7.封装红黑树实现mymap和myset

1. 源码及框架分析

SGI-STL30版本源代码，map和set的源代码在map/set/stl_map.h/stl_set.h/stl_tree.h等几个头文件中。

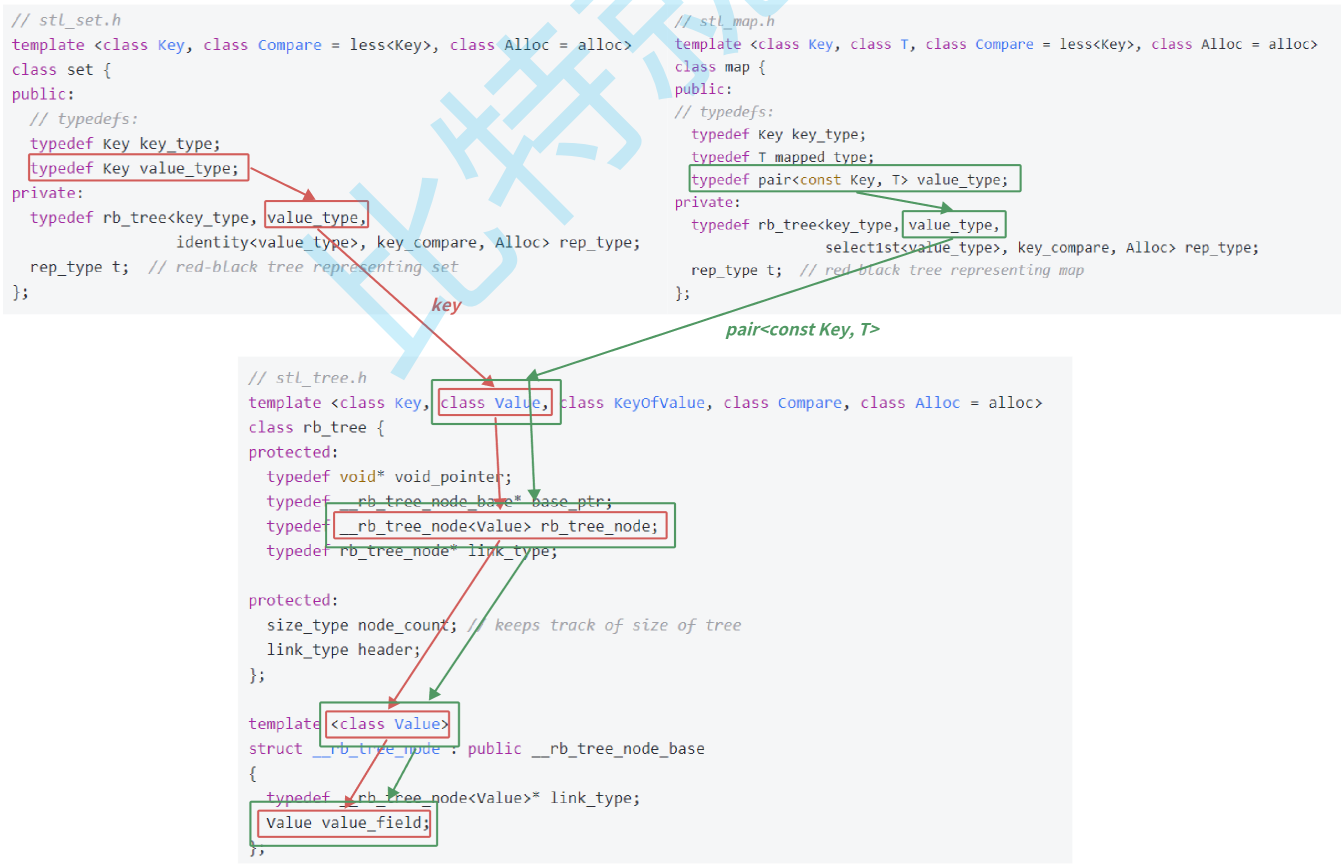
map和set的实现结构框架核心部分截取出来如下：

```
// set
#ifndef __SGI_STL_INTERNAL_TREE_H
#include <stl_tree.h>
#endif
#include <stl_set.h>
#include <stl_multiset.h>

// map
#ifndef __SGI_STL_INTERNAL_TREE_H
#include <stl_tree.h>
#endif
#include <stl_map.h>
#include <stl_multimap.h>

// stl_set.h
template <class Key, class Compare = less<Key>, class Alloc = alloc>
class set {
public:
  // typedefs:
  typedef Key key_type;
  typedef Key value_type;
private:
  typedef rb_tree<key_type, value_type,
                  identity<value_type>, key_compare, Alloc> rep_type;
  rep_type t;  // red-black tree representing set
};

// stl_map.h
template <class Key, class T, class Compare = less<Key>, class Alloc = alloc>
class map {
public:
// typedefs:
  typedef Key key_type;
  typedef T mapped_type;
```

```
35    typedef pair<const Key, T> value_type;
36  private:
37    typedef rb_tree<key_type, value_type,
38                    select1st<value_type>, key_compare, Alloc> rep_type;
39    rep_type t;  // red-black tree representing map
40  };
41
42  // stl_tree.h
43  struct __rb_tree_node_base
44  {
45    typedef __rb_tree_color_type color_type;
46    typedef __rb_tree_node_base* base_ptr;
47
48    color_type color;
49    base_ptr parent;
50    base_ptr left;
51    base_ptr right;
52  };
53
54  // stl_tree.h
55  template <class Key, class Value, class KeyOfValue, class Compare, class Alloc
    = alloc>
56  class rb_tree {
57  protected:
58    typedef void* void_pointer;
59    typedef __rb_tree_node_base* base_ptr;
60    typedef __rb_tree_node<Value> rb_tree_node;
61    typedef rb_tree_node* link_type;
62    typedef Key key_type;
63    typedef Value value_type;
64  public:
65    // insert用的是第二个模板参数左形参
66    pair<iterator,bool> insert_unique(const value_type& x);
67
68    // erase和find用第一个模板参数做形参
69    size_type erase(const key_type& x);
70    iterator find(const key_type& x);
71  protected:
72    size_type node_count; // keeps track of size of tree
73    link_type header;
74  };
75
76  template <class Value>
77  struct __rb_tree_node : public __rb_tree_node_base
78  {
79    typedef __rb_tree_node<Value>* link_type;
80    Value value_field;
```

```
81 };
```

