1. Nêu bài toán

Cây tìm kiếm nhị phân (Binary Search Tree - BST) là một cấu trúc dữ liệu cơ bản nhưng cực kỳ quan trọng.

* Các thao tác với BST (vd: search, max, min, insert, delete,..) có độ phức tạp thời gian là O(h) với h là độ cao của cây, trong trường hợp tổ chức tree tệ nhất (h = n), thao tác trên tree lúc này không khác gì trên linked-list.
* Cây tìm kiếm nhị phân tự cân bằng (Self-balancing Binary Search Tree) ra đời để giải quyết vấn đề này. Red-Black tree là một cây tìm kiếm nhị phân tự cân bằng khác rất phổ biến hiện nay, mỗi node trong cây này có thêm 1 bit để lưu trữ màu sắc (red hoặc black). Các bit màu sắc này dùng để đảm bảo cây duy trì độ cân bằng tương đổi trong quá trình insert hoặc delete.
* Tính chất

Mỗi nút của cây đỏ-đen có thuộc tinh "màu" nhận một trong hai giá trị "đỏ" hoặc "đen". Ngoài ra:

* + Một nút hoặc là đỏ hoặc đen.
  + Gốc 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.

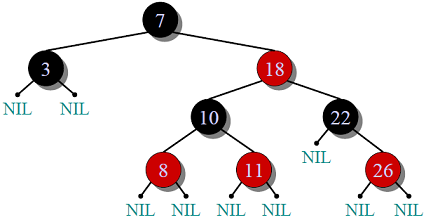
1. Độ phức tạp

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 (ít ra cũng là cây gần 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 còn được bổ sung một số đắc điểm.

Trong cây đỏ đen, việc cân bằng được thực thi trong khi chèn, xóa. Khi thêm một phần tử thì thủ tục chèn sẽ kiểm tra xem tính chất 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. Mô tả chi tiết khi thêm và xóa nút trên cây

**Thêm nút**

Phép chèn bắt đầu bằng việc bổ sung một nút [như](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)trong cây nhị phân tìm kiếm bình thườngvà 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:

* Tính chất 3 (Tất cả các lá -là các nút *null* là đen) giữ nguyên.
* Tính chất 4 (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.
* Tính chất 5 (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 **N** sẽ dùng để chỉ nút đang chèn vào, **P** chỉ nút cha của **N**, **G** chỉ ông của **N**, và **U** chỉ chú bác của **N**. 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.

Mỗi trường hợp được giới thiệu bằng một đoạn mã **C**. Nút chú bác và nút ông dễ dàng xác định nhờ các hàm sau:

struct node \*grandparent(struct node \*n) {

return n->parent->parent;

}

struct node \*uncle(struct node \*n) {

if (n->parent == grandparent(n)->left)

return grandparent(n)->right;

else

return grandparent(n)->left;

}

**Trường hợp 1:** Nút mới thêm **N** ở tại gốc. Trong trường hợp này, gán lại màu đen cho **N**, để bảo toàn tính chất 2 (Gốc là đen). Vì mới chỉ bổ sung một nút, Tính chất 5 được bảo đảm vì mọi đường đi chỉ có một nút.

void insert\_case1(struct node \*n) {

if (n->parent == NULL)

n->color = BLACK;

else

insert\_case2(n);

}

**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 ẩnh hưởng tới số nút đen trên tất cả đường đi.

void insert\_case2(struct node \*n) {

if (n->parent->color == BLACK)

return; /\* Tree is still valid \*/

else

insert\_case3(n);

}

*Chú ‎ý:* Trong trường hợp tiếp theo nếu **N** có ông là nút **G**, vì nếu cha **P** là đỏ và **P** không ở gốc thì **G** là đen. Như vậy, **N** cũng có chú bác là **U**, although it may be a leaf in cases 4 and 5.

