

# ADS 复习

---

## 00 目录

---

[00 目录](#)

### [01 AVL Trees](#)

平衡因子

插入

效果

证明

### [02 Splay Trees](#)

翻转

均摊分析

计算方法

### [03 Red-Black Trees](#)

调整

[Insert](#)

[Delete](#)

效果

### [04 B+ Trees](#)

操作

[Insert](#)

[Delete](#)

效果

### [05 Inverted File Index](#)

生成索引

[Word Stemming](#)

[Stop Words](#)

分布式索引

动态索引

索引压缩

查询阈值

针对文档

针对查询

评价指标

[Data Retrieval](#)

[Information Retrieval](#)

相关性评价

### [06 Leftist Heaps & Skew Heaps](#)

Merge

修改 Skew

效果

[Leftist](#)

[Skew](#)

### [07 Binomial Queue](#)

实现

效果

证明1

证明2

### [08 Backtracking](#)

八皇后

Turnpike

Tic-tac-toe

剪枝

α

β

## 09 Divide & Conquer

时间复杂性分析

替代法

递归树法

Master 定理

形式1

形式2

形式3

## 10 DP

矩阵相乘

最优二叉查找树

最短路径

产品组装

## 11 Greedy

任务选取问题

Huffman Code

证明

## 12 NPC

计算模型

递归函数

λ演算

图灵机

可计算性

一张宝图

规约

可满足问题

其他NPC

一些坑

形式语言

映射

一个记号

## 13 Approximate Algorithms

近似比

多项式时间近似算法

案例

装箱问题

背包问题

K-center

## 14 Local Search

顶点覆盖

Hopfiled 神经网络

最大割集 The Maximum Cut Problem

效果

改进

Big-improvement-flip

K-L heuristic

## 15 随机算法

雇人问题

快排

## 16 并行算法

并行模型

PRAM

# 01 AVL Trees

动机：加速动态查找，降低树高度

如何定义平衡：子树平衡，并且 $|h_L - h_R| \leq 1$

空树的高度被定义为-1

## 平衡因子

$$BF(node) = h_L - h_R$$

AVL树中，BF应在-1, 0, 1

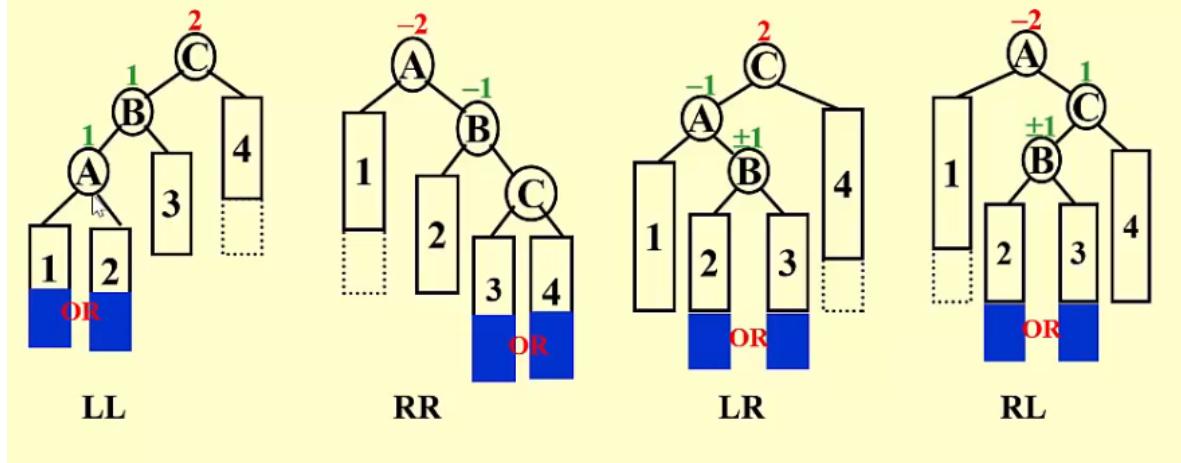
## 插入

先插入，后调整

从被破坏者到破坏者，三层中进行调整，也就是选取离破坏者最近的被破坏者

trouble maker定位

### 四种模式以及调整目标



RR、LL 单旋

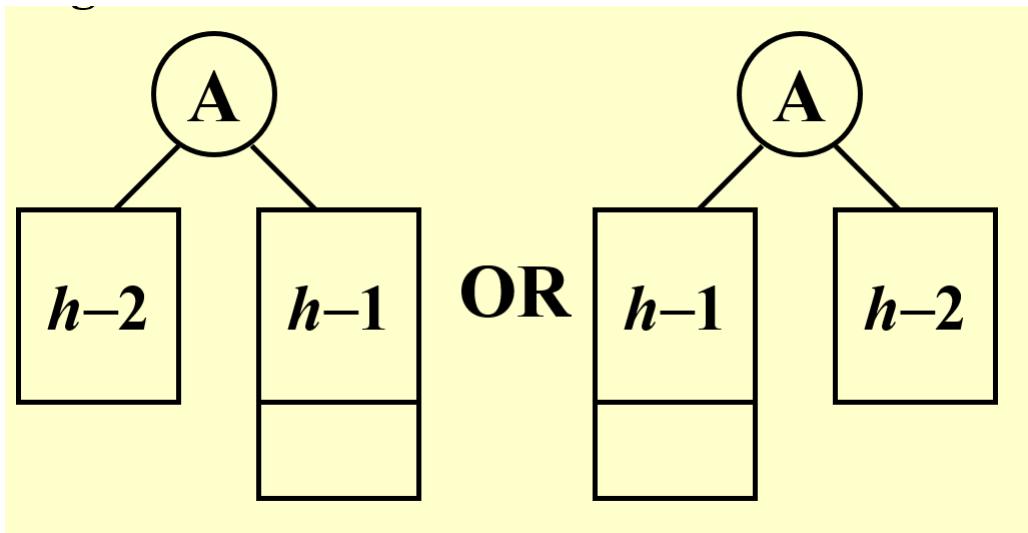
LR、RL 双旋 (直接3+4模式)

调整完后需要刷新所有节点的BF，或者直接维护每个节点的高度属性

## 效果

$h = O(\log n)$

证明



最坏情况,  $n_h = n_{h-1} + n_{h-2} + 1$

fib数列, 容易知道 $h$ 为 $\log n$ 级别

## 02 Splay Trees

动机: 利用时间局部性, 使均摊复杂度达到 $\log N$ , 此时不再利用平衡性, 所以splay树不是平衡树

Any  $M$  consecutive tree operations starting from an empty tree take at most  $O(M \log N)$  time.

