1　根据历年题目整理出的题库

> Karry

1.1　一、单选题（10 * 2）

1.1.1　考点一：数据结构最基本的概念

- 数据结构在计算机中的表示称为数据的（存储结构）——下面的有可能考辨析选择题
    - DataStructure = (D, S)　S 是定义在D中的数据元素之间关系的有限集合。
    - 数据结构涵盖三个方面的内容1. 数据之间的逻辑关系 2. 数据在计算机中存储的物理结构 3. 对数据进行的操作 <理清这个说法>
    - 数据的逻辑结构与存储无关
- 线性表的相关知识点
    - 判断哪一个不是线性表（线性表中的元素类型一定要相同！）
    - 链式线性表插入结点和删除结点的操作

```
// 插入
newNode->next = pre -> next;
pre->next = newNode;
// 删除
targetNode = pre -> next;
pre -> next = targetNode-> next;
delete targetNode;
```

    - 性能的比较 哪一种线性表最好 —— 普顺 or 链式普通 or 链式循环 or 链式双向？ —— 最基本的特点
        - 线性表的顺序存储结构：可直接随机访问任一元素 所需空间与线性表长度不成正比
        - 循环链表：从表中任一结点出发均可找到其他结点。
        - 可能的问答题：单链表中由于只能结点指向只能向一个方向移动，且到达结尾的时候无法返回至开头，遂从当前结点出发并不一定能访问到任一结点；而双向链表则可以，因为由当前结点出发既可以向前访问，又可以向后访问。

- 栈和队列的相关知识点
  - 一定要明确栈和队列都是线性表，只不过受了些限制
  - 顺序栈下的栈共享技术：利用“两个栈的栈底位置不变，分别在一维数组的两端，栈顶位置动态变化” 直到两个栈顶相遇时才发生溢出。
  - 循环队列是一种**顺序存储结构**，存在溢出的可能。单纯的顺序队列具有假溢现象（明明还有空间但是条件上看不出来）

    循环队列的相关操作 // 如果出得比较偏得话 还是可能出到的

    - 队列初始化：front = rear = 0;
    - 元素出队：队头front自加1 front = (front+1) % maxSize;
    - 元素人队：队尾rear自加1 rear = (rear+1) % maxSize;
    - 返回队列长度：return (rear - front + maxSize) % maxSize;
    - 队满条件：(rear+1) % maxSize == front

    // 如果用了上面的方法 衍生出来的问题就是 rear == front 的条件下不能判断是空 还是 满 two ways

    - way 1 另设置一个标志来区别队列是空还是满

    引入一个 maxSize 来确定到底是满还是空只有空的时候才会有rear = front 满的时候用 length = maxSize - 1

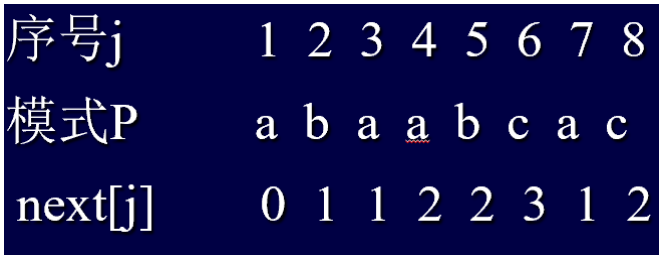
    - way 2 少用一个元素空间，约定队头在队尾指针的下一个位置时作为队满的标志。

    也就是说不再是以rear == front 来判断队满 而是 front == rear - 1 来判断

- **字符串**的相关知识点：—— 做了这么多卷子基本上没有看到过和串相关的考点
  - 唯一一个可能考察的概念：如果用链式结构存储串**密度极低**因此一个结点里尽可能多存储几个串
  - 子串的相关知识 —— 这个还是比较重要的
    - 会计算子串的个数：设S为一个长度为 n的字符串，其中的字符各不相同，则 S中的互异的非平凡子串（非空且不同于S本身）的个数为

