

Project3 红黑树和区间树实验报告

徐煜森 PB16110173

一. 实验要求

实验 1: 实现红黑树的基本算法, 分别对整数 $n = 20, 40, 60, 80, 100$, 随机生成 n 个互异的正整数 ($K_1, K_2, K_3, \dots, K_n$), 以这 n 个正整数作为节点的关键字, 向一棵初始空的红黑树中依次插入 n 个节点, 然后随机选择其中 $n/10$ 个节点进行删除, 统计插入和删除算法运行所需时间, 画出时间曲线。

实验 2: 实现区间树的基本算法, 随机生成 30 个正整数区间, 以这 30 个正整数区间的左端点作为关键字构建红黑树, 向一棵初始空的红黑树中依次插入 30 个节点, 然后随机选择其中 3 个区间进行删除。实现区间树的插入、删除、遍历和查找算法。

二. 实验环境

1. Windows10 64 位 x86, 机器内存 8G, 时钟主频 2.59GHz
2. 软件环境: Visual Studio 2017

三. 实验过程

0. 编译选项

注意编译选项 `-std = c++11`

1. Ex1 红黑树

a. 随机生成数据

```
int N = 101;
bool history[65536];
for (int i = 0; i < 65536; i++) {
    history[i] = false;
}
int rdm;
srand(unsigned(time(0)));
for (int i = 0; i < N; i++) {
    rdm = (rand() + rand() + rand()) % 65536;
    while (history[rdm]) {
        rdm = (rand() + rand() + rand()) % 65536;
    }
    history[rdm] = true;
    inputA << rdm << endl;
}
```

- b. 按照书上代码实现红黑树的各个基本操作，按照实验要求调用操作，并进行格式化输出。期间遇到了不少问题，代码中也有部分需要注意的细节，将在关键代码部分和总结部分叙述。

c. 程序运行截图

将部分内容进行屏幕输出方便调试。

D:\Study\算法基础\lab3\RedBlack

```
n = 20
插入10个节点用时:7902ns
插入20个节点用时:1645037ns
删除1个节点用时:1581ns
删除2个节点用时:5183603ns
删除的关键字为:
41757
44088

n = 40
插入10个节点用时:3556ns
插入20个节点用时:4282467ns
插入30个节点用时:6648096ns
插入40个节点用时:9462910ns
删除1个节点用时:1185ns
删除2个节点用时:1189136ns
删除3个节点用时:2378666ns
删除4个节点用时:3537382ns
删除的关键字为:
27791
40898
28304
43301

n = 60
插入10个节点用时:3950ns
插入20个节点用时:1528888ns
插入30个节点用时:2840492ns
插入40个节点用时:4050171ns
插入50个节点用时:5314763ns
```

2. Ex2 区间树

a. 随机生成数据

```
bool lefthistory[51];
for (int i = 0; i < 51; i++) {
    lefthistory[i] = false;
}

int left, right;

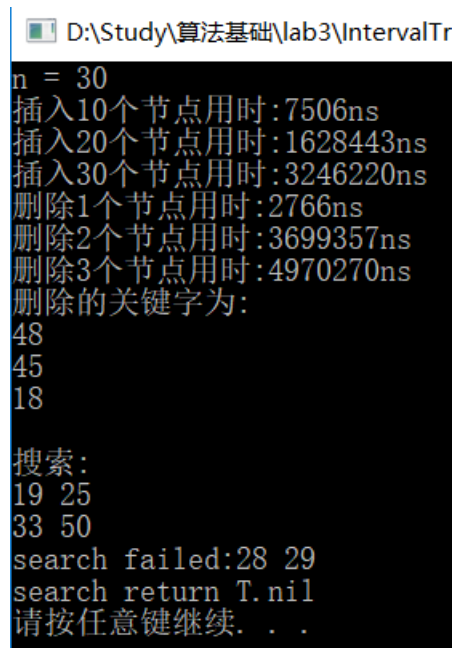
for (int i = 0; i < N; i++) {
    if (rand() % 2) {
        left = rand() % 25;
        while (lefthistory[left]) {
            left = rand() % 25;
        }
        right = left + 1 + rand() % (25 - left);
        lefthistory[left] = true;
        inputB << left << ' ' << right << endl;
    }
    else {
        left = 30 + rand() % 20;
        while (lefthistory[left]) {
            left = 30 + rand() % 20;
        }
        right = left + 1 + rand() % (50 - left);
        lefthistory[left] = true;
        inputB << left << ' ' << right << endl;
    }
}
```