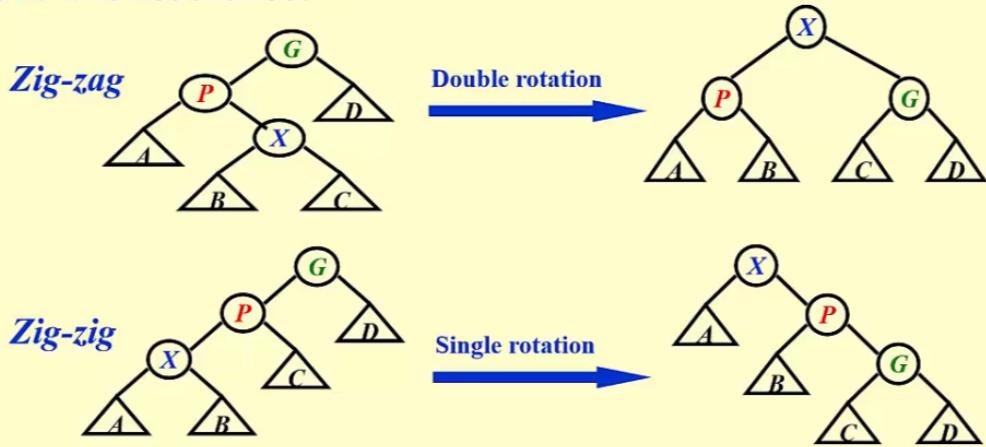
此时不可能达到任何一次操作都是 $\log n$ 复杂度

翻转

思路: 将每次访问的结点翻到树根 node  $X$ : parent  $P$ , grandparent  $G$

Case 1:  $P$  is the root  $\rightarrow$  Rotate  $X$  and  $P$

Case 2:  $P$  is not the root



一步一步地翻转到根节点, 其中Case1只可能在最后一步做

均摊分析

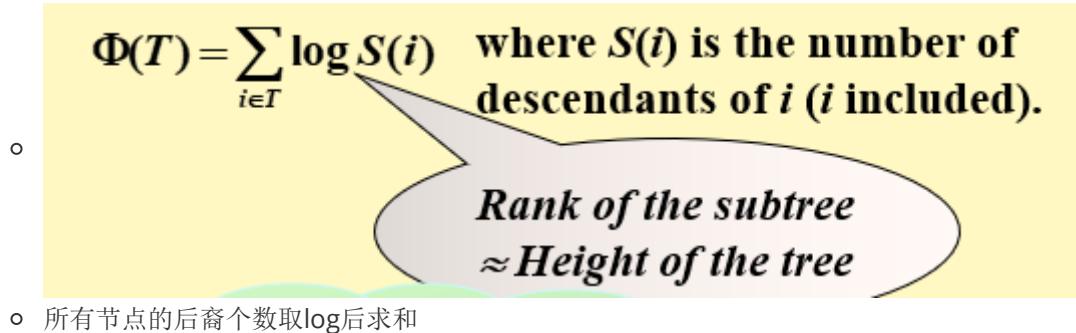
均摊时间的定义：

任意M次

- 不是平均时间，平均时间等价于M趋向于无穷大的情况下的均摊时间

## 计算方法

- 直接计算
- 记账法 Accounting method
- 势能法 Potential method



- 所有节点的后裔个数取 $\log$ 后求和

## 03 Red-Black Trees

动机：加速动态查找，并且降低结构的变化（AVL  $\log N$ ，红黑树  $O(1)$ ）

约束：不能双红、任何分支黑节点个数一样

本质：任何两个分支长度差不超1倍

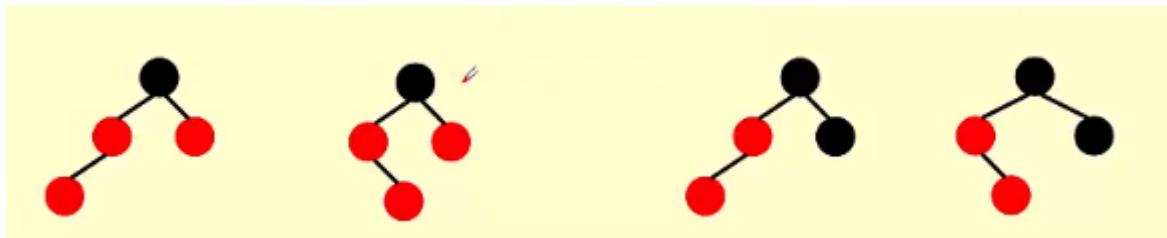
## 调整

- 改颜色
- 调结构

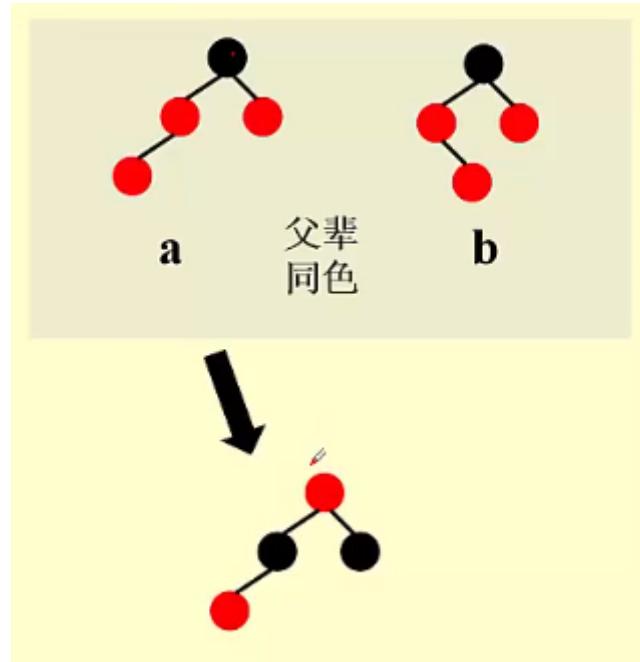
## Insert

先插入红色，然后调整双红

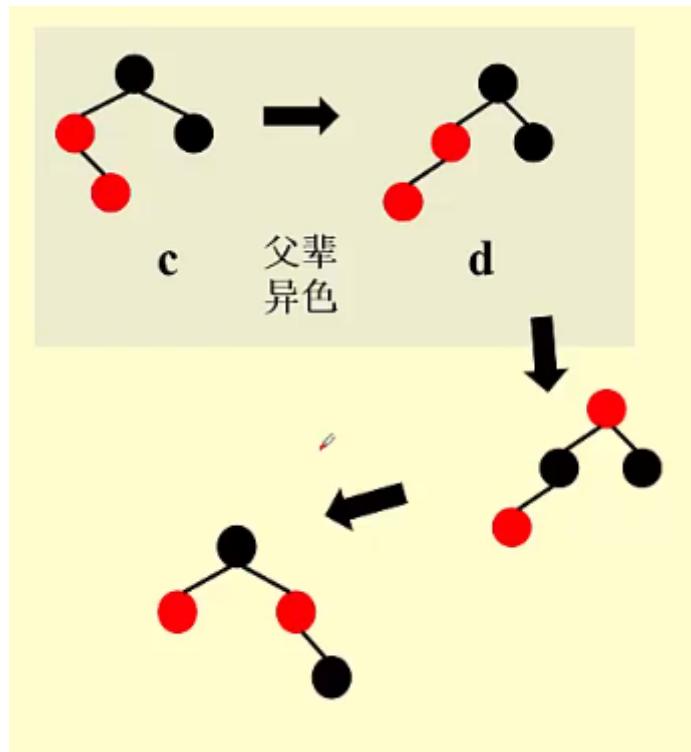
总共四种情况



第一类，仅调色，但是可能递归，也就是顶部会进一步触发双红



第二类，改结构，调颜色



## Delete

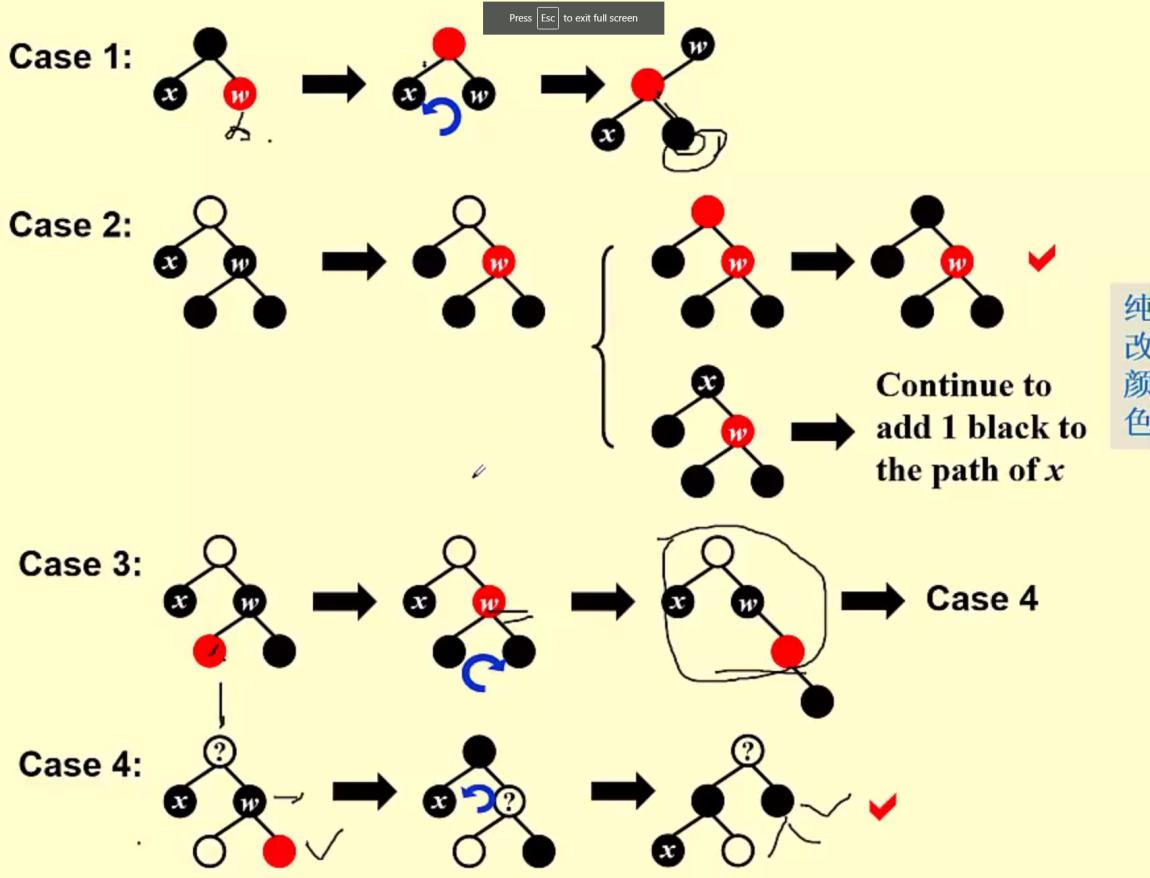
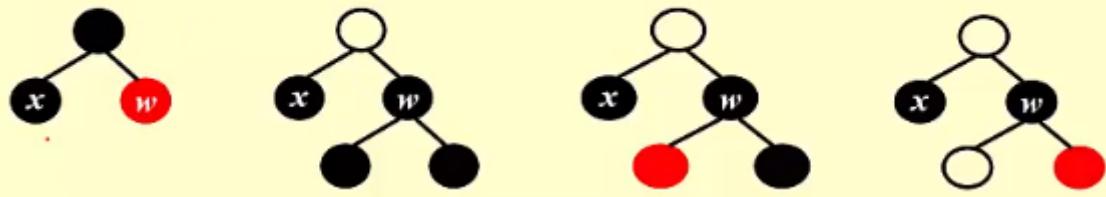
先删除，再调整

从那个被删的点向下看，左子树选个最大的换上，或者右子树找个最小的

红色无所谓，黑色要调整，即在被删的分支上加一个黑节点

范围：上下3代（上、下、自己）5个节点

➤ 归类处理：按中间层（兄弟）分4种情况：



case1会转化为下面3种的某一种

## 效果

**black-height  $bh(x)$ :** the number of **black** nodes from x to a leaf.

**【Lemma】** A red-black tree with  $N$  internal nodes has height at most  $2\ln(N+1)$ .

**Proof:**

① For any node  $x$ ,  $\text{sizeof}(x) \geq 2^{bh(x)} - 1$ .

Prove by induction on  $h(x)$  : height of tree

②  $bh(Tree) \geq h(Tree) / 2$  ?

$$\text{Sizeof}(root) = N \geq 2^{bh(Tree)} - 1 \geq 2^{h/2} - 1$$

证明：每个节后裔个数都大于 $2^{\text{黑高度}} - 1$

黑高度至少占总高度的一半

完事

## 04 B+ Trees

动机：计算机的层次储存结构

要求：高度相等，宽度 $M/2 \sim M$

叶节点存放数据，非叶节点作为索引

除了第一个block，每个block的首个元素都会作为索引

### 操作

#### Insert

先插入，再调整

超出M，分裂

存在递归（即造成上一层的上溢）

#### Delete

少于 $M/2$ ，借，或者合并

存在递归，即造成上一层下溢

### 效果

- 树高： $\text{Depth}(M, N) = O(\lceil \log_{[M/2]} N \rceil)$ 
  - 每一层都是最少节点数
- 插入成本： $T_{Insert}(M, N) = O(M \log_M N) = O((M / \log M) \log N)$ 
  - 底数是M，忽略掉常数部分，每层插入需要操作M次（数组），以=一共 $\log M(N)$ 层
- 查找成本： $T_{Find}(M, N) = O(\log M \log_M N) = O(\log N)$ 
  - 每层查找使用二分需要操作 $\log M$ 次（数组），一共 $\log M(N)$ 层

注意到插入前面有一个常数 $M/\log M$ ，因此为了降低常数，最好的选择是 $M=3, 4$

当然，实际上因为block是对应磁盘的block的，所以M都是100数量级的

## 05 Inverted File Index

倒排文件索引

正向索引：文本中包含哪些单词

倒排索引：单词出现在哪些文本里

| Doc | Text   |
|-----|--|
| 1   | Gold silver truck                            |
| 2   | Shipment of gold damaged in a fire           |
| 3   | Delivery of silver arrived in a silver truck |
| 4   | Shipment of gold arrived in a truck          |



| No. | Term     | Times; Documents Words |
|-----|----------|------------------------|
| 1   | a        | <3; (2;6),(3;6),(4;6)> |
| 2   | arrived  | <2; (3;4),(4;4)>       |
| 3   | damaged  | <1; (2;4)>             |
| 4   | delivery | <1; (3;1)>             |
| 5   | fire     | <1; (2;7)>             |
| 6   | gold     | <3; (1;1),(2;3),(4;3)> |
| 7   | of       | <3; (2;2),(3;2),(4;2)> |
| 8   | in       | <3; (2;5),(3;5),(4;5)> |
| 9   | shipment | <2; (2;1),(4;1)>       |
| 10  | silver   | <2; (1;2),(3;3,7)>     |
| 11  | truck    | <3; (1;3),(3;8),(4;7)> |

Term Dictionary      Posting List

## 生成索引

### Index Generator

```

while ( read a document D ) {
    while ( read a term T in D ) {
        if ( Find( Dictionary, T ) == false )
            Insert( Dictionary, T );
        Get T's posting list;
        Insert a node to T's posting list;
    }
}
Write the inverted index to disk;

```

Token Analyzer  
Stop Filter

Vocabulary  
Scanner

Vocabulary  
Insertor

Memory management

### Word Stemming

将词语预处理来保留主要枝干（提炼词语）

|                  |   |                                 |
|------------------|---|---------------------------------|
| <p>【Example】</p> | <p>Process<br/>processing<br/>processes<br/>processed</p> | <p>says<br/>said<br/>saying</p> |
|                  | <p>process</p>  | <p>say</p>                      |

### Stop Words

烂大街的词语，“的地得”，a the it，被称为stop words

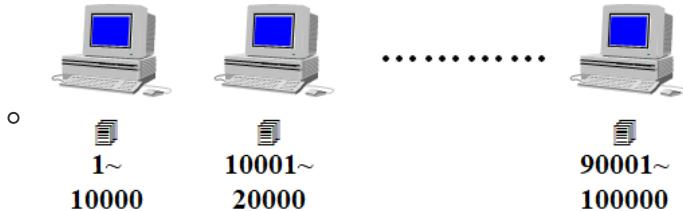
索引时这些词要被删掉

## 分布式索引

- 按照词语分布

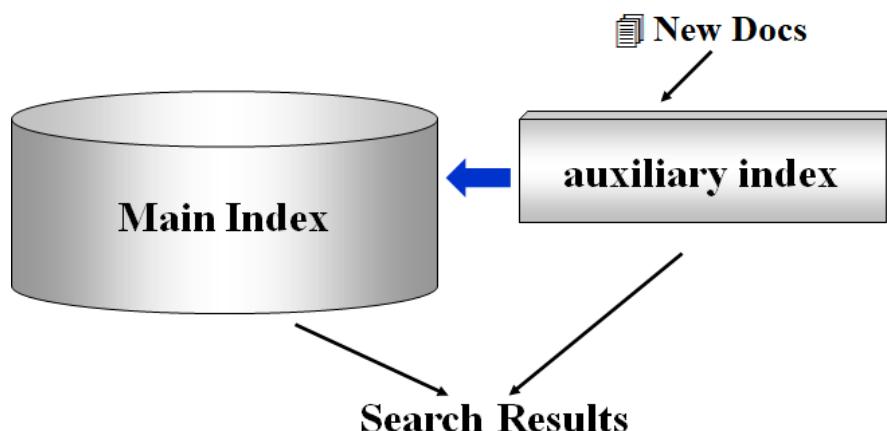


- 按照文章分布



## 动态索引

- 动机：文章随时会增删，同时导致术语也可能增删



增加一个辅助索引（类似buffer?）

## 索引压缩

Posting List

数组存地址的话，会超int

压缩，直接存gap（两个数之间的差）

## 查询阈值

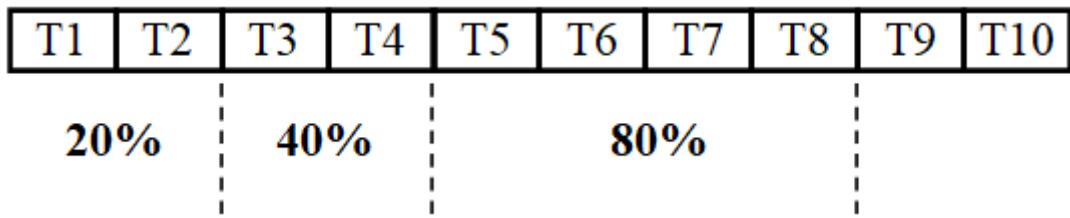
## 针对文档

取回的文档打个分，然后按照权重排序，按照threshold取前x个

缺点就是会剪掉一些相关的文档，而且对于布尔查询不合适

## 针对查询

把查询的词按照出现频率升序排列，然后按照threshold，只取前几个词进行查询



其目的就是，很多次太大众化了，比如build web with react，前三个词就比较频繁，取回的文章没有针对性，这个时候先取出相对小众的react去查，就可以找到更相关的信息

