**一、算法基础**

**Q：LRU算法实现cache，用什么数据结构？**

A： 利用链表和hashset。

当需要插入新的数据项的时候，如果新数据项在链表中存在（一般称为命中），则把该节点移到链表头部，如果不存在，则新建一个节点，放到链表头部，若缓存满了，则把链表最后一个节点删除即可。在访问数据的时候，如果数据项在链表中存在，则把该节点移到链表头部，否则返回-1。这样一来在链表尾部的节点就是最近最久未访问的数据项。

**Q：深度优先搜索和广度优先搜索比较**

A： 深度优先搜索：按树的深度进行搜索，最深的节点最先被搜索到。

广度优先搜索：按照树的层次进行的搜索。

**对比：**

a、广度优先适用于所有情况，深度优先在处理无穷分支的问题时，可能会陷入，无法找到目标点，所以深度优先搜索是不完备的。

b、在深度已知，且树体系庞大时，深度优先搜索的效率往往更高。

c、深度优先占用内存较小。深度优先使用栈的方式，内存大小为O(logn)，在处理叶子节点时，栈最大；广度优先采用队列，在树宽度最高的位置占用内存最大，占用内存大小为O（n）。

**在舱室识别算法中，使用广度优先的原因：**

1、广度更直观、一层一层便于调试；

2、每个舱室的边界一般不到300个，数据量不大。此时广度和深度优先搜索占用的内存大小相差不大。

3、在舱室识别中，需要排除已探测块，使用广度优先搜索只需要跟当前层和上一层进行对比就可以判断重复。深度优先需要跟所有已探测位置比对，速度慢。

<http://blog.csdn.net/sweetgum2012/article/details/46739789>

**Q：贪心、回溯、动态规划、分治、分支界限**

A： 贪心：通过解局部最优解来达到全局最优解；贪心算法能得到最优解的前提是，某状态以后的情况不会影响当前选择的最优性。（例如：哈夫曼编码）

回溯：采用深度优先搜索，在搜索尝试过程中寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。（例如：八皇后问题）

动态规划：将求解的原问题转化成求若干子问题，按照顺序求解子阶段，前一问题的解，为后一子问题提供决策信息。（例如：0-1背包问题）

分治：将复杂问题递归划分为子问题，直到子问题可以直接求解。（例如：归并排序）

分支界限：采用广度优先，从一节点出发，然后生成全部儿子节点；再从儿子节点出发继续搜索。（例如：求单源最短路径）

**Q：最小生成树算法**

A： 生成树是指一个连通图，它用n - 1条边连接了图中的全部n个点。最小生成树是指这些生成树中，边的代价和最小的生成树。

**算法：**

**Kruskal算法**（加边法）：

将边按照代价排序，依次选择代价最小，且两端点属于两个树的边；每次选择将边连接的两个树合成一棵树；直到找到n - 1条边。

**Prim算法**（加点法）：

初始化两个集合：已探测到的点集合U、未探测点集合V。查找代价最小的边，将两端点从V中删除，添加到U中；从V中选择到U中各点距离最小的边，并将该点从V移到U中，直到V为空。

**Q：最短路径算法**

A： a、深度优先搜索、广度优先搜索

b、Dijkstra算法，和广度优先搜索的思想一致（无法解决负值边情况）

将节点分为已探测、待探测、未探测三类。从目标节点开始，将目标节点可以探测到的视为待探测。每次从待探测节点中取出一个进行探测，并将其标记为已探测。每次探测，更新到各节点的距离权值，直到所有节点都探测到。

c、Floyd算法（时间复杂度O（n^3），空间复杂度O（N^2））

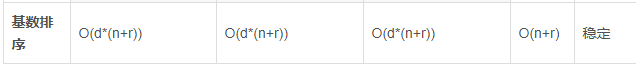
d、SPFA算法

**二、STL**

**Q：各排序算法的比较**

A：





其中，d 为位数，r 为基数，n 为原数组个数。

<http://blog.chinaunix.net/uid-21457204-id-3060260.html>

注：1、快排需要递归，递归深度为O（logn），所以空间复杂度也为O（logn）

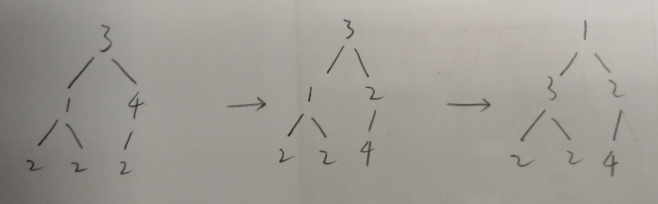
2、选择排序在同一数组中进行时，不稳定。例如5，5，2排序后2，5，5

<https://www.zhihu.com/question/20926405>

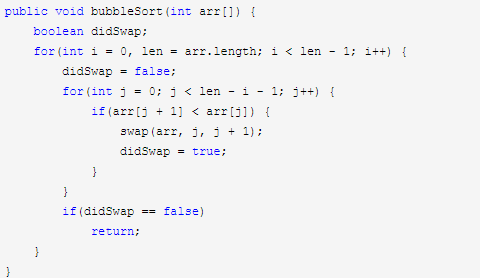
3、快排的最坏情况发生在主元素选取不当时，比如1，2，3，4，5，6的序列，从小到达排列，每次都选择首元素做主元素。

