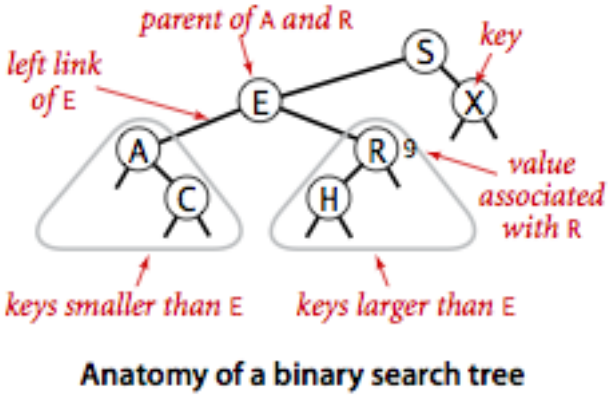
TH 01 - CÂY NHỊ PHÂN TÌM KIẾM (BINARY SEARCH TREE)

01/2018

1 Thực hành

Cây nhị phân tìm kiếm (Binary search tree - BST) là một cấu trúc dữ liệu mở rộng của cây nhị phân thỏa tính chất quan trọng: *giá trị được lưu ở node hiện tại lớn hơn giá trị được lưu ở node bên phải và nhỏ hơn giá trị được lưu ở node bên trái.*

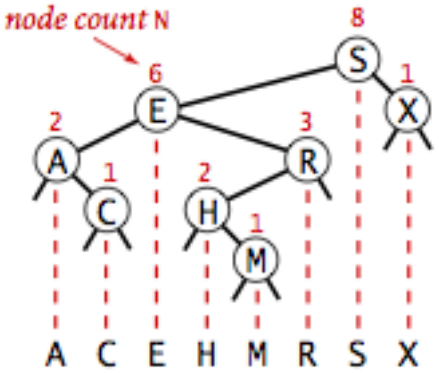


Hình 1: Phân tích cây BST

1.1 Khai báo Node của cây

Tạo một lớp *Node* để định nghĩa một node của cây BST. Mỗi lớp chứa một khoá (key), một liên kết trái (left), liên kết phải (right) và một biến đếm

số lượng node trong cây (size). Liên kết trái sẽ trỏ đến các phần tử có giá trị nhỏ hơn và liên kết phải trỏ đến các phần tử có giá trị lớn hơn. Biến size dùng lưu trữ số lượng node trong cây tại node đang xét.



Hình 2: Cây BST với node chứa thông tin và size (node count)

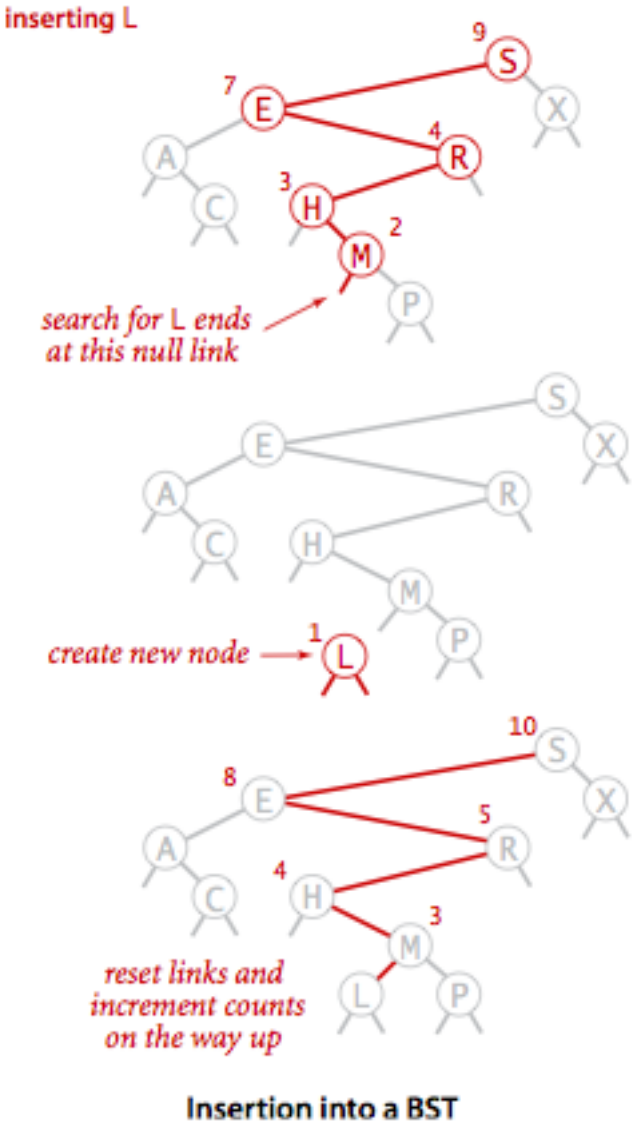
```
private class Node {
    private Integer key;
    private Node left, right;
    private int size;

    public Node(Integer key, int size) {
        this.key = key;
        this.size = size;
    }
}
```

