1. 单线程集合

1.1 ArrayList

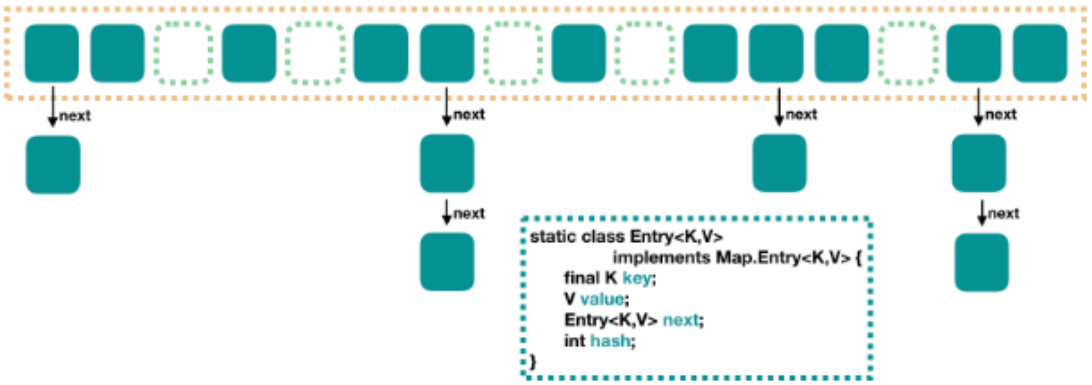
- ArrayList 实际上是通过一个数组去保存数据的。当我们构造 ArrayList 时；若使用默认构造函数，则 ArrayList 的默认容量大小是 10。
- 当 ArrayList 容量不足以容纳全部元素时，ArrayList 会重新设置容量：新的容量=“(原始容量 x3)/2 + 1”。
- ArrayList 的克隆函数，即是将全部元素克隆到一个数组中。
- ArrayList 实现 java.io.Serializable 的方式。当写入到输出流时，先写入“容量”，再依次写入“每一个元素”；当读出输入流时，先读取“容量”，再依次读取“每一个元素”。
- ArrayList 中的操作不是线程安全的。

1.2 LinkedList

- LinkedList 是一个继承于 AbstractSequentialList 的双向链表。它也可以被当作堆栈、队列或双端队列进行操作。
- LinkedList 的克隆函数，即是将全部元素克隆到一个新的 LinkedList 对象中。
- LinkedList 实现 java.io.Serializable 接口，这意味着 LinkedList 支持序列化，能通过序列化去传输。
- LinkedList 中的操作不是线程安全的。

1.3 HashMap

Java7 HashMap 结构



HashMap 的主体是一个数组，数组中的每个元素是一个单向链表，链表的一个节点是嵌套类 Entry 的实例，Entry 包含四个属性：key, value, hash 值和用于单向链表的 next。

- capacity：当前数组容量，始终保持 2^n，可以扩容，扩容后数组大小为当前的 2 倍。默认的初始容量为 16。
- loadFactor：负载因子，默认为 0.75。
- threshold：扩容的阈值，等于 capacity * loadFactor。当 HashMap 的大小>=阈值，并且新值要插入的数组位置已经有元素了，则进行扩容。

Put 方法：

HashMap 会对 null 值 key 进行特殊处理，总是放到 table[0]位置。

put 过程是先计算 key 的 hash 然后通过 hash 与 table.length 取模计算 index 值，然后将键值对放到 table[index]位置，当 table[index]已存在其它元素时，会在 table[index]位置形成一个单向链表，将新添加的元素放在 table[index]所对应链表的头部，原来的元素通过 Entry 的 next 进行链接，这样以链表形式解决 hash 冲突问题，当元素数量达到临界值(capactiy*factor)时，则进行扩容，是 table 数组长度变为 table.length*2

get 方法：

同样当 key 为 null 时会进行特殊处理，在 table[0]的链表上查找 key 为 null 的元素。

get 的过程是先计算 key 的 hash 然后通过 hash 与 table.length 取摸计算 index 值，然后遍历 table[index]上的链表，直到找到目标值，然后返回。

resize 方法：

这个方法实现了非常重要的 hashmap 扩容，具体过程为：先创建一个容量为 table.length*2 的新数组，修改临界值，然后把 table 里面元素计算 hash 值并使用 hash 与 table.length*2 重新计算 index 放入到新的 table 里面。

