第十八讲

Binary Search Tree

薛浩

2023 年 6 月 6 日

www.stickmind.com

今日话题

话题 1：编程基础 初学编程的新手，一般应该熟练使用函数和库处理字符串相关的编程任务。

话题 2：抽象数据类型的使用 在尝试实现抽象数据类型之前，应该先熟练使用这些工具解决问题。

话题 3：递归和算法分析 递归是一种强有力的思想，一旦掌握就可以解决很多看起来非常难的问题。

话题 4：类和内存管理 使用 C++ 实现数据抽象之前，应先学习 C++ 的内存机制。

话题 5：常见数据结构 在熟练使用抽象数据类型解决常见问题之后，学习如何实现它们是一件很自然的事情。

话题 5：常见数据结构

在熟练使用抽象数据类型解决常见问题之后，学习如何实现它们是一件很自然的事情。

- 链表
- 动态数组
- 二叉堆
- 二叉搜索树

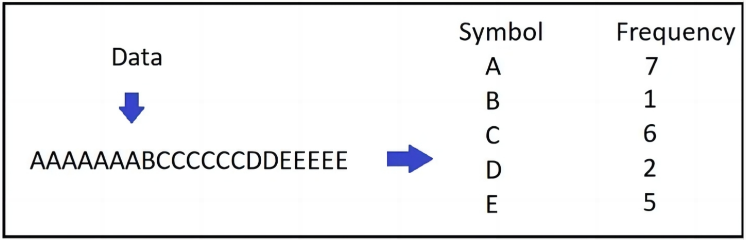


Figure 1: 数据结构和算法

如何用树实现映射和集合?

目录

复习：链表 Linked List

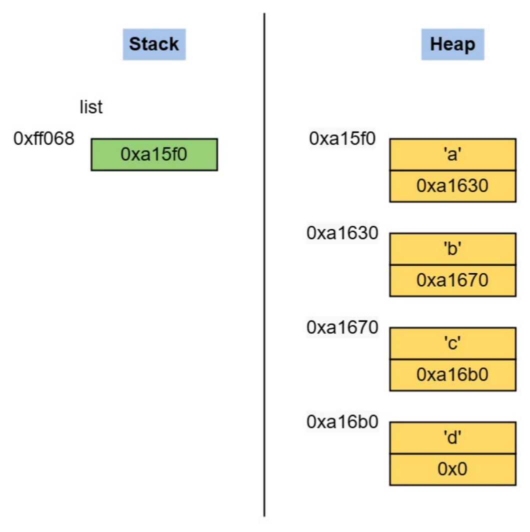
链表 Linked List

链表（LinkedList）是由节点（node）组成的链式数据结构。节点可以使用类似如下递归结构体来表示：

```
struct Node {
    char ch;
    Node* next;
};
```

利用链表实现 Stack 和 Queue 这种无需遍历的线性容器可以得到常数时间的插入和删除操作。但是，当涉及线性搜索和访问时，时间复杂度并不理想。

链表 LinkedList



树的概念 Tree

树的概念 Tree

利用数组索引之间的关系，可以模拟出二叉堆这样的数据结构，改进了动态数组的某些缺点。利用指针，在链表的基础上同样可以模拟出一种层次关系结构，我们称之为**树**（tree）。