1.2 Thêm một khoá vào cây

Việc thêm một phần tử mới vào cây BST phải đảm bảo điều kiện ràng buộc của cây: *giá trị được lưu ở node hiện tại lớn hơn giá trị được lưu ở node bên phải và nhỏ hơn giá trị được lưu ở node bên trái*. Để đáp ứng được điều kiện ràng buộc này ta có thể thêm vào nhiều vị trí trên cây sao cho thoả mãn điều kiện của cây, tuy nhiên phương pháp đơn giản nhất vẫn là thêm vào nút lá của cây.

Như vậy việc thêm vào nút lá của cây sẽ hoàn toàn tương tự với thao tác tìm kiếm, kết thúc quá trình tìm kiếm cũng chính là vị trí cần thêm.



Hình 3: Thêm một khoá vào cây

```
private Node put(Node x, Integer key) {
    if (x == null)
        return new Node(key, 1);
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = put(x.left, key);
```

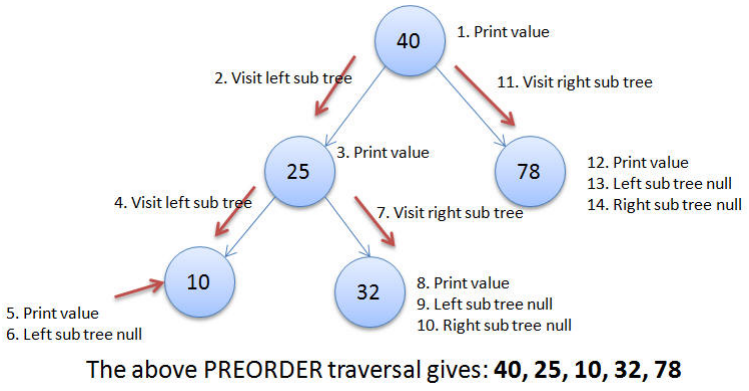
```
    else if (cmp > 0)
        x.right = put(x.right, key);
    else
        x.key = key;
    x.size = 1 + size(x.left) + size(x.right);
    return x;
}
```

1.3 Duyệt cây

Duyệt cây BST là duyệt qua các node trong cây và thực hiện một thao tác nào đó tương ứng với mỗi node được duyệt, trong trường hợp này thì thao tác duyệt cây đơn giản là thực hiện việc xuất giá trị của khoá trong node ra màn hình.

Khi duyệt một cây, chúng ta sẽ lần lượt duyệt giá trị của node hiện tại, sau đó lần lượt duyệt qua cây con trái và cây con phải theo phương pháp đệ quy. Thứ tự chúng ta thực hiện 3 thao tác duyệt giá trị, duyệt cây con trái, duyệt cây con phải sẽ tạo ra ba kỹ thuật duyệt cây khác nhau.

1. PreOrder (node-left-right) Trong cách duyệt này thì giá trị của nút đang xét sẽ được in ra trước, sau đó cây con bên trái của nút đang duyệt sẽ được duyệt tiếp theo và cuối cùng là duyệt cây con bên phải. Quá trình này sẽ được thực hiện một cách đệ quy cho tất cả các node trong cây.

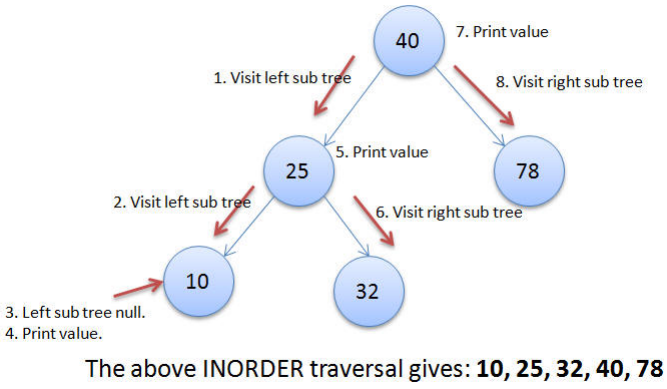


Hình 4: Duyệt cây PreOrder (node-left-right)

```
    public void nlr(Node x) {
```

```
        if (x != null) {
            System.out.print(x.key + " ");
            nlr(x.left);
            nlr(x.right);
        }
    }
```

2. InOrder (left-node-right) Trong cách duyệt này thì cây con bên trái của nút được chỉ định duyệt sẽ được duyệt trước, sau đó giá trị của nút đang duyệt sẽ được in ra và cuối cùng là duyệt cây con bên phải. Quá trình này sẽ được thực hiện một cách đệ quy cho tất cả các node trong cây.

