南 京 航 空 航 天 大 学

二〇二二 ～ 二〇二三 学年 第 1 学期 **《 数据结构 》考试试题**

考试日期：2023 年 月 日 试卷类型： 试卷代号：

班号 学号 姓名

题号	一	二	三	四	五	六	七	八	九	十	总分
得分											

本题分数	40
得 分	

一、解答题 (共 4 题，每题 10 分)

1. （10 分） 义表 A=((a,b),c,(d,(e)))，画出存储 A 两种结构，并说明两种结构的区 。

2. （10分）输入数据序列为(22,40,10,32,52,78,80,86,50,60),给出3 B-树的 建过程示意图， 建完成 分 画出删除80,78的示意图。

3. （10 分）解释哈希表的原理，并用哈希函数 H(x)=x%10 线性 法将（12,4,25,60,87,34,99,52）存入表长为 10 的哈希表中，画出执行过程。

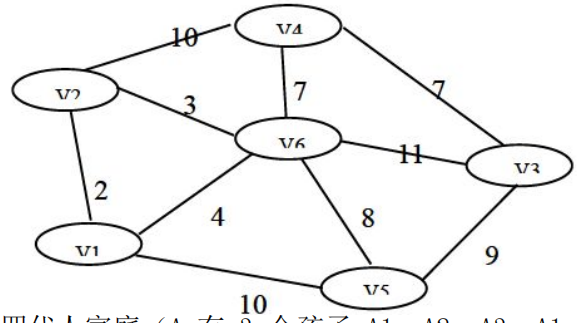
4. （10分）解释 并排序原理，对（18,499,5,22,81,24,1,35,7,46）进行 并排序，画出执行过程示意图。

本题分数	30
得 分	

二、应用题 (共 3 题，每题 10 分)

5. （10分）已知序列(85,55,48,15,46,34,28,16,92,60)，设计一种数据结构存储10个数据，数据区未满时一直 数据，数据区满之 每增加一个数据 删除当前最大数据。 添加50，88，62,则删除的应该是92，88，84。给出算法思想，并画出执行的过程。

6. （10 分）下图为 路线图，现需在 间 设 ，边权为 设线路所用成本，请 一种算法，给出！得" 可以＃通的最小代＄方%。要求说明算法思想，给出执行过程示意图。



7．（10 分）一个包含四代人家庭（A 有 3 个孩子 A1、A2、A3；A1 有 1 个孩子 A11； A2 有 2 个孩子 A21、A22；A3 有 3 个孩子 A31、A32、A33；A21 有 1 个孩子 A21；A31 有 2 个孩子A311、A312；A32 有1个孩子 （ A321），画出该家&的&' 示意图和存储结构图。

给出求解第 k 代所有人员的算法思想（成员 A 表示 k=1）。

本题分数	30
得　分	

三、编程题(共 3 题，每题 10 分)

8．（10 分）设单链表的元素为整型数据，编写函数删除）复元素的结点。要求先给出算法思想，再写出相应代码，并分析算法的时间复杂度。

9.（10 分）采用* 递　的方法＋计二叉树的，子结点数- 。要求先给出算法思想，再写出相应代码。

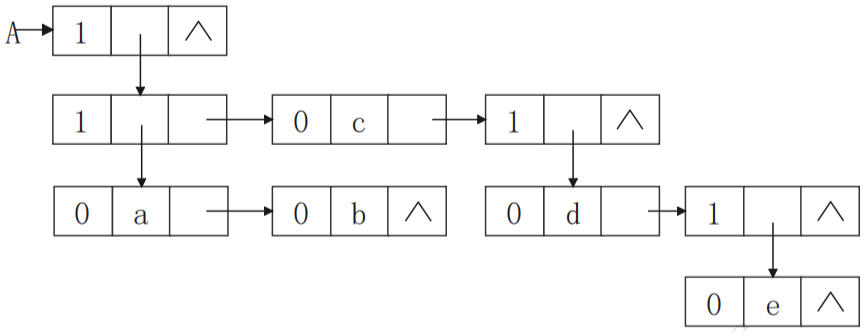
10.（10 分）设计一种算法.　有／图是否存在〇路。要求先给出算法思想，再写出相应代码。

