

# 北京交通大学

## 《数据结构（A）》第 6 章

### “树与二叉树”基本作业与设计作业

专    业： 计算机科学与技术

班    级：                     

学生姓名：                     

学    号：                     

北京交通大学计算机与信息技术学院

2021 年 11 月 20 日



## 《数据结构（A）》第 6 章基本作业<sup>①</sup>

提醒同学：本章基本作业与设计型作业合并在一起，即没有另外专门的设计作业题目。

### 1 作业题目

6.1 按先序遍历的扩展序列建立一棵二叉树的二叉链表存储结构。注意，关于“先序遍历的扩展序列”请参考课程幻灯片的第 92 屏至第 94 屏。

6.2 分别实现二叉树先序、中序、后序遍历的递归算法。

6.3（参见教材：《数据结构（C 语言版）》，第 6 章，第 130 页与第 131 页）实现二叉树中序遍历的非递归算法。

6.4（《数据结构题集（C 语言版）》，第 6 章，第 41 页，第 6.27 题，难度系数为 3）假设一颗二叉树的先序遍历序列为 EBADCFHGIKJ 和中序遍历序列为 ABCDEFGHIJK，

（1）请画出该树。说明你所采用的画图工具软件名称，要求在你所采用的画图工具软件之下，多画出的图示是可修改的。

（2）设计与实现你的算法，并设计一种简明的输出一棵二叉树的方式。

6.5（《数据结构题集（C 语言版）》，第 6 章，第 41 页，第 6.28 题，难度系数为 3）假设一颗二叉树的中序序列为 DCBGEAHFIJK 和后序序列为 DCEGBFHKJIA，

---

<sup>①</sup> 这是《数据结构（A）》第 5 章的基本作业，第 11 月 15 日发布，学生提交的截止日期是 2021 年 11 月 28 日。

(1) 请画出该树。说明你所采用的画图工具软件名称，要求在你所采用的画图工具软件之下，多画出的图示是可修改的。

(2) 设计与实现你的算法，并设计一种简明的输出一棵二叉树的方式。

**6.6** (《数据结构题集 (C 语言版)》，第 6 章，第 44 页，第 6.65 题，难度系数为 4) 已知 一颗二叉树的前序序列和中序序列分别存于两个一维数组中，试编写算法建立该二叉树的二叉链表。

要求：提供两个结点数目不少于 10 个的不同的典型二叉树，来测试你所设计的程序。

**6.7** (《数据结构题集 (C 语言版)》，第 6 章，第 42 页，第 6.29 题，难度系数为 3) 假设一颗二叉树的层序遍历序列为 **ABCDEFGHIJ** 和中序遍历序列为 **DBGHJACIF**，

(1) 请画出该树。说明你所采用的画图工具软件名称，要求在你所采用的画图工具软件之下，多画出的图示是可修改的。

(2) 设计与实现你的算法，并设计一种简明的输出一棵二叉树的方式。

**6.8** (《数据结构题集 (C 语言版)》，第 6 章，第 43 页，第 6.47 题，难度系数为 4) 实现二叉树层次遍历的非递归算法，并总结应用队列解决问题的基本模式。

**6.9** 应用后序遍历方式，求二叉树的深度。

请参考课程幻灯片的第 90 屏与第 91 屏。

**6.10** (《数据结构题集 (C 语言版)》，第 6 章，第 44 页，第 6.66 题，难度系数为 4) 假设有  $n$  个结点的树  $T$  采用了双亲表示法，写出由此建立一般树的二叉树表示法的存储结构，即孩子-兄弟 (链表) 表示法，或二叉链表表示法 (参见教材：《数据结构 (C 语言版)》，第 6 章，第 136 页)。

要求：提供两个结点数目不少于 10 个的不同的树，来测试你所设计的程序。

6.11 《数据结构题集 (C 语言版)》, 第 6 章, 第 44 页, 第 6.63 题, 难度系数为 3) 求树的深度。

要求: 提供两个结点数目不少于 10 个的不同的树, 来测试你所设计的程序。

6.12 《数据结构题集 (C 语言版)》, 第 6 章, 第 44 页, 第 6.68 题, 难度系数为 3) 已知一颗树的由根至叶子结点按层次输入的结点序列及每个结点的度 (每层中自左至右输入), 试写出构造此树的孩子-兄弟链表的算法。

6.13 采用自然语言或伪代码形式, 分析与总结以下算法:

- (1) 二叉树先序遍历方式的线索化算法;
- (2) 二叉树中序遍历方式的线索化算法;
- (3) 二叉树后序遍历方式的线索化算法;
- (4) 对于先序线索化二叉树, 如何求任意一个结点的前驱结点;
- (5) 对于先序线索化二叉树, 如何求任意一个结点的后继结点;
- (6) 对于中序线索化二叉树, 如何求任意一个结点的前驱结点;
- (7) 对于中序线索化二叉树, 如何求任意一个结点的后继结点;
- (8) 对于后序线索化二叉树, 如何求任意一个结点的前驱结点;
- (9) 对于后序线索化二叉树, 如何求任意一个结点的后继结点。

