|  |
| --- |
| **ĐẠI HỌC BÁCH KHOA HÀ NỘI**  **TRƯỜNG ĐIỆN – ĐIỆN TỬ**  **A picture containing schematic  Description automatically generated**  **BÁO CÁO BÀI TẬP LỚN MÔN**  **Cấu trúc dữ liệu và Giải thuật**  **Project 4: Priority Search Tree**  **Giảng viên hướng dẫn: Tạ Thị Kim Huệ**  **Sinh viên thực hiện: Lê Việt Anh 20200020**  **Nguyễn Thành Luân 20200375**  **Đỗ Tuấn Minh 20200389**  **Lớp: CTTN-ĐTVT-K65**  **Mã lớp: 136093**  **HÀ NỘI, NĂM 2023** |

Contents

[I. Giới thiệu 3](#_Toc129679020)

[II. Tổng quan, ví dụ về PST 4](#_Toc129679021)

[III. Truy vấn 5](#_Toc129679022)

[IV. Xây dựng cây tìm kiếm ưu tiên 7](#_Toc129679023)

[V. Chèn hoặc xóa một Node 10](#_Toc129679024)

[Reference 10](#_Toc129679025)

# Giới thiệu

Trong khoa học máy tính, cây tìm kiếm ưu tiên (priority search tree) là một cấu trúc dữ liệu dạng cây để lưu trữ các điểm trong không gian hai chiều (Oxy). Ban đầu cây tìm kiếm ưu tiên được giới thiệu bởi Edward McCreight năm 1985. Ban đầu cây tìm kiếm ưu tiên là sự mở rộng của hàng đợi ưu tiên (priority queue) với mục đích cải thiện thời gian tìm kiếm từ: O(n) đến O(s+logn) trong đó n là số điểm trong cây và s là số trong tổng số điểm được trả về bởi tìm kiếm. Sau đó, cây tìm kiếm ưu tiên được sử dụng để lưu trữ một tập hợp các điểm 2 chiều được sắp xếp theo mức độ ưu tiên(priority) và theo một giá trị khóa (key value). Điều này được thực hiện bằng cách tạo kết hợp giữa hàng đợi ưu tiên (priority queue) và cây tìm kiếm nhị phân (binary search tree). Kết quả là một cây trong đó mỗi nút đại diện cho một điểm trong tập dữ liệu gốc. Điểm được chứa bởi nút là điểm có mức độ ưu tiên thấp nhất. Ngoài ra, mỗi nút còn chứa một giá trị khóa dùng để chia các điểm còn lại (thường là trung vị của các khóa, không kể điểm của nút) thành cây con trái và phải. Các điểm được chia bằng cách so sánh các giá trị khóa của chúng với khóa nút, ủy nhiệm các giá trị có khóa thấp hơn cho cây con bên trái và các giá trị lớn hơn cho cây con bên phải.

Diagram

Description automatically generated

Hình 1: Ví dụ về cây tìm kiếm ưu tiên

# Tổng quan, ví dụ về PST

Cấu trúc cây PST là kết hợp của cấu trúc (min) Heap nổi tiếng và Binary Search Tree. Min-Heap là một cấu trúc cây nhị phân thỏa mãn các tính chất sau:

1. Tính chất Heap (Heap Property): Mỗi nút có một giá trị (khóa) nhỏ hơn hoặc bằng giá trị khóa của hai nút con.
2. Tính cân bằng mạnh (Strong Balance Property): Mức thứ i không phải là mức cao nhất có 2i nút­. Ở đây, nút gốc có mức 0 và mức của một nút là mức của nút cha +1. Các nút của mức cao nhất (các nút lá) được đính vào cây theo thứ tự liên tục từ trái sang phải. Ví dụ trong hình dưới đây nếu ta đặt nút 7 làm nút con phải của 3 thì các nút lá không còn liên tục theo thứ tự từ trái sang phải nữa.

