区间索引问题的形式化描述：给定一个区间[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]其中A1≤B1A2≤B2A3≤B3……An-1≤Bn-1AN≤BN，求这个区间与区间[C,D]的交集是什么。这个问题的关键是如何存储给定的区间，存储给定区间的数据结构要满足插入区间、删去区间和索引区间都有着比较低的时间空间复杂度。

# 线性结构——数组

采取一个数组结构存储给定的区间，假设给定了N个区间：[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]，那么按照各个区间边界的大小顺序A1,B1,A2,B2,……AN,BN把这2N个元素存储到一个数组中。存储结构如图所示：

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A1 | B1 | A2 | B2 | A3 | B3 | …… | An-1 | Bn-1 | AN | BN |

这样空间复杂度就是O（N）。下面分析这种数据结构存储给定区间是算法的时间复杂度：

1. 检索区间

若要检索的区间是[C,D]，使用二分法查找C和D在数组中的位置，在这两个位置之间对数组进行遍历，就能找出给定的区间和要检索的区间的交集。二分查找的时间复杂度是O（lgN），结合C、D两个位置的信息找交集的时间复杂度不确定，最好为O（1），最差为O（N）。

1. 插入区间

若要插入的区间是[C,D]，使用二分法查找C和D在数组中的位置，结合C、D的位置信息对原来的区间进行修改。插入区间这一操作可能导致原来的区间的小区间发生合并，可能产生一个新的原来没有的小区间，所以插入区间这一操作时间复杂度不确定，最好为O（1），最差为O（N）。

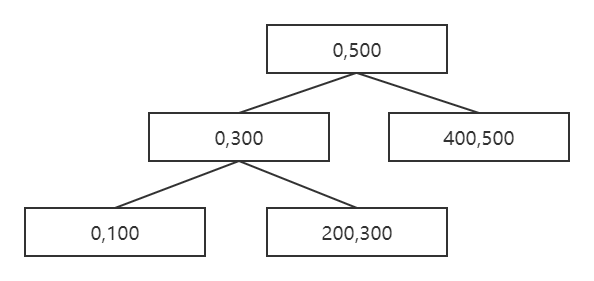
1. 删除区间

若要删除的区间是[C,D]，使用二分法查找C和D在数组中的位置， 结合C、D的位置信息对原来的区间进行修改。删除区间可能会导致原来的区间分裂，可能导致原来区间的小区间数量减少。删除区间这一操作时间复杂度不确定，最好为O（1），最差为O（N）。

# 树形结构——二叉树

采用二叉树（类似线段树）来存储给定的区间，假设给定了N个区间：[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]，线段树的每个节点存储两个值，线段树的叶子节点来存储每个小区间的两个端点值Ai、Bi，非叶子节点存储它的两个子节点的四个端点值Ai、Bi、Aj、Bj中的最小值和最大值Ax、Bx。

假设给定的区间是[0,100]、[200,300]、[400,500]，对于这个区间存储它的二叉树如下：



这样空间复杂度就是O（N）。下面分析这种数据结构存储给定区间是算法的时间复杂度：

1. 检索区间

若要检索的区间是[C,D]，从线段树的根节点开始进行带剪枝操作的递归先序遍历。

如果当前要处理的节点是非叶子节点且存储的两个值为Ax、Bx，若[C,D]与[Ax,Bx]有交集，则进行对这个节点的两个子节点进行递归操作；若[C,D]与[Ax,Bx]没有交集，则不对这个节点的两个子节点进行递归先序遍历。

如果当前要处理的节点是非叶子节点且存储的两个值为[Ax,Bx]，那么直接求[C,D]与[Ax,Bx]的交集。

如果二叉树是平衡的话，上述的操作的时间复杂度是O（lgN）；树非平衡时间复杂度为O（N）。

1. 插入区间和删除区间

同样采用树的遍历，时间复杂取决于树是否平衡。

# 树形结构——B+树

这种方法在数据库中应用很多，B+树是为磁盘或其他直接存储辅助设备设计的一种平衡查找树。B+树中，所有记录节点都是按键值的大小顺序存放在同一层的叶子节点上，由各叶子节点指针进行连接。因此B+树就相当于一个平衡多叉树加上有序数组链表。

m阶B+树满足以下条件：

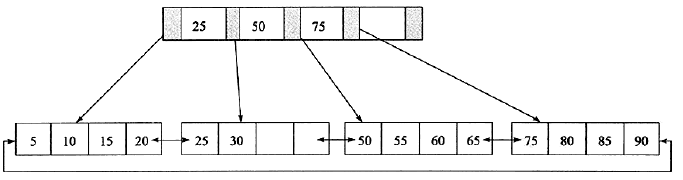
(1) 每个结点至多有m个孩子。

(2) 除根节点和叶结点外，每个结点至少有（m+1）/2个孩子。

(3) 如果根节点不为空，根结点至少有两个孩子。

(4) 所有叶子结点增加一个链指针，所有关键字都在叶子结点出现。

(5) 除了叶节点，结点的孩子数目等于关键字数目。B+树中非叶子结点存储的不是关键字数据的地址，而是指向叶子结点中关键字的索引。（所以任何关键字的查找必须走一条从根结点到叶子结点的路）。举例来说明B+树的原理：



