基于 R-BTree 实现`map`

课程名称　　数据结构　　成绩评定＿＿＿＿
实验项目名称　基于 R-BTree 实现`map`　指导老师　干晓聪
实验项目编号　11　实验项目类型　设计性　实验地点　数学系机房
学生姓名　郭彦培　学号　2022101149
学院　信息科学技术学院　系　数学系　专业　信息管理与信息系统
实验时间　2024 年 6 月 13 日上午 ~ 2024 年 7 月 13 日中午

1. 实验目的

基于 R-BTree 实现`map`

2. 实验环境

计算机：PC X64

操作系统：Windows + Ubuntu20.0LTS

编程语言：C++：GCC std20

IDE：Visual Studio Code

3. 程序原理

`map`维护一个$\mathbb{S} \to \mathbb{R}$的映射，对于$\mathbb{R}$的规模$n$保证：

总体空间复杂度$\mathbb{O}(\log_2 n)$，访问时间复杂度$\mathbb{O}(\log_2 n)$，插入删除复杂度$\mathbb{O}(\log_2 n)$。

在代码中，每个映射体现为`std::pair<KEY_TYPE,VALUE_TYPE>`

在`RB_Tree`的基础上，在每个节点上添加数据项`VALUE`，并且对应的修改泛型系统。

额外实现一个基于节点推断后续节点的函数，以提供对迭代器的支持，具体实现见代码。

4. 程序代码

4.1. `map.hpp`

```cpp
#ifndef RBTREE_MAP_HPP
#define RBTREE_MAP_HPP

#ifdef __PRIVATE_DEBUGE
#include <iostream>
#endif

#include <vector>
#include <stdlib.h>
#include "Dev\02\myVector.h"

using std::vector;


namespace myDS
{
    template <typename KEY,typename VALUE>
    class map{
    private:
        // using int size_t;
        enum COLOR {RED,BLACK};

    protected:
        //节点类
        class Node{
        public:
            KEY key;
            VALUE value;
            COLOR color;
            Node *leftSubTree,  //左子树根节点指针
                 *rightSubTree, //右子树根节点指针
                 *parent;       //父节点指针

            explicit Node() : //构造函数
                key(KEY()),
                color(COLOR::RED),
                leftSubTree(nullptr),
                rightSubTree(nullptr),
                parent(nullptr),
                value(VALUE()) { };

            //获取父节点指针
            inline Node * getParent() {
                return parent;
            }

            //获取祖父节点指针
            inline Node * getGrandParent() {
                if(parent == nullptr) return nullptr;
                else return parent->parent;
            }

            //获取叔叔节点指针
            inline Node * getUncle() {
                Node* __gp = this->getGrandParent();
                if(__gp == nullptr) return nullptr;
                else if(parent == __gp->rightSubTree) return __gp->leftSubTree;
                else return __gp->rightSubTree;
            }

            //获取兄弟节点指针
            inline Node * getSibling(){
                if(parent == nullptr) return nullptr;
                else if(parent->leftSubTree == this) return parent->rightSubTree;
                else return parent->leftSubTree;
            }

        };

        class iterator{
        friend map;
        protected:
            Node * ptr;
            Node * NIL;

            void loop2Begin() {
                if(ptr == NIL){ptr = nullptr;return;}
                while(ptr->leftSubTree != NIL) ptr = ptr->leftSubTree;
            }

            void loop2End() {
                if(ptr == NIL){ptr = nullptr;return;}
                if(ptr->parent == nullptr){ ptr = nullptr;return;}
                while(ptr->parent->leftSubTree != ptr) {
                    ptr = ptr->parent;
                    if(ptr->parent == nullptr){ ptr = nullptr;return;}
                }
                ptr = ptr->parent;
            }

            void getNextNode() {
                if(ptr->rightSubTree != NIL){
                    ptr = ptr->rightSubTree;
                    loop2Begin();
                } else {
                    loop2End();
                }
            }

        public:
            iterator(Node * _ptr,Node * _NIL) {
                ptr = _ptr;
                NIL = _NIL;
            }

            const std::pair<KEY,VALUE> operator*()
            {
                return {ptr->key,ptr->value};
            }

            KEY *operator->() //?
            {
                return ptr;
            }

             myDS::map<KEY,VALUE>::iterator operator++() {
                auto old = *this;
                getNextNode();
                return old;
            }

             myDS::map<KEY,VALUE>::iterator operator++(int) {
                getNextNode();
                return (*this);
            }

            bool operator==( myDS::map<KEY,VALUE>::iterator _b) {
                return ptr == _b.ptr;
            }

            bool operator!=( myDS::map<KEY,VALUE>::iterator _b) {
                return ptr != _b.ptr;
            }
        };

        //树结构
        Node *root, *NIL;

    public:

        map() {
            NIL = new Node();
            NIL->color = COLOR::BLACK;
            root = nullptr;
        };


        ~map(){
            auto DeleteSubTree = [&](auto self,Node *p) -> void{
                if(p == nullptr || p == NIL) return;
                self(self,p->leftSubTree);
                self(self,p->rightSubTree);