|  |
| --- |
| [Diagram of case 3](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_insert_case_3.png)  **Trường hợp 3:** Cả cha **P** và bác **U** là đỏ, thì thể đổi cả hai thành đen còn **G** thành đỏ (để bảo toàn tính chất 5.Khi đó nút mới **N** có cha đen. Vì đường đi bất kỳ đi qua cha và bác của "N" phải đi qua ông của *N* 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 gọi một thủ tục đệ quy trên **G** từ trường hợp 1. Note that this is the only recursive call, and it occurs prior to any rotations, which proves that a constant number of rotations occur. |

void insert\_case3(struct node \*n) {

if (uncle(n)!= NULL && uncle(n)->color == RED) {

n->parent->color = BLACK;

uncle(n)->color = BLACK;

grandparent(n)->color = RED;

insert\_case1(grandparent(n));

}

else

insert\_case4(n);

}

*Chú ‎ý:* Trong các trường hợp tiếp theo, giả sử rằng nút cha **P** là con trái của cha của nó. Nếu nó là con phải, *left* và *right* đổi chỗ cho nhau trong cases 4 and 5.

|  |
| --- |
| [Diagram of case 4](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_insert_case_4.png)  **Trường hợp 4:** Nút cha **P** là đỏ nhưng nút chú bác **U** là đen, nút mới **N** 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 **N** và nút cha **P** do đó định dạng lại nút **P** bằng Trường hợp 5 (đổi vai trò **N** 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 **N**, nhưng vì **N** là đỏ nên không làm chúng vi pham tính chất 5 |

void insert\_case4(struct node \*n) {

if (n = = n->parent->right && n->parent = = grandparent(n)->left) {

rotate\_left(n->parent);

n = n->left;

} else if (n = = n->parent->left && n->parent = = grandparent(n)->right) {

rotate\_right(n->parent);

n = n->right;

}

insert\_case5(n);

}

|  |
| --- |
| [Diagram of case 5](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_insert_case_5.png)  ***Trường hợp 5:'*** Nút cha **P** là đỏ nhưng nút bác **U** là đen, nút mới **N** 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 phep quay là trong cây mới nút **P** trở thành cha của cả hai nít **N** 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**. |

void insert\_case5(struct node \*n) {

n->parent->color = BLACK;

grandparent(n)->color = RED;

if (n == n->parent->left && n->parent == grandparent(n)->left) {

rotate\_right(grandparent(n));

} else {

/\* Here, n == n->parent->right && n->parent == grandparent(n)->right \*/

rotate\_left(grandparent(n));

}

}

**Xóa nút**

Trong cây tìm kiếm nhị phân bình thường khi xóa một nút có cả hai con (không là *lá null*), ta tìm phần tử lớn nhất trong cây con trái hoặc phần tử nhỏ nhất trong cây con phải, chuyển giá trị của nó vào nút đang muốn xóa. Khi đó chúng ta xóa đi nút đã được copy giá trị, nút này có ít hơn hai con (không là *lá null*). Vì việc copy giá trị không làm mất tính chất đỏ đen nên không cần phải sửa chữa gì cho thao tác này. Việc này chỉ đặt ra khi xóa các nút có nhiều nhất một con (không là *lá null*).

Chúng ta sẽ thảo luận về việc xóa một nút có nhiều nhất một con (không là lá null).

Nếu ta xóa một nút đỏ, ta có thể chắc chắn rằng con của nó là nút đen. Tất cả các đường đi đi qua nút bị xóa chỉ đơn giản bớt đi một nút đỏ do đó tính chất 5 không thay đổi. Ngoài ra, cả nút cha và nút con của nút bị xóa đều là nút đen, do đó tính chất 3 và 4 vẫn giữa nguyên.. Một trường hợp đơn giản khác là khi xóa một nút đen chỉ có một con là nút đỏ. Khi xóa nút đó các tính chất 4 và 5 bị phá vỡ, nhưng nếu gán lại màu cho nút con là đen thì chúng lại được khôi phục.

Trường hợp phức tạp xảy ra khi cả nút bị xóa và nút con của nó đều là đen. Chúng ta sẽ bắt đầu bằng việc thay nút bị xóa bằng nút con của nó. Chúng ta sẽ gọi nút con này (trong vị trí mới của nó là **N**, và anh em với nó (con khác của nút cha mới) là **S**. Tiếp theo ta vẫn dùng **P** chỉ cha mới của **N**, **SL** chỉ con trái của **S**, và **SR** chỉ con phải của **S**(chúng tồn tại vì **S** không thể là lá).

