第6章

线性表——链式描述

本章内容

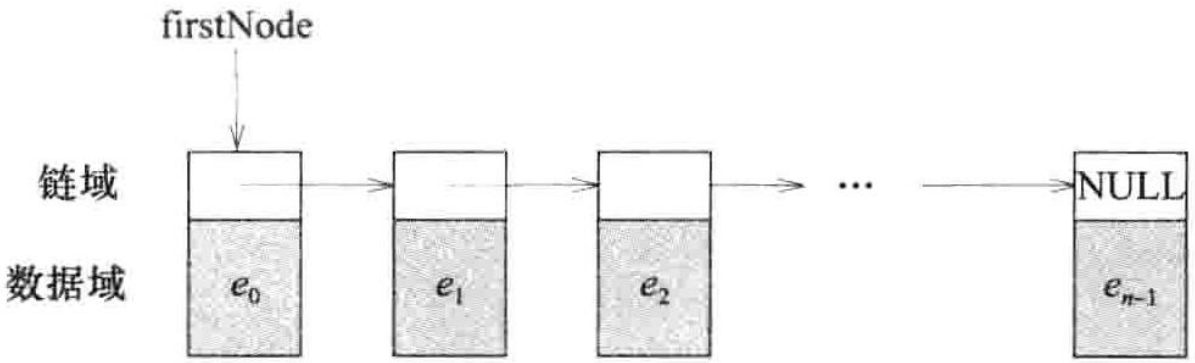
- 6.1 单向链表
- 6.2 循环链表和头结点
- 6.3 双向链表
- 6.4 链表用到的词汇表
- 6.5 应用：
  - 箱子排序(桶排序)
  - 基数排序
  - 凸包
  - 并查集

链表描述

- 数据对象实例的每个**元素**都放在单元（cell）或**节点（node）**中进行描述。
- 每个节点中都包含了**与该节点相关的其他节点**的位置信息。
  - 这种关于其他节点的位置信息被称之为**链（link）或指针（pointer）**。

6.1 单向链表

- 线性表：$L = (e_0, e_1, \ldots, e_{n-1})$
  - 每个**元素e_i**都放在**单独的节点**中加以描述。
  - 每个节点都包含一个链域，其值是线性表中下一个元素的地址。
  - 元素e_i的节点链接着e_{i+1}的节点，最后一个元素e_{n-1}没有下一个元素，元素e_{n-1}的节点无节点链接，它的链接域为NULL
  - 这种结构也被称作链（chain）.



结构‘chainNode’

```
template <class T>
struct chainNode {
    //数据成员
     T element;
     chainNode<T> *next;
    //方法
    chainNode(){}
    chainNode(const T& element)
          {this->element = element;}
    chainNode(const T& element, chainNode<T>* next )
          {this->element = element;
           this->next = next;}
};
```

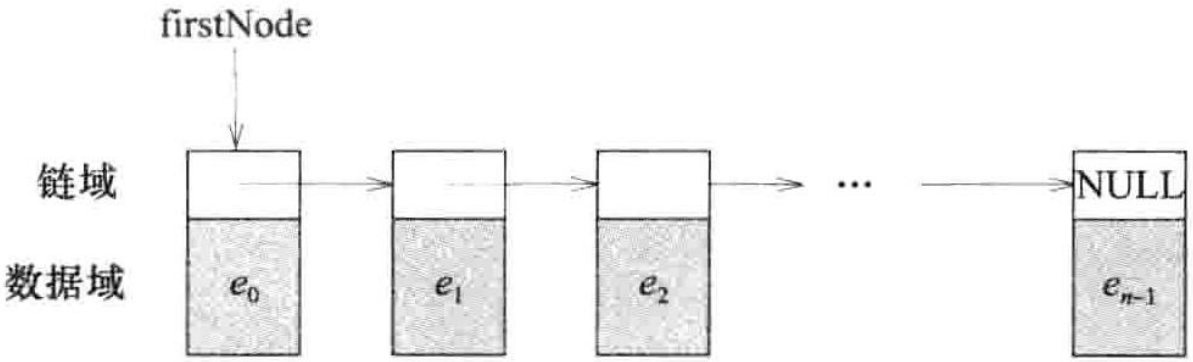
类chain（1/2）

```
template <class T>
class chain ： public linearList<T>
{
  public:
     //构造函数、复制构造函数、析构函数
    chain(int initialCapacity = 10);
    chain(const chain <T>&);
    ~chain ();

    //ADT方法
    bool empty() const {return listSize = = 0;}
    int size() const {return listSize;}
    T& get(int theIndex) const;
    int indexOf(const T& theElement) const;
    void erase(int theIndex);
    void insert(int theIndex, const T& theElement);
    void output(ostream& out) const;
```

类chain（2/2）

```
protected:
    void checkIndex(int theIndex) const;
            // 若索引theIndex无效，则抛出异常

    chainNode <T> *firstNode;
                        //指向链表中第一个节点的指针
    int  listSize;          //线性表的元素个数
};
```

类‘chain’



- 链表空时： firstNode=NULL
- listSize：线性表的元素个数 = 链表中节点的个数

chain的构造函数

- Chain的构造函数形式必须与arrayList相容
  - chain(int initialCapacity = 10)；
  - 初始化建立一个空链表
- 程序代码，程序6-3

```
template<class T>
chain<T>::chain(int initialCapacity)
{// 构造函数
   if (initialCapacity < 1)
   {ostringstream s;
    s << "Initial capacity = " << initialCapacity << " Must be > 0";
    throw illegalParameterValue(s.str());
   }
   firstNode = NULL;
   listSize = 0;
}
```

chain的复制构造函数(1/2)

```
template<class T>
chain<T>::chain (const chain<T>& theList)
{   //复制构造函数
    listSize = theList.listSize;

  //链表theList为空
   if(listSize ==0)
    {firstNode=NULL;
     return;
    }
```

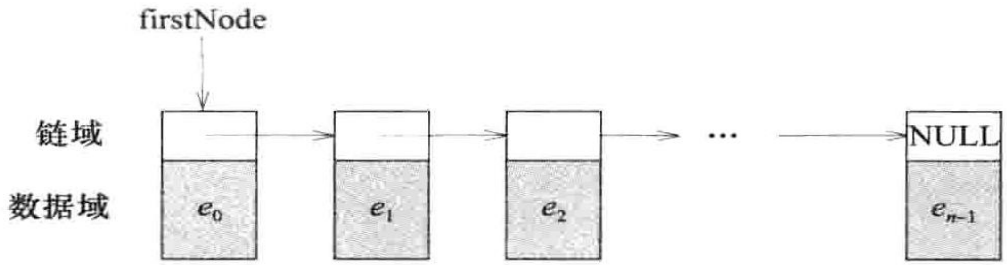
chain的复制构造函数(2/2)

```
//链表theList不为空
chainNode<T>*  sourceNode=theList.firstNode;
                          //要复制的theList中的节点
firstNode=new chainNode<T>(sourceNode->element);
                          //复制theList中的首元素
sourceNode= sourceNode->next;
chainNode<T>*  targetNode=firstNode;
                  //当前链表*this的最后一个节点
while(sourceNode!=NULL)
   {//复制剩余元素
   targetNode->next=new chainNode<T>
                                   (sourceNode->element);
   targetNode=targetNode->next;
   sourceNode= sourceNode->next;
   }
   targetNode->next=NULL;
}
```

- 时间复杂性：Θ(theList.listSize)

析构函数‘~Chain’

```
template <class T>
chain<T>::~Chain()
{ //链表的析构函数，删除链表中的所有节点
  while (firstNode)
  { //删除首节点
   chainNode<T>* next = firstNode->next;
    delete firstNode;
    firstNode = next;
    }
}
```

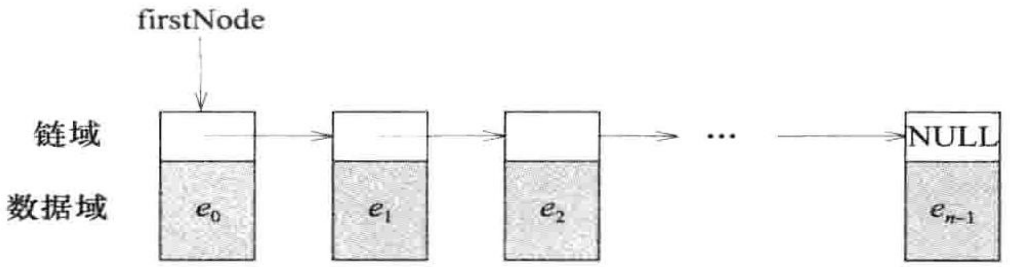


- 时间复杂性：Θ(listSize)

方法‘get’

```
template <class T>
T& chain<T>::get(int theIndex) const
{  // 返回索引为theIndex的元素
   //若此元素不存在，则抛出异常
    checkIndex(theIndex);
   //移动到所需要的节点
   chainNode<T> *currentNode = firstNode;
    for (int i=0;i<theIndex;i++)
       currentNode = currentNode->next; //移向下一个节点

    return currentNode->element;
}
```

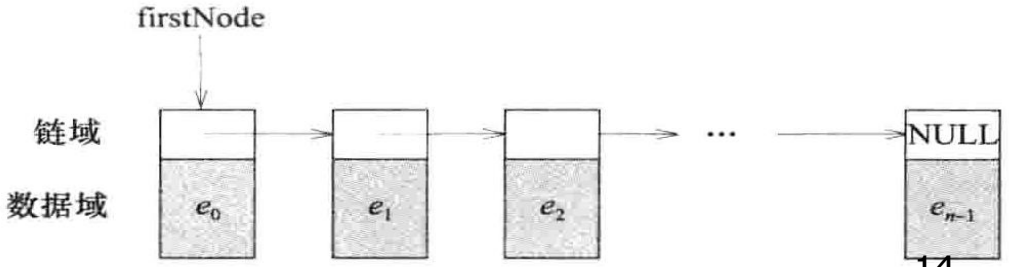
- 时间复杂性：O(theIndex)



方法‘indexOf’

```
template<class T>
int chain<T>::indexOf(const T& theElement) const
{ //返回元素theElement首次出现时的索引,如果theElement不存在, 则返回-1
  //查找元素
  chainNode<T> *currentNode = firstNode;
   int index = 0; // currentNode的索引
   while (currentNode!=NULL &&
             currentNode->element != theElement)
         { currentNode = currentNode->next;
            index++ ;
           }
   if (currentNode==NULL) return -1;
   return  index;
}
```

