

B-Tree



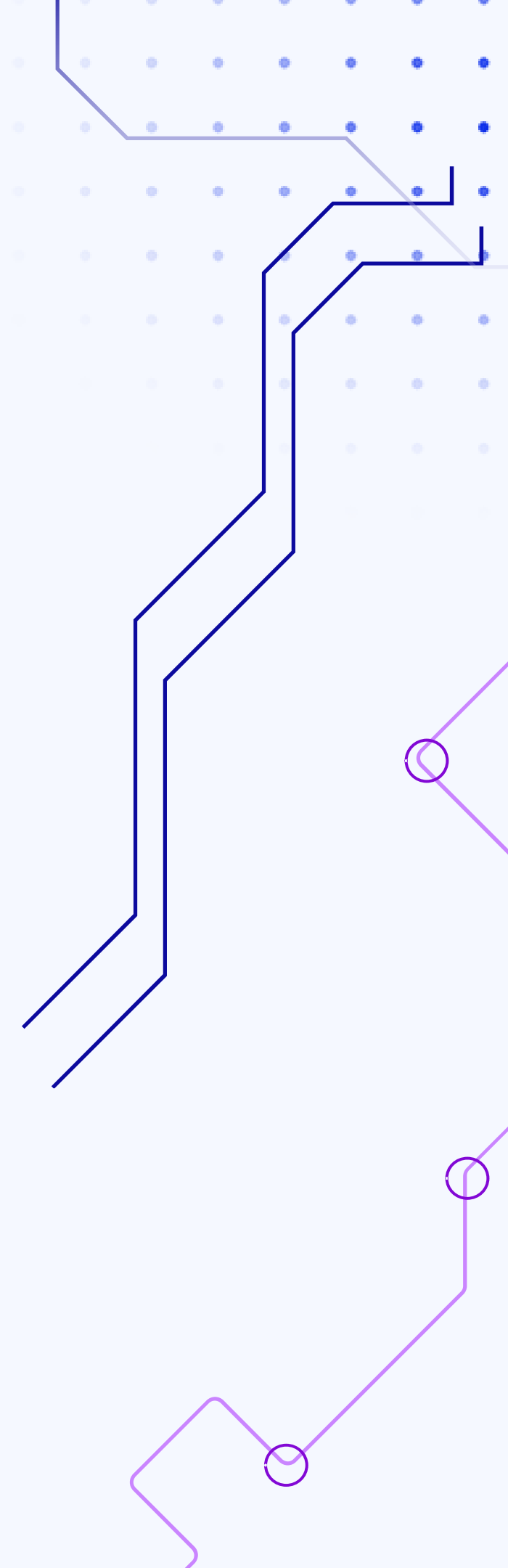
A. Giới thiệu

B. Các thao tác trên B-Tree:

- Tìm kiếm
- Chèn khóa
- Xóa khóa

C. Độ phức tạp

D. Ứng dụng của B-Tree



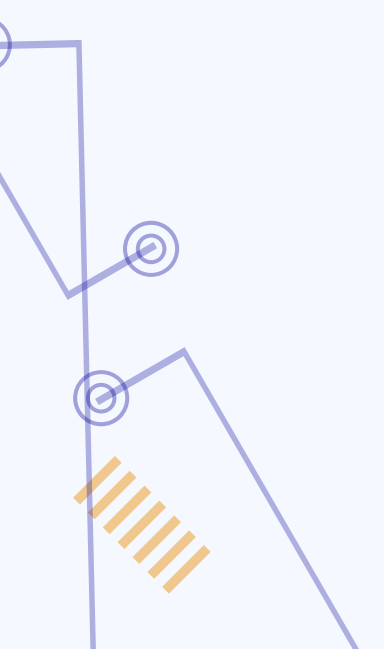


A. Giới thiệu

1, Một số khái niệm liên quan

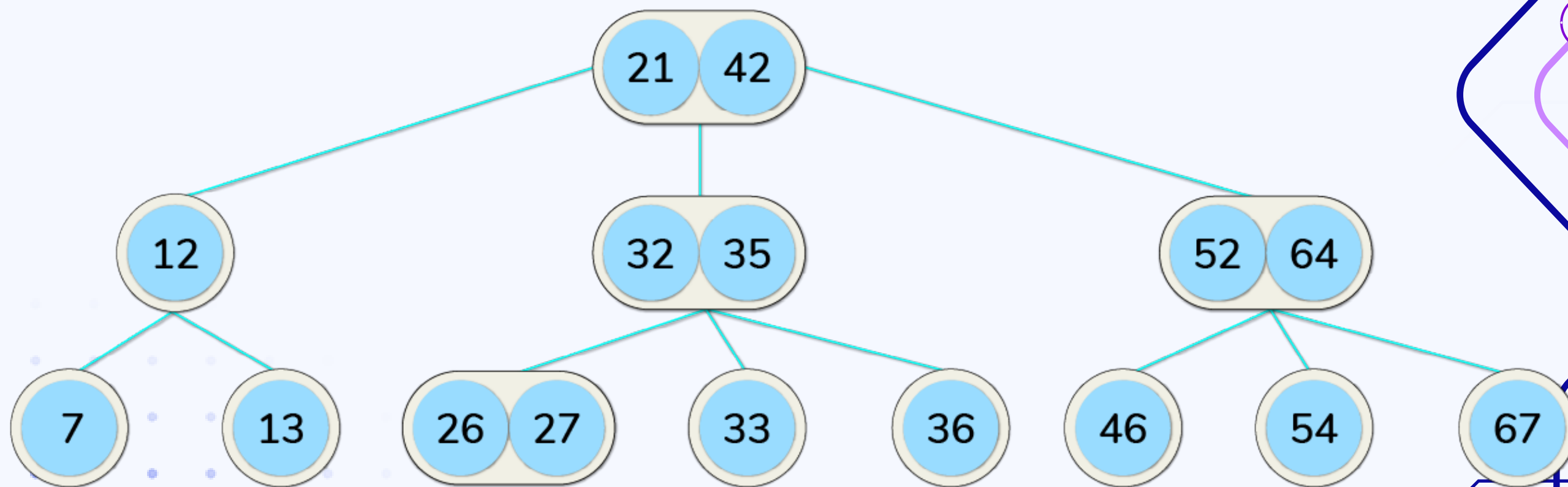
Cây cân bằng: Là loại cây mà số lượng nút ở cây con trái và ở cây con phải lệch nhau không quá một phần tử.

Cây nhị phân tìm kiếm cân bằng: Là cây nhị phân tìm kiếm mà độ cao của cây con trái và của cây con phải lệch nhau không quá một đơn vị.



2, B-Tree

- B-Tree là một loại cây nhị phân tìm kiếm tự cân bằng đặc biệt.
- B-Tree có khả năng lưu trữ số lượng lớn dữ liệu, mỗi nút có thể chứa nhiều khóa được sắp xếp theo thứ tự tăng dần.



2, B-Tree

Cấu trúc của BNode và Btree:

```
struct node
{
    int* keys;
    //t là số khóa tối thiểu --> tối đa 2*t-1 khóa
    int t;
    struct node** Children;
    int n;
    bool leaf;
};
typedef struct node NODE;
```

```
struct btree
{
    NODE* root;
    //t là bậc tối thiểu
    int t;
};
typedef struct btree
```

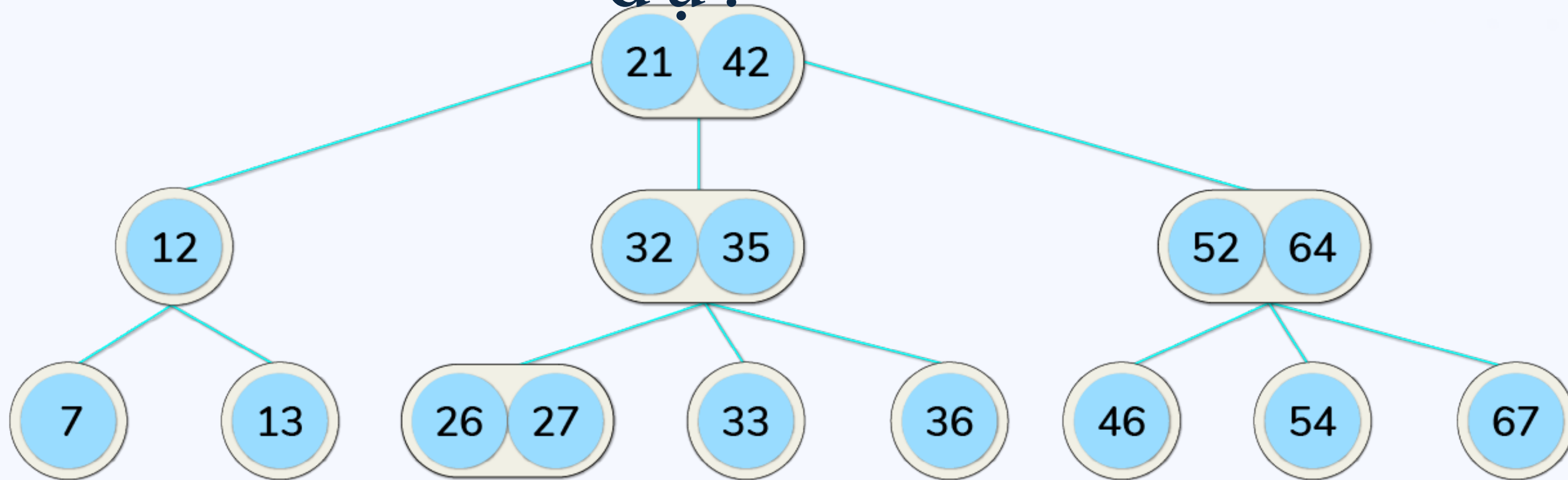
Các ràng buộc trong B - Tree:

Gọi n là bậc của cây, khi đó:

- Trong mỗi nút, các khóa được lưu trữ theo thứ tự tăng dần.
- Mỗi nút trong cây chứa tối đa $n - 1$ khóa, tối thiểu $n/2$ khóa và một con trỏ tới mỗi nút con.
- Mỗi nút (trừ nút gốc) chứa tối đa n nút con và tối thiểu $\lceil n/2 \rceil$ nút con.
- Trong mỗi nút, có một giá trị kiểu boolean là `x.isLeaf` sẽ có giá trị là `true` nếu x là nút lá. Tất cả các nút lá đều có cùng độ sâu (tức là chiều cao h của cây).
- Nút gốc có ít nhất hai nút con và chứa tối thiểu một khóa.

