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

```c
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;
```

2. 创建新节点

```c
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}
```

3. 插入操作

```c
void insert(Node** root, int data) {
    if (*root == NULL) {
        *root = createNode(data);
        return;
    }
    if (data < (*root)->data)
        insert(&(*root)->left, data);
    else if (data > (*root)->data)
        insert(&(*root)->right, data);
    // 重复值不插入
}
```

4. 查找操作

```c
Node* search(Node* root, int key) {
    if (root == NULL || root->data == key){return root;}
    if (key < root->data){return search(root->left, key);}
    else{return search(root->right, key);}
}
```

5. 删除操作

```c
// 找最小节点（用于删除）
Node* findMin(Node* root) {
    while (root->left != NULL){root = root->left;}
    return root;
}
Node* deleteNode(Node* root, int key) {
    if (root == NULL){return NULL;}
    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
            if (root->left == NULL) {
           Node* temp = root->right;
```

```
            free(root);
            return temp; // 父节点的right指针需要指向temp
        } else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp; // 父节点的left指针需要指向temp
        }
        // 双子节点：找右子树的最小值替换
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

- 关于删除双子节点，用以下图示可以理解：

```
原始树：
       10
      /  \
     5    15
    / \     \
   2   7     20
要删除节点 10：
1. 找到右子树中的最小值节点：15
2. 替换节点 10 的值为 15：
       15
      /  \
     5    15
    / \     \
   2   7     20
3. 删除重复的 15 节点：
       15
      /  \
     5    20
    / \
   2   7
```

BST中的增删改查(C++实现)

1. 节点结构定义

```
struct Node {
    int key;          // 键（用于排序）
    Node* left;
    Node* right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};
```

2. BST类定义

```
class BST {
private:
    Node* root;
```

```
    // 辅助函数（递归实现）
    Node* insert(Node* node, int key);
    Node* deleteNode(Node* node, int key);
    Node* findMin(Node* node);
    Node* search(Node* node, int key);
    void inorderTraversal(Node* node);
    void destroy(Node* node);

public:
    BST() : root(nullptr) {}
    ~BST() { destroy(root); } // 析构函数释放内存

    // 接口函数
    void insert(int key);
    void remove(int key);
    void inorder() { inorderTraversal(root);}
};
```

3. 插入操作（递归实现）

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

4. 删除操作（递归实现）

```
Node* BST::deleteNode(Node* node, int key) {
    if (node == nullptr){return node;}
    if (key < node->key)
        node->left = deleteNode(node->left, key);
    else if (key > node->key)
        node->right = deleteNode(node->right, key);
    else {
        // Case 1 & 2: 无子节点或一个子节点
        if (!node->left || !node->right) {
            Node* temp = node->left ? node->left : node->right;
            delete node;
            return temp;
        }
        // Case 3: 两个子节点（用右子树最小值替换）
        else {
            Node* temp = findMin(node->right);
            node->key = temp->key;
            node->right = deleteNode(node->right, temp->key);
        }
```

```
    }
    return node;
}
void BST::remove(int key) {
    root = deleteNode(root, key);
}
```

5. 查找操作（递归实现）

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