线性表

概述与定义

核心定义

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。

数学表示：若将线性表记为 L，则可表示为：

$$L = (a_1, a_2, \ldots, a_i, \ldots, a_n)$$

其中 a_i 代表线性表中的第 i 个元素。

关键术语：

- 表长：线性表中元素的个数 n。
- 空表：当 $n = 0$ 时，称为空表。
- 位序：i 是数据元素 a_i 在线性表中的位序（注意：在数学定义中位序通常从 1 开始，而在 C++ 数组中下标从 0 开始）。
- 前驱：除第一个元素 a_1 外，每一个元素 a_i 有且仅有一个直接前驱 a_{i-1}。
- 后继：除最后一个元素 a_n 外，每一个元素 a_i 有且仅有一个直接后继 a_{i+1}。

> 线性表是一种逻辑结构，它描述的是数据元素之间“一对一”的线性关系，而不涉及数据在计算机内存中具体是如何存储的。

逻辑特征

线性表在逻辑上具有以下鲜明的特点：

1. 集合的有限性：元素个数 n 是有限的。
2. 有序性：元素之间有先后次序，即 a_1 是第一个，a_2 是第二个...
3. 同质性：所有元素的数据类型相同，这意味着每个元素占用的存储空间大小相同。
4. 抽象性：仅讨论元素间的逻辑关系，不考虑物理实现。

线性表的存储结构分类

1. 顺序存储结构--顺序表

- 实现方式：使用一块连续的内存空间。
- 特点：逻辑上相邻的元素，物理位置也相邻。

2. 链式存储结构--链式表

- 实现方式：使用任意的（非连续）内存空间，通过指针将节点连接。
- 特点：逻辑上相邻，物理位置不一定相邻，靠指针维系关系。

抽象数据类型定义

在 C++ 中，我们通常用类或结构体来封装这些操作。以下是线性表标准 ADT 的伪代码/C++ 风格描述：

ADT LinearList {
数据对象：$D = \{a_i | a_i \in \text{ElemSet}, i = 1, 2, \ldots, n, n \geq 0\}$
数据关系：$R = \{< a_{i-1}, a_i > | a_{i-1}, a_i \in D, i = 2, \ldots, n\}$

} ADT LinearList

基本操作：

(以下参数 `&L` 表示对线性表的引用，会修改原表)

1. InitList(&L):
   - *描述*：初始化操作。
   - *结果*：构造一个空的线性表 L。
2. DestroyList(&L):
   - *描述*：销毁操作。
   - *结果*：销毁线性表，释放内存。
3. ListInsert(&L, i, e):
   - *描述*：插入操作。
   - *结果*：在 L 的第 i 个位置插入新元素 e。表长 n 变为 $n+1$。
4. ListDelete(&L, i, &e):
   - *描述*：删除操作。
   - *结果*：删除 L 中第 i 个位置的元素，并用 e 返回被删除的值。表长 n 变为 $n-1$。
5. GetElem(L, i, &e):
   - *描述*：按位查找。
   - *结果*：获取 L 中第 i 个位置的元素值赋给 e。
6. LocateElem(L, e):
   - *描述*：按值查找。
   - *结果*：在 L 中查找与给定值 e 相等的元素，返回其位序；若不存在则返回 0（或失败标识）。
7. Length(L):
   - *描述*：求表长。
   - *结果*：返回 L 中数据元素的个数。
8. PrintList(L):
   - *描述*：遍历输出。
   - *结果*：按前后顺序输出 L 的所有元素值。

顺序表

定义与概述

顺序表是线性表的顺序存储实现。

- 核心定义：用一组地址连续的存储单元，依次存储线性表中的数据元素。
- 物理特征：
  - 逻辑上相邻的数据元素（如 a_i 和 a_{i+1}），其物理存储位置（内存地址）也是相邻的。
  - 映射关系：逻辑顺序 $=$ 物理顺序。

> 顺序表最核心的特性是“随机访问”。因为物理地址连续，我们可以在 $O(1)$ 的时间复杂度内直接找到第 i 个元素，这与磁带和光盘的区别类似。

顺序表的特点

特性	说明	评价
随机访问	可通过下标直接访问任意位置元素	最大优点 (速度极快)
存储密度高	只需要存储数据本身，不需要存储额外的指针	节省空间
扩展性差	需预先分配一段连续空间，空间满了扩容困难	容易造成空间浪费或溢出
插入/删除慢	插入或删除元素需移动大量后续元素	效率较低 (时间复杂度 O(n))

存储结构模型

在 C++ 中，顺序表通常通过数组来实现。

- 一维数组：用于存储数据元素。
- 连续内存：数组在内存中占据一块完整的、连续的空间。

假设线性表 L 存储的起始位置为 `LOC(A)`，每个元素占用 `sizeof(ElemType)` 个字节。

存储地址计算

假设：

1. 线性表的起始地址为 $LOC(A)$。
2. 每个数据元素占用的存储空间大小为 L 。
3. 我们要计算第 i 个元素的地址。

- 通用定义（下标从 1 开始）：

如果认为第一个元素是 a_1，第二个是 a_2... 则第 i 个元素 a_i 的存储地址公式为：

$$LOC(a_i) = LOC(a_1) + (i-1) \times L$$

要找到第 i 个元素，需要跳过它前面的 $i-1$ 个元素。

- C++ 语言实战（下标从 0 开始）：

在 C++ 中，数组下标 `index` 也就是这里的 i 是从 0 开始的。 即 $a[0]$ 是第 1 个元素，$a[1]$ 是第 2 个元素... 则数组元素 $a[i]$ 的存储地址公式为：

$$LOC(a[i]) = LOC(a[0]) + i \times L$$

这里的 i 直接代表了偏移量。

顺序表的 C++ 定义

静态分配

在编写代码时，就直接指定了数组的大小。数组占据的内存空间是固定的，程序运行期间无法改变。

代码实现：

```cpp
// 顺序表的最大容量
#define MaxSize 50

// 定义表中元素的类型（这里用int，可方便改为其他类型）
typedef int ElemType;

// 顺序表的结构
struct SqList {
    ElemType data[MaxSize]; // 存储元素的数组
    int length;             // 表中当前元素个数
};
```

