线性表List

$(a_1, a_2, a_3, \ldots, a_n)$

hwdong

B站 和微博： hw-dong

回顾

- 抽象数据类型ADT(逻辑结构): 数学/逻辑上描述数据特征和操作含义。

  如： 线性表List就是一列元素(a_1,a_2,...,a_n)，可以根据位置读取、修改、插入、删除、...

- 物理结构： ADT(逻辑结构)在计算机中的表示和实现。

  顺序映像： 数组array

  链式映像： 链表Linked List

- 抽象数据类型：线性表List
- 顺序表Array List
- 链式表LinkedList

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作：

  init： 创建空的List ( )

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作:

  init： 创建空的List
  insert： 插入元素

(a)

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作:

  init： 创建空的List
  insert： 插入元素 (a, b)

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作:

  init： 创建空的List
  insert： 插入元素

(a, c, b)

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作:

  init： 创建空的List
  insert： 插入元素

(a, c, d, b)

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作:

  init： 创建空的List
  insert： 插入元素
  remove ： 删除元素

(a, c, d, b)

(a, c, b)

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作:

  init： 创建空的List
  insert： 插入元素
  remove ： 删除元素

(a, c, d, b)

(c, b)

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作：

  init： 创建空的List
  insert： 插入元素
  remove ： 删除元素
  read： 读元素

(c, b)

read(2) : b

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作:

  init： 创建空的List
  insert： 插入元素
  remove ： 删除元素　　(c, b)
  read： 读元素
  write/modify ： 写/修改元素　　(c, h)　write(2,h)

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作：

  init： 创建空的List
  insert： 插入元素
  remove ： 删除元素
  read： 读元素
  write/modify ： 写/修改元素
  size: 查询元素的个数

(c, h)

size( ) : 2

线性表List

- n个元素的有限序列（a_1, a_2, ..., a_n）

  不关注数据元素的具体类型：int, book, student, ...

- 操作：

  init： 创建空的List
  insert： 插入元素
  remove ： 删除元素
  read： 读元素
  write/modify ： 写/修改元素
  size: 查询元素的个数
  find: 查询满足某条件的元素

(c, h)

size( ) : 2

线性表List

ADT List{

元素关系：线性关系（a_1, a_2, ..., a_n）

操作：

```
bool initList( );              // 初始化一个空的线性表L
bool insert(i, e);             // 在位置i插入新数据元素e
bool  remove(i);               // 删除第i个元素
bool get(i, &e);               // 得到第i个元素，放入e
bool set(i, e);               // 修改第i个元素值为e
```

线性表List

```
    bool insert_front(e);        // 新元素e插入在第一个
    bool push_back (e);          //  新元素e加到最后面

    bool remove_front();         // 删除第一个元素
    bool pop_back ();            // 删除最后一个元素
    int size();                  // 得到线性表中的元素个数
    int find((*fun)( ));         // 查询满足某条件(比如相等)的元素
    int traverse((*fun)( ));     // 遍历访问每个元素，执行fun操作
}
```

线性表的顺序实现：顺序表Array List

- 用一连续的存储空间来存储数据元素并以物理位置相邻性表示逻辑关系；在此基础上实现线性表的基本操作。

$(a_1, a_2, a_3, \ldots, a_n)$

- 连续存储空间： 静态数组、动态内存块

```
char data[100];  int  n = 0;


int  capacity=10,  n= 0;
char *data = new char[capacity];
```

顺序表Array List

```
char data[100]; int  n = 0;
```

顺序表Array List


```
char data[100]; int  n = 1;

push_back('H');
```

顺序表Array List

char data[100]; int n = 2;

push_back(‘H’);

push_back(‘o’);

顺序表Array List

char data[100]; int n = 3;

push_back(‘H’);

push_back(‘o’);

push_back(‘e’);

顺序表Array List

char data[100]; int n = 3;

push_back(‘H’);

push_back(‘o’);

push_back(‘e’);

insert(2,’L’);

顺序表Array List

char data[100]; int n = 3;

push_back(‘H’);

push_back(‘o’);

push_back(‘e’);

insert(2, ’L’);

顺序表Array List

char data[100]; int n = 3;

push_back(‘H’);

push_back(‘o’);

push_back(‘e’);

insert(2, ‘L’);

顺序表Array List

char data[100]; int n = 4;

push_back(‘H’);

push_back(‘o’);

push_back(‘e’);

insert(2, ‘L’);

顺序表Array List

char data[100]; int n = 4;

push_back(‘H’);

push_back(‘o’);

push_back(‘e’);

insert(2, ‘L’);

remove(1);

顺序表Array List

char data[100]; int n = 4;

push_back(‘H’);

push_back(‘o’);

push_back(‘e’);

insert(2, ‘L’);

remove(1);

顺序表Array List

char data[100]; int n = 4;

insert(‘H’);

insert(‘o’);

insert(‘e’);

insert(2, ‘L’);

remove(1);

顺序表Array List

char data[100]; int n = 4;

push_back(‘H’);

push_back(‘o’);

push_back(‘e’);

insert(2, ‘L’);

remove(1);

顺序表Array List

char data[100]; int n = 3;

push_back(‘H’);

push_back(‘o’);

push_back(‘e’);

insert(2, ‘L’);

remove(1);

顺序表Array List

char data[100]; int n = 3;

push_back(‘H’);

push_back(‘o’);

push_back(‘e’);

insert(2, ‘L’);

remove(1);

顺序表Array List

- 静态数组的缺点：

  1） 大小固定，不能动态改变，实际情况万一超过呢?

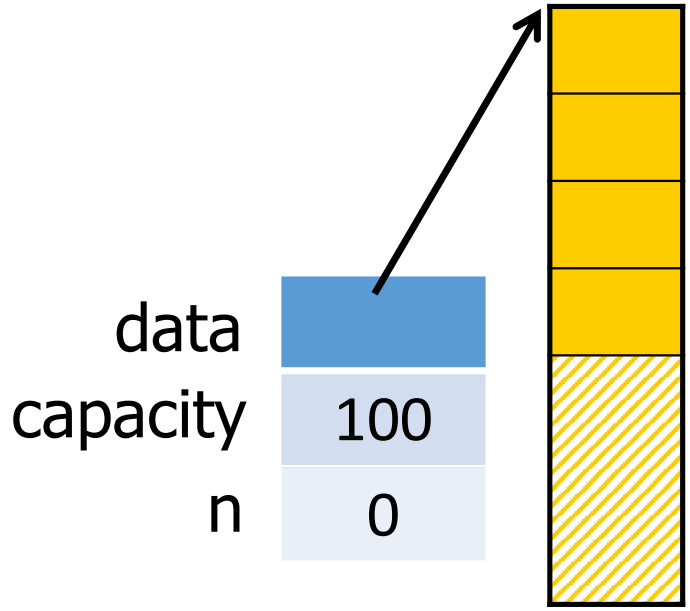
  2） 一开始需要足够大的数组空间，造成空间浪费。可容纳1000个学生成绩的程序实际使用中通常只有60左右。

- 解决方法：动态分配内存块

```
int  capacity=10,  n= 0;
T   *data = new T[capacity];
```

SqList

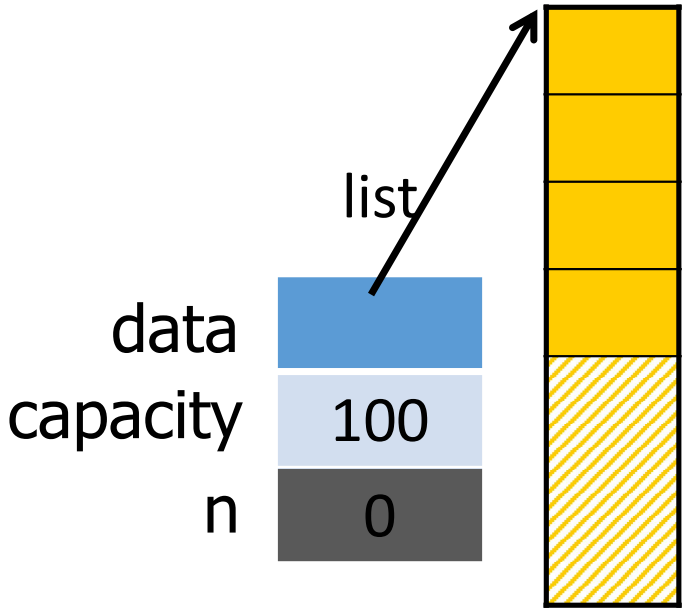
```
template<class T>
class SqList{
    T *data;
    int capacity, n;
public:
    SqList(int cap = 100) {data = new T[cap];capacity = cap ;n = 0;}
    bool insert (int i, T e) ;
    bool remove (int i);
    bool push_back (T e);
```



```
    bool insert_front (T e);
    bool get (int i, T &e);
    bool set (int i, T e);
    int size () { return n; }
    int find ( (*fun)());
    void traverse ( (*fun)());
    …
private:
    bool realloc( );
};
```

