**二叉树的创建、递归遍历、抽象数据类型方法实现**

#include <iostream>

#include<malloc.h>

#include<assert.h>

using namespace std;

#define ElemType char

typedef struct BinTreeNode {

ElemType data;

struct BinTreeNode \*leftChild;

struct BinTreeNode \*rightChild;

}BinTreeNode;

typedef struct BinTree {

BinTreeNode \*root; //二叉树的根 用结点类型的指针表示

ElemType refvalue; //stop flag

}BinTree;

//定义二叉树 实际上是一个递归的概念

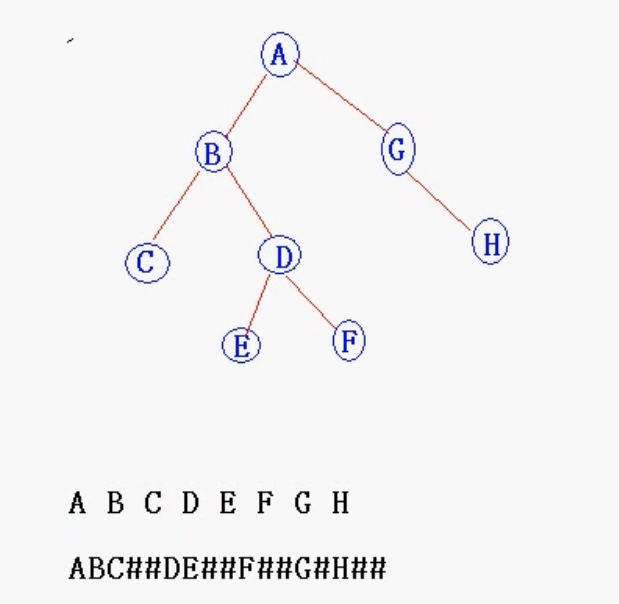
void InitBinTree(BinTree &bt,ElemType ref) {

bt.root = nullptr;

bt.refvalue = ref;

}

**一、二叉树的创建**



//ABC##DE##F##G#H##

void CreatBinTree\_1(BinTree&bt, BinTreeNode\*& t) { //重载函数 要采用引用的方式传递根结点

ElemType Item;

cin >> Item;

if (Item == bt.refvalue) {

t = nullptr;

}

else {

t = (BinTreeNode\*)malloc(sizeof(BinTreeNode));

assert(t != nullptr);

t->data = Item;

CreatBinTree\_1(bt, t->leftChild);

CreatBinTree\_1(bt, t->rightChild);

}

}

void CreatBinTree\_1(BinTree& bt) { //明面上的函数

CreatBinTree\_1(bt,bt.root); //内部函数（真实实现二叉树创建的函数）

}

// //方法二 实际上是对上述方法的升级 不用我们自己输入数据

//void CreatBinTree\_4(BinTree& bt, BinTreeNode\*& t, const char\* &str) { //此时的\*str 其实是第一个字符 //注意这里一定要用引用？

// if (\*str== bt.refvalue) {

// t = NULL;

// }

// else {

// t= (BinTreeNode\*)malloc(sizeof(BinTreeNode));

// assert(t != nullptr);

// t->data = \*str;

// CreatBinTree\_4(bt, t->leftChild, ++str);

// CreatBinTree\_4(bt, t->rightChild, ++str);

// }

//}

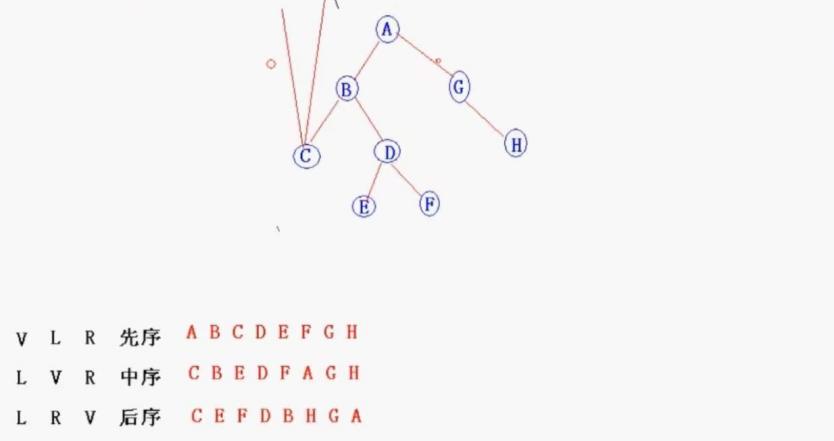
//

//void CreatBinTree\_4(BinTree& bt, const char\* str) {

// CreatBinTree\_4(bt, bt.root, str);

//}

**二、二叉树的遍历**



//先序

void PreOrder(BinTreeNode\*& t) {

if (t != nullptr) {

cout << t->data;

PreOrder(t->leftChild); //先遍历左树

PreOrder(t->rightChild); //后遍历右树

}

}

void PreOrder(BinTree& bt) {

PreOrder(bt.root);

}

//中序

void InOrder(BinTreeNode\*& t) {

if (t != nullptr) {

InOrder(t->leftChild);

cout << t->data;

