1.树

1.1树的概念

树（Tree）是n（n≥0）个节点的有限集合T，它满足两个条件：

1. 有且仅有一个特定的称为根（Root）的节点；
2. 其余的节点可以分为m（m≥0）个互不相交的有
   限集合T1、T2、……、Tm，其中每一个集合又是一棵树，
   并称为其根的子树（Subtree）。

表示方法： **树形表示法、目录表示法。**

1.2树的度

一个节点的子树的个数称为该节点的**度数**，一棵树的度数是指该树中节点的最大度数。

度数为零的节点称为**树叶**或**终端节点**，度数不为零的节点称为分支节点，除根节点外的

分支节点称为内部节点。一个节点的子树之根节点称为该节点的**子节点**，该节点称为它

们的**父节点**，同一节点的各个子节点之间称为兄弟节点。一棵树的根节点没有父节点，

叶节点没有子节点。

1.3树的边数和高度

一个节点系列k1,k2, ……,ki,ki+1, ……,kj,并满足ki是ki+1的父节点，就称为一条从k1到kj的**路径**，

路径的长度为j-1,即路径中的**边数**。路径中前面的节点是后面节点的祖先，后面节点是前面

节点的子孙。节点的层数等于父节点的层数加一，根节点的层数定义为一。树中节点层数

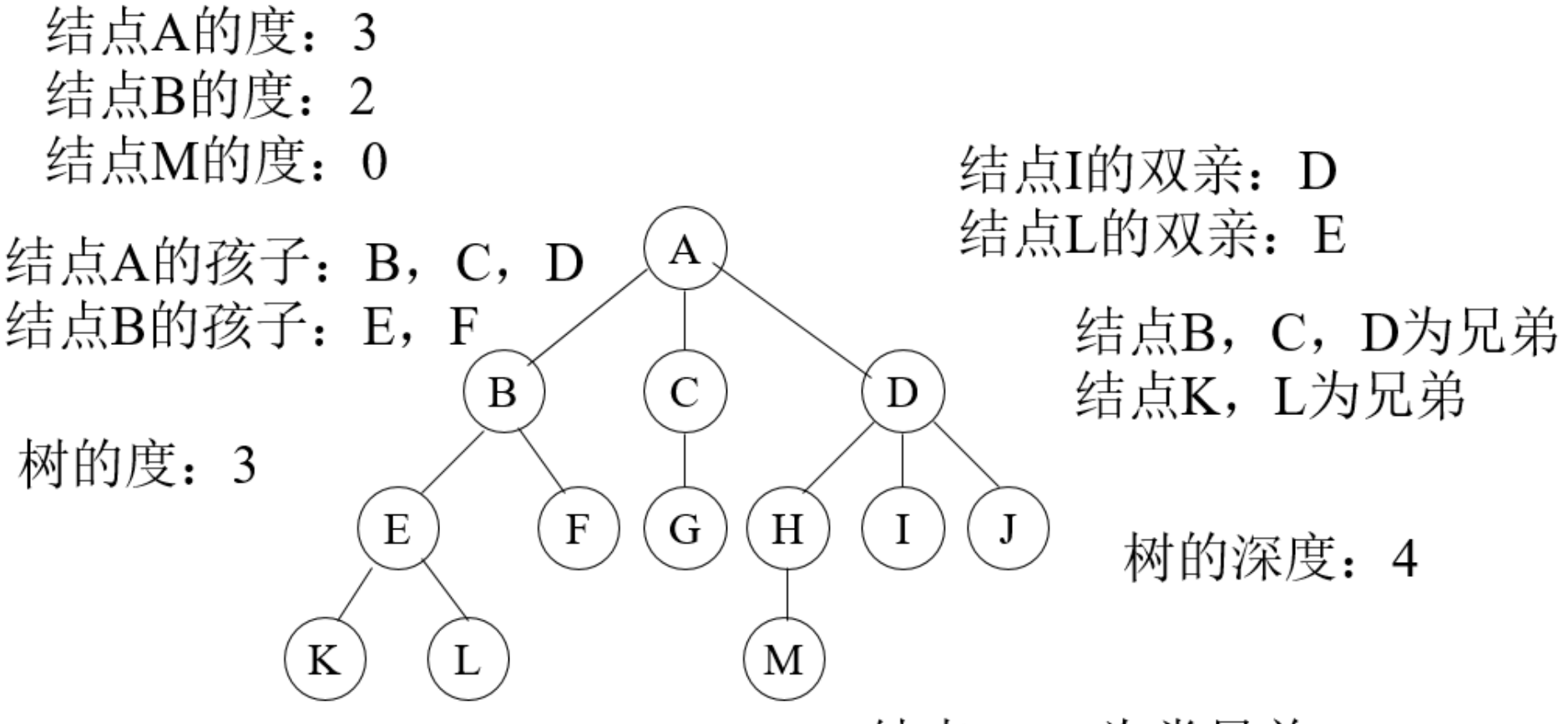
的最大值称为该树的**高度**或**深度**。若树中每个节点的各个子树的排列为从左到右，不能交

