



电子信息学院



武汉大学
Wuhan University

算法与数据结构

(基于现代C++的方法及实践)

ALGORITHM & DATA STRUCTURE IN MODERN C++

第6章 线性表

王文伟 Wang Wenwei, Dr.-Ing.

Tel: 18971562600


Email: wwwang@aliyun.com

课程QQ群: 珞珈EIS数据结构与算法, 668792335


电子信息学院	Table of Contents	武汉大学 Wuhan University
第1章 绪论		
第2章 C++编程基础		
第3章 遍历、迭代与递归		
第4章 字符串		
第5章 排序算法		
第6章 线性表		本章首先学习 线性表逻辑结构 的特性，然后讨论以 顺序存储结构 实现的线性表和以 链式存储结构 实现的线性表的结点结构和各种操作的实现，分析、比较这些不同实现的优缺点。
第7章 栈与队列		
第8章 数组和广义表		
第9章 树和二叉树		
第10章 图		
第11章 查找算法		


本章位置

第6章 线性表

电子信息学院	Table of Contents	武汉大学 Wuhan University	
<hr/>			
6.0 简介			
6.1 线性表的概念及类型定义			
6.2 线性表的顺序存储结构			
6.3 线性链表			

6.0 Introduction

- ◆ **线性表**(linear list)是一种基本的线性数据结构。它的数据元素间具有线性逻辑关系，可以在线性表的任意位置进行插入和删除数据元素的操作。
- ◆ 线性表（逻辑结构）可以用**顺序存储结构**和**链式存储结构**（物理结构）实现(顺序表和链表)。
- ◆ 本章讨论线性表的逻辑结构，以顺序存储结构实现的线性表（顺序表）和以链式存储结构实现的线性表（链表）在结点结构和操作实现方面的特性，分析、比较顺序表和链表的优缺点。

第6章 线性表


4

6.1 线性表的概念及类型定义

- ◆ 线性表是一组具有某种共性的数据元素的有序排列，数据元素之间具有**顺序关系**。除第一个和最后一个元素外，每个元素只有一个**前驱**元素和一个**后继**元素，第一个元素没有前驱，最后一个元素没有后继。
- ◆ 线性表可在任意位置进行插入和删除元素的操作。
- ◆ 线性表中元素的类型可以是**数值型**或**字符串型**，也可以是其他更复杂的**自定义数据类型**。线性表中的数据元素至少具有一种相同的属性，我们称，这些数据元素属于**同一种抽象数据类型**。

6.1.1 抽象数据类型层面的线性表

6.1.2 C++中的线性表类



19

第6章 线性表

5

抽象数据类型的描述方法

ADT 线性表{

- 数据对象D**: 〈数据对象的定义〉
- 数据关系S**: 〈数据关系的定义〉
- 基本操作P**: 〈基本操作的定义〉

} ADT 线性表

其中基本**操作**的定义格式为:

- 基本操作名 (参数表)**
- 初始条件**: 〈初始条件描述〉
- 操作结果**: 〈操作结果描述〉

PL

第6章 线性表

6

6.1.1 抽象数据类型层面的线性表

D、S

- ◆ **线性表的数据元素**: 线性表是由 n 个数据元素组成的有限序列, 记作:
 $LinearList = \{a_0, a_1, a_2, \dots, a_{n-1}\}$
- ◆ **线性表的长度**: = 线性表的元素个数, n
- ◆ 第 i 个数据元素 a_i , 有且仅有一个直接**前驱**数据元素 a_{i-1} 和一个直接**后继**数据元素 a_{i+1}
- ◆ 线性表中的数据元素至少具有一种相同的属性, 属于同一种**抽象数据类型 ADT Node**。
- ◆ 线性表有两种存储结构实现方式: **顺序存储结构**(顺序表 Sequenced List)和**链式存储结构**(链表 Linked List)。

TPL

第6章 线性表

7

线性表的基本操作

P

- ◆ **初始化(Initialize)**: 创建一个线性表实例, 并对该实例进行指定方式的初始化。
- ◆ **访问(Get/Set)**: 对线性表中指定位置的数据元素进行取值或置值等操作。
- ◆ **求长度(Count)**: 求线性表的数据元素个数。
- ◆ **插入(Insert)**: 在指定位置插入新的元素, 插入后仍为一个线性表。
- ◆ **删除(Remove)**: 删除指定位置的元素, 更改后的线性表仍然具有线性表的连续性。
- ◆ **复制(Copy)**: 重新复制一个线性表。

TPL

第6章 线性表

8

线性表的基本操作(II)

- ◆ **合并(Join)**: 将两个或两个以上的线性表合并起来, 形成一个新的线性表。
- ◆ **查找(Search)**: 在线性表中查找满足某种条件的数据元素。
- ◆ **排序(Sort)**: 对线性表中的数据元素按关键字的值, 以递增或递减的次序进行排列。
- ◆ **遍历(Traversal)**: 按次序访问线性表中的所有数据元素, 并且每个数据元素恰好访问一次。

TPL

第6章 线性表

9

6.1.2 C++中的线性表类

- ◆ C++标准库的线性表类模板`vector`和`list`, 线性表实例所包含的元素数目可按需动态增加, 可在表中任意位置进行插入和删除数据元素的操作 (**动态数组**), 一般的数组不具备这种方便的特性。是编程中常用的数据集合类型。

强类型集合: `vector<int>` `vector<string>`

- ◆ **vector**类的成员函数:

- ◆ `vector()`; //创建 `vector`新实例
- ◆ `vector(int initSize)`; 初始化新`vector`实例, 它具有指定的元素个数。
- ◆ `vector(const vector&)`; 复制构造函数。
- ◆ `vector(Iterator begin, Iterator end)`; 初始化新`vector`实例, 它复制另一容器`[begin,end)`区间内的元素。

```
vector<int> v1;  
vector<int> v2(v1);
```

TPL

第6章 线性表

10

vector的公共方法

- ◆ `int size() const`; 返回线性表的长度
- ◆ `int capacity() const`; 能容纳的最大元素数目
- ◆ `const T& operator[](int i) const`; 获取指定索引处的元素
- ◆ `T& operator[] (int i)`; 重载取下标运算符, 设置指定索引处的元素

```
vector<int> v;  
v.push_back(100); v.push_back(5);  
int t = v[0];    v[1] = 100;  
int i = v.size();
```

TPL

第6章 线性表

11

vector的公共方法 (II)

- ◆ `Iterator insert(Iterator it, const T& x)`; 将数据元素`x`插入指定位置
- ◆ `void push_back(const T& x)`; 将元素`x`添加到表尾处
- ◆ `Iterator erase(Iterator first, Iterator last)`; 删除范围`[first,last)`的数据元素
- ◆ `void pop_back()`; 删除表中最后一个元素
- ◆ `void clear()`; 清空表中所有元素

TPL

第6章 线性表

12

声明并构造特定类型的列表举例

```
vector<int> a; 声明并构造int型数列表
a.push_back(86); a.push_back(100);
向列表中添加整型元素
vector<int> nums {0,1,2,3};
vector<string> s;
// 声明并构造字符串列表
s.push_back("Hello"); s.push_back("C++20");
vector<Student> st; // 声明并构造学生列表
st.push_back(Student(8001, "王兵", "男", 18, 92));
```

IPL

第6章 线性表

13

【例6.1】以顺序表求解约瑟夫环问题

- ◆ 约瑟夫（Josephus）环问题：有n个人围坐在一个圆桌周围，把这n个人依次编号为1, ..., n。从编号是s的人开始报数，数到第d个人离席，然后从离席的下一个个人重新开始报数，数到d的人离席，...，如此反复直到最后剩一个人在座位上为止。
- ◆ 例：n=5, s=1, d=2的时候，离席的顺序依次是2, 4, 1, 5，最后留在座位上的是3号。
- ◆ 算法：用有n个元素的线性表，元素表示个人，利用取模运算实现环形位置记录，当某人该出环时，删除表中相应位置的数据元素。

IPL

第6章 线性表

14

```
void Show(const vector<int>& alist) {
    for (const auto& o : alist) {
        cout << o << " ";
    }
    cout << endl;
}
int main() {
    // JosephusRing(50000, 1, 2);
    JosephusRing(5, 1, 2);
    return 0;
}
```

```
void JosephusRing(int n, int s, int d) {
    // n为总人数，从第s个人开始数起，每次数到d。
    int i, j, k; vector<int> aRing;
    for (i = 0; i < n; i++) aRing.push_back(i+1);
    Show(aRing); i = s - 2;
    //第s个人的下标为s-1, i初始指向第s个人的前一位置
    k = n; //每轮的当前人数
    while (k > 1) { //n-1个人依次出环
        j = 0;
        while (j < d) { //将线性表看成环
            j++; i = (i + 1) % k;
        }
        cout << "out: " << aRing[i] << endl;
        aRing.erase(begin(aRing) + i);
        k--; i = (i - 1) % k; Show(aRing);
    }
    cout << aRing[0] << " is the last person" << endl;
}
```

程序运行结果

```
1 2 3 4 5 out: 2
1 3 4 5 out: 4
1 3 5 out: 1
3 5 out: 5
No.3 is the last person.
```

上面这个解决方案的思路是简单直接的，但是算法的实际运行效率非常低。

IPL

第6章 线性表

17

6.2 线性表的顺序存储结构SequencedList

- ◆ 用顺序存储结构实现的线性表称为**顺序表**。
- ◆ 顺序表用一组连续的内存空间顺序存放线性表的数据元素，数据元素在内存空间的物理存储次序与它们在线性表中的逻辑次序是一致的，即元素 a_i 与其前驱元素 a_{i-1} 及后继数据元素 a_{i+1} 无论在逻辑上还是在物理存储上，它们的位置都是相邻的。

下标	元素内容	元素地址
0	a_0	$Loc(a_0)$
1	a_1	$Loc(a_0)+c$
...
i	a_i	$Loc(a_0)+i \times c$
$i+1$	a_{i+1}	...
...
$n-1$	a_{n-1}	$Loc(a_0)+(n-1) \times c$

6.2.1 顺序表的类型定义

6.2.2 顺序表的操作

6.2.3 顺序表操作的算法分析

IPL

第6章 线性表

18

顺序表具有随机访问特性

- ◆ 第*i*个数据元素 a_i 的地址为:

$$\text{Loc}(a_i) = \text{Loc}(a_0) + i \times c, i = 0, 1, 2, \dots, n-1$$

下标	元素内容	元素地址
0	a_0	$\text{Loc}(a_0)$
1	a_1	$\text{Loc}(a_0) + c$
...
i	a_i	$\text{Loc}(a_0) + i \times c$
$i+1$	a_{i+1}	...
...
$n-1$	a_{n-1}	$\text{Loc}(a_0) + (n-1) \times c$

