实验 3　红黑树和顺序统计树

Pb15111604
金泽文

1. 实验要求

- ex1：　实现红黑树的基本算法，对 n 的取值分别为 12、24、36、48、60，随机生成 n 个互异的正整数（K1，K2，K3，……，Kn）作为节点的关键字，向一棵初始空的红黑树中依次插入这 n 个节点，统计算法运行所需时间，画出时间曲线。（红黑树采用三叉链表）。
- ex2：　对上述生成的红黑树，找出树中的第 n/3 小的节点和第 n/4 小的节点，并删除这两个节点，统计算法运行所需时间，画出时间曲线。

2. 实验环境

编译环境：Ubuntu 16.04.3 LTS (WSL- Windows Subsystem for Linux)
Openjdk version "1.8.0_151"
OpenJDK Runtime Environment (build 1.8.0_151)
编程语言：Java SE8
机器内存：16G
时钟主频：2.3GHz

3. Build

为了方便构建，我写了个 Makefile，使用方法如下：

```
Reaper@KZ:/mnt/g/PB15111604-project3/src$ make help
make                - 只生成class并且运行ex1,ex2相关代码
make RedBlack       - 只生成class并且运行ex1,ex2相关代码
make RedBlack check - 每次insert,delete都会输出红黑树
make gen            - 生成随机数存到input中
make clean          - 删除*.class
make help           - 打印以上信息
```

最后的目录结构如下：

```
Reaper@KZ:/mnt/g/PB15111604-project3$ tree -L 2
.
├── input
│   └── input.txt
├── output
│   ├── size12
│   ├── size24
│   ├── size36
│   ├── size48
│   └── size60
├── PB15111604-金泽文-project3.doc
└── src
    ├── GenerateIntegers.java
    ├── Makefile
    ├── RBNode.class
    ├── RBTree.class
    ├── RedBlack.class
    └── RedBlack.java

8 directories, 8 files
```

```
Reaper@KZ:/mnt/g/PB15111604-project3$ tree output/size60
output/size60
├── delete_data.txt
├── inorder.txt
├── postorder.txt
├── preorder.txt
├── time1.txt
└── time2.txt

0 directories, 6 files
```

4. 实验过程

1. 生成随机数

i. 生成不相同的随机数。生成一个之后检查是否已在数组中，不在则加入，并计数加一。

2. Ex1 红黑树的基本算法：

i. 构建红黑树节点类 RBNode：

RBNode 的 filed 如下：

```
int key;
boolean color;
int size = 0;
RBNode left;
RBNode right;
RBNode parent;
```

ii. 构建红黑树类 RBTree：

1. 一共 4 个 filed：

```
public RBNode root;                          // 根节点
public RBNode NIL;                           // NIL 节点

public static final boolean RED    = false;
public static final boolean BLACK  = true;
```

2. 构造函数：
   初始化 root 和 NIL
3. 左旋函数 leftRotate（右旋同理）：
   与书上代码相似，需要调整 size
4. insert 函数:
   插入函数分为 2 个,
   一个是作为 API,参数为 int;
   另一个内部调用,参数是 RBNode
   a）insert(int key)作为 API,外部调用.
   b） insert(RBNode node)内部调用,后者与书上代码相似.
5. insertFixUp 函数:
   与书上类似,需要补全另一种情况,注释很清晰.
6. delete 函数:
   与书上相似,需要注意 size 的调整.所移除的节点的祖先的 size 都要减 1
7. deleteFixUp 函数:
   与树上相似, 需要补全另一种情况,注释很清晰.
8. 遍历函数
   3 个函数,以 printPreOrder 为例:
   有一个供外部调用的 api 接口,有一个内部递归调用的内部函数.
9. osSelect 函数
   与书上相似

10. 其他辅助函数,比如 minimum,Transplant 等.

iii. 构建 main class – RedBlack 类:

1. field:
   其中需要说明的是,
   ifIntelliJ 表示是否工作与 IntelliJ 中.(除了最后为了提交是在 wsl 中,其他时间都是在 IntelliJ 中工作).
   CHECKMODE 表示是否为 check 模式,如果是,则每次 insert,delete 都会输出一遍树的信息.

```
private static boolean ifIntelliJ = false;
public static boolean CHECKMODE = false;
public static final int N = 100;
private static final String ABSPATH = "G:/PB15111604-project3/";

private static RBTree tree;
private static int[] originalInts ;
```

2. process 函数-对于每个 size 进行处理的主要函数:
3. check 函数,以及几个 print 函数:
4. select 函数和 partition 函数
   为了检查 delete 结果而设置.

