

# 二叉树的非递归遍历

## 二叉树的非递归遍历


### 二叉树的非递归遍历

二叉树是一种非常重要的数据结构，很多其它数据结构都是基于二叉树的基础演变而来的。对于二叉树，有前序、中序以及后序三种遍历方法。因为树的定义本身就是递归定义，因此采用递归的方法去实现树的三种遍历不仅容易理解而且代码很简洁。而对于树的遍历若采用非递归的方法，就要采用栈去模拟实现。在三种遍历中，前序和中序遍历的非递归算法都很容易实现，非递归后序遍历实现起来相对来说要难一点。


#### 一.前序遍历

前序遍历按照“根结点-左孩子-右孩子”的顺序进行访问。

##### 1.递归实现



```
void preOrder1(BinTree *root)    //递归前序遍历
{
    if(root!=NULL)
    {
        cout<<root->data<<" ";
        preOrder1(root->lchild);
        preOrder1(root->rchild);
    }
}
```



##### 2.非递归实现

根据前序遍历访问的顺序，优先访问根结点，然后再分别访问左孩子和右孩子。即对于任一结点，其可看做是根结点，因此可以直接访问，访问完之后，若其左孩子不为空，按相同规则访问它的左子树；当访问其左子树时，再访问它的右子树。因此其处理过程如下：

对于任一结点P：

1)访问结点P，并将结点P入栈；

2)判断结点P的左孩子是否为空，若为空，则取栈顶结点并进行出栈操作，并将栈顶结点的右孩子置为当前的结点P，循环至1)；若不为空，则将P的左孩子置为当前的结点P；

3)直到P为NULL并且栈为空，则遍历结束。



```
void preOrder2(BinTree *root)    //非递归前序遍历
{
    stack<BinTree*> s;
    BinTree *p=root;
    while(p!=NULL||!s.empty())
    {
        while(p!=NULL)
        {
            cout<<p->data<<" ";
            s.push(p);
            p=p->lchild;
        }
        if(!s.empty())
        {
            p=s.top();
            s.pop();
            p=p->rchild;
        }
    }
}
```



## 二.中序遍历

中序遍历按照“左孩子-根结点-右孩子”的顺序进行访问。

### 1.递归实现



```
void inOrder1(BinTree *root)    //递归中序遍历
{
    if(root!=NULL)
    {
        inOrder1(root->lchild);
        cout<<root->data<<" ";
        inOrder1(root->rchild);
    }
}
```



### 2.非递归实现

根据中序遍历的顺序，对于任一结点，优先访问其左孩子，而左孩子结点又可以看做一根结点，然后继续访问其左孩子结点，直到遇到左孩子结点为空的结点才进行访问，然后按相同的规则访问其右子树。因此其处理过程如下：

对于任一结点P，

- 1)若其左孩子不为空，则将P入栈并将P的左孩子置为当前的P，然后对当前结点P再进行相同的处理；
- 2)若其左孩子为空，则取栈顶元素并进行出栈操作，访问该栈顶结点，然后将当前的P置为栈顶结点的右孩子；
- 3)直到P为NULL并且栈为空则遍历结束



```
void inOrder2(BinTree *root)    //非递归中序遍历
{
    stack<BinTree*> s;
    BinTree *p=root;
    while(p!=NULL||!s.empty())
    {
        while(p!=NULL)
        {
            s.push(p);
            p=p->lchild;
        }
        if(!s.empty())
        {
            p=s.top();
            cout<<p->data<<" ";
            s.pop();
            p=p->rchild;
        }
    }
}
```



### 三.后序遍历

后序遍历按照“左孩子-右孩子-根结点”的顺序进行访问。

#### 1.递归实现



```
void postOrder1(BinTree *root)    //递归后序遍历
{
    if(root!=NULL)
    {
        postOrder1(root->lchild);
        postOrder1(root->rchild);
        cout<<root->data<<" ";
    }
}
```



