

《数据结构》

实

验

报

告

|  |  |
| --- | --- |
| 项目名称： | 二叉树的实验  二叉树的主要遍历算法 |
| 班 级： |  |
| 学生姓名**:** |  |
| 学生学号： |  |
| 指导教师： |  |
| 成 绩： |  |

20年月 日

成都信息工程大学计算机学院

1. 实现功能描述

1.编程实现前、中、后序的递归与非递归算法（共六个算法）。

特别要求：设计并实现构造二叉树链式存储的算法。

2.进一步提供如下算法：

（1）设计与实现层序遍历的递归与非递归算法；

（2）提供另一种构造二叉树链式存储的算法；

（3）提供另外一种后序非递归遍历的实现算法。

2. 方案比较与选择

1. 从数据结构的逻辑结构与存储结构角度提供多种解决方案；
2. 从时空效率角度分析决定最终采用方案的原因。

二叉树存储有链式和顺序

链式存储可读性更好，空间效率更好

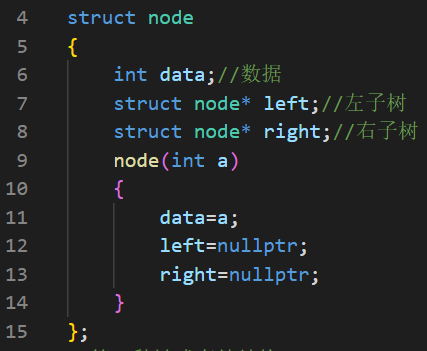
顺序存储更好实现，时间效率更好

我选择链式存储

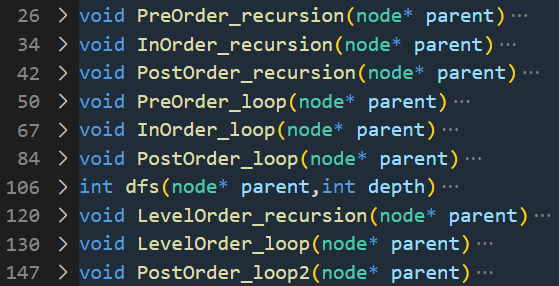
递归用系统的函数栈来实现，非递归自己申请栈或者队列来实现

3. 设计算法描述

1. 用简单示例结合所设计算法采用的数据逻辑结构图、存储结构图说明算法思想。



1. 进行模块划分，给出主要功能组成框图。形式如下：



# 4. 算法实现（即完整源程序，带注解）

#include<bits/stdc++.h>

using namespace std;

//第一种链式存储结构

struct node

{

int data;//数据

struct node\* left;//左子树

struct node\* right;//右子树

node(int a)

{

data=a;

left=nullptr;

right=nullptr;

}

};

//第二种链式存储结构

struct node21//节点

{

int data;//数据

struct node\* parent;//父母节点

};

struct node2//存储叶子节点

{

vector<node21\*> v;

};

void PreOrder\_recursion(node\* parent)

{

if(parent==nullptr)

return;

cout<<parent->data<<" ";

PreOrder\_recursion(parent->left);

PreOrder\_recursion(parent->right);

}

void InOrder\_recursion(node\* parent)

{

if(parent==nullptr)

return;

InOrder\_recursion(parent->left);

cout<<parent->data<<" ";

InOrder\_recursion(parent->right);

}

void PostOrder\_recursion(node\* parent)

{

if(parent==nullptr)

return;

PostOrder\_recursion(parent->left);

PostOrder\_recursion(parent->right);

cout<<parent->data<<" ";

}

void PreOrder\_loop(node\* parent)

{

stack<node\*> sta;

node\* p;

if(parent!=nullptr)

sta.push(parent);

while(!sta.empty())

{

p=sta.top();

sta.pop();

cout<<p->data<<" ";

if(p->right!=nullptr)

sta.push(p->right);

if(p->left!=nullptr)

sta.push(p->left);

}

}

void InOrder\_loop(node\* parent)

