实验08二叉树及应用（二）

实验周次：第15周 学时：2学时 地点：学院机房

学号：3190707121 姓名：武新纪 班级：人工智能191 序号：21

提示：请务必填写 以上个人信息。

实验每题20分，共计5题。

**【实验目的】**

1.掌握二叉树遍历算法的应用，熟练使用先序、中序和后序和层次遍历算法进行二叉树问题的求解。

2.领会二叉树的构造过程以及构造二叉树的算法设计。

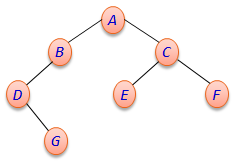
3.领会线索二叉树的构造过程以及构造线索二叉树的算法设计以及线索二叉树的遍历算法。

4.领会哈夫曼树的构造过程以及哈夫曼编码的生成过程。

**【实验内容】**

**在上节实验课针对以下二叉树**，创建的二叉链式存储结构b的基础上继续完成以下实验。

该二叉树的括号表示法对应字符串为“*A*(*B*(*D*(，*G*))，*C*(*E*，*F*))



实验1.编写一个程序exp-7-3.cpp,实现以下功能：

1）输出二叉树b的所有结点（的值，结点之间用空格隔开）和总结点个数。

2）输出二叉树b的所有叶子结点（的值，结点之间用空格隔开）和总的叶子结点个数。

3）求出结点值为'E'的结点的层次（即在二叉树b的第几层，根结点为第1层）。

4）求出二叉树b第3层的结点个数。

5）输出结点值为'E'的结点的所有祖先结点（的值）。

**程序:**

#include <stdio.h>

#include <malloc.h>

#define MaxSize 100

typedef char ElemType;

typedef struct node

{

    ElemType data;       //数据元素

    struct node \*lchild; //指向左孩子结点

    struct node \*rchild; //指向右孩子结点

} BTNode;

void CreateBTree(BTNode \*&b, const char \*str) //创建二叉树

{

    BTNode \*St[MaxSize], \*p = NULL;

    int top = -1, k, j = 0;

    char ch;

    b = NULL; //建立的二叉树初始时为空

    ch = str[j];

    while (ch != '\0') //str未扫描完时循环

    {

        switch (ch)

        {

        case '(':

            top++;

            St[top] = p;

            k = 1;

            break; //为左孩子结点

        case ')':

            top--;

            break;

        case ',':

            k = 2;

            break; //为孩子结点右结点

        default:

            p = (BTNode \*)malloc(sizeof(BTNode));

            p->data = ch;

            p->lchild = p->rchild = NULL;

            if (b == NULL) //\*p为二叉树的根结点

                b = p;

            else //已建立二叉树根结点

            {

                switch (k)

                {

                case 1:

                    St[top]->lchild = p;

                    break;

                case 2:

                    St[top]->rchild = p;

                    break;

                }

            }

        }

        j++;

        ch = str[j];

    }

}

void DestroyBTree(BTNode \*&b) //销毁二叉树

{

    if (b != NULL)

    {

        DestroyBTree(b->lchild);

        DestroyBTree(b->rchild);

        free(b);

    }

}

int Nodes(BTNode \*b) //求二叉树b的结点个数

{

    int num1, num2;

    if (b == NULL)

        return 0;

    else if (b->lchild == NULL && b->rchild == NULL)

        return 1;

    else

    {

        num1 = Nodes(b->lchild);

        num2 = Nodes(b->rchild);

        return (num1 + num2 + 1);

    }

}

int LeafNodes(BTNode \*b) //求二叉树b的叶子结点个数

{

    int num1, num2;

    if (b == NULL)

        return 0;

    else if (b->lchild == NULL && b->rchild == NULL)

        return 1;

    else

    {

        num1 = LeafNodes(b->lchild);

        num2 = LeafNodes(b->rchild);

        return (num1 + num2);

