ĐẠI HỌC BÁCH KHOA HÀ NỘI

TRƯỜNG ĐIỆN – ĐIỆN TỬ

BÁO CÁO BÀI TẬP LỚN

Môn: Cấu trúc dữ liệu và giải thuật

Sinh viên thực hiện

Tên	MSSV	Lớp
Ngô Quang Vinh	20224464	ET – E9 01 K67

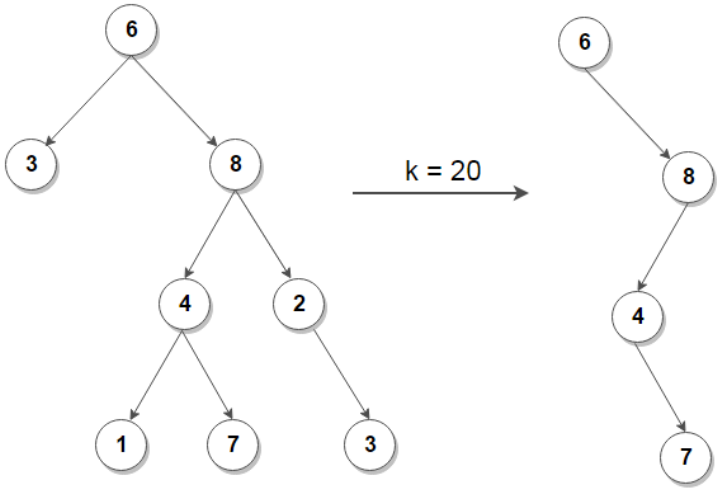
Giáo viên hướng dẫn: Tạ Thị Kim Huệ

Hà Nội, tháng 1 năm 2025

Đề Bài: Cắt bớt một cây nhị phân để loại bỏ các nút nằm trên một đường dẫn có tổng nhỏ hơn `k`

Cho một cây nhị phân và một số k, hãy xóa các nút khỏi cây nằm trên một đường dẫn đầy đủ có tổng nhỏ hơn k. Một đường dẫn đầy đủ trong cây nhị phân được định nghĩa là một đường dẫn từ gốc đến một lá. Tổng của tất cả các nút trên đường dẫn đó được định nghĩa là tổng của đường dẫn đó.

Một nút có thể là một phần của nhiều đường dẫn. Vì vậy, chúng ta chỉ phải xóa nó nếu tất cả các đường dẫn từ nó có tổng nhỏ hơn k. Ví dụ, hãy xem xét cây nhị phân được hiển thị bên trái bên dưới. Chuyển đổi nó thành cây nhị phân được hiển thị bên phải. Lưu ý rằng việc xóa phải được thực hiện theo cách từ dưới lên cho đến khi đường dẫn có tổng gốc-lá lớn hơn hoặc bằng k.



I. Phân tích bài toàn

1. Đầu vào (Input):

- Một **cây nhị phân** T, trong đó mỗi nút chứa một giá trị số nguyên.
- Một số nguyên k, biểu thị ngưỡng tổng của các đường dẫn.

2. Đầu ra (Output):

- Một cây nhị phân đã được cắt tỉa, trong đó:
  - Các nút nằm trên **đường dẫn gốc đến lá** có tổng nhỏ hơn k sẽ bị loại bỏ.
  - Một đường dẫn gốc đến lá được định nghĩa là chuỗi các nút bắt đầu từ gốc cây và kết thúc ở một nút lá (nút không có con).

3. **Phân tích yêu cầu:**

- **Định nghĩa đường dẫn hợp lệ:**
  - Một đường dẫn gốc-lá hợp lệ là đường dẫn có tổng sum ≥ k
- **Loại bỏ nút:**
  - Một nút trong cây cần bị loại bỏ nếu tất cả các đường dẫn gốc-lá đi qua nó có tổng nhỏ hơn k.
  - Việc kiểm tra phải được thực hiện từ **dưới lên trên** (tức là bắt đầu từ các nút lá).