6.14 《数据结构题集 (C 语言版)》, 第 6 章, 第 41 页, 第 6.26 题, 难度系数为 3) 假设用于通信的电文仅由 8 个字母组成, 字母在电文中出现的频率分别为 0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10。试为这 8 个字母设计哈夫曼编码。使用 0~7 的二进制表示形式是另一种编码方案, 对于上述实例, 比较两种方案的优缺点。

特别说明: 本章作业还是比较多的, 而且有一定难度。

## 2 作业题目解答

部分题目过于简单且为上课原样内容，其思路和调试结果略去。

**【6.1 题解答】：**根据本课程幻灯片的第 92 屏至第 94 屏，

思路：

按照 ppt 讲义正常建立 struct 和 creat\_tree()函数。

代码：

```
01:int creat_BiTree(BiTNode *T)
02:{
03:    char ch;
04:    scanf(&ch);
05:    if(ch == '!') T=NULL;
06:    else
07:    {
08:        if(!(T=(BiTNode*)malloc(sizeof(BiTNode))));
09:        exit(0);
10:        T->data = ch;
11:        creat_BiTree(T->lchild);
12:        creat_BiTree(T->rchild);
13:    }
14:    return 1;
15:}
```

注：ppt 中以空格符作为表示空的标志，实际中 scanf 无法读入空白符，故使用 “!” 进行替代。

**【6.2 题解答】：**

(1) 二叉树先序遍历的递归算法：

```
01:void preorder(BiTNode *T)
02:{
03:    if(T)
04:    {
05:        visit_tree(T->data);
06:        preorder(T->lchild);
07:        preorder(T->rchild);
08:    }
```

```
09:}
```

(2) 二叉树中序遍历的递归算法:

```
01:void inorder(BiTNode *T)
02:{
03:    if(T)
04:    {
05:        inorder(T->lchild);
06:        visit_tree(T->data);
07:        inorder(T->rchild);
08:    }
09:}
```

(3) 二叉树后序遍历的递归算法:

```
01:void postorder(BiTNode *T)
02:{
03:    if(T)
04:    {
05:        postorder(T->lchild);
06:        postorder(T->rchild);
07:        visit_tree(T->data);
08:    }
09:}
```

**【6.3 题解答】:** (参见教材:《数据结构 (C 语言版)》, 第 6 章, 第 130 页与第 131 页) 实现二叉树中序遍历的非递归算法。

```
01:void push(BiTNode **a, BiTNode *elem)
02:{
03:    a[++top] = elem;
04:}
05:void pop()
06:{
07:    if(top == -1)
08:        return;
09:    --top;
10:}
11:BiTNode * getTop(BiTNode **a)
```

```

12:{
13:    return a[top];
14:}
15:
16:void inOrderTraverse(BiTNode *T)
17:{
18:    BiTNode *a[200];
19:    BiTNode *p;
20:    push(a, T);
21:    while(top!=-1)
22:    {
23:        while((p = getTop(a))&& p){
24:            push(a ,p->lchild); // has no lchild, thus push NULL to stack
25:        }
26:        pop(); // when circulation ending, the top elem is certainly NULL
27:        if(top != -1)
28:        {
29:            p = getTop(a);
30:            pop();
31:            visit_tree(p);
32:            push(a, p->rchild);
33:        }
34:    }
35:}
36:

```

**【6.4 题解答】：**《数据结构题集（C 语言版）》，第 6 章，第 41 页，第 6.27 题，难度系数为 3。

(1) 画出该树：

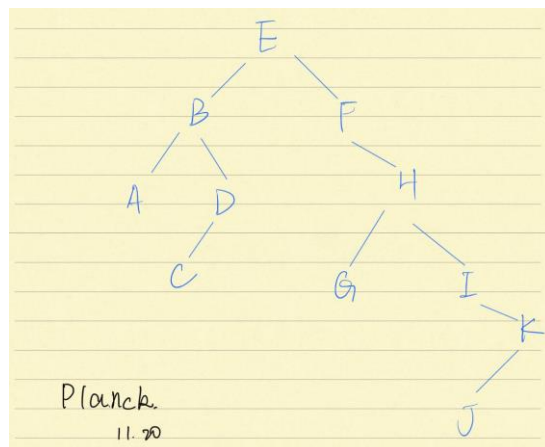


图 6.4



软件 notability11.05 似乎无图例（可为什么要有图例呢）

（2）算法与输出：

```
01:char * cut_string(char *string, int start, int count)
02:{
03:    char *tem= (char*) malloc(sizeof(char)*11);
04:    for(int i= start, j =0; j<count; ++i, ++j)
05:    {
06:        tem[j] = string[i];
07:    }
08:    tem[count] = '\0';
09:    return tem;
10:}
11:
12:BiTNode * pre_in_to_tree(char *pre, char*in)
13:{
14:    BiTNode * tem = (BiTNode *) malloc(sizeof (BiTNode));
15:    char p= pre[0];
16:    tem->data = p;
17:
18:    for(int i=0 ; i<strlen(in); ++i)
19:        if(p== in[i])
20:        {
21:            if(i==0) tem->lchild=NULL;
22:            else
23:                tem->lchild = pre_in_to_tree( cut_string(pre, 1, i),
cut_string(in, 0, i));
24:            if(strlen(in)==i+1) tem->rchild= NULL;
25:            else
26:                tem->rchild = pre_in_to_tree( cut_string( pre, i+1,
strlen(pre)-i-1), cut_string(in, i+1, strlen(in)-i-1));
27:            break;
28:        }
29:    return tem;
30:
31:}
32:///
33://show bitree by curcusion
34:void show_bitree(BiTNode *T)
35:{
36:    if(T==NULL) printf("");
37:    else
```

```
38:  {
39:      printf("%c", T->data);
40:      printf("(");
41:      show_bitree(T->lchild);
42:      printf(" ");
43:      show_bitree(T->rchild);
44:      printf(")");
45:  }
46:}
47:// mian function to debug sample
48:int main() {
49:    char pre[20];
50:    strcpy(pre, "EBADCFHGIJK");
51:    char in[20];
52:    strcpy(in, "ABCDEFGHGIJK");
53:    BiTNode *T;
54:    T = pre_in_to_tree( pre, in);
55:    inorder(T);
56:    printf("\n");
57:
```

调试结果:



```
A B C D E F G H I J K
进程已结束, 退出代码为 0
```

图 6.4

调试正常

**【6.5 题解答】:**《数据结构题集 (C 语言版)》, 第 6 章, 第 41 页, 第 6.28 题, 难度系数为 3。

(1) 画出该树:

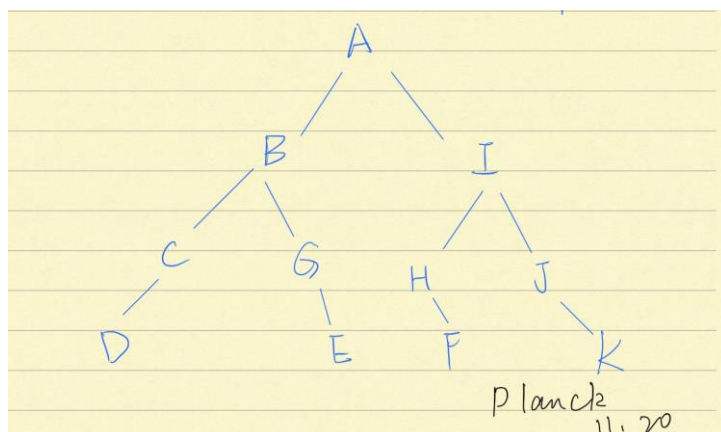


图 6.5

(2) 算法与输出

```

01: void show_bitree(BitNode *T)
02: {
03:     if(T==NULL) printf("");
04:     else
05:     {
06:         printf("%c", T->data);
07:         printf("(");
08:         show_bitree(T->lchild);
09:         printf(" ");
10:         show_bitree(T->rchild);
11:         printf(")");
12:     }
13: }
14: // main function for debug sample
15: char pre[20];
16: strcpy(pre, "EBADCFHGIKJ");
17: char in[20];
18: strcpy(in, "ABCDEFGHGIJK");
19: BitNode *T;
20: T = pre_in_to_tree( pre, in);
21: // inorder(T);
22: printf("\n");
23: // 6.5
24: char * cut_string(char *string, int start, int count)
25: {
26:     char *tem= (char*) malloc(sizeof(char)*11);
27:     for(int i= start, j =0; j<count; ++i, ++j)
  
```

```

28:  {
29:      tem[j] = string[i];
30:  }
31:  tem[count] = '\0';
32:  return tem;
33:}
34:
35:// ust recursion algorithm, and test sample is from question 6.5
36:// output algorithm is by brand sequence, which is written above.
37:BiTNode * in_post_to_bitree(char *in, char *post)
38:{
39:    if(strlen(in) == 0) return NULL;
40:    BiTNode *tem = (BiTNode *) malloc(sizeof (BiTNode));
41:    char p= post[strlen(post)-1];
42:    tem->data = p;
43:
44:    for(int i=0; i<strlen(in) ; ++i)
45:    {
46:        if(p == in[i])
47:        {
48:            if(i == 0) tem->lchild =NULL;
49:            else
50:                tem->lchild = in_post_to_bitree(cut_string(in, 0, i),
cut_string(post, 0, i));
51:            if(i == strlen(in)-1) tem->rchild = NULL;
52:            else
53:                tem->rchild = in_post_to_bitree( cut_string( in, i+1,
strlen(in)-i-1), cut_string(post, i, strlen(in)-i-1));
54:            break;
55:        }
56:    }
57:    return tem;
58:}

```

```

E(B(A(* *) D(C(* *) *)) F(* H(G(* *) I(* K(J(* *) *))))))
进程已结束, 退出代码为 0

```

图 6.5-1

```

D.(B(J(I(subject(sophomore(data structure(chapter_8(make=01
A(B(C(D(* *) *) G(* E(* *))) I(H(* F(* *)) J(* K(* *))))))
进程已结束, 退出代码为 0