- 通过下图对框架的分析，我们可以看到源码中rb_tree用了一个巧妙的泛型思想实现，rb_tree是实现key的搜索场景，还是key/value的搜索场景不是直接写死的，而是由第二个模板参数Value决定_rb_tree_node中存储的数据类型。
- set实例化rb_tree时第二个模板参数给的是key，map实例化rb_tree时第二个模板参数给的是pair<const key, T>，这样一颗红黑树既可以实现key搜索场景的set，也可以实现key/value搜索场景的map。
- 要注意一下，源码里面模板参数是用T代表value，而内部写的value_type不是我们我们日常key/value场景中说的value，源码中的value_type反而是红黑树结点中存储的真实的数据的类型。
- rb_tree第二个模板参数Value已经控制了红黑树结点中存储的数据类型，为什么还要传第一个模板参数Key呢？尤其是set，两个模板参数是一样的，这是很多同学这时的一个疑问。要注意的是对于map和set，find/erase时的函数参数都是Key，所以第一个模板参数是传给find/erase等函数做形参的类型的。对于set而言两个参数是一样的，但是对于map而言就完全不一样了，map insert的是pair对象，但是find和ease的是Key对象。
- 吐槽一下，这里源码命名风格比较乱，set模板参数用的Key命名，map用的是Key和T命名，而rb_tree用的又是Key和Value，可见大佬有时写代码也不规范，乱弹琴。



2. 模拟实现map和set

2.1 实现出复用红黑树的框架，并支持insert

- 参考源码框架，map和set复用之前我们实现的红黑树。
- 我们这里相比源码调整一下，key参数就用K，value参数就用V，红黑树中的数据类型，我们使用T。
- 其次因为RBTree实现了泛型不知道T参数导致是K，还是pair<K, V>，那么insert内部进行插入逻辑比较时，就没办法进行比较，因为pair的默认支持的是key和value一起参与比较，我们需要时的任何时候只比较key，所以我们在map和set层分别实现一个MapKeyOfT和SetKeyOfT的仿函数传给RBTree的KeyOfT，然后RBTree中通过KeyOfT仿函数取出T类型对象中的key，再进行比较，具体细节参考如下代码实现。

```
// 源码中pair支持的<重载实现
template <class T1, class T2>
bool operator<  (const pair<T1,T2>& lhs, const pair<T1,T2>& rhs)
{ return lhs.first<rhs.first || (!(rhs.first<lhs.first) &&
lhs.second<rhs.second); }

// Mymap.h
namespace bit
{
    template<class K, class V>
    class map
    {
        struct MapKeyOfT
        {
            const K& operator()(const pair<K, V>& kv)
            {
                return kv.first;
            }
        };
    public:
        bool insert(const pair<K, V>& kv)
        {
             return _t.Insert(kv);
        }
    private:
        RBTree<K, pair<K, V>, MapKeyOfT> _t;
    };
}

// Myset.h
namespace bit
{
    template<class K>
```

```cpp
    class set
    {
        struct SetKeyOfT
        {
            const K& operator()(const K& key)
            {
                    return key;
            }
        };
    public:
        bool insert(const K& key)
        {
            return _t.Insert(key);
        }
    private:
            RBTree<K, K, SetKeyOfT> _t;
    };
}

// RBTree.h
enum Colour
{
    RED,
    BLACK
};

template<class T>
struct RBTreeNode
{
    T _data;

    RBTreeNode<T>* _left;
    RBTreeNode<T>* _right;
    RBTreeNode<T>* _parent;
    Colour _col;

    RBTreeNode(const T& data)
        : _data(data)
        , _left(nullptr)
        , _right(nullptr)
        , _parent(nullptr)
    {}
};

// 实现步骤:
//  1、实现红黑树
//  2、封装map和set框架，解决KeyOfT
```

```cpp
// 3、iterator
// 4、const_iterator
// 5、key不支持修改的问题
// 6、operator[]
template<class K, class T, class KeyOfT>
class RBTree
{
private:
    typedef RBTreeNode<T> Node;
    Node* _root = nullptr;

public:
    bool Insert(const T& data)
    {
        if (_root == nullptr)
        {
                _root = new Node(data);
                _root->_col = BLACK;
                return true;
        }

        KeyOfT kot;
        Node* parent = nullptr;
        Node* cur = _root;
        while (cur)
        {
                if (kot(cur->_data) < kot(data))
                {
                        parent = cur;
                        cur = cur->_right;
                }
                else if (kot(cur->_data) > kot(data))
                {
                        parent = cur;
                        cur = cur->_left;
                }
                else
                {
                        return false;
                }
        }

        cur = new Node(data);
        Node* newnode = cur;

        // 新增结点。颜色给红色
        cur->_col = RED;
```

```
        if (kot(parent->_data) < kot(data))
        {
                parent->_right = cur;
        }
        else
        {
                parent->_left = cur;
        }
        cur->_parent = parent;

        //...

        return true;
    }
}
```

