数据结构

第2章 线性表

2.1 线性表的基本概念

2.1.2 线性表的抽象数据类型

线性表的抽象数据类型从逻辑上定义线性表这种数据结构的数据对象、数据对象之间的关系，以及相关的基本操作。其中，数据对象说明线性表中的每个数据元素均属于某个类型（如整型、实型或字符型等），这里用一个集合表示：

$$D = \{a_i | a_i \in ElemSet, i = 1, 2, \cdots, n, n \geq 0\}$$

数据对象定义完后，我们来定义数据对象中数据元素之间的关系，这里线性关系运用 $< a_{i-1}, a_i >$ 序偶对来表示前驱和后继的关系。数据对象以及数据元素之间的关系定义完后，再给出发生在这些数据对象上的操作。线性表的抽象数据类型定义如下。

```
ADT List
{
    数据对象：$D = \{a_i | a_i \in ElemSet, i=1,2,\cdots,n, n \geq 0\}$
    数据关系：$R1 = \{<a_{i-1}, a_i> | a_{i-1}, a_i \in D, i=2,\cdots,n\}$
    基本操作：
        InitList(&L)          // 初始化线性表
        CreateList(&L)        // 创建线性表
        DestroyList(&L)       // 销毁线性表
        ListLength(L)         // 求表 L 的长度
        Locate(L, e)          // 查找表 L 中值为 e 的元素
        GetElem(L, i, &e)     // 取元素 $a_i$，由 e 返回 $a_i$
        PriorElem(L, ce, &pre_e) // 求 ce 的前驱，由 pre_e 返回
        InsertElem(&L, i, e)  // 在元素 $a_i$ 之前插入新元素 e
        DeleteElem(&L, i)     // 删除第 i 个元素
        EmptyList(L)          // 判断 L 是否为空表
}
```

这里抽象数据类型定义的操作均是常见的基本操作，是构成复杂操作的基础。我们可把这些基本操作看成搭建模型的基本零件，复杂算法就是由这些基本操作根据不同的组合搭建而成的。

2.2 线性表顺序存储结构及实现

2.2.1 顺序存储结构的线性表定义

顺序存储结构的线性表定义如下。

```
#define MaxLength 100
typedef struct {
    ElemType elem[MaxLength]; // 下标为 0, 1, ..., MaxLength-1
    int length;               // 当前长度
    int last;                 // $a_n$ 的位置
} SqList;
```

精简后对线性表顺序存储结构的定义如下。

```
#define MaxLength 100
typedef struct {
    ElemType elem[MaxLength]; // 下标为 0, 1, ..., MaxLength-1
    int length;               // 表长
} SqList;
```

其中， `elem` 是一个大小为 `MaxLength` 的数据元素数组； `length` 为线性表表长； `SqList` 为此结构类型定义的名称。

2.2.2 顺序表的基本操作实现

动态分配的顺序存储结构

动态分配，即当数据元素超过所分配存储空间的大小时，在堆空间中再找一片更大的连续空间重新分配，将所有的数据元素放入。顺序存储的动态分配定义如下。

```
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef struct {
    ElemType *elem;           // 存储空间基地址
    int length;               // 表长
    int listsize;             // 当前分配的存储容量，以 sizeof(ElemType) 为单位
} SqList;
```

其中， `LIST_INIT_SIZE` 表示第一次为顺序表分配的存储空间大小； `LISTINCREMENT` 表示每次需要扩充存储空间时的增量； `elem` 是一个元素的指针，保存存储空间中第一个数据元素的地址，并且一旦需要扩充线性表的存储空间，可能需要改变 `elem` ，使其指向新空间的起始位置。由于存储空间的大小不是固定的，因此这里增加了一个属性 `listsize` 来记录当前实际存储空间的大小。

静态分配插入算法

静态分配插入算法的基本思想为：先判断插入的位置是否合理，接着判断表长是否达到分配空间的最大值，然后从线性表中的最后一个元素到插入位置的所有元素，依次往后移动一个元素的位置，这样给待插入的元素留出一个空位置，最后把新增元素插入这个空位置，表长增加 1，插入成功返回。

```
// 静态分配顺序表插入算法，用引用参数表示被操作的线性表
Status Insert(SqList *L, int i, ElemType e)
{
    int j;
    if (i < 1 || i > L->length + 1) return ERROR; // i 值不合法
    if (L->length >= MaxLength) return OVERFLOW;  // 溢出
    for (j = L->length - 1; j >= i - 1; j--) {
        L->elem[j + 1] = L->elem[j]; // 向后移动元素
    }
    L->elem[i - 1] = e;              // 插入新元素
    L->length++;                     // 长度变量增 1
    return OK;                       // 插入成功
}
```

动态分配插入算法

```
// 动态分配顺序表插入算法
Status Insert(SqList *L, int i, ElemType e)
{
    int j;
    if (i < 1 || i > L->length + 1) // i 的合法取值为 1 ~ length+1
        return ERROR;
    if (L->length >= L->listsize) { // 溢出时扩充
        ElemType *newbase;
        newbase = (ElemType *) realloc(L->elem, (L->listsize + LISTINCREMENT) * sizeof(ElemType)
        if (newbase == NULL) return OVERFLOW; // 扩充失败
        L->elem = newbase;
        L->listsize += LISTINCREMENT;
    }
    for (j = L->length - 1; j >= i - 1; j--) { // 向后移动元素，空出第 i 个元素的分量 elem[i-1]
        L->elem[j + 1] = L->elem[j];
    }
    L->elem[i - 1] = e; // 新元素插入
    L->length++;        // 线性表长度加 1
    return OK;
}
```

删除元素算法

顺序表删除元素算法的基本思想为：首先判断删除元素的下标是否存在，然后用一个 `for` 循环来移动元素，移动元素下标范围为 `i ~ length-1` ，最后修改表长为原表长减 1。算法如下所示。

```
// 顺序表删除元素
Status Delete(SqList* L, int i)
{
    if (i < 1 || i > L->length)
        return ERROR;
    int j;
    for (j = i; j <= L->length - 1; j++) {
        L->elem[j - 1] = L->elem[j];
    }
    L->length--;
    return OK;
}
```

2.3 线性表链式存储结构定义及实现

链式存储结构概述

链式存储结构是指将线性表中的数据元素存放到计算机存储器内一组非连续存储单元中。在链式结构中，只能通过指针来维护数据元素间的关系。由于这个原因，对线性表中的元素只能进行顺序访问，也就是要访问第 i 个元素，必须先访问前面的 $i-1$ 个元素。

单链表的基本概念

单链表是链式存储结构中最基础、也是最具代表性的一种存储结构形式。单链表是指线性表的每个结点分散地存储在内存空间中，先后依次用一个指针串联起来。单链表可以分为不带表头结点和带表头结点两种情形。

1. 不带表头结点的单链表

其中，线性表中的每个元素通过结点进行存储，结点包含两个属性： `data` 称为数据域，它用于保存元素值； `next` 称为指针域/链域，它用于保存直接后继元素结点的指针、维护元素间的线性关系。 `head` 为头指针，它用于存放首元素结点的指针。当 `head == NULL` 时，表示为空表，否则表示为非空表。