- 时间复杂性：O(listSize)



方法‘erase’

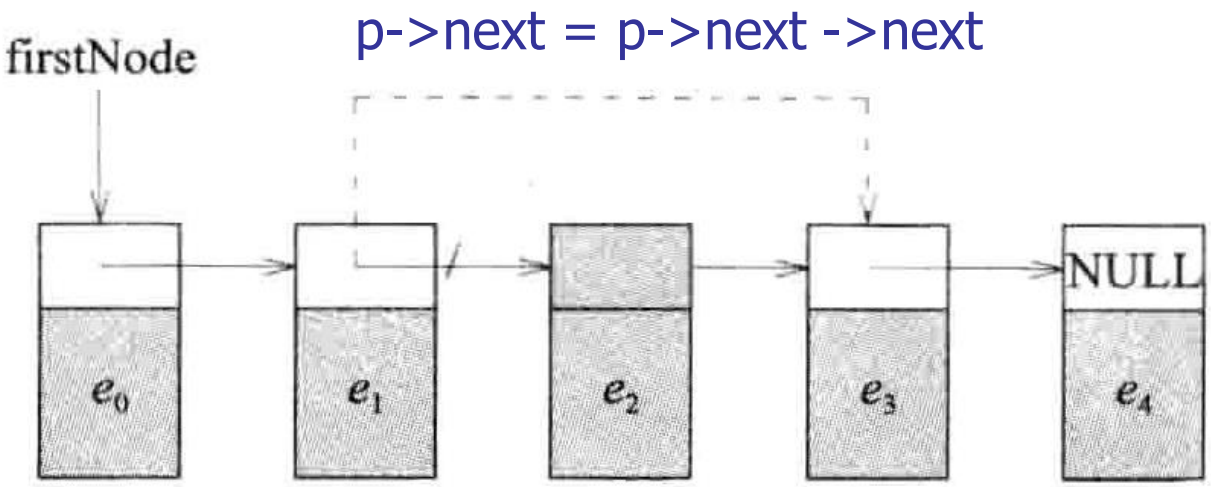
删除第2个元素步骤:

1.定位第1和第2个节点

2.将第1个节点的指针指向第3个节点

3.释放第2个节点空间

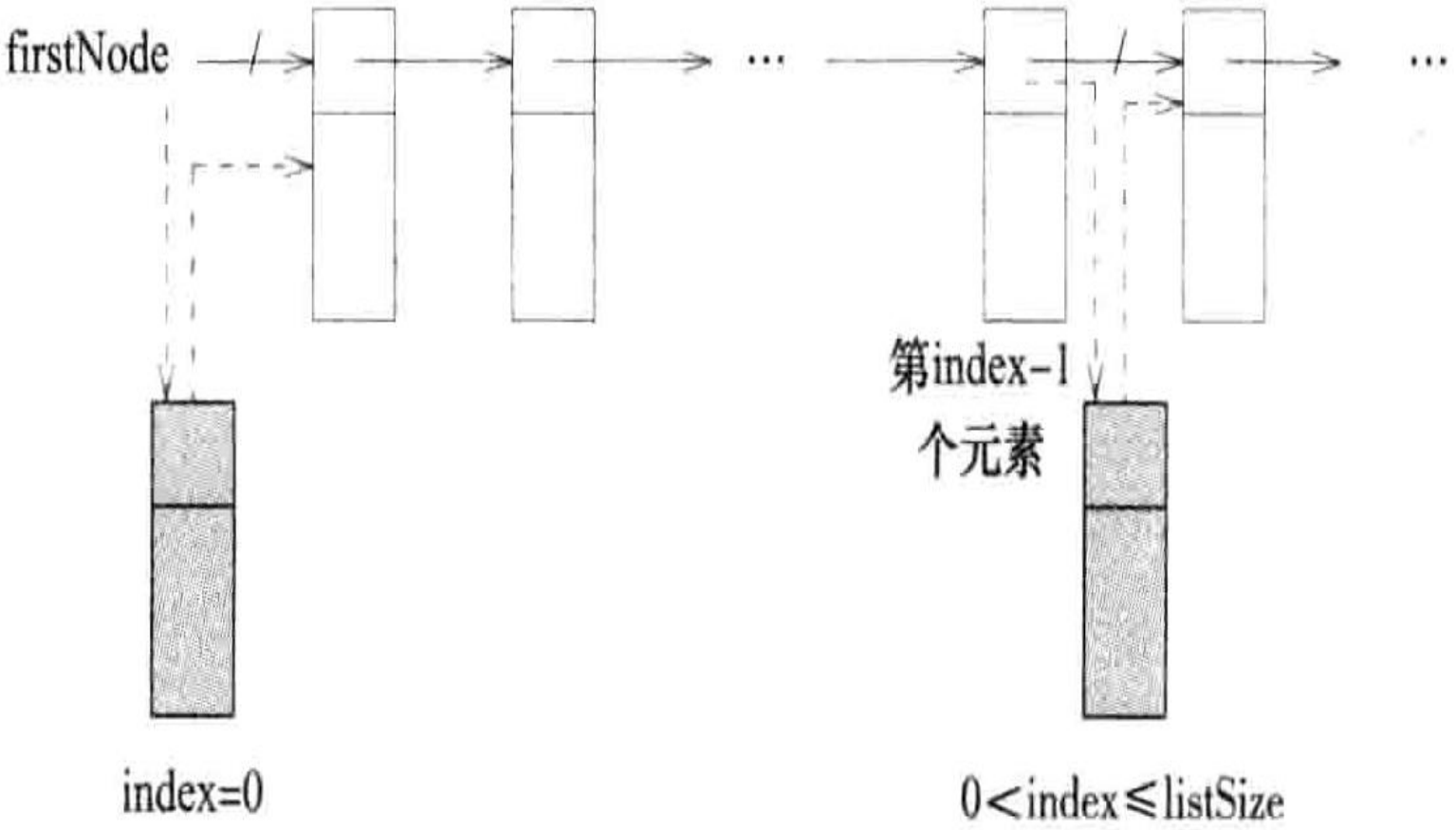
4.listSize减1



```
template<class T>
void chain<T>::erase(int theIndex)
{//删除表中索引为theIndex的元素
  //如果元素不存在，则抛出异常.
  checkIndex(theIndex)；
  //索引有效，需要找要删除的元素节点
 chainNode<T> *deleteNode = firstNode;
 if (theIndex = = 0)
     {//删除表中首节点
     deleteNode = firstNode;
     firstNode = firstNode->next;
     }
 else { //用指针p指向第theIndex-1个节点
         chainNode<T> *p = firstNode;
         for (int i = 0; i<theIndex-1; i++)
                        p=p->next;
         deleteNode = p->next;
          p->next = p->next ->next;//删除deleteNode指向的节点
        }
 listSize--;
 delete deleteNode;
}
```

时间复杂性：O(theIndex)

方法‘insert’



```
template<class T>
void chain<T>::insert(int theIndex, const T& theElement)
{   //在索引theIndex处插入元素theElement;
    //如果theIndex无效，则引发异常
    if (theIndex<0 || theIndex>listSize) {……}

  //在链表表头插入
  if (theIndex=0)
    firstNode=new chainNode<T>(theElement, firstNode);

 else
   {//寻找新元素的前驱(第theIndex-1个元素 )
    chainNode<T> *p = firstNode;
    for (int i = 0; i < theIndex-1; i++)
            p = p->next; //将p移动至第theIndex-1个元素

    //在p之后插入
   p->next=new chainNode<T>(theElement, p->next);
  }
 listSize++;
}
```

- 时间复杂性： O(theIndex)

方法‘Output’

```
template<class T>
 void chain<T>::Output(ostream& out) const
 {//将链表元素送至输出流 .
 for (chainNode<T>* currentNode = firstNode;
          currentNode!=NULL;
          currentNode = currentNode->next)
    out << currentNode->element << " ";
 }

 //重载<<
 template <class T>
 ostream& operator<<(ostream& out, const chain<T>& x)
 {x.Output(out); return out;}
```

- 时间复杂性：Θ(listSize)

链表的成员类iterator

- 为chain定义一个向前迭代器iterator

- 具备操作符：*、->、++、 ==、！=
  - *操作符，获得迭代器所指的数据
  - ->操作符，获得迭代器所指数据的地址
  - 前++、后++：迭代器移到下(后)一个元素
  - ==、！=：判断是否相等

链表的成员类iterator

```
class iterator
{
  public:
  //C++的向前迭代器所需要的typedef 语句省略
  //构造函数
   iterator(chainNode<T>* theNode = NULL)
                          {node = theNode;}
  //解引用操作符
   T& operator*() const {return node->element;}
   T* operator->() const {return &node->element;}
  //迭代器加法操作
  iterator & operator++()   //前++
    {node=node->next; return *this;}
  iterator operator++(int)   //后++
    {iterator old=*this; node=node->next; return old;}
```

链表的成员类iterator

```
//测试是否相等
bool operator!=(const iterator right) const
    {return node!= right.node;}
bool operator==(const iterator right) const
    {return node== right.node;}

protected:
    chainNode<T>* node;//指向表节点的指针
}
```

链表的成员类iterator

- 为类chain增加：
  - begin():返回指向线性表首节点的指针
  - end()：返回指向线性表尾节点的下一个位置

- iterator begin(){return iterator (firstNode);}
- iterator end(){return iterator (NULL);}

扩充类linearList

- 在抽象数据类型linearList增加操作：
  - clear():

    删除表中的所有元素;
  - push_back(theElement):

    在表尾插入元素theElement;

扩充类linearList

```
template <class T>
class extendedLinearList   :  linearList<T>
{
  public:
      virtual  ~ extendedLinearList() {};
      virtual void clear() const = 0;
                  //删除表中的所有元素;
      virtual  push_back(const T& theElement) = 0;
                  //在表尾插入元素theElement;
}
```

类extendedChain

- 类extendedChain:
  - 作为类extendedLinearList的链表描述
- 类extendedChain
  - 从类chain派生而来

方法 'clear'

```
template <class T>
void extendedChain<T>:: clear()
{//删除表中的所有元素
   while (firstNode!=NULL)
       {//删除节点firstNode
        chainNode<T> *nextNode = firstNode->next;
         delete firstNode;
        firstNode = nextNode;
       }
listSize=0;
}
```