2.2 支持iterator的实现

iterator核心源代码

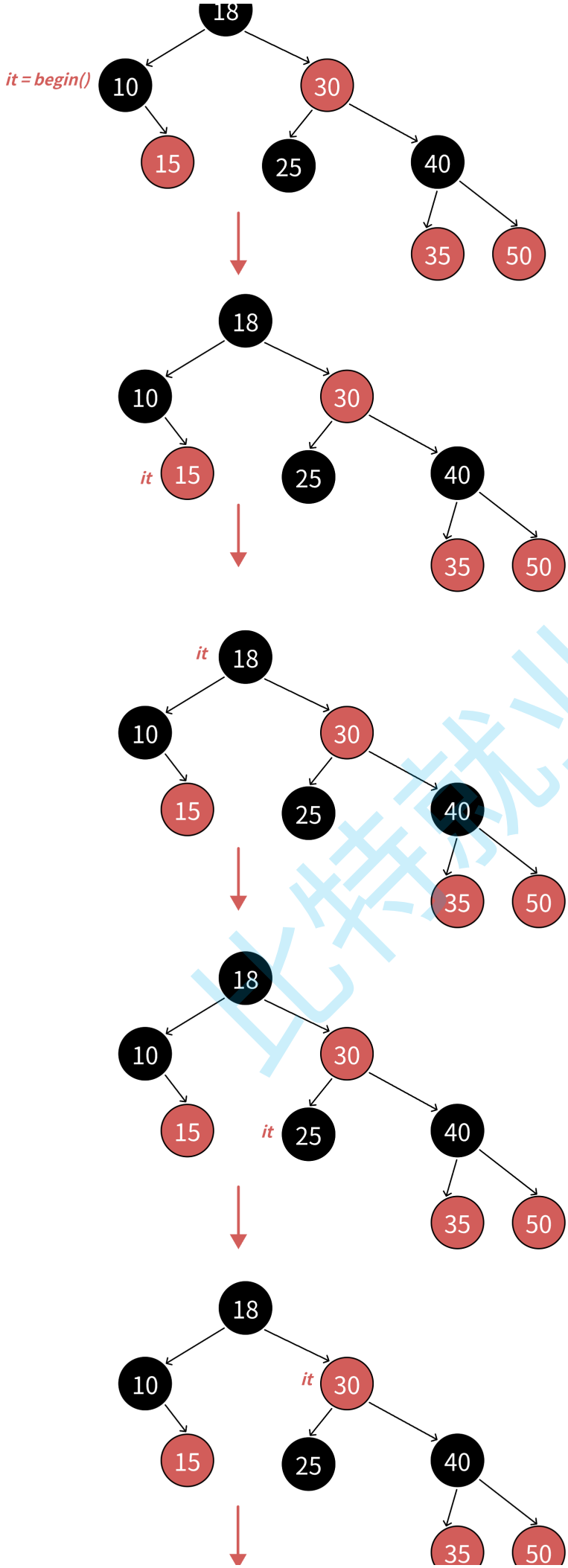
```
struct __rb_tree_base_iterator
{
  typedef __rb_tree_node_base::base_ptr base_ptr;
  base_ptr node;

  void increment()
  {
    if (node->right != 0) {
      node = node->right;
      while (node->left != 0)
        node = node->left;
    }
    else {
      base_ptr y = node->parent;
      while (node == y->right) {
        node = y;
        y = y->parent;
      }
      if (node->right != y)
        node = y;
    }
  }

  void decrement()
  {
```