一、解答题

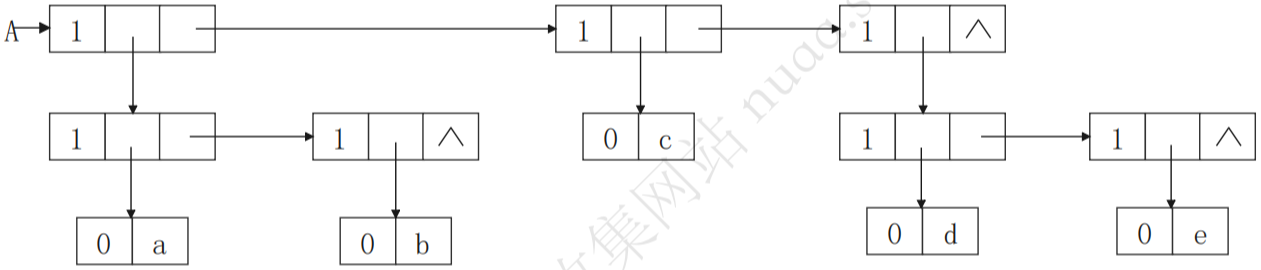
1.【解析】

存储广义表的结构有两种：层次结构和表头表尾结构。

层次结构：



表头表尾结构：



广义表中的层次结构和表头表尾结构是两种不同的表示方式，它们都用于描述广义表的结构和元素之间的关系。

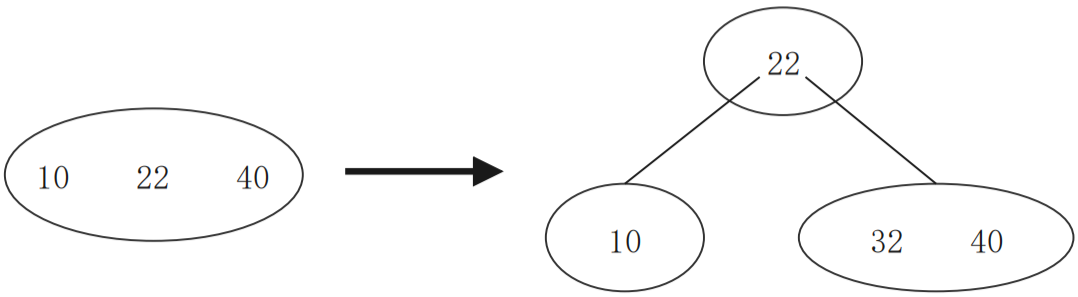
层次结构指的是广义表中元素之间的嵌套关系，即广义表可以由多个子表组成，而每个子表又可以包含其他子表。

