

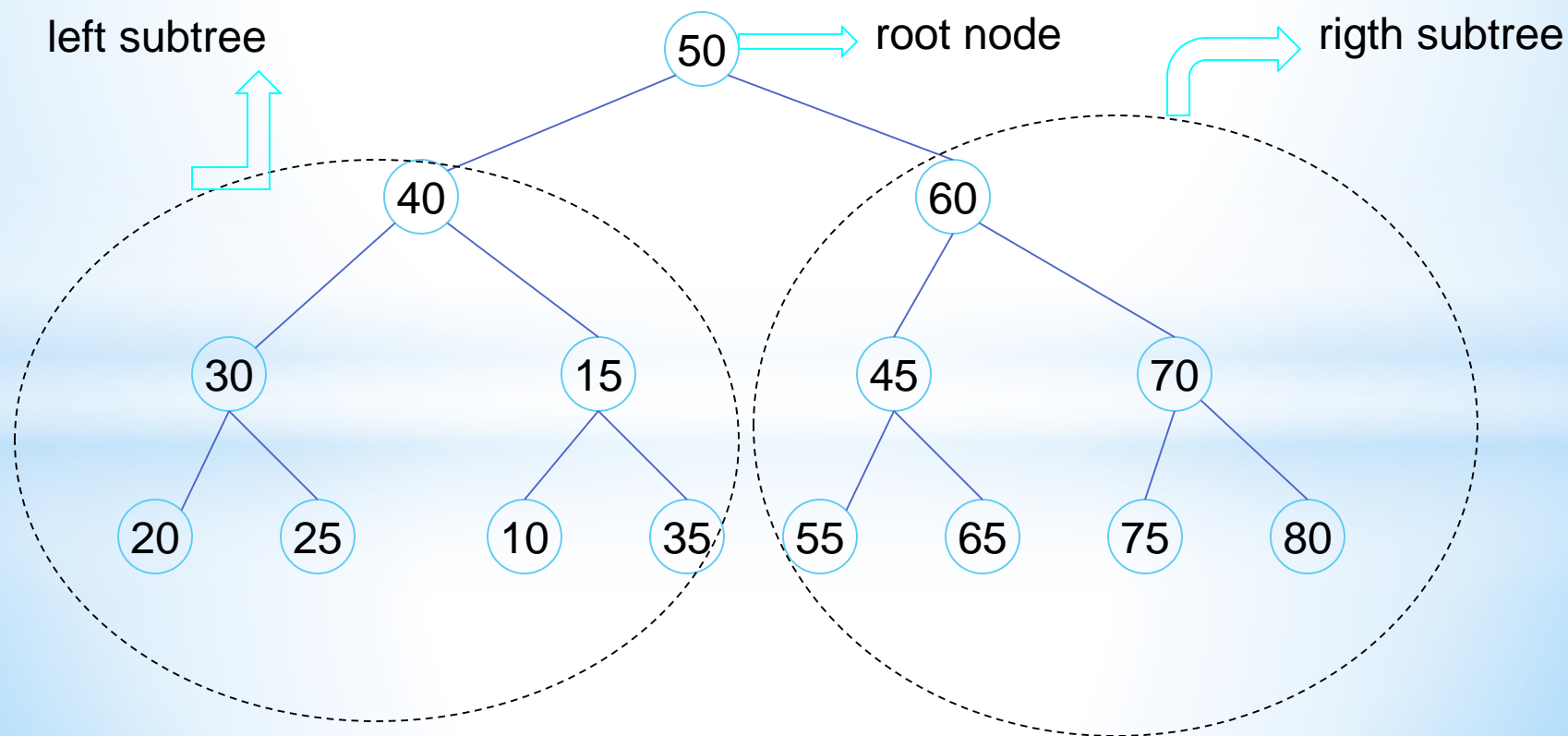
NỘI DUNG:

- 4.1. Định nghĩa và khái niệm
- 4.2. Biểu diễn cây nhị phân
- 4.3. Các thao tác trên cây nhị phân
- 4.4. Ứng dụng cây nhị phân
- 4.5. Cấu trúc Heap
- 4.6. Cây nhị phân tìm kiếm
- 4.7. Cây nhị phân tìm kiếm cân bằng (AVL)
- 4.8. Cây nhị phân tìm kiếm B-Tree
- 4.9. Một số cây nhị phân tìm kiếm quan trọng khác
- 4.10. CASE STUDY:

4.1. Định nghĩa và khái niệm

Định nghĩa: Tập hợp hữu hạn các node có cùng kiểu dữ liệu (có thể là tập \emptyset) được phân thành 3 tập:





- Tập thứ nhất có thể là \emptyset hoặc chỉ có một node gọi là node gốc (root).
- Hai tập con còn lại tự hình thành hai cây con bên trái (left subtree) và cây con bên phải (right subtree) của node gốc (hai tập con này cũng có thể là tập \emptyset).

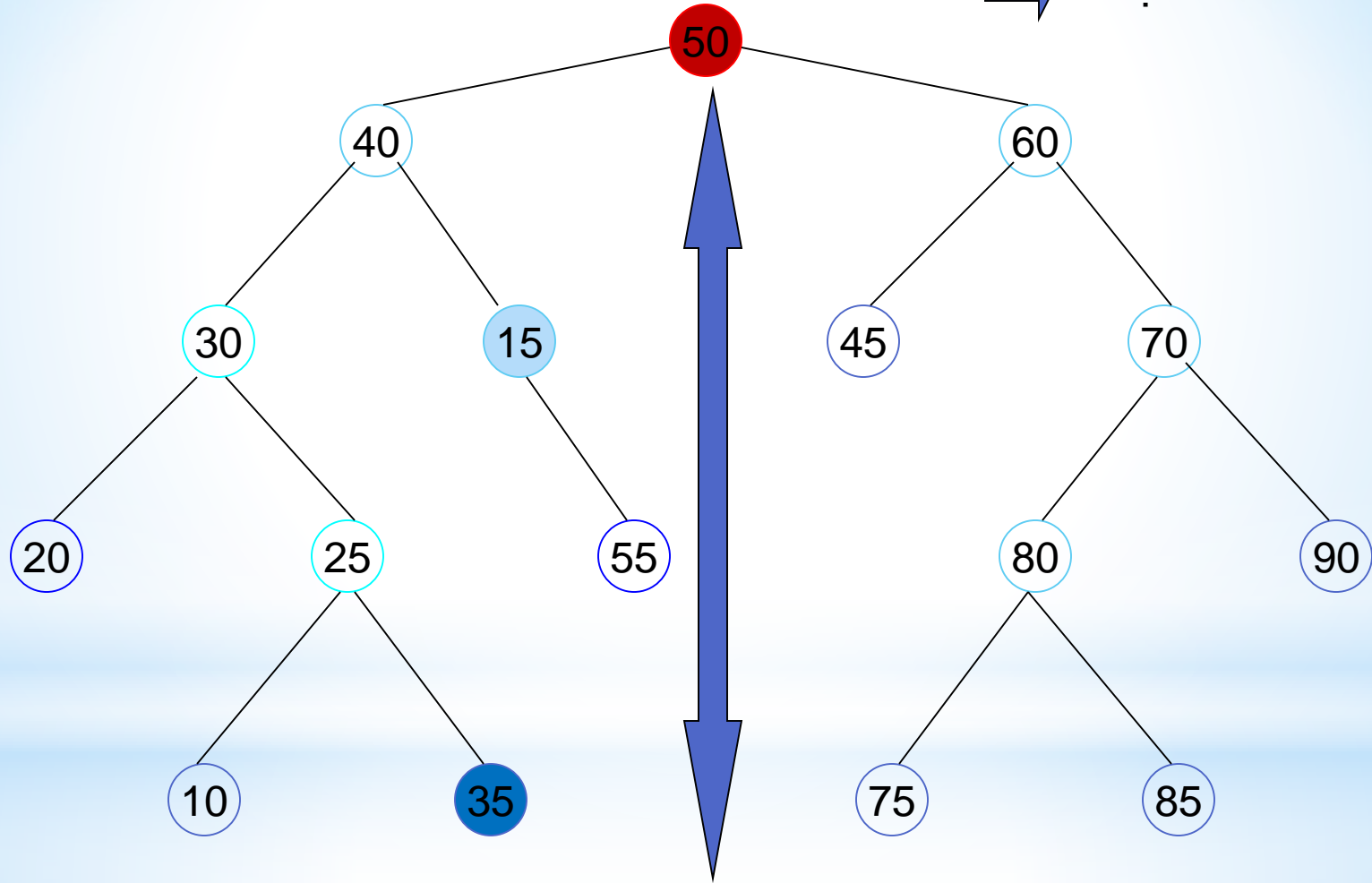


Một số khái niệm:

- **Node gốc (Root):** Node đầu tiên định hình cây.
- **Node cha (Father):** Node A là node cha của node B nếu B hoặc là node con bên trái của node A (left son) hoặc B là node con bên phải của node A (right son).
- **Node lá (Leaf):** Node không có node con trái, không có node con phải.
- **Node trung gian (Internal Node):** Node hoặc có node con trái, hoặc có node con phải, hoặc cả hai hoặc cả hai.
- **Node trước (Ancestor):** Node A gọi là node trước của node B nếu cây con node gốc là A chứa node B.
- **Node sau trái (left descendent):** node B là node sau bên trái của node A nếu cây con bên trái của node A chứa node B.
- **Node sau phải (right descendent):** node B là node sau bên phải của node A nếu cây con bên phải của node A chứa node B.
- **Node anh em (brother):** A và B là anh em nếu cả A và B là node con trái và node con phải của cùng một node cha.
- **Bậc của node (degree of node):** Số cây con tối đa của node.
- **Mức của node (level of node):** mức node gốc có bậc là 0, mức của các node khác trên cây bằng mức của node cha cộng thêm 1.
- **Chiều sâu của cây (depth of tree):** mức lớn nhất của node lá trong cây. Như vậy, độ sâu của cây bằng đúng độ dài đường đi dài nhất từ node gốc đến node lá.

Một số khái niệm:

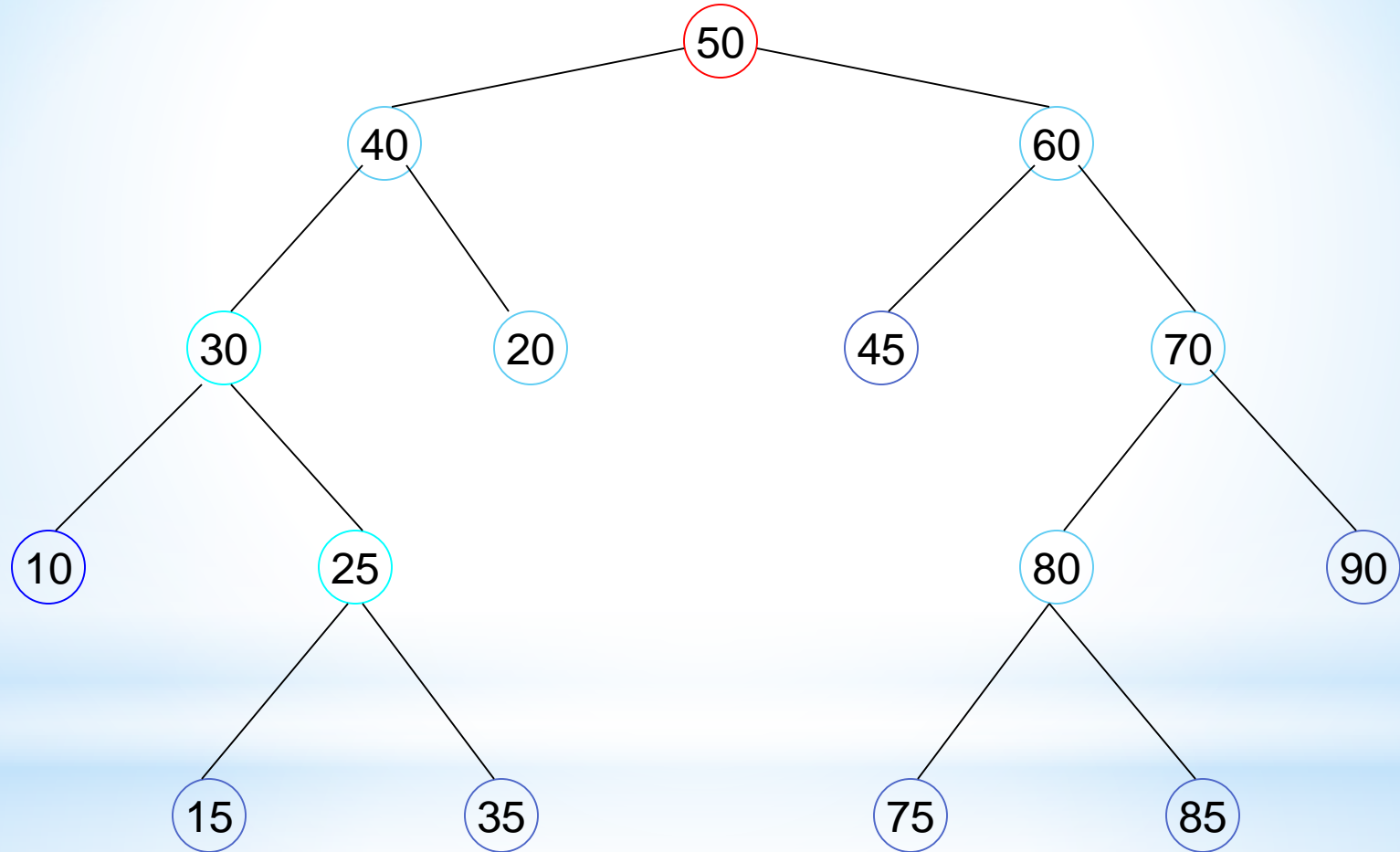
-  Node gốc
-  Node trung gian
-  Node lá
-  Độ sâu của cây



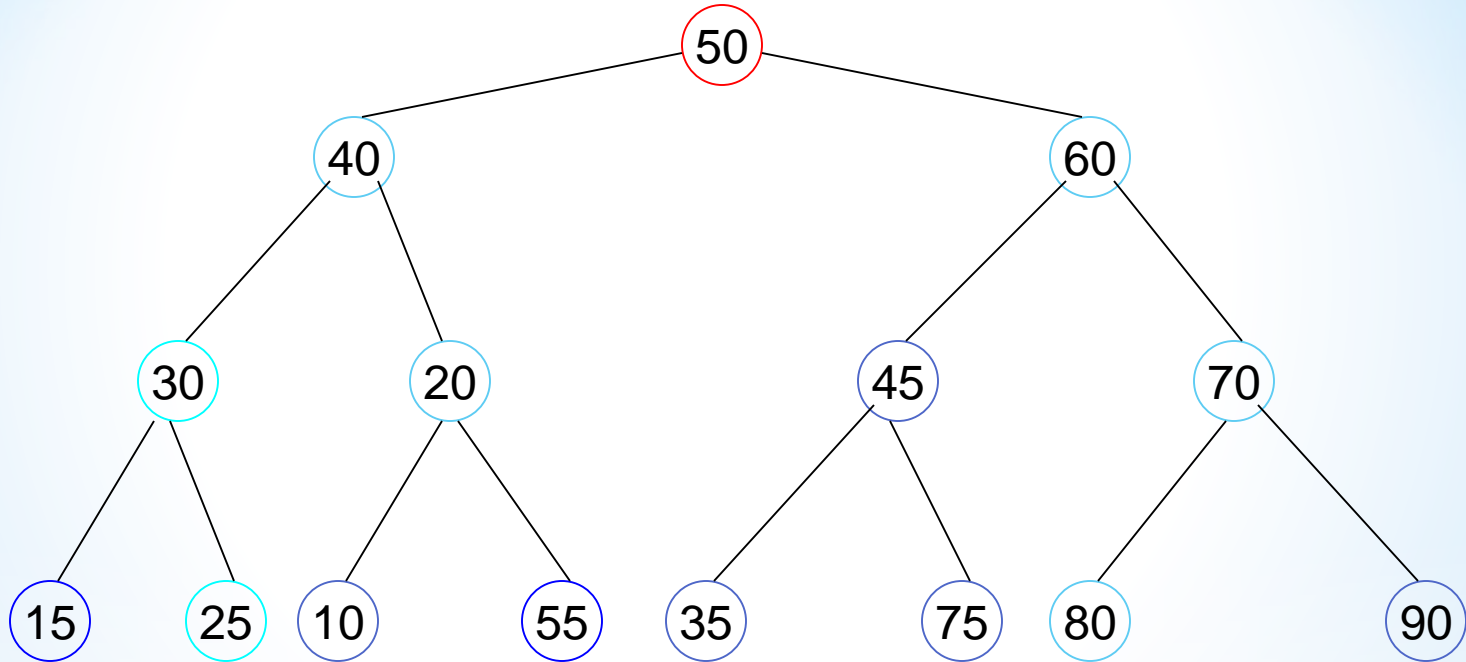
Các loại cây nhị phân:

- **Cây lệch trái:** Cây chỉ có node con bên trái.
- **Cây lệch phải:** Cây chỉ có node con bên phải.
- **Cây nhị phân đúng (strickly binary tree):** Node gốc và tất cả các node trung gian có đúng hai node con.
- **Cây nhị phân đầy (complete binary tree):** Cây nhị phân đúng và tất cả node lá đều có mức là d .
- **Cây nhị phân gần đầy (almost complete binary tree):** Cây nhị phân gần đầy có chiều sâu d là cây nhị phân thỏa mãn:
 - Tất cả node con có mức không nhỏ hơn $d-1$ đều có hai node con.
 - Các node ở mức d đầy từ trái qua phải.
- **Cây nhị phân hoàn toàn cân bằng.** Cây nhị phân có số node thuộc nhánh cây con trái và số node thuộc nhánh cây con phải chênh lệch nhau không quá 1.
- **Cây nhị phân tìm kiếm.** Cây nhị phân thỏa mãn điều kiện:
 - Hoặc là rỗng hoặc có một node gốc.
 - Mỗi node gốc có tối đa hai cây con. Nội dung node gốc lớn hơn nội dung node con bên trái và nhỏ hơn nội dung node con bên phải.
 - Hai cây con bên trái và bên phải cũng hình thành nên hai cây tìm kiếm.
- **Cây nhị phân tìm kiếm hoàn toàn cân bằng.** Cây nhị phân tìm kiếm có chiều sâu cây con bên trái và chiều sâu cây con bên phải chênh lệch nhau không quá 1.

Cây nhị phân đúng: Node gốc và tất cả các node trung gian có đúng hai node con.

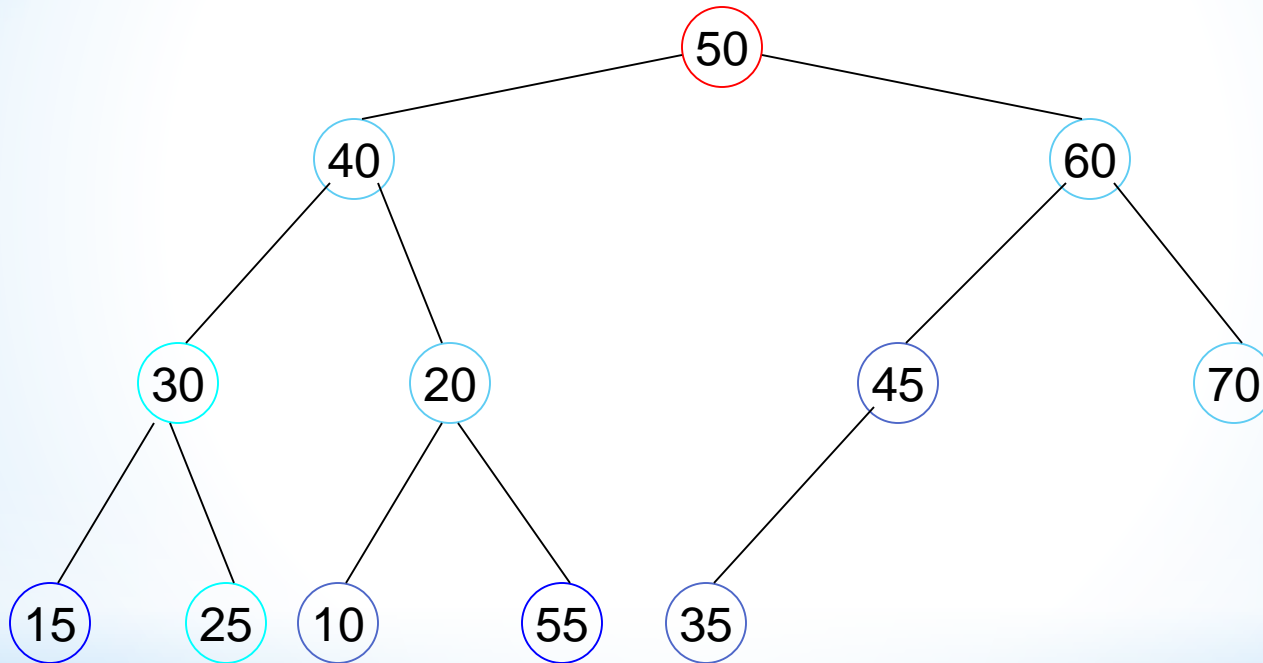


Cây nhị phân đầy: Cây nhị phân đúng và tất cả node lá đều có mức là d.

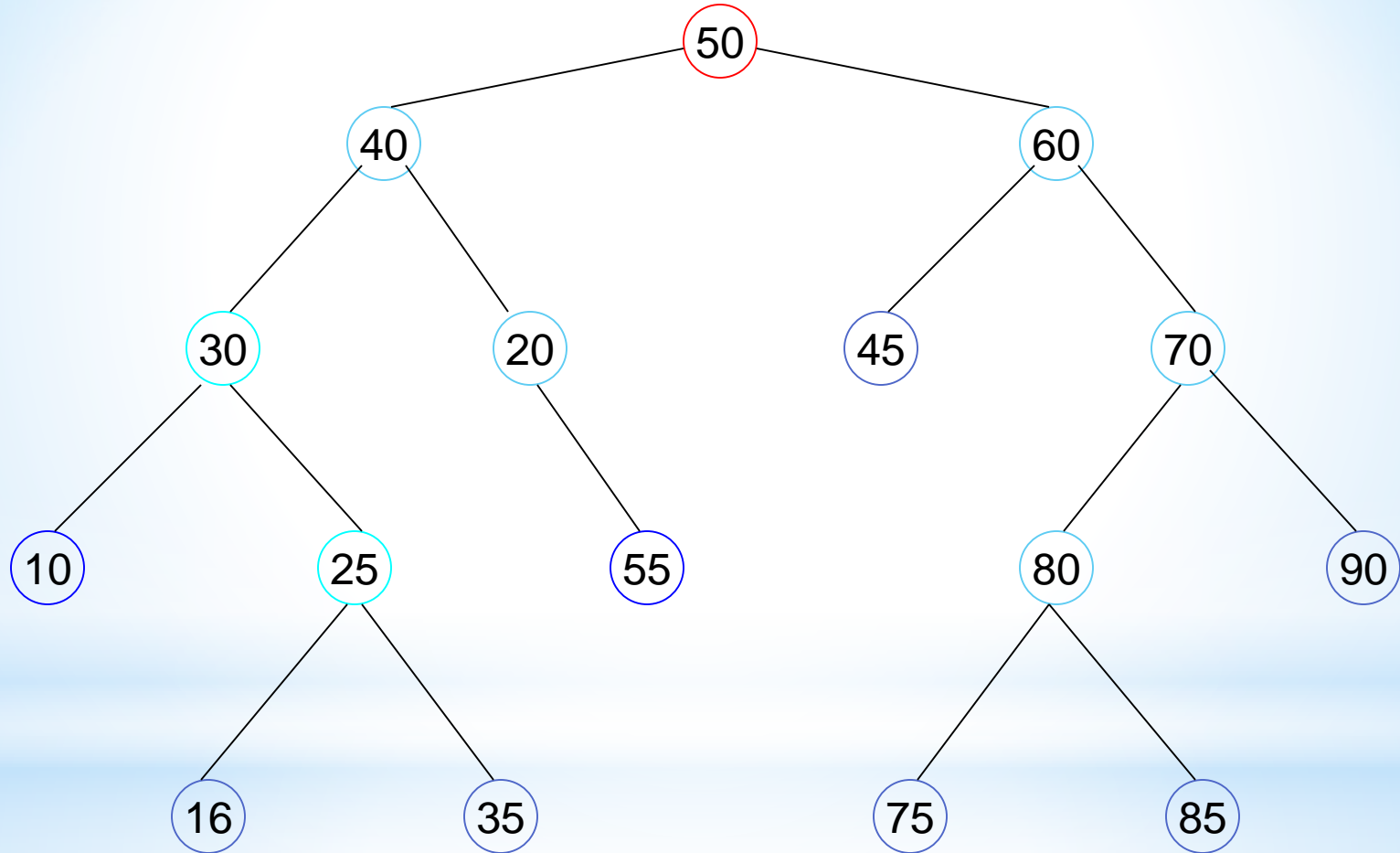


Cây nhị phân gần đầy: Cây nhị phân gần đầy có chiều sâu d là cây nhị phân thỏa mãn:

- Tất cả node con có mức không nhỏ hơn $d-1$ đều có hai node con.
- Các node ở mức d đầy dần từ trái qua phải.

