

**课程设计报告书**

|  |  |  |  |
| --- | --- | --- | --- |
| **课堂表现及考勤30%** | **系统成绩50%** | **报告成绩20%** | **总成绩** |
|  |  |  |  |

**课 程 名： 数据结构课程设计**

**专业班级： 118030602**

**学 号： 11803060217**

**姓 名： 林海**

**联系电话： 13364033950**

**指导老师： 陈媛**

**时 间： 2021-06-15**

**计算机科学与工程学院**

**一、课程设计概述**

本次数据结构课程设计共完成三个题目：11.2.8二叉排序树与平衡二叉树的实现、11.2.10图的基本操作与实现和11.2.19滑雪场问题。

使用语言：C语言

编译环境：VisualStudioCode

**二、课程设计题目**

**【实验内容】**

11.2.8二叉排序树与平衡二叉排序树的实现

**【问题描述】**

分别使用链式存储和顺序存储完成二叉排序树的实现，并完成相应的操作，从磁盘读取文件，生成排序树，中序遍历，查找元素并删除

**【需求分析】**

经过分析，本系统需完成的主要功能如下：

1. 建立data.dat文件用于读取数列
2. 生成二叉排序树
3. 中序遍历
4. 查找节点，输出结果或者删除节点
5. 计算查找成功下的平均查找长度
6. 生成平衡二叉树
7. 换成顺序存储再实现一遍以上功能

**【概要设计】**

-=ADT=-

//中序遍历

void inorderTraverse(bitree bt, int \*num, int \*deep, int \*nodes)

//顺序存储中序遍历

void inorderTraverseGroup(int \*pobt, int \*position)

//查找节点 参数oprate代表是插入节点还是删除节点

bitree findNode(bitree bt, int value, int operation, int\* deep, int\* num, int\* obt, int \*position)

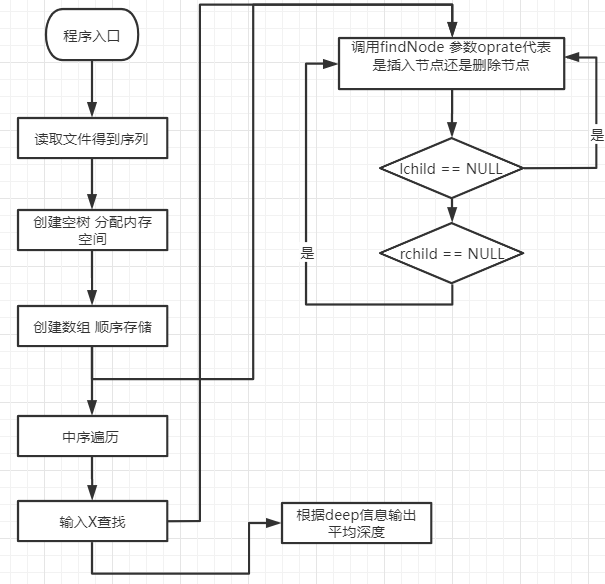
//读取文件创造数列

int \* creatIntGroup()

**【存储结构】**

链表，数组

**【流程图】**



**【详细设计】**

节点是简单的二叉树节点类型，由于题目要求随意数据域为整型，还有两个分别指向相同类型的左右孩子指针域。

//定义链表结构

typedef struct tnode{

    int data;

    struct  tnode\* lchild;

    struct  tnode\* rchild;

}\*bitree, bitnode;

//分别定义平衡因子数

#define LH +1

#define EH  0

#define RH -1

实现创建二叉排序树大部分功能的核心代码，我使用了一个通用的函数来完成各个操作，bitree findNode(bitree bt, int value, int operation, int\* deep, int\* num, int\* obt, int \*position)

下面说明每个参数的作用

Bittree bt：作为二叉树中的每个节点递归传递

Int value：数据域，涉及到查找和创建二叉树的过程

Int operation：操作指示器，0代表插入节点，1代表查找删除节点

Int\* deep：深度信息，便于计算平均查找深度

Int\* num：平均查找长度的分子

Int\* obt：顺序表的指针，用于存放二叉树

Int\* position：顺序表的下标，用于指示节点所在顺序表中的位置

这里是插入节点的核心代码

   {

        //没找到 插入节点

        if(bt == NULL)

        {

            //链式存储添加

            bitree btn = (bitree)malloc(sizeof(bitree));

            btn->data = value;

            btn->lchild = NULL;

            btn->rchild = NULL;

            //顺序存储添加

            obt[\*position] = value;

            return btn;

        }

        else

        //递归查找

        if (value < bt->data)

