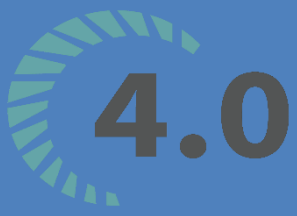
KHOA CÔNG NGHỆ THÔNG TIN

ĐẠI HỌC KHOA HỌC TỰ NHIÊN THÀNH PHỐ HỒ CHÍ MINH, ĐẠI HỌC QUỐC GIA TP HCM

BÁO CÁO ĐỒ ÁN LÝ THUYẾT
CẤU TRÚC DỮ LIỆU & GIẢI THUẬT



Giảng viên lý thuyết: Nguyễn Ngọc Đức

Giảng viên thực hành: Nguyễn Trần Duy Minh

Sinh viên thực hiện: 20120077 – Nguyễn Quang Hiển

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

HỌC KỲ II – NĂM HỌC 2022-2023

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
227 Nguyễn Văn Cừ, Phường 4, Quận 5, TP.HCM
Điện Thoại: (08) 38.354.266 - Fax:(08) 38.350.096

BẢNG THÔNG TIN CHI TIẾT

MSSV	Họ tên	Email	Điện thoại
20120077	Nguyễn Quang Hiển	20120077@student.hcmus.edu.vn	0925919454
20120087	Dương Đình Bảo Hoàng	20120087@student.hcmus.edu.vn	

YÊU CẦU ĐỒ ÁN - BÀI TẬP

Ngày bắt đầu	16/05/2023
Ngày kết thúc	19/05/2023

MỤC LỤC

1.Splay Tree

1.1. Giới thiệu (Introduction)

1.1.1.Khái quát

Splay Tree (cây splay) là một cấu trúc dữ liệu cây nhị phân tự cân bằng. Một điểm đặc biệt của Splay Tree là nút được truy cập gần đây nhất sẽ được đưa lên đỉnh cây thông qua một quá trình gọi là "splaying" (căng). Quá trình này làm tăng hiệu suất của các phép toán tiếp theo trên cùng một nút, vì nút đã được đưa gần nhất vào vị trí gần đỉnh. Quá trình splaying được thực hiện thông qua một số phép xoay và tráo đổi nút.

1.1.2.Ưu điểm

Cấu trúc cây splay cũng có một số ưu điểm khác nhau. Đầu tiên, nó không yêu cầu thông tin bổ sung (như các chỉ số cân bằng) để duy trì cân bằng cây. Thay vào đó, nó sử dụng phép xoay và tráo đổi nút để cân bằng cây dựa trên lịch sử truy cập. Việc splay (căng) nút gần đây nhất lên đỉnh cây giúp cải thiện hiệu suất cho các phép toán truy cập tiếp theo trên cùng một nút. Điều này đảm bảo thời gian trung bình của các phép toán là $O(\log n)$.

Thứ hai, cây splay tương đối dễ hiểu và triển khai so với các cây tự cân bằng phức tạp hơn như cây RB hoặc AVL. Thứ ba, mặc dù cây splay không đảm bảo độ cân bằng tuyệt đối như các cấu trúc cây khác nhưng nó thường cho kết quả tốt và có hiệu suất tốt trong nhiều ứng dụng thực tế.

1.1.3.Nhược điểm

Trong trường hợp xấu nhất, cây splay có thể trở thành một danh sách liên kết đơn, dẫn đến hiệu suất là $O(n)$, tuy vậy trường hợp này rất hiếm xảy ra.

Ngoài ra, do không đảm bảo độ cân bằng tuyệt đối như cây RB hoặc AVL dẫn đến một số trường hợp ảnh hưởng đến hiệu suất.

Cuối cùng, cây Splay không phù hợp với một số ứng dụng yêu cầu hiệu suất dự đoán và đảm bảo như hệ thống real-time hoặc hệ thống quan trọng về an toàn (SCS).

1.1.4.Cải tiến (các biến thể)

- **Top-Down Splay Tree:** là một biến thể của cây splay gốc mà tập trung vào việc thực hiện các phép toán từ đỉnh cây xuống. Thay vì splay nút gần đây nhất lên đỉnh cây, top-down splay tree splay các nút trên đường đi từ gốc đến nút được truy cập. Điều này giúp tránh một số trường hợp xấu nhất và cải thiện hiệu suất trong một số tình huống.
- **Bottom-Up Splay Tree:** là một biến thể khác của cây splay, trong đó splay được thực hiện từ nút được truy cập đến đỉnh cây. Thay vì di chuyển nút gần đây nhất lên đỉnh, bottom-up splay tree splay các nút trên đường đi từ nút được truy cập đến gốc. Biến thể này nhằm cải thiện hiệu suất trong các trường hợp khác nhau so với top-down splay tree.
- **Zig-Zag Splay Tree:** là một biến thể của cây splay mà sử dụng phép xoay kết hợp giữa hai nút con của nút được truy cập để cân bằng cây. Thay vì chỉ splay một nút đơn lẻ, Zig-Zag Splay Tree splay cả cụm nút theo một cấu trúc zig-zag, giúp cân bằng cây một cách hiệu quả hơn.
- **Multi-Splay Tree:** là một biến thể mở rộng của cây splay, trong đó nhiều nút được splay cùng một lúc thay vì chỉ một nút. Việc splay nhiều nút cùng một lúc có thể giúp cải thiện hiệu suất cho các phép toán trên tập hợp các nút liên quan.
- **Cache-Oblivious Splay Tree:** là một biến thể của cây splay được thiết kế để tận dụng tối đa hiệu quả của bộ nhớ cache trong việc truy cập dữ liệu. Nó tối ưu hóa việc sắp xếp nút trong cây dựa trên các quy tắc tự động thích ứng để tận dụng cấu trúc bộ nhớ cache mà không cần biết chi tiết về kiến trúc bộ nhớ cache cụ thể.
- **Rank-Based Splay Tree:** là một biến thể của cây splay nơi cây được cân bằng dựa trên thứ hạng (rank) của các nút thay vì các khóa (key). Thứ hạng của một nút là chỉ số của nó trong thứ tự tăng dần khi duyệt cây theo thứ tự inorder. Việc splay các nút dựa trên thứ hạng giúp đảm bảo tính cân bằng của cây và cải thiện hiệu suất trong các trường hợp đặc biệt.
- **Dynamic Finger Splay Tree:** là một biến thể của cây splay nhằm cải thiện hiệu suất cho việc truy cập tại các vị trí gần "ngón tay" (finger). Nút được truy cập gần đây nhất được giữ trong bộ nhớ tạm thời, và việc splay được thực hiện bằng cách di chuyển các nút từ vị trí gần finger đến gốc cây. Điều này giúp tăng tốc độ truy cập cho các nút gần "ngón tay" mà không làm thay đổi các phần tử khác trong cây.
- **Top-K Splay Tree:** là một biến thể của cây splay nhằm hỗ trợ các phép toán truy vấn "tìm K phần tử lớn nhất" hoặc "tìm K phần tử nhỏ nhất" trong cây. Cây được sắp xếp theo thứ tự tăng dần hoặc giảm dần, và việc splay các nút gần các phần tử lớn nhất hoặc nhỏ nhất giúp cải thiện hiệu suất của các truy vấn này.

1.1.5.Ứng dụng

- **Tổ chức dữ liệu:** cây splay thường được sử dụng để triển khai các cấu trúc dữ liệu như bảng băm (hash table) hoặc các phần tử có thứ tự (ordered set).
- **Bộ nhớ cache:** cây Splay có thể được sử dụng để quản lý bộ nhớ cache trong các hệ thống quản lý bộ nhớ cache. Việc di chuyển các phần tử được truy cập thường xuyên lên đỉnh cây giúp cải thiện hiệu suất truy cập dữ liệu trong bộ nhớ cache, vì các phần tử đã được truy cập gần đây có khả năng được truy cập nhanh hơn.
- **Tạo chỉ mục cho cơ sở dữ liệu:** cây Splay có thể được sử dụng để tạo chỉ mục cho cơ sở dữ liệu, giúp tìm kiếm và truy xuất dữ liệu nhanh hơn.
- **Hệ thống tập tin:** cây Splay có thể được sử dụng để lưu trữ các siêu dữ liệu hệ thống tệp, chẳng hạn như bảng phân bổ, cấu trúc thư mục và thuộc tính của tệp.
- **Nén dữ liệu:** cây Splay có thể được sử dụng để nén dữ liệu bằng cách xác định và mã hóa các mẫu lặp lại.
- **Xử lý văn bản:** cây Splay có thể được sử dụng trong các ứng dụng xử lý văn bản, chẳng hạn như kiểm tra chính tả, nơi các từ được lưu trữ trong cây Splay để tìm kiếm và truy xuất nhanh chóng.
- **Thuật toán đồ thị:** cây Splay có thể được sử dụng để triển khai các thuật toán đồ thị, chẳng hạn như tìm đường đi ngắn nhất trong đồ thị có trọng số.
- **Trò chơi trực tuyến:** cây Splay có thể được sử dụng trong trò chơi trực tuyến để lưu trữ và quản lý điểm cao, bảng xếp hạng và thống kê của người chơi.

1.2. Phương pháp (Method)

1.2.1.Splaying

- Kiểm tra nếu node gốc là nullptr hoặc giá trị key của node gốc trùng với key cần tìm kiếm, trả về node gốc.
- Nếu giá trị key cần tìm kiếm nhỏ hơn giá trị key của node gốc, xử lý nhánh trái của node gốc:
    - Nếu nhánh trái là nullptr, trả về node gốc.
    - Nếu giá trị key cần tìm kiếm nhỏ hơn giá trị key của node con trái của node gốc, thực hiện xoay Zig-Zig:
        - Xoay phải (rotateRight) node gốc.
        - Xoay phải (rotateRight) node gốc thứ hai.