方法 'push_back'

```
template < class T >
void extendedChain<T>:: push_back(const T& theElement)
{ //在链表尾部添加元素theElement .
    chainNode<T> *newNode
                    = new chainNode<T>(theElement ,NULL);
    if (firstNode==NULL) {//链表空
            firstNode = lastNode = newNode;
    else  //新元素节点附加到lastNode 指向的节点
           {lastNode ->next = newNode;
           lastNode  = newNode;}
  listSize++;
}
```

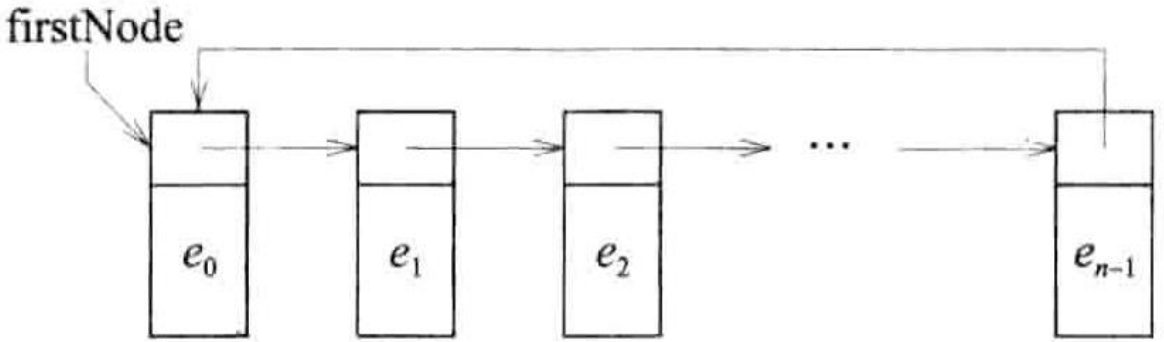
- 时间复杂性： Θ(1)

6.2 循环链表和头结点

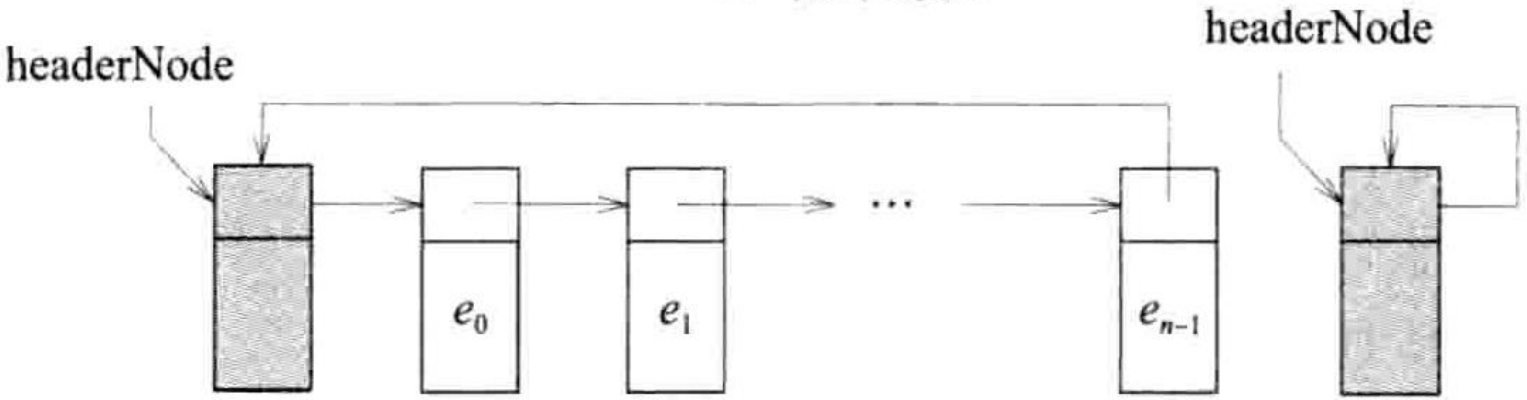
- 采取下面的一条或两条措施，使用链表的应用代码可以更简洁、更高效：

1. 把线性表描述成一个单向循环链表（singly linked circular list），或简称循环链表（circular list），而不是一个单向链表；
2. 在链表的前部增加一个附加的节点，称之为头节点（ header node）。

循环单链表



a）循环链表



b）有头节点的循环链表　　c）空表

带有头结点的循环链表构造函数

```
template < class T >
circularListwithHeader<T>:: circularListwithHeader()
{ //构造函数
    headerNode = new chainNode<T>();
    headerNode ->next = headerNode;
    listSize=0;
}
```

带头结点的循环链表中的搜索

```
template<class T>
int circularListwithHeader<T>::indexOf(const T& theElement)
    const
{//返回元素theElement首次出现时的索引,如果theElement不存在,则返回-1
//将元素theElement放入头节点
 headerNode->element= theElement;

 //在链表中搜索元素
  chainNode<T> *currentNode = headerNode->next;
   int index = 0; // currentNode的索引
   while (currentNode->element != theElement)
         {
           currentNode = currentNode->next;
           index++ ;
          }

   //确定元素theElement是否存在
   if (currentNode==headerNode) return -1;
   else  return  index;
}
```

- 时间复杂性：O(listSize)

与“单链表中的搜索”比较

```
template<class T>
int chain<T>::indexOf(const T& theElement) const
{ //返回元素theElement首次出现时的索引,如果theElement不存在, 则返回-1
   chainNode<T> *currentNode = firstNode;
   int index = 0; // currentNode的索引

    //查找元素
    while (currentNode!=NULL &&
           currentNode->element != theElement)
       { currentNode = currentNode->next;
         index++ ;
        }

   //确定元素theElement是否存在
   if (currentNode==NULL) return -1;
   return  index;
}
```

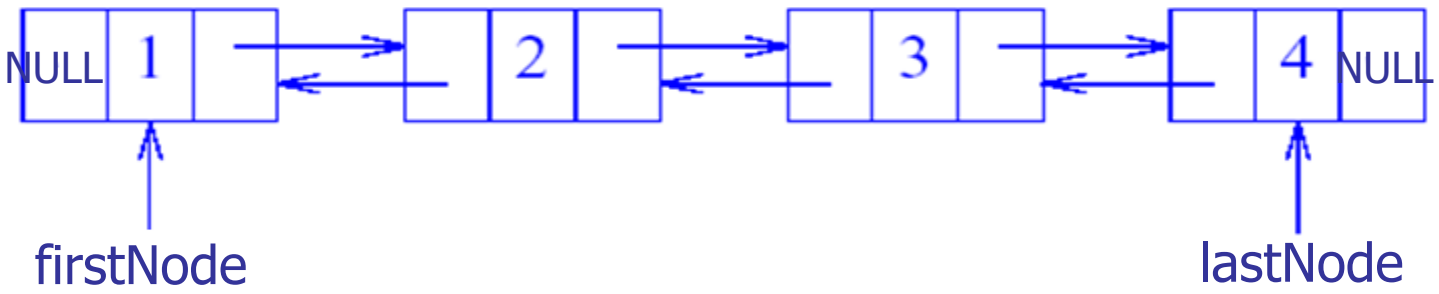
- 时间复杂性：O(listSize)

