

## Chương 7. Cây (Tree)

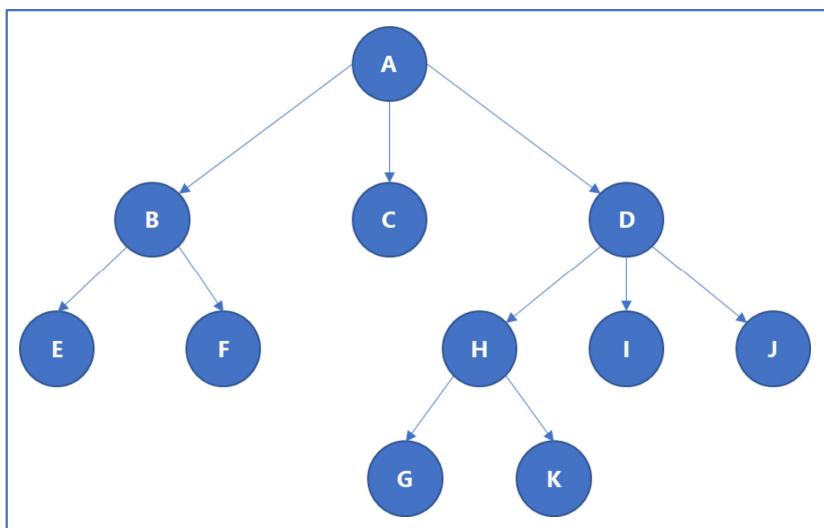
### 7.1. Cấu trúc cây

#### 7.1.1. Khái niệm cây

Cây là một tập hợp các phần tử, hay còn gọi là các nút, gồm:

- Nút gốc (root).
- Các nút còn lại chia thành các tập con  $T_1, T_2, \dots, T_n$ . Trong đó các  $T_i$  cũng là một cây (cây con)

- **Ví dụ 7.1:**



**Hình 7.1** Ví dụ minh họa 1 cấu trúc cây.

#### 7.1.2. Tính chất cây

##### 7.1.2.1. Độ tuổi của nút:

Là số cây con của nút đó.

- **Ví dụ 7.2:** Xét cây trong **ví dụ 7.1**:

- Độ tuổi của nút A là 3
- Độ tuổi của nút B là 2
- Độ tuổi của nút C là 0

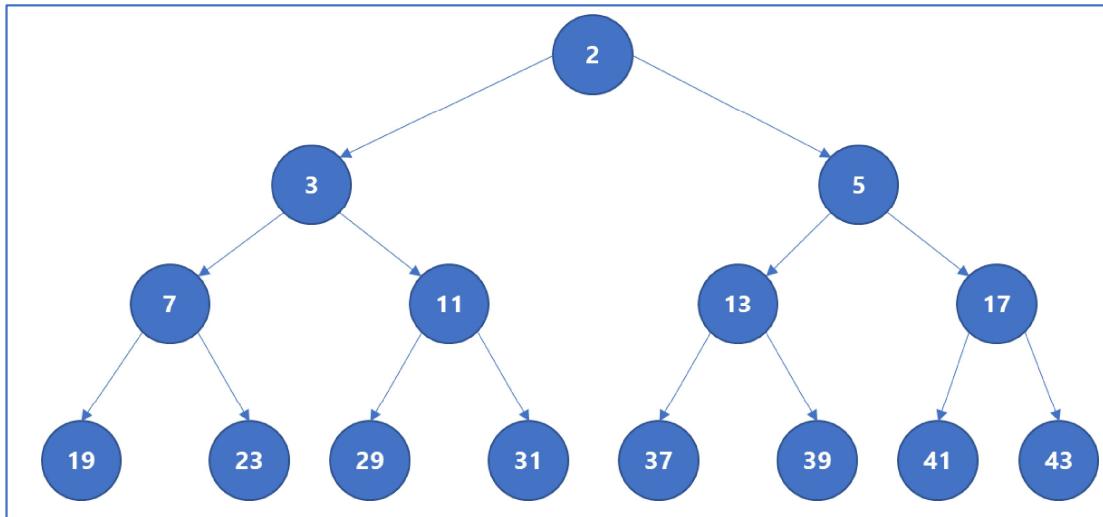
(?) *Câu hỏi: Hãy liệt kê độ tuổi của các nút H, D, G, K của cây ở **ví dụ 7.1***

##### 7.1.2.2. Độ tuổi của cây

- Là độ tuổi của nút có độ tuổi lớn nhất trên cây.
- Cây có độ tuổi n được gọi là cây n-phân.

- **Ví dụ 7.3:** Độ tuổi của cây trong **ví dụ 7.1** là 3, gọi là cây tam phân.

(?) *Câu hỏi: Xác định độ tuổi của cây trong hình dưới đây:*



**Hình 7.2.** Minh họa 1 cấu trúc cây để xác định bậc.

### 7.1.2.3. Nút lá

Là nút có bậc bằng 0 (không chứa cây con nào).

- **Ví dụ 7.4:** Ở **Hình 7.2** có nút 19, 23 là nút lá.

(?) Câu hỏi:

- Liệt kê tất cả các nút lá của cây trong **Hình 7.2**.
- Nút C của cây ở **Hình 7.1** có phải nút lá không?
- Liệt kê tất cả các nút lá của cây trong **Hình 7.1**.

### 7.1.2.4. Nút nhánh

- Là nút có bậc khác 0, và không phải là nút gốc.
- Nút nhánh hay còn gọi là nút trung gian.

- **Ví dụ 7.5:** Ở **Hình 7.2** các nút nhánh là 3, 5, 7, 11, 13, 17.

(?) Câu hỏi:

- Nút I của cây trong **Hình 7.1** có phải nút nhánh không?
- Liệt kê tất cả các nút nhánh của cây trong **Hình 7.1**.

### 7.1.2.5. Mức của nút

- Mức của gốc  $T_0 = 0$ .
- Xét  $T_1, T_2, \dots, T_n$  là các cây con của  $T_0$ .
- Khi đó: Mức ( $T_1$ ) = Mức ( $T_2$ ) = ... = Mức ( $T_n$ ) = Mức( $T_0$ ) + 1.

- **Ví dụ 7.6:** Xét cây trong **Hình 7.2**, ta có:

- Mức của nút 2 là 0.
- Mức của nút 3 và nút 5 là 1.
- Mức của nút 7, 11, 13, 17 là 2.

(?) Câu hỏi: Bạn hãy cho biết mức của nút G của cây trong **Hình 7.1**.

### 7.1.2.6. Chiều cao

Chiều cao (hay còn gọi là chiều sâu) của cây: là mức lớn nhất của các nút lá.

- **Ví dụ 7.7:** Cây trong **Hình 7.2** có mức là 3.

(?) *Câu hỏi: Xác định mức của cây trong **Hình 7.1**.*

### 7.1.2.7. Độ dài đường đi của nút x

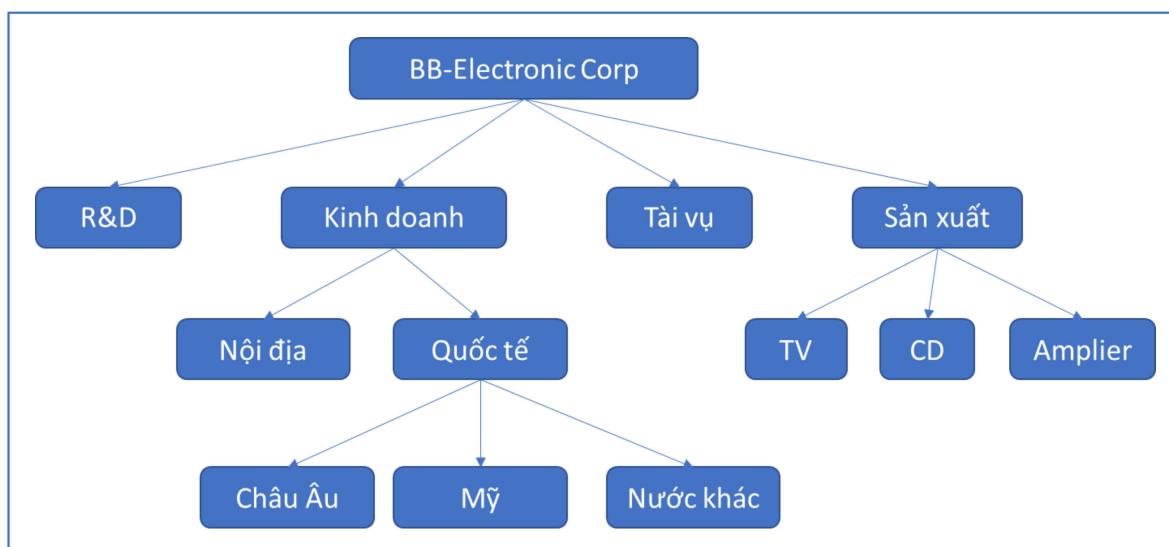
- Là số nhánh (cạnh) cần đi qua kể từ nút gốc (root) đến nút x.
- Một nút tại mức l có độ dài đường đi là i.

- **Ví dụ 7.8:** Trong **Hình 7.2**:

- Nút 13 có độ dài đường đi là 2.
- Nút 37 có độ dài đường đi là 3.

(?) *Câu hỏi: Xác định độ dài đường đi của nút F, I, J, K của cây trong **Hình 7.1**.*

### 7.1.3. Ví dụ về đối tượng có cấu trúc dạng cây



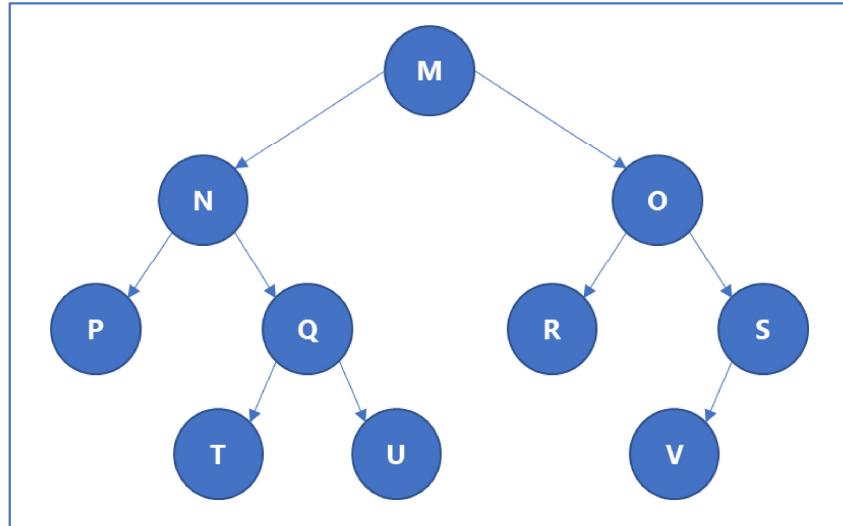
**Hình 7.3.** Mô hình một công ty được tổ chức dưới dạng cây.

## 7.2. Cây nhị phân

### 7.2.1. Khái niệm cây nhị phân

Cây nhị phân là cây mà mỗi nút có tối đa 2 cây con (hoặc là cây rỗng).

