

王道408 数据结构

第1章 绪论

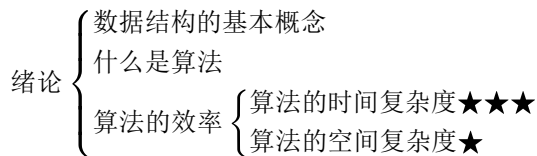
- 1.0 数据结构在学什么
- 1.1 数据结构的基本概念
 - 1.1.1 数据结构的基本概念
 - 1.1.2 数据结构的三要素
 - 1.1.2.1 逻辑结构
 - 1.1.2.2 数据的运算
 - 1.1.2.3 物理结构（存储结构）
 - 1.1.2.4 数据类型、抽象数据类型
- 1.2 算法和算法评价
 - 1.2.1 算法的基本概念
 - 1.2.1.1 什么是算法
 - 1.2.1.2 算法的五个特性
 - 1.2.1.3 “好”算法的特质
 - 1.2.2 算法的时间复杂度
 - 1.2.3 算法的空间复杂度

第2章 线性表

- 2.1 线性表的定义和基本操作
 - 2.1.1 线性表的定义
 - 2.1.2 线性表的基本操作（C++写法）
- 2.2 线性表的顺序表示
 - 2.2.1 顺序表的定义
 - 2.2.2 顺序表的实现
 - 2.2.2.1 静态分配
 - 2.2.2.1.1 静态顺序表结构定义
 - 2.2.2.1.1.1 C语言格式
 - 2.2.2.1.1.2 C++语言格式
 - 2.2.2.1.2 静态顺序表初始化和销毁
 - 2.2.2.1.2.1 C语言格式
 - 2.2.2.1.2.2 C++语言格式
 - 2.2.2.1.3 静态顺序表插入和删除
 - 2.2.2.1.3.1 C语言格式
 - 2.2.2.1.3.2 C++语言格式
 - 2.2.2.1.4 静态顺序表的查找
 - 2.2.2.1.4.1 C语言格式
 - 2.2.2.1.4.2 C++语言格式
 - 2.2.2.1.5 静态顺序表的其他常用函数
 - 2.2.2.1.5.1 C语言格式
 - 2.2.2.1.5.2 C++语言格式
 - 2.2.2.2 动态分配
 - 2.2.2.2.1 动态顺序表结构定义
 - 2.2.2.2.1.1 C语言格式
 - 2.2.2.2.1.2 C++语言格式
 - 2.2.2.2.2 动态顺序表初始化和销毁
 - 2.2.2.2.2.1 C语言格式
 - 2.2.2.2.2.2 C++语言格式
 - 2.2.2.2.3 动态顺序表插入和删除
 - 2.2.2.2.3.1 C语言格式
 - 2.2.2.2.3.2 C++语言格式
 - 2.2.2.2.4 动态顺序表的查找
 - 2.2.2.2.4.1 C语言格式
 - 2.2.2.2.4.2 C++语言格式
 - 2.2.2.2.5 动态顺序表的其他常用函数
 - 2.2.2.2.5.1 C语言格式
 - 2.2.2.2.5.2 C++语言格式
- 2.3 线性表的链式表示

第3章 栈、队列和数组

第1章 绪论



1.0 数据结构在学什么

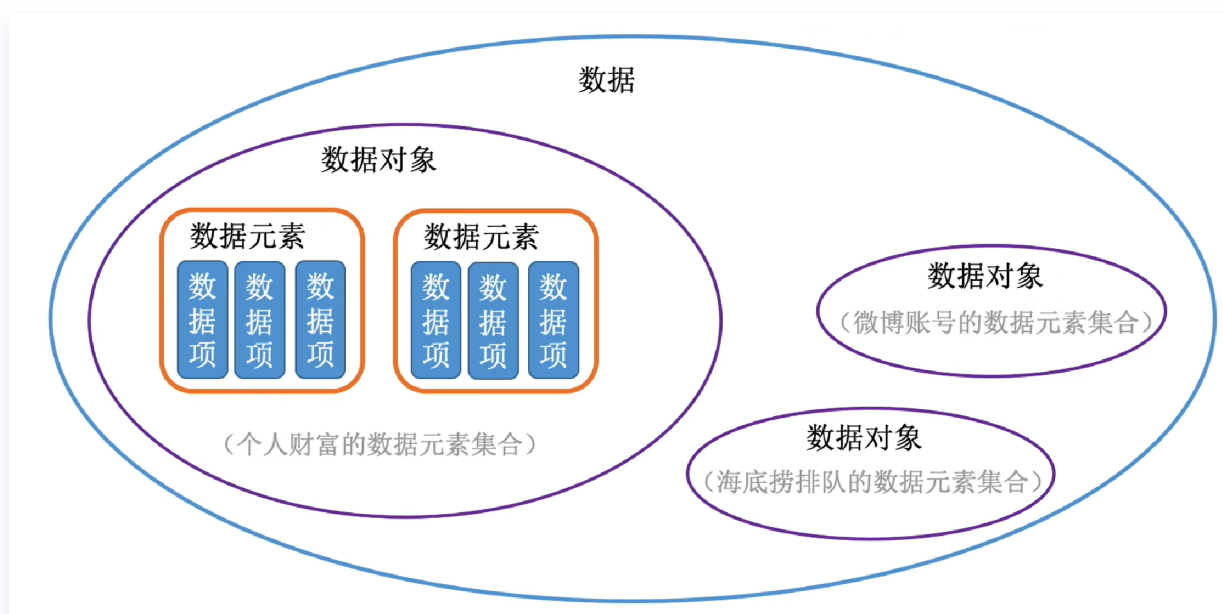
数据结构在学什么

- 学如何用程序代码把现实世界的问题 **信息化**
- 学如何用计算机高效地处理这些信息 **从而创造价值**

1.1 数据结构的基本概念

1.1.1 数据结构的基本概念

- **数据**
 - 数据是 **信息的载体** ,是描述客观事物属性的数、字符及所有能输入到计算机中并被 **计算机程序识别和处理** 的符号的集合。
 - 数据是计算机程序加工的原料。
- **数据元素**
 - **数据元素** 是数据的基本单位, 通常作为一个 **整体** 进行考虑和处理。
- **数据项**
 - 一个数据元素可由若干 **数据项** 组成, 数据项是构成数据元素的 **不可分割的最小单位** 。
- **数据对象**
 - 数据对象是具有 **相同性质** 的数据元素的集合, 是数据的一个子集。