        {

            \*deep+=1;

            \*position \*= 2;

            bt->lchild = findNode(bt->lchild,value,0,deep,num,obt,position);

        } else

        {

            deep++;

            \*position = \*position \* 2 + 1;

            bt->rchild = findNode(bt->rchild,value,0,deep,num,obt,position);

        }

    }

**【调试分析】**

问题不是很大，主要是平衡二叉树的实现太难了，没做出来

【**问题一】**

**现象：**二叉树生成有误，仅在一条子树上创建子节点，并且导致中序遍历有误

**原因：**这个问题主要是由于一开始递归语句错误造成的，每个操作数从建立的时候应该从根节点开始进行查找递归，由于我忘记在每次添加新节点的循环入口初始化节点指针指向根节点，导致了每次遍历都是从最后一次插入的节点开始进行遍历，在循环入口加上这句代码即可。

    //插入节点

    int max = 0;

    for (; o[i] != 0; i++)

    {

        findNode(bt,o[i],0,q, p, pobt, t);

        if (max < deep)

        {

            max = deep;

        }

        deep = 1;

        position = 1;

    }

**【问题二】**

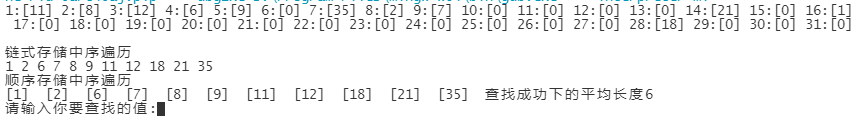
**现象：**顺序存储结构有误

**原因：**这个问题是由于数组下标导致的，顺序存储结构中，父节点下标如果为i则左右孩子节点下标分别为2i和2i + 1，但是由于根节点下标为0，导致左孩子的下标也为0，出现问题，于是修改数组从下标1开始加入数据。

**【运行结果及分析】**

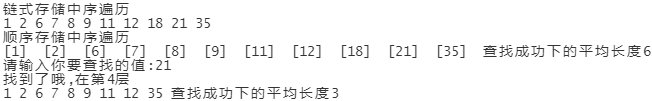
首先展示测试用的数列，用字母E结尾

11 8 6 9 12 35 2 1 7 21 18 E



程序最上输出为顺序存储的结果，下面分别输出了两种存储结构的中序遍历结果以及查找成功状态下的平均查找长度。

输入查找的节点21



重新更新二叉树重新计算平均查找长度

**【实验内容】**

11.2.10图的基本操作与实现

**【问题描述】**

要求使用邻接表存储结构，实现对图11-3所示的有向带权网络G的操作

**【需求分析】**

经过分析，本系统需完成的主要功能如下：

1. 原本是输入顶点数和边数来创建图，我这里为了方便选择用磁盘读取的方式创建
2. 求每个顶点的出度入度
3. 指定顶点DFS遍历
4. 指定顶点BFS遍历
5. 查找顶点删除点和与之相关的边并DFS遍历
6. 判断是否连通

**【概要设计】**

-=ADT=-

//定义邻接表基本结构

//最大顶点数

#define MaxVertices 100

int visited[MaxVertices]; //访问标志数组

//边表

typedef struct node {

    //边点

    char adjvex;

    //下标

    int tag;

    //权值

    int value;

    node\* next;

}EdgeNode;

//顶点表

typedef struct {

    //顶点

    char vertex;

    //下标

    int tag;

    EdgeNode\* edgenext;

}VertexNode;

typedef VertexNode AdjList[MaxVertices];//顶点表数组

typedef struct {

    //顶点

    char vertex;

    //入度

    int in;

    //出度

    int out;

}InOutInfo;

typedef struct {

    AdjList adjlist;

    int n,e;

}AdjMatrix;

//为实现广度优先遍历 队列结构

typedef struct Qnode

{

    int data;

    Qnode\* next;

}Qnode, \*Qptr;

typedef struct LINKQUEUE

{

    Qptr front;

    Qptr rear;

}LinkQueue;

AdjMatrix\* createGraph(AdjMatrix\*);

VertexNode\* addVertex(VertexNode\* vn, char vertex);

int DFS(AdjMatrix \*G, int i, int num);

int DFSTraverse(AdjMatrix \*G, int start, bool ismodified);

void InitQueue(LinkQueue \*q);

void PushQueue(LinkQueue \*q, int e);

void DetQueue(LinkQueue \*q, int &e);

int QueueEmpty(LinkQueue \*q);

void BFSTraverse(AdjMatrix \*G, int tag);

