

# 第一章 线性表

Linear list(ordered list)

## 学习目标

- 掌握线性表的逻辑结构
- 线性表的顺序存储结构和链式存储结构的描述方法
- 熟练掌握线性表在顺序存储结构和链式存储结构的结构特点以及相关的查找、插入、删除等基本操作的实现
- 从时间和空间复杂性的角度综合比较两种存储结构的不同特点

## 本章内容

- 线性表的定义和特点
- 线性表的顺序表示和实现
- 线性表的链式表示和实现
- 顺序表和链表的比较
- 线性表的应用
- 案例分析与实现

## 线性表定义和特点

## 线性表定义

- **线性表的定义**：是由 $n$  ( $n \geq 0$ ) 个性质（类型）相同的元素组成的序列。记为：

$L = (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$   
 $a_i$  ( $1 \leq i \leq n$ ) 称为数据元素；下角标  $i$  表示该元素在线性表中的位置或序号。

$n$  为线性表中元素个数，称为线性表的长度；  
 当  $n=0$  时，为空表，记为  $L = ()$ 。

- **图示表示**：线性表  $L = (a_1, a_2, \dots, a_i, \dots, a_n)$  的图形表示如下：



## 线性表的逻辑结构

- **逻辑特征**：  $L = (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$
- **有限性**：线性表中数据元素的个数是有穷的。
- **相同性**： $a_i$  为线性表中的元素，元素类型相同
- **相继性**：
  - $a_1$  为表中第一个元素，无前驱元素； $a_n$  为表中最后一个元素，无后继元素；
  - 对于  $\dots a_{i-1}, a_i, a_{i+1} \dots$  ( $1 < i < n$ )，称  $a_{i-1}$  为  $a_i$  的直接前驱， $a_{i+1}$  为  $a_i$  的直接后继。
  - 中间不能有缺项。

## 线性表示例

- 分析26个英文字母组成的英文表
  - (A, B, C, D, ....., Z)
  - 数据元素都是字母; 元素间关系是线性
- 学生情况登记表

学号	姓名	性别	班级
161810205	于春梅	女	16级软件1班
161810260	何仕鹏	男	16级软件2班
161810284	王 爽	女	16级软件3班
161810360	王亚武	男	16级软件1班
:	:	:	:

- 数据元素都是记录; 元素间关系是线性

## 线性表案例

- 一元多项式的运算

$$P_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$$

线性表P = (p<sub>0</sub>, p<sub>1</sub>, p<sub>2</sub>, ..., p<sub>n</sub>)

$$P(x) = 10 + 5x - 4x^2 + 3x^3 + 2x^4$$

指数下标i	0	1	2	3	4
系数p[i]	10	5	-4	3	2

数组表示

(每一项的指数i隐含在其系数p的序号中)

$$R_n(x) = P_n(x) + Q_m(x)$$

线性表R = (p<sub>0</sub> + q<sub>0</sub>, p<sub>1</sub> + q<sub>1</sub>, p<sub>2</sub> + q<sub>2</sub>, ..., p<sub>m</sub> + q<sub>m</sub>, p<sub>m+1</sub>, ..., p<sub>n</sub>)



稀疏多项式如何表达?

$$S(x) = 1 + 3x^{10000} + 2x^{20000}$$

## 线性表操作

- 常用线性表操作如下：
  - 创建一个线性表
  - 销毁一个线性表
  - 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

9

## 线性表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
empty():     如果表为空则返回true, 否则返回false
size():      返回表的大小(即表中元素个数)
get(k):      返回索引为k的元素
indexOf(x):   返回元素x在表中的索引; 如果x不在表中, 返回-1
delete(k):   删除表中第k个元素, 索引大于k的元素的索引-1
insert(k, x): 在第k个元素之后插入x; 索引大于k的元素索引+1
output():    从起始位置到终止位置输出元素
  
```

}

Ps. ADT根据实际应用的不同可以进行自行定义。

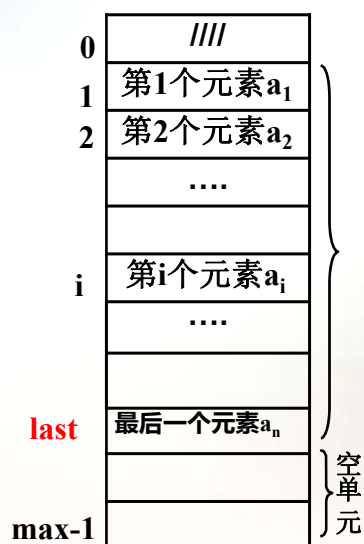
10

# 线性表的顺序存储实现

## 顺序存储实现

把线性表的元素按照逻辑顺序依次存放在数组的连续单元内；再用一个整型量表示最后一个元素所在单元的下标，即表长。

- 表存储结构特点：
  - 元素之间逻辑上的相继关系
  - 随机存取结构



## 顺序存储实现算法分析(2)

- empty()
  - size()
  - get(k)
- $O(1)$
  - $O(1)$
  - $O(1)$

## 顺序存储实现算法分析(1)

- indexOf (x)

