R-BTree 的基本实现

课程名称 数据结构 成绩评定
实验项目名称 R-BTree 的基本实现 指导老师 干晓聪
实验项目编号 08 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

实现 RB-tree 的基本结构

2. 实验环境

计算机：PC X64

操作系统：Windows + Ubuntu20.0LTS

编程语言：C++：GCC std20

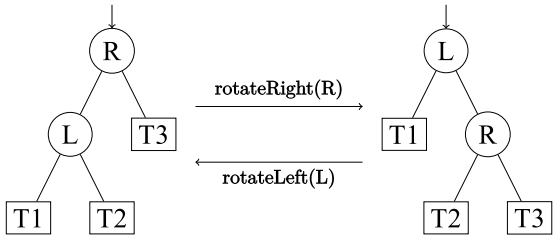
IDE：Visual Studio Code

3. 程序原理

对于一个二叉搜索树，标记所有叶节点为`NIL`，并在路径上标记红黑节点，使得：

1. `NIL`节点为黑色
2. 红色节点的子节点为黑色
3. 从根节点到`NIL`节点路径上的黑色节点数量相同

定义旋转



对于每次插入与删除，需要基于红黑节点性质进行平衡维护。具体实现见代码。

容易证明，满足红黑性质的红黑树，为近似平衡二叉搜索树。

可得插入复杂度为$\mathbb{O}(\log_2 n)$，删除复杂度为$\mathbb{O}(\log_2 n)$，随机访问复杂度为$\mathbb{O}(\log_2 n)$

4. 程序代码

4.1. `memDeleteTest.cpp`

```cpp
#include <iostream>
#include <new>
#include <stdlib.h>
using namespace std;

class testClass{
public:
  int a = 0;
  testClass(){a=1;};
  ~testClass(){cout << "Distroy TestClass\n";};
};
int main()
{
  testClass * arr = new testClass[10];
  cout << "Finish Alloc\n";
  for(int i = 0;i < 10;i ++)
    arr[i].~testClass();
  if(arr)
    //delete[] arr;
    ::operator delete[](arr);
  else cout << "nullPtr\n";
  cout << "Finish Delete\n";
  return 0;
}
```

