实训3　完全二叉树

3.1　系统概要

二叉树是一种重要的数据结构，因此要了解完全二叉树的有关概念，掌握二叉树的存储结构及基本运算规则。系统利用给定的结点数据(a,s,d,f,g,h,j,k,l)，采用链式存储结构，建立一个完全二叉树；查找给定结点的父结点；查找给定结点的子结点。

3.2　基本功能要求

要求系统主要应提供以下功能：

(1) 简单的操作菜单；

(2) 建立完全二叉树；

(3) 查找给定结点的父结点；

(4) 查找给定结点的子结点；

(5) 按某种方式访问完全二叉树 t 中的每个结点，且使每个结点只被访问一次。

选做功能：

(1) 把以 s 为结点的树插入到树 t 中作为结点 x 的第 i 棵子树；

(2) 在树 t 中删除结点 x 的第 i 棵子树。

3.3　主要知识点

1. 树的基本概念

树形结构是一种非常重要的非线性结构，它用于描述元素之间的层次关系。

树的一种参考定义：任何一个非空树是满足以下条件的一个包含 n 个结点的有限集合。

(1) 唯一存在一个结点 R，它没有前驱，被称为根；

(2) 当 n>1 时，除了根之外，其他结点可分为 m(m>0)个不相交的子集 T1,T2,…,Tm，其中每一个子集都是一棵树，它们的根的前驱是 R，这些子集被称为 R 的子树，图 3.1 是树的两种表示法。

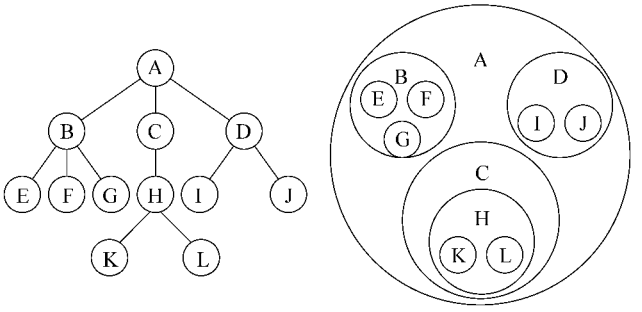

图 3.1 树的图示

树的特点：

(1) 树的根结点没有前驱结点，且除了根结点之外的所有结点有且只有一个前驱结点；

(2) 树结点可以有零个或多个后继结点。

2. 树的常用术语

(1) 树的结点：包含一个数据元素及若干指向其子树的分支。

(2) 结点的度：结点所拥有的子树的个数称为该结点的度。

(3) 叶子：度为 0 的结点。

(4) 非终端结点：度不为 0 的结点。

(5) 孩子：结点子树的根称为该结点的孩子。

(6) 兄弟：同一个双亲的孩子之间互称兄弟。

(7) 结点的祖先：从根到该结点所经分支上的所有结点。

(8) 子孙：以某结点为根的子树中的任意结点都称为该结点的子孙。

(9) 层次性：从根开始定义起，根为第一层，根的孩子为第二层；某结点在第 L 层，则其子树的根就在第 L+1 层。

(10) 树的深度：树中各结点层次的最大值称作该树的深度。

(11) 有序树：将树中结点的各子树看成从左向右是有次序的，则称该树为有序树。

3. 树的基本操作

(1) 初始化一棵树 t。

(2) 求结点 x 所在树的根结点。

(3) 求树 t 中结点 x 的双亲结点。

(4) 求树 t 中结点 x 的第 i 个孩子结点。

(5) 把以 s 为结点的树插入到树 t 中作为结点 x 的第 i 棵子树。

(6) 在树 t 中删除结点 x 的第 i 棵子树。

(7) 按某种方式访问树 t 中的每个结点，且使每个结点只被访问一次。

4. 二叉树及其遍历

二叉树(Binary Tree)：是一种典型的树形结构，它的特点是每个结点至多只有两棵子树，并且，二叉树的子树有左右之分，其次序不能任意颠倒。

满二叉树：一棵深度为 k 且有 2^k-1 个结点的二叉树。即最后一层都是叶子结点，其他各层的结点都有左、右子树的二叉树，如图 3.2 所示。

完全二叉树：深度为 k 的，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应。即一棵二叉树最多只有最后两层有度数小于 2 的结点，且最下层的结点都集中在该层的最左边的若干位置上，如图 3.3 所示。