4. **Các ràng buộc:**

- **Tính tổng trên đường dẫn:**
  - Tổng các giá trị từ gốc đến nút hiện tại phải được duy trì để so sánh với k.
- **Xóa nút theo điều kiện:**
  - Nếu cả hai nhánh con của một nút bị xóa và đường dẫn qua nút đó có tổng nhỏ hơn k, thì nút cũng cần bị loại bỏ.
- **Duyệt cây:**
  - Cần duyệt cây theo thứ tự **hậu tự (postorder traversal)** để đảm bảo các nút con được xử lý trước khi quyết định xóa nút cha.

II. Xây dựng thuật toán

1. Giải thích thuật toán

Thuật toán nhằm mục đích **loại bỏ các nút trên cây nhị phân** nếu tất cả các đường dẫn từ gốc đến lá chứa nút đó có **tổng nhỏ hơn kkk**. Thuật toán thực hiện việc này bằng cách **kiểm tra từng đường dẫn từ gốc đến lá** và quyết định giữ hay xóa nút dựa trên tổng các giá trị trên đường dẫn.

Thuật toán hoạt động từ **dưới lên** (từ lá lên gốc) để đảm bảo rằng:

- Các nút con được kiểm tra trước, sau đó mới đến nút cha.
- Một nút chỉ bị xóa nếu **tất cả các nhánh đi qua nút đó không đạt ngưỡng kkk**.

Cách hoạt động được chia thành các bước sau:

Bước 1: Duyệt cây từ lá lên gốc

- Duyệt cây theo thứ tự hậu tự (Postorder Traversal), tức là:
  - Xử lý cây con bên trái.
  - Xử lý cây con bên phải.
  - Sau đó xử lý nút hiện tại.

Duyệt từ dưới lên đảm bảo rằng các nút con được xử lý trước khi quyết định giữ hay xóa nút cha.

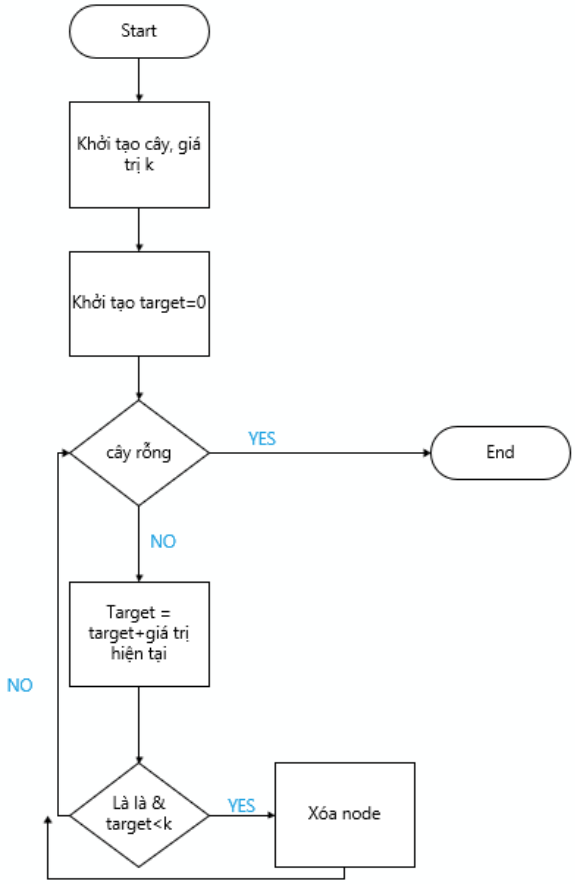
Bước 2: Tính tổng từ gốc đến nút hiện tại

- Mỗi khi duyệt đến một nút, tính tổng giá trị từ gốc đến nút đó (tổng này được tích lũy dọc theo đường đi).