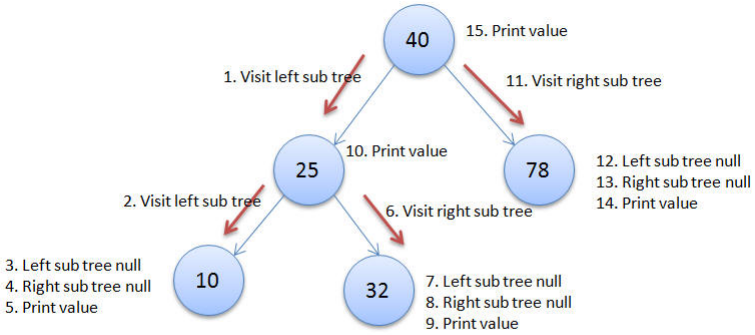


Hình 5: Duyệt cây InOrder (left-node-right)

```
    public void lnr(Node x) {
        // your code
    }
```

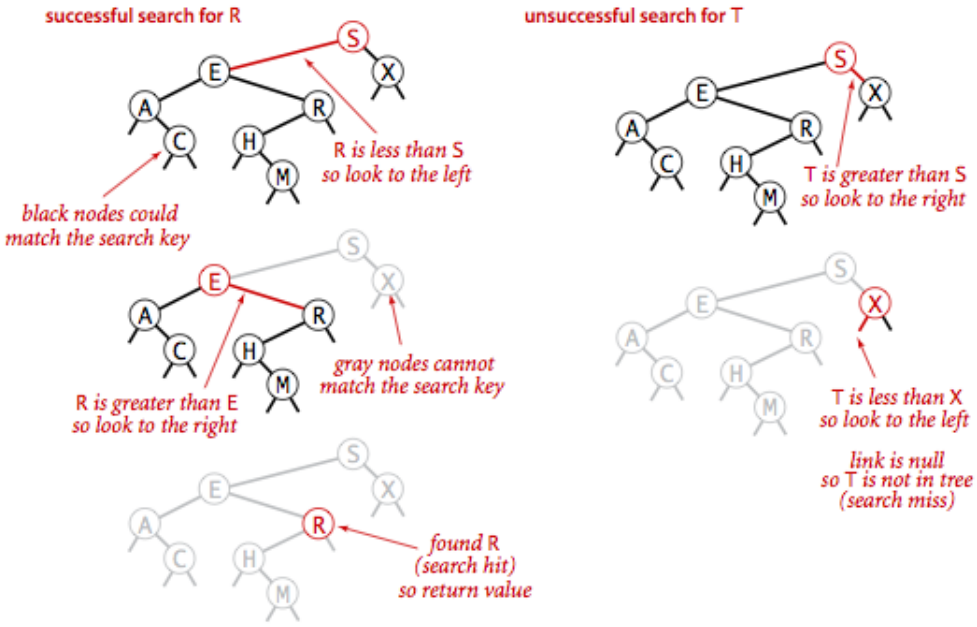
3. PostOrder (left-right-node) Trong cách duyệt này thì cây con bên trái của nút được chỉ định duyệt sẽ được duyệt trước, sau đó thì cây con bên phải sẽ được duyệt tiếp theo và cuối cùng là giá trị của nút đang duyệt sẽ được in ra. Quá trình này sẽ được thực hiện một cách đệ quy cho tất cả các node trong cây.

```
    public void lrn(Node x) {
        // your code
    }
```



Hình 6: Duyệt cây PostOrder (left-right-node)

1.4 Tìm một khoá trong cây



Hình 7: Tìm một khoá trong cây

Do cây BST có cấu trúc đệ quy nên việc tìm kiếm trên cây sẽ dễ dàng

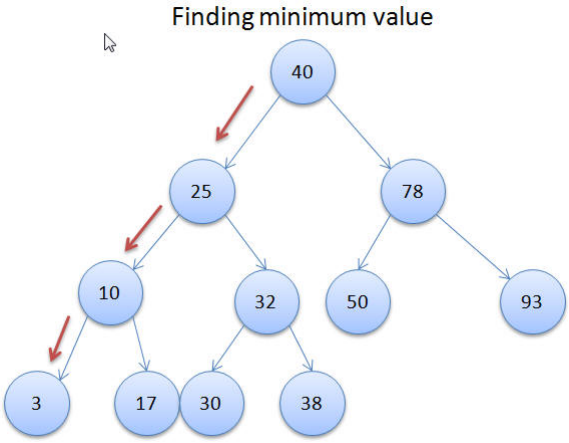
được thực hiện bởi một thuật toán đệ quy: Nếu cây rỗng, quá trình tìm kiếm kết thúc với giá trị trả về là null; nếu khoá cần tìm bằng với khoá ở nút gốc, kết thúc quá trình tìm kiếm với kết quả trả về là nút gốc. Ngược lại, chúng ta sẽ tìm kiếm (đệ quy) trong các cây con tương ứng.

```
private Node get(Node x, Integer key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        return get(x.left, key);
    else if (cmp > 0)
        return get(x.right, key);
    else
        return x;
}
```

