常用数据结构与程序设计例解

PKU EECS zld3794955

2017 年 7 月 16 日

前言

前言

（首先声明一下，课件中的图片大部分来源于网络，少部分是我用画图做的。）

前言

（首先声明一下，课件中的图片大部分来源于网络，少部分是我用画图做的。）

常用数据结构想必大家都能意会到是些什么。

前言

（首先声明一下，课件中的图片大部分来源于网络，少部分是我用画图做的。）

常用数据结构想必大家都能意会到是些什么。

但是程序设计例解这个部分我也不太清楚是什么，想来就是凭感觉

前言

（首先声明一下，课件中的图片大部分来源于网络，少部分是我用画图做的。）

常用数据结构想必大家都能意会到是些什么。

但是程序设计例解这个部分我也不太清楚是什么，想来就是凭感觉，比如我的感觉就是一片空白。

前言

（首先声明一下，课件中的图片大部分来源于网络，少部分是我用画图做的。）

常用数据结构想必大家都能意会到是些什么。

但是程序设计例解这个部分我也不太清楚是什么，想来就是凭感觉，比如我的感觉就是一片空白。

今天需要大家掌握了解的知识较多，故配备的例题会少一些，大家将重点放在对数据结构工作原理的理解上即可。

前言

（首先声明一下，课件中的图片大部分来源于网络，少部分是我用画图做的。）

常用数据结构想必大家都能意会到是些什么。

但是程序设计例解这个部分我也不太清楚是什么，想来就是凭感觉，比如我的感觉就是一片空白。

今天需要大家掌握了解的知识较多，故配备的例题会少一些，大家将重点放在对数据结构工作原理的理解上即可。

经过大学一年的艰(*xían*)难(*yú*)生活后，讲课人非常深刻地体会到了很多东西。

前言

（首先声明一下，课件中的图片大部分来源于网络，少部分是我用画图做的。）

常用数据结构想必大家都能意会到是些什么。

但是程序设计例解这个部分我也不太清楚是什么，想来就是凭感觉，比如我的感觉就是一片空白。

今天需要大家掌握了解的知识较多，故配备的例题会少一些，大家将重点放在对数据结构工作原理的理解上即可。

经过大学一年的艰(*xían*)难(*yú*)生活后，讲课人非常深刻地体会到了很多东西。

~~比如我就是个傻逼。~~

前言

（首先声明一下，课件中的图片大部分来源于网络，少部分是我用画图做的。）

常用数据结构想必大家都能意会到是些什么。

但是程序设计例解这个部分我也不太清楚是什么，想来就是凭感觉，比如我的感觉就是一片空白。

今天需要大家掌握了解的知识较多，故配备的例题会少一些，大家将重点放在对数据结构工作原理的理解上即可。

经过大学一年的艰(*xían*)难(*yú*)生活后，讲课人非常深刻地体会到了很多东西。

~~比如我就是个傻逼。~~

所以，若课件中有什么不明白的地方欢迎提问，有错误的话恳请指正，谢谢!

今天的内容I

今天的内容I

先是一些比较基础的东西：

今天的内容I

先是一些比较基础的东西：

1. 线性表

今天的内容I

先是一些比较基础的东西：

1. 线性表
2. 栈与队列

今天的内容I

先是一些比较基础的东西：

1. 线性表
2. 栈与队列
3. 树与Trie

今天的内容I

先是一些比较基础的东西：

1. 线性表
2. 栈与队列
3. 树与Trie
4. 优先队列与堆

今天的内容I

先是一些比较基础的东西：

1. 线性表
2. 栈与队列
3. 树与Trie
4. 优先队列与堆
5. 图与并查集

今天的内容I

先是一些比较基础的东西：

1. 线性表
2. 栈与队列
3. 树与Trie
4. 优先队列与堆
5. 图与并查集
6. Hash

今天的内容II

今天的内容II

接下来是高级一些的东西：

今天的内容II

接下来是高级一些的东西：

1. ST表与LCA问题

今天的内容II

接下来是高级一些的东西：

1. ST表与LCA问题
2. 树状数组

今天的内容II

接下来是高级一些的东西：

1. ST表与LCA问题
2. 树状数组
3. 线段树

今天的内容II

接下来是高级一些的东西：

1. ST表与LCA问题
2. 树状数组
3. 线段树

本来想讲讲C++STL，后来觉得一是现在应该还有P选手，讲这个不太友好，二是这些东西大家随便百度下就会了，最后就是如果讲这个会导致下午出的题代码量较大，因此我就不讲了。

今天的内容II

接下来是高级一些的东西：

1. ST表与LCA问题
2. 树状数组
3. 线段树

本来想讲讲C++STL，后来觉得一是现在应该还有P选手，讲这个不太友好，二是这些东西大家随便百度下就会了，最后就是如果讲这个会导致下午出的题代码量较大，因此我就不讲了。

下午三题中T1所需知识在内容I内，T3需要用到内容II中的东西，T2不定（因为写课件的时候我还没选好题呢）。

今天的内容II

接下来是高级一些的东西：

1. ST表与LCA问题
2. 树状数组
3. 线段树

本来想讲讲C++STL，后来觉得一是现在应该还有P选手，讲这个不太友好，二是这些东西大家随便百度下就会了，最后就是如果讲这个会导致下午出的题代码量较大，因此我就不讲了。

下午三题中T1所需知识在内容I内，T3需要用到内容II中的东西，T2不定（因为写课件的时候我还没选好题呢）。

至于题目难度的话，我会在合理的范围内尽可能压低的啦，出题人可是很善良哒~

线性表

线性表

线性表是一种逻辑结构，在线性表中，数据按照一定的顺序串在一起。

线性表

线性表是一种逻辑结构，在线性表中，数据按照一定的顺序串在一起。

线性表在存储实现上一般有两种方式——顺序表和链表。

线性表

线性表是一种逻辑结构，在线性表中，数据按照一定的顺序串在一起。

线性表在存储实现上一般有两种方式——顺序表和链表。

顺序表即为我们平常所说的数组，在内存中相邻元素存放的位置一定是连续的。

线性表

线性表是一种逻辑结构，在线性表中，数据按照一定的顺序串在一起。

线性表在存储实现上一般有两种方式——顺序表和链表。

顺序表即为我们平常所说的数组，在内存中相邻元素存放的位置一定是连续的。

因此，我们只要知道第一个元素的位置，就可以在常数的时间内找到任意一个标号的元素所在的位置，这对应着数组取下标的操作。

线性表

线性表是一种逻辑结构，在线性表中，数据按照一定的顺序串在一起。

线性表在存储实现上一般有两种方式——顺序表和链表。

顺序表即为我们平常所说的数组，在内存中相邻元素存放的位置一定是连续的。

因此，我们只要知道第一个元素的位置，就可以在常数的时间内找到任意一个标号的元素所在的位置，这对应着数组取下标的操作。

在顺序表开头或者结尾增减元素都比较方便，但是在顺序表内部就比较困难（不会就自己想想），一次操作平均时间复杂度是$O(n)$的。

线性表

线性表是一种逻辑结构，在线性表中，数据按照一定的顺序串在一起。

线性表在存储实现上一般有两种方式——顺序表和链表。

顺序表即为我们平常所说的数组，在内存中相邻元素存放的位置一定是连续的。

因此，我们只要知道第一个元素的位置，就可以在常数的时间内找到任意一个标号的元素所在的位置，这对应着数组取下标的操作。

在顺序表开头或者结尾增减元素都比较方便，但是在顺序表内部就比较困难（不会就自己想想），一次操作平均时间复杂度是$O(n)$的。

顺序表的二维化也十分简单——用二维数组即可。

链表

链表

链表跟顺序表不同的是，它通过在每个数据上标记下一个数据的位置来保持顺序。

链表

链表跟顺序表不同的是，它通过在每个数据上标记下一个数据的位置来保持顺序。

对于每个数据节点p，记p的数据为$p \to data$，p的下一个数据节点为$p \to next$（不存在则记为NULL），这样，我们只要知道链表的开头（记为$head$），就可以沿着$head$往下走到NULL为止，按顺序遍历到所有的数据。

链表

链表跟顺序表不同的是，它通过在每个数据上标记下一个数据的位置来保持顺序。

对于每个数据节点p，记p的数据为$p \to data$，p的下一个数据节点为$p \to next$（不存在则记为NULL），这样，我们只要知道链表的开头（记为$head$），就可以沿着$head$往下走到NULL为止，按顺序遍历到所有的数据。

显然，若要在链表中访问指定位置的元素，那么我们必须从开头开始遍历，直到找到对应的元素，单次平均时间复杂度$O(n)$。

链表

链表跟顺序表不同的是，它通过在每个数据上标记下一个数据的位置来保持顺序。

对于每个数据节点p，记p的数据为$p \to data$，p的下一个数据节点为$p \to next$（不存在则记为NULL），这样，我们只要知道链表的开头（记为$head$），就可以沿着$head$往下走到NULL为止，按顺序遍历到所有的数据。

显然，若要在链表中访问指定位置的元素，那么我们必须从开头开始遍历，直到找到对应的元素，单次平均时间复杂度$O(n)$。

若要在指定节点p后插入节点q，那么我们改变一下连接顺序——令$q \to next = p \to next$，再令$p \to next = q$即可。

链表

链表跟顺序表不同的是，它通过在每个数据上标记下一个数据的位置来保持顺序。

对于每个数据节点p，记p的数据为$p \to data$，p的下一个数据节点为$p \to next$（不存在则记为NULL），这样，我们只要知道链表的开头（记为$head$），就可以沿着$head$往下走到NULL为止，按顺序遍历到所有的数据。