```
template<class T>
int arrayListNoSTL<T>::
    indexOf(const T& theElement) const
{
    for (int i = 0; i < listSize; i++)
        if (element[i] == theElement)
            return i;
    return -1;
}
```

最好情况 ( $i=n+1$ ) :

基本语句执行0次，时间复杂度为 $O(1)$ 。

最坏情况 ( $i=1$ ) :

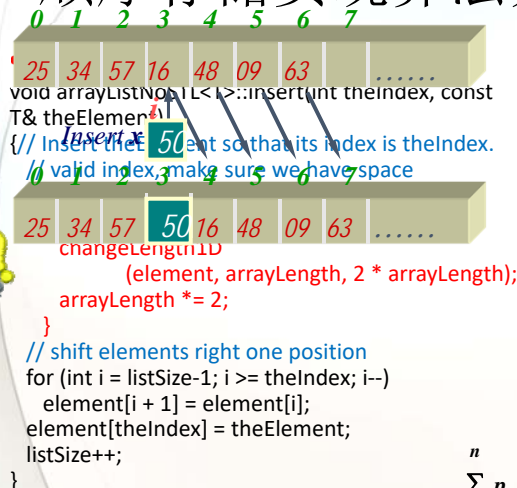
基本语句执行 $n+1$ 次，时间复杂度为 $O(n)$ 。

平均情况 ( $1 \leq i \leq n+1$ ) :

$$\sum_{i=1}^{n+1} p_i (n-i+1) = \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{n}{2} = O(n)$$

备注：示例引自《数据结构与算法分析》第5章

## 顺序存储实现算法分析(2)



```

void ArrayListNo<T>::insert(int theIndex, const
T& theElement)
{
    // Insert the element so that its index is theIndex.
    // valid index, make sure we have space
    changeLengthID
    (element, arrayLength, 2 * arrayLength);
    arrayLength *= 2;
}
// shift elements right one position
for (int i = listSize-1; i >= theIndex; i--)
    element[i + 1] = element[i];
element[theIndex] = theElement;
listSize++;
}
    
```

数组长度加倍:

$\Theta(n)$

最好情况 ( $i=n$ ):

基本语句执行0次, 时间复杂度为 $O(1)$ 。

最坏情况 ( $i=1$ ):

基本语句执行 $n$ 次, 时间复杂度为 $O(n)$ 。

平均情况 ( $1 \leq i \leq n$ ):

$$\sum_{i=1}^n p_i (n-i+1) = \frac{1}{n} \sum_{i=1}^n (n-i+1) = \frac{n+1}{2} = O(n)$$

备注: 示例引自《数据结构与算法分析》第5章

## 数组容量增长

- 数组长度每次增加1, 虽然不影响插入操作的最坏时间复杂度 $\Theta(\text{listsize})$ , 但是影响连续插入时的渐近时间复杂度。
- 连续插入表尾 $n=2^k+1$ 次操作
  - 每次容量+1:  $n$ 次增长长度为 $\Theta(\sum_{i=0}^{n-1} i) = \Theta(n^2)$ ,  $n$ 次插入时间 $\Theta(n^2)$
  - 每次容量\*2:  $n$ 次增长长度为 $\Theta(\sum_{i=0}^k i) = \Theta(n)$ ,  $n$ 次插入时间 $\Theta(n)$
- 定理: 如果按一个乘法因子来增加数组长度, 实施一系列线性表的操作所需要的时间与不用改变数组长度是对比, 至多增加一个常数因子。

ps: 扩容因子根据实际情况选择不同, 为1到2之间, 1.3、1.5、2是常见的。



## 顺序存储实现算法分析(3)

### • delete(k)

