目录

1. 常用数据结构简介

https://www.jianshu.com/p/44a1e5bc077a

这些类均在 java.util 包中,结构如下:

Collection

1.List 子接口

1) LinkedList

2) ArrayList

3) Vector

4) Stack

2.Set 接口

Map

1) Hashtable

2) HashMap

3) WeakHashMap

Collection 接口

Collection 是最基本的集合接口，一个 Collection 代表一组 Object，即 Collection 的元素（Elements）。一些 Collection 允许相同的元素而另一些不行。一些能排序而另一些不行。（Java SDK 不提供直接继承自 Collection 的类，Java SDK 提供的类都是继承自

Collection 的“子接口”如 List 和 Set。）

所有实现 Collection 接口的类都必须提供两个标准的构造函数：

无参数的构造函数用于创建一个空的 Collection，有一个 Collection 参数的构造函数用于创建一个新的 Collection，这个新的 Collection 与传入的 Collection 有相同的元素。

后一个**构造函数允许用户复制一个 Collection**。

如何遍历 Collection 中的每一个元素？

不论 Collection 的实际类型如何，它都支持一个 iterator()的方法，该方法返回一个迭代子，使用该迭代子即可逐一访问 Collection 中每一个元素。典型的用法如下：

```
Iterator it = collection.iterator(); // 获得一个迭代子
while(it.hasNext()) {
Object obj = it.next(); // 得到下一个元素
}
```

由 Collection 接口派生的两个接口是 List 和 Set。

主要方法:

✓ boolean add(Object o)：添加对象到集合

✓ boolean remove(Object o)：删除指定的对象

✓ int size()：返回当前集合中元素的数量

✓ boolean contains(Object o)：查找集合中是否有指定的对象

- ✓ boolean isEmpty()：判断集合是否为空
- ✓ Iterator iterator()：返回一个迭代器
- ✓ boolean containsAll(Collection c)：查找集合中是否有集合 c 中的元素
- ✓ boolean addAll(Collection c)：将集合 c 中所有的元素添加给该集合
- ✓ void clear()：删除集合中所有元素
- ✓ void removeAll(Collection c)：从集合中删除 c 集合中也有的元素
- ✓ void retainAll(Collection c)：从集合中删除集合 c 中不包含的元素

（1）List 接口

List 是有序的 Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在 List 中的位置，类似于数组下标）来访问 List 中的元素，这类似于 Java 的数组。

和下面要提到的 Set 不同，List 允许有相同的元素。

除了具有 Collection 接口必备的 iterator()方法外，List 还提供一个 listIterator()方法，返回一个 ListIterator 接口，和标准的 Iterator 接口相比，ListIterator 多了一些 add()之类的方法，允许添加，删除，设定元素，还能向前或向后遍历。

实现 List 接口的常用类有 LinkedList，ArrayList，Vector 和

Stack。

主要方法:

- ✓ void add(int index,Object element)：在指定位置上添加一个对象
- ✓ boolean addAll(int index,Collection c)：将集合 c 的元素添加到指定的位置
- ✓ Object get(int index)：返回 List 中指定位置的元素
- ✓ int indexOf(Object o)：返回第一个出现元素 o 的位置.
- ✓ Object removeint(int index)：删除指定位置的元素
- ✓ Object set(int index,Object element)：用元素 element 取代位置 index 上的元素,返回被取代的元素

1）LinkedList 类

LinkedList 实现了 List 接口，允许 null 元素。此外 LinkedList 提供额外的 get，remove，insert 方法在 LinkedList 的首部或尾部。这些操作使 LinkedList 可被用作堆栈（stack），队列（queue）或双向队列（deque）。

注意：LinkedList 没有同步方法。如果多个线程同时访问一个 List，则必须自己实现访问同步。一种解决方法是在创建 List 时构造一个同步的 List：

```
List list = Collections.synchronizedList(new LinkedList(...));
```

2）ArrayList 类

ArrayList 实现了可变大小的数组。它允许所有元素，包括 null。ArrayList 没有同步。

size，isEmpty，get，set 方法运行时间为常数。但是 add 方法开销为分摊的常数，添加 n 个元素需要 O(n)的时间。其他的方法运行时间为线性。

每个 ArrayList 实例都有一个容量（Capacity），即用于存储元素的数组的大小。这个容量可随着不断添加新元素而自动增加，但是增长算法并没有定义。当需要插入大量元素时，在插入前可以调用 ensureCapacity 方法来增加 ArrayList 的容量以提高插入效率。

和 LinkedList 一样，ArrayList 也是非同步的（unsynchronized）。

主要方法:

- ✓ Boolean add(Object o)将指定元素添加到列表的末尾
- ✓ Boolean add(int index,Object element)在列表中指定位置加入指定元素
- ✓ Boolean addAll(Collection c)将指定集合添加到列表末尾
- ✓ Boolean addAll(int index,Collection c)在列表中指定位置加入指定集合
- ✓ Boolean clear()删除列表中所有元素
- ✓ Boolean clone()返回该列表实例的一个拷贝
- ✓ Boolean contains(Object o)判断列表中是否包含元素

- ✓ Boolean ensureCapacity(int m)增加列表的容量,如果必须,该列表能够容纳 m 个元素
- ✓ Object get(int index)返回列表中指定位置的元素
- ✓ Int indexOf(Object elem)在列表中查找指定元素的下标
- ✓ Int size()返回当前列表的元素个数

3）Vector 类

Vector 非常类似 ArrayList，但是 Vector 是同步的。由 Vector 创建的 Iterator，虽然和 ArrayList 创建的 Iterator 是同一接口，但是，因为 Vector 是同步的，当一个 Iterator 被创建而且正在被使用，另一个线程改变了 Vector 的状态（例如，添加或删除了一些元素），这时调用 Iterator 的方法时将抛出 ConcurrentModificationException，因此必须捕获该异常。

4）Stack 类

Stack 继承自 Vector，实现一个后进先出的堆栈。Stack 提供 5 个额外的方法使得 Vector 得以被当作堆栈使用。基本的 push 和 pop 方法，还有 peek 方法得到栈顶的元素，empty 方法测试堆栈是否为空，search 方法检测一个元素在堆栈中的位置。Stack 刚创建后是空栈。

Set 接口

Set 是一种不包含重复的元素的 Collection，即任意的两个元素

e1 和 e2 都有 e1.equals(e2)=false，Set 最多有一个 null 元素。很明显，Set 的构造函数有一个约束条件，传入的 Collection 参数不能包含重复的元素。

注意：必须小心操作可变对象（Mutable Object）。如果一个 Set 中的可变元素改变了自身状态导致 Object.equals(Object)=true 将导致一些问题。

Map 接口

Map 没有继承 Collection 接口，Map 提供 key 到 value 的映射。一个 Map 中不能包含相同的 key，每个 key 只能映射一个 value。Map 接口提供 3 种集合的视图，Map 的内容可以被当作一组 key 集合，一组 value 集合，或者一组 key-value 映射。

主要方法:

- ✓ boolean equals(Object o)比较对象
- ✓ boolean remove(Object o)删除一个对象
- ✓ put(Object key,Object value)添加 key 和 value