表头表尾结构是一种将广义表表示为表头和表尾的形式，其中表头是广义表的第一个元素，表尾是剩余的元素。如果广义表为空，则表头和表尾都是空列表。

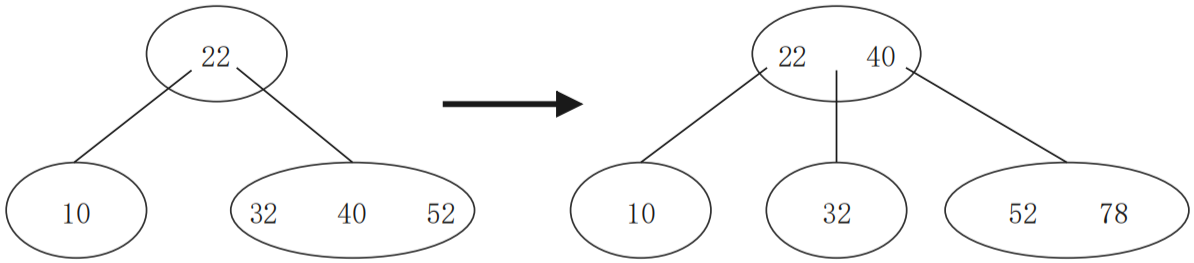
2.【解析】

3 阶 B-树中，每个结点最多存储三个关键字，最多有四个子结点。

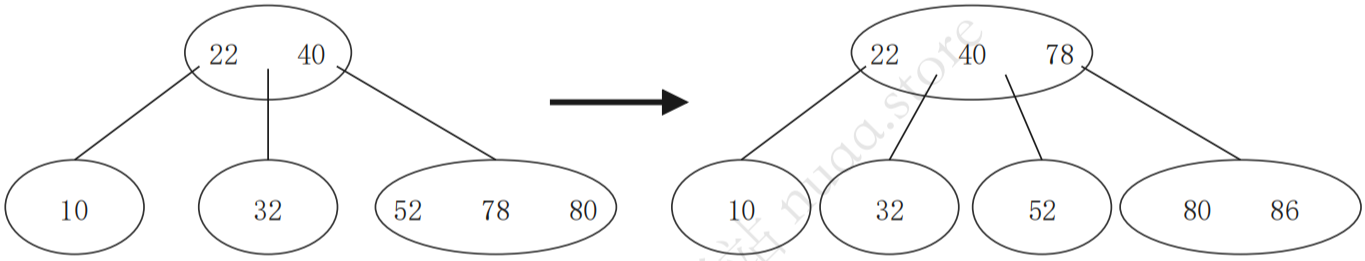
首先依次插入 10、22、40 构成一个结点。当插入 32 时，目前有四个关键字，因此结点需要分裂，将 22 作为新根结点，



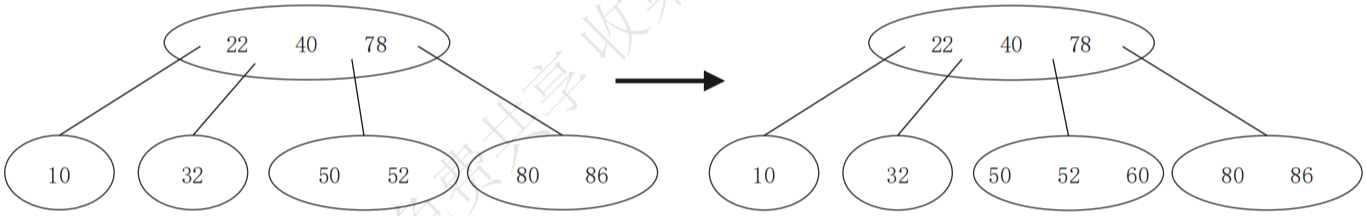
然后插入 52，根据 B-树性质可知，直接插入在 40 之后即可。当插入 78 时，右下角结点已经饱和，需要将中间元素 40 移动到父结点中。



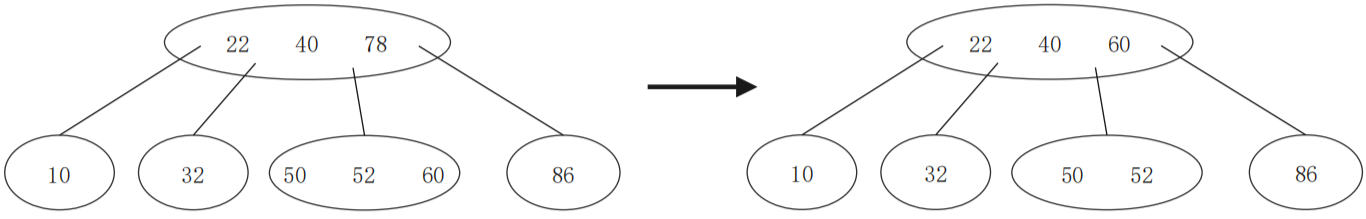
接下来插入 80。根据 B-树性质直接插入到 78 之后即可。当插入 86 时，右下角结点已经饱和，需要将中间元素 78 移动到父结点中。



再插入 50 和 60。直接插入到第二层第三个结点中即可。



现在删除 80。直接将 80 从 B-树中移除即可。最后删除 78。78 移除后，顶层结点缺少一个关键字元素，则需要根据 B-树性质，选择孩子结点中左侧结点最大值代替 78 原位置，即将 60 替换 78 即可。



3. 【解析】

哈希表的定义：哈希表实际上是一个具有固定大小的数组，其中每条记录的存储位置和它的关键字之间有一个确定的对应关系 H，称为哈希函数。通过哈希函数的映射得到的关键字在哈希表中的存储位置称为哈希地址。将关键字从原本很大的取值空间压缩至较小的查找表

