# 北京邮电大学软件学院

# 2019-2020学年第1学期实验报告

**课程名称： 算法与数据结构**

**实验名称： 实验四**

**实验完成人：**

**姓名：**\_\_王衔飞\_\_\_**学号：**\_ \_\_**成绩：**\_\_\_\_\_\_\_\_

**指导教师：**\_\_\_\_\_\_贾红娓 \_李璐路\_\_\_\_\_\_\_\_

**日 期： 2019 年 11 月 7 日**

1. **实验目的**

树是一种应用极为广泛的数据结构，也是这门课程的重点。它们的特点在于非线性。广义表本质上是树结构。本章实验继续突出了数据结构加操作的程序设计观点，但根据这两种结构的非线性特点，将操作进一步集中在遍历操作上，因为遍历操作是其他众多操作的基础。遍历逻辑的（或符号形式的）结构，访问动作可是任何操作。本次实验希望帮助学生熟悉各种存储结构的特征，以及如何应用树结构解决具体问题（即原理与应用的结合）。

1. **实验内容**

**必做内容**

* + 1. **二叉树的建立与遍历**

**[问题描述]**

　　建立一棵二叉树，并对其进行遍历（先序、中序、后序），打印输出遍历结果。

**[基本要求]**

　　从键盘接受输入（先序），以二叉链表作为存储结构，建立二叉树（以先序来建立），并采用递归算法对其进行遍历（先序、中序、后序），将遍历结果打印输出。

**[测试数据]**

ABCффDEфGффFффф（其中ф表示空格字符）则输出结果为

先序：ABCDEGF

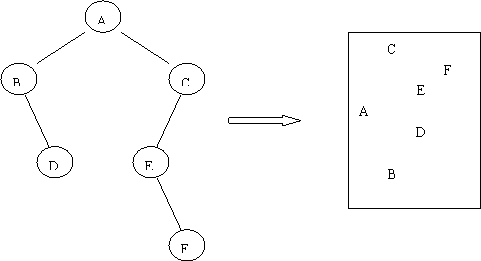
　　中序：CBEGDFA

　　后序：CGBFDBA

* + 1. **打印二叉树结构**

**[问题描述]**

　　按凹入表形式横向打印二叉树结构，即二叉树的根在屏幕的最左边，二叉树的左子树在屏幕的下边，二叉树的右子树在屏幕的上边。例如：



**[测试数据]**

　　由学生依据软件工程的测试技术自己确定。注意测试边界数据，如空二叉树。

**[实现提示]**

　　（1）利用RDL遍历方法；

（2）利用结点的深度控制横向位置。

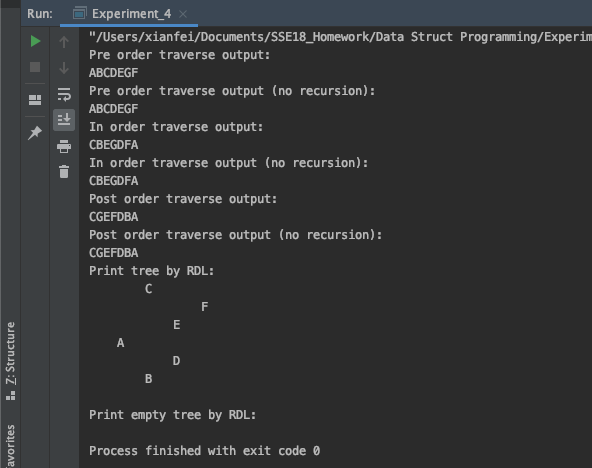
**选做内容**

**采用非递归算法实现二叉树遍历。**

1. **实验环境**

OS：macOS 10.15 IDE：Clion 2019.2 Compiler：GCC8.1

1. **实验结果**

完成了所有任务，包括前序创建树、前中后序递归和不递归遍历以及打印二叉树。其中递归按照题目所给测试数据进行测试，打印按照题目所给及空树进行测试，截图如下。

1. **附录**

心得：递归遍历很简单，更改输出语句位置即可。主要难度在于非递归遍历。前序不递归依靠一个stack用于存储结点地址，思路为遇到右子树就入栈，遍历完左子树后弹出右子树所在的栈，继续进行，直到stack为空。中序遍历也依靠一个栈，思路为一路寻找左子树，并将双亲结点入栈。后序遍历非递归最为复杂，简单思路为使用两个栈，一个存结点地址一个存数据，把DLR前序遍历的LR交换，变为DRL，方法几乎同前序遍历。使用stack逆序输出变为LRD即后序。

