前言：因为高级数据结构这门课是专业课，平时成绩 20%，Lab 30%，闭卷期末考试 50%。这门课平时笔记内容不多，主要是节奏非常快以及一些可视化的过程难以记笔记。故开此章节，以冀自己能够及时复习，不要把疑惑拖到期末考试。同时，老师在课上也说过，期末考试有一种题型可能是示例代码挖空来让同学填空。这就要求不仅仅是对概念了解，对于 PPT 上列出来的源码也要尽可能剖析开。

上到目前第八周，总体脉络还是很清楚的。首先是跳表，然后是哈希，然后是 BST（二叉搜索树），其中的 BBST（平衡二叉搜索树）又讲了 AVL（自适应平衡二叉搜索树）、Splay Tree（伸展树）、B 树、红黑树。由于节奏很快，作业的覆盖面又相对较少，针对课程中比较难以理解的概念，提供一个目录文档，来帮助期末复习。

目录

Splay Tree

为了介绍伸展树，首先我们要引入两个局部性原理：

·时间局部性：如果一个信息项正在被访问，那么在近期它很可能还会被再次访问。程序循环、堆栈等是产生时间局部性的原因。
空间局部性：在最近的将来将用到的信息很可能与正在使用的信息在空间地址上是临近的。

换句话说，AVL 树针对理想随机插入已经获得了比较好的情况。但是没有利用在现实中，数据具有局部性的特点，进一步优化性能。Splay Tree 为了解决这个问题，就采用了一个比较直观的方法：节点一旦被访问，随机调整至树根。虽然这个逻辑很简单，但是实现起来效率还是天差地别，主要可以分为：

1. 逐层伸展还是双层伸展
2. zig-zig 和 zag-zag 是否采用了微调过的与 AVL 树不同的版本
3. 自顶向下查找+自底向上伸展跑两遍还是自顶向下一遍解决

https://blog.csdn.net/canot/article/details/79968748
思路：上升了以后要尽可能地把周围的结点也往上提。

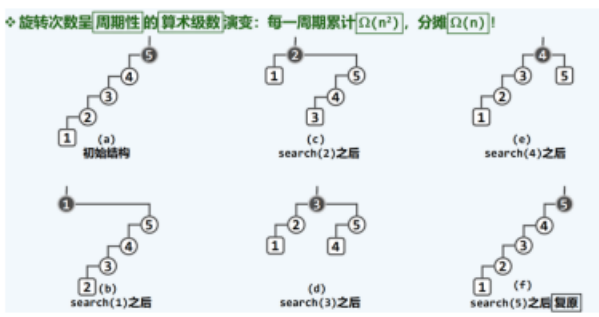
伸展树的查找会引起整棵树的结构调整，所以要重写查找功能。

逐层伸展

我们首先考虑伸展树的逐层伸展

概念：节点 v 一旦被访问，随机转移至树根。采用的方法是一步一步往上爬，即自下而上、逐层进行单旋（zig(v->parent),zag(v->parent)），直到 v 最终被旋转到树根。

但是，很容易我们发现，这样是非常低效的，很容易出现周期性的单链情况，取决于树的初始形态以及访问节点的顺序。



经过研究后，我们发现出现低效率的问题原因在于：

1. 全树拓扑始终呈单链条结构，等价于一维列表。
2. 被访问节点的深度，呈周期性的算数级数演变，平均为 $\Omega(n)$ 。

双层伸展

为了解决单层伸展低效的问题，我们引入了双层伸展。比起逐层伸展我们只往上追溯一层的情况，我们的双层伸展向上追溯两层，考察当前节点、父节点和祖父节点的相对位置关系，进行对应的双旋操作。

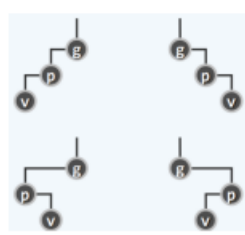


图 1 向上追溯两层后对应的四种位置关系

如果是第二层的两种情况，我们只需要 zag-zig（p 先左旋 g 再右旋）、zig-zag（p 先右旋 g 再左旋）即可，和 AVL 树中的两种双旋完全等价。

针对第一层的两种情况，我们针对伸展树的特性进行改进，如下图所示：

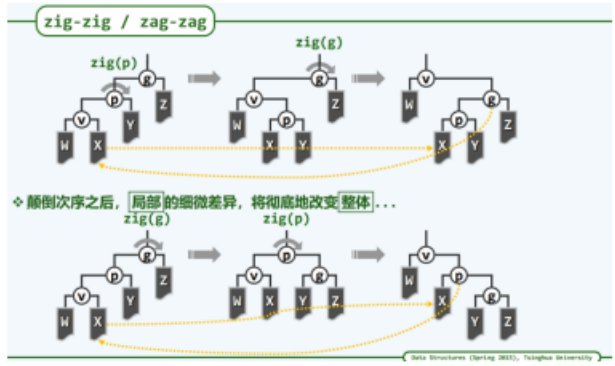


图 2 zig-zig 情况和 zag-zag 情况的改进

原先的情况是先旋转父节点 p 再旋转祖父节点 g，我们修改为先旋转祖父节点 g 再旋转父结点 p。这样的好处是，靠近待访问节点 v 的父结点 p 再旋转完后深度更低，根据局部性原理，下次访问 p 需要查找的深度更低；同样地，v 的右子树 x 现在直接连到了 p 之下，也更加符合了局部性原理，提升了效率。

一个边界情况就是 v 没有祖父的情况，此时 v 的父节点一定是根节点，我们只需要根据 v 的位置关系，做一次单旋即可将 v 调整到根。

伸展树的接口

类的定义

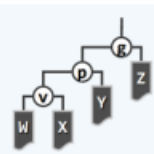
```
template <typename T>
class Splay : public BST<T> { //由BST派生
protected: BinNodePosi(T) splay( BinNodePosi(T) v ); //将v伸展至根
public: //伸展树的查找也会引起整树的结构调整，故search()也需重写
   BinNodePosi(T) & search( const T & e ); //查找 重写
   BinNodePosi(T) insert( const T & e ); //插入 重写
   bool remove( const T & e ); //删除 重写
};
```

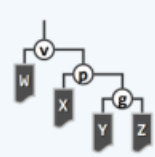
伸展算法

```
template <typename T> BinNodePosi(T) Splay<T>::splay( BinNodePosi(T) v ) {
   if ( ! v ) return NULL; BinNodePosi(T) p; BinNodePosi(T) g; //父亲、祖父
   while ( (p = v->parent) && (g = p->parent) ) { //自下而上，反复双层伸展
     BinNodePosi(T) gg = g->parent; //每轮之后，v都将以原曾祖父为父
     if ( IsLChild( * v ) )
          if ( IsLChild( * p ) ) { /* zig-zig */ } else { /* zig-zag */ }
     else if ( IsRChild( * p ) ) { /* zag-zag */ } else { /* zag-zig */ }
     if ( ! gg ) v->parent = NULL; //若无曾祖父gg，则v现即为树根；否则，gg此后应以v为左或右
     else ( g == gg->lc ) ? attachAsLChild(gg, v) : attachAsRChild(gg, v); //孩子
     updateHeight( g ); updateHeight( p ); updateHeight( v );
   } //双层伸展结束时，必有g == NULL，但p可能非空
   if ( p = v->parent ) { /* 若p果真是根，只需在额外单旋（至多一次） */ }
   v->parent = NULL; return v; //伸展完成，v抵达树根
}
```



```
if ( IsLChild( * v ) )
   if ( IsLChild( * p ) ) { //zIg-zIg
      attachAsLChild( g, p->rc );
      attachAsLChild( p, v->rc );
      attachAsRChild( p, g );
      attachAsRChild( v, p );
   } else { /* zIg-zAg */ }
else
   if ( IsRChild( * p ) ) { /* zAg-zAg */ }
   else { /* zAg-zIg */ }
```





注意，比起AVL树中的3-4联结，这里封装了更为简单的接口attachAsLChild和attachAsRChild。

考试的时候可能会把某个空去掉。

查找算法

```
template <typename T> BinNodePosi(T) & Splay<T>::search( const T & e ) {
   // 调用标准BST的内部接口定位目标节点
      BinNodePosi(T) p = searchIn( _root, e, _hot = NULL );
   // 无论成功与否，最后被访问的节点都将伸展至根
      _root = splay( p ? p : _hot ); //成功、失败
   // 总是返回根节点
      return _root;
   }
```