动态分配

在定义时不直接指定数组大小，而是使用一个指针指向一块内存。程序运行时，根据需要向系统申请内存（堆内存）。如果空间不够，可以申请一块更

大的新区域，将数据搬过去。

代码实现：

```cpp
// 顺序表的初始容量
#define InitSize 100

// 定义表中元素的类型
typedef int ElemType;

// 动态分配的顺序表结构
struct SeqList {
    ElemType *data;     // 指向动态数组首元素的指针
    int MaxSize;        // 当前数组的最大容量（分配的空间大小）
    int length;         // 表中当前元素个数
};
```

如何分配内存： 仅仅定义 struct 只是定义了一个指针，还没有分配实际存放数据的房间。必须在初始化函数中使用 new 或 malloc 来申请内存。

```cpp
// 初始化顺序表（动态分配版本）
void InitList(SeqList &L) {
    L.data = new ElemType[InitSize];  // 动态分配初始空间
    // L.data 指向这块内存的起始地址
    L.length = 0;                     // 初始为空表
    L.MaxSize = InitSize;             // 设置当前最大容量
}
```

两种方式的对比总结

维度	静态分配	动态分配
内存位置	通常在栈上	数据存放在堆上
大小调整	不可调整，编译时定死	可调整，运行时重新申请
适用场景	数据量已知且固定，规模较小	数据量未知，或规模波动大
访问效率	O(1)	O(1)

无论静态还是动态，它们在物理存储上都是连续的，逻辑上都是顺序表。动态分配并不是链表，它依然是一块连续的内存，只是这块内存的大小是可以

更换的。

顺序表的初始化

核心目标

初始化的目的是构造一个空的线性表。对于动态分配的顺序表，具体要做三件事：

1. 申请内存：在堆区开辟一段连续空间。
2. 绑定指针：让结构体中的 data 指针指向这段空间。
3. 重置状态：将当前长度 length 设为 0，设置最大容量 MaxSize。

动态分配的初始化

以下代码是一个完整的、可直接运行的 .cpp 文件结构。它展示了如何定义结构体、如何编写初始化函数，以及如何在 main 函数中调用它。

```cpp
#include <iostream>
#include <cstdlib>  // 用于 exit() 函数
using namespace std;

// 顺序表的初始容量
#define InitSize 10

// 动态分配的顺序表结构
struct SeqList {
    int *data;     // 指向动态数组首元素的指针
    int MaxSize;   // 当前数组的最大容量
    int length;    // 表中当前元素个数
};

// 初始化顺序表
void InitList(SeqList &L) {
    L.data = new int[InitSize];   // 分配初始空间

    // 检查内存分配是否成功
    if (L.data == nullptr) {
        cout << "内存分配失败！" << endl;
        exit(0);  // 终止程序
    }

    L.length = 0;          // 初始为空表
    L.MaxSize = InitSize;  // 设置初始容量
```

```
    cout << "初始化成功！容量: " << L.MaxSize << ", 长度: " << L.length << endl;
}

// 主函数（测试）
int main() {
    SeqList L;  // 声明顺序表变量

    InitList(L);  // 初始化

    // 后续可进行插入、删除等操作...

    return 0;
}
```

静态分配的初始化

使用静态分配，不需要申请内存：

```
// 静态分配的顺序表结构
struct SqList {
    int data[50];    // 固定大小的存储数组
    int length;      // 表中当前元素个数
};

// 初始化静态顺序表
void InitList(SqList &L) {
    L.length = 0;  // 设置长度为0，表示空表
}
```

动态扩容

```
/**
 * 增加动态数组的长度
 * @param L 顺序表
 * @param len 增加的长度
 */
void IncreaseSize(SeqList &L, int len) {
    // 1. 暂存指向原数据的指针
    int *p = L.data;

    // 2. 申请新的、更大的内存空间
    L.data = new int[L.MaxSize + len];

    // 3. 将数据复制过去
    for(int i = 0; i < L.length; i++) {
        L.data[i] = p[i];
    }

    // 4. 修改最大容量
    L.MaxSize = L.MaxSize + len;

    // 5. 释放旧的内存空间
    delete[] p;
}
```

顺序表的基本操作

插入操作

逻辑定义：在顺序表 L 的第 i 个位置插入一个新的元素 e。若 L 的长度为 n，则插入后长度变为 $n+1$。

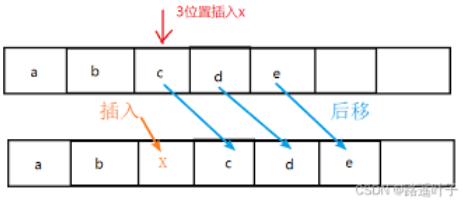
核心难点：因为顺序表的物理空间是连续的，要在一个中间位置“硬塞”进去一个数据，必须把后面的数据依次向后移动，腾出空位。

算法步骤：

1. 判断合法性：检查插入位置 i 是否有效（不能小于0，也不能大于当前长度），以及表是否已满。
2. 挪位：将第 i 个位置及之后的所有元素，从最后一个元素开始，依次向后移动一位（$a_j \to a_{j+1}$）。
3. 填坑：将新元素 e 放入腾出的第 i 个位置。
4. 更新：线性表长度 `length` 加 1。

时间复杂度：

- 最好情况：在表尾插入（即 $i=n$），不需要移动元素。复杂度 $O(1)$。
- 最坏情况：在表头插入（即 $i=0$），所有 n 个元素都要后移。复杂度 $O(n)$。
- 平均情况：平均移动 $n/2$ 个元素。复杂度 $O(n)$。



代码实现：

```
/**
 * 在顺序表的第 i 个位置插入元素 e
 * @param L  顺序表（需要修改）
 * @param i  插入位置（0到表长之间）
 * @param e  要插入的新元素
 * @return   成功返回true，失败返回false
 */
bool ListInsert(SeqList &L, int i, int e) {
    // 1. 检查表是否已满
    if (L.length >= L.MaxSize) {
        return false;
    }

    // 2. 检查插入位置是否合法
    if (i < 0 || i > L.length) {
        return false;
    }

    // 3. 将插入位置后的元素后移（从后往前移动）
    for (int j = L.length; j > i; j--) {
        L.data[j] = L.data[j-1];
    }

    // 4. 放入新元素
    L.data[i] = e;

    // 5. 更新表长
    L.length++;

    return true;
```

```
}
```

