基数树实验报告

邓杰天 522031910441

2024 年 4 月 22 日

1 背景介绍

本实验需要实现基数树，也就是 Radix Tree. 基数树是一种多叉搜索树，通过将共有的前缀抽象成结点，来提供相对快速的查找操作。在本实验中，每两个比特的前缀构成一个节点，因此生成一个四叉树，从而存储 32 位整型的集合。

但这样做的效率相对低下，因为固定大小的结点完全可以压缩成一个节点，从而进一步提高搜索速度。因此，本实验还实现了结点压缩，让单个节点尽可能存放足够多比特的前缀，直到部分节点只共享该节点的一部分，才对其进行拆分；当节点删除时，也应当及时合并结点。这样，基数树的高度下降，且可以在单个节点中检查更多的比特数，从而又提升了基数树的效率。

最后，本实验基于 YCSB 测试，尽可能模拟真实世界的工作负载，并使用三种工作模版来测试基数树、压缩基数树和红黑树的性能。

2 系统实现

2.1 基数树

在基数树中，每个节点均为两个比特；基数树为四叉树，故需要维护四个孩子；删除时需要自下而上删除父结点，故需要维护父亲。

在删除时也需要找到对应的结点，因此可以设置一个内置函数 findInter，它可以找到搜索结束时停在哪个结点，并设置引用参数 isFound 来判断找到与否。

插入节点相对简单，只需要循环 16 次，每次读取两个比特，如果不存在对应的孩子，就进行插入。

删除结点时，先用 findInter 找到停在哪个结点，再自下而上找到父亲，如果删除后父亲没有孩子，那也需要删除。

在上述操作中就可以顺带维护节点数和高度，故只需要维护这两个变量即可。

2.2 结点压缩

结点压缩的难点在于不能确定结点的大小，因此需要额外维护比特数，为了在确定高度时深度优先遍历，可以维护是否为叶子节点，但不是必须的，也可以通过孩子数来判断。结点结构体如下：

这时插入、删除、查找都需要先找到节点位置，因此还是需要 findInter 函数，该函数自上而下依次查找，并维护已经检查的比特数，直到检查完所有 32 位；如果结点内容不匹配，或没有对应的孩子，则返回停下的节点。

这里的难点在于位操作，需要反复地左右位移，为了防止出现算术右移，获得正确的前缀值，需要将所有 32 位整型强制类型转换为无符号整型。

插入结点是本次实验的难点。找到停留位置后，先检查数据是否匹配，如果匹配，说明是没有对应的孩子，需要插入孩子；如果不匹配，则需要分裂结点。使用循环两位两位地检查，直到找到最长共有前缀。接下来尤其需要关注孩子与孩子数的重新分配：新建两个结点，一个放置原结点非共享部分，并继承原结点所有的孩子和孩子数；另一个的数据应当是插入数据的全部剩余内容。考虑到删除结点时，孩子不会变成 NULL，因此要将原结点的孩子全设置成 NULL 后再插入新建的两个节点。

删除结点时同样要注意删除结点时，孩子不会变成 NULL，需要重新设置好 NULL；同时当孩子数为一时需要合并。

压缩节点的难点在于需要维护的信息非常多，每次操作后，都要严格检查是否重新维护了结点的所有信息，再进行下一步操作；另一个难点则是位操作，注意记录移动位数

3 测试

3.1 YCSB 测试

3.1.1 测试配置

本次测试的测试对象为基数树、压缩基数树和红黑树，C++ 内置的 std::set 正是用红黑树实现的，因此将其封装好，并继承 Tree，便于进行测试。本次测试的工作负载有三种，分别是：50% 执行 find，50% 执行 insert；总是执行 find；25% 执行 insert，25% 执行 remove，50% 执行 find. 每组测试执行 30 秒，并记录各操作的时延。为了实现上述工作负载，需要生成随机数，并用余数来生成操作类型，并将时延压入 vector 中。最后，为了能够执行测试文件 myYCSB.cpp，需要修改 Makefile，从而能使用 make test 命令来执行测试

