数据结构第四讲

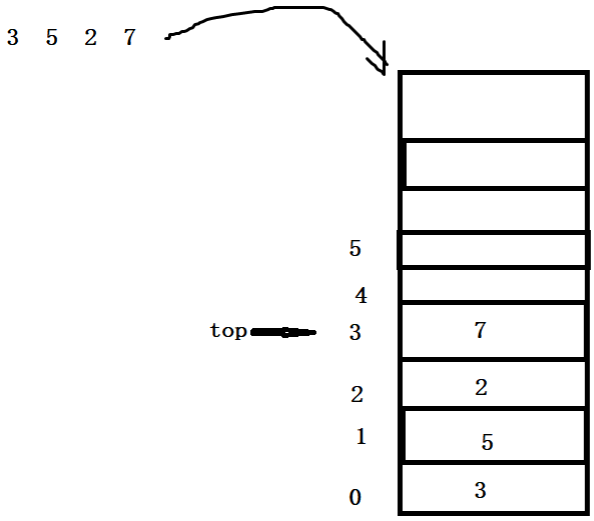
一、栈

1.1 栈的概念

1> 栈是操作受限的线性表：只能在同一端进行插入和删除操作的线性表
2> 特点：先进后出（FILO）、后进先出（LIFO）
3> 允许操作的一端被称为栈顶，不能操作的一段称为栈底
4> 注意：区分数据结构中的栈，跟内存分配的栈不是一个概念
5> 存储结构有两种：
顺序栈：顺序存储的栈
链式栈：链式存储的栈
6> 基本操作：创建、判空、判满、入栈、出栈、遍历栈

1.2 顺序栈结构体类型

提供一个数组，存放栈；还要提供一个变量记录栈顶元素的下标




```
#define MAX 8
typedef int datatype;
typedef struct
{
    datatype data[MAX];          //存放栈的数组
    int top;                     //记录栈顶元素的下标
}seqStack;

```

1.3 顺序表创建

```
seqStack *stack_create()
{
    seqStack *S = (seqStack*)malloc(sizeof(seqStack));
    if(NULL==S)
    {
        printf("创建失败\n");
        return NULL;
    }

    //初始化
    memset(S->data, 0, sizeof(S->data));
    S->top = -1;            //栈顶元素初始化为-1

    printf("创建成功\n");
    return S;
}
```

1.4 判空判满

```
//判空
int stack_empty(seqStack *S)
{
    return S->top==-1 ? 1:0;      //1表示空   0表示非空
}

//判满
int stack_full(seqStack *S)
{
    return S->top==MAX-1 ? 1:0;      //1表示满，0表示非满
}

```

1.5 入栈、进栈、压栈

先加后压

```
int stack_push(seqStack *S, datatype e)
{
    //判断逻辑
    if(NULL==S || stack_full(S))
    {
        printf("入栈失败\n");
```

```
        return -1;
    }

    //入栈逻辑
    S->top++;
    S->data[S->top] = e;

    printf("入栈成功\n");
    return 0;
}
```

1.6 遍历栈

```
void stack_show(seqStack *S)
{
    //判断逻辑
    if(NULL==S || stack_empty(S))
    {
        printf("遍历失败\n");
        return ;
    }

    //遍历逻辑
    printf("从栈顶到栈底元素分别是：");
    for(int i=S->top; i>=0; i--)
    {
        printf("%d\t", S->data[i]);
    }
    printf("\n");

}
```

1.7 出栈、弹栈

先弹后减

```
//出栈、弹栈：先弹后减
int stack_pop(seqStack *S)
{
    //判断逻辑
    if(NULL==S || stack_empty(S))
    {
        printf("出栈失败\n");
        return -1;
```

```
    }

    //出栈逻辑
    printf("%d出栈成功\n", S->data[S->top]);
    S->top--;
}
```

1.8 销毁栈

```
void stack_free(seqStack *S)
{
    //判断逻辑
    if(NULL==S)
    {
        printf("释放失败\n");
        return;
    }
    //释放
    free(S);
    S = NULL;
    printf("释放成功\n");
}
```