元素的地址是该元素在顺序表中位置(下标)的线性函数,而且每次寻址所花费的时间都是相同的。

IPL

第6章 线性表

19

6.2.1 顺序表的类型定义

```
template <typename T> class SequencedList {
private:
    T* _items;           // 存储数据元素的数组
    int _count;          // 顺序表当前元素个数
    int _capacity;       // 顺序表当前容量
public:  // 其他成员.....
}
```

- ◆ 用SequencedList类(蓝图)来刻画顺序表对象: 当需要使用一个具体的顺序表时, 就创建该类的一个实例, 它表示具体的线性表对象, 通过对这个对象调用类中定义的公共方法来进行相应的操作。
- ◆ 类中数据成员一般设置为私有的(private), 对外是不可见的; 对于客户端, 关心的是该类的功能。面向对象程序设计所要求的类的封装性

IPL

第6章 线性表

20

6.2.2 顺序表的操作

- ◆ 顺序表的初始化
- ◆ 返回顺序表长度
- ◆ 判断顺序表的空与满状态
- ◆ 获取或设置顺序表的容量
- ◆ 获取或设置指定位置的数据元素值
- ◆ 查找
- ◆ 在顺序表的指定位置插入数据元素
- ◆ 删除顺序表指定位置的数据元素

各种操作算法作为SequencedList类的方法成员予以实现

IPL

第6章 线性表

21

顺序表的类图

```
SequencedList<T>
+ _capacity
+ _count
+ _items
+ ~SequencedList()
+ back() const
+ begin() const
+ capacity() const
+ clear()
+ contains(const T &) const
+ count() const
+ data()
+ empty() const
+ end() const
+ front() const
+ full(int) const
+ insert(const T &) const
+ insert(int, const T &)
+ operator[](int)
+ operator=(const SequencedList &)
+ pop_back()
+ push_back(const T &)
+ push_range_back(ITR, int)
+ remove(const T &)
+ removeAt(int)
+ SequencedList(const SequencedList &)
+ SequencedList(int)
+ SequencedList<ITR>(const ITR, int)
+ const_iterator
+ iterator
+ setCapacity(int)
```

1) 顺序表的初始化

- ◆ 使用构造函数创建并初始化顺序表对象:
 - 形式1: 为顺序表实例分配指定大小的空间, 空顺序表。
 - 形式2: 缺省构造方法, 空顺序表。
 - 形式3: 以一个数组的多个元素构造一个线性表。

```
// 构造空顺序表, 分配c个存储单元,
// 不带参数时, 构造具有16个单元的空表
SequencedList(int c = DefaultCapacity) {
    _items = new T[c]; // 分配c个存储单元
    _capacity = c;
    _count = 0; // 此时顺序表长度为0
}
```

使用

```
SequencedList<int> s11;
SequencedList<int> s12(64);
```

➢形式3: 以一个数组(或其他容器)中的元素构造顺序表

```
template<typename ITR>
SequencedList(ITR first, int cnt) {
    _count = cnt > 0 ? cnt : 1; // 元素个数
    _capacity = _count + DefaultCapacity;
    _items = new T[_capacity];
    for (int i = 0; i < cnt; i++)
        _items[i] = *first++;
}
```

使用

```
int[] a = {1, 2, 3, 4, 5};
SequencedList<int> s1(a, 5);
```

构造函数相关特殊成员函数

RAII原则

```
// copy constructor
SequencedList(const SequencedList& a);

// Copy assignment operator
SequencedList& operator=(const SequencedList& rhs);

// move constructor
SequencedList(SequencedList&& sl);

// move assignment operator
SequencedList& operator=(SequencedList&& s);

// destructor
~SequencedList() {delete[] _items;}
```

TPL

第6章 线性表

29

2) 返回顺序表长度

```
int size() const{return _count;}
```

3) 判断顺序表的空状态与满状态

```
int c = al.size();
// 返回表al的元素个数
```

◆ 定义布尔型的成员函数empty()来判断顺序表为空。当_count等于0时，顺序表为空状态，此时empty()返回true，否则返回false。

◆ 定义布尔型的成员函数full()来判断顺序表预分配的空间已满。

◆ 顺序表预分配的存储空间可以而且应该根据需要而调整。

```
bool empty() const {
    return _count == 0;}
```

```
bool full(int cnt=0) const {
    return _count+cnt>=_capacity; }
```

TPL

第6章 线性表

26

4) 获取或设置顺序表的容量

◆ 私有成员_capacity记录顺序表的当前容量，定义公有函数capacity()供外部获取顺序表的容量。

```
int capacity() const{ return _capacity;}
```

◆ 成员函数setCapacity()供外部为顺序表设置新的容量。需要依次进行：

- 重新分配指定大小的存储空间作为顺序表的“数据仓库”，
- 将原数组中的数据元素逐个拷贝到新数组，释放原数组占据的空间。
- 设置_capacity的新值。

TPL

第6章 线性表

27

设置顺序表的容量

```
void setCapacity(int newCapa) {
    T* newspace = new T[newCapa];
    if (_count>newCapa)_count= newCapa;
    for (int i = 0; i < _count; i++)
        newspace[i] = _items[i];
    delete[] _items; // 释放旧存储空间
    _items = newspace;
    _capacity = newCapa; }
```

5) 获取或设置指定位置的数据元素值

◆ 重载“[]”运算符以提供对类的实例进行类似于数组的访问。

```
x = myList[5];
```

```
const T& operator [] (int i) const{
    if(i>=0&&i<_count)return _items[i];
    else throw out_of_range("Index"); }
T& operator [] (int i) {
    if(i >=0&&i<_count)return _items[i];
    else throw out_of_range("Index"); }
```

```
myList[10] = 10.5;
```

