrow \text{Left} = T_1 \rightarrow \text{Right}; (1)$
 $T_1 \rightarrow \text{Right} = T; (2)$

Cài đặt cân bằng lại cho trường hợp 5

Phần cài đặt cho trường hợp **LB** này được cân bằng lại hoàn toàn giống với **LL**.

Cân bằng lại trường hợp 6



$T \rightarrow \text{Right} = T_1 \rightarrow \text{Left}; \quad (1)$
 $T_1 \rightarrow \text{Left} = T; \quad (2)$

Cài đặt cân bằng lại cho trường hợp 6

Phần cài đặt cho trường hợp **RB** (**R**ight **B**alance) này
được cân bằng lại hoàn toàn giống với **RR** (**R**ight **R**ight).

Cài đặt cân bằng lại cây AVL khi lệch trái

```
int balanceLeft(AVLNode* &T)
{//khi cây T lệch bên trái cần cân bằng Lại
    AVLNode* T1=T→Left;
    switch(T1→balFactor)
    {
        case LH: LL(T); return 2;
        case EH: LL(T); return 1;
        case RH: LR(T); return 2;
    }
    return 0; //Trường balance bị sai
}
```

Cài đặt cân bằng lại cây AVL khi lệch phải

```
int balanceRight(AVLNode* &T)
{//khi cây T lệch bên phải cần cân bằng Lại
    AVLNode* T1=T→Right;
    switch(T1→balFactor)
    {
        case LH: RL(T);    return 2;
        case EH: RR(T);   return 1;
        case RH: RR(T);   return 2;
    }
    return 0; //Trường balance bị sai
}
```

Thêm 1 nút vào cây AVL

- ❖ Thêm bình thường như trường hợp cây NPTK
- ❖ Nếu cây tăng trưởng chiều cao thì:
 - Lần ngược về gốc để phát hiện nút bị mất cân bằng.
 - Tiến hành cân bằng lại nút đó bằng thao tác cân bằng thích hợp.
- ❖ Việc cân bằng lại chỉ cần thực hiện 1 lần nơi mất cân bằng
- ❖ **Code:** Xem giáo trình lý thuyết

Hủy 1 nút ra khỏi cây AVL

- ❖ Hủy bình thường như trường hợp cây NPTK
- ❖ Nếu cây giảm chiều cao:
 - Lần ngược về gốc để phát hiện nút bị mất cân bằng.
 - Tiến hành cân bằng lại nút đó bằng thao tác cân bằng thích hợp.
 - Tiếp tục lần ngược lên nút cha...
- ❖ Việc cân bằng lại có thể lan truyền lên tận gốc.
- ❖ **Code:** Xem giáo trình lý thuyết

Lưu ý

- ❖ 2 trường hợp bị mất cân bằng LR, RL: *thì lấy cháu thay cho ông nội.*
- ❖ 4 trường hợp còn lại: LL, LB, RR, RB: *thì lấy con thay cho cha.*

Nội dung

- Cấu trúc cây
- Cây nhị phân
- Cây nhị phân tìm kiếm
- Duyệt cây
- Cây AVL
- **Cây đỏ đen**
- Cây B-Tree

Demo

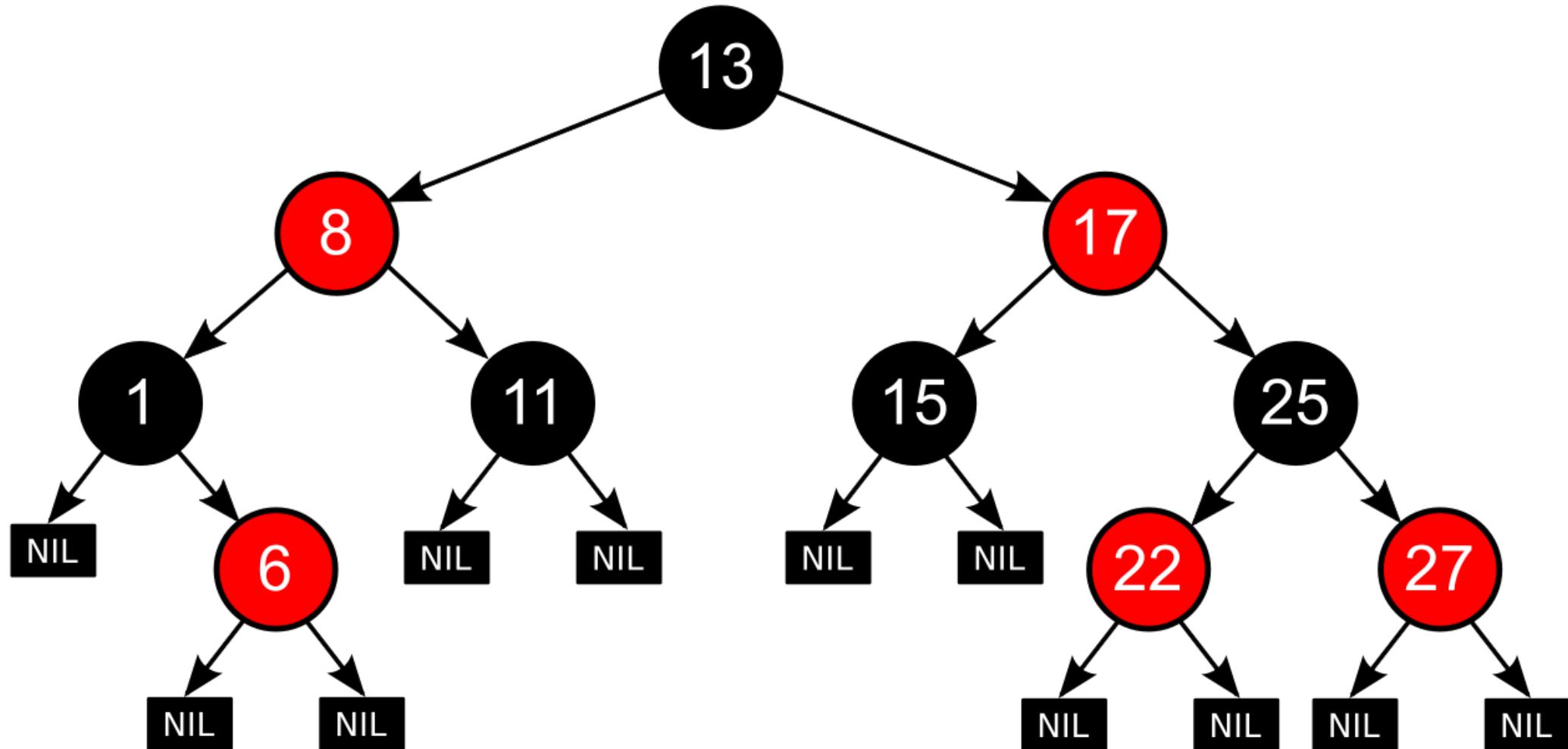
<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

Định nghĩa cây đỏ đen

Cây đỏ-đen là một nhị phân tìm kiếm (BST) thỏa các điều kiện sau:

1. Mọi node đều được tô màu đỏ hoặc màu đen.
2. Node gốc luôn đen.
3. Node lá (NIL) luôn đen.
4. Cả hai con của mọi nút đỏ là đen. (và suy ra mọi nút đỏ có nút cha là đen)
5. Tất cả các đường đi từ một nút bất kỳ tới các lá có số nút đen bằng nhau. Số các nút đen trên một đường đi từ gốc tới mỗi lá được gọi là độ dài đen của đường đi đó.

Cây đỏ đen



Chèn node

- ❖ Phép chèn bắt đầu bổ sung một nút như BST và gán cho nó màu đỏ. Bảo toàn tính chất đỏ đen từ các nút lân cận với nút mới bổ sung.
- ❖ Thuật ngữ *nút chú bác* sẽ dùng để chỉ nút anh (hoặc em) với nút cha của nút đó như trong cây phả hệ.
- ❖ **N** sẽ dùng để chỉ nút đang chèn vào, **P** chỉ nút cha của **N**, **G** chỉ ông của **N**, và **U** chỉ chú bác của **N**.

Chèn node

❖ Node chú bác và node ông:

```
struct node *grandparent(struct node *n) {  
    return n->parent->parent;  
}  
  
struct node *uncle(struct node *n) {  
    if (n->parent == grandparent(n)->left)  
        return grandparent(n)->right;  
    else  
        return grandparent(n)->left;  
}
```

Chèn node

❖ **TH1:** Chèn **N** tại gốc. Trong trường hợp này, gán lại màu đen cho **N**, để bảo bảo tính chất gốc là đen. Vì chỉ bổ sung một nút, Tính chất 4 được bảo đảm vì mọi đường đi chỉ có một nút.

Chèn node

```
void insert_case1(struct node *n) {  
    if (n->parent == NULL)  
        n->color = BLACK;  
    else  
        insert_case2(n);  
}
```

Chèn node

❖ **TH2:** Nút cha P của nút mới thêm là đen, khi đó Tính chất 4 (Cả hai nút con của nút đỏ là đen) không bị vi phạm vì nút mới thêm có hai con là "null" là đen. Tính chất 5 cũng không vi phạm vì nút mới thêm là đỏ không ảnh hưởng tới số nút đen trên tất cả đường đi.

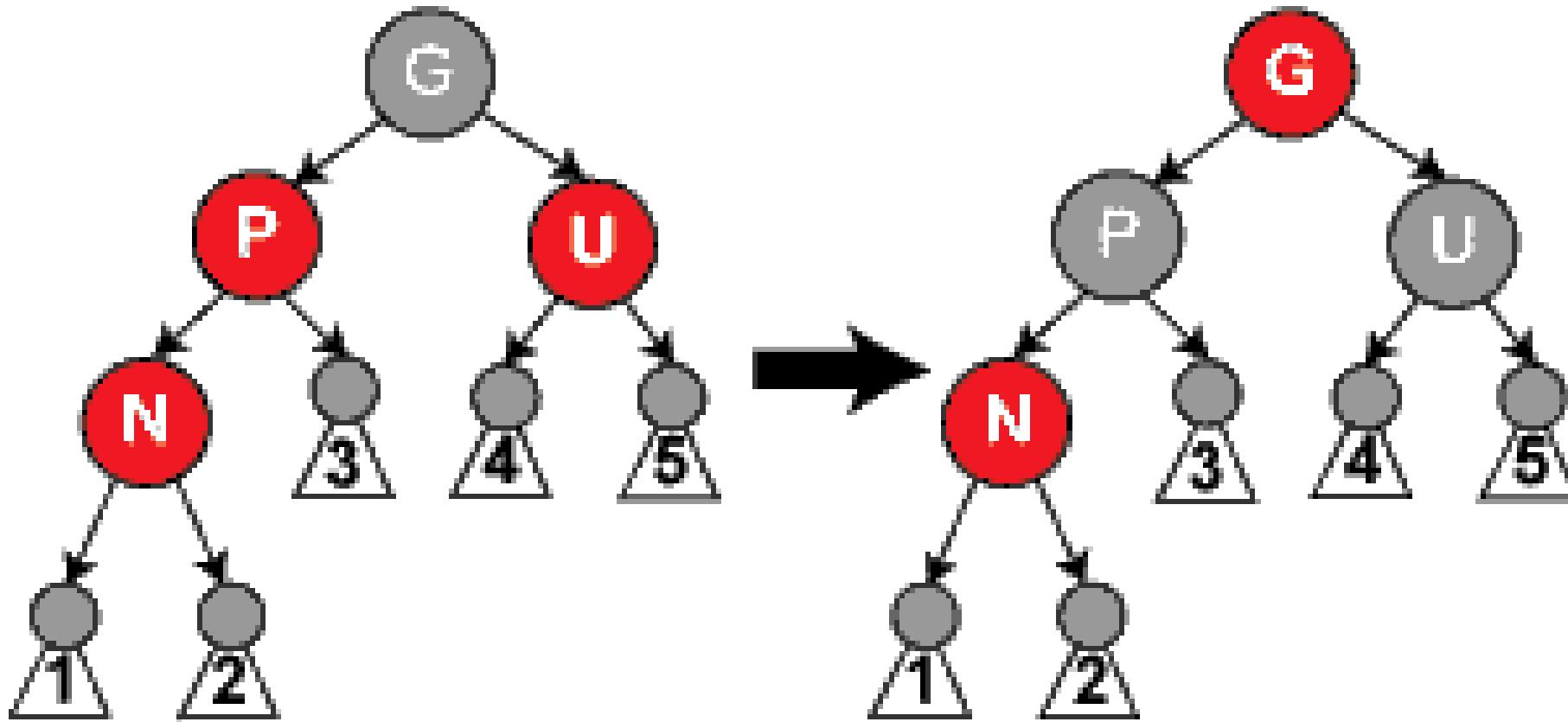
Chèn node

```
void insert_case2(struct node *n) {  
    if (n->parent->color == BLACK)  
        return; /* Tree is still valid */  
    else  
        insert_case3(n);  
}
```

Chèn node

- ❖ **TH3:** Cả cha **P** và bác **U** là đỏ, thì đổi cả hai thành đen còn **G** thành đỏ (để bảo toàn tính chất 5) .Khi đó nút mới **N** có cha đen. Vì đường đi bất kỳ đi qua cha và bác của "**N**" phải đi qua ông của **N** nên số các nút đen trên đường đi này không thay đổi.
- ❖ Để sửa chữa trường hợp này gọi một thủ tục đệ quy trên **G** từ trường hợp 1.

Chèn node



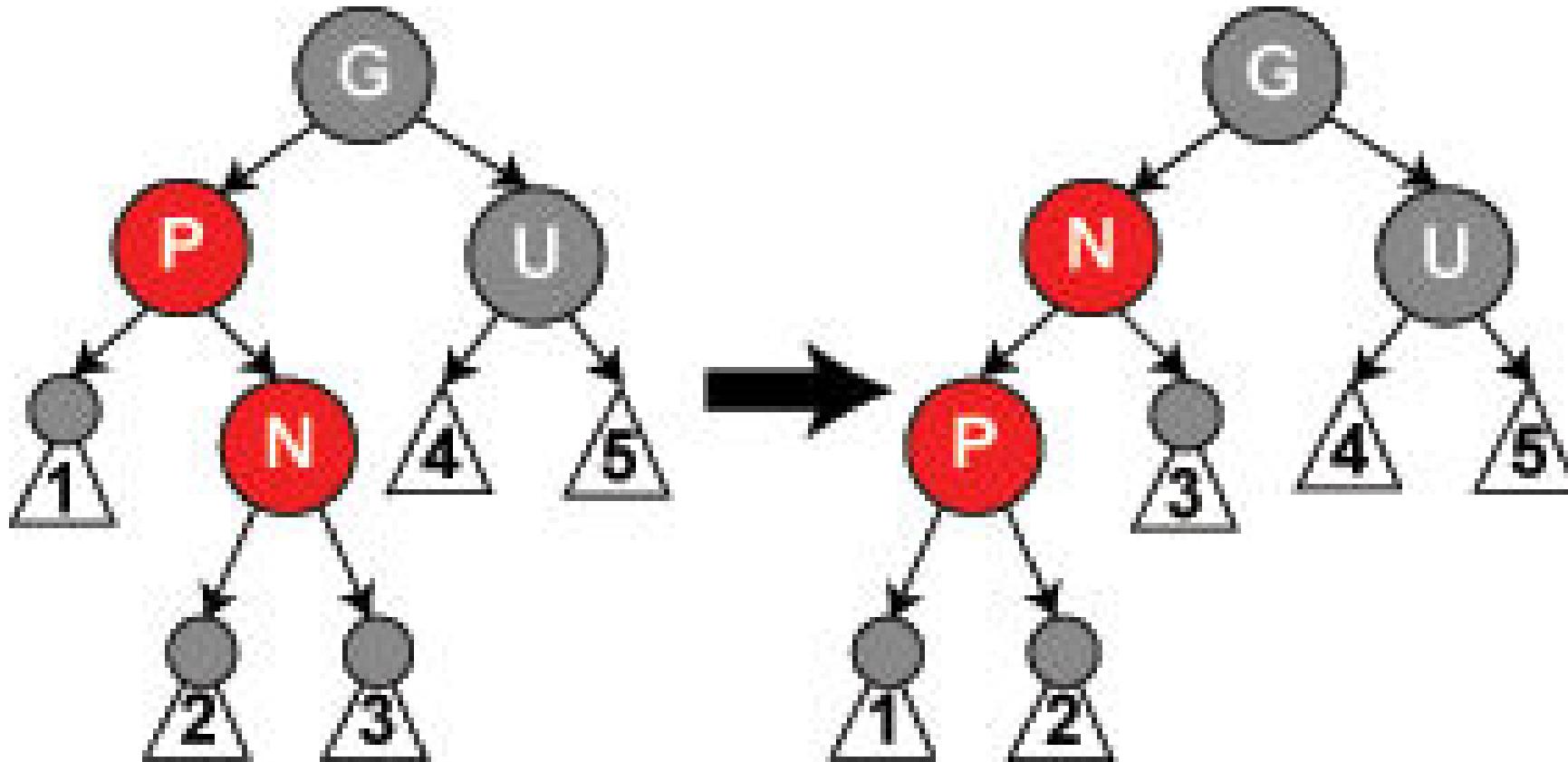
Chèn node

```
void insert_case3(struct node *n) {  
    if (uncle(n) != NULL && uncle(n)->color == RED)  
    {  
        n->parent->color = BLACK;  
        uncle(n)->color = BLACK;  
        grandparent(n)->color = RED;  
        insert_case1(grandparent(n));  
    } else  
        insert_case4(n);  
}
```

Chèn node

- ❖ **TH4:** Nút cha **P** là đỏ nhưng nút chú bác **U** là đen, nút mới **N** là con phải của nút **P**, và **P** là con trái của nút **G**.
- ❖ Thực hiện quay trái chuyển đổi vai trò của nút mới **N** và nút cha **P** do đó định dạng lại nút **P** bằng Trường hợp 5 (đổi vai trò **N** và **P**) vì tính chất 4 bị vi phạm (Cả hai con của nút đỏ là đen).
- ❖ Phép quay cũng làm thay đổi một vài đường đi (các đường đi qua cây con nhãn "1") phải đi qua thêm nút mới **N**, nhưng vì **N** là đỏ nên không làm chúng vi phạm tính chất 5