    }

}

int Level(BTNode \*b, ElemType x, int h) //求二叉树b中结点值为x的结点的层次

{

    int l;

    if (b == NULL)

        return (0);

    else if (b->data == x)

        return (h);

    else

    {

        l = Level(b->lchild, x, h + 1); //在左子树中查找

        if (l != 0)

            return (l);

        else //在左子树中未找到,再在右子树中查找

            return (Level(b->rchild, x, h + 1));

    }

}

int LevelK(BTNode \*b, int k)

{

    if (b == NULL)

        return 0;

    if (k == 1)

        return 1;

    return LevelK(b->lchild, k - 1) + LevelK(b->rchild, k - 1);

}

void DispBTree(BTNode \*b) //输出二叉树所有结点

{

    if (b != NULL)

    {

        printf("%c ", b->data);

        if (b->lchild != NULL || b->rchild != NULL)

        {

            DispBTree(b->lchild); //递归处理左子树

            DispBTree(b->rchild); //递归处理右子树

        }

    }

}

void DisLeafNodes(BTNode \*b) //输出二叉树b的叶子结点

{

    if (b != NULL)

    {

        if ((b->lchild == NULL) && (b->rchild == NULL))

            printf("%c ", b->data);

        DisLeafNodes(b->lchild);

        DisLeafNodes(b->rchild);

    }

}

bool DispAncestor(BTNode \*b, ElemType c) //输出二叉树b中值为c的所有祖先结点

{

    if (b == NULL)

        return false;

    else if ((b->lchild != NULL && b->lchild->data == c) || (b->rchild != NULL && b->rchild->data == c))

    {

        printf("%c ", b->data);

        return true;

    }

    else if (DispAncestor(b->lchild, c) || DispAncestor(b->rchild, c))

    {

        printf("%c ", b->data);

        return true;

    }

    else

        return false;

}

int main()

{

    BTNode \*b;

    CreateBTree(b, "A(B(D(,G)),C(E,F))");

    printf("二叉树b的所有结点:");

    DispBTree(b);

    printf("\n二叉树b的总的结点个数:%d\n", Nodes(b));

    printf("二叉树b的所有叶子结点:");

    DisLeafNodes(b);

    printf("\n二叉树b的总的叶子结点个数:%d\n", LeafNodes(b));

    printf("二叉树b中值为%c的结点的层次:%d\n", 'E', Level(b, 'E', 1));

    printf("二叉树b第3层的结点个数:%d\n", LevelK(b, 3));

    printf("二叉树b中值为%c的所有祖先结点为:", 'E');

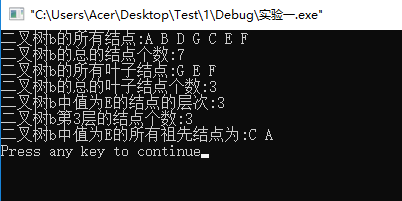
    DispAncestor(b, 'E');

    DestroyBTree(b);

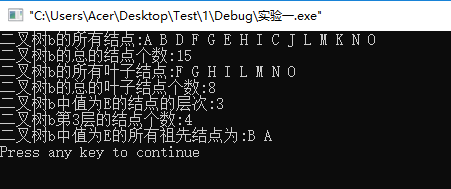
    return 0;

}

**运行结果:**

****

**修改二叉树b为: A(B(D(F,G),E(H,I)),C(J(L,M),K(N,O))),调试结果如下:**



实验2.编写一个程序exp-7-4.cpp，给出根据先序序列和中序序列构建二叉树的算法。编写main函数调用，并以以上二叉树的先序序列和中序序列作为输入实参字符数组，进行验证。

验证方法：可以输出构建的二叉树（二叉树的括号表示法对应的字符串）验证算法是否正确。

**程序：**

#include <stdio.h>

#include <malloc.h>

#define MaxSize 100

#define MaxWidth 40

typedef char ElemType;

typedef struct node

{

    ElemType data;       //数据元素

    struct node \*lchild; //指向左孩子结点

    struct node \*rchild; //指向右孩子结点

} BTNode;

BTNode \*CreateBT(char \*pre, char \*in, int n)