1）Hashtable 类

Hashtable 继承 Map 接口，实现一个 key-value 映射的哈希表。任何非空（non-null）的对象都可作为 key 或者 value。

添加数据使用 put(key, value)，取出数据使用 get(key)，这两个基本操作的时间开销为常数。

Hashtable 通过 initial capacity 和 load factor 两个参数调整性能。通常缺省的 load factor 0.75 较好地实现了时间和空间的均衡。增大 load factor 可以节省空间但相应的查找时间将增大，这会影响像 get 和 put 这样的操作。

```
Hashtable numbers = new Hashtable();
numbers.put(“one”, new Integer(1));
numbers.put(“two”, new Integer(2));
numbers.put(“three”, new Integer(3));
```

要取出一个数，比如2，用相应的key：

```
Integer n = (Integer)numbers.get(“two”);
System.out.println(“two = ” + n);
```

由于作为 key 的对象将通过计算其散列函数来确定与之对应的 value 的位置，因此任何作为 key 的对象都必须实现 hashCode 和 equals 方法。

hashCode 和 equals 方法继承自根类 Object，如果你用自定义的类当作 key 的话，要相当小心，按照散列函数的定义，如果两个对象相同，即 obj1.equals(obj2)=true，则它们的 hashCode 必须相同。

但如果两个对象不同，则它们的 hashCode 不一定不同，如果两个不同对象的 hashCode 相同，这种现象称为冲突，冲突会导致操作哈希表的时间开销增大，所以尽量定义好的 hashCode()方法，能加快哈希表的操作。

如果相同的对象有不同的 hashCode，对哈希表的操作会出现意想不到的结果（期待的 get 方法返回 null），要避免这种问题，只

需要牢记一条：要同时复写 equals 方法和 hashCode 方法，而不要只写其中一个。

Hashtable 是同步的。

2）HashMap 类

HashMap 和 Hashtable 类似,不同之处在于 HashMap 是非同步的，并且允许 null，即 null value 和 null key。，但是将 HashMap 视为 Collection 时（values()方法可返回 Collection），其迭代子操作时间开销和 HashMap 的容量成比例。因此，如果迭代操作的性能相当重要的话,不要将 HashMap 的初始化容量设得过高,或者 load factor 过低。

3）WeakHashMap 类

WeakHashMap 是一种改进的 HashMap，它对 key 实行“弱引用”，如果一个 key 不再被外部所引用，那么该 key 可以被 GC 回收。

总结

（1）如果涉及到堆栈，队列等操作，应该考虑用 List，对于需要快速插入，删除元素，应该使用 LinkedList，如果需要快速随机访问元素，应该使用 ArrayList。

（2）如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高，如果多个线程可能同时操作一个

类，应该使用同步的类。

（3）要特别注意对哈希表的操作，作为 key 的对象要正确复写 equals 和 hashCode 方法。

（4）尽量返回接口而非实际的类型，如返回 List 而非 ArrayList，这样如果以后需要将 ArrayList 换成 LinkedList 时，客户端代码不用改变。这就是针对抽象编程。

2. 数组和链表的区别

1.从内存存储角度：

1) 数组从栈中分配空间，对程序员方便快速，自由度小。

2) 链表从堆中分配内存。自由度大但申请管理比较麻烦。

2.从逻辑结构角度：

1) 数组必须事先定义固定的长度（元素个数），不能适应数据动态地增减情况（数据插入、删除比较麻烦）。当数据增加时，可能会超出数组的最大空间（越界）；当数据减少时，造成内存浪费。

2) 链表动态地进行存储分配，可以适应数据动态地增减情况（数据插入删除简单）（数组中插入、删除数据项时，需要移动其他项），但链表查找元素时需要遍历整个链表。

3. 二叉树的深度优先遍历和广度优先遍历的具体实现

https://blog.csdn.net/fantasy_lin_/article/details/52751559

二叉树的深度优先遍历的非递归的通用做法是采用栈，广度优先遍历的非递归的通用做法是采用队列。

1.**深度优先遍历：**对每一个可能的分支路径深入到不能再深入为止，而且每个结点只能访问一次。要特别注意的是，二叉树的深度优先遍历比较特殊，可以细分为**先序遍历、中序遍历、后序遍历**。具体说明如下：

（1）**先序遍历：**对任一子树，先访问根，然后遍历其左子树，最后遍历其右子树。

（2）**中序遍历：**对任一子树，先遍历其左子树，然后访问根，最后遍历其右子树。

（3）**后序遍历：**对任一子树，先遍历其左子树，然后遍历其右子树，最后访问根。

关于先序遍历、中序遍历、后序遍历，代码在后序有：

2.广度优先遍历：又叫层次遍历，从上往下对每一层依次访问，在每一层中，从左往右（也可以从右往左）访问结点，访问完一层就进入下一层，直到没有结点可以访问为止。

关于广度优先遍历，代码在后序有：

二叉树插入方法代码：

```
public void insertNode(E value) {
    if (root == null) {
        root = new Node<E>(value);
        return;
    }
    Node<E> currentNode = root;
    while (true) {
        if (value.compareTo(currentNode.value) > 0) {
            if (currentNode.right == null) {
                currentNode.right = new Node<E>(value);
                break;
            }
            currentNode = currentNode.right;
        } else {
            if (currentNode.left == null) {
                currentNode.left = new Node<E>(value);
                break;
            }
            currentNode = currentNode.left;
        }
    }
}
```

关于先序遍历、中序遍历、后序遍历的代码：

分非递归性代码及递归性代码：

递归性代码：

（1）先序遍历

```
/**
 * 先序遍历二叉树（递归）
 * @param node
 */
public void preOrderTraverse(Node<E> node) {
    System.out.print(node.value + " ");
    if (node.left != null)
            preOrderTraverse(node.left);
    if (node.right != null)
            preOrderTraverse(node.right);
}
```

（2）中序遍历

```java
/**
 * 中序遍历二叉树（递归）
 * @param node
 */
public void inOrderTraverse(Node<E> node) {
    if (node.left != null)
            inOrderTraverse(node.left);
    System.out.print(node.value + " ");
    if (node.right != null)
            inOrderTraverse(node.right);
}
```

（3）后序遍历

```java
/**
 * 后序遍历二叉树（递归）
 * @param node
 */
public void postOrderTraverse(Node<E> node) {
    if (node.left != null)
            postOrderTraverse(node.left);
    if (node.right != null)
            postOrderTraverse(node.right);
    System.out.print(node.value + " ");
}
```

pop()
从此列表表示的堆栈中弹出一个元素。

push(E e)
将元素推送到由此列表表示的堆栈上。

Pop 是指把第一个弹出来，push 是将数放到第一个

非递归性代码：

（1）先序遍历（非递归）

```
/**
 * 先序遍历二叉树（非递归）
 * @param root
 */
public void preOrderTraverseNoRecursion(Node<E> root) {
    LinkedList<Node<E>> stack = new LinkedList<Node<E>>();
    Node<E> currentNode = null;
    stack.push(root);
    while (!stack.isEmpty()) {
            currentNode = stack.pop();
            System.out.print(currentNode.value + " ");
            if (currentNode.right != null)
                    stack.push(currentNode.right);
            if (currentNode.left != null)
                    stack.push(currentNode.left);
    }
}
```