Chèn node



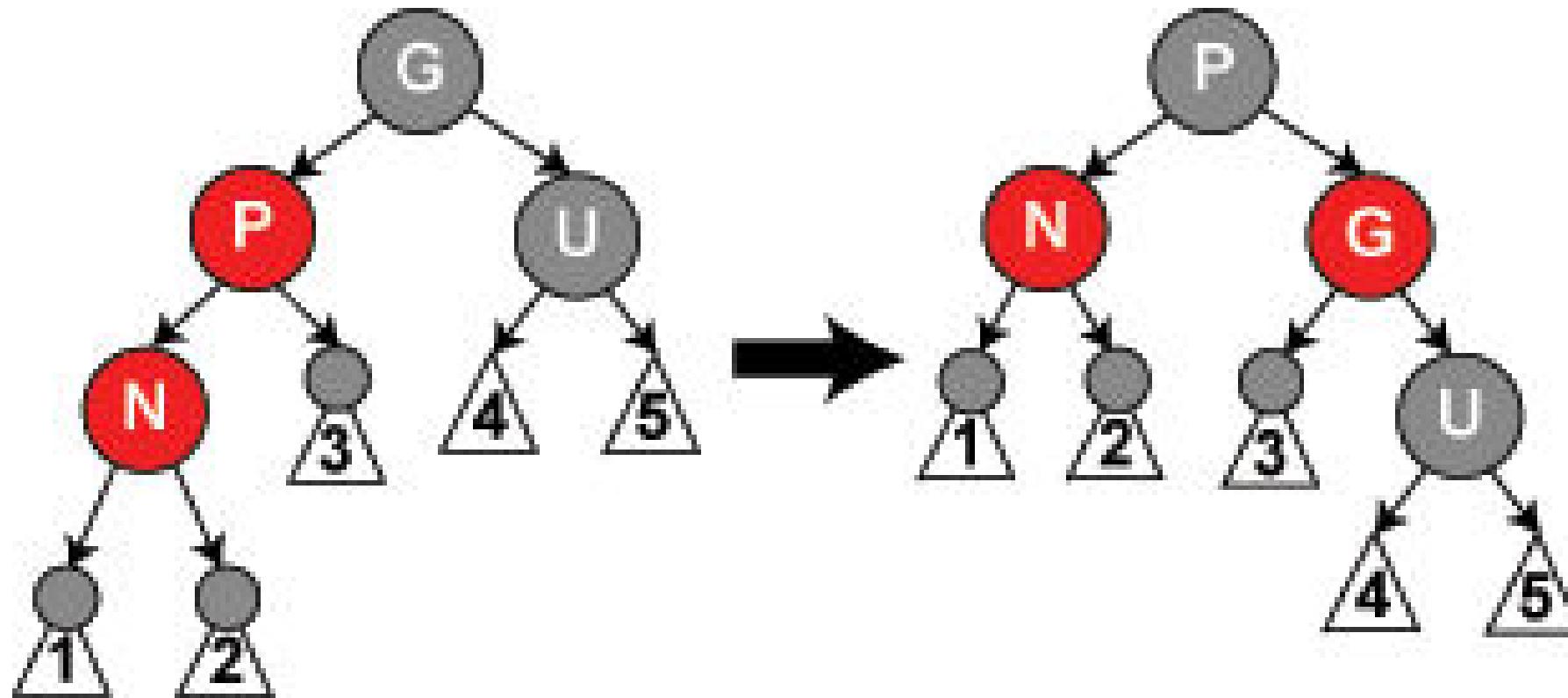
Chèn node

```
void insert_case4(struct node *n) {  
    if(n==n->parent->right && n->parent==grandparent(n)->left)  
    {  
        rotate_left(n->parent);  
        n = n->left;  
    } else if(n==n->parent->left &&  
              n->parent==grandparent(n)->right) {  
        rotate_right(n->parent);  
        n = n->right;  
    }  
    insert_case5(n);  
}
```

Chèn node

- ❖ **TH5:** Nút cha **P** là đỏ nhưng nút bác **U** là đen, nút mới **N** là con trái của nút **P**, và **P** là con trái của nút ông **G**.
- ❖ Phép quay phải trên nút ông **G** được thực hiện; kết quả của phép quay là trong cây mới nút **P** trở thành cha của cả hai nút **N** và nút **G**. **G** là đen, vì bây giờ nó là con của **P**.
- ❖ Đổi màu của **P** và **G** thì cây thỏa mãn tính chất 4.
- ❖ Tính chất 5 không bị vi phạm vì các đường đi qua **G** trước đây bây giờ đi qua **P**.

Chèn node



Chèn node

```
void insert_case5(struct node *n)
{
    n->parent->color = BLACK;
    grandparent(n)->color = RED;
    if (n==n->parent->left && n->parent==grandparent(n)->left)
    {
        rotate_right(grandparent(n));
    } else {
        rotate_left(grandparent(n));
    }
}
```

Xóa node

- ❖ Hàm tìm người anh em của N:

```
struct node *sibling(struct node *n) {  
    if (n == n->parent->left)  
        return n->parent->right;  
    else  
        return n->parent->left;  
}
```

Xóa node

- ❖ Hàm thay thế node con vào vị trí node bị xóa trên cây

```
void delete_one_child(struct node *n) {  
    struct node *child = is_leaf(n->right) ? n->left : n->right;  
    replace_node(n, child);  
    if (n->color == BLACK) {  
        if (child->color == RED)  
            child->color = BLACK;  
        else  
            delete_case1(child);  
    }  
    free(n);  
}
```

Xóa node

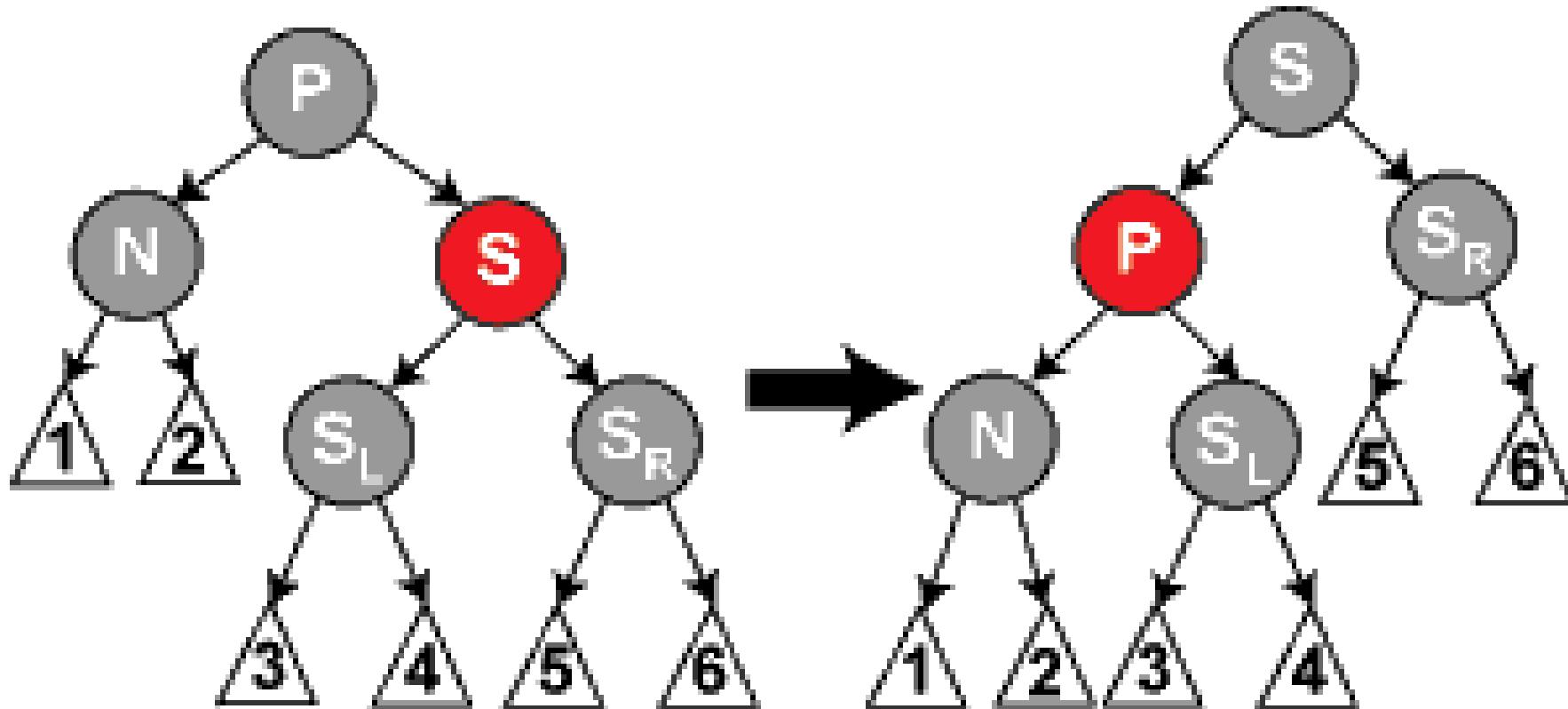
❖ **TH1:** N là gốc mới. Ta dừng lại. Ta đã giải phóng một nút đen khỏi mọi đường đi và gốc mới lại là đen. Không tính chất nào bị vi phạm.

```
void delete_case1(struct node *n) {  
    if (n->parent == NULL)  
        return;  
    else  
        delete_case2(n);  
}
```

Xóa node

- ❖ **TH2:** **S** là đỏ. Tráo đổi màu của **P** và **S**, và sau đó quay trái tai **P**, **S** trở thành nút ông của **N**.
- ❖ **P** có màu đen và có một con màu đỏ. Tất cả các đường đi có số các nút đen giống nhau
- ❖ **N** có một anh em màu đen và cha màu đỏ, chúng ta có thể tiếp tục với các trường hợp 4, 5, hoặc 6. (anh em mới của nó là đen ví chỉ có một con của nút đỏ **S**.)
- ❖ Trong các trường hợp sau ta sẽ gọi anh em mới của **N'** là **S**.

Xóa node



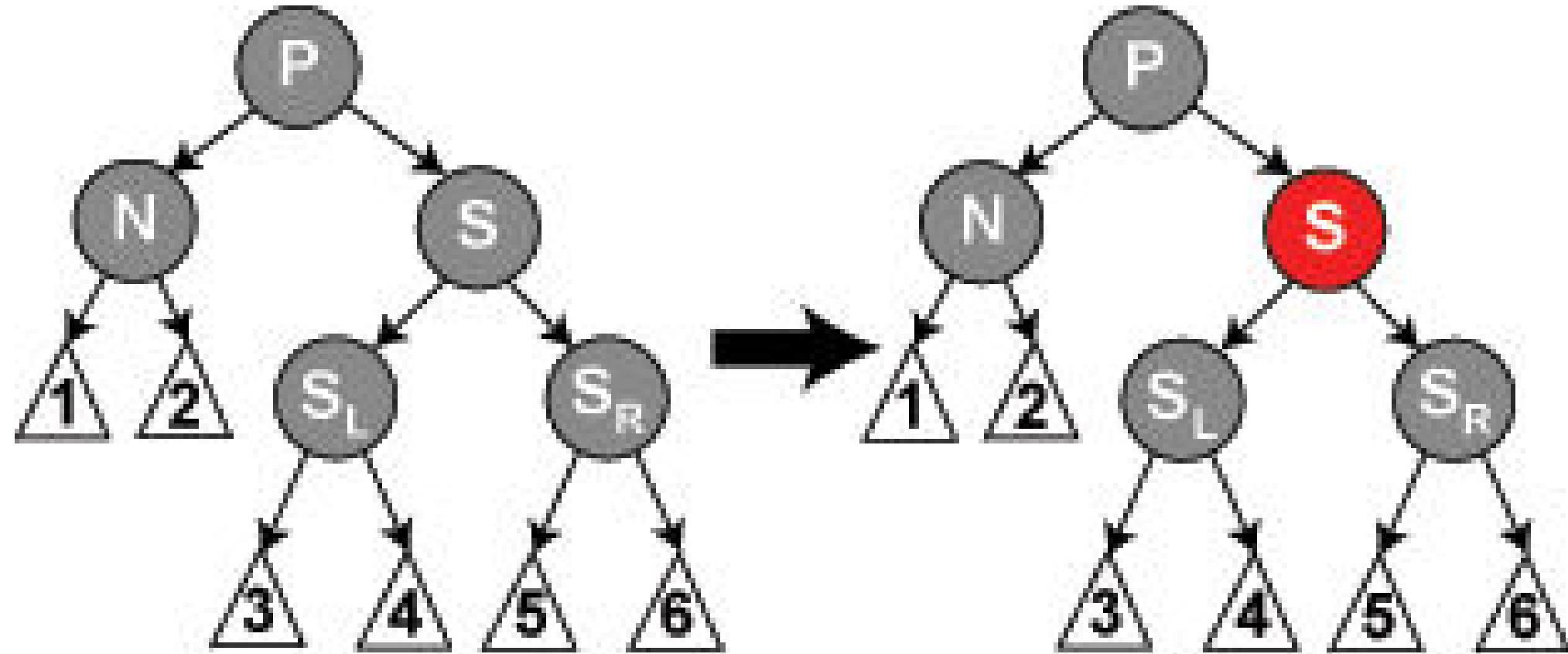
Xóa node

```
void delete_case2(struct node *n) {
    if (sibling(n)->color == RED) {
        n->parent->color = RED;
        sibling(n)->color = BLACK;
        if (n == n->parent->left)
            rotate_left(n->parent);
        else
            rotate_right(n->parent);
    }
    delete_case3(n);
}
```

Xóa node

- ❖ **TH3:** **P**, **S**, và các con của **S** là đen. Gán lại cho **S** màu đỏ. Kết quả là mọi đường đi qua **S**, (tất nhiên chúng không qua **N**, có ít hơn một nút đen).
- ❖ Vì việc xóa đi cha trước đây của **N'** làm tất cả các đường đi qua **N** bớt đi một nút đen, nên chúng bằng nhau.
- ❖ Tuy nhiên tất cả các đường đi qua **P** bây giờ có ít hơn một nút đen so với các đường không qua **P**, do đó Tính chất 5 (Tất cả các đường đi từ gốc tới các nút lá có cùng số nút đen) sẽ bị vi phạm. Để sửa chữa tái cân bằng tại **P**, bắt đầu từ trường hợp 1

Xóa node



Xóa node

```
void delete_case3(struct node *n) {  
    if (n->parent->color == BLACK && sibling(n)->color == BLACK && sibling(n)->left->color == BLACK && sibling(n)->right->color == BLACK) {  
        sibling(n)->color = RED;  
        delete_case1(n->parent);  
    }  
    else  
        delete_case4(n);  
}
```

Nội dung

- Cấu trúc cây
- Cây nhị phân
- Cây nhị phân tìm kiếm
- Duyệt cây
- Cây AVL
- Cây đỏ đen
- **Cây B-Tree**

Demo

<https://www.cs.usfca.edu/~galles/visualization/BTree.html>

NỘI DUNG

1. Giới thiệu
2. Khái niệm
3. Đặc điểm và cấu trúc
4. Chèn phần tử vào cây
5. Xóa phần tử khỏi cây

GIỚI THIỆU

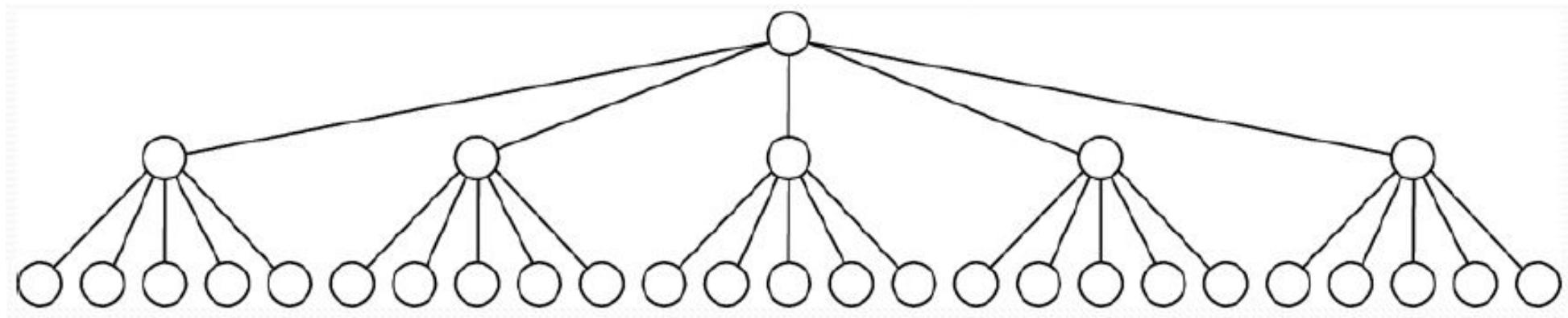
- ❖ Cây là một cách tiếp cận hoàn chỉnh để tổ chức dữ liệu trong bộ nhớ. Vậy cây có thể làm việc tốt với hệ thống tập tin hay không?
- ❖ B-Tree là cấu trúc dữ liệu phù hợp cho việc lưu trữ ngoài do R.Bayer và E. M. McCreight đưa ra năm 1972.

GIỚI THIỆU

- ❖ B-Tree là cây tìm kiếm tự cân bằng. Trong hầu hết các cây tìm kiếm tự cân bằng khác (như AVL và Red Black Trees), giả định rằng mọi thứ đều nằm trong bộ nhớ chính.
- ❖ Để hiểu được việc sử dụng B-Tree, chúng ta phải nghĩ đến số lượng dữ liệu khổng lồ mà không thể lưu trong bộ nhớ chính. Khi số lượng keys lớn, dữ liệu được đọc từ hard disk dưới dạng khối (blocks). Thời gian đọc dữ liệu từ hard disk rất lâu so với thời gian truy xuất bộ nhớ chính → sử dụng B-Tree là giảm số lần truy cập đĩa.
- ❖ Hầu hết các hoạt động của cây (tìm kiếm, chèn, xóa, max, min, ..etc) yêu cầu truy cập đĩa $O(h)$ chính là chiều cao của B-Tree.
- ❖ Chiều cao của B-Tree được hạn chế tối đa bằng việc bố trí nhiều nhất keys tại các node (kích thước node tương đương kích thước block). Vì khả năng hạn chế h, tổng số lần đĩa truy cập cho hầu hết các hoạt động được giảm đáng kể so với cây AVL, Red Black Tree, ..etc.

