**Cấu trúc cây nhị phân là gì? Hoạt động ra sao?**

Trong bài này mình sẽ giới thiệu các bạn một trong các cấu trúc dữ liệu tiếp theo đó chính là cấu trúc dữ liệu dạng cây. Đây là một dạng cấu trúc được sử dụng rất nhiều trong tìm kiếm, nó được tối ưu nhất trong các cấu trúc dữ liệu mà mình đã giới thiệu.



Chúng ta sẽ cùng nhau tìm hiểu về cấu trúc dữ liệu cây là gì? Có các loại cấu trúc dữ liệu cây nào và cách thức hoạt động của nó.

**Mục lục**

* [1. Cấu trúc dữ liệu cây là gì?](https://freetuts.net/cau-truc-cay-nhi-phan-3020.html#goto-h2-0)
* [2. Cây nhị phân (Binary tree)](https://freetuts.net/cau-truc-cay-nhi-phan-3020.html#goto-h2-1)
* [3. Cây nhị phân tìm kiếm (Binary search tree)](https://freetuts.net/cau-truc-cay-nhi-phan-3020.html#goto-h2-2)
  + [Biểu diễn cây nhị phân tìm kiếm](https://freetuts.net/cau-truc-cay-nhi-phan-3020.html#goto-h3-0)
  + [Các thao tác trên cây nhị phân tìm kiếm](https://freetuts.net/cau-truc-cay-nhi-phan-3020.html#goto-h3-1)
* [4. Kết luận](https://freetuts.net/cau-truc-cay-nhi-phan-3020.html#goto-h2-3)

**1. Cấu trúc dữ liệu cây là gì?**

Cấu trúc dữ liệu cây là một cấu trúc biểu diễn các Node dưới dạng cây. Như các bạn đã học ở môn lập trình C/C++ thì khi chúng ta muốn lưu các phần tử, ta có thể lưu chúng dưới dạng mảng một chiều. Hoặc có thể lưu dưới dạng một danh sách liên kết. Tương tự như vậy các bạn cũng có thể lưu dưới dạng cây nhị phân.

Ưu điểm của cấy trúc dữ liệu cây so với các cấu trúc khác là:

*Bài viết này được đăng tại [free tuts .net]*

* Phân cấp dữ liệu.
* Tìm kiếm dễ dàng hơn.
* Thao tác trên các danh sách dữ liệu đã sắp xếp.

Trong cấu trúc dữ liệu cây, có hai cấu trúc chính đó là cấu trúc cây nhị phân và cấu trúc cây nhị phân tìm kiếm. Sau đây chúng ta sẽ tìm hiểu qua về hai cấu trúc dữ liệu này nhé.

**2. Cây nhị phân (Binary tree)**

Cây nhị phân là một cấu trúc dữ liệu được sử dụng cho mục đích lưu trữ dữ liệu. Một cây nhị phân bao gồm các Node và mỗi Node bao gồm 3 thành phần:

* **Data**: Giá trị của một phần tử
* **Left pointer**: Con trỏ trỏ đến cây nhị phân bên trái Node.
* **Right pointer**: Con trỏ trỏ đến cây nhị phân bên phải Node.



Các thành phần cơ bản của cây nhị phân bao gồm:

* **Root**: Được gọi là Node gốc của cây (là một Node cha), một cây chỉ có một Node gốc duy nhất và nó không có Node cha nào.
* **Parent Node**: Là Node cha của một Node cụ thể nào đó.
* **Child Node**: Là Node con của một Node cụ thể nào đó.
* **Sub-tree**: Là cây con biểu diển các con của một Node.
* **LeafNode**: Là Node không có Node con.
* **Siblings**: Các Node có cùng một cha.
* **Internal Node**: Node có ít nhất một Node con.
* **External Node**: Node không có Node con nào.

**3. Cây nhị phân tìm kiếm (Binary search tree)**

Cây nhị phân tìm kiếm là một dạng đặc biệt của cây nhị phân. Về cơ bản nó có đủ tất cả các thành phần của cây nhị phân. Nhưng tất cả các Node của nó đều có chung một đặc điểm sau:

* Cây con bên trái của một Node luôn luôn có giá trị nhỏ hơn hoặc bằng giá trị của Node cha phía trên nó.
* Cây con bên phải của một Node luôn luôn có giá trị lớn hơn hoặc bằng giá trị của Node cha phía trên nó.

**Biểu diễn cây nhị phân tìm kiếm**

Cây nhị phân tìm kiếm là một tập hợp các Node được sắp xếp và lưu trữ theo một quy tắc nhất định. Dựa vào quy tắc đó chúng ta có thể duy trì và thực hiện các thao tác trên cây nhị phân tìm kiếm. Các bạn hãy xem hình dưới đây để hiểu rõ hơn về quy tắc của nó:



Giả sử ta có các phần tử số nguyên như sau: 27, 14, 35, 10, 19, 31, 32. Quá trình lưu trữ các phần tử này theo cấu trúc cây nhị phân tìm kiếm được thực hiện như sau:

1. Số 27 sẽ được lưu trữ vào cây đầu tiên và nó được lấy làm key để so sánh.
2. Số 14 được so sánh với số 27 (key), vì 14 < 27 nên sẽ được lưu trữ vào bên trái số 27 (key).
3. Số 35 > 27 vì vậy sẽ được lưu trữ vào bên phải số 27.
4. Tiếp tục số 10 < 27 và 10 < 14 vì vậy nó sẽ nằm bên trái số 14.
5. Số 19 < 27 và 19 > 14 vì vậy nó sẽ nằm bên phải số 14.
6. Số 31 > 27 và 31 < 35 vì vậy nó sẽ nằm bên trái số 35.
7. Cuối cùng số 42 > 27 và 42 > 35 vị vậy nó sẽ nằm bên phải số 35.

Sau khi thực hiện lưu trữ các phần tử số nguyên trên vào cây ta được một cây như trong hình.

**Các thao tác trên cây nhị phân tìm kiếm**

Trong cây nhị phân tìm kiếm ta có thể thực hiện các thao tác sau:

* Chèn một phần tử vào trong một cây.
* Tìm kiếm phần tử trong cây.
* Duyệt cây.
* Đo chiều cao của cây.

Trên đây là các thao tác thường được sử dụng nhiều trong cây. Đặc biệt là tìm kiếm phần tử trong cây, như cái tên của nó là cây nhị phân tìm kiếm. Đây là một cấu trúc dữ liệu được sử dụng trong các bài toán tìm kiếm rất nhiều, bởi tính chính xác và tốc độ của nó.

**4. Kết luận**

Như vậy là chúng ta đã tìm hiểu xong về cấu trúc dữ liệu cây là gì. Và các hai cấu trúc dữ liệu cây nhị phân và nhị phân tìm kiếm. Các bạn hãy sử dụng các cấu trúc dữ liệu thật linh hoạt nhé, bởi mỗi cấu trúc dữ liệu đều có các ưu điểm nhất định. Ở bài tiếp theo mình sẽ hướng dẫn các cấu trúc dữ liệu của cây và cách thêm Node vào cây, hãy chú ý theo dõi nhé !!!

**Thêm Node vào cây nhị phân tìm kiếm**

**Trong hướng dẫn này mình sẽ giới thiệu các bạn về cấu trúc dữ liệu của cây nhị phân tìm kiếm và cách thêm một Node vào cây nhị phân tìm kiếm.**



Chúng ta sẽ tìm hiểu lần lượt về cấu trúc một Node trong cây và khởi tạo cho cây như thế nào. Sau đó sẽ thực hiện thêm Node vào cây.

**Mục lục**

* [1. Cấu trúc dữ liệu của cây nhị phân tìm kiếm](https://freetuts.net/them-node-vao-cay-nhi-phan-tim-kiem-3021.html#goto-h2-0)
  + [Khởi tạo Node](https://freetuts.net/them-node-vao-cay-nhi-phan-tim-kiem-3021.html#goto-h3-0)
  + [Khởi tạo cây](https://freetuts.net/them-node-vao-cay-nhi-phan-tim-kiem-3021.html#goto-h3-1)
* [2. Thêm Node vào cây nhị phân tìm kiếm](https://freetuts.net/them-node-vao-cay-nhi-phan-tim-kiem-3021.html#goto-h2-1)
* [3. Kết luận](https://freetuts.net/them-node-vao-cay-nhi-phan-tim-kiem-3021.html#goto-h2-2)

**1. Cấu trúc dữ liệu của cây nhị phân tìm kiếm**

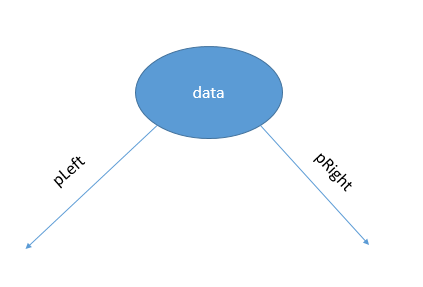
Như các bạn đã học ở bài trước thì cây nhị phân tìm kiếm là một cấu trúc dữ liệu, vì vậy ta cần khởi tạo cấu trúc dữ liệu cho nó. Trong phần này ta có khởi tạo cấu trúc của một Node và khởi tạo cây.

**Khởi tạo Node**

Trong một Node ta có ba thành phần như sau:

*Bài viết này được đăng tại [free tuts .net]*

* **Data**là giá trị của Node
* **pLeft**là con trỏ trỏ đến cây con bên trái Node
* **pRight**là con trỏ trỏ đến cây con bên phải Node



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | struct node  {  **int** data; // dữ liệu của node ==> dữ liệu mà node sẽ lưu trữ      struct node \*pLeft; // node liên kết bên trái của cây <=> cây con trái      struct node \*pRight; // node liên kết bên phải của cây <=> cây con phải  }; |

**Khởi tạo cây**

Để làm việc được với cây ta cần khởi tạo cây và gán giá trị cho nó như danh sach liên kết đơn vậy.

Chúng ta sẽ tạo một cây **t** có tham số truyền vào cho hàm khởi tạo cây là một con trỏ \*Node (đây là node vừa được khởi tạo ở trên). Trong hàm KhoiTaoCay() ta thực hiện cho khởi tạo cây bằng rỗng: t = null

|  |  |
| --- | --- |
| 1  2  3  4  5 | /\* khởi tạo cây \*/  void KhoiTaoCay(TREE &t)  {      t = NULL; // cây rỗng  } |

**2. Thêm Node vào cây nhị phân tìm kiếm**

Sau khi chúng ta đã tạo cấu trúc dữ liệu cho cây, ta sẽ thực hiện thêm Node vào cây nhị phân tìm kiếm.

Trong hàm ThemNodeVaoCay() ta truyền hai tham số là cây **t**và Node x.

Để thêm Node mới vào cây ta cần xét hai trường hợp:

**Trường hợp 1:**Trong cây không có phần tử nào (cây rỗng).

Ta sử dụng struct dode tạo một Node mới để thêm vào cây, sau đó thêm dữ liệu x vào data (p->data = x).

Khởi tạo cho con trỏ pLeft và pRight bằng Null. Vì cây ban đầu không có phân tử nào, vì vậy phần tử đầu tiên khi thêm vào chính là phần tử gốc t = p

**Trường hợp 2:**Trong cây có tồn tại phần tử.

Khi này trong cây đã có phần tử, cũng như đã có phần tử gốc (root). Vì vậy khi ta thêm Node mới vào cần so sánh Node này với Node gốc để có thể thêm chính xác vị trí cần thêm.

Như ta đã học ở bài trước thì một phần tử khi được thêm vào nếu nhỏ hơn phần tử gốc thì nằm ở bên cây con trái và nếu lớn hơn phần tử gốc thì nằm ở bên cây con phải.

Trong trường hợp này ta sẽ sử dụng đệ quy để thực hiện gọi lại hàm ThemNodeVaoCay() duyệt qua trái để thêm phần tử x nếu x < root và duyệt qua phải nếu x > root

/\* hàm thêm phần tử x vào cây nhị phân \*/

void ThemNodeVaoCay(TREE &t, int x)

{

// nếu cây rỗng

if (t == NULL)

{

NODE \*p = new NODE; // khởi tạo 1 cái node để thêm vào cây

p->data = x;// thêm dữ liệu x vào data

p->pLeft = NULL;

p->pRight = NULL;

t = p;// node p chính là node gốc <=> x chính là node gốc

}

else // cây có tồn tại phần tử

{

// nếu phần tử thêm vào mà nhỏ hơn node gốc ==> thêm nó vào bên trái

if (t->data > x)

{

ThemNodeVaoCay(t->pLeft, x); // duyệt qua trái để thêm phần tử x

}

else if (t->data < x) // nếu phần tử thêm vào mà lớn hơn node gốc ==> thêm nó vào bên phải

{

ThemNodeVaoCay(t->pRight, x); // duyệt qua phải để thêm phần tử x

}

}

}

**3. Kết luận**

Như vậy là chúng ta đã tìm hiểu xong về cấu trúc dữ liệu của cây nhị phân tìm kiếm, cũng như cách thêm một Node vào cây. Đây là bước đầu tiên để tạo dữ liệu cho cây. Ở bài tiếp theo mình sẽ thực hiện duyệt cây để có thể kiếm tra được các thao tác của ta trên cây đúng hay sai. Các bạn chú ý theo dõi nhé!!

**Duyệt cây nhị phân tìm kiếm**

Trong bài này mình sẽ giới thiệu các bạn các cách duyệt cây nhị phân tìm kiếm. Đây là một bước rất quan trọng để kiếm ra kết quả và hiển thị các phần tử trong cây.



Chúng ta sẽ tìm hiểu lần lượt 6 cách duyệt cây nhị phân tìm kiếm:

* Duyệt NLR cây nhị phân tìm kiếm.
* Duyệt NRL cây nhị phân tìm kiếm.
* Duyệt LNR cây nhị phân tìm kiếm.
* Duyệt RNL cây nhị phân tìm kiếm.
* Duyệt LRN cây nhị phân tìm kiếm.
* Duyệt RLN cây nhị phân tìm kiếm.

**Mục lục**

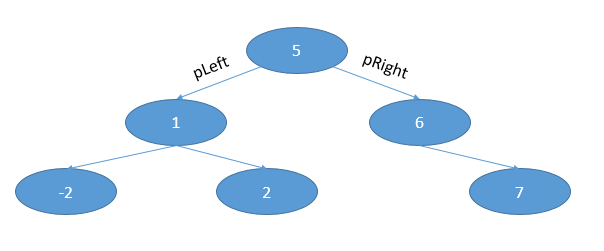
* [1. Duyệt NLR cây nhị phân tìm kiếm](https://freetuts.net/duyet-cay-nhi-phan-tim-kiem-3024.html#goto-h2-0)
* [2. Duyệt NRL cây nhị phân tìm kiếm.](https://freetuts.net/duyet-cay-nhi-phan-tim-kiem-3024.html#goto-h2-1)
* [3. Duyệt LNR cây nhị phân tìm kiếm.](https://freetuts.net/duyet-cay-nhi-phan-tim-kiem-3024.html#goto-h2-2)
* [4. Duyệt RNL cây nhị phân tìm kiếm.](https://freetuts.net/duyet-cay-nhi-phan-tim-kiem-3024.html#goto-h2-3)
* [5. Duyệt LRN cây nhị phân tìm kiếm.](https://freetuts.net/duyet-cay-nhi-phan-tim-kiem-3024.html#goto-h2-4)
* [6. Duyệt RLN cây nhị phân tìm kiếm.](https://freetuts.net/duyet-cay-nhi-phan-tim-kiem-3024.html#goto-h2-5)
* [7. Code mẫu trong C++](https://freetuts.net/duyet-cay-nhi-phan-tim-kiem-3024.html#goto-h2-6)
* [Kết luận](https://freetuts.net/duyet-cay-nhi-phan-tim-kiem-3024.html#goto-h2-7)

**1. Duyệt NLR cây nhị phân tìm kiếm**

Trong phần này mình sẽ giới thiệu các bạn duyệt cây theo cách NLR (Node -> Left -> Right).