二、链式栈

要实现一个链式栈，其实就是一个单链表

1> 用单链表的头插完成入栈、头删完成出栈
此时，链表的头部就是栈顶，链表的尾部就是栈底

2> 用单链表的尾插完成入栈、尾删完成出栈
此时，链表的尾部就是栈顶、链表的头部就是栈底

三、队列

3.1 队列理论知识

1> 队列也是操作受限的线性表，只允许在一端进行插入、另一端进行删除

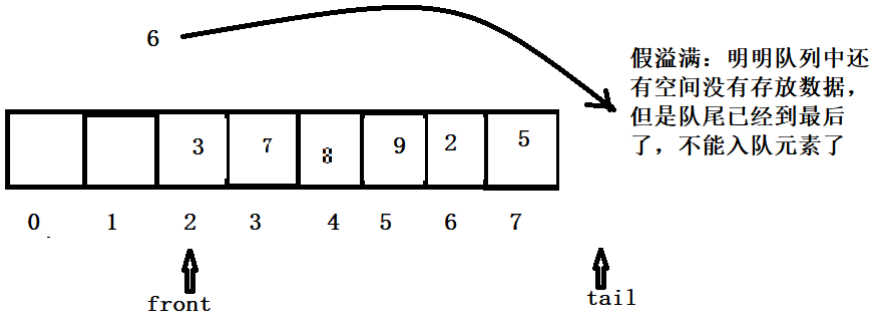
2> 能插入的一端称为队尾、能删除的一端称为队头

3> 特点：先进先出（FIFO）

4> 顺序存储的栈叫顺序队列、链式存储的栈叫链队列

3.2 顺序队列

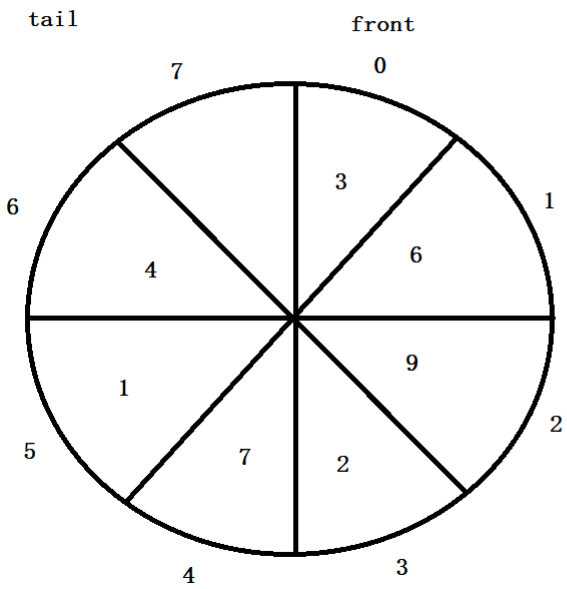
在数据结构中，很少使用普通顺序队列，因为会存在假溢满现象



为了解决假溢满现象，我们引入循环顺序队列，如果记录队头汇总队尾的变量，达到了最后一个元素，通过处理后，会记录第一个元素

3.3 循环顺序队列结构体类型

```
#define MAX 8
typedef int datatype;

typedef struct
{
    datatype data[MAX];      //存放队列的数组
    int front;                 //存放对头所在的下标
    int tail;                  //存放队尾所在的下标
}seQueue;
对头：记录的是第一个元素的下标
队尾：记录的是最后一个元素的下一个空位置
```



如果数组中所有元素全部使用的话，队伍的空状态和满状态，对头和队伍都相等，此时无法进行判断。
我们采用人为浪费一个存储单元的方法，当对头和队尾相等时为空，相差1时为满

3.4 创建队伍

```
seQueue *queue_create()
{
    seQueue *Q = (seQueue *)malloc(sizeof(seQueue));
    if(NULL==Q)
    {
        printf("创建失败\n");
        return NULL;
    }

    //初始化
    Q->front = 0;
    Q->tail = 0;

    printf("创建成功\n");
    return Q;
}
```

3.5 判空、判满

```
//判空
int queue_empty(seQueue *Q)
{
    return Q->front==Q->tail ? 1:0;   //1表示空   0表示非空
}

//判满
int queue_full(seQueue *Q)
{
    return (Q->tail+1)%MAX==Q->front ? 1:0;   //1表示满   0表示非满
}
```

3.6 入队

```
int queue_push(seQueue *Q, datatype e)
{
    //判断逻辑
    if(NULL==Q || queue_full(Q))
    {
        printf("入队失败\n");
        return -1;
```

```
    }

    //入队逻辑
    Q->data[Q->tail] = e;        //将数据放到队尾所占位置

    //队尾后移
    Q->tail = (Q->tail+1)%MAX;    //对MAX取余为了解决队尾到最后位置
    printf("入队成功\n");
    return 0;
}
```