    - Nếu giá trị key cần tìm kiếm lớn hơn giá trị key của node con trái của node gốc, thực hiện xoay Zag-Zig:
        - Xoay trái (rotateLeft) node con trái của node gốc.
        - Xoay phải (rotateRight) node gốc.
    - Trả về node gốc sau khi đã splay.
- Nếu giá trị key cần tìm kiếm lớn hơn giá trị key của node gốc, xử lý nhánh phải của node gốc:
    - Nếu nhánh phải là nullptr, trả về node gốc.
    - Nếu giá trị key cần tìm kiếm lớn hơn giá trị key của node con phải của node gốc, thực hiện xoay Zag-Zag:
        - Xoay trái (rotateLeft) node gốc.
        - Xoay trái (rotateLeft) node gốc thứ hai.
    - Nếu giá trị key cần tìm kiếm nhỏ hơn giá trị key của node con phải của node gốc, thực hiện xoay Zig-Zag:
        - Xoay phải (rotateRight) node con phải của node gốc.
        - Xoay trái (rotateLeft) node gốc.
    - Trả về node gốc sau khi đã splay.

1.2.2. Thêm

- Nếu cây đang rỗng, tạo một node mới với giá trị key và trả về node đó.
- Splay node gốc với giá trị key. (Điều này sẽ đưa node gốc về gần nhất với key).
- Nếu giá trị key của node gốc bằng key, có nghĩa là node đã tồn tại trong cây. Trả về node gốc.
- Tạo một node mới với giá trị key.
- Nếu key nhỏ hơn giá trị key của node gốc, thực hiện các bước sau:
    - Gán node mới là con phải của node gốc.
    - Gán nhánh trái của node gốc là nullptr.
    - Gán node mới là con trái của node gốc.
- Nếu key lớn hơn giá trị key của node gốc, thực hiện các bước sau:
    - Gán node mới là con trái của node gốc.
    - Gán nhánh phải của node gốc là nullptr.
    - Gán node mới là con phải của node gốc.
- Trả về node mới là node gốc mới của cây.

1.2.3.Xoá

- Trước khi tiến hành xoá, kiểm tra xem node gốc có tồn tại hay không. Nếu nút gốc là null, tức là cây rỗng, trả về node gốc ban đầu.
- Tiến hành splay node tìm kiếm lên làm node gốc của cây. Điều này đảm bảo rằng node cần xoá sẽ nằm ở gốc cây.
- So sánh giá trị của node gốc với giá trị cần xoá. Nếu giá trị không trùng khớp, tức là node cần xoá không tồn tại trong cây, trả về nút gốc ban đầu.
- Nếu giá trị trùng khớp, tiến hành xoá node. Có hai trường hợp:
    - Nếu nhánh trái của nút gốc là null, gán nút gốc bằng nhánh phải và xoá node ban đầu.
    - Nếu nhánh trái của nút gốc không phải là null, tiến hành splay node phía trái lên làm nút gốc. Sau đó, gán nhánh phải của nút gốc ban đầu cho nhánh phải của nút trái và xoá node ban đầu.
- Sau khi xoá node, giải phóng bộ nhớ của node đã xoá và trả về nút gốc đã được cập nhật.

1.2.4.Tìm kiếm

- Kiểm tra nếu node gốc là nullptr hoặc giá trị key của node gốc bằng key cần tìm, trả về node gốc.
- Nếu key cần tìm nhỏ hơn giá trị key của node gốc, ta tiến hành tìm kiếm trong nhánh trái của node gốc:
    - Nếu nhánh trái là nullptr, trả về node gốc.
    - Nếu key cần tìm nhỏ hơn giá trị key của node con trái của node gốc, ta thực hiện phép xoay Zig Zig để đưa node con trái lên đỉnh cây.
    - Nếu key cần tìm lớn hơn giá trị key của node con trái của node gốc, ta thực hiện phép xoay Zig Zag để đưa node con trái và node con phải lên đỉnh cây.
- Nếu key cần tìm lớn hơn giá trị key của node gốc, ta tiến hành tìm kiếm trong nhánh phải của node gốc:
    - Nếu nhánh phải là nullptr, trả về node gốc.
    - Nếu key cần tìm nhỏ hơn giá trị key của node con phải của node gốc, ta thực hiện phép xoay Zag Zig để đưa node con phải và node con trái lên đỉnh cây.
    - Nếu key cần tìm lớn hơn giá trị key của node con phải của node gốc, ta thực hiện phép xoay Zag Zag để đưa node con phải lên đỉnh cây.
- Trả về node gốc sau khi hoàn thành việc tìm kiếm.

1.3. Cài đặt (Experiment)

1.3.1.Bài toán minh hoạ

Bài toán minh hoạ cho các phép quay splay trong cây Splay

Giả sử tôi có 4 nút A, B, C, D trong cây Splay. Sau đây tôi sẽ minh hoạ từng phép xoay cây.

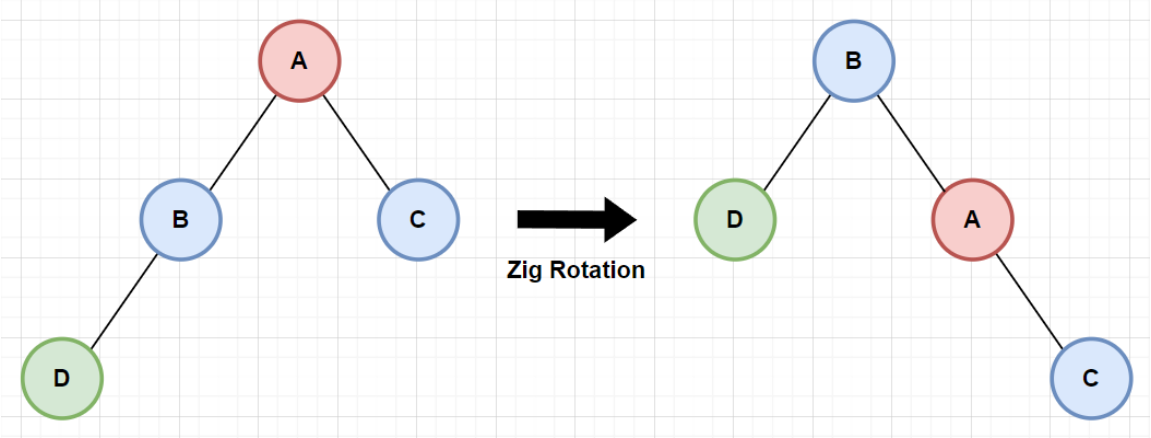
1. **Phép quay Zig**

   Phép xoay Zig trong cây Slay tương tự như phép xoay phải trong cây AVL. Sau đây tôi xin minh hoạ phép xoay Zig tại nút A (nút gốc) như sau:

   Bước 1: Gán con trái của A cho con phải của B.

   Bước 2: Gán con phải của B cho A.

   Bước 3: Cập nhật lại nút gốc.



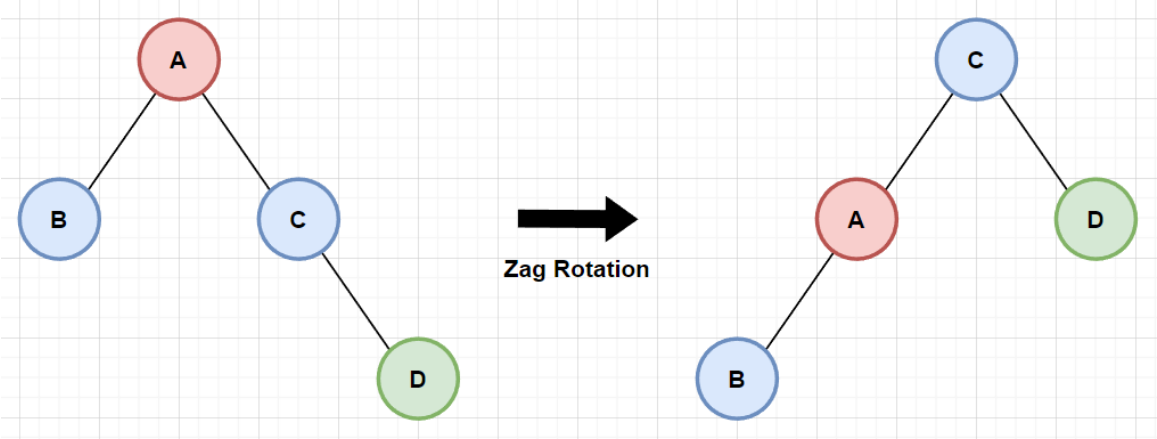
2. **Phép quay Zag**

   Phép xoay Zag trong cây Splay tương tự như phép xoay trái trong cây AVL. Sau đây tôi minh hoạ phép xoay Zag tại nút A (nút gốc) như sau:

   Bước 1: Gán con phải của A cho con trái của C.

   Bước 2: Gán con trái của C cho A.

   Bước 3: Cập nhật lại nút gốc.

