数据结构知识点乱滚

徐骁扬 Xun_Xiaoyao

动态开点线段树

有一个长度为 10^9 的初始全为 0 的序列，要求支持区间加，查询区间和。**强制在线**。

Bonus：如果初始序列为 $1, 2 \ldots 10^9$，如何处理。

动态开点线段树

~~平衡树也可以做~~。

由于有强制在线，就没有办法进行离散化了，所以直接建一个大小为 $V = 10^9$ 的线段树！

此时，区间加和查询区间和的时间复杂度均为 $O(\log V)$，可以接受。但是建立线段树的复杂度为 $O(V)$，无法接受。

假设有 m 次操作，那么修改和查询的复杂度就是 $O(m \log V)$。

动态开点线段树

发现 $O(m\log V)$ 比 $O(V)$ 小，虽然开了 $O(V)$ 的空间建立线段树，但是实际上用到的节点的数量只有最多 $O(m\log V)$ 个，其余的节点都是没有被访问过的。它的权值都是初始值，是可以根据对应区间直接得到的。

所以实际上，需要维护的，只有那 $O(m\log V)$ 个，被修改过，**权值不一定为初值**的节点。

动态开点线段树

因此，可以尝试这样的算法：

初始的时候线段树只包含一个根节点（甚至可以不包含空节点），其他的节点，只有在**首次**被修改的时候再建立节点并附上初值。

这样，空间复杂度就和时间复杂度一致，为 $O(m\log V)$。这个在修改的过程中动态的增加线段树的节点，就叫做**动态开点**。

可持久化线段树

有一个长度为 10^9 的初始全为 0 的序列，要求支持区间加，查询区间和，回到某一个历史版本。**强制在线**。

可持久化线段树

如果要能够支持在线地回到某一个历史版本，也就意味着每一个历史版本的线段树都需要被存储下来。

但是每一次暴力的复制一遍所有节点肯定不现实。

可持久化线段树

仿照上面动态开点的思路，由于每一次修改只会修改 $O(\log V)$ 个节点，只有这些节点的值被改变了，其余节点的值完全相同。

对于这需要修改的 $O(\log V)$ 个节点重新建立一个节点修改，其余的节点直接继承上一个版本的。

时间复杂度和空间复杂度均为 $O(m\log V)$。

可持久化线段树

事实上所有的可持久化算法，基本都是基于这个原理，每一次修改的节点的数量是**严格** $O(\log n)/O(\log^2 n)$ 的可接受复杂度，那么对于这些要修改的节点重新建立节点并修改，就能够支持可持久化了。

这也就意味着，能够可持久化的数据结构算法，必须要求单次的复杂度是严格或期望正确的，不能只是总体势能正确的。

平衡树

要求维护一个集合，支持：

- 加入一个数 x，
- 删除一个数 x，
- 查询一个数的前驱，
- 查询一个数的后继，
- 查询第 k 小的数，
- 查询 $> x$ 的最小数。

Splay

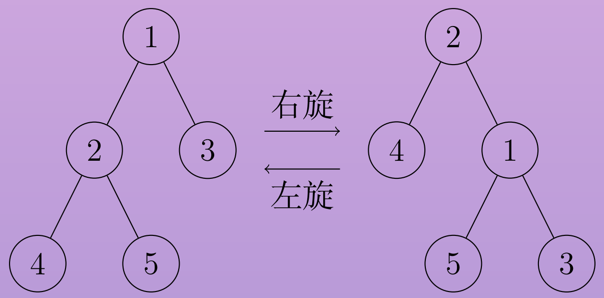
一种正常人无法理解的平衡树算法。

一种通过旋转来保证时间复杂度的算法。

Splay

其核心算法为 `Splay` 操作，可以将一个指定的节点通过旋转移动到根。

而 `Splay` 移动到根的方式，就是通过“旋转” `rotate` ，也就是在不改变中序遍历的情况下，交换 1 和 2 的父子关系。



具体的算法实现，根据 x 和 fa_x 的父子关系分两种情况处理即可。

Splay

`Splay` 操作是将指定的节点 x 移动到根，具体的过程中有三种操作：

1. zig：如果 x 的父亲为根，直接旋转即可。
2. zig-zig：如果 x 的父亲不为根，且 x 和 fa_x 同时为左/右儿子，则先旋转 fa_x 再旋转 x。
3. zig-zag：如果 x 的父亲不为根，且 x 和 fa_x 为不同侧儿子，则旋转两次 x。
4.