换，即兄弟之间是有序的，则该树称为**有序树**。一般的树是有序树。

m（m≥0）棵互不相交的树的集合称为**森林**。树去掉根节点就成为森林，森林加上一个新

的根节点就成为树。

1.4树概念相关的图



1.5树的逻辑结构

树的逻辑结构： 树中任何节点都可以有零个或多个直接后继节点（子节点），

但至多只有一个直接前趋节点（父节点），根节点没有前趋节点，叶节点没有

后继节点。

2.二叉树

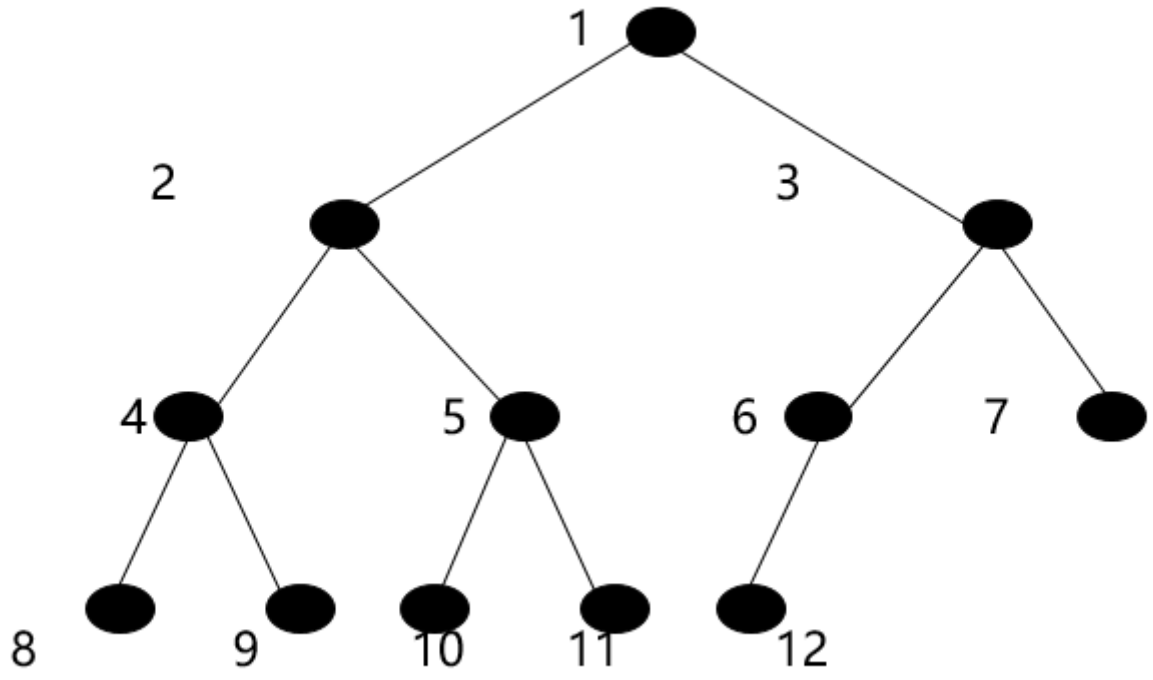
2.1二叉树的概念

二叉树的定义： 二叉树（BinaryTree）是n（n≥0）个节点的有限集合

，它或者是空集（n = 0），或者是由一个根节点以及两棵互不相交的、

分别称为左子树和右子树的二叉树组成。二叉树与普通有序树不同，

二叉树严格区分左孩子和右孩子，即使只有一个子节点也要区分左右。



2.2二叉树的性质

1. 二叉树第i（i≥1）层上的节点最多为2^{i-1}个。
2. 深度为k（k≥1）的二叉树最多有2^k-1个节点。
3. 在任意一棵二叉树中，树叶的数目比度数为2的节点的数目多一。

   总节点数为各类节点之和：n = n0 + n1 + n2

   总节点数为所有子节点数加一：n = n1 + 2*n2 + 1

   故得：n0 = n2 + 1 ;
4. **满二叉树**： 深度为k（k≥1）时有2^k - 1个节点的二叉树。
5. **完全二叉树**： 只有最下面两层有度数小于2的节点，且最下面一层的叶节点集中在最左边的若干位置上。具有n个节点的完全二叉树的深度为（log2n） + 1或log2(n+1)。

2.3完全二叉树特性

完全二叉树节点的编号方法是从上到下，从左到右，根节点为1号节点。

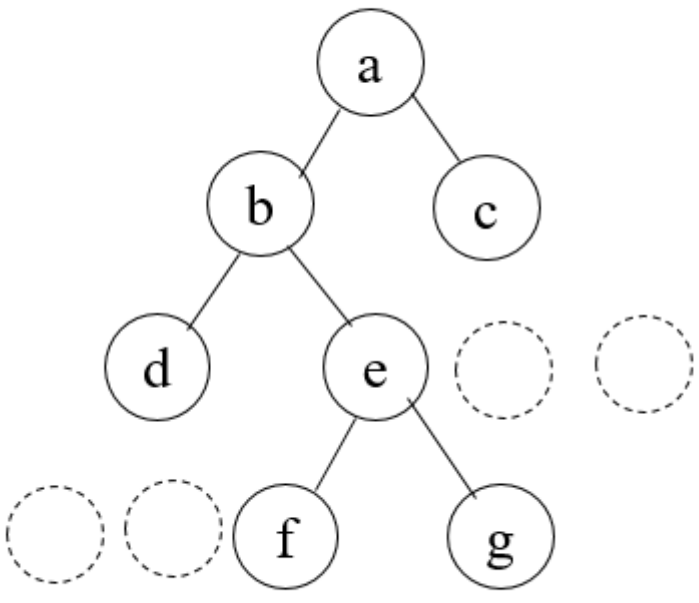
设完全二叉树的节点数为n，某节点编号为i

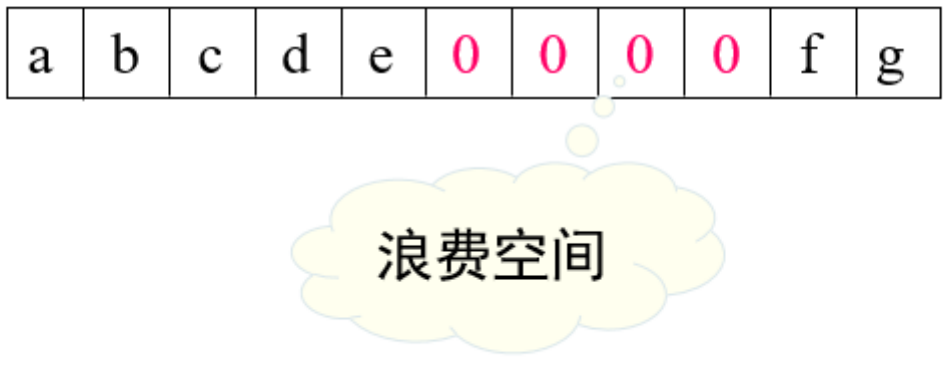
1. 当i > 1（不是根节点）时，有父节点，父节点编号为i/2;
2. 当2*i≤n时，有左孩子，其编号为2*i ,否则没有左孩子，本身是叶节点;
3. 当2*i + 1≤n时，有右孩子，其编号为2*i+1 ,否则没有右孩子;
4. 当i为奇数且不为1时，有左兄弟，其编号为i-1,否则没有左兄弟;
5. 当i为偶数且小于n时，有右兄弟，其编号为i + 1,否则没有右兄弟;

