TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN LẬP TRÌNH TÍNH TOÁN

XÂY DỰNG THƯ VIỆN, MÔ PHỎNG CÂY TÌM KIẾM NHỊ PHÂN

Người hướng dẫn**: Nguyễn Thị Lệ Quyên**
Sinh viên thực hiện**:**
Nguyễn Vũ Khánh Uy **LỚP: 21TCLC_DT3 NHÓM: 1**
Mai Trịnh Xuân Quý **LỚP: 21TCLC_DT3 NHÓM: 1**

Đà Nẵng, 06/2022

MỤC LỤC

DANH MỤC HÌNH VẼ

MỞ ĐẦU

Trong việc xây dựng các hệ thống phần mềm, ta thường xuyên gặp những bài toán theo dạng kiểm tra một phần tử có thuộc một tập hợp không, thêm phần từ đó vào, đếm xem có bao nhiêu phần tử trong tập hợp.. Các thao tác trên có nhiều cách để xử lý tùy theo tính chất bài toán nhưng để xử lí hiệu quả đối với một bộ dữ liệu lớn và tổng quát ta cần một thuật toán tối ưu cho bài toán đó.

Đề tài sẽ nghiên cứu xây dựng(bằng ngôn ngữ C) và ứng dụng cây tìm kiếm nhị phân vào giải quyết bài toán trên, cải tiến thành một cây nhị phân cân bằng (cụ thể là cây đỏ đen) để xử lí trường hợp dữ liệu làm cây bị suy biến. Bổ sung các tính năng tương tự như những cây tìm kiếm nhị phân hiện nay(cụ thể là cấu trúc set/map trong c++) và chuyển thành file header (.h) để dùng như thư viện bình thường.

Phương pháp nghiên cứu sẽ là nghiên cứu các nguồn tài liệu trên mạng và thay đổi để giải các bài toán trên các online judge(cụ thể là CSES).

1. TỔNG QUAN ĐỀ TÀI

Hãy cùng xét bài toán trên và bàn luận về sự cần thiết của ứng dụng cây tìm kiếm nhị phân!

Bài toán: Xây dựng một hệ thống xử lý 3 loại truy vấn:

- Thêm một phần tử vào tập hợp nếu chưa thêm.
- Xóa một phần tử khỏi tập hợp nếu có.
- In ra số lượng phần tử trong tập hợp.

Hướng giải tự nhiên: Mỗi lần thêm thì duyệt qua cả tập hợp xem có tồn tại chưa, nếu chưa thì thêm. Mỗi lần xóa thì tìm xem có phần tử đó không và xóa nếu có. Thao tác in ra số lương phần tử thì có thể duyệt một lần nữa để đếm hoặc cập nhật theo mỗi lần thêm/xóa ở trên. Độ phức tạp O(n*q) với n là số phần tử có thể có trong tập hợp và q là số truy vấn => Không phù hợp với bộ dữ liệu lớn (Vd facebook muốn làm hệ thống đếm xem có bao nhiêu người dùng ở khu vực châu Á đang online.. thì số bước tính sẽ rất lớn).

Hướng giải trả lời sau khi đọc hết truy vấn và số lượng các phần tử trong các truy vấn không quá đa dạng: Lưu lại các phần tử của mỗi thao tác thêm/xóa và một mảng, sắp xếp theo thứ tự tăng dần và gán nhẫn các phần tử giống nhau bằng các nhẫn là các số nguyên tăng dần từ 1. Như vậy các giá trị phần tử của truy vấn giờ sẽ được nén xuống một số không quá lớn và ta có thể dung một mảng để dánh dấu liệu phần tử đó có ở trong tập hợp hay không. Nhờ vậy ta có thể xử lý mỗi truy vấn trong O(1) nhưng đổi lại phải tiền xử lý trong O(q*logq) với q là số lượng các truy vấn thêm/xóa. => Điều này không phải lúc nào cũng làm được vì có nhiều bài toán yêu cầu phải xử lý ngay sau khi đọc truy vấn và độ đa dạng các phần tử có thể nhiều (như vd facebook ở trên ta có thể thấy số người dung châu á có thể đạt mức rất cao > 10^9 người dung hoặc hệ thống cần xử lí thời gian thực..)

Qua 2 ví dụ trên ta đã có thể thấy có nhiều bài toán thực tế yêu cầu việc xử lý các truy vấn trên trong thời gian thực với độ phức tạp đủ tốt. Cây nhị phân tìm kiếm là cấu trúc dữ liệu có thể giải quyết bài toán trên với độ phức tạp mỗi truy vấn là O(logn) với n là số phần tử có thể có trong tập hợp và nó không cần bước tiền xử lí => Độ phức tạp tổng quát O(qlogn).

2. CƠ SỞ LÝ THUYẾT

2.1. Ý tưởng

Ý tưởng ban đầu là lưu các phần tử trong tập hợp trong một cây nhị phân với các tính chất:

- Nút con bên trái nhỏ hơn nút đang xét hiện tại => **Mọi nút con nằm bên nhánh bên trái đều nhỏ hơn nút hiện tại.(1)**
- Nút con bên phải lớn hơn nút đang xét hiện tại => **Mọi nút con nằm bên nhánh bên phải đều lớn hơn nút hiện tại.(2)**

Từ cách lưu trên ta có thể xử lí các thao tác thêm/xóa như sau:

- **Nếu phần tử cần thực hiện thao tác bằng phần tử đang xét:** Nếu đang cần xóa thì ta sẽ xóa và cố giữ lại các tính chất như đã nói ở trên. Nếu đang cần thêm thì kết thúc vì phần tử đã ở trong tập hợp từ trước.
- **Nếu phần tử đang xét là rỗng(đã duyệt hết chiều sâu của cây):** Nếu đang cần xóa thì ta kết thúc vì phần tử đó không tồn tại. Nếu đang cần thêm thì t chỉ cần thêm nó vào vị trí đó.
- **Nếu phần tử cần thực hiện thao tác nhỏ hơn phần tử đang xét:** Duyệt nhánh con bên trái vì vị trí cần thực hiện thao tác chỉ có thể nằm bên nhánh trái(theo tính chất 1 ở trên).
- **Nếu phần tử cần thực hiện thao tác lớn hơn phần tử đang xét:** Duyệt nhánh con bên phải vì vị trí cần thực hiện thao tác chỉ có thể nằm bên nhánh phải(theo tính chất 2 ở trên).

Cơ bản thì các thao tác thêm/ xóa sẽ tìm vị trí cần thực hiện thao tác trước sau đó đưa ra quyết định.

Còn thao tác tìm số lượng phần tử thì khá dễ và có thể cập nhật theo mỗi lần thêm/xóa.

Từ ý tưởng trên ta thấy việc thực hiện mỗi truy vấn sẽ phải đi qua tối đa h vị trí với h là chiều cao có thể của cây nên độ phức tạp trung bình sẽ là O(logn) mỗi truy vấn và O(qlogn) đối với bài toán trên của ta.

