

第二章

第二章

2.2 设有数据结构 (D, R) ，其中

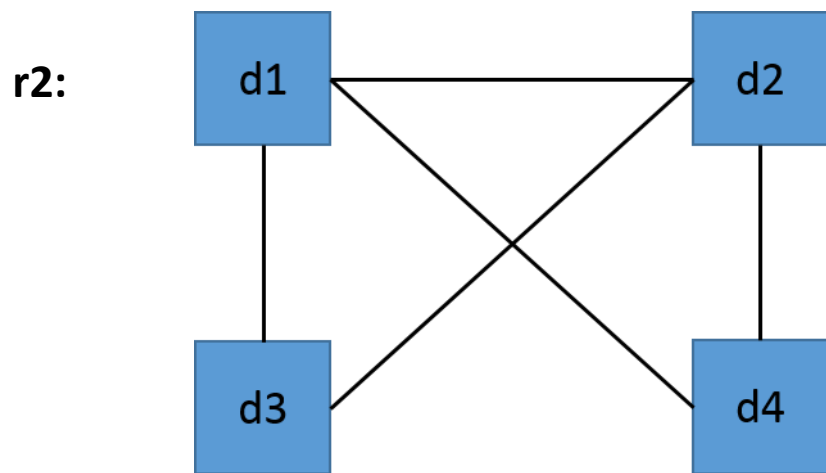
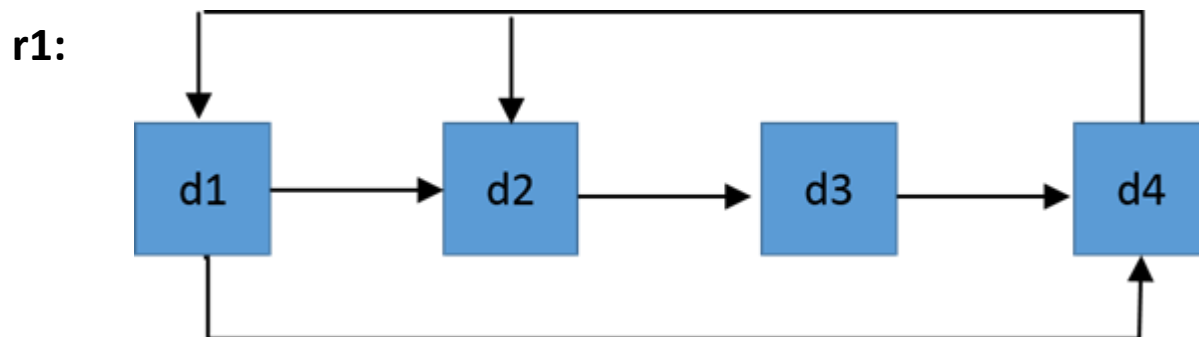
$D = \{d1, d2, d3, d4\}$

$R = \{r1, r2\}$

$r1 = \{ \langle d1, d2 \rangle, \langle d2, d3 \rangle, \langle d3, d4 \rangle, \langle d1, d4 \rangle, \langle d4, d2 \rangle, \langle d4, d1 \rangle \}$

$r2 = \{ (d1, d2), (d1, d3), (d1, d4), (d2, d4), (d2, d3) \}$

试绘出其逻辑结构示意图。



2.3 设 n 是正整数。试写出下列程序段中用记号 “ \triangle ” 标注的语句的频率：

```
(1)      i=1; k=0;
          while(i<=n-1) {
             $\triangle$       k+=10*i;
                        i++;
          }
```

频度： $n-1(n \geq 1)$

```
(2)      i=1; k=0;
          do {
             $\triangle$       k+=10*i;
                        i++;
          }while(i<=n-1)
```

频度： $n \geq 2$ 时 $n-1$ ； $n=1$ 时 1 // do 语句先执行

```
(3)      i=1; k=0;
          do {
             $\triangle$       k+ = 10*i; i++;
          }while(i==n);
```

频度： $n=2$ 时候，频度为 2 ，否则频度为 1 //分类讨论

(4) `i=1; j=0;`
 `while(i+j≤n) {`
 \triangle `if(i<j) i++; else j++;`//此处是执行if else语句的频
度
 `}`

频度: **n**

(5) `x=n; y=0; //n是不小于1的常数`
 `while(x≥(y+1)*(y+1)){`
 \triangle `y++;`
 `}`

频度: **round(sqrt(n))** //round代表向下取整函数

(6) `x=91; y=100;`
 `while (y>0)`
 \triangle `if(x>100) { x-=10; y--; }`
 `else x++ ;`
 `}`

频度: **1100次** //x每次增加到 **101**, y--, 注意计算if else 语句 频
度

(7) for(i=0; i<n; i++)
 for(j=i; j<n; j++)
 for(k=j; k<n; k++)
 \triangle x+=2;

频度: $\sum_{i=1}^n \frac{i^2}{2} + \frac{1}{4}n(n+1) = \frac{n(n+1)(n+2)}{6}$

$$\begin{aligned} \sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j 1 &= \sum_{i=1}^n \sum_{j=1}^i j = \sum_{i=1}^n \frac{i \times (i+1)}{2} = \frac{1}{2} \sum_{i=1}^n (i^2 + i) \\ &= \frac{1}{2} \left(\frac{n \times (n+1) \times (2n+1)}{6} + \frac{n \times (n+1)}{2} \right) \\ &= \frac{n \times (n+1) \times (2n+1)}{12} + \frac{n \times (n+1)}{4} \\ &= \frac{n(n+1)(n+2)}{6} \end{aligned}$$