2. 带表头结点的单链表

(1) 非空表

单链表中至少存储一个元素为非空表
其中，头指针 `head` 指向表头结点，表头结点的数据域不放元素，指针域指向首元素结点 a_1。

(2) 空表

单链表中还没有存储数据元素为空表

当 `head->next != NULL` 时，表示为空表，否则表示为非空表。

在具体使用中，究竟是使用带表头结点还是使用不带表头结点取决于实际的应用场景。

2.3.1 单链表存储结构定义

首先定义单链表的结构类型，每个结点有两个部分：一个部分是数据域 `data`；另一个部分是指针域 `next`。具体定义如下：

```
typedef struct node
{
    ElemType data;          // data 为抽象元素类型
    node* next;             // next 为指针类型
} node, *Linklist;
```

指向结点的指针变量 `head`、`p`、`q` 可定义为：

```
node *head, *p, *q;
Linklist head, p, q;
```

单链表是数据结构中非常重要、也是非常基础的一个类型，接下来所讲解的算法均是在这个数据结构定义的基础上进行的。

2.3.2 单链表的实现

下面阐述基于单链表的实例，第一个要讲解的算法是如何生成单链表。

1. 先进先出单链表

该单链表中一个结点的数据结构按如下方式定义：

```
#define LENG sizeof(node)     // 结点所占的单元数
struct node
{
    int data;                 // data 为整型
    node* next;               // next 为指针类型
};
```

首先定义结点所占空间大小，结点的数据域为整型数，然后定义指针域 `next`。

由于单链表元素结点次序与元素的输入次序相同，因此每次输入一个元素后，均将新结点插入单链表的尾部作为最后一个结点。为了提高算法效率，使用了一个尾指针 `tail` 指向单链表的最后一个结点，这

样就能方便新结点的插入。每次新结点链接到 `tail` 指向的结点之后，再修改 `tail` 指向新结点，使用这种插入方式创建单链表的方法俗称尾插法。算法步骤如下：

1. 生成表头结点， `head` 和 `tail` 都指向表头结点。
2. 输入元素的值 `e` ，当元素不是结束标记时，重复下列操作，否则转至步骤 (3)：
   - (1) 生成新结点 `p` ， `e` 保存到 `p` 结点的数据域。
   - (2) 使用 `tail->next = p` ；将 `p` 结点链接到单链表的表尾。
   - (3) 使用 `tail = p` ；让 `tail` 指向当前的表尾结点。
3. 使用 `tail->next = NULL` ；将最后一个结点的指针域赋值为空。
4. 返回 `head` ，完成“先进先出”单链表的创建。

由此算法步骤得到算法代码如下：

```
// 算法：生成“先进先出”单链表（链式队列）
node* create1()
{
    node *head, *tail, *p;          // 变量说明
    int e;
    head = (node*)malloc(LENG);     // 生成表头结点
    tail = head;                    // 尾指针指向表头
    scanf("%d", &e);                // 输入第一个数
    while (e != 0)                  // 不为 0
    {
        p = (node*)malloc(LENG);    // 生成新结点
        p->data = e;                // 装入输入的元素 e
        tail->next = p;             // 新结点链接到表尾
        tail = p;                   // 尾指针指向新结点
        scanf("%d", &e);            // 再输入一个数
    }
    tail->next = NULL;              // 尾结点的 next 置为空指针
    return head;                    // 返回头指针
}
```

2. 先进后出单链表

为实现创建“先进后出”单链表，每当输入一个元素后，生成的结点不是放在表尾而是插入表头，成为新的首元素结点，使用这种插入方式创建单链表的方法俗称首插法。如图 2.15 所示，当前单链表中已有 i 个元素结点，元素输入次序为 $a_1, \cdots, a_i$，现输入第 $i+1$ 个元素 a_{i+1}，具体操作为：

- 第①步生成新结点 p，并保存新元素 a_{i+1}；
- 第②步通过 $p->next = head->next$ 使得新结点指针指向原首结点；

- 第③步通过 $head->next=p$ 让表头结点的指针域指向新结点 a_{i+1}，不再指向 a_i，将新结点作为首元素，即可完成将新结点插入表头的操作。

根据以上分析所设计的生成“先进后出”单链表的算法代码如下：

```
node* create2()
{
    node *head, *p;
    int e;
    head = (node*)malloc(LENG);      // 生成表头结点
    head->next = NULL;               // 置为空表
    scanf("%d", &e);                 // 输入第一个数
    while (e != 0)                   // 不为 0
    {
        p = (node*)malloc(LENG);     // 生成新结点
        p->data = e;                 // 输入数送新结点的 data
        p->next = head->next;        // 新结点指针指向原首结点
        head->next = p;              // 表头结点的指针指向新结点
        scanf("%d", &e);             // 再输入一个数
    }
    return head;                     // 返回头指针
}
```

3. 插入元素

通过以上两个算法的分析，发现不管是生成“先进先出”单链表还是生成“先进后出”单链表，都是不断插入元素的过程，只不过插入元素的位置有所不同。下面说明一般情况下插入元素的操作。

(1) 在已知 p 指针指向的结点后插入一个元素 x

首先用一个指针 f 指向新结点，该结点的数据域为 x，然后此新结点 next 域赋值为 p 指针指向结点的 next 域，最后 p 指针指向结点的 next 域赋值为 f，如图 2.16 所示。其具体操作可表示如下：

1. $f=(node*)malloc(LENG);$ // 生成
2. $f->data=x;$ // 装入元素 x
3. $f->next=p->next;$ // 新结点指向 p 的后继
4. $p->next=f;$ // 新结点成为 p 的后继

(2) 在已知 p 指针指向的结点前插入一个元素 x

因为单链表每个结点只有一个指针指向其后继结点，如果在结点前插入一个新结点，就需要得到 p 指向结点的前驱结点指针，假设该指针为 q。这样问题就转换成在指针 q 指向的结点之后插入一个结

点，即将该问题 (2) 转换成问题 (1) 求解。这类前后指针的方式在单链表的操作中经常出现，一个指针 `p` 在单链表上移动访问结点，另一个指针 `q` 指向刚访问过的结点，一前一后 2 个指针，使得在单链表中完成结点的插入或删除操作非常方便。

其具体操作可表示如下：

1. $f = (node*)malloc(LENG)$; // 生成
2. $f->data = x$; // 装入元素 x
3. $f->next = p$; // 新结点成为 p 的前驱
4. $q->next = f$; // 新结点成为 q 的前驱结点的后继

2.3.3 循环单链表

上文提及的单链表中最后一个结点，其 `next` 域为空。从一已知结点出发，只能访问到该结点及其后续结点，无法找到该结点之前的其他结点。

有的时候，为了应用方便，我们可以将链表中最后一个结点的 `next` 域指向链表的第一个结点而形成一个环，这种单链表称为循环单链表。在循环单链表中，从任一结点出发都可访问到表中所有结点，这一优点使某些运算在循环链表上易于实现。

2.3.4 双向链表

单链表和循环单链表每个结点中只有一个指针指向其后继。对于循环单链表，一个结点需要访问其前驱结点时要顺着 `next` 域扫描整个链表一遍，此时效率显然不高。这里为了方便访问结点的前驱结点而引入双向链表。其中，每个结点除了数据域之外，还有两个指针域（一个指向其直接前驱结点，另一个指向其直接后继结点）。