内的存储空间，这将导致不同的关键字经过哈希函数的映射后，得到相同哈希地址的现象，称为冲突，而产生冲突的关键字也叫作同义词。

首先构造哈希表如下：

0	1	2	3	4	5	6	7	8	9

按照题目所给的哈希函数，分别将 12,4,25,60,87 对 10 取模，得到 2,4,5,0,7 则将对应的值填入哈希表中，得到：

0	1	2	3	4	5	6	7	8	9
60		12		4	25		87		

然后将 34 对 4 取模，得到 4，但地址 4 已经有元素占用。按照题意使用线性探测法解决冲突问题。将地址自增并对 10 取模，得到地址 5，仍然被占用。继续自增并取模，得到地址 6，因此将 34 填入地址 6。再将 99 对 10 取模，得到 9，将 99 填入地址 9，得到：

0	1	2	3	4	5	6	7	8	9
60		12		4	25	34	87		99

此时还剩下 52。将 52 对 10 取模，得到 2，地址 2 已有元素占用，使用线性探测法解决冲突问题。将地址自增并对 10 取模，得到地址 3，将 52 填入地址 3，得到：

0	1	2	3	4	5	6	7	8	9
60		12	52	4	25	34	87		99

4. 【解析】

归并排序定义：归并排序是将两个或两个以上的有序序列组合成一个新的有序序列。

2 路归并排序：设初始序列含有 n 个记录，则可将其看成 n 个有序的子序列，每个子序列的长度为 1。然后两两归并，得到 n/2 个长度为 2（剩余的不变）的有序子序列，再两两归并。如此重复，直至得到一个长度为 n 的有序序列为，这种排序方法称为 2 路归并排序。

将下划线连接的两个大括号内的元素排序为一个有序整体，多次重复即可。排序过程如下：

开　始：{18} {499}　{5}　{22}　{81}　{24}　{1}　{35}　{7}　{46}

第 1 次：{18　499}　{5　22}　{24　81}　{1　35}　{7　46}

第 2 次：{5　18　22　499}　{1　24　35　81}　{7　46}

第 3 次：{1 5 18 22 24 35 81 499} {7 46}

第 4 次：{1 5 7 18 22 24 35 46 81 499}

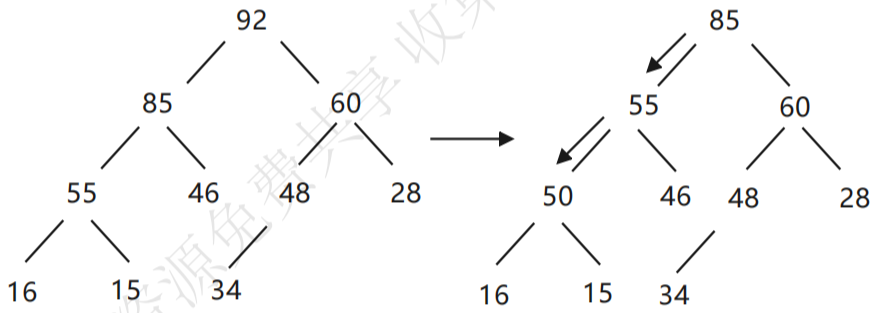
二、应用题

1. 【解析】

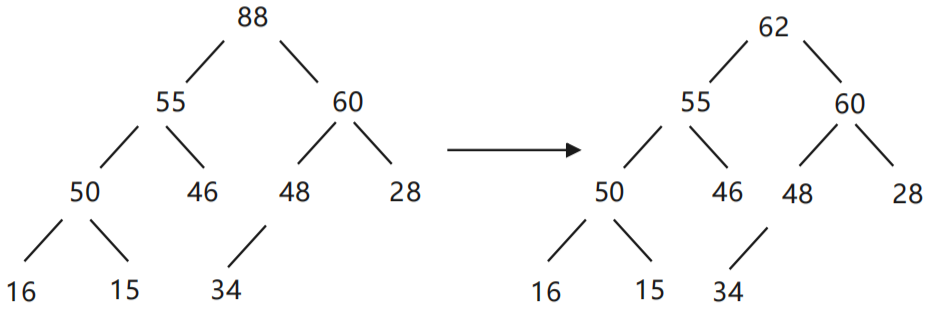
根据题目要求“数据区未满时一直填充数据，数据区满之后每增加一个数据就删除当前最大数据”可知，堆这一数据结构能够很好地满足。整个过程如下：

