

线性表的逻辑结构知识点

线性表的定义	<ul style="list-style-type: none"><li>线性表是n个系统数据元素的有限序列</li><li>线性表示一种数据结构/逻辑结构</li></ul>
线性表的特点	<ul style="list-style-type: none"><li>表中元素的个数有限</li><li>表中元素具有逻辑上的顺序性，表中元素有其先后顺序。</li><li>表中元素都是数据元素，每个元素都是单个元素</li><li>表中元素的数据类型都相同， 这意味着每个元素占有相同大小的存储空间</li><li>表中元素具有抽象性， 即仅讨论元素间的逻辑关系， 而不考虑元素究竟表示什么内容</li></ul>
线性表的主要操作	<ul style="list-style-type: none"><li>InitList(&amp;L): 初始化表，构造一个空的线性表</li><li>Length(L): 求表长，返回线性表L的长度，即L中数据元素的个数</li><li>LocateElem(L,e): 按值查找操作。在表L中查找具有给定关键字值的元素</li><li>GetElem(L,i): 按位查找操作。获取表L中第i个位置的元素的值。</li><li>ListInsert(&amp;L,i,&amp;e): 插入操作。在表中的第i个位置插入指定元素e</li><li>ListDelete(&amp;L,i,&amp;e): 删除操作。删除表中第i个位置的元素，并用e返回删除元素的值</li><li>PrintList(L): 输出操作。按前后顺序输出线性表的所有元素值。</li><li>DestroyList(&amp;L): 销毁操作。销毁线性表，并释放空间。</li></ul>

线性表的存储结构知识点

线性表的顺序表示

1. 顺序表相关概念

定义	<ul style="list-style-type: none"><li>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</li><li>从而使得逻辑上相邻的两个元素在物理位置上也相邻</li></ul>
特点	<ul style="list-style-type: none"><li>表中元素的逻辑顺序与其物理顺序相同</li><li>是一种存储结构</li><li>随机访问，即通过首地址和元素序号能在时间O(1)内找到指定的元素</li><li>顺序表的存储密度高，每个节点只存储数据元素</li><li>顺序表逻辑上相邻的元素物理上也相邻，所以插入和删除需要移动大量元素</li></ul>

2. 静态分配和动态分配

	静态分配	动态分配
定义	<ul style="list-style-type: none"><li>数组的大小和空间事先已经固定</li><li>一旦空间占满，再加入新的数据就会产生溢出，进而导致程序崩溃</li></ul>	<ul style="list-style-type: none"><li>存储数组的空间是在程序执行过程中通过动态存储分配语句分配的</li><li>一旦数据空间占满，就另外开辟一块更大的存储空间，用以替换原来的存储空间</li></ul>
顺序存储类型	<pre>#define MaxSize 50 //定义线性表的最大长度 typedef struct {     ElemType data[MaxSize]; //顺序表的元素     int length; //顺序表当前长度 } SqList; //顺序表的类型定义</pre>	<pre>#define InitSize 100 //表长度的初始定义 typedef struct {     ElemType *data; //指示动态分配数组的指针     int Maxsize, length; //数组的最大容量和当前个数 } SeqList; //动态分配数组顺序表的类型定义</pre>

3. 动态分配语句

C的初始动态分配语句	L.data = (ElemType *)malloc(sizeof(ElemType) * InitSize);
C++的初始动态分配语句	L.data = new ElemType[InitSize];

4. 顺序表相关操作

	代码	最好情况	最坏情况	平均情况
插入	<pre>bool ListInsert(SqList &amp;L, int i, ElemType e) {     if (i &lt; 1    i &gt; L.length + 1) //判断i的范围是否有效         return false;     if (L.length &gt;= Maxsize) //当前存储空间已满，不能插入         return false;     for (int j = L.length; j &gt;= i; j--) //将第1个元素及以后的元素后移         L.data[j] = L.data[j - 1];     L.data[i - 1] = e; //在位置i处放入e     L.length++;     return true; }</pre>	O(1)	O(n)	O(n)
删除	<pre>bool ListDelete(SqList &amp;L, int i, ElemType &amp;e) {     if (i &lt; 1    i &gt; L.length + 1) //判断i的范围是否有效         return false;     e = L.data[i - 1]; //将被删除的元素赋值给e     for (int j = i; j &lt; L.length; j++) //将第i个元素以后的元素前移         L.data[j - 1] = L.data[j];     L.length--; //线性表长度减1     return true; }</pre>	O(1)	O(n)	O(n)
按值查找	<pre>int LocateElem(SqList &amp;L, int i, ElemType &amp;e) {     int i;     for (i = 0; i &lt; L.length; i++)         if (L.data[i] == e)             return i + 1; //下标为i的元素值等于e，返回其位序i+1     return 0; //退出循环，说明查找失败 }</pre>	O(1)	O(n)	O(n)

线性表的链式表示

1. 单链表相关概念

定义	<ul style="list-style-type: none"><li>单链表是线性表的链式存储</li><li>通过一组任意的存储单元来存储线性表中的数据元素</li></ul>
特点	<ul style="list-style-type: none"><li>通过“链”建立起数据元素之间的逻辑关系</li><li>插入和删除操作不需要移动元素</li><li>只需要修改指针，但也会失去顺序表可随机存取的优点</li></ul>
头结点和头指针的区别	<ul style="list-style-type: none"><li>不管带不带头结点，头指针都始终指向链表的第一个节点</li><li>头结点是带头结点的链表中的第一个节点，节点内通常不存储信息</li></ul>
引入头结点的优点	<ol style="list-style-type: none"><li>由于第一个数据节点的位置被存放在头结点的指针域中，因此在链表的第一个位置上的操作和在表的其他位置上的操作一致，无需进行特殊处理</li><li>无论链表是否为空，其头指针都是指向头结点的非空指针（空表中头结点的指针域为空），因此空表和非空表的处理也就得到了统一</li></ol>