Giả sử chúng ta có một dãy số bao gồm các số: 5, 1, 2, -2, 6, 7. Ta sẽ thêm lần lượt các số này vào cây, sau khi thêm ta được cây như sau:

*Bài viết này được đăng tại [free tuts .net]*



Duyệt NLR là chúng ta sẽ thực hiện xuất Node trước rồi thực hiện cây con bên trái và cây con bên phải. Mình sẽ lấy ví dụ trên để thực hiện duyệt NLR:

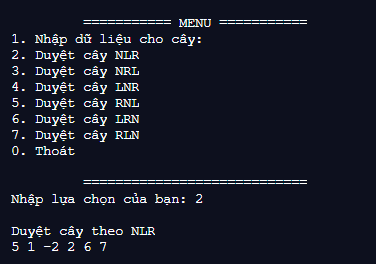
1. Đầu tiên ta vào Node đầu tiên là số 5, theo quy tắc NLR thì ta sẽ xuất số **5** ra rồi thực hiện duyệt Left.
2. Khi duyệt Left ta có một Node mới là số **1,**theo quy tắc NLR thì ta sẽ xuất số **1** ra thồi thực hiện duyệt Left.
3. Tiếp tục duyệt Left ta có một số Node -2 , theo quy tắc NLR thì ta xuất số **-2** ra rồi thực hiện duyệt Left. Khi này Left không còn Node ta duyệt qua Right cũng không còn Node. Ta sử dụng đệ quy để quay lên duyệt Right ở Node số 1.
4. Duyệt Right ở Node số 1 ta có một Node 2, theo quy tắc NLR thì ta xuất số **2** ra và thực hiện duyệt Left. Left không còn phần tử nào ta duyệt qua Right cũng không còn phần tử nào. Khi đó ta sử dụng đệ quy quay lại Node số 5 để thực hiện duyệt Right.
5. Cứ như vậy ta duyệt Right Node số 5 ta được Node số **6** và Node số **7**.

Sau khi duyệt theo tuần tự như vậy ta được một dãy số mới: 5, 1, -2, 2, 6, 7. Khác với dãy số ban đầu thêm vào cây, từ đây ta có một kết luận rằng:

**Kết luận:**Với mỗi cách duyệt khác nhau sẽ cho ra kết quả khác nhau, ta có tất cả 6 cách duyệt và các cách duyệt này đều cho ra kết quả khác nhau.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\* hàm xuất cây nhị phân theo NLR \*/  void Duyet\_NLR(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          cout << t->data << " "; // xuất dữ liệu trong node          Duyet\_NLR(t->pLeft); // duyệt qua trái          Duyet\_NLR(t->pRight); // duyệt qua phải      }  } |

**Kết quả:** Thực hiên Code trong C++



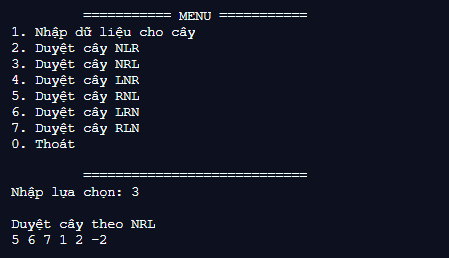
**2. Duyệt NRL cây nhị phân tìm kiếm.**

Tương tự như duyệt NLR cây nhị phân, ta thực hiện xuất lần lượt các Node theo thứ tự NRL.

Với dãy số như trên: 5, 1, 2, 2, 6, 7 ta thực hiện duyệt NRL. Sau khi duyệt ta được kết quả như sau: 5, 6, 7, 1, 2, -2.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // hàm xuất cây nhị phân theo NRL  void Duyet\_NRL(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          cout << t->data << " "; // xuất dữ liệu trong node          Duyet\_NRL(t->pRight); // duyệt qua phải          Duyet\_NRL(t->pLeft); // duyệt qua trái      }  } |

**Kết quả:**

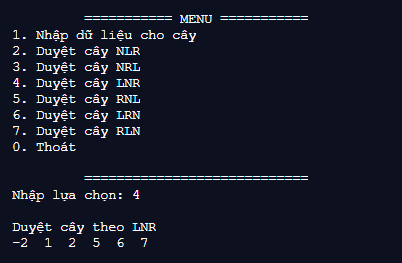


**3. Duyệt LNR cây nhị phân tìm kiếm.**

Duyệt LNR cây nhị phân tìm kiếm ta thực hiện duyệt theo thứ tự Left -> Node -> Right. Ta sử dụng các số 5, 1, 2, -2, 6, 7. Khi ta sử dụng cách này để thực hiện duyệt cây, các phần tử sẽ được sắp xếp tăng dần khi xuất ra màn hình.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // hàm xuất cây nhị phân theo LNR <=> xuất ra các phần tử từ bé đến lớn  void Duyet\_LNR(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          Duyet\_LNR(t->pLeft); // duyệt qua trái          cout << t->data << "  "; // xuất giá trị của node          Duyet\_LNR(t->pRight); // duyệt qua phải      }  } |

**Kết quả:**

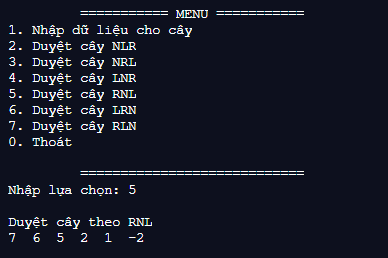


**4. Duyệt RNL cây nhị phân tìm kiếm.**

Duyệt RNL cây nhị phân tìm kiếm ta thực hiện duyệt theo thứ tự Right -> Node -> Left. Sử dụng cách duyệt này sẽ cho chúng ta kết quả là một dãy số sắp xếp theo thứ tự giảm dần.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // hàm xuất cây nhị phân theo RNL <=> xuất ra các phần tử từ lớn đến bé  void Duyet\_RNL(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          Duyet\_RNL(t->pRight); // duyệt qua phải          cout << t->data << "  "; // xuất giá trị của node          Duyet\_RNL(t->pLeft); // duyệt qua phải      }  } |

**Kết quả:**

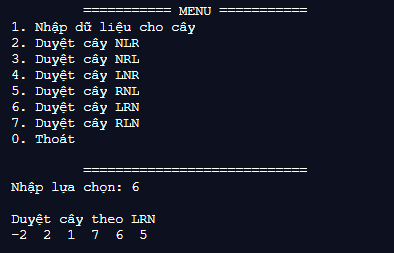


**5. Duyệt LRN cây nhị phân tìm kiếm.**

Duyệt LRN cây nhị phân tìm kiếm ta thực hiện duyệt theo thứ tự Left -> Right -> Node. Về cơ bản các cách duyệt này hoạt động tương tự nhau, chỉ khác thứ tự duyệt mà thôi.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // hàm xuất cây nhị phân theo LRN  void Duyet\_LRN(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          Duyet\_LRN(t->pLeft); // duyệt qua trái          Duyet\_LRN(t->pRight); // duyệt qua phải          cout << t->data << "  "; // xuất giá trị của node      }  } |

**Kết quả:**

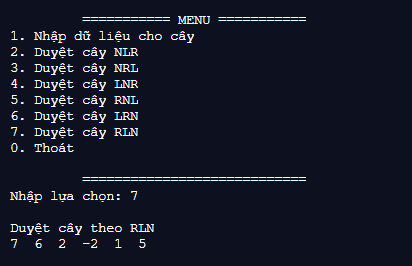


**6. Duyệt RLN cây nhị phân tìm kiếm.**

Và cuối cùng chúng ta sẽ thực hiện duyệt RLN cây nhị phân tìm kiếm. Ta cũng sẽ thực hiện theo thứ tự Right -> Left -> Node, mình sẽ lấy ví dụ dãy số ở trên để chạy thử cho cách duyệt này nhé.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | // hàm xuất cây nhị phân theo RLN  void Duyet\_RLN(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          Duyet\_RLN(t->pRight); // duyệt qua phải          Duyet\_RLN(t->pLeft); // duyệt qua trái          cout << t->data << "  "; // xuất giá trị của node      }  } |

**Kết quả:**



**7. Code mẫu trong C++**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193 | #include<iostream>  using namespace std;  // khai báo cấu trúc  1 cái node  struct node  {  **int** data; // dữ liệu của node ==> dữ liệu mà node sẽ lưu trữ      struct node \*pLeft; // node liên kết bên trái của cây <=> cây con trái      struct node \*pRight; // node liên kết bên phải của cây <=> cây con phải  };  typedef struct node NODE;  typedef NODE\* TREE;    // khởi tạo cây  void KhoiTaoCay(TREE &t)  {      t = NULL; // cây rỗng  }    // hàm thêm phần tử x vào cây nhị phân  void ThemNodeVaoCay(TREE &t, **int** x)  {      // nếu cây rỗng      if (t == NULL)      {          NODE \*p = new NODE; // khởi tạo 1 cái node để thêm vào cây          p->data = x;// thêm dữ liệu x vào data          p->pLeft = NULL;          p->pRight = NULL;          t = p;// node p chính là node gốc <=> x chính là node gốc      }      else // cây có tồn tại phần tử      {          // nếu phần tử thêm vào mà nhỏ hơn node gốc ==> thêm nó vào bên trái          if (t->data > x)          {              ThemNodeVaoCay(t->pLeft, x); // duyệt qua trái để thêm phần tử x          }          else if (t->data < x) // nếu phần tử thêm vào mà lớn hơn node gốc ==> thêm nó vào bên phải          {              ThemNodeVaoCay(t->pRight, x); // duyệt qua phải để thêm phần tử x          }      }  }    // hàm xuất cây nhị phân theo NLR  void Duyet\_NLR(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          cout << t->data << " "; // xuất dữ liệu trong node          Duyet\_NLR(t->pLeft); // duyệt qua trái          Duyet\_NLR(t->pRight); // duyệt qua phải      }  }    // hàm xuất cây nhị phân theo NRL  void Duyet\_NRL(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          cout << t->data << " "; // xuất dữ liệu trong node          Duyet\_NRL(t->pRight); // duyệt qua phải          Duyet\_NRL(t->pLeft); // duyệt qua trái      }  }    // hàm xuất cây nhị phân theo LNR <=> xuất ra các phần tử từ bé đến lớn  void Duyet\_LNR(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          Duyet\_LNR(t->pLeft); // duyệt qua trái          cout << t->data << "  "; // xuất giá trị của node          Duyet\_LNR(t->pRight); // duyệt qua phải      }  }    // hàm xuất cây nhị phân theo RNL <=> xuất ra các phần tử từ lớn đến bé  void Duyet\_RNL(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          Duyet\_RNL(t->pRight); // duyệt qua phải          cout << t->data << "  "; // xuất giá trị của node          Duyet\_RNL(t->pLeft); // duyệt qua phải      }  }    // hàm xuất cây nhị phân theo LRN  void Duyet\_LRN(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          Duyet\_LRN(t->pLeft); // duyệt qua trái          Duyet\_LRN(t->pRight); // duyệt qua phải          cout << t->data << "  "; // xuất giá trị của node      }  }    // hàm xuất cây nhị phân theo RLN  void Duyet\_RLN(TREE t)  {      // nếu cây còn phần tử thì tiếp tục duyệt      if (t != NULL)      {          Duyet\_RLN(t->pRight); // duyệt qua phải          Duyet\_RLN(t->pLeft); // duyệt qua trái          cout << t->data << "  "; // xuất giá trị của node      }  }      // hàm nhập dữ liệu  void Menu(TREE &t)  {      while (true)      {          cout << "\n\n\t\t =========== MENU ===========";          cout << "\n1. Nhập dữ liệu cho cây: ";          cout << "\n2. Duyệt cây NLR";          cout << "\n3. Duyệt cây NRL";          cout << "\n4. Duyệt cây LNR";          cout << "\n5. Duyệt cây RNL";          cout << "\n6. Duyệt cây LRN";          cout << "\n7. Duyệt cây RLN";          cout << "\n0. Thoát";          cout << "\n\n\t\t ============================";    **int** luachon;          cout << "\nNhập lựa chọn của bạn: ";          cin >> luachon;          if (luachon < 0 || luachon > 7)          {              cout << "\nLựa chọn không hợp lệ";          }          else if (luachon == 1)          {  **int** x;              cout << "\nNhập số nguyên x: ";              cin >> x;              ThemNodeVaoCay(t, x);          }          else if (luachon == 2)          {              cout << "\nDuyệt cây theo NLR\n";              Duyet\_NLR(t);          }          else if (luachon == 3)          {              cout << "\nDuyệt cây theo NRL\n";              Duyet\_NRL(t);          }          else if (luachon == 4)          {              cout << "\nDuyệt cây theo LNR\n";              Duyet\_LNR(t);          }          else if (luachon == 5)          {              cout << "\nDuyệt cây theo RNL\n";              Duyet\_RNL(t);          }          else if (luachon == 6)          {              cout << "\nDuyệt cây theo LRN\n";              Duyet\_LRN(t);          }          else if (luachon == 7)          {              cout << "\nDuyệt cây theo RLN\n";              Duyet\_RLN(t);          }          else          {              break;          }      }  }    **int** main()  {      TREE t;      KhoiTaoCay(t);      Menu(t);        cout<<"\n-------------------------------------\n";    cout<<"Chương trình này được đăng tại Freetuts.net";  } |

**Kết luận**

Như vậy là chúng ta đã thực hiện xong các cách duyệt cây nhị phân. Tuy nhiên trong mỗi trường hợp khác nhau ta cần sử dụng cách duyệt khác nhau, các bạn hãy sử dụng nó một cách linh hoạt. Như các bạn đã thấy thì mỗi cách duyệt đều cho ra kết quả khác nhau mặc dù đầu vào của nó là giống nhau.

Ở bài tiếp theo mình sẽ hướng dẫn các bạn cách tìm kiếm trong cây, đây là một thao tác được xem như là đặc biệt nhất của cây nhị phân tìm kiếm (như cái tên của nó). Các bạn hãy chú ý theo dõi nhé !!

**Tìm kiếm Node trên cây nhị phân tìm kiếm**

Trong hướng dẫn này mình sẽ giới thiệu các bạn cách tìm kiếm một Node trên cây nhị phân tìm kiếm. Đây là thao tác đặc biệt nhất trong cây nhị phân tìm kiếm, vì nó được thực hiện một cách dễ dàng và nhanh chóng.



Chúng ta sẽ tìm hiều về cách hoạt động của hàm tìm kiếm trong cây như thế nào, sau đó thực hiện một ví dụ tìm kiếm cụ thể để các bạn hiểu rõ hơn.