（2）中序遍历（非递归）

```java
/**
 * 中序遍历二叉树（非递归）
 * @param root
 */
public void inOrderTraverseNoRecursion(Node<E> root) {
    LinkedList<Node<E>> stack = new LinkedList<Node<E>>();
    Node<E> currentNode = root;
    while (currentNode != null || !stack.isEmpty()) {
            // 一直循环到二叉排序树最左端的叶子结点（currentNode是null）
            while (currentNode != null) {
                    stack.push(currentNode);
                    currentNode = currentNode.left;
            }
                    currentNode = stack.pop();
                    System.out.print(currentNode.value + " ");
                    currentNode = currentNode.right;
    }
}
```

（3）后序遍历（非递归）

```java
/**
 * 后序遍历二叉树（非递归）
 * @param root
 */
public void postOrderTraverseNoRecursion(Node<E> root) {
    LinkedList<Node<E>> stack = new LinkedList<Node<E>>();
    Node<E> currentNode = root;
    Node<E> rightNode = null;
    while (currentNode != null || !stack.isEmpty()) {
            // 一直循环到二叉排序树最左端的叶子结点（currentNode是null）
            while (currentNode != null) {
                    stack.push(currentNode);
                    currentNode = currentNode.left;
            }
            currentNode = stack.pop();
            // 当前结点没有右结点或上一个结点（已经输出的结点）是当前结点的右结点，则输出当前结
            while (currentNode.right == null || currentNode.right == rightNode) {
                    System.out.print(currentNode.value + " ");
                    rightNode = currentNode;
                    if (stack.isEmpty()) {
                            return; //root以输出，则遍历结束
                    }
                    currentNode = stack.pop();
            }
            stack.push(currentNode); //还有右结点没有遍历
            currentNode = currentNode.right;
    }
}
```

广度优先遍历的代码

```java
/**
 * 广度优先遍历二叉树，又称层次遍历二叉树
 * @param node
 */
public void breadthFirstTraverse(Node<E> root) {
    Queue<Node<E>> queue = new LinkedList<Node<E>>();
    Node<E> currentNode = null;
    queue.offer(root);
    while (!queue.isEmpty()) {
            currentNode = queue.poll();
            System.out.print(currentNode.value + " ");
            if (currentNode.left != null)
                    queue.offer(currentNode.left);
            if (currentNode.right != null)
                    queue.offer(currentNode.right);
    }
}
```

4. 堆的结构

https://blog.csdn.net/1294265421/article/details/50927538

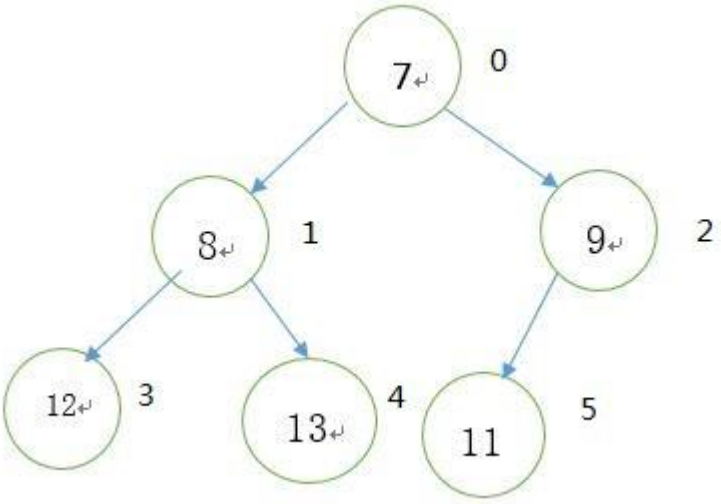
堆是一颗完全二叉树。

在这棵树中，所有父节点都满足大于等于其子节点的堆叫大根堆。

所有父节点都满足小于等于其子节点的堆叫小根堆。

堆虽然是一颗树，但是通常存放在一个数组中，**父节点和孩子节**

点的父子关系通过数组下标来确定。如下图的小根堆及存储它的数组：



创建

```java
public int left(int i) {
    return (i + 1) * 2 - 1;
}

public int right(int i) {
    return (i + 1) * 2;
}

public int parent(int i) {
    //i为根结点
    if (i == 0) {
        return -1;
    }
    return (i - 1) / 2;
}
```

除了知道怎么计算一个节点的父节点和孩子节点的索引外，我们还需要两个算法，即**保持堆的性质**和**建堆**。我们将看到建堆的方法对大根堆和小根堆也是一样的。

保持堆的性质：

保持堆的性质，对大根堆和小根堆很相似，但不完全一样，所以得分开说。

（1）**对大根堆来说是：**已经存在两个大根堆，现在要把一个元素作为这两个大根堆根的父节点，构成一个新的堆，但是这个堆的根结点

可能不满足大根堆的性质，也就是说，它可能比它的孩子节点小，所以需要对它进行操作。

操作的方式就是，我们从这个节点和它的孩子节点中选出最大的，如果最大的节点是这个节点本身，堆就已经满足大根堆性质了。

否则，将这个节点与最大节点交换，交换后该节点在新的位置上也可能违背大根堆性质，所以需要递归的进行，直至这个节点比孩子节点都大或者是子节点为止（这个过程也叫做元素下降，因为元素从根结点开始一步一步往下移）（符合就成了；不符合就交换数值然后递归，直到符合成了）。

```
public void heapify(T[] a, int i, int heapLength) {
    int l = left(i);
    int r = right(i);
    int largest = -1;
    /**
 * 下面两个if条件句用来找到三个元素中的最大元素的位置largest；
 * l < heapLength说明l在数组内，i非叶子结点；
 */
    if (l < heapLength && a[i].compareTo(a[l]) < 0) {
        largest = l;
    } else {
        largest = i;
    }
    //r < heapLength说明r在数组内
    if (r < heapLength && a[largest].compareTo(a[r]) < 0) {
        largest = r;
    }
    //如果i处元素不是最大的，就把i处的元素与最大处元素交换，交换会使元素下降
    if (i != largest) {
        T temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;
        //交换元素后，以a[i]为根的树可能不在满足大根堆性质，于是递归调用该方法
        heapify(a, largest, heapLength);
    }
}
```

（2）**对小根堆**：保持堆的性质，过程是，已经存在两个小根堆，现在要把一个元素作为这两个堆的根，作为新的小根堆，但是这个元素可能会违背小根堆的性质，所以需要将它下降（也就是说，不断将它与它和孩子节点中的最小节点交换，直到它是它和它孩子节点中最小的或者是叶子节点为止）。对应的 Java 代码如下：

```
public void heapify(T[] a, int i, int heapLength) {
    int l = left(i);
    int r = right(i);
    int smallest = -1;
    /**
 * 下面两个if条件句用来找到三个元素中的最小元素的位置smallest；
 * s < heapLength说明l在数组内，i非叶子结点；
 */
    if (l < heapLength && a[i].compareTo(a[l]) > 0) {
        smallest = l;
    } else {
        smallest = i;
    }
    // r < heapLength说明r在数组内
    if (r < heapLength && a[smallest].compareTo(a[r]) > 0) {
        smallest = r;
    }
    // 如果i处元素不是最小的，就把i处的元素与最小处元素交换，交换会使元素下降
    if (i != smallest) {
        T temp = a[i];
        a[i] = a[smallest];
        a[smallest] = temp;
        // 交换元素后，以a[i]为根的树可能不在满足大根堆性质，于是递归调用该方法
        heapify(a, smallest, heapLength);
    }
}
```