4.2. `RB_Tree.h`

```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 VALUE_TYPE>
    class RBtree{
    private:
```

```
        // using int size_t;
        enum COLOR {RED,BLACK};

    protected:
        //节点类
        class Node{
        public:
            VALUE_TYPE 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() :
                value(VALUE_TYPE()),
                color(COLOR::RED),
                leftSubTree(nullptr),
                rightSubTree(nullptr),
                parent(nullptr) { };

            //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{
```

```
69          friend RBtree;
70          protected:
71              Node * ptr;
72              Node * NIL;
73
74              void loop2Begin() {
75                  if(ptr == NIL){ptr = nullptr;return;}
76                  while(ptr->leftSubTree != NIL) ptr = ptr->leftSubTree;
77              }
78
79              void loop2End() {
80                  if(ptr == NIL){ptr = nullptr;return;}
81                  if(ptr->parent == nullptr){ ptr = nullptr;return;}
82                  while(ptr->parent->leftSubTree != ptr) {
83                      ptr = ptr->parent;
84                      if(ptr->parent == nullptr){ ptr = nullptr;return;}
85                  }
86                  ptr = ptr->parent;
87              }
88
89              void getNextNode() {
90                  if(ptr->rightSubTree != NIL){
91                      ptr = ptr->rightSubTree;
92                      loop2Begin();
93                  } else {
94                      loop2End();
95                  }
96              }
97
98          public:
99              iterator(Node * _ptr,Node * _NIL) {
100                 ptr = _ptr;
101                 NIL = _NIL;
102             }
103
104             const VALUE_TYPE & operator*()
105             {
106                 return ptr->value;
107             }
108
109             VALUE_TYPE *operator->() //?
110             {
111                 return ptr;
112             }
113
114              myDS::RBtree<VALUE_TYPE>::iterator operator++() {
115                 auto old = *this;
116                 getNextNode();
117                 return old;
118             }
119
120              myDS::RBtree<VALUE_TYPE>::iterator operator++(int) {
```

```
121                     getNextNode();
122                     return (*this);
123                 }
124 
125                 bool operator==( myDS::RBtree<VALUE_TYPE>::iterator _b) {
126                     return ptr == _b.ptr;
127                 }
128 
129                 bool operator!=( myDS::RBtree<VALUE_TYPE>::iterator _b) {
130                     return ptr != _b.ptr;
131                 }
132             };
133 
134         public:
135             //树结构
136             Node *root, *NIL;
137 
138             RBtree() {
139                 NIL = new Node();
140                 NIL->color = COLOR::BLACK;
141                 root = nullptr;
142             };
143 
144             ~RBtree(){
145                 auto DeleteSubTree = [&](auto self,Node *p) -> void{
146                     if(p == nullptr || p == NIL) return;
147                     self(self,p->leftSubTree);
148                     self(self,p->rightSubTree);
149                     delete p;
150                     return;
151                 };
152                 if(!(root == nullptr)) DeleteSubTree(DeleteSubTree,root);
153                 delete NIL;
154             }
155 
156             void insert(VALUE_TYPE data) {
157                 if(root == nullptr) {
158                     root = new Node();
159                     root->color = COLOR::BLACK;
160                     root->leftSubTree = NIL;
161                     root->rightSubTree = NIL;
162                     root->value = data;
163                 } else {
164                     if(this->locate(data,root)) return;
165                     subInsert(root,data);
166                 }
167             }
168 
169             VALUE_TYPE find(VALUE_TYPE tar) {
170                 if(locate(tar,root) != nullptr) return locate(tar,root)-
>value;
```

```
171             else return -1;
172         }
173 
174         bool erase(VALUE_TYPE data) {
175             return subDelete(root,data);
176         }
177 
178          myDS::RBtree<VALUE_TYPE>::iterator begin(){
179             auto rt = iterator(root,NIL);
180             rt.loop2Begin();
181             return rt;
182         }
183 