显然，若要在链表中访问指定位置的元素，那么我们必须从开头开始遍历，直到找到对应的元素，单次平均时间复杂度$O(n)$。

若要在指定节点p后插入节点q，那么我们改变一下连接顺序——令$q \to next = p \to next$，再令$p \to next = q$即可。

而对于一个节点p，如果我们要删除$p \to next$，那么只要令$p \to next = p \to next \to next$即可，这也是常数时间的。

Some variants

Some variants

为了使删除一个节点方便一些，我们可以对每个节点p再记录其上一个数据节点$p \to last$，这样我们就得到了双向链表。

Some variants

为了使删除一个节点方便一些，我们可以对每个节点p再记录其上一个数据节点$p \to last$，这样我们就得到了双向链表。

插入时，除了$next$之外，对$last$也进行一遍类似的操作即可。

Some variants

为了使删除一个节点方便一些，我们可以对每个节点p再记录其上一个数据节点$p \to last$，这样我们就得到了双向链表。

插入时，除了$next$之外，对$last$也进行一遍类似的操作即可。

删除节点p时，只要将其上一个节点与下一个节点（如果存在）跨过p直接连起来即可，即进行操作$p \to last \to next = p \to next$

和$p \to next \to last = p \to last$。

Some variants

为了使删除一个节点方便一些，我们可以对每个节点p再记录其上一个数据节点$p \to last$，这样我们就得到了双向链表。

插入时，除了$next$之外，对$last$也进行一遍类似的操作即可。

删除节点p时，只要将其上一个节点与下一个节点（如果存在）跨过p直接连起来即可，即进行操作$p \to last \to next = p \to next$

和$p \to next \to last = p \to last$。

若将链表尾的下一个定义为链表头，那么我们就得到了一个循环链表，操作类似，注意链表中只有一个元素的边界情况即可。

Some variants

为了使删除一个节点方便一些，我们可以对每个节点p再记录其上一个数据节点$p \to last$，这样我们就得到了双向链表。

插入时，除了$next$之外，对$last$也进行一遍类似的操作即可。

删除节点p时，只要将其上一个节点与下一个节点（如果存在）跨过p直接连起来即可，即进行操作$p \to last \to next = p \to next$

和$p \to next \to last = p \to last$。

若将链表尾的下一个定义为链表头，那么我们就得到了一个循环链表，操作类似，注意链表中只有一个元素的边界情况即可。

而二维链表既可以用链表套链表实现，也可以通过记录每个节点在两维上的下一个节点（$nextx$与$nexty$）分别是什么来实现，这与普通链表的实现也类似。

栈与队列

栈与队列

栈与队列都是特殊的线性表，其中，栈是一种先进后出(FILO)的结构，而队列是先进先出(FIFO)的结构。

栈与队列

栈与队列都是特殊的线性表，其中，栈是一种先进后出(FILO)的结构，而队列是先进先出(FIFO)的结构。

一个栈支持以下两种操作：

栈与队列

栈与队列都是特殊的线性表，其中，栈是一种先进后出(FILO)的结构，而队列是先进先出(FIFO)的结构。

一个栈支持以下两种操作：

1、将一个元素放在当前栈的顶端。

栈与队列

栈与队列都是特殊的线性表，其中，栈是一种先进后出(FILO)的结构，而队列是先进先出(FIFO)的结构。

一个栈支持以下两种操作：

1、将一个元素放在当前栈的顶端。

2、获得/弹出当前栈顶端的元素，或判断该栈为空。

栈与队列

栈与队列都是特殊的线性表，其中，栈是一种先进后出(FILO)的结构，而队列是先进先出(FIFO)的结构。

一个栈支持以下两种操作：

1、将一个元素放在当前栈的顶端。

2、获得/弹出当前栈顶端的元素，或判断该栈为空。

一个队列则支持以下两种操作：

栈与队列

栈与队列都是特殊的线性表，其中，栈是一种先进后出(FILO)的结构，而队列是先进先出(FIFO)的结构。

一个栈支持以下两种操作：

1、将一个元素放在当前栈的顶端。

2、获得/弹出当前栈顶端的元素，或判断该栈为空。

一个队列则支持以下两种操作：

1、将一个元素放在当前队列的尾部。

栈与队列

栈与队列都是特殊的线性表，其中，栈是一种先进后出(FILO)的结构，而队列是先进先出(FIFO)的结构。

一个栈支持以下两种操作：

1、将一个元素放在当前栈的顶端。

2、获得/弹出当前栈顶端的元素，或判断该栈为空。

一个队列则支持以下两种操作：

1、将一个元素放在当前队列的尾部。

2、获得/弹出当前队列头部的元素，或判断队列为空。

栈与队列

栈与队列都是特殊的线性表，其中，栈是一种先进后出(FILO)的结构，而队列是先进先出(FIFO)的结构。

一个栈支持以下两种操作：

1、将一个元素放在当前栈的顶端。

2、获得/弹出当前栈顶端的元素，或判断该栈为空。

一个队列则支持以下两种操作：

1、将一个元素放在当前队列的尾部。

2、获得/弹出当前队列头部的元素，或判断队列为空。

可以看到，栈和队列本质上都是线性表，故它们都可以用数组或者链表实现。

扩展——双端队列

扩展——双端队列

将队列的概念推广一下，我们可以得到双端队列，其可在队列的头部和尾部自由地插入或删除元素。

扩展——双端队列

将队列的概念推广一下，我们可以得到双端队列，其可在队列的头部和尾部自由地插入或删除元素。

不难看出，栈和队列的操作都是双端队列操作的子集。

扩展——双端队列

将队列的概念推广一下，我们可以得到双端队列，其可在队列的头部和尾部自由地插入或删除元素。

不难看出，栈和队列的操作都是双端队列操作的子集。

双端队列的一个用途是在动态规划(DP)中维护最优决策，此时也称为单调队列，等过几天讲DP的时候你们应该就会知道了。

树

树

还记得在原始的链表中，我们为每个节点存储了其相邻的后继节点*next*么？

树

还记得在原始的链表中，我们为每个节点存储了其相邻的后继节点*next*么？

如果不限制一个节点的*next*节点个数，并且稍加限制，那么我们就得到了一棵数据结构意义上的树。

树

还记得在原始的链表中，我们为每个节点存储了其相邻的后继节点*next*么？

如果不限制一个节点的*next*节点个数，并且稍加限制，那么我们就得到了一棵数据结构意义上的树。

树的一种比较简单的定义如下：

树

还记得在原始的链表中，我们为每个节点存储了其相邻的后继节点*next*么？

如果不限制一个节点的*next*节点个数，并且稍加限制，那么我们就得到了一棵数据结构意义上的树。

树的一种比较简单的定义如下：

1、单个节点是一棵树，定义其树根为自己。

树

还记得在原始的链表中，我们为每个节点存储了其相邻的后继节点*next*么？

如果不限制一个节点的*next*节点个数，并且稍加限制，那么我们就得到了一棵数据结构意义上的树。

树的一种比较简单的定义如下：

1、单个节点是一棵树，定义其树根为自己。

2、若有若干棵两两之间没有公共节点的树，那么我们可以新建一个节点p，令p的*next*节点集为这些树的根节点，

树

还记得在原始的链表中，我们为每个节点存储了其相邻的后继节点*next*么？

如果不限制一个节点的*next*节点个数，并且稍加限制，那么我们就得到了一棵数据结构意义上的树。

树的一种比较简单的定义如下：

1、单个节点是一棵树，定义其树根为自己。

2、若有若干棵两两之间没有公共节点的树，那么我们可以新建一个节点p，令p的*next*节点集为这些树的根节点，则我们就得到了一棵新树，定义这棵树的根节点为p。

树

还记得在原始的链表中，我们为每个节点存储了其相邻的后继节点*next*么？

如果不限制一个节点的*next*节点个数，并且稍加限制，那么我们就得到了一棵数据结构意义上的树。

树的一种比较简单的定义如下：

1、单个节点是一棵树，定义其树根为自己。

2、若有若干棵两两之间没有公共节点的树，那么我们可以新建一个节点p，令p的*next*节点集为这些树的根节点，则我们就得到了一棵新树，定义这棵树的根节点为p。

特别地，一个链表也是一棵树。

一些其它的概念

一些其它的概念

若节点q在节点p的$next$集合里，那么称节点q为节点p的一个儿子，并且称节点p为节点q的父亲。

一些其它的概念

若节点q在节点p的$next$集合里，那么称节点q为节点p的一个儿子，并且称节点p为节点q的父亲。

若一个节点没有儿子，那么称其为叶节点。

一些其它的概念

若节点q在节点p的$next$集合里，那么称节点q为节点p的一个儿子，并且称节点p为节点q的父亲。

若一个节点没有儿子，那么称其为叶节点。

定义p是q的祖先，当且仅当$p = q$或者p是q的父节点的祖先。

一些其它的概念

若节点q在节点p的$next$集合里，那么称节点q为节点p的一个儿子，并且称节点p为节点q的父亲。

若一个节点没有儿子，那么称其为叶节点。

定义p是q的祖先，当且仅当$p = q$或者p是q的父节点的祖先。

定义q为p的后代，当且仅当p为q的祖先。

一些其它的概念

若节点q在节点p的$next$集合里，那么称节点q为节点p的一个儿子，并且称节点p为节点q的父亲。

若一个节点没有儿子，那么称其为叶节点。

定义p是q的祖先，当且仅当$p = q$或者p是q的父节点的祖先。

定义q为p的后代，当且仅当p为q的祖先。

定义一个节点的深度为其祖先节点的个数（不含自己）。

一些其它的概念

若节点q在节点p的$next$集合里，那么称节点q为节点p的一个儿子，并且称节点p为节点q的父亲。

若一个节点没有儿子，那么称其为叶节点。

定义p是q的祖先，当且仅当$p = q$或者p是q的父节点的祖先。

定义q为p的后代，当且仅当p为q的祖先。

定义一个节点的深度为其祖先节点的个数（不含自己）。

定义以树上任意一个节点p为根的子树为树上所有为p的后代的节点所构成的树。

一些其它的概念

若节点q在节点p的$next$集合里，那么称节点q为节点p的一个儿子，并且称节点p为节点q的父亲。

若一个节点没有儿子，那么称其为叶节点。

定义p是q的祖先，当且仅当$p = q$或者p是q的父节点的祖先。

定义q为p的后代，当且仅当p为q的祖先。

定义一个节点的深度为其祖先节点的个数（不含自己）。

定义以树上任意一个节点p为根的子树为树上所有为p的后代的节点所构成的树。

定义森林为若干棵树的集合。

对树的理解

对树的理解

树结构具有递归性，子结构清楚，因此对于树上的一些信息（比如以各个节点为根的子树的节点个数），我们可以利用这个结构来统计，具体的方法在明天会讲到。

对树的理解

树结构具有递归性，子结构清楚，因此对于树上的一些信息（比如以各个节点为根的子树的节点个数），我们可以利用这个结构来统计，具体的方法在明天会讲到。

我们还可以用这个结构来定义树的高度：定义单个节点构成的树的树高为0或1，一棵树的树高为以其某个子节点为根的所有子树中树高的最大值+1.

对树的理解

树结构具有递归性，子结构清楚，因此对于树上的一些信息（比如以各个节点为根的子树的节点个数），我们可以利用这个结构来统计，具体的方法在明天会讲到。

我们还可以用这个结构来定义树的高度：定义单个节点构成的树的树高为0或1，一棵树的树高为以其某个子节点为根的所有子树中树高的最大值+1.

如果将树看作链表的扩展，那么我们可以将根节点看作头，从根开始，每次随意选择一个子节点往下走到叶节点为止，便能得到一条链，这个在后面理解Trie的时候帮助挺大。

对树的理解

树结构具有递归性，子结构清楚，因此对于树上的一些信息（比如以各个节点为根的子树的节点个数），我们可以利用这个结构来统计，具体的方法在明天会讲到。

我们还可以用这个结构来定义树的高度：定义单个节点构成的树的树高为0或1，一棵树的树高为以其某个子节点为根的所有子树中树高的最大值+1.

如果将树看作链表的扩展，那么我们可以将根节点看作头，从根开始，每次随意选择一个子节点往下走到叶节点为止，便能得到一条链，这个在后面理解Trie的时候帮助挺大。

利用树的父亲节点的定义，我们也可以感受到，树上除了根节点之外，每个节点都有唯一的一个父节点，从任意一个节点开始沿父节点往上走，最终都会走到根节点，这个可以用于后面理解并查集的实现。

一些特殊的树

一些特殊的树

若一棵树除了叶节点外，每个节点都有k个子节点，那么称这棵树为k叉树，特别地，若$k = 2$，则我们称其为二叉树。

一些特殊的树

若一棵树除了叶节点外，每个节点都有k个子节点，那么称这棵树为k叉树，特别地，若$k = 2$，则我们称其为二叉树。

若计较二叉树中子节点的顺序，那么我们可以定义左儿子与右儿子的概念及二叉树的三种遍历方式：

一些特殊的树

若一棵树除了叶节点外，每个节点都有k个子节点，那么称这棵树为k叉树，特别地，若$k = 2$，则我们称其为二叉树。

若计较二叉树中子节点的顺序，那么我们可以定义左儿子与右儿子的概念及二叉树的三种遍历方式：

1、先序遍历：根节点+先序左儿子+先序右儿子。

一些特殊的树

若一棵树除了叶节点外，每个节点都有k个子节点，那么称这棵树为k叉树，特别地，若$k = 2$，则我们称其为二叉树。

若计较二叉树中子节点的顺序，那么我们可以定义左儿子与右儿子的概念及二叉树的三种遍历方式：

1、先序遍历：根节点+先序左儿子+先序右儿子。

2、中序遍历：中序左儿子+根节点+中序右儿子。

一些特殊的树

若一棵树除了叶节点外，每个节点都有k个子节点，那么称这棵树为k叉树，特别地，若$k = 2$，则我们称其为二叉树。

若计较二叉树中子节点的顺序，那么我们可以定义左儿子与右儿子的概念及二叉树的三种遍历方式：

1、先序遍历：根节点+先序左儿子+先序右儿子。

2、中序遍历：中序左儿子+根节点+中序右儿子。

3、后序遍历：后序左儿子+后序右儿子+根节点。

一些特殊的树

若一棵树除了叶节点外，每个节点都有k个子节点，那么称这棵树为k叉树，特别地，若$k = 2$，则我们称其为二叉树。

若计较二叉树中子节点的顺序，那么我们可以定义左儿子与右儿子的概念及二叉树的三种遍历方式：

1、先序遍历：根节点+先序左儿子+先序右儿子。

2、中序遍历：中序左儿子+根节点+中序右儿子。

3、后序遍历：后序左儿子+后序右儿子+根节点。

同时我们通过将二叉树按层次画在图上，还能定义完全二叉树与满二叉树的概念~~（懒得写在课件上了）~~。

一般树的遍历

一般树的遍历

由于树上的节点有分支，因此其也有多种多样的遍历所有节点的方式。

一般树的遍历

由于树上的节点有分支，因此其也有多种多样的遍历所有节点的方式。

最常见的就是两种：深度优先遍历(DFS)和广度优先遍历(BFS)。

一般树的遍历

由于树上的节点有分支，因此其也有多种多样的遍历所有节点的方式。

最常见的就是两种：深度优先遍历(DFS)和广度优先遍历(BFS)。

我们明天会讲DFS回溯法，而BFS过几天在n+e大佬的课上会提到，这里我就不深入展开了。

一般树的遍历

由于树上的节点有分支，因此其也有多种多样的遍历所有节点的方式。

最常见的就是两种：深度优先遍历(DFS)和广度优先遍历(BFS)。

我们明天会讲DFS回溯法，而BFS过几天在n+e大佬的课上会提到，这里我就不深入展开了。

下面我们来介绍一下Trie。

Trie

Trie

中文名字典树/前缀树，是用于存储和检索字符串的一种树结构，其基本思想是将字符串按照前缀分类构成树。

Trie

中文名字典树/前缀树，是用于存储和检索字符串的一种树结构，其基本思想是将字符串按照前缀分类构成树。

对每个节点，我们维护一个与字符集一一对应的子节点集，其中字符ch对应的子节点即为从当前状态读入字符ch后走到的子节点。

Trie

中文名字典树/前缀树，是用于存储和检索字符串的一种树结构，其基本思想是将字符串按照前缀分类构成树。

对每个节点，我们维护一个与字符集一一对应的子节点集，其中字符ch对应的子节点即为从当前状态读入字符ch后走到的子节点。

插入一个字符串时，我们从根节点开始，一个一个读入字符并尝试走到下一子节点，如果不存在就新创建一个子节点。

Trie

中文名字典树/前缀树，是用于存储和检索字符串的一种树结构，其基本思想是将字符串按照前缀分类构成树。

对每个节点，我们维护一个与字符集一一对应的子节点集，其中字符ch对应的子节点即为从当前状态读入字符ch后走到的子节点。

插入一个字符串时，我们从根节点开始，一个一个读入字符并尝试走到下一子节点，如果不存在就新创建一个子节点。

在处理完当前字符串后，需要在最终的节点上做一个标记，表示从根节点走到这里的路径上形成的字符串是存在的。

Trie

中文名字典树/前缀树，是用于存储和检索字符串的一种树结构，其基本思想是将字符串按照前缀分类构成树。

对每个节点，我们维护一个与字符集一一对应的子节点集，其中字符ch对应的子节点即为从当前状态读入字符ch后走到的子节点。

插入一个字符串时，我们从根节点开始，一个一个读入字符并尝试走到下一子节点，如果不存在就新创建一个子节点。

在处理完当前字符串后，需要在最终的节点上做一个标记，表示从根节点走到这里的路径上形成的字符串是存在的。

查找字符串过程类似，大家可以自己想想，下面来道题加深一下大家的理解~

一道例题

一道例题

给定一堆的非空字符串（可重），你的任务就是寻找这些字符串中最受欢迎的前缀和后缀。

一道例题

给定一堆的非空字符串（可重），你的任务就是寻找这些字符串中最受欢迎的前缀和后缀。

其中，一个前缀的受欢迎程度为该前缀长度×该前缀在字符串中出现的次数，后缀的定义也类似，

一道例题

给定一堆的非空字符串（可重），你的任务就是寻找这些字符串中最受欢迎的前缀和后缀。

其中，一个前缀的受欢迎程度为该前缀长度×该前缀在字符串中出现的次数，后缀的定义也类似，

字符串总长≤ 200000。

The solution

The solution

在Trie中，如果从根节点开始读入一个字符串s后走到的节点p非空，那么显然s是Trie中至少一个串的前缀，并且以p为根的子树内存储的字符串个数，就是s作为前缀出现的次数，记这个次数为$sum[p]$，则我们只需对每个节点p求出$sum[p]$即可。

The solution

在Trie中，如果从根节点开始读入一个字符串s后走到的节点p非空，那么显然s是Trie中至少一个串的前缀，并且以p为根的子树内存储的字符串个数，就是s作为前缀出现的次数，记这个次数为$sum[p]$，则我们只需对每个节点p求出$sum[p]$即可。

这可以在Trie插入字符串的过程中进行维护。

The solution

在Trie中，如果从根节点开始读入一个字符串s后走到的节点p非空，那么显然s是Trie中至少一个串的前缀，并且以p为根的子树内存储的字符串个数，就是s作为前缀出现的次数，记这个次数为$sum[p]$，则我们只需对每个节点p求出$sum[p]$即可。

这可以在Trie插入字符串的过程中进行维护。

插入一个新的字符串s并走到路上任一节点p时，节点p所代表的字符串一定是s的前缀，那么插入s后，节点p作为前缀出现的次数增加了1，我们直接令$sum[p] = sum[p] + 1$即可。

The solution

在Trie中，如果从根节点开始读入一个字符串s后走到的节点p非空，那么显然s是Trie中至少一个串的前缀，并且以p为根的子树内存储的字符串个数，就是s作为前缀出现的次数，记这个次数为$sum[p]$，则我们只需对每个节点p求出$sum[p]$即可。

这可以在Trie插入字符串的过程中进行维护。

插入一个新的字符串s并走到路上任一节点p时，节点p所代表的字符串一定是s的前缀，那么插入s后，节点p作为前缀出现的次数增加了1，我们直接令$sum[p] = sum[p] + 1$即可。