   Select 类似树上的 $\text{RANDOMIZED-SELECT}(A, p, r, i)$ 而 partition 函数则与快排部分类似. $\text{PARTITION}(A, p, r)$

5. 输入输出.

3. Ex2 顺序统计树
   已经包含在了 Ex1 的叙述中.
4. 制图

5. 实验关键代码截图（结合文字说明）
最后的目录结构

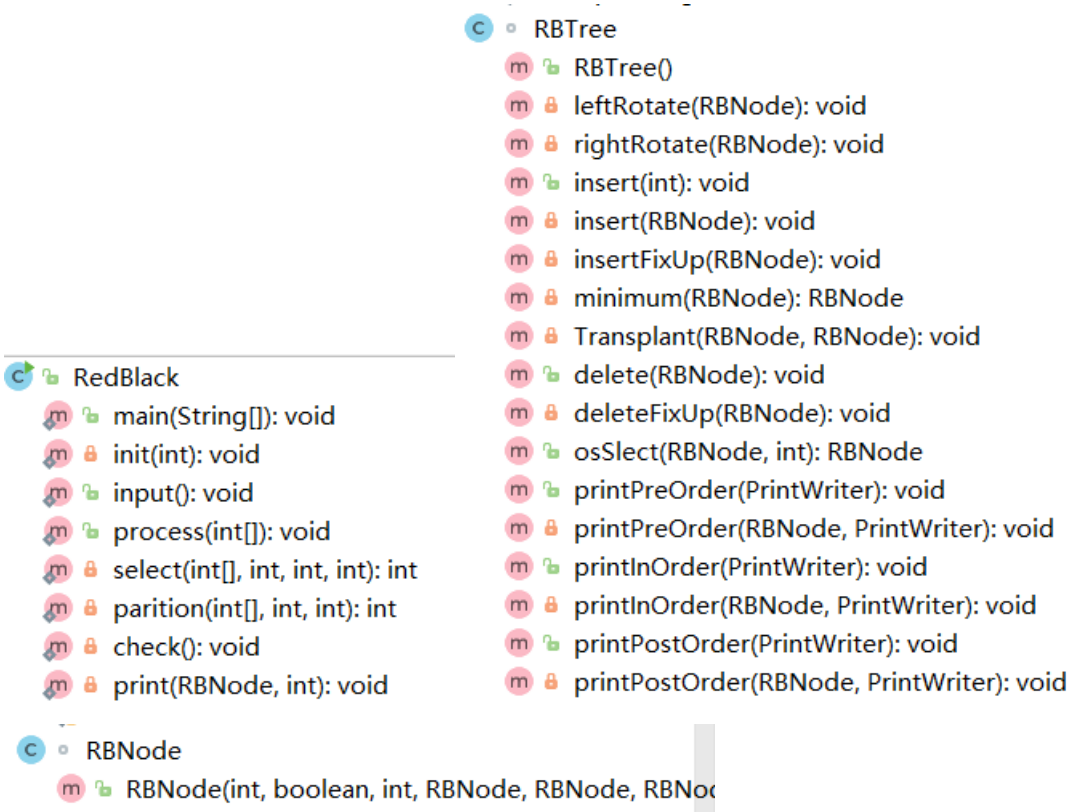
```
Reaper@KZ:/mnt/g/PB15111604-project3$ tree -L 2
.
├── input
│   └── input.txt
├── output
│   ├── size12
│   ├── size24
│   ├── size36
│   ├── size48
│   └── size60
├── PB15111604-金泽文-project3.doc
└── src
    ├── GenerateIntegers.java
    ├── Makefile
    ├── RBNode.class
    ├── RBTree.class
    ├── RedBlack.class
    └── RedBlack.java

8 directories, 8 files
```

```
Reaper@KZ:/mnt/g/PB15111604-project3$ tree output/size60
output/size60
├── delete_data.txt
├── inorder.txt
├── postorder.txt
├── preorder.txt
├── time1.txt
└── time2.txt

0 directories, 6 files
```