TPL

第6章 线性表

29

6) 查找具有特定值的元素

查找线性表是否包含k，查找成功返回true，否则false

```
bool contains(const T& k) const {
    int j = index(k);
    if (j != -1)return true;
    else return false;}
```

查找k在表中的位置，成功，返回k首次出现位置，否则-1

```
int index(const T& k) const {
    int j = 0;
    while(j<count&&items[j]!=k)j++;
    if(j==_count) return -1;
    else return j;}
```

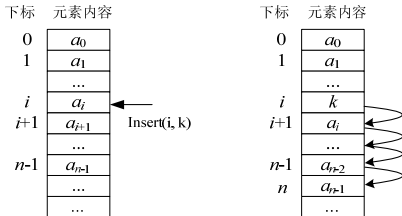
TPL

第6章 线性表

30

7) 在指定位置插入数据元素 insert(i, k)

- 将第 $n-1$ 到第 i 个位置上的数据元素依次向后移动一个位置，空出第 i 个内存单元位置，然后在第 i 个位置上放入给定值 k 。



插入前 插入后
插入一个新的数据元素后，线性表的所有元素仍构成一个线性表

IPL

第6章 线性表

31

在指定位置插入数据元素(II)

```
void insert(int i, const T& k) {
    if(full())setCapacity(2*_capacity);
    if(i<0)i=0; if(i>_count)i=_count;
    if (i<_count) {
        for(int j=_count-1;j>=i;j--)
            _items[j+1] = _items[j];
        _items[i] = k; _count++; return;}
}
```

一种常见的插入操作是在表尾添加push_back一个新元素

```
void push_back(const T& k) {
    if(full())setCapacity(2*_capacity);
    _items[_count] = k; _count++;}
}
```

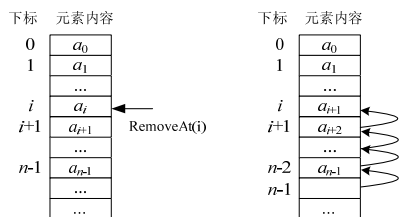
IPL

第6章 线性表

32

8) 删除指定位置的数据元素

- 删除第 i 个元素后要保持线性表的连续性，需将顺序表中原来的第 $i+1$ 到第 $n-1$ 位置上的数据元素依次向前移动。将 a_{i+1} 移动到位置 i 上，实际上就是删除了 a_i 。



删除前

删除后

IPL

第6章 线性表

33

删除指定位置的数据元素(II)

```
void removeAt(int i) {
    if (i>=0 && i<_count) {
        for(int j=i+1;j<_count;j++)
            _items[j-1] = _items[j];
        _count--;}
    else throw out_of_range("Index"); }
```

IPL

第6章 线性表

34

一种常见的删除: 删除首个出现的k值元素

```
void remove(const T& k) {
    int i = index(k); // 查找k值的位置
    if (i != -1) {
        for(int j=i+1; j<_count; j++)
            // 删除第j个值
            _items[j-1] = _items[j];
        _count--;
    }
    else throw out_of_range("值未找到");}
```

IPL

第6章 线性表

35

9) 输出顺序表

str(true)
str()= str(false)

```
ostream& operator<<(ostream& os, const SequencedList<T>& rhs) {
    os << "SequencedList: ";
    if (!rhs.empty())
        for(int i=0;i<rhs.count();i++)os<<rhs[i]<<" ";
    os << endl; return os;}
}
```

```
string str(bool needTypeName = false) const {
    ostringstream ss;
    if (needTypeName)ss<<"SequencedList: ";
    if (!empty())
        for(int i=0;i<_count;i++)ss<<_items[i]<<" ";
    ss << endl; return ss.str();}
```

IPL

第6章 线性表

36

10) 迭代器对象的设计

```
设计 using iterator = T*;
using const_iterator = const T*;
iterator begin() const{return _items;}
iterator end() const{return _items+_count;}

应用 double da[] {9.9, 2.2, 7.7, 8.8, 5.5 };
SequencedList<double> b(da, 5);
for (auto& item: b)
    cout << item << endl;
```

TPL

第6章 线性表

37

6.2.3 顺序表操作的算法分析

- ◆ 时间复杂度为 $O(1)$ 的操作：判断空/满状态，求长度，获取或设置指定位置的元素值等操作。操作实现中所蕴涵的原操作的执行次数都与元素的个数 n 无关。
- ◆ 时间复杂度为 $O(n)$ 的操作：查找给定值，插入和删除数据元素等操作。操作实现所蕴涵的原操作的执行次数都与元素的个数 n 有关。
- ◆ 插入和删除操作所花费的时间主要用于移动元素。设在第 i 个位置插入新的数据的概率为 p_i ，则插入一个数据元素所做的平均移动次数为：

$$\text{平均移动次数} = \sum_{i=0}^n (n-i) \times p_i \quad p_0 = p_1 = \dots = p_n = \frac{1}{n+1}$$
$$C_{\text{Move}} = \sum_{i=0}^n (n-i) \times p_i = \frac{1}{n+1} \sum_{i=0}^n (n-i) = \frac{1}{n+1} \times \frac{n(n+1)}{2} = \frac{n}{2}$$