Cây nhiều nhánh: M-Phân

- ❖ Mỗi node có tối đa M node con (cây con)
- ❖ Một cây M-Phân đầy đủ có chiều cao $\log_M N$
- ❖ Ví dụ: Cây 5-Phân đầy đủ như sau



Định nghĩa

- ❖ Một B-Tree bậc M là cây **M-Phân** là cây thỏa:
 - ❖ Mỗi node có **tối đa M** cây con, và có **M-1** khóa.
 - ❖ Các khóa trong mỗi node (cây con) được **sắp xếp tăng**.
 - ❖ Các khóa trong cây con thứ i đều nhỏ hơn khóa i.
 - ❖ Các khóa trong cây con thứ (i+1) đều lớn hơn khóa i.

Tính chất

❖ Cây **M-Phân** tìm kiếm có các tính chất sau:

- ❖ Tất cả node lá có cùng mức.
- ❖ Tất cả các node trung gian (*trừ node gốc*) có nhiều nhất M cây con và có ít nhất $M/2$ cây con (khác rỗng).
- ❖ Mỗi node hoặc là node lá hoặc có $k+1$ cây con (k là số khoá của node này).
- ❖ Node gốc có nhiều nhất M cây con hoặc có thể có 2 cây con (Node gốc có 1 khoá và không phải là node lá) hoặc không chứa cây con nào (Node gốc có 1 khoá và cũng là node lá).

Tính chất

❖ Cây **M-Phân** tìm kiếm có các tính chất sau:

- ❖ Tại thời điểm chèn một nút đầy đủ, cây chia thành hai phần và khóa có giá trị trung bình được chèn tại nút cha.
- ❖ Hoạt động hợp nhất diễn ra khi các nút bị xóa.

Ý nghĩa

- ❖ B-cây là cây cân bằng hoàn toàn.
- ❖ Mỗi node được lấp đầy ít nhất 50%. Các phân tích và thử nghiệm thực tế cho thấy các node của B-cây trong trường hợp bình thường được lấp đầy ~70%
- ❖ B-cây sử dụng số phép truy xuất đĩa tối thiểu cho các thao tác.
- ❖ Thích hợp với việc lưu trữ trên bộ nhớ ngoài.
- ❖ Có thể quản lý số phần tử rất lớn.

Ý nghĩa

- ❖ Hạn chế số thao tác đọc mỗi lần tìm kiếm trên cây.
- ❖ Thích hợp cho việc tìm kiếm trên bộ nhớ ngoài.
- ❖ Chiều cao h của cây $\sim \log_M N$, N là số nút của cây.

Cụ thể: Nếu N là số khoá ($N \geq 1$), M là bậc của cây ($M > 2$) thì:

$$h \leq \log_M \frac{(N + 1)}{2}$$

- ❖ Tăng M chiều cao cây giảm rất nhanh.

Chèn node vào cây

Ý tưởng: Tìm vị trí khóa có thể thêm vào cây. Việc tìm kiếm sẽ kết thúc tại một nút lá. Khóa mới sẽ được thêm vào nút lá:

1. Nếu chưa đầy → Việc thêm hoàn tất.
2. Nếu đầy → Phân đôi nút lá cần thêm:

- ❖ **Tách nút lá ra làm hai nút cạnh nhau trong cùng một mức.**
- ❖ **Chuyển phần tử giữa lên nút cha.**

Quá trình phân đôi các nút có thể được lan truyền ngược về đến nút gốc và kết thúc khi có một nút cha nào đó cần được thêm một khóa từ dưới lên mà chưa đầy.

Ví dụ

- ❖ Cho B-Tree bậc 4 rỗng.
- ❖ Hãy xây dựng B-Tree theo thứ tự từ trái sang phải cho dãy số sau:

1	12	8	2	25	5	14	28	17	7	52	16	48	68	3	26	29	53	55	45
---	----	---	---	----	---	----	----	----	---	----	----	----	----	---	----	----	----	----	----

Ví dụ

1	12	8	2	25	5	14	28	17	7	52	16	48	68	3	26	29	53	55	45
---	----	---	---	----	---	----	----	----	---	----	----	----	----	---	----	----	----	----	----

Chèn 1:

1

Chèn 12:

1	12
---	----

Chèn 8:

1	8	12
---	---	----

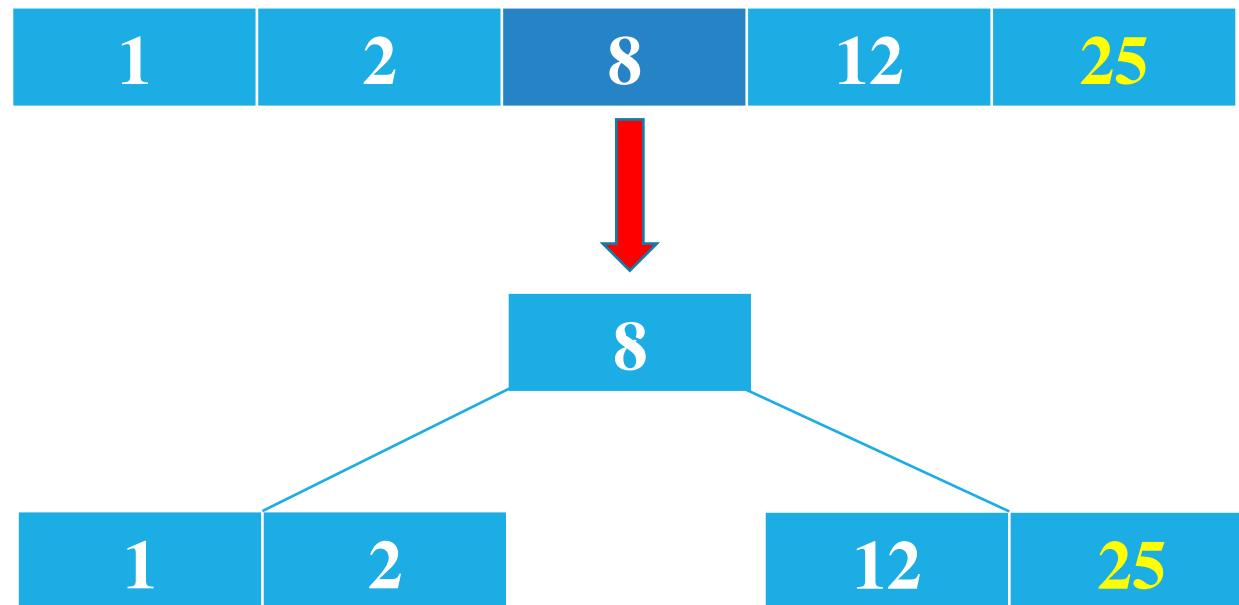
Chèn 2:

1	2	8	12
---	---	---	----

Ví dụ

1 | 12 | 8 | 2 | 25 | 5 | 14 | 28 | 17 | 7 | 52 | 16 | 48 | 68 | 3 | 26 | 29 | 53 | 55 | 45

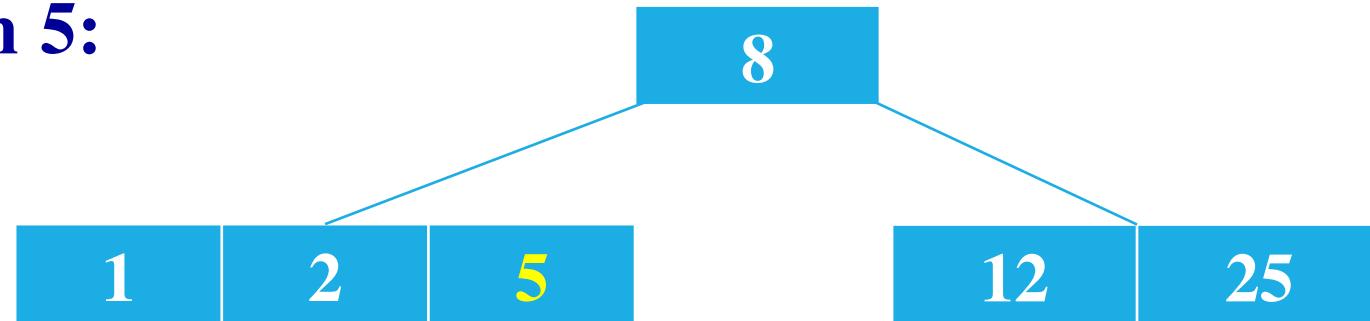
- ❖ Do nút gốc đã đầy (4 phần tử) → Chèn 25 vào nút gốc sẽ tách nút gốc thành 2 nút và đưa khóa ở giữa lên trên tạo thành nút gốc mới.



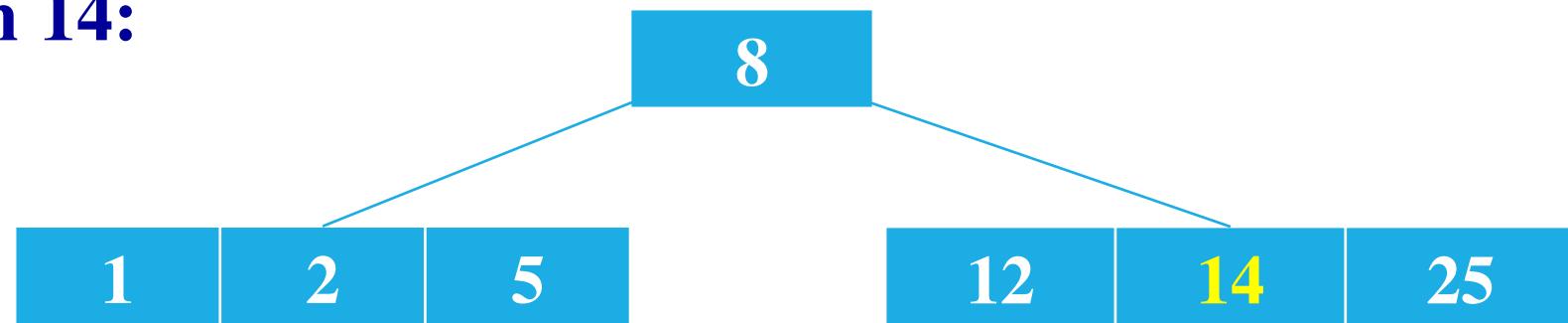
Ví dụ

1 | 12 | 8 | 2 | 25 | 5 | 14 | 28 | 17 | 7 | 52 | 16 | 48 | 68 | 3 | 26 | 29 | 53 | 55 | 45

❖ Chèn 5:



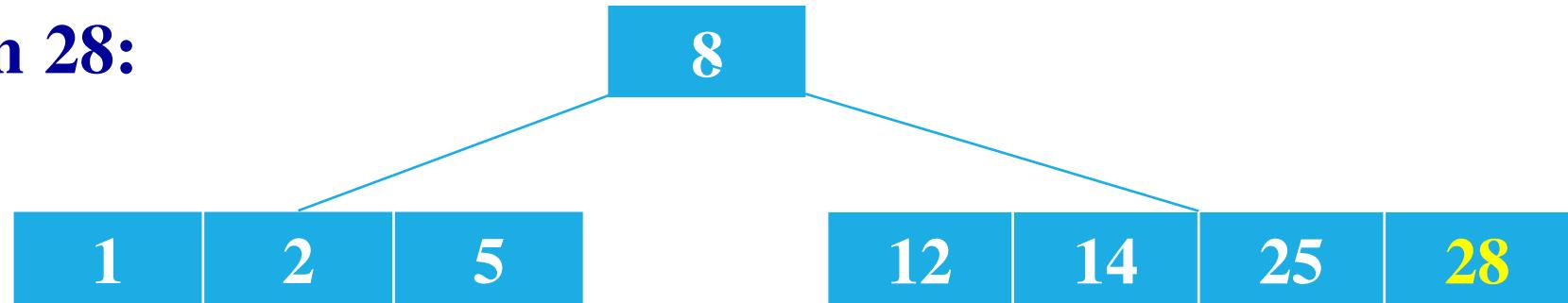
❖ Chèn 14:



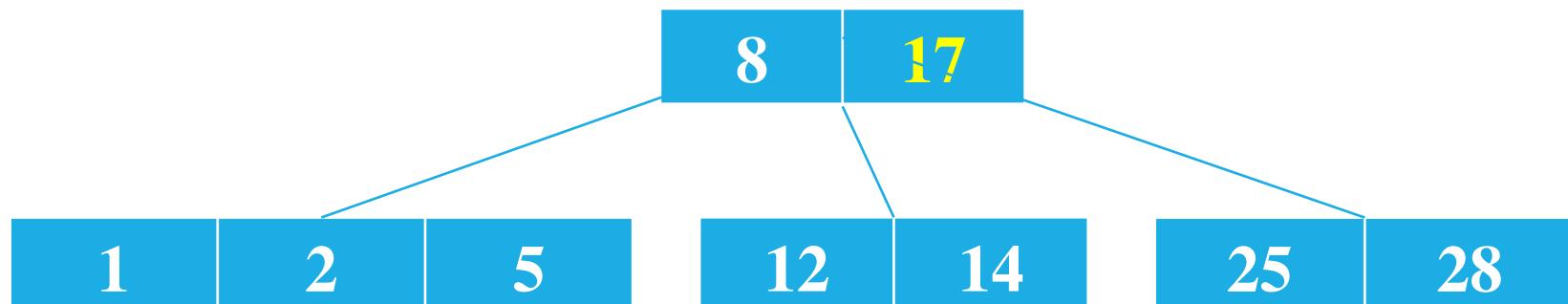
Ví dụ

1 | 12 | 8 | 2 | 25 | 5 | 14 | 28 | 17 | 7 | 52 | 16 | 48 | 68 | 3 | 26 | 29 | 53 | 55 | 45

❖ Chèn 28:



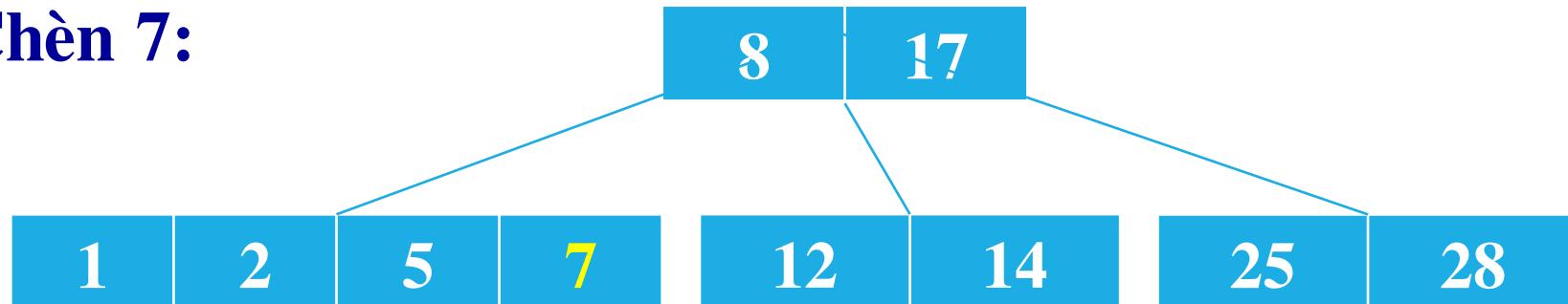
❖ Chèn 17: do nút lá bên phải đã đầy nên phân đôi và đưa nút giữa lên trên nút cha (nút gốc).



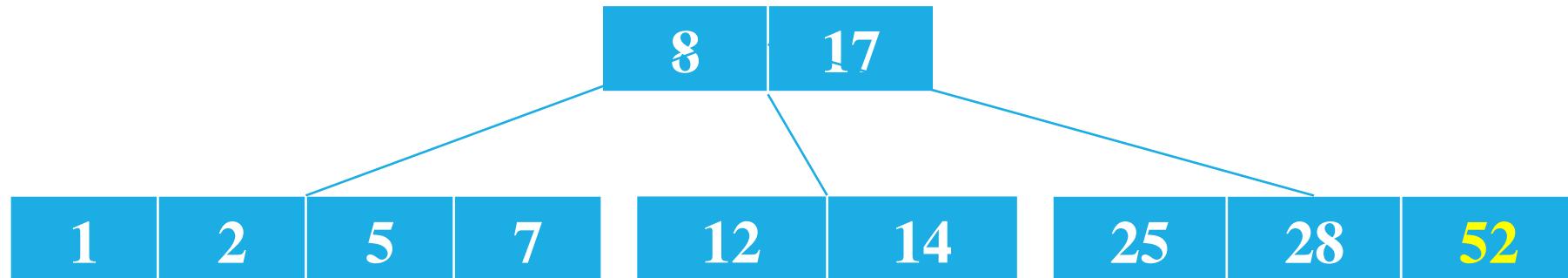
Ví dụ

1 | 12 | 8 | 2 | 25 | 5 | 14 | 28 | 17 | 7 | 52 | 16 | 48 | 68 | 3 | 26 | 29 | 53 | 55 | 45

❖ Chèn 7:



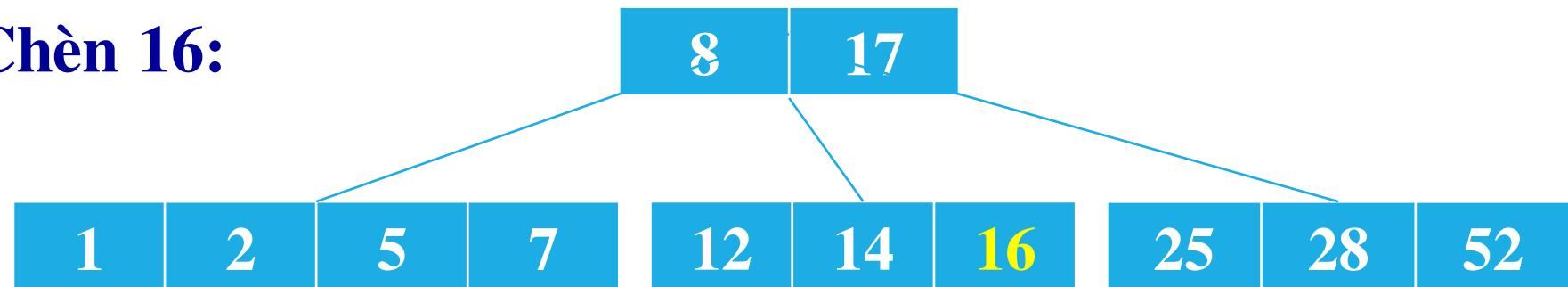
❖ Chèn 52:



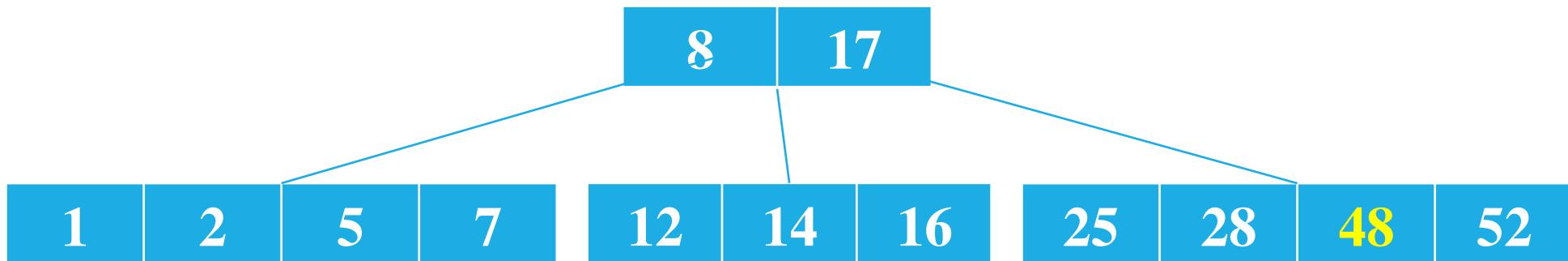
Ví dụ

1 | 12 | 8 | 2 | 25 | 5 | 14 | 28 | 17 | 7 | 52 | 16 | 48 | 68 | 3 | 26 | 29 | 53 | 55 | 45

❖ Chèn 16:



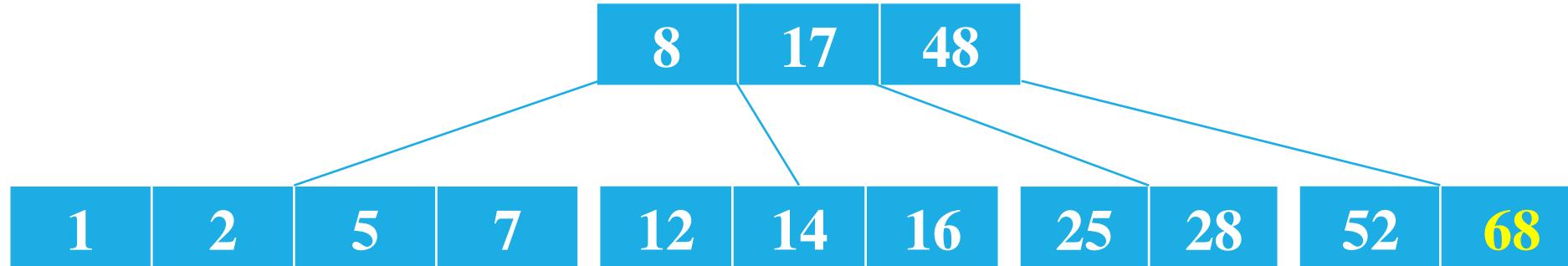
❖ Chèn 48:



Ví dụ

1 | 12 | 8 | 2 | 25 | 5 | 14 | 28 | 17 | 7 | 52 | 16 | 48 | 68 | 3 | 26 | 29 | 53 | 55 | 45

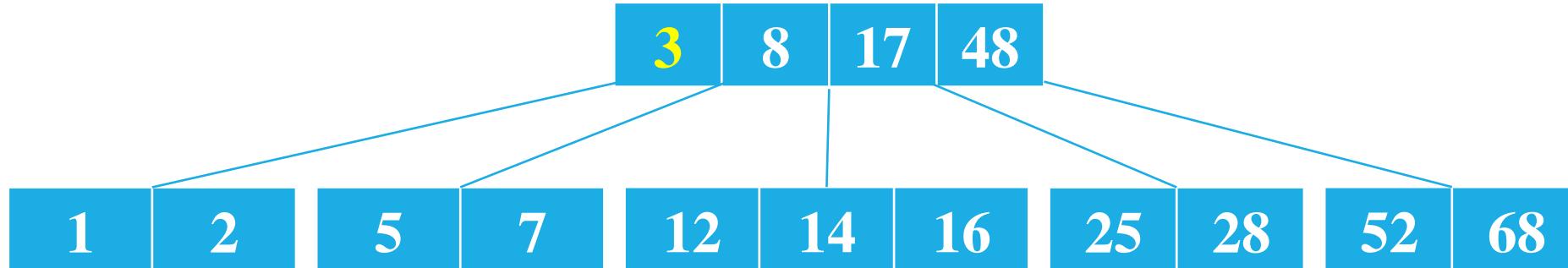
❖ **Chèn 68:** do nút lá bên phải đã đầy nên phân đôi nút lá và đưa nút giữa lên nút cha (nút gốc).



Ví dụ

1 | 12 | 8 | 2 | 25 | 5 | 14 | 28 | 17 | 7 | 52 | 16 | 48 | 68 | 3 | 26 | 29 | 53 | 55 | 45

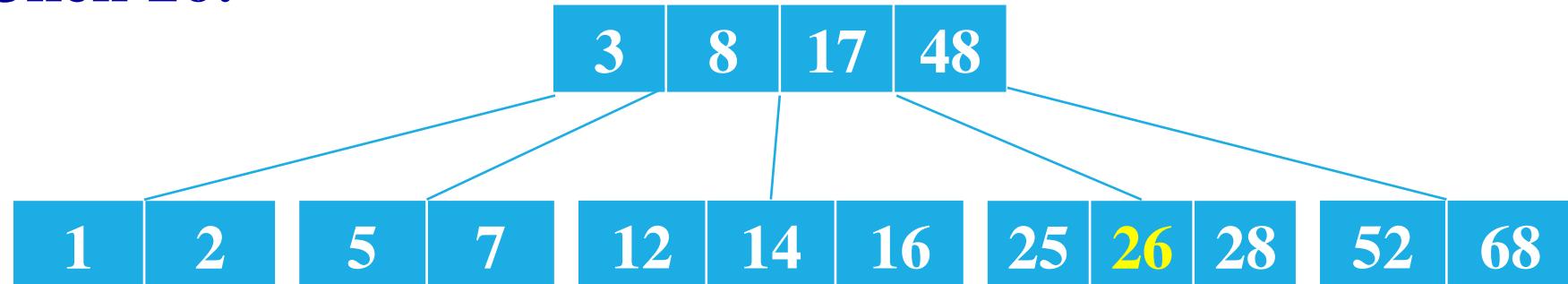
❖ **Chèn 3:** do nút lá bên phải đã đầy nên phân đôi nút lá và đưa nút giữa lên nút cha (nút gốc).



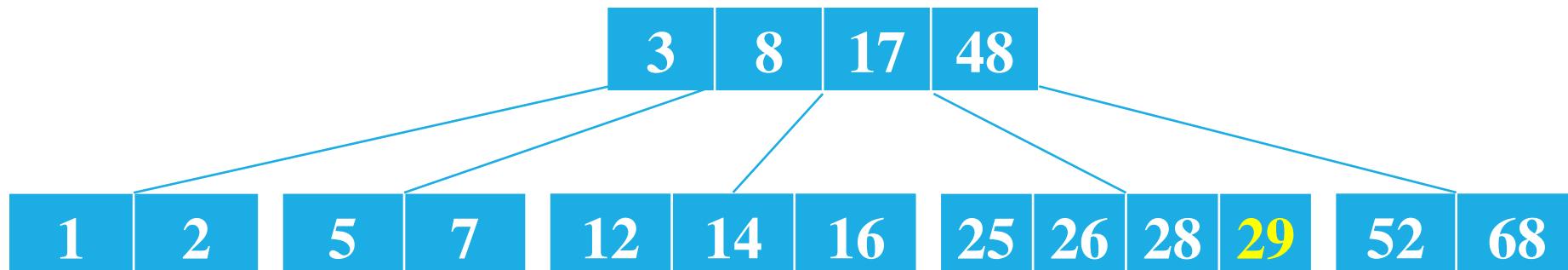
Ví dụ

1 | 12 | 8 | 2 | 25 | 5 | 14 | 28 | 17 | 7 | 52 | 16 | 48 | 68 | 3 | 26 | 29 | 53 | 55 | 45

❖ Chèn 26:



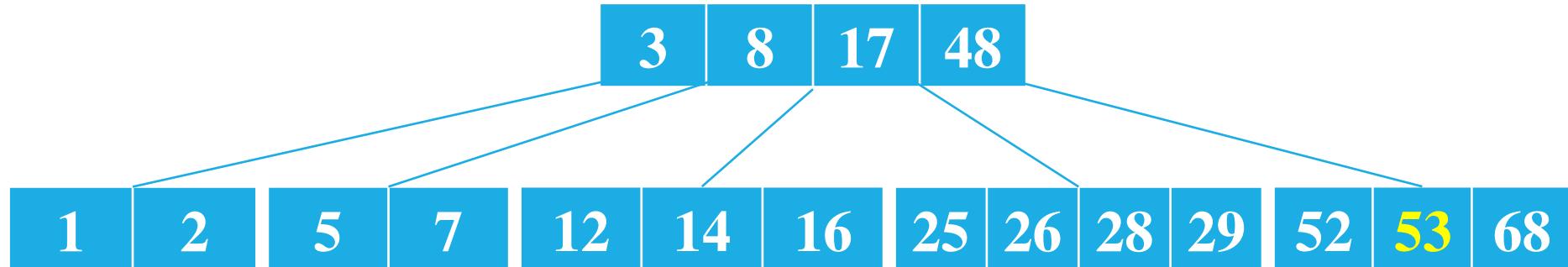
❖ Chèn 29:



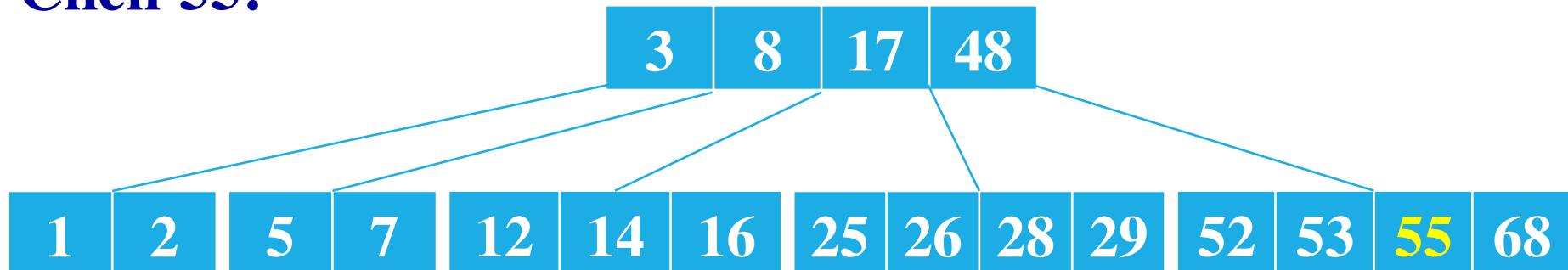
Ví dụ

1	12	8	2	25	5	14	28	17	7	52	16	48	68	3	26	29	53	55	45
---	----	---	---	----	---	----	----	----	---	----	----	----	----	---	----	----	----	----	----

❖ Chèn 53:



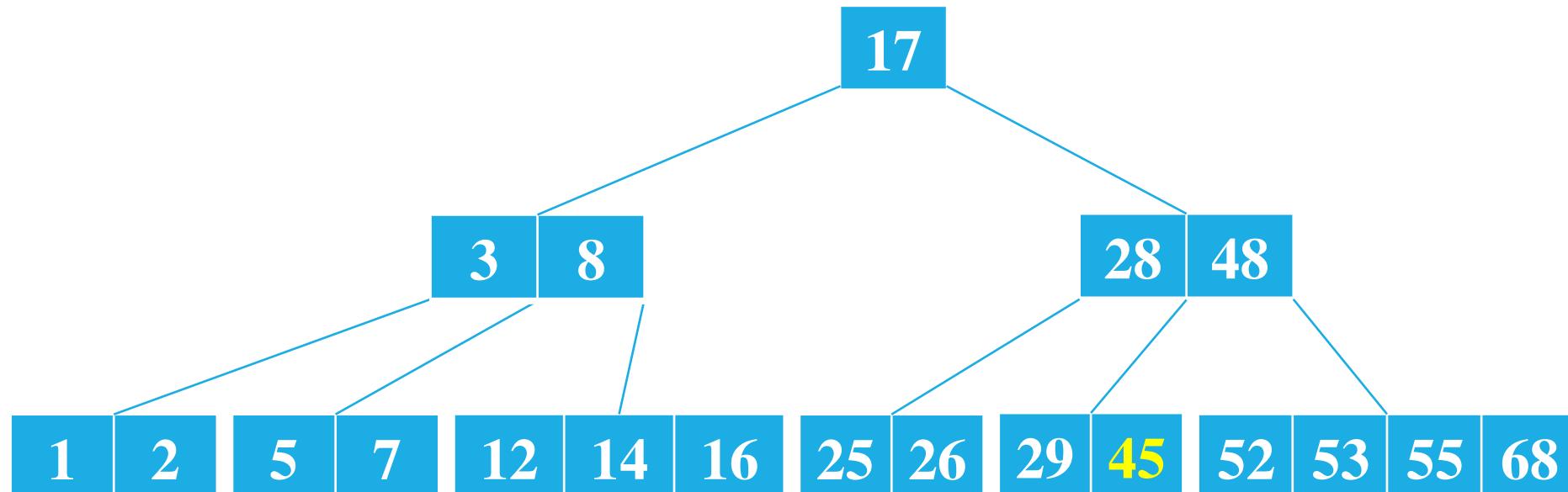
❖ Chèn 55:



Ví dụ

1	12	8	2	25	5	14	28	17	7	52	16	48	68	3	26	29	53	55	45
---	----	---	---	----	---	----	----	----	---	----	----	----	----	---	----	----	----	----	----

❖ **Chèn 45:** do nút lá thứ 3 đầy nên phân đôi và đưa nút giữa lên trên nút cha, nút cha cũng đầy nên phân đôi tiếp nút cha và đưa nút giữa lên trên, tạo thành nút gốc mới.



Bài tập áp dụng

Cho dãy số: 1, 3, 9, 2, 4, 10, 25, 30, 11, 40, 21, 8, 7, 6, 19, 5, 30, 12, 15, 16

Hãy trình bày từng bước vẽ cây B-Tree bậc 4?

Xóa khóa trên cây

1. Nếu khóa cần xóa nằm ở nút lá → Xóa.
2. Nếu khóa cần xóa không nằm ở nút lá → Xóa khóa cần xóa, đưa khóa có giá trị gần nhất từ nút lá lên thay thế cho khóa đã xóa.

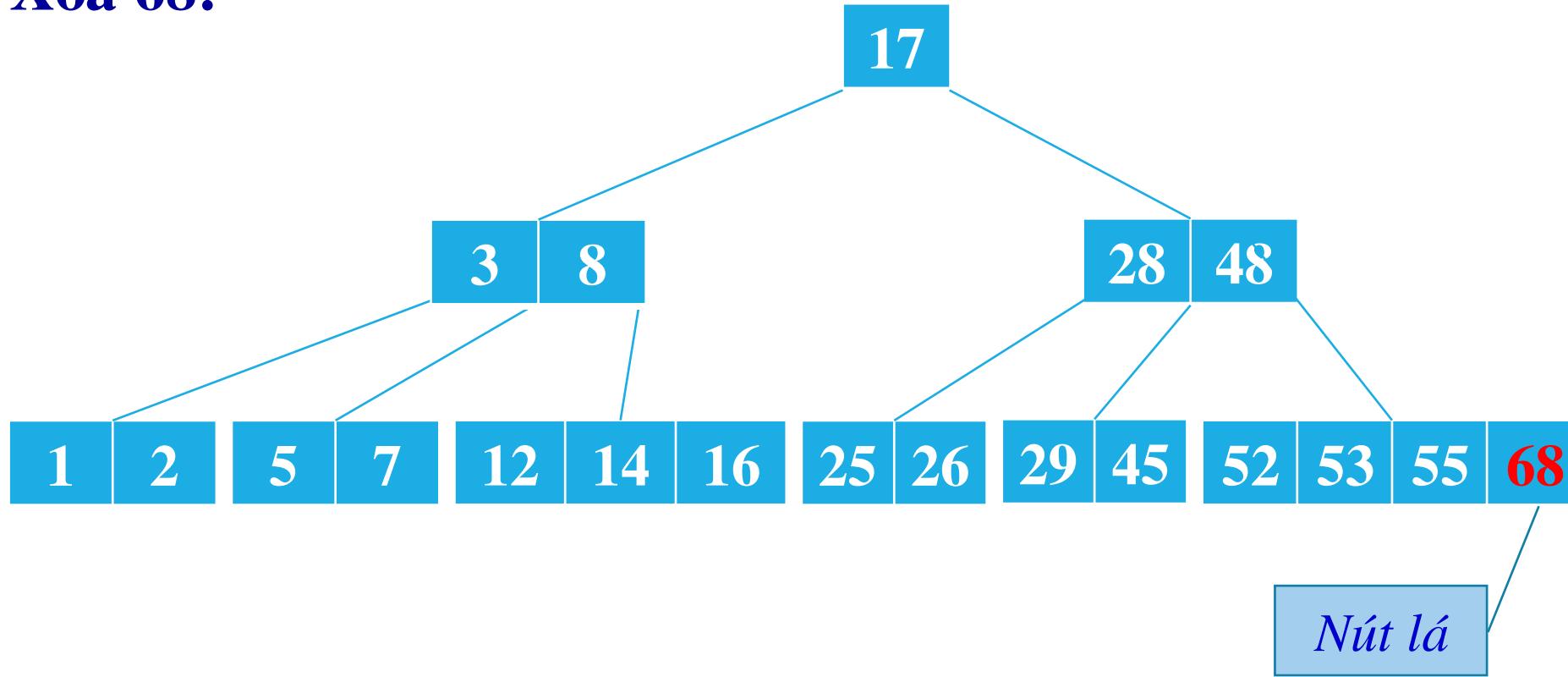
Xóa khóa trên cây

Nếu trong trường hợp (1) hay (2), nút lá còn lại quá ít khóa thì xét các nút anh em kế cận nút đang xét:

3. Nếu một trong các nút anh em kế cận nút đang xét có số lượng khóa nhiều hơn số lượng tối thiểu, đưa một khóa của nút anh em lên nút cha và đưa khóa ở nút cha xuống nút lá có khóa quá ít.
 4. Nếu tất cả các nút anh em kế cận nút đang xét đều có số lượng khóa vừa đủ số lượng tối thiểu, chọn một nút anh em kế cận và hợp nhất nút anh em này với nút lá có khóa quá ít.
- Nếu nút cha trở nên thiếu khóa, lặp lại quá trình này.