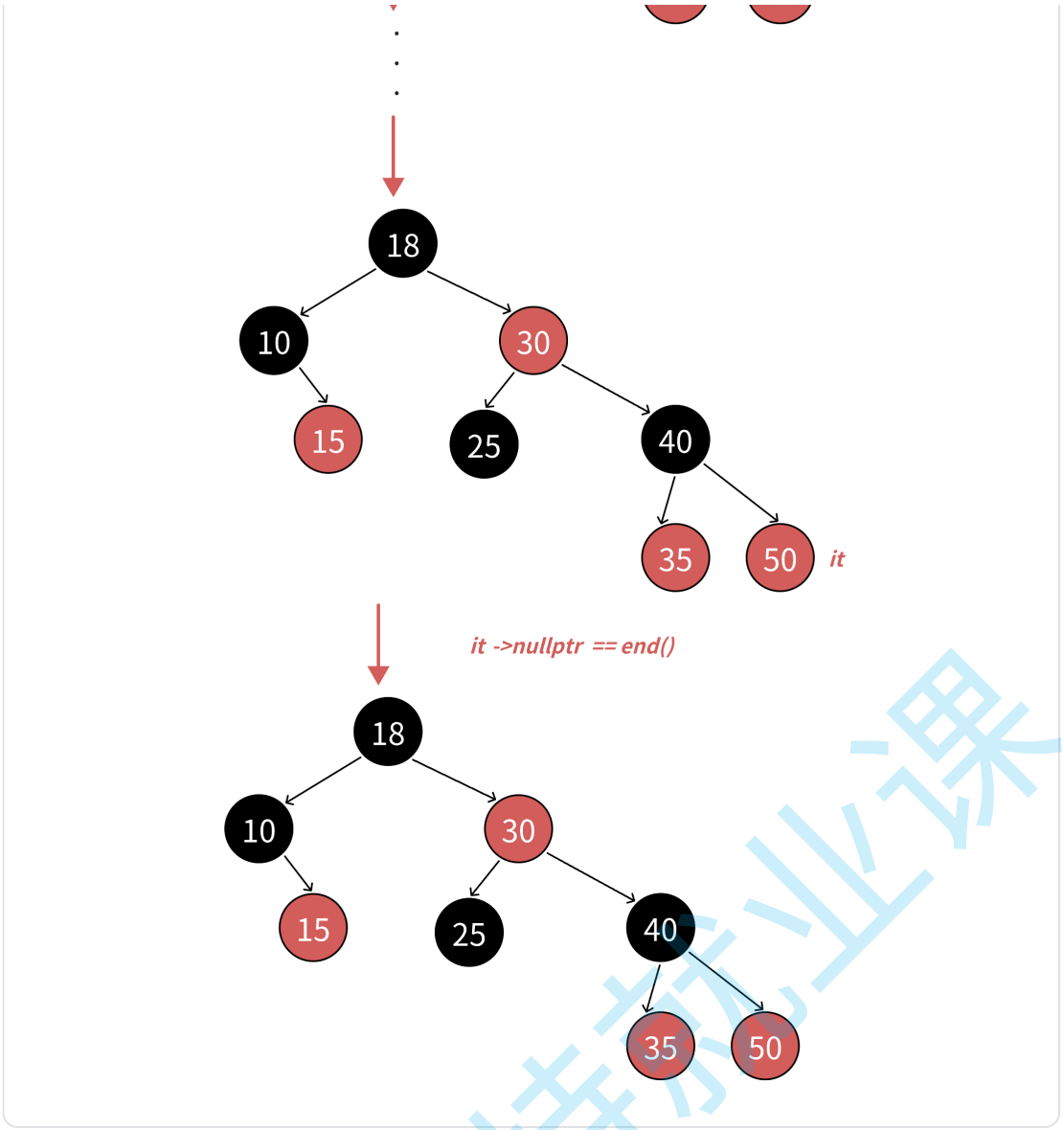
```
26      if (node->color == __rb_tree_red &&
27          node->parent->parent == node)
28        node = node->right;
29      else if (node->left != 0) {
30        base_ptr y = node->left;
31        while (y->right != 0)
32          y = y->right;
33        node = y;
34      }
35      else {
36        base_ptr y = node->parent;
37        while (node == y->left) {
38          node = y;
39          y = y->parent;
40        }
41        node = y;
42      }
43    }
44  };
45
46  template <class Value, class Ref, class Ptr>
47  struct __rb_tree_iterator : public __rb_tree_base_iterator
48  {
49    typedef Value value_type;
50    typedef Ref reference;
51    typedef Ptr pointer;
52    typedef __rb_tree_iterator<Value, Value&, Value*>              iterator;
53    __rb_tree_iterator() {}
54    __rb_tree_iterator(link_type x) { node = x; }
55    __rb_tree_iterator(const iterator& it) { node = it.node; }
56
57    reference operator*() const { return link_type(node)->value_field; }
58  #ifndef __SGI_STL_NO_ARROW_OPERATOR
59    pointer operator->() const { return &(operator*()); }
60  #endif /* __SGI_STL_NO_ARROW_OPERATOR */
61
62    self& operator++() { increment(); return *this; }
63    self& operator--() { decrement(); return *this; }
64
65  inline bool operator==(const __rb_tree_base_iterator& x,
66                         const __rb_tree_base_iterator& y) {
67    return x.node == y.node;
68  }
69
70  inline bool operator!=(const __rb_tree_base_iterator& x,
71                         const __rb_tree_base_iterator& y) {
72    return x.node != y.node;
```