给出一种算法实现：

```
for(int k=s[x].fa;k=s[x].fa,k!=rt;rotate(x))
        if(s[k].fa!=rt) rotate(be_lson(k)==be_lson(x)?k:x);
```

Splay

可以证明，该算法的均摊复杂度为单次 $O(\log n)$。

注意，Splay 算法并不保证每一个节点的深度的复杂度，所以通过二分查找某一个点 x 的信息之后，需要将 x `Splay` 到根以保证复杂度。

FHQ-Treap

Treap 就是 Heap（堆）和 Tree（树）合成的词。

也就是一个有大根堆性质的二叉查找树。

构建一棵以 $1,2\dots n$ 为中序遍历的二叉树，给每一个点赋一个随即权值 val，按照 val 建立一个大根堆，即每一次选择 val 最大的点为根节点，递归到左右两侧分别处理。

FHQ-Treap

由于 val 是均匀随机选择的，所以构建大根堆的过程等价于每次随机选择一个点位根。

可以证明，此时每一个点的期望深度是 $O(\log n)$ 的。

FHQ-Treap

FHQ-Treap 的算法核心为 `split` 和 `merge` 操作，也就是分裂和合并。

`split` 操作是将节点 $1 \sim n$ 构成的平衡树分成节点 $1 \sim k$ 和节点 $k+1 \sim n$ 构成的两棵平衡树 T_1 和 T_2。

`merge` 操作是将点 $1 \sim k$ 和节点 $k+1 \sim n$ 构成的两棵平衡树 T_1 和 T_2 合并成一个节点 $1 \sim n$ 构成的平衡树（该过程并不支持归并）。

FHQ-Treap

`split` 算法流程

考虑当前根节点 rt 是属于 T_1 还是 T_2。

- 如果是属于 T_1，则将 rt 设为 T_1 的根 x，x 的左儿子不变，递归处理将 rt 的右儿子分裂成 x 的右儿子和 T_2。
- 如果是属于 T_2，则将 rt 设为 T_2 的根 y，y 的右儿子不变，递归处理将 rt 的左儿子分裂成 T_1 和 y 的左儿子。

FHQ-Treap

`split` 实现范例

```
void split(int pos,int k,int &x,int &y)
{
        if(!pos) return x=y=0,void();
        if(s[pos].num<=k) x=pos,split(s[pos].rson,k,s[pos].rson,y);
        else y=pos,split(s[pos].lson,k,x,s[pos].lson);
        update(pos);
        return;
}
```

注：这里实现的是按照 `num` 的大小分成 $\leq k$ 和 $> k$ 的两部分。

FHQ-Treap

`merge` 算法流程

比较 T_1 和 T_2 的根 x 和 y 的权值 `val` 大小：

- 如果 x 的 `val` 更大，则令 x 为新树的根 rt，递归合并 x 的右儿子和 y。
- 如果 y 的 `val` 更大，则令 y 为新树的根 rt，递归合并 x 和 y 的左儿子。

FHQ-Treap

`merge` 实现范例

```cpp
int merge(int x,int y)
{
        if(x&&y)
        {
                if(s[x].val>s[y].val) return s[x].rson=merge(s[x].rson,y),update(x),x;
                else return s[y].lson=merge(x,s[y].lson),update(y),y;
        }
        else return x^y;
}
```

WBLT

前面两种平衡树都是 Unleafy 的，有的时候在对于复杂信息的 `update` 中并不方便，因为需要同时考虑左右儿子和当前节点。如果平衡树是 Leafy 的，也就是类似于线段树，那么 `update` 的过程将会简单一些。

那么最简单的想法就是维护一个支持插入节点的“线段树”。

WBLT

线段树的复杂度正确是基于左右儿子的大小相近，因此可以对于每一个系欵但维护一个平衡度 $\rho = \dfrac{\min(siz_{ls}, siz_{rs})}{siz_x}$，显然有 $\rho \in (0, 0.5]$。最理想的情况就是 $\rho = 0.5$，但是在动态的过程中保持这一点并不现实。

