# 2.3线性表的链式存储结构（续）

DS/SS

## 循环链表

- **特点：**

  在单链表基础上，让表中最后一个结点的指针域指向头结点，整个链表形成一个环。

- **作用：**

  从表中任一结点出发可以找到表中其他结点，从而提高整个表的平均查找效率。与单链表相比，平均可提高三分之一的时间。

- **操作：**

  与单链表基本一致

# 2.3线性表的链式存储结构（续）

DS/SS

## 双向链表

- 特点：

  在单链表基础上，增加一个指针域，让一个指针域指向当前结点的直接前驱，一个指向当前结点的直接后继。

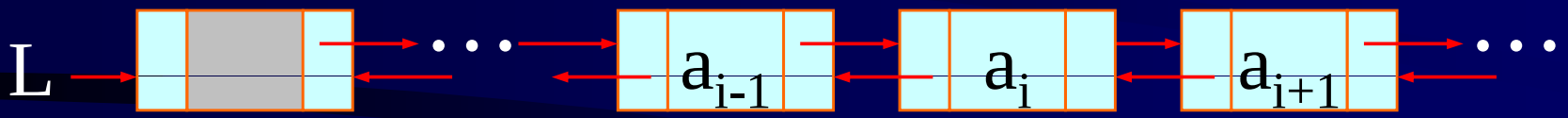
- 作用：

  从映射关系上讲，元素之间的关联关系增加，从表中任一结点出发可以沿着两个方向找到表中其他结点，从而提高整个表的平均查找效率。与单链表相比，平均可提高二分之一的时间。

# 2.3线性表的链式存储结构（续）

DS/SS

## 双向链表的数据类型描述




```
//----------用结构指针描述-----------
typedef  struct  DuLNode{
        ElemType          data      //数据域
        struct DuLNode    *prior    //前驱指针域
        struct DuLNode    *next     //后继指针域
} DuLnode, *DuLinkList
```

# 2.3线性表的链式存储结构（续）

## 双向链表上插入运算的实现（一）

$(a_1, \ldots, a_i, a_{i+1}, \ldots, a_n)$ 表长为n

ListInsert(&L, i ,e)

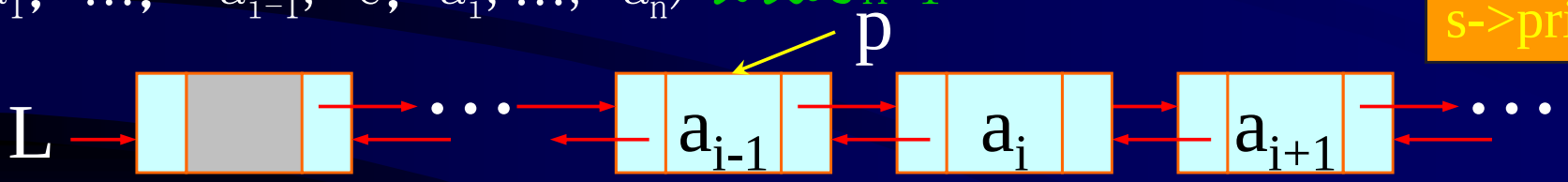
$(a_1, \ldots, a_{i-1}, e, a_i, \ldots, a_n)$ 表长为n+1

