CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

CHƯƠNG 5

CÂY (tree)

GV: Bùi Văn Thượng
Email: thuong.bui@ut.edu.vn

NỘI DUNG CHÍNH

TỔNG QUAN VỀ CẤU TRÚC CÂY

CÂY NHỊ PHÂN

CÂY NHỊ PHÂN TÌM KIẾM

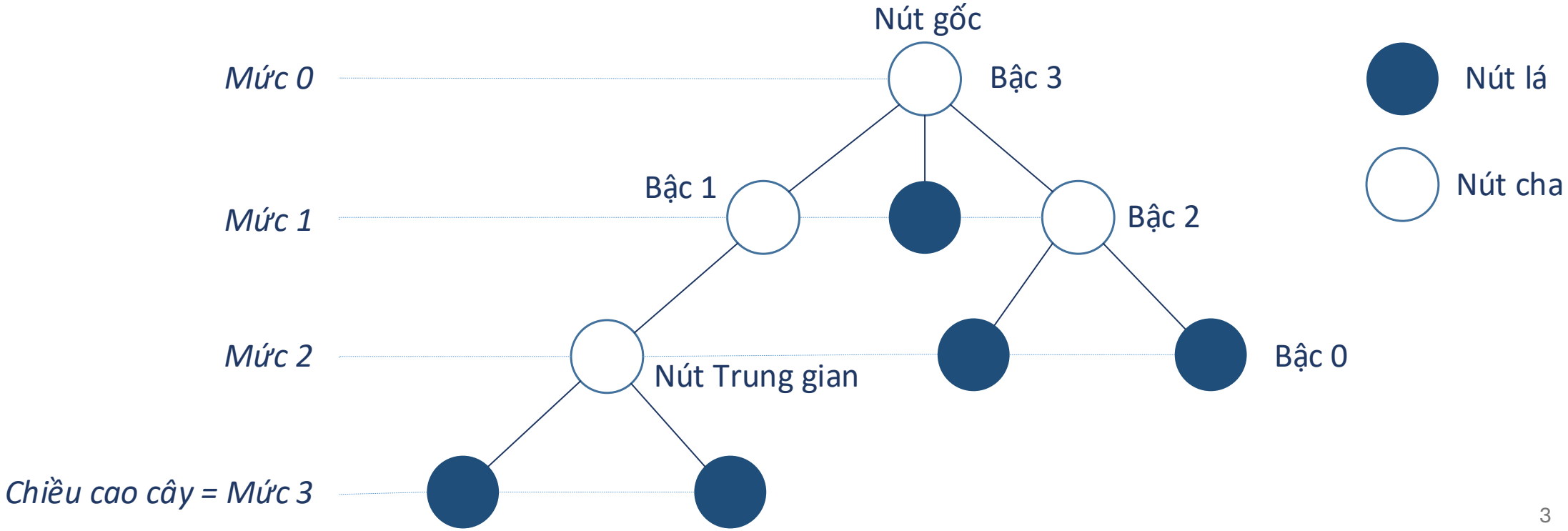
CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

B - TREE

1 TỔNG QUAN VỀ CẤU TRÚC CÂY

Định nghĩa cây:

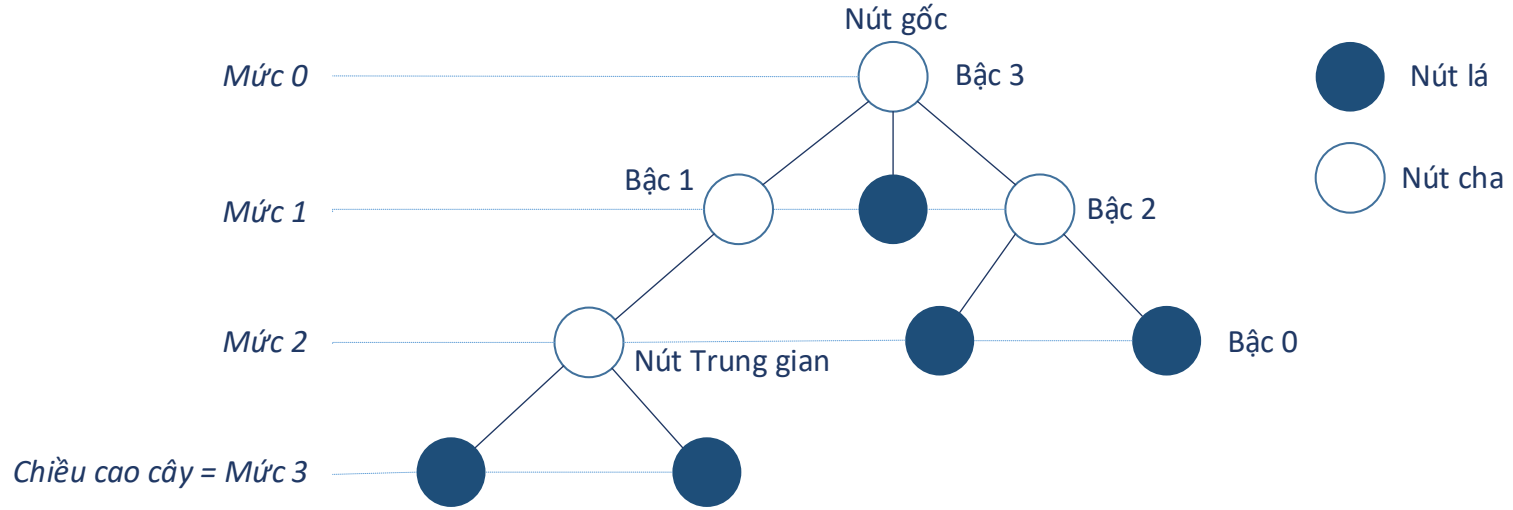
Cây là một tập hợp T các phần tử (gọi là nút của cây), trong đó có một nút đặc biệt gọi là nút gốc, các nút còn lại được chia thành những tập rời nhau T_1, T_2, …,T_n theo quan hệ phân cấp, trong đó T_i cũng là 1 cây. Mỗi nút ở cấp i sẽ quản lý một số nút ở cấp i+1. Quan hệ này người ta gọi là quan hệ cha – con.



1 TỔNG QUAN VỀ CẤU TRÚC CÂY

Một số khái niệm cơ bản:

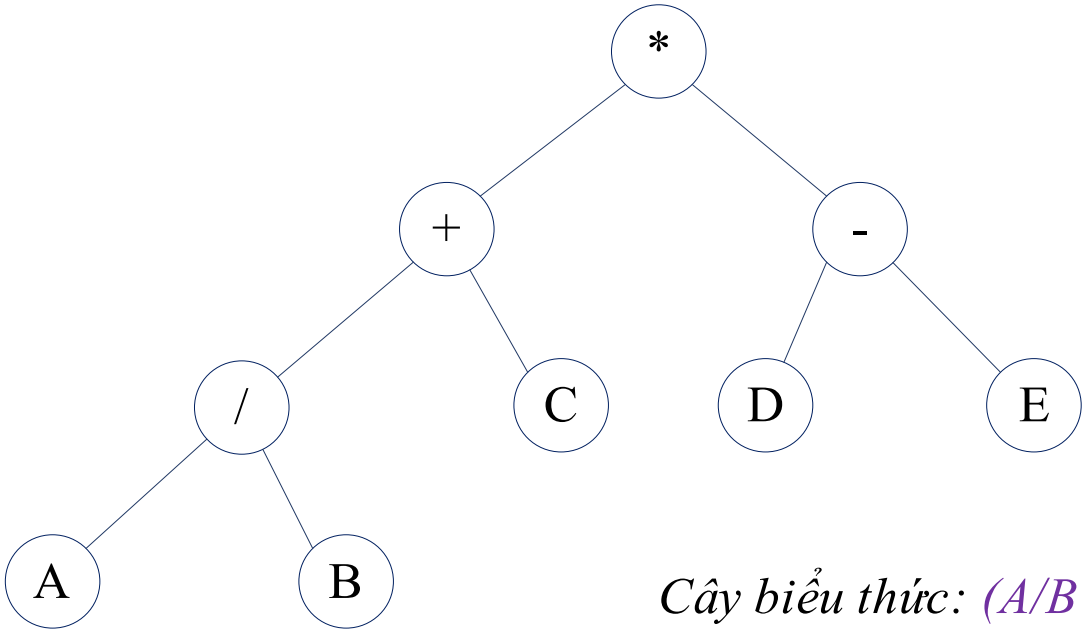
- Bậc của một nút: Là số cây con của nút đó.
- Bậc của một cây: Là bậc lớn nhất của các nút trong cây.
- Nút gốc: Là nút không có nút cha.
- Nút lá: Là nút có bậc bằng 0.
- Nút trung gian: Là nút có bậc khác không và không phải nút gốc.
- Mức của một nút: Mức của nút gốc bằng 0, mức của các nút con bằng mức của nút cha cộng thêm 1.
- Độ dài đường đi từ gốc đến nút x: Là số nhánh cần đi qua kể từ gốc đến x.
- Chiều cao của cây: Là mức lớn nhất trong cây.



1 TỔNG QUAN VỀ CẤU TRÚC CÂY

Một số ví dụ về cây:

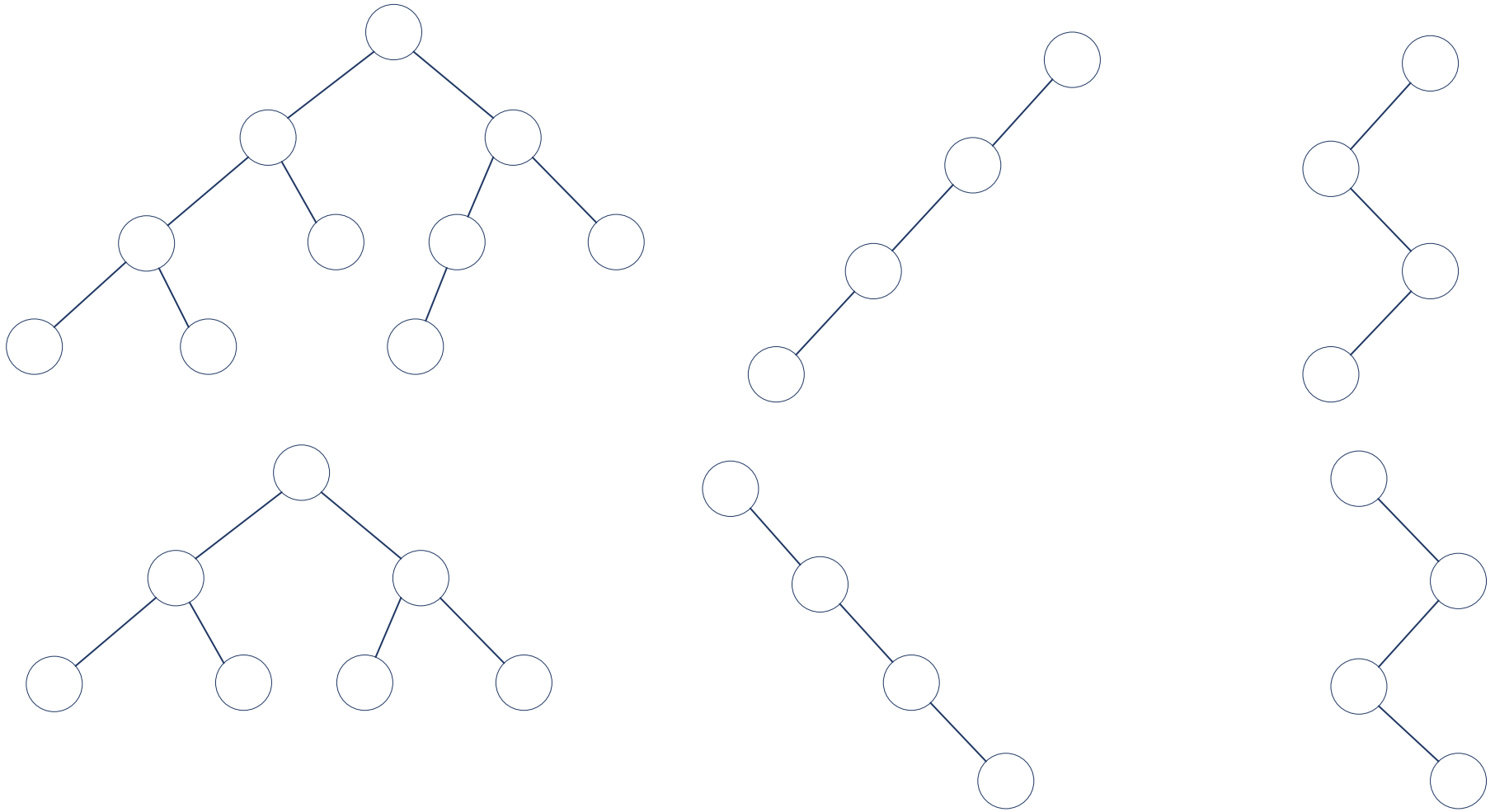
- Sơ đồ tổ chức của 1 công ty;
- Mục lục của một cuốn sách;
- Gia phả của một họ tộc;
- Cấu trúc của thư mục trên ổ đĩa;
- Biểu thức toán học cũng có thể biểu diễn bằng cây.



Cây biểu thức: (A/B + C)(D-E)*

2 CÂY NHỊ PHÂN

Khái niệm: Cây nhị phân (Binary tree) là một dạng quan trọng của cấu trúc cây. Nó có đặc điểm là mọi nút trên cây chỉ có tối đa hai nhánh con.



2 CÂY NHỊ PHÂN

Một số tính chất của cây nhị phân:

- Số lượng tối đa các nút ở mức i của cây nhị phân là 2^i, tối thiểu là 1 ($i \geq 1$).
- Số lượng tối đa các nút trên một cây nhị phân có chiều cao h là: $2^{h+1}-1$ và số lượng nút tối thiểu là: h+1 ($h \geq 1$).
- Cây nhị phân hoàn chỉnh có n nút thì chiều cao của nó là:

$$h = \log_2(n+1) - 1.$$

2 CÂY NHỊ PHÂN

Biểu diễn cây nhị phân bằng danh sách liên kết

- Mỗi nút của cây nhị phân có tối đa 2 nút con, do vậy sử dụng DSLK để biểu diễn cây nhị phân sẽ hữu hiệu nhất. Mỗi nút của cây nhị phân khi đó sẽ có 3 thành phần:
  - Thành phần item chứa thông tin về nút.
  - Con trỏ left trỏ đến nút con bên trái.
  - Con trỏ right trỏ đến nút con bên phải.
- Nếu nút có ít hơn 2 nút con thì các thành phần left hoặc right sẽ trỏ tới NULL.
- Để tăng tính di chuyển trong cây ta có thể thêm con trỏ parent trỏ đến nút cha.

2 CÂY NHỊ PHÂN

Biểu diễn cây nhị phân bằng danh sách liên kết

Cấu trúc dữ liệu của nút và khai báo cây:

```
struct NodeTree {//Cấu trúc của nút
        int key;
        NodeTree *left;
        NodeTree *right;
};
typedef struct NodeTree *TREE; //Khai báo cây
```

Duyệt cây nhị phân: Phép duyệt cây nhị phân được chia thành 3 loại:

- Duyệt thứ tự trước: duyệt nút gốc trước tiên (**NLR** hoặc **NRL**).
- Duyệt thứ tự giữa: duyệt hai bên sau đó tới nút gốc (**LNR** hoặc **RNL**).
- Duyệt thứ tự sau: duyệt nút gốc sau cùng (**LRN** hoặc **RLN**).

Độ phức tạp của giải thuật là $O(\log_2 h)$. Trong đó h là chiều cao của cây.

2 CÂY NHỊ PHÂN

Biểu diễn cây nhị phân bằng danh sách liên kết

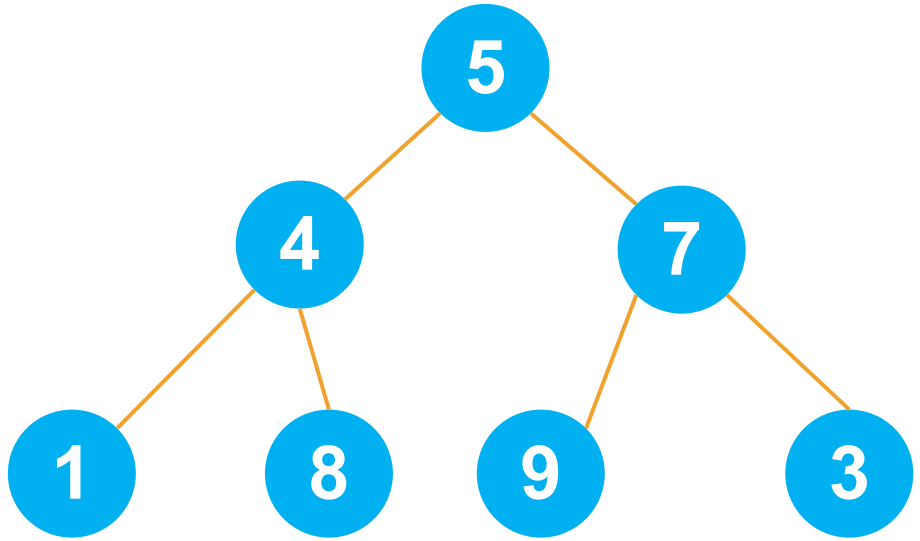
```
//Duyệt trước NLR
void NodeLeftRight(TREE test){
    if(test != NULL){
        cout<<test->key<<"  ";
        NodeLeftRight(test->left);
        NodeLeftRight(test->right);
    }
}
```

```
//Duyệt giữa LNR
void LeftNodeRight(TREE test){
    if(test != NULL){
        LeftNodeRight(test->left);
        cout<<test->key<<"  ";
        LeftNodeRight(test->right);
    }
}
```

```
//Duyệt sau LRN
void LeftRightNode(TREE test){
    if(test != NULL)    {
        LeftRightNode(test->left);
        LeftRightNode(test->right);
        cout<<test->key<<"  ";
    }
}
```

2 CÂY NHỊ PHÂN

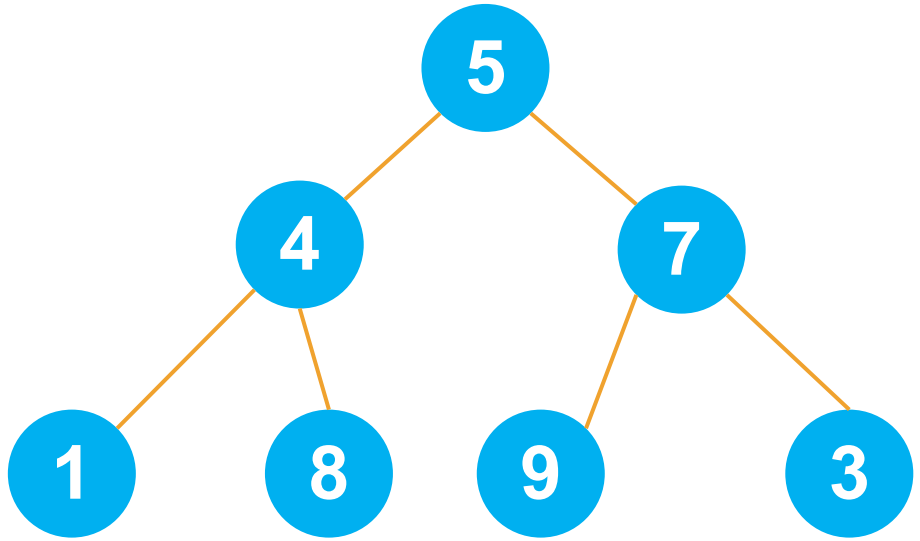
Biểu diễn cây nhị phân bằng mảng



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	**4**	**7**	**1**	**8**	**9**	**3**