**Mục lục**

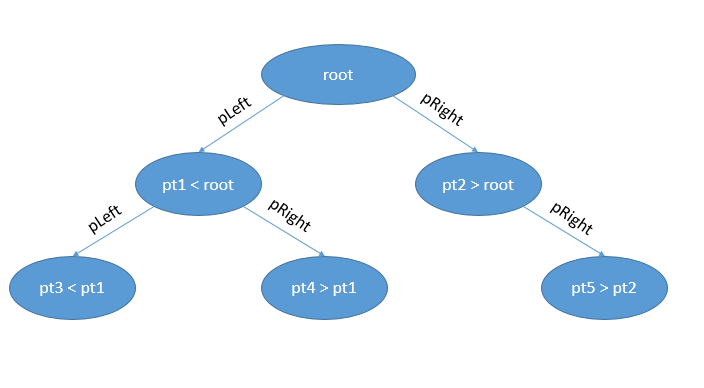
* [1. Tìm kiếm Node trong cây nhị phân tìm kiếm](https://freetuts.net/tim-kiem-node-tren-cay-nhi-phan-tim-kiem-3025.html#goto-h2-0)
* [2. Ví dụ: Tìm kiếm phần tử x trong cây nhị phân tìm kiếm](https://freetuts.net/tim-kiem-node-tren-cay-nhi-phan-tim-kiem-3025.html#goto-h2-1)
* [3. Kết luận](https://freetuts.net/tim-kiem-node-tren-cay-nhi-phan-tim-kiem-3025.html#goto-h2-2)

**1. Tìm kiếm Node trong cây nhị phân tìm kiếm**

Như các bạn đã biết thì thao tác tìm kiếm là một thao tác rất phổ biến trong các cấu trúc dữ liệu. Mỗi cấu trúc có mỗi cách tìm kiếm khác nhau, trong phần này các bạn hãy cùng mình tìm hiểu xem cách tìm kiếm trong cây nhị phân tìm kiếm như thế nào nhé.

Để thực hiện tìm kiếm đầu tiên các bạn phải nắm được cấu trúc lưu trữ của cây nhị phân tìm kiếm như thế nào, khi đó các bạn mới dựa vào đó để tìm kiếm. Khi lưu trữ các phần tử nhỏ hơn Node gốc (root) được lưu trữ ở cây con trái và các phần tử lớn hơn root được lưu ở cây con phải. Vì vậy khi tìm kiếm ta cũng thực hiện so sánh với root để tìm.

*Bài viết này được đăng tại [free tuts .net]*



Việc tìm kiếm một Node trong cây đơn giản chỉ là kiểm tra xem phần tử đó còn tồn tại trong cây hay không, nếu có thì trả về Node đó, ngược lại nếu không thì trả về NULL.

Ta có một hàm TimKiem() với tham số truyền vào là một cây **t**và một số nguyên **x**(ta sẽ tìm một số nguyên x trong cây số nguyên).

Việc đầu tiền là ta sẽ kiểm tra xem cây rỗng hay không, nếu cây rỗng thì trả về NULL, nếu cây có phần tử thì khi đó ta bắt đầu thực hiện tìm kiếm. Ta sẽ có 3 trường hợp khi thực hiện tìm kiếm trong cây:

**Trường hợp 1: Phần tử x cần tìm kiếm nhỏ hơn t -> data.**

Khi đó ta sẽ thực hiện gọi đệ quy hàm TimKiem() để duyệt sang bên cây con trái đề tìm phần tử x.

**Trường hợp 2: Phần tử x cần tìm kiếm lớn hơn t -> data.**

Khi đó ta sẽ thực hiện gọi đệ quy hàm TimKiem() để duyệt sang bên cây con phải để tìm phần tử x.

**Trường hợp 3: Phần tử x cần tìm kiếm bằng t -> data.**

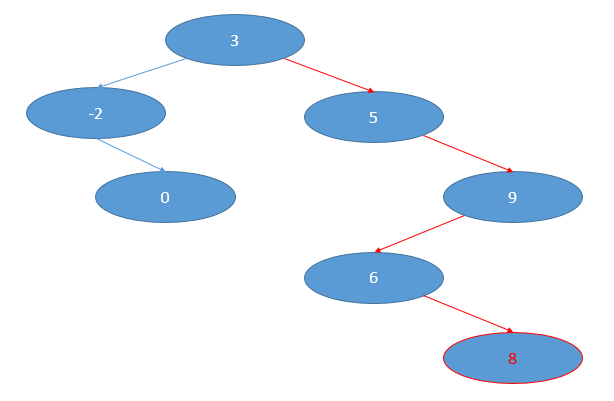
Trong trường hợp này đơn giản ta chỉ cần trả về t -> data, vì đây chính là giá trị cần tìm.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | // nếu node có tồn tại trong cây thì trả về cái node đó - còn không tồn tại thì trả về NULL  NODE\* TimKiem(TREE t, **int** x)  {      // nếu cây rỗng      if (t == NULL)      {          return NULL;      }      else      {          // nếu phần tử cần tìm kiếm mà nhỏ hơn node gốc thì duyệt(đệ quy) sang bên trái để tìm          if (x < t->data)          {              TimKiem(t->pLeft, x);          }          else if (x > t->data) // duyệt sang bên phải          {              TimKiem(t->pRight, x);          }          else // <=> t->data == x          {              return t; // trả về node cần tìm kiếm          }      }    } |

**2. Ví dụ: Tìm kiếm phần tử x trong cây nhị phân tìm kiếm**

Trong ví dụ về tìm kiếm phần tử x trong cây nhị phân, mình sẽ thực hiện tìm kiếm một số nguyên x trong cây số nguyên bao gồm các số sau: 3, 5, -2, 0, 9, 6, 8.

Giả sử mình cần tìm số 8 trong dãy các số trên, khi đó việc tìm kiếm sẽ thực hiện theo các bước sau: Biểu diễn bằng mũi tên màu đỏ.

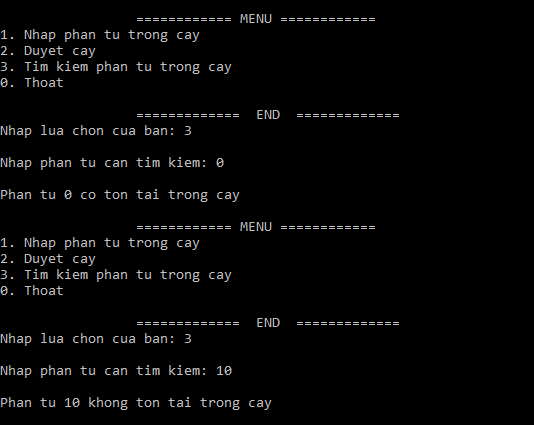


Đầu tiên nó sẽ so sánh số 8 (số cần tìm) với root là số 3, vì 8 > 3 nên nó sẽ gọi đệ quy và duyệt sang cây con phải. Tiếp tục so sánh với số 5, 8 > 5 vì vậy gọi đệ quy và duyệt sang cây con phải.

Đến đây tiếp tục so sánh với số 9, khi này 8 < 9 nên sẽ gọi đệ quy và duyệt sang trái. Khi duyệt sang trái tiếp tục só sánh với số 6 , vì 8 > 6 nên sẽ gọi đệ quy và duyệt qua bên phải. Đến đây khi so sánh thì số cần tìm là số 8 lại bằng chính với t -> data, khi đó đã tìm thấy số 8.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146 | #include<iostream>  using namespace std;  // nhập vào cây nhị phân tìm kiếm các số nguyên  // khai báo cấu trúc 1 cái node trong cây nhị phân tìm kiếm  struct node  {  **int** data; // dữ liệu chứa trong 1 cái node      struct node \*pLeft; // con trỏ liên kết với cái node bên trái <=> cây con trái      struct node \*pRight; // con trỏ liên kết với cái node bên phải <=> cây con phải  };  typedef struct node NODE;  typedef NODE\* TREE;    // hàm khởi tạo cây nhị phân tìm kiếm  void KhoiTaoCay(TREE &t)  {      t = NULL;  }    // hàm thêm 1 cái phần tử vào cây  void ThemNodeVaoCay(TREE &t, **int** x)  {      // nếu cây rỗng      if (t == NULL)      {          NODE \*p = new NODE;          p->data = x;          p->pLeft = NULL;          p->pRight = NULL;          t = p;      }      else      {          // nếu phần tử thêm vào mà nhỏ hơn nút gốc thì sẽ duyệt qua bên trái          if (x < t->data)          {              ThemNodeVaoCay(t->pLeft, x);          }          else if (x > t->data)          {              ThemNodeVaoCay(t->pRight, x);          }      }  }    // hàm xuất phần tử trong cây nhị phân tìm kiếm  void NLR(TREE t)  {      if (t != NULL)      {          // xử lí          cout << t->data << "  ";          NLR(t->pLeft);          NLR(t->pRight);      }  }    // nếu node có tồn tại trong cây thì trả về cái node đó - còn không tồn tại thì trả về NULL  NODE\* TimKiem(TREE t, **int** x)  {      // nếu cây rỗng      if (t == NULL)      {          return NULL;      }      else      {          // nếu phần tử cần tìm kiếm mà nhỏ hơn node gốc thì duyệt(đệ quy) sang bên trái để tìm          if (x < t->data)          {              TimKiem(t->pLeft, x);          }          else if (x > t->data) // duyệt sang bên phải          {              TimKiem(t->pRight, x);          }          else // <=> t->data == x          {              return t; // trả về node cần tìm kiếm          }      }  }    // hàm nhập cây nhị phân tìm kiếm  void Menu(TREE &t)  {  **int** luachon;      while(true)      {          cout << "\n\n\t\t ============ MENU ============";          cout << "\n1. Nhập phần tử trong cây";          cout << "\n2. Duyệt cây";          cout << "\n3. Tìm kiếm phần tử trong cây";          cout << "\n0. Thoát";          cout << "\n\n\t\t =============  END  =============";            cout << "\nNhập lựa chọn của bạn: ";          cin >> luachon;        if(luachon < 0 || luachon > 3){        cout<<"\nLựa chọn của bạn không hợp lệ";      }          else if (luachon == 1)          {  **int** x;              cout << "\nNhập giá trị: ";              cin >> x;              ThemNodeVaoCay(t, x);          }          else if (luachon == 2)          {              cout << "\n\t Cây nhị phân tìm kiếm\n";              NLR(t);          }          else if (luachon == 3)          {  **int** x;              cout << "\nNhập phần tử cần tìm kiếm: ";              cin >> x;              NODE \*p = TimKiem(t, x);              if (p == NULL)              {                  cout << "\nPhần tử " << x << " không tồn tại trong cây";              }              else              {                  cout << "\nPhần tử " << x << " có tồn tại trong c";              }          }          else          {              break;          }      }  }      **int** main()  {      TREE t;      KhoiTaoCay(t);      Menu(t);        cout<<"\n-------------------------------------\n";          cout<<"Chương trình này được đăng tại Freetuts.net";  } |

**Kết quả:**Thực hiện kiểm tra hai số x = 0 và x = 10 trong dãy số: 3, 5, -2, 0, 9, 6, 8.



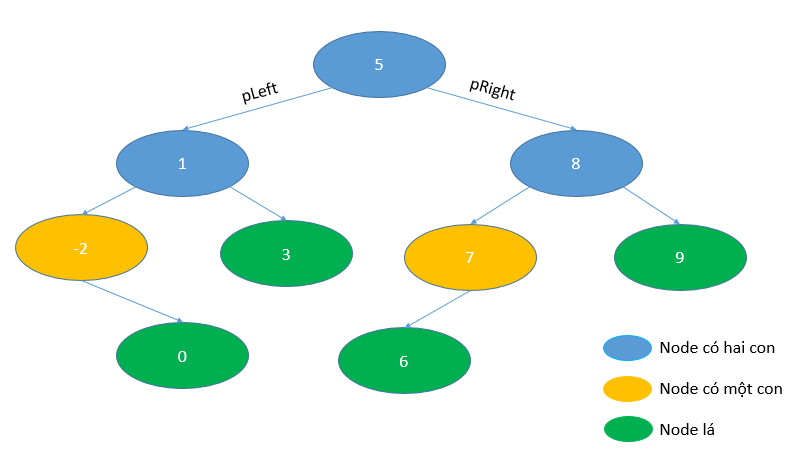
**3. Kết luận**

Như vậy là chúng ta đã tìm hiểu xong về cách tìm kiếm phần tử trong cây nhị phân tìm kiếm. Ở ví dụ trên mình thực hiện tìm kiếm số nguyên x trong cây số nguyên, các bạn có thể tự thực hành bằng cách tìm số nguyên tố trong cây số nguyên để có thể luyện tập. Ở bài tiếp theo mình sẽ tiếp tục thực hiện các thao tác thường gặp trong cây nhị phân, các bạn hay chú ý theo dõi nhé !!

**Xuất Node con và lá trong cây nhị phân tìm kiếm**

Trong hướng dẫn này mình sẽ giới thiệu các bạn cách xuất các Node con và Node lá trong cây nhị phân tìm kiếm. Đây là một bài toán thường gặp khi thực hiện thao tác với cây nhị phân tìm kiếm.



****

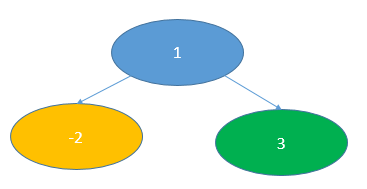
Chúng ta sẽ cùng nhau tìm hiểu về Node con và Node lá là gì? Sau đó thực hiện xuất số Node con và Node lá trong cây nhị phân tìm kiếm thông qua ví dụ.

**Mục lục**

* [1. Xuất các Node có hai con trong cây nhị phân tìm kiếm](https://freetuts.net/xuat-node-con-va-la-trong-cay-nhi-phan-tim-kiem-3026.html#goto-h2-0)
* [2. Xuất các Node có một con trong cây nhị phân tìm kiếm](https://freetuts.net/xuat-node-con-va-la-trong-cay-nhi-phan-tim-kiem-3026.html#goto-h2-1)
* [3. Xuất các Node lá trong cây nhị phân tìm kiếm](https://freetuts.net/xuat-node-con-va-la-trong-cay-nhi-phan-tim-kiem-3026.html#goto-h2-2)
* [4. Ví dụ: Xuất các Node hai con, Node một con, Node lá trong cây nhị phân tìm kiếm](https://freetuts.net/xuat-node-con-va-la-trong-cay-nhi-phan-tim-kiem-3026.html#goto-h2-3)
* [5. Kết luận](https://freetuts.net/xuat-node-con-va-la-trong-cay-nhi-phan-tim-kiem-3026.html#goto-h2-4)

**1. Xuất các Node có hai con trong cây nhị phân tìm kiếm**

Node có hai con là Node vừa có cây con trái vừa có cây con phải. Vì vậy ta dựa vào đây để viết hàm xuất ra Node có hai con.



Cụ thể trong hình trên Node số 1 là Node có hai con, vì cây con bên trái và cây con bên phải nó đều tồn tại.

*Bài viết này được đăng tại [free tuts .net]*

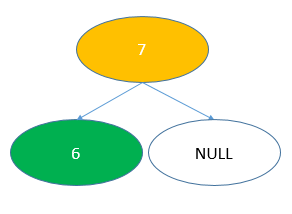
Đầu tiên ta xét điều kiện cây nhị phân tìm kiếm không rỗng (có tồn tại phần tử), khi đó ta tiếp tục xét điều kiện nếu cây con trái khác NULL (t -> pLeft != NULL) và cây con phải khác NULL (t -> pRight != NULL).

Sau khi xét điều kiện của Node có hai con ta thực hiện gọi đệ quy để tiếp tục duyệt qua cây con trái và cây con phải

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // xuất các node có 2 con  void Node\_Co\_2\_Con(TREE t)  {      if (t != NULL)      {          // xử lí          if (t->pLeft != NULL && t->pRight != NULL) // con trái và con phải có tồn tại phần tử          {              cout << t->data << "  ";          }          Node\_Co\_2\_Con(t->pLeft); // duyệt sang cây con trái của node hiện tại          Node\_Co\_2\_Con(t->pRight); // duyệt sang cây con phải của node hiện tại      }  } |

**2. Xuất các Node có một con trong cây nhị phân tìm kiếm**

Node có một con trong cây nhị phân tìm kiếm là Node chỉ tồn tại cây con trái hoặc chỉ tồn tại cây con phải. Một bên sẽ tồn tại giá trị và bên còn lại bằng NULL.