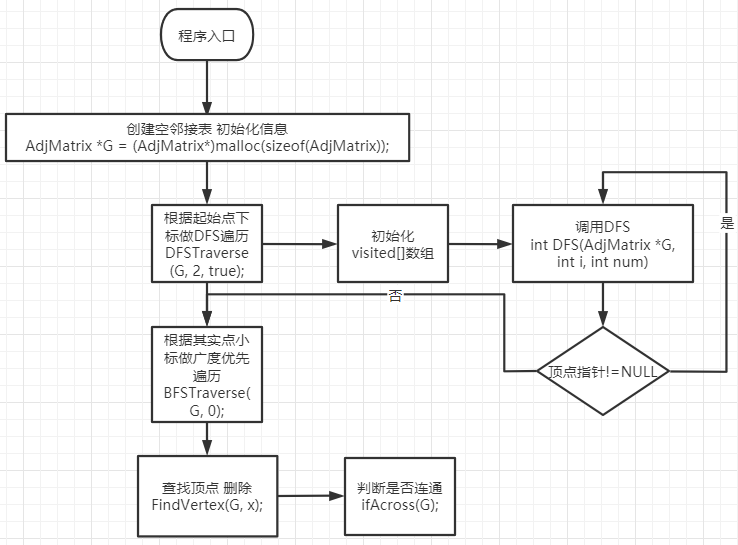
void FindVertex(AdjMatrix\* G, char key);

void ifAcross(AdjMatrix\* G);

**【存储结构】**

邻接表

**【流程图】**



**【详细设计】**

首先是邻接表的结构体设计，一个单链表代表边表，两个int类型数据域分别代表顶点数和边数，因为我这边采用的是磁盘读取数据，所以边数没啥用。

然后是DFS遍历和BFS遍历的实现，DFS遍历主要还是递归实现，需要判断访问数组的状态来避免重复遍历，此外因为需求里面还涉及到删除节点和判断连通图的操作，所以在这里写了一个前置函数，来判断具体是哪些操作，ismodified就是用来判断是否已经修改过图的信息，start是起始点的下标，当一次DFS遍历的节点数小于邻接表中的节点数则代表这不是连通图。

int DFSTraverse(AdjMatrix \*G, int start, bool ismodified)

{

    int num = 0;

    if (ismodified)

    {

        for (int i = 0; i < G->e; i++)

        {

            if (!visited[start])

            {

                DFS(G, start, num);

            }

        }

    } else

    {

        for (int i = 0; i < G->e; i++)

        {

            visited[i] = 0;//初始化标记数组为0

        }

        for (int i = 0; i < G->e; i++)

        {

            if (!visited[start])

            {

                num = DFS(G, start, num);

                if (num < G->e)

                {

                    printf("我不是连通图哦");

                    return 1;

                }

            }

        }

        printf("我是连通图哦");

    }

    return 0;

}

//邻接表的深度优先搜索

int DFS(AdjMatrix \*G, int i, int num)

{

    EdgeNode \*p;

    visited[i] = 1;

    printf("%c->", G->adjlist[i].vertex);

    num++;

    p = G->adjlist[i].edgenext;//让p指向边表的第一个结点

    while (p)

    {

        if (!visited[p->tag])

        {

            num = DFS(G, p->tag, num);

        }

        p = p->next;

    }

    return num;

}

BFS遍历主要是靠队列来实现的

//广度优先遍历

void BFSTraverse(AdjMatrix \*G, int tag)

{

    int k;

    LinkQueue Q;

    EdgeNode \*e;

    InitQueue(&Q);

    for (int i = 0; i < G->e; i++)

    {

        visited[i] = 0;//初始化标记数组

    }

    for (int i = 0; i < G->e; i++)

    {

        if (!visited[tag])

        {

            visited[tag] = 1;

            printf("%c->", G->adjlist[tag].vertex);

            PushQueue(&Q, tag);

        }

        while (QueueEmpty(&Q))

        {

            DetQueue(&Q, k);

            e = G->adjlist[k].edgenext;

            while (e)

            {

                if (!visited[e->tag])

                {

                    visited[e->tag] = 1;

                    printf("%c->", G->adjlist[e->tag].vertex);

                    PushQueue(&Q, e->tag);

                }

                e = e->next;

