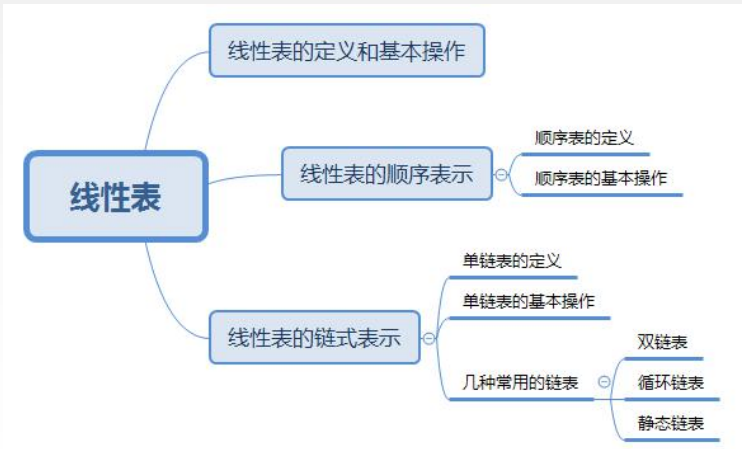
王道考研——数据结构

WWW.CSKAOYAN.COM

第二章 线性表

本章总览



本节内容

线性表

线性表
定义&基本操作

线性表的定义

线性表是具有相同类型的n（n>=0）个元素的有限序列，其中n为表长，当n=0时，该表为空表。

若L命名为线性表，则一般表示为

$L = (a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n)$

线性表的定义

线性表的特点：

表中元素个数**有限**

表中元素具有逻辑上的顺序性，在序列中各个元素排序有其**先后次序**

表中元素都是**数据元素**，每个元素都是单个元素

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

表中元素具有**抽象性**，即讨论元素间一对一的逻辑关系，而不考虑元素究竟表示的内容

线性表是一种**逻辑结构**，表示元素之间一对一相邻的关系

线性表的基本操作

线性表的九种基本操作：

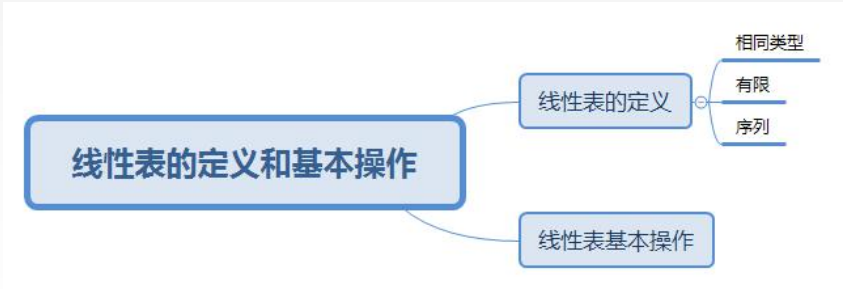
InitList(&L): 初始化表。构造一个空的线性表。
DestroyList(&L): 销毁操作。销毁线性表，并释放线性表L所占用的内存空间。

LocateElem(L,e): 按值查找操作。在表中L查找具有给定关键字值得元素。
GetELem(L,i): 按位查找操作。获取表L中第i个位置的元素的值。

ListInsert(&L,i,e): 插入操作。在表L中的第i个位置上插入指定元素e。
LIstDelete(&L,i,&e): 删除操作。删除表L中第i个位置的元素，并用e返回删除元素的值。

PrintList(L): 输出操作。按前后顺序输出线性表L的所有元素值。
Empty(L): 判空操作。若L为空表，则返回TRUE,否则返回FALSE。
Length(L): 求表长。返回线性表L的长度，即L中数据元素的个数。

本节回顾



本节内容

线性表

线性表
顺序表示

顺序表的定义

线性表的顺序存储又称**顺序表**

逻辑顺序与物理顺序相同

一组地址连续存放的存储单元依次存放线性表的元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻。

顺序表的定义

$L = (a_1, a_2, \ldots, a_i, a_{i+1}, \ldots, a_n)$

数组下标	顺序表	内存地址
0	a_1	LOC(A)
1	a_2	LOC(A)+sizeof(ElemType)
2	a_3	LOC(A)+2*sizeof(ElemType)
	....	
i-1	a_i	LOC(A)+(i-1)*sizeof(ElemType)
	...	
n-1	a_n	LOC(A)+(n-1)*sizeof(ElemType)
	...	
MaxSize-1	...	LOC(A)+(MaxSize-1)*sizeof(ElemType)

顺序表的定义

数组静态分配

```
#define MaxSize 50
typedef struct{
    ElemType data[MaxSize];
    int length;
}SqList;
```

数组动态分配

```
#define MaxSize 50
typedef struct{
    ElemType *data;
    int length;
}SqList;
```

动态分配语句

C L.data = (Elemtype*)malloc(sizeof(ElemType)*InitSize);

C++ L.data = new ElemType[InitSize];

