* 目的：实现线性表抽象数据类型。
* 要求：掌握顺序、链式两种存储结构实现线性表。
* 重点：顺序表，单链表，循环双链表。
* 难点：使用“对象引用”方式实现链式存储结构。

## 1、线性表抽象数据类型

线性表（Linear List）（a0,a1,…,an-1）是由n（n>=0）个类型相同的数据元素组成的有限序列，其中，元素类型可以是基本类型或类；n是线性表长度（Length），即元素个数。

若n=0，空表；

若n>0，（0<i<n-1）有且仅有一个前驱元素和一个后继元素，a0没有前驱元素，an-1没有后继元素。

ADT List<T> {

boolean isEmpty() //判断线性表是否为空

int size() //返回线性表长度

T get(int i) //返回第i个元素

void set(int i, T x) //设置第i个元素为x

String toString() //所有元素的描述字符串

int insert(int i, T x) //插入x作为第i个元素

int insert(T x) //在线性表最后插入x元素

T remove(int i) //删除第i个元素

void clear() //删除所有元素

int search(T key) //查找与key相等元素

boolean contains(T key) //判断是否包含key元素

T remove(T key) //删除与key相等元素

boolean equals(Object obj) //比较是否相等

void addAll(List<T> list) //添加list所有元素

}

## 2、线性表的顺序存储结构和实现



### 2.1线性表的顺序存储结构

1）数组是实现顺序存储结构的基础。

* 数组（Array）存储具有相同数据类型的元素结合。
* 一维数组占用一块连续内存空间，每个存储单元的地址是连续的，计算第i个元素地址所需时间复杂度是一个常量，与元素序号i无关。
* 存取任何一个元素的时间复杂度是O(1)的数据结构称为随机存取结构。因此数组是随机存取结构。

2）顺序表

采用一维数组存储数据元素，数据元素在内存的物理存储次序反映了线性表数据元素之间的逻辑次序。

顺序表是随机存储结构。



### 2.2顺序表（Sequential List）

1）顺序表类声明、存储操作及效率分析

public class SeqList<T> //顺序表类，泛型类

{

protected Object[] element; //对象数组

protected int n; //元素个数（长度）

public SeqList(int length) //构造空表

public SeqList() //构造方法重载

{ this(64); //调用本类指定参数列表的构造方法

}

public SeqList(T[] values) //由values数组提供元素

public boolean isEmpty() //判断顺序表是否空

public int size() //返回元素个数

}

SeqList<T>类设计说明：

1. 泛型类

String values[]={"A","B","C","D","E"};

SeqList<String> lista = new SeqList<String>(values);

//lista引用顺序表实例，元素是String对象



1. 隐藏成员变量
2. 构造方法
3. 析构方法
4. 对象引用赋值
5. 存取操作，当指定元素序号不正确时的处理原则

public T get(int i) //返回第i个元素，0≤i<n。若i越界，返回null

public void set(int i, T x)//设置第i个元素为x，0≤i<n。若i越界，抛出序号越界异常；若x==null，抛出空对象异常

list.get(-1).toString()

//get()方法返回null时抛出空对象异常

1. 遍历输出及运行时多态

//返回顺序表所有元素的描述字符串，形式为“(,)”。

//覆盖Object类的toString()方法

public String toString() {

String str=this.getClass().getName()+"(";//返回类名

if (this.n>0)

str += this.element[0].toString();

for (int i=1; i<this.n; i++)

str += ", "+this.element[i].toString();

//执行T类的toString()方法，运行时多态

return str+") "; //空表返回()

}

1. 操作的效率分析
   1. isEmpty()、size()、get(i)、set(I,x)方法，时间复杂度为O(1)。
   2. toString()方法需要遍历顺序表，事件复杂度为O(n)。

2）顺序表插入操作

int insert(int i, T x) //插入x作为第i个元素

int insert(T x) //尾插入x元素。重载





若数组满，则扩充顺序表的数组容量。



3）顺序表删除操作

T remove(int i) //返回删除第i（0≤i<n）个元素

void clear() //删除线性表所有元素



4）顺序表查找操作

顺序查找算法：

//顺序查找首次出现的与key相等元素，返回元素序号i；查找不成功返回-1。

public int search(T key) {

for (int i=0; i<this.n; i++)

if (key.equals(this.element[i]))