这个4阶B+树高度为2，所有记录都在叶子节点上，每个叶子节点都有指向相邻叶子节点的指针，叶子节点元素顺序存放的。

这样对于给定了的N个区间：[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]可以把这区间上的每个元素都存在B+树的叶子节点上，若给定的N个区间：[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]中一共有X个元素，若要检索的区间是[C,D]其中有Y个元素。这样就可以很方便的进行区间的查找、删除、插入。B+树的空间复杂度为O（X），下面分析这种数据结构存储给定区间是算法的时间复杂度：

1. 检索区间

B+树进行一个值的检索的时间复杂度是O（lgX），考虑到最坏情况，若N个区间中每个小区间都只有一个元素，即Ai=Bi，那么时间复杂度是O（Y\*lgX）；考虑到最好情况，只需在B+树中检索一次就好，时间复杂度是O（lgX+Y）。

1. 插入区间和删除区间

若给定的N个区间：[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]中一共有X个元素，若要检索的区间是[C,D]其中有Y个元素。时间复杂度为O（Y\*lgX）。

# 树形结构——红黑树

红黑树本质上是一种自平衡的二叉查找树，但是基于提升插入删除操作的效率考虑，平衡的要求没有AVL树高。红黑树通过以下几个性质实现平衡：

1. 树中每个节点是红色或黑色的。
2. 根节点是黑色的。
3. 所有叶子都是黑色（叶子都是NIL节点）。
4. 每个红色节点的子节点必须有两个且都为黑色节点。（从每个叶子到根的所有路径上不能有两个连续的红色节点。）
5. 从任意节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。

满足以上性质的红黑树结构如图所示：



其中性质（4）和性质（5）能保证任意节点到其每个叶子节点路径最长不会超过这个节点到它的任意子节点最短路径的2倍。这是因为：当某条路径最短时，这条路径必然都是由黑色节点构成。当某条路径长度最长时，这条路径必然是由红色和黑色节点相间构成（性质（4）限定了不能出现两个连续的红色节点）。而性质（4）又限定了从任意节点到其每个叶子节点的所有路径必须包含相同数量的黑色节点。此时，在路径最长的情况下，路径上红色节点数量 = 黑色节点数量。该路径长度为两倍黑色节点数量，也就是最短路径长度的2倍。

红黑树的插入、删除节点操作比AVL树的插入、删除节点操作复杂很多，查找操作是普通的二叉查找树的查找操作。红黑树在业界应用很广泛，比如 Java 中的 TreeMap，JDK 1.8 中的 HashMap、C++ STL 中的 map 均是基于红黑树结构实现的，再比如Linux管理进程的容器。

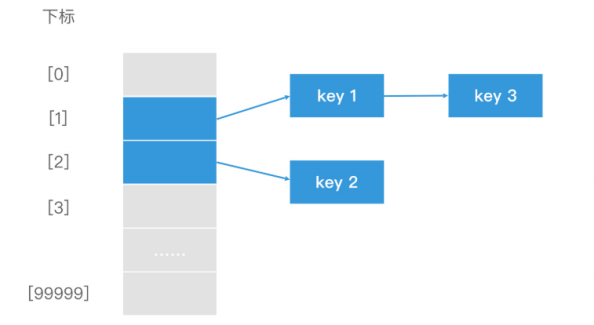
假设用红黑树存储给定的区间[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]，可以把这区间上的每个元素单独存储在红黑树上。若给定的N个区间：[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]中一共有X个元素，若要检索的区间是[C,D]其中有Y个元素。红黑树的空间复杂度为O（X），下面分析这种数据结构存储给定区间是算法的时间复杂度：

1. 检索区间、插入区间和删除区间

因为红黑树不像B+树一样有线性的索引，它整个都是树形的索引，红黑树本身的检索、插入、删除时间复杂度都是O（lgN）,那么对被检索区间[C,D]的Y个元素来说检索、插入、删除时间复杂度都是O（Y\*lgX）。

# 线性结构和树形结构的结合——哈希索引

就是哈希表与前面讲的红黑树的结合。哈希表中有一种常见的解决冲突的方法是“链表法”。“链表法”是指在哈希数组中不存储具体元素，而是存储一个链表头，一个关键字经过哈希函数计算得到相应的数组下标，那么我们就将他加入该位置所存链表的尾部，如下图所示：