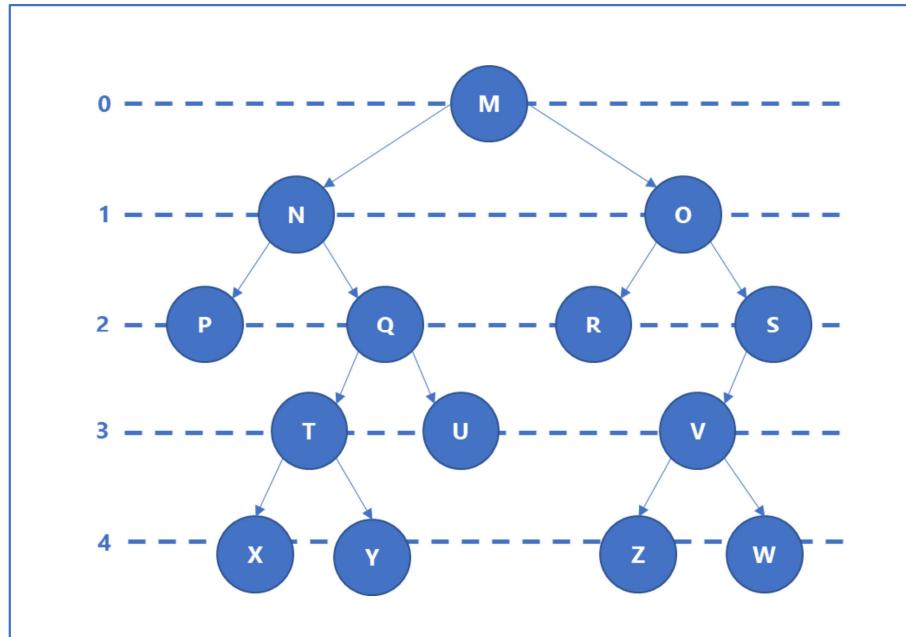
- **Ví dụ 7.9:**



**Hình 7.4.** Ví dụ cây nhị phân

### 7.2.2. Tính chất cây nhị phân

- Số nút nằm ở mức  $i \leq 2^i$ .
- Số nút lá  $\leq 2^{(h-1)}$ , với  $h$  là chiều cao của cây.
- Chiều cao của cây  $h \geq \log_2(N)$  ( $N$  là số nút trong cây)
- Số nút trong cây  $\leq 2^{(h-1)}$ .



**Hình 7.5.** Cây nhị phân với minh họa các mức

### 7.2.3. Biểu diễn cây nhị phân

- Để biểu diễn cây nhị phân, ta dùng cách cấp phát các vùng nhớ liên kết với nhau bằng con trỏ tương tự như biểu diễn danh sách liên kết.

- Cấu trúc 1 nút gồm:
  - Trường chứa dữ liệu của nút.
  - Con trỏ lưu địa chỉ nút gốc của cây con bên trái.
  - Con trỏ lưu địa chỉ nút gốc của cây con bên phải.



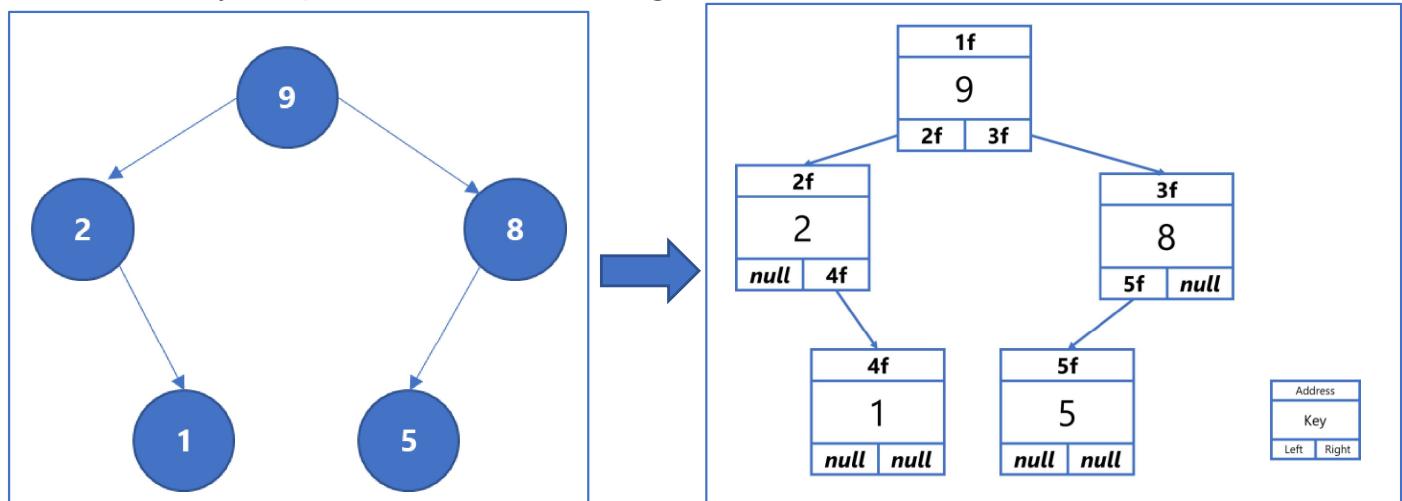
**Hình 7.6.** Mô hình cấu trúc 1 nút của cây nhị phân

Định nghĩa 1 nút của cây nhị phân bằng C++:

```
struct node {
    string data;
    node* left;
    node* right;
};
```

**Hình 7.7.** Cấu trúc một nút của cây nhị phân (lưu trữ số nguyên) được định nghĩa bằng C++.

- Cách cây nhị phân được lưu trữ trong bộ nhớ:



**Hình 7.8.** Mô tả cách cây nhị phân được lưu trữ trong bộ nhớ

#### 7.2.4. Các phép duyệt cây nhị phân

- Duyệt cây là một quá trình truy cập tất cả các nút của một cây và thao tác với các giá trị của cây này (thông thường là in ra giá trị tại các nút).
- Có 3 phép duyệt chính:

- Duyệt trước: Gốc – Trái – Phải (Node – Left – Right)
- Duyệt giữa: Trái – Gốc – Phải (Left – Node – Right)
- Duyệt sau: Trái – Phải – Gốc (Left – Right – Node)

Các phép duyệt cây cho độ phức tạp  $O(\log_2 h)$  với  $h$  là chiều cao của cây.

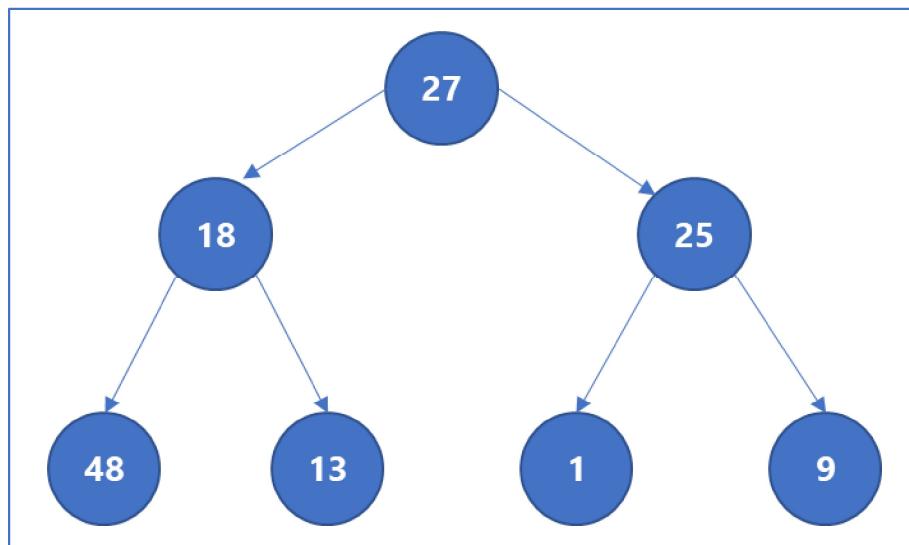
#### 7.2.4.1. Duyệt trước (Node – Left – Right)

- Trong phép duyệt này, nút gốc sẽ được truy cập đầu tiên, sau đó sẽ tiến hành truy cập cây con bên trái và cuối cùng là cây con bên phải. Tiếp tục áp dụng thứ tự duyệt này cho các cây con, ta sẽ truy cập được toàn bộ các nút của cây.
- Mô tả thuật toán duyệt trước (Node – Left – Right) bằng C++:

```
void NLR(tree& t){
    if(t!=NULL){
        cout<<t->data<<" ";
        NLR(t->left);
        NLR(t->right);
    }
}
```

#### - Ví dụ 7.10:

- Ta sẽ áp dụng thuật toán trên để duyệt trước (Node – Left – Right) cây sau:



- Ta kí hiệu:

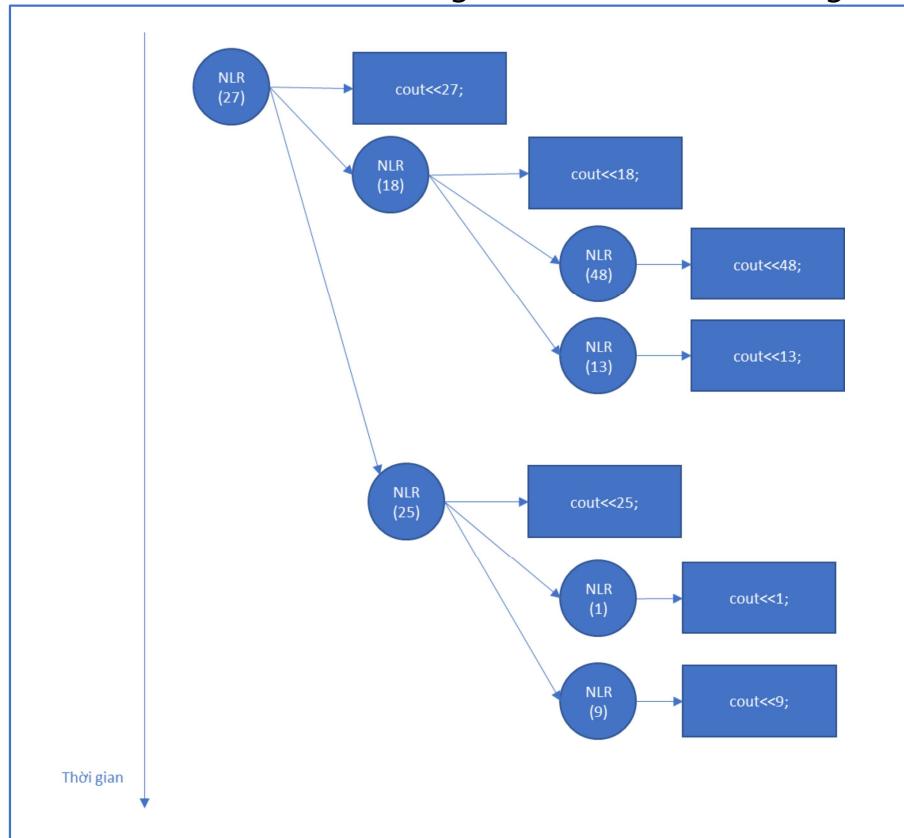
NLR  
(27)

: là thao tác truy cập vào nút có giá trị khóa là 27

`cout<<27;`

: là thao tác in giá trị của nút có giá trị khóa là 27 ra màn hình

- Ta có thể minh họa thuật toán trên bằng sơ đồ theo thứ tự thời gian như sau:



- Giải thích:

- Đầu tiên, nút 27 sẽ được truy cập, vì đây là phép duyệt trước (Node – Left – Right) nên khóa 27 sẽ được in ra trước.
- Tiếp đó, con trỏ T sẽ tiến hành truy cập cây con bên trái và cây con bên phải.
- Trước tiên, con trỏ T sẽ truy cập vào cây con bên trái, tức là nút 18 và tiến hành in giá trị khóa 18 ra.
- Theo như thuật toán được mô tả bằng C++, thì dòng lệnh `NLR(t->right)` sẽ được thực thi ngay sau dòng lệnh `NLR(t->left)`, cụ thể hơn `NLR(25)` sẽ được thực hiện ngay sau `NLR(18)`.
- Tuy nhiên, `NLR(25)` chính là một phép duyệt cây có nút gốc là nút 25, chứ không phải chỉ là thao tác in ra số 25, vậy nên phải tiến hành duyệt hết các cây con của nút 25, trước khi tiến hành duyệt cây con có nút gốc là nút 18.