Bước 3: Kiểm tra và xóa nút

- Khi đến một **nút lá**:
  - Nếu tổng giá trị từ gốc đến nút lá **nhỏ hơn k**, nút lá đó sẽ bị xóa.
  - Nếu nút lá bị xóa, nhánh từ cha đến nút lá sẽ bị "cắt".
- Khi đến một **nút không phải lá**:
  - Nếu cả hai nhánh con đều bị xóa (do không đạt ngưỡng), nút đó sẽ trở thành nút lá.
  - Nút này sau đó cũng sẽ được kiểm tra xem có nên giữ lại hay không (dựa trên tổng từ gốc đến nó).

2. Lưu đồ thuật toán



Bắt đầu:

- Khởi tạo cây nhị phân và giá trị ngưỡng k.
- Khởi tạo target=0 để lưu tổng giá trị từ gốc đến nút hiện tại.

Kiểm tra cây rỗng:

Nếu cây rỗng, thuật toán kết thúc (vì không có nút nào để xử lý).

Cập nhật tổng trên đường đi:

- Tính tổng giá trị đường đi từ gốc đến nút hiện tại bằng cách cộng giá trị nút hiện tại vào target.

Kiểm tra nút lá và điều kiện xóa:

- Nếu nút hiện tại là lá (không có con trái và con phải) và target<k
- Xóa nút lá vì đường dẫn từ gốc đến lá này không đạt tổng yêu cầu.

Quay lại và kiểm tra các nút khác:

- Sau khi xử lý, quay lại các nút cha để tiếp tục duyệt và xử lý những đường dẫn còn lại trong cây.

3. Mã giả

```
function isLeaf(node):
    return (node.left == nullptr and node.right == nullptr)
function trunc(curr, k, target):
    if curr == nullptr:
        return
    target = target + curr.data
    // Gọi đệ quy xử lý cây con trái và phải
    trunc(curr.left, k, target)
    trunc(curr.right, k, target)
    // Nếu tổng nhỏ hơn k và là nút lá, xóa nút
    if target < k and isLeaf(curr):
        delete(curr)
        curr = nullptr
function truncate(root, k):
    target = 0
    trunc(root, k, target)
function inorder(root):
```

```
        if root == nullptr:
            return
        inorder(root.left)
        print(root.data)
        inorder(root.right)
    // Main
    root = cây nhị phân
    k = 20
    truncate(root, k)
    inorder(root)
```

4. Giải thích code

a. Hàm isLeaf dùng để kiểm tra nếu một nút là nút lá:

```
bool isLeaf(Node* node) {
   return (node->left == nullptr && node->right == nullptr);
}
```

Một nút lá là nút **không có con trái và con phải**.
Chỉ nút lá mới được kiểm tra tổng để quyết định có xóa hay không.

b. Hàm trunc thực hiện việc duyệt cây và kiểm tra các nút:

```
void trunc(Node* &curr, int k, int target)
{
   if (curr == nullptr) {
      return;
   }
   target = target + (curr->data);

   // Gọi đệ quy xử lý cây con trái và phải
   trunc(curr->left, k, target);
   trunc(curr->right, k, target);

   // Nếu tổng nhỏ hơn k và là nút lá, xóa nút
   if (target < k && isLeaf(curr))
```

```
    {
        delete(curr);
        curr = nullptr;
    }
}
```

Nếu nút hiện tại curr là nullptr, kết thúc (do không có nút để xử lý).

Tính tổng từ gốc đến nút hiện tại bằng cách cộng giá trị của nút hiện tại curr->data vào target.