Ví

dụ:



B. Các thao tác trên B -

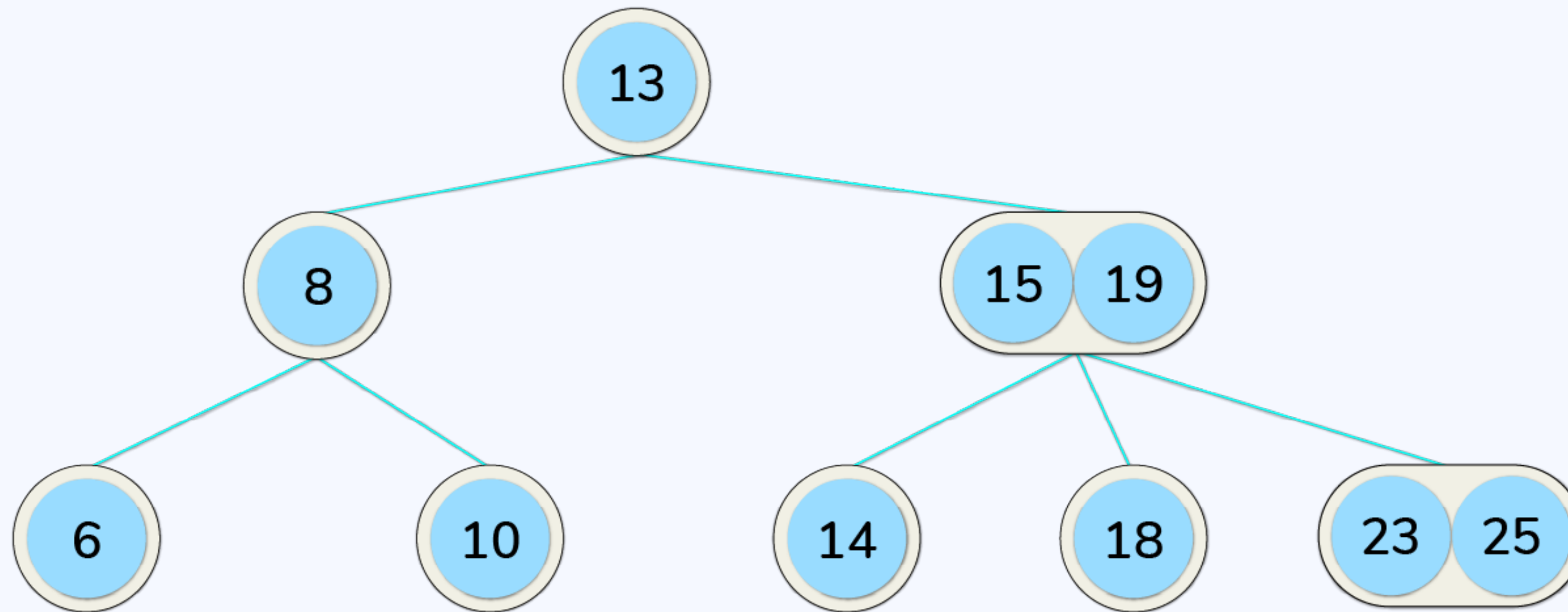
1, Thao tác tìm kiếm:

Tìm kiếm phần tử là hình thức tổng quát của việc tìm kiếm phần tử trong cây tìm kiếm nhị phân. Các bước được thực hiện như sau:

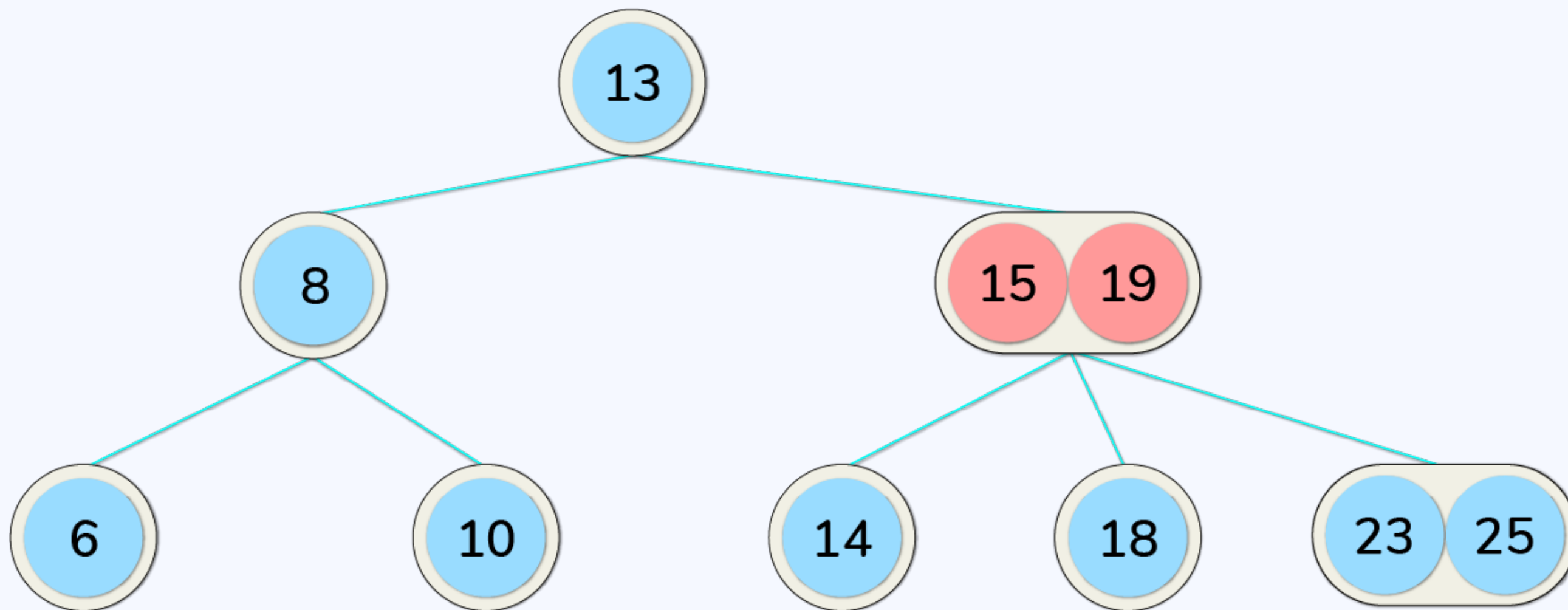
1. Bắt đầu từ nút gốc, so sánh k với khóa đầu tiên của nút. Nếu $k =$ khóa đầu tiên của nút, trả về nút và chỉ số của nó.
2. Nếu $k.\text{leaf}$ có giá trị là true, trả về NULL (tức là không tìm thấy).
3. Nếu k nhỏ hơn khóa đầu tiên của nút gốc, ta sẽ tìm kiếm đệ quy cho nút con bên trái của khóa này.
4. Nếu có nhiều hơn một khóa trong nút hiện tại và k lớn hơn khóa đầu tiên, ta sẽ so sánh k với khóa tiếp theo trong nút. Nếu k nhỏ hơn khóa tiếp theo, ta sẽ tìm kiếm nút con bên trái của khóa này. Nếu không, ta sẽ tìm kiếm nút con bên phải của khóa.
5. Lặp lại các bước từ 1 đến 4 cho đến khi tìm thấy k hoặc đạt đến nút lá.

Ví dụ: Tìm kiếm khóa $k=18$ trong B-Tree bậc 3:

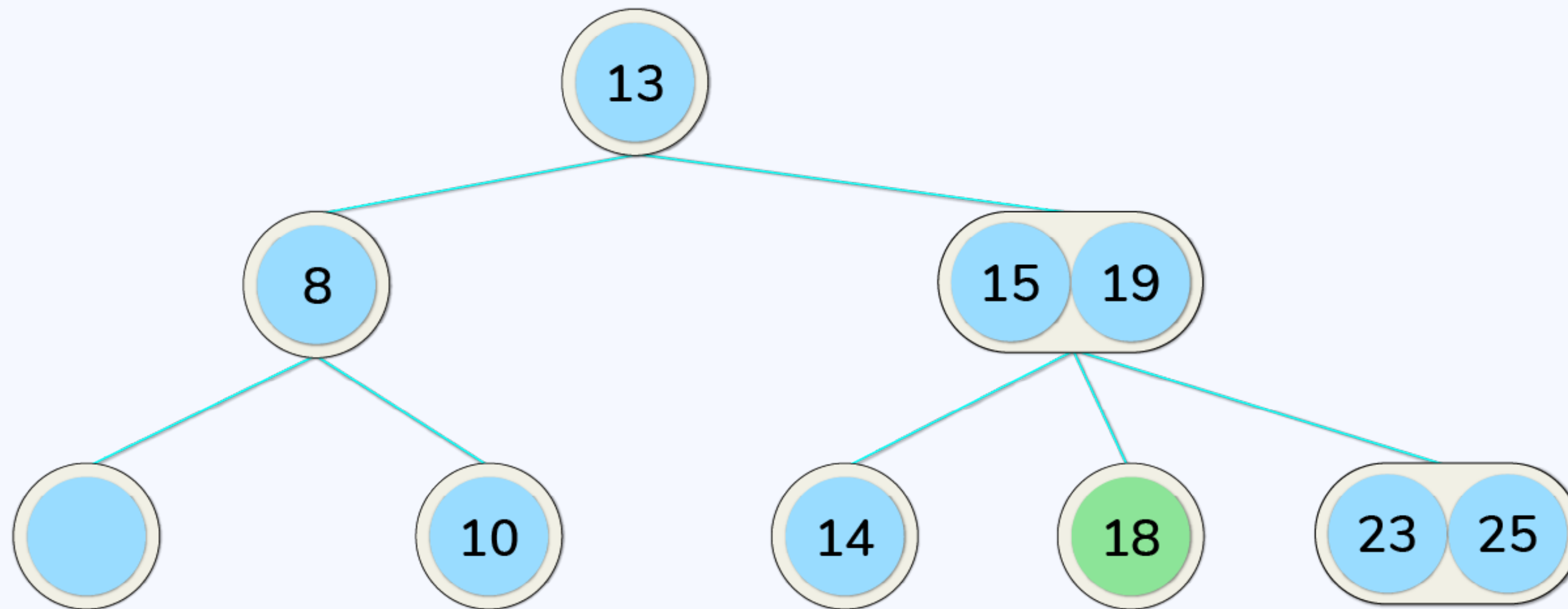
- So sánh khóa k với nút gốc, nhận thấy $k > 13$, ta chuyển đến nút con bên phải của nút gốc:



- So sánh k với 15. Vì $k > 15$ nên ta sẽ so sánh k với khóa tiếp theo là 19:



- Vì $k < 19$ nên k nằm trong khoảng từ 15 đến 19. Ta sẽ tìm kiếm nút con bên phải của khóa 15 hoặc nút con bên trái của 19.
- Và cuối cùng thì khóa k được tìm thấy là 18.





```
NODE* Bsearch(BTREE bt, int k)
{
    if (bt.root == NULL)
        return NULL;
    return search(bt.root, k);
}
```



```
NODE* search(NODE* p, int k)
{
    int i = 0;
    while ((i < p->n) && (k > p->keys[i]))
        i++;

    if (p->keys[i] == k)
        return p;

    if (p->leaf == true)
        return NULL;

    return search(p->Children[i], k);
}
```

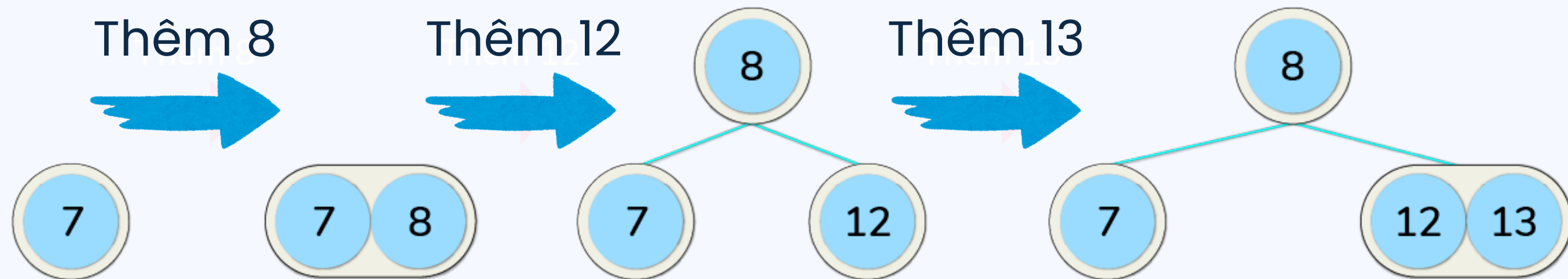
2, Thao tác chèn khóa:

Việc chèn một phần tử trên cây B-tree bao gồm hai sự kiện: tìm kiếm nút thích hợp để chèn phần tử và tách nút nếu cần thiết. Hoạt động chèn luôn diễn ra theo phương pháp từ dưới lên.

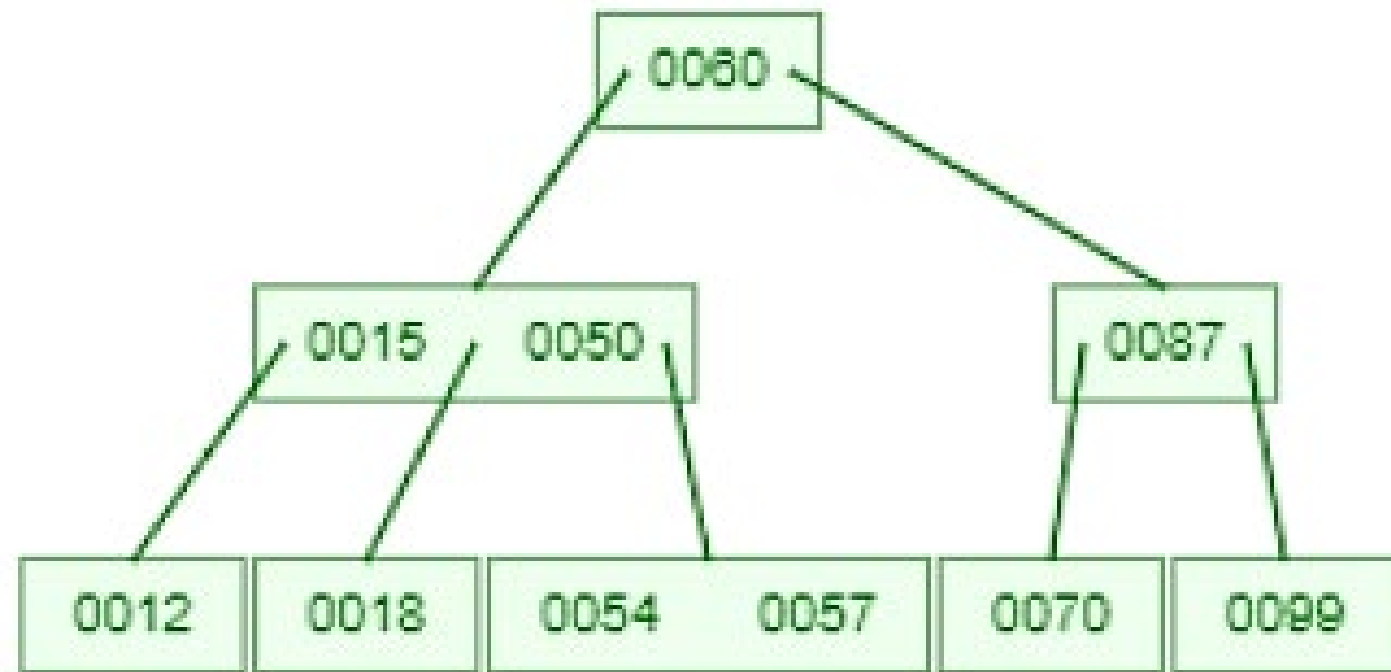
Thao tác chèn được thực hiện như sau:

1. Nếu cây trống rỗng, ta sẽ tạo một nút gốc và chèn khóa.
2. Cập nhật số lượng khóa được cho phép trong nút.
3. Tìm kiếm nút thích hợp để chèn.
4. Nếu nút đã đầy, ta sẽ làm theo các bước bên dưới.
5. Chèn các phần tử theo thứ tự tăng dần.
6. Có những phần tử lớn hơn so với giới hạn của nút. Vì vậy, ta sẽ tách ra ở phần giữa.
7. Đẩy khóa giữa lên trên và đặt khóa bên trái làm khóa của nút con bên trái và khóa bên phải làm nút con bên phải.
8. Nếu nút chưa đầy, ta sẽ làm theo bước bên dưới.
9. Chèn nút theo thứ tự tăng dần.