删除操作

逻辑定义：删除顺序表 L 中第 i 个位置的元素，并可选地将删除的值返回。删除后长度变为 $n-1$。

核心难点：删除中间的一个元素后，会留下一个“空洞”。为了保持物理上的连续性，必须把后面的数据依次向前移动，填补空位。

算法步骤：

1. 判断合法性：检查删除位置 i 是否有效（范围应在 0 到 $length-1$ 之间）。
2. 取值：将被删除的元素赋值给变量 e 保存。
3. 挪位：将第 i 个位置之后的所有元素，从前向后依次向前移动一位（$a_j \to a_{j-1}$）。
4. 更新：线性表长度 `length` 减 1。

时间复杂度：

- 最好情况：删除表尾元素，不需要移动。复杂度 $O(1)$。
- 最坏情况：删除表头元素，后续 $n-1$ 个元素都要前移。复杂度 $O(n)$。
- 平均情况：平均移动 $(n-1)/2$ 个元素。复杂度 $O(n)$。

代码实现：

```
/**
 * 删除顺序表中第 i 个位置的元素
 * @param L  顺序表（需要修改）
 * @param i  删除位置（0到表长-1之间）
 * @param e  引用参数，用于返回被删除的元素值
 * @return   成功返回true，失败返回false
 */
bool ListDelete(SeqList &L, int i, int &e) {
    // 1. 检查删除位置是否合法
    if (i < 0 || i >= L.length) {
        return false;
    }

    // 2. 保存被删除的元素值
    e = L.data[i];

    // 3. 将删除位置后的元素前移（填补空位）
    for (int j = i; j < L.length - 1; j++) {
        L.data[j] = L.data[j + 1];
    }

    // 4. 更新表长
    L.length--;

    return true;
```

```
}
```

- 插入：合法范围是 `[0, length]`。因为可以插在最后一个元素的后面。
- 删除：合法范围是 `[0, length - 1]`。你不能删除“第 length 个”元素，因为那里是空的。

按位查找

逻辑定义：获取顺序表中第 i 个位置的元素。

核心特征：这是顺序表的最强项。由于地址计算公式 $LOC(a[i]) = LOC(a[0]) + i \times L$，计算机可以直接算出内存地址。

算法步骤：

1. 判断合法性：检查 i 是否越界。
2. 直接访问：返回 `data[i]`。

时间复杂度：

- 始终为 $O(1)$。这被称为随机存取特性。

```
/**
 * 按位查找：获取顺序表中第 i 个位置的元素
 * @param L  顺序表
 * @param i  要查找的位置（0到表长-1之间）
 * @param e  引用参数，用于返回找到的元素值
 * @return   成功返回true，失败返回false
 */
bool GetElem(SeqList L, int i, int &e) {
    // 检查位置是否合法
    if (i < 0 || i >= L.length) {
        return false;
    }

    // 通过下标直接访问元素
    e = L.data[i];

    return true;
}
```

按值查找

逻辑定义：在顺序表中查找第一个值等于 e 的元素，并返回其位序。

核心特征：因为数据是无序存放的，只能挨个对比。

算法步骤：

1. 遍历：从第一个元素开始，依次与 e 比较。
2. 命中：如果发现 `data[i] == e`，则查找成功，返回 i。
3. 失败：如果遍历完整个表都没找到，返回失败标识。

时间复杂度：

- 最好情况：第一个就是，1 次比较。$O(1)$。
- 最坏情况：最后一个才是，或者没找到，比较 n 次。$O(n)$。
- 平均情况：平均比较 $n/2$ 次。复杂度 $O(n)$。

代码实现：

```
```

```
/**
 * 按值查找：查找第一个等于 e 的元素下标
 * @param L  顺序表
 * @param e  要查找的目标值
 * @return   找到返回元素下标，没找到返回 -1
 */
int LocateElem(SeqList L, int e) {
    // 遍历顺序表查找目标值
    for (int i = 0; i < L.length; i++) {
        if (L.data[i] == e) {
            return i;  // 找到返回下标
        }
    }

    return -1;  // 未找到返回 -1
}
```

操作	函数名	时间复杂度	评价
按位查	GetElem	O(1)	极快
按值查	LocateElem	O(n)	较慢

销毁操作

防止内存泄漏

```
// 销毁顺序表，释放动态分配的内存
void DestroyList(SeqList &L) {
    if (L.data != nullptr) {
        delete[] L.data;   // 释放数组空间
        L.data = nullptr;  // 指针置空，防止误用
    }
    L.length = 0;    // 重置表长
    L.MaxSize = 0;   // 重置容量
}
```

遍历操作

验证代码正确性

```
// 打印顺序表的所有元素
void PrintList(const SeqList &L) {
    cout << "顺序表内容: ";
    for (int i = 0; i < L.length; i++) {
        cout << L.data[i] << " ";
    }
    cout << endl; // 换行
}
```

判空与求表长

数据封装

虽然在 struct 里我们可以直接访问 L.length，但在标准的软件工程中，我们不希望外部使用者随意修改 length。通常会提供只读接口 GetLength() 或 IsEmpty()。