• 数据结构

- 数据结构是相互之间存在一种或多种 **特定关系** 的数据元素的集合。

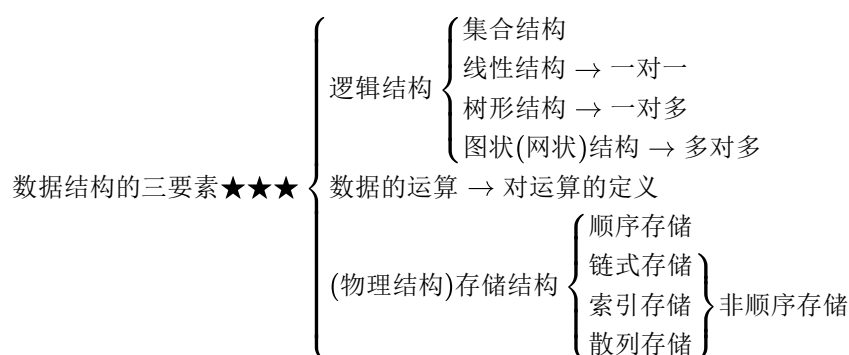
数据对象VS数据结构

数据对象更强调相同性质，而数据结构更强调数据元素之间的关系。

- 相同的数据对象，在不同的条件下，可以组成不同的数据结构。
- 同样的数据元素，可以组成不同的数据结构。
- 不同的数据元素，可以组成相同的数据结构。

数据结构这门课着重关注的是 **数据元素之间的关系**，和对这些数据元素的 **操作**，而不关心具体的数据项内容。

1.1.2 数据结构的三要素



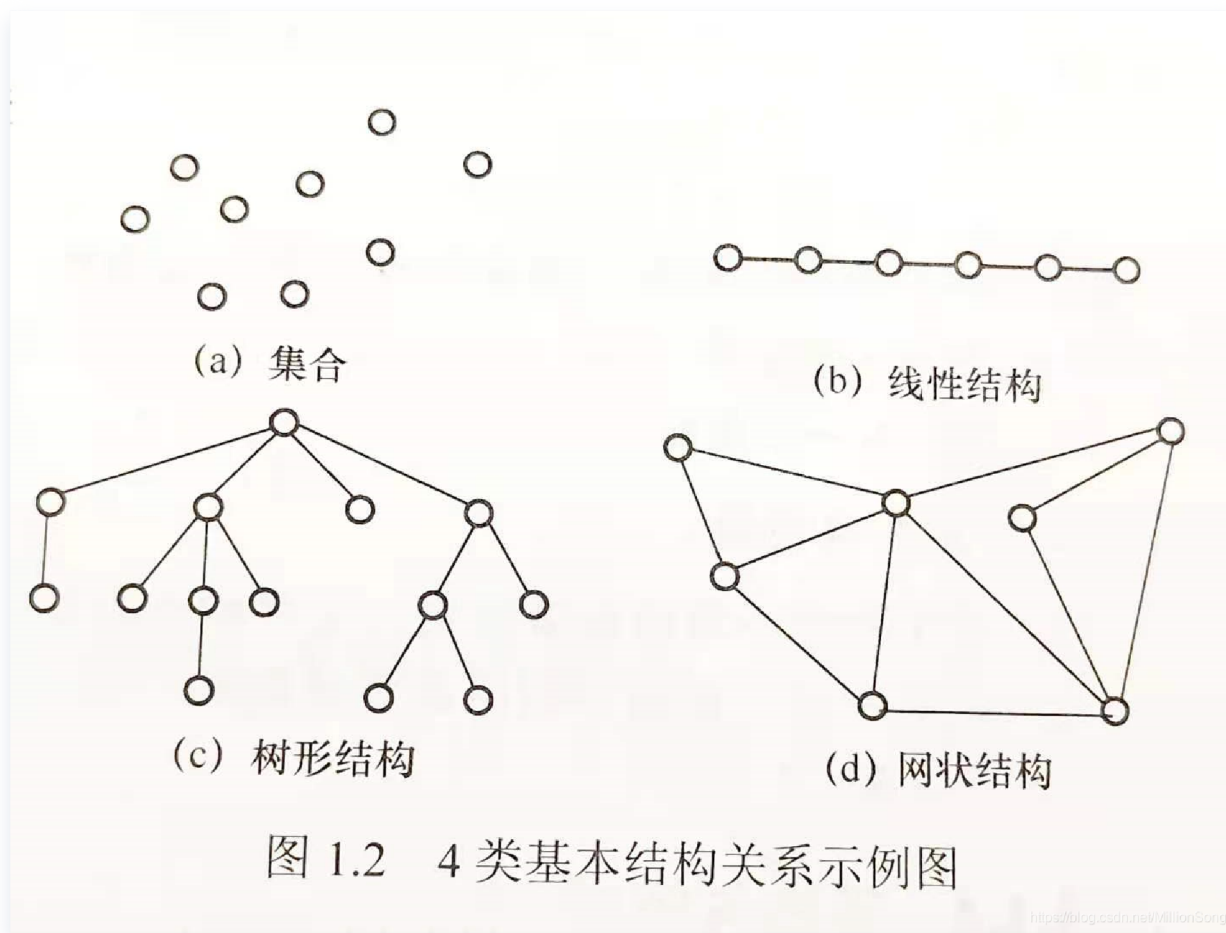
1.1.2.1 逻辑结构

• 逻辑结构

- **集合结构**：各个元素同属一个集合，别无其他关系。
- **线性结构**：数据元素之间是 **一对一** 的关系。
 - 除了第一个元素，所有元素都有 **唯一前驱**；
 - 除了最后一个元素，所有元素都有 **唯一后继**。
- **树形结构**：数据元素之间是 **一对多** 的关系。