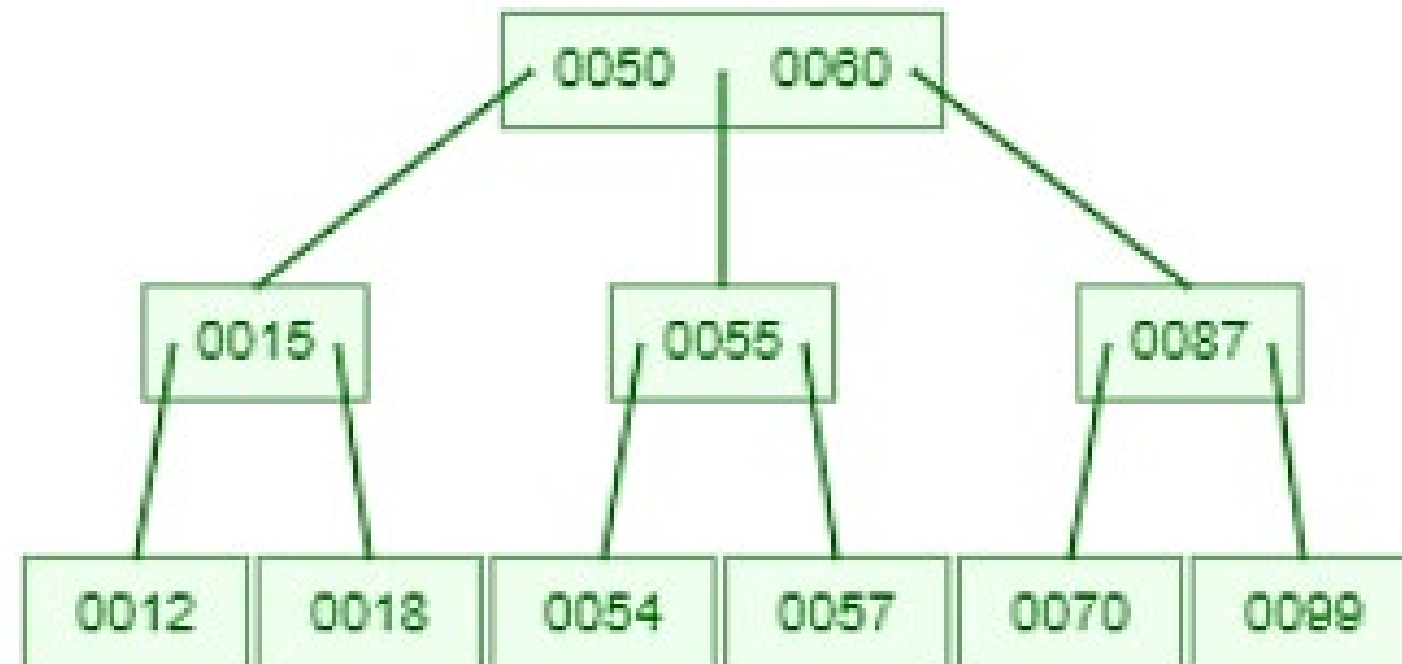
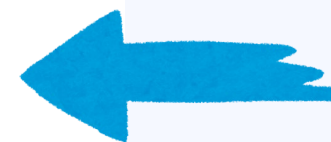
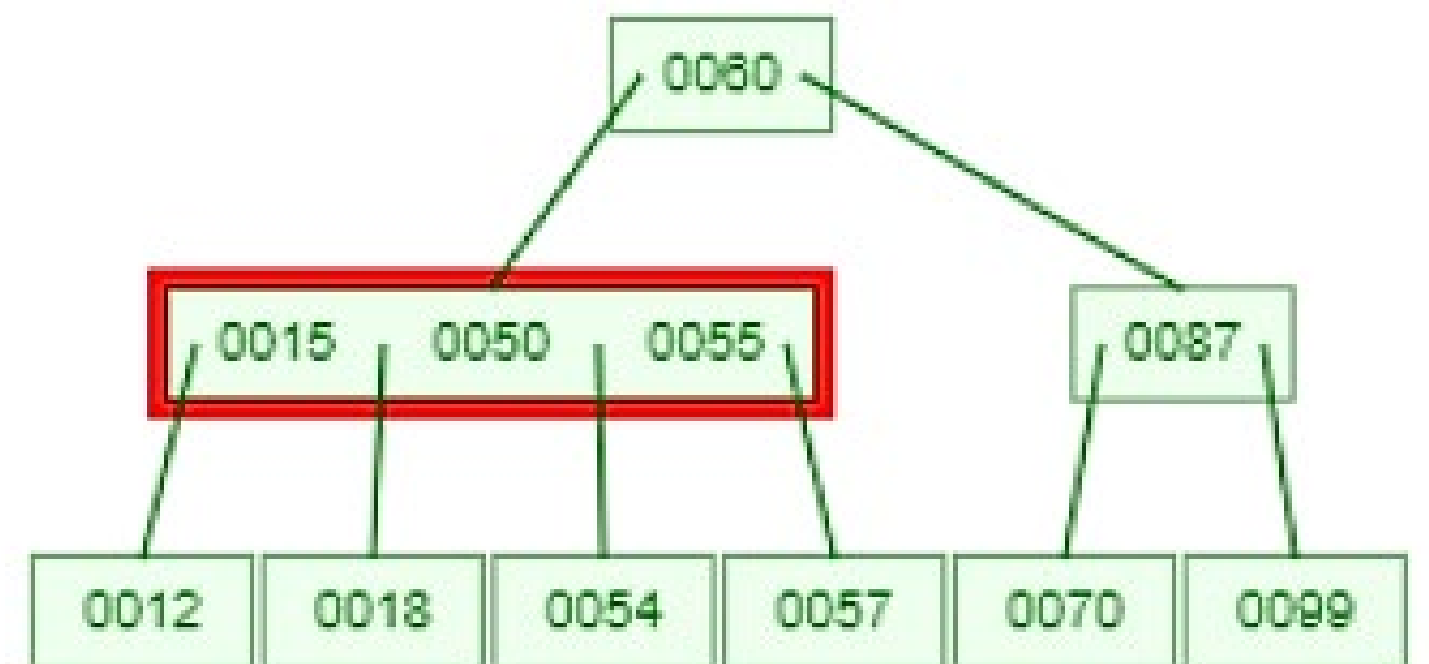
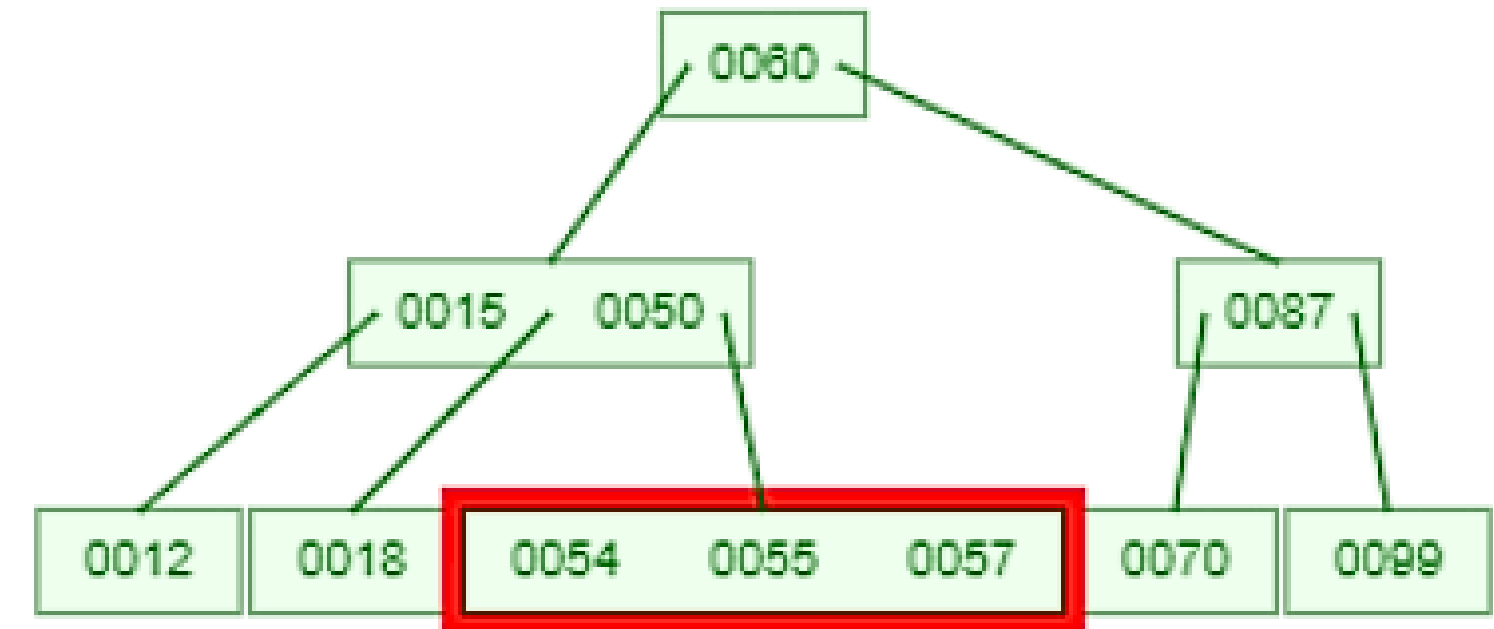
Ví
dụ :



Ví dụ :



Chèn 55




```

void Binsert(BTREE& bt, int k)
{
    if (bt.root == NULL)
    {
        bt.root = getNode(bt.t, true);
        bt.root->keys[0] = k;
        bt.root->n = 1;
    }

    else
    {
        if (bt.root->n == 2 * bt.t - 1)
        {
            // Cấp phát bộ nhớ cho nút gốc mới
            NODE* s = getNode(bt.t, false);

            // Đặt nút cũ là con của nút gốc mới
            s->Children[0] = bt.root;

            // Chia nút gốc cũ và di chuyển 1 khóa sang nút gốc mới
            splitChild(0, s, bt.root);

            // Nút gốc mới có hai con bây giờ. Quyết định con nào sẽ chứa khóa mới
            int i = 0;
            if (s->keys[0] < k)
                i++;
            insertNonFull(s->Children[i], k);

            //Thay đổi nút gốc
            bt.root = s;
        }
        else // nút gốc không đầy
            insertNonFull(bt.root, k);
    }
}

```

```

NODE* getNode(int t, bool lf)
{
    NODE* p = new NODE;

    p->t = t;
    p->leaf = lf;

    p->keys = new int[2 * t - 1];
    p->Children = new NODE * [2 * t];

    p->n = 0;

    return p;
}

```

```

void splitChild(int i, NODE* p, NODE* y)
{
    // Tạo một nút mới mà sẽ lưu (t-1) khóa của y
    NODE* z = getNode(y->t, y->leaf);
    z->n = p->t - 1;

    // Sao chép (t-1) khóa cuối cùng của y sang z
    for (int j = 0; j < p->t - 1; j++)
        z->keys[j] = y->keys[j + p->t];

    // Sao chép t con cuối cùng của y sang z
    if (y->leaf == false)
    {
        for (int j = 0; j < p->t; j++)
            z->Children[j] = y->Children[j + p->t];
    }

    // Giảm số lượng khóa trong y
    y->n = p->t - 1;

    // Vì nút này sẽ có một con mới,
    // tạo không gian cho con mới
    for (int j = p->n; j >= i + 1; j--)
    {
        p->Children[j + 1] = p->Children[j];
    }

    // Liên kết con mới này với nút này
    p->Children[i + 1] = z;

    // Một khóa của y sẽ di chuyển sang nút này. Tìm vị trí của
    // khóa mới và di chuyển tất cả các khóa lớn hơn một vị trí
    for (int j = p->n - 1; j >= i; j--)
        p->keys[j + 1] = p->keys[j];

    // Sao chép khóa ở giữa của y sang nút này
    p->keys[i] = y->keys[p->t - 1];

    // Tăng số lượng khóa trong nút này
    p->n += 1;
}