Gọi đệ quy để xử lý:

- Cây con trái: trunc(curr->left, k, target)
- Cây con phải: trunc(curr->right, k, target)

Sau khi xử lý xong cây con:

- Nếu nút hiện tại là **nút lá** và tổng đường dẫn từ gốc đến nút nhỏ hơn kkk:
    - Xóa nút bằng delete(curr).
    - Gán con trỏ của nút thành nullptr.

c. Hàm chính (truncate)

```
void truncate(Node* &root, int k)
{
    int target = 0;
    trunc(root, k, target);
}
```

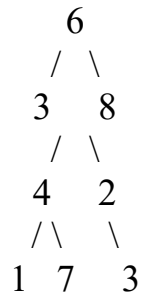
Hàm này gọi hàm trunc với gốc cây ban đầu (root), ngưỡng kkk, và tổng ban đầu target = 0.

d. Duyệt cây để in kết quả

```
void inorder(Node* root)
{
    if (root == nullptr) {
        return;
    }
    inorder(root->left);       // Duyệt cây con trái
    cout << root->data << " "; // In giá trị nút hiện tại
    inorder(root->right);      // Duyệt cây con phải
}
```

Hàm này duyệt theo thứ tự trung tự (Inorder Traversal) để in các giá trị nút còn lại sau khi cắt tỉa.

5. Ví dụ



Ngưỡng k=20:

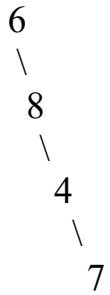
Ta cần giữ lại các nút trên đường dẫn có tổng từ gốc đến lá lớn hơn hoặc bằng k.

Quá trình thực hiện:

Duyệt cây theo hậu tự (postorder):

- **Nút lá 1:**
  - Tổng đường dẫn 6+8+4+1=19<20→ Xóa nút 1.
- **Nút lá 3 (ở nhánh phải):**
  - Tổng đường dẫn 6+8+2+3=19<20→ Xóa nút 3.
- **Nút lá 7:**
  - Tổng đường dẫn 6+8+4+7=25≥20 → Giữ lại nút 7.
- **Nút 4:**
  - Sau khi xử lý các nút con, giữ lại vì nhánh 6+8+4+7≥20
- **Nút 2:**
  - Sau khi xử lý nhánh con, không còn nhánh hợp lệ → Xóa nút 2.
- **Nút 8:**
  - Có nhánh hợp lệ 6+8+4+7≥20 → Giữ lại nút 8.
- **Nút gốc 6:**
  - Có nhánh hợp lệ 6+8+4+7≥20→ Giữ lại nút 6.

⇨ **Cây kết quả**



6. Độ phức tạp và thời gian

a. Độ phức tạp thời gian

Cây nhị phân có tổng số n nút. Với mỗi nút:

- Hàm trunc duyệt qua nó đúng 1 lần (duyệt hậu tự – postorder traversal).

- Hàm inorder duyệt qua nó đúng 1 lần (duyệt trung tự – inorder traversal).

Do đó:

- **Độ phức tạp của hàm trunc** là O(n)
- **Độ phức tạp của hàm inorder** là O(n)

Tổng quát, độ phức tạp thời gian cho toàn bộ thuật toán là:

$$O(n)+O(n)=O(n)$$

b. Độ phức tạp không gian

Bộ nhớ cho cây nhị phân

Cây nhị phân có nnn nút, và mỗi nút chiếm O(1) bộ nhớ. Do đó, bộ nhớ cho cây là O(n).

Bộ nhớ ngăn xếp cho đệ quy

Thuật toán sử dụng đệ quy, nên mỗi lần gọi hàm đệ quy sẽ lưu trạng thái hiện tại vào ngăn xếp. Trong trường hợp xấu nhất (cây là cây nhị phân **một nhánh duy nhất** hoặc **cây không cân bằng**), độ sâu tối đa của đệ quy là O(n) Trong trường hợp tốt nhất (cây là cây nhị phân **cân bằng hoàn toàn**), độ sâu của đệ quy là O(log n).

Tổng hợp

- Trường hợp xấu nhất: O(n) cho ngăn xếp.
- Trường hợp tốt nhất: O(log(n)) cho ngăn xếp.

c. Chạy code