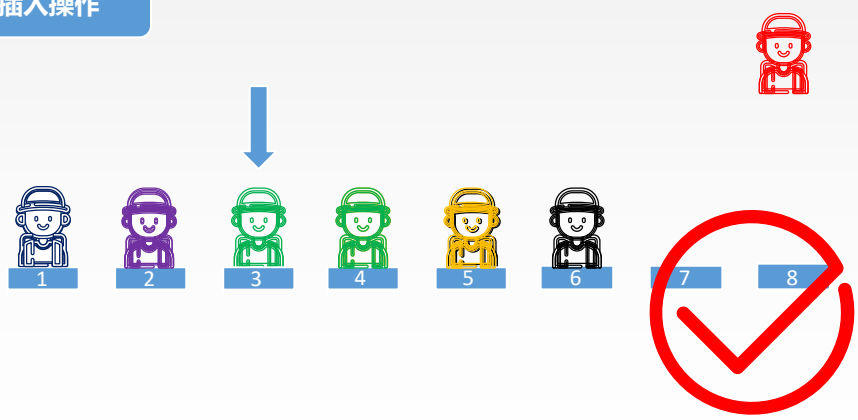
本节内容

线性表

顺序表
基本操作

顺序表的定义

插入操作



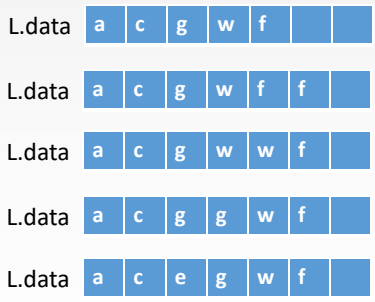
顺序表的定义

数组下标	顺序表
0	a_1
1	a_2
2	a_3
	...
i-1	a_i
	...
n-1	a_n
MaxSize-1	

```
bool ListInsert(SqList &L, int i, ElemType e){
    if(i<1||i>L.length+1)
        return false;
    if(L.length>=MaxSize)
        return false;
    for(int j=L.length;j>=i;j--)
        L.data[j]=L.data[j-1];
    L.data[i-1]=e;
    L.length++;
    return true;
}
```

顺序表的定义

```
ListInsert(L, 3, 'e');
```

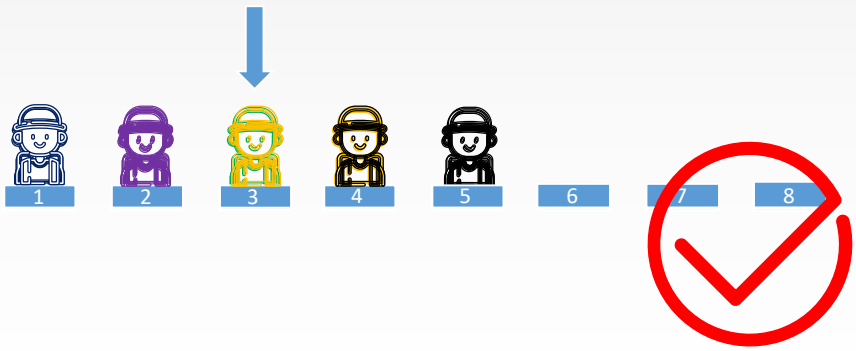



```
bool ListInsert(SqList &L, int i, ElemType e){
    if(i<1||i>L.length+1)
        return false;
    if(L.length>=MaxSize)
        return false;
    for(int j=L.length;j>=i;j--)
        L.date[j]=L.data[j-1];
    L.data[i-1]=e;
    L.length++;
    return true;
}
```

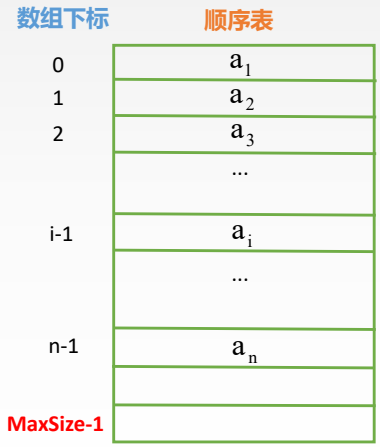
O(n) O(n)

顺序表的定义

删除操作



顺序表的定义

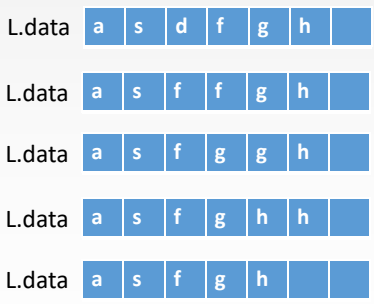



```
bool ListDelete(SqList &L, int i, ElemType &e){
    if(i<1||i>L.length)
        return false;
    e=L.data[i-1];
    for(int j=i;j<L.length;j++)
        L.data[j-1]=L.data[j];
    L.length--;
    return true;
}
```


顺序表的定义

```
ListDelete(L, 3, e);
```

e




```
bool ListDelete(SqList &L, int i, ElemType &e){
    if(i<1||i>L.length)
        return false;
    e=L.data[i-1];
    for(int j=i;j<L.length;j++)
        L.data[j-1]=L.data[j];
    L.length--;
    return true;
}
```

O(1) O(n) O(n)