//执行T类的equals(Object)，运行时多态

return i;

return -1; //空表或未找到时

}

5）顺序表的浅拷贝与深拷贝

类的拷贝构造方法声明格式如下，方法名同类名，参数为本类对象。没有默认。

类(类 对象) //拷贝构造方法

{

this.成员变量 = 参数对象.成员变量; //逐域赋值，以参数的实例值初始化当前实例

}

顺序表的浅拷贝：

public SeqList(SeqList<T> list) //浅拷贝构造方法

{

this.n = list.n; //int整数赋值，复制了整数值

this.element = list.element;

//数组引用赋值，两个变量共用一个数组，错误

}

顺序表的浅拷贝及其错误：

SeqList<String> listb = new SeqList<String>(lista);

lista.remove(0);



顺序表的深拷贝：

复制数组



复制对象



6）顺序表比较相等

两个线性表相等是指，它们各对应元素相等并且长度相同。

比较两个顺序表对象是否相等：



7）顺序表小结

顺序表的静态性很好，动态性很差。

* 顺序表元素的物理存储顺序直接反映线性表元素的逻辑顺序，顺序表是一种随机存取结构。get()、set()方法事件复杂度是O(1)。
* 插入和删除操作效率很低。如果在各位置插入元素的概率相同，则有：



### 2.3排序顺序表

排序顺序表是指，各数据元素按照关键字值递增或递减排列。

排序线性表是一种特殊的线性表。

线性表类型的应用

例1：已知集合A和B，求两个集合的并集，使A = A ∪ B，且B不再单独存在。

分析：以线性表LA和LB分别表示集合A和集合B，对集合B中的所有元素一个一个地检查，将存在于线性表LB中而不存在于线性表LA中的数据元素插入到线性表LA中去。

具体操作步骤为：

从线性表LB中取出一个数据元素

依值在线性表LA中进行查询；

若不存在，则将它插入到LA中。

重复上述三步直至LB为空表停止。

对应的线性表基本操作

ListDelete(LB, 1, e)；

LocateElem(LA, e, equal());

ListInsert(LA, n+1, e);

void union(List &LA, List &LB) {

La\_len = ListLength(LA); //求得线性表LA长度

while(!ListEmpty(LB)) { //依次处理LB中元素直至LB为空表

// 从LB中删除第一个数据并赋值给e

ListDelete(LB, 1, e);

// 当LA中不存在和e值相同的数据元素时进行插入

if(!LocateElem(LA, e, equal()))

ListInsert(LA, ++La\_len, e);

}

DestroyList(LB); //销毁线性表LB

}

例2：已知一个“非纯集合”B，试构造一个集合A，使A中只包含B中所有值各不相同的数据元素。

分析：将每个从B中取出的元素和已经加入到集合A中的元素相比较。

{ //构造线性表LA，使其只包含LB中所有值不相同的数据元素,算法不改变线性表LB  
 InitList(LA); //创建一个空的线性表LA  
 La\_len=0;  
 Lb\_len=ListLength(LB); //求线性表LB的长度  
 for(i=1;i<=Lb\_len;i++){ //处理LB中每个元素  
 GetElem(LB,i,e); //取LB中第i个数据赋给e  
 //当LA中没有和e值相同的数据元素时进行插入

if(!LocateElem(LA,e,equal())  
 ListInsert(LA,++La\_len,e);

}//for

}//purge

例3：判断两个集合是否相等。

分析：首先判别两者的表长是否相等；在表长相等的前提下，对于一个表中的所有元素，是否都能在另一个表中找到和它相等的元素。

如果“集合”中的元素不能保证都相异，还应反过来判别LB中每个元素都能在LA中找到相同者。

2.2线性表的顺序存储结构

线性表的顺序存储结构是指用一组连续的存储单元依次存储线性表中的每个元素。

L为每个数据元素所占据的存储单元数目。

位置的计算公式为：LOC(ai+1) = LOC(a1)+(i-1)\*L



2.2.1顺序存储结构特点

利用数据元素的存储位置表示线性表中相邻数据元素之间的前后关系，即线性表的逻辑结构与存储结构（物理结构）一致。

在访问线性表时，可以利用上述给出的数学公式，快速地计算出任何一个数据元素的存储地址。

//--------------------------线性表的动态分配顺序存储结构------------------------------

#define LIST\_INIT\_SIZE 100 //线性表存储空间的初始分配量

#define LISTINCREMENT 10 //线性表存储空间的分配增量

typedef struct {

EntryType \* item; //存储空间基址

int length; //线性表的当前长度

int listsize; //当前分配的存储容量(以sizeof(ElemType)为单位)

