第三章 线性表

刘杰

人工智能学院

南开大学 Nankai University

目录

线性表的定义

线性表：一个线性表（list）是由同类型数据元素构成的有序序列，记作:

$(a_0, a_1, \dots, a_{n-1})$。

- 例（2，6，8，3，9，4）是一个线性表，其中的数据元素是整数，按表中列出的顺序排列，表中共有6个数据元素。
- 例（A，B，C，D，E，…，X，Y，Z）是一个线性表，其中的数据元素是英文大写字母，按字母表的顺序排列，共有26个数据元素。

线性表的定义

- 学生入学成绩表也是一个线性表。其中的数据元素是每个学生在表中所占的一行，包括姓名、准考证号、性别、年龄和总分等共5个数据项。

姓名	准考证号	性别	年龄	总分
陈义平	030010023	男	18	589
陆东	030010025	男	19	568
王晓敏	030010037	女	18	574
⋮	⋮	⋮	⋮	⋮
李志强	030010123	男	19	617

数据元素

相关术语

- 当线性表中不包含任何元素时，称之为空表，记作（）。
- 表中具有元素的个数称为线性表的长度。
- 线性表的第一个元素称为表头，最后一个元素称为表尾。
- 对于表中第i个元素a_i来说，第$i-1$个元素a_{i-1}称为它的直接前驱（简称前驱），第$i+1$个元素a_{i+1}称为它的直接后继（简称后继）。当然，表头没有直接前驱，表尾没有直接后继。

线性表的典型操作

- `InitList(&L)`:初始化一个空的线性表
- `Length(L)`:求表长度
- `LocateElem(L,e)`:按值查找指定元素在的位置
- `ListInsert(&L,i,e)`:在表L中第i个位置插入指定元素
- `ListDelete(&L,i)`:删除表中第i个位置的元素
- `GetElem(L,i)`:获取表中第i个位置的元素
- `PrintList(L)`:按序输出线性表L的所有元素值
- `Empty(L)`:判断表L是否为空表
- `DestroyList(L)`:销毁线性表，释放内存空间。

实现线性表的必要性

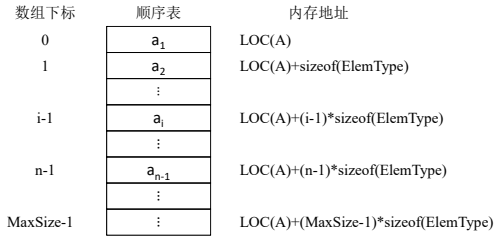
- 思考：
  - 数组是否等同线性表？
  - 是否可以简单地用数组代替线性表？

线性表的存储结构

- 线性表基本操作的实现依赖于线性表的存储结构，不同的存储结构各有其优劣，用以解决不同的问题场景。
- 常用的存储结构：
  - 顺序存储结构
  - 链式存储结构

线性表的顺序存储结构

- 基本思想：用一组连续的存储单元依次存放线性表中各个元素。在C++中，可以用一个一维数组来表示。



顺序表类的实现

- 由于线性表的长度可以改变，只用固定长度的数组是不足以表示线性表的实际情况的，线性表的当前长度也必须要用一个整数来记录。因此，线性表的顺序存储类型描述如下：

```
class list {// 基于数组的线性表类
private:
   int msize;// 表的最大长度
   int numinlist;// 表中元素的实际个数
   int curr;// 表中当前元素的位置
   ELEM* listarray;// 存储表元素的数组
```

```
public:
   list(const int sz = LIST_SIZE);  // 构造函数
   ~list();              // 析构函数
   void clear();          // 从表中清除所有元素
   void insert(const ELEM&);// 在当前位置插入一个元素
   ELEM remove();         // 删除并返回当前元素的值
      void setFirst();       // 置光标于第一个位置
   void prev();           // 移动光标到前一位置
   void next();           // 移动光标到下一位置
   void setPos(const  int);      // 置光标于指定位置
   void setValue(const  ELEM&);// 设置当前元素的值
   int  length() const;          // 返回表的当前长度
   ELEM currValue() const;       // 返回当前元素的值
   bool isEmpty() const;     // 如果表为空则返回TRUE
   bool isInList() const; // 如果光标在表内则返回TRUE
   bool find(const  ELEM&);  // 从当前位置开始寻找元素值
};
```