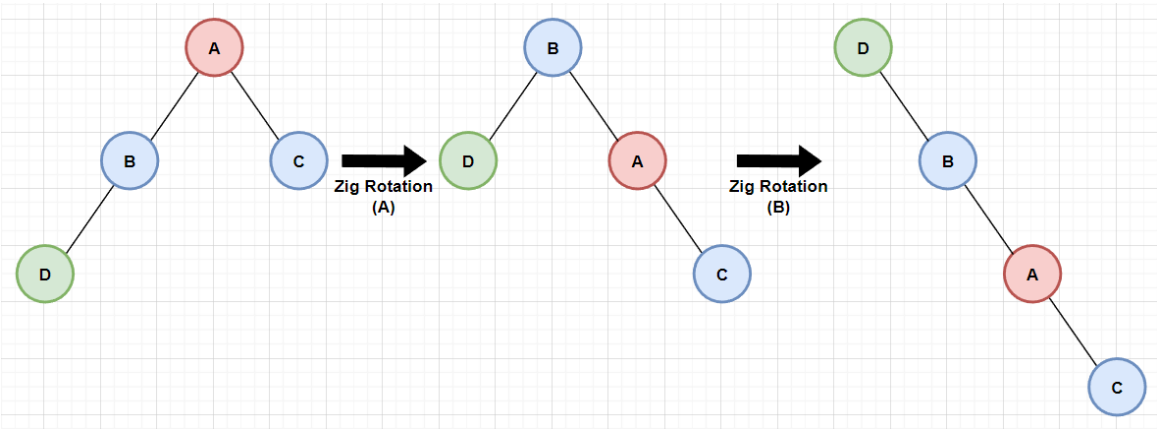


3. Phép quay Zig-Zig

Phép xoay Zig-Zig trong cây Splay tương tự như phép xoay phải-phải trong cây AVL. Sau đây tôi minh hoạ phép xoay như sau:

Bước 1: Đầu tiên, thực hiện phép quay Zig tại nút A.

Bước 2: Lúc này, nút gốc là nút B, tiếp tục thực hiện phép quay Zig tại nút B.

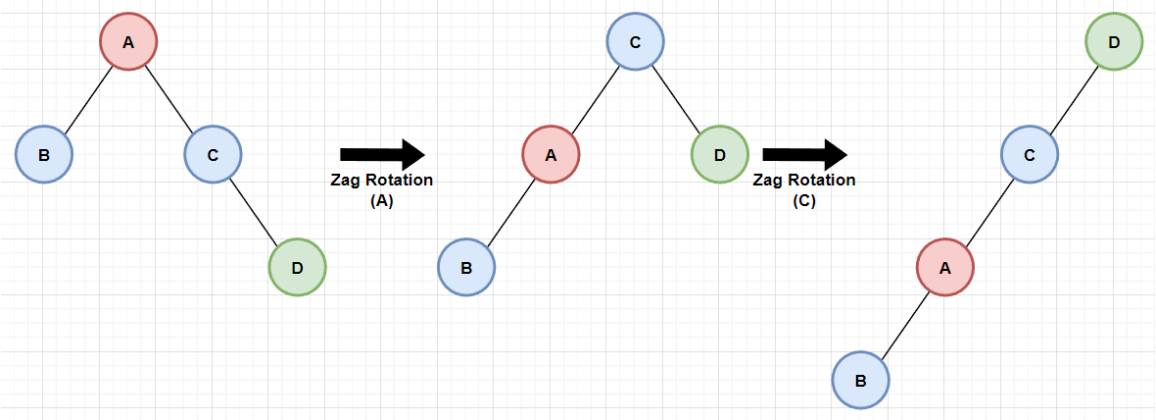


4. Phép quay Zag-Zag

Phép xoay Zag-Zag trong cây Splay tương tự như phép xoay trái-trái trong cây AVL. Sau đây tôi minh hoạ phép xoay như sau:

Bước 1: Đầu tiên, thực hiện phép quay Zag tại nút A.

Bước 2: Lúc này, nút gốc là nút C, tiếp tục thực hiện phép quay Zig tại nút C.

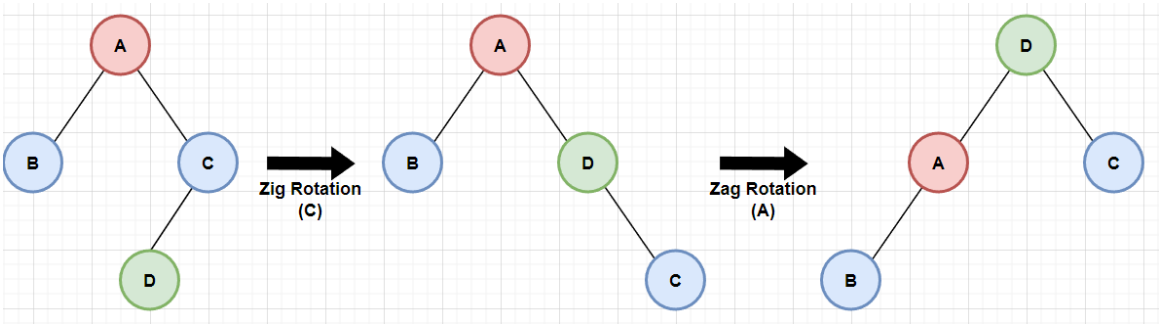


5. Phép quay Zig-Zag

Phép xoay Zig-Zag trong cây Splay tương tự như phép xoay phải-trái trong cây AVL. Sau đây tôi minh hoạ phép xoay như sau:

Bước 1: Đầu tiên, thực hiện phép quay Zig tại nút con phải của nút A là nút C.

Bước 2: Sau đó, thực hiện phép quay Zag tại nút A.

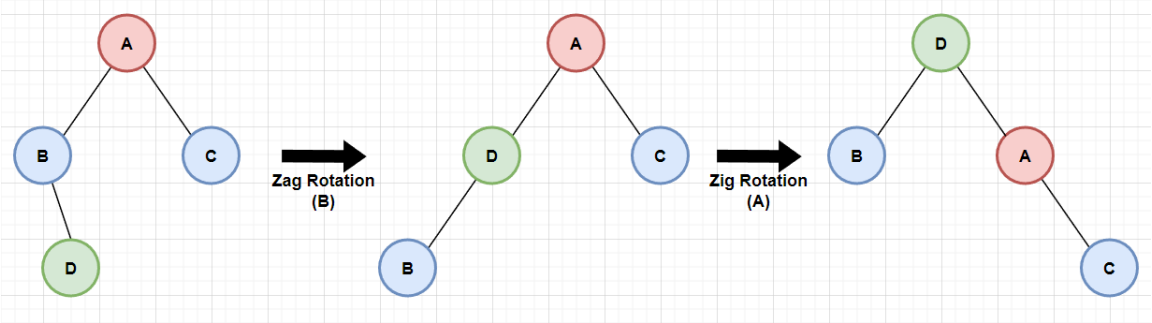


6. Phép quay Zag-Zig

Phép xoay Zag-Zig trong cây Splay tương tự như phép xoay trái-phải trong cây AVL. Sau đây tôi minh hoạ phép xoay như sau:

Bước 1: Đầu tiên, thực hiện phép quay Zag tại nút con trái của nút A là nút B.

Bước 2: Sau đó, thực hiện phép quay Zig tại nút A.



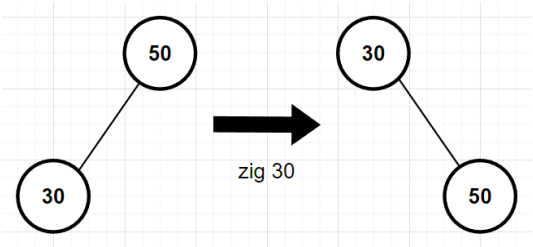
Bài toán minh hoạ thêm các node trong cây Splay

Giả sử tôi có một cây Splay rỗng và sau đó tôi sẽ lần lượt thêm các nút có giá trị 50, 30, 70, 20, 40, 60, 80.

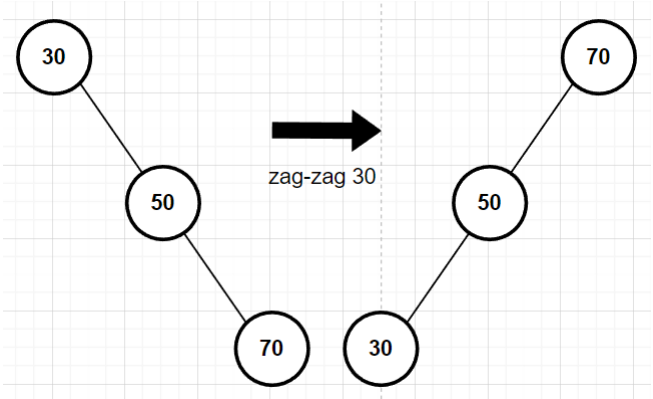
Để cho ngắn gọn phần trình bày tôi gọi "nút (node) có giá trị 50" là "nút 50".

Bước 1: Thêm nút 50 vào cây rỗng, nút 50 là nút gốc hiện tại của cây.

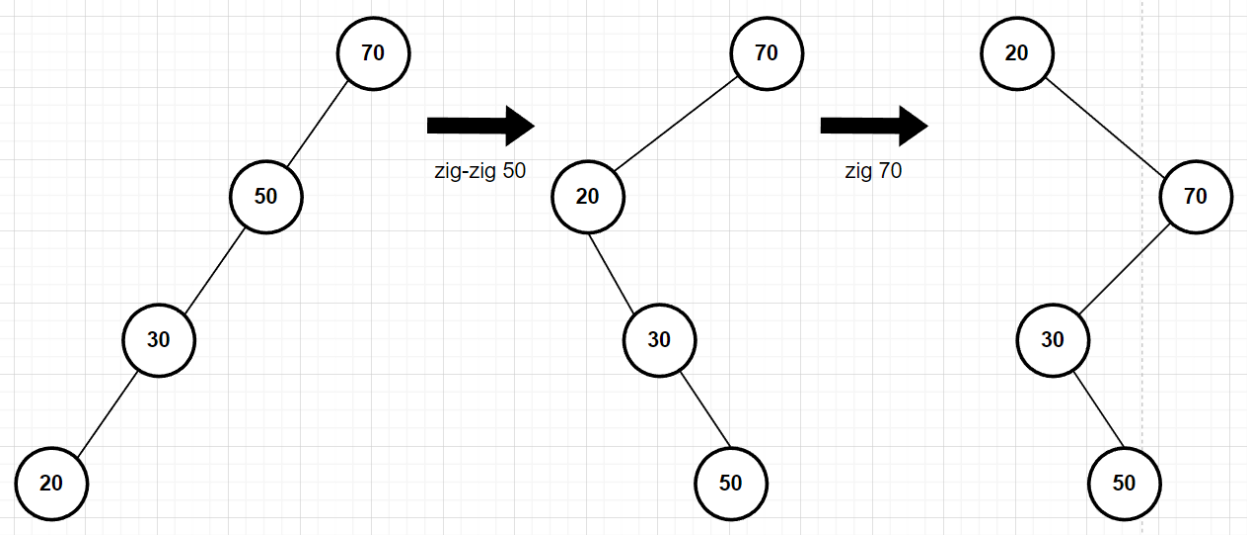
Bước 2: Thêm nút 30 vào cây. Vì nút mới thêm có giá trị nhỏ hơn nút gốc nên được thêm vào cây con bên trái. Sau đó thực hiện phép xoay Zig tại nút 50 để đưa nút vừa thêm vào lên làm nút gốc (đây là đặc tính quan trọng nhất của cây Splay).