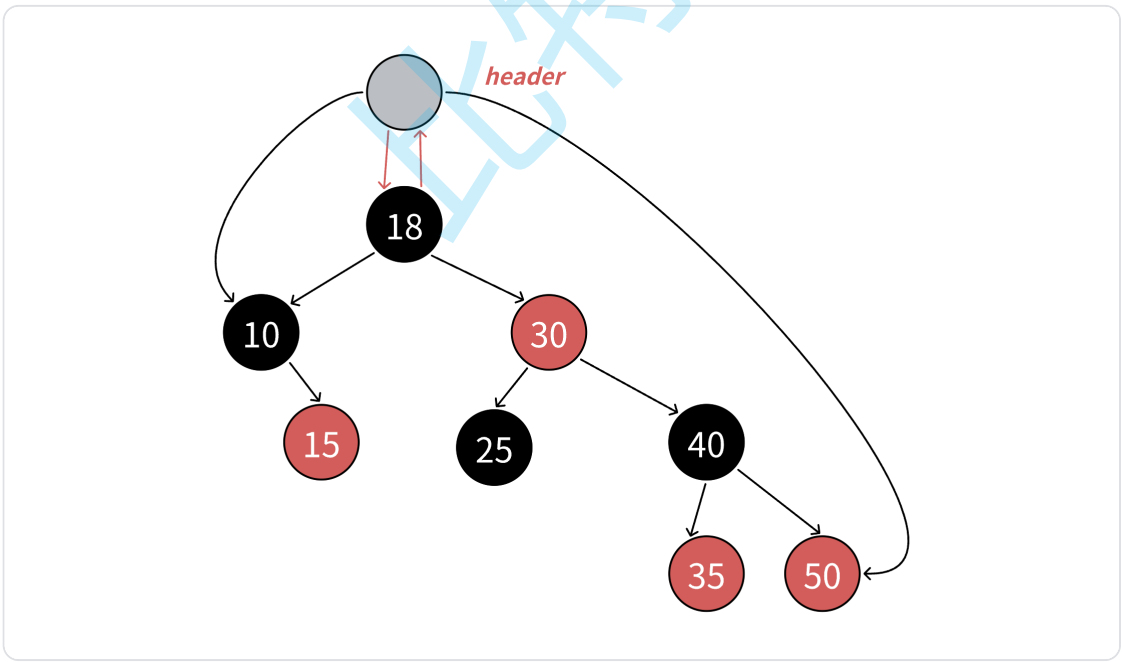
```
73 }
```

iterator实现思路分析

- iterator实现的大框架跟list的iterator思路是一致的，用一个类型封装结点的指针，再通过重载运算符实现，迭代器像指针一样访问的行为。
- 这里的难点是operator++和operator--的实现。之前使用部分，我们分析了，map和set的迭代器走的是中序遍历，左子树->根结点->右子树，那么begin()会返回中序第一个结点的iterator也就是10所在结点的迭代器。
- 迭代器++的核心逻辑就是不看全局，只看局部，只考虑当前中序局部要访问的下一个结点。
- 迭代器++时，如果it指向的结点的右子树不为空，代表当前结点已经访问完了，要访问下一个结点是右子树的中序第一个，一棵树中序第一个是最左结点，所以直接找右子树的最左结点即可。
- 迭代器++时，如果it指向的结点的右子树空，代表当前结点已经访问完了且当前结点所在的子树也访问完了，要访问的下一个结点在当前结点的祖先里面，所以要沿着当前结点到根的祖先路径向上找。
- 如果当前结点是父亲的左，根据中序左子树->根结点->右子树，那么下一个访问的结点就是当前结点的父亲；如下图：it指向25，25右为空，25是30的左，所以下一个访问的结点就是30。
- 如果当前结点是父亲的右，根据中序左子树->根结点->右子树，当前当前结点所在的子树访问完了，当前结点所在父亲的子树也访问完了，那么下一个访问的需要继续往根的祖先中去找，直到找到孩子是父亲左的那个祖先就是中序要问题的下一个结点。如下图：it指向15，15右为空，15是10的右，15所在子树话访问完了，10所在子树也访问完了，继续往上找，10是18的左，那么下一个访问的结点就是18。
- end()如何表示呢？如下图：当it指向50时，++it时，50是40的右，40是30的右，30是18的右，18到根没有父亲，没有找到孩子是父亲左的那个祖先，这是父亲为空了，那我们就把it中的结点指针置为nullptr，我们用nullptr去充当end。需要注意的是stl源码空，红黑树增加了一个哨兵位头结点做为end()，这哨兵位头结点和根互为父亲，左指向最左结点，右指向最右结点。相比我们用nullptr作为end()，差别不大，他能实现的，我们也能实现。只是--end()判断到结点时空，特殊处理一下，让迭代器结点指向最右结点。具体参考迭代器--实现。
- 迭代器--的实现跟++的思路完全类似，逻辑正好反过来即可，因为他访问顺序是右子树->根结点->左子树，具体参考下面代码实现。
- set的iterator也不支持修改，我们把set的第二个模板参数改成const K即可，`RBTree<K, const K, SetKeyOfT> _t;`
- map的iterator不支持修改key但是可以修改value，我们把map的第二个模板参数pair的第一个参数改成const K即可，`RBTree<K, pair<const K, V>, MapKeyOfT> _t;`
- 支持完整的迭代器还有很多细节需要修改，具体参考下面题的代码。


