第5-6章　线性表

----初窥门径：由第一种数据结构线性表谈起

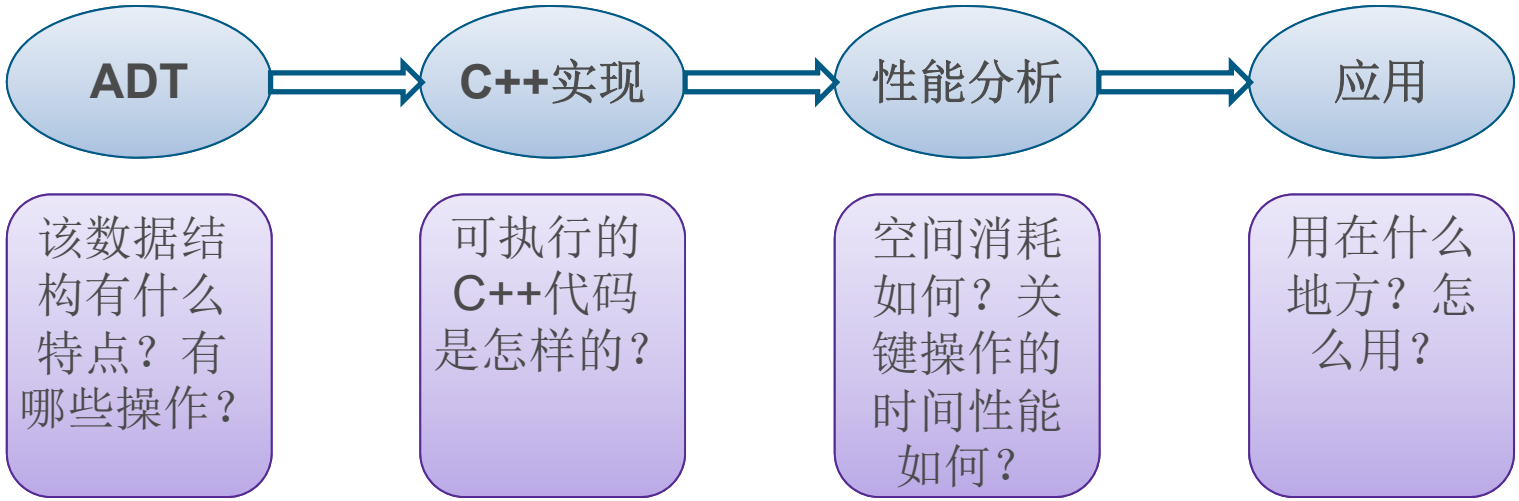
计算机学院

主要内容

- **基本概念**
- 线性表的描述与分析
  - 公式化描述（顺序表示）
  - 链表描述（链式表示）
- 线性表的应用

数据结构

- 概念
  - 包括数据对象和实例以及构成实例的每个元素之间所存在的各种关系。
- 学习顺序



线性表

- 特征
  - linear-list
  - 是最常用且最简单的一种数据结构
  - 是n个数据元素的有限序列
  - 其中数据元素可以是各种各样的，包括一个数、一个符号、一条记录等
  - 同一线性表中的元素必定具有相同特性，即属同一数据对象
  - 相邻数据元素之间存在序偶关系

线性表

- 实例形式：（e_1，e_2, …，e_n）
  - n——有穷自然数，表的长度
  - e_i——表中元素，视为原子
  - $n = 0$，空表
  - $n > 0$，e_1——第一个元素，e_n——最后一个元素
  - e_1优先于e_2，e_2优先于e_3，…—仅有的元素间关系

线性表举例

- 字母表（A, B, C, ……, Z）
- 奥斯卡近10年最佳影片

年份	片名
2012	The Artist
2013	Argo
2014	12 years a slave
2015	Birdman
2016	Spotlight
2017	Moonlight
2018	The Shape of Water
2019	Parasite
2020	Nomadland
2021	Children of Deaf Adults

线性表操作

- 创建一个线性表
- 确定线性表是否为空
- 确定线性表的长度
- 查找第k个元素
- 查找指定的元素
- 删除第k个元素
- 在第k个元素之后插入一个新元素

数据是计算机的处理对象，也是计算机的关键所在。

数据操作包括**增、删、改、查**四种

线性表之ADT

抽象数据类型*LinearList* {

实例

0或多个元素的有序集合

操作

Create()： 创建一个空线性表

Destroy()： 删除表

IsEmpty()： 如果表为空则返回true，否则返回false

Length()： 返回表的大小(即表中元素个数)

Find(*k*, *x*)： 寻找表中第*k*个元素，并把它保存到*x*中；如果不存在，则返回false

Search(*x*)： 返回元素x在表中的位置；如果*x*不在表中，返回0

Delete(*k*, *x*)： 删除表中第*k*个元素，并把它保存到*x*中；函数返回修改后的线性表

Insert(*k*, *x*)： 在第*k*个元素之后插入*x*；函数返回修改后的线性表

Output(*out*)： 把线性表放入输出流*out*之中

}

提出问题

- 我们已经知道了
  - 线性表应该是什么样子的
  - 线性表应该具备哪些功能
  - 这些构成了线性表的**逻辑结构**

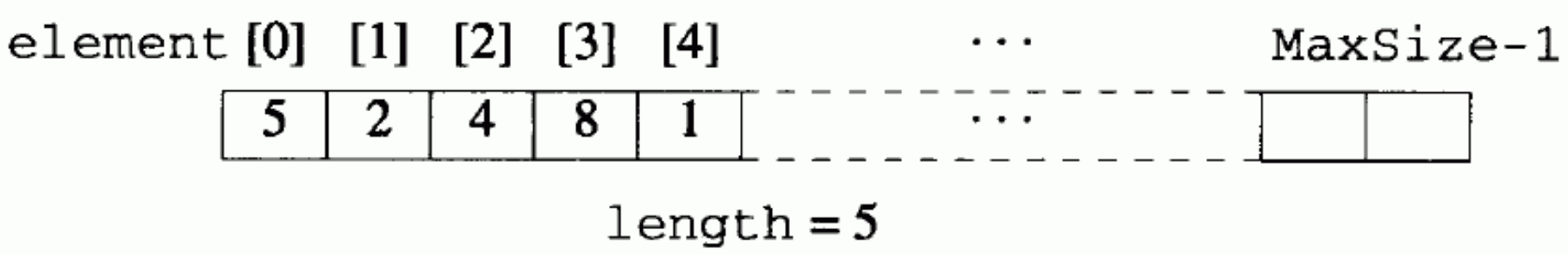
- 接下来，一个自然而然的问题是
  - 线性表的**物理结构**应该是怎样的？

主要内容

- 基本概念
- 线性表的描述与分析
  - 公式化描述（顺序表示）
  - 链表描述
- 线性表的应用

H1. 线性表的公式化描述

- 公式化描述（顺序存储）
  - 用数组来描述线性表实例
  - 数组位置——单元（cell）、节点（node）
  - 每个数组单元保存线性表的一个元素



C++类定义

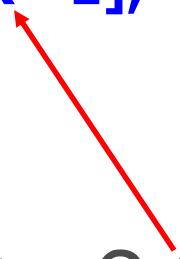
```
template<class T>
class LinearList {
  public:
    LinearList(int MaxListSize = 10); // 构造函数
    ~LinearList() {delete [] element;} // 析构函数
    bool IsEmpty() const {return length == 0;}
    int Length() const {return length;}
    bool Find(int k, T& x) const;  // 返回列表的第k个元素
    int Search(const T& x) const;  // 返回x的位置
    LinearList<T>& Delete(int k, T& x);  // 删除第k个元素，将其保存在x中
    LinearList<T>& Insert(int k, const T& x);  // 在第k个元素后插入x
    void Output(ostream& out) const;
  private:
    int length;
    int MaxSize;
    T *element; // 一维动态数组
};
```

