树与二叉树

本文档需要用到的编程相关知识点：

C语言：struct（结构体）、typedef、malloc（free）、enum、函数做参数。

本文档的参考教材：

数据结构（C语言版） 严蔚敏

一. 树的实现

树的实现参照教材135页。

```c
/**
 * 树的双亲表存储表示与实现
 * 参照教材135页
 * */

#include <stdio.h>
#include <stdlib.h>

/**
 * 宏定义及类型定义
 * */
#define ERROR -1
#define OK 1
#define TRUE 1
#define FALSE 0
typedef int Status;

#define MAX_TREE_SIZE 100
typedef struct PTNode {  // 结点结构
    TElemType data;
    int parent;  // 双亲位置
} PTNode;
typedef struct {  // 树结构
    PTNode nodes[MAX_TREE_SIZE];
    int r, n;  // 根的位置和结点数
} PTree;
```

二. 二叉树的实现

1. 二叉树的基本操作实现

二叉树的实现参照教材127页。

```c
/**
 * 二叉树的二叉链表存储表示与实现
 * 参照教材127页
 * */

#include <stdio.h>
#include <stdlib.h>

/**
 * 宏定义及类型定义
 * */
#define ERROR -1
#define OK 1
#define TRUE 1
#define FALSE 0
typedef int Status;
typedef char TElemType;  // 定义二叉树结点的数据元素类型

/**
 * 定义二叉树的结点
 * */
typedef struct BiTNode {
    TElemType data;
    struct BiTNode *lchild, *rchild;  // 左右孩子指针
} BiTNode, *BiTree;

typedef BiTree QElemType;
// 定义链队列结点
typedef struct QNode {
    QElemType data;
    struct QNode *next;
} QNode, *QueuePtr;

typedef struct {     // 定义链队列
    QueuePtr front;  // 队头指针
    QueuePtr rear;   // 队尾指针
} LinkQueue;

/**
 * 构造一个空队列Q
 */
Status InitQueue(LinkQueue &Q)
{
    Q.front = (QNode *)malloc(sizeof(QNode));
    if (!Q.front)
        return ERROR;
    Q.front->next = NULL;
    Q.rear = Q.front;
    return OK;
}
```

```
51
52  /**
53   * 判断Q是否为空，是返回TRUE，否则返回FALSE
54   */
55  Status QueueEmpty(LinkQueue Q)
56  {
57      if (Q.front == Q.rear)
58          return TRUE;
59      else
60          return FALSE;
61  }
62
63  /**
64   * 插入元素e为Q的新的队尾元素
65   */
66  Status EnQueue(LinkQueue &Q, QElemType e)
67  {
68      QueuePtr p;
69      p = (QNode *)malloc(sizeof(QNode));  // 生成新的结点
70      if (!p)
71          return ERROR;
72      p->data = e;
73      p->next = NULL;
74      Q.rear->next = p;
75      Q.rear = p;
76      return OK;
77  }
78
79  /**
80   * 若队列不空，删除队列Q的队头元素，用e返回其值，返回OK，否则返回ERROR
81   */
82  Status DeQueue(LinkQueue &Q, QElemType &e)
83  {
84      QueuePtr p;
85      p = Q.front->next;
86      if (Q.front == Q.rear)
87          return ERROR;
88      e = p->data;
89      if (Q.front->next == Q.rear)  // 队列中只有一个元素时，还要修改尾指针
90          Q.rear = Q.front;
91      Q.front->next = p->next;
92      free(p);
93      p = NULL;
94      return OK;
95  }
96
97  /**
98   * 算法6.4，按先序顺序构造二叉树
99   */
100 Status CreateBiTree(BiTree &T)
101 {
102     char ch;
103     scanf("%c", &ch);
104     if (' ' == ch)
105         T = NULL;
106     else {
107         T = (BiTNode *)malloc(sizeof(BiTNode));
108         if (!T)
```

```
109             return ERROR;
110         T->data = ch;
111         CreateBiTree(T->lchild);  // 构造左子树
112         CreateBiTree(T->rchild);  // 构造右子树
113     }
114     return OK;
115 }
116 
117 /**
118  * 打印一个元素
119  */
120 Status display(TElemType e)
121 {
122     printf("%c", e);
123     return OK;
124 }
125 
126 /**
127  * 算法6.1，先序遍历二叉树，对每个元素调用函数visit
128  */
129 Status PreOrderTraverse(BiTree T, Status (*visit)(TElemType e))
130 {
131     if (T) {
132         if (OK == visit(T->data))
133             if (OK == PreOrderTraverse(T->lchild, visit))
134                 if (OK == PreOrderTraverse(T->rchild, visit))
135                     return OK;
136         return ERROR;
137     } else
138         return OK;
139 }
140 
141 /**
142  * 中序遍历二叉树，对每个元素调用函数visit
143  */
144 Status InOrderTraverse(BiTree T, Status (*visit)(TElemType e))
145 {
146     if (T) {
147         if (OK == InOrderTraverse(T->lchild, visit))
148             if (OK == visit(T->data))
149                 if (OK == InOrderTraverse(T->rchild, visit))
150                     return OK;
151         return ERROR;
152     } else
153         return OK;
154 }
155 
156 /**
157  * 后序遍历二叉树，对每个元素调用函数visit
158  */
159 Status PostOrderTraverse(BiTree T, Status (*visit)(TElemType e))
160 {
161     if (T) {
162         if (OK == PostOrderTraverse(T->lchild, visit))
163             if (OK == PostOrderTraverse(T->rchild, visit))
164                 if (OK == visit(T->data))
165                     return OK;
166         return ERROR;
```

```
    } else
        return OK;
}

/**
 * 层序遍历二叉树，利用队列实现
 */
Status LevelOrderTraverse(BiTree T, Status (*visit)(TElemType e))
{
    BiTree p;
    LinkQueue Q;
    InitQueue(Q);
    p = T;
    while (p || !QueueEmpty(Q)) {
        if (p) {
            visit(p->data);
            if (p->lchild)
                EnQueue(Q, p->lchild);
            if (p->rchild)
                EnQueue(Q, p->rchild);
            if (!QueueEmpty(Q))
                DeQueue(Q, p);
            else  // 队列为空时，退出while循环
                break;
        }
    }
    return OK;
}
```