这里需要注意下是用每个元素的 hash 全部重新计算 index ,而不是简单的把原 table 对应 index 位置元素简单的移动到新 table 对应位置。

clear 方法：

遍历 table 然后把每个位置置为 null，同时修改元素个数为 0。

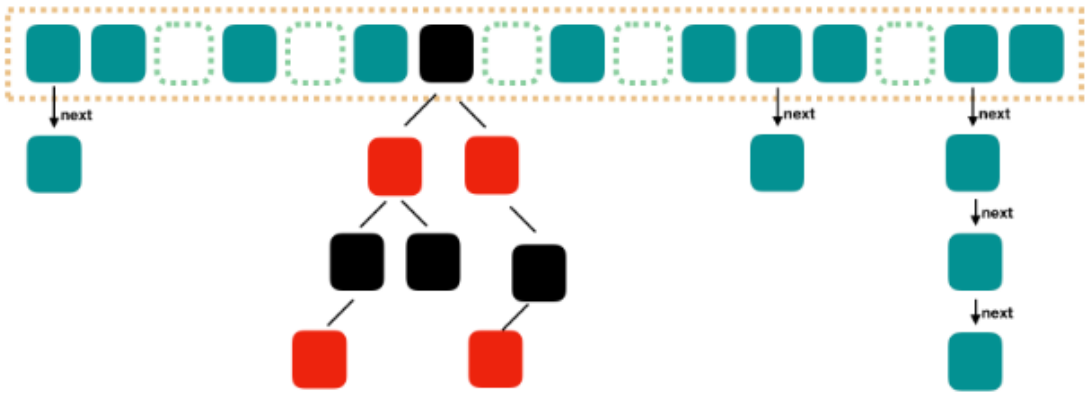
需要注意的是 clear 方法只会清除里面的元素，并不会重置 capactiy。

containsKey 和 containsValue:

containsKey 方法是先计算 hash 然后使用 hash 和 table.length 取模得到 index 值，遍历 table[index]元素查找是否包含 key 相同的值。

containsValue 方法就比较粗暴了，就是直接遍历所有元素直到找到 value。

Java8 HashMap 结构



Java8 对 HashMap 进行了一些修改，最大的不同就是利用了红黑树，所以其由 数组+链

表+红黑树 组成。在 Java8 中，当链表中的元素超过了 8 个以后，会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度为 O(logN)。

void	clear() Removes all of the mappings from this map.
Object	clone() Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	entrySet() 返回一个包含HashMap的键值对的set Returns a Set view of the mappings contained in this map.
V	get(Object key) 根据键获取值 Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
boolean	isEmpty() Returns true if this map contains no key-value mappings.
Set<K>	keySet() 返回一个HashMap键的Set Returns a Set view of the keys contained in this map.
V	put(K key, V value) Associates the specified value with the specified key in this map.
void	putAll(Map<? extends K,? extends V> m) Copies all of the mappings from the specified map to this map.
V	remove(Object key) 根据键移除 Removes the mapping for the specified key from this map if present.
int	size() Returns the number of key-value mappings in this map.
Collection<V>	values() 返回一个HashMap的值的Collection Returns a Collection view of the values contained in this map.

1.4 HashSet

HashSet 实现 Set 接口，不能有重复的元素，不保证 Set 的迭代顺序，特别是它不保证该顺序恒久不变。HashSet 允许使用 null 元素。

HashSet 是基于 HashMap 实现的，在 HashSet 中，元素都存到 HashMap 键值对的 Key 上面，而 Value 都是一个统一的值 private static final Object PRESENT = new Object();。

2. 多线程集合

2.1 Vector

- vector 是矢量队列，支持相关的添加、删除、修改、遍历等功能。
- vector 继承于 AbstractList，实现了 List, RandomAccess, Cloneable 这些接口。
- Vector 中的操作是线程安全的。

数据结构：

1) elementData 是"Object[]类型的数组" ,它保存了添加到 Vector 中的元素。elementData 是个动态数组，如果初始化 Vector 时，没指定动态数组的大小，则使用默认大小 10，当 Vector 容量不足以容纳全部元素时，Vector 的容量会增加。
2) elementCount 是动态数组的实际大小。
3) capacityIncrement 是动态数组的增长系数。如果在创建 Vector 时，指定了 capacityIncrement 的大小，则每次当 Vector 中动态数组容量增加时，增加的大小都是 capacityIncrement；否则将容量大小增加一倍。