代码结构:



RedBlack::process(int[] array)

RedBlack.java | GenerateIntegers.java

```
79      // 针对每个size进行处理的主力函数.
80  @   public static void process(int[] array){
81          long start, end, tenStart = 0, tenEnd;
82          int n = array.length, j = 0;
83          long[] tenArray = new long[n/10];
84          // 开始计时
85          start = System.nanoTime();
86
87          // 初始化树
88          tree = new RBTree();
89          for(int i = 0; i < n; i++) {
90              if (i % 10 == 0)
91                  // 十个记一次
92                  tenStart = System.nanoTime();
93              // 插入i
94              tree.insert(array[i]);
95              check();
96              if ((i + 1) % 10 == 0) {
97                  tenEnd = System.nanoTime();
98                  tenArray[j++] = tenEnd - tenStart;
99              }
100         }
101         // 结束计时
102         end = System.nanoTime();
103
104         for(int i = 0; i < tenArray.length; i++){
105             printTime1.printf("%2d - %2d slice:  %9d nanoseconds.\n",
    10*i+1, 10*(i+1), tenArray[i]);
106             printTime1.flush();
107         }
108
109         printTime1.printf("Total building: %9d nanoseconds.\n", end -
110                 start);
111         printTime1.flush();
```

RedBlack.java

```
112
113         // 遍历输出
114         tree.printPreOrder(printPreOrder);
115         tree.printInOrder(printInOrder);
116         tree.printPostOrder(printPostOrder);
117
118         // 删除节点
119         int[] tmpDelete = {3,4};
120         for(int i : tmpDelete){
121             start = System.nanoTime();
122             RBNode node = tree.osSlect(tree.root,  i: n/i);
123             tree.delete(node);
124             end = System.nanoTime();
125
126             check();
127             printTime2.printf("Time to delete node %d/%d: %9d nanoseconds
    .\n", n, i, end - start);
128             printTime2.flush();
129             printDelete.printf("key:%5d, color:%s, size:%2d\n",
    node.key, (node.color == tree.RED ? "red   ":"black"), node
130                     .size);
131             printDelete.flush();
132
133             // 通过select(a,p,r,i)检查是否删除正确.
134             int s = select(array,  p: 0,  r: array.length-1,  i: n/i);
135             System.out.printf("%d/%d:\n", n, i);
136             System.out.printf("    Selected from the input data by
    select(a, p, r, i): %d\n", s);
137             System.out.printf("    Deleted from the RedBlack tree with
    osSelect:       %d\n", node.key);
138             n--;
139         }
140     }
```

对于每个 size,在 process 中构造红黑树,增加节点,删除节点,计时,输出,各种遍历,通过 select 比较来检查等.

注释比较清晰.

RBTree::leftRotate(RBNode x)

```
216         // 左旋,以x为开始的子树父节点
217     @   private void leftRotate(RBNode x){
218             // y是x的右孩子
219             RBNode y = x.right;
220 
221             // y的左孩子给x当右孩子
222             x.right = y.left;
223 
224             // 如果y有左孩子,新爸爸是x
225             if(y.left != NIL)
226                 y.left.parent = x;
227 
228             y.parent = x.parent;
229 
230             if(x.parent == NIL)
231                 root = y;               // 如果x是根,y成为根
232             else{
233                 if(x.parent.left == x)      // 如果x是左孩子
234                     x.parent.left = y;      // y成为左孩子
235                 else
236                     x.parent.right = y;     // 否则,y是右孩子
237             }
238 
239             y.left = x;
240             x.parent = y;
241             y.size = x.size;
242             x.size = 1;
243             x.size += x.left == NIL ? 0:x.left.size;
244             x.size += x.right == NIL ? 0:x.right.size;
245         }
```

需要注意调整 size,其他描述已在注释中清晰地给出.

