数据结构与算法

1. 数组和链表

数组：以一片连续的内存空间存放数据，内存大小在声明数组的时候已经确认。
链表：则是以非连续的内存空间存放数据，通过指针将每一个内存单元连接在一起。不需要提前分配好内存大小。

2. 队列、栈、出栈、入栈

队列：前进先出的数据结构，队尾插入，队头访问或删除。(PriorityQueue 优先队列)
栈：先进后出的数据结构，查询、插入、删除操作都在栈顶中进行。

出栈：相当于删除栈中元素，在栈顶执行操作。
入栈：相当于插入一个元素，同样在栈顶操作。

3. 链表的删除、插入、反向。

删除：找到链表节点位置，将前一元素的 Next 指针指向节点的下一节点位置 Next。
插入：找到需要插入的链表位置，将前一节点的 Next 指针指向插入节点，再将插入节点的 Next 指针指向后一节点。
反向：改变节点的指向位置，将头节点指向 null，后面的节点的 next 节点反向指回来。

```
//单链表反转 主要是逐一改变两个节点间的链接关系来完成
static Node<String> revList(Node<String> head) {

    if (head == null) {
        return null;
    }

    Node<String> nodeResult = null;
    Node<String> nodePre = null;
    Node<String> current = head;

    while (current != null) {
        Node<String> nodeNext = current.next;
        if (nodeNext == null) {
            nodeResult = current;
        }
        current.next = nodePre;
        nodePre = current;
        current = nodeNext;
    }
  return nodeResult;
}
```

4. 字符串操作

朴树匹配算法（暴力匹配）
KMP 算法：Next[ ]数组，失配后回溯的位置。

5. Hash 表（散列表）的 hash 函数，冲突解决办法有哪些？

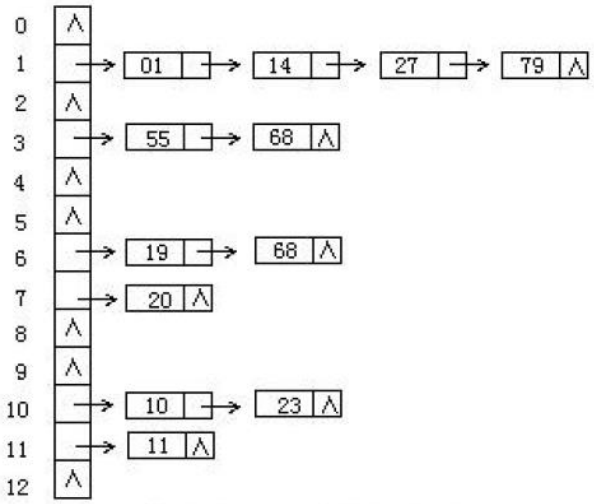
解决冲突：

1 开放定址法

由关键码得到的散列地址产生冲突，就去寻找下一个空的散列地址。（hash 函数）

2 链地址法

将由关键码得到的散列地址产生冲突后，将数据存储在同一个单链表中。



链地址法处理冲突时的哈希表
（同一链表中关键字有序）

3.再哈希法

将由关键码得到的散列地址产生冲突后，通过第二个 hash、第三个 hash 函数。。计算地址，直至无冲突。

6. 各种排序：冒泡、选择、插入、归并、希尔、快排、堆排。平均复杂度，空间发杂度、稳定？

各种常用排序算法						
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	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	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	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	不稳定
归并排序		$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
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						

7. **快排的 partion 函数 和归并的 Merge 函数。**

```
//快排 Partion, 返回分治中间位置数
int quick_sort(int s[], int l, int r)
```

```
//归并 Merge
void merge (int a[], int first, int mid, int last, int temp[])
```

8. **冒泡与快排的改进。**

```
int Partition(int *arr,int left,int right)   //快速排序实现升序排序数据
{
    int key=arr[left];
    while(left < right) {
        while(left < right && arr[right] >= key) {  //从元素右面查找第一个比轴枢小的元素
            --right;
        }
        arr[left]=arr[right];
        while(left < right && arr[left] <= key) {   //从元素左面查找第一个比轴枢大的元素
            ++left;
        }
        arr[right]=arr[left];
    }
    return left;
}
void quick_sort(int arr[],int left,int right) {
    int key=0;
    if(left < right) {
        key=Partition(arr,left,right);  //轴枢 key
        quick_sort(arr,left,key-1);  //位于轴枢左面的数字进行排序
        quick_sort(arr,key+1,right);  //位于轴枢右面的数字进行排序
    }
}
```

9. 二分查找

```
int binSearch(int[] arr, int n, int key){
    int low = 0;
    int high = n – 1;
    int mid;
    while(low < high){
        mid = (low + high) / 2;
        if(key == arr[mid]){
            return mid;
        }else if(key < mid){
            high = mid – 1;
        }else{
            low = mid + 1;
        }
    }
}
```

10. 二叉树 和 B+树 + AVL 树和红黑树、哈夫曼树。

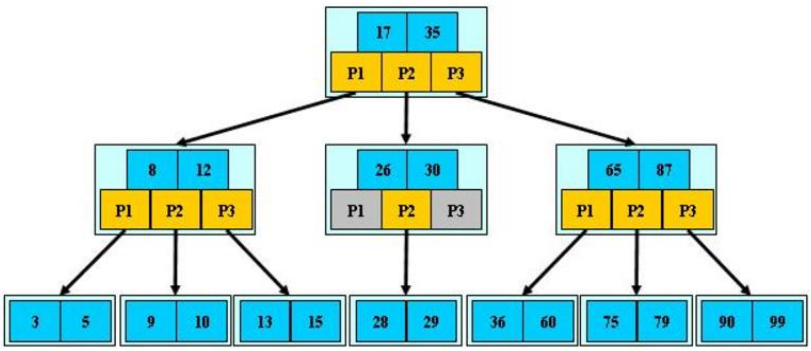
BST 树：即二叉搜索树，非叶子结点的左指针指向小于其关键字的子树，右指针指向大于其关键字的子树。

