

《数 据 结 构》

实验报告

|  |  |
| --- | --- |
| 项目名称： |  |
| 班 级： |  |
| 学生姓名**:** |  |
| 学生学号： |  |
| 指导教师： |  |
| 成 绩： |  |

201 年 月 日 成都信息工程大学 计算机学院

1. 实现功能描述

在这段代码中，程序的功能集中于二叉树的创建和遍历。代码首先定义了二叉树节点的结构体 BiTNode，并通过 BiTree 类型定义了二叉树指针。此外，代码也定义了基本的返回状态 Status 和节点数据类型 TElemType，这些类型定义为后续操作提供了基础。

创建二叉树的核心函数是 CreateBiTree，通过递归方式实现。函数从用户输入读取节点数据，当输入为0时表示该节点为空，不创建新的节点；否则，将输入值赋予当前节点的数据域，并递归创建其左子树和右子树。这样的设计允许根据用户输入逐步构建出一个二叉树结构。

代码接下来实现了三种递归方式的遍历：前序、中序和后序。在前序遍历函数 PreOrderTraverse 中，程序按「根节点 -> 左子树 -> 右子树」的顺序依次访问节点；在中序遍历函数 InOrderTraverse 中，程序按「左子树 -> 根节点 -> 右子树」顺序遍历，而在后序遍历函数 PostOrderTraverse 中，节点的访问顺序为「左子树 -> 右子树 -> 根节点」。这三种递归函数在访问每个节点的子树时，都通过递归调用自身来实现，直到子树为空。

此外，代码还实现了三种非递归的遍历方式，分别模拟了前序、中序和后序的遍历过程。每种非递归遍历方法都使用栈来保存待访问的节点，以此实现递归的效果。在前序遍历的非递归实现 PreOrderTraverse\_No\_Recursion 中，程序首先访问根节点并将其压入栈中，再持续访问左子树的节点。若左子树为空，程序从栈中弹出节点并访问右子树。在中序遍历的非递归实现 InOrderTraverse\_No\_Recursion 中，程序将所有左子树节点依次压入栈中，直到无法继续；此时弹出栈顶节点并访问其右子树。而后序遍历的非递归实现 PostOrderTraverse\_No\_Recursion 相对更复杂，程序不仅通过栈记录节点，还需记录是否访问过右子树，以避免重复遍历。

在 main 函数中，程序依次调用二叉树创建函数和六种遍历方式（递归与非递归各三种），并在控制台输出每种遍历的结果。通过这些函数，程序实现了二叉树的基本构建和多种遍历方式，为进一步的二叉树操作打下了基础。

1. 方案比较与选择

在实现二叉树的方案中，从数据结构的逻辑结构与存储结构出发，可以有多种方式。通常，二叉树的实现可以选择链式存储结构或顺序存储结构。这两种存储结构在具体的应用场景和性能表现上各有优势，因此我们可以从多个角度来比较和选择。

首先，从逻辑结构的角度来看，二叉树的逻辑结构为树状，每个节点至多有两个孩子节点，且各层的节点数在逻辑上成指数增长。针对这样的结构，采用链式存储结构比较合适，因为它可以动态地为每个节点分配存储空间，不需要考虑树的深度或节点的密集程度；每个节点包含指向左、右子节点的指针，构成一组链表形式的连接。另一方面，顺序存储结构将二叉树的节点顺序存储在数组中，通常适合满二叉树或完全二叉树等密集结构，可以通过下标计算找到父子关系。但在一般的二叉树或稀疏树结构中，顺序存储会浪费大量空间，因为节点间关系固定，空节点也需要占用位置。

其次，从存储结构的具体实现角度来看，链式结构由于采用动态分配，可以按需创建节点；每个节点只存储必要的信息（数据和指针），从而在存储空间上更高效。而顺序存储结构需要预分配较大空间，尤其在稀疏结构中浪费空间较大，不够灵活，且树的节点位置不可随意改变，否则会破坏父子关系。

从时空效率角度分析，链式存储结构在时间复杂度上和顺序存储结构差别不大，都可以实现O(n)的遍历效率。在空间效率上，链式结构的效率优于顺序结构，因为链式结构仅为非空节点分配存储空间，不存在额外的冗余；而顺序结构在节点较少或分布不均时浪费较大。因此最终选择链式存储结构作为实现方案，因为链式存储更符合一般二叉树不规则、动态的特点，能够更灵活地实现节点的添加、删除和遍历，同时节省空间，避免冗余数据存储带来的浪费。

综合而言，链式存储结构在满足逻辑结构、空间效率等多方面要求的基础上，是更优的实现方案。这种结构适应性强且操作灵活，特别是在稀疏或非完全二叉树中具有显著的优势。

1. 设计算法描述

算法的设计主要围绕二叉树的构建和遍历展开。在这套算法中，我们采用链式存储结构，使用节点指针构建二叉树的结构。通过递归和非递归的方式分别实现前序、中序和后序遍历。在此，我将分别描述算法的核心思想，并提供逻辑结构示例和模块功能划分图，以帮助更好地理解每个模块的作用及流程。

