**TRƯỜNG ĐẠI HỌC BÁCH KHOA – ĐẠI HỌC ĐÀ NẴNG**

**KHOA CÔNG NGHỆ TIÊN TIẾN**

****

**BÁO CÁO**

**CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN**

**Sinh viên thực hiện: LÊ VĂN HIỀN**

**NGUYỄN LÊ THÀNH NHÂN**

**NGUYỄN TRUNG NGHĨA**

***Đà Nẵng, ngày 12/12/2023***

1. **PHÂN CHIA CÔNG VIỆC**

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Họ và tên** | **Nội dung công việc** | **Ghi chú** |
| 1 | Nguyễn Trung Nghĩa | Bài 1 | + Giải thích và lựa chọn thuật toán sử dụng  + Phân tích các bước phải làm  + Vẽ Lưu đồ thuật toán cho cả bài 1 và bài 2  + Tạo test case cho bài 1 |
| 2 | Lê Văn Hiền | Bài 2 | + Giải thích và lựa chọn thuật toán sử dụng  + Phân tích các bước phải làm  + Tạo test case cho bài 2 |
| 3 | Nguyễn Lê Thành Nhân | Báo cáo | + Làm báo cáo , viết mã giả, phân tích độ phức tạp của thuật toán |

1. **GIẢI THÍCH CÁC THUẬT TOÁN VÀ PHÂN TÍCH**
2. **Bài 1. Mê Cung**
3. **Gỉai thích thuật toán:**

**Breadth First Search (BFS)** là một thuật toán tìm kiếm theo chiều rộng trong đồ thị hoặc cấu trúc dữ liệu cây. Trong trường hợp của bài toán mê cung, BFS giúp chúng ta tìm kiếm đường đi từ một điểm xuất phát đến một điểm đích sao cho đường đi là ngắn nhất.

1. **Phân tích các bước chính phải làm để giải quyết vấn đề:**

**⮚ Khởi Tạo Hàng Đợi và Mảng Đánh Dấu:**

⚫ Hàng đợi (queue) được sử dụng để lưu trữ các ô chưa được kiểm tra.

⚫ Mảng đánh dấu (visited) được sử dụng để ghi nhận các ô đã được kiểm tra, tránh lặp lại.

**⮚ Đưa Điểm Xuất Phát Vào Hàng Đợi:**

⚫ Bắt đầu từ điểm xuất phát, đưa nó vào hàng đợi và đánh dấu đã thăm.

**⮚ Lặp BFS Cho Đến Khi Hàng Đợi Trống:**

⚫ Lặp qua các ô trong hàng đợi.

⚫ Lấy một ô ra khỏi hàng đợi (q.front()) và kiểm tra xem nó có phải là điểm đích hay không.

⚫ Duyệt qua các ô kề cạnh chưa thăm, đưa chúng vào hàng đợi và đánh dấu đã thăm.

**⮚ Đánh Dấu Đường Đi (Nếu Cần):**

⚫ Đối với mỗi ô, lưu trữ thông tin về ô cha (parent) để có thể theo dõi đường đi sau cùng.

**⮚ Kiểm Tra Điều Kiện Dừng:**

⚫ Nếu tìm thấy điểm đích, dừng lại và trả về kết quả "YES".

**⮚ Kết Thúc Nếu Hàng Đợi Trống:**

⚫ Nếu hàng đợi trống mà không tìm thấy đường đi đến đích, trả về kết quả "NO".

1. **Đánh giá thuật toán:**

**Ưu Điểm:**

⚫ Đảm bảo tìm kiếm đường đi ngắn nhất.

⚫ Thích hợp cho các vấn đề liên quan đến đường đi và tìm kiếm trong đồ thị.

**Nhược Điểm:**

⚫ Yêu cầu bộ nhớ lớn, đặc biệt khi đồ thị rất lớn.

⚫ Trong một số trường hợp, có thể không hiệu quả như DFS (tìm kiếm theo chiều sâu).

1. **Áp dụng thuật toán vào bài toán mê cung**

**Thuật toán BFS (Breadth-First Search)** được sử dụng trong bài này để tìm đường đi từ ô (a, b) đến ô (c, d). Phân tích các bước chính để giải quyết bài toán:

**⮚ Bước 1: Đọc Dữ Liệu Từ File:**

⚫ Sử dụng thư viện ‘ifstream’ để mở và đọc dữ liệu từ file văn bản (inpMC.txt).

⚫ Đọc hai số M, N là kích thước của ma trận mê cung.