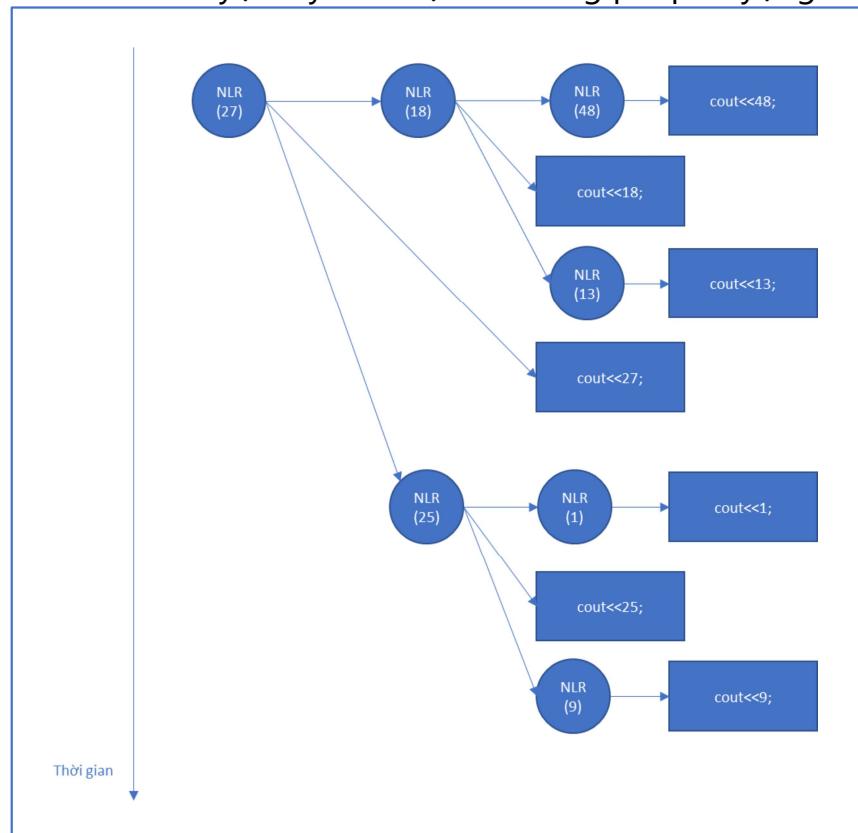
- Kết quả của phép duyệt in ra theo thứ tự thời gian, nút nào được duyệt và in ra giá trị khóa trước sẽ được hiển thị trước. Căn cứ theo sơ đồ duyệt theo thời gian như trên, ta có thứ tự duyệt các nút của cây trên theo thứ tự Node – Left – Right là: **27, 18, 48, 13, 25, 1, 9.**

#### 7.2.4.2. Duyệt giữa (Left – Node – Right)

- Trong phép duyệt này, cây con bên trái sẽ được truy cập đầu tiên, sau đó là nút gốc, và cuối cùng là cây con bên phải. Tiếp tục áp dụng thứ tự truy cập này cho các cây con, ta sẽ truy cập được toàn bộ các nút của cây.
- Mô tả thuật toán duyệt giữa (Left – Node – Right) bằng C++:

```
void LNR(tree& t){
    if(t!=NULL){
        LNR(t->left);
        cout<<t->data<<" ";
        LNR(t->right);
    }
}
```

- **Ví dụ 7.11:** Ta sẽ tiến hành duyệt cây ở Ví dụ 7.10 bằng phép duyệt giữa (LNR).





- Giải thích:
  - Đầu tiên, nút 27 sẽ được truy cập, vì đây là phép duyệt trước (Left – Node – Right) nên cây con bên trái của nút 27 sẽ được truy cập trước thao tác in ra khóa 27.
  - Cây con bên trái của nút 27 là cây có nút gốc là nút 18, cây có nút gốc là nút 18 có cây con bên trái, nên cây con bên trái này cũng sẽ được truy cập trước khi in ra khóa 18.
  - Cây con bên trái của nút 18 là cây có nút gốc là nút 48. Nút 48 này không có cây con trái nên sẽ không có phép duyệt cây con nào của nút 48, vậy nên sẽ tiến hành **in ra khóa 48**.
  - Sau khi in khóa 48, thì nút cha của nút 48 tức là **nút 18 sẽ được in ra**, trước khi truy cập cây con bên phải.
  - Sau khi đã in ra nút 18, thì cây con bên phải của nút 18 sẽ được truy cập. Cây con này có nút gốc là nút 13 và không có cây con, nên **nút 13 sẽ được in ra**.
  - Vậy là ta đã duyệt xong cây con bên trái của nút 27, nên **nút 27 sẽ được in ra**.
  - Tương tự, ta tiến hành duyệt cây con bên phải của nút 27, và lần lượt các nút được in ra theo thứ tự là **1, 25, 9**
- Kết quả của phép duyệt in ra theo thứ tự thời gian, nút nào được duyệt và in ra giá trị khóa trước sẽ được hiển thị trước. Căn cứ theo sơ đồ duyệt theo thời gian như trên, ta có thứ tự duyệt các nút của cây trên theo thứ tự Left – Node – Right là: **48, 18, 13, 27, 1, 25, 9**.

#### 7.2.4.3. Duyệt sau (Left – Right – Node)

- Ở phép duyệt này, nút gốc sẽ được duyệt cuối cùng, đầu tiên ta sẽ tiến hành duyệt cây con bên trái, tiếp theo là đến cây con bên phải và cuối cùng là duyệt nút gốc.
- Mô tả thuật toán duyệt sau (Left – Right – Node) bằng C++:

```
void LRN(tree& t){  
    if(t!=NULL){  
        LRN(t->left);  
        LRN(t->right);  
        cout<<t->data<<" ";  
    }  
}
```

(?) Câu hỏi:

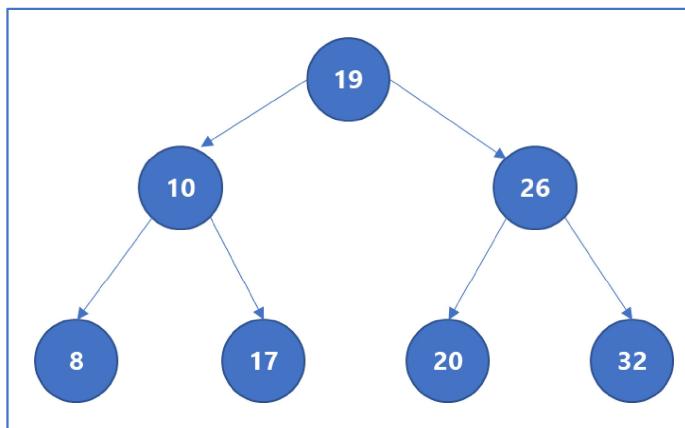
- i) Khi tiến hành duyệt cây ở Ví dụ 7.10, bạn có đoán trước được nút 27 sẽ được in ra ở vị trí nào không?
- ii) Hãy tiến hành duyệt sau cây ở Ví dụ 7.10.

### 7.3. Cây nhị phân tìm kiếm

### 7.3.1. Khái niệm

- Là cây nhị phân.
- Bảo đảm nguyên tắc bố trí khóa tại mỗi nút, sao cho:
  - Các nút trong cây con bên trái nhỏ hơn nút hiện hành.
  - Các nút trong cây con bên phải lớn hơn nút hiện hành.
- Ví kiểu dữ liệu được lưu trữ tại khóa của một nút trong cây rất đa dạng, bởi vì vậy phép so sánh “lớn hơn” và “nhỏ hơn” không đơn giản như phép so sánh  $>$  (lớn hơn) hay  $<$  (nhỏ hơn) của các số nguyên hay số thực. Ở trường hợp nếu muốn lưu trữ các kiểu dữ liệu phức tạp vào các nút của cây, các bạn phải tự định nghĩa phép so sánh giữa 2 giá trị khóa của 2 nút.

- **Ví dụ 7.12:**



**Hình 7.9.** Ví dụ cây nhị phân tìm kiếm

(?) *Câu hỏi: Nếu thêm 1 nút có giá trị khóa bằng 30 thì nút này sẽ ở vị trí nào?*

### 7.3.2. Các thao tác trên cây

Ta nhận thấy rằng cây con của một nút cũng là một cây nên các thao tác thực hiện với cây cha, cũng có thể thực hiện được với cây con. Do đó, một số thao tác với cấu trúc dữ liệu cây có thể thực hiện một cách dễ dàng bằng đệ quy.

#### 7.3.2.1. Tạo một cây rỗng

```

void createTree(tree& t)
{
    t=NULL;
}
  
```

- Cây rỗng có giá trị địa chỉ nút gốc bằng NULL.

(?) *Câu hỏi: Đây là thao tác đơn giản nhưng nếu không thực hiện thì chương trình sẽ có lỗi. Các bạn hãy giải thích tại sao?*



### 7.3.2.2. Tạo một nút có khóa bằng x

- Việc tạo một nút của cây tương tự như tạo một nút của danh sách liên kết.
- Thao tác bao gồm:
  - Cấp phát một vùng nhớ cho 1 nút (kiểu dữ liệu của nút đã được định nghĩa như mục 4.2.3)
  - Gán giá trị cho khóa của nút cần tạo
  - Gán giá trị cho địa chỉ của nút bên trái nút cần tạo (Nếu chưa xác định địa chỉ thì gán bằng NULL)
  - Gán giá trị cho địa chỉ của nút bên phải nút cần tạo (Nếu chưa xác định địa chỉ thì gán bằng NULL)
- Biểu diễn bằng C++:

```
node* createOneNode(string x){  
    node* p = new node;  
    if(p==NULL){  
        exit(1);  
    }  
    else{  
        p->data = x;  
        p->left = NULL;  
        p->right = NULL;  
    }  
    return p;  
}
```

### 7.3.2.3. Tìm 1 nút có khóa bằng x trên cây

- Ý tưởng: Xuất phát từ nút gốc của cây, ta sẽ tiến hành tìm vị trí của nút có khóa là x, bằng cách so sánh giá trị của nút hiện hành với x:
  - Nếu  $x <$  giá trị của nút hiện hành  $\Rightarrow$  Nút có khóa là x sẽ nằm ở cây con bên trái của nút hiện hành (nếu có). Ta sẽ tiến hành duyệt các nút ở cây con bên trái.
  - Nếu  $x >$  giá trị của nút hiện hành  $\Rightarrow$  Nút có khóa là x sẽ nằm ở cây con bên phải của nút hiện hành (nếu có). Ta sẽ tiến hành duyệt các nút ở cây con bên phải.
  - Nếu  $x =$  giá trị của nút hiện hành  $\Rightarrow$  Đây là vị trí cần tìm.
- Khi đã duyệt hết các vị trí có thể có (con trỏ duyệt có giá trị là null) mà vẫn không thấy nút nào có khóa là x  $\Rightarrow$  Không tồn tại nút có khóa là x trong cây.
- Hiện thực hóa ý tưởng trên bằng C++:

```
node* searchNode(tree& t, const string& x){
```



```
if(t==NULL){  
    return t;  
}  
else{  
    if(t->data==x){  
        return t;  
    }  
    else{  
        if(t->data>x){  
            return searchNode(t->left,x);  
        }  
        else{  
            return searchNode(t->right,x);  
        }  
    }  
}
```

