平衡树初步

李淳风

长郡中学

2024 年 7 月 2 日

BST

给定一棵二叉树，树上的每个节点都带有一个权值，称为节点的“关键码”。所谓“BST”性质，是指，对于树中的任意一个节点：

- 该节点的关键码不小于它的左子树中任意节点的关键码；
- 该节点的关键码不大于它的右子树中任意节点的关键码。

满足上述性质的二叉树就是一棵“二叉查找树”(BST)。显然，二叉查找树的中序遍历是一个关键码非严格单调递增的节点序列。

BST 的建立

为了避免越界，减少边界情况的特殊判断，我们一般在 BST 中额外插入一个关键码为正无穷，和一个关键码为负无穷的节点。仅由这两个节点构成的 BST 就是一棵初始的空 BST。

为了简便起见，在接下来的操作中，假设 BST 不会含有关键码相同的节点。若含有关键码相同的节点，只需给每个节点多记录一个信息 cnt，表示该节点的出现次数即可。

```
struct BST{
  int l,r;//左儿子和右儿子在数组中下表
  int val;//节点关键码
}a[SIZE];//数组模拟链表
int tot,root,INF=1<<30;
int NewNode(int val){
  a[++tot].val=val;
  return tot;
}
void Build(){
  NewNode(-INF),NewNode(INF);
  root=1,a[1].r=2;
}
```

BST 的检索

在 BST 中检索是否存在关键码为 val 的节点。
将变量 p 初始为根节点 $root$，执行以下操作：

- 若 p 的关键码等于 val，则已经找到。
- 若 p 的关键码大于 val，则递归 p 的左儿子进行检索。若左儿子为空则不存在 val。
- 若 p 的关键码小于 val，则递归 p 的右儿子进行检索。若右儿子为空则不存在 val。

```
int Get(int p,int val){
  if(p==0) return 0;       //检索失败
  if(val==a[p].val) return p;
  return val<p.val ? Get(a[p].l,val) : Get(a[p].r,val);
}
```

BST 的插入

在 BST 中插入一个新的值 val。
与 BST 的检索过程类似。在发现要走向的 p 的子节点为空，说明 val 不存在时，直接建立关键码为 val 的新节点作为 p 的子节点。

```
void Insert(int &p,int val){
  int (p==0){
    p=NewNode(val);
    return;
  }
  if(val==a[p].val) return;
  if(val<a[p].val) Insert(a[p].l,val);
  else Insert(a[p].r,val);
}
```

BST 求前驱/后继

以后继为例，val 的后继指的是在 BST 中关键码大于 val 的前提下，关键码最小的节点。
初始化 ans 为关键码为正无穷的节点编号。然后，在 BST 中检索 val。在检索过程中，每经过一个节点，都检查该节点的关键码，判断能否更新所求的后继 ans。
检索完成后，有三种情况：

- 没有找到 val。
- 找到了关键码为 val 的节点 p，但 p 没有右子树。
- 找到了关键码为 val 的节点 p，且 p 有右子树。

对于前两种情况，ans 即为所求。而对于第三种情况，我们在 p 的右子树中一直往左走，就找到了 p 的后继。

BST 求前驱/后继

```
int GetNext(int val){
  int ans=2;
  int p=root;
  while(p){
    if(val==a[p].val){
      if(a[p].r>0){
        p=a[p].r;
        while(a[p].l>0) p=a[p].l;
        ans=p;
      }
      break;
    }
    if(a[p].val>val && a[p].val<a[ans].val) ans=p;
    p= val<a[p].val ? a[p].l : a[p].r;
  }
  return ans;
}
```

BST 的节点删除

从 BST 中删除关键码为 val 的节点。
首先，在 BST 中检索 val，找到节点 p。
若 p 没有儿子或者只有一个儿子，则直接删除 p，用 p 的子节点代替 p 的位置。注意我们这样做并没有实质上删除 p 节点，只是从 $root$ 出发不再能到达 p，p 节点本身的信息还是存在的，相当于把 p 节点从 BST 中摘了出来。
若 p 同时有两个儿子，则在 BST 中求出 val 的后继节点 $next$。因为 $next$ 最多只有一个儿子，所以这个节点是可以直接被删除的。所以我们可以先把 $next$ 从 BST 中删掉，再让 $next$ 顶替 p 的位置。