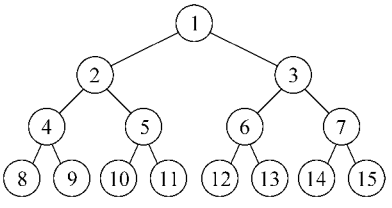

图 3.2　满二叉树

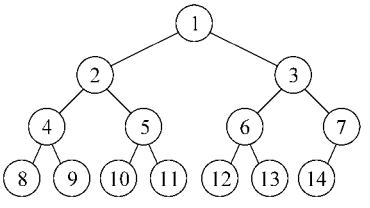

图 3.3　完全二叉树

树的遍历是树应用中一种重要的操作。在树的一些应用中，需要在树中查找某些满足要求的结点，或者对树中的所有结点逐一进行某些操作，即进行树的遍历，每个结点访问且仅访问一次。

二叉树的遍历(Traversal)：按某指定规则访问树中每个结点，使得每个结点均被访问一次，且仅被访问一次。二叉树的遍历方案包括先序遍历、中序遍历、后序遍历、层次遍历。

5. 队列

(1) 队列(Queue)：是操作受限的线性表，允许在表的一端进行插入，而在表的另一端进行删除。是一种先进先出(FIFO)的线性表。$(a_1,a_2,\cdots,a_n)$，如图 3.4 所示。

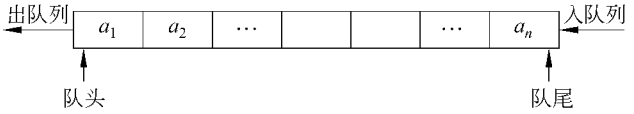

图 3.4　队列

(2) 队尾(rear)：允许插入的一端叫做队尾。

(3) 队头(front)：允许删除的一端则称队头。

(4) 单向队列：若限制插入在表的一端进行，而限制删除在表的另一端进行，称此队列为“单向队列”。

(5) 双向队列：若限制插入和删除均能在表的两端进行，称此队列为“双向队列”。

3.4　系统设计思路

1. 先序遍历二叉树(DLR)

算法：

S1：访问根结点。

S2：先序遍历根结点的左子树。

S3：先序遍历根结点的右子树。

2. 中序遍历二叉树(LDR)

算法：

S1：中序遍历根结点的左子树。

S2：访问根结点。

S3：中序遍历根结点的右子树。

3. 后序遍历二叉树(LRD)

算法：

S1：后序遍历根结点的左子树。

S2：后序遍历根结点的右子树。

S3：访问根结点。

4. 二叉树的层次遍历

算法：

按层次遍历所得的结点访问序列中各结点的序号与按层次编号所得的编号一致。

5. 系统设计思想

S1：定义程序中所用到的二叉树的结点结构和队列结点结构。二叉树结点结构至少应包括三个成员，一个是字符类型，用来存储结点数据；另外两个分别是指向结点左、右孩子的指针。

S2：定义一个完全二叉树的根结点指针变量，作为访问完全二叉树的入口。

S3：定义各种操作的函数，主要完成建立完全二叉树中用到的函数，查找指定结点的父结点、子结点所需的函数。

S4：函数 buildBTree()，根据给定的结点数据构建完全二叉树。在构建完全二叉树的过程中，需要用到辅助数据结构“队列”。每往完全二叉树中加入一个结点，就将这个结点也插入到队列中。每次从队首取出一个结点，给这个结点加上左、右孩子，将左、右孩子再依次加入到队尾。

S5：函数 parent()，查找指定结点的父结点。要用到递归，采用前序遍历算法，在已经建立好的完全二叉树中，逐个比较遍历到的结点的子结点数据是否是指定的结点值。

S6：函数 child()，查找指定结点的子结点。需要用到递归，也采用前序遍历算法。将遍历到的结点的结点值，与指定的结点值做比较，直到找到指定的结点。将该结点的左、右孩子返回。