                delete p;
                return;
            };
            if(!(root == nullptr)) DeleteSubTree(DeleteSubTree,root);
            delete NIL;
        }

        void insert(KEY key) {
            if(root == nullptr) {
                root = new Node();
                root->color = COLOR::BLACK;
                root->leftSubTree = NIL;
                root->rightSubTree = NIL;
                root->key = key;
            } else {
                if(this->locate(key,root)) return;
                subInsert(root,key);
            }
        }

        void insert(KEY key,VALUE value) {
            if(root == nullptr) {
                root = new Node();
                root->color = COLOR::BLACK;
                root->leftSubTree = NIL;
                root->rightSubTree = NIL;
                root->key = key;
                root->value  = value;
            } else {
                if(this->locate(key,root)) return;
                insert(key);
                (*this)[key] = value;
            }
        }

         myDS::map<KEY,VALUE>::iterator find(KEY tar) {
            if(this->locate(tar,root) != nullptr) return myDS::map<KEY,VALUE>::iterator(this->locate(tar,root));
            else return this->end();
        }

        bool erase(KEY key) {
            return subDelete(root,key);
        }

         myDS::map<KEY,VALUE>::iterator begin(){
            auto rt = iterator(root,NIL);
            rt.loop2Begin();
            return rt;
        }

         myDS::map<KEY,VALUE>::iterator end(){
            return iterator(nullptr,NIL);
        }

        VALUE & operator[] (std::size_t key) {
            if(root == nullptr) {
                root = new Node();
                root->color = COLOR::BLACK;
                root->leftSubTree = NIL;
                root->rightSubTree = NIL;
                root->key = key;
                return locate(key,root)->value;
            } else {
                if(locate(key,root) == nullptr) this->insert(key);
                return locate(key,root)->value;
            }
        }

#ifdef __PRIVATE_DEBUGE
        void printDfsOrder()
        {
            auto dfs = [&](auto self,Node * p ) -> void {
                if(p == nullptr){ std::cout << "ED\n";return;}
                if(p->leftSubTree == nullptr && p->rightSubTree == nullptr) {std::cout << "[NIL] \n";return;}
                std::cout << "["<< p->key << " : " << p->value << "] ";
                self(self,p->leftSubTree);
                self(self,p->rightSubTree);
                return;
            };
            dfs(dfs,root);
        }

        vector<int> printList;
        void printIterOrder()
        {
            auto dfs = [&](auto self,Node * p) -> void{
                if(p->leftSubTree == nullptr && p->rightSubTree == nullptr) {std::cout << "[NIL] \n";return;}
                self(self,p->leftSubTree);
                std::cout << "["<< p->key << " : " << p->value << "] ";
                self(self,p->rightSubTree);
            };
            dfs(dfs,root);
        }
#endif
    private:
        Node * locate(KEY t,Node * p) {
            if(p == NIL) return nullptr;
            else if(p->key == t) return p;
            else if(p->key > t) return locate(t,p->leftSubTree);
            else return locate(t,p->rightSubTree);
        }

        //右旋某个节点
        void rotateRight(Node *p)
        {
            Node * _gp = p->getGrandParent();
            Node * _pa = p->getParent();
            Node * _rotY = p->rightSubTree;
            _pa->leftSubTree = _rotY;
            if(_rotY != NIL) _rotY->parent = _pa;
            p->rightSubTree = _pa;
            _pa->parent = p;

            if(root == _pa) root = p;
            p->parent = _gp;
            if(_gp != nullptr) if(_gp->leftSubTree == _pa) _gp->leftSubTree = p;
                else _gp->rightSubTree = p;
            return;
        }

        //左旋某个节点
        void rotateLeft(Node *p)
        {
            if(p->parent == nullptr){
                root = p;
                return;
            }
            Node *_gp = p->getGrandParent();
            Node *_pa = p->parent;
            Node *_rotX = p->leftSubTree;

#ifdef __DETIL_DEBUG_OUTPUT
            printIterOrder();
#endif
            _pa->rightSubTree = _rotX;

#ifdef __DETIL_DEBUG_OUTPUT
            printIterOrder();
#endif

            if(_rotX != NIL)
                _rotX->parent = _pa;

#ifdef __DETIL_DEBUG_OUTPUT
            printIterOrder();
```

```
#endif
            p->leftSubTree = _pa;
#ifdef __DETIL_DEBUG_OUTPUT
            printIterOrder();
#endif
            _pa->parent = p;