二叉树测试程序：（测试程序代码放在实现代码下方即可）

```
int main()
{
    BiTree t1, t2;

    printf(
        "请按先序次序输入二叉树 t1 "
        "中结点的值（一个字符），空格字符表示空树：\n");
    // 在交互界面输入如下语句："I-oY  o  vu  !  Le  -  "
    CreateBiTree(t1);
    printf("二叉树 t1 的层序遍历结果为：\n");
    LevelOrderTraverse(t1, display);  // 测试二叉树的层序遍历

    printf(
        "\n\n请按先序次序输入二叉树 t2 "
        "中结点的值（一个字符），空格字符表示空树：\n");
    // 在交互界面输入如下语句：",n-oY  u  a  -d  em   -hon  t  ni  g  teb  t
re  !   "
    CreateBiTree(t2);
    printf("二叉树 t2 的中序遍历结果为：\n");
    InOrderTraverse(t2, display);

    return 0;
}
```

测试程序运行结果为：（按注释中所给输入字段的运行结果）

请按先序次序输入二叉树 t1 中结点的值（一个字符），空格字符表示空树：
I-oY o vu ! Le -
二叉树 t1 的层序遍历结果为：
I-Love-You!

请按先序次序输入二叉树 t2 中结点的值（一个字符），空格字符表示空树：
,n-oY u a -d em -hon t ni g teb t re !
二叉树 t2 的中序遍历结果为：
You-and-me,nothing-better!