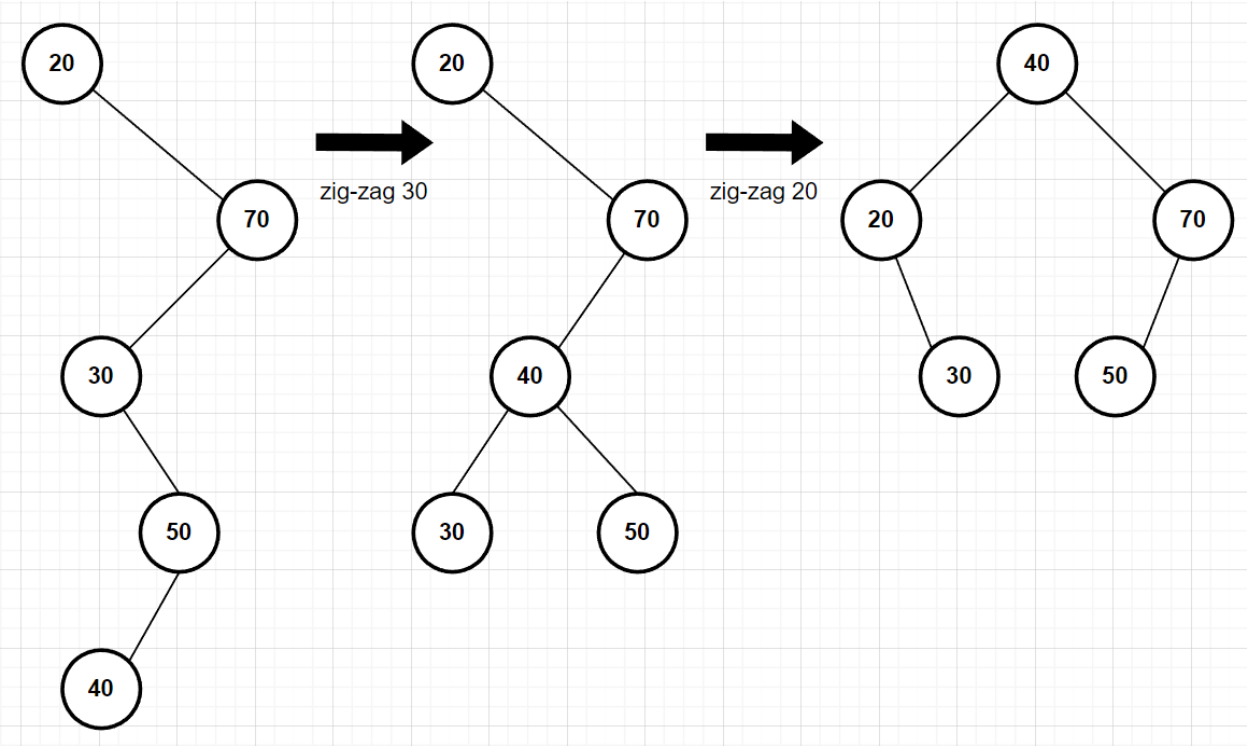
Bước 3: Thêm nút 70 vào cây. Vì nút mới thêm lớn hơn 50 nên sẽ được thêm vào cây con bên phải của nút 50. Sau đó thực hiện phép xoay Zag-Zag tại nút 30 để đưa nút 70 lên làm nút gốc.



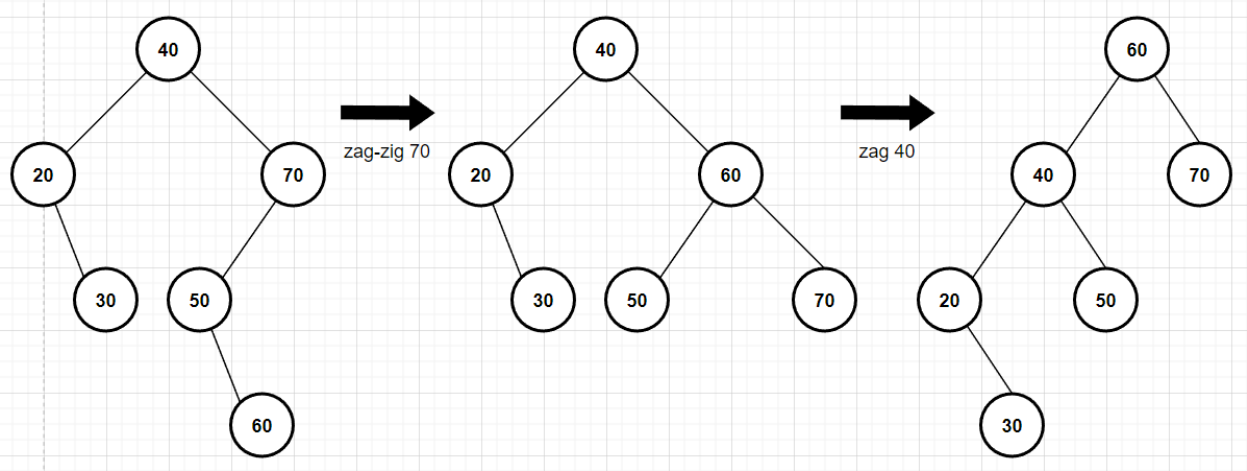
Bước 4: Thêm nút 20 vào cây. Vì nút mới thêm nhỏ hơn 30 nên sẽ được thêm vào cây con bên trái của nút 30. Sau đó thực hiện phép xoay Zig-Zig tại nút 50. Cuối cùng thực hiện phép xoay Zig tại nút 70 để đưa nút mới thêm lên làm nút gốc



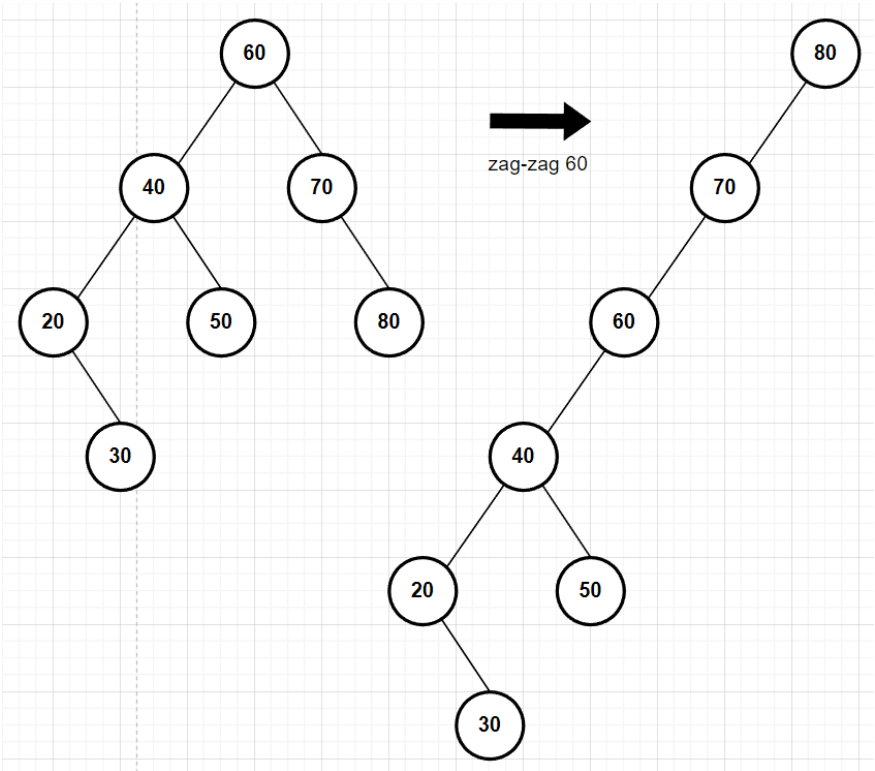
Bước 5: Thêm nút 40 vào cây. Vì nút mới nhỏ hơn 50 nên sẽ được thêm vào cây con bên trái của nút 50. Sau đó, thực hiện phép xoay Zig-Zag tại nút 30. Cuối cùng thực hiện phép xoay Zig-Zag tại nút 20 để đưa nút mới thêm lên làm nút gốc.



Bước 6: Thêm nút 60 vào cây. Vì nút mới lớn hơn 50 nên sẽ được thêm vào cây con bên phải của nút 50. Sau đó thực hiện phép xoay Zag-Zig tại nút 70. Cuối cùng thực hiện phép xoay Zag tại nút 40 để đưa nút mới thêm lên làm nút gốc.