*Chú ‎ý*: Giữa các trường hợp khác nhau, vai trò và nhãn của các nút có thể thay đổi, nhưng trong một trường hợp mọi nhãn giữ vai trò không thay đổi. Trong hình vẽ các màu đỏ đen được thể hiện khi màu của nút đã rõ ràng, màu trắng biểu thị một màu chưa rõ (hoặc đỏ hoặc đen).

Chúng ta sẽ sử dụng hàm sau tìm người anh em của *N'*:

struct node \*sibling(struct node \*n) {

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

return n->parent->right;

else

return n->parent->left;

}

*Chú ‎ý*: Tất nhiên, chúng ta cần hoàn chỉnh các lá null sau mọi phép thay đổi. Nếu nút bị xóa không có con **N** khác "lá null", dễ dàng thấy rằng các tính chất được thỏa mãn. Còn nếu **N** là một "lá nulll", có thể sửa chữa lược đồ (hoặc code) để trong tất cảc các trường hợp các tính chất được thỏa mãn.

Trước mỗi bước ta có thể dùng hàm (function) replace\_node thay thế nút con child vào vị trí của nút bị xóa trên cây. Để thuận tiện các đoạn code trong mục này quy ước rằng các "lá null" được biểu diễn bằng các đối tượng nút thực sự khác biệt một chút với NULL (code trong *phép chèn* có biểu diễn không nư vậy).

void delete\_one\_child(struct node \*n) {

/\* Giả thiết: n có ít nhất một nút con *null* \*/

struct node \*child = is\_leaf(n->right) ? n->left: n->right;

replace\_node(n, child);

if (n->color == BLACK) {

if (child->color == RED)

child->color = BLACK;

else

delete\_case1(child);

}

free(n);

}

*Ghi chú*: Nếu **N** là "lá nul" và ta khong muốn biểu diễn các "lá null" bằng các đối tượng nút thực, ta có thể sửa giải thuật bằng cách trước hết gọi delete\_case1() trên cha của nó (nghĩa là nút bị xóa n trong đoạn code trên) rồi sau đó mới xóa nó. Ta có thể làm như vậy vì cha của nó là đen, do đó có diễn biến như với "lá null" (một số người gọi là "lá ảo", "lá ma"). Ta cũng có thể xóa nó khỏi cuối của n và sẽ khôi phục lại sau tất cả các phép toán.

Nếu cả **N** và gốc ban đầu của nó là đen thì sau khi xóa các đường qua "N" giảm bớt một nút đen. Do đó vi phạm Tính chất 5, cây cần phải cân bằng lại. Có các trường hợp sau:

**Trường hợp 1:** **N** là gốc mới. Trong trường hợp này chúng ta dừng lại. Ta đã giải phóng một nút đen khỏi mọi đường đi và gôc mới lại là đen. Không tính chất nào bị vi phạm.

void delete\_case1(struct node \*n) {

if (n->parent == NULL)

return;

else

delete\_case2(n);

}

*Chú ‎ý*: Trong các trường hợp 2, 5, và 6, ta quy ước **N** là con trái của cha **P**. Nếu no là con phải, *left* và *right* sẽ tráo đổi cho nhau. Tuy nhiên code ví dụ làm cho cả hai trường hợp.

