LAB1 基数树的压缩

522031910213 朱涵

April 12, 2024

1 系统实现

1.1 代码简述

1.1.1 字典树的实现

字典树类的结构包含根节点 root 和节点数量 size，节点结构包含一个 child 数组和 parent 指针。字典树实际上就是固定多叉树，题目中的第一种“基数树”实际上也就是 4 叉字典树，字符表为 00,01,10,11，因此实现逻辑较简单，在此简述，不占太多篇幅。下面依次简述插入、删除等操作的逻辑。

1. **查找**：如图 1 所示，是字典树操作最基本的逻辑，从根部开始访问，index 是当前前缀的前 2 位，通过位运算可以直接算得前缀应该在第几个儿子中，由于储存的都是 32 位 int 型，所以遍历到 i 为 0 即可结束，途中如果有前缀不匹配就返回 false。
2. **插入**：同查找，先找到第一个儿子为空的位置，然后插入新节点即可。
3. **删除**：同查找，找到节点位置，删除后递归的向上判断是否为叶子节点，因为维护了 parent 指针，所以一路向上循环即可。
4. **节点个数**：在字典树的 size 中维护，插入和删除的时候进行更新即可。
5. **字典树高**：由于字典树的高度是固定的，即不是 17 就是 1，只需要判断一下 root 是否为叶即可。

1.1.2 基数树的实现

基数树在字典树的结构上进行了一些调整：因为每个节点都有可能是压缩节点，所以需要额外的空间开销来维护前缀以及长度；除此之外还添加了 is_leaf 判断是否为叶节点。同时还维护了两个工具函数——**first_k_bits** 和 **last_k_bits** 用于位运算。基数树的**实现难点**主要在于插入操作如何进行分裂，由于 int 型并非字符串，在操作前缀

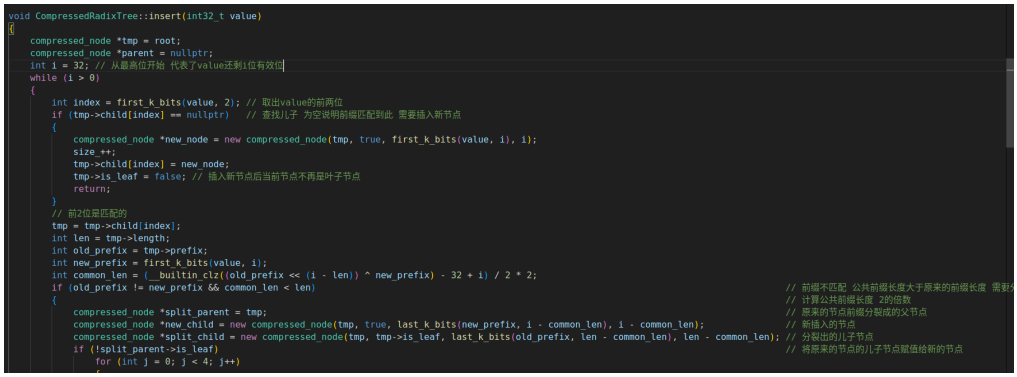
```cpp
bool RadixTree::find(int32_t value)
{
    radix_tree_node *tmp = root;
    for (int i = 30; i >= 0; i -= 2)
    {
        int index = (value >> i) & 0x3; // 00, 01, 10, 11
        if (tmp->child[index] == nullptr)
        {
            return false;
        }
        tmp = tmp->child[index];
    }
    return true;
}
```

Figure 1: 字典树的查找

时只能使用位运算等价截断，而且后续的前缀拆分等等都比操作字符串要复杂很多。本人进行了大约 5-6 个小时才完成调试，不知道是不是我的实现逻辑比较复杂的原因。下面阐述本人实现的基数树类的各个操作。