6. 数据结构定义

我们将与树结点及操作相关的定义和说明存储在 btree.h 中，将与队列操作相关的定义和说明存储在 queue.h 中。

btree.h 中的内容如下：

```
#ifndef BTREE_H
#define BTREE_H
typedef struct BTreeNode {
```

```
    char data;
    struct BTreeNode * lChild;
    struct BTreeNode * rChild;
} BTREENODE;
BTREENODE * T;                                    /* 定义指向根结点的指针 */
Void buildBTree(const char * szNodeData);         /* 建立二叉树 */
BTREENODE * parent(char data);          /* 获取指定结点的父结点,若不存在返回 NULL */
void child(char data,BTREENODE ** lChild,BTREENODE ** rChild);
                                                  /* 获取指定结点的子结点 */
#endif
```

queue.h 中的内容如下：

```
#ifndef QUEUE_H
#define QUEUE_H
#include "bTree.h"
typedef struct queueNode {
    void * pItem;
    struct queueNode * preNode;
    struct queueNode * postNode;
} QUEUENODE;
QUEUENODE * queueFront;                           /* 定义指向队列头的指针 */
QUEUENODE * queueRear;                            /* 定义指向队列尾的指针 */
void initQueue();                                 /* 初始化队列 */
void addQueue(void * pItem);                      /* 向队列中增加一个元素 */
void * deleteQueue();                   /* 删除队列的一个元素,若队列为空,返回 NULL */
#endif
```

