

第6章

线性表——链式描述

本章内容

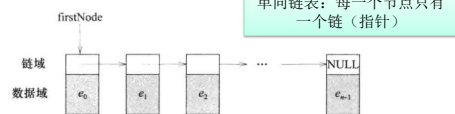
- 6.1 单向链表
- 6.2 循环链表和头结点
- 6.3 双向链表
- 6.4 链表用到的词汇表
- 6.5 应用:
 - 箱子排序(桶排序)
 - 基数排序
 - 凸包
 - 异重集

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

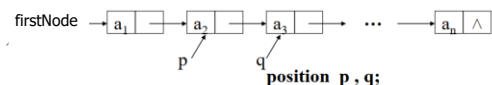
- 线性表: $L = (e_0, e_1, \dots, e_{n-1})$
 - 每个元素 e_i 都放在单独的节点中加以描述。
 - 每个节点都包含一个链域, 其值是线性表中下一个元素的地址。
 - 元素 e_i 的节点链接着 e_{i+1} 的节点, 最后一个元素 e_{n-1} 没有下一个元素, 元素 e_{n-1} 的节点无节点链接, 它的链接域为 NULL
 - 这种结构也被称作链 (chain)。



结构 'chainNode'

```
template <class T>
struct chainNode { //为链表的节点定义了数据类型
    //数据成员
    T element; //节点的数据域, 存储表元素
    chainNode<T> *next; //节点的链域, 存储下一节点的指针
    //方法
    chainNode() {}
    chainNode(const T& element) { this->element = element; }
    chainNode(const T& element, chainNode<T>* next) { this->element = element; this->next = next; }
};
```

单向链表



记法: $a_2: (*p).element;$ \Rightarrow $a_2: p \rightarrow element;$
 $q: (*p).next;$ \Rightarrow $q: p \rightarrow next;$

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;

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

7

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, 等于链表的节点个数
};

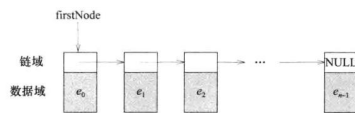
对比: 数组描述的线性表

T *element; //存储线性表元素的一维数组
int arrayLength; //一维数组的容量
int listSize; //线性表的元素个数

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

8

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

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

9

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 P116, 程序6-3
 - 时间复杂性: $\Theta(1)$

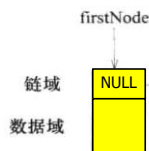
山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

10

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;
}



山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

11

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;
//sourceNode指向源chain第2个元素
chainNode<T>* targetNode=firstNode;
//targetNode指向目标chain的第一个结点
while(sourceNode!=NULL)
{//复制剩余元素
targetNode->next=new chainNode<T>(sourceNode->element);
//执行第1次是复制theList第2个元素, 第2次是复制theList第3个元素
targetNode=targetNode->next; //指向目的链第i个结点
sourceNode= sourceNode->next; //指向源链第i+1个结点
targetNode->next=NULL; }
//时间复杂性: $\Theta(\text{theList.listSize})$

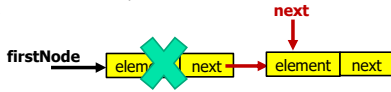
山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

12

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;
  }
}
```

实现策略：重复消除链表的首元素节点，直到链表为空



■ 时间复杂性： $O(\text{listSize})$

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

13

方法 'get'

```
template <class T>
T& chain<T>::get(int theIndex) const
{ // 返回索引为theIndex的元素(theIndex从0开始)
  //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;
}
```

必须从第一个结点开始，跟踪链域next直至找到所需的元素节点指针，即必须跟踪theIndex个指针

■ 时间复杂性： $O(\text{theIndex})$

区别：数组描述时间复杂性： $O(1)$

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

14

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： $O(\text{listSize})$

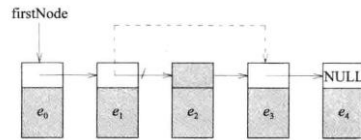
山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

15

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



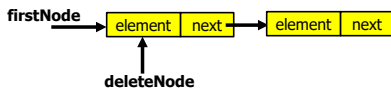
山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

16

```
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;
```

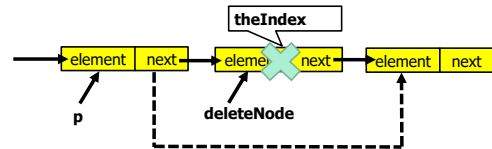
```
  if (theIndex == 0)
  { //删除表中首节点
    deleteNode = firstNode;
    firstNode = firstNode->next;
  }
}
```



山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

17

```
else { //用指针p指向要删除节点的前驱节点, theIndex-1
  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： $O(\text{theIndex})$

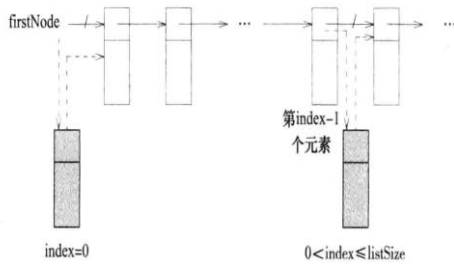
数组描述： $O(\text{listSize} - \text{theIndex})$

