HỌC VIỆN KỸ THUẬT QUÂN SỰ

**KHOA CÔNG NGHỆ THÔNG TIN**

****

**BÀI TẬP LỚN**

**PHÂN TÍCH VÀ THIẾT KẾ GIẢI THUẬT**

**BÀI TOÁN TÌM KIẾM VÀ** **CÂY ĐỎ ĐEN (RED-BLACK TREE)**

Giáo viên hướng dẫn: **Nguyễn Mậu Uyên**

Sinh viên thực hiện: **Nguyễn Thị Ngọc Ánh**

Lớp: **KTPM 14**

Hà Nội, 12/2017

# MỤC LỤC

[**1.** **Bài toán** 4](#_Toc421541057)

[**2.** **Mô tả chi tiết thuật toán khi thêm nút trên cây** 4](#_Toc421541058)

[**3.** **Đánh giá độ phức tạp thuật toán** 4](#_Toc421541059)

[**4.** **Thực hiện các bước của thuật toán**  5](#_Toc421541060)

[**5.** **Cài đặt** 7](#_Toc421541061)

1. **Bài toán**
   * 1. **Giới thiệu thuật toán cây đỏ đen**

Một cây đỏ-đen là một [cây nhị phân](https://vi.wikipedia.org/wiki/C%C3%A2y_(l%C3%BD_thuy%E1%BA%BFt_%C4%91%E1%BB%93_th%E1%BB%8B)), là một cấu trúc dữ liệu trong [khoa học máy tính](https://vi.wikipedia.org/wiki/Khoa_h%E1%BB%8Dc_m%C3%A1y_t%C3%ADnh) để tổ chức các thành phần dữ liệu có thể so sánh, chẳng hạn các số. Trong cây nhị phân các thành phần dữ liệu được đưa vào các nút (node). Trong các nút có một nút nằm ở vị trí khởi đầu không là con của nút nào được gọi là **gốc**. Các nút khác đều là con của một nút nào đó và có không quá hai con. Từ nút gốc có một đường đi duy nhất đến mỗi nút khác trên cây.

Các cây đỏ đen thường đảm bảo một thời gian tốt nhất trong trường hợp xấu nhất cho các phép toán chèn (insertion), xóa (deletion),và tìm kiếm (search)..

* + - * 1. Quy tắc:

Một nút hoặc là đỏ hoặc đen.

Gốc là đen.

Tất cả các lá -là các nút *null* là đen

Cả hai con của mọi nút đỏ là đen. (và suy ra mọi nút đỏ có nút cha là đen.)

Tất cả các đường đi từ một nút đã cho tới các lá chứa một số như nhau các nút đen.

* + 1. **Nêu bài toán áp dụng**

Cho dãy số a gồm n phần tử. Hãy đưa dãy số này vào “ cây đỏ đen”, sau đó thực hiện yêu cầu thêm “node” có giá trị bằng k vào “cây đỏ đen” trên.

1. **Mô tả chi tiết thuật toán khi thêm nút trên cây**
   * 1. **Nguyên tắc khi thêm 1 nút**

Phép thêm 1 nút bắt đầu bằng việc bổ sung một nút [như trong cây tìm kiếm nhị phân bình thường](https://vi.wikipedia.org/wiki/C%C3%A2y_t%C3%ACm_ki%E1%BA%BFm_nh%E1%BB%8B_ph%C3%A2n#ph.C3.A9p_ch.C3.A8n) và gán cho nó màu đỏ. Ta xem xét để bảo toàn tính chất đỏ đen từ các nút lân cận với nút mới bổ sung. Thuật ngữ *nút chú bác* sẽ dùng để chỉ nút anh (hoặc em) với nút cha của nút đó như trong cây phả hệ.