2 CÂY NHỊ PHÂN

Biểu diễn cây nhị phân bằng mảng các nút cha



A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
5	**4**	**7**	**1**	**8**	**9**	**3**
0	5	5	4	4	7	7

2 CÂY NHỊ PHÂN

Thêm con trỏ parent vào các nút:

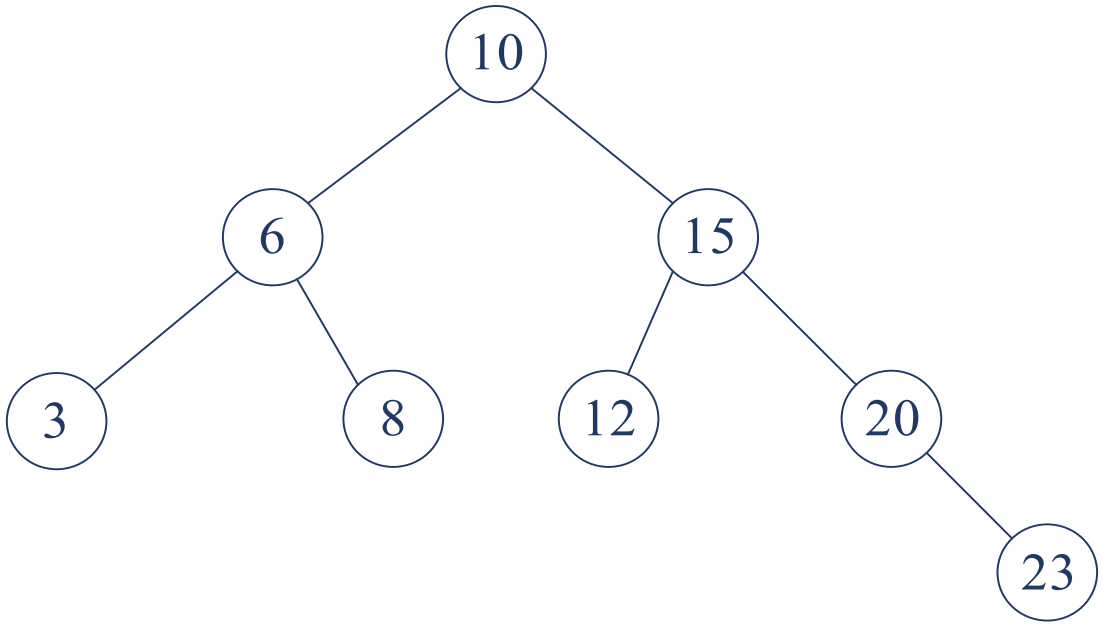
```
struct NodeTree {
        int key;
        NodeTree *left;
        NodeTree *right;
        NodeTree *parent;
};
typedef struct NodeTree *TREE;
TREE test;
```

3 CÂY NHỊ PHÂN TÌM KIẾM

Tổng quan về cậy nhị phân tìm kiếm

Cây nhị phân tìm kiếm (Binary Search Tree - sau đây viết tắt là NPTK) là cây nhị phân mà khoá tại mỗi nút cây lớn hơn khoá của tất cả các nút thuộc cây con bên trái và nhỏ hơn khoá của tất cả các nút thuộc cây con bên phải.

Lưu ý: Dữ liệu lưu trữ tại mỗi nút có thể rất phức tạp như là một bản ghi chẳng hạn, trong trường hợp này khoá của nút được tính dựa trên một trường nào đó, ta gọi là trường khoá. Trường khoá phải chứa các giá trị không trùng nhau và có thể so sánh được, tức là nó phải lấy giá trị từ một tập hợp có thứ tự.



3 CÂY NHỊ PHÂN TÌM KIẾM

Khai báo và khởi tạo cây nhị phân tìm kiếm:

```
struct Node {//Cấu trúc của nút
        int key;
        Node *left;
        Node *right;
};
typedef struct Node *TREE; //Khai báo cây
void Create (TREE &test) {//Khởi tạo cây rỗng
        test = NULL;
}
```

Các thao tác chính trên cây NPTK:

- *Tạo 1 nút có trường Key bằng x.*
- *Thêm 1 nút vào cây nhị phân tìm kiếm.*
- *Xoá 1 nút có Key bằng x trên cây.*
- *Tìm 1 nút có khoá bằng x trên cây.*

3 CÂY NHỊ PHÂN TÌM KIẾM

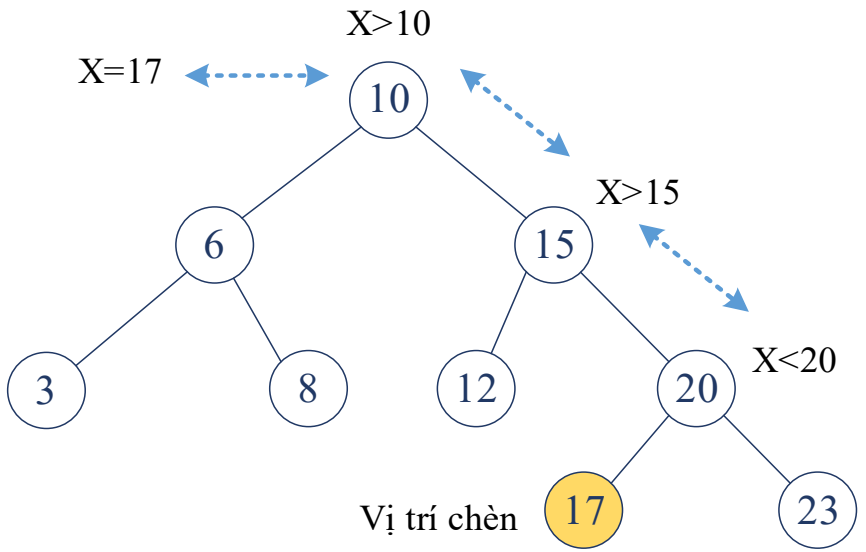
Tạo một nút có khóa bằng x:

```
Node* CreateNode (int x) {
        Node *data = new Node[1];
        if (data == NULL) return NULL; //Bộ nhớ đầy
        data->key = x;
        data->left = data->right = NULL;
       return data;
}
```

3 CÂY NHỊ PHÂN TÌM KIẾM

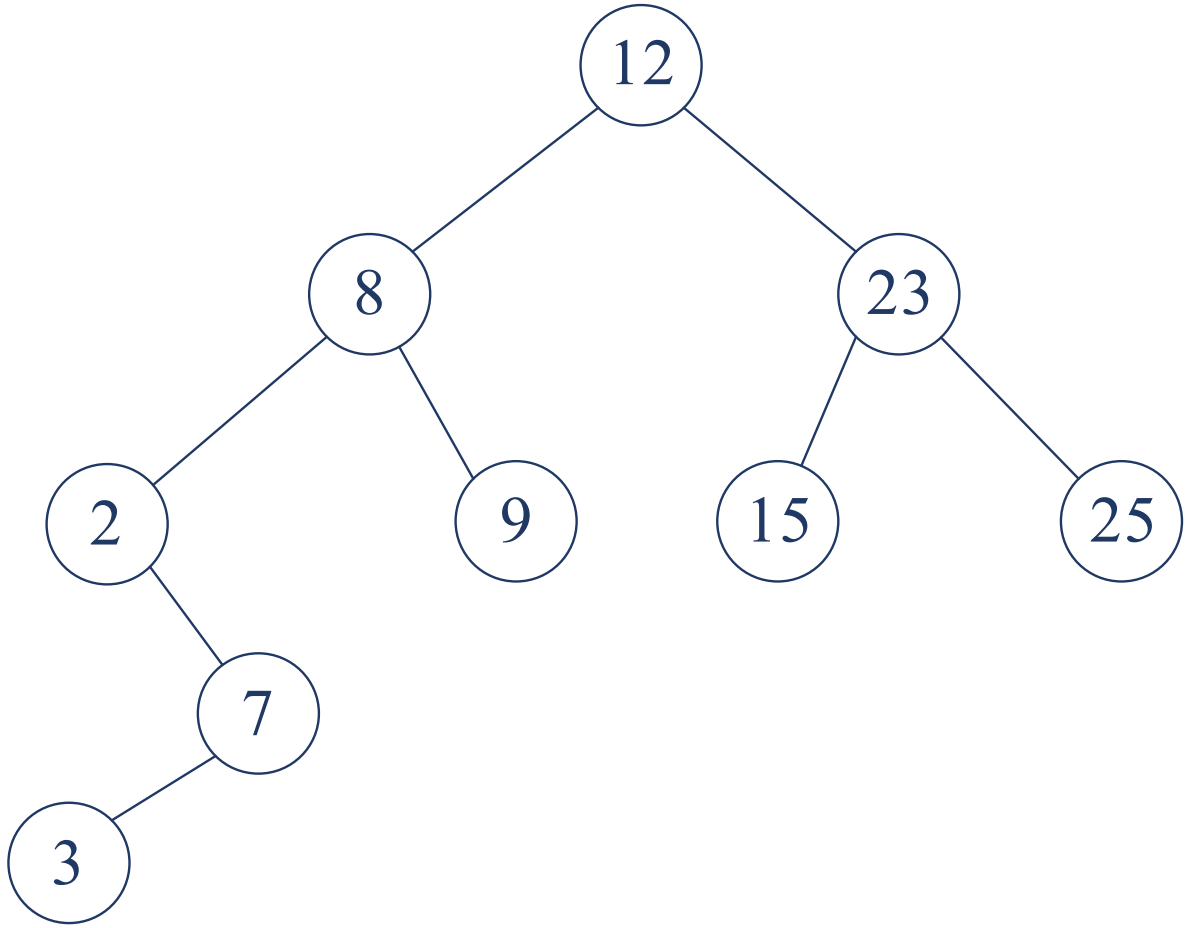
Thêm một nút có khóa x:

```
bool AddNode (TREE &test, int x) {
        if (test != NULL){ //Chưa có chỗ để thêm nút
              if(test->key == x) return false; //key không được giống nhau
              if(test->key > x)
            return AddNode(test->left, x); //Thêm vào nhánh trái
              else return AddNode(test->right, x); //Thêm vào nhánh bên phải
        }
        test = CreateNode(x); //Đã có chỗ để thêm nút
        return true;
}
```



3 CÂY NHỊ PHÂN TÌM KIẾM

Ví dụ: Tạo cây nhị phân tìm kiếm từ dãy: 12, 8, 23, 15, 2, 7, 9, 3, 25.



3 CÂY NHỊ PHÂN TÌM KIẾM

Tìm nút có khóa x không dùng đệ quy:

```
Node* Search (TREE t, int x) {
        Node *p = t;
        while (p != NULL) {
                if (x == p->key) return p; //Tìm được nút p có khóa bằng x
                if (x < p->key) p = p->left; //Tìm tiếp ở nhánh bên trái
                if (x > p->key) p = p->right; //Tìm tiếp ở nhánh bên phải
        }
        return NULL; //Không tìm được nút nào có khóa x
}
```

3 CÂY NHỊ PHÂN TÌM KIẾM

Tìm nút có khóa x dùng đệ quy:

```
Node* recursionSearch (TREE t, int x) {
        if (t != NULL) {
                if (x == t->key) return t;
                if (x < t->key) return recursionSearch(t->left, x);
                if (x > t->key) return recursionSearch(t->right, x);
        }
        return NULL;
}
```

3 CÂY NHỊ PHÂN TÌM KIẾM

Hủy nút có khóa x trên cây:

Có 3 trường hợp khi hủy 1 nút X trên cây:

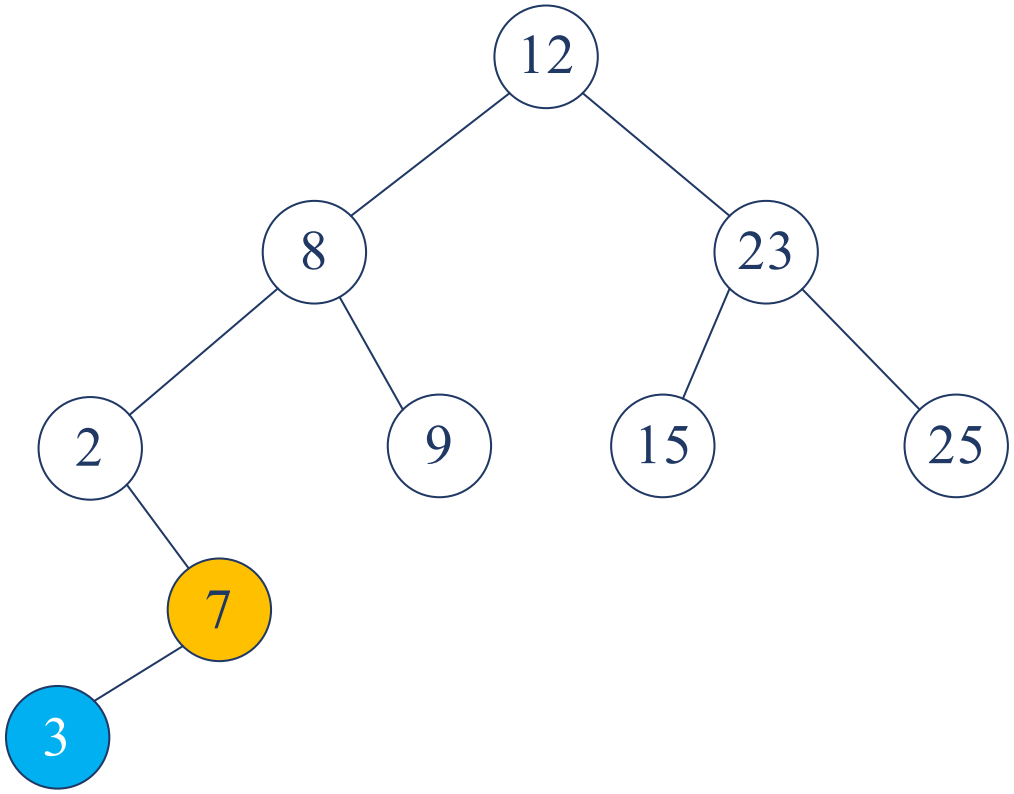
- X là nút lá: Hủy bình thường mà không ảnh hưởng đến các nút khác.
- X chỉ có 1 cây con: Trước khi xoá, ta móc nối cha của X với con duy nhất của X.
- X có đầy đủ 2 cây con: Ta dùng cách xoá gián tiếp.

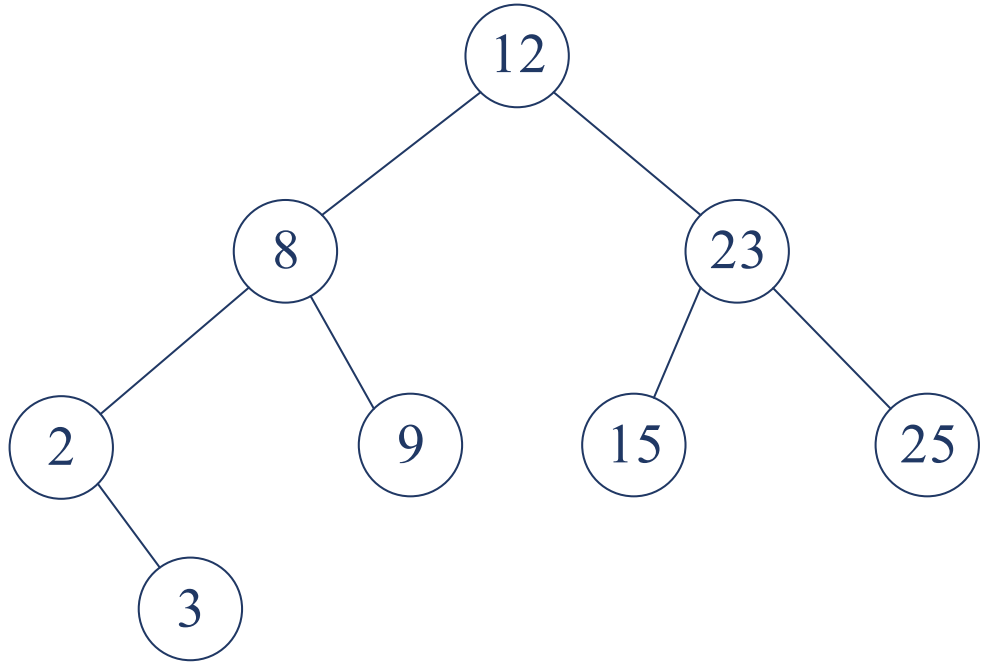
Cách xóa gián tiếp như sau:

- Thay vì hủy X ta tìm phần tử thế mạng Y. Nút Y có tối đa 1 cây con.
- Thông tin lưu tại nút Y sẽ được chuyển lên lưu tại X.
- Ta tiến hành xoá hủy nút Y (xoá Y giống 2 trường hợp đầu).
- Có 2 cách tìm nút thế mạng Y cho X:
  - *Cách 1: Nút Y là nút có khoá nhỏ nhất (trái nhất) bên cây con phải X.*
  - *Cách 2: Y là nút có khoá lớn nhất (phải nhất) bên cây con trái của X.*

3 CÂY NHỊ PHÂN TÌM KIẾM

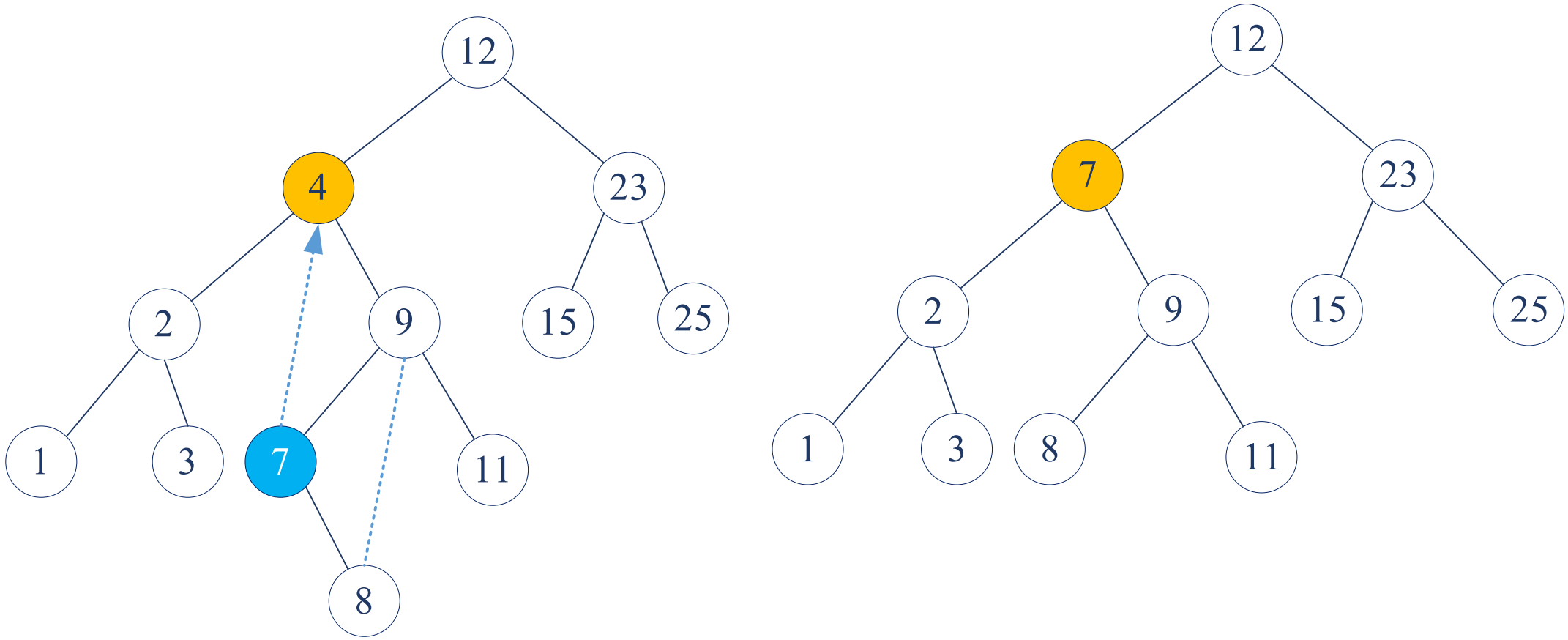
Ví dụ: Hủy nút có 1 cây con (X = 7, phần tử thế mạng Y = 3).





