浅谈一些信息合并的处理方式

冯政玮

重庆市巴蜀中学

2024.1.14

声明与主要内容

信息合并在信息学竞赛中是非常重要的部分，今天我们将从朴素信息合并，结构信息合并共两个角度简要探讨一下在算法竞赛中常见的一些信息合并在具体问题中的处理方式与技巧。

声明与主要内容

信息合并在信息学竞赛中是非常重要的部分，今天我们将从朴素信息合并，结构信息合并共两个角度简要探讨一下在算法竞赛中常见的一些信息合并在具体问题中的处理方式与技巧。
今天的内容比较通俗易懂，希望同学们能认真听，也希望同学们能理解

声明与主要内容

信息合并在信息学竞赛中是非常重要的部分，今天我们将从朴素信息合并，结构信息合并共两个角度简要探讨一下在算法竞赛中常见的一些信息合并在具体问题中的处理方式与技巧。
今天的内容比较通俗易懂，希望同学们能认真听，也希望同学们能理解
交流分享内容仅为论文一部分更多例题和内容详见我的论文

声明与主要内容

信息合并在信息学竞赛中是非常重要的部分，今天我们将从朴素信息合并，结构信息合并共两个角度简要探讨一下在算法竞赛中常见的一些信息合并在具体问题中的处理方式与技巧。
今天的内容比较通俗易懂，希望同学们能认真听，也希望同学们能理解
交流分享内容仅为论文一部分更多例题和内容详见我的论文
作者才疏学浅，若所述内容有误请指出。

前置知识

信息一般指在具体问题中需要维护的东西，它可以是一个权值，一个节点，一条边，一个式子等。

前置知识

信息一般指在具体问题中需要维护的东西，它可以是一个权值，一个节点，一条边，一个式子等。
而通常信息合并的过程指将信息类比于集合间的元素，将若干信息按照某些方式并在一起，维护一个信息结构，使得可以在结构中可以快速查找到想要的信息，不同的信息对应着不同的处理方式。

启发式合并

启发式合并是一种信息合并方式，通常形如有若干个信息集合总大小为为 n，操作形如合并两个存在的集合。

启发式合并

启发式合并是一种信息合并方式，通常形如有若干个信息集合总大小为为 n，操作形如合并两个存在的集合。
具体地，对于两个大小分别是 $A, B(A < B)$ 的信息集合，我们将 A 集合的元素取出，插入到 B 集合中。

启发式合并

启发式合并是一种信息合并方式，通常形如有若干个信息集合总大小为为 n，操作形如合并两个存在的集合。
具体地，对于两个大小分别是 $A, B(A < B)$ 的信息集合，我们将 A 集合的元素取出，插入到 B 集合中。
复杂度分析：对于每个信息元素，当它被执行插入操作时，其所处的集合大小一定翻倍，所以对于每个信息元素只会被执行至多插入 $\log n$ 次。

启发式合并与整体二分结合

例题:
给定一张 n 个点 m 条边的带权无向图，q 次询问，每次给定 x, y 询问 x 到 y 的路径的最小权值，一条路径的权值定义为路径上边权中的次大值，如果仅经过一条边，次大值定义为 0。可视为 n, m, q 同阶。

启发式合并与整体二分结合

特殊讨论最大值的一条边，最小化次大值，整体二分处理。

启发式合并与整体二分结合

特殊讨论最大值的一条边，最小化次大值，整体二分处理。
问题转化为如何快速验证通过所有边权在 $1 \sim mid$ 和一条额外边使得 x, y 相互可达。

启发式合并与整体二分结合

特殊讨论最大值的一条边，最小化次大值，整体二分处理。
问题转化为如何快速验证通过所有边权在 $1 \sim mid$ 和一条额外边使得 x, y 相互可达。
维护连通块之间边的映射关系，启发式合并递归再撤回。

启发式合并与整体二分结合

特殊讨论最大值的一条边，最小化次大值，整体二分处理。
问题转化为如何快速验证通过所有边权在 $1 \sim mid$ 和一条额外边使得 x, y 相互可达。
维护连通块之间边的映射关系，启发式合并递归再撤回。
注意到启发式合并的复杂度为均摊 $O(n \log n)$ 次插入操作，单次合并的时间复杂度并没有确切保证

