

二叉树 (C与C++)

什么是二叉树 (Binary Tree)?

想象一下一个家族树，但每个“人”（我们称之为**节点 Node**）最多只能有两个孩子。这就是二叉树的基本思想。

- **节点 (Node)**: 树的基本组成单元。每个节点包含：
 - **数据 (Data)**: 节点存储的信息。在你的代码中，这分为key(键)和data(值)。键通常用于比较和组织树的结构。
 - **左子节点指针 (Left Child Pointer)**: 指向该节点的左边孩子。如果没有左孩子，则为NULL。
 - **右子节点指针 (Right Child Pointer)**: 指向该节点的右边孩子。如果没有右孩子，则为NULL。
- **根节点 (Root)**: 树的最顶端的节点，它没有父节点。
- **叶节点 (Leaf Node)**: 没有子节点的节点。
- **父节点 (Parent Node)**: 一个节点的直接上级节点。
- **子节点 (Child Node)**: 一个节点的直接下级节点。

遍历方式

二叉树的常见遍历方式包括：

- **前序遍历**（根→左→右）
- **中序遍历**（左→根→右）
- **后序遍历**（左→右→根）
- **层序遍历**（按层次从左到右）

特殊的二叉树：二叉搜索树 (Binary Search Tree - BST)

二叉搜索树有以下重要特性，使得查找、插入和删除操作非常高效：

1. 对于树中的任意一个节点：

- 其左子树中所有节点的键 (key) 都小于该节点的键。
- 其右子树中所有节点的键 (key) 都大于该节点的键。

2. 它的左右子树也分别是二叉搜索树。

这个特性意味着，当你要查找一个值时，你可以从根节点开始，通过比较目标值和当前节点的键，来决定是向左走还是向右走，从而快速定位。

Key和Data之间是有区别的：

概念	作用	是否必需
key	用于维持BST的结构（左子树<key，右子树>key）	必须
data	存储与key关联的实际数据	并非

BST中的增删改查(C语言实现)

1. 节点结构定义

```
1 typedef struct Node {
2     int data;
3     struct Node* left;
4     struct Node* right;
5 } Node;
```

2. 创建新节点

```
1 Node* createNode(int data) {
2     Node* newNode = (Node*)malloc(sizeof(Node));
3     newNode->data = data;
4     newNode->left = newNode->right = NULL;
5     return newNode;
6 }
```

3. 插入操作

```

1 void insert(Node** root, int data) {
2     if (*root == NULL) {
3         *root = createNode(data);
4         return;
5     }
6     if (data < (*root)->data)
7         insert(&(*root)->left, data);
8     else if (data > (*root)->data)
9         insert(&(*root)->right, data);
10    // 重复值不插入
11 }

```

4. 查找操作

```

1 Node* search(Node* root, int key) {
2     if (root == NULL || root->data == key){return root;}
3     if (key < root->data){return search(root->left, key);}
4     else{return search(root->right, key);}
5 }

```

5. 删除操作

```

1 // 找最小节点（用于删除）
2 Node* findMin(Node* root) {
3     while (root->left != NULL){root = root->left;}
4     return root;
5 }
6 Node* deleteNode(Node* root, int key) {
7     if (root == NULL){return NULL;}
8     if (key < root->data)
9         root->left = deleteNode(root->left, key);
10    else if (key > root->data)
11        root->right = deleteNode(root->right, key);
12    else {
13        if (root->left == NULL) {
14            Node* temp = root->right;
15            free(root);
16            return temp; // 父节点的right指针需要指向temp
17        } else if (root->right == NULL) {
18            Node* temp = root->left;
19            free(root);

```

```

20         return temp; // 父节点的left指针需要指向temp
21     }
22     // 双子节点：找右子树的最小值替换
23     Node* temp = findMin(root->right);
24     root->data = temp->data;
25     root->right = deleteNode(root->right, temp->data);
26 }
27 return root;
28 }

```

。关于删除双子节点，用以下图示可以理解：

```

1  原始树：
2      10
3     /  \
4    5   15
5   / \   \
6  2  7   20
7  要删除节点 10：
8  1. 找到右子树中的最小值节点：15
9  2. 替换节点 10 的值为 15：
10     15
11    /  \
12   5   15
13  / \   \
14 2  7   20
15 3. 删除重复的 15 节点：
16     15
17    /  \
18   5   20
19  /  \
20 2   7

```

BST中的增删改查(C++实现)

1. 节点结构定义

```

1 struct Node {
2     int key;           // 键 (用于排序)
3     Node* left;
4     Node* right;
5     Node(int k) : key(k), left(nullptr), right(nullptr) {}
6 };

```

2. BST类定义

```

1 class BST {
2 private:
3     Node* root;
4
5     // 辅助函数 (递归实现)
6     Node* insert(Node* node, int key);
7     Node* deleteNode(Node* node, int key);
8     Node* findMin(Node* node);
9     Node* search(Node* node, int key);
10    void inorderTraversal(Node* node);
11    void destroy(Node* node);
12
13 public:
14     BST() : root(nullptr) {}
15     ~BST() { destroy(root); } // 析构函数释放内存
16
17     // 接口函数
18     void insert(int key);
19     void remove(int key);
20     void inorder() { inorderTraversal(root); }
21 };

```

3. 插入操作 (递归实现)

```

1 Node* BST::insert(Node* node, int key) {
2     if (node == nullptr){return new Node(key);}
3     if (key < node->key){node->left = insert(node->left, key);}
4     else if (key > node->key){node->right = insert(node->right, key);}
5     // 键已存在时不做任何操作 (保持唯一性)
6     return node;
7 }
8 void BST::insert(int key) {
9     root = insert(root, key); // 函数重载
10 }

```

4. 删除操作 (递归实现)

```

1 Node* BST::deleteNode(Node* node, int key) {
2     if (node == nullptr){return node;}
3     if (key < node->key)
4         node->left = deleteNode(node->left, key);
5     else if (key > node->key)
6         node->right = deleteNode(node->right, key);
7     else {
8         // Case 1 & 2: 无子节点或一个子节点
9         if (!node->left || !node->right) {
10             Node* temp = node->left ? node->left : node->right;
11             delete node;
12             return temp;
13         }
14         // Case 3: 两个子节点 (用右子树最小值替换)
15         else {
16             Node* temp = findMin(node->right);
17             node->key = temp->key;
18             node->right = deleteNode(node->right, temp->key);
19         }
20     }
21     return node;
22 }
23 void BST::remove(int key) {
24     root = deleteNode(root, key);
25 }

```

5. 查找操作 (递归实现)

```
1  bool BST::search(Node* node, int key) {
2      if (!node) return false;
3      if (key == node->key) return true;
4      else if (key < node->key) return search(node->left, key);
5      else return search(node->right, key);
6  }
```