4、快排是不稳定的，因为在从左向右和从右向左查找待交换位置时，遇到相同元素会停下来。例如：5a，5b，5c，3，4；当基准是4时，3和5a交换，4和5b交换，最后状态为3，4，5c，5a，5b。

5、堆排是不稳定的，例如：



6、冒泡排序最优复杂度是O（n）。将传统冒泡排序过程进行优化，当一轮比较过程中，没有出现一次逆序时，认为已经有序，不需要再进行下一步搜索。



<https://www.cnblogs.com/melon-h/archive/2012/09/20/2694941.html>

7、如何实现稳定的选择排序：在一轮扫描过程中，将需要交换的元素放在单独的空间中，并将右侧不需要移动的数据往右靠，左侧不需要移动的数据往左靠。遍历完成后，记录终止位置，将对应需要交换的数据拷贝到终止位置两侧。

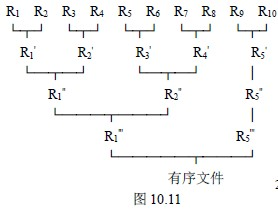
<https://blog.csdn.net/liuchenjane/article/details/72902325>

**Q：外部排序**

A： a、按可用内存的大小，将外存文件分片，分别对子文件进行内部排序。排序后放回磁盘。

b、对各有序的分片进行多次归并，使其在更大范围内有序，直至全部有序。

c、从2路到k路，增大k可以减小外存读写时间，但是会增加每次选择元素时的比较次数。



**外部排序的优化：**

采用败者树，K路归并时，每次选择元素的时间从O（K）减为O（logK）。

<http://www.360doc.com/content/15/0719/22/22633806_486084820.shtml>

**Q：快排的优化**

A： a、三数取中选择枢纽；

b、当快排序列的长度缩减到一定大小后，使用插排；

c、如果数组有大量重复元素，采用三向切分。将数组切分成小于、等于和大于切分元素的三部分；

d、尾递归优化，缩减堆栈深度（方式：将if改成while，以循环代替递归，让栈的深度从O（n）缩减为O（logn））；

**STL中的sort：**

a、用三数取中选择枢纽；

b、当区间小于16时，使用插排；

c、当区间大于16，且递归深度超过lg（n）时，改用堆排；

d、未超过lg（n）时，尾递归。

<https://www.cnblogs.com/babybluevino/p/3714022.html>

**Q：STL各数据结构的底层实现**

A： **序列式容器**

a、vector：连续空间，只需要记录三个迭代器；动态空间增长

b、list：环状双向链表，在环状链表尾端加上一个空白节点，即last迭代器。这样一个指针即可表示一个链表；

c、deque：双端队列。以多段连续空间组合而成。由中控器保存每个分段的头指针。

d、stack：配接器，由deque配接而成；用list也可以实现stack。stack没有迭代器。

e、queue：配接器，由deque配接而成；用list也可以实现queue。queue没有迭代器。

f、heap：通过list或者完全二叉树实现。没有迭代器。（完全二叉树用vector实现）

g、priority\_queue：优先队列，以heap实现；

h、slist：单向链表。

**关联容器**

a、set、map、multiset、multimap：以红黑树实现；

b、unordered\_set、unordered\_map：以hashtable实现。

**迭代器失效问题：**

a、对于vector，删除元素，元素之后的所有迭代器失效；增加元素，如果引起内存重新配置，所有迭代器失效；但是erase操作会返回下一个可用迭代器。

b、对于deque，在deque首尾插入元素不会导致任何迭代器失效；在deque中间插入或者删除操作会导致指向该容器元素的所有迭代器全部失效。

c、list、set、map等做删除操作，指向该删除节点的迭代器失效，但是其它位置正常。

**Q：deque**

A： **deque的数据结构：**

a、由多段连续空间组成，中控器保存每个分段的头指针；

b、迭代器中包括三个指针：指向当前位置的指针、指向当前分段的头的指针、指向当前分段的尾的指针、指向中控器中对应该分段的位置的指针；

c、deque的模板类定义中，包括以下成员：指向中控器的指针、指向第一个分段第一个元素的指针、指向最后一个分段最后一个元素的指针、中控器的大小。

**deque和vector的比较：**

a、deque允许常数时间内对任意位置的插入操作；

b、deque没有容量观念，它随时可以增加一段空间链接起来。不存在vector的重新配置空间问题。

c、deque的空间整体上不连续，所以其迭代器不是普通指针，进行元素查找、随机访问的效率不如vector。在对deque的排序操作中，将deque复制到vector中，用vector排序后复制回去，这样效率往往更好。