- 伸展树的查找操作，与常规BST::search()不同
  很可能会改变树的拓扑结构，不再属于静态操作

_hot 应该是查找到 null 前，访问的最后一个结点。虽然查找失败了，但是还是可以以_hot 节点进行伸展。

这样子是比较低效的，先要下去找到它，在向上移动到根。有没有优化的方法？

自顶向下的伸展树

https://blog.csdn.net/yw8355507/article/details/47108351

这样就需要 top-down spray，也就是一边往下找，一边把这件事完成。从上往下到合理的地方就停止了，把查找和删除合在一起，只做一次。

定义概念：L是结点 x 的左子树，R 是结点 x 的右子树，T是以结点 x 为根的树。

一开始 T 只有根节点，我们把 T 的子树越变越大，最后把全部左右子树的内容包进来，所以我们做一次就可以把事情完成。也就是把现有的那棵树按照某种算法插进去。T 一次往下压两层，我们考虑单旋和两个双旋，对称后共六种情况。

最后，在查找到节点后，将三棵树合并。如图：]

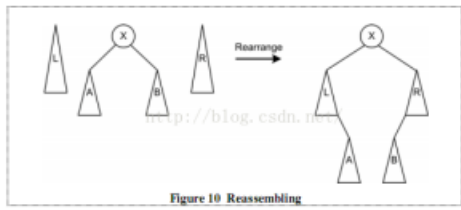
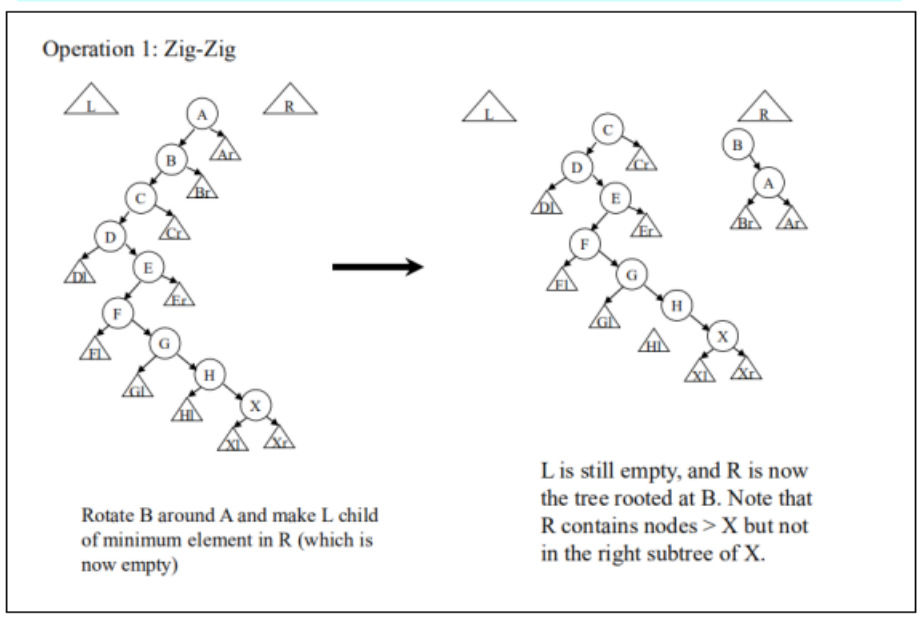
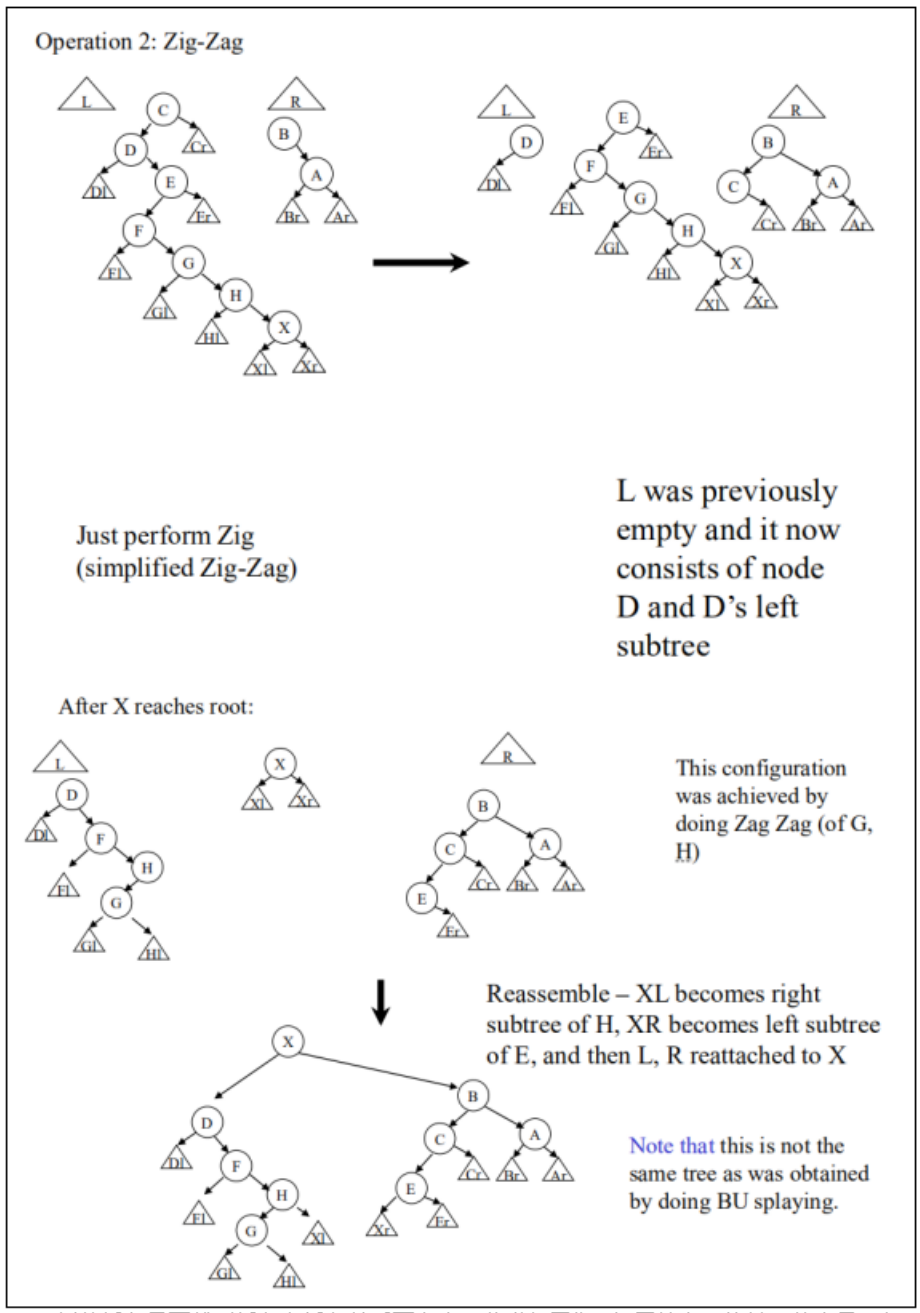


Figure 10 Reassembling

将中树的左右子树分别连接到左树的右子树和右树的左子树上，将左右树作为X的左右子树。重新最成了一所查找的节点为根的树。

When the value to be splayed to the root is at the root of the "center" tree, we have reached the point where we are ready to reassemble the tree. This is accomplished by a) making XL the right child of the maximum element in L, b) making XR the left child of the minimum element in R, and then making L and R the left and right children of X