C++实现：创建和释放

```
template<class T>
LinearList<T>::LinearList(int MaxListSize)
{// Constructor for formula-based linear list.
   MaxSize = MaxListSize;
   element = new T[MaxSize];
   length = 0;
}

~LinearList() {delete [] element;}
```

C++实现：Find(1~n)

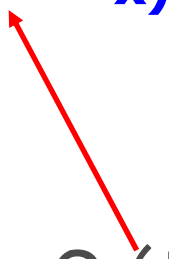
```
template<class T>
bool LinearList<T>::Find(int k, T& x) const
{// Set x to the k'th element of the list.
 // Return false if no k'th; true otherwise.
   if (k < 1 || k > length) return false; // no k'th
   x = element[k - 1];
   return true;
}
```



- 时间复杂性： $\Theta(1)$
  - 因其关键操作只有一条赋值语句

C++实现：Search

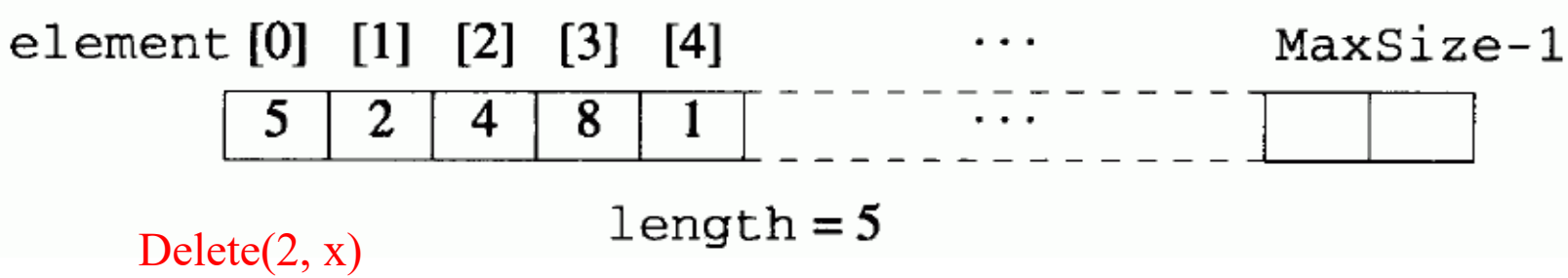
```
template<class T>
int LinearList<T>::Search(const T& x) const
{// Locate x.  Return position of x if found.
 // Return 0 if x not in list.
   for (int i = 0; i < length; i++)
      if (element[i] == x) return ++i;
   return 0;
 }
```

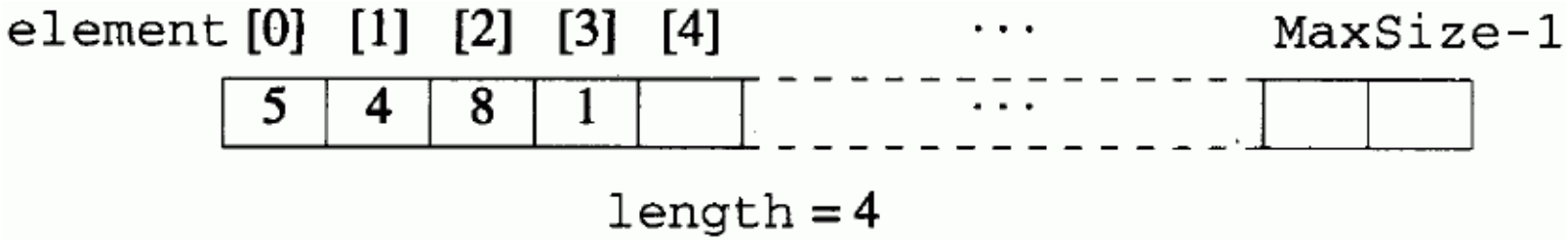


- 时间复杂性： Θ(length)

删除元素——Delete

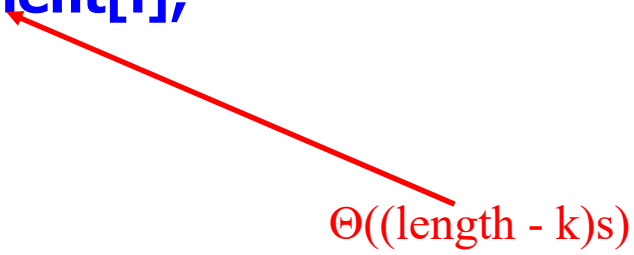
- 删除第k个元素
  - k是否合法——OutOfBounds异常， Θ(1)
  - 元素k+1，k+2，…，length向前移动一个位置——消除空位，Θ((length - k)s)





C++实现：Delete

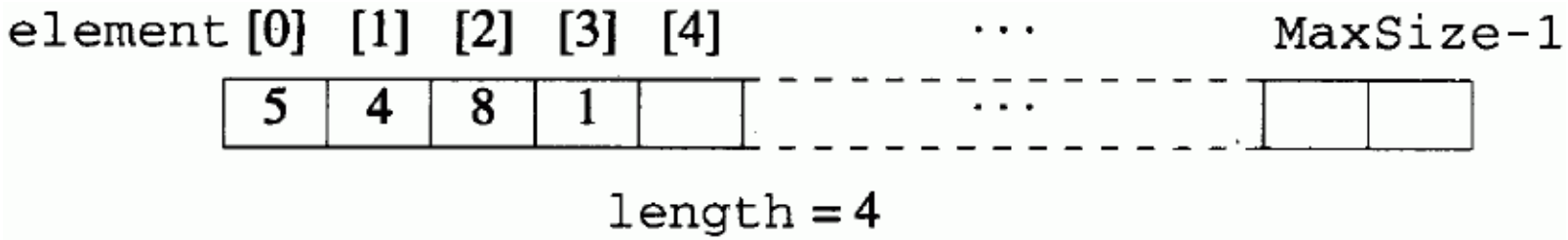
```
template<class T>
LinearList<T>& LinearList<T>::Delete(int k, T& x)
{
  if (Find(k, x)) // 当k是个合理的值时
  {
      for (int i = k; i < length; i++)
         element[i-1] = element[i];
      length--;
      return *this;
   }

  else throw OutOfBounds(); // 当k不是个合理的值时
  return *this;  // visual needs this
}
```



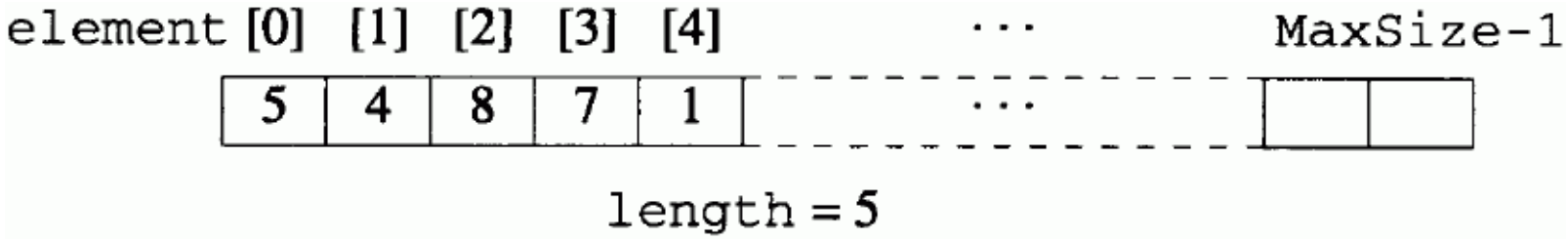
插入元素——Insert

- 插入到第k个元素之后
  - k是否合法——OutOfBounds异常， Θ(1)
  - 另一种错误：表满——数组无空间容纳新元素，抛出NoMem异常， Θ(1)
  - 常规情况下，元素k+1， k+2， …， length向后移动一个位置
    Θ((length - k)s)