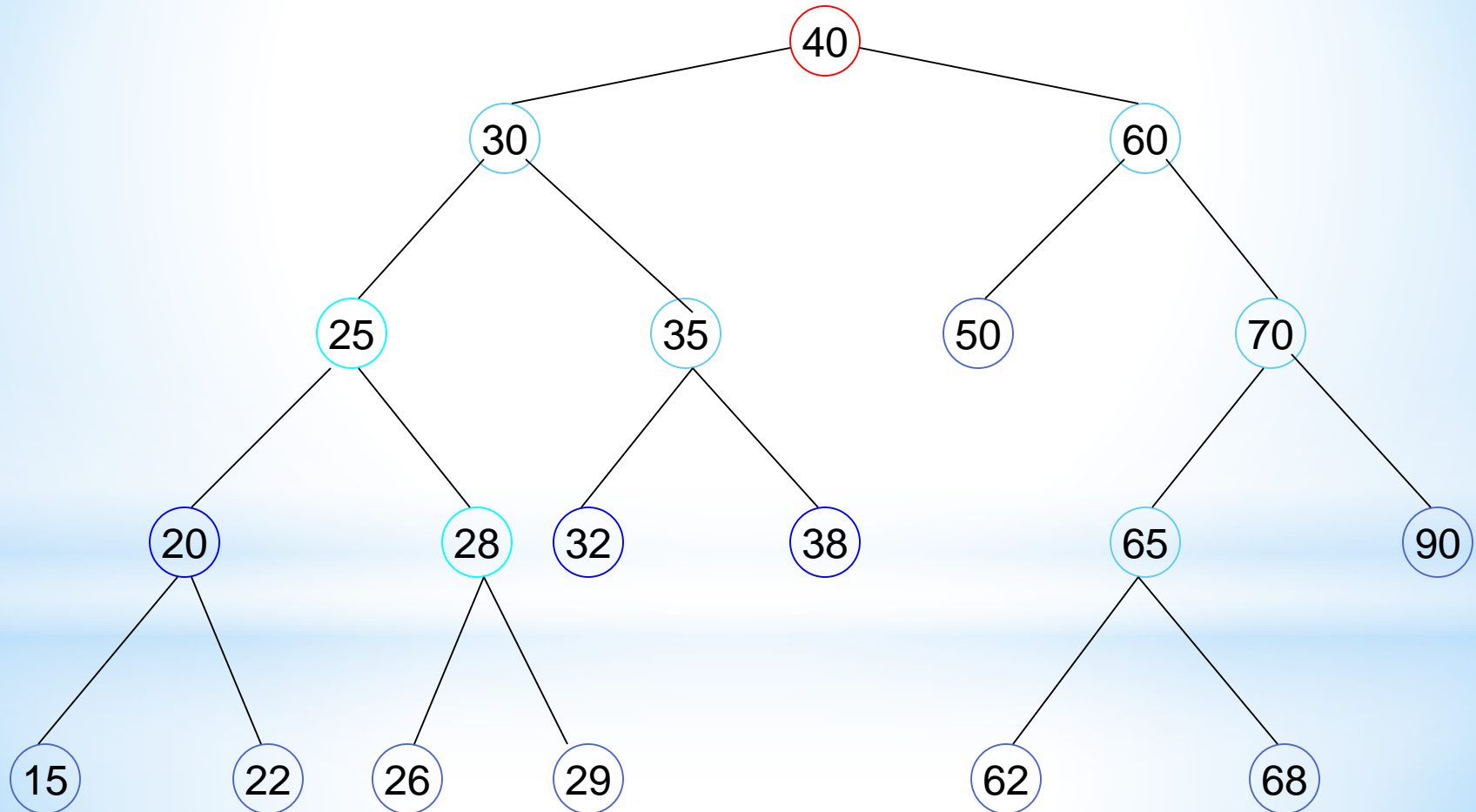


Cây nhị phân hoàn toàn cân bằng: Cây nhị phân có số node thuộc nhánh cây con bên trái và số node thuộc nhánh cây con bên phải chênh lệch nhau không quá 1.

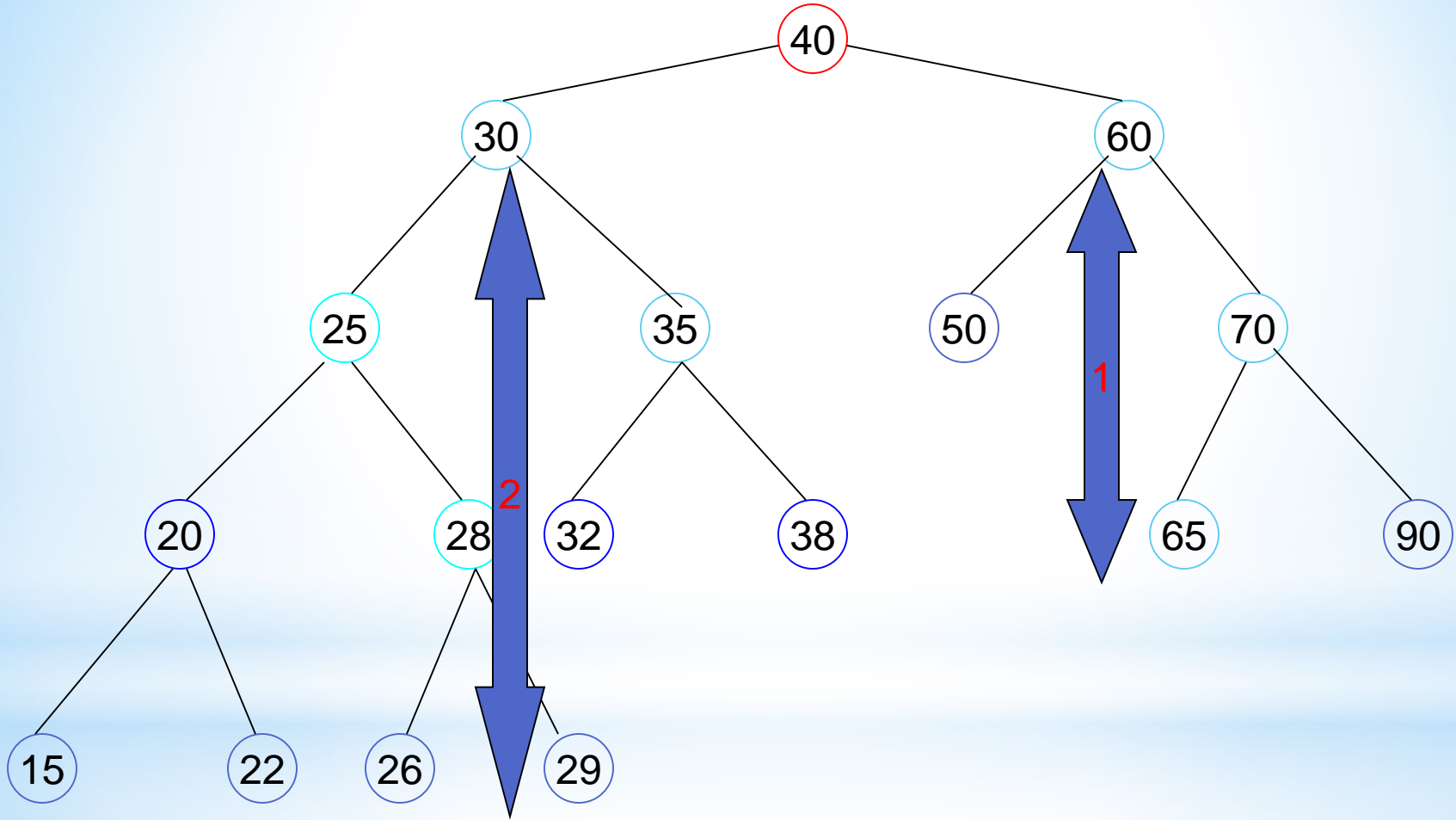


Cây nhị phân tìm kiếm: Cây nhị phân thỏa mãn điều kiện:

- Hoặc là rỗng hoặc có một node gốc.
- Mỗi node gốc có tối đa hai cây con. Nội dung node gốc lớn hơn nội dung node con bên trái và nhỏ hơn nội dung node con bên phải.
- Hai cây con bên trái và bên phải cũng hình thành nên hai cây tìm kiếm.



Cây nhị phân tìm kiếm hoàn toàn cân bằng: Cây nhị phân tìm kiếm có chiều sâu cây con bên trái và chiều sâu cây con bên phải chênh lệch nhau không quá 1.



4. 2. Biểu diễn liên tục

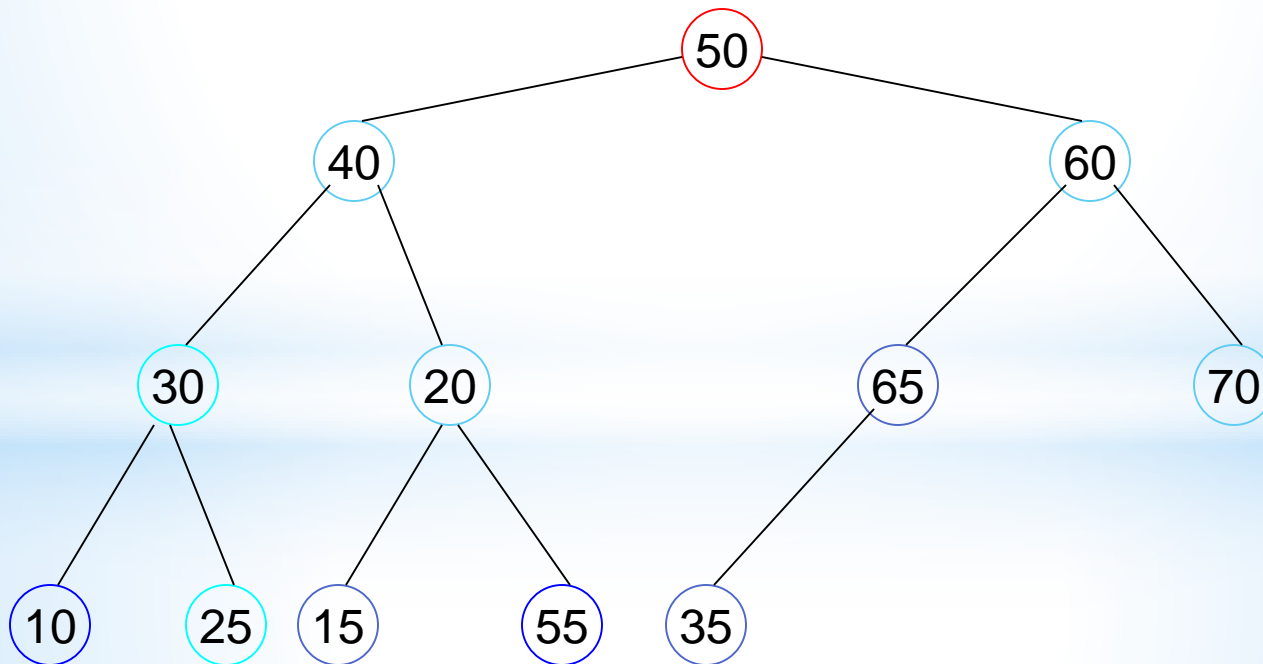
- **Biểu diễn liên tục:** Sử dụng mảng, trong đó:

- **Node gốc:** Lưu trữ ở vị trí 0.

- Nếu node cha lưu trữ ở vị trí p thì node con bên trái của nó được lưu trữ ở vị trí $2p+1$, node con phải được lưu trữ ở vị trí $2p+2$.

Ví dụ: với cây dưới đây sẽ được lưu trữ trong mảng `Tree[MAX]` như sau:

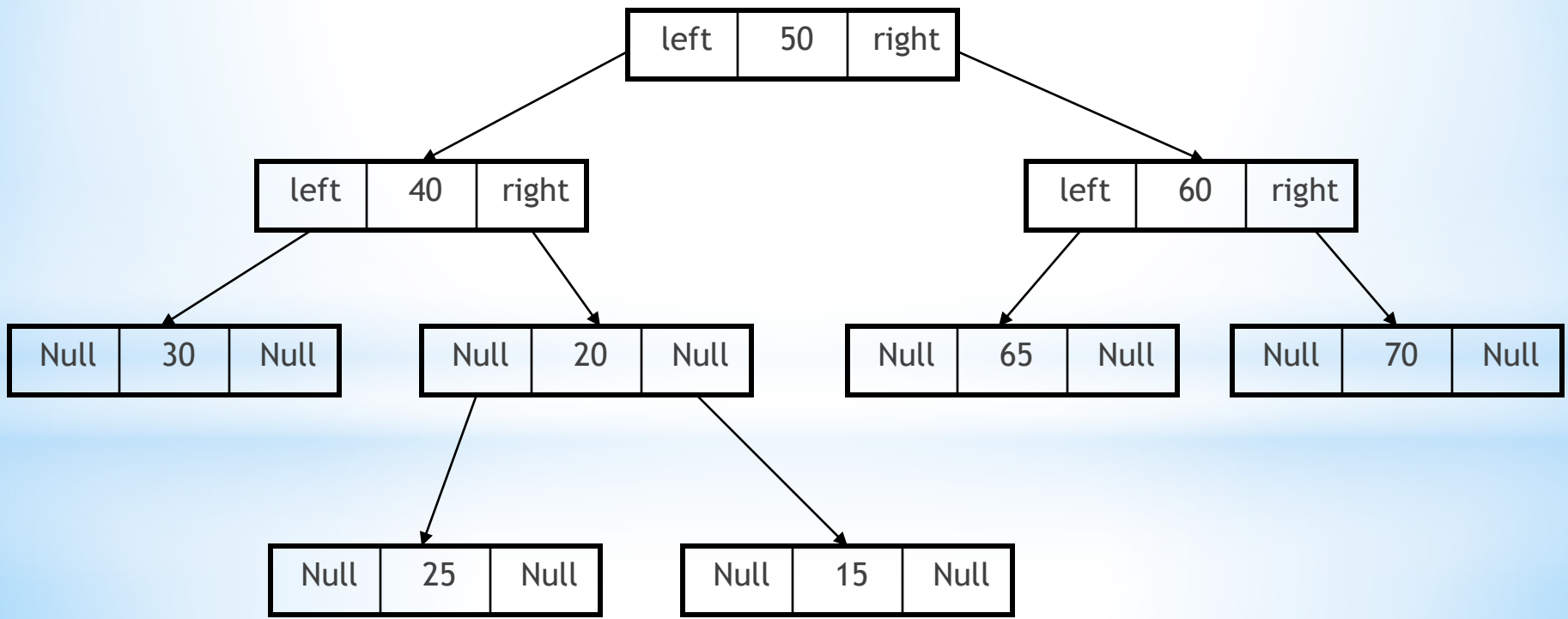
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	...
50	40	60	30	20	65	70	20	25	15	55	35	∅	∅	∅



4.2.1. Biểu diễn rời rạc:

- **Biểu diễn rời rạc:** Sử dụng danh sách liên kết.

```
typedef struct node {  
    Item Infor; //Thông tin của node  
    struct nde *left; //Con trỏ node bên trái  
    struct nde *right; //Con trỏ node bên phải  
} *Tree;
```



4. 3. Các thao tác trên cây nhị phân

- Tạo node gốc cho cây.
- Thêm vào node lá bên trái node p.
- Thêm vào node lá bên phải node p.
- Loại bỏ node lá bên trái node p.
- Loại bỏ node lá bên phải node p.
- Loại bỏ cả cây.
- Tìm kiếm node trên cây.
- Duyệt cây theo thứ tự trước.
- Duyệt cây theo thứ tự giữa.
- Duyệt cây theo thứ tự sau.

Biểu diễn cây nhị phân (sử dụng danh sách liên kết)

```
typedef struct node {  
    Item data; //Thông tin của node  
    struct nde *left;//Con trỏ cây con trái bên trái  
    struct nde *right;//Con trỏ cây con trái phải  
} *Tree;
```

Biểu diễn cây nhị phân (sử dụng danh sách liên kết)

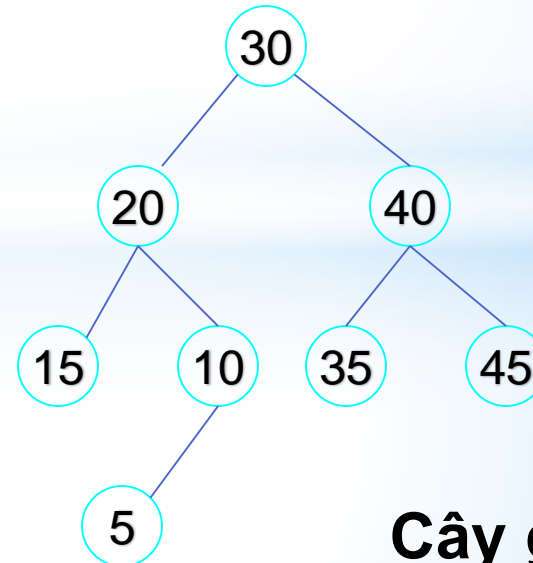
```
typedef struct node {  
    Item Infor; //Thông tin của node  
    struct nde *left; //Con trỏ node bên trái  
    struct nde *right; //Con trỏ node bên phải  
} *Tree;
```

Tạo node rời rạc có giá trị value:

```
node *Create_node(int value) {  
    node *tmp = new node; //cấp phát miền nhớ cho node  
    tmp->data = value; //thiết lập thành phần dữ liệu  
    tmp->left = NULL; //thiết lập liên kết cây con trái  
    tmp->right = NULL; //thiết lập liên kết cây con phải  
}
```


Thêm node vào cây một cách tùy ý:

```
node *Insert_random(node *root) {  
    root = Create_node(30); // node gốc là 30  
    root->left = Create_node(20); // node lá trái 30 là 20  
    root->right = Create_node(40); // node lá phải 30 là 40  
    (root->left)->left = Create_node(15); // node lá trái 20 là 15  
    (root->left)->right = Create_node(10); // node lá phải 20 là 10  
    (root->right)->left = Create_node(35); // node lá trái 40 là 35  
    (root->right)->right = Create_node(45); // node lá phải 40 là 45  
    ((root->left)->right)->left = Create_node(5); // node lá trái 10 là 5  
    return root;  
}
```



Cây gốc 30

Loại bỏ node lá trên cây/Loại bỏ cả cây:

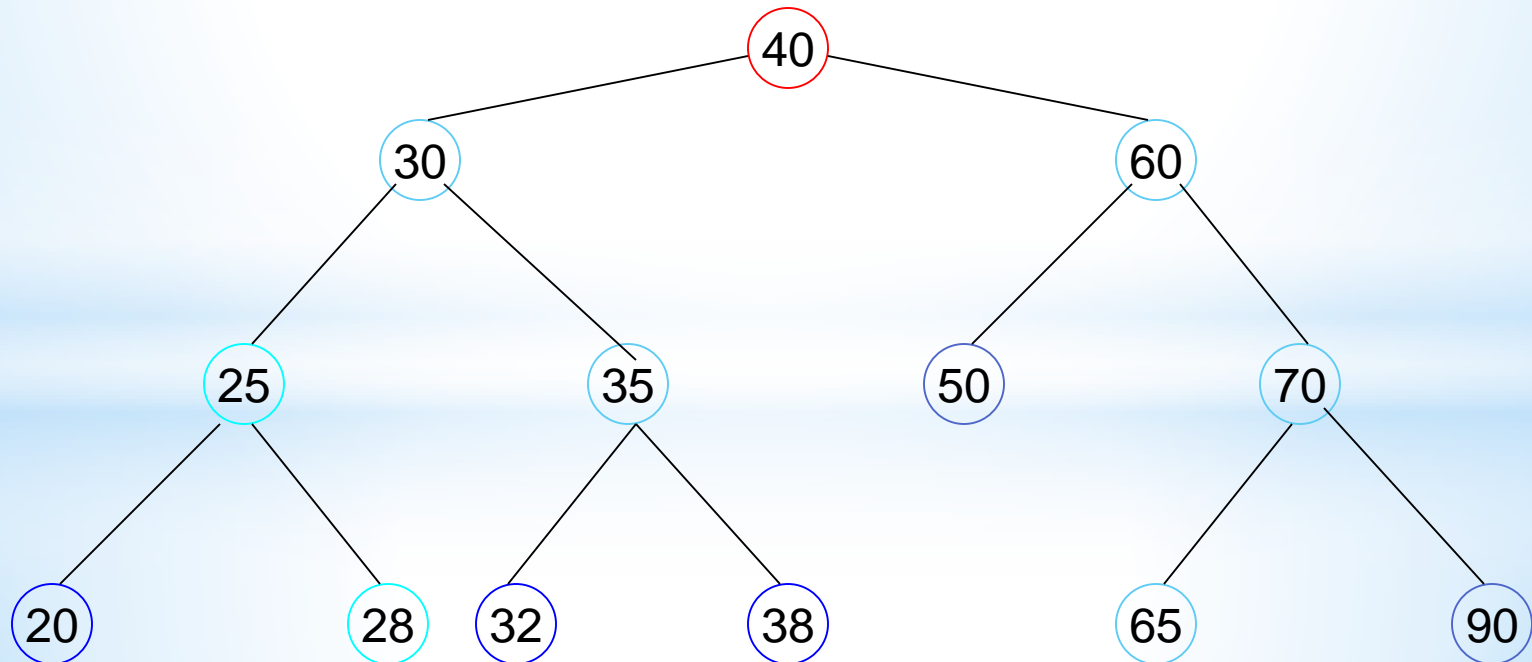
```
node *Delete_node(node *root) {  
    if(root!=NULL) {//nếu cây không rỗng  
        Delete_node(root->left);//loại bỏ cây con trái  
        Delete_node(root->right); //loại bỏ cây con phải  
        cout<<root->data<<" ";//node được loại bỏ  
        delete(root);//giải phóng miền nhớ cho node  
    }  
    root=NULL;//cây trở nên rỗng  
    return root;  
}  
  
//duyet cây theo thứ tự trước  
void preOrder(node *root) {  
    if(root!=NULL) { //nếu cây khác rỗng  
        cout<<root->data<<" ";//đưa ra nội dung node gốc  
        preOrder (root->left); //duyet cây theo thứ tự trước cây con trái  
        preOrder(root->right);//duyet cây theo thứ tự trước cây con phải  
    }  
}
```

Duyệt cây theo thứ tự trước (Node-Left-Right):

```
void preOrder(Tree *root ) {  
    if (root !=NULL ) {  
        <Thăm node>;  
        preOrder ( root -> left); //duyet thứ tự trước cây con trái  
        preOrder ( root -> right); //duyet thứ tự trước cây con phải  
    }  
}
```

Ví dụ.

preOrder(root) = 40, 30, 25, 20, 28, 35, 32, 38, 60, 50, 70, 65, 90.

