线性表

线性结构是最简单、最常用的数据结构，其特点是数据元素之间的逻辑关系是线性关系。线性结构是数据元素之间约束力最强的一种数据结构：在非空线性结构的有限集合中，存在唯一一个被称为“第一个”的数据元素；存在唯一一个被称为“最后一个”的数据元素；除“第一个”数据元素无前驱外，集合中的每个元素数据均有且只有一个“直接”前驱；除“最后一个“元素无后继外，集合中的每个数据元素均有且只有一个”直接“后继。

线性表的类型定义

线性表的概念

线性表是一种线性结构，一个线性表是n个数据元素的有限序列，元素可以是各种各样的，但必须具有相同的性质，同属于一种数据对象。在较为复杂的线性表中，一个元素可以由若干数据项(item)组成。这种线性表中的元素也常称为记录(record)。为了方便以后的使用，含有大量记录的线性表往往存放在外部存储介质上，称为文件(file)。

线性表的概念和术语说明如下：

1. 线性表是具有相同数据类型的$n(n \geq 0)$个元素的有限序列，通常记为：

   $$(a_0, a_1, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1})$$

   表中相邻元素之间存在着顺序关系，a_{i-1}先于a_i，a_i先于a_{i+1}，称a_{i-1}是a_i的前驱，a_{i+1}是a_i的后继。也就是说，在这个集合中，除了a_0和a_{n-1}以外，每个元素都有唯一的前驱和后继。而a_0是第一个元素，只有后继没有前驱；a_{n-1}是最后一个元素，只有前驱没有后继。
2. 线性表中元素的个数$n(n \geq 0)$称为线性表的长度，当$n = 0$时称为空表
3. a_0被称为首节点，a_{n-1}被称为尾节点。
4. 在非空线性表中，每个元素都有一个确定位置，设$a_i(0 \leq i \leq n-1)$是线性表中的元素，则称i为元素a_i在线性表中的位序（位置）。

线性表的抽象数据类型

线性表的抽象数据类型定义如下：

```
template<class T>
class List{
public:
    virtual void clear() = 0; // 清空线性表
    virtual bool empty()const = 0; // 判空，表空返回true，非空返回false
    virtual int size()const = 0; // 求线性表的长度
    virtual void insert(int i, const T &value) = 0; // 在位序i处插入值为value的元素
    virtual void remove(int i) = 0; // 删除位序i处的元素
    virtual int search(const T &value)const = 0; // 查找值为value的元素第一次出现的位序
    virtual T visit(int i)const = 0; // 查找位序为i的元素并返回其值
    virtual void traverse()const = 0; // 遍历线性表
    virtual void inverse() = 0; // 逆置线性表
    virtual ~List(){};
};
```

自定义异常处理类：

```
class outOfRange : public exception{ // 用于检查范围的有效性
public:
    const char* what()const throu(){return "ERROR! OUT OF RANGE.\n";}
} ;
class badSize : public exception{ // 用于检查长度的有效性
public:
    const char* what()const throw(){return "ERROR! BAD SIZE.\n";}
};
```

线性表的顺序表现和实现

线性表的顺序表示

线性表在计算机内部可以用几种方法表示，最简单和最常用的方法是用顺序存储方式表示，即在内存中用一块连续有限的存储空间存放线性表的哥哥元素，用这种存储形式存储的线性表称为顺序表。顺序表用物理上的相邻（内存中的地址空间是线性的）实现元素之间的逻辑相邻关系。假定线性表的每个元素占据L个存储单元，若知道第一个元素的地址（基地址）为$Loc(a_0)$，则位序为i的元素的地址为：

$$Loc(a_i) = Loc(a_0) + i \times L \qquad (0 \leq i \leq n-1)$$

只要已知顺序表首地址$Loc(A_0)$和每个元素的大小L就可以通过上述公式求出位序为i的元素的地址，时间复杂度为$O(1)$，因此顺序表具有按元素的序号随机存取的特点。

顺序表的类型定义如下

```
template <class elemType> // elemType为顺序表存储的元素类型
class seqList : public List<elemType>{
private:
    elemType *data; // 动态数组
    int curLength; // 当前顺序表中存储的元素个数
    int maxSize; // 顺序表的最大长度
    void resize(); // 表满时扩大表空间
public:
    seqList(int initSize = 10); // 构造函数
    seqList(seqList &sl); // 复制构造函数
    ~seqList(){delete [] data;} // 析构函数
    void clear(){curLength = 0;} // 清空表，只需要设置curLength = 0即可
    bool empty()const{return curLength == 0;} // 判空
    int size()const{return curLength;} // 返回顺序表中当前存储元素的个数
    void traverse()const; // 遍历顺序表
    void inverse(); // 逆置顺序表
    void insert(int i, const elemType &value); // 在位序i处插入值为value的元素，表长+1
    void remove(int i); // 删除位序i处的元素，表长-1
    int search(const elemType &value)const; // 查找值为value的元素第一次出现的位序
    elemType visit(int i)const; // 访问位序为i的元素的值，"位序0"表示第一个元素
}
```

注意：在插入、删除、查找等运算中涉及的参数i指的是元素在线性表中的位序（即下表）。假定线性表有curLength个元素，则数组下标为0到curLength-1

顺序表基本运算的实现

构造函数

```
template <class elemType>
seqList<elemType>::seqList(int initSize = 100){
    if (initSize <= 0) throw badSize();
    maxSize = initSize;
    data = new elemType[maxSize];
    curLength = 0;
}
```

复制构造函数

在构造函数里动态分配了内存资源，这时需要用户自定义复制构造函数进行深拷贝。

```
template <class elemType>
seqList<elemType>::seqList(seqList& sl){
    maxSize = sl.maxSize;
    curLength = sl.curLength;
    data = new elemType[maxSize];
    for (int i = 0; i < curLength; ++i){
        data[i] = sl.data[i];
    }
}
```