BST 的节点删除

```
void Remove(int &p,int val){
  if(p==0) return;
  if(val==a[p].val){
    if(a[p].l & a[p].r == 0)  //p 至多只有一个儿子
      p = a[p].l+a[p].r;
    else{
      int nt=a[p].r;
      while(a[nt].l) nt=a[nt].l;      //找后继
      Remove(a[p].r,a[nt],val);       //先把后继节点从树中摘出来
      a[nt].l=a[p].l,a[nt].r=a[p].r;  //再替换 p
      p=next;
    }
  }
  if(val<a[p].val) Remove(a[p].l,val);
  else Remove(a[p].r,val);
}
```

Treap

在随机数据中，BST 一次操作的期望复杂度是 $O(\log n)$。然而，BST 很容易退化。例如，在 BST 中依次插入一个有序序列，将会得到一条链，平均每次操作的复杂度为 $O(N)$。我们称这种左右子树大小相差很大的 BST 是“不平衡”的。自然，我们可以通过各种方法来维持 BST 的平衡，从而产生了各种平衡树。接下来我们介绍一种入门级平衡树：Treap。

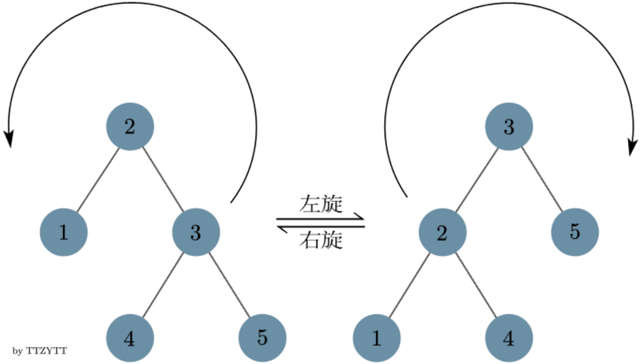
Treap

在随机数据中，BST 一次操作的期望复杂度是 $O(\log n)$。然而，BST 很容易退化。例如，在 BST 中依次插入一个有序序列，将会得到一条链，平均每次操作的复杂度为 $O(N)$。我们称这种左右子树大小相差很大的 BST 是“不平衡”的。自然，我们可以通过各种方法来维持 BST 的平衡，从而产生了各种平衡树。接下来我们介绍一种入门级平衡树：Treap。

中序遍历为相同序列的二叉查找树是不唯一的。它们维护的是同一组数值，只不过树的形态不同。因此，我们可以在维护 BST 性质的基础上，通过改变二叉查找树的形态，使得树上每个节点的左右子树大小达到平衡，从而使得整棵树的深度维护在 $O(\log n)$。

旋转

改变形态并保持 BST 性质的方法就是旋转。最基本的操作称为“单旋转”，它又分为“左旋”和“右旋”。原本的父亲节点变成左儿子就是“左旋”，变成右儿子就是“右旋”。



旋转

左、右旋可以看作一个节点绕父亲节点向左或者向右旋转。但由于我们保存的 Treap 信息中不包括父亲节点，因此我们统一以“旋转前处于父亲节点位置”的节点作为函数参数。
设旋转前父亲节点为 y,y 的左儿子为 x，x 的左右子树分别为 A、B，我们进行一次右旋操作 zig。
右旋操作在保持 BST 性质的基础上，把 x 变为 y 的父亲节点。因为 y 的关键码大于 x 的关键码，所以 y 应该作为 x 的右子节点。在这之后，y 的左儿子空了出来，而原本 x 的右子树 B 则需要另外找一个地方安放。于是 B 正好作为 y 的左子树。

```
void zig(int &p){
  int q=a[p].l;
  a[p].l=a[q].r,a[q].r=p;
  p=q;
}
void zag(int &p){
  int q=a[p].r;
  a[p].r=a[q].l,a[q].l=p;
  p=q;
}
```

