lab3 红黑树和区间树

实验内容及要求

■实验3.1：红黑树

□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（K1, K2, K3, ……, Kn），以这 n 个正整数作为结点的关键字，向一棵初始空的红黑树中依次插入n 个节点，统计算法运行所需时间，画出时间曲线。

□随机删除红黑树中n/4个结点，统计删除操作所需时间，画出时间曲线图。

■实验3.2：区间树

□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。

实验设备和环境

- 编译运行环境
  - Windows10-mingw-w64
  - Clion 2020.2.4
  - vscode
- 硬件
  - 处理器: 英特尔 Core i7-8750H @ 2.20GHz 六核
  - 速度 2.21 GHz (100 MHz x 22.0)
  - 处理器数量 核心数: 6 / 线程数: 12
  - 一级数据缓存 6 x 32 KB, 8-Way, 64 byte lines
  - 一级代码缓存 6 x 32 KB, 8-Way, 64 byte lines
  - 二级缓存 6 x 256 KB, 4-Way, 64 byte lines
  - 三级缓存 9 MB, 12-Way, 64 byte lines
  - 内存: 海力士 DDR4 2666MHz 8GB

实验方法和步骤

文件读写

- 考虑使用fsteam头文件下的ofstream以及ifstream流来轻松实现文件读写。
- 打印时利用`<iomanip`来格式化输出
- 需注意文件目录组织下，采用相对路径较为简单，个人因为使用Clion缘故，其运行时当前目录为cmake-build-debug,所以文件路径设置为

  `"..\\..\\input\\input.txt"`等
- 此外，Clion不支持中文路径，需要注意
- 在使用vscode时，则需换成绝对路径。

计时

- 采用网上推荐的格式，在经过试验后，发现微秒级的计时即可让结果显示较为方便。
- 考虑采用<windows.h>头文件下一个us级的计时方式

```
//clock计时参数的声明
LARGE_INTEGER nFreq;
LARGE_INTEGER t1;
LARGE_INTEGER t2;
double dt;
```

```
QueryPerformanceFrequency(&nFreq);
QueryPerformanceCounter(&t1);
具体需要计时的操作;//矩阵链乘计算
QueryPerformanceCounter(&t2);
dt =(t2.QuadPart-t1.QuadPart)/(double)nFreq.QuadPart;
time=dt*1000000;
```

- 实验结果发现此时计时精度较高，每个数据规模下均有较为合理的测量结果。

红黑树

tree.h

- 考虑构造两个类：红黑树类以及结点类，前者实现各种方法并有一个根节点root以及一个哨兵结点nil
- 哨兵结点也是一个结点，不过key置为-1，color置为BLACK，其余随意
- ```
  class Node
  {
  public:
      int key;//关键字
      int color;//0是黑，1是红
      Node* left;//左子树
      Node* right;//右子树
      Node* p;//父亲结点
  };

  class RB_Tree
  {
  public:
      Node* root;//红黑树的根节点
      Node* nil;//红黑树的哨兵结点

      RB_Tree(Node* root,Node* nil);//构造函数
      void inorder_RB_walk(Node* x,ofstream &f);//中序遍历
      Node* RB_search(Node* x,int k);//从树根开始查找
      Node* RB_maximum(Node* x);//找最大
      Node* RB_minimum(Node* x);//找最小
      Node* RB_successor(Node* x);//找后继
      Node* RB_predecessor(Node* x);//找前驱
      void left_rotate(Node* x);//左旋
      void right_rotate(Node* x);//右旋
      //插入
      void RB_insert(Node* z);
      void RB_insert_fixup(Node* z);
  ```
  ```