插入操作示意



Insert(3, 7)



C++实现：Insert

```
template<class T>
LinearList<T>& LinearList<T>::Insert(int k, const T& x)
{
  if (k < 0 || k > length) throw OutOfBounds();//k非法
  if (length == MaxSize) throw NoMem(); //表空间满
  for (int i = length-1; i >= k; i--) //常规情况
    element[i+1] = element[i];
  element[k] = x;
  length++;
  return *this;
}
```

C++实现：Output

```
template<class T>
void LinearList<T>::Output(ostream& out) const
{// Put the list into the stream out.
  for (int i = 0; i < length; i++)
    out << element[i] << "  ";
}
// overload <<
template <class T>
ostream& operator<<(ostream& out, const
  LinearList<T>& x)
  {x.Output(out); return out;}
```

- Θ(length)

使用示例

- 创建一个大小为5的整数线性表L
- 输出该表的长度（0）
- 在第0个元素之后插入2（表为2）
- 在第一个元素之后插入6（表为2，6）
- 寻找并输出第一个元素（2）
- 输出表的长度（2）
- 删除并输出第一个元素（6）

程序

```
#include <iostream.h>
#include "llist.h"
#include "xcept.h"
void main(void){
  try {
    LinearList<int> L(5);
    cout << "Length = " << L.Length() << endl;
    cout << "IsEmpty = " << L.IsEmpty() << endl;
    L.Insert(0,2).Insert(1,6);
    cout << "List is " << L << endl;
    cout << "IsEmpty = " << L.IsEmpty() << endl;
```

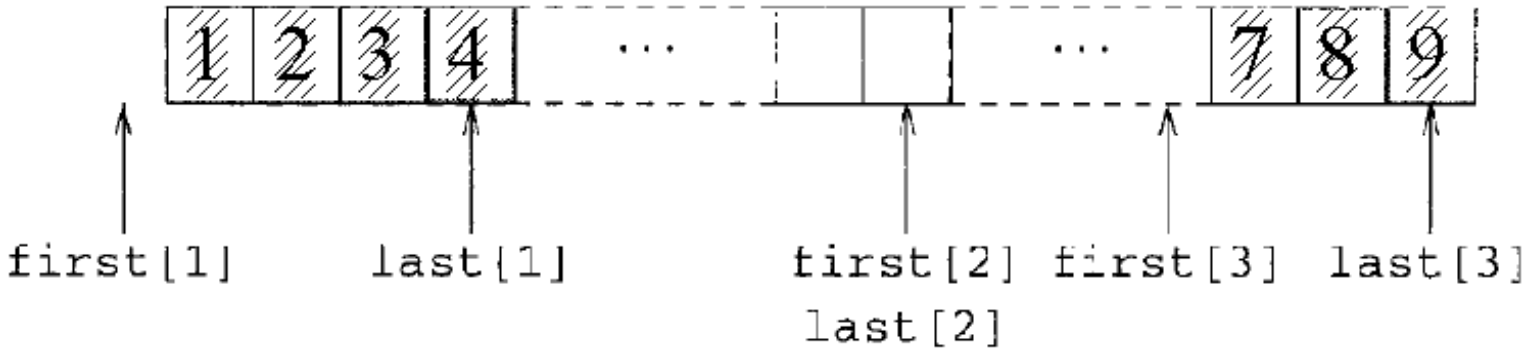
程序（续）

```
    int z;
    L.Find(1,z);
    cout << "First element is " << z << endl;
    cout << "Length = " << L.Length() << endl;
    L.Delete(1,z);
    cout << "Deleted element is " << z << endl;
    cout << "List is " << L << endl;
    }
  catch (...) {
    cerr << "An exception has occurred" << endl;
    }
}
```

评价

- 空间效率低
  - 三个表，元素总数不超过5000个，但任何一个表的元素都可能达到5000个
  - 必须为每个表分配5000个元素的空间，共需15000个元素的空间
  - **提高空间效率思路：三个表共用一个数组**
  - 两个额外数组
    - first[i]——第i个表的第一个元素在数组中的位置
    - last[i]——第i个表最后一个元素在数组中的位置

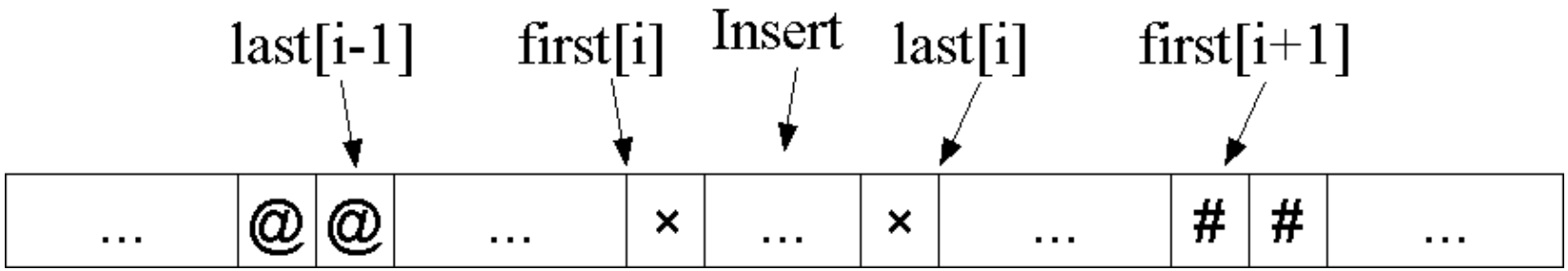
多个表共享空间



- 设置两个虚拟边界表
  - first[0]=last[0]=-1
  - first[m+1]=last[m+1]=MaxSize-1
  - 可使所有表的处理均相同，无需特别处理表1和表m