2.4二叉树的顺序存储

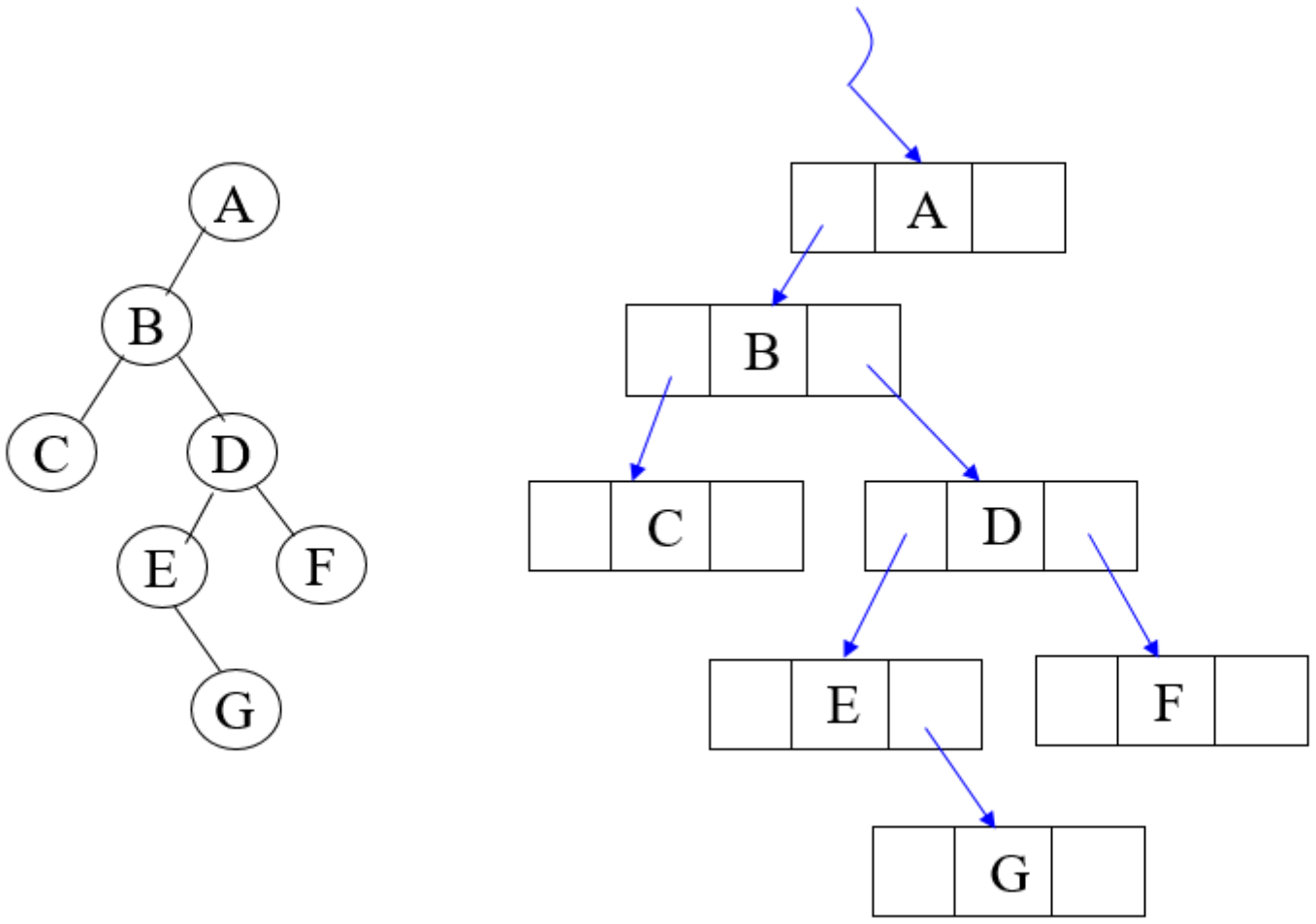
有n个节点的完全二叉树可以用有n+1个元素的数组进行顺序存储，节点号和数组下标一一对应，下标为零的元素不用。

利用以上特性，可以从下标获得节点的逻辑关系。不完全二叉树通过添加虚节点构成完全二叉树，然后用数组存储，这要浪费一些存储空间。





2.5二叉树的链式存储



2.6完全二叉树的创建

bitree.h

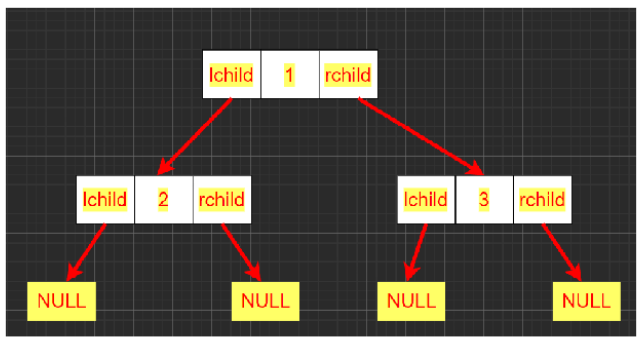
```h
#ifndef __BITREE_H__
#define __BITREE_H__

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

#define datatype int

typedef struct bitree{
    datatype data;
    struct bitree *lchild,*rchild;
}bitree_t;

// 创建一颗完全二叉树
// n是完全二叉树的节点个数
// i是节点的编号，i>=1
bitree_t * BiTreeCompleteCreate(int n,int i);
#endif
```

bitree.c

```
bitree_t* BiTreeCompleteCreate(int n, int i)
{
    bitree_t* root;                          3     1
    root = (bitree_t*)malloc(sizeof(*root));
    root->data = i;
         2*1<=3
    if ((2 * i) <= n)
        root->lchild = BiTreeCompleteCreate(n, 2 * i);
    else
        root->lchild = NULL;
         2*1+1<=3
    if ((2 * i + 1) <= n)
        root->rchild = BiTreeCompleteCreate(n, 2 * i+1);
     else
        root->rchild = NULL;
    return root;
}
```

```
bitree_t* BiTreeCompleteCreate(int n, int i)
{                                            3     2
    bitree_t* root;
    root = (bitree_t*)malloc(sizeof(*root));
    root->data = i;
         2*2<=3
    if ((2 * i) <= n)
        root->lchild = BiTreeCompleteCreate(n, 2 * i);
    else
        root->lchild = NULL;
         2*2+1<=3
    if ((2 * i + 1) <= n)
        root->rchild = BiTreeCompleteCreate(n, 2 * i+1);
     else
        root->rchild = NULL;
    return root;
}
```

```
bitree_t* BiTreeCompleteCreate(int n, int i)
{                                            3     3
    bitree_t* root;
    root = (bitree_t*)malloc(sizeof(*root));
    root->data = i;
         2*3<=3
    if ((2 * i) <= n)
        root->lchild = BiTreeCompleteCreate(n, 2 * i);
    else
        root->lchild = NULL;
         2*3+1<=3
    if ((2 * i + 1) <= n)
        root->rchild = BiTreeCompleteCreate(n, 2 * i+1);
     else
        root->rchild = NULL;
    return root;
}
```