首先将序列建成一个大顶堆。在顺序结构的[low,high]范围内，实现“筛选”算法。设 low 为根结点，其左右子树均已是大顶堆，将根结点值与左、右子树的根结点大者进行比较:若较之更大或相等，则已是大顶堆；否则，将根结点值与大者交换，并继续向下比较，直到已是大顶堆为止。

建堆结果如下，将新加入的数据 50 覆盖根节点（最大值），然后继续进行堆调整，得到右侧结果。



接下来将最大值 85 替换为 88，最大值仍然是 88。然后将 88 替换为 62，得到右侧结果：



2. 【解析】

本题考察最小生成树算法——普里姆算法。在计算最小生成树时，普里姆算法的初始状态仅

有一个顶点在最小生成树的顶点集合 U 中，其他顶点都在另一个由不在最小生成树上的顶点构成的集合 V 中。在后续的每一步中，通过选择所有连接最小生成树上的顶点 和不在树上的顶点之间的边中权值最小的边 (u,v)，将对应的顶点拉入最小生成树的顶点集合中。当图中所有的顶点都已加入到树中时，算法运行结束，此时得到的 n 个顶点和 n-1 条边就构成了一棵最小生成树。以下是算法执行过程：

首先将辅助表格初始化如下：

	1	2	3	4	5	6
lowcost	0	2	∞	∞	10	4
adjvex	1	1	1	1	1	1
flag	1	0	0	0	0	0

选中 lowcost 最小且 flag 为 false 的顶点 2，更新整个表格：

	1	2	3	4	5	6
lowcost	0	2	∞	10	10	3
adjvex	1	1	1	1	1	1
flag	1	1	0	0	0	0

选中 lowcost 最小且 flag 为 false 的顶点 6，更新整个表格：

	1	2	3	4	5	6
lowcost	0	2	11	7	8	3
adjvex	1	1	3	4	6	1
flag	1	1	0	0	0	1

选中 lowcost 最小且 flag 为 false 的顶点 4，更新整个表格：

	1	2	3	4	5	6
lowcost	0	2	7	7	8	3
adjvex	1	1	4	4	6	1
flag	1	1	0	1	0	1

选中 lowcost 最小且 flag 为 false 的顶点 3，更新整个表格：

	1	2	3	4	5	6
lowcost	0	2	7	7	8	3
adjvex	1	1	4	4	6	1

flag	1	1	1	1	0	1

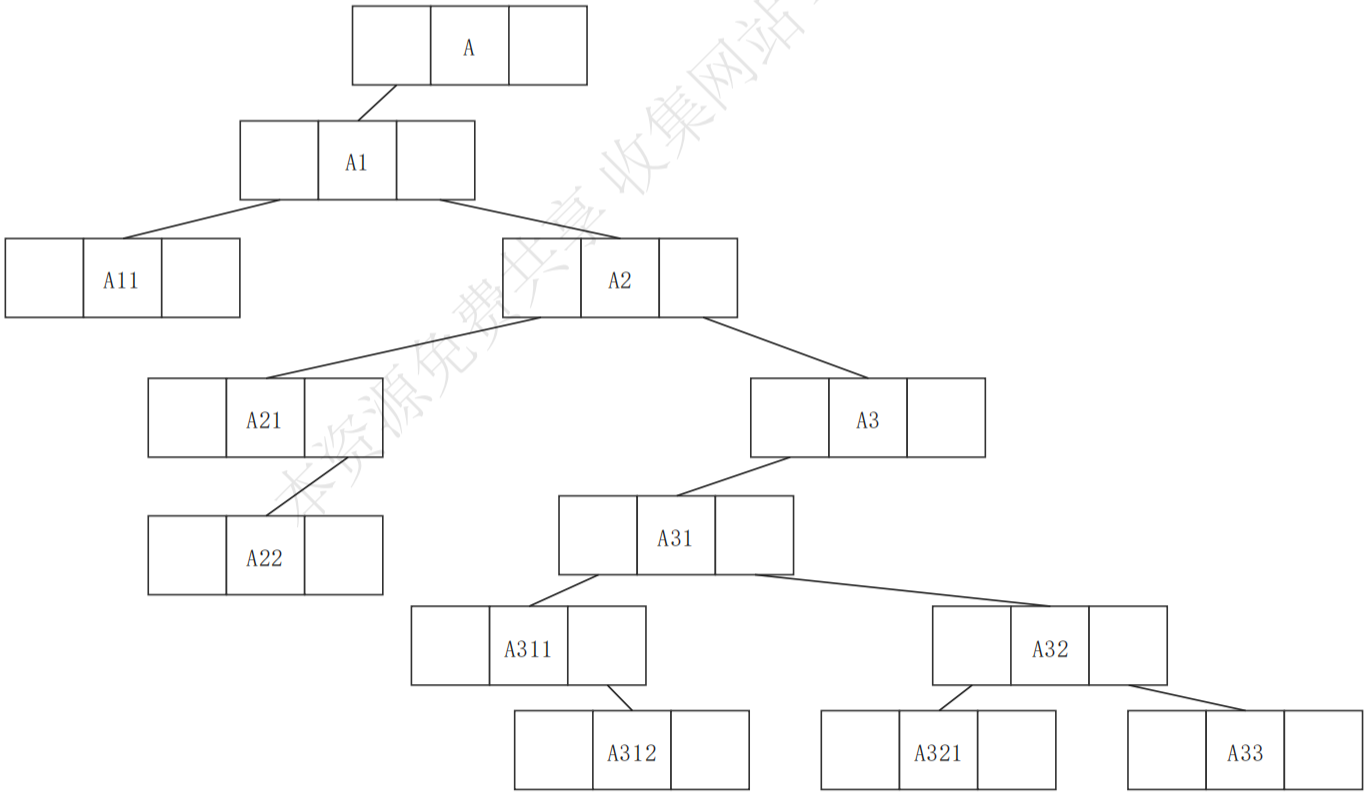
最后选中顶点 5，算法结束。

	1	2	3	4	5	6
lowcost	0	2	7	7	8	3
adjvex	1	1	4	4	6	1
flag	1	1	1	1	1	1

3. 【解析】

使用树的孩子兄弟表示法表示该家族的族谱示意图。在这棵树的结点中包含三个元素：孩子结点指针、数据域和兄弟结点指针（从左往右）。

运用上述表示方法可将族谱表示如下：



结点定义如下：