AVL 树：平衡二叉搜索树，在 BST 树基础上，左右子树深度之差的绝对值不超过 1。左右子树仍然为平衡二叉树。

B/B-/+树：平衡多路搜索树，常见应用于文件系统和数据库管理系统。 定义：一棵 m 阶二叉树。节点非空时，节点至少有两棵子树。树中每个节点最多含有 m 棵子树。

B- 树：

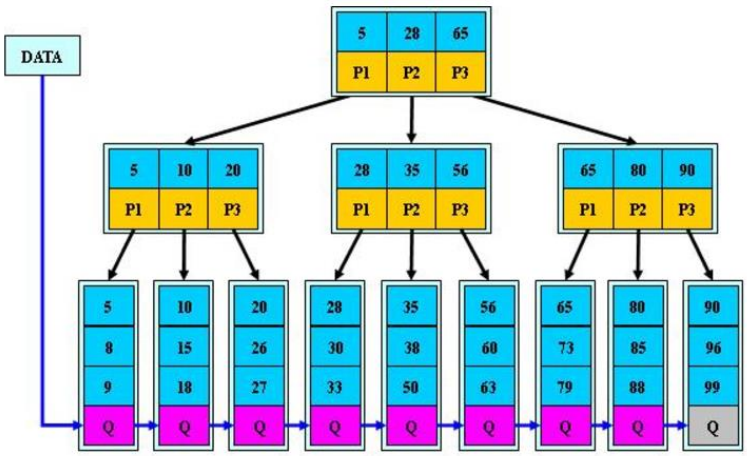
如：（**M=3**）



B-树的搜索，从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点；

B+树：

如：（M=3）



B+的搜索与B-树也基本相同，区别是B+树只有达到叶子结点才命中（B-树可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找；

B 树：二叉树，每个结点只存储一个关键字，等于则命中，小于走左结点，大于走右结点；

B-树：多路搜索树，每个结点存储 M/2 到 M 个关键字，非叶子结点存储指向关键字范围的子结点；

所有关键字在整颗树中出现，且只出现一次，非叶子结点可以命中；

B+树：在 B-树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中；

红黑树：一个更高效的检索二叉树,常用来实现关联数组。JDK 中的 TreeMap 底层就是红黑树的实现。特征：

1)每个节点要么是红色，要么是黑色

2)根节点永远是黑色的

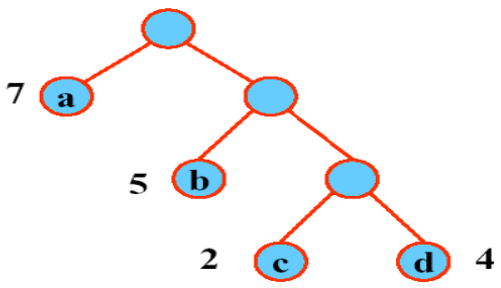
3)所有的叶节点都是空节点(null)，并且是黑色的，被称为黑哨兵

4)每个红色节点的两个子节点都是黑色

5)从任一节点到其子树中每个叶子节点的路径都包含相同数量的黑色节点

通过对任何一条从根到叶子的简单路径上各个节点的颜色进行约束，确保没有一条路径会比其他路径长 2 倍，因而是近似平衡的。所以相对于严格要求平衡的 AVL 树来说，它的旋转保持平衡次数较少。用于搜索时，插入删除次数多的情况下我们就用红黑树来取代 AVL。

哈夫曼树：也称最优二叉树，是一类带权路径最短的二叉树。



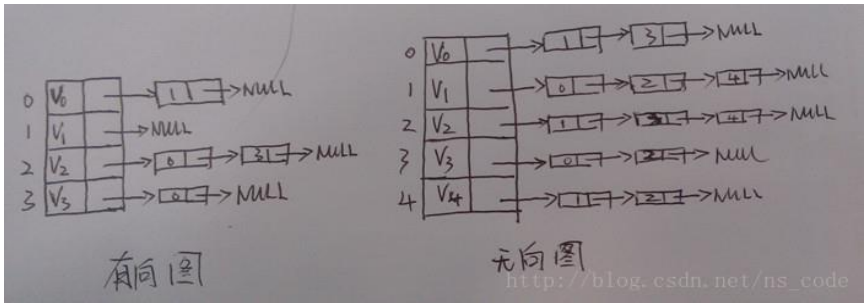
WPL=7*1+5*2+2*3+4*3=35

11. 二叉树的前序(根左右)、中序(左根右)、后序遍历(左右根)。递归与非递归写法，层序遍历。

```
//前序(根左右)
void preOrder(BinNode<T> root){
    if(root != null){
        visit(root);
        preOrder(root.left);
        preOrder(root.right);
    }
}
//中序(左根右)
void preOrder(BinNode<T> root){
    if(root != null){
        preOrder(root.left);
        visit(root);
        preOrder(root.right);
    }
}
//后序遍历(左右根)
void preOrder(BinNode<T> root){
    if(root != null){
        preOrder(root.left);
        preOrder(root.right);
        visit(root);

    }
}
```

12. 图的 BFS、DFS。最小生成树 Prim 算法与最短路径 Dijkstra 算法。



13. KMP 算法，高效匹配字符串 Next[ ] 字符失配后的回溯。

14. 排列组合问题

15. 动态规划 dp、贪心、分治算法（了解）

16. 大数据处理 –10 条数据找出最大的 1000 个数。