第三章 栈和队列

3.7 假设一个算术表达式中可以包含三种符号：圆括号“(”和“)”、方括号“[”和“]”、花括号“{”和“}”，且这三种括号可按任意次序嵌套使用。编写判别给定表达式中所含的括号是否正确配对的算法(已知表达式已存入数据元素为字符的顺序表中)。

```

Status Compare(){
    char c,e;
    SqStack S;
    InitStack(S);
    printf("请输入算术表达式:\n");
    while((c = getchar()) != '\n'){
        switch(c){
            case '(':
                //左括号，直接入栈
                Push(S,c);
                break;
            case '[':
            case '{':
                if(Pop(S,e) == ERROR || e != '(')
                    return FALSE;
                break;
            case ')':
                //右括号，与栈顶元素做比对
                if(Pop(S,e) == ERROR || e != '[')
                    return FALSE;
                break;
            case ']':
                if(Pop(S,e) == ERROR || e != '{')
                    return FALSE;
                break;
            case '}':
                if(Pop(S,e) == ERROR || e != '(')
                    return FALSE;
                break;
            default :
                break;
        }
    }
    if(isEmpty(S))
        return TURE;
    else
        return FALSE;
}

```


3.8 设表达式由单字母变量、双目运算符和圆括号组成（如：“ $(a*(b+c)-d)/e$ ”）。试写一个算法，将一个书写正确的表达式转换为逆波兰式。

(1) 首先，需要分配2个栈，栈s1用于临时存储运算符（含一个结束符号），此运算符在栈内遵循越往栈顶优先级越高的原则；栈s2用于输入逆波兰式，为方便起见，栈s1需先放入一个优先级最低的运算符，在这里假定为‘#’；

(2) 从中缀式的左端开始逐个读取字符x，逐序进行如下步骤：

1. 若x是操作数，则分析出完整的运算数（在这里为方便，用字母代替数字），将x直接压入栈s2；

2. 若x是运算符，则分情况讨论：

若x是‘(’，则直接压入栈s1；

若x是‘)’，则将距离栈s1栈顶的最近的‘(’之间的运算符，逐个出栈，依次压入栈s2，此时抛弃‘)’；

若x是除‘(’和‘)’外的运算符，则再分如下情况讨论：

若当前栈s1的栈顶元素为‘(’，则将x直接压入栈s1；

若当前栈s1的栈顶元素不为‘(’，则将x与栈s1的栈顶元素比较，若x的优先级大于栈s1栈顶运算符优先级，则将x直接压入栈s1。否则，将栈s1的栈顶运算符弹出，压入栈s2中，直到栈s1的栈顶运算符优先级别低于（不包括等于）x的优先级，或栈s2的栈顶运算符为‘(’，此时再则将x压入栈s1；

(3) 在进行完(2)后，检查栈s1是否为空，若不为空，则将栈中元素依次弹出并压入栈s2中（不包括‘#’）；

(4) 完成上述步骤后，栈s2便为逆波兰式输出结果。但是栈s2应做一下逆序处理，因为此时表达式的首字符位于栈底；

```

int compare_dominant(char a, char b)
{//比较运算符的优先级

    if(a == '+' || a == '-' && b == '/' || b == '*')
        return 0;
    if(a == '/' || a == '*' && b == '+' || b == '-')
        return 1;
    if(a == '+' || a == '-' && b == '+' || b == '-')
        return 0;
    if(a == '/' || a == '*' && b == '/' || b == '*')
        return 0;
}

```

```

void trans_post_exp(char exp[], char newexp[])
{//将中缀表达式（已存储在数组中）转换成后缀表达式，将后缀表达式存储在另一个数组中；
    int i=0;
    char stack[maxsize]; int top = -1;
    for(int k=0; exp[k] != '#'; k++)
    {
        //扫描到左括号，则直接入栈
        if(exp[k] == '(')
            stack[++top] = exp[k];
        //若扫描到右括号，则令栈顶元素出栈，并存入新数组中
        else if(exp[k] == ')' && top != -1)
        {
            newexp[i++] = stack[top--];
            if(stack[top] == '(')
                top--;
        }
        //若扫描到字母，则直接存入新数组中；
        else if(exp[k] != '+' && exp[k] != '-' && exp[k] != '/' && exp[k] != '*')
            newexp[i++] = exp[k];

        //若扫描到运算符，则将其与栈顶运算符比较优先级，如果新运算符优先级较高，将其入栈，
        //否则栈顶元素出栈并入新数组，再将新运算符入栈
        else
        {
            if(top == -1 || stack[top] == '(') //栈空或者栈顶元素为左括号时直接入栈
                stack[++top] = exp[k];

            else if(compare_dominant(exp[k], stack[top]) == 1)
                stack[++top] = exp[k];
            else //新运算符优先级不高于栈顶元素时一直出栈，直到新运算符优先级高于栈顶元素
            {
                while(compare_dominant(exp[k], stack[top]) == 0 && top != -1)
                    newexp[i++] = stack[top--];
                stack[++top] = exp[k];
            }
        }
    }
    while(top != -1 )
    {
        if(stack[top] != '(')
            newexp[i++] = stack[top];
        top--;
    }
}