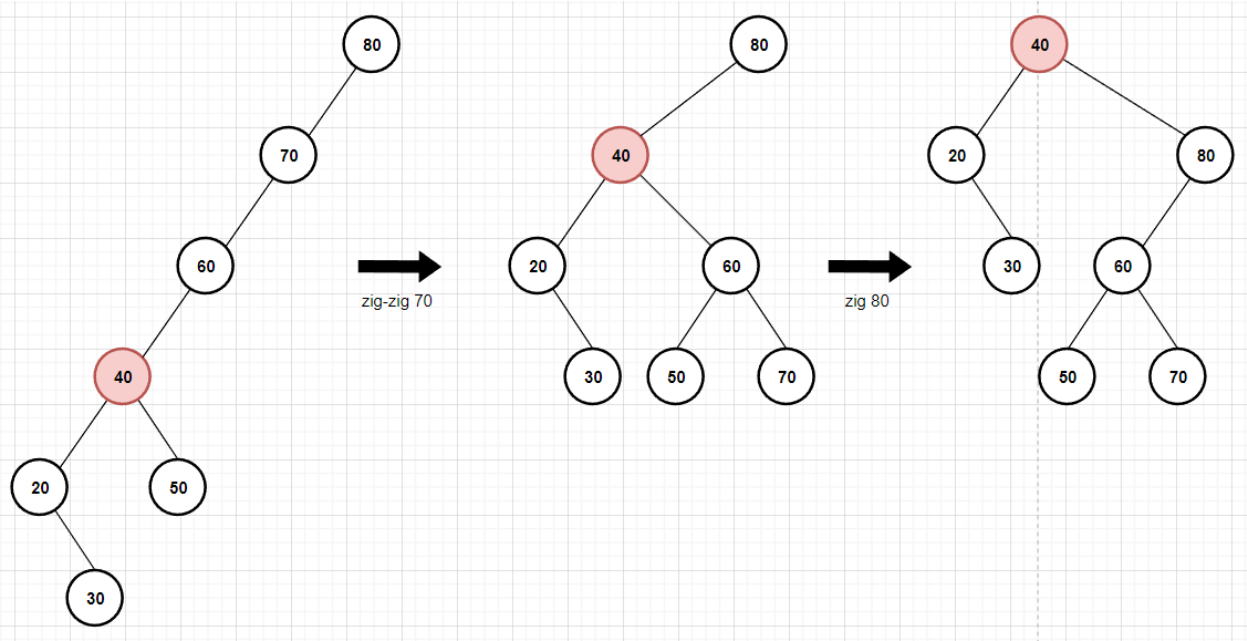
Bước 7: Thêm nút 80 vào cây. Vì nút mới lớn hơn 70 nên sẽ được thêm vào cây con bên phải của nút 70. Sau đó thực hiện phép xoay Zag-Zag tại nút 60 để đưa nút mới thêm lên làm nút gốc. Ta đã hoàn thành xây dựng cây.



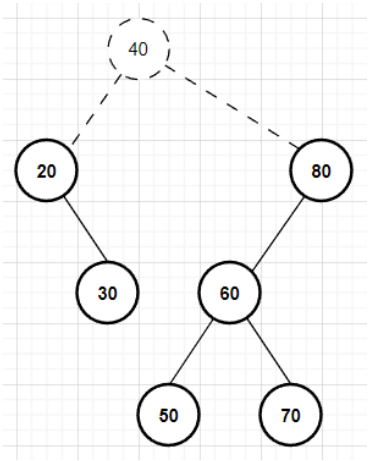
Tiếp theo, tôi sẽ minh hoạ tìm kiếm 1 node và xoá node bất kỳ trong cây Splay. Ở đây tôi sẽ sử dụng lại kết quả cây Splay phía trên và sẽ tiến hành xoá node có giá trị 40. Vì thực chất của việc xoá đã bao gồm tìm kiếm nên tôi sẽ không minh hoạ phần tìm kiếm node.

Bước 1: Thực hiện phép duyệt cây nhị phân tìm kiếm để tìm ra vị trí nút có chứa giá trị 40 (tôi sẽ không nói các trường hợp như nút gốc null hoặc không tìm được nút cần xoá vì ở phần trên tôi đã phân tích)

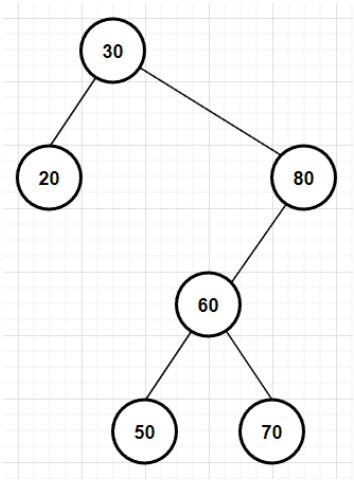
Bước 2: Tiến hành splay nút 40 lên làm nút gốc của cây.



Bước 3: Tiến hành xoá nút 40



Bước 4: Thực hiện phép xoay splay nút con bên trái nút 40 và gán cây con phải của nút đó cho nút 80. Khi này nút 30 trở thành nút gốc của cây.



Bước 5: Giải phóng nút đã xoá và trả về nút gốc hiện tại là nút 30.

1.3.2.Đánh giá độ phức tạp

Chi phí thêm một nút vào cây: $O(\log n)$.

Chi phí xoá một nút ra khỏi cây: $O(\log n)$.

Chi phí tìm kiếm: $O(\log n)$ trong trường hợp trung bình và $O(n)$ trong trường hợp xấu (tôi đã giải thích ở phần trên).

1.4. Kết luận (Conclusion)

Cây Splay cung cấp hiệu suất tốt cho các phép toán truy cập gần đây nhất. Việc splay nút gần đây nhất lên đỉnh cây giúp cải thiện hiệu suất cho các truy cập tiếp theo trên cùng một nút. Cây Splay tự cân bằng trong quá trình thực hiện các phép chèn, xoá và truy cập. Mặc dù cây Splay cung cấp hiệu suất tốt trong trường hợp trung bình, nó không đảm bảo độ phức tạp tối ưu trong trường hợp xấu nhất. Có thể xảy ra trường hợp cây trở thành một danh sách liên kết, khiến các phép toán trở nên chậm hơn. Cấu trúc của cây Splay đơn giản và dễ triển khai. Nó không đòi hỏi các thuật toán phức tạp như AVL trees hay red-black trees. Cây Splay được sử dụng rộng rãi trong nhiều lĩnh vực như caching (bộ nhớ cache), chỉ mục trong cơ sở dữ liệu, hệ thống tập tin, nén dữ liệu, xử lý văn bản, thuật toán đồ thị, trò chơi trực tuyến, ...

2.Splay Tree

2.1. Giới thiệu (Introduction)

2.1.1.Khái quát

Cây K-Dimensional (K-Dimensional Tree), hay còn gọi tắt là cây KD, là một cấu trúc dữ liệu được sử dụng để tổ chức điểm trong không gian K chiều. Nó là một dạng của cây nhị phân không tự cân bằng, trong đó mỗi nút đại diện cho một điểm trong không gian K chiều. Mỗi nút không phải là lá có thể được ngầm định tạo ra một siêu phẳng (hyperplane) phân chia không gian thành hai phần, được gọi là nửa không gian (half-spaces). Các điểm bên trái của siêu mặt phẳng này được biểu diễn bởi cây con bên trái của nút đó và các điểm bên phải của siêu mặt phẳng được biểu diễn bởi cây con bên phải.

2.1.2.Ưu điểm

Cây K-Dimensional có thể được sử dụng để giải quyết nhiều bài toán như tìm kiếm điểm gần nhất, tìm kiếm các điểm trong một khoảng cách xác định, xếp hạng và phân loại dữ liệu và hỗ trợ truy vấn phạm vi trong không gian K chiều.

2.1.3.Nhược điểm

Tuy cây K-Dimensional có nhiều ưu điểm, nhưng cũng có một số hạn chế. Trong trường hợp dữ liệu không cân đối, cây KD có thể trở nên không cân bằng, dẫn đến hiệu suất tìm kiếm kém. Đồng thời, cây KD cũng khá nhạy cảm với sự thay đổi về dữ liệu và không phản hồi tốt cho các truy vấn cập nhật.

2.1.4.Cải tiến (các biến thể)

- **Balanced KD Tree:** một vấn đề chính của cây KD là việc nó có thể trở nên không cân bằng khi dữ liệu không đồng đều. Một cách để giải quyết vấn đề này là sử dụng cây K-Dimensional cân bằng, như cây K-Dimensional cân bằng theo phương pháp median. Trong phương pháp này, việc chọn đường chia được thực hiện bằng cách lựa chọn phần tử trung vị của dữ liệu trong nút hiện tại.