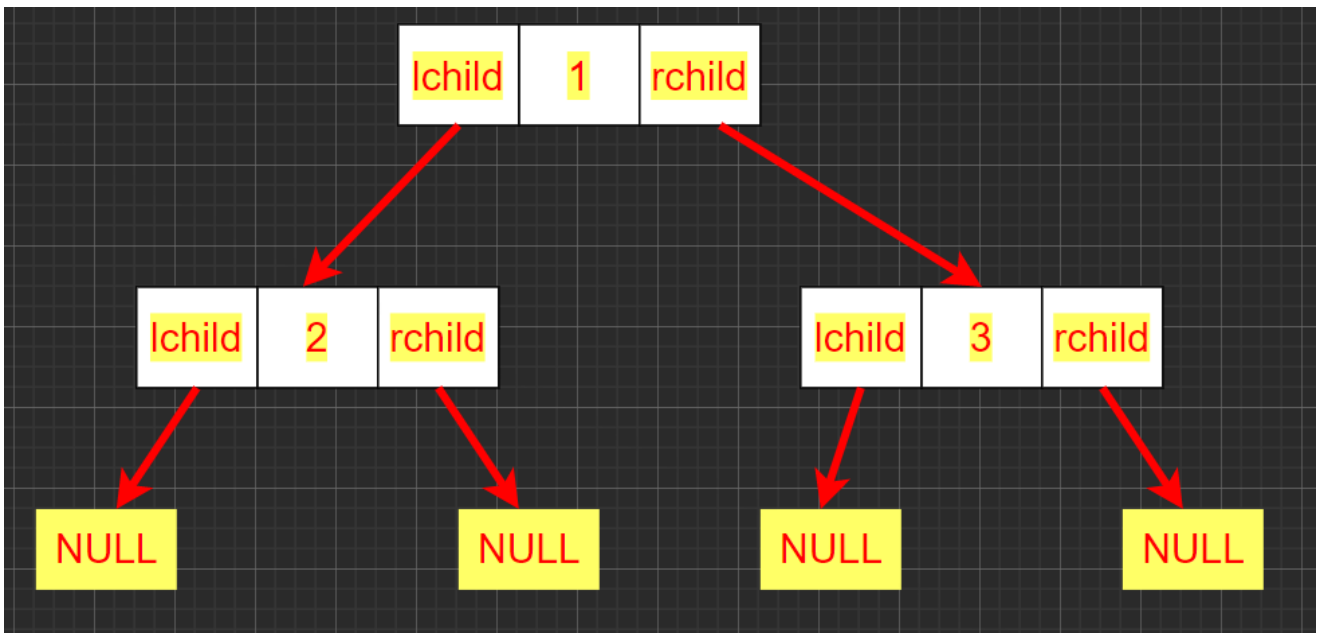



```c
#include "bitree.h"

bitree_t* BiTreeCompleteCreate(int n, int i)
{
    bitree_t* root;

    root = (bitree_t*)malloc(sizeof(*root));
    if (root == NULL) {
        printf("%s malloc memory error\n", __func__);
        return NULL;
    }
    root->data = i; // 将根节点的编号存入
    // 当2*i≤n时，有左孩子，其编号为2*i ,否则没有左孩子，本身是叶节点;
    if ((2 * i) <= n) {
        // 有左孩子
        root->lchild = BiTreeCompleteCreate(n, 2 * i);
    } else {
        // 没有左孩子
        root->lchild = NULL;
    }
    // 当2*i＋1≤n时，有右孩子，其编号为2*i+1 ,否则没有右孩子;
    if ((2 * i + 1) <= n) {
        // 有右孩子
        root->rchild = BiTreeCompleteCreate(n, 2 * i+1);
    } else {
        // 没有右孩子
        root->rchild = NULL;
    }

    return root;
}
```

main.c

```c
#include "bitree.h"

int main(int argc, const char* argv[])
{
    bitree_t* root;

    root = BiTreeCompleteCreate(7, 1);
    if (root == NULL)
        return -1;

    return 0;
}
```

2.7借助队列完成二叉树的遍历(BFS)



遍历的思想：先让根节点入队，让根节点出队，打印根节点的值，

判断根节点是否有左右孩子，如果有左右孩子让他们入队。先让根

节点的左孩子出队，打印左孩子的值，在判断它是否有左右孩子，

如果没有就不入队了。先让根节点的右孩子出队，打印右孩子的值，

判断它是否有左右孩子，如果没有就不入队了。此时队列为空，循环

退出即可。

bitree.h

```c
#ifndef __BITREE_H__
#define __BITREE_H__

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

#define datatype int

typedef struct bitree{
    datatype data;
    struct bitree *lchild,*rchild;
}bitree_t;

// 创建一颗完全二叉树
// n是完全二叉树的节点个数
// i是节点的编号，i>=1
bitree_t * BiTreeCompleteCreate(int n,int i);
#endif
```

bitree.c

```c
#include "bitree.h"
//  BiTreeCompleteCreate(3, 1)
//     1
//    /  \
//   2    3
bitree_t* BiTreeCompleteCreate(int n, int i)
{
    bitree_t* root;

    root = (bitree_t*)malloc(sizeof(*root));
    if (root == NULL) {
        printf("%s malloc memory error\n", __func__);
        return NULL;
    }
```

```
    root->data = i; // 将根节点的编号存入
    // 当2*i≤n时，有左孩子，其编号为2*i ,否则没有左孩子，本身是叶节点;
    if ((2 * i) <= n) {
        // 有左孩子
        root->lchild = BiTreeCompleteCreate(n, 2 * i);
    } else {
        // 没有左孩子
        root->lchild = NULL;
    }
    // 当2*i＋1≤n时，有右孩子，其编号为2*i+1 ,否则没有右孩子;
    if ((2 * i + 1) <= n) {
        // 有右孩子
        root->rchild = BiTreeCompleteCreate(n, 2 * i+1);
    } else {
        // 没有右孩子
        root->rchild = NULL;
    }

    return root;
}

```

linkqueue.h

```
#ifndef __LINKQUEUE_H__
#define __LINKQUEUE_H__

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

#define datatype int
typedef struct node {
    datatype data;
    struct node* next;
} node_t;

typedef struct {
    node_t *front, *rear;
} linkqueue_t;

linkqueue_t* LinkQueueCreate(void);
int LinkQueueEnQueue(linkqueue_t* q, datatype data);
int LinkQueueIsEmpty(linkqueue_t* q);
datatype LinkQueueDeQueue(linkqueue_t* q);
void LinkQueueShow(linkqueue_t* q);
#endif
```

linkqueue.c

```
#include "linkqueue.h"

linkqueue_t* LinkQueueCreate(void)
{
    node_t* h;
    linkqueue_t* q;
    h = (node_t*)malloc(sizeof(*h));
    if (h == NULL) {
        printf("%s malloc node_t error\n", __func__);
        return NULL;
    }
    h->data = (datatype)0;
    h->next = NULL;

    q = (linkqueue_t*)malloc(sizeof(*q));
    if (q == NULL) {
        printf("%s malloc linkqueue_t error\n", __func__);
        return NULL;
    }
    q->front = q->rear = h;

    return q;
}

int LinkQueueEnQueue(linkqueue_t* q, datatype data)
{
    node_t* tmp;
    // 1.分配tmp(node_t*)
    tmp = (node_t*)malloc(sizeof(*tmp));
    if (tmp == NULL) {
        printf("%s malloc node_t memory error\n", __func__);
        return -1;
    }
    tmp->data = data;
```

```
    // 2.将tmp入队
    tmp->next = q->rear->next;
    q->rear->next = tmp;
    // 3.让rear记录当前队列队尾
    q->rear = tmp;

    return 0;
}
int LinkQueueIsEmpty(linkqueue_t* q)
{
    return q->front == q->rear ? 1 : 0;
}

datatype LinkQueueDeQueue(linkqueue_t *q)
{
    node_t *tmp;
    datatype data;
    if(LinkQueueIsEmpty(q)){
        printf("%s linkqueue empty\n",__func__);
        return (datatype)-1;
    }

    tmp = q->front->next;
    q->front->next = tmp->next;
    if(tmp->next == NULL){
        q->rear = q->front;
    }

    data = tmp->data;
    if(tmp!=NULL){
        free(tmp);
        tmp = NULL;
    }

    return data;
}

void LinkQueueShow(linkqueue_t* q)
{
    node_t *h = q->front;

    while(h->next){
        printf("-%d",h->next->data);
        h = h->next;
    }
    printf("-\n");
}
```

main.c

```
#include "bitree.h"
#include "linkqueue.h"

int main(int argc, const char* argv[])
{
    bitree_t *root, *tmp;
    linkqueue_t* q;
    // 1.创建完全二叉树
    root = BiTreeCompleteCreate(9, 1);
    if (root == NULL)
        return -1;
    // 2.创建队列
    q = LinkQueueCreate();
    if (q == NULL)
        return -1;
    // 3.将根节点入队
    LinkQueueEnQueue(q, (datatype)root);

    // 4.循环遍历,队列为空退出，不为空循环
    while (!LinkQueueIsEmpty(q)) {
        tmp = (bitree_t*)LinkQueueDeQueue(q); // 出队
        printf(" %d", tmp->data); // 访问数据

        if (tmp->lchild != NULL) {//如果有左孩子让左孩子入队
            LinkQueueEnQueue(q, (datatype)tmp->lchild);
        }
        if (tmp->rchild != NULL) {//如果有右孩子让右孩子入队
            LinkQueueEnQueue(q, (datatype)tmp->rchild);
        }
    }
    printf("\n");
    return 0;
}
```