(1) 算法思想说明

通过链式存储结构，每个节点包含数据域 data 以及指向左右子节点的指针 lchild 和 rchild。在二叉树创建过程中，用户输入各个节点的数据值并递归创建左右子树，输入 0 表示空节点。这样，每个节点通过指针关联构成了二叉树的链式结构。

以示例二叉树为例，假设用户输入了序列 1 2 0 0 3 0 4 0 0，创建的二叉树结构如下：

1

/ \

2 3

\

4

在此结构中：

根节点为 1，其左子节点为 2，右子节点为 3。

节点 2 的左右子节点为空。

节点 3 的右子节点为 4，左子节点为空。

利用递归遍历可以轻松访问二叉树的各节点，而非递归遍历通过栈实现节点的顺序访问。栈模拟递归调用过程，特别在前序、中序和后序遍历时，通过判断左右子节点是否存在来决定访问顺序。

(2) 模块划分与系统功能框图

算法的模块设计主要包含以下几个部分：

二叉树创建模块：负责动态地接收用户输入数据，递归地创建并连接左右子节点。

遍历模块：包含前序、中序和后序遍历三种方式，每种方式又分为递归和非递归两种实现。

主程序模块：负责调用创建和遍历功能，并输出遍历结果。

以下是系统的主要功能组成框图：

+-----------------------------+

| 系统功能组成图 |

+-----------------------------+

| |

| 二叉树创建模块 |

| 创建二叉树节点 |

+-----------------------------+

| |

| 遍历模块 |

| 包括前序、中序、后序遍历 |

+-----------------------------+

| |

| 主程序模块 |

| 调用创建和遍历功能 |

+-----------------------------+

(3) 模块设计符合书中对应章节的模块要求

该设计符合二叉树章节中的基本功能模块要求。每个功能模块都通过函数实现，结构清晰，尤其在遍历实现中，递归和非递归方式提供了不同的实现思路，使得结构更加完善。

(4) 关键算法流程图

以下流程图展示了二叉树的创建过程以及前序和中序遍历的非递归实现：

1. 二叉树创建流程图

+------------------------+

| 输入节点数据 |

+------------------------+

|

v

+------------------------+

| 判断数据是否为 0 |

| (0 表示空节点) |

+------------------------+

/ \

/ \

v v

+---------+ +--------------------+

| 返回空 | | 创建新节点 |

+---------+ +--------------------+

/ \

/ \

v v

+-----------------+ +----------------+

| 创建左子树 | | 创建右子树 |

+-----------------+ +----------------+

2. 前序遍历（非递归）流程图

+---------------------+

| 初始化栈和指针 p |

+---------------------+

|

v

+---------------------+

| 判断 p 是否存在或栈 |

| 是否为空 |

+---------------------+

|

|

v

+---------------------+

| 输出 p 的数据 |

+---------------------+

|

v

+---------------------+

| p 入栈，访问左子树 |

+---------------------+

|

v

+---------------------+

| 若左子树为空， |

| 则从栈中取节点访问 |

| 右子树 |

+---------------------+

3. 中序遍历（非递归）流程图

+----------------------+

| 初始化栈和指针 p |

+----------------------+

|

v

+----------------------+

| 判断 p 或栈是否为空 |

+----------------------+

|

v

+----------------------+

| 若 p 存在，p 入栈 |

| 继续访问左子树 |

+----------------------+

|

v

+----------------------+

| 若左子树为空， |

| 从栈中弹出节点， |

| 输出数据并访问右子树 |

+----------------------+

1. 算法实现

#include<stdio.h>

#include<stdlib.h>

#define OVERFLOW -2

#define OK 1

typedef int TElemType;

typedef int Status;

typedef struct BiTNode

{

TElemType data;

struct BiTNode \*lchild, \*rchild;

}BiTNode, \*BiTree;

Status CreateBiTree(BiTree \*T);

Status PreOrderTraverse(BiTree T);

Status InOrderTraverse(BiTree T);

Status PostOrderTraverse(BiTree T);

Status PreOrderTraverse\_No\_Recursion(BiTree T);

Status InOrderTraverse\_No\_Recursion(BiTree T);

Status PostOrderTraverse\_No\_Recursion(BiTree T);

int main(void)

{

BiTree T;

CreateBiTree(&T);

printf("PreOrderTraverse: ");

PreOrderTraverse(T);

printf("\nInOrderTraverse: ");

InOrderTraverse(T);

printf("\nPostOrderTraverse: ");

PostOrderTraverse(T);

printf("\nPreOrderTraverse\_No\_Recursion: ");

PreOrderTraverse\_No\_Recursion(T);

printf("\nInOrderTraverse\_No\_Recursion: ");

InOrderTraverse\_No\_Recursion(T);

printf("\nPostOrderTraverse\_No\_Recursion: ");