2.单链表上基本操作的实现

	1. 采用头插法建立单链表	2. 采用尾插法建立单链表	3. 按序号查找节点值
代码	<pre>LinkList List_HeadInsert(LinkList &amp;L) { //逆向建立单链表     LNode *s;     int x;     L = (LinkList)malloc(sizeof(LNode)); //创建头结点     L-&gt;next = NULL; //初始为空链表     scanf("%d", &amp;x); //输入节点的值     while (x != 9999)     { //输入9999表示结束         s = (LNode *)malloc(sizeof(LNode)) //创建新节点         s-&gt;data = x;         s-&gt;next = L-&gt;next;         L-&gt;next = s; //将新节点插入表中，L为头指针         scanf("%d", &amp;x);     }     return L; }</pre> <p>插入的时间为O(1) 总时间复杂度为O(n)</p>	<pre>LinkList List_TailInsert(LinkList &amp;L) { //正向建立单链表     int x;     L = (LinkList)malloc(sizeof(LNode));     LNode *s, *r = L; // r为表尾指针     scanf("%d", &amp;x); //输入节点的值     while (x != 9999)     { //输入9999表示结束         s = (LNode *)malloc(sizeof(LNode)) //创建新节点         s-&gt;data = x;         r-&gt;next = s;         r = s; // r指向新的表尾节点         scanf("%d", &amp;x);     }     r-&gt;next = NULL; //尾节点指针置空     return L; }</pre> <p>时间复杂度为O(n)</p>	<pre>LNode *GetElem(LinkList L, int i) {     int j = 1; //计数，初始为1     LNode *p = L-&gt;next; //第1个节点指针赋给p     if (i == 0)         return L; //若i=0，则返回头结点     if (i &lt; 1)         return NULL; //若i无效，则返回NULL     while (p &amp;&amp; j &lt; i)     { //从第1个节点开始找，查找第i个节点         p = p-&gt;next;         j++;     }     return p; //返回第i个节点的指针，若i大于表长，则返回NULL }</pre> <p>时间复杂度为O(n)</p>
	4. 按值查找表节点	5. 插入节点操作	6. 删除节点操作
代码	<pre>LNode *LocateElem(LinkList L, ElemType e) {     LNode *p = L-&gt;next;     while (p != NULL &amp;&amp; p-&gt;data != e)         //从第1个节点开始查找data域为e的节点         p = p-&gt;next;     return p; }</pre>	<pre>p = GetElem(L, i - 1); //查找插入位置的前驱结点 s-&gt;next = p-&gt;next; p-&gt;next = s;</pre>	<pre>p = GetElem(L, i - 1); //查找删除位置的前驱结点 q = p-&gt;next; //令q指向被删除节点 p-&gt;next = q-&gt;next; //将*q节点从链中断开 free(q) //释放节点的存储空间</pre>
	7. 求表长操作（不含头结点）		
代码	<ul style="list-style-type: none"><li>设置一个计数器，每访问一个节点，计数器加1，直到访问到空节点为止</li></ul>		

3.双链表

为什么引入双链表?	<ul style="list-style-type: none"><li>单链表的缺点：只能从头结点依次顺序地向后遍历。</li><li>为解决这个问题引入双链表：其中有两个指针prior和next，分别指向其前驱节点和后继节点</li></ul>
双链表的插入操作	<pre>s-&gt;next = p-&gt;next; //将节点*s插入到*p之后 p-&gt;next-&gt;prior = s; s-&gt;prior = p; p-&gt;next = s</pre>
双链表的删除操作	<pre>p-&gt;next = q-&gt;next; q-&gt;next-&gt;prior = p; free(q)</pre>

4.循环链表

循环单链表	<ul style="list-style-type: none"><li>最后一个节点的指针不是NULL，而是改为指向头结点，从而整个链表形成一个环</li><li>可以从任意一个节点开始遍历整个链表</li><li>仅设置尾指针</li></ul>
循环双链表	<ul style="list-style-type: none"><li>最后一个节点的next指针指向头结点。头结点的prior指针指向最后一个节点</li></ul>

5.静态链表

定义	<ul style="list-style-type: none"><li>借助数组来描述线性表的链式存储结构</li><li>节点也有数据域data和指针域next</li><li>但这里的指针是节点的相对位置（数组下标），又称游标</li></ul>
结构描述	<pre>#define Maxsize 50 typedef struct {     //定义单链表节点类型     ElemType data; //数据域     int next; //下一个元素的数组下标 } SLinkList[Maxsize];</pre>

顺序表和链表的比较（👉之后时间复杂度没写最好或者最差的话，就是表示平均复杂度）

	顺序表	链表
读取方式	能随机存取	不能随机存取
逻辑结构与物理结构	相邻	不一定相邻
按值查找操作时间复杂度	无序为O(n) 有序可采用折半查找O(log <sub>2</sub> n)	O(n)
按序号查找	O(1)	O(n)
插入操作时间复杂度	O(n)	O(1)
删除操作时间复杂度	O(n)	O(1)
空间分配	需要预先按需分配存储空间	可以在需要时申请分配，只要内存有空间就可以分配

应该怎样选取存储结构？

	顺序表	链表
基于存储的考虑	难以估计时不宜用顺序表	链表不用估计，但存储密度较低
基于运算的考虑	若经常按序号访问，选择顺序表	经常插入、删除则选择链表
基于环境的考虑	顺序表容易实现	链表是基于指针的