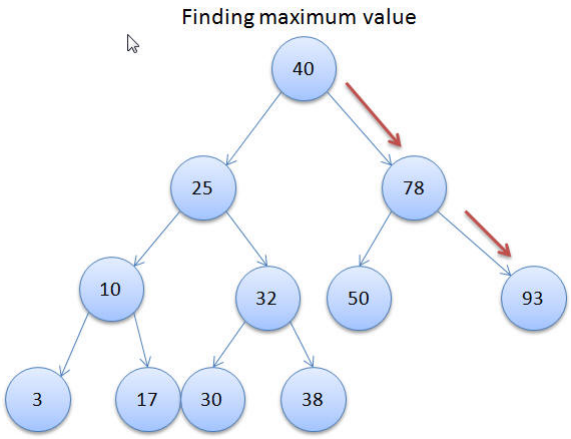
1.5 Tìm giá trị lớn nhất, nhỏ nhất trong cây

Do tính chất ràng buộc của cây BST nên dễ thấy nút trái nhất của cây cũng chính là nút có giá trị nhỏ nhất trong cây, tương tự nút phải nhất của cây là nút có giá trị lớn nhất trong cây.

Như vậy để tìm giá trị nhỏ nhất của cây chúng ta đơn giản duyệt cây về phía bên trái cho đến khi nào nút đang duyệt không có cây con trái thì đó chính là nút trái nhất hay nút có giá trị nhỏ nhất của cây. Tương tự duyệt bên phải để tìm nút có giá trị lớn nhất.



Hình 8: Node chứa giá trị nhỏ nhất



Hình 9: Node chứa giá trị lớn nhất

```
private Node min(Node x) {
    if (x.left == null)
        return x;
    else
        return min(x.left);
}

private Node max(Node x) {
    // your code
}
```

2 Bài tập

1. Viết hàm createTree nhận đầu vào là một chuỗi các số nguyên (cách nhau bởi ký tự space). Hàm tạo ra cây BST với các khoá lần lượt là các số nguyên trong chuỗi đầu vào.

```
    public void createTree(String strKey) {
        // your code
    }
```

2. Viết hàm in ra màn hình giá trị của cây theo thứ tự tăng dần.

3. Viết hàm in ra màn hình giá trị của cây theo thứ tự giảm dần.

4. Viết hàm *contains* với tham số đầu vào là một *key*, hàm trả về *True* nếu *key* có trong cây, ngược lại trả về *False*.

```
public boolean contains(Integer key) {
    // your code
}
```

5. Thêm thuộc tính *height* vào Node của cây nhị phân trong bài thực hành như sau:

```
private class Node {
    private Integer key;
    private Node left, right;
    private int size;
    private int height;

    public Node(Integer key, int size, int height) {
        this.key = key;
        this.size = size;
        this.height = height;
    }
}
```

(a) Viết lại hàm *put* để khi thêm một khoá vào cây thì sẽ cập nhật lại chiều cao của từng node trong cây.

```
private Node put(Node x, Integer key) {
    if (x == null)
        // your code
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = put(x.left, key);
    else if (cmp > 0)
        x.right = put(x.right, key);
    else
        x.key = key;
    x.size = 1 + size(x.left) + size(x.right);
    // your code
    return x;
}
```

(b) Viết hàm *height(Node x)* trả về chiều cao của một node x bất kỳ

trong cây.

```
    private int height(Node x) {
        // your code
    }
```

(c) Viết hàm *height()* trả về chiều cao của cây.

```
    private int height() {
        // your code
    }
```

6. Viết hàm *sum(Node x)* tính tổng giá trị các khoá trong Node x bất kỳ.

```
public Integer sum(Node x) {
    // your code
}
```

7. Viết hàm *sum* tính tổng giá trị các khoá trong cây.

```
public Integer sum() {
    // your code
}
```

8. Xây dựng cây nhị phân tìm kiếm mới với Node của cây được khai báo như sau:

```
private class Node {
    private String key;
    private Integer value;
    private Node left, right;
    private int size;

    public Node(Integer key, int size) {
        this.key = key;
        this.size = size;
    }
}
```

Viết hàm createTree() để tạo cây với tham số đầu vào là một câu. Hàm trả về cây BST với *key* và các từ có trong câu và *value* là số lần xuất hiện của từ đó trong câu.

```java
public void createTree(String sent) {
    // your code
}
```