一种新的数据结构—-TT树

顾顺

2023 年 12 月 10 日

目录

1 引言 **2**

2 朴素的TT结构 **3**

2.1 核心算法介绍 3

2.2 时间复杂度分析 3

2.3 代码实现(C++,插入函数) 4

3 TT插入删除的优化 **6**

3.1 优化算法 6

3.2 时间复杂度分析 6

4 TT的特殊性质及一些应用 **7**

4.1 性质 7

5 结语 **8**

摘要

关键词: 数据结构平衡树 二叉索引树 自平衡

1 引言

平衡二叉索引树是一种用途广泛的数据结构,可以实现快速插入,删除,查询排名,第k大等,同时也是C++语言中set与map的底层实现(红黑树).

本文将提出一种自平衡二叉索引树,称之为Twist Tree(TT).

该结构没有采用一般的平衡方式旋转(rotation),而是用了一种全新的维护方式Twist.

2 朴素的TT结构

2.1 核心算法介绍

这里仅介绍插入和删除算法,其他同普通二叉索引树
首先定义一棵合法的TT树,当且仅当满足如下条件:

(1) 满足二叉索引树的所有条件

(2) 左右子树的大小(size)相差不超过1

(3) 左右子树均为空结点或一颗合法TT树

维护方式比较简单.以插入为例,假设需要向一棵合法TT树的左子树插入一个键值(未出现过),那么可以分为以下两种情况:

(1) 插入后,左右子树大小相差依然不超过1.这种情况下,只需要向左插入并保证左子树依然是合法TT树,显然可以直接递归

(2) 若插入后破坏了条件2,那么显然,左子树大小比右子树大2,那么可以进行如下操作:向右子树插入当前根节点的键值,将当前根节点键值替换为左子树最大值,删去左子树最小值.这些操作可以直接递归,这就是朴素的Twist(缠绕)操作.

删除同理,不再赘述

2.2 时间复杂度分析

以下是对于插入及删除的时间复杂度分析.由于实现原理上类似,假定插入和删除的复杂度同级.事实上,可以通过计算得知确实如此.

设$i(p,v),d(p,v)$表示在p为根的子树上插入/删除v,那么在递归的过程中,显然有:

左子树大小为奇数,偶数等概率,右子树同理,于是可知上文中,情况(1)出现概率为$\frac{3}{4}$,情况(2)出现概率为$\frac{1}{4}$.

记:T(n)表示i(p,v) (p子树大小为n)的复杂度,那么有递推式:

$$T(n)=\frac{1}{4}\cdot 3T(\frac{n}{2})+\frac{3}{4}\cdot T(\frac{n}{2})+O(1)$$

根据主定理:

$$T(n)=\Theta(n^{log_2\frac{3}{2}})\approx\Theta(n^{0.5849625})$$

2.3 代码实现(C++,插入函数)

```
const int MAXN=1e5+5;
const int INF=INT_MAX;
class TT_node{
    friend class simple_TT;
    private:
        int left,right,v,sz,fr,to;
        TT_node(void){
            left=right=0;
            sz=0;
            fr=to=INF;
        }
        void operator=(TT_node x){
            v=x.v;
            fr=x.fr;
            to=x.to;
        }
};
class simple_TT{
    private:
        int n;
        int root;
        TT_node tr[MAXN];
        int new_node(int v){
            n++;
            tr[n]={0,0,v,1,pre(v),suc(v)};
            return n;
        }
        int new_node(void){
            n++;return n;
        }
        void pushup(int p){
            tr[p].sz=tr[tr[p].left].sz+tr[tr[p].
               right].sz+1;
        }
    public:
        const int pre(const int v){
            //......
            int p;
            return p;
        }
        const int suc(const int v){
            //.......
            int p;
            return p;
        }
        void ins(int& p=root,const int id,const int
            v){
            if(p==0){
                p=new_node();
                tr[p]=tr[id];
                if(tr[p].fr) tr[tr[p].fr].to=n;
```

```
50                  if(tr[p].to) tr[tr[p].to].fr=n;
51                  return;
52              }
53              if(v<tr[p].v){
54                  if(tr[tr[p].left].sz>tr[tr[p].right
                        ].sz){
55                      ins(tr[p].right,p,tr[p].v);
56                      if(tr[p].fr) tr[tr[p].fr].to=n;
57                      if(tr[p].to) tr[tr[p].to].fr=n;
58                      int f=tr[p].fr;
59                      tr[p]=tr[f];
60                      del(tr[p].left,f,tr[f].v);
61                      if(tr[p].fr) tr[tr[p].fr].to=p;
62                      if(tr[p].to) tr[tr[p].to].fr=p;
63                  }
64                  ins(tr[p].left,id,v);
65              }else{
66                  if(tr[tr[p].left].sz<tr[tr[p].right
                        ].sz){
67                      ins(tr[p].left,p,tr[p].v);
68                      if(tr[p].fr) tr[tr[p].fr].to=n;
69                      if(tr[p].to) tr[tr[p].to].fr=n;
70                      int f=tr[p].to;
71                      tr[p]=tr[f];
72                      del(tr[p].right,f,tr[f].v);
73                      if(tr[p].fr) tr[tr[p].fr].to=p;
74                      if(tr[p].to) tr[tr[p].to].fr=p;
75                  }
76                  ins(tr[p].left,id,v);
77              }
78              pushup(p);
79          }
80
81  };
```

3 TT插入删除的优化

3.1 优化算法

在朴素算法的基础上,增加函数$i_d(p, vi, vd)$,表示同时在以p为根的子树上删去vd并插入vi,达到降低时间复杂度的目的.

实现方面,i_d函数相当于插入和删除函数的复合,只需要将同一棵子树上的插入和删除合并即可.

3.2 时间复杂度分析

类似的,假设插入和删除函数复杂度同级,为$T1(n)$,i_d函数复杂度为$T2(n)$,通过类似于朴素算法的分析,可以得到递推式:

$$T1(n) = T1(\frac{n}{2}) + \frac{1}{4}T2(\frac{n}{2}) + O(1)$$

$$T2(n) = \frac{3}{4}T1(\frac{n}{2}) + \frac{3}{4}T2(\frac{n}{2}) + O(1)$$

记$a_n = T1(2^n), b_n = T2(2^n)$ 可以设计出:

$$c_n = a_n + \frac{-1+\sqrt{13}}{6}b_n d_n = a_n + \frac{-1-\sqrt{13}}{6}b_n$$

于是有

$$c_n \approx (\frac{7+\sqrt{13}}{8})^n$$

$$d_n \approx (\frac{7-\sqrt{13}}{8})^n$$

得到

$$T1(n) = \Theta(n^{log_2 \frac{7+\sqrt{13}}{8}}) \approx \Theta(n^{0.4067477})$$

4 TT的特殊性质及一些应用

4.1 性质

首先是结构上的性质.

显然,任何一对左右子树大小相差不超过1,那么可知树高严格$\lceil logn \rceil$,也是完美平衡的(任何叶子高度相差不超过1).

更重要的是Twist操作的特殊性.与旋转不同,Twist操作每次只会改变不超过3个结点,而无关节点的绝对位置是没有变化的,也就意味着 TT树支持堆式存储(p的左儿子为$p*2$,右儿子为$p*2+1$).堆式存储具有实现简单,空间小,速度快等优点,一定程度上弥补了其插入删除复杂度稍高(约$n^{0.4}$)的缺点.

同时,TT树也是笔者知道的唯一支持堆式存储的二叉索引树.

5 结语

TT树作为二叉索引树,插入删除复杂度稍高,但其严格树高,特别是支持堆式存储的优点,使它可以用于静态存储,而无需指针结构.

参考文献

无