#ifdef __DETIL_DEBUG_OUTPUT
            printIterOrder();
#endif
            if(root == _pa)
                root = p;
            p->parent = _gp;

#ifdef __DETIL_DEBUG_OUTPUT
            printIterOrder();
#endif
            if(_gp != nullptr){
                if(_gp->leftSubTree == _pa)
                    _gp->leftSubTree = p;
                else
                    _gp->rightSubTree = p; //?!
            }
#ifdef __DETIL_DEBUG_OUTPUT
            printIterOrder();
#endif
        }

        //插入节点递归部分
        void subInsert(Node *p,KEY key)
        {
            if(p->key >= key){ //1 2
                if(p->leftSubTree != NIL) //3
                    subInsert(p->leftSubTree, key);
                else {
                    Node *tmp = new Node();//3
                    tmp->key = key;
                    tmp->leftSubTree = tmp->rightSubTree = NIL;
                    tmp->parent = p;
                    p->leftSubTree = tmp;
                    resetStatus_forInsert(tmp);
                }
            } else {
                if(p->rightSubTree != NIL) //1 2
                    subInsert(p->rightSubTree, key);
                else {
                    Node *tmp = new Node();
                    tmp->key = key;
                    tmp->leftSubTree = tmp->rightSubTree = NIL;
                    tmp->parent = p;
                    p->rightSubTree = tmp;
                    resetStatus_forInsert(tmp);
                }
            }
        }

        //插入后的平衡维护
        void resetStatus_forInsert(Node *p) {
            //case 1:
            if(p->parent == nullptr){
                root = p;
                p->color = COLOR::BLACK;
                return;
            }
            //case 2-6:
            if(p->parent->color == COLOR::RED){
                //case 2: pass
                if(p->getUncle()->color == COLOR::RED) {
                    p->parent->color = p->getUncle()->color = COLOR::BLACK;
                    p->getGrandParent()->color = COLOR::RED;
                    resetStatus_forInsert(p->getGrandParent());
                } else {
                    if(p->parent->rightSubTree == p && p->getGrandParent()->leftSubTree == p->parent) {
                        //case 3:
                        rotateLeft(p);
                        p->color = COLOR::BLACK;
                        p->parent->color = COLOR::RED;
                        rotateRight(p);
                    } else if(p->parent->leftSubTree == p && p->getGrandParent()->rightSubTree == p->parent) {  //this
                        //case 4:
                        rotateRight(p);
                        p->color = COLOR::BLACK;
                        p->parent->color = COLOR::RED;
                        rotateLeft(p);
                    } else if(p->parent->leftSubTree == p && p->getGrandParent()->leftSubTree == p->parent) {
                        //case 5:
                        p->parent->color = COLOR::BLACK;
                        p->getGrandParent()->color = COLOR::RED;
                        rotateRight(p->parent);
                    } else if(p->parent->rightSubTree == p && p->getGrandParent()->rightSubTree == p->parent) {
                        //case 6: BUG HERE
                        p->parent->color = COLOR::BLACK;
                        p->getGrandParent()->color = COLOR::RED;
                        rotateLeft(p->parent);
                    }
                }
            }
        }

        //删除时的递归部分
        bool subDelete(Node *p, KEY key){

            //获取最接近叶节点的儿子
            auto getLowwestChild = [&](auto self,Node *p) -> Node*{
                if(p->leftSubTree == NIL) return p;
                return self(self,p->leftSubTree);
            };

            if(p->key > key){
                if(p->leftSubTree == NIL){
                    return false;
                }
                return subDelete(p->leftSubTree, key);
            } else if(p->key < key){
                if(p->rightSubTree == NIL){
                    return false;
                }
                return subDelete(p->rightSubTree, key);
            } else if(p->key == key){
                if(p->rightSubTree == NIL){
                    deleteChild(p);
                    return true;
                }
                Node *smallChild = getLowwestChild(getLowwestChild,p->rightSubTree);
                std::swap(p->key, smallChild->key);
                std::swap(p->value,smallChild->value);
                deleteChild(smallChild);

                return true;
            }else{
            return false;
            }
        }