184          myDS::RBtree<VALUE_TYPE>::iterator end(){
185             return iterator(nullptr,NIL);
186         }
187 
188 #ifdef __PRIVATE_DEBUGE
189         void printDfsOrder()
190         {
191             auto dfs = [&](auto self,Node * p ) -> void {
192                 if(p == nullptr){ std::cout << "ED\n";return;}
193                 if(p->leftSubTree == nullptr && p->rightSubTree == nullptr) {std::cout << "[NIL] \n";return;}
194                 std::cout << "["<< p->value << " : " << (p->color == COLOR::BLACK ?"BLACK":"RED") << "] ";
195                 self(self,p->leftSubTree);
196                 self(self,p->rightSubTree);
197                 return;
198             };
199             dfs(dfs,root);
200         }
201 
202         vector<int> printList;
203         void printIterOrder()
204         {
205             auto dfs = [&](auto self,Node * p) -> void{
206                 if(p->leftSubTree == nullptr && p->rightSubTree == nullptr) {std::cout << "[NIL] \n";return;}
207                 self(self,p->leftSubTree);
208                 std::cout << "["<< p->value << " : " << (p->color == COLOR::BLACK ?"BLACK":"RED") << "] ";
209                 self(self,p->rightSubTree);
210             };
211             dfs(dfs,root);
212         }
213 #endif
214     private:
215         Node * locate(VALUE_TYPE t,Node * p) {
216             if(p == NIL) return nullptr;
217             else if(p->value == t) return p;
```

```
218             else if(p->value > t) return locate(t,p->leftSubTree);
219             else return locate(t,p->rightSubTree);
220         }
221
222         //右旋某个节点
223         void rotateRight(Node *p)
224         {
225             Node * _gp = p->getGrandParent();
226             Node * _pa = p->getParent();
227             Node * _rotY = p->rightSubTree;
228             _pa->leftSubTree = _rotY;
229             if(_rotY != NIL) _rotY->parent = _pa;
230             p->rightSubTree = _pa;
231             _pa->parent = p;
232
233             if(root == _pa) root = p;
234             p->parent = _gp;
235             if(_gp != nullptr) if(_gp->leftSubTree == _pa) _gp->leftSubTree = p;
236                 else _gp->rightSubTree = p;
237             return;
238         }
239
240         //左旋某个节点
241         void rotateLeft(Node *p)
242         {
243             if(p->parent == nullptr){
244                 root = p;
245                 return;
246             }
247             Node *_gp = p->getGrandParent();
248             Node *_pa = p->parent;
249             Node *_rotX = p->leftSubTree;
250
251 #ifdef __DETIL_DEBUG_OUTPUT
252             printIterOrder();
253 #endif
254             _pa->rightSubTree = _rotX;
255
256 #ifdef __DETIL_DEBUG_OUTPUT
257             printIterOrder();
258 #endif
259
260             if(_rotX != NIL)
261                 _rotX->parent = _pa;
262
263 #ifdef __DETIL_DEBUG_OUTPUT
264             printIterOrder();
265 #endif
266             p->leftSubTree = _pa;
267 #ifdef __DETIL_DEBUG_OUTPUT
```

```
268             printIterOrder();
269 #endif
270             _pa->parent = p;
271 #ifdef __DETIL_DEBUG_OUTPUT
272             printIterOrder();
273 #endif
274             if(root == _pa)
275                 root = p;
276             p->parent = _gp;
277 
278 #ifdef __DETIL_DEBUG_OUTPUT
279             printIterOrder();
280 #endif
281             if(_gp != nullptr){
282                 if(_gp->leftSubTree == _pa)
283                     _gp->leftSubTree = p;
284                 else
285                     _gp->rightSubTree = p; //?!
286             }
287 #ifdef __DETIL_DEBUG_OUTPUT
288             printIterOrder();
289 #endif
290         }
291 
292         //插入节点递归部分
293         void subInsert(Node *p,VALUE_TYPE data)
294         {
295             if(p->value >= data){ //1 2
296                 if(p->leftSubTree != NIL) //3
297                     subInsert(p->leftSubTree, data);
298                 else {
299                     Node *tmp = new Node();//3
300                     tmp->value = data;
301                     tmp->leftSubTree = tmp->rightSubTree = NIL;
302                     tmp->parent = p;
303                     p->leftSubTree = tmp;
304                     resetStatus_forInsert(tmp);
305                 }
306             } else {
307                 if(p->rightSubTree != NIL) //1 2
308                     subInsert(p->rightSubTree, data);
309                 else {
310                     Node *tmp = new Node();