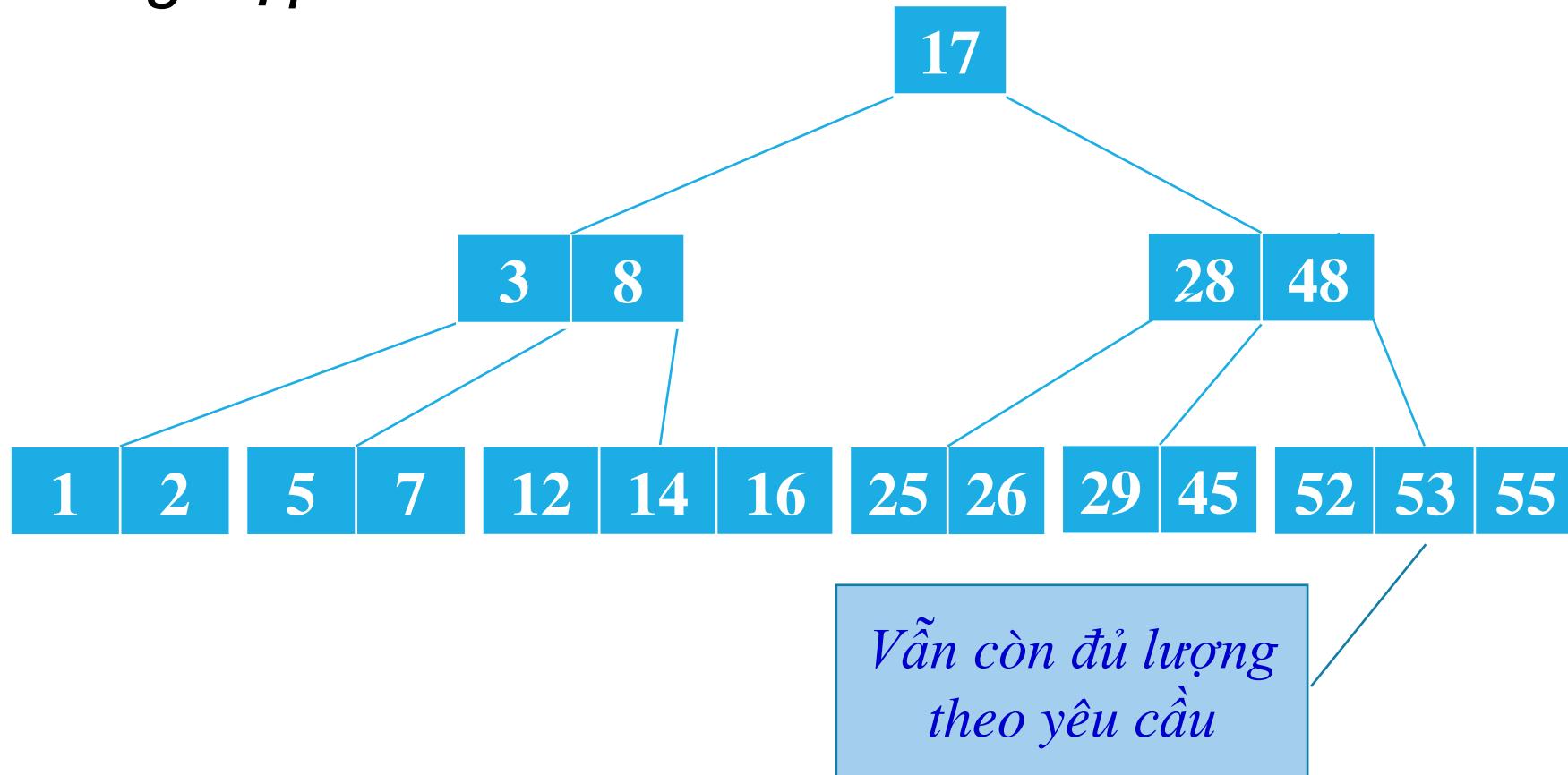
Xóa khóa trên cây

❖ Xóa 68:



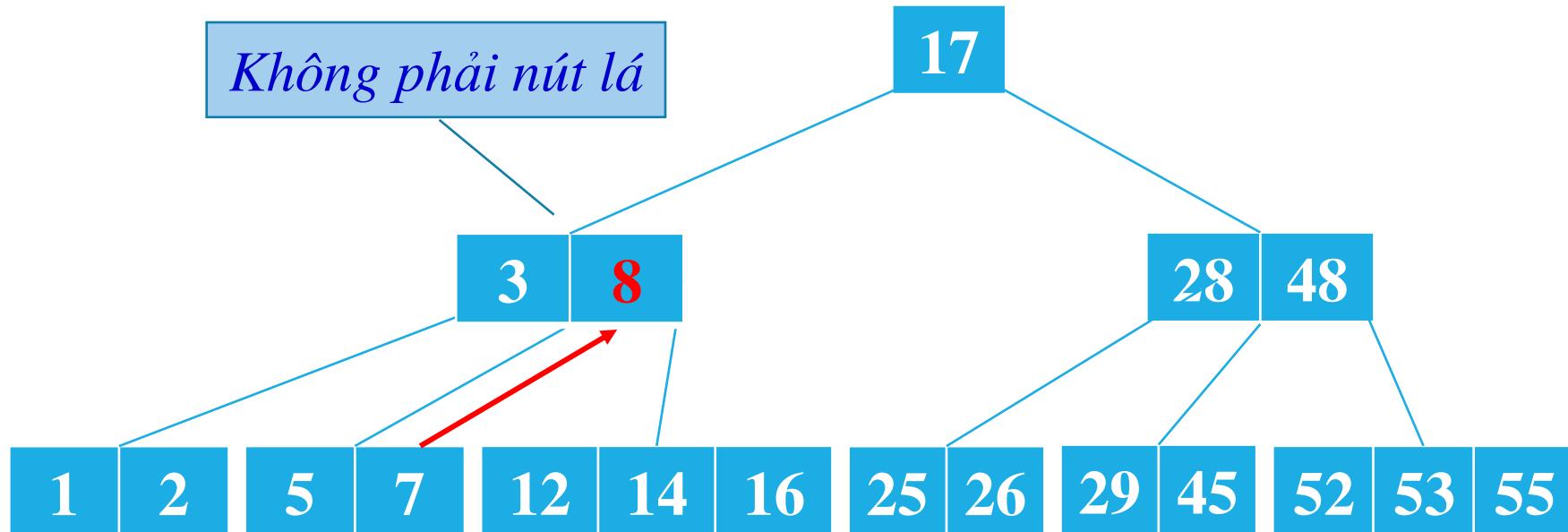
Xóa khóa trên cây

Trường hợp 1:



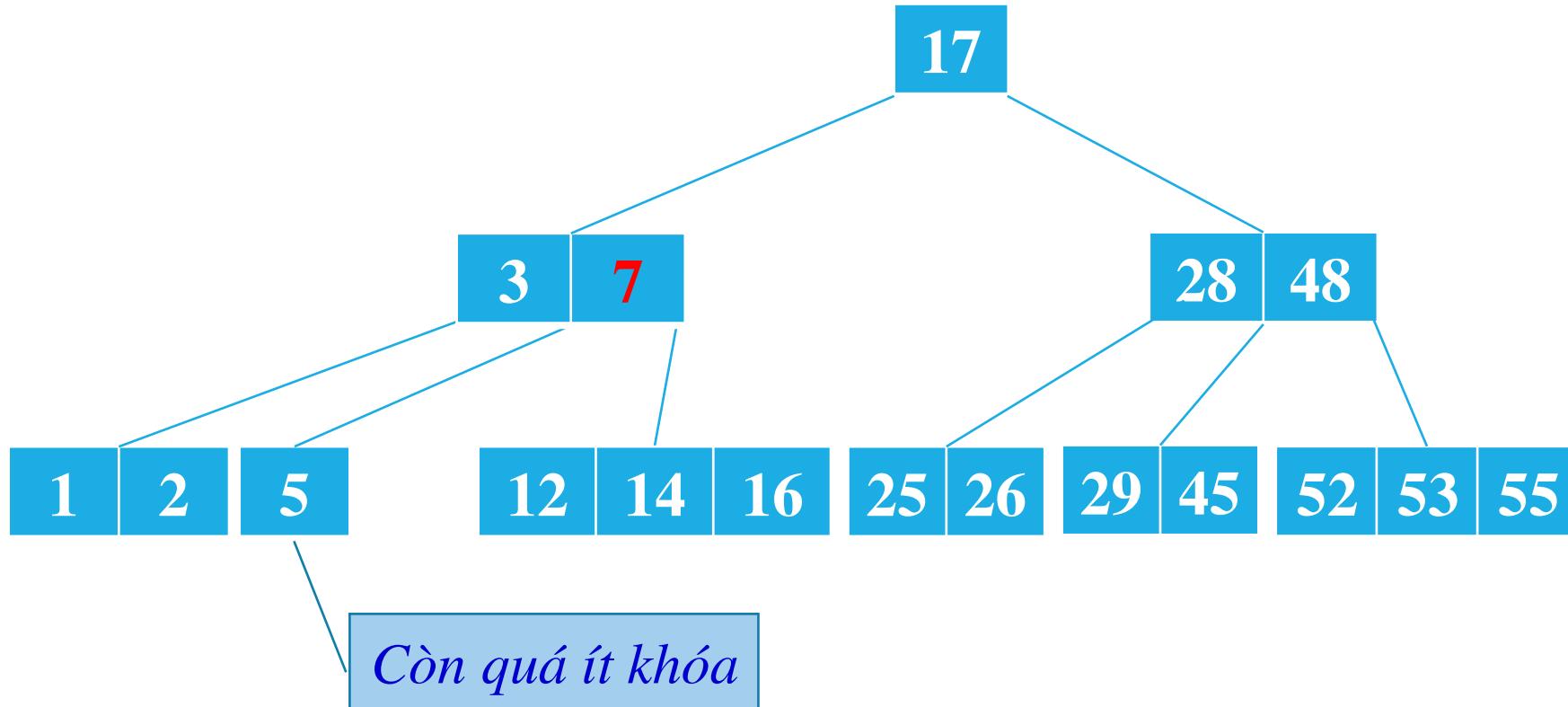
Xóa khóa trên cây

❖ Xóa 8:



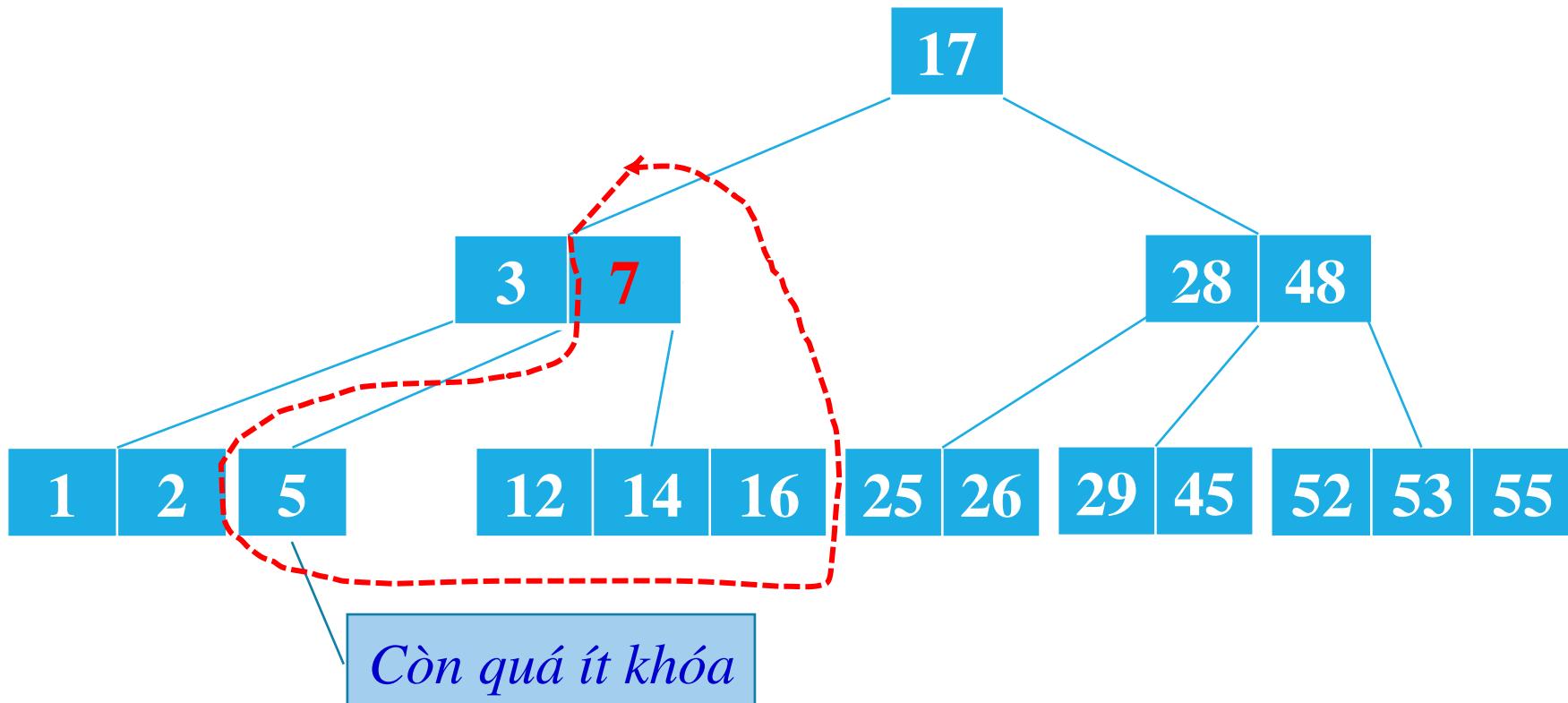
Xóa khóa trên cây

Trường hợp 2: đưa khóa có giá trị gần nhất từ nút lá lên thay thế cho khóa đã xóa.