Cải tiến: có thể thấy nếu các phần tử được thêm vào cây có thứ tự tăng dần hoặc giảm dần thì cây sẽ bị mất cân bằng và có độ sâu rất lớn => Các thao tác thêm phần tử vào nhánh cây bị mất cân bằng sẽ phải đi qua nhiều phần tử mới tới được vị trí cần thêm => Độ phức tạp trường hợp xấu nhất của một cây tìm kiếm nhị phân bình thường có thể

là O(n^2). Để khắc phục điều đó thì ta cần thêm/xóa các phần tử sao cho cây vẫn giữ độ cao không vượt quá xa logn. Điều đó có thể thực hiện bằng cây đỏ đen.

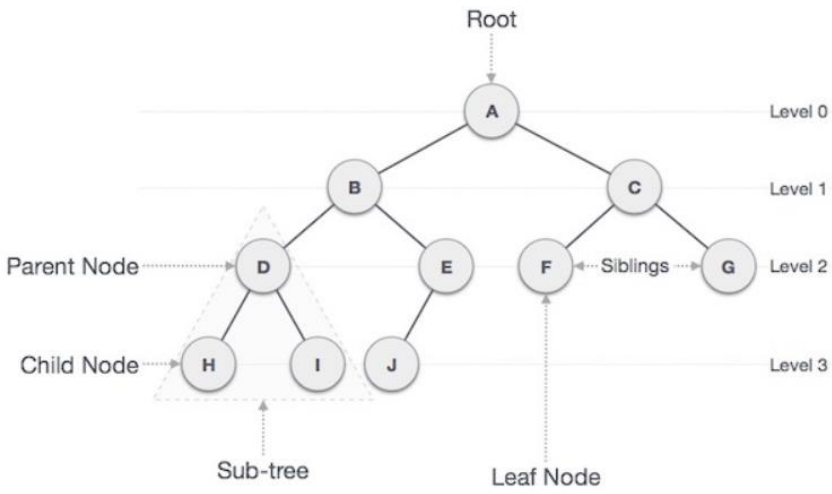
Các tính chất của cây đỏ đen:

- Một nút hoặc là đỏ hoặc là đen.
- Gốc là đen.
- Tất cả các lá (NULL) là đen.
- Cả hai con của mọi nút đỏ là đen => mọi nút đỏ có nút cha là đen.
- Tất cả các đường đi từ một nút bất kỳ tới các lá có số nút đen bằng nhau.

Từ các tính chất trên suy ra trong các đường đi từ gốc tới các lá đường đi dài nhất không vượt quá hai lần đường đi ngắn nhất. Giải thích là vì số nút đen trên đường đi bằng nhau nên đường đi dài nhất chính là đường đi có nút đỏ đen xen kẽ và đường đi ngắn nhất là đường đi toàn nút đen.

Do đó cây đỏ đen là gần cân bằng. Vì các thuật toán chèn, xóa, tìm kiếm trong trường hợp xấu nhất đều tỷ lệ với chiều cao của cây nên cây đỏ đen rất hiệu quả trong các trường hợp xấu nhất. Cụ thể độ phức tạp sau khi cải tiến thành cây đỏ đen sẽ ổn định là O(logn) mỗi truy vấn và O(qlogn) đối với bài toán trên.

2.2. Cơ sở lý thuyết



Hình 1: Ảnh về cấu trúc cây

Cấu trúc cây (Tree) là một tập hợp các phần tử gọi là nút (node), mỗi cây có một nút gốc (root) chứa nhiều nút con, mỗi nút con lại là một tập hợp các nút khác gọi là cây con (subtree).

Bậc của nút: là số nút con của nút đó.

Bậc của cây: là bậc lớn nhất của nút trong cây đó, cây bậc n sẽ được gọi là cây n – phân.

Nút lá: nút lá là nút có bậc bằng 0.

Nút nhánh: là nút có bậc khác 0 mà không phải nút gốc (hay còn gọi là nút trung gian).

Mức của nút: là số nguyên đếm từ 0, các nút ngang hàng nhau thì có cùng mức.

Chiều cao (chiều sâu): là mức lớn nhất của các nút lá.

Độ dài đường đi đến nút x: là số nhánh (cạnh nối hai nút) cần đi qua tính từ nút gốc đến nút x. Hay độ dài đường đi đến nút mức i chính là i.

3. TỔ CHỨC CẤU TRÚC DỮ LIỆU VÀ THUẬT TOÁN

3.1. Phát biểu bài toán

Input: Đọc từ file data.txt một số nguyên q – số lương truy vấn, q dòng tiếp theo trong file trên là các truy vấn có một trong những dạng sau:

- 1 a b: Thêm vào hoặc ghi đè giá trị của phần từ trong tập hợp có khóa a và giá trị là b.
- 2 a: Xóa phần tử có khóa a nếu có khỏi tập hợp.
- 3 a: Kiểm tra xem có phần tử khóa a trong tập hợp không.

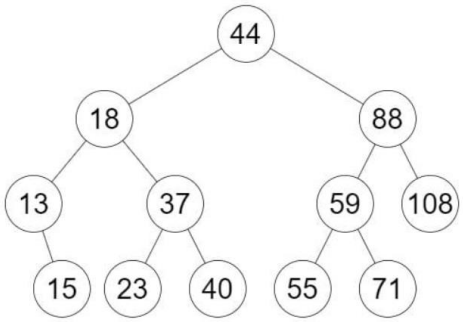
Output: In ra màn hình mỗi truy vấn một dòng:

- Nếu đó là truy vấn loại 1 và 2 thì in ra số lượng phần tử sau khi thực hiện truy vấn đó.
- Nếu đó là truy vấn loại 3 thì in "YES" nếu có và "NO" ngược lại.

Ngoài bài toán nhưng cần in ra do đề tài yêu cầu mô phỏng thao tác của cây và xây dựng thư viện (phần này sẽ không tính vào độ phức tạp về sau):

- Với mỗi truy vấn in ra các nút của cây theo thứ tự duyệt trước (NLR).
- Dòng cuối in ra các phần tử trong tập theo thứ tự tăng dần của khóa.

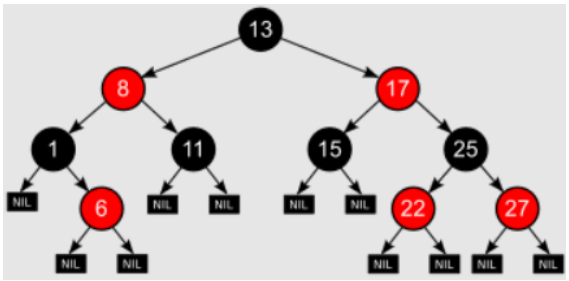
3.2. Cấu trúc dữ liệu



Hình 2: Ảnh về cấu trúc câu tìm kiếm nhị phân bình thường

Cây nhị phân là một trường hợp đặc biệt của cấu trúc cây và nó cũng phổ biến nhất. Đúng như tên gọi của nó, cây nhị phân có bậc là 2 và mỗi nút trong cây nhị phân đều có bậc không quá 2.

Cây nhị phân cân bằng: số phần tử của cây con bên trái chênh lệch không quá 1 so với cây con bên phải.