所以可以设置一个平衡系数 α，在 $\rho < \alpha$ 的时候再做调整，$\rho \geq \alpha$ 的时候就保持不变。

此时，每一个节点的深度便是 $O(\log_{\frac{1}{1-\alpha}} n)$，也就是 α 直接决定了算法的常数。

WBLT

插入 x 的时候，直接找到需要加入的节点对应的前驱 w，新建一个节点代替 w 的位置，并让其左右儿子为 w 和 x。

然后从 x 开始向上检查，每一个节点是否还满足 α 的限制，如果不满足，就通过函数 `maintain` 来调整使得其满足。

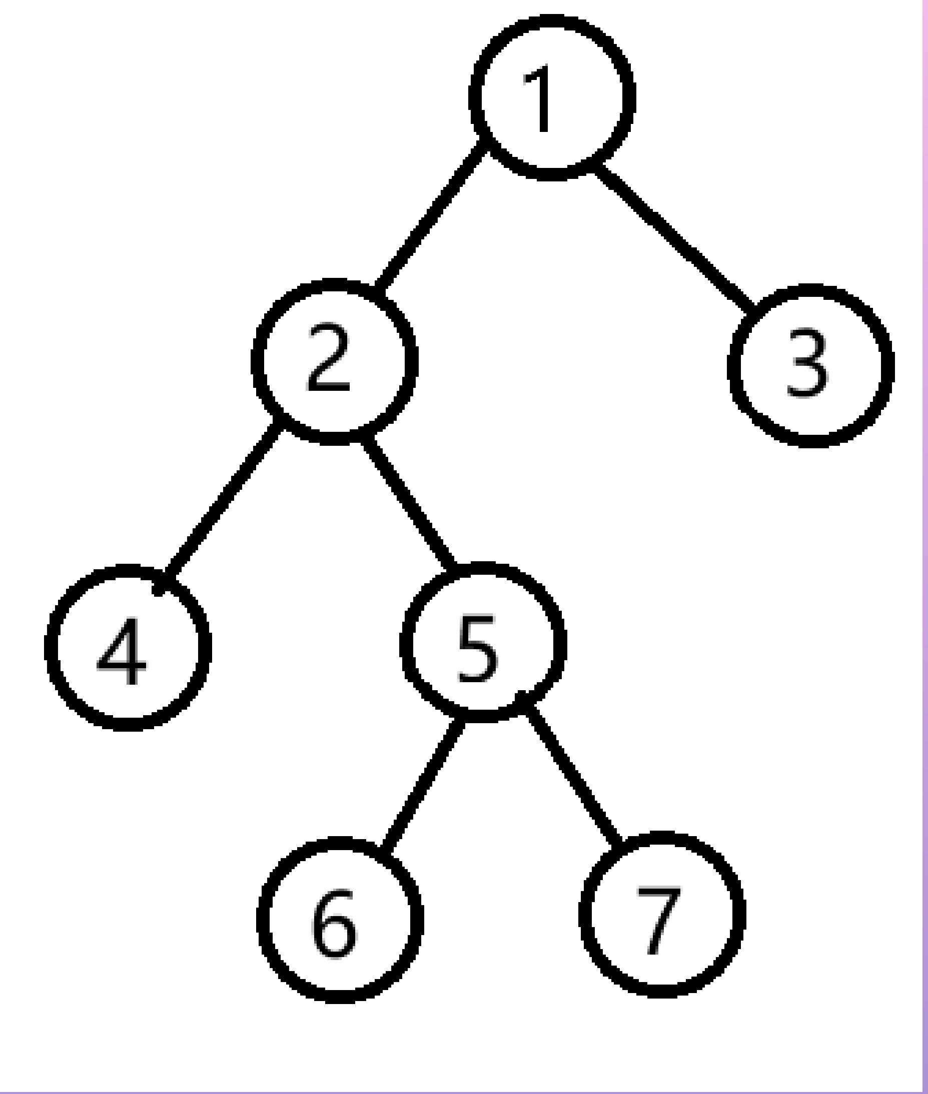
WBLT

记住这张图，后面要用：

假设更重的那边是左儿子 2，也就是

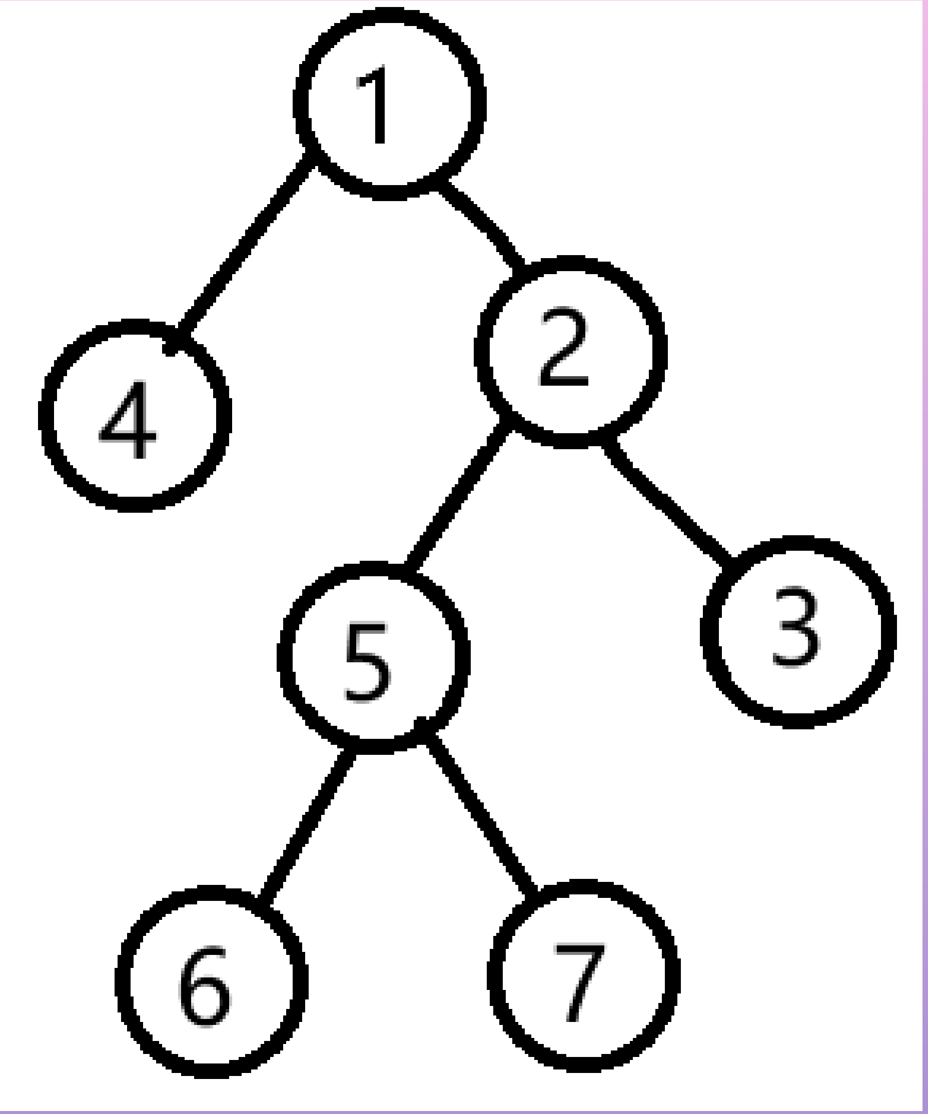
$siz_2 > (1-\alpha)siz_1$，即 $siz_2 > \dfrac{1-\alpha}{\alpha}siz_3$。