构造函数与析构函数

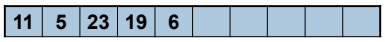
- 使用构造函数来初始化线性表，析构函数释放线性表，clear函数用于清空线性表中的元素。

```
list::list(const int sz) // 构造函数：初始化
{
   msize = sz;
   numinlist = curr = 0;
   listarray = new ELEM[sz];
}
list::~list()    // 析构函数：释放数组所占空间
{
   delete[] listarray;
}
```

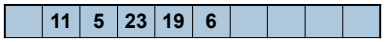
顺序结构插入元素示例

- 在线性表(11,5,23,19,6)的表头插入元素27:

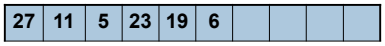
原线性表元素



对应位置元素后移



插入新元素27



线性表的插入代码实现

- 线性表的插入：先移动后面的元素然后完成插入赋值。

```
// 在当前位置插入元素item
void list::insert(const ELEM& item) {
   // 数组必须未满并且curr必须是一个合理位置
   assert((numinlist < msize) && (curr >= 0)
&& (curr <= numinlist));
   for (int i = numinlist; i>curr; i--)
      listarray[i] = listarray[i - 1];
   listarray[curr] = item;
   numinlist++;
}
```

顺序表的插入复杂度分析

- 设P_i是在第i个元素之前插入一个元素的概率，则在长度为n的线性表中插入一个元素时，所需移动的元素次数的平均次数为：

$$E_{is} = \sum_{i=1}^{n+1} P_i(n - i + 1)$$

若认为 $P_i = \frac{1}{n+1}$

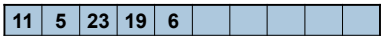
则 $E_{is} = \frac{1}{n+1}\sum_{i=1}^{n+1}(n - i + 1) = \frac{n}{2}$

$\therefore T(n) = O(n)$

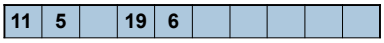
顺序结构删除元素示例

- 在线性表(11,5,23,19,6)中删除第三个元素23:

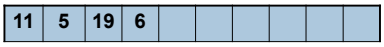
原线性表元素



删除元素23



23之后的元素前移



线性表的删除代码实现

- 线性表的删除：先获取要删除的元素，后面的元素依次前移。

```
// 删除并返回当前元素
ELEM list::remove() {
   assert(!isEmpty() && isInList());      // 要删除的
元素必须在表中
      ELEM temp = listarray[curr];     // 保存被删除的
元素
   for (int i = curr; i< numinlist - 1; i++) // 被删
元素后面的全部元素前移一个位置
      listarray[i] = listarray[i + 1];
   numinlist--;     // 表的当前长度减1
   return temp;
}
```

顺序表的删除复杂度分析

- 设Q_i是删除第i个元素的概率，则在长度为n的线性表中删除一个元素所需移动的元素次数的平均次数为：

$$E_{de} = \sum_{i=1}^{n} Q_i(n - i)$$

若认为 $Q_i = \frac{1}{n}$

则 $E_{de} = \frac{1}{n}\sum_{i=1}^{n}(n - i) = \frac{n-1}{2}$

$\therefore T(n) = O(n)$

顺序表上的其他操作

```
void list::clear()// 从表中清除所有元素
{   numinlist = curr = 0;}
void list::setFirst()  // 置光标于第一个位置
{   curr = 0; }
void list::prev()   // 移动光标到前一个位置
{   curr --; }
void list::next()   // 移动光标到下一个位置
{   curr++;}
int list::length() const   // 返回表的当前长度
{   return numinlist;}
void list::setPos(const int pos)     // 置光标于指定位置
{   curr = pos;}
void list::setValue(const ELEM& val) {//为当前元素ELEM赋值
    assert(isInList());
    listarray[curr] = val;
}
```

顺序表上的其他操作

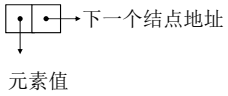
```
ELEM list::currValue() const {       // 返回当前元素的值
    assert(isInList())
       return  listarray[curr];
}
bool list::isEmpty() const    // 如果表为空则返回TRUE
{   return  numinlist == 0; }
bool list::isInList () const //如果当前位置在表中则返回TRUE
{   return  (curr >= 0)  && (curr < numinlist); }
bool list::find(const ELEM& val) // 从当前位置开始查找表元素
{
    while (isInList())
       if (currValue() == val)
          return  TRUE;// 如果找到则返回TRUE
       else next();
       return  FALSE;   // 如果找不到则返回FALSE
}
```