```

```

void insertNonFull(NODE* p, int k)
{
    // Khởi tạo chỉ mục là chỉ mục của phần tử bên phải nhất
    int i = p->n - 1;

    if (p->leaf == true)
    {
        // Vòng lặp sau thực hiện hai việc
        // a) Tìm vị trí của khóa mới cần chèn
        // b) Di chuyển tất cả các khóa lớn hơn một vị trí
        while ((i >= 0) && (p->keys[i] > k))
        {
            p->keys[i + 1] = p->keys[i];
            i--;
        }

        // Chèn khóa mới vào vị trí đã tìm được
        p->keys[i + 1] = k;
        p->n += 1;
    }
    else
    {
        // Tìm con sẽ chứa khóa mới
        while ((i >= 0) && (p->keys[i] > k))
            i--;

        // Kiểm tra xem con đã tìm có đầy không
        if (p->Children[i + 1]->n == 2 * p->t - 1)
        {
            splitChild(i + 1, p, p->Children[i + 1]);

            // Sau khi chia, khóa ở giữa của Children[i] lên trên và
            // Children[i] được chia thành hai phần. Xem con nào sẽ chứa khóa mới
            if (p->keys[i + 1] < k)
                i++;
        }
        insertNonFull(p->Children[i + 1], k);
    }
}

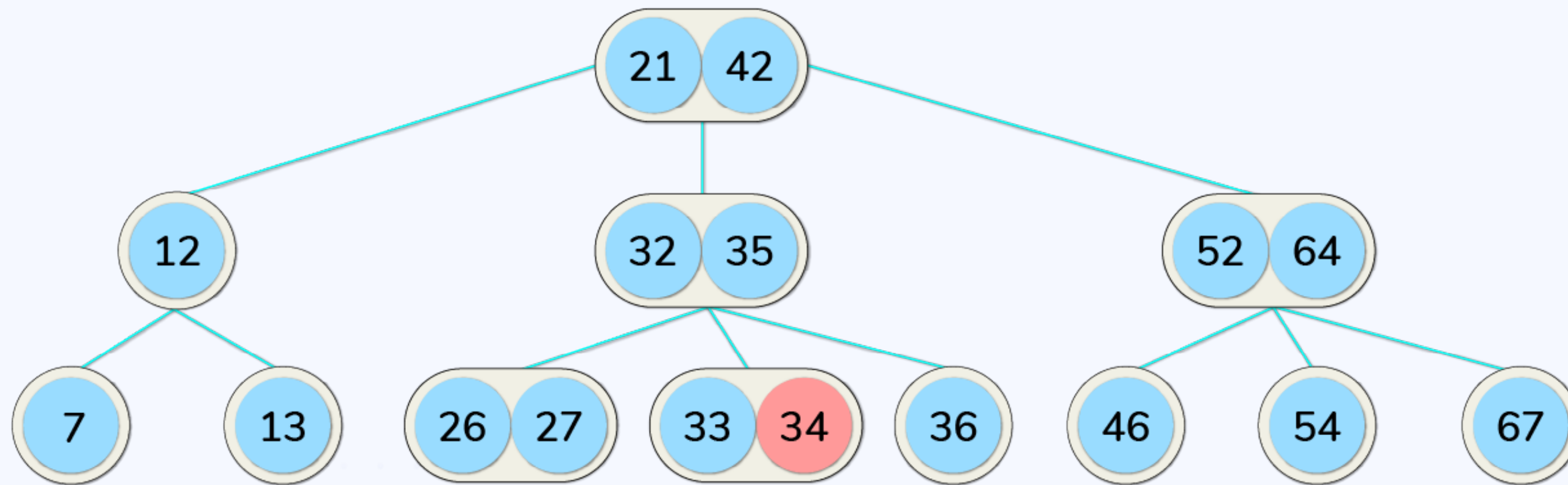
```

3, Thao tác xóa khóa:

Thao tác xóa khóa thường xảy ra Underflow – Vi phạm điều kiện về số khóa tối thiểu trong mỗi nút. (Mỗi nút trong cây chứa tối đa $n - 1$ khóa, tối thiểu $n/2$ khóa và một con trỏ tới mỗi nút con).

- TH1: Khóa cần xóa nằm trên nút

lấy: TH1.1: Việc xóa khóa không vi phạm điều kiện về số lượng khóa tối thiểu trong nút:
Delete khóa cần xóa và kết thúc.



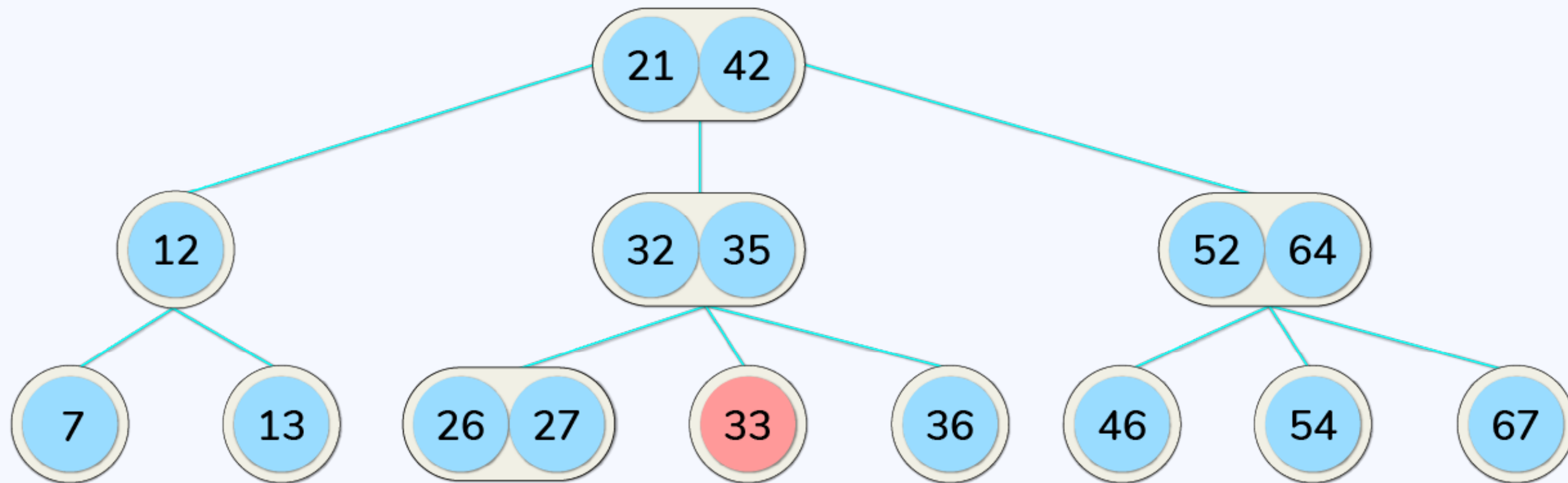


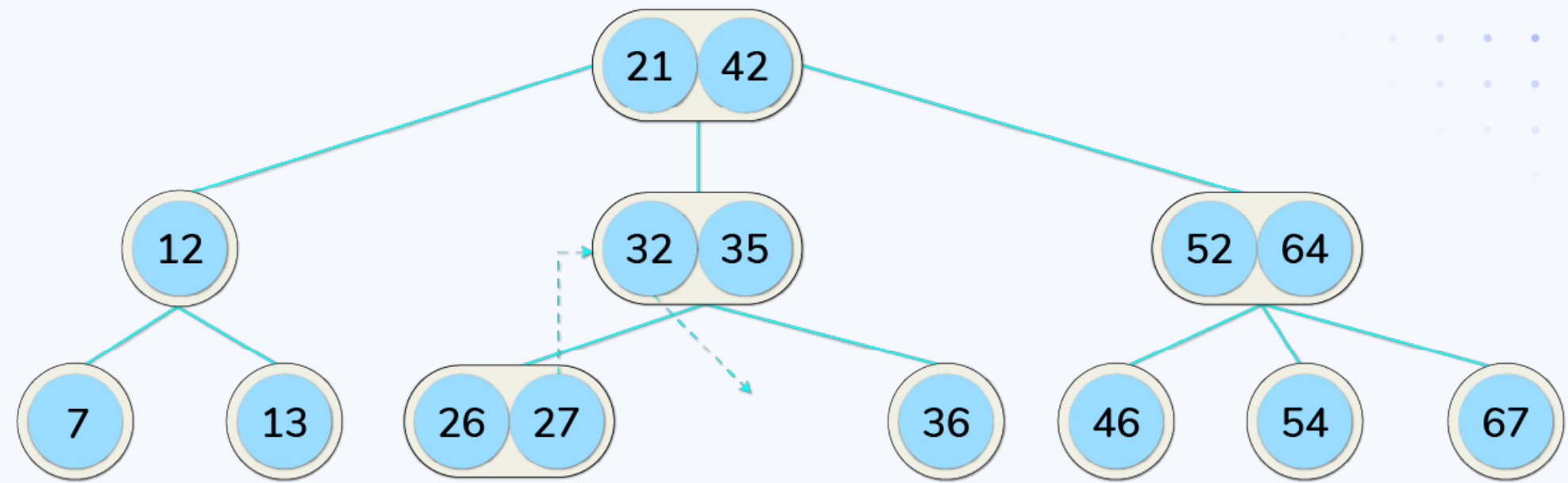
+TH1.2: Việc xóa khóa vi phạm điều kiện về số lượng khóa tối thiểu trong một nút:

- Mượn khóa : Mượn khóa của node anh em có nhiều hơn số lượng khóa tối thiểu, thứ tự duyệt để mượn từ trái qua phải.
- Trộn node : Nếu không mượn được khóa thì trộn node anh em trái và phải thông qua node cha.

