

Lab1-基数树 report

张越 522031910791

2024 年 4 月 14 日

1 背景介绍

在本次 lab 中，要求自主实现了两种基数树（RadixTree、Compressed RadixTree），实现了两种于存储 `int32_t`（32 位整型）类型集合（Set）的 4 叉基数树。并且实现两类树：`find`, `insert`, `remove`, `size`, `height` 的功能。

RadixTree 的简介如下：树的根节点有四个指针分别指向四个子节点，分别为 00, 01, 10, 11，代表这棵子树下的数字的二进制表示的前两位（其中 01,10,11 下没有存储数值，因此根节点维护的是空指针，图中用虚线表示这三个节点）。00 节点同样有四个子节点，每个节点代表该节点下子树存储的数值有不相同的第 3、4 位（00,01,10,11）。根据这样的规律，将根节点到一个叶子节点路径上的 4 个节点的 2 比特拼接起来，即可得到叶子节点所代表的数字。

而 Compressed radixTree 压缩了冗余的节点，简而言之：如果树中一个非根节点只有一个子节点，我们可以将该节点与其子节点进行合并，合并后的节点所代表的值为被合并节点代表的值拼接后的结果。

2 系统实现

简要介绍：本次 Lab 中 RadixTree 部分的代码实现比较简单，因为每个节点所包含的比特数是固定为 2 的，所以每次循环取值都只要操作 2 位 bit 进行节点的构造即可，主要难点在于对于 Remove 时，需要进行对于删除哪几个结点截止的判断，比较复杂，但总体难度较低。

难点主要集中在 Compressed RadixTree，在本次 Lab 实验中，由于 Compressed RadixTree 结点的 bit 数是不固定的，所以我在 Node 的结构体中添加了 `index min` 以及 `index max` 值，用来表示其结点中保存值对应 Value 的比特位（偏移量）。在 `insert` 以及 `remove` 函数中，我也使用循环找 path 的策略，但比较 tricky 的是位运算的操作，因为每一次 loop 的结点长度和偏移量都不同，所以需要使用比较复杂的操作进行处理，容易出现 bug（事实上，我也在这一部分调试了很久。。。)

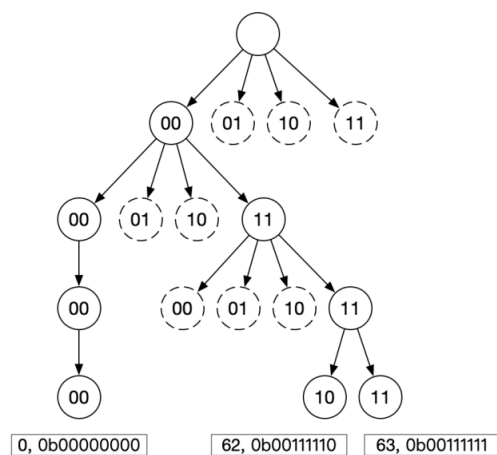


图 1: RadixTree

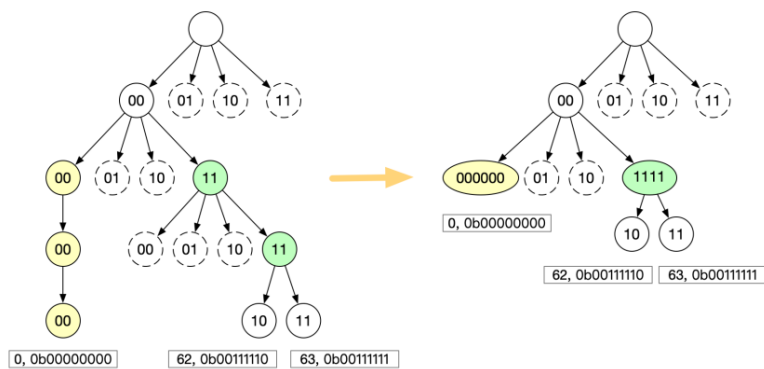


图 2: Compressed RadixTree

而 remove 部分主题上和 insert 以及 RadixTree 中的策略类似，有了之前的经验实现起来较为容易，唯一不同的地方在于合并结点的部分，这一部分我使用了递归的思路，但对于合并后子结点以及孙子结点的释放比较麻烦，所以我在 merge 函数部分也调试了很久。。。