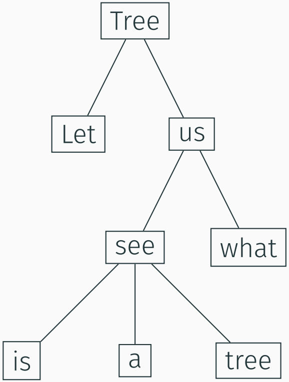
树由节点构成，其特性如下：

- 树必须有一个根节点（root），位于最顶层
- 每个节点和根节点有且只有一条出路

树的概念 Tree

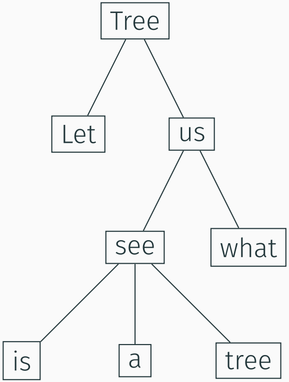
树由节点构成，其特性如下：

- 树必须有一个根节点（root node），位于最顶层
- 每个节点和根节点有且只有一条出路



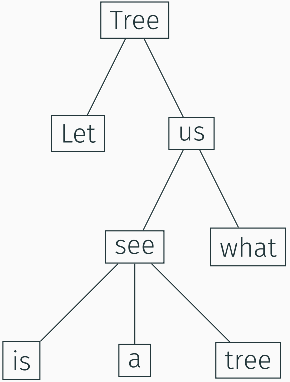
树的概念 Tree

- us，see 等称为父节点（parent node），what，is 等称为子节点（child node）
- 每个节点可以有多个子节点，但只能有一个父节点
- is，a 拥有同一个父节点，称为兄弟节点（sibling nodes）



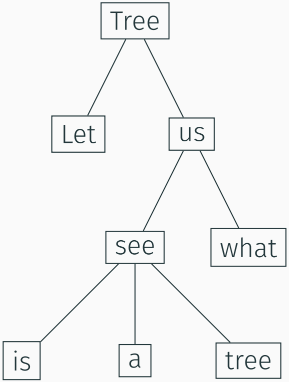
树的概念 Tree

- what，is 等没有子节点的称为叶子节点（leaf node）
- see，us 既不是根节点也不是叶子节点，称为内部节点（interior node）
- 从根节点到叶子节点的最长路径，称为高度（height）



树的递归属性 Recursive Tree

类似链表结构，树也有天然的递归属性。
树的每个节点都可以看作以其为根的子树。

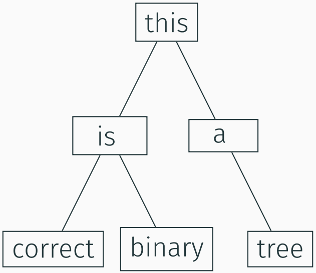


二叉搜索树 BST

二叉树 Binary Tree

在树的基础上，增加如下约定就得到了**二叉树**（binary tree）：

- 每个节点最多只能有两个子节点
- 子节点要么是左节点（left child），要么是右节点（right child）



二叉搜索树 BST

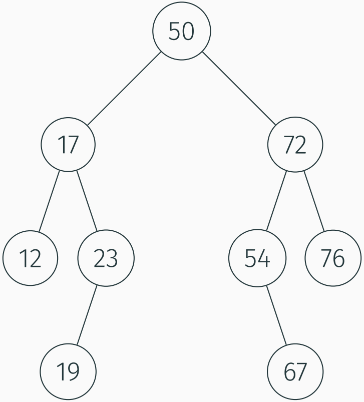
二叉树的这种几何关系，有助于构造一种有序的特殊结构，称为**二叉搜索树** (binary search tree):

- 每个节点包含一个称为键（key）的值
- 每个键值都是唯一的
- 节点的键值满足：left child < root node < right child

尝试根据规则，创建如下键值构造的 BST 树：

`{50, 17, 72, 12, 76, 54, 23, 67, 19}`

二叉搜索树 BST



二叉搜索树动机

- 二叉搜索树的特性适用于二分查找算法
- 基于此结构存储的元素集，可以实现对数级插入、删除和查找
- 元素的遍历访问仍然是 O(N)，但提供了有序访问的特性
- 有利于实现 Map 和 Set 这样的容器

BST 实现

节点的表示

为了表示树的结构，在链表节点的基础上，增加了一个指针用于区分左右节点：

```
struct Node {
    value_type key;
    Node* left;
    Node* right;
};
```

