


《京程一灯》精英班第八周笔试题 姓名：

请先盖住答案，在右侧空白处作答已加深印象。

- 1. 通常我们从哪几个方面来考量一个算法？（5分）


答：

 这是基础问题，尽管面试时不会直接问你，但是你仍然要知道！！尤其是这些名词都是什么意思！！本题考点分为如下：

时间复杂度、空间复杂度、正确性、可读性、健壮性

- 2. 请简述算法都有哪些特征？（5分）


答：

 同样是基础性问题，自己在做算法时按照这些特征进行检查。本题考点分为如下：

- 1、有穷性，算法必须在有限的步骤之后完成。
- 2、确切性，每一步必须有确切意义；
- 3、具备0个或多个输入项；
- 4、具有一个或多个输出；
- 5、具有可行性（也叫有效性）

- 3. 请简述栈和队列这两种数据结构的特征。（10分）

答：


 本题考点分为如下：

栈：通常使用链表或数组来实现，只允许从栈顶（称为top）位置进行操作，具有入栈和出栈两种主要操作，遵循后入先出规则（FILO）；

队列：通常用链表或者数组来实现。队列只允许在后端（称为rear）进行插入操作，在前端（称为front）进行删除操作。队列的操作方式和栈类似，唯一的区别在于队列只允许新数据在后端进行添加。遵循先入先出规则（FIFO）

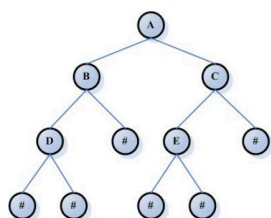
- 4.请列举线性表的基本操作都有哪些 (5分)

答：

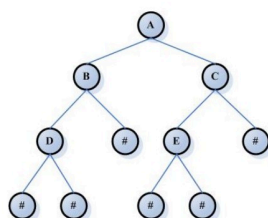
 本题考点分为如下：

初始化（创建）、插入元素、查找元素、删除元素、修改指定位置的元素、得到指定位置的元素、清空表等

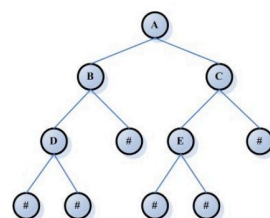
- 5.请在下图中画出二叉树先序遍历、中序遍历和后序遍历的路径和序号(10分)



先序遍历



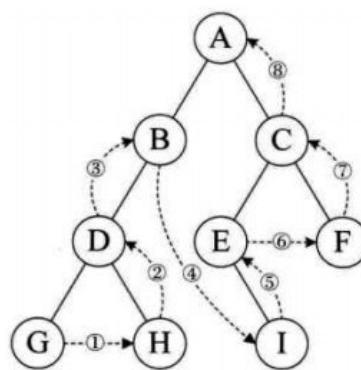
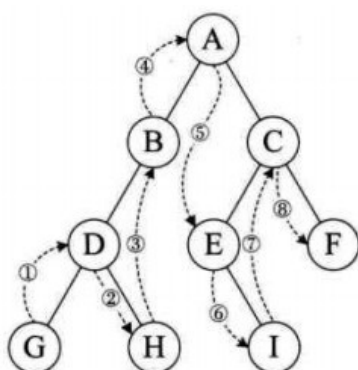
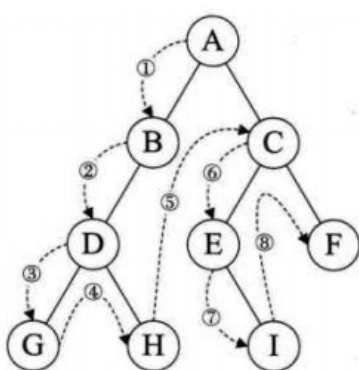
中序遍历



后序遍历

答：

 这些遍历顺序很容易问道！本题考点分为如下：



- 6. 请简述散列表的特点和用途 (10分)

答：

 这些原理性的东西要能够用自己的话说出来。本题考点分为如下：


散列表（也叫哈希表）是一种查找算法，与链表、树等算法不同的是，散列表算法在查找时不需要进行一系列和关键字（关键字是数据元素中某个数据项的值，用以标识一个数据元素）的比较操作。

散列表算法希望能尽量做到不经过任何比较，通过一次存取就能得到所查找的数据元素，因而必须要在数据元素的存储位置和它的关键字（可用key表示）之间建立一个确定的对应关系，使每个关键字和散列表中一个唯一的存储位置相对应。因此在查找时，只要根据这个对应关系找到给定关键字在散列表中的位置即可。这种对应关系被称为散列函数(可用 $h(key)$ 表示)。

