Chapter 2 Linear Structure

Lists, Stacks, Queues, String and Array

学习目标

➢掌握线性表的逻辑结构，线性表的顺序存储结构和链式存储结构的描述方法；熟练掌握线性表在顺序存储结构和链式存储结构的结构特点以及相关的查找、插入、删除等基本操作的实现；并能够从时间和空间复杂性的角度综合比较两种存储结构的不同特点。

➢掌握栈和队列的结构特性和描述方法，熟练掌握栈和队列的基本操作的实现，并且能够利用栈和队列解决实际应用问题。

➢掌握串的结构特性以及串的基本操作，掌握针对字符串进行操作的常用算法和模式匹配算法。

➢掌握多维数组的存储和表示方法，掌握对特殊矩阵进行压缩存储时的下标变换公式，了解稀疏矩阵的压缩存储表示方法及适用范围。

➢了解广义表的概念和特征。

2.1 Lists 表结构

- Definition and operations
- Implementation
- Applications

2.1.1 Definition

A list is defined as a sequence of a_1, …， a_n ,$n(n \geq 0)$.

$L=(a_1,...,a_{i-1},a_i,a_{i+1},...,a_n)$

***L*=('a','b',…,'x','y','z')**

***L*=('good','student','work')**

a_i（$1 \leq i \leq n$）称为数据元素；下角标*i*表示该元素在线性表中的位置或序号。*n*为线性表中元素的个数，称为表长度。

逻辑特征

$L=(a_1,...,a_{i-1},a_i,a_{i+1},...,a_n)$

✓ 有限性：个数是有穷的。

✓ 相同性：元素类型相同。

✓ 相继性：

a_1为表中第一个元素，无前驱元素，a_n为表中最后一个元素，无后继元素；

对于$1<i<n$，a_{i-1}为a_i的直接前驱，a_{i+1}为a_i的直接后继。

Operations

设***L*是类型为线性表实例，*e*为数据元素实例，*i*为位置变量。**

1）LenList (*L*);

2）GetElem (*L*,*i*);

3）SearchElem (*L*,*e*);

4）InsertElem (*L*,*i*,*e*);

5）DeleteElem (*L*,*i*)

ADT Sqlist

Data

$D=\{a_i|a_i \in \text{ElementType}, i=1,2,\ldots,n, n\geq 0\}$

$R=\{<a_{i-1},a_i>|a_{i-1},a_i \in D, i=2,\ldots,n\}$

Operations

求表的长度

LenList( )

输出：表的长度；

后件：计算表的长度。

取表L中第i个数据元素赋值给e

GetElem(i,e)

输入：位置参数i；

输出：e是ElementType类型；

前件：$1\leq i\leq$LenList(L)

后件：e被赋予表中第i个位置的元素值。

在L中第i个位置插入新的数据元素e

InsertElem(i,e)

输入：位置参数i, e是ElementType类型；

前件：$1\leq i\leq$LenList(L)+1

后件：在L中第i个位置插入新的数据元素e，表长加1。

删除表中第i个数据元素

DeleteElem(i)

输入：位置参数i；

前件：$1\leq i\leq$LenList(L)

后件：删除L中第i个位置的数据元素，表长减1。

按值查找

SearchElem(e,i)

输入：e是ElementType的类型；

输出：位置参数i；

前件：$1\leq$LenList(L)

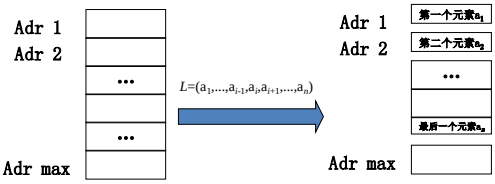
后件：若e存在表中，返回第一个e的位置为i，否则返回0。

End ADT

2.1.2 Physical structure design

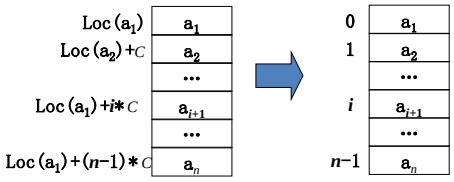
1. Sequence -- Storage cell with continuous location address

存储方式：将线性表中的元素**依次**存放在连续的存储空间中。



Sequence --Array implementation

$LOC(a_i) = LOC(a_1) + (i-1) * C \quad 1<=i<=n$



Characters:

How to express the logical structure in the computer?