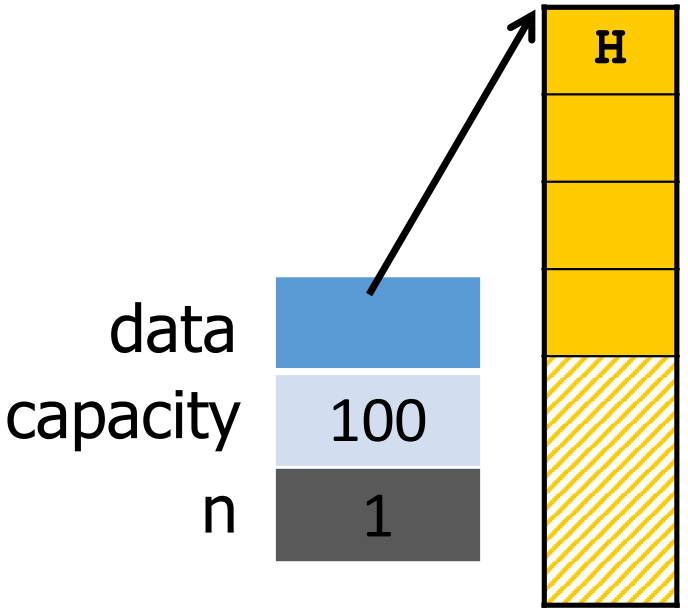
测试SqList

```
int main(){
   SqList<char>  list;




}
```



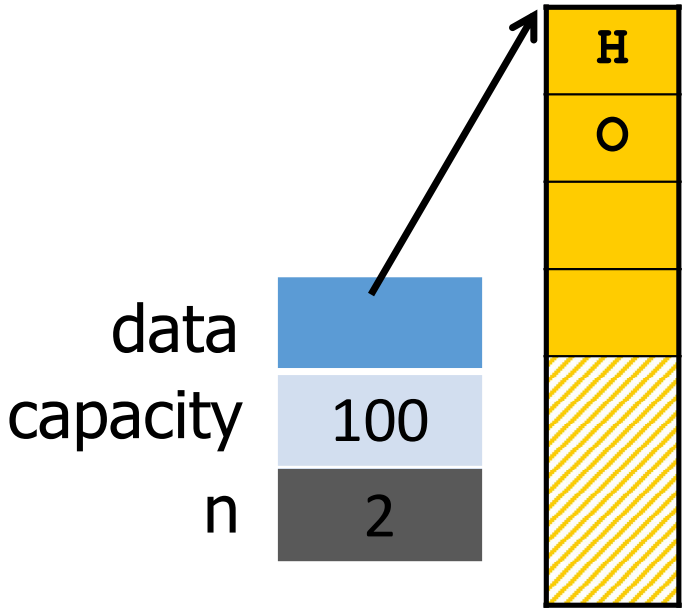
测试SqList

```
int main(){
   SqList<char>  list;
  list.push_back ( 'H');



}
```



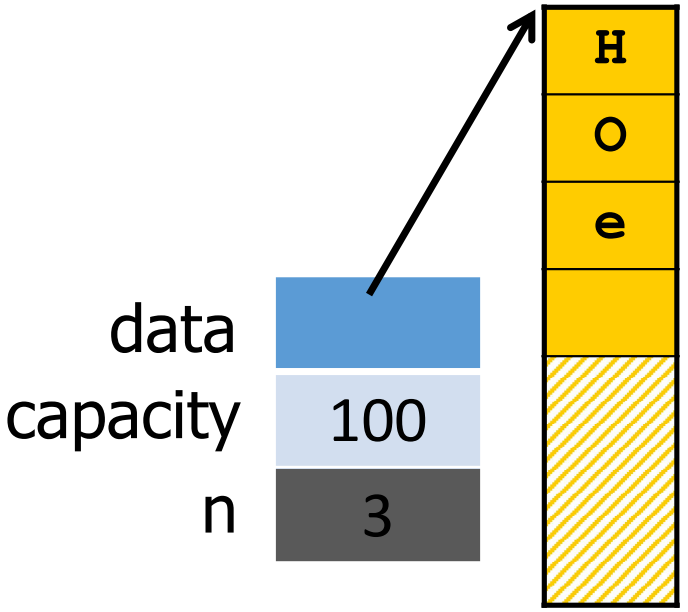
测试SqList

```
int main(){
   SqList<char>  list;
  list.push_back ( 'H');
  list.push_back ( ''o');



}
```



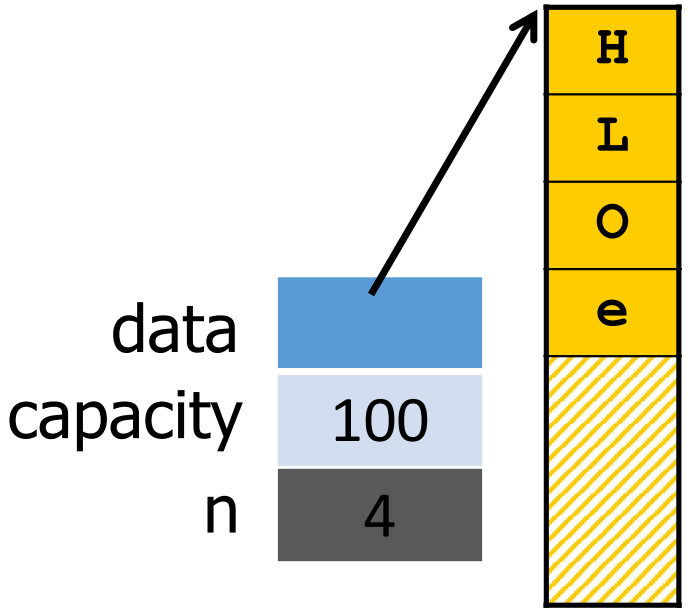
测试SqList

```
int main(){
   SqList<char>  list;
  list.push_back ( 'H');
  list.push_back ( "o');
  list.push_backt ( "e');


}
```



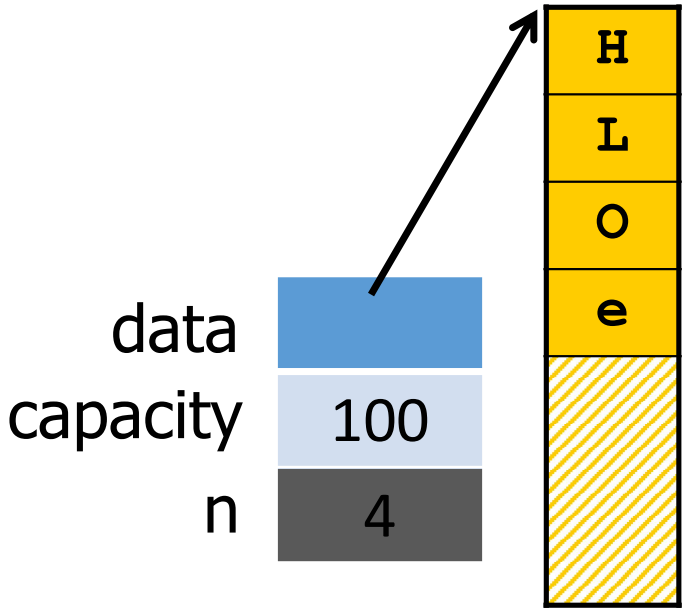
测试SqList

```
int main(){
   SqList<char>  list;
  list.push_back ( 'H');
  list.push_back ( ''o');
  list.push_back ( ''e');
  list.insert ( 2, 'L');


}
```



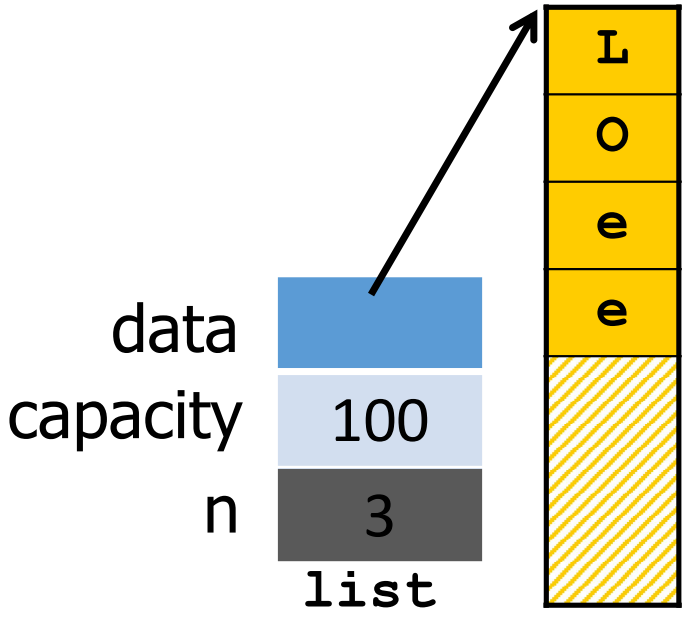
测试SqList

```
int main(){
   SqList<char>  list;
  list.push_back ( ‘H’);
  list.push_back ( ‘‘o’);
  list.push_back ( ‘‘e’);
  list.insert ( 2, ‘L’);
  list.remove ( 1);
}
```