6.3 双向链表

- 为了能快速找到一个节点的前驱，可以在单链表中的节点中增加一个指针域指向它的前驱.
- 每个节点：

previous	element	next

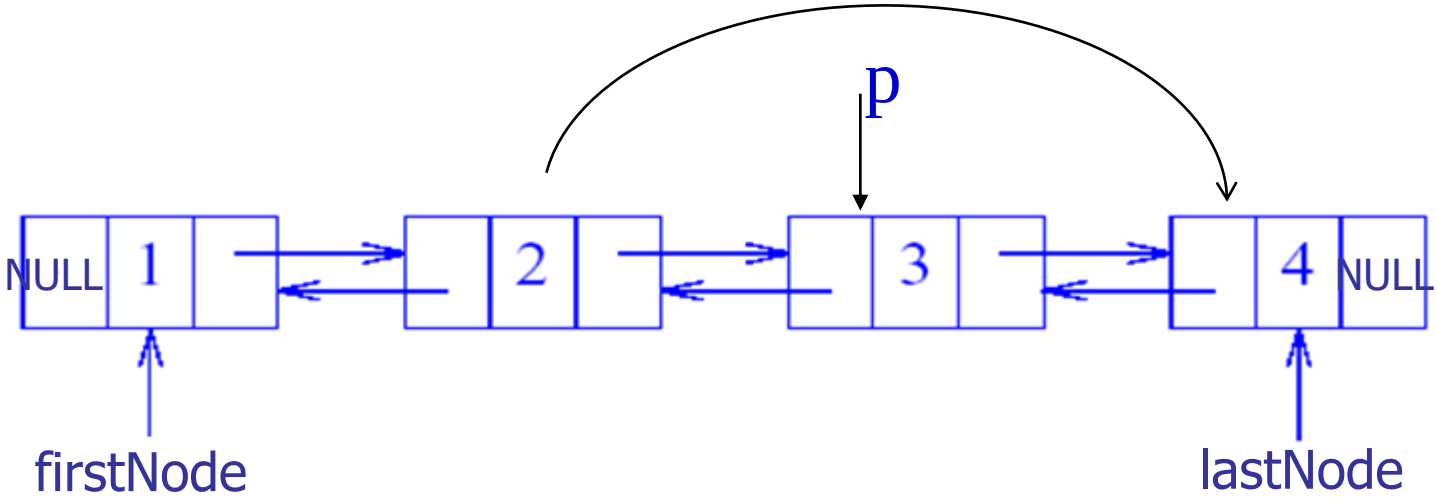
- L=(1，2，3，4)的双向链表描述



删除元素

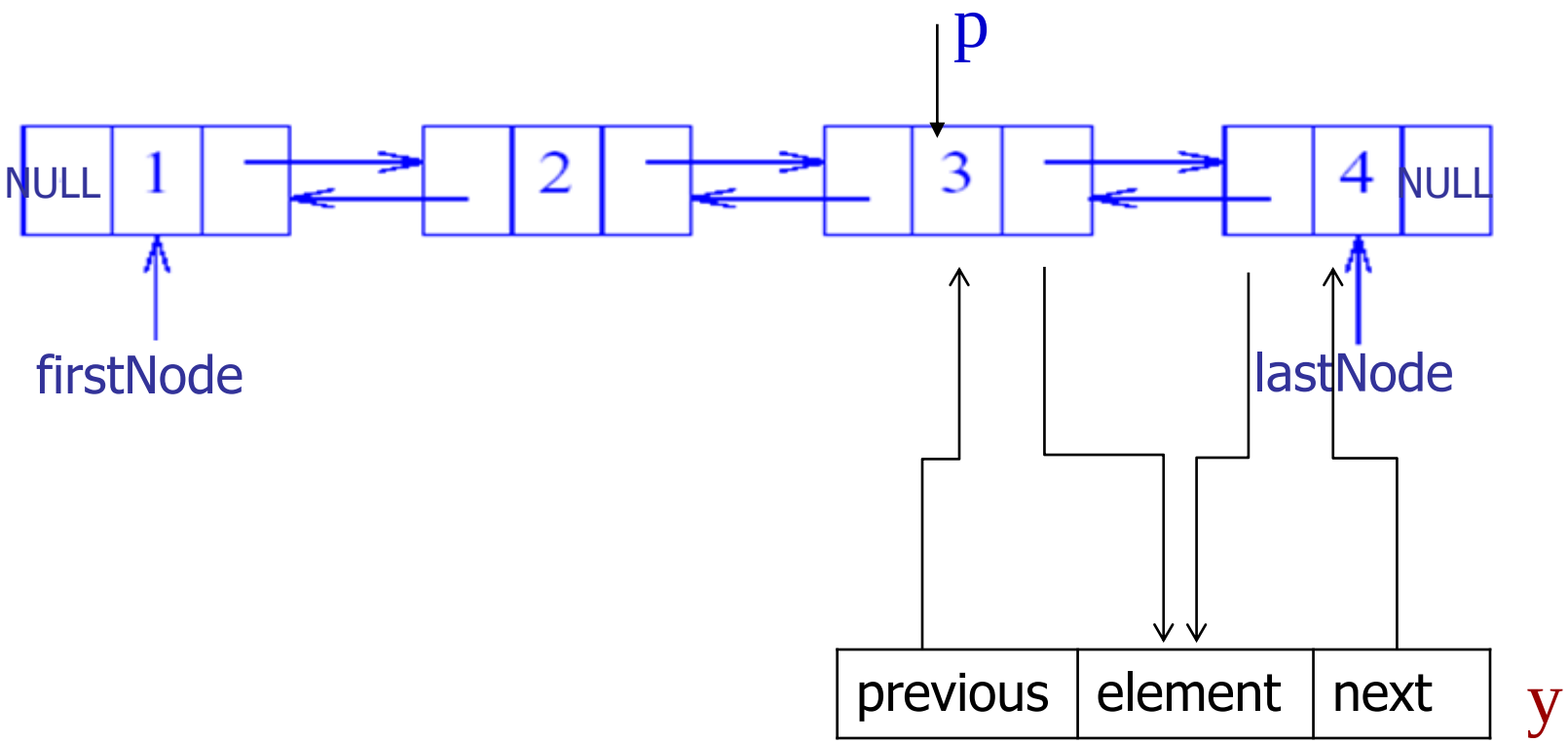
- L=(1，2，3，4)的双向链表描述

p->next->previous = p->previous;
p->previous> next- = p-> next;

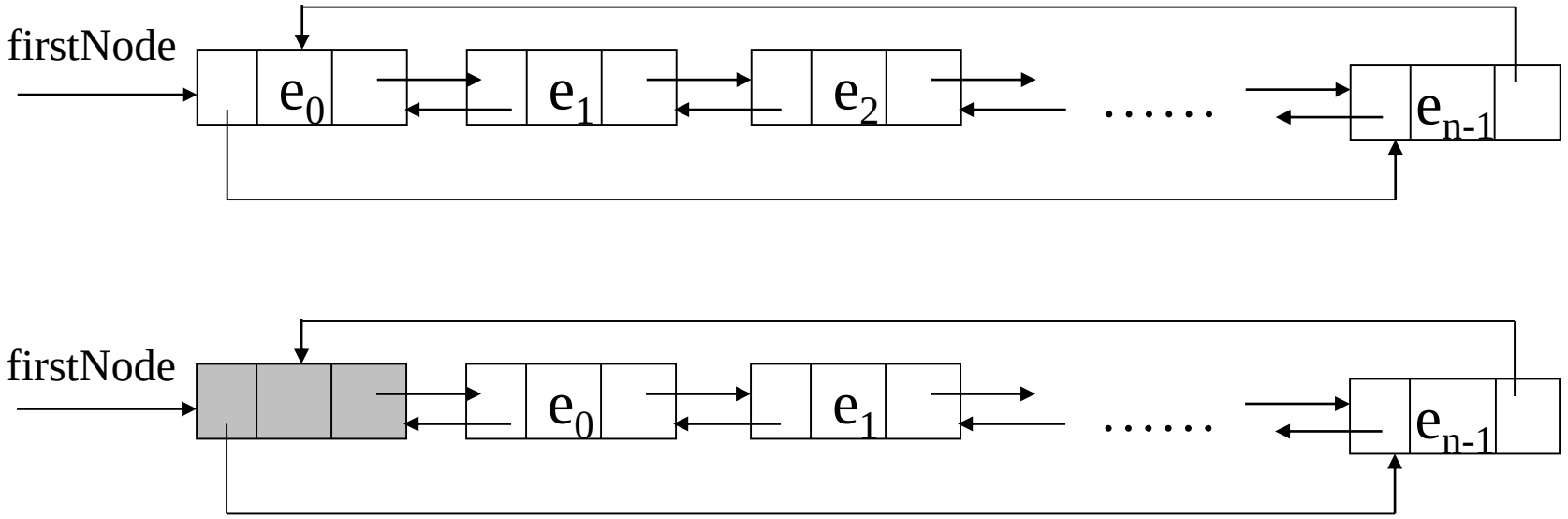


插入元素

```
y->next = p-> next;
p-> next = y;
y->previous = p;
y-> next ->previous = y;
```

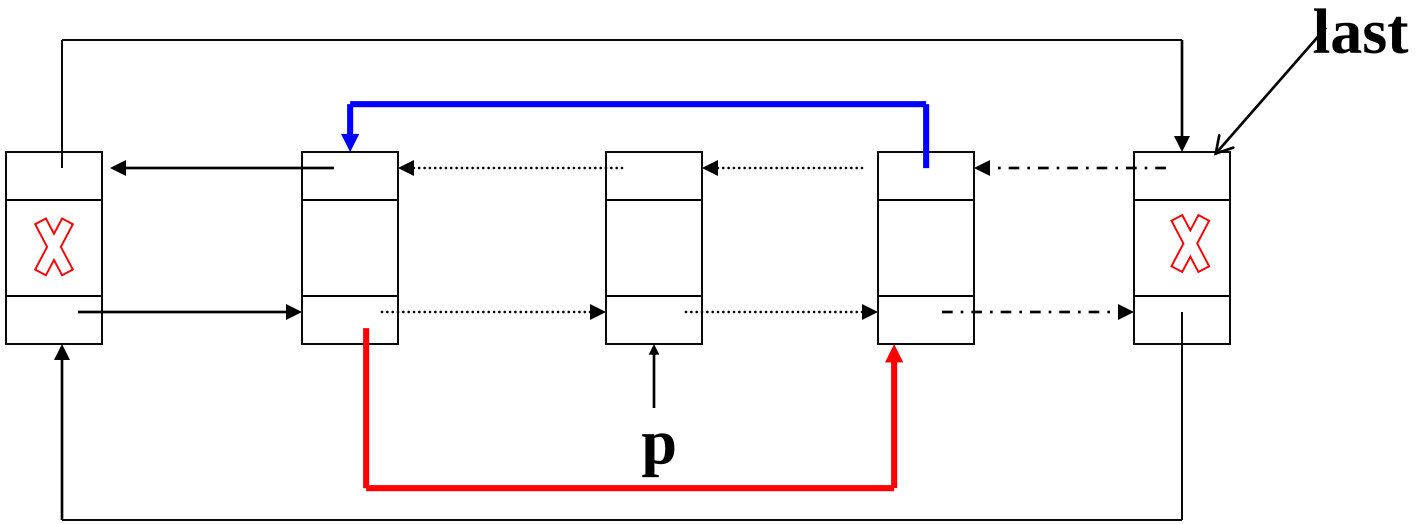


双向循环链表



- 双向链表中的插入、删除操作实现？

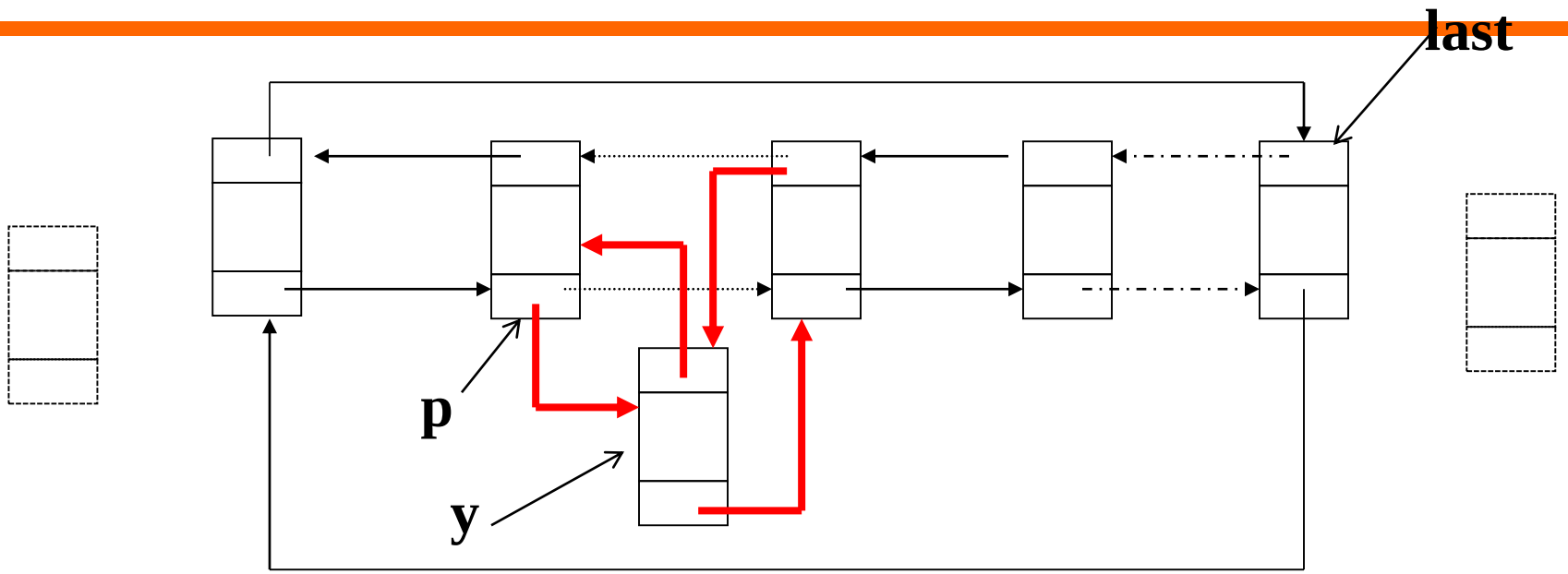
双向循环链表的结点删除



p->next->previous = p->previous;

p->previous> next- = p-> next;