遍历顺序表

算法思想：所谓遍历，就是访问线性表中的每个元素，并且每个元素只访问一次。访问的含义包括查询、输出元素和修改元素等等。如果顺序表是空表，没有元素，则输出`empty`，否则从第一个元素开始一次输出所有元素。由于线性表中当前元素的个数为`curLength`，因此，下标的范围是`[0, curLength-1]`。遍历顺序表的时间复杂度为$O(n)$

```
template <class elemType>
seqList<elemType>::traverse()const{
    if(empty()){cout<<"is empty"<<endl;} // 空表没有元素
    else{
        cout<<"output element:"<<endl;
        for (int i = 0; i < curLength; i++){ // 依次访问顺序表中的所有元素
            cout<<data[i]<<" ";
        }
        cout<<endl;
    }
}
```

查找值为value的元素

算法思想：顺序查找值为value的元素在线性表中第一次出现的位置，需要遍历线性表，将线性表中的每个元素依次与value进行比较。若`value == data[i]`，`i`的取值范围是`[0,curLength-1]`，则查找成功，返回`data[i]`的位序`i`，否则查找失败返回`-1`

```
template <class elemType>
seqList<elemType>::search(const elemType& value)const{
    for (int i = 0; i < curLength; i++){
        if (value == data[i]){return i;}
    }
    return -1;
}
```

顺序表的顺序查找算法的主要操作是比较数据。在查找成功的情况下，最好情况是：要找的元素是第1个元素，比较次数是1次；最坏情况是：要查找的元素是第n个元素，比较次数为n次。设查找表中第i个元素的概率为p_i，找到第i个元素所需的比较次数为c_i，则查找的平均期望为$\sum_{i=1}^{n} p_i \times c_i$。在等概率的情况下，查找成功的平均期望值为$\frac{n+1}{2}$。所以元素定位算法的时间复杂度为$O(n)$

求前驱和后继

算法思想：求顺序表中位序i处的元素的前驱和后继，若`i = 0`，则为第1个元素，无前驱，否则其前驱是`data[i-1]`；若`i = curLength - 1`，则为最后一个元素，无后继，否则其后继是`data[i+1]`。通过元素的下标可以直接定位其前驱和后继，算法的时间复杂度为$O(1)$

插入运算

算法思想：设线性表$L = (a_0, a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_{n-1})$，线性表的插入是指在位序$i$处插入一个值为`value`的新元素，使$L$变为：

$$(a_0, a_1, \ldots, a_{i-1}, \text{value}, a_i, a_{i+1}, \ldots, a_{n-1})$$

`value`的插入会使得a_{i-1}和a_i的逻辑关系发生变化，并且表长由n变为$n+1$。i的取值范围是$0 \le i \le n$，其中，0表示插入点在表头，n表示插入点在表尾。当$i = n$时，只需在a_{n-1}后面插入`value`即可；当$0 \le i \le n-1$时，需要将$a_{n-1} \sim a_i$的顺序向后移动，为新元素让出位置，将值为`value`的元素放入空出的位序为i的位置，并修改表的长度。

```
template <class elemType>
seqList<elemType>::insert(int i, const elemType& value){
    if (i < 0 || i > curLength) throw outOfRange(); // 合法的插入范围为[0,curLength]
    if (curLength == maxSize) resize();
    for (int j = curLength; j > i; j--){
        data[j] = data[j-1]; // 下标在[curLength-1,i]范围内的元素往后移动一步
    }
    data[i] = value; // 将值为value的元素放入位序为i的位置
    ++curlength; // 表的实际长度+1
}
```

本算法中要注意以下问题：

1. 检验插入位置的有效性，这里`i`的有效范围是`[0, curLength]`。注意，在表尾元素`data[curLength-1]`的后面插入元素称为新的表尾是合法的。
2. 要检查表空间是否已满。在表满的情况下不能再做插入操作，否则产生溢出错误。此时有两种解决方法：一种是不执行插入操作，报错后退出；另一种是扩大数组的容量。
3. 注意数据的移动方向，最先移动的是表尾元素。

下面我们分析插入算法的时间复杂度。顺序表的插入元素的基本操作是移动数据。位序i处插入值为`value`的元素，从a_{n-1}到a_i都要向后移动一个位置，需要移动$n-i$个元素，而i的取值范围为$0 \le i \le n$，即有$n+1$个位置可以插入。设在第i个位置插入的概率为p_i，则平均移动元素的次数（期望值）为：

$$E_{\text{insert}} = \sum_{i=0}^{n} p_i \times (n-i)$$

假设$p_i = \frac{1}{n+1}$，即等概率情况，则：

$$E_{\text{insert}} = \sum_{i=0}^{n} p_i \times (n-i) = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1} \times \frac{(n+1)\times n}{2} = \frac{n}{2}$$

所以在顺序表中做插入操作，平均需要移动表中一半的元素。算法的时间复杂度为$O(n)$。

删除运算

算法思想：设线性表$L=(a_0,a_1,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_{n-1})$，线性表的删除运算是将在位序$i$处的元素$a_i$从线性表中去掉，使$L$变为：

$$(a_0,a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_{n-1})$$

a_i的删除使得a_{i-1}、a_i和a_{i+1}的逻辑关系发生变化，并且表长由n变为$n-1$。i的取值范围是$0 \le i \le n-1$，当$i=n-1$时，删除的是表尾元素，无需移动任何元素，只要修改表长即可；当$i<n-1$时，删除元素a_i需要将其后的元素$a_{i+1} \sim a_{n-1}$顺序向前移动，并修改表长。

```
template <class elemType>
seqList<elemType>::remove(int i){
    if ( i < 0 || i > curLength - 1) throw outOfRange(); // 合法的删除范围为[0,curLength-1]
    for (int j = i; j < curLength - 1; j++){
        data[j] = data[j+1]; // 下标在[i+1,curLength-1]范围内的元素往前移动一步
    }
    --curLength; // 表的实际长度-1
}
```