```
void arrayListNoSTL<T>::delete(int theIndex)
{ // Delete the element whose index is theIndex.
  // Throw illegalIndex exception if no such element.
  checkIndex(theIndex);
  // valid index, shift elements with higher index
  for (int i = theIndex + 1; i < listSize; i++)
    element[i-1] = element[i];

  element[--listSize].~T(); // destructor for T
}
```

### 最好情况 (i=n) :

基本语句执行0次, 时间复杂度为O(1)。

### 最坏情况 (i=1) :

基本语句执行n-1次, 时间复杂度为O(n)。

### 平均情况 (1 ≤ i ≤ n) :

$$\sum_{i=1}^n p_i (n-i-1) = \frac{1}{n} \sum_{i=1}^n (n-i-1)$$

自《数据结构与算法分析》第5章

## 线性表顺序存储优缺点

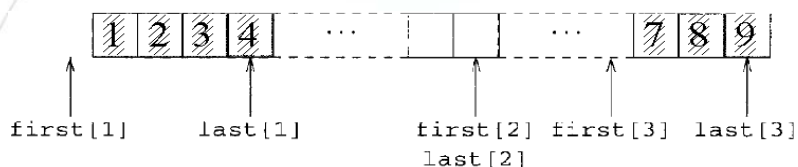
### 优点

- 无须为表示表中元素之间的逻辑关系增加额外的存储空间
- 可以快速的存取表中任意位置的元素

### 缺点

- 插入删除需要移动大量元素
- 线性表长度变化较大时, 难以确定存储空间的容量
- 造成存储空间碎片

## 其他应用-多个表共享空间



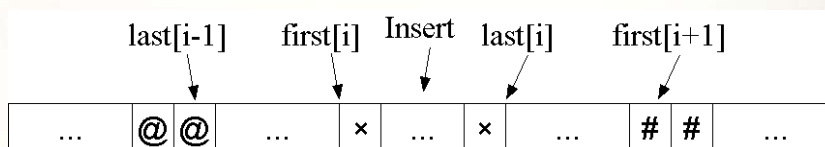
- 设置两个虚拟边界表
  - $\text{first}[0] = \text{last}[0] = -1$
  - $\text{first}[m+1] = \text{last}[m+1] = \text{MaxSize} - 1$
  - 可使所有表的处理均相同，无需特别处理表1和表m

备注：示例引自《数据结构与算法分析》第5章

19

## 共享空间插入数据

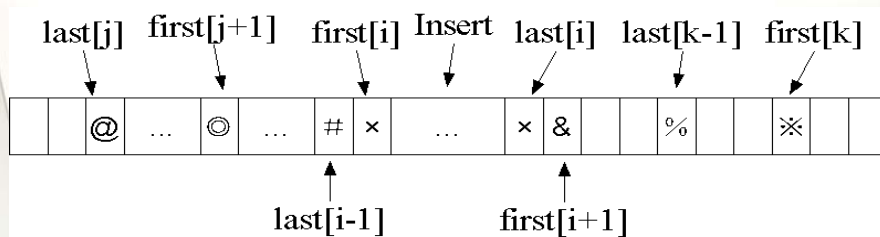
- 不仅要考虑表内元素的移动，由于共享一个数组，还要考虑相邻表的衔接
- 最好情况：仅需表i内部元素移动
  - 表i和i+1之间有空位： $\text{last}[i] < \text{first}[i+1]$ ， $k+1 \sim \text{length}$ 后移一个位置
  - 表i-1和表i之间有空位： $\text{last}[i-1] < \text{first}[i]$ ，元素 $1 \sim k-1$ 前移一个位置



20

## 插入操作（续）

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



21

## 线性表顺序存储小结

- 常数级操作( $\Theta(1)$ )
  - empty
  - size
  - get
- 线性操作( $\Theta(n)$ )
  - indexOf
  - delete
  - insert
  - output

22

# 线性表的链式存储实现

Linked Lists

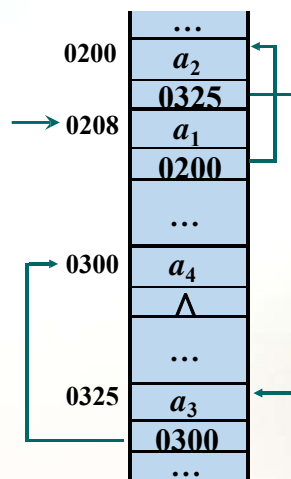
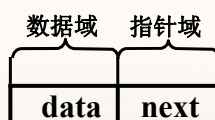
## 单链表

一个线性表由若干个结点组成，  
每个结点均含有两个域：存放元素的信息域和存放其后继结点的指针域

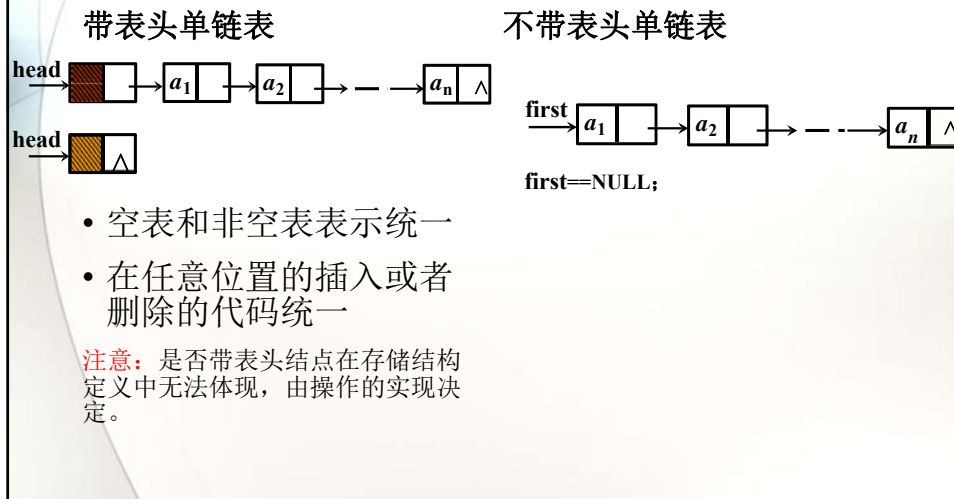
例：(a1, a2, a3, a4)的存储示意图

- 存储结构特点：逻辑次序和物理次序不一定相同；
- 元素之间的逻辑关系用指针表；
- 需要额外空间存储元素之间的关系
- 非随机存取结构

结点结构：



## 两种类型单链表实现



## 单向链表(chain)定义

### class chain