启发式合并与整体二分结合

特殊讨论最大值的一条边，最小化次大值，整体二分处理。
问题转化为如何快速验证通过所有边权在 $1\sim mid$ 和一条额外边使得 x,y 相互可达。
维护连通块之间边的映射关系，启发式合并递归再撤回。
注意到启发式合并的复杂度为均摊 $O(n\log n)$ 次插入操作，单次合并的时间复杂度并没有确切保证
注意到对于 i 点而言，每次加入时总是在加入完 $1\sim i-1$ 后再加入 i 节点信息，可以视为顺序的执行了 $\log n$ 次从 $1\sim n$ 的启发式合并，所以复杂度正确。

朴素信息合并

更多朴素信息合并的不同角度与例题详见我的论文，限于篇幅不在今天介绍。

朴素信息合并

更多朴素信息合并的不同角度与例题详见我的论文，限于篇幅不在今天介绍。
接下来我们从平衡树与线段树的角度简单介绍以下结构信息合并。

平衡树合并

平衡树一般维护有序信息集合，在集合上可以支持若干操作如区间求和，区间权值更新等，平衡树的合并可以视为两集合求并集后得到的一颗新的平衡树。

平衡树合并

平衡树一般维护有序信息集合，在集合上可以支持若干操作如区间求和，区间权值更新等，平衡树的合并可以视为两集合求并集后得到的一颗新的平衡树。

一些平衡树合并可以做到较优的复杂度，但是由于其考察较少，平衡树自身常数较大，所以在 OI 中较优复杂度的平衡树合并少有人提及与实现，通常对于平衡树的合并都是以启发式合并的方式进行处理。

平衡树合并

平衡树一般维护有序信息集合，在集合上可以支持若干操作如区间求和，区间权值更新等，平衡树的合并可以视为两集合求并集后得到的一颗新的平衡树。
一些平衡树合并可以做到较优的复杂度，但是由于其考察较少，平衡树自身常数较大，所以在 OI 中较优复杂度的平衡树合并少有人提及与实现，通常对于平衡树的合并都是以启发式合并的方式进行处理。
我们将详细介绍复杂度较优的平衡树合并如何实现及具体细节并简要说明其复杂度。

finger search

finger search 指搜索的一种扩展，即维护上一次搜索得到的元素称为 finger，再基于 finger 元素做搜索操作。

finger search

finger search 指搜索的一种扩展，即维护上一次搜索得到的元素称为 finger，再基于 finger 元素做搜索操作。
具体地对于有序序列，定义 $d(x,y)$ 表示元素 x,y 集合中的排名之差，若上一次查找到的元素为 x，下一次查找元素为 y，其查找复杂度可以描述为一个关于 $d(x,y)$ 的函数如 $O(f(d(x,y)))$，我们称这个过程为 finger search。

finger search

finger search 指搜索的一种扩展，即维护上一次搜索得到的元素称为 finger，再基于 finger 元素做搜索操作。
具体地对于有序序列，定义 $d(x,y)$ 表示元素 x,y 集合中的排名之差，若上一次查找到的元素为 x，下一次查找元素为 y，其查找复杂度可以描述为一个关于 $d(x,y)$ 的函数如 $O(f(d(x,y)))$，我们称这个过程为 finger search。
finger search tree 定义为一种二叉搜索树，保留指向上一次操作的节点的指针，保留的节点称为 finger，同样的，我们定义 $d(x,y)$ 为中序遍历排名之差。

Splay

Splay 是一类二叉搜索树，在执行操作过程中使用旋转平衡其树高，保证其均摊时间复杂度，其结构比较经典，这里不再介绍其操作及复杂度证明。

Splay

Splay 是一类二叉搜索树，在执行操作过程中使用旋转平衡其树高，保证其均摊时间复杂度，其结构比较经典，这里不再介绍其操作及复杂度证明。

由于其每次操作后会将操作节点旋转到根，可以视根为 finger，于是每次操作可视为在上次的基础上做新的 finger search 操作。

Splay

Splay 是一类二叉搜索树，在执行操作过程中使用旋转平衡其树高，保证其均摊时间复杂度，其结构比较经典，这里不再介绍其操作及复杂度证明。

由于其每次操作后会将操作节点旋转到根，可以视根为 finger，于是每次操作可视为在上次的基础上做新的 finger search 操作。

在 [10, 11] 中有证明，其单次操作复杂度可以描述为 $O(\log d)$。

Splay

考虑 Splay 合并，我们将小的一颗平衡树按照中序遍历取出（即取出元素有序）顺序将其直接按照 Splay 的 insert 插入至第二棵平衡树中。

Splay

考虑 Splay 合并，我们将小的一颗平衡树按照中序遍历取出（即取出元素有序）顺序将其直接按照 Splay 的 insert 插入至第二棵平衡树中。不妨设小树的大小为 m 大树的大小为 n，设 m 个节点在 n 中的 rk 分别为 $D_1, D_2, ..., D_m$，对于第 $i(i > 1)$ 次插入的复杂度为 $O(\log d_i), d_i = D_i - D_{i-1}$。