**Q：list和slist比较**

A： a、在STL中，这是两种不同的数据结构。list的迭代器是双向迭代器，slist迭代器是单向迭代器；

b、stl的插入都是在指定位置之前插入，list在任何位置插入和删除元素的效率都是O（1），但是由于slist无法直接获取上一个元素，插入和删除时需要遍历链表查找前置元素。

c、slist提供了erase\_after、insert\_after用于在指定位置之后插入；

**Q：map和hash\_map的区别；**

A： a、内部实现上，map采用的是红黑树，unordered\_map采用的hashtable；

b、map需要重载<，unordered\_map需要重载==并且指定哈希函数；

c、红黑树本身是一种二叉搜索树，所以map中的元素是有序的，unordered\_map中的元素无序。

d、从插入、查找、删除的效率上看，map为O(logn)， unordered\_map为O(1)；

e、从占用空间上看，unordered\_map比map要大；

f、hash\_map需要考虑hash冲突和扩容问题；

当元素较多时，使用hash\_map效率较高。但如果对内存的使用较为严格，或者当元素数量较少时，采用map效率更高。

**Q：hash表处理冲突的方式有哪些，这些方式分别适用于哪些场合？hash\_map内部如何处理冲突？**

A： **处理冲突的方式：**

a、开放寻址法：采用线性探测再散列、二次探测再散列（k^2），装载因子不应该超过0.5;

b、再散列法：建立多个hash函数，发生冲突时，使用下一个hash函数；

c、拉链法（链地址法）：在冲突的位置建立一个链表，然后将冲突的元素插入到链表尾端，该方法装载因子应该尽量接近1；

d、多级哈希：采用多个数组来处理冲突，适用于大数据处理。

e、将冲突域单独保存，以key-value的形式放在内存中

**适用场合：**

开放寻址法实现起来更简单，但是内存开销较大，适用于数据量较小的情况；

拉链法可以保证很高的装载因子，但是有额外的指针开销，适用于数据量大，数据分布较均衡的情况；（hash\_map内部使用拉链法处理冲突）

**Q：如何实现hashtable，hash在高并发环境下如何扩容？**

A：**实现：**

采用vector和list配合实现hashtable；

**hashtable的扩容：**

a、在SGI STL中，当装载因子大于等于1时，对哈希表进行扩容。开辟下一个质数大小的空间，然后将原来哈希表中的元素hash到新桶中。缺点：容量扩张是一次性完成的，期间要花很多时间转移hash表中的所有元素，当哈希表中元素较多时，扩张需要等候很长时间。

b、分摊转移：在redis数据库中，使用两个哈希表进行扩容和转移工作：当从第一个hash表的装载因子为1时，假设要往字典里插入一个元素。首先为第二个hash表开辟2倍第一个hash表的容量。同一时候将第一个hash表的一个非空桶中元素所有转移到第二个hash表中。然后把待插入元素存储到第二个hash表里。继续往字典里插入第二个元素，又会将第一个hash表的一个非空桶中元素所有转移到第二个hash表中，然后把元素存储到第二个hash表里……直到第一个hash表为空。这样的策略就把第一个hash表全部元素的转移分摊为多次转移，并且每次转移的期望时间复杂度为O(1)。

c、多级哈希。增加数组层数。

在高并发环境下，为了防止多个进程因为hashtable扩容而阻塞，最好采用分摊转移进行扩容。

**Q：SGI STL中，哈希表数组的容量为什么要选用质数？**

A： 通常通过hash求值后除以哈希数组容量的方式确认数组位置。当集合数据比较均匀的情况下，数组容量取任意值都没问题。当数据不均匀时，如果数组容量为合数，则当hash结果为合数的公因数时，其最终的位置将具有一定的局限性。这将导致最后的分布并不均匀。