{

stack<node\*> sta;

node\* p=parent;

while(!sta.empty() || p!=nullptr)

{

while(p!=nullptr)

{

sta.push(p);

p=p->left;

}

p=sta.top();

sta.pop();

cout<<p->data<<" ";

p=p->right;

}

}

void PostOrder\_loop(node\* parent)

{

stack<node\*> sta1,sta2;

node\* p;

if(parent!=nullptr)

sta1.push(parent);

while(!sta1.empty())

{

p=sta1.top();

sta1.pop();

sta2.push(p);

if(p->left!=nullptr)

sta1.push(p->left);

if(p->right!=nullptr)

sta1.push(p->right);

}

while(!sta2.empty())

{

cout<<sta2.top()->data<<" ";

sta2.pop();

}

}

int dfs(node\* parent,int depth)

{

if(depth==0)

{

cout<<parent->data<<" ";

return 1;

}

int res=0;

if(parent->left!=nullptr)

res+=dfs(parent->left,depth-1);

if(parent->right!=nullptr)

res+=dfs(parent->right,depth-1);

return res;

}

void LevelOrder\_recursion(node\* parent)

{

if(parent==nullptr)

return;

for(int i=0;1;++i)

{

if(!dfs(parent,i))

break;

}

}

void LevelOrder\_loop(node\* parent)

{

queue<node\*> q;

node\* p;

if(parent!=nullptr)

q.push(parent);

while(!q.empty())

{

p=q.front();

q.pop();

cout<<p->data<<" ";

if(p->left!=nullptr)

q.push(p->left);

if(p->right!=nullptr)

q.push(p->right);

}

}

void PostOrder\_loop2(node\* parent)

{

stack<node\*> s;

node \*p,\*pre=nullptr;

if(parent!=nullptr)

s.push(parent);

while(!s.empty())

{

p=s.top();

if((p->left==nullptr && p->right==nullptr) || pre!=nullptr && (pre==p->left || pre==p->right))

{

cout<<p->data<<" ";

s.pop();

pre=p;

}

else

{

if(p->right!=nullptr)

s.push(p->right);

if(p->left!=nullptr)

s.push(p->left);

}

}

}

int main()

{

std::ios::sync\_with\_stdio(0);

std::cin.tie(0);

std::cout.tie(0);

node\* n1 = new node(1);

node\* n2 = new node(2);

node\* n3 = new node(3);

node\* n4 = new node(4);

node\* n5 = new node(5);

n1->left = n2;

n1->right = n3;

n2->left = n4;

n2->right = n5;

PreOrder\_recursion(n1);

cout << endl;

InOrder\_recursion(n1);

cout << endl;

PostOrder\_recursion(n1);

cout << endl;

PreOrder\_loop(n1);

cout << endl;

InOrder\_loop(n1);

cout << endl;

PostOrder\_loop(n1);

cout << endl;

LevelOrder\_recursion(n1);

cout << endl;

LevelOrder\_loop(n1);

cout << endl;

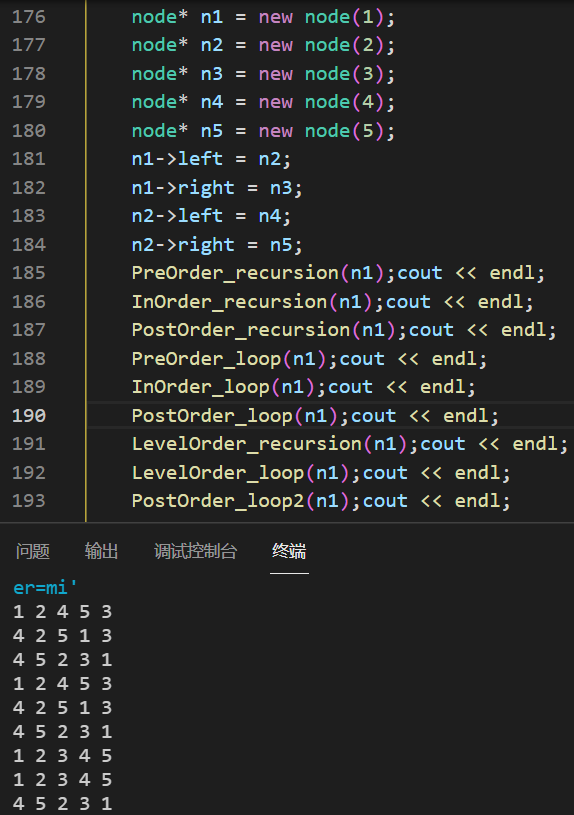
PostOrder\_loop2(n1);