RBTree::insert(int key)

```
278
279         // API - 插入大小为key的node
280         public void insert(int key) {
281             RBNode node = new RBNode(key, RED,
        SIZE: 1, PARENT: null, LEFT: null, RIGHT: null);
282             node.parent = NIL;
283             node.left = NIL;
284             node.right = NIL;
285             insert(node);
286         }
287
```

作为 API,以 int 为参数

RBTree::insert(RBNode node)

```
288         // 插入一个node,内部调用
289     @   private void insert(RBNode node){
290             RBNode y = NIL;
291             RBNode x = root;
292
293             int key = node.key;
294             while(x != NIL){
295                 y = x;
296                 y.size++;
297                 if(key < x.key)
298                     x = x.left;
299                 else
300                     x = x.right;
301             }
302
303             node.parent = y;
304             if(y != NIL){
305                 // 如果y不是根
306                 if(key < y.key)
307                     y.left = node;
308                 else
309                     y.right = node;
310             }
311             else
312                 // 如果是,那node就是根
313                 root = node;
314
315             insertFixUp(node);
316             // 修正
317         }
```

与书上类似,作为内部函数,添加 RBNode.

RBTree::insertFixUp(RBNode node)

RedBlack.java | GenerateIntegers.java

```java
    // insert函数调整颜色的的fixup
    private void insertFixUp(RBNode node){
        RBNode parent, gparent, uncle;

        while((parent = node.parent) != NIL && parent.color == RED){
            // parent是红色
            gparent = parent.parent;

            if(parent == gparent.left){
                // 如果parent是左孩子
                uncle = gparent.right;

                if(uncle != NIL && uncle.color == RED){
                    // uncle是红色,进入case1
                    parent.color = BLACK;        // case1
                    uncle.color = BLACK;         // case1
                    gparent.color = RED;         // case1
                    node = gparent;              // case1
                    continue;
                }
                else if(node == parent.right){
                    // uncle是黑色,node是右孩子,进入case2
                    leftRotate(parent);
                    node = parent;
                    parent = node.parent;
                }
                // uncle是黑色,node是左孩子,进入case3
                parent.color = BLACK;            // case3
                gparent.color = RED;             // case3
                rightRotate(gparent);            // case3
            }
            else{
                // 否则parent是右孩子
                uncle = gparent.left;

                if(uncle != NIL && uncle.color == RED){
                    // uncle是红色,进入case1
                    parent.color = BLACK;        // case1
                    uncle.color = BLACK;         // case1
                    gparent.color = RED;         // case1
                    node = gparent;              // case1
                    continue;
                }
                else if(node == parent.left){
                    // uncle是黑色,node是左孩子,进入case2
                    rightRotate(parent);
                    node = parent;
                    parent = node.parent;
                }
                // uncle是黑色,node是右孩子,进入case3
                parent.color = BLACK;            // case3
                gparent.color = RED;             // case3
                leftRotate(gparent);             // case3
            }
        }
        root.color = BLACK;
    }

    // 返回以root节点为根节点的树的最小节点.
```

3 个 case 已标注.
描述信息在注释中.