顺序表的定义

```
ListDelete(L, 3, e);
```

e

L.data a s d f g h

L.data a s f f g h

L.data a s f g g h

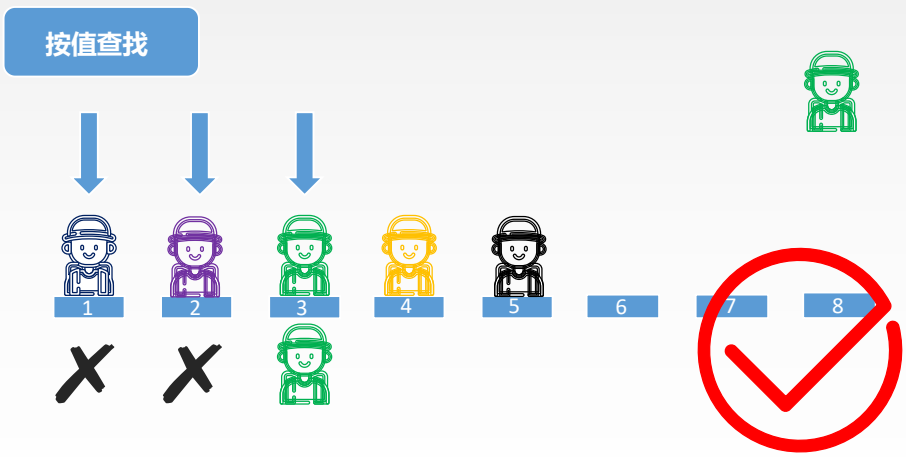
L.data a s f g h h

L.data a s f g h

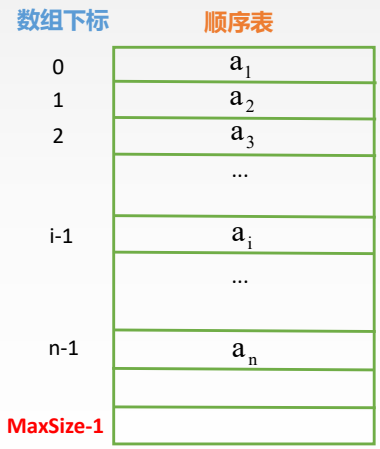
```
bool ListDelete(SqList &L, int i, ElemType &e){
    if(i<1||i>L.length+
        return false;
    e=L.data[i-1];
    for(int j=i;j<L.length;j++)
        L.data[j-1]=L.data[j];
    L.length--;
    return true;
}
```

O(1) O(n) O(n)

顺序表的定义



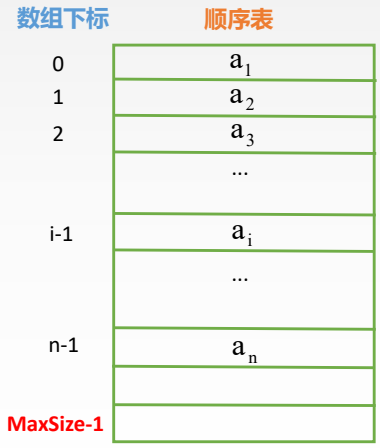
顺序表的定义




```
int LocateElem(SqList L, ElemType e){
    int i;
    for(i=0;i<L.length;i++)
        if(L.data[i]==e)
            return i+1;
    return 0;
}
```

O(1) O(n) O(n)

顺序表的定义




```
int LocateElem(SqList L, ElemType e){
    int i;
    for(i=0;i<L.length;i++)
        if(L.data[i]==e)
            return i+1;
    return 0;
}
```

O(1) O(n) O(n)