```

图 6.5-2

【6.6 题解答】:《数据结构题集 (C 语言版)》,第 6 章,第 44 页,第 6.65 题,难度系数为 4。

算法:

```
01:char * cut_string(char *string, int start, int count)
02:{
03:    char *tem= (char*) malloc(sizeof(char)*11);
04:    for(int i= start, j =0; j<count; ++i, ++j)
05:    {
06:        tem[j] = string[i];
07:    }
08:    tem[count] = '\0';
09:    return tem;
10:}
11:BiTNode * pre_in_to_tree(char *pre, char*in)
12:{
13:    BiTNode * tem = (BiTNode* ) malloc(sizeof (BiTNode));
14:    char p= pre[0];
15:    tem->data = p;
16:
17:    for(int i=0 ; i<strlen(in); ++i)
18:        if(p== in[i])
19:        {
20:            if(i==0) tem->lchild=NULL;
21:            else
22:                tem->lchild = pre_in_to_tree(cut_string(pre, 1, i),
cut_string(in, 0, i));
23:            if(strlen(in)==i+1) tem->rchild= NULL;
24:            else
25:                tem->rchild = pre_in_to_tree(cut_string( pre, i+1,
strlen(pre)-i-1), cut_string(in, i+1, strlen(in)-i-1));
26:            break;
27:        }
28:    return tem;
29:}
30:
31:
32:int main() {
33:    char pre[20];
34:    strcpy(pre,"EBADCFHGIKJ");
35:    char in[20];
```

```
36: strcpy(in, "ABCDEFGHIIJK");
37: BiTNode *T;
38: T = pre_in_to_tree(pre, in);
39: inorder(T);
40:
41: printf("\n");
42: strcpy(pre, "ABCDGEIHFJK");
43: strcpy(in, "DCBGEAHFIJK");
44: T = pre_in_to_tree(pre, in);
45: inorder(T);
46: }
```

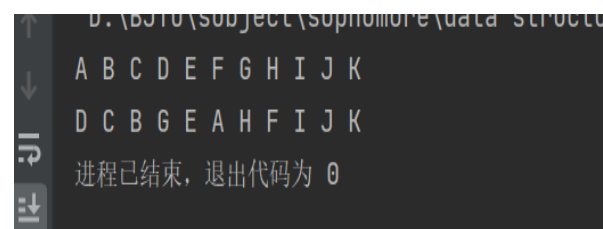
(1) 测试实例 1:



```
D:\B310\subject\sophomore\data_structu
A B C D E F G H I J K
D C B G E A H F I J K
进程已结束, 退出代码为 0
```

图 6.6.1

(2) 测试实例 2:



```
D:\B310\subject\sophomore\data_structu
A B C D E F G H I J K
D C B G E A H F I J K
进程已结束, 退出代码为 0
```

图 6.6.2

结果分析:

两次示例同时输出, 故图使用同一张。

采用中序输出显示，所以与代码中两次 in 字符串匹配成功说明正。两棵树为 6.4 和 6.5 题目。

**【6.7 题解答】：**《数据结构题集 (C 语言版)》，第 6 章，第 42 页，第 6.29 题，难度系数为 3。

(1) 画出该树：

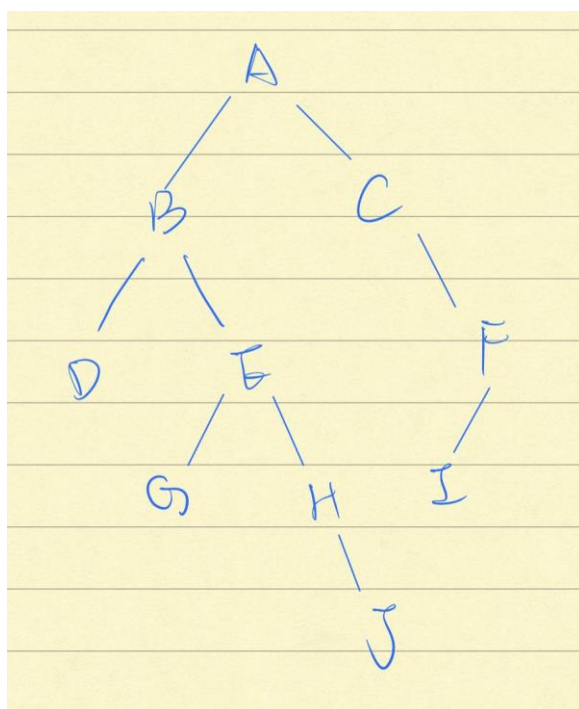


图 6.7

(2) 算法及其输出：

```
01:// by traversing the level_ and in_, find the lchild' level_ and rhcild's
02:level_. then recursive it
03:BiTNode * level_in_to_bitree(char *level, char *in)
04:{
05:    if(strlen(level) == 0 ) return NULL;
06:    BiTNode * tem = (BiTNode*) malloc(sizeof (BiTNode));
07:    char left[15];
08:    char right[15];
09:    int count_left;
10:    int count_right;
```