**Trường hợp 2**

|  |
| --- |
| [Diagram of case 2](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_delete_case_2.png)  **Trường hợp 2:** **S** là đỏ. Trong trường hợp này tráo đổi màu của **P** và **S**, và sau đó [quay trái](https://vi.wikipedia.org/w/index.php?title=Ph%C3%A9p_quay&action=edit&redlink=1) tai **P**, nó sẽ làm cho **S** trở thành nút ông của **N**. Chú ý rằng **P** có màu đen và có một con màu đỏ. Tất cả các đường đi có số các nút đen giống nhau, bây giờ **N**có một anh em màu đen và cha màu đỏ, chúng ta có thể tiếp tục với các trường hợp 4, 5, hoặc 6. (anh em mới của nó là đen ví chỉ có một con của nút đỏ **S**.) Trong các trường hợp sau la sẽ gọi anh em mới của **N'** là **S**. |

void delete\_case2(struct node \*n) {

if (sibling(n)->color == RED) {

n->parent->color = RED;

sibling(n)->color = BLACK;

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

rotate\_left(n->parent);

else

rotate\_right(n->parent);

}

delete\_case3(n);

}

**Trường hợp 3**

|  |
| --- |
| [Diagram of case 3](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_delete_case_3.png)  **Trường hợp 3:** **P**, **S**, và các con của **S** là đen. Trong trường hợp này, chúng ta gán lại cho **S** màu đỏ. Kết quả là mọi đường đi qua **S**, (tất nhiên chúng không qua **N**,có ít hơn một nút đen. Vì việc xóa đi cha trước đây của **N'** làm tất cả các đương đi qua **N** bớt đi một nút đen, nên chúng bằng nhau. Tuy nhiên tất cả các đường đi qua **P** bây giờ có ít hơn một nút đen so với các đường không qua **P**, do đó Tính chất 5 (Tất cả các đường đi từ gốc tới các nút lá có cùng số nút đen) sẽ bị vi phạm. Để sửa chữa nó chúng ta lại tái cân bằng tại **P**, bắt đầu từ trường hợp 1. |

void delete\_case3(struct node \*n) {

if (n->parent->color == BLACK &&

sibling(n)->color == BLACK &&

sibling(n)->left->color == BLACK &&

sibling(n)->right->color == BLACK)

{

sibling(n)->color = RED;

delete\_case1(n->parent);

}

else

delete\_case4(n);

}

**Trường hợp 4**

**Trường hợp 4:** **S** và các con của **S** là đen nhưng **P** là đỏ. Trong trường hợp này, chúng ta đổi ngược màu của **S** và **P**. Điều này không ảnh hưởng tới số nút đen trên các đường đi không qua **N**, nhưng thêm một nút đen trên các đường đi qua **N**, thay cho nút đen đã bị xóa trên các đường này.

|  |
| --- |
|  |

void delete\_case4(struct node \*n) {

if (n->parent->color == RED &&

sibling(n)->color == BLACK &&

sibling(n)->left->color == BLACK &&

sibling(n)->right->color == BLACK)

{

sibling(n)->color = RED;

n->parent->color = BLACK;

}

else

delete\_case5(n);

}

**Trường hợp 5**

[](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_delete_case_5.png)

**Trường hợp 5:** **S** là đen, con trái của **S** là đỏ, con phải của **S** là đen, còn **N** là con trái của cha nó. Trong trường hợp này chúng ta quay phải tại **S**, khi đó con trái của **S** trở thành cha của **S** và **N** là anh em mới của nó. Sau đó ta tráo đổi màu của **S** và cha mới của nó. Tất cả các đường đi sẽ có số nút đen như nhau, nhưng bây giờ **N** có một người anh em đen mà con phải của nó lại là đỏ, chúng ta chuyển sang Trường hợp 6. Hoặc **N** hoặc cha của nó bị tác động bởi việc dịch chuyên này.

(Lưu ý trong trường hợp 6, ta đặt lại nút anh em mới của **N** là **S**.)

|  |
| --- |
|  |

void delete\_case5(struct node \*n) {

if (n == n->parent->left &&

sibling(n)->color == BLACK &&

sibling(n)->left->color == RED &&

sibling(n)->right->color == BLACK)

{

sibling(n)->color = RED;

sibling(n)->left->color = BLACK;

rotate\_right(sibling(n));

}

else if (n == n->parent->right &&

sibling(n)->color == BLACK &&

sibling(n)->right->color == RED &&

sibling(n)->left->color == BLACK)

{

sibling(n)->color = RED;

sibling(n)->right->color = BLACK;

rotate\_left(sibling(n));

