Thuật toán cây nhị phân tìm kiếm (BST)

Cây nhị phân tìm kiếm (BST) là một cấu trúc dữ liệu rất mạnh mẽ và được sử dụng rộng rãi trong các ứng dụng thực tế. Nó là một cấu trúc dữ liệu dạng cây, trong đó mỗi nút chứa một giá trị và hai con trỏ tới hai nút con của nó: nút con bên trái và nút con bên phải. Giá trị của nút con bên trái luôn nhỏ hơn giá trị của nút cha, và giá trị của nút con bên phải luôn lớn hơn giá trị của nút cha.

BST được sử dụng để lưu trữ tập hợp các phần tử và cung cấp các thao tác tìm kiếm, chèn và xóa phần tử với độ phức tạp O(log n). Điều này làm cho nó trở thành một trong những cấu trúc dữ liệu phổ biến nhất trong lĩnh vực khoa học máy tính.

Các thao tác trên cây nhị phân tìm kiếm

Tìm kiếm (search)

Thao tác tìm kiếm trong BST được thực hiện bằng cách so sánh giá trị cần tìm với giá trị của nút hiện tại. Nếu giá trị cần tìm bằng giá trị của nút hiện tại, trả về nút đó. Nếu giá trị cần tìm nhỏ hơn giá trị của nút hiện tại, chuyển đến nút con bên trái và tiếp tục tìm kiếm. Nếu giá trị cần tìm lớn hơn giá trị của nút hiện tại, chuyển đến nút con bên phải và tiếp tục tìm kiếm.

Chèn (insert)

Thao tác chèn một giá trị vào BST được thực hiện bằng cách tìm kiếm vị trí phù hợp cho giá trị đó và thêm giá trị đó vào BST.

Xóa (delete)

Thao tác xóa một giá trị khỏi BST được thực hiện bằng cách tìm kiếm giá trị đó trong BST và xóa nút chứa giá trị đó. Trong trường hợp nút chứa giá trị đó có ít hơn hai con, nút đó được loại bỏ khỏi BST. Trong trường hợp nút chứa giá trị đó có hai con, nút đó được thay thế bằng giá trị lớn nhất trong cây con bên trái của nó hoặc giá trị nhỏ nhất trong cây con bên phải của nó. Sau đó, nút đó được loại bỏ khỏi BST.

Code mẫu

Dưới đây là một số đoạn mã Python cho các thao tác trên cây nhị phân tìm kiếm:

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def search(self, val):
        return self._search(val, self.root)

    def _search(self, val, node):
        if node is None or node.val == val:
            return node
        elif val < node.val:
            return self._search(val, node.left)
        else:
            return self._search(val, node.right)

    def insert(self, val):
        self.root = self._insert(val, self.root)

    def _insert(self, val, node):
        if node is None:
            node = TreeNode(val)
        elif val < node.val:
            node.left = self._insert(val, node.left)
        else:
            node.right = self._insert(val, node.right)
        return node

    def delete(self, val):
        self.root = self._delete(val, self.root)

    def _delete(self, val, node):
        if node is None:
            return node
        elif val < node.val:
```

```
            node.left = self._delete(val, node.left)
        elif val > node.val:
            node.right = self._delete(val, node.right)
        else:
            if node.left is None:
                return node.right
            elif node.right is None:
                return node.left
            else:
                temp = self._max(node.left)
                node.val = temp.val
                node.left = self._delete(temp.val, node.left)
        return node

    def _max(self, node):
        while node.right is not None:
            node = node.right
        return node
```

Với các thao tác tìm kiếm, chèn và xóa trong cây nhị phân tìm kiếm được triển khai trong đoạn mã trên, bạn có thể sử dụng BST để giải quyết các bài toán liên quan đến tìm kiếm, chèn và xóa phần tử. Việc hiểu và ứng dụng cấu trúc dữ liệu này sẽ giúp các nhà phát triển tăng cường hiệu suất và tính ổn định cho các ứng dụng của họ.

Revision #1
Created 27 June 2023 09:25:55 by Laptrinh.vn
Updated 27 June 2023 09:26:40 by Laptrinh.vn