实验三：红黑树和顺序统计树

PB15111672 林郅琦

实验要求

实验1：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）

实验2：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。

实验环境

- 系统环境：MacOS High Sierra
- 编译环境：g++ 4.2.1（require C++14）
- 机器内存：8G
- 时钟主频：2.5 GHz

实验过程

- 随机数据生成

  由于红黑树和顺序统计树要求树中每个 key 值均不同，因此生成的60个随机数中也要求两两不相同。此处 key 作为整数，可以先生成一个顺序数组 `std::vector<int>` , 其值为1，2，3，...，60。 然后使用 C++ 提供的 `std::shuffle(...)` 函数对数组进行随机洗牌，最后获得打乱后的数组输入到文件中。

- 红黑树和顺序统计树的数据结构实现

  首先，本次实验对红黑树的实现和顺序统计树的实现均采用 C++ 模板类的形式，提供了以下基本操作：

  - 获得根节点（返回结果 const)
  - 查找（返回结果 const）
  - 插入
  - 删除
  - 打印树（层次遍历）
  - 前序遍历、中序遍历、后序遍历
  - 排名
  - 选择排名

  在实现的时候不能认为 T.nil 哨兵和 nullptr 等同，使用哨兵的好处可以大量避免非法访问，因此，此类中将有一个（仅有一个）哨兵充当叶子节点。由于算法导论课本上已经给了以上大部分函数详细的注解，在此不再展开讨论。此处仅讨论打印树的实现过程：打印树的实现过程采用层次遍历，算法实现采用的数据结构为队列，并维护每一层的节点个数和当前层对应的下一层的节点个数。典型的一个节点的表示为 `/key(bit | rank)\` ，其中 '/' 表示这个节点有左孩子，'\'

表示这个节点有右孩子，key 表示这个节点的值，bit = 0 表示这个节点为红色，bit = 1 表示这个节点为黑色。

- 验证程序-中位数Select 算法

  需要验证顺序统计数的选择排名后结果的正确性，因此采用中位数 Select 算法来验证结果。里面采用快速排序所使用的 Partition 算法，通过递归来计算出结果。由于算法导论课本上已经给出了详细的注解，在此不再展开详细说明。

- 验证红黑树基本操作

  由于已经实现了打印操作，因此可以很主观的看出红黑树在插入、删除后是否出现问题。为了验证插入、删除操作的正确性，与随机数据生成类似，先生成一个1，2，3，...，10 的数组，然后进行10次迭代，每次迭代使用一下操作：

  - 对数组进行随机洗牌后，从第一个开始一个个插入
  - 再对数组随机洗牌，从第一个开始一个个删除
  - 观察插入前后和删除前后树的情况，来验证操作的正确性

  多运行几次程序，并多观察几次程序，就能遍历插入删除的各种不同情况，随即能验证程序的正确性。

实验关键代码截图

- 随机数生成-random shuffle, 此函数位于头文件 lzq.h （以我的名字首字母命名的头文件）中

```
template<typename T>
void VecShuffle(std::vector<T>& vec){
    std::shuffle(std::begin(vec), std::end(vec), rng_);
}
```

- 红黑树和顺序统计树-树打印操作：

```
template<typename T>
void RedBlackTree<T>::print(){
    if(root_ == leaf_){
        std::cout << "empty redblack tree" << std::endl;
        return;
    }
    std::queue<RBTNode<T>* > que;
    if(root_ != leaf_)
        que.push(root_);
    std::size_t curr_layer_num = 1;
    std::size_t next_layer_num = 0;
    while(!que.empty()){
        auto tmp = que.front();
        que.pop();
        if(tmp->lchild_ != leaf_){
            std::cout << '/';
            que.push(tmp->lchild_);
            ++next_layer_num;
```

```
        }
        std::cout << tmp->key_ << '(' << (int)tmp->color_  << " | " <<
tmp->size << ')';
        if(tmp->rchild_ != leaf_){
            std::cout << '\\';
            que.push(tmp->rchild_);
            ++next_layer_num;
        }
        std::cout << '\t';
        --curr_layer_num;
        if(curr_layer_num == 0){
            std::cout << std::endl;
            curr_layer_num = next_layer_num;
            next_layer_num = 0;
        }
    }
}
```