## 评价指标

### Data Retrieval

数据方面

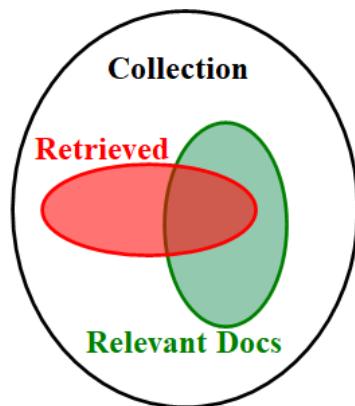
- 响应速度
- 索引所占空间（越小越好）

### Information Retrieval

- 相关性

相关性评价

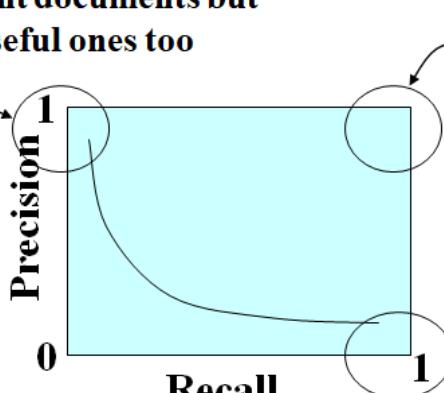
|               | Relevant | Irrelevant |
|---------------|----------|------------|
| Retrieved     | $R_R$    | $I_R$      |
| Not Retrieved | $R_N$    | $I_N$      |



$$\text{Precision } P = R_R / (R_R + I_R)$$

$$\text{Recall } R = R_R / (R_R + R_N)$$

Returns relevant documents but misses many useful ones too



The ideal

Returns most relevant documents but includes lot of junk

# 06 Leftist Heaps & Skew Heaps

动机：加速合并（严格的二叉堆合并成本过高）

破坏结构要求，加速合并操作

**NPL**: 某个节点到没有两个儿子的节点的最短路径（到海边的最短距离）

与之对应的，堆的高度是到海边的最远距离

定义：**左子树NPL值大于右子树NPL**，也就是左倾

## Merge

递归，先把新的root确定，然后左子树不动，右子树与另一棵树合并后挂到root右边，依次递归

```
PriorityQueue Merge ( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1 == NULL )  return H2;
    if ( H2 == NULL )  return H1;
    if ( H1->Element < H2->Element )  return Merge1( H1, H2 );
    else return Merge1( H2, H1 );
}
```

```
static PriorityQueue
Merge1( PriorityQueue H1, PriorityQueue H2 )
{
    if ( H1->Left == NULL )  /* single node */
        H1->Left = H2;      /* H1->Right is already NULL
                                and H1->Npl is already 0 */
    else {
        H1->Right = Merge( H1->Right, H2 );  /* Step 1 & 2 */
        if ( H1->Left->Npl < H1->Right->Npl )
            SwapChildren( H1 );                  /* Step 3 */
        H1->Npl = H1->Right->Npl + 1;
    } /* end else */
    return H1;
}                                          $T_p = O(\log N)$ 
```

合并完了之后有可能破坏了左倾特性，此时需要对换两个子树

最后注意NPL的更新

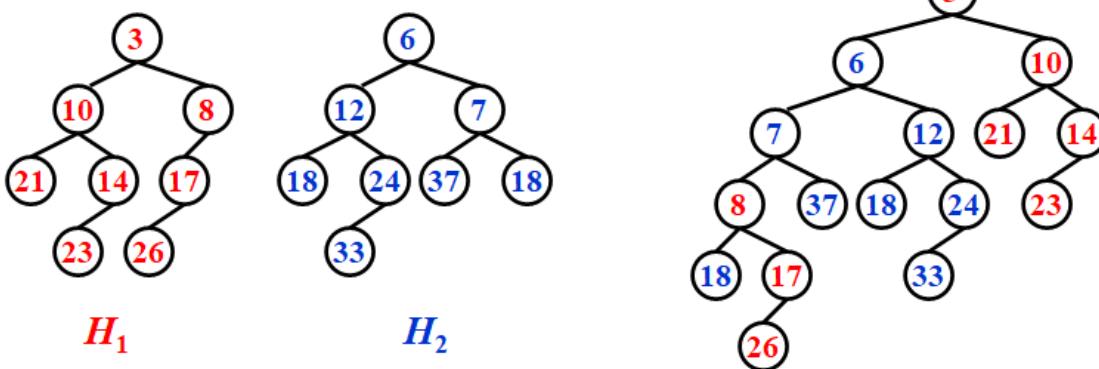
## 修改 Skew

还是先定root，然后root的左子树放到右子树，原来的右子树与另一棵树合并挂到左边

也就是说：每次不比较**NPL**，总是进行交换

好处：不用比较，也不用维护NPL值了

## ☞ Merge (iterative version):



效果

## Leftist

定理：一棵右边路径有 $r$ 个节点的左倾树，至少一共有 $2^r - 1$ 个节点

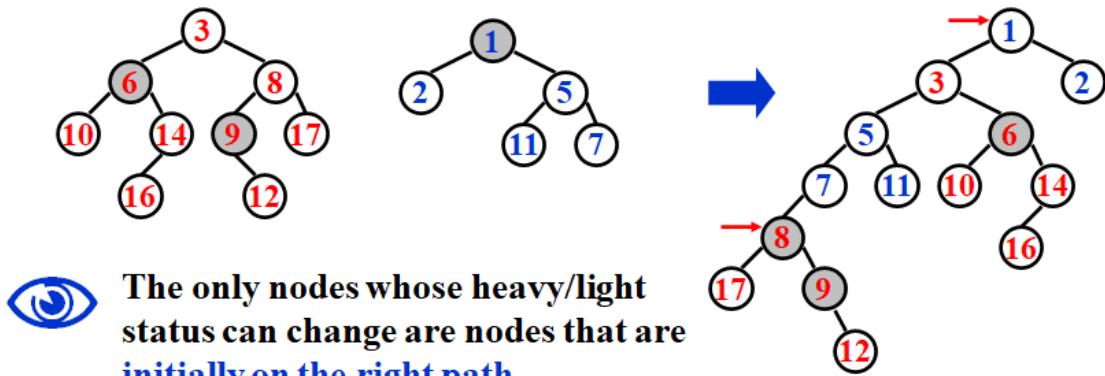
还算比较好想象，最极端就是一个完全二叉树了

由此可以容易推出右路径最多 $\lfloor \log(N + 1) \rfloor$ 个

然后因为操作（merge, insert, delete，其中后两者本质都是merge）都是在右边进行的，所以就知道复杂度压在了 $O(\log N)$

## Skew

重节点：右边的节点个数至少占了一半，也就是出现右倾趋势



观察：merge时，只有右边路径上的节点会发生属性改变，因为所有操作都是在这条路径上进行的。

势能函数：树中重节点个数

考察编号为 $i$ 的节点的右边路径，上面存在 $l_i$ 个轻节点， $h_i$ 个重节点

因为操作都发生在右边，所以最坏复杂度，也就是最多操作次数就是两个堆的右边路径节点数之和，如下图的 $T_{worst}$ 所示：

$$H_i : l_i + h_i \quad (i = 1, 2) \quad \rightarrow \quad T_{worst} = l_1 + h_1 + l_2 + h_2$$

Along the right path

合并前，势能函数=两个堆的右边路径上的重节点的个数之和+其他可能存在的重节点的个数。

$$\text{Before merge: } \Phi_0 = h_1 + h_2 + h$$

此时观察，因为每一次合并操作中，都是要翻转的，并且都是用右边的子树去跟别人合并，那么对于一个重节点，本身右子树就比左子树大，再一合并一定更大，这个时候进行翻转，就一定得到一个轻节点。

而与之相对的，是左子树翻转之后，其类型不再确定。

也就是说，新增的重节点只可能来源于原来的右边路径上的轻节点。

因此合并后，势能函数最多等于原来右边节点上的轻节点个数+原来的其他的重节点个数（这些节点在merge时没有被动过）。

$$\text{Before merge: } \Phi_0 = h_1 + h_2 + h$$

$$\text{After merge: } \Phi_N \leq l_1 + l_2 + h$$

这个时候可以来看看均摊复杂度了。

假设最开始就是最糟糕的情况，那么经过合并之后，一次合并的均摊时间复杂度必然小于两倍的原有右边路径上轻节点的个数和。

$$\begin{aligned} T_{\text{amortized}} &= T_{\text{worst}} + \Phi_N - \Phi_0 \\ &\leq 2(l_1 + l_2) \end{aligned}$$

然后因为右侧路径上的轻节点个数是 $\log N$ 级别的（轻节点都是要求右边比左边重的，用递推可以证明整棵树上的轻节点都是 $\log N$ 级别），就可以得到总的均摊复杂度也是 $\log N$ 级。

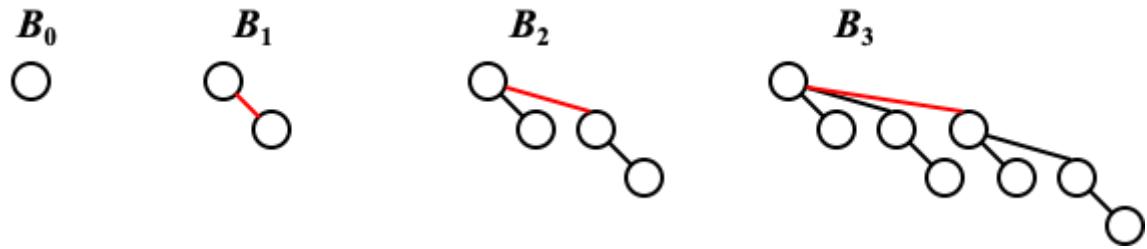
$$l = O(\log N) \rightarrow T_{\text{amortized}} = O(\log N)$$

完事。

## 07 Binomial Queue

动机：加速优先队列的合并

思路：借鉴二进制加法



**查找最小 (top):** 在每棵树的根节点中找最小的

$O(\log N)$ , 记忆化最小值之后可以降到  $O(1)$

**Merge:** 二进制加法, 从最低位开始, 取小的作为树根, 然后合并, 进位

$O(\log N)$ , 与进位次数一样

**Insert:** 合并单节点和一棵树

**Delete:** 找最小, 去掉树根, 新的森林和原来剩下的森林合并

$O(\log N)$

**Step 1: FindMin in  $B_k$**   
 $\text{/* O}(\log N) \text{ */}$

**Step 2: Remove  $B_k$  from  $H$**   
 $\text{/* O}(1) \text{ */}$

**Step 3: Remove root from  $B_k$**   
 $\text{/* O}(\log N) \text{ */}$

**Step 4: Merge ( $H, H'$ )**  
 $\text{/* O}(\log N) \text{ */}$

## 实现

---

左儿子右兄弟

```

typedef struct BinNode *Position;
typedef struct Collection *BinQueue;
typedef struct BinNode *BinTree; /* missing from p.176 */

struct BinNode
{
    ElementType    Element;
    Position       LeftChild;
    Position       NextSibling;
};

struct Collection
{
    int            CurrentSize; /* total number of nodes */
    BinTree        TheTrees[ MaxTrees ];
};

```

```

BinQueue Merge( BinQueue H1, BinQueue H2 )
{   BinTree T1, T2, Carry = NULL;
    int i, j;
    if ( H1->CurrentSize + H2-> CurrentSize > Capacity ) ErrorMessage();
    H1->CurrentSize += H2-> CurrentSize;
    for ( i=0, j=1; j<= H1->CurrentSize; i++, j*=2 ) {
        T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i]; /*current trees */
        switch( 4*!!Carry + 2*!!T2 + !!T1 ) {
            case 0: /* 000 */
            case 1: /* 001 */ break;
            case 2: /* 010 */ H1->TheTrees[i] = T2; H2->TheTrees[i] = NULL; break;
            case 4: /* 100 */ H1->TheTrees[i] = Carry; Carry = NULL; break;
            case 3: /* 011 */ Carry = CombineTrees( T1, T2 );
                      H1->TheTrees[i] = H2->TheTrees[i] = NULL; break;
            case 5: /* 101 */ Carry = CombineTrees( T1, Carry );
                      H1->TheTrees[i] = NULL; break;
            case 6: /* 110 */ Carry = CombineTrees( T2, Carry );
                      H2->TheTrees[i] = NULL; break;
            case 7: /* 111 */ H1->TheTrees[i] = Carry;
                      Carry = CombineTrees( T1, T2 );
                      H2->TheTrees[i] = NULL; break;
        } /* end switch */
    } /* end for-loop */
    return H1;
}