现在，对这个方法做出一些改进，数组每个位置不在对应一个链表而是一个红黑树，这样可以减少大量冲突导致的链表过长带来的时间复杂度升高。

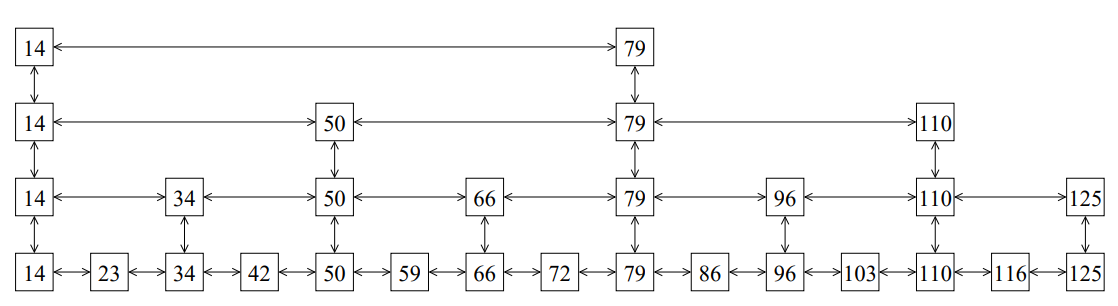
假设用哈希索引表存储给定的区间[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]，可以把这区间上的每个元素单独存储在哈希索引表中。若给定的N个区间：[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]中一共有X个元素，若要检索的区间是[C,D]其中有Y个元素。红黑树的空间复杂度为O（X），下面分析这种数据结构存储给定区间是算法的时间复杂度：

1. 检索区间、插入区间和删除区间

因为哈希函数的不确定性，所以这些的时间复杂度也不一样。如果全部没遇上哈希碰撞为O（Y），若恰好都遇上碰撞则时间复杂度O（Y\*lgX）。

# 链式结构——跳表

是基于链表实现的一种类似“二分”的算法。它可以快速的实现插入、删除、查找操作。它的原理是利用链表存储顺序的元素，然后给这个顺序存储加上“索引”来加速操作。举例来说，对于顺序序列：{14，23，34，42，50，59，66，72，79，86，96，103，110，116，125}，先用顺序链表存储这些元素，然后加上三级索引：



这样原本要查找元素“72”，在顺序链表中需要进行比对8次，但在跳表中只需对比5次就够了。一般来说，每一级链表是上一级链表长度的一般，就能达到比较好的索引效果。

假设用跳表存储给定的区间[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]，可以把这区间上的每个元素单独存储在跳表中。若给定的N个区间：[A1,B1]、[A2,B2]、[A3,B3]……[An-1,Bn-1]、[AN,BN]中一共有X个元素，若要检索的区间是[C,D]其中有Y个元素。跳表的空间复杂度为O（X），下面分析这种数据结构存储给定区间是算法的时间复杂度：

1. 检索区间

因为跳表是按照顺序存储的，所以这种算法时间复杂度不确定。若给定区间和被检索区间恰当，给定区间的一个小区间正好是被检索区间，可以借助于顺序结构来进行匹配的话，时间复杂度为O（lgX+Y）。如何给定区间比较分散，用不上顺序结构，时间复杂度O（Y\*lgX）。

1. 插入、删除区间

时间复杂度都是O（Y\*lgX）。

# 总结

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数据结构 | 检索 | 插入 | 删除 | 特点 |
| 数组 | 最好为O（1），最差为O（N） | 最好为O（1），最差为O（N）。 | 最好为O（1），最差为O（N）。 | 易于实现，时间复杂度高，需要连续内存。 |
| 二叉树（非平衡） | O（lgN）或O（N） | O（lgN）或O（N） | O（lgN）或O（N） | 插入删除后容易产生非平衡树，非平衡树导致时间复杂度上升。 |
| B+树（平衡） | 给定区间有X个元素，待查区间有Y个元素。最好为O （lgX+Y），最差为O（Y\*lgX） | 给定区间有X个元素，待查区间有Y个元素。O（Y\*lgX） | 给定区间有X个元素，待查区间有Y个元素。O（Y\*lgX） | 实现复杂，一般在数据库系统中使用。因为存在线性索引，适合给定区间连续的情况。 |
| 红黑树（平衡） | 给定区间有X个元素，待查区间有Y个元素。O（Y\*lgX） | 给定区间有X个元素，待查区间有Y个元素。O（Y\*lgX） | 给定区间有X个元素，待查区间有Y个元素。O（Y\*lgX） | 实现复杂，使用最广泛，各种语言、系统最常使用这个。时间复杂度稳定偏高，适合于待查区间较短的情况。 |
| 哈希+红黑树 | 给定区间有X个元素，待查区间有Y个元素。没遇上哈希碰撞为O（Y），若恰好都遇上碰撞则时间复杂度O（Y\*lgX）。 | 给定区间有X个元素，待查区间有Y个元素。没遇上哈希碰撞为O（Y），若恰好都遇上碰撞则时间复杂度O（Y\*lgX）。 | 给定区间有X个元素，待查区间有Y个元素。没遇上哈希碰撞为O（Y），若恰好都遇上碰撞则时间复杂度O（Y\*lgX）。 | 实现复杂，设计好哈希索引是关键。 |
| 跳表 | 给定区间有X个元素，待查区间有Y个元素。最好为O （lgX+Y），最差为O（Y\*lgX） | 给定区间有X个元素，待查区间有Y个元素。O（Y\*lgX）。 | 给定区间有X个元素，待查区间有Y个元素。O（Y\*lgX）。 | 实现比红黑树稍微简单，不需要连续的空间。在redis中使用。因为存在线性索引，适合给定区间连续的情况。 |