⚫ Đọc ma trận mê cung với ký tự 'O' hoặc 'X'.

⚫ Đọc số K là số lượng truy vấn.

⚫ Đọc K dòng tiếp theo, mỗi dòng chứa 4 số nguyên a, b, c, d là tọa độ của ô xuất phát (a, b) và ô đích (c, d) cho mỗi truy vấn.

**⮚ Bước 2: Xác Định Có Đường Đi Bằng BFS:**

⚫ Định nghĩa một hàm isValidMove để kiểm tra xem ô (x, y) có thể đi qua không. Hàm này kiểm tra xem ô có nằm trong biên của ma trận, có phải là ô 'O' và chưa được thăm.

⚫ Viết một hàm bfs nhận vào điểm xuất phát và điểm đích của mỗi truy vấn. Trong hàm này:

⚫ Sử dụng hàng đợi để duyệt các ô theo chiều rộng.

⚫ Đánh dấu các ô đã thăm và lưu thông tin về ô cha.

⚫ Kiểm tra khi nào đến được ô đích, nếu có, trả về "YES".

**⮚ Bước 3: Kiểm Tra và Ghi Kết Quả:**

⚫ Đối với mỗi truy vấn:

⚫ Kiểm tra điều kiện hợp lệ cho ô xuất phát và ô đích, nếu không hợp lệ, ghi "NO" vào file kết quả.

⚫ Nếu hợp lệ, thực hiện BFS để kiểm tra đường đi từ ô xuất phát đến ô đích.

⚫ Ghi "YES" nếu có đường đi, ngược lại ghi "NO".

⚫ Lặp lại cho tất cả các truy vấn.

**⮚ Bước 4: Ghi Kết Quả Và Đóng File:**

⚫ Mở và ghi kết quả vào file văn bản (outMC.txt).

⚫ Đóng các file đã mở.

1. **Bài 2. Thêm bớt phần tử**
2. **Giải thích thuật toán:**

**⮚** Thuật toán cây Heap được sử dụng trong bài này là thuật toán Heapify, cùng với các thao tác thêm và loại bỏ phần tử.

**⮚**Cây Heap là một cấu trúc dữ liệu cây nhị phân đặc biệt có hai thuộc tính quan trọng:

⚫ Thuộc tính Heap (Heap Property): Trong một cây Heap nhỏ nhất, giá trị của mỗi nút cha là nhỏ nhất trong các giá trị của nút con của nó. Tương tự, trong một cây Heap lớn nhất, giá trị của mỗi nút cha là lớn nhất trong các giá trị của nút con của nó.

⚫ Cấu trúc cây nhị phân đầy đủ: Cây Heap là một cây nhị phân hoàn chỉnh, có nghĩa là tất cả các tầng (trừ có thể là tầng cuối cùng) đều được điền đầy.

1. **Phân tích các bước chính phải làm để giải quyết vấn đề:**

**⮚** Heapify

⚫ ’heapify’ là quá trình đảm bảo rằng một cây con tại một nút cụ thể là một cây Heap.

⚫ Bắt đầu từ nút cha, so sánh giá trị của nút với giá trị của các nút con.

⚫ Nếu giá trị của một trong các nút con nhỏ hơn giá trị của nút cha, hoán đổi giá trị của nút cha và nút con có giá trị nhỏ nhất. Lặp lại quá trình cho đến khi cây con tại nút đó đều là một Heap.

**⮚** Thao tác Thêm phần tử

⚫ Thêm một phần tử vào cuối mảng để giữ cho cây là một cây nhị phân đầy đủ.

⚫ Sau khi thêm, di chuyển phần tử lên trên cây Heap nếu giá trị của nó nhỏ hơn giá trị của nút cha.

⚫ Thực hiện việc đổi chỗ giữa phần tử và nút cha cho đến khi cây lại trở thành cây Heap.

**⮚** Thao tác Loại bỏ phần tử

⚫ Loại bỏ phần tử nhỏ nhất từ đỉnh của cây Heap.

⚫ Nếu cây chỉ có một phần tử, trả về giá trị của nó.

⚫ Nếu cây có nhiều hơn một phần tử, thay thế giá trị của nút gốc bằng giá trị của phần tử cuối cùng.

⚫ Gọi ‘heapify’ để đảm bảo rằng cây vẫn là một cây Heap.

1. **Đánh giá thuật toán**

**Ưu điểm:**

**⮚** Thời gian Chèn và Loại bỏ phần tử:

⚫ Thời gian chèn một phần tử mới vào cây Heap là O(log N), với N là số lượng phần tử trong cây.