本算法中要注意以下问题：

1. 检验删除位置的有效性，这里`i`的有效范围是`[0, curLength-1]`。
2. 当顺序表为空时不能做删除操作。当顺序表为空时，`curLength`的值为0。
3. 删除a_i之后，该数据被覆盖，如果需要保留它的值，则先取出a_i，再做删除操作。

下面分析删除算法的时间复杂度。删除算法的主要操作仍是移动元素。删除位序i处的元素时，其后面的元素$a_{i+1} \sim a_{n-1}$都要向前移动一个位置，共移动了$n-i-1$个元素，所以平均移动元素的次数（期望值）为：

$$E_{\text{remove}} = \sum_{i=1}^{n-1} q_i \times (n-i-1)$$

在等概率情况下，$q_i = \frac{1}{n}$，则：

$$E_{\text{remove}} = \sum_{i=1}^{n-1} q_i \times (n-i-1) = \frac{1}{n}\sum_{i=1}^{n-1}(n-i-1) = \frac{1}{n} \times \frac{n\times(n-1)}{2} = \frac{n-1}{2}$$

所以在顺序表中做删除操作平均需要移动表中一半的元素，算法的时间复杂度为$O(n)$。

逆置运算

算法思想：设线性表$L=(a_0,a_1,\ldots,a_{n-2},a_{n-1})$，线性表的逆置是指调整线性表中元素的顺序，使$L$变为$(a_{n-1},a_{n-2},\ldots,a_1,a_0)$。可用两两交换元素的方法求解，即交换$a_0$和$a_{n-1}$，$a_1$和$a_{n-2}$，……共进行$\frac{n}{2}$次交换。算法中循环控制变量终值的设置是关键。因为首尾对称交换，所以循环控制变量的终值是线性表长度的一半。算法的时间复杂度为$O(n)$

```
template <class elemType>
seqList<elemType>::inverse(){
    elemType tmp;
    for (int i = 0; i < curLength/2; i++){

    }
}
```

扩大表空间

算法思想：由于数组空间在内存中必须是连续的，因此，扩大数组空间的操作需要重新申请一个更大规模的新数组，将原有数组的内容复制到新数组中，释放原有数组空间，将新数组作为线性表的存储空间。算法的时间复杂度为$O(n)$。

```
template <class elemType>
seqList<elemType>::resize(){
    elemType* p = data; // p指向原顺序表空间
    maxSize *= 2; // 表空间扩大2倍
    data = new elemType[maxSize]; // data指向新的表空间
    for (int i = 0; i < curLength; i++){
        data[i] = p[i]; // 复制元素
    }
    delete [] p;
}
```

合并顺序表

顺序表A与B的节点关键字为整数，A表与B表中的元素均为非递减有序，试给出一种高效的算法，将B表中的元素合并到A表中，使新的A表中的元素仍保持非递减有序。高效是指最大限度避免移动元素。

算法思想：分别设立三个指针i、j和k，其中i、j为线性表A和B的工作指针，分别赋值为A和B的表尾元素的下标；k时结果线性表的工作指针，赋值为两表合并之后得到的新表尾的下标。对i和j所指的元素进行比较，将大者（假定为i所指）元素加入新表尾k处，同时，其指针（假定为i）和k一起向前移，i指针所指元素继续与j指针所指元素进行比较，直至一个表为空。若B表中还有元素剩余，则再将其剩余元素插入k处，此时j、k指针一起向前移动。算法的时间复杂度为$O(m+n)$。

```
template <class elemType>
bool seqList<elemType>::Union(seqList<elemType> &B){
    int m, n, k, i, just;
    m = this->curLength; // 当前对象为线性表A
    n = B.curLength; // m, n分别为线性表A和B的长度
    k = m + n - 1; // k为结果线性表的工作指针(下标)
    i = m-1, j = n-1; // i, j分别为线性表A和B的工作指针(下标)
    if (m + n > this->maxSize){resize();} // 判断表A的空间是否足够大，如果不够则需要扩大表空间
    while (i >= 0 && j >= 0){ // 合并顺序表，直到一个表为空
        if (data[i] >= B.data[j]){
            data[k--] = data[i--]; // 默认当前对象,this指针可省略
        }else{data[k--] = B.data[j--];}
    }
    while (j >= 0){ // 将表B中剩余元素复制到表A中
        data[k--] = B.data[j--]
    }
    curLength = m+n; // 修改表A长度
    return true;
}
```

顺序表的特点

1. 顺序表需要预先分配存储空间，很难恰当预留空间，分配大了会造成浪费，分配小了对有些运算会造成溢出。
2. 由于逻辑顺序和物理顺序的一致性，因此顺序表能够按照元素序号（下标）直接存取元素，具有随机存取的优点。
3. 由于要保持逻辑顺序和物理顺序的一致性，顺序表在进行插入、删除操作时需要移动大量的数据，因此顺序表的插入和删除效率低。
4. 改变顺序表的大小需要重新创建一个新的顺序表，把原表里的数据复制到新表中，然后释放原表空间。
5. 顺序表比较适合静态的、经常做定位访问的线性表。

线性表的链式表现和实现

以链式结构存储的线性表被称为线性链表，简称链表。链表克服了顺序表需要预先确定表长的缺点，可以充分利用计算机内存空间，实现灵活的动态内存管理。因为链表中的节点是在运行时动态申请和释放的，在插入元素时申请新的存储空间，在删除元素时释放其占有的内存空间。

