题目：

建立以左右孩子链接结构表示的二叉树，实现二叉树的先序、中序、后序的递归和非递归方式遍历，分层遍历、统计树的高度。

代码：

#include<stdio.h>

#include<stdlib.h>

#include<malloc.h>

#define MAX 30

\_CRT\_SECURE\_NO\_WARNINGS;

typedef struct node

{

char c;

struct node\* l;

struct node\* r;

}node;

void\* precreat(node\*\*, FILE\*);//前序建立二叉树

void\* pre1(node\*, FILE\*);//递归前序

void\* pre2(node\*, FILE\*);//非递归前序

void\* mid1(node\*, FILE\*);//递归中序

void\* mid2(node\*, FILE\*);//非递归中序

void\* beh1(node\*, FILE\*);//递归后序

void\* beh2(node\*, FILE\*);//非递归后序

void level(node\*, FILE\*);//层次遍历

int Depth(node\*);//计算树的高度

void\* free1(node\*);//释放空间

int main()

{

node\* T;

T = (node\*)malloc(sizeof(node));

FILE\* fp = fopen("data.txt", "r+");

precreat(&T, fp);

fclose(fp);

if (T == NULL)

{

printf("error\n");

exit(-1);

}

FILE\* fp1 = fopen("result.txt", "w+");

fprintf(fp1, "递归前序\n");

pre1(T, fp1);

fprintf(fp1, "\n非递归前序\n");

pre2(T, fp1);

fprintf(fp1, "\n递归中序\n");

mid1(T, fp1);

fprintf(fp1, "\n非递归中序\n");

mid2(T, fp1);

fprintf(fp1, "\n递归后序\n");

beh1(T, fp1);

fprintf(fp1, "\n非递归后序\n");

beh2(T, fp1);

fprintf(fp1, "\n层次遍历\n");

level(T, fp1);

int h = Depth(T) - 1;

fprintf(fp1,"\n树的高度为 %d ", h);

fclose(fp1);

free1(T);//释放内存

return 0;

}

//前序建立二叉树

void\* precreat(node\*\* pT, FILE\* fp)

{

char c = 1;

if (feof(fp) != 1)

{

fscanf(fp, "%c ", &c);

if (c == '#')

{

\*pT = (node\*)malloc(sizeof(node));

(\*pT)->c = c;

(\*pT)->l = NULL;

(\*pT)->r = NULL;

return;

}

\*pT = (node\*)malloc(sizeof(node));

if (\*pT == NULL)

exit(-1);

(\*pT)->c = c;

precreat(&(\*pT)->l, fp);

precreat(&(\*pT)->r, fp);

}

else

\*pT = NULL;

}

//递归前序

void\* pre1(node\* T, FILE\* fp)

{

if (T == NULL)

return;

if(T->c != '#')

fprintf(fp, "%c ", T->c);

pre1(T->l, fp);

pre1(T->r, fp);

}

//非递归前序

void\* pre2(node\* T, FILE\* fp)

{

node\* p;

p = T;

node\* n[MAX] = { 1 };

int top;

top = 0;

do

{

while (p != NULL)

{

if (p->c != '#')

fprintf(fp, "%c ", p->c);

n[top] = p;

++top;

p = p->l;

}

if (top > 0)

{

--top;

p = n[top];

p = p->r;

}

} while (top > 0 || p != NULL);

}

//递归中序

void\* mid1(node\* T, FILE\* fp)

{

if (T == NULL)

return;

mid1(T->l, fp);

if (T->c != '#')

fprintf(fp, "%c ", T->c);

mid1(T->r, fp);

}

//非递归中序

void\* mid2(node\* T, FILE\* fp)

{

node\* s[MAX] = { 1 };

int i = 0;

node\* p = T;

do

{

while (p != NULL)

{

++i;

s[i] = p;

p = p->l;

}

if (i != 0)

{

p = s[i];

--i;

if (p->c != '#')

fprintf(fp, "%c ", p->c);

p = p->r;

}

} while ((i != 0) || (p != NULL));

}

//递归后序

void\* beh1(node\* T, FILE\* fp)

{

if (T == NULL)

return;

beh1(T->l, fp);

beh1(T->r, fp);

if (T->c != '#')

fprintf(fp, "%c ", T->c);

}

//非递归后序

void\* beh2(node\* T, FILE\* fp)

{

typedef struct stack

{

node\* p;

int tag;

}STACK;

STACK s[MAX] = { 1 };

node\* p = T;

int top = 0;

int tag = 0;

do

{

if (p != NULL)

{

tag = 0;

s[top].p = p;

s[top].tag = tag;

++top;

p = p->l;

}

else

{

if (top == 0)

break;

--top;

p = s[top].p;

tag = s[top].tag;

if (tag == 0)

{

tag = 1;

s[top].p = p;

s[top].tag = tag;

++top;

p = p->r;

}

else

{

if (p->c != '#')

fprintf(fp, "%c ", p->c);

p = NULL;

}

}

} while (top != 0);

}

//层次遍历

void level(node\* T, FILE\* fp)

{

node\* a[MAX] = { 1 };//队列

node\* p = T;

int front = 0;

int rear = 0;

a[rear] = T;

++rear;

while (front < rear)