打印输出树时，将中序遍历LDR的LR交换变为RDL，并在每次递归时记录深度用于控制缩进。

源代码：

// main.cpp文件

#include <iostream>  
#include <stdexcept>  
#include "stack.hpp"  
  
template<typename Type>  
struct BiTree {  
 Type data;  
 BiTree \*lchild = nullptr, \*rchild = nullptr;  
};  
  
// 将数组转化为指针  
template<typename Type,typename T>  
void createBiTree(BiTree<Type> \*&biTreePtr, T ptr\_) {  
 const Type\* ptr=ptr\_;  
 createBiTree(biTreePtr,ptr);  
}  
  
template<typename Type>  
void createBiTree(BiTree<Type> \*&biTreePtr, const Type \*&arrayPtr) {  
 if (!\*arrayPtr)return;  
 if (\*arrayPtr == '#') {  
 arrayPtr++;  
 return;  
 } else {  
 if (!biTreePtr)biTreePtr = new BiTree<Type>;  
 biTreePtr->data = \*arrayPtr;  
 arrayPtr++;  
 }  
 createBiTree(biTreePtr->lchild, arrayPtr);  
 createBiTree(biTreePtr->rchild, arrayPtr);  
}  
  
// 前序遍历（参数：二叉树的指针，遍历执行函数指针）  
template<typename Type, typename FunPtr>  
void preOrderTraverse(BiTree<Type> \*biTreePtr, FunPtr fun) {  
 if (biTreePtr == nullptr) return;  
 fun(biTreePtr->data);  
 preOrderTraverse(biTreePtr->lchild, fun);  
 preOrderTraverse(biTreePtr->rchild, fun);  
}  
  
// 非递归前序遍历（参数：二叉树的指针，遍历执行函数指针）  
template<typename Type, typename FunPtr>  
void preOrderTraverse\_noRecursion(BiTree<Type> \*biTreePtr, FunPtr fun) {  
 if (biTreePtr == nullptr) return;  
 Stack<BiTree<Type> \*> stack;  
 stack.push(biTreePtr);  
 while (!stack.isEmpty()) {  
 biTreePtr = stack.pop();  
 while (biTreePtr) {  
 fun(biTreePtr->data);  
 if (biTreePtr->rchild)stack.push(biTreePtr->rchild);  
 biTreePtr = biTreePtr->lchild;  
 }  
 }  
}  
  
// 中序遍历（参数：二叉树的指针，遍历执行函数指针）  
template<typename Type, typename FunPtr>  
void inOrderTraverse(BiTree<Type> \*biTreePtr, FunPtr fun) { // 中序遍历  
 if (biTreePtr == nullptr) return;  
 inOrderTraverse(biTreePtr->lchild, fun);  
 fun(biTreePtr->data);  
 inOrderTraverse(biTreePtr->rchild, fun);  
}  
  
// 非递归中序遍历（参数：二叉树的指针，遍历执行函数指针）  
// 思路：一路寻找左子树 找不着了就找左子树的双亲的右子树继续  
template<typename Type, typename FunPtr>  
void inOrderTraverse\_noRecursion(BiTree<Type> \*biTreePtr, FunPtr fun) { // 中序遍历  
 if (biTreePtr == nullptr) return;  
 Stack<BiTree<Type> \*> stack;  
 while(biTreePtr||!stack.isEmpty()){  
 if(biTreePtr){  
 stack.push(biTreePtr);  
 biTreePtr=biTreePtr->lchild;  
 }else{  
 biTreePtr=stack.pop();  
 fun(biTreePtr->data);  
 biTreePtr=biTreePtr->rchild;  
 }  
 }  
}  
  