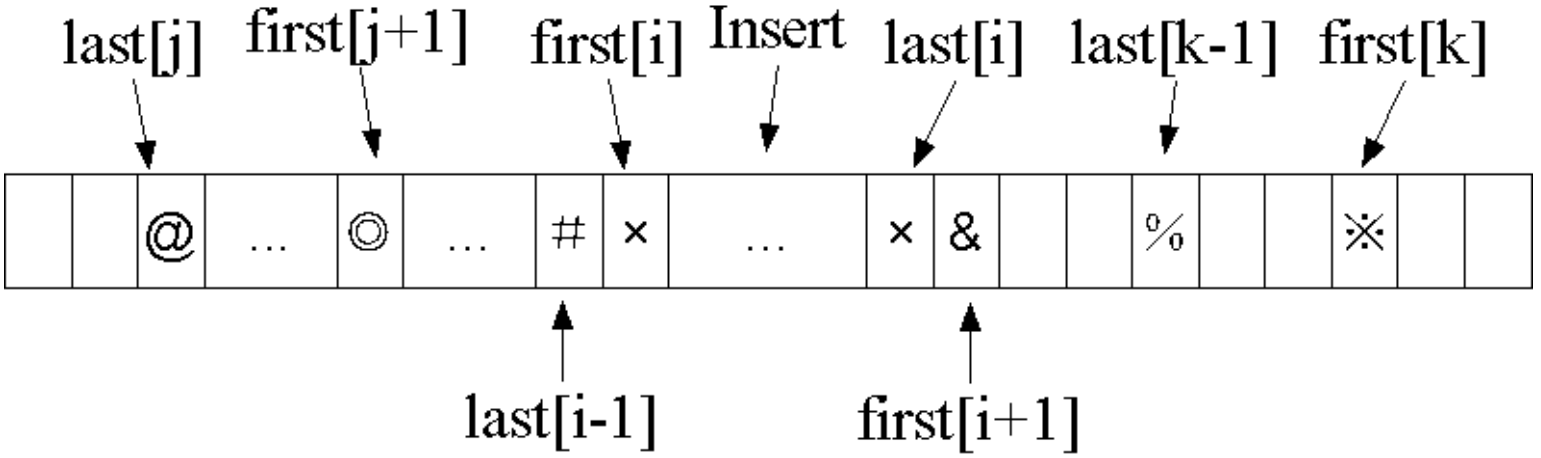
新元素插入表i的位置k之后

- 不仅要考虑表内元素的移动，由于共享一个数组，还要考虑相邻表的衔接
- 最好情况：仅需表i内部元素移动
  - 表i和i+1之间有空位：last[i] < first[i+1]，k+1~length**后移**一个位置
  - 表i-1和表i之间有空位：last[i-1] < first[i]，元素1~k-1**前移**一个位置



插入操作（续）

- 需相邻表移动的情况
  - 表j-1~表j（j<i）之间有空位，表j~i-1**前移**
  - 表k~表k+1（k>i）之间有空位，表i+1~k**后移**
  - 效率差！
- 为了改善空间复杂性，搞坏了时间复杂性



H1. 线性表顺序存储小结

- 常数级操作($\Theta(1)$)
  - IsEmpty
  - Length
  - Find
- 线性操作($\Theta(n)$)-----近似
  - Search
  - Delete
  - Insert
  - Output

主要内容

- 基本概念
- **线性表的描述与分析**
  - 公式化描述
  - **链表描述**
- 线性表的应用

H2. 线性表的链表描述

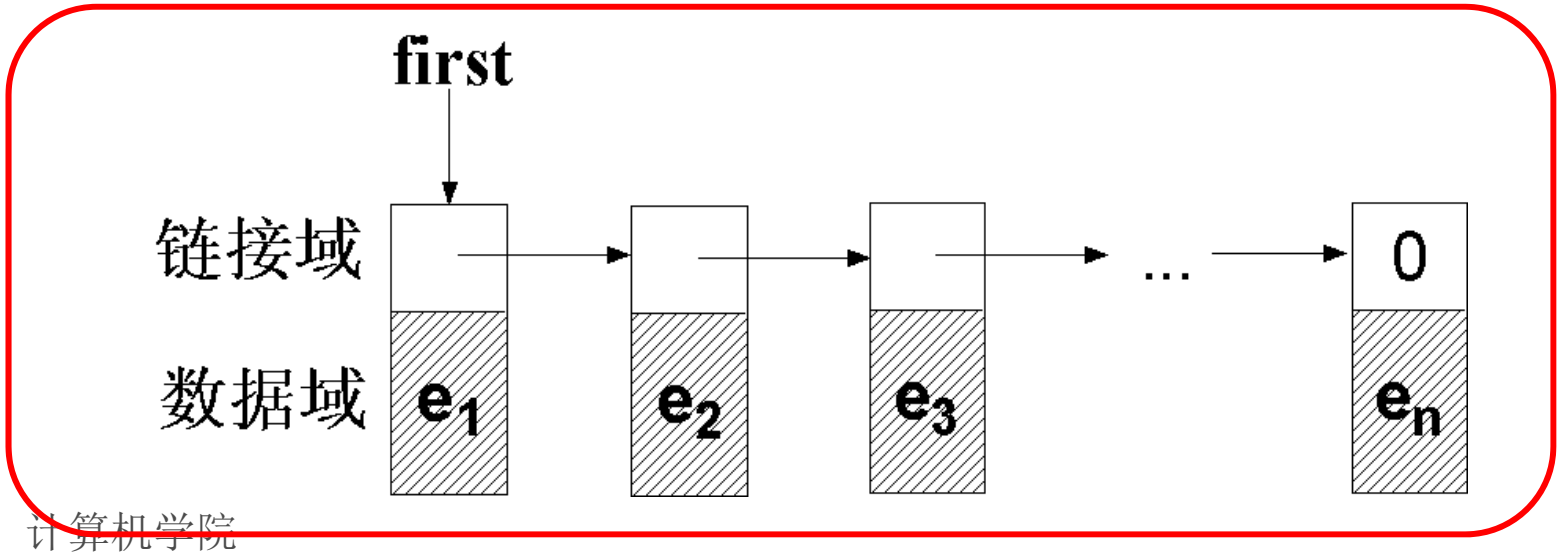
- 与公式化描述的本质区别：
  数据结构中的关联元素不保证存储空间上的关联——定位O(1)→O(n)
- 节点内容
  - 数据对象实例的元素
  - 为了正确定位，必须保留关联节点的位置

线性表的链表描述

- 线性表L=(e_1， e_2, …， e_n)
- 每个元素e_i放在不同节点的**数据域**内
- 仅有的结构就是元素次序，“e_1优先于e_2，e_2优先于e_3，…”——所谓“关联”，就是下一个节点
- 节点e_i的**链接域**（指针域）保存指向节点e_{i+1}的指针

线性表的链表描述（续）

- e_n的链接域设置为NULL（0）——无后继节点——尾节点
- e_1——首节点，没有其他节点的链接域指向它，首指针——正确定位数据的关键！
- 单向链表——链（chain）



链表节点：ChainNode

```
template <class T>
class ChainNode {
  friend Chain<T>;
  friend ChainIterator<T>;
private:
    T data;   //数据域
    ChainNode<T> *link;   //链接域
}
```

单向链表：Chain

```
template<class T>
class Chain {
  friend ChainIterator<T>;
  public:
    Chain() {first = 0;}
    ~Chain();
    bool IsEmpty() const {return first == 0;}
    int Length() const;
    bool Find(int k, T& x) const;
    int Search(const T& x) const;
    Chain<T>& Delete(int k, T& x);
    Chain<T>& Insert(int k, const T& x);
    void Output(ostream& out) const;
  private:
    ChainNode<T> *first; // 首指针：特别重要！！！
};
```

使用链
Chain<int> L;

资源，而非内容

析构函数：删除链表中所有节点

```
template<class T>
Chain<T>::~Chain()
{
  ChainNode<T> *next;  // 临时变量，用于控制指针
  while (first) {   //当下一个节点存在
    next = first->link;
    delete first;
    first = next;
  }
}
```

【绘图演示】时间复杂性Θ(n)

length操作——确定链表长度

```
template<class T>
int Chain<T>::Length() const
{
  ChainNode<T> *current = first; // 临时变量，用于控制指针
  int len = 0;
  while (current) {
    len++;
    current = current->link;
    }
  return len;
}
```

时间复杂性Θ(n)

查找操作

情况1：目标非法，k过小

情况2：目标合法，且存在

情况3：目标非法，k过大

```
template<class T>
bool Chain<T>::Find(int k, T& x) const
{
  if (k < 1) return false;
  ChainNode<T> *current = first;
  int index = 1;  // 用于与k比对
  while (index < k && current) {
    current = current->link;
    index++;
  }
  if (current) {x = current->data;
            return true;}
  return false;
}
```