```

```

ElementType DeleteMin( BinQueue H )
{   BinQueue DeletedQueue;
    Position DeletedTree, OldRoot;
    ElementType MinItem = Infinity; /* the minimum item to be returned */
    int i, j, MinTree; /* MinTree is the index of the tree with the minimum item */

    if (IsEmpty(H)) { PrintErrorMessage(); return -Infinity; }

    for (i = 0; i < MaxTrees; i++) { /* Step 1: find the minimum item */
        if (H->TheTrees[i] && H->TheTrees[i]->Element < MinItem) {
            MinItem = H->TheTrees[i]->Element; MinTree = i; } /* end if */
    } /* end for-i-loop */
    DeletedTree = H->TheTrees[MinTree];
    H->TheTrees[MinTree] = NULL; /* Step 2: remove the MinTree from H => H' */
    OldRoot = DeletedTree; /* Step 3.1: remove the root */
    DeletedTree = DeletedTree->LeftChild; free(OldRoot);
    DeletedQueue = Initialize(); /* Step 3.2: create H'' */
    DeletedQueue->CurrentSize = (1 << MinTree) - 1; /* 2^MinTree - 1 */
    for (j = MinTree - 1; j >= 0; j--) {
        DeletedQueue->TheTrees[j] = DeletedTree;
        DeletedTree = DeletedTree->NextSibling;
        DeletedQueue->TheTrees[j]->NextSibling = NULL;
    } /* end for-j-loop */
    H->CurrentSize -= DeletedQueue->CurrentSize + 1;
    H = Merge(H, DeletedQueue); /* Step 4: merge H' and H'' */
    return MinItem;
}

```

## 效果

连续插入N个， $O(N)$

### 证明1

$$N + N \left( \frac{1}{4} + 2 \times \frac{1}{8} + 3 \times \frac{1}{16} + \dots \right) = O(N)$$

合并成本+进位成本

### 证明2

势能函数：树的个数

每次的成本：1+进位次数

$$\text{有 } c_i + (\Phi_i - \Phi_{i-1}) = 2$$

注意到每次进位都会导致树的个数减少一个，因此每次树的个数的减少等于进位次数-1，需要减一是因为新插入的节点一开始会引入一棵树。用  $c_i$  替换掉进位次数的话，可以直到树的个数的减少量等于  $2 - c_i$ 。

累加得到

$$\begin{aligned} \sum_{i=1}^N c_i + \Phi_N - \Phi_0 &= 2N \\ \sum_{i=1}^N c_i &= 2N - \Phi_N \leq 2N = O(N) \end{aligned}$$

由此得到插入操作的分摊复杂度为  $O(1)$

$T_{worst} = O(\log N)$ , but  $T_{amortized} = 2$

## 08 Backtracking

### 八皇后

经典问题，略

### Turnpike

根据给定的  $N(N - 1)/2$  个距离重建出  $N$  个点的坐标 ( $x_1 = 0$ )

每次按照最大的距离进行分类

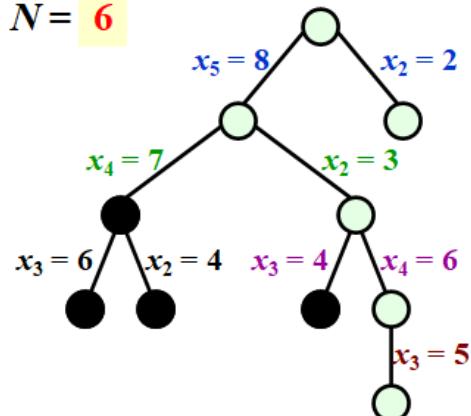
【Example】Given  $D = \{ 1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10 \}$

Step 1:  $N(N - 1)/2 = 15$  implies  $N = 6$

Step 2:  $x_1 = 0$  and  $x_6 = 10$

Step 3: find the next largest distance and check

●○○○●○●●○●  
( 0, 3, 5, 6, 8, 10 )



### Tic-tac-toe

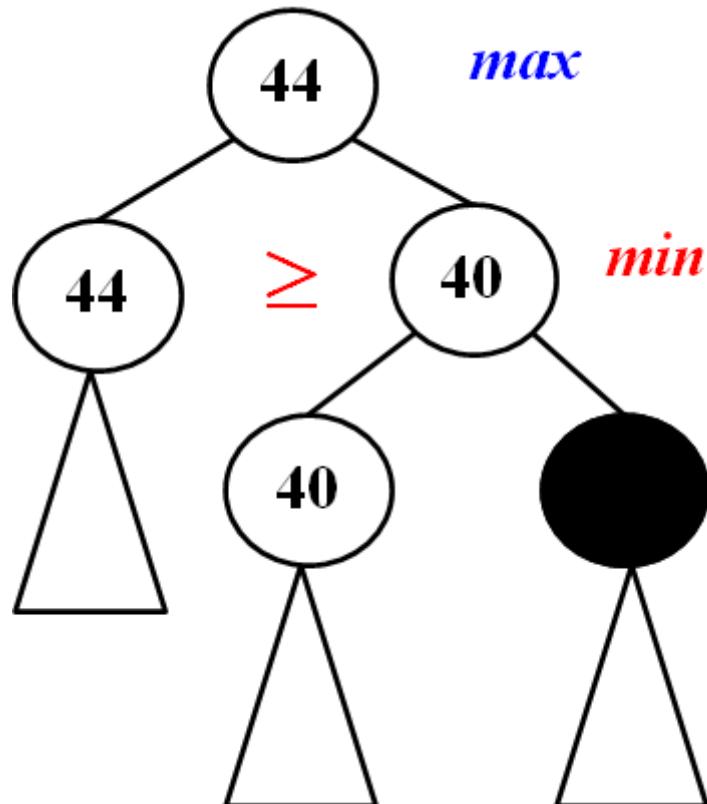
赢面计算：

$$f(P) = W_{Computer} - W_{Human}$$

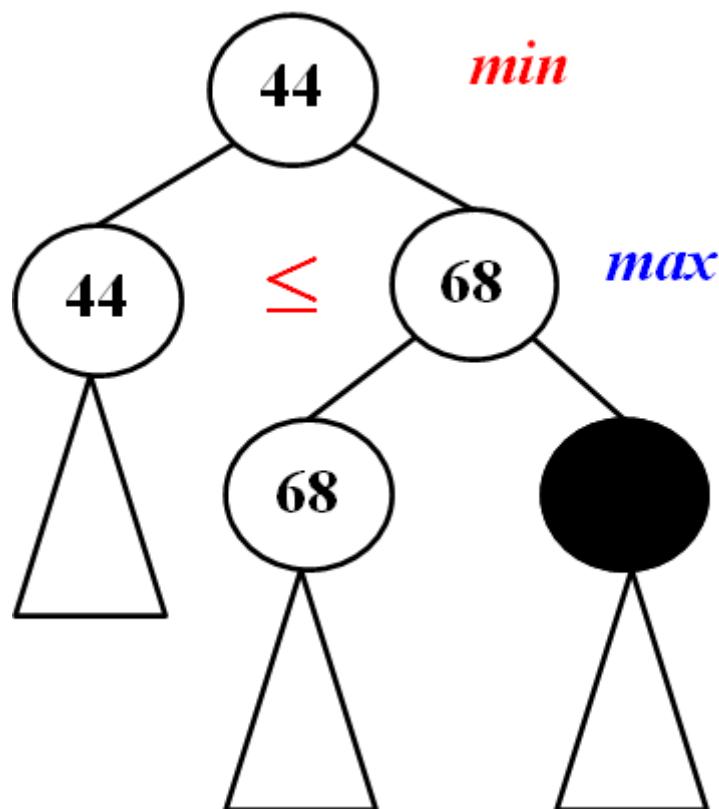
本质：寻找鞍点

### 剪枝

a



$\beta$



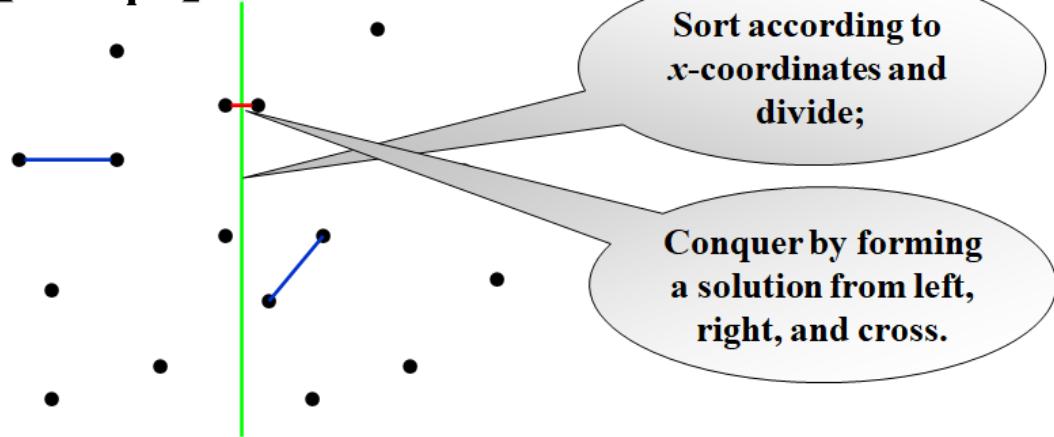
在一棵大小为 $n$ 的搜索树里，同时使用 $\alpha$ - $\beta$ 剪枝可以将搜索空间降低到 $O(\sqrt{n})$

## 09 Divide & Conquer

例子：快速排序、归并排序、最近点距离

Closest Points Problem

## 【Example】



## 时间复杂性分析

### 替代法

Substitution method

先猜后证

$$【Example】 \quad T(N) = 2 T(\lfloor N/2 \rfloor) + N$$

$$\text{Guess: } T(N) = O(N \log N)$$

**Proof:** Assume it is true for all  $m < N$ , in particular for  $m = \lfloor N/2 \rfloor$ .

Then there exists a constant  $c > 0$  so that  
 $T(\lfloor N/2 \rfloor) \leq c \lfloor N/2 \rfloor \log \lfloor N/2 \rfloor$

Substituting into the recurrence:

$$\begin{aligned} T(N) &= 2 T(\lfloor N/2 \rfloor) + N \\ &\leq 2 c \lfloor N/2 \rfloor \log \lfloor N/2 \rfloor + N \\ &\leq c N (\log N - \log 2) + N \\ &\leq c N \log N \quad \text{for } c \geq 1 \end{aligned}$$

注意不能猜错（也就是必须猜到具体形式）

【Example】  $T(N) = 2 T(\lfloor N/2 \rfloor) + N$

Wrong guess:  $T(N) = O(N)$

**Proof:** Assume it is true for all  $m < N$ , in particular for  $m = \lfloor N/2 \rfloor$ .

$$T(\lfloor N/2 \rfloor) \leq c \lfloor N/2 \rfloor$$

Substituting into the recurrence:

$$T(N) = 2 T(\lfloor N/2 \rfloor) + N$$

$$\leq 2c \lfloor N/2 \rfloor + N$$

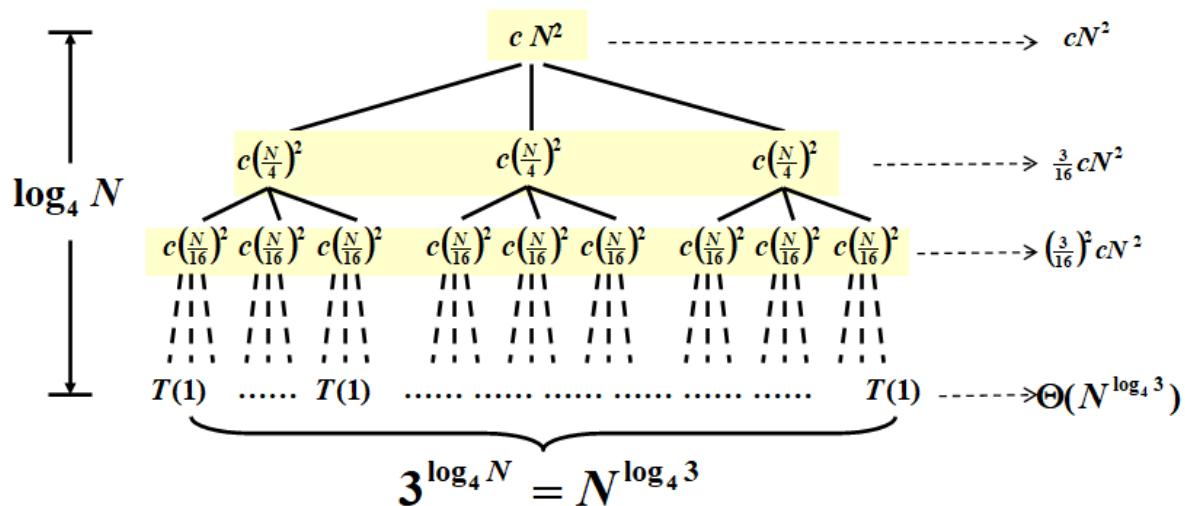
$$\leq cN + N = O(N) \times$$

Must prove the *exact form*

递归树法

Recursion-tree method

【Example】  $T(N) = 3 T(N/4) + \Theta(N^2)$



时间由两部分构成

1. 每一层后面的那个尾巴（也就是最后跟着的 $f(N)$ ，这里是 $\Theta(N^2)$ ）
2. 最后一层的一堆 $T(1)$

$$\begin{aligned} T(N) &= \sum_{i=0}^{\log_4 N - 1} \left(\frac{3}{16}\right)^i cN^2 + \Theta(N^{\log_4 3}) < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cN^2 + \Theta(N^{\log_4 3}) \\ &= \frac{cN^2}{1 - 3/16} + \Theta(N^{\log_4 3}) = O(N^2) \end{aligned}$$

