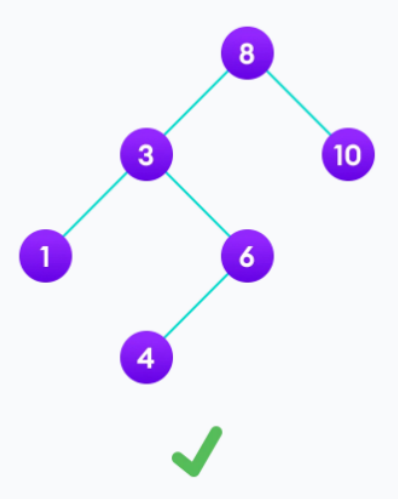
Binary Search Tree

Search Tree

Cây tìm kiếm có tính chất sau:

> Giá trị của key bất kỳ luôn lớn hơn giá trị của các key trong node bên trái và nhỏ hơn giá trị của các key trong node bên phải.



Ở đây dùng từ "key" (khóa) bởi vì key và node là khác nhau, một node có thể có nhiều key (sẽ được nhắc lại trong phần cây 2 - 3 - 4).

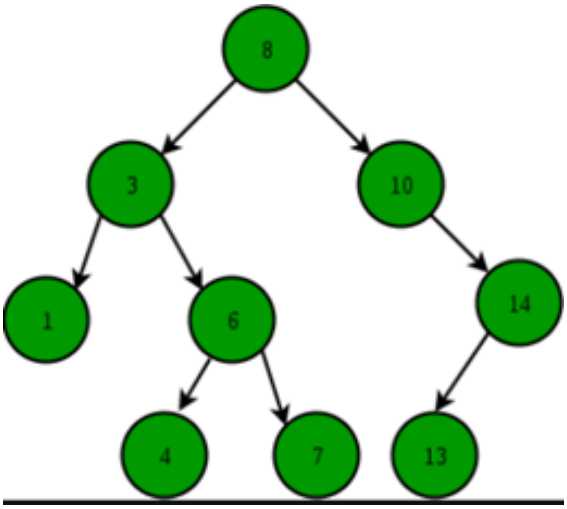
Không được có hai node có key trùng nhau. (Giống như là khóa chính trong SQL, chúng là unique và có thể có nhiều trường).

Binary Search Tree

Cây nhị phân tìm kiếm là cây nhị phân mà thỏa tính chất của cây tìm kiếm ở trên. Tức là:

- Mỗi node phải có giá trị lớn hơn giá trị của node con bên trái và bé hơn giá trị của node con bên phải.
- Hai cây con của một node bất kỳ cũng phải là cây nhị phân tìm kiếm.

Cây nhị phân tìm kiếm phụ thuộc vào dữ liệu đầu vào, nếu như dữ liệu ấy là xấu thì các thao tác thực hiện sẽ có chi phí cao.



Fact thú vị: khi duyệt giữa cây nhị phân tìm kiếm, do cấu trúc của nó được sắp theo thứ tự là left root right nên kết quả của phép duyệt sẽ là một danh sách các số có thứ tự.

Operation

Search

Idea

Ta bắt đầu duyệt từ node gốc. Nếu giá trị cần tìm nhỏ hơn node gốc, tìm bên cây con trái. Ngược lại tìm bên cây con phải. Trường hợp node đó rỗng hoặc là node cần tìm thì trả về.

Code

```
NODE *Search(NODE *root, int x)
{
    if (root == nullptr)
        return root;
    if (x < root->key)
```

```
        return Search(root->left, x);
    else if (x > root->key)
        return Search(root->right, x);
    else
        return root;
}
```

Insert

Idea

Tiến hành duyệt trước qua các phần tử trong cây. Mỗi lần duyệt ta sẽ kiểm tra node cần thêm, nếu nó nhỏ hơn node gốc thì ta duyệt cây con bên trái, ngược lại duyệt cây con bên phải. Nếu tại node đó rỗng, ta sẽ tạo một node mới và thêm vào cây.