链表是一种链式存取的数据结构，结点既存储数据元素本身的信息（数据域），又存储数据元素间的链接信息（地址域，也叫指针域），每个结点在运行时动态生成，存放在一个独立的存储单元中，结点间的逻辑关系依靠存储单元中附加的指针来给出。

线性链表的分类：

1. 结点只有一个地址域的链表，称为单链表。
2. 结点有两个地址域的链表，称为双链表。
3. 首尾相连的链表，称为循环链表。

线性表的链式表示

单链表的概念和术语说明如下：

1. 单链表是链式结构中最简单的一种，每个结点只包含一个指针，指向后继，所以称其为单链表。单链表的每个节点都是动态申请的，我们并不知道结点的名字，因此，单链表的任何操作都必须从第一个结点开始，采用“顺藤摸瓜”的方式，从第一个结点的地址域找到第二个结点，从第二个结点的地址域找到第三个结点，直到最后一个结点，其地址域为空，就是表尾。由此看出，单链表失去了顺序存储结构的随机存取的特点，定位运算需要$O(n)$的复杂度，要比顺序表慢，但是单链表的插入和删除操作不需要大量移动元素，只需要修改指针的指向即可，要比顺序表方便的多。
2. 头指针。单链表中第一个结点的地址存放在一个指针变量中，这个指针变量称为头指针。头指针具有标识一个单链表的作用，所以经常用头指针代表单链表的名字。
3. 首元结点是指单链表中存储线性表第一个元素的结点，也被称为第一元素结点。
4. 头节点。在整个单链表的第一个结点之前加入一个结点，称为头结点。它的数据域可以不存储任何信息（也可以作为监视哨或用于存放线性表长度等附加信息），指针域中存放的是首元结点的地址。当带有头节点的单链表为空时，头节点的指针域为`nullptr`，当不带头节点的单链表为空时，头指针为`nullptr`，表示空表。

为什么要引入头节点？为了运算方便统一。在a_i和a_{i+1}之间插入数据域为x的结点，需要申请新结点用于存放x，然后让新结点的指针域指向a_{i+1}，修改a_i的指针指向x。而在删除数据域为a_i的结点时，只需要修改a_{i-1}的指针域指向a_{i+1}即可。

若单链表中没有头节点，则在首元结点的前面插入一个数据域为x的结点，使得x称为新的首元结点；或者删除首元结点。这都将修改头指针head，与在其他结点位置上的插入、删除算法是不一致的。而头节点的类型与数据结点一致，标识链表的头指针head中存放头结点的地址，这样即使是空表，头指针head也不为空。头节点的加入使得插入和删除算法的差异不复存在，也使得空表和非空表的处理一致。

前面已知，链表结点的存储空间不是预先分配的，是在运行中根据需要动态申请的。那么，如何申请结点空间呢？假如单链表结点类型定义如下：

```
template <class elemType>
struct Node{
    elemType data; // 数据域
    Node* next; // 指针域
    Node(const elemType value, Node* p = nullptr){ // 两个参数的构造函数
        data = value;
        next = p;
    }
    Node(Node* p = nullptr){ // 一个参数的构造函数
        next = p;
    }
};
```

我们利用C++中的new和delete操作符给对象分配和释放空间，例如：

```
Node<elemType> *p = new Node<elemType>(value, nullptr);
```

该语句完成了两个操作：首先是申请一块`Node<elemType>`类型的存储单元，并为其数据域`data`赋值`value`，指针域`next`赋值`nullptr`；其次是将这块存储单元的首地址复制给指针变量`p`，若系统没有足够内存可用，则`new`在申请内存失败时抛出一个`bad_alloc exception`异常。`p`的类型为`Node<elemType>*`型，所以该无名结点可以用指针`p`间接引用，数据域为`(*p).data`或`p->data`，指针域为`(*p).next`或`p->next`。

下面给出单链表的类型定义：

```
template <class elemType>
class linkList : public List<elemType>{
private:
    struct Node{ // 结点类型
        elemType data; // 结点的数据域
        Node* next; // 结点的指针域
        Node(const elemType value, Node* p = nullptr){ // 两个参数的构造函数
            data = value;
            next = p;
        }
        Node(Node* p = nullptr){ // 一个参数的构造函数
            next = p;
        }
    };
    Node* head; // 单链表的头指针
    Node* tail; // 单链表的尾指针
    int curLength; // 单链表的当前长度，牺牲空间换时间
    Node* getPosition(int i)const; // 返回指向位序为i的结点的指针
public:
    linkList(); // 构造函数
    ~linkList(); // 析构函数
    void clear(); // 将单链表清空，使其成为空表
    bool empty()const{return head->next == nullptr;} // 带头结点的单链表，判空
    int size()const{return curLength;} // 返回单链表的当前实际长度
    void insert(int i, const elemType& value); // 在位序i处插入值为value的结点，表长+1
    void remove(int i); // 删除位序i处的结点，表长-1
    int search(const elemType& value)const; // 查找值为value的结点第一次出现的位序
```

```
    int prior(const elemType& value)const; // 查找值为value的结点的前驱的位序
    elemType visit(int i)const; // 访问位序为i的结点的值，0定位到首元结点
    void traverse()const; // 遍历单链表
    void headCreate(); // "头插法"创建单链表
    void tailCreate(); // "尾插法"创建单链表
    void inverse(); // 逆置单链表
};
```

单链表上基本运算的实现

单链表的初始化

算法思想：单链表的初始化就是创建一个带头结点的空链表。实际上就是申请一个新的结点作为头结点，无需设置头结点的数据域，只需设置其指针域为空即可，算法的时间复杂度为$O(1)$。

```
template <class elemType>
linkList<elemType>::linkList(){
    head = tail = new Node(); // 创建带有头节点的空表
    curLength = 0;
}
```