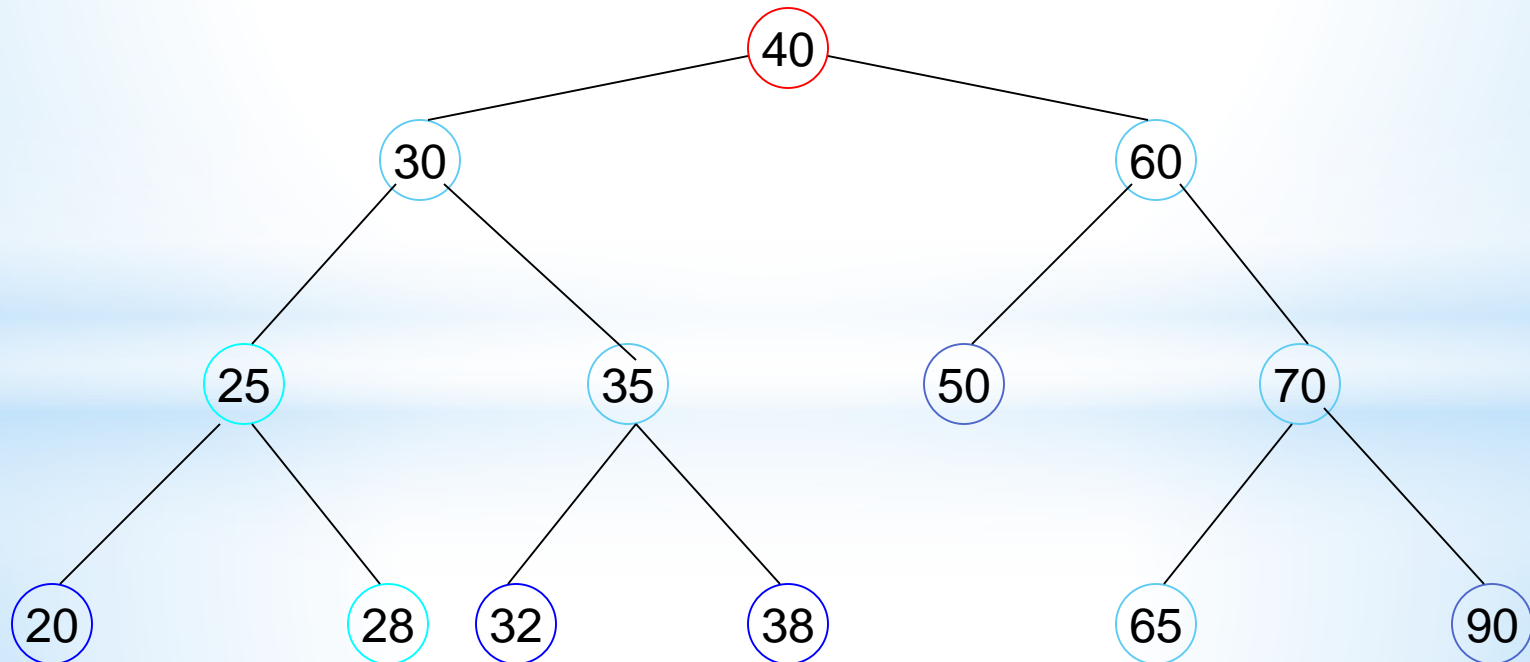


Duyệt cây theo thứ tự giữa (Left-Node-Right):

```
void inOrder(Tree *root) {  
    if (root != NULL) {  
        inOrder ( root -> left); //duyet thứ tự giữa cây con trái  
        <Thăm node>;  
        inOrder ( root -> right); //duyet thứ tự giữa cây con phải  
    }  
}
```

Ví dụ.

inOrder(root) = 20, 25, 28, 30, 32, 35, 38, 40, 50, 60, 65, 70, 90.

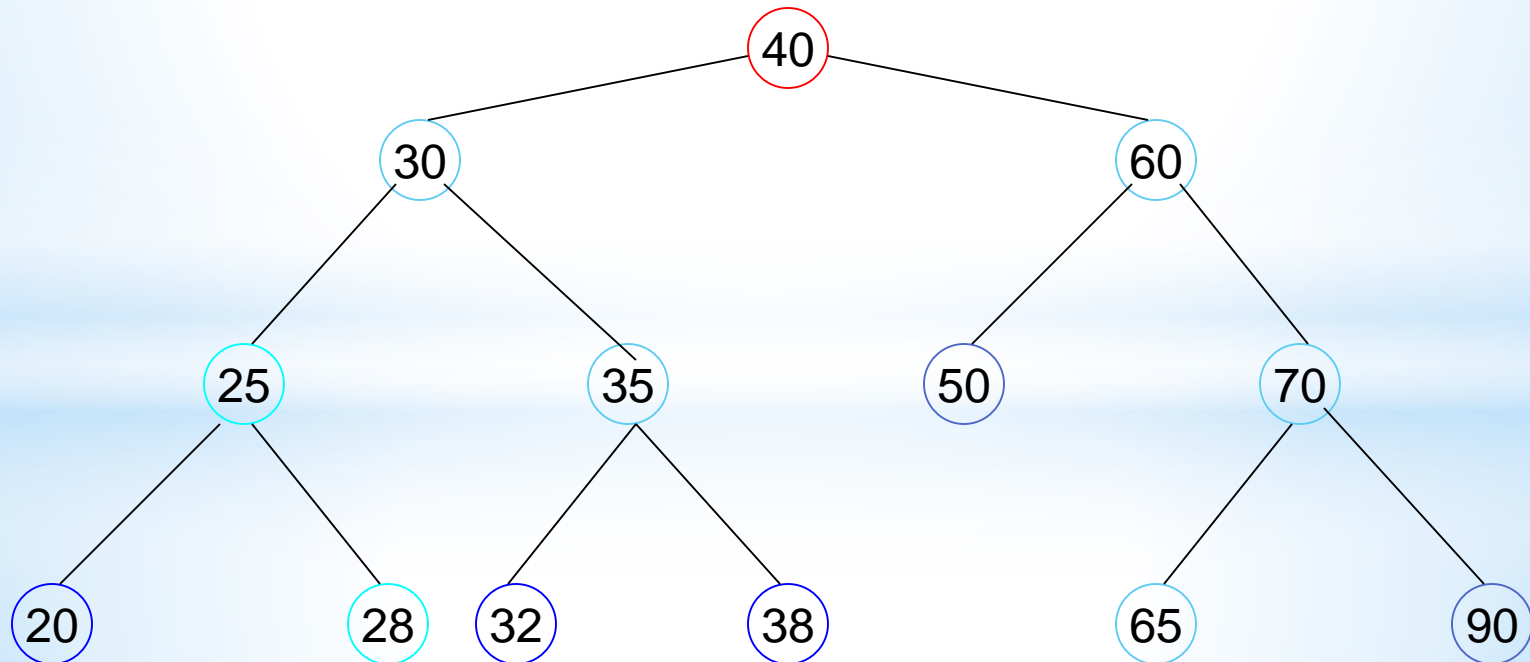


Duyệt cây theo thứ tự sau (Left-Right-Node):

```
void postOrder(Tree *root) {  
    if (root != NULL) {  
        postOrder ( root -> left); //duyet thứ sau cây con trái  
        postOrder ( root -> right); //duyet thứ tự sau cây con phải  
        <Thăm node>;  
    }  
}
```

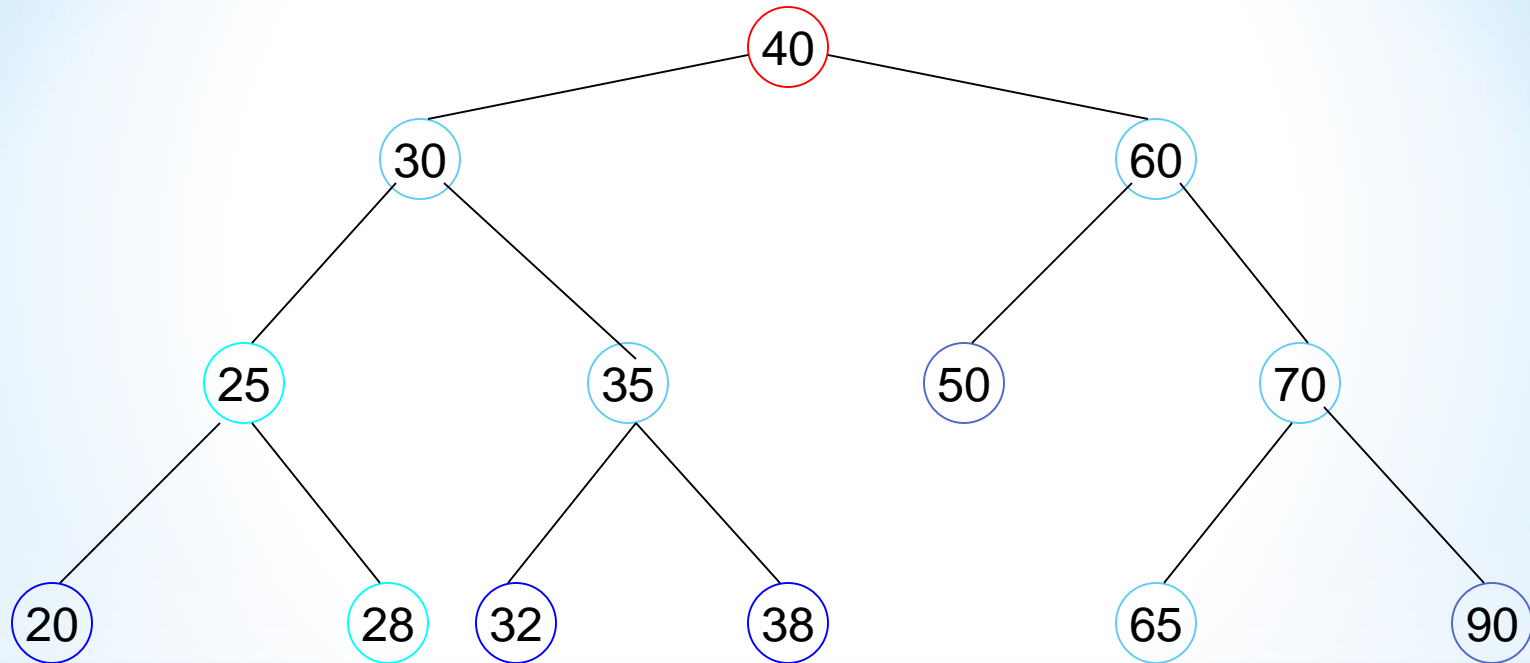
Ví dụ.

postOrder(**T**) = **20, 28, 25, 32, 38, 35, 30, 50, 65, 90, 70, 60, 40.**



Duyệt cây theo mức (Level Order): xem xét cây roof như một đồ thị vô hướng liên thông không có chu trình và sử dụng thuật toán BFS(roof).

LevelOrder(roof) = 40, 30, 60, 25, 35, 50, 70, 20, 28, 32, 65, 70.



LevelOrder(roof) = 40, 30, 60, 25, 35, 50, 70, 20, 28, 32, 65, 70.

Duyệt cây theo mức (Level Order): xem xét cây roof như một đồ thị vô hướng liên thông không có chu trình và sử dụng thuật toán BFS(roof).

//duyet cây theo mức

void levelOrder(Node* node) {

//bước cơ sở:

if (node == NULL) //nếu cây rỗng

return; //ta không có gì để duyệt

// bước lặp:

*queue<Node *> q; //tạo hàng đợi q thích nghi với tùy biến node*

q.push(node); //đưa node gốc vào hàng đợi

while (q.empty() == false) { //lặp đến khi hàng đợi rỗng

*Node *node = q.front(); //lấy node đầu hàng đợi*

cout << node->data << " "; //đây là node được duyệt

q.pop(); //loại node khỏi hàng đợi

if (node->left != NULL) //nếu node có cây con trái

q.push(node->left); //đưa node con trái vào hàng đợi

if (node->right != NULL) //nếu node có cây con phải

q.push(node->right); //đưa node con phải vào hàng đợi

}

}

4. 4. Một số bài toán quan trọng trên cây nhị phân

Bài toán biểu diễn cây theo mức. Một cây nhị phân được biểu diễn như một string trong đó các số trong string là giá của node theo phép duyệt theo mức, các xâu ký tự có giá trị “N” biểu diễn giá trị NULL của cây con trái hoặc cây con phải. Cho một biểu diễn cây nhị phân theo mức, hãy xây dựng lại cây nhị phân với độ phức tạp $O(n)$, trong đó n là số node của cây nhị phân.

Input:

- * Dòng đầu tiên đưa vào số lượng test T .
- * Các dòng tiếp theo, mỗi dòng đưa vào một test, mỗi test là một xâu ký tự biểu diễn cây nhị phân theo mức.

Output:

- * Đưa ra kết quả phép duyệt preOrder(inOrder, postOrder) của cây theo từng dòng.

BIỂU DIỄN CÂY NHỊ PHÂN THEO MỨC

Input:

1

1 2 3 N N 4 6 N 5 N N 7 N

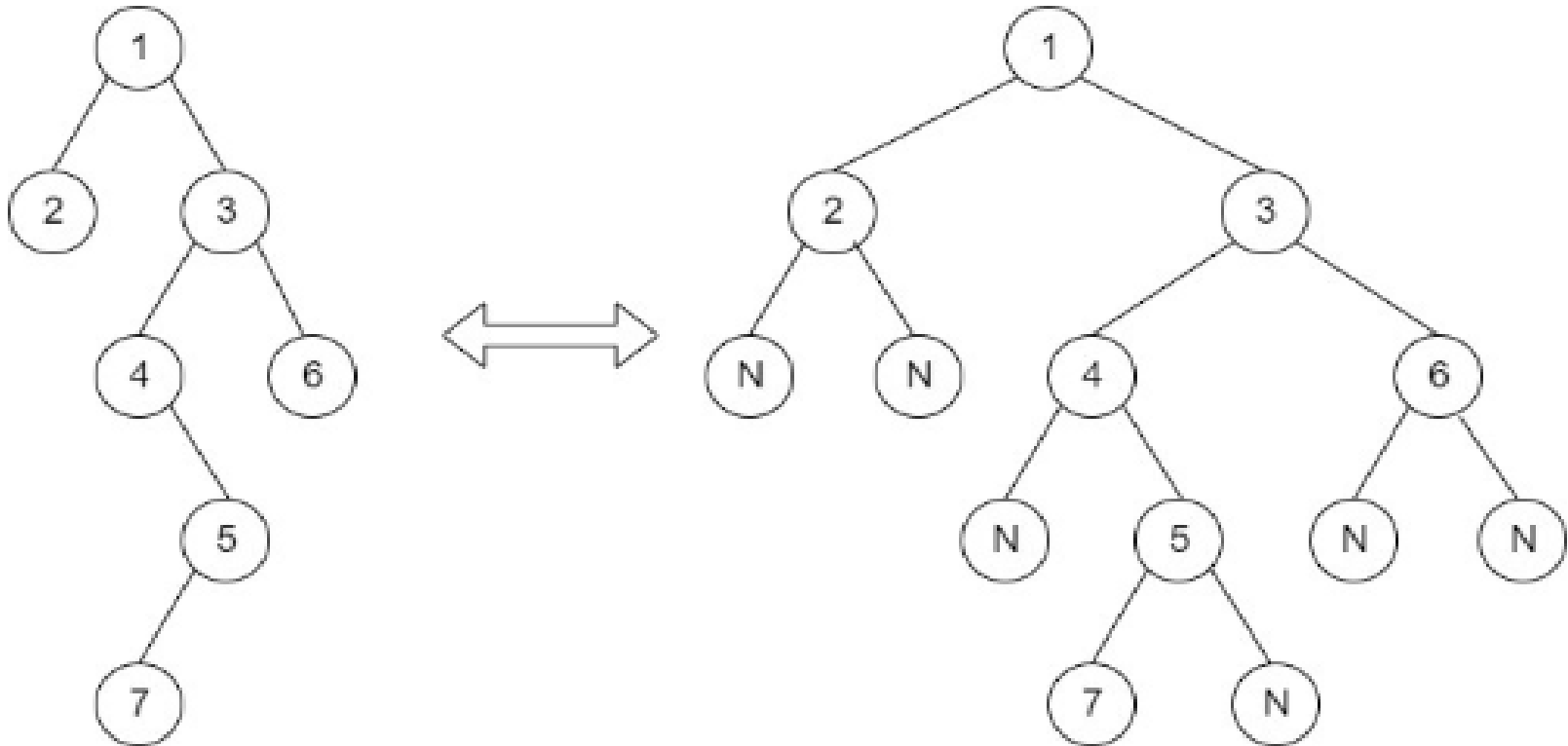
Output:

1 2 3 4 5 7 6 (preOrder)

2 1 4 7 5 3 6 (inOrder)

2 7 5 4 6 3 1 (postOrder)

1 2 3 4 6 5 7 (levelOrder)



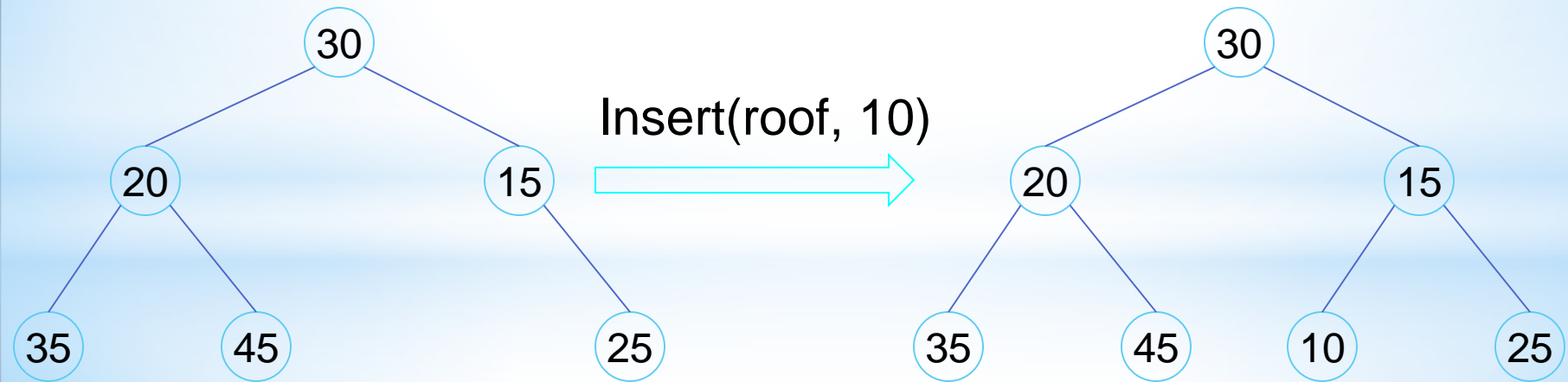
Bài toán bổ sung node theo mức theo mức (Level Insertion). Thêm node vào cây theo mức được thực hiện bằng cách tìm vị trí trống đầu tiên theo mức để thêm node vào cây.

Input:

- * Dòng đầu tiên đưa vào số lượng test T.
- * Các dòng tiếp theo, mỗi dòng đưa vào một test, mỗi test là một xâu ký tự biểu diễn cây nhị phân theo mức.

Output:

- * Đưa ra kết quả phép duyệt preOrder(inOrder, postOrder) của cây theo từng dòng.



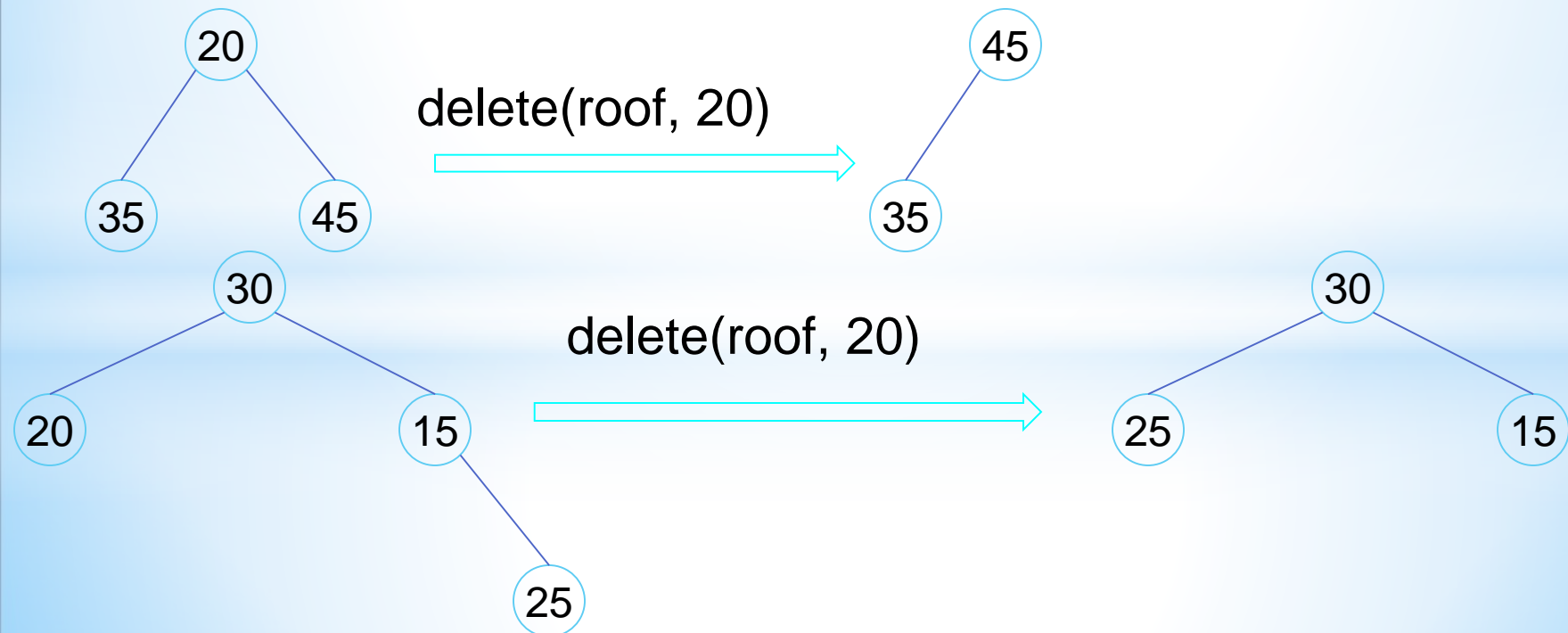
Bài toán loại bỏ node (deletion). Loại node trên cây được thực hiện bằng cách thay thế node loại bỏ bằng node sâu nhất trên cây.

Input:

- * Dòng đầu tiên đưa vào số lượng test T.
- * Các dòng tiếp theo, mỗi dòng đưa vào một test, mỗi test là một xâu ký tự biểu diễn cây nhị phân theo mức.

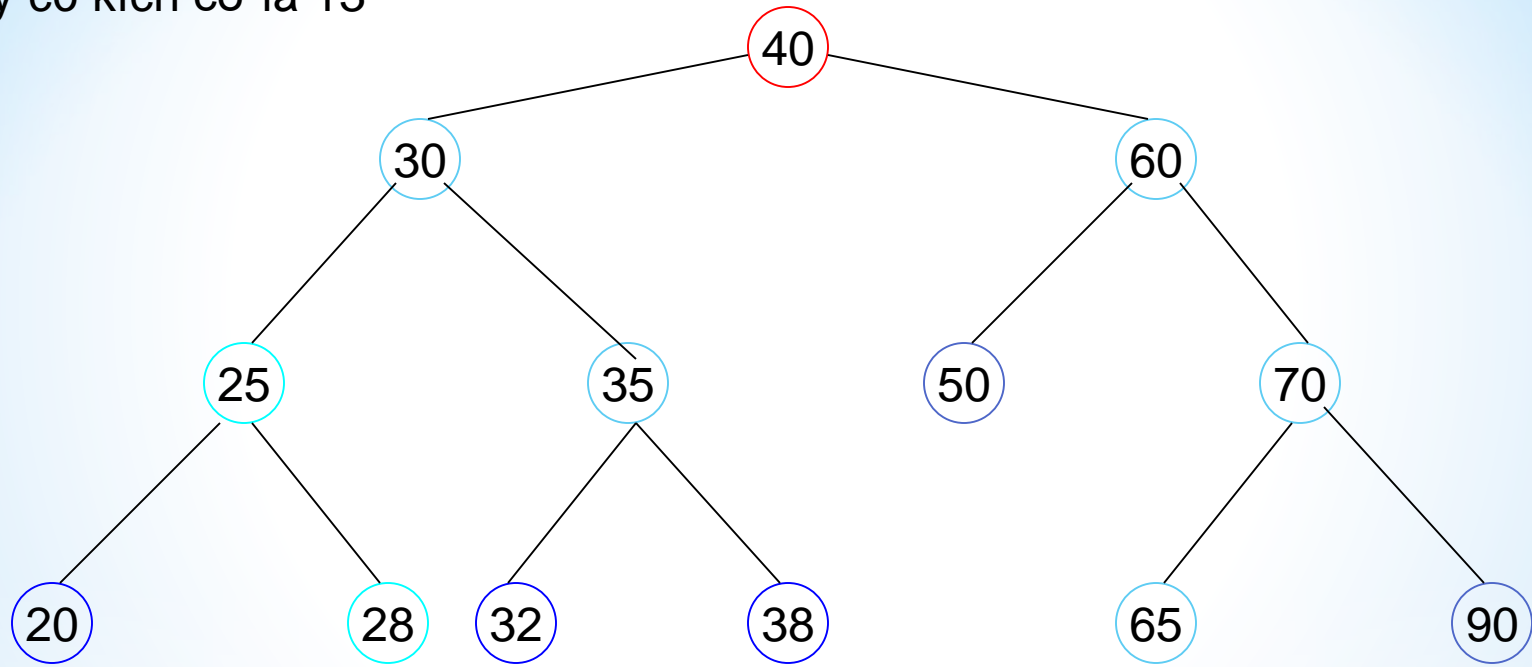
Output:

- * Đưa ra kết quả phép duyệt preOrder(inOrder, postOrder) của cây theo từng dòng.