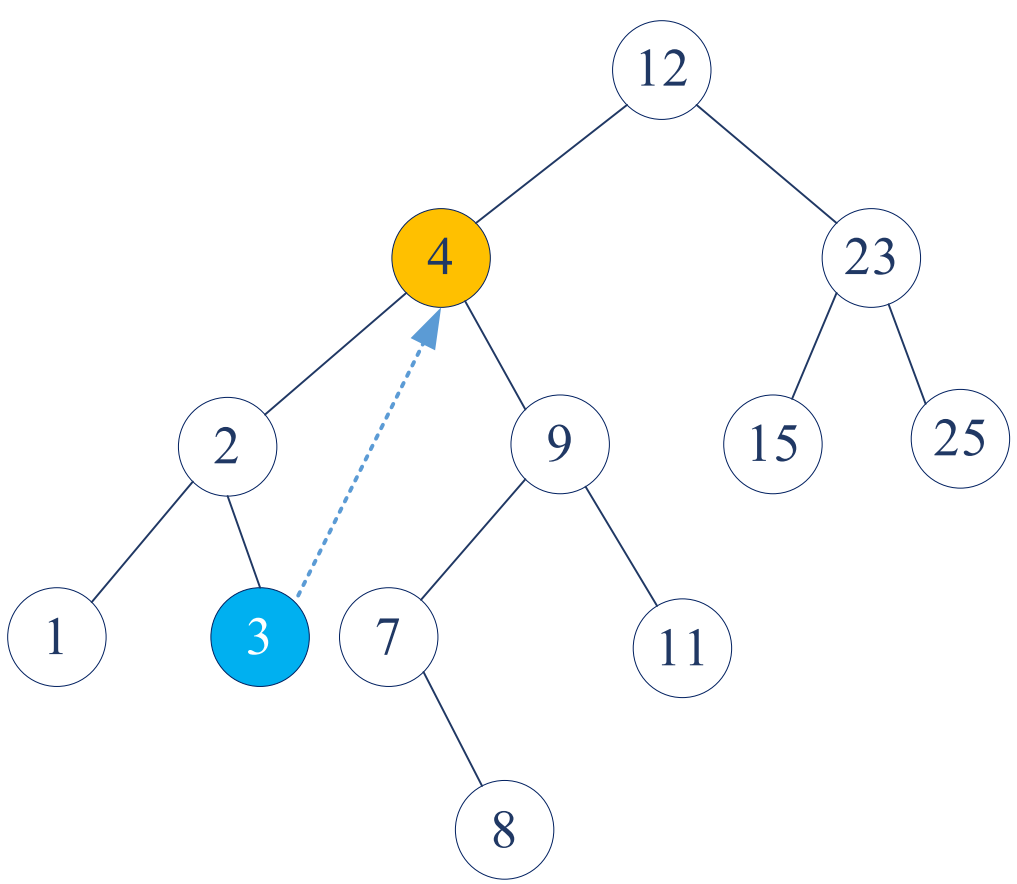
3 CÂY NHỊ PHÂN TÌM KIẾM

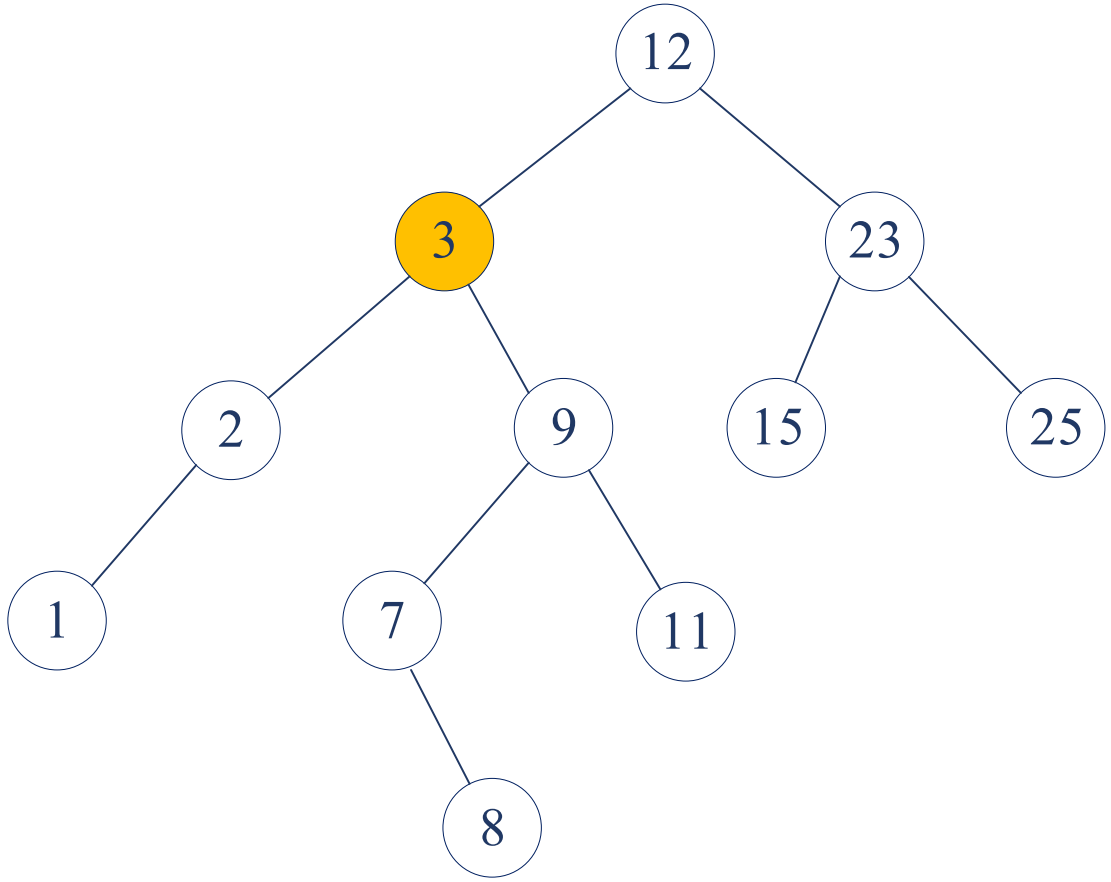
Ví dụ: Hủy nút có 2 cây con (X = 4, Y = 7).



3 CÂY NHỊ PHÂN TÌM KIẾM

Ví dụ: Hủy nút có 2 cây con (X = 4, Y = 3).





3 CÂY NHỊ PHÂN TÌM KIẾM

Hàm tìm phần tử thay thế:

```
void ReplaceNode (TREE &node, TREE &NodeLeft) {
        if (NodeLeft->left != NULL)
                ReplaceNode (node, NodeLeft->left);
        else {
//Thay key cho 'node' bằng key của node nhỏ nhất bên phải mới tìm được
                node->key = NodeLeft->key;
                node = NodeLeft;
                NodeLeft = NodeLeft->right;
        }
}
```

3 CÂY NHỊ PHÂN TÌM KIẾM

Cài đặt hàm xóa nút có key là x:

```
void RemoveNode (TREE &test, int x) {
        if (test == NULL) cout<<"Node not found!\n";
        else {
                if (test->key < x)
                        RemoveNode(test->right, x);
                else {
                        if (test->key > x) RemoveNode(test->left,x);
                        else {
                                Node *p; p = test;
                                if (test->left == NULL) test = test->right;
                                else {
                                        if (test->right == NULL) test = test->left;
                                        else ReplaceNode (p, test->right);
                                }
                                delete p;
                        }
                }
        }
}
```

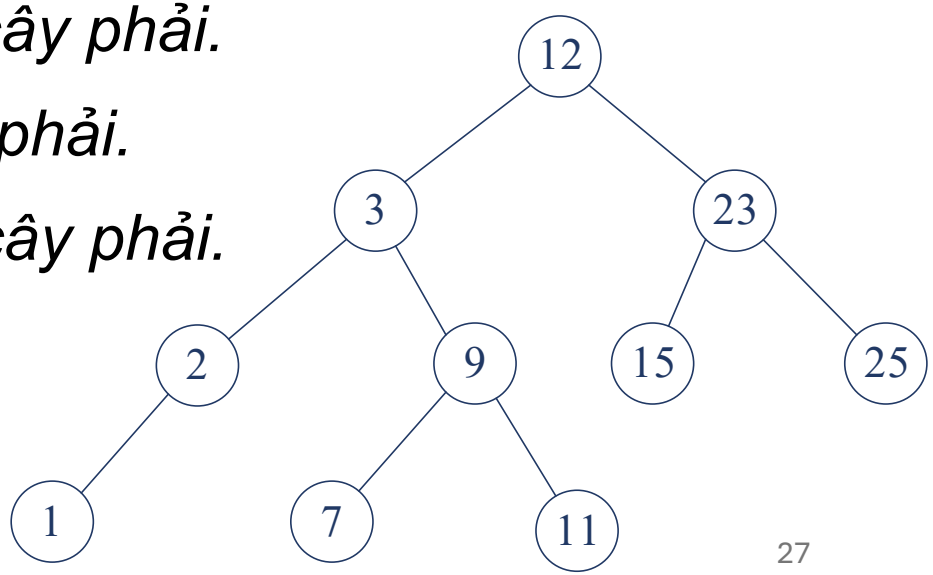
4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Định nghĩa: Cây nhị phân tìm kiếm cân bằng (Balanced binary search tree) 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. Cây nhị phân tìm kiếm cân bằng còn có tên là cây AVL, do G.M. **A**delsen **V**elskii và E.M. **L**endis đưa ra năm 1962.

Độ lệch giữa cây trái và cây phải của một nút được gọi là chỉ số cân bằng.

Các trường hợp hợp lệ của chỉ số cân bằng (CSCB):

- *CSCB = 1: Độ cao cây trái nhỏ hơn độ cao cây phải.*
- *CSCB = 0: Độ cao cây trái bằng độ cao cây phải.*
- *CSCB = -1: Độ cao cây trái lớn hơn độ cao cây phải.*



4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Cấu trúc dữ liệu của cây nhị phân tìm kiếm cân bằng:

```
#define LH -1 //Cây con trái cao hơn
#define EH 0 //E-Equilibrium Cây con trái bằng cây con phải
#define RH 1 //Cây cơn phải cao hơn
struct Node {
        char BalanceFactor; //Chỉ số cân bằng
        int key;
        Node *Left;
        Node *Right;
};
typedef Node *Tree;
```

4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Các thao tác trên cây cân bằng

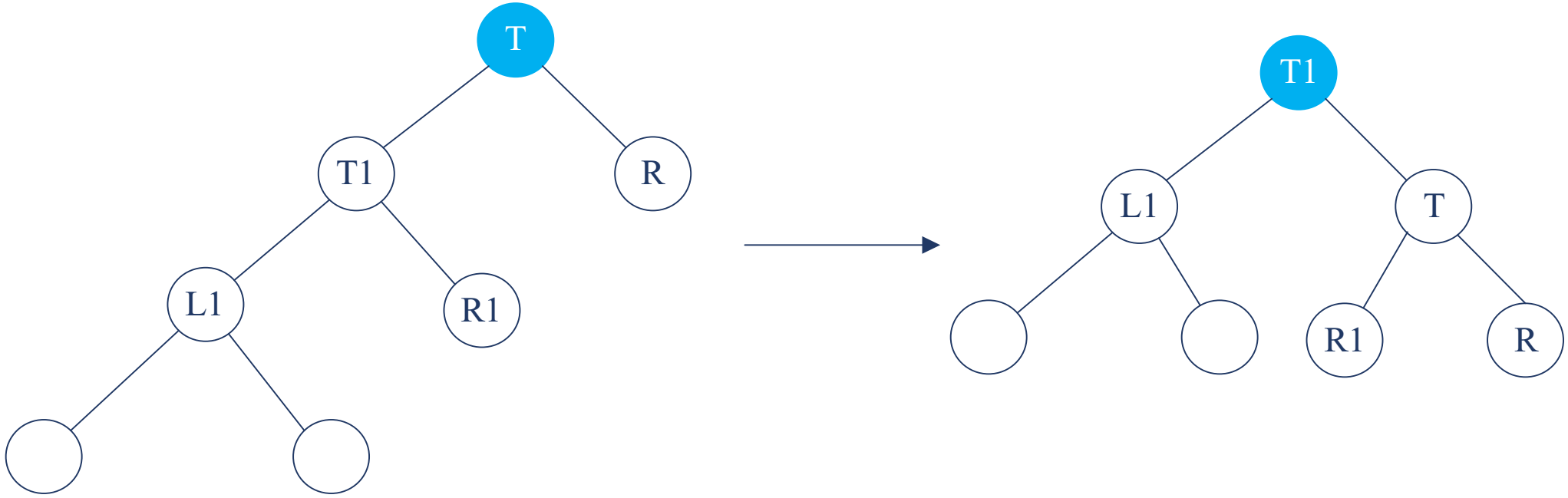
Khi thêm hay xoá 1 nút trên cây, có thể làm cho cây mất tính cân bằng, khi ấy ta phải tiến hành cân bằng lại. Cây có khả năng mất cân bằng khi thay đổi chiều cao:

- *Thêm bên trái -> lệch nhánh trái.*
- *Thêm bên phải -> lệch nhánh phải.*
- *Hủy bên phải -> lệch nhánh trái.*
- *Hủy bên trái -> lệch nhánh phải.*

Để cân bằng lại cây ta tìm cách bố trí lại cây sao cho chiều cao 2 cây con cân đối với nhau bằng cách kéo nhánh cao bù cho nhánh thấp sao cho bảo đảm cây sau đó vẫn là cây nhị phân tìm kiếm.

4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Cân bằng cho cây lệch trái, nhánh trái tại nút T (Left – Left):



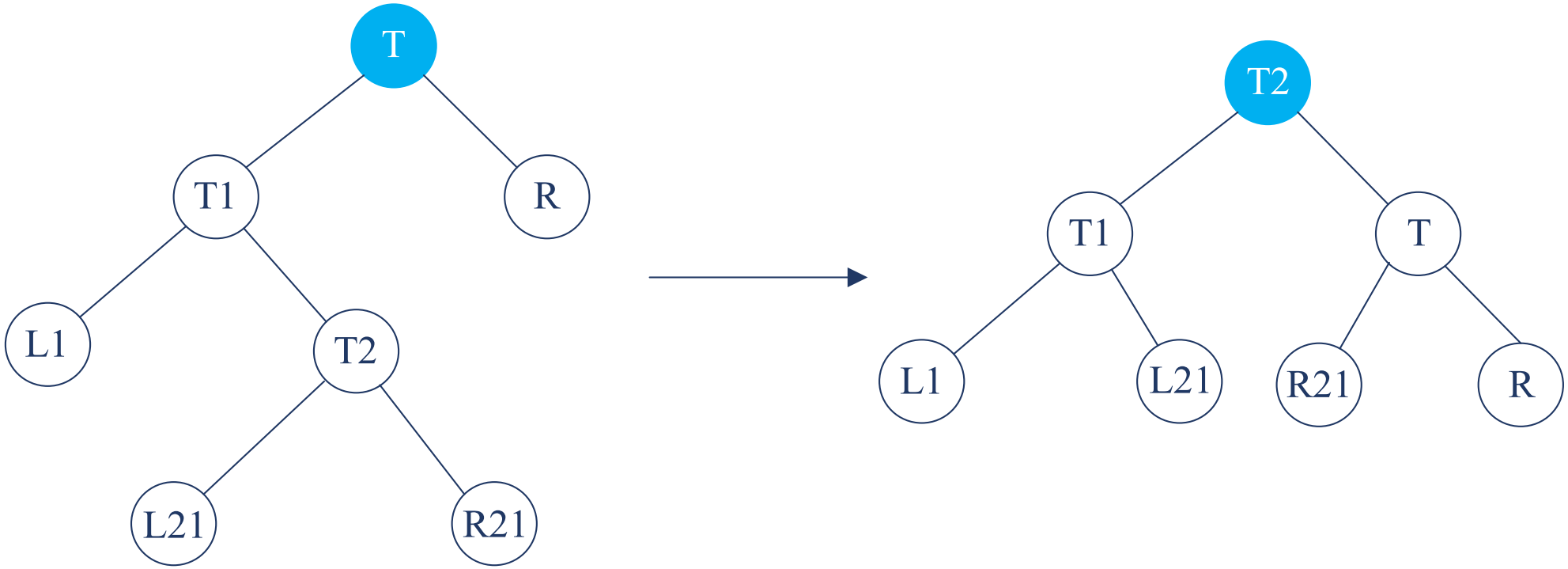
4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Hàm cân bằng cho trường hợp lệch trái Left – Left:

```
void Lech_trai_trai (Tree &T) {
        Node *T1 = T->Left;
        T->Left = T1->Right;
        T1->Right = T;
        switch(T1->BalanceFactor) {
                case LH:
                        T->BalanceFactor = EH;
                        T1->BalanceFactor = EH; break;
                case EH:
                        T->BalanceFactor = EH;
                        T1->BalanceFactor = RH; break;
        }
        T = T1;
}
```

4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Cân bằng cho cây lệch trái, nhánh phải tại nút T (Left – Right):