- KMP 算法 —— 如果题目比较难的话 选择题也是可能出到的

  考察方法 —— 给一个字符串让你求解它的 next 数组 而手工求解太简单了 谨记 next[1] = 0



- 广义表的相关知识点 —— 这个常考基本概念，并且考到了会让很多人恼火
  - 表头和表尾的辨析 写出一个表头等于表尾的表 (())

    - n = 0 的广义表为空表， 没有表头表尾的概念 # 空表无表头表尾
    - n > 0的时候表的第一个表元素称为广义表的表头 除此以外其它表元素组成的表称为广义表的表尾 # 表尾肯定是一个表

  - 长度和深度

    广义表的长度：就是原始表有多少个大块元素（不论是原子还是表）

    例如

    - C=(a,(b,c,d))——列表C的长度为2
    - B=(e)——广义表B只有一个原子，长度为1。

    广义表的深度：最简单的方法就是看括号层数

    depth(GL) = 0 if GL 为原子元素
    = 1 if GL 为空表
    = 1 + Max(depth(ai)(1 <= i <= n)) 其他情况

    例如

    - 2 深度为0 (括号层数为 0)
    - () 深度为1 (括号层数为 1)
    - (1, (2, 3))深度为 2(1 + 1) (括号层数为2)

  - （8）下面关于广义表的叙述中，不正确的是（ ）。↵
    A）广义表可以是一个多层次的结构 B）广义表至少有一个元素↵
    C）广义表可以被其他广义表所共享 D）广义表可以是一个递归表↵

- 以下数据结构中哪一个是线性结构? —— 只有完全二叉树才能用线性结构表示

> A）有向图 B）栈 C）二叉树 D）B树

- （10）下列关于数据结构的叙述中，正确的是（ D ）。↵
  A）数组是不同类型值的集合 ↵
  B）递归算法的程序结构比迭代算法的程序结构更为精炼↵
  C）树是一种线性结构↵
  D）用一维数组存储一棵完全二叉树是有效的存储方法↵

1.1.2 考点二：有关二叉树的基本内容

- 一棵深度为5的完全二叉树的结点树至少为 **16**
  - 深度为k的完全二叉树最少有多少个结点 2^(k - 1) - 1 + 1

涉及到的相关概念：

- 深度（高度）（层次）：谨记根结点的深度为 1 根的高度为 1 根的层次为 1 <树不是从0开始的>
- 二叉树的种类
  - 正则二叉树 <度数要么满要么为0>
  - 完全二叉树 一个个结点往后追加的二叉树 <满二叉树一定完全><完全二叉树不一定满>
  - 满二叉树 每层结点都是<满的> 深度为 k 且有 2^k - 1个结点

- 在二叉树结点的先、中、后序列中所有叶子结点的先后顺序 **完全相同**
- 用 n 个键值构建一棵二叉排序树，最低高度为 **log2n下取整再加1**

> (这个貌似考察了二叉排序树， 但是本质还是完全二叉树的高度)
> 本质就是两个不等式找范围

  - 变式：含n个结点的完全三叉树的高度是多少？
- 完全二叉树找寻父子结点：

从根结点开始编号，根结点的编号为 i = 1 最大编号为 n

双亲结点

- 当 i > 1 时 其双亲结点的序号为 (i/2) 下取整

左子节点

- 当 2i > n 时说明无左孩子，否则左孩子的序号为 2i

右子节点

- 当 2i + 1 > n 时说明无右孩子，否则右孩子的序号为 2i + 1

- 选择题中也可能会有二叉树的遍历，尤其是二叉树表示的算术表达式的读取，能从表达式 => 二叉树 也能从二叉树 => 表达式<不要乱加括号>

1.1.3 考点三：有关图的基本内容

- 对于具有n个顶点的强连通有向图，其边条数最小值为 **n** 这个题很难去论证但是**举例子太好举了**
- **关键路径的相关问题** —— 后面有相关介绍
    - 关键路径长度