数据结构的定义

```
typedef struct Dnode
{
    ElemType data;          // data 为抽象元素类型
    Dnode *prior, *next;    // prior、next 为指针类型
} Dnode, *DList;            // DList 为指针类型
```

在实际应用中，一般会在双向链表中加上循环，形成双向循环链表。双向循环链表的一般形式如图 2.21 所示，我们可以根据 `L->next == L` 或 `L->prior == L` 是否成立来判断是否为空表。

在双向循环链表中，对于非空表，如果 `p` 指向某个结点 a_i，$1 \leq i \leq n$，则有 `p->next` 指向 a_{i+1}，当 $i = n$ 时，`p->next->prior` 又回头指向 a_i，所以有关系 `p == p->next->prior`；同理可以得到 `p == p->prior->next`。

从双向链表中删除结点时，需要注意两个指针的变化。例如，已知双向链表中包含结点 A、B、C，指针 `p` 指向结点 B，删除 B，那么所做的操作如下：

1. `p->prior->next = p->next;` // 结点 A 的 `next` 指向结点 C
2. `p->next->prior = p->prior;` // 结点 C 的 `prior` 指向结点 A
3. `free(p);` // 释放结点 B 占有的空间

向双向链表中插入结点时，也需要注意两个指针的变化。例如，已知双向链表中包含两个相邻结点 A 和 C，指针 `p` 指向结点 C，现在插入一个新的结点到 A 和 C 之间，由 `f` 指向该待插入的结点 B，那么所做的操作如下：

1. `f->prior = p->prior;` // 结点 B 的 `prior` 指向结点 A
2. `f->next = p;` // 结点 B 的 `next` 指向结点 C
3. `p->prior->next = f;` // 结点 A 的 `next` 指向结点 B
4. `p->prior = f;` // 结点 C 的 `prior` 指向结点 B

2.5 应用实例

2.5.2 单链表插入、删除算法

单链表插入算法是在单链表的指定位置插入新元素，其中输入参数主要包括头指针 `L`、位置 `i`、数据元素 `e`，而输出为成功返回 `OK`，否则 `ERROR`。首先分析这个算法，在指定位置 i 上插入一个新的元素，使得插入的元素成为链表中的第 i 个元素，那么需要运用指针 `p` 从头扫描单链表，并对所访问的结点进行计数。想一想，计数是在第 i 个位置上结束吗？如果在此结束，指针 `p` 指向第 i 个位置，新元素就要插入指针 `p` 指向的结点之前；之前分析过，如果插入某个结点之前，就需要另一个辅助指针 `q` 来指向指针 `p` 的前驱结点。

实际上，我们是可以简化此过程的，即计数到 $i-1$ 的时候就停止，指针 `p` 指向这个结点，然后新元素插入指针 `p` 所指向结点的后面就达到目的了。扫描的过程即是执行 `p = L`，当指针 `p` 指向的结点不为空时，则需执行 `p = p->next` 指令 $i-1$ 次，从而定位到第 $i-1$ 个结点。当 $i < 1$ 或指针 `p` 所指向的结点为空时，插入点位置出错。

根据以上的分析过程，可以得知在指定位置插入一个新结点的算法如下：

```
Status insert(Linklist L, int i, ElemType e)
{
    node* p = L;
    int j = 1;
    while (p && j < i)
    {
        p = p->next;
        j++;
    }
    if (i < 1 || p == NULL)
        return ERROR;
    node* f = (node *) malloc(LENG);
    f->data = e;
    f->next = p->next;
    p->next = f;
    return OK;
}
```

上文中的几个单链表应用算法均是与在单链表中插入元素的操作相关，下面举例说明在单链表中如何删除元素。在单链表中删除某个结点，一定先找到将要被删除的结点，还要找到其前驱结点。例如，单链表中有 3 个结点，分别是 A、B、C，`q` 指针指向数据域为 A 的结点，`p` 指针指向数据域为 B 的结点。如果删除数据域为 B 的结点，首先执行 `q->next = p->next;`，即 A 的 `next` 域指向的地址（B 的 `next` 域），然后执行 `free(p)`，释放 `p` 所指向的结点空间。

第一个删除算法是在带表头结点的单链表中删除元素值为 e 的结点，那么首先来分析此算法的思想。第一步扫描此单链表以找到元素值为 e 的结点，由于删除此结点需要运用到此结点的前驱结点，那么在扫描过程中需要用到一对指针 `q` 和 `p` 来记录找到的结点和它的前驱结点，一般用如下语句表示：

```
q = head; p = head->next;          // 通过 q、p 扫描
while (p && p->data != e)          // 查找元素为 e 的结点
{
    q = next = p->next;            // 删除该结点
    free(p);                       // 释放结点所占的空间
    return YES;
}
```

第二个算法是在单链表中删除指定位置的元素。在这个 `Delete` 算法中，删除第 i 个元素时，该结点的前驱结点同样重要，因此定位结点也是定位在 $i-1$ 这个位置上，即执行 `p = L;`，`L` 是头指针，当 `p->next` 不为空时执行 `p = p->next; i - 1` 次，从而定位到第 $i-1$ 个结点。接着，判断 i 值，当

$i < 1$ 或 `p->next` 为空时删除点位置出错，否则 `p` 指向后继结点的后继，从而将第 i 个元素结点从单链表中删除。

```
Status Delete(Linklist L, int i)
{
    node* p = L;
    int j = 1;
    while (p->next && j < i)
    {
        p = p->next;
        j++;
    }
    if (i < 1 || p->next == NULL)
        return ERROR;
    node* q = p->next;
    p->next = q->next;
    free(q);
    return OK;
}
```

2.5.3 单链表合并算法

现用一个例子来说明两个有序单链表的合并算法。

例 2-9： 将两个带表头结点的有序单链表 La 和 Lb 合并为有序单链表 Lc，该算法利用原单链表的结点。单链表合并示意图如图 2.24 所示。

首先是表头结点的征用，即使用 La 的表头结点，释放 Lb 的表头结点，可表示为：

```
struct node *pa, *pb, *pc;
pa = La->next;                    // pa 指向表 La 的首结点
pb = Lb->next;                    // pb 指向表 Lb 的首结点
pc = La;                          // 使用表 La 的表头结点，pc 为尾指针
free(Lb);                         // 释放表 Lb 的表头结点
```

接下来，就是比较 `pa` 和 `pb` 指向结点的数据域值大小。`pa`、`pb`、`pc` 指针根据不同的情况发生变化，直到 `pa` 或 `pb` 为空为止。

```
while (pa && pb)
{
    if (pa->data <= pb->data)
    {
        pc->next = pa;
        pc = pa;
        pa = pa->next;
    }
    else
    {
        pc->next = pb;
        pc = pb;
        pb = pb->next;
    }
}
```

若 `pa` 或 `pb` 不为空，则将剩余的结点插入表 Lc 的尾部。假定 La 和 Lb 两个单链表的表长分别是 m 和 n，由于合并过程中每个单链表的结点最多被访问一次，因此算法的时间复杂度为 $O(m+n)$。