```

```

void print(char exp[])
{
    for(int i=0; exp[i] != '#'; i++)
        printf("%c", exp[i]);
}

int main()
{
    char exp[maxsize];
    char x; int i=0;
    printf("请输入正确的表达式: \n");

    while(x != '#')
    {
        x = getchar();
        exp[i++] = x;
    }
    getchar(); //用于接收回车键

    char newexp[maxsize];
    for(int i=0; i<maxsize; i++)
        newexp[i] = '#';
    trans_post_exp(exp, newexp);

    print(newexp);

    return 0;
}

```

3.9 试用类 C 写一个算法，对逆波兰式求值。

假设逆波兰式用 s 表示。

```
InitStack(S);
```

```
i = 0;
```

```
while(s[i] != '\0'){
```

```
    if(!In(s[i],OP))    //操作数，直接入栈
```

```
        Push(S,s[i]);
```

```
    else{
```

```
        Pop(S,b);
```

```
        Pop(S,a);
```

```
        Push(S,Operate(a,s[i],b));
```

```
    }
```

```
    i++;
```

```
}
```

```
.
```

3.10 假设以带头结点的单循环链表表示队列，只设一个尾指针指向队尾元素，不设头指针。试编写相应的队列初始化、入队和出队的算法。

初始化:

```
Status InitQueue(LinkQueue &Q){  
    //带头结点，先分配头结点  
    Q.rear = (QueuePtr) malloc(sizeof(QNode));  
    if(!Q.rear) exit(OVERFLOW);  
    Q.rear->next = Q.rear;  
    return OK;  
}
```

入队:

```
Status EnQueue(LinkQueue &Q, QElemType e){  
    p = (QueuePtr) malloc(sizeof(QNode));  
    if(!p) exit(OVERFLOW);  
    p->data = e; p->next = Q.rear->next;    //插入队尾  
    Q.rear->next = p;  
    Q.rear = p;  
    return OK;  
}
```

出队:

```
Status DeQueue(LinkQueue &Q, QElemType &e){  
    if(Q.rear->next == Q.rear) return ERROR;  
    p = Q.rear->next->next;    //队首结点  
    e = p->data;  
    Q.rear->next->next = p->next;    //删除队首结点  
    free(p);  
    return OK;  
}
```

3.11 假设将循环队列定义为：以 rear 和 length 分别指示队尾元素和队列长度。试给出此循环队列的队满条件，并写出相应的入队和出队算法（在出队算法中要传递回队头元素的值）。

队满条件：

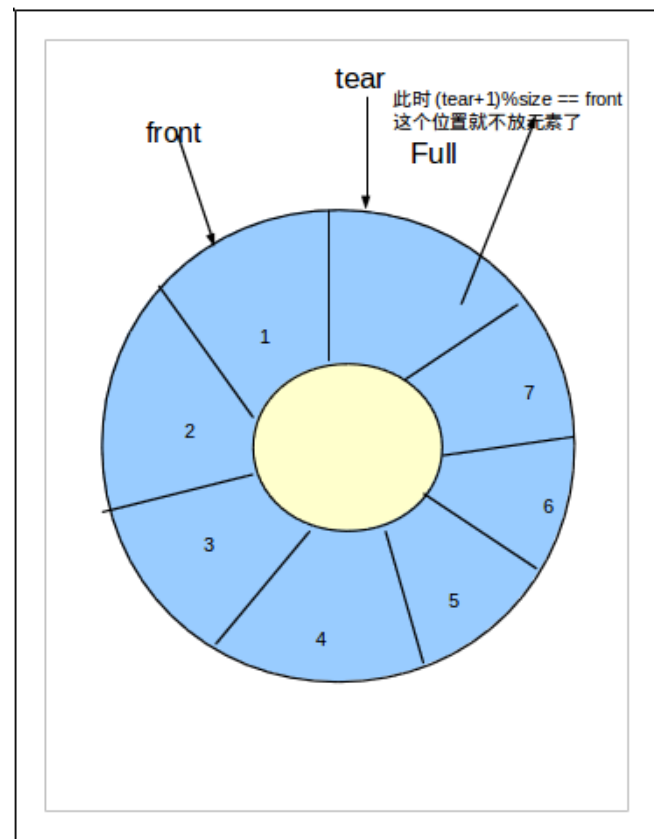
$(Q.rear+1)\%MAXQSIZE == (Q.rear-Q.length+MAXQSIZE)\%MAXQSIZE$; 或者 $Q.length=MAXQSIZE-1$

入队：

```
Status EnQueue(SqQueue &Q, QElemType e){
    if((Q.rear+1)%MAXQSIZE == (Q.rear-Q.length+MAXQSIZE)%MAXQSIZE) //队满
        return ERROR;
    Q.rear = (Q.rear+1)%MAXQSIZE;
    Q.length++;
    Q.base[Q.rear] = e;
    return OK;
}
```

出队：

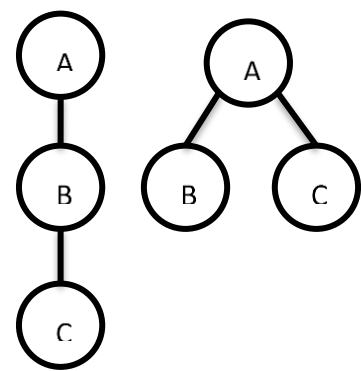
```
Status DeQueue(SqQueue &Q, QElemType &e){
    if(Q.length == 0) return ERROR;
    e = Q.base[(Q.rear-Q.length+MAXQSIZE)%MAXQSIZE];
    Q.length--;
    return OK;
}
```



