动态绘制数据结构的抽象结构的构建过程

前端设计

前端采用Qt进行设计

代码界面

支持客户直接输入代码或者上传单个文件。保存到后端

图形生成界面

前端界面通过读取map.txt文件解析数据结构的抽象结构，并将其绘制成图形化的形式。
map.txt会参照如下的形式描述

```
10 0 0
1 2
1 3
2 3
2 4
3 4
3 5
4 5
4 6
5 6
5 7
6 7
6 8
7 8
7 9
8 9
8 10
9 10
```

其中，第一行的10表示共有10个节点，0表示为无向图（1表示有向图），第二个0表示边不可重复，后续每一行表示一条边。

前端读取以后生成对应的图形界面，并支持保存为png方式

后端设计

后端通过调用api的方式要求ai生成单步调试的特定函数，要求每执行一步代码就可以生成当前数据结构抽象结构所对应的map.txt文件。
将代码提供给大模型让大模型进行修改。以达到能够生成代码查询函数的目的
其中被注释掉的代码部分是希望大模型生成的部分

```cpp
#include <iostream>
//#include <queue>
//#include <vector>
//#include <unordered_map>
//#include <algorithm>
//#include <fstream>
using namespace std;

// 二叉排序树节点结构
struct BSTNode {
    int data;
    BSTNode* left;
    BSTNode* right;
    BSTNode(int val) : data(val), left(nullptr), right(nullptr) {}
};

// 插入节点函数
BSTNode* insertNode(BSTNode* root, int value) {
    if (root == nullptr) {
        return new BSTNode(value);
    }
    if (value < root->data) {
        root->left = insertNode(root->left, value);
    }
    else if (value > root->data) {
        root->right = insertNode(root->right, value);
    }
    return root;
}

// 查找节点函数
bool searchNode(BSTNode* root, int key) {
    if (root == nullptr) return false;
    if (root->data == key) return true;
    return key < root->data ? searchNode(root->left, key) : searchNode(root->right, key);
}

// 找到子树最小值节点
BSTNode* minValueNode(BSTNode* node) {
```

```
    BSTNode* current = node;
    while (current && current->left != nullptr)
        current = current->left;
    return current;
}

// 删除节点函数
BSTNode* deleteNode(BSTNode* root, int key) {
    if (root == nullptr) return root;

    if (key < root->data) {
        root->left = deleteNode(root->left, key);
    }
    else if (key > root->data) {
        root->right = deleteNode(root->right, key);
    }
    else {
        // 节点只有一个子节点或没有子节点
        if (root->left == nullptr) {
            BSTNode* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            BSTNode* temp = root->left;
            delete root;
            return temp;
        }

        // 节点有两个子节点：找到右子树的最小值节点
        BSTNode* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// 中序遍历打印（验证BST结构）
void inorderTraversal(BSTNode* root) {
    if (root != nullptr) {
```

```
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}
//// 辅助函数：BFS遍历树并收集节点信息和边
//void collectTreeInfo(BSTNode* root, int& nodeCount, vector<pair<int, int>>& edges) {
//    if (!root) return;
//
//    queue<BSTNode*> q;
//    q.push(root);
//
//    while (!q.empty()) {
//        BSTNode* current = q.front();
//        q.pop();
//        nodeCount++;
//
//        if (current->left) {
//            edges.push_back({ current->data, current->left->data });
//            q.push(current->left);
//        }
//        if (current->right) {
//            edges.push_back({ current->data, current->right->data });
//            q.push(current->right);
//        }
//    }
//}
//
//// 将二叉排序树输出为指定图格式
//void printTreeAsGraph(BSTNode* root) {
//    int nodeCount = 0;
//    vector<pair<int, int>> edges;
//
//    // 收集树的信息
//    collectTreeInfo(root, nodeCount, edges);
//    ofstream file("map.txt");
//    if (!file.is_open()) {
//        cerr << "无法打开文件" << endl;
//        exit(-1);
//    }
```

```
//    // 输出图格式
//    file << nodeCount << " 0 0" << endl;
//    for (auto& edge : edges) {
//        file << edge.first << " " << edge.second << endl;
//    }
//    file.close();
//    system("pause");
//}
// 主函数
int main() {
    BSTNode* root = nullptr;
    int values[] = { 1, 4, 2, 8, 5, 7, 11, 14, 99, 33 };
    int n = sizeof(values) / sizeof(values[0]);

    // 插入所有值
    for (int i = 0; i < n; i++) {
        root = insertNode(root, values[i]);
        //printTreeAsGraph(root);
    }

    cout << "中序遍历结果: ";
    inorderTraversal(root);
    cout << endl;

    // 测试查找功能
    cout << "查找 7: " << (searchNode(root, 7) ? "存在" : "不存在") << endl;
    cout << "查找 99: " << (searchNode(root, 99) ? "存在" : "不存在") << endl;
    cout << "查找 100: " << (searchNode(root, 100) ? "存在" : "不存在") << endl;

    // 测试删除功能
    root = deleteNode(root, 8);  // 删除有两个子节点的节点
    root = deleteNode(root, 1);  // 删除只有一个子节点的节点
    root = deleteNode(root, 14); // 删除叶子节点

    cout << "删除 8, 1, 14 后的中序遍历: ";
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```