**Chú ý rằng:**

* (Tất cả các lá -là các nút *null* là đen) giữ nguyên.
* (Cả hai con của nút đỏ là đen) nếu bị thay đổi chỉ bởi việc thêm một nút đỏ có thể sửa bằng cách gán màu đen cho một nút đỏ hoặc một phép quay.
* (Tất cả các đường đi từ gôc tới các lá có cùng một số nút đen) nếu bị thay đổi chỉ bởi việc thêm một nút đỏ có thể sửa bằng cách gán màu đen cho một nút đỏ hoặc một phép quay.

*Chú ý*: Nhãn **T** sẽ dùng để chỉ nút đang chèn vào, **P** chỉ nút cha của **T**, **G** chỉ ông của **T**, và **U** chỉ chú bác của **T**. Nhớ rằng,giữa các trường hợp, vai trò và nhãn của các nút có thể thay đổi còn trong cùng một trường hợp thì không.

* + 1. **Mô tả chi tiết thuật toán thêm nút trên cây**

***Nút gốc bắt buộc phải là nút đen:***

***if (root == t)***

***{***

***t->color = 'b';***

***return;***

***}***

**Trường hợp 1:** Nút mới thêm **T** ở tại gốc. Trong trường hợp này, nút **T** được gán màu đen (vì root -> color = ‘b’), Tính chất 5 được bảo đảm vì mọi đường đi chỉ có một nút.

root = t;

t->parent = NULL;

**Trường hợp 2:** Nút cha **P** của nút mới thêm là đen, khi đó Tính chất 4 (Cả hai nút con của nút đỏ là đen) không bị vi phạm vì nút mới thêm có hai con là "null' là đen. Tính chất 5 cũng không vi phạm vì nút mới thêm là đỏ không ảnh hưởng tới số nút đen trên tất cả đường đi. Vì vậy chỉ cần thêm nút **T** bình thường như khi thêm 1 nút ở cây nhị phân.

else

{

while (p != NULL)

{

q = p;

if (p->key<t->key)

p = p->right;

else

p = p->left;

}

t->parent = q;

if (q->key<t->key)

q->right = t;

else

q->left = t;

}

*Chú ‎ý:* Trong trường hợp tiếp theo nếu **T** có ông là nút **G**, vì nếu cha **P** là đỏ và **P** không ở gốc thì **G** là đen. Như vậy, **T** cũng có chú bác là **U**

**Trường hợp 3:** Cả cha **P** và bác **U** là đỏ, thì phải đổi cả hai thành đen còn **G** thành đỏ (để bảo toàn tính chất 5.Khi đó nút mới **T** có cha đen. Vì đường đi bất kỳ đi qua cha và bác của "**T**" phải đi qua ông của **T** nên số các nút đen trên đường đi này không thay đổi. Tuy thế nút ông **G** có thể vi phạm tính chất 2 (Gốc là đen) hoặc 4 (Cả hai con của nút đỏ là nút đen) (tính chất 4 bị vi phạm khi cha của **G** là đỏ). Để sửa chữa trường hợp này ta cần gán lại màu cho các nút ông, cha, bác.

while (t->parent != NULL&&t->parent->color == 'r')

{

node \*g = t->parent->parent;

if (g->left == t->parent)

{

if (g->right != NULL)

{

u = g->right;

if (u->color == 'r')

{

t->parent->color = 'b';

u->color = 'b';

g->color = 'r';

t = g;

}

**Trường hợp 4:** Nút cha **P** là đỏ nhưng nút chú bác **U** là đen, nút mới **T** là con phải của nút **P**, và **P** là con trái của nút **G**. Trong trường hợp này, thực hiện [quay trái](https://vi.wikipedia.org/w/index.php?title=Ph%C3%A9p_quay_(c%C3%A2y)&action=edit&redlink=1) chuyển đổi vai trò của nút mới **T** và nút cha **P** do đó định dạng lại nút **P** bằng Trường hợp 5 (đổi vai trò **T** và **P**) vì tính chất 4 bị vi phạm (Cả hai con của nút đỏ là đen). Phép quay cũng làm thay đổi một vài đường đi (các đường đi qua cây con nhãn "1") phải đi qua thêm nút mới **T**, nhưng vì **T** là đỏ nên không làm chúng vi pham tính chất 5.