- 图状(网状)结构：数据元素之间是 多对多 的关系。

■



1.1.2.2 数据的运算

- 数据的运算：针对于某种逻辑结构，结合实际需求，定义 基本运算
 - 常有的基本运算：增删改查

1.1.2.3 物理结构（存储结构）

- 物理结构（存储结构）：指数据在计算机内存中的组织形式或存储方式。它描述了数据元素之间的逻辑关系和物理存储关系。
 - 顺序存储：把 逻辑上相邻的元素 存储在 物理位置上也相邻 的 存储单元中，元素之间的关系由 存储单元的邻接关系 来体现。
 - 链式存储： 逻辑上相邻的元素在物理上可以不相邻，借助 指示元素存储地址的指针 来表示元素之间的逻辑关系。
 - 索引存储：在存储数据元素信息的同时，还 建立附加的索引表。
 - 索引表中的每项称为 索引项，索引项的一般形式是（关键字，地址）。
 - 散列存储：根据元素的 关键字 直接 计算出该元素的存储地址，又称 哈希(Hash)存储。
 - 第六章中会详细讲解散列存储。
- 链式存储、索引存储、散列存储都是 非顺序存储结构。
- 若采用 顺序存储，则各个数据元素在物理上 必须是连续的；
- 若采用 非顺序存储，则各个数据元素在物理上 可以是离散的。
- 数据的存储结构会
 - 影响存储空间分配的方便程度
 - 影响对数据运算的速度

1.1.2.4 数据类型、抽象数据类型

- 数据类型：是一个 值的集合 和 定义在此集合上的一组操作 的总称。
 - 原子类型：其值 不可再分 的数据类型。
 - 结构类型：其值 可再分解为若干成分(分量) 的数据类型。
 - 抽象数据类型(Abstract Data Type, ADT)：是 抽象数据组织 及 与之相关的操作 。

1.2 算法和算法评价

1.2.1 算法的基本概念

1.2.1.1 什么是算法

- 程序 = 数据结构 + 算法
 - 数据结构： 要处理的信息
 - 算法： 处理信息的步骤
- 算法的定义：算法是 对特定问题求解步骤的一种描述 ，它是指令的有限序列，其中的每条指令表示一个或多个操作。

1.2.1.2 算法的五个特性

- 算法的特性
 - 有穷性：一个算法必须总是在执行有穷步之后结束，且每一步都可在有穷时间内完成。
 - 注：算法是 有穷 的，而程序可以是 无穷 的。
 - 确定性：算法中每条指令必须 有确切的含义 ，对于 相同的输入 只能得出 相同的输出 。
 - 可行性：算法中描述的操作都可以通过 已经实现的基本运算执行有限次 来实现
 - 输入：一个算法 有零个或多个输入 ，这些输入 取自于某个特定的对象的集合 。
 - 输出：一个算法 有一个或多个输出 ，这些输出是 与输入有着某种特定关系的量 。

1.2.1.3 “好”算法的特质

- “好”算法的特质
 - 正确性：算法应能够 正确地解决求解问题 。
 - 可读性：算法应具有 良好 的可读性，以帮助人们理解。
 - 健壮性：输入 非法数据 时，算法能 适当地做出反应或进行处理 ，而不会产生莫名其妙的输出结果。
 - 高效率与低存储量需求：算法效率的度量是通过 时间复杂度 和 空间复杂度 来描述的；
 - 高效率：花的时间少， 时间复杂度低 。
 - 低存储量需求：不费内存， 空间复杂度低 。

1.2.2 算法的时间复杂度

- 算法的时间复杂度：事前预估 算法时间开销 $T(n)$ 与 问题规模 n 的关系。（ T 表示“time”）
- 语句的频度：语句在算法中被重复执行的次数
- 算法中所有语句的 频度之和 记作 $T(n)$ ，即：对应算法问题规模 n 的函数，时间复杂度主要是来分析 $T(n)$ 的数量级；
- 算法的时间复杂度不仅依赖于问题的规模 n ，也取决于待输入的数据的性质（例如：输入元素的初始状态）
- 常见的时间复杂度

- $O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$
- 口诀：常对幂指阶
- 计算时间复杂度的技巧分析
 - 顺序执行的代码只会影响常数项，可以忽略
 - 只需挑循环中的一个基本操作分析它的执行次数与 n 的关系即可
 - 如果有多层嵌套循环，只需关注最深层循环循环了几次
- 计算时间复杂度的技巧总结
 1. 找到一个基本操作（最深层循环）
 2. 分析该基本操作的执行次数 \times 与问题规模 n 的关系 $x = f(n)$
 3. x 的数量级 $O(x)$ 就是算法时间复杂度 $T(n)$ ($T(n) = O(f(n))$)
- 时间复杂度分类
 - 最好时间复杂度：最好的情况下，算法的时间复杂度
 - 最坏时间复杂度：最坏的情况下，算法的时间复杂度
 - 平均时间复杂度：所有可能输入实例在同等概率出现的情况下，算法的期望运行时间
 - 一般情况下，考虑最坏情况的时间复杂度（即：最坏时间复杂度），保证算法的运行时间不会更长
- $O(n)$ 的计算规则
 - 加法规则： $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$
 - 多项相加，只保留最高阶的项，且系数变为1
 - 乘法规则： $O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$
 - 多项相乘，都保留

1.2.3 算法的空间复杂度

- 算法原地工作——算法所需内存空间为 常量

$$\text{计算空间复杂度} \begin{cases} \text{普通程序} \begin{cases} \text{①找到所占空间大小与问题规模相关的变量} \\ \text{②分析所占空间}x\text{与问题规模}n\text{的关系}x = f(n) \\ \text{③}x\text{的数量级}O(x)\text{就是算法空间复杂度}S(n) \end{cases} \\ \text{递归程序} \begin{cases} \text{①找到递归调用的深度}x\text{与问题规模}n\text{的关系}x = f(n) \\ \text{②}x\text{的数量级}O(x)\text{就是算法空间复杂度}S(n) \end{cases} \end{cases}$$

注：有的算法各层函数所需存储空间不同，分析方法略有区别

第2章 线性表

2.1 线性表的定义和基本操作

2.1.1 线性表的定义

- 线性表是具有 相同数据类型 的 $n(n \geq 0)$ 个 数据元素 的 有限序列 ，其中 n 为 表长 ，当 $n=0$ 时线性表是一个 空表 。若用 L 命名线性表，则其一般表示为

$$L = (a_1, a_2, \dots, a_i, a_{i+1}, \dots, a_n)$$

- 每个数据元素所占空间 一样大
- 线性表是 有限 且 有序 的
- 几个概念：
 - a_i 是线性表中的“第 i 个”元素线性表中的 位序
 - a_1 是 表头元素 ； a_n 是 表尾元素
 - 除第一个元素外，每个元素有且仅有一个 直接前驱
 - 除最后一个元素外，每个元素有且仅有一个 直接后继