Cụ thể trong hình ta thấy Node số 7 chỉ tồn tại cây con trái là số 6, còn cây con phải bằng NULL. Một Node như vậy được xem là Node có một con.

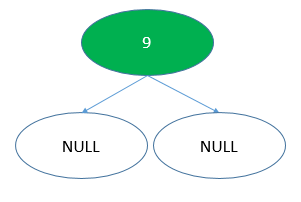
Thuật toán cho trường hợp này khá đơn giản, ta chỉ cần thay đổi điều kiện một chút từ trường hợp xuất Node có hai con là xong.

Ta xét hai điều kiện: t->pLeft != NULL && t->pRight == NULL và t->pLeft == NULL && t->pRight != NULL. Ta vẫn sẽ gọi đệ quy để tiếp tục duyệt sang cây con trái và cây con phải.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | // xuất các node có 1 con  void Node\_Co\_1\_Con(TREE t)  {      if (t != NULL)      {          // xử lí          if ((t->pLeft != NULL && t->pRight == NULL) || (t->pLeft == NULL && t->pRight != NULL))          {              cout << t->data << "  ";          }          Node\_Co\_1\_Con(t->pLeft); // duyệt sang cây con trái của node hiện tại          Node\_Co\_1\_Con(t->pRight); // duyệt sang cây con phải của node hiện tại      }  } |

**3. Xuất các Node lá trong cây nhị phân tìm kiếm**

Node lá trong cây nhị phân tìm kiếm là Node không tồn tại cây con trái và cây con phải. Cả cây con trái và cây con phải của nó đều bằng giá trị NULL.



Thuật toán cho trường hợp này ngược lại với trường hợp Node có hai con, ta chỉ cần cho t -> pLeft == NULL và t -> pRight == NULL. Rồi gọi đệ quy để duyệt qua cây con trái và cây con phải là xong.

// xuất các node lá

void Node\_La(TREE t)

{

if (t != NULL)

{

// xử lí

if (t->pLeft == NULL && t->pRight == NULL)

{

cout << t->data << " ";

}

Node\_La(t->pLeft); // duyệt sang cây con trái của node hiện tại

Node\_La(t->pRight); // duyệt sang cây con phải của node hiện tại

}

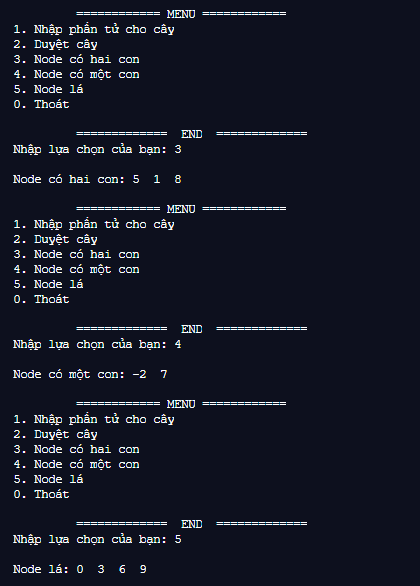
}

**4. Ví dụ: Xuất các Node hai con, Node một con, Node lá trong cây nhị phân tìm kiếm**

Trong ví dụ này mình sẽ thực hiện một Menu gồm các thao tác xuất Node hai con, Node một con và Node lá. Các bạn hãy tham khảo code dưới đây nhé.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163 | #include<iostream>  using namespace std;  // khai báo cấu trúc 1 cái node trong cây nhị phân tìm kiếm  struct node  {  **int** data; // dữ liệu chứa trong 1 cái node      struct node \*pLeft; // con trỏ liên kết với cái node bên trái <=> cây con trái      struct node \*pRight; // con trỏ liên kết với cái node bên phải <=> cây con phải  };  typedef struct node NODE;  typedef NODE\* TREE;    // hàm khởi tạo cây nhị phân tìm kiếm  void KhoiTaoCay(TREE &t)  {      t = NULL;  }    // hàm thêm 1 cái phần tử vào cây  void ThemNodeVaoCay(TREE &t, **int** x)  {      // nếu cây rỗng      if (t == NULL)      {          NODE \*p = new NODE;          p->data = x;          p->pLeft = NULL;          p->pRight = NULL;          t = p;      }      else      {          // nếu phần tử thêm vào mà nhỏ hơn nút gốc thì sẽ duyệt qua bên trái          if (x < t->data)          {              ThemNodeVaoCay(t->pLeft, x);          }          else if (x > t->data)          {              ThemNodeVaoCay(t->pRight, x);          }      }  }    // hàm xuất phần tử trong cây nhị phân tìm kiếm  void NLR(TREE t)  {      if (t != NULL)      {          // xử lí          cout << t->data << "  ";          NLR(t->pLeft);          NLR(t->pRight);      }  }    // xuất các node có 2 con  void Node\_Co\_2\_Con(TREE t)  {      if (t != NULL)      {          // xử lí          if (t->pLeft != NULL && t->pRight != NULL) // con trái và con phải có tồn tại phần tử          {              cout << t->data << "  ";          }          Node\_Co\_2\_Con(t->pLeft); // duyệt sang cây con trái của node hiện tại          Node\_Co\_2\_Con(t->pRight); // duyệt sang cây con phải của node hiện tại      }  }  // xuất các node có 1 con  void Node\_Co\_1\_Con(TREE t)  {      if (t != NULL)      {          // xử lí          if ((t->pLeft != NULL && t->pRight == NULL) || (t->pLeft == NULL && t->pRight != NULL))          {              cout << t->data << "  ";          }          Node\_Co\_1\_Con(t->pLeft); // duyệt sang cây con trái của node hiện tại          Node\_Co\_1\_Con(t->pRight); // duyệt sang cây con phải của node hiện tại      }  }    // xuất các node lá  void Node\_La(TREE t)  {      if (t != NULL)      {          // xử lí          if (t->pLeft == NULL && t->pRight == NULL)          {              cout << t->data << "  ";          }          Node\_La(t->pLeft); // duyệt sang cây con trái của node hiện tại          Node\_La(t->pRight); // duyệt sang cây con phải của node hiện tại      }  }    // hàm nhập cây nhị phân tìm kiếm  void Menu(TREE &t)  {  **int** luachon;      while(true)      {          cout << "\n\n\t\t ============ MENU ============";          cout << "\n1. Nhập phần tử cho cây";          cout << "\n2. Duyệt cây";          cout << "\n3. Node có hai con";          cout << "\n4. Node có một con";          cout << "\n5. Node lá";          cout << "\n0. Thoát";          cout << "\n\n\t\t =============  END  =============";            cout << "\nNhập lựa chọn của bạn: ";          cin >> luachon;            if (luachon == 1)          {  **int** x;              cout << "\nNhập giá trị: ";              cin >> x;              ThemNodeVaoCay(t, x);          }          else if (luachon == 2)          {              cout << "\nCây nhị phân tìm kiếm\n";              NLR(t);          }            else if (luachon == 3)          {              cout << "\nNode có hai con: ";              Node\_Co\_2\_Con(t);          }          else if (luachon == 4)          {              cout << "\nNode có một con: ";              Node\_Co\_1\_Con(t);          }          else if (luachon == 5)          {              cout << "\nNode lá: ";              Node\_La(t);          }          else          {              break;          }      }  }      **int** main()  {      TREE t;      KhoiTaoCay(t);      Menu(t);        cout<<"\n---------------------------------\n";    cout<<"Chương trình này được đăng tại Freetuts.net";  } |

**Kết quả:**Mình sẽ sử dụng dữ liệu như hình ở đầu bài viết, bao gồm các số: 5, 1, -2, 0, 8, 7, 9, 3, 6.



**5. Kết luận**

Như vậy là chúng ta đã tìm hiểu xong Node hai con, Node một con, Node lá là gì? và cách xuất các Node lá trong cây nhị phân tìm kiếm. Các bạn hãy luyện tập thật nhiều để có thể nhớ được lâu nhé. Ở bài tiếp theo mình sẽ thực hiện tìm giá trị MIN và giá trị MAX trong cây nhị phân tìm kiếm, các bạn hãy chú ý theo dõi nhé !!!

**Tìm Node MAX và MIN trong cây nhị phân tìm kiếm**

Trong hướng dẫn này mình sẽ giới thiệu các bạn cách tìm giá trị MAX và MIN trong cây nhị phân tìm kiếm. Đối với cây nhị phân tìm kiếm thì việc tìm ra giá trị lớn nhất và nhỏ nhất trong cây rất đơn giản.



Chúng ta sẽ thực hiện một vài cách tìm ra giá trị MAX và MIN để xem cách nào tối ưu nhất. Để làm được các bạn cần có kiến thức cơ bản về C++ và các nẵm vững cấu trúc dữ liệu cây.

**Mục lục**

* [1. Tìm giá trị MAX trong cây nhị phân tìm kiếm](https://freetuts.net/tim-node-max-va-min-trong-cay-nhi-phan-tim-kiem-3033.html#goto-h2-0)
  + [Cách 1:](https://freetuts.net/tim-node-max-va-min-trong-cay-nhi-phan-tim-kiem-3033.html#goto-h3-0)
  + [Cách 2](https://freetuts.net/tim-node-max-va-min-trong-cay-nhi-phan-tim-kiem-3033.html#goto-h3-1)
* [2. Tìm giá trị MIN trong cây nhị phần tìm kiếm](https://freetuts.net/tim-node-max-va-min-trong-cay-nhi-phan-tim-kiem-3033.html#goto-h2-1)
* [3. Ví dụ: Tìm giá trị MAX, MIN trong cây nhị phân tìm kiếm](https://freetuts.net/tim-node-max-va-min-trong-cay-nhi-phan-tim-kiem-3033.html#goto-h2-2)
* [4. Kết luận](https://freetuts.net/tim-node-max-va-min-trong-cay-nhi-phan-tim-kiem-3033.html#goto-h2-3)

**1. Tìm giá trị MAX trong cây nhị phân tìm kiếm**

Trong phần này mình sẽ thực hiện hai cách để tìm ra giá trị MAX trong cây nhị phân tìm kiếm. Các bạn hãy theo dõi và xem thử cách nào tối ưu nhất nhé.

**Cách 1:**

Trong cách này mình sẽ sử dụng biến toàn cục **INT\_MIN**để giá cho biến MAX, sau đó so sánh với các Node trong cây để tìm ra giá trị lớn nhất, cụ thể như sau:

*Bài viết này được đăng tại [free tuts .net]*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | //------cách 1------------  **int** MAX = INT\_MIN; // gán cho biến MAX là giá trị nhỏ nhất của kiểu integer  //// hàm tìm phần tử lớn nhất trong cây  void TimMax(TREE t)  {      if (t != NULL)      {          // xử lí          if (MAX < t->data)          {              MAX = t->data; // cập nhật lại giá trị cho biến MAX          }          TimMax(t->pLeft);          TimMax(t->pRight);      }  } |

Ta sẽ sử dụng giá trị nhỏ nhất của kiểu integer và gán cho biến MAX, sau đó ta xét điều kiện nếu cây tồn tại phần tử thì ta thực hiện so sánh MAX với t -> data. Nếu MAX < t -> data thì ta cập nhật lại biến MAX, ngược lại ta sẽ đệ quy để duyệt sang trái và sang phải.

Tuy nhiên đây là một cách không tối ưu, vì chúng ta cần phải duyệt từ đầu đến cuối cây. Vì vậy các bạn hãy xem cách hai mình thực hiện, sẽ nhanh hơn rất nhiều nhé.

**Cách 2**

Như các bạn đã học thì cây nhị phân tìm kiếm có một cấu trúc lưu trữ dữ liệu rất đặc biệt, vì vậy ta sẽ dựa vào đó để thực hiện tìm ra giá trị MAX một cách nhanh nhất có thể.

Cây nhị phân tìm kiếm luôn luôn lưu giá trị nhỏ hơn ở cây con trái và giá trị lớn hơn ở cây con phải, vì vậy thay vì ta duyệt hết cả cây, ta chỉ cần duyệt sang cây con phải là có thể tìm ra giá trị MAX được rồi.

Các bạn hãy chú ý rằng giá trị ngoài cùng của cây con trái luôn luôn là giá trị nhỏ nhất và giá trị ngoài cùng của cây con phải luôn luôn là giá trị lớn nhất. Dựa vào điều này ta cho hàm duyệt sang bên cây con phải đến giá trị ngoài cùng nhất, khi đó giá trị MAX chính là t -> data của Node ngoài cùng.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **int** TimMax(TREE t)  {      // CÁCH TỐI ƯU NHẤT      // nếu node cha mà không tồn cây con phải - thì node cha này chính là node ngoài cùng nhất của cây con phải - cũng chính là node có giá trị lớn nhất      if (t->pRight == NULL)      {          return t->data;      }      return TimMax(t->pRight); // duyệt đến node cuối cùng nhất bên cây con phải ==> để tìm giá trị lớn nhất  } |

**2. Tìm giá trị MIN trong cây nhị phần tìm kiếm**

Tương tự như tìm giá trị MAX trong cây nhị phân tìm kiếm, việc tìm giá trị MIN chỉ đơn giản là ta duyệt sang cây con trái, vì cây con trái luôn chứa giá trị nhỏ hơn Node cha. Như mình đã nói ở trên thì Node ngoài cùng bên trái luôn luôn là Node nhỏ nhất, vậy nên chỉ cần duyệt sang trái đến Node cuối cùng nhất thì đó chính là giá trị MIN.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //hàm tìm giá trị MIN  **int** TimMin(TREE t)  {      // CÁCH TỐI ƯU NHẤT      // nếu node cha mà không tồn cây con trái - thì node cha này chính là node ngoài cùng nhất của cây con trái - cũng chính là node có giá trị nhỏ nhất      if (t->pLeft == NULL)      {          return t->data;      }      return TimMin(t->pLeft); // duyệt đến node cuối cùng nhất bên cây con trái ==> để tìm giá trị nhỏ nhất  } |

**3. Ví dụ: Tìm giá trị MAX, MIN trong cây nhị phân tìm kiếm**

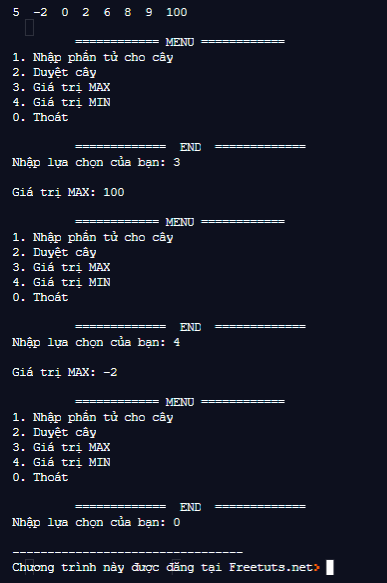
Trong ví dụ này mình sẽ thực hiện nhập vào một dãy các số nguyên sau đó in ra số có giá trị lớn nhất và nhỏ nhất. Để làm được bài toán này ta cần có các hàm sau:

* Cấu trúc dữ liệu của cây và Node.
* Hàm thêm Node vào cây.
* Hàm duyệt cây (chỉ cần một trong 6 hàm duyệt).
* Hàm tìm MAX.
* Hàm tìm MIN.
* Hàm menu để hiển thị.