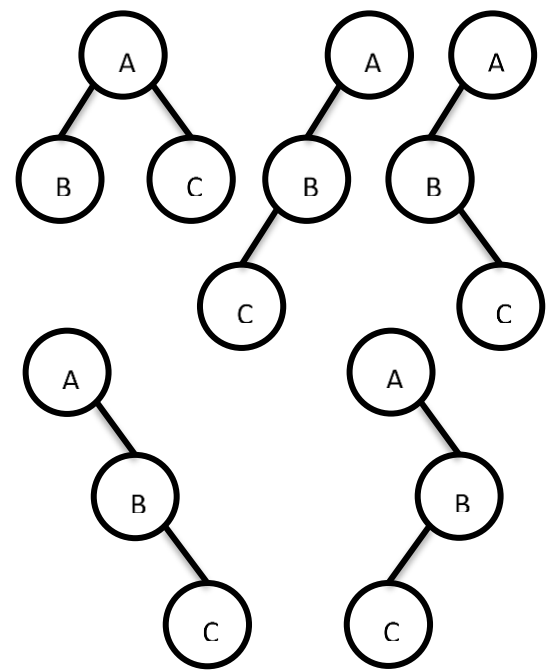
$$(rear+1)\%maxsize=front$$

第六章

6.1试分别绘出具有 3 个结点的树和 3 个结点的二叉树的所有不同形态。



具有 3 结点的树



具有 3 结点的二叉树

6.2 设结点 x 是二叉树上一个度为 1 的结点， x 有几个子树？

2个子树（空树、本身）

6.3描述满足下列条件的二叉树形态：

(1) 先序遍历序列与中序遍历序列相同； 不含左子树

(2) 后序遍历序列与中序遍历序列相同； 不含右子树

(3) 先序遍历序列与后序遍历序列相同； 既不含左子树，也不含右子树

6.4 一个深度为 H 的满 k 叉树有如下性质：第 H 层上所有结点都是叶子结点，其余各层上每个结点都有 k 棵非空子树。如果从 1 开始按自上而下、自左向右的次序对全部结点编号，问：

- (1) 各层的结点数目是多少？
- (2) 编号为 i 的结点的父结点(若存在)的编号是多少？
- (3) 编号为 i 的结点的第 j 个孩子(若存在)的编号是多少？
- (4) 编号为 i 的结点有右兄弟的条件是什么？其右兄弟的编号是多少？

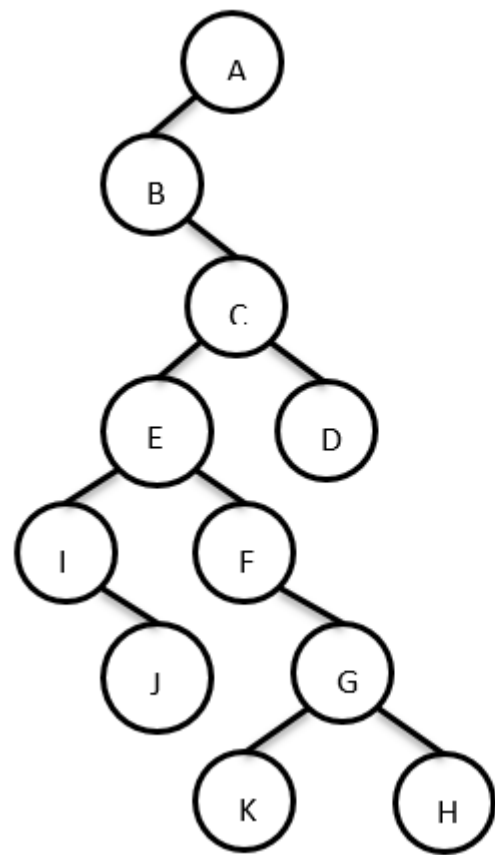
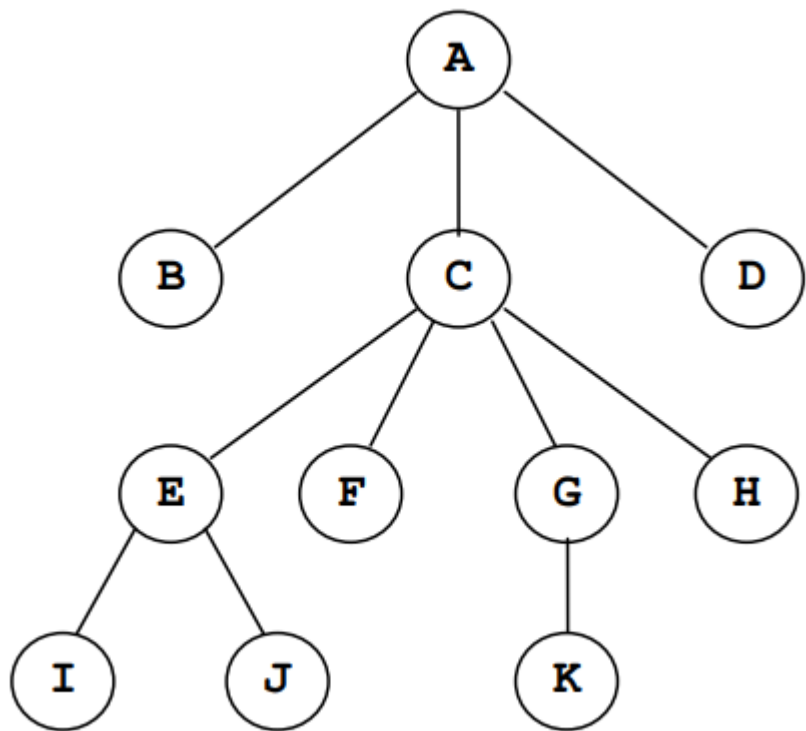
(1) k^{i-1}

