ĐẠI HỌC ĐÀ NẴNG

TRƯỜNG ĐẠI HỌC BÁCH KHOA

KHOA KHOA HỌC CÔNG NGHỆ TIÊN TIẾN

BÁO CÁO
DỰ ÁN CUỐI KỲ

BỘ MÔN:

CẤU TRÚC DỮ LIỆU NÂNG CAO VÀ THUẬT TOÁN

NHÓM 5

Mục Lục

Phân chia công việc

1. Danh sách thành viên

Lê Trần Duy Tân	123210174
Lê Minh Mạnh	123210063
Nguyễn Thái Pháp	123210067

2. Phân chia công việc

Bài toán 1:

Lê Trần Duy Tân: Cài đặt chung

- Phụ trách xây dựng cấu trúc của cây BST, cài đặt các hàm cơ bản như tạo nút, chèn, tìm kiếm, xóa.
- Kiểm tra tính đúng đắn của các hàm và viết mã giả cho chương trình chính.

Lê Minh Mạnh: Tạo Test Cases và Kiểm thử

- Tạo test cases cho các trường hợp thông thường, đặc biệt và lớn.
- Thực hiện kiểm thử bằng cách chạy các test cases và kiểm tra xem kết quả thực tế có khớp với kết quả mong đợi từ mã giả hay không.

Nguyễn Thái Pháp: Tối ưu hóa và Báo cáo

- Nếu cần, tối ưu hóa mã nguồn để cải thiện hiệu suất.
- Viết báo cáo mô tả cách mà cây BST được cài đặt, các vấn đề đã gặp và giải quyết, cũng như kết quả kiểm thử.

Bài toán 2:

Nguyễn Thái Pháp: Cài đặt chung

- Phụ trách cài đặt hàm DFS cho việc duyệt đồ thị sâu trước trên ma trận.
- Đảm bảo rằng việc đánh dấu và xác định các thành phần liên thông được thực hiện đúng.

Lê Trần Duy Tân: Tạo Test Cases và Kiểm thử

- Tạo test cases cho các trường hợp thông thường và đặc biệt của ma trận.
- Thực hiện kiểm thử bằng cách chạy các test cases và kiểm tra xem kết quả thực tế có khớp với kết quả mong đợi từ mã giả hay không.

Lê Minh Mạnh: Tối ưu hóa và Báo cáo

- Nếu cần, tối ưu hóa mã nguồn để cải thiện hiệu suất.
- Viết báo cáo mô tả cách mà thuật toán DFS được cài đặt, các vấn đề đã gặp và giải quyết, cũng như kết quả kiểm thử.

Bài 1:

Cho một ma trận với (x,y) là vị trí/ô chứa phần tử nằm trên hàng x cột y (x, y ≥ 0). Một đường để di chuyển từ một vị trí (x, y) đến (x',y') theo một quy tắc cho trước và mỗi lần di chuyển chỉ có thể sang các ô kề cạnh (cùng hàng hoặc cùng cột), thì ta gọi (x,y) và (x', y') liên thông với nhau. Vùng liên thông là một tập tất cả các vị trí mà từ một vị trí trong tập có thể đến bất kỳ vị trí nào cùng thuộc tập đó. Viết chương trình thực hiện các yêu cầu sau:

a) Nhập vào ma trận A (MxN) từ file văn bản có tên inpLT.txt với cấu trúc như sau:
- Dòng đầu 2 số M, N (M, N ≤ 100)
- M dòng tiếp theo, mỗi dòng gồm N ký tự 'O' hoặc 'X'

b) Xác định các vùng liên thông cho các vị trí chứa các ký tự X và đánh số thứ tự cho các vùng liên thông.

c) Xuất ra file văn bản có tên outLT.txt số vùng liên thông chỉ chứa 1 ô và ma trận số nguyên không âm B (MxN) theo quy tắc B[i][j] = 0 nếu A[i][j]= 'O' và B[i][j] = số thứ tự của vùng liên thông chứa vị trí (i,j) nếu A[i][j]= 'X'.

Thuật toán DFS:

Thuật toán DFS (Depth-First Search) là một thuật toán duyệt đồ thị sử dụng cách tiếp cận theo chiều sâu. Nó bắt đầu từ một đỉnh xuất phát, sau đó đi theo một nhánh cụ thể cho đến khi không thể đi tiếp được nữa. Sau đó, nó quay lại và thử một nhánh khác.