- **Partitioned KD Tree:** là một biến thể của cây KD được sử dụng để tăng cường hiệu suất của cây KD trong việc xử lý dữ liệu có kích thước lớn. Trong Partitioned KD Tree, không gian dữ liệu được chia thành các phân vùng nhỏ hơn để giảm số lượng phép so sánh cần thiết trong quá trình tìm kiếm.
- **Randomized KD-Tree:** một cải tiến khác của cây KD là Randomized KD-Tree, trong đó việc chọn trục chia không cố định mà được chọn ngẫu nhiên. Điều này giúp tránh trường hợp xấu nhất khi dữ liệu đã được sắp xếp theo trục của cây.
- **KD-B+ Tree:** là sự kết hợp cây KD và cây B+ để tận dụng lợi thế của cả hai. Cây KD được sử dụng để tìm kiếm và xác định vị trí của các nút lá, trong khi cây B+ được sử dụng để lưu trữ các nút lá một cách hiệu quả hơn. Kết hợp này giúp tăng cường hiệu suất truy vấn và tiết kiệm bộ nhớ.
- **AKD-Tree (Adaptive KD-Tree):** là một biến thể của cây KD mà có khả năng tự điều chỉnh cấu trúc cây dựa trên dữ liệu. Thay vì cố định trục chia cho mỗi mức của cây, AKD-Tree điều chỉnh trục chia dựa trên sự phân bố của dữ liệu. Điều này giúp cân bằng cây và tăng cường hiệu suất truy vấn.
- **Cover Tree:** là một biến thể của cây KD được thiết kế đặc biệt cho việc xử lý không gian khoảng cách. Cover Tree tổ chức các nút dựa trên mức độ "phủ" của các nhánh con. Điều này giúp tìm kiếm các điểm gần nhất một cách hiệu quả hơn và giảm thời gian truy vấn.
- **Multithreaded KD Tree:** để tăng tốc độ tìm kiếm, một biến thể của cây KD có thể được thiết kế để hỗ trợ xử lý song song. Bằng cách sử dụng nhiều luồng (threads) để tìm kiếm đồng thời, cây KD có thể cải thiện hiệu suất trong các hệ thống đa luồng.

2.1.5.Ứng dụng

- **Tìm kiếm điểm gần nhất (nearest neighbor searches):** Cây KD rất hữu ích trong việc tìm kiếm điểm gần nhất trong không gian K chiều. Bằng cách sử dụng thuật toán tìm kiếm theo chiều dọc (recursive descent), cây KD có thể giúp tìm ra điểm gần nhất một cách hiệu quả. Ví dụ, trong lĩnh vực thị giác máy tính, cây KD có thể được sử dụng để tìm kiếm các điểm gần nhất đến một điểm ảnh trên hình ảnh.
- **Xử lý truy vấn đa chiều:** Trong các hệ thống có dữ liệu đa chiều như dữ liệu không gian, âm nhạc, hình ảnh và video, cây KD có thể hỗ trợ trong việc truy vấn dữ liệu một cách hiệu quả. Với khả năng tìm kiếm nhanh chóng và tổ chức dữ liệu theo không gian K chiều, cây KD giúp tăng cường hiệu suất và giảm thời gian truy

vấn. Ví dụ, trong lĩnh vực thị giác máy tính, cây KD có thể được sử dụng để tìm kiếm các đối tượng trong không gian 3D, như các đối tượng trong video game hoặc các đối tượng trong xử lý hình ảnh y tế.

- **Phân cụm dữ liệu (clustering):** Cây KD có thể được sử dụng để phân cụm dữ liệu, nơi các điểm dữ liệu được phân tán trên không gian đa chiều. Ví dụ, trong lĩnh vực khai phá dữ liệu, cây KD có thể được sử dụng để phân cụm các điểm dữ liệu trong không gian đa chiều.
- **Hình ảnh và đồ họa máy tính:** Trong lĩnh vực hình ảnh và đồ họa máy tính, cây KD được sử dụng để tìm kiếm và xử lý các đối tượng hoặc điểm trong không gian K chiều. Ví dụ, cây KD có thể được sử dụng để tìm kiếm các điểm gần nhất trong không gian 3D hoặc để phân loại các hình ảnh dựa trên đặc trưng của chúng.

2.2. Phương pháp (Method)

2.2.1.Khởi tạo

- Đầu tiên, chọn một điểm gốc từ tập dữ liệu ban đầu để làm nút gốc của cây KD. Điểm gốc này có thể được chọn ngẫu nhiên hoặc dựa trên một phương pháp khác như chọn điểm có giá trị lớn nhất hoặc trung vị trên mỗi chiều.
- Tiếp theo, dữ liệu trong tập dữ liệu ban đầu được sắp xếp theo giá trị của trục phân chia tương ứng với nút hiện tại. Điều này giúp chia dữ liệu thành hai phần đối xứng tương ứng với các cây con bên trái và bên phải.
- Chọn một điểm phân chia trong tập dữ liệu đã sắp xếp. Điểm phân chia này thường được chọn là điểm ở giữa (median) của dữ liệu theo trục phân chia. Điểm phân chia này sẽ trở thành nút hiện tại và được sử dụng để tạo siêu mặt phẳng phân chia không gian.
- Sau khi chọn điểm phân chia, dữ liệu được chia thành hai phần tương ứng với các cây con bên trái và bên phải. Quá trình này được thực hiện đệ quy bằng cách lặp lại các bước từ 2 đến 4 cho mỗi cây con.
- Quá trình đệ quy dừng lại khi không còn dữ liệu nào hoặc khi đạt đến một điều kiện dừng khác như đạt đến độ sâu tối đa cho phép hoặc số lượng điểm dữ liệu nhỏ hơn một ngưỡng nhất định.
- Khi cây KD được xây dựng, cần lưu trữ thông tin của mỗi nút gồm điểm phân chia, các cây con và các thông tin bổ sung khác như vùng không gian bao quanh nút.

Sau đây là mã giả của thuật toán khởi tạo cây KD (Nguồn Internet)

Algorithm 1: K-D Tree Construction

Function *kdtree(pointList, depth)*:
Input: a list of points, *pointList*, and an integer, *depth* ;
Output: a k-d tree rooted at the median point of *pointList* ;

```
/* Select axis based on depth so that axis cycles
   through all valid values                          */
```
let $axis := depth \mod k$;

```
/* Sort point list and choose median as pivot element
   */
```
select median by *axis* from *pointList*;

```
/* Create node and construct subtree                 */
```
let *node.location* := *median*;
let *node.leftChild* := kdtree(points in *pointList* before *median*, $depth + 1$);
let *node.rightChild* := kdtree(points in *pointList* after *median*, $depth + 1$);

return *node*;

2.2.2.Thêm

- Bắt đầu từ gốc của cây, xác định chiều (axis) hiện tại dựa trên độ sâu (depth) của nút hiện tại trong cây. Chiều này sẽ giúp quyết định nút mới sẽ nằm ở phía trái hay phải của nút hiện tại.
- So sánh giá trị của chiều (axis) của nút mới với giá trị của nút hiện tại để xác định xem nút mới sẽ nằm bên trái hay bên phải.
- Nếu nút con tương ứng với hướng di chuyển đã được xác định không tồn tại (null), tôi tạo một nút mới và gán cho nút con đó.
- Nếu nút con tương ứng đã tồn tại, ta tiếp tục đệ quy gọi phương pháp thêm nút vào nút con đó, tăng giá trị độ sâu lên 1.
- Lặp lại quá trình từ bước 1 cho đến khi tìm được vị trí phù hợp để chèn nút mới vào cây.

2.2.3.Xoá

(tương tự xoá một nút trong cây nhị phân tìm kiếm)

- Bắt đầu từ gốc của cây, tìm kiếm nút cần xoá. Để tìm nút này, di chuyển cây KD theo chiều tương ứng với giá trị của nút cần xoá và so sánh nút hiện tại với nút cần xoá.
- Khi tìm thấy nút cần xoá, tôi xác định các trường hợp sau:
    - Nút cần xoá không có nút con bên trái hoặc bên phải: xoá nút này trực tiếp.
    - Nút cần xoá có một trong hai nút con (bên trái hoặc bên phải): thay thế nút cần xoá bằng nút con đó.
    - Nút cần xoá có cả hai nút con: tìm kiếm nút thế thế cho nút cần xoá. Thông thường, nút thế thế là nút có giá trị gần nhất với nút cần xoá trong cây con bên phải của nút cần xoá.
- Xoá nút cần xoá theo các trường hợp đã xác định ở bước 2.
- Nếu nút cần xoá có hai nút con và đã tìm thấy nút thế thế, thay thế nút cần xoá bằng nút thế thế và tiếp tục quá trình xoá với nút thế thế.
- Lặp lại quá trình từ bước 1 cho đến khi nút cần xoá được xoá khỏi cây.

2.2.4.Tìm kiếm gần nhất (Nearest Neighbour Search)

- Bắt đầu từ gốc của cây, di chuyển qua cây KD dựa trên giá trị của nút hiện tại và so sánh với nút mục tiêu (nút cần tìm gần nhất).
- Theo chiều di chuyển, tôi so sánh giá trị của nút hiện tại với giá trị của nút mục tiêu trong cùng chiều đó. Dựa trên kết quả so sánh, quyết định di chuyển sang nút con bên trái hoặc nút con bên phải.
- Đệ quy gọi phương pháp tìm kiếm với nút con tương ứng (trái hoặc phải) của nút hiện tại.
- Tại mỗi nút, tôi tính khoảng cách giữa nút hiện tại và nút mục tiêu. Nếu khoảng cách nhỏ hơn khoảng cách nhỏ nhất đã tìm thấy cho đến hiện tại, cập nhật nút gần nhất và khoảng cách nhỏ nhất.
- Sau khi đã xem xét tất cả các nhánh con và nút hiện tại, kiểm tra xem có cần tiếp tục tìm kiếm ở nhánh con bên kia (nhánh con không được xem xét). Quyết định này dựa trên việc xem xét khoảng cách giữa giá trị của nút hiện tại và giá trị của nút mục tiêu trên chiều tương ứng.
- Lặp lại quá trình từ bước 1 cho đến khi đã xem xét tất cả các nhánh con và quay trở lại nút gốc của cây.

2.3. Cài đặt (Experiment)

2.3.1.Bài toán minh hoạ

Bài toán minh hoạ khởi tạo cây KD từ các node cho trước

Giả sử tôi có một cây KD (với k = 2) gồm 7 nút có giá trị như sau: [(3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)]. Sau đây tôi sẽ minh hoạ từng bước xây dựng một cây KD.