**else{**

**if (t->parent->right == t)**

**{**

**t = t->parent;**

**leftrotate(t);**

**}**

**}**

***Trường hợp 5:*** Nút cha **P** là đỏ nhưng nút bác **U** là đen, nút mới **T** là con trái của nút **P**, và **P** là con trái của nút ông **G**. Trong trường hợp này, một phép quay phải trên nút ông **G** được thực hiện; kết quả của phép quay là trong cây mới nút **P** trở thành cha của cả hai nút **T** và nút **G**. Đã biết **G** là đen, vì bây giờ nó là con của **P**. Đổi màu của **P** và **G** thì cây thỏa mãn tính chất 4. Tính chất 5 không bị vi phạm vì các đường đi qua G trước đây bây giờ đi qua **P**.

t->parent->color = 'b';

g->color = 'r';

rightrotate(g);

***Trường hợp 6:*** Nút cha **P** đỏ, Nút chú **U** đen. **T** là con phải của **P**. **P** là con phải của ông **G**. Thì ta đổi màu **P** thành đen, **G** thành đỏ và thực hiện phép quay trái. Kết quả của phép quay trái là **P** trở thành cha của **G** và **T**.

***t->parent->color = 'b';***

***g->color = 'r';***

***leftrotate(g);***

***Trường hợp 7:*** Nút cha **P** đỏ, Nút chú **U** đen. **T** là con trái của **P**. **P** là con phải của ông **G**. Thì ta thực hiện quay phải **T** với **P**. Sau đó đổi màu cha sang đen, ông sang đỏ và thực hiện quay trái. Kết quả của phép quay là nút cha trở thành cha của nút ông và nút con.

if (t->parent->left == t)

{

t = t->parent;

rightrotate(t);

}

t->parent->color = 'b';

g->color = 'r';

leftrotate(g);

***Lefttrotate(node \*p)***

void RBtree::leftrotate(node \*p)

{

if (p->right == NULL)

return;

else

{

node \*y = p->right;

if (y->left != NULL)

{

p->right = y->left;

y->left->parent = p;

}

else

p->right = NULL;

if (p->parent != NULL)

y->parent = p->parent;

if (p->parent == NULL)

root = y;

else

{

if (p == p->parent->left)

p->parent->left = y;

else

p->parent->right = y;

}

y->left = p;

p->parent = y;

}

}

***Rightrotate(node \*p)***

void RBtree::rightrotate(node \*p)

{

if (p->left == NULL)

return;

else

{

node \*y = p->left;

if (y->right != NULL)

{

p->left = y->right;

y->right->parent = p;

}

else

p->left = NULL;

if (p->parent != NULL)

y->parent = p->parent;

if (p->parent == NULL)

root = y;

else

{

if (p == p->parent->left)

p->parent->left = y;

else

p->parent->right = y;

}

y->right = p;

p->parent = y;

}

}

1. **Độ phức tạp của thuật toán:**

Khi cây một nhánh, sẽ trở thành một danh sách liên kết, dữ liệu sẽ là một chiều thay vì hai chiều. Trong trường hợp này, thời gian truy xuất giảm về O(N), thay vì O(logN) đối với cây cân bằng.

Để bảo đảm thời gian truy xuất nhanh O(logN) của cây, chúng ta cần phải bảo đảm cây luôn luôn cân bằng. Điều này có nghĩa là mỗi node trên cây phải có xấp xỉ số node con bên phải bằng số node con bên trái.

Một cách tiếp cận giải quyết vấn đề cân bằng lại cây: đó là cây đỏ đen – là cây tìm kiếm nhị phân vì thế nó có các tính chất của cây tìm kiếm nhị phân ví dụ: node con trái nhỏ hơn node cha, node cha nhỏ hơn node con phải, bên cạnh đó cây đỏ đen được bổ sung một số đặc điểm.