2.1.2 线性表的基本操作（C++写法）

函数	功能	具体说明	备注
InitList(&L)	初始化表	构造一个空的线性表L,分配内存空间。	从无到有
DestroyList(&L)	销毁操作	销毁线性表，并释放线性表L所占用的内存空间。	从有到无
ListInsert(&Li,e)	插入操作	在表L中的第i个位置上插入指定元素e。	
ListDelete(&L,i,&e)	删除操作	删除表L中第i个位置的元素，并用e返回删除元素的值。	
LocateElem(L,e)	按值查找操作	在表L中查找具有给定关键字值的元素。	
GetElem(L,i)	按位查找操作	获取表L中第i个位置的元素的值	
Length(L)	求表长	返回线性表L的长度，即L中数据元素的个数。	
PrintList(L)	输出操作	按前后顺序输出线性表L的所有元素值。	
Empty(L)	判空操作	若L为空表，则返回true,否则返回false。	

- 对数据的操作（记忆思路）——创销、增删改查
- 函数名和参数的形式、命名都可改变，此处参考了严蔚敏版《数据结构》

2.2 线性表的顺序表示

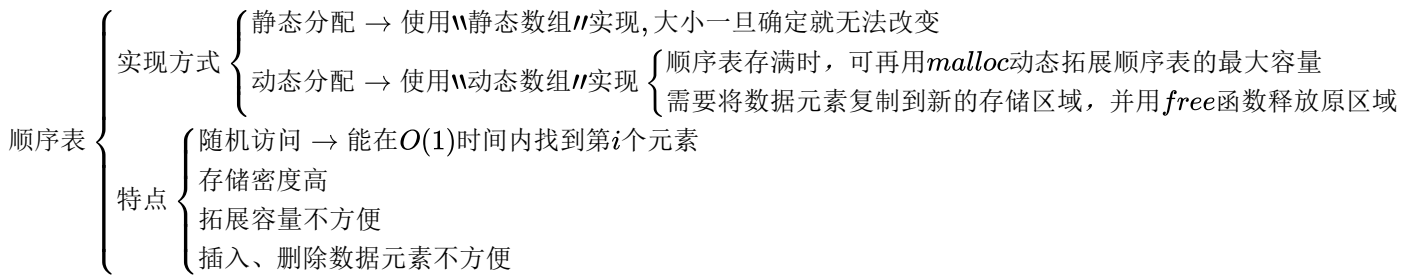
2.2.1 顺序表的定义

- 顺序表——用 顺序存储 的方式实现 线性表顺序存储 ：把 逻辑上相邻的元素 存储在 物理位置上也相邻 的 存储单元 中，元素之间的关系由 存储单元的邻接关系 来体现。
- 设线性表第一个元素的存放位置是 $LOC(L)$ （LOC是location的缩写），则第 i 位元素的地址为：

$$LOC(i) = LOC(L) + i \times \text{数据元素的大小}$$

- c语言通过 `sizeof(ElemType)` 函数知道一个数据元素的大小，其中，ElemType就是顺序表中存放的数据元素类型

2.2.2 顺序表的实现



- 顺序表的 **静态分配** 和 **动态分配** 是两种不同的 **内存管理方式**。
- 静态分配:**
 - 定义:**

在静态分配中, 数组的大小在 **编译** 时确定, 程序在 **运行前** 就会为数组分配 **固定大小的内存空间**。
 - 优点:**
 - 速度较快**: 数组的大小是 **确定** 的, 不需要额外的 **内存分配操作**, 因此访问元素的速度较快。
 - 简单**: 不需要额外的内存管理操作, 代码相对 **简单**。
 - 缺点:**
 - 内存浪费**: 数组的大小是固定的, 可能会浪费内存空间。
 - 大小固定**: 数组的大小在编译时确定, 无法根据实际需要动态调整大小。
- 动态分配:**
 - 定义:**

在动态分配中, 数组的大小是在 **运行** 时 **动态分配** 的, 程序可以根据需要 **动态申请和释放内存空间**。
 - 优点:**
 - 内存利用率高**: 可以根据实际需要动态调整数组的大小, 节省内存空间。
 - 灵活性高**: 数组大小可动态调整, 适应性强。
 - 缺点:**
 - 复杂性**: 需要额外的内存管理操作, 可能增加程序的复杂性。
 - 内存碎片化**: 频繁分配和释放内存可能会导致内存碎片问题。

2.2.2.1 静态分配

2.2.2.1.1 静态顺序表结构定义

2.2.2.1.1.1 C语言格式

```
1  #define MaxSize 10           // 定义最大长度
2  typedef struct
3  {
4      ElemType data[MaxSize];  // 用静态的“数组”存放数据元素
5      int length;              // 顺序表的当前长度
6  }SqList;                     // 顺序表的类型定义（静态分配方式）
```

- 注释:**
- MaxSize** 定义了顺序表的最大长度。
- data** 是一个静态数组, 用于存储顺序表的元素。

- `length` 表示当前顺序表中的元素个数。

2.2.2.1.1.2 C++语言格式

```
1  #include <iostream>
2  #define MaxSize 10           // 定义最大长度
3  using namespace std;
4
5  template<class T>
6  class SqList
7  {
8  private:
9      T data[MaxSize];         // 用静态的“数组”存放数据元素
10     int length;               // 顺序表的当前长度
11 public:
12     SqList();                 // 默认构造函数声明
13     ~SqList();                // 析构函数声明
14 }
```

- 注释:

- `MaxSize` 定义了顺序表的最大长度。
- `data` 是一个静态数组，用于存储顺序表的元素。
- `length` 表示当前顺序表中的元素个数。
- `SqList()` 是默认构造函数，用于初始化顺序表。
- `~SqList()` 是析构函数，用于释放顺序表的内存。

2.2.2.1.2 静态顺序表初始化和销毁

2.2.2.1.2.1 C语言格式

[[#2.2.2.1.1.1 C语言格式|点击查看静态顺序表C语言格式定义]]

```
1  void InitList(SqList *L) {
2      // 可选：初始化每个元素的默认值
3      for (int i = 0; i < MaxSize; ++i)
4      {
5          L->data[i] = defaultValue;
6      }
7      // 初始化顺序表的长度为0
8      L->length = 0;
9  }
10
11 void DestroyList(SqList *L) {
12     // 销毁静态顺序表，释放内存
13     // 将顺序表的长度置为0，表示清空顺序表
14     L->length = 0;
15     // 无需释放内存，静态顺序表的内存是在栈上分配的，自动释放
16 }
17
18 int main() {
19     // 声明一个静态顺序表变量
20     SqList L;
21     // 初始化静态顺序表
```

```

22     InitList(&L);
23     // 在这里可以对静态顺序表进行其他操作
24     DestroyList(&L);
25     return 0;
26 }