1. **查找**：基本和字典树没有区别，仍然是取出前缀进行对比，唯一不同的只有每次取出前缀的位数需要根据当前节点的 length 来调整。

2. **插入**：如图 2 所示，每次访问到一个节点需要根据节点所含前缀 prefix 和当前关键值的 value 的比对结果来分类。由于 int 数操作的特殊性，在此规定 prefix 包含的最低 length 位就是实际的前缀，value 的最高 i 位就是当前关键值的实际有效位（剩余位）。比较后有三种情况，1——根据 index 索引到的**儿子为空**，此时需要插入新的节点，前缀就是 value 的有效位，2——根据 index **索引到了一个儿子节点**，且儿子节点所含的前缀 prefix 和 value 完全匹配，而且是叶节点，说明 value 已经存在，直接返回，3——索引到了儿子节点，但是**前缀和关键值不匹配**，此时需要根据公共前缀长度进行分裂。先计算出 oldprefix 和 newprefix 的公共前缀长度 commonlen，如果等于 len，说明节点的前缀是关键值的子串，此时无需分裂，只需要和第 2 中情况一样，把关键值的剩下部分插入新节点即可；如果小于 len，说明两者只有部分前缀匹配，此时根据一系列的位运算把分裂后的前缀计算出来，并对分裂后的父节点 splitparent、分裂后的子节点 splitchild 以及新节点 newchild 进行连接，最后返回。由于本人不熟悉，插入操作中的位运算调试了很久才完成，不知道是否有更加简洁的实现逻辑。



```
void CompressedRadixTree::insert(int32_t value)
{
    compressed_node *tmp = root;
    compressed_node *parent = nullptr;
    int i = 32; // 从最高位开始 代表了value还剩i位有效位
    while (i > 0)
    {
        int index = first_k_bits(value, 2); // 取出value的前两位
        if (tmp->child[index] == nullptr)   // 查找儿子 为空说明前缀匹配到此 需要插入新节点
        {
            compressed_node *new_node = new compressed_node(tmp, true, first_k_bits(value, i), i);
            size_++;
            tmp->child[index] = new_node;
            tmp->is_leaf = false; // 插入新节点后当前节点不再是叶子节点
            return;
        }
        // 前2位是匹配的
        tmp = tmp->child[index];
        int len = tmp->length;
        int old_prefix = tmp->prefix;
        int new_prefix = first_k_bits(value, i);
        int common_len = (__builtin_clz((old_prefix << (i - len)) ^ new_prefix) - 32 + i) / 2 * 2;
        if (old_prefix != new_prefix && common_len < len)
        {
            compressed_node *split_parent = tmp;
            compressed_node *new_child = new compressed_node(tmp, true, last_k_bits(new_prefix, i - common_len), i - common_len);
            compressed_node *split_child = new compressed_node(tmp, tmp->is_leaf, last_k_bits(old_prefix, len - common_len), len - common_len);
            if (!split_parent->is_leaf)
                for (int j = 0; j < 4; j++)
                {
                                                                    // 前缀不匹配 公共前缀长度大于原来的前缀长度 需要
                                                                    // 计算公共前缀长度 2的倍数
                                                                    // 原来的节点前缀分裂成的父节点
                                                                    // 新插入的节点
                                                                    // 分裂出的儿子节点
                                                                    // 将原来的节点的儿子节点赋值给新的节点
```

Figure 2: 部分插入代码

3. **删除**：如图 3 所示，仍旧先找到节点，删除之后判断父节点是否只有一个儿子。如果是，进行合并。**无需递归的判断上下是否有其他需要合并的节点**，因为删除操作只会影响父亲的儿子数目，其他的节点的儿子数目应该在之前就保持了不等于一。

```
void CompressedRadixTree::erase(int32_t value)
{
    compressed_node *tmp = root;
    while (1) // 查找value的节点
    {
        int index = first_k_bits(value, 2); // 取出value的前两位
        if (tmp->child[index] == nullptr)   // 查找儿子 为空说明没有找到

            return;

        // 找到儿子
        tmp = tmp->child[index];
        int len = tmp->length;
        // 取出前len位进行比对
        if (tmp->prefix != first_k_bits(value, len))
            return;
        // 比对成功 如果是叶子节点说明找到了
        if (tmp->is_leaf)
            break;
        value <<= len;
    }
```

Figure 3: 部分删除代码

4. **节点个数**：同字典树，在 size 里维护。

5. **基数树高**：由于不好维护，遂直接递归遍历一次树得到树高。

2 测试

2.1 YCSB 测试

本实验的测试大体逻辑和文档中要求一致，即先利用 zipf 函数生成随机分布的 1000 个数据插入到对象中，然后持续运行 **60s** 的工作负载，利用 **chrono 库**的工具函数记录每一次操作的时间并计算出时延，输出到文件中。

2.1.1 测试配置

- 工作负载：共 3 种工作负载，分别是 **0.5 的 find+0.5 的 insert、1.0 的 find 以及 0.5 的 find+0.25 的 insert+0.25 的 remove**。
- 测试对象：基数树和压缩基数树的实现已在第一部分阐述。红黑树结构则使用了 stl 的 set 结构[1]。
- 系统配置：在 VMware 虚拟机上运行，操作系统为 Ubuntu。
- 机器配置：虚拟机内存为 4GB，主机内存为 32GB，处理器为 Intel i5-12400f。