顺序表的特点

- 表中的元素存储在数组中相邻的位置，对表中任一元素的访问只需$O(1)$时间。
- 线性表中元素需要满足有序性，线性表的插入、删除操作需要移动大量元素。所需移动的元素次数的平均次数为$O(n)$

链式存储结构

- 链表
  - 链表是由一系列叫做结点（node）的对象连接起来的数据结构。一般来说，结点之间用指针来连接。
- 链表结点类(linkNode)由两部分组成：
  - 存储元素值的element域；
  - 存储结点指针的next域；

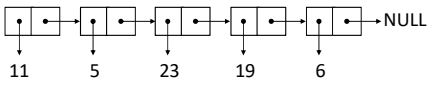


链表结点类的定义

```
//链表结点类的定义
class linkNode {    // 一个单链表结点
public:
    ELEM element;      // 该结点的数据元素的值
    linkNode *next;     // 链表中指向下一个结点的指针
    linkNode(const ELEM& elemval, linkNode* nextval
= NULL)       // 构造函数1
    {
        element = elemval;  next = nextval;
    }      // 给出数据元素ELEM的值
    linkNode(linkNode* nextval = NULL) { next =
nextval; }       // 构造函数2
    ~linkNode() { }         // 析构函数
};
```

链表类的定义

- 链表类由多个链表结点构成，不同于顺序表的连续存储，这些链表结点离散分布在存储空间中，通过链表结点的指针进行连接。
- 例：由(11,5,23,19,6)等元素构成的链表：



链表类的定义

- 链表类的数据元素包含链表的首尾结点以及当前结点的指针，通过指针来访问链表中的所有元素。

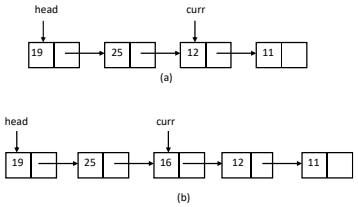
```
class list {                          // 单链表类
   private:
      linkNode* head;     // 指向链表头的指针
      linkNode* tail;     // 指向链表尾的指针
      linkNode* curr;     // 指向当前元素的指针
   public:
      list(const int sz = LIST_SIZE);  // 构造函数
      ~list();          // 析构函数
      void clear();     // 从表中清除所有元素
```

链表类的定义

```
      void insert(const ELEM&); // 在当前位置插入一个新元素
      ELEM remove();      // 删除并返回当前元素
      void setFirst();    // 置光标于第一个位置
      void next();        // 移动光标到下一位置
      void prev();        // 移动光标到前一位置
      void setPos(const int);    // 置光标于指定位置
      void setValue(const ELEM&);   // 给当前位置的元素赋值
      int length() const;    // 返回表的当前长度
      ELEM currValue() const;    // 返回当前元素的值
      bool isEmpty() const;  // 如果表为空则返回TRUE
      bool isInList() const; // 如果光标在表内则返回TRUE
      bool find(const ELEM&);//从当前位置开始寻找某个元素
};
```

示例---- 链表的插入

- 在结点12前插入元素16：



(a) 插入16之前　(b)插入16之后的应有结果

插入新元素的步骤

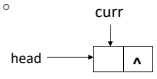
- (1).创建一个新的结点，并且赋给它一个值；
- (2).新结点的next值要指向当前结点；
- (3).当前结点的前驱结点的next指针要指向新插入的结点。