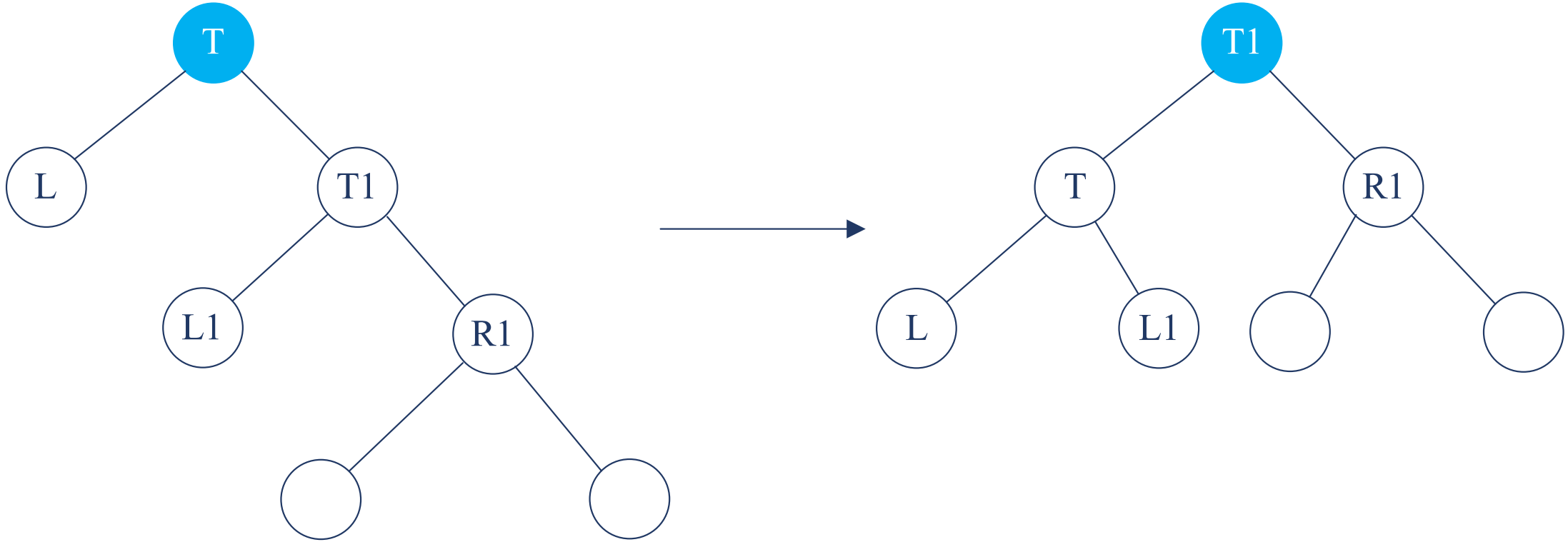
4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Hàm cân bằng cho trường hợp lệch trái Left – Right:

```
void Lech_trai_phai (Tree &T) {
        Node *T1 = T->Left;
        Node *T2 = T1->Right;
        T->Left = T2->Right;
        T2->Right = T;
        T1->Right = T2->Left;
        T2->Left = T1;
        switch(T2->BalanceFactor) {
                case LH:
                        T->BalanceFactor = EH;T1->BalanceFactor = RH; break;
                case EH:
                        T->BalanceFactor = EH;T1->BalanceFactor = EH; break;
                case RH:
                        T->BalanceFactor = LH; T1->BalanceFactor = EH; break;
        }
        T2->BalanceFactor = EH;
        T = T2;
}
```

4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Cân bằng cho cây lệch phải, nhánh phải tại nút T (Right – Right):



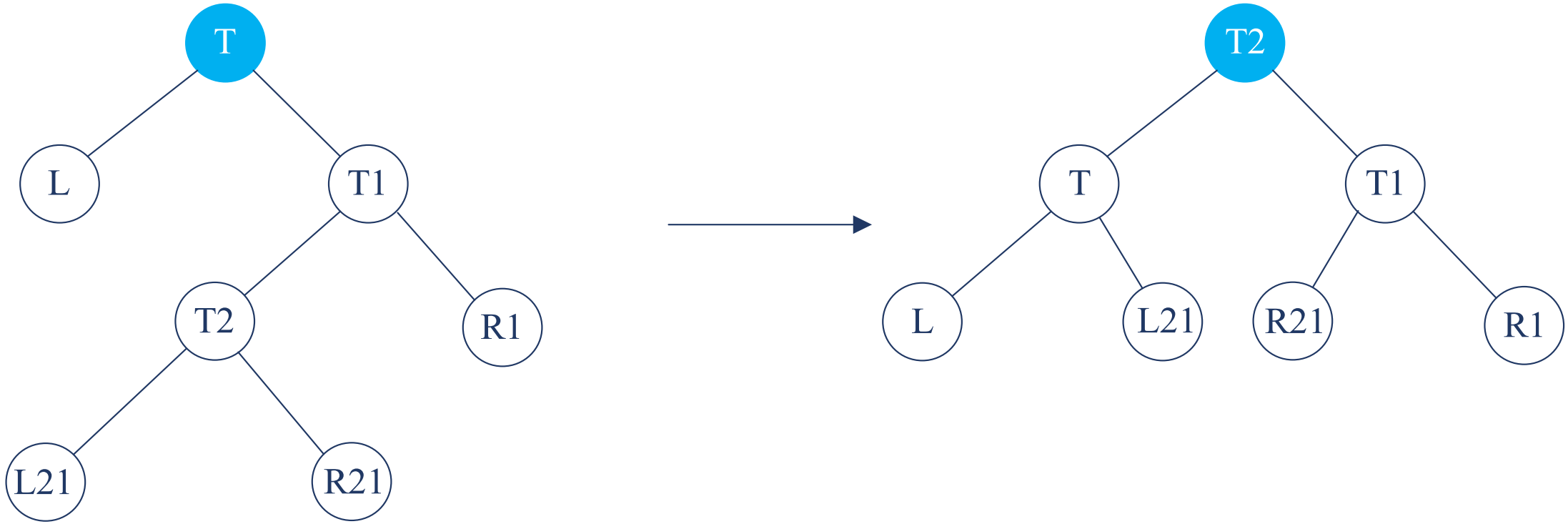
4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Hàm cân bằng cho trường hợp Right – Right:

```
void Lech_phai_phai (Tree &T) {
        Node *T1 = T->Right;
        T->Right = T1->Left;
        T1->Left = T;
        switch(T1->BalanceFactor) {
                case RH:
                        T->BalanceFactor = EH;
                        T->BalanceFactor = EH;
                        break;
                case EH:
                        T->BalanceFactor = EH;
                        T1->BalanceFactor = LH;
                        break;
        }
        T = T1;
}
```

4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Cân bằng cho cây lệch phải, nhánh trái tại nút T (Right – Left):



4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Hàm cân bằng cho trường hợp Right – Left:

```
void Lech_phai_trai (Tree &T) {
      Node *T1 = T->Right;
      Node *T2 = T1->Left;
      T->Right = T2->Left;
      T2->Left = T;
      T1->Left = T2->Right;
      T2->Right = T1;
      switch(T2->BalanceFactor) {
            case RH: T->BalanceFactor = EH; T1->BalanceFactor = LH; break;
            case EH: T->BalanceFactor = EH; T1->BalanceFactor = EH; break;
            case LH: T->BalanceFactor = RH;T1->BalanceFactor = EH; break;
      }
      T2->BalanceFactor = EH;
      T = T2;
}
```

4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Thêm nút vào cây nhị phân tìm kiếm cân bằng:

- Thêm bình thường như trường hợp cây NPTK.
- Nếu cây tăng trưởng 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.
- Việc cân bằng lại chỉ cần thực hiện 1 lần nơi mất cân bằng.

4 CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

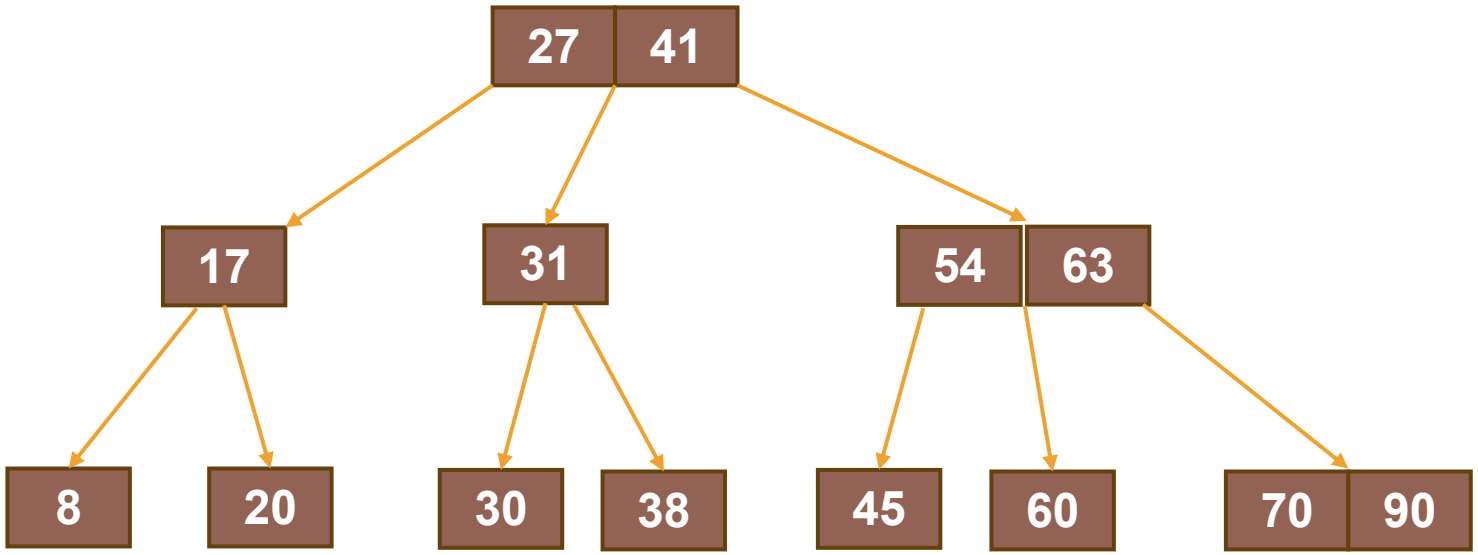
Hủy nút trên cây nhị phân tìm kiếm cân bằng:

- 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.

5 B - TREE

B-tree là một cây tìm kiếm tự cân bằng, được sắp xếp để giữ các khóa trong mỗi nút theo thứ tự.

Đặc điểm nổi bật của B-tree là mỗi nút có thể chứa nhiều khóa và nhiều con. Điều này giúp giảm chiều cao của cây một cách đáng kể, từ đó giảm số lần truy cập đĩa khi tìm kiếm.



5 B - TREE

Các Đặc Trưng Chính của B-tree

Một B-tree có các đặc trưng sau đây, được xác định bởi một tham số gọi là **bậc** của cây (t hoặc m).

Số khóa của nút: Tối đa m−1 khóa, tối thiểu [m/2]−1 khóa (trừ nút gốc)

Số con của nút: Tối đa m con, tối thiểu [m/2] con (trừ nút gốc)

Tính cân bằng: Tất cả các nút lá nằm trên cùng một mức

Nút gốc: Có ít nhất 1 khóa và 2 con (trừ khi cây rỗng)

5 B - TREE

Các Đặc Trưng Chính của B-tree

1. Nút (Node)

Mỗi nút trong B-tree có thể chứa một số lượng khóa (key) và con trỏ tới các nút con.

- Mỗi nút có tối đa $m-1$ khóa và m con.
- Các khóa trong mỗi nút được sắp xếp theo thứ tự tăng dần.

2. Cấu trúc Nút

- Một nút không phải là lá (non-leaf node) với k khóa sẽ có $k+1$ con.
- Mỗi khóa K_i trong một nút sẽ phân chia các khóa của cây con: tất cả các khóa trong cây con thứ i (con trỏ C_i) nhỏ hơn K_i, và tất cả các khóa trong cây con thứ $i+1$ (con trỏ C_{i+1}) lớn hơn K_i.

5 B - TREE

Các Đặc Trưng Chính của B-tree

3. Bậc của Cây (Order of the Tree)

Bậc của cây, thường ký hiệu là **m** (hoặc đôi khi 2t), là số con tối đa mà một nút có thể có.

- Số lượng khóa tối đa trong một nút là m−1.
- Số lượng khóa tối thiểu trong một nút là [m/2]−1 (với nút gốc, số khóa tối thiểu là 1).

4. Tính Cân Bằng (Balanced Structure)

Tất cả các nút lá đều nằm trên cùng một mức (cùng một độ sâu). Điều này đảm bảo rằng đường đi từ gốc đến bất kỳ nút lá nào đều có cùng chiều dài, giúp cho thời gian tìm kiếm luôn ổn định và hiệu quả.

Các Đặc Trưng Chính của B-tree

5. Điều Kiện Của Nút Gốc (Root Node)

Nút gốc phải có ít nhất 1 khóa (tức là 2 con) trừ khi nó là nút duy nhất của cây (trường hợp cây rỗng hoặc chỉ có 1 nút).

6. Điều Kiện của các Nút Khác (Non-root Nodes)

Mỗi nút không phải là gốc và không phải là lá phải có ít nhất $\lceil m/2 \rceil$ con. Điều này tương đương với việc mỗi nút có ít nhất $\lceil m/2 \rceil - 1$ khóa.

5 B - TREE

Cài đặt B-tree bằng DSLK

```
struct KeyNode {// Cấu trúc của một nút trong danh sách liên kết khóa
    int value;
    KeyNode* next;
};
struct ChildNode {// Cấu trúc của một nút trong danh sách liên kết con trỏ con
    BTreeNode* child;
    ChildNode* next;
};
struct BTreeNode {// Cấu trúc của một nút B-tree
    KeyNode* keysHead; // Con trỏ đầu tiên của danh sách liên kết khóa
    ChildNode* childrenHead; //Con trỏ đầu tiên của danh sách liên kết con trỏ con
    BTreeNode* parent; // Con trỏ tới nút cha
    bool is_leaf; // Biến cờ để xác định nút lá
};
```

5 B - TREE

Thuật toán tìm kiếm trên B-tree

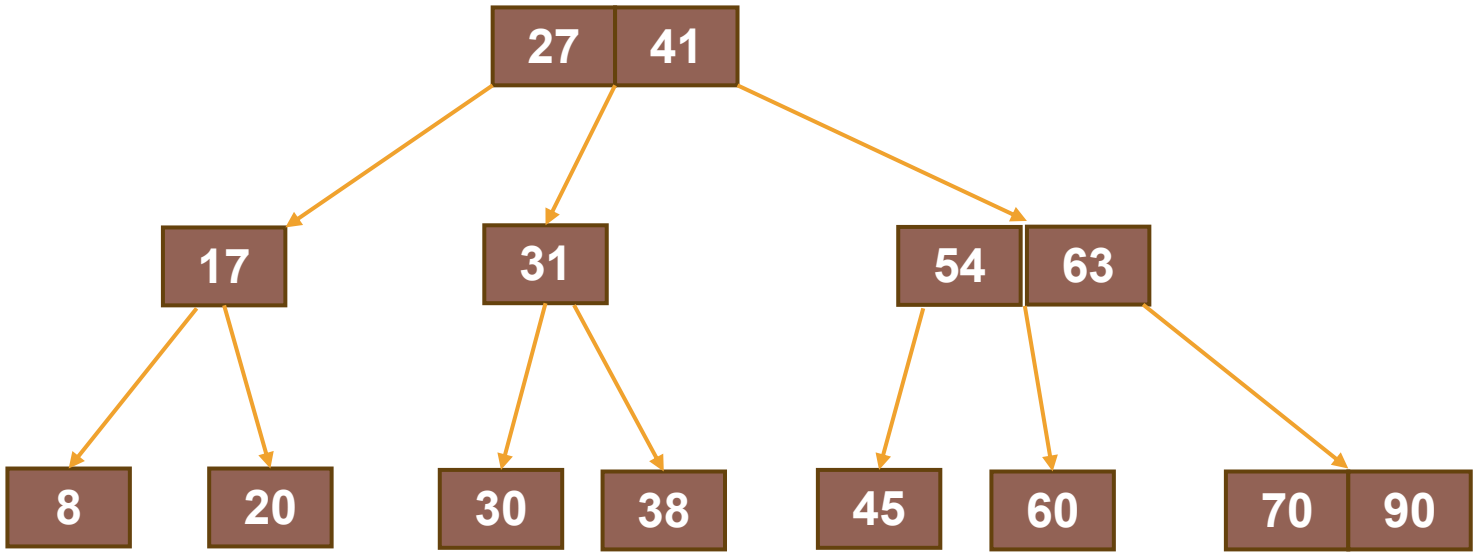
1. Bắt đầu từ nút gốc (root).
2. Trong mỗi nút, duyệt qua các khóa.
   - Nếu tìm thấy khóa cần tìm, thao tác kết thúc.
   - Nếu khóa cần tìm nhỏ hơn khóa đầu tiên của nút, di chuyển đến nút con đầu tiên.
   - Nếu khóa cần tìm nằm giữa hai khóa K_i và K_{i+1}, di chuyển đến nút con nằm giữa hai khóa đó.
   - Nếu khóa cần tìm lớn hơn tất cả các khóa trong nút, di chuyển đến nút con cuối cùng.
3. Lặp lại bước 2 cho đến khi tìm thấy khóa hoặc đến một nút lá mà không tìm thấy. Nếu không tìm thấy ở nút lá, khóa đó không tồn tại trong cây.

5 B - TREE

Thuật toán tìm kiếm trên B-tree

Ví dụ: Tìm khóa 45 trong B-tree sau:

45

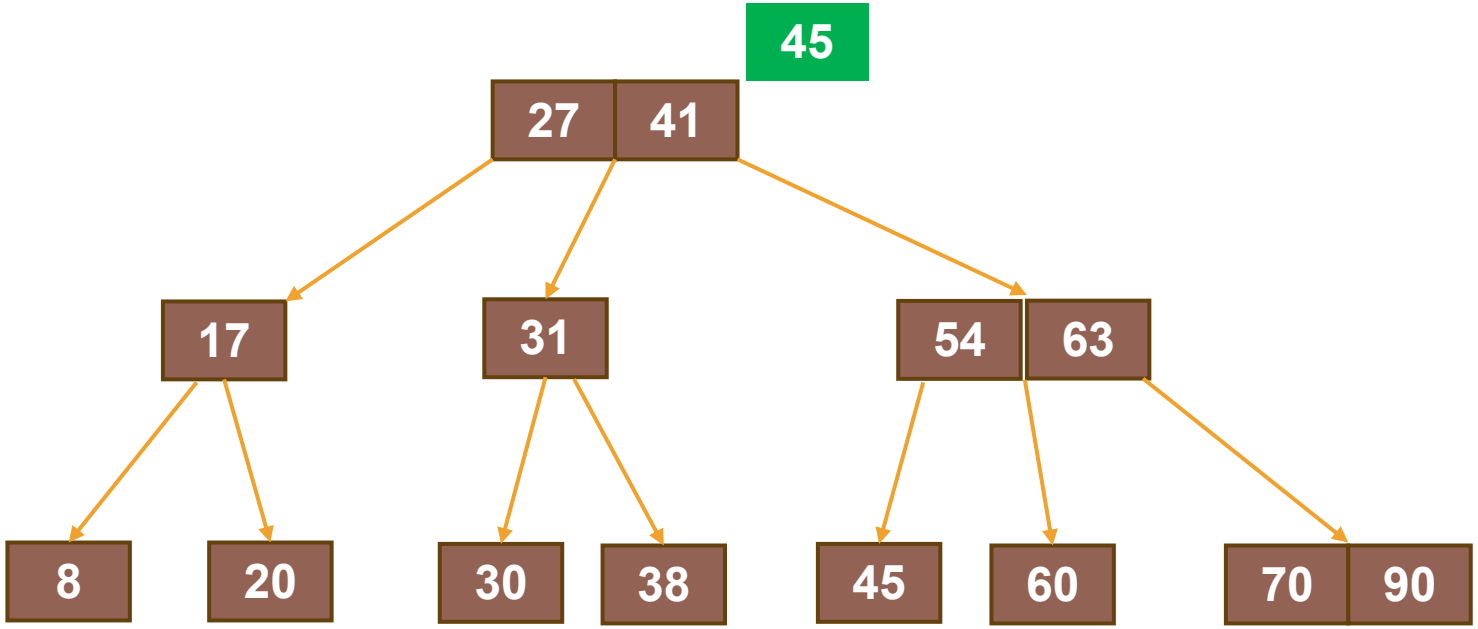


5 B - TREE

Thuật toán tìm kiếm trên B-tree

Ví dụ: Tìm khóa 45 trong B-tree sau:

Đầu tiên tìm ở nút gốc: khóa cần tìm lớn hơn tất cả các khóa nên ta sẽ tìm ở nút con cuối cùng

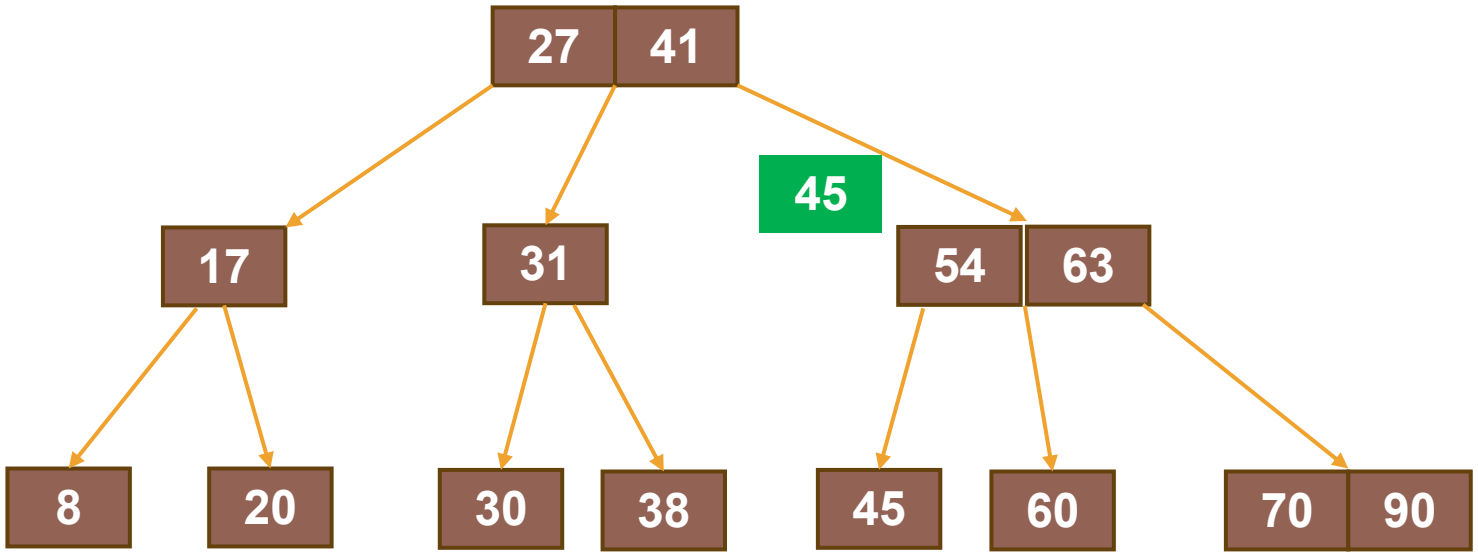


5 B - TREE

Thuật toán tìm kiếm trên B-tree

Ví dụ: Tìm khóa 45 trong B-tree sau:

Khóa cần tìm cũng nhỏ hơn tất cả các khóa trên nút đang tìm (các khóa 54, 63) nên ta sẽ tìm ở nút con đầu tiên của nút này.

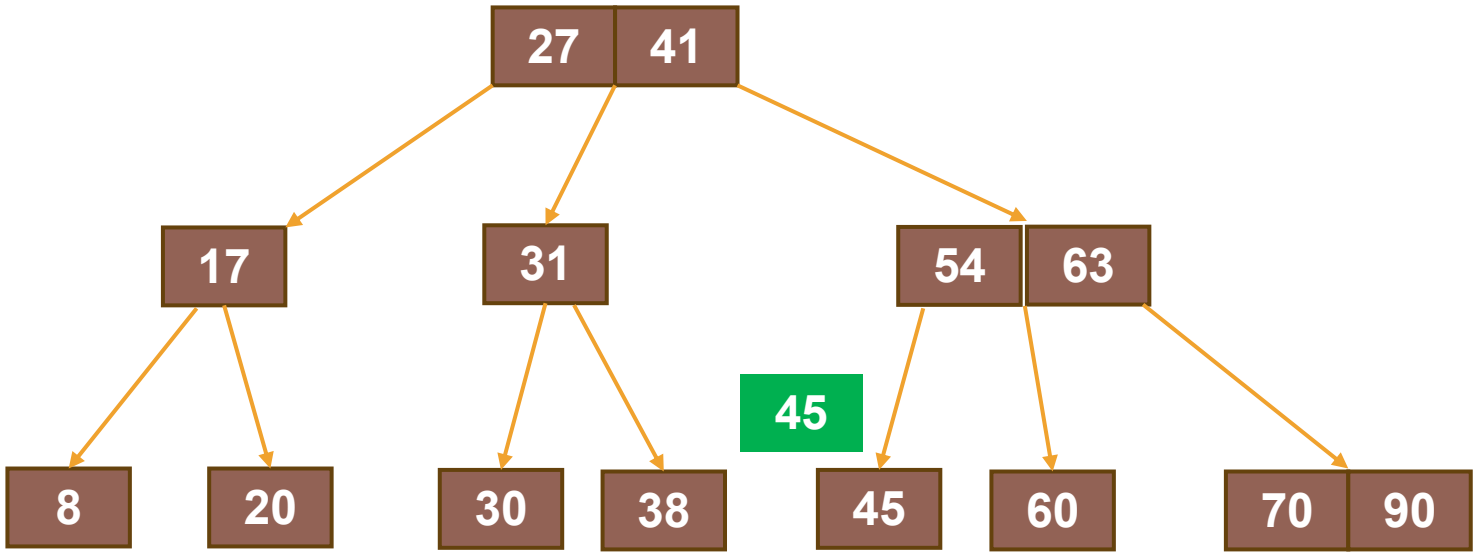


5 B - TREE

Thuật toán tìm kiếm trên B-tree

Ví dụ: Tìm khóa 45 trong B-tree sau:

Nút tìm kiếm có duy nhất khóa 45 và bằng giá trị ta cần tìm



5 B - TREE

Thuật toán chèn khóa vào B-tree

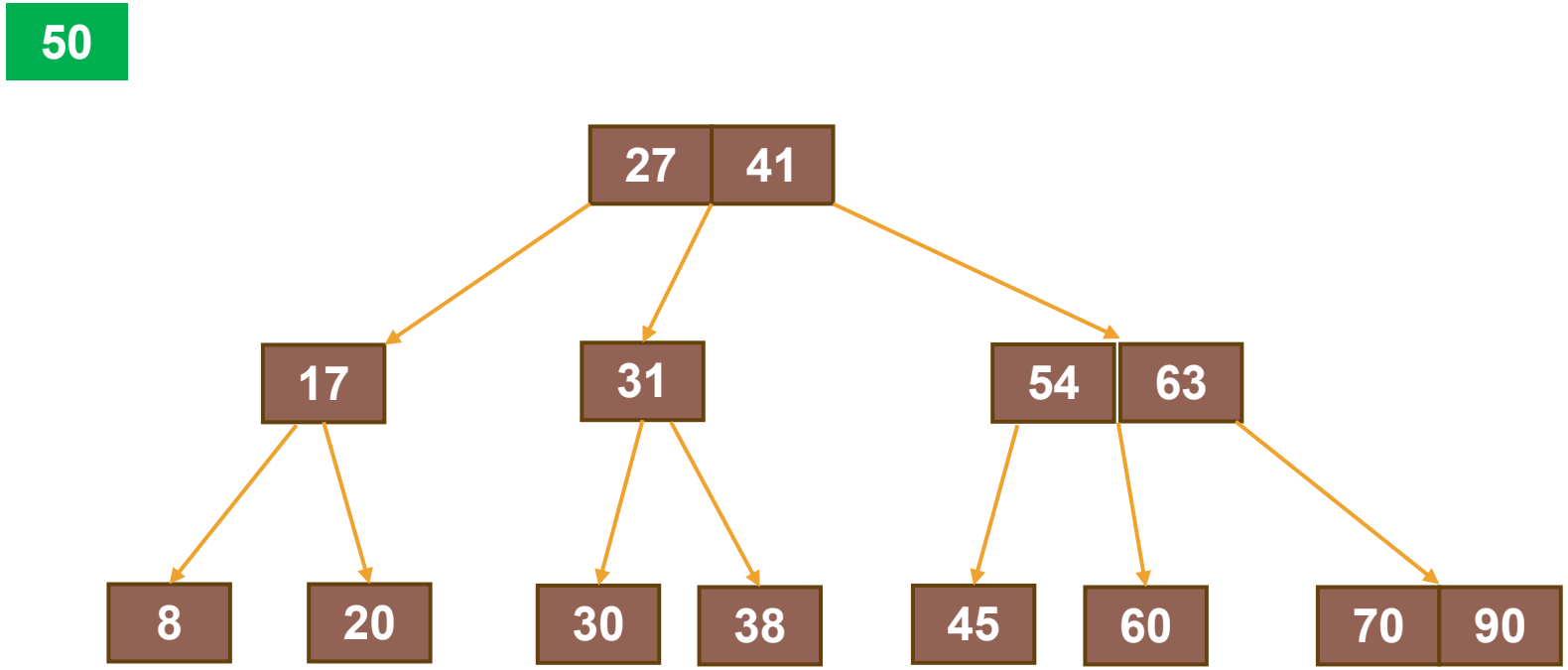
Việc chèn luôn được thực hiện tại một nút lá. Thao tác này có thể phức tạp nếu nút lá đã đầy. Các bước thực hiện như sau:

1. Tìm vị trí nút lá thích hợp để chèn khóa mới.
2. Nếu nút lá đó **chưa đầy** (số khóa < m−1), chỉ cần chèn khóa vào và sắp xếp lại các khóa trong nút.
3. Nếu nút lá đã **đầy** (số khóa = m−1), ta phải thực hiện **tách nút (split)**:
   - Chia nút đầy thành hai nút mới.
   - Đẩy khóa ở giữa lên nút cha.
   - Nếu nút cha cũng đầy, quá trình tách này sẽ tiếp tục "lan truyền" lên trên (splitting propagates upward) cho đến khi gặp một nút chưa đầy, hoặc tạo thành một nút gốc mới.

5 B - TREE

Thuật toán chèn khóa vào B-tree

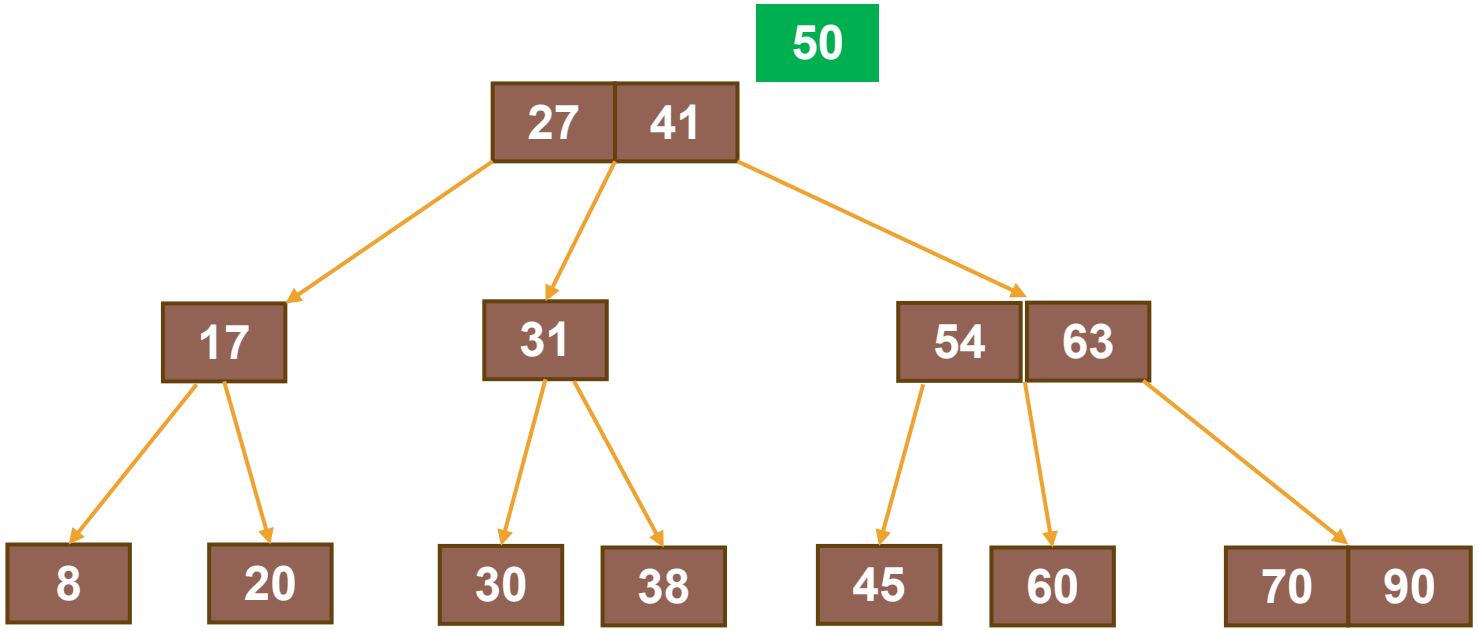
Ví dụ: Chèn nút có khóa 50 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán chèn khóa vào B-tree

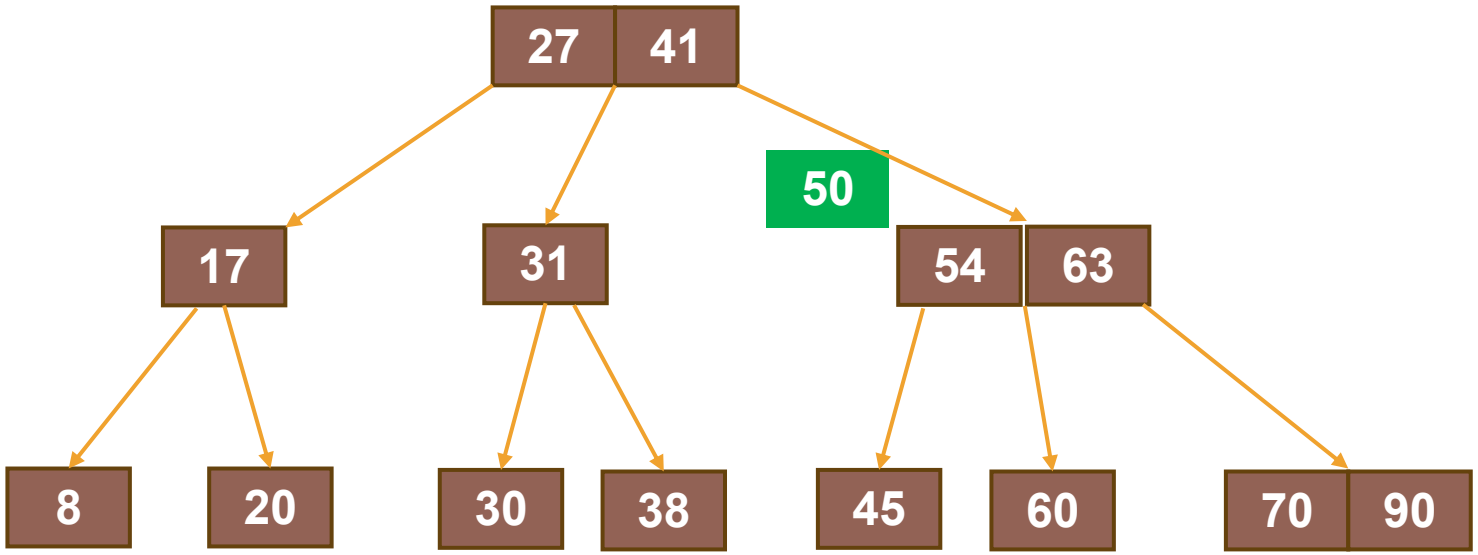
Ví dụ: Chèn nút có khóa 50 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán chèn khóa vào B-tree

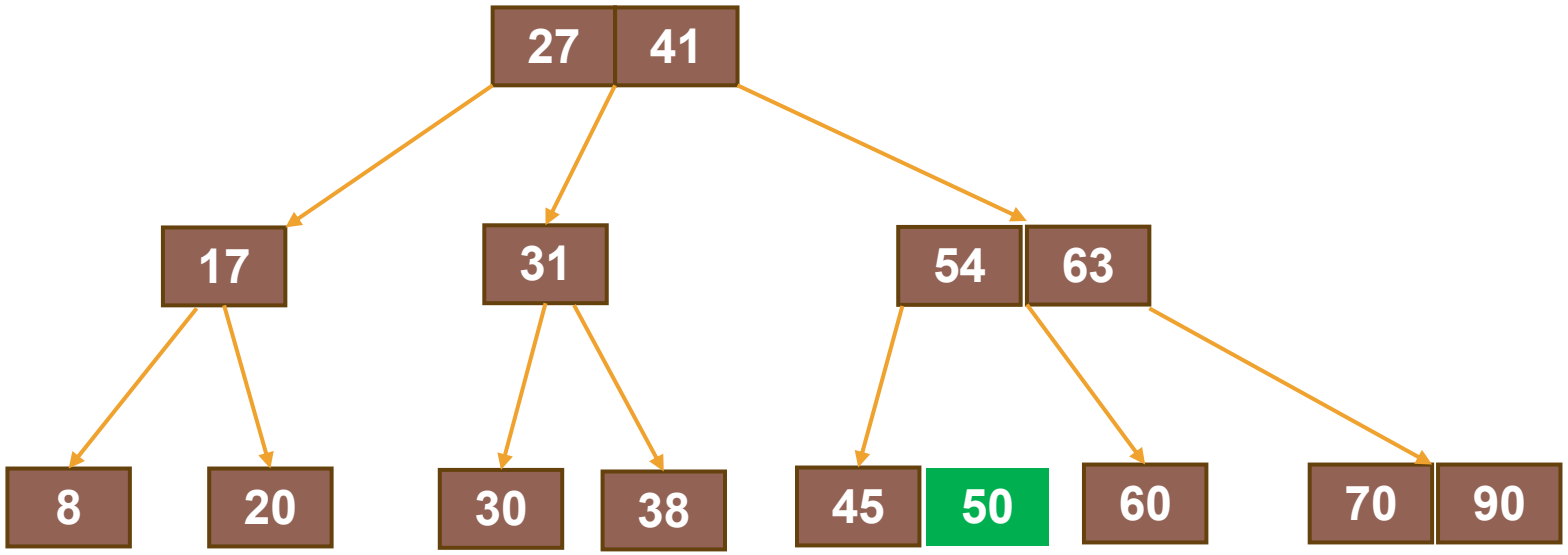
Ví dụ: Chèn nút có khóa 50 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán chèn khóa vào B-tree