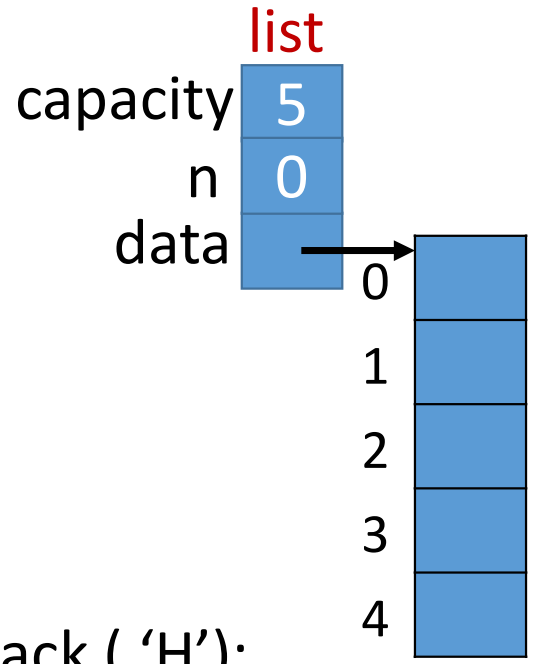
测试SqList

```
int main(){
   SqList<char>  list;
  list.push_back ( ‘H’);
  list.push_back ( ‘‘o’);
  list.push_back ( ‘‘e’);
  list.insert ( 2, ‘L’);
  list.remove ( 1);
}
```



Push_back

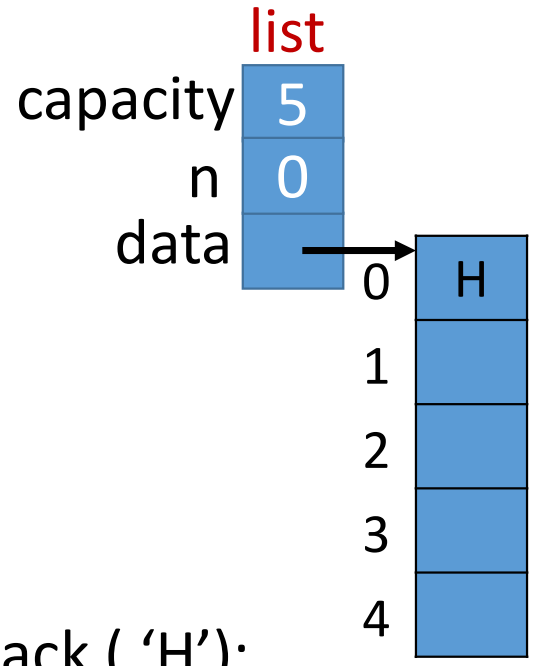
```
bool SqList<T>::push_back(T e){




   return true;
}
```



list.push_back (‘H’);

Push_back

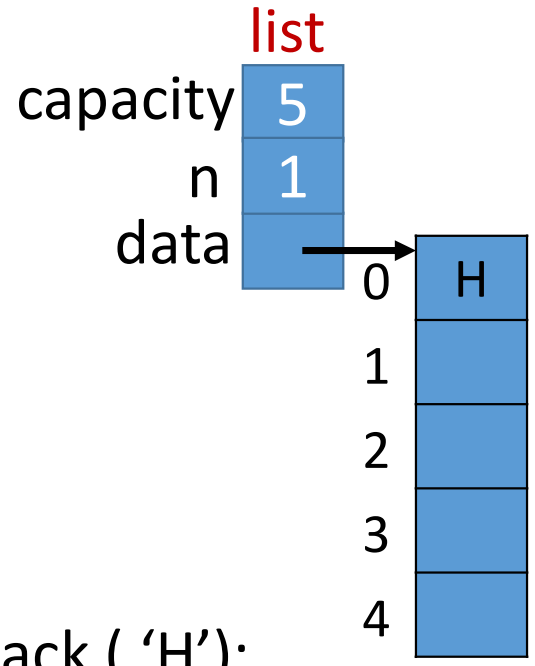
```
bool SqList<T>::push_back(T e){



   data[n] = e;


   return true;
}
```



list.push_back (‘H’);

Push_back

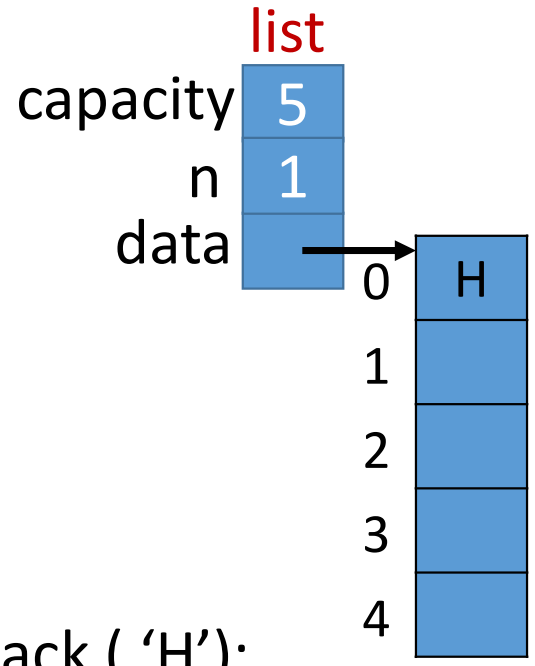
```
bool SqList<T>::push_back(T e){



   data[n] = e;
   n++;
   return true;
}
```



list.push_back (‘H’);

Push_back

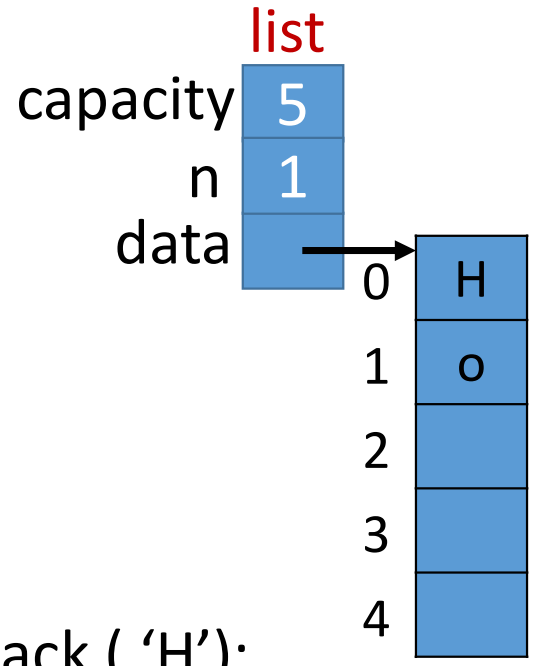
```
bool SqList<T>::push_back(T e){



   data[n] = e;
   n++;
   return true;
}
```




```
list.push_back ( 'H');
list.push_back ( 'o');
```

Push_back

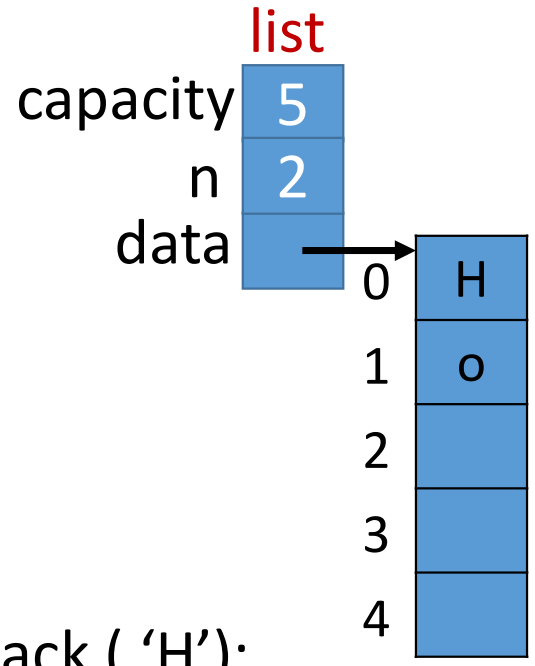
```
bool SqList<T>::push_back(T e){


   data[n] = e;
   n++;
   return true;
}
```




```
list.push_back ( 'H');
list.push_back ( 'o');
```