寻找第k个元素
复杂性为Θ(k)

公式描述为Θ(1)

搜索操作

```
template<class T>
int Chain<T>::Search(const T& x) const
{
  ChainNode<T> *current = first;
  int index = 1;  // 用于记录x所在位置k
  while (current && current->data != x) {
    current = current->link;
    index++;
    }
  if (current) return index;
  return 0;
}
```

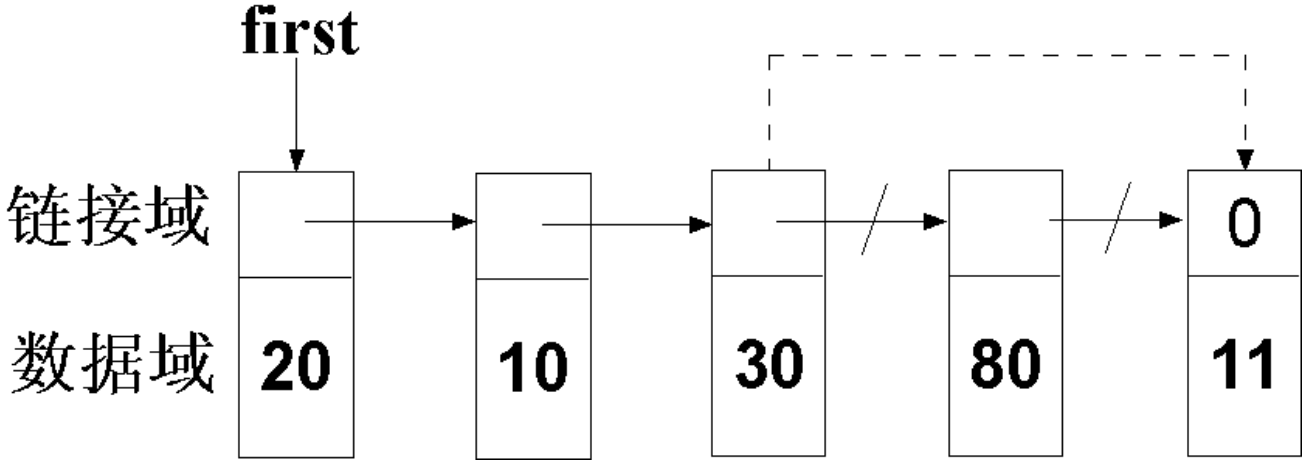
- 复杂性为Θ（n）
- 公式描述也为Θ（n）

输出函数

```
template<class T>
void Chain<T>::Output(ostream& out) const
{
  ChainNode<T> *current;
  for (current = first; current; current = current->link)
    out << current->data << "  ";
}

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

- 复杂性为Θ(n)

删除元素的方法



1. 找到前驱和后继
2. 令前驱的链接域指向后继
3. 释放被删除节点所占用的内存空间

删除操作的实现

情况1：目标非法，因其过小或表为空

情况2：目标合法，删第一个元素

情况3：目标合法，删其他元素

情况4：目标非法，因其过大

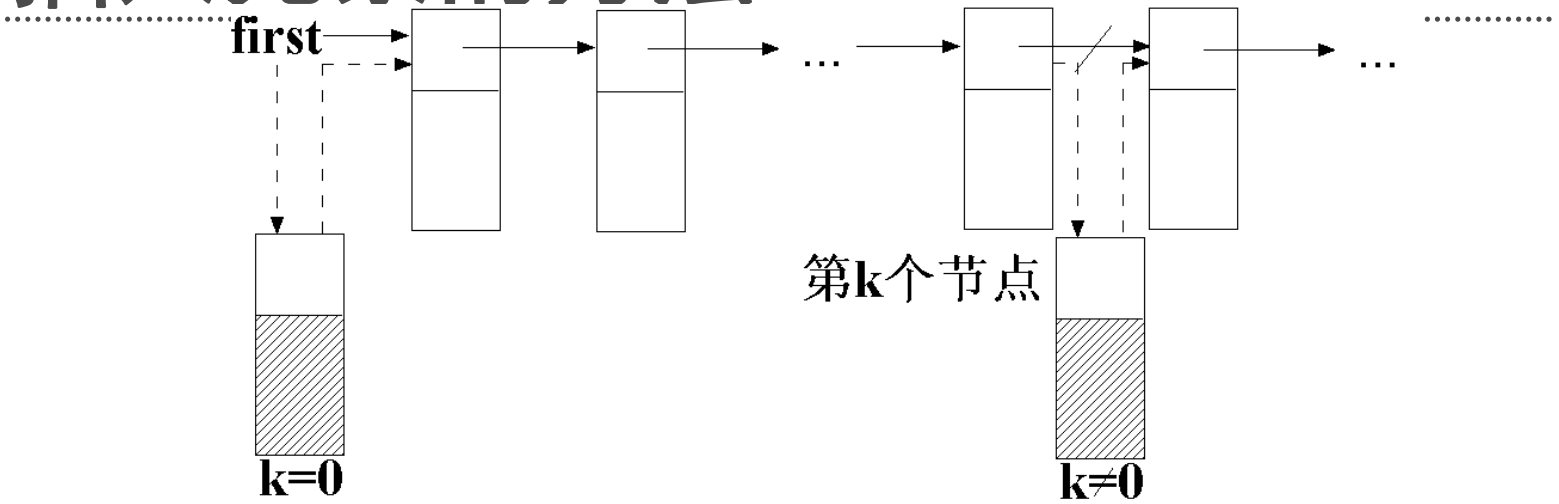
```
template<class T>
Chain<T>& Chain<T>::Delete(int k, T& x)
{
   if (k < 1 || !first)
     throw OutOfBounds(); // k不合法或列表空
   ChainNode<T> *p = first; // 临时变量，最终指向第k个元素
  if (k == 1) // 删除第一个元素
     first = first->link; // 令列表头指向下一节点——删除表头
```

删除操作的实现

```
  else { // p获取k-1个元素
    ChainNode<T> *q = first;
    for (int index = 1; index < k - 1 && q;  index++)
      q = q->link; //这句话是关键：指针右移！！！
    if (!q || !q->link)
      throw OutOfBounds(); // k>列表长度
    p = q->link; // k'th
    q->link = p->link;} // k-1项指向k+1项
  // 释放第k项占用的内存空间
  x = p->data;   delete p;   return *this;
}
```

【绘图演示】

插入元素的方法



- 新节点插入在第k个元素的后面，O(k)
  - k=0：新节点成为新的首节点，将它的link指向原首节点，而线性表的first指针指向新节点
  - k≠0：令节点k的link域指向新节点，新节点的link指向原来的节点k+1

插入操作的实现

情况1：目标非法，因其过小

情况2：目标非法，因其过大

情况3：目标合法，插入到表中或表后

情况4：目标合法，插入到表头

```
template<class T>
Chain<T>& Chain<T>::Insert(int k, const T& x)
{
  if (k < 0) throw OutOfBounds();
  ChainNode<T> *p = first; // 临时变量，最终指向第k个元素
  for (int index = 1; index < k && p;  index++)
    p = p->link;
  if (k > 0 && !p) throw OutOfBounds(); // k>列表长度
```

【绘图演示】

插入操作的实现（续）