对于后缀，我们只需将原题中字符串全部逆序插入一棵新的Trie来处理即可。

优先队列

优先队列

普通的队列遵循FIFO的原则，但有时，我们希望元素有优先级，优先级较大的那个先出去。

优先队列

普通的队列遵循FIFO的原则，但有时，我们希望元素有优先级，优先级较大的那个先出去。

于是，我们可以得到优先队列这么一种概念上的结构，其支持以下两种操作：

优先队列

普通的队列遵循FIFO的原则，但有时，我们希望元素有优先级，优先级较大的那个先出去。

于是，我们可以得到优先队列这么一种概念上的结构，其支持以下两种操作：

1、在优先队列中加入一个已知优先级的元素。

优先队列

普通的队列遵循FIFO的原则，但有时，我们希望元素有优先级，优先级较大的那个先出去。

于是，我们可以得到优先队列这么一种概念上的结构，其支持以下两种操作：

1、在优先队列中加入一个已知优先级的元素。

2、获得/弹出优先队列中优先级最高的元素（不妨假设元素间优先级两两不同）。

优先队列

普通的队列遵循FIFO的原则，但有时，我们希望元素有优先级，优先级较大的那个先出去。

于是，我们可以得到优先队列这么一种概念上的结构，其支持以下两种操作：

1、在优先队列中加入一个已知优先级的元素。

2、获得/弹出优先队列中优先级最高的元素（不妨假设元素间优先级两两不同）。

而优先队列的一种最经典的实现就是堆，我们下面介绍一下。

堆

堆

堆是一种特殊的完全二叉树，其每个节点都有权值，并且保证任意一个节点的权值均不大于/不小于其子节点的权值。

堆

堆是一种特殊的完全二叉树，其每个节点都有权值，并且保证任意一个节点的权值均不大于/不小于其子节点的权值。

这样，二叉树的根的权值一定是最大的或者是最小的，分别称为大根堆和小根堆。

堆

堆是一种特殊的完全二叉树，其每个节点都有权值，并且保证任意一个节点的权值均不大于/不小于其子节点的权值。

这样，二叉树的根的权值一定是最大的或者是最小的，分别称为大根堆和小根堆。

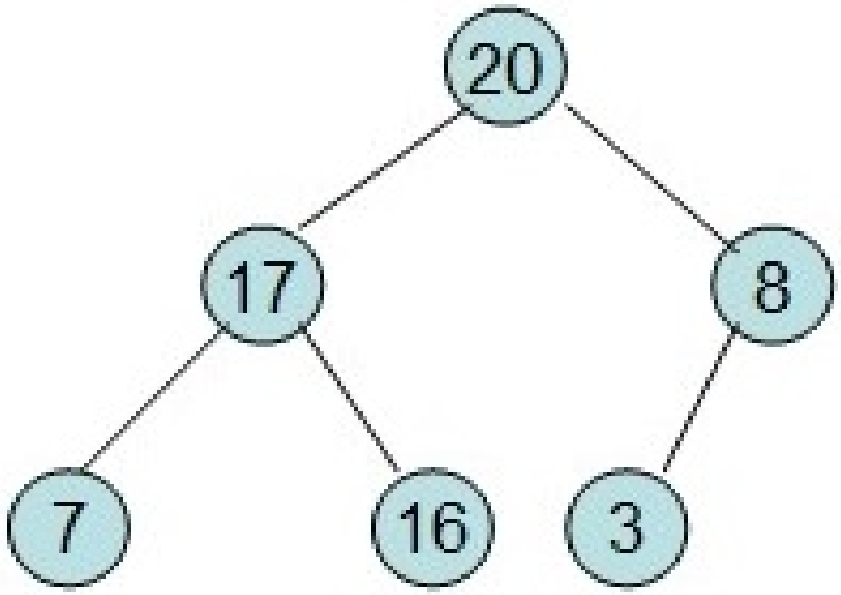
让我们以下面的大根堆为例来介绍一下堆的操作。

堆

堆是一种特殊的完全二叉树，其每个节点都有权值，并且保证任意一个节点的权值均不大于/不小于其子节点的权值。

这样，二叉树的根的权值一定是最大的或者是最小的，分别称为大根堆和小根堆。

让我们以下面的大根堆为例来介绍一下堆的操作。



插入

插入

1、在完全二叉树的末尾增加以这个元素为权值的节点。

插入

1、在完全二叉树的末尾增加以这个元素为权值的节点。

2、如果当前节点不为根，则比较当前节点与其父节点的权值，如果比父节点大，那么就将该节点与父节点交换并重复这个比较过程。

插入

1、在完全二叉树的末尾增加以这个元素为权值的节点。

2、如果当前节点不为根，则比较当前节点与其父节点的权值，如果比父节点大，那么就将该节点与父节点交换并重复这个比较过程。

我们可以用mspaint这个软件模拟一下在上一页的那个大根堆上插入10之后所进行的操作……

插入

1、在完全二叉树的末尾增加以这个元素为权值的节点。

2、如果当前节点不为根，则比较当前节点与其父节点的权值，如果比父节点大，那么就将该节点与父节点交换并重复这个比较过程。

我们可以用mspaint这个软件模拟一下在上一页的那个大根堆上插入10之后所进行的操作……

单次操作的时间复杂度很显然是$O(\log n)$的。

删除

删除

在堆中我们一般都是删除根节点，但是直接删除根节点并让子节点依次递补的方式很可能会破坏完全二叉树的结构。

删除

在堆中我们一般都是删除根节点，但是直接删除根节点并让子节点依次递补的方式很可能会破坏完全二叉树的结构。

事实上，我们通过将完全二叉树根节点与末尾节点权值交换，再删除末尾节点的方式来进行处理。

删除

在堆中我们一般都是删除根节点，但是直接删除根节点并让子节点依次递补的方式很可能会破坏完全二叉树的结构。

事实上，我们通过将完全二叉树根节点与末尾节点权值交换，再删除末尾节点的方式来进行处理。

接下来，我们就可以根据堆的性质自顶到下递补了。

删除

在堆中我们一般都是删除根节点，但是直接删除根节点并让子节点依次递补的方式很可能会破坏完全二叉树的结构。

事实上，我们通过将完全二叉树根节点与末尾节点权值交换，再删除末尾节点的方式来进行处理。

接下来，我们就可以根据堆的性质自顶到下递补了。

以大根堆为例，如果当前节点的两个子节点权值都不大于当前权值，那么这个递补过程就可以停止，否则，我们须将两个子节点中权值较大的那个与当前节点交换，并往下重复这一过程。

删除

在堆中我们一般都是删除根节点，但是直接删除根节点并让子节点依次递补的方式很可能会破坏完全二叉树的结构。

事实上，我们通过将完全二叉树根节点与末尾节点权值交换，再删除末尾节点的方式来进行处理。

接下来，我们就可以根据堆的性质自顶到下递补了。

以大根堆为例，如果当前节点的两个子节点权值都不大于当前权值，那么这个递补过程就可以停止，否则，我们须将两个子节点中权值较大的那个与当前节点交换，并往下重复这一过程。

我们可以继续用mspaint在上一页的堆上模拟一下删除最值时所进行的操作……

删除

在堆中我们一般都是删除根节点，但是直接删除根节点并让子节点依次递补的方式很可能会破坏完全二叉树的结构。

事实上，我们通过将完全二叉树根节点与末尾节点权值交换，再删除末尾节点的方式来进行处理。

接下来，我们就可以根据堆的性质自顶到下递补了。

以大根堆为例，如果当前节点的两个子节点权值都不大于当前权值，那么这个递补过程就可以停止，否则，我们须将两个子节点中权值较大的那个与当前节点交换，并往下重复这一过程。

我们可以继续用mspaint在上一页的堆上模拟一下删除最值时所进行的操作……

单次操作的时间复杂度很显然也是$O(\log n)$的。

初始化

初始化

假如你现在有一堆无序的数据，并且想要把它们组织成一个堆……

初始化

假如你现在有一堆无序的数据，并且想要把它们组织成一个堆……

一个很显然的想法就是在空堆上不停地插入这些数据，这样做的时间复杂度是$O(n \log n)$。

初始化

假如你现在有一堆无序的数据，并且想要把它们组织成一个堆……

一个很显然的想法就是在空堆上不停地插入这些数据，这样做的时间复杂度是$O(n \log n)$。

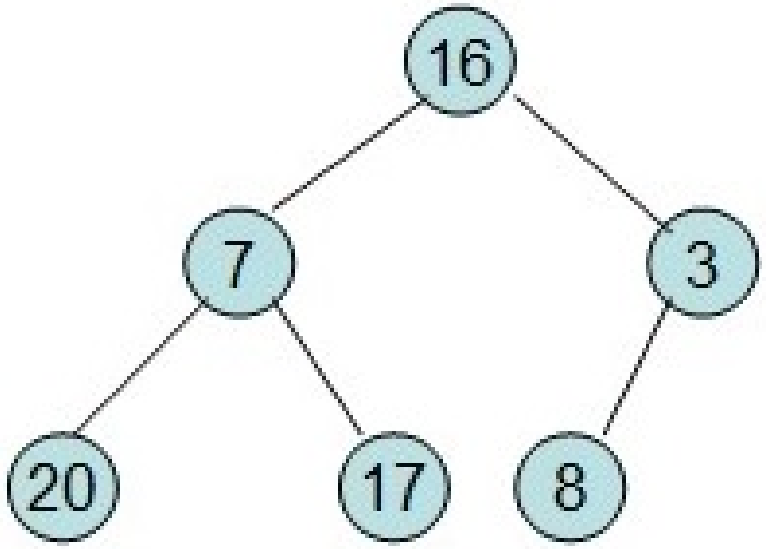
事实上有一种更好的线性建堆的方式，开始时，我们先将数据随意组织成二叉树，比如下图：

初始化

假如你现在有一堆无序的数据，并且想要把它们组织成一个堆……

一个很显然的想法就是在空堆上不停地插入这些数据，这样做的时间复杂度是$O(n \log n)$。

事实上有一种更好的线性建堆的方式，开始时，我们先将数据随意组织成二叉树，比如下图：



Next...

Next...

从堆的最下层开始对每个节点都进行处理，做到任意一个节点时，我们都要使得以该节点为根的子树是堆。

Next...

从堆的最下层开始对每个节点都进行处理，做到任意一个节点时，我们都要使得以该节点为根的子树是堆。

也就是说，我们通过自下而上，对每个节点都做一遍类似于删除时的递补操作来完成建堆。

Next...

从堆的最下层开始对每个节点都进行处理，做到任意一个节点时，我们都要使得以该节点为根的子树是堆。

也就是说，我们通过自下而上，对每个节点都做一遍类似于删除时的递补操作来完成建堆。

（可以用强大的mspaint工具来模拟一下操作，当然手画我并不反对）

Next...

从堆的最下层开始对每个节点都进行处理，做到任意一个节点时，我们都要使得以该节点为根的子树是堆。

也就是说，我们通过自下而上，对每个节点都做一遍类似于删除时的递补操作来完成建堆。

（可以用强大的mspaint工具来模拟一下操作，当然手画我并不反对）

由于堆结构的递归性，因此这个算法正确性比较显然，至于时间复杂度的话也很显然是线性的。

Next...

从堆的最下层开始对每个节点都进行处理，做到任意一个节点时，我们都要使得以该节点为根的子树是堆。

也就是说，我们通过自下而上，对每个节点都做一遍类似于删除时的递补操作来完成建堆。

（可以用强大的mspaint工具来模拟一下操作，当然手画我并不反对）

由于堆结构的递归性，因此这个算法正确性比较显然，至于时间复杂度的话也很显然是线性的。

接下来我们来简单提提堆的实现方式。

堆的实现

堆的实现

由于堆是完全二叉树，因此其可以很方便地在数组上实现。

堆的实现

由于堆是完全二叉树，因此其可以很方便地在数组上实现。

具体来说，可以令1号节点为根，第i号节点的左儿子为$i \times 2$号节点，右儿子为$i \times 2 + 1$号节点，那么完全二叉树便存储在了一个连续的区域上，其末尾的节点即为连续区域的最后一个元素。

堆的实现

由于堆是完全二叉树，因此其可以很方便地在数组上实现。

具体来说，可以令1号节点为根，第i号节点的左儿子为$i \times 2$号节点，右儿子为$i \times 2 + 1$号节点，那么完全二叉树便存储在了一个连续的区域上，其末尾的节点即为连续区域的最后一个元素。

而且显然，对第$k \geq 2$号节点，其父节点的编号都是$k/2$下取整。

堆的实现

由于堆是完全二叉树，因此其可以很方便地在数组上实现。

具体来说，可以令1号节点为根，第i号节点的左儿子为$i \times 2$号节点，右儿子为$i \times 2 + 1$号节点，那么完全二叉树便存储在了一个连续的区域上，其末尾的节点即为连续区域的最后一个元素。

而且显然，对第$k \geq 2$号节点，其父节点的编号都是$k/2$下取整。

接下来我们来一道题，让大家领会一下使用堆的技巧。

一道题

一道题

一个可重数集一开始为空，现在要不停地在数集中添加或删除元素，你的任务便是实时维护数集中的中位数。

一道题

一个可重数集一开始为空，现在要不停地在数集中添加或删除元素，你的任务便是实时维护数集中的中位数。

操作次数n满足$1 \leq n \leq 500000$。

一道题

一个可重数集一开始为空，现在要不停地在数集中添加或删除元素，你的任务便是实时维护数集中的中位数。

操作次数n满足$1 \leq n \leq 500000$。

假设我们只能用堆来解决这个问题。

The solution

The solution

先假设没有删除操作，来考虑如何维护中位数。

The solution

先假设没有删除操作，来考虑如何维护中位数。

我们可以实时用小根堆$T1$和大根堆$T2$维护数集中较大的一半元素和较小的一半元素。那么中位数便可以通过$T1$的最小值和$T2$的最大值求出。

The solution

先假设没有删除操作，来考虑如何维护中位数。

我们可以实时用小根堆$T1$和大根堆$T2$维护数集中较大的一半元素和较小的一半元素。那么中位数便可以通过$T1$的最小值和$T2$的最大值求出。

加入元素时，先判断其应该在$T1$还是$T2$中。

The solution

先假设没有删除操作，来考虑如何维护中位数。

我们可以实时用小根堆$T1$和大根堆$T2$维护数集中较大的一半元素和较小的一半元素。那么中位数便可以通过$T1$的最小值和$T2$的最大值求出。

加入元素时，先判断其应该在$T1$还是$T2$中。

加入之后，如果两个堆元素个数过于不平衡，即大小差距超过2，则不妨设$T1$的大小比$T2$大2，那么我们就可以将$T1$中最小的元素弹出并加入$T2$，反过来类似。

The solution

先假设没有删除操作，来考虑如何维护中位数。

我们可以实时用小根堆$T1$和大根堆$T2$维护数集中较大的一半元素和较小的一半元素。那么中位数便可以通过$T1$的最小值和$T2$的最大值求出。

加入元素时，先判断其应该在$T1$还是$T2$中。

加入之后，如果两个堆元素个数过于不平衡，即大小差距超过2，则不妨设$T1$的大小比$T2$大2，那么我们就可以将$T1$中最小的元素弹出并加入$T2$，反过来类似。