Giá trị trả về nên là int. Với 1 là thêm thành công và 0 là thêm thất bại.

Code

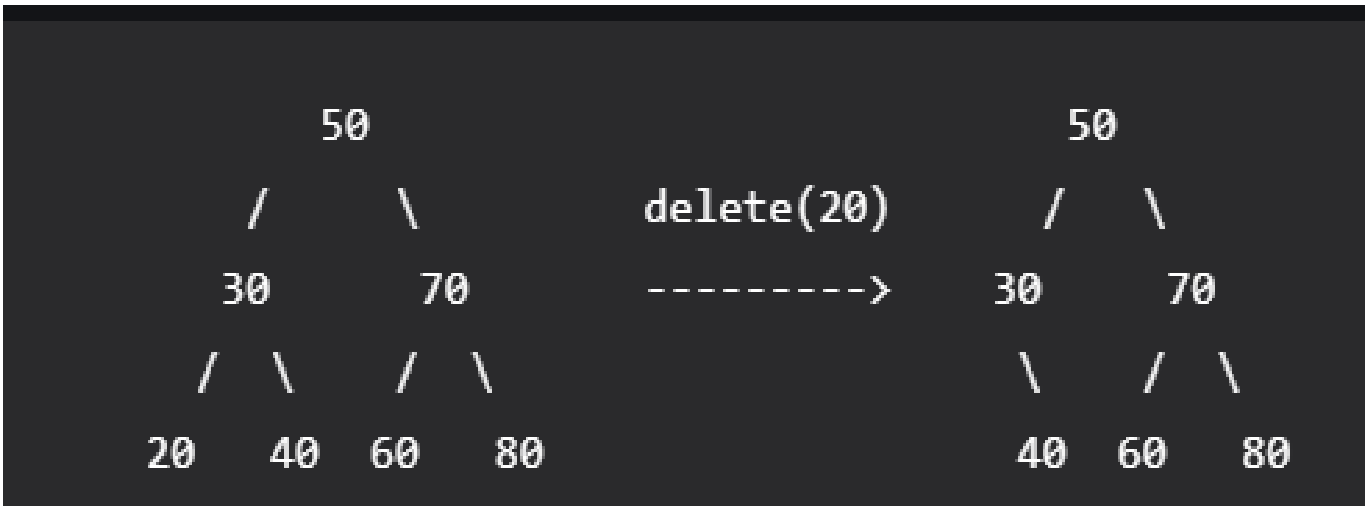
```
void Insert(NODE *&root, int x)
{
    if (root == nullptr)
        root = createNODE(x);

    if (x < root->key)
        Insert(root->left, x);
    else if (x > root->key)
        Insert(root->right, x);
    else
        return;
}
```