```
template<class T>
class chain : public linearList<T> { // constructor, copy constructor and
    destructor
    chain(int initialCapacity = 10);
    chain(const chain<T>&);
    ~chain();
    // ADT methods
    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;
protected:
    void checkIndex(int theIndex) const;
    chainNode<T>* firstNode; // pointer to first node in chain
    int listSize;           // number of elements in list
};
```

### class ChainNode

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

**备注：**示例引自《数据结构与算法分析》第6章

## chain实现算法分析(1)

- empty()
- size()
- 时间复杂度 $\Theta(1)$
- 时间复杂度 $\Theta(1)$

size可以通过常量的时间维护一个变量来获取时间复杂度为 $\Theta(1)$

## chain实现算法分析(2)

- 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;
    }
}
```

时间复杂度 $\Theta(n)$

## chain实现算法分析(3)

- 根据序号获取节点值

```
template<class T>
T& chain<T>::get(int theIndex) const
{
    // Return element whose index is theIndex.
    // move to desired node
    chainNode<T>* currentNode = firstNode;
    for (int i = 0; i < theIndex; i++)
        currentNode = currentNode->next;

    return currentNode->element;
}
```

时间复杂度依赖于index的值

**最好情况 ( i=1 ) :**

基本语句执行1次, 时间复杂度为  $O(1)$ 。

**最坏情况 ( i=n ) :**

基本语句执行n次, 时间复杂度为  $O(n)$ 。

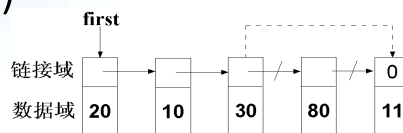
**平均情况 (  $1 \leq i \leq n$  ) :**

$$\sum_{i=1}^n p_i (i) = \frac{1}{n} \sum_{i=1}^n (i) = \frac{n+1}{2} = O(n)$$

## chain实现算法分析(4)

- 根据节点序号删除节点实现

```
void chain<T>::erase(int theIndex){
    .....
    // valid index, locate node with element to delete
    chainNode<T>* deleteNode;
    if (theIndex == 0)
    {
        // remove first node from chain
        deleteNode = firstNode;
        firstNode = firstNode->next;
    }
    else
    {
        // use p to get to predecessor of desired node
        chainNode<T>* p = firstNode;
        for (int i = 0; i < theIndex - 1; i++)
            p = p->next;
        deleteNode = p->next;
        p->next = p->next->next; // remove
    }
    listSize--;
    delete deleteNode;
}
```



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

时间复杂度依赖于index的值

**最好情况 ( i=1 ) :**

后移指针的基本语句执行1次, 删除节点执行1次, 时间复杂度为  $O(1)$ 。

**最坏情况 ( i=n ) :**

后移指针的基本语句执行n次, 删除节点执行1次, 时间复杂度为  $O(n)$ 。

**平均情况 (  $1 \leq i \leq n$  ) :**

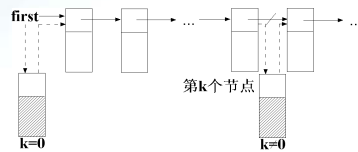
$$\sum_{i=1}^n (i+1) = \frac{1}{n} \sum_{i=1}^n (i+1) = 1 + \frac{n+1}{2}$$