b. 按照书上算法实现区间树的数据结构扩张，和区间搜索函数。

思考后修改左旋、右旋、插入、删除函数，维护节点 `max` 值。

c. 程序运行截图

输出运行信息到屏幕方便调试。



```
D:\Study\算法基础\lab3\IntervalTr
n = 30
插入10个节点用时:7506ns
插入20个节点用时:1628443ns
插入30个节点用时:3246220ns
删除1个节点用时:2766ns
删除2个节点用时:3699357ns
删除3个节点用时:4970270ns
删除的关键字为:
48
45
18

搜索:
19 25
33 50
search failed:28 29
search return T.nil
请按任意键继续. . .
```

四. 关键代码截图

1. Ex1 红黑树

树结构的实现:

```

15  enum NodeColor
16  {
17      RED,
18      BLACK,
19  };
20
21  typedef struct _tmpRedBlackTreeNode {
22      struct _tmpRedBlackTreeNode *p;
23      struct _tmpRedBlackTreeNode *left;
24      struct _tmpRedBlackTreeNode *right;
25
26      NodeColor color;
27      int key;
28  }RedBlackTreeNode;
29
30  class RedBlackTree {
31  public:
32      RedBlackTreeNode *root;
33      RedBlackTreeNode *nil;
34  };

```

a. 左旋

按照书上的方法实现左旋函数，图中注释标明了三个步骤：

```

45  void left_rotate(RedBlackTree &T, RedBlackTreeNode *x) {
46      RedBlackTreeNode *y;
47      y = x->right;
48
49      x->right = y->left;          // put y.left on x.right
50      if (y->left != T.nil) {
51          y->left->p = x;
52      }
53
54      y->p = x->p;                // link x.p to y
55      if (x->p == T.nil) {
56          T.root = y;
57      }
58      else if (x == x->p->left) {
59          x->p->left = y;
60      }
61      else {
62          x->p->right = y;
63      }
64
65      y->left = x;                // put x on y.left
66      x->p = y;
67  }

```

b. 右旋

按照书上的方法实现右旋函数，与左旋对称：

```

69 void right_rotate(RedBlackTree &T, RedBlackTreeNode *x) {
70     RedBlackTreeNode *y;
71     y = x->left;
72
73     x->left = y->right;          // put y.right on x.left
74     if (y->right != T.nil) {
75         y->right->p = x;
76     }
77
78     y->p = x->p;                // link x.p to y
79     if (x->p == T.nil) {
80         T.root = y;
81     }
82     else if (x == x->p->left) {
83         x->p->left = y;
84     }
85     else {
86         x->p->right = y;
87     }
88
89     y->right = x;              // put x on y.right
90     x->p = y;
91 }

```

c. 插入

按照书上方法实现插入函数，其中 y 为插入节点的父亲：

```

137 void RB_insert(RedBlackTree &T, RedBlackTreeNode *z) {
138     RedBlackTreeNode *x, *y;
139
140     y = T.nil;                // y = x.p
141     x = T.root;
142     while (x != T.nil) {
143         y = x;
144         if (z->key < x->key) {
145             x = x->left;
146         }
147         else {
148             x = x->right;
149         }
150     }
151     z->p = y;
152     if (y == T.nil) {
153         T.root = z;
154     }
155     else if (z->key < y->key) {
156         y->left = z;
157     }
158     else {
159         y->right = z;
160     }
161     z->left = T.nil;
162     z->right = T.nil;
163     z->color = RED;
164     RB_insert_fixup(T, z);
165 }