Trong bài toán minh hoạ này đa số các nguồn tài liệu mà tôi tham khảo được trên mạng đa số sẽ dùng phương pháp chọn phần tử đầu tiên khi được thêm vào cây làm nút gốc, tuy nhiên phương pháp trên không đảm bảo cây sẽ cân bằng khi khởi tạo cây từ một số nút cho trước, trong trường hợp xấu cây mất cân bằng dẫn đến việc tìm kiếm không được tối ưu. Thay vào đó, tôi sẽ tìm ra điểm trung vị chọn làm nút gốc để đảm bảo cây KD của tôi cân bằng ít nhất là khi vừa khởi với một số nút có giá trị cho trước. Tuy nhiên, với phương pháp chọn điểm trung vị sẽ có tổng độ phúc tạp khi khởi tạo cây là sẽ cao hơn so với phương pháp chọn nút đầu tiên làm gốc (tôi sẽ nói rõ hơn ở phần sau).

Bước 1: Chọn trục và sắp xếp danh sách ban đầu

- Vì cây KD trong minh hoạ là cây 2D nên k sẽ có giá tri là 2 và độ sâu (depth) khi này mới khởi tạo là 0 nên asix = depth % k = 0, tức là trục x.
- Sắp xếp danh sách theo giá trị trục x: [(2, 7), (3, 6), (6, 12), (9, 1), (10, 19), (13, 15), (17, 15)].

Bước 2: Chọn phần tử trung vị

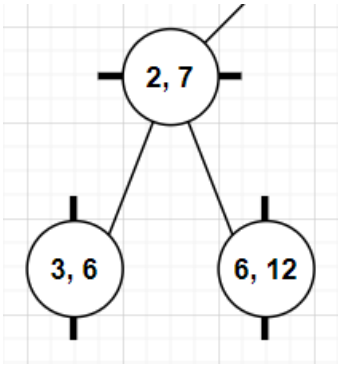
- Phần tử trung vị là median = (9, 1).

Bước 3: Xây dựng nút gốc và các nút con

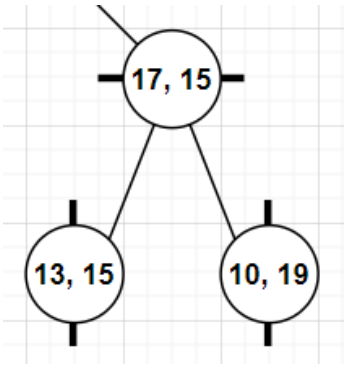
- Tạo một nút gốc và gán vị trí của nó tại nút trung vị (9, 1).
- Tạo nút con bên trái và nút con bên phải giá trị null.

Bước 4: Đệ quy xây dựng cây KD cho danh sách con bên trái và bên phải

- Với danh sách con bên trái: [(2, 7), (3, 6), (6, 12)]
    - Gọi lại phương thức kdtree với danh sách con bên trái và depth + 1 = 1.
    - Sắp xếp danh sách con bên trái theo giá trị trục y: [(3, 6), (2, 7), (6, 12)].
    - Chọn phần tử trung vị: median = (2, 7).
    - Xây dựng nút gốc và các nút con cho danh sách con bên trái.
    - Kết quả là một cây KD như sau:



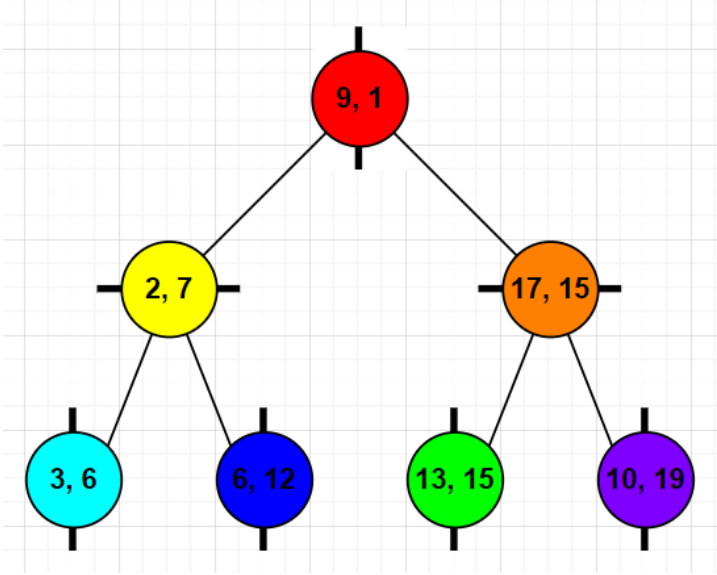
- Với danh sách con bên phải: [(10, 19), (13, 15), (17, 15)]
  - Gọi lại phương thức kdtree với danh sách con bên phải và depth + 1 = 1.
  - Sắp xếp danh sách con bên phải theo giá trị trục y: [(13, 15), (17, 15), (10, 19)].
  - Chọn phần tử trung vị: median = (17, 15).
  - Xây dựng nút gốc và các nút con cho danh sách con bên phải.
  - Kết quả là một cây KD như sau:

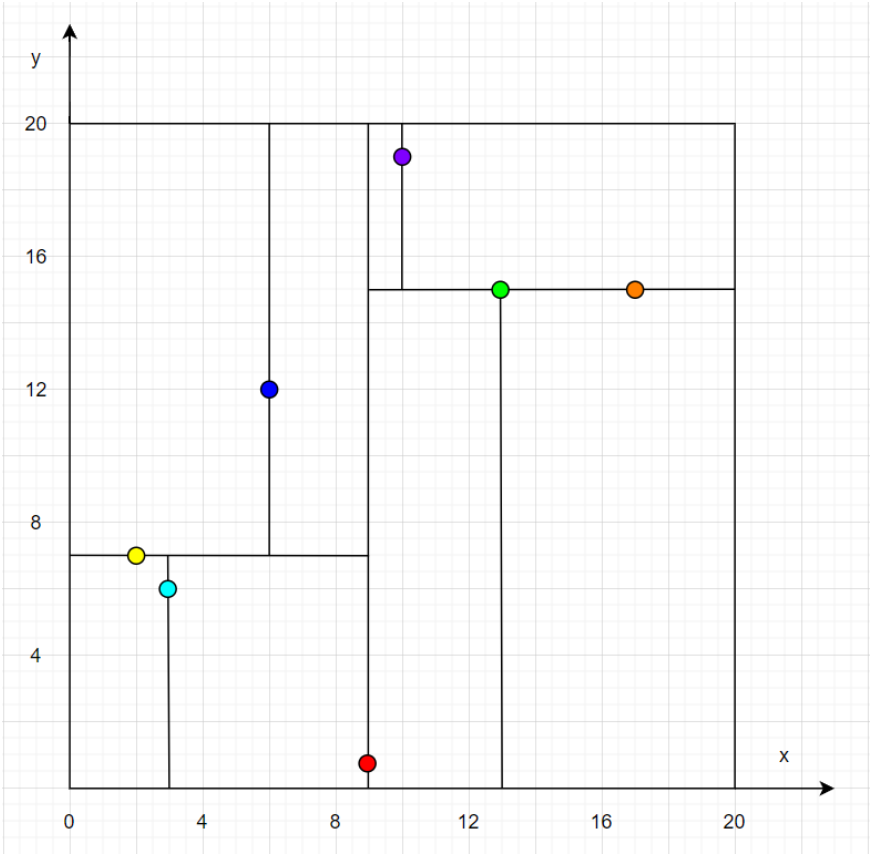


Bước 5: Gán giá trị nút con bên trái và nút con bên phải

- Gán nút con bên trái bằng cây KD đã được xây dựng từ danh sách con bên trái.
- Gán nút con bên phải bằng cây KD đã được xây dựng từ danh sách con bên phải.

Kết quả cây KD sau khi được khởi tạo với 7 nút có giá trị cho trước:





Bài toán minh hoạ thêm một node trong cây KD

Tiếp tục với với bài toán minh hoạ trên, sau đây tôi sẽ minh hoạ từng bước thêm một nút có giá trị (8, 5) vào cây KD.

Bước 1: Bắt đầu từ nút gốc (9, 1)

- Vì (8, 5) có giá trị trục x (8) nhỏ hơn giá trị trục x của nút gốc (9), tôi đi xuống nút con bên trái.

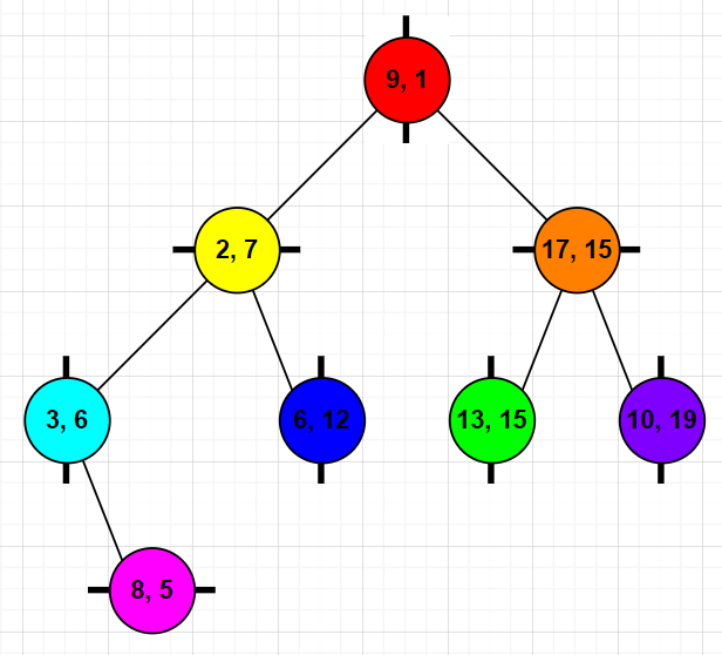
Bước 2: Đến nút con (2, 7)