元素之间逻辑上的关系，用物理上的相邻关系来表示（用物理上的连续性刻画逻辑关系）。

逻辑上相邻的元素，在物理位置上也相邻。

是一种随机访问存取的结构，也就是说可以按元素的位置之间的关系进行访问，其位置可以由公式直观的计算出来。

Operations

Insert element *e* at location *i*

InsertElem(*i*,*e*)

Input：*i* is location, *e* is ElementType；

Pre-condition：1≤*i*≤L.last+1

Post-condition：Insert element *e* at location *i* and add 1 to Length of *L*

- 插入前：$(a_1, \dots, a_{p-1}, a_p, \dots, a_n)$
- 插入后：$(a_1, \dots, a_{p-1}, x, a_p, \dots, a_n)$

a_{p-1}和a_p之间的逻辑关系发生了变化

⇩

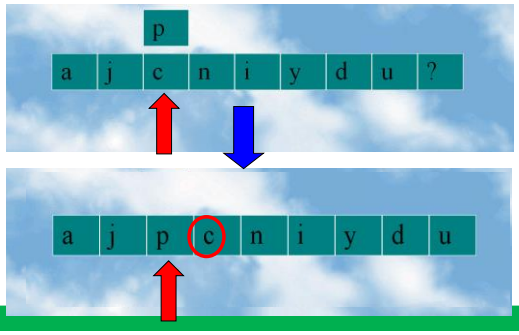
顺序存储要求存储位置反映逻辑关系

⇩

存储位置要反映这个变化

Case:

i=3, *e*= 'P'

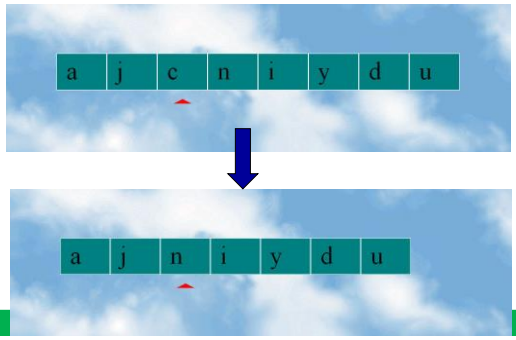



```
int InsertElem(sqlist *L,DataType x,int i)
{ int j;
  Pre-condition
  Move element
}
```

Delete element at location *i*

```
DeleteElem(i)
    Input：location i;
    Pre-condition：1≤i≤L.last
    Post-condition：delete the element
    at location i.
```

Case: i=3




```
int  DeleteElem(sqlist *L,int i)
{ int j;
```

Pre-condition
Move element
Post-condition

```
}
```

Analysis

```
int InsertElem(sqlist *L,DataType x,int i)
{ int j;
  if (((*L).last)>=maxsize－1)
    { printf("overflow\n"); return -1;} \\溢出
  else
    if((i<1)||(i>((*L).last)＋1)
      { printf("error\n"); return -1;}  \\非法位置
    else
    {   for (j=(*L).last;j>=i;j--)
          (*L).data[j+1]＝(*L).data[j];
        (*L).data[i]＝x;
        (*L).last＝(*L).last+1;
      }
    return(1);
  }
```

```
int  DeleteElem(sqlist *L,int i)
{ int j;
  if ((i<1)||(i>(*L).last+1))
    { printf("error\n"); return -1;}  \\非法位置
  else
  {   for(j=i;j<=(*L).last;j++)
        (*L).data[j-1]＝(*L).data[j];
      (*L).last--;              \\表长减1
  }
  return(1);
}
```

Analysis of operations(Insert and delete)

