数据结构

笔记本：	我的第一个笔记本		
创建时间：	2020/6/14 22:53	**更新时间：**	2020/6/21 19:31
作者：	So fLy		
URL：	https://baike.baidu.com/item/%E6%A0%91/2699484?fr=aladdin#3		

一维数据结构

数组、链表、跳表

1. Queue源码分析

1. 继承了collection接口；
2. 相关方法：
   2.1 add方法：插入相关元素，返回一个Boolean类型的操作状态返回值；
   2.2 offer方法：功能同add方法，但是插入失败时会抛出异常；
   2.3 remove方法：取出并移除队列头部的元素，区别与poll方法的是，如果容器为空，则会抛出异常；
   2.4 poll方法：功能同remove方法，不会抛出异常；
   2.5 element方法：取出队列头顶的元素并移除，当队列为空时会抛出异常；
   2.6 peek方法：功能同element方法，但是不移除元素。

2. PriorityQueue源码解读

1. 继承了abstractqueue父类，实现了序列化接口；
2. 队列初始容量为11；
3. 优先队列【n】的实现是由队列【2* n+1】和队列【2* （n+1）】两个队列实现的；
4. 优先级的定义方式有如下几种：
   4.1 通过实现comparator接口进行排序；
   4.2 根据元素自身属性进行自然徐排序；
   4.3 如果传入的类没有实现comparator接口，元素则根据自身值进行升序排序。
5. 构造器参数可以传入：比较器、初始容量大小、集合、其他priorityqueue、sortedset；

6. 有从传入参数为priorityqueue、collection的初始化方法；
7. grow方法：
   7.1 首先判断新的队列容量大小：

- 如果老队列的大小小于64，则新队列的大小为：老+（老+2）；
- 如果老队列的大小大于64，则新队列的大小为：老+老/2；
  7.2 再判断新队列大小与默认容量大小的值：
- 如果新队列的大小-默认最大值小于零，则直接使用Arrays工具类中的copyof方法拷贝老队列中的元素进进的队列，并且赋予新队列的容量大小为当前新队列的大小；
- 如果新队列的大小-默认最大值大于零，则调用hugeCapacity方法进行新队列容量大小的确认；
- hugeCapacity方法主要判断用户传入的容量大小与默认最大容量值进行比较，如果大于默认值，则容量取integer的最大值；如果小于则取默认容量值作为新队列的大小。

8. add与offer方法：add直接调用offer方法，向队列中加入一个指定元素，并返回插入结果；
9. remove方法：移除队列中的目标值，以及通过equals函数判断与目标值相等的元素。如果队列包含且移除成功，则返回true。

哈希表

哈希表的定义

将keyword，经过哈希函数转换后，得到该值在哈希表中的地址（下标），将该值存放在该下标所在位置的一种数据结构。

哈希表会出现的的问题以及解决方式

哈希碰撞：keyword经过哈希函数转换后得到同一个存放地址；常用解决办法：拉链法、再哈希法、建立公共溢出区。

基于哈希表的实现

set系：

可能有序，但元素不能重复

HashSet：
底层使用Hash表实现；

LinkedSet:
HashSet的子类，底层为双向链表实现，其存于元素有序（指取出顺序与放入的顺序一致）；
SortedSet:
不仅有序，还要求放入的元素能够排序，即实现了comparable接口；
但SortedSet本身为接口，不可直接使用，所以使用其实现类TreeSet；
NavigableSet：？？？

map系：

key不能重复；与Collection接口同级,其子类是否可排序的概念针对key来说的。

SortedMap
实现类为TreeMap，存入的数据有序
NavigableMap
实现类为TreeMap，提供快速查找元素的方法
HashMap

Map接口的实现，底层实现为挂载链或者树的hash表;

1.7 版本实现：

1. 使用哈希表+链表实现，先根据输入信息的k确定哈希表的下标，即确定桶的位置，如果当前桶为空，则直接写入该entry实例；
2. 如果不为空，即出现哈希冲突，则在该桶内形成链表进行保存；O（n）

1.8 版本实现：

1. 使用node类代替了entry类，插入时会先按老板进行链表储存，如果链表大于8，且容量大于64，会转化为红黑树的插入与查询；查询时会总是返回该桶的第一个值，后再判断该桶是链表还是树；O（logn）
2. hashmap容量都均为2的幂次方，可以通过按位与操作计算余数；
3. hashmap不安全的原因：多线程put时，当容量超过填充因子时，会触发resize方法，多线程使用头插法会造成链表形成死循环；
4. hashmap的源码分析

LinkedHashMap
底层实现为使用链表的方式维持插入元素的顺序
ConcurrentHashmap

1. 线程安全且高效的HashMap实现，常用于高并发，桶类中的next还有value使用volatile关键字修饰，使用segment代替原hashmap的哈希表，
2. 1.7版本put方法中使用分段锁lock+volatile关键字保证线程安全，get方法因为value由volatile修饰，所以无需加锁；
3. 1.8版本底层在Hash表的基础上，加入红黑树的形式储存数据，锁的方式换成了CAS+synchronized保证数据安全；