```

d. 插入后修正

按照书上方法实现 `insert_fixup` 函数：

其中两种情况的 case1/2/3 都已在注释中标明。值得注意的是因为执行 case2 之后一定会进入 case3，所以将 case2/3 按顺序写在同一个 else 语句块中。

```
93 void RB_insert_fixup(RedBlackTree &T, RedBlackTreeNode *z) {
94     RedBlackTreeNode *y;
95     while (z->p->color == RED) {
96         if (z->p == z->p->p->left) {
97             y = z->p->p->right;
98             if (y->color == RED) {           // case 1
99                 z->p->color = BLACK;
100                 y->color = BLACK;
101                 z->p->p->color = RED;
102                 z = z->p->p;
103             }
104             else {
105                 if (z == z->p->right) {      // case 2
106                     z = z->p;
107                     left_rotate(T, z);
108                 }
109                 z->p->color = BLACK;         // case 3
110                 z->p->p->color = RED;
111                 right_rotate(T, z->p->p);
112             }
113         }
114     }
115     else {
116         y = z->p->p->left;
117         if (y->color == RED) {           // case 1
118             z->p->color = BLACK;
119             y->color = BLACK;
120             z->p->p->color = RED;
121             z = z->p->p;
122         }
123         else {
124             if (z == z->p->left) {          // case 2
125                 z = z->p;
126                 right_rotate(T, z);
127             }
128             z->p->color = BLACK;           // case 3
129             z->p->p->color = RED;
130             left_rotate(T, z->p->p);
131         }
132     }
133 }
134 T.root->color = BLACK;
135 }
```

e. 删除

按照第二版书上方法实现删除函数，其中各个步骤已在注释中标明：

```

246 void RB_delete(RedBlackTree &T, RedBlackTreeNode *z) {
247     RedBlackTreeNode *x, *y;
248     // y is node to be deleted
249     if (z->left == T.nil || z->right == T.nil) {
250         y = z;
251     }
252     else {
253         y = Successor(T, z);          // successor only has right child
254     }
255     // x is node to replaced y
256     if (y->left != T.nil) {
257         x = y->left;
258     }
259     else {
260         x = y->right;
261     }
262     x->p = y->p;
263
264     if (y->p == T.nil) {
265         T.root = x;
266     }
267     else if (y == y->p->left) {
268         y->p->left = x;
269     }
270     else {
271         y->p->right = x;
272     }
273
274     if (y != z) {
275         // copy y.key to z.key
276         z->key = y->key;
277     }
278     if (y->color == BLACK) {          // replace y with x, so x.color += black
279         RB_delete_fixup(T, x);
280     }
281     delete y;
282 }

```

f. 删除后修正

本函数中的 case1/2/3/4 已在注释中标明，值得注意的是 case1 执行后有可能进入 case2/3/4，所以没有在调整好 case1 后就进入下一循环。另外，case3 执行后一定进入 case4，所以将 case3/4 按顺序写在同一个 else 语句块中。


```

167 void RB_delete_fixup(RedBlackTree &T, RedBlackTreeNode *x) {
168     RedBlackTreeNode *w;
169     while (x != T.root && x->color == BLACK) {
170         if (x == x->p->left) {
171             w = x->p->right;
172             if (w->color == RED) { // case 1
173                 w->color = BLACK;
174                 x->p->color = RED;
175                 left_rotate(T, x->p);
176                 w = x->p->right;
177             }
178             if (w->left->color == BLACK && w->right->color == BLACK) { // case 2
179                 w->color = RED;
180                 x = x->p;
181             }
182             else {
183                 if (w->right->color == BLACK) { // case 3
184                     w->left->color = BLACK;
185                     w->color = RED;
186                     right_rotate(T, w);
187                     w = x->p->right;
188                 }
189                 w->color = x->p->color; // case 4
190                 x->p->color = BLACK;
191                 w->right->color = BLACK;
192                 left_rotate(T, x->p);
193                 x = T.root;
194             }
195         }
196     }

```

```

197     else {
198         w = x->p->left;
199         if (w->color == RED) { // case 1
200             w->color = BLACK;
201             x->p->color = RED;
202             right_rotate(T, x->p);
203             w = x->p->left;
204         }
205         if (w->left->color == BLACK && w->right->color == BLACK) { // case 2
206             w->color = RED;
207             x = x->p;
208         }
209         else {
210             if (w->left->color == BLACK) { // case 3
211                 w->right->color = BLACK;
212                 w->color = RED;
213                 left_rotate(T, w);
214                 w = x->p->left;
215             }
216             w->color = x->p->color; // case 4
217             x->p->color = BLACK;
218             w->left->color = BLACK;
219             right_rotate(T, x->p);
220             x = T.root;
221         }
222     }
223 }
224 x->color = BLACK;
225 }