Splay

考虑 Splay 合并，我们将小的一颗平衡树按照中序遍历取出（即取出元素有序）顺序将其直接按照 Splay 的 insert 插入至第二棵平衡树中。

不妨设小树的大小为 m 大树的大小为 n，设 m 个节点在 n 中的 rk 分别为 $D_1, D_2, ..., D_m$，对于第 $i(i > 1)$ 次插入的复杂度为 $O(\log d_i), d_i = D_i - D_{i-1}$。

讨论第一次，剩下插入复杂度为 $\sum_{i=2}^{m} \log d_i$。

Splay

考虑 Splay 合并，我们将小的一颗平衡树按照中序遍历取出（即取出元素有序）顺序将其直接按照 Splay 的 insert 插入至第二棵平衡树中。

不妨设小树的大小为 m 大树的大小为 n，设 m 个节点在 n 中的 rk 分别为 $D_1, D_2, ..., D_m$，对于第 $i(i > 1)$ 次插入的复杂度为 $O(\log d_i), d_i = D_i - D_{i-1}$。

讨论第一次，剩下插入复杂度为 $\sum_{i=2}^{m} \log d_i$。

有 $\log A + \log B \leq 2 \log \frac{A+B}{2}$，根据类似均值的推导，我们有 $\sum_{i=2}^{m} \log d_i \leq m \log \frac{n}{m}$。

Splay

考虑 Splay 合并，我们将小的一颗平衡树按照中序遍历取出（即取出元素有序）顺序将其直接按照 Splay 的 insert 插入至第二棵平衡树中。

不妨设小树的大小为 m 大树的大小为 n，设 m 个节点在 n 中的 rk 分别为 $D_1, D_2, ..., D_m$，对于第 $i(i > 1)$ 次插入的复杂度为 $O(\log d_i), d_i = D_i - D_{i-1}$。

讨论第一次，剩下插入复杂度为 $\sum_{i=2}^{m} \log d_i$。

有 $\log A + \log B \leq 2 \log \frac{A+B}{2}$，根据类似均值的推导，我们有 $\sum_{i=2}^{m} \log d_i \leq m \log \frac{n}{m}$。

即合并时总插入复杂度不超过 $O(\log n + m \log \frac{n}{m})$。

Splay

对于一个元素，考虑其若干次合并的贡献，设其所属集合 siz 的变化量为一个序列 $p_1 = 1 < p_2 < ... < p_k = n$，故当前节点消耗总复杂度为 $O(\sum_{i=1}^{k-1} \log \frac{p_{i+1}}{p_i}) = O(\log n)$

Splay

对于一个元素，考虑其若干次合并的贡献，设其所属集合 siz 的变化量为一个序列 $p_1 = 1 < p_2 < ... < p_k = n$，故当前节点消耗总复杂度为 $O(\sum_{i=1}^{k-1} \log \frac{p_{i+1}}{p_i}) = O(\log n)$

总合并复杂度为 $O(n \log n)$。

无旋 Treap

Treap 是一种通过随机权值优化的二叉搜索树，每个节点有一个键值和关联的随机优先级，每个节点的键值满足二叉搜索树的性质，每个节点的随机优先级满足堆的性质，顾名思义 Tree+Heap = Treap。

无旋 Treap

Treap 是一种通过随机权值优化的二叉搜索树，每个节点有一个键值和关联的随机优先级，每个节点的键值满足二叉搜索树的性质，每个节点的随机优先级满足堆的性质，顾名思义 Tree+Heap = Treap。
其保证复杂度的直接一种理解方式为视为当前是一个序列，然后每次选择随机优先级最高的作为根，然后划分为左右子树，注意到找随机优先值最高形如随机划分，则期望树高为 $O(\log n)$。