**Hình 7.14.** Biểu diễn thao tác tìm 1 nút có khóa là x bằng C++

#### 7.3.2.4. Thêm một nút vào cây nhị phân tìm kiếm

- Điều kiện: Sau khi thêm nút vào cây, vẫn đảm bảo cây là cây nhị phân tìm kiếm.
- Ý tưởng: Ta cần xác định vị trí của nút cần thêm:
  - Nằm ở cây con bên phải của tất cả các nút có khóa nhỏ hơn nó.
  - Nằm ở cây con bên trái của tất cả các nút có khóa lớn hơn nó.
  - Là nút bên trái của nút không có cây con trái hoặc nút bên phải của nút không có cây con bên phải.
- Xuất phát từ nút gốc. Tiến hành duyệt cây để tìm vị trí thích hợp.
- So sánh giá trị khóa của nút cần thêm với khóa của nút hiện hành.
- Nếu nút hiện hành khác NULL:
  - Nếu khóa của nút cần thêm = khóa của nút hiện hành  
=> Thông báo đã tồn tại 1 nút có giá trị cần thêm trong cây.
  - Nếu khóa của nút cần thêm < khóa của nút hiện hành  
=> Tiến hành duyệt cây con bên trái của nút hiện hành để tìm vị trí thích hợp.
  - Nếu khóa của nút cần thêm > khóa của nút hiện hành  
=> Tiến hành duyệt cây con bên phải của nút hiện hành để tìm vị trí thích hợp.
- Nếu nút hiện hành bằng NULL: Tiến hành thêm khóa cần thêm vào cây vào nút hiện hành.

```
void insertNode(tree& t, const string& x){
```



```
if(t!=NULL){
{
    if(t->data==x){
        return;
    }
    else if (t->data>x) {
        insertNode(t->left,x);
    }
    else{
        insertNode(t->right,x);
    }
}
else{
    t=new node;
    if(t==NULL){
        exit(1);
    }
    else{
        t->data = x;
        t->left = t->right = NULL;
    }
}
}
```

#### **7.3.2.5. Xóa 1 nút có Key bằng x trên cây**

- Điều kiện: Sau khi xóa xong, cây vẫn là cây nhị phân tìm kiếm
  - Ý tưởng:
    - Đầu tiên, ta cần xác định được vị trí của nút cần xóa (nút chứa khóa x).
    - Tiếp theo ta thực hiện thay thế vị trí của nút cần xóa bằng một nút khác trong cây. Có các trường hợp sau xảy ra:

**TH1:** Nút cần xóa là nút lá.

Ở trường hợp này, ta xóa nút lá sẽ không gây ảnh hưởng đến cấu trúc cây.

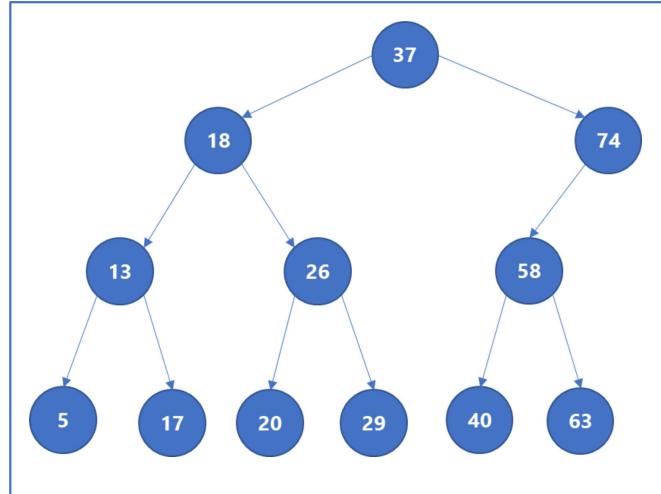
**TH2:** Nút cần xóa có duy nhất 1 cây con.

Ở trường hợp này, ta chỉ cần nối nút cha của X với nút con duy nhất của X và hủy nút X.

**TH3:** Nút cần xóa có đầy đủ 2 cây con.

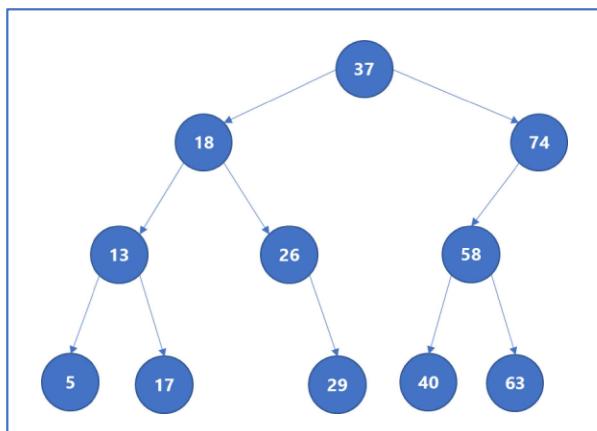
Ở trường hợp này, ta cần tìm nút thế mạng, nút này sẽ thay thế vị trí của nút cần xóa. Thông thường, ta sẽ chọn nút thế mạng là nút lớn nhất (phải nhất) của cây con bên trái, hoặc nút nhỏ nhất (trái nhất) của cây con bên phải.

- **Ví dụ 7.13:** Xét cây nhị phân tìm kiếm dưới đây, ta sẽ tiến hành xóa 3 nút của cây thuộc 3 trường hợp xóa đã nêu ở bên trên.



**Hình 7.16.** Minh họa thao tác xóa nút cây nhị phân tìm kiếm

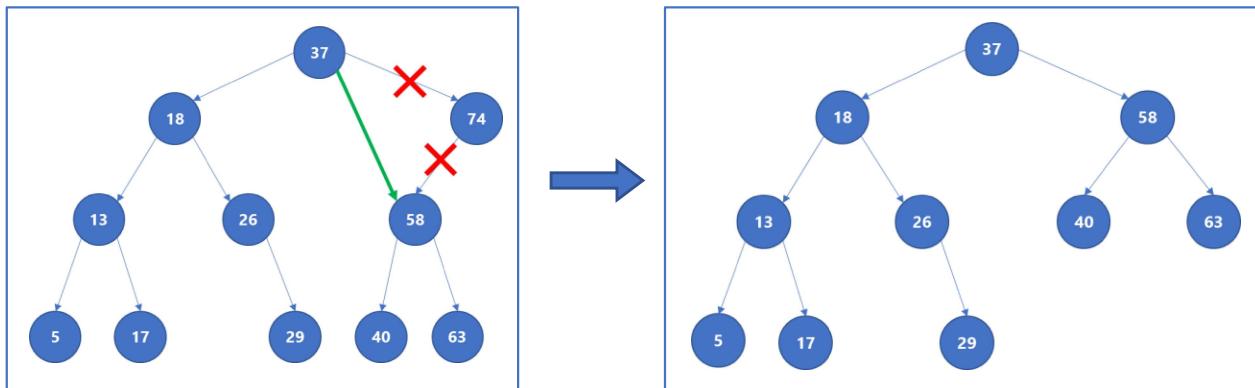
**TH1:** Xóa nút lá. Ta sẽ tiến hành xóa 1 nút lá bất kỳ, ở đây ta sẽ chọn xóa nút 20. Vì xóa nút lá sẽ không ảnh hưởng đến cấu trúc cây, nên việc xóa nút lá rất đơn giản, ta chỉ cần hủy nút đó đi. Cây sau khi xóa nút 20.



**Hình 7.17.** Minh họa thao tác xóa nút cây nhị phân tìm kiếm

**TH2:** Xóa nút có duy nhất 1 cây con. Ví dụ, ta cần xóa nút 74.

Ở trường hợp này, ta sẽ mốc nối nút con của nút cần xóa với nút cha của nút cần xóa. Ở đây nút cần xóa là nút 74. Nút cha của nút cần xóa là nút 37, nút con của nút cần xóa là nút 58.

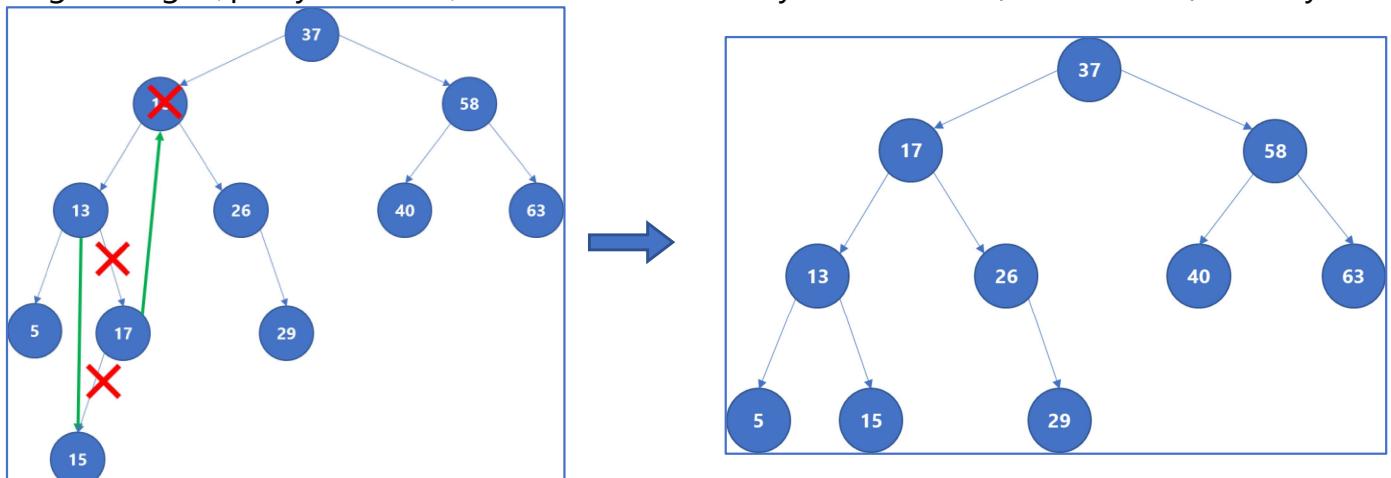


**Hình 7.18.** Minh họa thao tác xóa nút cây nhị phân tìm kiếm

**TH3:** Xóa nút có đầy đủ 2 cây con. Ví dụ ở đây ta sẽ xóa nút 18. Ta cần chọn nút thay thế, ở đây có 2 cách chọn:

1. Chọn nút thay thế là nút nhỏ nhất của cây con bên phải (tức là nút 26)
2. Chọn nút thay thế là nút lớn nhất của cây con bên trái (tức là nút 17)

Trong trường hợp này, ta sẽ chọn nút lớn nhất của cây con bên trái (tức là nút 17) để thay thế.