**思考：**如果已知结点对象，删除的时间复杂度？

## chain实现算法分析(5)

- 根据节点序号插入节点实现

```
void chain<T>::insert(int theIndex, const T&
theElement)
{
    if (theIndex < 0 || theIndex > listSize)
    { .....
    }
    if (theIndex == 0)    // insert at front
        firstNode =
            new chainNode<T>(theElement, firstNode);
    else
    { // find predecessor of new element
        chainNode<T>* p = firstNode;
        for (int i = 0; i < theIndex - 1; i++)
            p = p->next;
        // insert after p
        p->next =
            new chainNode<T>(theElement, p->next);
    }
    listSize++;
}
```



- $k=0$ : 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

- $k \neq 0$ : 令节点k的link域指向新节点, 新节点的link指向原来的节点

时间复杂度依赖于index的值

最好情况 ( $i=1$ ):

创建节点执行1次, 时间复杂度为 $O(1)$ 。

最坏情况 ( $i=n$ ):

后移指针的基本语句执行 $n-1$ 次, 创建节点执行1次, 时间复杂度为 $O(n)$ 。

平均情况 ( $1 \leq i \leq n$ ):

$$\sum_{i=1}^n p_i (i) = \frac{1}{n} \sum_{i=1}^n (i) = \frac{n+1}{2} = O(n)$$

## chain其他可能需要的操作

- 获取结点的后驱 • 时间复杂度 $\Theta(1)$
- 获取结点的前驱 • 时间复杂度 $\Theta(n)$

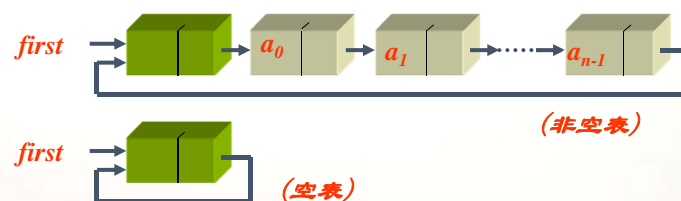


## 链表的其它实现

- 双链表 **doubly linked list**
- 循环链表 **circular linked list**

## 循环链表存储结构

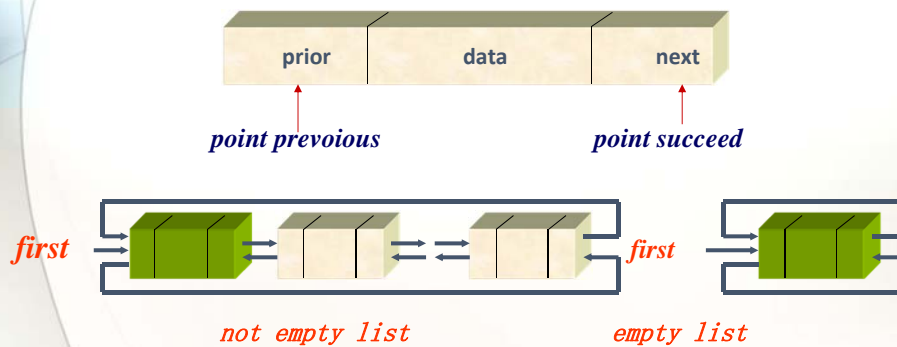
- The last node keep a pointer back to the first. We can search all nodes if we have any one node address.



- **思考:** 头指针如何设置? 如何判断循环结束?

## 双向链表存储结构

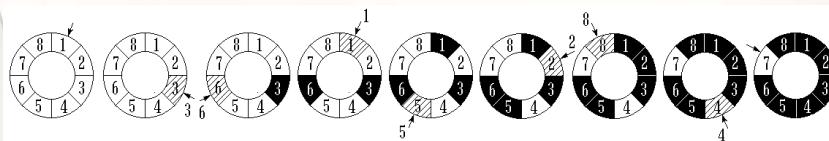
### • Node structure of doubly linked list:



## 循环链表应用

### • Use circularly linked list to solve Josephus problem

- N people sitting around. First person begins to number from one, and continue doing this one by one at clockwise, when it comes to the person who numbers m, then let him out. The next person begins to number from one, when it comes to the person who numbers m, then let him out. Continue doing this until there is only one person left. Then this person is the winner.
- For example:  $n = 8$   $m = 3$