it = begin()
18
10
30
15
25
40
35
50
18
10
30
it
15
25
40
35
50
it
18
10
30
15
25
40
35
50
18
10
30
15
it
25
40
35
50
18
10
it
30
15
25
40
35
50





2.3 map支持[]

- map要支持[]主要需要修改insert返回值支持，修改RBtree中的insert返回值为 `pair<Iterator, bool> Insert(const T& data)`

- 有了insert支持[]实现就很简单了，具体参考下面代码实现

2.4 bit::map和bit::set代码实现

```cpp
// Myset.h
#include"RBTree.h"
namespace bit
{
    template<class K>
    class set
    {
        struct SetKeyOfT
        {
                const K& operator()(const K& key)
                {
                        return key;
                }
        };

    public:
        typedef typename RBTree<K, const K, SetKeyOfT>::Iterator iterator;
        typedef typename RBTree<K, const K, SetKeyOfT>::ConstIterator
const_iterator;

        iterator begin()
        {
                return _t.Begin();
        }

        iterator end()
        {
                return _t.End();
        }

        const_iterator begin() const
        {
                return _t.Begin();
        }

        const_iterator end() const
        {
                return _t.End();
        }

        pair<iterator, bool> insert(const K& key)
        {
```

```
42                  return _t.Insert(key);
43          }
44
45          iterator find(const K& key)
46          {
47                  return _t.Find(key);
48          }
49
50      private:
51          RBTree<K, const K, SetKeyOfT> _t;
52      };
53
54      void Print(const set<int>& s)
55      {
56          set<int>::const_iterator it = s.end();
57          while (it != s.begin())
58          {
59                  --it;
60                  // 不支持修改
61                  //*it += 2;
62
63                  cout << *it << " ";
64          }
65          cout << endl;
66      }
67
68      void test_set()
69      {
70          set<int> s;
71          int a[] = { 4, 2, 6, 1, 3, 5, 15, 7, 16, 14 };
72          for (auto e : a)
73          {
74                  s.insert(e);
75          }
76
77          for (auto e : s)
78          {
79                  cout << e << " ";
80          }
81          cout << endl;
82
83          Print(s);
84      }
85  }
86
87  // Mymap.h
88  #include"RBTree.h"
```

```cpp
namespace bit
{
    template<class K, class V>
    class map
    {
        struct MapKeyOfT
        {
            const K& operator()(const pair<K, V>& kv)
            {
                    return kv.first;
            }
        };
    public:
        typedef typename RBTree<K, pair<const K, V>, MapKeyOfT>::Iterator iterator;
        typedef typename RBTree<K, pair<const K, V>, MapKeyOfT>::ConstIterator const_iterator;

        iterator begin()
        {
                return _t.Begin();
        }

        iterator end()
        {
                return _t.End();
        }

        const_iterator begin() const
        {
                return _t.Begin();
        }

        const_iterator end() const
        {
                return _t.End();
        }

        pair<iterator, bool> insert(const pair<K, V>& kv)
        {
                return _t.Insert(kv);
        }

        iterator find(const K& key)
        {
                return _t.Find(key);
        }
```

```
134
135         V& operator[](const K& key)
136         {
137                 pair<iterator, bool> ret = insert(make_pair(key, V()));
138                 return ret.first->second;
139         }
140
141     private:
142         RBTree<K, pair<const K, V>, MapKeyOfT> _t;
143     };
144
145     void test_map()
146     {
147         map<string, string> dict;
148         dict.insert({ "sort", "排序" });
149         dict.insert({ "left", "左边" });
150         dict.insert({ "right", "右边" });
151
152         dict["left"] = "左边，剩余";
153         dict["insert"] = "插入";
154         dict["string"];
155
156         map<string, string>::iterator it = dict.begin();
157         while (it != dict.end())
158         {
159             // 不能修改first，可以修改second
160             //it->first += 'x';
161             it->second += 'x';
162
163             cout << it->first << ":" << it->second << endl;
164             ++it;
165         }
166         cout << endl;
167     }
168 }
169
170 // RBtree.h
171 enum Colour
172 {
173     RED,
174     BLACK
175 };
176
177 template<class T>
178 struct RBTreeNode
179 {
180     T _data;
```

```
181
182     RBTreeNode<T>* _left;
183     RBTreeNode<T>* _right;
184     RBTreeNode<T>* _parent;
185     Colour _col;
186
187     RBTreeNode(const T& data)
188             : _data(data)
189             , _left(nullptr)
190             , _right(nullptr)
191             , _parent(nullptr)
192     {}
193 };
194
195     template<class T, class Ref, class Ptr>
196     struct RBTreeIterator
197     {
198     typedef RBTreeNode<T> Node;
199     typedef RBTreeIterator<T, Ref, Ptr> Self;
200
201     Node* _node;
202     Node* _root;
203
204     RBTreeIterator(Node* node, Node* root)
205             :_node(node)
206             ,_root(root)
207     {}
208
209     Self& operator++()
210     {
211         if (_node->_right)
212         {
213                 // 右不为空，右子树最左结点就是中序第一个
214                 Node* leftMost = _node->_right;
215                 while (leftMost->_left)
216                 {
217                         leftMost = leftMost->_left;
218                 }
219
220                 _node = leftMost;
221         }
222         else
223         {
224                 // 孩子是父亲左的那个祖先
225                 Node* cur = _node;
226                 Node* parent = cur->_parent;
227                 while (parent && cur == parent->_right)
```

```
228                 {
229                         cur = parent;
230                         parent = cur->_parent;
231                 }
232 
233                 _node = parent;
234         }
235 
236         return *this;
237     }
238 
239     Self& operator--()
240     {