{

    BTNode \*b;

    char \*p;

    int k;

    if (n <= 0)

        return NULL;

    b = (BTNode \*)malloc(sizeof(BTNode)); //创建二叉树结点\*b

    b->data = \*pre;

    for (p = in; p < in + n; p++)                     //在中序序列中找等于\*pre的位置k

        if (\*p == \*pre)                                  //pre指向根结点

            break;                                       //在in中找到后退出循环

    k = p - in;                                          //确定根结点在in中的位置

    b->lchild = CreateBT(pre + 1, in, k);                //递归构造左子树

    b->rchild = CreateBT(pre + k + 1, p + 1, n - k - 1); //递归构造右子树

    return b;

}

void DestroyBTree(BTNode \*&b) //销毁二叉树

{

    if (b != NULL)

    {

        DestroyBTree(b->lchild);

        DestroyBTree(b->rchild);

        free(b);

    }

}

void DispBTree(BTNode \*b) //以括号表示法输出二叉树

{

    if (b != NULL)

    {

        printf("%c", b->data);

        if (b->lchild != NULL || b->rchild != NULL)

        {

            printf("(");          //有孩子结点时才输出(

            DispBTree(b->lchild); //递归处理左子树

            if (b->rchild != NULL)

                printf(",");      //有右孩子结点时才输出,

            DispBTree(b->rchild); //递归处理右子树

            printf(")");          //有孩子结点时才输出)

        }

    }

}

int main()

{

    BTNode \*b;

    ElemType pre[] = "ABDGCEF";

    ElemType in[] = "DGBAECF";

    int n = 7;

    b = CreateBT(pre, in, n);

    printf("先序序列:%s\n", pre);

    printf("中序序列:%s\n", in);

    printf("构造的二叉树b为:");

    DispBTree(b);

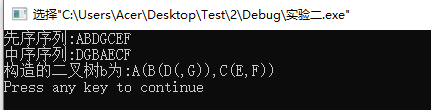
    DestroyBTree(b);

    printf("\n");

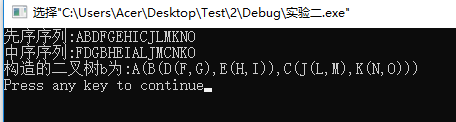
    return 0;

}

**运行结果:**



**修改二叉树b的先序序列为’ABDFGEHICJLMKNO’，后序序列为为:’** **FDGBHEIALJMCNKO’，调试结果如下:**



实验3.编写一个程序exp-7-5.cpp，给出根据中序序列和后序序列构建二叉树的算法。编写main函数调用，并以以上二叉树的中序序列和后序序列作为输入实参字符数组，进行验证。

验证方法：可以输出构建的二叉树（二叉树的括号表示法对应的字符串）验证算法是否正确。

**程序：**

//exp7-5.cpp

#include <stdio.h>

#include <malloc.h>

#define MaxSize 100

#define MaxWidth 40

typedef char ElemType;

typedef struct node

{

    ElemType data;       //数据元素

    struct node \*lchild; //指向左孩子结点

    struct node \*rchild; //指向右孩子结点

} BTNode;

BTNode \*CreateBT2(char \*post, char \*in, int n)

{

    BTNode \*b;

    char r, \*p;

    int k;

    if (n <= 0)

        return NULL;

    r = \*(post + n - 1);                  //根结点值

    b = (BTNode \*)malloc(sizeof(BTNode)); //创建二叉树结点\*b

    b->data = r;

    for (p = in; p < in + n; p++) //在in中查找根结点

        if (\*p == r)

            break;

    k = p - in;                                        //k为根结点在in中的下标

    b->lchild = CreateBT2(post, in, k);                //递归构造左子树

    b->rchild = CreateBT2(post + k, p + 1, n - k - 1); //递归构造右子树

    return b;

}

void DestroyBTree(BTNode \*&b) //销毁二叉树

{

    if (b != NULL)

    {

        DestroyBTree(b->lchild);

        DestroyBTree(b->rchild);

        free(b);

    }

}

void DispBTree(BTNode \*b) //以括号表示法输出二叉树

{

    if (b != NULL)