TPL

第6章 线性表

38

线性表顺序存储结构的特点

- 随机访问特性：存储次序直接反映其逻辑次序，可以直接访问任意位置的数据元素；
- 存储密度高：所有的存储空间都可以用来存放数据元素；
- 插入和删除操作效率不高：每插入或删除一个数据元素，都需要移动大量的数据元素，其平均移动次数是线性表长度的一半；时间复杂度为 $O(n)$ 。
- 需预分配一定大小的内存空间：为顺序表内部数组预分配内存空间时，需要给出内存空间大小，这个数值只能根据不同的情况估算。可能出现因空间估算过大而造成系统内存资源浪费，或因空间估算过小而在随后的某个操作中不得不重新分配存储空间。

TPL

第6章 线性表

39

代码组织：类class，库lib，项目project

- ◆ SequencedList类的源代码文件 SequencedList.h(.cpp)；该类及其他自定义数据结构和算法类都置于名为dsa的类库型项目中。
- ◆ 各章使用这些基础类的测试与应用程序则各自独立定义在相应的应用程序型项目（xxxtest）。
- ◆ 项目设置：1) 添加引用（dsa类库项目）。2) include头文件



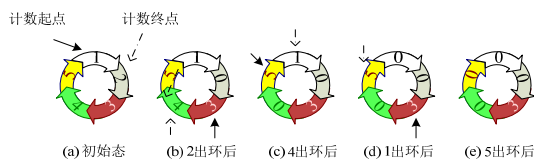
TPL

第6章 线性表

40

【例6.2】以顺序表求解约瑟夫环问题的改进算法

- ◆ 例6.1中的算法多次使用删除操作，即每当一个元素出环时，删除表中相应位置的数据元素，这时必须移动其他元素，操作的时间复杂度高。不使用删除操作，而是将应出环元素相应位置的值设为空值标志，并且在计数时跳过值为空的单元。



TPL

第6章 线性表

41

6.3 线性表的链式存储结构

- ◆ 链式存储结构是指将元素分别存放在一个个链结点（Link Node）中，结点由数据域和链域组成，链指向其后继结点，元素之间的逻辑次序就由结点间的链接关系来实现，逻辑上相邻的结点在物理上不一定相邻。用链式存储结构实现的线性表称为线性链表（linear linked list），简称链表。链表的数据结点个数称为链表的长度。

- ◆ 线性链表根据结点中链的个数分为单向链表和双向链表两种。

6.3.1 线性链表的结点结构

6.3.2 单向链表

6.3.3 单向循环链表

6.3.4 双向链表

```
struct SLNode {
    T item;
    SLNode<T>* next;
    .... }
```

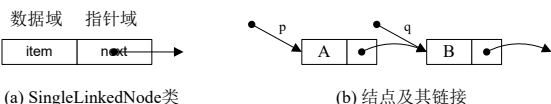
TPL

第6章 线性表

42

6.3.1 线性链表的结点结构

- 在线性表的链式存储结构中，元素分别存放在一个个**结点**中。结点由**数据域**和**指针域**组成，值域保存数据元素的值，指针域则包含指向其他结点的指针。线性表元素间的逻辑次序由结点间的链接关系表示。



- 定义“**自引用结构/类**”（self referential structure / class）来表示链表的结点结构，自引用结构包含指向同类实例的指针成员变量。

IPL

第6章 线性表

43

声明结点结构(Node Struct): 一种自引用类型

```
template<typename T> struct SLNode {
    T item;           //存放结点值
    SLNode<T>* next; //指向后继结点的指针
    ..... }

```

- 一个**结点**就是SLNode<T>类型的一个**实例**: SLNode first;
- 成员next是指向SLNode<T>类型的指针变量，保存（另一）结点实例的地址，称为**链（link）**。
- 通过用指针类型的链域将多个实例（结点对象）链接起来，就可以实现多种动态的数据结构，如链表、二叉树、图等结构。

IPL

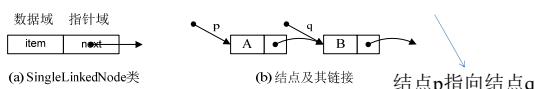
第6章 线性表

44

创建并使用结点对象

SLNode<string> *p, *q; //声明p和q是指向SLNode<string>的指针变量
p= new SLNode<string>(); //创建新实例，由p指向

- 创建和维护动态数据结构需要**动态内存分配**（Dynamic Memory Allocation）。
- C++使用**new**操作符创建对象并为之分配内存。
- 由p引用对象中成员变量的语法为：p->item和p->next。
- 通过下述语句可将p、q两个对象链接起来：p->next = q;



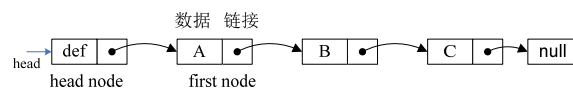
IPL

第6章 线性表

45

6.3.2 单向链表

- 单向链表**（Single Linked List）：每个结点只有一个链的线性链表。单向链表各结点的链指向其后继结点。
- 在单向链表中，从**头指针** head开始找到**头结点**，头结点的链指向第一个数据结点，沿链表的方向前进，就可以顺序访问链表中的每个结点。



IPL

第6章 线性表

46

单向链表的结点类代码

```
struct SLNode {
    T item;           //存放结点值
    SLNode<T>* next; //指向后继结点的指针
    // 构造函数，构造值为k的结点
    SLNode(const T& k):
        item(k), next(nullptr) {}
    // 缺省构造函数，构造缺省值的结点
    SLNode():next(nullptr), item{} {}
    //析构函数
    ~SLNode() {}
};