241         if (_node == nullptr) // end()
242         {
243                 // --end()，特殊处理，走到中序最后一个结点，整棵树的最右结点
244                 Node* rightMost = _root;
245                 while (rightMost && rightMost->_right)
246                 {
247                         rightMost = rightMost->_right;
248                 }
249 
250                 _node = rightMost;
251         }
252         else if (_node->_left)
253         {
254                 // 左子树不为空，中序左子树最后一个
255                 Node* rightMost = _node->_left;
256                 while (rightMost->_right)
257                 {
258                         rightMost = rightMost->_right;
259                 }
260 
261                 _node = rightMost;
262         }
263         else
264         {
265                 // 孩子是父亲右的那个祖先
266                 Node* cur = _node;
267                 Node* parent = cur->_parent;
268                 while (parent && cur == parent->_left)
269                 {
270                         cur = parent;
271                         parent = cur->_parent;
272                 }
273 
274                 _node = parent;
```

```
275
276         }
277
278         return *this;
279     }
280
281     Ref operator*()
282     {
283             return _node->_data;
284     }
285
286     Ptr operator->()
287     {
288             return &_node->_data;
289     }
290
291     bool operator!= (const Self& s) const
292     {
293             return _node != s._node;
294     }
295
296     bool operator== (const Self& s) const
297     {
298             return _node == s._node;
299     }
300 };
301
302 template<class K, class T, class KeyOfT>
303 class RBTree
304 {
305     typedef RBTreeNode<T> Node;
306 public:
307     typedef RBTreeIterator<T, T&, T*> Iterator;
308     typedef RBTreeIterator<T, const T&, const T*> ConstIterator;
309
310     Iterator Begin()
311     {
312             Node* leftMost = _root;
313             while (leftMost && leftMost->_left)
314             {
315                     leftMost = leftMost->_left;
316             }
317
318             return Iterator(leftMost, _root);
319     }
320
321     Iterator End()
```

```
322     {
323             return Iterator(nullptr, _root);
324     }
325
326     ConstIterator Begin() const
327     {
328             Node* leftMost = _root;
329             while (leftMost && leftMost->_left)
330             {
331                     leftMost = leftMost->_left;
332             }
333
334             return ConstIterator(leftMost, _root);
335     }
336
337     ConstIterator End() const
338     {
339             return ConstIterator(nullptr, _root);
340     }
341
342     RBTree() = default;
343
344     ~RBTree()
345     {
346             Destroy(_root);
347             _root = nullptr;
348     }
349
350     pair<Iterator, bool> Insert(const T& data)
351     {
352             if (_root == nullptr)
353             {
354                     _root = new Node(data);
355                     _root->_col = BLACK;
356                     return make_pair(Iterator(_root, _root), true);
357             }
358
359             KeyOfT kot;
360             Node* parent = nullptr;
361             Node* cur = _root;
362             while (cur)
363             {
364                     if (kot(cur->_data) < kot(data))
365                     {
366                             parent = cur;
367                             cur = cur->_right;
368                     }
```

```
369                     else if (kot(cur->_data) > kot(data))
370                     {
371                             parent = cur;
372                             cur = cur->_left;
373                     }
374                     else
375                     {
376                             return make_pair(Iterator(cur, _root), false);
377                     }
378             }
379
380             cur = new Node(data);
381             Node* newnode = cur;
382
383             // 新增结点。颜色红色给红色
384             cur->_col = RED;
385             if (kot(parent->_data) < kot(data))
386             {
387                     parent->_right = cur;
388             }
389             else
390             {
391                     parent->_left = cur;
392             }
393             cur->_parent = parent;
394
395             while (parent && parent->_col == RED)
396             {
397                     Node* grandfather = parent->_parent;
398                     //    g
399                     //  p   u
400                     if (parent == grandfather->_left)
401                     {
402                             Node* uncle = grandfather->_right;
403                             if (uncle && uncle->_col == RED)
404                             {
405                                     // u存在且为红 -》变色再继续往上处理
406                                     parent->_col = uncle->_col = BLACK;
407                                     grandfather->_col = RED;
408
409                                     cur = grandfather;
410                                     parent = cur->_parent;
411                             }
412                             else
413                             {
414                                     // u存在且为黑或不存在 -》旋转+变色
415                                     if (cur == parent->_left)
```

```
416                                 {
417                                         //    g
418                                         //  p   u
419                                         //c
420                                         //单旋
421                                         RotateR(grandfather);
422                                         parent->_col = BLACK;
423                                         grandfather->_col = RED;
424                                 }
425                                 else