2.8二叉树的前中后序遍历（DFS）

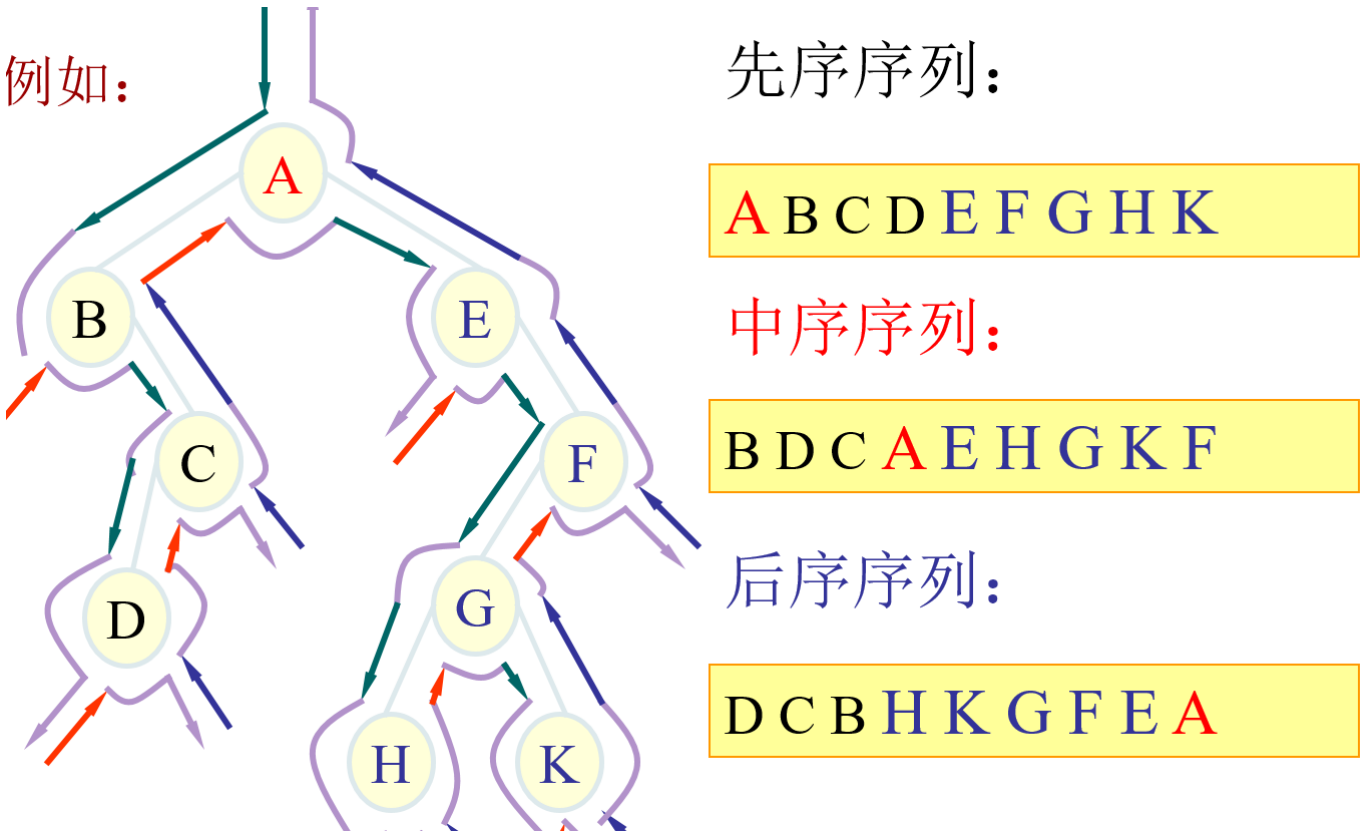
2.8.1先中后序遍历的思想

先序遍历（根左右）：先访问树根，再访问左子树，最后访问右子树；

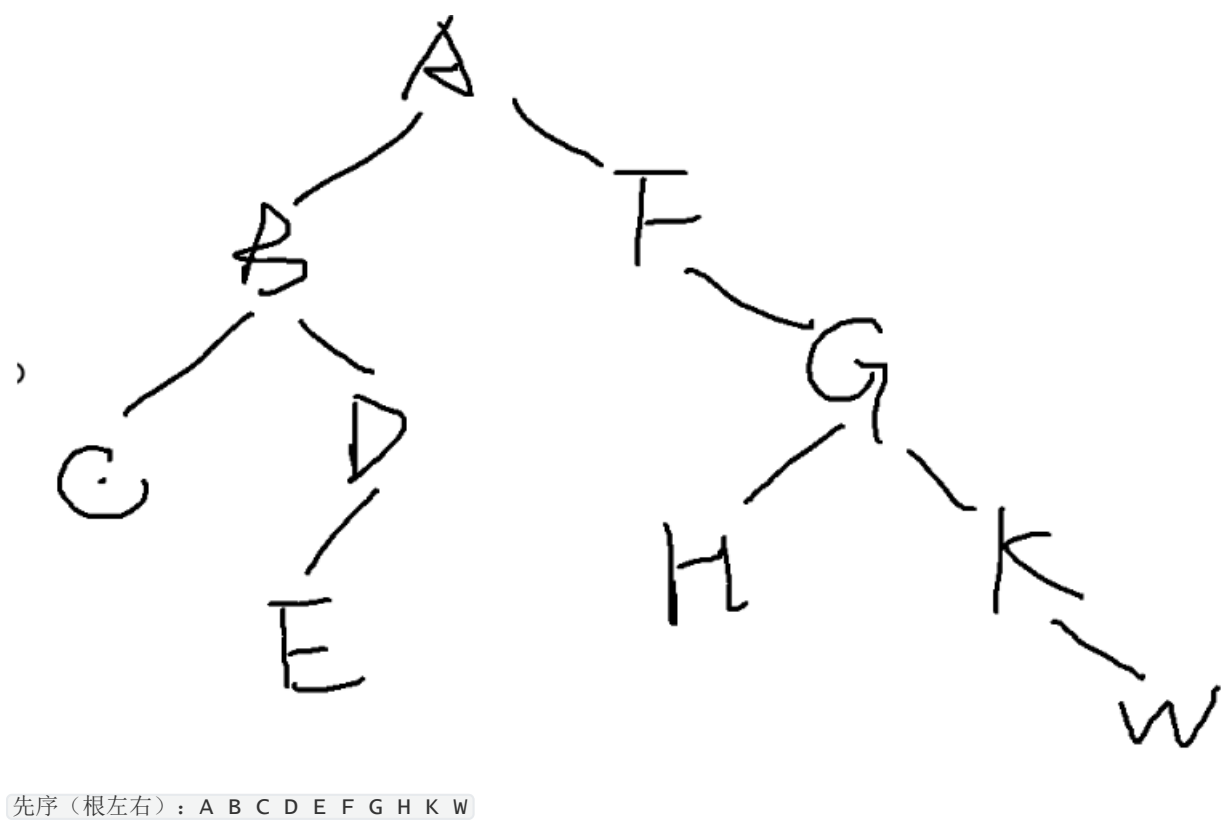
中序遍历（左根右）：先访问左子树，再访问树根，最后访问右子树；

后序遍历（左右根）：先访问左子树，再访问右子树，最后访问树根；

2.8.2先中后序遍历的实例1



2.8.3先中后序遍历实例2

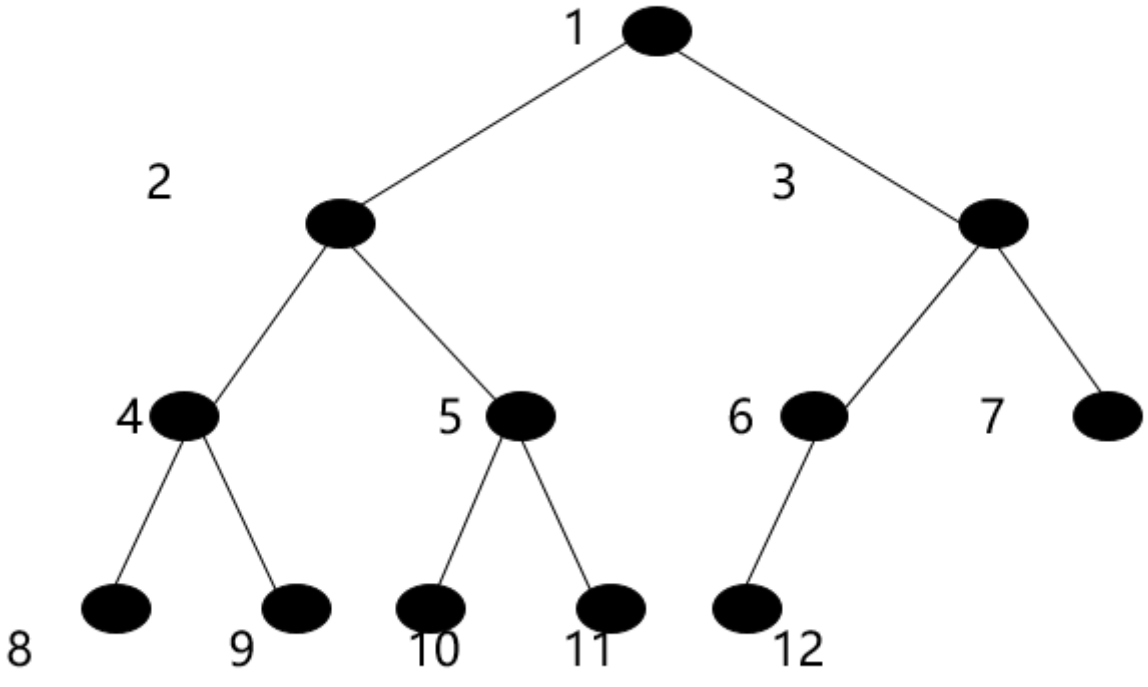


先序（根左右）：A B C D E F G H K W

中序（左根右）：C B E D A F H G K W

后序（左右根）：C E D B H W K G F A

2.8.4先中后序遍历的实例3



先：1 2 4 8 9 5 10 11 3 6 12 7

中：8 4 9 2 10 5 11 1 12 6 3 7

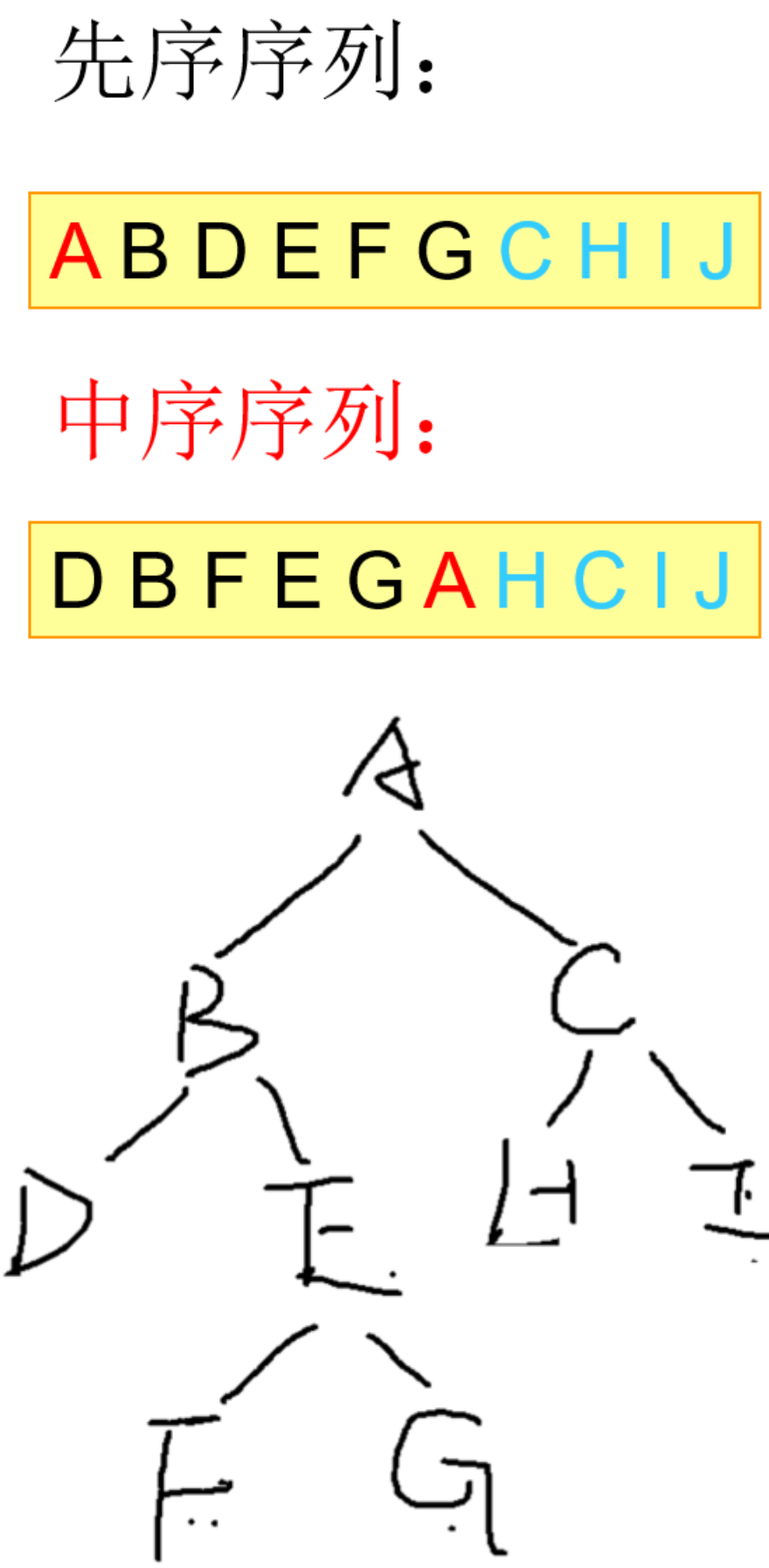
后：8 9 4 10 11 5 2 12 6 7 3 1

2.8.5先中后序遍历的实例4

给出先和中序遍历，推导后序遍历

给出中和后序遍历，推导先序遍历

> 注：如果让你推导二叉树的遍历方式，一定会给出中序遍历。



后序遍历的结果：D F G E B H J I C A

2.8.6先中后序遍历代码实现

bitree.h

```h
#ifndef __BITREE_H__
#define __BITREE_H__

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

#define datatype int

typedef struct bitree{
    datatype data;
    struct bitree *lchild,*rchild;
}bitree_t;

// 创建一颗完全二叉树
// n是完全二叉树的节点个数
// i是节点的编号，i>=1
bitree_t * BiTreeCompleteCreate(int n,int i);
void BiTreeFrontOrder(bitree_t *root);
void BiTreeMiddleOrder(bitree_t *root);
void BiTreeRearOrder(bitree_t *root);
#endif
```

bitree.c

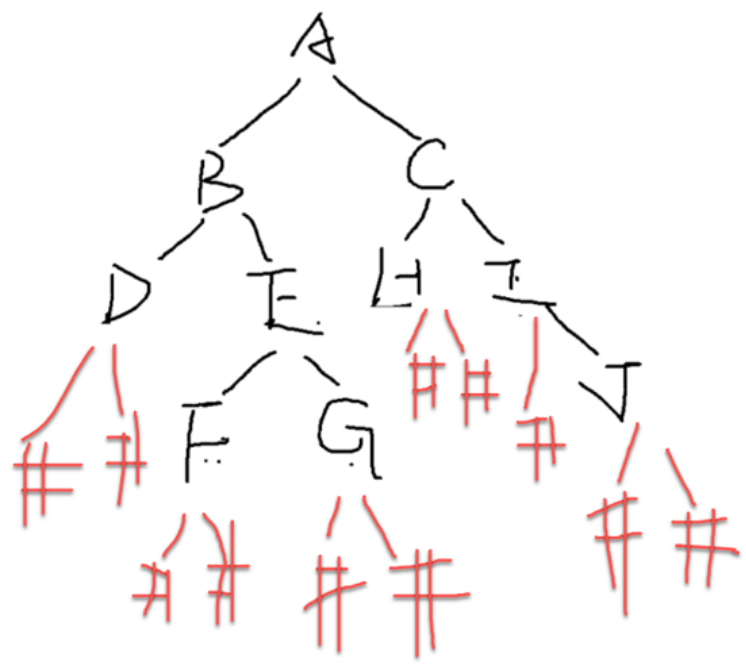
```c
#include "bitree.h"
//  BiTreeCompleteCreate(3, 1)
//     1
//    /  \
//   2    3
bitree_t* BiTreeCompleteCreate(int n, int i)
{
    bitree_t* root;

    root = (bitree_t*)malloc(sizeof(*root));
    if (root == NULL) {
        printf("%s malloc memory error\n", __func__);
        return NULL;
    }
    root->data = i; // 将根节点的编号存入
    // 当2*i≤n时，有左孩子，其编号为2*i ,否则没有左孩子，本身是叶节点;
    if ((2 * i) <= n) {
        // 有左孩子
        root->lchild = BiTreeCompleteCreate(n, 2 * i);