完整代码

SeqList_Dynamic.cpp

```cpp
/**
 * 顺序表（动态分配）完整实现与测试
 * 功能：初始化、插入、删除、查找、打印、销毁
 */

#include <iostream>
#include <cstdlib>

using namespace std;

// 1. 定义顺序表结构与常量
#define InitSize 10   // 默认初始容量
typedef int ElemType; // 数据元素类型

struct SeqList {
    ElemType *data;   // 动态数组首指针
    int MaxSize;      // 当前最大容量
    int length;       // 当前元素个数
};

// 2. 函数声明
void InitList(SeqList &L);
void DestroyList(SeqList &L);
void PrintList(const SeqList &L);
bool ListInsert(SeqList &L, int i, ElemType e);
bool ListDelete(SeqList &L, int i, ElemType &e);
bool GetElem(const SeqList &L, int i, ElemType &e);
int LocateElem(const SeqList &L, ElemType e);

// 3. 函数实现
// 初始化顺序表
void InitList(SeqList &L) {
    L.data = new ElemType[InitSize];
    if (L.data == nullptr) {
        cerr << "内存分配失败！" << endl;
        exit(0);
    }
    L.length = 0;
    L.MaxSize = InitSize;
}

// 销毁顺序表
void DestroyList(SeqList &L) {
    if (L.data != nullptr) {
        delete[] L.data;
        L.data = nullptr;
    }
    L.length = 0;
    L.MaxSize = 0;
}

// 打印顺序表
void PrintList(const SeqList &L) {
    if (L.length == 0) {
        cout << "顺序表为空" << endl;
        return;
    }
    cout << "顺序表内容（长度" << L.length << "）：[ ";
    for (int i = 0; i < L.length; i++) {
```

```
        cout << L.data[i] << " ";
    }
    cout << "]" << endl;
}

// 插入元素
bool ListInsert(SeqList &L, int i, ElemType e) {
    if (i < 0 || i > L.length) return false;
    if (L.length >= L.MaxSize) return false;

    for (int j = L.length; j > i; j--) {
        L.data[j] = L.data[j - 1];
    }

    L.data[i] = e;
    L.length++;
    return true;
}

// 删除元素
bool ListDelete(SeqList &L, int i, ElemType &e) {
    if (i < 0 || i >= L.length) return false;

    e = L.data[i];
    for (int j = i; j < L.length - 1; j++) {
        L.data[j] = L.data[j + 1];
    }

    L.length--;
    return true;
}

// 按位查找
bool GetElem(const SeqList &L, int i, ElemType &e) {
    if (i < 0 || i >= L.length) return false;
    e = L.data[i];
    return true;
}

// 按值查找
int LocateElem(const SeqList &L, ElemType e) {
    for (int i = 0; i < L.length; i++) {
        if (L.data[i] == e) return i;
    }
    return -1;
}

// 4. 主函数测试
int main() {
    SeqList L;
    ElemType val;
    int index;

    cout << "=== 顺序表测试 ===" << endl;

    // 初始化测试
    cout << "\n1. 初始化顺序表: " << endl;
    InitList(L);
    PrintList(L);

    // 插入测试
    cout << "\n2. 插入元素测试: " << endl;
    ListInsert(L, 0, 10);
    ListInsert(L, 1, 20);
    ListInsert(L, 2, 30);
```

```
    ListInsert(L, 1, 99);
    PrintList(L);

    // 按位查找测试
    cout << "\n3. 按位查找测试: " << endl;
    if (GetElem(L, 2, val)) {
        cout << "第2个元素是: " << val << endl;
    }

    // 按值查找测试
    cout << "\n4. 按值查找测试: " << endl;
    index = LocateElem(L, 20);
    if (index != -1) {
        cout << "元素20的位置是: " << index << endl;
    }

    // 删除测试
    cout << "\n5. 删除元素测试: " << endl;
    if (ListDelete(L, 1, val)) {
        cout << "删除的元素值是: " << val << endl;
    }
    PrintList(L);

    // 销毁测试
    cout << "\n6. 销毁顺序表: " << endl;
    DestroyList(L);
    cout << "顺序表已销毁" << endl;

    return 0;
}
```

链表

概述与定义

链表 (Linked List) 是线性表的链式存储实现。

- 物理存储：它不要求逻辑上相邻的两个元素在物理地址上也相邻。
- 实现机制：通过“指针”建立数据元素之间的逻辑关系。
- 内存分布：数据元素可以散落在内存的任意角落。

基本组成要素

在链表中，每一个独立的存储单元被称为节点 (Node)。一个完整的节点包含两个部分：

1. 数据域：
- 存放具体的数据元素。
- 对应数学定义中的 a_i。
2. 指针域：
- 存放下一个节点的内存地址。
- 通过它维系 a_i 和 a_{i+1} 的线性关系。

头指针：指向链表中第一个节点的指针。它是链表的入口，必须存在。如果头指针丢失，整个链表的数据就找不到了。

头节点：为了操作方便，在第一个真实数据节点之前附加的一个节点。它通常不存数据（或存链表长度）。不是必须的，但强烈建议使用。

3. 头节点的重要性

引入头节点有两个主要优点：

1. 统一操作：如果没有头节点，插入/删除第一个元素时，需要修改头指针，逻辑与其他位置不同。有了头节点，第 1 个位置的操作和第 i 个位置的操作逻辑完全一致。
2. 统一空表判定：无论表是否为空，头指针始终指向头节点，非空指针。

链表的分类

根据指针的指向和连接方式，链表主要分为三类：

1. 单链表 (Singly Linked List)

- 结构：每个节点只有一个 next 指针，指向后继。
- 特点：只能单向遍历（从头到尾），不能回头。
- 适用：最基础、最常用的形式。

2. 双链表 (Doubly Linked List)

- 结构：每个节点有两个指针，prior 指向前驱，next 指向后继。
- 特点：可以双向遍历。牺牲空间换取时间。

3. 循环链表 (Circular Linked List)

- 结构：最后一个节点的 next 指针不是指向 NULL，而是指向头节点。
- 特点：形成一个环，从表中任一节点出发均可找到表中其他节点。

顺序表与链表的对比

维度	顺序表	链表
内存空间	连续，需预分配，易产生碎片	离散，动态申请，利用率高
访问效率	$O(1)$ 随机访问	$O(n)$ 顺序访问
插入/删除	$O(n)$	$O(1)$
空间开销	小	大
弹性	差	强

单链表

定义与核心结构

单链表是指通过一组任意的存储单元来存储线性表的数据元素。为了建立数据元素之间的线性关系，对每个链表结点，除了存放元素自身的信息外，还

需要存放一个指向其后继的指针。

单向性：节点中只有一个指向下一个节点的指针 `next`。我们可以从头走到尾，但无法从尾退回一步。

节点 (Node) 的构成：

- 数据域 (data)：存放元素值。
- 指针域 (next)：存放下一个节点的地址。

带头节点的链表

这是我们在 C++ 实现中强烈推荐的模式。

- 头指针：是指向链表中第一个节点的指针（变量名为 `L`）。无论链表是否为空，头指针永远指向头节点。
- 头节点：是链表中的第一个物理节点，但不存储有效数据（或者只存储链表长度）。它的作用是作为链表的“哨兵”。
- 首元节点：链表中第一个真正存储数据的节点，即头节点后面的那个节点。

为什么要带头节点?