3.1.2 测试结果

三种负载的测试结果如下：

workload1	Radix.find	insert	CRadix.find	insert	RB.find	insert
Average	1833	1862	854	856	890	915
P50	1900	1900	800	800	800	800
P90	2300	2300	1200	1200	1100	1200
P99	3900	4100	2500	2500	2400	2500

workload2	Radix.find	CRadix.find	RB.find
Average	434	423	549
P50	400	400	500
P90	500	500	600
P99	1000	800	1000

workload3	Radix.insert	remove	find	CRadix.insert	remove	find	RB.insert	remove	find
Average	2427	2367	1595	1042	1033	809	973	1029	856
P50	2300	1800	1300	1000	900	700	900	1000	800
P90	3100	4300	2600	1400	1600	1100	1200	1300	1000
P99	5600	7100	4400	2700	2900	2300	2600	2700	2400

图 1和图 2分别对比了不同工作负载下各类树的查找时延和 workload3 下不同操作的时延

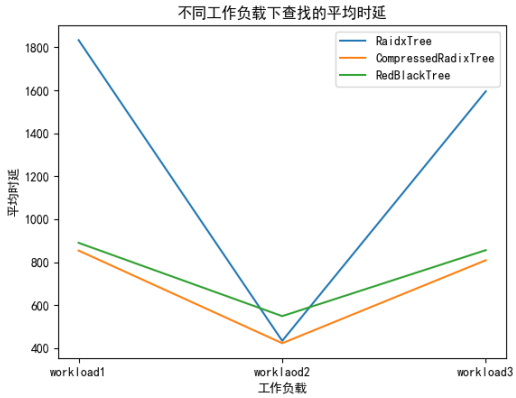

图 1: 不同工作负载下各类树的查找时延

3.1.3 结果分析

实验结果表明，压缩基数树的性能和红黑树比较接近，并且都显著大于基数树。压缩基数树需要更多节点存储，但空间复杂度更低，且共享前缀能有助于不遍历树的高度就判断查找值是否存在；红黑树则需要额外维护平衡信息，两者在大样本下孰有优劣，而压缩基数树可能略有优势，可见测试结果是符合预期的。

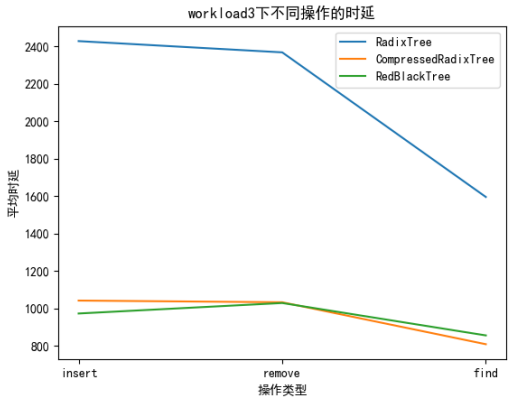

图 2: workload3 下不同操作的时延

在 wordload1 时，大样本地插入导致插入和搜索的性能相近，而压缩基数树可以共享前缀，且无需维护平衡性，这可能是压缩基数树性能略微大于红黑树的原因；在 wordload2 时，样本数恒为 1000，较小的样本数导致压缩节点与否不能体现差异，而基数树可以在搜索中途就发现是否查找失败，导致性能高于红黑树；在 workload3 时，删除时常常删除不存在的结点，同时会导致树的形状反复变化，性能普遍发生下降，而因为插入和删除前都要先查找，故查找性能优于其余两种。

各个操作有一个共性，即 P90 与 P50 差距不大，但 P99 和 P90 的差距相当显著，说明三种树都可能出现时间复杂度改变的最坏情况。

4 结论

本次实验实现了基数树和压缩基数树，加深了对位操作和数据维护的理解。测试时模拟 YCSB 测试，能更好地模拟现实数据。压缩基数树通过共享前缀和压缩节点，在处理共享前缀较多的字符串存储方面优势甚至略高于红黑树，可用于未来存储字符串。

5 建议

助教给的 github 的红黑树跑不过测试，如果能优化一下这方面就更好了。