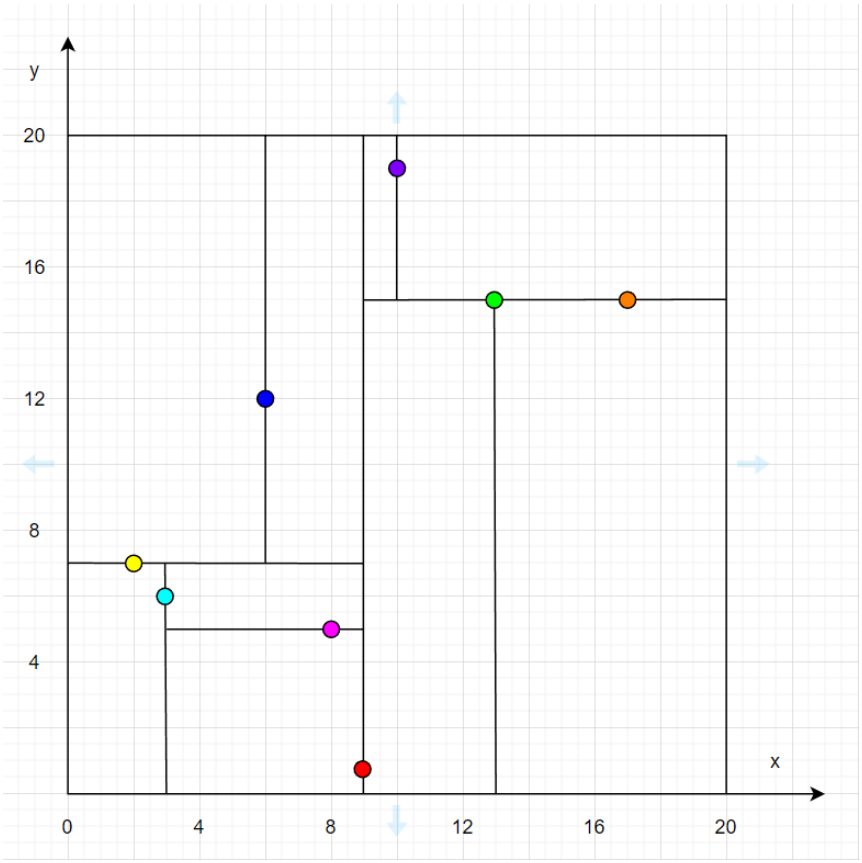
- Vì (8, 5) có giá trị trục y (5) nhỏ hơn giá trị trục y của nút con (7), tôi đi xuống nút con bên trái của nút con này.

Bước 3: Đến nút lá (3, 6)

- Vì (8, 5) có giá trị trục x (8) lớn hơn giá trị trục x của nút lá (3), tôi đi xuống nút con bên phải của nút lá này.

Kết quả cây KD sau khi tiếp tục thêm nút có giá trị (8, 5):





Bài toán minh hoạ xoá một node trong cây KD

Cuối cùng, tôi sẽ minh hoạ từng bước xoá một nút có giá trị (6, 12) trong cây KD ở trên:

Ở phần này, xoá một nút trong cây KD tương tự như xoá một nút trong cây BST, có 3 trường hợp là nút cần xoá có 1 con, 2 con hoặc không có con nào. Tôi sẽ không minh hoạ hết cả 3 trường hợp này và tôi sẽ hợp xoá nút không có con để dễ dàng minh hoạ.

Bước 1: Bắt đầu từ nút gốc (9, 1)

- Vì (6, 12) có giá trị trục x (6) nhỏ hơn giá trị trục x của nút gốc (9), tôi đi xuống nút con bên trái.

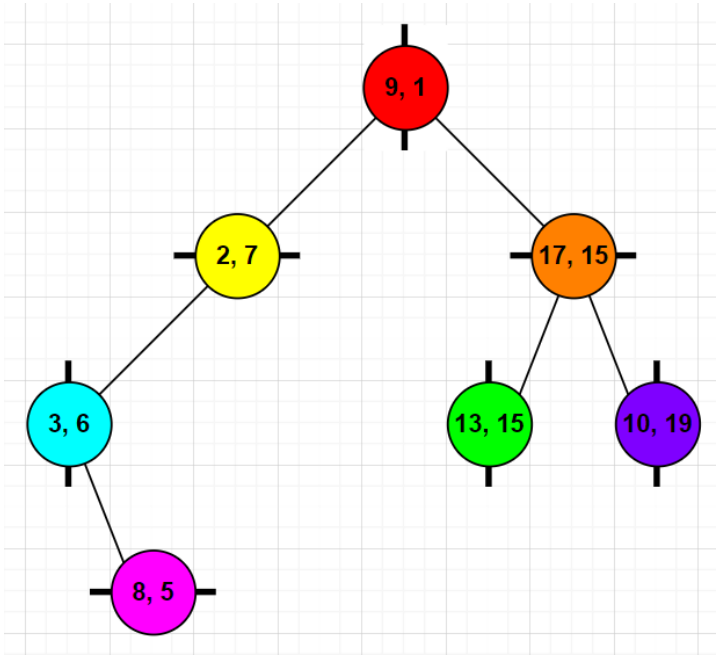
Bước 2: Đến nút con (2, 7)

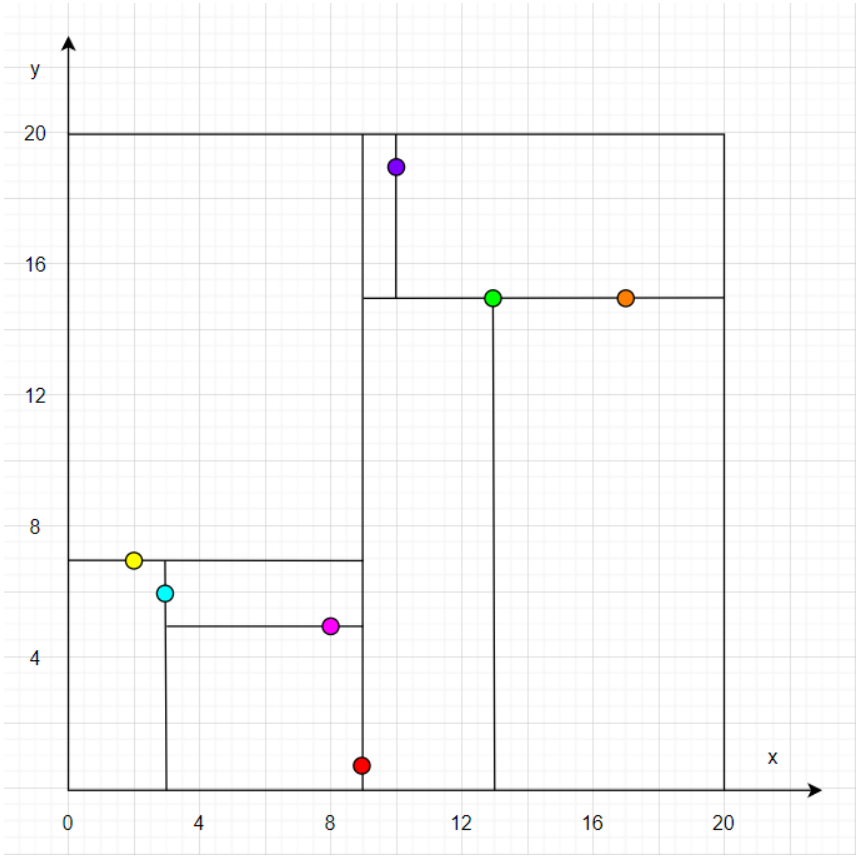
- Vì (6, 12) có giá trị trục x và trục y khớp với giá trị của nút lá này, tôi xử lý xoá node này.

Bước 3: Xoá nút lá (6, 12)

- Vì nút lá này không có nút con bên trái hoặc bên phải, tôi xoá nút lá này khỏi cây.

Kết quả cây KD sau khi tiếp tục xoá nút có giá trị (6, 12):





2.3.2.Đánh giá độ phức tạp

Chi phí khởi tạo cây: $O(n\log^2 n)$.

Điều này là do thuật toán sắp xếp trong việc tìm kiếm trung vị có chi phí là $O(n\log n)$. Và cây KD là cây nhị phân nên có số bậc là $O(\log n)$. Vậy tổng độ phức tạp sẽ trở thành $O(n\log^2 n)$.

Chi phí thêm một nút vào cây: $O(\log n)$

Chi phí xoá một nút ra khỏi cây: $O(\log n)$

Chi phí tìm kiếm gần nhất (Nearest Neighbour Search): $O(\log n)$

2.4. Kết luận (Conclusion)

Cây KD (K-Dimensional Tree) là một cấu trúc dữ liệu cây nhị phân có khả năng phân cụm được sử dụng để tổ chức và tìm kiếm dữ liệu trong không gian đa chiều. Cây KD được sử dụng rộng rãi trong nhiều lĩnh vực, đặc biệt là thị giác máy tính và khai thác dữ liệu. Dựa trên nguyên lý chia vùng không gian và sắp xếp dữ liệu theo từng trục, cây KD cung cấp khả năng tìm kiếm nhanh và hiệu quả trong không gian K chiều. Ngoài ra, cây KD cho phép tìm kiếm nhanh chóng một nút gần nhất hoặc trong một phạm vi cho trước. Tuy nhiên, vì cây KD tương tự cây nhị phân tìm kiếm không tự cân bằng, có thể cải tiến thành cây nhị phân cân bằng nhưng chi phí cho phép quay cây là tương đối cao nếu chèn hoặc xoá một nút xảy ra với tần suất cao vì vậy ta nên cân nhắc khi sử dụng cho bộ dữ liệu thường xuyên bị thay đổi.