Mô tả cơ bản *:*

Khởi tạo:

- Chọn một đỉnh xuất phát.
- Đánh dấu đỉnh xuất phát là đã thăm.

Duyệt đỉnh hiện tại:

- Xử lý đỉnh hiện tại (ví dụ: in ra, lưu giữ, xử lý).

Duyệt qua các đỉnh kề chưa thăm:

- Duyệt qua tất cả các đỉnh kề của đỉnh hiện tại.
- Nếu đỉnh kề chưa được thăm, gọi đệ quy DFS trên đỉnh kề đó.

Quay lại:

- Quay lại đỉnh trước đó và kiểm tra xem còn nhánh chưa thử không.

Sơ lược các bước chính phải làm để giải quyết vấn đề:

Bước 1: Khởi tạo và Đọc dữ liệu:

- Mở tệp đầu vào "inpLT.txt".
- Đọc giá trị M và N từ tệp.
- Khai báo và khởi tạo mảng A và B.

Bước 2: Định nghĩa hàm testxy:

- Kiểm tra xem một vị trí (x, y) có nằm trong ranh giới của ma trận không.

Bước 3: Định nghĩa hàm dfs:

- Hàm sử dụng thuật toán đệ quy để duyệt các ô liên thông có giá trị 'X'.
- Đánh dấu và đếm các ô thuộc vùng liên thông.

Bước 4: Chương trình chính:

- Duyệt qua từng ô của ma trận.
- Nếu gặp một ô chưa được đánh dấu và có giá trị 'X', thì gọi hàm dfs.
- Đếm số lượng vùng liên thông chỉ có 1 ô.

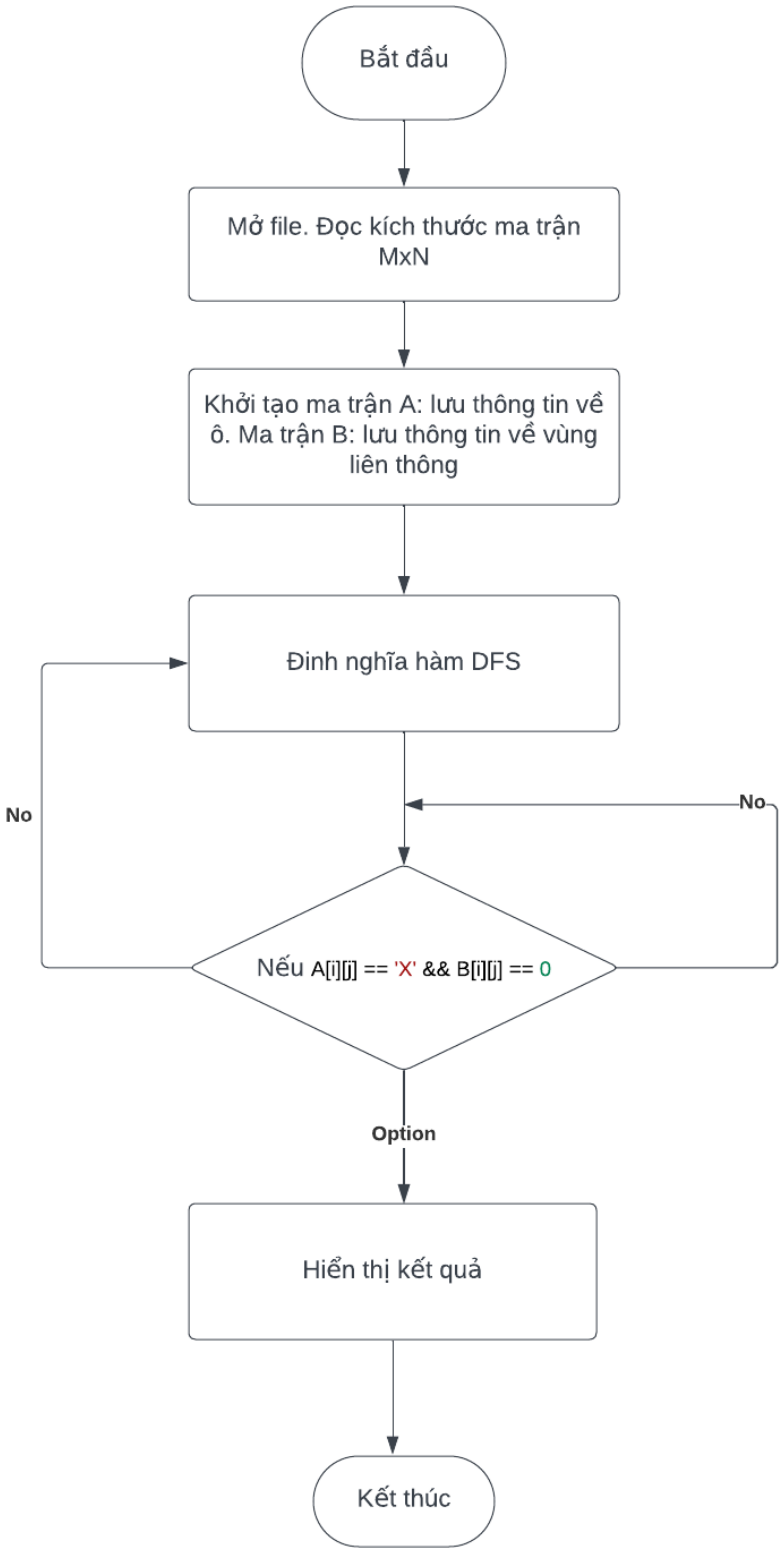
Bước 5: Xuất kết quả:

- Mở tệp đầu ra "outLT.txt".
- Ghi số lượng vùng liên thông chỉ có 1 ô.
- Ghi ma trận kết quả.

Bước 5: Đóng tệp và Kết thúc:

- Đóng tệp đầu vào và tệp đầu ra.
- Kết thúc chương trình.

Lưu đồ thuật toán



Mã giả

1. **Hàm kiểm tra vị trí:**

   Hàm testxy(x, y, M, N):

   Nếu x >= 0: *// Kiểm tra xem x có lớn hơn hoặc bằng 0 không*

   Nếu x < M: *// Kiểm tra xem x có nhỏ hơn M không*

   Nếu y >= 0: *// Kiểm tra y có lớn hơn hoặc bằng 0 không*

   Nếu y < N: *// Kiểm tra y có nhỏ hơn N không*

   Trả về true *// Trả về true nếu tất cả các điều kiện đều đúng*

   // Trả về false nếu bất kỳ điều kiện nào không thỏa mãn

   Trả về false

2. **Hàm DFS (Depth First Search Tree)**

   Hàm DFS(x, y, mark, M, N, A, B, count):

   // Đánh dấu ô hiện tại là đã thăm và tăng biến đếm

   B[x][y] = mark

   count++

   // Các hướng di chuyển (lên, xuống, trái, phải)

   dx[] = {-1, 1, 0, 0}

   dy[] = {0, 0, -1, 1}

   // Duyệt qua các hướng

   for i từ 0 đến 3:

   newX = x + dx[i]

   newY = y + dy[i]

   // Kiểm tra xem ô mới có hợp lệ không và chưa thăm

   if testxy(newX, newY, M, N) và A[newX][newY] = 'X' và B[newX][newY] = 0:

   // Gọi đệ quy DFS cho ô mới

   DFS(newX, newY, mark, M, N, A, B, count)

Quá trình tạo trường hợp để kiểm tra thuật toán (test case)

1. **Quá trình tạo và kiểm tra test case**

   Bước 1: Xác định Đặc điểm Quan trọng

   Xác định các đặc điểm quan trọng của thuật toán cây liên thông, chẳng hạn như:

   - Ma trận đầu vào có kích thước lớn nhỏ như thế nào.

- Ma trận có bao nhiêu vùng liên thông.
- Có bao nhiêu ô trong mỗi vùng liên thông.
- Có bao nhiêu vùng liên thông chỉ có một ô.