```

• 注释:

- `InitList` 函数用于初始化静态顺序表 `L`。
 - 可以选择性地为每个元素赋予默认值。
 - 将顺序表的长度初始化为0。
- 调用 `InitList(&L)` 函数对静态顺序表 `L` 进行初始化。
- `DestroyList` 函数用于销毁静态顺序表 `L`，释放其内存。
 - 将顺序表的长度置为0，表示清空顺序表。
 - 由于静态顺序表的内存是在栈上分配的，因此无需手动释放内存，内存会在函数执行结束时自动释放。
- 在 `main()` 函数中，声明了一个静态顺序表变量 `L`。
- 在 `main()` 函数中，可以对静态顺序表 `L` 进行其他操作。

2.2.2.1.2.2 C++语言格式

[[#2.2.2.1.1.2 C++语言格式|点此查看静态顺序表C++语言格式定义]]

```

1  template<class T>
2  SqList<T>::SqList() {
3      length = 0;           // 初始化顺序表的长度为0
4  }
5
6  template<class T>
7  SqList<T>::~~SqList() {
8      // 析构函数内容
9  }

```

• 注释:

- `SqList()` 是默认构造函数，用于初始化顺序表，将长度初始化为0。
- `~SqList()` 是析构函数，用于清理顺序表的资源。
 - 由于静态顺序表的内存是在栈上分配的，因此无需手动释放内存，内存会在函数执行结束时自动释放，所以该析构函数为空函数。

2.2.2.1.3 静态顺序表插入和删除

2.2.2.1.3.1 C语言格式

[[#2.2.2.1.1.1 C语言格式|点此查看静态顺序表C语言格式定义]]

```

1  bool ListInsert(SqList *L, int i, ElemType e)
2  {
3      // 判断位置合法性
4      if (i < 1 || i > L->length + 1)
5      {
6          printf("插入位置错误\n");
7          return false;

```

```

8     }
9
10    // 判断顺序表是否已满
11    if (L->length ≥ MaxSize)
12    {
13        printf("顺序表已满\n");
14        return false;
15    }
16
17    // 将第i个元素及之后的元素后移
18    for (int j = L->length; j ≥ i; --j)
19    {
20        L->data[j] = L->data[j - 1];
21    }
22
23    // 赋值,表长加一
24    L->data[i - 1] = e;
25    L->length++;
26
27    return true;
28 }
29
30 bool ListDelete(SqList *L, int i, ElemType *e)
31 {
32     // 判断位置合法性
33     if (i < 1 || i > L->length) {
34         printf("删除位置错误\n");
35         return false;
36     }
37
38     // 返回删除的元素值
39     *e = L->data[i - 1];
40
41     // 第i位往后的每一位均前移一位
42     for (int j = i; j < L->length; ++j)
43     {
44         L->data[j - 1] = L->data[j];
45     }
46
47     // 表长减一
48     L->length--;
49     return true;
50 }

```

• 注释:

- `ListInsert` 函数用于在顺序表 `L` 的第 `i` 个位置插入元素 `e`。
- `ListDelete` 函数用于删除顺序表 `L` 的第 `i` 个位置的元素,并将被删除的元素值通过参数 `e` 返回。
- 在插入和删除时,需要进行位置的合法性检查,包括检查插入位置是否合法,顺序表是否已满以及删除位置是否合法。
- 插入时,需要将第 `i` 个位置及其后的元素依次后移,并将元素 `e` 插入到第 `i` 个位置。
- 删除时,将第 `i` 个位置及其后的元素依次前移,同时将被删除的元素值通过参数 `e` 返回。
- 索引 `i` 从1开始。

- 插入和删除的时间复杂度为 $O(n)$ 。

2.2.2.1.3.2 C++语言格式

[[#2.2.2.1.1.2 C++语言格式|点此查看静态顺序表C++语言格式定义]]

```
1  template<class T>
2  bool SqList<T>::ListInsert(int i, const T& e) {
3      if (i < 1 || i > length + 1) {
4          cout << "插入位置错误" << endl;
5          return false;
6      }
7      if (length ≥ MaxSize) {
8          cout << "顺序表已满" << endl;
9          return false;
10     }
11     for (int j = length; j ≥ i; --j) {
12         data[j] = data[j - 1];
13     }
14     data[i - 1] = e;
15     ++length;
16     return true;
17 }
18
19 template<class T>
20 bool SqList<T>::ListDelete(int i, T& e) {
21     if (i < 1 || i > length) {
22         cout << "删除位置错误" << endl;
23         return false;
24     }
25     e = data[i - 1];
26     for (int j = i; j < length; ++j) {
27         data[j - 1] = data[j];
28     }
29     --length;
30     return true;
31 }
```

• 注释:

- `ListInsert` 函数用于在顺序表的第 `i` 个位置插入元素 `e`。
- `ListDelete` 函数用于删除顺序表的第 `i` 个位置的元素，并将被删除的元素值通过引用参数 `e` 返回。
- 在插入和删除时，需要进行位置的合法性检查，包括检查插入位置是否合法，顺序表是否已满以及删除位置是否合法。
- 插入时，需要将第 `i` 个位置及其后的元素依次后移，并将元素 `e` 插入到第 `i` 个位置。
- 删除时，将第 `i` 个位置及其后的元素依次前移，同时将被删除的元素值通过引用参数 `e` 返回。
- 索引 `i` 从1开始。
- 插入和删除的时间复杂度为 $O(n)$ 。

2.2.2.1.4 静态顺序表的查找

2.2.2.1.4.1 C语言格式

[[#2.2.2.1.1.1 C语言格式|点此查看静态顺序表C语言格式定义]]

```
1  int LocateElem(SqList L, ElemType e)
2  {
3      for (int i = 0; i < L.length; ++i)
4      {
5          if (L.data[i] == e) {
6              return i + 1;    // 返回元素在顺序表中的位序
7          }
8      }
9      return 0;    // 表示查找失败
10 }
11
12 ElemType GetElem(SqList L, int i)
13 {
14     if (i < 1 || i > L.length)
15     {
16         printf("位置不合法\n");
17         return ERROR;
18     }
19     return L.data[i - 1];
20 }
```