---

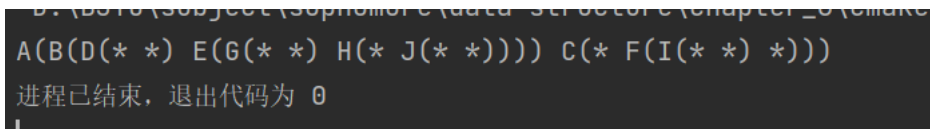
```
11:   count_left = count_right=0;
12:
13:   tem->data = level[0];
14:   char p = level[0];
15:
16:   for(int i =0; i<strlen(in); ++i)
17:   {
18:       if(in[i] == p)
19:       {
20:           for(int j=1; j<=strlen(level); ++j )
21:           {
22:               int flag=0;
23:               for(int k=0; k<i; ++k){
24:                   if(level[j] == in[k])
25:                   {
26:                       left[count_left++] = level[j];
27:                       flag = 1;
28:                       break;
29:                   }
30:               }
31:               if(flag ==0){
32:                   right[count_right++] = level[j];
33:               }
34:           }
35:
36:           //if(count_left == i) break;
37://
38://       for(int j=1; j<=strlen(level); ++j )
39://           for(int k=i+1; k<=strlen(in)-1; ++k)
40://               if(level[j] == in[k])
41://                   {
42://                       right[count_right++] = level[j];
43://                       break;
44://                   }
45:// v1 is a bit of complicated
46:           //if(count_right == strlen(in)-i-1) break;
47:           left[count_left]= '\0';
48:           right[count_right] = '\0';
49:
50:           tem->lchild = level_in_to_bitree(left, cut_string(in, 0, i));
51:           tem->rchild = level_in_to_bitree(right, cut_string(in, i+1 ,
           strlen(in)-i-1));
52:           break;
```



```

53:     }
54: }
55: return tem;
56:}
57:// test sample is from this question
58:
59:char level[15] ="ABCDEFGHJIJ";
60:char in[15] = "DBGEHJACIF";
61:BiTNode * T = level_in_to_bitree(level, in);
62:show_bitree(T);

```



```

A(B(D(* *) E(G(* *) H(* J(* *) ))) C(* F(I(* *) *)))
进程已结束, 退出代码为 0

```

图 6.7

**【6.8 题解答】:**《数据结构题集 (C 语言版)》, 第 6 章, 第 43 页, 第 6.47 题, 难度系数为 4。

非递归算法:

```

01:void level_traversal(BiTNode *T)
02:{ //just a very simple queue struct, which locate by two int front and rear
03:    BiTNode queue[100];
04:    int front, rear;
05:    front = rear =0;
06:    BiTNode tem;
07:    if(T == NULL) return;
08:    else
09:    {
10:        queue[rear] = *T;
11:        ++ rear;
12:        do {
13:            tem = queue[front];
14:            printf("%c ", tem.data); // jump out queue, then print it
15:            ++ front; // general operator
16:            if(tem.lchild != NULL)
17:            {
18:                queue[rear] = *tem.lchild; // out one, in tow
19:                ++ rear; // general operator
20:            }
21:            if(tem.rchild != NULL)

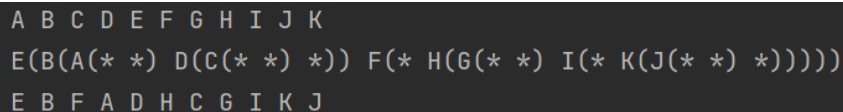
```

```

22:         {
23:             queue[rear] = *tem.rchlid;
24:             ++rear;
25:         }
26:     } while (front != rear);
27: }
28:
29:}
30:// main function. The test sample is question 1 and 2
31: char pre[20];
32: strcpy(pre, "EBADCFHGIKJ");
33: char in[20];
34: strcpy(in, "ABCDEFGHGIJK");
35: BiTNode *T;
36: T = pre_in_to_tree( pre, in);
37: show_bitree(T);
38: printf("\n"); // if the show_bitree is equal to level_traversal, the
39:// algorithm is perfect 🍷
40: level_traversal(T);
41: printf("\n");

```

Debug:



```

A B C D E F G H I J K
E(B(A(* *) D(C(* *) *)) F(* H(G(* *) I(* K(J(* *) *))))
E B F A D H C G I K J

```

图 6.8

总结应用队列解决问题的基本模式:

设定出队访问或入队访问。元素出队时可以加入几个相关元素, 设定好结束条件, 即 **front = rear** (这里只讨论最简单情况)。队列结构只是解决问题的工具, 其算法最关键的还是找到元素出队时再入队的元素及其模式。

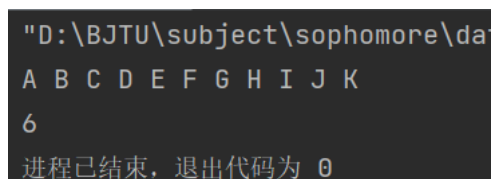
**【6.9 题解答】:** 根据课程幻灯片的第 90 屏与第 91 屏, 应用后序遍历方式,

求二叉树深度的算法如下：

```

01:int depth(BiTNode *T)
02:{
03:    if(!T) return 0;
04:    else
05:    {
06:        int left = depth(T->lchild);
07:        int right = depth(T->rchild);
08:        return 1 + (left >= right ? left : right);
09:    }
10;}
11:// main function. The test sample is the former Tree mentioned above

```



```

"D:\BJTU\subject\sophomore\data
A B C D E F G H I J K
6
进程已结束，退出代码为 0

```

图 6.9

**【6.10 题解答】：**数据结构题集（C 语言版）》，第 6 章，第 44 页，第 6.66

题，难度系数为 4。

```

01:typedef struct PTNode
02:{
03:    char data;
04:    int parent;
05:}PTNode;
06:
07:typedef struct PTree
08:{
09:    PTNode nodes[20];
10:    int r, n;
11:};
12:
13:typedef struct CSNode
14:{
15:    char data;
16:    struct CSNode * first_child, *next_sibling;
17:};

