第三章 链表的实现与应用

主讲教师：同济大学电子与信息工程学院 陈宇飞
同济大学电子与信息工程学院 龚晓亮

目录

- 链式结构的基本概念
- 单链表的基本操作
- 双向链表的基本操作
- 二叉搜索树

单链表

循环链表

双向链表

3.1 链式结构的基本概念

- 数组的不足
  - 大小必须在定义时确定，导致空间浪费

    是否可以按需分配空间
  - 占用连续空间，导致小空间无法充分利用

    是否可以充分利用不连续的空间
  - 在插入/删除元素时必须前后移动元素

    插入/删除时能否不移动元素
- 链表

  不连续存放数据，用指针指向下一数据的存放地址

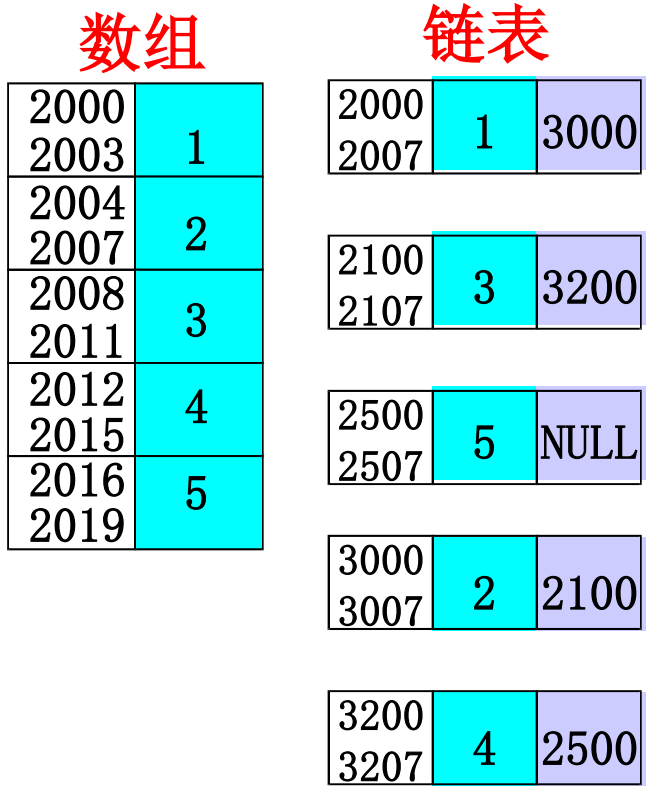
3.1 链式结构的基本概念

例：数据1，2，3，4，5，分别存放在数组和链表中

存放5个元素：

数组：连续的20字节

链表：非连续的40字节(每个结点的8字节连续)



3.1 链式结构的基本概念

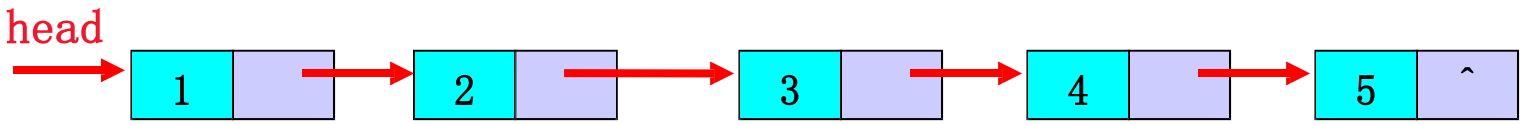
- 链表与数组的比较

数组	链表
大小在声明时固定	大小不固定
处理的数据个数有差异时，须按最大值声明	根据需要随时增加/减少结点
内存地址连续，可直接计算得到某个元素的地址	内存地址不连续，必须依次查找
逻辑上连续，物理上连续	逻辑上连续，物理上不连续

3.1 链式结构的基本概念

- 结点：存放数据的基本单位
  - 数据域：存放数据的值
  - 指针域：存放下一个同类型结点的地址
- 链表：由若干结点构成的链式结构
- 表头结点：第一个结点
- 表尾结点：链表的最后一个结点，指针域为NULL(空)
- 头指针：指向链表的表头结点的指针

链表的表示方法：



- 建立一个有5个结点的链表，学生的基本信息从键盘进行输入，假设键盘输入为：

```
Zhang    1001    m
Li       1002    f
Wang     1003    m
Zhao     1004    m
Qian     1005    f
```

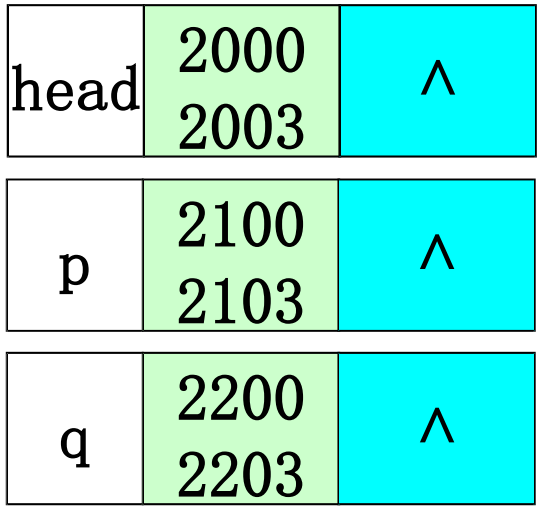
```
struct student {
    string name;
    int num;
    char sex;
    struct student *next;
};
```

//指向结构体自身的指针成员类型不允许是自身的结构体类型，但可以是指针(因为指针占用空间已知)

```
int main()
{   student *head=NULL, *p=NULL, *q=NULL;
    int i;
    for(i=0; i<5; i++) {
       if (i > 0)  q = p;
       p = new(nothrow) student;
       if (p == NULL)  return -1;
       if (i == 0)  head = p;
       else  q->next = p;
       cout << "请输入第" << i+1 << "个人的基本信息" << endl;
       cin >> p->name >> p->num >> p->sex;
       p->next = NULL;
     }
}    //是否完整?
```

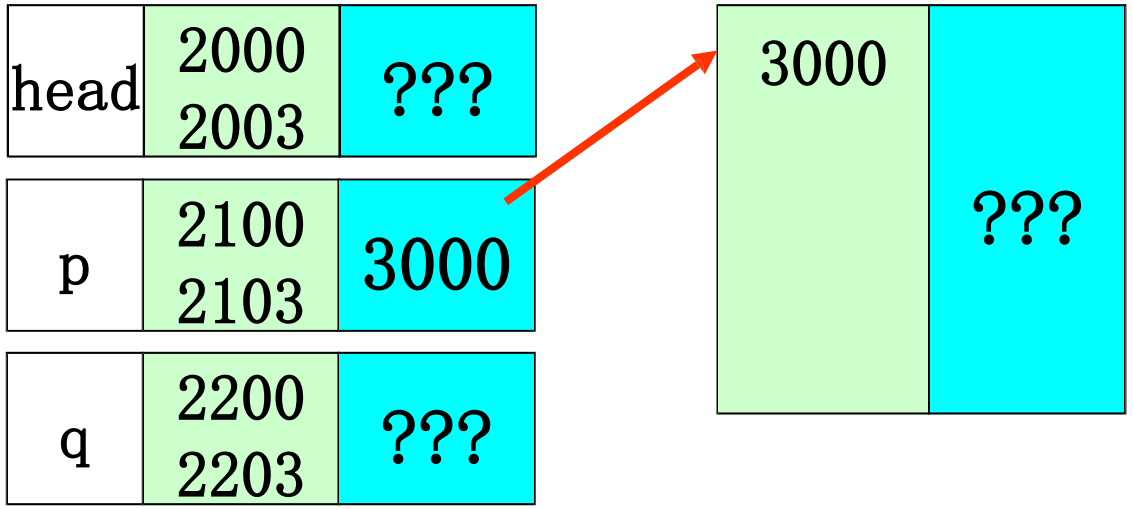
```
student *head=NULL, *p=NULL, *q=NULL; int i;
```