王道考研/CSKAOYAN.COM

本节回顾

本节内容

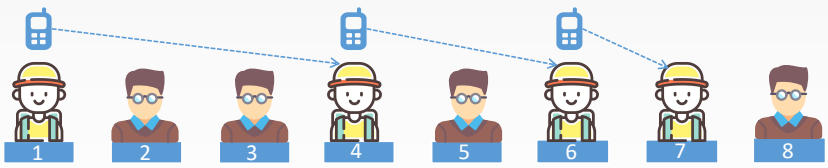
线性表

线性表
链式表示

单链表的定义

线性表的链式存储又称**单链表**

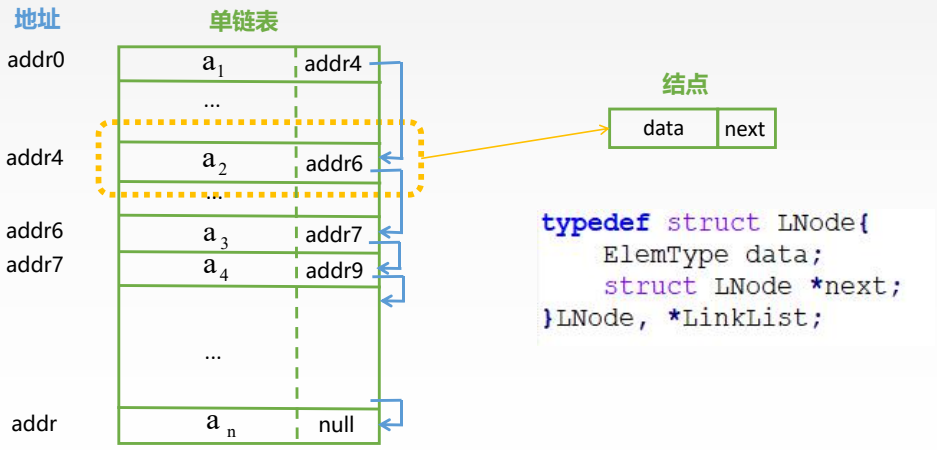
通过指针实现
线性逻辑关系



通过一组**任意**的存储单元来存储线性表中的数据元素

单链表的定义

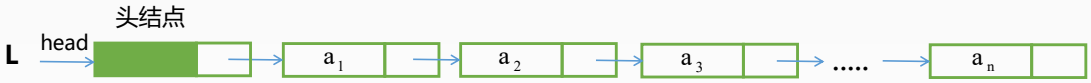
$L=(a_1,a_2,a_3,a_4,\ldots\ldots,a_n)$




```
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode, *LinkList;
```

单链表的定义

head为null时，单链表L为空



head->next为null时，单链表L为空

优点 链表的第一个位置和其他位置的操作统一
空表和非空表的操作统一

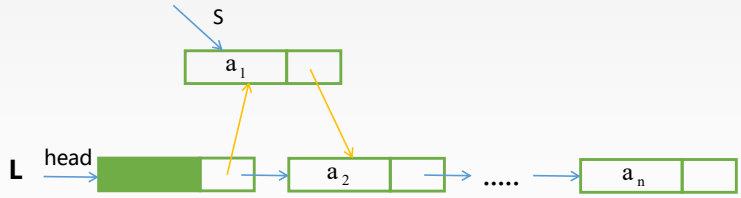
本节内容

线性表

单链表
基本操作

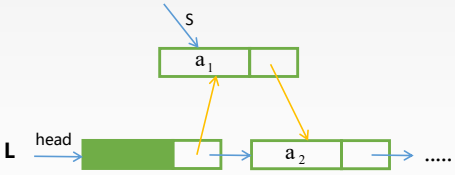
单链表的基本操作

头插法建立




```
s->next = L->next;
L->next = s;
```


单链表的基本操作

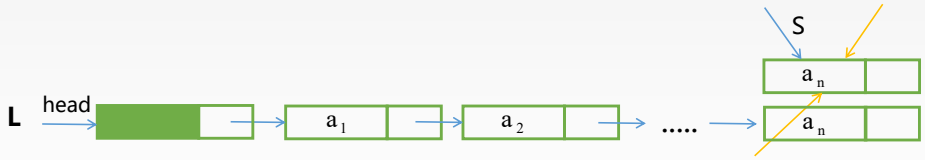


O(n)

```
LinkList List_HeadInsert (LinkList &L){
    LNode *s; int x;
    L=(LinkList)malloc(sizeof(LNode));
    L->next=NULL;
    scanf("%d", &x);
    while(x!=9999){
        s=(LNode*)malloc(sizeof(LNode))
        s->data=x;
        s->next=L->next;
        L->nest=s;
        scanf("%d",&x);
    }
    return L;
}
```

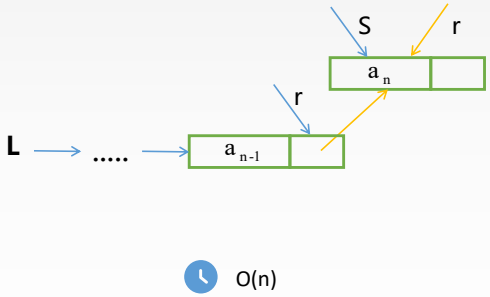
单链表的基本操作

尾插法建立




```
r->next = s;
r = s;
```


单链表的基本操作

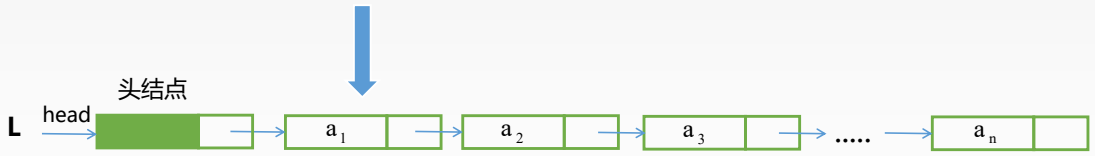


O(n)

```
LinkList List_TailInsert (LinkList &L){
    int x;
    L=(LinkList)malloc(sizeof(LNode));
    LNode *s, *r=L;
    scanf("%d", &x);
    while(x!=9999){
        s=(LNode*)malloc(sizeof(LNode))
        s->data=x;
        r->next=s;
        r=s;
        scanf("%d", &x);
    }
    r->next=NULL;
    return L;
}
```

单链表的基本操作

按序号查找&按值查找



count == 1

单链表的基本操作

```
LNode *GetElem(LinkList L,int i){
    int j=1;
    LNode *p=L->next;
    if(i==0)
        return L;
    if(i<1)
        return NULL;
    while(p&&j<i){
        p=p->next;
        j++;
    }
    return p;
}
```