AVL 树的插入是要找到插入点插入以后再向上回溯进行平衡。根开始向下旋转可能也是可行的。

代码大全：http://users.cis.fiu.edu/~weiss/dsaa_c++3/code/

B 树

数据正在不断扩大，但是内存还是很小。
以前是 CPU->RAM->DISK
现在是 CPU 到 RAM 和 NVM。甚至可以做成一个内存池。
存储金字塔

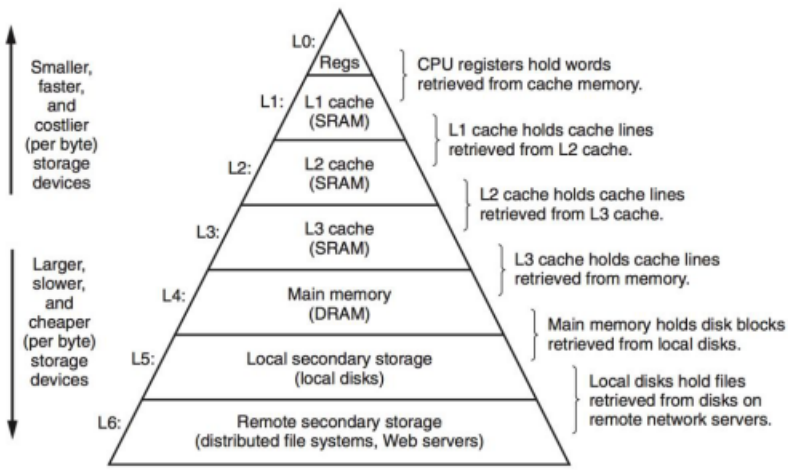


Figure 6.23 **The memory hierarchy.**

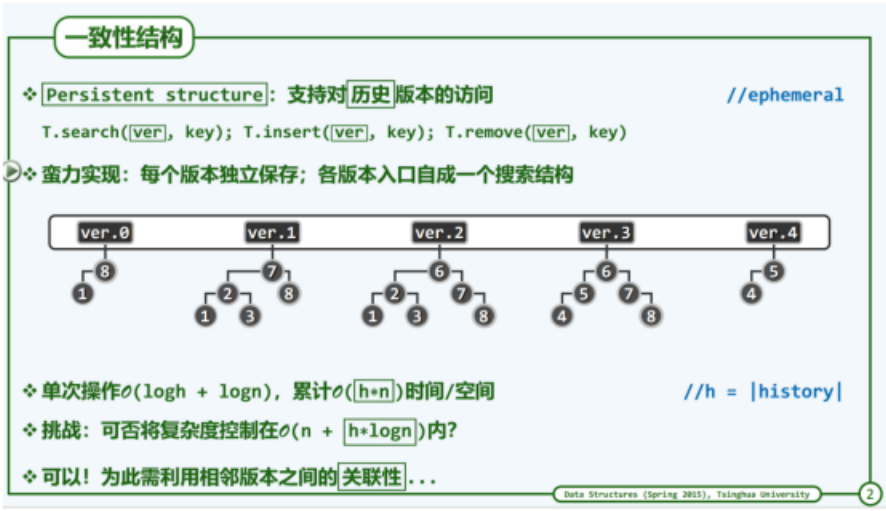
从磁盘中读写 1B，与读写 1KB 几乎一样快。

B 树复杂度的计算要分成内存中的和磁盘中的。尽可能减少对磁盘的访问。

B 树的上溢和下溢问题
https://blog.csdn.net/qq_44096670/article/details/113074190

从独自分裂到联合分裂。节点上溢后，由 k 个饱和的兄弟均摊新关键码。

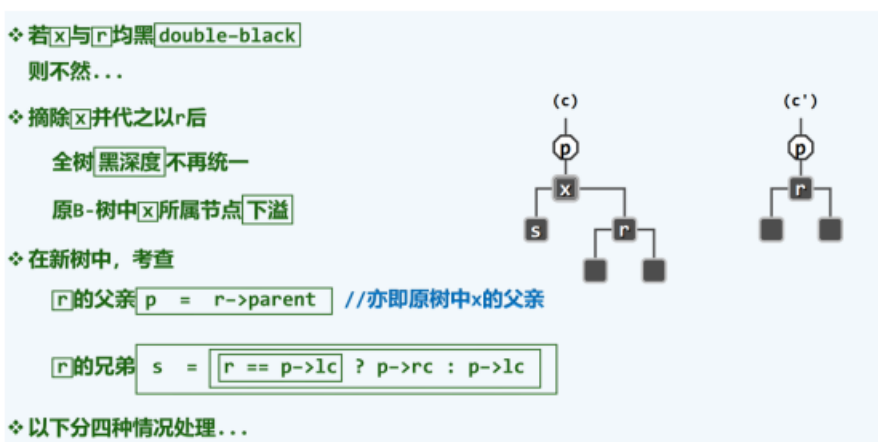
红黑树



2021/4/15 高级数据结构

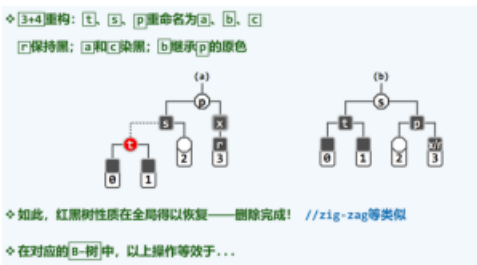
红黑树的删除

有两种情况是比较好的，被删的结点是红孩子，右儿子提升去代替他。可以直接调平衡。最后一种情况就是黑黑情况，如果直接代提上去，黑高度就会不平衡，BB 情况又分好几种。



考察 x 的左儿子的情况。

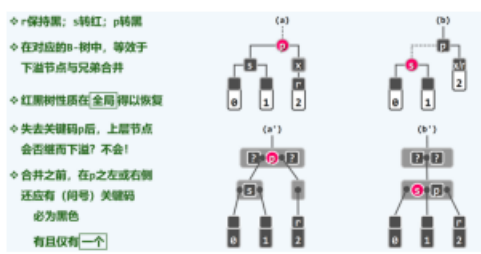
情况 1：左儿子黑 s，有一个红孩子。



t 是红的，不限左右。我们做一次旋转，相当于 B 树的下溢。换句话说，我们去掉红孩子的红色，还有一个黑的，保证黑高度的平衡。s 提高一层，整个黑高度不变，p 是什么颜色，s 还是什么颜色。这就是一种平衡。

情况 2：BB2 情况

BB2 分为两种情况，一种情况是双黑儿子 p 是红（BB-2R）的情况，另一种情况是双黑儿子 p 是黑(BB-2B)的情况。刚刚 p 是任意的情况，现在我们需要特殊讨论：



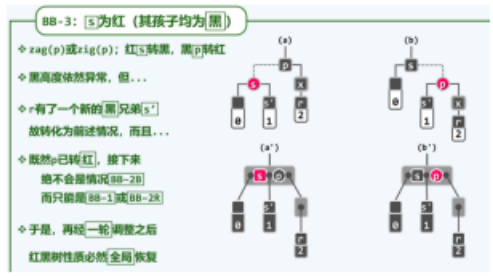
如果 p 是红的，黑高度不变，我们做一次变换颜色即可。现在红的在 s 结点上即可。把颜色改一改即可。这是最快的一种情况。

还有一种是 p 黑，p 黑这一种，就比较麻烦。我们删完之后，本来就不够了，删完之后整棵树就无法维持了，高度必须下降一层。必然会影响其他树的部分，会继续调整（递归情况）。只要有红的，我们就可以调整过来，但是全黑的情况就比较紧凑了，比较麻烦。这样的好处就是我们高度一直在降，最多 $O(\log N)$ 级别完成。

我们通过合并一下，可以把 BB2 合在一起用代码实现;

情况 3：BB3

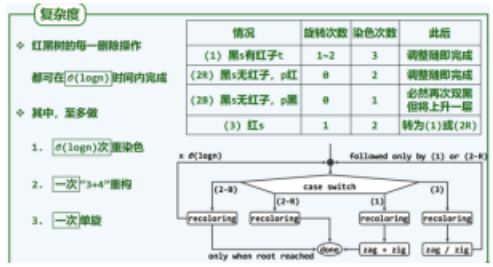
红 s 的情况



这个地方也会用到递归。既然有红色结点，能不能移来移去搞平衡？我们试图这样做，把红变黑，旋转过去（图 a 到图 b 的情况）。但是把图 B 中的 x 删除后，我们发现 s'的黑高度和 r 的黑高度不同，r 少了一个黑高度。但是转到图 B 的情况以后，我们发现了双黑情况，兄弟是黑色的，第一个情况是 BB-1 情况，或者是父亲是红色的，双黑儿子，但是不可能是 BB-2B（全黑情况）。换句话说，我们在这个情况中，旋转后转化到了之前的某种情况。

问题：这种情况下，递归会不会终止呢？会不会来回旋转导致算法不能终止？

解答：我们在写递归的时候，一定要保证在良序集上最终收敛到 0。但是转到 BB2R 和 BB1 做一步就终止了。也可以理解为红黑树的操作的是一个 B 树平衡的特例，B 树都不会递归，这里这么会递归呢。



除了 BB2B 要递归上去，其他的只需要常数级操作。

KD 树

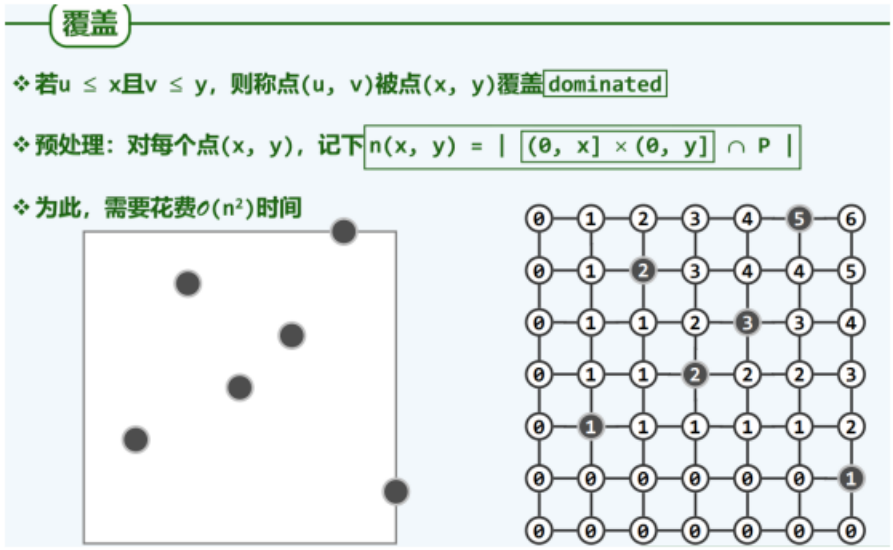
有时候我们希望区间查询，找一个范围的查询。范围查询有两个主要的操作，先做一维的，求在一个区间内，有多少个点（counting），有哪些点（reporting）。如果这个点非常多，甚至内存都放不下，怎么办。Naive 就是排序遍历，如果在这个范围内就记录下来，这是 O(n)。如果点集固定，我们就可以离线处理，先排序再遍历。如果在线给出呢？