    //     //删除入口
    //     bool deleteChild(Node *p, int key){
    //     if(p->value > key){
    //         if(p->leftSubTree == NIL){
    //             return false;
    //         }
    //         return deleteChild(p->leftSubTree, key);
    //     } else if(p->value < key){
    //         if(p->rightSubTree == NIL){
    //             return false;
    //         }
    //         return deleteChild(p->rightSubTree, key);
    //     } else if(p->value == key){
```

```
    //         if(p->rightSubTree == NIL){
    //             delete_one_child (p);
    //             return true;
    //         }
    //         Node *smallest = getSmallestChild(p->rightTree);
    //         swap(p->value, smallest->value);
    //         delete_one_child (smallest);

    //         return true;
    //     }else{
    //         return false;
    //     }
    // }

        //删除处理：删除某个儿子
        void deleteChild(Node *p){
            Node *child = p->leftSubTree == NIL ? p->rightSubTree : p->leftSubTree;
            if(p->parent == nullptr && p->leftSubTree == NIL && p->rightSubTree == NIL){
                p = nullptr;
                root = p;
                return;
            }
            if(p->parent == nullptr){
                delete  p;
                child->parent = nullptr;
                root = child;
                root->color = COLOR::BLACK;
                return;
            }
            if(p->parent->leftSubTree == p) p->parent->leftSubTree = child;
            else p->parent->rightSubTree = child;

            child->parent = p->parent;
            if(p->color == COLOR::BLACK){
                if(child->color == COLOR::RED){
                    child->color = COLOR::BLACK;
                } else
                    resetStatus_forDelete(child);
            }

            delete p;
        }

        //删除后的平衡维护
        void resetStatus_forDelete(Node *p){
            if(p->parent == nullptr){
                //case 0-0:
                p->color = COLOR::BLACK;
                return;
            }
            if(p->getSibling()->color == COLOR::RED) {
                //case 0-1:
                p->parent->color = COLOR::RED;
                p->getSibling()->color = COLOR::BLACK;
                if(p == p->parent->leftSubTree) rotateLeft(p->parent);
                else rotateRight(p->parent);
            }
            if( p->parent->color == COLOR::BLACK &&
                p->getSibling()->color == COLOR::BLACK &&
                p->getSibling()->leftSubTree->color == COLOR::BLACK &&
                p->getSibling()->rightSubTree->color == COLOR::BLACK) {
                //case 1-1:
                p->getSibling()->color = COLOR::RED;
                resetStatus_forDelete(p->parent);
            } else if(p->parent->color == COLOR::RED && p->getSibling()->color == COLOR::BLACK&& p->getSibling()->leftSubTree->color == COLOR::BLACK && p->getSibling()->rightSubTree->color == COLOR::BLACK) {
                //case 1-2:
                p->getSibling()->color = COLOR::RED;
                p->parent->color = COLOR::BLACK;
            } else {
                if(p->getSibling()->color == COLOR::BLACK) {
                    if(p == p->parent->leftSubTree && p->getSibling()->leftSubTree->color == COLOR::RED && p->getSibling()->rightSubTree->color == COLOR::BLACK) {
                        //case 1-3:
                        p->getSibling()->color = COLOR::RED;
                        p->getSibling()->leftSubTree->color = COLOR::BLACK;
                        rotateRight(p->getSibling()->leftSubTree);
                    } else if(p == p->parent->rightSubTree && p->getSibling()->leftSubTree->color == COLOR::BLACK && p->getSibling()->rightSubTree->color == COLOR::RED) {
                        //case 1-4:
                        p->getSibling()->color = COLOR::RED;
                        p->getSibling()->rightSubTree->color = COLOR::BLACK;
                        rotateLeft(p->getSibling()->rightSubTree);
                    }
                }
                p->getSibling()->color = p->parent->color;
                p->parent->color = COLOR::BLACK;
                //case 1-5:
                if(p == p->parent->leftSubTree){
                    //case 0-3
                    p->getSibling()->rightSubTree->color = COLOR::BLACK;
                    rotateLeft(p->getSibling());
                } else {
                    //case 0-4
                    p->getSibling()->leftSubTree->color = COLOR::BLACK;
                    rotateRight(p->getSibling());
                }
            }
        }


    };
} // namespace myDS
#endif
```