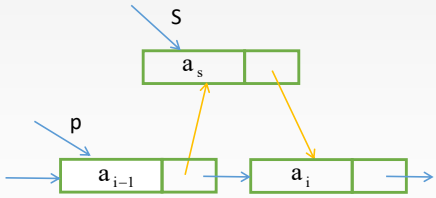
O(n)

```
LNode *LocateElem(LinkList L, ElemType e){
    LNode *p=L->next;
    while(p!=NULL&&p->data!=e)
        p=p->next;
    return p;
}
```

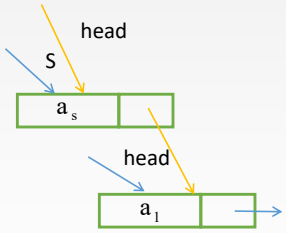
O(n)

单链表的基本操作

插入节点



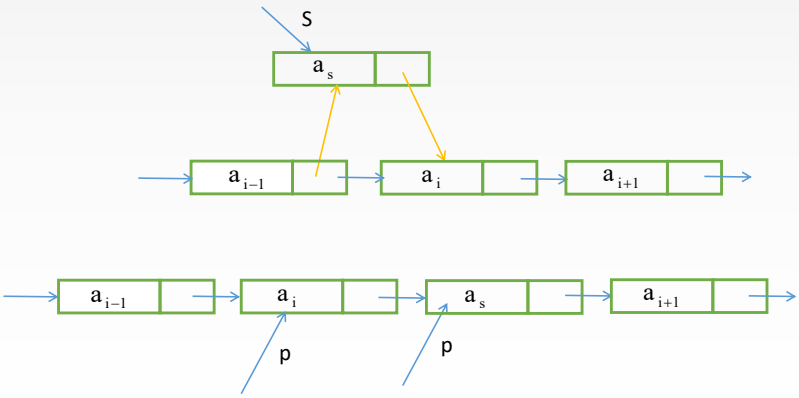

```
p = GetElem(L, i-1);
s->next = p->next;
p->next = s;
```




```
s->next = head;
head = s;
```

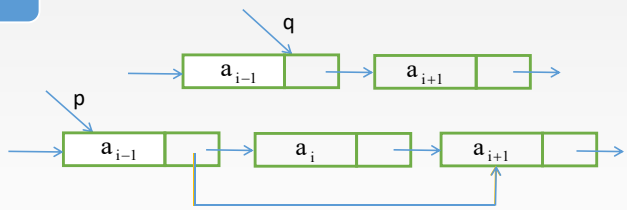

单链表的基本操作

前插法 VS 后插法



单链表的基本操作

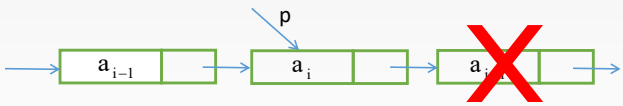
删除节点




```
p = GetElem(L, i-1);
q = p->next;
p->next = q->next;
free(q);
```


单链表的基本操作

删除给定节点 *p



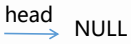

```
q = p->next;
p->data = p->next->data;
p->next = q->next;
free(q);
```

单链表的基本操作

求表长


```
int count=0;
p=head
while(p->next!=NULL){
    count++;
    p=p->next;
}
```

head->next == NULL;



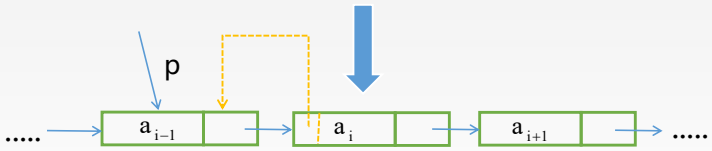
head == NULL;

本节内容

线性表

特殊链表

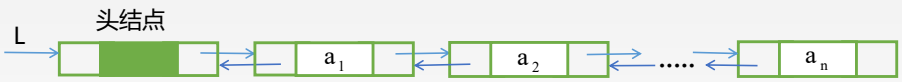
双链表



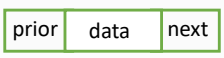
p = GetElem(L, i-1);

O(n)

双链表



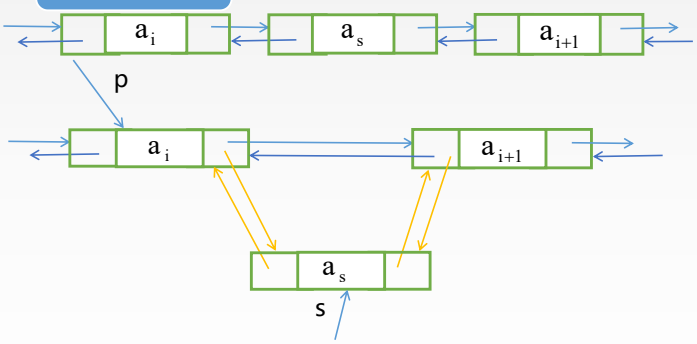
双链表结点




```
typedef struct DNode{
        ElemType data;
        struct DNode *prior, *next;
}DNode, *DLinklist;
```