Hình 3: Ảnh về cấu trúc cây đỏ đen

Cây đỏ đen (red-black tree) là một dạng cây tìm kiếm nhị phân tự cân bằng. Nó là cấu trúc phức tạp nhưng cho kết quả tốt về thời gian trong trường hợp xấu nhất. Các phép toán trên chúng như tìm kiếm (search), chèn (insert), và xóa (delete) trong thời gian O (log n), trong đó n là số các phần tử của cây.

3.3. Thuật toán

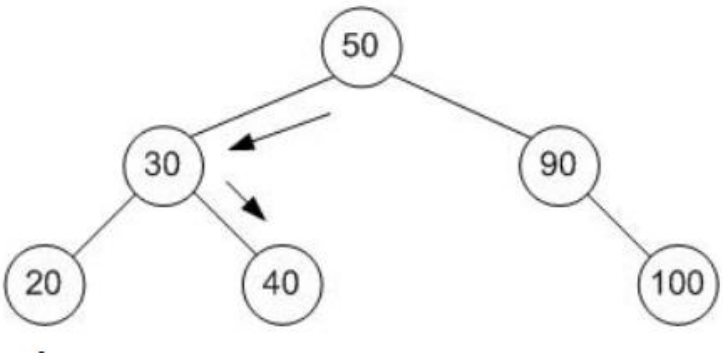
Như đã nói ở trên, về việc xây dựng thư viện cây nhị phân tìm kiếm ta sẽ cố bổ sung các tính năng tương tự như cấu trúc set/map của c++ và thêm hàm mô phỏng cấu trúc cây. Vì vậy cây tìm kiếm nhị phân của chúng ta sẽ có những hàm chức năng như sau:

- addNode(key, value): thêm vào hoặc ghi đè phần tử trong cây có khóa là key, giá trị là value.
- delNode(key): xóa phần tử có khóa key nếu có trong cây.

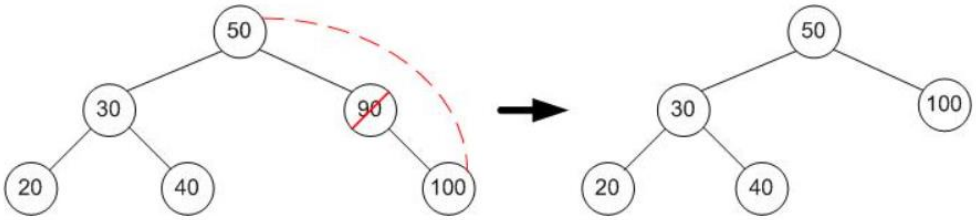
- searchNode(key): trả về con trỏ trỏ tới phần tử có khóa key, nếu không có trả về NULL.
- nextNode(node): trả về con trỏ trỏ tới phần tử tiếp theo theo độ tăng dần của khóa của phần tử node trong cây, nếu không có trả về NULL.
- getBeginNode(): trả về con trỏ trỏ tới phần tử có khóa nhỏ nhất.
- showTree(): in ra cây trên một dòng theo thứ tự duyệt trước (NLR).
- getSize(): trả về số lượng phần tử của cây.

Thuật toán được sử dụng cho những hàm trên:

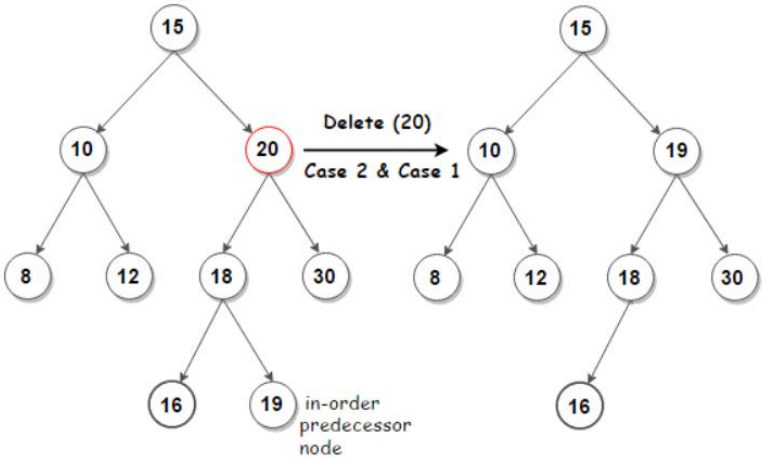
- searchNode: B1: Đăt u là đỉnh gốc. B2: Nếu u bằng khóa cần tìm hoặc NULL thì thoát. B3: Nếu u có khóa nhỏ hơn khóa cần tìm thì đặt u bằng nút con bên phải của u, ngược lại đặt u là nút con bên trái. B4: Quay lại B2. Độ phức tạp: O(h) với h là độ cao của cây.
- addNode: làm như hàm searchNode ở trên, nếu có tìm thấy thì thay đổi giá trị của nút đó và thoát, không thì thêm nút mới. Độ phức tạp O(h).
- delNode: làm như hàm searchNode ở trên, nếu không tìm thấy thì thoát, nếu có thì đặt nút tìm thấy là u. Nếu u không có nút con thì chỉ cần xóa, nếu có 1 con thì chỉ cần dịch nút con lên thế chỗ u, nếu có 2 nút con thì lấy nút có khóa cao nhất trong nhánh cây con của nút trái thế chỗ cho u. Độ phức tạp O(h).
- getBeginNode: cứ đi về trái đến khi nút con bên trái là NULL thì dừng. Độ phức tạp O(h).
- getSize: có một biến lưu số lượng phần tử của cây, cập nhật biến đó mỗi thao tác xóa/ thêm nút.
- nextNode:B1: Đặt u là nút hiện tại. B2: Nếu u có nút phải thì tìm phần tử nhỏ nhất trong nhánh cây nút phải đó và thoát. B3: Nếu u bằng NULL hoặc u là nút trái của nút cha của u thì thoát. B4: Gán u bằng cha của u và quay lại bước 3. Độ phức tạp O(h).
- showTree: Với nút u hiện tại thì in ra thông tin nút u sau đó đi thăm hết cây con trái rồi đến cây con phải. Độ phức tạp là O(n) với n là số nút của cây (thực ra đây là hàm mô phỏng và không nằm trong bài toán).