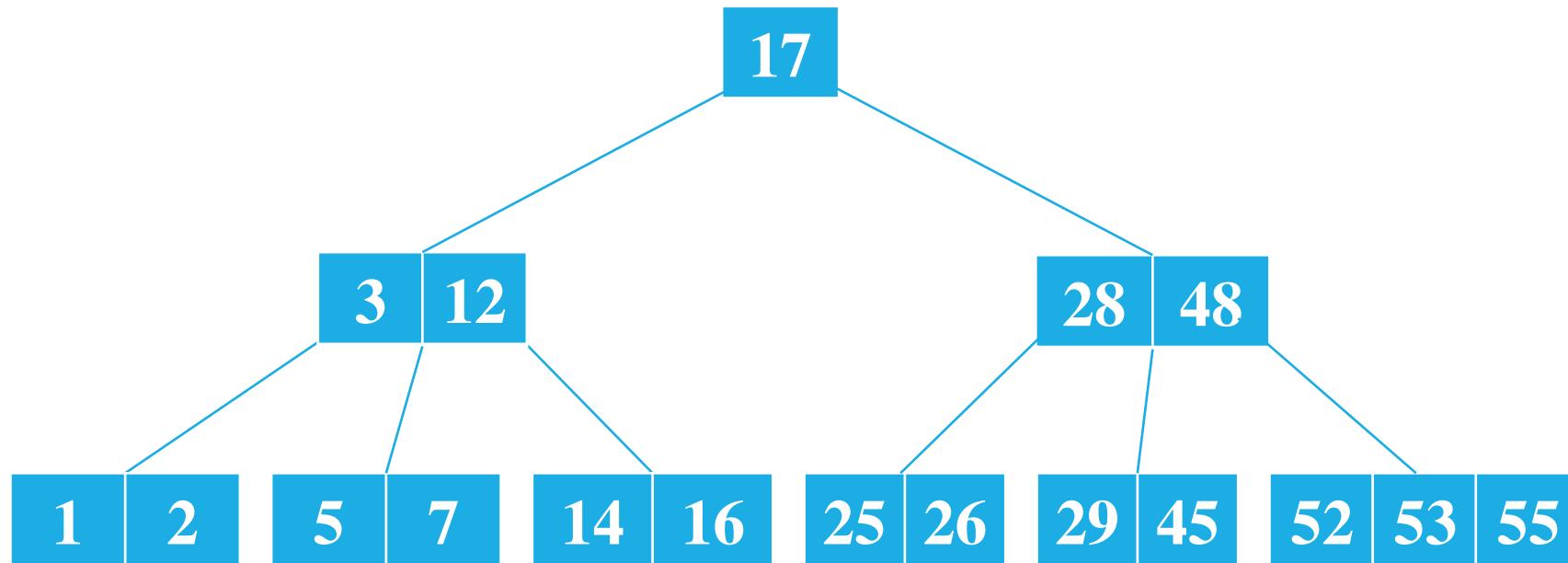
Xóa khóa trên cây

Trường hợp 3: Cân bằng các khóa giữa các nút anh em



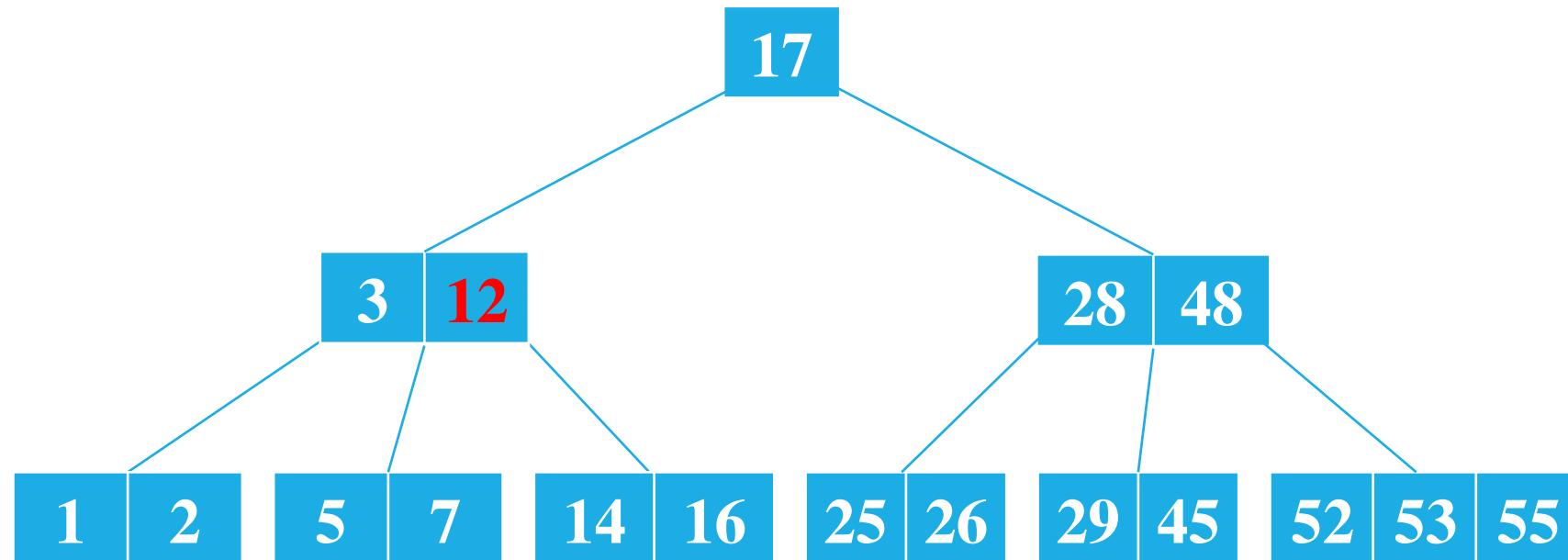
Xóa khóa trên cây

Trường hợp 3: Cân bằng các khóa giữa các nút anh em



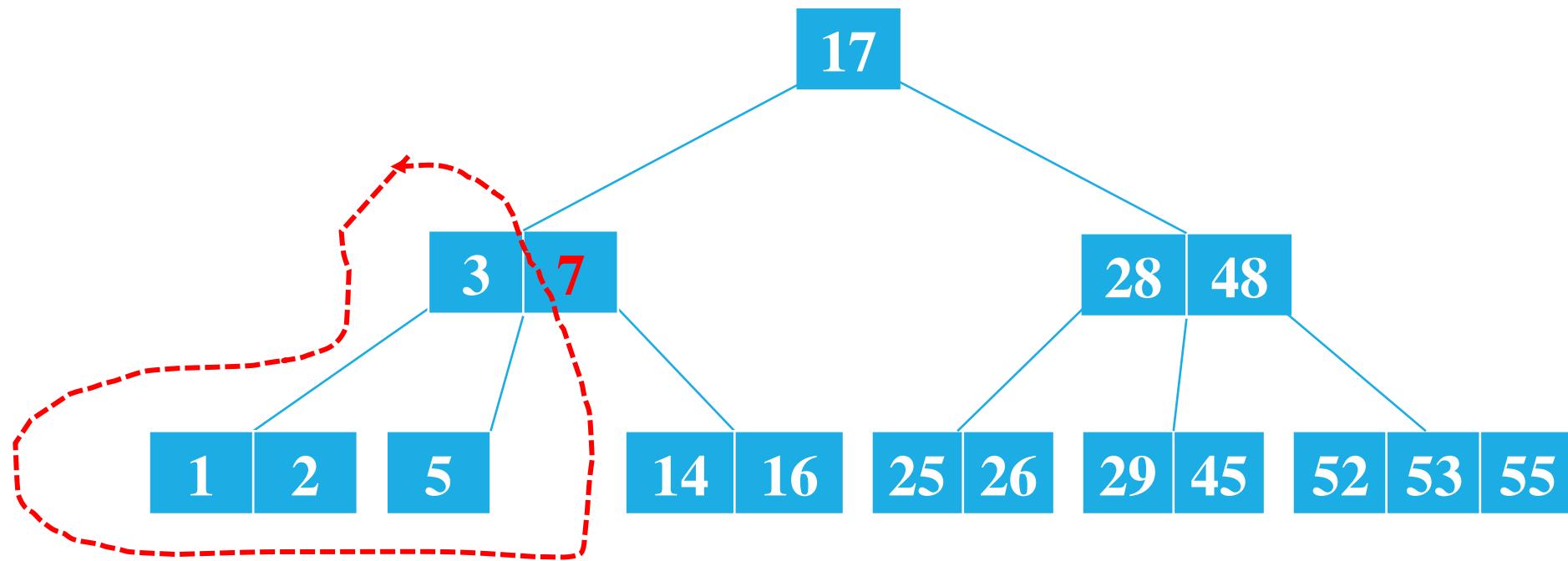
Xóa khóa trên cây

❖ Xóa 12:



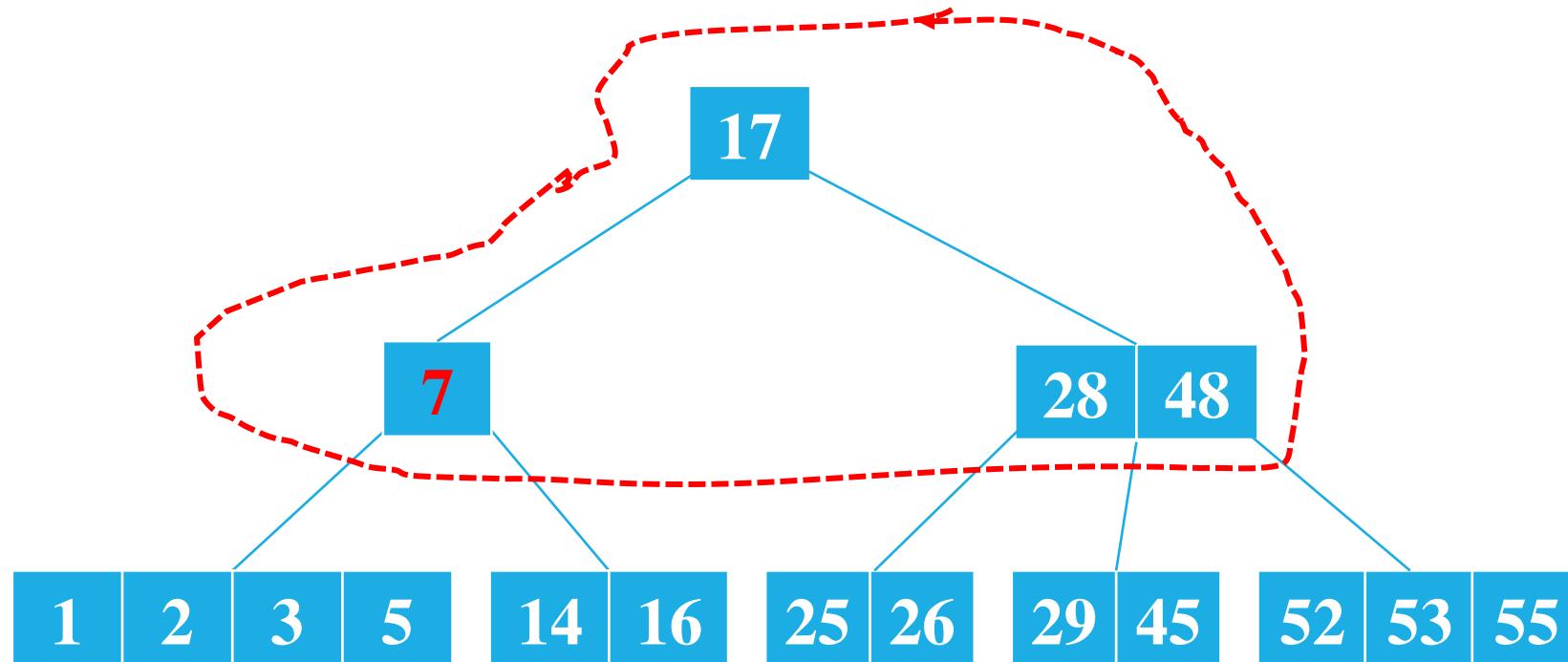
Xóa khóa trên cây

❖ Xóa 12:



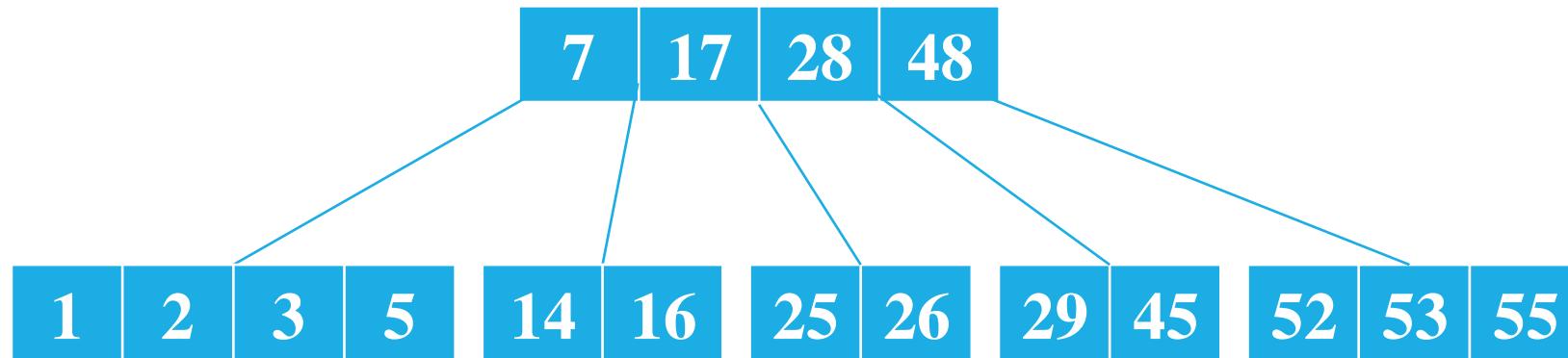
Xóa khóa trên cây

❖ Xóa 12:



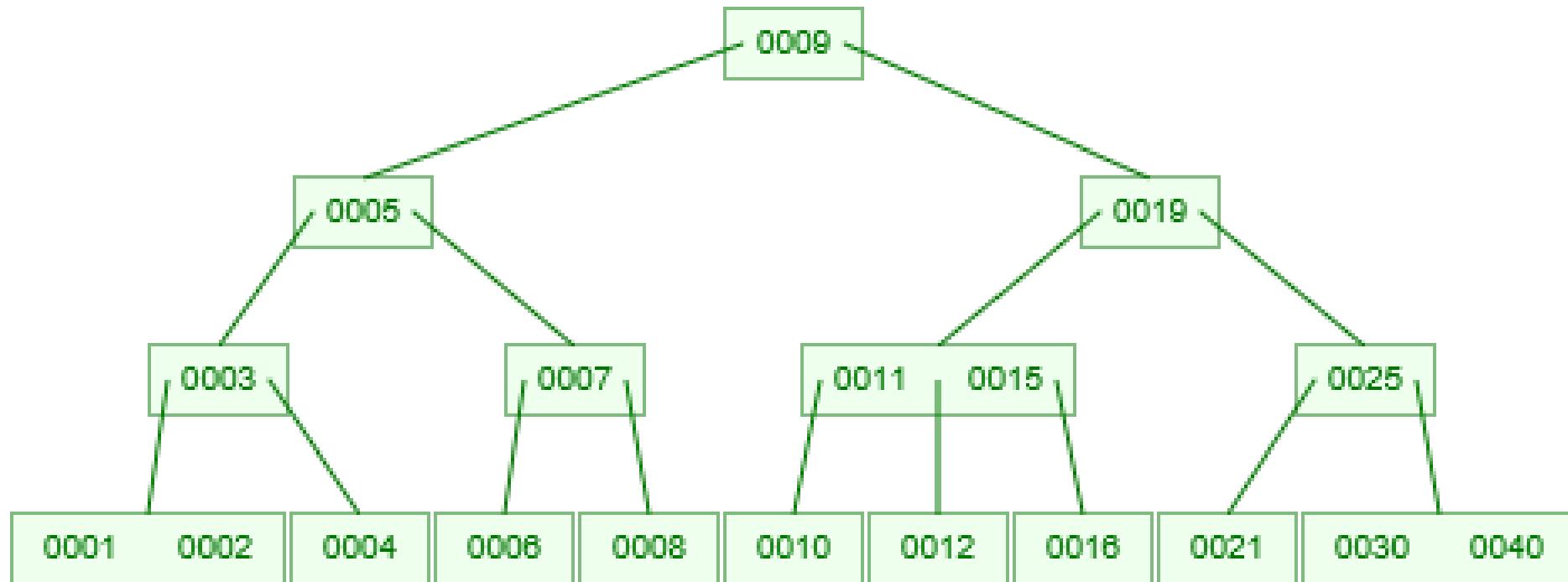
Xóa khóa trên cây

❖ Xóa 12:



Bài tập áp dụng

❖ Cho B-Tree bậc 3



Hãy trình bày từng bước lần lượt xóa các node 1, 2, 3, 4, 5, 11?

Cơ sở để so sánh	Cây B	Cây nhị phân
Hạn chế thiết yếu	Một nút có thể có tối đa số M của các nút con (trong đó M là thứ tự của cây).	Một nút có thể có tối đa 2 số cây con.
Đã sử dụng	Nó được sử dụng khi dữ liệu được lưu trữ trên thiết bị ngoại vi (đĩa).	Nó được sử dụng khi các bản ghi và dữ liệu được lưu trữ trong RAM.
Chiều cao của cây	$\log_M N$ (trong đó M là thứ tự của cây M-way, N là số nút)	$\log_2 N$
Ứng dụng	Mã cấu trúc dữ liệu lập chỉ mục trong nhiều DBMS.	Tối ưu hóa mã, mã hóa Huffman, v.v.

Khai báo

```
#define M 5

typedef int ItemType;

struct BNode {
    int numTree; /* numTree < M (Số khóa trong nút sẽ luôn ít hơn M của cây B) */
    ItemType Keys[M - 1]; /* Mảng chứa các khóa */
    BNode* Branch[M]; /* numTree + 1 (M con trỏ sẽ được sử dụng) */
};

struct BTree {
    BNode* Root; /* Con trỏ quản lý node gốc */
};

typedef BNode* NodePtr; /* Kiểu dữ liệu con trỏ node */
```

Duyệt cây (*xuất nội dung ra màn hình*)

```
void displayBTree(NodePtr proot, int blanks) {  
    if (proot) {  
        int i;  
        for (i = 1; i <= blanks; i++)  
            printf(" ");  
        for (i = 0; i < proot->numTree; i++)  
            printf("%d ", proot->Keys[i]);  
        printf("\n");  
        for (i = 0; i <= proot->numTree; i++)  
            displayBTree(proot->Branch[i], blanks + 10);  
    } /*End of if*/  
} /*End of displayBTree()*/
```

Tìm kiếm

❖ **Hàm `searchPosition()`** Trả về vị trí nhỏ nhất của khóa trong nút p bắt đầu lớn hơn hay bằng key. Trường hợp k lớn hơn tất cả các khóa trong nút p thì trả về vị trí `p->NumTrees-1`

```
int NodeSearch (NodePtr p, ItemType key) {  
    int i = 0;  
    for (i=0; i < p->NumTrees -1 && p->Key[i] < key; i++);  
    return i;  
}
```

Hàm `searchPosition()` được dùng để tìm khóa key có trong nút p hay không. Nếu khóa key không có trong nút p thì hàm này trả về vị trí giúp chúng ta chọn nút con phù hợp của p để tiếp tục tìm khóa key trong nút con này.

Tìm kiếm

Hàm **searchNode()** Tìm khóa key trên B-Tree. Con trả p xuất phát từ gốc và đi xuống các nhánh cây con phù hợp để tìm khóa key có trong một nút p hay không.

❖ Nếu có khóa key tại nút p trên cây:

- + Biến found trả về giá trị TRUE
- + Hàm **searchNode()** trả về con trỏ chỉ nút p có chứa khóa key
- + Biến position trả về vị trí của khóa key có trong nút p này

❖ Nếu không có khóa key trên cây: lúc này p = NULL và q (nút cha của p) chỉ nút lá có thể thêm khóa key vào nút này được.

- + Biến found trả về giá trị FALSE
- + Hàm **searchNode()** trả về con trỏ q là nút lá có thêm nút key vào
- + Biến position trả về vị trí có thể chèn khóa key vào nút lá q này.