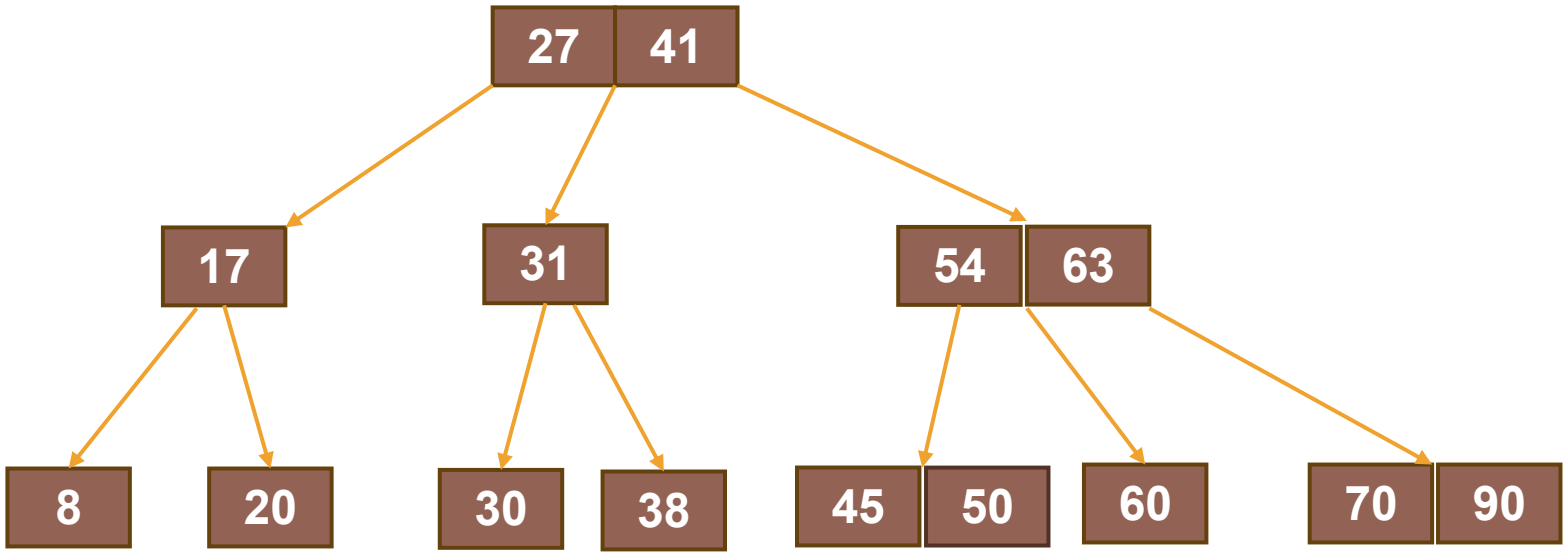
Ví dụ: Chèn nút có khóa 50 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán chèn khóa vào B-tree

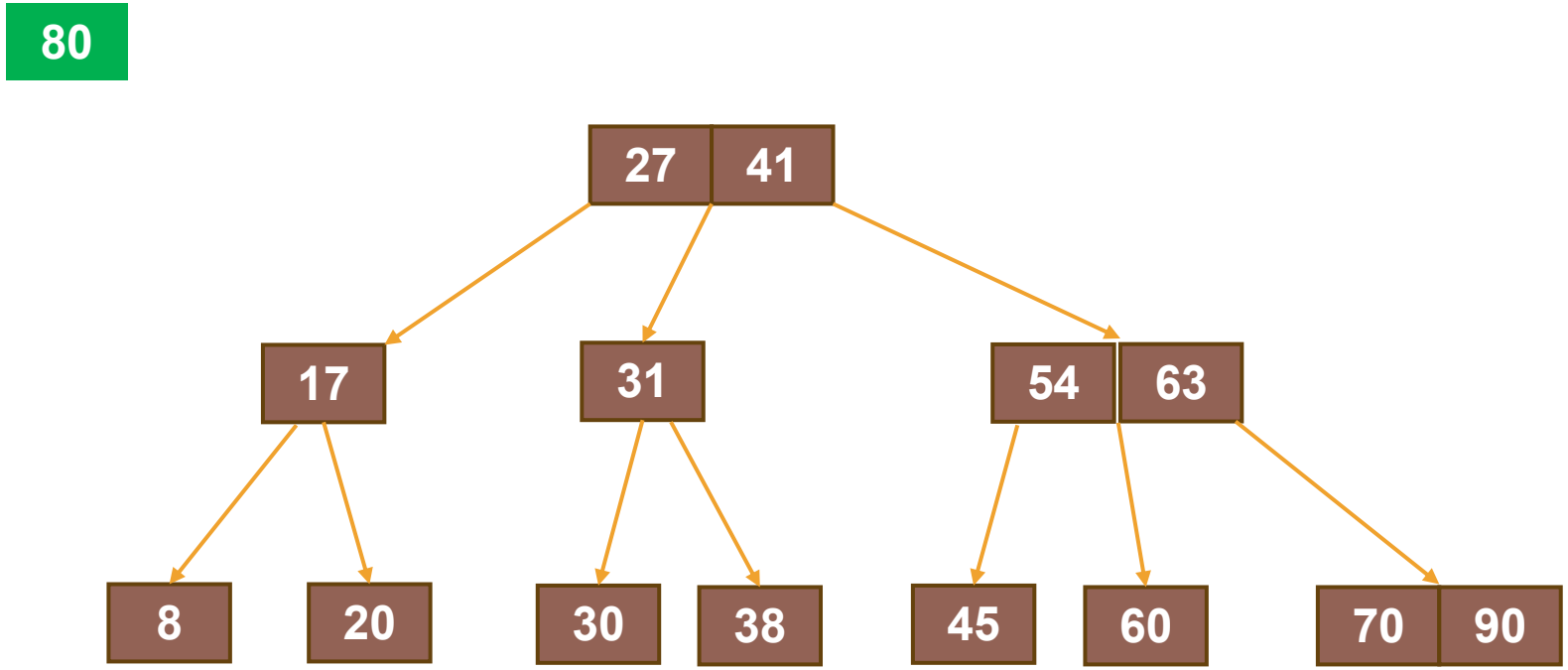
Ví dụ: Chèn nút có khóa 50 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán chèn khóa vào B-tree

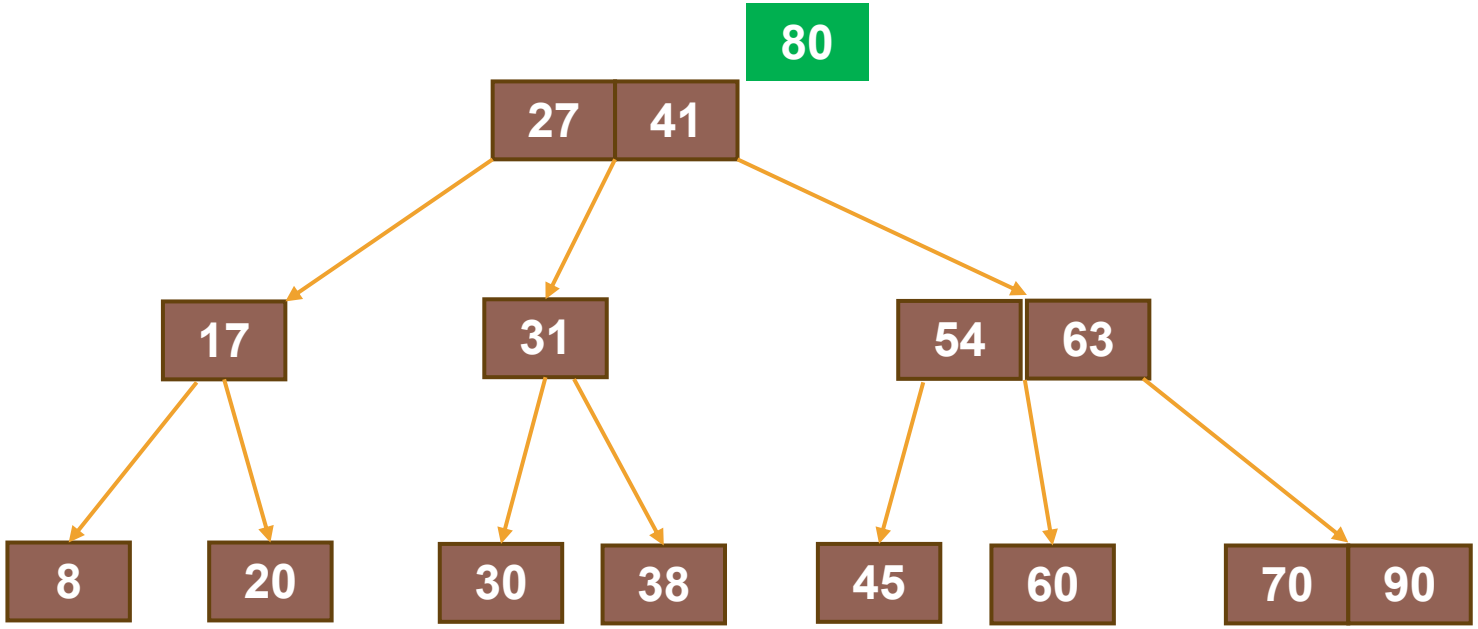
Ví dụ: Chèn nút có khóa 80 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán chèn khóa vào B-tree

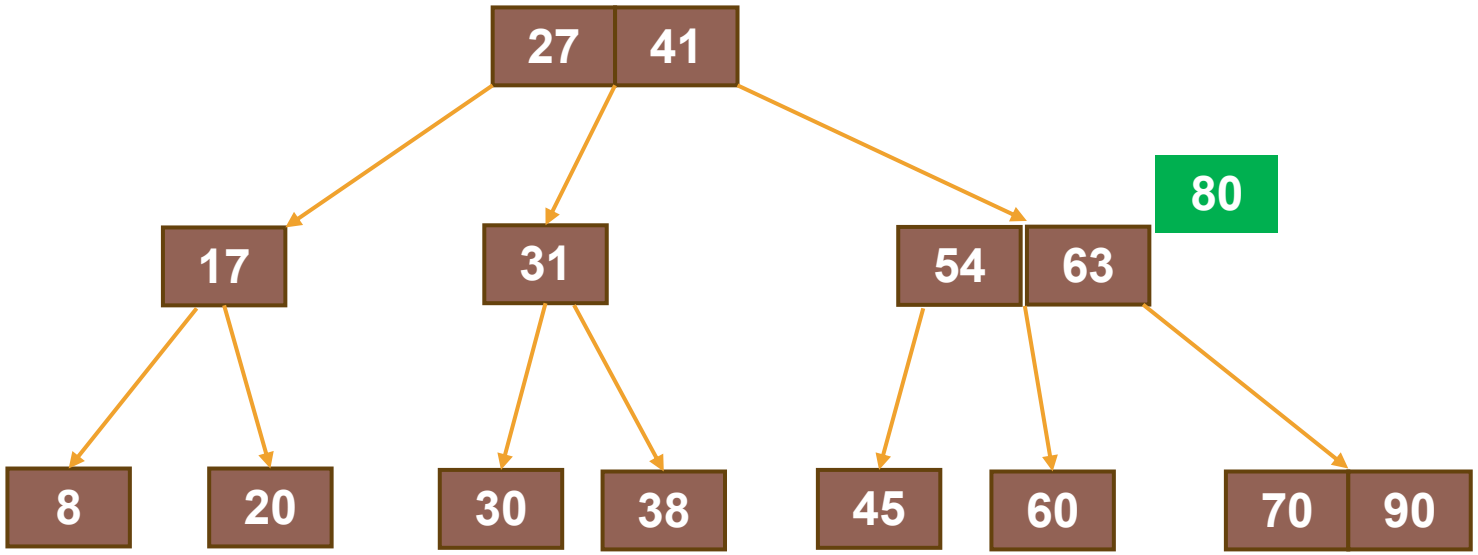
Ví dụ: Chèn nút có khóa 80 vào B-tree sau (bậc 3).



5 B - TREE

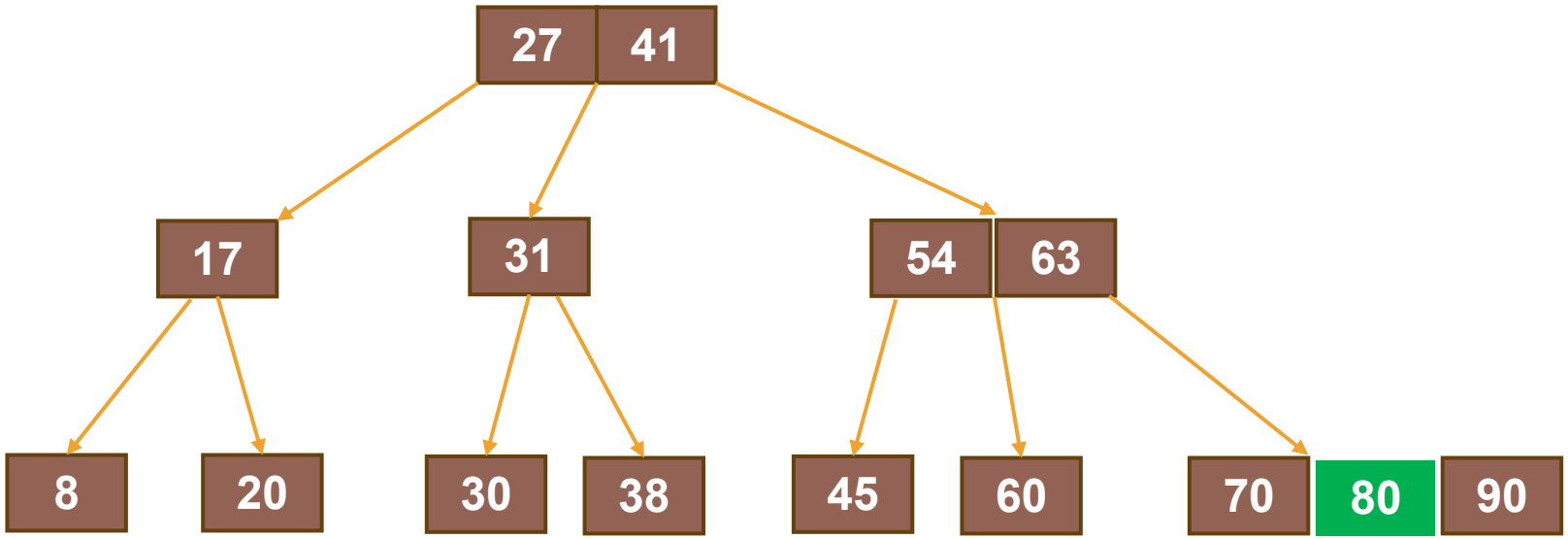
Thuật toán chèn khóa vào B-tree

Ví dụ: Chèn nút có khóa 80 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán chèn khóa vào B-tree

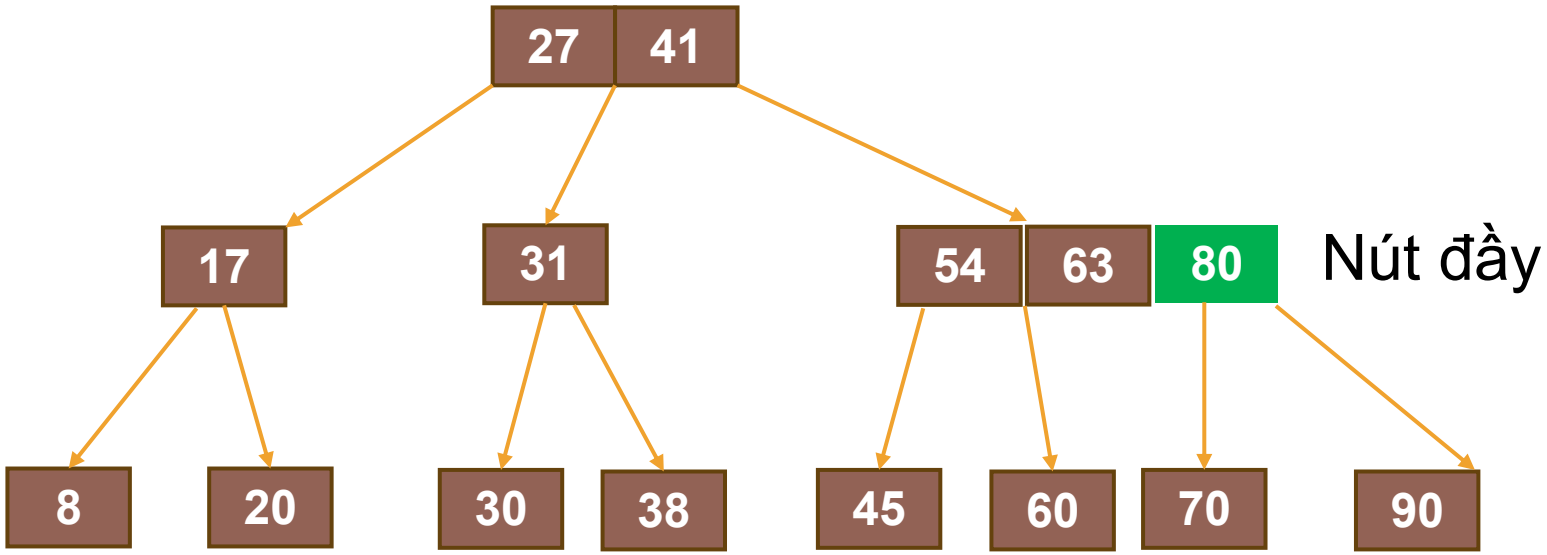


Nút đầy

5 B - TREE

Thuật toán chèn khóa vào B-tree

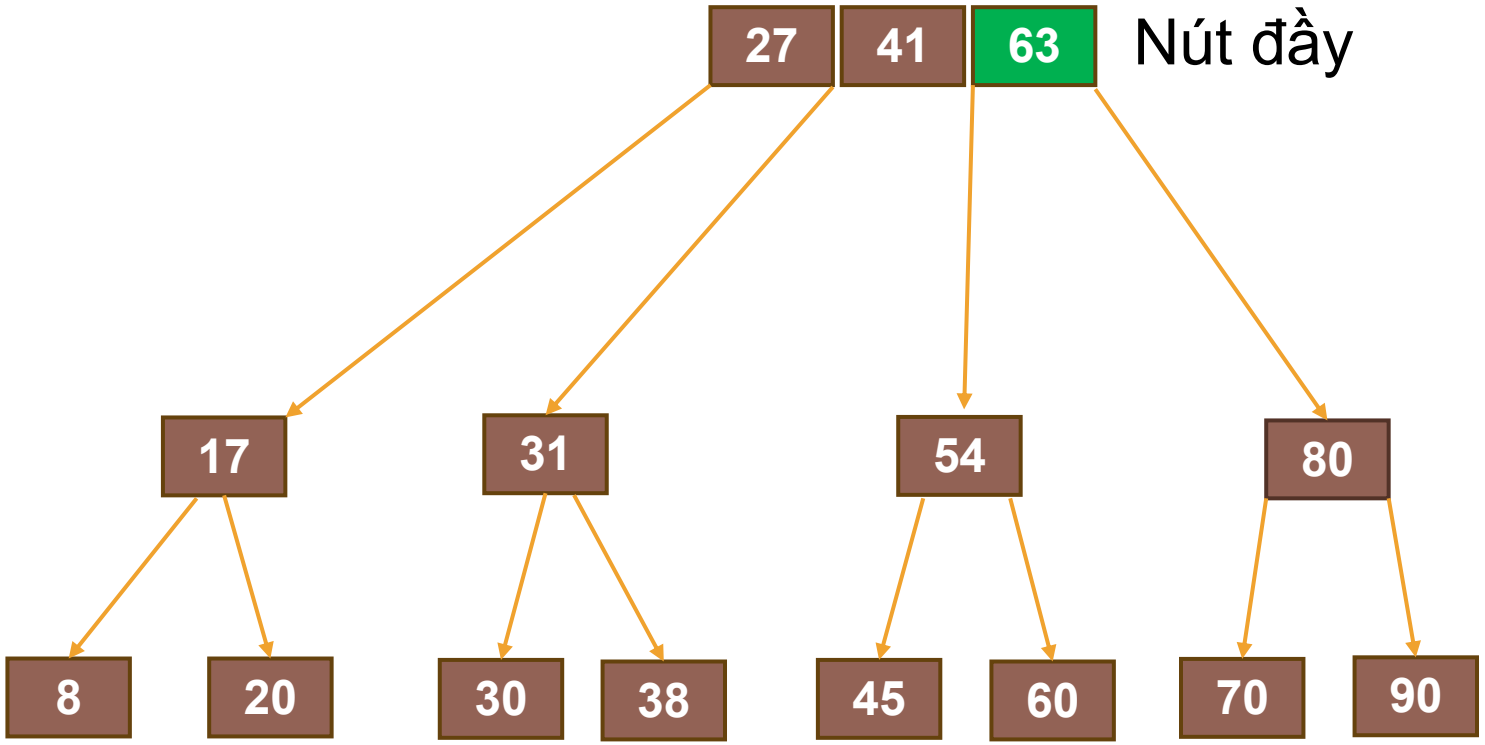
Ví dụ: Chèn nút có khóa 80 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán chèn khóa vào B-tree