其中 `root_` 为根节点，`RBTNode<T>*` 为红黑树的一个节点指针，他有成员 `key_`，`parent_`，`lchild_`，`rchild_`，`size`。`leaf_` 即为哨兵。

- 验证程序-Select

```
int Partition(int* array, int begin, int end){
    /* Partition the array from [begin, end] */
    int x = array[end];
    int i = begin - 1;
    for(int j = begin; j < end; ++j){
        if(array[j] < x){
            ++i;
            std::swap(array[i], array[j]);
        }
    }
    std::swap(array[i + 1], array[end]);
    return i + 1;
}

int Select(int* array, int begin, int end, int rank){
    if(begin == end)
        return array[begin];
    int q = Partition(array, begin, end);
    int k = q - begin + 1;
    if(rank == k)
        return array[q];
    else if(rank < k)
        return Select(array, begin, q - 1, rank);
    else
        return Select(array, q + 1, end, rank - k);
}
```

实现方式与算法导论上基本一致，因此原理不再展开描述。

实验结果、分析

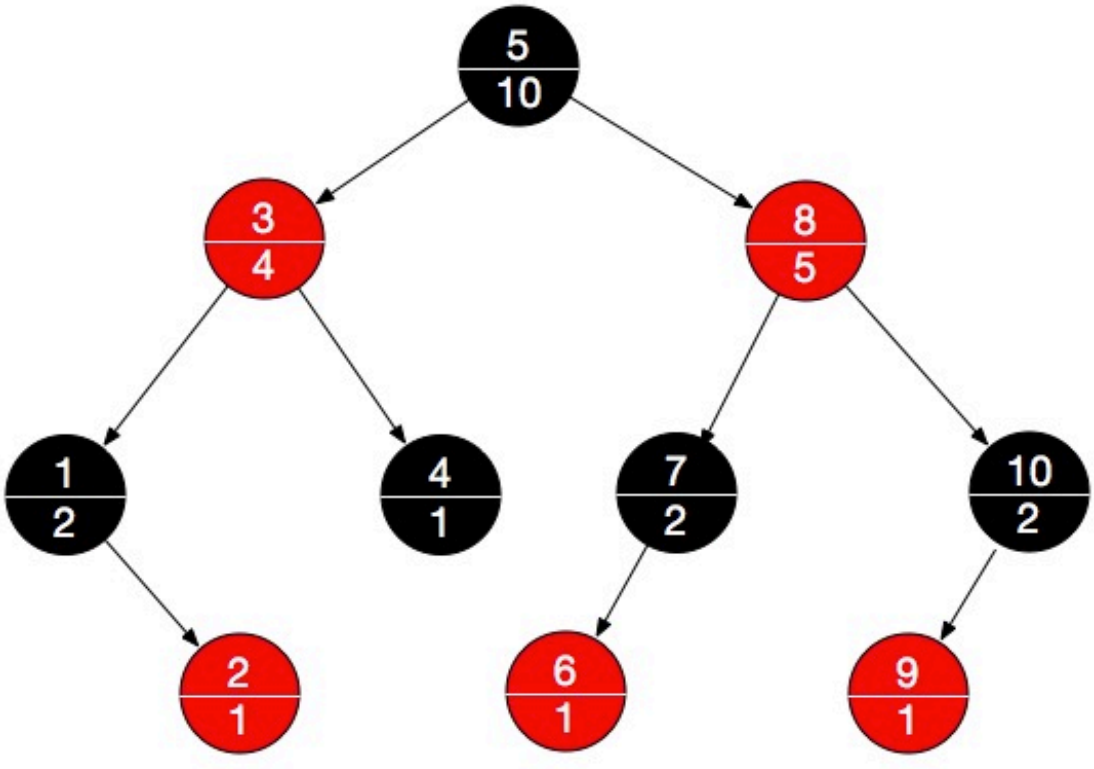
实验结果经过比对和输出验证，没有出现问题。

- 树的打印示意（接口为 print() ）

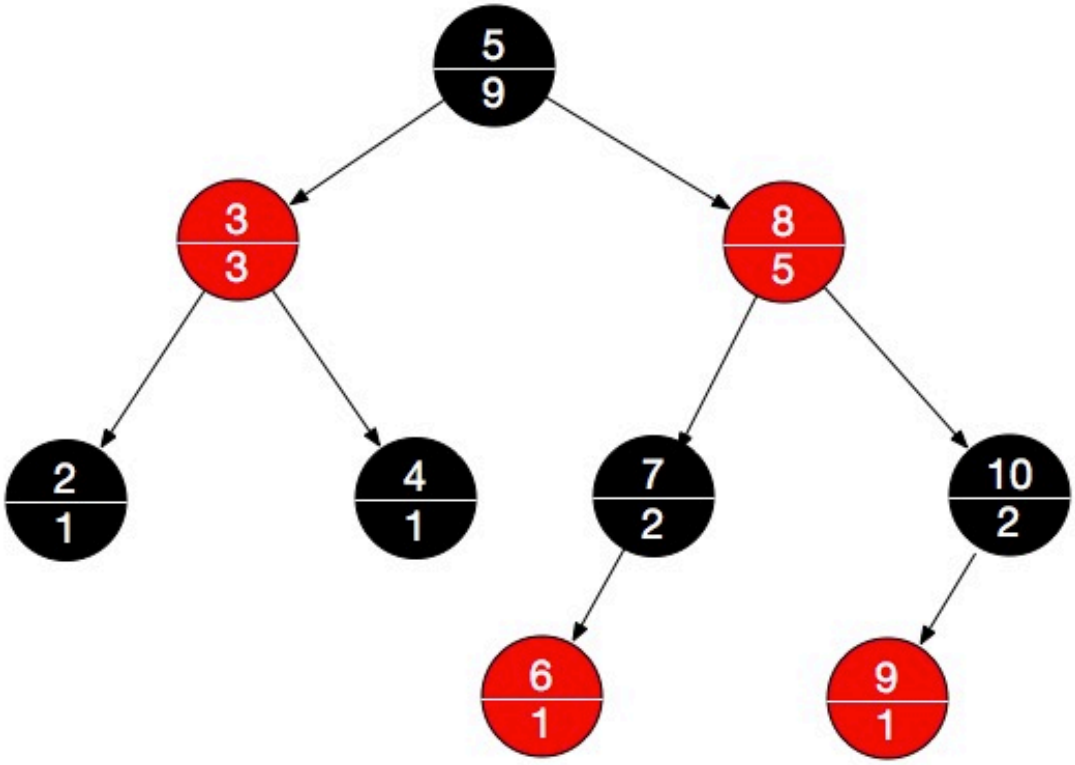
```
Insert key array: 3,1,4,8,5,7,10,2,6,9,
/5(1 | 10)\
/3(0 | 4)\      /8(0 | 5)\
1(1 | 2)\       4(1 | 1)       /7(1 | 2)       /10(1 | 2)
2(0 | 1)        6(0 | 1)       9(0 | 1)
Delete key: 1
/5(1 | 9)\
/3(0 | 3)\      /8(0 | 5)\
2(1 | 1)        4(1 | 1)       /7(1 | 2)       /10(1 | 2)
6(0 | 1)        9(0 | 1)
```

从数组中构建出一棵红黑树的结果如下：（根据终端打印输出结果）

典型的一个节点的表示为 `/key(bit | rank)\` ，其中 '/' 表示这个节点有左孩子，'\' 表示这个节点有右孩子，key 表示这个节点的值，bit = 0 表示这个节点为红色，bit = 1 表示这个节点为黑色。