    } else {
        // 没有左孩子
        root->lchild = NULL;
    }
    // 当2*i+1≤n时，有右孩子，其编号为2*i+1 ,否则没有右孩子;
    if ((2 * i + 1) <= n) {
        // 有右孩子
        root->rchild = BiTreeCompleteCreate(n, 2 * i + 1);
    } else {
        // 没有右孩子
        root->rchild = NULL;
    }

    return root;
}

// 根左右
void BiTreeFrontOrder(bitree_t* root)
{
    if (root == NULL)
        return;
    printf(" %d", root->data);
    BiTreeFrontOrder(root->lchild);
    BiTreeFrontOrder(root->rchild);
}
void BiTreeMiddleOrder(bitree_t* root)
{
    if (root == NULL)
        return;
    BiTreeMiddleOrder(root->lchild);
    printf(" %d", root->data);
    BiTreeMiddleOrder(root->rchild);
}
void BiTreeRearOrder(bitree_t* root)
{
    if (root == NULL)
```

```
        return;
    BiTreeRearOrder(root->lchild);
    BiTreeRearOrder(root->rchild);
    printf(" %d", root->data);
}
```

main.c

```
#include "bitree.h"

int main(int argc, const char* argv[])
{
    bitree_t *root;
    // 1.创建完全二叉树
    root = BiTreeCompleteCreate(12, 1);
    if (root == NULL)
        return -1;

    printf("先序：");
    BiTreeFrontOrder(root);
    putchar(10);

    printf("中序：");
    BiTreeMiddleOrder(root);
    putchar(10);

    printf("后序：");
    BiTreeRearOrder(root);
    putchar(10);

    return 0;
}
```

2.9普通二叉树创建



先序序列：

A B D E F G C H I J

中序序列：

D B F E G A H C I J

后序序列：

D F G E B H J I C A

输入顺序

ABD##EF##G##CH##I#J##

ABD##EF##G##CH##I#J##

bitree.h

```
#ifndef __BITREE_H__
#define __BITREE_H__

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

#define datatype int

typedef struct bitree {
    datatype data;
