Lời giải [IT003.P21.CTTN.1] Assigment4 (Advance và Basic)

Bảo Quý Định Tân - 24520028

Ngày 15 tháng 4 năm 2025

Mục lục

1 Advance

1.1 khangtd.KetBan

Để thực hiện việc chuyển chỗ bạn x sang bên tay trái bạn y, ta có thể tưởng tượng như đang có một cái doubly linked list xoay vòng (1 bạn đầu nối với n) theo hướng cùng chiều kim đồng hồ là tay phải (hơi ngược lại với đề).

Ví dụ: Ban đầu ta sẽ có vòng n số với số 2 nằm bên tay phải của 1 và 6 nằm bên tay trái của 1.

```
struct Node {
    int id;
    Node* left;
    Node* right;
    Node(int _id) {
        id = _id;
        left = right = nullptr;
    }
};

Node* a[MAX_N];
```

được tạo bằng

```
for (int i = 1; i <= n; i++) {
    a[i] = new Node(i);
}
for (int i = 1; i <= n; i++) {
    a[i]->left = a[i - 1];
    a[i]->right = a[i + 1];
}
a[1]->left = a[n];
a[n]->right = a[1];
```

Ở mỗi truy vấn, ta chỉ đơn giản là chuyển phần tử x sang bên phải y trên linked list của mình.

Xoá phần tử x.

```
a[x]->left->right = a[x]->right;
a[x]->right->left = a[x]->left;
```

Nối x vào bên phải y.

```
a[x]->left = a[y];
a[x]->right = a[y]->right;
```

Và chỉnh lại các kết nối của 2 phần tử bên cạnh vị trí mới của x.

```
a[x]->left->right = a[x];
a[x]->right->left = a[x];
```

Cuối cùng ta in ra lại vòng tròn của mình bắt đầu từ vị trí 1.

```
Node* p = a[1];
for (int i = 1; i <= n; i++) {
    cout << p->id << ' ';
    p = p->right;
}
```

1.2 khangtd.HuffmanCoding

Ta thực hiện **Huffman** bằng cách đếm số lượng của từng loại ký tự:

```
for (char h : s) {
    int x = h;
    cnt[x]++;
}
```

và lấy ra các loại ký tự khác nhau cùng với số lượng của chúng:

```
n = 0;
for (int i = 0; i < 256; i++) {
    if (cnt[i]) {
        n++;
        pq.push({cnt[i], n});
        a[n] = cnt[i];
    }
}
```

Với mỗi loại ký tự, ta sẽ cho nó là một node lá của cây, ta sẽ bắt đầu gộp các node theo thứ tự lấy các cặp node có số lượng ký tự là nhỏ nhất rồi gộp chúng lại, làm vậy lần lượt cho đến khi chỉ còn lại một node:

```
if (pq.size() == 1) {
    n++;
    pq.push({0, n});
}
while (pq.size() > 1) {
    auto it1 = pq.top();
    pq.pop();
    auto it2 = pq.top();
    pq.pop();
    n++;
    adj[n].pb(it1.se);
    adj[n].pb(it2.se);
    pq.push({it1.fi + it2.fi, n});
```

```
}
```

Cuối cùng ta có thể dfs lại một lần trên cây để tính ra kết quả:

```
void dfs(int u, int h = 0) {
    bool haveChild = false;
    for (int v : adj[u]) {
        haveChild = true;
        dfs(v, h + 1);
    }
    if (!haveChild) {
        ans += a[u] * h;
    }
}

dfs(n);
cout << ans;
```

1.3 khangtd.HoaNhac

Với mỗi người i từ 1 đến n, ta sẽ đếm số lượng người j $(1 \leq j < i)$ sao cho hai người này có thể nhìn thấy nhau.

Khi tới vị trí i, ta sẽ tính được một vector lưu các vị trí j sao cho từ j đến $i-1$ thì a_j là max.

Nhờ vào vector này ta có thể dễ dàng tính được số lượng j mà j và i có thể nhìn thấy nhau là số lượng j trong vector sao cho từ $j+1$ đến $i-1$ toàn là các số $\leq a_i$.

Tính từ cuối vector về đầu (vector sắp xếp tăng dần theo vị trí của các j), vị trí j cuối cùng mà i có thể nhìn thấy là vị trí j đầu tiên (tính từ cuối về) mà $a_j > a_i$.

Còn nếu mọi j trong vector đều $\leq a_i$ thì ta lấy mọi j trong vector luôn.

Ta có thể thực hiện việc tìm vị trí này nhanh bằng chặt nhị phân.