现在有两种调整方法。



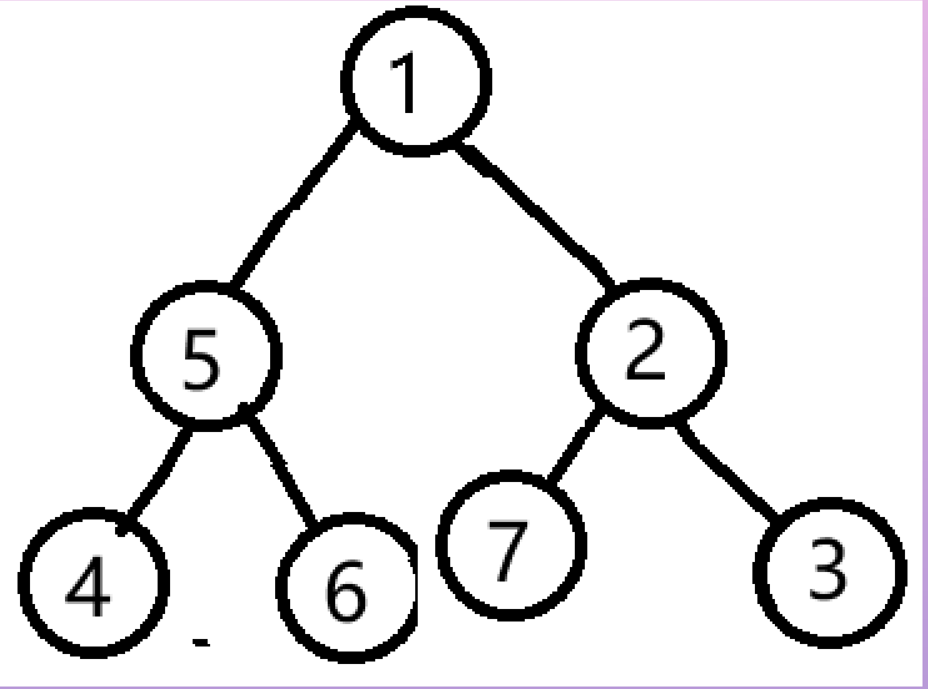
WBLT

在 5 的大小较小，有 $siz_5 \le \beta siz_2$ 的时候，对 2 进行一次旋转，使得其变成右图满足条件。



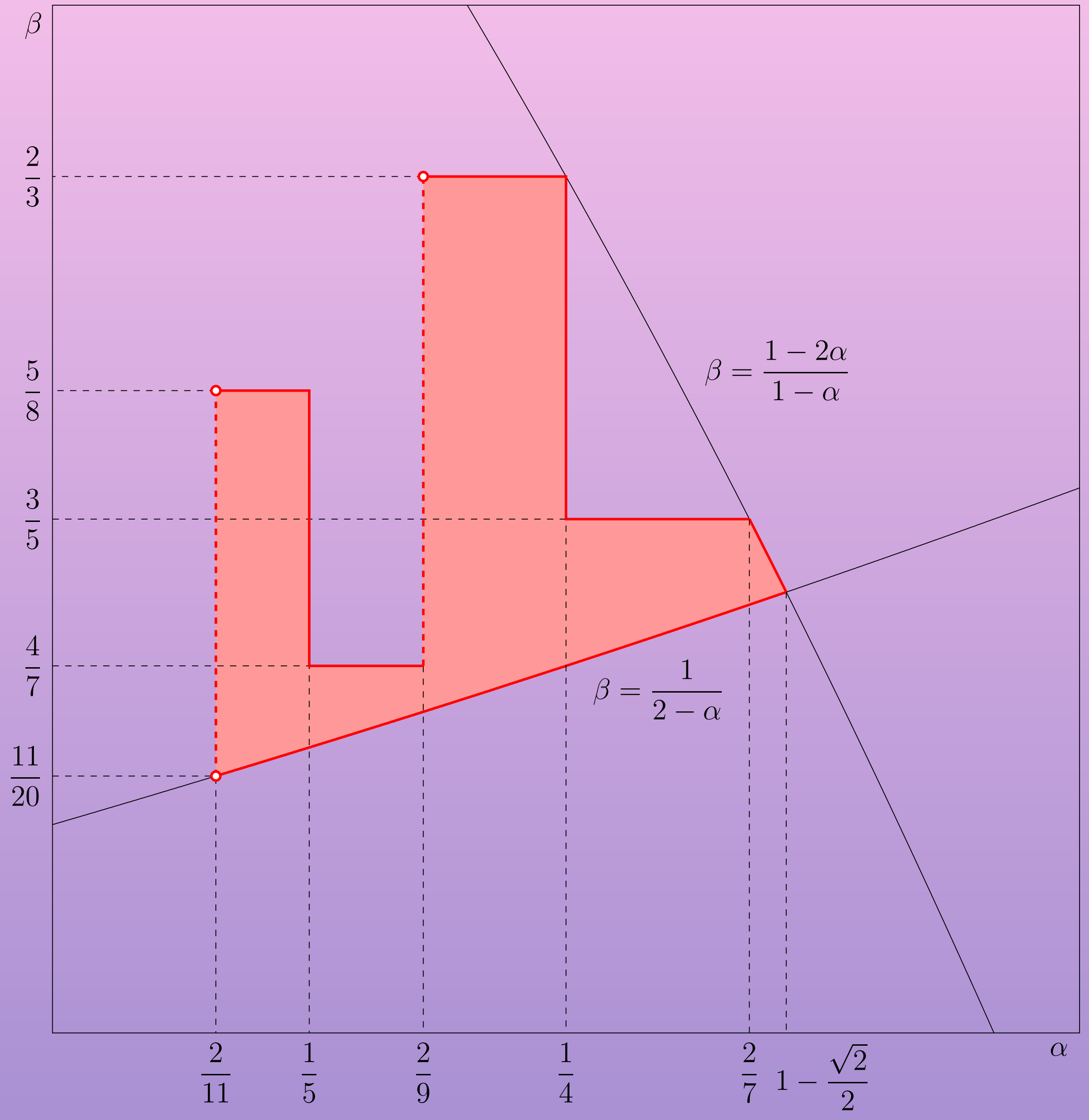
WBLT

否则，说明 5 的过大的，需要将其拆成两个部分，先对 5 进行一次旋转，然后对 2 进行一次旋转，得到右图。



WBLT

通过数学证明，在右图红色区域选取 α 和 β 时，上述策略时可行的。下面将给出取 $\alpha=\dfrac{1}{4},\beta=\dfrac{2}{3}$ 时的代码实现。



WBLT

`maintain` 实现范例

```
void maintain(int x)
{
        if(s[s[x].ls].siz>3*s[s[x].rs].siz)
        {
                if(s[s[s[x].ls].rs].siz>2*s[s[s[x].ls].ls].siz)
                        rotate(s[x].ls,0);
                rotate(x,1);
        }
        else if(s[s[x].rs].siz>3*s[s[x].ls].siz)
        {
                if(s[s[s[x].rs].ls].siz>2*s[s[s[x].rs].rs].siz)