Remove

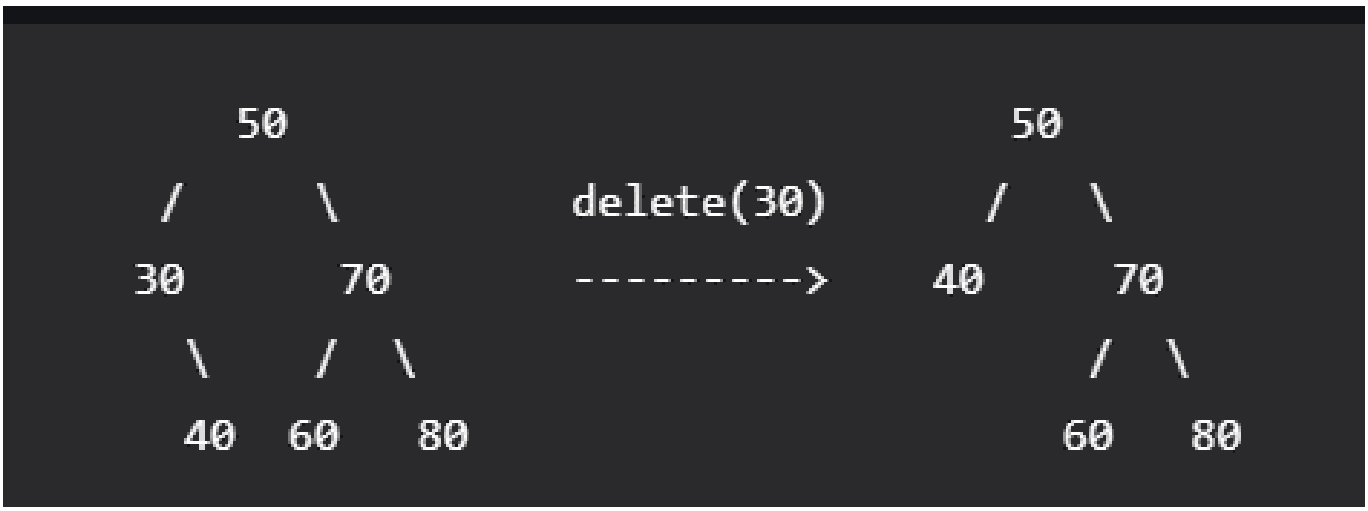
Idea

Có ba khả năng xảy ra khi remove một node trong BST.

1. Node đó là một lá: Chúng ta chỉ đơn giản xóa nó ra khỏi cây.



2. Node đó có duy nhất một con: Sao chép con đó cho node cần xóa và xóa node con.



3. Node có 2 con:

- Tìm node lớn nhất cây con trái hoặc node nhỏ nhất cây con phải của node hiện tại, tạm gọi là C.
- Tạo một node tạm để lưu node cha của C.
- Hoán đổi giá trị của node cần xóa với C.
- Nếu C là node lớn nhất cây con trái thì xóa con phải của node cha. Ngược lại nếu C là node nhỏ nhất cây con phải thì xóa con trái của node cha.

Code

Code tìm node cha của node nhỏ nhất bên cây con phải:

```
NODE *searchRightMin(NODE *&curr)
{
    NODE *parent = curr;
    // Search in right subtree
```

```
    curr = curr->right;
    while (curr->left != nullptr)
    {
        parent = curr; // Parent be prev root
        curr = curr->left;
    }
    return parent;
}
```

Code tìm node cha của node lớn nhất bên cây con trái:

```
NODE *searchLeftMax(NODE *&curr)
{
    NODE *parent = curr;
    // Search in left subtree
    curr = curr->left;
    while (curr->right != nullptr)
    {
        parent = curr; // Parent be prev root
        curr = curr->right;
    }
    return parent;
}
```

Code xóa một node trong BST:

```
void Remove(NODE *&pRoot, int x)
{
    if (pRoot == nullptr)
        return;

    if (x < pRoot->key)
        Remove(pRoot->left, x);
    else if (x > pRoot->key)
        Remove(pRoot->right, x);
    else
    {
        // Trường hợp có không hoặc một con xử lý đồng thời
        if (pRoot->left == nullptr)
        {
            pRoot = pRoot->right;
        }
        else if (pRoot->right == nullptr)
        {
            pRoot = pRoot->left;
        }
```

```
        // Trường hợp có hai con
        else
        {
            // Tìm node cha của thế mạng
            NODE *move = pRoot;
            NODE *parent = searchRightMin(move);

            // Ghi đè giá trị
            pRoot->key = move->key;

            // Xóa node con thế mạng
            if (parent->left == move)
            {
                temp = parent->left;
                parent->left = nullptr;
            }
            else if (parent->right == move)
            {
                temp = parent->right;
                parent->right = nullptr;
            }
            delete temp;
        }
    }
}
```

Complexity of Search, Insert and Remove

Worst case Xảy ra khi đây là một cây lệch, lúc này cần duyệt qua N phần tử để có thể thực hiện thao tác.