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

18

方法 'insert'

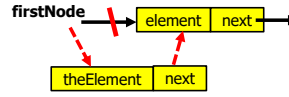
插入一个未来索引为index的节点



```
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)

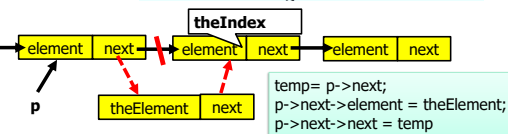
```
firstNode=new chainNode<T>(theElement, firstNode);
chainNode(const T& element, chainNode<T>* next)
{this->element = element;
 this->next = next;}
```



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);
chainNode(const T& element, chainNode<T>* next)
{this->element = element;
 this->next = next;}
```



• 时间复杂性: $O(\text{theIndex})$

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: $O(\text{listSize})$

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;

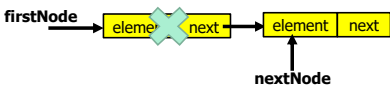
扩充类linearList

```
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

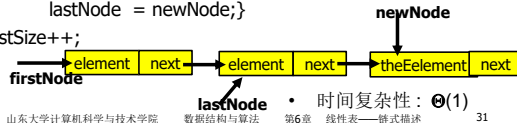
- 类extendedChain:
 - 作为类extendedLinearList的链表描述
- 类extendedChain
 - 从类chain派生而来

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; firstNode = 
}
```

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++;
}
```



山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述 31

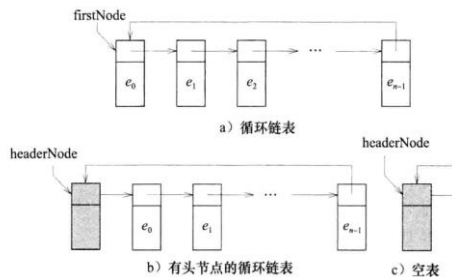
6.2 循环链表和头结点

- 下面有两条措施，可以使链表的应用代码更简洁、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）。将单向链表的尾节点与头节点链接起来，单向链表就成为循环链表。

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述 32

循环单链表



山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述 33

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;
}
```

区别单向链表： firstNode=NULL

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述 34

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)

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述 35

与“单链表中的搜索”比较

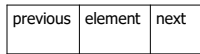
```
template<class T>
int chain<T>::indexOf(const T& theElement) const
{ //返回元素theElement首次出现时的索引,如果theElement不存在,则返回-1
  chainNode<T> *currentNode = firstNode;
  int index = 0; // currentNode的索引
  //查找元素
  while (currentNode!=NULL &&
    currentNode->element != theElement)
  {
    currentNode = currentNode->next;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

时间复杂性：O(listSize)

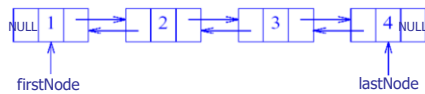
山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述 36

6.3 双向链表

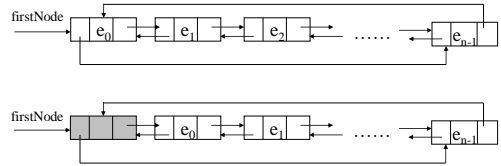
- 为了能快速找到一个节点的前驱，可以在单链表中的节点中增加一个指针域指向它的前驱。
- 每个节点：



- $L=(1, 2, 3, 4)$ 的双向链表描述



双向循环链表



- 双向链表中的插入、删除操作实现？

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？（不能）

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)

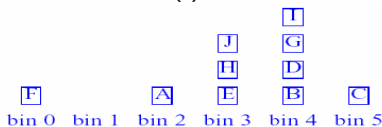
6.5.1 箱子排序

- 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. 然后把箱子链接起来就得到有序的链表。

箱子排序举例

A2 → B4 → C5 → D4 → E3 → F0 → G4 → H3 → I4 → J3

(a)输入链表



(b)箱子中的节点

F0 → A2 → E3 → H3 → J3 → B4 → D4 → G4 → I4 → C5

(c)排好序的链表

链表元素结构定义1

```
struct studentRecord//为学生记录定义的结构
{
    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);
    }
}
```

- 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

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;
```

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

多次调用new,delete
元素的物理空间不断变化

- 时间复杂性: $\Theta(n+range)$.