Ví du tập các giá trị {3, 1, 1, 4, 5, 7} có cấu trúc Heap biểu diễn minh họa trong hình dưới đây:

Diagram

Description automatically generated

Từ tính chất Heap ta có thể chứng minh (bằng quy nạp) được rằng giá trị khóa ở gốc là giá trị nhỏ nhất trong tập các khóa của cây (do đó gọi là min-heap). Từ tính cân bằng mạnh ta có thể suy ra chiều cao của một Heap với n nút tối đa là [ log(n) ]. Tính cân bằng này gọi là mạnh vì nút ở mức lá phải là liên tục từ trái sang phải. So với các cây nhị phân tìm kiếm thì Heap đơn giản hơn, mềm dẻo hơn và là một cấu trúc cân bằng mà sự cân bằng đó có thể được duy trì khá đơn giản khi thực hiện các thao tác trên heap mà không cần phải có thêm thông tin phụ. Do đó các thao tác với Heap thông thường chỉ mất thời gian O(log n).

Một cây nhị phân PST biểu diễn tập điểm P là một cây thỏa mãn tính chất sau:

1. Mỗi nút của cây biểu diễn một điểm trong P.
2. Tính chất Heap Tọa độ x của nút cha nhỏ hơn hoặc bằng tọa độ x của hai nút con.
3. Tính cân bằng yếu Mức thứ i không phải là mức cao nhất có 2i nút.
4. Tính chia đôi Mỗi nút v có một giái trị yv , được gọi là điểm chia, trong đó mỗi điểm p thuộc nút nằm trong cây con trái gốc tại v thỏa mãn yp ≤ yv và mỗi điểm p’ thuộc nút nằm trong cây con phải gốc tại v thỏa mãn yv ≤ yp’ Giá trị này ở các nút lá là tung độ của điểm trong P biểu diễn bởi nút đó.

Chú ý trong tính chia đôi, ta cho phép cả cây con trái và con phải của v có các điểm tung độ bằng yv . Điều này là cần thiết để đảm bảo tính cân bằng. Ví dụ nếu ta chỉ cho cây con trái có điểm có tọa độ yv và con phải có tọa độ > yv thì không thể đảm bảo tính cân bằng trong trường hợp tất cả các điểm trong P có cùng tung độ.

Ví dụ cây PST với tập P như trên được biểu diễn ở hình sau. Các cặp giá trị ở phần background màu xanh của mỗi nút là một điểm biểu diễn bởi nút đó. Giá trị ở phần dưới là điểm chia yv.

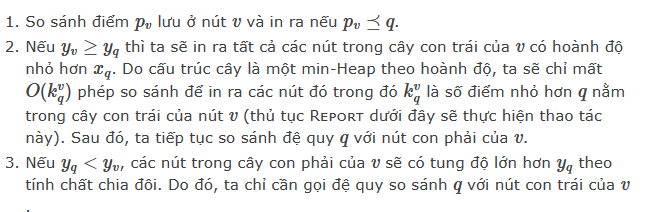
Diagram

Description automatically generated

Ta có thể quan sát thấy việc tìm kiếm theo tọa độ x của các điểm được "tích hợp" vào cây thông qua tính chất Heap và tìm kiếm theo tọa độ y của các điểm được "tích hợp" thông qua tính chia đôi. Tính cân bằng đảm bảo chiều cao của cây là [log n] . Với các tính chất đó, ta có thể nhận thấy rằng PST là một cấu trúc phù hợp để tìm kiếm hai chiều với độ phức tạp của mỗi thao tác tìm kiếm là O(log n). Dễ thấy cấu trúc cây PST có bộ nhớ là O(n).

# Truy vấn

Giả sử chúng ta đã xây dựng được cây PST, kí hiệu là T, biểu diễn tập điểm P, ta sẽ thực hiện truy vấn với mỗi điểm q ∈ Q. Để in ra các điểm nhỏ hơn q, ta sẽ so sánh q với cây T, bắt đầu từ nút gốc. Gọi v là một nút trong T mà ta đang so sánh với q:



Giả mã của thủ tục truy vấn như sau:

Graphical user interface, text, application

Description automatically generated

Code:

**void** query(node \*root, point q){

**if**(root == null) **return**;

**if**(root-&gt;p.x &lt;= q.x &amp;&amp; root-&gt;p.y &lt;= q.y){

print\_point(root-&gt;p);

}

**if**(root-&gt;ymid &lt;= q.x){

report(root-&gt;left, q);

query(root-&gt;right, q);

}**else**{

query(root-&gt;left, q);