```
// insert
ChainNode<T> *y = new ChainNode<T>;
y->data = x;
if (k) {// 插入p后面
    y->link = p->link; // 新元素指向原k+1项
    p->link = y;}    // 第k项指向新元素
else {// 插入第一个位置
    y->link = first; // 新元素指向原表头
    first = y;}          // 新元素作为新表头
return *this;
}
```

这两句话能互换顺序吗？

链表描述引出的新操作

- Erase：删除链表所有节点，释放内存空间

```
template<class T>
void Chain<T>::Erase()
{
  ChainNode<T> *next;
  while (first) {
    next = first->link;
    delete first;
    first = next;
    }
}
```

Zero操作

- 清空列表，但不删除任何节点，不释放内存空间，仅断开first指针与链表节点的连接
- **void Zero() {first = 0;}**

Append操作

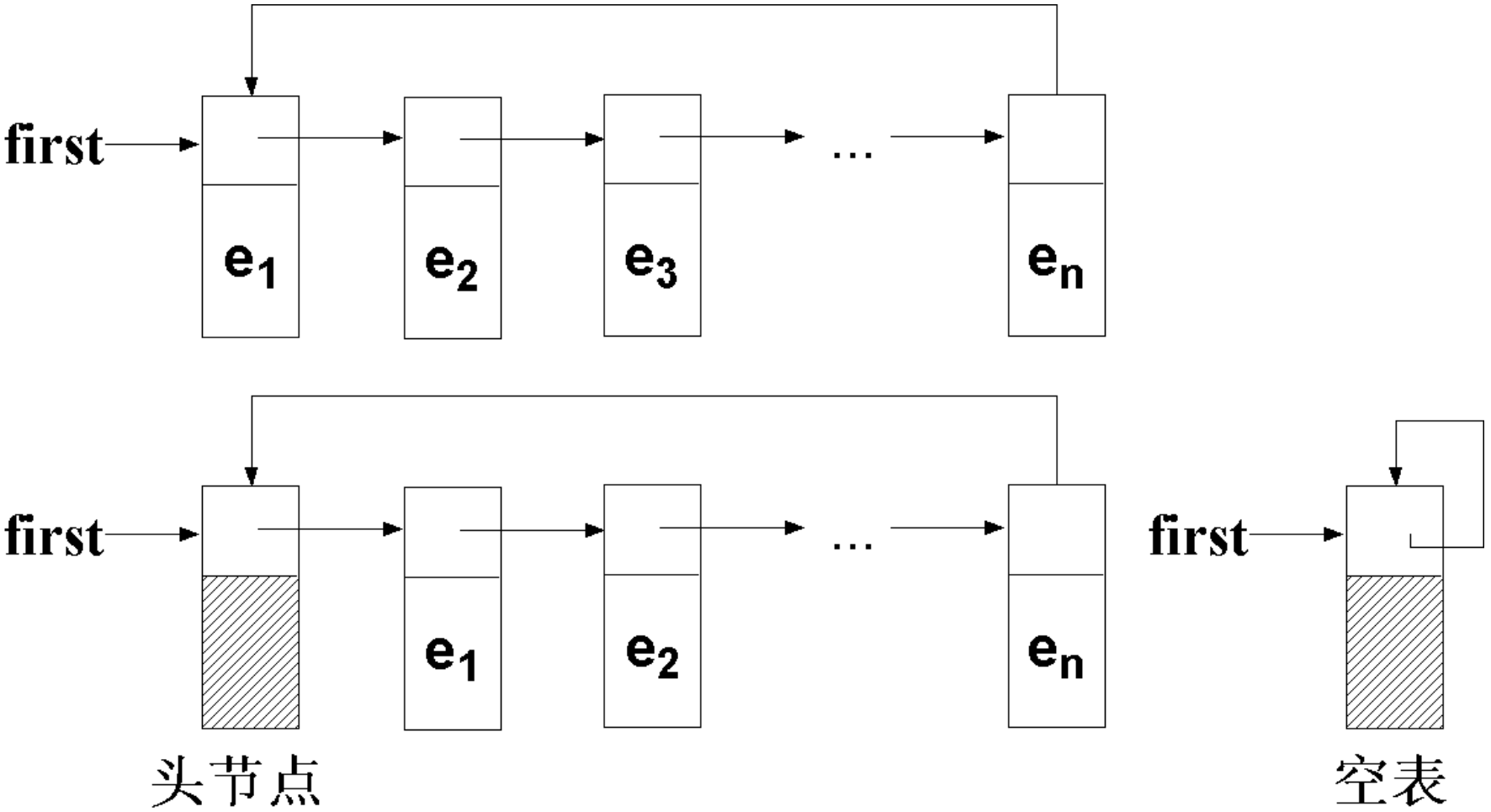
- 在链表末端添加一个元素
  - 新增一个数据成员last来跟踪尾节点

```
template<class T>
Chain<T>& Chain<T>::Append(const T& x)
{
  ChainNode<T> *y;
  y = new ChainNode<T>;
  y->data = x; y->link = 0;
  if (first) {// chain is not empty
            last->link = y;
            last = y;}
  else // chain is empty
      first = last = y;
  return *this;
}
```

循环链表

- 优化操作实现的两种链表形式
  - 单向循环链表（singly linked circular list），简称循环链表（circular list）
    ——链表最后一个节点指向第一个节点
  - 链表前部附加一个头节点（head node）

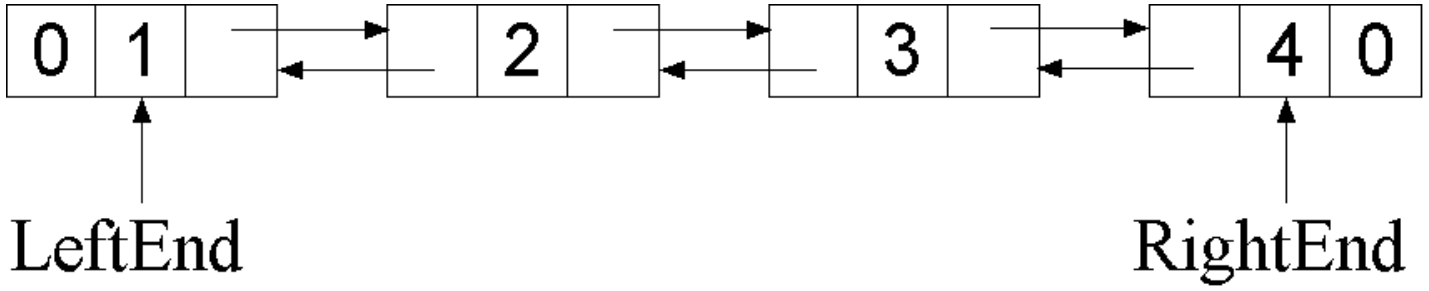
两种改进形式的结构



优化的Search函数【带头节点】

```
template<class T>
int CircularList<T>::Search(const T& x) const
{
  ChainNode<T> *current = first->link;
  int index = 1;
  first->data = x; //关键：设置一个终止条件
  while (current->data != x) {
    current = current->link;
    index++;
    }
  // are we at head?
  return ((current == first) ? 0 : index);
}
```

双向链表



- 每个节点两个指针
  - Left——指向前一节点
  - Right——指向后一节点
- 进一步简化代码设计

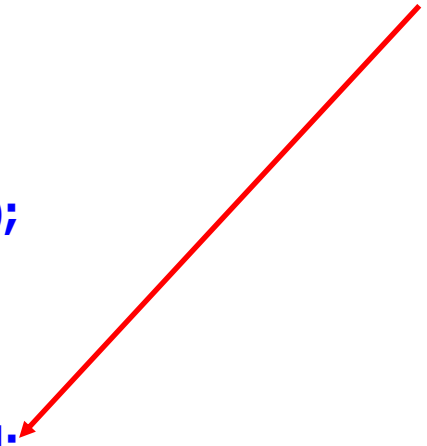
双向链表节点