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];
        if (y != NULL) y->next = NULL;

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

- 时间复杂性: $\Theta(n+range)$

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

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.
 $((3725)_{10} = (125)_{60})$
 $3725 = 5 * 1 + 2 * 60 + 1 * 3600$

基数排序举例

216 → 521 → 425 → 116 → 91 → 515 → 124 → 34 → 96 → 24

输入链表

521 → 91 → 124 → 34 → 24 → 425 → 515 → 216 → 116 → 96

按最后一位数字利用箱子排序后的链表

515 → 216 → 116 → 521 → 124 → 24 → 425 → 34 → 91 → 96

按倒数第2位数字利用箱子排序后的链表

24 → 34 → 91 → 96 → 116 → 124 → 216 → 425 → 515 → 521

按最高位数字利用箱子排序后的链表

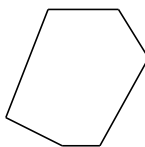
箱子排序vs基数排序

- 对 $0 \sim 999$ 之间的10个整数进行排序
- 使用 $\text{range} = 1000$ 的箱子排序, 箱子链表初始化需要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.
- 使用基数排序, 根据个位、十位、百位数字分别进行箱子排序, 每次都使用 $\text{range} = 10$ 个箱子. 每次箱子的初始化需要10个执行步, 节点分配需要10个执行步, 收集箱子节点需要10个执行步, 总的执行步数为90.

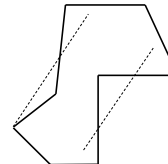
基数排序时间复杂性

- 对范围在 $0 \sim n^c - 1$ (c :常量) 之间的 n 个整数进行排序.
- 基数 $r = n$
- $\text{range} = n$
- 每个数分解的数字的个数 $= c$
- 因此, 对 n 个数, 可以用 c 次 $\text{range} = n$ 个箱子排序, 所以整个排序时间复杂性: $\Theta(cn) = \Theta(n)$.

6.5.3 凸包

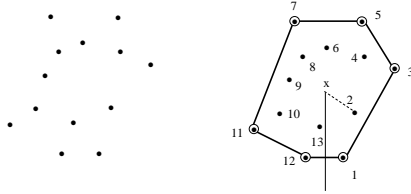


► 凸多边形: 它的任意两个点的连线都不包含该凸多边形以外的点。



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。

6.5.4 并查集

- 等价类定义：
 - $U = \{1, 2, 3, 4, \dots, n\}$ // n 个元素的集合
 - $R = \{(i_1, j_1), (i_2, j_2), \dots, (i_r, j_r)\}$ // 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 $a \in U$ ，有 $(a, a) \in R$ (即关系是自反的)。
 - $(a, b) \in R$ ，当且仅当 $(b, a) \in R$ (即关系是对称的)。
 - 若 $(a, b) \in R$ 且 $(b, c) \in R$ ，则有 $(a, c) \in R$ (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：
 - $n = 14$;
 - $R = \{(1, 11), (7, 11), (2, 12), (12, 8), (11, 12), (3, 13), (4, 13), (13, 14), (14, 9), (5, 14), (6, 10)\}$
 忽略了形如 (a, a) 的数对，以及所有对称的数对如 $(11, 1)$ ，由传递性可以得到的数对如 $(7, 12)$

等价类定义

- 元素 a 和 b 等价
 - 如果 $(a, b) \in R$ ，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。因为一个元素只能属于一个等价类，等价关系把集合 U 划分为不相交的等价类。

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：
 - $\text{classA} = \text{find}(a);$
 - $\text{classB} = \text{find}(b);$
 - if ($\text{classA} \neq \text{classB}$)
 $\text{unite}(\text{classA}, \text{classB});$

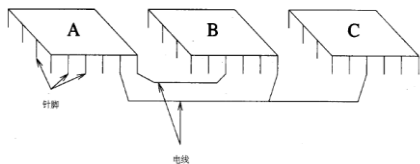
//unite (或union)

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。
- $\{1\}, \{2\}, \dots, \{n\}$
- 向 R 中添加新关系 (a, b) :
 - $\text{classA} = \text{find}(a);$
 - $\text{classB} = \text{find}(b);$
 - if ($\text{classA} \neq \text{classB}$)
 $\text{unite}(\text{classA}, \text{classB});$
- 在线等价类问题，通常又称之为并查集 (union-find) 问题。

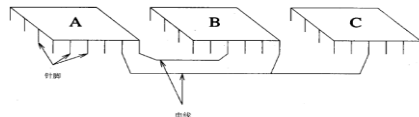
在线等价类应用—布线

- 例6-6 [布线] 一个电路由构件、管脚和电线构成。



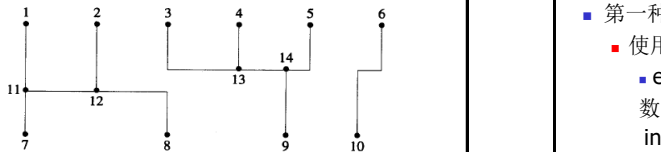
- 每根电线连接了一对管脚。

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, \dots, a_k , 使得 $a, a_1, a_2, a_3, \dots, a_{k-1}, a_k$ 和 a, 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
- 数组定义如下
- ```
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;//n是元素个数
 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;//把B类元素
 的值都修改成A类的值
}
int find(int e)
{
 return equivClass[e]; //搜索包含元素e的类
}
```

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

73

## 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

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

74

- 作业
- P124 3
- P125 15 17

山东大学计算机科学与技术学院 数据结构与算法 第6章 线性表——链式描述

75