2.1.2 测试结果

实验结果如下表（单位：纳秒）。

Latency	Average	P50	P90	P99
Find	440	405	621	946
Insert	443	407	623	952
Total	441	406	622	949

(a) RadixTree

Average	P50	P90	P99
320	303	449	733
318	299	447	730
319	301	448	731

(b) CompressedRadixTree

Average	P50	P90	P99
485	441	684	1015
505	461	706	1045
495	451	695	1030

(c) RB-Tree

Figure 4: WORKLOAD1：0.5 find + 0.5 insert

Latency	Average	P50	P90	P99
Find	65	54	84	188
Total	65	54	84	188

(a) RadixTree

Average	P50	P90	P99
79	70	97	162
79	70	97	162

(b) CompressedRadixTree

Average	P50	P90	P99
186	174	207	279
186	174	207	279

(c) RB-Tree

Figure 5: WORKLOAD2：1.0 find

Latency	Average	P50	P90	P99
Find	613	457	1121	1602
Insert	739	679	1147	1648
Delete	814	592	1544	2124
Total	695	571	1297	1866

(a) RadixTree

Average	P50	P90	P99
250	222	374	633
276	245	401	689
293	250	468	748
267	234	408	681

(b) CompressedRadixTree

Average	P50	P90	P99
424	369	592	960
494	433	686	1105
544	478	754	1206
471	411	667	1079

(c) RB-Tree

Figure 6: WORKLOAD3：0.5 find + 0.25 insert + 0.25 remove

[1]源码参考 C++ 标准库中的 set.h。

2.1.3 结果分析

- **横向对比**：在工作负载 1 下，总体性能为压缩树 > 基数树 > 红黑树；在工作负载 2 下，总体性能为基数树 > 压缩树 > 红黑树；在工作负载 3 下，总体性能为压缩树 > 基数树 > 红黑树。总体上压缩后的字典树的性能是最优的。
- **纵向对比**：三种数据结构在工作负载 2 下都是时延最小的，很显然这是因为工作负载 2 的数据量最小，只有原本的 1000 个数据。工作负载 3 进行了 0.25 的插入，工作负载 3 进行了 0.5 的插入，显然时延和数据量大小正相关。
- **结论**：

1. 可以发现，两种基数树和红黑树对比下，查找所占比越大基数树的性能优势越明显，这应该是因为基数树作为字典树，查找的复杂度虽然和二叉搜索树一样是 $O(logN)$ 的，但是字典树每次查找是通过哈希直接索引，而且本实验的字典树有四叉，从而树高也就远小于红黑树，因此查找性能要比红黑树好很多。
2. 将基数树压缩前和压缩后进行对比，发现当设计插入和删除操作时压缩后都优于压缩前，显然是因为压缩后树高降低从而引起的性能变高。而且注意到工作负载 3，压缩前的基数树的性能远远差于压缩后的，推测是因为压缩前的基数树删除节点后需要递归的向上进行节点删除，而压缩后的基数树最多只需要合并其父节点一个节点。
3. 工作负载 2 下的情况有些特殊，压缩前的基数树的查找性能要比压缩后好一点，理论上压缩后的基数树树高变低从而循环次数变少，应该性能更优。推测是因为压缩后的基数树在查找时除了进行单纯的位运算哈希索引以外，还需要取出树节点结构中的 length，这一步或许导致了性能变差，甚至超过了树高带来的优势。

3 结论

由以上的实验结果和分析得出了最终的结论：

1. 基数树是一种特殊的字典树，具有字典树的优点——**良好的查找性能**，以及**较低的空间开销**。在存储 32 位 int 型的数据时，基数树的查找性能要远优于红黑树结构，而插入和删除则相近甚至要稍差。因此基数树适用于**多读少写**的场景，作为类似于字典的结构进行使用；在读写均衡的场景下则是红黑树更为合适。
2. 基数树可以进行前缀压缩，压缩后的基数树具有**更好的写性能**，并且读性能于压缩前相当。压缩后的基数树同样适用于多读少写的场景，甚至可以在一部分需要多写的场景下使用。

4 实验不足

本实验仍存在一些需要完善的地方，比如基数树的压缩或许可以通过空间开销更小的标记压缩法进行实现，并且红黑树结构直接采用了 stl 中的 set 结构，或许导致了一些性能估计上的误差，可以考虑使用自行实现的红黑树来进行测试。