Bước 2: Tạo Test Case

Dựa trên các đặc điểm quan trọng đã xác định, tạo ra các test case đa dạng để kiểm tra các khía cạnh của thuật toán. Ví dụ:

1. Trường hợp thông thường:
   - Ma trận với nhiều vùng liên thông có nhiều ô.
   - Có một số vùng liên thông chỉ có một ô.
2. Trường hợp đặc biệt:
   - Ma trận có kích thước nhỏ (3x3 hoặc 4x4).
   - Ma trận chỉ chứa 'O' hoặc chỉ chứa 'X'.
   - Ma trận không chứa 'X', hoặc không chứa 'O'.
3. Trường hợp lớn:
   - Ma trận có kích thước lớn (10x10 hoặc 20x20).
   - Ma trận có nhiều vùng liên thông và mỗi vùng có nhiều ô.

Bước 3: Kết quả Mong đợi

Cho mỗi test case, đặt ra kết quả mong đợi dựa trên phân tích của bạn về đầu vào. Xác định vùng liên thông, số lượng vùng liên thông chỉ có một ô, và các giá trị B[i][j] mong đợi.

Bước 4: Chạy Chương trình và Kiểm tra Kết quả

Chạy chương trình với từng test case và kiểm tra kết quả được tạo ra bởi chương trình. So sánh kết quả thực tế với kết quả mong đợi. Nếu chương trình trả về kết quả đúng cho tất cả các test case, có thể kết luận rằng thuật toán hoạt động chính xác.

Bước 5: Điều Chỉnh và Lặp lại (nếu cần)

Nếu có bất kỳ lỗi hoặc không đồng thuận, điều chỉnh chương trình và test case và thực hiện lại quy trình kiểm thử.

2. Kết quả thu được

Test case 1:
Trường hợp thông thường

inpLT.txt	**outLT.txt**
3 3 X O X X X O O O X	2 1 0 2 1 1 0 0 0 3

Test case 2:
Trường hợp không có vùng liên thông 1 ô nào

inpLT.txt	outLT.txt
5 5 X O X X O X X O O O O O X O X X X X O X	0 1 0 2 2 0 1 1 0 0 0 0 0 3 0 4 3 3 3 0 4 0 0 0 0 0

Test case 3:

Trường hợp chỉ có 'X' hoặc 'O'

inpLT.txt	outLT.txt
4 4 X X X X X X X X X X X X X X X X	0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Test case 4:

Trường hợp ma trận kích thước lơn

inpLT.txt	outLT.txt
10 10 X O X O X X O X O O X O X X X X X X X O O X X X X O X X X O O X X X X X X X X O O O O X X X X X X X O X X X X X X X X X O O O O X O X O X X X X X X X X X X X X O X X X X X X X X X O X X X O X X X X X	0 1 0 2 0 2 2 0 2 0 0 1 0 2 2 2 2 2 2 2 0 0 2 2 2 2 0 2 2 2 0 0 2 2 2 2 2 2 2 2 0 0 0 0 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 2 2 0 0 0 0 2 0 2 0 2 2 2 2 2 2 2 2 2 2 2 2 0 2 2 2 2 2 2 2 2 2 0 2 2 2 0 2 2 2 2 2

Độ phức tạp của bài toán:

Độ phức tạp của bài toán vùng liên thông thường được đánh giá dựa trên kích thước của ma trận, tức là số ô trong ma trận. Gọi M là số hàng và N là số cột của ma trận.

Trong trường hợp tệ nhất, nếu mỗi ô đều thuộc một vùng liên thông khác nhau, ta cần kiểm tra và đánh số tất cả M×N ô. Do đó, độ phức tạp làO(M×N).

Tuy nhiên, trong trường hợp tốt nhất, nếu chỉ có một vùng liên thông duy nhất, ta chỉ cần kiểm tra và đánh số một lần. Do đó, độ phức tạp có thể là O(1).

Trong trường hợp trung bình, khi có nhiều vùng liên thông, độ phức tạp có thể được mô tả là O(K×M×N), trong đó K là số vùng liên thông.

Tóm lại, độ phức tạp của bài toán vùng liên thông phụ thuộc vào cách mà các vùng liên thông được phân bố trong ma trận và có thể được mô tả trong phạm vi từ O(1) đến O(M×N).