Master 定理

$$T(N) = aT(N/b) + f(N)$$

## 形式1

1. If  $f(N) = O(N^{\log_b a - \varepsilon})$  for some constant  $\varepsilon > 0$ , then  $T(N) = \Theta(N^{\log_b a})$
2. If  $f(N) = \Theta(N^{\log_b a})$ , then  $T(N) = \Theta(N^{\log_b a} \log N)$  regularity condition
3. If  $f(N) = \Omega(N^{\log_b a + \varepsilon})$  for some constant  $\varepsilon > 0$ , and if  $af(N/b) < cf(N)$  for some constant  $c < 1$  and all sufficiently large  $N$ , then  $T(N) = \Theta(f(N))$

## 形式2

**【Master Theorem】** The recurrence  $T(N) = aT(N/b) + f(N)$  can be solved as follows:

1. If  $af(N/b) = \kappa f(N)$  for some constant  $\kappa < 1$ , then  $T(N) = \Theta(f(N))$
2. If  $af(N/b) = Kf(N)$  for some constant  $K > 1$ , then  $T(N) = \Theta(N^{\log_b a})$
3. If  $af(N/b) = f(N)$ , then  $T(N) = \Theta(f(N) \log_b N)$

## 形式3

**【Theorem】** The solution to the equation

$T(N) = a T(N/b) + \Theta(N^k \log^p N)$ ,  
where  $a \geq 1, b > 1$ , and  $p \geq 0$  is

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{if } a > b^k \\ O(N^k \log^{p+1} N) & \text{if } a = b^k \\ O(N^k \log^p N) & \text{if } a < b^k \end{cases}$$

# 10 DP

核心：找状态转移方程（划分子问题）+ 递归变递推（或者记忆化递归）

## 矩阵相乘

- 状态转移方程

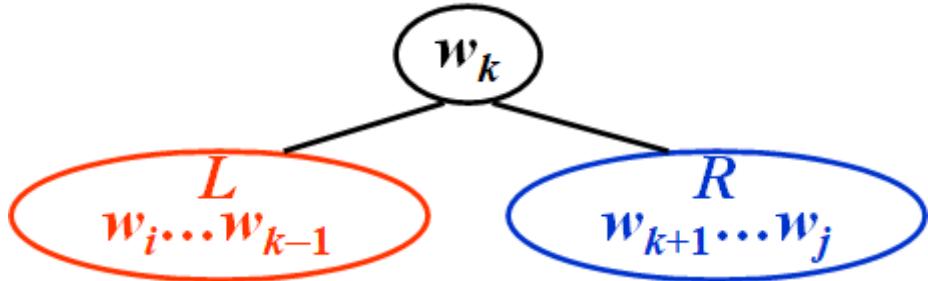
$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq l < j} \{ m_{il} + m_{l+1,j} + r_{i-1}r_l r_j \} & \text{if } j > i \end{cases}$$

$$\begin{array}{ccccccc}
 & & m_{1,N} & & & & \\
 & m_{1,N-1} & m_{2,N} & & & & \\
 & \vdots & \vdots & \ddots & & & \\
 m_{1,2} & m_{2,3} & \cdots & m_{N-1,N} & & & \\
 m_{1,1} & m_{2,2} & \cdots & m_{N-1,N-1} & m_{N,N} & &
 \end{array}$$

每个元素要求min, 就需要 $O(N)$ , 总复杂度 $O(N^3)$

## 最优二叉查找树

- 状态转移方程



左边树的cost+右边树的cost+因为多了一层而增加的cost (即weight) +该点的weight

$$\begin{aligned}
 c_{ij} &\equiv p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R) \\
 &= p_k + c_{i, k-1} + c_{k+1, j} + w_{i, k-1} + w_{k+1, j} = w_{ij} + c_{i, k-1} + c_{k+1, j}
 \end{aligned}$$

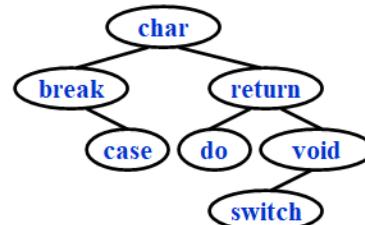
- 推状态表

$$c_{ij} = \min_{i < l \leq j} \{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

| word        | break | case | char | do   | return | switch | void |
|-------------|-------|------|------|------|--------|--------|------|
| probability | 0.22  | 0.18 | 0.20 | 0.05 | 0.25   | 0.02   | 0.08 |

| break..break   |       | case..case    |      | char..char    |        | do..do      |        | return..return  |        | switch..switch |        | void..void |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|----------------|-------|---------------|------|---------------|--------|-------------|--------|-----------------|--------|----------------|--------|------------|------|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| 0.22           | break | 0.18          | case | 0.20          | char   | 0.05        | do     | 0.25            | return | 0.02           | switch | 0.08       | void |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| break.. case   |       | case.. char   |      | char.. do     |        | do.. return |        | return.. switch |        | switch.. void  |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 0.58           | break | 0.56          | char | 0.30          | char   | 0.35        | return | 0.29            | return | 0.12           | void   |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| break.. char   |       | case.. do     |      | char.. return |        | do.. switch |        | return.. void   |        |                |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1.02           | case  | 0.66          | char | 0.80          | return | 0.39        | return | 0.47            | return |                |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| break.. do     |       | case.. return |      | char.. switch |        | do.. void   |        |                 |        |                |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1.17           | case  | 1.21          | char | 0.84          | return | 0.57        | return |                 |        |                |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| break.. return |       | case.. switch |      | char.. void   |        |             |        |                 |        |                |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1.83           | char  | 1.27          | char | 1.02          | return |             |        |                 |        |                |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| break.. switch |       | case.. void   |      |               |        |             |        |                 |        |                |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1.89           | char  | 1.53          | char |               |        |             |        |                 |        |                |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| break.. void   |       |               |      |               |        |             |        |                 |        |                |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2.15           | char  |               |      |               |        |             |        |                 |        |                |        |            |      |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

$$T(N) = O(N^3)$$



## 最短路径

i到j的最短路径[k]

即从i到j允许中间经过0-k的最短路径

取 $[i][j][k-1]$ ,  $[i][k][k-1]+[k][j][k-1]$ 的最小值，也就是k-1的情况下最短路径和一定经过k的最短路径，中间取一个较小的

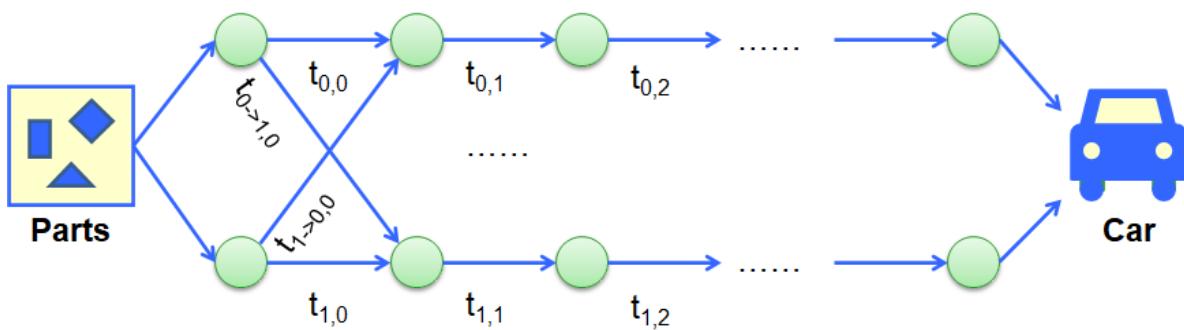
然后从k=0开始建表即可

## Method 2 Define

$D^k[i][j] = \min\{ \text{length of path } i \rightarrow \{l \leq k\} \rightarrow j\}$   
and  $D^{-1}[i][j] = \text{Cost}[i][j]$ . Then the length of the shortest path from  $i$  to  $j$  is  $D^{N-1}[i][j]$ .

$$\therefore D^k[i][j] = \min\{D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j]\}, k \geq 0$$

产品组装



每一个阶段的最优解都基于前一个阶段的最优解，所以顺次递推就行

```
f[0][0]=0; L[0][0]=0;
f[1][0]=0; L[1][0]=0;
for(stage=1; stage<=n; stage++){
    for(line=0; line<=1; line++){
        f_stay = f[line][stage-1] + t_process[line][stage-1];
        f_move = f[1-line][stage-1] + t_transit[1-line][stage-1];
        if (f_stay<f_move){
            f[line][stage] = f_stay;
            L[line][stage] = line;      不换线路
        }
        else {
            f[line][stage] = f_move;
            L[line][stage] = 1-line;    换线路
        }
    }
}
```

$O(N)$  time +  $O(N)$  space

# 11 Greedy

贪心问题里，不是找出所有最优解，也不保证最优解的唯一性，只是保证找出的解一定属于最优解

## 任务选取问题

先找最先结束的任务

Resource become free as soon as possible

$O(N \log N)$

## Huffman Code

- 避免二义性
- 最短编码

哈夫曼树

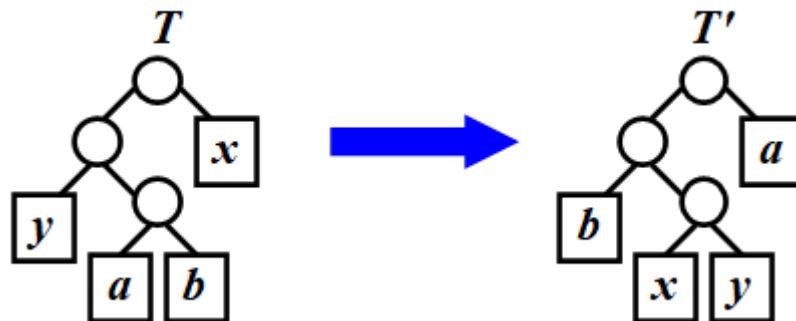
$N$ 个字符， $O(N \log N)$

### 证明

#### 1. 最优性

一定存在一个最优解，它最底层是频率最小的字符

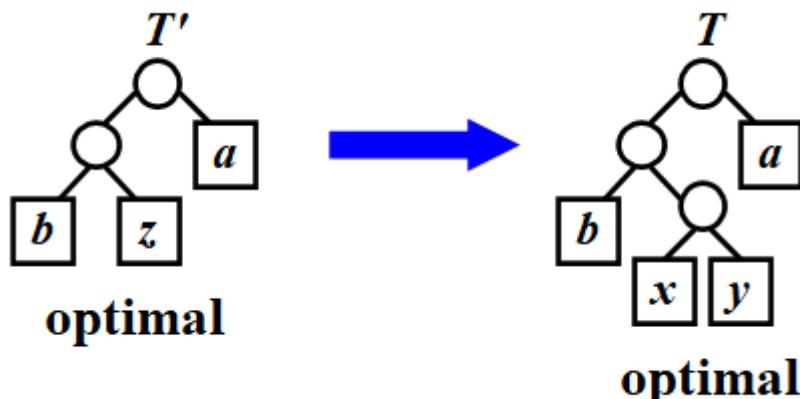
假如频率最低的 $x, y$ 不在最下面，那就换到最下面，一定weight变小



#### 2. 子问题也是最优解问题

也就是得到一个最优解后，替换到原有地方（ $z$ ），得到的还是最优解

直接反证，如果子问题不是最优解，那么换成子问题的最优解，全局一定更优



# 12 NPC

## 计算模型

### 递归函数

recursive function

与程序设计的递归不同

初始函数、构造复杂函数的算子（含递归算子）；

$$f(n) = h(n, f(n - 1))$$

加法： $f(m, 0) = m, f(m, n + 1) = s(f(m, n))$ ; s:后继函数 (+1)

