暨南大学本科实验报告专用纸(附页)

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___于晓聪___
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`

```
#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
```

```
48              inline Node * getGrandParent() {
49                  if(parent == nullptr) return nullptr;
50                  else return parent->parent;
51              }
52
53              //获取叔叔节点指针
54              inline Node * getUncle() {
55                  Node* __gp = this->getGrandParent();
56                  if(__gp == nullptr) return nullptr;
57                  else if(parent == __gp->rightSubTree) return __gp->leftSubTree;
58                  else return __gp->rightSubTree;
59              }
60
61              //获取兄弟节点指针
62              inline Node * getSibling(){
63                  if(parent == nullptr) return nullptr;
64                  else if(parent->leftSubTree == this) return parent->rightSubTree;
65                  else return parent->leftSubTree;
66              }
67
68          };
69
70          class iterator{
71          friend map;
72          protected:
73              Node * ptr;
74              Node * NIL;
75
76              void loop2Begin() {
77                  if(ptr == NIL){ptr = nullptr;return;}
78                  while(ptr->leftSubTree != NIL) ptr = ptr->leftSubTree;
79              }
80
81              void loop2End() {
82                  if(ptr == NIL){ptr = nullptr;return;}
83                  if(ptr->parent == nullptr){ ptr = nullptr;return;}
84                  while(ptr->parent->leftSubTree != ptr) {
85                      ptr = ptr->parent;
86                      if(ptr->parent == nullptr){ ptr = nullptr;return;}
87                  }
88                  ptr = ptr->parent;
89              }
90
91              void getNextNode() {
92                  if(ptr->rightSubTree != NIL){
93                      ptr = ptr->rightSubTree;
94                      loop2Begin();
95                  } else {
96                      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(){
```

```
149             auto DeleteSubTree = [&](auto self,Node *p) -> void{
150                 if(p == nullptr || p == NIL) return;
151                 self(self,p->leftSubTree);
152                 self(self,p->rightSubTree);
153                 delete p;
154                 return;
155             };
156             if(!(root == nullptr)) DeleteSubTree(DeleteSubTree,root);
157             delete NIL;
158         }
159 
160         void insert(KEY key) {
161             if(root == nullptr) {
162                 root = new Node();
163                 root->color = COLOR::BLACK;
164                 root->leftSubTree = NIL;
165                 root->rightSubTree = NIL;
166                 root->key = key;
167             } else {
168                 if(this->locate(key,root)) return;
169                 subInsert(root,key);
170             }
171         }
172 
173         void insert(KEY key,VALUE value) {
174             if(root == nullptr) {
175                 root = new Node();
176                 root->color = COLOR::BLACK;
177                 root->leftSubTree = NIL;
178                 root->rightSubTree = NIL;
179                 root->key = key;
180                 root->value  = value;
181             } else {
182                 if(this->locate(key,root)) return;
183                 insert(key);
184                 (*this)[key] = value;
185             }
186         }
187 
188          myDS::map<KEY,VALUE>::iterator find(KEY tar) {
189             if(this->locate(tar,root) != nullptr) return
myDS::map<KEY,VALUE>::iterator(this->locate(tar,root));
190             else return this->end();
191         }
192 
193         bool erase(KEY key) {
194             return subDelete(root,key);
195         }
196 
197          myDS::map<KEY,VALUE>::iterator begin(){
198             auto rt = iterator(root,NIL);
```

```
199             rt.loop2Begin();
200             return rt;
201         }
202 
203          myDS::map<KEY,VALUE>::iterator end(){
204             return iterator(nullptr,NIL);
205         }
206 
207         VALUE & operator[] (std::size_t key) {
208             if(root == nullptr) {
209                 root = new Node();
210                 root->color = COLOR::BLACK;
211                 root->leftSubTree = NIL;
212                 root->rightSubTree = NIL;
213                 root->key = key;
214                 return locate(key,root)->value;
215             } else {
216                 if(locate(key,root) == nullptr) this->insert(key);
217                 return locate(key,root)->value;
218             }
219         }
220 
221 #ifdef __PRIVATE_DEBUGE
222         void printDfsOrder()
223         {
224             auto dfs = [&](auto self,Node * p ) -> void {
225                 if(p == nullptr){ std::cout << "ED\n";return;}
226                 if(p->leftSubTree == nullptr && p->rightSubTree == nullptr) {std::cout << "[NIL] \n";return;}
227                 std::cout << "["<< p->key << " : " << p->value << "] ";
228                 self(self,p->leftSubTree);
229                 self(self,p->rightSubTree);
230                 return;
231             };
232             dfs(dfs,root);
233         }
234 
235         vector<int> printList;
236         void printIterOrder()
237         {
238             auto dfs = [&](auto self,Node * p) -> void{
239                 if(p->leftSubTree == nullptr && p->rightSubTree == nullptr) {std::cout << "[NIL] \n";return;}
240                 self(self,p->leftSubTree);