2.2 HashTable

HashTable 已经被淘汰了 ,如果不需要线程安全 ,使用 HashMap ;如果需要线程安全，使用 ConcurrentHashMap。

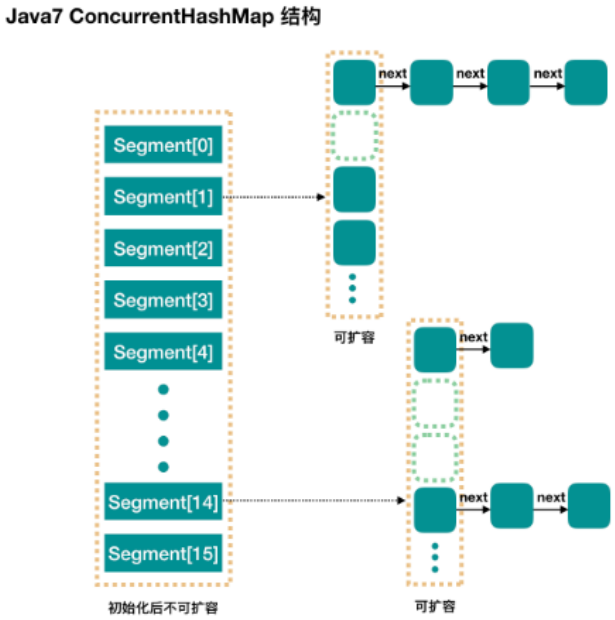
HashTable 的 key 和 value 都不能为空。

HashTable 会尽量使用素数、奇数，而 HashMap 则总是使用 2 的幂作为哈希表的大小。我们知道当哈希表的大小为素数时，简单的取模哈希的结果会更加均匀，所以单从这一点上看，HashTable 的哈希表大小选择，似乎更高明些。但另一方面我们又知道，在取模计算时，如果模数是 2 的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法。所以从 hash 计算的效率上，又是 HashMap 更胜一筹。

2.3 ConcurrentHashMap

https://blog.csdn.net/justloveyou_/article/details/72783008

HashMap 在并发环境下使用中最为典型的一个问题，就是在 HashMap 进行扩容重哈希时导致 Entry 链形成环。一旦 Entry 链中有环 ,势必会导致在同一个桶中进行插入、查询、删除等操作时陷入死循环。



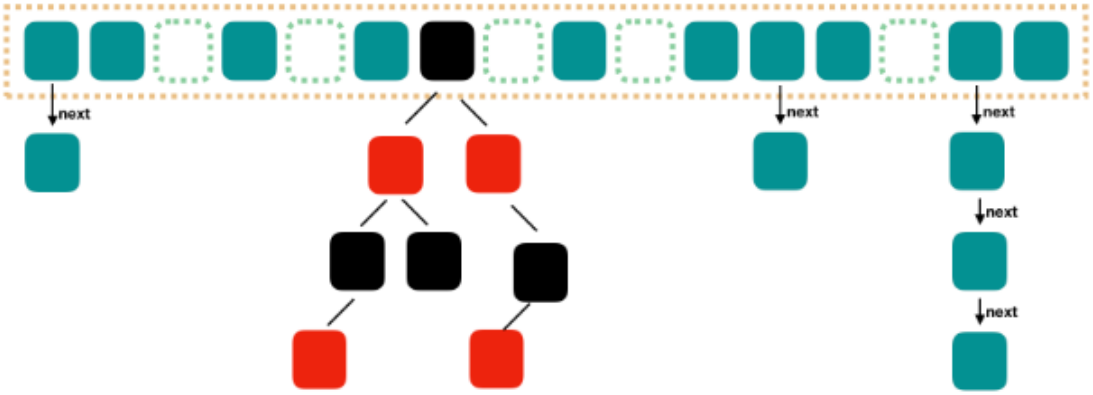
ConcurrentHashMap 允许多个修改(写)操作并发进行 ,其关键在于使用了锁分段技术，它使用了不同的锁来控制对哈希表的不同部分进行的修改(写) ,而 ConcurrentHashMap 内部使用段(Segment)来表示这些不同的部分。实际上，每个段就是一个小的哈希表，每个段都有自己的锁(Segment 类继承了 ReentrantLock 类)。这样，只要多个修改(写)操作发生在不同的段上，它们就可以并发进行。

ConcurrentHashMap 实现线程安全的关键点：

- Segment 类继承了 ReentrantLock 类，对每个段进行写操作时都会加锁。
- 在 HashEntry 类中，key，hash 和 next 域都被声明为 final 的，value 域被 volatile 所修饰，因此 HashEntry 对象几乎是不可变的，这是 ConcurrentHashmap 读操作并不需要加锁的一个重要原因。

- ConcurrentHashMap 中 key 和 value 都不允许为空，但在读操作时有可能会出现键值对存在但读出来的 value 值为空的情形。这种情形发生的场景是 :初始化 HashEntry 时发生的指令重排序导致的，也就是在 HashEntry 初始化完成之前便返回了它的引用。这时，JDK 给出的解决之道就是加锁重读。
- size 方法主要思路是先在没有锁的情况下对所有段大小求和，这种求和策略最多执行 RETRIES_BEFORE_LOCK 次(默认是两次) ;在超过 RETRIES_BEFORE_LOCK 之后，如果还不成功就在持有所有段锁的情况下再对所有段大小求和。