### Solution of Josephus problem

```
#include <iostream.h>
#include "CircList.h"
void Josephus ( int n, int m ) {
    for ( int i=0; i<n-1; i++ ) { //执行n-1次
        for ( int j=0; j<m-1; j++ ) Next ( );
        cout << "Delete person " <<
            getData ( ) << endl; //数m-1个人
        Remove ( ); //删去
    }
}
```

```
void main ( ) {
    CircList<int> clist;
    int n, m;
    cout << "Enter the Number of Contestants?";
    cin >> n >> m;
    for ( int i=1; i<=n; i++ ) clist.insert (i);
    //形成约瑟夫环
    clist.Josephus (n, m); //解决约瑟夫问题
}
```

## 链表和顺序表对比

### 链表

- 存储分配方式
  - 灵活，易扩充，存储空间为 $n(s+4(8))$
- 时间性能
  - get  $\Theta(n)$
  - Insert & delete  $\Theta(n)$  or  $\Theta(1)$

### 顺序表

- 存储分配方式
  - 不宜扩充，需要预分配，占用空间在 $ns \sim 4ns$ 之间
- 时间性能
  - get  $\Theta(1)$
  - Insert & delete  $\Theta(n)$

## 顺序表 VS 单向链表

操作(ADT)	顺序表	单向链表
<i>Destroy</i>	$\Theta(1)$	$\Theta(n)$
<i>Empty</i>	$\Theta(1)$	$\Theta(1)$
<i>get</i>	$\Theta(1)$	$\Theta(n)$
<i>indexOf</i>	$O(n)$	$O(n)$
<i>Delete</i>	$O((n-k)s)$	$O(k)$ 或 $O(1)$
<i>Insert</i>	$O((n-k)s)$	$O(k)$ 或 $O(1)$
<i>Output</i>	$\Theta(n)$	$\Theta(n)$

例如：find and delete  
or find and insert

## 链表的实际性能测量

操作 (50000)	arrayList	chain
最好插入	0.0399484	0.072168
平均插入	0.780289	11.6358
最坏插入	1.03867	7.98563
最好删除	0.00157794	2.68836
平均删除	0.152727	12.0154
最坏删除	0.257369	10.6086

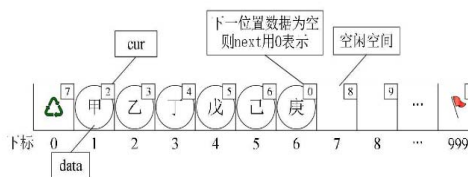
### • 示例：arraylist平均插入代码

```
linearList<double> *x = new arrayList<double>(50000);
LARGE_INTEGER t1, t2, t3, t4, tc;
QueryPerformanceFrequency(&tc);
QueryPerformanceCounter(&t1);
for (int i = 0; i < 50000; i++) {
    if (i % 1000 == 0) {
        int j = rand() % i;
        else int j = 0;
    }
    QueryPerformanceCounter(&t2);
    cout << " randtime50000:" <<
        (t2.QuadPart - t1.QuadPart)*1.0 / tc.QuadPart << endl;

    QueryPerformanceCounter(&t3);
    for (int i=0; i < 50000; i++) {
        if (i % 1000 == 0) {
            x->insert(rand() % i, i);
            else x->insert(i, i);
        }
    }
    QueryPerformanceCounter(&t4);
    cout << " time50000:" <<
        (t4.QuadPart - t3.QuadPart - (t2.QuadPart - t1.QuadPart))*1.0
        / tc.QuadPart << endl;
    //扣除生成随机数的时间
```

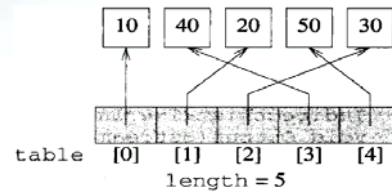
## 静态链表

- 把线性表的元素存放在数组的单元中（不一定按逻辑顺序连续存放），每个单元不仅存放元素本身，而且还要存放其后继元素所在的数组单元的下标（游标）。
- 如右图的实现：
  - 一个元素的cur存储下一个空闲空间的下标，最后一个元素的cur存储第一个元素的下标。



备注：示例引自《大话数据结构》3.12

## 间接寻址



- 元素的存储——链接方式，动态存储，通过指针访问，连续元素存储不连续
- 指针的组织——公式化方式，连续元素的指针在存储上是连续的

备注：示例引自《数据结构、算法与应用-第一版》3.12

43

## 元素i的定位方式

- 首先找到元素指针 $\text{table}[i-1]$ ——公式化
- 再由指针找到元素——多一次间接寻址
- 链表描述：指针分散在节点中——类似超链接方式
- 间接寻址：指针集中在数组中——类似目录索引方式