1.1.4 考点四：查找方法

- 折半查找法又称为二分法查找法，这种方法要求待查找的列表必须是**按关键字大小有序排列**的顺序表。
- 在折半查找中第 i 次查找的记录个数最多为 **2 的 (i - 1) 次方**——二叉折半查找（排序）树叶子结点的个数

    > 要会计算二叉排序树的平均查找长度

- 要记住折半查找效果一般都是最优秀的，看到选择题让选择复杂度最小的一般都选他
- 平衡二叉树
    - 一个深度为k的平衡二叉树，其每个非终端结点的平衡因子均为0，则该树有 **2的k次方 - 1** 个结点（**满完全二叉树**）
    - 含12个结点的平衡二叉树的最大深度（设根结点层次数位1）是多少
- 一些特殊的平均查找长度还是要记忆一下
    - 顺序查找法：表中元素可以任意存放；
    - 折半查找法：表中元素必须以关键字的大小递增或递减的次序存放；
    - 分块查找法：表中元素每块内的元素可任意存放，但块与块之间必须以关键字的大小递增（或递减）存放，即前一块内所有元素的关键字都不能大于（或小）后一块内任何元素的关键字。

- 顺序查找法：查找成功的平均查找长度为$\frac{n+1}{2}$；
- 折半查找法：查找成功的平均查找长度为 log2(n+1)-1；
- 分块查找法：若用顺序查找确定所在的块，平均查找长度为$\frac{1}{2}(\frac{n}{s}+s)+1$；若用折半确定所在块，平均查找长度为$\log_2(\frac{n}{s}+1)+\frac{s}{2}$。

1.1.5 考点五：排序方法

- 最基本的考点就是下面一张图里的内容——200%会考

> 考前一定要再过一遍这张表里面的所有排序方法

表 9.1 各种排序方法性能

排序分类	排序名称	时间复杂度		辅助空间	稳定性
		平均时间	最坏时间		
简单排序	直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
	起泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
	简单选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
高级排序	快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
	归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
其他排序	Shell 排序	由增量序列确定		$O(1)$	不稳定
	基数排序	$O(d(2n+r))$	$O(d(2n+r))$	$O((n+r)n)$	稳定

从表中可以得到如下的结论。

- 下列算法中，操作时间不随文件的初始状态变化的排序算法是 基数排序

（2）有 n 个记录的文件，如关键字位数为 d，基数为 r，则基数排序共要进行（ ）遍分配与收集。
A）n B）d C）r D）n-d

- 堆排序——堆的构建
  - 设有关键字序列('q', 'g', 'm', 'z', 'a', 'n', 'p', 'x', 'h')，下面哪一个序列是从上述序列出发建堆的结果?

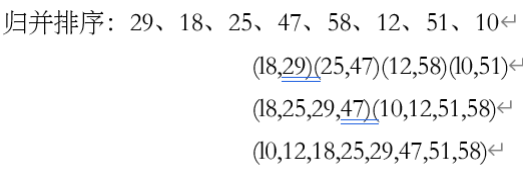
> ('a', 'g', 'm', 'h', 'q', 'n', 'p', 'x', 'z')

- 猜排序方法的选择题 这种题比较难，要求掌握各种排序算法的基本操作方式

（5）若数据元素序列 11，12，13，7，8，9，23，4，5 是采用下列排序方法之一得到的第二趟排序后的结果，则该排序算法只能是（ ）。
A）起泡排序 B）插入排序 C）选择排序 D）二路归并排序

  - 基本插入排序几次后 => 在数列队头形成从小到大的序列

- Shell 排序是特殊的插入排序 过程一定清清楚楚
- 简单选择排序几次后 => 在数列对头形成从小到大的序列 **并且是最小的几个数！！！**
- 堆排序是特殊的选择排序 => 一定要会调整堆
- 冒泡排序是最简单的交换排序 => 在数列尾形成从小到大的序列 并且是**最大的几个数！！！**
- 快速排序是特殊的交换排序 一定要会过程
- 归并排序 —— 和 shell 有几分相似注意过程怎么做的 || 以二路归并为例子