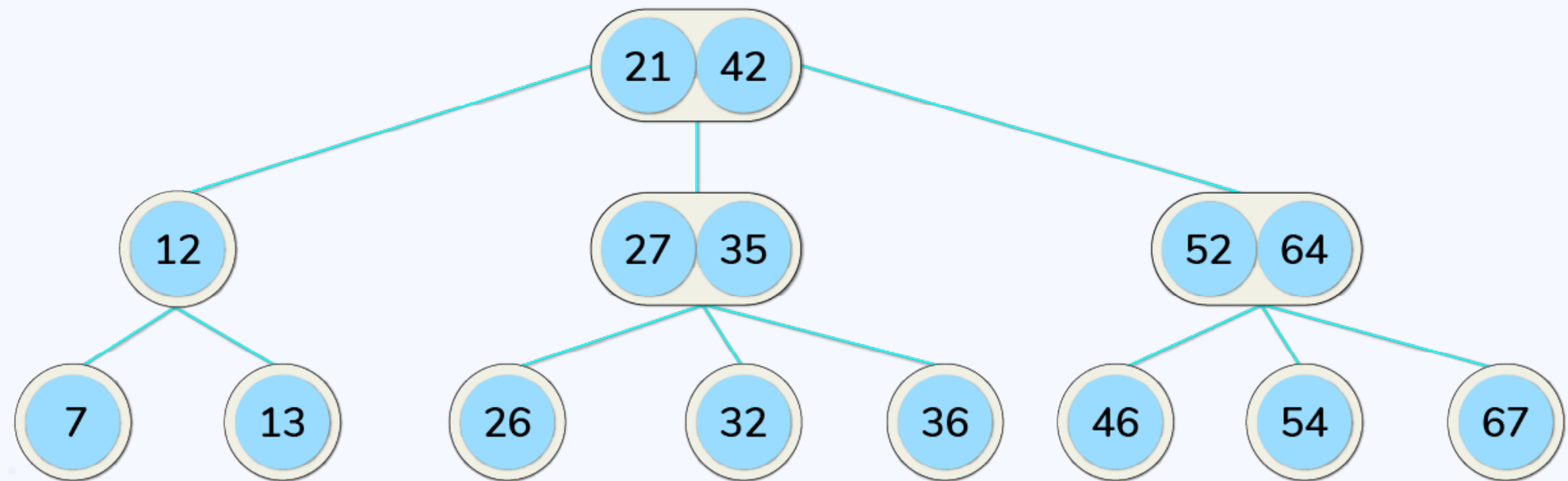
Mượn khóa:

- Đầu tiên, ta sẽ đến nút anh em bên trái. Nếu nút anh em bên trái có nhiều hơn một số lượng khóa tối thiểu, thì ta sẽ mượn khóa từ nút này.
- Nếu không, ta sẽ kiểm tra để mượn từ nút anh em bên phải.
- Trong cây bên dưới, xóa khóa có giá trị là 33 dẫn đến điều kiện trên.



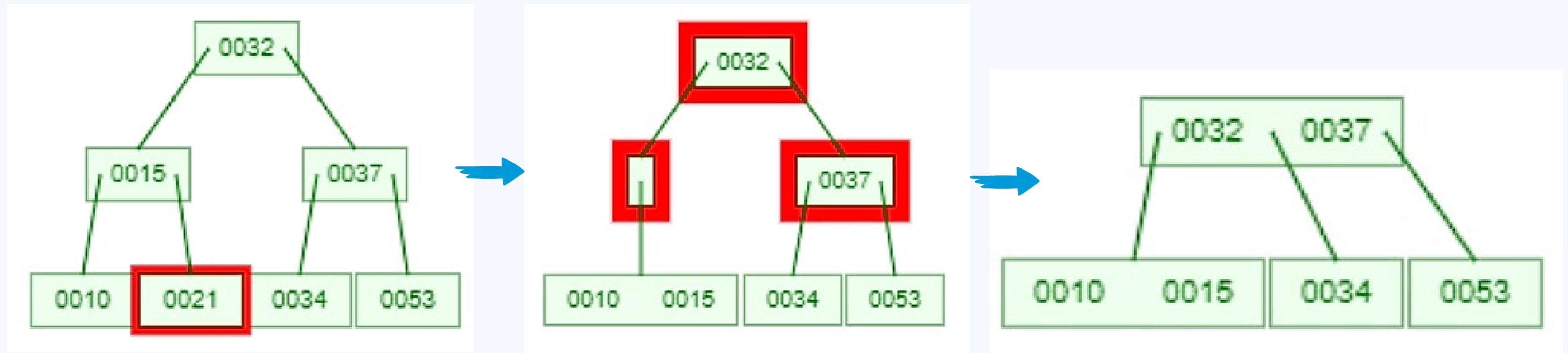


Mượn khóa 27 bên nút
anh em trái để thay thế.



Trộn node:

- Nếu cả hai nút anh em đã có số lượng khóa tối thiểu, thì ta sẽ hợp nhất nút đó với nút anh em bên trái hoặc nút anh em bên phải. Việc hợp nhất này được thực hiện thông qua nút cha.
- Ví dụ: xóa khóa 21 trong cây bên dưới:

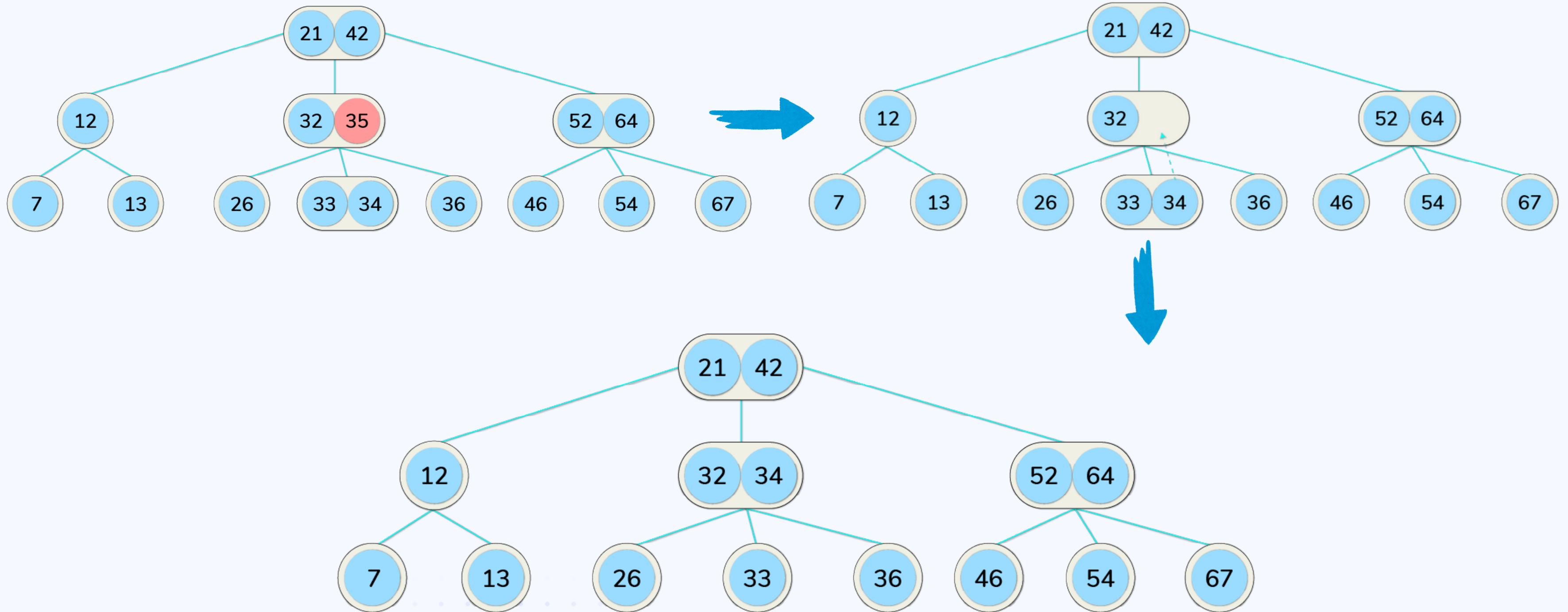


- TH2: Khóa cần xóa nằm trên node

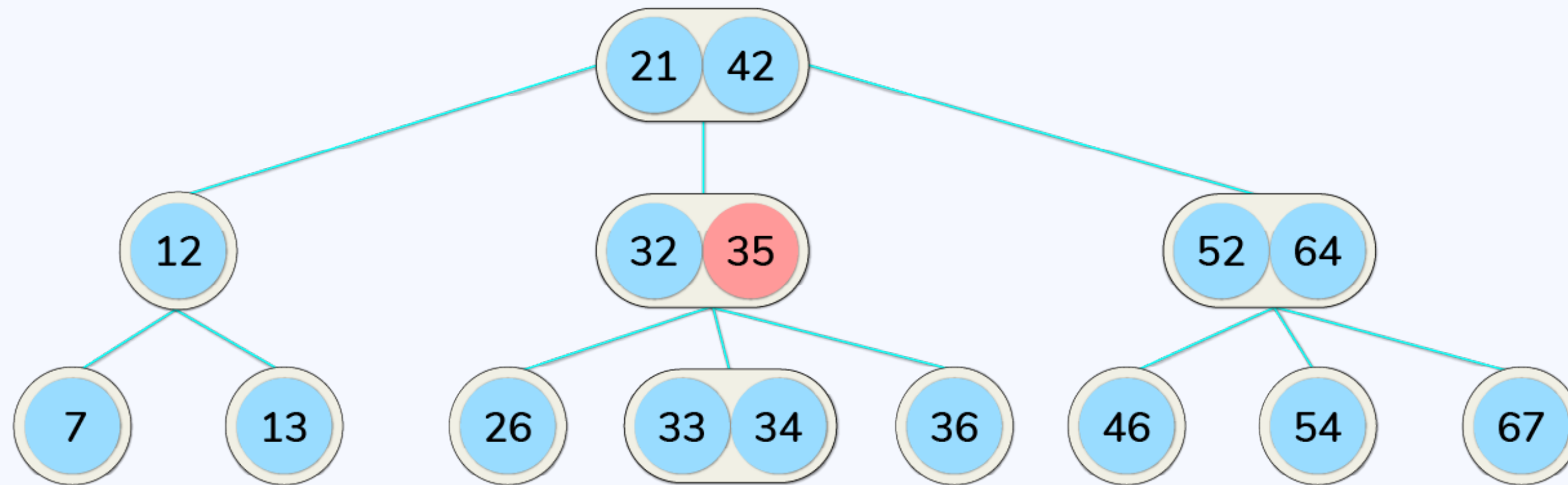
Các thuật ngữ cần hiểu: trong

1. Nút tiền nhiệm Inorder: Khóa lớn nhất của nút con bên trái được gọi là khóa tiền nhiệm Inorder.
2. Nút kế nhiệm Inorder: Khóa nhỏ nhất ở nút con bên phải được gọi là khóa kế nhiệm Inorder.

+TH2.1: Nút bên trong, bị xóa, được thay thế bằng khóa tiền nhiệm inoder nếu nút con bên trái có nhiều hơn số lượng khóa tối thiểu.
Ví dụ: Xóa khóa 35 trong cây B-Tree bên dưới:

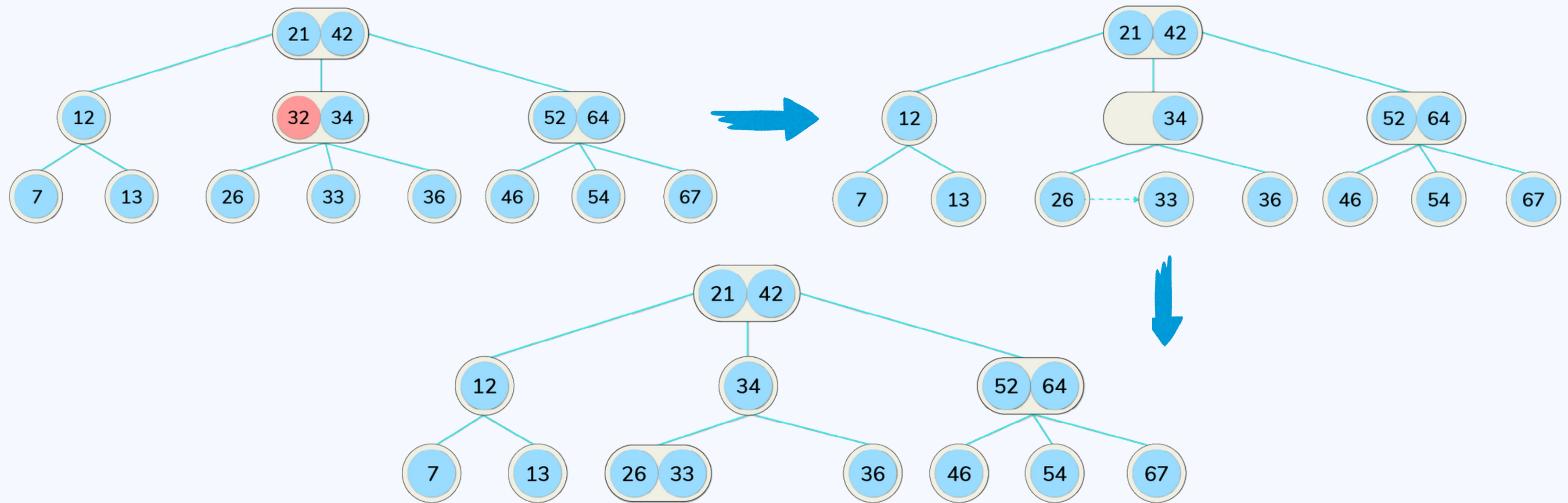


+TH2.2: Nút bên trong, bị xóa, được thay thế bằng khóa kế nhiệm inoder nếu nút con bên phải có nhiều hơn số lượng khóa tối thiểu.
Ví dụ: Xóa khóa 32 trong cây B-Tree trên:



+TH2.3: Nếu nút con có chính xác số lượng khóa tối thiểu thì ta sẽ hợp nhất nút con bên trái và bên phải.

Ví dụ: Xóa khóa 32 trong cây B-Tree bên dưới:

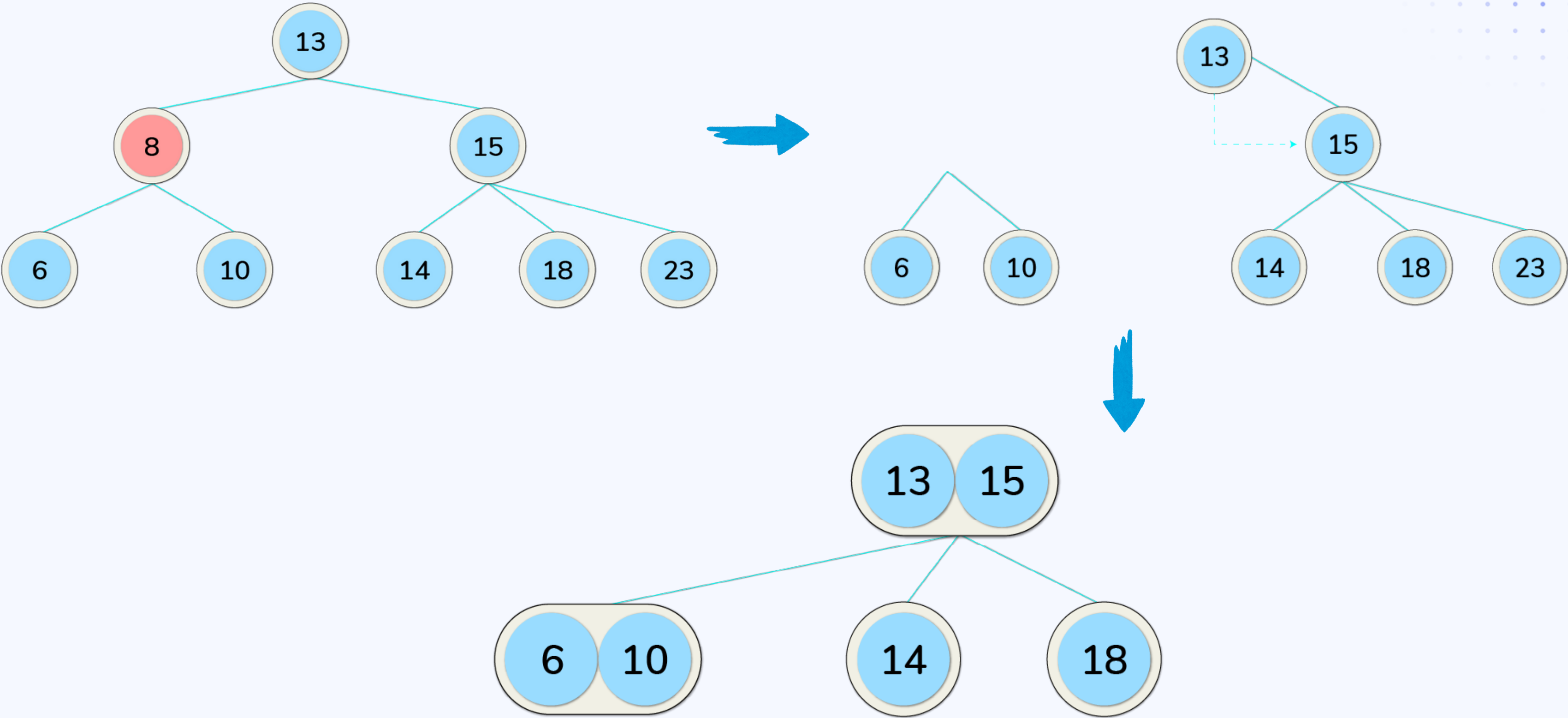


- Sau khi hợp nhất, nếu nút cha có ít hơn số khóa tối thiểu thì ta sẽ tìm các nút anh em như trong Trường hợp 1.

- TH3

- Trong trường hợp này, chiều cao của cây thu hẹp lại. Nếu khóa cần xóa nằm trong một nút bên trong và việc xóa khóa dẫn đến số lượng khóa trong nút ít hơn (tức là ít hơn mức tối thiểu cần thiết), thì ta sẽ tìm nút tiền nhiệm Inorder và nút kế nhiệm theo thứ tự Inorder. Nếu cả hai nút con đều chứa một số lượng khóa tối thiểu thì việc mượn không thể diễn ra. Điều này dẫn đến Trường hợp II (3) tức là hợp nhất các nút con.
- Ta sẽ tìm các nút anh em để mượn khóa. Nhưng, nếu nút anh em cũng chỉ có một số khóa tối thiểu thì ta sẽ hợp nhất nút với nút anh em cùng với nút cha. Sắp xếp các nút con phù hợp (thứ tự tăng dần).