Mã nguồn:

```
#include <iostream>

#include <fstream>

#include <queue>

using namespace std;

struct position {
    int x, y;
};


bool testxy(int x, int y, int M, int N) {
    return (x >= 0 && x < M && y >= 0 && y < N);
}


void dfs(int x, int y, int mark, int M, int N, char A[100][100], int
B[100][100], int& count) {
    B[x][y] = mark;
    count++;


    int dx[] = {-1, 1, 0, 0};
    int dy[] = {0, 0, -1, 1};

    for (int i = 0; i < 4; ++i) {
        int newX = x + dx[i];
        int newY = y + dy[i];
```

```
        if (testxy(newX, newY, M, N) && A[newX][newY] == 'X' &&
B[newX][newY] == 0) {
            dfs(newX, newY, mark, M, N, A, B, count);
        }
    }
}

int main() {
    ifstream inputFile("inpLT.txt");
    ofstream outputFile("outLT.txt");

    int M, N;
    inputFile >> M >> N;

    char A[100][100];
    int B[100][100] = {0};


    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            inputFile >> A[i][j];
        }
    }

    int countlt = 0;
    int dem=0;

    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            if (A[i][j] == 'X' && B[i][j] == 0) {
                int count = 0;
                dfs(i, j, ++countlt, M, N, A, B,count);
                if (count == 1) {
                    dem++;
                }
            }
        }
    }

    outputFile << dem << endl;
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < N; ++j) {
            outputFile << B[i][j] << " ";
        }
        outputFile << endl;
    }
```

```
    inputFile.close();
    outputFile.close();
    return 0;
}
```

Bài 2:

Viết chương trình dùng cây nhị phân tìm kiếm (BST) để thực hiện:

a) Nhập vào từ file văn bản có tên inpLN.txt với cấu trúc như sau:
   - Dòng đầu gồm một số N ($N \leq 105$)
   - Dòng tiếp theo là một mảng A gồm N phần tử, mỗi phần tử là một số nguyên không âm không vượt quá 109.
   - Dòng thứ ba gồm một số M ($M \leq 103$)
   - M dòng tiếp theo mỗi dòng 1 số nguyên K ($K \leq 109$).

b) Xuất ra file đầu ra có tên outLN.txt M dòng tương ứng với M dòng cuối của file đầu vào, mỗi dòng là số lớn nhất trên cây không vượt quá K.

Thuật toán BST (Binary Search Tree)

Thuật toán Binary Search Tree (BST) là một cấu trúc dữ liệu cây nhị phân đặc biệt, được thiết kế để hỗ trợ việc tìm kiếm và sắp xếp dữ liệu một cách hiệu quả. Dưới đây là chi tiết về các thao tác cơ bản và độ phức tạp của BST:

Cấu Trúc BST:

Mỗi nút trong BST có hai con, một con bên trái và một con bên phải. Nút bên trái có giá trị nhỏ hơn nút cha, và nút bên phải có giá trị lớn hơn nút cha.

Plaintext

```
struct Node {
    int key;
    Node* left;
    Node* right;
};
```

Sơ lược các bước phải làm để giải quyết vấn đề:

Bước 1: Định nghĩa cấu trúc Node cho cây nhị phân tìm kiếm:

- Mỗi Node có thông tin (data), con trái (left), và con phải (right).

Bước 2: Viết hàm để chèn một phần tử vào cây BST:

- Nếu cây rỗng, tạo một Node mới.
- Nếu giá trị cần chèn nhỏ hơn giá trị của Node hiện tại, chèn vào con trái.

- Nếu giá trị cần chèn lớn hơn giá trị của Node hiện tại, chèn vào con phải.
- Lặp lại quá trình trên cho đến khi đạt đến một Node rỗng.

Bước 3: Viết hàm để tìm phần tử lớn nhất không vượt quá một giá trị K:

- Bắt đầu từ gốc của cây, di chuyển xuống cây theo chiều con phải nếu giá trị của Node nhỏ hơn hoặc bằng K, ngược lại di chuyển xuống cây theo chiều con trái.
- Lưu giữ giá trị của Node khi di chuyển xuống cây theo chiều con phải, vì giá trị này là giá trị lớn nhất không vượt quá K.