Best case Khi cây là gần như hoàn hảo hoặc hoàn hảo. Lúc này chiều cao của cây là tối thiểu $log_2(N+1)$ nên các thao tác thêm và xóa sẽ tốn ít chi phí nhất.

Đối với thao tác xóa, nếu phần tử cần xóa nằm ở đầu thì chi phí cho việc tìm kiếm phần tử cần xóa là $O(1)$. Tuy nhiên chi phí để tìm kiếm phần tử thế mạng sẽ là $O(log_2(n))$ nên tổng độ phức tạp thời gian vẫn là $O(log_2(n))$. Nói cách khác, chi phí tìm kiếm phần tử cần xóa và phần tử thế mạng là bù trừ cho nhau.

Best case of search Trường hợp tốt nhất xảy ra khi tìm kiếm là phần tử cần tìm nằm ở node gốc, nên có độ phức tạp thời gian $O(1)$.

Cases	Complexity

Cases	Complexity
Best case	$O(log_2(n))$
Best case of Search	$O(1)$
Worst case	$O(n)$
Average case	$O(log_2(n))$

Space Complexity: $O(n)$ (Lưu giữ các lời gọi đệ quy).

Counting

Count less

Đếm số node bé hơn một giá trị cho trước.

```
int countLess(NODE *root, int x)
{
    if (root == nullptr)
        return 0;

    // Nếu lớn hơn node hiện tại, cộng tất cả các node bên cây con trái và node
hiện tại.
    // Sau đó duyệt sang cây con phải.
    if (x > root->key)
        return 1 + countNode(root->left) + countLess(root->right, x);

    // Hoặc có thể dùng cách khác, duyệt sang cả hai bên của cây khi x > root-
>key.
    // return 1 + countLess(root->left,x) + countLess(root->right, x);

    // Nếu nhỏ hơn hoặc bằng thì chỉ duyệt sang cây con trái.
    else
        return countLess(root->left, x);
}
```

Count Greater

Đếm số node lớn hơn một giá trị cho trước. Idea ngược lại với Count Less.

```
int countGreater(NODE *root, int x)
{
    if (root == nullptr)
    {
        return 0;
    }

    // Nếu nhỏ hơn, đếm tất cả các node bên cây con phải và node hiện tại.
    // Sau đó duyệt sang cây con trái.
    if (x < root->key)
        return 1 + countNode(root->right) + countGreater(root->left, x);

    // Nếu lớn hơn hoặc bằng thì chỉ duyệt sang cây con phải.
    else
        return countGreater(root->right, x);
}
```

Is BST ?

Method 1

Idea

Thuật toán chứng minh một cây là cây nhị phân tìm kiếm có hai phần.

Phần đầu tiên là chứng minh nó là cây nhị phân tìm kiếm cục bộ, tức là xét với mỗi node thì nó đều lớn hơn con trái và nhỏ hơn con phải.

Phần thứ hai là xét toàn cục, với mỗi node thì tìm node nhỏ nhất của cây con bên phải so với nó, nếu lớn hơn thì là BST. Tương tự tìm node lớn nhất của cây con bên trái, nếu nhỏ hơn thì là BST.

Code

```
bool isBST(NODE *pRoot)
{
    if (pRoot != nullptr)
    {
        NODE *rightMin = pRoot;
        NODE *leftMax = pRoot;
```

```
        // Tìm phần tử lớn nhất cây con trái
        if (pRoot->left != nullptr)
            searchLeftMax(leftMax);
        // và nhỏ nhất cây con phải
        else if (pRoot->right != nullptr)
            searchRightMin(rightMin);

        // So sánh điều kiện 1
        if ((pRoot->left != nullptr && pRoot->left->key > pRoot->key)
        || (pRoot->right != nullptr && pRoot->right->key < pRoot->key))
            return false;

        // So sánh điều kiện 2
        else if (rightMin->key < pRoot->key || leftMax->key > pRoot->key)
            return false;

        // Xét các cây con
        else
            return isBST(pRoot->left) && isBST(pRoot->right);
    }
    //if it's null, it means it's parent is satisfy BST
    else
        return true;
}
```

Complexity

Ở mỗi node, cần phải đi tìm phần tử lớn nhất cây con trái (maxLeft) và nhỏ nhất cây con phải (minRight) nên tốn chi phí $O(log_2(n))$.