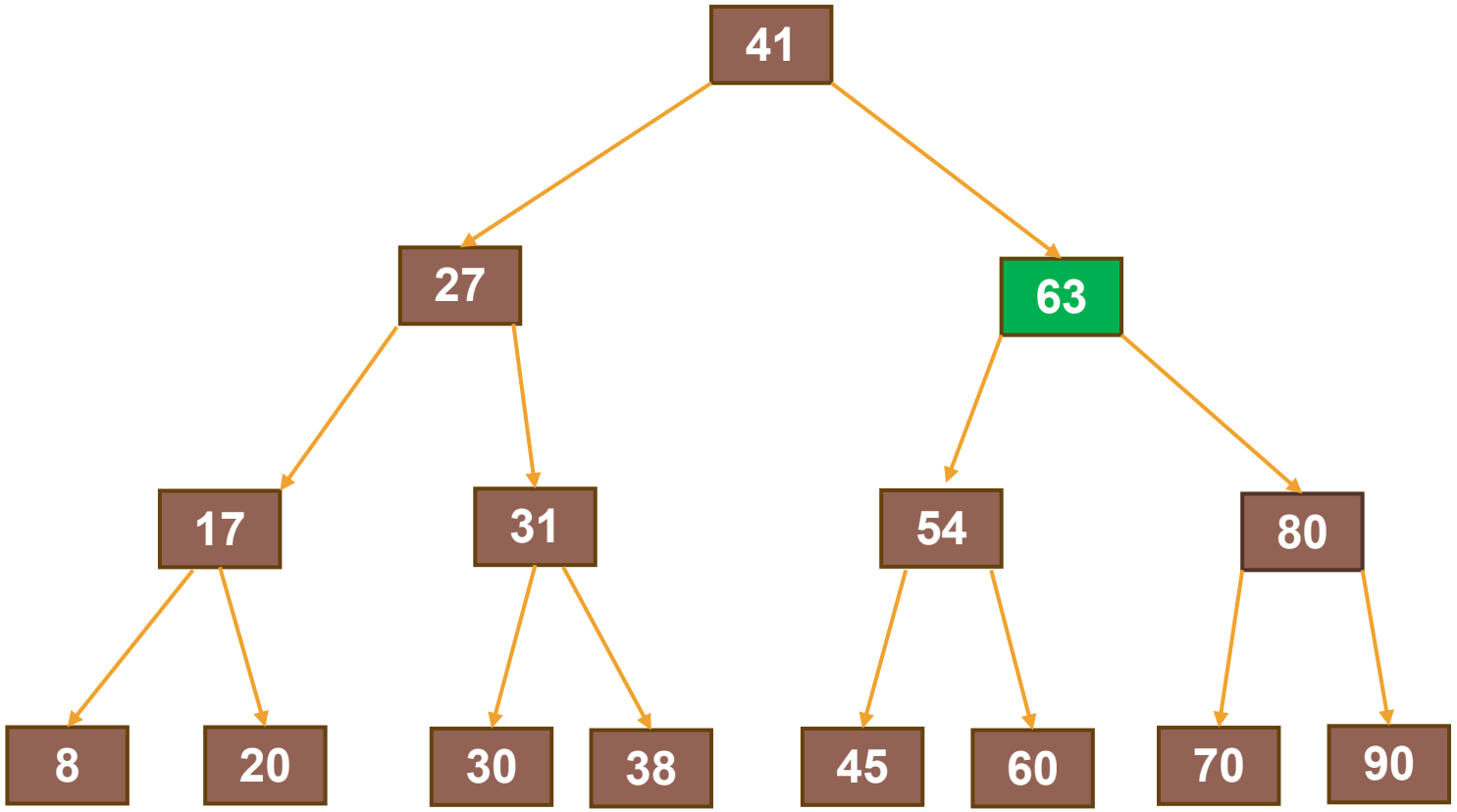
Ví dụ: Chèn nút có khóa 80 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán chèn khóa vào B-tree

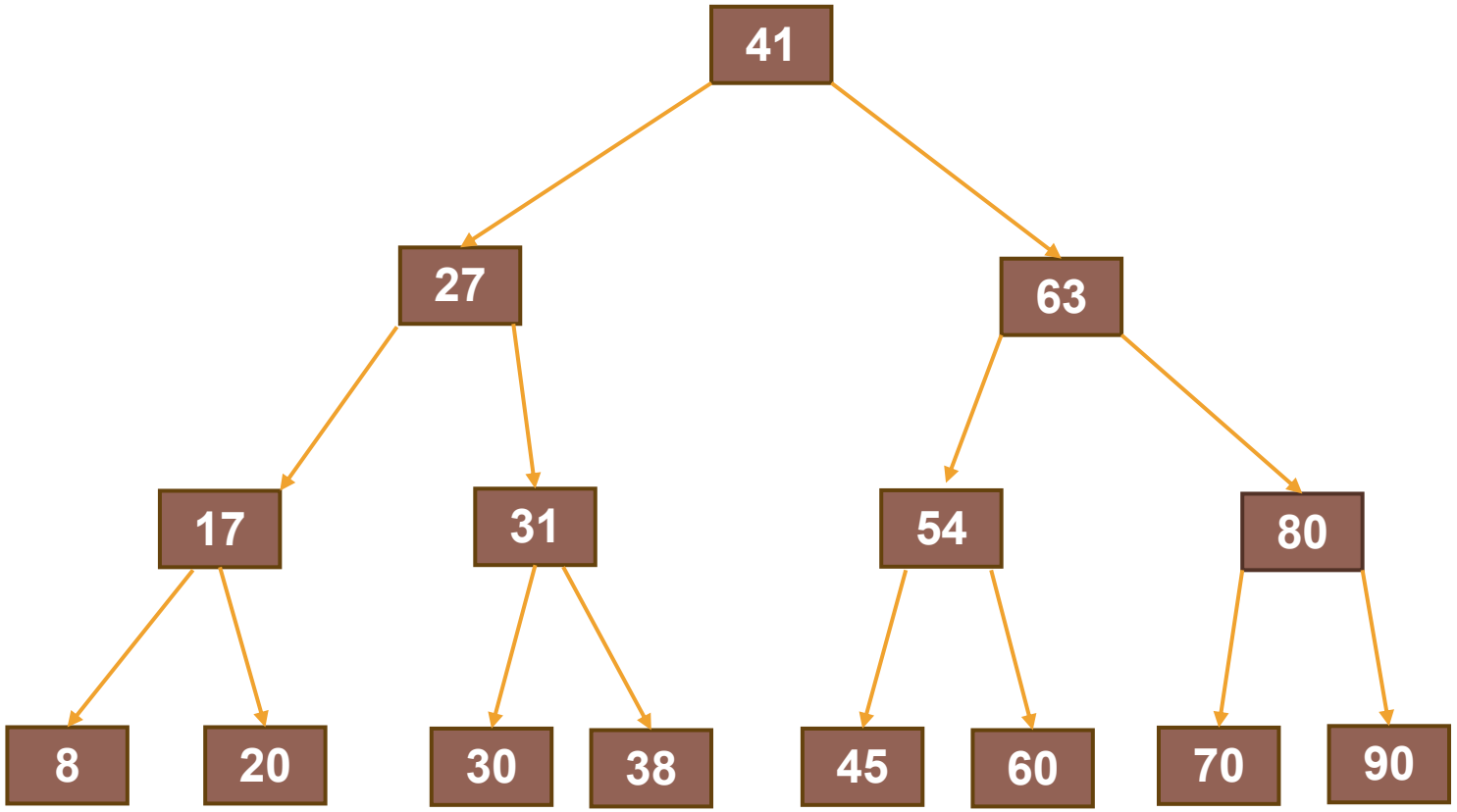
Ví dụ: Chèn nút có khóa 80 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán chèn khóa vào B-tree

Ví dụ: Chèn nút có khóa 80 vào B-tree sau (bậc 3).



5 B - TREE

Thuật toán xóa khóa khỏi B-tree

Xóa khóa trong B-tree là thao tác phức tạp nhất, vì nó có thể làm mất cân bằng cây. Có hai trường hợp chính:

- Trường hợp 1: Xóa khóa tại nút lá
- Trường hợp 2: Xóa khóa tại nút không phải lá

5 B - TREE

Thuật toán xóa khóa khỏi B-tree

Trường hợp 1: Xóa khóa tại nút lá:

Bước 1: Tìm và xóa khóa.

Bước 2: Kiểm tra số khóa còn lại trong nút.

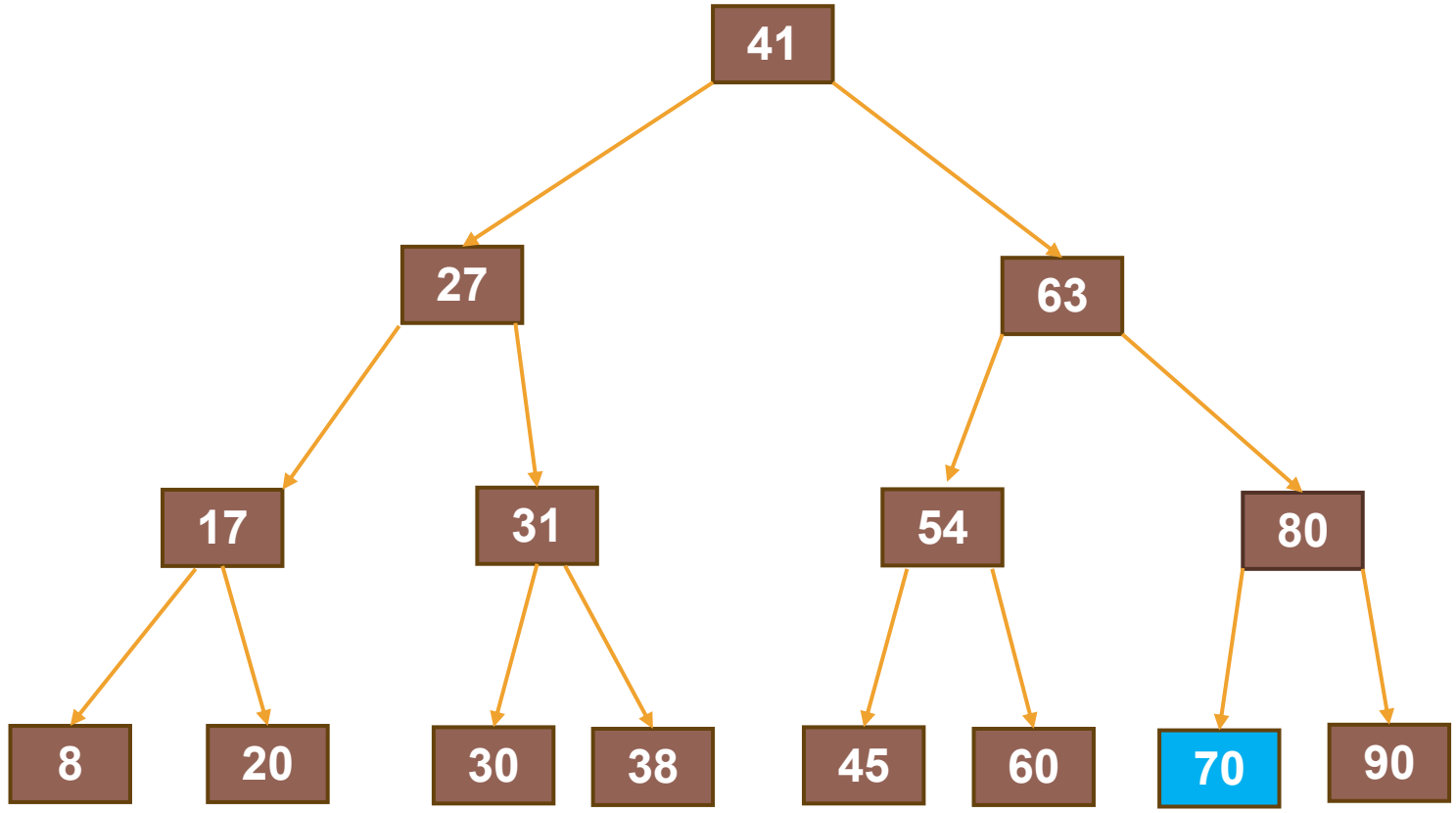
- Nếu nút vẫn có đủ số khóa tối thiểu ($\geq(m/2)-1$), xong.
- Nếu nút có ít khóa hơn mức tối thiểu, ta phải thực hiện **tái cân bằng**:
  - **Mượn (borrow)**: Nếu một nút anh em gần kề có thừa khóa, ta mượn một khóa từ nó và di chuyển một khóa từ nút cha xuống để cân bằng.
  - **Hợp nhất (merge)**: Nếu các nút anh em cũng chỉ có số khóa tối thiểu, ta hợp nhất nút hiện tại, nút anh em và một khóa từ nút cha thành một nút duy nhất. Thao tác này có thể tiếp tục "lan truyền" lên trên nếu nút cha cũng bị thiếu khóa.

5 B - TREE

Thuật toán xóa khóa khỏi B-tree

Trường hợp 1: Xóa khóa tại nút lá:

Ví dụ: Xóa nút có khóa 70 khỏi B-tree sau (bậc 3).

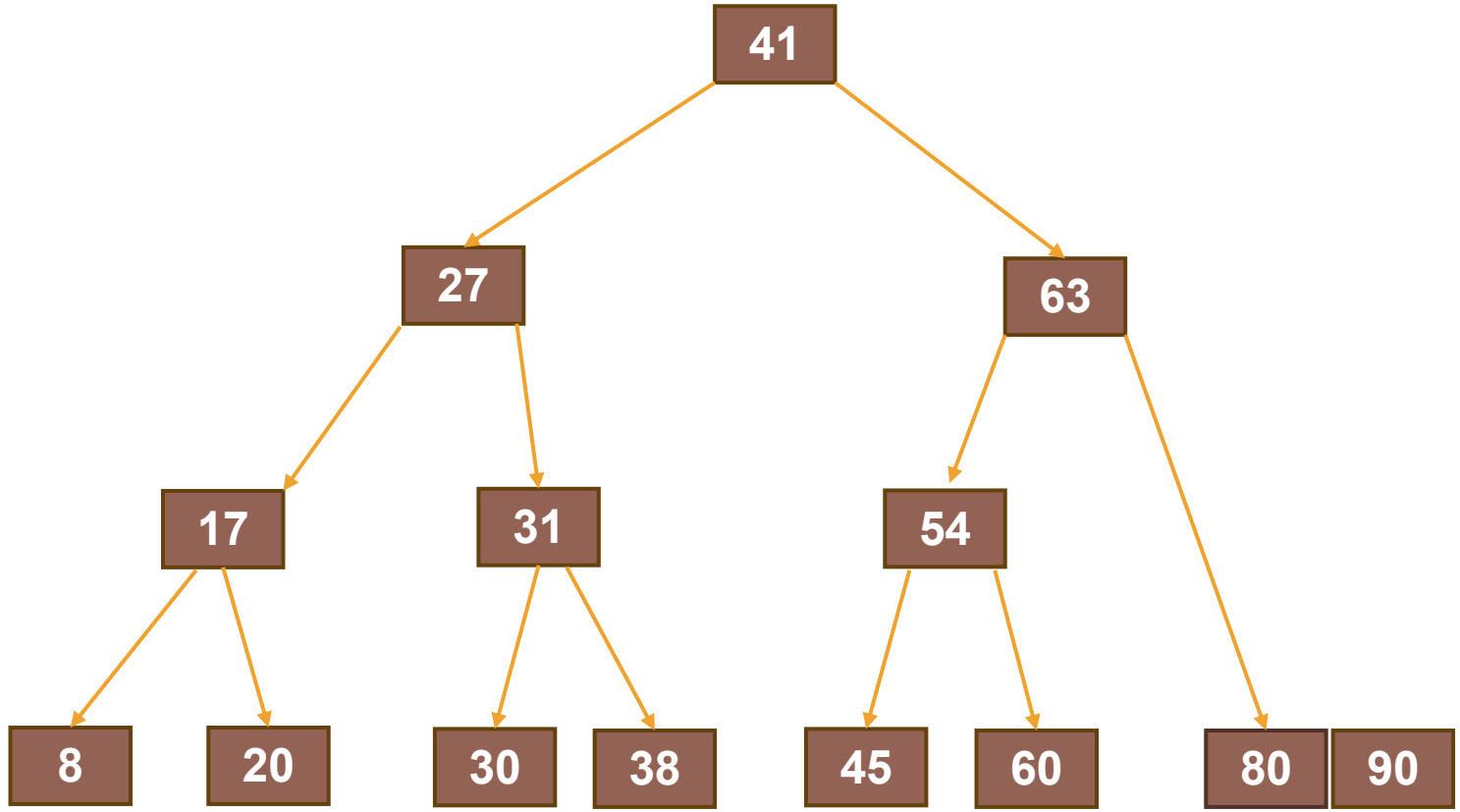


5 B - TREE

Thuật toán xóa khóa khỏi B-tree

Trường hợp 1: Xóa khóa tại nút lá:

Ví dụ: Xóa nút có khóa 70 khỏi B-tree sau (bậc 3).