2.5.4 单链表的逆置

例 2-10： 假定以单链表作为线性表 $L=(a_1,a_2,\cdots,a_n)$ 的存储结构，现要求设计算法，将线性表逆置为 $L=(a_n,a_{n-1},\cdots,a_1)$。这个问题就涉及将一个单链表中结点翻转的操作，下面给出该问题求解的 4 种算法，并进行效率分析。

1. 递归算法一

采用递归的思想完成逆置操作。当为空单链表时，作为递归出口，直接返回；对非空单链表，首先将最后一个结点从单链表中移出，将剩下长度减一的单链表翻转过来，再将刚移出的结点作为首结点插入这个单链表中。该递归求解算法描述如下：

```
void reverse1(LinkList L)
{
    LinkList p, q;
    if (L->next == NULL) return;      // 空表时返回递归出口
    p = L; q = L->next;
    while (q->next)
    {
        p = q;
        q = q->next;
    }
    p->next = NULL;                   // 被移出的最后一个结点被 q 指向
    reverse1(L);                      // 递归调用，将剩余的链表结点翻转
    q->next = L->next;                // 被移出的结点再插入作为首结点
    L->next = q;
}
```

该算法每次为了移出最后一个结点，需要对单链表遍历一次，共需要做 n 次移出结点操作，平均每次访问结点的个数为 $n/2$，所以算法的时间复杂度 $T(n)=O(n^2)$；递归深度为 n，每一次递归调用时，都会在运行时逻辑空间的栈空间中分配单元，所以空间复杂度 $S(n)=O(n)$。

2. 递归算法二

同样是采用递归算法，这里换一种移出结点的方法。当单链表结点个数不大于 1 时，作为递归出口没什么区别，但仔细分析一下会发现，移出第一个结点不需要遍历单链表。同时将移出的结点作为尾结点插入翻转之后的长度减一单链表中也不需要遍历单链表，因为递归后移出的结点指针所指向的结点正好是尾结点。递归求解算法描述如下：

```
void reverse2(LinkList L)
{
    LinkList p = L->next;             // 获得首元素结点
    // 空单链表或只剩一个结点时返回递归出口
    if (L->next == NULL || p->next == NULL)
        return;
    L->next = p->next;
    reverse2(L);
    p->next->next = p;
    p->next = NULL;
}
```

该算法每次为了移出和插入一个结点，都不需要对单链表遍历，共需要做 n 次移出插入结点操作，所以算法的时间复杂度 $T(n)=O(n)$；递归深度为 n，所以空间复杂度 $S(n)=O(n)$。

3. 折半与递归算法

同时利用折半和递归的思想将一个长度大于 1 的单链表分成两个等长的子单链表，并分别进行翻转，再合并在一起；单链表长度小于或等于 1 作为递归出口处理。为了方便将单链表一分为二，使用了 `p` 和 `q` 这两个一慢一快的指针，快指针 `q` 每向后移动两次， `p` 才移动一次，这样当 `q` 为空时， `p` 正好指向向前一半的最后一个结点，再增加一个 `L1` 指向的头结点作为后一半结点单链表的头结点，完成单链表的一趟拆分。折半与递归求解算法描述如下：

```
Linklist reverse3(Linklist L)
{
    node* p, *q;
    if (!L->next || !L->next->next)
        return L;                    // 递归出口
    node* L1 = (node*)malloc(LENG);
    p = q = L;
    while (q)
    {
        q = q->next;
        if (q) { q = q->next; }
        p = p->next;
    }
    L1->next = q;
    p->next = NULL;
    L1 = reverse3(L1);
    q->next = L->next;
    free(L);
    L = L1;
    return L;
}
```

该算法大约进行 $\log n$ 趟拆分，能将各子单链表结点的个数减少到 1。每趟拆分时，要遍历全部 n 个结点，所以算法的时间复杂度 $T(n) = O(n \log n)$；递归深度为 $\log n$，则空间复杂度 $S(n) = O(\log n)$。

4. 优化算法

上述 3 种算法中，第二种算法的时间复杂度最高，为 $O(n)$。由于逆置过程中必须访问所有结点，因此我们可以认为时间复杂度为 $O(n)$ 已经是最优了，但其空间复杂度为 $O(n)$，这样说明该算法还有优化的空间。优化算法思想是：用一个指针 `p` 指向首结点，再将头指针的指针域赋值为空，变成一个空单链表，接着通过 `p` 依次取原单链表中的结点，用首插法将每个结点再插入回单链表中作为首结点，从而完成单链表的逆置。优化算法描述如下：

```
void reverse4(LinkList L)
{
    LinkList p, q;
    p = L->next;
    L->next = NULL;
    while (p)
    {
        q = p->next;
        p->next = L->next;
        L->next = p;
        p = q;
    }
}
```

该优化算法在逆置过程中，每个结点都被处理 1 次，所以时间复杂度 $T(n)=O(n)$；空间复杂度 $S(n)=O(1)$。由于使用的空间对于问题规模来说是常量，因此优化算法为单链表的就地逆置，也可称为原地工作。

第3章 栈与队列

3.1 栈

3.1.1 栈的基本概念

栈是限定在表尾做插入、删除操作的线性表。向栈中插入元素叫作进栈，从栈中删除元素叫作出栈。栈的表头叫作栈底，栈的表尾叫作栈顶。

栈的相关概念包括以下几个：

- **进栈**：向栈中插入一个元素。其也称为入栈、推入、压入、 `push` 。
- **出栈**：从栈删除一个元素。其也称为退栈、上托、弹出、 `pop` 。
- **栈顶**：允许插入、删除元素的一端（表尾）。
- **栈顶元素**：处在栈顶位置的元素（表尾元素）。
- **栈底**：表中不允许插入、删除元素的一端（表头）。
- **空栈**：不含元素的栈。

栈中元素的进出原则：“后进先出”（Last In First Out, LIFO）。

栈的别名：“后进先出”表、LIFO 表、反转存储器、堆栈。

3.1.2 栈的抽象数据类型

栈的抽象数据类型如下：

```
ADT Stack
{
    数据对象：D = {a_i | a_i ∈ ElemSet, i = 1, 2, ..., n, n ≥ 0}
    数据关系：R1 = {<a_{i-1}, a_i> | a_{i-1}, a_i ∈ D, i = 2, ..., n}，其中 a_1 端为栈底，a_n 端
    基本操作：
        InitStack(&S)          // 初始化栈 S
        DestroyStack(&S)       // 销毁栈 S
        ClearStack(&S)         // 清空栈 S
        StackLength(S)         // 求栈 S 的长度
        Push(&S, e)            // 在栈 S 的栈顶插入元素 e
        Pop(&S, &e)            // 删除栈 S 的栈顶元素，并赋予变量 e
        GetTop(S, &e)          // 将栈 S 的栈顶元素复制到变量 e
        StackEmpty(S)          // 判断栈 S 是否为空栈
}
End ADT
```

3.1.4 栈的顺序存储结构

(1) 静态分配

```
typedef struct {
    ElemType elem[maxlength];  // 栈元素空间
    int top;                    // 栈顶标识
} SqStack;
SqStack s;
```