---

2. 二叉树的拓展

- 二叉线索树

  二叉线索树的实现参照教材133页。

```c
/**
 * 二叉树的二叉线索存储表示与实现
 * 参照教材133页
 * */

#include <stdio.h>
#include <stdlib.h>

/**
 * 宏定义及类型定义
 * */
#define ERROR -1
#define OK 1
#define TRUE 1
#define FALSE 0
typedef int Status;
typedef enum PointerTag { Link, Thread };  // Link == 0, 指针; Thread == 1, 线索

/**
 * 定义二叉线索树
 * */
typedef struct BiThrNode {
    TElemType data;
    struct BiThrNode *lchild, *rchild;  // 左右孩子指针
    PointerTag LTag, RTag;              // 左右标志
} BiThrNode, *BiThrTree;

/**
 * 按先序顺序构造二叉线索树
 */
Status CreateBiThrTree(BiThrTree &T)
{
    char ch;
    //  scanf("%c", &ch);
```

```
    ch = getchar();
    if (' ' == ch)
        T = NULL;
    else {
        T = (BiThrNode *)malloc(sizeof(BiThrNode));
        if (!T)
            return ERROR;
        T->data = ch;
        T->LTag = Link;
        T->RTag = Link;
        CreateBiThrTree(T->lchild);  // 构造左子树
        CreateBiThrTree(T->rchild);  // 构造右子树
    }
    return OK;
}

/**
 * 算法6.5，T指向头结点，头结点的左链lchild指向根结点，右链指向遍历的最后一个结点，
 * 中序遍历二叉线索树的非递归算法，对每个数据元素调用函数visit
 */
Status InOrderTraverse_Thr(BiThrTree T, Status (*visit)(TElemType e))
{
    BiThrTree p;
    p = T->lchild;     // p指向根结点
    while (p != T) {  // 空树或遍历结束时，p == T
        while (Link == p->LTag)
            p = p->lchild;  // 走到左子树的尽头
        if (ERROR == visit(p->data))
            return ERROR;  // 访问其左子树为空的结点
        while (
            Thread == p->RTag &&
            p->rchild !=
                T) {  // 右链为线索，指向后继结点，当p为最后一个结点时，p->rchild
                      // == T (头结点)
            p = p->rchild;
            if (ERROR == visit(p->data))  // 访问后继结点
                return ERROR;
        }
        p = p->rchild;  // 右链不是线索时，将p指向其右子树，或者p指向头结点，表明遍历结束
    }
    return OK;
}

/**
 * 算法6.6，中序遍历二叉树T，并将其中序线索化，Thrt指向头结点
 */
Status InOrderThreading(BiThrTree &Thrt, BiThrTree T)
{
    printf("");  // 不知道什么原因，不加这一句就什么都显示不出来，真他妈的奇怪
    if (!(Thrt = (BiThrNode *)malloc(sizeof(BiThrTree))))  // 建头结点
        return ERROR;
    Thrt->LTag = Link;
    Thrt->RTag = Thread;
    Thrt->rchild = Thrt;
    if (!T)
```

```
90          Thrt->lchild = Thrt;  // T为空树，则头结点左指针指向头结点
91      else {
92          Thrt->lchild = T;  // 头结点左链指向根结点
93          pre = Thrt;        // pre初始时指向头结点
94          InThreading(T);    // 中序遍历进行线索化
95          pre->rchild = Thrt;  // 最后一个结点线索化，最后一个结点的右链指向头结
    点
96          pre->RTag = Thread;
97          Thrt->rchild = pre;  // 头结点的右链指向最后一个遍历的结点
98      }
99      return OK;
100 }
101
102 /**
103  * 算法6.7
104  */
105 void InThreading(BiThrTree p)
106 {
107     if (p) {
108         InThreading(p->lchild);  // 左子树线索化
109         if (!p->lchild) {        // 左链为空时，前继线索
110             p->LTag = Thread;
111             p->lchild = pre;  // pre为中序遍历访问的前一个结点
112         }
113         if (!pre->rchild) {  // 前一个结点的右链为空时，后继线索
114             pre->RTag = Thread;
115             pre->rchild = p;
116         }
117         pre = p;                 // 保持pre指向p的前驱
118         InThreading(p->rchild);  // 右子树线索化
119     }
120 }
```