**Full code**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148 | #include<iostream>  using namespace std;  // khai báo cấu trúc 1 cái node trong cây nhị phân tìm kiếm  struct node  {  **int** data; // dữ liệu chứa trong 1 cái node      struct node \*pLeft; // con trỏ liên kết với cái node bên trái <=> cây con trái      struct node \*pRight; // con trỏ liên kết với cái node bên phải <=> cây con phải  };  typedef struct node NODE;  typedef NODE\* TREE;    // hàm khởi tạo cây nhị phân tìm kiếm  void KhoiTaoCay(TREE &t)  {      t = NULL;  }    // hàm thêm 1 cái phần tử vào cây  void ThemNodeVaoCay(TREE &t, **int** x)  {      // nếu cây rỗng      if (t == NULL)      {          NODE \*p = new NODE;          p->data = x;          p->pLeft = NULL;          p->pRight = NULL;          t = p;      }      else      {          // nếu phần tử thêm vào mà nhỏ hơn nút gốc thì sẽ duyệt qua bên trái          if (x < t->data)          {              ThemNodeVaoCay(t->pLeft, x);          }          else if (x > t->data)          {              ThemNodeVaoCay(t->pRight, x);          }      }  }    // hàm xuất phần tử trong cây nhị phân tìm kiếm  void NLR(TREE t)  {      if (t != NULL)      {          // xử lí          cout << t->data << "  ";          NLR(t->pLeft);          NLR(t->pRight);      }  }    //Hàm tìm giá trị MAX  //------cách 1------------  // int MAX = INT\_MIN; // gán cho biến MAX là giá trị nhỏ nhất của kiểu integer  // //// hàm tìm phần tử lớn nhất trong cây  // void TimMax(TREE t)  // {  //  if (t != NULL)  //  {  //      // xử lí  //      if (MAX < t->data)  //      {  //          MAX = t->data; // cập nhật lại giá trị cho biến MAX  //      }  //      TimMax(t->pLeft);  //      TimMax(t->pRight);  //  }  // }  **int** TimMax(TREE t)  {      // CÁCH TỐI ƯU NHẤT      // nếu node cha mà không tồn cây con phải - thì node cha này chính là node ngoài cùng nhất của cây con phải - cũng chính là node có giá trị lớn nhất      if (t->pRight == NULL)      {          return t->data;      }      return TimMax(t->pRight); // duyệt đến node cuối cùng nhất bên cây con phải ==> để tìm giá trị lớn nhất  }  //hàm tìm giá trị MIN  **int** TimMin(TREE t)  {      // CÁCH TỐI ƯU NHẤT      // nếu node cha mà không tồn cây con trái - thì node cha này chính là node ngoài cùng nhất của cây con trái - cũng chính là node có giá trị nhỏ nhất      if (t->pLeft == NULL)      {          return t->data;      }      return TimMin(t->pLeft); // duyệt đến node cuối cùng nhất bên cây con trái ==> để tìm giá trị nhỏ nhất  }    // hàm nhập cây nhị phân tìm kiếm  void Menu(TREE &t)  {  **int** luachon;      while(true)      {          cout << "\n\n\t\t ============ MENU ============";          cout << "\n1. Nhập phần tử cho cây";          cout << "\n2. Duyệt cây";          cout << "\n3. Giá trị MAX";          cout << "\n4. Giá trị MIN";          cout << "\n0. Thoát";          cout << "\n\n\t\t =============  END  =============";            cout << "\nNhập lựa chọn của bạn: ";          cin >> luachon;            if (luachon == 1)          {  **int** x;              cout << "\nNhập giá trị: ";              cin >> x;              ThemNodeVaoCay(t, x);          }          else if (luachon == 2)          {              cout << "\nCây nhị phân tìm kiếm\n";              NLR(t);          }          else if (luachon == 3)          {              cout << "\nGiá trị MAX: "<<TimMax(t);          }          else if (luachon == 4)          {              cout << "\nGiá trị MAX: "<<TimMin(t);          }          else          {              break;          }      }  }    **int** main()  {      TREE t;      KhoiTaoCay(t);      Menu(t);        cout<<"\n---------------------------------\n";    cout<<"Chương trình này được đăng tại Freetuts.net";  } |

**Kết quả:**



**4. Kết luận**

Như vậy là chúng ta tìm hiểu xong cách tìm giá trị MAX và MIN trong cây nhị phân tìm kiếm. Đây là một trong các thao tác đặc biệt trong cây nhị phân tìm kiếm, vì nó sinh ra để làm điều này. Trên đây là một số cách mà mình thực hiện, các bạn có thể thực hiện theo các cách khác nhau để tìm ta cách tối ưu nhất. Ở bài tiếp theo mình sẽ hướng dẫn các bạn cách xóa Node khỏi cây nhị phân tìm kiếm, các bạn nhớ chú ý theo dõi nhé !!!

**Xóa Node khỏi cây nhị phân tìm kiếm**

Trong hướng dẫn này mình sẽ giới thiệu các bạn cách xóa một Node bất kì khỏi cây nhị phân tìm kiếm. Đây là một thao tác rất quan trong và đòi hỏi các bạn phải nắm vững kiến thức về C++ cũng như cấu trúc dữ liệu cây.



Chúng ta sẽ cùng nhau thực hiện xóa Node có 1 con, Node có 2 con và Node lá trong cây nhị phân tìm kiếm.

**Mục lục**

* [1. Xoá Node có một con và Node lá](https://freetuts.net/xoa-node-khoi-cay-nhi-phan-tim-kiem-3034.html#goto-h2-0)
* [2. Xóa Node có hai con](https://freetuts.net/xoa-node-khoi-cay-nhi-phan-tim-kiem-3034.html#goto-h2-1)
* [3. Ví dụ xóa Node trong cây nhị phân tìm kiếm](https://freetuts.net/xoa-node-khoi-cay-nhi-phan-tim-kiem-3034.html#goto-h2-2)

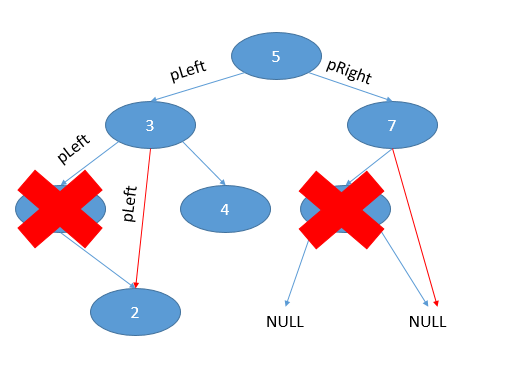
**1. Xoá Node có một con và Node lá**

Việc xóa Node có một con và Node lá được thực hiện khá đơn giản, cụ thể như sau:

Việc đầu tiên ta cần kiểm tra xem cây có tồn tại phần tử hay không, nếu không tồn tại ta return rồi kết thúc hàm, ngược lại nếu tồn tại phần tử ta bắt đầu duyệt sang cây con trái và cây con phải để tìm phần tử cần xóa.

*Bài viết này được đăng tại [free tuts .net]*

Nếu như data nhỏ hơn Node gốc thì ta duyệt sang nhánh trái của cây và nếu data lớn hơn Node gốc thì ta duyệt qua nhánh phải của cây.



Sau khi tìm được giá trị cần xóa, khi này ta cần cập nhật lại mối liên kết trước khi bắt đầu tiến hành xóa Node. Ta sẽ tạo một Node \*X = t (X là Node thế mạng, lát nữa ta sẽ xóa nó).

Trong trường hợp Node có một con sẽ có hai trường hợp khác đó chính là Node có cây con trái và Node cây con phải.

* Nếu Node có một cây con trái thì ta thực hiện duyệt sang trái để cập nhật lại mối liên kết giữa Node cha và Node con của Node cần xóa
* Nếu Node có một cây con phải thì ta thực hiện duyệt sang phải để cập nhật lại mối liên kết giữa Node cha và Node con của Node cần xóa.

Sau khi cập nhật xong mối liên kết giữa Node cần xóa với các Node con và Node cha, thì ta sẽ thực hiện xóa Node đó.

**\*Lưu ý:**Trong trường hợp xóa Node có một con, hệ quả của nó có thể dùng để xóa Node lá luôn, bởi vì Node lá là Node không tồn tại cây con trái và cây con phải. Vì vậy khi cập nhật lại mối liên kết thì cây con trái hay cây con phải đều bằng NULL.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | void XoaNode(TREE &t, **int** data) // data chính là giá trị của cái node cần xóa  {      // nếu như cây đang rỗng      if (t == NULL)      {          return; // kết thúc hàm      }      else      {          // nếu như data nhỏ hơn node gốc          if (data < t->data)          {              XoaNode(t->pLeft, data); // duyệt sang nhánh trái của cây          }          else if (data > t->data)          {              XoaNode(t->pRight, data); // duyệt sang nhánh phải của cây          }          else // data == t->data - đã tìm ra cái node cần xóa          {              NODE \*X = t; // node X là node thế mạng - tí nữa chúng ta sẽ xóa nó              // nếu như nhánh trái bằng NULL <=> đây là cây có 1 con chính là con phải              if (t->pLeft == NULL)              {                  // duyệt sang phải của cái node cần xóa để cập nhật mối liên kết giữa node                  // cha của node cần xóa với node con của node cần xóa                  t = t->pRight;              }              else if (t->pRight == NULL)              {                  // duyệt sang trái của cái node cần xóa để cập nhật mối liên kết giữa node                  // cha của node cần xóa với node con của node cần xóa                  t = t->pLeft;              }              delete X; // xóa node cần xóa          }      }  } |

**2. Xóa Node có hai con**

Trường hợp xóa Node có hai con về cơ bản thì phức tạp hơn cách xóa Node có một con và Node lá, nhưng nếu như các bạn nắm được bản chất của nó thì cũng sẽ rất đơn giản.

Trước khi xóa một Node có hai con ta cần chiều chỉnh lại mối liên kết giữa Node cần xóa với Node cha và Node con của nó. Để làm được điều này ta cần thực hiện như sau:

* Đầu tiên ta cần tìm ta Node thế mạng cho Node cần xóa, vì khi chúng ta xóa Node X đi thì ta cần bỏ vào đó một Node khác nhưng vẫn thõa mãn cấu trúc dữ liệu cây nhị phân tìm kiếm. Để có được Node thế mạng ta cần tạo một hàm NodeTheMang() để tìm ra Node có thể thay thế được Node cần xóa. Để tìm được Node này ta sẽ có hai cách: Tìm Node phải nhất bên cây con trái và tìm Node trái nhất của cây con phải.
* Sau khi tìm được Node thay thế, ta chỉ cần cập nhật lại data của Node cần xóa chính là data của Node thế mạng. Cũng như cập nhật lại mối liên kết cho Node cha của Node thế mạng và Node con của Node thế mạng.
* Và ở hàm XoaNode() ngoài hai trường hợp là Node cần xóa là Node có một con và Node lá ra, ta sẽ thêm một trường hợp là Node có hai con rồi gọi đệ quy duyệt sang bên phải (mình đang làm cách tìm Node trái nhất của cây con phải).

Dưới đây là hàm xóa Node bất kì khỏi cây nhị phân tìm kiếm mà mình đã viết, trong đó mình có chú thích rất rõ ràng, các bạn có thể đọc tham khảo nhé.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | // hàm tìm node thế mạng  void NodeTheMang(TREE &X, TREE &Y) // NODE Y là node thế mạng cho node cần xóa - node này sẽ đảm nhận nhiệm vụ tìm ra node trái nhất(TÌM NODE TRÁI NHẤT CÂY CON PHẢI) hoặc phải nhất(TÌM NODE PHẢI NHẤT CỦA CÂY CON TRÁI)  {      // CÁCH 1: TÌM NODE TRÁI NHẤT CỦA CÂY CON PHẢI      // duyệt sang bên trái nhất      if (Y->pLeft != NULL)      {          NodeTheMang(X, Y->pLeft);// tìm ra node trái nhất ?      }      else // tìm ra được node trái nhất rồi nek..      {          X->data = Y->data; // cập nhật cái data của node cần xóa chính là data của node thế mạng          X = Y; // cho node X(là node mà chúng ta sẽ đi xóa sau này) trỏ đến node thế mạng ==> ra khỏi hàm thì ta sẽ xóa node X          Y = Y->pRight; // bản chất chỗ này chính là cập nhật lại mối liên kết cho node cha của node thế mạng(mà chúng ta sẽ xóa) với node con của node thế mạng      }        /\* CÁCH 2: TÌM NODE PHẢI NHẤT CỦA CÂY CON TRÁI       duyệt sang bên phải       if (Y->pRight != NULL)      {          DiTimNodeTheMang(X, Y->pRight);// tìm ra node phải nhất ?      }      else // tìm ra được node phải nhất rồi nek..      {          X->data = Y->data; // cập nhật cái data của node cần xóa chính là data của node thế mạng          X = Y; // cho node X(là node mà chúng ta sẽ đi xóa sau này) trỏ đến node thế mạng ==> ra khỏi hàm thì ta sẽ xóa node X          Y = Y->pLeft; // bản chất chỗ này chính là cập nhật lại mối liên kết cho node cha của node thế mạng(mà chúng ta sẽ xóa) với node con của node thế mạng      } \*/  }    // hàm xóa 1 cái node bất kì trong cây nhị phân tìm kiếm  void XoaNode(TREE &t, **int** data) // data chính là giá trị của cái node cần xóa  {      // nếu như cây đang rỗng      if (t == NULL)      {          return; // kết thúc hàm      }      else      {          // nếu như data nhỏ hơn node gốc          if (data < t->data)          {              XoaNode(t->pLeft, data); // duyệt sang nhánh trái của cây          }          else if (data > t->data)          {              XoaNode(t->pRight, data); // duyệt sang nhánh phải của cây          }          else // data == t->data - đã tìm ra cái node cần xóa          {              NODE \*X = t; // node X là node thế mạng - tí nữa chúng ta sẽ xóa nó              // nếu như nhánh trái bằng NULL <=> đây là cây có 1 con chính là con phải              if (t->pLeft == NULL)              {                  // duyệt sang phải của cái node cần xóa để cập nhật mối liên kết giữa node                  // cha của node cần xóa với node con của node cần xóa                  t = t->pRight;              }              else if (t->pRight == NULL)              {                  // duyệt sang trái của cái node cần xóa để cập nhật mối liên kết giữa node                  // cha của node cần xóa với node con của node cần xóa                  t = t->pLeft;              }              else // node cần xóa là node có 2 con              {                  // CÁCH 1: Tìm node trái nhất của cây con phải(cây con phải của cái node cần xóa)                  NodeTheMang(X, t->pRight);                  // CÁCH 2: Tìm node phải nhất của cây con trái(cây con trái của cái node cần xóa)                  //DiTimNodeTheMang(X, t->pLeft);              }              delete X; // xóa node cần xóa          }      }  } |

**3. Ví dụ xóa Node trong cây nhị phân tìm kiếm**

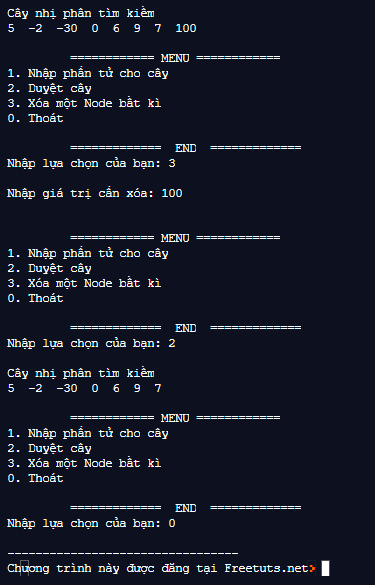
Trong ví dụ này mình sẽ thực hiện xóa một Node bất kì trong cây nhị phân tìm kiếm. Để làm được bài toán này ta cần có các hàm khác sau:

* Cấu trúc dữ liệu cây và cấu trúc Node.
* Hàm thêm Node vào cây.
* Hàm duyệt cây.
* Hàm xóa Node khỏi cây.