311                     tmp->value = data;
312                     tmp->leftSubTree = tmp->rightSubTree = NIL;
313                     tmp->parent = p;
314                     p->rightSubTree = tmp;
315                     resetStatus_forInsert(tmp);
316                 }
317             }
318         }
```

```
319
320         //插入后的平衡维护
321         void resetStatus_forInsert(Node *p) {
322             //case 1:
323             if(p->parent == nullptr){
324                 root = p;
325                 p->color = COLOR::BLACK;
326                 return;
327             }
328             //case 2-6:
329             if(p->parent->color == COLOR::RED){
330                 //case 2: pass
331                 if(p->getUncle()->color == COLOR::RED) {
332                     p->parent->color = p->getUncle()->color = 
COLOR::BLACK;
333                     p->getGrandParent()->color = COLOR::RED;
334                     resetStatus_forInsert(p->getGrandParent());
335                 } else {
336                     if(p->parent->rightSubTree == p && p-
>getGrandParent()->leftSubTree == p->parent) {
337                         //case 3:
338                         rotateLeft(p);
339                         p->color = COLOR::BLACK;
340                         p->parent->color = COLOR::RED;
341                         rotateRight(p);
342                     } else if(p->parent->leftSubTree == p && p-
>getGrandParent()->rightSubTree == p->parent) {  //this
343                         //case 4:
344                         rotateRight(p);
345                         p->color = COLOR::BLACK;
346                         p->parent->color = COLOR::RED;
347                         rotateLeft(p);
348                     } else if(p->parent->leftSubTree == p && p-
>getGrandParent()->leftSubTree == p->parent) {
349                         //case 5:
350                         p->parent->color = COLOR::BLACK;
351                         p->getGrandParent()->color = COLOR::RED;
352                         rotateRight(p->parent);
353                     } else if(p->parent->rightSubTree == p && p-
>getGrandParent()->rightSubTree == p->parent) {
354                         //case 6: BUG HERE
355                         p->parent->color = COLOR::BLACK;
356                         p->getGrandParent()->color = COLOR::RED;
357                         rotateLeft(p->parent);
358                     }
359                 }
360             }
361         }
362
363         //删除时的递归部分
```

```
364         bool subDelete(Node *p, VALUE_TYPE data){
365 
366             //获取最接近叶节点的儿子
367             auto getLowwestChild = [&](auto self,Node *p) -> Node*{
368                 if(p->leftSubTree == NIL) return p;
369                 return self(self,p->leftSubTree);
370             };
371 
372             if(p->value > data){
373                 if(p->leftSubTree == NIL){
374                     return false;
375                 }
376                 return subDelete(p->leftSubTree, data);
377             } else if(p->value < data){
378                 if(p->rightSubTree == NIL){
379                     return false;
380                 }
381                 return subDelete(p->rightSubTree, data);
382             } else if(p->value == data){
383                 if(p->rightSubTree == NIL){
384                     deleteChild(p);
385                     return true;
386                 }
387                 Node *smallChild = getLowwestChild(getLowwestChild,p->rightSubTree);
388                 std::swap(p->value, smallChild->value);
389                 deleteChild(smallChild);
390 
391                 return true;
392             }else{
393             return false;
394             }
395         }
396 
397     //      //删除入口
398     //      bool deleteChild(Node *p, int data){
399     //      if(p->value > data){
400     //          if(p->leftSubTree == NIL){
401     //              return false;
402     //          }
403     //          return deleteChild(p->leftSubTree, data);
404     //      } else if(p->value < data){
405     //          if(p->rightSubTree == NIL){
406     //              return false;
407     //          }
408     //          return deleteChild(p->rightSubTree, data);
409     //      } else if(p->value == data){
410     //          if(p->rightSubTree == NIL){
411     //              delete_one_child (p);
412     //              return true;
413     //          }
```

```