```
struct TreeNode {
    int data;
    TreeNode* firstChild;
```

```
    TreeNode* nextSibling;
};
```

求解算法如下：

```
int countNodesAtLevel(TreeNode* root, int level) {
    if (root == nullptr || level < 1) {
        return 0;
    }

    std::queue<TreeNode*> q;
    q.push(root);
    int currentLevel = 1;
    int count = 0;

    while (!q.empty()) {
        int size = q.size();

        // 遍历当前层的所有节点
        for (int i = 0; i < size; ++i) {
            TreeNode* node = q.front();
            q.pop();

            if (currentLevel == level) {
                // 如果当前层和目标层级匹配，则增加计数
                ++count;
            }

            // 将孩子节点加入队列
            TreeNode* child = node->firstChild;
            while (child != nullptr) {
                q.push(child);
```

```
                child = child->nextSibling;
            }
        }

        // 更新当前层级
        ++currentLevel;

        if (currentLevel > level) {
            // 已经超过目标层级，可以提前结束遍历
            break;
        }
    }

    return count;
}
```

三、应用题

1. 【解析】

算法思想：使用两个指针，外层指针 current 用于遍历链表，内层指针 runner 用于检查 current 后面的节点是否与 current 相同。对于每个 current 节点，从其下一个节点开始，与后面的节点逐一比较。如果找到与 current 相同的节点，则将该节点从链表中删除。如果未找到相同节点，则将 runner 指针向后移动一位。继续遍历下一个 current 节点，重复上述步骤，直到遍历完整个链表。