```

数据成员初始化list

IPL

第6章 线性表

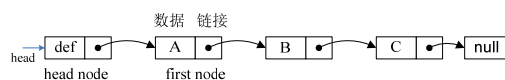
47

单向链表类

```
template <typename T> class SLinkedList{
private:
    SLNode<T>* _head; // _head指向链表头结点
    int _count; // _count记录数据结点的数目
};

```

- SLinkedList类的一个实例（对象）表示一条具体的**单向链表**，类的（实例）成员_head作为链表的**头指针**，指向链表的头结点。若链表为空，则头结点的next域为nullptr，否则指向第一个**数据结点**。



IPL

第6章 线性表

48

单向链表的操作

- ◆ 单向链表的**初始化**，建立单向链表
- ◆ 返回链表的**长度**
- ◆ 判断单向链表**是否为空**
- ◆ 获取或设置**指定位置的数据元素值**
- ◆ **输出**单向链表
- ◆ 在链表的指定位置**插入**数据元素
- ◆ **删除**链表指定位置的数据元素

各种操作作为SLinkedList类的方法成员予以实现

1) 单向链表的创建与初始化和链表的销毁

- ◆ 用SLinkedList类的**构造函数**建立一条链表。

```
SLinkedList(): _count(0) {
    _head = new SNode<T>(); //头结点是个标志结点
}
```

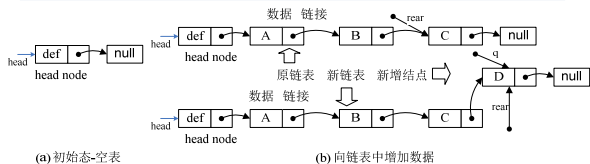
使用 `SLinkedList<int> sll;`

用一个数组的值构造单向链表

```
SLinkedList(const T* first, int cnt);
```

- ◆ 创建结点对象，依次链入表尾

```
q = new SLinkedList<T>(k); //建立结点
rear->next = q; //q结点链入表尾
rear = q; //更新rear, 指向新链尾结点
```



以一个数组的值构造单向链表的代码

```
//以一个数组(或其他容器)中的元素构造顺序表
SLinkedList(const T* first, int cnt) {
    _count = cnt > 0 ? cnt : 1; // 链表长度
    _head = new SNode<T>(); //头结点是个标志结点
    SNode<T>* rear = _head;
    for (int i = 0; i < _count; i++) {
        q = new SNode<T>(first[i]); //建立结点q
        rear->next = q; rear = q; } }
```

使用 `int a[20]; SLinkedList<int> sll(a, 20);`

构造函数相关特殊成员函数

RAII原则

```
// copy constructor
```

```
SLinkedList(const SLinkedList& a);
```

```
// Copy assignment operator
```

```
SLinkedList& operator=(const SLinkedList& rhs)
```

```
// move constructor
```

```
SLinkedList(SLinkedList && s1);
```

```
// move assignment operator
```

```
SLinkedList & operator=(SLinkedList && s);
```

```
// destructor
```

```
~SLinkedList() {::dispose(_head);}
```

2) 返回链表的长度

```
int count() const{return _count;}
int size() const {
    int n = 0;
    SNode<T>* p = _head->next;
    while (p != nullptr) {
        n++; p = p->next;
    }
    return n;
}
```

假设没有设立专门的成员变量_count记录表中的元素个数

3) 判断单向链表是否为空

- ◆用bool型的函数empty()实现该操作，当头结点的链域_head->next为nullptr时，表示链表为空，empty()应指示true；否则，false。或检测_count是否为零。

```
bool empty() const {  
    return (_head->next==nullptr)  
        && (_count==0);  
}
```

判断单向链表是否已满

- ◆**动态分配结点内存空间**：当有一个新数据元素需要加入链表时，向系统申请一个结点的存储空间，可以认为系统能提供的可用空间是足够大的，因此不必判断链表是否已满。如果空间已用完，系统无法分配新的存储单元，则产生运行时异常。如果需要在链表类型中实现判断链表是否已满的功能，则可以按下列方式实现：

```
bool full() const {return false;}
```

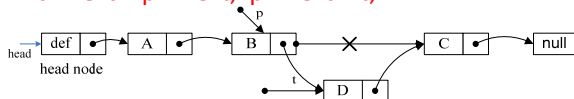
4) 获取或设置指定位置的数据元素值

- ◆以索引参数i来指定结点的位置，则必须从表头顺着链找到相应的结点以获取或设置该结点的值。**链表不具有随机访问特性**

```
const T& operator [] (int i) const {  
    SLNode<T>* p = findNode(i);  
    if (p == nullptr) x = myList[5];  
        throw out_of_range("Index");  
    return p->item; }  
T& operator [] (int i) {  
    SLNode<T>* p = findNode(i);  
    if (p == nullptr)  
        throw out_of_range("Index");  
    return p->item; } myList[10] = 3;
```

6) 插入结点 insert(int i, const T& k)

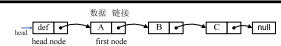
- ◆生成新的结点并插入单向链表中：
SLNode<T>* t = new SLNode<T>(k);
t->next = p->Next; p->next = t;



在单向链表中插入结点，只要修改相关的几条链，而不需移动数据元素。

- ◆如果以索引参数i来指定结点的位置，则必须先从表头顺着链找到相应的结点，再插入新的结点。

5) 输出单向链表



```
void show(bool showTypeName = false) {  
    if (showTypeName) cout << "SLinkedList: ";  
    SLNode<T>* q = _head->next;  
    while (q != nullptr) {  
        cout << q->item << " -> ";  
        q = q->next;  
    }  
    cout << ". " << endl; } a.show( );  
a.show(true);
```

由结点q到达其后继结点: **q = q->next**

输出结果
1) A->B->C->|.
2) SingleLinkedList: A->B->C->|.