            }

        }

    }

}

查找节点的代码过于冗长设计到相当多状态的保存和循环控制，推荐在文件里自行查看，此处不予赘述。

**【调试分析】**

问题全部都集中在解决图的遍历的算法上，实在是太惨烈了，我感觉我创建图的思路有点绕，以至于我使用了很多循环和很多变量来控制循环条件，出现了很多问题

**【问题1】**

**现象：**DFS遍历不完全

**原因：**关键在于状态的传递和保存

**【运行结果及分析】**

首先展示图的基本数据，用了一些符号来控制程序

a: g-15, b-2, f-9,\*

b: g-6, c-4,\*

c: i-15, d-2,\*

d: i-1, e-1,\*

e: h-3, f-6,\*

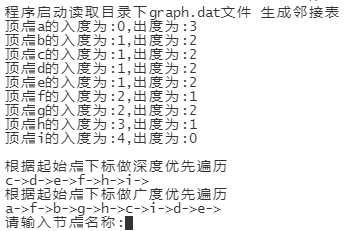
f: h-11,\*

g: h-15, i-2,\*

h: i-4,\*

i:,\*]

这边是程序的运行结果一部分



DFS和BFS遍历的起始下标在代码里改，因为最开始方便测试我没有写成输入的形式，现在也不想改了。

    printf("根据起始点下标做深度优先遍历\n");

    int a = DFSTraverse(G, 2, true);

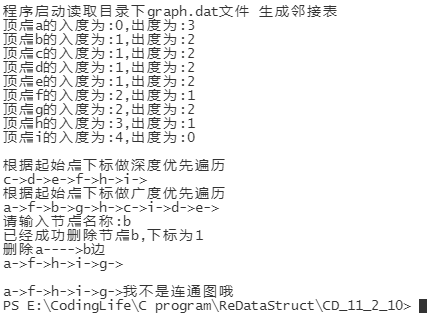
    printf("\n");

    printf("根据起始点下标做广度优先遍历\n");

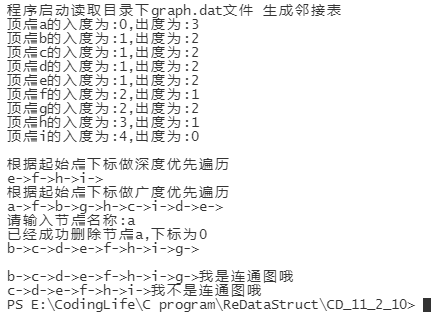
    BFSTraverse(G, 0);

    printf("\n");

然后是查找节点以及删除节点判断连通性的部分



然后我们再换个起始点试试吧，我这边换成了下标4，也就是e开始DFS遍历



**【实验内容】**

11.2.19滑雪场问题

**【问题描述】**

Michael喜欢滑雪这并不奇怪， 因为滑雪的确很刺激。可是为了获得速度，滑的区域必须向下倾斜，而且当你滑到坡底，你不得不再次走上坡或者等待升降机来载你。Michael想知道一个区域中最长的滑坡。

**【需求分析】**

经过分析，本系统需完成的主要功能如下：

1. 找到二维数组中按降序排列最长的一段记录

**【概要设计】**

-=ADT=-

#define N 7

#define max(a,b) ((a>b)?a:b)

//定义四个方向，上、左、下、右

int dir[4][2] ={{-1,0},{0,-1},{1,0},{0,1}};

//保存最大长度

int maxRuslt = 0;

//定义矩阵

int a[N][N]={{INT\_MAX,INT\_MAX,INT\_MAX,INT\_MAX,INT\_MAX,INT\_MAX,INT\_MAX},//四周用最大数防止溢出

{INT\_MAX,1,2,3,4,5,INT\_MAX},

{INT\_MAX,16,17,18,19,6,INT\_MAX},

{INT\_MAX,15,24,25,20,7,INT\_MAX},

{INT\_MAX,14,23,22,21,8,INT\_MAX},

{INT\_MAX,13,12,11,10,9,INT\_MAX},

{INT\_MAX,INT\_MAX,INT\_MAX,INT\_MAX,INT\_MAX,INT\_MAX,INT\_MAX}};

//定义dp

int dp[N][N] = {0}; //全局矩阵记录每个点能够滑行的最长长度

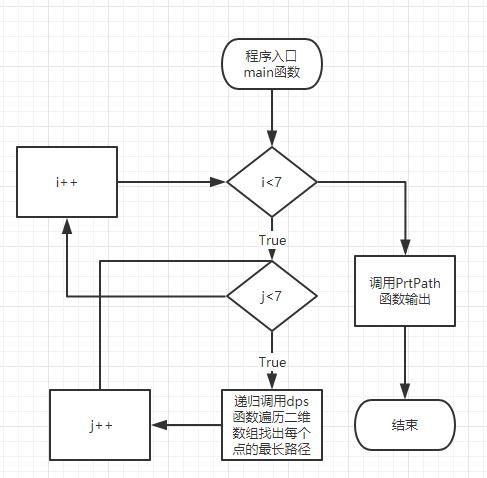
int dps(int, int); //深度遍历计算每个点的最长滑行路径长度

void PrtPath(int, int); //输出最长路径

**【存储结构】**

二维数组

**【流程图】**



**【详细设计】**

int dfs(int i,int j)//深度优先遍历搜索

{

//递归出口

if((a[i][j] < a[i-1][j]) && (a[i][j] < a[i][j-1]) && (a[i][j] < a[i+1][j]) && (a[i][j] < a[i][j+1]))