无旋 Treap

Treap 是一种通过随机权值优化的二叉搜索树，每个节点有一个键值和关联的随机优先级，每个节点的键值满足二叉搜索树的性质，每个节点的随机优先级满足堆的性质，顾名思义 Tree+Heap = Treap。
其保证复杂度的直接一种理解方式为视为当前是一个序列，然后每次选择随机优先级最高的作为根，然后划分为左右子树，注意到找随机优先值最高形如随机划分，则期望树高为 $O(\log n)$。
无旋 Treap 由于其易于实现，且能支持信息可持久化，在 OI 中比较常用。

无旋 Treap

Treap 是一种通过随机权值优化的二叉搜索树，每个节点有一个键值和关联的随机优先级，每个节点的键值满足二叉搜索树的性质，每个节点的随机优先级满足堆的性质，顾名思义 Tree+Heap = Treap。
其保证复杂度的直接一种理解方式为视为当前是一个序列，然后每次选择随机优先级最高的作为根，然后划分为左右子树，注意到找随机优先值最高形如随机划分，则期望树高为 $O(\log n)$。
无旋 Treap 由于其易于实现，且能支持信息可持久化，在 OI 中比较常用。
这里简单介绍无旋 Treap 的基本操作与启发式合并的方式，注意当前一颗平衡树维护的内部键值形如一个集合（互不相同）。

无旋 Treap

- Split(A,val)={C,D,x} 将 A 按照 val 分裂成 <val 与 >val 的两个部分，=val 权值的点 x（不存在则为空节点）。
- Merge(A,B)=C 对于 A,B 集合满足 $\max A < \min B$ （这里比较 A,B 的键值）新集合为 A,B 集合的并。
- Union(A,B)=C := $A \cup B$ 求 A,B 集合的并集合。

无旋 Treap

- Split(A,val)={C,D,x} 将 A 按照 val 分裂成 <val 与 >val 的两个部分，=val 权值的点 x（不存在则为空节点）。
- Merge(A,B)=C 对于 A,B 集合满足 $\max A < \min B$ （这里比较 A,B 的键值）新集合为 A,B 集合的并。
- Union(A,B)=C := $A \cup B$ 求 A,B 集合的并集合。

对于前两个函数是为人熟知的无旋 Treap 的普通合并方式，这里不再详细介绍，其期望复杂度分别为 $O(\log |A|), O(\log |A| + \log |B|)$。

无旋 Treap

对于启发式合并的过程我们可以视为 Treap 的 Union，这里详细介绍其处理方式：

无旋 Treap

对于启发式合并的过程我们可以视为 Treap 的 Union，这里详细介绍其处理方式：
对于合并 A,B Treap，不妨 A,B 根中 A 根随机权值优先级更高，取 A 根作为合并后的根，令 A 根的键值 k，将 B 按照键值 k 分为 C,D,x 分别对应 $< k, > k, = k$ 的信息，舍弃 x 节点，将 C 递归与 A 左子树合并得到当前左子树，D 递归与 B 右子树合并得到当前右子树。

无旋 Treap

对于启发式合并的过程我们可以视为 Treap 的 Union，这里详细介绍其处理方式：
对于合并 A,B Treap，不妨 A,B 根中 A 根随机权值优先级更高，取 A 根作为合并后的根，令 A 根的键值 k，将 B 按照键值 k 分为 C,D,x 分别对应 $< k, > k, = k$ 的信息，舍弃 x 节点，将 C 递归与 A 左子树合并得到当前左子树，D 递归与 B 右子树合并得到当前右子树。
对于 Union 合并两颗大小分别为 $n, m(m \leq n)$ 的平衡树其复杂度均为 $O(m \log \frac{n}{m})$。

无旋 Treap

对于启发式合并的过程我们可以视为 Treap 的 Union，这里详细介绍其处理方式：
对于合并 A,B Treap，不妨 A,B 根中 A 根随机权值优先级更高，取 A 根作为合并后的根，令 A 根的键值 k，将 B 按照键值 k 分为 C,D,x 分别对应 $< k, > k, = k$ 的信息，舍弃 x 节点，将 C 递归与 A 左子树合并得到当前左子树，D 递归与 B 右子树合并得到当前右子树。
对于 Union 合并两颗大小分别为 $n, m(m \leq n)$ 的平衡树其复杂度均为 $O(m \log \frac{n}{m})$。
上述复杂度分析在原论文 [14] 有详细介绍，较为复杂不在此处说明。