双向循环链表的结点插入




```
y->next = p-> next;
p-> next = y;
y->previous = p;
y-> next ->previous = y;
```

6.4 链表用到的词汇表

- 1.单向链表(chain or single linked list)
- 2.单向循环链表(single linked circular list)
- 3. 头节点(header node)
- 4.双向链表(doubly linked list)
- 5.双向循环链表(circular doubly linked list)

链表描述优缺点

- 优点
  - 插入、删除操作的时间复杂性不依赖于元素大小
  - 插入、删除元素不需要移动表内数据
  - 空间不需要事先申请
- 缺点
  - 每个节点中需要额外的空间用于保存链接指针
  - 不能随机存取表中的任一节点
  - 如果数据有序，可使用折半搜索？

6.5 应用

- 6.5.1 箱子排序(Bin Sort)
- 6.5.2 基数排序(Radix Sort)
- 6.5.3 凸包(Convex Hull)
- 6.5.4 并查集(Union-Find)

6.5.1 箱子排序

- 对0 ~ 100范围内的分数，如果采用第2章中所给出的任一种排序算法进行排序，所需要花费的时间均为$O(n^2)$.
- 一种更快的排序方法为箱子排序（bin sort）。
- **箱子排序：**
  - 1. 元素被分配箱子之中，相同分数的放在同一个箱子中
  - 2. 按箱子的序号收集箱子里的元素。

链表元素结构定义1

```
struct  studentRecord
{
  int score;
  string* name;
  int operator !=(const studentRecord& x) const
       {return (score!= x.score);}
};


ostream& operator<<(ostream& out,
                                const studentRecord & x)
{out << x.score << '  ' <<*x.name<<endl; return out;}
```

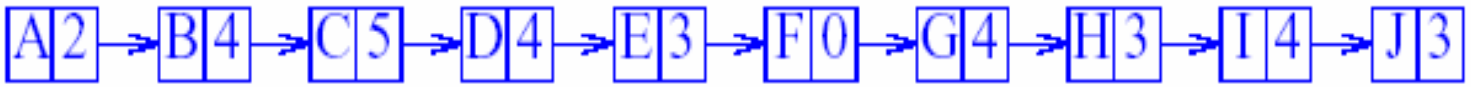
链表元素结构定义2

```
struct  studentRecord
{
   int score;
   string* name;

   //从studentRecord到int的类型转换
   operator int( ) const  {return  score;}
};


ostream& operator<<(ostream& out,
                                const studentRecord & x)
{out << x.score << '  ' <<*x.name<<endl; return out;}
```

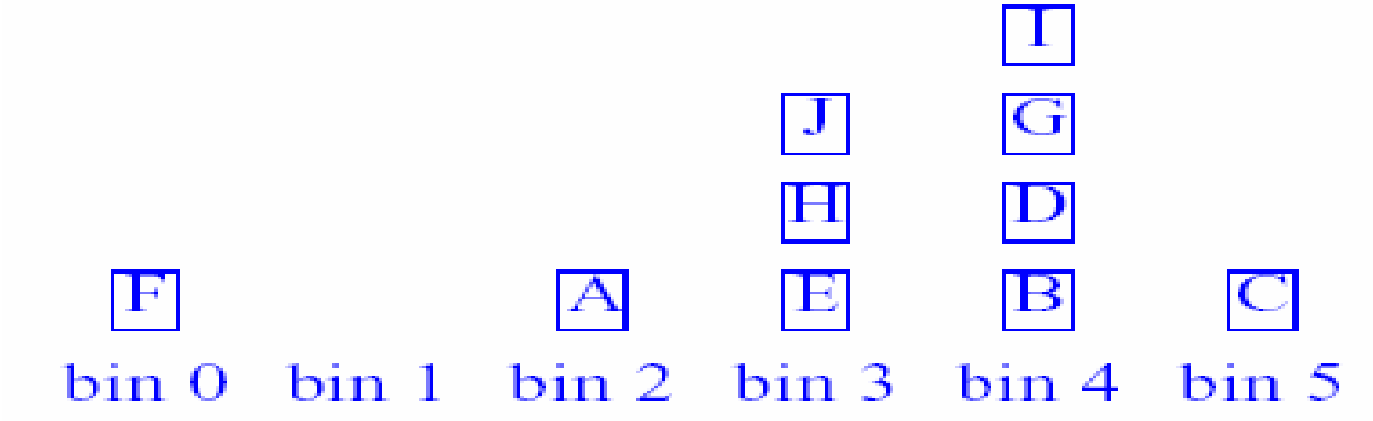
链表元素结构定义3

```
struct  studentRecord
{
  int score;
  string* name;

int operator !=(const studentRecord& x) const
       {return (score!= x.score);}
  //从studentRecord到int的类型转换
  operator int( ) const  {return  score;}
};

ostream& operator<<(ostream& out,
                              const studentRecord & x)
{out << x.score << '  ' <<*x.name<<endl; return out;}
```

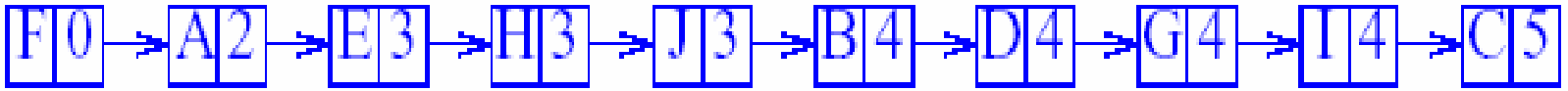
箱子排序举例



(a)输入链表



(b)箱子中的节点



(c)排好序的链表

箱子排序实现方法1

- 实现思想:
  - 1. 元素被分配到箱子之中
    - 从输入链表的首部开始，逐个删除每个元素
    - 把所删除的元素插入到相应箱子链表的表头
  - 2. 把箱子中的元素收集起来, 创建一个有序的链表.
    - 从最后一个箱子开始，从箱子链表的首部开始删除元素 。
    - 插入元素在有序表的首部

使用chain的方法进行箱子排序

```
void binSort(Chain<studentRecord>& theChain, int range)
{// 按分数排序

//对箱子初始化
Chain<studentRecord> *bin;
bin=new Chain<studentRecord> [range+1];

//把学生记录从输入链表取出，分配到相应箱子中
int numberofElements = theChain.size();
for (int i = 1; i <= numberofElements; i++)
    {
     studentRecord x=theChain.get(0);
     theChain.erase(0);
     bin[x.score].insert(0,x);
    }
```

使用chain的方法进行箱子排序

```
//从箱子中收集各元素
for (int j = range; j >= 0; j--)
    while (!bin[j].empty( ))
        {
         studentRecord x=bin[j].get(0);
         bin[j].erase(0);
         theChain.Insert(0,x);
        }
delete[ ]bin;
}
```

多次调用new,delete
元素的物理空间不断变化

- 时间复杂性：$\Theta(n+\text{range})$.

箱子排序实现方法2

- 1.元素被分配到箱子时，使用相同的物理节点
- 2. 把箱子中的元素收集时，把箱子链接起来

箱子排序作为chain类的成员方法(1/3)

```
template<class T>
void chain<T>::binSort(int range)
{// 对链表中的节点排序

   //创建并初始化箱子
   chainNode<T> **bottom, **top;
   bottom = new chainNode<T>* [range + 1];
   top = new chainNode<T>* [range + 1];
    for (int b = 0; b <= range; b++)
            bottom[b] = NULL;
```

箱子排序作为chain类的成员方法(2/3)

```
//把链表节点分配到相应箱子中
for (; firstNode; firstNode = firstNode->next)
   {// 将首节点firstNode添加到箱子中
     int theBin = firstNode->element; //元素类型转换到整型int
     if (bottom[theBin]==NULL) {//箱子为空
         bottom[theBin] = top[theBin] = firstNode;
     else //箱子非空, 放到箱子中top[theBin]之后的位置
       {
        top[theBin]->next = firstNode;
        top[theBin] = firstNode;
       }
   }
```

箱子排序作为chain类的成员方法(3/3)

```
//把箱子中的节点收集到有序链表
chainNode<T> *y = NULL;
for (int theBin = 0; theBin <= range; theBin++)
        if (bottom[theBin]!= NULL) {//箱子非空
                if (y = NULL)    // 第一个非空的箱子
                        {firstNode = bottom[theBin];
                         y = top[theBin];}
                else //不是第一个非空的箱子
                        { y->next = bottom[theBin];
                        y=  top[theBin];}
         if  (y!= NULL)    y->next = NULL;

delete [ ] bottom;
delete [ ] top;
}
```

- 时间复杂性：$\Theta(n+\text{range})$

稳定排序

- 如果一个排序算法能够保持同值元素之间的相对次序，则该算法被称之为稳定排序（stable sort）。
- 箱子排序是稳定的吗？
- (A,2),(B,4),(C,5),(D,4),(E,3),(F,0),(G,4),(H,3),(I,4),(J,3)
- (F,0),(A,2),(E,3),(H,3),(J,3),(B,4),(D,4),(G,4),(I,4),(C,5)

- 计数排序、选择排序、冒泡排序、插入排序是稳定的吗？

6.5.2 基数排序

- 对范围在0~n^c–1(c:常量)之间的n个整数进行排序
- ✓ 使用箱子排序binsort
  - range = n^c.
  - 时间复杂性: $\Theta(n+\text{range})= \Theta(n+n^c)= \Theta(n^c)$.