```
template <class T>
class DoubleNode {
  friend Double<T>;
  private:
    T data;
    DoubleNode<T> *left, *right;
};
```

双向链表类实现（续）

```
template<class T>
class Double {
  public:
    Double() {LeftEnd = RightEnd = 0;};
    ~Double();
    int Length() const;
    bool Find(int k, T& x) const;
    int Search(const T& x) const;
    Double<T>& Delete(int k, T& x);
    Double<T>& Insert(int k, const T& x);
    void Output(ostream& out) const;
  private:
    DoubleNode<T> *LeftEnd, *RightEnd;
};
```

双向循环链表可省掉RightEnd



H2小结

- 单向链表
- 单向循环链表
- 带头节点的单向循环链表
- 双向链表
- 双向循环链表

H2小结（续）

- 空间复杂性
  - 公式化——空间全部用来保存列表元素
    链表——额外空间保存链接指针
  - 链表——动态分配，占用空间与当前列表大小成比例，利用率高
    公式化——静态分配，无法预测实际需求，浪费或不足
- 时间复杂性
  - 插入、删除操作，链表描述性能更好
  - 随机访问，公式化描述性能更优

快速练习

- 已知单向链表节点的定义，请写出以下函数的C++实现

```
~Chain();
int Length() const;
bool Find(int k, T& x) const;
int Search(const T& x) const;
Chain<T>& Delete(int k, T& x);
Chain<T>& Insert(int k, const T& x);
```

```
template <class T>
class ChainNode {
   friend Chain<T>;
private:
      T data;
      ChainNode<T> *link;
}
```

顺序表 VS 单向链表

操作(*ADT*)	顺序表	单向链表
Destroy	Θ(1)	Θ(n)
IsEmpty	Θ(1)	Θ(1)
Length	Θ(1)	Θ(n)
Find	Θ(1)	O(k)
Search	O(n)	O(n)
Delete	O((n−k)s)	O(k)
Insert	O((n−k)s)	O(k)
Output	Θ(n)	Θ(n)

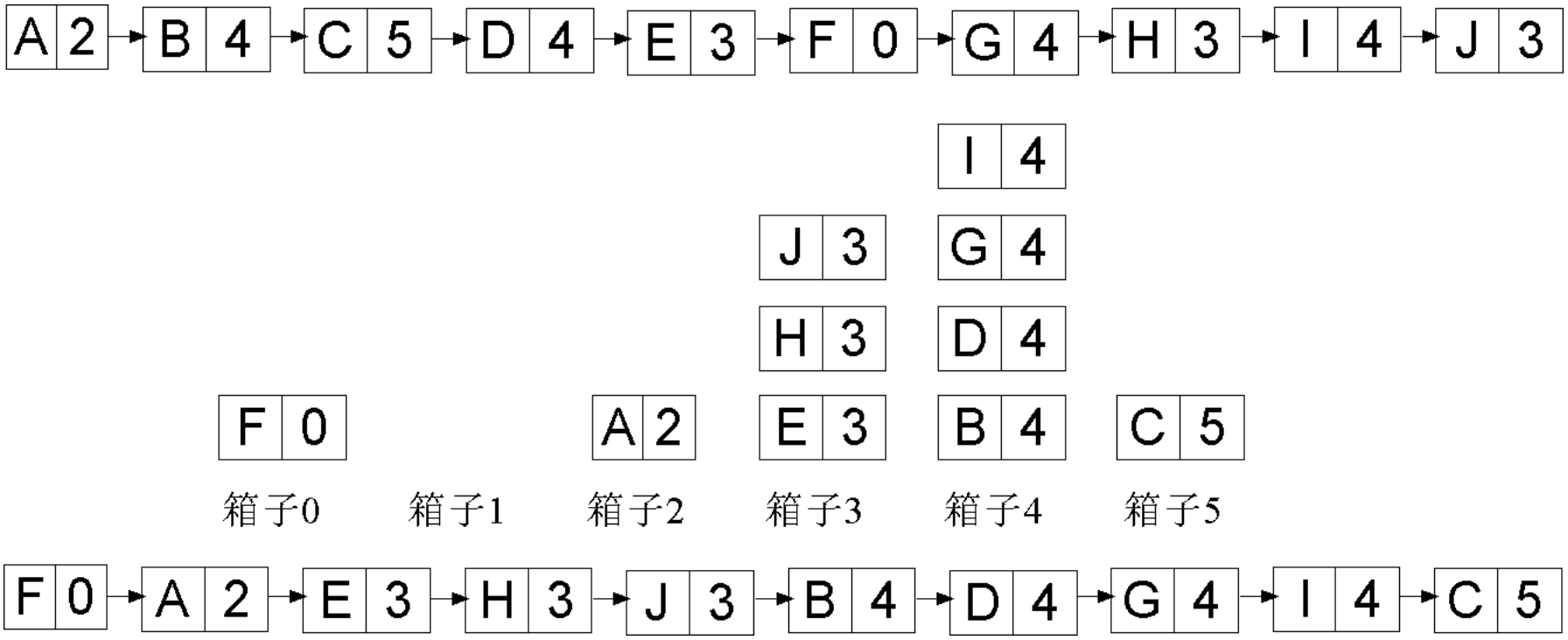
主要内容

- 基本概念
- 线性表的描述与分析
  - 公式化描述
  - 链表描述
- **线性表的应用**

箱子排序（bin sort）

- 班级学生信息——链表存储
- 按成绩排序：$O(n^2)$
- 成绩特点：0～5分，有限个
- 箱子←→分数，相同成绩→同一箱子
- 箱子按顺序连接→按分数排序

箱子排序例


A 2
B 4
C 5
D 4
E 3
F 0
G 4
H 3
I 4
J 3
I 4
J 3
G 4
H 3
D 4
F 0
A 2
E 3
B 4
C 5
箱子0
箱子1
箱子2
箱子3
箱子4
箱子5
F 0
A 2
E 3
H 3
J 3
B 4
D 4
G 4
I 4
C 5

箱子排序实现方法

- 箱子——链表
- 排序方法
  - 逐个删除链表每个节点，所删除节点放入适当的箱子中（即：插入相应链表）
  - 收集并链接所有箱子，产生排序链表

箱子排序实现方法

- 输入链表为Chain类型：
  - 连续地删除链表首元素并将其插入到相应箱子链表的首部
  - 逐个删除每个箱子中的元素（从最后一个箱子开始）并将其插入到一个初始为空的链表的首部

链表数据域——Node类

```
class Node {
  friend ostream& operator<<(ostream&, const Node &);
  friend void BinSort(Chain<Node>&, int);

public:
    int operator !=(Node x) const
      {return (score != x.score);}
  private:
    int score;
    char *name;
};
ostream& operator<<(ostream& out, const Node& x)
  {out << x.score << ' '; return out;}
```

Chain类需要Node类支持这两个操作

箱子排序实现

```
void BinSort(Chain<Node>& X, int range)
{// Sort by score.
  int len = X.Length();
  Node x;
  Chain<Node> *bin;
  bin = new Chain<Node> [range + 1];
  // 以下部分为关键：将元素放入箱子
  for (int i = 1; i <= len; i++) {
    X.Delete(1,x);
    bin[x.score].Insert(0,x);
    }
```

元素的取值范围

箱子排序实现（续）