(2) 动态分配

对比线性表静态（动态）分配方式与栈的静态（动态）分配方式可知，在栈的分配方式中，使用了栈顶标识 `top` 替代线性表分配方式中的表长。其原因为：通过基于 `top` 的简单计算，可以得出栈中元素的个数（即得出表长）。

根据上述分析，约定 `top` 指向栈顶元素下一位置，给出如下动态分配顺序栈的基本操作算法。

例 3-1：设计动态分配顺序栈的初始化算法—— InitStack 函数

首先，调用 malloc 函数为顺序栈分配存储空间，然后为栈顶标识 top 和当前空间大小 stacksize 赋值。

```
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10

typedef struct {
    ElemType *base;             // 指向栈元素空间
    int top;                    // 栈顶标识
    int stacksize;             // 栈元素空间大小，相当于 maxlength
} SqStack;

void InitStack(SqStack &S) {
    S.base = (ElemType *)malloc(STACK_INIT_SIZE * sizeof(ElemType));
    S.top = 0;                  // 初始化为空栈
    S.stacksize = STACK_INIT_SIZE;
}
```

例 3-2：设计进栈算法—— Push 函数

首先，判断栈是否已满，如果栈已满，就运用 realloc 函数重新开辟更大的栈空间。如果 realloc 函数返回值为空，提示“溢出”，则更新栈的地址以及栈的当前空间大小。最终，新元素入栈，栈顶标识 top 加 1。

```
Status Push(SqStack &S, ElemType e) {
    if (S.top >= S.stacksize) {
        // 发生溢出，扩充
        ElemType *newbase = (ElemType *)realloc(S.base,
                                                (S.stacksize + STACKINCREMENT) * sizeof(ElemType
        if (!newbase) {
            printf("溢出");
            return ERROR;
        }
        S.base = newbase;
        S.stacksize += STACKINCREMENT;  // 栈空间扩大
    }
    S.base[S.top] = e;                  // 装入元素 x
    S.top++;                            // 修改顶指针
    return OK;
}
```

例 3-3：设计出栈算法—— Pop 函数

首先，根据栈顶标识 top 判断当前栈是否是一个空栈，如果当前栈是一个空栈，提示“下溢”，否则，更新栈顶标识，取出栈顶元素。

```
Status Pop(SqStack &S, ElemType &e) {
    if (S.top == 0) {
        printf("下溢");
        return ERROR;
    } else {
        S.top--;                    // 修改栈顶指针
        e = S.base[S.top];          // 取出栈顶元素
        return OK;                  // 成功退栈，返回 OK
    }
}
```

上述的 3 个基本算法 InitStack 、 Push 和 Pop 都可以在主函数中进行调用。假如要对一个栈 S 进行操作，首先对其进行类型定义，然后调用 InitStack 初始化栈，再进行进栈 Push 与退栈 Pop 操作，具体代码如下。

```
int main() {
    SqStack S;
    ElemType e;
    InitStack(S);
    Push(S, 10);
    if (Push(S, 20) == ERROR)
        printf("进栈失败！");
    if (Pop(S, e) == OK)
        printf("退栈成功！e=%d", e);
    else
        printf("退栈失败！");
    return 0;
}
```

3.1.5 栈的链式存储结构

对于栈的链式存储结构，通常只考虑采用单链表作为栈的存储结构。首先对结点进行如下定义。

```
struct node {
    ElemType data;          // data 为抽象元素类型
    struct node *next;      // next 为指针类型
} *top = NULL;             // 初始化，置 top 为空栈
```

例 3-4：设计链式栈的进栈算法

链式栈的进栈即是压入元素 e 到以 top 为栈顶指针的链式栈，相当于将新元素 e 插入栈顶元素之前。该操作遵循单链表中插入元素的操作方法：首先为新元素分配存储空间，用一个指针 p 指向它“p = (struct node *)malloc(length);”，然后对新元素结点的数据域进行赋值“p->data = e;”，再对新元素结点的指针域进行赋值，指向首结点“p->next = top;”，最后修改 top 指针指向新元素结点“top = p;”，使其成为新的首结点。

```
struct node *Push_link(struct node *top, ElemType e) {
    struct node *p;
    int length = sizeof(struct node);     // 确定新结点空间的大小
    p = (struct node *)malloc(length);    // 生成新结点
    p->data = e;                          // 装入元素 e
    p->next = top;                        // 插入新结点
    top = p;                              // top 指向新结点
    return top;                           // 返回指针 top
}
```

例 3-5：设计链式栈的出栈算法

链式栈的退栈，即将首元素删除。首先，运用一个新指针指向栈顶结点“p = top;”，然后修改 top 指针，使其指向第二个结点，从而删除栈顶结点“top = top->next;”，最后释放原栈顶结点占据的存储空间“free(p);”。

```
struct node *Pop_link(struct node *top, ElemType *e) {
    struct node *p;
    if (top == NULL) return NULL;         // 首先判断是否空栈，如果是，则返回 NULL
    p = top;                              // p 指向原栈的栈顶结点
    (*e) = p->data;                       // 取出原栈的元素送 (*e)
    top = top->next;                      // 删除原栈的栈顶结点
    free(p);                              // 释放原栈顶结点的空间
    return top;                           // 返回新的栈顶指针 top
}
```

3.2 队列

3.2.1 队列的基本概念

队列也是插入和删除操作位置受限的线性表，只允许在表的一端删除元素，在另一端插入元素。

与队列有关的概念包括以下几个：

- **空队列**：不含元素的队列。
- **队首**：队列中只允许删除元素的一端。其一般称为 `head` 、 `front` 。
- **队尾**：队列中只允许插入元素的一端。其一般称为 `rear` 、 `tail` 。
- **队首元素**：处于队首的元素。
- **队尾元素**：处于队尾的元素。
- **进队**：插入一个元素到队列中。其也称为入队。
- **出队**：从队列中删除一个元素。

与栈的元素进出原则不同，队列的元素进出原则是“先进先出”（First In First Out, FIFO）。队列的别名是“先进先出”表、FIFO 表、queue 等。

队列的进队是指将新元素插入队尾，出队是指将队首元素删除。最常见的一个实际例子是排队买票。买票队伍就相当于一个队列，新来的人只能排在后面，买完票的人就从队伍前面离开。

3.2.2 队列的抽象数据类型

队列的抽象数据类型如下：

```
ADT Queue
{
    数据对象：D = {a_i | a_i ∈ ElemSet, i = 1, 2, ..., n, n ≥ 0}
    数据关系：R1 = {<a_{i-1}, a_i> | a_{i-1}, a_i ∈ D, i = 2, ..., n}，其中 a_1 端为队首，a_n 端
    基本操作：
        InitQueue(&Q)           // 初始化队列 Q
        DestroyQueue(&Q)        // 销毁队列 Q
        ClearQueue(&Q)          // 清空队列 Q
        QueueLength(Q)          // 求队列 Q 的长度
        EnQueue(&Q, e)          // 将 e 插入队列 Q 的尾端
        DeQueue(&Q, &e)         // 取走队列 Q 的队首元素，放入 e
        GetHead(Q, &e)          // 读取队列 Q 的队首元素，放入 e
        QueueEmpty(Q)           // 判断队列 Q 是否为空队列
}
End ADT
```