接下来我们考虑删除。

续

续

堆是不支持高效查找的，因此要直接删除任意一个元素非常困难。

续

堆是不支持高效查找的，因此要直接删除任意一个元素非常困难。

但注意到，使用堆的时候一般也只取堆顶的数据，因此我们考虑对$T1$和$T2$建立两个新的堆$T1'$与$T2'$，当要删除一个元素x一次的时候，根据大小将x分别塞入$T1'$与$T2'$中，表示x应该被删除，但现在还删不了，暂时记在那里。

续

堆是不支持高效查找的，因此要直接删除任意一个元素非常困难。

但注意到，使用堆的时候一般也只取堆顶的数据，因此我们考虑对$T1$和$T2$建立两个新的堆$T1'$与$T2'$，当要删除一个元素x一次的时候，根据大小将x分别塞入$T1'$与$T2'$中，表示x应该被删除，但现在还删不了，暂时记在那里。

我们在任意时刻，均检查$T1$的堆顶与$T1'$的堆顶是否相同，若相同，我们就同时弹出，进行实际的删除操作，对于$T2$堆类似。

续

堆是不支持高效查找的，因此要直接删除任意一个元素非常困难。

但注意到，使用堆的时候一般也只取堆顶的数据，因此我们考虑对$T1$和$T2$建立两个新的堆$T1'$与$T2'$，当要删除一个元素x一次的时候，根据大小将x分别塞入$T1'$与$T2'$中，表示x应该被删除，但现在还删不了，暂时记在那里。

我们在任意时刻，均检查$T1$的堆顶与$T1'$的堆顶是否相同，若相同，我们就同时弹出，进行实际的删除操作，对于$T2$堆类似。

这样我们就解决了删除的问题，总时间复杂度$O(n \log n)$。

续

堆是不支持高效查找的，因此要直接删除任意一个元素非常困难。

但注意到，使用堆的时候一般也只取堆顶的数据，因此我们考虑对$T1$和$T2$建立两个新的堆$T1'$与$T2'$，当要删除一个元素x一次的时候，根据大小将x分别塞入$T1'$与$T2'$中，表示x应该被删除，但现在还删不了，暂时记在那里。

我们在任意时刻，均检查$T1$的堆顶与$T1'$的堆顶是否相同，若相同，我们就同时弹出，进行实际的删除操作，对于$T2$堆类似。

这样我们就解决了删除的问题，总时间复杂度$O(n \log n)$。

接下来我们讲讲图。

图

图

如果我们不加限制地任取每一个节点的*next*节点集（即从这个点开始，下一步可以到达哪些节点），那么我们就可以得到一张图。

图

如果我们不加限制地任取每一个节点的*next*节点集（即从这个点开始，下一步可以到达哪些节点），那么我们就可以得到一张图。

如果u在v的*next*节点集当且仅当v在u的*next*节点集里，那么这张图就可以用于表示节点之间的相邻关系。

图

如果我们不加限制地任取每一个节点的*next*节点集（即从这个点开始，下一步可以到达哪些节点），那么我们就可以得到一张图。

如果u在v的*next*节点集当且仅当v在u的*next*节点集里，那么这张图就可以用于表示节点之间的相邻关系。

在图论中会以一种数学化的语言对图进行定义和介绍，我们这里只是比较简单地提提。

图的实现

图的实现

实现图最基本的想法就是保存每个节点的*next*节点集。

图的实现

实现图最基本的想法就是保存每个节点的*next*节点集。

而最基本的实现方式便是邻接矩阵和邻接表。

图的实现

实现图最基本的想法就是保存每个节点的*next*节点集。

而最基本的实现方式便是邻接矩阵和邻接表。

邻接矩阵就是用一个二维矩阵来存储任两个节点之间的信息，而邻接表是对每个节点来保存其*next*节点集（即该节点的出边）。

图的实现

实现图最基本的想法就是保存每个节点的*next*节点集。

而最基本的实现方式便是邻接矩阵和邻接表。

邻接矩阵就是用一个二维矩阵来存储任两个节点之间的信息，而邻接表是对每个节点来保存其*next*节点集（即该节点的出边）。

我们一般通过对每个节点建立一个链表来实现邻接表，当然，利用动态数组或者C++STL里的vector则可以用顺序表来实现邻接表。

图的实现

实现图最基本的想法就是保存每个节点的*next*节点集。

而最基本的实现方式便是邻接矩阵和邻接表。

邻接矩阵就是用一个二维矩阵来存储任两个节点之间的信息，而邻接表是对每个节点来保存其*next*节点集（即该节点的出边）。

我们一般通过对每个节点建立一个链表来实现邻接表，当然，利用动态数组或者C++STL里的vector则可以用顺序表来实现邻接表。

与树类似，对于图，可以从任意一个节点*u*开始进行DFS或BFS遍历找到与*u*相关的所有节点，这里也略去。

并查集的引入

并查集的引入

假如说我们有一张表示相邻关系的图……

并查集的引入

假如说我们有一张表示相邻关系的图……

考虑一个问题，如何判断任意两个节点u, v之间是否存在直接或间接的联系。

并查集的引入

假如说我们有一张表示相邻关系的图……

考虑一个问题，如何判断任意两个节点u, v之间是否存在直接或间接的联系。

一种想法是，从节点u开始遍历，如果能够遍历到节点v，那么u, v间便存在关联。

并查集的引入

假如说我们有一张表示相邻关系的图……

考虑一个问题，如何判断任意两个节点u, v之间是否存在直接或间接的联系。

一种想法是，从节点u开始遍历，如果能够遍历到节点v，那么u, v间便存在关联。

但是这样太慢，为了找到v，很可能要遍历整张图的大部分节点。

并查集的引入

假如说我们有一张表示相邻关系的图……

考虑一个问题，如何判断任意两个节点u, v之间是否存在直接或间接的联系。

一种想法是，从节点u开始遍历，如果能够遍历到节点v，那么u, v间便存在关联。

但是这样太慢，为了找到v，很可能要遍历整张图的大部分节点。

注意到两点间存在关联是一种等价关系，下面我们介绍一种专门用于等价关系的数据结构——并查集。

并查集

并查集

并查集主要用于处理等价关系（自反、对称、传递），其支持以下两种操作：

并查集

并查集主要用于处理等价关系（自反、对称、传递），其支持以下两种操作：

1、询问任意两个元素是否等价。

并查集

并查集主要用于处理等价关系（自反、对称、传递），其支持以下两种操作：

1、询问任意两个元素是否等价。

2、添加一条等价关系(u, v)，即使得u与v等价。

并查集

并查集主要用于处理等价关系（自反、对称、传递），其支持以下两种操作：

1、询问任意两个元素是否等价。

2、添加一条等价关系(u, v)，即使得u与v等价。

朴素地维护等价类并进行查找当然是可行的，不过速度较慢，下面介绍一种较快的实现方式：

并查集

并查集主要用于处理等价关系（自反、对称、传递），其支持以下两种操作：

1、询问任意两个元素是否等价。

2、添加一条等价关系(u, v)，即使得u与v等价。

朴素地维护等价类并进行查找当然是可行的，不过速度较慢，下面介绍一种较快的实现方式：

其基本思想是在一个等价类中寻找一个代表元，使得两节点之间存在关联等价于它们的代表元相同。

并查集

并查集主要用于处理等价关系（自反、对称、传递），其支持以下两种操作：

1、询问任意两个元素是否等价。

2、添加一条等价关系(u, v)，即使得u与v等价。

朴素地维护等价类并进行查找当然是可行的，不过速度较慢，下面介绍一种较快的实现方式：

其基本思想是在一个等价类中寻找一个代表元，使得两节点之间存在关联等价于它们的代表元相同。

我们考虑一下在已有基础上添加等价关系(u, v)时应该做什么——求u的代表元ru，v的代表元rv，将rv设成ru所代表的等价类的代表元（当然反过来也可以）。

实现

实现

假设一开始没有任何的等价关系，我们定义数组$root$，考虑一个元素p，我们给$root[p]$如下定义：

实现

假设一开始没有任何的等价关系，我们定义数组$root$，考虑一个元素p，我们给$root[p]$如下定义：

1、如果$root[p] = p$，那么p就是自己的代表元。

实现

假设一开始没有任何的等价关系，我们定义数组$root$，考虑一个元素p，我们给$root[p]$如下定义：

1、如果$root[p] = p$，那么p就是自己的代表元。

2、否则，p的代表元便是$root[p]$的代表元。

实现

假设一开始没有任何的等价关系，我们定义数组$root$，考虑一个元素p，我们给$root[p]$如下定义：

1、如果$root[p] = p$，那么p就是自己的代表元。

2、否则，p的代表元便是$root[p]$的代表元。

当查询p的代表元时，我们可以沿着$root$往上找。

实现

假设一开始没有任何的等价关系，我们定义数组$root$，考虑一个元素p，我们给$root[p]$如下定义：

1、如果$root[p] = p$，那么p就是自己的代表元。

2、否则，p的代表元便是$root[p]$的代表元。

当查询p的代表元时，我们可以沿着$root$往上找。

当添加(u, v)的时候，找到代表元ru与rv，令$root[ru] = rv$即可。

实现

假设一开始没有任何的等价关系，我们定义数组$root$，考虑一个元素p，我们给$root[p]$如下定义：

1、如果$root[p] = p$，那么p就是自己的代表元。

2、否则，p的代表元便是$root[p]$的代表元。

当查询p的代表元时，我们可以沿着$root$往上找。

当添加(u, v)的时候，找到代表元ru与rv，令$root[ru] = rv$即可。

很显然，处于一个等价类的所有元素按这种方式组织成了一棵树，树上所有非根节点p的父亲为$root[p]$。

实现

假设一开始没有任何的等价关系，我们定义数组$root$，考虑一个元素p，我们给$root[p]$如下定义：

1、如果$root[p] = p$，那么p就是自己的代表元。

2、否则，p的代表元便是$root[p]$的代表元。

当查询p的代表元时，我们可以沿着$root$往上找。

当添加(u, v)的时候，找到代表元ru与rv，令$root[ru] = rv$即可。

很显然，处于一个等价类的所有元素按这种方式组织成了一棵树，树上所有非根节点p的父亲为$root[p]$。

而所有等价类中的元素组织成了一片森林，森林中的树根即为某个等价类的代表元。

实现的优化

实现的优化

这种实现方式在特殊情况下效率会极大地退化（比如一条链）。

实现的优化

这种实现方式在特殊情况下效率会极大地退化（比如一条链）。

优化有两个——路径压缩和按秩合并。

实现的优化

这种实现方式在特殊情况下效率会极大地退化（比如一条链）。

优化有两个——路径压缩和按秩合并。

路径压缩的思想是，对于一个节点p，如果我们查询到了其代表元rp，那么我们直接令$root[p] = rp$即可，这样做的话，下次查询p的代表元时只需要看一次$root[p]$即可。

实现的优化

这种实现方式在特殊情况下效率会极大地退化（比如一条链）。

优化有两个——路径压缩和按秩合并。

路径压缩的思想是，对于一个节点p，如果我们查询到了其代表元rp，那么我们直接令$root[p] = rp$即可，这样做的话，下次查询p的代表元时只需要看一次$root[p]$即可。

按秩合并考虑的是森林中每棵树的高度，合并时把较矮的树合并到较高的树上。

实现的优化

这种实现方式在特殊情况下效率会极大地退化（比如一条链）。

优化有两个——路径压缩和按秩合并。

路径压缩的思想是，对于一个节点p，如果我们查询到了其代表元rp，那么我们直接令$root[p] = rp$即可，这样做的话，下次查询p的代表元时只需要看一次$root[p]$即可。

按秩合并考虑的是森林中每棵树的高度，合并时把较矮的树合并到较高的树上。

实际实现时，仅采用路径压缩即可获得足够快的速度，而且这个改动相当简单；按秩合并改动起来不方便，一般懒得写。

实现的优化

这种实现方式在特殊情况下效率会极大地退化（比如一条链）。

优化有两个——路径压缩和按秩合并。

路径压缩的思想是，对于一个节点p，如果我们查询到了其代表元rp，那么我们直接令$root[p] = rp$即可，这样做的话，下次查询p的代表元时只需要看一次$root[p]$即可。

按秩合并考虑的是森林中每棵树的高度，合并时把较矮的树合并到较高的树上。

实际实现时，仅采用路径压缩即可获得足够快的速度，而且这个改动相当简单；按秩合并改动起来不方便，一般懒得写。

单次操作（合并和查询）时间复杂度可以认为是$O(1)$，如果你的良心感到不安，那么你可以课后翻阅有关资料来搞清楚~

维护一些其它的信息

维护一些其它的信息

注意到我们在维护过程中使用了树结构，因此在并查集的过程中，我们还可以顺带维护一些别的东西。

维护一些其它的信息

注意到我们在维护过程中使用了树结构，因此在并查集的过程中，我们还可以顺带维护一些别的东西。

一个比较简单的是每一个等价类的大小，记其为*size*，这个值存储在每一个代表元中。

维护一些其它的信息

注意到我们在维护过程中使用了树结构，因此在并查集的过程中，我们还可以顺带维护一些别的东西。

一个比较简单的是每一个等价类的大小，记其为$size$，这个值存储在每一个代表元中。

当两个代表元ru, rv合并的时候（设$root[ru] = rv$），我们令$size[rv] += size[ru]$即可，其余均不变。

维护一些其它的信息

注意到我们在维护过程中使用了树结构，因此在并查集的过程中，我们还可以顺带维护一些别的东西。

一个比较简单的是每一个等价类的大小，记其为$size$，这个值存储在每一个代表元中。

当两个代表元ru, rv合并的时候（设$root[ru] = rv$），我们令$size[rv] += size[ru]$即可，其余均不变。

我们接下来来道例题来讲讲另一类数据的维护。

A classic example

A classic example

树上有n只猴子，编号1到n。1号猴子用它的尾巴勾着树枝。剩下的猴子都被其他的猴子用手抓着。

A classic example

树上有n只猴子，编号1到n。1号猴子用它的尾巴勾着树枝。剩下的猴子都被其他的猴子用手抓着。

每只猴子的每只手都可以抓住另一只猴子的尾巴。从0时刻开始m秒，每一秒都有一只猴子松开它的一只手。这会导致一些猴子掉到地上（它们在地上也能继续松开它们的手，猴子落地的时间很短可以不计）。

A classic example

树上有n只猴子，编号1到n。1号猴子用它的尾巴勾着树枝。剩下的猴子都被其他的猴子用手抓着。

每只猴子的每只手都可以抓住另一只猴子的尾巴。从0时刻开始m秒，每一秒都有一只猴子松开它的一只手。这会导致一些猴子掉到地上（它们在地上也能继续松开它们的手，猴子落地的时间很短可以不计）。

已知猴子间抓与被抓住的关系信息，和它们放开手的顺序，求出每一只猴子落地的时间。

A classic example

树上有n只猴子，编号1到n。1号猴子用它的尾巴勾着树枝。剩下的猴子都被其他的猴子用手抓着。

每只猴子的每只手都可以抓住另一只猴子的尾巴。从0时刻开始m秒，每一秒都有一只猴子松开它的一只手。这会导致一些猴子掉到地上（它们在地上也能继续松开它们的手，猴子落地的时间很短可以不计）。