无旋 Treap

对于启发式合并的过程我们可以视为 Treap 的 Union，这里详细介绍其处理方式：
对于合并 A,B Treap，不妨 A,B 根中 A 根随机权值优先级更高，取 A 根作为合并后的根，令 A 根的键值 k，将 B 按照键值 k 分为 C,D,x 分别对应 $< k, > k, = k$ 的信息，舍弃 x 节点，将 C 递归与 A 左子树合并得到当前左子树，D 递归与 B 右子树合并得到当前右子树。
对于 Union 合并两颗大小分别为 $n, m(m \leq n)$ 的平衡树其复杂度均为 $O(m \log \frac{n}{m})$。
上述复杂度分析在原论文 [14] 有详细介绍，较为复杂不在此处说明。
根据类似 Splay 合并的上述分析有，总合并复杂度为 $O(n \log n)$。

无旋 Treap

这里请大家提醒注意，当存在键值重复元素，一般的写法是重定义 Split 函数为分裂成 $\leq val$ 和 $> val$ 的两个部分。

无旋 Treap

这里请大家提醒注意，当存在键值重复元素，一般的写法是重定义 Split 函数为分裂成 $\leq val$ 和 $> val$ 的两个部分。
注意，此时使用 Union 函数进行合并时可能导致树高退化，以下是一个极端情况的例子：

无旋 Treap

这里请大家提醒注意，当存在键值重复元素，一般的写法是重定义 Split 函数为分裂成 $\leq val$ 和 $> val$ 的两个部分。
注意，此时使用 Union 函数进行合并时可能导致树高退化，以下是一个极端情况的例子：
A,B 两树的所有节点的键值相同，考虑 Union(A,B)=D 的过程：