s->next = p->next

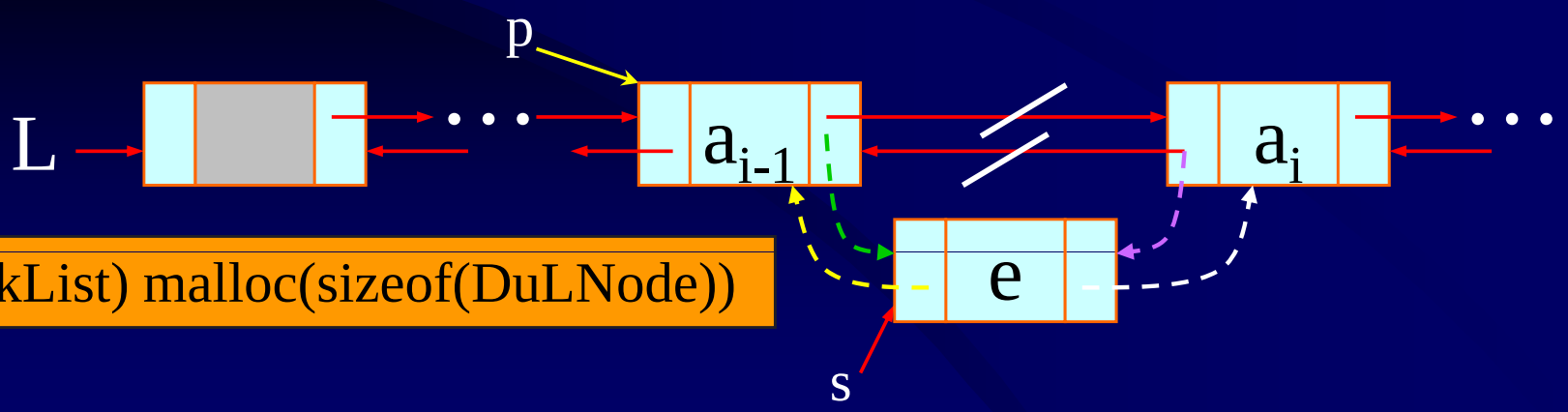
p->next->prior = s

p->next = s

s->prior = p




插入




s=(DuLinkList) malloc(sizeof(DuLNode))

# 2.3线性表的链式存储结构（续）

DS/SS

## 双向链表上插入运算的实现（二）

Status DuListInsert_DuL(DuLinkList &L, int i, ElemType e) {//在带头结点的双向链表L的第i个元素之前插入

第一步：判断参数是否合法合理，否则出错；

第二步：在物理空间中找到插入位置(仅仅在i-1处么？)；

第三步：插入前的准备工作;

第四步：插入；

} //DuListInsert_DuL

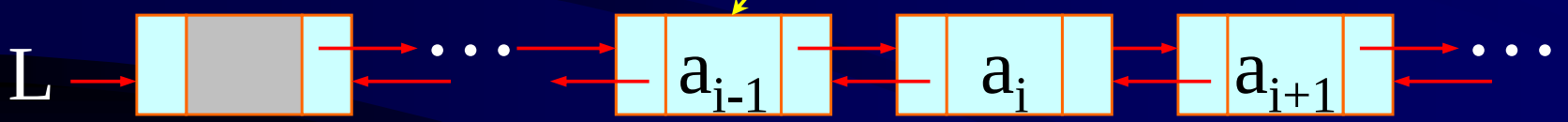
# 2.3线性表的链式存储结构（续）

## 双向链表上删除运算的实现（一）

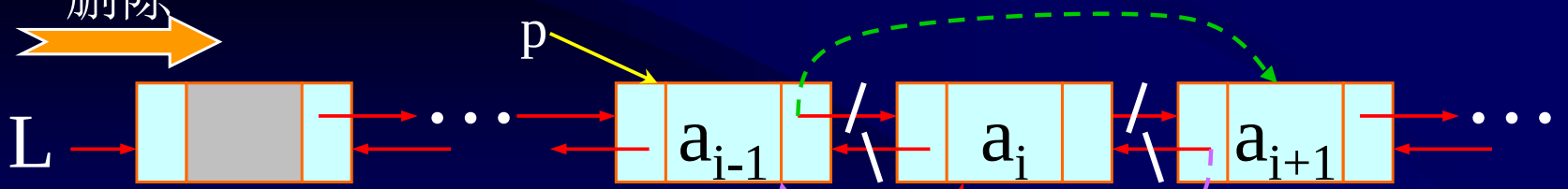
$(a_1, \ldots, a_i, a_{i+1}, \ldots, a_n)$ 表长为n

ListDelete(&L, i ,&e)

$(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$ 表长为n-1




删除




p->next->next->prior=p

p->next=p->next->next

# 2.3线性表的链式存储结构（续）

DS/SS

## 双向链表上删除运算的实现（二）

```
Status DuListDelete_DuL(DuLinkList &L, int i, ElemType &e) {
    第一步：判断参数是否合法合理，否则出错；
    第二步：在物理空间中找到删除位置(仅仅在i-1处么？)；
    第三步：删除；
    第四步：删除后的善后工作
} // DuListDelete_DuL
```