InOrder(t->rightChild);

}

}

void InOrder(BinTree& bt) {

InOrder(bt.root);

}

//后序

void PostOrder(BinTreeNode\*& t) {

if (t != nullptr) {

PostOrder(t->leftChild);

PostOrder(t->rightChild);

cout << t->data;

}

}

void PostOrder(BinTree& bt) {

PostOrder(bt.root);

}

//层次遍历(一层一层，每一层从左到右）

//借助队列，先将根结点入队，队不空出队，再将其左右结点（左右结点不为空的情况下）入队，以此类推

#define ElemType BinTreeNode\*

typedef struct QueueNode {

ElemType data;

struct QueueNode\* next;

}QueueNode;

typedef struct LinkQueue {

QueueNode\* front;

QueueNode\* tail;

}LinkQueue;

void InitQueue(LinkQueue& Q) ;

void EnQueue(LinkQueue& Q, ElemType x) ;

void DeQueue(LinkQueue& Q) ;

void GetHead(LinkQueue& Q, ElemType& v);

bool QueueIsEmpty(LinkQueue& Q) ;

void LevelOrder(BinTreeNode\* t) {

if (t != NULL) {

BinTreeNode \*v;

LinkQueue Q;

InitQueue(Q);

EnQueue(Q, t);

while (!QueueIsEmpty(Q)) {

GetHead(Q, v);

DeQueue(&Q);

cout << v->data;

if (v->leftChild != NULL) {

EnQueue(&Q, t->leftChild);

}

if (v->rightChild != NULL) {

EnQueue(&Q, t->rightChild);

}

}

}

}

void LevelOrder(BinTree\* bt) {

LevelOrder(bt->root);

}

1. **二叉树的方法实现**

//求二叉树的结点个数

int Size(BinTreeNode\* t) {

if (t ==nullptr) {

return 0;

}

else {

return Size(t->leftChild) + Size(t->rightChild) + 1; //递归方法实现

}

}

int Size(BinTree&bt){

return Size(bt.root);

}

//求二叉树的层数（高度）

int Height(BinTreeNode\* t) {

if (t == nullptr) {

return 0;

}

else {

int left\_height = Height(t->leftChild);

int right\_height = Height(t->rightChild);

return (left\_height > right\_height ? left\_height : right\_height) + 1;

}

}

int Height(BinTree &bt){

return Height(bt.root);

}

//查找二叉树的结点

BinTreeNode\* Search(BinTreeNode\* t, ElemType key) {

if (t == nullptr) {

return NULL;

}

if (t->data == key) {

return t;

}

BinTreeNode\* p = Search(t->leftChild, key);

if (p != nullptr) {

return p; //先查找左树，如果找到了，直接返回，不用再查找右树了

}

return Search(t->rightChild, key); //如果不在左树并且存在 肯定在右树 直接返回即可

}

BinTreeNode\* Search(BinTree bt, ElemType key) { //返回值为结点的地址

return Search(bt.root,key);

}

//查找父结点

BinTreeNode\* Parent(BinTreeNode\* t, BinTreeNode\* p) {

if (t == NULL || p == NULL) { //先讨论特殊情况1

return NULL;

}

if (t->leftChild == p || t->rightChild == p) { //先讨论特殊情况2

return t;

}

BinTreeNode\* q = Parent(t->leftChild, p);

if (q != nullptr) {

return q;

}

return Parent(t->rightChild, p);

}

BinTreeNode\* Parent(BinTree bt, BinTreeNode\* p) {

return Parent(bt.root, p);

}

//查找左子树

BinTreeNode\*LeftChild(BinTreeNode\* p){

if (p != nullptr) {

return p->leftChild;

}

return nullptr;

}

//查找右子树

BinTreeNode\* RightChild(BinTreeNode\* p) {

if (p != nullptr) {

return p->rightChild;

}

return nullptr;

}

//判空

bool BinTreeEmpty(BinTree bt) {

return bt.root == nullptr; //判空 直接返回为空的条件即可

}

//拷贝构造二叉树

void Copy(BinTreeNode\*&t1, BinTreeNode\*t2) {

if (t2 == nullptr)

t1 = nullptr;

else {

t1 = (BinTreeNode\*)malloc(sizeof(BinTreeNode));

assert(t1 != nullptr);

t1->data = t2->data;

Copy(t1->leftChild, t2->leftChild);

Copy(t1->rightChild, t2->rightChild);

}

}

void Copy(BinTree& bt1, BinTree bt2) {

return Copy(bt1.root,bt2.root);

}

//清除二叉树

void BinTreeClear(BinTreeNode\*t) { //注意千万不能先释放根结点 一旦释放就再也找不到左树和右树了

if (t != nullptr) {

BinTreeClear(t->leftChild);

BinTreeClear(t->rightChild);

free(t);

t = nullptr;

}

}

void BinTreeClear(BinTree &bt) {

BinTreeClear(bt.root);

}

**测试函数**

void main() {

/\*const char\* str = "ABC##DE##F##G#H##";\*/

BinTree mytree;

InitBinTree(mytree,'#'); //'#'为结束标记

CreatBinTree\_1(mytree);

}