    struct bitree *lchild, *rchild;
} bitree_t;

bitree_t* BiTreeNormalCreate(void);
void BiTreeFrontOrder(bitree_t* root);
void BiTreeMiddleOrder(bitree_t* root);
```

```h
void BiTreeRearOrder(bitree_t* root);
#endif
```

bitree.c

```c
#include "bitree.h"

bitree_t* BiTreeNormalCreate(void)
{
    bitree_t* root;
    char ch;
    ch = getchar();
    if (ch == '#') {
        return NULL;
    }

    root = (bitree_t*)malloc(sizeof(*root));
    if (root == NULL) {
        printf("%s malloc memory error\n", __func__);
        return NULL;
    }
    root->data = ch;
    root->lchild = BiTreeNormalCreate();
    root->rchild = BiTreeNormalCreate();

    return root;
}

// 根左右
void BiTreeFrontOrder(bitree_t* root)
{
    if (root == NULL)
        return;
    printf(" %c", root->data);
    BiTreeFrontOrder(root->lchild);
    BiTreeFrontOrder(root->rchild);
}
void BiTreeMiddleOrder(bitree_t* root)
{
    if (root == NULL)
        return;
    BiTreeMiddleOrder(root->lchild);
    printf(" %c", root->data);
    BiTreeMiddleOrder(root->rchild);
}
void BiTreeRearOrder(bitree_t* root)
{
    if (root == NULL)
        return;
    BiTreeRearOrder(root->lchild);
    BiTreeRearOrder(root->rchild);
    printf(" %c", root->data);
}
```

main.c

```c
#include "bitree.h"

int main(int argc, const char* argv[])
{
    bitree_t *root;
    // 1.创建普通二叉树
    root = BiTreeNormalCreate();
    if (root == NULL)
        return -1;

    printf("先序：");
    BiTreeFrontOrder(root);
    putchar(10);

    printf("中序：");
    BiTreeMiddleOrder(root);
    putchar(10);

    printf("后序：");
    BiTreeRearOrder(root);
    putchar(10);

    return 0;
}
```