Java8 ConcurrentHashMap 结构



相较于 JDK1.7，在 JDK1.8 中，对 ConcurrentHashMap 做了较大的改动，主要有两方面：

- 取消 segments 字段，直接采用 transient volatile HashEntry<K,V>[] table 保存数据，采用 table 数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。
- 将原先 table 数组 + 单向链表的数据结构，变更为 table 数组 + 单向链表 + 红黑树的结构。

2.4 CopyOnWriteArrayList

CopyOnWrite 容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时

候，不直接往当前容器添加，而是先将当前容器进行 Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对 CopyOnWrite 容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。

CopyOnWriteArrayList 在添加的时候是需要加锁的，否则多线程写的时候会 Copy 出 N 个副本出来。读的时候不需要加锁，如果读的时候有多个线程正在向 CopyOnWriteArrayList 添加数据，读还是会读到旧的数据，因此 CopyOnWrite 容器只能保证数据的最终一致性，不能保证数据的实时一致性。

3. 树

3.1 二叉查找树

二叉查找树(没有重复元素)的特征是:对于树中的每一个结点，它的左子树中结点的值都小于该结点的值，而它的右子树中结点的值都大于该结点的值。

可以采用前序插入元素的方法重构一棵二叉查找树，重构的树为原始的二叉查找树保留了父结点和子结点的关系。

二叉查找树的中序遍历是一个排好序的线性表。

删除 BST 中的一个元素：

- 情况 1：当前节点没有左子节点。

  只需要将当前节点的父节点和当前节点的右子节点相连。

- 情况 2：当前节点有左子节点。

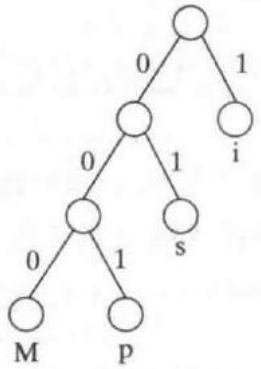
  假设 rightMost 指向包含 current 结点的左子树中最大元素的结点（rightMost 结点不能有右子结点，但是可能会有左子结点），而 parentOfRightMost 指向 rightMost 结点的父结点。使用 rightMost 结点中的元素值替换 current 结点中的元素值，将 parentOfRightMost

结点和 rightMost 结点的左子结点相连，然后删除 rightMost 结点。

霍夫曼编码树

霍夫曼编码通过使用更少的比特对经常出现的字符编码来压缩数据。字符的编码是基于字符在文本中出现的次数使用二叉树来构建的.该树称为霍夫曼编码树。

例如，假设文本为 Mississippi，则它的霍夫曼树如下图：



a）霍夫曼编码树

字符	编码	出现次数
M	000	1
p	001	2
s	01	4
i	1	4

b）字符编码表

对应的编码方案为：

编码为 解码为

Mississippi ========> 00010101101011O010011 ========> Mississippi

为了构建一棵霍夫曼编码树，使用如下算法:

1) 从由树构成的森林开始。每棵树都包含一个字符结点。每个结点的权重就是文本中字符的出现次数。

2) 重复以下步骤来合并树，直到只有一棵树为止:选择两棵有最小权重的树，创建一个新结点作为它们的父结点。这棵新树的权重是子树的权重和。

3) 对于每个内部结点，给它的左边赋值 0 ，而给它的右边赋值 1 。所有的叶子结点都表示文本中的字符。

没有编码是另外一个编码的前缀，整个属性保证了流可以无二义性地解码。

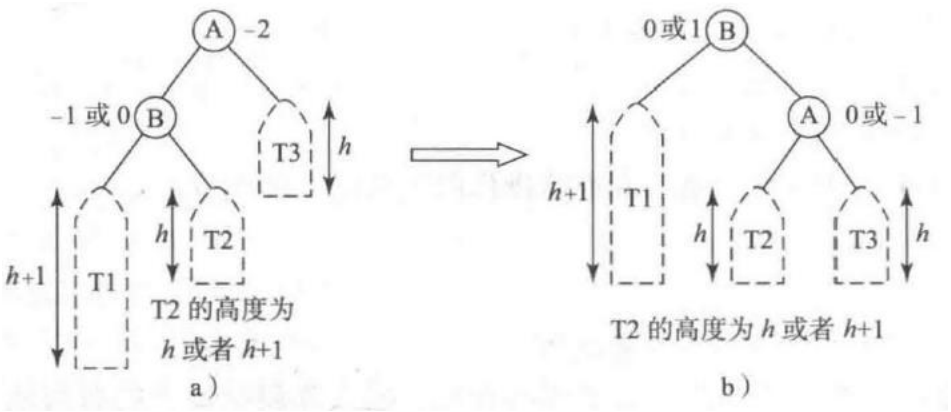
3.2 AVL 树（平衡二叉查找树）

二叉树的查找、插入和删除操作的时间依赖于树的高度，最坏情形下，高度为 O(n) 。

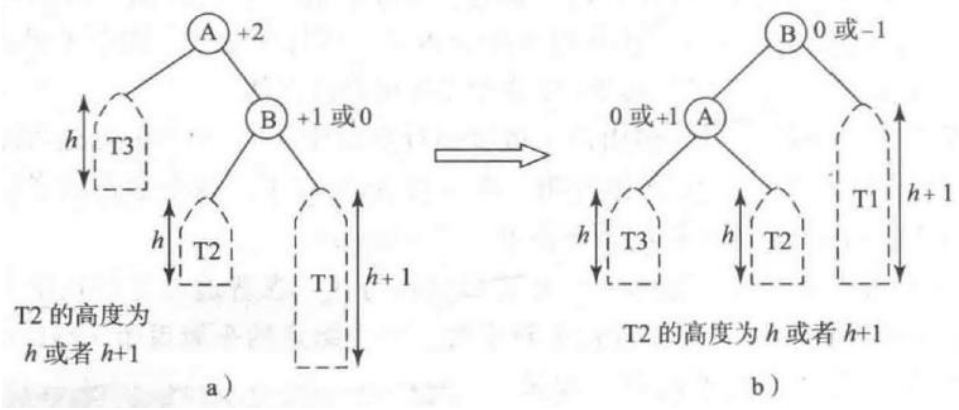
如果一棵树是完全平衡的(perfectly balanced)——即一棵完全二叉树——它的高度是 logn 。保持一颗完全平衡的树代价比较大，妥协的做法是保持一颗良好平衡的树。AVL 树就是良好平衡的，它的每个节点的子树的高度差距为 0 或 1。

