Lab1：无旋Treap

程序性能分析

测试工具

关于性能分析，由于要求必须使用Cycle作为时钟的单位，所以我本来想参考了PDF里面测量时钟周期的方法，具体来说代码如下（特别注意：Arm机器上不能运行下面的代码，编译会报错，不支持M1的Mac）：

```
static inline uint64_t rdtsc(){
    uint32_t low, high;
    asm volatile ("rdtsc" : "=a" (low), "=d" (high));
    return ((uint64_t)high << 32) | low;
}

void testFunction(){
    uint64_t start, end;
    start = rdtsc();

    /* 计时器代码开始 */

    /*
    * TODO
    */

    /* 计时器代码结束 */
    end = rdtsc();
    cout << end - start << endl;
}
```

但是，我写了一部分就发现写不下去了。因为我的一些函数都是诸如下面的样子的！因为有一些中途就直接返回了的结果，那我每一次return之前都需要加一段代码，简直是噩梦。

```
int merge(node* argv1, int val){
    if(/***/){
        /* code */
        return 1;
    }
    /* some code */
    /* some code */
    /* some code */
    if(/***/){
        /* code */
        return 2;
    }

    return 3;
}
```

所以我查阅了 **【程序性能调优】使用gprof 统计函数时间占比_黑皮花生的博客-CSDN博客** 发现有工具可以很好的分析函数的时间占比，这也很好的符合了这个实验的要求。

具体说来有下面的点值得一提

- gprof这个工具和gcc似乎是一家的，所以编译的时候可以通过加上一个标识符直接就能为这个工具所用
- 首先编译的时候必须要加上一些标记，因为待会要检查程序运行的函数的时间，编译的时候不能直接编译
- 然后就是，程序必须要先运行一次，这样生成一个gmon.out文件，然后根据这个文件（是个二进制的文件）分析之后得到分析结果

测试方法原理

- 数据部分，我采用的是随机生成 N 个不同的数据，然后测试插入、查找、删除的操作
- 为了测量时钟周期，我还是坚持 `rdtsc` 完成了部分代码，主要是统计插入、删除、查找的时钟周期（可以参见result.csv文件）
- 之后我修改了Makefile，这样可以得到不同数据的测试结果

```
LINK.o = $(LINK.cc)
# 必须加上pg 便于性能分析
CXXFLAGS = -std=c++14 -Wall -pg -g

test: test.o

all: test


clean:
	-rm -f test *.o gmon.out result

analysis:
	./test 1024 >> result.csv
	gprof test gmon.out -b >> result-1024.txt
	./test 4096 >> result.csv
	gprof test gmon.out -b >> result-4096.txt
	./test 16384 >> result.csv
	gprof test gmon.out -b >> result-16384.txt
	./test 65536 >> result.csv
	gprof test gmon.out -b >> result-65536.txt
```

测试结果

一、数据表

dataSize	insertTimeCycle	searchTimeCycle	deleteTimeCycle	height
1024	3409348	1049612	3589714	26
4096	15165088	4099824	14913926	30
16384	67646452	20195294	65121098	32
65536	314651268	101161684	296651666	39
262144	1419886106	641984332	1727484320	48
1048576	6121362440	2306860906	6926739624	50

对于的CSV文件如下

```
dataSize,insertTimeCycle,searchTimeCycle,height,deleteTimeCycle
1024,3409348,1049612,26,3589714
4096,15165088,4099824,30,14913926
16384,67646452,20195294,32,65121098
65536,314651268,101161684,39,296651666
262144,1419886106,641984332,48,1727484320
1048576,6121362440,2306860906,50,6926739624
```

那我们分析这几组结果可以发现如下结论：

- 首先插入和删除，都涉及到树的调整问题，所以消耗的时钟周期都非常大
- 此外，观察插入的时钟周期，当数据量增加4倍的时候，大致上插入的时钟周期也一样增长了4倍。基本上符合数据越大，时钟周期越大的结论，甚至可以说接近线性。