Trong cây đỏ đen, việc cân bằng được thực thi trong khi thêm, xóa. Khi thêm một phần tử thì thủ tục chè sẽ kiểm tra xem tính cân bằng của cây có bị vi phạm hay không. Nếu có, sẽ xây dựng lại cấu trúc cây. Bằng cách này, cây luôn luôn được giữ cân bằng.

1. **Thực hiện thuật toán**
   1. *Bộ dữ liệu 1: Cho bộ dữ liệu: 1,2,14,1,7,15,5,8 được đưa vào “cây đỏ đen”. Sau đó thêm nút có giá trị “=4” vào “cây đỏ đen” trên.*

* *Bước 1: Đưa bộ dữ liệu vào cây đỏ đen. Ta được:*

2

15

8

5

* *Bước 2: Thêm nút có giá trị “=4” vào “cây đỏ đen”, nút mới thêm phải là nút màu đỏ*

2

15

8

5

4

* *Bước 3: “ Cây đỏ đen vi phạm tính chất 4(cả 2 con của mọi nút đỏ phải là đen), theo trường hợp 3 ta có:*

2

15

7

4

* *Bước 4: “ Cây đỏ đen” vẫn vi phạm quy tắc 4, theo trường hợp 4 ta có:*

7

2

15

4

* *Bước 5: “ Cây đỏ đen” vẫn vi phạm quy tắc 4, theo trường hợp 5 ta có:*

11

15

4

2

* ***Hoàn thành việc thêm nút có giá trị “=4” vào “cây đỏ đen”.***
  1. *Bộ dữ liệu 2: Cho bộ dữ liệu: 11,2,14,1,13,5,8,12 được đưa vào “cây đỏ đen”. Sau đó thêm nút có giá trị “=9” vào “cây đỏ đen” trên.*
* *Bước 1: Đưa bộ dữ liệu vào cây đỏ đen. Ta được:*

14

12

8

2

* *Bước 2: Thêm nút có giá trị “=9” vào “cây đỏ đen”, nút mới thêm phải là nút màu đỏ. Ta được:*

2

14

12

8

9

*Bước 3: “ Cây đỏ đen vi phạm tính chất 4(cả 2 con của mọi nút đỏ phải là đen), theo trường hợp 6 ta có:*

5

14

12

9

2

9

5

14

12

2

* ***Hoàn thành việc thêm nút có giá trị “=9” vào “cây đỏ đen”.***

1. **Cài đặt**

#include<iostream>

using namespace std;

struct node

{

int key;

node \*parent;

char color;

node \*left;

node \*right;

};

class RBtree

{

node \*root;

public:

RBtree()

{

root = NULL;

}

void insert();

void insertfix(node \*);

void leftrotate(node \*);

void rightrotate(node \*);

void del();

node\* successor(node \*);

void delfix(node \*);

void disp();

void display(node \*);

void search();

};

void RBtree::insert()

{

int z;

cout << "\nNhap gia tri cua node : ";

cin >> z;

node \*p, \*q;

node \*t = new node;

t->key = z;

t->left = NULL;

t->right = NULL;

t->color = 'r';

p = root;

q = NULL;

if (root == NULL)

{

root = t;

t->parent = NULL;

}

else

{

while (p != NULL)

{

q = p;

if (p->key<t->key)

p = p->right;

else

p = p->left;

}

t->parent = q;

if (q->key<t->key)

q->right = t;

else

q->left = t;

}

insertfix(t);

}

void RBtree::insertfix(node \*t)

{

node \*u;

if (root == t)

{

t->color = 'b';

return;

}

while (t->parent != NULL&&t->parent->color == 'r')

{

node \*g = t->parent->parent;

if (g->left == t->parent)

{

if (g->right != NULL)

{

u = g->right;

if (u->color == 'r')

{

t->parent->color = 'b';

u->color = 'b';

g->color = 'r';

t = g;