Hình 4: Ảnh mô phỏng thao tác tìm kiếm nút có khóa 40 trong cây



Hình 5: Ảnh mô phỏng thao tác xóa nút có khóa 90 (nút có 1 con)



Hình 6: Ảnh mô phỏng thao tác xóa nút có khóa 20(nút có 2 con)

Đối với cây đỏ đen mục đích của ta chỉ là để kiểm nghiệm tốc độ sau khi cải tiến so với cây tìm kiếm nhị phân bình thường nên ta chỉ có 3 hàm thao tác cơ bản:

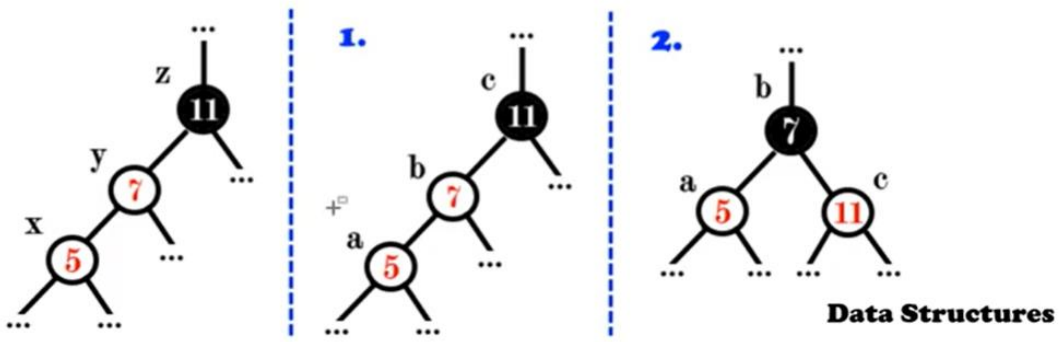
- addElement(key): thêm nút có khóa key.
- delElement(key): xóa nút có khóa key nếu có.
- getSize(): trả về số lượng nút trong cây.

Thuật toán cho những hàm trên:

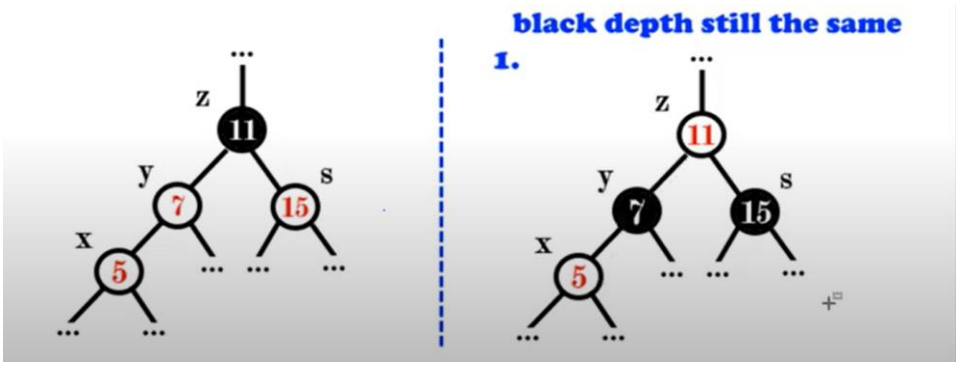
- addElement: B1: làm như cây tìm kiếm nhị phân bình thường, tìm đến vị trí cần thêm nút. Đó là một nút lá NULL nào đó, gọi nút đó là x. B2: Kiểm tra nếu cây rỗng thì thêm nút mới làm gốc và chọn màu cho nó là màu đen

sau đó thoát. B3: Thêm giá trị vào nút x đó và đặt x màu đỏ. B4: Đặt a là nút x, b là nút cha của x, c là nút cha của b, d là nút con của c mà không phải b. Nếu b màu đen và không NULL thì thoát. B5: Nếu x là gốc thì đặt x là màu đen và thoát. B6: Nếu nút d màu đỏ thì đặt b, d thành màu đen, nút c thành màu đỏ sau đó đặt x là nút c và quay lại B4. B7: Nếu d màu đen thì quay cây theo hướng ngược lại của nhánh x, cụ thể nút b sẽ thế chỗ cho c, a và c sẽ thành con của b, d sẽ vẫn là con của c, c sẽ lấy màu đỏ và b sẽ lấy màu đen sau đó thoát. Độ phức tạp O(h) ~ O(logn) với n là số nút của cây.

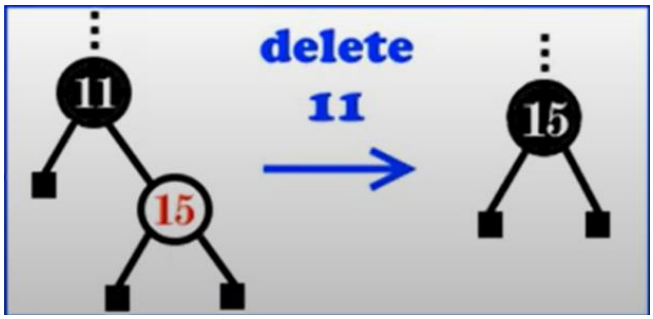
- delElement: B1: làm như cây tìm kiếm nhị phân bình thường, tìm đến vị trí cần xóa nút, gọi nút đó là x. B2: Nếu x có nhiều nhất1 nút con thì ta xóa x và dịch nút con đó lên, nếu nút cũ ở vị trí đó màu đen thì nút mới dịch lên cũng đánh màu đen sau đó đi tới B4. B3: Nếu x có 2 nút con thì ta tráo đổi x với nút con có khóa lớn nhất nằm bên nhánh cây trái sau đó xóa như B2 ở nút x vị trí mới. B4: Gọi x là a, cha của a là b, con của b không phải a là c, con của c là d. B5: Nếu c là nút đen và d là nút đỏ thì xoay theo hướng ngược lại với nhánh d, cụ thể c là cha của d và b. Sau đó đánh màu b, d là đen và c sẽ mang màu cũ của b và kết thúc. B6: Nếu c là nút đen và d cũng là nút đen thì đánh màu cho c là đỏ sau đó nếu b là đỏ thì ta đánh màu cho b là đen sau đó nữa nếu b chưa phải gốc ta đặt x là b sau đó quay lại B4. B7: Nếu c là nút đỏ thì ta cho c lên làm cha của b sau đó đánh màu cho c là đen, b là đỏ và quay lại B4. Độ phức tạp O(h) ~ O(logn).
- getSize: có một biến lưu số lượng phần tử của cây, cập nhật biến đó mỗi thao tác xóa/ thêm nút.



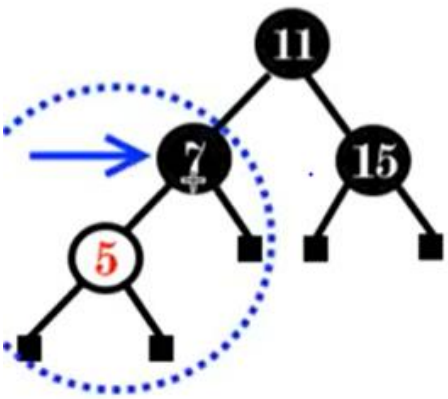
Hình 7: Ảnh khi thêm nút khóa 5 vào cây và uncle của nó là nút đen



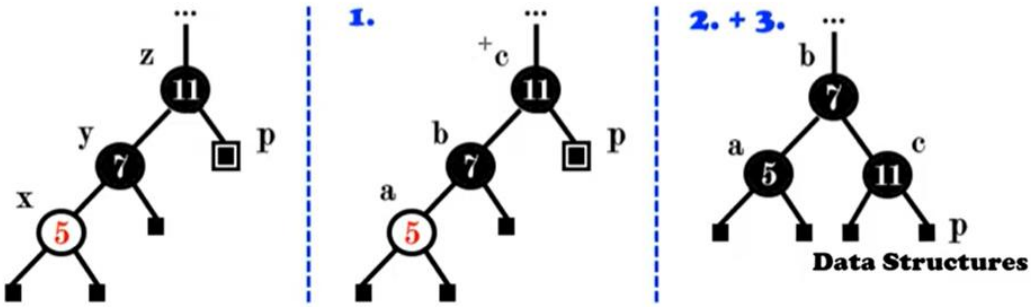
Hình 8: *Ảnh khi thêm nút có khóa 5 vào cây và uncle của nó là nút đỏ*