241                 std::cout << "["<< p->key << " : " << p->value << "] ";
242                 self(self,p->rightSubTree);
243             };
244             dfs(dfs,root);
245         }
246 #endif
247     private:
```

```
248         Node * locate(KEY t,Node * p) {
249             if(p == NIL) return nullptr;
250             else if(p->key == t) return p;
251             else if(p->key > t) return locate(t,p->leftSubTree);
252             else return locate(t,p->rightSubTree);
253         }
254 
255         //右旋某个节点
256         void rotateRight(Node *p)
257         {
258             Node * _gp = p->getGrandParent();
259             Node * _pa = p->getParent();
260             Node * _rotY = p->rightSubTree;
261             _pa->leftSubTree = _rotY;
262             if(_rotY != NIL) _rotY->parent = _pa;
263             p->rightSubTree = _pa;
264             _pa->parent = p;
265 
266             if(root == _pa) root = p;
267             p->parent = _gp;
268             if(_gp != nullptr) if(_gp->leftSubTree == _pa) _gp->leftSubTree = p;
269                 else _gp->rightSubTree = p;
270             return;
271         }
272 
273         //左旋某个节点
274         void rotateLeft(Node *p)
275         {
276             if(p->parent == nullptr){
277                 root = p;
278                 return;
279             }
280             Node *_gp = p->getGrandParent();
281             Node *_pa = p->parent;
282             Node *_rotX = p->leftSubTree;
283 
284 #ifdef __DETIL_DEBUG_OUTPUT
285             printIterOrder();
286 #endif
287             _pa->rightSubTree = _rotX;
288 
289 #ifdef __DETIL_DEBUG_OUTPUT
290             printIterOrder();
291 #endif
292 
293             if(_rotX != NIL)
294                 _rotX->parent = _pa;
295 
296 #ifdef __DETIL_DEBUG_OUTPUT
297             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);
                    }
                }
            }
```

```
394         }
395 
396         //删除时的递归部分
397         bool subDelete(Node *p, KEY key){
398 
399             //获取最接近叶节点的儿子
400             auto getLowwestChild = [&](auto self,Node *p) -> Node*{
401                 if(p->leftSubTree == NIL) return p;
402                 return self(self,p->leftSubTree);
403             };
404 
405             if(p->key > key){
406                 if(p->leftSubTree == NIL){
407                     return false;
408                 }
409                 return subDelete(p->leftSubTree, key);
410             } else if(p->key < key){
411                 if(p->rightSubTree == NIL){
412                     return false;
413                 }
414                 return subDelete(p->rightSubTree, key);
415             } else if(p->key == key){
416                 if(p->rightSubTree == NIL){
417                     deleteChild(p);
418                     return true;
419                 }
420                 Node *smallChild = getLowwestChild(getLowwestChild,p->rightSubTree);
421                 std::swap(p->key, smallChild->key);
422                 std::swap(p->value,smallChild->value);
423                 deleteChild(smallChild);
424 
425                 return true;
426             }else{
427             return false;
428             }
429         }
430 
431     //      //删除入口
432     //      bool deleteChild(Node *p, int key){
433     //      if(p->value > key){
434     //          if(p->leftSubTree == NIL){
435     //              return false;
436     //          }
437     //          return deleteChild(p->leftSubTree, key);
438     //      } else if(p->value < key){
439     //          if(p->rightSubTree == NIL){
440     //              return false;
441     //          }
442     //          return deleteChild(p->rightSubTree, key);
443     //      } else if(p->value == key){
```

```
444     //          if(p->rightSubTree == NIL){
445     //              delete_one_child (p);
446     //              return true;
447     //          }
448     //          Node *smallest = getSmallestChild(p->rightTree);
449     //          swap(p->value, smallest->value);
450     //          delete_one_child (smallest);
451     //
452     //          return true;
453     //      }else{
454     //         return false;
455     //       }
456     // }
457
458          //删除处理：删除某个儿子
459          void deleteChild(Node *p){
460              Node *child = p->leftSubTree == NIL ? p->rightSubTree : p->leftSubTree;
461              if(p->parent == nullptr && p->leftSubTree == NIL && p->rightSubTree == NIL){
462                  p = nullptr;
463                  root = p;
464                  return;
465              }
466              if(p->parent == nullptr){
467                  delete  p;
468                  child->parent = nullptr;
469                  root = child;
470                  root->color = COLOR::BLACK;
471                  return;
472              }
473              if(p->parent->leftSubTree == p) p->parent->leftSubTree = child;
474              else p->parent->rightSubTree = child;
475
476              child->parent = p->parent;
477              if(p->color == COLOR::BLACK){
478                  if(child->color == COLOR::RED){
479                      child->color = COLOR::BLACK;
480                  } else
481                      resetStatus_forDelete(child);
482              }
483
484              delete p;
485          }
486
487          //删除后的平衡维护
488          void resetStatus_forDelete(Node *p){
489              if(p->parent == nullptr){
490                  //case 0-0:
491                  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。