- 关键点：在单链表的情况下，需要从表头开始遍历，找到当前结点的前驱结点，这样做平均要花费$O(n)$的时间（n为表的当前长度）。

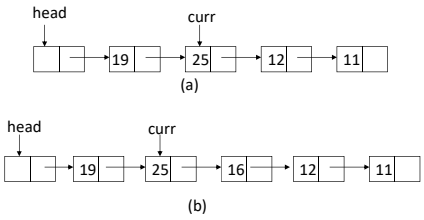
新的问题以及解决方法

新的问题：如果表中只有一个元素，那么curr所要指向的前驱结点就不存在了。因此表中只有一个结点或没有结点的时候就属于难于处理的特殊情况。

解决的办法：是在链表中增加一个表头结点。这个结点是链表的第一个结点，它和表中其他结点的结构一样，只是不存放任何数据元素。

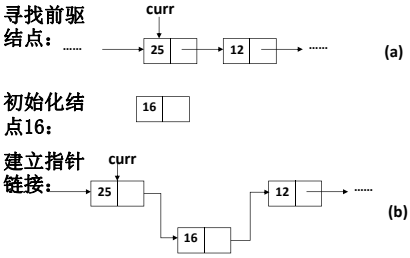


带表头链表的插入



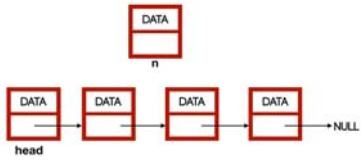
(a)插入数值16之前的链表　(b)插入数值16之后的应有结果

插入一个结点的过程



(a)插入前的链表 (b)插入后的链表

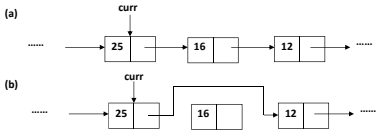
插入操作的实现




```
//在当前位置插入元素item
void list::insert(const ELEM& item) {
   assert(curr != NULL);
   curr->next = new linkNode (item, curr-
>next);
}
```

从链表中删除结点

从链表中删除一个结点只需要先用一个指针指向被删结点，然后将curr所指结点的next直接指向被删结点的后继结点即可。




```
linkNode* ltemp = curr->next;
curr->next = ltemp->next;
```

(a) 删除（结点16）前的链表 (b) 删除后的链表

删除操作的实现

删除过程也只需要O (1)的时间。

```
//删除当前位置的元素
ELEM list::remove() {
    assert(isInList());
    ELEM temp = curr->next->element;
    linkNode* ltemp = curr->next;
    curr->next = ltemp->next;  //更新链接关系
    if (tail == ltemp) tail = curr;
    delete ltemp;        //删除链表结点
    return temp;
}
```

清空表

链表的清空：逐步释放所有链表结点。

```
void list::clear() {   // 从表中删除所有元素
   while (head->next != NULL) {
      // 返回链表的所有结点到自由空间（保留表头结点）
      curr = head->next;
      head->next = curr->next;
      delete curr;
   }
   curr = head;
}
```

其他链表操作

```
//构造析构
list::list(const int sz)  // 构造函数
{
    head = curr = new linkNode;
}
list::~list() {   // 析构函数
    while (head != NULL) {  // 释放链表的所有节点占用的空间
        curr = head;
        head = head->next;
        delete curr;
    }
}
void list::setFirst() //置curr为链表头
{
    curr = head;
}
```

其他链表操作

```
//置curr为下一个位置
void list::next()
{
   if (curr != NULL)
      curr = curr->next;
}
//置curr为上一个位置
void list::prev() {
   linkNode* temp = head;
   if ((curr == NULL) || (curr == head))
   {  curr = NULL;  return;  }
   while ((temp != NULL) && (temp->next != curr)
      temp = temp->next;
   curr = temp;
}
```

其他链表操作

```
int list::length() const {  //计算链表长度
   int cnt = 0;
   for (linkNode* temp = head->next; temp != NULL; temp =
temp->next)
      cnt++;
   return cnt;
}

void list::setPos(const int pos) { //使用pos来设定curr位置
   curr = head;
   for (int i = 0; (curr != NULL) && (i<pos); i++)
      curr = curr->next;
}
void list::setValue(const ELEM& val) //设定结点的值
{
   assert(isInList());  curr->element = val;
}
```

其他链表操作

```
ELEM list::currValue() const // 返回当前结点的值
{
   assert(isInList());
   return  curr->next->element;
}
bool list::isEmpty() const  // 判断表是否为空
{
   return head->next == NULL;
}
bool list::isInList() const  // 如果光标在表内则返回
TRUE
{
   return (curr != NULL) && (curr->next != NULL);
}
```

链表查找

find方法是从curr开始查找val值在表中最先出现的地方。

```
//find方法是从curr开始查找val值
bool list::find(const ELEM& val) {
   while (isInList())
      if (cuur->next->element == val)
         return TRUE;
      else curr = curr->next;
      return FALSE;
}
```

顺序表与链表的比较

	空间效率	时间效率
顺序表	存储每个数据元素时没有浪费空间。	访问：在顺序表中是直接定位的，所需的操作次数是O(1)； 插入和删除：平均时间和最差时间均为O(n)；
链表	每个结点上除存储数据元素外，还要存放一个指针，这个指针一般称为结构性开销。	访问：单链表不能直接访问上述元素，只能从表头开始逐个查找，直到找到第i个结点为止。平均时间和最差时间均为O(n)； 插入和删除：链表的insert和remove操作所需时间仅为O(1)；