# 2.3线性表的链式存储结构（续）

## 双向循环链表

- **特点：**

  在双向链表基础上，让表中最后一个结点的后继指针域指向头结点，让头结点的前驱指针域指向最后一个结点，整个链表形成一个环。

- **作用：**

  从映射关系上讲，元素之间的关联关系增加，从表中任一结点出发可以沿着两个方向找到表中其他结点，从而提高整个表的平均查找效率。与单链表相比，平均可提高三分之二的时间。

# 2.3线性表的链式存储结构（续）

## 静态链表

- 特点：

  用一维数组来描述线性表，但数据元素的相邻关系仍然借助于额外的数组下标号（游标）来描述，处理手段仍然是链式存储结构的思想。

- 作用：

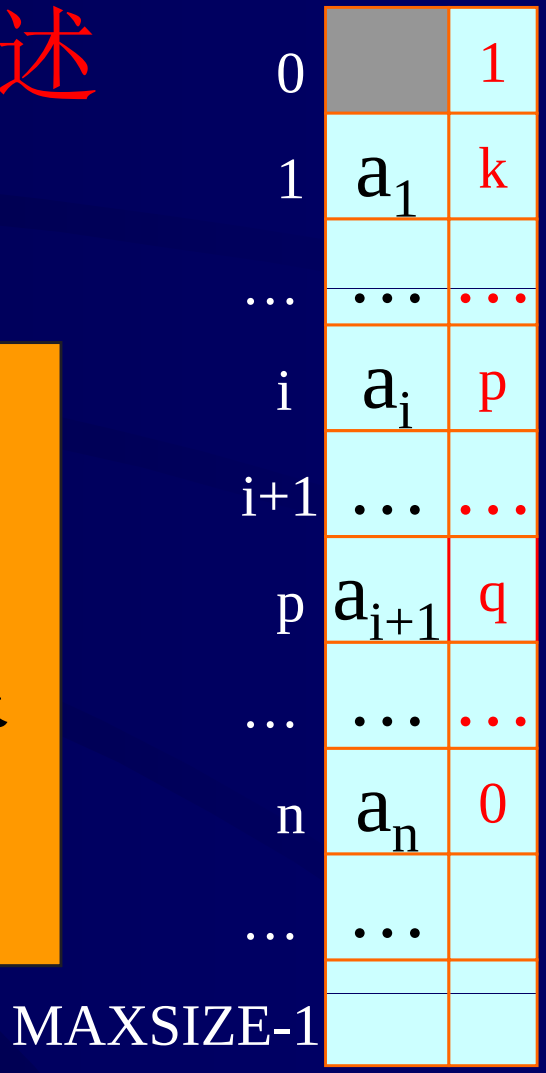
  有的高级语言不支持指针类型，但有些应用需要用链式存储。

# 2.3线性表的链式存储结构（续）

## 静态链表的数据类型描述

$(a_1, \ldots, a_i, a_{i+1}, \ldots, a_n)$

```
//----线性表的静态单链表存储结构---
#define  MAXSIZE          1000
typedef  struct {
             ElemType          data       //数据域
             int          cur          //游标域
} component,  SLinkList[MAXSIZE]
```



# 2.3线性表的链式存储结构（续）

## 静态链表基本运算的实现（一）

- **查找**：

  与单链表上元素的查找过程类似，沿着游标方向不断求元素后继，从而找到待检索元素位置。

- **插入**：

  与单链表上元素的插入过程类似，需要修改游标，维护元素在逻辑存储上的线性关系。但申请元素时，不能用malloc，需要采用其他手段。

- **删除**：

  与单链表上元素的删除过程类似，需要修改游标。但释放被删除元素所在结点时，不能用free，需要采用其他手段。

# 2.3线性表的链式存储结构（续）

## 静态链表基本运算的实现（二）

- 申请与释放空间的做法：

将所有未被使用过以及被删除的分量用游标链成一个备用链表（仍然是静态链表），插入时申请空间从备用链表中取（相当于在备用链表中删除元素）；删除时释放空间到备用链表中去（相当于在备用链表中插入元素）。