3.2.3 顺序队列的基本运算及实现

下面在分析循环队列特性的基础上，说明顺序循环队列的入队算法和出队算法。首先，对队列的结构类型 SeQueue 进行如下定义。

```
#define MAXLENGTH 100
typedef struct {
    ElemType elem[MAXLENGTH];
    int front, rear;
} SeQueue;
SeQueue Q;  // 定义结构变量 Q 表示队列
```

其中定义 MAXLENGTH 为 100，即分配给队列的存储空间为可保存 100 个元素的存储单元；这里采取静态分配的顺序存储方式，用一维数组来定义数据类型；两个标识 front 和 rear 分别指向队首结点和队尾结点后的一个空单元；队列 Q 被定义为 SeQueue 类型。

例 3-9：设计循环队列进队算法 En_Queue

假设用 Q 表示顺序队列，头指针 front 指向队头元素， rear 指向尾元素的后一个空位，e 为进队元素。首先要判断队列是否为满队列，如果队列已满，则退出；如果队列不是满队列，则插入新元素； rear 往后移动一个位置；由于是循环队列，还需对分配的存储空间大小取余。

```
Status En_Queue(SeQueue &Q, Elemtype e) {
    if ((Q.rear + 1) % MAXLENGTH == Q.front) {
        return ERROR;  // 若 Q 已满，退出
    }
    Q.elem[Q.rear] = e;  // 装入新元素 e
    Q.rear++;            // 尾指针后移一个位置
    Q.rear = Q.rear % MAXLENGTH;  // 为循环队列
    return OK;
}
```

例 3-10：设计循环队列出队算法 De_Queue

用 Q 表示顺序队列，头指针 front 指向队头元素， rear 指向尾元素的后一个空位。

首先判断队列是否为空队列，如果队列为空，则退出，否则，取出队头元素，并放在 e 中。 front 往后移动一个位置，并且由于是循环队列，因此 front 往后移动时还需对存储空间大小取余。

```
Status De_Queue(SeQueue &Q, Elemtype &e) {
    if (Q.front == Q.rear) {
        return ERROR;  // Q 为空队列，退出
    }
    e = Q.elem[Q.front];  // 取出队头元素，放在 e 中
    Q.front = (Q.front + 1) % MAXLENGTH;  // 循环后移到一个位置
    return OK;
}
```

3.2.4 链式队列的基本运算及实现

存放元素的结点类型定义（此定义与单链表结点类型定义一致，其包括数据域和指针域）如下。

```
typedef struct Qnode {
    ElemType data;      // 数据域 data 为抽象元素类型
    struct Qnode *next; // 指针域 next 为指针类型
} Qnode, *QueuePtr;     // Qnode 为结点类型；QueuePtr 为指向 Qnode 的指针类型
```

由头、尾指针组成的结点类型定义如下。

```
typedef struct {
    Qnode *front; // 头指针
    Qnode *rear;  // 尾指针
} LinkQueue;      // 链式队列类型
```

例 3-11：设计生成空队列算法，即初始化队列算法 InitQueue

```
#define LENGTH sizeof(Qnode)
void InitQueue(LinkQueue &Q) {
    Q.front = Q.rear = (QueuePtr)malloc(LENGTH);  // 生成表头结点
    Q.front->next = NULL;                         // 表头结点的 next 为空指针
}
```

例 3-12：设计队列插入算法

根据以上的分析过程，编写出插入元素 e 的算法 EnQueue 如下。

```
Status EnQueue(LinkQueue &Q, ElemType e) {
    Qnode *p;
    p = (Qnode *)malloc(LENGTH);  // 生成新元素结点
    if (!p) return ERROR;
    p->data = e;                  // 装入元素 e
    p->next = NULL;               // 设置为队尾结点
    Q.rear->next = p;             // 插入新结点
    Q.rear = p;                   // 修改尾指针
    return OK;
}
```

主函数可以调用 `InitQueue` 函数进行初始化，然后调用 `EnQueue` 函数进行元素的插入。例如定义一个队列 `que`，初始化后可插入一个新元素 10。

```
int main() {
    LinkQueue que;  /* 定义一个队列 */
    InitQueue(que);
    EnQueue(que, 10);
    return 0;
}
```

下面分析删除操作，注意队列删除元素是删除队首元素。如果队列有两个或者两个以上的结点，只需要变化表头结点的指针即可完成删除操作。表头结点的指针指向队首结点，定义一个指针 p 指向此结点 "$p = Q.front->next$"，然后为删除队首结点变化表头结点的指针，表头结点的指针指向首结点的后继结点 "$Q.front->next = p->next$"，最后释放点原首结点所占据的空间，即 "$free(p)$"。

如果队列只有一个结点，除了变化表头结点的指针，还需考虑队尾指针（因为队列只有一个结点，删除该结点后的队列就是空队列）。具体操作方法为：首先删除首结点 "$Q.front->next = p->next$"，然后释放首结点的空间 "$free(p)$"，再将尾指针指向表头结点，表明队列是一个空队列 "$Q.rear = Q.front$"。

例 3-13：设计队列删除算法

根据上述分析，编写出删除队列元素 e 的算法 `DeQueue` 如下。

```
Status DeQueue(LinkQueue &Q, ElemType &e) {
    if (Q.front == Q.rear) {
        return ERROR;  // 若原队列为空
    }
    Qnode *p;
    p = Q.front->next;  // p 指向队头结点
    e = p->data;        // 取出元素，e 指向它
    Q.front->next = p->next;  // 删除队头结点
    if (Q.rear == p) {
        Q.rear = Q.front;  // 修改尾指针
    }
    free(p);               // 释放被删除结点的空间
    return OK;             // 返回出队后的 Q
}
```

第4章 字符串、多维数组与广义表

4.1 字符串

4.1.1 字符串的定义

字符串是由零个或者多个字符组成的有限序列，一般记为：

$$S = ”a_1 a_2 \cdots a_n” \quad (n \geq 0)$$

其中：

- S 是字符串的名称；
- 双引号 ("") 括起来的字符序列为串值，双引号本身不属于串值，只是代表串的起止标记；
- 序列中的 $a_i\ (1 \leq i \leq n)$ 可以是字母、数字和其他字符，i 称为字符 a_i 在该串中的位置；
- n 表示串的长度，即串中包含的字符个数；
- 当 $n = 0$ 时，称为空串（null string）；仅含若干个空格的串称为空格串。

抽象数据类型（ADT）定义

```
ADT String
{
    数据对象：D = {a_i | a_i ∈ 字符集合，i = 1, 2, ..., n, n ≥ 0}
    数据关系：R1 = {<a_{i-1}, a_i> | a_{i-1}, a_i ∈ D, i = 2, ..., n}
    基本操作：
        StrAssign(&T, S)        // 根据串常量 S，创建串 T
        StrDestroy(&S)          // 销毁串 S
        StrCopy(&T, S)          // 将串 S 复制到串 T
        StrLength(S)            // 求串 S 的长度
        StrComp(S, T)           // 比较串 S 和串 T 的值
        StrSub(&T, S, pos, len)// 从串 S 位置 pos 取长度为 len 的子串赋予 T
        StrConcat(&T, S)        // 将串 S 的字符连接到 T 的尾部
        StrIndex(S, T, pos)     // 求串 T 在 S 中位置 pos 后第一次出现的位置
        StrInsert(&T, S, pos)  // 将串 S 插入串 T 的第 pos 个字符之前
        StrDelete(&S, pos, len)// 将串 T 位置 pos 开始的长度为 len 的子串删除
}
End ADT
```