                        rotate(s[x].rs,1);
                rotate(x,0);
        }
}
```

树链剖分

很多时候树上的问题并不好处理，但是序列上却很容易去做。

那么就不妨直接将树分成若干条链来进行处理。

重链剖分

对于一条树上路径上的信息，希望将其分成尽可能少的链上尽可能少的区间。

考虑最特殊的一种路径：从一个点到根节点。我们的目标是让每一个点到根的路径上，经过的不同的链数量尽可能少，或者说在一个能接受的范围内。

假设对于每一个点，称其大小最大的儿子为重儿子，其余的儿子为轻儿子。保留其与重儿子的连边构成重链，与其余儿子的边为轻边。那么一个节点到根经过的链的数量，就是一个节点到根路径上轻边的数量 $+1$。

重链剖分

给出一棵有 n 个节点的树，点有点权，要求支持：给路径 (u, v) 上的点权值加 x，查询路径 (u, v) 上的点权值之和。

重链剖分

而根据轻重儿子的性质，每当从一个点跳轻边移动到父亲时，所在的子树大小将会翻倍。而最终根节点的大小是 $O(n)$ 的，因此跳轻边的次数为 $O(\log n)$ 次。

进而可以证明，任何一条树上路径经过的轻边数量都是 $O(\log n)$ 的。

重链剖分

此时，所有在序列上能够做得操作：例如区间加，区间和查询。在树上都可以拆成在 $O(\log n)$ 条链上做对应的操作。因此树链剖分一般会和线段树、平衡树等数据结构结合在一起使用。