- 基数排序 几个关键词： 关键字个数为d 关键字可能的选择数为基数r 排序数目为n

1.2 中间有七道计算题

1.2.1 考点一：栈的出入以及所成序列

- 设栈最大的容量为3，入栈序列为1、2、3、4、5、6，则不可能的出栈序列为 **4、3、2、1、5、6**
- 五个元素的入栈次序为 A、B、C、D、E 则以元素C 第一个出栈 D 第二个出栈的次序有
  - C D B A E
  - C D B E A
  - C D E B A
- 新瓶装旧酒 输入次序为123PA 进栈，问哪些出栈序列可以构成一个变量名字

1.2.2 考点二： 三元组表

> 一定要注意都是从1开始数的

$$M_{6\times7}=\begin{bmatrix}0&12&9&0&0&0&0\\0&0&0&0&0&0&0\\-3&0&0&0&0&14&0\\0&0&24&0&0&0&0\\0&18&0&0&0&0&0\\15&0&0&-7&0&0&0\end{bmatrix}\quad N_{7\times6}=\begin{bmatrix}0&0&3&0&0&15\\12&0&0&0&18&0\\9&0&0&24&0&0\\0&0&0&0&0&-7\\0&0&0&0&0&0\\0&0&14&0&0&0\\0&0&0&0&0&0\end{bmatrix}$$

	1	2	12
1	1	3	9
2	3	1	−3
3	3	6	14
4	4	3	24
5	5	2	18
6	6	1	15
7	6	4	−7

(a) 矩阵 M 的三元组表

	1	3	3
1	1	6	15
2	2	1	12
3	2	5	18
4	3	1	9
5	3	4	24
6	4	6	−7
7	6	3	14
8			

(b) 矩阵 N 的三元组表

> 两大转置算法会不会考察呢？ —— 有时间还是要关注一下 自己要动手去写 才能体会到哪个点会卡壳

1.2.3 考点三：散列表法 或者 再扫描探测法解决 Hash 冲突

> 千万不要忘了计算平均查找长度

1.2.3.1 散列表法更简单些

1.2.3.2 再探测稍微难一点

- 散列表的装载因子 元素数目 / 总空间数

  > 例如 散列表装载因子为0.7 表示有7个数待排序到10个空间里

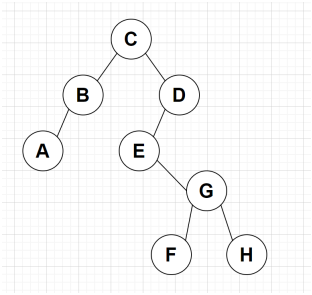
1.2.4 考点四：树和森林的简单操作

- 写出二叉树的三种遍历结果那就是送分题
- 森林和树的遍历只有 先序=先根 || 中序=后根遍历 => 这个基本不会考 但是考到要会迁移
- 前中后遍历叶子结点的相对顺序会发生改变吗

  > 二叉树任两个中叶结点必在某结点的左/右子树中，三种遍历方法对左右子树的遍历都是按左子树在前、右子树在后的顺序进行遍历的。所以在三种遍历序列中叶结点的相对次序是不会发生改变的。

