

# 《数据结构》 PA3辅导课

2018年12月26日 6C300

## 主要内容: PA3辅导

- ❖ CST2018 3-1-2 Max : 邢健开
- ❖ CST2018 3-2 Not Found : 王聿中
- ❖ CST2018 3-3 Hacker : 蒋林
- ❖ CST2018 3-4 kth : 唐天映
- ❖ CST2018 3-5 Prefix : 张子键
- ❖ CST2018 3-6 Match : 赵成钢

**CST2018 3-1-2 Max**

**邢健开**

## 问题描述

- ❖ 题面很简单，给出一个序列，询问给定区间最大连续子段和
- ❖ 单点修改
- ❖ 序列长度、操作次数在 $5 \times 10^5$ 级别

## 问题分析

- ❖ 首先这是一个序列问题
- ❖ 其次询问是关于区间的信息
- ❖ 所以我们自然就会想到使用在序列上维护区间信息的数据结构
- ❖ 这样的数据结构有树状数组、线段树、各种平衡树等等
- ❖ 由于询问不满足区间减法（假设询问为 $[1, r]$ ，最优答案不为 $[1-r]$ 的最优解减去 $[1, 1-1]$ 的最优解），所以不方便用树状数组
- ❖ 又本题序列长度是固定的，且没有翻转操作，所以思路简单的线段树就是很好的选择了

# 线段树

- ❖ 我们先来回顾一下什么是线段树
- ❖  $\log$ 层的二叉树；每个节点代表一段区间；父节点的信息通过合并子节点的信息得到
- ❖ 单点修改：找到叶节点，更新后依次更新路径上的每个区间
- ❖ 区间查询：任意一个区间被最多 $\log$ 个节点所代表的区间覆盖，要询问任意一个区间的信息，就将分解出的这些节点的信息依次合并即可

## 难点要点

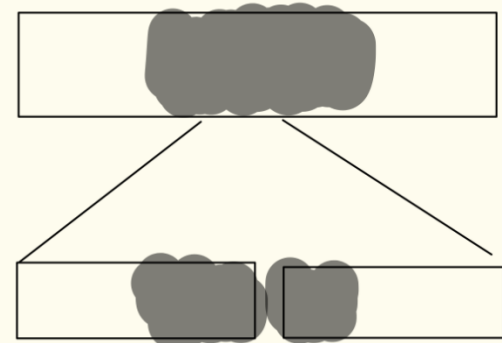
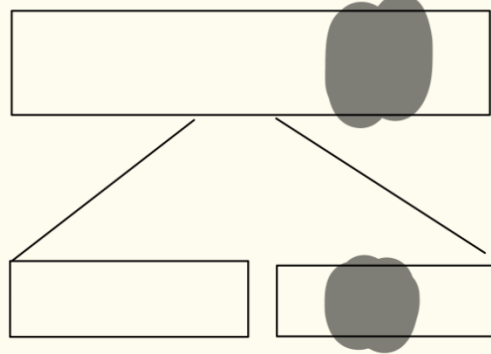
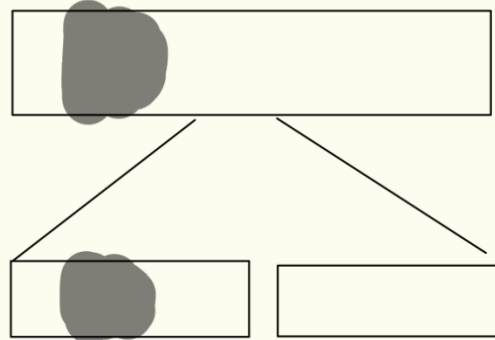
- ❖ 使用线段树，我们对每个节点要维护哪些信息呢？
- ❖ 显然，肯定要维护这个节点代表区间的最大连续子段和，而线段树上每个区间的信息是通过其左右两个子区间合并而来的，所以我们要考虑如何将两个区间信息合并

## 难点要点

❖ 对于一个区间来说，其最大子段和和其两个子区间可能有如下几种关系

1. 最大连续子段只存在于其中一个子区间，这样答案就是子区间的答案
2. 最大连续子段为左区间从最右开始的最大连续子段和右区间从最左开始的最大连续子段拼起来

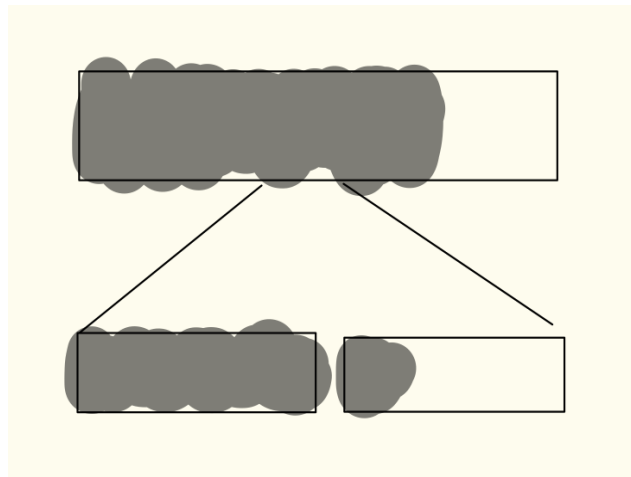
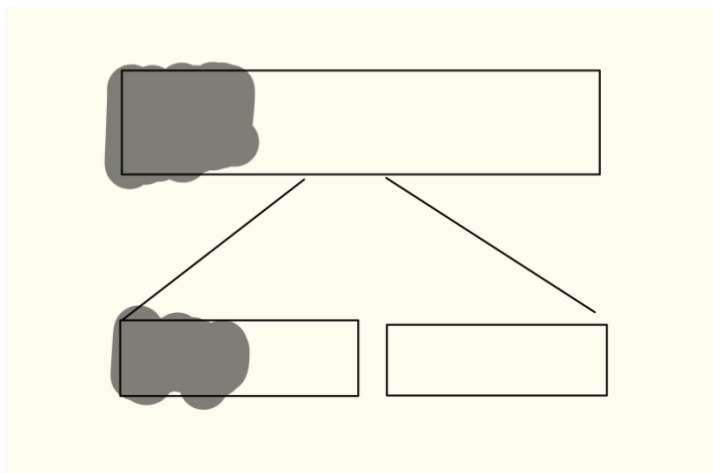
# 难点要点