44

## 间接寻址列表类定义

```
template<class T>
class IndirectList {
public:
    IndirectList(int MaxListSize = 10); // constructor
    ~IndirectList(); // destructor
    bool IsEmpty() const {return length == 0;}
    int Length() const {return length;}
    bool Find(int k, T& x) const;
    int Search(const T& x) const;
```

$\Theta(1)$ , 与顺序存储描述的实现非常相似

45

## 间接寻址列表类定义（续）

```
    IndirectList<T>& Delete(int k, T& x);
    IndirectList<T>& Insert(int k, const T& x);
    void Output(ostream& out) const;
private:
    T **table; // 1D array of T pointers
    int length, MaxSize;
};
```

46

## 构造函数和析构函数

```
template<class T>
IndirectList<T>::IndirectList(int MaxListSize)
{ // Constructor.
    MaxSize = MaxListSize;
    table = new T *[MaxSize];
    length = 0;
}
template<class T>
IndirectList<T>::~~IndirectList()
{ // Delete the list.
    for (int i = 0; i < length; i++)
        delete table[i];
    delete [] table;
}
```

数组保存元素指针，比元素所需空间少很多，可一定程度解决数组描述的缺点

47

## Find函数的实现

```
template<class T>
bool IndirectList<T>::Find(int k, T& x) const
{ // Set x to the k'th element in the chain.
    // Return false if no k'th; return true otherwise.
    if (k < 1 || k > length) return false; // no k'th
    x = *table[k - 1];
    return true;
}  $\Theta(1)$ ，与公式化描述实现相似
```

- Find、Length、IsEmpty——随机访问（根据元素索引编号访问）

48



## 删除操作

```
template<class T>
IndirectList<T>& IndirectList<T>::Delete(int k, T& x)
{
    // Set x to the k'th element and delete it.
    // Throw OutOfBounds exception if no k'th element.
    if (Find(k, x)) {
        // move pointers k+1, ..., down
        for (int i = k; i < length; i++)
            table[i-1] = table[i];
        length--;
        return *this;
    }
    else throw OutOfBounds();
    return *this; // Visual needs this line
}
```

数组：定位 $\Theta(1)$ ，删除元素移动 $\Theta((\text{length} - k)s)$   
 链表：定位 $\Theta(k)$ ，删除 $\Theta(1)$   
 间接：定位 $\Theta(1)$ ，删除指针移动 $\Theta(\text{length} - k)$

49

## 插入操作

```
template<class T>
IndirectList<T>& IndirectList<T>::Insert(int k, const T& x)
{
    // Insert x after the k'th element.
    if (k < 0 || k > length) throw OutOfBounds();
    if (length == MaxSize) throw NoMem();
    // move one up
    for (int i = length-1; i >= k; i--)
        table[i+1] = table[i];
    table[k] = new T;
    *table[k] = x;
    length++;
    return *this;
}
```

时间复杂性类似删除操作

50

## 间接寻址小结

- 结合公式化描述和链表描述的优点
  - 定位元素是 $\Theta(1)$
  - 其他多数操作也是 $\Theta(1)$ ，而不是 $\Theta(n)$ !
  - 插入、删除无需移动数据
  - 空间上接近链表，优于公式化

51

## 几种描述方法的比较

描述方法	操作		
	查找第k个元素	删除第k个元素	插入第k元素后
数组	$\Theta(1)$	$O((n-k)s)$	$O((n-k)s)$
链表	$O(k)$	$O(k)$	$O(k+s)$
间接寻址	$\Theta(1)$	$O(n-k)$	$O(n-k)$