}

delete\_case6(n);

}

**Trường hợp 6**

|  |
| --- |
| [Diagram of case 6](https://vi.wikipedia.org/wiki/T%E1%BA%ADp_tin:Red-black_tree_delete_case_6.png)  **Trường hợp 6:** **S** là đen, con phải của **S** là đỏ và **N** là con trái của nút cha **P**. Trong trường hợp này chúng ta quay trái tại **P**, khi đó **S** trở thành cha của **P** và con phải của **S**. Chúng ta hoán đổi màu của **P** và **S**, và gán cho con phải của **S** màu đen. Cây con giữ nguyên màu của gốc do đó Tính chất 4 (Cả hai con của nút đỏ là đen) và Tính chất 5 không bị vi phạm trong cây con này. Tuy nhiên, **N** bây giờ có thêm một nút đen tiền nhiệm: hoặc **P** mới bị tô đen, nó đã là đen và **S** là nút ông của nó trở thành đen. Như cậy các đương đi qua **N** có thêm một nút đen.  Trong lúc đó, với một đường đi không đi qua **N**, có hai khả năng:   * đi qua nút anh em của **N**. Khi đó cả trước và sau khi quay nó phải đi qua **S** và **P**, khi thay đổi màu sắc hai nút này đã tráo đổi màu cho nhau. Như vây đường đi này không bị thay đổi số nút đen. * đi qua nút bác của **N'**, là con phải của **S**. Khi đó trước khi quay nó đi qua **S**, cha của **S**, và con phải của **S**, nhưng sau khi quay nó chỉ đi qua nút **S** và con phải của **S**, khi này **S** đã nhận màu cũ của cha **P** còn con phải của **S'**s đã đổi màu từ đỏ thành đen. Kết quả là số các nút đen trên đường đi này không thay đổi.   Như vậy, số các nút đen trên các đường đi là không thay đổi. Do đó các tính chất 4 và 5 đã được khôi phục. Nút trắng trong hình vẽ có thể là đỏ hoặc đen, nhưng phải ghi lại trước và sau khi thay đổi. |

void delete\_case6(struct node \*n) {

sibling(n)->color = n->parent->color;

n->parent->color = BLACK;

if (n == n->parent->left) {

/\* Here, sibling(n)->right->color == RED \*/

sibling(n)->right->color = BLACK;

rotate\_left(n->parent);

}

else

{

/\* Here, sibling(n)->left->color == RED \*/

sibling(n)->left->color = BLACK;

rotate\_right(n->parent);

}

}



Bộ dữ liệu thứ nhất: 7, 3, 18, 10, 11, 22

7

3 11

10 18

22

Ta thêm số 9: sẽ là con trái của 10 và rơi vào insert\_case2 ( cha của 9 là đen ) nên ta chỉ chỉ cần return màu đỏ của 9 là ok

7

3 11

10 18

9 22

Ta thêm tiếp số 8: nó sẽ là con trái của 9 thì sẽ nhảy đến trường hợp 5, ta dổi màu cho cha của 8 là 9 thành màu đen, ông của 8 thành đỏ. Vì 8 là con trái của 9, 9 là con trái của 10 nên ta thực hiên xoay phải: rotate\_right(grandparent(8), root);

7

3 11

9 18

8 10 22

**\*phép xóa**

1,75

1,25 4

1 1,5 2 5

1,125 3

Ta xóa 1,75 sẽ nhảy vào trường hợp 1 ( nút bị xóa là nút gốc)

1,5

1,25 4

1 2 5

1,125 3

Bộ dữ liệu thứ 2: 313, 334, 51, 434, 283, 24, 484

313

51 434

24 283 334 484

Ta xóa số 51, trước tiên thuật toán sẽ đi tìm node có giá trị là 51, sau khi tìm được thì thực hiện hàm BS\_Delete() để xóa node 51. Lúc này node con phải của 51 sẽ lên chèn vào vị trí của 51 ( rơi vào trường hợp 51 là đen, con của nó là đỏ) vì vậy ta đổi màu con của 51 thành đen là được

313

283 434

24 334 484