例 4-1：串替换算法

```
void StrReplace(SeqString* S, SeqString T, SeqString V)
{
    int i = 1, len_T = StrLength(T), len_V = StrLength(V);
    while ((i = StrIndex(*S, T, i)) != 0)
    {
        StrDelete(S, i, len_T);
        StrInsert(S, V, i);
        i = i + len_V;
    }
}
```

4.1.2 字符串的存储结构及其基本运算的实现

与线性表类似，通常字符串也有顺序存储和链式存储两大类，对应的有顺序串和链串。而顺序存储又可以细分为静态存储分配和动态存储分配，这样字符串的存储结构就可以分为 3 类。

1. 静态存储分配的顺序串

该存储结构就是通过字符数组的方式分配连续的存储空间来保存串值。数组的存储空间是在编译时确定的，并且运行时不能改变连续空间的大小，这样能表示的字符串长度最大值就固定下来了，所以这种形式表示的串也称为串的定长顺序存储表示。例如：

```
#define MAXLENGTH 256
typedef unsigned char SeqString[MAXLENGTH];
SeqString S;
```

（1）串插入操作 StrInsert(&T, S, pos)

假定 T 和 S 都是类型为 SeqString 的串变量，现要将串 S 插入串 T 的第 pos 个位置之前。首先验证插入点是否正确，T 是一个长度为 $T[0]$ 的串，插入位置 pos 的取值范围应该是 $1 \sim T[0]+1$；接着判断 T 的空闲空间是否足够大，能否再增加 $S[0]$ 个字符；如果这些条件都满足，此时就将 T 中从第 pos 个字符开始直到最后一个字符区间内的所有字符向后移动 $S[0]$ 个字符位置，最后将 S 的串值从 T 的位置 pos 开始复制到 T 中，完成插入操作。

例 4-2：串插入算法

```
status StrInsert(SeqString T, SeqString S, int pos)
{
    if (pos < 1 || pos > T[0] + 1) return ERROR;
    if (T[0] + S[0] >= MAXLENGTH) return ERROR;
    int j;
    for (j = T[0]; j >= pos; j--)
        T[j + S[0]] = T[j];
    for (j = 0; j < S[0]; j++)
        T[pos + j] = S[j + 1];
    T[0] = T[0] + S[0];
    return OK;
}
```

（2）串比较操作 StrComp(S, T)

串比较操作是将两个字符串进行比较，即：如果 $S > T$，返回一个正整数；如果 S 与 T 相等，返回 0；否则返回负整数。该操作算法实现时，首先需要将 S 和 T 对应位置上的字符进行比较，这时需要计算 $S[0]$ 和 $T[0]$ 间的较小值并赋予 m，表示 S 和 T 在位置 $1 \sim m$ 上的字符可进行比较，如果在位置 i 上出现不相等的情况，即 $S[i] \neq T[i]$ 时，返回 $S[i]-T[i]$，结果为正就表示 $S > T$，为负表示 $S < T$。如果位置 $1 \sim m$ 上的字符都对应相等，则两个串相等，返回 0；第二种情况是 $S[0] > T[0]$，表示 T 中的字符都比较过了，而 S 中还有若干个字符，在 T 中没有对应位置的字符可进行比较，所

以 $S > T$，返回正整数；剩下的第三种情况就是 $S[0] < T[0]$，需要返回一个负整数，这 3 种情况很容易合并在一起，返回值用 $S[0] - T[0]$ 即可。

例 4-3：串比较算法

```
int StrComp(SeqString S, SeqString T)
{
    int i, m = S[0] < T[0] ? S[0] : T[0];
    for (i = 1; i <= m; i++)
        if (S[i] != T[i]) return S[i] - T[i];
    return S[0] - T[0];
}
```

静态存储分配的顺序串还可以沿用线性表中的静态存储分配的顺序表，其定义如下：

```
typedef struct {
    unsigned char ch[MAXLENGTH];
    int length;
} SeqString;
```

这个类型定义用 `SeqString` 说明变量表示的串，最大长度为 256。两种定义方式只是形式上的差异，本质上是相同的。

2. 动态存储分配的顺序串

由于定长顺序串的空间是在编译阶段就确定的，运行阶段不能够改变空间大小，这样就会出现一些常见的问题：预留空间太大、串长较小而造成空间的浪费；如果空间不是足够大，在做插入、联接操作时可能会舍弃超长部分，造成数据的丢失。为此，我们可以考虑采用线性表中动态存储分配的顺序表，利用动态分配函数 `malloc`，根据串长来申请分配串需要的空间，并用 `free` 函数来释放串空间。通过这种方式，就能有效地避免前述静态存储分配顺序串的缺陷。由于使用 `malloc` 函数申请内存空间时是在程序运行时逻辑空间中的堆空间（heap space）进行的，所以动态存储分配的顺序串也被称为串的堆存储分配表示。其数据类型的定义如下：

```
typedef struct {
    unsigned char *ch;
    int length;
} HString;
```

（1）串赋值操作 StrAssign(&T, S)

假定 T 是类型为 HString 的串，S 是以 “\0” 结束的串常量。赋值操作后，T 原来的值会被替换掉，所以操作前需要判断 T 是否为空串，非空串就需要释放原有串的空间；接着计算 S 的串长，根据 S 串长为 T 分配存放串值的空间，并将 S 的串值复制到 T 中，修改 T 的 length 属性，完成串赋值操作。

例 4-4：串赋值算法

```
status StrAssign(HString *T, char *S)
{
    int i;
    for (i = 0; *(S + i); i++);
    char *new_ch;
    if (!i) new_ch = NULL;
    else if (!(new_ch = (unsigned char *)malloc(i * sizeof(char))))
        return OVERFLOW;
    if (T->ch) free(T->ch);
    T->ch = new_ch;
    T->length = i;
    for (i = 0; i < T->length; i++)
        T->ch[i] = S[i];
    return OK;
}
```

（2）求子串操作 StrSub(&T, S, pos, len)

假定 T 和 S 是类型为 HString 的串，现要从 S 的位置 pos 取长度为 len 的子串赋予 T。首先要判断 pos 是否为 S 中一个正确的位置，以及从位置 i 开始，在 S 中是否能取到一个长度为 len 的子串。如果正确，就从 S 位置 pos 开始读取 len 个字符得到子串，并将其赋予 T。

例 4-5：求子串算法

```
status StrSub(HString *T, HString S, int pos, int len)
{
    if (pos < 1 || pos > S.length || len < 0 || S.length - pos + 1 < len)
        return ERROR;
    char *new_ch;
    if (!len) new_ch = NULL;
    else if (!(new_ch = (unsigned char *)malloc(len * sizeof(char))))
        return OVERFLOW;
    if (T->ch) free(T->ch);
    T->ch = new_ch;
    T->length = len;
    int j;
    for (j = 0; j < len; j++)
        T->ch[j] = S.ch[pos + j - 1];
    return OK;
}
```