Push_back

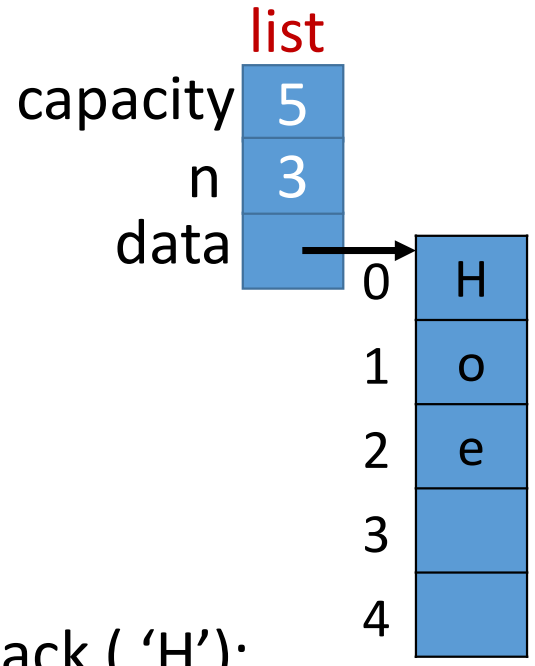
```
bool SqList<T>::push_back(T e){



   data[n] = e;
   n++;
   return true;
}
```




```
list.push_back ( 'H');
list.push_back ( 'o');
```

Push_back

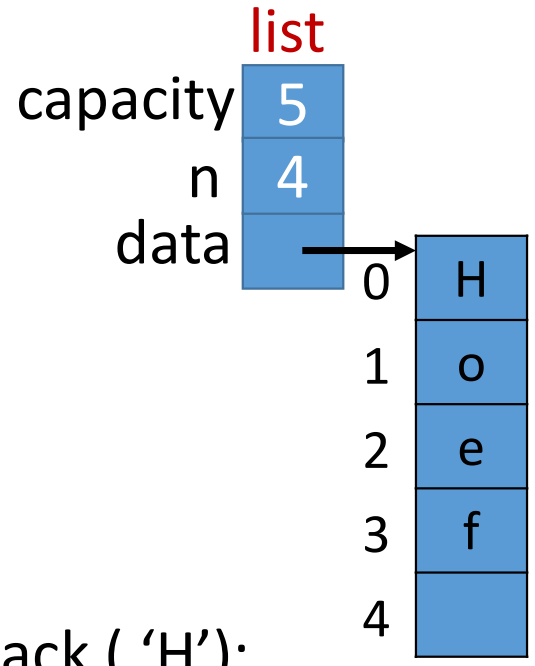
```
bool SqList<T>::push_back(T e){



   data[n] = e;
   n++;
   return true;
}
```




```
list.push_back ( 'H');
list.push_back ( 'o');
list.push_back ( 'e');
```

Push_back

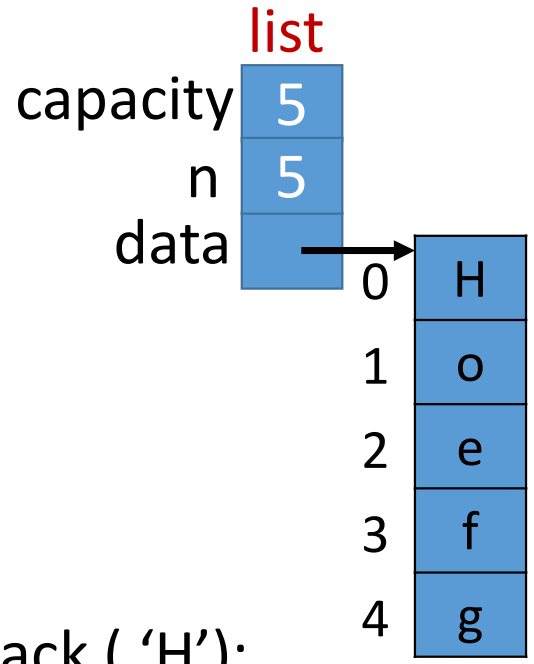
```
bool SqList<T>::push_back(T e){



   data[n] = e;
   n++;
   return true;
}
```




```
list.push_back ( 'H');
list.push_back ( 'o');
list.push_back ( 'e');
list.push_back ( 'f');
```

Push_back

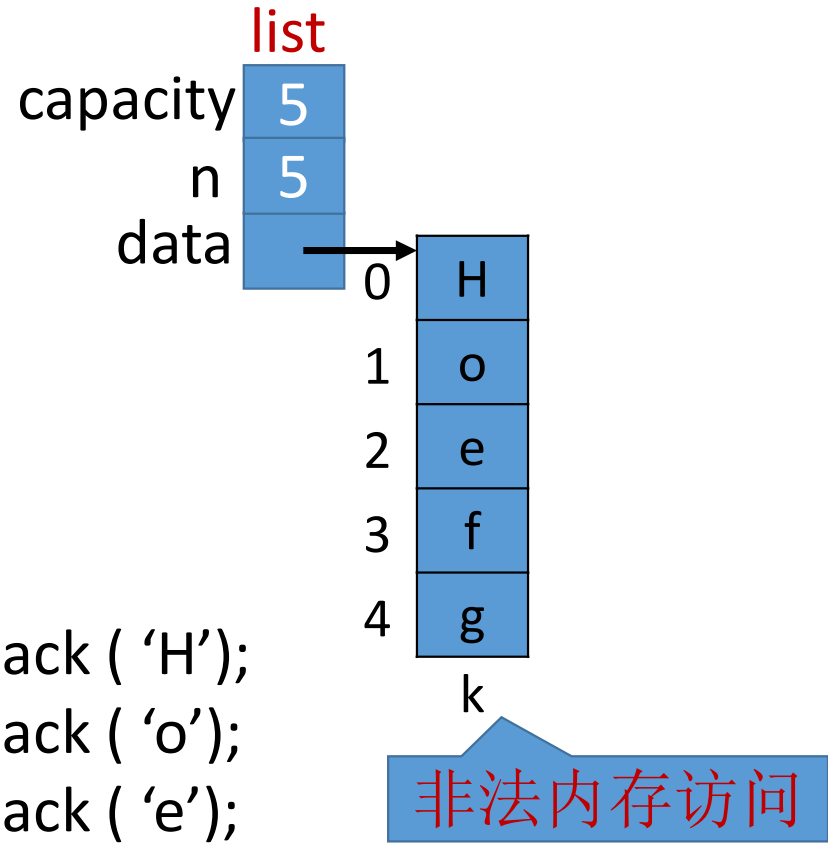
```
bool SqList<T>::push_back(T e){



   data[n] = e;
   n++;
   return true;
}
```




```
list.push_back ( 'H');
list.push_back ( 'o');
list.push_back ( 'e');
list.push_back ( 'f');
list.push_back ( 'g');
```

Push_back

```
bool SqList<T>::push_back(T e){



   data[n] = e;
   n++;
   return true;
}
```

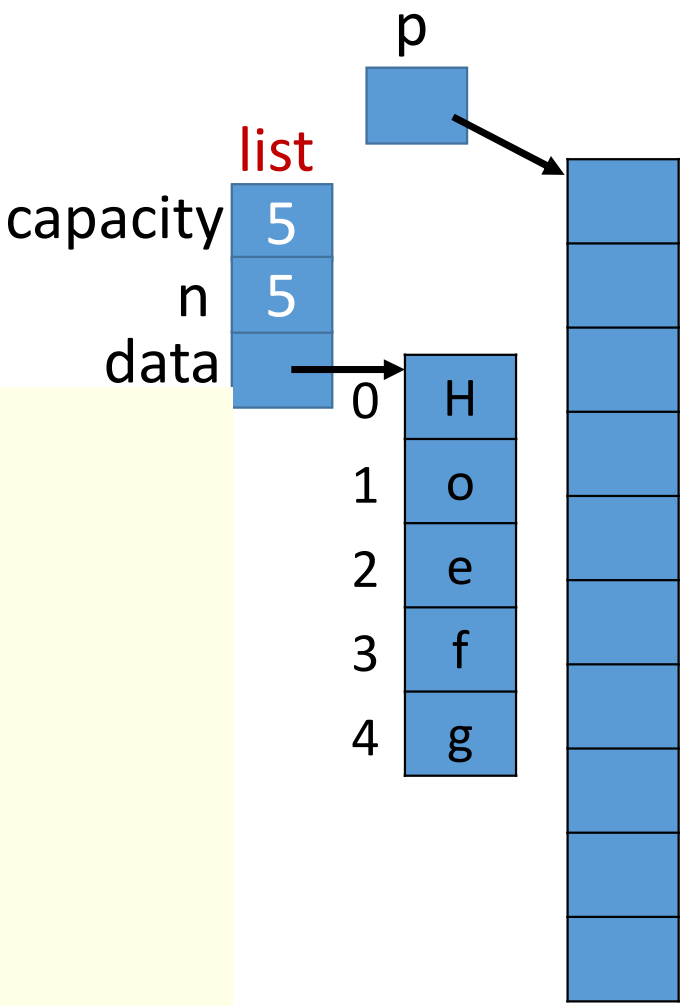



```
list.push_back ( 'H');
list.push_back ( 'o');
list.push_back ( 'e');
list.push_back ( 'f');
list.push_back ( 'g');
list.push_back ( 'k');
```