## 难点要点

❖ 所以，我们需要维护的信息就出现了，即每个区间的最大子段和，从最左开始的最大连续子段和和从最右开始的最大连续子段和

❖ 怎么维护这两个值？



❖ 还需维护整个区间的和

❖ 叶节点很容易处理，然后依次向上合并即可。

## 难点要点

- ❖ 如果使用各种平衡树、那么思路也是类似的，只不过在合并信息的时候分成了左区间、节点本身和右区间三段，相比而言复杂了一点。
- ❖ 无论使用哪种数据结构，其时间复杂度都是 $O(n \log n)$ ，空间复杂度都是 $O(n)$ 。

## 小结

- ❖ 本题考查了基础数据结构的实现和在数据结构上维护信息的技巧
- ❖ 首先我们经过初步分析得出可以使用哪些数据结构
- ❖ 然后推导出为了维护题目所需的信息，需要维护哪些额外的信息
- ❖ 这样就解决了这个题目

**CST2018 3-2 Not Found**

**计76**

**王津中**

## 问题描述

❖ 给定一个01串 $S$ ，求01串 $T$ ，使得 $T$ 不为 $S$ 的子串，且长度最小。

❖ 如有多解，要求给出字典序最小的 $T$ 。

❖ 数据规模：

❖  $|S| \leq 2^{24}$

## 问题分析

❖ 不难设计出 $O(n^2)$ 的算法（想一想，怎么做？）

➤ 枚举 $T$ 的长度，逐一排除

❖ 不难发现，答案不会超过 $O(\log n)$ （想一想，为什么？）

➤ 鸽巢原理

❖ 由此不难设计出时间复杂度 $O(n \log n)$ 的算法

## 难点要点

❖ 如何优化时间复杂度？（想一想，时间都浪费在哪了？）

➤ 逆向思维！

❖ 如何优化空间？（想一想，布尔数组的空间开销？）

➤ bitset的实现

## 小结

- ❖ 在本题中，我们很容易得到一个相对较差的算法
- ❖ 考虑相对较差算法的本质，可从本质中发掘优化途径
- ❖ 基本的空间的优化技巧

# CST2018 3-3 Hacker

蒋林

## 问题描述

❖ 给定一个密码加盐后的CRC32值，判断这个密码是否是五位以内的密码或以前发现过的弱密码。

❖ 盐和密码都只由 “0123456789tsinghua” 中的字符组成。

❖ 两个操作：

➤ 0 x : 查询x这个密文对应的密码。

- 如果是五位以内的密码或以前发现过的弱密码则输出该密码。
- 如果找不到则输出 No，如果多个密码对应到该密文则输出 duplicate。

➤ 1 : 探知一个可能的弱口令（之前发现的7个密文的首字母）。

## 问题分析

- ❖ 使用哈希表存储CRC值和对应的密码：CRC32值是key，密码是value
- ❖ 最初将所有不超过五位的密码的CRC值求出来，并插入哈希表。
  - 共  $18^0 + 18^1 + 18^2 + 18^3 + 18^4 + 18^5 = 2,000,719$  个密码
- ❖ 每次发现弱口令时将发现的弱口令的CRC32值求出来，并插入哈希表。
  - 最多 2,000,000 次操作，最坏情况下每查询出一个密码就发现一次。
  - 最多能发现小于 1,000,000 个弱口令
- ❖ 哈希表中最多有大约3,000,000个词条

## 难点要点优化方法

- ❖ 时限3秒，写得慢都能过。
- ❖ 输入输出巨大，读入输出优化可节省很多时间。
- ❖ 密码最多7位，且只能由18个字符构成：
  - $19^7 < 2147483647$ ：可以用int存下
  - 用int代替string存储密码可以大幅加快速度，并且节省空间。
- ❖ 哈希表不宜开得太大，开得太大内存寻址时间会变长，反而会慢。

## 小结

- ❖ 这道题就是一道很普通的考察哈希表的题。
- ❖ 过掉这道题很简单，不过优化的空间很大。

**CST2018 3-4 kth**

**唐天映**

## 问题描述

给定三个数组  $a, b, c$ ，长度为  $n$

从三个数组中各取一个数  $a[x], b[y], c[z]$ ，一共  $n * n * n = n^3$  种取法

对取出来的数求和， $a[x] + b[y] + c[z]$ ，一共  $n^3$  个数（数值相同的和记多次）

请找出其中的第  $k$  小数的值

$n \leq 500000$ ， $k \leq 2000000$

交互库：给  $x_1, y_1, z_1, x_2, y_2, z_2$ ，返回  $a[x_1] + b[y_1] + c[z_1]$  是

否小于  $a[x_2] + b[y_2] + c[z_2]$ ，无次数限制

## 问题分析

穷举所有和， $O(n^3)$ ，不现实

简化一下问题

一、一个数组，交互库： $a[x1] < a[x2]$ ，找第k小

## 问题分析

穷举所有和， $O(n^3)$ ，不现实

简化一下问题

一、一个数组，交互库： $a[x1] < a[x2]$ ，找第k小

枚举所有和，有比较器就可以排序

快速排序、归并排序、堆排序.....