NodePtr Search(int k, int *pPosition, int *pFound)

```
{  
    int i = 0;  
    NodePtr p = Root, q = NULL;  
    while (p !=NULL)  
    {  
        i = NodeSearch (p, k);  
  
        if (i< p->numtress-1 && k == p->Key[i])  
        {  
            *pFound = TRUE;  
            *pPosition = i; //Vị trí tìm thấy khóa k  
  
            return p; // Node có chứa khóa K  
        }  
        q = p;  
        p = p ->Branch[i];  
    }  
  
    /* Không tìm thấy, lúc này p = NULL, và q là node lá có thể thêm khóa k vào node này,  
     * pPosition là vị trí có thể chèn khóa k */  
    *pFound = FALSE;  
    *pPosition = i;  
  
    return q; // Trả về node lá  
}
```

Thêm node mới

- ❖ Tìm node newKey nếu có trên cây thì kết thúc công việc này tại node lá (*không thêm vào nữa*)
- ❖ Thêm newKey vào node lá, nếu chưa đầy thì thực hiện thêm vào và kết thúc Node đầy là node có số khoá = (bậc của cây)-1

Thêm Node mới

Thêm khóa key vào vị trí position của nút lá s (s và position do hàm searchPosition() trả về)

- ❖ Nếu nút lá s chưa đầy: gọi hàm insNode để chèn khóa key vào nút s
- ❖ Nếu nút lá s đã đầy: tách nút lá này thành 2 nút nửa trái và nửa phải

```
void Insert (NodePrt s, int key, int position)  
{ ... }
```

```

void Insert (NodePtr s, int k, int position)
{
    NodePtr nd = s, nd2, f = father(nd), newnode = NULL;
    int pos = position, newkey = k, midkey;

    /* Vòng lặp tác các node đầy nd */
    while (f != NULL && nd -> nemtrees == ODER)
    {
        Split(nd, newkey, newnode, pos, &nd2, &midkey);

        /* Gắn lại các trị sau lần tách node trước */
        nd = f;
        newkey = midkey;
        newnode = nd2;
        pos = nodeSearch (f, midkey);
        f = father (nd);
    }

    /* Trường hợp node nd chưa đầy và không phải là node gốc */
    if (nd -> numTree < ORDER)
    {
        /* Chèn newkey và newnode tại vị trí pos của node nd */
        InsNode (nd, newkey, newnode, pos);
        return;
    }

    /* Trường hợp nd là node gốc bị đầy, tác node gốc này và tạo node gốc mới */
    Split (nd, newkey, newnode, pos, &nd2, &midkey);

    /* Tạo node gốc mới và gắn các node nd và nd2 vào gốc */
    Root = makeroot (midkey);
    Root -> Branch[0] = nd;
    Root -> Branch[1] = nd2;
}

```

Khi thêm một khóa vào B-Tree chúng ta có thể viết như sau:

```
printf( "\n Nguồn dữ liệu mới: " );
scanf( "%d", &k );

/* Trường hợp B-Tree bị rỗng khi tạo node gốc */
if (Root == NULL)
    Root = makeroott( k );
else
{
    s = Search(k, &pos, &found);
    if ( found == TRUE )
        printf ("Bi trung khoa, khong them khoa %d vao B-Tree duoc", k);
    else
        Insert (s, k, pos);
}
```

Tách node

Tách node đầy nd, phép toán này được gọi bởi phép toán **Insert**

- nd là nút đầy bị tách, sau khi tách xong nút nd chỉ còn lại một nửa số khóa bên trái
- newkey, newnode và pos là khóa mới, nhánh cây con và vị trí chèn vào nút nd
- Nút nd2 là nút nửa phải có được sau lần tách, nút nd2 chiếm một nửa số khóa bên phải
- Midkey là khóa ngay chính giữa sẽ được chèn vào nút cha

```
void Split (NodePtr nd, int newkey, NodePtr  
newnode, int pos, NodePtr *pnd2, int *pmidkey)  
{ ... }
```

```
void Split (NodePtr nd, int newkey, NodePtr newnode, int pos, NodePtr *pnd2, int *pmidkey)
{
    NodePtr p;
    P = GetNode( ); // Cập nhật node nửa phải

    /* Trường hợp chèn newkey và newnode vào node nửa phải */
    if (pos > Ndiv2)
    {
        Copy(nd, Ndiv2+1, ORDER - 2, p);
        InsNode (p, newkey, newnode, pos-Ndiv2 -1);
        nd->numTree = Ndiv2+1; /* Số nhánh cây con còn lại của node nửa trái */
        *pmidkey = nd -> Key[Ndiv2];

        *pnd2 = p;
        return;
    }
}
```

```
/* Trường hợp newkey là midkey */
else if (pos == Ndiv2)
{
    Copy(nd, Ndiv2, ORDER-2, p);
    nd->numTree = Ndiv2+1; /* Số nhánh cây con còn lại của node nửa trái */

    /* Điều chỉnh lại node con đầu tiên của node nửa phải */
    p -> Branch[0] = newnode;
    *pmidkey = nd -> Key[Ndiv2];
    *pnd2 = p;

    return ;
}
```

```
/* Trường hợp chèn newkey và newnode vào node nửa trái */
else if (pos < Ndiv2)
{
    Copy(nd, Ndiv2, ORDER-2, p);
    nd->numTree = Ndiv2+1; /* Số nhánh cây con còn lại của node nửa trái */

    *pmidkey = nd -> Key[Ndiv2 - 1];
    InsNode(nd, newkey, newnode, pos);
    *pnd2 = p;

    return;
}
}
```

Hàm insNode

- ❖ Chèn khóa newkey vào vị trí pos của nút chưa đầy p, và chèn nhánh cây con newNode vào vị trí bên phải của khóa newKey

```
void insNode (NodePtr p, int newKey, NodePtr  
newNode, int pos)  
{ ... }
```

```
void InsNode (NodePtr p, int newkey, NodePtr newnode, int pos)
{
    /* Dời các nhánh con và các khóa từ vị trí pos trở về sau xuống một vị trí */
    for (int i = p->numtress - 1; i >= pos+1; i--)
    {
        p -> Branch[i+1] = p -> Branch[i];
        p -> Key[i] = p -> Key[i - 1];
    }

    /* Gắn khóa newkey vào vị trí pos */
    p -> Key[pos] = newkey;

    /* Gắn nhánh newnode là nhánh cây con bên phải của khóa newkey */
    p -> Branch[pos + 1] = newnode;

    /* Tăng số nhánh cây con của node p lên 1 */
    p -> numTree +=1;
}
```

Hàm Copy

- ❖ Chép các khóa (và nhánh cây con) từ vị trí first đến vị trí fast của nút nd (nút nửa trái) sang nút nd2 (nửa nút phải). Hàm này được gọi bởi hàm **Split**

```
void copy (NodePtr nd, int first, int last,  
          NodePtr nd2)  
{ ... }
```

```
void Copy(NodePtr nd, int first, int last, NODPTR nd2)
{
    /* Copy các khóa từ node nd qua node nd2 */
    for (int i = first; i < last, i++)
        nd2 -> Key[i-first] = nd -> Key[i];

    /* Copy các nhánh từ node nd qua nd2 */
    for (i = first; i < last+1, i++)
        nd2 -> con[i-first] = nd -> Branch[i];

    /* Số nhánh cây con của node nd2 */
    nd2 ->numTree = last - first +2;
}
```

Xóa Node

Sinh viên tự tìm hiểu xem như bài tập về nhà.

Câu hỏi và Bài tập

1. Nêu định nghĩa và các tính chất của cây B-Tree.
2. Cài đặt tất cả các thao tác trên cây B-Tree.
3. Cho B-Tree bậc 5 gồm các khóa sau (chèn vào theo thứ tự): 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56

Thực hiện các yêu cầu sau:

- Thêm các khóa: 2, 6, 12, 10, 11
- Xóa khóa: 4, 5, 7, 3, 14

4. Khởi tạo B-Tree bậc 7 với các thao tác Insert: 34, 12, 55, 21, 6, 84, 5, 33, 15, 74, 54, 28, 10, 19

Thực hiện các chuỗi thao tác sau:

- Insert(11), Delete(15), Delete(6), Insert(98), Delete(34), Delete(5)





CẤU TRÚC DỮ LIỆU & GIẢI THUẬT (DATA STRUCTURES & ALGORITHMS)

Chương 4

BẢNG BĂM VÀ TÌM KIẾM DỮ LIỆU

Khoa Công nghệ thông tin
Bộ môn Công nghệ phần mềm

Nội dung

1. Mô tả bảng băm
2. Bảng băm với PP Kết nối trực tiếp
3. Bảng băm với phương địa chỉ mở

1. MÔ TẢ BẢNG BĂM

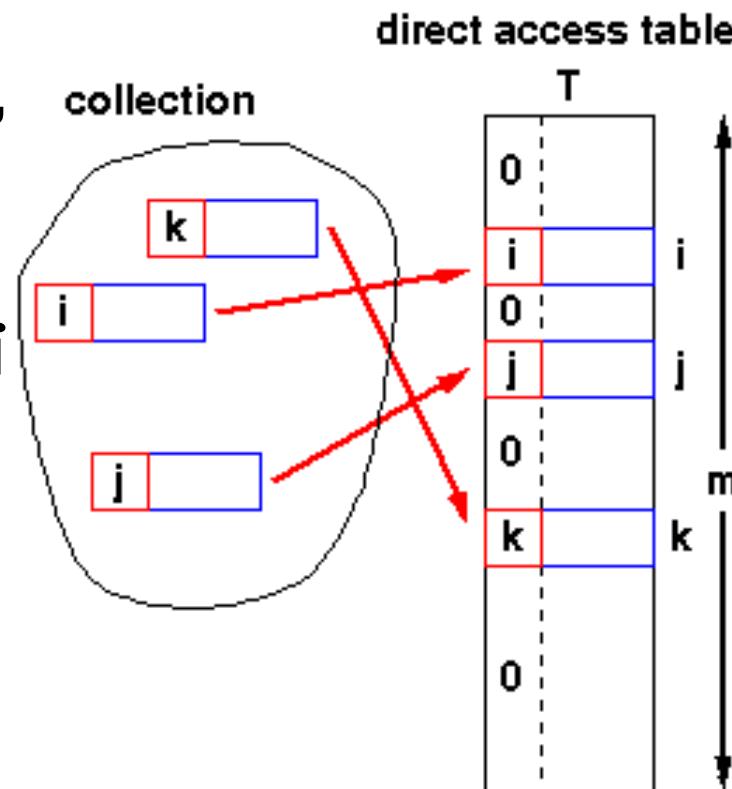
- ✓ Các thuật toán tìm kiếm đều dựa vào việc so sánh giá trị khoá (Key)
 - ❖ Phụ thuộc kích thước của tập các phần tử
 - ❖ Thời gian tìm kiếm không nhanh do phải thực hiện nhiều phép so sánh có thể không cần thiết ($O(n)$, $O(\log n)$, ...)
- => Có phương pháp lưu trữ nào cho phép thực hiện tìm kiếm với **hiệu suất cao** hơn không (độ phức tạp hằng số)?

1. MÔ TẢ BẢNG BĂM

✓ Bảng gồm m phần tử được lưu trữ dưới dạng bảng chỉ mục:

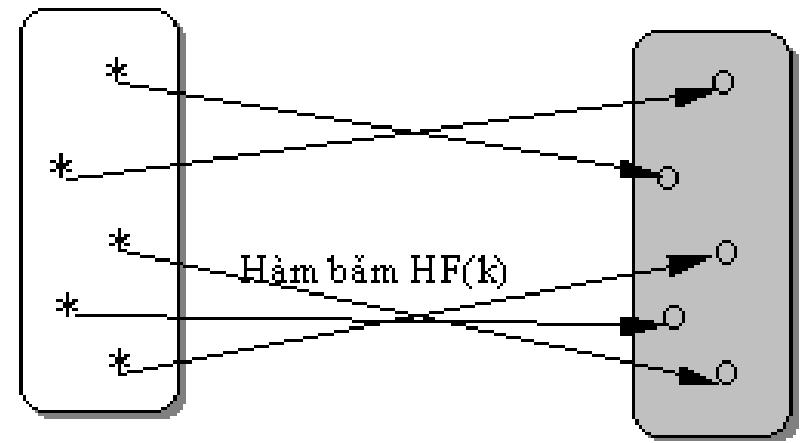
- ❖ Phần tử có giá trị khoá k được lưu trữ tương ứng tại vị trí thứ k.
- ❖ Tìm kiếm bằng cách tra trong bảng chỉ mục.
- ❖ Thời gian tìm kiếm là $O(1)$

→ Đây là dạng bảng băm cơ bản



1. MÔ TẢ BẢNG BĂM

- ✓ **K**: tập các giá trị khoá (set of keys) cần lưu trữ.
 - ✓ **A**: tập các địa chỉ (set of addresses) trong bảng băm.
 - ✓ **HF(k)**: hàm băm dùng để ánh xạ một khoá k từ tập các khoá K thành một địa chỉ tương ứng trong tập các địa chỉ A.



Tập khóa K

Tập địa chỉ A

HF: K → A

$$k \longrightarrow a = H(k)$$

1. MÔ TẢ BẢNG BĂM

✓ *Bảng băm đóng:*

- ❖ Số phần tử cố định.
- ❖ Mỗi khóa ứng với một địa chỉ.
- ❖ Không thể thực hiện các thao tác thêm, xóa trên bảng băm.
- ❖ Thời gian truy xuất là hằng số.

✓ *Bảng băm mở:*

- ❖ Số phần tử không cố định.
- ❖ Một số khóa có thể có cùng địa chỉ.
- ❖ Có thể thực hiện các thao tác thêm, xóa phần tử.
- ❖ Thời gian truy xuất có thể bị suy giảm đôi chút.

1. MÔ TẢ BẢNG BĂM

- Là hàm biến đổi giá trị khoá (số, chuỗi...) của phần tử thành địa chỉ, chỉ mục trong bảng băm



Ví dụ: hàm băm biến đổi khóa chuỗi thành 1 địa chỉ (số nguyên)

✓ Tính địa chỉ của khoá “AB”:

HashFunction(“AB”, 2) → 131

✓ Tính địa chỉ của khoá “BA”:

HashFunction(“BA”, 2) → 131

```
int HashFunction(char *s, int n)
{
    int sum = 0;
    while( n-- )
        sum = sum + *s++;
    return (sum % 256);
}
```

Khi hàm băm 2 khoá vào cùng 1 địa chỉ gọi là đụng độ (Collision).

1. MÔ TẢ BẢNG BĂM

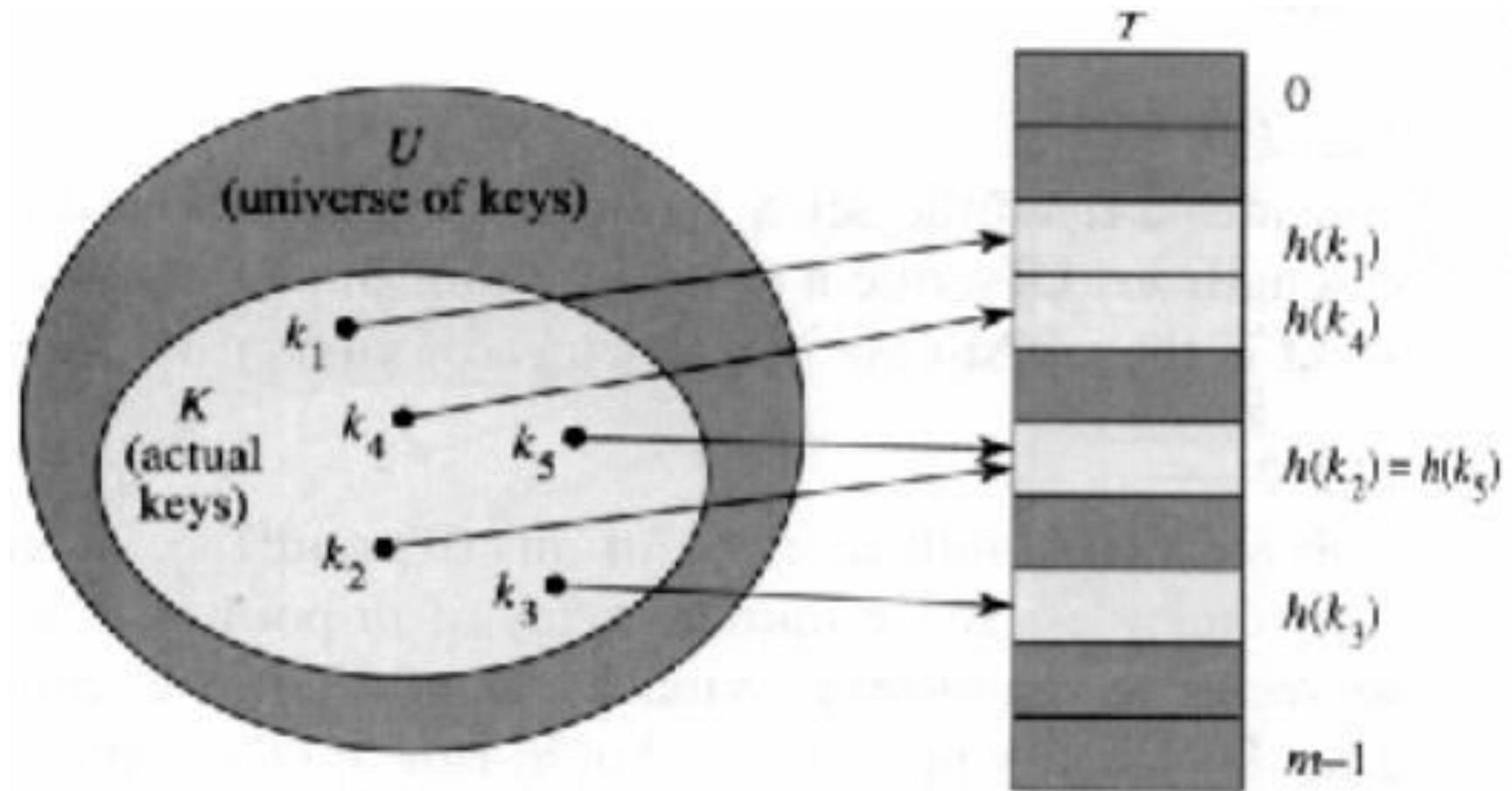
✓ SỰ ĐỤNG ĐỘ (Collision):

$$\exists k_1, \exists k_2 \in K:$$

$$k_1 \neq k_2, H(k_1) = H(k_2)$$

1. MÔ TẢ BẢNG BĂM

- ✓ Độ giũa 2 khoá k_2 và k_5



1. MÔ TẢ BẢNG BĂM

Tiêu chuẩn đánh giá hàm băm:

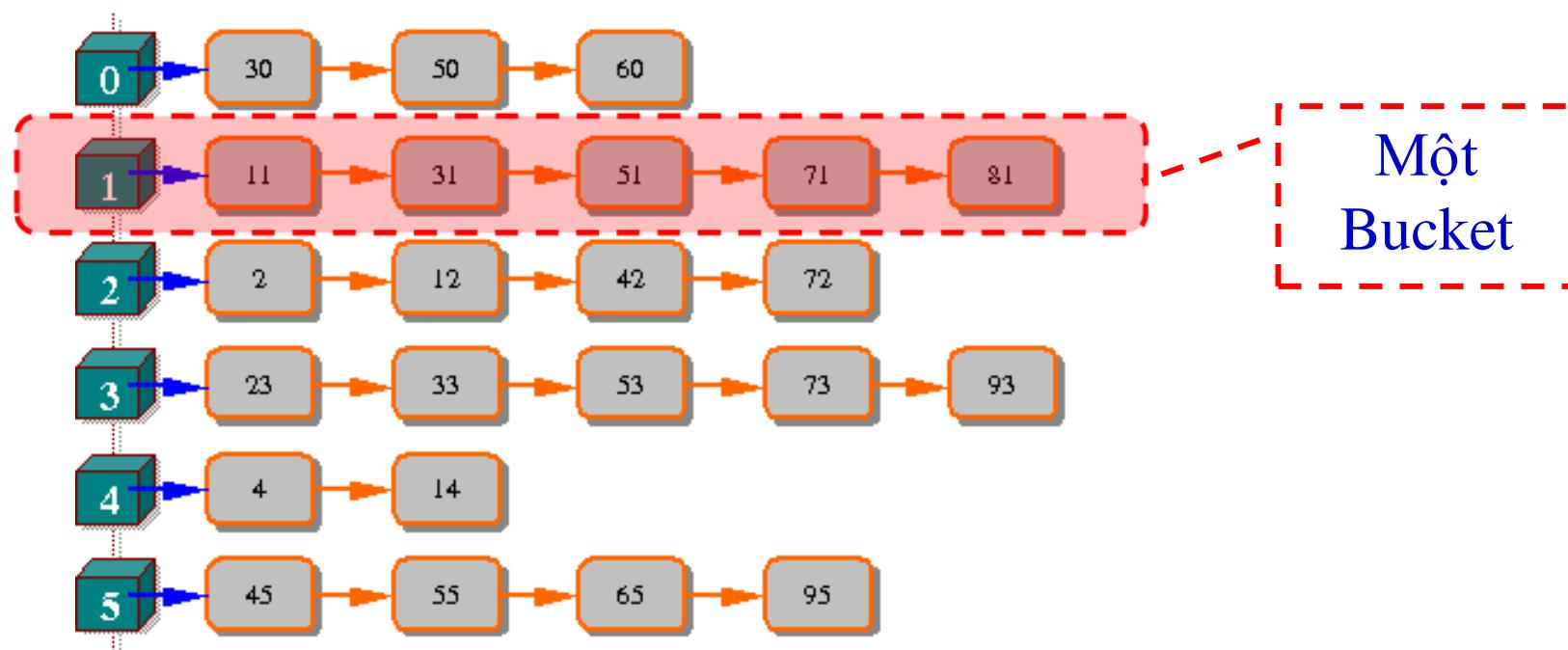
- Tính toán nhanh.
- Giảm thiểu sự xung đột (*Ít xảy ra đụng độ*).
- Các khóa được phân bố đều trong bảng (*Phân bố đều các nút trên MAXSIZE địa chỉ khác nhau của bảng băm*).
- Xử lý được các loại khóa có kiểu dữ liệu khác nhau.