已知猴子间抓与被抓住的关系信息，和它们放开手的顺序，求出每一只猴子落地的时间。

$1 \leq n \leq 200000, 1 \leq m \leq 2n$

The solution

The solution

我们将整个过程倒过来看，那么问题就是在已知部分连接关系的基础上再添加一些连通关系，并求出每只猴子与1号猴子连通的时间。

The solution

我们将整个过程倒过来看，那么问题就是在已知部分连接关系的基础上再添加一些连通关系，并求出每只猴子与1号猴子连通的时间。

这个问题就可以运用并查集解决，但是我们这里需要维护额外的信息。

维护

维护

在并查集基本信息$root$的基础上，记t_i为第i只猴子与第$root_i$只猴子相连通的时间，特别地，$t_1 = 0$。

维护

在并查集基本信息$root$的基础上，记t_i为第i只猴子与第$root_i$只猴子相连通的时间，特别地，$t_1 = 0$。

在合并时，若其中一棵树含有1，就将不含1的子树（记其根为r）合并到含1的子树上，并取t_r为当前时间。

维护

在并查集基本信息$root$的基础上，记t_i为第i只猴子与第$root_i$只猴子相连通的时间，特别地，$t_1 = 0$。

在合并时，若其中一棵树含有1，就将不含1的子树（记其根为r）合并到含1的子树上，并取t_r为当前时间。

从一个节点向上找根的时候，取祖先各个t中时间最晚的那个即为该节点与根所连通的时间。

维护

在并查集基本信息$root$的基础上，记t_i为第i只猴子与第$root_i$只猴子相连通的时间，特别地，$t_1 = 0$。

在合并时，若其中一棵树含有1，就将不含1的子树（记其根为r）合并到含1的子树上，并取t_r为当前时间。

从一个节点向上找根的时候，取祖先各个t中时间最晚的那个即为该节点与根所连通的时间。

t_i与$root_i$一样，在路径压缩过程中也能进行类似的维护，大家可以自己想一想。

维护

在并查集基本信息$root$的基础上，记t_i为第i只猴子与第$root_i$只猴子相连通的时间，特别地，$t_1 = 0$。

在合并时，若其中一棵树含有1，就将不含1的子树（记其根为r）合并到含1的子树上，并取t_r为当前时间。

从一个节点向上找根的时候，取祖先各个t中时间最晚的那个即为该节点与根所连通的时间。

t_i与$root_i$一样，在路径压缩过程中也能进行类似的维护，大家可以自己想一想。

最后，对每只猴子都进行一次向上找根的操作，判断其是否与1号猴子连通以及连通的时间。

Hash

Hash

中文名哈希（或散列），就是把任意长度的输入通过某种确定的算法变换成固定长度的值（可以是数、字符串等），该输出就是Hash值。

Hash

中文名哈希（或散列），就是把任意长度的输入通过某种确定的算法变换成固定长度的值（可以是数、字符串等），该输出就是Hash值。

Hash最基本的一个性质就是两个相同的元素必定拥有相同的hash值。

Hash

中文名哈希（或散列），就是把任意长度的输入通过某种确定的算法变换成固定长度的值（可以是数、字符串等），该输出就是Hash值。

Hash最基本的一个性质就是两个相同的元素必定拥有相同的hash值。

但是，不保证不同元素拥有不同的hash值，事实上，这个性质一般是不成立的。

Hash

中文名哈希（或散列），就是把任意长度的输入通过某种确定的算法变换成固定长度的值（可以是数、字符串等），该输出就是Hash值。

Hash最基本的一个性质就是两个相同的元素必定拥有相同的hash值。

但是，不保证不同元素拥有不同的hash值，事实上，这个性质一般是不成立的。

Hash在OI中最主要的应用就是存储和检索，即保存一堆数据并支持查询某个数据是否存在，当然还能够用来判定你的答案是否正确。

Hash检索

Hash检索

基本思想就是将数据库中每个数据通过某种hash算法变成一个值，将数据按hash值分类，这样我们就可以通过值来寻找数据。

Hash检索

基本思想就是将数据库中每个数据通过某种hash算法变成一个值，将数据按hash值分类，这样我们就可以通过值来寻找数据。

在查询时将待查数据也hash成一个值，如果待查数据的hash值在数据库中不存在，那么我们就可以判定待查数据不存在。

Hash检索

基本思想就是将数据库中每个数据通过某种hash算法变成一个值，将数据按hash值分类，这样我们就可以通过值来寻找数据。

在查询时将待查数据也hash成一个值，如果待查数据的hash值在数据库中不存在，那么我们就可以判定待查数据不存在。

否则，我们找到拥有那个值的所有元素并一一比对。

Hash检索

基本思想就是将数据库中每个数据通过某种hash算法变成一个值，将数据按hash值分类，这样我们就可以通过值来寻找数据。

在查询时将待查数据也hash成一个值，如果待查数据的hash值在数据库中不存在，那么我们就可以判定待查数据不存在。

否则，我们找到拥有那个值的所有元素并一一比对。

这样做的好处就在于能将不方便索引的数据与便于索引的值联系起来，降低每次检索的代价。

Hash检索

基本思想就是将数据库中每个数据通过某种hash算法变成一个值，将数据按hash值分类，这样我们就可以通过值来寻找数据。

在查询时将待查数据也hash成一个值，如果待查数据的hash值在数据库中不存在，那么我们就可以判定待查数据不存在。

否则，我们找到拥有那个值的所有元素并一一比对。

这样做的好处就在于能将不方便索引的数据与便于索引的值联系起来，降低每次检索的代价。

对于hash值，我们可以用多种多样的高效的数据结构来存储和查询，最简单的一种便是hash表。

Hash表

Hash表

Hash表即建立一个顺序表，每个hash值对应的位置用来存放拥有该hash值的元素。

Hash表

Hash表即建立一个顺序表，每个hash值对应的位置用来存放拥有该hash值的元素。

假如对于一个数据集，我们用hash算法求出来的值两两不同，那么便相安无事。

Hash表

Hash表即建立一个顺序表，每个hash值对应的位置用来存放拥有该hash值的元素。

假如对于一个数据集，我们用hash算法求出来的值两两不同，那么便相安无事。

否则，我们不妨设数据*data*1和数据*data*2拥有相同的hash值，此时称发生了一次hash值的碰撞。

Hash表

Hash表即建立一个顺序表，每个hash值对应的位置用来存放拥有该hash值的元素。

假如对于一个数据集，我们用hash算法求出来的值两两不同，那么便相安无事。

否则，我们不妨设数据*data*1和数据*data*2拥有相同的hash值，此时称发生了一次hash值的碰撞。

下面来讨论一下处理碰撞常见的几种方式。

处理碰撞

处理碰撞

1、开散列闭寻址法：即对每个hash值建立一个链表来存储拥有该hash值的所有元素，hash表中存储对应链表的表头。

处理碰撞

1、开散列闭寻址法：即对每个hash值建立一个链表来存储拥有该hash值的所有元素，hash表中存储对应链表的表头。

2、闭散列开寻址法：当这个hash值已经有数据时，我们进行开放寻址，比如线性探查，即判断当前hash值$+1, +2, +3...$对应的位置是否为空，或二次探查（即$+1, +4, +9...$）

处理碰撞

1、开散列闭寻址法：即对每个hash值建立一个链表来存储拥有该hash值的所有元素，hash表中存储对应链表的表头。

2、闭散列开寻址法：当这个hash值已经有数据时，我们进行开放寻址，比如线性探查，即判断当前hash值$+1, +2, +3...$对应的位置是否为空，或二次探查（即$+1, +4, +9...$）

在查询时，若待查hash值对应的元素存在且与待查元素不同，则也要按照开放寻址的过程查找，直到找到待查元素或到达一个空位置为止。

处理碰撞

1、开散列闭寻址法：即对每个hash值建立一个链表来存储拥有该hash值的所有元素，hash表中存储对应链表的表头。

2、闭散列开寻址法：当这个hash值已经有数据时，我们进行开放寻址，比如线性探查，即判断当前hash值$+1, +2, +3...$对应的位置是否为空，或二次探查（即$+1, +4, +9...$）

在查询时，若待查hash值对应的元素存在且与待查元素不同，则也要按照开放寻址的过程查找，直到找到待查元素或到达一个空位置为止。

3、再散列：将当前hash值再进行一次hash来寻找下一个地址。

关于哈希的性能

关于哈希的性能

设数据集大小为n，hash表大小m，并且假设hash值是均匀的。

关于哈希的性能

设数据集大小为n，hash表大小m，并且假设hash值是均匀的。

如果n远小于m，那么hash的操作基本上可以看作是$O(1)$的。

关于哈希的性能

设数据集大小为n，hash表大小m，并且假设hash值是均匀的。

如果n远小于m，那么hash的操作基本上可以看作是$O(1)$的。

根据生日攻击理论，当$n = O(\sqrt{m})$时，就有相当大的概率发生至少一次碰撞。

关于哈希的性能

设数据集大小为n，hash表大小m，并且假设hash值是均匀的。

如果n远小于m，那么hash的操作基本上可以看作是$O(1)$的。

根据生日攻击理论，当$n = O(\sqrt{m})$时，就有相当大的概率发生至少一次碰撞。

当n达到m的一半时，若用闭散列法，则hash的性能会开始快速下降。

关于哈希的性能

设数据集大小为n，hash表大小m，并且假设hash值是均匀的。

如果n远小于m，那么hash的操作基本上可以看作是$O(1)$的。

根据生日攻击理论，当$n = O(\sqrt{m})$时，就有相当大的概率发生至少一次碰撞。

当n达到m的一半时，若用闭散列法，则hash的性能会开始快速下降。

下面我们简单讨论一下OI中字符串的hash。

字符串hash

字符串hash

OI中，字符串一种最简单的处理方式便是hash，在单纯检索字符串的时候，hash比Trie实现更为方便。

字符串hash

OI中，字符串一种最简单的处理方式便是hash，在单纯检索字符串的时候，hash比Trie实现更为方便。

在OI中，对字符串最常用的hash方式便是强行进制转换。

字符串hash

OI中，字符串一种最简单的处理方式便是hash，在单纯检索字符串的时候，hash比Trie实现更为方便。

在OI中，对字符串最常用的hash方式便是强行进制转换。

将字符串中每个字符转变为ASCII码，并将该串强行看作一个*base*（≥ 128，最好是素数）进制的数，再转化为十进制即可。

字符串hash

OI中，字符串一种最简单的处理方式便是hash，在单纯检索字符串的时候，hash比Trie实现更为方便。

在OI中，对字符串最常用的hash方式便是强行进制转换。

将字符串中每个字符转变为ASCII码，并将该串强行看作一个*base*（≥ 128，最好是素数）进制的数，再转化为十进制即可。

当然这个数可能很大，因此我们可以对其取模（比如10^9+7或者其它质数），便得到了hash值。

字符串hash

OI中，字符串一种最简单的处理方式便是hash，在单纯检索字符串的时候，hash比Trie实现更为方便。

在OI中，对字符串最常用的hash方式便是强行进制转换。

将字符串中每个字符转变为ASCII码，并将该串强行看作一个*base*（≥ 128，最好是素数）进制的数，再转化为十进制即可。

当然这个数可能很大，因此我们可以对其取模（比如10^9+7或者其它质数），便得到了hash值。

在解决问题的时候，如果hash取值范围足够大（远大于字符串hash值之间两两比较的次数），碰撞概率极低，那么我们就可以通过hash值的比较来直接判断两字符串是否相等。

字符串hash

OI中，字符串一种最简单的处理方式便是hash，在单纯检索字符串的时候，hash比Trie实现更为方便。

在OI中，对字符串最常用的hash方式便是强行进制转换。

将字符串中每个字符转变为ASCII码，并将该串强行看作一个*base*（≥ 128，最好是素数）进制的数，再转化为十进制即可。

当然这个数可能很大，因此我们可以对其取模（比如10^9+7或者其它质数），便得到了hash值。

在解决问题的时候，如果hash取值范围足够大（远大于字符串hash值之间两两比较的次数），碰撞概率极低，那么我们就可以通过hash值的比较来直接判断两字符串是否相等。

下面介绍种用hash来实现的字符串匹配算法。

字符串匹配RK算法

字符串匹配RK算法

假设有字符串S和T，长度分别为n, m，我们现在想要判断T作为子串在S中出现的次数。

字符串匹配RK算法

假设有字符串S和T，长度分别为n, m，我们现在想要判断T作为子串在S中出现的次数。

那么很显然的一个想法就是求出S串中所有长度为m的子串的hash值，然后与T的hash值比较。

字符串匹配RK算法

假设有字符串S和T，长度分别为n, m，我们现在想要判断T作为子串在S中出现的次数。

那么很显然的一个想法就是求出S串中所有长度为m的子串的hash值，然后与T的hash值比较。

为此，我们使用刚刚提到的字符串hash算法，记所使用的进制为$base$。

字符串匹配RK算法

假设有字符串S和T，长度分别为n, m，我们现在想要判断T作为子串在S中出现的次数。

那么很显然的一个想法就是求出S串中所有长度为m的子串的hash值，然后与T的hash值比较。

为此，我们使用刚刚提到的字符串hash算法，记所使用的进制为$base$。

先求出S长度为m的前缀的hash值，接下来我们考虑求该子串删掉第一个字符并在末尾添加一个字符后的hash值。

字符串匹配RK算法

假设有字符串S和T，长度分别为n, m，我们现在想要判断T作为子串在S中出现的次数。

那么很显然的一个想法就是求出S串中所有长度为m的子串的hash值，然后与T的hash值比较。

为此，我们使用刚刚提到的字符串hash算法，记所使用的进制为$base$。

先求出S长度为m的前缀的hash值，接下来我们考虑求该子串删掉第一个字符并在末尾添加一个字符后的hash值。

记原串为ab..c，新串为b..cd，那么新串hash值即为原串hash值扣掉a的部分，再乘上一个$base$，最后加上d的部分。

字符串匹配RK算法

假设有字符串S和T，长度分别为n, m，我们现在想要判断T作为子串在S中出现的次数。

那么很显然的一个想法就是求出S串中所有长度为m的子串的hash值，然后与T的hash值比较。

为此，我们使用刚刚提到的字符串hash算法，记所使用的进制为$base$。

先求出S长度为m的前缀的hash值，接下来我们考虑求该子串删掉第一个字符并在末尾添加一个字符后的hash值。

记原串为ab..c，新串为b..cd，那么新串hash值即为原串hash值扣掉a的部分，再乘上一个$base$，最后加上d的部分。

可以看到，算法总体时间复杂度是$O(n+m)$的。

思考与讨论（可能不讲）

思考与讨论（可能不讲）

1、如何快速求出给定字符串的任一子串的hash值（或乘上*base*的若干次幂的结果）?

思考与讨论（可能不讲）

1、如何快速求出给定字符串的任一子串的hash值（或乘上*base*的若干次幂的结果）?

2、如何估计字符串hash算法中hash值两两比较的次数（考虑一下生日攻击和RK算法的情况）?