```
    //删除
    void RB_transplant(Node* u,Node* v);//用于将另一棵子树替换一棵子树并成为其双亲的孩子结点
    void RB_delete(Node* z);
    void RB_delete_fixup(Node* x);//用于通过改变颜色和执行旋转来恢复红黑性质
};
```

- 把书上所讲的红黑树的有关算法均作了实现, 各个方法的名字及功能均与书上一一对应。

tree.cpp

- 主要是对`tree.h`里的红黑树类`RB_Tree`的各个方法完成其实现，下面对几个重点方法做说明
- `void inorder_RB_walk(Node* x,ofstream &f);//中序遍历`
  - 考虑到中序遍历方法的递归性以及要求输入文件的格式，考虑在main.cpp里调用此方法时，传入对应的输入文件的流来完成文件输入`f<<x->key<<" ";`
- `void RB_insert(Node* z);`与`void RB_insert_fixup(Node* z);`
  - 主要参考书上算法实现，不过感觉算法的核心还是理清楚红黑树在原来叶结点处插入一个新结点后，可能导致整个红黑树部分性质被破坏的几种情况，捋清楚后再对各个情况具体实现就很容易了，而且也利于构造样例来验证。
- `void RB_delete(Node* z);`和`void RB_delete_fixup(Node* x);`
  - 同样，对12章二叉搜素树中删除的四种case以及13章里对删除后可能导致的红黑树性质被破坏的四种case作理论以及例子的学习后，完成算法并不困难
  - 当然还需要注意书上省略的对称结构部分的代码的补全，以及书上代码缩进有点错误，把RB_delete_fixup里case3和case4的情况写混淆了，需要自己做一定修改。

main.cpp

- 完成各种随机数的生成，以及构造红黑树类实例并完成各种操作，计时，文件读写。
- 声明了几个全局变量，便于复用以及减少了函数传参的复杂性

```
int N[5]={20,40,60,80,100};//整数n
vector<int> data;//存放随机生成的n个互异正整数
vector<Node> tree_node;//存放以n个正整数作为结点关键字的红黑树结点
```

- 随机生成函数

```
int random_generate(int n,int low, int high,ofstream &write_file){
//随机生成n个互异的正整数，范围在[low,high],按生成顺序放入data容器并写入文件
```

```
int random_delete(int n,int low, int high,ofstream &write_file)
```

以上两个函数大体内容一致，其功能见上面注释,主要函数体如下

```
srand((unsigned)time(NULL));//随机数生成
for(int i = 0; i < n;i++ ){
    temp=rand()%(high-low+1)+low;//限定随机生成的数的范围
    auto iter = find(data.begin(), data.end(), temp);
    if (iter != data.end()){
        i--;
        continue;
    }
    data.push_back(temp);
}
```

  - 为了生成互异的n个数，考虑用迭代器的find方法来比较每次新生成的数是否在容器内已经存在
  - 上面两个函数的区别在于写回文件的内容不一致，一个写入input.txt,后者写入delete_data.txt
- main函数的整体执行流程即在主for循环里，每次对一个量级n的数据生成新的Node结点并存入vector，然后作插入操作，中序遍历，再随机生成n/4的待删除结点，再删除之，最后做一次中序遍历。‘
- 需注意到，随机删除的n/4个结点是通过随机生成数组下标来实现的，因为插入红黑树的结点依照随机生成的顺序存放在tree_node这个vector里。

区间树

- 基本的方法实现等与红黑树一致，下面主要讨论每个文件里的一些新的补充或修改

tree.h

- ```
  class Node
  {
  public:
      int key;
      int interval[2];//0对应low,1对应high
      int max;//以x为根的子树中所有区间的端点的最大值
      int color;//0是黑，1是红
      Node* left;
      Node* right;
      Node* p;
  };
  ```

  - 增加了max域以及interval域
  - 考虑nil结点 max=0，interval={0,0}, key置为-1，color置为BLACK，其余随意

tree.cpp

- 对RB_delete、RB_insert、left_rotate、right_rotate共四个函数做了修改，主要是为了传播max属性的值
  - 新加了`int maxnum(int x,int y,int z)`函数，求三个数中的最大值
  - 旋转操作里，考虑到旋转前后，仅x和y结点的max域可能发生改变，所以只需做两行补充，当然需要注意，对两个结点max域的修改要按从下往上的顺序。

    ```
    //更改x和y的max属性
        x->max=maxnum(x->interval[1],x->left->max,x->right->max);
        y->max=maxnum(y->interval[1],y->left->max,y->right->max);
    ```
    ```

- 在插入操作里，考虑到有结点新加到树底部所以应该自下而上传播max域，因为可能新加的结点的max域最大。从根到该新加结点的路径上所有结点的max域都可能改变。