**Hình 7.19.** Minh họa thao tác xóa nút cây nhị phân tìm kiếm

- Minh họa ý tưởng xóa một nút khóa x bằng C++:
  - Hàm tìm nút thay thế:

```
void replaceNode(tree& p, tree& t){
    if(t->right != NULL){
        replaceNode(p,t->right);
        // Tìm nút phải nhất cây con trái
    }
    else{
        cout<<"Nut thay the: "<<t->data<<endl;
        p->data = t->data;
```



```
// Nút p là nút cần xóa (ta đã gán địa chỉ của t cho p ở hàm
deleteNodeX),
    // ta tiến hành gán lại data của t (nút thay thế) cho p (nút cần
xóa)
    // bây giờ ta chỉ cần xóa đi vùng nhớ của nút thay thế
    p=t;
    // Ta gán địa chỉ p bằng vị trí của nút thay thế
    t=t->left;
    // Nếu nút thay thế có cây con bên trái,
    // nút trái của nút thay thế sẽ trở thành nút phải của nút cha nút
thay thế
}
}
```

- Hàm xóa nút khóa x:

```
void deleteNodeX(tree& t, const string& x){
    if(t!=NULL){
        if(t->data<x){
            deleteNodeX(t->right,x);
        }
        else if(t->data>x){
            deleteNodeX(t->left,x);
        }
        else{
            node* p = t;
            // ta gán địa chỉ của t cho p
            // để sau khi tìm được nút thay thế cho t,
            // ta tiến hành cập nhật data của nút thay thế cho vị trí này
            if(t->left == NULL){
                t=t->right;
            }
            else if (t->right == NULL){
                t=t->left;
            }
            else {
                replaceNode(p,t->left);
            }
            delete p;
            //p bây giờ lưu địa chỉ của nút thay thế - không còn cần sử dụng
nữa
        }
    }
}
```

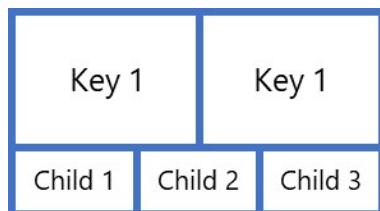
}

## 7.4. B-Tree

### 7.4.1. Định nghĩa

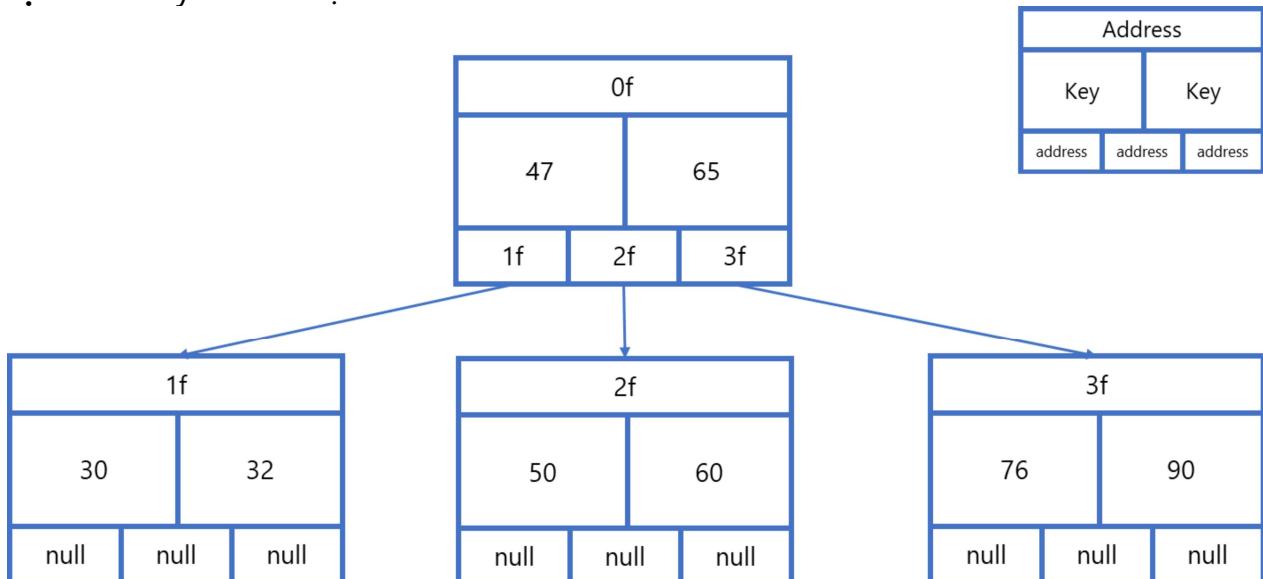
#### 7.4.1.1. Một nút:

- Gồm: khóa và các cây con.
- Số cây con = số khóa + 1.



**Hình 7.20.** Minh họa 1 nút của cây B-tree với 2 khóa và 3 cây con

- **Ví dụ 7.14:** Cây B-tree bậc 3:



**Hình 7.21.** Minh họa cấu trúc cây B-tree

#### 7.4.1.2. Các tính chất:

Cho số tự nhiên  $k > 0$ , B-Trees bậc  $m$  (với  $m \geq k$ ) là một cây nhiều nhánh tìm kiếm thỏa mãn các tính chất :

*i/* Tất cả các node lá ở cùng một mức.

*ii/* Tất cả các node trung gian trừ node gốc có tối đa  $m$  cây con ( $m-1$  khóa) và tối thiểu  $m/2 + 1$  cây con khác rỗng ( $m/2$  khóa).

*iii/* Mỗi node hoặc là node lá hoặc node có  $k$  khóa thì có  $k+1$  cây con và phân chia các khóa trong các nhánh con theo cách của cây tìm kiếm.

**iv/** Node gốc có nhiều nhất  $m$  cây con hoặc ít nhất có 2 cây con khi node gốc không là node lá hoặc không có cây con nào khi cây chỉ có 1 node gốc.

#### 7.4.1.3. Tips :

Cây B-tree bậc  $m$ :

- Mỗi nút sẽ chứa khóa và các cây con của nó, số cây con bằng số khóa + 1.
- Nút gốc:
  - Hoặc là NULL
  - Không có cây con nào nếu là nút lá
  - Có ít nhất 2 cây con (nút gốc chứa 1 khóa) nếu không là nút lá
  - Có nhiều nhất  $m$  cây con (nút gốc chứa  $m-1$  khóa)
- Nút trung gian:
  - Có ít nhất  $m/2$  cây con (nếu  $m$  chẵn) và  $(m+1)/2$  cây con (nếu  $m$  lẻ)
  - Có nhiều nhất  $m$  cây con
- Nút lá:
  - Tất cả các nút lá sẽ nằm cùng một mức
- Khóa:
  - Các khóa nằm trên các nút sẽ được sắp xếp theo thứ tự nhất định (thường sẽ theo thứ tự tương tự cây nhị phân tìm kiếm)

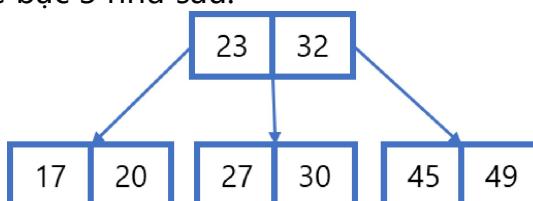
#### 7.4.2. Các thao tác với cây B-tree:

##### 7.4.2.1. Các khái niệm:

###### \*Thao tác tách node (split):

- Là thao tác sẽ được thực hiện khi thêm 1 khóa vào nút mà số khóa của nút sẽ vượt quá số khóa tối đa.
- Tiến hành lấy nút chính giữa đem gộp với nút cha của nút đang xét trở thành nút cha của 2 nút mới, 2 nút mới này chính là nút chứa các khóa bên trái và nút chứa các khóa bên phải của nút chính giữa.
- Nếu nút cha vừa được thêm khóa cũng vượt quá số khóa tối đa, ta sẽ tiếp tục thực hiện thao tác split (tách) cho nút đó.

- **Ví dụ 7.15:** Cho cây B-Tree bậc 3 như sau:



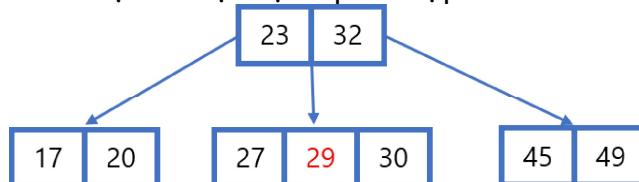
Đầu tiên ta xác định số cây con, số khóa tối đa, tối thiểu.

- Số cây con tối đa: 3
- Số cây con tối thiểu:  $(3+1)/2 = 2$

- Số khóa tối đa:  $3 - 1 = 2$
- Số khóa tối thiểu  $(3-1)/2 = 1$

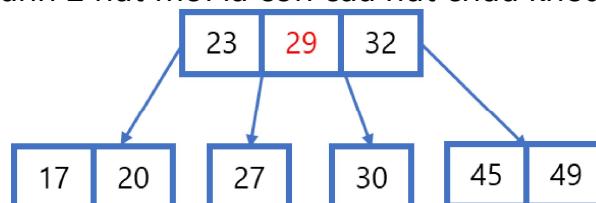
Hãy tiến hành thêm vào cây khóa có giá trị 29 sao cho cây vẫn là cây B-Tree bậc 3.

Nhìn vào cây B-Tree, thì ta xác định được vị trí phù hợp với số 29 chính là giữa nút 27, 30.

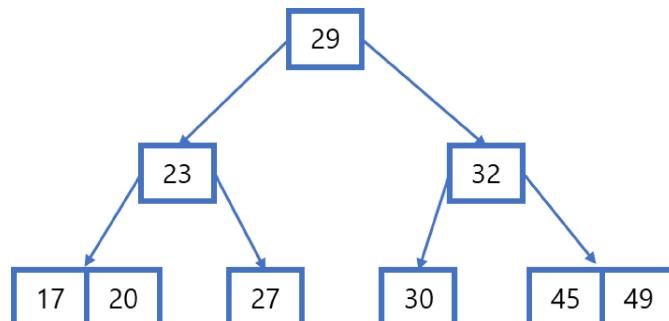


Tuy nhiên, khi ta thêm khóa 29 vào nút này thì số khóa sẽ vượt quá số khóa tối đa là 3.

Vậy nên ta sẽ tiến hành thao tác split node (tách). Khóa 29 sẽ gộp với nút cha của nút đang xét, khóa 27, 30 sẽ tách thành 2 nút mới là con của nút chứa khóa 29.



Tuy nhiên, khi khóa 29 nằm ở nút 23, 32 thì số khóa của nút này cũng vượt quá số khóa tối đa, nên sẽ tiếp tục thực hiện thao tác split (tách). Cây B-Tree sau khi thực hiện xong thêm khóa 29.