二、具体的函数执行时间（insert）

首先我们以最大的一组数据来1048576分析插入、删除里面具体的一些函数的时间，下面的结果是输出的报告里面的数据。因为分析工具最小的时间粒度是0.01秒。

由于直接看文本不太方便，我感觉把图搬上来：

- 在插入的函数中，主要的时间都是在执行split的函数，`split`行对应的0.1代表函数本身执行的时间，0.17代表函数的子函数执行的时间，总加起来高达0.27s，所以可以看出是非常的耗时间！Split这个函数
- 此外，Merge消耗的时间就没有那么严重尽管Merger执行次数高达200万次

```
234   ----------------------------------------------
235                  0.00    0.39 1048576/1048576     runInsertTest() [10]
236   [12]    18.0   0.00    0.39 1048576         Treap<int>::insert(int) [12]
237                  0.10    0.17 1048575/2097151     Treap<int>::split(TreapNode<int>*, int) [8]
238                  0.07    0.00 1048575/3145727     Treap<int>::find(TreapNode<int>*, int) [16]
239                  0.03    0.00 2097150/4194302     Treap<int>::merge(TreapNode<int>*, TreapNode<int>*) [32]
240                  0.01    0.00 1048576/1048576     TreapNode<int>::TreapNode() [65]
241                  0.00    0.01       1/1           Treap<int>::pre_traverse[abi:cxx11]() [72]
242                  0.00    0.00 1048576/1048576     RandGenerator::treap_rand() [113]
243   ----------------------------------------------
```

- 下图表示运行插入测试的时候，主要都是在执行Insert函数，这个也是符合逻辑的

```
-------------------------------------------------
                0.01    0.41       1/1          runTest(unsigned long) [2]
[10]    19.7    0.01    0.41       1        runInsertTest() [10]
                0.00    0.39 1048576/1048576     Treap<int>::insert(int) [12]
                0.00    0.02 1048577/4194549     std::map<int, int, std::less<int>, std::allocator<std::pair<int const, int> >
                0.00    0.00 1048576/3145728     std::_Rb_tree_iterator<std::pair<int const, int> >::operator->() const [56]
                0.00    0.00 1048577/3145731     std::operator!=(std::_Rb_tree_iterator<std::pair<int const, int> > const&, st
                0.00    0.00 1048576/3145728     std::_Rb_tree_iterator<std::pair<int const, int> >::operator++(int) [60]
                0.00    0.00       1/3           std::map<int, int, std::less<int>, std::allocator<std::pair<int const, int> >
                0.00    0.00       2/6           rdtsc() [127]
-------------------------------------------------
```

```
-------------------------------------------------
                0.00    0.39 1048576/1048576     runInsertTest() [10]
[12]    18.0    0.00    0.39 1048576         Treap<int>::insert(int) [12]
                0.10    0.17 1048575/2097151     Treap<int>::split(TreapNode<int>*, int) [8]
                0.07    0.00 1048575/3145727     Treap<int>::find(TreapNode<int>*, int) [16]
                0.03    0.00 2097150/4194302     Treap<int>::merge(TreapNode<int>*, TreapNode<int>*) [32]
                0.01    0.00 1048576/1048576     TreapNode<int>::TreapNode() [65]
                0.00    0.01       1/1           Treap<int>::pre_traverse[abi:cxx11]() [72]
                0.00    0.00 1048576/1048576     RandGenerator::treap_rand() [113]
-------------------------------------------------
```

