树解题大法

树中最重要的就是模板,针对特定得问题就会有特定得模板去套用。拿到树的题,不要凭感觉直接弄,除非遇到原题,否则画个题先手算找规律。

树有四种遍历:

1. 先序遍历
2. 中序遍历
3. 后序遍历
4. 层次遍历

很多学弟学妹对树得这四种遍历拿捏不住,不知道用那种遍历去解决题目所要实现得功能。今天在这里就跟大家聊聊怎么用对应得树搜索去解决特定得应用场景吧。

首先先序遍历是所有搜索中最重要得,也是最简单粗暴得树搜索方式,他得应用场景如下:

(1)不求任何要求得访问树中任意节点,然后对访问的节点进行操作

(2)求树中特定性质得节点,然后对特定性质的节点进行操作

(3)顶点得路径问题:深度(高度),顶点间得路径

我来逐条解释(1)~(3)条所示得场景下用先序遍历。

应用场景1:先序遍历是最简单直接地遍历,只要单纯地搜索树中节点可以直接使用先序遍历。

typedef struct BiTNode

{

int data;

struct BiTNode \*lchild;

struct BiTNode \*rchild;

}BiTNode, \*BiTNode;

Status preOrderFunc(BiTree t) //这个函数的功能就是对某个节点为根的树进行操作

{

if (当前访问的节点不为空) //不为空就开始执行访问,题目要求操作即进行操作

{

访问语句or操作语句;

preOrderFunc(t->lchild); //以左孩子为根的子树也是树,可以直接调用这个函数是实现操作

preOrderFunc(t->rchild); //以右孩子为根的子树也是树,可以直接调用这个函数是实现操作

}

}

注:考试的时候会将这个操作和另外的操作结合考,这个只是基础操作。有时候树是顺序存储,那么t->child改成根孩子的索引,假设根的索引号为i,左孩子索引为2\*i,右孩子为2\*i+1.

Status preOrderFunc(int arra, int tIndex)

{

if (当前访问的节点不为空,即arra[tIndex] != INF)

{

访问语句or操作语句;

preOrderFunc(arra, tIndex \* 2); //以左孩子为根的子树也是树,可以直接调用这个函数是实现操作

preOrderFunc(arra, tIndex \* 2 + 1); //以右孩子为根的子树也是树,可以直接调用这个函数是实现操作

}

}

应用场景二:先序遍历能访问树中的每个节点,所以使用先序遍历判断每个访问到的节点,使用if条件判断对各种节点进行区分,然后再按照题目要求进行操作。

Status preOrderFunc(BiTree t) //这个函数的功能就是对树中某种类型的节点进行操作

{

if (当前访问为空,则不存在子树) //

{

//对于返回节点地址场景,retur NULL;对于返回统计数据,返回return 0;

不进行任何操作;

}

else if (判断当前节点是否满足题意要求的条件) //因为if已经执行,则当前节点不可能为空

{

//查找节点地址or值,则返回地址or值;

//统计数据,则返回return 1,因为条件成立,统计符合条件的节点又增加了1;

满足要求之后进行的操作;

}

else //当前节点存,则不满足条件,则从其的左右子树继续递归查找

{

preOrderFunc(t->lchild);

preOrderFunc(t->rlchild);

}

}

具体中的例子

执行操作类型:查找树中值为x的节点,查找到之后返回其地址。

BiTNode \*preOrderFunc(BiTree t, int x)

{

if (t == NULL) //节点为空,执行空操作

{

return NULL; //进行操作

}

else if (t->data == x) //节点满足条件,执行规定操作

{

return t;

}

else //从左子树和右子树中递归查找x,将查找结果返回

{

preOrderFunc(t->lchild)==NULL ? return preOrderFunc(t->rchild) : return preOrderFunc(t->lchild);

}

}

执行统计类型:

统计树中符合某条件,或者是某种类型的节点。例如:求树中值为x的节点个数。

int preOrderFunc(BiTree t, int x)

{

if (t == NULL) //节点为空,执行空操作

{

return 0; //进行操作

}

else if (t->data == x) //节点满足条件,又有一个顶点满足条件，计数加1

{

return 1;

}

else //从左子树和右子树中递归进行统计,将两者之和返回

{

return preOrderFunc(t->lchild) + preOrderFunc(t->rchild);

}

}

应用场景三:树的高度问题是考试中最常见的问题。二叉树又三种类型的节点组成:空节点,非空非叶子节点(有孩子),叶子节点(无孩子)。有孩子的节点高度,则是其孩子的高度最大值+1,因为其是左孩子为根的子树上再加了他作为根,树中在根上再加一个根,高度必须加1。叶子节点是子树中最后一个节点了,所以他的高度只能为1。树中就存在三种节点,依次判断每种情况,就能求得树得高度。

int preOrderFunc(BiTree t)//函数就是求以某个为根得高度

{

if (NULL == t) //空节点,高度为零

{

return 0;