比如hash的结果大部分是偶数时，这时候如果HASH数组容量是偶数，容易使原始数据HASH后不会均匀分布：

2 4 6 8 10 12这6个数，如果对 6 取余 得到 2 4 0 2 4 0 只会得到3种HASH值，冲突会很多。

如果对 7 取余 得到 2 4 6 1 3 5 得到6种HASH值，没有冲突。

**Q：hash函数如何保证冲突最小**

A：降低装载因子，改进hash函数

**Q：AVL树和红黑树**

A： a、AVL树要求高度平衡，从根节点到任何一个叶子节点路径的最长和最短长度差不超过2；红黑树是一种近似平衡，从任意节点到任何一个叶子节点的最长路径长度为最短路径长度的两倍；AVL树在不平衡时，只能通过旋转来解决；红黑树还可以通过改变节点颜色调整。

b、从查找效率上来说，两种平衡树的时间开销都是O（logn），但是最坏情况下，红黑树查找的深度是AVL树的两倍；

c、增加节点时，红黑树和AVL树最多通过两次旋转实现。红黑树还需要调整部分颜色；

d、删除节点时，AVL树最多需要O（logn）次旋转，最坏情况下，需要从叶子节点一直调整到根节点；红黑树最多需要三次单旋转，O（logn）次颜色变换。

红黑树和AVL树的选用：

对于以查询、增加为主的情况下，AVL树要优于红黑树；但是当存在大量删除操作时，红黑树性能要高于AVL树。

<https://www.cnblogs.com/gofighting/p/5437998.html>

**Q：海量数据的存储和查找？**

A：采用BItmap、hashtable、布隆过滤器；

详见：程序员进阶-》数据结构与算法-》BitMap和布隆过滤器

**Q：布隆过滤器**

A： 优点：不需要存储key，节省空间；

缺点：

a、可能存在错判key的情况；

b、不支持对集合元素的删除更新。

**Q：STL有哪些优缺点，为什么有时候很慢？**

A： **优点：**

a、代码易读性强；

b、除了容器和迭代器外，还集成了很多在时间和空间上优化地很好的算法；

**缺点：**

a、STL线程不安全。

b、对算法基础有一定的要求，容器选用不当可能会导致速度很慢（例如对vector进行大量的插入、删除操作）、某些敏感操作还会导致异常（例如迭代器失效）；

b、STL不开优化时会很慢，在编译器设置O1优化下，stl中的成员函数不会优化成inline函数。

**为什么有时候很慢：**

a、需要扩容，例如vector、hashtable；

b、对于vector和deque的erase操作，可能会导致移动内存；

b、没有开优化

**Q：STL是不是线程安全的？**

A： **STL的对多线程的支持仅限两点**：

a、多个读取者是安全的。即多个线程可以同时读取一个容器中的内容；

b、对不同容器的多个写入者是安全的。即多个线程对不同容器的同时写入合法；

当对同一容器有并发的读写操作时，需要通过加锁的方式实现同步。

**需要同步的操作：**

a、每次调用容器的成员函数期间需要锁定；（insert、earse）

b、容器返回迭代器时需要锁定；（begin（）、end（））

c、容器在调用算法执行器需要锁定；（find（））

<https://www.cnblogs.com/ztteng/archive/2013/11/07/3411738.html>

**Q：特化和偏特化？**

A： 偏特化：对模板类进行部分参数特化，例如特化为const版本、指针版本；

全特化：将模板类特化成特定的类型的类，例如特化成int型版本。

**Q：红黑树，插入和删除节点，左旋、右旋、双旋转**

A： **红黑树的特性：**

a、每个节点或者是黑色，或者是红色。

b、根节点是黑色。

c、如果一个节点是红色的，则它的子节点必须是黑色的。

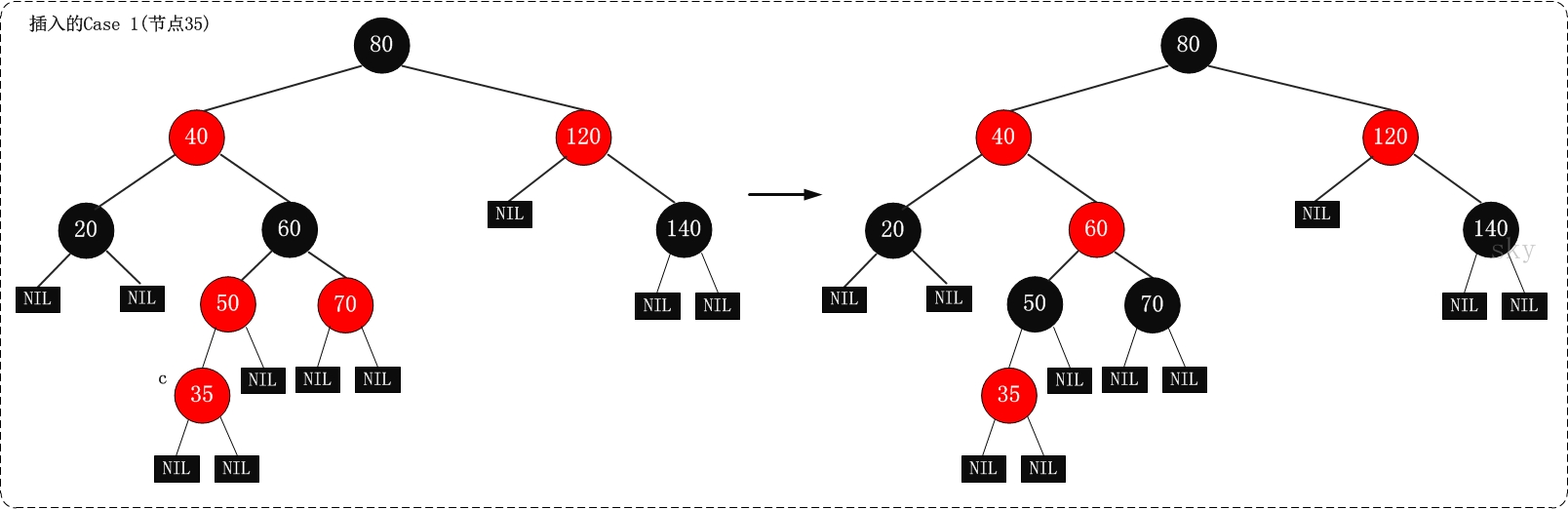
d、从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