1. 蛮力法，依次检查并计数，满足条件则加入到查找结果中。如果操作不好的话，会反复读写磁盘中的数据。导致常数项很大。
2. 计数法，先排序变成有序向量。二分查找定位到最大点，然后向前计数。也是线性复杂度，只和区间长度有关。预处理只需要做一次。Counting 可以简化到 o(logn)

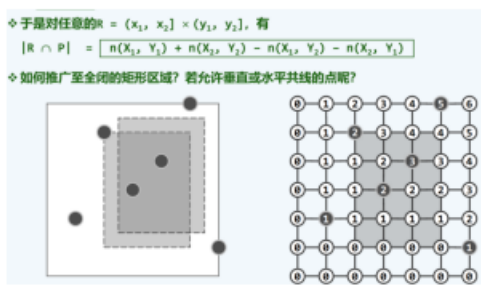
如果是平面版本的区间查找呢？计数法还行得通吗？

1. 容斥原理

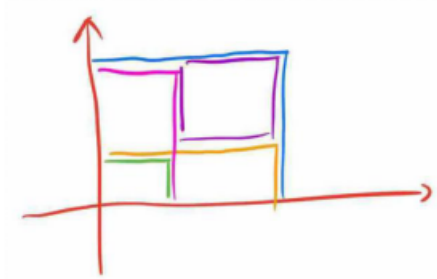
容斥原理的最简单的版本就是 $|A \cup B| = |A| + |B| - |A \cap B|$，同理，可以推广到多维的情况。

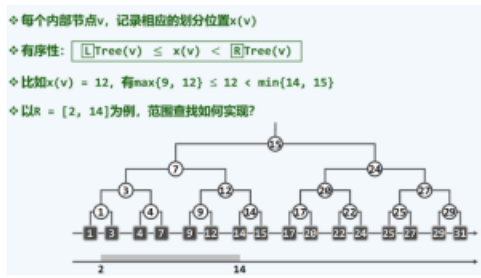


所谓的覆盖就是这个点控制的范围就是<=x，<=y。所以可以通过覆盖操作把任意一个矩形区域求出来。



其实就是 2D 前缀和。

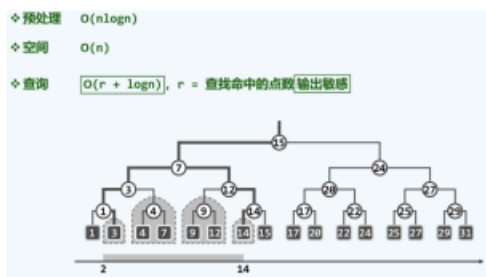




只需要查询头尾结点，其间的就可以比较快的求出来。

这个怎么优化呢？我们知道有一个 LCA 算法（最近公共祖先）。分别找到 2 和 14，我们找到 2 和 14 的最近公共祖先，定位 3 和 14 以后往上找找到 7。找到 LCA 以后，找到左儿子的右子树，和右儿子的左子树，这两个子树就是所有结点的范围。还有一种优化方法，就是排序（穿线）。

LCA 优化的方法的效率，我们可以看到只需要做一次预处理（生成这个二叉树）。

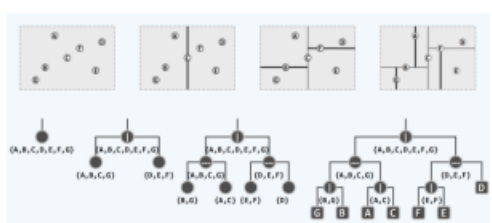


所以这个 BBST 效率还可以，使用 BBST 精确定位点，然后用 LCA 算法输出子树。所以，我们只需要把 LCA 下的子树存进内存里，不需要管其他的结点。

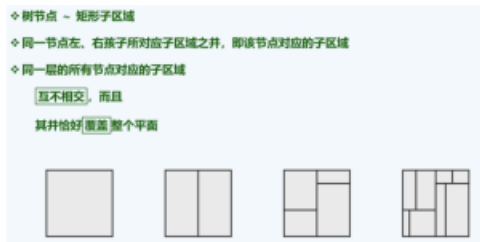
接下来，我们希望直接嵌入多维的信息，只做一遍就把范围中的结点找出来。

切的时候，我们希望切在中位数上，这样每一次保证了一半一半的划分。不停地划分，这个世界的格子就会越来越精细。

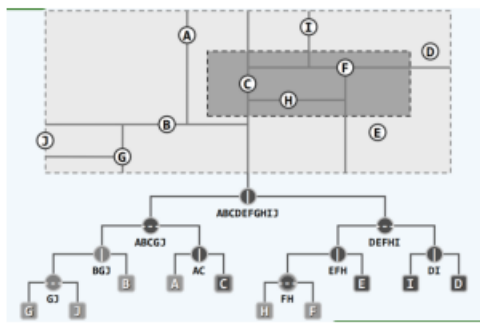
构建 KD 树的实例：



正规子集：不相交的划分，既不重合又能铺满整个平面



任意一个二叉树，可以创造出一个高维空间。
查找算法也很简单，其实就是一个二分查找和递归的过程。这里稍微复杂一点，要看是横着切还是竖着切。如果属于相交的范围之内，就去找对应的子树。



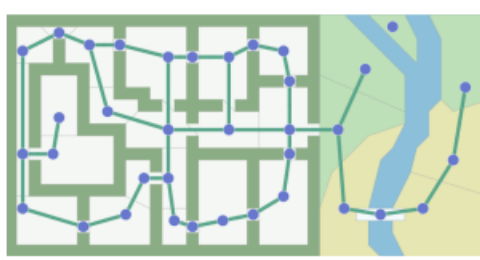
我们发现 C 是在这个范围之内的，所以要递归找 C 的左右子树。第二层 C 的左子树：B 不在范围之内，所以要递归 B 的上面（右子树）。我们继续考虑 A 这个点，我们发现 A 的右边和我们所求区域有相交。我们发现只有 C，所以输出 C。

同样地，查找 C 的右子树，我们最终可以得到 H 和 F。

2021/4/29

A*算法

接下来讲的内容有些未必是书上的。
A*的基础算法还是 dijkstra。要找到最短路径，我们把地图变成一个图。



但是 dijkstra 算法太慢了，需要搜一跳的最短范围、两跳的最短范围、……、一直到包含最终点。需要历史的经验保留下来，帮助我们往前走。我们是否能优化一下，事实上如果我们往终点的反方向走，大概率是徒劳的。所以真正的导航里面是 A*算法，想法是基于 dijkstra 的，我有一个权重 f(n)=h(n)+g(n) h 是历史信息，g 是启发式信息。比如我们可以设置

为到目标的距离，把距离作为一个权重放进来，反方向找 g 显然比较大，不合理。这样我们就得到了 A*算法，如果我们永远设置 g(n)=0，那么 A*就退化为一个 dijkstra 算法。h 和 g 的权重也可以变化，但是如果主要基于预测，找到的未必是最短路径。

<演示视频和具体细节请查阅网站>https://www.redblobgames.com/pathfinding/a-star/introduction.html

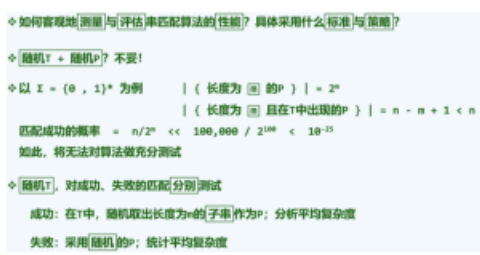
怎么对 A*算法在大数据时代进一步优化呢？自动驾驶的情况下，路特别多，有什么特别的优化方法呢？事先建立很多路标，已经使用离线算法找到了路标两两之间的最短路径。这样 A*可以找到起点和终点最近的一个路标的路径，这样就可以直接导航了。这样虽然未必一定是最优的，但是大数据时代，接近最优就可以了。书上还有基于三角不等式的 ALT 对 A*的优化。