realloc

- 申请更大的内存块

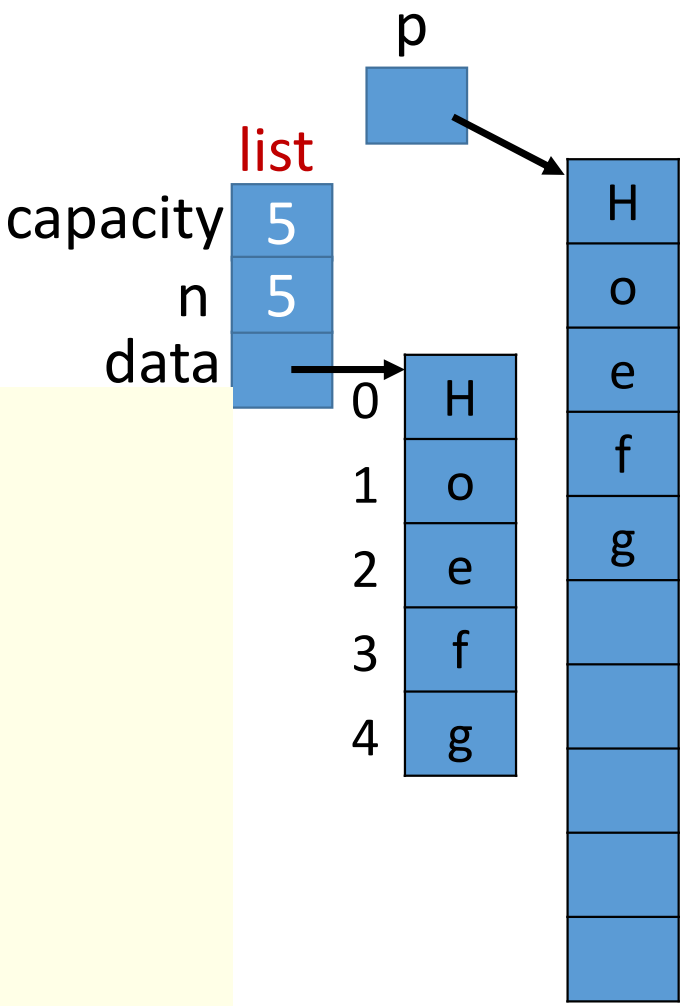
```
bool SqList<T>::realloc(){
  T * p = new T[2*capacity];
  if(!p)          return false;




  return true;
}
```



realloc

- 申请更大的内存块

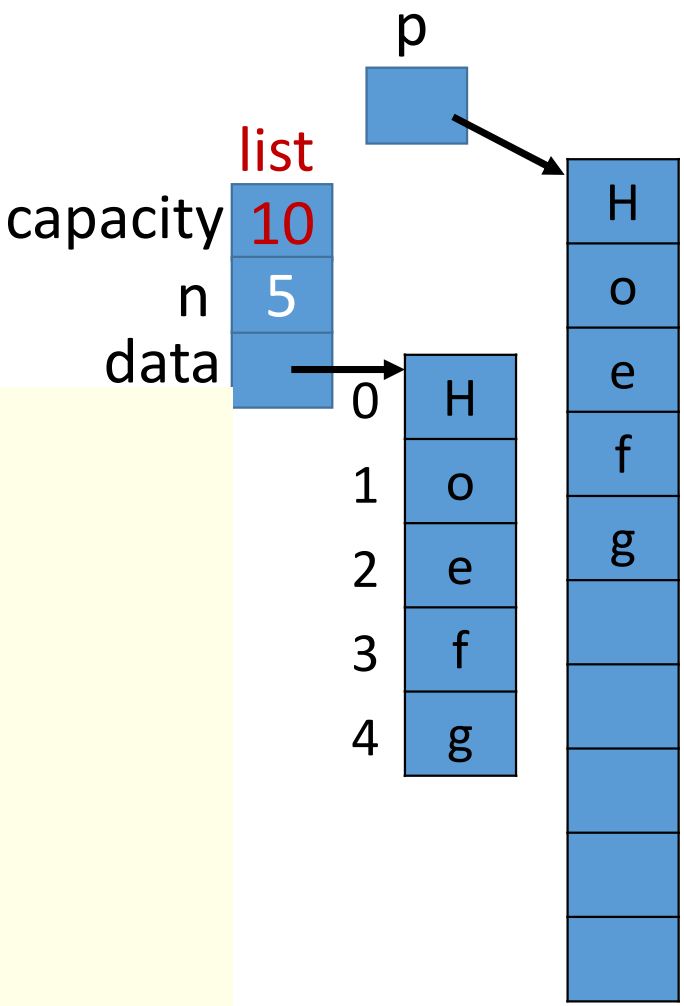
```
bool SqList<T>::realloc(){
   T * p = new T[2*capacity];
   if(!p)          return false;
   for(int i = 0 ; i<capacity;i++)
      p[i] = data[i];



   return true;
}
```



realloc

- 申请更大的内存块

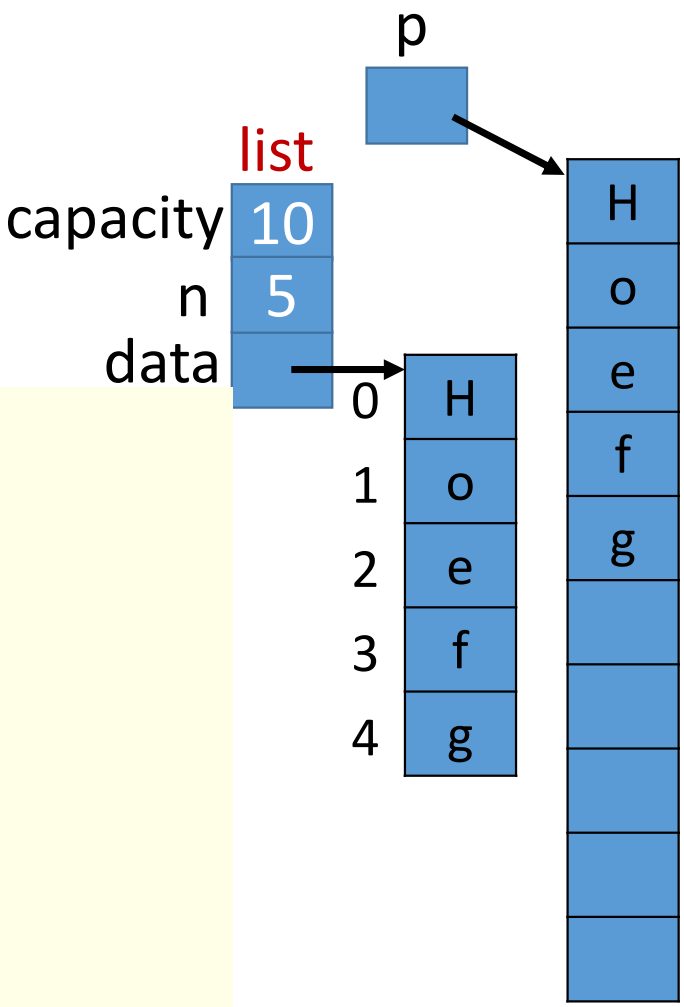
```
bool SqList<T>::realloc(){
   T * p = new T[2*capacity];
   if(!p)          return false;
   for(int i = 0 ; i<capacity;i++)
      p[i] = data[i];
   capacity *=2;


   return true;
}
```



realloc

- 申请更大的内存块

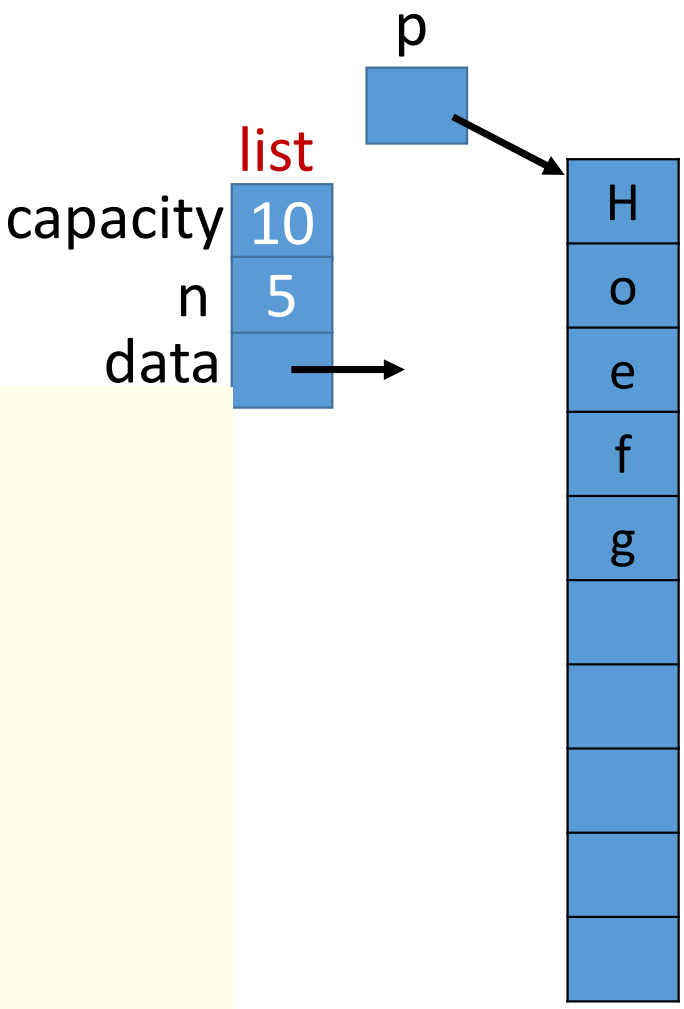
```
bool SqList<T>::realloc(){
   T * p = new T[2*capacity];
   if(!p)          return false;
   for(int i = 0 ; i<capacity;i++)
      p[i] = data[i];
   capacity *=2;
   delete[] data;
   return true;
}
```