RBTree::delete(RBNode node)

RedBlack.java | GenerateIntegers.java

```java
    // API - 删除节点node
    public void delete(RBNode node){
        RBNode y = node;
        RBNode x;
        boolean originalColor = node.color;

        if(node != NIL && node.left == NIL){
            // node最多有一个右孩子
            x = node.right;
            Transplant(node, x);
        }
        else if(node != NIL && node.right == NIL) {
            // node最多有一个左孩子
            x = node.left;
            Transplant(node, x);
        }
        else{
            // node有两个孩子
            y = minimum(node.right);
            // y是node后继,将被换上来.
            originalColor = y.color;
            x = y.right;
            if(y.parent == node)
                x.parent = y;
            else{
                Transplant(y, x);
                y.right = node.right;
                y.right.parent = y;
            }
            Transplant(node, y);
            y.size = node.size;
            y.left = node.left;
            y.left.parent = y;
            y.color = node.color;
        }

        // 调整size
        RBNode tmp = x.parent;
        while(tmp != NIL){
            tmp.size--;
            tmp = tmp.parent;
        }
        // 如果原颜色为黑色,修补
        if (originalColor == BLACK)
            deleteFixUp(x);
    }

    // delete函数调整颜色的fixup
    private void deleteFixUp(RBNode node){...}

    public RBNode osSlect(RBNode x, int i){
        int r = x.left.size + 1;
        if(i == r)
            return x;
        else if(i < r)
            return osSlect(x.left, i);
        else
            return osSlect(x.right, i - r);
    }

    // API - 先序遍历
    public void printPreOrder(PrintWriter pw){
        printPreOrder(root, pw);
    }
```

RBTree::deleteFixUp(RBNode node)

RedBlack.java | GenerateIntegers.java

```
444     // delete函数调整颜色的fixup
445     private void deleteFixUp(RBNode node){
446         RBNode sibling, parent = node.parent;
447         while (node != root && (node.color == BLACK )){
448             if(node == parent.left){
449                 // node是左孩子
450                 sibling = parent.right;                     // case1
451                 if(sibling.color == RED){
452                     // 兄弟红色,进入case1
453                     sibling.color = BLACK;                  // case1
454                     parent.color = RED;                     // case1
455                     leftRotate(parent);                     // case1
456                     sibling = parent.right;                 // case1
457                 }
458                 if(sibling.left.color == BLACK && sibling.right.color == BLACK){
459                     // 兄弟黑色,兄弟双子黑色,进入case2
460                     sibling.color = RED;                    // case2
461                     node = parent;                          // case2
462                     parent = parent.parent;                 // case2
463                 }
464                 else{
465                     if(sibling.right.color == BLACK){
466                         // 兄弟黑色,兄弟右孩子黑色,进入case3
467                         sibling.left.color = BLACK;         // case3
468                         sibling.color = RED;                // case3
469                         rightRotate(sibling);               // case3
470                         sibling = parent.right;             // case3
471                     }
472                     // 兄弟红色,兄弟右孩子红色,进入case4
473                     sibling.color = parent.color;           // case4
474                     parent.color = BLACK;                   // case4
475                     sibling.right.color = BLACK;            // case4
476                     leftRotate(parent);                     // case4
477                     node = root;                            // case4
478                 }
479             }
```

RBTree › deleteFixUp()

RedBlack.java

```
479         }
480         else{
481             // node是右孩子
482             sibling = parent.left;
483             if(sibling.color == RED){
484                 // 兄弟红色,进入case1
485                 sibling.color = BLACK;                      // case1
486                 parent.color = RED;                         // case1
487                 rightRotate(parent);                        // case1
488                 sibling = parent.left;                      // case1
489             }
490             if(sibling.right.color == BLACK && sibling.left.color == BLACK){
491                 // 兄弟黑色,兄弟双子黑色,进入case2
492                 sibling.color = RED;                        // case2
493                 node = parent;                              // case2
494                 parent = parent.parent;                     // case2
495             }
496             else{
497                 if(sibling.left.color == BLACK){
498                     // 兄弟黑色,兄弟左孩子黑色,进入case3
499                     sibling.right.color = BLACK;            // case3
500                     sibling.color = RED;                    // case3
501                     leftRotate(sibling);                    // case3
502                     sibling = parent.left;                  // case3
503                 }
504                 // 兄弟红色,兄弟左孩子红色,进入case4
505                 sibling.color = parent.color;               // case4
506                 parent.color = BLACK;                       // case4
507                 sibling.left.color = BLACK;                 // case4
508                 rightRotate(parent);                        // case4
509                 node = root;                                // case4
510             }
511         }
512     }
513     node.color = BLACK;
514 }
```

RBTree › deleteFixUp()

类似.