#### 2.非递归实现

后序遍历的非递归实现是三种遍历方式中最难的一种。因为在后序遍历中，要保证左孩子和右孩子都已被访问并且左孩子在右孩子前访问才能访问根结点，这就为流程的控制带来了难题。下面介绍两种思路。

第一种思路：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，因此其右孩子还未被访问。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。



```
void postOrder2(BinTree *root)    //非递归后序遍历
{
    stack<BTNode*> s;
    BinTree *p=root;
    BTNode *temp;
    while(p!=NULL||!s.empty())
    {
        while(p!=NULL)            //沿左子树一直往下搜索，直至出现没有左子树的结点
        {
            BTNode *btn=(BTNode *)malloc(sizeof(BTNode));
            btn->btnode=p;
            btn->isFirst=true;
            s.push(btn);
            p=p->lchild;
        }
        if(!s.empty())
        {
            temp=s.top();
            s.pop();
            if(temp->isFirst==true)    //表示是第一次出现在栈顶
            {
                temp->isFirst=false;
                s.push(temp);
                p=temp->btnode->rchild;
            }
            else                        //第二次出现在栈顶
            {
                cout<<temp->btnode->data<<" ";
                p=NULL;
            }
        }
    }
}
```




第二种思路：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。



```
void postOrder3(BinTree *root)    //非递归后序遍历
{
    stack<BinTree*> s;
    BinTree *cur;                //当前结点
    BinTree *pre=NULL;           //前一次访问的结点
    s.push(root);
    while(!s.empty())
    {
        cur=s.top();
        if((cur->lchild==NULL&&cur->rchild==NULL)||
            (pre!=NULL&&(pre==cur->lchild||pre==cur->rchild)))
        {
            cout<<cur->data<<" "; //如果当前结点没有孩子结点或者孩子节点都已被访问过
            s.pop();
            pre=cur;
        }
        else
        {
            if(cur->rchild!=NULL)
                s.push(cur->rchild);
            if(cur->lchild!=NULL)
                s.push(cur->lchild);
        }
    }
}
```

#### 四.整个程序完整的代码



```
/*二叉树的遍历* 2011.8.25*/

#include <iostream>
#include<string.h>
#include<stack>
using namespace std;

typedef struct node
{
    char data;
    struct node *lchild,*rchild;
}BinTree;

typedef struct node1
{
    BinTree *bnode;
    bool isFirst;
}BTNode;
```

```

void creatBinTree(char *s, BinTree *&root) //创建二叉树, s为形如A(B,C(D,E))形式的字符串
{
    int i;
    bool isRight=false;
    stack<BinTree*> s1; //存放结点
    stack<char> s2; //存放分隔符
    BinTree *p,*temp;
    root->data=s[0];
    root->lchild=NULL;
    root->rchild=NULL;
    s1.push(root);
    i=1;
    while(i<strlen(s))
    {
        if(s[i]=='(')
        {
            s2.push(s[i]);
            isRight=false;
        }
        else if(s[i]==',')
        {
            isRight=true;
        }
        else if(s[i]==')')
        {
            s1.pop();
            s2.pop();
        }
        else if(isalpha(s[i]))
        {
            p=(BinTree *)malloc(sizeof(BinTree));
            p->data=s[i];
            p->lchild=NULL;
            p->rchild=NULL;
            temp=s1.top();
            if(isRight==true)
            {
                temp->rchild=p;
                cout<<temp->data<<"的右孩子是"<<s[i]<<endl;
            }
            else
            {
                temp->lchild=p;
                cout<<temp->data<<"的左孩子是"<<s[i]<<endl;
            }
            if(s[i+1]=='(')
                s1.push(p);
        }
        i++;
    }
}