426                                 {
427                                         //    g
428                                         //  p   u
429                                         //    c
430                                         //双旋
431                                         RotateL(parent);
432                                         RotateR(grandfather);
433
434                                         cur->_col = BLACK;
435                                         grandfather->_col = RED;
436                                 }
437
438                                 break;
439                         }
440                 }
441                 else
442                 {
443                         //  g
444                         // u   p
445                         Node* uncle = grandfather->_left;
446                         // 叔叔存在且为红，-》变色即可
447                         if (uncle && uncle->_col == RED)
448                         {
449                                 parent->_col = uncle->_col = BLACK;
450                                 grandfather->_col = RED;
451
452                                 // 继续往上处理
453                                 cur = grandfather;
454                                 parent = cur->_parent;
455                         }
456                         else // 叔叔不存在，或者存在且为黑
457                         {
458                                 // 情况二：叔叔不存在或者存在且为黑
459                                 // 旋转+变色
460                                 //      g
461                                 //   u     p
462                                 //           c
```

```
463                                     if (cur == parent->_right)
464                                     {
465                                             RotateL(grandfather);
466                                             parent->_col = BLACK;
467                                             grandfather->_col = RED;
468                                     }
469                                     else
470                                     {
471                                             //               g
472                                             //   u     p
473                                             //      c
474                                             RotateR(parent);
475                                             RotateL(grandfather);
476                                             cur->_col = BLACK;
477                                             grandfather->_col = RED;
478                                     }
479                                     break;
480                             }
481                     }
482             }
483
484             _root->_col = BLACK;
485
486             return make_pair(Iterator(newnode, _root), true);
487     }
488
489     Iterator Find(const K& key)
490     {
491             Node* cur = _root;
492             while (cur)
493             {
494                     if (cur->_kv.first < key)
495                     {
496                             cur = cur->_right;
497                     }
498                     else if (cur->_kv.first > key)
499                     {
500                             cur = cur->_left;
501                     }
502                     else
503                     {
504                             return Iterator(cur, _root);
505                     }
506             }
507
508             return End();
509     }
```

```
510
511 private:
512     void RotateL(Node* parent)
513     {
514             Node* subR = parent->_right;
515             Node* subRL = subR->_left;
516
517             parent->_right = subRL;
518             if (subRL)
519                     subRL->_parent = parent;
520
521             Node* parentParent = parent->_parent;
522
523             subR->_left = parent;
524             parent->_parent = subR;
525
526             if (parentParent == nullptr)
527             {
528                     _root = subR;
529                     subR->_parent = nullptr;
530             }
531             else
532             {
533                     if (parent == parentParent->_left)
534                     {
535                             parentParent->_left = subR;
536                     }
537                     else
538                     {
539                             parentParent->_right = subR;
540                     }
541
542                     subR->_parent = parentParent;
543             }
544     }
545
546     void  RotateR(Node* parent)
547     {
548             Node* subL = parent->_left;
549             Node* subLR = subL->_right;
550
551             parent->_left = subLR;
552             if (subLR)
553                     subLR->_parent = parent;
554
555             Node* parentParent = parent->_parent;
556
```

```
557             subL->_right = parent;
558             parent->_parent = subL;
559 
560             if (parentParent == nullptr)
561             {
562                     _root = subL;
563                     subL->_parent = nullptr;
564             }
565             else
566             {
567                     if (parent == parentParent->_left)
568                     {
569                             parentParent->_left = subL;
570                     }
571                     else
572                     {
573                             parentParent->_right = subL;
574                     }
575 
576                     subL->_parent = parentParent;
577             }
578 
579     }
580 
581     void Destroy(Node* root)
582     {
583             if (root == nullptr)
584                     return;
585 
586             Destroy(root->_left);
587             Destroy(root->_right);
588             delete root;
589     }
590 
591 private:
592     Node* _root = nullptr;
593 };
```