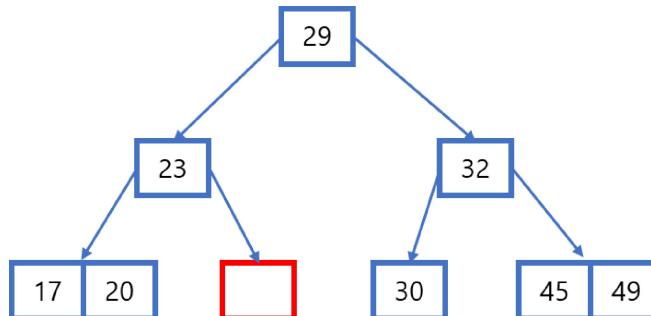


**Hình 7.22.** Cây B-Tree bậc 3 sau khi hoàn tất thêm khóa 29.

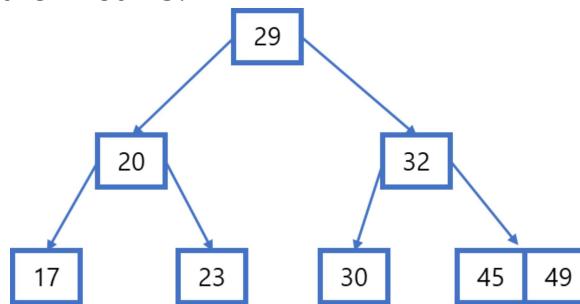
#### \*Thao tác nhường khóa (underflow)

- Nếu nút sau khi xóa bớt khóa, có số khóa nhỏ hơn số khóa tối thiểu trong 1 nút, thì ta sẽ xét các nút kề của nút này.
- Nếu có nút kề có số khóa lớn hơn số khóa tối thiểu trong 1 nút, ta sẽ tiến hành nhường khóa của nút kề này cho nút đang thiếu khóa.
- Khóa được nhường sẽ lên thế chỗ 1 khóa của nút cha, khóa của nút cha sẽ vào vị trí của nút bị thiếu khóa.

- **Ví dụ 7.16:** Xét cây ở Hình 7.22. Hãy tiến hành xóa nút 27 khỏi cây.



- Khi xóa nút lá 27 khỏi cây, ta thấy nút hiện hành có 0 khóa, ít hơn số khóa tối thiểu.
- Ta tiến hành xét nút kề của nút 27 là nút (17, 20). Nút này có số khóa nhiều hơn số khóa tối thiểu, nên sẽ tiến hành thao tác underflow. Khóa 23 sẽ xuống thay thế cho khóa 27. Khóa 20 sẽ lên thay thế khóa 23.



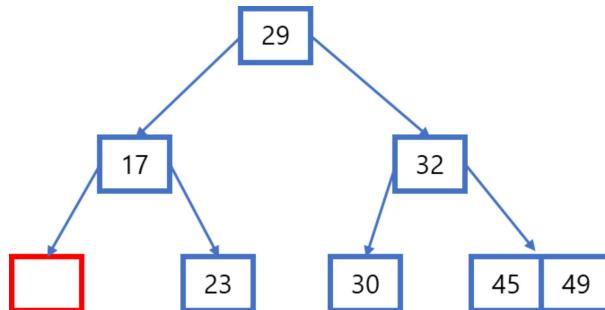
**Hình 7.23.** Cây B-Tree sau khi xóa khóa 27

#### \*Thao tác hợp (catenate)

- Trong trường hợp không có nút kề nào có số khóa lớn hơn số khóa tối thiểu, ta sẽ tiến hành thao tác **catenate**.
- Thao tác **catenate** là thao tác, gom các khóa của nút đang bị thiếu khóa, với nút cha và nút anh chị của nó (nút có chung khóa cha) trở thành 1 nút. Khi này, nút cha của nút bị thiếu khóa sẽ mất đi một nút, và các khóa và nút được gom sẽ trở thành nút con mới của nút cha này.
- Ở nút cha có 1 khóa bị mất, 2 cây con bị mất, và 1 cây con mới được tạo. Vậy nút cha mất đi 1 khóa và 1 cây con. Nghĩa là số cây con vẫn đảm bảo hơn số khóa 1. Cấu trúc cây vẫn được đảm bảo.
- Trong trường hợp, sau khi **catenate**, nút cha của nút vừa bị xóa khóa không đảm bảo số khóa tối thiểu => Xem xét để thực hiện thao tác **underflow**, nếu không thực hiện được thao tác **underflow** thì ta sẽ thực hiện thao tác **catenate** cho nút này. Cứ thế đến khi nào không có nút nào bị thiếu số khóa tối thiểu, hoặc nút vừa gom trở thành nút gốc thì dừng việc **catenate**.

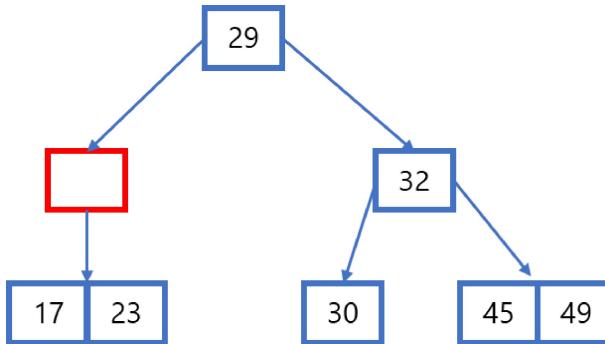
- **Ví dụ 7.17:** Xét cây ở **Hình 7.23** Hãy xóa khóa 20 khỏi cây.

Khi xóa khóa 20, là nút trung gian. Ta chọn 17 là khóa thế mạng của nó. Việc xóa khóa 20 quy về xóa khóa của nút lá vị trí nút khóa 17.



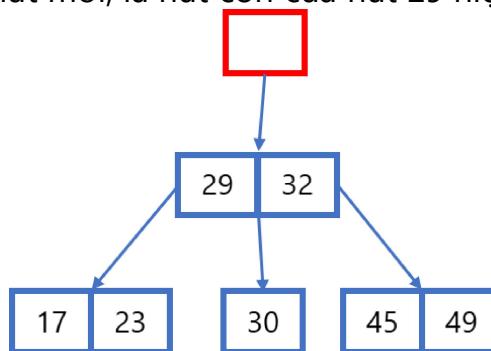
Ở vị trí nút lá vừa bị xóa khóa để làm khóa thế mạng, số khóa của nút hiện tại là 0, nhỏ hơn số khóa tối thiểu. Tuy nhiên, nút kề duy nhất của nút này là nút chứa khóa 23, chỉ có 1 khóa. Nên không thể thực hiện **underflow**. => Ta thực hiện thao tác **catenate**.

Ta tiến hành gom tất cả khóa nút kề của nút đang thiếu khóa và nút cha thành một nút mới. Tức là ta gom 17, 23 thành 1 nút mới. Và làm con của nút cha hiện tại là nút chứa khóa 17. Nút 17 mất đi 1 khóa.

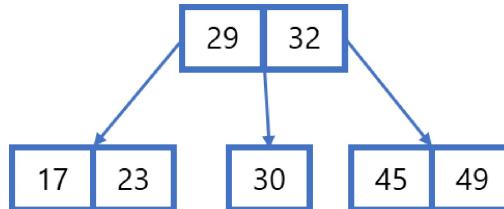


Tuy nhiên nút cha này, tức là nút chứa khóa 17 trước đó, lại không có khóa nào, và nút kề của nó là 32 cũng chỉ có 1 khóa, không nhiều hơn số khóa tối thiểu, nên không thể thực hiện thao tác **underflow**. Ta tiếp tục tiến hành **catenate** cho nút này.

Gom 29, 32 tạo thành nút mới, là nút con của nút 29 hiện tại.



Tuy nhiên nút cha lúc này không có khóa nào, tuy nhiên vì đây là nút gốc, nên chúng ta không cần thực hiện thao tác **underflow** hay **catenate**, mà chỉ cần cho nút (29, 32) trở thành nút gốc mới. Cây sau khi hoàn tất xóa nút 20:



#### 7.4.2.2. Xác định số cây con, số khóa tối đa, tối thiểu:

Các bước tạo cây B-tree với bậc m:

- Số cây con tối đa: m
- Số cây con tối thiểu (cho nút trung gian):  $m/2$  nếu m chẵn,  $(m+1)/2$  nếu m lẻ
- Số khóa tối đa:  $m-1$
- Số khóa tối thiểu: số cây con tối thiểu – 1.

- **Ví dụ 7.15:** Xác định số cây con tối đa, tối thiểu (cho nút trung gian), số khóa tối đa, số khóa tối thiểu (trừ nút gốc) của cây B-tree bậc 5.

- Số cây con tối đa: 5
- Số cây con tối thiểu (cho nút trung gian): 3
- Số khóa tối đa: 4
- Số khóa tối thiểu (trừ nút gốc): 2

#### 7.4.2.3. Thêm nút vào cây B-tree:

##### Thuật toán thêm khóa:

- Xác định vị trí của khóa cần thêm
- Thêm khóa vào nút
- Nếu số khóa vượt quá số khóa tối đa, thực hiện thao tác split (tách)
- Nếu khóa được đưa lên làm khóa cha, nằm ở nút có số khóa vượt quá số khóa tối đa khi thêm nút vừa tách, ta tiếp tục thực hiện thao tác split (tách) cho nút này.

**Ví dụ 7.16:** Hãy vẽ cây B-tree bậc 5, thêm vào cây các khóa theo thứ tự sau: **3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56**

Xác định số cây con, số khóa tối đa, tối thiểu:

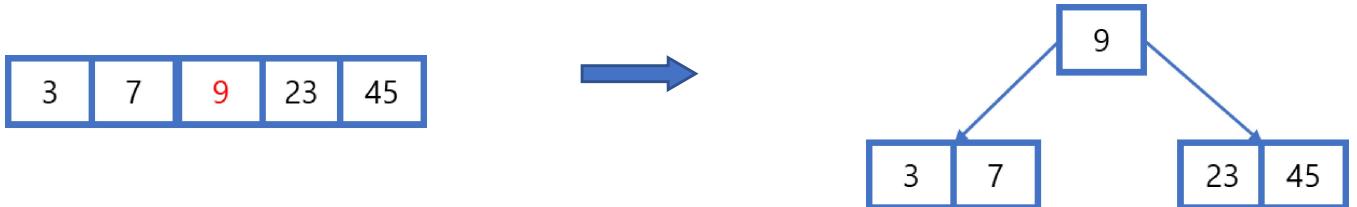
- Số cây con tối đa: 5
- Số cây con tối thiểu:  $(5+1)/2 = 3$
- Số khóa tối đa:  $5 - 1 = 4$
- Số khóa tối thiểu:  $3 - 1 = 2$

Thêm các khóa 3, 7, 9, 23:

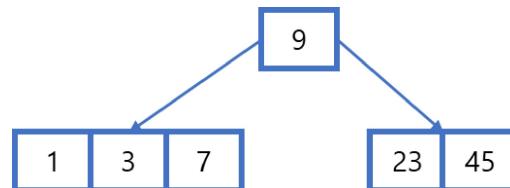
- Vì số khóa tối đa của 1 nút là 4, nên 4 khóa đầu tiên đều nằm ở nút gốc. Cây B-tree sau khi thêm 4 khóa 3, 7, 9, 23



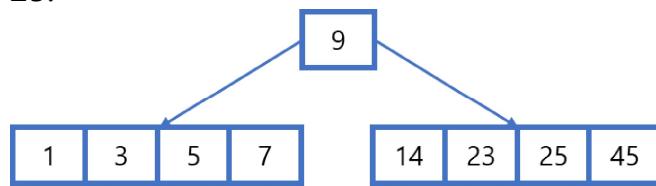
Thêm khóa 45, khi thêm khóa 45 vào nút hiện hành thì số nút sẽ vượt quá số khóa tối đa của 1 nút, vậy nên thao tác Split (tách) sẽ được thực hiện:



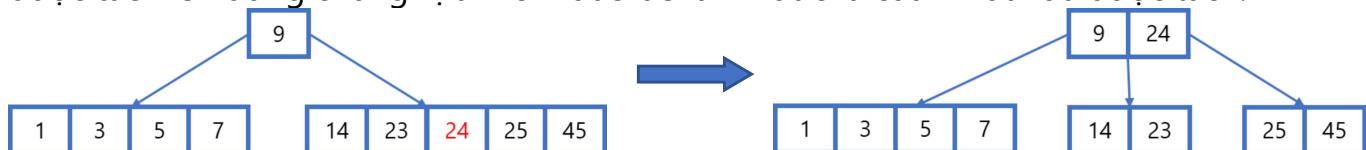
Thêm khóa 1, vì số khóa hiện tại của nút chứa 3, 7 < số khóa tối đa. Nên có thể thêm khóa 1 vào nút đó.