有点类似函数式编程

### $\lambda$ 演算

$\lambda$ -calculus

函数定义方式、变换规则 (变量替换)

$$f(x) = x + 1 : \lambda x. x + 1$$

$$f(2) : (\lambda x. x + 1)2$$

$$f(x, y) = x - y : \lambda x. \lambda y. x - y$$

### 图灵机

Turing Machine

抽象模型，定义计算过程

图灵机是一个五元组 ( $K, \Sigma, \delta, s, H$ )，其中：

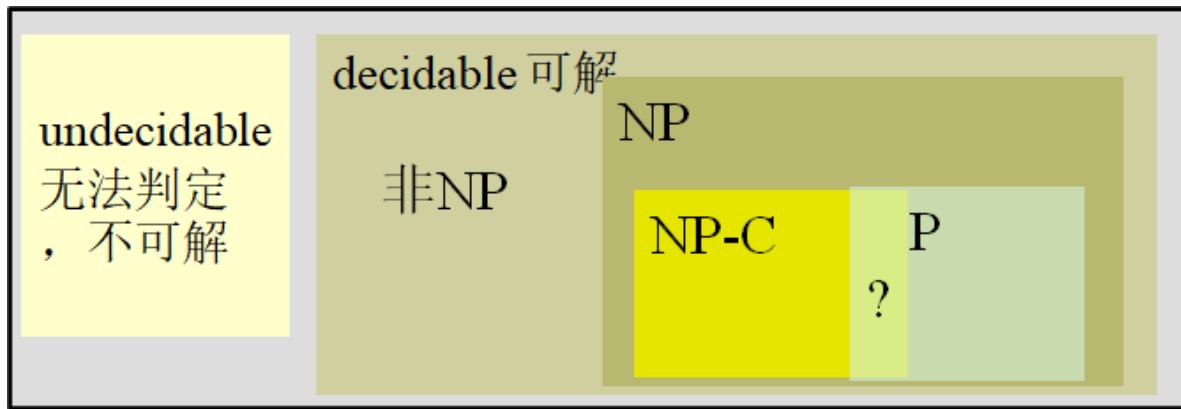
- $K$  是有穷个状态的集合
- $\Sigma$  是字母表，即符号的集合
- $s \in K$  是初始状态
- $H \subseteq K$  是停机状态的集合，当控制器内部状态为停机状态时图灵机结束计算
- $\delta$  是转移函数，即控制器的规则集合



## 可计算性

- 确定图灵机 Deterministic Turing Machine 所有行为都根据确定规则进行
- 不确定图灵机 Nondeterministic Turing Machine 对于任何问题自动做正确选择（玄学机器）

- P问题：确定图灵机上多项式时间内可解问题 polynomial-time
- NP问题：不确定图灵机上多项式时间内可解问题 Nondeterministic polynomial-time, **而且需要在多项式时间内验证解的正确性**
- NPC问题：任何一个NP问题都可以归约到这个问题上（一解百解）
- NP-Hard问题：多项式时间内都无法验证正确性（NP-Hard, “至少跟NP一样难”）



?的含义：目前没有找到NPC问题在确定图灵机上的多项式算法，但是不代表不存在

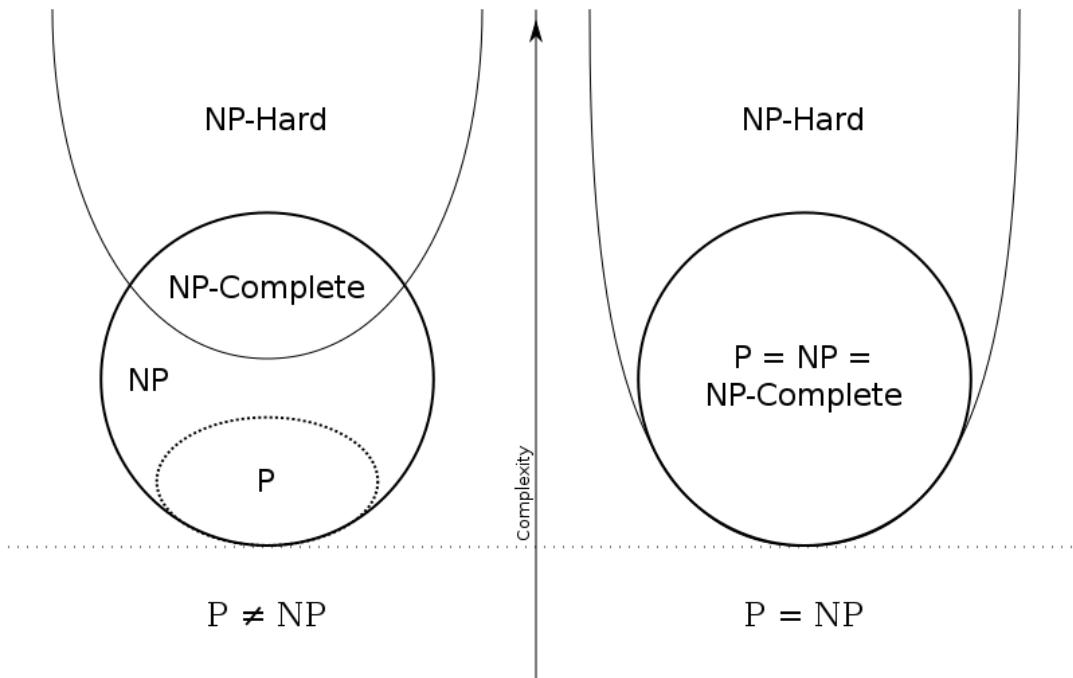
## 一张宝图

左边是NP不等于P的情况，右边是NP=P的情况

如果所有NP问题都可以多项式时间归约到某个问题，则称该问题为NP困难。

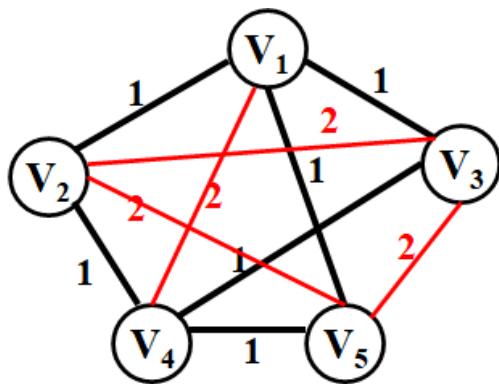
NPC是NP和NP-Hard的交集

一点疑问：为何NPH和NP的交集是NPC？（也许需要计算理论课程进行深入



## 规约

例子：哈密尔回路 -> TSP



## 可满足问题

给定一个布尔表达式，问是否存在使其满足的命题

第一个NPC问题

## 其他NPC

- Circuit satisfiability
- Clique problem
- Vertex-cover problem
- Hamiltonian-cycle problem
- Traveling-salesman problem
- Subset-sum problem
- Subgraph-isomorphism problem
- 0-1 integerprogramming problem
- Set-partition problem
- Longest-simple-cycle problem

## 一些坑

| P                     | NPC                  |
|-----------------------|----------------------|
| Shortest simple paths | longest simple paths |
| Euler tour            | Hamiltonian cycle:   |
| 2-CNF satisfiability  | 3-CNF satisfiability |

## 形式语言

### 映射

问题（一个句子，一个字符串） $\rightarrow$ 映射到01序列 $\rightarrow$ 序列映射到解（Y/N，或者一个答案）

序列对于映射存在合法性（比如可以区分一句话是不是中文）

如果存在一个算法能够在多项式时间内判定一个序列是否属于自己的这种语言，就称这个语言为P类语言

依此类推可以定义出NP类语言

并且可以定义出语言之间的规约关系

然后以此类推定义出NPC类语言

## 一个记号

$\leq_P$  不难于, no harder than

比如  $L_1 \leq_P L_2$ , 就表明L1问题可以在多项式时间内归约到L2问题

# 13 Approximate Algorithms

近似算法

## 近似比

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

如果一个算法达到了 $\rho(n)$ 的近似比, 就称为 $\rho(n)$  – approximation

另一种记号

$(1 + \varepsilon)$  – approximation

## 多项式时间近似算法

polynomial-time approximation scheme (PTAS)

$O(n^{2/\varepsilon})$

$O((1/\varepsilon)^2 n^3)$

后者也被称为完全多项式时间近似算法 fully polynomial-time approximation scheme (FPTAS)

## 案例

### 装箱问题

- Next-Fit

从上次装箱的箱子开始装, 装不下就新开一个, 不回头

```
void NextFit ()  
{  read item1;  
  while ( read item2 ) {  
    if ( item2 can be packed in the same bin as item1 )  
      place item2 in the bin;  
    else  
      create a new bin for item2;  
      item1 = item2;  
    } /* end-while */  
}
```

效果:

假如最优解是M个箱子，那么Next Fit最多使用 $2M-1$ 个

- 证明

### A simple proof for Next Fit:

If Next Fit generates  $2M$  (or  $2M+1$ ) bins, then the optimal solution must generate at least  $M+1$  bins.

Let  $S(B_i)$  be the size of the  $i$ th bin. Then we must have:

$$\begin{aligned} S(B_1) + S(B_2) &> 1 \\ S(B_3) + S(B_4) &> 1 \\ \dots\dots\dots \\ S(B_{2M-1}) + S(B_{2M}) &> 1 \end{aligned} \quad \rightarrow \quad \sum_{i=1}^{2M} S(B_i) > M$$

The optimal solution needs at least  $\lceil \text{total size of all the items} / 1 \rceil$  bins

$$\rightarrow \lceil \text{total size of all the items} / 1 \rceil = \left\lceil \sum_{i=1}^{2M} S(B_i) \right\rceil \geq M + 1$$

- First-Fit

每次都从第一个开始装，能塞下就塞下，都塞不下新开

```
void FirstFit ()  
{ while ( read item ) {  
    scan for the first bin that is large enough for item;  
    if ( found )  
        place item in that bin;  
    else  
        create a new bin for item;  
    } /* end-while */  
}
```

效果：

最多使用 $1.7M$ 个箱子，并且存在 $1.7(M-1)$ 的序列

另外，offline mode下不超过 $11M/9+6/9$ 个箱子

- Best-Fit

每次不仅从头开始装，而且要找利用率最高的方案

$O(\log N)$

最多 $1.7M$

## 背包问题

Knapsack Problem

容量M，N个物体，重量 $w_i$ ，价值 $p_i$

0-1背包  $\&&$   $x_i$  (可拆解)

后者直接找性价比最高的即可

0-1背包DP解法

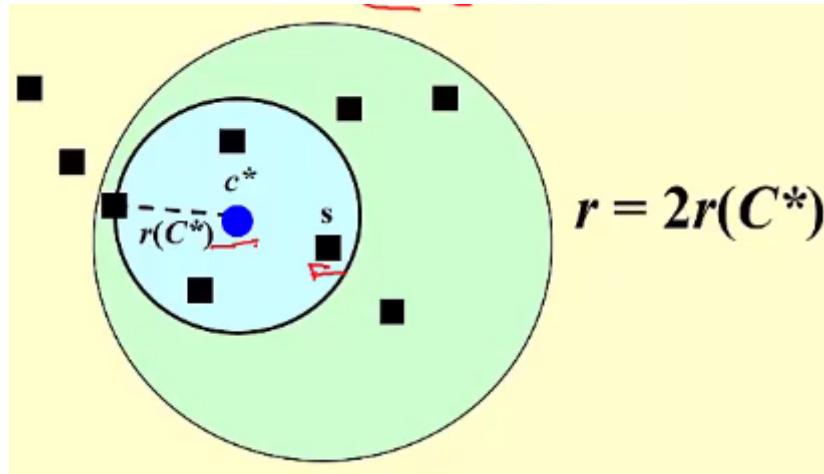
$$W_{i,p} = \begin{cases} \infty & i = 0 \\ W_{i-1,p} & p_i > p \\ \min\{W_{i-1,p}, w_i + W_{i-1,p-p_i}\} & otherwise \end{cases}$$

$$\mathbf{O}(n^2 p_{max})$$

注意这不是多项式时间，因为 $p_{max}$ 其实是一个指数了， $2^{位数}$

## K-center

K个圆，覆盖所有点，求最小半径



先猜一个 $r$ ，然后以任意一个点为圆心、用 $2r$ 去圈，如果 $k$ 次以内能全部圈进，就知道 $r$ 大于最优解 $r(C^*)$

反之则 $r$ 小了

于是就可以用二分来找较为精确的 $r$ ，也就是不超过最优解2倍的 $r$

然后因为 $r(C) \leq 2r(C^*)$ ，于是就得到说是2-appr? ? ?

没有更精确的算法，因为它与DOMINATING-SET问题等价？

### Sketch of the proof: By contradiction.

If we can obtain a  $(2-\varepsilon)$ -approximation in polynomial time, then we can solve DOMINATING-SET (which is NP-complete) in polynomial time.

Dominating set problem has a solution of size  $K$  iff there exists  $K$  centers  $C^*$  with  $r(C^*) = 1$ .

Then a  $(2-\varepsilon)$ -approximation must give the optimal solution since all the distances involved are integers.