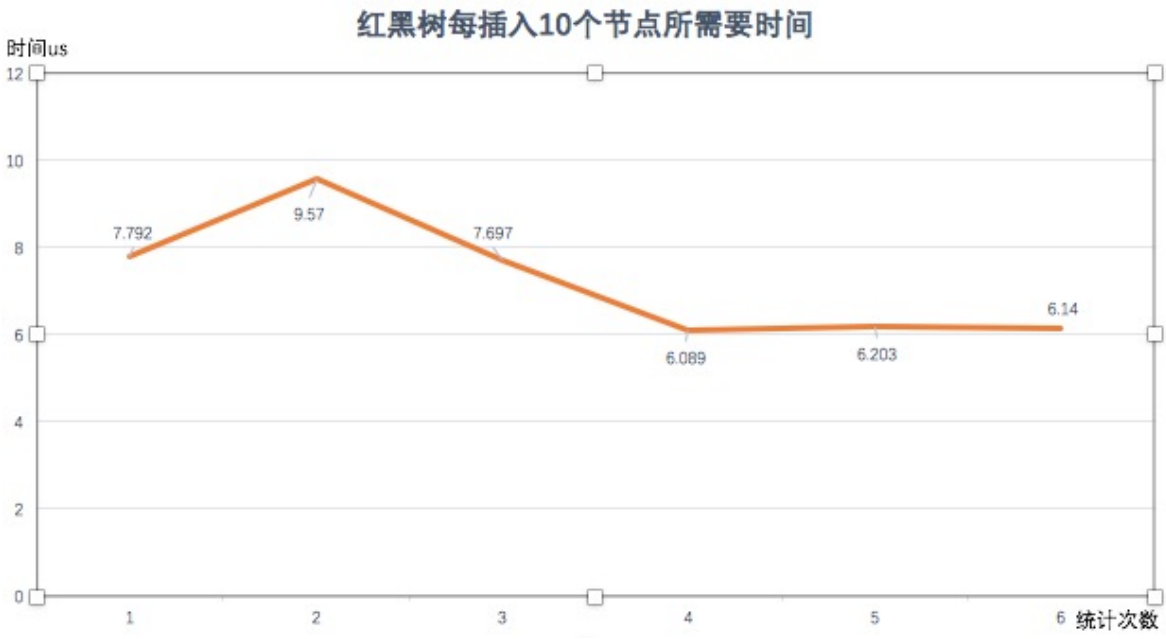
之后删除了key 值为1的节点，结果为下图：（根据终端打印输出信息）



由此可以看见在这个例子中是正确的。

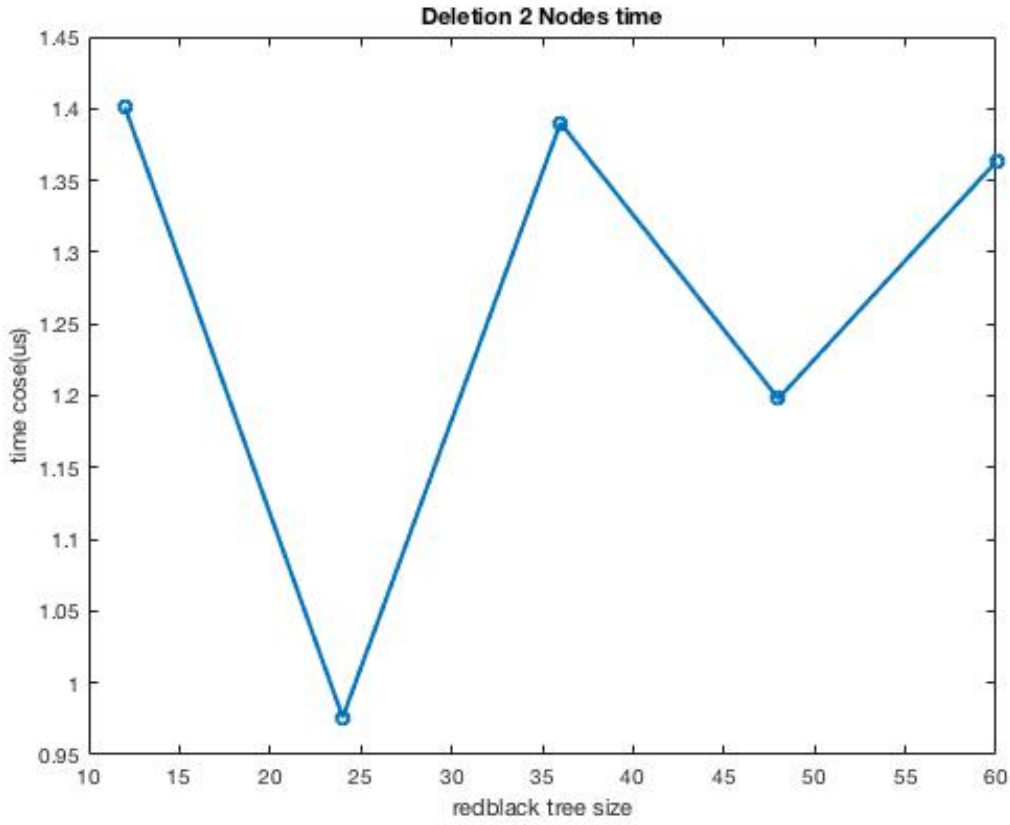
通过随机以及大量大观察，插入和删除操作没有出现任何异常。

- 实验1实验结果



从实验结果看出，随着 n 逐渐增大，插入的时间却反而下降，不过总体时间接近。原因可能是：插入操作（不包括 Fixup）所需的时间开销很小，主要集中在 fixup 上，而 fixup 的时间以插入节点时周围的环境来决定，如果环境好，可能不需要 fixup，如果环境不好，最差要 O（logn）的时间进行插入。因此，有一定的随机性，从而时间分布也具备一定的随机性。

- 实验2实验结果



从实验结果可以看出，删除操作也有一定的波动，由于删除操作核心耗时也位于 fixup 上，而每次删除的节点所需要的 fixup 时间也会不同，所以结果必定带有随机波动性。

- 验证程序-中位数 Select 结果

  通过从 input.txt 中读取数据再选择，与 OS-SELECT 算法所得结果（OS-SELECT 算法所得结果在 output/ 文件夹中的 delete_data.txt 中）比较，正确。

```
zhiqideMacBook-Pro:source zhiqilin$ ./check
Selecting for data size 12: 16,8
Selecting for data size 24: 23,18