Sau khi thực hiện tính số vị trí mà i nhìn thấy được, ta cập nhật vector bằng vị trí i.

```
v.pb(a[1]);
for (int i = 2; i <= n; i++) {
    int low = 0, high = v.size() - 1;
    int pos = 0;
    while (low <= high) {
        int mid = (low + high) >> 1;
        if (v[mid] > a[i]) {
            pos = mid;
            low = mid + 1;
        }
        else {
```

```
            high = mid - 1;
        }
    }
    ans += v.size() - pos;
    while (!v.empty() && a[i] > v.back()) {
        v.pop_back();
    }
    v.pb(a[i]);
}
```

1.4 Tree: levelOrder Traversal - Duyệt cây BST theo chiều rộng

Bài này ta duyệt cây theo thứ tự BFS nên ta có thể sử dụng cấu trúc dữ liệu queue để thực hiện:

```
void levelOrder(Node *root) {
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* curr = q.front();
        q.pop();
        cout << curr->data << ' ';
        if (curr->left) {
            q.push(curr->left);
        }
        if (curr->right) {
            q.push(curr->right);
        }
    }
}
```

2 Basic

2.1 Tree: Hieght of Tree

Để tính độ cao của một cây, ta sẽ dfs qua cây đó, mỗi hàm con sẽ trả ra độ cao của một gốc cây con.

Vậy với mỗi node, ta sẽ lấy **max** của độ cao của các gốc cây con và cộng thêm 1 là ra độ cao của cây hiện tại.

```
int height(Node* root) {
    int mx = 0;
    if (root->left) {
```

```
        mx = max(mx, height(root->left));
    }
    if (root->right) {
        mx = max(mx, height(root->right));
    }
    return mx + 1;
}
```

2.2 Binary Search Tree: Insert

Để thực hiện thêm node vào cây nhị phân, ta sẽ đi lần lượt trên cây nhị phân theo thứ tự:

- Nếu giá trị thêm vào bé hơn hoặc bằng giá trị của node hiện tại, ta đi sang trái.
- Nếu lớn hơn thì ta đi sang phải.

Ta đi cho tới khi nào không đi được nữa thì ta dừng và thêm vào đỉnh mới là một node lá:

```
Node * insert(Node * root, int data) {
    if (!root) {
        return root = new Node(data);
    }

    if (data < root->data) {
        root->left = insert(root->left, data);
    }
    else {
        root->right = insert(root->right, data);
    }

    return root;
}
```

2.3 Binary Search Tree: Insert (không dùng đệ quy)

Để thực hiện việc trên mà không sử dụng đệ quy, ta nhận thấy rằng việc đi xuống các nhánh chỉ có một đường đi duy nhất thôi, nên ta chỉ việc đi theo quy luật trên và khi đến đích thì thêm node mới là node lá:

```
Node * insert(Node * root, int data) {
    if (!root) {
        return root = new Node(data);
```

```
    }

    Node* p = root;
    while (p) {
        if (data < p->data) {
            if (p->left) {
                p = p->left;
            }
            else {
                p->left = new Node(data);
                break;
            }
        }
        else {
            if (p->right) {
                p = p->right;
            }
            else {
                p->right = new Node(data);
                break;
            }
        }
    }

    return root;
}
```

2.4 Tree: levelOrder Traversal - Duyệt cây BST theo chiều rộng

Bài này ta duyệt cây theo thứ tự BFS nên ta có thể sử dụng cấu trúc dữ liệu queue để thực hiện:

```
void levelOrder(Node *root) {
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* curr = q.front();
        q.pop();
        cout << curr->data << ' ';
        if (curr->left) {
            q.push(curr->left);
        }
        if (curr->right) {
            q.push(curr->right);
        }
    }
```

```
}
```

2.5 Tree: Inorder Traversal (LNR) - Duyệt cây BST theo LNR

Để duyệt theo thứ tự LNR, với mỗi node, ta sẽ đi sang nhánh bên trái trước rồi xuất ra giá trị hiện tại, rồi đi sang nhánh bên phải:

```
void inOrder(Node *root) {
    if (!root) return;
    inOrder(root->left);
    cout << root->data << ' ';
    inOrder(root->right);
}
```

2.6 Tree: Inorder Traversal (LNR) II - Duyệt cây BST theo LNR không đệ quy

Để thực hiện việc trên mà không sử dụng đệ quy, ta sẽ sử dụng stack để mô phỏng lại quá trình đệ quy.

Ta nhận thấy rằng sau mỗi lần đi hết nhánh trái, ta sẽ xuất ra số hiện tại. Nhưng ta đang khử đệ quy nên ta không thể truy cập lại số hiện tại thuận tiện được như đệ quy khi đã đi xuống các nhánh con.

Để giải quyết vấn đề này, ta sẽ tạo thêm một stack *outputBuffer*, trước khi sang nhánh trái thì ta sẽ bỏ đỉnh hiện tại vào *outputBuffer* để sau này khi sang nhánh phải ta sẽ xuất ra số của đỉnh hiện tại từ nó.