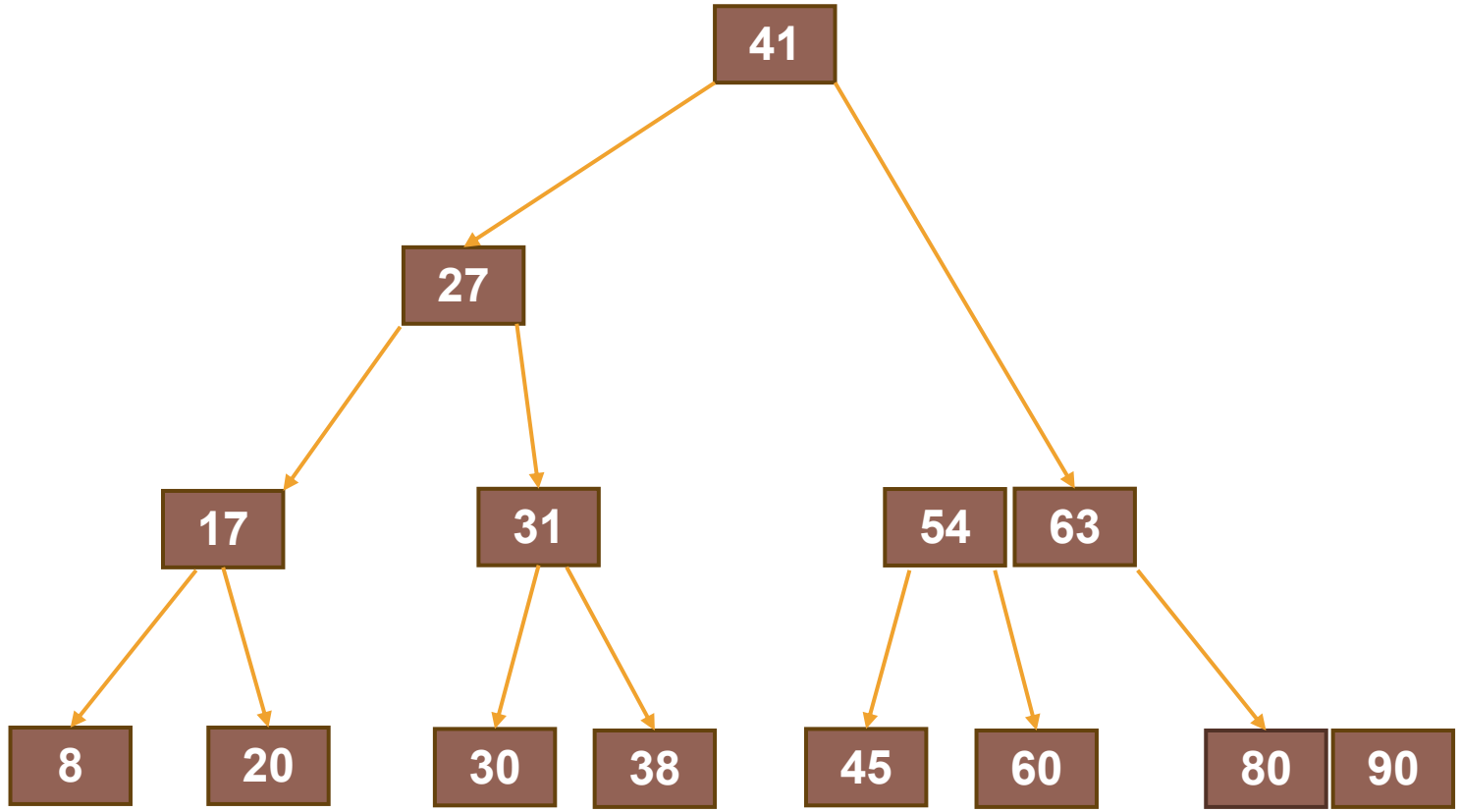


5 B - TREE

Thuật toán xóa khóa khỏi B-tree

Trường hợp 1: Xóa khóa tại nút lá:

Ví dụ: Xóa nút có khóa 70 khỏi B-tree sau (bậc 3).

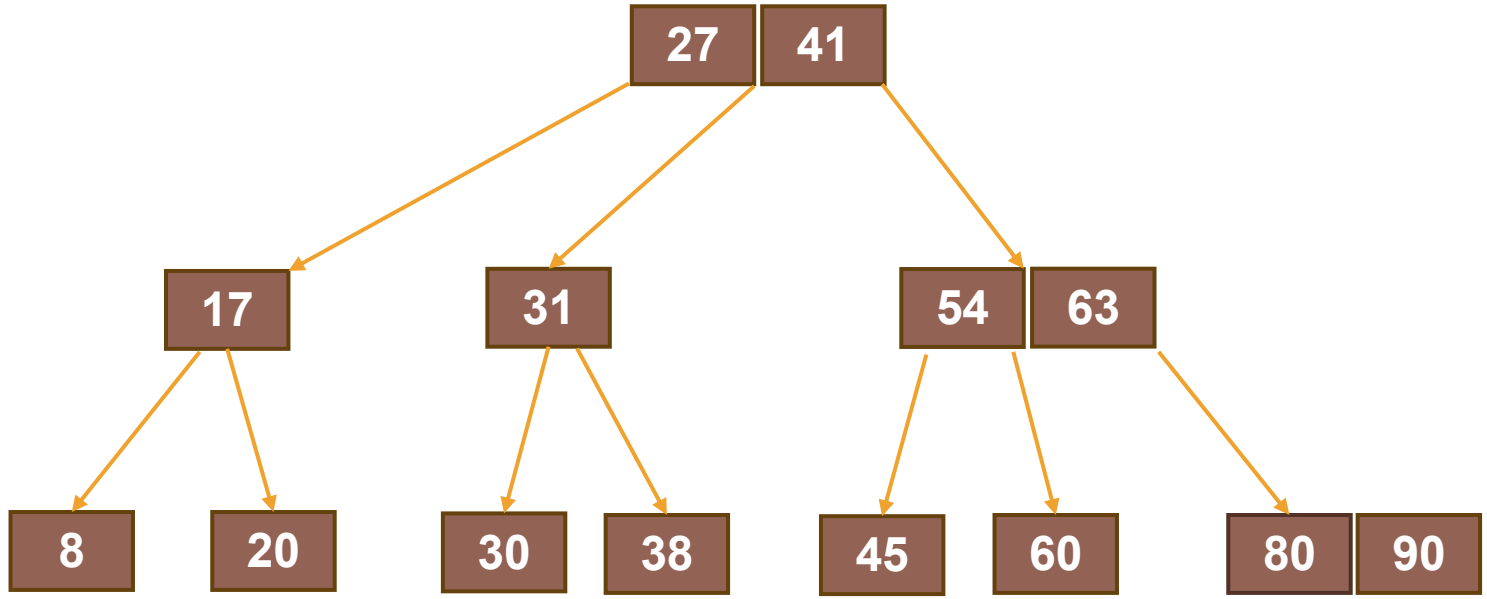


5 B - TREE

Thuật toán xóa khóa khỏi B-tree

Trường hợp 1: Xóa khóa tại nút lá:

Ví dụ: Xóa nút có khóa 70 khỏi B-tree sau (bậc 3).



5 B - TREE

Thuật toán xóa khóa khỏi B-tree

Trường hợp 2: Xóa khóa tại nút không phải lá

Bước 1: Tìm và xóa khóa.

Bước 2: Thay thế khóa đã xóa bằng **khóa lớn nhất** của cây con bên trái hoặc **khóa nhỏ nhất** của cây con bên phải.

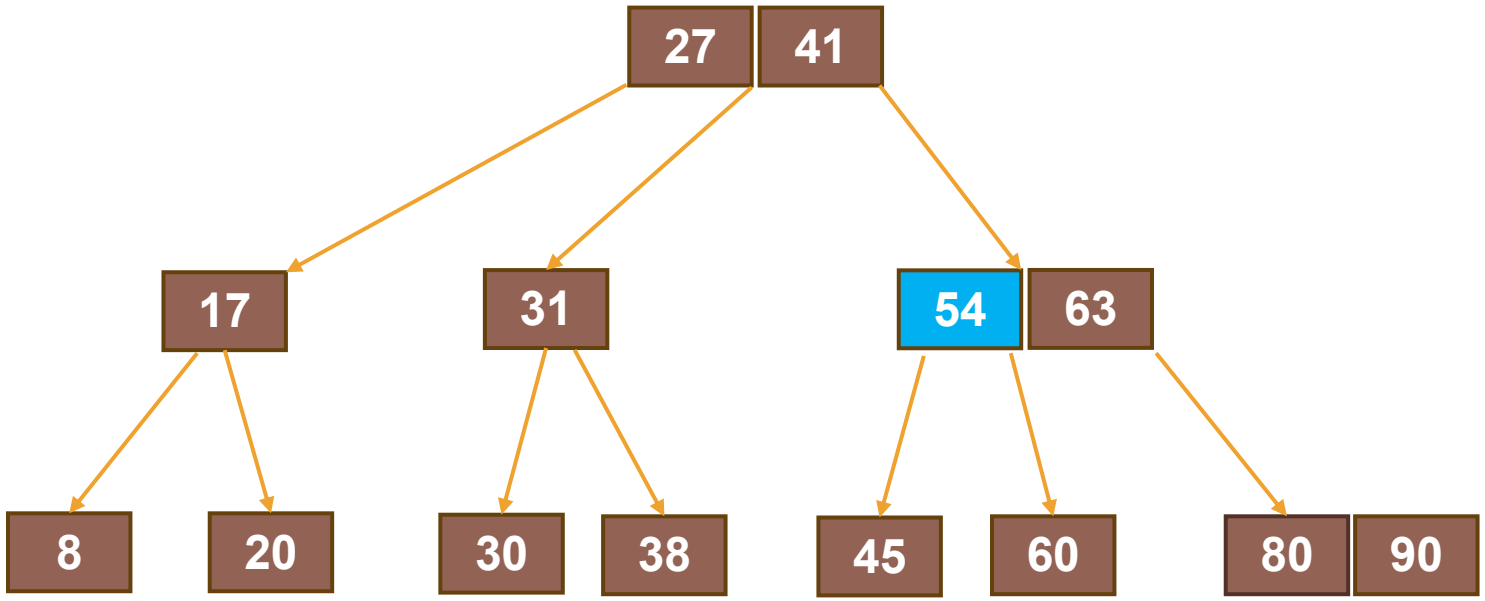
Bước 3: Xóa khóa đã thay thế đó ở nút lá (vị trí ban đầu của nó). Thao tác này sẽ trở về trường hợp 1.

5 B - TREE

Thuật toán xóa khóa khỏi B-tree

Trường hợp 2: Xóa khóa tại nút không phải lá

Ví dụ: Xóa khóa 54 khỏi cây B sau:

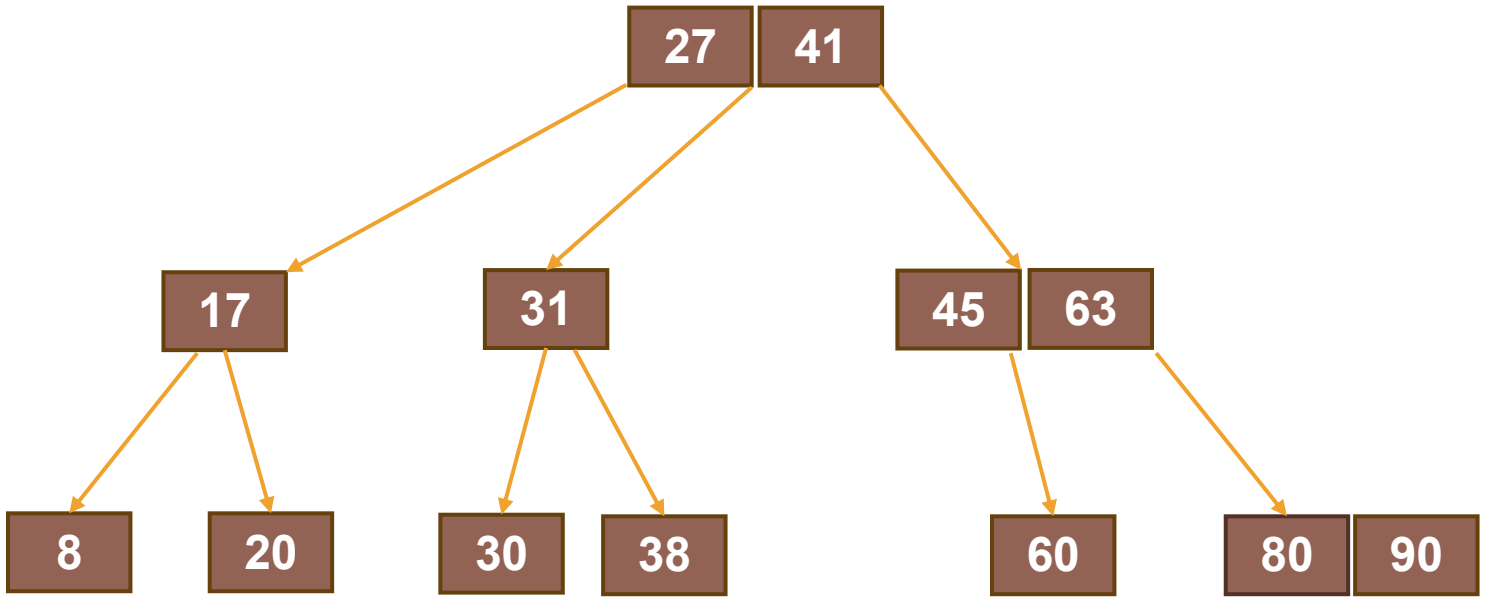


Thuật toán xóa khóa khỏi B-tree

Trường hợp 2: Xóa khóa tại nút không phải lá

Ví dụ: Xóa khóa 54 khỏi cây B sau:

Cách 1:

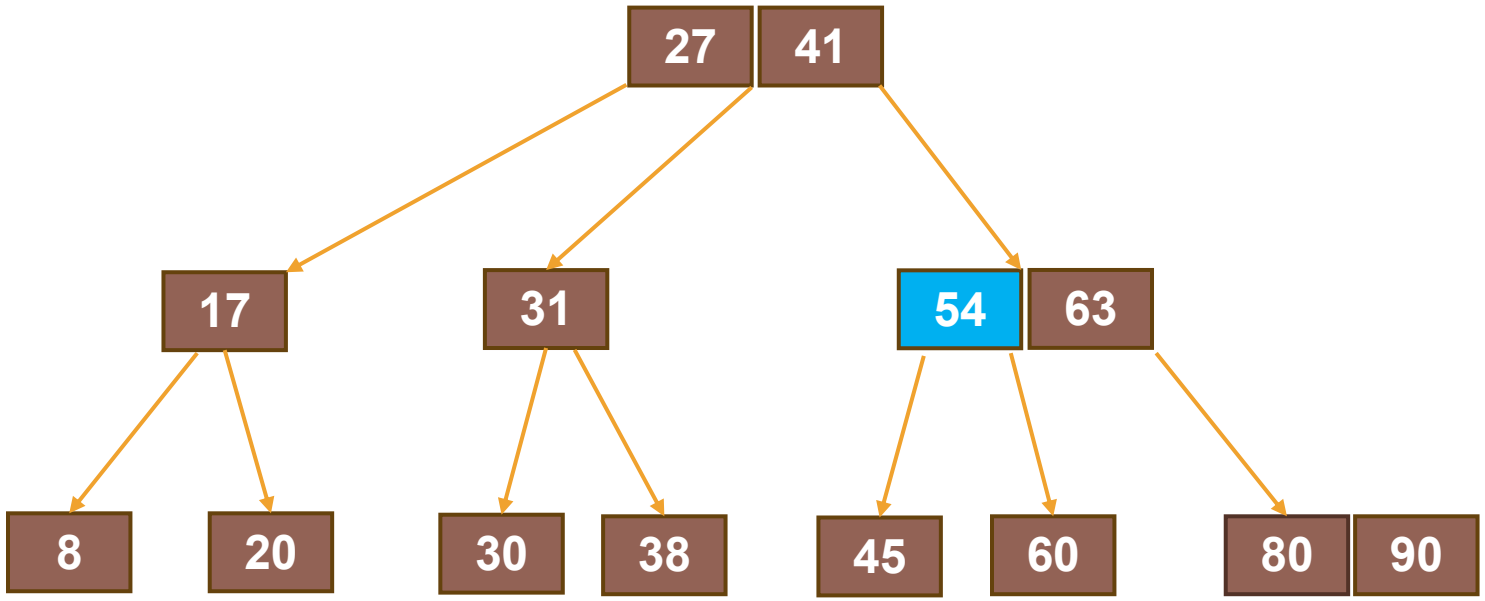


5 B - TREE

Thuật toán xóa khóa khỏi B-tree

Trường hợp 2: Xóa khóa tại nút không phải lá

Ví dụ: Xóa khóa 54 khỏi cây B sau:

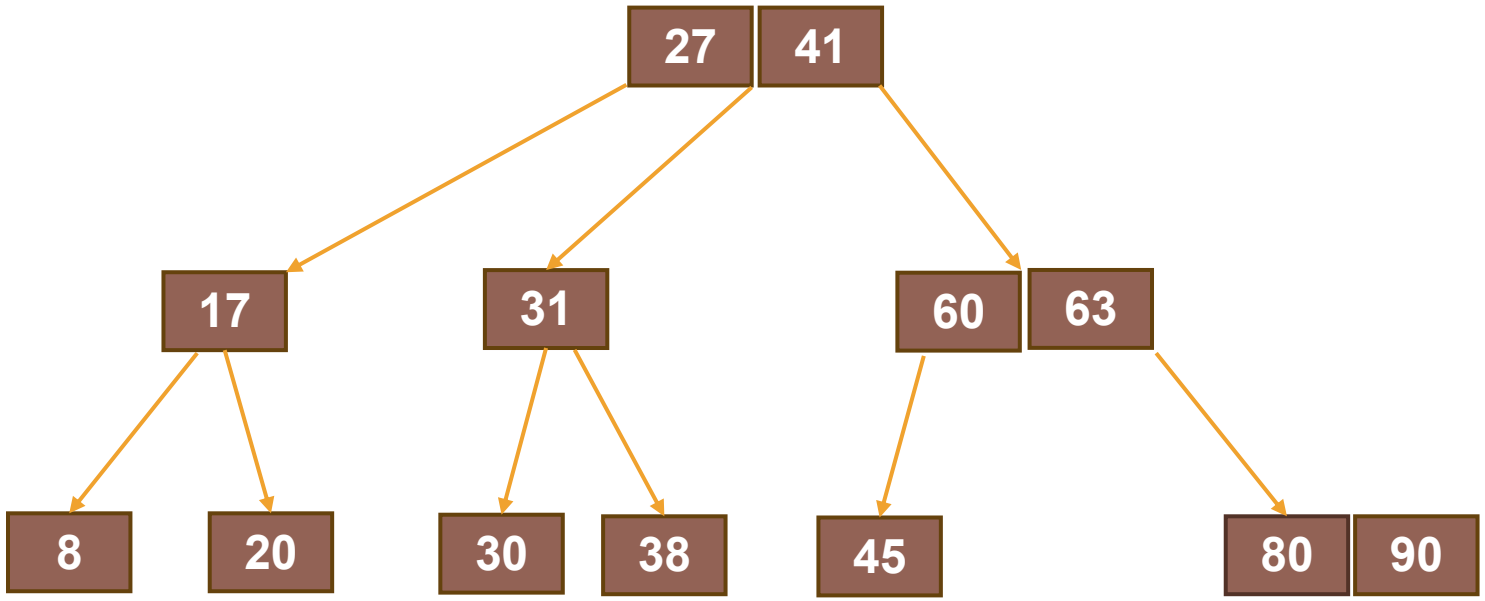


Thuật toán xóa khóa khỏi B-tree

Trường hợp 2: Xóa khóa tại nút không phải lá

Ví dụ: Xóa khóa 54 khỏi cây B sau:

Cách 2:



BÀI TẬP CHƯƠNG 5

Bài 1. Vẽ hình cây nhị phân tìm kiếm tạo ra từ cây rỗng bằng cách lần lượt thêm vào các khoá là các số nguyên: 64, 31, 23, 29, 45, 27, 20, 26, 88, 56.

a) Cho biết dãy kết quả khi duyệt qua cây trên theo các cách LNR, RNL, LRN, RLN, NLR, NRL.

b) Vẽ lại cây trên khi xóa nút có khóa bằng 31.

c) Cân bằng lại cây trên để được cây NPTK cân bằng.

Viết chương trình minh họa cho bài tập.

Bài 2. Viết chương trình minh họa cài đặt các giải thuật trên cây nhị phân tìm kiếm cân bằng.

Bài 3. Cây bị bệnh thì thành CTDL nào (cây bị bệnh là cây mà mỗi nút cha chỉ có 1 nút con)? Tại sao phải dùng cây AVL? So sánh Big-O giữa cây bị bệnh và cây AVL.