有了上面的这些准备工作，我们终于可以建堆了。正如前面所说，建堆的过程对大根堆和小根堆是一样的。我们可以把单个元素看作大根堆或者小根堆。假设数组中最后一个堆元素的下标为 i，则数组中

从 0 下标开始，最后一个有孩子节点的元素就是 j=parent(i)。于是，我们从 j 到 0，对一个元素都调用 heapify 方法，堆就建好了。下图是算法导论给出的伪代码：

```
public  void buildHeap(T[] a, int heapLength) {
    //从后往前看，lengthParent - 1处的元素是第一个有孩子节点的节点
    int lengthParent = parent(heapLength - 1);
    //最初，parent(length)之后的所有元素都是叶子结点；
    //因为大于length/2处元素的孩子节点如果存在，那么
    //它们的数组下标值必定大于length，这与事实不符；
    //在数组中，孩子元素必定在父亲元素的后面，从后往前
    //对元素调用maxHeapify，保证了元素的孩子都是
    //大根堆
    for(int i = lengthParent; i >= 0; i--){
        heapify(a, i, heapLength);
    }
}
```

可以用堆构建优先级队列，因为我们可以在常数时间内获得堆中（优先级）最大或者最小的元素，这对于优先级队列来说是很重要的。（可能就是排大根堆跟小根堆）

5. 堆和树的区别

来一个离题的开头：

heap 和 tree 结合，生了个孩子叫 treap。中文名叫树堆。

首先它每个节点有 2 个值 value 和 weight

其中只看 weight 的话，满足 heap 二叉堆的特性(父亲比儿子都

小/大)，只看 value 的话，满足排序二叉树特性(以左儿子为根的子树元素都比父亲小，右儿子为根的子树都比父亲大)

value 是要维护的值，weight 是随机生成的值。由于随机生成的堆使整棵 treap 变得平衡(严格证明请谷歌百度～)，所以 treap 是一种比较短小精悍的平衡树的实现～

只要无环无向联通图都叫树，具体就是 n 个点 n-1 条无向边连接且任意两点联通的一种拓扑结构

如果我们选定一个节点作为根，那么这棵树就是有根树，遍历一遍就可以确定所有的父亲-儿子的关系了。。。

如果一棵有根树的每一个结点至多有两个儿子，那么这棵树称为**二叉树**

如果一棵二叉树的每一个节点都带着一个值，且父亲的值总是比儿子的值要大，我们称这棵树为**大顶二叉堆**，如果是父亲比儿子都要小，那就是**小顶二叉堆**，统称为**二叉堆**。(其实一般都把二叉两个字省略掉，毕竟通常说的堆都是二叉堆，然而堆不止二叉堆)。

这一个良好的性质注定了堆可以用来当作优先队列使用。

优先队列支持以下操作：

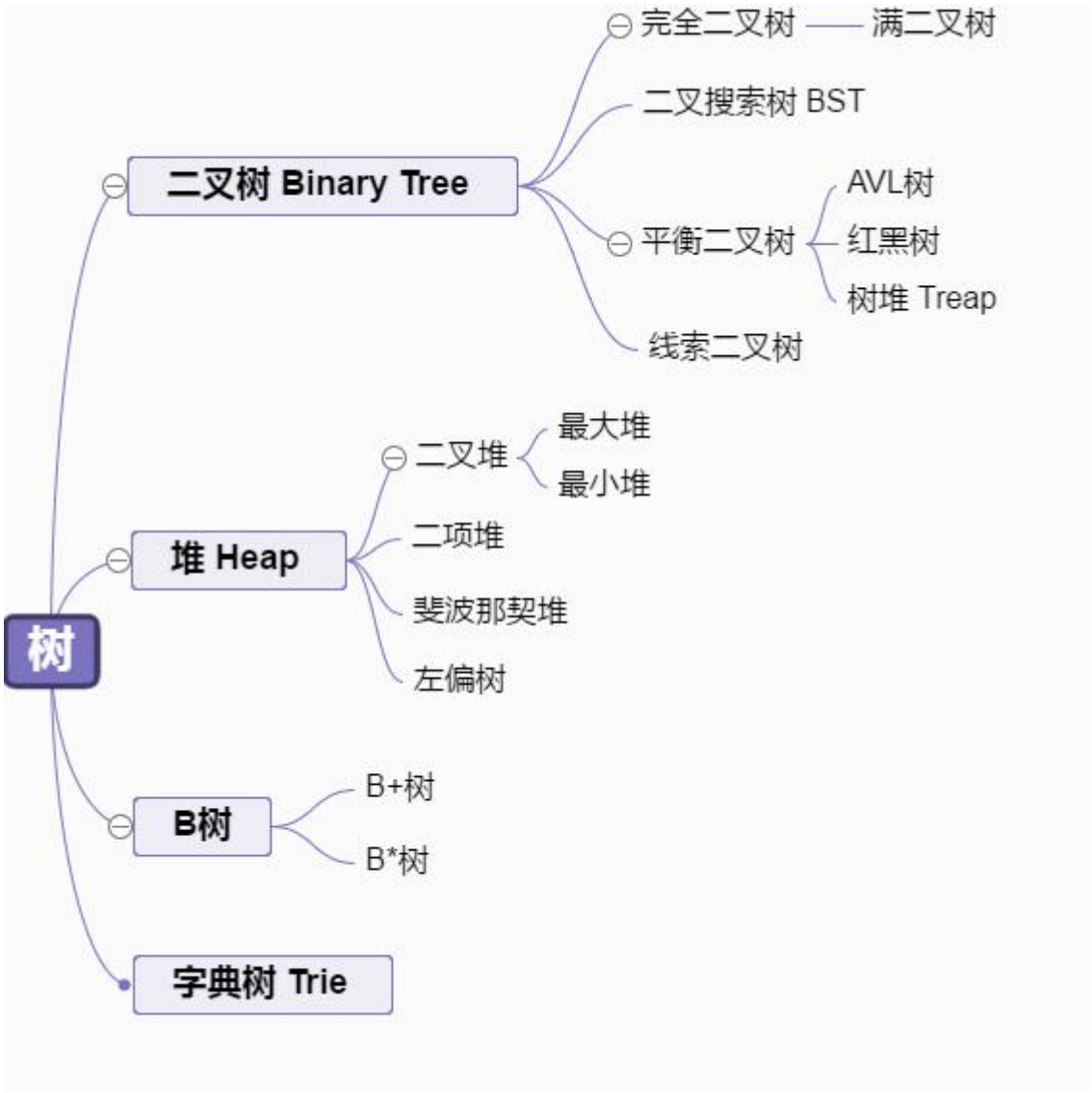
1. 放一个元素进去
2. 总是能取出一个最大的元素出来(大，小的规矩可以通过一个比较函数来定义)