}

}

**void** report(node \*root, point q){

**if**(root == null) **return**;

**if**(root-&gt;p.x &lt;= q.x){

print\_point(root-&gt;p);

report(root-&gt;left, q);

report(root-&gt;right, q);

}

}

# Xây dựng cây tìm kiếm ưu tiên

Gọi P = {P[1], P[2], …, P[n]} là tập các điểm trong P. Ta sẽ xây dựng cây PST một cách đệ quy như sau:

Text, letter

Description automatically generated

Ví dụ với tập P = {(1,4), (1,5), (3,2), (4,3), (5,6), (7,1), (7,7)}, các bước xây dựng cây PST được mô tả trong hình sau:

Diagram

Description automatically generated

Giả mã của thuật toán:

Text

Description automatically generated

**Phân tích thuật toán:** Ta thấy chọn điểm với hoành độ nhỏ nhất có thể thực hiện trong O(n) bằng cách duyệt qua tất cả các phần tử trong mảng. Sắp xếp các điểm trong P theo tọa độ y ngay từ ban đầu với thời gian O(n log n). Khi đó, việc chọn median có thể thực hiện trong thời gian O(n) vì các điểm trong P là đã sắp xếp theo y. Chia P ra thành 2 phần PL và PR mất O(n). Gọi T(n) là thời gian xây dựng cây PST. Do PL và PR xấp xỉ n/2, ta suy ra công thức đệ quy:



Giải công thức truy hồi trên ta được T(n) = O(n log n). Do đó:



Code:

**void** simple\_pst\_construction(point \*A){

**qsort**(A,N,**sizeof**(\*A), comp\_Y);

//  print\_array(A,N);

root = recursive\_pst\_const(A,0, N-1);

}

node \*recursive\_pst\_const(point \*A, **int** l, **int** h){

**if**(h &lt; l) **return** null;

**else** **if**(h== l) **return** create\_node(A[l], A[l].y);

**else** **if**(h-l == 1){

node \*u = create\_node(A[l], A[l].y);

node \*v = create\_node(A[h], A[h].y);

**if**(u-&gt;p.x &lt; v-&gt;p.x){

u-&gt;right = v; // remember here u-&gt;y &lt; v-&gt;y

**return** u;

}**else** {

v-&gt;left = u;

**return** v;

}

}**else** {

**int** k = -1;

**int** min  = INFTY;

**int** i = l;

// find the point of minimum x-coordinate

**for**(; i &lt;= h; i++){

**if**(A[i].x &lt; min){

min = A[i].x;

k = i;

}

}

// put A[k] to the end of the array

point tmp = A[k];

**int** j = k+1;

**for**(; j &lt;= h; j++){

A[j-1] = A[j];

}

A[h] = tmp;

// call recursive-pst-construction

**int** m = (l + h-1) &gt;&gt;1;

node \*v = create\_node(A[h], A[m].y);

v-&gt;left = recursive\_pst\_const(A, l, m);

v-&gt;right  = recursive\_pst\_const(A,m+1, h-1);

**return** v;

}

}

node \*create\_node(point p, **int** yv){

node \*v = (node \*)**malloc**(**sizeof**(node));

v-&gt;p = p;

v-&gt;ymid = yv;

v-&gt;left = null;

v-&gt;right = null;

}

// the compare function for quick sort

**int** comp\_Y(**const** **void** \*a, **const** **void** \*b){

point \*pa = (point \*)a;

point \*pb = (point \*)b;

**return** pa-&gt;y - pb-&gt;y;

}

# Chèn hoặc xóa một Node

Khi thực hiện thao tác thêm mới hoặc xóa, do cây PST cơ bản không có cơ chế tự cân bằng, nên khi thêm hoặc xóa Node sẽ làm cây mất cân bằng, tăng độ phức tạp của bài toán khi thực hiện các thao tác với cây.

Khi đó để có thể thêm hoặc xóa Node cần phải cài đặt cây PST động (Dynamic Priority Search Tree)

# Reference

[Giải Thuật Lập Trình · Cây tìm kiếm ưu tiên — Priority Search Tree (giaithuatlaptrinh.com)](https://www.giaithuatlaptrinh.com/?p=643)

[lecturerev.dvi (brown.edu)](https://cs.brown.edu/courses/cs252/misc/resources/lectures/pdf/notes18.pdf)