二维数据结构

树

树的逻辑结构

基本概念：

1. 节点的度：一个节点含有的子树的个数；
2. 树的度：一颗树中，最大的节点的度数：
3. 树的深度：定义一棵树的根结点层次为1，其他结点的层次是其父结点层次加1。一棵树中所有结点的层次的最大值称为这棵树的深度；
4. 链表可以堪称时一种特殊的树结构；
5. 根节点与父节点的区别：根节点没有父节点且仅有一个，父节点可以有父节点。

树的节点代码

```
public class TreeNode{
public int  val;
public TreeNode left, right;
public TreeNode(int val){

* this.val = val;

this.left = null;
this.right = null;
 }}
```

树的遍历：

1. 先序（根左右）、中序（左根右）、后续（左右根）、层次（从根节点层开始，逐层的从上到下，左右顺序看遍历顺序）；
2. 遍历的实现代码：

```
代填
```

树的基本类型

1. 二叉树：每个节点至多只有两颗子树；
2. 满二叉树：深度为k且有2k-1个节点的二叉树，每个子节点都是二叉树；
3. 完全二叉树：若二叉树的深度为h，除第h层外，其他层的节点数都达到最大个数，第h层所有的节点都连续几种在最左边的二叉树；
4. 二叉排序树：左右子树皆为二叉排序树且都不为空，且左子树上的所有节点的值均小于根节点的值，右子树上所有的值均大于根节点的值；
5. 平衡二叉排序树（AVL）：左子树和右子树的深度之差的绝对值不超过1；
6. 平衡二叉排序红黑树：关注局部平衡
   6.1 使用颜色保证二叉排序树的时间复杂度，平衡二叉树存在四种旋转的情况，来保证二叉树的平衡；
   6.2 每个节点可以是红色或者黑色，根节点必为黑色；
   6.3 每个叶节点都追加一个新的叶节点的叶节点nil，必为黑色；
   6.4 一个节点是红色，则他的两个子节点均为黑；
   6.5 对每个节点，从该节点到其所有后代新叶节点的简单路径上，均包含相同数目的黑色节点；

二叉排序树

查询与增删操作时间复杂度：log(N);

二叉排序树的CRUD

1. 删除有子树的节点：
   选择与该节点值相近的节点，对该节点进行替换，一般选择该节点右侧的最小节点。
2. 二叉排序树的CRUD：

[二叉排序树的CRUD](%3Ca
href="https://visualgo.net/zh/bst"%3Ehttps://visualgo.net/zh/bst%3C/a%3E)

B树

m叉树中每个节点至多有个m棵子树，且每个节点可以存储多个关键字，常使用在文件索引上，可有效减少磁盘IO的次数；

1. B树中最大子节点个数称为B树的阶；
2. B树的关键字分布在整个节点中，且每个关键字只出现在一个节点中；
3. 一个m阶B树有如下特点：

- 非叶节点最多有m颗子树；
- 根节点至少有2颗子树，非根非叶节点，至少有m/2颗子树，向上取整；
- 非叶节点保存的关键字个数，等于该节点子树个数-1；
- 非叶节点中的关键字，大小有序；
- 每个节点中的关键字，其左子树的所有关键字总是小于该关键字，右边亦然；
- 所有叶子节点，都在同一层；

4. B树的关键字查找流程：首先从根节点开始，对根节点的关键字进行二分查找，如果找到了，则结束；如果没有找到，则进入查找关键字范围的子树内，递归遍历，直到叶节点；

B+树

与B树基本相同，存在些许差异，更适合做索引系统。

1. B+树的特点：

- 节点中的关键字数目，与子树的数目相同；
- 关键字对应的子树节点，都大于等于关键字；子树中，包括关键字自身；
- 所有关键字，都出现在叶节点中；
- 所有叶节点，都有指向下一个叶节点的指针；
- 遍历时，不会在非叶节点命中，一定会查询到叶子节点；
- 叶子节点相当于存储层，保存关键字对应的数据，而叶节点只保存关键字和指向叶节点的指针，不保存关键字对应的数据，所以同样关键字的B+树要比B树容量小的多；

2. 适用的场景：索引系统，原因如下：

- 叶节点画像之间有指针相连，适合范围索引；
- 因为非叶节点只保存关键字与指针，所以每个节点可以保存更多的关键字，且降低树高，减少磁盘IO；
- B+树查询速度较为稳定；所有关键字的查询路径都遵循根节点到叶节点的流程；