显然堆就是可以这么做。

堆不止二叉堆，还有更复杂的二项堆，斐波那契堆，配对堆等

等。。。

总结，堆是一种特殊的树。



6. 堆和栈在内存中的区别是什么(解答提示：可以从数据结构方面以及实际实现方面两个方面去回答)？

https://www.jianshu.com/p/947a76e2ddbc

java 中的内存，分为两种，一为堆内存，二为栈内存。

栈内存

在函数中定义的基本类型的变量和对象的引用变量都是在函数的栈内存中分配。

当在一段代码块中声明了一个变量时，java 就会在栈内存中为这个变量分配内存空间，当超过变量的作用域之后，java 也会自动释放为该变量分配的空间，而这个回收的空间可以即刻用作他用。

堆内存

堆内存用于存放由 new 创建的对象和数组。

在堆内存中分配的内存空间，由 java 虚拟机自动垃圾回收器来管理。

在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，变量的值就等于数组或对象在堆内存中的首地址，而这个栈中的特殊变量，也就成为数组或对象的引用变量。以后可以在程序中使用栈内存中的引用变量访问堆内存中的数组或对象了。引用变量相当于是为数组或对象起的一个别名，或者是代号。

数组和对象在没有引用变量指向它的时候，才变成垃圾，不能被继续使用，但是仍然会占用堆内存空间，而后在一个不确定的时间内，由 java 虚拟机自动垃圾回收器回收，这也是 java 程序为什么会占用很大内存的原因。

java 中的内存分配策略及堆和栈的比较

1. 内存分配策略

根据编译原理的观点，程序运行时的内存分配，有三种策略，分别为**静态的、堆式的、栈式的**。

(1)静态存储分配：指的是在**编译时就能确定每个数据目标在运行时的存储空间需求**，因而在编译时就给它们分配了固定的内存空间。这种分配方式要求程序代码中不能有可变数据结构(例如可变数组)的存在，也不允许有嵌套或者递归的结构出现，因为它们都会导致编译时编译程序无法准确计算所需的存储空间大小。

(2)栈式存储分配：也可以成为动态存储分配，是由一个类似于堆栈的运行栈来实现的。和静态存储分配相反，在栈式分配方案中，**程序对于存储空间的需要在编译时是未知的，只有在运行的时候才能知道**。但是规定在运行进入一个程序模块时，必须要知道该程序模块所需的数据区大小才能为其分配内存。同我们日常了解的栈一致，栈式存储分配遵照先进后出的原则进行分配。

(3)堆式存储分配：专门负责在编译时或运行时程序模块入口处都无法确定存储要求的数据结构的分配，比如可变长度串和对象实例。堆内存由大片的可利用块或空闲块组成，堆中的内存也可以根据任意的顺序进行分配和释放。

堆和栈的比较

从通俗化的角度来说，**堆是用来存放对象的，栈是用来存放执行**

程序的

栈：

在编程中（例如在 C/C++中），所有的方法调用都是通过栈来进行的，所有的局部变量、形式参数也都是通过栈来分配内存的。

使用的时候，根据栈的工作原理，从栈顶向上用即可，Stack Pointer 会自动指引程序到存储位置，程序只需要进行存储即可。退出函数的时候，修改栈指针即可将栈中存储的内容销毁。

这种模块速度最快，适合用户存储执行程序。但是应当注意的是，在进行内存分配是，比如为一个即将要调用的程序模块分配数据区时，应当实现知道这个所需数据区的大小，也就是说虽然分配工作是在运行程序的时候进行的，但是分配的大小是在运行程序之前就知道的，这个是编译时确定的，而不是运行时。

堆：

是应用程序在运行过程中请求操作系统给分配的内存，由于是操作系统管理的内存分配，所以在分配和销毁是都需要占用时间，因此堆的工作效率比较低。

但是堆的优点在于，编译器不需要知道从堆中分配了多少的内存空间，也不需要知道存储的数据要在堆中停留多长的时间，这也就使得用堆来保存数据有着更大的灵活性。

事实上，面向对象的多态性的实现，堆内存的分配必不可少，因为多态对象所需的数据区大小只有在运行时确定了对象以后才能知

道。在 java 中，创建对象只需要使用 new 关键字即可，执行这些代码时，就会在堆中自动进行数据的保存。也就是因为这种灵活分配存储空间的特性，堆内存分配的工作效率不高。

JVM 中的堆和栈

堆和栈都是 java 用来在内存中存放数据的地方，与 C++不同的是，java 自动管理堆和栈，程序员不能自行设置堆和栈。

java 中的堆就是运行时存储数据的区域，类的实例对象可以通过 new、new Array 等指令建立从中分配空间，这些空间不需要程序代码来显式释放。堆是由 jvm 自动垃圾回收器负责的，堆的优势是可以动态的分配内存大小，生存周期也不用实现告诉编译器，因为空间是在运行时动态进行内存分配的。如果堆中的数据确认为垃圾，则 jvm 的自动垃圾回收器汇自动回收相应的空间。但是缺点是，由于实在运行时动态进行空间分配，存取速度较慢。

栈的优势是：存取的速度都比堆要快，仅次于寄存器。栈数据可以共享，但是缺点时，栈空间中的数据大小和生存期必须是确定的，缺乏灵活性。栈主要存放一些基本类型的变量 int, short, long, byte, float, double, boolean, char 和对象句柄。

栈有一个很重要的特殊性，就是存在栈中的数据可以共享。

栈有一个很重要的特殊性，就是存在栈中的数据可以共享。假设我们同时定义：

```
int a = 3;
int b = 3;
```

编译器先处理int a = 3;首先它会在栈中创建一个变量为a的引用，然后查找栈中是否有3这个值，如果没找到，就将3存放进来，然后将a指向3。接着处理int b = 3;在创建完b的引用变量后，因为在栈中已经有3这个值，便将b直接指向3。这样，就出现了a与b同时均指向3的情况。这时，如果再令a=4;那么编译器会重新搜索栈中是否有4值，如果没有，则将4存放进来，并令a指向4;如果已经有了，则直接将a指向这个地址。因此a值的改变不会影响到b的值。要注意这种数据的共享与两个对象的引用同时指向一个对象的这种共享是不同的，因为这种情况a的修改并不会影响到b, 它是由编译器完成的，它有利于节省空间。而一个对象引用变量修改了这个对象的内部状态，会影响到另一个对象引用变量。

7. 什么是深拷贝和浅拷贝

https://blog.csdn.net/u014727260/article/details/55003402

浅拷贝：所谓的浅拷贝就是拷贝指向对象的指针（拷贝出来的目标对象的指针和源对象的指针指向的内存空间是同一块空间）。

浅拷贝只是一种简单的拷贝, 让几个对象公用一个内存, 然而当内存销毁的时候, 指向这个内存空间的所有指针需要重新定义, 不然会造成野指针错误

深拷贝：所谓的深拷贝指拷贝对象的具体内容, 其内容地址是自助分配的, 拷贝结束之后, 内存中的值是完全相同的, 但是内存地址是不一样的, 两个对象之间相互不影响, 也互不干涉.

我们来总结一下两者之间的原理 ： 如果现在有一个 A 对象,拷贝之后得到一份新的对象 A_Copy。

如果时浅拷贝,那么 A 对象和 A_Copy 对象指向的就是同一个内存的资源,它拷贝的只是一个指针而已,对象的内容并没有拷贝.也就是说对象的资源还是只有一份.如果这个时候我们对 A_copy 对象进行修改操作,那么 A 对象的内容同样会被修改.