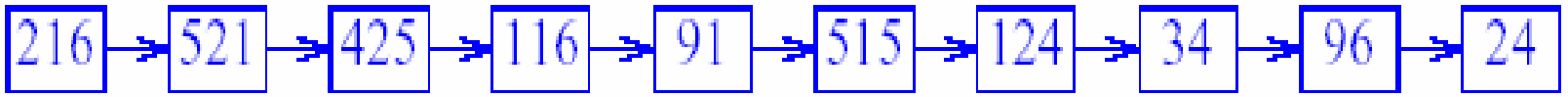
假定对范围在0~999之间的10个整数进行排序。

如果使用range=1000来调用BinSort，那么箱子的初始化将需要1000个执行步，节点分配需要10个执行步，从箱子中收集节点需要1000个执行步，总的执行步数为2010。

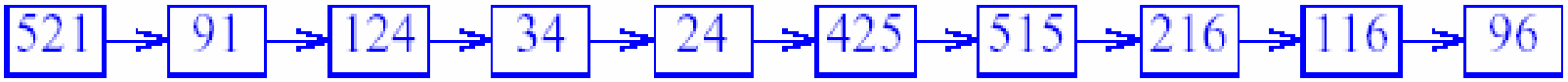
基数排序

- 基数排序（radix sort）：
  - 把数按照某种基数*r*分解为数字，然后对数字进行排序.
- 十进制数928可以按照基数10分解为数字9，2和8 .
- 用基数10来分解3725可得到3，7，2和5.
- 3725用基数60来进行分解则可以得到1，2和5. $((3725)_{10}=(125)_{60}$

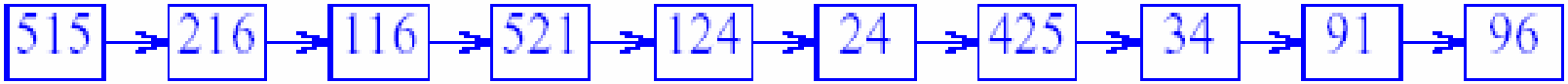
基数排序举例



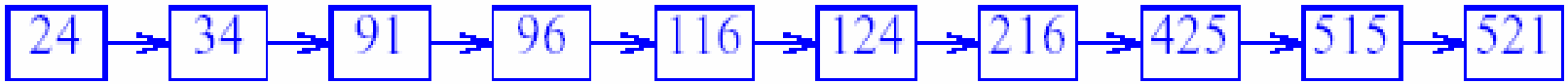
输入链表



按最后一位数字排序后的链表



按倒数第2位数字排序后的链表



按最高位数字排序后的链表

时间复杂性

对范围在0 ~ n^c-1 (c:常量)之间的n个整数进行排序。

r=n，range =n ，每个数分解的数字的个数=c

时间复杂性： Q(cn)= Q(n).

上述例子中：

进行三次排序，每次排序range=10

需要10个执行步来对箱子进行初始化

10个执行步用来把数分配至相应的箱子节点，

10个执行步用来收集箱子节点

总的执行步数为90。

比较

对1000个范围在0~10^6-1之间的整数进行排序。

使用基数r=10_6的排序方法。需要10^6执行步对箱子初始化，1000个执行步分配箱子节点，另外10^6执行步收集箱子节点，因此总的执行步数为2001000。

对于r=1000的排序。每次排序都需要3000个执行步，所以排序完成时共需要6000个执行步。

比较

若使用r=100。则需使用三次箱子排序过程依次对每两位数字进行排序，每次箱子排序需要1200个执行步，总的执行步数为3600。

如果使用r=10。则要进行六次箱子排序，每次针对一位数字，总的执行步数为6(10+1000+10)=6120。

因此，对于本例，采用基数r=100的排序效率最高。

结论

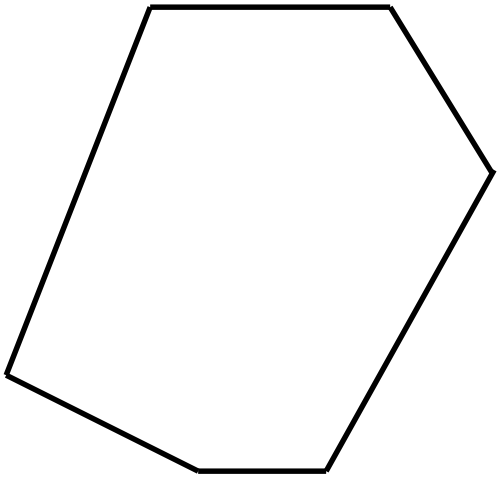
对于一般的基数r，相应的分解式为：

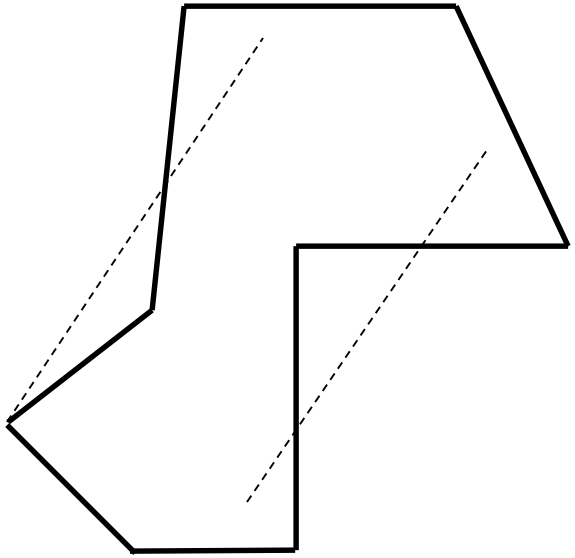
$x\%r;(x\%r^2)/r;(x\%r^3)/r^2;...$

当使用基数r=n对n个介于0~n^c-1范围内的整数进行分解时，每个数将可以分解出c个数字。

因此，可以采用c次箱子排序，每次排序时取range=n。整个排序所需要的时间为Θ(cn)= Θ(n)（因为c是一个常量）。

6.5.3 凸包

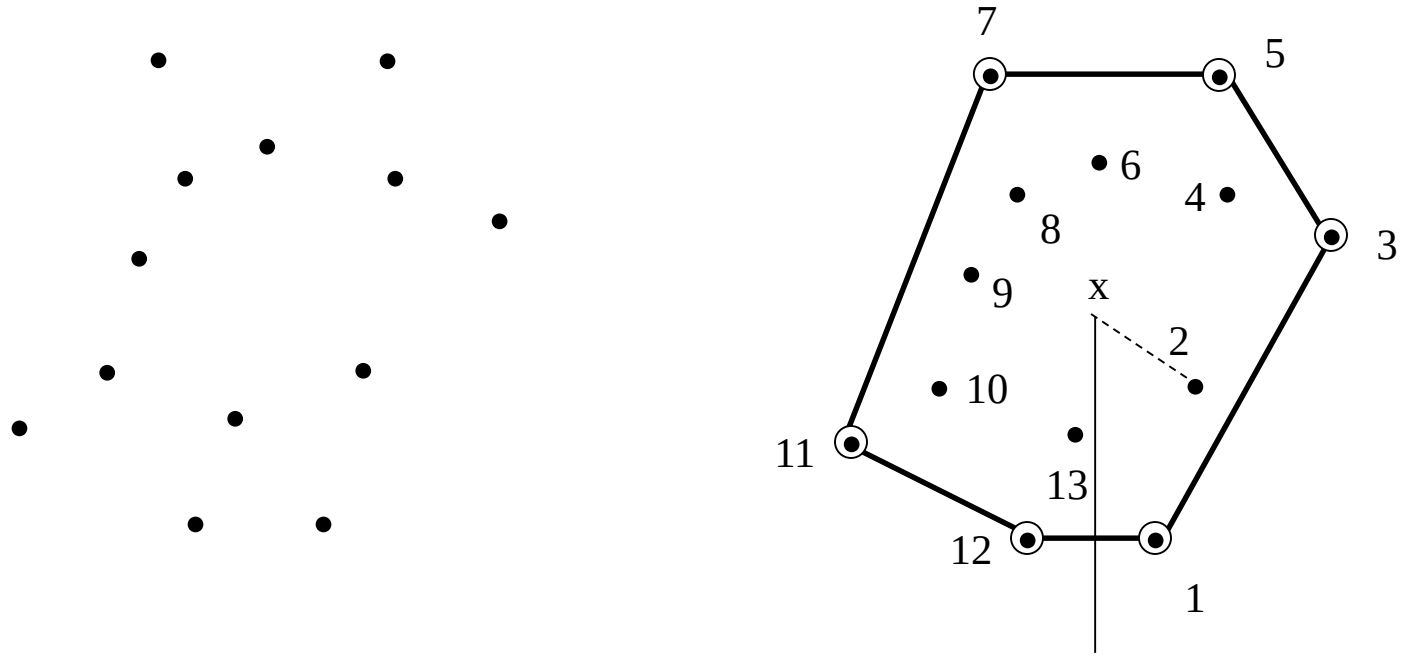




- 凸多边形:它的任意两个点的连线都不包含该凸多边形以外的点。

非凸多边形

凸包



- 一个平面点集S的凸包:包含S的最小凸多边形。

凸包

步骤 1）[处理退化情况]

如果 S 的点少于 3 个，则返回 S

如果 S 的所有点都在一条直线上，即共线，则计算并返回包含 S 所有点的最短直线的两个端点

步骤 2）[按极角排序]

在 S 的凸包内找到一个点 X

按照极角递增次序来排列 S 的点，对于极角相同的点，按照它们与 X 的距离从小到大来排列

创建一个以 S 的点为元素，按照上述顺序排列的双向循环链表

令 right 指向后继，left 指向前驱

步骤 3）[删除非极点的点]

令 p 是 y 坐标最小的点（也可以是 x 坐标最大的）

```
for(x=p,rx=x 右边的下一个点; p!=rx;)
{
      rrx= rx 右边的点;
      if(x,rx 和 rrx 的逆时针夹角小于或等于 180 度)
      {
          从链表中删除 rx;
          rx=x; x=rx 左边的点;
       }
      else{x=rx; rx=rrx;}
}
```

6.5.4 并查集

- **等价类**定义：
- U= {1,2,3,4,……n}　　//n个元素的集合
- R={(i_1,j_1),(i_2,j_2),……(i_r,j_r)}　//具有r个关系的集合

- 关系*R*是一个等价关系（equivalence relation），当且仅当如下条件为真时成立：
  - 对于所有的*a*，有(a,a)∈R （即关系是自反的）。
  - 当且仅当(b,a)∈R 时，(a,b)∈R（即关系是对称的）。
  - 若(a,b)∈R 且 (b,c)∈R，则有(a,c)∈R（即关系是传递的）。