PostOrderTraverse\_No\_Recursion(T);

return 0;

}

Status CreateBiTree(BiTree \*T)

{

TElemType ch;

scanf("%d", &ch);

if (ch == 0) // 以0表示空节点

{

\*T = NULL;

}

else

{

\*T = (BiTree)malloc(sizeof(BiTNode));

if (!\*T)

exit(OVERFLOW);

(\*T)->data = ch;

CreateBiTree(&(\*T)->lchild);

CreateBiTree(&(\*T)->rchild);

}

return OK;

}

Status PreOrderTraverse(BiTree T)

{

if (T)

{ printf("%d", T->data);

PreOrderTraverse(T->lchild);

PreOrderTraverse(T->rchild);

}

return OK;

}

Status InOrderTraverse(BiTree T)

{

if (T)

{

InOrderTraverse(T->lchild);

printf("%d", T->data);

InOrderTraverse(T->rchild);

}

return OK;

}

Status PostOrderTraverse(BiTree T)

{

if (T)

{

PostOrderTraverse(T->lchild);

PostOrderTraverse(T->rchild);

printf("%d", T->data);

}

return OK;

}

Status PreOrderTraverse\_No\_Recursion(BiTree T)

{

BiTree Stack[100];

int top = -1;

BiTree p = T;

while (p || top != -1)

{

while (p)

{

printf("%d", p->data);

Stack[++top] = p;

p = p->lchild;

}

if (top != -1)

{

p = Stack[top--];

p = p->rchild;

}

}

return OK;

}

Status InOrderTraverse\_No\_Recursion(BiTree T)

{

BiTree Stack[100];

int top = -1;

BiTree p = T;

while (p || top != -1)

{

while (p)

{

Stack[++top] = p;

p = p->lchild;

}

if (top != -1)

{

p = Stack[top--];

printf("%d", p->data);

p = p->rchild;

}

}

return OK;

}

Status PostOrderTraverse\_No\_Recursion(BiTree T)

{

BiTree Stack[100];

int top = -1;

BiTree p = T;

BiTree q = NULL;

while (p || top != -1)

{

while (p)

{

Stack[++top] = p;

p = p->lchild;

}

if (top != -1)

{

p = Stack[top];

if (p->rchild && p->rchild != q)

{

p = p->rchild;

Stack[++top] = p;

p = p->lchild;

}

else

{

p = Stack[top--];

printf("%d", p->data);

q = p;

p = NULL;

}

}

}

return OK;

}

1. 实验结果测试与分析

PS D:\College\_Work> & 'c:\Users\18080\.vscode\extensions\ms-vscode.cpptools-1.23.1-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-t4mhwdg0.yak' '--stdout=Microsoft-MIEngine-Out-fnsfqy5a.jli' '--stderr=Microsoft-MIEngine-Error-kzkavkcb.llq' '--pid=Microsoft-MIEngine-Pid-f3zyfc4o.nvc' '--dbgExe=C:\Program Files\mingw64\bin\gdb.exe' '--interpreter=mi'

1 2 0 0 3 4 0 0 5 0 0

PreOrderTraverse: 12345

InOrderTraverse: 21435

PostOrderTraverse: 24531

PreOrderTraverse\_No\_Recursion: 12345

InOrderTraverse\_No\_Recursion: 21435

PostOrderTraverse\_No\_Recursion: 24531

1. 思考及学习心得

在此次实验过程中，通过构建二叉树和实现多种遍历算法，我对二叉树的链式存储结构有了更深入的认识。起初，二叉树这种树状结构看起来较为抽象，但通过逐步实现节点的动态创建和递归连接，我逐渐体会到链式结构的优点——特别是在需要高效地操作动态节点数据时，链式结构在灵活性和空间节约上显得尤为合适。同时，通过遍历实现，链式结构的递归调用也更加自然，而非递归遍历时的栈结构设计，则为理解递归过程提供了另一种思路。

在学习方法上，我的体会是动手实践与逻辑思考的结合能有效加深理解。尤其在非递归遍历的实现过程中，我遇到了难点，例如如何在后序遍历时避免节点重复访问。此时，通过画图模拟操作，推敲每一步的执行过程，我加深了对栈在非递归遍历中的应用理解。反复的代码调试帮助我理解每个指针的作用，学习到如何在复杂逻辑中保持清晰的思路。

针对前面的思考题，我深刻体会到选择链式存储结构是合理的。在树节点分布较稀疏时，链式结构的动态分配节省了存储空间，使得操作更加灵活，而顺序存储结构的固定布局则显得浪费。通过链式结构的灵活性，我们可以更高效地实现节点的插入和删除等操作，特别是在需要动态修改数据结构的场景下，链式存储显然更为优越。

总体而言，此次实验让我加深了对二叉树结构及其操作方式的理解，同时让我意识到分析数据结构的存储方式对解决具体问题的重要性。在未来的学习中，我会继续通过结合理论与实践，探索不同的数据结构特性，以更全面地掌握计算机科学的基础知识。