初始状态：



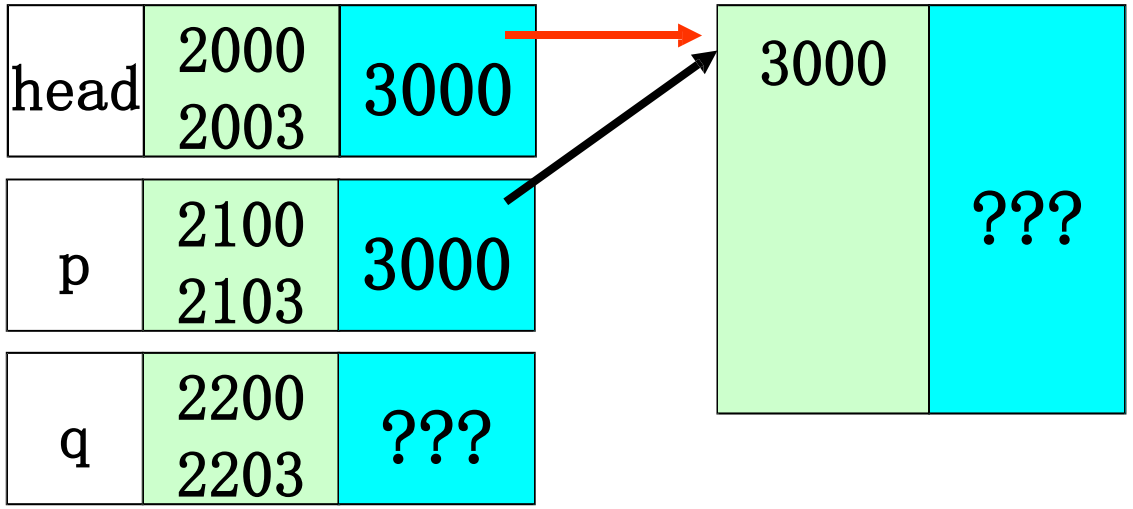
```
p = new(nothrow) student;
```

i=0的循环：



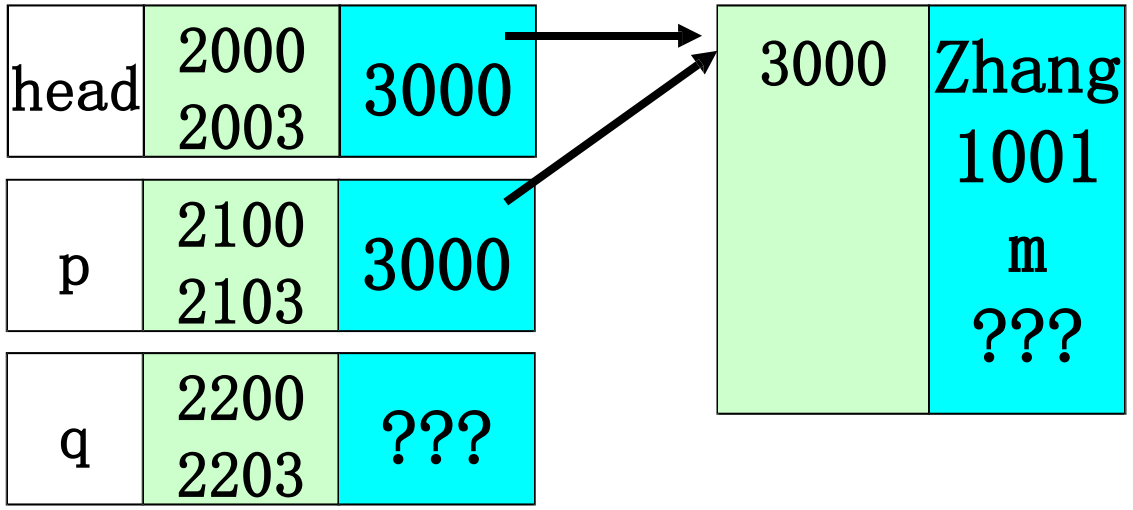
```
head = p; //head指向第1个结点
```

i=0的循环：



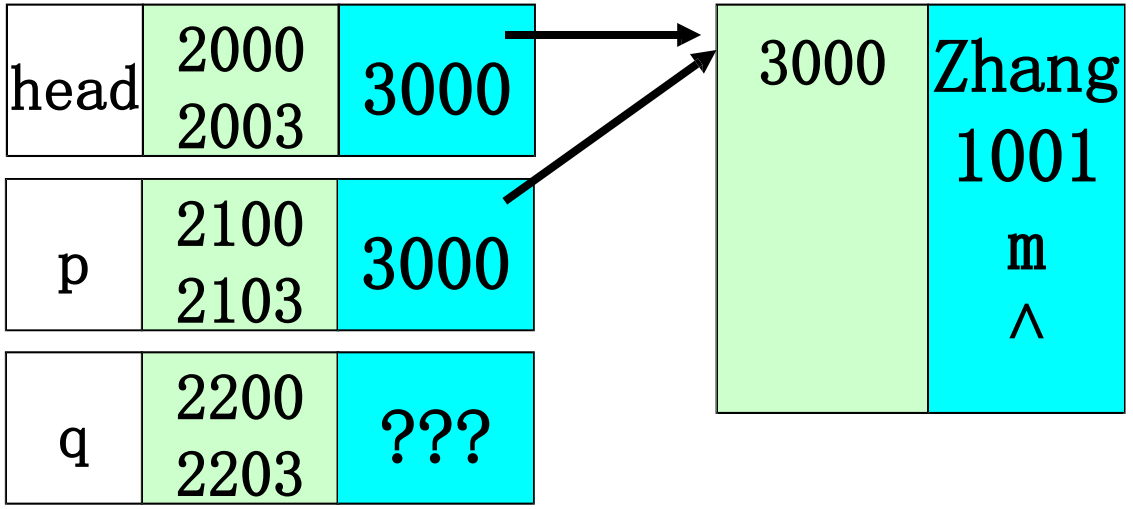
```
cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

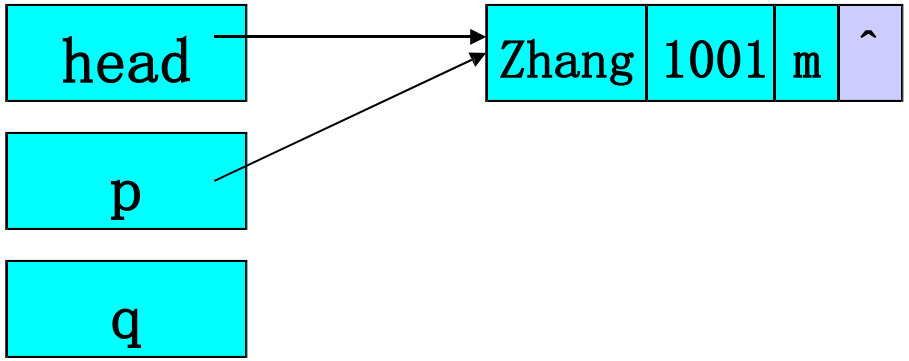
i=0的循环：