析构函数

当单链表对象脱离其作用域时，系统自动执行析构函数来释放单链表空间，算法的时间复杂度为$O(n)$

```
template <class elemType>
linkList<elemType>::~linkList(){
    clear(); // 清空单链表
    delete head; // 释放头结点
}
```

清空单链表

算法思想：清空单链表的主要操作是将工作指针从头结点一直移动到表尾，边移动指针边释放结点，因此，算法的时间复杂度为$O(n)$。在实际应用中，一般不修改头指针head的指向，而是引入一个工作指针来完成单链表的遍历操作。

```
template <class elemType>
void linkList<elemType>::clear(){
    Node *p, *tmp; // p为工作指针，指向首元结点
    p = head->next; // 引入工作指针是为了防止随意修改头指针
    while(p != nullptr){
        tmp = p;
        p = p->next; // 指针后移
        delete tmp;
    }
    head->next = nullptr; // 头结点的指针域置空
    tail = head; // 头尾指针均指向头结点
    curLength = 0;
}
```

求表长

```
template <class elemType>
int linkList<elemType>::size()const{
    return curLength; // 直接返回curLength
}
```

若单链表类型定义中没有设置变量curLength用于存储表长，就需要从第一个结点开始，一个结点一个结点地计数，直至表尾。为此，设置一个工作指针p和计数器count，每当p指针向后移动一次，计数器count就加1，直至表尾，算法的时间复杂度为$O(n)$。

```
template <class elemType>
int linkList<elemType>::size()const{
    Node* p = head->next;
    int count = 0;
    while(p){
        count++;
        p = p->next;
    }
    return count;
}
```

遍历单链表

算法思想：遍历单链表需要从头到尾访问单链表中的每个结点，并依次输入各结点的数据域，算法的时间复杂度为$O(n)$。

```
template <class elemType>
void linkList<elemType>::traverse()const{
    Node* p = head->next; // 工作指针p指向首元结点
    while (p != nullptr){
        cout << p->data << " ";
        p = p->next; // 向后移动指针
    }
}
```

查找位序为i的结点的内存地址

算法思想：合法的查找范围为[-1,curLength-1]。当i==0时，表示要查找的是首元结点；当i==-1时，表示要查找的是头结点。若i的值是非法的，则没有位序为i的结点，返回nullptr；否则，设一个移动工作指针p和计数器count，初始时p指向头结点，每当指针p移向下一个结点时，计数器count++，直到p指向位序为i的节点为止，返回p。算法的时间复杂度为$O(n)$
。

```
template <class elemType>
typename linkList<elemType>::Node* linkList<elemType>::getPosition(int i)const{
    if (i < -1 || i > curLength-1) // 合法查找范围为[-1,CurLength-1]
        return nullptr; // 当i非法时返回nullptr
    Node* p = head; // 工作指针p指向头结点
    int count = 0;
    while (count <= i){
        p = p->next;
        count++;
    }
    return p; // 返回指向位序为i的结点的指针
}
```

查找值为value的结点的位序

算法思想：查找值为`value`的结点的位序，需要设置计数器`count`，从单链表的第一个结点起，判断当前结点的值是否等于给定值`value`，若查找成功，则返回结点的位序，否则继续查找，直到单链表结束为止；若查找失败，则返回-1。算法的时间复杂度为$O(n)$。

```
template <class elemType>
int linkList<elemType>::search(const elemType& value)const{
    Node* p = head->next; // 工作指针p指向首元结点
    int count = 0; // 首元结点的位序为0
    while (p != nullptr && p->data != value){
        p = p->next;
        count++;
    }
    if (p == nullptr)return -1; // 查找失败返回-1，这里-1并非头结点
    else return count; // 查找成功，count为结点的位序
}
```

查找值为value的结点的前驱的位序

算法思想：求值为`value`的结点的前驱，需要从单链表的第一个结点开始遍历。我们设置两个指针`p`和`pre`，分别指向当前正在访问的结点和它的前驱，还需要一个计数器`count`。从单链表的第一个节点开始遍历：

1. 若`p == nullptr`，则查找值为`value的结点失败，返回-1
2. 若找到值为`value`的结点，且该结点是首元结点，则无前驱，返回-1
3. 若找到值为`value`的结点，且该结点不是首元结点，则返回其前驱的位序。

```
template <class elemType>
int linkList<elemType>::prior(const elemType& value)const{
    Node* p = head->next; // p是工作指针，指向首元结点
    Node* pre = nullptr; // pre指向p的前驱
    int count = -1; // 注意：-1表示首元结点无前驱
    while(p && p->data != value){
        pre = p; // 前驱指针后移
        p = p->next; // 指向下个待处理结点
        count++;
    }
    if(p == nullptr) return -1; // 查找失败返回-1，这里-1并非头结点
    else return count; // 查找成功，count为结点的位序
}
```

求某个结点的后继

算法思想：求值为`value`的结点的后继，从单链表的第一个结点开始查找，若查找成功，则查看该结点的指针域，若其指针域为空，说明该结点是尾节点，无后继；否则，结点的指针域指向其后继。若查找失败，则说明单链表中没有值为`value`的结点，更谈不上后继。上面已有算法，这里不再赘述。

插入结点

算法思想：在位序i处插入值为value的新结点q。因为单链表中的结点只有一个指向后继的指针，因此需要先找到位序为i-1的结点p，让q的指针域指向p原来的后继，然后修改p的后继为q。需要注意的是，不要因为修改指针而使得单链表断开。对于有curLength个结点的线性表，合法的插入范围是[0,curLength]，其中，0表示插入点在首元结点，curLength表示插入点在尾结点的后面。插入算法的主要操作是移动指针查找结点，因此算法的时间复杂度为$O(n)$。

```
template <class elemType>
void linkList<elemType>::insert(int i, const elemType& value){
    Node *p, *q;
    if (i < 0 || i > curLength) // 合法的插入范围为[0,curLength]
        throw outOfRange(); // 插入位置非法，抛出异常
    p = getPosition(i-1); // p是位序为i的结点的前驱
    q = new Node(value, p->next); // 申请新结点q，数据域为value，指针域为p->next
    p->next = question; // q结点插到p结点的后面
    if (p == tail) tail = q; // 若插入点在链表尾，则q成为新的尾节点
    curLength++;
}
```