重新平衡树：

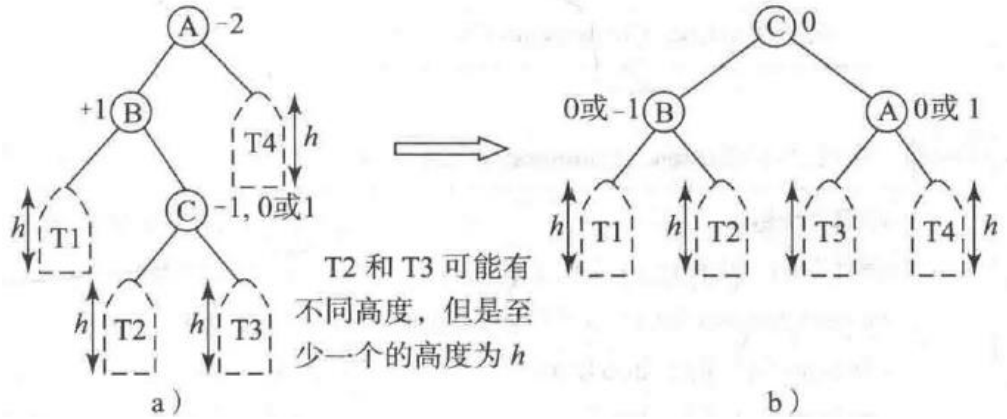
- LL 旋转：



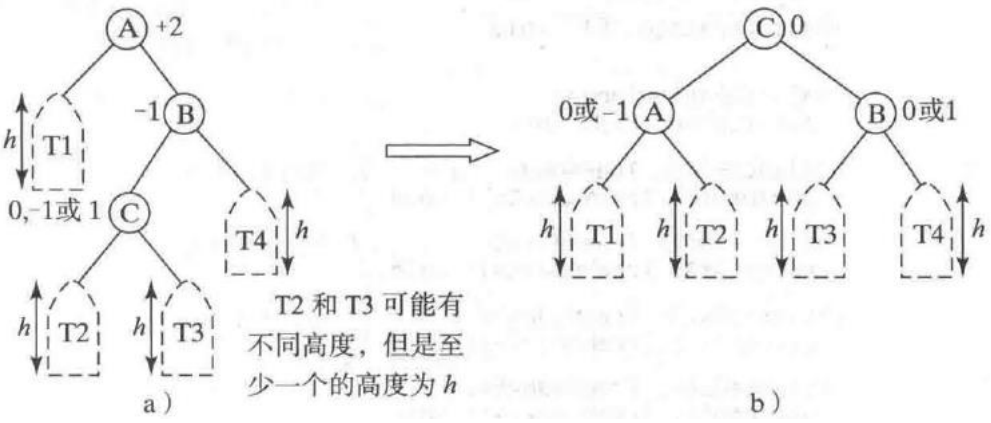
- RR 旋转：



- LR 旋转：



- RL 旋转：

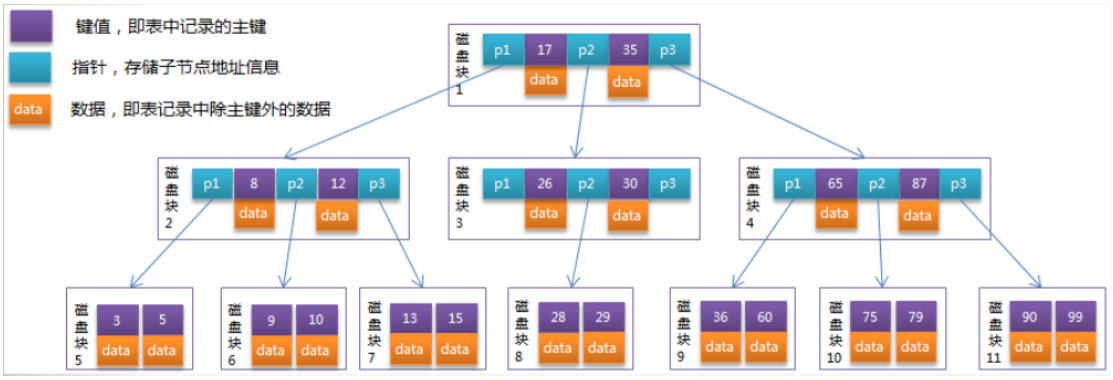


3.3 B-Tree，B+Tree

B-Tree：

一棵 m 阶的 B-Tree 有如下特性：

1. 每个节点最多有 m 个孩子。
2. 除了根节点和叶子节点外，其它每个节点至少有 Ceil(m/2)个孩子。
3. 若根节点不是叶子节点，则至少有 2 个孩子。
4. 所有叶子节点都在同一层，且不包含其它关键字信息。
5. 每个非终端节点包含 n 个关键字信息（P0,P1,…Pn, k1,…kn）。
6. 关键字的个数 n 满足：ceil(m/2)-1 <= n <= m-1。
7. ki(i=1,…n)为关键字，且关键字升序排序。
8. Pi(i=1,…n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于 ki，但都大于 k(i-1)。