```

g. 寻找节点的中序遍历后继

在这里我写错了一行代码，但是因为在 `delete` 函数中只有 $1/2$ 的概率执行本函数（ $1/2$ 的节点为非叶子节点），所以这个 bug 一直时隐时现。

```
227 RedBlackTreeNode* Successor(RedBlackTree T, RedBlackTreeNode *x) {
228     RedBlackTreeNode *y;
229     if (x->right != T.nil) {
230         x = x->right;           // bug : miss this line
231                                 // because this function have 1/2 possibility to run,
232                                 // so sometimes can run correctly
233         while (x->left != T.nil) {
234             x = x->left;
235         }
236         return x;
237     }
238     y = x->p;
239     while (y != T.nil && x == y->right) {
240         x = y;
241         y = y->p;
242     }
243     return y;
244 }
```

h. 遍历

```
298 void RB_preorder(RedBlackTree T, RedBlackTreeNode *x) {
299     if (x == T.nil) {
300         return;
301     }
302     Traversal << x->key << (x->color ? "B" : "R") << endl;
303     RB_preorder(T, x->left);
304     RB_preorder(T, x->right);
305 }
306
307 void RB_inorder(RedBlackTree T, RedBlackTreeNode *x) {
308     if (x == T.nil) {
309         return;
310     }
311     RB_inorder(T, x->left);
312     Traversal << x->key << (x->color ? "B" : "R") << endl;
313     RB_inorder(T, x->right);
314 }
315
316 void RB_postorder(RedBlackTree T, RedBlackTreeNode *x) {
317     if (x == T.nil) {
318         return;
319     }
320     RB_postorder(T, x->left);
321     RB_postorder(T, x->right);
322     Traversal << x->key << (x->color ? "B" : "R") << endl;
323 }
```

i. 精确查找（用于删除，返回指向查找节点的指针）

```
284 RedBlackTreeNode* RB_search(RedBlackTree T, RedBlackTreeNode *node, int key) {
285     if (key == node->key || node == T.nil) {
286         return node;
287     }
288     if (key < node->key) {
289         return RB_search(T, node->left, key);
290     }
291     else {
292         return RB_search(T, node->right, key);
293     }
294 }
```

2. Ex2 区间树

区间树的代码与红黑树类似，只是增加了对节点 `max` 值的维护，下面为与红黑树有区别的代码：

树的结构增加 `max` 值，将 `key` 换为结构体“区间”，其包含左端点和右端点，把左端点当 `key` 建树。

```
21 typedef struct _interval{
22     int low;
23     int high;
24 }RB_interval;
25
26 typedef struct _tmpRedBlackTreeNode {
27     struct _tmpRedBlackTreeNode *p;
28     struct _tmpRedBlackTreeNode *left;
29     struct _tmpRedBlackTreeNode *right;
30
31     NodeColor color;
32     int max;
33     RB_interval interval;
34 }RedBlackTreeNode;
```

a. 左旋与右旋

在左旋、右旋函数最后增加代码如下：

```

// update max
y->max = x->max;
x->max = max(
    x->interval.high,
    (x->left != T.nil) ? x->left->max : 0,
    (x->right != T.nil) ? x->right->max : 0
);

```

b. 插入

在图中如下位置（沿 **T.root** 向下走的路径上）加入更新 **max** 值的语句：

```

170 void RB_insert(RedBlackTree &T, RedBlackTreeNode *z) {
171     RedBlackTreeNode *x, *y;
172
173     y = T.nil;           // y = x.p
174     x = T.root;
175     while (x != T.nil) {
176
177         // update max along T.root to insert location
178         x->max = max(x->max, z->max);
179
180         y = x;
181         if (z->interval.low < x->interval.low) {
182             x = x->left;
183         }
184         else {
185             x = x->right;
186         }
187     }

```

c. 删除

在调用 **fixup** 函数前加入更新 **max** 值的语句，从实际要删除的 **y** 节点一路向上更新 **max**，直到 **T.root**：

```

321     // update max alone y.p .. to T.root
322     z = y->p;
323     while (z != T.nil) {
324         z->max = max(
325             z->max,
326             (z->left != T.nil) ? z->left->max : 0,
327             (z->right != T.nil) ? z->right->max : 0
328         );
329         z = z->p;
330     }
331
332     // replace y with x, so x.color += black
333     if (y->color == BLACK) {
334         RB_delete_fixup(T, x);
335     }
336     delete y;