}

else if (NULL == t->lchild && NULL == t->rchild) //叶子节点

{

retur 1; //高度为1

}

else //当前节点有孩子,则选择左右孩子中高度最大者

{

int lchildHeight = preOrderFunc(t->lchild); //左子树也是树,求其深度可以递归调用函数

int rchildHeight = preOrderFunc(t->rchild); //右子树也是树,求其深度可以递归调用函数

//最大值的基础上加1

return lchildHeight > rchildHeight ? lchildHeight + 1 : rchildHeight + 1;

}

}

树的深度问题。先序遍历从某个顶点的左孩子和右孩子进行递归,就代表往下走了一层。加入当前的子树根节点深度为d,则访问左孩子时深度变为d+1,访问右孩子时深度变为d+1。孩子节点都在父亲节点的深度上加1,最后逐层往下传,就能求出每个节点的深度。

例如:求解值为x的节点对应在树中的深度。

int d = 0; //全局变量能使得递归求解变得简单

int preOrderFunc(BiTree t, int x, int currentDepth)

{

if (NULL == t)

{

return 0;

}

else if (t->data == x)

{

d = currentDepth;

return d;

}

else //从左右孩子为根的子树处开始查找,深度分别都在原有基础上增加1

{

preOrderFunc(t->lchild, x, currentDepth + 1);

preOrderFunc(t->rchild, x, currentDepth + 1);

}

}

中序的应用场景比较少:目前只有两种最为常见。第一种是表达式存储到树中,然后从树中读出表达式的相应题型一定时用中序遍历。例如:a+b\*c+d存到表达式树中,则用中序可以读出该表达式。当然,考试的时候他还会绑定其他条件,但是一定时用中序去解决问题。第二种就时二叉排序树的相关题,因为二叉排序树使用中序遍历就能获得一个有序的序列,所以经常会根据这一条件出题。比如求比k小的题,则用中序遍历搜索二叉排序树,然后在遇到k之前,搜到的顶点都小于k,在k之后搜到的顶点都大于k。

或者,将一棵二叉排序树分成两棵树,一棵树中所有节点都大于t,另外一棵树的所有节点都小于t。这种题设计到二叉排序树的节点插入,可以使用中序遍历搜索树,将没有访问到t之前的顶点都插入到小于t的那颗二叉排序树中,然后遇到t之后,将t的左孩子挂到小于t的那棵树中,将t的右孩子挂到大于t的那棵树对应位置。总之,看到二叉排序树的题出现大于,第k大,等跟大小有序挂钩的情景,请直接使用中序遍历操作。

后序遍历就更为简单:求算数表达式中运算结果的题,除非题目要求写后序遍历解决,基本上很难再有场景需要后序遍历。

层次遍历一般用于求顶点编号(因为顶点编号是从左到右,从上往下编号),另外完全二叉树,树的宽度都需要使用层次遍历。

平衡二叉树的题看下题库吧,就那两题可以出。

最后留下作业:知道树的层次遍历和中序遍历,怎么构建一颗二叉树 ?砸门明天晚上给出算法的代码。

最后给计算机的学弟学妹留下几个题图的题刷刷:

求解以邻接表存储的有向图每个顶点的度。

求解从v开始的所有长度为K的最短路径。提示:只要题目中出现最短二字,全部都要用BFS.

一个小学弟问得题:

数组A[1---8,-2---6,0----6]以行序为主,第一个元素首地址为78,每一个元素得长度为4,试求元素A[4,2,3]存储地址。

首先这个题大家要明白,1---8是啥意思。正常得数组索引事0~N,但是这类数组是经过特殊处理得,索引可以随意取,但是索引值必须连续。1---8代表元素的索引值为1，2，3，4，5，6，7，8

-2--6代表元素的索引值为-2，-1，0，1，2，3，4，5，6

1. --6代表元素的索引值为0,1,2,3,4,5,6

这是一个三维数组,但是我们可以类比二维数组。对于二维数组a[N][M],a[1][1]有一行满的(a[0]~a[M-1]),总共M个元素,然后再加上a[1]那行几个散的(a[1][0]~a[1][1])。考试的时候只要求高维数组一定要知道满的是多少，散的是多少。

现在我们来看到这个题,他要求A[4,2,3]。则满的是A[0]~A[3]，散的是A[4][2,3].

满的是0~3个二维数组,这个二维数组大小是9\*7,9第二维有9个元素,7是第三维有7个元素(0~6),这个地方怕漏,就写出所有的索引去挨个数。总共有4\*(7\*9)个单元。

散的是A[2,3]，二位里面-2~1都是满的，散的是a[2][0]~a[2][3].

所以总共有4\*7个存储单元。

一维散的是a[2][0]~a[2][3]，0，1，2，3，一共是4个存储单元。

所以最终从a[0][0][0]~a[4,2,3]一共有4\*7\*9 + 4 \* 7 + 4单元，包括了基地址a[0][0][0]。

所以A[1---8,-2---6,0----6]距离A[0][0][0]基地址有4\*7\*9+4\*7+4-1个存储单元。

在基地址的基础上增加这个多个字节,就是地址。

78(基地址) + (4\*7\*9+4\*7+4 - 1个存储单元)\*4(每个单元的长度) = A[4][2][3]的地址