1. 统一处理：如果不带头节点，插入第 1 个元素时需要修改头指针 `L` 本身；带了头节点后，插入第 1 个元素和插入第 i 个元素的操作逻辑完全一致。
2. 空表统一：无论表是否为空，`L` 始终指向一个有效的内存地址，避免了处理 `NULL` 指针的麻烦。

C++ 结构体定义

```cpp
typedef int ElemType; // ① 类型别名：方便统一修改数据类型

struct LNode {          // ② 链表节点定义
    ElemType data;      // 数据域：存储实际数据
    struct LNode *next;// 指针域：指向下一个节点（自引用结构）
};

typedef LNode *LinkList; // ③ 指针类型别名
```

初始化

构造一个空链表。 具体来说，就是向内存申请一个头节点的空间，并将头节点的 next 指针指向 NULL，表示后面没有数据了。

代码实现：

```cpp
/**
 * @brief 初始化带头节点的单链表
 * @param L 链表头指针的引用
 * @return true 初始化成功
 * @return false 内存分配失败
 */
bool InitList(LinkList &L) {
    // 分配头节点内存
    L = new LNode;

    // 内存分配失败检查
    if (L == nullptr) {
        return false;
    }

    // 头节点next置空，表示空链表
    L->next = nullptr;

    return true;
}
```

单链表的核心操作

操作分类	函数名示例	描述	时间复杂度
创建/建立	`List_HeadInsert`	头插法建立单链表 (逆序)	$O(n)$
创建/建立	`List_TailInsert`	尾插法建立单链表 (正序)	$O(n)$
插入	`ListInsert`	在第 i 个位置插入元素	$O(n)$
删除	`ListDelete`	删除第 i 个位置的元素	$O(n)$
按位查	`GetElem`	获取第 i 个节点的数据	$O(n)$
按值查	`LocateElem`	查找值为 e 的节点位置	$O(n)$
求长	`Length`	统计节点个数	$O(n)$
销毁	`DestroyList`	释放所有节点内存	$O(n)$

1. 头插法

- 逻辑：每次将新结点插入到头节点之后。
- 特点：读入数据的顺序与链表生成的顺序相反（逆序）。
- 应用：常用于链表的逆置。

2. 尾插法

- 逻辑：每次将新结点插入到链表的尾部。
- 关键点：需要定义一个尾指针 (tail pointer) 始终指向当前链表的最后一个节点，避免每次都从头遍历去找尾巴。
- 特点：读入顺序与存储顺序一致（正序）。
- 应用：最常用的建表方式。

3. 按位插入

目标：在第 i 个位置插入元素 e。

核心思路：

- 找前驱：必须找到第 $i-1$ 个节点（设为 p）。
- 新节点：new 一个新节点 s，存入数据 e。
- 断链重连：
  - 第一步：s->next = p->next; (让新节点先拉住后面的节点，防止断链)
  - 第二步：p->next = s; (让前驱节点拉住新节点)

4. 按位删除

目标：删除第 i 个位置的节点。

核心思路：

- 找前驱：必须找到第 $i-1$ 个节点（设为 p）。
- 定位目标：`q = p->next;` (令 q 指向要被删除的第 i 个节点)。
- 断链重连：`p->next = q->next;` (让前驱直接跳过 q，指向 q 的后继)。
- 释放内存：`delete q;` (C++ 必须手动释放，否则内存泄漏)。

5. 按位查找

目标：获取第 i 个节点的数据。逻辑：

- 指针 p 指向首元节点。
- 设置计数器 j=1。
- while(p && j < i)：指针后移 p = p->next，计数器 j++。
- 时间复杂度：$O(n)$。

6. 按值查找

目标：找到数据域等于 e 的节点。逻辑：

- 从头遍历。
- while(p != nullptr && p->data != e)：如果不相等就后移。
- 返回找到的节点指针或位序。
- 时间复杂度：$O(n)$。

7. 求表长

逻辑：设置计数器 len = 0，指针 p 指向头节点。只要 p->next != nullptr，p 就后移，同时 len++。

注意：头节点不计算在长度内。

8. 销毁/清空

逻辑：不能只 delete 头节点，否则后面的节点地址全丢了，造成内存泄漏。

必须使用两个指针（当前指针 p 和 临时指针 q），循环执行：

- q = p; (标记当前节点)
- p = p->next; (先移到下一个安全位置)
- delete q; (炸掉刚才标记的节点)

尾插法建立单链表

尾插法的特点是将新节点每次都插到链表的最后面。

- 输入顺序：1, 2, 3
- 链表顺序：Head -> 1 -> 2 -> 3

代码实现：

```
/**
 * @brief 尾插法建立单链表（正序建立）
 * @param L 链表头指针的引用
 * @return LinkList 返回链表头指针
 */
LinkList List_TailInsert(LinkList &L) {
    int x; // 临时存储输入值

    // 初始化头节点和尾指针
    L = new LNode;
    LNode *r = L;  // r始终指向当前尾节点

    cin >> x;

    while (x != 9999) {
        // 创建新节点并赋值
        LNode *s = new LNode;
        s->data = x;

        // 将新节点连接到尾部并更新尾指针
        r->next = s;
        r = s;
```

```
        cin >> x;
    }

    // 尾节点next置空
    r->next = nullptr;

    return L;
}
```

头插法建立单链表

头插法是指每次将新申请的节点，插入到头节点之后。

- 输入顺序：1, 2, 3
- 链表顺序：Head -> 3 -> 2 -> 1 -> NULL

```
/**
 * @brief 头插法建立单链表（逆序建立）
 * @param L 链表头指针的引用
 * @return LinkList 返回链表头指针
 */
LinkList List_HeadInsert(LinkList &L) {
    int x; // 临时存储输入值

    // 初始化头节点
    L = new LNode;
    L->next = nullptr;

    cin >> x;
    while (x != 9999) {
        // 创建新节点
        LNode *s = new LNode;
        s->data = x;

        // 头插操作：将新节点插入到头节点之后
        s->next = L->next;
        L->next = s;

        cin >> x;
    }

    return L;
}
```