Bước 4: Đọc dữ liệu từ file và xây dựng cây BST:

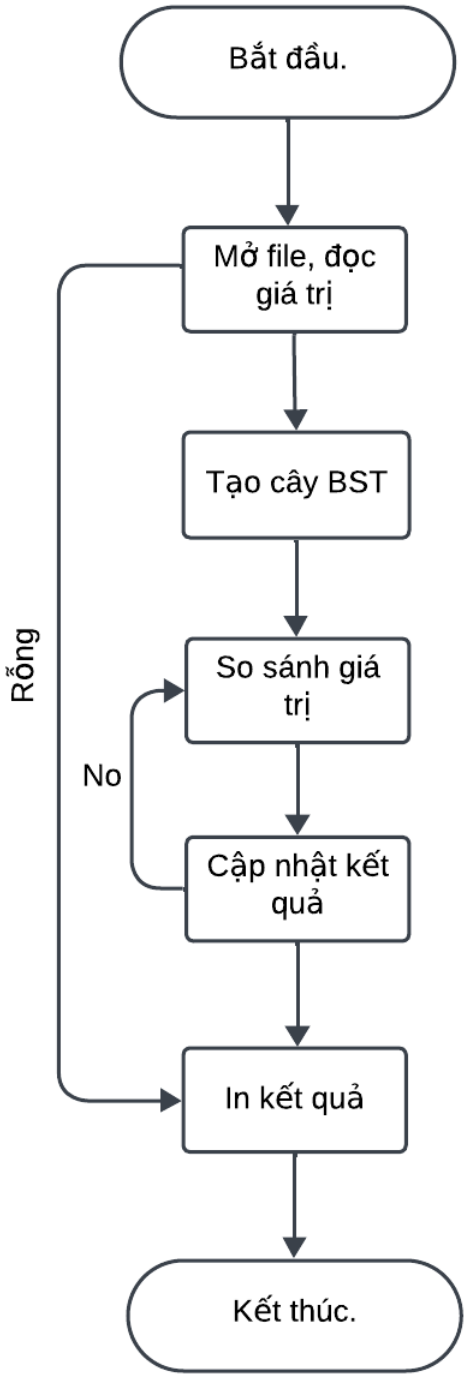
- Mở file để đọc.
- Đọc số phần tử và mảng A.
- Tạo cây BST từ mảng A.

Bước 5: Đọc số M từ file và xử lý từng số K:

- Đọc số M.
- Đọc và xử lý mỗi số K:
- Tìm giá trị lớn nhất không vượt quá K từ cây BST.
- Ghi kết quả vào file outLN.txt.

Bước 6: Ghi kết quả vào file outLN.txt.

Lưu đồ thuật toán:



Mã giả

```
// Khai báo cấu trúc Node cho cây nhị phân tìm kiếm
struct nutbst {
    int data;
```

```
    nutbst* left;
    nutbst* right;
};

// Hàm tạo một Node mới với giá trị cho trước
nutbst* taonut(int value) {
    nutbst* newnut = new nutbst;
    newnut->data = value;
    newnut->left = newnut->right = nullptr;
    return newnut;
}

// Hàm chèn một giá trị vào cây BST
nutbst* insert(nutbst* root, int value) {
    // Nếu cây rỗng, tạo một Node mới
    if (root == nullptr) {
        return taonut(value);
    }

    // Nếu giá trị cần chèn nhỏ hơn giá trị của Node hiện tại, chèn vào con trái
    if (value < root->data) {
        root->left = insert(root->left, value);
    }
    // Nếu giá trị cần chèn lớn hơn giá trị của Node hiện tại, chèn vào con phải
    else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    return root;
}
```

Quá trình tạo trường hợp để kiểm tra thuật toán (test case)

1. **Quá trình tạo và kiểm tra test case**

Bước 1: Xác định Đặc điểm Quan trọng

- Kích thước của cây BST: Cây BST có thể có kích thước lớn nhỏ khác nhau.
- Giá trị của các nút trong cây BST: Các giá trị của nút có thể là số nguyên không âm không vượt quá 10^9.
- Số lượng truy vấn tìm giá trị lớn nhất: Có thể có một hoặc nhiều truy vấn tìm giá trị lớn nhất.
- Giá trị K trong các truy vấn: Các giá trị K có thể thay đổi và không vượt quá 10^9.

Bước 2: Tạo Test Case

Trường hợp thông thường:

- Tạo cây BST với kích thước 10-20 nút.
- Thực hiện một hoặc nhiều truy vấn tìm giá trị lớn nhất với các giá trị K khác nhau.