- 与表长及位置有关。
- 插入：
  - 最坏：i=1，移动次数为n
  - 最好：i=表长+1，移动次数为0
  - 平均：等概率情况下，平均移动次数$n/2$
- 删除：
  - 最坏：i=1，移动次数为n-1
  - 最好：i=表长，移动次数为0
  - 平均：等概率情况下，平均移动次数(n-1)/2

Search(Value) 按值查找

LocateElem1(e,i)
Input: element e;
Output: location of element e;
Pre-condition: L is not empty;
Post-condition: If e is in List L
then return location i.

```
int LocateElem1(SqList L, DataType e)
{ i=1;
  while ( i<=L.last && e != L.data[i-1]) ++i;
  if (i<=L.last) return i-1;
       else return -1;
}
```

Search(location)

LocateElem1(*i,e*)
Input: location *i*;
Output: element *e*;
Pre-condition: 1≤*i*≤L.last
Post-condition: return the element at location *i*.

Search(2)

```
DataType LocateElem2(SqList L, int i)
{
 If ((i<1)||(i>((*L).last))
   { printf("error\n");
    return -1;}  \\非法位置
  return (*L).data[i-1];
 }
```

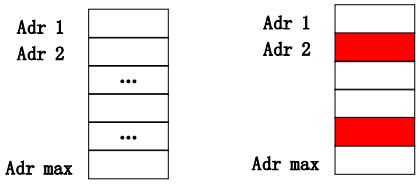
Analysis of implementation in array

1）优点
- 顺序表的结构简单
- 顺序表的存储效率高，是紧凑结构,无须为表示节点间的逻辑关系而增加额外的存储空间
- 顺序表是一个随机存储结构（直接存取结构）

2）缺点
- 在顺序表中进行插入和删除操作时，需要移动数据元素，算法效率较低。
- 对长度变化较大的线性表，或者要预先分配较大空间或者要经常扩充线性表，给操作带来不方便。

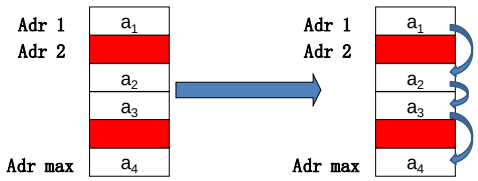
2. Linked

In the computer



散列在计算机中的存储单元，地址不一定连续。存储单元通过保存存储的地址来关联。

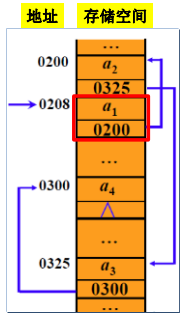
2. Linked $L=(a_1,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n)$



散列在计算机中的存储单元，地址不一定连续。存储单元通过保存存储的地址（指针）来关联。

一个存贮单元由结点组成，每个结点含有存放元素的数据域和存放其逻辑关系的指针域，这样就形成链接式存储结构。

(a1, a2 ,a3, a4)的存储示意图



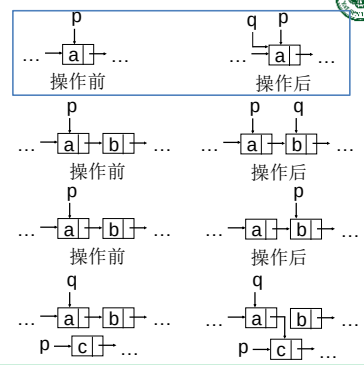
常见的指针操作

① q=p；

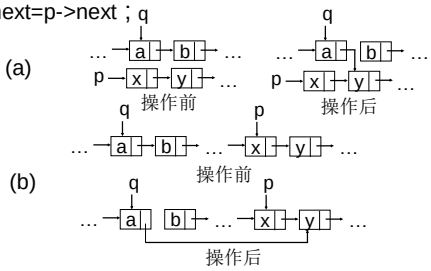
② q=p->next；

③ p=p->next；

④ q->next=p；

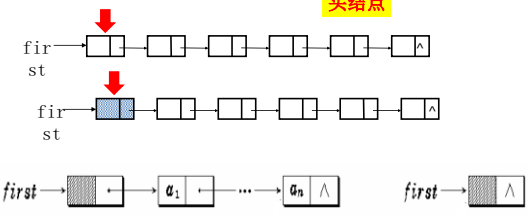


⑤ q->next=p->next；



Express of linked list

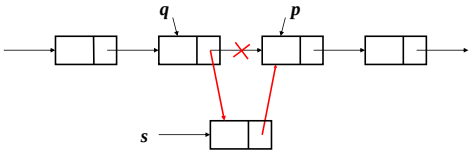
头指针
头结点



Characters of linked list:

Insert

（在*p*结点所在的位置插入：在*p*结点前插入）

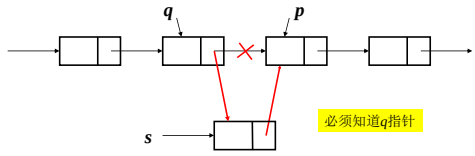


建立新结点*s*，向新结点中添入内容

查找插入结点的位置*p*, 以及*p*结点的前驱结点*q*

将新结点链入链中

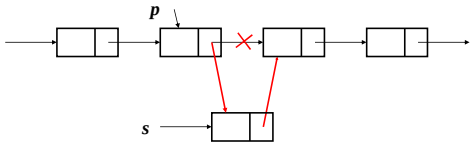
Insert（在*p*结点所在的位置插入：在*p*结点前插入）




```
New(S);
S->data=a;
S->next=p;
q->next=S;
```

```
New(S);
S->data=a;
q->next=S;
S->next=p;
```

Insert（在*p*结点之后插入）




```
New(S);
S->data=a;
p->next=S;
S->next=p->next;
```

```
New(S);
S->data=a;
S->next=p->next;
p->next=S;
```

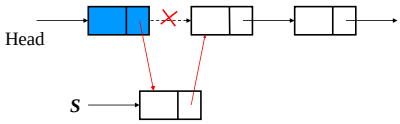
Operations of Linked list

- Create
- Search
- Delete
- Union

Create

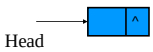
Algorithm:

1. To create Head
2. To create new node S
3. Insert S



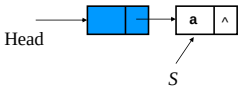
Case: L=(a,b,c,d,e)

Step 1




```
New(Head);
Head->next=NULL;
```

Step 2: insert a




```
New(S);
S->data=a;
S->next=Head->next;
Head->next=S;
```

Algorithm

1. To create Head
2. To create *S*
3. Insert *S*

```
Void CreateList(Head)
{
  new(Head);
  Head->next=NULL;
  scanf("%c",ch);
  while (ch<>'#') do
    {
      new(S);
      S->data=ch;
      S->next=Head->next;
      Head->next=S;
      scanf("%c",ch);
    }
}
```

```
Void CreateList(Head)
{ new(Head);
  Head->next=NULL;
  Last=Head;
  scanf("%c",ch);
  while (ch<>'#') do
    {  new(S);
       S->data=ch;
       S->next=NULL;
       Last->next=S;
       (          )=S;
       scanf("%c",ch);
    }
}
```

自查：熟练单链表上的操作

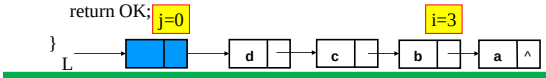
建立单链表
单链表上按位置查找
单链表上按值查找
删除单链表中的元素
两个递增有序的单链表合并

Insert element at location *i*：

```
Status ListInsert_L(LinkList &L,int i,DataType e)
{  LinkList p,s;
   p=L;  int j=0;
   while (p && j<i-1) { p=p->next; j=j+1;}
   if (!p) return ERROR;
   s = (LinkList) malloc( sizeof (LNode));
   s->data = e; s->next = p->next;
   p->next = s;
   return OK;
}
```