```

---

```
18:///multiple pointer point the same guy, then be convenient to find parent
19:/// \param T
20:/// \return
21:CSNode * creat_cstree_by_ptree(PTree *T)
22:{
23:    CSNode *tem[15];
24:    CSNode *p, *q;
25:    int parent;
26:    if(T->n <= 0) return NULL;
27:
28:    for(int i=0; i<T->n; ++i)
29:    {
30:        p = (CSNode* ) malloc(sizeof (CSNode));
31:        p->data = T->nodes[i].data;
32:        p->first_child = p->next_sibling =NULL;
33:        parent = T->nodes[i].parent;
34:        if(parent != -1)
35:        {
36:            if(tem[parent]->first_child == NULL) tem[parent]->first_child =
p;
37:            else {
38:                q = tem[parent]->first_child;
39:                while(q->next_sibling != NULL)
40:                {
41:                    q = q->next_sibling;
42:                }
43:                q->next_sibling = p;
44:            }
45:        }
46:        tem[i] = p; // the tem group is to help find the parent
47:    }
48:    return tem[0];
49:}
50:void show_cstree(CSNode *T)
51://{//
52:
53://    if(T==NULL) return;
54:////// recursion is not perfect
55://    else printf("%c", T->data);
56://    show_cstree(T->next_sibling);
57://    show_cstree(T->first_child);
58://
59:    CSNode tem[30];
```

```
60:   int front, rear;
61:   front =rear =0;
62:   CSNode p;
63:   if(T == NULL) return;
64:   else
65:   {
66:       tem[rear] = *T;
67:       ++rear;
68:       do {
69:           p = tem[front];
70:           ++front;
71:           printf("%c ", p.data);
72:           if(p.first_child == NULL) continue;
73:           else
74:           {
75:               p = *p.first_child;
76:               tem[rear] = p;
77:               ++rear;
78:           }
79:           while (p.next_sibling != NULL) {
80:               tem[rear] = *p.next_sibling;
81:               ++rear;
82:               p = *p.next_sibling;
83:           }
84:       } while (front <rear);
85:   }
86:}

87:// sample one
88:   PTNode general_tree[10];
89:   PTree pTree;
90:   CSNode * CST;
91:   general_tree[0].data = 'R';
92:   general_tree[0].parent = -1;
93:   for(int i = 1; i<=9; ++i)
94:   {
95:       general_tree[i].data = 'A' +i -1;
96:   }
97:   general_tree[1].parent = general_tree[2].parent =
98:   general_tree[3].parent =0;
99:   general_tree[4].parent = general_tree[5].parent =1;
100:  general_tree[6].parent = 3;
101:  general_tree[7].parent = general_tree[8].parent =
102:  general_tree[9].parent = 6;
```

```
103:   for(int i = 0; i< 10; ++i)
104:       pTree.nodes[i] = general_tree[i];
105:   pTree.n=10;
106:   pTree.r = 0;
107:   CST = creat_cstree_by_ptree(&pTree);
108:   show_cstree(CST);
109:   return 0;
110:// sample two
111:
112:   PTNode general_tree[10];
113:   PTree pTree;
114:   CSNode * CST;
115:   general_tree[0].data = 'R';
116:   general_tree[0].parent = -1;
117:   for(int i = 1; i<=9; ++i)
118:   {
119:       general_tree[i].data = 'Z' -i+1 ;
120:   }
121:   general_tree[1].parent = general_tree[2].parent =
122:   general_tree[3].parent =0;
123:   general_tree[4].parent = general_tree[5].parent =1;
124:   general_tree[6].parent = 3;
125:   general_tree[7].parent = general_tree[8].parent =
126:   general_tree[9].parent = 6;
127:   for(int i = 0; i< 10; ++i)
128:       pTree.nodes[i] = general_tree[i];
129:   pTree.n=10;
130:   pTree.r = 0;
131:   CST = creat_cstree_by_ptree(&pTree);
132:   show_cstree(CST);
133:
134:
```

(1) 测试实例 1:



图 6.10.1

(2) 测试实例 2:

调试正常



```

D:\C++\Subject\Subject\...
R Z Y X W V U T S R
进程已结束, 退出代码为 0

```

图 6.10.2

**【6.11 题解答】:**《数据结构题集 (C 语言版)》, 第 6 章, 第 44 页, 第 6.63 题, 难度系数为 3。

求树的深度算法:

```

63: int depth_tree_cs(CSNode * T){
64:     if(T == NULL) return 0;
65:     else{
66:         int depth = depth_tree_cs(T->first_child);
67:         while(T->next_sibling)
68:         {
69:             int count = depth_tree_cs(T->next_sibling);
70:             if(count > depth) depth = count;
71:             return depth; // the same level return, need not to plus one
72:         }
73:         return depth+1; // finish one level then depth plus one
74:     }
75: }
76: //sample one
77: PTree pTree1;
78: pTree1.n = 11;
79: pTree1.r = 0;
80: pTree1.nodes[0].data = 'R';
81: pTree1.nodes[0].parent = -1;
82: for(int i=1; i<=11; ++i)
83: {
84:     pTree1.nodes[i].data = 'A'+i -1;
85: }
86:
87: pTree1.nodes[1].parent = 0;
88: pTree1.nodes[2].parent = 0;
89: pTree1.nodes[3].parent = 0;
90: pTree1.nodes[4].parent = 2;
91: pTree1.nodes[5].parent = 2;

