# 1.Arraylist和linkedlist底层实现的区别

答：

ArrayList和LinkedList都实现了List接口，他们有以下的不同点：   
ArrayList是基于索引的数据接口，它的底层是数组。它可以以O(1)时间复杂度对元素进行随机访问。与此对应，LinkedList是以元素列表的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是O(n)。   
相对于ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。   
LinkedList比ArrayList更占内存，因为LinkedList为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

2.数组和链表存储上谁大点怎么量化大多少的，我想应该跟指针域内存位数有关但没说.

3.hashmap底层实现数据结构。

4.评判一个代码段的时间复杂度和空间复杂度的依据是怎样的。

5. 还有设计模式，单例模式，工厂模式你了解吗？代码看过吗？我说懒汉模式饿汉模式，他就问为什么他要区分这些呢？

# 进程和线程

**（1）进程**

进程是程序的一次执行过程，是一个动态概念，是程序在执行过程中分配和管理资源的基本单位，每一个进程都有一个自己的地址空间，至少有 5 种基本状态，它们是：初始态，执行态，等待状态，就绪状态，终止状态。

**（2）线程**

 线程是CPU调度和分派的基本单位，它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

**（3）联系**

 线程是进程的一部分，一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。

**（4）区别：理解它们的差别，我从资源使用的角度出发。（所谓的资源就是计算机里的中央处理器，内存，文件，网络等等）**

根本区别：进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位

在开销方面：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。在java中每个线程有自己的**程序计数器**、**虚拟机栈**和**本地方法栈**

所处环境：在操作系统中能同时运行多个进程（程序）；而在同一个进程（程序）中有多个线程同时执行（通过CPU调度，在每个时间片中只有一个线程执行）

内存分配方面：系统在运行的时候会为每个进程分配不同的内存空间；而对线程而言，除了CPU外，系统不会为线程分配内存（线程所使用的资源来自其所属进程的资源），线程组之间只能共享资源。

包含关系：没有线程的进程可以看做是单线程的，如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

Thread.sleep(0)是干什么用的呢？它的作用就是：**强制操作系统触发一次CPU计算优先级并分配时间片的动作**

# 非递归中序遍历二叉树

void middleOrder(TreeNode pRoot) {

if (pRoot == null)

return;

Stack<TreeNode> stack = new Stack<>();

TreeNode node = pRoot;

do {

if (node != null) {

stack.push(node);

node = node.left;

} else {

node = stack.pop();

System.out.println(node);//与前序遍历的区别在于获取值的位置

node = node.right;

}

} while (node != null || !stack.isEmpty());

}

# 非递归前序遍历二叉树

void frontOrder(TreeNode pRoot) {

if (pRoot == null)

return;

Stack<TreeNode> stack = new Stack<>();

TreeNode node = pRoot;

do {

if (node != null)

System.out.println(node);//与中序遍历的区别在于获取值的位置

stack.push(node);

node = node.left;

} else {

node = stack.pop();

node = node.right;

}

} while (node != null || !stack.isEmpty());

}

# 非递归后序遍历二叉树（自己写的可能有问题？）

void behindOrder(TreeNode pRoot) {

if (pRoot == null)

return;

Stack<TreeNode> stack = new Stack<>();

TreeNode pCur = pRoot;

TreeNode pLast = null;

do {

//先找到到该子树的最左节点

if (pCur != null) {

stack.push(pCur);

pCur = pCur.left;

}

else {

pCur = stack.pop();

//只有当根节点没有右节点或者其右节点已经被访问过时，才能去访问该根节点

if (pCur.right == null || pCur.right == pLast) {

System.out.println();//在这里访问节点哟

pLast = pCur;

//该根节点已经访问完了，置当前指针为空，使其去访问上层节点

pCur = null;

} else {

//根节点的右节点还未被访问过，先将根节点入栈，转而开始处理右子树

stack.push(pCur);

pCur = pCur.right;

}

}

}while(pCur != null || !stack.isEmpty());

}

# 选择排序

//升序

public static void select(int[] a) {

for (int i = 0; i < a.length - 1; i++) {

int min = i;

for (int j = i + 1; j < a.length; j++) {

if (a[min] > a[j])

min = j;

}

if (min != i) {

int temp = a[min];

a[min] = a[i];

a[i] = temp;

}

}

}

# 冒泡排序

public static void pop(int[] a) {

for (int i = 0; i < a.length - 1; i++) {

boolean flag = false;

for (int j = 0; j < a.length - 1 -i; j++) {

if (a[j] > a[j+1]) {

int temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

flag = true;

}

if (!flag)

break;

}

}

}

# 插入排序

public static void insert(int[] a) {

for (int i = 1; i < a.length; i ++) {

int j = i;

while (j > 0 && a[j] > a[j-1]) {

int temp = a[j];

a[j] = a[j-1];

a[j-1] = temp;

j--;