4. 4. Một số bài toán quan trọng trên cây nhị phân

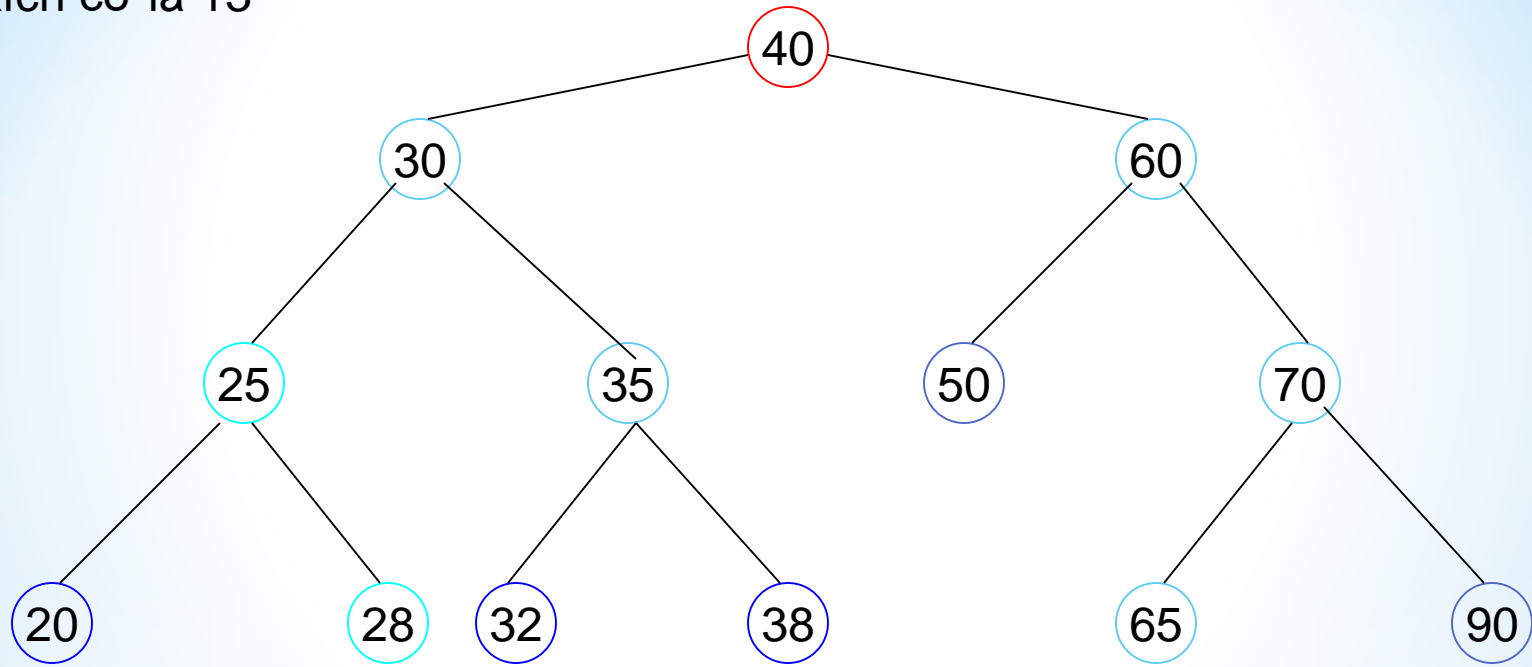
Bài toán bổ sung node theo mức. Ta định nghĩa kích cỡ của một cây là số node có thực trên cây. Bài toán đặt ra là cho trước một cây hãy tìm kích cỡ của cây. Ví dụ: cây dưới đây có kích cỡ là 13



```
int Size(struct node* T) {  
    if (T==NULL)  
        return 0;  
    else  
        return(Size(T->left) + 1 + Size(T->right));  
}
```

4. 4. Một số bài toán quan trọng trên cây nhị phân

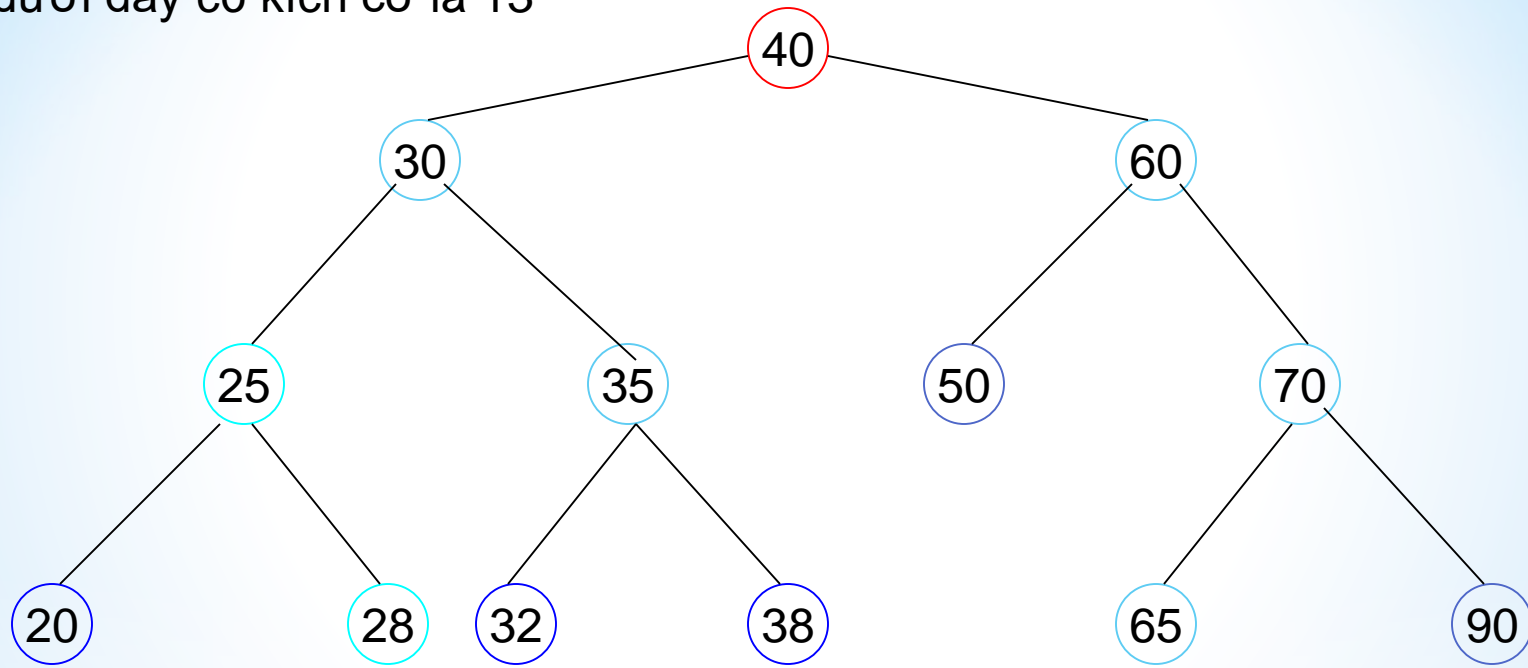
Bài toán loại node theo mức. Ta định nghĩa kích cỡ của một cây là số node có thực trên cây. Bài toán đặt ra là cho trước một cây hãy tìm kích cỡ của cây. Ví dụ: cây dưới đây có kích cỡ là 13



```
int Size(struct node* T) {  
    if (T==NULL)  
        return 0;  
    else  
        return(Size(T->left) + 1 + Size(T->right));  
}
```

4. 4. Một số bài toán quan trọng trên cây nhị phân

4.1. 1. Kích cỡ cây nhị phân (size of a tree). Ta định nghĩa kích cỡ của một cây là số node có thực trên cây. Bài toán đặt ra là cho trước một cây hãy tìm kích cỡ của cây. Ví dụ: cây dưới đây có kích cỡ là 13



```
int Size(struct node* T) {  
    if (T==NULL)  
        return 0;  
    else  
        return(Size(T->left) + 1 + Size(T->right));  
}
```


4.4. 2. Xác định hai cây nhị phân giống nhau. Ta định nghĩa hai cây nhị phân giống nhau nếu chúng có cùng chung node và mỗi node được sắp đặt giống nhau trên cây. Cho hai cây nhị phân bất kỳ, hãy xác định xem hai cây có giống nhau hay không?

Lời giải. Thuật toán giải quyết bài toán được thực hiện như sau:

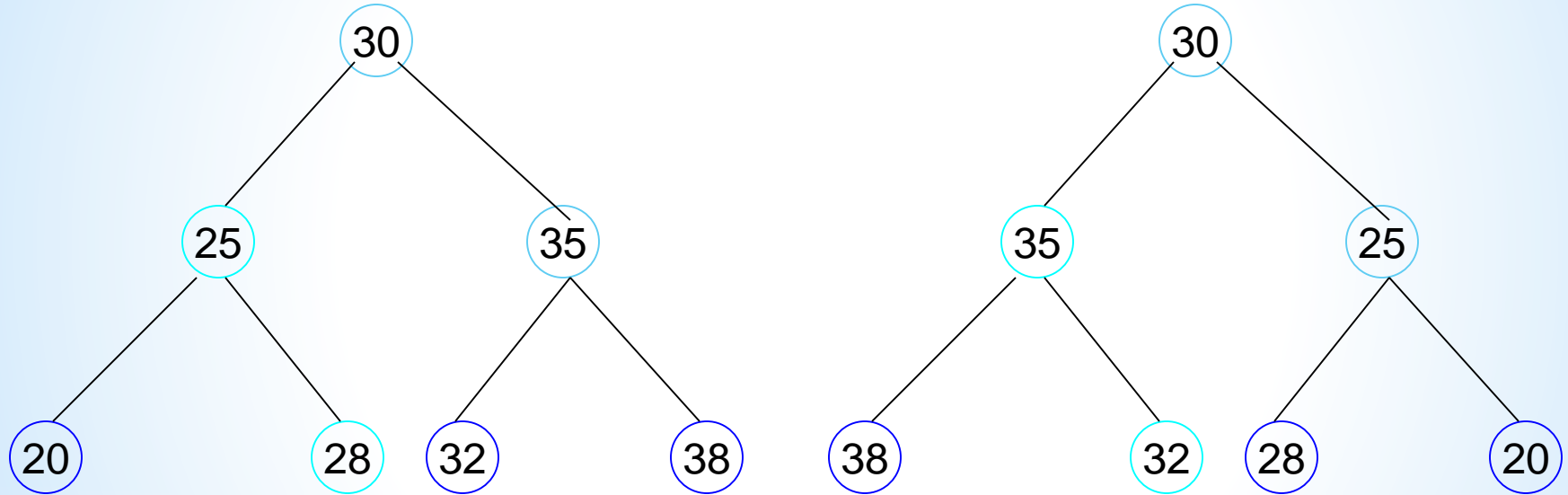
```
in identicalTrees(struct node* T1, struct node* T2) {  
    if (T1==NULL && T2==NULL) //nếu cả hai cây T1 và T2 đều rỗng  
        return 1; //rõ ràng chúng giống nhau  
    if (T1!=NULL && T2!=NULL){ //nếu cả hai cây khác rỗng  
        return (T1->data == T2->data &&  
            identicalTrees(T1->left, T2->left) &&  
            identicalTrees(T1->right, T2->right) );  
    }  
    return 0; //một trong hai cây khác rỗng  
}
```

4.4.3. Tìm độ cao của cây. Độ cao của cây được định nghĩa là đường đi dài nhất từ node gốc đến node lá. Cho một cây nhị phân bất kỳ, hãy xác độ cao của cây?

Lời giải. Thuật toán giải quyết bài toán được thực hiện như sau:

```
int maxDepth(struct node * T) {  
    if (T==NULL)  
        return 0;  
    else {  
        int lDepth = maxDepth(T->left); //Tìm độ cao của cây con trái  
        int rDepth = maxDepth(T->right); //Tìm độ cao của cây con phải  
        if (lDepth > rDepth)  
            return(lDepth);  
        else  
            return(rDepth);  
    }  
}
```

4.4.4. Cây phản chiếu (Mirror Tree). Cây phản chiếu của cây nhị phân T là cây nhị phân M(T), trong đó cây con bên trái của cây T trở thành cây con bên phải của M(T) và cây con bên phải của T trở thành cây con trái của M(T). Bài toán được đặt ra là hãy dịch chuyển cây nhị phân T cho trước thành cây nhị phân phản chiếu của T là cây M(T).



```
void mirror(struct node* node) {  
    if (node==NULL) return; //Nếu cây rỗng thì không phải làm gì  
    else { //Nếu cây không rỗng  
        struct node* temp; //Sử dụng node trung gian temp  
        mirror(node->left); //Gọi đến cây phản chiếu bên trái  
        mirror(node->right); //Gọi đến cây phản chiếu bên phải  
        temp = node->left; node->left = node->right;  
        node->right = temp; //Tráo đổi hai node trái và phải  
    }  
}
```

4.4.5. Tìm độ cao của cây. Độ cao của cây được định nghĩa là đường đi dài nhất từ node gốc đến node lá. Cho một cây nhị phân bất kỳ, hãy xác độ cao của cây?

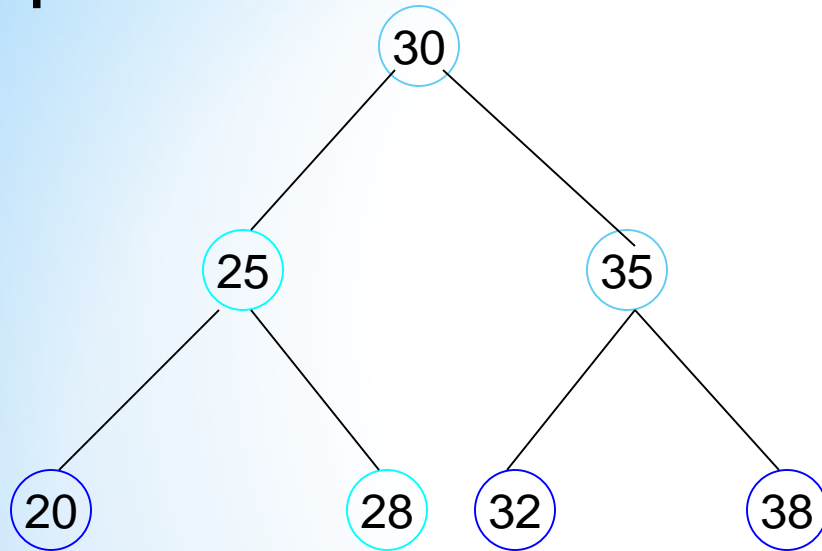
Lời giải. Thuật toán giải quyết bài toán được thực hiện như sau:

```
int maxDepth(struct node * T) {  
    if (T==NULL)  
        return 0;  
    else {  
        int lDepth = maxDepth(T->left); //Tìm độ cao của cây con trái  
        int rDepth = maxDepth(T->right); //Tìm độ cao của cây con phải  
        if (lDepth > rDepth)  
            return(lDepth);  
        else  
            return(rDepth);  
    }  
}
```

4.4.6. Tìm tất cả đường đi từ node gốc đến node lá

Lời giải. Thuật toán giải quyết bài toán được thực hiện như sau:

Input:



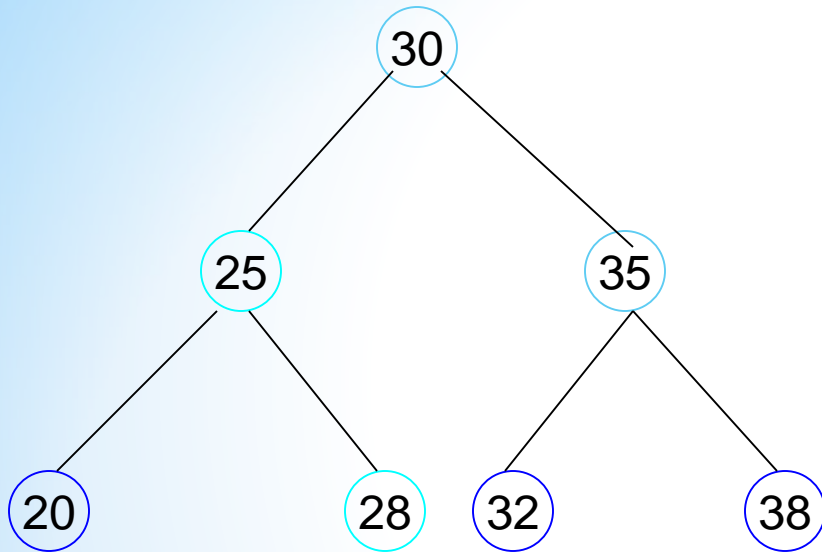
Output:

30	25	20
30	25	28
30	35	32
30	35	38

```
void printPathsRecur(struct node* T, int path[], int pathLen) {  
    if (T==NULL) return; //Nếu cây rỗng thì không phải làm gì  
    path[pathLen] = node->data; pathLen++; //Ghi nhận đường đi bắt đầu từ gốc  
    if (node->left==NULL && node->right==NULL) //Nếu node là lá  
        printArray(path, pathLen); //Đưa ra đường đi từ node gốc đến node lá  
    else {  
        printPathsRecur(node->left, path, pathLen); //Đi tiếp sang cây bên trái  
        printPathsRecur(node->right, path, pathLen); //Đi tiếp sang bên phải  
    }  
}
```

4.4.7. Đếm tất cả các node lá trên cây

Input:

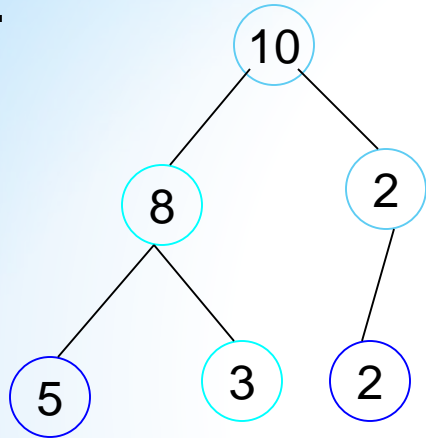


Output:

4

```
unsigned int getLeafCount(struct node* node) {  
    if(node == NULL) //Nếu cây rỗng  
        return 0; //Số node lá là 0  
    if(node->left == NULL && node->right==NULL) //Nếu cây có một node  
        return 1; //Số node lá là 1  
    else //Nếu cây có ít nhất một cây con  
        return getLeafCount(node->left)+ getLeafCount(node->right);  
}
```

4.4.8. Kiểm tra một cây thỏa mãn điều kiện node trung gian bằng tổng hai node con trái và phải hay không? Node không có node lá trái hoặc phải được xem có giá trị 0.



```
int isSumProperty(struct node* node) {  
    int left_data = 0, right_data = 0;  
    if(node == NULL || (node->left == NULL && node->right == NULL)) return 1;  
    else {  
        if(node->left != NULL)  
            left_data = node->left->data;  
        if(node->right != NULL)  
            right_data = node->right->data;  
        if((node->data == left_data + right_data)&& isSumProperty(node->left) &&  
            isSumProperty(node->right))  
            return 1;  
        else return 0;  
    }  
}
```

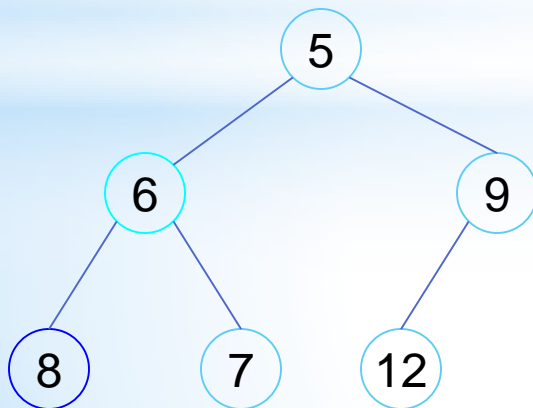

4.5. Cấu trúc Bainary Heap (Heap)

Định nghĩa. Cấu trúc dữ liệu Binary Heap là một cây nhị phân T cùng với các đặc tính như sau:

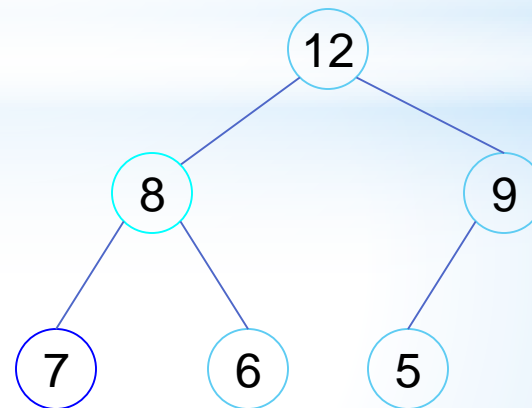
- 1) T là cây nhị phân gần đầy: Mọi node trung gian đều có đầy đủ hai node con ngoại trừ mức cuối cùng có các node đầy từ trái sang phải. Đặc tính này của Heap làm cho T thích hợp với việc sắp xếp các phần tử trong một mảng.
- 2) Một Binary Heap có thể là Max Heap hoặc Min Heap. Trong Min Heap, giá trị khóa của node gốc là node có giá trị nhỏ nhất trong các khóa được biểu diễn. Tất cả các node còn lại cũng được định nghĩa đệ qui như node gốc. Max Heap cũng được định nghĩa tương tự như Min Heap.

Ví dụ ta cần xây dựng Min Heap và Max Heap cho dãy $A[] = \{ 9, 7, 12, 8, 6, 5 \}$.

Min Heap



Max Heap



Biểu diễn Heap. Giống như biểu diễn cây nhị phân thông thường: Biểu diễn liên tục dựa vào mảng, biểu diễn rời rạc sử dụng danh sách liên kết.