KMP 算法

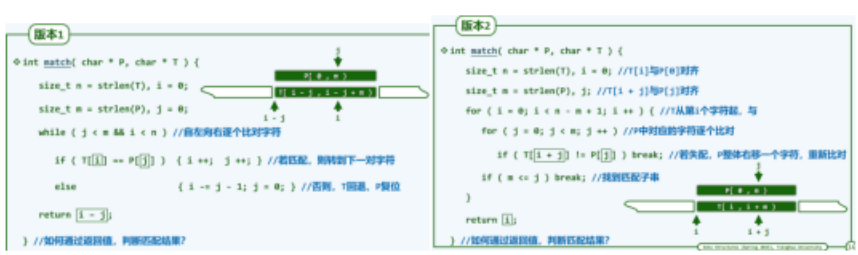
今天我们介绍这本书上的最后一个算法。为什么讲 KMP 呢？因为这个算法代表了一类算法，可以基于它自身的数据结构进行优化。

串、子串、前缀、后缀的定义，此处略。特别地，空串是任何串的子串、前缀、后缀。

现在我们的问题是，如何在一个字符串里面找到一个子串，是否出现(detection)、在哪里出现(location)、出现几次(counting)、各出现在哪里(enumeration)。

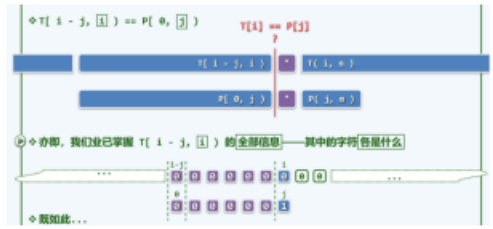


最原始的方法：找 P 是否出现，我们就一个一个比，逐个往后对齐匹配。二重循环，从每个 T 开始，最坏情况下做 m 次匹配。复杂度为 $O(n \cdot m)$



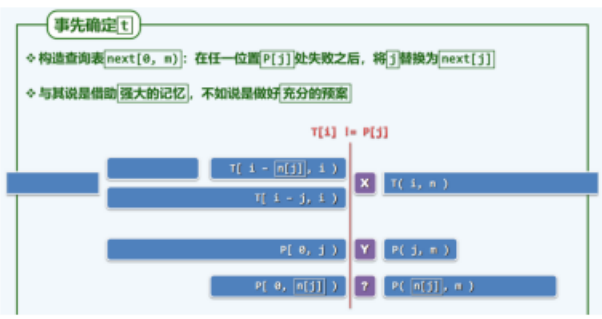
版本二判断匹配结果：如果返回值等于 n-m+1，那么就匹配失败。否则就匹配成功。

蛮力低效的原因:我们没有利用到历史经验，比如 P 和 T 我们只有最后一个字符匹配失败了。蛮力的话是从 i+1 开始比，但是我们已知了 i 开始的 m-1 个都是已经匹配过了，其实是重复地比较。所以 KMP 算法也是根据字符串的信息，建立一个小模型表达字符串，可以建立一个小的自动机进行状态转移。

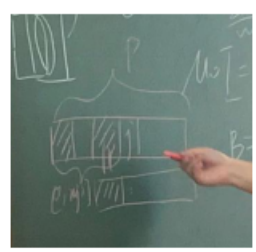


此时，前面的 j 个字符已经匹配成功，当前正在匹配 T[i]和 P[j]。后面还有各自的两个后缀 T(i,n)和 P(j,m)。

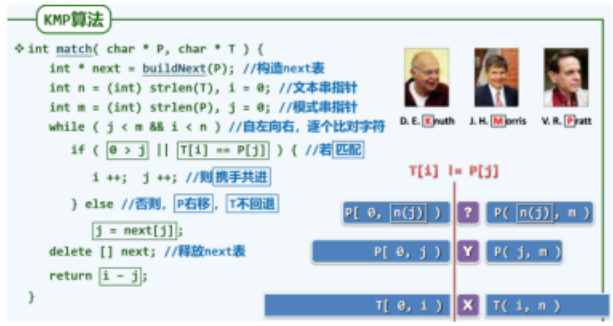
如果 T[i]!=P[j]，前面的 j 个已经比过了，信息都是有的。事实上，我们只需要对 P 自身做一个字符串匹配，从 P 自身提取了一个小的自动机，如果失败了，它会告诉你快速地往后移动。



j 失败了以后，不要从 j-1 开始，可以从 n[j](next j)开始，再判断是否相等。最好的情况可以直接滑过来很多。我们最终的算法就是，每当在 j 匹配失败以后，我们就从 n[j] + 1 这个位置开始匹配。我们发现 next[j]只和 P 本身有关系，我们为什么可以移动到下一个位置呢？因为我们发现 T[i - n[j], i)和 P[0, n[j])是一模一样的，在 P 中，P[0, n[j])和 P[j - n[j], j)是自我匹配的，我们只要对齐以后就可以直接移动了。



这个自我匹配的结构就是字符串 P 的内在结构，如果这个内在结构很好，那么匹配失败的快速移动效率很高。



这个算法代码量非常少，但是非常精要，和之前的算法不同的是，buildNext 和 j = next[j]的位置。

考试的时候：给你一个字符串，让你建 next 表。

我们可以用更简单的方法建表，实际上就是在匹配串 P 的所有前缀中，后缀等于前缀的情况。

3.2 next数组第一个位置的值

next的第一个值无法进行计算，因为它之前没有元素，就没有办法计算相似度。这里是设置默认值的。

那么默认值的要求是什么呢?

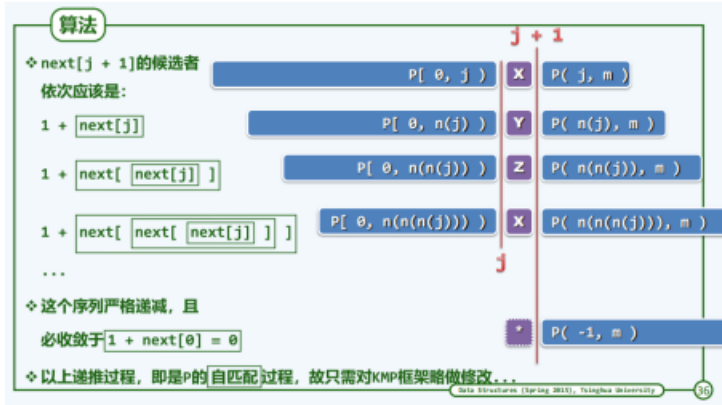
我们已经知道next的实际上就是下标，而next的第一个元素的值不得与现有的下标冲突:

- 如果下标从1开始，则next[1]可以是小于1的任意整数，因此默认使用0。此时ij的初始值分别为i=1;j=0;
- 如果下标从0开始，则next[0]可以是小于0的任意整数，因此默认使用-1。此时ij的初始值分别为i=0;j=-1;

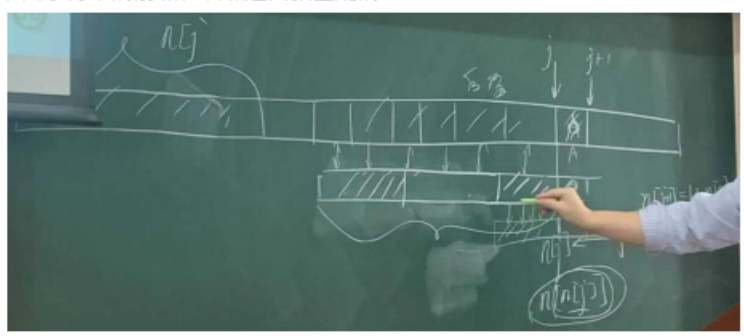
0506

构建 next 表的三种方法

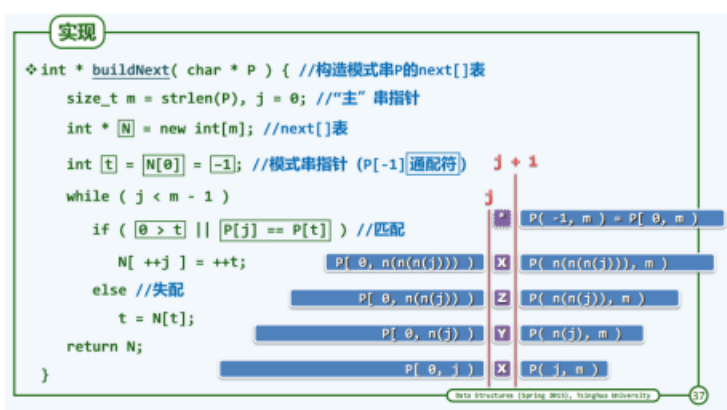
1. 硬编码
2. 自动机 FSM
3. 递归 next 表（把算法复杂度从 nm 降低到 n+m 的精髓）



nj 已经求出来了，nj 告诉我们目前已经匹配字符串的后缀和匹配串的前缀是相等的，也就是待匹配的字符串中的前缀和后缀是对应的。现在到 j+1 时，如果一样，得到匹配的结果，即 nj+1 = 1 + nj。如果不匹配呢？我们把 nj 看成 j，如果对下面的字符串求一个 nj，我就马上得到了下面字符串的前缀和后缀是自我匹配的。