如果是深拷贝,拷贝的不仅仅是指针,还有内容,拷贝的对象 B_Copy 会自助分配内存,两个对象的指针指向的是不同的内存空间,因为 A 对象和 B_Copy 对象的内存地址是不一样的,所以,如果我们对 B_Copy 进行修改操作的话是不会影响到 A 对象,它们之间是互不干涉的

总结：

浅拷贝就想是您和您的影子之间的关系 ： 你挂了， 你的影子也跟着挂了

深拷贝就像是您的克隆人， 你挂啦， 可你的克隆人还活着

8. 手写链表逆序代码

https://blog.csdn.net/u012571415/article/details/46955535

```java
public class SingleLinkedReverse {

    //定义单链表，每个节点包含数据和下一个节点的地址
    class Node {
        int data;
        Node next;

        public Node(int data) {
            this.data = data;
        }
    }
```

```java
public static void main(String[] args) {
    SingleLinkedReverse slr = new SingleLinkedReverse();
    //定义两个链表
    Node head, tail;
    //新建一个链表头结点
    head = tail = slr.new Node(0);
    //新建一个长度为10的链表tail
    for (int i = 1; i < 10; i++) {
        Node p = slr.new Node(i);
        tail.next = p;
        tail = p;
    }
    //tail返回到头节点0 此时head在头结点
    tail = head;
    //循环链表
    while (tail != null) {
        System.out.print(tail.data + "  ");
        tail = tail.next;
    }

    head = reverse(head);

    while (head != null) {
        System.out.print(head.data + "  ");
        head = head.next;
    }
}
```

```java
/* 核心算法
   前后交换位置
*/
private static Node reverse(Node head) {
    Node p1, p2 = null;
    p1 = head;

    while (head.next != null) {
        p2 = head.next;
        head.next = p2.next;
        p2.next = p1;
        p1 = p2;
    }
    return p2;
}
```

Node 是 org.w3c.dom 包下定义接口其子接口很多

获取两个数，将第一个数放在第二个数的后面，也就是链表后面

比如先获取了 1,2 两个数，将 1 放在 2 后面，形成链表 A（2,1）

当继续原列表中不为空时，继续读，并将链表 A 放在其后。如不为空，再读出来是 3，将 A 放在 3 后面，形成新的链表 P2（3,2,1,），以此类推

Head.next = p2.next 只是将列表的数往前移，为了更方便的判断下一个数

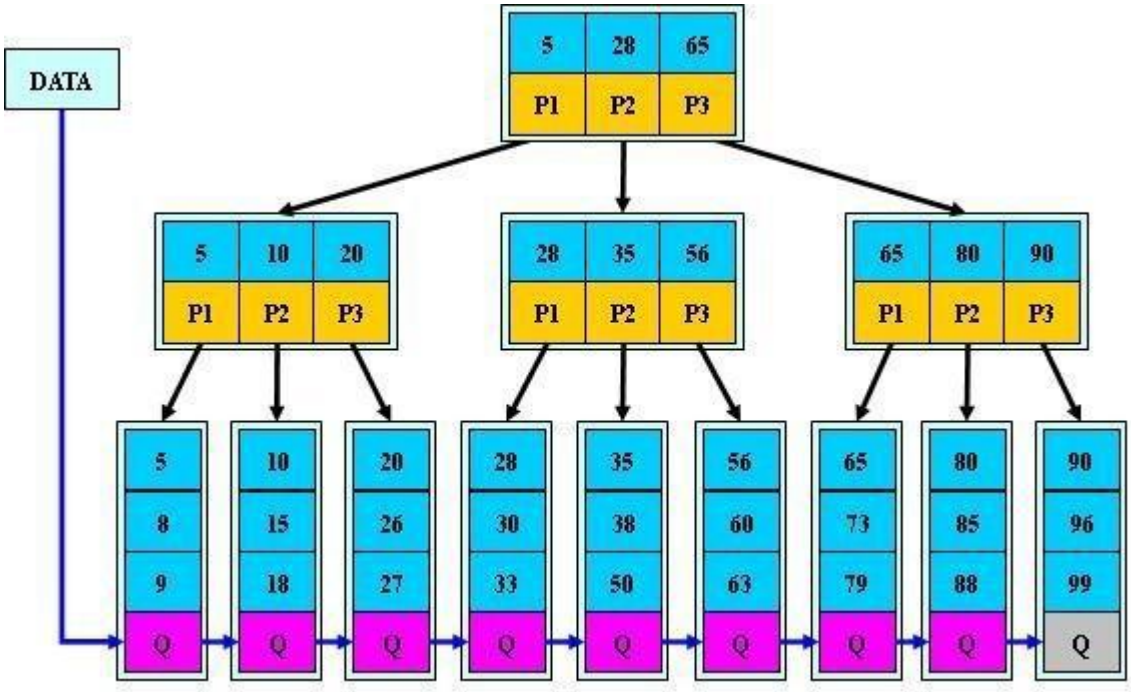
9. 讲一下对树，B+树的理解

https://www.jianshu.com/p/6f68d3c118d6

B 树是**为磁盘或其他直接存取的辅助存储设备而设计的一种平衡搜索树**。B 树类似于红黑树，但它们在降低磁盘 I/O 操作数方面要更好一些

B+树是 B 树的一种变形，**它把所有的卫星数据都存储在叶节点中，内部节点只存放关键字和孩子指针**，因此最大化了内部节点的分支因子，所以 B+树的遍历也更加高效(B 树需要以中序的方式遍历节点，而 B+树只需把所有叶子节点串成链表就可以从头到尾遍历)。