BST 类

```cpp
class BST {
public:
    using value_type = int;
    BST();
    ~BST();
    void add(value_type key);          // O(log N)
    bool containsKey(value_type key);  // O(log N)
    void clear();                      // O(N)
    void print();                      // O(N)
    void remove(value_type key);       // O(log N)
private:
    Node* root;
}
```

BST 类: add

由于子节点可以看作是以当前节点为根节点的子树，所以该过程符合递归范式：

- 如果当前 tree 为 nullptr，则直接添加为 root 节点
- 如果不是 root 节点，则根据键值大小判断插入左树 tree->left 还是右树 tree->right

需要注意的是，辅助函数需要使用引用传递，否则无法修改参数；另外，如果键值等于当前节点，则不作处理。

BST 类: add

```
void BST::insertNode(Node*& tree, value_type key) {
    if (tree == nullptr) {
        tree = new Node(key);
    } else {
        if (key < tree->key) {
            insertNode(tree->left, key);
        } else if (key > tree->key) {
            insertNode(tree->right, key);
        }
    }
}
void BST::add(value_type key) {
    insertNode(root, key);
}
```

清空操作也可以使用递归处理:

- 如果 tree 为 nullptr，则不作处理
- 否则，先分别递归清空左树 tree->left 和右树 tree->right，最后删除当前节点

需要注意的是，delete 操作后记得将 root 指针设置为 nullptr 以防二次访问。

BST 类: clear

```cpp
void BST::freeTree(Node* tree) {
    if (tree != nullptr) {
        freeTree(tree->left);
        freeTree(tree->right);
        delete tree;
    }
}
void BST::clear() {
    freeTree(root);
    root = nullptr;
}
```

BST 类: containsKey

查询操作过程也同样适用于递归：

- 如果 tree 为 nullptr，返回 false
- 如果当前 node 就是所查询的 key，返回 true
- 递归处理左树 tree->left 和右数 tree->right

BST 类: containsKey

```cpp
bool BST::findNode(Node* tree, value_type key) {
    if (tree == nullptr)
        return false;
    if (key == tree->key)
        return tree;
    if (key < tree->key) {
        return findNode(tree->left, key);
    } else {
        return findNode(tree->right, key);
    }
}
bool BST::containsKey(value_type key) {
    return findNode(root, key);
}
```

BST 类: print

遍历操作和清空非常相似，但处理节点的顺序形成了三种不同的遍历方式：

先序遍历 Pre-order 先处理父节点，再依次处理左子节点、右子节点（中-左-右）

中序遍历 In-order 先处理左子节点，再处理父节点，最后处理右子节点（左-中-右）

后序遍历 Post-order 先处理左子节点，再处理右子节点，最后处理父节点（左-右-中）

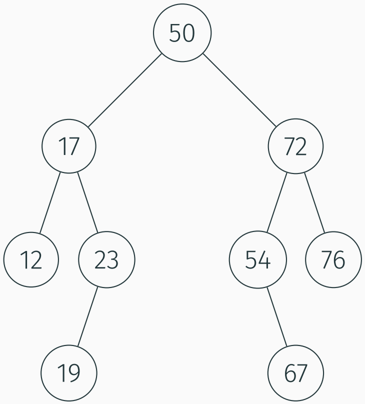
BST 类: print

```
void BST::listTree(Node* tree) {
    if (tree != nullptr) {
        listTree(tree->left);
        std::cout << tree->key << std::endl;
        listTree(tree->right);
    }
}
void BST::print() {
    listTree(root);
}
```

BST 类: remove

删除叶子节点相对简单，例如 12，只需要使用 nullptr 替换该节点。

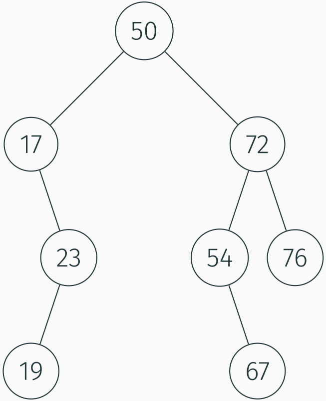
```
Node* oldNode = tree;
tree = nullptr;
delete oldNode;
```