(2) $j = \left\lfloor \frac{i+k-2}{k} \right\rfloor$ 如果 i 是其双亲的最小的孩子,则 i 减去根结点的一个结点, 应是 k 的整数倍, 该整数即为所在的组数, 每一组为一棵满 k 叉树, 正好应为双亲结点的编号。如果 i 是其双亲的最大的孩子, 则 $i+k-1$ 为其最小的弟弟, 再减去一个根结点, 除以 k , 即为其双亲结点的编号

(3) 结点 i 的右孩子的编号为 $ki+1$, 左孩子的编号为 $ki+1-k+1=k(i-1)+2$, 第 j 个孩子的编号为 $k(i-1)+2+j-1=ki-k+j+1$

(4) 当 $(i-1)\%k \neq 0$ 时, 结点 i 有右兄弟, 其右兄弟的编号为 $i+1$ 。

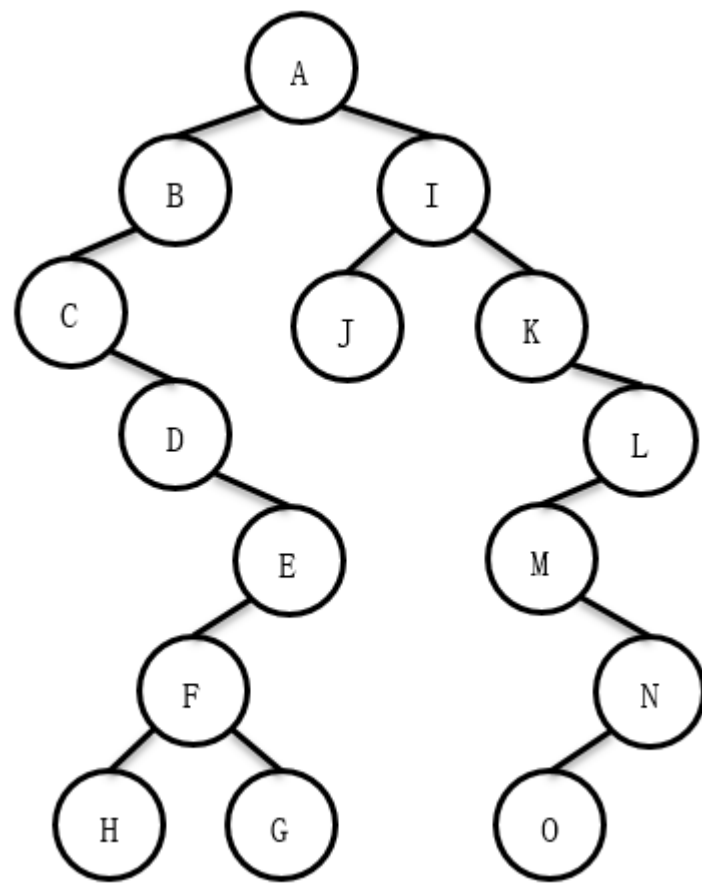
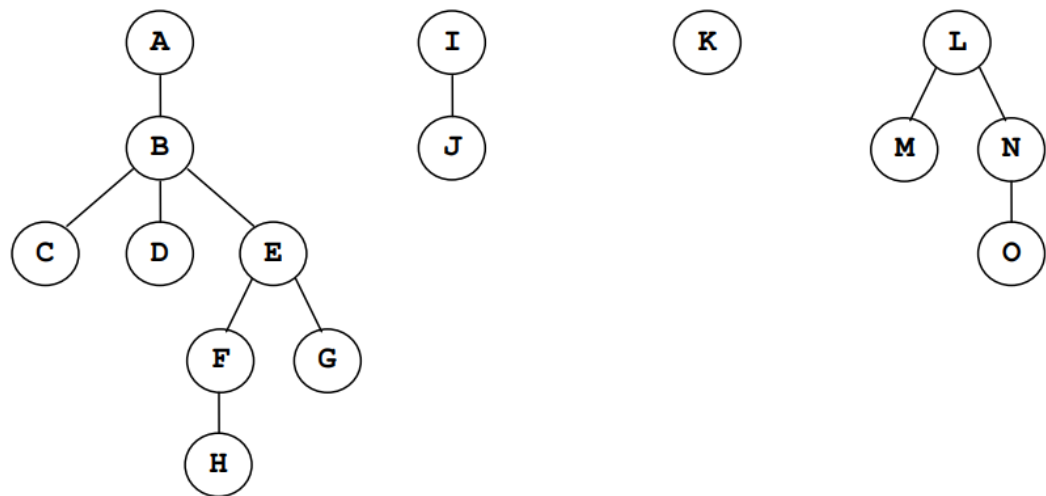
6.8 已知一棵树如图所示，画出与该树对应的二叉树，并写出该树的先根遍历序列和后根遍历序列。