- 画出原树还是要多多练习一下

- 某二叉树的中序序列为 ABCEFGHD 后序序列为 ABFHGEDC 画出此二叉树



- 分别画出具有三个结点的树和具有三个结点的二叉树所有的不同形态

  这个地方牵连到树的计数 n 个结点二叉树的个数：考前还是要记一下

$$b_n = \frac{1}{n+1}C_{2n}^{n}$$

具有 n 个结点的树的数目 == 具有 n-1 个结点的二叉树的数目相同

具有 n 个结点的森林数目 == 具有 n 个结点的二叉树的数目相同

- 树和二叉树的转换 树和森林的转换 **孩子兄弟表示法** —— 这个一定要会

> 双亲表示<数组中只表明双亲> 孩子双亲表示<数组中表明双亲 + 孩子>

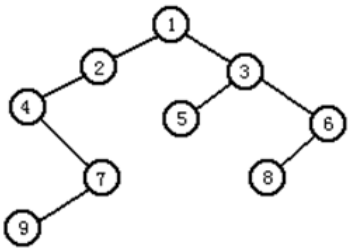
1.2.5 考点五：二叉树的相关计算

- 已知一棵度为m的树中有n1个度为1的结点，n2个度为2的结点……nm个度为m的结点，**该树中有多少个叶子结点**？

  一定要注意：这个地方结点的度不是图中的那个度而是说一个点有多少向下分支的边

> 只用两个方程 一个表示结点的个数 一个用度来表示边的个数 二叉树基本上所有的运算都是围绕着这两个方程做展开

- **二叉树的表示**



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	1	2	3	4		5	6		7					8				9

1.2.6 考点六：哈夫曼树

- 构造哈夫曼树是送分题 注意仔细 + 算带权路径和
- 一棵哈夫曼树中有n个叶子结点，那么该树有多少个结点？ <哈夫曼树是正则二叉树>

> 两个方程得到结果：2n - 1

1.2.7 考点七：图的表示方法 <最近几年题目都没有出现过 我估计出现的概率不大>

1.2.7.1 邻接表法

1.2.7.2 邻接矩阵法

1.2.8 考点八：图的遍历方法 —— 极容易考察

> 当出现说一个位置可能有两个走向的时候，选择值更小的那个

1.2.8.1 DFS 深度优先搜索

- 类比先序遍历

1.2.8.2 BFS 广度优先搜索

- 类比层次遍历

1.2.9 考点九：最小代价生成树算法 Prim<加点法> Kruskal<加边>

- 基本的知识点：
  - Prim 的时间复杂度 o(n^2)，此算法适用于边较多的稠密图
  - Kruskal 的时间复杂度 o(e log2e)，此算法适用于边较少的稀疏图

1.2.10 考点十： 二叉平衡树 —— 较难

二叉排序树

- 平均查找长度——如何查找要明白
- 二叉排序树的建立 + 添加——不同的顺序可能建立的不一样 但整体来说还是很简单的
- 二叉排序树的删除——难点
  - step1 找到待删除结点P的中序序列的直接前驱S 一定要找准直接前驱
  - step2 P的左子树直接上移，右子树变为S的右子树
  - step3 删除P

二叉平衡树——首先是二叉排序树 再加上一个平衡的性质

- 定义：结点的左子树深度与右子树深度之差不能超过正负1
  - 平均查找长度：O(log2n)
  - 平衡二叉树的构造——难点-需要来回旋转
  - 一定要找到最近的失衡点，对这一个点进行旋转，不要找错了

> 例子： w={23,24,27,80,28} 需要两次旋转

1.2.11 考点十一： 拓扑排序

1.2.11.1 给图 —— 很好做

1.2.11.2 只给点集 + 边集 一定要注意用栈存储和用队列存储二者的区别

已知一个图的顶点集 V 和边集 E 分别为：↵

V={1,2,3,4,5,6,7};↵

E={<2,1>,<3,2>,<3,6>,<4,3>,<4,5>,<4,6>,<5,1>,<5,7>,<6,1>,<6,2>,<6,5>};↵

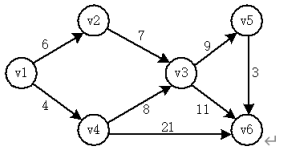
若采用邻接表存储结构，并且每个顶点邻接表中的边结点都是按照终点序号从小到大的次序链接的，试给出得到的拓扑排序的序列（入度为零的顶点可采用栈或队列来进行存储）。↵