realloc

- 申请更大的内存块

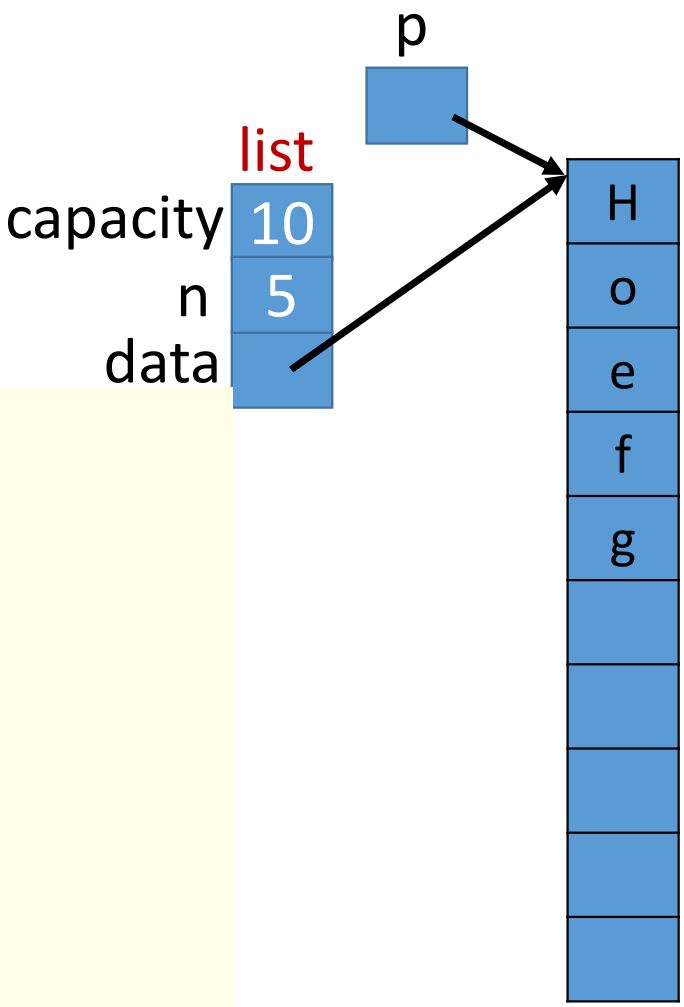
```
bool SqList<T>::realloc(){
   T * p = new T[2*capacity];
   if(!p)          return false;
   for(int i = 0 ; i<capacity;i++)
      p[i] = data[i];
   capacity *=2;
   delete[] data; data = p;
   return true;
}
```



realloc

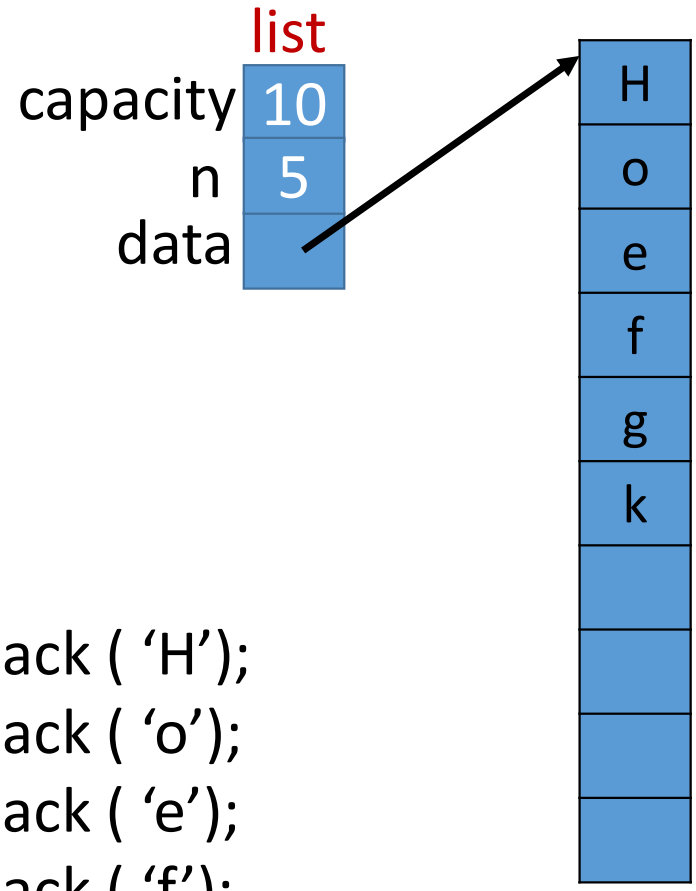
- 申请更大的内存块

```
bool SqList<T>::realloc(){
   T * p = new T[2*capacity];
   if(!p)          return false;
   for(int i = 0 ; i<capacity;i++)
      p[i] = data[i];
   capacity *=2;
   delete[] data; data = p;
   return true;
}
```



push_back

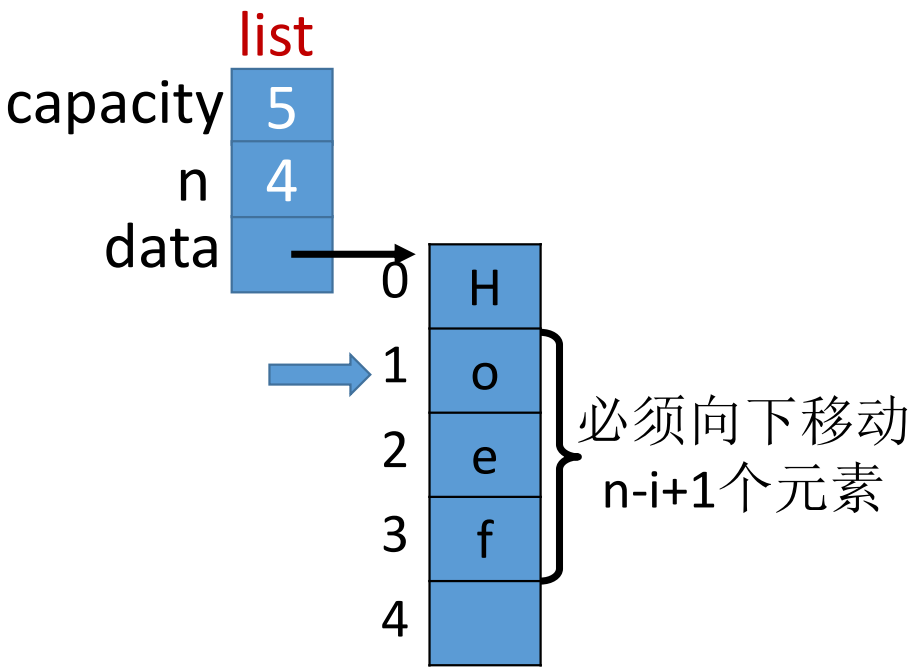
```
bool SqList<T>::push_back(T e){
  if(n >= capacity )
     if(!realloc()) return false;
  data[n] = e;
  n++;
  return true;
}
```




```
list.push_back ( 'H');
list.push_back ( 'o');
list.push_back ( 'e');
list.push_back ( 'f');
list.push_back ( 'g');
list.push_back ( 'k');
```

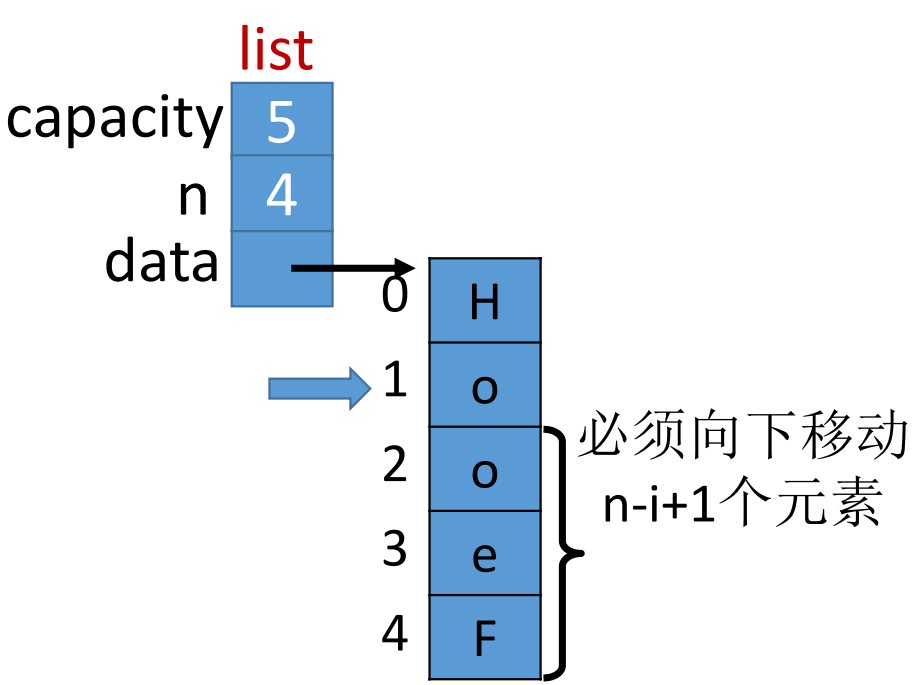
insert

Insert(2,' Q')