按位插入

在链表 L 的第 i 个位置（从 1 开始计数）插入一个新的元素 e。

插入成功后，原第 i 个节点及其后续节点都会变成新节点的后继。

- 前提：单链表无法像数组那样直接通过下标 `L[i]` 访问。
- 策略：要操作第 i 个位置，必须先找到它的前驱节点（即第 $i-1$ 个节点）。

```
/**
 * @brief 在单链表指定位置插入元素
 * @param L 链表头指针
 * @param i 插入位置（从1开始）
 * @param e 待插入数据
 * @return true 插入成功
```

```
 * @return false 插入失败（位置无效）
 */
bool ListInsert(LinkList L, int i, int e) {
    // 检查位置合法性
    if (i < 1) return false;

    // 寻找第i-1个节点
    LNode *p = L;
    int j = 0;    // 当前节点位置

    while (p != nullptr && j < i - 1) {
        p = p->next;
        j++;
    }

    // 位置越界检查
    if (p == nullptr) return false;

    // 执行插入
    LNode *s = new LNode;
    s->data = e;

    s->next = p->next;
    p->next = s;

    return true;
}
```

按位删除

将单链表 L 中第 i 个位置的节点切断，并释放其占用的内存空间。
通常我们会用一个引用参数 &e 把被删除节点里的数据带出来，以便调用者知道删了什么。

```
/**
 * @brief 删除单链表指定位置的节点
 * @param L 链表头指针
 * @param i 删除位置（从1开始）
 * @param e 引用参数，存储被删除节点的数据
 * @return true 删除成功
 * @return false 删除失败（位置无效）
 */
bool ListDelete(LinkList L, int i, int &e) {
    // 检查位置合法性
    if (i < 1) return false;

    // 查找第i-1个节点
    LNode *p = L;
    int j = 0;

    while (p != nullptr && j < i - 1) {
        p = p->next;
        j++;
    }

    // 检查删除位置是否有效
    if (p == nullptr || p->next == nullptr) return false;

    // 执行删除操作
    LNode *q = p->next;   // q指向待删除节点
    e = q->data;          // 保存被删除节点的数据
    p->next = q->next;    // 跳过待删除节点
    delete q;             // 释放内存
```

```
    return true;
}
```

按位查找

核心定义：取出单链表 L 中第 i 个位置的节点指针或数据。

核心差异：

- 顺序表：支持随机存取，时间复杂度 $O(1)$。
- 单链表：只能顺序存取。想找第 10 个，必须先经过第 1 到第 9 个。时间复杂度 $O(n)$。

```
/**
 * @brief 按位查找节点
 * @param L 链表头指针
 * @param i 目标位置（0表示头节点，1表示第一个数据节点）
 * @return LNode* 第i个节点的指针，不存在则返回nullptr
 */
LNode *GetElem(LinkList L, int i) {
    // 位置合法性检查
    if (i < 0) return nullptr;

    // 遍历查找第i个节点
    LNode *p = L;
    int j = 0;

    while (p != nullptr && j < i) {
        p = p->next;
        j++;
    }

    return p;
}
```

按值查找

核心定义：在单链表中查找数据域等于 e 的节点，返回其指针。

算法步骤：从第一个数据节点（`L->next`）开始，挨个比对 `data`。

```
/**
 * @brief 按值查找节点
 * @param L 链表头指针
 * @param e 目标值
 * @return LNode* 第一个数据域为e的节点指针，未找到返回nullptr
 */
LNode *LocateElem(LinkList L, int e) {
    // 从首元节点开始遍历
    LNode *p = L->next;

    // 查找匹配的节点
    while (p != nullptr && p->data != e) {
        p = p->next;
    }

    return p;
}
```

完整代码

LinkList_Singly.cpp

```cpp
/**
 * @file LinkList_Singly.cpp
 * @brief 带头节点单链表完整实现
 */

#include <iostream>
using namespace std;

// 数据类型定义
typedef int ElemType;

// 节点结构
struct LNode {
    ElemType data;
    struct LNode *next;
};
typedef LNode *LinkList;

// 链表初始化
bool InitList(LinkList &L) {
    L = new LNode;
    if (L == nullptr) return false;
    L->next = nullptr;
    return true;
}

// 尾插法建立链表
LinkList List_TailInsert(LinkList &L) {
    int x;
    LNode *s, *r = L;

    cout << "请输入整数建立链表 (输入 9999 结束): ";
    cin >> x;
    while (x != 9999) {
        s = new LNode;
        s->data = x;
        r->next = s;
        r = s;
        cin >> x;
    }
    r->next = nullptr;
    return L;
}

// 按位插入
bool ListInsert(LinkList L, int i, ElemType e) {
    if (i < 1) return false;

    LNode *p = L;
    int j = 0;
    while (p != nullptr && j < i - 1) {
        p = p->next;
        j++;
    }

    if (p == nullptr) return false;

    LNode *s = new LNode;
    s->data = e;
```

```
    s->next = p->next;
    p->next = s;

    return true;
}

// 按位删除
bool ListDelete(LinkList L, int i, ElemType &e) {
    if (i < 1) return false;

    LNode *p = L;
    int j = 0;
    while (p != nullptr && j < i - 1) {
        p = p->next;
        j++;
    }

    if (p == nullptr || p->next == nullptr) return false;

    LNode *q = p->next;
    e = q->data;
    p->next = q->next;
    delete q;

    return true;
}

// 按位查找
LNode *GetElem(LinkList L, int i) {
    if (i < 0) return nullptr;
    LNode *p = L;
    int j = 0;
    while (p != nullptr && j < i) {
        p = p->next;
        j++;
    }
    return p;
}

// 按值查找
LNode *LocateElem(LinkList L, ElemType e) {
    LNode *p = L->next;
    while (p != nullptr && p->data != e) {
        p = p->next;
    }
    return p;
}

// 求表长
int Length(LinkList L) {
    int len = 0;
    LNode *p = L;
    while (p->next != nullptr) {
        p = p->next;
        len++;
    }
    return len;
}

// 打印链表
void PrintList(LinkList L) {
    LNode *p = L->next;
    cout << "当前链表: Head -> ";
    while (p != nullptr) {
        cout << p->data << " -> ";
```

```
        p = p->next;
    }
    cout << "NULL" << endl;
}

// 销毁链表
void DestroyList(LinkList &L) {
    LNode *p = L;
    LNode *q;
    while (p != nullptr) {
        q = p->next;
        delete p;
        p = q;
    }
    L = nullptr;
}

// 主函数测试
int main() {
    LinkList L;

    // 初始化
    InitList(L);

    // 尾插法建表
    List_TailInsert(L);
    PrintList(L);
    cout << "当前表长: " << Length(L) << endl;

    // 插入测试
    cout << "\n--- 测试：在第 2 个位置插入 99 ---" << endl;
    ListInsert(L, 2, 99);
    PrintList(L);

    // 删除测试
    cout << "\n--- 测试：删除第 3 个位置的元素 ---" << endl;
    int delVal;
    if (ListDelete(L, 3, delVal)) {
        cout << "删除成功，删除的值为: " << delVal << endl;
    } else {
        cout << "删除失败" << endl;
    }
    PrintList(L);

    // 查找测试
    cout << "\n--- 测试：查找 ---" << endl;
    LNode *p = GetElem(L, 2);
    if (p) cout << "第 2 个位置的值是: " << p->data << endl;

    LNode *q = LocateElem(L, 10);
    if (q) cout << "找到了值 10，地址为: " << q << endl;
    else cout << "未找到值 10" << endl;

    // 销毁链表
    cout << "\n--- 销毁链表 ---" << endl;
    DestroyList(L);
    cout << "链表已销毁，程序结束。" << endl;

    return 0;
}
```