```

```

92:  pTree1.nodes[6].parent = 2;
93:  pTree1.nodes[7].parent = 3;
94:  pTree1.nodes[8].parent = 6;
95:  pTree1.nodes[9].parent = 7;
96:  pTree1.nodes[10].parent = 9;
97:  CSNode* CST = creat_cstree_by_ptree(&pTree1);
98:  show_cstree(CST);
99:  printf("\n%d",depth_tree_cs(CST));
100:// sample two
101:  PTNode general_tree[10];
102:  PTree pTree;
103:  CSNode * CST;
104:  general_tree[0].data = 'R';
105:  general_tree[0].parent = -1;
106:  for(int i = 1; i<=9; ++i)
107:  {
108:      general_tree[i].data = 'A' +i-1 ;
109:  }
110:  general_tree[1].parent = general_tree[2].parent =
111:  general_tree[3].parent =0;
112:  general_tree[4].parent = general_tree[5].parent =1;
113:  general_tree[6].parent = 3;
114:  general_tree[7].parent = general_tree[8].parent =
115:  general_tree[9].parent = 6;
116:  for(int i = 0; i< 10; ++i)
117:      pTree.nodes[i] = general_tree[i];
118:  pTree.n=10;
119:  pTree.r = 0;
120:  CST = creat_cstree_by_ptree(&pTree);
121:  show_cstree(CST);

```

(1) 测试实例 1:



```

R A B C D E F G H I J
5
进程已结束，退出代码为 0

```

图 6.11.1



## (2) 测试实例 2:



```
R A B C D E F G H I
4
```


非递归实现 递归实现为 0

图 6.11.2

**【6.12 题解答】:**《数据结构题集 (C 语言版)》, 第 6 章, 第 44 页, 第 6.68 题, 难度系数为 3。

```
01://6.12
02:typedef struct DTree{
03:    char data;
04:    int degree;
05: };
06:CSNode * creat_by_tree_degree(DTree * T, int n)
07:{
08:    if(!T) return NULL;
09:    CSNode *tem = (CSNode*) malloc(sizeof(CSNode) * n);
10:    int index=1;
11:    for(int i =0; i<n; ++i)
12:    {
13:        tem[i].data = T[i].data;
14:        tem[i].next_sibling = tem[i].first_child =NULL;
15:    }
16:
17:    for(int i=0; i<n; ++i){
18:        int degree = T[i].degree;
19:        if(degree == 0) index++;
20:        else if(degree == 1)
21:        {
22:            tem[i].first_child = &tem[index++];
23:        }
24:        else if(degree >1)
25:        {
```

```
26:         tem[i].first_child = &tem[index++];
27:         for (int j=0; j<degree-1; ++j){
28:             tem[index-1].next_sibling = &tem[index++];
29:         }
30:     }
31: }
32: return &tem[0];}
33://sample
34://6.12
35:int n=8;
36:DTree dtree[8];
37:dtree[0].data = 'R';
38:dtree[0].degree = 3;
39:for(int i=1; i<=8; ++i)
40:{
41:    dtree[i].data = 'A' +i -1;
42:}
43:dtree[1].degree = dtree[3].degree = 1;
44:dtree[2].degree = 2;
45:dtree[4].degree = dtree[5].degree =dtree[6].degree =dtree[7].degree= 0;
46:
47: CSNode *T = creat_by_tree_degree(dtree, n);
48: show_cstree(T);
49:
```



```
R A B C D E F G
进程已结束，退出代码为 0
```

图 6.12

**【6.13 题解答】:**

若结点有左子树, 则其 `lchild` 域指示其左孩子, 否则令 `lchild` 域指示其前驱; 若结点有右子树, 则其 `rchild` 指示其右孩子, 否则令 `rchild` 域指示其后继。为避免混淆, 尚需改变结点结构, 增加两个标志域 `LTAG` 和 `RTAG`。`LTAG=0` 时, `lchild` 域指示结点的左孩子; `LTAG=1` 时, `lchild` 域指示结点的前驱。`RTAG=0` 时, `rchild` 域指示结点的右孩子; `RTAG=1` 时, `rchild` 域指示结点的后继。

**(1) 二叉树先序遍历方式的线索化算法;**

先序遍历即是根左右的顺序。

修改递归遍历算法即可实现。多加入一个 `pre` 指针指向前继, 当左子树 `NULL` 时, 修改 `ltag` 为 1, 并把 `lchild` 指向 `pre` (有可能为 `root`); 并把 `rtag` 修改为 1。

`Pre` 和 `p` 移动。如果右子树为空, 则把 `pre` 指向 `p`。

遍历右子树时如果 `p` 的左子树空, 改 `ltag`, `lchild` 指向 `pre`。改变 `pre` 和 `p`。如果 `pre` 右子树为空, 改 `rtag`, `rchild` 指向 `root`