3.7 遍历

```
void queue_show(seQueue *Q)
{
    //判断逻辑
    if(NULL==Q || queue_empty(Q))
    {
        printf("遍历失败\n");
        return ;
    }

    //遍历逻辑
    printf("从队头到队尾元素分别是: ");
    for(int i=Q->front; i!=Q->tail; i=(i+1)%MAX)
    {
        printf("%d\t", Q->data[i]);
    }
    printf("\n");

}
```

3.8 出队

```
int queue_pop(seQueue *Q)
{
    //判断逻辑
    if(NULL==Q || queue_empty(Q))
    {
        printf("出队失败\n");
        return -1;
    }

    //出队逻辑
```

```
    printf("%d出队成功\n", Q->data[Q->front]);

    //队头后移
    Q->front = (Q->front+1)%MAX;     ///对MAX取余为了解决队尾到最后位置

}
```

3.9 销毁队伍

```
void queue_free(seQueue *Q)
{
    //判断逻辑
    if(NULL==Q)
    {
        printf("销毁失败\n");
        return;
    }
    //销毁
    free(Q);
    Q = NULL;
    printf("销毁成功\n");

}
```

四、链式队列

1> 链式存储的队列叫做链式队列

2> 实现方式

单链表头插法完成入队、尾删法完成出队：此时链表的头部称为队列的尾部，链表的尾部称为队列的头部

单链表尾插法完成入队、头删法完成出队：此时链表的头部称为队列的头部，链表的尾部称为队列的尾部

3> 以上两种实现方式，都要涉及到对尾部的操作，每次对尾部进行处理时，都要遍历整条链表，不方便

4> 此时，可以单独再定义一个指针，指向尾节点进行操作；这时，头结点指向链表的头部，尾指针指向链表的尾部，操作起来比较方便，没必要每次都要遍历了

4.1 链式队列结构体定义

```
typedef int datatype;

//定义节点结构体
```

```c
typedef struct Node
{
    union{
        int len;          //头结点数据域
        datatype data;      //普通节点数据域
    };

    struct Node *next;          //指针域
}Node;

//定义队列的结构体
typedef struct
{
    Node *head;         //指向链表头部
    Node *tail;         //指向链表尾部
}linkQueue;


```

4.1 创建

1> 先创建一个队列，会生成两个野指针
2> 申请一个头节点，并初始化
3> 将队列的两个指针都指向头节点

```c
linkQueue *queue_create()
{
    //创建队伍
    linkQueue *L = (linkQueue*)malloc(sizeof(linkQueue));
    if(NULL==L)
    {
        printf("队伍创建失败\n");
        return NULL;
    }
    //此时创建出两个野指针 L->head   L->tail

    //创建一个头结点，赋值给这两个野指针
    L->head =  (Node*)malloc(sizeof(Node));
    if(NULL == L->head)
    {
        printf("节点申请失败\n");
        return NULL;
    }
    //给头结点初始化
    L->head->len = 0;
```

```
    L->head->next = NULL;

    L->tail = L->head;       //将两个指针同时指向头结点
    printf("创建成功\n");
    return L;                //将初始化好的链式队列返回

}
```

4.3 判空

当两个指针相等时，为空队

```
int queue_empty(linkQueue *L)
{
    return L->head==L->tail ? 1:0;    //1表示空   0表示非空
}
```

4.4 入队

1> 判断逻辑：只要队伍合法就可以入队
2> 申请节点封装数据
3> 将封装好数据的节点连接到队尾指针指向的节点后面
4> 更新队尾指针的指向

```
int queue_push(linkQueue *L, datatype e)
{
    //1>    判断逻辑：只要队伍合法就可以入队
    if(NULL==L)
    {
        printf("所给队伍不合法\n");
        return -1;
    }

    //2>    申请节点封装数据
    Node *p = (Node*)malloc(sizeof(Node));
    if(NULL==p)
    {
        printf("节点申请失败\n");
        return -2;
    }
    p->data = e;
    p->next = NULL;

    //3>    将封装好数据的节点连接到队尾指针指向的节点后面
```

```
    L->tail->next = p;

    //4>    更新队尾指针的指向
    L->tail = p;

    L->head->len++;           //队列长度自增
    printf("入队成功\n");
    return 0;
}