```
p->next = NULL;
```

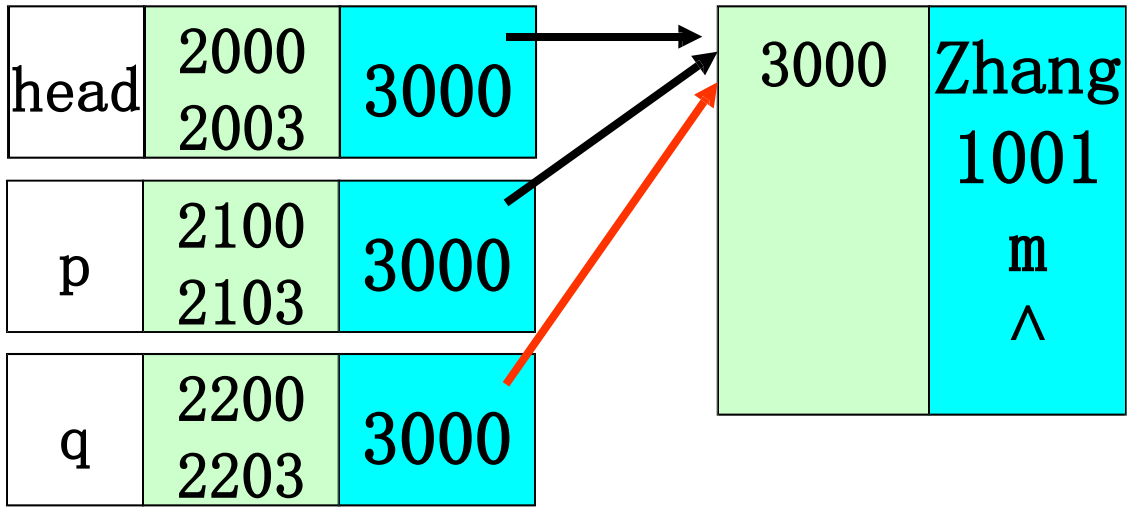
i=0的循环结束：





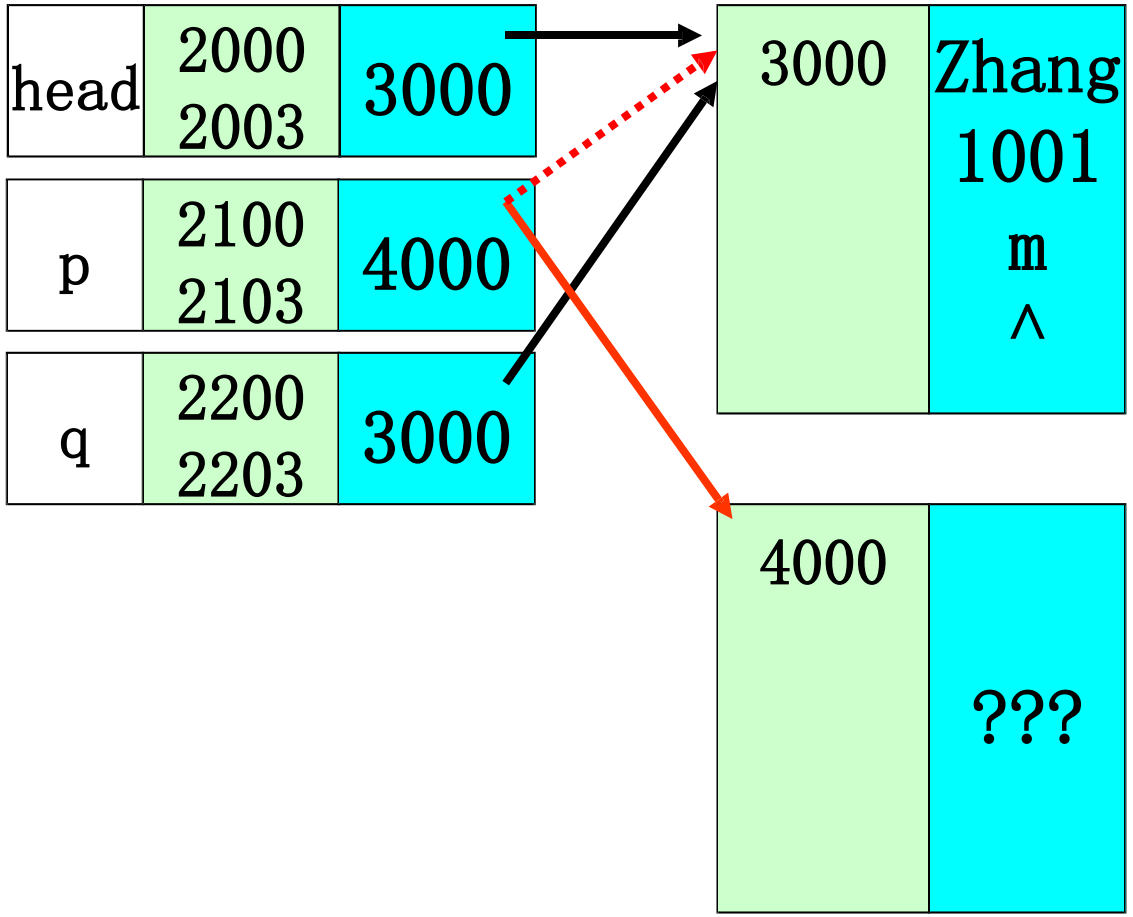
```
if (i > 0)   q = p;
```

i=1的循环：



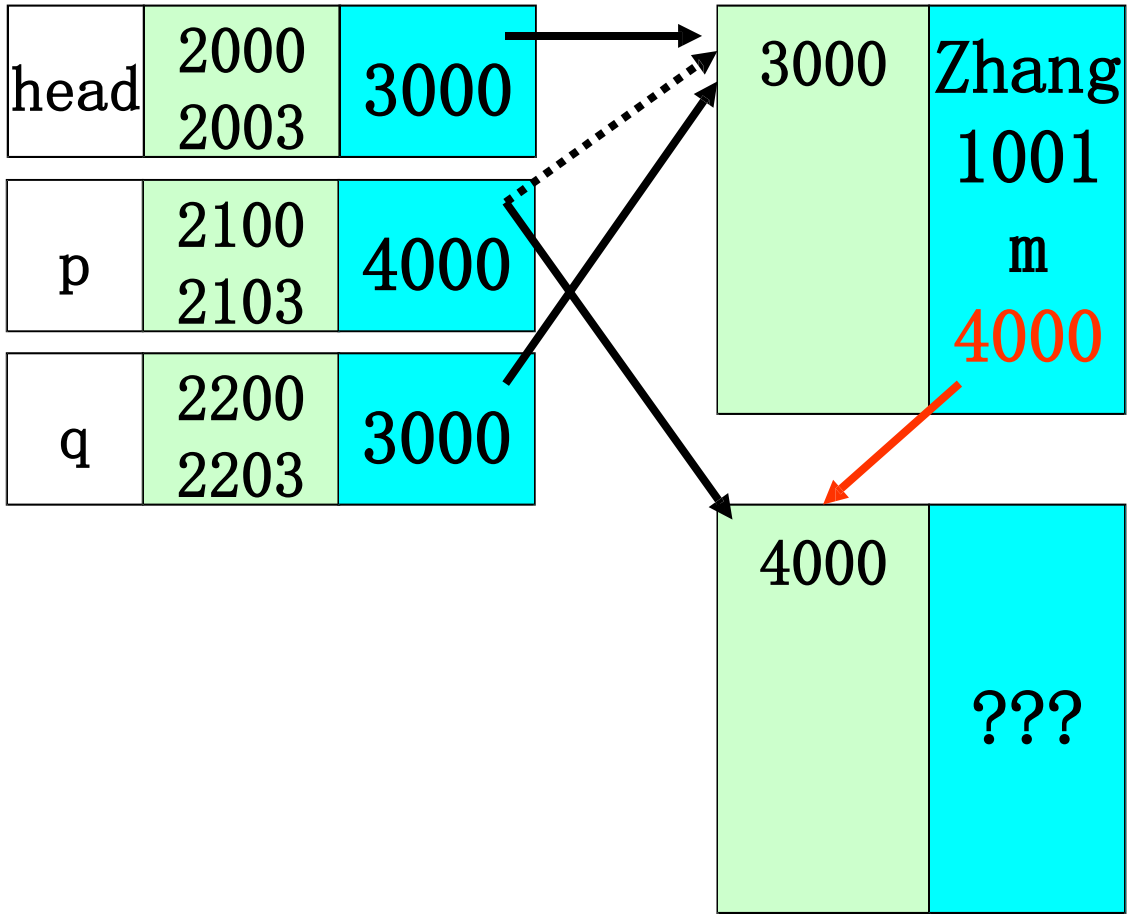
```
p = new(nothrow) student;
```

i=1的循环：



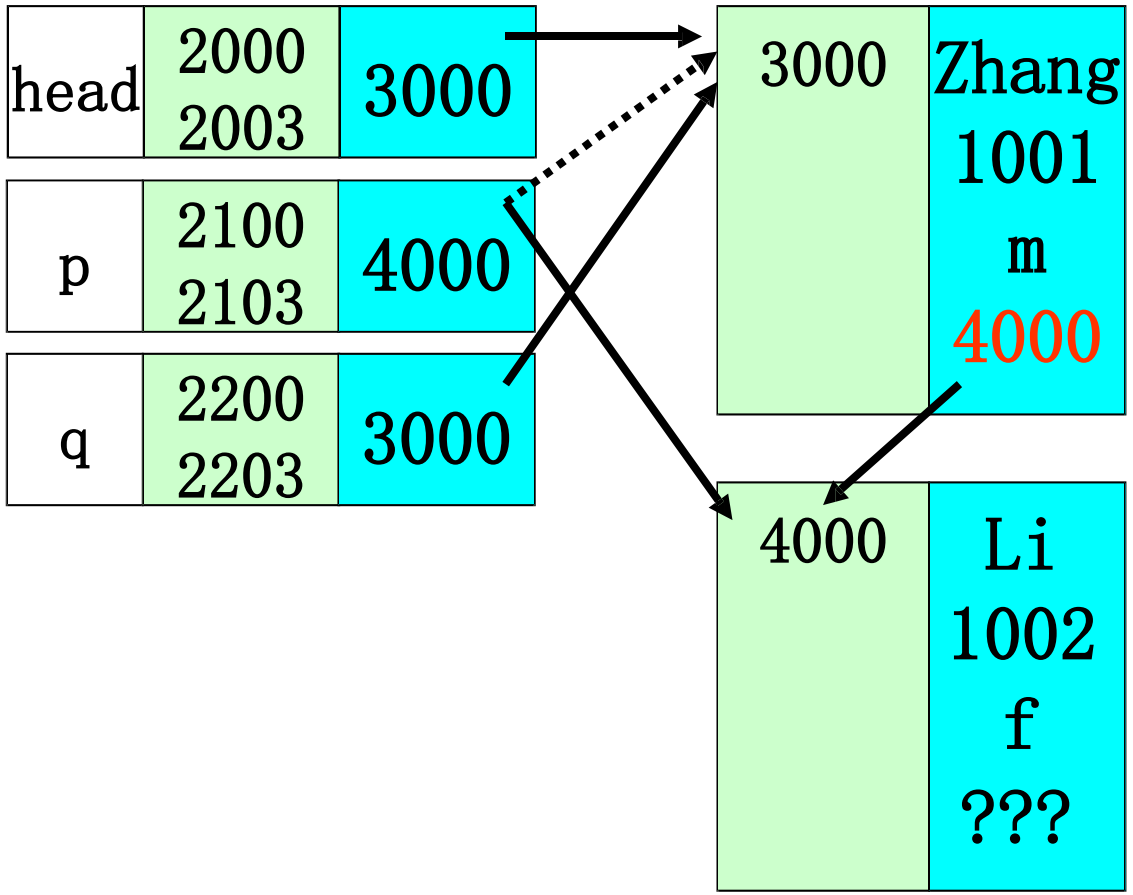
q->next = p;

i=1的循环：



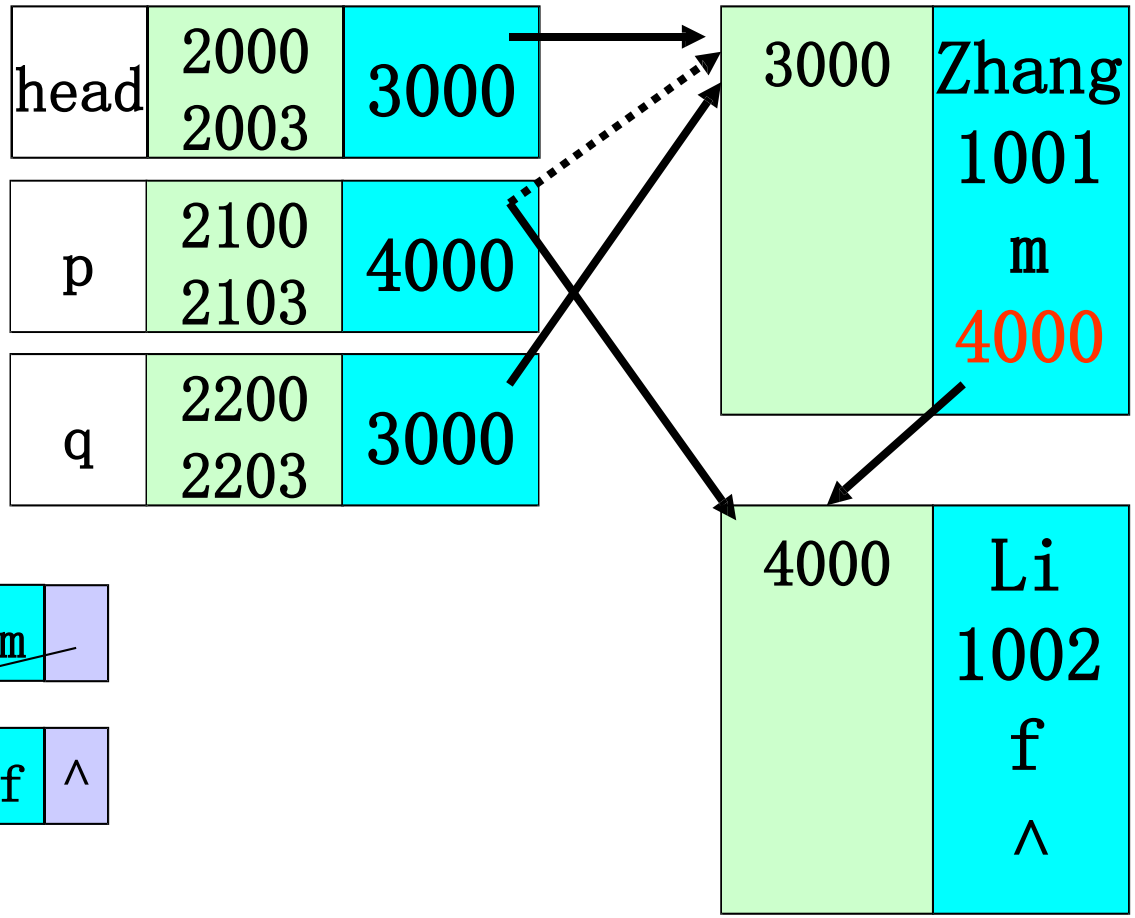
```
cin >> p->name >> p->num >> p->sex; //键盘输入基本信息
```

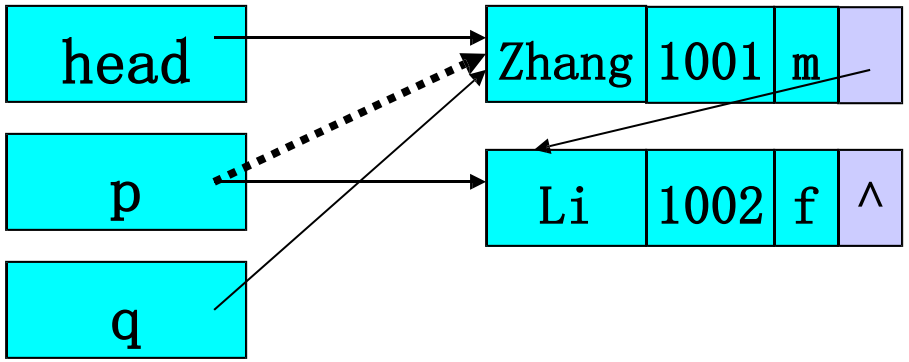
i=1的循环：



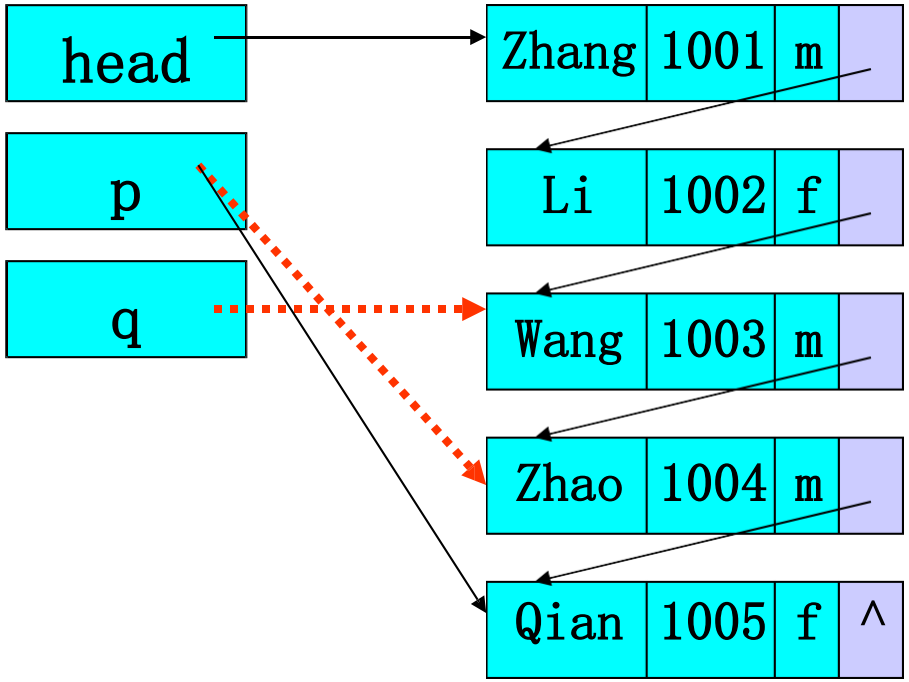
p->next = NULL;

i=1的循环结束：

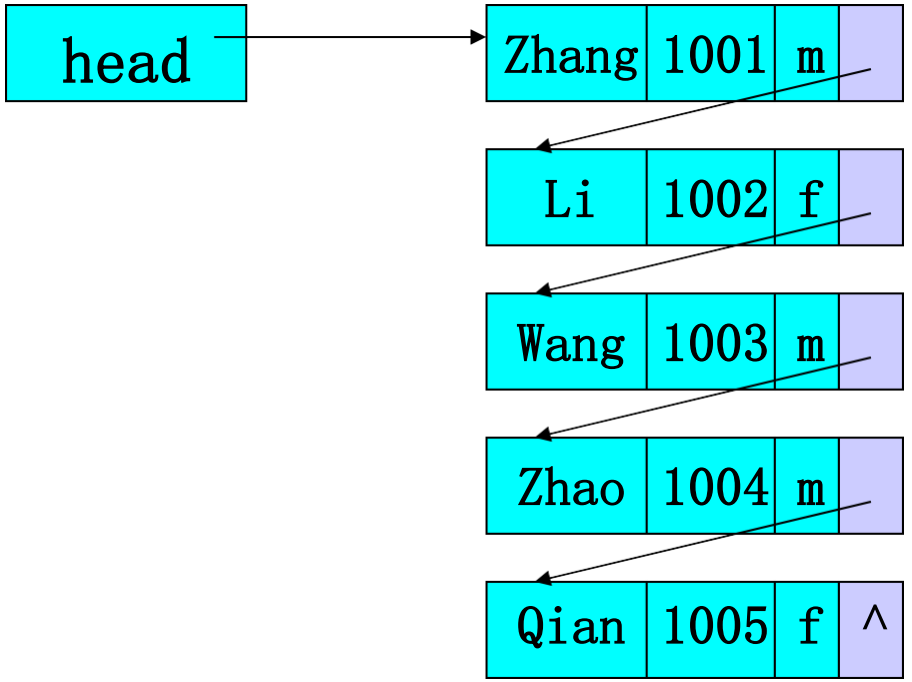




i=4的循环结束：



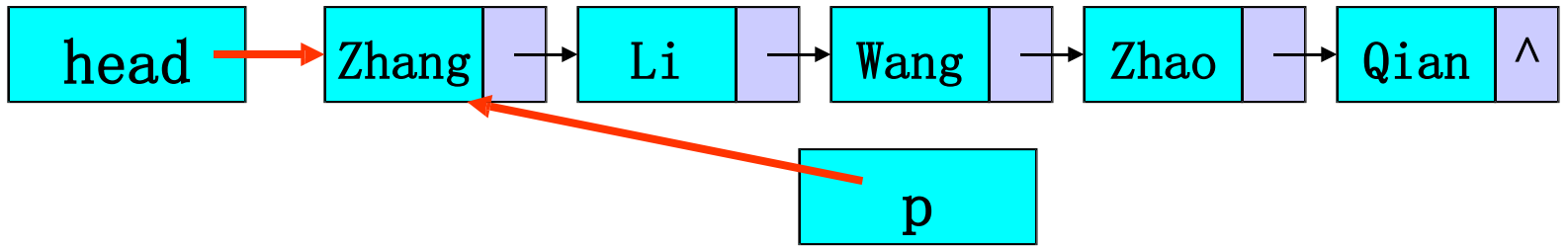
循环完成：



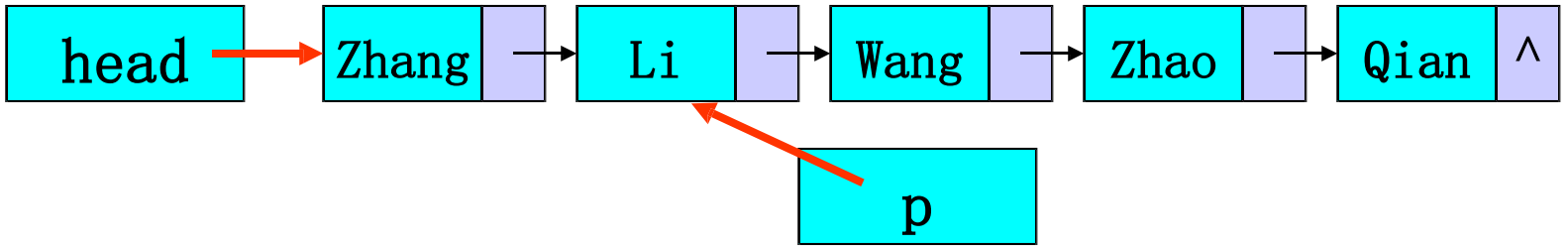
//在前例基础上完整程序