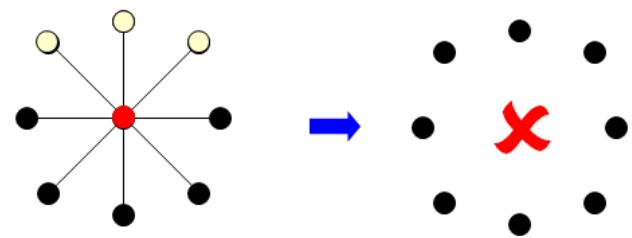
## 14 Local Search

## 顶点覆盖

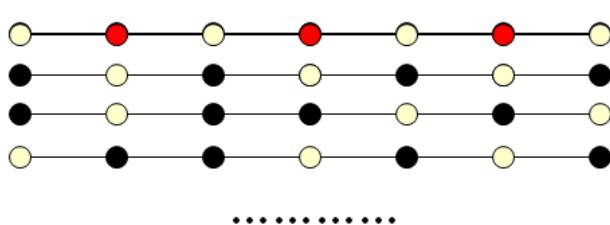
**Case 0:**



**Case 1:**



**Case 2:**



```

SolutionType Metropolis()
{ Define constants  $k$  and  $T$ ;
  Start from a feasible solution  $\mathbf{S} \in FS$ ;
  MinCost = cost( $\mathbf{S}$ );
  while (1) {
     $\mathbf{S}'$  = Randomly chosen from  $N(\mathbf{S})$ ;
    CurrentCost = cost( $\mathbf{S}'$ );
    if (CurrentCost < MinCost) {
      MinCost = CurrentCost;  $\mathbf{S}$  =  $\mathbf{S}'$ ;
    }
    else {
      With a probability  $e^{-\Delta \text{cost}/(kT)}$ , let  $\mathbf{S} = \mathbf{S}'$ ;
      else break;
    }
  }
  return  $\mathbf{S}$ ;
}

```

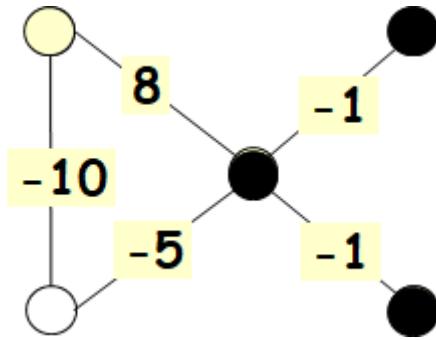
模拟退火， $T$ 温度越来越低

## Hopfield 神经网络

对于每条边， $w_e s_u s_x < 0$

但是并不能保证有解，所以追求每个点的 $\sum_{v:e=(u,v) \in E} w_e s_u s_v \leq 0$

也就是每个点的边中，好的边的权重和大于差的边



谁不满足就翻谁

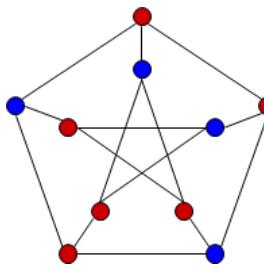
```
ConfigType State_flipping()
{
    Start from an arbitrary configuration S;
    while ( ! IsStable(S) ) {
        u = GetUnsatisfied(S);
        su = - su;
    }
    return S;
}
```

算法一定会停下，因为

$$\Phi(S') = \Phi(S) - \sum_{\substack{e: e=(u,v) \in E \\ e \text{ is bad}}} |w_e| + \sum_{\substack{e: e=(u,v) \in E \\ e \text{ is good}}} |w_e| \geq \Phi(S) + 1$$

也就是每次操作对于全局来说都是递增的

## 最大割集 The Maximum Cut Problem



没错就是离散里的那个

但是不是每个图都能做到

于是又来..... 要让  $w(A, B) := \sum_{u \in A, v \in B} w_{uv}$  最大

跟前一个例子一模一样的翻转策略

### 效果

最少能达到最有权重的一半

- How good is this local optimum?

**Claim:** Let  $(A, B)$  be a local optimal partition and let  $(A^*, B^*)$  be a global optimal partition. Then  $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$ .

**Proof:** Since  $(A, B)$  is a local optimal partition, for any  $u \in A$

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

Summing up for all  $u \in A$

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B)$$

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B)$$

$$w(A^*, B^*) \leq \sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} + w(A, B) \leq 2w(A, B)$$

改进

### Big-improvement-flip

只有当改进幅度大于  $\frac{2\varepsilon}{|V|} w(A, B)$  的时候才能允许变动

**Claim:** Upon termination, the big-improvement-flip algorithm returns a cut  $(A, B)$  so that

$$(2 + \varepsilon) w(A, B) \geq w(A^*, B^*)$$

**Claim:** The big-improvement-flip algorithm terminates after at most  $O(n/\varepsilon \log W)$  flips.

根据时间简单描述证明：

1. 每次 flip 至少增加  $(1 + \varepsilon/n)$  倍，其实是  $(1 + 2\varepsilon/n)$  倍
2.  $n/\varepsilon$  次 flip 之后，总增长至少是 2 倍。利用  $(1 + 1/x)^x \geq 2$ , 如果  $x \geq 1$
3. 总量不超过  $W$ ，而 cut 翻倍的次数不能超过  $\log W$

### K-L heuristic

KL启发式

**Step 1:** make 1-flip as good as we can –  $O(n)$   $\rightarrow (A_1, B_1)$   
and  $v_1$

**Step k:** make 1-flip of an *unmarked* node as good as we can –  
 $O(n-k+1)$   $\rightarrow (A_k, B_k)$  and  $v_1 \dots v_k$

**Step n:**  $(A_n, B_n) = (B, A)$

**Neighborhood of  $(A, B)$**  = {  $(A_1, B_1), \dots, (A_{n-1}, B_{n-1})$  }

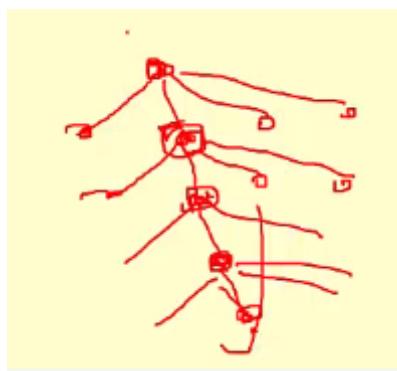
$O(n^2)$

每一步探索n个方向，然后找一个最小的

而且哪怕最好的比现在差也要继续

另外为了防止翻来翻去，一个点只翻一次（只翻转*unmarked*），翻过的就mark掉

然后再在这n层探索之中，选一个最好的



—何老师的灵魂图解

## 15 随机算法

算法本身随机，而不是用来处理随即输入的算法

### 雇人问题

按顺序，比当前最好的还要好就录用

最坏情况：一个比一个好，全都录用了.....  $O(NC_h)$

改进：

1. 离线算法，先将序列打乱然后进行处理

将最厉害的人出现在每个位置的概率平均到 $1/i$ ，然后算一算就能把worst降到  
 $O(C_h \ln N + NC_i)$

2. 在线算法

顺序面试，前K个人只面不录取

- What is the probability we hire the best qualified candidate for a given k?

$$\frac{k}{N} \ln\left(\frac{N}{k}\right) \leq \Pr[S] \leq \frac{k}{N} \ln\left(\frac{N-1}{k-1}\right)$$

- What is the best value of k to maximize the above probability?

$$\frac{N}{e}$$

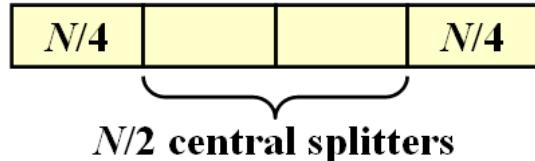
## 快排

中央分布：每边都至少 $1/4$

一直挑选到符合中央分布？不划算

直接按照期望值，做两次（？**warning**似乎可能有理解偏差，tmd太难了

**Claim:** The expected number of iterations needed until we find a central splitter is at most 2.



$$\Pr[\text{find a central splitter}] = 1/2$$

然后就分治法

算出来复杂度 $O(N \log N)$

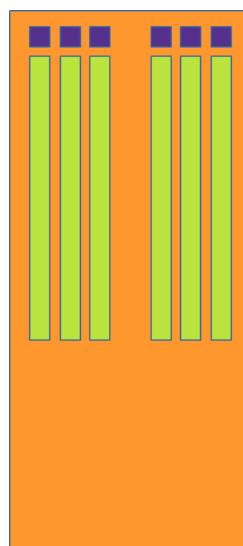
## 16 并行算法

### 并行模型

#### PRAM

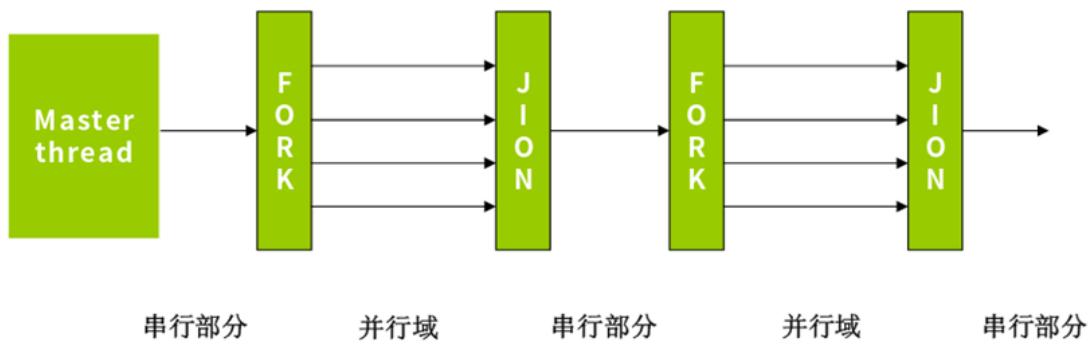
一开始设定好并行数量，然后一直并行

Parallel Random Access Machine



显然不灵活

#### Work-Depth



可以在不同阶段设计不同的并行策略

评价标准:  $W(n), T(n)$  工作量 + 时间复杂度

工作量Work 时间Depth, 顾名思义

## 锁

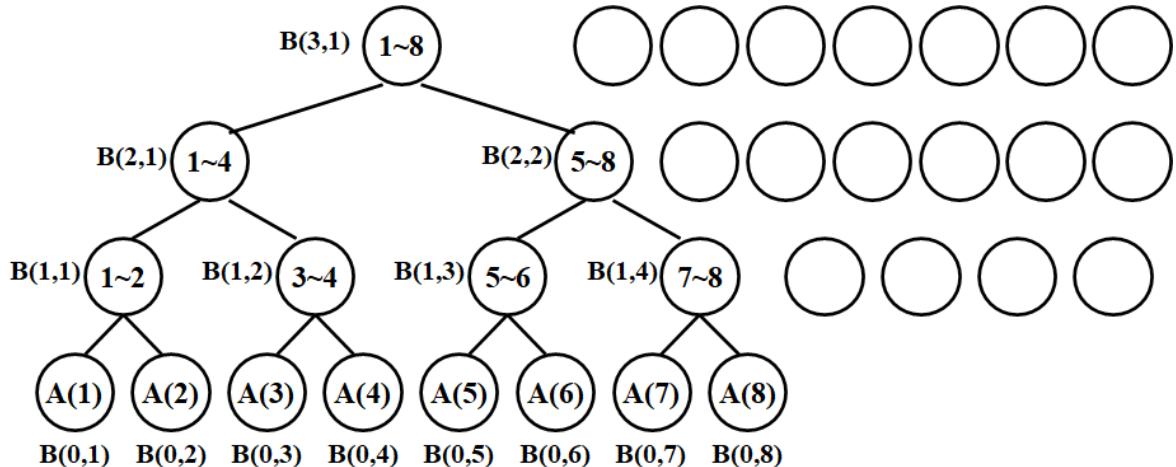
为了解决读写的冲突, 引入了锁机制

- Exclusive-Read Exclusive-Write (EREW) 读写互斥
- Concurrent-Read Exclusive-Write (CREW) 并发读取, 互斥写
- Concurrent-Read Concurrent-Write (CRCW) 并发读写

## 典例

### 求和

Reduction



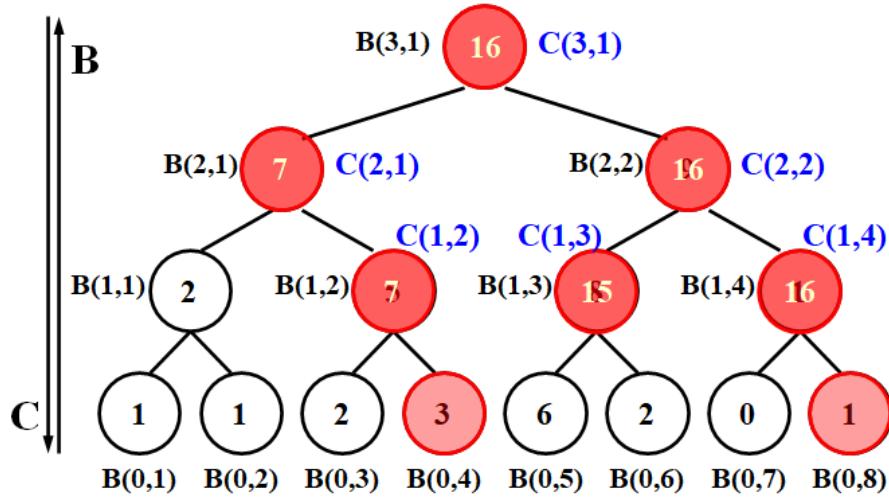
$$B(h, i) = B(h-1, 2i-1) + B(h-1, 2i)$$

二维数组, 右边是空闲的空间

$$T(n) = \log n + 2$$

$$W(n) = 2n$$

### 前缀和



人话：“把某个节点开始向下，走到最右下角的点之前的所有元素相加”