**插入**

在向红黑树上插入节点时，如果父节点为黑色，直接插入节点，设置节点颜色为红色。如果父节点为红色，则需要进行以下处理：

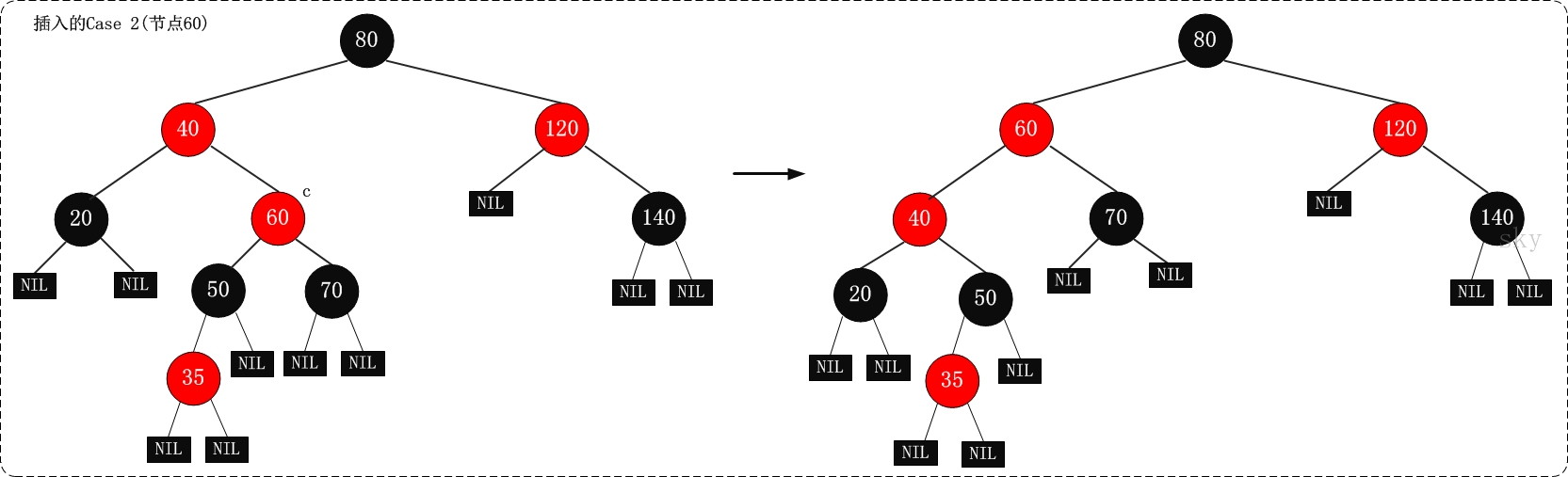
a、叔叔节点为红色

设置父节点和叔叔节点为黑色，祖父节点为红色，自己为红色插入。如果曾祖父节点为红色，进入b、c。



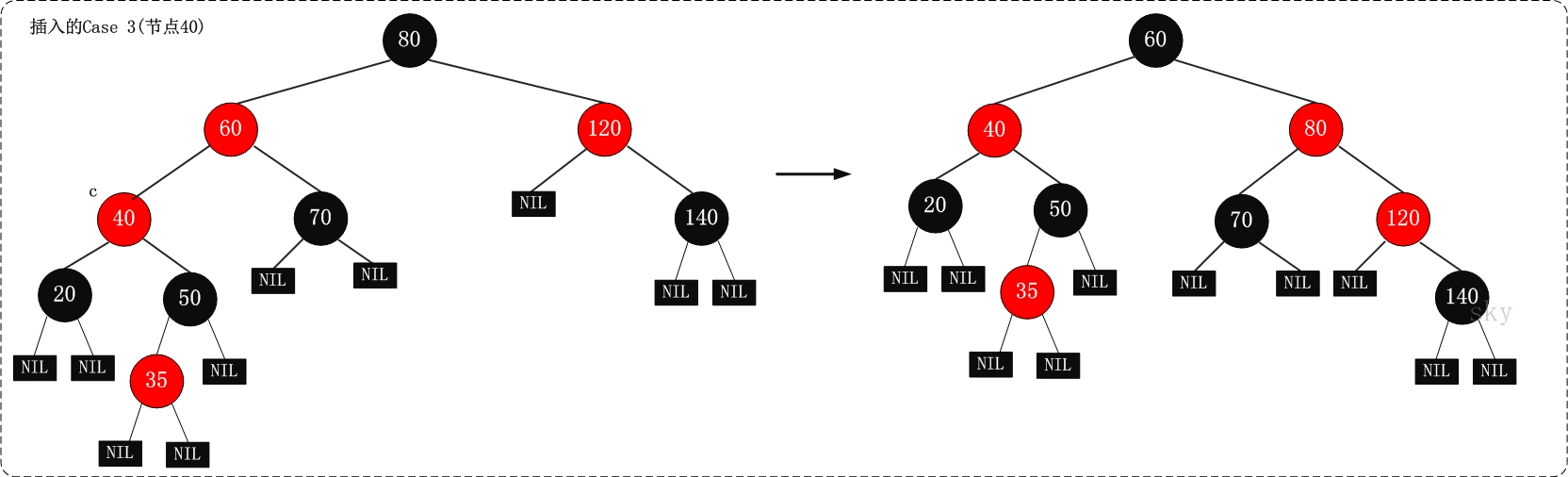
b、叔叔节点为黑色，内侧插入（如上图60节点）；

以父节点为支点左旋；



c、叔叔节点为黑色，外侧插入（如上图40）

将“父节点”设为“黑色”，将“祖父节点”设为“红色”，以“祖父节点”为支点进行右旋。



**删除**

删除操作分两步进行：

1、将红黑树当作一颗二叉查找树，将节点删除。

a、被删除节点没有儿子，即为叶节点。那么，直接将该节点删除就OK了。

b、被删除节点只有一个儿子。那么，直接删除该节点，并用该节点的唯一子节点顶替它的

位置。

c、被删除节点有两个儿子。那么，先找出它的后继节点；然后把“它的后继节点的内容”复制

给“该节点的内容”之后，删除“它的后继节点”。

2、通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。

**Q：字典树的构造及其优化与应用？**

A： **Trie树的基本性质可以归纳为：**

a、根节点不包含字符，除根节点以外每个节点只包含一个字符。

b、从根节点到某一个节点，路径上经过的字符连接起来，为该节点对应的字符串。