以下先放一张我所依据的 B+树的图示(这张图有所简化，下面讲完定义后会贴一张更加详细的图，两图本质并无差异)：



（B+树的图示）

B+树的定义如下：

每个节点 node 有下面的属性： n 个关键字

key[1],key[2], … ,key[n]，以非降序存放，使得 key[1]≤key[2]≤…≤key[n]；

isRoot，一个布尔值，如果 node 是根节点，则为 TRUE；否则为 FALSE；

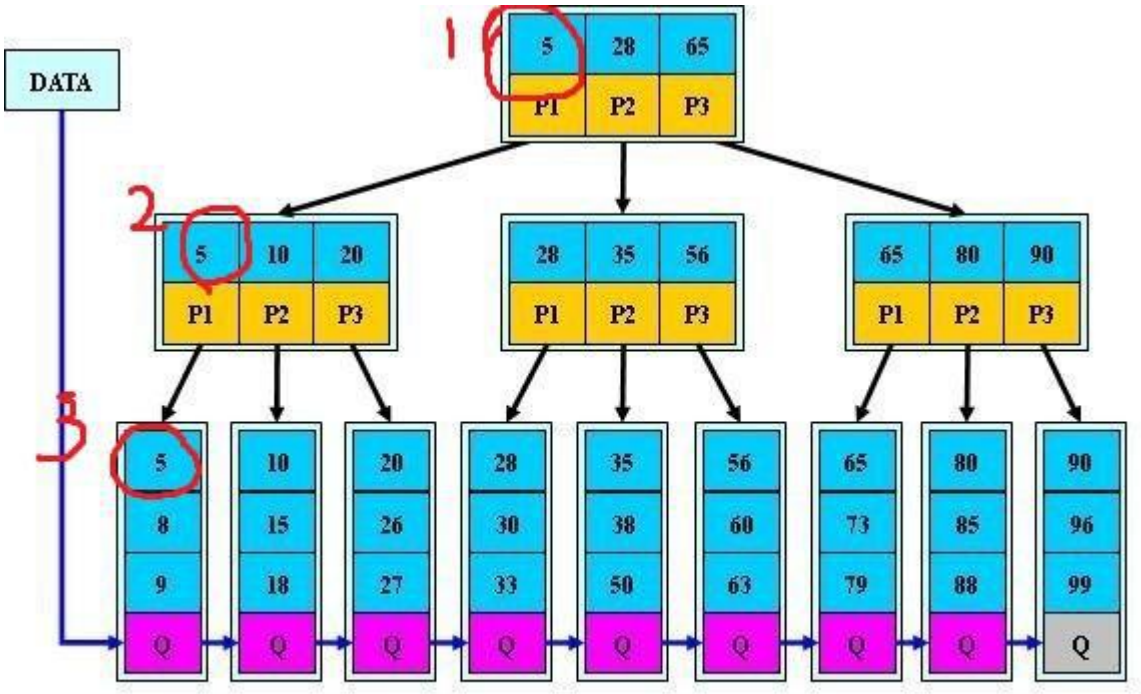
isLeaf，一个布尔值，如果 node 是叶子节点，则为 TRUE；否则为 FALSE；

Node*类型的 parent 指针，指向该节点的父节点

每个内部节点还包含 n 个指向其孩子 children[0],children[1], … , children[n]，叶子节点没有孩子。（注：此处有争议，B+树到底是与 B 树 n-1 个关键字有 n 棵子树保持一致，还是 B+树 n 个关键字的结点中含有 n 棵子树；两种定义都可以，只要自己实现的时候统一用一种就行。如无特殊说明，以下的都是后者：**即 n 个关键字对应 n 棵子树**）；

内部节点的关键字对存储在各子树中的关键字范围加以分割：如果 key[i]为任意一个存储在内部节点中的关键字，childNum[i]为该节点的对应下标的子树指针指向的节点的任意一个关键字，那么 **key[1] ≤ childNum[1] < key[2] ≤ childNum[2] < key[3] ≤ childNum[3] < … < key[n] ≤ childNum[n]**

内部节点并不存储真正的信息，而是保存其叶子节点的最小值作为索引。（比如下图，标注 1 和标注 2 都是内部节点，里面保存的并不是真正的信息，而是标注 3 所示的节点中的最小值。）（注：此处有争议以最大值作为索引，同样也是不影响的争议）



（内部节点图示）

任何和关键字相联系的“卫星数据(satellite information)”将与关键字一样存放在叶子节点中，一般地，可能只是为每个关键字对应的”卫星数据”存放一个指针，指针指向存放实际数据的磁盘页，匹配了某个叶子节点的关键字即可通过该指针找到其他对应数据。

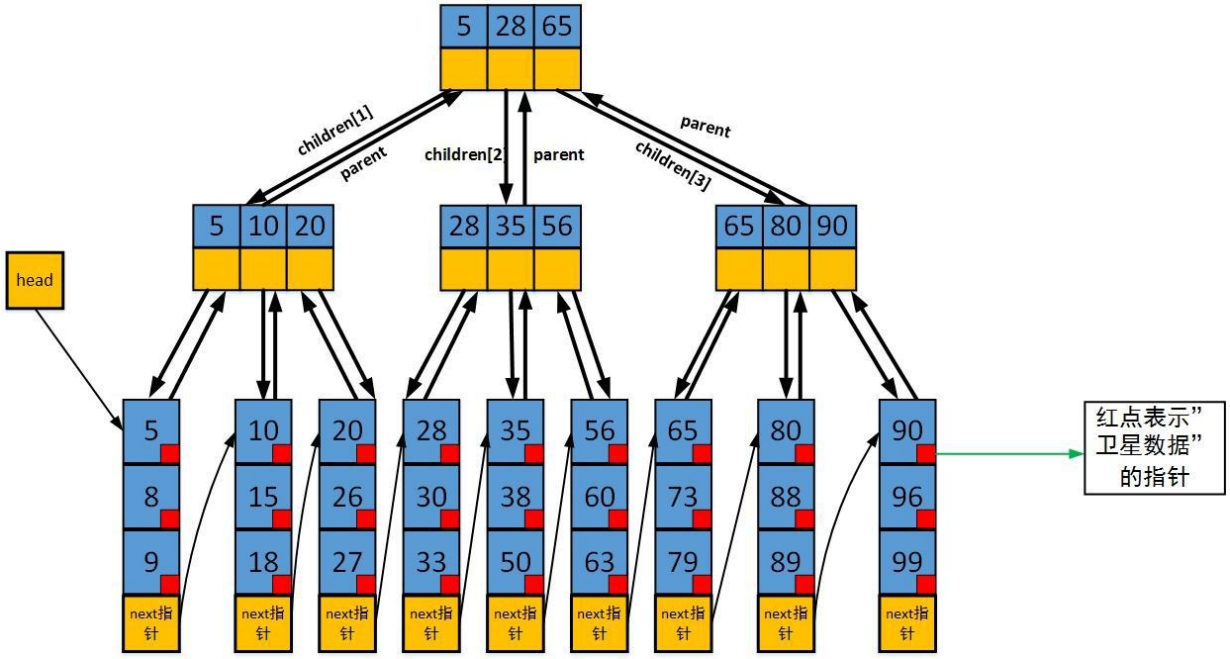
每个叶子节点还有指向下一个节点的指针 next，方便遍历整棵 B+树。

每个叶子节点具有相同的深度，即树的高度 h。

每个节点所包含的关键字个数有上界和下界，用一个被 B+树的**最小度数(minmum degree)**的固定整数 t≥2 来表示这些界： 除了根节点以外的每个节点必须至少有 t 个关键字。因此，除了根节点以外的每个内部节点至少有 t 个孩子

每个节点至多有 2t 个关键字，因此，一个内部节点至多可有 2t 个孩子。当一个节点恰好有 2t 个关键字时，称该节点是**满的**。

结合以上的具体定义，下面这张图更加详细的描述了一棵具体的B+树



（B+树图示）

B+树还有一个最大的好处，方便扫库，B 树必须用中序遍历的方法按序扫库，而 B+树直接从叶子结点挨个扫一遍就完了，B+树支持 range-query 非常方便，而 B 树不支持。这是数据库选用 B+树的最主要原因。