Trường hợp đặc biệt:

- Tạo cây BST với kích thước nhỏ (3-5 nút) để kiểm tra trường hợp cơ bản.
- Tạo cây BST chỉ chứa các giá trị nhỏ hơn giá trị K trong các truy vấn.
- Tạo cây BST chỉ chứa các giá trị lớn hơn giá trị K trong các truy vấn.

Trường hợp lớn:

- Tạo cây BST với kích thước lớn (100-1000 nút).
- Thực hiện nhiều truy vấn với các giá trị K ngẫu nhiên.

Bước 3: Kết quả Mong đợi

Đối với trường hợp thông thường:

- Xác định giá trị lớn nhất trên cây BST cho mỗi truy vấn với giá trị K.

Đối với trường hợp đặc biệt:

- Xác định giá trị lớn nhất trên cây BST khi cây có kích thước nhỏ.
- Xác định giá trị lớn nhất khi cây chỉ chứa các giá trị nhỏ hơn K hoặc lớn hơn K.

Đối với trường hợp lớn:

- Xác định giá trị lớn nhất trên cây BST với nhiều giá trị K ngẫu nhiên.

Bước 4: Chạy Chương trình và Kiểm tra Kết quả

- Chạy chương trình với từng test case và kiểm tra kết quả được tạo ra.
- So sánh kết quả thực tế với kết quả mong đợi.

Bước 5: Điều Chỉnh và Lặp lại (nếu cần)

- Nếu có lỗi hoặc không đồng thuận, điều chỉnh chương trình và test case.
- Lặp lại quy trình kiểm thử để đảm bảo rằng chương trình hoạt động chính xác với mọi trường hợp

2. Kết quả thu được

Test case 1:

Trường hợp thông thường

inpLN.txt	outLN.txt
10 20 15 25 10 18 22 30 5 12 27 2 20 25	18 22

Test case 2:

Trường hợp đặc biệt

inpLN.txt	outLN.txt
7 10 5 15 3 7 12 20 3 5 10 20	3 7 15

Test case 3:

Trường hợp lớn:

inpLN.txt	outLN.txt
20 45 20 80 10 30 60 90 5 15 25 35 55 70 85 95 3 8 12 28 42 5 20 50 80 10 90	15 45 70 8 85

Độ phức tạp của bài toán:

Độ phức tạp của bài toán tìm giá trị lớn nhất không vượt quá một giá trị K trong cây nhị phân tìm kiếm (BST) là O(log N)

Trong đó N là số lượng nút trong cây. Điều này phản ánh việc cây BST được cân bằng, giúp giảm độ phức tạp so với trường hợp tồi nhất O(N) của một cây không cân bằng.

Nếu cây BST không cân bằng, độ phức tạp có thể lên đến O(N). Điều này xảy ra khi cây được xây dựng theo một dạng đường dọc.

Vì vậy, trong trường hợp tồi nhất, độ phức tạp là O(N), nhưng trong trường hợp trung bình và trường hợp tốt nhất, độ phức tạp là O(log N).

Mã nguồn

```
#include <iostream>
#include <fstream>

using namespace std;

struct nutbst {
    int data;
    nutbst* left;
    nutbst* right;
};

nutbst* taonut(int value) {
    nutbst* newnut = new nutbst;
    newnut->data = value;
    newnut->left = newnut->right = nullptr;
    return newnut;
}

nutbst* insert(nutbst* root, int value) {
```

```
    if (root == nullptr) {
        return taonut(value);
    }

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

    return root;
}

int Max(nutbst* root, int k) {
    int result = k;

    while (root != nullptr) {
        if (root->data < k) {
            result = root->data;
            root = root->right;
        } else {
            root = root->left;
        }
    }

    return result;
}

int main() {
    ifstream inputFile("inpLN.txt");
    ofstream outputFile("outLN.txt");

    int N;
    inputFile >> N;

    nutbst* root = nullptr;

    for (int i = 0; i < N; ++i) {
        int value;
        inputFile >> value;
        root = insert(root, value);
    }

    int M;
    inputFile >> M;
```

```
    for (int i = 0; i < M; ++i) {
        int K;
        inputFile >> K;
        int result = Max(root, K);
        outputFile << result << endl;
    }

    inputFile.close();
    outputFile.close();

    return 0;
}
```