等价关系

- 在给出等价关系*R*时，我们通常会忽略其中的某些关系，这些关系可以利用等价关系的自反、对称和传递属性来得到.
- *例:*
  - $n = 14$;
  - $R = \{ (1, 11), (7, 11), (2, 12), (12, 8), (11, 12), (3, 13), (4, 13), (13, 14), (14, 9), (5, 14), (6, 10) \}$

等价类定义

- 元素*a* 和*b* 等价
  - 如果(a,b)∈R，则元素*a* 和*b* 是等价的。

- 等价类（equivalence class）
  - 等价类是指相互等价的元素的最大集合。“最大”意味着不存在类以外的元素，与类内部的元素等价。

等价类举例

- n= 14;
- R= { (1 , 11)，(7 , 11)，(2 , 12)，(12 , 8)，
  (11 , 12)，(3 , 13)，(4 , 13)，(13 , 14)，
  (14 , 9)，(5 , 14)，(6 , 10) }

- 等价类:
  - {1, 2, 7, 8, 11, 12}
  - {3, 4, 5, 9, 13, 14}
  - {6, 10}

离线等价类

- 已知n和R，确定所有的等价类。
- 注意每个元素只能属于某一个等价类。

在线等价类

- 初始时有n 个元素，每个元素都属于一个独立的等价类。
- 需要执行的操作：
  - (1) **combine(a,b)**— 把包含a和b的等价类合并成一个等价类。
  - (2) **find(theElement)**— 确定元素**theElement**在哪个类中。
- combine(a,b) 等价于：
  - classA=find(a);
  - classB=find(b);
  - if (classA!=classB)
    unite(classA, classB);

//unite（或union）

在线等价类

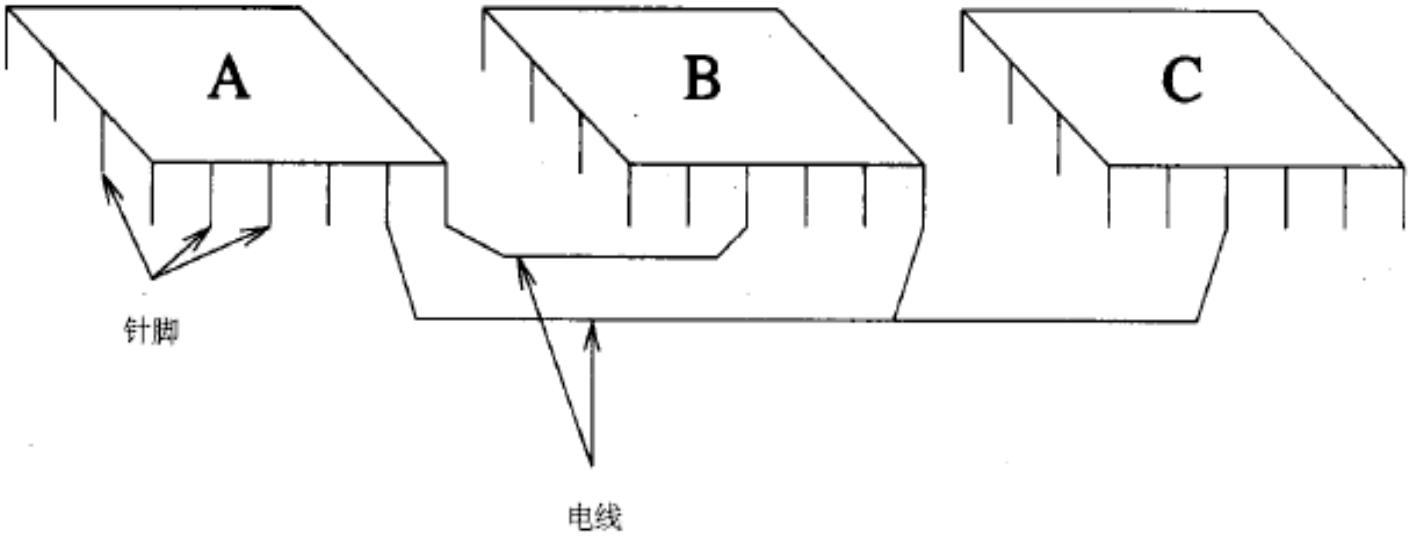
- **初始**时有n 个元素，每个元素都属于一个独立的等价类。
- {1} ,{2},.....,{n}

- 向R中添加新关系（a, b）：
  - classA=**find**(a);
  - classB=**find**(b);
  - if (classA!=classB)
    unite(classA, classB);

- 在线等价类问题，通常又称之为**并查集(union-find)问题**.

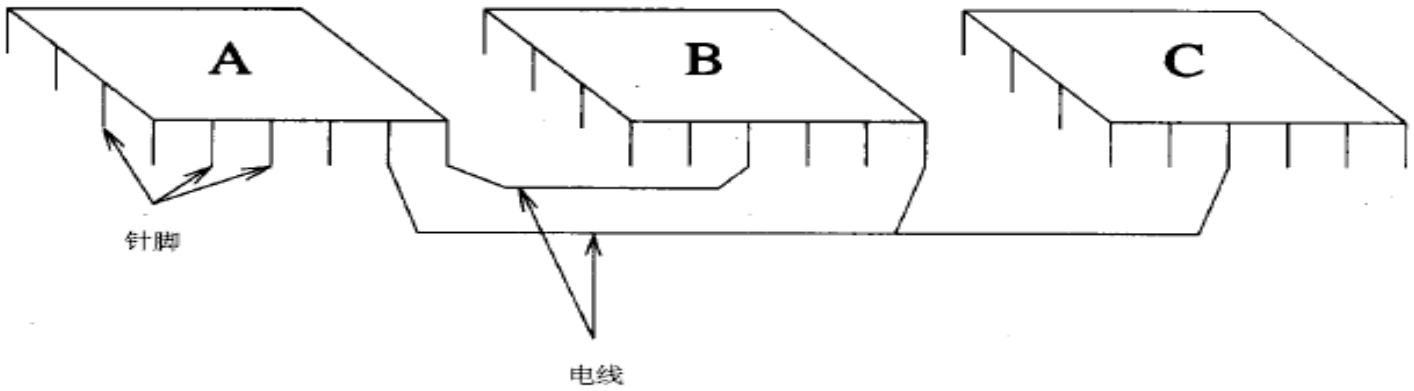
在线等价类应用—布线

- 例6-6 [布线] 一个电路由构件、管脚和电线构成。



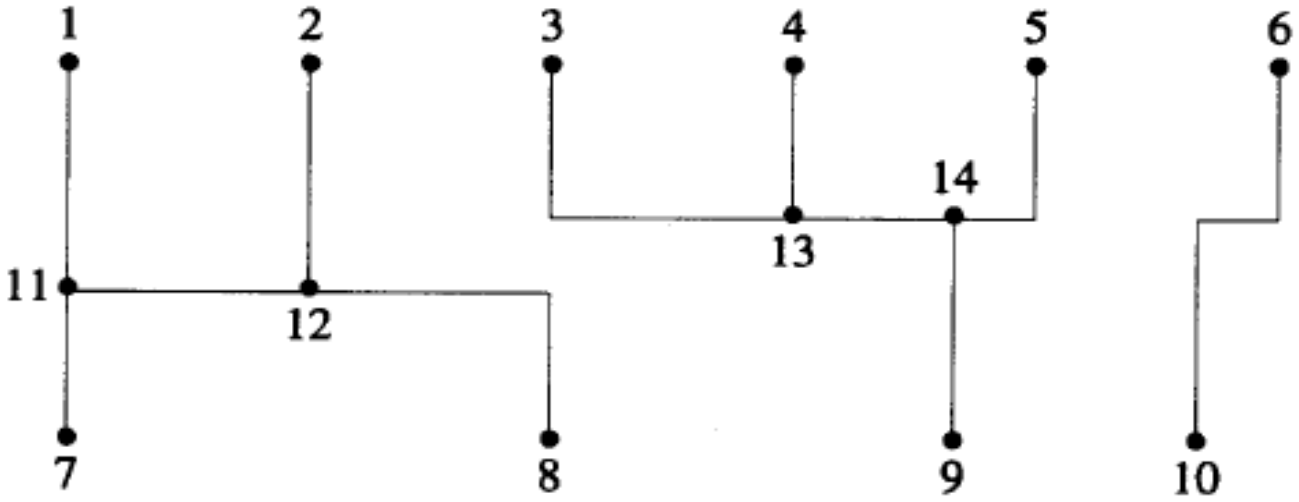
- 每根电线连接了一对管脚。

网络搜索示例



- 两个管脚a和b是电子等价(electrically equivalent),当且仅当
  - 要么有一根电线直接连接了*a*和*b*,
  - 要么存在一个管脚序列a_1，a_2，…，a_k，使得*a*, a_1；a_1, a_2；a_2,a_3；…；a_{k-1}，a_k；和a_k，*b* 均由电线直接相连
- **网络(net)**是指电子等价管脚的最大集合，“最大”是指不存在网络外的管脚与网络内的管脚电子等价。
- 问题：确定一个电路中相应的网络

网络搜索



- 管脚：1至14。连接管脚1和11的电线可以表示为(1,11)，它与(11,1)等价。
- 电线的集合为{(1,11)，(7,11)，(2,12)，(12,8)，(11,12)，(3,13)，(4,13)，(13,14)，(14,9)，(5,14)，(6,10)}。
- 该电路中所存在的网络如下：{1，2，7，8，11，12}，{3，4，5，9，13，14}和{6，10}

在线等价类第一种解决方案

- 第一种解决方案:
  - 使用一个数组equivClass
    - equivClass[e]：包含元素e 的等价类

```
int    *equivClass ,//等价类数组
         n;              //元素个数
```

第一种解决方案实现

```
void   initialize(int numberOfElements)
{//初始化numberOfElements个类，每个类仅有一个元素
   n = numberOfElements;
   equivClass=new  int [n+1];
   for (int  e=1; e<=n; e++)
       equivClass[e]=e;
}
void   unite(int classA, int  classB)
{//合并类classA和类classB,假设classA!=classB
for (int k=1; k<=n; k++)
    if   (equivClass[k]= =classB)     equivClass[k]=classA;
}
int    find(int  e)
{
  return  equivClass[e]; //搜索包含元素e的类
}
```

第二种解决方案

- 实现思想：
  - 针对每个等价类设立一个相应的链表——等价类链表
  - 每个元素都在一个等价类链表中
  - initialize ： 为每个元素设置一个只拥有该元素的链表
  - Unite： 合并两个链表
  - Find：查找元素所在的链表