长链剖分

给定一棵树，要求离线地确定每一个点 x 的每一个深度的儿子的数量。

长链剖分

需要维护的信息和深度有关，那么一个很棒的想法就是让每个节点直接继承某个儿子的信息，然后将其他节点的信息暴力加入。

仿照重链剖分的思想，选择重儿子的信息继承，就可以获得 DSU on tree 算法！时间复杂度 $O(n \log n)$。

长链剖分

发现对于一个节点而言，它的信息量并不是 $O(siz)$ 的，而是 $O(dep)$ 的。那么选择 siz 最大的儿子继承信息看着就很别扭。

所以选择 dep 最大，也就是深度最深的儿子作为长儿子，$O(1)$ 继承它的信息。将其余短儿子的信息以 $O(dep)$ 的代价加入。

发现此时可以证明时间复杂度时 $O(n)$ 的。

长链剖分

为了方便实现 $O(1)$ 继承这个操作，一般会使用指针实现。将所有信息都存在一个足够长的数组上。对于 x 及其长儿子 y，直接令指针 `f[x]=f[y]-1`。因为 y 子树内深度为 i 的点，在 x 子树内的深度应为 $i+1$，对应了前面指针的 `f[x][i+1]=f[y][i]`。

因此，长链剖分可以用于一些和深度相关的动态规划问题，例如 DP 状态为 $f_{i,j}$，其中 i 为节点，j 为深度。如果 DP 信息可以直接通过继承来维护，就可以将复杂度优化至 $O(n)$。

线段树合并

有 n 个集合 $S_i = \{i\}$，需要支持以下操作：

1. 合并两个集合 S_x 和 S_y 得到新集合 S_z，保证 S_x 和 y 之后不再参与任何操作。
2. 查询一个集合 S_x 内 $[l, r]$ 之前的元素数量。

线段树合并

对于操作 2，线段树平衡树都可以作。但是对于平衡树，操作 1 使用启发式合并的复杂度是 $O(n\log^2 n)$ 的。

那么就尝试用线段树来处理操作 1。

线段树合并

对于每一个集合建立一个动态开点线段树，那么对于每一个初始集合，只有一条链上是有有效信息的。

而将两个集合 S_x 和 S_y 合并时，合并得到的新的集合 S_z 对应的动态开点线段树上，有有效信息的节点，对应的应当就是 S_x 和 S_y 对应的动态开点线段树上节点的并。

由于 S_x 和 S_y 之后不参与任何操作，所以直接将 S_x 和 S_y 对应的两棵线段树合并起来，作为 S_z 的线段树。

线段树合并

考虑合并 pos_1 和 pos_2 对应的子树：

1. 如果 pos_1 和 pos_2 只有一个有值，则保留其中有值的那个。
2. 如果 pos_1 和 pos_2 均有值，则递归至其左右儿子分别处理，然后合并此处的信息。

线段树合并

参考实现：

```
void merge(int &pos1,int pos2,int l,int r)
{
  if(pos1&&pos2)
  {
    merge(s[pos1].lson,s[pos2].lson,l,mid);
    merge(s[pos1].rson,s[pos2].rson,mid+1,r);
    merge_info(pos1,pos2)
  }
  else pos1+=pos2;
  return;
}
```

线段树合并

发现只有在遇到第二种情况的时候才会有额外的递归，同时也会产生一个不会再使用的节点（上页代码中的 `pos2` ）。因此，线段树合并的复杂度和合并过程中弃用的节点数量是相同的。

每一个节点只会被弃用一次，且初始只有 $O(n\log n)$ 个节点，所以线段树合并的时间复杂度是 $O(n\log n)$。

线段树合并

由于线段树合并的复杂度是势能的，所以每一个点只能被合并一次，因此对其使用可持久化仅能支持查询某一个版本的信息，而不能支持将一棵线段树与多棵线段树分别进行合并，这样的复杂度是没有保证的。

线段树合并

由于这个合并是树形的，所以一般线段树合并都会用来解决一些树上的问题。而且不需要可持久化的问题上，线段树合并的空间是可以优化到 $O(n)$ 的。