⚫ Thời gian loại bỏ phần tử nhỏ nhất cũng là O(log N), với N là số lượng phần tử trong cây.

⚫ Cả hai thao tác này đều hiệu quả đối với cây Heap.

**⮚** Ưu tiên Phần tử Nhỏ nhất (hoặc Lớn nhất):

⚫ Cây Heap cho phép truy cập nhanh chóng vào phần tử nhỏ nhất (hoặc lớn nhất) trong dãy, giúp giải quyết một số vấn đề tối ưu hóa.

**⮚** Được sử dụng trong Ưu tiên Động:

⚫ Cây Heap thường được sử dụng trong các cấu trúc dữ liệu như hàng đợi ưu tiên (priority queue), nơi mà ưu tiên của phần tử dựa trên giá trị của chúng.

**Nhược điểm:**

**⮚** Không hiệu quả cho Thao tác Truy vấn Ngẫu nhiên:

⚫ Cây Heap không phải là cấu trúc dữ liệu tốt cho việc truy cập ngẫu nhiên vào các phần tử. Điều này có thể ảnh hưởng đến hiệu suất trong trường hợp cần truy cập các phần tử không phải ở đầu hoặc cuối của cây.

**⮚** Không linh hoạt với Cập nhật Giá trị:

⚫ Khi giá trị của một phần tử trong cây thay đổi, cây Heap cần được xây dựng lại từ đầu để đảm bảo tính chất Heap, điều này có thể tốn kém (O(N)).

**⮚** Yêu cầu Bộ nhớ Liên tục:

⚫ Cây Heap yêu cầu mảng hoặc danh sách liên kết để lưu trữ các phần tử, và nếu không gian bộ nhớ không liên tục, có thể dẫn đến tăng chi phí của thao tác chèn và loại bỏ.

**⮚** Không Linh hoạt với Dữ liệu Biến đổi:

⚫ Nếu tập dữ liệu thường xuyên thay đổi, cây Heap có thể không phải là cấu trúc dữ liệu tốt nhất. Các cấu trúc khác như cổng thông tin có thể phù hợp hơn.

1. **Áp dụng thuật toán vào thêm bớt phần tử**

**⮚ Bước 1: Đọc dữ liệu:**

⚫Mở tệp đầu vào (inpTB.txt) để đọc số lượng phần tử N, mảng A, và số lượng truy vấn M.

⚫Đọc giá trị mảng A và số lượng truy vấn từ tệp.

**⮚ Bước 2: Sắp xếp thành cây Heap nhỏ nhất:**

⚫Sử dụng hàm heapify để biến mảng A thành một cây Heap nhỏ nhất.

**⮚ Bước 3: Thực hiện các truy vấn:**

⚫Vòng lặp qua từng truy vấn:

⚫Nếu Q = -1, loại bỏ và xuất phần tử nhỏ nhất của cây Heap.

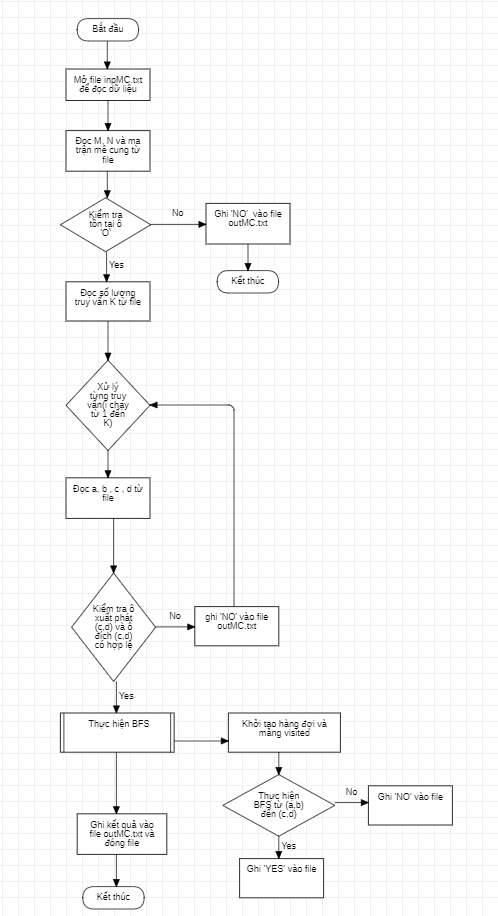
⚫Nếu Q là số không âm, thêm phần tử Q vào cây Heap.