• 注释:

- `LocateElem` 函数用于按值查找元素 `e` 在顺序表 `L` 中的位置，返回其位序（从1开始）。
- `GetElem` 函数用于按位查找顺序表 `L` 中位序为 `i` 的元素，返回其值。
- 若查找失败，`LocateElem` 返回0，`GetElem` 返回错误标志。

2.2.2.1.4.2 C++语言格式

[[#2.2.2.1.1.2 C++语言格式|点此查看静态顺序表C++语言格式定义]]

```
1  template<class T>
2  int LocateElem(const SqList<T>& L, const T& e)
3  {
4      for (int i = 0; i < L.length; ++i)
5      {
6          if (L.data[i] == e)
7          {
8              return i + 1;    // 返回元素在顺序表中的位序
9          }
10     }
11     return 0;    // 表示查找失败
12 }
13
14 template<class T>
15 T GetElem(const SqList<T>& L, int i)
16 {
17     if (i < 1 || i > L.length)
18     {
```

```

19         cout << "位置不合法" << endl;
20         return ERROR;
21     }
22     return L.data[i - 1];
23 }

```

• 注释:

- `LocateElem` 函数用于按值查找元素 `e` 在顺序表 `L` 中的位置, 返回其位序 (从1开始)。
- `GetElem` 函数用于按位查找顺序表 `L` 中位序为 `i` 的元素, 返回其值。
- 若查找失败, `LocateElem` 返回0, `GetElem` 返回错误标志。

2.2.2.1.5 静态顺序表的其他常用函数

2.2.2.1.5.1 C语言格式

[[#2.2.2.1.1.1 C语言格式|点此查看静态顺序表C语言格式定义]]

```

1  int Length(SqList L)
2  {
3      return L.length;           // 返回顺序表L的长度
4  }
5
6  void PrintList(SqList L)
7  {
8      for (int i = 0; i < L.length; ++i)
9      {
10         cout << L.data[i] << " ";    // 按前后顺序输出顺序表L的所有元素值
11     }
12     cout << endl;
13 }
14
15 bool Empty(SqList L)
16 {
17     return L.length == 0;        // 判断顺序表L是否为空表
18 }

```

• 注释:

- `Length` 函数用于返回顺序表 `L` 的长度, 即其中数据元素的个数。
- `PrintList` 函数用于按前后顺序输出顺序表 `L` 的所有元素值。
- `Empty` 函数用于判断顺序表 `L` 是否为空表, 若为空表则返回 `true`, 否则返回 `false`。

2.2.2.1.5.2 C++语言格式

[[#2.2.2.1.1.2 C++语言格式|点此查看静态顺序表C++语言格式定义]]

```

1  template<class T>
2  int SqList<T>::Length()
3  {
4      return length;           // 返回顺序表的长度
5  }
6
7  template<class T>
8  void SqList<T>::PrintList()

```

```

9   {
10      for (int i = 0; i < length; ++i)
11      {
12          cout << data[i] << " ";          // 按前后顺序输出顺序表的所有元素值
13      }
14      cout << endl;
15  }
16
17  template<class T>
18  bool SqList<T>::Empty()
19  {
20      return length == 0;                  // 判断顺序表是否为空表
21  }

```

- 注释:

- `Length` 成员函数用于返回顺序表的长度，即其中数据元素的个数。
- `PrintList` 成员函数用于按前后顺序输出顺序表的所有元素值。
- `Empty` 成员函数用于判断顺序表是否为空表，若为空表则返回 `true`，否则返回 `false`。

2.2.2.2 动态分配

2.2.2.2.1 动态顺序表结构定义

2.2.2.2.1.1 C语言格式

```

1   typedef struct
2   {
3       ElemType *data;    // 指向动态分配数组的指针
4       int length;        // 顺序表的当前长度
5       int maxSize;       // 顺序表的最大容量
6   } DynamicSqList;

```

- 注释:

- `data` 是一个指向 `ElemType` 类型的指针，用于动态分配存储空间。
- `length` 表示当前顺序表中的元素个数。
- `maxSize` 表示当前动态顺序表的最大容量。

2.2.2.2.1.2 C++语言格式

```

1   #include <iostream>
2   using namespace std;
3
4   template<class T>
5   class DynamicSqList
6   {
7   private:
8       T *data;          // 指向动态分配数组的指针
9       int length;       // 顺序表的当前长度
10      int maxSize;      // 顺序表的最大容量
11  public:
12      DynamicSqList();   // 默认构造函数声明
13      ~DynamicSqList();  // 析构函数声明

```

```
14     };
```

注释:

- `data` 是一个指向模板类型 `T` 的指针，用于动态分配存储空间。
- `length` 表示当前顺序表中的元素个数。
- `maxSize` 表示当前动态顺序表的最大容量。
- `DynamicSqList()` 是默认构造函数，用于初始化顺序表。
- `~DynamicSqList()` 是析构函数，用于释放顺序表的内存。

2.2.2.2.2 动态顺序表初始化和销毁

2.2.2.2.2.1 C语言格式

[[#2.2.2.2.1.1 C语言格式|点击查看动态顺序表C语言格式定义]]

```
1  #include <cstdlib> // 为了使用 malloc 和 free 函数
2  void InitList(DynamicSqList *L)
3  {
4      // 动态分配数组空间
5      L->data = (ElemType *)malloc(MaxSize * sizeof(ElemType));
6      if (L->data == NULL)
7      {
8          // 内存分配失败，进行相应错误处理
9          exit(-1);
10     }
11     L->length = 0;          // 初始化顺序表的长度为0
12     L->maxSize = MaxSize; // 初始化顺序表的最大容量
13 }
14
15 void DestroyList(DynamicSqList *L)
16 {
17     // 释放动态分配的数组空间
18     free(L->data);
19     L->data = NULL; // 避免野指针问题
20     L->length = 0;  // 重置顺序表的长度为0
21     L->maxSize = 0; // 重置顺序表的最大容量为0
22 }
23
24
25 int main()
26 {
27     // 测试初始化函数
28     DynamicSqList list;
29     InitList(&list);
30     DestroyList(&list);
31 }
32
```

• 注释:

- `InitList` 函数用于初始化动态顺序表 `L`。
- `DestroyList` 函数用于销毁动态顺序表 `L`。

- 使用 `malloc()` 函数动态分配数组空间。
 - `malloc()` : `void *malloc(size_t size);`
 - `malloc()` 函数用于在堆上动态分配指定大小的内存空间，并返回一个指向该内存空间的指针。如果分配失败，则返回 `NULL`。
- 使用 `free()` 函数销毁数组空间。
 - `free()` : `void free(void *ptr);`
 - `free()` 函数用于释放之前使用 `malloc()`、`calloc()` 或 `realloc()` 分配的动态内存空间。被释放的内存空间会被归还给系统，并可以被系统重新分配给其他用途。
 - 将顺序表的指针 `data` 置为 `NULL`。
 - 在释放动态分配的内存空间后，将指向该内存空间的指针设置为 `NULL`，可以防止程序继续引用已经释放的内存，避免产生野指针。
- 如果内存分配失败，需要进行相应的错误处理。
- 将顺序表的长度初始化为0，最大容量初始化为 `MaxSize`。

2.2.2.2.2.2 C++语言格式

[[#2.2.2.2.1.2 C++语言格式|点此查看动态顺序表C++语言格式定义]]

```

1  template<class T>
2  DynamicSgList<T>::DynamicSgList()
3  {
4      maxSize = 10;          // 默认初始化为容量为10的顺序表
5      data = new T[maxSize];
6      length = 0;
7  }
8
9  template<class T>
10 DynamicSgList<T>::~~DynamicSgList()
11 {
12     delete[] data;
13 }
```

• 注释:

- `DynamicSgList()` 是默认构造函数，用于初始化顺序表，将容量初始化为10。
 - 默认构造函数是在没有提供任何参数的情况下调用的构造函数。在该情况下，`DynamicSgList()` 会被调用，用于创建顺序表对象并将其容量初始化为10。
- 使用 `new` 关键字动态分配数组空间。
 - `new` 关键字用于在堆上动态分配内存空间。在这里，`new` 被用来分配一个大小为 `maxSize` 的数组空间，并返回指向该空间的指针。
- 使用 `delete` 关键字销毁动态分配的数组空间。
 - `delete` 关键字用于释放之前使用 `new` 关键字分配的动态内存空间。在这里，`delete[]` 被用来释放动态分配的数组空间。
- `~DynamicSgList()` 是析构函数，用于释放顺序表的内存。
 - 析构函数是在对象销毁时自动调用的函数。在这里，`~DynamicSgList()` 被用来释放顺序表对象所分配的内存空间。

2.2.2.2.3 动态顺序表插入和删除

2.2.2.2.3.1 C语言格式

[[#2.2.2.2.1.1 C语言格式|点击查看动态顺序表C语言格式定义]]

```
1  bool ListInsert(DynamicSqlList *L, int i, ElemType e)
2  {
3      // 判断插入位置合法性
4      if (i < 1 || i > L->length + 1)
5      {
6          printf("插入位置不合法\n");
7          return false;
8      }
9
10     // 判断是否需要扩容
11     if (L->length ≥ L->maxSize)
12     {
13         if(!IncreaseSize(L))    // 如果顺序表已满，则增加表最大长度
14         {
15             return false;
16         }
17     }
18
19     // 开始插入操作
20     for (int j = L->length; j > i - 1; j--)
21     {
22         L->data[j] = L->data[j - 1];
23     }
24     L->data[i - 1] = e;
25     L->length++;
26     return true;
27 }
28
29
30 bool ListDelete(DynamicSqlList *L, int i, ElemType *e)
31 {
32     // 判断操作位置合法性
33     if (i < 1 || i > L->length)
34     {
35         printf("删除位置不合法\n");
36         return false;
37     }
38
39     // 返回删除元素的值
40     *e = L->data[i - 1];
41
42     // 开始删除操作
43     for (int j = i; j < L->length; j++)
44     {
45         L->data[j - 1] = L->data[j];
46     }
47     L->length--;
48     return true;
49 }
50
```

```

51 bool IncreaseSize(DynamicSqList *L)
52 {
53     L->maxSize *= 2;        // 表最大长度增加一倍
54     ElemType *newData = (ElemType *)malloc(L->data, L->maxSize * sizeof(ElemType));
55     // 如果分配失败
56     if (!newData)
57     {
58         printf("增加表最大长度失败\n");
59         return false;
60     }
61     // 转移数据到新内存空间
62     memcpy(newData, L->data, L->length * sizeof(ElemType));
63     // 释放旧内存
64     free(L->data);
65     // 指向新内存
66     L->data = newData;
67     return true;
68 }

```

• 注释:

- `ListInsert` 函数用于在顺序表 `L` 的第 `i` 个位置插入元素 `e`。
 - 如果插入位置不合法，则输出提示信息并返回。
 - 如果顺序表已满，则调用 `IncreaseSize` 函数增加表最大长度。
 - 插入元素后，顺序表的长度增加1。
- `ListDelete` 函数用于删除顺序表 `L` 的第 `i` 个位置的元素，并将删除的元素值存放到 `e` 中。
 - 如果删除位置不合法，则输出提示信息并返回。
 - 删除元素后，顺序表的长度减少1。
- `IncreaseSize` 函数用于增加顺序表的最大长度。
 - 将表的最大长度增加为原来的两倍。
 - 使用 `realloc` 函数重新分配内存空间，并将新空间地址赋给 `L->data`。
 - 如果分配失败，则输出提示信息。

2.2.2.2.3.2 C++语言格式

[[#2.2.2.2.1.2 C++语言格式|点击查看动态顺序表C++语言格式定义]]

```

1  template<class T>
2  bool DynamicSqList<T>::ListInsert(int i, T e)
3  {
4      if (i < 1 || i > length + 1)
5      {
6          cout << "插入位置不合法" << endl;
7          return false;
8      }
9      if (length ≥ maxSize)
10     {
11         if(!IncreaseSize()) // 如果顺序表已满，则增加表最大长度
12         {
13             return false;
14         }
15     }