}

}

}

# 堆排序

//从下向上构建堆的时间复杂度为O（n）。因为从n/2-1的位置开始构建，最多进行两次比较和一次互换。

//在有堆的基础上调整堆的时间复杂度为O（logn），因为完全二叉树的堆的深度大致为logn。

public static int[] heapSort(int[] arr) {

//1.将无序数构建为大顶堆

for(int i = (arr.length)/2-1; i >= 0; i--)

//1.从最后一个非叶子结点从下至上，从右至左调整结构

adjustHeap(arr, i, arr.length);

for (int i = arr.length - 1; i > 0; i--) {

//2.交换堆顶元素与末尾元素+从上往下调整堆结构

arr[0] = arr[0] + arr[i];

arr[i] = arr[0] - arr[i];

arr[0] = arr[0] - arr[i];

adjustHeap(arr, 0, i);

}

return arr;

}

//调整大顶堆

public static void adjustHeap(int []arr,int root,int length) {

int temp = arr[root];//根节点值

for(int i = root\*2+1; i < length; i = i\*2+1) {

if (i+1<length && arr[i] < arr[i+1])//如果左子结点小于右子结点，i指向右子结点

i++;

if (arr[i] > temp) {

arr[root] = arr[i];//将根节点的值赋值为子节点的最大值

root = i;//该子节点变成了下次循环的根节点，其值应该是temp，只是没有赋值，因为有可能下次还要交换，所以先不赋值

}

else break;

}

arr[root] = temp;//将temp值也就是最先前的根节点值放到最终的位置

}

# 归并排序

<https://www.cnblogs.com/chengxiao/p/6194356.html>

public static void sort(int[] arr) {

int[] temp = new int[arr.length];

sort(arr,0, arr.length-1, temp);//在排序前，先建好一个长度等于原数组长度的临时数组，避免递归中频繁开辟空间

}

public static void sort(int[] arr, int begin, int end, int[] temp) {

if(begin < end) {

int mid = (begin + end)/2;

sort(arr, begin, mid, temp);//左子序列并归排序

sort(arr, mid+1, end, temp);//右子序列并归排序

merge(arr, begin, mid, end, temp);//将两个有序子数组合并操作

}

}

private static void merge(int[] arr, int begin, int mid, int end, int[] temp) {

int left = begin;//左边序列起始位置

int right = mid +1;//右边序列起始位置

int t = 0;//临时数组的起始位置

while (left <= mid && right <= end){

if (arr[left] < arr[right]) {

temp[t++] = arr[left++]; //把小的值放在前面，所以是升序

} else {

temp[t++] = arr[right++];

}

}

//没放完的子序列继续放

while (left <= mid)

temp[t++] = arr[left++];

while (right <= end)

temp[t++] = arr[right++];

//将temp中的元素全部拷贝到原数组中

t = 0;

while (begin <= end)

arr[begin++] = temp[t++];

}

# 快速排序

public static int[] quickSort(int[] arr) {

quickSort(arr, 0, arr.length-1);

return arr;

}

//前后指针法

public static void quickSort(int[] arr, int begin, int end) {

if (begin < end) {

int left = begin;

int right = begin + 1;

while (right <= end) {

if (arr[right] < arr[begin]) {

left++;

if(left != right)

swap(arr, left, right);

}

right++;

}

swap(arr, left, begin);

quickSort(arr,begin,left-1);

quickSort(arr,left+1,end);

}

}

private static void swap(int[] arr, int a, int b) {

int temp = arr[a];

arr[a] = arr[b];

arr[b] = temp;

}

//挖坑法

public static void quickSort(int[] arr, int low, int high) {

if(low < high) {

int l = low;

int h = high;

int base = arr[low];

while (l < h) {

while(l < h && arr[h] >= base)

h--;

arr[l] = arr[h];

while(l < h && arr[l] <= base)

l++;

arr[h] = arr[l];

}

arr[l] = base;

quickSort(arr, low, l-1);

quickSort(arr, l+1, high);

}

}

# 各种排序的稳定性，时间复杂度和空间复杂度总结



快速排序的空间复杂度应该为O(logn)？

# URI和URL的区别



URI是统一资源标识符，在某种规则下能够唯一标识网络中的某个资源。URL是统一资源定位符，是URI的一个子集，是一个具体的URI，他不仅能够唯一标识资源，还提供了定位该资源的信息，能够通过URL去访问到这个资源。URI 是一种语义上的抽象概念，可以是绝对的，也可以是相对的，而URL则必须提供足够的信息来定位，是绝对的。

# 在浏览器中输入URL并回车后都发生了什么

**url的基本组成部分**



www是服务器名，还可以是mail。aspxfans.com是域名，服务器名+域名=主机名。

* DNS 解析:将域名解析成 IP 地址
* TCP 连接：TCP 三次握手
* 发送 HTTP 请求
* 服务器处理请求并返回 HTTP 报文
* 浏览器解析渲染页面
* 断开连接：TCP 四次挥手

参考：<https://github.com/ljianshu/Blog/issues/24>

<https://www.cnblogs.com/tisikcci/p/5866753.html>