10. 讲一下对图的理解（暂时无法回答）

11. 判断单链表成环与否？

https://blog.csdn.net/njr465167967/article/details/52634352

```java
//内部静态类定义结点类
static class Node{
        int val;
        Node next;
        public Node(int val){
                this.val = val;
        }
}
```

```java
//判断单链表是否有环的方法
public static boolean hasLoop(Node head){
        Node p1 = head;    //定义一个引用指向头结点
        Node p2 = head.next;    //定义另一个引用指向头结点的下一个结点

        /**
         * 因为引用p2要比p1走的快，所以要用它作为循环的结束标志，为了防止当链表中个数为
         * 偶数时出现p2.next=null空指针异常，这时可以在循环中进行一下判断，如果这种情况
         * 出现一定是无环的。
         */
        while(p2 != null && p2.next != null){
                p1 = p1.next;
                p2 = p2.next.next;
                if(p2 == null)
                        return false;
                //为了防止p2.val出现空指针异常，需要对p2进行判断
                int val1 = p1.val;
                int val2 = p2.val;
                if(val1 == val2)
                        return true;
        }
        return false;
}
```

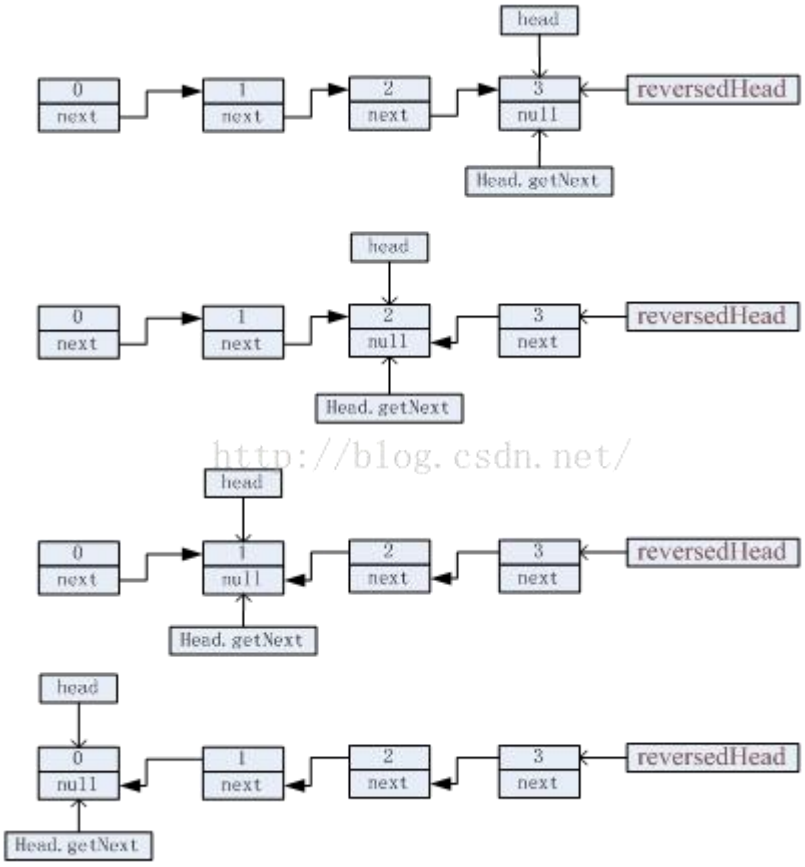
利用 P2 比 P1 走得快的原理，以此形成差距，对比其内部特定数值，如果如果相同，则表示闭环

12. 链表翻转（即：翻转一个单项链表）

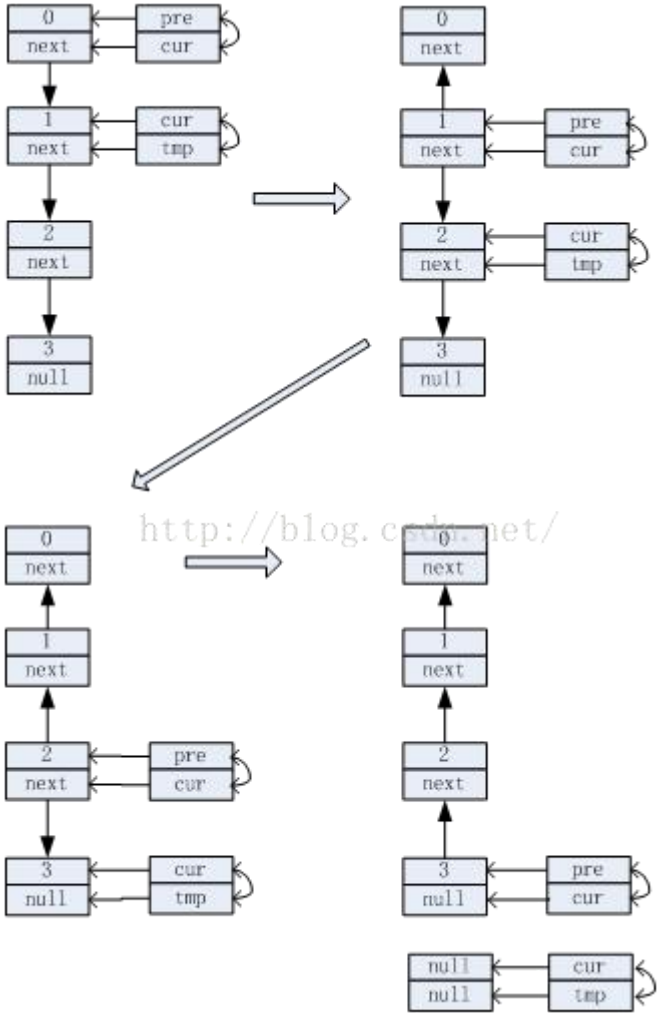
https://blog.csdn.net/guyuealian/article/details/51119499

递归反转法：同上第 8 问

类似这样



遍历反转法：



```java
/**
 * 遍历，将当前节点的下一个节点缓存后更改当前节点指针
 */
public static Node reverse2(Node head) {
        if (head == null)
                return head;
        Node pre = head;// 上一结点
        Node cur = head.getNext();// 当前结点
        Node tmp;// 临时结点，用于保存当前结点的指针域（即下一结点）
        while (cur != null) {// 当前结点为null，说明位于尾结点
                tmp = cur.getNext();
                cur.setNext(pre);// 反转指针域的指向

                // 指针往下移动
                pre = cur;
                cur = tmp;
        }
        // 最后将原链表的头节点的指针域置为null，还回新链表的头结点，即原链表的尾结
        head.setNext(null);

        return pre;
}
```

13. 合并多个单有序链表（假设都是递增的）

大体思路:使用递归

步骤:

1. 判断 L1,L2 是否为空
2. 创建一个头指针
3. 判断当前 L1,L2 指向的节点值的大小.根据结果,让头指针指向小节点,并让这个节点往下走一步,作为递归函数调用的参数放入,返回的就是新的两个值的比较结果,则新的比较结果放入头结点的下一个节点.

4. 返回头结点

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* mergeTwoLists(struct ListNode* l1, struct ListNode* l2)
{
    //2.创建一个头指针
    struct ListNode *head = NULL;
    //1：判断L1、L2是否为空
    if(l1 == NULL)
    {
        return l2;
    }
    if(l2 == NULL)
    {
        return l1;
    }

//3.根据计算判断结果及将指针后移
    if(l1->val <= l2->val)
    {
        head = l1;
        head->next = mergeTwoLists(l1->next,l2);
    }
    else
    {
        head = l2;
        head->next = mergeTwoLists(l1,l2->next);
    }
//4.返回头结点
    return head;
}
```