BST 类: remove

删除叶子节点相对简单，例如 12，只需要使用 nullptr 替换该节点。

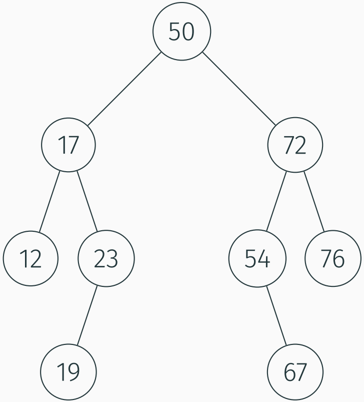
```
Node* oldNode = tree;
tree = nullptr;
delete oldNode;
```



BST 类: remove

删除仅包含一个子节点的父节点也比较简单，例如 23 或 54，只需要使用非空子节点替换该节点即可。

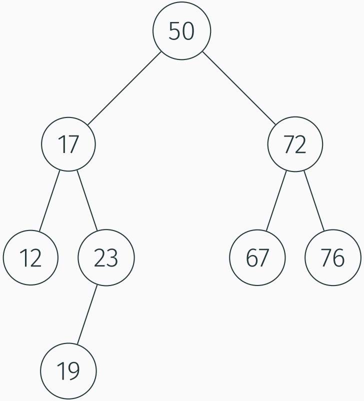
```
Node* oldNode = tree;
if (tree->left != nullptr)
    tree = tree->left;
else
    tree = tree->right;
delete oldNode;
```



BST 类: remove

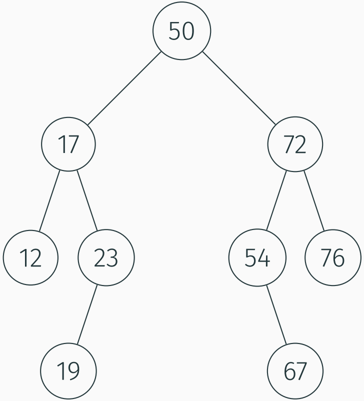
删除仅包含一个子节点的父节点也比较简单，例如 23 或 54，只需要使用非空子节点替换该节点即可。

```
Node* oldNode = tree;
if (tree->left != nullptr)
    tree = tree->left;
else
    tree = tree->right;
delete oldNode;
```



BST 类: remove

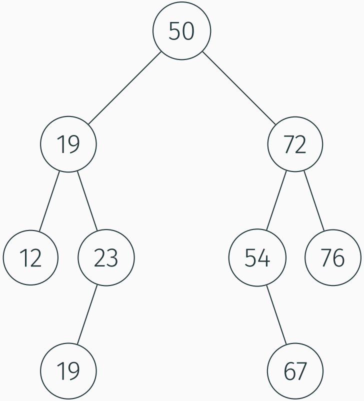
删除既有左树又有右树的节点相对复杂，这里采用的策略是使用右树最小节点替换当前节点。



BST 类: remove

这时，右树的最小节点发生冗余。从递归的角度看，接下来的任务是删除右树最小节点。

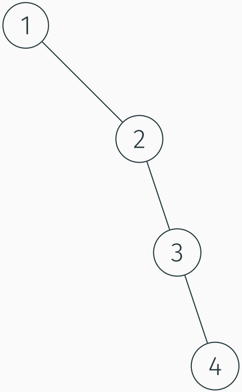
```
tree->key = findMin(tree->right)->key;
removeNode(tree->right, tree->key);
```



One more thing: 平衡二叉树

如果按顺序插入元素，最后形成的树很像一个链表，效率将会退化和链表一致。
真正能达到理想效率的是平衡的二叉树结构，常见的平衡算法有：

- AVL 树
- Red/Black 树
- ……



如何用树实现映射和集合?