堆

可以迅速找到一堆数中的最大值 **或者** 最小的一种数据结构。
常见实现为二叉堆、斐波那契堆。

1. 不遵循二叉排序树的节点分布，仅从根节点到叶子节点的每一条路径是降序排列的；
2. 堆不支持顺序遍历，仅为快速移除最大值或者最小值，快速插入新节点；

3. 大根堆、小根堆：一颗完全二叉树中，任何节点的所有子孙都是大于/小于等于当前这个节点；

二叉堆

使用 **数组** 来实现的完全二叉树。

1. 二叉堆的性质：

- 其本身是一颗完全二叉树；
- 树中的任意节点的值，总是大于等于其子节点的值。

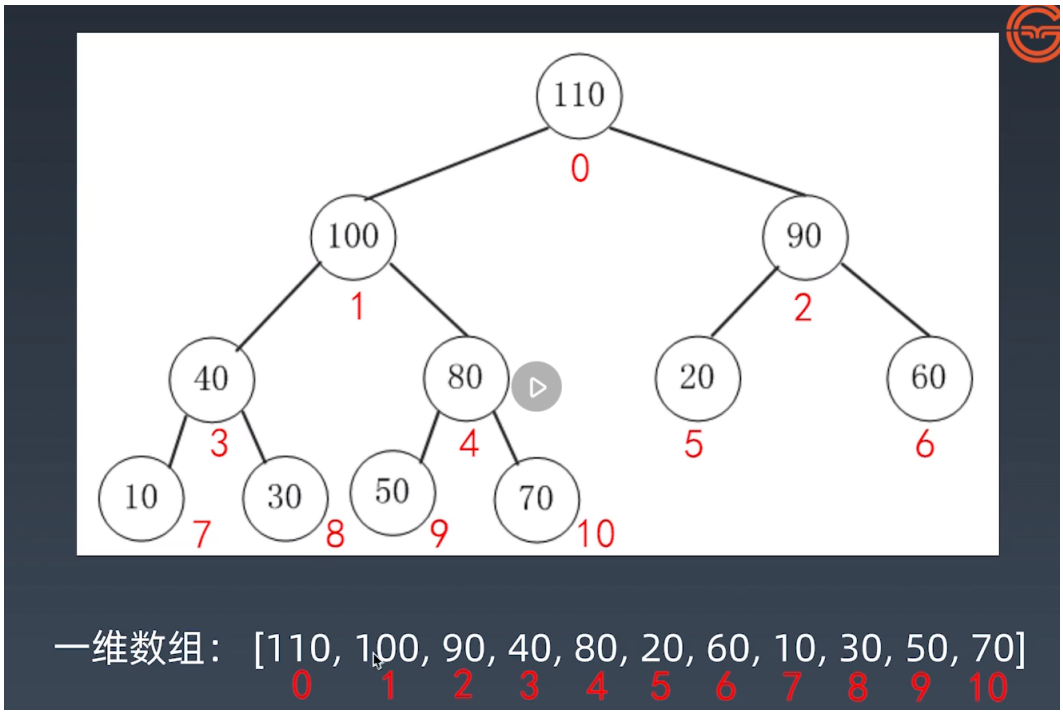
2. 使用一维数组存储的规律：

- 第一个元素放在数组index为0的位置；
- 索引为i的左子节点的索引是:

- 索引为i的右子节点的索引是：

- 索引为i的父节点的索引是：

- 存放顺序如下图：



3. 二叉堆的插入操作

- 从最后端插入，通过类似冒泡排序的方式重定位自己的位置；
- 时间复杂度log(N)
  最坏情况下需要将插入元素从尾部移动到根节点，需要移动树的深度的层数，且数据结构为树结构且总节点为N，所以树的深度为log(N)；
- 插入操作的源码：

```
	public void insert(int x) {
		if (isFull()) {
			throw new NoSuchElementException("Heap is full, No space to insert new element");
		}
		heap[heapSize] = x;
		heapSize ++;
		heapifyUp(heapSize - 1);
	}

	private void heapifyUp(int i) {
		int insertValue = heap[i];
		while (i > 0 && insertValue > heap[parent(i)]) {
			heap[i] = heap[parent(i)];
			i = parent(i);
		}
		heap[i] = insertValue;
```

```
        }
```

4. 二叉堆的堆顶删除操作

- 将堆尾元素替换的到顶部；（为了保证完全二叉树的性值，所以取末尾节点进行替换）
- 再依次从根部向下调整整个堆的结构。选择子节点中的较大值进行互换;
- 删除操作的源码:

```
        public int delete(int x) {
                if (isEmpty()) {
                        throw new NoSuchElementException("Heap is empty, No element
to delete");
                }
                int maxElement = heap[x];
                heap[x] = heap[heapSize - 1];
                heapSize--;
                heapifyDown(x);
                return maxElement;
        }

        private void heapifyDown(int i) {
                int child;
                int temp = heap[i];
                while (kthChild(i, 1) < heapSize) {
                        child = maxChild(i);
                        if (temp >= heap[child]) {
                                break;
                        }
                        heap[i] = heap[child];
                        i = child;
                }
                heap[i] = temp;
        }
```