第5章

线性表——数组描述

本章内容

- **5.1 数据对象和数据结构**
- **5.2 线性表数据结构**
- **5.3 数组描述**
- 5.4 vector的描述
- 5.5 在一个数组中实现的多重表
- 5.6 性能测量

5.1 数据对象和数据结构

- **数据对象(Data Object)**
  - 一组**实例或值**的集合
  - 例:
    - boolean = {false, true}
    - digit = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    - letter = {A, B, C, …, Z, a, b, c, …, z}
    - naturalNumber={0,1,2,……}
    - integer={0,±1,±2,±3,……}
    - string = {a, b, …, aa, ab, ac,…}
  - 数据对象：boolean、 digit、 letter、 naturalNumber、 integer、 string

元素

- 数据对象的每个实例：
  - 是一个**原子**(atomic)——不可再分
  - **复合实例**—实例由其他数据对象的实例复合而成。
- **元素(element)**——表示对象实例的单个组成成员。
- 例：
- naturalNumber的实例可以看作是原子，也可以看作是复合实例。
  - naturalNumber的实例675——由digit的实例6,7,5组成

操作或函数

- 任何一个数据对象通常都有一组相关的操作或函数，这些操作或函数可以把对象的某个实例转换成另一个实例，或者转换成另一个对象的实例
  - 整数的“+”、“<”、“取整数的最高有效位”.....
- 数据对象实例以及构成实例的元素之间的关系由数据对象的相关函数（操作）来规定。

数据结构

- **数据结构**（Data Structure）
  - 是一个**数据对象**，同时这个对象的实例以及构成实例的元素之间存在着各种联系(关系)。这些联系(关系)是由**相关的函数（操作）**来规定。
- C++的基本类型：最常用的数据对象及其操作
  - int、bool、char、float……
- 研究数据结构
  - **数据对象（实际上是实例）的描述**
  - **相关函数的具体实现**
  - 数据对象的良好描述可以有效地促进函数的高效实现。

5.2 线性表数据结构

- 线性表定义
  - 线性表（linear list），它的每一个实例都是元素的有序集合，其**实例形式为：$(e_0,e_1,\ldots,e_{n-1})$**，其中n 是有穷自然数。
  - e_i 是表中的元素，i是e_i的索引
  - n是**表的长度**或大小，当$n = 0$ 时，表为空
  - n>0时，e_0是第0个元素或首元素，e_{n-1}是最后一个元素
  - e_0先于e_1, e_1先于e_2,.....如此等等.
- 例：
  - 一个班级学生按姓名字母顺序排列的列表；
  - 按递增次序排列的考试分数表；

线性表操作

- 对于线性表有必要执行的操作：
  - 创建一个线性表。
  - 撤销一个线性表。
  - 确定线性表是否为空。
  - 确定线性表的长度。
  - 按一个给定的索引查找一个元素。
  - 按一个给定的元素查找其索引。
  - 按一个给定的索引删除一个元素。
  - 按一个给定的索引插入一个元素。
  - 从左到右的顺序输出线性表元素。

抽象数据类型ADT

- 抽象数据类型 (Abstract Data Type— ADT)
  - ADT 给出了**实例**及**相关操作**的描述 .
  - **抽象数据类型说明独立于任何描述方法。**
  - **抽象数据类型的描述方法**必须满足**抽象数据类型说明**
  - 使用非形式化语言，**不依赖程序设计语言**

5.2.1线性表的ADT—*LinearList*

抽象数据类型*LinearList*

{

实例

有限个元素的有序集合，实例形式为：$(e_0,e_1,\ldots,e_{n-1})$

操作

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(*index*): 返回表中索引为*index*的元素

indexOf(*x*): 返回表中第一次出现的*x*的索引；如果*x*不在表中，则返回-1

erase(*index*): 删除表中索引为*index*的元素，索引大于*index*的元素其索引值减1

insert(*index*, *x*): 把*x*插入到表中索引为*index*的位置上，索引大于等于*index*的元素其索引值加1

Output(out): 从左到右输出表中元素

}

5.2.2 抽象类—LinearList

```
template <class T>
class linearList
{
 public:
     virtual  ~linearList() {};
     virtual bool empty()   const = 0;
       //如果表为空则返回true，否则返回false
     virtual int size() const = 0;
       //返回表的大小(即表中元素个数)
     virtual T& get(int theIndex) const = 0;
       //返回表中索引为theIndex的元素
     virtual int indexOf(const T& theElement) const = 0;
       //返回表中theElement元素第一次出现时的索引；如果
        theElement不在表中，则返回-1
     virtual void erase(int theIndex) = 0;
       //删除表中索引为theIndex的元素，索引大于theIndex的元素其索
        引值减1
     virtual void insert(int theIndex, T& theElement) = 0;
       //把theElement插入到表中索引为theIndex的位置上，索引大于等
        于theIndex的元素其索引值加1
     virtual void Output(out) const = 0;
       //从左到右插入表中元素到输出流out
}
```