Ví dụ về tạo max heap:

```
void max_heapify(int *A, int i, int n){
    int j, temp = A[i]; j = 2 * i;
    while (j <= n){
        if (j < n && A[j+1] > A[j]) j = j + 1;
        if (temp > A[j]) break;
        else if (temp <= A[j]){
            A[j/2] = A[j]; j = 2 * j;
        }
    }
    A[j/2] = temp;
}

void build_maxheap(int *A,int n){
    for(int i = n/2; i >= 1; i--){
        max_heapify(A,i,n);
    }
}
```

Ví dụ về tạo min heap:

```
void min_heapify(int *a,int i,int n){
    int j, temp;temp = a[i]; j = 2 * i;
    while (j <= n){
        if (j < n && a[j+1] < a[j])    j = j + 1;
        if (temp < a[j])    break;
        else if (temp >= a[j]){
            a[j/2] = a[j];    j = 2 * j;
        }
    }
    a[j/2] = temp;
}

void build_minheap(int *a, int n){
    for(int i = n/2; i >= 1; i--){
        min_heapify(a,i,n);
    }
}
```

Các thao tác trên Min Heap:

```
class MinHeap { //Định nghĩa lớp MinHeap
    int *harr; //Mảng các node trong heap
    int capacity; // Kích cỡ tối đa của Heap
    int heap_size; //Kích cỡ hiện tại của heap
public:
    MinHeap::MinHeap(int cap){// constructor của lớp MinHeap
        heap_size = 0; capacity = cap;
        harr = new int[cap];
    }
    void MinHeapify(int k);// Tạo heap con với khóa k
    int parent(int i) { return (i-1)/2; }//lấy vị trí node cha
    int left(int i) { return (2*i + 1); }//lấy vị trí node con trái
    int right(int i) { return (2*i + 2); }//lấy vị trí node con phải
    int extractMin();//loại bỏ phần tử nhỏ nhất hoặc gốc của heap
    void decreaseKey(int i, int new_val);//sử dụng thêm hoặc loại bỏ
    int getMin() { return harr[0]; }//nhận giá trị nhỏ nhất roof của heap
    void deleteKey(int i);//Loại bỏ vị trí i nhưng vẫn là một heap
    void insertKey(int k);//thêm k vào heap để nhận được một heap mới
};
```

Thao tác thêm mới khóa k vào heap:

```
void MinHeap::insertKey(int k) {  
    if (heap_size == capacity) { //Nếu heap đã đầy  
        cout << "\n Không thêm được node\n";  
        return;  
    }  
    // Trước tiên thêm node k vào cuối cùng  
    heap_size++; int i = heap_size - 1; harr[i] = k;  
    // Xác định lại vị trí của k trong min heap  
    while (i != 0 && harr[parent(i)] > harr[i]) {  
        swap(&harr[i], &harr[parent(i)]);  
        i = parent(i);  
    }  
}
```

Thao tác giảm giá trị khóa tại vị trí i:

```
void MinHeap::decreaseKey(int i, int new_val) {  
    harr[i] = new_val;  
    while (i != 0 && harr[parent(i)] > harr[i]) {  
        swap(&harr[i], &harr[parent(i)]);  
        i = parent(i);  
    }  
}
```

Thao tác loại bỏ phần tử nhỏ nhất từ heap hoặc gốc:

```
int MinHeap::extractMin(){
    if (heap_size <= 0) //nếu heap rỗng
        return INT_MAX;
    if (heap_size == 1) {//Nếu heap chỉ có một node
        heap_size--; return harr[0]; //Gốc của heap bị loại bỏ
    }
    // Lưu trữ giá trị nhỏ nhất và loại bỏ khỏi heap
    int root = harr[0]; harr[0] = harr[heap_size-1];
    heap_size--; MinHeapify(0);
    return root;
}
```

Thao tác vun đống:

```
void MinHeap::MinHeapify(int i) {
    int l = left(i); int r = right(i); int smallest = i;
    if (l < heap_size && harr[l] < harr[i]) smallest = l;
    if (r < heap_size && harr[r] < harr[smallest]) smallest = r;
    if (smallest != i) { swap(&harr[i], &harr[smallest]); MinHeapify(smallest); }
}
```

Thao tác loại bỏ node trên heap:

```
void MinHeap::deleteKey(int i) {
    decreaseKey(i, INT_MIN); //Trước hết giảm giá trị
    extractMin(); //Sau đó loại bỏ
}
```


Ứng dụng của Heap:

- Heap dùng vào các kỹ thuật sắp xếp với độ phức tạp $O(n\log(n))$.
- Xây dựng hàng đợi ưu tiên: sử dụng các thao tác trên heap để xây dựng hàng đợi ưu tiên với độ phức tạp $\log(n)$.
- Sử dụng trong cài đặt hiệu quả các thuật toán trên đồ thị.
- Giải quyết nhiều bài toán quan trọng khác.

Bài tập về heap

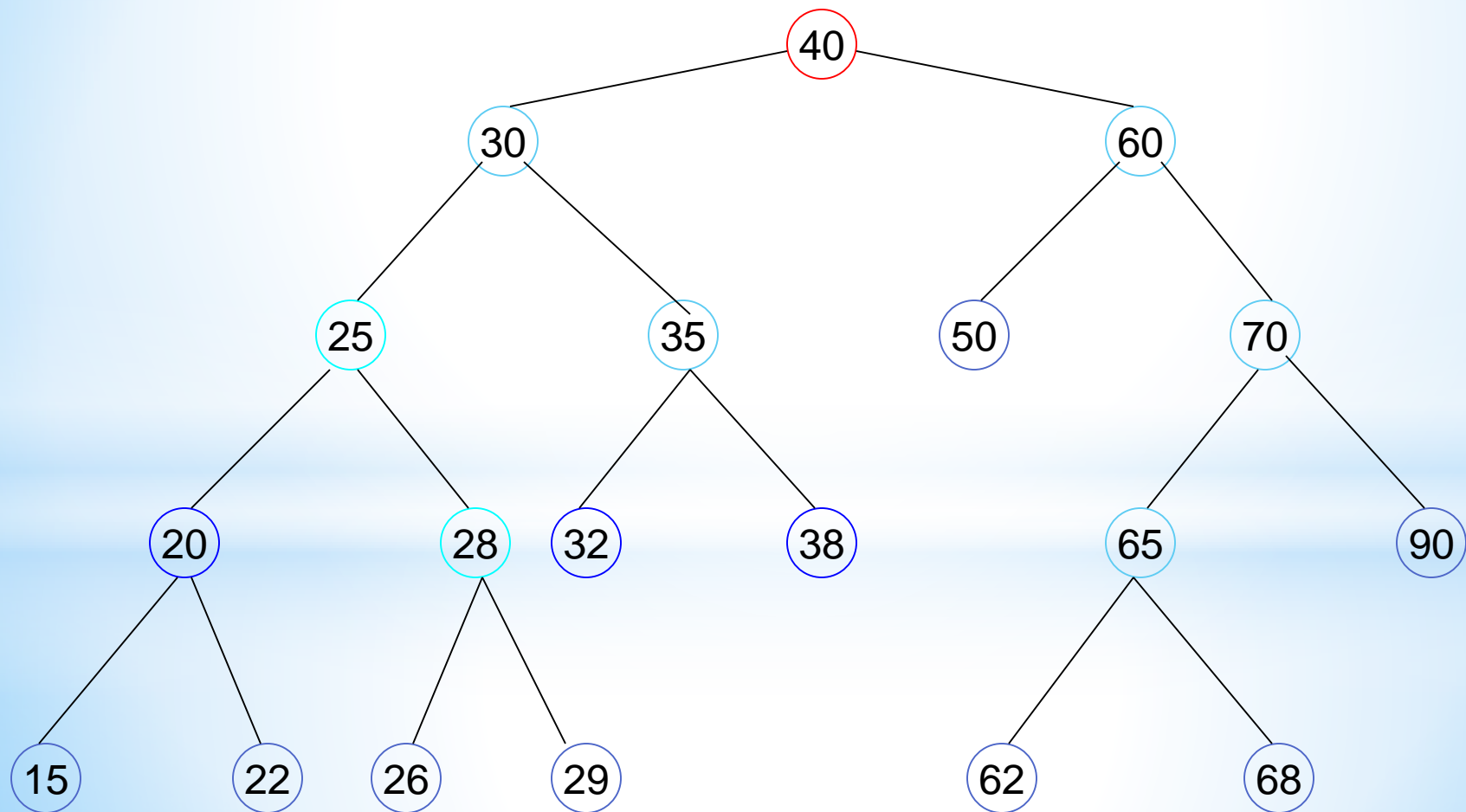
- Cài đặt các thao tác trên Max Heap.
- Cài đặt các thao tác trên Binary Heap.
- Cài đặt các thao tác trên Weak Heap.
- Cài đặt các thao tác trên Binomial Tree.
- Cài đặt các thao tác trên Binomial Heap.
- Cài đặt các thao tác trên Fibonacci Heap.
- Cài đặt các thao tác trên Left List Heap.
- Cài đặt các thao tác trên Skew Heap.
- Cài đặt các thao tác trên Ternary Heap.
- Cài đặt các thao tác trên D-ary Heap.
- Cài đặt các thao tác trên Pairing Heap.
- Cài đặt các thao tác trên Treap Heap.

4.6. Cây nhị phân tìm kiếm

4.6.1. Định nghĩa

Cây nhị phân thỏa mãn điều kiện:

- Hoặc là rỗng hoặc có một node gốc. Mỗi node gốc có tối đa hai cây con. Nội dung node gốc lớn hơn nội dung node con bên trái và nhỏ hơn nội dung node con bên phải.
- Hai cây con bên trái và bên phải cũng hình thành nên hai cây tìm kiếm.



4.6.2. Biểu diễn

Giống như cây nhị phân thông thường.

```
typedef struct node {  
    Item infor; //Thông tin của node  
    struct node *left; //Con trỏ sang cây con trái  
    struct node *right; //Con trỏ sang cây con phải  
} *Tree;
```

4.6.3. Thao tác

- Tạo node gốc cho cây.
- Thêm vào node vào cây tìm kiếm.
- Loại bỏ node trên cây tìm kiếm.
- Tìm kiếm node trên cây.
- Xoay trái cây tìm kiếm
- Xoay phải cây tìm kiếm
- Duyệt cây theo thứ tự trước.
- Duyệt cây theo thứ tự giữa.
- Duyệt cây theo thứ tự sau.

Biểu diễn cây nhị phân tìm kiếm:

```
typedef struct node {  
    Item Infor; //Thông tin của node  
    struct nde *left; //Con trỏ node bên trái  
    struct nde *right; //Con trỏ node bên phải  
} *Tree;
```

Khởi tạo cây nhị phân tìm kiếm:

```
void Init(Tree *T){  
    *T=NULL; //Đưa cây T về trạng thái rỗng  
}
```

Cấp phát miền nhớ cho một node:

```
Tree GetNode(){  
    Tree p; //Khai báo một node kiểu Tree  
    p=new node; //Cấp phát miền nhớ cho node  
    return (p); //Trả lại node được cấp phát  
}
```

Giải phóng một node p cho cây T:

```
void FreeNode(Tree p){  
    delete (p); //giải phóng node p  
}
```

Kiểm tra tính rỗng của cây T:

```
int isEmpty(Tree *T){  
    if(*T==NULL) //nếu T rỗng  
        return 1; //trả lại giá trị đúng  
    return 0; //trả lại giá trị sai  
}
```

Tạo một node cho cây T:

```
Tree MakeNode( Item x){ //x là giá trị node cần bổ sung vào cây  
    Tree p; //Khai báo một node kiểu Tree  
    p = GetNode(); //cấp phát miền nhớ cho p  
    p->infor = x; //thiết lập thành phần thông tin cho node  
    p->left=NULL; //tạo liên kết trái cho node  
    p->right=NULL; //Tạo liên kết phải cho node  
    return (p); //trả lại node được tạo ra  
}
```

Tìm node có nội dung là x trên cây tìm kiếm T:

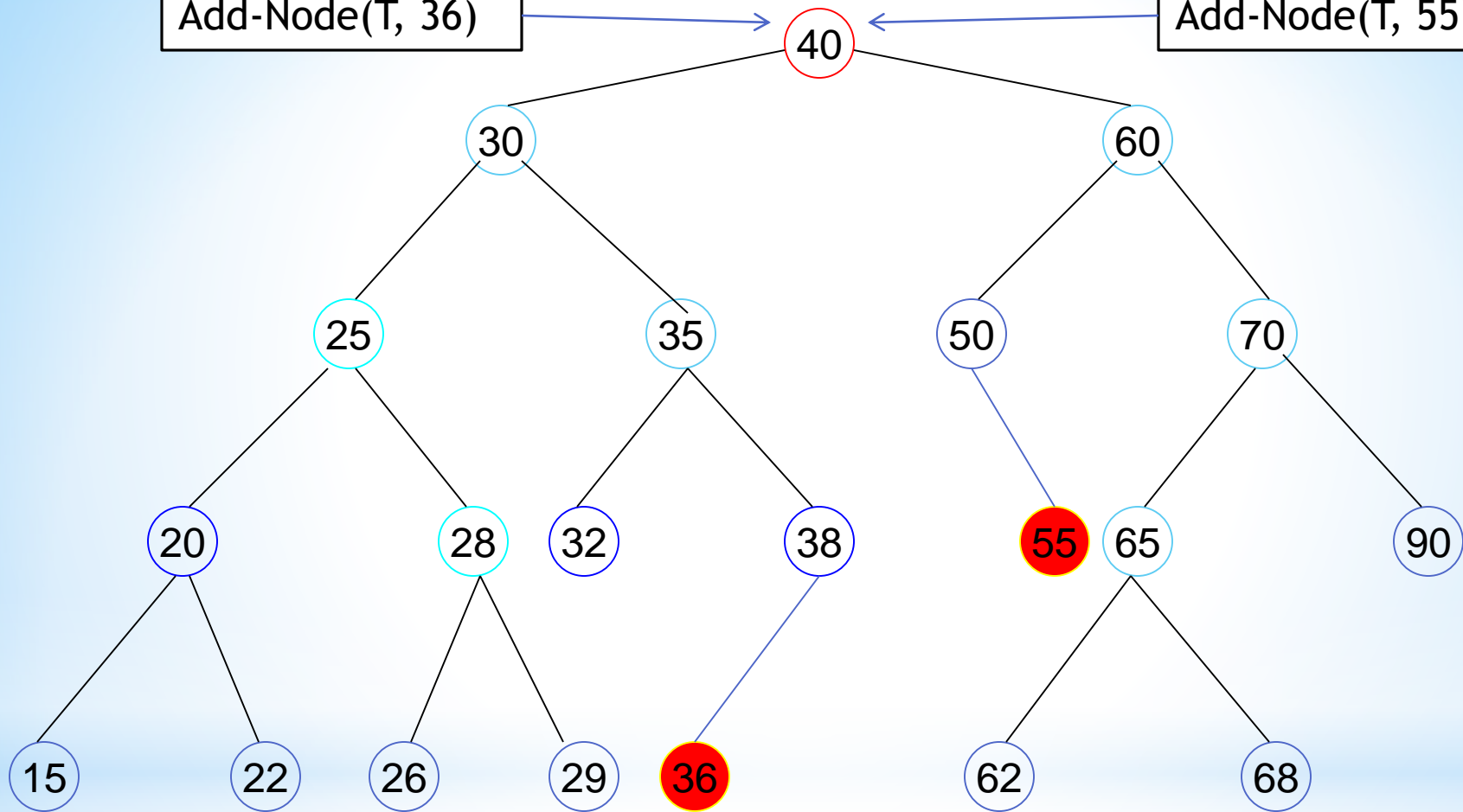
```
void Search(Tree T, Item x){
    Tree p =T; //p trở đến node gốc của cây
    if ( p!=NULL ) { //nếu p không phải là NULL
        if (x < T->infor ) p = Search( T->left, x);
        else p = Search(T->right, x);
    }
    return(p);
}
```

Tìm node có nội dung là x trên cây tìm kiếm T (không đệ qui):

```
Tree Search(Tree T, Item x){ Tree p =T; //p trở đến node gốc của cây
    while (p!=NULL) { //Lặp trong khi p khác NULL
        if (x == p->infor) return (p); //Nếu x trùng với p
        else if (x < p->infor) p = p-> left; //nếu x nhỏ hơn p
        else p = p-> right; // nếu x lớn hơn p
    }
    return(p);
}
```

Add-Node(T, 36)

Add-Node(T, 55)



Thêm node vào cây tìm kiếm

Thêm node vào cây tìm kiếm T:

```
void Add-Node(Tree T, Item x){
    Tree p, q; //p, q là hai biến kiểu Tree
    p = q = T; //p, q trở đến T
    while (p ->infor!=x&&q!=NULL) { //Lặp nếu p->infor!=x và q khác NULL
        p = q; //p là node cha của q
        if (x< p->infor) //nếu x nhỏ hơn nội dung của p
            q = p ->left; // q trở sang cây con trái
        else //nếu x lớn hơn nội dung của p
            q = p -> right; // q trở sang cây con phải
    }
    if ( x == p->infor) { <Trùng node>; return;}
    Tree r = MakeNode(T,x); // r là node cần thêm vào cây
    if ( x < p -> infor) //Nếu x chỉ có thể là node lá bên trái p
        p - > left = r;
    else //Nếu x chỉ có thể là node lá bên phải p
        p -> right =r;
}
```


Loại bỏ node trên cây tìm kiếm:

```
Tree Remove (Tree p ) {
```

```
    Tree rp, f; // rp là node thay thế cho p, f là node cha của rp
```

```
    if ( p==NULL) { <Node không có thực>; return(p); }
```

```
    else if (p -> right == NULL && p->left!=NULL) // nếu p chỉ có cây con trái
```

```
        rp = p -> left; //Node thay thế p là p ->left
```

```
    else if (p -> right != NULL && p->left==NULL) // nếu p chỉ có cây con phải
```

```
        rp = p -> right; // Node thay thế p là p ->right
```

```
    else { //nếu p có cả hai cây con
```

```
        f = p ; rp = p -> right; //f là node cha của rp;
```

```
        while ( rp ->left!=NULL ) { // quay trái cây con bên phải
```

```
            f = rp; rp = rp -> left;
```

```
        }
```

```
        if (f !=p) { f ->left = rp ->right; rp ->right = p->right; rp ->left = p ->left; }
```

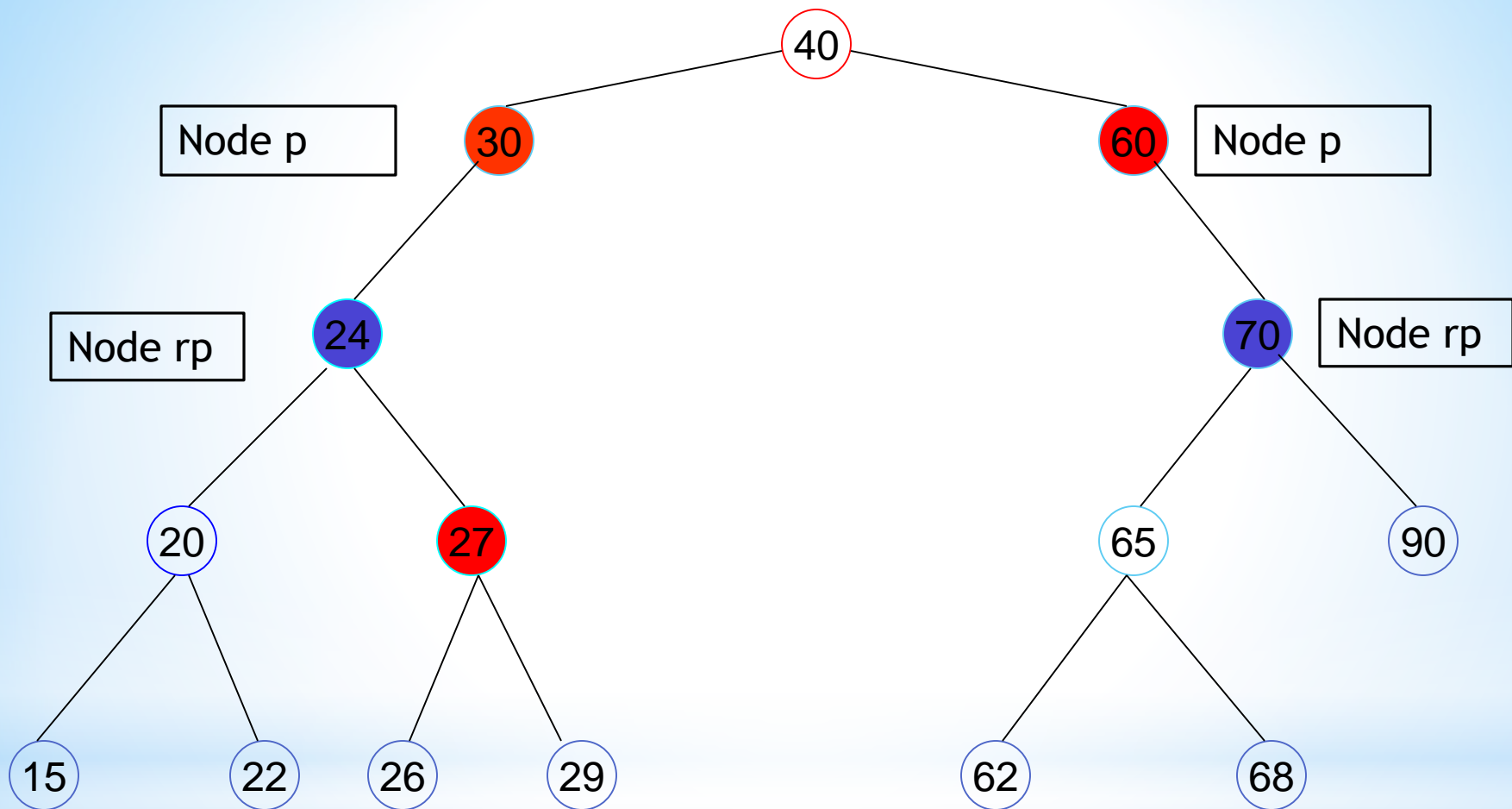
```
        else rp -> left = p ->left;
```

```
    }
```

```
    FreeNode(p);
```