    {

        printf("%c", b->data);

        if (b->lchild != NULL || b->rchild != NULL)

        {

            printf("(");          //有孩子结点时才输出(

            DispBTree(b->lchild); //递归处理左子树

            if (b->rchild != NULL)

                printf(",");      //有右孩子结点时才输出,

            DispBTree(b->rchild); //递归处理右子树

            printf(")");          //有孩子结点时才输出)

        }

    }

}

int main()

{

    BTNode \*b;

    ElemType in[] = "DGBAECF";

    ElemType post[] = "GDBEFCA";

    int n = 7;

    printf("中序序列:%s\n", in);

    printf("后序序列:%s\n", post);

    b = CreateBT2(post, in, n);

    printf("构造一棵二叉树b:\n");

    printf(" 括号表示法:");

    DispBTree(b);

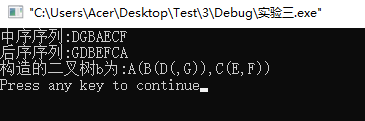
    DestroyBTree(b);

    printf("\n");

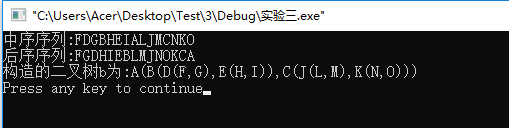
    return 0;

}

**运行结果:**



**修改二叉树b的中序序列为FDGBHEIALJMCNKO，后序序列为: FGDHIEBLMJNOKCA,调试结果如下:**



实验4.编写一个程序exp-7-6.cpp，给出构建中序线索二叉树的算法以及中序遍历线索二叉树的递归算法并输出线索二叉树的中序序列，并以上图给出的二叉树b为例，对算法进行验证。

程序：

//文件名:exp7-6.cpp

#include <stdio.h>

#include <malloc.h>

#define MaxSize 100

typedef char ElemType;

typedef struct node

{

    ElemType data;

    int ltag, rtag;      //增加的线索标记

    struct node \*lchild; //左孩子指针

    struct node \*rchild; //右孩子指针

} TBTNode;

void CreateTBTree(TBTNode \*&b, const char \*str) //由str串建立含空线索域的二叉链b

{

    TBTNode \*St[MaxSize], \*p = NULL;

    int top = -1, k, j = 0;

    char ch;

    b = NULL; //建立的二叉树初始时为空

    ch = str[j];

    while (ch != '\0') //str未扫描完时循环

    {

        switch (ch)

        {

        case '(':

            top++;

            St[top] = p;

            k = 1;

            break; //为左结点

        case ')':

            top--;

            break;

        case ',':

            k = 2;

            break; //为右结点

        default:

            p = (TBTNode \*)malloc(sizeof(TBTNode));

            p->data = ch;

            p->lchild = p->rchild = NULL;

            if (b == NULL) //p为二叉树的根结点

                b = p;

            else //已建立二叉树根结点

            {

                switch (k)

                {

                case 1:

                    St[top]->lchild = p;

                    break;

                case 2:

                    St[top]->rchild = p;

                    break;

                }

            }

        }

        j++;

        ch = str[j];

    }

}

void DispTBTree(TBTNode \*b) //输出含空线索域的二叉树b

{

    if (b != NULL)

    {

        printf("%c", b->data);

        if (b->lchild != NULL || b->rchild != NULL)

        {

            printf("(");

            DispTBTree(b->lchild);

            if (b->rchild != NULL)

                printf(",");

            DispTBTree(b->rchild);

            printf(")");

        }

    }

}

TBTNode \*pre;            //全局变量

void Thread(TBTNode \*&p) //中序线索化二叉树，被CreateThread调用

{

    if (p != NULL)

    {

        Thread(p->lchild);     //左子树线索化

        if (p->lchild == NULL) //若p结点的左指针为空

        {

            p->lchild = pre; //建立当前结点的前驱线索

            p->ltag = 1;