将所有数装进一个堆，每次删最小值删除k次

nth\_element

## 问题分析

二、两个数组，交互库： $a[x1]+b[y1]<a[x2]+b[y2]$ ，找第k小

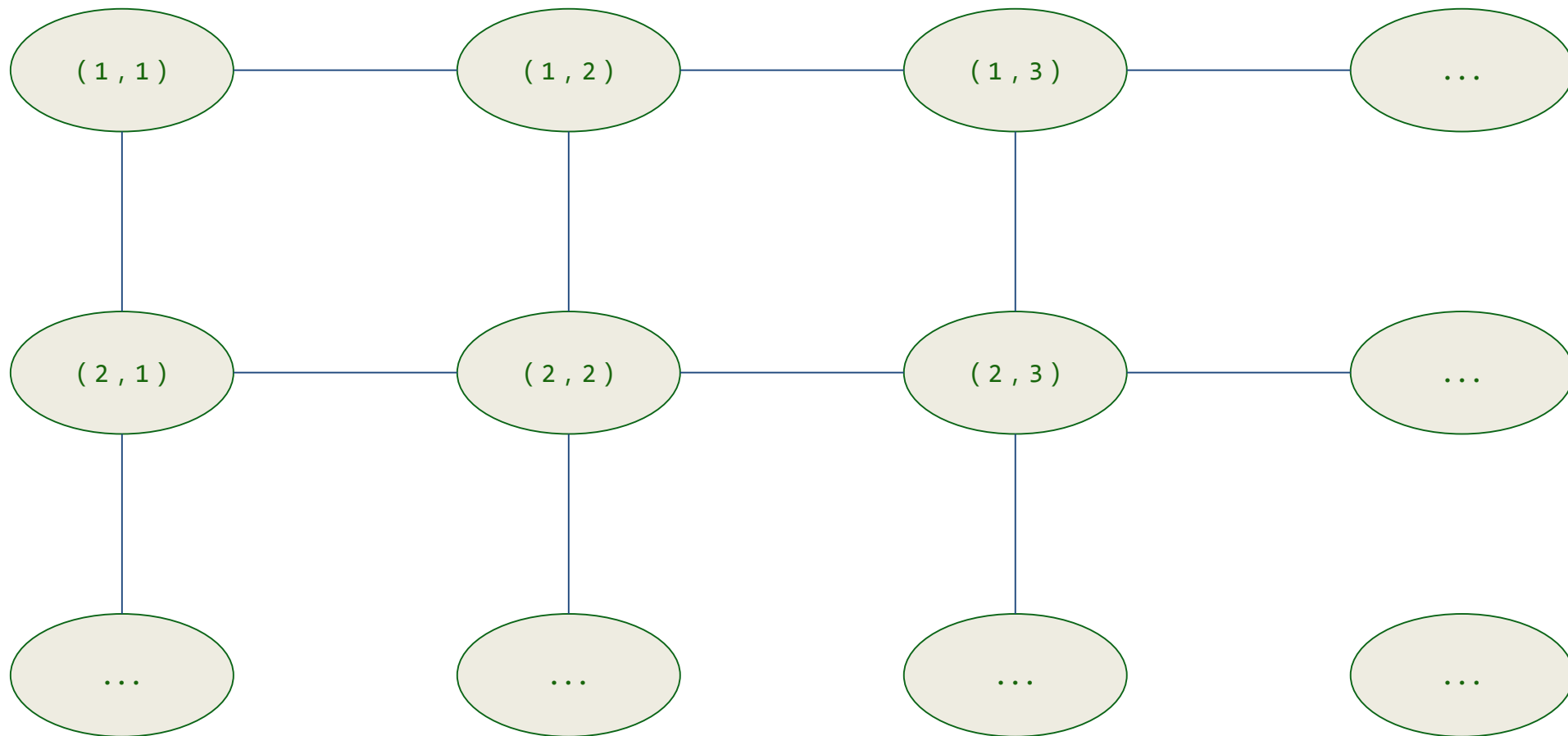
## 问题分析

二、两个数组，交互库： $a[x1]+b[y1]<a[x2]+b[y2]$ ，找第k小排序？

令 $x1=x2$ ，可以给b排序；令 $y1=y2$ ，可以给a排序

最小值： $a[1]+b[1]$

## 问题分析：排序之后



## 问题分析：排序之后

将所有数装进一个容器，每次删最小值，删除k次

所有数？

若 $x_1 \leq x_2$ ， $y_1 \leq y_2$ 且等号不同时成立，则 $(x_1, y_1)$ 比 $(x_2, y_2)$ 小

若 $(x_1, y_1)$ 没有被删除，则 $(x_2, y_2)$ 没有进入容器的必要

## 问题分析：排序之后

维护容器如下操作：

```
heap -> push(p(1,1))
```

```
while k:
```

```
    curans = p(x,y) = heap -> pop()
```

```
    push(p(x+1,y))
```

```
    push(p(x,y+1))
```

```
    k --
```

```
print(curans)
```

