**周报**

本周学习了《零基础学算法》的第三章。本章的内容学习的主要内容有：树和图的基本结构，和他们的基本操作，还有生成最小树的prim算法和求最短路径的Dijkstra算法。学习的内容概括如下：

1. 复杂数据结构（树、图）
2. 层次关系结构：（树）
3. 树的定义：n节点的集合，集合中有一个称为根节点的特殊节点
   1. 在一一棵树中，有且仅有一个节点没有前驱，这个节点就是树的根节点。
   2. 除根节点外，其余每个点有且仅有一个前驱
   3. 每个节点可以有任意多个后续。
4. 相关术语：
   1. 父节点、子节点、兄弟节点
   2. 节点的度、数的度
   3. 叶节点和分支节点
   4. 节点的层数
   5. 树的深度
   6. 有序树和无序树、森林
5. 二叉树与树的区别
6. 树中节点的最大度树没有限制而二叉树节点的最大度为2
7. 树的节点无左右之分，而二叉树节点有左右之分
8. 特殊类型二叉树：满二叉树、完全二叉树
9. 满二叉树肯定是完全二叉树，而完全二叉树不一定是满二叉树。
10. 二叉树的存储
11. 顺序存储：
    * 1. 顺序存储数据定义：

#define MAXSIZE 100

typedef int PADA;

typedef DATA SeqBinTinree[MAXSIZE];

SeqBinTree SBT;

* + 1. 如果顺序存储的二叉树不是完全二叉树，会比较麻烦。一般采取的方法是：将非完全二叉树转换为完全二叉树，将左侧缺点虚设为无数据的节点。（这种方法浪费空间。一般只适用于一些特殊情况）

1. 链式存储：一个节点可以由节点元素和两个分别指向左右子树的指针组成（二叉树链表结构）
2. 二叉链式结构：

typedef struct ChainTree //定义二叉树结点类型

{

DATA data; //元素数据

struct ChainTree \*left; //左子树结点指针

struct ChainTree \*right; //右子树结点指针

}ChainBinTree;

1. 三叉式链式结构
2. 操作二叉树：
   1. 定义二叉树链式结构：

typedef struct ChainTree //定义二叉树结点类型

{

DATA data; //元素数据

struct ChainTree \*left; //左子树结点指针

struct ChainTree \*right; //右子树结点指针

}ChainBinTree;

* 1. 初始化二叉树：将已有的节点设置为二叉树的根节点。
  2. 添加节点到二叉树：
  3. 获取二叉树左右树
  4. 获取二叉树状态：判断二叉树是否为空、求二叉树深度
  5. 在二叉树中查找：在二叉树中查找数据时，需要遍历二叉树中的每个节点，逐个比较每一个节点的数据，然后然后该节点的指针。
  6. 清空二叉树：

if(bt)

{

BinTreeClear(bt->left); //清空左子树

BinTreeClear(bt->right);//清空右子树

free(bt);//释放当前结点所占内存

bt=NULL;

}

return;

1. 遍历二叉树：
2. 先序遍历：先访问根节点，再按先序遍历左子树，最后按先序遍历右子树。

BinTree\_DLR（BinTree）{

If(二叉树不为空)

{

访问根节点；

BinTree\_DLR（left）；//先序遍历左子树

BinTree\_DLR（right）；//先序遍历右子树

}

}

1. 中序遍历：先按中序遍历左子树，在访问根节点，最后访问右子树。
2. 后序遍历：按后序遍历左子树，再按后续遍历右子树，最后访问根节点。
3. 线索二叉树
4. 线索二叉树的表示：将每个节点中为空的做指针与右指针分别用于指针节点的前驱和后续，即可得到线索二叉树。
5. 分类：先序线索二叉树，中序线索二叉树，后续线索二叉树
6. 增加线索标志域后，二叉链表的结构如下：

typedef enum

{

SubTree,

Thread

}NodeFlag; //枚举值SubTree(子树)和Thread(线索)分别为0，1

typedef struct ThreadTree //定义线索二叉树结点类型

{

DATA data; //元素数据

NodeFlag lflag; //左标志

NodeFlag rflag; //右标志

struct ThreadTree \*left; //左子树结点指针

struct ThreadTree \*right; //右子树结点指针

}ThreadBinTree;