        }

        else

            p->ltag = 0;

        if (pre->rchild == NULL) //若p结点的右指针为空

        {

            pre->rchild = p; //建立前驱结点的后继线索

            pre->rtag = 1;

        }

        else

            pre->rtag = 0;

        pre = p;

        Thread(p->rchild); //右子树线索化

    }

}

TBTNode \*CreateThread(TBTNode \*b) //创建中序线索化二叉树

{

    TBTNode \*root;

    root = (TBTNode \*)malloc(sizeof(TBTNode)); //创建根结点

    root->ltag = 0;

    root->rtag = 1;

    root->rchild = b;

    if (b == NULL) //空二叉树

        root->lchild = root;

    else

    {

        root->lchild = b;

        pre = root;         //pre结点是p结点的前驱结点,供加线索用

        Thread(b);          //中序遍历线索二叉树

        pre->rchild = root; //最后处理,加入指向根结点的线索

        pre->rtag = 1;

        root->rchild = pre; //根结点右线索化

    }

    return root;

}

void InOrder(TBTNode \*tb) //被ThInOrder算法调用

{

    if (tb->lchild != NULL && tb->ltag == 0) //有左孩子

        InOrder(tb->lchild);

    printf("%c ", tb->data);

    if (tb->rchild != NULL && tb->rtag == 0) //有右孩子

        InOrder(tb->rchild);

}

void ThInOrder(TBTNode \*tb) //中序线索二叉树的中序遍历递归算法

{

    InOrder(tb->lchild);

}

void DestroyTBTree1(TBTNode \*tb) //被DestroyTBTree算法调用

{

    if (tb != NULL)

    {

        if (tb->lchild != NULL && tb->ltag == 0) //有左孩子

            DestroyTBTree1(tb->lchild);

        if (tb->rchild != NULL && tb->rtag == 0) //有右孩子

            DestroyTBTree1(tb->rchild);

        free(tb);

    }

}

void DestroyTBTree(TBTNode \*tb) //释放中序线索二叉树的所有结点

{

    DestroyTBTree1(tb->lchild);

    free(tb);

}

int main()

{

    TBTNode \*b, \*tb;

    CreateTBTree(b, "A(B(D(,G)),C(E,F))");

    printf("二叉树:");

    DispTBTree(b);

    printf("\n");

    tb = CreateThread(b);

    printf("线索中序序列:\n");

    printf("递归算法:");

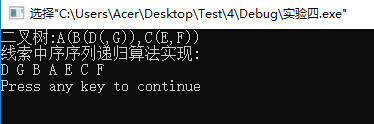
    ThInOrder(tb);

    DestroyTBTree(tb);

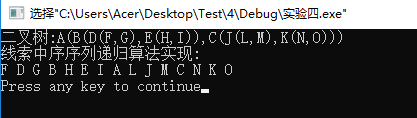
    return 0;

}

**运行结果:**



**修改二叉树b为: A(B(D(F,G),E(H,I)),C(J(L,M),K(N,O))),调试结果如下:**



实验5.问题：假设一报文中只有如下4中单词，分别在报文中出现的频次如下表所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 单词 | The | a | and | at |
| 频次 | 92 | 47 | 68 | 22 |

请给这段报文进行编码，要求编码总长度最短。

以以上报文编码问题为例，每个单词为结点，每个单词在报文中出现的频次为结点权重，构造哈夫曼树并对哈夫曼树进行编码实现问题求解。完成以下任务：

编写一个程序exp-7-7.cpp，实现 ① 给出构造一棵哈夫曼树的算法(权最小的作为左孩子，权次小的作为右孩子)；② 对哈夫曼树进行编码，并得到每个单词的哈夫曼编码；③ 输出每个单词的哈夫曼编码；④ 求构造的哈夫曼树的带权路径长度WPL（即报文编码总长度）并输出。⑤ 求出哈夫曼编码的平均长度（等于报文编码总长度WPL除以报文中单词总数（等于每个单词出现的次数之和））并输出。

提示：每个单词可以看做是字符串，所有单词可以通过字符指针数组的数组元素指向各字符串。

char \*str[]={"The","a","and","at"};

double fnum[]={92,47,68,22};