无旋 Treap

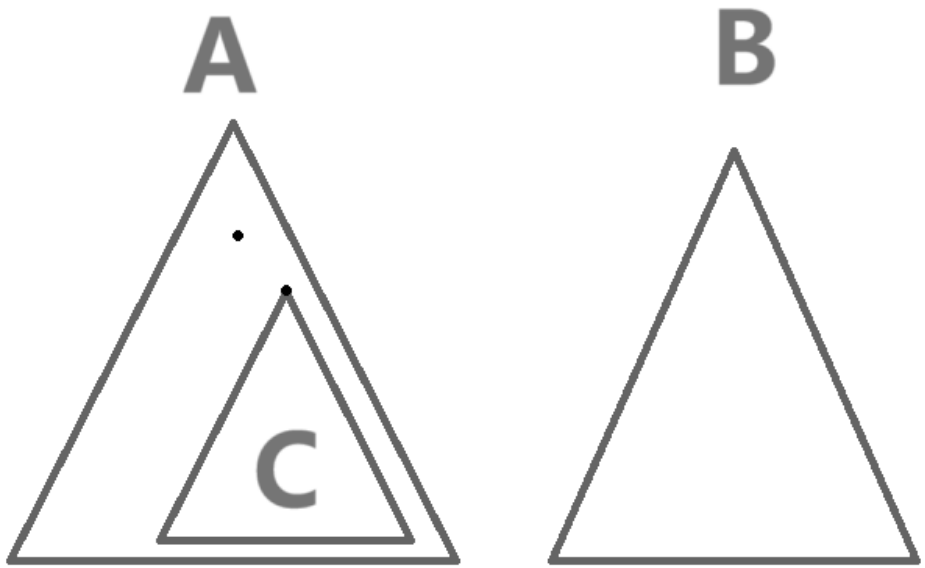

图: 相同权值 Union 的例子

无旋 Treap

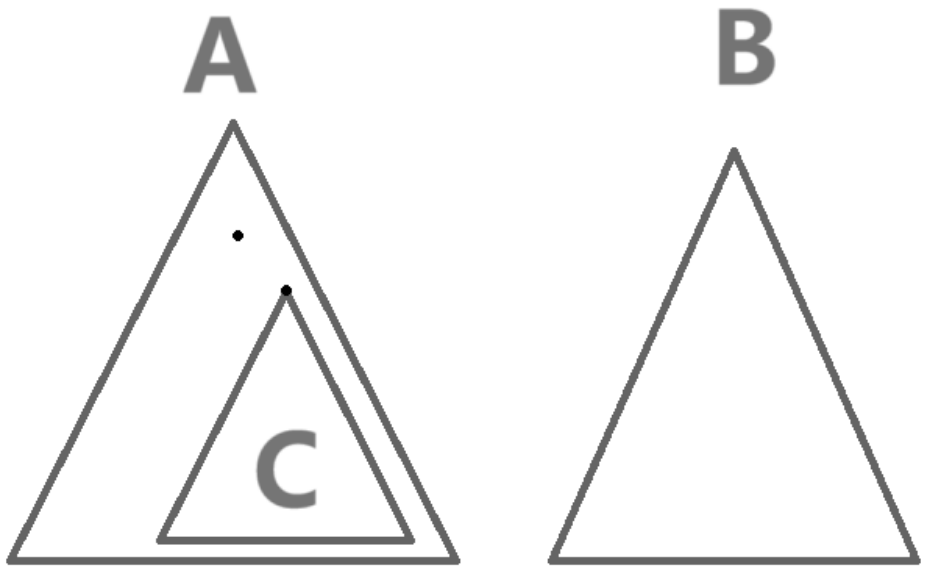

图: 相同权值 Union 的例子

不妨设 A 树随机优先级高于 B 树，此时 Split 的过程就是将 B 树和 A 的右子树 C 进行合并，通过归纳，可以发现合并出来的 D 树其右链长度为 A,B 右链长度之和，此时树高并不满足高度期望为 $O(\log n)$。

线段树合并

线段树同样维护有序信息集合，相对平衡树其优势在于结构简洁明确，实现简单，可以支持大部分平衡树可处理的操作。

线段树合并

线段树同样维护有序信息集合，相对平衡树其优势在于结构简洁明确，实现简单，可以支持大部分平衡树可处理的操作。
线段树的结构是为人熟知的，此处不再赘述。

线段树合并

线段树同样维护有序信息集合，相对平衡树其优势在于结构简洁明确，实现简单，可以支持大部分平衡树可处理的操作。
线段树的结构是为人熟知的，此处不再赘述。
线段树合并用于维护线段树间信息集合合并，由于相比平衡树其常数较小，时间复杂度易于分析，有较优拓展性，所以我们常用线段树合并。

线段树合并

线段树同样维护有序信息集合，相对平衡树其优势在于结构简洁明确，实现简单，可以支持大部分平衡树可处理的操作。
线段树的结构是为人熟知的，此处不再赘述。
线段树合并用于维护线段树间信息集合合并，由于相比平衡树其常数较小，时间复杂度易于分析，有较优拓展性，所以我们常用线段树合并。
复杂度分析：对于两颗合并的线段树，结构类似，两颗线段树合并的时候当一颗线段树节点为空时可以直接继承另一颗线段树的信息，每次合并后两个节点会变成一个节点，以节点个数作为势能，注意到设插入的操作次数为 $O(n)$ 次，总插入节点个数为 $O(n\log n)$，总线段树合并的过程复杂度也为 $O(n\log n)$。

带新建节点标记的线段树合并

考虑以下问题：给定一棵树，对于树上节点有权值 A，需要维护支持区间 $[1, A], [A+1, n], [1, n]$ 区间加，信息合并为下标对位乘法，对于所有点需要求得整个子树的信息。

带新建节点标记的线段树合并

考虑以下问题：给定一棵树，对于树上节点有权值 A，需要维护支持区间 $[1, A], [A+1, n], [1, n]$ 区间加，信息合并为下标对位乘法，对于所有点需要求得整个子树的信息。

对位乘积可以描述为线段树合并的过程，若一边为一个节点，则可以对另一侧打区间乘法标记，区间加法仅建立遍历到的节点，下放加法标记时需要新建线段树节点。

带新建节点标记的线段树合并

考虑以下问题：给定一棵树，对于树上节点有权值 A，需要维护支持区间 $[1, A], [A+1, n], [1, n]$ 区间加，信息合并为下标对位乘法，对于所有点需要求得整个子树的信息。

对位乘积可以描述为线段树合并的过程，若一边为一个节点，则可以对另一侧打区间乘法标记，区间加法仅建立遍历到的节点，下放加法标记时需要新建线段树节点。

考虑其合并复杂度：对于任意线段树节点，要么同时有左右儿子，要么同时没有左右儿子，在所有标记下放的时候均不会造成节点的新建，故总复杂度仍 $=$ 节点个数，即 $O(n \log n)$。