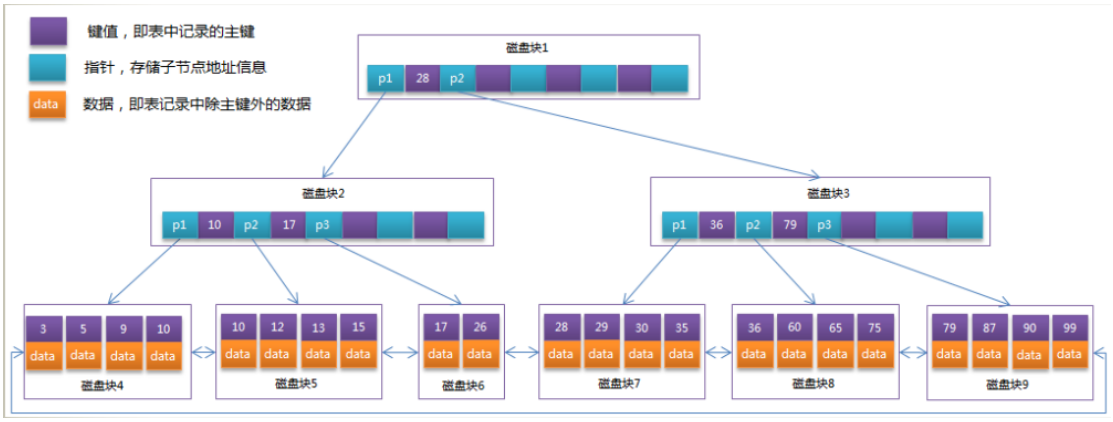
B+Tree：

B+Tree 相对于 B-Tree 有几点不同：

1. 非叶子节点只存储键值信息。

2. 所有叶子节点之间都有一个链指针。

3. 数据记录都存放在叶子节点中。



3.4 红黑树

除了二叉树的基本要求外，红黑树必须满足以下几点性质：

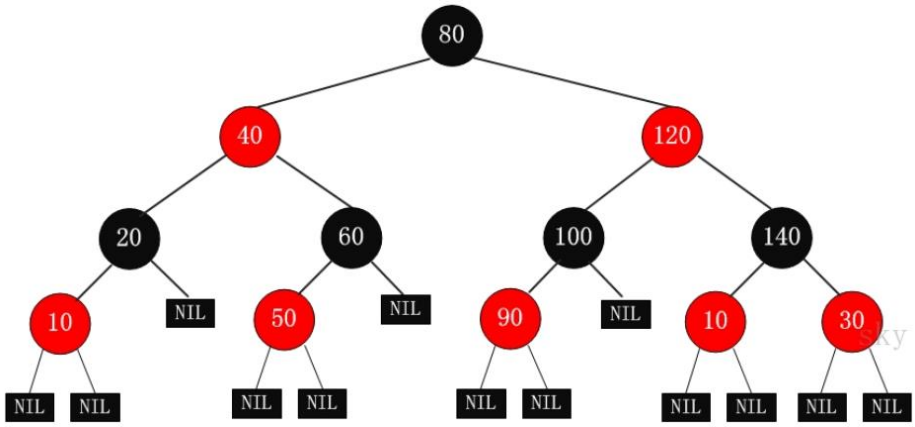
1. 节点必须是红色或者黑色的；

2. 根节点必须是黑色的；

3. 叶节点（NIL）是黑色的（NIL 节点无数据，是空节点）；

4. 红色节点的子节点必须是黑色的（黑色节点的子节点可以是红色的也可以是黑色的）；

5. 从任一节点出发到其每个叶子节点的路径，黑色节点的数量是相等的。



从根节点到叶子节点的路径中，最短的全是黑色节点，最长的是黑红相间的（性质 4），因此树中任意两条从根节点到叶子节点的路径中的节点数相差不会超过一倍（性质 5）。

4. 图

4.1 DFS

DFS 一般使用递归来实现，也可以利用栈来实现。

DFS 的应用：

- 检测图是否是连通的。由任何一个顶点开始搜索图，如果搜索的顶点的个数与图中顶点的个数一致，那么图是连通的；否则，图就不是连通的。
- 检测两个顶点之间是否存在路径。
- 找出两个顶点之间的路径。
- 找出所有连通的部分。
- 检测图中是否存在回路（进行拓扑排序）。
- 找出图中的回路。
- 找出哈密尔顿路径/回路。探索所有可能的 DFS，找到具有最长路径的那个。

4.2 BFS

BFS 一般利用队列来实现。

BFS 的应用：

- 检测图是否是连通的。
- 检测在两个顶点之间是否存在路径。
- 找出两个顶点之间的最短路径。在无向图中，根结点和广度优先搜索树中的任意一个结点之间的路径是根结点和该结点之间的最短路径。
- 找出所有的连通部分。
- 检测图中是否存在回路。
- 找出图中的回路。
- 检测一个图是否是二分的(如果图的顶点可以分为两个不相交的集合，而且同一个集合中的顶点之间不存在边，那么这个图就是二分的)。

4.3 最小生成树

- **Kruskal 算法：**从图中的最小边开始，进行避圈式扩张。
- **管梅谷的破圈法：**从赋权图 G 的任意圈开始，去掉该圈中权值最大的一条边，称为破圈。不断破圈，直到 G 中没有圈为止，最后剩下的 G 的子图为 G 的最小生成树。
- **Prim 算法：**从图 G 中的任意一个顶点 u 开始，选择与 u 关联且权值最小的边作为最小生成树的第一条边 e1，在接下来的边 e2,e3,…,en-1 ,在与一条已经选取的边只有一个公共端点的的所有边中，选取权值最小的边。

4.4 加权图的最短路径

- **Dijkstra 算法：**时间复杂度 O(N^2)

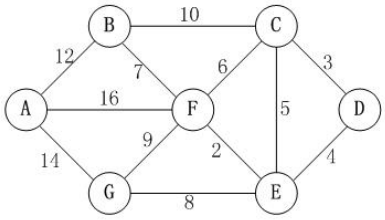
1. 将起点 A 放入集合中，A 点的权值为 0,因为 A->A=0。
2. 与起点 A 相连的所有点的权值设置为 A->点的距离 ,连接不到的设置为无穷。并且找出其中最小权值的 B 放入集合中（此时 A->B 必定为最小距离）。
3. 与 B 点相连的所有点的权值设置为 B->点的距离，并且找出其中最小权值的 C 点

放入集合中（此时 C 的权值必定为其最小距离）。

4. 重复步骤 3，直至所有点加入集合中。便能得到所有点与 A 点的最短距离。

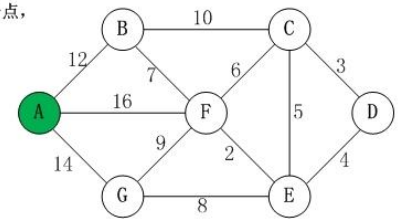
- **Floyd 算法：**时间复杂度 O(N^3)，空间复杂度 O(N^2)