- 赫夫曼树

  赫夫曼树的实现参照教材147页。

```
1  /**
2   * 赫夫曼树和赫夫曼编码的存储表示与实现
3   * 参照教材147页
4   * */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9
10 /**
11  * 宏定义及类型定义
12  * */
13 #define ERROR -1
14 #define OK 1
15 #define TRUE 1
16 #define FALSE 0
17 typedef int Status;
18
19 /**
```

```
 * 定义赫夫曼树
 * */
typedef struct {
    unsigned int weight;                    // 节点权重
    unsigned int parent, lchild, rchild;  // 定义节点双亲，左孩子，右孩子位置
} HTNode, *HuffmanTree;  // 动态分配数组存储赫夫曼树

typedef char **HuffmanCode;  // 动态分配数组存储赫夫曼编码表

/**
 * s1, s2返回HT的s1, s2, n序中权值最小的两个
 */
void max(HuffmanTree HT, int &s1, int &s2, int n)
{
    if (HT[s1].weight <= HT[s2].weight) {
        if (HT[n].weight < HT[s2].weight)
            s2 = n;
    } else {
        if (HT[n].weight < HT[s1].weight)
            s1 = n;
    }
}

/**
 * 在数组HT[1...n]中选择parent为0且weight最小的两个结点，其序号分别为s1,s2,
 * 并且s1的权小于s2的权
 */
Status Select(HuffmanTree HT, int n, int &s1, int &s2)
{
    int i;
    for (i = 1; i <= n; i++) {    // 遂个查找权值最小的两个结点
        if (0 == HT[i].parent) {  // 在parent为0的结点中寻找
            if (0 == s1)
                s1 = i;
            else if (0 == s2)
                s2 = i;
            else
                max(HT, s1, s2, i);
        }
    }
    if (s1 > s2) {
        i = s1;
        s1 = s2;
        s2 = i;
    }
    return OK;
}

/**
 * 算法6.12，求赫夫曼编码的算法
 * w存放n个字符的权值（均>0），构造赫夫曼树HT，并求出n个字符的赫夫曼编码HC
 */
Status HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *w, int n)
{
    int m, i, s1, s2, start, current, further;
    HuffmanTree p;
    char *code;
```

```
77      s1 = 0;
78      s2 = 0;
79      if (n <= 1)
80          return ERROR;
81      m = 2 * n - 1;  // 一棵有n个结点的赫夫曼树有2*n-1个结点
82      HT = (HuffmanTree)malloc((m + 1) * sizeof(HTNode));  // 0号单元未用
83      for (p = HT + 1, i = 1; i <= n;
84           ++i, ++p, ++w) {  // 初始化前n个结点，并将n个权值依次赋给它们
85          (*p).weight = *w;
86          (*p).parent = 0;
87          (*p).lchild = 0;
88          (*p).rchild = 0;
89      }
90      for (; i <= m; ++i, ++p) {  // 初始化剩余的结点
91          (*p).weight = 0;
92          (*p).parent = 0;
93          (*p).lchild = 0;
94          (*p).rchild = 0;
95      }
96      for (i = n + 1; i <= m; ++i) {  // 建赫夫曼树
97          s1 = 0;
98          s2 = 0;
99          Select(HT, i - 1, s1, s2);  // 找到权最小的两个根结点，且s1的权小于
    s2
100         HT[s1].parent = i;
101         HT[s2].parent = i;
102         HT[i].lchild = s1;
103         HT[i].rchild = s2;
104         HT[i].weight = HT[s1].weight + HT[s2].weight;
105     }
106
107     /* 从叶子到根逆向求每个字符的赫夫曼编码 */
108     HC = (HuffmanCode)malloc(
109         (n + 1) * sizeof(char *));  // 分配n个字符编码的头指针向量，0单元未
    用
110     //分配求编码的工作空间，n个字符的编码位数最多为n-1位，最后一位存储'\0'
111     code = (char *)malloc(n * sizeof(char));
112     code[n - 1] = '\0';
113     for (i = 1; i <= n; i++) {  // 逐个字符求赫夫曼编码
114         start = n - 1;
115         for (current = i, further = HT[i].parent; further != 0;
116              current = further,
117             further = HT[further].parent) {  // 从叶子开始到根逆向求编码
118             if (current == HT[further].lchild)
119                 code[--start] = '0';
120             else
121                 code[--start] = '1';
122         }
123         HC[i] = (char *)malloc((n - start) *
124                                sizeof(char));  // 为第i个字符编码分配空间
125         strcpy(HC[i], &code[start]);
126     }
127     free(code);
128     code = NULL;
129     return OK;
130 }
131
132 /**
```

```
 * 算法6.13，遍历赫夫曼树求编码,HT为已存在的赫夫曼树，HC保存字符编码，
 * n为字符个数,即树的叶子个数
 */
void get_huffmanCode(HuffmanTree HT, HuffmanCode &HC, int n)
{
    int i;
    char *code;   // 编码的工作空间
    int codeLen;  // 编码长度
    char p;
    HC = (HuffmanCode)malloc(
        (n + 1) * sizeof(char *));  // 分配指向编码的头指针空间，0号单元未用
    code = (char *)malloc(n * sizeof(char));
    codeLen = 0;
    p = 2 * n - 1;  // p指向根结点

    for (i = 1; i <= p; i++)
        HT[i].weight = 0;  // 遍历赫夫曼树时用作结点标志
    while (p) {
        if (0 == HT[p].weight) {  // 向左
            HT[p].weight = 1;
            if (HT[p].lchild != 0) {
                p = HT[p].lchild;
                code[codeLen++] = '0';
            } else if (0 == HT[p].rchild) {
                HC[p] = (char *)malloc((codeLen + 1) * sizeof(char));
                code[codeLen] = '\0';
                strcpy(HC[p], code);
            }
        } else if (1 == HT[p].weight) {  // 向右
            HT[p].weight = 2;
            if (HT[p].rchild != 0) {
                p = HT[p].rchild;
                code[codeLen++] = '1';
            }
        } else {
            HT[p].weight = 0;
            p = HT[p].parent;
            codeLen--;
        }
    }
}
```

Author: XF

Finished time: 2019/12/7