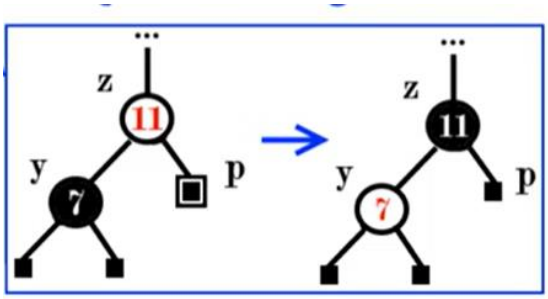
Hình 9: *Ảnh khi xóa nút khóa 11 của cây (nút có 1 con)*



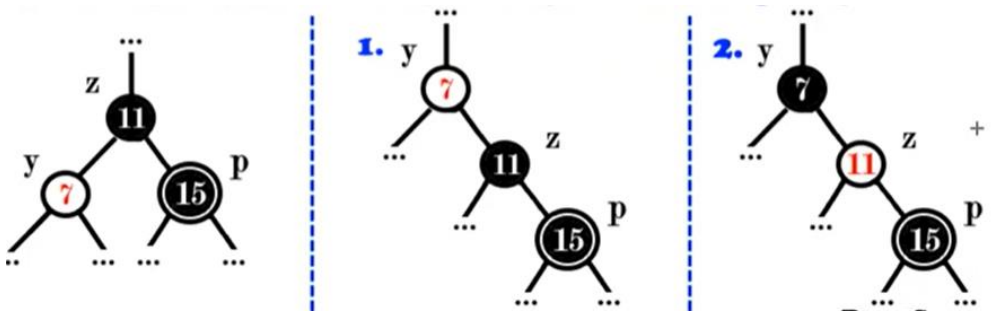
Hình 10: *Ảnh khi xóa nút khóa 11 của cây (nút có 2 con), trước tiên tìm đỉnh có khóa lớn nhất nằm trong nhánh cây trái, cụ thể ở đây là nút có khóa 7*



Hình 11: *Ảnh cân bằng đen khi nút sibling của nút làm mất cân bằng đang xét có màu đen và con của nút sibling đó có màu đỏ*



Hình 12: Ảnh cân bằng đen khi nút sibling của nú đang xétt làm mất cân bằng có màu đen và con của nút sibling đó có màu đen



Hình 13: Sibling của nút làm mất cân bằng đang xét có màu đỏ

4. CHƯƠNG TRÌNH VÀ KẾT QUẢ

4.1. Tổ chức chương trình

Chương trình đơn giản chỉ có 2 file main.c để lập trình cây tìm kiếm nhị phân bình thường và cây đỏ đen. Có file BinarySearchTree.h và RedBlackTree.h để người lập trình sau này có thể thêm vào dự án và include như một header của một thư viện bình thường.

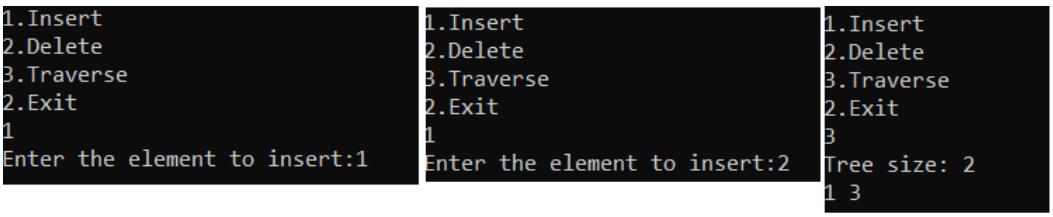
4.2. Ngôn ngữ cài đặt

Ngôn ngữ C (C18), GCC 8.1.0

4.3. Kết quả

4.3.1. Giao diện chính của chương trình

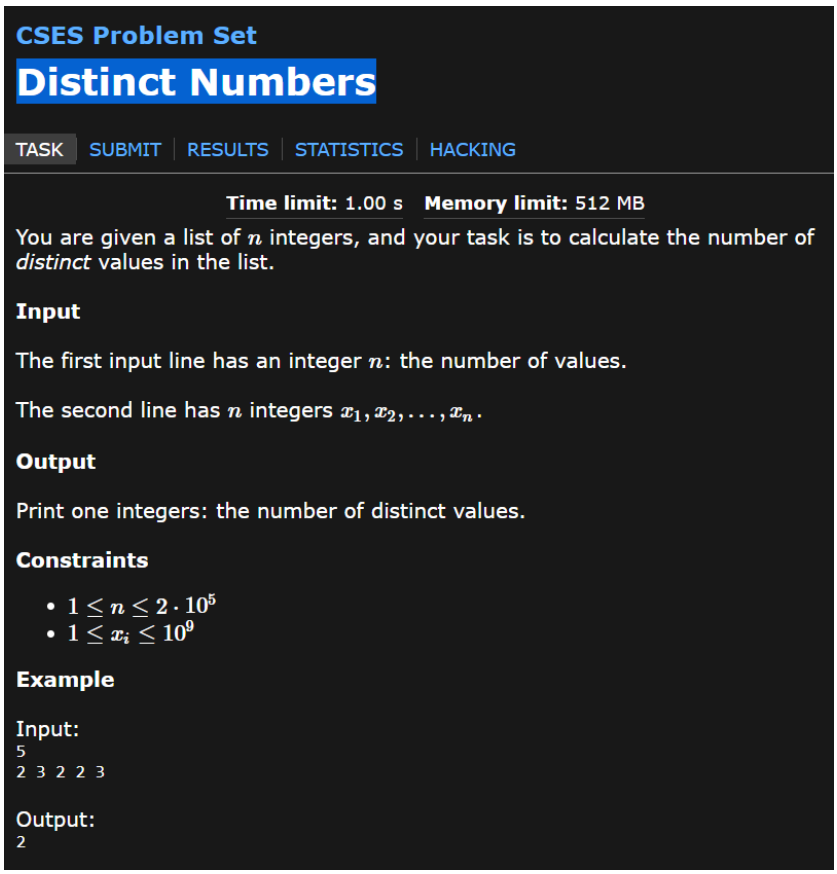
Chương trình của cây tìm kiếm nhị phân bình thường nhập xuất từ file nên không có giao diện. Chương trình cây đỏ đen thì có giao diện đơn giản như sau:



Hình 14: Giao diện chương trình

4.3.2. Kết quả thực thi của chương trình

Kết quả thực thi của chương trình đúng như dự tính. Để cho chính xác ta sẽ chọn một bài toán trên online judge để làm. Cụ thể bài đó là Distinct Numbers trên CSES, đây là một bài khá đơn giản và có thể làm bằng cách không sử dụng cây tìm kiếm nhị phân nhưng mục đích của ta là đánh giá độ chính xác và tốc độ của chương trình nên bài này khá ổn. Link đề bài: cses.fi/problemset/task/1621.