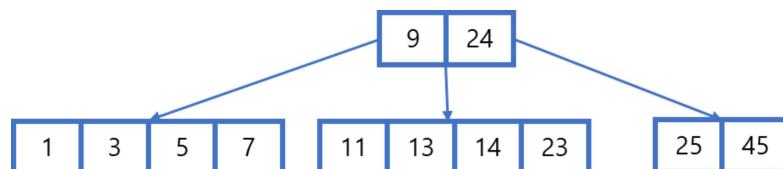
Tương tự với khóa 5, 14, 25:



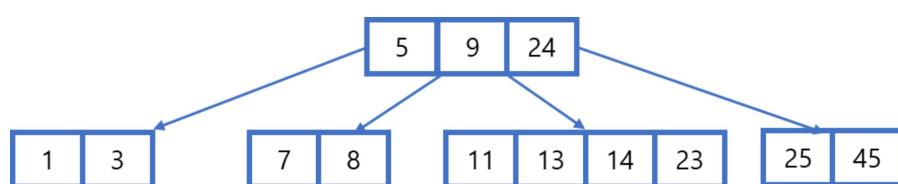
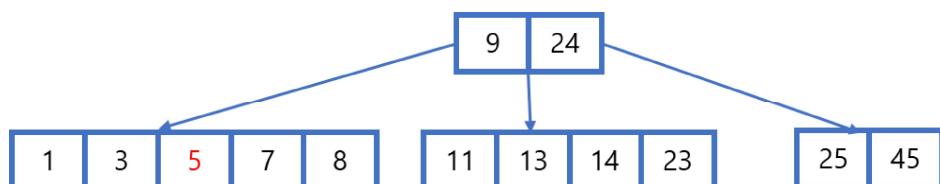
Thêm khóa 24: tương tự như khóa 45, số khóa ở nút 14, 23, 25, 45 đã đạt tối đa nên thao tác split sẽ được thực hiện. Lúc này (14, 23) và (25, 45) sẽ được tách thành nút mới, khóa 24 sẽ được tách lên đứng chung vị trí với nút 9 để làm nút cha của 2 nút vừa được tách.



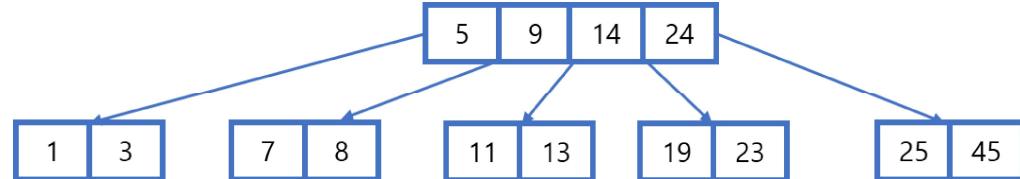
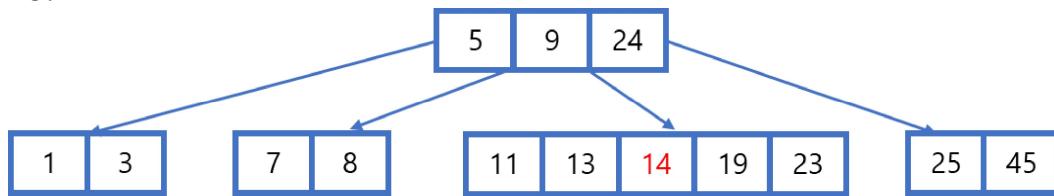
Thêm khóa 13, 11:



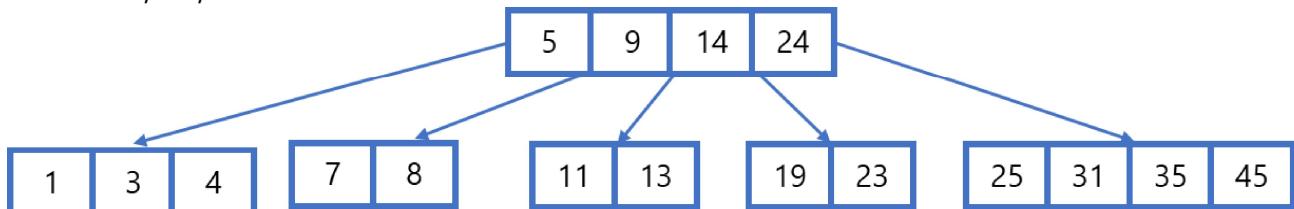
Thêm khóa 8:



Thêm khóa 19:

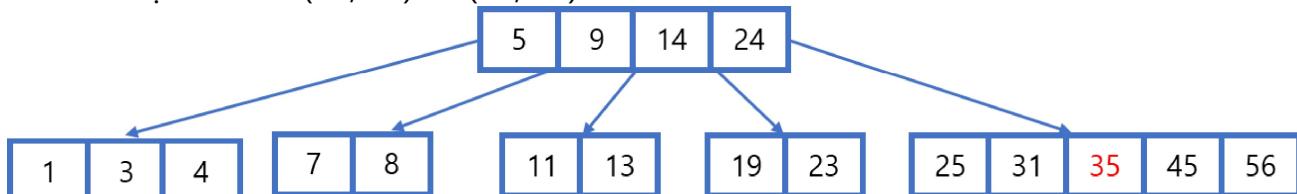


Thêm khóa 4, 31, 35:

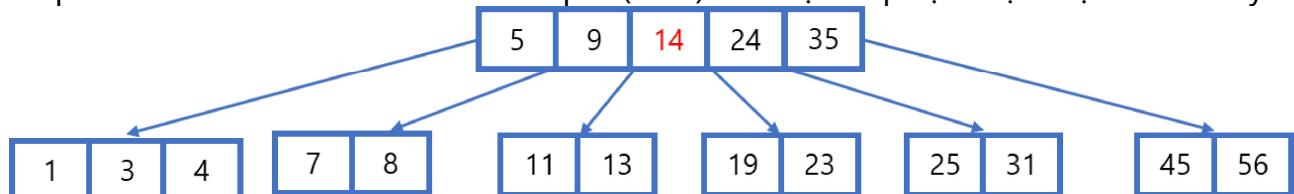


Thêm khóa 56:

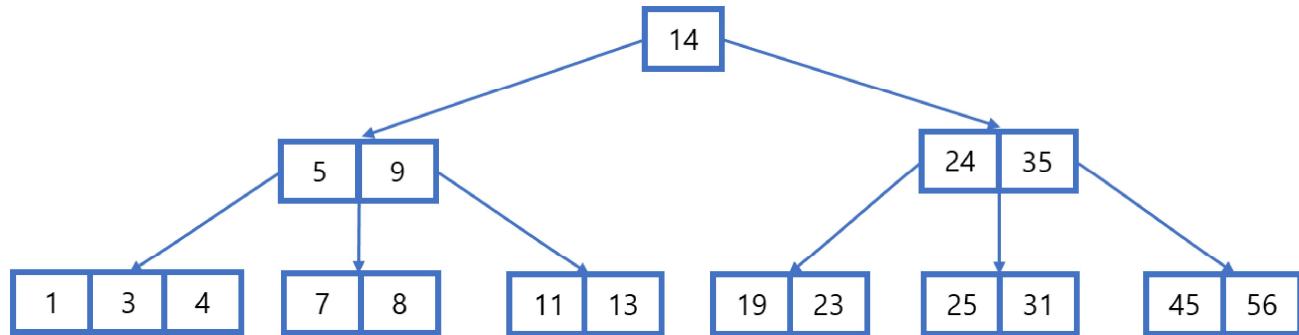
- Khóa 56 sau khi thêm vào nút (25, 31, 35, 45) thì tổng số khóa của nút này sẽ vượt quá số khóa tối đa trên 1 nút. Vậy nên nút chính giữa là 35 sẽ được tách lên làm cha của 2 nút được tách là (25, 31) và (45, 56)



- Khi khóa 35 được tách lên nằm trong nút 5, 9, 14, 24 thì số khóa của nút này lại vượt quá số khóa tối đa. Nên thao tác split (tách) sẽ được tiếp tục thực hiện ở nút này.



- Khóa chính giữa là 14 sẽ được tách lên làm cha của 2 nút được tách ra là (5, 19) và (24, 35)



Hình 7.22. Cấu trúc cây B-tree sau khi thêm các nút

#### 7.4.2.4. Xóa nút trên cây B-Tree

##### \*Thuật toán xóa khóa:

- Xóa khóa tại mọi nút được quy đổi về xóa khóa trong một nút lá. Vì khi xóa 1 nút trung gian, ta sẽ tìm nút thế mạng tương tự như khi ta xóa nút trung gian trong cây nhị phân tìm kiếm (Sẽ là nút trái nhất các cây con bên phải, hoặc nút phải nhất trong các cây con trái). Lúc này thì số lượng các khóa tại nút trung gian không bị thay đổi, và nút lá thì bị mất một khóa, nên ta có thể xem như việc xóa 1 khóa của nút lá.
- Đặt  $k = (m-1)/2$  nếu  $m$  lẻ, và  $k = m/2 - 1$  nếu  $m$  chẵn (Chương trình UIT chỉ xét cây B-tree bậc lẻ).
- Nếu nút lá chứa khóa cần xóa có nhiều hơn  $k$  khóa, thực hiện xóa khóa đó mà không sợ ảnh hưởng đến cấu trúc cây.
- Nếu nút lá chỉ chứa  $k$  khóa, sau khi xóa chỉ còn  $k-1$  khóa, ta thực hiện thao tác **underflow**:

i) Nếu một nút kề nút vừa mất khóa có nhiều hơn  $k$  khóa, chuyển khóa đó sang cho nút vừa xóa đi 1 khóa.

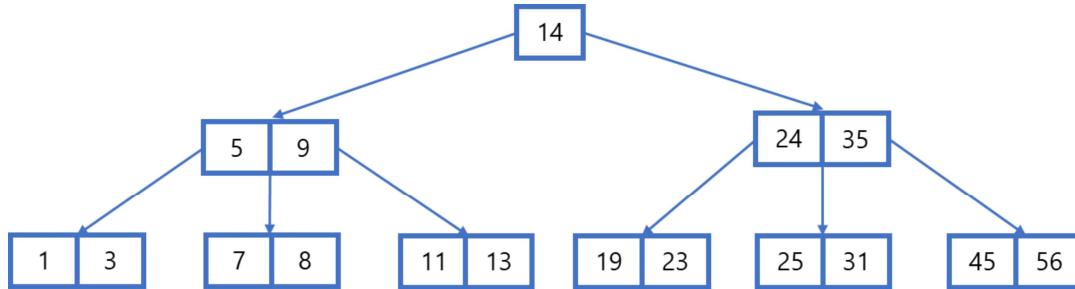
ii) Ngược lại, thực hiện **catenate** với một nút kề.

Lặp lại thao tác **underflow** như trên cho nút trung gian nếu nút trung gian có ít hơn  $k$  khóa.