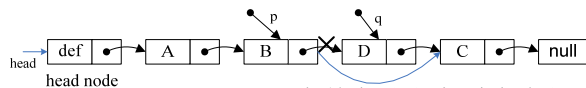
```
void insert(int i, const T& k) {  
    SLNode<T>* p = _head;  
    SLNode<T>* q = p->next;  
    if (i < 0) i = 0; int j = 0;  
    while (q != nullptr) {  
        if (j == i) break;  
        p = q; q = q->next;  
        j++; } // p = findPrevNode(i);  
    SLNode<T>* t = new SLNode<T>(k);  
    t->next = p->next; p->next = t;  
    _count++; }
```

设置p作为q的前驱结点，q每前进一步，p也跟随前进。

7) 删除结点

- ◆ 在单向链表中删除给定位置的结点，需要把该结点从链表中退出，并改变相邻结点的链接关系。在C++中，该结点所占的存储单元必须归还给系统。

- ◆ `remove(const T& k)`和`removeAt(int i)`见讲义。

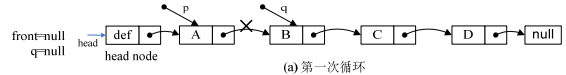


`p->next = q->next;`
`delete q;`

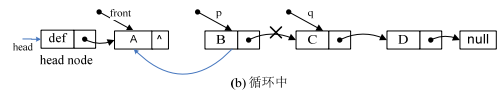
在单向链表中删除结点，只要修改相关的几条链，而不需移动数据。

8) 单向链表逆转reverse

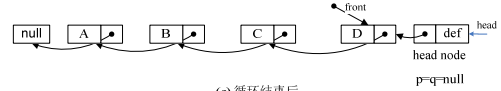
- ◆ 已建立单向链表，将各结点的next链改为指向前驱结点，使得单向链表逆转过来。



(a) 第一次循环



(b) 循环中



(c) 循环结束后

```
void reverse() {
    SLNode<T>* p = _head->next;
    SLNode<T>* q = nullptr;
    SLNode<T>* front = nullptr;
    while (p != nullptr) {
        q = p->next;
        p->next=front; //p->next指向p结点的前驱结点
        front = p;
        p = q; }
    _head->next = front; }
```

【例6.3】单向链表逆转实现与测试

线性表的两种存储结构性能的比较

- ◆ **元素的随机访问特性**
 - 顺序表能够直接访问数据元素。 $O(1)$
 - 链表不能直接访问任意指定位置的数据元素。 $O(n)$
- ◆ **插入和删除操作**
 - 顺序表的插入和删除操作不太方便，有时需要移动大量元素。
 - 链表则容易进行插入和删除操作，只要改动相关结点的链即可，不需移动数据元素。
- ◆ **存储密度**
 - 顺序表存储密度高，全部空间都用来存放数据
 - 链表存储密度较低，结点包含数据和指针

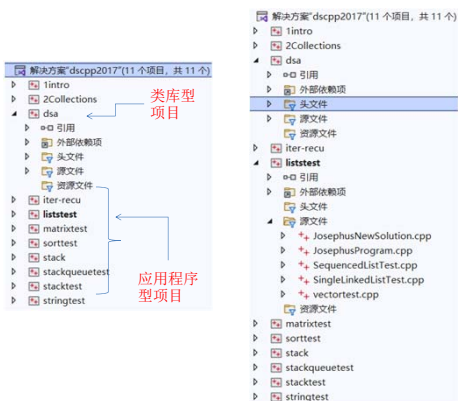
两种存储结构性能的比较(II)

- ◆ **存储空间的动态利用特性**
 - 顺序表不易动态利用存储空间，存在使用空间的浪费和频繁的存储空间重分配问题。
 - 链表向系统动态申请存储单元，也没有数据移动的问题。
- ◆ **查找和排序**
 - 顺序表和链表都可以采用查找和排序的一些简单算法，如顺序查找、插入排序、选择排序等
 - 顺序表还可以采用多种复杂的查找和排序算法，包括折半查找、快速排序、堆排序等

应用中首先关注数据结构的抽象功能

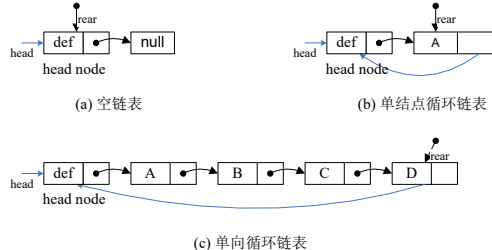
- ◆ 由以上多个操作的算法实现分析可知，顺序表`SequencedList`和链表`SLinkedList`，都实现了“线性表”这个抽象数据结构的基本操作。无论是`SequencedList`类还是`SLinkedList`类，都可以用来建立具体的线性表实例，通过线性表实例调用插入或删除方法进行相应的操作。
- ◆ 一般情况下，解决某个问题关注的是线性表的抽象功能，而不必关注线性表的存储结构及其实现细节。

代码组织：类项目



6.3.3 单向循环链表

◆ 单向循环链表（Circular Linked List）：单向链表中，将最后一个结点的链设置为指向链表的头结点，则该链表成为环状。



IPL

第6章 线性表

68

循环链表及其结点

- ◆ 循环链表的结点与普通的单向链表的结点类型相同。（Single Linked Node）
- ◆ 循环链表类的实现也不必从头设计，我们可以利用面向对象技术，从单向链表类SLinkedList中导出一个新类作为循环链表类的实现。

```
template <typename T>
class CircularLinkedList:public SLinkedList<T>{
    SListNode<T>* rear;
    ..... };