```
//设置max属性并向上传播
    z->max=z->interval[1];//因为z左右子树都是nil结点
    Node* temp=z->p;
    while(temp!=nil){
        temp->max=maxnum(temp->interval[1],temp->left->max,temp->right->max);
        temp=temp->p;
    }
```

- 在删除操作里，因为有结点被删除，所以从根到该删除结点的路径上所有结点的max域都可能改变，尽管删除结点里会导致树结构的一些变化，但是在参考书p167页对于二叉搜索树删除的四种情况的图的分析后，发现都可以找到一个x结点是变化的起始，所以只需从该结点往上一直到根的路径上对每个结点的max域做修改即可。代码同插入操作。
- 对RB_insert_fixup和RB_delete_fixup不需要作处理，因为这两个函数主要是通过改变红黑树中结点颜色以及左旋右旋操作来完成对红黑树性质的恢复，所以只需在旋转操作函数里完成修改即可。

- `Node* interval_search(int i[2]);`：新增search函数，对区间查找
  - 理解清楚书上所讲的两个区间重叠的概念及其判定准则即可较为简单的实现。

main.cpp

- 全局数据结构

```
vector<int> ldata;//存放区间左端点
vector<int> rdata;//存放区间右端点
vector<Node> tree_node;//存放以n个正整数作为结点关键字的红黑树结点
vector<int> data;
```

- 主要讨论对于随机生成区间的实现
  - `void random_lgenerate(int n,int low, int high)`

    随机生成n个互异的，范围在[0,24]或[30,49]的左端点值。随机以及互异的实现与红黑树中随机函数一致

    主要加了对生成[25,30)间数字时的判定与剔除

    调用:`random_lgenerate(n,0,49);`
  - `void random_rgenerate(int n,int low,int high)`

    按顺序依次生成n个与左端点值对应的右端点值，需要做一定判定，看生成的数是否满足区间形成的要求

```
temp=rand()%(high-low+1)+low;
if(temp > ldata[i] && temp<=25){//区间两端点在[0,25]
    rdata.push_back(temp);
}
else if(temp > ldata[i] && ldata[i]>=30){//区间两端点在[30,50]
    rdata.push_back(temp);
}
else{
    i--;
    continue;
}
```

调用：`random_rgenerate(n,1,50);`

- `int random_delete(int n,int low, int high,ofstream &write_file)` 函数与红黑树中基本一致，只是输入到delete_data.txt的数据格式不同

- main()函数的运行流程主要是随机生成30个区间，并以此生成结点存储在tree_node这个容器里。再完成插入操作，以及中序遍历操作(需要修改输入文件的格式)；

  对查找操作：随机生成的代码较为简单,主要有一个(25,30)内的区间需要单独生成。

```
random_lgenerate(2,0,49);
random_rgenerate(2,1,50);
ltemp=rand()%(29-26+1)+26;
rtemp=rand()%(29-26+1)+26;
while(rtemp<ltemp){
    rtemp=rand()%(29-26+1)+26;
}
```

  最后进行删除操作，参考红黑树一节，同样通过随机生成下标的方式得到需要删除的数据。

实验结果与分析

注

- 因为已经随机生成过一次数据放入input.txt，所以为了与实验报告内容一致，所以在两个实验的main.cpp里把这部分随机生成的代码注释掉了。
- 为了各要求文件的可读性更高，在input.txt，inorder.txt，delete_data.txt里都为每组数据都添加了额外的一行n的值

红黑树

- n=20的结果截图
  - 中序遍历序列