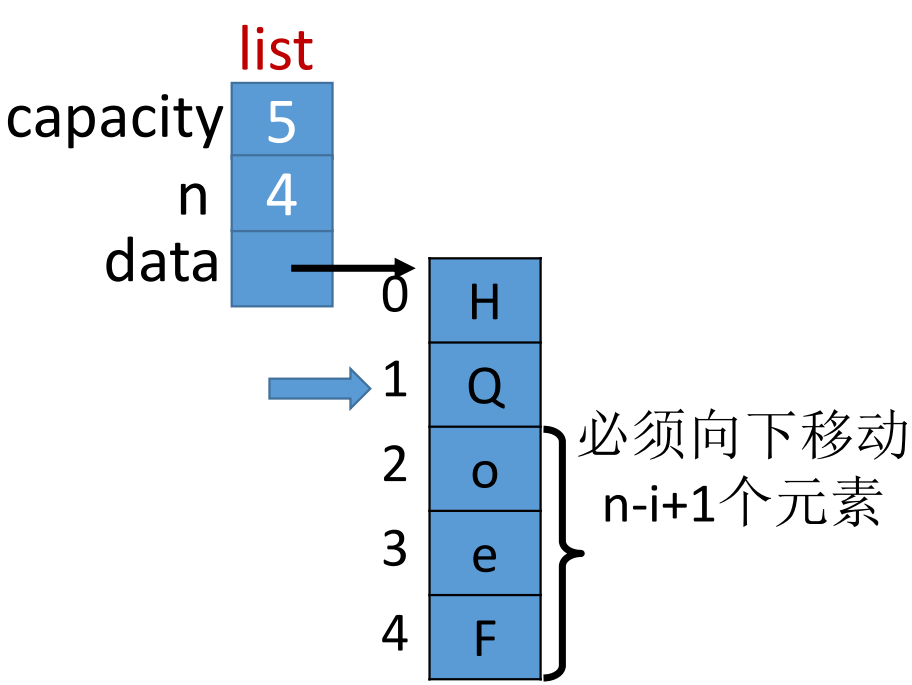
insert

Insert(2,' Q')



insert

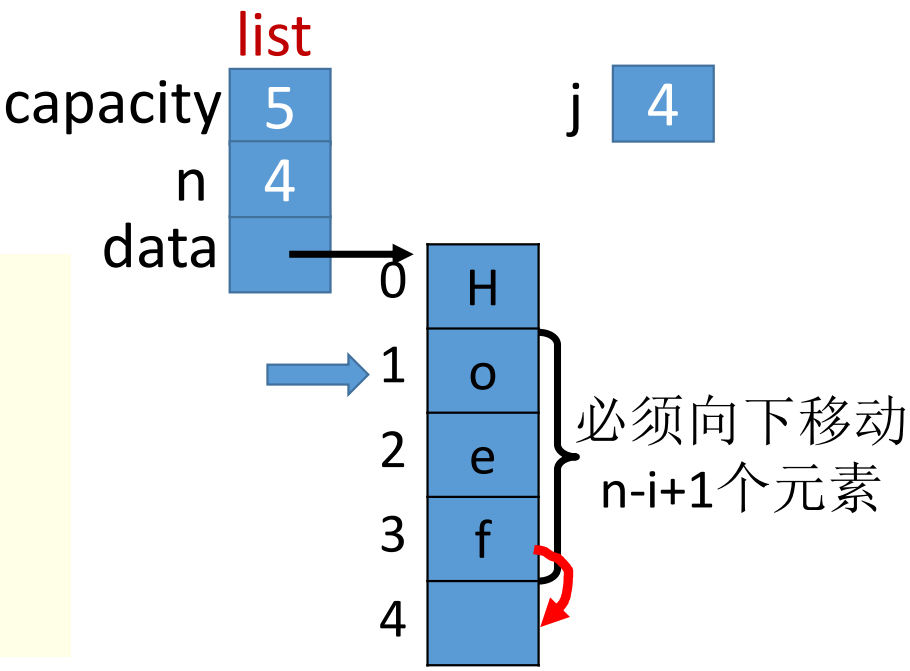
Insert(2,' Q')



insert

Insert(2,' Q')

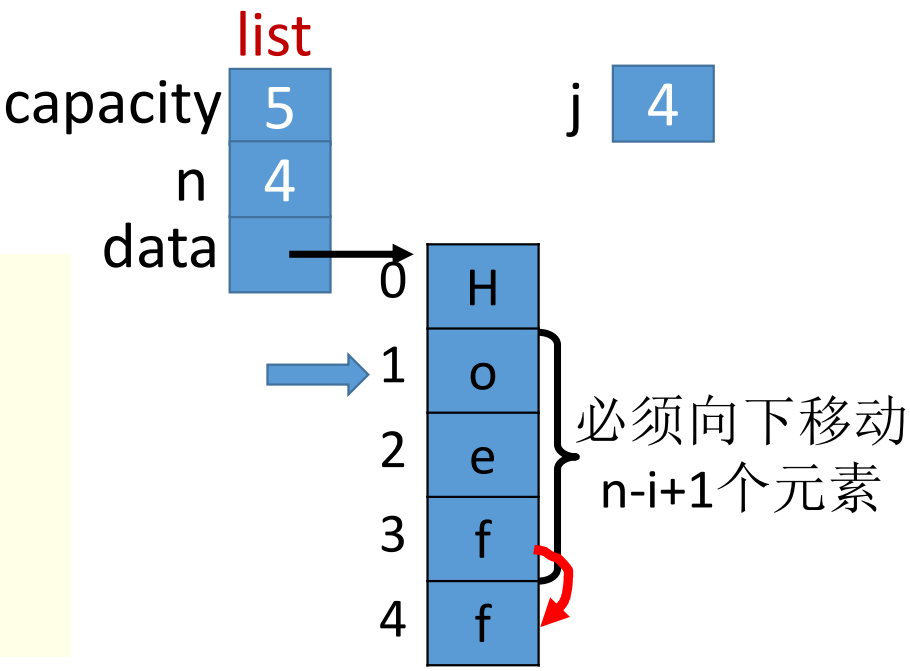
```
bool SqList<T>::insert(int i, T e){
    for(int j = n;          )
            data[j] = data[j-1];

}
```



insert

Insert(2,' Q')

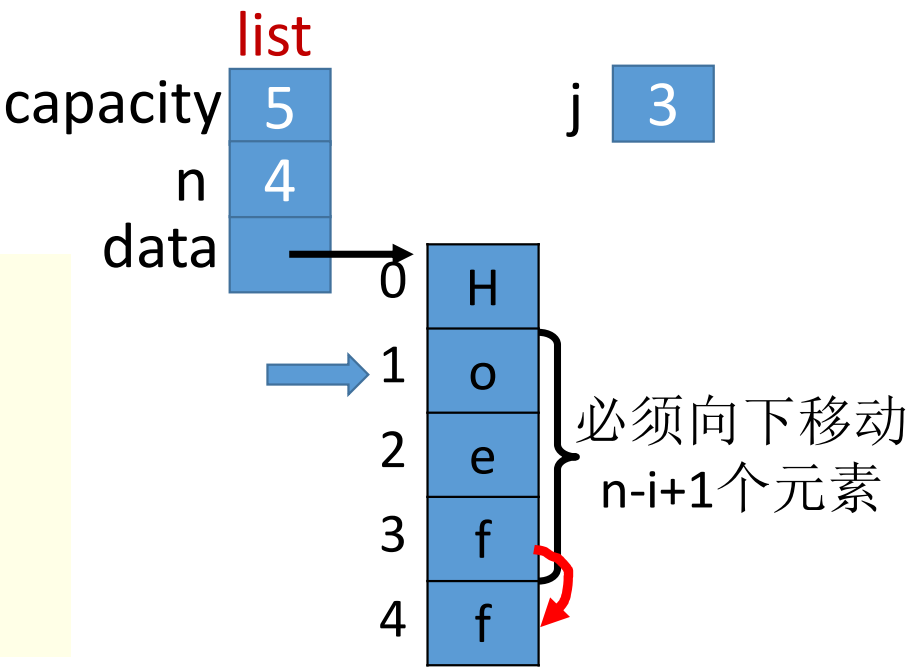
```
bool SqList<T>::insert(int i, T e){
    for(int j = n;        j--)
            data[j] = data[j-1];

}
```



insert

Insert(2,' Q')