若要求在时间复杂度为$O(1)$的前提下，将s所指的结点插在p所指的结点的前面，我们可以先将s节点插在p几诶单的后面，然后交换他们的数据域即可，语句序列如下：

```
s->next = p->next;
p->next = s; // s节点插到p的后面
tmp = p->data, p->data = s->data, s->data = tmp; // 交换结点s和p的数据域
```

删除结点

算法思想：删除位序为i的结点，对于有curLength个结点的单链表，合法的删除范围为[0,curLength-1]，其中，0表示删除首元结点，curLength-1表示删除尾结点。算法的关键是查找位序为i-1的结点，并修改指针的连接关系，算法的时间复杂度为$O(n)$。

```
template <class elemType>
void linkList<elemType>::remove(int i){
    Node *pre, *p; // p是待删结点，pre是其前驱
    if (i < 0 || i > curLength-1) // 合法的删除范围为[0,cueLength-1]
        throw outOfRange(); // 当待删结点不存在时，抛出异常
    pre = getPosition(i-1);
    p = pre->next; // p是真正的待删结点
    if (p == tail){ // 待删结点为尾节点，则修改尾指针
        tail = pre;
        pre->next = nullptr;
        delete p;
    }else{ // 修改指针并删除结点p
        pre->next = p->next;
        delete p;
    }
    curLength--;
}
```

头插法创建单链表

头插法是指在链表的头部插入结点建立单链表，也就是每次将新增结点插在头结点之后、首元结点之前。在构造函数中已经建立了具有头结点（指针域置空）的空链表，现在需要做的是将结点逐个插在头结点之后和首元结点之前（当然，在空表情况下插入的结点就是首元结点）。链表与顺序表不同，它是一种动态管理的存储结构，链表中的每个结点占用的存储空间不是预先分配的，而是运行时系统根据需求生成的。因此建立单链表从空表开始，每读入一个元素则申请一个结点，然后插在单链表的头部。算法的时间复杂度为$O(n)$。

```
template <class elemType>
void linkList<elemType>::headCreate(){
    Node *p;
    elemType value, flag;
    cout << "input elements, ended with:";
    cin >> flag; // 输入结束标志
    while (cin >> value, value != flag){
        p = new Node(value, head->next);
        head->next = p; // 结点p插在头结点的后面
        if (head == tail) tail = p; // 原链表为空，新结点p成为尾结点
        curLength++;
    }
}
```

尾插法创建单链表

头插法建立的单链表，输入元素的顺序与生成的单链表中元素的顺序是相反的。若希望输入元素的顺序与生成的单链表中元素的顺序一致，则用尾插法。尾插法是指在单链表的尾部插入结点建立单链表，单链表类`linkList`中的尾指针`tail`将派上用场。

算法思想：在初始状态时创建一个带有头节点的空链表，头指针`head`、尾指针`tail`都指向头结点。按线性表中元素的顺序依次读入元素并申请结点，将新结点插在`tail`所指结点的后面，然后`tail`指向新的尾节点。算法的时间复杂度为$O(n)$。

```
template <class elemType>
void linkList<elemType>::tailCreate(){
    Node *p;
    elemType value, flag;
    cout << "input elements, ended with:";
    cin >> flag; // 输入结束标志
    while (cin >> value, value != flag){
        p = new Node(value, nullptr);
        tail->next = p; // 结点p插在尾结点的后面
        tail = p; // 结点p成为新的尾结点
        curLength++;
    }
}
```

逆置单链表

算法思想：利用头插法建立的单链表，其中元素的顺序与读入的元素的顺序是相反的。因此，在本算法中用工作指针p依次访问单链表中的每个节点，每访问一个结点，就将它插在头结点的后面，然后向后移动工作指针p，知道所有节点都重新插入单链表中，就实现了单链表的逆置。算法的时间复杂度为$O(n)$。

```
template <class elemType>
void linkList<elemType>::inverse(){
    Node *p, *tmp;
    p = head->next; // p为工作指针指向首元结点
    head->next = nullptr; // 头结点的指针域置空，构成空链表
    if(p) tail = p; // 逆置后，原首元结点将变为尾节点
    while(p){
        tmp = p->next; // 暂存p的后继
        p->next = head->next;
        head->next = p; // 结点p插在头结点的后面
        p = tmp; // 继续处理下一个结点
    }
}
```

合并单链表

将非递减有序的单链表la和lb合并成新的非递减有序单链表lc，要求利用原表空间。

算法思想：因为新创建的单链表lc仍然是非递减有序的，所以用尾插法创建lc。设立三个工作指针，指针pa和pb分别指向单链表la和lb的首元结点，指针pc指向单链表lc的头结点，比较pa和pb的数据域，将小者（假设pa为小者）连接到lc的表尾，然后向后移动pa指针，继续比较pa、pb的数据域，直到其中一个表为空，将另一个表的剩余结点全部连接到lc的表尾。算法的时间复杂度为$O(m+n)$。

```
template <class elemType>
typename linkList<elemType> * linkList<elemType>::Union(linkList<elemType> *lb){
    Node *pa, *pb, *pc;
    linkList<elemType>* lc = this; // lc利用la的空间
    pa = head->next; head->next = nullptr; // la头结点的指针域置为nullptr，构成空链表
    pb = (lb->head)->next; (lb->head)->next = nullptr; // lb头结点的指针域置为nullptr，构成空链表
    pc = lc->head; // lc直接利用la的头结点
    while (pa && pb){ // 如果la和lb都非空
        if (pa->data <= pb->data){ // pa所指结点用尾插法插入lc中
            pc->next = pa;
            pc = pa;
            pa = pa->next;
        }else{ // pb所指结点用尾插法插入lc中
            pc->next = pb;
            pc = pb;
            pb = pb->next;
        }
    }
    if(pa){ // 若pa未到表尾，则将pc指向pa
        pc->next = pa;
        lc->tail = tail;
    }else{// 若pb未到表尾，则将pc指向pb
        pc->next = pb;
        lc->tail = lb->tail;
    }
    lc->curLength = curLength + lb->curLength;
    delete lb;
    return lc;
}
```