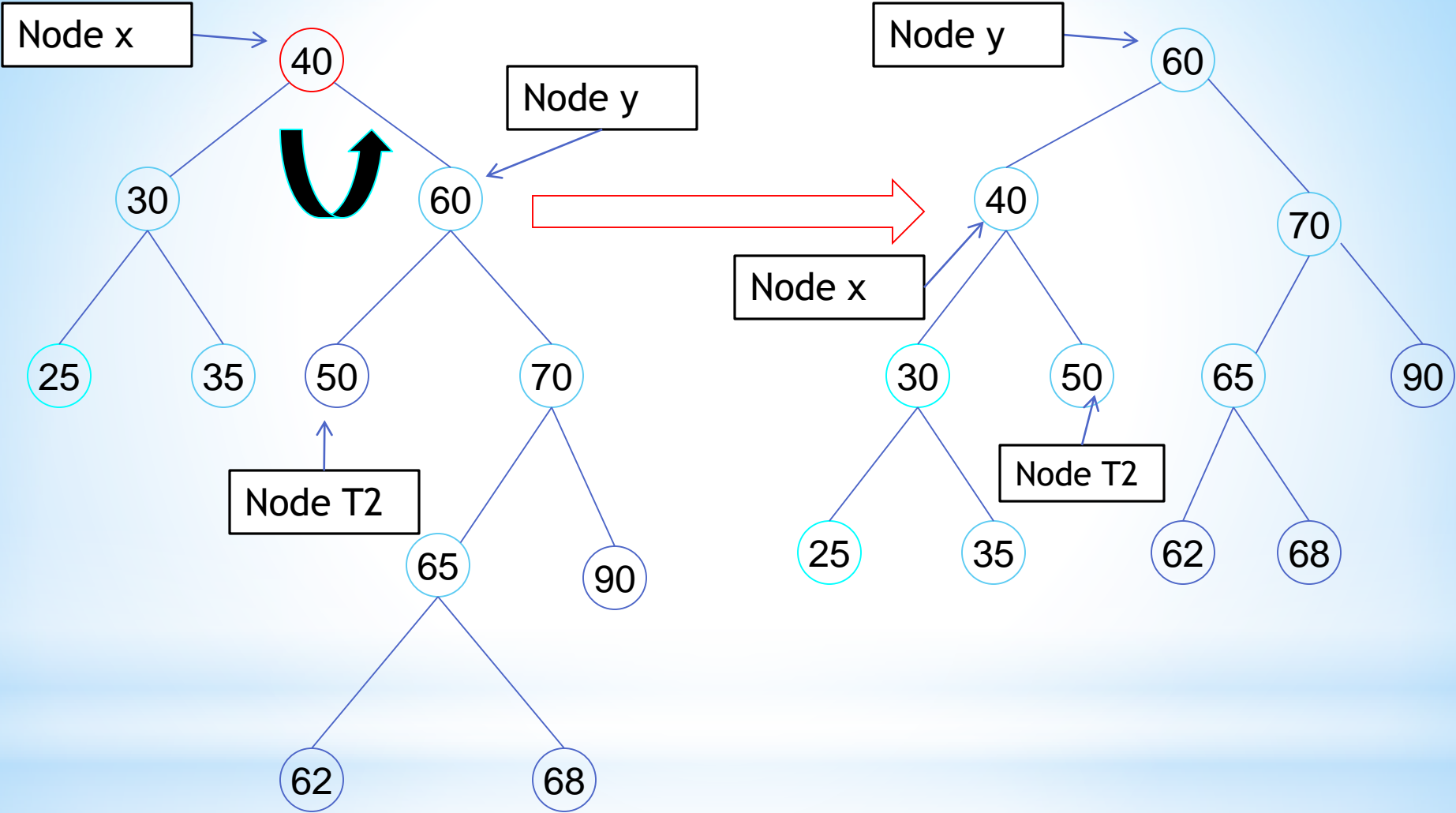
```
    Return(rp);
```

```
}
```



Loại node trên cây tìm kiếm

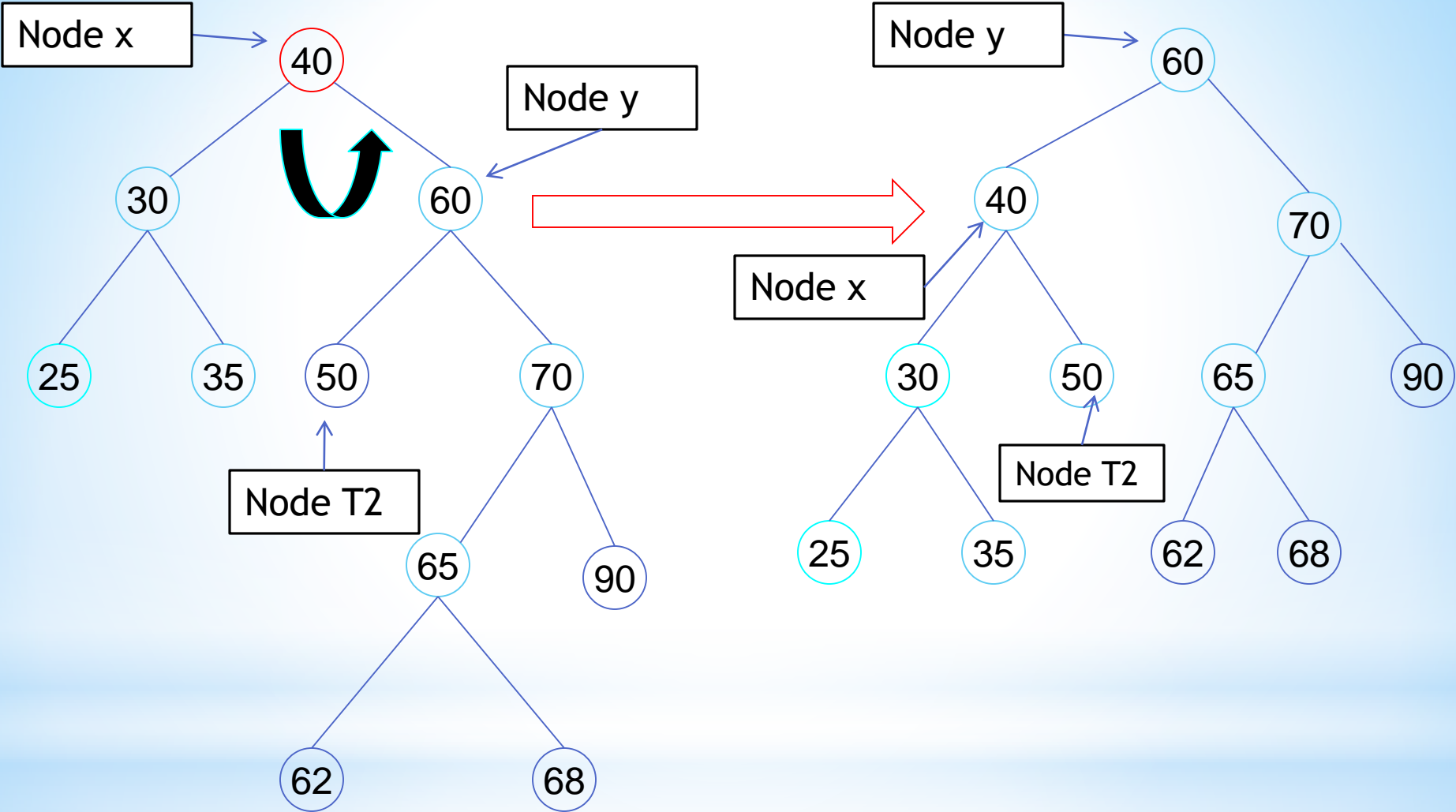
Xoay trái cây nhị phân tìm kiếm:



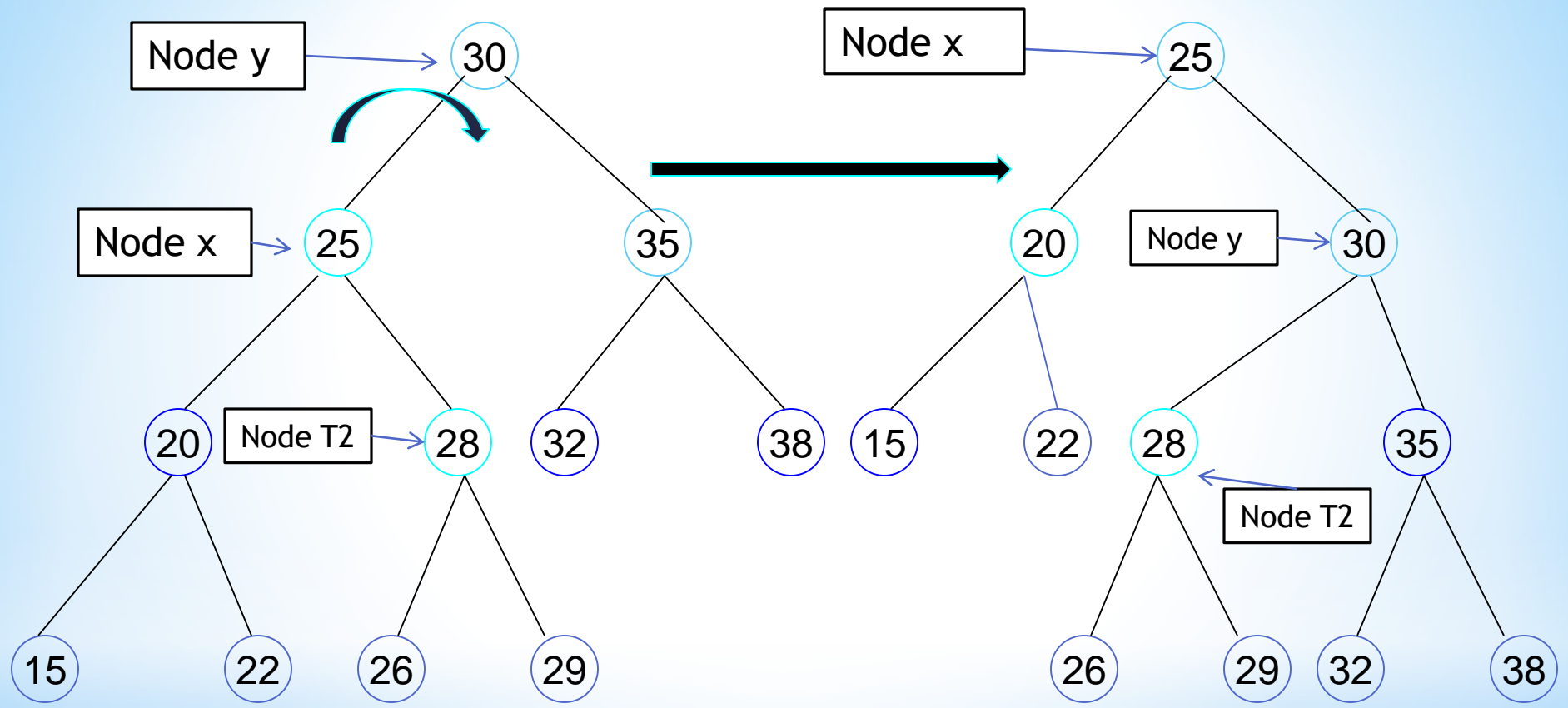
Xoay trái cây nhị phân tìm kiếm:

```
Tree RotateLeft (Tree T ) {  
    Tree p = T; // p trở đến node gốc của cây  
    if ( T==NULL)  
        <Cây rỗng>;  
    else if (T -> right == NULL)  
        <T không có cây con phải>;  
    else {  
        p = T -> right; //p trở đến cây con phải  
        T -> right = p ->left; //T trở đến node trái của p  
        p -> left = T; //Thiết lập liên kết trái cho p  
    }  
    return(p);  
}
```

Xoay trái cây nhị phân tìm kiếm:



Xoay phải cây nhị phân tìm kiếm:



Xoay phải cây nhị phân tìm kiếm:

```
Tree RotateRight (Tree T ) {  
    Tree p = T; // p trở đến node gốc của cây  
    if ( T==NULL)  
        <Cây rỗng>;  
    else if (T -> Left == NULL)  
        <T không có cây con trái>;  
    else {  
        p = T -> left; //p trở đến cây con trái  
        T -> left = p ->right; //T trở đến node phải của p  
        p -> right = T; //Thiết lập liên kết phải cho p  
    }  
    return(p);  
}
```


Ví dụ. *Bài toán tách từ.* Cho file dữ liệu văn bản data.in. Hãy tìm tập các từ và số lần xuất hiện mỗi từ trong file?

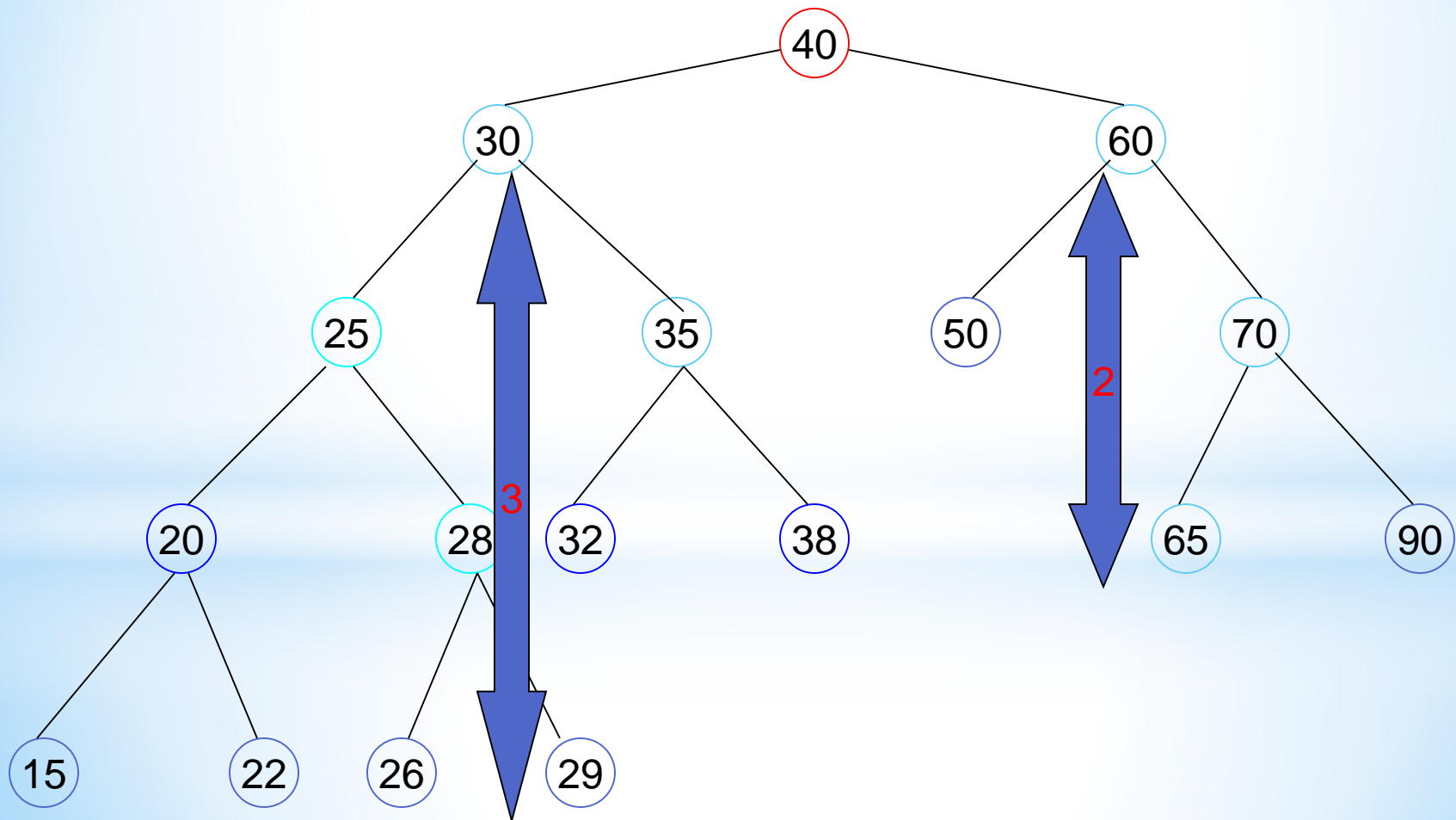
data.in					ketqua.out	
AB	AC	AD	AE	AF	5	
AB	AC	AD	AE	AF	AB	2
					AC	2
					AD	2
					AE	2
					AF	2

Lời giải.

- *Sử dụng mảng:* không xử lý được các file dữ liệu văn bản có kích cỡ lớn.
- *Sử dụng danh sách liên kết:* xử lý được các file dữ liệu văn bản có kích cỡ lớn, nhưng chi phí thời gian cập nhật từ mới hoặc tìm số lần xuất hiện của từ lớn.
- *Sử dụng cây nhị phân:* xử lý được các file dữ liệu văn bản có kích cỡ lớn, nhưng chi phí thời gian cập nhật từ mới hoặc tìm số lần xuất hiện của từ lớn.
- *Sử dụng cây nhị phân tìm kiếm:* xử lý được các file dữ liệu văn bản có kích cỡ lớn, chi phí thời gian cập nhật từ mới hoặc tìm số lần xuất hiện của từ là $\log(n)$.

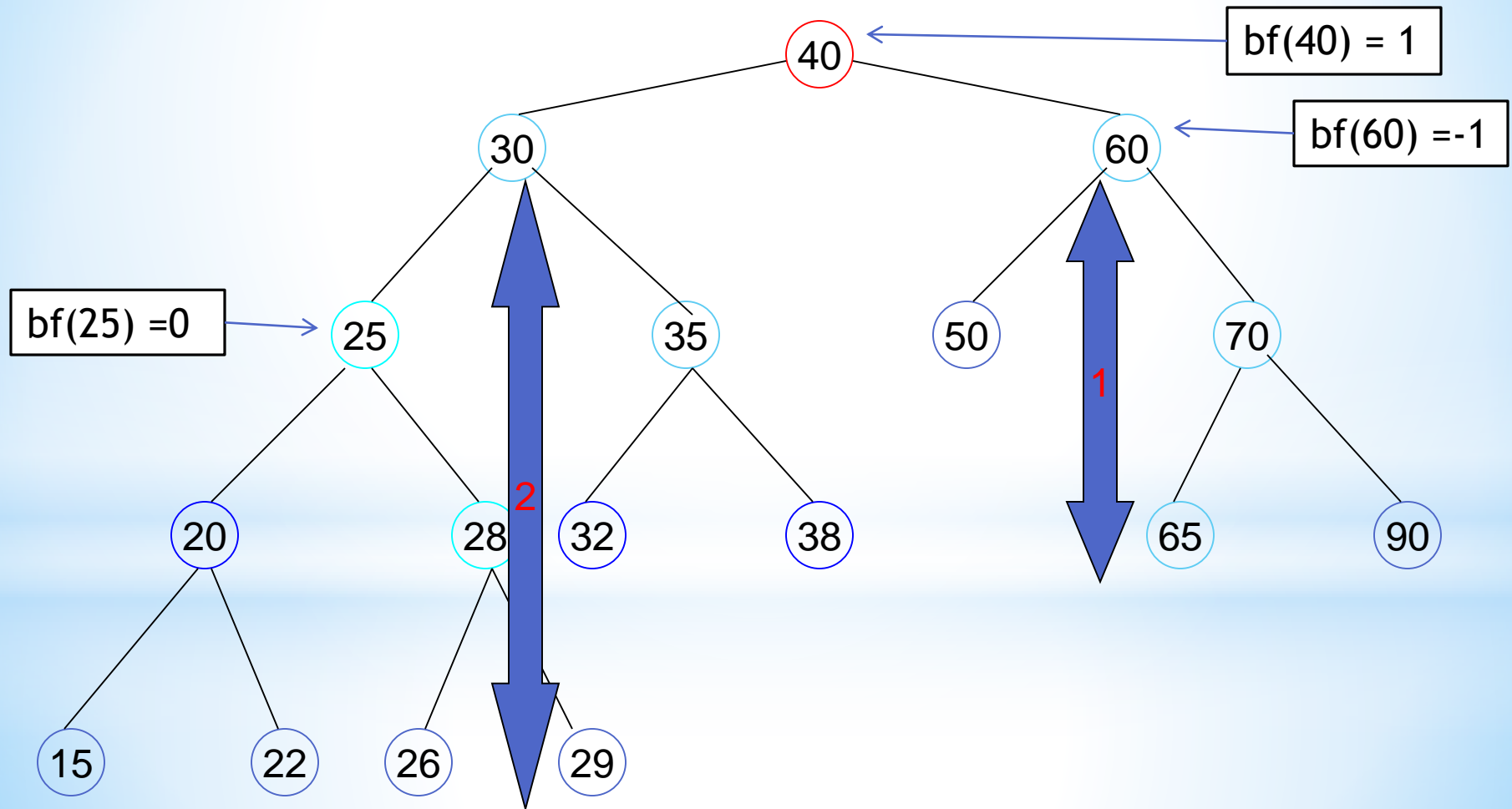
4.7. Cây nhị phân tìm kiếm cân bằng

- **Định nghĩa.** Là cây nhị phân tìm kiếm tự cân bằng. Cây tìm kiếm cân bằng có tính chất độ cao của cây con bên trái và độ cao cây con bên phải chênh lệch nhau không quá 1.
- **Cây AVL:** Cho phép ta tìm kiếm, bổ sung, loại bỏ node nhanh hơn cây nhị phân thông thường vì độ cao của cây luôn là $O(\log(n))$.



Cây nhị phân tìm kiếm cân bằng (AVL): gọi $lh(p)$, $rh(p)$ là chiều sâu của cây con phải và chiều sâu của cây con trái. Khi đó, $bf(p) = lh(p) - rh(p)$ là chỉ số cân bằng của node p và cây tìm kiếm cân bằng xảy ra trong các trường hợp sau:

1. $lh(p) = rh(p) \Leftrightarrow bf(p) = 0$: node p cân bằng tuyệt đối.
2. $lh(p) = rh(p) + 1 \Leftrightarrow bf(p) = 1$: node p lệch về phía trái.
3. $lh(p) = rh(p) - 1 \Leftrightarrow bf(p) = -1$: node p lệch về phía phải.



Biểu diễn cây tìm kiếm cân bằng: Giống như biểu diễn cây thông thường

```
struct avl_node { //định nghĩa node của cây
    int data; //thành phần thông tin
    struct avl_node *left; //thành phần con trỏ sang cây con trái
    struct avl_node *right; //thành phần con trỏ sang cây con phải
} *root; //đây là gốc của cây avl
```

Các thao tác trên cây tìm kiếm cân bằng:

```
class avlTree {
    public:
        int height(avl_node *); //trả lại độ cao của cây avl
        int diff(avl_node *); //trả lại chỉ số cân bằng của cây avl
        avl_node *rr_rotation(avl_node *); //thao tác quay phải –phải
        avl_node *ll_rotation(avl_node *); //thao tác quay trái-trái
        avl_node *lr_rotation(avl_node *); //thao tác quay trái –phải
        avl_node *rl_rotation(avl_node *); //thao tác quay phải –trái
        avl_node* balance(avl_node *); //cân bằng lại cây
        avl_node* insert(avl_node *, int ); //thêm node vào cây avl
        void display(avl_node *, int); //thao tác quay phải –phải
        void inorder(avl_node *);
        void preorder(avl_node *);
        void postorder(avl_node *);
        avlTree() { root = NULL; }
};
```

4.7.1. Thêm node vào cây AVL

Hai bài toán cơ bản trên cây AVL được quan tâm nhiều hơn cả là thêm node vào cây tìm kiếm để được một cây tìm kiếm cân bằng. Loại node trên cây tìm kiếm cũng nhận được một cây tìm kiếm cân bằng. Phương pháp thêm node được tiến hành như sau:

Thuật toán thêm node w vào cây AVL:

Bước 1. Thực hiện thêm node w vào cây tìm kiếm giống như cây thông thường.

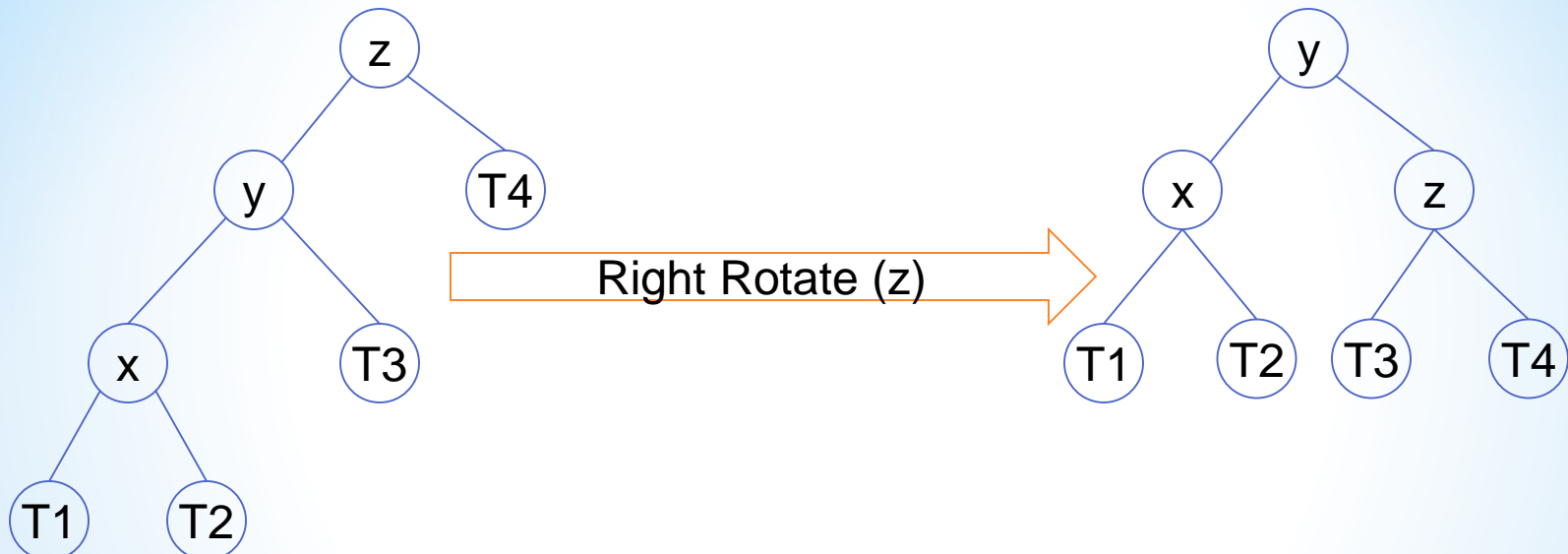
Bước 2. Xuất phát từ node w , duyệt lên trên để tìm node mất cân bằng đầu tiên. Gọi z là node mất cân bằng đầu tiên, y là con của z và x là cháu của z tính từ đường đi từ w đến z .