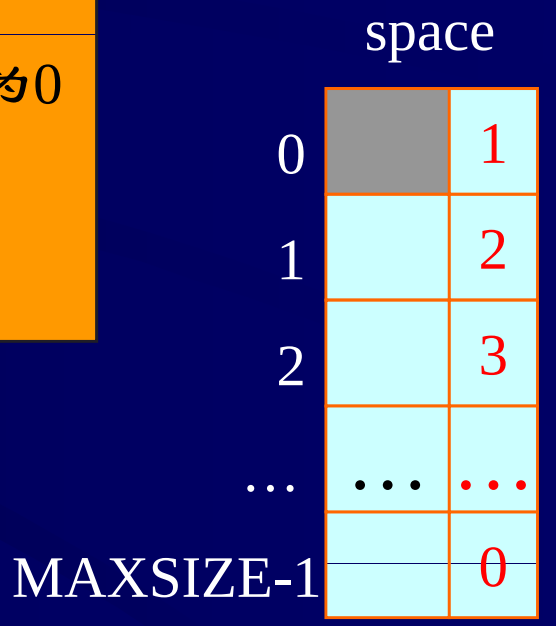
具体而言，备用链表中插入和删除的位置都在备用链表头结点后的那一个位置。

# 2.3线性表的链式存储结构（续）

DS/SS

## 静态链表基本运算的实现（三）

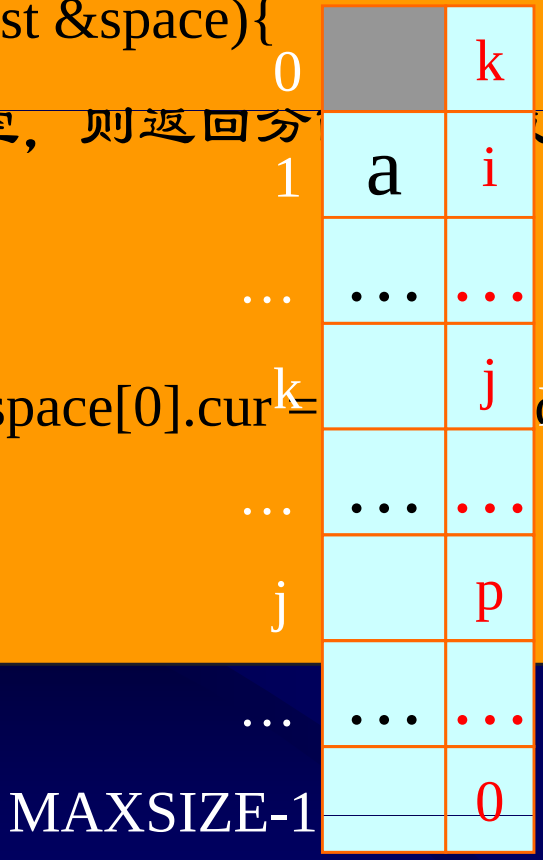
```
Void InitSpace_SL(SLinkList &space){
    //space[0].cur为备用链表头指针,空指针表示为0
    for(i=0; i<MAXSIZE-1; ++i) space[i].cur = i+1
     space[MAXSIZE-1].cur = 0;}//InitSpace_SL
```



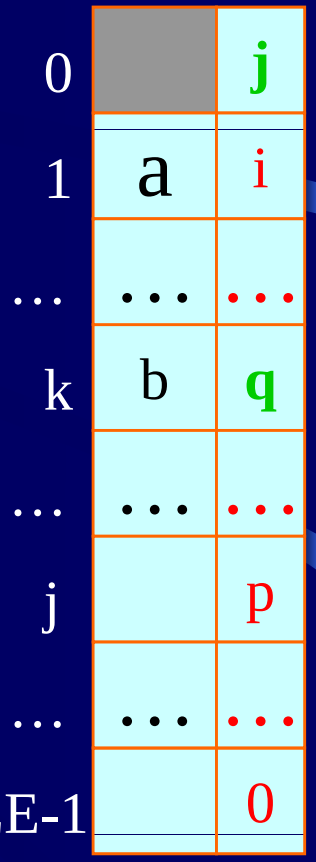
# 2.3线性表的链式存储结构（续）

## 静态链表基本运算的实现（四）

```
int Malloc_SL(SLinkList &space){
        //若备用链表非空，则返回分配的结点下标，
否则返回0
        i=space[0].cur ;
        if (space[0].cur) space[0].cur= space[i].cur;
        return i;
}//Malloc_SL
```