思考与讨论（可能不讲）

1、如何快速求出给定字符串的任一子串的hash值（或乘上*base*的若干次幂的结果）?

2、如何估计字符串hash算法中hash值两两比较的次数（考虑一下生日攻击和RK算法的情况）?

3、之前讲过用Trie来解决最受欢迎的前后缀问题，那么用字符串hash能够解决吗？

思考与讨论（可能不讲）

1、如何快速求出给定字符串的任一子串的hash值（或乘上*base*的若干次幂的结果）?

2、如何估计字符串hash算法中hash值两两比较的次数（考虑一下生日攻击和RK算法的情况）?

3、之前讲过用Trie来解决最受欢迎的前后缀问题，那么用字符串hash能够解决吗？

4、为什么hash的模和*base*最好取成素数？

第一部分到此结束

第一部分到此结束

~~现在，你有两分钟的时间将试卷上的答案转涂到答题卡上。~~

第一部分到此结束

~~现在，你有两分钟的时间将试卷上的答案转涂到答题卡上。~~

于是我们就讲完了上面的基础数据结构，顺带可以休息一会儿。

第一部分到此结束

~~现在，你有两分钟的时间将试卷上的答案转涂到答题卡上。~~

于是我们就讲完了上面的基础数据结构，顺带可以休息一会儿。

接下来我们讲讲处理序列问题时会用到的一些方法以及高级数据结构。

ST表

ST表

ST表英文全名Sparse Table，其思想基于倍增算法，我们先以序列为例子。

ST表

ST表英文全名Sparse Table，其思想基于倍增算法，我们先以序列为例子。

设当前我们有长度为n的序列$a_1, a_2, ...a_n$，要维护的信息为最大值。

ST表

ST表英文全名Sparse Table，其思想基于倍增算法，我们先以序列为例子。

设当前我们有长度为n的序列$a_1, a_2, ...a_n$，要维护的信息为最大值。

那么，我们记$f[i][j]$表示从第i个点开始的2^j个元素的最大值，则显然$f[i][0] = a_i$，并且对任意正整数k，都有

$$f[i][k] = \max(f[i][k-1], f[i+2^{k-1}][k-1])。$$

ST表

ST表英文全名Sparse Table，其思想基于倍增算法，我们先以序列为例子。

设当前我们有长度为n的序列$a_1, a_2, ...a_n$，要维护的信息为最大值。

那么，我们记$f[i][j]$表示从第i个点开始的2^j个元素的最大值，则显然$f[i][0] = a_i$，并且对任意正整数k，都有

$$f[i][k] = \max(f[i][k-1], f[i+2^{k-1}][k-1])。$$

这样，我们就知道了所有含有2^k个元素的区间$[i, i+2^k)$的信息。

运用ST表

运用ST表

接下来，我们就可以用这些信息来查询该序列任意一个区间的最大值了。

运用ST表

接下来，我们就可以用这些信息来查询该序列任意一个区间的最大值了。

假设我们当前要查询的区间是$[l, r)$，那么很显然一定存在k，使得$l \leq r - 2^k \leq l + 2^k \leq r$。

运用ST表

接下来，我们就可以用这些信息来查询该序列任意一个区间的最大值了。

假设我们当前要查询的区间是$[l, r)$，那么很显然一定存在k，使得$l \leq r - 2^k \leq l + 2^k \leq r$。

则$[l, r]$可以被两个区间$[l, l + 2^k)$ 与$[r - 2^k, r)$完全覆盖，于是我们只要求$f[l][k]$与$f[r - 2^k][k]$的最大值即可。

运用ST表

接下来，我们就可以用这些信息来查询该序列任意一个区间的最大值了。

假设我们当前要查询的区间是$[l, r)$，那么很显然一定存在k，使得$l \leq r - 2^k \leq l + 2^k \leq r$。

则$[l, r]$可以被两个区间$[l, l + 2^k)$ 与$[r - 2^k, r)$完全覆盖，于是我们只要求$f[l][k]$与$f[r - 2^k][k]$的最大值即可。

预处理ST表的时候，第二维只有$O(\log n)$，故处理整个ST表的时间复杂度为$O(n \log n)$。

运用ST表

接下来，我们就可以用这些信息来查询该序列任意一个区间的最大值了。

假设我们当前要查询的区间是$[l, r)$，那么很显然一定存在k，使得$l \leq r - 2^k \leq l + 2^k \leq r$。

则$[l, r]$可以被两个区间$[l, l + 2^k)$ 与$[r - 2^k, r)$完全覆盖，于是我们只要求$f[l][k]$与$f[r - 2^k][k]$的最大值即可。

预处理ST表的时候，第二维只有$O(\log n)$，故处理整个ST表的时间复杂度为$O(n \log n)$。

利用*math*库的log函数或者预处理，我们可以对每个区间长度均求出对应的k，那么查询便是$O(1)$的。

在树结构上试试……

在树结构上试试……

在树上我们可以很显然地得到一个节点的第k代祖先的概念。

在树结构上试试……

在树上我们可以很显然地得到一个节点的第k代祖先的概念。

记$f[i][k]$表示节点i的第2^k代祖先，那么很显然$f[i][0]$即为节点i的父节点，可以做类似的递推。

在树结构上试试……

在树上我们可以很显然地得到一个节点的第k代祖先的概念。

记$f[i][k]$表示节点i的第2^k代祖先，那么很显然$f[i][0]$即为节点i的父节点，可以做类似的递推。

我们接下来介绍一下LCA问题及其运用ST表的在线倍增算法。

LCA问题

LCA问题

全名Lowest Common Ancestors，即最近公共祖先问题。

LCA问题

全名Lowest Common Ancestors，即最近公共祖先问题。

给定一棵树，根为$root$，LCA问题是对两个节点u, v求出如下一个点p，满足：

LCA问题

全名Lowest Common Ancestors，即最近公共祖先问题。

给定一棵树，根为$root$，LCA问题是对两个节点u, v求出如下一个点p，满足：

1、p是u, v的公共祖先。

LCA问题

全名Lowest Common Ancestors，即最近公共祖先问题。

给定一棵树，根为$root$，LCA问题是对两个节点u, v求出如下一个点p，满足：

1、p是u, v的公共祖先。

2、所有满足条件1的点都是p的祖先。

LCA问题

全名Lowest Common Ancestors，即最近公共祖先问题。

给定一棵树，根为$root$，LCA问题是对两个节点u, v求出如下一个点p，满足：

1、p是u, v的公共祖先。

2、所有满足条件1的点都是p的祖先。

这个点就称为两个节点u, v的LCA，记为$LCA(u, v)$。

LCA问题

全名Lowest Common Ancestors，即最近公共祖先问题。

给定一棵树，根为$root$，LCA问题是对两个节点u, v求出如下一个点p，满足：

1、p是u, v的公共祖先。

2、所有满足条件1的点都是p的祖先。

这个点就称为两个节点u, v的LCA，记为$LCA(u, v)$。

联系一下Trie，我们可以知道，两个串的最长公共前缀即为两个串在Trie中对应的节点的LCA。

LCA问题的在线倍增算法

LCA问题的在线倍增算法

先预处理一张ST表，其中$f[i][k]$表示节点i的第2^k代祖先。

LCA问题的在线倍增算法

先预处理一张ST表，其中$f[i][k]$表示节点i的第2^k代祖先。

设两个节点u, v的深度分别为$d[u]$与$d[v]$，不妨设$d[u] \leq d[v]$。

LCA问题的在线倍增算法

先预处理一张ST表，其中$f[i][k]$表示节点i的第2^k代祖先。

设两个节点u, v的深度分别为$d[u]$与$d[v]$，不妨设$d[u] \leq d[v]$。

我们先利用ST表将节点v往上走$O(\log n)$次（每次找一个最大的k，使得$2^k \leq d[v] - d[u]$，令$v = f[v][k]$），直到u, v深度相同，如果此时$u = v$，那么u即为所求。

LCA问题的在线倍增算法

先预处理一张ST表，其中$f[i][k]$表示节点i的第2^k代祖先。

设两个节点u, v的深度分别为$d[u]$与$d[v]$，不妨设$d[u] \leq d[v]$。

我们先利用ST表将节点v往上走$O(\log n)$次（每次找一个最大的k，使得$2^k \leq d[v] - d[u]$，令$v = f[v][k]$），直到u, v深度相同，如果此时$u = v$，那么u即为所求。

否则，考虑如果pu, pv分别是u, v的祖先，并且$pu \neq pv$，那么$LCA(u, v) = LCA(pu, pv)$这一显然的性质，我们可以类似地，令u, v节点在保持二者父节点不相同的情况下同时往上走$O(\log n)$次，最后$f[u][0]$即为所求。

LCA问题的在线倍增算法

先预处理一张ST表，其中$f[i][k]$表示节点i的第2^k代祖先。

设两个节点u, v的深度分别为$d[u]$与$d[v]$，不妨设$d[u] \leq d[v]$。

我们先利用ST表将节点v往上走$O(\log n)$次（每次找一个最大的k，使得$2^k \leq d[v] - d[u]$，令$v = f[v][k]$），直到u, v深度相同，如果此时$u = v$，那么u即为所求。

否则，考虑如果pu, pv分别是u, v的祖先，并且$pu \neq pv$，那么$LCA(u, v) = LCA(pu, pv)$这一显然的性质，我们可以类似地，令u, v节点在保持二者父节点不相同的情况下同时往上走$O(\log n)$次，最后$f[u][0]$即为所求。

注意到每做一次之后所得的k都是单调递减的，故单次查询实现起来的复杂度就是$O(\log n)$，同时该算法还可以顺带维护一下节点u, v在往上跳的过程中，经过的树边的信息，这里不详细提了。

另一个序列问题

另一个序列问题

我们有长度为n的序列$a_1, a_2, ..., a_n$，以及若干个询问。

另一个序列问题

我们有长度为n的序列$a_1, a_2, ..., a_n$，以及若干个询问。

每个询问给定一个区间$[l, r]$，询问这个区间内所有数的和。

另一个序列问题

我们有长度为n的序列$a_1, a_2, ..., a_n$，以及若干个询问。

每个询问给定一个区间$[l, r]$，询问这个区间内所有数的和。

则我们只要处理出序列前k项和S_k，那么区间$[l, r]$的和即为$S_r - S_{l-1}$。

另一个序列问题

我们有长度为n的序列$a_1, a_2, ..., a_n$，以及若干个询问。

每个询问给定一个区间$[l, r]$，询问这个区间内所有数的和。

则我们只要处理出序列前k项和S_k，那么区间$[l, r]$的和即为$S_r - S_{l-1}$。

我们加一个操作——给序列中一个数加上某个值。

另一个序列问题

我们有长度为n的序列$a_1, a_2, ..., a_n$，以及若干个询问。

每个询问给定一个区间$[l, r]$，询问这个区间内所有数的和。

则我们只要处理出序列前k项和S_k，那么区间$[l, r]$的和即为$S_r - S_{l-1}$。

我们加一个操作——给序列中一个数加上某个值。

那么这时候S_k便不能预处理出来了，但是基于这个思路，我们可以引入一种新的数据结构——树状数组来快速地维护序列的前缀和，

树状数组

树状数组

英文名Binary Indexed Tree(B.I.T)或Fenwick Tree。

树状数组

英文名Binary Indexed Tree(B.I.T)或Fenwick Tree。

在树状数组中，我们维护一个新序列$c_1, c_2, ..., c_n$，并且取$c_1 = a_1$，$c_2 = a_1 + a_2$，$c_3 = a_3$，$c_4 = a_1 + ... + a_4$，...

树状数组

英文名Binary Indexed Tree(B.I.T)或Fenwick Tree。

在树状数组中，我们维护一个新序列$c_1, c_2, ..., c_n$，并且取$c_1 = a_1$，$c_2 = a_1 + a_2$，$c_3 = a_3$，$c_4 = a_1 + ... + a_4$，...

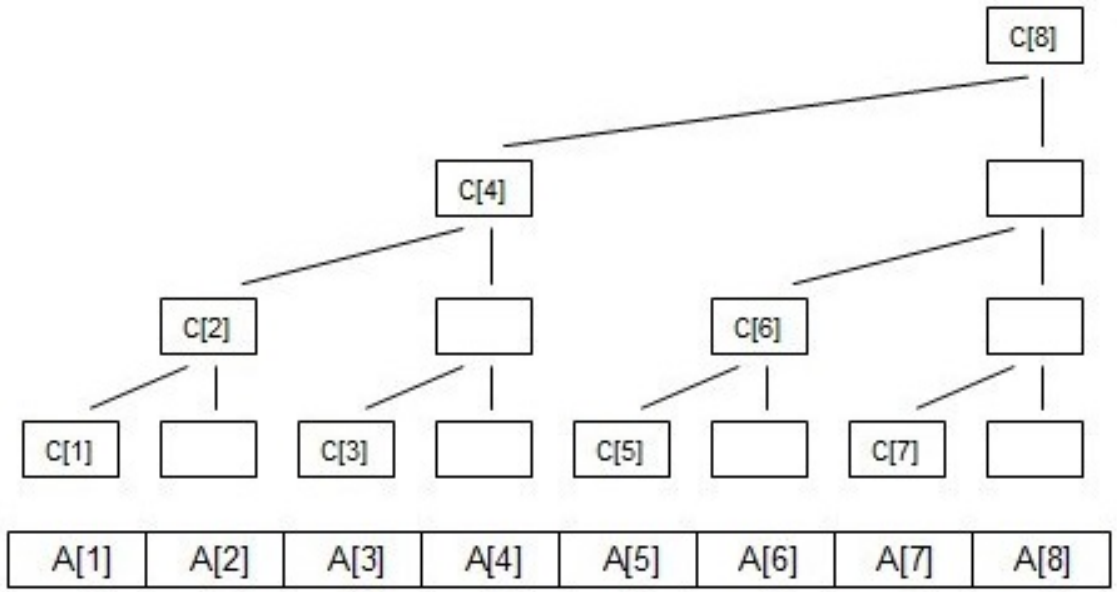
即取c_i为i及i之前的某些元素的和，具体关系如下图，c的每个元素管辖的范围即为其在序列a中的求和范围。

树状数组

英文名Binary Indexed Tree(B.I.T)或Fenwick Tree。

在树状数组中，我们维护一个新序列$c_1, c_2, ..., c_n$，并且取$c_1 = a_1$，$c_2 = a_1 + a_2$，$c_3 = a_3$，$c_4 = a_1 + ... + a_4$，...