三、具体的函数执行时间（remove）

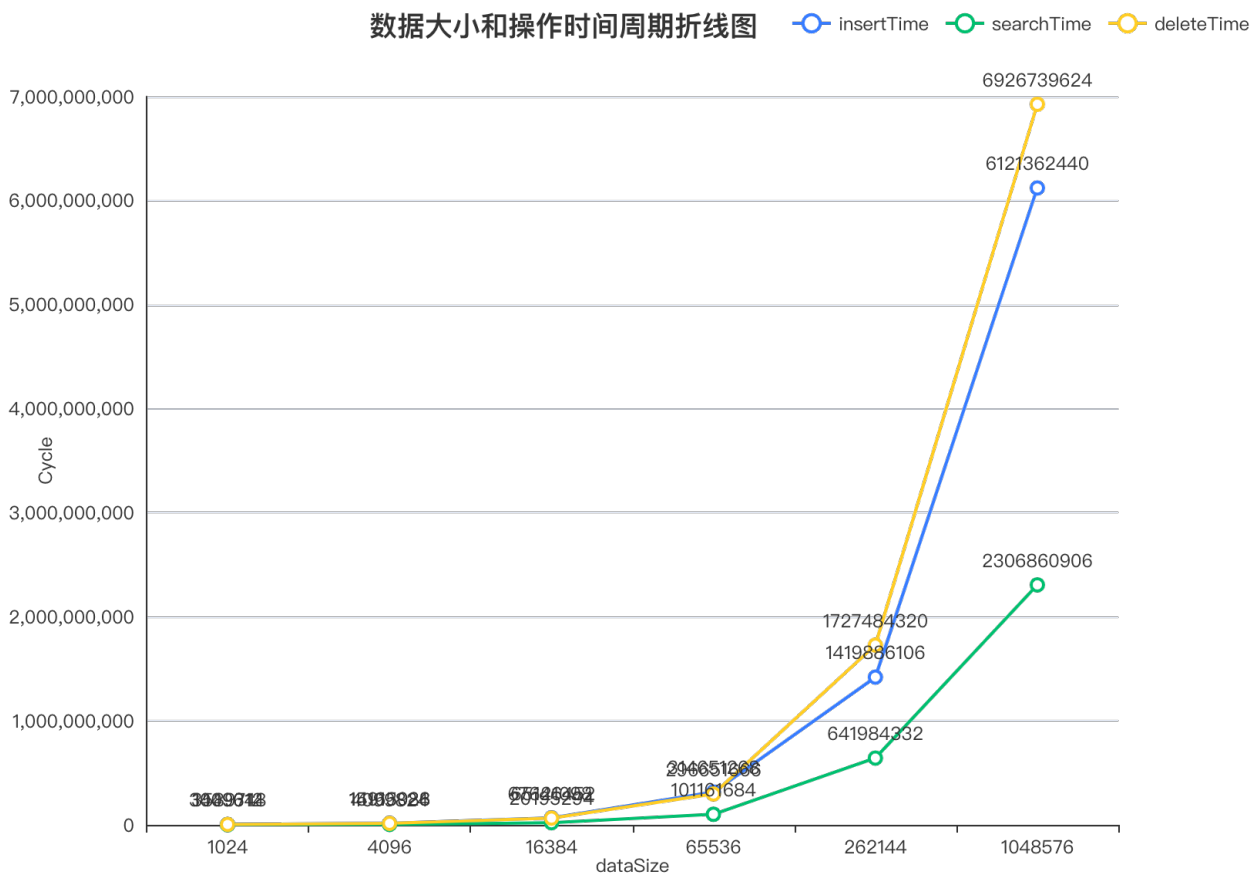
- 如下图所示，remove执行的时间主要还是消耗在了split的这个函数里面，消耗的时间0.1s，递归的时间0.17s
- 相比执行Merge的操作消耗的时间就没有那么多。

```
-----------------------------------
0.02    0.39 1048576/1048576     runDeleteTest() [9]
0.02    0.39 1048576         Treap<int>::remove(int) [11]
0.10    0.17 1048576/2097151     Treap<int>::split(TreapNode<int>*, int) [8]
0.07    0.00 1048576/3145727     Treap<int>::find(TreapNode<int>*, int) [16]
0.03    0.00 2097152/4194302     Treap<int>::merge(TreapNode<int>*, TreapNode<int>*) [32]
0.01    0.00 1048576/1048577     Treap<int>::removeAllNode(TreapNode<int>*) [64]
0.00    0.01 1048576/1048576     Treap<int>::splitForDelete(TreapNode<int>*, int) [81]
-----------------------------------
```

- 为什么Split的时间更多呢？推测是由于Split的机制比较复杂，递归更严格，当树的结构比较复杂的时候，分裂涉及到很多次的合并，比如PDF里面所叙述的多次分割，这样的处理带来了额外的时间开销。相比之下Merge的操作就简单了很多，
- 关于更详细的时间周期数据，可以参考测量的结果，在文件的result目录下