用栈存储入度为零的顶点得到的拓扑排序为： 4 3 6 5 7 2 1↵

用队列存储入度为零的顶点得到的拓扑排序为： 4 3 6 2 5 1 7↵

1.2.12 考点十二： 关键路径

设有如下图所示的 AOE 网（其中 vi（i=l，2，…，6）表示事件，有向边上的权值表示活动的天数）。↵



（1）找出所有的关键路径。↵

（2）v3 事件的最早开始时间是多少。↵

这类题最主要就是让求四个内容 某一个事件<点>的最早发生时间 某一个事件<点>的最迟发生时间 某一个活动<边>的最早开始时间 某一个活动<边>的最迟开始时间

那么怎么解决这类问题呢？

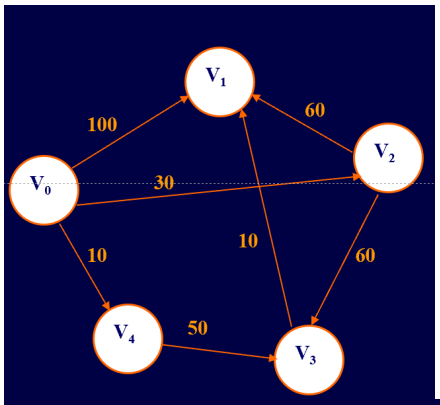
- step 1 根据拓扑排序画出表格
- step 2 求出每一个点最早发生时间 也就是从源头到该点的最长路径 + 最迟发生时间(从后往前来) => 这样关键路径就出来了

	v1	v2	v4	v3	v5	v6
ve	0	6	4	13	22	25
vl	0	6	4	13	22	25

	12	14	23	43	35	56	36	46
ee	0	0	6	4	13	22	13	4
el	0	0	6	5	13	22	14	4

1.2.13 考点十三：最短路径生成算法 迪杰斯拉算法 —— 考的可能性不大

这个考点写代码当然是很难，但是手推还是很简单的 参看下面一个例子吧



终点	i=1	i=2	i=3	i=4
v_1	dist[1]=100 path[1]=0	dist[1]=100 path[1]=0	dist[1]=90 path[1]=2	dist[1]=70 path[1]=3
v_2	dist[2]=30 path[2]=0	dist[2]=30 path[2]=0		
v_3	dist[3]=∞ path[3]=-1	dist[3]=60 path[3]=4	dist[3]=60 path[3]=4	
v_4	dist[4]=10 path[4]=0			
v_j	v_4	v_2	v_3	v_1
U	$\{v_0,v_4\}$	$\{v_0,v_2,v_4\}$	$\{v_0,v_2,v_3,v_4\}$	$\{v_0,v_1,v_2,v_3,v_4\}$

1.2.14 考点十四：写出典型排序算法的排序过程

给出一组关键字T=(12,2,16,30,8,28,4,10,20,6,18)。写出用下列算法从小到大排序时第一趟结束时的序列

- 希尔排序（第一趟排序的增量为6）(4, 2, 16, 6, 8, 28, 12, 10, 20, 30, 18)
- 快速排序（选第一个记录为枢轴）(6, 2, 10, 4, 8, 12, 28, 30, 20, 16, 18)

构建大顶堆和小顶堆 —— 很重要 一定要会调整

1.2.15 考点十五：判断对错 一般来说都是错的 关键要举出反例——近几年的题目看起来都还是比较简单的

- 只有一个顶点入度为0、其他顶点入度为1的有向图不一定是有向树（**可以有一个孤立的结点**）
- 假设把n个元素的序列（a1,a2,...an）满足条件ak<max{at|1≤t≤k}的元素ak称为“逆序元素”。若在一个无序序列中有一对元素ai>aj(i<j)，试问，当ai与aj相互交换后，该序列中逆序元素的个数一定不会增加，这句话对不对？如果对，请说明为什么？如果不对，请举一例说明。 so easy(1 0 3 2) 1 和 3 交换位置
- 带权图（权值非负，表示边连接的两顶点间的距离）的最短路径问题是找出从初始顶点到目标顶点之间的一条最短路径。假定从初始顶点到目标顶点之间存在路径，现有一种解决该问题的方法：