最后是 size 和 height 接口，这一部分我使用了传统的递归算法，（但也许效率比较差？ anyway）总体而言，对于功能实现部分，由于吸取了 Lab0-哈夫曼压缩的经验，总体主体部分的实现比较顺利（除了比较 tricky 的位运算操作），也让我对于各种树的构造以及功能有了更深刻的认识！

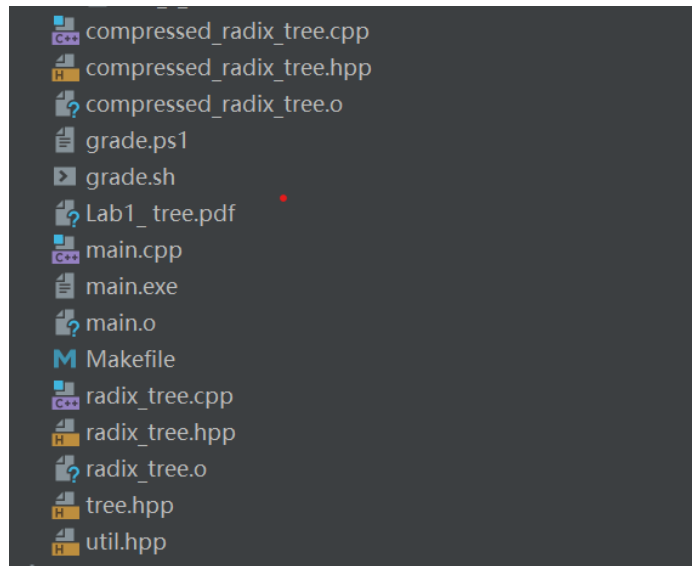


图 3: 代码主体部分框架

3 测试

3.1 正确性测试

```
PS C:\Users\zhangyue\Desktop\Lab1-handout> make clean
PS C:\Users\zhangyue\Desktop\Lab1-handout> make all
test_a: OK
test_b: OK
test_c: OK
grade: 100/100
<<<<<<< grade RadixTree over >>>>>>>
<<<<<<< grade CompressedRadixTree >>>>>>>
test_a: OK
test_b: OK
test_c: OK
grade: 100/100
<<<<<<< grade CompressedRadixTree over >>>>>>>
```

图 4: score-test

: 可见基础正确性测试没有问题

3.2 YCSB 测试

3.2.1 测试配置

工作负载: 3 种不同的工作负载, 在所有工作负载下, 测试程序不断循环对测试对象调某一种基础操作, 查询、插入、删除的数值均服从 zipfian 分布, 并加载 1000 个均匀随机分布的 `int32_t` 到测试对象中, 工作负载操作分别如下:

工作负载 1: 在该负载下, 每轮循环 50% 几率执行 `find` 操作, 50% 几率执行 `insert`。

工作负载 2: 在该负载下, 每轮循环 100% 执行 `find` 操作。

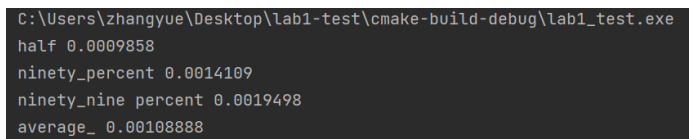
工作负载 3: 在该负载下, 每轮循环 50% 执行 `find`, 25% 执行 `insert`, 25% 执行 `del`。

测试对象: 分别是 `RadixTree`, `Compressed RadixTree` 以及 `RedBlack Tree` [1]

系统配置: Windows 系统, 使用的 IDE 是 Clion, 语言: C++ 17。

机器配置: 处理器: 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz

3.2.2 测试结果



```
C:\Users\zhangyue\Desktop\lab1-test\cmake-build-debug\lab1_test.exe
half 0.0009858
ninety_percent 0.0014109
ninety_nine_percent 0.0019498
average_ 0.00108888
```