```

4.5 遍历

```
void queue_show(linkQueue *L)
{
    //判断逻辑
    if(NULL==L || queue_empty(L))
    {
        printf("遍历失败\n");
        return ;
    }

    //遍历
    printf("从队头到队尾分别是: ");
    Node *q = L->head->next;          //定义遍历指针指向第一个节点
    while(q != NULL)
    {
        printf("%d\t", q->data);
        q = q->next;
    }

    printf("\n");
}
```

4.6 出队

```
int queue_pop(linkQueue *L)
{
    //判断逻辑
    if(NULL==L || queue_empty(L))
    {
        printf("删除失败\n");
        return -1;
    }
```

```
9
10      //删除逻辑
11      Node *p = L->head->next;        //保存要删除的节点
12      L->head->next = p->next;         //孤立要删除的节点
13
14      printf("%d出队成功\n", p->data);
15      free(p);                    //释放
16
17      //队伍变化
18      L->head->len--;
19
20      if(L->head->next == NULL)         //判断是否全部删完
21      {
22          L->tail = L->head;
23      }
24      return 0;
25  }
```

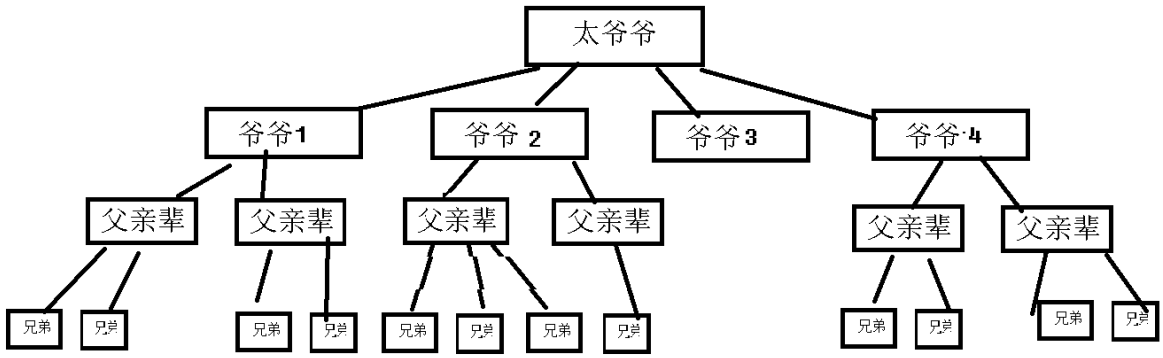
4.7 销毁

```
1   void queue_free(linkQueue *L)
2   {
3       //判断逻辑
4       if(NULL==L)
5       {
6           printf("释放失败\n");
7           return;
8       }
9       //将所有节点释放
10      while(L->head != L->tail)
11      {
12          queue_pop(L);
13      }
14
15      //释放头结点
16      free(L->head);
17      L->head = L->tail = NULL;
18
19      //释放队列
20      free(L);
21      L = NULL;
22      printf("释放成功\n");
23  }
```

五、树形结构

【树形结构】元素之间存在一对多的关系
【树】树是由根结点和若干棵子树构成的树形结构
【节点】结点是树中的数据元素
【父亲节点】当前结点的直接上级结点
【孩子节点】定义：孩子结点是指当前结点的直接下级结点
【兄弟节点】定义：兄弟结点是有相同父结点的结点
【堂兄弟节点】定义:堂兄弟结点是父结点在同一层的结点
【祖先节点】定义: 祖先结点是当前结点的直接及间接上级结点
【子孙节点】定义: 子孙结点是当前结点的直接及间接下级结点

【根结点】定义: 根结点是没有父结点的结点
【叶子节点】定义: 叶结点是度为0的结点
【分支节点】定义: 分支结点是度不为0的结点
【节点的度】结点的度是结点含有子树的个数
【节点的层次】定义：结点的层次是从根结点到某结点所经路径上的层次数
【树的度】定义: 树的度是树内各结点度的最大值
【树的深度】定义：树的深度是树中所有结点层次的最大值