代码如下：

```
struct ListNode {
    int data;
    ListNode* next;
};

void deleteDuplicates(ListNode* head) {
    if (head == nullptr) {
```

```
        return;
    }

    ListNode* current = head;

    while (current != nullptr) {
        ListNode* runner = current;

        while (runner->next != nullptr) {
            if (runner->next->data == current->data) {
                // 找到与 current 相同的节点，删除该节点
                ListNode* duplicate = runner->next;
                runner->next = runner->next->next;
                delete duplicate;
            } else {
                // 未找到相同节点，继续向后移动 runner 指针
                runner = runner->next;
            }
        }

        current = current->next;
    }
}
```


2. 【解析】

算法思想：使用非递归的方式遍历二叉树，在遍历的过程中统计叶子节点的数量。使用栈来辅助进行遍历。从根节点开始，将根节点入栈。进入循环，直到栈为空。在循环中，执行以下操作：弹出栈顶节点，并检查其左右子节点。如果左右子节点都为空，则说明当前节点是叶子节点，将叶子节点的数量加一。如果左子节点不为空，则将左子节点入栈。如果右子节点不为空，则将右子节点入栈。循环结束后，得到叶子节点的数量。

代码如下：

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
};

int countLeafNodes(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }

    int leafCount = 0;
    std::stack<TreeNode*> nodeStack;
    nodeStack.push(root);

    while (!nodeStack.empty()) {
        TreeNode* current = nodeStack.top();
        nodeStack.pop();

        if (current->left == nullptr && current->right == nullptr) {
            // 当前节点是叶子节点
            leafCount++;
        }

        if (current->right != nullptr) {
            // 右子节点入栈
            nodeStack.push(current->right);
        }
```

```
            if (current->left != nullptr) {
                // 左子节点入栈
                nodeStack.push(current->left);
            }
        }

        return leafCount;
    }
```

3.【解析】

可以使用深度优先搜索（DFS）算法来检测有向图是否存在回路。具体思想如下：对于每个节点，维护一个状态，有三种可能的状态：未访问（0）、正在访问（1）和已完成访问（2）。对于每个未访问的节点，以该节点为起点进行深度优先搜索。在深度优先搜索的过程中，对于当前节点，将其状态设置为正在访问（1）。对于当前节点的每个邻接节点，进行递归调用深度优先搜索。如果邻接节点的状态为正在访问（1），则说明存在回路，直接返回 true。如果邻接节点的状态为未访问（0），则继续以邻接节点为起点进行深度优先搜索。当前节点的所有邻接节点都完成访问后，将当前节点的状态设置为已完成访问（2）。如果在遍历完所有节点后没有发现回路，则返回 false。

代码如下：

```
enum class State {
    Unvisited,
    Visiting,
    Visited
};

bool hasCycleDFS(int node, vector<std::vector<int>>& adjList, vector<State>& states, vector<int>& result) {
    states[node] = State::Visiting;

    for (int neighbor : adjList[node]) {
        if (states[neighbor] == State::Visiting) {
```

```
            // 邻接节点正在访问，存在回路
            return false;
        } else if (states[neighbor] == State::Unvisited) {
            // 以邻接节点为起点进行深度优先搜索
            if (!hasCycleDFS(neighbor, adjList, states, result)) {
                return false;
            }
        }
    }

    states[node] = State::Visited;
    result.push_back(node);  // 将节点添加到结果列表中
    return true;
}

vector<int> topologicalSort(vector<vector<int>>& adjList) {
    int numNodes = adjList.size();
    std::vector<State> states(numNodes, State::Unvisited);
    std::vector<int> result;

    for (int i = 0; i < numNodes; i++) {
        if (states[i] == State::Unvisited) {
            if (!hasCycleDFS(i, adjList, states, result)) {
                // 存在回路，返回空的结果列表
                return {};
            }
        }
    }
```

```
    std::reverse(result.begin(), result.end());  // 将结果列表逆序，得到拓扑排序的顺序
    return result;
}

bool hasCycle(vector<std::vector<int>>& adjList) {
    std::vector<int> result = topologicalSort(adjList);
    return result.empty();  // 如果结果列表为空，说明存在回路
}
```