图 5: 部分测试结果

3.2.3 结果分析

对于工作负载 1 而言: `Compressed Radix Tree` 的时延远超 `Radix Tree` 以及 `Red-Black Tree`。大约各个数据的时延是 `Radix Tree` 和 `Red-Black Tree` 的 30-100 倍, 插入查询删除随机 1000 组数据的时延达到了 20-30ms, 而 `Radix Tree` 的时延较小, 大约是 1-2ms, 是 `Red-Black Tree` 的 3-5 倍, 红黑树的时延最小, 大约仅有 0.29-0.38ms, 并且稳定性较强, 从时延上来分析, 红黑树优于基数树优于压缩基数树, 从算法的时间复杂度来分析, 红黑树是 $O(\log n)$ 的时间复杂度, 是对数级的, 而基数树和压缩基数树是 $O(n)$ 的时间复杂度, 是线性级的, 所以对于处理数据量较多的情况, 红黑树的效率优于基数树, 但是牺牲了空间复杂度, 而压缩基数树的 `insert` 和 `find` 过程更为复杂, 调用的包函数更多, 所以可能效率较差, 但是空间比较节省。

对于工作负载 2 而言: 三类树的时延都较小, 都仅为 0.2-1ms 之间, 事实上对于一颗空树的查询本身时间复杂度就是 $O(1)$, 所以这组数据也是合理的。

对于工作负载 3 而言: 我们发现在加入了删除操作以后, 各项时延数据与没有 `remove` 的工作负载 1 的数据相差不大, 可见 `remove` 函数本身与 `insert` 函数的时间复杂度差别并不大, 同