```

```

void display(BinTree *root)           //显示树形结构
{
    if (root != NULL)
    {
        cout << root->data;
        if (root->lchild != NULL)
        {
            cout << ' (';
            display (root->lchild);
        }
        if (root->rchild != NULL)
        {
            cout << ', ';
            display (root->rchild);
            cout << ') ';
        }
    }
}

void preOrder1 (BinTree *root)        //递归前序遍历
{
    if (root != NULL)
    {
        cout << root->data << " ";
        preOrder1 (root->lchild);
        preOrder1 (root->rchild);
    }
}

void inOrder1 (BinTree *root)         //递归中序遍历
{
    if (root != NULL)
    {
        inOrder1 (root->lchild);
        cout << root->data << " ";
        inOrder1 (root->rchild);
    }
}

void postOrder1 (BinTree *root)       //递归后序遍历
{
    if (root != NULL)
    {
        postOrder1 (root->lchild);
        postOrder1 (root->rchild);
        cout << root->data << " ";
    }
}

void preOrder2 (BinTree *root)        //非递归前序遍历
{
    stack<BinTree*> s;
    BinTree *p = root;
    while (p != NULL || !s.empty())

```

```

{
    while(p!=NULL)
    {
        cout<<p->data<<" ";
        s.push(p);
        p=p->lchild;
    }
    if(!s.empty())
    {
        p=s.top();
        s.pop();
        p=p->rchild;
    }
}
}

void inOrder2(BinTree *root)          //非递归中序遍历
{
    stack<BinTree*> s;
    BinTree *p=root;
    while(p!=NULL||!s.empty())
    {
        while(p!=NULL)
        {
            s.push(p);
            p=p->lchild;
        }
        if(!s.empty())
        {
            p=s.top();
            cout<<p->data<<" ";
            s.pop();
            p=p->rchild;
        }
    }
}

void postOrder2(BinTree *root)        //非递归后序遍历
{
    stack<BTNode*> s;
    BinTree *p=root;
    BTNode *temp;
    while(p!=NULL||!s.empty())
    {
        while(p!=NULL)                //沿左子树一直往下搜索，直至出现没有左子树的结点
        {
            BTNode *btn=(BTNode *)malloc(sizeof(BTNode));
            btn->btnode=p;
            btn->isFirst=true;
            s.push(btn);
            p=p->lchild;
        }
        if(!s.empty())
        {

```



```

        temp=s.top();
        s.pop();
        if(temp->isFirst==true)           //表示是第一次出现在栈顶
        {
            temp->isFirst=false;
            s.push(temp);
            p=temp->btnode->rchild;
        }
        else                               //第二次出现在栈顶
        {
            cout<<temp->btnode->data<<" ";
            p=NULL;
        }
    }
}

void postOrder3(BinTree *root)           //非递归后序遍历
{
    stack<BinTree*> s;
    BinTree *cur;                        //当前结点
    BinTree *pre=NULL;                   //前一次访问的结点
    s.push(root);
    while(!s.empty())
    {
        cur=s.top();
        if((cur->lchild==NULL&&cur->rchild==NULL)||
            (pre!=NULL&&(pre==cur->lchild||pre==cur->rchild)))
        {
            cout<<cur->data<<" "; //如果当前结点没有孩子结点或者孩子节点都已被访问过
            s.pop();
            pre=cur;
        }
        else
        {
            if(cur->rchild!=NULL)
                s.push(cur->rchild);
            if(cur->lchild!=NULL)
                s.push(cur->lchild);
        }
    }
}

int main(int argc, char *argv[])
{
    char s[100];
    while(scanf("%s",s)==1)
    {
        BinTree *root=(BinTree *)malloc(sizeof(BinTree));
        creatBinTree(s,root);
        display(root);
        cout<<endl;
        preOrder2(root);
    }
}

```

```
        cout<<endl;
        inOrder2(root);
        cout<<endl;
        postOrder2(root);
        cout<<endl;
        postOrder3(root);
        cout<<endl;
    }
    return 0;
}
```



作者: [海子](#)

出处: <http://www.cnblogs.com/dolphin0520/>

本博客中未标明转载的文章归作者[海子](#)和博客园共有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。

« 上一篇: [KMP算法](#)

» 下一篇: [Dijkstra算法（单源最短路径）](#)

posted @ 2011-08-25 20:12 海子 阅读(88206) 评论(...) 编辑 收藏