## 问题分析：排序之后

复杂度？

堆里最开始有1个元素

每次循环堆pop()一次push()两次，规模增大1

k次操作，规模为k+1

堆复杂度 $O(k \log k)$

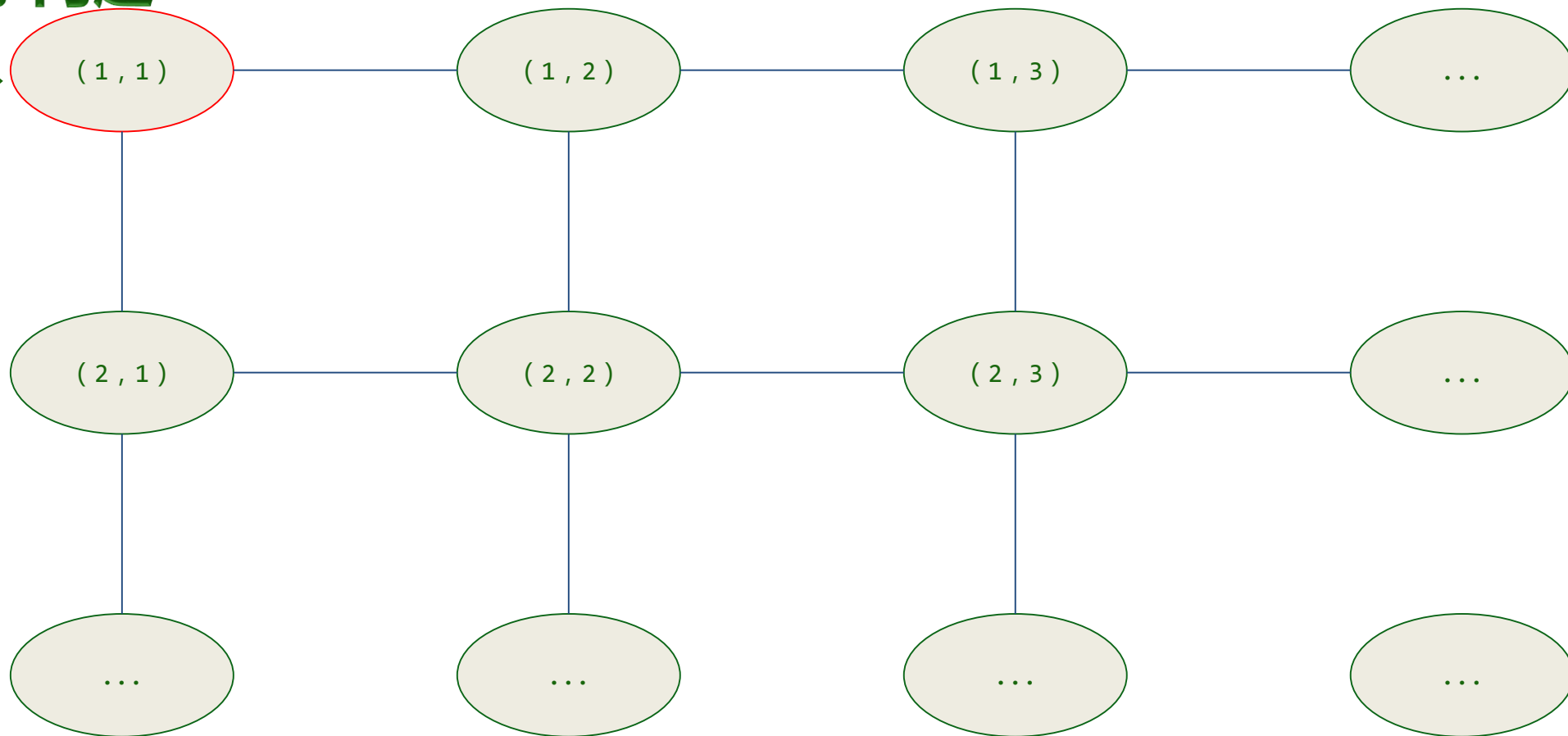
加上排序

总复杂度 $O(n \log n + k \log k)$ ，可以接受

# 难点要点

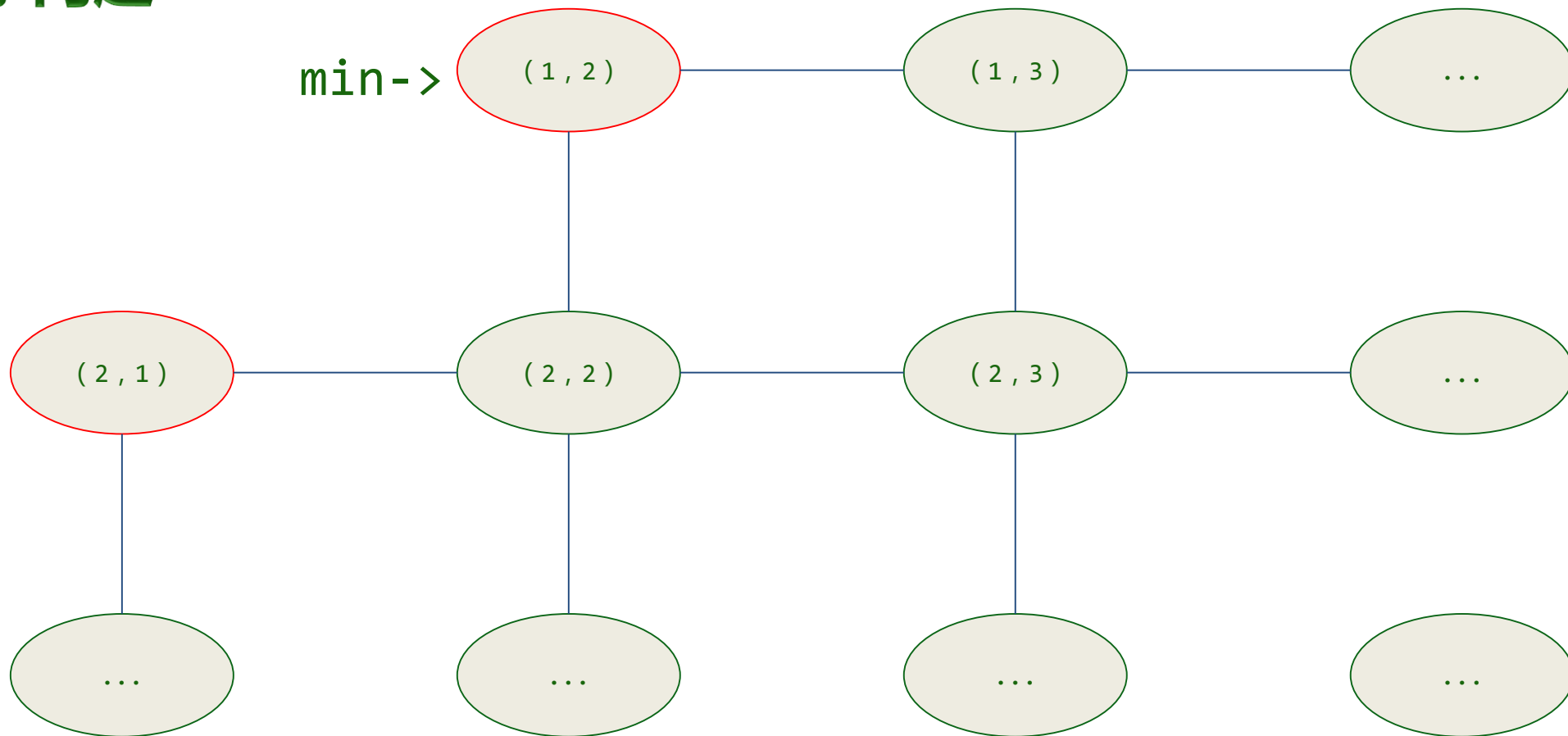
## 有点小问题

min->



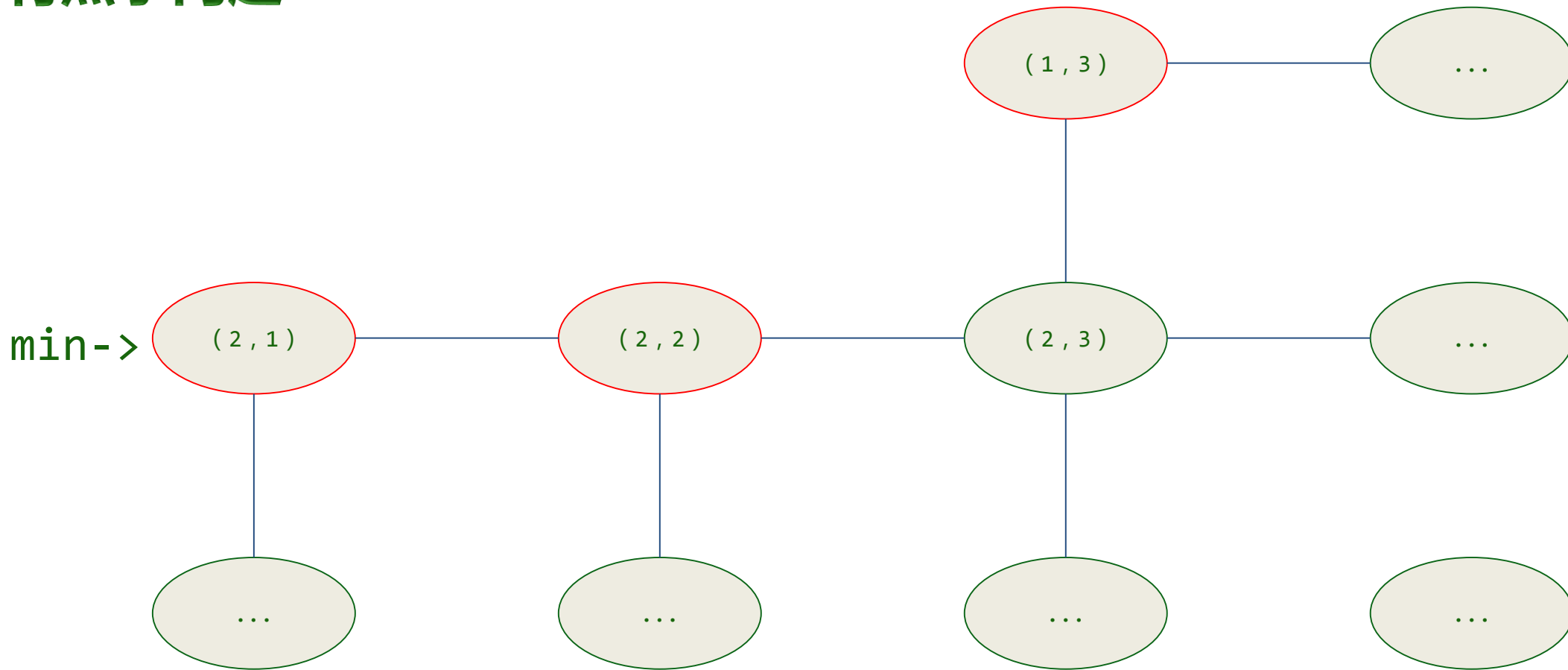
# 难点要点

## 有点小问题



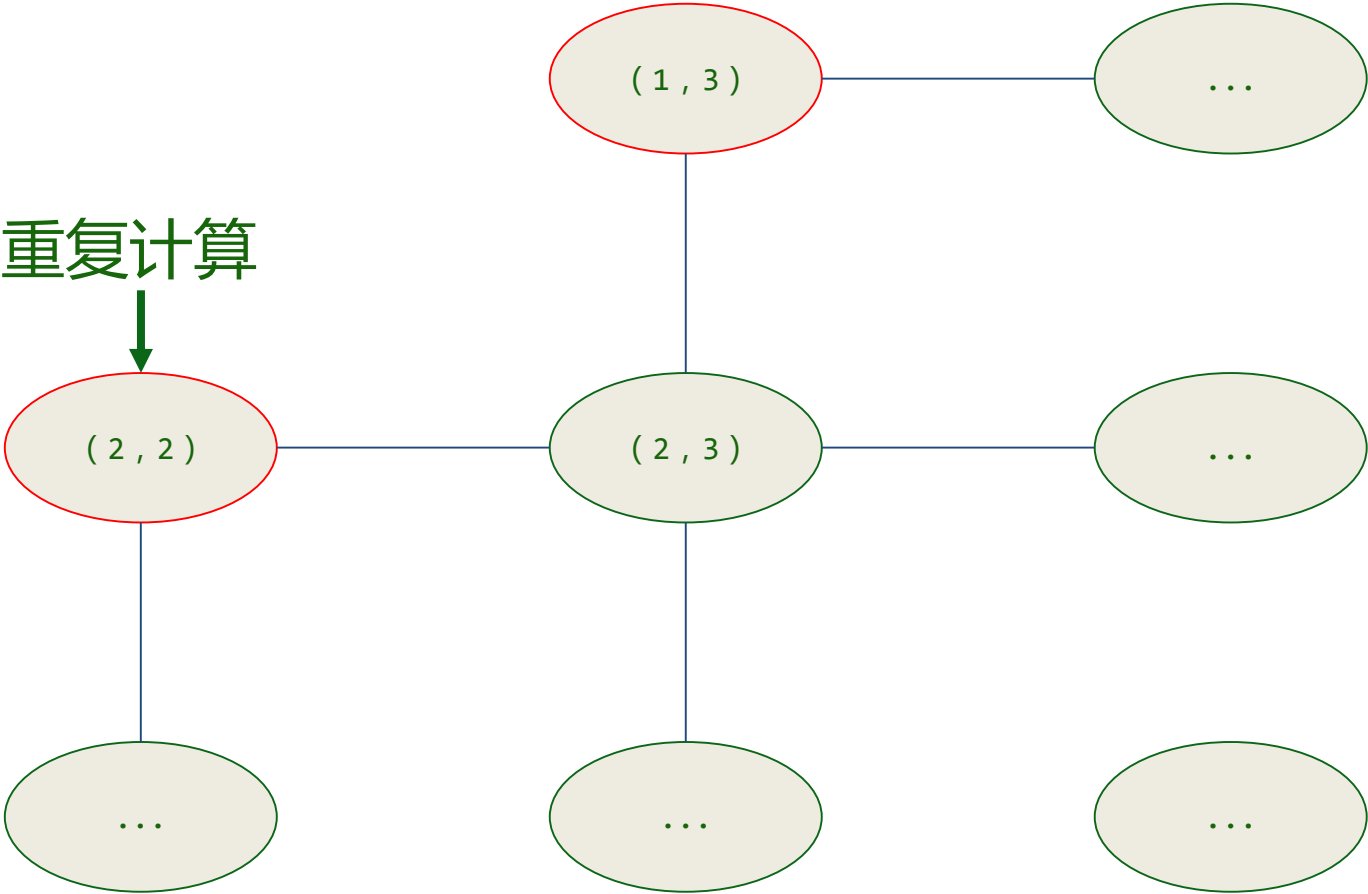
## 难点要点

## 有点小问题



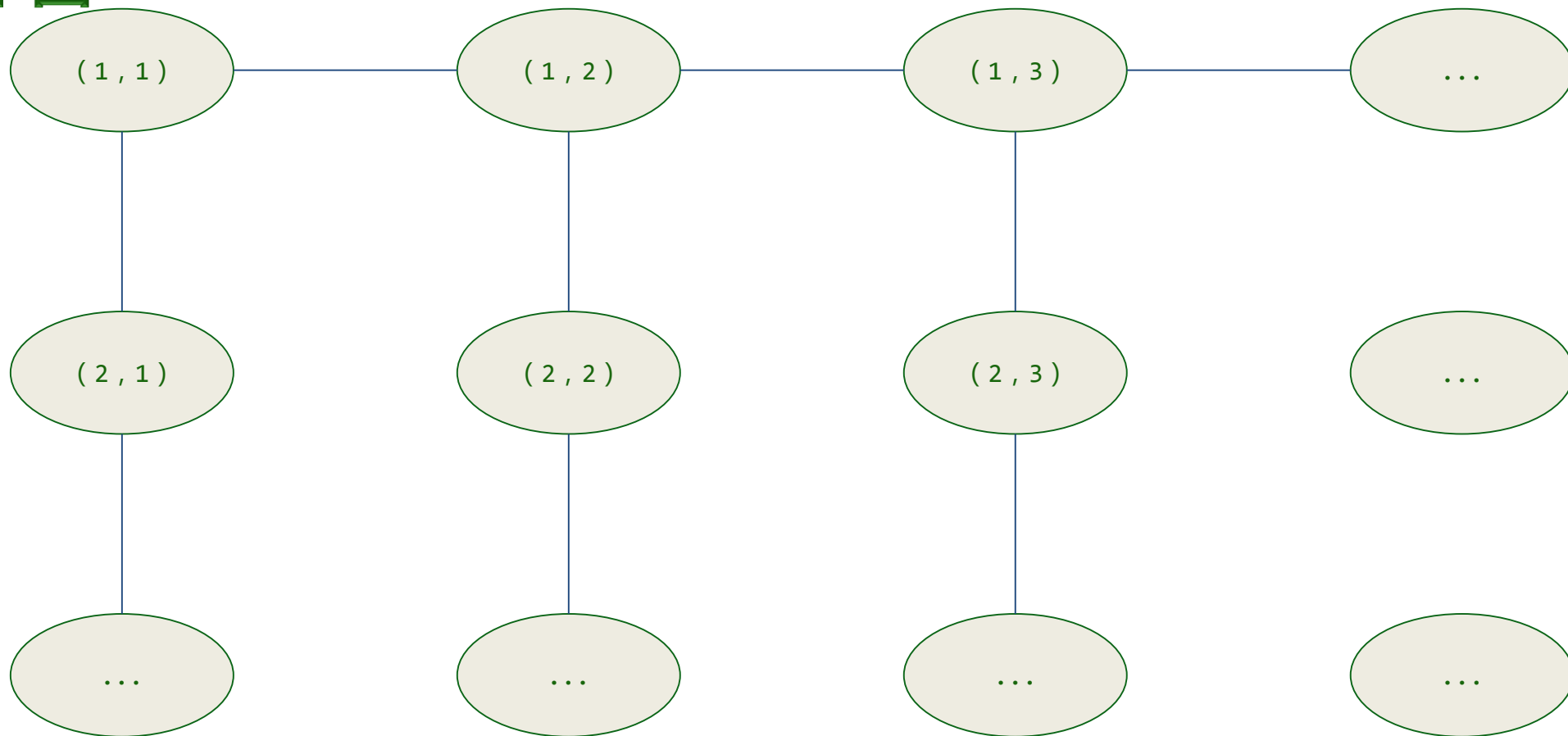
## 难点要点

## 有点小问题



# 难点要点

改一下图



## 难点要点

为什么可以删边？

每次拿出来的一定是最小值

## 难点要点

为什么可以删边？

每次拿出来的一定是最小值