严谨表述：

$$C(h, i) = \sum_{k=1}^{\alpha} A(k)$$

**where  $(0, \alpha)$  is the  
rightmost descendant  
leaf of node  $(h, i)$ .**

那么首先跟求和的方式一样，把B填进去

然后填C

填C有三类情况：

1.  $i == 0$ ,  $C(h, i) = B(h, i)$
2.  $i \% 2 == 0$ , 也就是所有最右边的点,  $C(h, i) = C(h+1, i/2)$
3. 剩下的 ( $i \% 2 == 1 \&& i != 1$ ),  $C(h, i) = D(h+1, (i-1)/2) + B(h, i)$

三种C都可以并行计算，最后  $T(n) = O(\log n)$ ,  $W(n) = O(n)$

```

for Pi, 1 ≤ i ≤ n pardo
    B(0, i) := A(i)
    for h = 1 to log n
        for i, 1 ≤ i ≤ n/2h pardo
            B(h, i) := B(h - 1, 2i - 1) + B(h - 1, 2i)
        for h = log n to 0
            for i even, 1 ≤ i ≤ n/2h pardo
                C(h, i) := C(h + 1, i/2)
            for i = 1 pardo
                C(h, 1) := B(h, 1)
            for i odd, 3 ≤ i ≤ n/2h pardo
                C(h, i) := C(h + 1, (i - 1)/2) + B(h, i)
        for Pi, 1 ≤ i ≤ n pardo
    Output C(0, i)

```

## 归并

核心：分块

思路：转化为rank的计算

RANK(j, A) 返回B[j]在A中的rank / index

RANK(i, B) 返回A[i]在B中的rank / index

于是：

```

for Pi, 1 ≤ i ≤ n pardo
    C(i + RANK(i, B)) := A(i)
for Pi, 1 ≤ i ≤ m pardo
    C(i + RANK(i, A)) := B(i)

```

如果能给出rank的实现，就能在T=O(1), W=O(n+m)下完成

小序列里直接二分或者顺序

### Binary Search

```

for Pi, 1 ≤ i ≤ n pardo
    RANK(i, B) := BS(A(i), B)
    RANK(i, A) := BS(B(i), A)

```

$$T(n) = O(\log n)$$

$$W(n) = O(n \log n)$$

### Serial Ranking

```

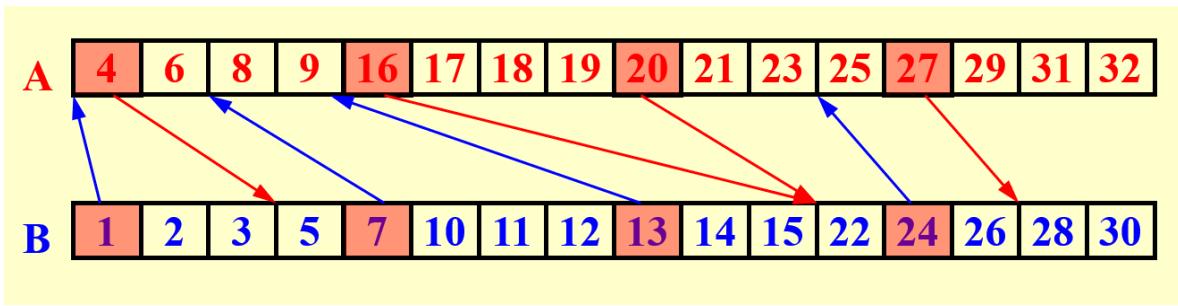
i = j = 0;
while ( i ≤ n || j ≤ m ) {
    if ( A(i+1) < B(j+1) )
        RANK(++i, B) = j;
    else RANK(++j, A) = i;
}

```

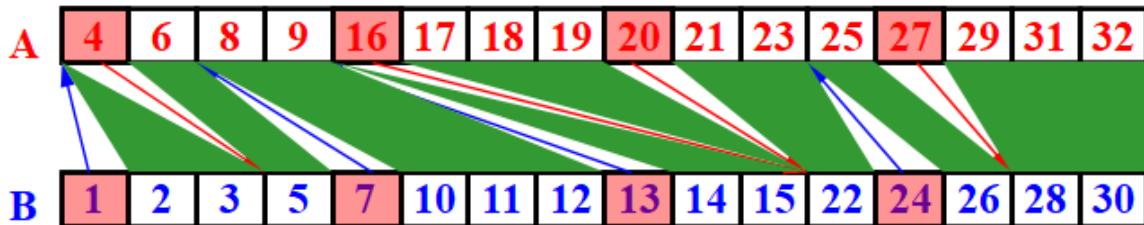
$$T(n) = W(n) = O(n + m)$$

大序列里进行并行ranking

先在两边各自抽取p=n/logn个元素，然后找他们在对方的index,  $T = O(\log n)$ ,  
 $W = O(p \log n) = O(n)$



然后根据这个对应关系，将每一个箭头中间的元素分为一个block，也就是下图中每一个绿色的部分



最多得到 $2n/\log n$ 个block，每个block最多 $O(\log n)$ 个元素， $T = O(\log n)$ ,  $W = O(p \log n) = O(n)$

于是整个工作量， $T = O(\log n)$ ,  $W = O(n)$

## 查找最大 Maximum Finding

- 方案1

Map Reduce 把+换为max

$$T(n) = O(\log n), W(n) = O(n)$$

- 方案2

使用“只写”，两两比较，小的标记1，然后最后找到没标记的就是max

$$T(n) = O(1), W(n) = O(n^2)$$

- 方案3 double log

分块

- 按照 $\sqrt{n}$ 分块

产生出 $\sqrt{n}$ 个块，每个块有 $\sqrt{n}$ 个元素，需要 $T(\sqrt{n})$ 和 $W(\sqrt{n})$

然后得到的最后 $\sqrt{n}$ 个临时结果，再在 $T = O(1)$ 时间内，用 $W = O(\sqrt{n}^2) = O(n)$ 的工作量处理掉，就好

因此得到： $T(n) \leq T(\sqrt{n}) + c_1$ ,  $W(n) \leq \sqrt{n}W(\sqrt{n}) + c_2 n$

最后总的时间和工作量： $T(n) = O(\log \log n)$ ,  $W(n) = O(n \log \log n)$

- 按照 $h = \log(\log n)$ 分块

分析与前面的一致，最后得到： $T(n) = O(h + \log \log(n/h)) = O(\log \log n)$ ,

$$W(n) = O(h \times (n/h) + (n/h) \log \log(n/h)) = O(n)$$

也就是说，时间没有降低，但是工作量降低了

- 方案4 随机采样

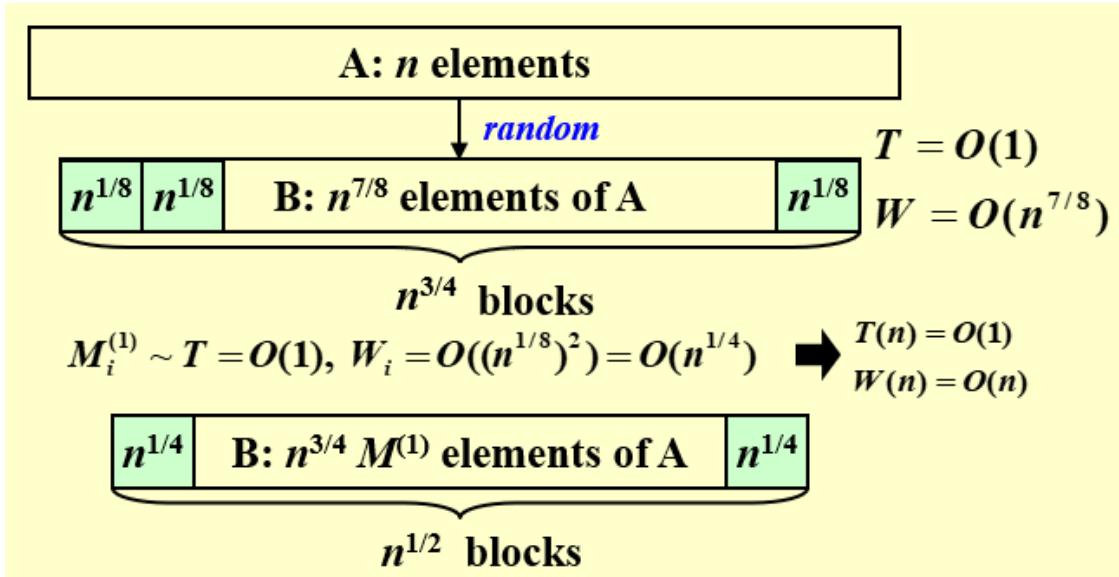
$n$ , 随机抽取 $n^{7/8}$ , 然后分成 $n^{3/4}$ 个 $n^{1/8}$ 大小的块，这一步需要 $T = O(1)$ ,  $W = O(n^{7/8})$

然后每一个块内部使用两两比较的方法，时间 $O(1)$ ，工作量 $O((n^{1/8})^2) = O(n^{1/4})$ ，这是每个block的，总的来说时间因为并行还是 $O(1)$ ，工作量乘上了block数量就等于 $O(n)$

然后对于得到的 $n^{3/4}$ 个最大值，再进行分块，分为 $n^{1/2}$ 个 $n^{1/4}$ 大小的块

如法炮制， $O(1)$   $O(n)$ 之后，又得到 $n^{1/2}$ 个最大值

每一步都是 $O(1)$   $O(n)$ , 步数为const, 所以整体为 $O(1)$   $O(n)$



但是这只能得到 $n^{7/8}$ 的元素中的最大值

因此多搞几轮,

```
while (there is an element larger than M) {
    for (each element larger than M)
        Throw it into a random place in a new B( $n^{7/8}$ );
        Compute a new M;
}
```

最后在 $O(1)$ ,  $O(n)$ 内找到最大值的概率非常的高

**Theorem】 The algorithm finds the maximum among  $n$  elements. With very high probability it runs in  $O(1)$  time and  $O(n)$  work. The probability of not finishing within this time and work complexity is  $O(1/n^c)$  for some positive constant  $c$ .**

## 17 外部排序

- 基本: run之间合并, 小run变大run
- 改进1: 增加每次合并的run数量, k-way合并 (使用heap)
  - 缺点: 需要过多磁盘空间
- 改进2: 3tape 2way
  - 注意: 使用fib数列进行分割
- 改进3: buffer



input buffers



output buffer

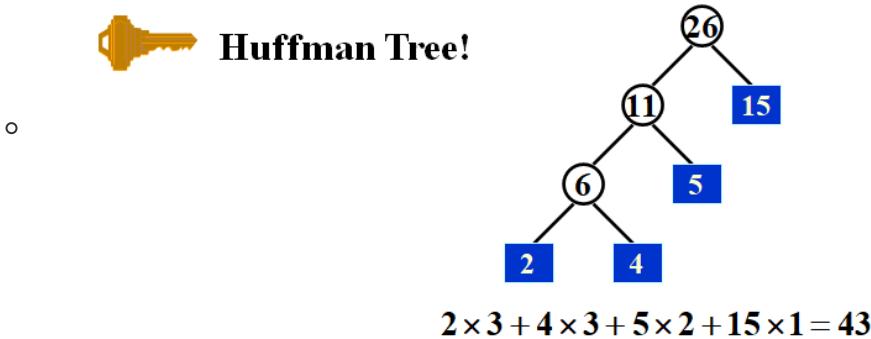
- input和output buffer都要按照2倍来划分, 这样output写到一半就能赶紧往磁盘写出, output处理完一半也能及时从硬盘读取

- input  $2k$  ( $k$ -way merge)
- output 2
- $k$ 不易太大



- 改进4: 增加run的长度 longer run
  - **Replacement selection**
    - 读进内存的数不应小于刚输出的数, 若小于则说明不是这个run的, 应当划分到下一个run
    - 效果: 平均获得两倍内存大小的run,  $L_{avg} = 2M$
    - 当输入已经基本排好序的时候效果最好
- 改进5: 降低合并次数
  - 本质: Huffman Tree

**【Example】 Suppose we have 4 runs of length 2, 4, 5, and 15, respectively. How can we arrange the merging to obtain minimum merge times?**



**Total merge time =  $O(\text{the weighted external path length})$**