CSES Problem Set

Distinct Numbers

TASK | SUBMIT | RESULTS | STATISTICS | HACKING

Time limit: 1.00 s Memory limit: 512 MB

You are given a list of n integers, and your task is to calculate the number of *distinct* values in the list.

Input

The first input line has an integer n: the number of values.

The second line has n integers $x_1, x_2, \ldots, x_n$.

Output

Print one integers: the number of distinct values.

Constraints

- $1 \le n \le 2 \cdot 10^5$
- $1 \le x_i \le 10^9$

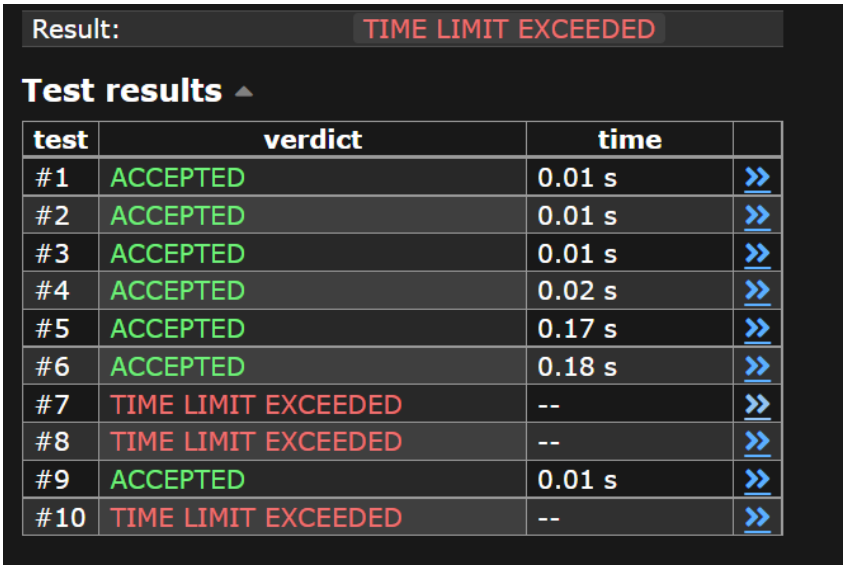
Example

Input:
5
2 3 2 2 3

Output:
2

Hình 15: Ảnh đề bài

Với cây nhị phân bình thường, ta bị chạy quá thời gian ở test 7, 8, 10 vì đề bài đã cố tình cài đặt cho dãy có đoạn tăng/giảm dần đủ dài để khiến độ cao của cây lớn. Link bài nộp: cses.fi/problemset/result/3805787/.



Result: TIME LIMIT EXCEEDED

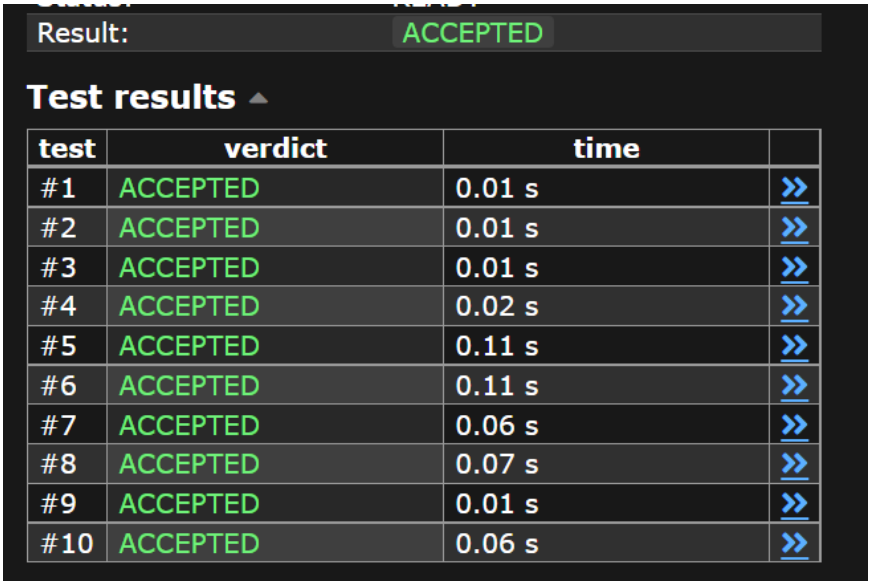
Test results

test	verdict	time	
#1	ACCEPTED	0.01 s	»
#2	ACCEPTED	0.01 s	»
#3	ACCEPTED	0.01 s	»
#4	ACCEPTED	0.02 s	»
#5	ACCEPTED	0.17 s	»
#6	ACCEPTED	0.18 s	»
#7	TIME LIMIT EXCEEDED	--	»
#8	TIME LIMIT EXCEEDED	--	»
#9	ACCEPTED	0.01 s	»
#10	TIME LIMIT EXCEEDED	--	»

Hình 16: Ảnh bài nộp với cây tìm kiếm nhị phân bình thường

Ở đây như đã nói trên vì bài đơn giản nên có rất nhiều cách làm, cách làm đơn giản mà vẫn sử dụng cây tìm kiếm nhị phân bình thường là ta dùng thuật toán random để trộn dãy số đó. Đã thử và ACCEPTED: cses.fi/problemset/result/4060835/.

Với cây đỏ đen thì ta cũng được ACCEPTED với thời gian chạy vài test thấp hơn cách trộn dãy số ở trên một chút: cses.fi/problemset/result/4073169/. Ảnh bài nộp:



Result: ACCEPTED

Test results

test	verdict	time	
#1	ACCEPTED	0.01 s	»
#2	ACCEPTED	0.01 s	»
#3	ACCEPTED	0.01 s	»
#4	ACCEPTED	0.02 s	»
#5	ACCEPTED	0.11 s	»
#6	ACCEPTED	0.11 s	»
#7	ACCEPTED	0.06 s	»
#8	ACCEPTED	0.07 s	»
#9	ACCEPTED	0.01 s	»
#10	ACCEPTED	0.06 s	»

Hình 17: Ảnh bài nộp với cây đỏ đen

4.3.3. *Nhận xét đánh giá*

Giải quyết được và tốt bài toán đề ra.

Xây dựng được thư viện cây tìm kiếm nhị phân.

Mô phỏng tạm được.

5. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

5.1. Kết luận

Đã hoàn thành mục tiêu tạo thư viện cây tìm kiến nhị phân với các tính năng tương tự cấu trúc set/map của c++. Chuyển được thành file header(.h) để người lập trình có thể sử dụng như thư viện.

Đã cải tiến lên cây đỏ đen để giải bài toán trên CSES và ACCEPTED.

Mô phỏng cây nhị phân chưa được trực quan lắm nhưng vẫn chấp nhận được.

5.2. Hướng phát triển

Thêm đầy đủ các tính năng của cấu trúc set/map.