复杂度分析

这个例子很好的告诉我们当线段树标记下放需要新建节点时复杂度仍尝试分析节点个数变化。

复杂度分析

这个例子很好的告诉我们当线段树标记下放需要新建节点时复杂度仍尝试分析节点个数变化。
考虑对于不满足此初始条件时，此类线段树合并时间复杂度，相似的，我们假定在合并时，存在一边只有单个节点（没有左右儿子）时可以快速将影响应用到另一棵树上。

复杂度分析

若直接考虑合并时分析其节点的新建情况，有例子形如，合并两条一致的黑色部分链，需要做标记下放而新建虚线节点，此时我们递归深度为 $\log n$ 但节点个数仅减少 1，似乎此时导致总复杂度为 $O(n \log^2 n)$。

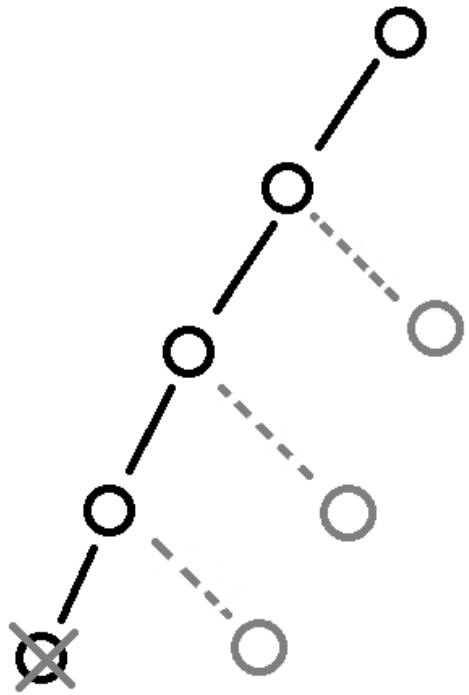

图: 合并的例子

复杂度分析

其实不然，注意到当前建立的叶子总是不存在左右儿子的，即会新建节点的线段树节点总是在原结构上只有一个儿子的节点，我们将其补齐，相似的，初始节点为 $O(n \log n)$ 个且满足例题性质。

复杂度分析

其实不然，注意到当前建立的叶子总是不存在左右儿子的，即会新建节点的线段树节点总是在原结构上只有一个儿子的节点，我们将其补齐，相似的，初始节点为 $O(n \log n)$ 个且满足例题性质。
有在此类线段树合并新建节点过程中我们仅会补齐上述节点，故总复杂度仍为 $O(n \log n)$。

线段树合并空间优化

考虑线段树合并的空间消耗，通常情况下我们空间复杂度的上界为 $O(n \log n)$。

线段树合并空间优化

考虑线段树合并的空间消耗，通常情况下我们空间复杂度的上界为 $O(n \log n)$。
我们尝试改变线段树合并的顺序，并在删除节点时回收利用空间，得到紧的空间上界。

线段树合并空间优化

具体地，考虑当前问题形如一棵树，每个节点有若干个单点修改需要对于每个子树都求出子树内所有操作在线段树合并后的形式。

线段树合并空间优化

具体地，考虑当前问题形如一棵树，每个节点有若干个单点修改需要对于每个子树都求出子树内所有操作在线段树合并后的形式。
若当前形如区间修改 $[l,r]$，我们考察其新建节点的情况，其新建的节点不会超过单点加入 l 并补全路径上右儿子与单点加入 r 并补全路径上左儿子，于是区间修改可以大致视为两次单点修改。

线段树合并空间优化

具体地，考虑当前问题形如一棵树，每个节点有若干个单点修改需要对于每个子树都求出子树内所有操作在线段树合并后的形式。
若当前形如区间修改 $[l,r]$，我们考察其新建节点的情况，其新建的节点不会超过单点加入 l 并补全路径上右儿子与单点加入 r 并补全路径上左儿子，于是区间修改可以大致视为两次单点修改。
我们尝试按照每个点的操作次数做重链剖分，当前处理方式为优先遍历重儿子，遍历完后递归轻儿子子树，与重儿子子树合并。