如果再一次移动后匹配的话，可以得到 n[j+1] = 1 + n[n[j]]，如果不匹配的话，再次移动，直至 1 + n[...] 匹配，或者 1 + n[...] = 0。这个算法是很快的，一旦不匹配了就去找 n[n[j]]。代码：

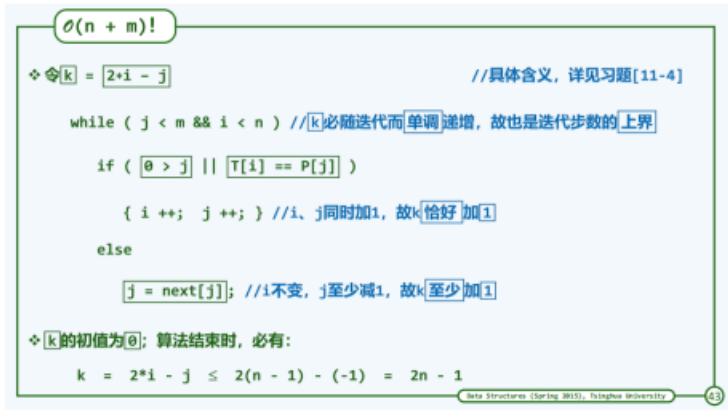


失配的情况，把 N[t]赋值给 t，但是 j 不动，所以会不断迭代，等待匹配或者 t<0 的情况，来

更新 N[j + 1]


c h i n e c h i n c h i l l a
c h i n e c h i n c h i l l a
-1 0 0 0 0 0
c h i n e c h i n c h i l l a
c h i n e c h i n c h i l l a
-1 0 0 0 0 0 0 1 2 3 4
c h i n e c h i n c h i l l a
c h i n e c h i n c h i l l a
-1 0 0 0 0 0 0 1 2 3 4 1 2 3 0 0

e 和 c 不匹配的情况下，我们就会去求 n[4] = n[n[4]] = n[0] = -1，这个匹配例子很重要，需要重点复习看懂。
性能分析：



用了一个 k=2i - j作为指标变量，使用它来跟踪这个循环，其实这个 k 就包括了失败的次数和成功的次数。故，建立 next 表的复杂度降低为 O(n)。加上遍历一遍待匹配字符串就可以匹配成功，此处的复杂度为 O(m)，故 KMP 算法最终的时间复杂度为 O(n + m)

并行编程

单机算法/单核算法已经远远落后于时代了，因为摩尔定律已经失效了。而且机器学习算力需求更大，三四个月就要翻一番。一种解决方法就是把 GPU 加进来，比如 GPU 动辄几千核，这么多核对解决问题是必要的，所以需要并行编程，让这些核能够共同解决一个问题。

并行算法渲染动画的伪代码

并行动画渲染

```
Frame frames[N];                    // 一共有N帧画面

renderFilm():
  如果角色是master:
    将N帧画面等分为M组
    分配M个slave机器，每个机器渲染N/M帧
    等待所有slave机器渲染完毕

  如果角色是slave:
    获取机器的id（id假设从0开始到M-1）
    start ← id * (N/M)
    end ← (id+1) * (N/M)        // 计算每一台机器渲染的范围[start, end)
    for i←start to end-1:
      frames[i].render()        // 对每一帧进行渲染
    通知master执行完成
```

Master 这个主控结点去分配任务，slave 就知道自己应该渲染哪一部分。我们这里不严格区分并发和并行：并发就是一个核，分时复用；并行就是很多核每个核做一件事情。
并行编程是非常困难的，这也是量化交易所对这方面的人才所需要的。量化交易要求机器和

算法性能要求特别高。

```cpp
#include <iostream>
#include <thread>          // C++11的多线程标准库

void foo() { std::cout << "thread 1 ...\n"; }
void bar(int x) { std::cout << "thread 2 ... " << x << '\n'; }

int main() {
  std::thread first(foo);     // 创建一个线程first并调用foo()
  std::thread second(bar,0);  // 创建一个线程second并调用bar(0)

  std::cout << "thread main ...\n";
  // 线程回收
  first.join();                // 等待first线程运行完
  second.join();               // 等待second线程运行完

  std::cout << "foo and bar completed.\n";
  return 0;
}
```

代码8.3. 简单的C++11 多线程程序

join 就是强制主线程同步一下，因为直接创建出来是野线程，操作系统会单独去调度它。

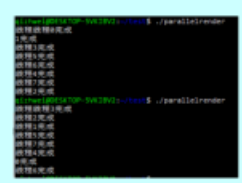
C++ 11 多线程库

```cpp
thread() noexcept;                  // 1. 默认构造器
template <class Fn, class... Args>
explicit thread (Fn&& fn, Args&&... args);      // 2. 初始化构造器
thread (const thread&) = delete;     // 3. 复制构造器[删除]
thread (thread&& x) noexcept;                   // 4. move构造器

void thread::detach();
void thread::join();
bool thread::joinable() const noexcept;
```

(a) 多个核上的并发（并行）　(b) 单核上的并发

这些都是为了解决指针问题，解决掉两个指针指向同一个地方的问题，如果 move 过来，控制权就转移了。joinable 可以判断能不能回收。多线程库就这么简单，把基本的工作做掉了。

```cpp
const int M = 8;  // 线程数
void slaveRenderFilm(int id) {  // 参数是自定义的线程序号
    int start = id * (N / M);
    int end = (id + 1) * (N / M);  // 确定各个线程的负责范围
    for (int i = start; i < end; i++)
        frames[i].render();
    cout << "线程" << id << "完成\n";
}
 void renderFilm() {
    vector<thread> threads;
    for (int i = 0; i < M; i++)
        threads.emplace_back(slaveRenderFilm, i); // 传入线程序号

    for (int i = 0; i < M; i++)
        threads[i].join();
}
```

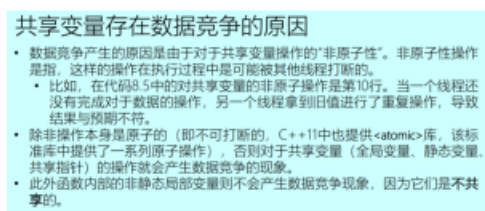


代码8.4. 并行的动画渲染

渲染的代码最后就等待所有线程完成回收。但是我们允许多次发现，光打印语句都有可能被分成两部分，不是我们需要的。因为在同一个进程里面，信息是共享的。我们发现虽然速度提升了，但是并没有按照我们所期望的进行完成。

数据竞争问题，我们发现两个线程把全局变量各加 100000，理论值应该是 200000。原因是什么？我们现在有个变量叫做 countNum，为什么一半被扔掉了呢？内存里应该是 0，线程 1 先读进寄存器里，把它加成 1，还没有写到内存里去的时候，另一个线程也从内存中读进来，把它加成 1。最终写回去的时候，1 被写了两遍。这是操作系统调度的时候出问题了。

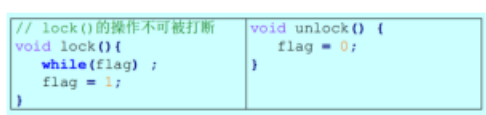
解决方法 1：强制每个线程访问全局变量的时候，确保访问的时候串行化，把这些访问排个队。但是这样会导致并行程序的性能下降。



方法 2：把共享变量改造为局部变量。比如我们可以让线程修改各自对应的全局变量，最终再加在一起，这样又多线程又全速运行，但是没有解决数据共享的问题。

在调试模式的时候，相对比较慢，bug 没那么容易出现。一旦 release 上线了，可能就出问题了。

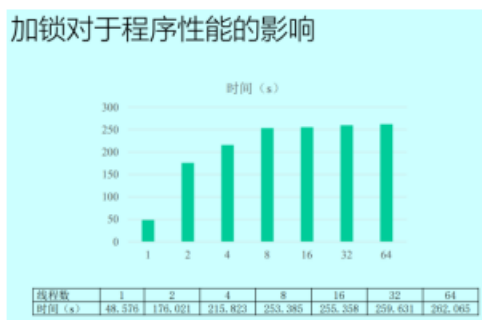
最简单的机制就是加锁，我们加上 mutex（互斥锁）。但是这个程序也不好写，在不同的操作系统或者 CPU 上，也可能出现不同的结果。因为乱序执行的机制，Arm 可能先把 flag = 1 放到前面去执行，但是在 x86 因为变量有依赖关系，不会乱序执行。