  1. 设最短路径初始时仅包含初始顶点，令当前顶点u为初始顶点；
  2. 选择离u最近且尚未在最短路径中的一个顶点v，加入到最短路径中，修改当前顶点u=v；
  3. 重复步骤2，直到u是目标顶点时为止。

  肯定错 除了迪杰斯拉算法其他的都不可以

1.3 九、手写代码题 —— 相关代码参见 附录：手写代码题

- 二叉树的层次遍历算法 —— 47开始 5 mins over correct!

```
template<class ElemType>
void BinaryTree<ElemType>::LevelOrderHelp(BinTreeNode<ElemType> *r,
void(*visit)(const ElemType&)){
    if(r != NULL){
        BinTreeNode<ElemType> *cur = r;
        LinkQueue<BinTreeNode<ElemType>* > q;

        q.InQueue(cur);
        while(!q.Empty()){
            q.OutQueue(cur);
            (*visit)(cur->data);

            if(r->leftChild != NULL){
                q.InQueue(r->leftChild);
            }
            if(r->rightChild != NULL){
                q.InQueue(r->rightChild);
```

```
            }
        }
    }
}
template<class ElemType>
void BinaryTree<ElemType>::LevelOrder(BinaryTree<ElemType> &bt,
void(*visit)(const ElemType&)){
    LevelOrderHelp(bt.GetRoot(), visit);
}
```

- 二叉树的前序非递归遍历

```
template <class ElemType>
void BinaryTree<ElemType>::PreOrderHelp(BinTreeNode<ElemType> *r,
void(*visit)(const ElemType&))const{
    if(r != NULL){
        BinTreeNode<ElemType> *cur = r;
        LinkStack<BinTreeNode<ElemType> *> s;

        while(cur!=NULL){
            (*visit)(r->data);
            s.Push(cur);

            if(r->leftChild != NULL){
                cur = cur->leftChild;
            }
            else if (!s.Empty()){
                while(!s.Empty()){
                    s.Pop(cur);
                    cur = cur->rightChild;
                    if(cur != NULL) break;
                }
            }
            else {
                cur = NULL;
            }
        }
```

```
    }
}

template <class ElemType>
void BinaryTree<ElemType>::PreOrder(BinaryTree<ElemType>&bt,
void(*visit)(const ElemType&))const{
    PreOrderHelp(bt.GetRoot(), visit);
}
```

- 中序遍历二叉树非递归算法

```
template <class ElemType>
BinTreeNode<ElemType>
BinaryTree<ElemType>::GetFarLeft(BinTreeNode<ElemType> *r,
LinkStack<BinTreeNode<ElemType>* >s){
    if(r == NULL) return NULL;
    else {
        BinTreeNode<ElemType> *cur = r;
        s.Push(cur);

        while(cur -> leftChild != NULL){
            cur = cur->leftChild;
            s.Push(cur);
        }
        return cur;
    }
}

template <class ElemType>
void BinaryTreeNode<ElemType>::InOrderHelp(BinTreeNode<ElemType>*r,
void(*visit)(const ElemType&)){
    if(r!=NULL){
        BinTreeNode<ElemType> *cur = r;
        LinkStack<BinTreeNode<ElemType>* > s;

        cur = GoFarLeft(cur, s);
        while(cur != NULL){
```

```
            (*visit)(cur -> data);
            if(cur -> rightChild != NULL){
                cur = GoFarLeft(cur, s);
            }
            else if(!s.Empty()){
                s.Pop(cur);
            }
            else cur = NULL;
        }
    }
}
```