}

else{

if (t->parent->right == t)

{

t = t->parent;

leftrotate(t);

}

t->parent->color = 'b';

g->color = 'r';

rightrotate(g);

}

}

else

{

if (t->parent->right == t)

{

t = t->parent;

leftrotate(t);

}

t->parent->color = 'b';

g->color = 'r';

rightrotate(g);

}

}

else

{

if (g->left != NULL)

{

u = g->left;

if (u->color == 'r')

{

t->parent->color = 'b';

u->color = 'b';

g->color = 'r';

t = g;

}

else{

g->color = 'r';

t->parent->color = 'b';

leftrotate(g);

}

}

else

{

if (t->parent->left == t)

{

t = t->parent;

rightrotate(t);

}

t->parent->color = 'b';

g->color = 'r';

leftrotate(g);

}

}

root->color = 'b';

}

}

void RBtree::del()

{

if (root == NULL)

{

cout << "\nCay rong.";

return;

}

int x;

cout << "\nNhap gia tri cua node can xoa : ";

cin >> x;

node \*p;

p = root;

node \*y = NULL;

node \*q = NULL;

int found = 0;

while (p != NULL&&found == 0)

{

if (p->key == x)

found = 1;

if (found == 0)

{

if (p->key<x)

p = p->right;

else

p = p->left;

}

}

if (found == 0)

{

cout << "\nKhong tim thay node .";

return;

}

else

{

cout << "\nXoa node : " << p->key;

cout << "\nMau : ";

if (p->color == 'b')

cout << "Black\n";

else

cout << "Red\n";

if (p->parent != NULL)

cout << "\nNode cha : " << p->parent->key;

else

cout << "\nKhong co node cha . ";

if (p->right != NULL)

cout << "\nCon phai: " << p->right->key;

else

cout << "\nKhong co con phai . ";

if (p->left != NULL)

cout << "\nCon trai : " << p->left->key;

else

cout << "\nKhong co con trai . ";

cout << "\n \nDa xoa .";

if (p->left == NULL || p->right == NULL)

y = p;

else

y = successor(p);

if (y->left != NULL)

q = y->left;

else

{

if (y->right != NULL)

q = y->right;

else

q = NULL;

}

if (q != NULL)

q->parent = y->parent;

if (y->parent == NULL)

root = q;

else

{

if (y == y->parent->left)

y->parent->left = q;

else

y->parent->right = q;

}

if (y != p)

{

p->color = y->color;

p->key = y->key;

}

if (y->color == 'b')

delfix(q);

}

}

void RBtree::delfix(node \*p)

{

node \*s;

while (p != root&&p->color == 'b')

{

if (p->parent->left == p)

{

s = p->parent->right;

if (s->color == 'r')

{

s->color = 'b';

p->parent->color = 'r';

leftrotate(p->parent);

s = p->parent->right;

}

if (s->right->color == 'b'&&s->left->color == 'b')

{

s->color = 'r';

p = p->parent;

}

else

{

if (s->right->color == 'b')

{

s->left->color == 'b';

s->color = 'r';

rightrotate(s);

s = p->parent->right;

}

s->color = p->parent->color;

p->parent->color = 'b';

s->right->color = 'b';

leftrotate(p->parent);

p = root;

}

}

else

{

s = p->parent->left;

if (s->color == 'r')

{

s->color = 'b';

p->parent->color = 'r';

rightrotate(p->parent);

s = p->parent->left;

}

if (s->left->color == 'b'&&s->right->color == 'b')

{

s->color = 'r';

p = p->parent;

}

else

{

if (s->left->color == 'b')

{

s->right->color = 'b';

s->color = 'r';

leftrotate(s);

s = p->parent->left;

}

s->color = p->parent->color;

p->parent->color = 'b';

s->left->color = 'b';

rightrotate(p->parent);

p = root;

}

}

p->color = 'b';

root->color = 'b';

}

}

void RBtree::leftrotate(node \*p)