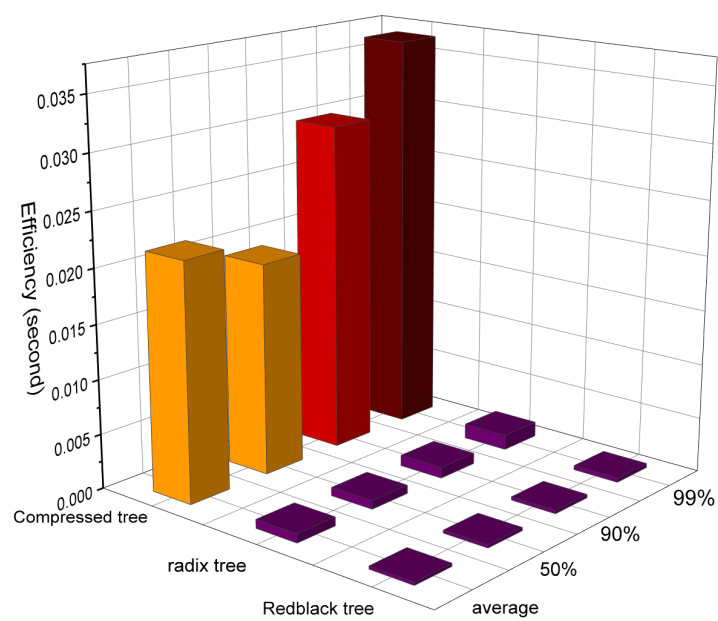


图 6: 测试结果-workload1

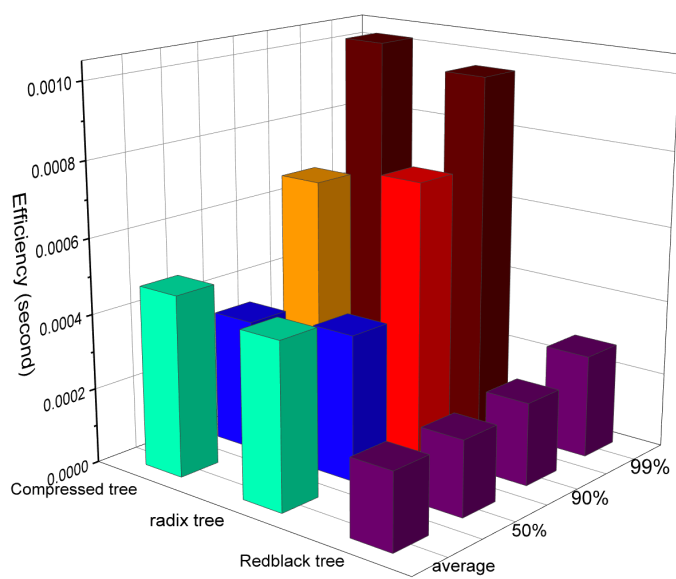


图 7: 测试结果-workload2

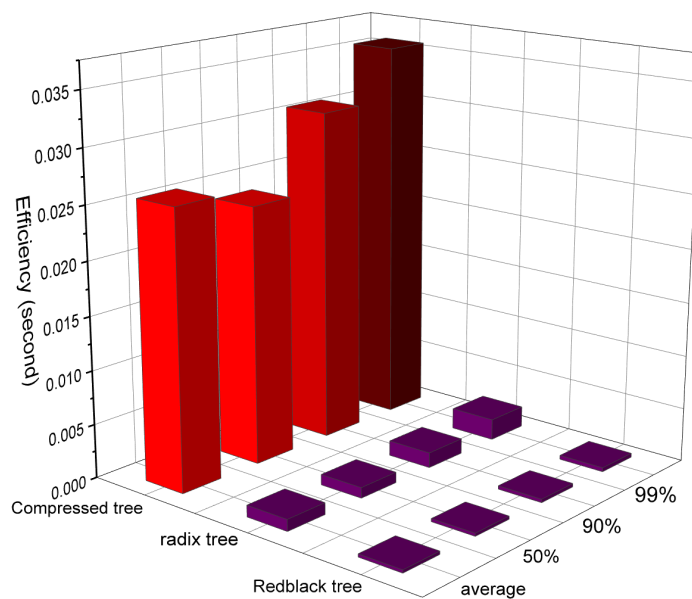


图 8: 测试结果-workload3

	时延(单位: 毫秒)	average	50%	90%	99%
工作负载1	Compressed tree	25.7813	24.0193	30.9455	35.6619
	Radix tree	1.08888	9.858	1.4109	1.9498
	Red Black tree	0.294739	0.2907	0.3241	0.3835
工作负载2	Compressed tree	0.4817	0.349	0.6749	1.0164
	Radix tree	0.4415	0.3873	0.729	0.9641
	Red Black tree	0.204411	0.2013	0.2199	0.2739
工作负载3	Compressed tree	21.7241	19.4036	30.1077	36.5986
	Radix tree	0.812697	0.7706	0.994	1.3714
	Red Black tree	0.284663	0.2803	0.3126	0.3942

图 9: 结果表格

时可能因为数据量不够大，导致有效删除的操作并不多，值得一提的是——在压缩基数树的输出操作中使用了 merge 这类递归函数，可能会影响到效率。

总体而言，红黑树的各种工作负载的时延数据大小都显著小于其他两种树，并且数据稳定性较强，对于各种操作不同比例混合的操作的时延相差不大。无论是性能还是稳定性都较强，因为他是一个比较稳定的树，但是他的空间复杂度比起基数树小，而压缩基数树的时延较大（主要原因应该我的实现比较冗杂。。。），但空间复杂度最小，总而言之，几种树各有利弊，可以根据不同情况进行取舍。

4 结论

在这次 Lab 中，我自主完成了 Radix Tree 和 Compressed Radix Tree 两种基本功能的实现，进行了正确性测试以及 YCSB 性能测试，通过与红黑树的性能对比，了解了各种树的优缺点。同时也加深了我对于树这一奇特的数据结构的理解，不限于 AVL-平衡树还有诸如 Splay Tree, KV-Tree, 红黑树, Radix tree 各种各样类型的树，可谓是百花齐放，特点各异。

这次 Lab 的 YCSB 测试也让我对于性能测试有了初步的入门，让我对于自动化测试有了基本的认识，可谓是受益匪浅。

总体而言，对于功能实现部分，由于吸取了 Lab0-哈夫曼压缩的经验，总体主体部分的实现比较顺利（除了比较 tricky 的位运算操作），也让我对于各种树的构造以及功能有了更深刻的认识！

5 建议

本次 Lab 中我对于树各类功能的实现还是不够优化，导致性能效率较低，我认为进一步可以减少代码功能主体部分的函数调用以及代码复用，以降低 waste time。同时 size 和 height 接口也可以优化，可以在 insert 和 remove 的过程中对 height 和 size 进行操作，以避免调用接口时递归函数较差的时间复杂度，进一步优化性能！

附言：总体而言感谢这次 Lab 的经历以及助教老师相助，让我对于不同数据结构的认识有了进一步的了解。

——由于本人学疏才浅，做这个 Lab1 时已经绞尽脑汁、同时又由于最近事务繁多、实属分身乏术，上述建议中提到进一步的程序优化恐怕已是无力为之，由于做 LAB 的时间较为紧迫，程序也未免十分周全，能将所有异常情况纳入考虑，只得匆匆为之，若有纰漏之处，恳请助教与老师的谅解。

参考文献

[1] Ananda Rao H anandarao. *Red-Black-Tree*. 2017.