**⮚ Bước 4: Ghi kết quả vào tệp đầu ra:**

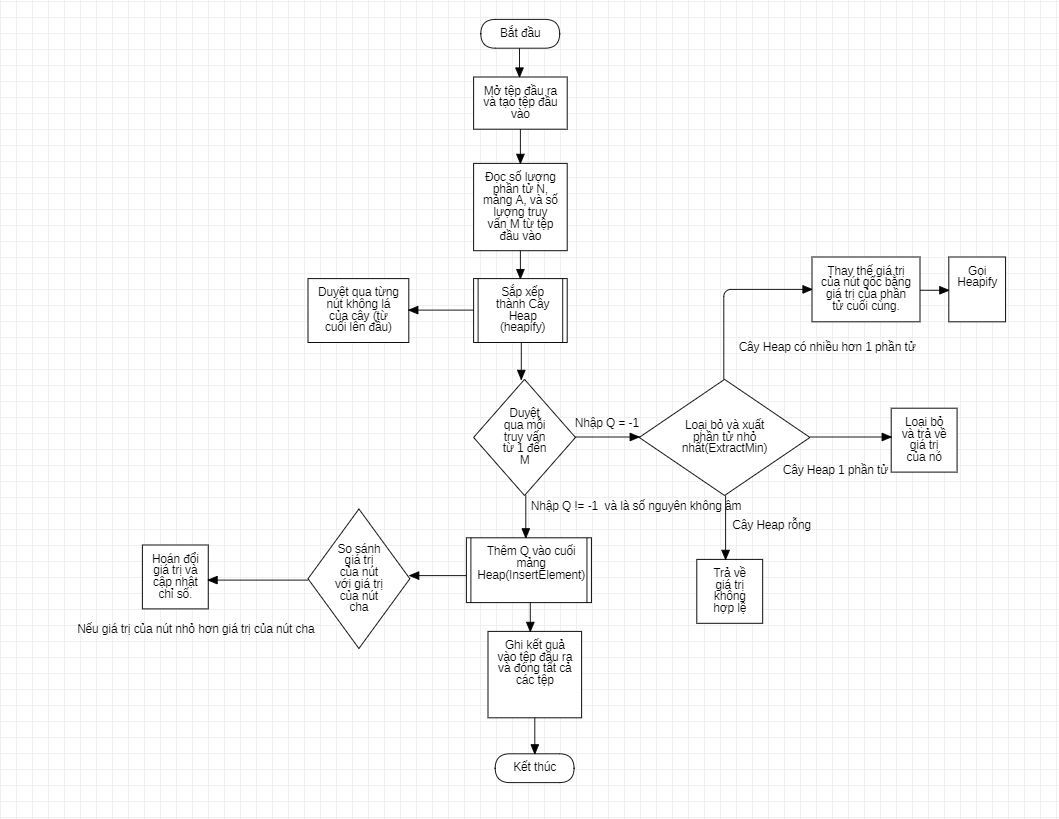
⚫Mở tệp đầu ra (outTB.txt) để ghi kết quả của các truy vấn -1.

⚫Xuất giá trị nhỏ nhất của mảng A tại mỗi truy vấn -1 vào tệp đầu ra.

1. **LƯU ĐỒ THUẬT TOÁN**
2. **Bài 1. Mê Cung**

****

**Bài 2. Thêm bớt phần tử**

****

1. **MÃ GIẢ**
2. **Bài 1. Mê Cung**
3. **Đọc dữ liệu từ file**

ifstream inp("inpMC.txt");

if (!inp.is\_open()) {

cout << "Error opening file" << endl;

return 1;

}

inp >> M >> N;

// Đọc ma trận mê cung từ file

for (int i = 0; i < M; i++) {

for (int j = 0; j < N; j++) {

inp >> maze[i][j];

if (maze[i][j] == 'O') {

hasOpenCell = true;

}

}

}

inp >> K; // Đọc số lượng truy vấn K

1. **Kiểm tra trường hợp không có ô ‘O’ trong ma trận**

if (!hasOpenCell) {

out << "NO\n";

inp.close();

out.close();

return 0;

}

1. **Xử lý mối truy vấn**

while (K--) {

int a, b, c, d;

inp >> a >> b >> c >> d;

// Kiểm tra tính hợp lệ của ô xuất phát và ô đích

if (a < 0 || a >= M || b < 0 || b >= N || c < 0 || c >= M || d < 0 || d >= N || maze[a][b] == 'X' || maze[c][d] == 'X') {

out << "NO\n";