(3) 串联接操作 StrConcat(&T, S)

假定 T 和 S 是类型为 HString 的串，如果 S 是一个非空串，就需要为 T 重新分配一个更大的空间，将 T 和 S 联接，结果保存在 T。

例 4-6：串联接算法

```
status StrConcat(HString *T, HString S)
{
    if (S.length) {
        int len = T->length + S.length;
        char *new_ch;
        if (!(new_ch = (unsigned char *)realloc(T->ch, len * sizeof(char))))
            return OVERFLOW;
        T->ch = new_ch;
        int i;
        for (i = 0; i < S.length; i++)
            T->ch[T->length + i] = S.ch[i];
        T->length = len;
        return OK;
    }
}
```

3. 串的链式存储

与线性表类似，串也可以采用链式存储结构来表示，如使用单链表，串的链式存储结构也称为链串，如图 4.3(a) 所示。使用链式存储结构能方便地进行插入与删除等操作，且能避免大量移动字符。链式存储结构类型定义如下：

```
typedef struct node {
    char data;
    struct node *next;
} LinkStrNode, *LinkString;
```

在链式存储结构中，首先引入一个存储密度的概念，其公式为：

$$存储密度 = \frac{串值所占存储字节数}{实际分配存储字节数} \times 100\%$$

如果在一个结点存放 1 个字符，占 1 字节，在 32 位系统中指针需要占 4 字节，所以链串的存储密度为 20%，存储密度是相当低的；如果是 64 位系统，存储密度会更低。为了提高存储密度，这里可以考虑在一个结点中存放多个字符，比如放 4 个字符，如图 4.3(b) 所示。通常将一个结点数据域存放的字符数定义为结点大小。对于一个非空串，由于串的长度不一定正好是结点大小的整数倍，因此在最后一个结点需要填充特殊的符号，代表串的结束。对于结点大小大于 1 的链式存储，其类型定义的一般形式如下：

```
#define NODESIZE 4
typedef struct node {
    char data[NODESIZE];
    struct node *next;
} LinkStrNode, *LinkString;
```

4.1.3 字符串的模式匹配算法

子串定位操作 StrIndex(S, T, pos)

子串定位操作 StrIndex(S, T, pos) 的目的是求串 T 在串 S 中第 pos 个字符后第一次出现的位置。此操作应用非常广泛，是较为重要的串操作之一。在各种文本处理系统中，一个好的定位算法能够极大地提升系统的响应性能。

子串定位操作也称为串的模式匹配，其中主串 S 称为目标串，子串 T 称为模式串。假定目标串 S 的长度为 n，模式串 T 的长度为 m，S 和 T 分别表示如下。在实际应用中，通常 m 远小于 n，即 $m \ll n$。

$$S = "s_1 s_2 \cdots s_n" \quad T = "t_1 t_2 \cdots t_m"$$

串模式匹配操作就是在目标串 S 中找到一个与模式串 T 相等的子串 "$s_i s_{i+1} \cdots s_{i+m-1}$"，这里 i 取符合条件的最小值（$pos \leq i \leq n - m$）。如果存在这样的 i，表示匹配成功；否则表示匹配失败，模式串 T 不是目标串 S 的子串。

1. 朴素的模式匹配算法

朴素的模式匹配算法是一种简单的字符串匹配方法，其核心思想是逐字符比较目标串和模式串，直到找到匹配或遍历完整个目标串。

示例：朴素模式匹配算法（例 4-7）

```
int StrIndex(SeqString S, SeqString T, int pos)
{
    int i, j;
    for (int i = pos, j = 1; i <= S[0] - T[0] + 1; i++, j++) {
        if (S.ch[i] != T.ch[j]) {
            i = i - j + 1; // 回溯到本次匹配起点，准备下次匹配
            j = 0;
        } else if (j == T[0]) { // 匹配成功，返回本次匹配起始位置
            return i - j + 1;
        }
    }
    return 0; // 匹配失败
}
```

算法分析：

- 时间复杂度：最坏情况下，朴素算法的时间复杂度为 $O(n \cdot m)$，因为每次不匹配时需要回溯。
- 空间复杂度：$O(1)$，仅使用常量级别的额外空间。

2. KMP 算法（Knuth-Morris-Pratt Algorithm）

KMP 算法是一种高效的字符串匹配算法，通过预处理模式串来避免不必要的回溯，从而显著提高匹配效率。

（1）KMP 算法的核心思想

KMP 算法的核心是利用模式串本身的性质，构建一个部分匹配表（Next 数组），用于指导匹配过程中如何高效地移动指针，避免不必要的回溯。

部分匹配表（Next 数组）

- Next 数组的定义：对于模式串

  $$T = "t_1 t_2 \cdots t_m"$$

  Next 数组记录了每个位置 j 的最长公共前后缀长度，即：

  $$Next[j] = \max\{k | 0 \leq k < j \text{ 且 } T[1..k] = T[j-k+1..j]\}$$

- Next 数组的作用：当匹配失败时，根据 Next 数组的值，可以快速确定模式串的下一个匹配位置，而不需要重新从头开始匹配。

构建 Next 数组的算法

- 初始化：$Next[1] = 0$
- 对于 $j = 2$ 到 m：
  - 如果 $T[j] = T[k+1]$，则 $Next[j] = k+1$
  - 否则，令 $k = Next[k]$，重复上述步骤，直到找到匹配或 $k = 0$

（2）KMP 算法的实现

示例：KMP 算法实现

```
// 构建 Next 数组
void GetNext(char *T, int next[]) {
    int m = strlen(T);
    next[0] = -1; // 初始化
    int k = -1;
    for (int j = 1; j < m; j++) {
        while (k != -1 && T[j] != T[k + 1]) {
            k = next[k];
        }
        if (T[j] == T[k + 1]) {
            k++;
        }
        next[j] = k;
    }
}

// KMP 算法
int KMPMatch(char *S, char *T) {
    int n = strlen(S);
    int m = strlen(T);
    int next[m];
    GetNext(T, next); // 构建 Next 数组

    int i = 0, j = 0;
    while (i < n && j < m) {
        if (j == -1 || S[i] == T[j]) {
            i++;
            j++;
        } else {
            j = next[j];
        }
    }

    if (j == m) {
        return i - j; // 匹配成功，返回匹配起始位置
    } else {
        return -1; // 匹配失败
    }
}
```

（3）KMP 算法的时间复杂度

- **预处理阶段（构建 Next 数组）：** 时间复杂度为 $O(m)$，其中 m 是模式串的长度。
- **匹配阶段：** 时间复杂度为 $O(n)$，其中 n 是目标串的长度。

因此，KMP 算法的总时间复杂度为 $O(n+m)$，比朴素算法的 $O(n \cdot m)$ 显著提高。

总结

- **朴素算法：** 简单易懂，但效率较低，时间复杂度为 $O(n \cdot m)$。
- **KMP 算法：** 高效，通过预处理模式串避免不必要的回溯，时间复杂度为 $O(n+m)$。

KMP 算法在实际应用中具有重要意义，尤其是在需要频繁进行字符串匹配的场景中，如文本编辑器、搜索引擎等。