**Full code**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186 | #include<iostream>  using namespace std;  // khai báo cấu trúc 1 cái node trong cây nhị phân tìm kiếm  struct node  {  **int** data; // dữ liệu chứa trong 1 cái node      struct node \*pLeft; // con trỏ liên kết với cái node bên trái <=> cây con trái      struct node \*pRight; // con trỏ liên kết với cái node bên phải <=> cây con phải  };  typedef struct node NODE;  typedef NODE\* TREE;    // hàm khởi tạo cây nhị phân tìm kiếm  void KhoiTaoCay(TREE &t)  {      t = NULL;  }    // hàm thêm 1 cái phần tử vào cây  void ThemNodeVaoCay(TREE &t, **int** x)  {      // nếu cây rỗng      if (t == NULL)      {          NODE \*p = new NODE;          p->data = x;          p->pLeft = NULL;          p->pRight = NULL;          t = p;      }      else      {          // nếu phần tử thêm vào mà nhỏ hơn nút gốc thì sẽ duyệt qua bên trái          if (x < t->data)          {              ThemNodeVaoCay(t->pLeft, x);          }          else if (x > t->data)          {              ThemNodeVaoCay(t->pRight, x);          }      }  }    // hàm xuất phần tử trong cây nhị phân tìm kiếm  void NLR(TREE t)  {      if (t != NULL)      {          // xử lí          cout << t->data << "  ";          NLR(t->pLeft);          NLR(t->pRight);      }  }      // hàm tìm node thế mạng  void NodeTheMang(TREE &X, TREE &Y) // NODE Y là node thế mạng cho node cần xóa - node này sẽ đảm nhận nhiệm vụ tìm ra node trái nhất(TÌM NODE TRÁI NHẤT CÂY CON PHẢI) hoặc phải nhất(TÌM NODE PHẢI NHẤT CỦA CÂY CON TRÁI)  {      // CÁCH 1: TÌM NODE TRÁI NHẤT CỦA CÂY CON PHẢI      // duyệt sang bên trái nhất      if (Y->pLeft != NULL)      {          NodeTheMang(X, Y->pLeft);// tìm ra node trái nhất ?      }      else // tìm ra được node trái nhất rồi nek..      {          X->data = Y->data; // cập nhật cái data của node cần xóa chính là data của node thế mạng          X = Y; // cho node X(là node mà chúng ta sẽ đi xóa sau này) trỏ đến node thế mạng ==> ra khỏi hàm thì ta sẽ xóa node X          Y = Y->pRight; // bản chất chỗ này chính là cập nhật lại mối liên kết cho node cha của node thế mạng(mà chúng ta sẽ xóa) với node con của node thế mạng      }        /\* CÁCH 2: TÌM NODE PHẢI NHẤT CỦA CÂY CON TRÁI       duyệt sang bên phải       if (Y->pRight != NULL)      {          DiTimNodeTheMang(X, Y->pRight);// tìm ra node phải nhất ?      }      else // tìm ra được node phải nhất rồi nek..      {          X->data = Y->data; // cập nhật cái data của node cần xóa chính là data của node thế mạng          X = Y; // cho node X(là node mà chúng ta sẽ đi xóa sau này) trỏ đến node thế mạng ==> ra khỏi hàm thì ta sẽ xóa node X          Y = Y->pLeft; // bản chất chỗ này chính là cập nhật lại mối liên kết cho node cha của node thế mạng(mà chúng ta sẽ xóa) với node con của node thế mạng      } \*/  }    // hàm xóa 1 cái node bất kì trong cây nhị phân tìm kiếm  void XoaNode(TREE &t, **int** data) // data chính là giá trị của cái node cần xóa  {      // nếu như cây đang rỗng      if (t == NULL)      {          return; // kết thúc hàm      }      else      {          // nếu như data nhỏ hơn node gốc          if (data < t->data)          {              XoaNode(t->pLeft, data); // duyệt sang nhánh trái của cây          }          else if (data > t->data)          {              XoaNode(t->pRight, data); // duyệt sang nhánh phải của cây          }          else // data == t->data - đã tìm ra cái node cần xóa          {              NODE \*X = t; // node X là node thế mạng - tí nữa chúng ta sẽ xóa nó              // nếu như nhánh trái bằng NULL <=> đây là cây có 1 con chính là con phải              if (t->pLeft == NULL)              {                  // duyệt sang phải của cái node cần xóa để cập nhật mối liên kết giữa node                  // cha của node cần xóa với node con của node cần xóa                  t = t->pRight;              }              else if (t->pRight == NULL)              {                  // duyệt sang trái của cái node cần xóa để cập nhật mối liên kết giữa node                  // cha của node cần xóa với node con của node cần xóa                  t = t->pLeft;              }              else // node cần xóa là node có 2 con              {                  // CÁCH 1: Tìm node trái nhất của cây con phải(cây con phải của cái node cần xóa)                  NodeTheMang(X, t->pRight);                  // CÁCH 2: Tìm node phải nhất của cây con trái(cây con trái của cái node cần xóa)                  //DiTimNodeTheMang(X, t->pLeft);              }              delete X; // xóa node cần xóa          }      }  }    // hàm nhập cây nhị phân tìm kiếm  void Menu(TREE &t)  {  **int** luachon;      while(true)      {          cout << "\n\n\t\t ============ MENU ============";          cout << "\n1. Nhập phần tử cho cây";          cout << "\n2. Duyệt cây";          cout << "\n3. Xóa một Node bất kì";          cout << "\n0. Thoát";          cout << "\n\n\t\t =============  END  =============";            cout << "\nNhập lựa chọn của bạn: ";          cin >> luachon;            if (luachon == 1)          {  **int** x;              cout << "\nNhập giá trị: ";              cin >> x;              ThemNodeVaoCay(t, x);          }          else if (luachon == 2)          {              cout << "\nCây nhị phân tìm kiếm\n";              NLR(t);          }          else if (luachon == 3)              {  **int** x;                  cout << "\nNhập giá trị cần xóa: ";                  cin >> x;                  XoaNode(t, x);              }          else          {              break;          }      }  }      **int** main()  {      TREE t;      KhoiTaoCay(t);      Menu(t);        cout<<"\n---------------------------------\n";    cout<<"Chương trình này được đăng tại Freetuts.net";  } |

**Kết quả:**



Như vậy là chúng ta đã thực hiện xong chương trình xóa một Node bất kì khỏi cây nhị phân tìm kiếm, các bạn có thể thực hiện bài toán này theo nhiều cách khác nhau. Chúc các bạn thực hiện thành công!!!

**Cây đỏ đen là gì? Cấu trúc của Red-Black Tree**

Trong hướng dẫn này mình sẽ giới thiệu các bạn một cấu trúc dữ liệu dạng cây nữa đó chính là cây đỏ đen. Đây là một dạng đặc biệt của cây nhị phân tìm kiếm, vì vậy các bạn cần nắm vững kiến thức về cây nhị phân tìm kiếm trước khi vào bài học này nhé.



Chúng ta sẽ cùng nhau tìm hiểu về khái niệm cây đỏ đen là gì? Và cấu trúc dữ liệu của nó, cũng như các thao tác cơ bản trong cây đỏ đen.

**Mục lục**

* [1. Cây đỏ đen (Red-Black Tree) là gì?](https://freetuts.net/cay-do-den-la-gi-3124.html#goto-h2-0)
* [2. Cấu trúc dữ liệu của cây đỏ đen](https://freetuts.net/cay-do-den-la-gi-3124.html#goto-h2-1)
* [3. Các tính chất của cây đỏ đen](https://freetuts.net/cay-do-den-la-gi-3124.html#goto-h2-2)
  + [Tính chất 1](https://freetuts.net/cay-do-den-la-gi-3124.html#goto-h3-0)
  + [Tính chât 2](https://freetuts.net/cay-do-den-la-gi-3124.html#goto-h3-1)
  + [Tính chất 3](https://freetuts.net/cay-do-den-la-gi-3124.html#goto-h3-2)
* [4. Các thao tác cơ bản trong cây đỏ đen](https://freetuts.net/cay-do-den-la-gi-3124.html#goto-h2-3)
* [5. Kết luận](https://freetuts.net/cay-do-den-la-gi-3124.html#goto-h2-4)

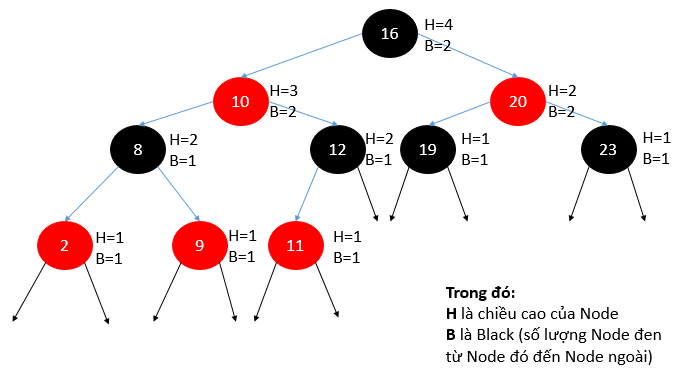
**1. Cây đỏ đen (Red-Black Tree) là gì?**

Cây đỏ đen (Red-Black Tree) là một cây nhị phân tìm kiếm (BST) tuân thủ theo các quy tắc sau:

1. Mọi Node trong cây phải là đỏ hoặc đen.
2. Node gốc là Node đen.
3. Các Node ngoài phải (NULL Node) luôn luôn đen.
4. Nếu một Node là đỏ thì những Node con của nó phải là đen (quy tắc xung đột).
5. Mọi đường dẫn từ Node gốc đến Node ngoài phải có cùng số lượng Node đen.

Một cây nhị phân tìm kiếm tuân thủ theo các quy tắc trên được gọi là cây đỏ đen, ví dụ như cây sau đây:

*Bài viết này được đăng tại [free tuts .net]*



Ta có một số khái niệm như sau:

* **Chiều cao đen (black height - hb(x))**: Là số Node đen trên đường đi từ x đến Node ngoài (không bao gồm x).
* **Hiện tượng xung đột đỏ - đỏ**: Đây là trường hợp vi phạm quy tắc số 4, khi Node cha và Node con trực tiếp cùng một màu đỏ.

**2. Cấu trúc dữ liệu của cây đỏ đen**

Từ định nghĩa ta có thể suy ra được cấu trúc dữ liệu của cây đỏ đen, cụ thể như sau:

* Thông tin lưu trữ tại Node (key).
* Địa chỉ Node gốc của cây con bên trái (\* pLeft).
* Địa chỉ Node gốc của cây con bên phải (\* pRight).
* Địa chỉ của Node cha (\* pParent).
* Thuộc tính màu của Node (color).

Dựa vào các thuộc tính trên ta có thể viết cấu trúc dữ liệu trong C++ như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | /\* khai báo thuộc tính màu cho Node \*/  enum Color {RED, BLACK};  /\* Khai báo cấu trúc Node \*/  struct Node  {  **int** data;  **bool** color;      Node \*left, \*right, \*parent;        // Constructor      Node(**int** data)      {         this->data = data;         left = right = parent = NULL;         this->color = RED;      }  }; |

**3. Các tính chất của cây đỏ đen**

Trong phần này chúng ta sẽ tìm hiểu về 3 tính chất của cây đỏ đen, cụ thể:

**Tính chất 1**

h <= 2 \* hb

**Trong đó:**

* h là chiều cao của cây.
* hb là chiều cao đen.

**Tính chât 2**

Giả sử cây đỏ đen có N Node thì khi đó:

h <= 2 \* log(N + 1)

**Tính chất 3**

Thời gian tìm kiếm O một Node:

O(log N)

**4. Các thao tác cơ bản trong cây đỏ đen**

Vì cây đỏ đen là một cây nhị phân tìm kiếm, vì vậy về cơ bản nó cũng có các thao tác như cây nhị phân tìm kiếm, cụ thể là các thao tác sau:

* Tìm kiếm và duyệt cây (giống BST).
* Thêm Node mới (insert Node).
* Xóa Node (delete Node).

Việc tìm kiếm các Node trong cây và duyệt cây hoàn toàn tương tự cây nhị phân tìm kiếm. Còn thao tác thêm Node và xóa Node thì khác so với cây nhị phân tìm kiếm, vì sau mỗi lần thêm Node hoặc xóa Node ta phải cập nhật lại thuộc tính color để không vi phạm các quy tắc của cây đỏ đen.

**5. Kết luận**

Như vậy là chúng ta đã tìm hiểu xong về cây đỏ đen là gì? và cấu trúc dữ liệu của nó ra sao. Để có thể tìm hiểu về cây đỏ đen đòi hỏi các bạn cần có kiến thức cơ bản về C/C++ và cây nhị phân tìm kiếm. Vì cây đỏ đen cũng chính là cây nhị phân tìm kiếm, vậy nên nó cũng có các quy tắc lưu trữ như cây nhị phân tìm kiếm. Ở bài tiếp theo mình sẽ hướng dẫn các bạn các thao tác cơ bản trong cây đỏ đen, hãy chú ý theo dõi nhé!!

**Thêm Node mới vào cây đỏ đen**

Trong bài này mình sẽ giới thiệu đến các bạn cách thêm một Node mới vào cây đỏ đen. Đây là một trong các thao tác cơ bản nhưng rất quan trọng của các cấu trúc dữ liệu nói chung và cây đỏ đen nói riêng.



Chúng ta sẽ cùng nhau tìm hiểu về cách thêm một Node mới như thế nào, sau đó thực hiện một ví dụ thêm Node vào cây đỏ đen.

**Mục lục**

* [1. Thêm Node mới vào cây đỏ đen](https://freetuts.net/them-node-moi-vao-cay-do-den-3139.html#goto-h2-0)
* [2. Điều chỉnh cây vi phạm quy tắc khi thêm Node](https://freetuts.net/them-node-moi-vao-cay-do-den-3139.html#goto-h2-1)
  + [Trường hợp 1](https://freetuts.net/them-node-moi-vao-cay-do-den-3139.html#goto-h3-0)
  + [Trường hợp 2](https://freetuts.net/them-node-moi-vao-cay-do-den-3139.html#goto-h3-1)
  + [Trường hợp 3](https://freetuts.net/them-node-moi-vao-cay-do-den-3139.html#goto-h3-2)
  + [Trường hợp 4](https://freetuts.net/them-node-moi-vao-cay-do-den-3139.html#goto-h3-3)
  + [Trường hợp 5](https://freetuts.net/them-node-moi-vao-cay-do-den-3139.html#goto-h3-4)
* [3. Ví dụ thêm Node mới vào cây đỏ đen](https://freetuts.net/them-node-moi-vao-cay-do-den-3139.html#goto-h2-2)
* [4. Kết luận](https://freetuts.net/them-node-moi-vao-cay-do-den-3139.html#goto-h2-3)

**1. Thêm Node mới vào cây đỏ đen**

Việc thêm một Node mới vào cây đỏ đen tương tự như cây nhị phân tìm kiếm, vì bản chất nó là một cây nhị phân tìm kiếm.

* Thực hiện thêm Node như cây nhị phân tìm kiếm.
* Node mới thêm vào luôn luôn có màu đỏ.
* Nếu xảy ra vi phạm quy tắc thì thực hiện điều chỉnh cây.