} else {

// Reset trạng thái của mảng visited

for (int i = 0; i < M; i++) {

for (int j = 0; j < N; j++) {

visited[i][j] = false;

}

}

// Gọi hàm bfs và ghi kết quả vào file

out << bfs({a, b}, {c, d}) << "\n";

}

}

1. **Bài 2. Thêm bớt phần tử**
2. **Thao tác thêm phần tử**

void insertElement(vector<int>& arr, int value) {

int i = arr.size();

arr.push\_back(value);

while (i != 0 && arr[(i - 1) / 2] > arr[i]) {

swap(arr[i], arr[(i - 1) / 2]);

i = (i - 1) / 2;

}

}

1. **Thao tác loại bỏ phần tử**

int extractMin(vector<int>& arr) {

if (arr.empty()) {

return INT\_MIN;

}

if (arr.size() == 1) {

int root = arr[0];

arr.pop\_back();

return root;

}

int root = arr[0];

arr[0] = arr.back();

arr.pop\_back();

heapify(arr, arr.size(), 0);

return root;

}

1. **TEST CASE**
2. **Bài 1: Mê cung**
3. **Kiểm tra các trường hợp đặc biệt (Kiểm tra thủ công)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Các trường hợp** | **Ví dụ** | **Kết quả** | **Giải thích** |
| Ma trận 2x2 không có ô mở | 2 2  X X  X X  1  0 0 1 1 | NO | Trong trường hợp này, không có ô nào được mở trong ma trận. Do đó, không có đường đi từ ô (0, 0) đến ô (1, 1). Kết quả là “NO”. |
| Ma trận 3x3 có chướng ngại vật ở giữa | 3 3  O X O  O X O  O O O  1  0 0 2 2 | YES | Ma trận 3x3 có chướng ngại vật ở giữa nhưng vẫn có đường đi từ ô (0,0) đến ô (2,2) qua biên. Kết quả là “YES”. |
| Ma trận 5x5 không có đường đi | 5 5  O X O X O  X O X O X  O X O X O  X O X O X  O X O X O  1  0 0 4 4 | NO | Ma trận 5x5 không có đường đi từ ô (0, 0) đến ô (4, 4) do không có đường đi giữa các ô mở. Kết quả là “NO” |

1. **Kiểm tra hiệu suất của chương trình (Kiểm tra tự động)**

**⮚** Hàm generateTestCase tạo ra một test case ngẫu nhiên bằng cách sử dụng hàm rand() để sinh các giá trị ngẫu nhiên.

⚫ Ô trong ma trận được chọn ngẫu nhiên có thể là 'O' hoặc 'X'.

Tọa độ của ô xuất phát (a, b) và ô đích (c, d) cũng được chọn ngẫu nhiên.

**⮚** Hàm main:

⚫ Hàm main sử dụng srand(time(0)) để khởi tạo seed cho hàm rand() dựa trên thời gian hiện tại, giúp sinh các giá trị ngẫu nhiên khác nhau mỗi lần chạy chương trình.

⚫ Các test case cuối cùng được sinh ra ở trong file inpTB.txt, sau đó sử dụng chương trình chính đã viết ở trên để kiểm tra xem mã nguồn của bạn có hoạt động đúng và hiệu quả không. Sau đó kết quả xuất ra trong file outTB.txt.

1. **Bài 2: Thêm bớt phần tử**

**Quá trình tạo trường hợp thủ công:**

|  |  |  |
| --- | --- | --- |
| **Các trường hợp** | **Ví dụ** | **Kết quả** |
| Mảng A rỗng | 0  1  -1 | INT\_MIN (INT\_MIN được định nghĩa là giá trị không hợp lệ) |
| Nhiều phần tử giống nhau | 4  5 5 5 5  4  5  3  -1  -1 | 3  5 |
| Mảng có phần tử âm | 5  -2 0 1 3 -5  1  -1 | Phần tử mảng không hợp lệ |

**Quá trình tạo trường hợp tự động:**

N: Số lượng phần tử của mảng A được sinh ngẫu nhiên từ 2 đến 100,000.

A: Mảng A chứa N phần tử được sinh ngẫu nhiên từ 0 đến 1,000,000,000.

M: Số lượng truy vấn được sinh ngẫu nhiên từ 1 đến 10,000.

Q: Mảng Q chứa M giá trị được sinh ngẫu nhiên. Có khoảng 50% khả năng mỗi giá trị sẽ là -1, và 50% còn lại là một số ngẫu nhiên từ 0 đến 1,000,000,000.