六、二叉树

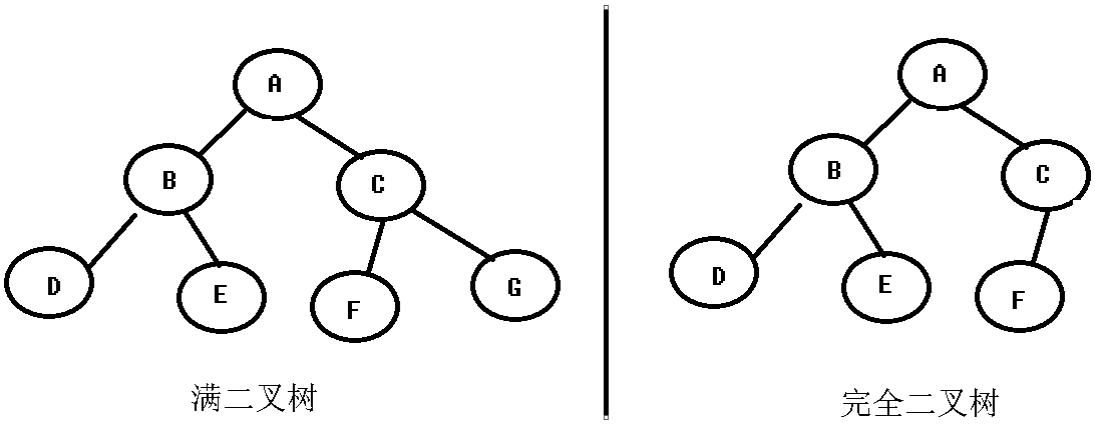
6.1 定义：

每个节点最多有两棵子树，并且严格区分左右的树形结构

6.2 二叉树相关概念

【左子树】左子树是以当前结点的左孩子结点作为根节点的子树
【右子树】右子树是以当前结点的右孩子结点作为根节点的子树
【满二叉树】满二叉树是最后一层是叶子结点，其余结点度是2的二叉树
在不增加层次的基础上，不能在添加节点的二叉树
【完全二叉树】完全二叉树是在一棵满二叉树基础上自左向右连续增加叶子结点得到的二叉树

完全二叉树是在一棵满二叉树基础上自右向左连续删除叶子结点得到的二叉树



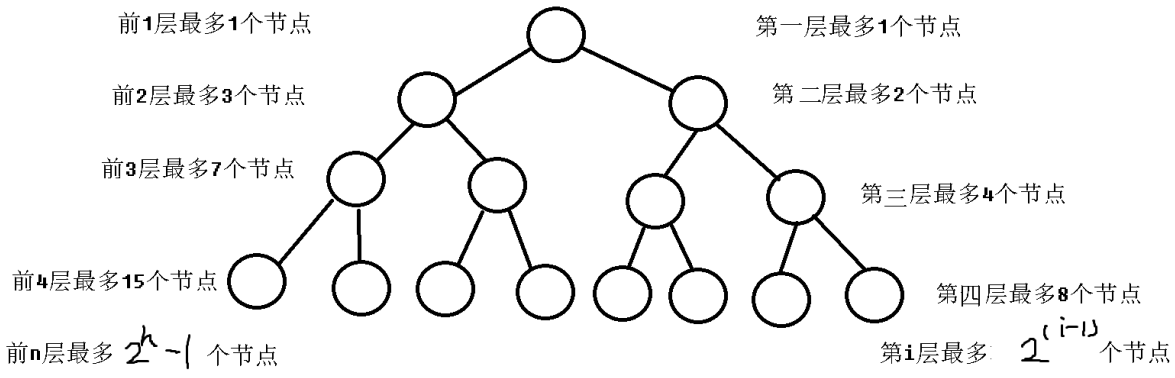
6.3 二叉树的五种状态

空二叉树、只有根结点、只有左子树、只有右子树、既有左子树也有右子树

6.4 二叉树的性质

性质1：在二叉树的第i层上最多有2^(i-1)个结点(i >= 1)

性质2：深度为k的二叉树最多有2^k-1个结点(k >= 1)



性质3：在任意一棵二叉树中，叶子结点的数目比度数为2的结点的数目多一

```
假设 n0:度数为0的所有节点的个数
     n1：度数为1的所有节点的个数
      n2：度数为2的所有节点的个数

     总度数 = 总结点个数 - 1
     总个数 = n0 + n1 + n2;
     总度数 = 2*n2 + n1

      最终结果：n2 = n0 - 1
```

```
性质4：如果第i个节点存在左孩子，那么其左孩子应该在树中第2*i个位置
       如果第i个节点存在右孩子，那么其右孩子应该在树中第2*i+1个位置
```