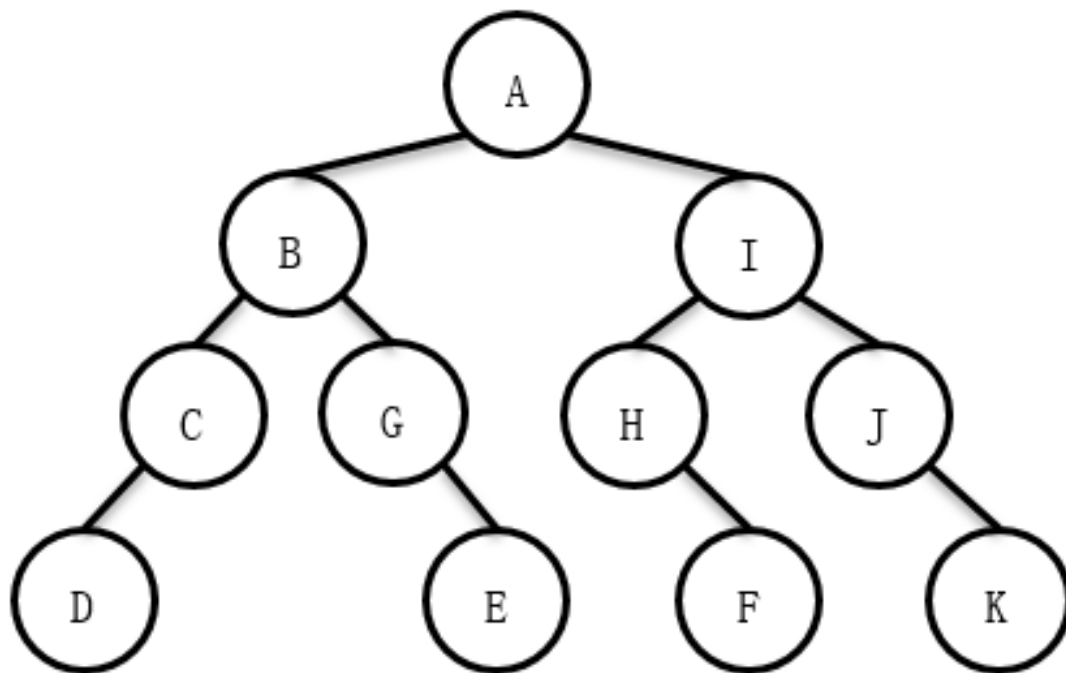
先根: ABCEIJFGKHD

后根: BIJEFKGHCDA

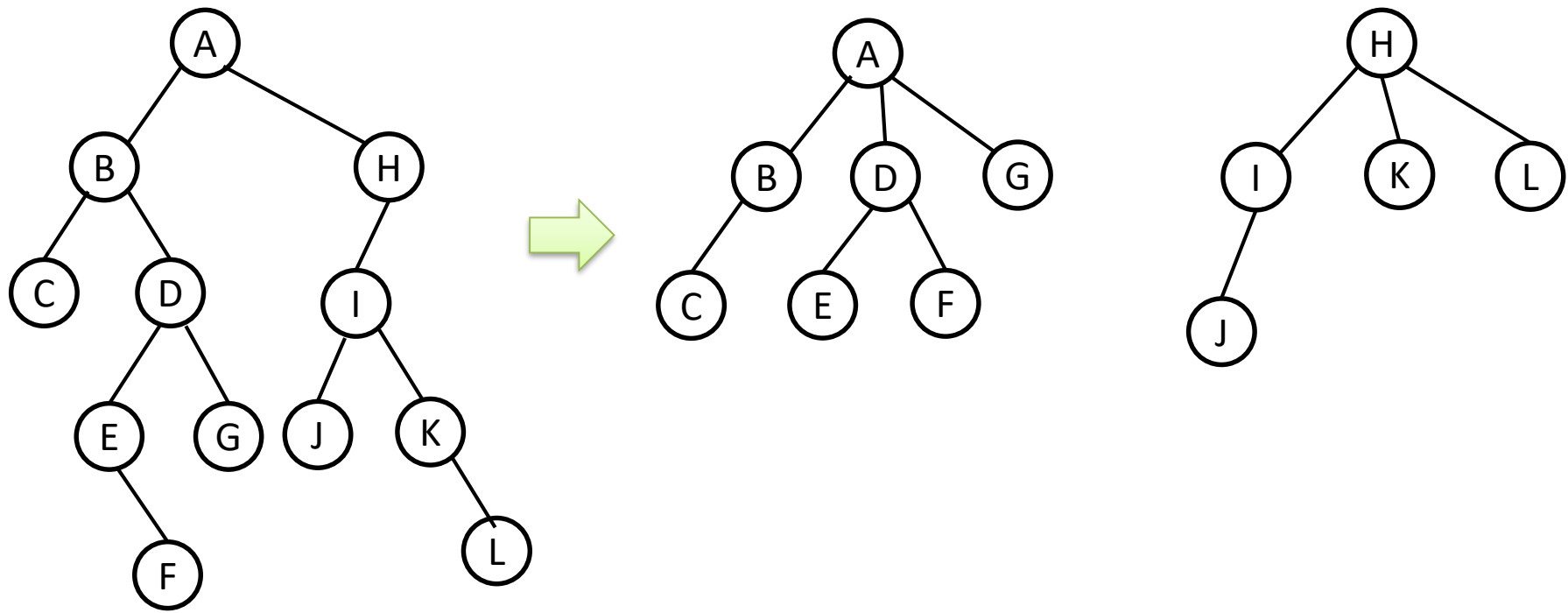
6.9 将如图 6-2 所示的森林转化为对应的二叉树。



6.11 已知某二叉树的中序序列为 DCBGEAHFIJK，后序序列为 DCEGBFHKJIA。请画出该二叉树。

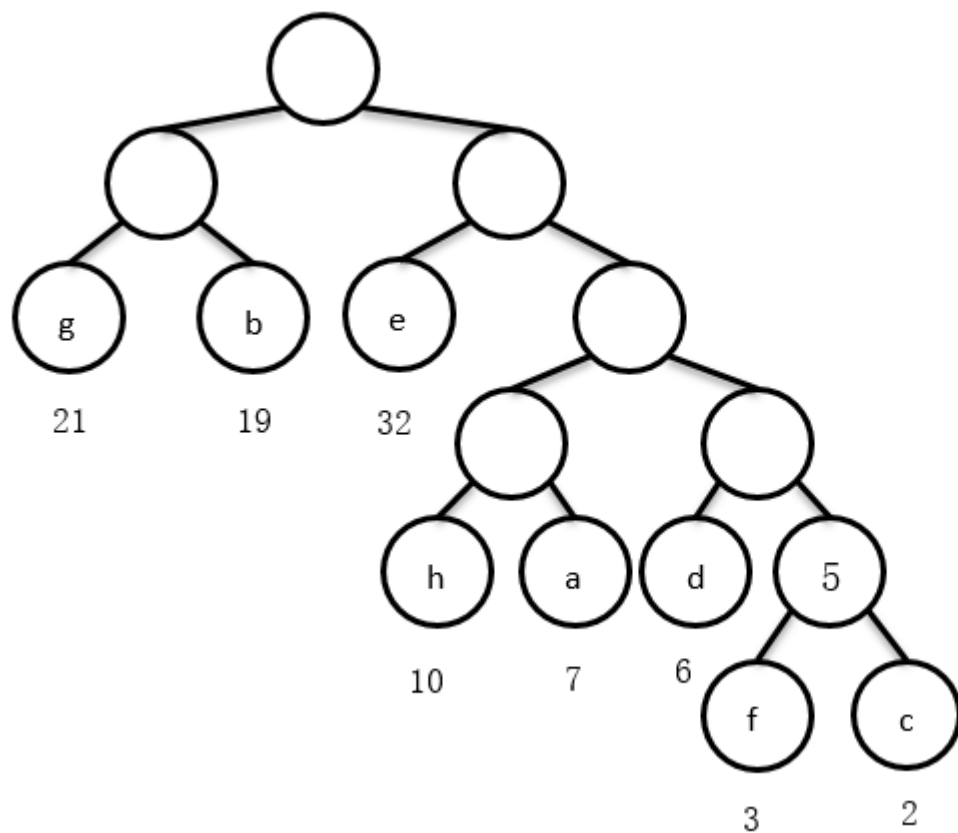


6.13 已知森林F的先根遍历访问序列为ABCDEFGH IJ KL，中根遍历访问序列为CBEFDGAJIKLH。请画出这个森林F。



6.14 假设某个电文由(a,b,c,d,e,f,g,h) 8 个字母组成，每个字母在电文中出现的次数分别为(7,19,2,6,32,3,21,10)，试解答下列问题：

- (1) 画出 Huffman 树；
- (2) 写出每个字母的 Huffman 编码；
- (3) 在对该电文进行最优二进制编码处理后，电文的二进制位数。



a:1101
b:01
c:11111
d:1110
e:10
f:11110
g:00
h:1100

电文二进制位数：
 $261 = 4 \times 7 + 2 \times 19 + 5 \times 2 + 4 \times 6 + 2 \times 32 + 5 \times 3 + 2 \times 21 + 4 \times 10$

6.16 试编写算法，实现将二叉树所有结点的左右子树互换。

```
Status ExchangeBiTree(BiTree &T)
{
    BiTree p;
    if(T){
        p=T->lchild;
        T->lchild=T->rchild;