即取c_i为i及i之前的某些元素的和，具体关系如下图，c的每个元素管辖的范围即为其在序列a中的求和范围。



具体描述一下……

具体描述一下……

我们比对一下下标的二进制表示以及c中对应下标元素求和范围内元素个数，对应的值如下：

具体描述一下……

我们比对一下下标的二进制表示以及c中对应下标元素求和范围内元素个数，对应的值如下：

下 标	二进制	范 围	下 标	二进制	范 围
1	001	1	5	101	1
2	010	2	6	110	2
3	011	1	7	111	1
4	100	4	8	1000	8

具体描述一下……

我们比对一下下标的二进制表示以及c中对应下标元素求和范围内元素个数，对应的值如下：

下 标	二进制	范 围	下 标	二进制	范 围
1	001	1	5	101	1
2	010	2	6	110	2
3	011	1	7	111	1
4	100	4	8	1000	8

再结合树状数组中元素的管辖范围图示，我们定义函数$lowbit(x)$，表示在2的幂次中，能够整除x的最大的数。

具体描述一下……

我们比对一下下标的二进制表示以及c中对应下标元素求和范围内元素个数，对应的值如下：

下 标	二进制	范 围	下 标	二进制	范 围
1	001	1	5	101	1
2	010	2	6	110	2
3	011	1	7	111	1
4	100	4	8	1000	8

再结合树状数组中元素的管辖范围图示，我们定义函数$lowbit(x)$，表示在2的幂次中，能够整除x的最大的数。

那么很显然，一个位置的管辖范围便是$[x - lowbit(x) + 1, x]$。

如何快速求lowbit(x)

如何快速求lowbit(x)

lowbit函数的一个作用就是将x的最低的非0位与其它部分分开。

如何快速求lowbit(x)

lowbit函数的一个作用就是将x的最低的非0位与其它部分分开。

记x最低的非0位为第$b \geq 0$位，则考虑$x - 1$。

如何快速求lowbit(x)

lowbit函数的一个作用就是将x的最低的非0位与其它部分分开。

记x最低的非0位为第$b \geq 0$位，则考虑$x-1$。

在$x-1$的二进制表示中，由于减法退位，第b位一定为0，而从最低位到第$b-1$位全部为1。

如何快速求lowbit(x)

lowbit函数的一个作用就是将x的最低的非0位与其它部分分开。

记x最低的非0位为第$b \geq 0$位，则考虑$x-1$。

在$x-1$的二进制表示中，由于减法退位，第b位一定为0，而从最低位到第$b-1$位全部为1。

而从第$b+1$位开始，$x-1$与x的二进制表示是相同的。

如何快速求lowbit(x)

lowbit函数的一个作用就是将x的最低的非0位与其它部分分开。

记x最低的非0位为第$b \geq 0$位，则考虑$x-1$。

在$x-1$的二进制表示中，由于减法退位，第b位一定为0，而从最低位到第$b-1$位全部为1。

而从第$b+1$位开始，$x-1$与x的二进制表示是相同的。

这样，我们令x与$x-1$做按二进制位与操作，便得到了$x - lowbit(x)$，那么求$lowbit(x)$也相当方便。

如何快速求lowbit(x)

lowbit函数的一个作用就是将x的最低的非0位与其它部分分开。

记x最低的非0位为第$b \geq 0$位，则考虑$x-1$。

在$x-1$的二进制表示中，由于减法退位，第b位一定为0，而从最低位到第$b-1$位全部为1。

而从第$b+1$位开始，$x-1$与x的二进制表示是相同的。

这样，我们令x与$x-1$做按二进制位与操作，便得到了$x-lowbit(x)$，那么求$lowbit(x)$也相当方便。

另外，计算机的补码特性使得$lowbit(x) = x\&(-x)$同样成立，大家可以按类似的过程自行讨论，事实上这个写法在OI界更常用一些。

如何求前缀和

如何求前缀和

假设我们现在要求区间$[1, x]$内元素的和。

如何求前缀和

假设我们现在要求区间$[1, x]$内元素的和。

那么我们考虑c_x，其管辖范围为$[x - lowbit(x) + 1, x]$，将其加入结果中，接下来我们只要考虑$[1, x - lowbit(x)]$内元素的和了。

如何求前缀和

假设我们现在要求区间$[1, x]$内元素的和。

那么我们考虑c_x，其管辖范围为$[x - lowbit(x) + 1, x]$，将其加入结果中，接下来我们只要考虑$[1, x - lowbit(x)]$内元素的和了。

注意到，$x - lowbit(x)$的二进制表示中1的个数总比x的少1，因此，进行$O(\log n)$步上述操作后x必然为0，求和完毕。

如何求前缀和

假设我们现在要求区间$[1, x]$内元素的和。

那么我们考虑c_x，其管辖范围为$[x - lowbit(x) + 1, x]$，将其加入结果中，接下来我们只要考虑$[1, x - lowbit(x)]$内元素的和了。

注意到，$x - lowbit(x)$的二进制表示中1的个数总比x的少1，因此，进行$O(\log n)$步上述操作后x必然为0，求和完毕。

而对于任意一个区间$[l, r]$，我们只需要求$[1, l - 1]$与$[1, r]$内元素的和即可。

如何单点修改

如何单点修改

假设我们现在要将a_x加上b，接下来我们就要考虑，c中哪些位置的求和范围包含了a_x。

如何单点修改

假设我们现在要将a_x加上b，接下来我们就要考虑，c中哪些位置的求和范围包含了a_x。

首先x本身是肯定包含的，即我们要令$c_x+=b$，我们考虑下一个包含x的位置在哪里。

如何单点修改

假设我们现在要将a_x加上b，接下来我们就要考虑，c中哪些位置的求和范围包含了a_x。

首先x本身是肯定包含的，即我们要令$c_x += b$，我们考虑下一个包含x的位置在哪里。

由于所有管辖范围大小为$lowbit(x)$的区间互不相交，那么下一个位置y的$lowbit$值应该比x的来得大，并且y本身也要比x大。

如何单点修改

假设我们现在要将a_x加上b，接下来我们就要考虑，c中哪些位置的求和范围包含了a_x。

首先x本身是肯定包含的，即我们要令$c_x += b$，我们考虑下一个包含x的位置在哪里。

由于所有管辖范围大小为$lowbit(x)$的区间互不相交，那么下一个位置y的$lowbit$值应该比x的来得大，并且y本身也要比x大。

则显然，取$y = x + lowbit(x)$即符合条件，以此类推，再反复令$y = y + lowbit(y)$，我们可以找到所有包含x的位置。

思考与讨论（可能不讲）

思考与讨论（可能不讲）

如果要求支持区间加法和单点查询这两个操作，那么应如何调整使得其能用树状数组实现？

思考与讨论（可能不讲）

如果要求支持区间加法和单点查询这两个操作，那么应如何调整使得其能用树状数组实现？

如果要求支持区间加法和区间求和这两个操作，如何使用多个树状数组实现？

思考与讨论（可能不讲）

如果要求支持区间加法和单点查询这两个操作，那么应如何调整使得其能用树状数组实现？

如果要求支持区间加法和区间求和这两个操作，如何使用多个树状数组实现？

如果我们要支持单点与一个值取max和查询区间前k个元素最大值，可以使用树状数组实现吗？如果是纯粹的单点修改呢？如果要查询任意区间的最大值呢？

线段树

线段树

接下来我们介绍解决以上两类区间问题的通用方法——线段树，其支持区间修改和区间查询，功能相当强大。

线段树

接下来我们介绍解决以上两类区间问题的通用方法——线段树，其支持区间修改和区间查询，功能相当强大。

线段树这一块的水很深，我们这里只能初步介绍一下线段树如何工作以及如何做一些非常基本的操作。

线段树

接下来我们介绍解决以上两类区间问题的通用方法——线段树，其支持区间修改和区间查询，功能相当强大。

线段树这一块的水很深，我们这里只能初步介绍一下线段树如何工作以及如何做一些非常基本的操作。

由于线段树画图注解太麻烦，这一部分我们很难离开mspaint的帮助……

Introduction

Introduction

线段树英文名segment tree，是一种特殊的二叉树。

Introduction

线段树英文名segment tree，是一种特殊的二叉树。

它的基本思想便是将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点，根节点对应区间为$[1, n]$。

Introduction

线段树英文名segment tree，是一种特殊的二叉树。

它的基本思想便是将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点，根节点对应区间为$[1, n]$。

对于线段树中的每一个非叶子节点$[a, b]$，它的左儿子表示的区间为$[a, \frac{a+b}{2}]$，右儿子表示的区间为$[\frac{a+b}{2} + 1, b]$，其中，除法均为向下取整。

Introduction

线段树英文名segment tree，是一种特殊的二叉树。

它的基本思想便是将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点，根节点对应区间为$[1, n]$。

对于线段树中的每一个非叶子节点$[a, b]$，它的左儿子表示的区间为$[a, \frac{a+b}{2}]$，右儿子表示的区间为$[\frac{a+b}{2} + 1, b]$，其中，除法均为向下取整。

在我们现在所接触到的线段树实现中，一般以存储堆的方式存储线段树，即节点i的左儿子为$i \times 2$，右儿子为$i \times 2 + 1$。

Introduction

线段树英文名segment tree，是一种特殊的二叉树。

它的基本思想便是将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点，根节点对应区间为$[1, n]$。

对于线段树中的每一个非叶子节点$[a, b]$，它的左儿子表示的区间为$[a, \frac{a+b}{2}]$，右儿子表示的区间为$[\frac{a+b}{2} + 1, b]$，其中，除法均为向下取整。