414     //          Node *smallest = getSmallestChild(p->rightTree);
415     //          swap(p->value, smallest->value);
416     //          delete_one_child (smallest);
417     
418     //          return true;
419     //      }else{
420     //         return false;
421     //      }
422     // }
423     
424         //删除处理：删除某个儿子
425         void deleteChild(Node *p){
426             Node *child = p->leftSubTree == NIL ? p->rightSubTree : p->leftSubTree;
427             if(p->parent == nullptr && p->leftSubTree == NIL && p->rightSubTree == NIL){
428                 p = nullptr;
429                 root = p;
430                 return;
431             }
432             if(p->parent == nullptr){
433                 delete  p;
434                 child->parent = nullptr;
435                 root = child;
436                 root->color = COLOR::BLACK;
437                 return;
438             }
439             if(p->parent->leftSubTree == p) p->parent->leftSubTree = child;
440             else p->parent->rightSubTree = child;
441     
442             child->parent = p->parent;
443             if(p->color == COLOR::BLACK){
444                 if(child->color == COLOR::RED){
445                     child->color = COLOR::BLACK;
446                 } else
447                     resetStatus_forDelete(child);
448             }
449     
450             delete p;
451         }
452     
453         //删除后的平衡维护
454         void resetStatus_forDelete(Node *p){
455             if(p->parent == nullptr){
456                 //case 0-0:
457                 p->color = COLOR::BLACK;
458                 return;
459             }
460             if(p->getSibling()->color == COLOR::RED) {
461                 //case 0-1:
```

```
462                 p->parent->color = COLOR::RED;
463                 p->getSibling()->color = COLOR::BLACK;
464                 if(p == p->parent->leftSubTree) rotateLeft(p->parent);
465                 else rotateRight(p->parent);
466             }
467             if( p->parent->color == COLOR::BLACK &&
468                 p->getSibling()->color == COLOR::BLACK &&
469                 p->getSibling()->leftSubTree->color == COLOR::BLACK &&
470                 p->getSibling()->rightSubTree->color == COLOR::BLACK) {
471                 //case 1-1:
472                 p->getSibling()->color = COLOR::RED;
473                 resetStatus_forDelete(p->parent);
474             } else if(p->parent->color == COLOR::RED && p->getSibling()->color == COLOR::BLACK&& p->getSibling()->leftSubTree->color == COLOR::BLACK && p->getSibling()->rightSubTree->color == COLOR::BLACK) {
475                 //case 1-2:
476                 p->getSibling()->color = COLOR::RED;
477                 p->parent->color = COLOR::BLACK;
478             } else {
479                 if(p->getSibling()->color == COLOR::BLACK) {
480                     if(p == p->parent->leftSubTree && p->getSibling()->leftSubTree->color == COLOR::RED && p->getSibling()->rightSubTree->color == COLOR::BLACK) {
481                         //case 1-3:
482                         p->getSibling()->color = COLOR::RED;
483                         p->getSibling()->leftSubTree->color = COLOR::BLACK;
484                         rotateRight(p->getSibling()->leftSubTree);
485                     } else if(p == p->parent->rightSubTree && p->getSibling()->leftSubTree->color == COLOR::BLACK && p->getSibling()->rightSubTree->color == COLOR::RED) {
486                         //case 1-4:
487                         p->getSibling()->color = COLOR::RED;
488                         p->getSibling()->rightSubTree->color = COLOR::BLACK;
489                         rotateLeft(p->getSibling()->rightSubTree);
490                     }
491                 }
492                 p->getSibling()->color = p->parent->color;
493                 p->parent->color = COLOR::BLACK;
494                 //case 1-5:
495                 if(p == p->parent->leftSubTree){
496                     //case 0-3
497                     p->getSibling()->rightSubTree->color = COLOR::BLACK;
498                     rotateLeft(p->getSibling());
499                 } else {
500                     //case 0-4
501                     p->getSibling()->leftSubTree->color = COLOR::BLACK;
502                     rotateRight(p->getSibling());
```

```
                }
            }
        }


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

4.3. `_PRIV_TEST.cpp`