双链表

插入操作

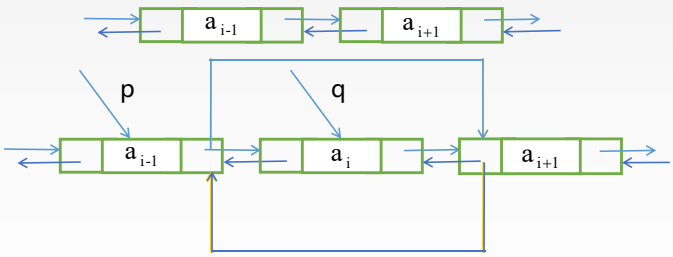



```
s->next = p->next;
p->next->prior = s;
s->prior = p;
p->next = s;
```

O(1)

双链表

删除操作

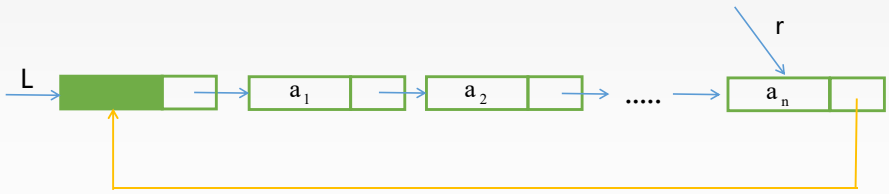



```
p->next = q->next;
q->next->prior = p;
free(q);
```

O(1)

循环链表

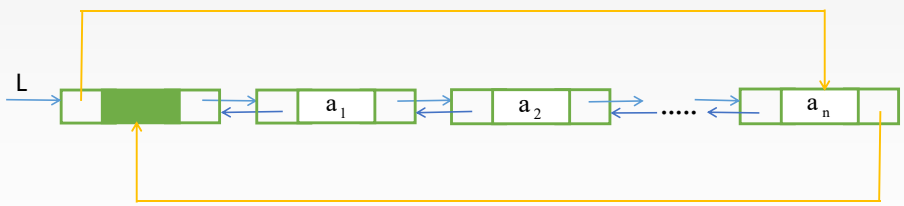
循环单链表



★ 仅设尾指针操作效率会更高

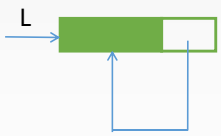
循环链表

循环双链表

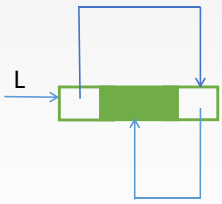


循环链表

? 空表判断

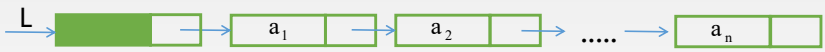


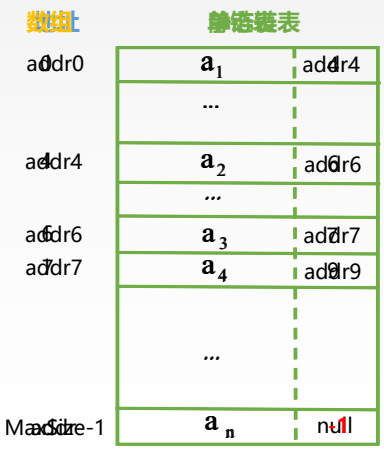
L->next == L;



L->next == L;
L->prior == L;

静态链表






```
#define MaxSize 50
typedef struct DNode{
        ElemType data;
        int next;
}SLinkList[MaxSize];
```