3.递归

在c语言中函数可以调用本身，那这样的函数就叫做递归函数。

递归函数在写的时候比较简单，但是递归函数在空间复杂度和

时间复杂度上都会比较高。有可能会出现栈溢出的现象。在设计

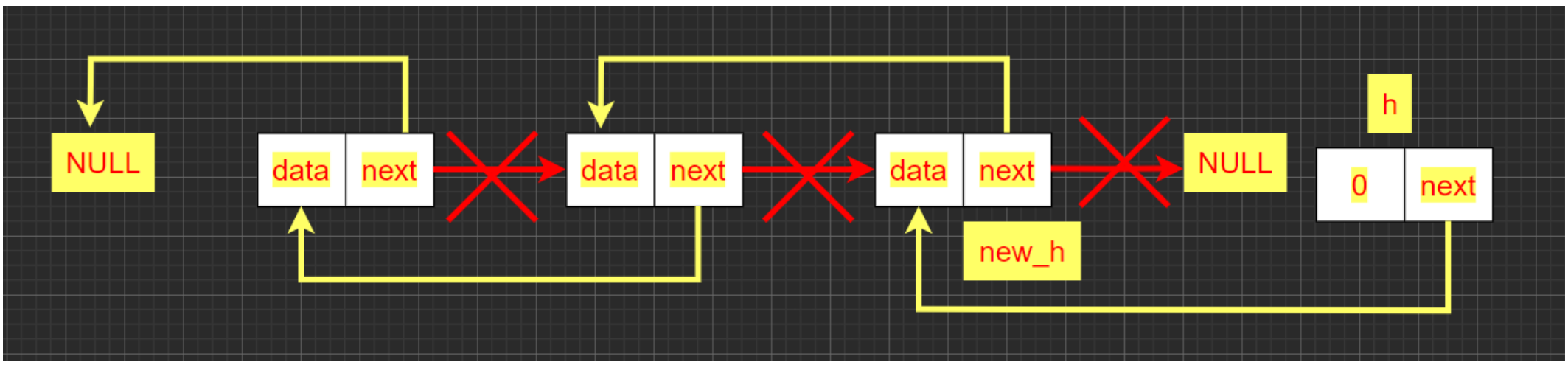
递归调用的时候一定要找到结束条件，否则就是死循环。

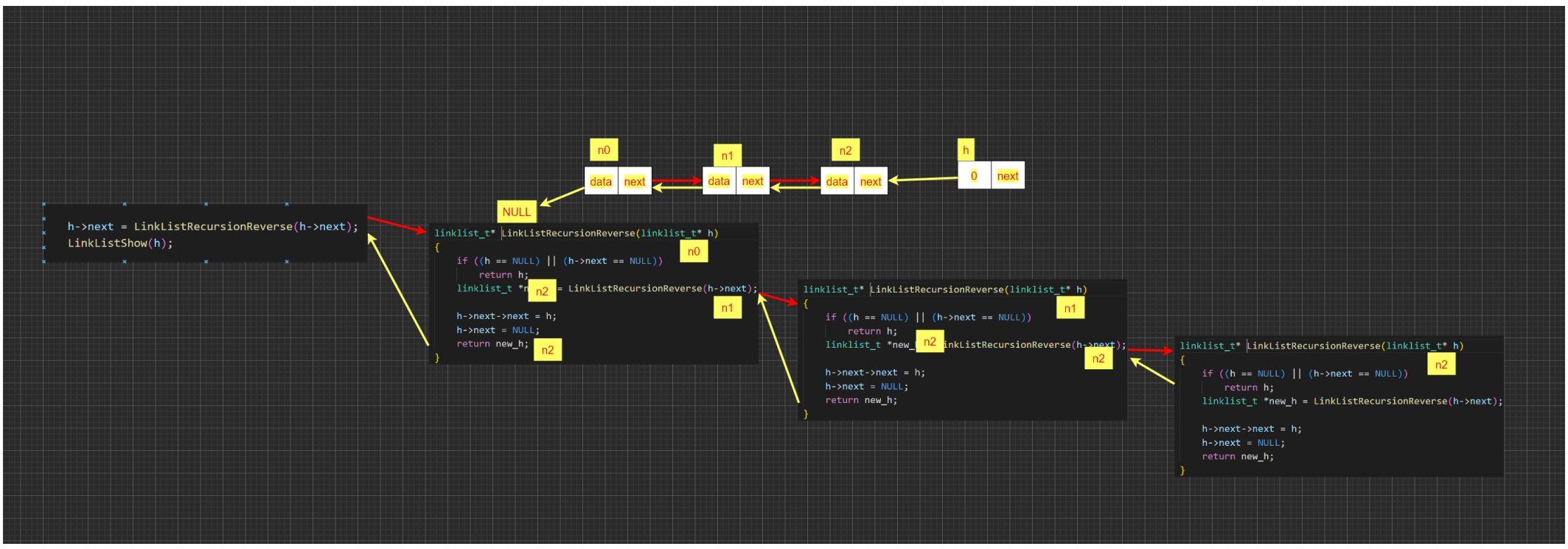
3.1使用递归实现n!阶乘

5! =5 * 4 * 3 * 2 * 1;

```c
#include <stdio.h>
// jicheng(5) :5*jicheng(4) : 5*4*3*2*1
// jicheng(4) :4*jicheng(3) : 4*3*2*1
// jicheng(3) :3*jicheng(2) : 3*2*1
// jicheng(2) :2*jicheng(1) : 2*1
// jicheng(1) :1
int jicheng(int n)
{
    if (n == 1)
        return 1;
    return n*jicheng(n-1);
}
int main(int argc, const char* argv[])
{
    int n = 5;
    printf("%d\n",jicheng(n));

    return 0;
}
```

3.2使用递归实现单链表逆序






```c
linklist_t* LinkListRecursionReverse(linklist_t* h)
{
    if ((h == NULL) || (h->next == NULL))
        return h;
    linklist_t *new_h = LinkListRecursionReverse(h->next);

    h->next->next = h;
    h->next = NULL;
    return new_h;
}
//在main.c中的调用方式
//h->next = LinkListRecursionReverse(h->next);
```