Bước 3. Cân bằng lại cây bằng các phép quay thích hợp tại cây con gốc z . Có 4 khả năng có thể xảy ra như sau:

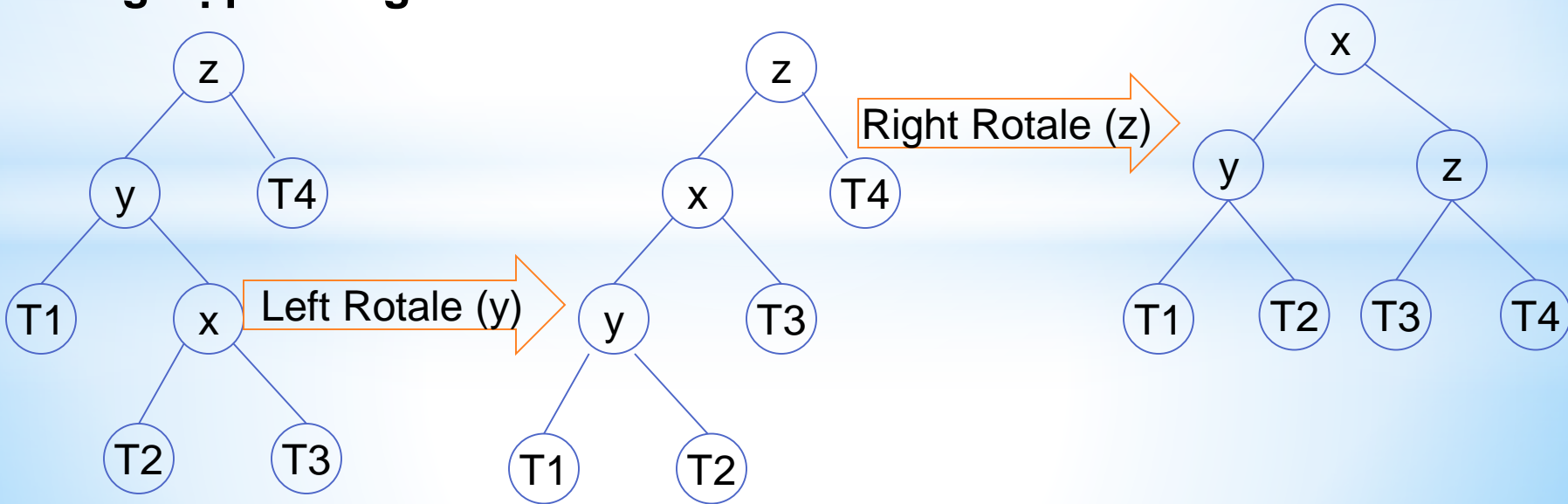
- Node y là node con trái của z và x là node con trái của y (left-left-case). Trường hợp này ta thực hiện phép soay phải (right rotation).
- Node y là node con trái của z và x là node con phải của y (left-right-case). Trường hợp này ta thực hiện phép soay trái sau đó soay phải.
- Node y là node con phải của z và x là node con phải của y (right-right-case). Trường hợp này ta thực hiện phép soay trái (left rotation).
- Node y là node con phải của z và x là node con trái của y (right-left-case). Trường hợp này ta thực hiện phép soay trái sau đó soay phải.

Giải sử T1, T2, T3, T4 là các cây con gốc z, khi đó các phép cân bằng lại (re-balance) được mô tả như sau:

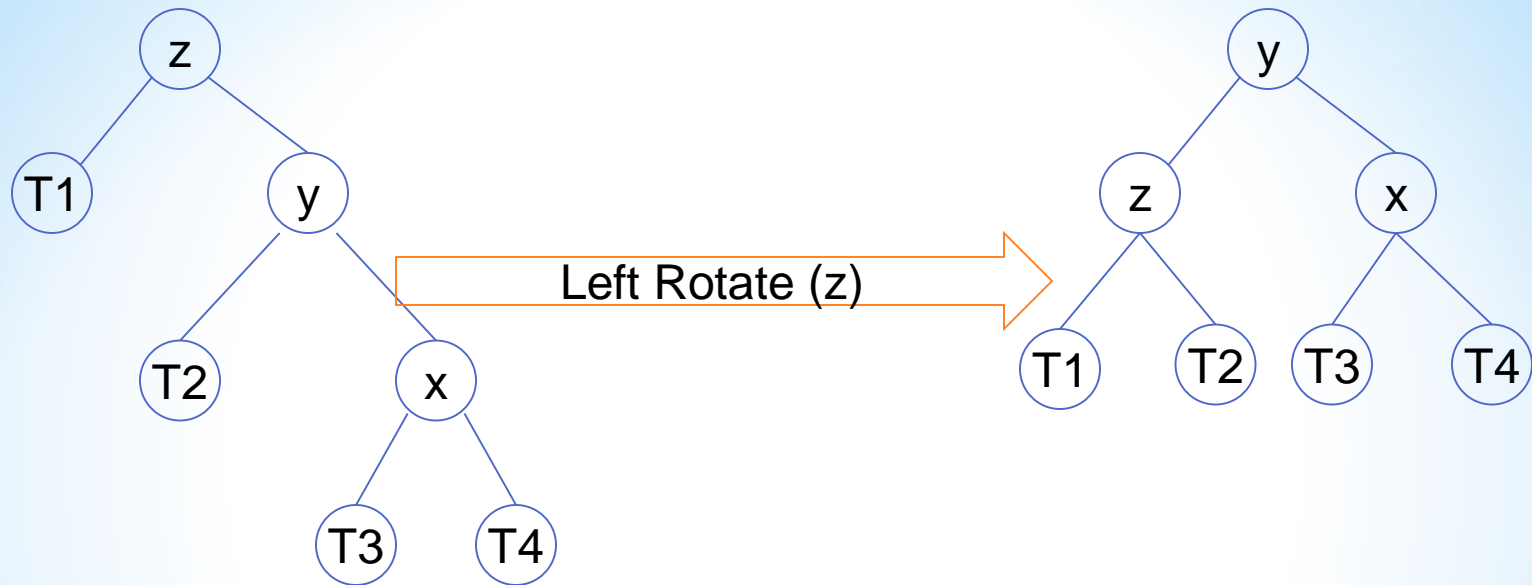
a) Trường hợp left-left-case:



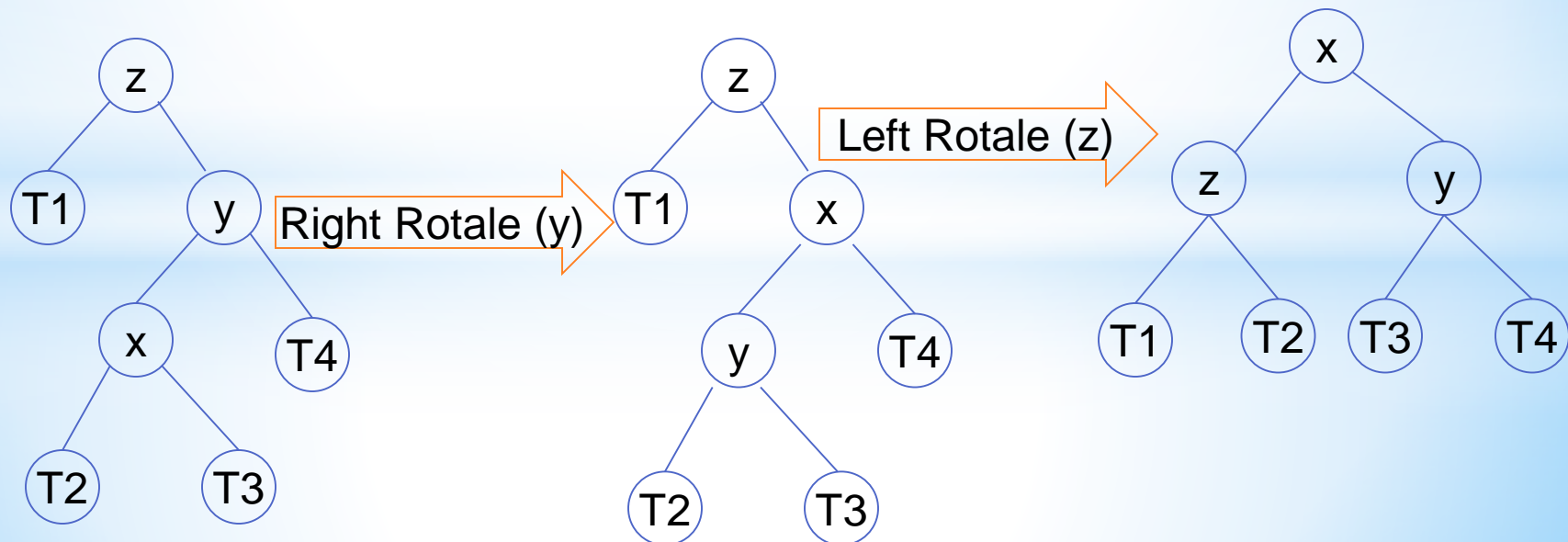
b) Trường hợp left-right-case:



c) Trường hợp right-right-case:



d) Trường hợp right-left-case:



4.7.2. Loại node trên cây AVL

Để chắc chắn sau khi loại node trên cây tìm kiếm cũng nhận được một cây tìm kiếm cân bằng ta tiến hành như sau:

Thuật toán loại node w trên cây AVL:

Bước 1. Thực hiện loại node w vào cây tìm kiếm giống như cây tìm kiếm thông thường.

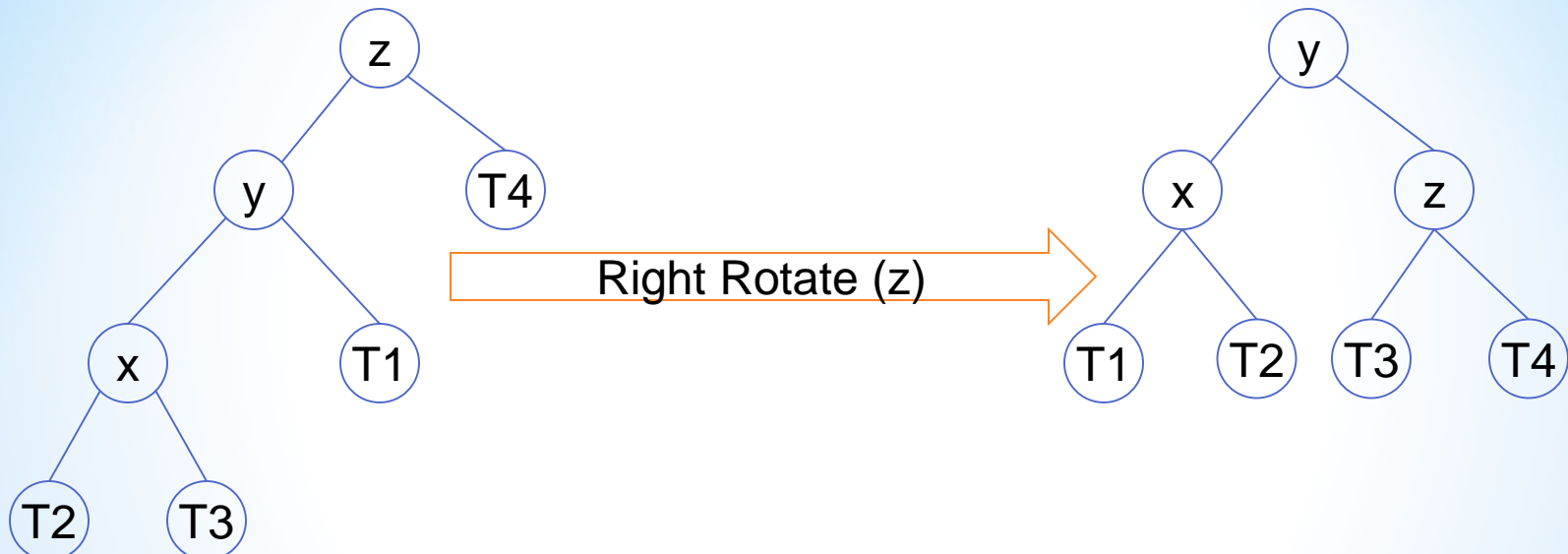
Bước 2. Xuất phát từ node w, duyệt lên trên để tìm node mất cân bằng đầu tiên. Gọi z là node mất cân bằng đầu tiên, y node cao hơn của z và x là cao hơn y.

Bước 3. Cân bằng lại cây bằng các phép quay thích hợp tại cây con gốc z. Có 4 khả năng có thể xảy ra như sau:

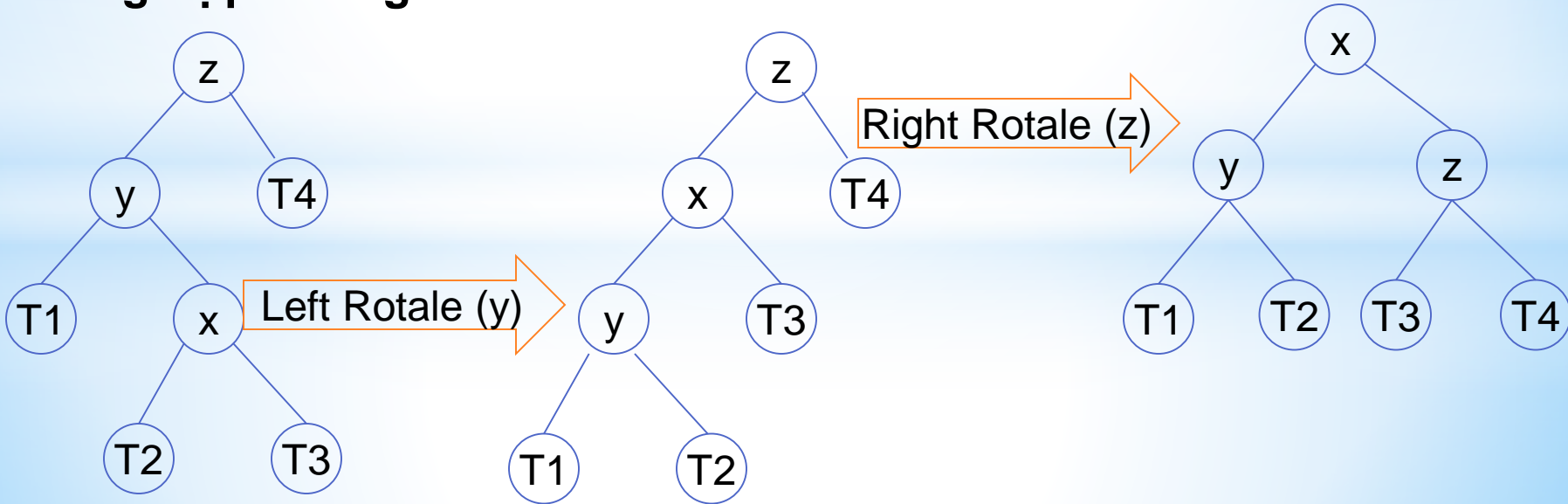
- a) Node y là node con trái của z và x là node con trái của y (left-left-case). Trường hợp này ta thực hiện phép soay phải (right rotation).
- b) Node y là node con trái của z và x là node con phải của y (left-right-case). Trường hợp này ta thực hiện phép soay trái sau đó soay phải.
- c) Node y là node con phải của z và x là node con phải của y (right-right-case). Trường hợp này ta thực hiện phép soay trái(left rotation).
- d) Node y là node con phải của z và x là node con trái của y (right-left-case). Trường hợp này ta thực hiện phép soay trái sau đó soay phải.

Giải sử T1, T2, T3, T4 là các cây con gốc z, khi đó các phép cân bằng lại (re-balance) được mô tả như sau:

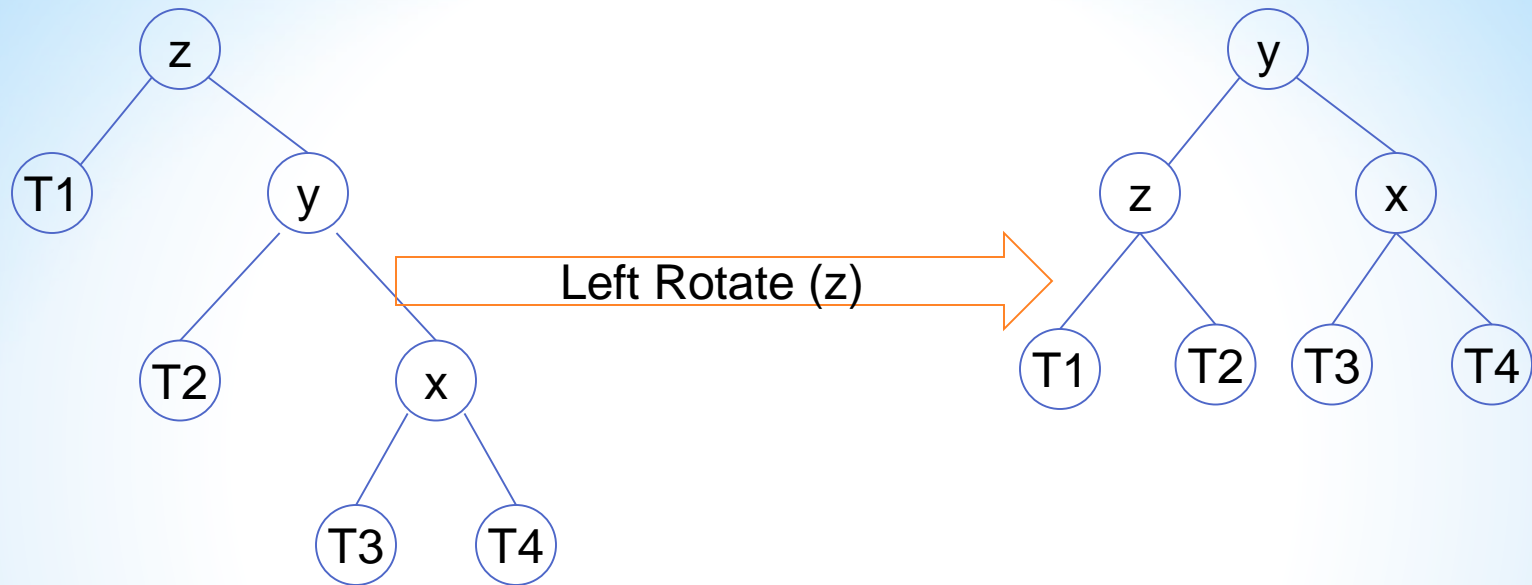
a) Trường hợp left-left-case:



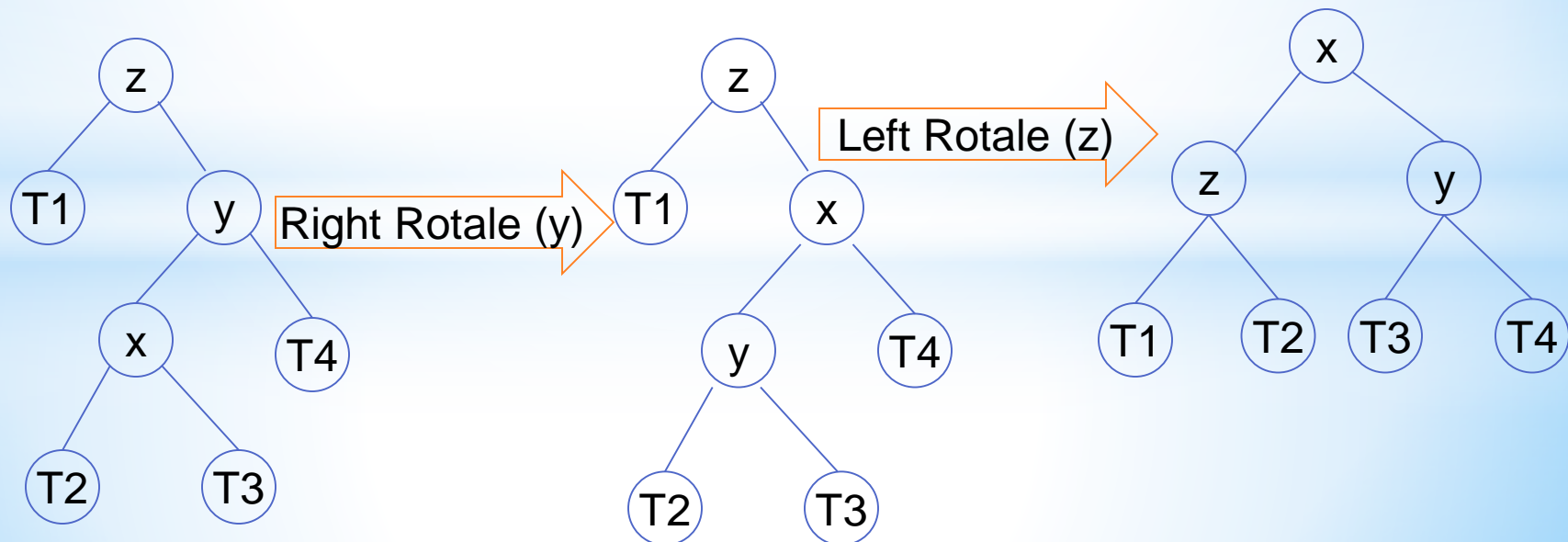
b) Trường hợp left-right-case:



c) Trường hợp right-right-case:



d) Trường hợp right-left-case:



4.7.3. Một số thao tác trên cây AVL

Khai báo node cây AVL:

```
struct node {  
    int key; //thành phần dữ liệu  
    struct node *left; //thành con trỏ đến cây con trái  
    struct node *right; //thành con trỏ đến cây con phải  
    int height; //chỉ số cân bằng của cây  
};
```

Phép quay phải (rightRotation):

```
struct node *rightRotate(struct node *y) {  
    struct node *x = y->left; //x trỏ đến node bên trái của y  
    struct node *T2 = x->right; //T2 là cây con phải của x->right  
    //Thực hiện quay phải tại y  
    x->right = y; y->left = T2;  
    //Cập nhật lại chiều cao  
    y->height = max(height(y->left), height(y->right))+1;  
    x->height = max(height(x->left), height(x->right))+1;  
    return x;  
}
```

Phép quay trái (leftRotation):

```
struct node *leftRotate(struct node *x) {  
    struct node *y = x->right;  
    struct node *T2 = y->left;  
    // thực hiện quay trái  
    y->left = x;  
    x->right = T2;  
    // cập nhật độ cao  
    x->height = max(height(x->left), height(x->right))+1;  
    y->height = max(height(y->left), height(y->right))+1;  
    // trả lại gốc mới root  
    return y;  
}
```

Thêm node vào cây AVL:

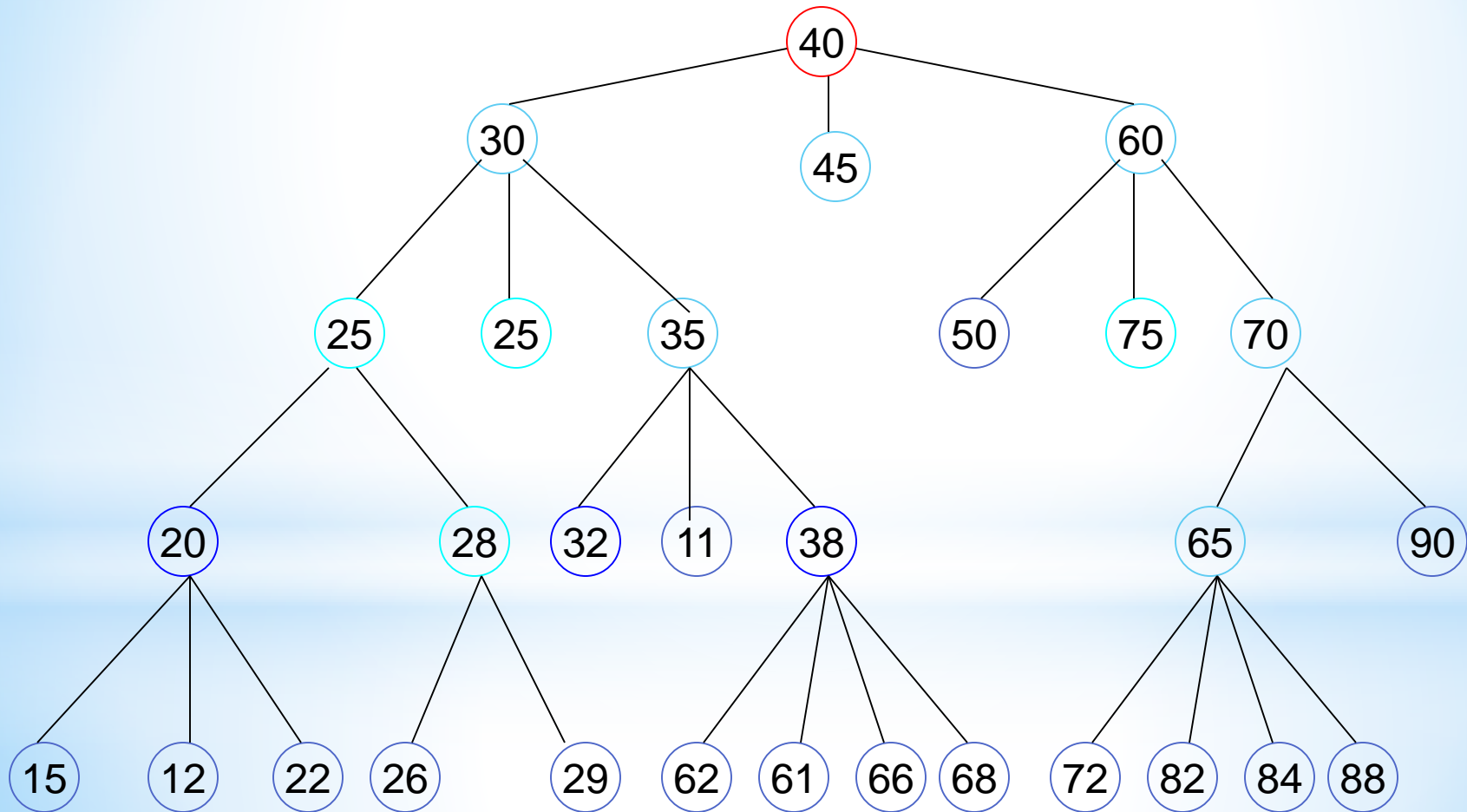
```
struct node* insert(struct node* node, int key) {  
    //Bước 1. Thực hiện thêm node trên cây tìm kiếm thông thường  
    if (node == NULL) return(newNode(key));  
    if (key < node->key) node->left = insert(node->left, key);  
    else node->right = insert(node->right, key);  
    //Bước 2. Cập nhật độ cao của node trước  
    node->height = max(height(node->left), height(node->right)) + 1;  
    //Bước 3. Cân bằng cây  
    int balance = getBalance(node);  
    if (balance > 1 && key < node->left->key) return rightRotate(node);  
    if (balance < -1 && key > node->right->key) return leftRotate(node);  
    if (balance > 1 && key > node->left->key) {  
        node->left = leftRotate(node->left); return rightRotate(node);  
    }  
    if (balance < -1 && key < node->right->key) {  
        node->right = rightRotate(node->right);  
        return leftRotate(node);  
    }  
    return node;  
}
```


Bài tập: Xây dựng các thao tác trên cây AVL:

```
class avlTree {  
    public:  
        int height(avl_node *); //trả lại độ cao của cây avl  
        int diff(avl_node *); //trả lại chỉ số cân bằng của cây avl  
        avl_node *rr_rotation(avl_node *); //thao tác quay phải –phải  
        avl_node *ll_rotation(avl_node *); //thao tác quay trái-trái  
        avl_node *lr_rotation(avl_node *); //thao tác quay trái –phải  
        avl_node *rl_rotation(avl_node *); //thao tác quay phải –trái  
        avl_node* balance(avl_node *); //cân bằng lại cây  
        avl_node* insert(avl_node *, int ); //thêm node vào cây avl  
        avl_node* delete(avl_node *, int ); //thêm node vào cây avl  
        void display(avl_node *, int); //thao tác quay phải –phải  
        void inorder(avl_node *);  
        void preorder(avl_node *);  
        void postorder(avl_node *);  
        avlTree() { root = NULL; }  
};
```

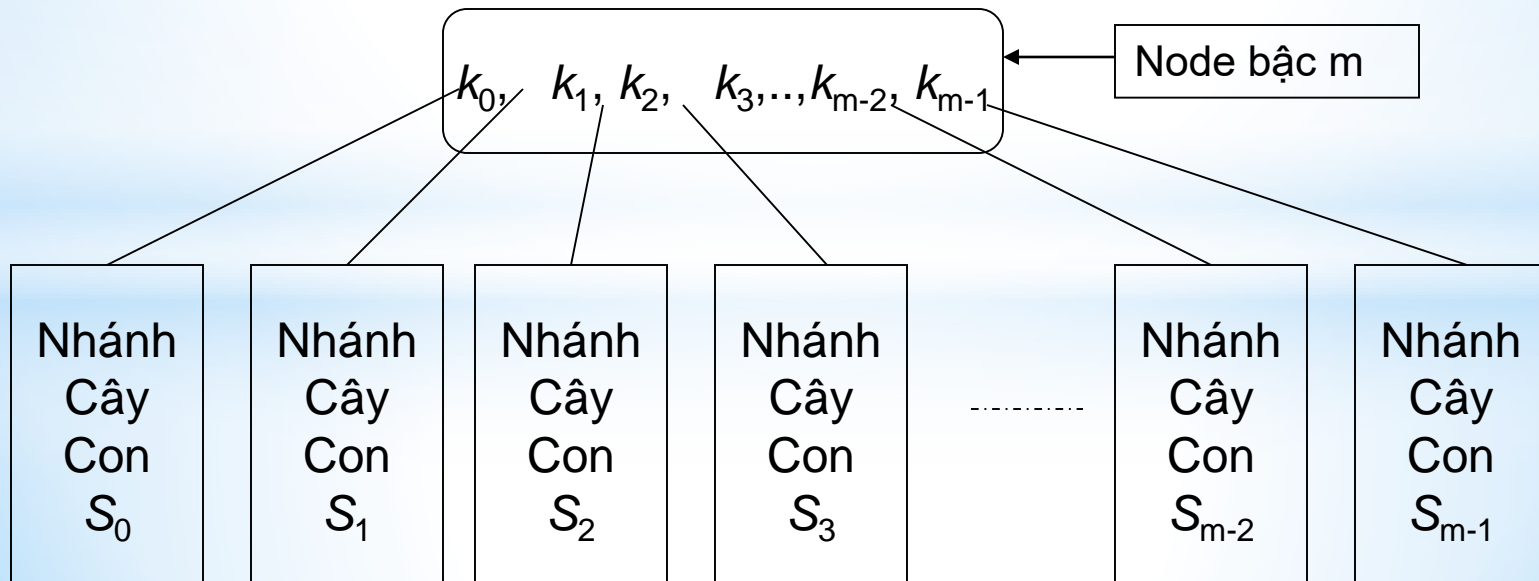

4.8. Cây nhiều nhánh: Tập hữu hạn các node có cùng kiểu dữ liệu được chia thành các tập, trong đó:

- Hoặc là tập rỗng hoặc có một node gốc, mỗi node gốc có thể có nhiều khóa đã được sắp xếp.
- Các tập con còn lại hoặc là rỗng hoặc cũng tự hình thành nên các cây nhiều nhánh.

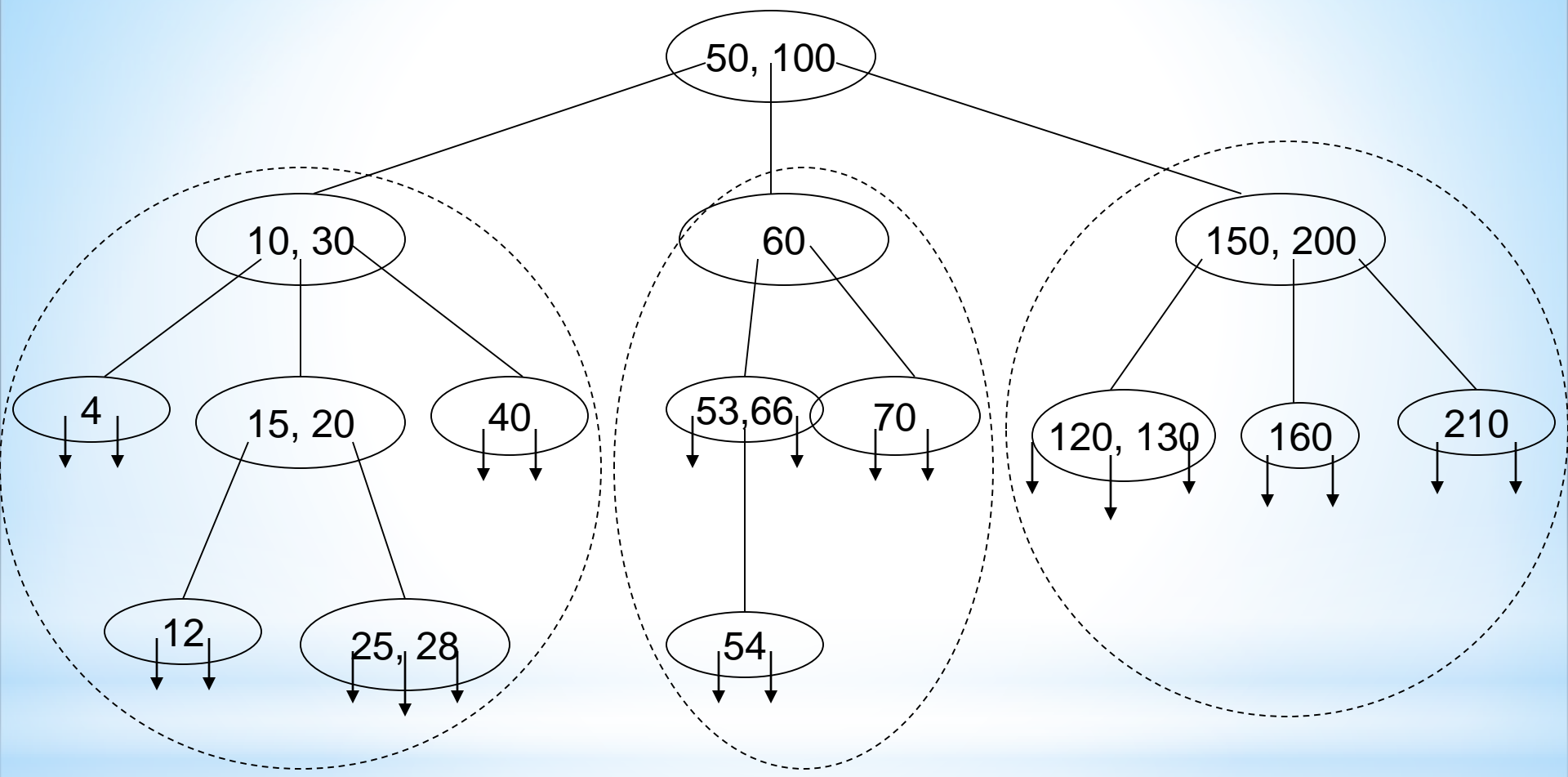


4.8.1. Cây nhiều nhánh tìm kiếm: Cây nhị phân nhiều nhánh tìm kiếm là cây mà mỗi node bậc m của cây thỏa mãn các tính chất sau:

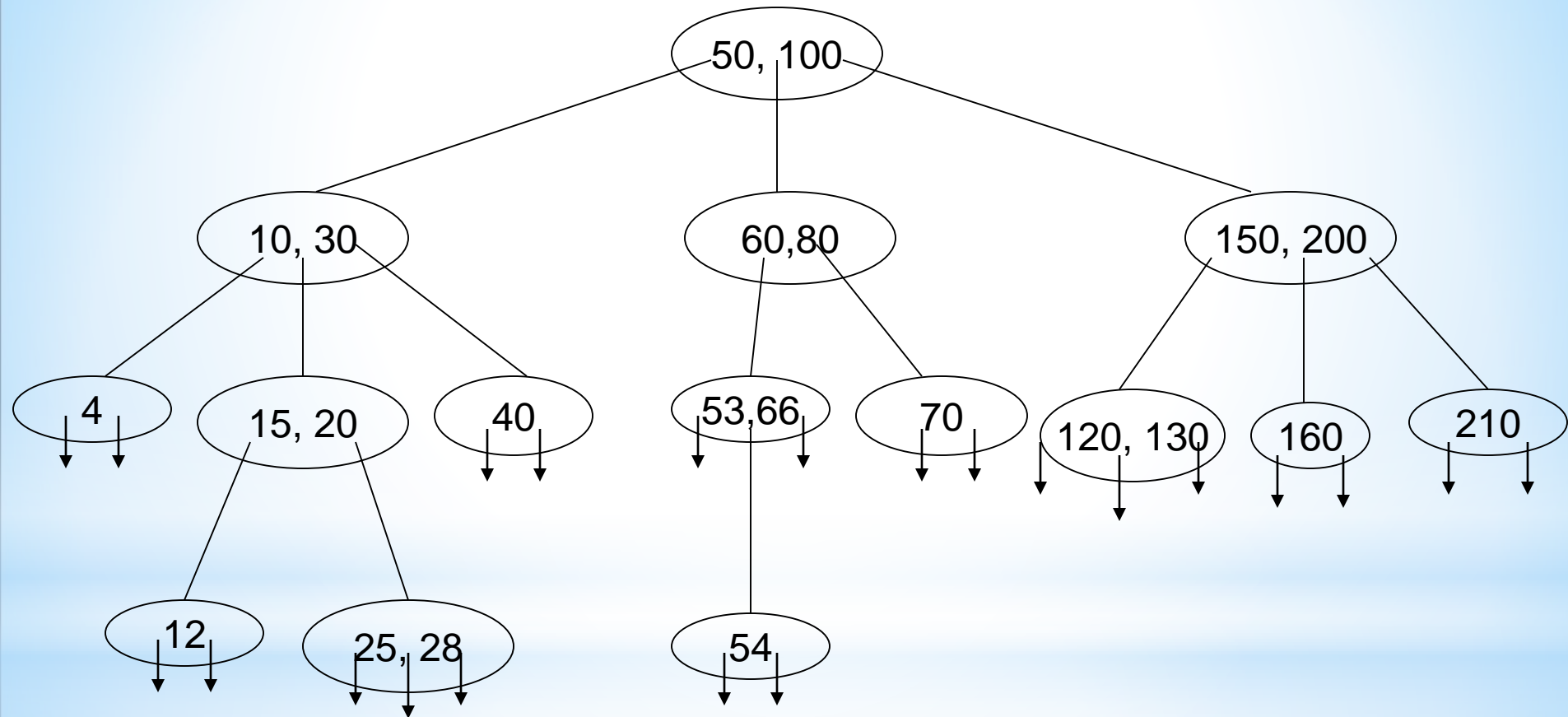
- Mỗi node bậc m của cây có thể bao gồm nhiều khóa $k_0, k_1, k_2, \dots, k_{m-1}$ đã được sắp xếp.
- Nội dung các khóa của node trên nhánh cây con bên trái nhất đều nhỏ hơn nội dung khóa k_0 . Nội dung các khóa của node trên nhánh cây con bên phải nhất đều lớn hơn nội dung khóa k_{m-1} .
- Nội dung các node nằm giữa khóa k_i, k_{i+1} có giá trị lớn hơn k_i và nhỏ hơn k_{i+1} ($i=1, 2, \dots, m-1$).
- Tất cả các cây con trái nhất, phải nhất và cây con ở giữa đều hình thành nên các cây nhiều nhánh tìm kiếm.



Ví dụ về cây nhiều nhánh tìm kiếm

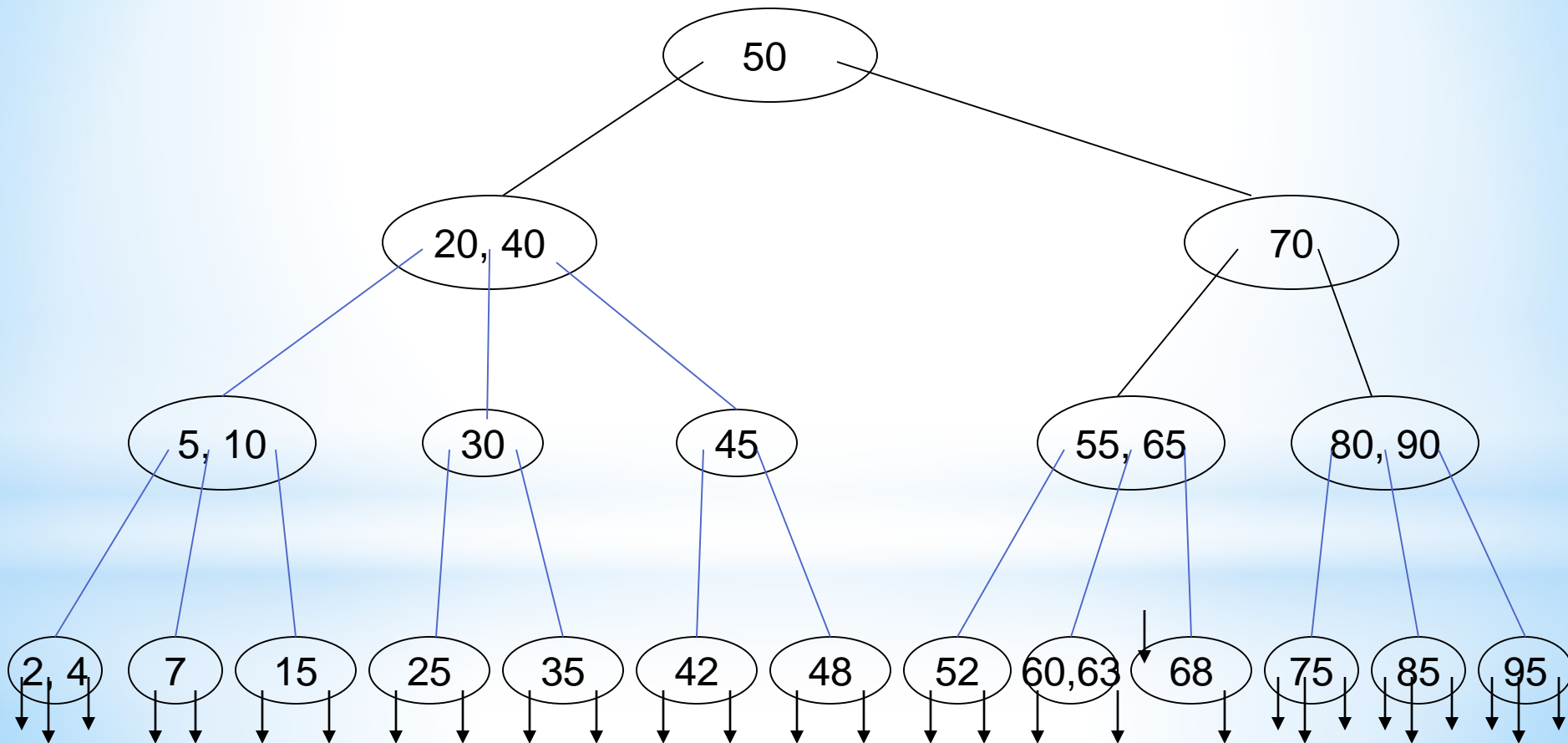


4.8.3. Cây Top-Down: Cây Top-Down là cây tìm kiếm nhiều nhánh mà các node không đầy đều là node lá. Trong đó, node gốc là node đầy.



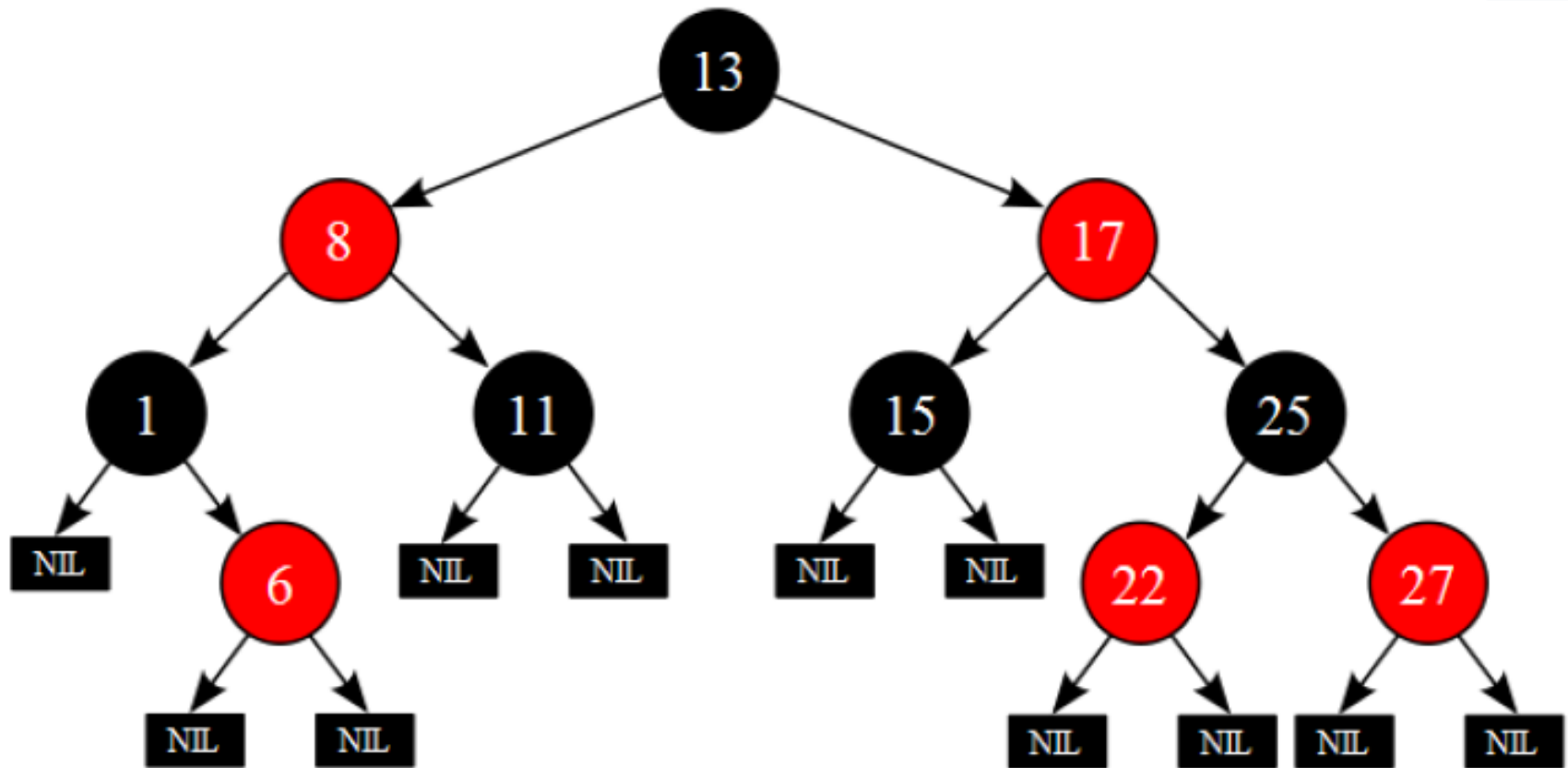
4.8.3. B - Tree: B-Cây bậc M là cây nhị phân tìm kiếm bậc M có tính chất:

- Mỗi node (ngoại trừ node gốc có ít nhất $M/2$ node con).
- Node gốc có ít nhất hai node con.
- Mọi node lá đều nằm cùng một mức
- Các khóa và cây con được sắp xếp theo cây tìm kiếm.



4.8.5. Cây đỏ đen: Là cây nhị phân tìm kiếm có tính chất:

- Mọi node phải có tính chất đỏ hoặc đen.
- Node gốc và các node luôn là node đen.
- Nếu một node là đỏ thì node con của nó là đen
- Đường đi từ node gốc đến node lá phải có cùng số lượng node đen.



4.9. CASE STUDY

- Xây dựng tập thao tác trên cây nhị phân.
- Xây dựng tập thao tác trên cây nhị phân tìm kiếm.
- Xây dựng tập thao tác trên cây nhị phân tìm kiếm cân bằng.
- Xây dựng tập thao tác trên cây tìm kiếm TOP-DOWN.
- Xây dựng tập thao tác trên cây tìm kiếm B-TREE.
- Xây dựng tập thao tác trên cây tìm kiếm RED BLACK TREE.
- Xây dựng tập thao tác trên cây tìm kiếm DECISION TREE.