c、每个节点的所有子节点包含的字符串不相同。

**优化：**

优化路线为字典树-》后缀树-》Patricia tree-》crit-bit tree

double array trie;

a、

**应用：**

快速检索、“串”排序、最长公共前缀问题、寻找热门查询。

**Q：SGI STL allocator（空间配置器）**

A： SGI空间配置器分两级，第一级配置器直接采用malloc和free管理内存；第二级采用内存池管理内存。申请空间时，如果区块够大，超过128B，移交第一级配置器处理。当区块小于128B时，以内存池管理。

**内存池管理流程：**

客户端请求分配大小为n bytes的内存（n < 128，n已经圆整为8的倍数）

1、检查n对应的free\_list是否为空，不为空，直接分配一个内存块给客户，如果为空，到第2步；

2、向内存池申请20个n bytes的内存，如果申请成功，一个返回给客户，另外19个挂到对应free\_list节点下；如果有不到20个n bytes的空间，返回一个给客户，另外的全部交给对应free\_list节点；如果连一个n bytes的空间都没法分配，进入第3步；

3、向堆申请40个n bytes的空间，如果申请成功，20个n bytes交给内存池，1个返回给客户，19个留给free\_list下的节点；如果堆内存不够，进入第四步；

4、从比当前n还大的free\_list节点中查找是否有内存可用，如果有，分n bytes给客户，另外部分交给内存池；如果n之后的所有free\_list上都没有内存可用，进入第5步；