RBTree::printPreOrder(PrintWriter pw)
RBTree::printPreOrder(RBNode node, PrintWriter pw)

```
526     // API - 先序遍历
527     public void printPreOrder(PrintWriter pw){
528         printPreOrder(root, pw);
529     }
530
531     // 先序遍历
532     private void printPreOrder(RBNode node, PrintWriter pw){
533         if(node == NIL)
534             return;
535         pw.printf("key:%5d, color:%s, size:%2d\n", node.key, (node
    .color == RED ? "red  ":"black"), node.size);
536         pw.flush();
537         printPreOrder(node.left, pw);
538         printPreOrder(node.right, pw);
539     }
```

3 种遍历相似,以先序说明.
上面是 API,下面是内部调用

RBTree::osSlect(RBNode x, int i)

```
516     public RBNode osSlect(RBNode x, int i){
517         int r = x.left.size + 1;
518         if(i == r)
519             return x;
520         else if(i < r)
521             return osSlect(x.left, i);
522         else
523             return osSlect(x.right,  i: i - r);
524     }
525
```

与 14 章代码类似.

RedBlack::select(int[] a, int p, int r, int i)

```
142     // select(a, p, r, i)
143     private static int select(int[] a, int p, int r, int i){
144         if(p == r)
145             return a[p];
146         int q = parition(a, p, r);
147         int k = q - p + 1;
148         if (i ==k){
149             return a[q];
150         }
151         else if (i < k)
152             return select(a, p,  r: q-1, i);
153         else
154             return select(a,  p: q+1, r,  i: i-k);
155     }
156
```

select 类似树上的 $\text{RANDOMIZED-SELECT}(A, p, r, i)$

RedBlack::partition(int[] a, int p, int r)

```
157     // 快排对应的partition函数
158 @   private static int parition(int[] a, int p, int r){
159         int x = a[r];
160         int i = p-1;
161         int tmp;
162         for(int j = p; j <= r-1; j++) {
163             if (a[j] <= x){
164                 i++;
165                 tmp = a[i];
166                 a[i] = a[j];
167                 a[j] = tmp;
168             }
169         }
170         tmp = a[i+1];
171         a[i+1] = a[r];
172         a[r] = tmp;
173         return i+1;
174     }
```

partition 函数则与快排部分 $\text{PARTITION}(A, p, r)$ 类似.

6. 实验结果、分析（结合相关数据图表分析）

//在实验过程中为了减少硬件 cache 策略对分析的影响,所以在给定的 size 之外,
//我首先跑了一个 size100,来避免对后面 60,48,36,24,12 的影响.