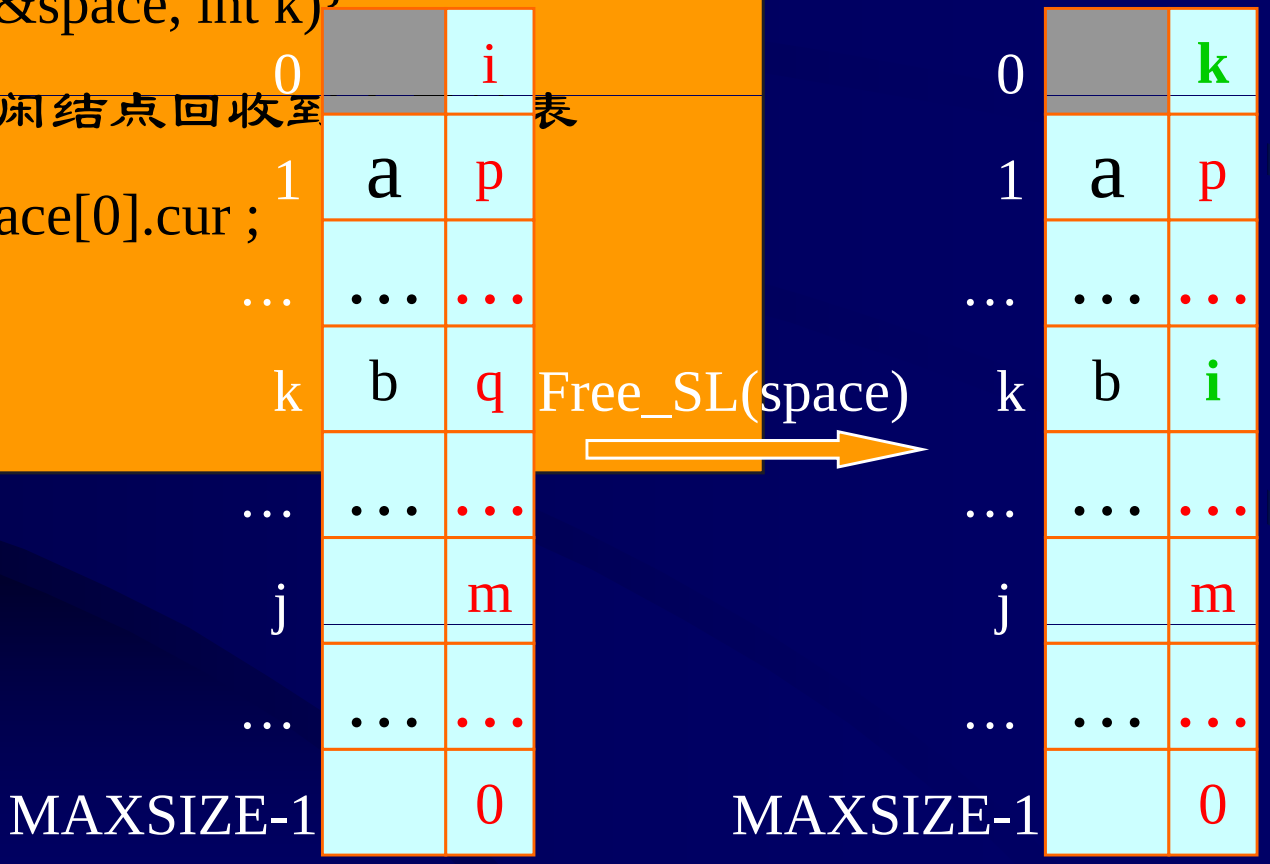

Malloc_SL(space)



# 2.3线性表的链式存储结构（续）

## 静态链表基本运算的实现（五）

```
int Free_SL(SLinkList &space, int k){
        //将下标为k的空闲结点回收到
        space[k].cur = space[0].cur ;
        space[0].cur = k;
}//Free_SL
```



# 2.3线性表的链式存储结构（续）

DS/SS

## 静态链表基本运算的实现（六）

例：求（A-B)U(B-A)，用S静态链表存储集合A，结果也在S中

```
void difference(SLinkList &space, int &S){
        InitSpace_SL(space);            //初始化备用空间
        S = Malloc_SL(space);           //生成S的头结点
        r = S;          //r指向S的当前最后结点
        ……
}// difference
```

# 2.3线性表的链式存储结构（续）

## 静态链表基本运算的实现（七）

上例中，设

A=(c,b,e,g,f,d),

B=(a,b,n,f)