本节内容

线性表

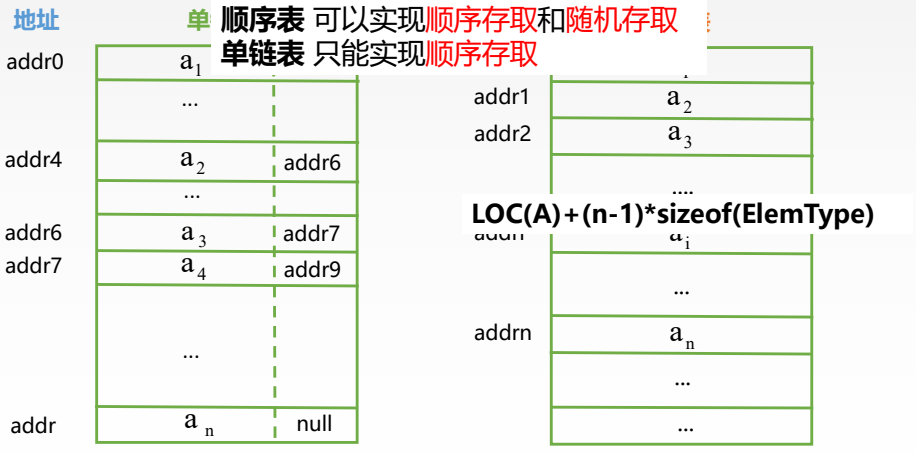
顺序表vs链表

顺序表与链表

存取方式

顺序表 可以实现顺序存取和随机存取

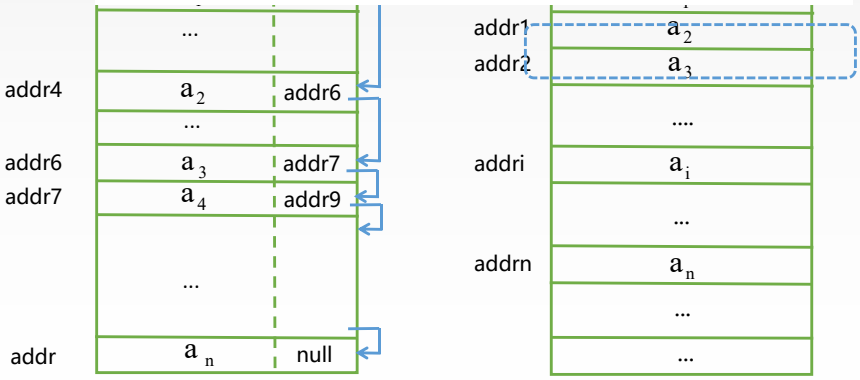
单链表 只能实现顺序存取



顺序表与链表

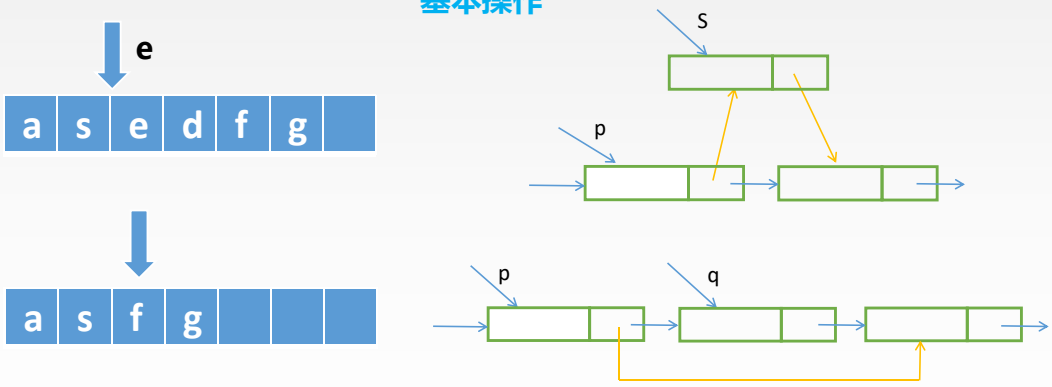
逻辑结构和物理结构

顺序表 逻辑相邻物理上也相邻，通过相邻表示逻辑关系
单链表 逻辑相邻物理上不一定相邻，通过指针表示逻辑关系



顺序表与链表

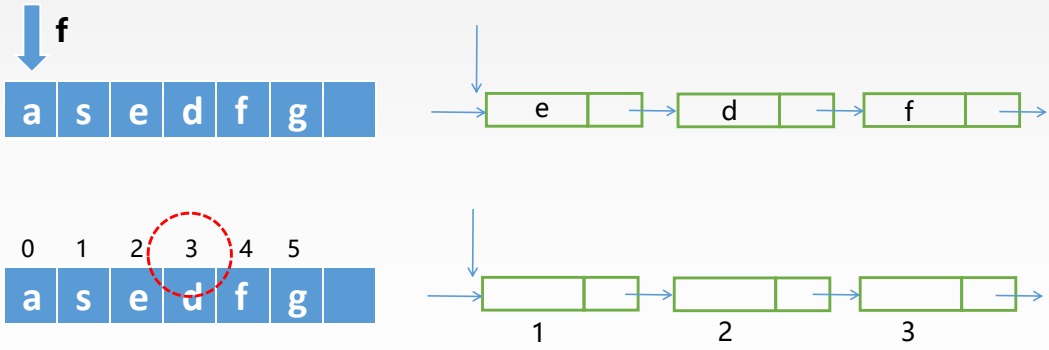
基本操作



插入&删除 单链表为O(1)(节点指针已知)；O(n)(节点指针未知)，但操作时只需修改指针
顺序表为O(n)且需要大量移动元素

顺序表与链表

基本操作



查找 按值查找中单链表和顺序表（无序）都为O(n)
按序查找中单链表为O(n)，顺序表为O(1)

顺序表与链表

内存空间

顺序存储 无论静态分配还是非静态都需要预先分配合适的内存空间
静态分配时预分配空间太大回造成浪费，太小会造成溢出
动态分配时虽不会溢出但是扩充需要大量移动元素，操作效率低

链式存储 在需要时分配结点空间即可，高效方便，但指针要使用额外空间

顺序表与链表

? 怎样选择线性表的存储结构

		稳定 顺序表	动态 单链表
存储	规模难估计		✓
	存储密度	✓	
效率	按序号访问	✓	
	插入和删除		✓
环境	基于数组	✓	
	基于指针		✓

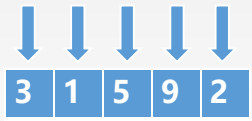
顺序表与链表

三个常用操作

最值	逆置	归并
L=(3, 1, 5, 9, 2)	L=(3, 1, 5, 9, 2)	L1=(1, 2, 3, 5) L2=(4, 6, 7, 8)
MAX=9 MIN=1	LR=(2, 9, 5, 1, 3)	L=(1, 2, 3, 4, 5, 6, 7, 8)