链表的特点

1. 不要求用地址连续的存储空间存储，每个结点在运行时动态生成。结点的存储空间在物理位置上可以相邻，也可以不相邻。
2. 插入和删除操作不需要移动结点，只需要修改指针，满足经常插入和删除结点的需求。
3. 链表不具备顺序表随机存取的优点。
4. 链表结点增加了指示元素间关系的指针域，空间开销比较大。

双链表

在单链表中，通过一个结点找到它的后继比较方便，其时间复杂度为$O(1)$。而要找到它的前驱，则很麻烦，只能从该链表的头指针开始沿着各结点的指针域进行查找，时间复杂度是$O(n)$。这是因为单链表的各结点只有一个指向其后继的指针域`next`，只能向后查找。如果某个链表需要经常进行查找结点前驱的操作，我们希望查找前驱的时间复杂度也达到$O(1)$，这时可以用空间换时间：即每个结点再增加一个指向前驱的指针域`prior`，使得链表可以进行双向查找，这种链表成为双向链表，简称双链表。双链表的结点类型定义如下：

```
template <class elemType>
struct Node{
    elemType data; // 数据域
    Node *prior, *next; // 指针域，两个指针分别指向前驱和后继
    Node(const elemType value, Node* p = nullptr, Node* n = nullptr){
        data = value;
        prior = p;
        next = n;
    }
    Node():next(nullptr), prior(nullptr){}
    ~Node(){}
};
```

假设p是指向双链表中某个结点的指针，则`p->prior->next`表示的是指向p的前驱的后继的指针，即p自身；类似的，`p->next->prior`表示的是指向p后继的前驱的指针，也是p自身。用代码表述如下：

```
p->prior->next == p;
p->next->prior == p;
```

为了运算的方便和统一，消除在表头和表尾插入、删除的边缘特殊情况，通常在实现双链表时除了设有“头结点”以外，还设有“尾部头结点”，其中，头结点的`prior`指针域为空，尾部头结点的`next`指针域为空。

双链表的插入和删除操作

双链表中结点的插入

设p是双链表中某结点，s是待插入的值为`value`的新结点，将s插在p的前面，这时不需要通过遍历该链表来查找p的前驱，因为p的前驱就是`p->prior`。插入结点s的主要语句序列如下：

```
s->prior = p->prior; // p原先的前驱成为s的前驱
p->prior->next = s; // s成为p原先的前驱的后继
s->next = p; // s的后继是p
p->prior = s; // 修改p的前驱为s
```

双链表中结点的删除

设p指向双链表中某个结点，删除p所指向的结点，其主要语句序列如下：

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

双链表的类型定义及实现

```
#include "List.h"
template <class elemType>
class double LinkList::public List<elemType>{
private:
    struct Node{
        elemType data; // 数据域
        Node *prior, *next; // 指针域，两个指针分别指向前驱和后继
        Node(const elemType value, Node* p = nullptr, Node* n = nullptr){
            data = value;
            prior = p;
            next = n;
        }
        Node():next(nullptr), prior(nullptr){}
        ~Node(){}
    };
    Node *head, *tail; // 头尾指针
    int curLength; // 双链表的当前长度
    Node* getPosition(int i)const; // 返回指向位序为i处结点的指针
public:
    doubleLinkList();
    ~doubleLinkList();
    void clear(); // 清空双链表，使之成为空表
    bool empty() const{return head->next == tail;} // 判空
    int size() const{return curLength;} // 求双链表的长度
    void insert(int i, const elemType& value); // 在位序i处插入值为value的结点，表长+1
    void remove(int i); // 删除位序i处的结点，表长-1
    int search(const elemType& value) const; // 查找值为value的结点的位序
    elemType visit(int i) const; // 访问位序为i的结点的值
    void traverse() const; // 遍历双链表
    void inverse(); // 逆置双链表
};
```

双链表的初始化

```
template <class elemType>
doubleLinkList<elemType>::doubleLinkList(){
    head = new Node; // 头指针指向头结点
    tail = new Node; // 尾指针指向尾部头结点
    head->next = tail;
    tail->prior = head;
    curLength = 0;
}
```

析构函数

```
template <class elemType>
doubleLinkList<elemType>::~doubleLinkList(){
    clear();
    delete head;
    delete tail;
}
```

清空双链表

```
template <class elemType>
void doubleLinkList<elemType>::clear(){
    // 清空操作时不再考虑结点的前驱域是否断链
    Node *p = head->next, *tmp;
    head->next = tail; // 头结点的后继时尾部头结点
    tail->prior = head; // 尾部头结点的前驱时头结点
    while (p != tail){
        tmp = p->next;
        delete p;
        p = tmp;
    }
    curLength = 0;
}
```

查找位序为i的结点的地址

```
template <class elemType>
typename doubleLinkList<elemType>::Node *doubleLinkList<elemType>::getPosition(int i) const{
    // 位序i的合法范围是[-1,curLength]，若i==-1，则定位到头结点，若i==curLength，则定位到tail指向的尾部头结点
    Node *p = head;
    int count = 0;
    if (i < -1 || i > curLength) return nullptr; // 当i非法时返回nullptr
    while (count <= i){
        p = p->next;
        count++;
    }
    return p; // 返回指向位序为i的结点的指针
}
```