这是初步得到的数据

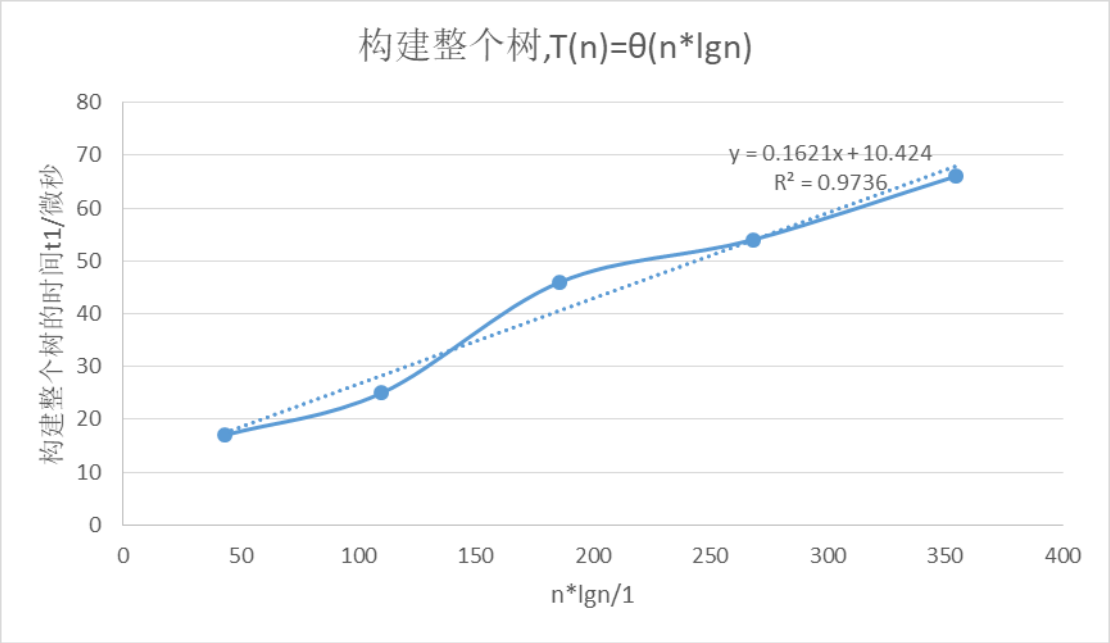
size\time	构建整个树的时间 t1/微秒	插入 10 个节点平均时间 t2/微秒	插入 1 个节点平均时间 t3/微秒	删除第 n/3 个节点的时间 t4/微秒	删除第 n/4 个节点的时间 t5/微秒
12	17	5	0.5	4	4
24	25	6	0.6	4	5
36	46	6.7	0.67	4	5
48	54	7.5	0.75	5	5
60	66	8.5	0.85	5	6

构建整棵树

根据表格:

nlgn	构建整个树的时间 t1/微秒
43.01955	17
110.0391	25
186.1173	46
268.0782	54
354.413436	66

得到如下拟合曲线:



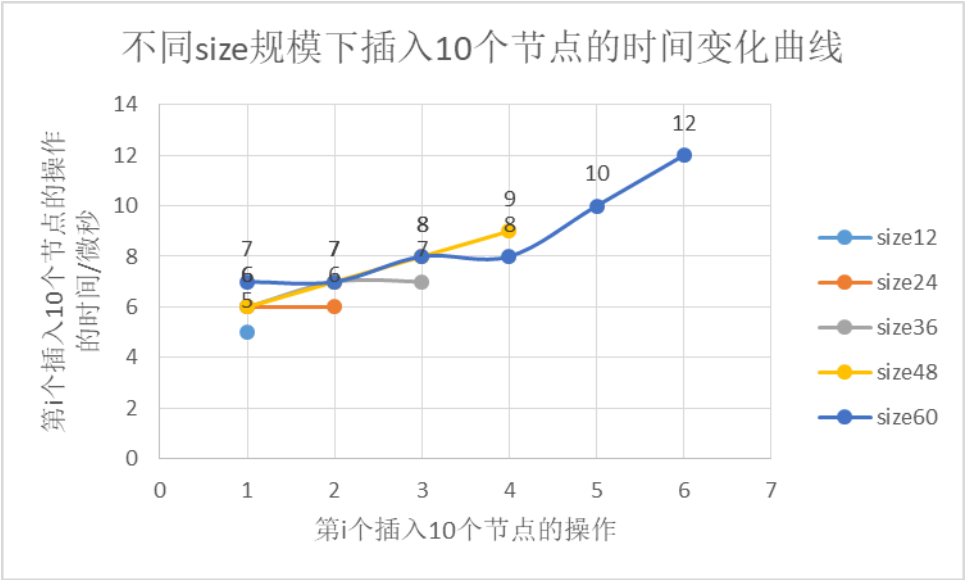
R^2=0.9736,T(n)和 n*lgn 基本呈线性关系,比较符合 θ(n*lgn)的渐进时间复杂度.