**(2) 二叉树中序遍历方式的线索化算法;**

中序遍历为左根右。如果 `p` 非空, 左子树递归线索化。如果 `p` 的左孩子为空, 则给 `p` 加上左线索, 将其 `LTAG` 置为 1, 让 `p` 的左孩子指针指向 `pre` 前驱); 否则将 `p` 的 `LTAG` 置为 0。如果 `pre` 的右孩子为空, 则给 `pre` 加上右线索, 将其 `RTAG` 置为 1, 让 `pre` 的右孩子指针指向 `p` (后继); 否则将 `pre` 的 `RTAG` 置为 0。将 `pre` 指向刚访问过的结点 `p`, 即 `pre=p`  
右子树递归线索化

**(3) 二叉树后序遍历方式的线索化算法;**

以递归形式线索化。

如果是空树, 结点的左右指针指向自身; 否则, 界定啊的左后指针指向根结点。把 `pre` 指向结点。开始调用自身函数。如果最后一个结点没有右孩子, `pre` 的右孩子指向头结点。

(4) 对于先序线索化二叉树，如何求任意一个结点的前驱结点；  
没什么好方法  
首先找到 **parent**（使用三链表没意思欸），**p** 为左孩子，则 **pre** 为 **root**；  
如果 **p** 为右孩子，要是 **root** 有左孩子，就是 **root** 左孩子的先序结果，否则为 **root**。

(5) 对于先序线索化二叉树，如何求任意一个结点的后继结点；  
如果 **rtag=1**，则后继为 **rchild** 指向的；如果左孩子存在，则后继为左孩子；其他情况就是右孩子。

(6) 对于中序线索化二叉树，如何求任意一个结点的前驱结点；  
如果 **ltag=1**，则前驱为 **lchild** 指向的；此外就是正常遍历顺序，即中序遍历左孩子的结果，即 **lchild the farthest rchild**

(7) 对于中序线索化二叉树，如何求任意一个结点的后继结点；  
如果 **rtag=1**，后继就是 **rchild** 指向的；此外就是正常遍历顺序，即中序遍历右孩子的结果，**rchild the farthest lchild**

(8) 对于后序线索化二叉树，如何求任意一个结点的前驱结点；  
如果 **ltag=1**，就是 **lchild** 指向的；如果 **rchild** 不 **NULL**，就是右孩子；其他情况即使左孩子。

(9) 对于后序线索化二叉树，如何求任意一个结点的后继结点。  
若结点是二叉树的根，则后继为空；若结点是其双亲的右孩子，或是其双亲的左孩子且其双亲没有右子树，则其后继就是其双亲结点；若结点是其双亲的左孩子，且其双亲有右子树，则其后继为双亲的右子树上按后序遍历列出的第一个结点

**Ps** 这种题还真是不知道怎么答 还是直接写算法更轻松。

【6.14 题解答】:《数据结构题集 (C 语言版)》, 第 6 章, 第 41 页, 第 6.26 题, 难度系数为 3。假设用于通信的电文仅由 8 个字母组成, 字母在电文中出现的频率分别为 0.07, 0.19, 0.02, 0.06, 0.32, 0.03, 0.21, 0.10。试为这 8 个字母设计哈夫曼编码。使用 0~7 的二进制表示形式是另一种编码方案, 对于上述实例, 比较两种方案的优缺点。

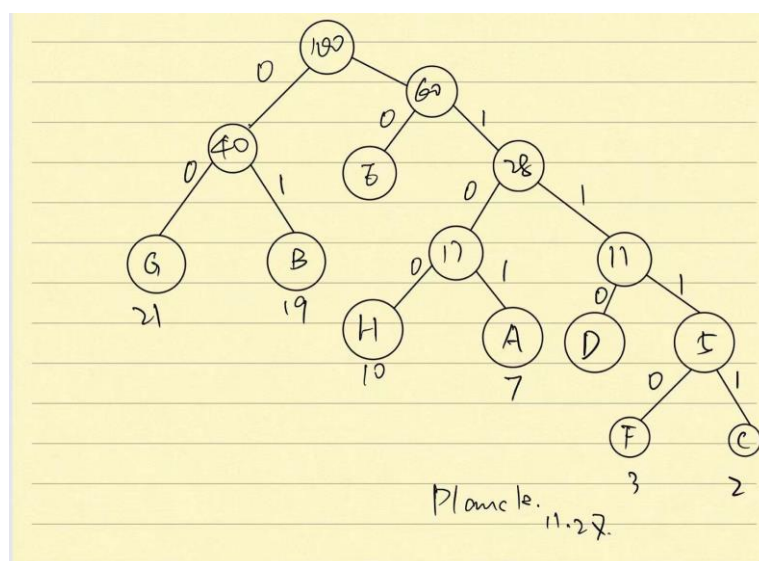


图 6.14

建立哈夫曼编码, 设权重为频率的 100 倍, 分别对应 A 到 H

A: 1101 B:01 C:11111 D:1110 E:10 F:11110 G:00 H:1100

第二种 即

000 001 010 011 100 101 110 111

第二种数学期望 为  $3 \times 100\% = 3$

结论:

哈夫曼树最短