        T->rchild=p;
        ExchangeBiTree(T->lchild);
        ExchangeBiTree(T->rchild);
    }
    return OK;
}
```

6.12 写出计算树的深度的算法，树用孩子兄弟链表表示。

```
int GetDepth_CSTree(CSTree T)//求孩子兄弟链表表示的树 T 的深度
{
    if(!T) return 0; //空树
    else
    {
        for(maxd=0,p=T->firstchild;p;p=p->nextsib)
            if((d=GetDepth_CSTree(p))>maxd) maxd=d; //子树的最大深度
        return maxd+1;
    }
}
//GetDepth_CSTree
```


6.12 写出计算二叉树第 K 层结点数的算法。

```
int GetNodeNum(Bitree T, int k)
{
    if (!T || k<1)
        return 0;
    if (k=1)
        return 1;
    return GetNodeNum(T->lchild, k-1)+ GetNodeNum(T->rchild, k-1);
}
```

第九章 查找

8.4

设有一组关键字{9,01,23,14,55,20,84,27},采用哈希函数: $H(\text{key}) = \text{key} \bmod 6$,表长为10,用开放地址法的二次探测再散列方法 $H_i = (H(\text{key}) + d_i) \bmod 10 (d_i = 1^2, 2^2, 3^2, \dots)$ 解决冲突。