```
int main()
{   student *head = NULL, *p = NULL, *q = NULL; int i;
    for(i = 0; i < 5; i++) { …//刚才建立链表的循环}
    p = head; //p复位，指向第1个结点
    while(p != NULL) { //循环进行输出
        cout << p->name << " " << p->num << " " << p->sex << endl;
        p = p->next;
    }
    p = head; //p复位，指向第1个结点
    while(p) { //循环进行各结点释放
        q = p->next;
        delete p;
        p = q;
    }
    return 0;
}
```

p = head; //p复位，指向第1个结点



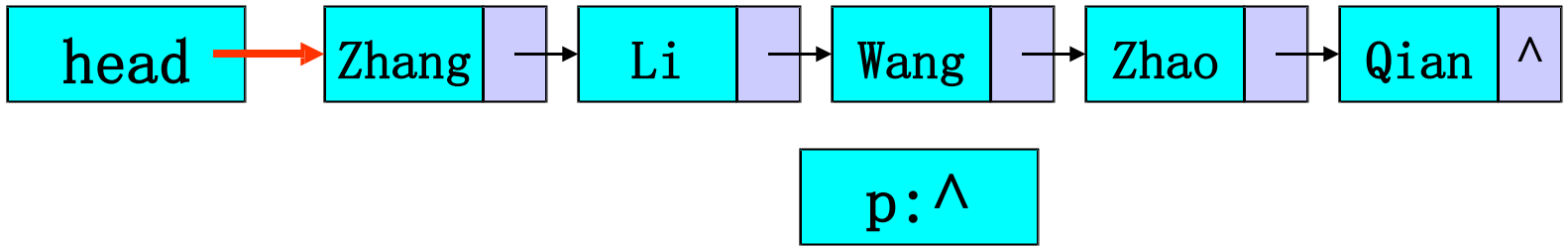
```
p = p->next;
```



Zhang	1001	m

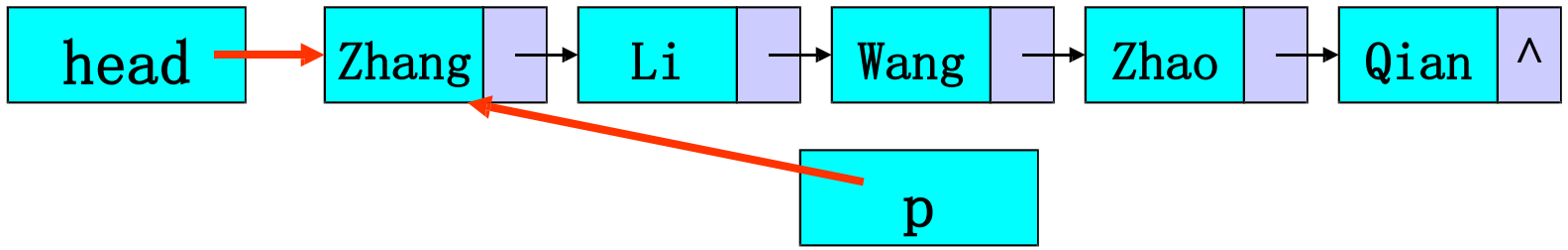
while(p!=NULL)

循环结束：

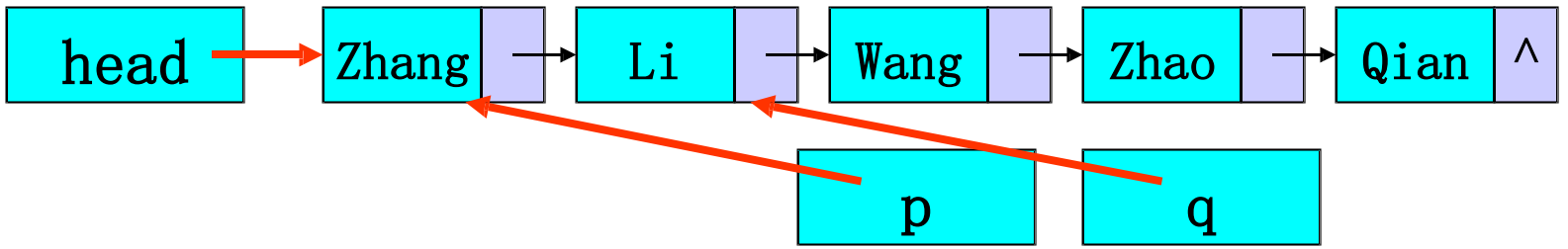



```
Zhang 1001 m
Li 1002 f
Wang 1003 m
Zhao 1004 m
Qian 1005 f
```

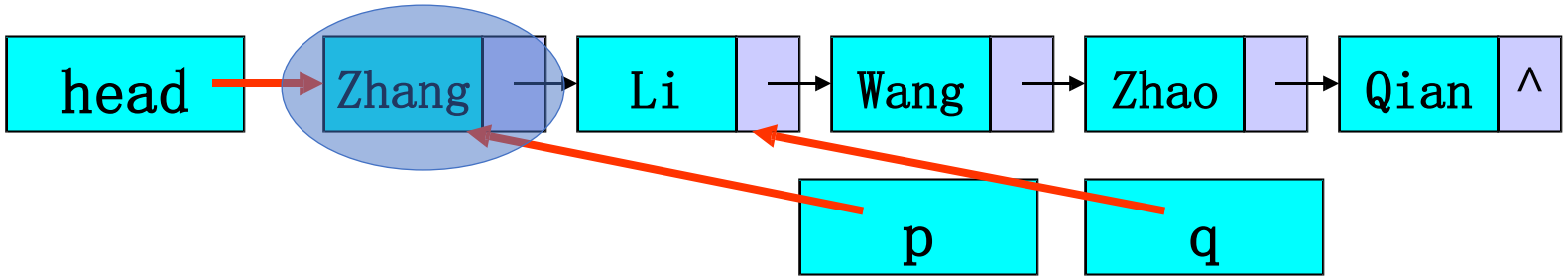
```
p = head; //p复位，指向第1个结点
```



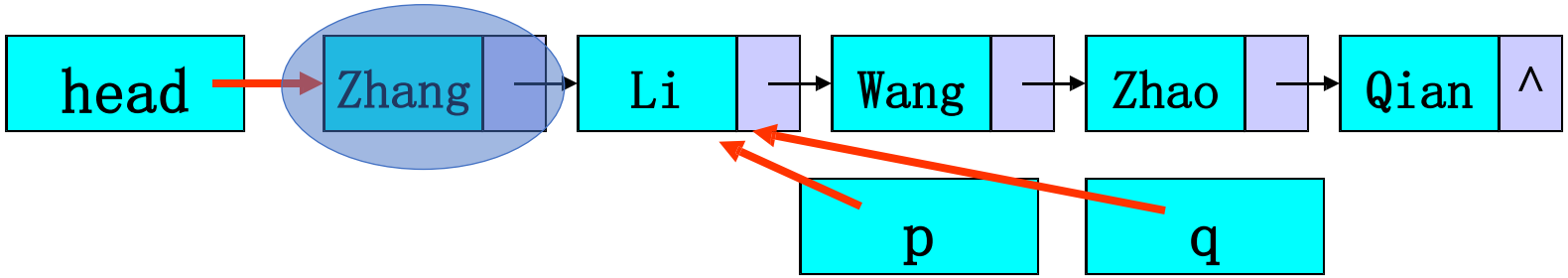
```
q = p->next;
```



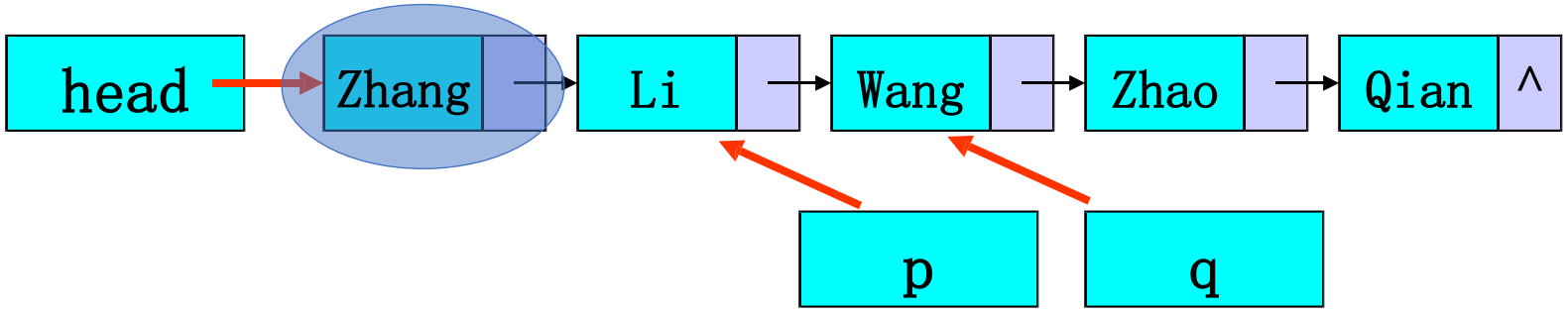
delete p;