3.5　部分程序代码

```
# define    MAX_LENGTH        128
QUEUENODE * mallocQueueNode(void * pItem)         /* 为队列结点分配存储空间 */
{
    QUEUENODE * pQueueNode;
    pQueueNode= (QUEUENODE * )malloc(sizeof(QUEUENODE));
    pQueueNode->pItem=pItem;
    pQueueNode->preNode=NULL;
    pQueueNode->postNode=NULL;
    return pQueueNode;
}
void initQueue()                                  /* 初始化队列 */
{
    queueFront=NULL;
    queueRear=NULL;
```

```
}

void addQueue(void * pItem)                    /* 向队列中增加一个元素 */
{
    QUEUENODE * mallocQueueNode(void * pItem);
    QUEUENODE * pQueueNode;
    if(queueFront==NULL && queueRear==NULL)
    {
        queueFront=mallocQueueNode(pItem);
        queueRear=queueFront;
    }
    else
    {
        pQueueNode=mallocQueueNode(pItem);
        pQueueNode->preNode=queueRear;
        queueRear->postNode=pQueueNode;
        queueRear=pQueueNode;
    }
}

void * deleteQueue()                           /* 删除队列的一个元素 */
{
    void * pItem;
    QUEUENODE * pQueueNode;
    if(queueFront==NULL)                       /* 若队列为空,返回 NULL */
    {
        return NULL;
    }

    pQueueNode=queueFront;
    if(queueFront==queueRear)                  /* 队列有一个元素 */
    {
        queueFront=NULL;
        queueRear=NULL;
    }
    Else                                       /* 第一个元素出队列 */
    {
        queueFront=queueFront->postNode;
        queueFront->preNode=NULL;
    }
    pItem=pQueueNode->pItem;                   /* 保存队列第一个元素 */
    free(pQueueNode);                          /* 释放队列第一个元素的空间 */
    return pItem;
}
```

```
BTREENODE * mallocBTreeNode(char c)                    /* 申请树结点空间 */
{
    BTREENODE * pBTreeNode= (BTREENODE * )malloc(sizeof(BTREENODE));
    pBTreeNode->data=c;
    pBTreeNode->lChild=NULL;
    pBTreeNode->rChild=NULL;
    return pBTreeNode;
}

void freeNode(BTREENODE * pBTreeNode)                  /* 释放树结点空间 */
{
    free(pBTreeNode);
}

void buildBTreeWithInputValue()                        /* 输入数据,构建完全二叉树 */
{
    void parseInput(const char * szInput,char szNodeValue[]);
    char szInput[MAX_LENGTH],szNodeValue[MAX_LENGTH];
    memset(szInput,'\0',sizeof(szInput));
     /* 设置 szInput 中的所有字节为'\0',szInput 数组的大小由 sizeof(szInput)给定; */
    memset(szNodeValue,'\0',sizeof(szNodeValue));
    printf("input the node values:");
    scanf("%s",szInput);
    parseInput(szInput,szNodeValue);                   /* 将输入数据存储到结点数据空间 */
    buildBTree(szNodeValue);                           /* 构建二叉树 */
    printf("\n\n\n\n\n");
    printf("The binary tree has been builded!");
    printf("\n\n\n\n\n");
    printf("press any key to continue...");
    while(!_kbhit());
    system("cls");
}

void findParent()                                      /* 查询一结点的父结点 */
{   BTREENODE * pParent;
    char nodeValue[MAX_LENGTH];
    memset(nodeValue,'\0',sizeof(nodeValue)); /* 对 nodeValue 初始化 */
    printf("input the char value of the node which you want to see its parent:");
    scanf("%s",nodeValue);                             /* 输入已知结点 */
    pParent=parent(nodeValue[0]);                      /* 调用求父结点函数 */
    printf("\n\n\n\n\n");
    if(pParent !=NULL)
    {
```

```
        printf("The the parent of node\"%c\"is\"%c\"!",nodeValue[0],pParent->data);
    }
    else
    {
        printf("Don't find the parent of the node \"%c\"!",nodeValue[0]);
    }
        printf("\n\n\n\n\n");
    printf("press any key to continue...");
    while(!_kbhit());
    system("cls");
}

void findChildren()                                    /*查询一结点的子结点*/
{
    BTREENODE** lChild,** rChild;
    char nodeValue[MAX_LENGTH];
    memset(nodeValue,'\0',sizeof(nodeValue));   /*对 nodeValue 初始化*/
    lChild=(BTREENODE**)malloc(sizeof(BTREENODE**));
    rChild=(BTREENODE**)malloc(sizeof(BTREENODE**));
    printf("input the char value of the node which you want to see its children:");
    scanf("%s",nodeValue);
    *lChild=NULL;
    *rChild=NULL;
    child(nodeValue[0],lChild,rChild);              /*调用求子结点函数*/
    printf("\n\n\n\n\n");
    if((*lChild)!=NULL)
    {
        printf("The lChild of node\"%c\"is\"%c\"!",nodeValue[0],(*lChild)->data);
    }
    else
    {
        printf("Don't find the lChild of the node \"%c\"!",nodeValue[0]);
    }
    printf("\n");
    if((*rChild)!=NULL)
    {
        printf("The rChild of node\"%c\"is\"%c\"!",nodeValue[0],(*rChild)->data);
    }
    else
    {
        printf("Don't find the rChild of the node \"%c\"!",nodeValue[0]);
    }
    printf("\n\n\n\n\n");
    printf("press any key to continue...");
```

```
    while(!_kbhit());
    system("cls");
}

/*构造 Binary Tree*/
void buildBTree(const char*szNodeData)
{
    BTREENODE*mallocBTreeNode(char c);
    BTREENODE*queueNode;
    int len,i;
        len=(int)strlen(szNodeData);
    for(i=0; i<len; i++)
    {
        if(!T)
        {
            T=mallocBTreeNode(szNodeData[i]);
            addQueue(T);
        }
        else
        {
            queueNode=(BTREENODE*)deleteQueue();
            if(queueNode==NULL)
            {
                continue;
            }
            queueNode->lChild=mallocBTreeNode(szNodeData[i]);
            addQueue(queueNode->lChild);
            if(++i<len)
            {
                queueNode->rChild=mallocBTreeNode(szNodeData[i]);
                addQueue(queueNode->rChild);
            }
        }
    }
}

/*将输入数据存储到结点数据空间*/
void parseInput(const char*szInput,char szNodeValue[])
{
    size_t len,i,j;
    if(szInput!=NULL)                          /*判断如果字符指针不为空*/
    {
        len=strlen(szInput);                   /*计算 szInput 的长度*/
        for(i=0,j=0; i<len; i++)               /*在计算的长度内循环*/
```

```
        {
            if(szInput[i]==',')
            {
                continue;                          /*结束本次循环*/
            }
            else
            {
                szNodeValue[j++]=szInput[i];
                                                   /*将 szInput 的值赋给 szNodeValue*/
            }
        }
    }
}

/*找到指定结点的父结点*/
BTREENODE*parent(char data)
{
    BTREENODE*getParent(BTREENODE*pBTreeNode,char data);
    if(T==NULL || T->data==data)
    {
        return NULL;
    }
    return getParent(T,data);
}

/*找到指定结点的子结点*/
void child(char data,BTREENODE**lChild,BTREENODE**rChild)
{
    void getChildren(BTREENODE*pBTreeNode,char data,BTREENODE**lChild,
BTREENODE**rChild);
    if(T==NULL)
    {
        *lChild=NULL;
        *rChild=NULL;
        return;
    }
    getChildren(T,data,lChild,rChild);
}

BTREENODE*getParent(BTREENODE*T,char data)/*用递归法求父结点*/
{
    BTREENODE*pBTreeNode=NULL;
    if((T->lChild!=NULL && T->lChild->data==data)||(T->rChild!=NULL &&
T->rChild->data==data))
```