Khi chúng ta thêm một Node mới là Node đỏ vào cây đỏ đen, có thể sẽ vi phạm một số quy tắc sau:

*Bài viết này được đăng tại [free tuts .net]*

1. Node gốc là Node đen: Khi chúng ta thêm Node đỏ vào cây đỏ đen, mà Node cần thêm đó chính là Node gốc thì khi đó sẽ vi phạm quy tắc số 1 (Node gốc luôn luôn là Node đẹn).
2. Xảy ra xung đột đỏ - đỏ: Khi chúng ta thêm một Node đỏ vào cây đỏ đen, mà Node cha của nó cũng là Node đó thì khi đó sẽ xảy ra xung đột đỏ - đỏ.

Như vậy khi ta thêm một Node vào cây đỏ đen có thể làm thay đổi quy tắc của nó, vì vậy chúng ta cần điều chỉnh nó ngay sau khi thêm vào cây. Để làm được điều này, các bạn hãy xem các trường hợp có thể xảy ra ở mục tiếp theo.

**2. Điều chỉnh cây vi phạm quy tắc khi thêm Node**

Trong phần này mình sẽ đưa ra các trường hợp có thể xảy ra làm thay đổi quy tắc của cây đỏ đen, cụ thể sẽ có 5 trường hợp và mỗi trường hợp mình sẽ có một đoạn code C++.

Trước khi bắt đầu các trường hợp mình có một số ký hiệu để dễ thực hiện hơn như sau:

* N dùng để chỉ Node đang chèn vào.
* P chỉ Node cha của N.
* G chỉ Node ông của N.
* U chỉ Node chú và bác của N.

**Trường hợp 1**

Node N mới thêm vào ngay tại Node gốc. Trong trường hơp này, gán lại màu đen cho N để đảm bảo tính chất 2 (Gốc luôn luôn đen). Vì mới chỉ bổ sung một nút nên tính chất 5 được bảo đảm vì mọi đường đi chỉ có một nút.

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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 ảnh hưởng tới số nút đen trên tất cả đường đi.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | void insert\_case2(struct node \*n) {    if (n->parent->color== BLACK)      return;    else      insert\_case3(n);  } |

**Trường hợp 3**

Khi Node cha P và Node chú U có màu đỏ và Node ông G có màu đen, thì ta thực hiện đổi cả hai Node P và U thành đen và G thành đỏ (để đảm bảo quy tắc số 5). Tuy nhiên khi đổi như vậy có thể Node G vi phạm quy tắc số 2 (Node gốc luôn đen), để khắc phục ta thực hiện gọi đệ quy trên G từ trường hợp 1.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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);  } |

**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 chuyển đổi vai trò của nút mới N và nút cha P.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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) {      otate\_right(n->parent); n = n->right;    }    insert\_case5(n);  } |

**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.

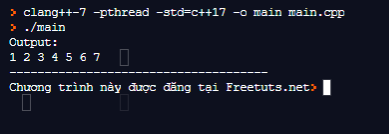
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | void insert\_case5(struct node \*n) {    n->parent->color = BLACK;    grandparent(n)->color = RED;    if (n == n->parent->left && n->parent == grandparent(n)->left) {      rotate\_right(grandparent(n)); }    else {      rotate\_left(grandparent(n));    }  } |

**3. Ví dụ thêm Node mới vào cây đỏ đen**

Trong ví dụ này mình sẽ thực hiện thêm vào một số Node là các số nguyên, sau đó thực hiện xuất nó ra màn hình.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239 | #include <iostream>  using namespace std;  /\* khai báo thuộc tính color \*/  enum Color {RED, BLACK};  /\* khai báo cấu trúc node \*/  struct Node  {  **int** data;  **bool** color;      Node \*left, \*right, \*parent;        // Constructor      Node(**int** data)      {      this->data = data;      left = right = parent = NULL;      this->color = RED;      }  };  /\* class đại diện cho cây đỏ-đen \*/  class RBTree  {  private:      Node \*root;  protected:      void rotateLeft(Node \*&, Node \*&);      void rotateRight(Node \*&, Node \*&);      void fixViolation(Node \*&, Node \*&);  public:      /\* Constructor \*/      RBTree() { root = NULL; }      void insert(const **int** &n);      void inorder();      // void levelOrder();  };    /\* Một hàm đệ quy để thực hiện việc duyệt thứ tự cấp độ \*/  void inorderHelper(Node \*root)  {      if (root == NULL)          return;        inorderHelper(root->left);      cout << root->data << " ";      inorderHelper(root->right);  }    /\* insert Node  \*/  Node\* BSTInsert(Node\* root, Node \*pt)  {      /\* Nếu cây trống thì trả về một Node mới  \*/      if (root == NULL)      return pt;        /\* Nếu không thì tiếp tục duyệt xuống dưới cây \*/      if (pt->data < root->data)      {          root->left = BSTInsert(root->left, pt);          root->left->parent = root;      }      else if (pt->data > root->data)      {          root->right = BSTInsert(root->right, pt);          root->right->parent = root;      }        /\* trả về con trỏ Node \*/      return root;  }    /\* thuật toán xoay trái \*/  void RBTree::rotateLeft(Node \*&root, Node \*&pt)  {      Node \*pt\_right = pt->right;        pt->right = pt\_right->left;        if (pt->right != NULL)          pt->right->parent = pt;        pt\_right->parent = pt->parent;        if (pt->parent == NULL)          root = pt\_right;        else if (pt == pt->parent->left)          pt->parent->left = pt\_right;        else          pt->parent->right = pt\_right;        pt\_right->left = pt;      pt->parent = pt\_right;  }  /\*thuật toán xoay phải\*/  void RBTree::rotateRight(Node \*&root, Node \*&pt)  {      Node \*pt\_left = pt->left;        pt->left = pt\_left->right;        if (pt->left != NULL)          pt->left->parent = pt;        pt\_left->parent = pt->parent;        if (pt->parent == NULL)          root = pt\_left;        else if (pt == pt->parent->left)          pt->parent->left = pt\_left;        else          pt->parent->right = pt\_left;        pt\_left->right = pt;      pt->parent = pt\_left;  }    /\* sửa lại cấu trúc khi chèn Node vào hoặc xóa node \*/  void RBTree::fixViolation(Node \*&root, Node \*&pt)  {      Node \*parent\_pt = NULL;      Node \*grand\_parent\_pt = NULL;        while ((pt != root) && (pt->color != BLACK) &&          (pt->parent->color == RED))      {            parent\_pt = pt->parent;          grand\_parent\_pt = pt->parent->parent;            /\* trường hợp A:              Node cha của pt là con trái của Node cha của pt \*/          if (parent\_pt == grand\_parent\_pt->left)          {                Node \*uncle\_pt = grand\_parent\_pt->right;                /\* trường hợp : 1              Node chú của pt là Node đỏ khi này chỉ cần đổi màu cho Node đó thành đen \*/              if (uncle\_pt != NULL && uncle\_pt->color == RED)              {                  grand\_parent\_pt->color = RED;                  parent\_pt->color = BLACK;                  uncle\_pt->color = BLACK;                  pt = grand\_parent\_pt;              }                else              {                  /\* trường hợp : 2                  pt là Node con phải của Node cha nó ta thực hiện xoay trái\*/                  if (pt == parent\_pt->right)                  {                      rotateLeft(root, parent\_pt);                      pt = parent\_pt;                      parent\_pt = pt->parent;                  }                    /\* trường hợp : 3                  pt là con trái của node cha nó ta thực hiện xoay phải \*/                  rotateRight(root, grand\_parent\_pt);                  swap(parent\_pt->color, grand\_parent\_pt->color);                  pt = parent\_pt;              }          }            /\* trường hợp : B          Node cha của pt là con phải của Node cha của pt \*/          else          {              Node \*uncle\_pt = grand\_parent\_pt->left;                /\* trường hợp : 1                  Node chú của pt là Node đỏ khi này chỉ cần đổi màu cho Node đó thành đen \*/              if ((uncle\_pt != NULL) && (uncle\_pt->color == RED))              {                  grand\_parent\_pt->color = RED;                  parent\_pt->color = BLACK;                  uncle\_pt->color = BLACK;                  pt = grand\_parent\_pt;              }              else              {                  /\* Case : 2                  pt là con trái của node cha nó ta thực hiện xoay phải \*/                  if (pt == parent\_pt->left)                  {                      rotateRight(root, parent\_pt);                      pt = parent\_pt;                      parent\_pt = pt->parent;                  }                    /\* Case : 3                  pt là Node con phải của Node cha nó ta thực hiện xoay trái \*/                  rotateLeft(root, grand\_parent\_pt);                  swap(parent\_pt->color, grand\_parent\_pt->color);                  pt = parent\_pt;              }          }      }        root->color = BLACK;  }    /\* chèn một Node mới với dữ liệu đã cho \*/  void RBTree::insert(const **int** &data)  {      Node \*pt = new Node(data);        /\* thực hiện chèn như bình thường \*/      root = BSTInsert(root, pt);        /\* sửa lại lỗi của quy tắc cây đỏ đen \*/      fixViolation(root, pt);  }    void RBTree::inorder()   { inorderHelper(root);}    /\* hàm main để thực hiện kiểm tra kết quả \*/  **int** main()  {      RBTree tree;        tree.insert(7);      tree.insert(6);      tree.insert(5);      tree.insert(4);      tree.insert(3);      tree.insert(2);      tree.insert(1);        cout << "Output: \n";      tree.inorder();      cout<<"\n-------------------------------------\n";    cout<<"Chương trình này được đăng tại Freetuts.net";  } |

**Kết quả:**Với input là 7, 6, 5, 4, 3, 2, 1.



**4. Kết luận**

Như vậy chúng ta đã tìm hiểu xong cách thêm một Node vào cây đỏ đen, cũng như các trường hợp và cách khắc phục khi bị vi phạm một số quy tắc của cây đỏ đen. Ở bài tiếp theo mình sẽ thực hiện xóa Node khỏi cây đỏ đen, các bạn chú ý theo dõi nhé !!!

**Xóa Node khỏi cây đỏ đen**

Trong hướng dẫn này mình sẽ giới thiệu các bạn cách xóa một Node khỏi cây đỏ đen. Đây là một thao tác thường gặp khi làm việc với các cấu trúc dữ liệu nói chung và cấu trúc cây đỏ đen nói riêng.



Chúng ta sẽ cùng nhau tìm hiểu về cách xóa một Node khỏi cây đỏ đen và khắc phục các trường hợp vi phạm quy tắc cây đỏ đen khi xóa Node.

**Mục lục**

* [1. Xóa Node khỏi cây đỏ đen](https://freetuts.net/xoa-node-khoi-cay-do-den-3140.html#goto-h2-0)
  + [Bước 1](https://freetuts.net/xoa-node-khoi-cay-do-den-3140.html#goto-h3-0)
  + [Bước 2](https://freetuts.net/xoa-node-khoi-cay-do-den-3140.html#goto-h3-1)
  + [Bước 3](https://freetuts.net/xoa-node-khoi-cay-do-den-3140.html#goto-h3-2)
* [2. Ví dụ xóa Node khỏi cây đỏ đen](https://freetuts.net/xoa-node-khoi-cay-do-den-3140.html#goto-h2-1)
* [3. Kết luận](https://freetuts.net/xoa-node-khoi-cay-do-den-3140.html#goto-h2-2)

**1. Xóa Node khỏi cây đỏ đen**

Trong thao tác chèn chúng ta dựa vào màu sắc của Node chú U để quyết định trường hợp phù hợp. Trong thao tác xóa, chúng ta dựa vào màu sắc của anh chị em Node N để thực quyết định trường hợp thích hợp.

Khi thực hiện thêm một Node mới vào ta sẽ vi phạm quy tắc số 4 đó là xảy ra xung đột đỏ - đỏ. Còn khi thực hiện thao tác xóa Node khỏi danh sách ta sẽ vi phạm quy tắc số 5 đó là sẽ làm thay đổi số Node đen tính từ Node gốc đến Node ngoài.

*Bài viết này được đăng tại [free tuts .net]*

Việc xóa là một quá trình khá phức tạp. Khi một Node đen bị xóa, ta cần phải thay thế vào đó một Node đen khác để đảm bảo rằng chiều cao từ Node gốc đến Node ngoài không bị thay đổi.

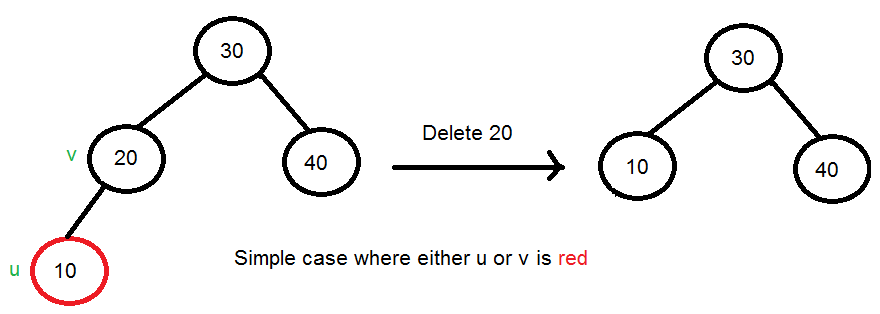
Chúng ta sẽ thực hiện lần lược các bước sau để xóa Node khỏi cây đỏ đen: v là Node bị xóa và u là Node thay thế v

**Bước 1**

Ta sẽ thực hiện viết một hàm xóa một Node tương tự như xóa nốt ở cây nhị phân tìm kiếm, sẽ có 3 trường hợp đó chính là Node lá, Node có một con và Node có hai con. ta sẽ xét từng trường hợp và xóa nó. Và lưu ý rằng Node lá sẽ có hai Node ngoài là Node đen.

**Bước 2**

Ta xét trường hợp nếu u hoặc v là màu đỏ ta thực hiện thay thế Node con là màu đen (không thay đổi chiều cao của đen của cây). Lưu ý rằng cả u và v không thể có màu đỏ vì v là cha của u và hai màu đỏ liên tiếp không được phép trong cây đỏ đen.

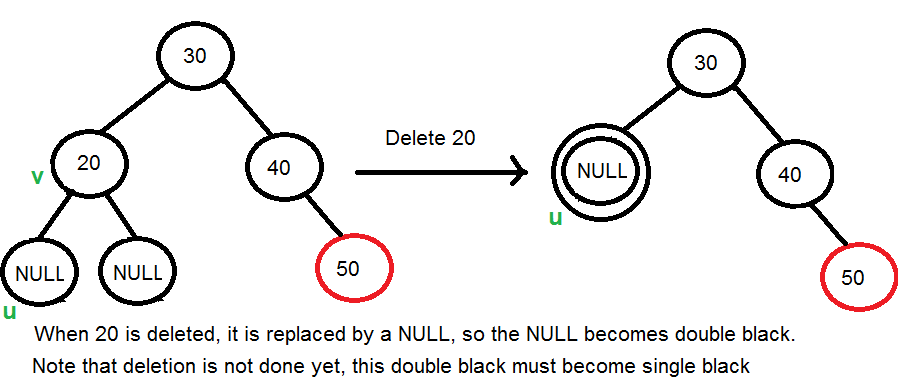


**Bước 3**

Trong trường hợp u va v đều là Node đen, ta sẽ xét tiếp các trường hợp sau:

**Trường hợp 1:**

Có hai Node u là màu đen, nhiệm vụ của ta bây giờ là giảm bớt một Node đen u đi để thay thế vào Node v cần xóa ở trên nó. Nếu v là Node lá thì hai Node u cũng là Node đen và có giá trị NULL.