```
for (int j = range; j >= 0; j--)
    while (!bin[j].IsEmpty()) {
      bin[j].Delete(1,x);
      X.Insert(0,x);
      }
  delete [] bin;
}
```

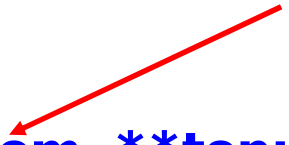
- Delete操作——Θ(1)➔
- 第一个循环Θ(n)，第二个循环Θ(n+range)
- 总复杂性Θ(n+range)

优化：作为Chain类的成员函数

- 直接操纵Chain的数据成员
- 节点重复被多个链表使用，避免对new和delete的频繁调用

```
template<class T>
void Chain<T>::BinSort(int range)
{// Sort by score.
  int b;  // bin index
  ChainNode<T> **bottom, **top;
  bottom = new ChainNode<T>* [range + 1];
  top = new ChainNode<T>* [range + 1];
  for (b = 0; b <= range; b++)
    bottom[b] = 0;
```

bottom[b]：箱子b链表首
top[b]：箱子b链表尾



优化：作为Chain类的成员函数

```
// distribute to bins
for (; first; first = first->link) {// add to bin
  b = first->data;
  if (bottom[b]) {// bin not empty
    top[b]->link = first;
    top[b] = first;}
  else // bin empty
    bottom[b] = top[b] = first;
  }
```

Delete、Insert调用变为对链表内部结构的直接操纵

旧版本：Delete——delete
Insert——new
新版本：节点从原链表摘除，直接放入箱子链表

优化：作为Chain类的成员函数

```
  // collect from bins into sorted chain
  ChainNode<T> *y = 0;
  for (b = 0; b <= range; b++)
    if (bottom[b]) {// bin not empty
      if (y) // not first nonempty bin
        y->link = bottom[b];
      else // first nonempty bin
        first = bottom[b];
      y = top[b];}
  if (y) y->link = 0;

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

时间复杂性Θ(n+2range)

稳定排序

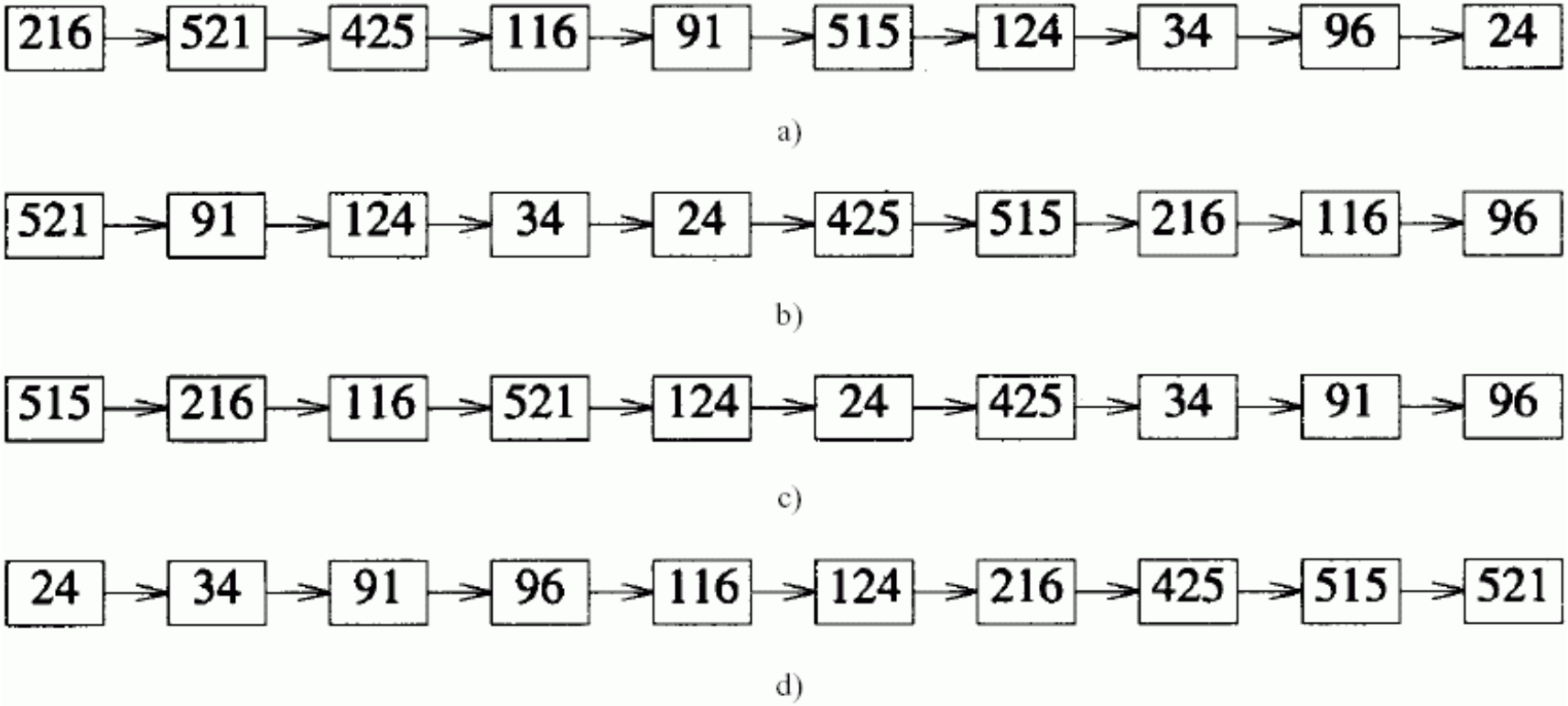
基数排序（radix sort）

- 箱子排序的优点，如果元素可能取值范围较小（n>>range），复杂性Θ(n+range)明显优于其他排序算法
- 但若n<<range，性能则会很差
- 基数排序——多阶段的箱子排序

基数排序的思想

- 将数据切为几段，每段对次序的影响力是不同的
  - 如0～r^c-1的n个整数排序，r^c-1（range）>>n
  - 将整数分解为c位r进制数
    十进制：928→9、2、8；3725→3、7、2、5
    60进制：3725→1、2、5；928→15、28
  - c个步骤：每个步骤对分解后的每一位进行箱子排序➔ Θ((n+r)*c)<<Θ(n+r^c)

基数排序例



- 箱子排序：2010个执行步（2*range+n）
- 基数排序：90个执行步（c*(2*r+n)）

基数的选定

- 1000个数，取值范围0～10^6–1
  - r=10^6——简单箱子排序：2001000步
  - r=1000：两个步骤，3000×2＝6000步
  - r=100：三个步骤，1200×3＝3600步
  - r=10：六个步骤，1020×6＝6120步
- 分解方法（由低至高）：
  - 10：x%10，（x%100）/10，（x%1000）/100，…
  - 100：x%100，（x%10000）/100，…
  - r：x%r，（x%r^2）/r，（x%r^3）/r，…

稳定性

- 箱子排序的稳定性是非常重要的
  - …926……925…
  - 个位整理：…925……926…
  - 十位整理，若不是稳定排序，可能是：
    …926……925…

本节课我们学习了：

- 第一种数据结构：线性表
- 两种存储形式：与数据结构无关
  - 顺序存储
  - 链表存储
- 如何将数据结构应用于实际问题

思考和练习

- 你是否能绘制出箱子排序的详细过程
- 咱们班共有100名同学，已知期末成绩预计符合正态分布，你是否能通过估算确定适合使用箱子排序还是基数排序？

本章结束