Thêm tính năng của Ordered Set với cây đỏ đen(cấu trúc Ordered Set của c++ hiện nay không phải cây đỏ đen nên chạy khá chậm và cấu trúc set của c++ thì là cây đỏ đen nhưng thiếu tính năng của Ordered Set).

TÀI LIỆU THAM KHẢO

[1] Các thao tác vơi cây tìm kiếm nhị phân:

geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

programiz.com/dsa/binary-search-tree

[2] Tìm phần tử kế tiếp theo độ tăng dần khóa:

geeksforgeeks.org/inorder-successor-in-binary-search-tree/#:~:text=In%20Binary%20Tree%2C%20Inorder%20successor,key%20of%20the%20input%20node.

[3] Cây đỏ đen:

nguyenvanhieu.vn/tag/cay-do-den/

programiz.com/dsa/red-black-tree

youtube.com/watch?v=qvZGUFHWChY&list=PL9xmBV_5YoZNqDI8qfOZgzbqahCUmUEin

PHỤ LỤC

Code cây tìm kiếm nhị phân bình thường

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

struct node
{
int key, value;
struct node *left, *right, *parent;
};

struct node *newNode(int nkey, int nval, struct node *pre)
{
struct node *temp = (struct node *)malloc(sizeof(struct node));
temp->key = nkey;
temp->value = nval;
temp->left = temp->right = NULL;
temp->parent = pre;
return temp;
}

struct node *find(struct node *node, int key) {
if (node == NULL)
return node;
if (key < node->key)
```

```
return find(node->left, key);
else if (key > node->key)
return find(node->right, key);
else
return node;
}

struct node *insert(struct node *node, int key, int value, struct node *pre, int isRoot)
{

if (node == NULL) {
return newNode(key, value, ((isRoot == 0) ? pre : NULL));
}

if (key < node->key)
node->left = insert(node->left, key, value, node, 0);
else if (key > node->key)
node->right = insert(node->right, key, value, node, 0);
else
node->value = value;

return node;
}

struct node *minValueNode(struct node *node)
{
```

```
struct node *current = node;

while (current != NULL && current->left != NULL)
current = current->left;

return current;
}

struct node *maxValueNode(struct node *node)
{
struct node *current = node;

while (current != NULL && current->right != NULL)
current = current->right;

return current;
}

struct node *deleteNode(struct node *root, int key)
{
if (root == NULL) return root;

if (key < root->key)
root->left = deleteNode(root->left, key);
else if (key > root->key)
root->right = deleteNode(root->right, key);
```

```
else
{
if (root->left == NULL)
{
struct node *temp = root->right;
free(root);
return temp;
}
else if (root->right == NULL)
{
struct node *temp = root->left;
free(root);
return temp;
}

struct node *temp = minValueNode(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);
}
return root;
}

//Binary search tree operations
struct node *root;
int treeSize;
```

```
void newTree() {
treeSize = 0;
root = NULL;
}

bool isEmpty() {
return root == NULL;
}

struct node *searchNode(int key) {
struct node *node = find(root, key);
return node;
}

struct node *getBeginNode() {
return minValueNode(root);
};

struct node *nextNode(struct node *curNode) {
if (curNode->right != NULL)
return minValueNode(curNode->right);
struct node *par = curNode->parent;
while (par != NULL && curNode->key > par->key) {
curNode = par;
par = par->parent;
}
```

```
return par;
}

void inorder() {
struct node *it = getBeginNode();
while (it != NULL) {
printf("%d:%d ", it->key, it->value);
it = nextNode(it);
}
printf("\n");
}

void showTree(struct node *curNode) {
if (curNode == NULL)
return;
printf("%d:%d ", curNode->key, curNode->value);
showTree(curNode->left);
showTree(curNode->right);
}

void addNode(int key, int val) {
if (searchNode(key) == NULL)
treeSize++;
root = insert(root, key, val, NULL, 1);
}

void delNode(int key) {
```

```
if (searchNode(key) != NULL)
treeSize--;
root = deleteNode(root, key);
}

int getSize() {
return treeSize;
}

int main()
{

newTree();
FILE *fp = fopen("data.txt", "r");
int q, t, a, b;
struct node *tmp;
fscanf(fp, "%d", &q);
while (q--) {
fscanf(fp, "%d", &t);
if (t == 1) {
//add or replace
fscanf(fp, "%d %d", &a, &b);
addNode(a, b);
printf("%d\n", getSize());
//inorder();
showTree(root); //visualize
printf("\n");
```

```
} else if (t == 2) {
//del
fscanf(fp, "%d", &a);
delNode(a);
printf("%d\n", getSize());
//inorder();
showTree(root); //visualize
printf("\n");
} else {
//is in set?
fscanf(fp, "%d", &a);
tmp = searchNode(a);
printf((tmp != NULL ? "YES\n" : "NO\n"));
}
}
inorder();

}
```

Code cây đỏ đen

```
#include <stdio.h>
#include <stdlib.h>

enum nodeColor
{
RED,
BLACK
```

```
};

struct rbNode
{
int data;
enum nodeColor color;
struct rbNode *link[2];
};

struct rbNode *createNode(int data)
{
struct rbNode *newnode;
newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
newnode->data = data;
newnode->color = RED;
newnode->link[0] = newnode->link[1] = NULL;
return newnode;
}

struct rbNode *insertion(int data, struct rbNode *root, int *treeSize)
{
struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
int dir[98], ht = 0, index;
ptr = root;
if (!root)
{
(*treeSize)++;
```

```
root = createNode(data);
return root;
}

stack[ht] = root;
dir[ht++] = 0;
while (ptr != NULL)
{
if (ptr->data == data)
{
return root;
}
index = (data - ptr->data) > 0 ? 1 : 0;
stack[ht] = ptr;
ptr = ptr->link[index];
dir[ht++] = index;
}
stack[ht - 1]->link[index] = newnode = createNode(data);
while ((ht >= 3) && (stack[ht - 1]->color == RED))
{
if (dir[ht - 2] == 0)
{
yPtr = stack[ht - 2]->link[1];
if (yPtr != NULL && yPtr->color == RED)
{
stack[ht - 2]->color = RED;
stack[ht - 1]->color = yPtr->color = BLACK;
```

```
ht = ht - 2;
}
else
{
if (dir[ht - 1] == 0)
{
yPtr = stack[ht - 1];
}
else
{
xPtr = stack[ht - 1];
yPtr = xPtr->link[1];
xPtr->link[1] = yPtr->link[0];
yPtr->link[0] = xPtr;
stack[ht - 2]->link[0] = yPtr;
}
xPtr = stack[ht - 2];
xPtr->color = RED;
yPtr->color = BLACK;
xPtr->link[0] = yPtr->link[1];
yPtr->link[1] = xPtr;
if (xPtr == root)
{
root = yPtr;
}
else
{
```