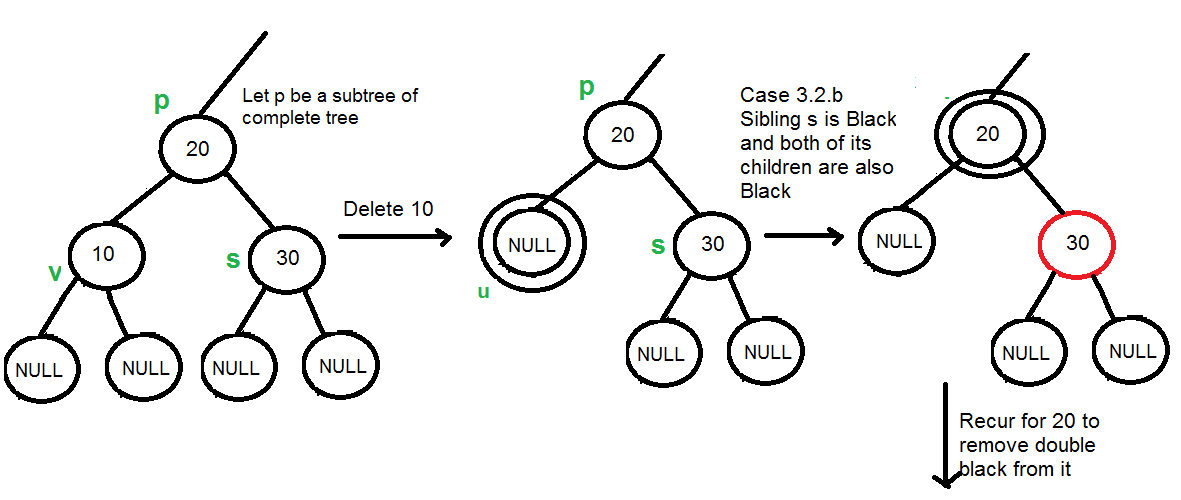
**Trường hợp 2:**

Nếu Node cần xóa có Node u là Node đen kép và nó không phải là Node gốc. khi này ta sẽ gọi s là Node anh chị em của v. Ta sẽ có 2 trường hợp con khác:

Thứ nhất: Nếu s là màu đen và một trong số con của s có màu đỏ ta sẽ thực hiện các phép quay. Ta gọi con màu đỏ của s đó là r thì việc thực hiện phép quay tùy thuộc vào vị trí của s và r.

* **Trường hợp Left Left**: s là con trái của cha và r là con trái của s.
* **Trường hợp Left Right**: s là con trái của cha và r là con phải.
* **Trường hợp Right Right**: s là con bên phải của cha và r là con bên phải của s.
* **Trường hợp Right Left**: s là con bên phải của cha và r là con bên trái của s.

Thú hai: Nếu s là màu đen và cả hai con của nó đều màu đen, ta thực hiện đổi màu và lặp lại cho cha của nó nếu cha của nó cũng màu đen. Trong trường hợp cha của nó màu đỏ thì ta không cần lặp lại, ta chỉ cần đặt lại cho nó màu đen.



**Trường hợp 3:**

Nếu Node cần xóa v là Node gốc, ta thực hiện đổi màu cho u thành đen và khi đó độ dài đen của cây chỉ giảm đi một.

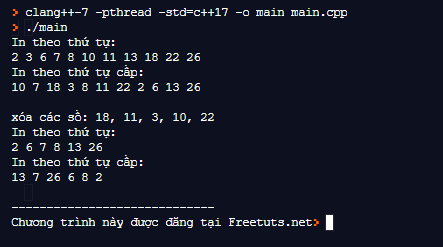
**2. Ví dụ xóa Node khỏi cây đỏ đen**

Trong ví dụ này mình sẽ thực hiện thêm vào một số Node là số nguyên, sau đó xóa một vài số và hiển thị kết quả ra màn hình. Trong code mình đã có chú thích các bạn chú ý theo dõi nhé.

**Full code:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340  341  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360  361  362  363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378  379  380  381  382  383  384  385  386  387  388  389  390  391  392  393  394  395  396  397  398  399  400  401  402  403  404  405  406  407  408  409  410  411  412  413  414  415  416  417  418  419  420  421  422  423  424  425  426  427  428  429  430  431  432  433  434  435  436  437  438  439  440  441  442  443  444  445  446  447  448  449  450  451  452  453  454  455  456  457  458  459  460  461  462  463  464  465  466  467  468  469  470  471  472  473  474  475  476  477  478  479 | #include <iostream>  #include<queue>  using namespace std;  //khai báo thuộc tính color  enum COLOR { RED, BLACK };  //tạo cấy trúc node  class Node {  public:  **int** val;    COLOR color;    Node \*left, \*right, \*parent;      Node(**int** val) : val(val) {      parent = left = right = NULL;      // Nút được tạo trong quá trình chèn      // Nút có màu đỏ khi chèn      color = RED;    }      // trả về con trỏ tới node chú    Node \*uncle() {      // Nếu không có node cha hoặc node ông, thì không có node chú      if (parent == NULL or parent->parent == NULL)        return NULL;        if (parent->isOnLeft())        // node chú bên phải        return parent->parent->right;      else        //node chú bên trái        return parent->parent->left;    }      // kiểm tra xem node có phải là node con của cha không  **bool** isOnLeft() { return this == parent->left; }      // trả về con trỏ cho node anh chị em    Node \*sibling() {      // Node anh rỗng nếu không tồn tại node cha      if (parent == NULL)        return NULL;        if (isOnLeft())        return parent->right;        return parent->left;    }      // di chuyển nút xuống và di chuyển nút đã cho vào vị trí của nó    void moveDown(Node \*nParent) {      if (parent != NULL) {        if (isOnLeft()) {          parent->left = nParent;        } else {          parent->right = nParent;        }      }      nParent->parent = parent;      parent = nParent;    }    **bool** hasRedChild() {      return (left != NULL and left->color == RED) or             (right != NULL and right->color == RED);    }  };    class RBTree {    Node \*root;      // xoay trái node đã cho    void leftRotate(Node \*x) {      // node cha mới sẽ là con bên phải của nút      Node \*nParent = x->right;        // cập nhật gốc nếu nút hiện tại là gốc      if (x == root)        root = nParent;        x->moveDown(nParent);        // kết nối x với phần tử bên trái của cha mẹ mới      x->right = nParent->left;      // kết nối phần tử bên trái của cha mẹ mới với nút      // nếu nó không phải là null      if (nParent->left != NULL)        nParent->left->parent = x;        // kết nối cha mẹ mới với x      nParent->left = x;    }      void rightRotate(Node \*x) {      // cha mẹ mới sẽ là con bên trái của nút      Node \*nParent = x->left;        // cập nhật gốc nếu nút hiện tại là gốc      if (x == root)        root = nParent;        x->moveDown(nParent);        // kết nối x với phần tử bên phải của cha mẹ mới      x->left = nParent->right;      //kết nối phần tử bên phải của cha mẹ mới với nút      // nếu nó không phải là null      if (nParent->right != NULL)        nParent->right->parent = x;        // kết nối cha mẹ mới với x      nParent->right = x;    }      void swapColors(Node \*x1, Node \*x2) {      COLOR temp;      temp = x1->color;      x1->color = x2->color;      x2->color = temp;    }      void swapValues(Node \*u, Node \*v) {  **int** temp;      temp = u->val;      u->val = v->val;      v->val = temp;    }      // sửa màu đỏ đỏ tại nút nhất định    void fixRedRed(Node \*x) {      // nếu x là màu gốc, nó là màu đen và trả về      if (x == root) {        x->color = BLACK;        return;      }        // khởi tạo cha mẹ, ông bà, chú      Node \*parent = x->parent, \*grandparent = parent->parent,           \*uncle = x->uncle();        if (parent->color != BLACK) {        if (uncle != NULL && uncle->color == RED) {          // chú màu đỏ, thực hiện tô màu và đệ quy          parent->color = BLACK;          uncle->color = BLACK;          grandparent->color = RED;          fixRedRed(grandparent);        } else {          // Các hoạt động khác LR, LL, RL, RR          if (parent->isOnLeft()) {            if (x->isOnLeft()) {              // cho left right              swapColors(parent, grandparent);            } else {              leftRotate(parent);              swapColors(x, grandparent);            }            // cho left left và left right            rightRotate(grandparent);          } else {            if (x->isOnLeft()) {              // cho right left              rightRotate(parent);              swapColors(x, grandparent);            } else {              swapColors(parent, grandparent);            }              // cho right right và right left            leftRotate(grandparent);          }        }      }    }      // tìm nút không có nút con bên trái    // trong cây con của nút đã cho    Node \*successor(Node \*x) {      Node \*temp = x;        while (temp->left != NULL)        temp = temp->left;        return temp;    }      // tìm nút thay thế nút đã xóa trong BST    Node \*BSTreplace(Node \*x) {      // khi nút có 2 con      if (x->left != NULL and x->right != NULL)        return successor(x->right);        // khi node lá      if (x->left == NULL and x->right == NULL)        return NULL;        // khi node có một con      if (x->left != NULL)        return x->left;      else        return x->right;    }      // xóa nút đã cho    void deleteNode(Node \*v) {      Node \*u = BSTreplace(v);        // Đúng khi u và v đều đen  **bool** uvBlack = ((u == NULL or u->color == BLACK) and (v->color == BLACK));      Node \*parent = v->parent;        if (u == NULL) {        // u là NULL do đó v là lá        if (v == root) {          // v là root, làm cho root là null          root = NULL;        } else {          if (uvBlack) {            // u và v đều đen            // v là lá, sửa màu đen kép tại v            fixDoubleBlack(v);          } else {            // u hoặc v là đỏ            if (v->sibling() != NULL)              // node anh chị em không rỗng, làm cho nó màu đỏ"              v->sibling()->color = RED;          }            // xóa v khỏi cây          if (v->isOnLeft()) {            parent->left = NULL;          } else {            parent->right = NULL;          }        }        delete v;        return;      }        if (v->left == NULL or v->right == NULL) {        // v có 1 node con        if (v == root) {          // v là gốc, gán giá trị của u cho v và xóa u          v->val = u->val;          v->left = v->right = NULL;          delete u;        } else {          // Tách v khỏi cây và di chuyển u lên          if (v->isOnLeft()) {            parent->left = u;          } else {            parent->right = u;          }          delete v;          u->parent = parent;          if (uvBlack) {            // u và v đều đen, sửa hai màu đen ở u            fixDoubleBlack(u);          } else {            // u hoặc v đỏ, màu u đen            u->color = BLACK;          }        }        return;      }        // v có 2 con, hoán đổi giá trị với kế nhiệm và đệ quy      swapValues(u, v);      deleteNode(u);    }      void fixDoubleBlack(Node \*x) {      if (x == root)        // x là node gốc thì return        return;        Node \*sibling = x->sibling(), \*parent = x->parent;      if (sibling == NULL) {        // Không có sibiling, màu đen kép được đẩy lên        fixDoubleBlack(parent);      } else {        if (sibling->color == RED) {          // Anh chị em màu đỏ          parent->color = RED;          sibling->color = BLACK;          if (sibling->isOnLeft()) {            // trường hợp left            rightRotate(parent);          } else {            // trường hợp right            leftRotate(parent);          }          fixDoubleBlack(x);        } else {          // Anh chị em đen          if (sibling->hasRedChild()) {            // ít nhất 1 trẻ em màu đỏ            if (sibling->left != NULL and sibling->left->color == RED) {              if (sibling->isOnLeft()) {                // left left                sibling->left->color = sibling->color;                sibling->color = parent->color;                rightRotate(parent);              } else {                // right left                sibling->left->color = parent->color;                rightRotate(sibling);                leftRotate(parent);              }            } else {              if (sibling->isOnLeft()) {                // left right                sibling->right->color = parent->color;                leftRotate(sibling);                rightRotate(parent);              } else {                // right right                sibling->right->color = sibling->color;                sibling->color = parent->color;                leftRotate(parent);              }            }            parent->color = BLACK;          } else {            // hai con đen            sibling->color = RED;            if (parent->color == BLACK)              fixDoubleBlack(parent);            else              parent->color = BLACK;          }        }      }    }      // in thứ tự cho node    void levelOrder(Node \*x) {      if (x == NULL)        return;      queue<Node \*> q;      Node \*curr;      q.push(x);        while (!q.empty()) {        curr = q.front();        q.pop();        cout << curr->val << " ";        if (curr->left != NULL)          q.push(curr->left);        if (curr->right != NULL)          q.push(curr->right);      }    }      // in đệ quy order    void inorder(Node \*x) {      if (x == NULL)        return;      inorder(x->left);      cout << x->val << " ";      inorder(x->right);    }    public:    RBTree() { root = NULL; }      Node \*getRoot() { return root; }    Node \*search(**int** n) {      Node \*temp = root;      while (temp != NULL) {        if (n < temp->val) {          if (temp->left == NULL)            break;          else            temp = temp->left;        } else if (n == temp->val) {          break;        } else {          if (temp->right == NULL)            break;          else            temp = temp->right;        }      }        return temp;    }      // chen giá trị đã cho vào cây    void insert(**int** n) {      Node \*newNode = new Node(n);      if (root == NULL) {        newNode->color = BLACK;        root = newNode;      } else {        Node \*temp = search(n);          if (temp->val == n) {          return;        }        newNode->parent = temp;          if (n < temp->val)          temp->left = newNode;        else          temp->right = newNode;          fixRedRed(newNode);      }    }      // chức năng tiện ích xóa nút có giá trị nhất định    void deleteByVal(**int** n) {      if (root == NULL)        // Tree is empty        return;        Node \*v = search(n), \*u;        if (v->val != n) {        cout << "Không tìm thấy nút nào để xóa với giá trị:" << n << endl;        return;      }        deleteNode(v);    }      // in theo thứ tự    void printInOrder() {      cout << "In theo thứ tự: " << endl;      if (root == NULL)        cout << "cây rỗng" << endl;      else        inorder(root);      cout << endl;    }      // in theo thứ tự cấp    void printLevelOrder() {      cout << "In theo thứ tự cấp: " << endl;      if (root == NULL)        cout << "cây rỗng" << endl;      else        levelOrder(root);      cout << endl;    }  };    **int** main() {    RBTree tree;    //insert dữ liệu    tree.insert(7);    tree.insert(3);    tree.insert(18);    tree.insert(10);    tree.insert(22);    tree.insert(8);    tree.insert(11);    tree.insert(26);    tree.insert(2);    tree.insert(6);    tree.insert(13);    //gọi hàm in    tree.printInOrder();    tree.printLevelOrder();      cout<<endl<<"xóa các số: 18, 11, 3, 10, 22"<<endl;    //thực hiện xóa node    tree.deleteByVal(18);    tree.deleteByVal(11);    tree.deleteByVal(3);    tree.deleteByVal(10);    tree.deleteByVal(22);    //gọi hàm in    tree.printInOrder();    tree.printLevelOrder();      cout<<"\n-----------------------------\n";    cout<<"Chương trình này được đăng tại Freetuts.net";  } |

**Kết quả:**



**3. Kết luận**

Như vậy là chúng ta đã tìm hiểu xong cách xóa một Node khỏi cây đỏ đen, cũng như tìm hiểu cách khắc phục các trường hợp vi phạm quy tắc cây đỏ đen. Cũng như thực hiện một ví dụ xóa các số nguyên trong cây đỏ đen số nguyên. Chúc các bạn thực hiện thành công !!!