线段树合并空间优化

具体地，考虑当前问题形如一棵树，每个节点有若干个单点修改需要对于每个子树都求出子树内所有操作在线段树合并后的形式。
若当前形如区间修改 $[l,r]$，我们考察其新建节点的情况，其新建的节点不会超过单点加入 l 并补全路径上右儿子与单点加入 r 并补全路径上左儿子，于是区间修改可以大致视为两次单点修改。
我们尝试按照每个点的操作次数做重链剖分，当前处理方式为优先遍历重儿子，遍历完后递归轻儿子子树，与重儿子子树合并。
注意在空间回收意义下即消耗空间复杂度为任意时刻最多存在的节点数量。

线段树合并空间优化

可以发现任意时刻形如至多维护 $O(\log n)$ 棵线段树，且第 i 棵大小不会超过 $\frac{n}{2^i}$，我们尝试精细分析其占用的空间。

线段树合并空间优化

可以发现任意时刻形如至多维护 $O(\log n)$ 棵线段树，且第 i 棵大小不会超过 $\frac{n}{2^i}$，我们尝试精细分析其占用的空间。

考虑线段树一共有 $\log n$ 层，对于加入 x 个节点的线段树，其第 i 层新建的节点数量为 $\min(x, 2^i)$，此时视为上界维护了 $\log n$ 棵线段树，第 $i \in [1, \log n]$ 棵有 2^i 个节点。

线段树合并空间优化

可以发现任意时刻形如至多维护 $O(\log n)$ 棵线段树，且第 i 棵大小不会超过 $\frac{n}{2^i}$，我们尝试精细分析其占用的空间。
考虑线段树一共有 $\log n$ 层，对于加入 x 个节点的线段树，其第 i 层新建的节点数量为 $\min(x, 2^i)$，此时视为上界维护了 $\log n$ 棵线段树，第 $i \in [1, \log n]$ 棵有 2^i 个节点。
当前形如总节点数的上界为

$$\sum_{i=0}^{\log n} 2^{i+1} + (\log n - i)2^i \le O(n) + \sum_{i=0}^{\log n} \sum_{j=i+1}^{\log n} 2^i =$$

$$O(n) + \sum_{j=0}^{\log n} \sum_{i \le j} 2^i \le O(n) + \sum_{j=0}^{\log n} 2^{j+1} = O(n)$$

即总消耗空间复杂度为 $O(n)$。

致谢

感谢中国计算机学会提供学习交流的平台。
感谢父母的养育之恩。
感谢巴蜀中学姜宗沛老师、黄新军老师的指导和关心。
感谢为与我讨论，审稿，提出建议的所有同学们。
感谢各位聆听。

参考文献 I

- 2015 年集训队论文, 任之洲,《浅谈启发式思想在信息学竞赛中的应用》
- 2021 年集训队论文, 周镇东,《浅谈一类树上路径相关问题》
- OI Wiki,dsu on tree,https://oi-wiki.org/graph/dsu-on-tree/
- xiaoziyao, 浅谈一类信息的暴力重构手法 https://www.cnblogs.com/xiaoziyao/p/17413029.html
- 2013 年集训队论文, 许昊然《浅谈数据结构题的几个非经典解法》
- Wikipedia,leftist tree,https://oi-wiki.org/ds/leftist-tree/
- Wikipedia,Finger search,https://en.wikipedia.org/wiki/Finger_search

参考文献 II


Wikipedia,Finger search tree,https://en.wikipedia.org/wiki/Finger_search_tree


D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. Journal of the ACM


R. Cole. On the dynamic finger conjecture for splay trees. part II: The proof. SIAM Journal of Computing, 30(1):44–85, 2000.


R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. part I: Splay sorting log n-block sequences. SIAM Journal of Computing, 30(1):1–43, 2000.


R. Seidel and C. R. Aragon. Randomized search trees. Algorithmica, 16(4/5):464–497,1996.

参考文献 III

- Gerth Stølting Brodal. Finger Search Trees. University of Aarhus, 2005
- Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In 10th Annual ACM Symposium on Parallel Algorithms and Architectures, Puerto Vallarta, Mexico, June -July 1998. ACM.
- 2018 年集训队论文, 董炜隽,《浅谈 Splay 与 Treap 的性质及其应用》
- 约瑟夫用脑玩,finger-search+ 启发式合并,https://www.luogu.com.cn/blog/ICANTAKIOI/finger-search-qi-fa-shi-ge-bing
- dpair, 记录一个线段树合并的 trick,https://dpair.gitee.io/articles/sgtmrgt/