第1步：
初始化矩阵S



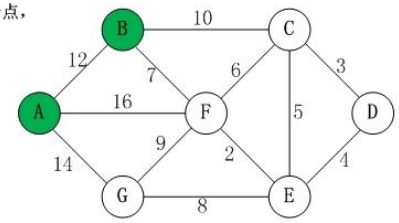
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	INF
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	INF	INF	INF	8	9	0

第2步：
以顶点A为中介点，
更新矩阵S。



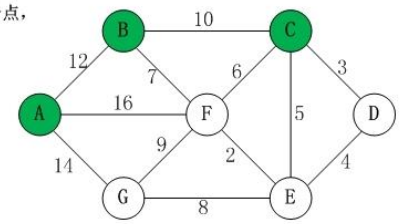
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	INF	INF	8	9	0

第3步：
以顶点B为中介点，
更新矩阵S。



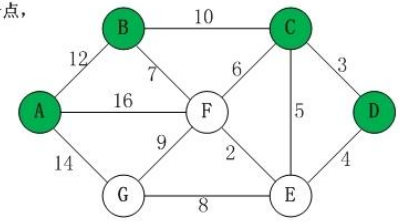
	A	B	C	D	E	F	G
A	0	12	22	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	22	10	0	3	5	6	36
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	36	INF	8	9	0

第4步：
以顶点C为中介点，
更新矩阵S。



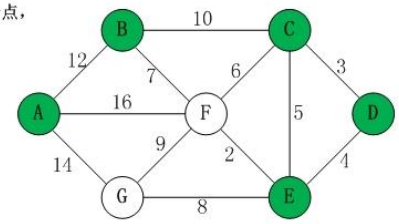
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0

第5步：
以顶点D为中介点，
更新矩阵S。



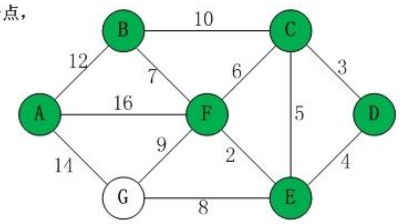
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0

第6步：
以顶点E为中介点，
更新矩阵S。



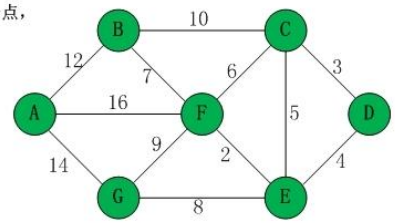
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	23
C	22	10	0	3	5	6	13
D	25	13	3	0	4	6	12
E	27	15	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	23	13	12	8	9	0

第7步：
以顶点F为中介点，
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	16	13	12	8	9	0

第8步：
以顶点G为中介点，
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	16	13	12	8	9	0

5. 查找

5.1 二分查找

使用二分查找的前提条件是数组元素必须已经排好序 ,二分查找法首先将关键字与数组

的中间元素进行比较，考虑下面三种情况：

- 如果关键字比中间元素小，那么只需在前一半数组元素中进行递归查找
- 如果关键字和中间元素相等，那么匹配成功，查找结束
- 如果关键字比中间元素大，那么只需在后一半数组元素中进行递归查找

6. 排序

7. 类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最差情况		
插入排序	插入排序	O(N^2)	O(N)	O(N^2)	O(1)	稳定
	Shell 排序	O(N^1.3)	O(N)	O(N^2)	O(1)	不稳定
选择排序	选择排序	O(N^2)	O(N^2)	O(N^2)	O(1)	不稳定
	堆排序	O(N*lgN)	O(N*lgN)	O(N*lgN)	O(1)	不稳定
交换排序	冒泡排序	O(N^2)	O(N)	O(N^2)	O(1)	稳定
	快速排序	O(N*lgN)	O(N*lgN)	O(N^2)	O(N*lgN)	不稳定
归并排序	归并排序	O(N*lgN)	O(N*lgN)	O(N*lgN)	O(N)	稳定
基数排序	基数排序	O(d(r+n))	O(d(n+rd))	O(d(r+n))	O(rd+n)	稳定
注：基数排序的复杂度中，r 代表关键字的基数，d 代表长度，n 代表关键字的个数。						

稳定的排序算法：插入排序、冒泡排序、归并排序、基数排序

与关键字的初始排列顺序无关：选择排序（N^2）、堆排序(N*lgN)、归并排序(N*lgN)、基数排序(N*K)