```
20
175 1057 1693 1872 2470 3893 4759 9838 10475 14361 16220 18629 18739 20358 21572 24670
27198 30023 31057 31818
```

  - 第二行为删除的结点关键字，第三行为删除后的中序遍历序列

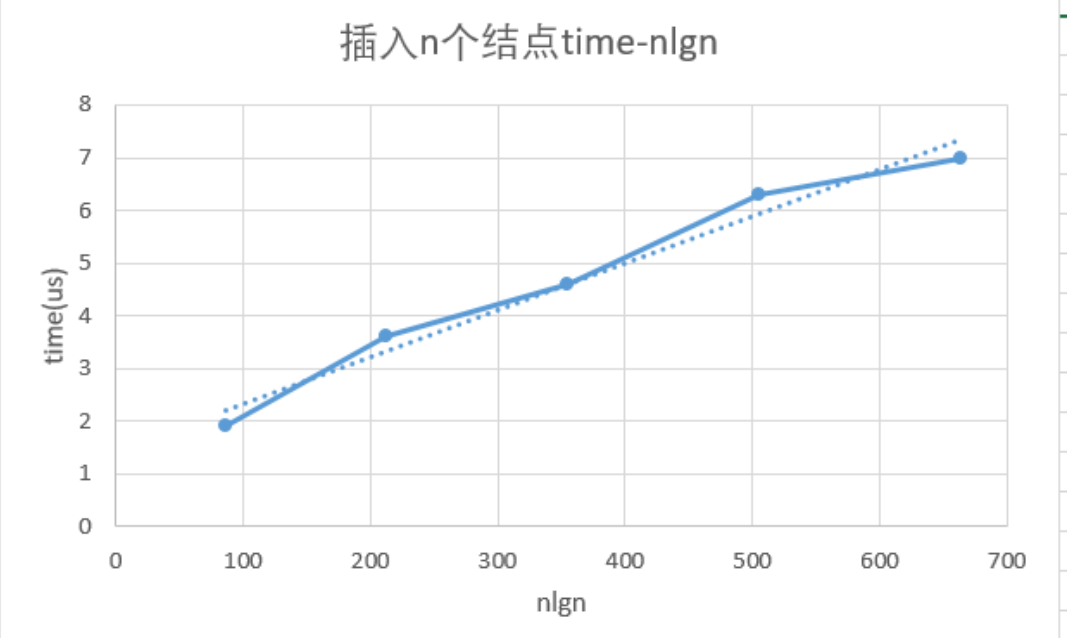
```
20
1693 4759 21572 3893 31818
175 1057 1872 2470 9838 10475 14361 16220 18629 18739 20358 24670 27198 30023 31057
```

- 时间复杂度分析

数据规模n	n*log(n)	time_insert(us)	time_delete(us)
20	86.4385619	1.9	0.7
40	212.8771238	3.6	0.9
60	354.4134357	4.6	1.1
80	505.7542476	6.3	1.3
100	664.385619	7	1.4

图表区

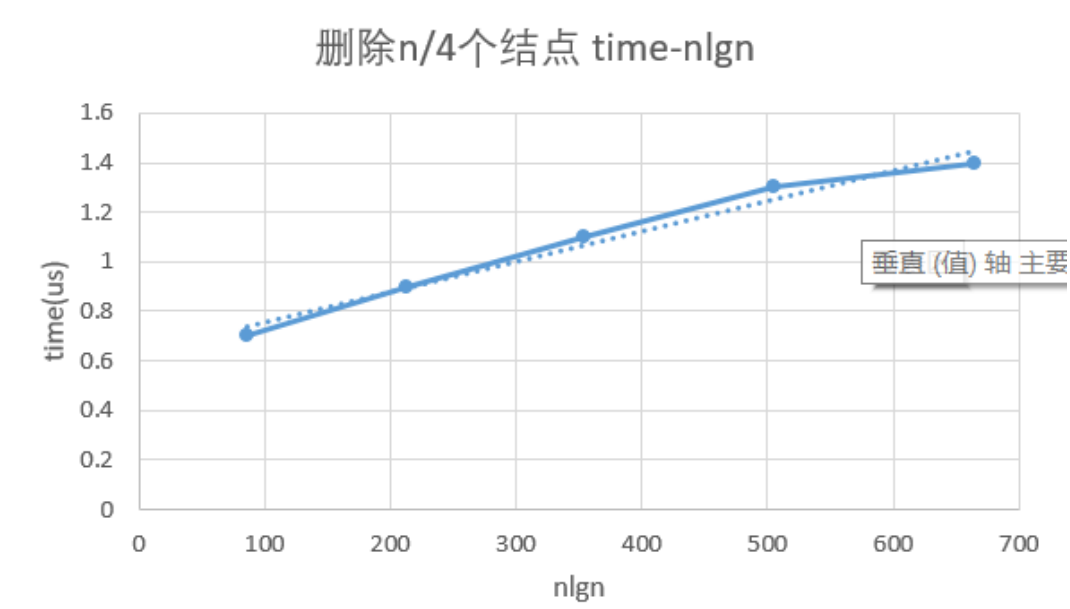
- 插入n个结点



  - 向一棵含n个结点的红黑树中插入一个新结点需要O（lgn）的时间，而构建红黑树的过程就是依次向含有0，1，2...，n - 1个结点的红黑树插入一个新结点，故理论时间复杂度为$O(lg1 + lg2 + lg3 + lg(n-1)) = O(nlgn)$

    由图像可知实际时间复杂度与理论分析的结果较为吻合