查找值为value的结点的位序

```
template <class elemType>
int doubleLinkList<elemType>::search(const elemType& value) const{
    Node *p = head->next;
    int i = 0;
    while (p != tail && p->data != value){
        p = p->next;
        i++;
    }
    if (p == tail) return -1;
    else return i;
}
```

在位序i处插入值为value的结点

```
template <class elemType>
void doubleLinkList<elemType>::insert(int i, const elemType& value){
    Node *p, *tmp;
    if (i < 0 || i > curLength) // 合法的插入范围为[0,curLength]
        throw outOfRange(); // 插入位置非法，抛出异常
    p = getPosition(i); // 若i==curLength，则定位到tail指向的结点
    tmp = new Node(value, p->prior, p); // tmp节点插在p节点之前
    p->prior->next = tmp; // p原先的前驱的后继指向tmp
    p->prior = tmp; // 修改p的前驱为tmp
    ++curLength;
}
```

删除位序为i的结点

```
template <class elemType>
void doubleLinkList<elemType>::remove(int i){
    Node *p;
    if (i < 0 || i > curLength-1) // 合法的删除范围为[0,curLength-1]
        throw outOfRange(); // 当待删结点不存在时，抛出异常
    p = getPosition(i);
    p->prior->next = p->next;
    p->next->prior = p->prior;
    delete p;
    --curLength;
}
```

访问位序为i的结点的值

```
template <class elemType>
elemType doubleLinkList<elemType>::visit(int i) const{
    // visit不能直接用getPosition判断范围是否合法，因为其定位范围为[-1,cueLength]
    if (i < 0 || i > curLength-1) // 合法的访问范围为[0,cueLength-1]
        throw outOfRange(); // 当节点不存在时，抛出异常
    Node *p = getPosition(i);
    return p->data;
}
```

遍历双链表

```
template <class elemType>
void doubleLinkList<elemType>::traverse() const{
    Node *p = head->next;
    while (p != tail){
        cout << p->data << " ";
        p = p->next;
    }
    cout << endl;
}
```

逆置双链表

```
template <class elemType>
void doubleLinkList<elemType>::inverse(){
    Node *tmp, *p = head->next; // p是工作指针，指向首元结点
    head->next = tail; // 构成空双链表
    tail->prior = head;
    while (p != tail){
        tmp = p->next; // 保存p的后继
        p->next = head->next;
        p->prior = head; // p结点插到头结点的后面
        head->next->prior = p;
        head->next = p;
        p = tmp;
    }
}
```

循环链表

单循环链表

单链表只能从头结点开始遍历整个链表，若希望从任意一个结点开始遍历整个链表，则可以将单链表通过指针域首尾相连，即尾结点的指针域指向头结点，这样刑场的链表成为单循环链表。但循环链表带来的主要优点之一是：从其中任意一个结点开始都可以访问到其他结点。线性表的基本操作在单循环链表中的实现与在单链表中的实现类似。主要差别在于，在单链表中，用指针是否为`nullptr`来判断是否到表尾；而在单循环链表中，用指针是否等于头指针来判断是否到表尾。需要指出的是，但循环链表往往只设尾指针而不设头指针，用一个指向尾结点的尾指针来标识单循环链表。其好处是既方便查找尾结点又方便查找头结点，因为通过尾结点的指针域很容易找到头结点。如果尾指针`tail`指向尾结点，需要将其与尾部头结点区分开来，`tail->next`为头结点，而`tail->next->next`为首元结点。

双向循环链表

双链表也可以做成循环结构，其最后一个结点的后继域`next`指向头结点，头结点的前驱域`prior`指向最后一个结点。当带头结点的双向循环链表为空，头结点的前驱域`prior`和后继域`next`都指向自身。

线性表实现方法的比较

1. **顺序表的主要优点和缺点**

   主要优点如下：

   1. 实现方法简单，各种高级语言中都有数组类型，容易实现。
   2. 按序号查找可通过下标直接定位，时间代价为$O(1)$。
   3. 元素间的逻辑顺序和物理存储顺序一致，不需要借助指针，不产生结构性存储开销。
   4. 顺序表是存储静态数据的理想选择。

   主要缺点如下：

   1. 需要预先申请固定长度的数组。
   2. 插入和删除操作需要移动大量的元素，时间代价为$O(n)$。

2. **链表的主要优点和缺点**

   主要优点如下：

1. 插入和删除操作不需要移动元素，只需要修改指针的指向，时间代价为$O(n)$；若不考虑查找，则插入和删除操作时间代价为$O(1)$。链表比较适合经常插入和删除元素的情况。
2. 动态地按照需要为表中新的元素分配储存空间，无需事先了解线性表的长度。当对线性表的长度难以估计时，采用链表比较合适。
3. 链表是存储动态变化数据的理想选择。

主要缺点如下：

1. 链表需要在每个结点上附加指针，用以体现元素间的逻辑关系，增加了结构性存储开销。
2. 按序号查找元素需要遍历链表，时间代价为$O(n)$。

3. **如何为线性表选取合适的存储结构**

   1. 顺序表具有按元素序号随机访问的特点，在顺序表中按序号访问元素的时间复杂度为$O(1)$；而在链表中按序号访问的时间复杂度为$O(n)$。如果经常按序号访问元素，则使用顺序表优于链表。
   2. 在顺序表中做插入、删除操作时，平均需要移动大约表中一半的元素。当表中元素的信息量较大且表较长时，顺序表的插入和删除操作效率低。在链表中做插入、删除操作时，虽然也要查找插入位置，但是操作主要是比较运算。从这个角度考虑，显然链表优于顺序表。