cout << endl;

return 0;

}

5. 实验结果测试与分析

1. 用各种可能数据测试程序，取截图；
2. 对结果进行分析，说明算法的有效性。

# 6. 思考及学习心得（得分与文字数量和描述真实性相关）

（1）描述实验过程中对此部分知识的认识；

前中后序递归遍历，层序非递归遍历，我原来就会的。把递归转换成迭代，我花了一些时间在纸上模拟，也写出来了。把迭代写成递归，我一开始只想到暴力，想了好久还是没什么好的优化，所以还是写了暴力。

（2）特别描述在学习方法上的收获及体会；

写后序遍历用了两个栈，加深了对栈先进后出的理解

（3）针对前面的思考题内容选择部分在此回答。

1.尝试在顺序存储结构上设计这些遍历算法，并在时空效率上与链式存储进行分析对比，并得出结论（仅是分析得出结论即可，可以不实现）。

时间效率：顺序结构更快

空间效率：树的结点数num和树的深度h满足2^h接近num时，顺序结构更好，因为没有指针要存，反之链式结构更好，因为顺序结构存在大量冗余

2.通过查阅资料整理各种后序非递归遍历的实现方法，大体描述这些方法的实现思想，并给出性能对比分析。

第一种: 对于任一结点P，将其入栈，然后沿其左子树一直往下搜索，直到搜索到没有左孩子的结点，此时该结点出现在栈顶，但是此时不能将其出栈并访问，因此其右孩子还为被访问。所以接下来按照相同的规则对其右子树进行相同的处理，当访问完其右孩子时，该结点又出现在栈顶，此时可以将其出栈并访问。这样就保证了正确的访问顺序。可以看出，在这个过程中，每个结点都两次出现在栈顶，只有在第二次出现在栈顶时，才能访问它。因此需要多设置一个变量标识该结点是否是第一次出现在栈顶。

第二种：要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点P，先将其入栈。如果P不存在左孩子和右孩子，则可以直接访问它；或者P存在左孩子或者右孩子，但是其左孩子和右孩子都已被访问过了，则同样可以直接访问该结点。若非上述两种情况，则将P的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子前面被访问，左孩子和右孩子都在根结点前面被访问。

第三种：

前序遍历：根-左子树-右子树

后序遍历：左子树-右子树-根 把前序遍历倒过来：右子树-左子树-根 ！左右子树相反，不能直接倒！

先左子树入栈，在右子树入栈 (和前序遍历相反，也就是颠倒了左右子树->根-右子树-左子树），最后输出颠倒一下即可！

时间复杂度一样，常数不一样，要比较，得写程序出测试数据。

包括，样子像单链的二叉树，满二叉树，普通二叉树，以及一些特定树（可以把时间卡到最差）。数据还得特别大，这样才能体现差距。

然后跑一下这些算法，才能知道他们的算法时间优劣。

至于空间，通常是足够的，所以不考虑。

由于我比较懒，就不弄了。QAQ

只需加上以下几句就可以知道运行时间。

这是c++<ctime>里的函数

clock\_t starttime,endtime;

starttime=clock();

程序主体

endtime=clock();

cout << "The run time is: " <<(double)(endtime - starttime) / CLOCKS\_PER\_SEC << "s" << endl;