```
p = q;
```



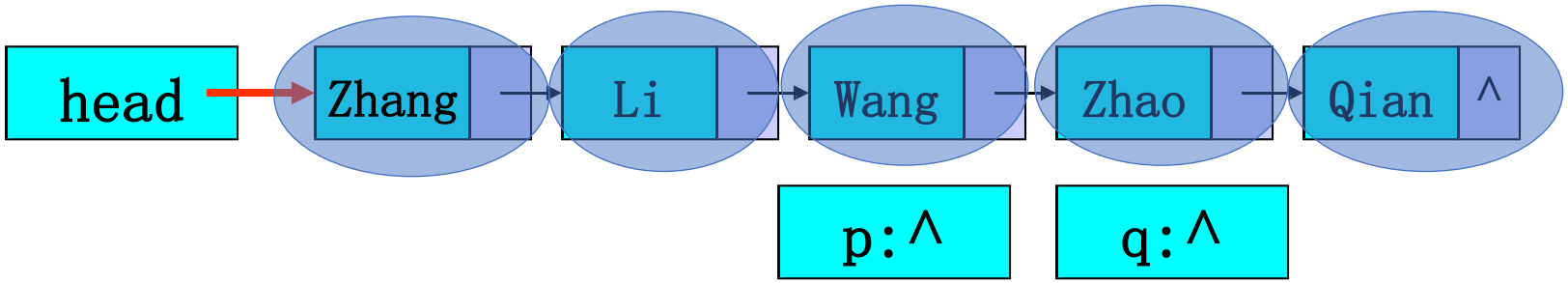
```
q = p->next;
```



while(p){…}

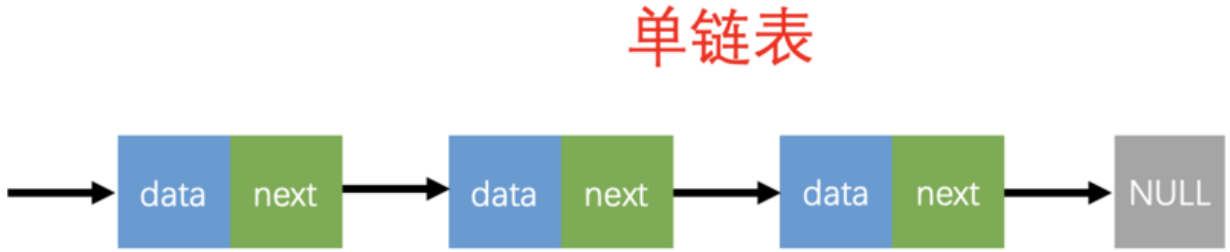
循环结束：new申请的5个空间已被释放

指针变量head/p/q自身不是动态申请空间，由操作系统回收



目录

- 链式结构的基本概念
- 单链表的基本操作
- 双向链表的基本操作
- 二叉搜索树



3.2 单链表的基本操作

- 链表的遍历
  - 由于链表的指针域中包含了后继结点的存储地址，所以只要知道该链表的头指针，即可依次对每个结点进行访问
  - 假设已创建包含一个数据域，且其类型为整型的单链表，head为表头指针

结点结构说明：

```
struct node
{
    int data;
    struct node *next;
};
```

头指针定义：struct node *head;

```
//输出已建立单链表的各结点的值
void print(struct node *head)
{
    struct node *p = head;
    while(p != NULL){
        cout << p->data << '\t';
        p = p->next;
    }
}
```

3.2 单链表的基本操作

- 统计结点个数
  - 设置工作指针从表头结点开始，每经过一个结点，计数器值加1

```
int count(struct node *head)
{
    struct node  *p = head;
    int n = 0;
    while (p != NULL){
       n++;
       p = p->next;
   }
    return(n);
}
```

3.2 单链表的基本操作

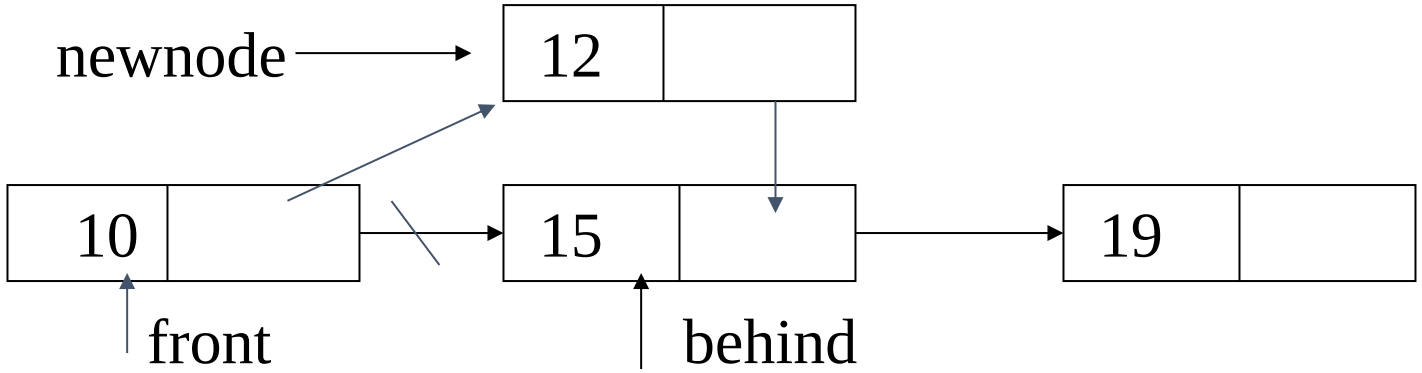
- 查找结点
  - 设置一个序号计数器j和一个工作指针p，从表头结点开始，顺着链表进行查找。仅当j==i并且p!= NULL时查找成功，否则查找不成功

```
void search(struct node *head, int i)
{   int j = 1;
    struct node *p = head;
    if(i < 0)
        cout << "illegal index\n";
    else
    {
        while(j != i && p != NULL){
            j++;
            p = p->next ;
        }
```

```
//接左边
  if(j == i && p != NULL)
     cout<<"index"<<i<<":"<<p>data;
  else
     cout<<"illegal index \n";
  }
}
```

3.2 单链表的基本操作

- 在链表中插入结点
  - 假定有一个指针behind指向链表中的某个结点，newnode指向待插入结点



  - 如果有一个指针front指向behind的前驱，则：

```
front->next = newnode;
newnode->next = behind;
```

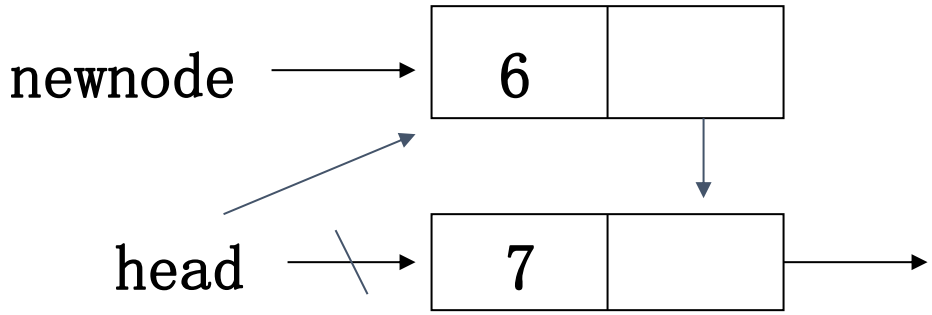
  - 如果没有behind指针，则：

```
newnode->next = front->next;
front->next = newnode;        //不可交换次序
```

3.2 单链表的基本操作

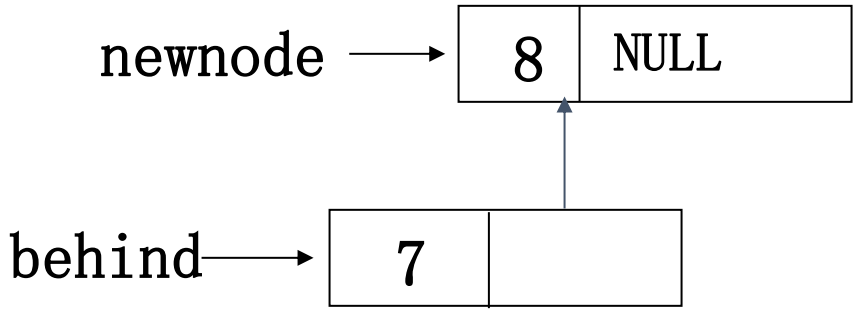
- 两种特殊情况：
  - 在表头结点之前插入：

```
newnode->next = head;

head = newnode;
```



  - 在尾结点之后插入：

```
behind->next = newnode;

newnode->next = NULL;
```



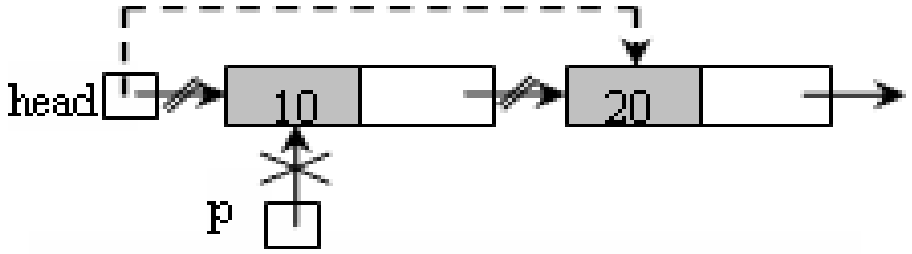
```
//实现在头结点为head的链表中插入值为x的结点（已按数据递增排序）。
struct node * insert(node *head,int x)
{  struct node *behind, *front, *newnode;
   newnode=new(nothrow) node; //未判断newnode是否为NULL，后续课程内容
   newnode->data = x;      behind = head;
   if(head == NULL) {head = newnode; newnode->next = NULL;} //空表
   else                    //非空表
   {   while(behind != NULL && x > behind->data)  //找插入位置
       {   front = behind; behind = behind->next; }
       if(behind == head)                  //插到第一个结点前
       {   newnode->next = head; head = newnode; }
       else if(behind == NULL)             //插到最后一个结点后
       {   front->next = newnode; newnode->next = NULL; }
       else                                //插到front之后,behind之前
       {   front->next = newnode; newnode->next = behind; }
   }
   return head;
}
```

3.2 单链表的基本操作

- 删除链表中的某个结点
  - 把被删除结点的后继结点的地址，赋给其前趋结点的指针域或表头指针head，无后继结点时，则赋NULL

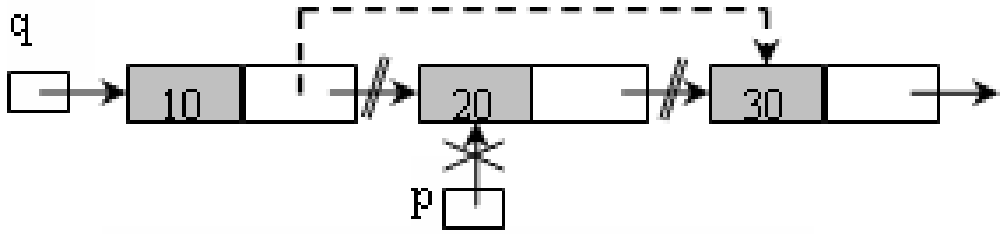
    假定p为指向要删除结点的指针，q为指向删除结点前趋的指针

    - 如删除第一个结点