```cpp
// #include <d:\Desktop\Document\Coding\C++\ProjectC\myDS\myVector.h>
// #include "myVector.h"
#define __PRIVATE_DEBUGE
// #define __DETIL_DEBUG_OUTPUT
#include "Dev\08\RB_Tree.h"
#include "Dev\08\eg2.h"
#include <iostream>
#include <vector>

using namespace myDS;

int main()
{
    // testingVector tc;
    int i = 0;
    // bst rbt;
    RBtree<int> rbt;
    while (1)
    {
        // i++;
        char q;
        std::cin >> q;
        switch (q)
        {
        case 'i':
        {
            int t;
            std::cin >> t;
            rbt.insert(t);
        }
            break;
        case 'p':
            std::cout << "===DFS  Order===\n";
            rbt.printDfsOrder();
            std::cout << "===Iter Order===\n";
            rbt.printIterOrder();
            // std::cout << "===Use  Itera===\n";
            // for(auto x:rbt) std::cout << x << " ";
```

```
            // cout << "\n";
            std::cout << "===Use  Bg Ed===\n";
            for(auto x = rbt.begin();x != rbt.end();x ++)
            {
                auto & y = *x;
                std::cout << y << " ";
            }cout << "\n";

            std::cout << "\n";
            // rbt.inorder();
            break;
        case 'd':
            {
                int t;
                std::cin  >> t;
                // rbt.delete_value(t);
                rbt.erase(t);
            }


}
```

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:

```
 i 1
 i 2
 i 3
 i 4
 i 5
 i 6
p
===DFS  Order===
[2 : BLACK] [1 : BLACK] [NIL]
[NIL]
[4 : RED] [3 : BLACK] [NIL]
[NIL]
[5 : BLACK] [NIL]
[6 : RED] [NIL]
[NIL]
===Iter Order===
[NIL]
[1 : BLACK] [NIL]
[2 : BLACK] [NIL]
[3 : BLACK] [NIL]
```

```
[4 : RED] [NIL]
[5 : BLACK] [NIL]
[6 : RED] [NIL]
d 2
p
===DFS  Order===
[3 : BLACK] [1 : BLACK] [NIL]
[NIL]
[5 : RED] [4 : BLACK] [NIL]
[NIL]
[6 : BLACK] [NIL]
[NIL]
===Iter Order===
[NIL]
[1 : BLACK] [NIL]
[3 : BLACK] [NIL]
[4 : BLACK] [NIL]
[5 : RED] [NIL]
[6 : BLACK] [NIL]
d 5
p
===DFS  Order===
[3 : BLACK] [1 : BLACK] [NIL]
[NIL]
[6 : BLACK] [4 : RED] [NIL]
[NIL]
[NIL]
===Iter Order===
[NIL]
[1 : BLACK] [NIL]
[3 : BLACK] [NIL]
[4 : RED] [NIL]
[6 : BLACK] [NIL]
d 3
p
===DFS  Order===
[4 : BLACK] [1 : BLACK] [NIL]
[NIL]
[6 : BLACK] [NIL]
[NIL]
===Iter Order===
[NIL]
[1 : BLACK] [NIL]
[4 : BLACK] [NIL]
```

```
[6 : BLACK] [NIL]
d 2
p
===DFS  Order===
[4 : BLACK] [1 : BLACK] [NIL]
[NIL]
[6 : BLACK] [NIL]
[NIL]
===Iter Order===
[NIL]
[1 : BLACK] [NIL]
[4 : BLACK] [NIL]
[6 : BLACK] [NIL]
i 2
p
===DFS  Order===
[4 : BLACK] [1 : BLACK] [NIL]
[2 : RED] [NIL]
[NIL]
[6 : BLACK] [NIL]
[NIL]
===Iter Order===
[NIL]
[1 : BLACK] [NIL]
[2 : RED] [NIL]
[4 : BLACK] [NIL]
[6 : BLACK] [NIL]


i 1
i 2
i 3
i 5
i 4
p
===DFS  Order===
[2 : BLACK] [1 : BLACK] [NIL]
[NIL]
[4 : BLACK] [3 : RED] [NIL]
[NIL]
[5 : RED] [NIL]
[NIL]
===Iter Order===
[NIL]
```

```
[1 : BLACK] [NIL]
[2 : BLACK] [NIL]
[3 : RED] [NIL]
[4 : BLACK] [NIL]
[5 : RED] [NIL]
===Use  Bg Ed===
1 2 3 4 5
```

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。

运行`_PRIV_TEST.cpp`中的内存测试模块，在保持 CPU 高占用率运行一段时间后内存变化符合预期，可以认为代码内存安全性良好。