- 间接寻址结合了公式化和链表的优点，适用于
  - 表元素本身很大
  - 插入、删除操作频繁
  - 确定表长度、按编号访问元素操作频繁

52

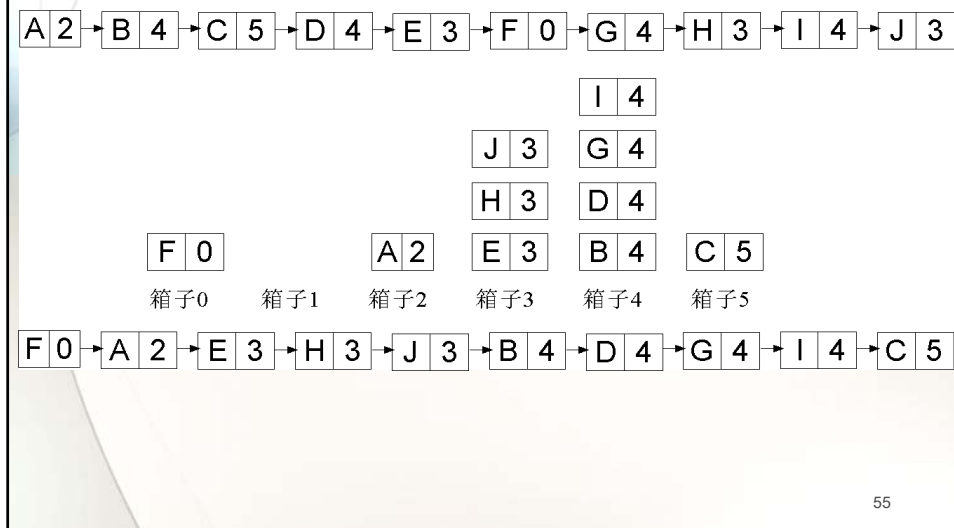
# 线性表应用

线性表作为数据结构的基础有很多的应用场景

## 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 $\leftrightarrow$ 分数, 相同成绩 $\rightarrow$ 同一箱子
- 箱子按顺序连接 $\rightarrow$ 按分数排序

## 箱子排序例



## 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

57

## 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

58

## 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

59

## 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—— $\Theta(1)$ →
- 第一个循环 $\Theta(n)$ ，第二个循环 $\Theta(n+range)$
- 总复杂性 $\Theta(n+range)$

60

## 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

61

## 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

62

## 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;
}
```

时间复杂度 $\Theta(n+2range)$

稳定排序

63

## 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 ( $n \gg range$ )，复杂度 $\Theta(n+range)$ 明显优于其他排序算法
- 但若 $n \ll range$ ，性能则会很差
- 基数排序——多阶段的箱子排序

64

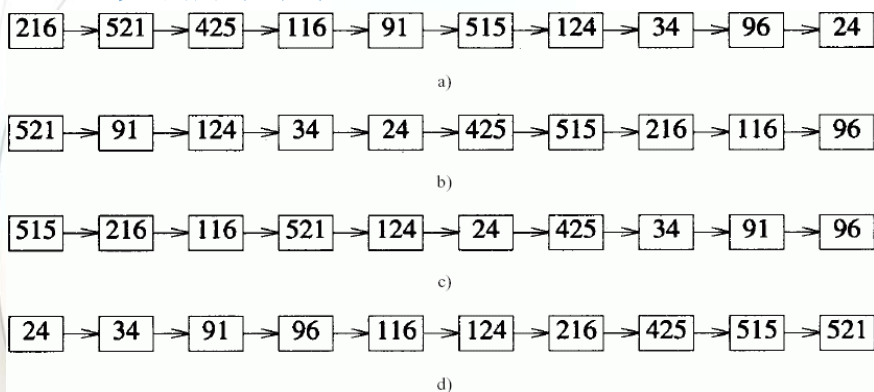


## 基数排序的思想

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

65

## 基数排序例



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

66

## 基数的选定

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

67

## 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...

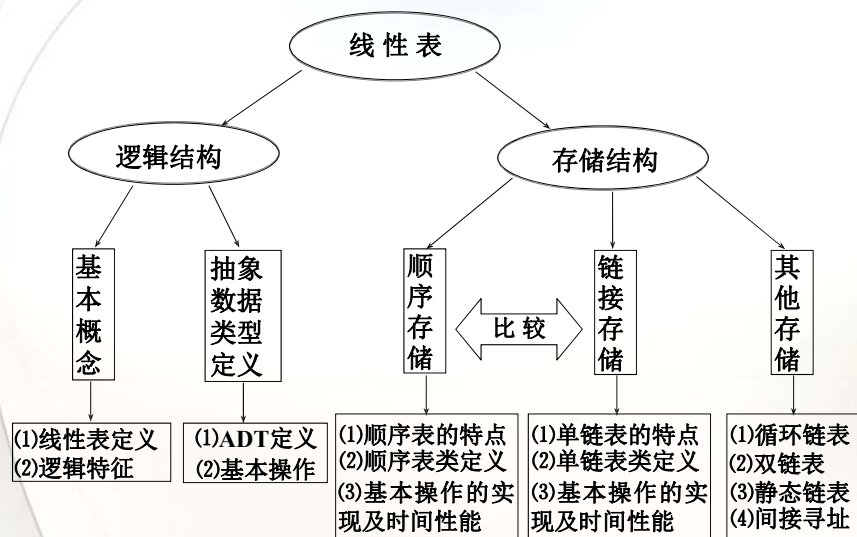
68

## 小结

- 第一种数据结构：线性表
- 三种存储形式：与数据结构无关
  - 顺序存储
  - 链表存储
  - 间接寻址
- 如何将数据结构应用于实际问题

69

## 知识点总结



## 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**126**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？

71

thanks

72