作业

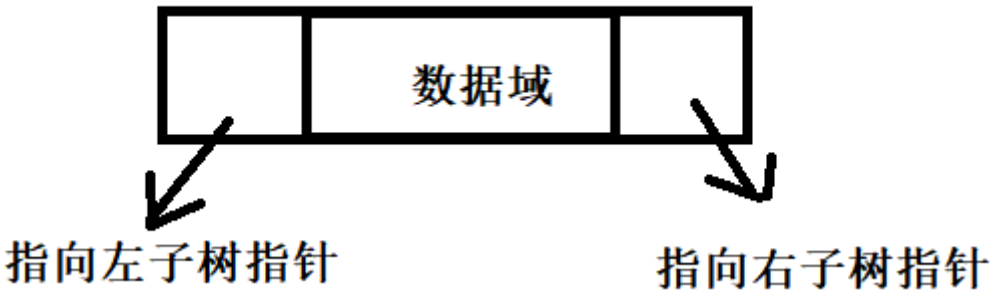
作业1：使用栈的特点完成进制转换

作业2：使用xmind软件，将线性表的知识点进行复盘，对于某些重点知识点，要将核心代码附上

6.5 二叉树存储

1> 顺序存储：由于在二叉树的存储中，空树也要占一个位置，导致会浪费大量的空间，所以，存储二叉树，不常使用顺序存储

2> 链式存储：



6.6 节点结构体类型

```
typedef char datatype;
typedef struct Node
{
    datatype data;              //数据域
    struct Node *left;          //指向左子树的指针
    struct Node *right;         //指向右子树的指针
}biTree;
```

6.7 创建：先序创建

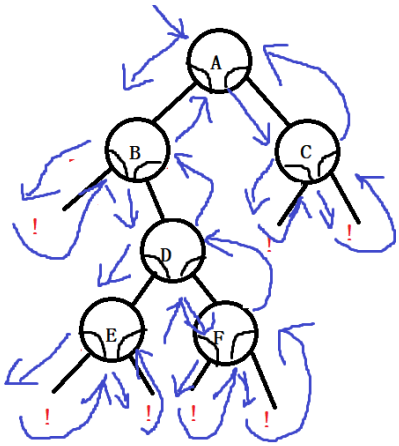
先创建根节点、接着创建左子树、最后创建右子树

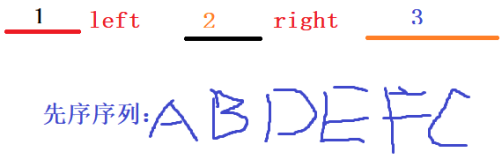
```
biTree *tree_create()
{
    //定义一个字符数据，存放用户输入的数据
    char ch = '\0';

    scanf("%c", &ch);
    getchar();

    if(ch == '!')     //如果是‘！’说明空节点
    {
        return NULL;
    }

    //申请节点保存数据
    biTree *B = (biTree *)malloc(sizeof(biTree));
    if(NULL==B)
    {
        printf("空间申请失败\n");
```

```
        return NULL;
    }

    B->data = ch;       //将数据存入节点

    B->left = tree_create();    //递归创建左子树
    B->right = tree_create();   //递归创建右子树

    return B;       //将创建好的树，返回
}
```

6.7 先序遍历

也称先根遍历，先访问根节点中的内容，然后访问左子树、最后访问右子树





先序序列：ABDEFC

```
void pri_order(biTree *B)
{
      if(NULL==B)
      {
         return ;
       }
      printf("%c\t", B->date);    //先访问根节点
      pri_order(B->left);       //先序遍历左子树
      pri_order(B->right);       //先序遍历右子树
}
```

```
void pri_order(biTree *B)
{
      if(NULL==B)
      {
         return ;
      }
      printf("%c\t", B->data);    //先访问根节点
      pri_order(B->left);       //先序遍历左子树
      pri_order(B->right);       //先序遍历右子树
}

```

6.8 中序遍历

也称中根遍历，先访问左子树、然后访问根节点内容、最后访问右子树

```
void in_order(biTree *B)
{
	if(NULL==B)
	{
	   return ;
	}
	in_order(B->left);	//中序遍历左子树

	printf("%c\t", B->data);	//访问根节点

	in_order(B->right);	//中序遍历右子树
}
```

6.9 后续遍历

也称后根遍历，先访问左子树、然后访问右子树、最后访问根节点内容

```
void post_order(biTree *B)
{
	if(NULL==B)
	{
	   return ;
	}
	post_order(B->left);	//后序遍历左子树

	post_order(B->right);	//后序遍历右子树

	printf("%c\t", B->data);	//访问根节点
}
```