Selecting for data size 36: 21,18
Selecting for data size 48: 20,16
Selecting for data size 60: 20,15
```

实验心得

- 本次实验能让我非常清楚的了解了红黑树的基本操作，以及哨兵对于红黑树的重要性。
- 在调试过程中我更熟悉了红黑树插入、删除所面对的不同情况。
- 通过对红黑树的拓展获得顺序统计树，也加深了我对红黑树拓展的领会。

源码目录说明

source 文件夹下：

- redblacktree.h: 实现红黑树和顺序统计树的C++模板文件，所有接口都在这里定义
- datagen.cc: 用于生成实验随机数据的文件

- select.cc: 作为验证顺序统计树的选择结果的中位数选择验证算法。
- test.cc: 作为验证红黑树基本插入删除操作正确性的文件。
- rbtree.cc: 用于实验1和实验2生成结果文件。
- lzq.h: 本人为自己常用函数所写的库。内含随机数生成、计时工具、线程安全队列等。
- Makefile：可以自动构建上面的3个 *.cc 文件，生成可执行文件 check, datagen, test, rbtree
    - datagen: 生成输入数据
    - rbtree: 生成所有输出结果文件
    - test：验证红黑树基本插入删除操作
    - check: 验证顺序生成树 Select 算法正确性
    - 注: `make clean` 可以删除所有可执行文件