代码8.8 加入mutex互斥锁的计数程序

代码8.9. 细粒度mutex互斥锁的计数程序

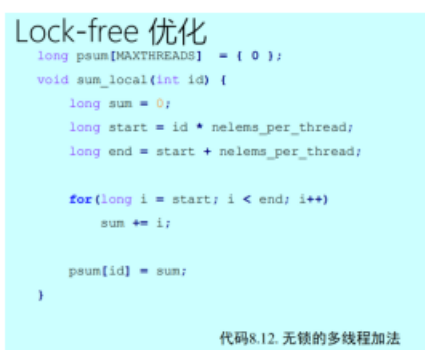
我们发现锁的本身也需要原子访问。如果有 1 个 CPU 在上锁，其他所有 CPU 只能死循环等着，当然这里面还有优化的空间。左图是加了一把大锁，使得退化到了串行的情况。细锁还是让两个 CPU 有互相跑的机会。



线程数	1	2	4	8	16	32	64
时间（s）	48.576	176.021	215.823	253.385	255.358	259.631	262.065

我们发现添加了 CPU 以后，性能反而变慢了。我们为了程序的正确，一定要加锁，但是这个锁加得好不好也很有讲究。

接下来我们看一下 lockfree 的无锁优化。



代码8.12. 无锁的多线程加法

线程数	1	2	4	8	16	32	64
时间（s）	5.160	2.636	1.445	0.846	0.849	0.851	0.853

线程数	1	2	4	8	16	32	64
时间（s）	48.576	176.021	215.823	253.385	255.358	259.631	262.065

买的 CPU 越多，加速越快，但是没有到达线性加速比。同一个程序，加锁与否性能会相差数十倍。所以加锁这个操作需要非常非常慎重。Linux 为了优化锁的效率，有各种各样的锁，比如排队锁、读写锁等，我们要选择合适的锁去优化我们的性能，当然最好的是改成无锁的优化。

第二种锁是编译器上的优化，叫做 lock_guard 锁。

C++11中的lock_guard锁

```
explicit lock_guard (mutex_type& m);
```

在构造器中，传入一个mutex互斥锁的引用，自构造完成之后，互斥锁的便上了锁，直到lock_guard类生命周期结束（退出作用域）调用析构器时，传入的互斥锁便解锁。这样，代码8.9中的counter()函数可以改写如代码8.13给出的形式。

```
void counter() {
        for(int i = 0; i < 100000; i++) {
                // lck的生命周期在这个作用域内
                lock_guard<mutex> lck (mtx);
                countNum++;
        }
}
```

代码8.13. 使用lock_guard锁

在一个很大型的程序中，有时候上了锁没有解锁，有时候解锁两次，所以推荐这种自带作用域的锁。

既然我们已经学了多线程编程，我们怎么改成一个多线程（thread-safe，线程安全）的链表。

并发List-线程安全

```
class List {
public:
    List() { head = NULL; }

    bool insert(int key) {
      try {
        Node *newHead = new Node;
        newHead->key = key;
        {
            lock_guard<mutex> lck(mtx);
            newHead->next = head;
            head = newHead;
        }
        return true;
      }
      catch (bad_alloc &e) {
        cerr << "bad_alloc caught: " << e.what() << endl;
        return false;
      }
    }
```

```
bool lookup(int key) {
        lock_guard<mutex> lck(mtx);
        for (Node *curr = head; curr; curr = curr->next) {
            if (curr->key == key) return true;
        }
        return false;
    }

private:
    struct Node {
        int key;
        Node *next;
    };
    Node *head;
    mutex mtx;
};
```

代码8.14. 并发链表

和一般的链表的区别就是找到所有访问共享变量的地方，只要访问了就老老实实加一把锁。只要有一个地方漏了，一定会出 bug，隐蔽的地方可能某一天才会出 bug。

2021/5/13

多线程的程序有一个很重要的概念就是线程安全。STL 库有可能在多线程中运行，所以很关注 STL 的线程安全。

Eg：并发散列表

```
const int BUCKET = 101;
class Hash {
public:
    bool insert(int key) {
        int bucket = key % BUCKET;
        return lists[bucket].insert(key);
    }

    bool lookup(int key) {
        int bucket = key % BUCKET;
        return lists[bucket].lookup(key);
    }
private:
    List lists[BUCKET];
};
```

数据结构的本地化，把线程的注意力分散到不同的桶中。如果抢到了同一个桶，List 内部维护了锁。如果抢到了不同的桶，自然不需要加锁。在 CPU 之间，也有类似的思路，CPU 会尽可能维护一些自己的数据结构。在看内核源代码的时候，也会看到一些对锁的精细操作。

Eg：并发队列

插入了一个 dummy 指针来存放头结点，这样 head 和 tail 永远不会冲突。

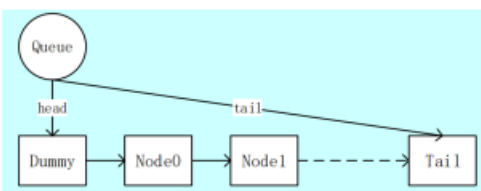


图8.2. Michael & Scott Queue

```
class Queue {
public:
    Queue() {
        Node *dummy = new Node( 0, NULL );
        head = tail = dummy;
    }

    void enqueue(int key) {
        Node *tmp = new Node( key, NULL );
        lock_guard<mutex> lock(tailMtx);
        tail->next = tmp;
        tail = tmp;
    }
```

执行模式	串行	并行
执行时间（秒）	3.737	2.028

```
    bool dequeue(int *value) {
        lock_guard<mutex> lock(headMtx);
        Node *tmp = head->next;
        if (tmp == NULL) return false;

        *value = tmp->key;
        delete head;
        head = tmp;
        return true;
    }
private:
    struct Node {
        int key;
        Node *next;
    };

    Node *head, *tail;
    mutex headMtx, tailMtx;
};
```

我们非常小心地加了一把锁，只要是对全局变量的操作，在此之前就要开始保护。

死锁问题

死锁通常是偶然发生的。比如数据库检测某段时间没响应了，就杀死以后，重新进行事务。（鸵鸟算法 https://blog.csdn.net/u011518120/article/details/51897845）

Eg：死锁的例子

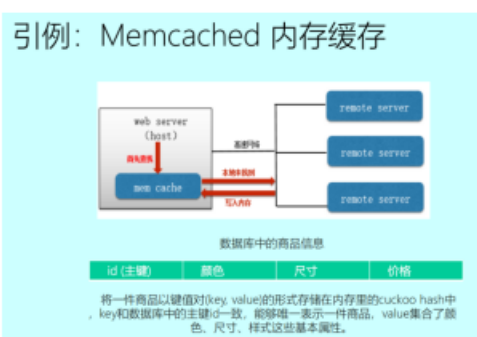
通常情况下，foo 抢占 mtx1，再抢占 mtx2；bar 抢占 mtx2，再抢占 mtx1。
但是考察 foo 已经抢占 1，bar 抢占 2 后，两个线程尝试抢下一把锁，但是进入了死锁状态。现代 OS 先回等待一会期待有线程放出锁，一段时间后，OS 会让线程进入 sleep 状态，不释放资源，程序停止在那里。
方法 1：重启一次，在现代 OS、数据库中，通常都是使用时，看门狗发现死锁然后重启。OS 中几千万行代码，把每个锁都检测一遍是不可能的。

代码的解决方法：
1. foo 和 bar 都先抢占 mtx1 再抢占 mtx2。
2. trylock 如果锁了一段时间发现不对劲就放弃这把锁。

还有一个问题是 livelock（活锁， https://www.zhihu.com/question/20566246 ），虽然活着，但是一直在死循环；或者调试的时候一切正常，运行的时候每隔一段时间出问题。

换句话说，这涉及到一个程序终止性的问题。

Cuckoo hash



常规的方法是本地维护一个 cache，保存着远程数据的备份，如果本地能找到，就直接返回，否则就从网络资源中再去找。
这个数据结构通常是读比较多（购物网站的游览）、写比较少（购物网站的下单）

两种普通哈希：线性探测哈希、链式哈希（冲突时就放到链表中），为什么在这个场景中不适用呢？在百万级的大并发下，顺序查找或者遍历操作可能就加上了一把大锁，用户越多性能越慢。局部形成了热点，使得性能急剧下降。

布谷鸟哈希的思想：

x 有 2 个哈希函数 h1 和 h2，

查找(get)操作：在 cuckoo hash 中，因为一个键仅可能出现在 table 中的 h1(k)位置或者 h2(k)位置，所以查找时仅需要探测这两个位置。

插入(put)操作：和其它 hash 方法一样，cuckoo hash 避免不了插入时的冲突。对于某个键 k，如果 h1(k)位置发生冲突，则查看 h2(k)位置，为空则将 k 插入至 h2(k)位置，但如果 h2(k)非空呢？

如果两个位置都满了怎么办？我们需要把 h1(x)或 h2(x)对应的一个 key 踢走，比如提走了 h1(y)。被踢掉的 y 会尝试插入到 h2(y)，进行递归。布谷鸟哈希有各种各样的变体，比如 3 个哈希函数。