}SQ\_LIST；

2.2.2典型操作的算法实现

初始化线性表L

int InitList(SQ\_LIST&L) {

L->item=(EntryType\*) malloc(LIST\_MAX\_LENGTH \*sizeof(EntryType)); //分配空间

if(L->item==NULL) return ERROR; //若分配空间不成功，返回ERROR

L->length=0; //将当前线性表长度置0

return OK; //成功返回OK

}

销毁线性表L

void DestroyList(SQ\_LIST&L)

{

if(L->item) free(L->item); //释放线性表占据的所有存储空间

}

清空线性表L

void ClearList(SQ\_LIST&L)

{

L->length=0; //将线性表的长度置为0

}

求线性表的长度

int GetLength(SQ\_LIST L)

{

return(L.length);

}

判断线性表L是否为空

int IsEmpty(SQ\_LIST L)

{

if(L.length==0) return TRUE;

else return FALSE;

}

获取线性表L中的某个数据元素的内容

int GetElem(SQ\_LIST L,int i,EntryType&e)

{

if(i<1||i>L.length) return ERROR; //判断i值是否合理，若不合理，返回ERROR

e=L.item[i-1]; //数组中第i-1的单元存储着线性表中第i个数据元素的内容

return OK;

}

在线性表L中检索值为e的数据元素

int LocateELem(SQ\_LIST L,EntryType e)

{

for(i=0;i<L.length;i++)

if(L.item[i]==e) return i+1;

return 0;

}

在线性表L中第I个数据元素之前插入数据元素e

int ListInsert(SQ\_LIST&L,int i,EntryType e)

{ //检查是否有剩余空间

if(L->length==LIST\_MAX\_LENGTH) return ERROR;

if(i<1||i>L->length+1) return ERROR; //i值是否合理

for(j=L->length-1;j>=i-1;i++) //将线性表第i个元素之后的所有元素向后移动

L.->item[j+1]=L->item[j];

L->item[i-1]=e; //将新元素的内容放入线性表的第i个位置

L->length++;

return OK;

}

将线性表L中第i个数据元素删除

int ListDelete(SQ\_LIST&L,int i,EntryType&e)

{

if(IsEmpty(L)) return ERROR; //检测线性表是否为空

if(i<1||i>L->length) return ERROR; //检查i值是否合理

e=L->item[i-1] ; //将欲删除的数据元素内容保留在e所指示的存储单元中

for(j=i;j<=L->length-1;j++) //将线性表第i+1个元素之后的所有元素向前移动

L->item[j-1]=L->item[j];

L->length--;

return OK;

}

2.3线性表的链式存储结构

线性表顺序存储结构的特点：

它是一种简单、方便地存储方式。它要求线性表的数据元素依次存放在连续的存储单元中，从而利用数据元素的存储顺序表示相应的逻辑顺序，这种存储方式属于静态存储形式。

暴露的问题：

——在做插入或删除元素的操作时，会产生大量数据的移动；

——对于长度变化较大的线性表，要一次性地分配足够的存储空间，但这些空间常常又得不到充分的利用；

——线性表的容量难以扩充。

2.3.1链式存储结构

链式存储结构是指用一组任意的存储单元（可以连续，也可以不连续）存储线性表中的数据元素。

为了反映数据元素之间的逻辑关系，对于每个数据元素不仅要表示它的具体内容，还要附加一个表示它的直接后继元素存储位置的信息。

假设有一个线性表（a,b,c,d）

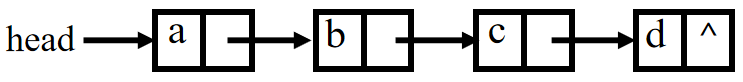


每个数据元素的两部分信息组合在一起被称为结点

其中表示数据元素内容的部分被称为数据域（data）

表示直接后继元素存储地址的部分被称为指针或指针域（next）

单链表简化的图形描述形式



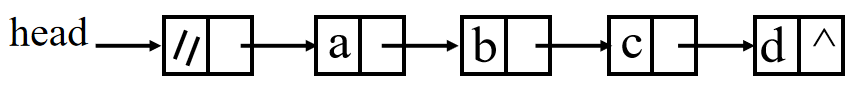
head是头指针，它指向单链表中的第一个结点，这是单链表操作的入口

由于最后一个结点没有直接后继结点，所以，它的指针域放入一个特殊的值NULL。

NULL值在图中常用（^）符号表示。

2.3.2带头结点的单链表

为了简化对链表的操作，人们经常在链表的第一个结点之前附加一个结点，并称为头结点。这样可以免去对链表第一个结点的特殊处理。



2.3.3链式存储结构的特点

线性表中的数据元素在存储单元中的存放顺序与逻辑顺序不一定一致

在对线性表操作时，只能通过头指针进入链表，并通过每个节点的指针域向后扫描其余节点，这样就会造成寻找第一个结点和寻找最后一个结点所花费的时间不等，具有这种特点的存取方式被称为顺序存取方式。