Chẳng hạn khi ở node gốc, cần phải lặp H vòng lặp để tìm được maxLeft và minRight . Node ở mức 1 thì cần $H-1$, tương tự ở mức i bất kỳ thì cần $H-i$ lần lặp.

Tuy nhiên tối đa vẫn là H vòng lặp.

Ta đã biết

$$log_2(N+1) \leq H \leq N$$

Từ đó suy ra chi phí cho các vòng lặp sẽ là: Worst case: $O(n)$ Best case: $O(log_2(n))$

Mà cần phải duyệt qua N node trong cây để kiểm chứng điều này. Do đó:

- Độ phức tạp thời gian: Best case: $O(nlog_2(n))$. Worst case: $O(n^2)$.
- Độ phức tạp không gian: $O(n)$ (Lưu giữ các lời gọi đệ quy).

Method 2

Ngoài ra cũng còn một cách kiểm tra nữa sử dụng một mảng phụ. Ta tiến hành duyệt giữa cây, mỗi lần đến một node thì lưu node đó vào mảng. Nếu mảng của chúng ta giảm dần (đệ quy cho ra mảng ngược) hoặc tăng dần (duyệt mảng ngược) thì là BST. Cách này cũng có thể kiểm tra phần tử trùng trong cây bằng cách xét hai phần tử liền kề.

Code

Code duyệt giữa và thêm vào mảng phụ:

```
void inorderToArray(NODE *pRoot, int *a, int &n)
{
    if (pRoot == nullptr)
        return;

    inorderToArray(pRoot->left, a, n);
    a[--n] = pRoot->key;
    inorderToArray(pRoot->right, a, n);
}
```

Code chứng minh BST:

```
bool isBST2(NODE *pRoot)
{
    int n = countNode(pRoot);
    int temp = n;
    int *a = new int[n];
    inorderToArray(pRoot, a, temp);

    for (int i = 0; i < n; i++)
    {
        if (a[i] <= a[i + 1])
            return false;
    }
    return true;
}
```

Có thể kiểm tra hai phần tử liền kề có tăng dần (giảm dần) mà không cần sử dụng mảng phụ:

```
bool isBST(NODE *root, NODE *&prev)
{
    // Nếu có thể chạm đến node lá, nghĩa là một số node ở trên đường đi đến nó đã thỏa mãn BST
    if (root == nullptr)
        return true;

    // Nếu như cây con trái không là BST thì return false
    // Nếu là BST thì xét tại node đó rồi xét con phải
    if (isBST(root->left, prev) == false)
        return false;

    // Nếu vi phạm BST thì phát hiện
    if (prev != nullptr && root->key <= prev->key)
        return false;
    prev = root;

    // Cây con phải thì return và duyệt tiếp.
    return isBST(root->right, prev);
}
```

Complexity

Do duyệt qua mọi phần tử trong cây nên độ phức tạp luôn là $O(n)$ trong cả ba case. Mảng phụ có thể triệt tiêu trong cách xây dựng code thứ hai. Do đó:

Time Complexity: $O(n)$. Space Complexity: $O(1)$.

Is Full BST ?

Để chứng minh một cây nhị phân tìm kiếm là đầy đủ thì cần hai điều kiện: nó là BST và nó là cây nhị phân đầy đủ. Thuật toán chứng minh cây nhị phân là đầy đủ có trong bài Binary Tree.

Tổng độ phức tạp thời gian sẽ là $O(n + n) \sim O(n)$. Độ phức tạp không gian là $O(1)$.