```
head = p->next;
delete p;
```

    - 如删除链表的中间结点




```
q->next = p->next;
delete p;
```

```
//实现在头结点为head的链表中删除值为x的结点。
struct node *delnode(node *head,int x)
{   struct node *p,*q;  //p为工作指针,q为p的前驱
    p = head;
    if(head == NULL)          //空表
       cout<<"The list is null!\n";
    else{                     //非空表
       while(p != NULL && p->data != x)  //找删除的结点
       {   q = p;  p = p->next; }
       if(p == head)                          //删除第一个结点
       {   head = p->next;  delete p; }
       else if(p != NULL)                     //删除非表头结点
       {   q->next = p->next ;  delete p; }
       else                                   //未找到要删除的元素
           cout << x <<"does not exist in the list!\n";
     }
     return head;
}
```

3.2 单链表的基本操作

- 带表头结点的单链表
  - 表头结点（又称伪结点）位于表的最前端，本身不带数据，仅标志表头
  - 设置表头结点的目的是统一空表与非空表、表头和表中位置的操作形式，简化链表操作的实现

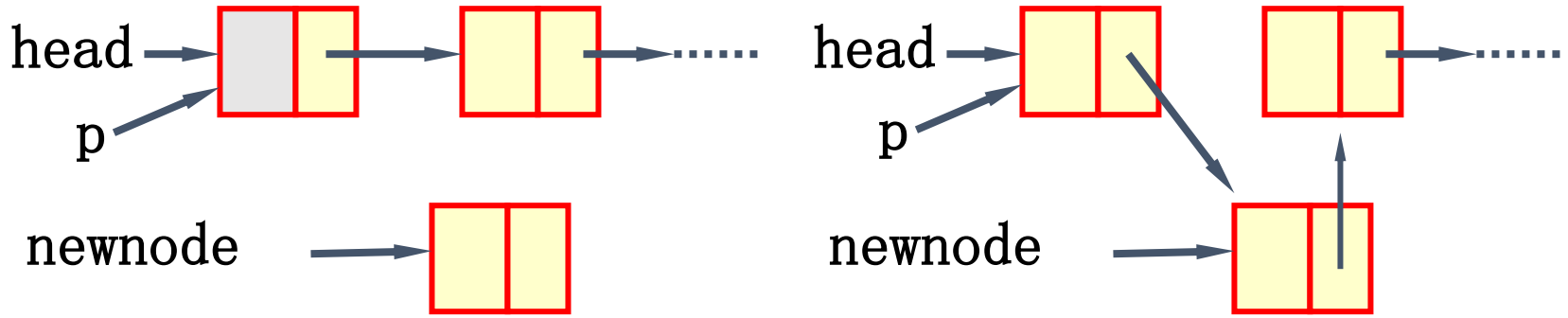
非空表

空表

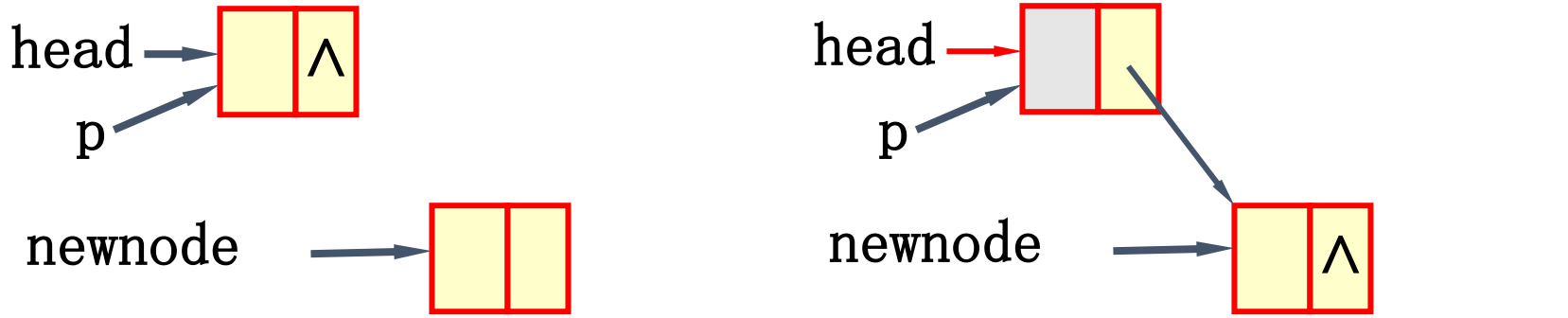
3.2 单链表的基本操作

- 例：在带表头结点的单链表最前端插入新结点

非空表



空表

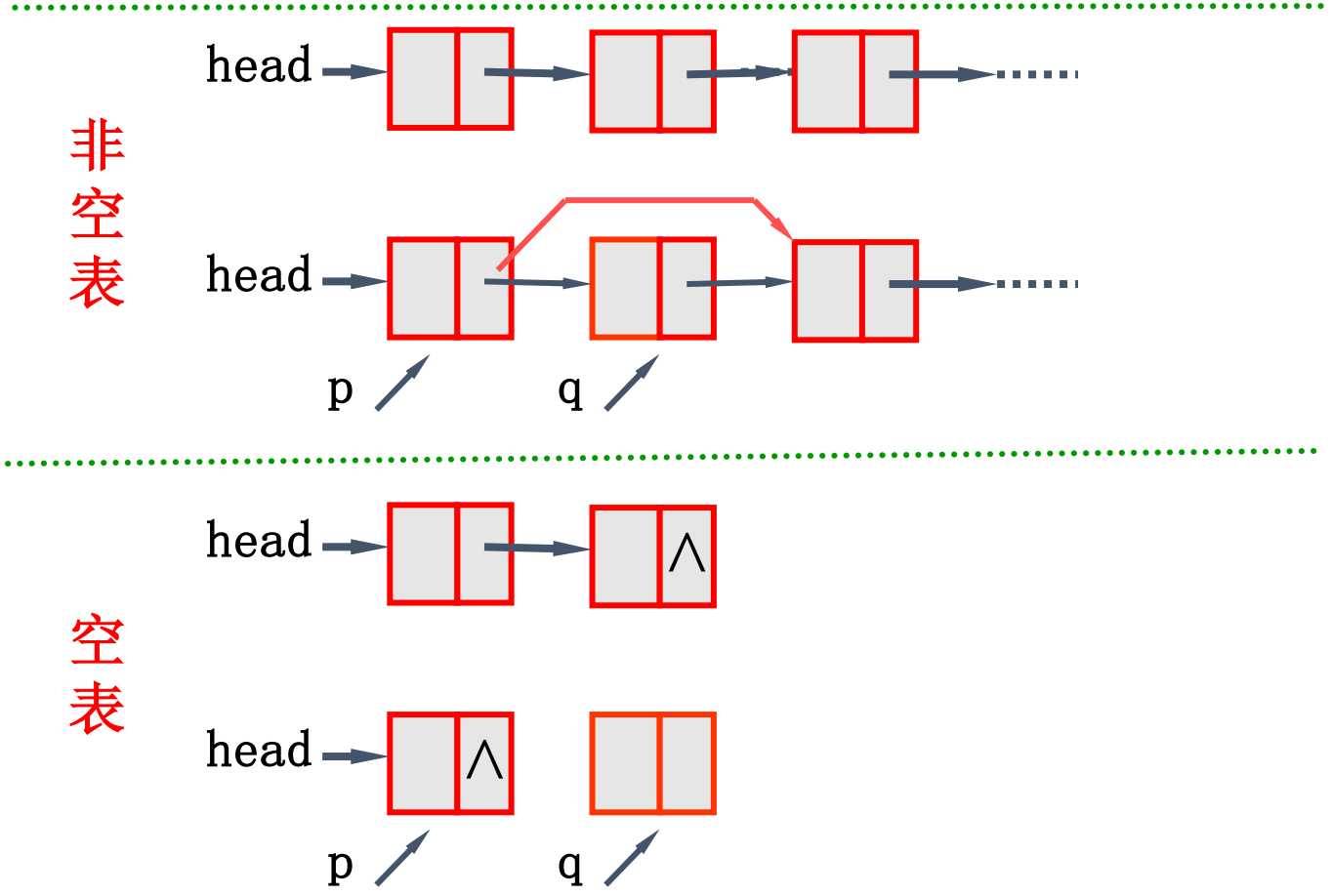


- 空表和非空表的操作是一致的

```
newnode->next = p->next;
p->next = newnode;
```

3.2 单链表的基本操作

- 例：从带表头结点的单链表中删除最前端的结点



- 即使删除后为空表，也无需修改head，空表和非空表的操作是一致的

```
q = p->next;
p->next = q->next;
delete q;
```

目录

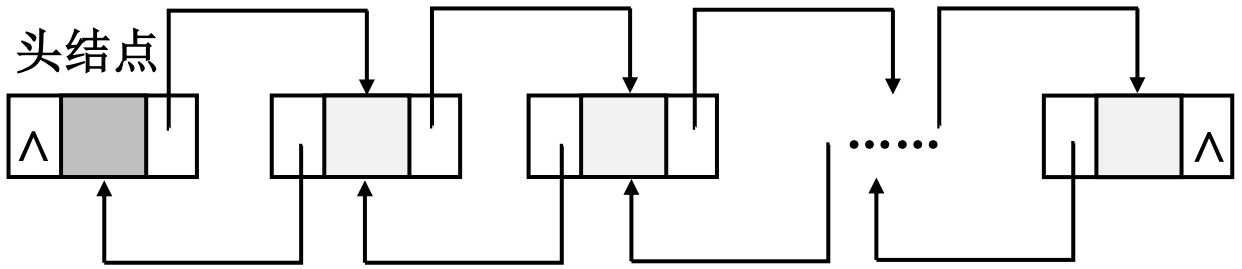
- 链式结构的基本概念
- 单链表的基本操作
- 双向链表的基本操作
- 二叉搜索树

3.3 双向链表的基本操作

- 单链表的缺陷
  - 只能从头结点开始访问链表中的数据元素，如果需要逆序访问单链表中的数据元素将极其低效
- 双向链表的引入
  - 每个数据结点中都有两个指针，分别指向直接后继和直接前驱

双向链表的定义：

```
typedef struct DbNode
{
    int data;  //结点数据
    DbNode *left; //前驱结点指针
    DbNode *right; //后继结点指针
}DbNode;
```



```
#include <iostream>
using namespace std;
typedef struct DbNode
{
    int data;//结点数据
    DbNode *left;//前驱结点指针
    DbNode *right;//后继结点指针
}DbNode;
//根据数据创建结点
DbNode *CreateNode(int data)
{
    DbNode *pnode = (DbNode *)malloc(sizeof(DbNode));
    if (pnode != NULL) //判断malloc是否成功申请到存储空间，后续课程内容
    {    pnode->data = data;
         pnode->left = pnode->right = NULL;
    }
    return pnode;
}
```

```
//创建链表头
DbNode *CreateList(int hdata)//头结点数据
{
    DbNode *pnode = (DbNode *)malloc
                    (sizeof(DbNode));
    if (pnode != NULL)
    {   pnode->data = hdata;
        pnode->left = pnode->right = NULL;
    }
    return pnode;
}
```

```
//插入新结点，总是在表尾插入
DbNode *AppendNode(DbNode *head, int data)
{
    DbNode *node = CreateNode(data);
    if (node != NULL)
    {
        DbNode *p = head, *q = NULL;

        while (p != NULL)
        {
            q = p;
            p = p->right;
        }
        if (q != NULL) q->right = node;
        node->left = q;
    }
    return head;
}
```

```
int main()
{
    DbNode *head = CreateList(0);
            //生成链表头结点，数据为0
    for (int i = 1; i < 10; i++)
            //添加9个结点，数据为1到9
    {
        head = AppendNode(head, i);
    }

    return 0;
}
```

3.3 双向链表的基本操作

- 双向链表的遍历
  - 使用right指针进行遍历，直到NULL为止

```
//双向链表的测长
int GetLength(DbNode *head)
{
    int count = 1;
    DbNode *pnode = NULL;
    if (head == NULL) //链表空
    {
        return 0;
    }
    pnode = head->right;
    //接左侧
    while (pnode != NULL)
    {
        pnode = pnode->right;
                //使用right指针遍历
        count++;
    }
    return count;
}
```

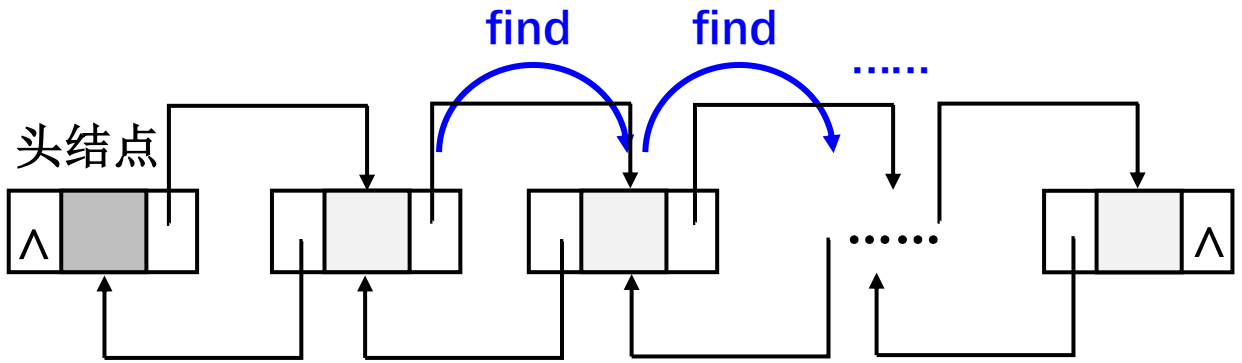
3.3 双向链表的基本操作

- 双向链表的遍历
  - 使用right指针进行遍历，直到NULL为止