```

IPL

第6章 线性表

69

单向循环链表的特征

- ◆ 循环链表的所有结点链接成一条环路。
- ◆ 仍用head指向头结点，设置变量rear指向循环链表的最后一个结点，则有rear->next==head。rear为尾指针。
- ◆ 当head->next==nullptr或head==rear时，循环链表为空。
- ◆ 当(head->next)->next==head时，循环链表只有一个数据结点。

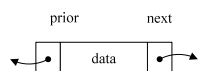
IPL

第6章 线性表

70

6.3.4 双向链表

- ◆ 双向链表(Doubly Linked List): 每个结点有两个（链）成员变量，其中prior指向前驱结点，next指向后继结点。方便实现既向前又向后的操作。



(a) 双向链表中的结点结构



(b) 双向链表

IPL

第6章 线性表

71

双向链表的结点类

- ◆ DLNode<T>类的实例表示双向链表中的结点对象。

```
template <typename T> struct DLNode {
    T item; //存放结点值
    DLNode<T> *prior, *next; //指向前驱与后继结点
    // 构造函数，构造值为k的结点
    DLNode(const T& k):
        item(k), prior(nullptr), next(nullptr) {}
    // 缺省构造函数，构造缺省值的结点
    DLNode(): item{},
        prior(nullptr), next(nullptr) {}
    //析构函数
    ~DLNode() {} };

```

双向链表类

- ◆ `DLinkedList<T>`类的对象表示双向链表

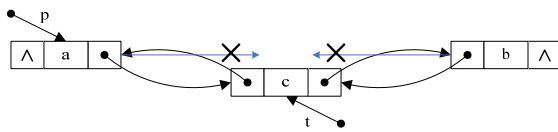
```
template <typename T>
class DLinkedList {
private:
    DLNode<T>* _head; // 指向链表头结点
    int _count; // _count记录数据结点的数目
public:
```

双向链表对象特征

- ◆ 线性表的头结点没有前驱结点，最后一个元素没有后继结点，所以有：
`_head->prior` 等于 `nullptr`
- ◆ 设 `p` 指向双向链表中的某一结点（除尾结点），则双向链表的本质特征如下：
`(p->prior)->next` 等于 `p`
`(p->next)->prior` 等于 `p`
- ◆ 双向链表能够沿着链向两个方向移动，既可以找到后继结点，也可以找到前驱结点。

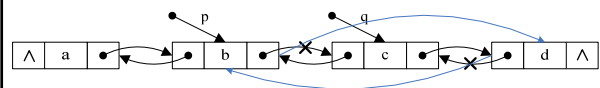
在双向链表中插入结点

- ◆ 在非空链表的 `p` 结点之后插入 `t` 结点，形成新的链表：
`t->prior = p; t->next = p->next;`
`(p->next)->prior = t; p->next = t;`



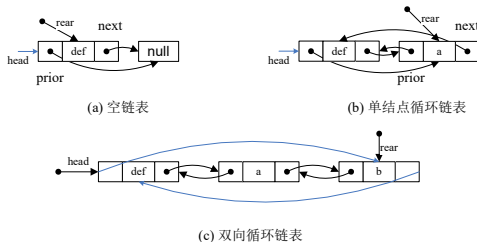
在双向链表中删除结点

- ◆ 删除给定位置的结点，需要把该结点从链表中退出，并改变相邻结点的链接关系。
- ◆ 在双向链表中删除指定位置结点 `q`，设它的前驱结点为 `p`：
`p->next = q->next;`
`(p->next)->prior = p;`



双向循环链表

- ◆ 双向循环链表最后一个结点的 `next` 链指向头结点，而头结点的 `prior` 链指向最后一个结点
- ◆ 即：`rear->next` 等于 `head` 且 `head->prior` 等于 `rear`



本章学习要点

1. 线性表的逻辑结构特性是数据元素之间存在着线性关系，在计算机中表示这种关系的两类不同的存储结构是顺序存储结构和链式存储结构。用前者表示的线性表简称为顺序表，用后者表示的线性表简称为链表。
2. 熟练掌握这两类存储结构的描述方法，以及线性表的各种基本操作的实现。
3. 能够从时间和空间复杂度的角度综合比较线性表两种存储结构的不同特点及其适用场合。

强调：加强编程实验。 You haven't really learned something well until you've taught it to a computer. (Don Knuth)

作业2

6.1 编程实现下列操作。在单向链表中：

1. 单向链表类的移动构造函数，`SLinkedList(SLinkedList&& str);`
2. 返回第*i*个结点的值，`T& GetNodeValue(int i);`
3. 在链表尾部加入数据*k*，`void push_back(const T& k);`
4. 查找值为*k*的节点，`bool contains(const T& k);`
5. 删除值为*x*的节点，`void remove(const T& k);`
- 6.2 分别在SequencedList和SLinkedList类中编程实现获取以字符串表示对象内容的操作：`string str();`

实习

◆ 实验目的

理解线性表的基本概念及其基本操作，熟练运用于编程任务中；熟练运用C++自引用类型的方式实现线性链表的基本操作。熟练掌握Visual Studio进行项目管理。

◆ 题意：编程实现一个不包含起标志作用的头结点的单向链表类。它的头结点是链表的第一个数据结点。提示：一些操作的实现需判断链表是否是单结点的情况。