双链表

概述与定义

双链表是在单链表的每个结点中，再设置一个指向其前驱结点的指针域。 因此，双链表中的结点包含三个部分：

1. 数据域 (data)：存放数据元素。
2. 前驱指针 (prior)：指向直接前驱节点。
3. 后继指针 (next)：指向直接后继节点。

C++ 结构体定义

```cpp
typedef int ElemType;

// 双向链表节点结构
struct DNode {
    ElemType data;       // 数据域
    struct DNode *prior; // 前驱指针
    struct DNode *next;  // 后继指针
};

typedef DNode *DLinkList;  // 双向链表类型定义
```

逻辑特征与对称性

假设 p 是双链表中除了头尾之外的任意一个节点，那么它一定满足：

$$p \to prior \to next == p == p \to next \to prior$$

解释：p 的“前驱的后继”是 p 自己；p 的“后继的前驱”也是 p 自己。

意义：这种环环相扣的关系，保证了链表的双向连通性。

核心操作实现

插入操作

场景：在节点 p 之后插入一个新节点 s。

核心难点：需要修改 4 个指针。

关键原则：先搞定新节点 s 的连接，再修改老节点 p 的指向

```cpp
/**
 * @brief 在双向链表节点p之后插入新节点s
 * @return true 插入成功
 */
bool InsertNextDNode(DNode *p, DNode *s) {
    if (p == nullptr || s == nullptr) return false;

    // 连接s与p的后继节点
```

```
    s->next = p->next;
    // 如果p有后继节点，更新其后继节点的前驱指针
    if (p->next != nullptr) {
        p->next->prior = s;
    }
    // 连接s与p
    s->prior = p;
    p->next = s;

    return true;
}
```

删除操作

场景：删除节点 p 的后继节点 q（或者直接删除指定节点 q）。

优势：双链表的巨大优势在于，如果我们想删除节点 q，我们不需要像单链表那样费力去查找它的前驱，因为 q->prior 就是它的前驱。

```
/**
 * @brief 删除双向链表中的指定节点p
 * @note p不能是头节点
 */
bool DeleteDNode(DNode *p) {
    if (p == nullptr) return false;

    // 获取p的前驱和后继节点
    DNode *pre = p->prior;
    DNode *post = p->next;

    // 更新前驱节点的next指针
    if (pre != nullptr) {
        pre->next = post;
    }

    // 更新后继节点的prior指针
    if (post != nullptr) {
        post->prior = pre;
    }

    // 释放节点内存
    delete p;

    return true;
}
```

双向遍历

```
// 向后遍历（从当前节点到尾部）
while (p != nullptr) {
    // 处理节点p
    p = p->next;
}

// 向前遍历（从当前节点到头部）
while (p != nullptr) {
    // 处理节点p
    p = p->prior;
}
```

循环链表

概述与定义

循环链表是一种首尾相接的链表。它的特点是表中最后一个节点的指针域不再指向 NULL (nullptr)，而是改为指向头节点。

- 物理形态：整个链表形成一个环。
- 最大特征：链表中没有任何一个节点的指针域是 NULL。
- 遍历特性：从链表中任意一个节点出发，通过后移操作，都可以访问到链表中的所有其他节点（单链表只能从头开始）。

循环单链表

这是最基础的循环结构。

- 结构变化：
  - 普通单链表：尾节点 `r->next = nullptr`。
  - 循环单链表：尾节点 `r->next = L` (L 为头节点)。
- 循环判断条件：
  - 在普通链表中，我们判断是否走到尽头用 `p->next == nullptr`。
  - 在循环链表中，判断条件变为 `p->next == L` 。
- 初始化状态：
  - 空表时，头节点的 `next` 指向它自己。
  - 代码：`L->next = L;`

循环双链表

这是双链表的闭环形态，也是操作系统等系统级软件中非常常用的结构。

- 结构变化：
  - 后继方向：尾节点的 next 指向头节点 L。
  - 前驱方向：头节点的 prior 指向尾节点 r。
- 空表判断：
  - 当表为空时，头节点自己指向自己。
  - 条件：L->next == L && L->prior == L。

C++ 结构体定义

循环链表的节点结构体与普通链表完全一致。区别仅在于初始化和操作逻辑

```cpp
// 循环单链表节点
struct LNode {
    int data;
    LNode *next;
};

// 循环双链表节点
struct DNode {
    int data;
    DNode *prior, *next;
};
```

静态链表

概述与定义

静态链表是指借助数组来描述线性表的链式存储结构。

- 物理结构：数据存储在一段连续的内存空间中。
- 逻辑结构：数据元素之间通过游标连接，体现出“一对一”的线性关系。
- 本质：它是顺序表的物理外壳 + 链表的逻辑内核。

组成要素

在静态链表中，数组的每一个单元包含两个部分：

- 数据域 (data)：用于存储数据元素。
- 游标 (cur/next)：用于存储下一个元素在数组中的下标。

这相当于链表中的 next 指针。

如果 next == -1 (或特殊值)，表示链表结束（类似于 NULL）。

C++ 结构体定义

```cpp
#define MaxSize 50  // 静态链表最大容量

typedef int ElemType;

// 静态链表节点结构
struct Node {
    ElemType data;  // 数据域
    int next;       // 下一个节点的数组下标
};

typedef Node StaticList[MaxSize];  // 静态链表类型
```