5、调用第一级配置器的out\_of\_memory处理机制，尝试释放其它内存。如果可以，返回给客户。

**Q：多级哈希？**

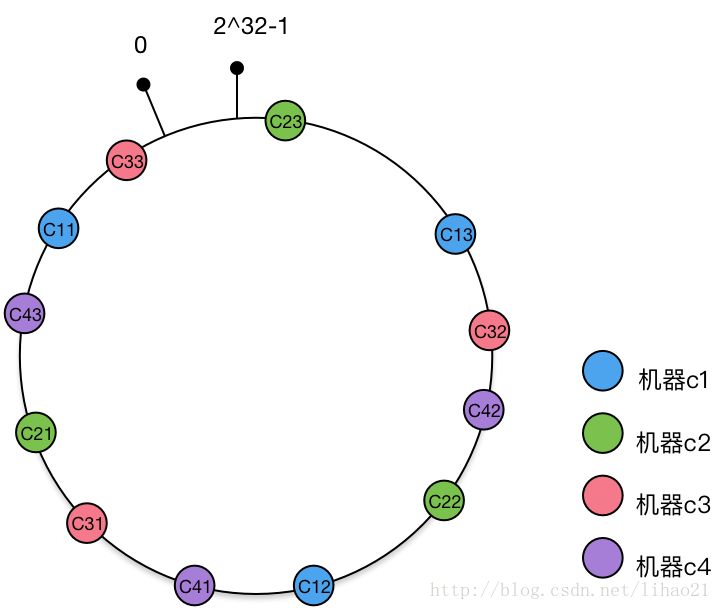
A： 实现：多阶hash表实际上是一个**锯齿数组**，每行一阶，每阶的元素个数都是质数，并从上到下，元素递减。其查找过程为：先将key在第一阶内取模，看是否是这个元素，如果这个位置为空，直接返回不存在；如果是这个KEY，则返回这个位置。如果这个位置有元素，但是又不是这个key，则说明hash冲突，再到第二阶去找，循环往复。

特点：多级hash行数越多，其空间利用率越高，然查找速度便逐渐减慢。反之，行数越少、每行元素越多，则多级hash查找速度越快，然空间利用率低下。且对于少量元素，可能存在在多级查找都找不到值的情况，此时可以直接将这部分数据存储到磁盘中，文件分为索引文件和数据文件，分别存储key和value，并把索引文件通过linux的mmap接口映射到内存中来，以方便快速查找。

**Q：一致性哈希？**

在分布式缓存系统中，由多台服务器提供服务，来自每个客户端的请求被hash到不同服务器上，计算方式是通过对服务器数量取模的方式。这样当服务器数量增加或减少时，将会出现大量的缓存访问不命中的情况。一致性哈希算法就是为了解决这个问题，一致性哈希采用一致性哈希环结构，环的起点为0，终点为2^32 - 1，且起点和终点相连。将机器和资源对象hash到对应环上的节点，哈希方法为求哈希值后对2^32取模，然后顺时针分配服务器。

当机器较少时，容易造成负载不均衡的问题，为了解决这个问题，引入虚拟节点，对每个服务器和资源对象用不同的hash函数计算出多个值，对每个位置都放置一个虚拟节点，节点多了，根据哈希函数的性质，平衡性自然就好了。



**Q：堆排序的实现**

A：利用一个vector存储待排序数据，然后将其构造出最大最小堆，将首元素和最后一个元素交换；对最后一个元素之前的数据继续构造最大堆，再交换，再排序。

**Q：resize和reserve函数的区别？vector删除元素时，容量会减小吗？如果想要减小怎么办？**

A： **resize和reserve函数：**

reserve：用于预留空间，不改变size大小。当输入参数n大于当前容量时，vector重新配置内存。当输入参数n小于等于当前容量时，原vector不做任何变化。

resize：改变vector的size大小，当设置的size大于当前capacity时，会导致重新配置空间；当size小于当前vector的size，会析构一部分对象。

这两种操作都可能导致容量的改变，但是只会增加，不会减少。

**如何减少容量：**

a、调用shrink\_to\_fit();使vector的容量和其size一致。

b、使用swap函数。

vector<int>vTmp1(1000);

vTmp1.resize(10);

int size1 = vTmp1.capacity(); //vTmp1的容量为1000

vector<int>vTmp2(vTmp1);

int size2 = vTmp2.capacity(); //vTmp2的容量为10

vTmp2.swap(vTmp1);

int size3 = vTmp1.capacity(); //vTmp1的容量为10，vTmp2的容量为1000

**Q：stack可以不用deque，而用vector来实现吗？queue可以用vector吗？**

A： 都可以。

a、stack的push和pop操作用vector的push\_back和pop\_back实现。往vector头部插入元素会导致vector元素的整体拷贝，效率很低。

b、另外，vector的size超过容量时，会发生重新配置空间，效率较慢。而且迭代器失效，但是由于stack和queue没有迭代器，所以这点不受影响。所以，如果一定要用vector实现queue和stack，最好在声明时就指明大小。以免后续多次配置空间的时间开销。