- 删除n/4个结点



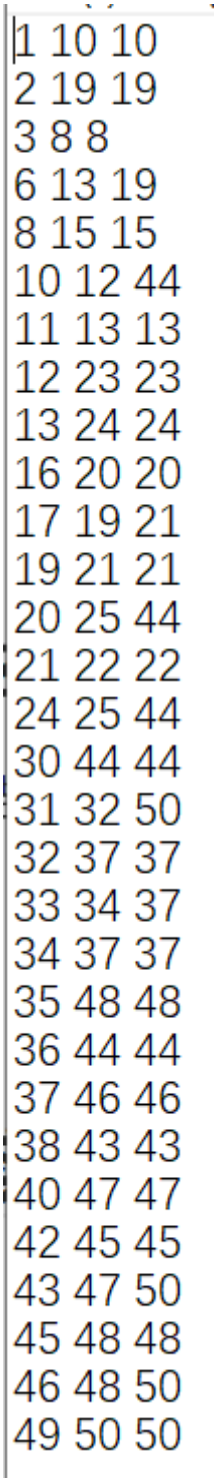
  - 从一个含有n个结点的红黑树中删除一个结点所需的时间为O（lgn），实验中逐个删除了n/4个结点，故理论时间复杂度为
    $O(lgn + lg(n-1) + lg(n-2) + ... + lg(n/4+1)) = O(nlgn)$

    由图像可知实际的时间复杂度与理论复杂度较为吻合。

区间树

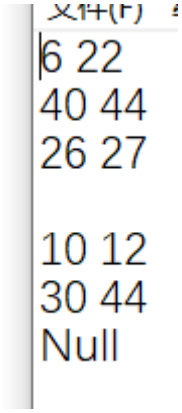
结果截图

- 构建好的区间树的中序遍历序列，每一行依次为 int.low, int.high, max

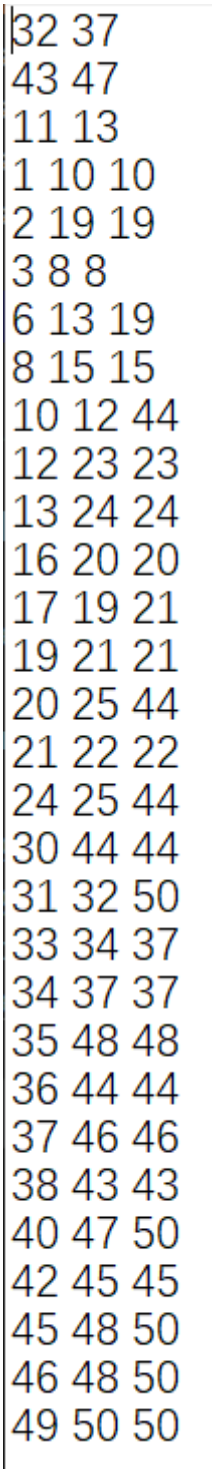


- 随机生成的搜索区间的搜索结果

  前三行为随机生成的查找区间，后三行为查找区间树后返回的结果



- 删除的结果，前面三行为要删除的数据的int域，后面的行 为删除完成后区间树的中序遍历结果



实验总结

- 通过本次实验，我对课上所学的红黑树相关算法有了更深刻的认识，以面向对象的方式组织了红黑树结构及其操作。并且因为一些小错误导致了一段时间的debug，对于这种复杂数据结构的debug方式有了更好的认识：构造简单而有效的样例，并从中研究错误的点。自己也通过这个方式把错误找出来了。
- 通过构造一些样例验证红黑树及区间树实现的正确性，自己对于删除和插入两个红黑树的难点操作的几种case情况更为熟悉，也更加得心应手。
- 熟悉了区间树里数据扩张定理的应用

参考文档