{

return dp[i][j] = 1;

}

//已经算出来则不需再计算(备忘录)

else if(dp[i][j] > 0)

{

return dp[i][j];

}

else

{

int k;

int count = 0;

int temp = 0;

for(k = 0; k < 4; k++)

{

if(a[i][j] > (a[i + dir[k][0]][j + dir[k][1]]))

{

//找到更低的一个点并递归

temp = dfs(i + dir[k][0],j + dir[k][1]) + 1;

count = max(count,temp);

dp[i][j] = count;

//最大值保存在全局变量

maxRuslt = max(maxRuslt,count);

//printf("temp = %d;count = %d;maxResult = %d;\n", temp, count, maxRuslt);

}

}

return count;

}

}

递归调用查找每一个点的最长路径并输出最大值

**【调试分析】**

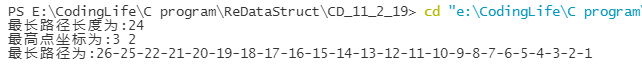
这是一个动态分析问题，主要问题就是递归调用的实现，包括最后的输出也是递归调用的实现

**【问题1】**

**现象：**在输出的时候无限循环

**原因：**由于我的输出条件是查找dp[i][j]四周存在的最大路径节点的坐标并将其对应的a[i][j]输出，并把这个节点的坐标作为递归的下一个起始点，因此总是会找到前一个更大的点，从而造成无限循环，解决方案很简单，跳过上一个递归过来的点即可

**【运行结果及分析】**



输出正确，运行正常

**三、心得体会**

前段时间完成了web的课程设计，同样也写了实验报告，对我而言，这两者的关系其实还蛮密切的，二者都是通过已学习的知识加以利用完成一项或多项完整的项目，整个过程就是对自己学习的成果的一种检验和强化。如果说web的课设使我学会了如何做出漂亮的网页，如何把自己的想法用直观的形式展现出来，那么数据结构则让我进一步加深了对数据间逻辑关系的理解，强化了把抽象的逻辑关系用代码表达出来的能力。我认为，数据结构这门课实际上就是在教我们如何把抽象的数据逻辑关系按一定规则存放在计算机内存中，这些规则就是算法，同时也是抽象关系的一种具象表达。在理解到这一层之前，学习数据结构是很痛苦的，因为我无法在抽象逻辑和具体代码之间建立一种联系。我们都知道指针是指向的一块内存地址，也知道课本和各种网站上所说的“如何利用指针创建一个单链表”，但如果我们理解不到指针在抽象逻辑中的含义，理解不到节点间的抽象逻辑关系和具体代码的联系，我们就只能读懂书中给出的代码，而无法自己利用起来。当我对指针，对数据结构有了自己的理解之后，这时写代码就变得容易起来了，因为不管你的存储结构如何变化，你其实都只在做一件事，把不同的数据节点按一定规则联系起来，至于规则是什么，书上网上都是随处可见。根据自己的理解，他人的代码也会清晰的在大脑中建立起模型，不管是修改还是查错都十分顺畅。

再说一些我对编程的理解吧，我想到以前看一本java的书的时候，作者提到一个观点，不要一行一行的按照代码的顺序抄下来，而是要理解这些代码是怎么出现的，因为编程其实并不是一种线性的思维，即使你的程序是线性的。在这次的课设我对此深有体会，在实现二叉树那道题时，最开始只是写了一个遍历的方法，但是随着逐步的深入我发现许多需求都是在遍历的过程中实现的，因此我想通过类似面向对象编程实现多态的效果，通过不同的参数来实现同一个函数的不同实现。实际上这样的情况在编程中是非常常见的，但是我们往往在阅读别人的代码的时候会忽视这样的思维，因此有时会出现跟不上别人的思路的情况，如果我们能更好的掌握这种非线性思维的思考方式和编码方式想必对于我们编码能力的提升将是巨大的。

以上就是我对本次课设的一些简单心得体会。

**四、参考文献**

1. 陈媛等编著．《算法与数据结构》(第二版)．清华大学出版社,2011