5.3 数组描述

- 数组描述：
  - 又称**公式化描述或顺序存储**
  - 采用**数组**来存储线性表对象的每个实例

- 每个数组单元应该足够大，以便能够容纳数据对象实例中的任意一个元素。

线性表的数组描述

- 线性表实例：$L=(e_0,e_1,\cdots,e_{n-1})$

映射

数组

- 数组描述中，元素在数组中的**位置**用一个**数学公式**来指明。

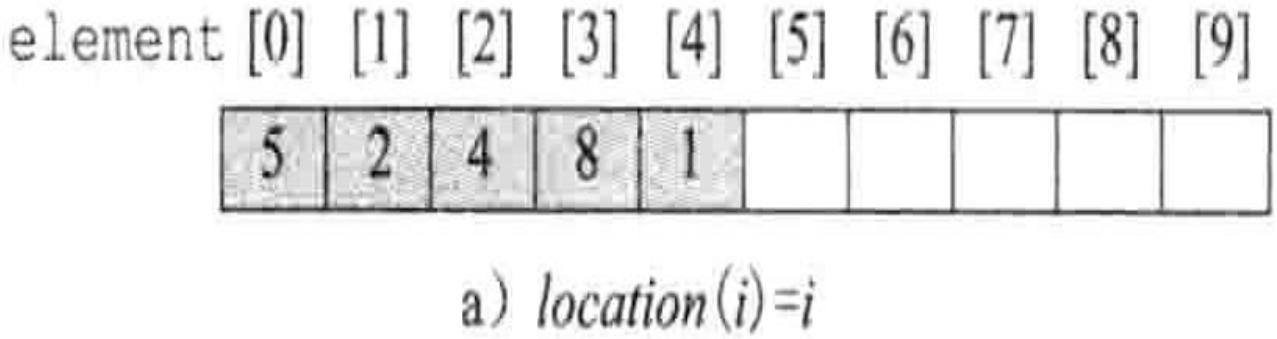
映射公式1

- 典型的映射公式:

  location(i) = i

  →第*i* 个元素（如果存在的话）位于数组中***i***位置处 。

  L=(5,2,4,8,1)



a) *location*(*i*)=*i*

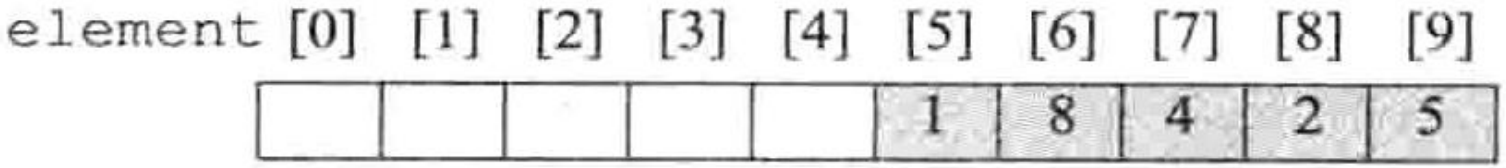
映射公式2

- 映射公式:

  location(i) = arrayLength-i-1

  →第*i* 个元素（如果存在的话）位于数组中arrayLength-i-1位置处 。

  L=(5,2,4,8,1)



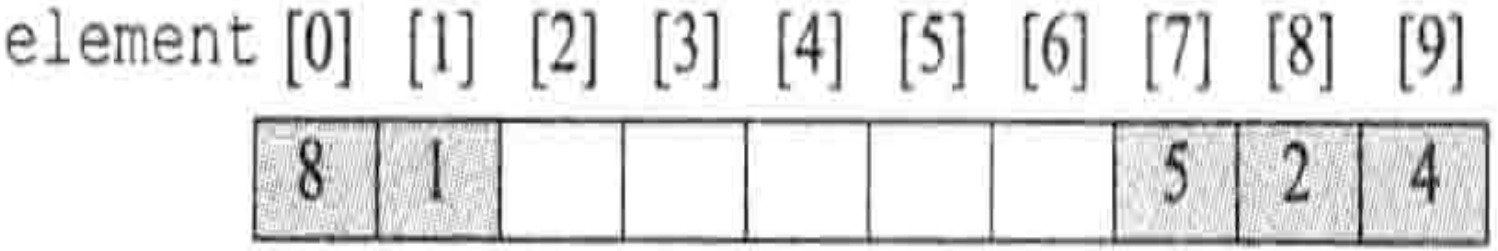
b） *location* (*i*) =9–*i*

映射公式3

- 映射公式:

  location(i) = (location(0) +i)%arrayLength

L=(5,2,4,8,1), location(0)=7



c) $location(i)=(i+7)\%10$

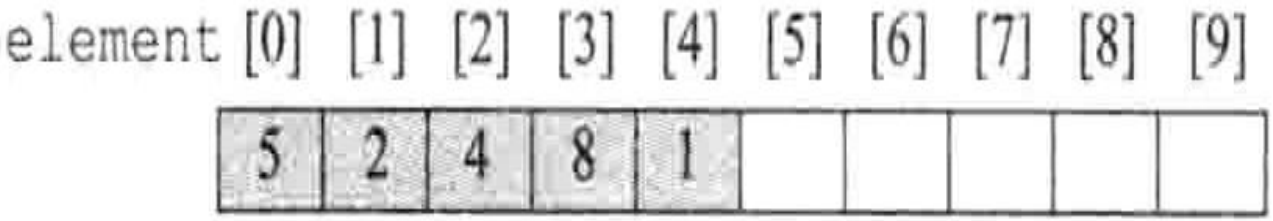
线性表的数组描述

- 映射公式： ***location(i) = i***

  数组：element

  数组长度：arrayLength

  表的长度：listSize



a) *location*(*i*)=*i*

LinearList的实现

- 数组元素类型：模板类
- 随着线性表的变化， arrayLength会变化
- element：变长一维数组
  - arrayLength由用户预估，数组空间变化时，动态增加

实现函数：程序5-2 P97

void changeLength1D(T*& a, int oldLength, int newLength)

linearList的派生类arrayList

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

```
    //其他方法
    int   capacity() const {return arrayLength;}

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

     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
};
```

构造函数 ‘arrayList’

```
template<class T>
arrayList<T>::arrayList(int initialCapacity)
{ //构造函数
  if(initialCapacity<1)
   {ostringstream s;
     s<<“initialCapacity =”<< initialCapacity <<“Must be >0”;
     throw illegalParameterValue(s.str());
    }
  arrayLength = initialCapacity;
  element = new T [arrayLength];
  listSize = 0;
}
```

- 时间复杂性： O(1); 当T是用户自定义类型时，是O(initialCapacity);

复制构造函数 ‘arrayList’

```
template<class T>
arrayList<T>::arrayList(const arrayList<T>& theList)
{  //复制构造函数
   arrayLength = theList.arrayLength;
   listSize = theList. listSize;
  element = new T [arrayLength];
  copy(theList.element, theList.element+ listSize, element);
   }
```

- 时间复杂性： O(n)　　n:复制的线性表的大小

arrayList实例化

- //创建两个容量为100的线性表

  linearList <int> *x=(linearList <int> *) new arrayList <int> (100);

  arrayList <double> y(100);

- //利用容量的缺省值创建一个线性表

  arrayList <char> z;

- //用线性表y复制创建一个线性表

  arrayList <double> w(y);

方法 ‘checkIndex’

- void checkIndex(int theIndex) const;
  - 若索引theIndex无效，则抛出异常

```
template<class T>
void arrayList<T>:: checkIndex(int theIndex) const
{  //确定索引theIndex在0和listSize-1之间
   if (theIndex<0 || theIndex>=listSize)  //索引theIndex无效
    {ostringstream s;
      s<<“index=”<<theIndex<<“size=”<<listSize;
      throw illegalindex(s.str());
    }
}
```

◆时间复杂性： $\Theta(1)$

方法 ‘get’

```
template<class T>
T& arrayList<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)；

    return element[theIndex];
}
```

◆时间复杂性： $\Theta(1)$

方法 ‘indexOf’

```
template<class T>
int arrayList<T>::indexOf(const T& theElement) const
{   //返回元素theElement第一次出现时的索引
   // 如果theElement不存在，则返回-1
//查找元素theElement
int  theIndex=(int) (find(element, element+listSize,
    theElement)-element);
//确定元素theElement是否找到
          if (theIndex ==listSize)  return -1;    //没有找到
     else   return theIndex;
    }
```

- 时间复杂性：O(listSize)

方法 ‘erase’

```
template<class T>
void arrayList<T>::erase(int theIndex)
{//删除表中索引为theIndex的元素，索引大于theIndex的元素其
   索引值减1
  //如果索引为theIndex元素不存在，则抛出异常.
  checkIndex(theIndex);
  //索引大于theIndex的元素向前(左)移动一个位置
   copy(element+theIndex+1, element+listSize,
    element+theIndex);
    element[--listSize].~T();
}
```

- 时间复杂性： O(listSize-theIndex)