顺序表与链表

L=(3, 1, 5, 9, 2)



min = 3; max = 3;

min = 1; max = 3;

min = 1; max = 5;

min = 1; max = 9;

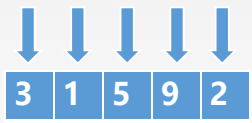
min = 1; max = 9;

```
int min = L[0];
int max = L[0];
for(int i = 0; i < n; i++){
    if(min > L[0])
        min = L[0];
    if(max < L[0])
        max = L[0];
}
```

O(n)

顺序表与链表

L=(3, 1, 5, 9, 2)



min = 3; max = 3;

min = 1; max = 3;

min = 1; max = 5;

min = 1; max = 9;

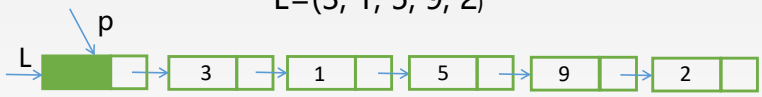
min = 1; max = 9;

```
int min = L[0];
int max = L[0];
for(int i = 0; i < n; i++){
    if(min > L[i])
        min = L[i];
    if(max < L[i])
        max = L[i];
}
```

O(n)

顺序表与链表

L=(3, 1, 5, 9, 2)



min = 3; max = 3;

min = 1; max = 3;

min = 1;

min = 1;

min = 1;

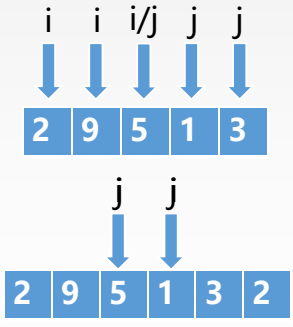
```
while(p != NULL){
    if(min > p ->data)
        min = p ->data;
    if(max < p ->data)
        max = p ->data;
    p = p ->next;
}
```

```
int min = p -> next ->data;
int max = p -> next ->data;
p = p -> next;
for( ; p != NULL; p = p ->next){
    if(min > p ->data)
        min = p ->data;
    if(max < p ->data)
        max = p ->data;
}
```

O(n)

顺序表与链表

L=(3, 1, 5, 9, 2)

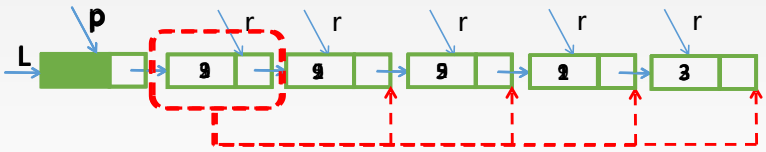



```
int i = 0;
int j = n-1;
while(i < j){
    temp = L[i];
    L[i] = L[j];
    L[j] = temp;
    i++; j--;
}
```

O(n)

顺序表与链表

L=(3, 1, 5, 9, 2)

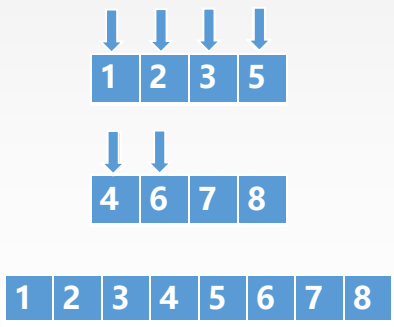



```
while(p ->next != r){
    temp = p -> next;
    p -> next = temp -> next;
    temp -> next = r -> next;
    r -> next = temp;
}
```

O(n)

顺序表与链表

L1=(1, 2, 3, 5) L2=(4, 6, 7, 8)

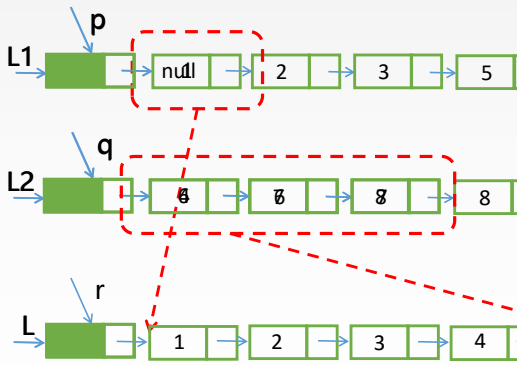



```
int i = 0, j = 0;
int k = 0;
for( ; i < L1_Size && j < L2_Size; k++){
    if(L1[i] < L2[j])
        L[k] = L1[i++];
    else
        L[k] = L2[j++];
}
while(i < L1_Size)
    L[k++] = L1[i++];
while(j < L2_Size)
    L[k++] = L2[j++];
```

O(n)

顺序表与链表

L1=(1, 2, 3, 5) L2=(4, 6, 7, 8




```
while(p->next!=NULL && q->next!=NULL){
    if(p->next->data < q->next->data){
        r->next = p->next;
        p->next= p->next->next;
        r = r->next;
    }
    else{
        r->next = q->next;
        q->next= q->next->next;
        r = r->next;
    }
}
if(p->next != NULL)   r -> next = p ->next;
if(q->next != NULL)   r -> next = q ->next;
free(p); free(q);
```

O(n)

本章总览