线性表的经典应用算法

合并两个有序线性表

这是最基础也是最重要的算法，是归并排序 (Merge Sort) 的核心步骤。

顺序表版本

问题描述：已知顺序表 A 和 B 中的元素有序，假设从小到大，将它们合并为一个新的有序顺序表 C。

核心思想：

- 双指针法：设置指针 `i` 指向 A，`j` 指向 B，`k` 指向 C。
- 竞标赛：比较 `A.data[i]` 和 `B.data[j]`，谁小谁先进入 C，并将对应指针后移。
- 清扫残局：当一个表走完后，将另一个表剩余的所有元素直接复制到 C 中。

C++ 代码实现：

```
bool MergeSeqList(SeqList A, SeqList B, SeqList &C) {
    // 检查C的容量是否足够
    if (C.MaxSize < A.length + B.length) return false;

    int i = 0, j = 0, k = 0;

    // 合并两个有序顺序表
    while (i < A.length && j < B.length) {
        if (A.data[i] <= B.data[j]) {
            C.data[k++] = A.data[i++];
        } else {
            C.data[k++] = B.data[j++];
        }
    }

    // 将剩余元素添加到 C 中
    while (i < A.length) C.data[k++] = A.data[i++];
    while (j < B.length) C.data[k++] = B.data[j++];

    C.length = k;
    return true;
}
```

复杂度：时间 $O(m+n)$，空间 $O(m+n)$。

单链表版本

问题描述：将两个有序单链表（带头节点）合并为一个有序单链表。

优化要求：利用原链表的节点空间，不允许 `new` 新节点，空间复杂度要求 $O(1)$。

核心思想：

- 利用 A 的头节点作为结果链表的头节点。
- 摘下 A 和 B 的节点，像穿针引线一样把它们串起来。

C++ 代码实现：

```
void MergeLinkList(LinkList &LA, LinkList &LB) {
    LNode *p = LA->next;
    LNode *q = LB->next;
    LNode *r = LA; // 使用LA的头节点作为结果链表头

    delete LB; // 释放LB的头节点

    // 合并两个有序链表
```

```
    while (p != nullptr && q != nullptr) {
        if (p->data <= q->data) {
            r->next = p;
            r = p;
            p = p->next;
        } else {
            r->next = q;
            r = q;
            q = q->next;
        }
    }

    // 连接剩余部分
    if (p != nullptr) r->next = p;
    if (q != nullptr) r->next = q;
}
```

复杂度：时间 $O(m+n)$，空间 $O(1)$ 。

单链表的就地逆置

问题描述：将带头节点的单链表原地倒置。例如 Head->1->2->3 变为 Head->3->2->1。

解题思路： 不能申请新数组，必须调整指针。主要有两种方法。

> 头插法

思想：

1. 先将头节点 L 与后续节点断开。
2. 像建立新表一样，依次把原链表中的节点摘下来，用头插法插回 L 后面。
3. 头插法的特性就是“逆序”，插完即逆置完成。

C++ 代码实现：

```
void ReverseList(LinkList &L) {
    LNode *p = L->next; // 当前处理节点
    LNode *r;           // 暂存后继节点

    L->next = nullptr;  // 将头节点与原链表断开

    while (p != nullptr) {
        r = p->next;    // 保存后继节点

        // 头插法将当前节点插入到头节点之后
        p->next = L->next;
        L->next = p;

        p = r;          // 处理下一个节点
    }
}
```

快慢指针法

这是一类极其巧妙的算法，专门解决链表“不知道长度”时的定位问题。

寻找链表的中间节点

问题：找到链表的中间节点（如果是偶数个，返回偏左或偏右的那个）。

思路：

- 设置两个指针 `fast` 和 `slow`，都从头开始。
- `slow` 每次走 1 步。
- `fast` 每次走 2 步。
- 当 `fast` 走到终点时，`slow` 刚好走到中点。

```
LNode* FindMidNode(LinkList L) {
    LNode *slow = L->next;  // 慢指针，每次移动一步
    LNode *fast = L->next;  // 快指针，每次移动两步

    // 快指针每次移动两步，直到链表末尾
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow; // 慢指针指向中间节点
}
```

寻找倒数第 k 个节点

问题：只遍历一次链表，找到倒数第 k 个节点。

思路：

- 让 `fast` 先走 k 步。
- 然后 `fast` 和 `slow` 同时走。
- 当 `fast` 走到链表尾部（空）时，`slow` 距离尾部刚好 k 步（即倒数第 k 个）。

```
LNode* FindKthFromEnd(LinkList L, int k) {
    LNode *fast = L->next;  // 快指针
    LNode *slow = L->next;  // 慢指针

    // 快指针先移动k步
    for (int i = 0; i < k; i++) {
        if (fast == nullptr) return nullptr; // 链表长度小于k
        fast = fast->next;
    }

    // 快慢指针同时移动，直到快指针到达链表末尾
    while (fast != nullptr) {
        fast = fast->next;
        slow = slow->next;
    }

    return slow; // 慢指针指向倒数第k个节点
}
```

删除有序表中的重复元素

问题：已知链表有序（如 `1, 2, 2, 3, 3, 3`），要求删除重复元素，使每个元素只出现一次（`1, 2, 3`）。

思路：

- 因为有序，重复元素必然相邻。
- 指针 `p` 扫描链表，比较 `p->data` 和 `p->next->data`。
- 如果相等，说明 `p->next` 是重复的，删除 `p->next`（注意保留 `p` 不动，继续比对新的 `next`）。
- 如果不等，`p` 后移。

```
void DeleteDuplicate(LinkList &L) {
    if (L->next == nullptr) return;

    LNode *p = L->next; // 当前节点
    LNode *q;           // 待删除节点

    while (p->next != nullptr) {
        // 如果当前节点与下一节点值相同
        if (p->data == p->next->data) {
            q = p->next;        // 标记要删除的节点
            p->next = q->next;  // 跳过重复节点
            delete q;           // 释放内存
            // p不移动，继续检查当前节点与新的下一个节点
        } else {
            p = p->next;        // 移动到下一个节点
        }
    }
}
```