```

d. 区间查找

其中 `Overlap` 函数用于判断两个节点是否重叠。

```

381 bool overlap(RB_interval a, RB_interval b) {
382     if (a.high < b.low || a.low > b.high) {
383         return false;
384     }
385     else return true;
386 }
387
388 RedBlackTreeNode* IntervalSearch(RedBlackTree T, RB_interval i) {
389     RedBlackTreeNode *x;
390     x = T.root;
391     while (x != T.nil && !overlap(x->interval, i)) {
392         if (x->left != T.nil && x->left->max >= i.low) {
393             x = x->left;
394         }
395         else x = x->right;
396     }
397     return x;
398 }

```

五. 实验结果及分析

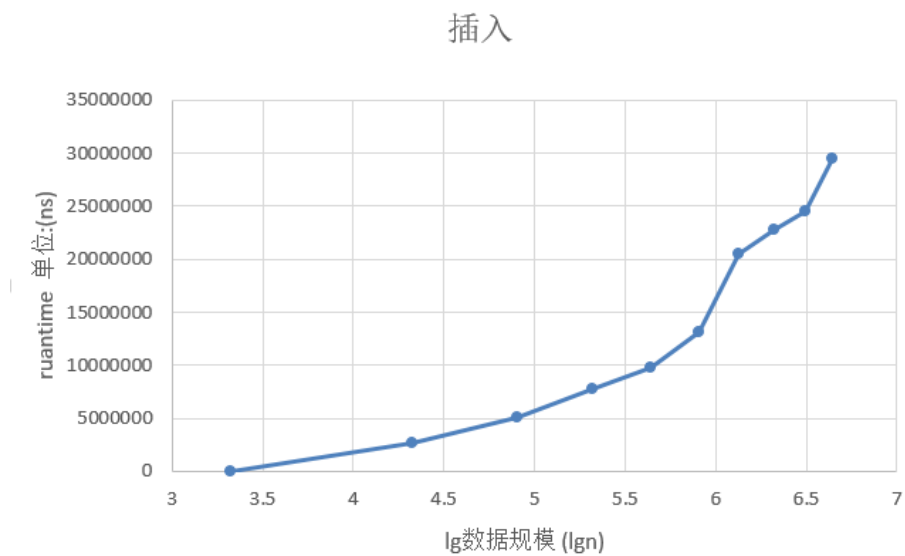
1. Ex1 红黑树

将 size20/40/60/80/100 中测出的运行时间相对应取平均值，期望

插入和删除的运行时间都是 $O(\log n)$ ，所以对 n 以 2 为底取对数，
处理结果如下：

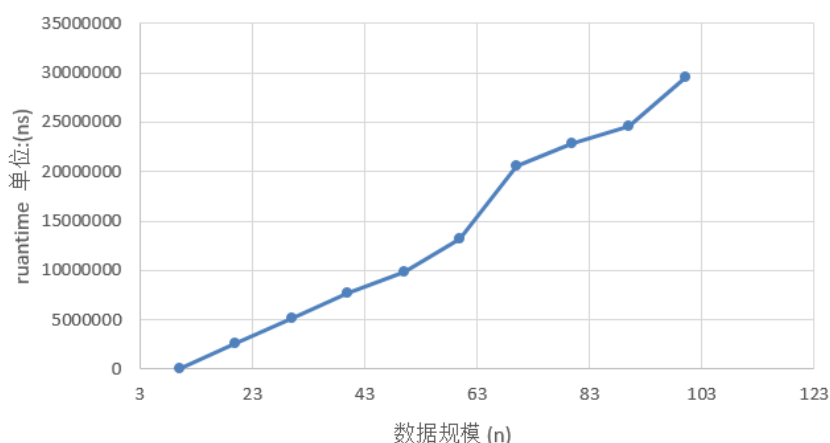
		插入	删除
	$\log(n)$	平均运行时间 (ns)	平均运行时间 (ns)
113			
114	3. 321928	5136	1264. 2
115	4. 321928	2642883	2369105
116	4. 906891	5134714	3656986
117	5. 321928	7751503	5457282
118	5. 643856	9818992	8545708
119	5. 906891	13170036	10929905
120	6. 129283	20522856	15671895
121	6. 321928	22827250	19535399
122	6. 491853	24537669	26344680
123	6. 643856	29507149	30198506
124			