带头结点的单链表

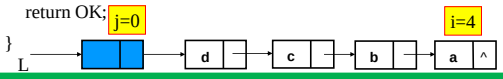
P指向哪个结点？



Insert element at location *i*：

```
Status ListInsert_L(LinkList &L,int i,DataType e)
{  LinkList p,s;
   p=L;  int j=0;
   while (p && j<i-1) { p=p->next; j=j+1;}
   if (!p) return ERROR;
   s = (LinkList) malloc( sizeof (LNode));
   s->data = e; s->next = p->next;
   p->next = s;
   return OK;
}
```

带头结点的单链表

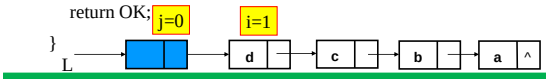


Insert element at location *i*：

```
Status ListInsert_L(LinkList &L,int i,DataType e)
{  LinkList p,s;
   p=L;  int j=0;
   while (p && j<i-1) { p=p->next; j=j+1;}
   if (!p) return ERROR;
   s = (LinkList) malloc( sizeof (LNode));
   s->data = e; s->next = p->next;
   p->next = s;
   return OK;
}
```

带头结点的单链表

需要完善的地方？

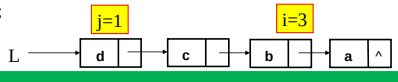


Insert element at location *i*：

```
Status ListInsert_L(LinkList &L,int i,DataType e)
{  LinkList p,s;
   p=L;  int j=1;
   while (p && j<i-1) { p=p->next; j=j+1;}
   if (!p) return ERROR;
   s = (LinkList) malloc( sizeof (LNode));
   s->data = e; s->next = p->next;
   p->next = s;
   return OK;
}
```

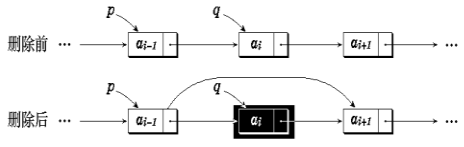
问题：不带头结点？

表空的判断



Delete

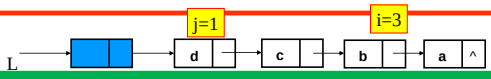
```
Status ListDelete_L(LinkList &L, int i, DataType &e)
{  LinkList p,q;
   p=L;  int j=0;
   while (p->next && j<i-1) { p=p->next; ++j;}
   if (!(p->next)) return ERROR;
   q=p->next;  p->next = q->next;
   e=q->data;  free(q);
   return OK;
}
```



Search (*i*)

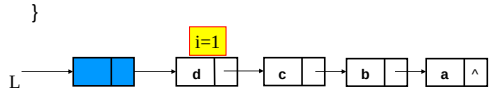
```
Status GetElem_L(LinkList L, int i,DataType &e)
{   LinkList p;
    p=L->next;  int j=1;
    while (p && j<i) { p=p->next; ++j; }
    if (!p) return ERROR;
    e=p->data;
    return OK;
}
```

P指向哪个结点？

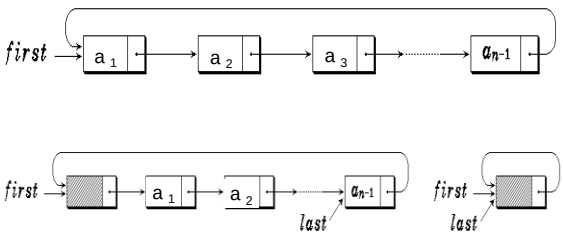


Search (*e*)

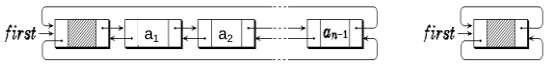
```
int LinkLocate_L (LinkList L, int x)
{  int i;   LinkList  p;
   p=L->next;  i=1;
   while (p!=NULL && p->data != x)
         {p= p->next; i++;}
   if (!p)  { printf ("Not found! \n");
              return(0);
            }
   else { printf ("i=%d\n",i);return (i); }
}
```