要求:对该关键字序列构造哈希表,并计算查找成功的平均查找长度。

散列地址	0	1	2	3	4	5	6	7	8	9
关键字	55	1	14	9	84	23	20	27		
比较次数	4	1	1	1	3	1	3	3		

$$\text{平均查找长度: } ASL_{\text{suss}} = \frac{4+1+1+1+3+1+3+3}{8} = \frac{17}{8} = 2.125$$

以关键字 55 为例:

$$H(55) = 55 \% 6 = 1 \text{ (冲突)}$$

$$H_1(1+1^2) \% 10 = 2 \text{ (冲突)}$$

$$H_2(1+2^2) \% 10 = 5 \text{ (冲突)}$$

$$H_3(1+3^2) \% 10 = 0$$

所以比较了 4 次

8.5

对关键字集{30,15,21,40,25,26,36,37},若查找表的装填因子为 0.8,采用线性探测再散列方法解决冲突:

- (1) 设计哈希函数;
- (2) 画出哈希表;
- (3) 计算查找成功和查找失败的平均查找长度;
- (4) 写出将哈希表中某个数据元素删除的算法。

(1) 由于装填因子为 0.8,关键字有 8 个,所以表长为 $8/0.8=10$ 。实践证明,用除留余数法,当 P 取小于哈希表长的最大质数时,产生的哈希函数较好,所以哈希函数为: $H(key)=key \% 7$ 。

(2)

散列地址	0	1	2	3	4	5	6	7	8	9
关键字	21	15	30	36	25	40	26	37		
比较次数	1	1	1	3	1	1	2	6		

$$(3) \text{ 平均查找长度: } ASL_{\text{succ}} = \frac{1+1+1+3+1+1+2+6}{8} = \frac{16}{8} = 2$$

$$\text{平均查找长度: } ASL_{\text{unsucc}} = \frac{9+8+7+6+5+4+3+2+1+1}{10} = \frac{46}{10} = 4.6$$

(4)

```
// 从哈希表h[n]中删除元素k, 若删除成功返回1, 否则返回0
int Delete(int h[n], int k){
    i = k%7; // 哈希函数用上面
    if (h[i]== maxint) {
        printf("无关键字%d\n", k); // maxint解释成空地址
        return (0);
    }
    if (h[i]==k) {
        h[i]=-maxint; // 被删元素换成最大机器数的负数
        return (1);
    }
    else{
        int j=i;
        for (d=1; d≤ n-1; d++) {
            i= (j+d) %n; // n为表长, 此处为10
            if (h[i]== maxint) return (0); // maxint解释成空地址
            if (h[i]==k) {h[i]=-maxint; return (1);
        }
    }
    printf ("无关键字%d\n", k);
    return (0);
}
```

8.7 已知关键字序列 {10, 25, 33, 19, 06, 49, 37, 76, 60}，哈希地址空间为 0~10，哈希函数为 $H(\text{Key}) = \text{Key} \% 11$ ，求：

- (1) 用开放定址线性探测法处理冲突，构造哈希表 HT1，分别计算在等概率情况下 HT1 查找成功和查找失败的 ASL；
- (2) 用开放定址二次探测法处理冲突，构造哈希表 HT2，计算在等概率情况下 HT2 查找成功的 ASL；
- (3) 用拉链法解决冲突，构造哈希表 HT3，计算 HT3 在等概率情况查找成功的 ASL。

(1) HT1

0	1	2	3	4	5	6	7	8	9	10
33	76		25	37	49	6	60	19		10
76					60	60				76

查找成功： $ASL = (7*1 + 2*3) / 9 = 1.44$

查找失败： $ASL = (3 + 2 + 1 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 4) / 11 = 3.45$

8.7 已知关键字序列 {10, 25, 33, 19, 06, 49, 37, 76, 60}，哈希地址空间为 0~10，哈希函数为 $H(\text{Key}) = \text{Key} \% 11$ ，求：

- (1) 用开放定址线性探测法处理冲突，构造哈希表 HT1，分别计算在等概率情况下 HT1 查找成功和查找失败的 ASL；
- (2) 用开放定址二次探测法处理冲突，构造哈希表 HT2，计算在等概率情况下 HT2 查找成功的 ASL；
- (3) 用拉链法解决冲突，构造哈希表 HT3，计算 HT3 在等概率情况查找成功的 ASL。

(2) HT2

0	1	2	3	4	5	6	7	8	9	10
33	60		25	37	49	6		19	76	10
76				60	60	60			60	76

查找成功： $ASL = (7*1 + 3 + 5) / 9 = 1.67$

(3) HT3

0	1	2	3	4	5	6	7	8	9	10
33			25	37	49	6		19		10
					60					76

查找成功： $ASL = (7*1 + 2*2) / 9 = 1.22$

3. 散列表中地址链接法的 ASL

ASL_成功 = 每个元素被访问（查找）的次数

ASL_失败 = 每个地址被访问（查找）的次数

8.12 写出判别一棵二叉树是否为二叉排序树的算法, 设二叉排序树中不存在关键字值相同的结点。

```
bool IsSearchTree(Bitree *T, Elemtype &e;)
{
    //递归遍历二叉树是否为二叉排序树, e 初始值为最小值
    if(!T) return TRUE;
    retl = IsSearchTree(T->lchild, e);
    if(T->data < e) return FALSE;           //当前元素小于中序序列的前一个元素
    e = T->data;
    retr = IsSearchTree(T->rchild, e);
    return retl && retr;
}
```




Thank you!