2.3.4链式存储结构的类型定义

typedef strcut node

{ //结点类型

EntryType item;

struct node \*next;

}NODE;

typedef struct

{ //链表类型

NODE \*head;

}LINK\_LIST;

2.3.5典型操作的算法实现

初始化链表L

int InitList(LINK\_LIST&L)

{

L->head=(\*NODE)malloc(sizeof(NODE));

//为头结点分配存储单元

if(L->head) {

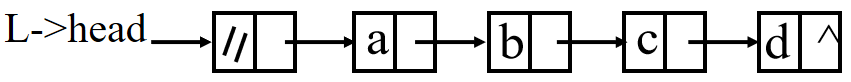
L->head->next=NULL;

return OK;

} else

return ERROR;

}



销毁链表L

void DestoryList(LINK\_LIST&L)

{

NODE\*p;

while(L->head) { //依次删除链表中的所有结点

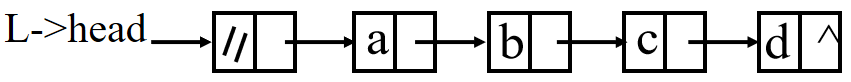
p=L->head;

L->head=L->head->next;

free(p);

}

}



清空链表L

void ClearList(LINK\_LIST&L)

{

NODE\*p;

while(L->head->next) {

p=L->head->next; //p指向链表中头结点后面的第一个结点

L->head->next=p->next; //删除p结点

free(p); //释放p结点占据的存储空间

}

}

求链表L的长度

int ListLength(LINK\_LIST L)

{

NODE\*p;

int len;

for(p=L.head,len=0;p->next==NULL;p=p->next,len++);

return(len);

}

判断链表L是否为空

int IsEmpty(LINK\_LIST L)

{

if(L.head->next==NULL) return TRUE;

else return FALSE;

}

通过e返回链表L中第i个数据元素的内容

void GetElem(LINK\_LIST L,int i,EntryType&e)

{

NODE\*p;

int j;

if(i<1||i>ListLength(L)) exit ERROR;

//检测i值的合理性

for(p=L.head,j=0;j!=i;p=p->next,j++);

//找到第i个结点

e=p->item;

//将第i个结点的内容赋给e指针所指向的存储单元中

}

在链表L中检索值为e的数据元素

NODE\*LocateELem(LINK\_LIST L,EntryType e)

{

NODE\*p;

for(p=L.head->next;p&&p->item!=e;p=p->next);

//寻找满足条件的结点

return(p);

}

返回链表L中结点e的直接前驱结点

NODE\*PriorElem(LINK\_LIST L,NODE\*e)

{

NODE\*p;

if(L.head->next==e) return NULL;

//检测第一个结点

for(p=L.head;p->next&&p->next!=e;p=p->next);

if(p->next==e) return p;

esle return NULL;

}

返回链表L中结点e的直接后继结点

NODE\*NextElem(LINK\_LIST L,NODE\*e)

{

NODE\*p;

for(p=L.head->next;p&&p!=e;p=p->next);

if(p) p=p->next;

return p;

}

在链表L中第i和数据元素之前插入数据元素e

int ListInsert(LINK\_LIST&L,int i,EntryType e)

{

NODE\*p,\*s;

int j;

if(i<1||i>ListLength(L)+1) return ERROR;

s=(NODE\*)malloc(sizeof(NODE));

if(s==NULL) return ERROR;

s->item=e;

for(p=L->head,j=0;p&&j<i-1;p=p->next,j++);

//寻找第i-1个结点

s->next=p->next;p->next=s; //将s结点插入

return OK;

}

将链表L中第i个数据元素删除，并将其内容保存在e中

int ListDelete(LINK\_LIST&L,int i,EntryType&e)

{

NODE\*p\*s;

int j;

if(i<1||i>ListLength(L)) return ERROR;

//检查i值的合理性

for(p=L->head,j=0;j<i-1;p=p->next,j++);

//寻找第i-1个结点

s=p->next; //用s指向将要删除的结点

e=s->item;

p->next=s->next; //删除s指针所指向的结点

free(s);

return OK;

}