```
stack[ht - 3]->link[dir[ht - 3]] = yPtr;
}
break;
}
}
else
{
yPtr = stack[ht - 2]->link[0];
if ((yPtr != NULL) && (yPtr->color == RED))
{
stack[ht - 2]->color = RED;
stack[ht - 1]->color = yPtr->color = BLACK;
ht = ht - 2;
}
else
{
if (dir[ht - 1] == 1)
{
yPtr = stack[ht - 1];
}
else
{
xPtr = stack[ht - 1];
yPtr = xPtr->link[0];
xPtr->link[0] = yPtr->link[1];
yPtr->link[1] = xPtr;
stack[ht - 2]->link[1] = yPtr;
```

```
}
xPtr = stack[ht - 2];
yPtr->color = BLACK;
xPtr->color = RED;
xPtr->link[1] = yPtr->link[0];
yPtr->link[0] = xPtr;
if (xPtr == root)
{
root = yPtr;
}
else
{
stack[ht - 3]->link[dir[ht - 3]] = yPtr;
}
break;
}
}
}
root->color = BLACK;
(*treeSize)++;
return root;
}

struct rbNode *deletion(int data, struct rbNode *root, int *treeSize)
{
struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
struct rbNode *pPtr, *qPtr, *rPtr;
```

```
int dir[98], ht = 0, diff, i;
enum nodeColor color;

if (!root)
{
return root;
}

ptr = root;
while (ptr != NULL)
{
if ((data - ptr->data) == 0)
break;
diff = (data - ptr->data) > 0 ? 1 : 0;
stack[ht] = ptr;
dir[ht++] = diff;
ptr = ptr->link[diff];
}
if (ptr == NULL)
return root;
(*treeSize)--;
if (ptr->link[1] == NULL)
{
if ((ptr == root) && (ptr->link[0] == NULL))
{
free(ptr);
root = NULL;
```

```
}
else if (ptr == root)
{
root = ptr->link[0];
free(ptr);
}
else
{
stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
}
}
else
{
xPtr = ptr->link[1];
if (xPtr->link[0] == NULL)
{
xPtr->link[0] = ptr->link[0];
color = xPtr->color;
xPtr->color = ptr->color;
ptr->color = color;

if (ptr == root)
{
root = xPtr;
}
else
{
```

```
stack[ht - 1]->link[dir[ht - 1]] = xPtr;
}

dir[ht] = 1;
stack[ht++] = xPtr;
}
else
{
i = ht++;
while (1)
{
dir[ht] = 0;
stack[ht++] = xPtr;
yPtr = xPtr->link[0];
if (!yPtr->link[0])
break;
xPtr = yPtr;
}

dir[i] = 1;
stack[i] = yPtr;
if (i > 0)
stack[i - 1]->link[dir[i - 1]] = yPtr;

yPtr->link[0] = ptr->link[0];

xPtr->link[0] = yPtr->link[1];
```

```
yPtr->link[1] = ptr->link[1];

if (ptr == root)
{
root = yPtr;
}

color = yPtr->color;
yPtr->color = ptr->color;
ptr->color = color;
}
}

if (ht < 1)
return root;

if (ptr->color == BLACK)
{
while (1)
{
pPtr = stack[ht - 1]->link[dir[ht - 1]];
if (pPtr && pPtr->color == RED)
{
pPtr->color = BLACK;
break;
}
```

```
if (ht < 2)
break;

if (dir[ht - 2] == 0)
{
rPtr = stack[ht - 1]->link[1];

if (!rPtr)
break;

if (rPtr->color == RED)
{
stack[ht - 1]->color = RED;
rPtr->color = BLACK;
stack[ht - 1]->link[1] = rPtr->link[0];
rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root)
{
root = rPtr;
}
else
{
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
}
dir[ht] = 0;
stack[ht] = stack[ht - 1];
```

```
stack[ht - 1] = rPtr;
ht++;

rPtr = stack[ht - 1]->link[1];
}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
(!rPtr->link[1] || rPtr->link[1]->color == BLACK))
{
rPtr->color = RED;
}
else
{
if (!rPtr->link[1] || rPtr->link[1]->color == BLACK)
{
qPtr = rPtr->link[0];
rPtr->color = RED;
qPtr->color = BLACK;
rPtr->link[0] = qPtr->link[1];
qPtr->link[1] = rPtr;
rPtr = stack[ht - 1]->link[1] = qPtr;
}
rPtr->color = stack[ht - 1]->color;
stack[ht - 1]->color = BLACK;
rPtr->link[1]->color = BLACK;
stack[ht - 1]->link[1] = rPtr->link[0];
rPtr->link[0] = stack[ht - 1];
```

```
if (stack[ht - 1] == root)
{
root = rPtr;
}
else
{
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
}
break;
}
}
else
{
rPtr = stack[ht - 1]->link[0];
if (!rPtr)
break;

if (rPtr->color == RED)
{
stack[ht - 1]->color = RED;
rPtr->color = BLACK;
stack[ht - 1]->link[0] = rPtr->link[1];
rPtr->link[1] = stack[ht - 1];

if (stack[ht - 1] == root)
{
root = rPtr;
```

```
}
else
{
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
}
dir[ht] = 1;
stack[ht] = stack[ht - 1];
stack[ht - 1] = rPtr;
ht++;

rPtr = stack[ht - 1]->link[0];
}
if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
(!rPtr->link[1] || rPtr->link[1]->color == BLACK))
{
rPtr->color = RED;
}
else
{
if (!rPtr->link[0] || rPtr->link[0]->color == BLACK)
{
qPtr = rPtr->link[1];
rPtr->color = RED;
qPtr->color = BLACK;
rPtr->link[1] = qPtr->link[0];
qPtr->link[0] = rPtr;
rPtr = stack[ht - 1]->link[0] = qPtr;
```

```
}
rPtr->color = stack[ht - 1]->color;
stack[ht - 1]->color = BLACK;
rPtr->link[0]->color = BLACK;
stack[ht - 1]->link[0] = rPtr->link[1];
rPtr->link[1] = stack[ht - 1];
if (stack[ht - 1] == root)
{
root = rPtr;
}
else
{
stack[ht - 2]->link[dir[ht - 2]] = rPtr;
}
break;
}
}
ht--;
}
}
return root;
}

void inorderTraversal(struct rbNode *node)
{
if (node)
{
```

```
inorderTraversal(node->link[0]);
printf("%d ", node->data);
inorderTraversal(node->link[1]);
}
return;
}

struct rbNode *root;
int treeSize;

void newTree() {
root = NULL;
treeSize = 0;
}

int getSize() {
return treeSize;
}

void delElement(int data) {
root = deletion(data, root, &treeSize);
}

void addElement(int data) {
root = insertion(data, root, &treeSize);
}
```

```
void inorder() {
inorderTraversal(root);
}

int main()
{
newTree();
int ch, data;
while (1)
{
printf("1.Insert\n2.Delete\n3.Traverse\n2.Exit\n");
scanf("%d", &ch);
switch (ch)
{
case 1:
printf("Enter the element to insert:");
scanf("%d", &data);
addElement(data);
break;
case 2:
printf("Enter the element to delete:");
scanf("%d", &data);
delElement(data);
break;
case 3:
printf("Tree size: %d\n", treeSize);
inorder();
```

```
printf("\n");
break;
case 4:
exit(0);
default:
printf("Not available\n");
break;
}
printf("\n");
}
return 0;
}
```