a. 插入



将 $\lg(n)$ -运行时间绘成图表发现，曲线的形状与线性有些差别。再
将 n -运行时间绘成图表

插入



发现运行时间与 n 几乎成线性，与期望不太相符。

为了探明原因，使用 C++ 中库的红黑树（`set` 的内部实现为红黑树）测试相同的数据，发现其运行时间也几乎成线性，与上图一致。于是推测：红黑树在节点数较少（10-100）时，其插入删除操作的运行时间与 n 近似成线性关系（类似对数函数曲线的局部放大曲线）。

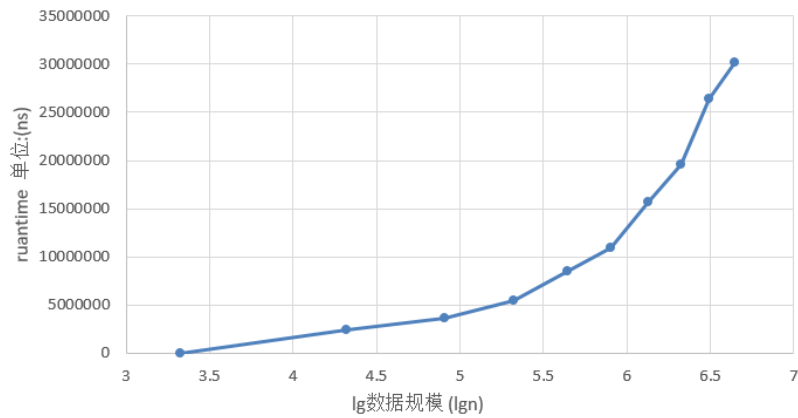
分析原因：

- 1) 在节点数较少时，红黑树插入删除操作之后调用的 `fixup` 函数运行时间可能与插入删除运行时间相当，使运行时间 $O(\log n)$ 在不同的范围有不同的常数级别的增加。
- 2) 在节点数较少时，运行时间可能受到硬件与操作系统影响较大，数据从 `cache` 中调入调出对运行时间的测量有影响。

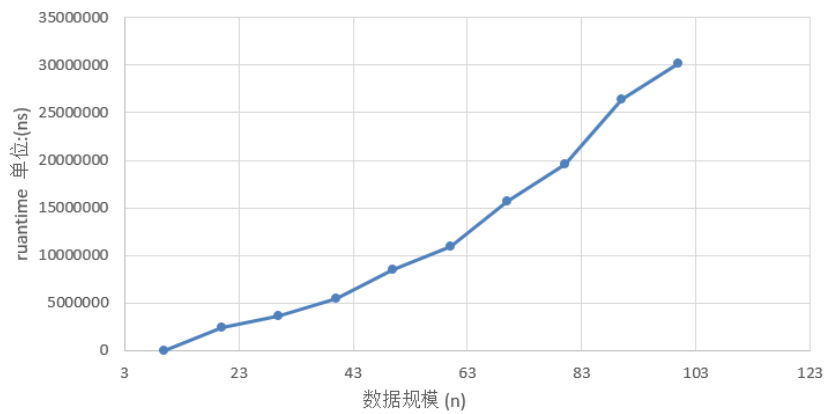
b. 删除

$\lg(n)$ -运行时间与 n -运行时间的图表如下，分析造成这种情况的原因与插入相同：

删除



删除



造成这种情况的原因与插入部分中分析的相同，即节点数较少时 `fixup` 函数运行时间和硬件影响较大。

六. 实验总结

总结一下本次实验中收获的经验：

1. C++中面向对象，默认函数参数传递对象为值传递，非引用传递。

2. 对一个输入流 `ofstream.open` 后需要 `close`，才能 `open` 下一次。
3. 插入删除 `fixup` 函数中最后两个 `case` 的执行一定是按顺序的（如插入中先执行 `case2`，之后一定进入 `case3`），所以最后两个 `case` 写在同一个 `else` 语句块里面。
4. 找节点后继函数写错了，发现 `bug` 时隐时见：因为只有约 $1/2$ 的节点为叶节点，所以删除里面找后继的语句只有约 $1/2$ 的概率运行。