Sau đó, ghi dữ liệu vào file inpTB.txt:

Dữ liệu được ghi vào file inpTB.txt theo định dạng yêu cầu: dòng đầu tiên là N, dòng thứ hai là mảng A, dòng thứ ba là M, và các dòng tiếp theo là giá trị của mảng Q.

Sử dụng chương trình chính đã viết ở trên để kiểm tra xem mã nguồn của bạn có hoạt động đúng và hiệu quả không. Sau đó kết quả xuất ra trong file outTB.txt.

1. **ĐỘ PHỨC TẠP CỦA THUẬT TOÁN**
2. **Bài 1. Mê cung**
3. **Độ phức tạp thời gian**

⚫ Việc duyệt qua mỗi ô của mê cung để đọc dữ liệu và kiểm tra tính hợp lệ có độ phức tạp là O(M \* N), với M và N là kích thước của mê cung.

⚫ Mỗi truy vấn sử dụng thuật toán BFS để tìm đường đi từ điểm xuất phát đến điểm đích. Độ phức tạp của BFS trong trường hợp tồi nhất là O(M \* N) khi toàn bộ mê cung được duyệt qua.

⚫ Do đó, tổng độ phức tạp thời gian là O(K \* M \* N), với K là số lượng truy vấn.

1. **Độ phức tạp không gian**

⚫ Sử dụng mảng ‘visited’ để theo dõi các ô đã được thăm trong quá trình BFS. Kích thước của mảng này là O(M \* N).

⚫ Sử dụng hàng đợi để lưu trữ các ô cần duyệt trong BFS. Trong trường hợp tồi nhất, khi toàn bộ mê cung đều thăm qua, độ phức tạp không gian của hàng đợi là O(M \* N).

1. **Tổng độ phức tạp**

⚫ Tổng độ phức tạp của chương trình là O(K \* M \* N) với K là số lượng truy vấn và M, N là kích thước của mê cung.

1. **Bài 2. Thêm bớt phần tử**

Độ phức tạp của thuật toán chủ yếu phụ thuộc vào hai thao tác chính: chèn một phần tử mới và loại bỏ phần tử nhỏ nhất. Dưới đây là phân tích độ phức tạp của các thao tác quan trọng trong thuật toán:

1. **Chèn Một Phần Tử Mới (insertElement):**

⚫ Thêm một phần tử vào cuối mảng và sau đó di chuyển nó lên trên cây Heap.

⚫ Độ phức tạp của việc chèn một phần tử là O(log N), với N là số lượng phần tử trong cây Heap.

⚫ Bởi vì chiều cao của cây Heap (log N) quyết định số lần phải di chuyển phần tử lên trên cây.

1. **Loại Bỏ Phần Tử Nhỏ Nhất (extractMin):**

⚫ Loại bỏ phần tử nhỏ nhất từ đỉnh cây Heap.

⚫ Độ phức tạp của việc loại bỏ là O(log N), với N là số lượng phần tử trong cây Heap.

⚫ Bởi vì sau khi loại bỏ, cần thực hiện heapify để khôi phục tính chất Heap.

1. **Heapify (heapify):**

⚫ Đảm bảo rằng một cây con tại một nút cụ thể là một cây Heap.

⚫ Độ phức tạp của heapify là O(log N), với N là số lượng phần tử trong cây Heap.

⚫ Bởi vì heapify chỉ cần thực hiện trên một cây con cụ thể, và chiều cao của cây con đó là log N.

1. **Thời gian Thực Hiện M Truy Vấn (M là số lượng truy vấn):**

⚫ Đối với mỗi truy vấn, có thể là chèn một phần tử hoặc loại bỏ phần tử nhỏ nhất.

⚫ Do mỗi truy vấn đều có độ phức tạp là O(log N), vậy thời gian thực hiện M truy vấn là O(M log N).

1. **Tổng độ phức tạp:**

⚫ Độ phức tạp của toàn bộ thuật toán là O(N log N + M log N), với N là số lượng phần tử ban đầu và M là số lượng truy vấn.

⚫ Trong trường hợp tỷ lệ M so với N lớn, độ phức tạp có thể gần với O(M log N).

Độ phức tạp của thuật toán Heap là hiệu quả đối với các tình huống yêu cầu thao tác ưu tiên nhưng có thể không phải là lựa chọn tốt nhất cho mọi trường hợp, đặc biệt là nếu có sự biến động thường xuyên trong dữ liệu.