方法 ‘insert’

```
template<class T>
void arrayList<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) {……}
 //如果数组已满，则数组长度倍增.
 if (listSize==arrayLength)
   {changeLength1D(element, arrayLength, 2*arrayLength);
    arrayLength*=2;
    }
    //索引大于等于theIndex的元素向后(右)移动一个位置
  copy_backward(element+theIndex, element+listSize;
    element+listSize+1);
    element[theIndex]=theElement;
    listSize++;
}
```

时间复杂性： O(listSize)

方法 ‘output’

```
template<class T>
void arrayList<T>::output(ostream& out) const
{ //把线性表插入输出流
   copy (element, element+listSize,
   ostream_iterator<T>(cout, “  ”));
}
```

- 时间复杂性：Θ (listSize)

方法 ‘output’

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

5.3.4 C++迭代器

- 程序5-9

```
int main ()
{
  int x[3]={0,1,2};
  //用指针y遍历数组x
  for(int* y=x;y!=x+3;y++)
     cout<<*y<<“ ”;
  cout<<endl;
  return 0;
}
```

```
for(int i=0;i!=3;i++)
    cout<<x[i]<<“ ”;
```

- 一个迭代器是一个指针，指向对象的一个元素
- 一个迭代器可以用来逐个访问对象的所有元素

arrayList的迭代器

- 为arrayList定义一个双向迭代器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
  - 前--、后--：迭代器移到上(前)一个元素
  - ==、！=：判断是否相等

arrayList的迭代器

```
class iterator
{
  public:
  //C++的双向迭代器所需要的typedef语句省略(P103)
  //构造函数
   iterator(T* thePosition=0){position=thePosition;}
  //解引用操作符
   T& operator*() const {return *position;}
   T* operator->() const {return &*position;}
  //迭代器的值增加
  iterator & operator++()   //前++
    {++position; return *this;}
  iterator operator++(int)   //后++
    {iterator old=*this; ++position; return old;}
```

arrayList的迭代器

```
//迭代器的值减少
 iterator& operator--()   //前--
    {--position; return *this;}
 iterator operator--(int)   //后--
    {iterator old= *this ;--position; return old;}
//测试是否相等
bool operator!=(const iterator right) const
    {return position!= right.position;}
bool operator==(const iterator right) const
    {return position== right.position;}

Protected:
    T* position;//指向表元素的指针
}
```

arrayList的迭代器

- 为类arrayList增加：
  - begin():返回指向线性表首元素element[0]的指针
  - end()：返回指向线性表尾元素的下一个位置element[listSize]的指针

```
class  iterator;
iterator begin(){return iterator (element);}
iterator end(){return iterator (element+listSize);}
```

- 创建迭代器实例及初始化
  - arrayList<int>:: iterator x=y.begin();

5.4 vector的描述

- vector：
  - STL提供的基于数组的类
  - 具有类arrayList的功能,功能的使用有所不同
- 类vectorList：
  - 使用vector来描述基于数组的线性表

linearList的派生类vectorList

```
template <class T>
class vectorList : public  linearList<T>
{
public:
    //构造函数、复制构造函数、析构函数
    ......
    //ADT方法
    bool empty() const {return element->empty();}
    int size() const {return (int) element->size();}
    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;
    //增加的方法
    int   capacity() const {return element->capacity();}
     ......
protected:
    void checkIndex(int theIndex) const;
    vector<T> *element;   //存储线性表元素的向量
};
```

数组描述优点

- 各种方法可以用非常简单的C++函数来实现。
- 执行搜索、删除和插入的函数都有一个最坏与表的大小呈线性关系的时间复杂性。
- 不需要为表示结点间的逻辑关系而增加额外的存储空间(存储密度高)
- 可以方便的随机存取表中的任一结点。

数组描述缺点

- 插入、删除元素需要移动表内数据。
- 空间需要事先申请，专用，空闲空间不能共享。
- 空间的低效利用。
  - 例：需要维持三个表，在任何时候这三个表所拥有的元素总数都不会超过5000个。然而，很有可能在某个时刻一个表就需要5000个元素，而在另一时刻另一个表也需要5000个元素。
    - 1、采用类arrayList，这三个表中的每一个表都需要有5000个元素的容量。因此，即使我们在任何时刻都不会使用5000以上的元素，也必须为此保留总共15000个元素的空间。
    - 2、采用变长数组，三个表的初始长度小，当一个表的长度由4000增加到5001时，表长倍增，复制元素过程中，源空间长度5000，申请新空间10000，至少需要15000.

在一个数组中实现的多重表

- 在一个数组中实现的多重表比每个表用一个数组来描述空间的利用率更高，但在最坏的情况下，插入操作将耗费更多的时间。