Khi đi dfs bằng stack, ta cũng lưu lại thêm một biến nữa để đánh dấu rằng ta đang xuống ở nhánh trái hay nhánh phải để biến mà xuất số từ *outputBuffer*.

```
void inOrder(Node *root) {
    stack<pair<Node*, int>> st;
    st.push({root, -1});
    stack<int> outputBuffer;
    outputBuffer.push(-1);

    while (!st.empty()) {
        Node* curr = st.top().first;
        int t = st.top().second;

        st.pop();

        if (~t) {
```

```
            cout << outputBuffer.top() << ' ';
            outputBuffer.pop();
        }

        if (!curr) continue;

        if (curr->right) {
        }
        st.push({curr->right, 0});

        if (curr->left) {
            st.push({curr->left, -1});
        }

        outputBuffer.push(curr->data);
    }
}
```

2.7 Tree: Postorder Traversal (LRN) - Duyệt cây BST theo LRN

Để duyệt theo thứ tự LRN, ta sẽ đi sang nhánh trái trước, rồi mới sang nhánh phải, xong cuối cùng mới xuất ra số hiện tại.

```
void postOrder(Node *root) {
    if (!root) return;
    postOrder(root->left);
    postOrder(root->right);
    cout << root->data << ' ';
}
```

2.8 Tree: Postorder Traversal (LRN) II - Duyệt cây BST theo LRN không đệ quy

Ta nhận thấy việc đi sang nhánh trái và nhánh phải trước xong mới xuất ra số của node hiện tại - thứ tự này giống như xuất một cái stack và quả đúng thật là như vậy.

Nên để khử đệ quy, ta sử dụng một stack để mô phỏng việc dfs, một stack để lưu lại các số mình sẽ xuất.

```
void inOrder(Node *root) {
    stack<Node*> st;
    st.push(root);
    stack<int> output;
```

```
5
6       while (!st.empty()) {
7           Node* curr = st.top();
8           st.pop();
9
10          if (!curr) continue;
11
12          output.push(curr->data);
13          st.push(curr->left);
14          st.push(curr->right);
15      }
16
17      while (!output.empty()) {
18          cout << output.top() << ' ';
19          output.pop();
20      }
21  }
```

2.9 Binary Search Tree: Nút chung gần nhất

Bài này ta đã từng làm qua từ trước rồi.

2.10 Tree: Top view

Ta nhận thấy thứ tự cây đề bài cho giống như một thứ tự cho của một cây nhị phân thông thường nên ta sẽ thực hiện việc thêm node vào cây nhị phân như ta thường làm.

Đồng thời ta sẽ lưu lại toạ độ và độ cao của node mà mình đi tới.

Độ cao thì là độ cao của cây bình thường hay làm.

Toạ độ ở đây, tưởng tượng ta đang có trục hoành Ox, với gốc được đặt tại gốc của cây nhị phân của ta:

- Khi ta đi sang nhánh trái trên cây, đồng nghĩa với việc ta sẽ đi về phía bên trái một bước trên trục hoành (-1).
- Tương tự khi đi sang nhánh phải, tương tự với việc ta đi về bên phải một bước trên trục hoành $(+1)$.

Với toạ độ trên trục hoành, ta sẽ xác định được những node nào ở cùng một hàng dọc thẳng (tưởng tượng giống như nằm trên một đường thẳng song song với trục tung) để lấy ra node mà không bị che.

Với độ cao, ta sẽ xác định được trên một hàng, node nào sẽ là node không bị che. Nếu có nhiều node không bị che thì ta sẽ lấy node có giá trị lớn nhất.

```cpp
struct Node {
    int data;
    int pos;
    Node* left;
    Node* right;
    Node (int _data) {
        data = _data;
        left = nullptr;
        right = nullptr;
    }
};

int n;
const int MAX_N = 2e6 + 10;
const int base = 1e6;
pii a[MAX_N];

void insertNode(Node* &root, int data, int pos = 0, int dist = 1) {
    if (!root) {
        int i = pos + base;
        if (a[i].se == dist) {
            maximize(a[i].fi, data);
        }
        else {
            if (!a[i].fi || a[i].se > dist) {
                a[i] = {data, dist};
            }
        }
        root = new Node(data);
        return;
    }

    if (data <= root->data) {
        insertNode(root->left, data, pos - 1, dist + 1);
    }
    else {
        insertNode(root->right, data, pos + 1, dist + 1);
    }
}

Node* root = nullptr;

for (int i = 1; i <= n; i++) {
    int x;
    cin >> x;
    insertNode(root, x);
}
```

```
48
49 vector<int> ansList;
50 for (int i = 0; i < MAX_N; i++) {
51     if (a[i].fi) {
52         ansList.pb(a[i].fi);
53     }
54 }
55 sort(all(ansList));
56
57 for (int& x : ansList) {
58     cout << x << ' ';
59 }
```