```
//打印整个链表
void PrintList(DbNode *head)
{
    DbNode *pnode = NULL;
    if (head == NULL)//链表为空
    {
        return;
    }
    pnode = head;
    //接左侧
    while (pnode != NULL)
        {
            printf("%d ", pnode->data);
            pnode = pnode->right;
                    //使用right指针遍历
        }
        printf("\n");
}
```

3.3 双向链表的基本操作

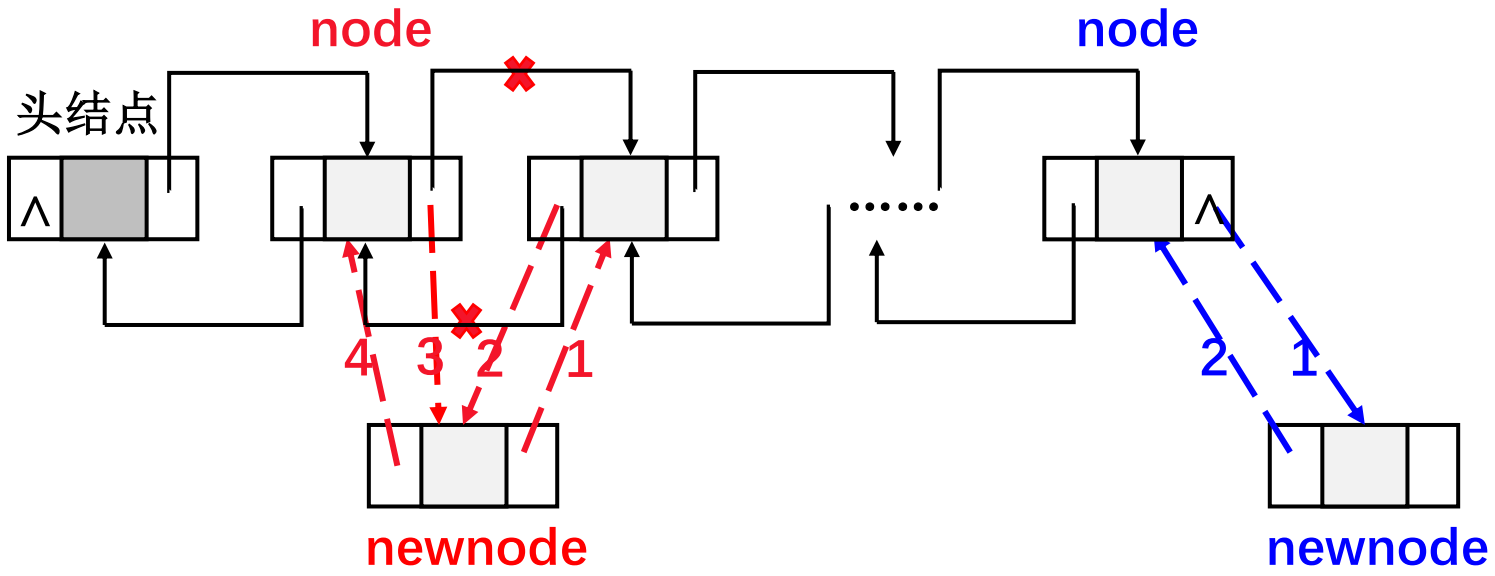
- 双向链表结点的查找
  - 使用right指针进行遍历，直到找到数据为data的结点。如果找到则返回结点，否则返回NULL



```
//查找数据为data的结点，如果找到则返回结点，否则返回NULL
DbNode *FindNode(DbNode *head, int data)
{
    DbNode *pnode = head;
    if (head == NULL) return NULL;    //链表空

    //找到数据或者到达链表末尾，退出while循环
    while (pnode != NULL && pnode->data != data)
    {
        pnode = pnode->right;          //使用right指针遍历
    }
    //没有找到数据为data的结点，返回NULL
    if (pnode == NULL)
    {
        return NULL;
    }
    return pnode;
}
```

3.3 双向链表的基本操作

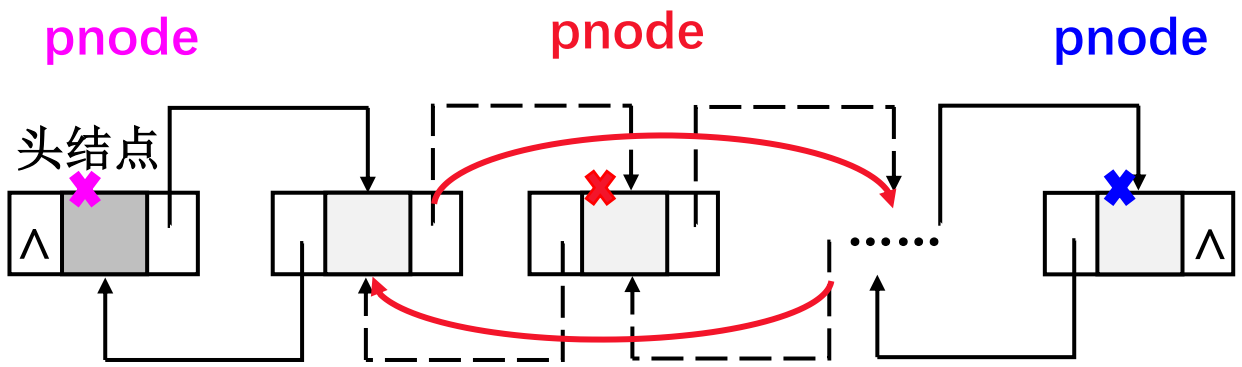
- 双向链表结点的插入
  - 插入位置在中间：原后继结点的前驱指针指向新插入结点
  - 插入位置在末尾：直接插入



```
//在node结点之后插入新结点
void InsertNode(DbNode *node, int data)
{
    DbNode *newnode = CreateNode(data);

    if (node == NULL) return;  //node为NULL时返回
    if (node->right == NULL)  //node为最后一个结点
    {
        node->right = newnode; newnode->left = node;
    }
    else   //node为中间结点
    {
        newnode->right = node->right;
        node->right->left = newnode;
        node->right = newnode;
        newnode->left = node;
    }
}
```

3.3 双向链表的基本操作

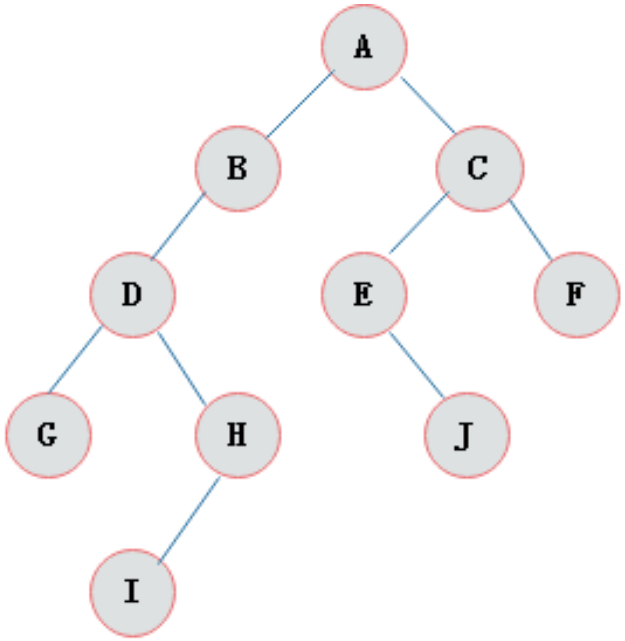
- 双向链表结点的删除
  - 删除头结点
  - 删除中间结点
  - 删除尾结点