**程序：**

//文件名:exp7-7.cpp

#include <stdio.h>

#include <string.h>

#define N 50        //叶子结点数

#define M 2 \* N - 1 //树中结点总数

typedef struct

{

    char data[5]; //结点值

    int weight;   //权重

    int parent;   //双亲结点

    int lchild;   //左孩子结点

    int rchild;   //右孩子结点

} HTNode;

typedef struct

{

    char cd[N]; //存放哈夫曼码

    int start;

} HCode;

void CreateHT(HTNode ht[], int n) //由ht的叶子结点构造完整的哈夫曼树

{

    int i, k, lnode, rnode;

    int min1, min2;

    for (i = 0; i < 2 \* n - 1; i++) //所有结点的相关域置初值-1

        ht[i].parent = ht[i].lchild = ht[i].rchild = -1;

    for (i = n; i < 2 \* n - 1; i++) //构造哈夫曼树的分支结点

    {

        min1 = min2 = 32767; //lnode和rnode为最小权重的两个结点位置

        lnode = rnode = -1;

        for (k = 0; k <= i - 1; k++) //查找最小和次小的结点

            if (ht[k].parent == -1)  //只在尚未构造二叉树的结点中查找

            {

                if (ht[k].weight < min1)

                {

                    min2 = min1;

                    rnode = lnode;

                    min1 = ht[k].weight;

                    lnode = k;

                }

                else if (ht[k].weight < min2)

                {

                    min2 = ht[k].weight;

                    rnode = k;

                }

            }

        ht[lnode].parent = i;

        ht[rnode].parent = i; //合并两个最小和次小的结点

        ht[i].weight = ht[lnode].weight + ht[rnode].weight;

        ht[i].lchild = lnode;

        ht[i].rchild = rnode;

    }

}

void CreateHCode(HTNode ht[], HCode hcd[], int n) //由哈夫曼树ht构造哈夫曼编码hcd

{

    int i, f, c;

    HCode hc;

    for (i = 0; i < n; i++) //根据哈夫曼树构造所有叶子结点的哈夫曼编码

    {

        hc.start = n;

        c = i;

        f = ht[i].parent;

        while (f != -1) //循环直到树根结点

        {

            if (ht[f].lchild == c) //处理左孩子结点

                hc.cd[hc.start--] = '0';

            else //处理右孩子结点

                hc.cd[hc.start--] = '1';

            c = f;

            f = ht[f].parent;

        }

        hc.start++; //start指向哈夫曼编码最开始字符

        hcd[i] = hc;

    }

}

void DispHCode(HTNode ht[], HCode hcd[], int n) //输出哈夫曼编码

{

    int i, k;

    int sum = 0, m = 0, j;

    printf("输出哈夫曼编码:\n");

    for (i = 0; i < n; i++)

    {

        j = 0;

        printf("      %s:\t", ht[i].data);

        for (k = hcd[i].start; k <= n; k++)

        {

            printf("%c", hcd[i].cd[k]);

            j++;

        }

        m += ht[i].weight;

        sum += ht[i].weight \* j;

        printf("\n");

    }

    printf("总长度=%d\n",sum);

    printf("平均长度=%g\n", 1.0 \* sum / m);

}

int main()

{

    int n = 4, i;

    const char \*str[] = {"The", "a", "and", "at"};

    int fnum[] = {92, 47, 68, 22};

    HTNode ht[M];

    HCode hcd[N];

    for (i = 0; i < n; i++)

    {

        strcpy(ht[i].data, str[i]);

        ht[i].weight = fnum[i];

    }

    CreateHT(ht, n);

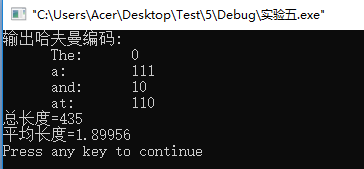
    CreateHCode(ht, hcd, n);

    DispHCode(ht, hcd, n);

    return 0;

}

**运行结果:**

****