```

```

16     for (int j = length; j > i - 1; j--)
17     {
18         data[j] = data[j - 1];
19     }
20     data[i - 1] = e;
21     length++;
22     return true;
23 }
24
25 template<class T>
26 bool DynamicSgList<T>::ListDelete(int i, T &e)
27 {
28     if (i < 1 || i > length)
29     {
30         cout << "删除位置不合法" << endl;
31         return false;
32     }
33     e = data[i - 1];
34     for (int j = i; j < length; j++)
35     {
36         data[j - 1] = data[j];
37     }
38     length--;
39     return true;
40 }
41
42 template<class T>
43 bool DynamicSgList<T>::IncreaseSize()
44 {
45     maxSize *= 2;          // 表最大长度增加一倍
46     T *newData = new T[maxSize];
47     if (!newData)
48     {
49         cout << "增加表最大长度失败" << endl;
50         return false;
51     }
52     for (int i = 0; i < length; ++i)
53     {
54         newData[i] = data[i];
55     }
56     delete[] data;
57     data = newData;
58     return true;
59 }

```

• 注释:

- `ListInsert` 函数用于在顺序表 `L` 的第 `i` 个位置插入元素 `e`。
 - 如果插入位置不合法，则输出提示信息并返回。
 - 如果顺序表已满，则调用 `IncreaseSize` 函数增加表的最大长度，以确保能够插入新元素。
 - 插入元素后，顺序表的长度增加1。
- `ListDelete` 函数用于删除顺序表 `L` 的第 `i` 个位置的元素，并将删除的元素值存放到 `e` 中。
 - 如果删除位置不合法，则输出提示信息并返回。

- 删除元素后，顺序表的长度减少1。
- `IncreaseSize` 函数用于增加顺序表的最大长度。
- 将表的最大长度增加为原来的两倍。
- 重新分配内存空间，并将原数据复制到新分配的内存空间中。
- 如果分配失败，则输出提示信息。

2.2.2.2.4 动态顺序表的查找

2.2.2.2.4.1 C语言格式

[[#2.2.2.2.1.1 C语言格式|点击查看动态顺序表C语言格式定义]]

```

1  int LocateElem(DynamicSqlList *L, ElemType e)
2  {
3      for (int i = 0; i < L->length; ++i)
4      {
5          if (L->data[i] == e)
6          {
7              return i + 1;
8          }
9      }
10     return 0;    // 未找到元素，返回0
11 }
12
13 ElemType GetElem(DynamicSqlList *L, int i)
14 {
15     if (i < 1 || i > L->length)
16     {
17         printf("位置不合法\n");
18         return ERROR;    // 位置不合法，返回错误值
19     }
20     return L->data[i - 1];
21 }
```

• 注释:

- `LocateElem` 函数用于按值查找顺序表 `L` 中元素 `e`，返回其在顺序表中的位置。
 - 如果找到元素，则返回其在顺序表中的位置（从1开始）。
 - 如果未找到元素，则返回0。
- `GetElem` 函数用于按位查找顺序表 `L` 中第 `i` 个位置的元素。
 - 如果位置不合法，则输出提示信息并返回错误值。
 - 如果位置合法，则返回第 `i` 个位置的元素值。
- 索引 `i` 从1开始。
- 插入和删除的时间复杂度为 $O(n)$ 。

2.2.2.2.4.2 C++语言格式

[[#2.2.2.2.1.2 C++语言格式|点击查看动态顺序表C++语言格式定义]]

```

1  template<class T>
2  int LocateElem(T e)
3  {
```

```

4     for (int i = 0; i < length; ++i)
5     {
6         if (data[i] == e)
7         {
8             return i + 1;
9         }
10    }
11    return 0;    // 未找到元素, 返回0
12 }
13
14 template<class T>
15 T GetElem(int i)
16 {
17     if (i < 1 || i > length)
18     {
19         cout << "位置不合法" << endl;
20         return ERROR;    // 位置不合法, 返回错误值
21     }
22     return data[i - 1];
23 }

```

• 注释:

- `LocateElem` 函数用于按值查找顺序表中元素 `e` , 返回其在顺序表中的位置。
 - 如果找到元素, 则返回其在顺序表中的位置 (从1开始)。
 - 如果未找到元素, 则返回0。
- `GetElem` 函数用于按位查找顺序表中第 `i` 个位置的元素。
 - 如果位置不合法, 则输出提示信息并返回错误值。
 - 如果位置合法, 则返回第 `i` 个位置的元素值。
- 索引 `i` 从1开始。
- 插入和删除的时间复杂度为 $O(n)$ 。

2.2.2.2.5 动态顺序表的其他常用函数

2.2.2.2.5.1 C语言格式

[[#2.2.2.2.1.1 C语言格式|点击查看动态顺序表C语言格式定义]]

```

1     int Length(DynamicSqlList L)
2     {
3         return L.length;    // 返回顺序表L的长度
4     }
5
6     void PrintList(DynamicSqlList L)
7     {
8         for (int i = 0; i < L.length; ++i)
9         {
10            printf("%d ", L.data[i]);
11        }
12        printf("\n");
13    }
14
15    bool Empty(DynamicSqlList L)

```

```

16 {
17     if (L.Length == 0)
18     {
19         return true;    // 若L为空表，则返回true
20     }
21     else
22     {
23         return false;   // 否则返回false
24     }
25 }

```

• 注释：

- `Length(L)` 函数用于返回顺序表 `L` 的长度，即其中数据元素的个数。
- `PrintList(L)` 函数用于按前后顺序输出顺序表 `L` 的所有元素值。
- `Empty(L)` 函数用于判断顺序表 `L` 是否为空表，若为空表则返回 `true`，否则返回 `false`。

2.2.2.2.5.2 C++语言格式

[[#2.2.2.1.2 C++语言格式|点击查看动态顺序表C++语言格式定义]]

```

1  template<class T>
2  int DynamicSgList<T>::Length()
3  {
4      return length;    // 返回顺序表L的长度
5  }
6
7  template<class T>
8  void DynamicSgList<T>::PrintList()
9  {
10     for (int i = 0; i < length; ++i)
11     {
12         cout << data[i] << " ";
13     }
14     cout << endl;
15 }
16
17 template<class T>
18 bool DynamicSgList<T>::Empty()
19 {
20     if (length == 0)
21     {
22         return true;    // 若L为空表，则返回true
23     }
24     else
25     {
26         return false;   // 否则返回false
27     }
28 }

```

• 注释：

- `Length()` 函数用于返回顺序表的长度，即其中数据元素的个数。
- `PrintList()` 函数用于按前后顺序输出顺序表的所有元素值。

- `Empty()` 函数用于判断顺序表是否为空表，若为空表则返回 `true`，否则返回 `false`。

2.3 线性表的链式表示

第3章 栈、队列和数组

第4章 串

第5章 树和二叉树

第6章 图

第7章 查找

第8章 排序