{

if (p->right == NULL)

return;

else

{

node \*y = p->right;

if (y->left != NULL)

{

p->right = y->left;

y->left->parent = p;

}

else

p->right = NULL;

if (p->parent != NULL)

y->parent = p->parent;

if (p->parent == NULL)

root = y;

else

{

if (p == p->parent->left)

p->parent->left = y;

else

p->parent->right = y;

}

y->left = p;

p->parent = y;

}

}

void RBtree::rightrotate(node \*p)

{

if (p->left == NULL)

return;

else

{

node \*y = p->left;

if (y->right != NULL)

{

p->left = y->right;

y->right->parent = p;

}

else

p->left = NULL;

if (p->parent != NULL)

y->parent = p->parent;

if (p->parent == NULL)

root = y;

else

{

if (p == p->parent->left)

p->parent->left = y;

else

p->parent->right = y;

}

y->right = p;

p->parent = y;

}

}

node\* RBtree::successor(node \*p)

{

node \*y = NULL;

if (p->left != NULL)

{

y = p->left;

while (y->right != NULL)

y = y->right;

}

else

{

y = p->right;

while (y->left != NULL)

y = y->left;

}

return y;

}

void RBtree::disp()

{

display(root);

}

void RBtree::display(node \*p)

{

if (root == NULL)

{

cout << "\nCay rong .";

return;

}

if (p != NULL)

{

cout << "\n\t NODE: ";

cout << "\n Key: " << p->key;

cout << "\n Mau : ";

if (p->color == 'b')

cout << "Black";

else

cout << "Red";

if (p->parent != NULL)

cout << "\n Cha: " << p->parent->key;

else

cout << "\n Node khong co node cha. ";

if (p->right != NULL)

cout << "\n Con phai: " << p->right->key;

else

cout << "\n Node khong co node con phai. ";

if (p->left != NULL)

cout << "\n Con trai : " << p->left->key;

else

cout << "\n Node khong co con trai . ";

cout << endl;

if (p->left)

{

cout << "\n\nTrai:\n";

display(p->left);

}

if (p->right)

{

cout << "\n\nPhai:\n";

display(p->right);

}

}

}

void RBtree::search()

{

if (root == NULL)

{

cout << "\nCay rong \n";

return;

}

int x;

cout << "\n Nhap vao key cua node can tim : ";

cin >> x;

node \*p = root;

int found = 0;

while (p != NULL&& found == 0)

{

if (p->key == x)

found = 1;

if (found == 0)

{

if (p->key<x)

p = p->right;

else

p = p->left;

}

}

if (found == 0)

cout << "\nKhong tim thay .";

else

{

cout << "\n\t Thay node: ";

cout << "\n Key: " << p->key;

cout << "\n Mau: ";

if (p->color == 'b')

cout << "Black";

else

cout << "Red";

if (p->parent != NULL)

cout << "\n Cha: " << p->parent->key;

else

cout << "\n Khong co node cha ";

if (p->right != NULL)

cout << "\n Con phai: " << p->right->key;

else

cout << "\n Khong co con phai. ";

if (p->left != NULL)

cout << "\n Con trai: " << p->left->key;

else

cout << "\n Khong co con trai . ";

cout << endl;

}

}

int main()

{

int ch, y = 0;

RBtree obj;

do

{

cout << "\n\t Cay do den ";

cout << "\n 1. Them ";

cout << "\n 2. Xoa";

cout << "\n 3. Tim";

cout << "\n 4. Hien thi cay ";

cout << "\n 5. Thoat ";

cout << "\nNhap lua chon : ";

cin >> ch;

switch (ch)

{

case 1: obj.insert();

cout << "\n Da them .\n";

break;

case 2: obj.del();

break;

case 3: obj.search();

break;

case 4: obj.disp();

break;

case 5: y = 1;

break;

default: cout << "\nNhap lua chon .";

}

cout << endl;

} while (y != 1);

return 1;

}