根据设定的散列函数 $h(key)$ 和处理冲突的方法将一组关键字key映像到一个有限的连续的地址区间上，并以关键字在地址区间中的像作为数据元素在表中的存储位置，这种表便被称为散列表，这一映像过程称为散列，所得存储位置称为散列地址。

- 7. 请说出冒泡排序与选择排序的优缺点。(10分)

答：

 本题考点分为如下：


冒泡排序是稳定的排序，选择排序是非稳定排序。

冒泡排序需要开辟新的内存空间以提供交换操作，交换次数多，效率低。

选择排序相对于冒泡排序交换次数少，效率相对较高

- 8. 请写出相关程序最快查找出数组[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]中31出现的位置。(15分)

答：

 只看代码是看不会的，你需要亲自写一遍，然后再调试一遍，基本上就没问题了。本题考点分为如下：

//二分查找

```
function find(arr, item) {
    var low = 0; // 设定下标
```


```

var high = arr.length - 1; // 设定上标
while (high > low) {
    var mid = Math.floor((low + high) / 2); // 二分查找的关键
    if (arr[mid] > arr[item]) {
        high = mid;
    } else if (arr[mid] < arr[item]) {
        low = mid;
    }
    else {
        return mid;
    }
}
return -1;
}

```

- 9. 请写出单向链表和双向链表的添加操作和删除操作的主要实现代码。（15分）

答：

 如果答这道题的时候觉得脑容量不足，可以借助在纸上画图，纸和笔是大脑的最佳缓存。本题考点分为如下：

// 单向链表的添加

```

function insert(newElement, item) {
    var newNode = new Node(newElement); // 新建一个节点
    var current = this.find(item); // 找到item的位置
    newNode.next = current.next; // 将新节点的后继指向item的后继
    current.next = newNode; // 修改item节点的后继指向新节点
}

```

// 单向链表的删除

// 首先查找要删除元素的上一个节点

```

function findPrevious(item) {
    var currNode = this.head;
    while (!(currNode.next == null) && (currNode.next.element != item)) {
        currNode = currNode.next;
    }
    return currNode;
}

```

```

function remove(item) {
    var prevNode = this.findPrevious(item);
}

```

```

var currentNode=this.find(item);//查找到当前需要删除的节点

    if (!(prevNode.next == null)) {
        prevNode.next = prevNode.next.next;//待删除节点的前驱的后继指向后继指向
        原本待删除节点的后继
        currentNode.next=null;//为了防止内存泄漏
    }
}
=====
//双向链表
//插入节点 注意插入的链指向
function insert(newElement, item) {
    var newNode = new Node(newElement);
    var current = this.find(item);
    newNode.next = current.next;
    newNode.previous = current;
    current.next = newNode;
    if(newNode.next!=null){//判断是否为尾节点
        newNode.next.previous=newNode;//将item原本的后继的前驱
        指向新节点
    }
}

function remove(item) {
    var currNode = this.find(item);
    if (!(currNode.next == null)) {
        currNode.previous.next = currNode.next;//删除节点的前驱的 后继指向删除节点的后
        继
        currNode.next.previous = currNode.previous;//删除节点的后继的前驱指向删除节点
        的前驱

        currNode.next = null;//释放节点
        currNode.previous = null;
    } else{//考虑尾节点的情况
        currNode.previous.next = null; //尾节点的前驱的后继指向
        null
        currNode.previous = null;//释放尾节点
    }
}

```

- 10.请用代码实现一个二叉搜索树，并写出相关方法查找最小值。（15分）

答：

🍄 本题考点分为如下：

//定义节点

```
function Node(data,left,right) {
    this.data = data;
    this.left = left;
    this.right = right ;
}

function insert(data){
    var n = new Node(data,null,null);//定义一个新节点
    if (this.root == null) {//判断根节点是否为空
        this.root = n;
    }else{
        var current = this.root;
        var parent;
        while(true){
            parent = current;
            if (data < current.data) {//比当前小就放在左数
                current = current.left;
                if (current == null) {//直到左边没有数，将待添加
                    的值放进去
                    parent.left = n;
                    break;
                }
            }else{
                current = current.right;
                if (current == null) {
                    parent.right = n;
                    break;
                }
            }
        }
    }
}

//最小值
function getSmalllest(root){//一直往左子树上去找，找到没有左节点即找到了最小值
    var current = this.root || root;
    while(!(current.left==null)){
        current = current.left;
    }
    return current;
}
```