Ví dụ: Xóa khóa có giá trị 8 trong cây B-Tree bên dưới:




```

void Bremove(BTREE& bt, int k)
{
    if (!bt.root)
    {
        cout << "Cây trống\n";
        return;
    }

    // Gọi hàm xóa cho nút gốc
    remove(bt.root, k);

    // Nếu nút gốc chỉ có một khóa và có một con, thì đặt nút gốc là con của nó và giải phóng nút gốc cũ
    if (bt.root->n == 0)
    {
        NODE* temp = bt.root;
        if (bt.root->leaf)
            bt.root = NULL;
        else
            bt.root = bt.root->Children[0];
        delete(temp);
    }
}

```

```

void remove(NODE* p, int k)
{
    int idx = findkey(p, k);

    // Khóa cần xóa có mặt trong nút này
    if ((idx < p->n) && (p->keys[idx] == k))
    {
        // Nếu nút là nút lá - gọi removeFromLeaf
        // Ngược lại, gọi removeFromNonLeaf
        if (p->leaf == true)
            removeFromLeaf(p, idx);
        else
            removeFromNonLeaf(p, idx);
    }
    else
    {
        // Nếu nút này là nút lá, thì khóa không tồn tại trong cây
        if (p->leaf == true)
        {
            cout << "Khóa " << k << " không tồn tại trong cây\n";
            return;
        }

        // Khóa cần xóa có mặt trong cây con gốc này
        // Chỉ ra liệu khóa có mặt trong con cuối cùng của nút này không
        bool flag;
        if (idx == p->n)
            flag = true;
        else
            flag = false;

        // Nếu con mà khóa cần xóa tồn tại có ít hơn t khóa,
        // chúng ta điền con đó
        if (p->Children[idx]->n < p->t)
            Fill(p, idx);

        // Nếu con cuối cùng đã được hợp nhất, nó phải đã được hợp nhất với con trước
        // chúng ta đệ quy trên con (idx-1). Ngược lại, chúng ta đệ quy trên con (idx)
        // mà bây giờ có ít nhất t khóa
        if ((flag == true) && (idx > p->n))
            remove(p->Children[idx - 1], k);
        else
            remove(p->Children[idx], k);
    }
}

```

```

void remove(NODE* p, int k)
{
    int idx = findkey(p, k);

    // Khóa cần xóa có mặt trong nút này
    if ((idx < p->n) && (p->keys[idx] == k))
    {
        // Nếu nút là nút lá - gọi removeFromLeaf
        // Ngược lại, gọi removeFromNonLeaf
        if (p->leaf == true)
            removeFromLeaf(p, idx);
        else
            removeFromNonLeaf(p, idx);
    }
    else
    {
        // Nếu nút này là nút lá, thì khóa không tồn tại trong cây
        if (p->leaf == true)
        {
            cout << "Khóa " << k << " không tồn tại trong cây\n";
            return;
        }

        // Khóa cần xóa có mặt trong cây con gốc này
        // Cờ chỉ ra liệu khóa có mặt trong con cuối cùng của nút này không
        bool flag;
        if (idx == p->n)
            flag = true;
        else
            flag = false;

        // Nếu con mà khóa cần xóa tồn tại có ít hơn t khóa,
        // chúng ta điền con đó
        if (p->Children[idx]->n < p->t)
            Fill(p, idx);

        // Nếu con cuối cùng đã được hợp nhất, nó phải đã được hợp nhất với con trước
        // chúng ta đệ quy trên con (idx-1). Ngược lại, chúng ta đệ quy trên con (idx)
        // mà bây giờ có ít nhất t khóa
        if ((flag == true) && (idx > p->n))
            remove(p->Children[idx - 1], k);
        else
            remove(p->Children[idx], k);
    }
}

```

```

void removeFromLeaf(NODE* p, int idx)
{
    // Di chuyển tất cả các khóa sau chỉ mục idx một vị trí về phía trước
    for (int i = idx + 1; i < p->n; ++i)
    {
        p->keys[i - 1] = p->keys[i];
    }
    p->n--;
}

```

```

void removeFromNonLeaf(NODE* p, int idx)
{
    int k = p->keys[idx];

    // Nếu con trước k (Children[idx]) có ít nhất t khóa,
    // tìm phần tử trước 'pred' của k trong cây con gốc
    // tại con[idx]. Thay thế k bằng pred. Đệ quy xóa pred trong con[idx]
    if (p->Children[idx]->n >= p->t)
    {
        int pred = getPredecessor(p, idx);
        p->keys[idx] = pred;
        remove(p->Children[idx], pred);
    }

    // Nếu con Children[idx] có ít hơn t khóa, xem xét con[idx+1].
    // Nếu con[idx+1] có ít nhất t khóa, tìm phần tử kế tiếp 'succ' của k trong
    // cây con gốc tại con[idx+1]. Thay thế k bằng succ. Đệ quy xóa succ trong con[idx+1]
    else if (p->Children[idx + 1]->n >= p->t)
    {
        int succ = getSuccessor(p, idx);
        p->keys[idx] = succ;
        remove(p->Children[idx + 1], succ);
    }

    // Nếu cả hai con con[idx] và con[idx+1] đều có ít hơn t khóa, hợp nhất k và tất cả con của
    con[idx+1]
    // vào con[idx]
    // Bây giờ con[idx] chứa 2t-1 khóa
    // Giải phóng con[idx+1] và đệ quy xóa k từ con[idx]
    else
    {
        merge(p, idx);
        remove(p->Children[idx], k);
    }
}

```




```
int getPredecessor(NODE* p, int idx)
{
    // Di chuyển đến nút phải nhất cho đến khi đến một nút lá
    NODE* cur = p->Children[idx];
    while (cur->leaf == false)
        cur = cur->Children[cur->n];

    // Trả về khóa cuối cùng của nút lá
    return cur->keys[cur->n - 1];
}

int getSuccessor(NODE* p, int idx)
{
    // Di chuyển đến nút trái nhất bắt đầu từ con[idx+1] cho đến khi đến một nút lá
    NODE* cur = p->Children[idx + 1];
    while (cur->leaf == false)
        cur = cur->Children[0];

    // Trả về khóa đầu tiên của nút lá
    return cur->keys[0];
}
```



```
// Một hàm để mượn một khóa từ Children[idx+1] và đặt
// nó vào con[idx]
void borrowFromNext(NODE* p, int idx)
{
    NODE* child = p->Children[idx];
    NODE* sibling = p->Children[idx + 1];

    // keys[idx] được chèn làm khóa cuối cùng trong Children[idx]
    child->keys[child->n] = p->keys[idx];

    // Con anh em con đầu tiên được chèn làm con cuối cùng
    // vào Children[idx]
    if (child->leaf == false)
        child->Children[(child->n) + 1] = sibling->Children[0];

    // Khóa đầu tiên từ anh em được chèn vào keys[idx]
    // Di chuyển tất cả các khóa trong anh em một bước sau
    for (int i = 1; i < sibling->n; i++)
        sibling->keys[i - 1] = sibling->keys[i];

    // Di chuyển các con trở sau đó
    if (sibling->leaf == false)
    {
        for (int i = 1; i <= sibling->n; i++)
            sibling->Children[i - 1] = sibling->Children[i];
    }

    // Tăng và giảm số lượng khóa của con và cấp cha
    child->n += 1;
    sibling->n -= 1;
}
```



```
// Một hàm để hợp nhất Children[idx] với Children[idx+1]
// Children[idx+1] được giải phóng sau khi hợp nhất
void merge(NODE* p, int idx)
{
    NODE* child = p->Children[idx];
    NODE* sibling = p->Children[idx + 1];

    // Rút một khóa từ nút hiện tại và chèn nó vào (t-1) th
    // vị trí của Children[idx]
    child->keys[p->t - 1] = p->keys[idx];

    // Sao chép các khóa từ Children[idx+1] sang Children[idx] vào cuối
    for (int i = 0; i < sibling->n; ++i)
        child->keys[i + p->t] = sibling->keys[i];

    // Sao chép các con trở từ Children[idx+1] sang Children[idx]
    if (child->leaf == false)
    {
        for (int i = 0; i <= sibling->n; i++)
            child->Children[i + p->t] = sibling->Children[i];
    }

    // Di chuyển tất cả các khóa sau idx trong nút hiện tại một bước trước -
    // để điền vào khoảng trống được tạo ra bằng cách di chuyển khóa[idx] sang Children[idx]
    for (int i = idx + 1; i < p->n; i++)
        p->keys[i - 1] = p->keys[i];

    // Di chuyển các con trở sau (idx+1) trong nút hiện tại một
    // bước trước
    for (int i = idx + 2; i <= p->n; i++)
        p->Children[i - 1] = p->Children[i];

    // Cập nhật số lượng khóa trong nút hiện tại và giảm số lượng con
    child->n += sibling->n + 1;
    p->n--;

    // Giải phóng bộ nhớ của con đang chứa khóa[idx] trong nút hiện tại
    delete(sibling);
}
```



C, Độ phức tạp của thao tác tìm kiếm trên B-Tree

- Độ phức tạp về thời gian của trường hợp xấu nhất: $O(\log n)$
- Độ phức tạp thời gian của trường hợp trung bình: $O(\log n)$
- Độ phức tạp về thời gian của trường hợp tốt nhất: $O(\log n)$
- Độ phức tạp không gian của trường hợp trung bình: $O(n)$
- Độ phức tạp không gian của trường hợp xấu nhất: $O(n)$



D, Ứng dụng của B-Tree

- Dùng trong cơ sở dữ liệu và hệ thống tệp.
- Lưu trữ các khối dữ liệu (phương tiện lưu trữ thứ cấp).
- Lập chỉ số đa mức.





THANKS FOR
ATTENTION!