第二种解决方案

- 针对每个等价类设立一个相应的链表
- node[1:n]用于描述n个元素（每个元素都有一个对应的等价类链表）
- node[e]：EquivNode类私有数据成员（int）：
  - E：元素e所在的等价类(用等价类链表中的首节点位置表示)
  - Size：元素e所在等价类中的元素数目（等价类链表中的节点数，仅当e是链表的首节点时，才定义node[e].size）
  - Link：链表指针（模拟指针）， 0表示空指针。

```
void Initialize(int n)
{// 初始化n个类，每个类仅有一个元素
    node = new EquivNode [n+1];
    for (int e = 1; e <= n; e++) {
        node[e].E = e;
        node[e].link = 0;
        node[e].size = 1;
    }
}
```

E[] 1 2 3 4 5 n

E	1	2	3	4	5					
LINK	0	0	0	0	0					
SIZE	1	1	1	1	1					

```
int Find(int e)
{ //搜索包含元素i 的类
return node[e].E;
}
```

E[] 1 2 3 4 5 n

E	1	2	3	4	5					
LINK	0	0	0	0	0					
SIZE	1	1	1	1	1					

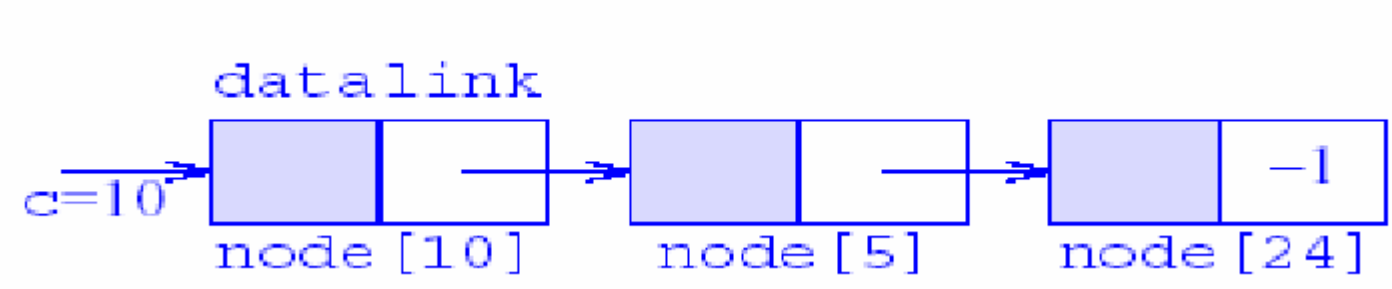
```
void Union(int i, int j)
{ //合并类i 和类j;
 // 使i 代表较小的类
    if (node[i].size > node[j].size)
        swap ( i , j ) ;   //改变较小类的E值
    int k;
    for (k = i; node[k].link; k = node[k].link)
          node[k].E = j;
    node[k].E = j; // 链尾节点
    //在链表j的首节点之后插入链表i;  并修改新链表的大小
    node[j].size += node[i].size;
    node[k].link = node[j].link;
    node[j].link = i;
}
```

E	1	2	2	4	5		
LINK	0	3	0	0	0		
SIZE	1	2	1	1	1		

E	1	2	2	2	5		
LINK	0	4	0	3	0		
SIZE	1	3	1	1	1		

补充 模拟指针

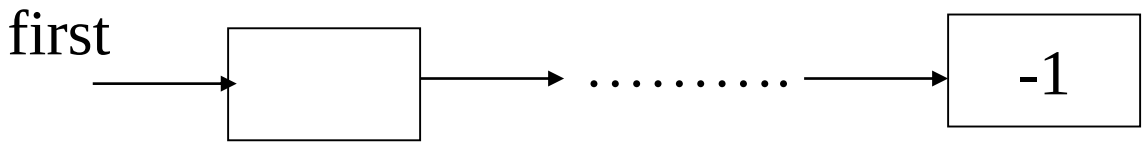
- 类似于链表描述。
  - 采用一个节点数组以及对该数组进行索引的模拟指针，可以使设计更方便、更高效。
- node[i] 表示节点 i



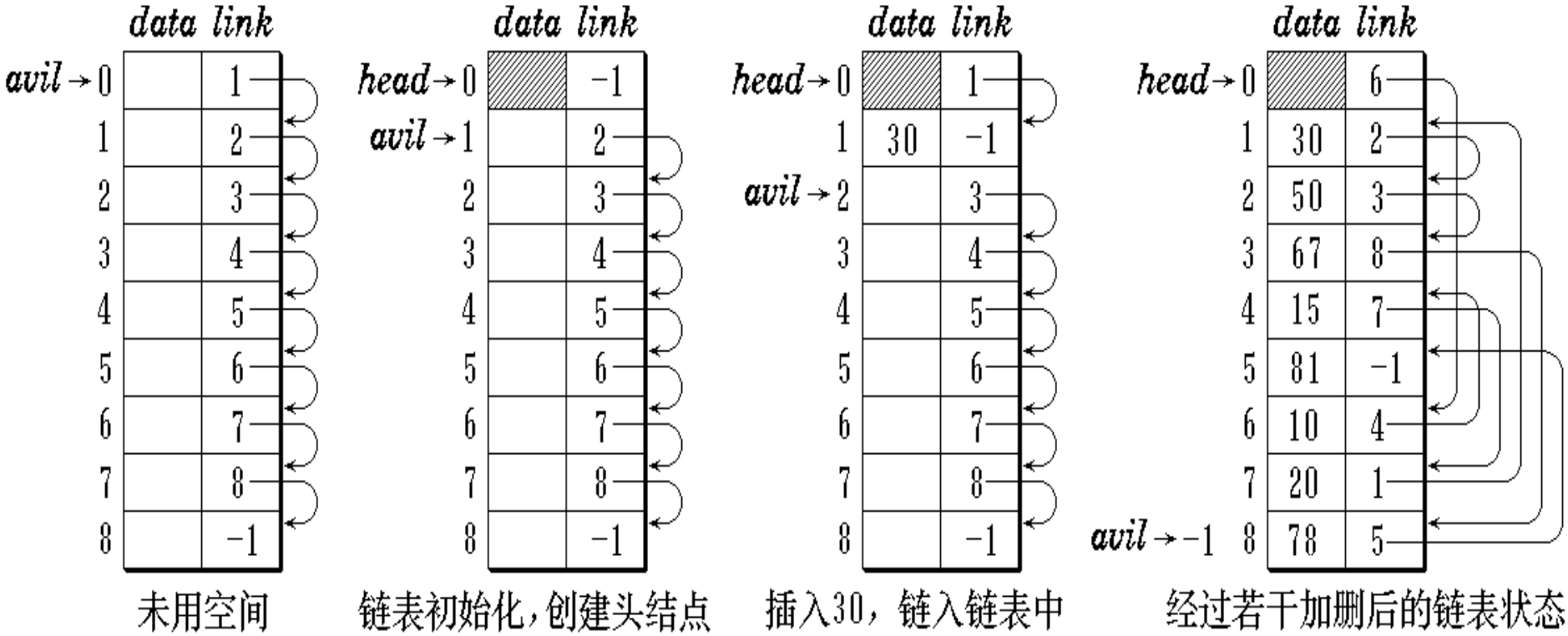
- node[10].link = 5
- node[5].link = 24
- node[24].link = -1

存储池（Storage Pool）

- 为了实现分配和释放一个节点。当前未被使用的节点将被放入一个存储池（storage pool）之中。
  - 开始时,存储池中包含了所有节点
    node[0：NumberOfNodes-1]。
  - Allocate从存储池中取出节点，每次取出一个。
  - Deallocate则将节点放入存储池中，每次放入一个。
- 用作存储池的链表被称之为可用空间表（available space list），其中包含了当前未使用的所有节点。



运算过程中存储空间大小不变

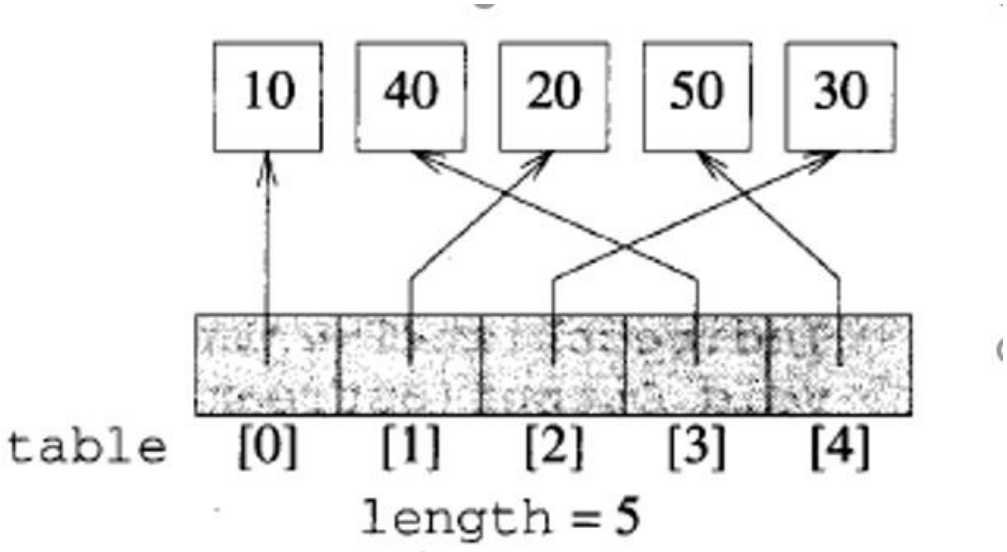


分配节点： ***j = avil; avil = A[avil].link;***

释放节点： ***A[i].link = avil; avil = i;***

补充 间接寻址

- 在间接寻址方式中，使用一个指针表来跟踪每个元素。
- 可采用一个公式来定位每个指针的位置，以便找到所需要的元素。
- 元素本身可能存储在动态分配的节点或节点数组中。



描述方法比较

描述方法	操作		
	查找第k个元素	删除第k个元素	在第k个元素后插入元素
公式化描述（公式3-1）	Θ(1)	O((n-k)s)	O((n-k)s)
链表描述（C++及模拟指针）	O(k)	O(k)	O(k+s)
间接寻址	Θ(1)	O(n-k)	O(n-k)

n：表的长度　　s：sizeof(T)

作业

6.15，6.16，6.19，6.20