在我们现在所接触到的线段树实现中，一般以存储堆的方式存储线段树，即节点i的左儿子为$i \times 2$，右儿子为$i \times 2 + 1$。

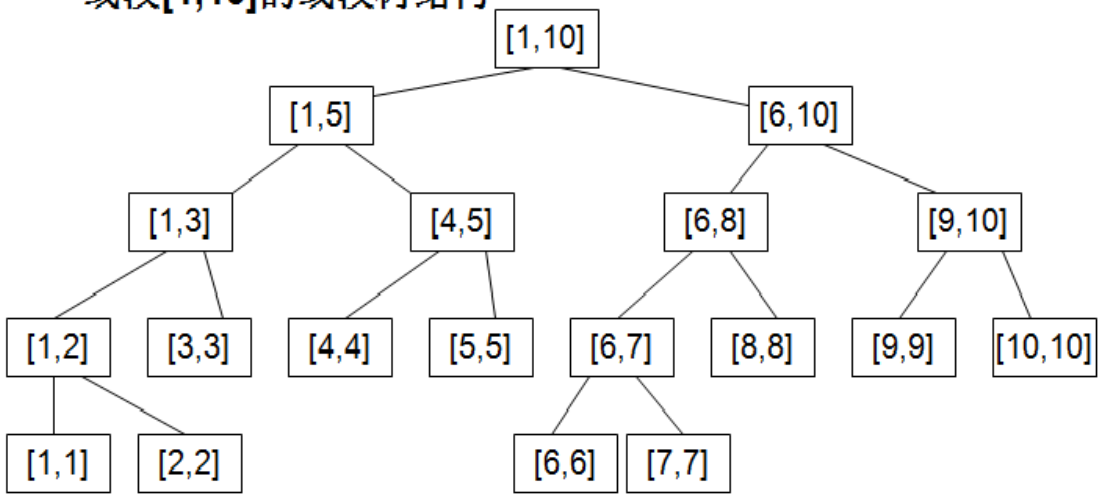
很显然，一棵线段树的树高是$O(\log n)$的。

一个例子

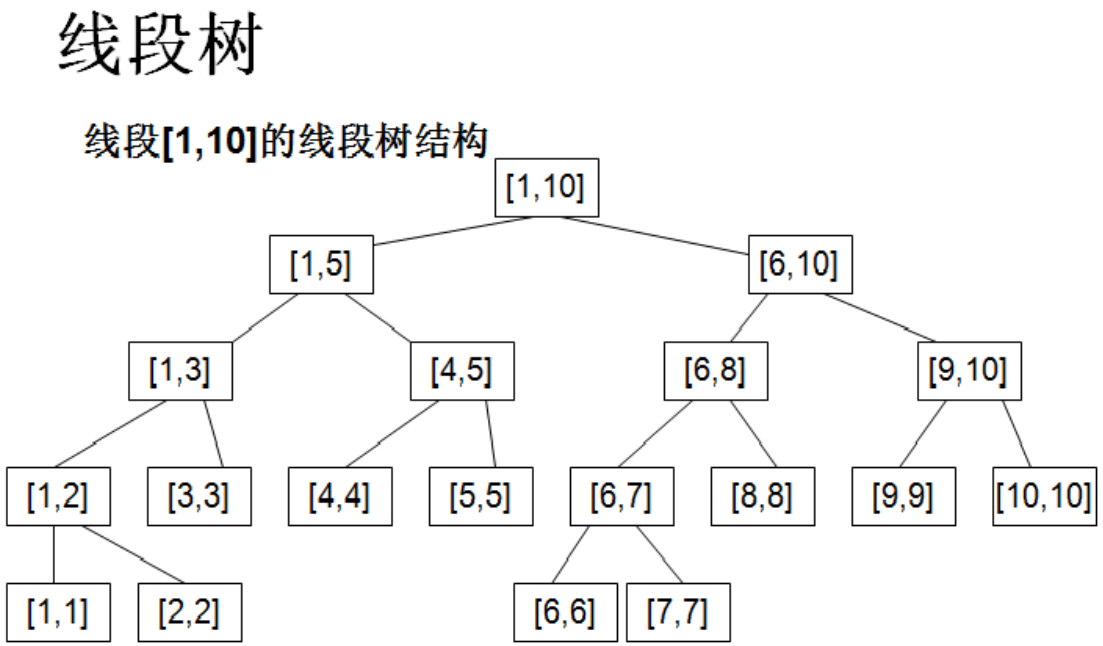
一个例子

线段树

线段[1,10]的线段树结构



一个例子



想必大家这时候对线段树都有了最基本的感觉，接下来我们以维护区间最大值为例，讲一讲线段树是如何工作的。

线段树上维护信息

线段树上维护信息

对线段树上的每个节点，记录其范围的左右边界，除此之外，我们再记录一个量$maxv$，表示该节点范围内所有元素的最大值。

线段树上维护信息

对线段树上的每个节点，记录其范围的左右边界，除此之外，我们再记录一个量*maxv*，表示该节点范围内所有元素的最大值。

对于一个叶节点，其*maxv*值显然，而对于一个非叶节点，其*maxv*值便是两个子节点的*maxv*值的最大值。

线段树上维护信息

对线段树上的每个节点，记录其范围的左右边界，除此之外，我们再记录一个量*maxv*，表示该节点范围内所有元素的最大值。

对于一个叶节点，其*maxv*值显然，而对于一个非叶节点，其*maxv*值便是两个子节点的*maxv*值的最大值。

接下来我们讲讲如何进行查询及修改。

查询

查询

假如我们要查询区间$I_0 = [2, 9]$。

查询

假如我们要查询区间$I_0 = [2, 9]$。

从根节点开始，如果当前节点的区间$I = [L, R]$满足$I \subseteq I_0$，那么我们直接利用节点I存储的区间信息即可。

查询

假如我们要查询区间$I_0 = [2, 9]$。

从根节点开始，如果当前节点的区间$I = [L, R]$满足$I \subseteq I_0$，那么我们直接利用节点I存储的区间信息即可。

否则，如果I与I_0相交，那么我们分别判断一下I_0是否与左右儿子的区间有公共部分，若有公共部分，则往下递归，获取I_0与对应子节点区间的交区间的信息。

查询

假如我们要查询区间$I_0 = [2, 9]$。

从根节点开始，如果当前节点的区间$I = [L, R]$满足$I \subseteq I_0$，那么我们直接利用节点I存储的区间信息即可。

否则，如果I与I_0相交，那么我们分别判断一下I_0是否与左右儿子的区间有公共部分，若有公共部分，则往下递归，获取I_0与对应子节点区间的交区间的信息。

显然，到叶子节点的时候，I要不然是I_0的子区间，要不然就与I_0不相交，即到叶节点的时候就不会再递归下去了。

查询

假如我们要查询区间$I_0 = [2, 9]$。

从根节点开始，如果当前节点的区间$I = [L, R]$满足$I \subseteq I_0$，那么我们直接利用节点I存储的区间信息即可。

否则，如果I与I_0相交，那么我们分别判断一下I_0是否与左右儿子的区间有公共部分，若有公共部分，则往下递归，获取I_0与对应子节点区间的交区间的信息。

显然，到叶子节点的时候，I要不然是I_0的子区间，要不然就与I_0不相交，即到叶节点的时候就不会再递归下去了。

可以看出，在线段树的每层上我们最多会遍历4个区间，所以单次查询的时间复杂度为$O(\log n)$。

单点修改

单点修改

设我们要修改位置x的信息。

单点修改

设我们要修改位置x的信息。

首先，我们可以很容易地找到保存位置x的信息的叶节点并进行修改。

单点修改

设我们要修改位置x的信息。

首先，我们可以很容易地找到保存位置x的信息的叶节点并进行修改。

但是显然不可能这么简单就结束了——所有包含了位置x的节点都需要修改信息。

单点修改

设我们要修改位置x的信息。

首先，我们可以很容易地找到保存位置x的信息的叶节点并进行修改。

但是显然不可能这么简单就结束了——所有包含了位置x的节点都需要修改信息。

在递归下去找到位置x后回溯的过程中，自底到上让每个节点重新利用其两个子节点的信息更新即可，时间复杂度$O(\log n)$。

区间修改

区间修改

假设我们要将区间$I_0 = [2, 9]$上的每个元素都加上一个常数c。

区间修改

假设我们要将区间$I_0 = [2, 9]$上的每个元素都加上一个常数c。

但是显然，逐一修改区间内每个元素的代价太高。

区间修改

假设我们要将区间$I_0 = [2, 9]$上的每个元素都加上一个常数c。

但是显然，逐一修改区间内每个元素的代价太高。

我们用一个新的修改方法——lazy_tag来解决这个问题。

lazy_tag

lazy_tag

首先，在线段树上的每个节点中加入一个叫lazy_tag的变量，其意义是该区间内所有的数本应共同加上，但是却没有真正加上的值，其初始值为0。

lazy_tag

首先，在线段树上的每个节点中加入一个叫lazy_tag的变量，其意义是该区间内所有的数本应共同加上，但是却没有真正加上的值，其初始值为0。

在修改区间l_0的时候，我们先用区间查询的方法找到一系列包含在l_0中的区间，我们不逐一修改这些区间中的元素，而是将该区间内需要修改的量记到lazy_tag上（打标记）——即令这些节点的lazy_tag 变量加上c。

lazy_tag

首先，在线段树上的每个节点中加入一个叫lazy_tag的变量，其意义是该区间内所有的数本应共同加上，但是却没有真正加上的值，其初始值为0。

在修改区间l_0的时候，我们先用区间查询的方法找到一系列包含在l_0中的区间，我们不逐一修改这些区间中的元素，而是将该区间内需要修改的量记到lazy_tag上（打标记）——即令这些节点的lazy_tag 变量加上c。

同时，这些节点所存储的区间最大值$maxv$也要加上c。

lazy_tag

首先，在线段树上的每个节点中加入一个叫lazy_tag的变量，其意义是该区间内所有的数本应共同加上，但是却没有真正加上的值，其初始值为0。

在修改区间l_0的时候，我们先用区间查询的方法找到一系列包含在l_0中的区间，我们不逐一修改这些区间中的元素，而是将该区间内需要修改的量记到lazy_tag上（打标记）——即令这些节点的lazy_tag 变量加上c。

同时，这些节点所存储的区间最大值$maxv$也要加上c。

在递归回溯的时候，我们同样也要更新遍历到的所有节点的$maxv$信息。

lazy_tag

首先，在线段树上的每个节点中加入一个叫lazy_tag的变量，其意义是该区间内所有的数本应共同加上，但是却没有真正加上的值，其初始值为0。

在修改区间l_0的时候，我们先用区间查询的方法找到一系列包含在l_0中的区间，我们不逐一修改这些区间中的元素，而是将该区间内需要修改的量记到lazy_tag上（打标记）——即令这些节点的lazy_tag 变量加上c。

同时，这些节点所存储的区间最大值$maxv$也要加上c。

在递归回溯的时候，我们同样也要更新遍历到的所有节点的$maxv$信息。

与区间查询一样，这个过程只会遍历到$O(\log n)$个区间，时间复杂度同样是$O(\log n)$。

标记的下传

标记的下传

我们刚才修改了区间[6, 8]的lazy_tag变量，但如果现在要查询区间[6, 7]的最大值，那么对应节点并没有进行相应的修改，这时候我们就需要将区间[6, 8]的lazy_tag下传，即进行一部分真实的修改。

标记的下传

我们刚才修改了区间[6, 8]的lazy_tag变量，但如果现在要查询区间[6, 7]的最大值，那么对应节点并没有进行相应的修改，这时候我们就需要将区间[6, 8]的lazy_tag下传，即进行一部分真实的修改。

lazy_tag的下传也相当简单——在修改和查询操作中，遍历到一个节点并且要访问它的子节点之前，将该节点的lazy_tag下传到其两个子节点即可，相当于对两个子节点所代表的区间进行本应该进行但还没进行的区间修改，下传后该节点lazy_tag重置。

标记的下传

我们刚才修改了区间[6, 8]的lazy_tag变量，但如果现在要查询区间[6, 7]的最大值，那么对应节点并没有进行相应的修改，这时候我们就需要将区间[6, 8]的lazy_tag下传，即进行一部分真实的修改。

lazy_tag的下传也相当简单——在修改和查询操作中，遍历到一个节点并且要访问它的子节点之前，将该节点的lazy_tag下传到其两个子节点即可，相当于对两个子节点所代表的区间进行本应该进行但还没进行的区间修改，下传后该节点lazy_tag重置。

现在，假如我们查询区间[6, 7]，我们在区间[6, 8]的时候，便会将[6, 8]上的修改下传到区间[6, 7]与[8, 8]，查询的结果也就正确了。

标记的下传

我们刚才修改了区间[6, 8]的lazy_tag变量，但如果现在要查询区间[6, 7]的最大值，那么对应节点并没有进行相应的修改，这时候我们就需要将区间[6, 8]的lazy_tag下传，即进行一部分真实的修改。

lazy_tag的下传也相当简单——在修改和查询操作中，遍历到一个节点并且要访问它的子节点之前，将该节点的lazy_tag下传到其两个子节点即可，相当于对两个子节点所代表的区间进行本应该进行但还没进行的区间修改，下传后该节点lazy_tag重置。

现在，假如我们查询区间[6, 7]，我们在区间[6, 8]的时候，便会将[6, 8]上的修改下传到区间[6, 7]与[8, 8]，查询的结果也就正确了。

值得提醒的是，由于我们并不需要访问区间[6, 7]的子节点，因此区间[6, 7]的lazy_tag没有必要下传。

线段树与树状数组、ST表的比较

线段树与树状数组、ST表的比较

不可否认，线段树在处理区间问题上的能力非常强大，那么我们为什么还要讲ST表与树状数组呢？

线段树与树状数组、ST表的比较

不可否认，线段树在处理区间问题上的能力非常强大，那么我们为什么还要讲ST表与树状数组呢？

因为ST表处理序列问题的查询复杂度为常数，在没有修改而询问很多的情况下，ST表有着自己的时间优势。

线段树与树状数组、ST表的比较

不可否认，线段树在处理区间问题上的能力非常强大，那么我们为什么还要讲ST表与树状数组呢？

因为ST表处理序列问题的查询复杂度为常数，在没有修改而询问很多的情况下，ST表有着自己的时间优势。

此外，ST表的思想也能运用于其它方面，比如LCA问题，或者NOIP那道开车旅行问题。

线段树与树状数组、ST表的比较

不可否认，线段树在处理区间问题上的能力非常强大，那么我们为什么还要讲ST表与树状数组呢？

因为ST表处理序列问题的查询复杂度为常数，在没有修改而询问很多的情况下，ST表有着自己的时间优势。

此外，ST表的思想也能运用于其它方面，比如LCA问题，或者NOIP那道开车旅行问题。

而讲树状数组的理由也很简单——树状数组常数小，代码简单，适合竞速。

线段树与树状数组、ST表的比较

不可否认，线段树在处理区间问题上的能力非常强大，那么我们为什么还要讲ST表与树状数组呢？

因为ST表处理序列问题的查询复杂度为常数，在没有修改而询问很多的情况下，ST表有着自己的时间优势。

此外，ST表的思想也能运用于其它方面，比如LCA问题，或者NOIP那道开车旅行问题。

而讲树状数组的理由也很简单——树状数组常数小，代码简单，适合竞速。

而且，如果你们以后了解树套树以及二维树状数组的时候，会更深刻地理解代码简单这个理由，并且知道二维树状数组可以解决一些二维线段树所不能解决的问题。

线段树与树状数组、ST表的比较

不可否认，线段树在处理区间问题上的能力非常强大，那么我们为什么还要讲ST表与树状数组呢？

因为ST表处理序列问题的查询复杂度为常数，在没有修改而询问很多的情况下，ST表有着自己的时间优势。

此外，ST表的思想也能运用于其它方面，比如LCA问题，或者NOIP那道开车旅行问题。

而讲树状数组的理由也很简单——树状数组常数小，代码简单，适合竞速。

而且，如果你们以后了解树套树以及二维树状数组的时候，会更深刻地理解代码简单这个理由，并且知道二维树状数组可以解决一些二维线段树所不能解决的问题。

下面我们来道例题结束一下今天早上的课。

一道例题

一道例题

设有一个长度为n的数列$a_1, a_2, ..., a_n$，要求支持以下几种操作：

一道例题

设有一个长度为n的数列$a_1, a_2, ..., a_n$，要求支持以下几种操作：

1、查询任意一个子区间内所有元素的和（当然输出的结果肯定是要模的）。

一道例题

设有一个长度为n的数列$a_1, a_2, ..., a_n$，要求支持以下几种操作：

1、查询任意一个子区间内所有元素的和（当然输出的结果肯定是要模的）。

2、区间乘法：给某一个子区间内所有的数乘上c。

一道例题

设有一个长度为n的数列$a_1, a_2, ..., a_n$，要求支持以下几种操作：

1、查询任意一个子区间内所有元素的和（当然输出的结果肯定是要模的）。

2、区间乘法：给某一个子区间内所有的数乘上c。

3、区间加法：给某一个子区间内所有的数加上c。

一道例题

设有一个长度为n的数列$a_1, a_2, \ldots, a_n$，要求支持以下几种操作：

1、查询任意一个子区间内所有元素的和（当然输出的结果肯定是要模的）。

2、区间乘法：给某一个子区间内所有的数乘上c。

3、区间加法：给某一个子区间内所有的数加上c。

我们就不说数据范围了，要求用线段树，单次操作$O(\log n)$即可。

The solution

The solution

每个区间上存储五个信息——区间左右端点，区间和，乘法的lazy_tag（记为变量x），加法的lazy_tag（记为变量y）。

The solution

每个区间上存储五个信息——区间左右端点，区间和，乘法的lazy_tag（记为变量x），加法的lazy_tag（记为变量y）。

为了方便，我们强制规定x先起作用，即lazy_tag表示的修改是先乘上x再加上y，初始时显然$x = 1, y = 0$。

The solution

每个区间上存储五个信息——区间左右端点，区间和，乘法的lazy_tag（记为变量x），加法的lazy_tag（记为变量y）。

为了方便，我们强制规定x先起作用，即lazy_tag表示的修改是先乘上x再加上y，初始时显然$x = 1, y = 0$。

考虑区间乘c的打标记，这时候我们要令x和元素和均乘c，但除此之外，因为我们的规定，变量y也要乘上c。

The solution

每个区间上存储五个信息——区间左右端点，区间和，乘法的lazy_tag（记为变量x），加法的lazy_tag（记为变量y）。

为了方便，我们强制规定x先起作用，即lazy_tag表示的修改是先乘上x再加上y，初始时显然$x = 1, y = 0$。

考虑区间乘c的打标记，这时候我们要令x和元素和均乘c，但除此之外，因为我们的规定，变量y也要乘上c。

而区间加c的打标记就比较简单了，大家自己想想。

The solution

每个区间上存储五个信息——区间左右端点，区间和，乘法的lazy_tag（记为变量x），加法的lazy_tag（记为变量y）。

为了方便，我们强制规定x先起作用，即lazy_tag表示的修改是先乘上x再加上y，初始时显然$x = 1, y = 0$。

考虑区间乘c的打标记，这时候我们要令x和元素和均乘c，但除此之外，因为我们的规定，变量y也要乘上c。

而区间加c的打标记就比较简单了，大家自己想想。

在下传标记的时候，也要注意先下传x再下传y。

The solution

每个区间上存储五个信息——区间左右端点，区间和，乘法的lazy_tag（记为变量x），加法的lazy_tag（记为变量y）。

为了方便，我们强制规定x先起作用，即lazy_tag表示的修改是先乘上x再加上y，初始时显然$x = 1, y = 0$。

考虑区间乘c的打标记，这时候我们要令x和元素和均乘c，但除此之外，因为我们的规定，变量y也要乘上c。

而区间加c的打标记就比较简单了，大家自己想想。

在下传标记的时候，也要注意先下传x再下传y。

线段树类的问题主要在于知道要运用线段树以及如何用线段树维护区间信息，当我们弄清楚这些之后，问题就迎刃而解了。

The solution

每个区间上存储五个信息——区间左右端点，区间和，乘法的lazy_tag（记为变量x），加法的lazy_tag（记为变量y）。

为了方便，我们强制规定x先起作用，即lazy_tag表示的修改是先乘上x再加上y，初始时显然$x = 1, y = 0$。

考虑区间乘c的打标记，这时候我们要令x和元素和均乘c，但除此之外，因为我们的规定，变量y也要乘上c。

而区间加c的打标记就比较简单了，大家自己想想。

在下传标记的时候，也要注意先下传x再下传y。

线段树类的问题主要在于知道要运用线段树以及如何用线段树维护区间信息，当我们弄清楚这些之后，问题就迎刃而解了。

听起来很简单，但是对于复杂的题目，维护区间信息的方式可能极为复杂，这里我们就不深入讨论了。

结束啦

结束啦

Questions are welcomed!

结束啦

Questions are welcomed!

内容比较多，大家好好啃一下，下午的题大家尽管屠，别手下留情~

结束啦

Questions are welcomed!

内容比较多，大家好好啃一下，下午的题大家尽管屠，别手下留情~

大家快去吃饭吧~