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Ứng với mỗi địa chỉ của bảng, ta có 1 danh sách liên kết chứa các phần tử có khóa khác nhau mà có cùng 1 địa chỉ đó.
- ✓ Vậy ta sẽ có 1 danh sách (bảng băm) gồm **MAXSIZE** phần tử chứa địa chỉ đầu của các danh sách liên kết.

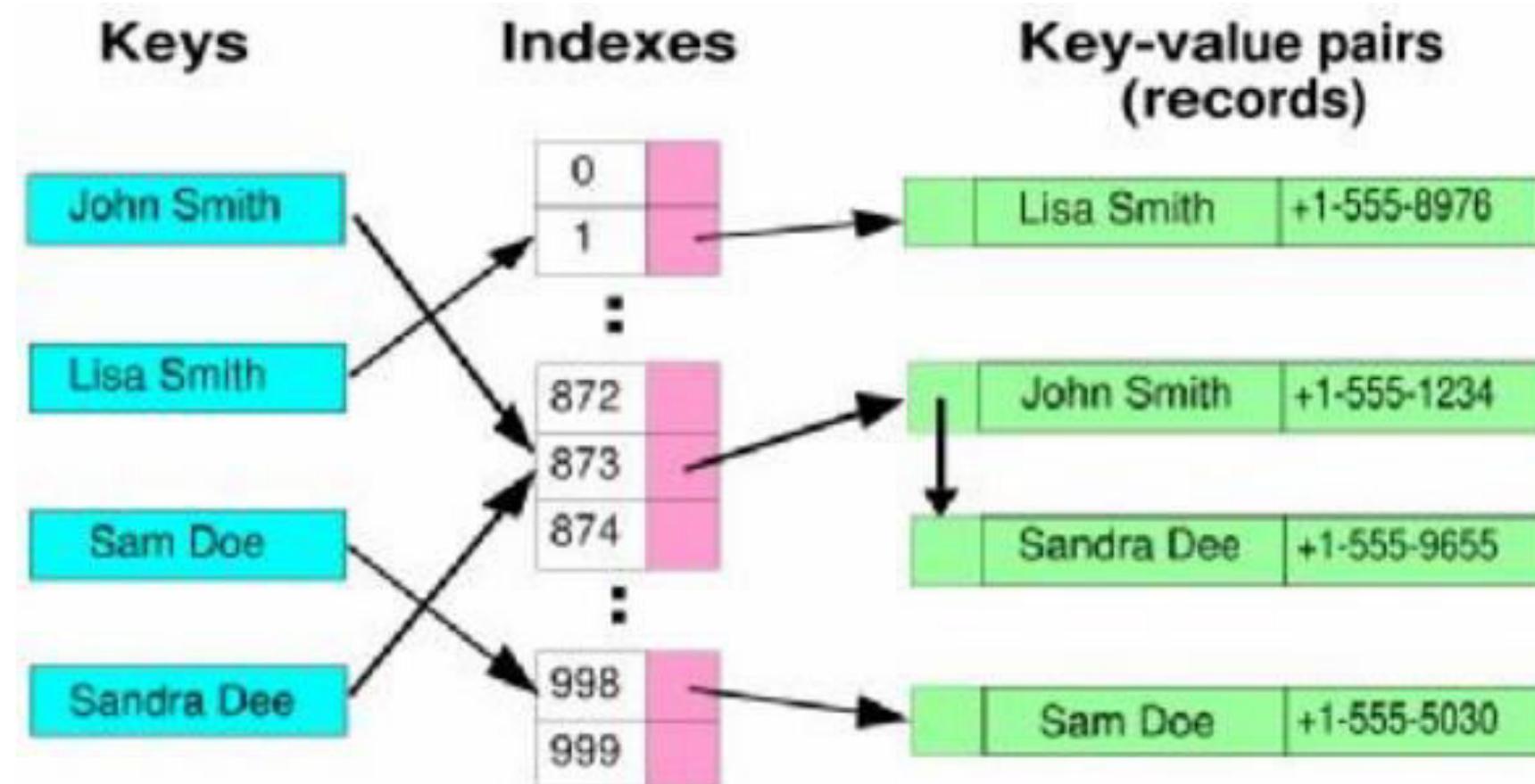
2. Bảng băm với PP Kết nối trực tiếp

- ✓ Các phần tử bị đụng độ được gom thành một danh sách liên kết (gọi là một bucket).



2. Bảng băm với PP Kết nối trực tiếp

- ✓ Giải quyết độ densus bằng phương pháp Chaining



2. Bảng băm với PP Kết nối trực tiếp

- ✓ Khai báo cấu trúc bảng băm:

```
#define TRUE 1
#define FALSE 0
#define MAXSIZE 100
struct HashNode
{
    int Key;
    HashNode* Next;
};
```

- ✓ Khai báo kiểu con trỏ chỉ nút:

```
typedef HashNode* NodePtr;
```

- ✓ Khai báo mảng bucket chứa MAXSIZE con trỏ đầu của MAXSIZE bucket

```
NodePtr bucket[M];
```

2. Bảng băm với PP Kết nối trực tiếp

✓ Hàm băm

```
int HashFunction(int Key)
{
    return (Key % MAXSIZE);
}
```

✓ Phép toán khởi tạo (InitBucket)

```
void InitBucket( )
{
    for(int b = 0; b < MAXSIZE; b++)
        bucket[b] = NULL;
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Phép toán kiểm tra bucket rỗng (IsEmptyBucket)

```
int IsEmptyBucket(int b){  
    return(bucket[b] == NULL ? TRUE : FALSE);  
}
```

- ✓ Phép toán thêm một nút vào đầu bucket:

```
void Push(int b, int x) {  
    NodePtr p = new HashNode;  
    p→Key = x;  
    p→Next = bucket[b];  
    bucket[b] = p;  
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Phép toán thêm vào bucket một Node mới sau Node p:

```
void InsertAfter(NodePtr p, int k) {  
    if(p == NULL)  
        printf("Khong them vao Node moi duoc");  
    else {  
        NodePtr q = new HashNode;  
        q→Key = k;  
        q→Next = p→Next;  
        p→Next = q;  
    }  
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Phép toán chèn khóa k vào danh sách liên kết:

```
void Place(int b, int k) {  
    NodePtr p, q;  
    q = NULL;  
    for(p = bucket[b]; p!=NULL && k>p→Key; p = p→Next)  
        q = p;  
    if(q == NULL)  
        Push(b, k);  
    else  
        InsertAfter(q, k);  
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Phép toán kiểm tra bảng băm rỗng IsEmpty:

```
int IsEmpty( )  
{  
    for(int b = 0; b<M; b++)  
        if(bucket[b] != NULL) return(FALSE);  
    return(TRUE);  
}
```

- ✓ Phép toán chèn phần tử có khóa k vào bảng băm:

```
void Insert(int k)  
{  
    int b = HashFunction(k);  
    Place(b, k); //chen k vao danh sach lien ket  
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ❖ Phép toán xóa một nút trong bộ nhớ

```
void FreeNode(NodePtr p)
{
    delete p;
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Phép toán xóa một nút ở đầu bucket:

```
int Pop(int b) {  
    NodePtr p;  
    int k;  
    if(IsEmptyBucket(b)) {  
        printf("Bucket %d bi rong, khong xoa duoc", b);  
        return FALSE;  
    }  
    p = bucket[b];          k = p→Key;  
    bucket[b] = p→Next;     FreeNode(p);  
    return k;  
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Phép toán xóa một nút ngay sau nút p

```
int DeleteAfter(NodePtr p) {
    NodePtr q;
    int k;
    if(p == NULL || p→Next == NULL) {
        printf("\n Khong xoa Node duoc");
        return FALSE;
    }
    q = p→Next;           k = q→Key;
    p→Next = q→Next;     FreeNode(q);
    return k;
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ❖ Phép toán hủy mục có khóa k trong bảng băm

```
void Remove(int k) {  
    NodePtr q, p;  
    int b = HashFunction(k);      p = bucket[b];  
    while(p != NULL && p→Key != k) {  
        q = p;      p = p→Next;  
    }  
    if(p == NULL)  
        printf("\n khong co nut co khoa %d", k);  
    else if(p == bucket[b])  
        Pop(b);  
    else  
        DeleteAfter(q); //xoá nut  
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Phép toán xóa bucket trong bảng băm

```
void ClearBucket(int b)
{
    NodePtr p, q; //q la nut truoc,p la nut sau
    q = NULL;
    p = bucket[b];
    while(p != NULL) {
        q = p;
        p = p->Next;
        FreeNode(q);
    }
    bucket[b] = NULL; //khoi dong lai bucket b
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Phép toán xóa tất cả các phần tử trong bảng băm.

```
void Clear( ) {  
    for(int b=0; b<MAXSIZE; b++) ClearBucket(b);  
}
```

- ✓ Phép toán duyệt các phần tử trong bucket b.

```
void TraverseBucket(int b) {  
    NodePtr p = bucket[b];  
    while(p != NULL) {  
        printf("%5d", p→Key);  
        p = p→Next;  
    }  
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Phép toán duyệt toàn bộ bảng băm:

```
void Traverse( )  
{  
    for(int b = 0; b < MAXSIZE; b++)  
    {  
        printf("\nBucket thu %d:", b);  
        TraverseBucket(b);  
    }  
}
```

2. Bảng băm với PP Kết nối trực tiếp

- ✓ Phép toán tìm kiếm một phần tử trong bảng

```
NodePtr Search(int k) {  
    int b = HashFunction(k);  
    NodePtr p = bucket[b];  
    while( (k > p→Key) && (p != NULL) )  
        p = p→Next;  
    if(p==NULL || k!=p→Key) //khong tim thay  
        return (NULL);  
    else  
        return (p);  
}
```

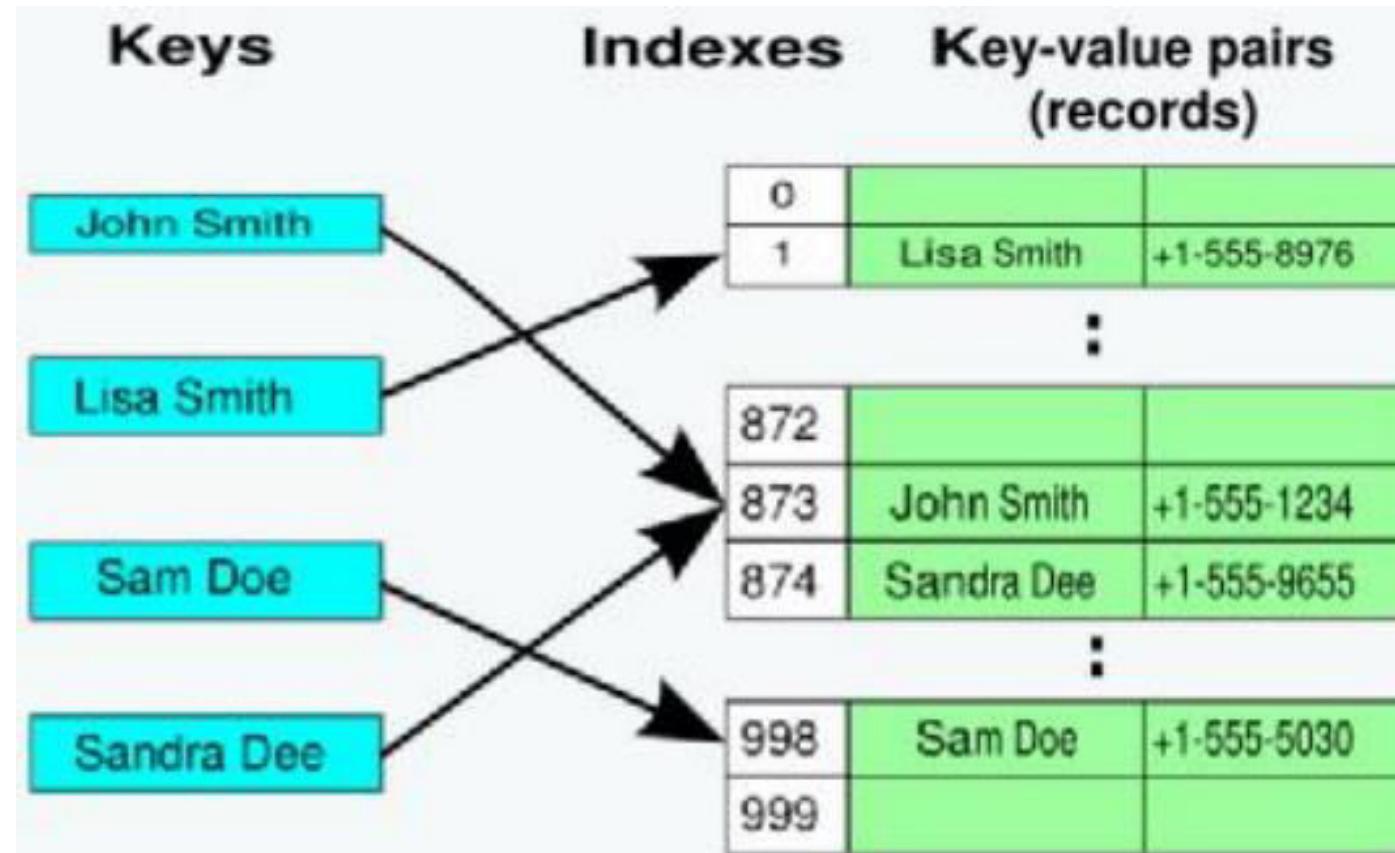
3. Bảng băm với phương địa chỉ mở (Open addressing)

- ✓ Khi đụng độ xảy ra, ta sẽ thử tìm đến vị trí kế tiếp nào đó trong bảng cho đến khi tìm thấy vị trí nào còn trống.
- ✓ Còn được biết đến là phương pháp: Phương pháp dò, **Phương pháp thử**

3. Bảng băm với phương địa chỉ mở

Linear probing

- ✓ Giải quyết đụng độ bằng phương pháp dò tuyến tính (Linear probing)



3. Bảng băm với phương địa chỉ mở

Linear probing

✓ Ý tưởng:

$$h(k, i) = (\mathbf{H}(k) + i) \bmod \text{MAXSIZE}$$

- ✓ i: thứ tự của lần thử ($i = 0, 1, 2, \dots$)
- ✓ $\mathbf{H}(k)$: hàm băm
- ✓ MAXSIZE: số phần tử của bảng băm

3. Bảng băm với phương địa chỉ mở

Quadratic probing

✓ Ý tưởng:

$$\checkmark h(k, i) = (\mathbf{H}(k) + i^2) \bmod \text{MAXSIZE}$$

✓ i: thứ tự của lần thử ($i = 0, 1, 2, \dots$)

✓ $\mathbf{H}(k)$: hàm băm

✓ MAXSIZE: số phần tử của bảng băm

3. Bảng băm với phương địa chỉ mở

Double hashing

✓ Ý tưởng:

$$\checkmark h(k, i) = (\mathbf{H1}(k) + i * \mathbf{H2}(k)) \text{ mod MAXSIZE}$$

- ✓ i: thứ tự của lần thử ($i = 0, 1, 2, \dots$)
- ✓ $\mathbf{H1}(k)$ và $\mathbf{H2}(k)$: hàm băm
- ✓ MAXSIZE: số phần tử của bảng băm

3. Bảng băm với phương địa chỉ mở

- ✓ Đơn giản khi cài đặt
- ✓ Sử dụng các cấu trúc dữ liệu cơ bản
- ✓ Open - addressing giải quyết được dung độ nhưng lại có thể gây ra những dung độ mới
- ✓ Chaining không bị ảnh hưởng về tốc độ khi bảng gần đầy
- ✓ Ít tốn bộ nhớ khi bảng thừa (chứa ít phần tử)

Bài tập

1. Nêu định nghĩa và các tính chất của bảng băm?
2. Thế nào là dung độ và trình bày cách giải quyết dung độ?
3. Cài đặt bảng băm theo phương pháp kết nối trực tiếp để lưu trữ thông tin của từ điển Anh – Việt đơn giản
4. Xây dựng cấu trúc bảng băm để lưu trữ tên đăng nhập và mật khẩu của chương trình: Cho phép thêm, xóa, tìm kiếm nhanh mật khẩu khi biết tên đăng nhập.