拿最小值 $\Leftrightarrow$ 排序

顺序在图上，一定是一个拓扑序

归纳可得：在任意时刻，剩余最小值一定满足度为0

堆的作用就是在度为0的点里取剩余最小值

## 小结

三个数组可类比上述做法

多注意条件（ $k$ 的大小）

学会另一个角度的缩减问题（数组个数）

学会画图（拓扑图）和找它和问题的关系

祝大家PA3顺利！

# CST2018 3-5 Prefix

张子键

## 问题描述

❖ 给定一个字符串  $S[1..n]$ ，定义前缀  $\text{prefix}[i] = S[1..i]$ ，现在想知道  $\text{prefix}[i]$  在  $S$  中出现的次数总和，一次出现定义为

➤  $\text{prefix}[i] = S[j..j+i-1]$

❖ 比如对于字符串  $\text{aabaab}$ ， $\text{a}$  在字符串中出现 4 次， $\text{aa}$  出现 2 次，剩余的 3 个前缀各出现 1 次，答案为 11。

❖  $1 \leq N \leq 20000000$

# 问题分析

## ❖ 字符串匹配？

- 暴力匹配，哈希表

## ❖ 暴力匹配和哈希表的时间花费太大，并且哈希表还要避免冲突

## ❖ 提示？

- KMP的next数组

- $\text{next}[i]$  表示最长的前缀  $l$  ( $1 < i$ )，使得前缀  $l$  是前缀  $i$  的一个后缀

## 难点要点

- ❖ 如果对前缀  $x$ ，不断取  $\text{next}$ ，即  $x, \text{next}[x], \text{next}[\text{next}[x]] \dots$  直到得到  $0$ ，我们在这里称作  $x$  的  $\text{next}$  序列，那么根据  $\text{next}$  数组的定义，如果  $\text{prefix}[i] = S[x-i+1 \dots x]$ ，那么就等价于  $i$  在  $x$  的  $\text{next}$  序列里。那么很明显答案就是  $1 \sim n$  的  $\text{next}$  序列长度之和。
- ❖ 之后的计算就十分简单了，可以采用动态规划，具体的转移方程留给大家思考。

## 小结

❖ 这道题算一道比较容易的题目了，但是很巧妙。在理解了KMP的next数组的真正意义之后还是比较好处理的。Next数组在处理和前缀有关的问题是往往能够起到比较好的作用。

**CST2018 3-6 Match**

**计75**

**赵成钢**

## 问题描述

- ❖ 维护一个字符串序列。
- ❖ 支持插入字符、删除字符、区间翻转。
- ❖ 询问两个区间的子串是否相同。

### ❖ 数据规模：

- $0 \leq n, m \leq 400,000$
- 时间限制：4秒
- 空间限制：256MB

# 问题分析

## ❖ 字符串判重？

➤ Hash算法

## ❖ 区间操作：支持插入、删除、区间翻转

➤ 平衡树！

## 问题分析

❖ 用平衡树维护Hash值即可

❖ 平衡树的选择：

- Splay, AVL, etc...
- 无旋Treap

# HASH

❖ Hash(一个很长的东西) = 一个很短的东西

❖ 几种比较常见的Hash

➤ BKDR Hash

➤ SDBM Hash

➤ AP Hash

➤ FNV Hash

➤ ...

## BKDR HASH

❖ seed(种子) = 31,131,1313 etc..