Circular linked list



Doubly Linked list

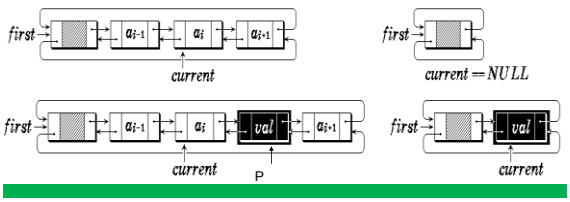


p == p→prior→next == p→next→prior

p→prior = p ; p→next = p;

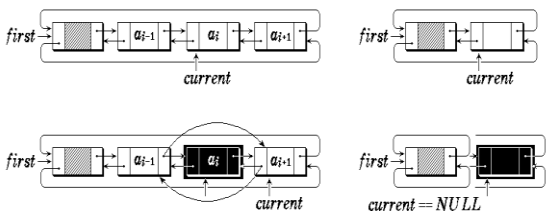
Insert at doubly circular Linked list

p→prior = current; **(1)**
p→next =current→next; **(2)**
current→next = p; **(3)**
p→next→prior = p; **(4)**

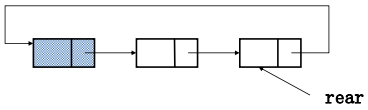


Delete

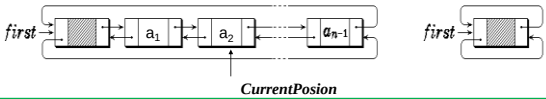
current→next→prior = current→prior;
current→prior→next = current→next;



With point rear



With current location



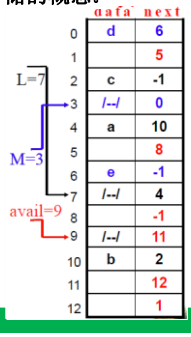
连续设计和链接设计的对比

比较参数	连续设计	链接设计
表的容量	固定，不易扩充	灵活，易扩充
存取操作	随机访问存取	顺序访问存取
时间	插入删除费时间	访问元素费时间
空间	估算长度，浪费空间	实际长度

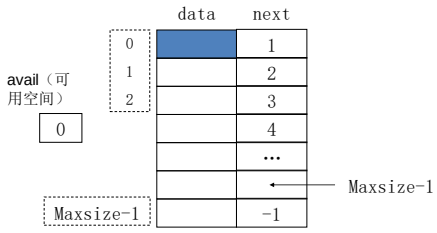
4. Implement Linked list in array

利用静态数组实现链接存储的概念。

举例：
线性表 L=(a,b,c)
线性表 M=(d,e)
空闲表 avail=9



结点说明

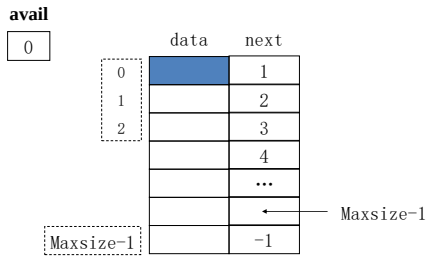



```
#define maxsize 1024    \\存储池最大容量
typedef int datatype;
typedef int cursor;
typedef struct          \\结点类型
{ datatype data;
  cursor next;
} node;

node nodepool[maxsize];    \\存储池
cursor avail;
```

Initial array nodepool

```
INITIALIZE()
{ int i;
  for (i=0;i<maxsize-1;i++)
    nodepool[i].next＝i+1;
  nodepool[maxsize-1].next＝-1;
  avail ＝0;
}
```



结点的分配算法

```
cursor GETNODE()
{ cursor p;
  if (avail ==-1) p＝-1;
  else
    { p＝ avail;
      avail ＝nodepool[avail].next;
    }
  return p;
}
```

结点的回收算法

```
FREENODE(cursor p)
{
  nodepool[p].next＝ avail;
  avail ＝p;
}
```

静态链表查找算法

```
cursor FINDSTLIST(cursor L,int i)
{ cursor p; int j;
  p＝L; j＝0;
  while ((nodepool[p].next!=-1)&&(j<i))
   { p＝nodepool[p].next;
     j++;
   }
  if (i==j) return(p);
  else return -1;
}
```

2.1.3 Applications