为什么这样并发就能特别好呢？首先，put 会一直在找空的地方，并且占据空的地方，使得空间利用率特别高，能占满整个空间。哪怕空间占满了，也只需要做两次哈希。

但是如果我们为了避免有环，需要遍历一遍有没有环，这会降低我们的性能。

类的结构代码：

```
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <thread>
#include <mutex>
#define SIZE (50)
namespace cuckoo{
typedef int KeyType;
class Cuckoo{
protected:
    std::mutex mtx;
    KeyType T[SIZE];
    // hash key by hash func 1
    int hash1(const KeyType &key);
    // hash key by hash func 2
    int hash2(const KeyType &key);
    // find key by hash func 1 in T, exist return key otherwise 0
    KeyType get1(const KeyType &key);
    // find key by hash func 2 in T, exist return key otherwise 0
    KeyType get2(const KeyType &key);
    void bt_evict(const KeyType &key, int which, int pre_pos);
public:
    Cuckoo();
    //~Cuckoo();
    KeyType get(const KeyType &key);
    void put(const KeyType &key);
};
}
```

并行 GET

并行Get

共享变量为何没有锁保护？

```
static const int TOTAL = 10;
int main(int argc, char* argv[]){
    Cuckoo test;
    // single-thread to put [1, TOTAL]
    for(int i = 1; i <= TOTAL; ++i){
        test.put(i);
    }
    // create multiple threads to get in parallel
    std::vector<std::thread> threads;
    threads.clear();
    for(int i = 1; i <= TOTAL; ++i){
        threads.emplace_back([&](int thread_id){
                printf("thread: %d get %d\n", thread_id, test.get(thread_id));
        }, i);
    }
    for(int i = 0; i < TOTAL; ++i){
        threads[i].join();
    }
    return 0;
}
```

因为 Cuckoo test 是只读不写的，自然不需要加锁。

Put

```
template <typename T>
inline void swap(T* a, T* b){
    assert(a != NULL && b != NULL);
    T tmp = *a;
    *a = *b;
    *b = tmp;
}
void Cuckoo::put(const KeyType &key){
    if(key == 0){
        printf("invalid key\n");
        return;
    }
    if(get(key) != 0){
        printf("duplicate key, put fail\n");
        return;
    }
    // basic way
    if(T[hash1(key)] == 0){
        T[hash1(key)] = key;
    }else if(T[hash2(key)] == 0){
        T[hash2(key)] = key;
    }else{ // two place for one certain key has been occupied, need evict others
        // basic way
        KeyType evicted = key;
        // determine which pos hash1 or hash2 to put key
        // 0 is hash1, 1 is hash2
        int which = 0;
        // first evict key in hash1
        int idx = hash1(evicted);
        // != 0 means place has been occupied
        // if there is a cycle, maybe cannot terminate
        int pre_pos = -1;
        while(T[idx] != 0){
            printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
            swap(&evicted, &T[idx]);
            pre_pos = idx;
            which = 1 - which;
            idx = (which == 0)?hash1(evicted):hash2(evicted);
        }
        printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
        T[idx] = evicted;
    }
}
```

串行 put，左边一列相对比较容易。右边的逻辑是考察需要被踢走的元素。首先踢走 hash1 所占据的元素。如果找到了新家，T[idx] = 0，那很好，就放进来。如果没有找到新家，继续踢新的位置的元素，所以存放新元素有一个 swap 操作。which 是主要为了 01 轮流踢 hash1 对应的位置或 hash2 对应的位置。

单线程操作的时候还好，一直踢就行。多线程的情况，踢走的元素保留在 evicted 临时变量中，而被踢走的位置还为空。这样可能会导致另外一个线程读取这个位置的时候得不到结果。我们希望把被踢走的信息保留起来，所以有基于回溯的实现。

基于回溯的实现

假定产生的踢出序列为A->B->C->D->nil

先踢出D，依次向上直到A，我们便可以发现：在保证将某个键插入到指定位置的操作是原子的前提下，就可以确保这些元素始终在hash table里

```
void Cuckoo::bt_evict(const KeyType &key, int which, int pre_pos){
    int idx = (which == 0)?hash1(key):hash2(key);
    // basic case: find a empty pos for the last evicted element
    if(T[idx] == 0){
        printf("evicted key %d from %d to %d\n", key, pre_pos, idx);
        T[idx] = key;
        return;
    }
    printf("evicted key %d from %d to %d\n", key, pre_pos, idx);
    KeyType cur = T[idx];
    // first evict latter elements
    bt_evict(cur, 1 - which, idx);
    T[idx] = key;
}
```

刚才的问题是踢到临时变量里，该元素在 hash 表中就消失了。我们保证插入操作是原子的，是不会被打断的。

递归是：调用 A->调用 B->调用 C->调用 D，我们发现 D 有空位，我们把 D 放到新位置；回溯把 C 放到 D 的位置；……以此类推放置整个递归调用栈上的所有元素。这样保证了无论在每个调用栈中，有并行的查询请求出现，访问的哈希表中都保留的全部的元素信息。这样的更改在单线程中是不被欣赏的，因为把一个快速的循环修改为的复杂、缓慢的递归。

并行Put

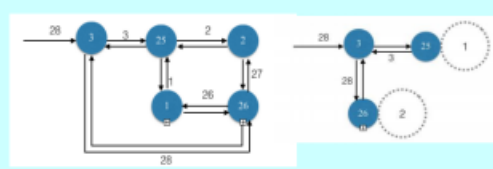
```
void Cuckoo::put(const KeyType &key){
    if(key == 0){
        printf("invalid key\n");
        return;
    }
    if(get(key) != 0){
        printf("duplicate key, put fail\n");
        return;
    }
    // basic way
    if(T[hash1(key)] == 0){
        T[hash1(key)] = key;
    }else if(T[hash2(key)] == 0){
        T[hash2(key)] = key;
    }else{ // two place for one certain key has been occupied, need evict others
        // lock way
        // need lock for write-operations
        std::unique_lock<std::mutex> lck(mtx);

        KeyType evicted = key;
        // determine which pos hash1 or hash2 to put key
        // 0 is hash1, 1 is hash2
        int which = 0;
        // first evict key in hash1
        int idx = hash1(evicted);
        // != 0 means place has been occupied
        // if there is a cycle, maybe cannot terminate
        int pre_pos = -1;
        while(T[idx] != 0){
            printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
            swap(&evicted, &T[idx]);
            pre_pos = idx;
            which = 1 - which;
            idx = (which == 0)?hash1(evicted):hash2(evicted);
        }
        printf("evicted key %d from %d to %d\n", evicted, pre_pos, idx);
        T[idx] = evicted;
    }
}
```

这个并行 put 没用到 bt_evict，因为在上面加了一把大锁。这个大锁是比较冗余的，可以思考一下是否能细化它的粒度呢。

死循环

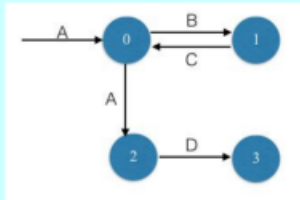
```
#include <vector>
#include "cuckoo.cpp"
using namespace cuckoo;
static const int TOTAL = 28;
int main(int argc, char* argv[]){
    Cuckoo test;
    // single-thread to put [1, TOTAL]
    for(int i = 1; i <= TOTAL; ++i){
        test.put(i);
    }
    return 0;
}
```

```
1 evicted key 28 from -1 to 3
2 evicted key 3 from 3 to 25
3 evicted key 2 from 25 to 2
4 evicted key 27 from 2 to 26
5 evicted key 26 from 26 to 1
6 evicted key 1 from 1 to 25
7 evicted key 3 from 25 to 3
8 evicted key 28 from 3 to 26
9 evicted key 27 from 26 to 2
10 evicted key 2 from 2 to 25
11 evicted key 1 from 25 to 1
12 evicted key 26 from 1 to 26
13 evicted key 28 from 26 to 3
14 evicted key 3 from 3 to 25
15 evicted key 2 from 25 to 2
```



哈希了半天找不到出路，所以就无法终止了。

可以终止的环



只有当一个键的两个可选位置都各自形成一个环结构时，才会导致整个过程无法终止

检测循环路径的方法也比较简单，可以预先设定一个阈值(threshold)，当循环次数或者递归调用次数超过阈值时，就可以认为产生了循环路径。一旦发生循环路径之后，常规方法就是进行rehash操作

New hash functions are chosen, and the whole data structure is **rebuilt** ("rehashed")

哈希的话，必须要两个位置 h1，h2 都形成一个环才会导致上述的死循环。判环的话可以看是不是访问了同一个结点，但是非常耗时，所以比较朴素的算法是设置一个访问阈值的上限。Rehash 就相当于用新的哈希函数把哈希表重建一遍。也是使用了鸵鸟算法，对于小概率事件就让它发生，发生了再去修。