- 复制二叉树非递归算法

```
// 基于中序遍历
template <class ElemType>
void CopyTree(BinaryTree<ElemType> &fromBT, BinaryTree<ElemType>
*&toBT){
    if (toBT != NULL) toBT = NULL;
    if(fromBT.Empty()) toBT = NULL;
    else{
        LinkQueue<BinTreeNode<ElemType> * > fromQ, toQ;
        BinTreeNode<ElemType>* fromPtr, toPtr, fromRoot, toRoot;
        fromRoot = fromBT.GetRoot();
        fromQ.InQueue(fromRoot);
        toRoot = fromRoot;
        toQ.InQueue(toRoot);

        while(!fromQ.Empty()){
            fromQ.OutQueue(fromPtr);
            toQ.OutQueue(toPtr);

            if(fromPtr->leftChild!=NULL){
                fromQ.InQueue(fromPtr->leftChild);
                toPtr->leftChild = new BinTreeNode<ElemType>
(fromPtr->leftChild->data);
                toQ.InQueue(toPtr->leftChild);
```

```
            }
            if(fromPtr->rightChild!=NULL){
                fromQ.InQueue(fromPtr->rightChild);
                toPtr->rightChild = new BinTreeNode<ElemType>
(fromPtr->rightChild->data);
                toQ.InQueue(toPtr->rightChild);
            }
        }

        toBt = new BinaryTree<ElemType>(toRoot)
    }
}
```

- 以左右子树的方法展示出二叉树

```
template<class ElemType>
void DisplayHelp(BinTreeNode<ElemType> *r, int level){
    if(r != NULL){
        DisPlayHelp(r->rightChild, level+1);
        cout << endl;
        for(int i = 0; i < level - 1; i++){
            cout << " ";
        }
        cout << r->data;
        DisPlayHelp(r->leftChild, level+1);
    }
}
template<class ElemType>
void Display(BinaryTree<ElemType> &bt){
    Display(bt.GetRoot(), 1);
}
```

- 求A B两个带头节点的链表的交集

```
template<class ElemType>
void AndLL(LinkList<ElemType>&la, LinkList<ElemType>&lb,
LinkList<ElemType>&lc){
```

```
    int aLength = la.Length();
    int bLength = lb.Length();
    int aPos = bPos = 1;
    ElemType aElem, bElem;

    while(aPos <= aLength && bPos <= bLength){
        la.GetItem(aPos, aElem);
        lb.GetItem(bPos, bElem);
        if(aElem < bElem){
            lc.Insert(lc.Length() + 1, aElem);
            aPos++;
        }
        else if(bElem < aElem){
            lc.Insert(lc.Length() +1, bElem);
            bPos++;
        }
        else{
            lc.Insert(lc.Length() + 1, aElem);
            bPos++;
            aPos++;
        }
    }
    while(aPos <= aLength()){
        la.GetItem(aPos, aElem);
        lc.Insert(lc.Length() +1, aElem);
        aPos++;
    }
    while(bPos <= bLength()){
        同上
    }
}
```

1.4 十、开放题

- 垃圾回收机制

垃圾回收技术所解决的有两个重要问题，一是如何识别当前内存中未被引用的内存；二是如何将失去管理的内存进行回收

way1:

使用引用计数法，对于一个内存块，除内存块本身外增加一个引用技术，每当这个内存块被外部的指针或内存块所引用的时候，引用计数+1；当引用他的指针释放了对他的引用的时候，则引用计数-1，同时检查引用计数的值，当引用计数为0时，就销毁该内存块并释放其所占用的内存。这种方法的优点是垃圾回收过程不需要打断进程运行，简单快捷，对资源的管理很有效。

way2:

使用标记清除法，两个阶段：标记阶段&清除阶段。在标记阶段，垃圾回收器扫描每个内存块，对被引用的内存块进行标记。在标记完成后，统一检测内存集中所有内存，将未被标记的内存块进行回收。