// 后序遍历（参数：二叉树的指针，遍历执行函数指针）  
template<typename Type, typename FunPtr>  
void postOrderTraverse(BiTree<Type> \*biTreePtr, FunPtr fun) { // 后序遍历  
 if (biTreePtr == nullptr) return;  
 postOrderTraverse(biTreePtr->lchild, fun);  
 postOrderTraverse(biTreePtr->rchild, fun);  
 fun(biTreePtr->data);  
}  
  
// 非递归后序遍历（参数：二叉树的指针，遍历执行函数指针）  
// 思路：把DLR前序遍历的LR交换，变为DRL。使用stack逆序输出变为LRD即后序。  
template<typename Type, typename FunPtr>  
void postOrderTraverse\_noRecursion(BiTree<Type> \*biTreePtr, FunPtr fun) {  
 if (biTreePtr == nullptr) return;  
 Stack<BiTree<Type> \*> stack;  
 Stack<Type> result;  
 stack.push(biTreePtr);  
 while (!stack.isEmpty()) {  
 biTreePtr = stack.pop();  
 while (biTreePtr) {  
 result.push(biTreePtr->data);  
 if (biTreePtr->lchild)stack.push(biTreePtr->lchild);  
 biTreePtr = biTreePtr->rchild;  
 }  
 }  
 while(!result.isEmpty())fun(result.pop());  
}  
  
// RDL中序遍历（参数：二叉树的指针，遍历执行函数指针）  
template<typename Type>  
void printTreeByRDL(BiTree<Type> \*biTreePtr, int depth= 0) { // 中序遍历  
 if (biTreePtr == nullptr) return;  
 ++depth;  
 printTreeByRDL(biTreePtr->rchild, depth);  
 for(int i=0;i<depth;i++)printf("\t");  
 std::cout << (biTreePtr->data);  
 putchar('\n');  
 printTreeByRDL(biTreePtr->lchild, depth);  
}  
  
int main() {  
 // 测试遍历  
 BiTree<char> \*tree= nullptr;  
 createBiTree(tree, "ABC##DE#G##F###");  
 printf("Pre order traverse output: \n");  
 preOrderTraverse(tree, putchar);  
 printf("\nPre order traverse output (no recursion): \n");  
 preOrderTraverse\_noRecursion(tree, putchar);  
 printf("\nIn order traverse output: \n");  
 inOrderTraverse(tree, putchar);  
 printf("\nIn order traverse output (no recursion): \n");  
 inOrderTraverse\_noRecursion(tree, putchar);  
 printf("\nPost order traverse output: \n");  
 postOrderTraverse(tree, putchar);  
 printf("\nPost order traverse output (no recursion): \n");  
 postOrderTraverse\_noRecursion(tree, putchar);  
  
 // 测试RDL打印  
 BiTree<char> \*treeForPrint= nullptr;  
 createBiTree(treeForPrint,"AB#D##CE#F###");  
 printf("\nPrint tree by RDL:\n");  
 printTreeByRDL(treeForPrint);  
  
 // 测试RDL打印  
 BiTree<char> \*treeForPrint\_empty= nullptr;  
 createBiTree(treeForPrint\_empty,"");  
 printf("\nPrint empty tree by RDL:\n");  
 printTreeByRDL(treeForPrint\_empty);  
 return 0;  
}

stack.hpp文件：

//  
// Created by xianfei on 2019.  
//  
  
#ifndef STACK\_H  
#define STACK\_H  
  
#include <stdexcept>  
  
template <class T>  
struct Stack{  
 const int DEFAULT\_SIZE=10;  
 T\* array=(T\*)malloc(DEFAULT\_SIZE\*sizeof(T));  
 int size=0;  
 int capacity=DEFAULT\_SIZE;  
 T getTop(){  
 if(size==0)throw std::out\_of\_range("stack is empty");  
 return array[size-1];  
 }  
 void push(const T &num){  
 if(size>=capacity-1){  
 array=(T\*)realloc(array,2\*capacity\*sizeof(T));  
 if(!array)throw std::overflow\_error("stack overflow");  
 capacity\*=2;  
 }  
 array[size++]=num;  
 }  
 T pop(){  
 if(size==0)throw std::out\_of\_range("stack is empty");  
 return array[--size];  
 }  
 bool isEmpty(){ return size==0;}  
};  
  
  
#endif //STACK\_H