插入节点

根据表格 time1.txt,得到:

size\插入每10个的时间(微秒)	1	2	3	4	5	6
12	5					
24	6	6				
36	6	7	7			
48	6	7	8	9		
60	7	7	8	8	10	12

得到:



可以看出,

对于不同 size,在相同 i 时略有差异

对于每个 size,随着 i 的增长,时间增长

可以推断,当数据更多一点,偶然误差减小时,相同规模下,随着 i 的增长,时间的增长也会稳步进行.

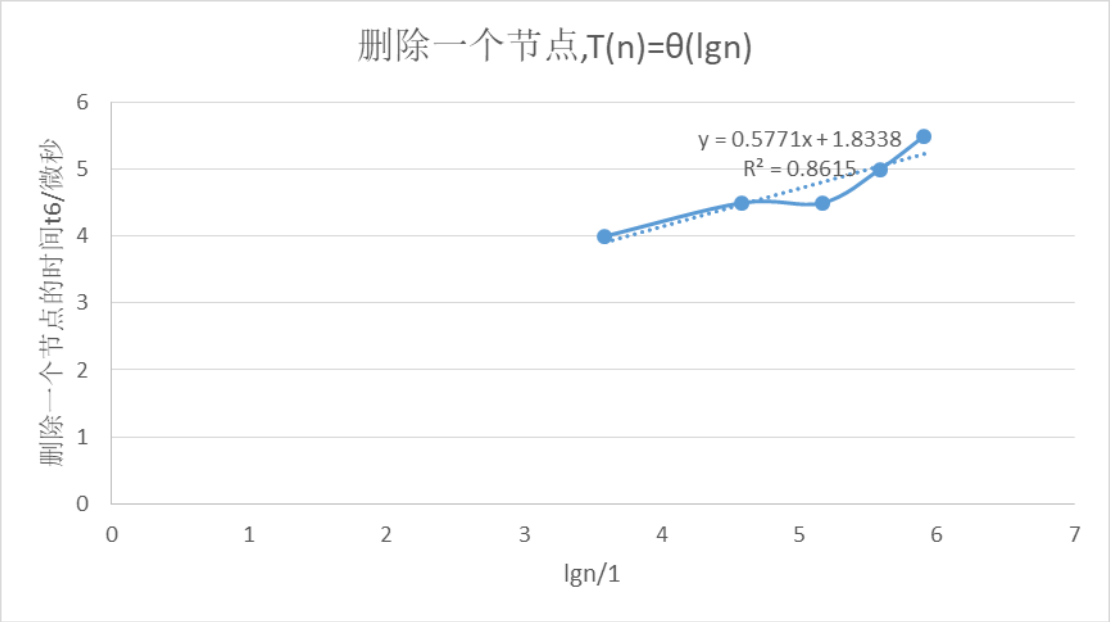
删除节点

忽略删除一个之后 n—的变化.

根据表格

lgn	删除 1 个节点平均时间 t6/微秒
3.5849625	4
4.5849625	4.5
5.169925	4.5
5.5849625	5
5.9068906	5.5

得到如下拟合曲线:



R^2=0.0.8615,R^2 比较小,拟合效果没有上面好.

原因是这个时间接近了硬件所能测量的时间最小间隔,误差想比上面更大,如果也能删除 10 个测一个时间,再取平均,就会效果好很多.

但这是删除一个测一个时间,所以误差会很大.

7. 实验心得

a) 首先，通过本次实验，加深了红黑树的理解，这是最重要的。尤其是通过切身地动手实现各个操作，发现了很多以前没有考虑到的细节。这是最大的收获。
b) 其次，通过本次实验，发现了红黑树的重要性，以及优秀的性能。
c) 再其次，通过本次实验，我对 java 有了更进一步的理解，得到了很多细节性的认识。尤其是对 IntelliJ 的工程结构的理解，中间因为这个花了很多时间，但确实 IntelliSense 在生产过程中还是很有效用的！
d) 最后，和上一次还有上上一次一样，感谢可爱哒助教读到这里，感谢的同时心疼一下。。。
e) 算法很美很重要，要加深理解与思考！
f) 祝好！

g) 马上就考试了,嘤嘤嘤.