```
//删除满足指定条件的结点，返回表头指针，删除失败（结点不存在）返回NULL
DbNode *DeleteNode(DbNode *head, int data)
{
    DbNode *pnode = FindNode(head, data); //查找结点
    if (pnode == NULL) return NULL;  //结点不存在时返回NULL
    else if (pnode->left == NULL)  //pnode为第一个结点
    {
        head = pnode->right;
        if (head != NULL)  head->left = NULL;  //链表不为空
    }
    else if (pnode->right == NULL) //node为最后一个结点
    {
        pnode->left->right = NULL;
    }
    else   //node为中间的结点
    {
        //接右侧
            pnode->left->right = pnode->right;
            pnode->right->left = pnode->left;
        }
        free(pnode);
        return head;
}
```

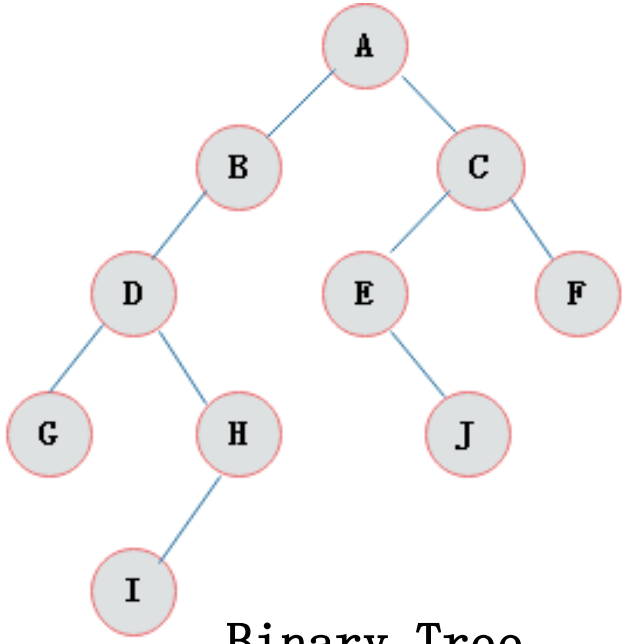
目录

- 链式结构的基本概念
- 单链表的基本操作
- 双向链表的基本操作
- 二叉搜索树



3.4 二叉搜索树（Binary Search Tree）

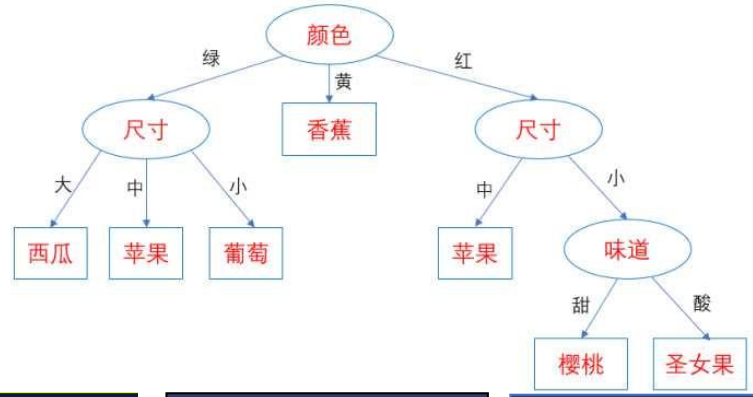
- 二叉树（Binary Tree）
  - 每个结点最多有两个子结点的树
  - 叶子结点有0个子结点，根结点或者内部结点有一个或两个子结点



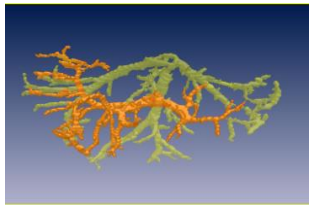
Binary Tree

如何存储？

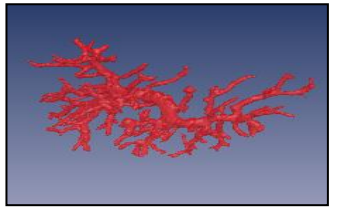
树的应用案例1：



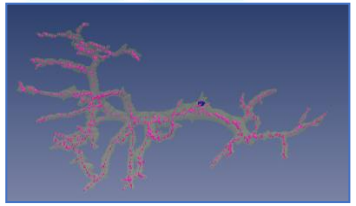
案例2：



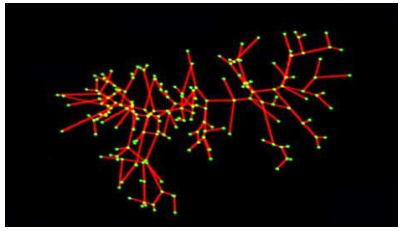
(a)：肝静脉与门静脉



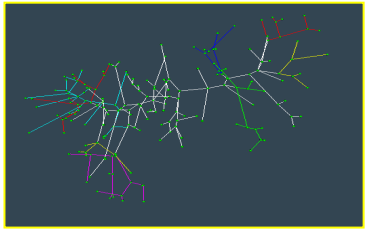
(b)：门静脉



(c)：门静脉骨架化



(d)：门静脉血管树图

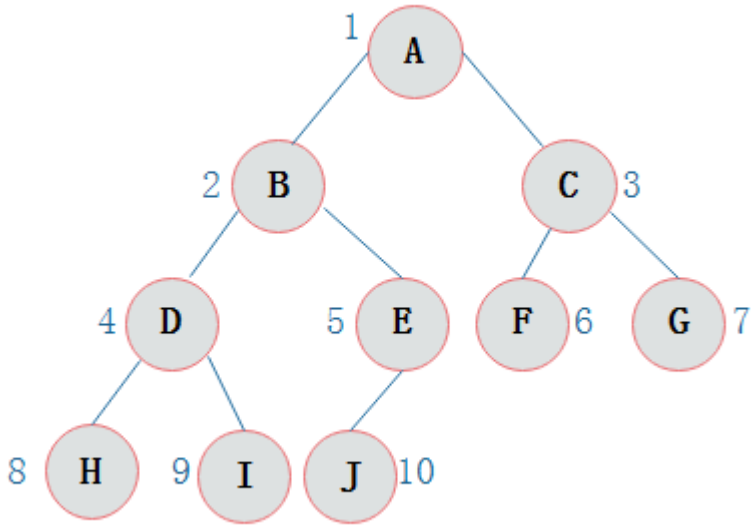


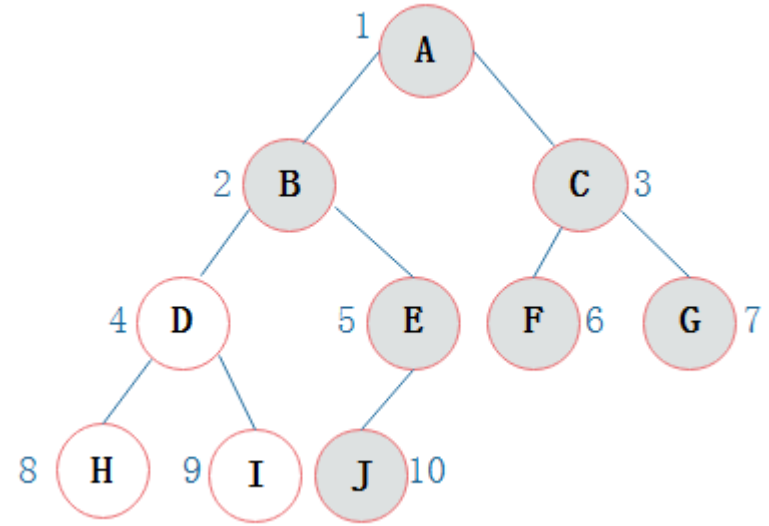
(e)：血管树分支归类

3.4 二叉搜索树（Binary Search Tree）

- 二叉树存储
  - ① 顺序存储：使用一维数组存储二叉树中的结点，并且结点的存储位置，就是数组的下标索引

缺点：一般只适合完全二叉树，否则采用顺序存储的方式十分浪费空间



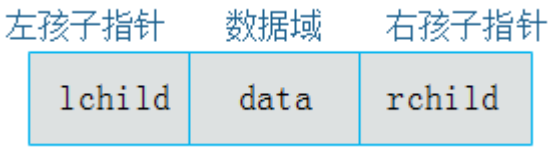


1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J

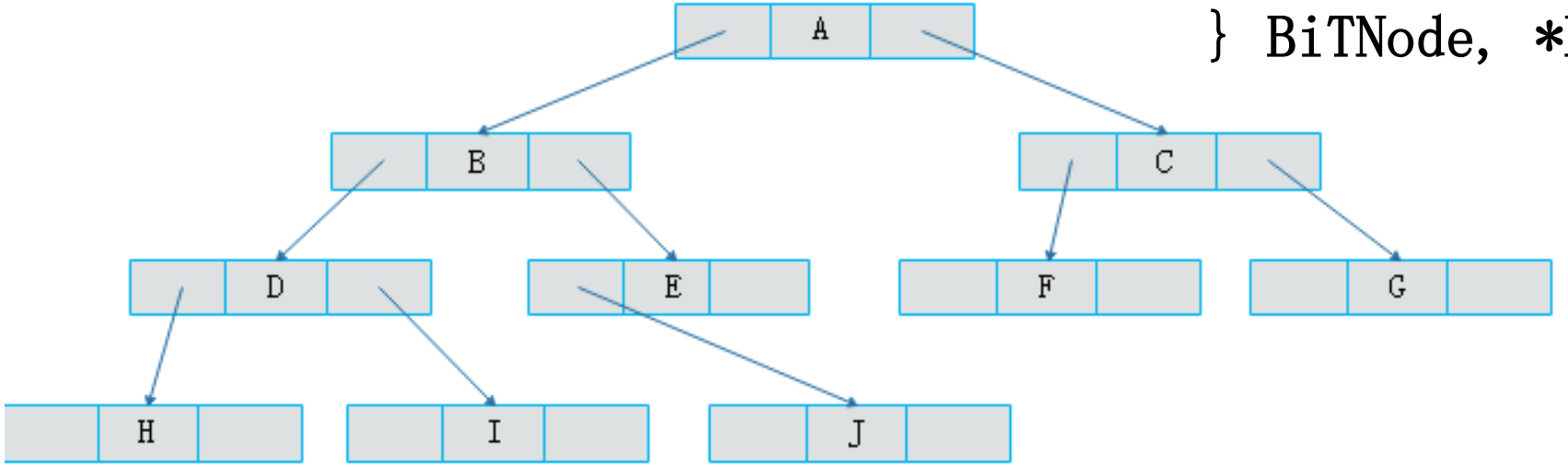
1	2	3	4	5	6	7	8	9	10
A	B	C	∧	E	F	G	∧	∧	J

3.4 二叉搜索树（Binary Search Tree）

- 二叉树存储
  - ② 链式存储：二叉树的每个结点最多有两个子结点，因此可以将结点数据结构定义为一个数据和两个指针域

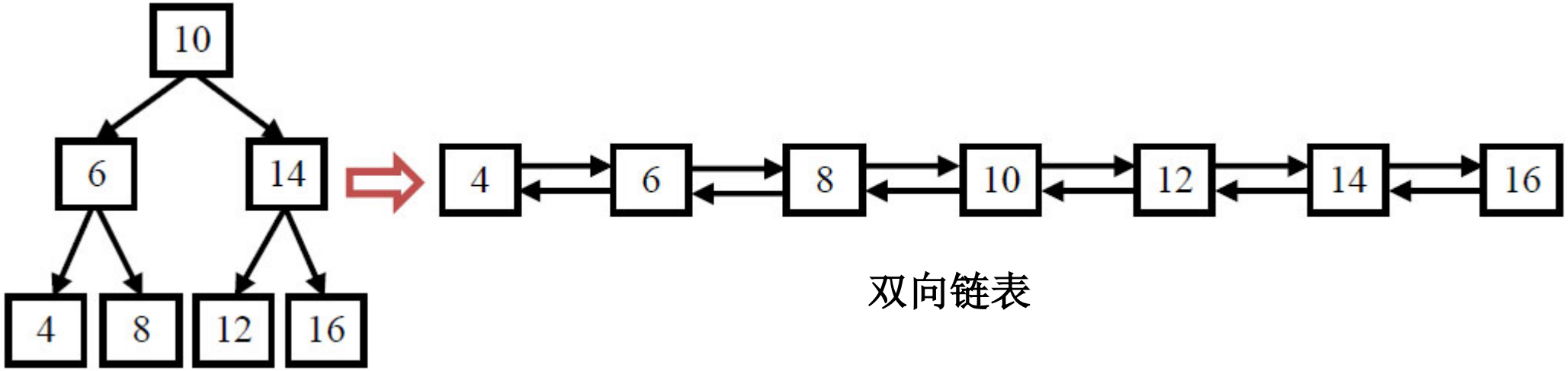



```
typedef struct BiTNode{
    TElemType data;//数据
    struct BiTNode *lchild, *rchild;
                          //左右孩子指针
} BiTNode, *BiTree;
```



3.4 二叉搜索树（Binary Search Tree）

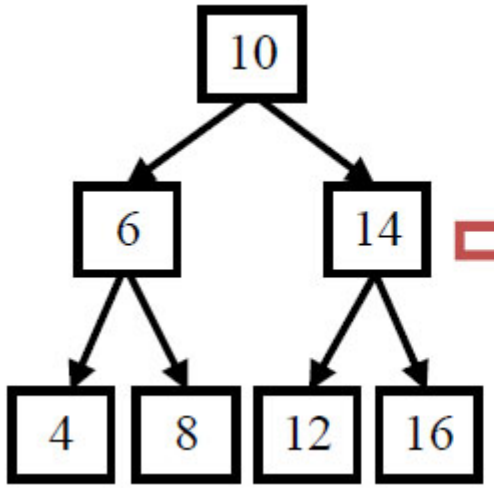
- 二叉搜索树（BST）
  - 若它的左子树非空，则左子树上所有结点的值均小于根结点的值
  - 若它的右子树非空，则右子树上所有结点的值均大于根结点的值
  - 左、右子树本身又各是一颗二叉搜索树



二叉搜索树

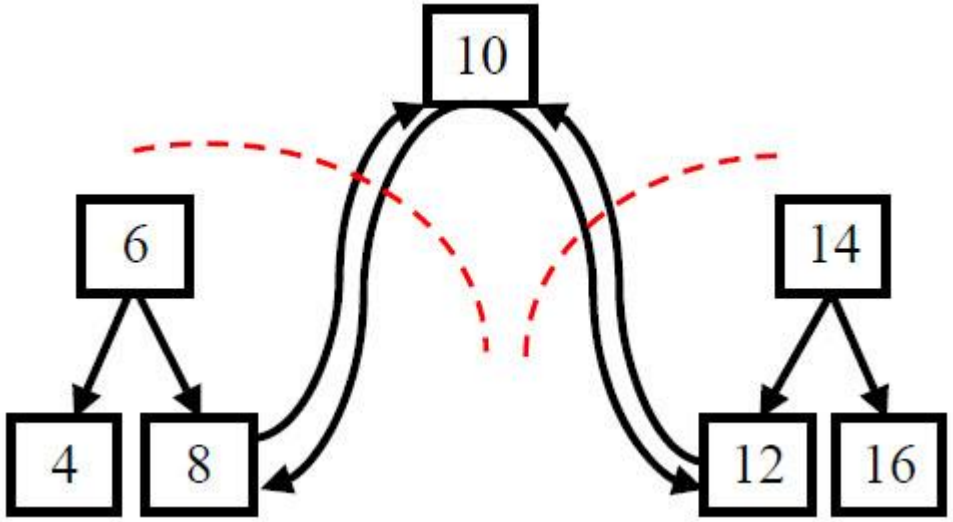
双向链表

- 输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。



二叉搜索树

双向链表



中序遍历
递归

数据结构课程，本课程了解应用即可

总结

- 链式结构的基本概念
- 单链表的基本操作（熟练掌握）
  - 遍历、统计、查找、插入、删除等
  - 带表头结点的链表
- 双向链表的基本操作（了解）
- 二叉搜索树（了解）