1. 对二叉树进行中序线索化。
2. 操作线索二叉树：
   1. 查找后续节点：
      1. 若父节点P的右子树为空，则p->right为右线索，直接指向p的中序后续。
      2. 若父节点p的右子树不为空，则p的中序后续必是其右子树中第一个中序遍历到的节点。
   2. 查找前驱节点：
      1. 若父节点p的左子树为空，则p->left为左线索，直接指向p的中序前驱节点
      2. 若父节点p的左子树不为空，则从p的左子树出发，沿该子树的右指针链表往下查找，直到一个没有右子树的节点为止，则该及诶按就是p的中序前驱节点。
   3. 遍历线索二叉树：由于二叉树被线索化，遍历时可根据后续指针快速完成，而不使用递归调用即可完成。
3. 最优二叉树（哈夫曼树）
4. 指的是对于一组具有确定权值的叶子树节点的具有最小带权路径长度的二叉树
5. 基本概念：节点的权，路径，路径的长度，树的路径长度，节点的带权路径长度，树的带权路径长度（WPL）
6. 构造哈弗曼树的步骤：
   1. 首先根据N个权值的节点构造N棵二叉树，每个节点为一个二叉树，其权值保持在根节点中，其中左右子树均为空。
   2. 在这N棵二叉树中查找出两颗根节点权值最小的树，用着两颗树作为左右子树构造一颗新的二叉树，将左右子树根节点的权值相加，作为新二叉树根节点的权值。
   3. 将上步找到的两颗权值最小的二叉树排除在下次查找查找之外，将新创建的二叉树添加到查找范围之内。
   4. 重复步骤b与c，直到查找范围只剩下一棵树为止。
7. 哈夫曼编码：首先在内存分配一片连续的区域，用来保存哈夫曼二叉树，可将这部分内存作为一个以为数组，然后通过数组的下标访问不同的二叉树节点。
8. 哈夫曼编码的一些操作：
   1. 创建哈弗曼树
   2. 根据哈弗曼树申城每一个字符的哈弗曼树编码
   3. 根据该哈夫曼编码将字符串进行编码，得到编码后的字符串
   4. 从子字符串中逐个获取二进制编码，并不断在哈弗曼树中进行查找，完成解码工作。
9. 网状关系：图（Graph）

元素之间的关系是任意的，每个数据元素都可以和其他数据元素相关

1. 图的存储：
2. 邻接矩阵：（用两个数组）
   1. 第一个数组：具有n个元素的一维数组，用来保存图中个顶点的信息。
   2. 第二个数组：n\*n的二维数组（矩阵），用来保存顶点之间相邻的关系。
3. 邻接表：是由邻接矩阵改进而来的一种连接结构，其特点值考虑邻接矩阵中的非零元素，因而比邻接矩阵保存图要节省空间。邻接表对图中的每个顶点建立一个邻接关系的单链表，邻接矩阵中的每一行对应一个单链表，将每个单链表的表头指针保存到一个数组中一方便随机访问任意一个顶点链表。
4. 创建图：
5. 用邻接矩阵：
   1. 创建邻接矩阵数据结构
   2. 创建邻接矩阵
   3. 输出邻接矩阵的内容。
6. 用邻接表保存图
   1. 定义邻接表数据结构（定义顶点的数据结构）
   2. 定义图的数据结构
   3. 创建邻接图
   4. 输出邻接图
7. 图的遍历：广度优先法和深度优先法
8. 广度优先遍历：广度优先遍历不是递归的过程。在遍历的过程中，先被访问的顶点，其邻接点也先被访问，算法的实现需要设置一个队列，用来一次记住被访问过的顶点。
9. 深度优先遍历：深度优先遍历是一个递归的过程
   1. 从isTrav数组中选择一个未被访问的顶点Vi，将其标记为已访问；
   2. 接着从Vi中的一个未被访问的邻接点出发进行深度优先遍历；
   3. 重复步骤b，直至图中所有和Vi有路径相通的顶点都被访问过；
   4. 重复a到c的步骤操作，直至所有的顶点都被访问过。
10. 生成最小树：对于有N个顶点的连通图，生成树有N-1条边。对于一个带权值的连通图，生成树不同，树中各边的权值总和也不同，权值最小的生成树称为图的最小生成树称为图的最小树。
11. 构造生成树的常用算法prim：要实现prim算法，需要连个辅助数组
    1. 数组1：tmpvertex在对应位置保存邻接顶点信息。
    2. 数组2：weight保存与上一个数组对应邻接边的权值。
12. 最短路径（Dijkstra算法）

求一个顶点到另一个顶点的最短路径。

实现的过程需要三个数组来辅助实现：（集合U是一个顶点集合，把求到的最短路径顶点添加到集合U中）

第一个：用来保存顶点集合，diagram权值为1时，表示对应的顶点被添加到U集合中

第二个：保存临时计算出来权值

第三个：表示是否存在对应的边。

**2016-10-3**

**系统控制与信息处理重点实验室**