四、小结结论

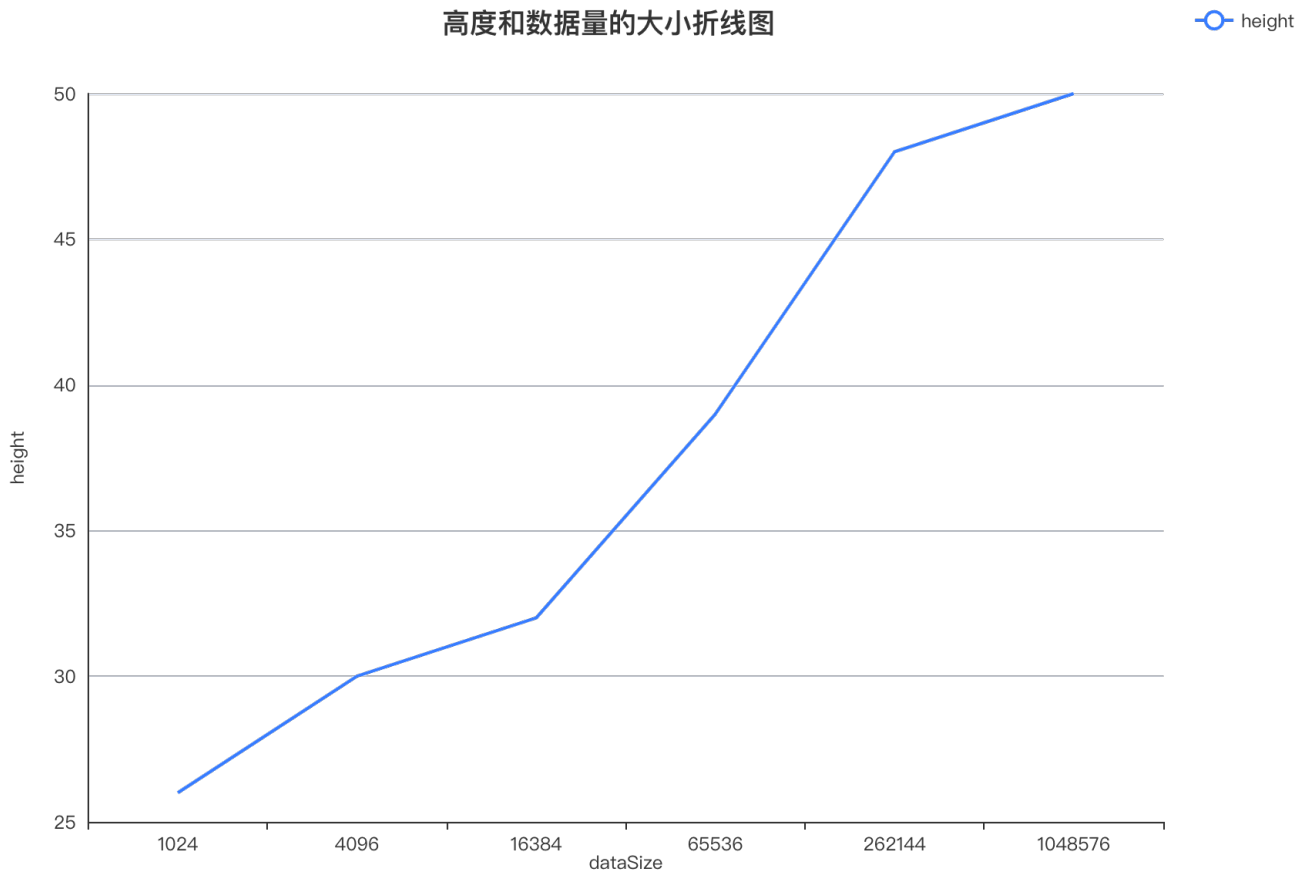
- 无论是插入还是删除，最消耗时间的步骤都是split，其次是merge操作，其他操作的时间几乎可以忽略不计
- 操作的时钟周期的大小基本上和数据的大小保持一致的关系



五、插入的数据和Height关系

刚刚的表格里面其实也有相关的数据，表示对应的数据量的大小和高度的关系。绘图得到如下结果。

- 首先，总体趋势是随着数据增大而消耗更多的高度来存储。
- 折线图的增长不是很严格，推测是数据的分布具有随机性，此外数据的权重都是随机生成的，所以也可能带来一定的干扰。
- 对比满二叉树，当楼层达到25的时候能够存储的数据远远高于1024，说明Treap Tree的性能和树的平衡还是有所欠缺的，相比较与AVL平衡来说。这也应征这个数据结构确实是一个弱平衡的数据结构。


高度和数据量的大小折线图
height
50
45
40
35
30
25
height
1024
4096
16384
65536
262144
1048576
dataSize