4.2. `_PRIV_TEST.cpp`

```
#include <iostream>
#define __PRIVATE_DEBUGE
#include <Dev\11\map.hpp>

using namespace std;

int main()
{
     myDS::map<int,int> testMP;
    while(1)
    {
        cout << ">>>";
        string s;
        cin >> s;
        if(s == "im") {
            int t;
            cin >> t;
            testMP.insert(t);
        } else if(s == "is") {
            int t,v;
            cin >> t >> v;
            testMP.insert(t,v);
        } else if(s == "p") {
            std::cout << "===DFS  Order===\n";
            testMP.printDfsOrder();
            std::cout << "===Iter Order===\n";
            testMP.printIterOrder();
            for(auto x:testMP) cout << "[" <<  x.first << " " << x.second << "] ";
            cout << "\n";
        } else if(s == "d") {
            int t;
            cin >> t;
            cout << testMP.erase(t) << "\n";
        } else if(s == "f") {
            int t;
            cin >> t;
            cout << testMP[t] << "\n";
        } else if(s == "m") {
            int t,v;
            cin >> t >> v;
            testMP[t] = v;
        }
    }
}
```

5. 测试数据与运行结果

运行上述`_PRIV_TEST.cpp`测试代码中的正确性测试模块，得到以下内容：

```
>>>is 1 3
>>>is 2 5
>>>is 3 7
>>>im 4
>>>p
===DFS  Order===
[2 : 5] [1 : 3] [NIL]
[NIL]
[3 : 7] [NIL]
[4 : 0] [NIL]
[NIL]
===Iter Order===
[NIL]
[1 : 3] [NIL]
[2 : 5] [NIL]
[3 : 7] [NIL]
[4 : 0] [NIL]
[1 3] [2 5] [3 7] [4 0]
>>>f 2
5
>>>m 2 1145
>>>p
===DFS  Order===
[2 : 1145] [1 : 3] [NIL]
[NIL]
[3 : 7] [NIL]
[4 : 0] [NIL]
[NIL]
```

```
===Iter Order===
[NIL]
[1 : 3] [NIL]
[2 : 1145] [NIL]
[3 : 7] [NIL]
[4 : 0] [NIL]
[1 3] [2 1145] [3 7] [4 0]
>>>d 2
1
>>>p
===DFS  Order===
[3 : 7] [1 : 3] [NIL]
[NIL]
[4 : 0] [NIL]
[NIL]
===Iter Order===
[NIL]
[1 : 3] [NIL]
[3 : 7] [NIL]
[4 : 0] [NIL]
[1 3] [3 7] [4 0]
```

可以看出，代码运行结果与预期相符，可以认为代码正确性无误。