**Q：跳表（skip list）**

A： 有序数组可以通过二分查找查找目标值，开销为O(logn)。普通链表由于迭代器类型为可双向移动迭代器，无法进行下表寻址。所以直接查找时间开销为O（n）。跳表以空间换时间，通过增加额外的跳跃层，实现在链表中的二分查找。

**实现：**

a、以第1层为原始链表；第1层可以看做步距为1的表。

b、从第二层开始，保存步距为2^(n - 1)的指针。

c、实际查找时，从顶层迭代器开始，逐步定位目标。开销为O(logn)；

<https://www.jianshu.com/p/fcd18946994e>

**Q：跳表和红黑树**

A： 从性能上看，跳表和红黑树的开销差不多。

a、因为要重复分层存节点，跳表占用的内存较大；

b、跳表的实现和维护更简单；

c、更新数据时，红黑树可能需要更新o（logn）个节点；跳表只需要更新MAX\_LEVEL个节点；

<https://www.jianshu.com/p/fcd18946994e>

<https://www.cnblogs.com/charlesblc/p/5987812.html>

**三、常见算法的求解**

**Q：二叉树**

A： **求二叉树深度、宽度**

a、是否是完全二叉树，如果是，直接计算；

b、否则，求深度用递归；求宽度用层次遍历，用队列，并记录每层宽度；

**求两节点距离**

a、用栈记录根节点到两节点的路径；

b、出栈，查找最近公共祖先；

c、计算节点到公共祖先距离，距离相加。（其中，公共祖先可能为其中一节点）

**Q：链表**

A： **用链表进行大数据量的加减**

利用链表从低位到高位记录各位，并使用一个bool做进位标志。

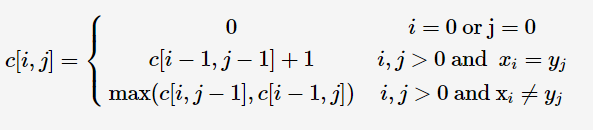
**判断链表是否有环、并查找环入口位置**

a、快慢指针判断是否有环

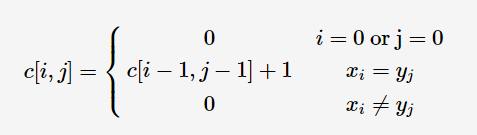
b、在a的快慢指针相遇位置和链表头部位置，同时启动一个单步链表，相遇点就是环入口；

**Q：字符串**

**A： 求最长公共子串**



**求最长公共子序列（连续）**



空间上，只需要保存两行数据，即O（N）

<https://blog.csdn.net/u012102306/article/details/53184446>

**求最长回文子串长度（不需要连续）**

将问题转化：求序列与其逆序列的最长公共子串的长度问题。

**Q：数组**

A： **数组中，找和为K的两个数**

a、如果数组已经排序，采用两个指针，一个从头到尾遍历，一个从尾到头遍历。求和，大于K，尾指针左移；小于K，头指针右移；

b、数组未排序，先排序，再进行a；

**数组中，求最长递增子序列**。数组vInput

时间复杂度O（nlogn）

a、构造数组vFlag、vMaxNums，vFlag[i]表示连续长度为i+1的子序列中，最小的值；vMaxNums[i]表示以vInput[i]结尾的最长递增子序列长度

b、从尾到头遍历vMaxNums，求最长子序列。

**数组中的逆序对**

时间复杂度O（nlogn）

冒泡、插排、选择排序、归并排序的过程本质上就是求逆序对的过程。其中，归并排序时间复杂度最低。

**反转字符串数组中的单词（时间复杂度：O（n）、空间复杂度O（1））**

例如：I am a student! student! a am I

a、实现一个函数，对数组中指定范围进行旋转。

b、对每个单词旋转；

c、对整个字符串旋转；

**无序数组中找出数组中第k大的数字**（时间复杂度：O（N），空间复杂度O（1））

类似STL的nth\_element函数，通过采用partition函数，递归找到对应位置；

**无序数组中找出前k个数字，并排序**（时间复杂度：O（nlogk），空间复杂度O（1））

类似STL中的partial\_sort，通过构造大小为k的最大堆。

**Q：位运算**

A： **求一个32位整数中二进制1的个数**

x = x & (x - 1)，每做一次运算，x去掉一个1，直到x = 0；

<https://www.cnblogs.com/graphics/archive/2010/06/21/1752421.html>

**Q：树**

A：

**Q：智力测试**

A： **如何通过rand7（）得到rand5（）**

rand7可以均匀得到0，1，2，3，4，5，6；需要得到rand5的话，需要凑成（0，1，2，3，4）的倍数。可以考虑：rand5 = （7\*rand7（）+rand7（））%5；