Treap

合理的旋转操作可以使 BST 平衡，现在我们有了旋转操作，就需要考虑如何使用它来保持平衡了。
之前我们提到过，随机的 BST 是趋于平衡的。Treap 正是利用了这一点，虽然 BST 不能随机，但我们可以把“随机”用于堆性质上，因为随机的堆高度也是 $O(\log n)$ 的。
Treap=Tree+Heap，树和堆的合体。Treap 在插入每个新节点时，给该节点随机生成一个额外的权值。然后类似二叉堆，插入一个节点后自底向上开始检查，如果某个节点不满足大根堆性质，就执行单旋操作。
特别地，对于删除操作，因为 Treap 支持旋转，我们可以直接找到需要删除的节点，并把它向下旋转到叶子节点，最后直接删除。
总而言之，Treap 通过适当的单旋转，在维持节点关键码满足 BST 性质的同时，还使每个节点上随机生成的额外权值满足大根堆性质。由于树高是 $O(\log n)$ 级别的，常规的二叉树操作都是 $O(\log n)$ 的复杂度。

非旋 Treap

需要注意的是，之前介绍的 Treap 只能够实现单点操作，对于区间操作是无能为力的。为了实现区间操作（例如区间翻转），我们介绍一种不基于旋转的 Treap，又称为 FHQ-Treap。
FHQ-Treap 没有旋转操作，它的核心操作是两类：

- split 操作。该操作把一棵平衡树，按照关键码或者排名拆分为两棵平衡树。若按关键码 x 拆分，则拆分出来的两棵树中，一棵所有关键码都小于 x，另一棵全都大于 x。
- merge 操作。把两棵 Treap 合并为一棵 Treap，需要保证一棵中的关键码全都比另一棵小。

通过这两种操作，FHQ-Treap 不但可以实现旋转 Treap 的所有功能，还可以进行区间操作，甚至可持久化。

Split 操作

```
void split(int pos,int val,int &l,int &r){
  if(!pos){   //pos 是当前 Treap 的根节点
    l=r=0;    //如果当前节点不存在，肯定没得分
    return;
  }
  push_down(pos);
  if(a[pos].val<=val){//如果当前节点关键码已经小于 val 了
    l=pos;             //那么左儿子全部都会划分到 l 树中
    split(a[pos].r,val,a[l].r,r); //接着分右子树
    push_up(l);
  }
  else{//否则把右儿子全部划分到 r 这棵树中，接着分左子树
    r=pos;
    split(a[pos].l,val,l,a[r].l);
    push_up(r);
  }
}
```

Merge 操作

```
int merge(int l,int r){//l,r 为两棵 Treap 根节点
  if(!l || !r) return l+r;
  push_down(l);
  push_down(r);
  if(a[l].randval<=a[r].randval){//满足大根堆性质
    a[l].r=merge(a[l].r,r);//l 作为根, 右子树与 r 合并
    push_up(l);   //如果维护了各种信息, 一定要记得更新
    return l;
  }
  else{
    a[r].l=merge(l,a[r].l);   //同上
    push_up(r);
    return r;
  }
}
```

其它操作

对于 insert 操作，如果插入的值为 val，可以先把 Treap 按照 val 拆分成两棵子树 A 和 B，其中 A 的所有节点关键码都不超过 val；再把 A 按照 $val-1$ 拆成两棵子树 C 和 D，如果 D 非空就把该节点出现次数加一，如果 D 为空则把 val 插入 D 中，再依次把 C、D 合并为 A，再把 A、B 合并。
对于 Delete 操作同理，拆成三棵子树之后把出现次数减一即可。
如果需要按照排名拆分为两棵子树，则需要额外记录子树大小 $size$。拆分思路不变。
如果我们用 FHQ-Treap 维护数组，现在需要操作区间 $[l,r]$；那么先按照排名 r 拆分为 A 和 B，再把 A 按照排名 l 拆分成 C 和 D 即可。可以和线段树一样打标记，同样下传标记，并依据儿子信息来更新自身。