{

p = a[front];//队头结点出队

++front;

if (p->c != '#')

fprintf(fp, "%c ", p->c);

if (p->l != NULL)//左孩子入队

{

a[rear] = p->l;

++rear;

}

if (p->r != NULL)//右孩子入队

{

a[rear] = p->r;

++rear;

}

}

}

//计算树的高度

int Depth(node\* T)

{

int m = 0;

int n = 0;

if (T == NULL)

return 0; //空树深度为0

else

{

m = Depth(T->l); //左子树深度

n = Depth(T->r); //右子树深度

}

if (m > n)

return (m + 1);

else

return (n + 1);

}

//释放空间

void\* free1(node\* T)

{

if (T != NULL)

{

free1(T->l);

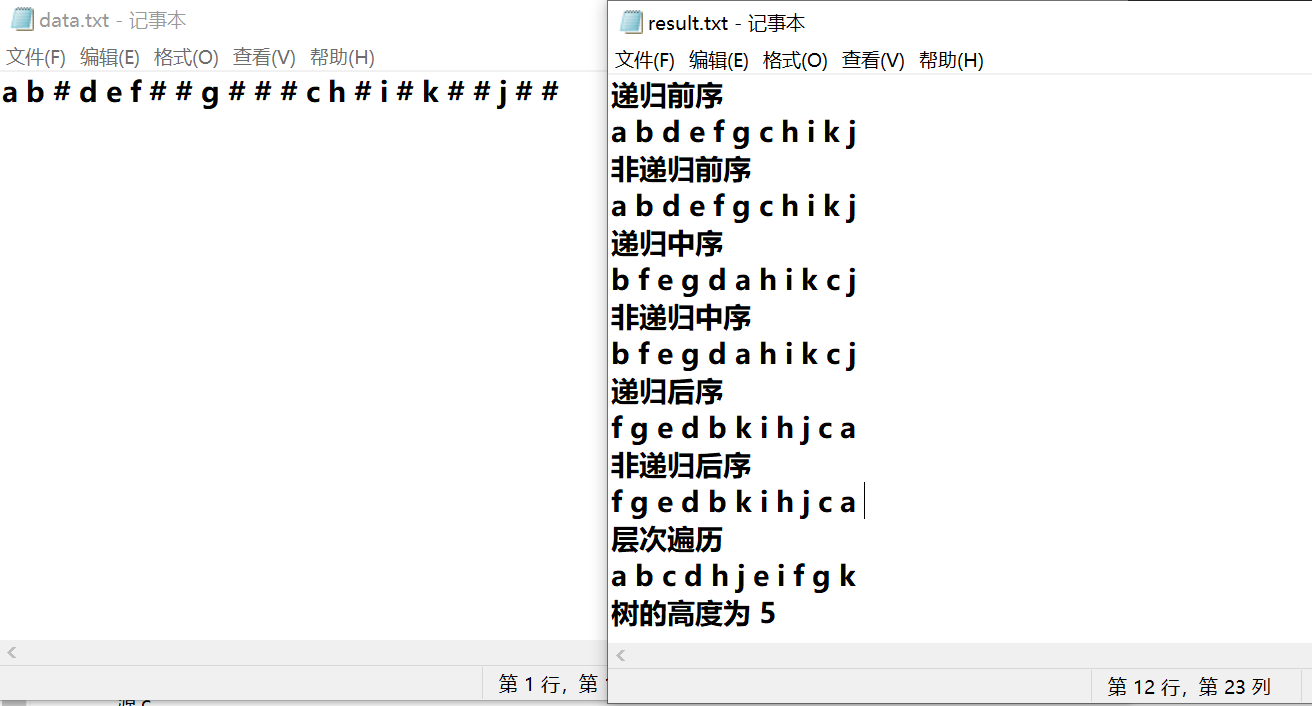
free1(T->r);

free(T);

}

}

运行结果：



总结：先序遍历：创建一个栈对象，根结点入栈。当栈为非空时，将栈顶结点弹出栈内并访问该结点。对当前访问结点的非空左孩子结点相继依次访问，并将当前访问结点的非空右孩子结点压入栈内。重复执行步骤2和3，直到栈为空为止。

中序遍历可以借助一个栈来记录遍历过程中所经历的而未被访问过的所有结点，以便遍历完一个结点左子树后能顺利返回到它的父结点。实现中序遍历操作的非递归算法的主要思想是:从二叉树的根结点出发，沿着该结点的左子树向下搜索，在搜索过程中将所遇到的每一个结点依次压栈，直到二叉树中最左下的结点压栈为止，然后从栈中弹出栈顶结点并对其进行访问，访问完后再进入该结点的右子树并用上述的同样的方法去遍历该结点的右子树，以此类推，直到二叉树中的所有结点都被访问为止。

后序遍历:

1. 初始化两个栈，一个用于保存中间遍历过程的栈称为s栈，一个用于保存最终结果的栈称为output栈。
2. push根结点到第一个栈s中。
3. 从第一个栈s中pop出一结点，并将其push到第二个栈output中。
4. 将第一个栈s中pop出的结点的孩子结点，按左孩子、右孩子的顺序push到第一个栈s中。
5. 重复步骤3和4，直到栈s为空。
6. 栈s为空时，所有结点都已push到栈output中，且按后序遍历顺序存放，依次将栈output的结点pop出并进行访问，即为二叉树后序遍历的结果。