➤ while(\*str) hash = (111 \* hash \* seed + (\*str++)) % mod;

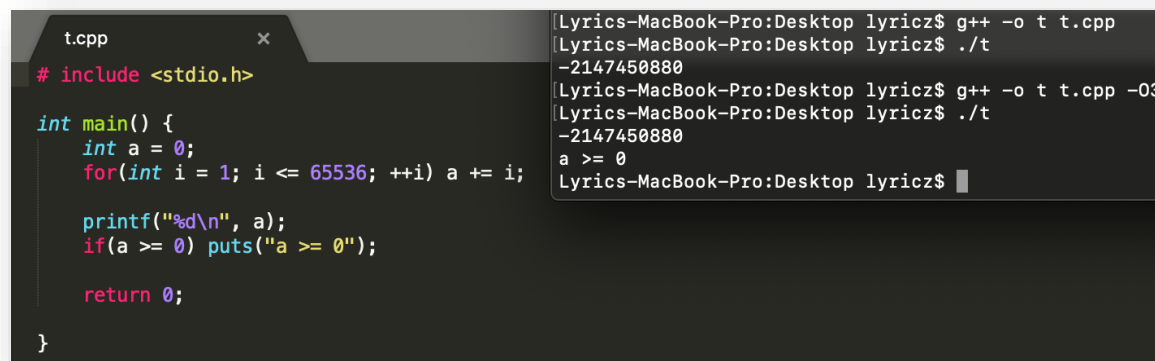
❖ unsigned int

➤ while(\*str) hash = hash \* seed + (\*str++);

➤ unsigned int自然溢出 (自动对 $2^{32}$ 取模)

➤ 为什么不能直接写int ?

➤ 缺点：可以构造数据



The image shows a code editor with a file named 't.cpp' and a terminal window. The code in 't.cpp' is as follows:

```
#include <stdio.h>

int main() {
    int a = 0;
    for(int i = 1; i <= 65536; ++i) a += i;

    printf("%d\n", a);
    if(a >= 0) puts("a >= 0");

    return 0;
}
```

The terminal output shows the following commands and results:

```
Lyrics-MacBook-Pro:Desktop lyricz$ g++ -o t t.cpp
Lyrics-MacBook-Pro:Desktop lyricz$ ./t
-2147450880
Lyrics-MacBook-Pro:Desktop lyricz$ g++ -o t t.cpp -O3
Lyrics-MacBook-Pro:Desktop lyricz$ ./t
-2147450880
a >= 0
Lyrics-MacBook-Pro:Desktop lyricz$
```

❖如何快速的知道一段区间的Hash值？

❖维护字符串前缀的Hash！

➤ `while(*str) hash = hash * seed + (*str++);`

➤可以理解成一个seed进制的数，在这里每一位是字符的ASCII码

➤abcd：维护a、ab、abc、abcd的Hash值

➤计算bc的Hash值？

▪  $\text{Hash}(\text{abc}) - \text{Hash}(\text{a}) * (\text{seed} ^ (\text{length}(\text{abc}) - \text{length}(\text{a})))$

▪  $33 = 233 - 2 * (10 ^ 2)$

## 平衡树

❖ 一个节点的子树是一个大的区间，维护这个区间的Hash即可

➤  $\text{Hash}(\text{parent}) =$

- $\text{Hash}(\text{lch}) * \text{seed} ^ (\text{length}(\text{rch}) + 1) +$  // 左子节点的Hash值
- $\text{Hash}(\text{char}(\text{parent})) * \text{seed} ^ \text{length}(\text{rch}) +$  // 这个字符的Hash值
- $\text{Hash}(\text{rch})$  // 右子节点的Hash值

# 平衡树

## ❖ 如何翻转？

### ➤ Lazy tag !

- 找到对应区间对应的子树
- 改变这个节点的信息，不对整个子树操作，打一个翻转标记
- 在用一个点的数据时首先释放标记

### ➤ 多维护一个倒序的Hash

### ➤ 交换左右子节点

# 无旋Treap

- ❖ 代码更短，更好理解的一种可以维护区间操作、可持久化的平衡树
- ❖ 同时满足堆的性质和平衡树的性质
- ❖ 如何保证平衡？
  - 每个节点随机一个修正值
  - 修正值的关系和堆一样
- ❖ 只用两种操作即可维护整个树
  - Merge：合并两颗树
  - Split：把一颗树劈成两颗

# 无旋Treap

❖ 如何插入、删除、询问？

❖ 插入：

- Split
- Merge \* 2

❖ 删除：

- Split \* 2
- Merge

❖ 询问：

- Split \* 2
- Merge \* 2

```
int merge(int x, int y) {
    if (x == 0 || y == 0) return x + y;
    ref(x), ref(y);
    if (pri[x] < pri[y]) {
        R[x] = merge(R[x], y);
        update(x); return x;
    } else {
        L[y] = merge(x, L[y]);
        update(y); return y;
    }
}

void split(int x, int k, int &l, int &r) {
    if (x == 0) {
        l = r = 0;
        return;
    }
    ref(x);
    if (siz[L[x]] < k) l = x, split(R[x], k - siz[L[x]] - 1, R[x], r);
    else r = x, split(L[x], k, l, L[x]);
    update(x);
    return;
}
```

## 难点要点

- ❖ Hash算法的理解
- ❖ 平衡树维护的理解
- ❖ 翻转标记
- ❖ 无旋Treap
- ❖ 代码相对繁琐

# 小结

## ❖ 分开看问题

- 如何判重？如何维护区间？
- 决定用什么算法和数据结构，并结合起来

## ❖ 如何维护平衡树？

## ❖ 另外一种数据结构：无旋Treap

- 优点：代码量相对其他平衡树少很多，比较好理解，可以可持久化
- 缺点：因为要不停Split和Merge，所以常数略大

## ❖ 谢谢大家！(｡•̀ㅂ•́)و✧