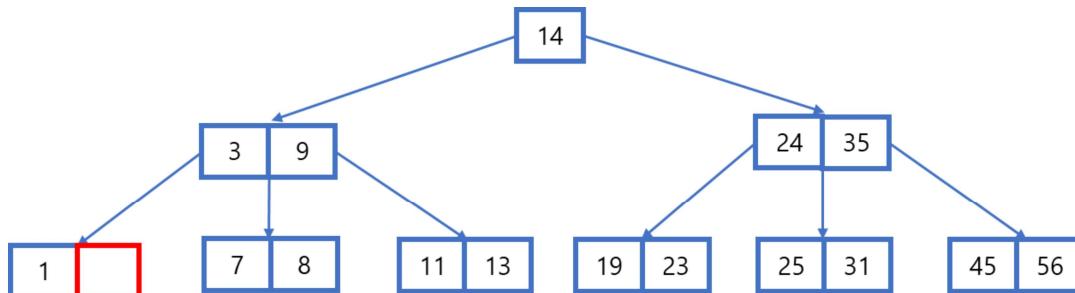
- Nếu chúng ta thực hiện **catenate** đến nút gốc và nút gốc chỉ còn lại 1 nút con, thì cho nút con làm nút gốc mới.

**- Ví dụ 7.17:** Hãy xóa lần lượt các khóa: 4, 5, 7, 3, 14 khỏi cây B-tree trong Hình 7.22.

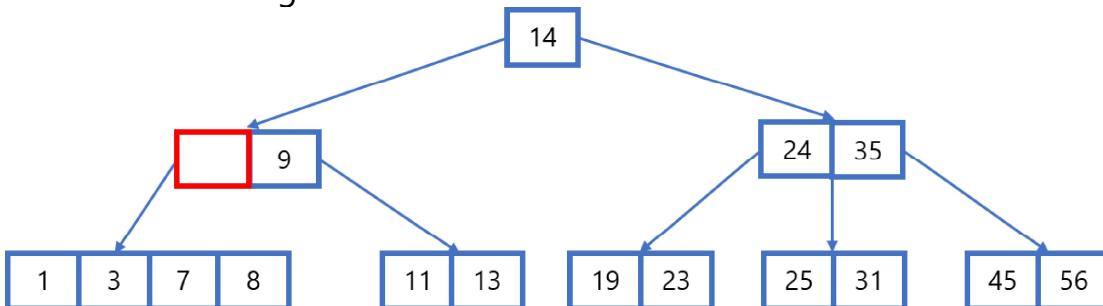
- Đầu tiên, ta xác định  $k = (5-1)/2 = 2$
- Nút hiện tại chứa khóa 4 là nút lá, có  $3 > k$ . Nên ta chỉ cần xóa khóa 4 mà không sợ ảnh hưởng đến cấu trúc cây. Cây sau khi xóa khóa 4:



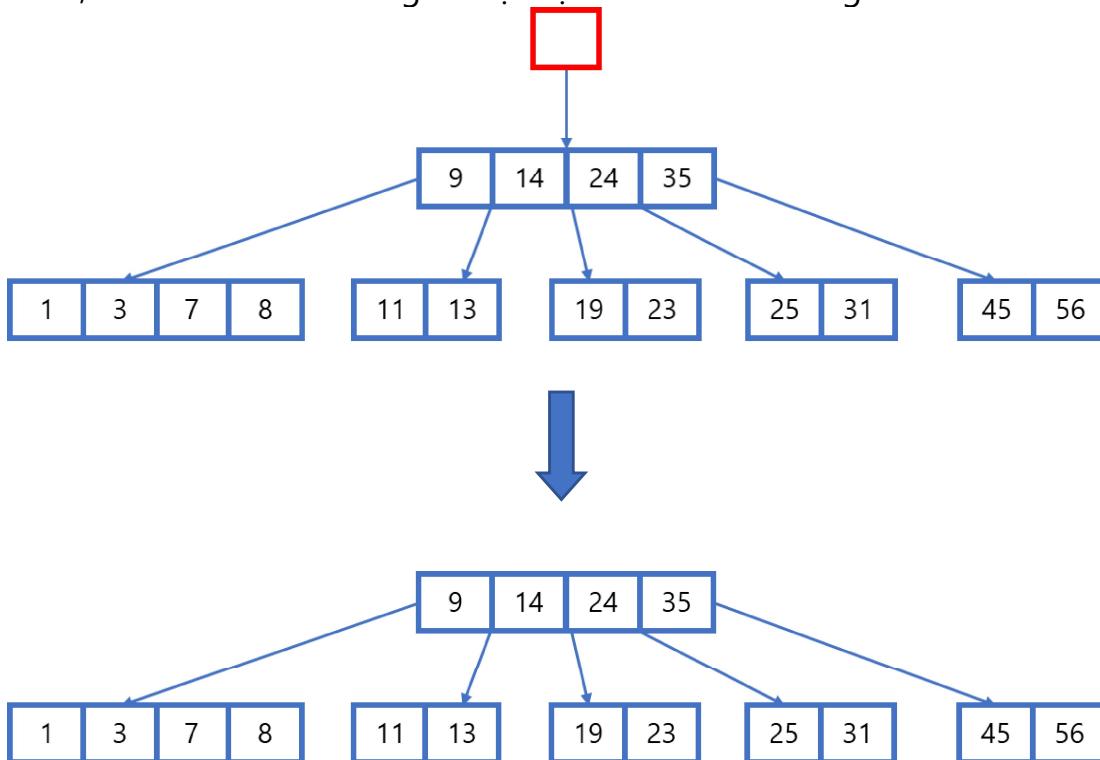
- Xóa khóa 5: Vì khóa 5 nằm ở nút trung gian, nên ta tìm khóa thế mạng, có thể chọn khóa 3 (phải nhất cây trái) hoặc 7 (trái nhất cây phải). Ở đây, giả sử ta chọn 3 là khóa thế mạng.
- Sau khi đẩy khóa 3 lên thay thế, ta quy việc xóa khóa trung gian về xóa 1 khóa ở nút lá (1,3).



- Và nút lá (1, 3) sau khi đem khóa 3 lên thế mạng thì số khóa không đảm bảo số lượng khóa tối thiểu trong 1 nút của cây bậc 5. Xét nút kề của nút 1 là nút (7, 8) chỉ có 2 khóa (không nhiều hơn k khóa) nên không thể thực hiện **underflow**.
- Vì vậy, ta phải thực hiện **catenate**.
- Ta tiến hành gom các nút anh chị của nút 1 (có chung cha) và khóa cha của nó thành 1 nút mới (ta quy ước khóa 3 là khóa cha của nút 1 và nút (7, 8)), và các nút được gom lại thành 1 nút sẽ làm con của nút chứa khóa cha. (Trong ví dụ đang xét, nút 1 chỉ có 1 khóa cha duy nhất là 3, và 1 nút anh chị duy nhất là (7, 8), nên chỉ có 1 cách gom duy nhất là gom 1, 3, 7, 8 thành 1 nút. Nếu gặp trường hợp có nhiều hơn 1 lựa chọn, tức nút 1 có 2 khóa cha và 2 nút anh chị, thì tùy theo yêu cầu xóa của đề bài để chọn gom bên nhánh anh chị nào (nếu không nói gì thêm thì cứ chọn 1 bên trái hoặc phải để gom). Như trong ví dụ trên, nút 1 chỉ có nhánh anh chị bên phải nên ta sẽ gom nó thành 1 nút mới). Và sau khi gom:



- Sau khi gom, nút 9 chỉ có 1 khóa, không đủ số khóa tối thiểu. Xét các nút kề của nút 9 thì không có nút nào có nhiều hơn k khóa (2 khóa) nên không thể thực hiện **underflow**. Vậy nên ta sẽ tiến hành **catenate**, gom 9, 14, 24, 35 thành 1 nút. Lúc này, khóa của nút gốc đã bị gom lại thành nút mới (9, 14, 24, 35) nên số khóa của nút gốc chỉ còn là 0, nhưng vì nút gốc không còn khóa nào nên không cần phải thực hiện underflow hay catenate, mà nút con của nút gốc hiện tại sẽ trở thành nút gốc.



## 7.5. Bài tập

**BT 7.5.1** Cho dãy số sau: **11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31**

Hãy thực hiện các yêu cầu sau:

- Xây dựng cây nhị phân tìm kiếm từ dãy số đã cho vào cây theo thứ tự thêm các số từ trái sang phải của dãy số.
- Duyệt cây trong câu a theo **NLR, RLN**.
- Xóa khỏi cây lần lượt các nút 8, 11, 43, 6 (vẽ hình từng trường hợp) sao cho cây vẫn là cây nhị phân tìm kiếm sau khi xoá nút.
- Viết hàm in ra màn hình các nút trên cây có duy nhất một nút con.
- Viết hàm đếm số lượng nút lá có trên cây.

**BT 7.5.2** Cho dãy ký tự như sau: **F, D, B, A, C, E, H, G, I**

Hãy thực hiện các yêu cầu sau:

- Vẽ cây nhị phân tìm kiếm bằng cách thêm lần lượt từng ký tự vào cây theo thứ tự từ trái qua phải của dãy ký tự trên, biết rằng giá trị của từng ký tự tương ứng



theo thứ tự xuất hiện của ký tự trong từ điển.

- b. Cho biết kết quả duyệt cây theo **RNL, NRL**.
- c. Huỷ lần lượt từng nút **D, E, F, H** trên cây, mỗi lần huỷ 1 nút vẽ lại cây nối tiếp theo như thứ tự huỷ.
- d. Viết hàm đếm số lượng nút có một nút con trên cây, nếu cây rỗng thì in ra giá trị -1.

**BT 7.5.3** Giả sử cho thông tin một sinh viên bao gồm các thông tin:

- Mã sinh viên: dạng số nguyên dương, mã sinh viên là giá trị duy nhất để phân biệt
- Họ sinh viên: dạng chuỗi
- Tên sinh viên: dạng chuỗi
- Điểm trung bình học tập: dạng số thực, có miền giá trị từ 0.0 đến 10.0

Hãy thực hiện các yêu cầu sau:

- a. Khai báo cấu trúc cây nhị phân tìm kiếm để lưu thông tin sinh viên theo mô tả ở trên.
- b. Viết hàm thêm các sinh viên vào cây.
- c. Viết hàm xóa các sinh viên có điểm trung bình < 5.0 ra khỏi cây.
- d. Viết hàm hiển thị danh sách sinh viên khi duyệt cây theo thứ tự trước.

**BT 7.5.4** Hãy tạo cây B-Tree bậc 3:

- a. Lần lượt thêm các khóa **A, D, Z, B, F, G, H, O, N, P, X, C** vào cây. Và cho biết ở thao tác nào thì có thao tác **split node**.
- b. Lần lượt xóa các khóa **P, D, F, C** khỏi cây. Xóa khóa nào thì chỉ cần thực hiện thao tác **underflow**, khóa nào thì phải thực hiện **catenate**.

**BT 7.5.5** Hãy tạo cây B-Tree bậc 5:

- a. Lần lượt thêm các khóa **C, N, G, A, H, E, K, Q, M, F, W, L, T, Z, D, P, R, X, Y, S** vào cây. Và cho biết ở thao tác nào thì có thao tác **split node**.
- b. Lần lượt xóa các khóa **H, T, R, E, C** khỏi cây. Xóa khóa nào thì chỉ cần thực hiện thao tác **underflow**, khóa nào thì phải thực hiện **catenate**.