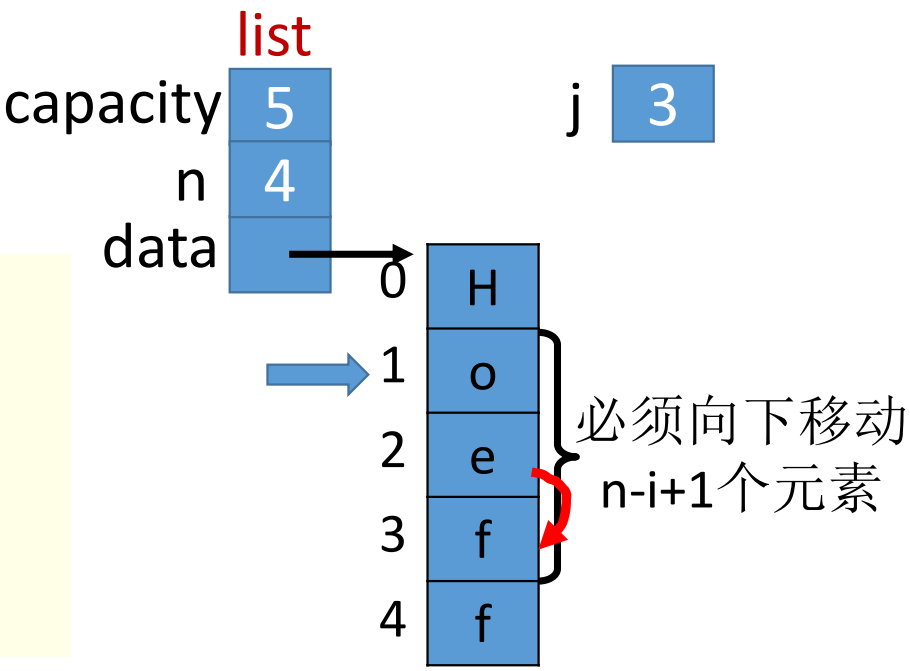
```
bool SqList<T>::insert(int i, T e){
    for(int j = n;  j>=i;  j--)
        data[j] = data[j-1];

}
```



insert

Insert(2,' Q')

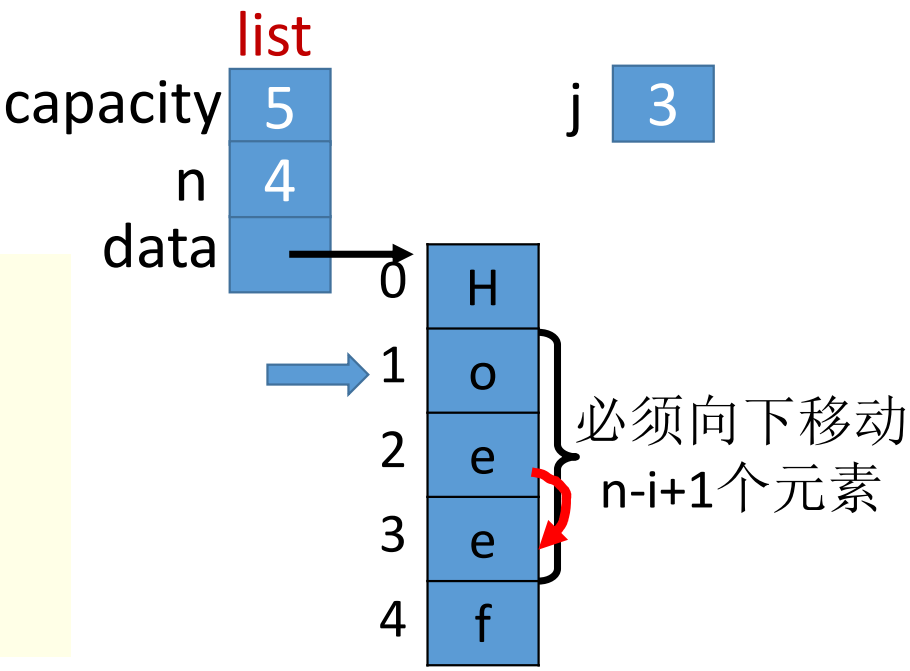
```
bool SqList<T>::insert(int i, T e){
    for(int j = n;  j>=i;  j--)
        data[j] = data[j-1];

}
```



insert

Insert(2,' Q')

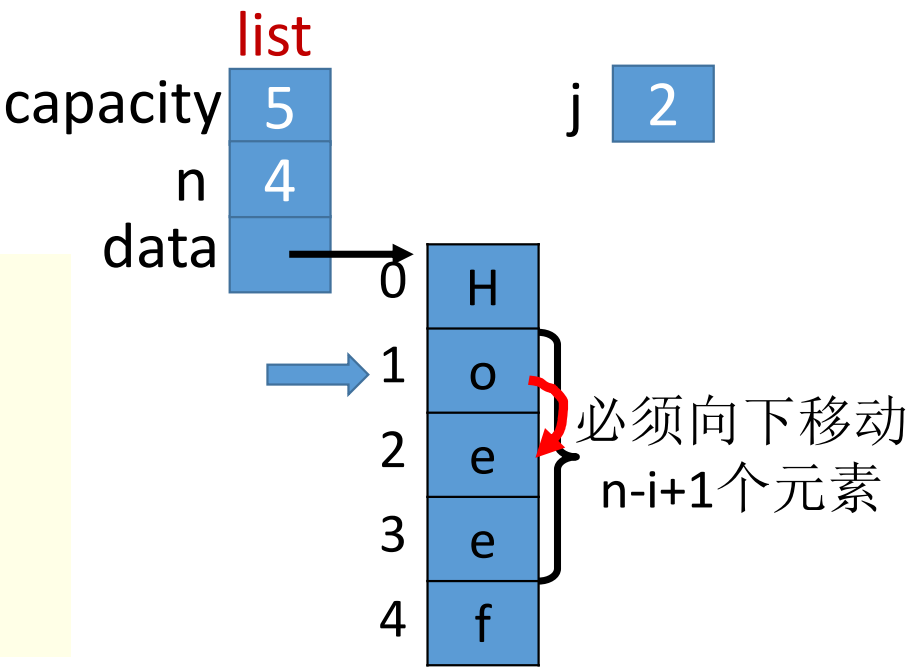
```
bool SqList<T>::insert(int i, T e){
    for(int j = n;  j>=i;  j--)
        data[j] = data[j-1];

}
```



insert

Insert(2,' Q')

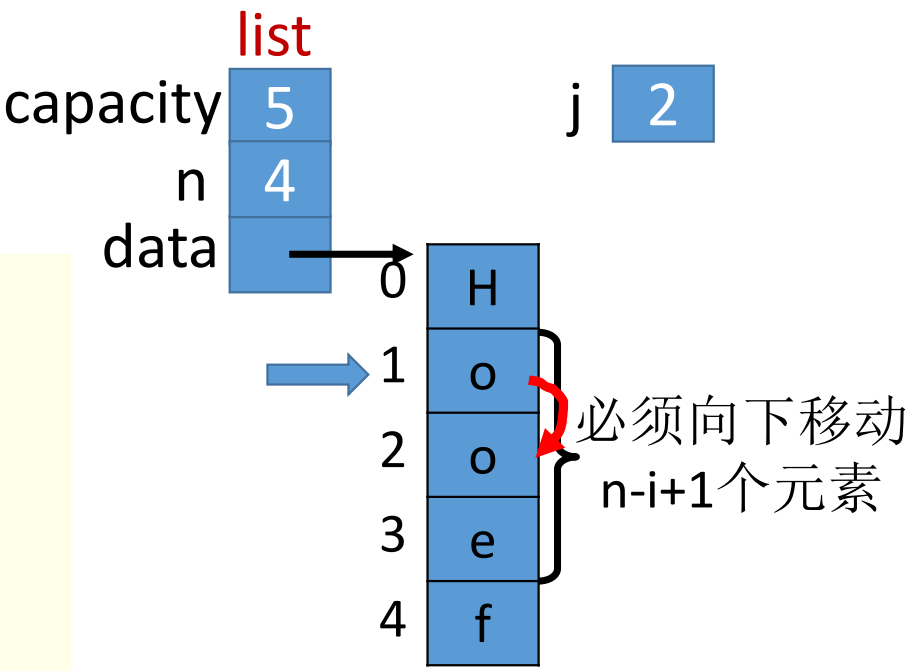
```
bool SqList<T>::insert(int i, T e){
    for(int j = n;  j>=i;  j--)
        data[j] = data[j-1];

}
```



insert

Insert(2,' Q')

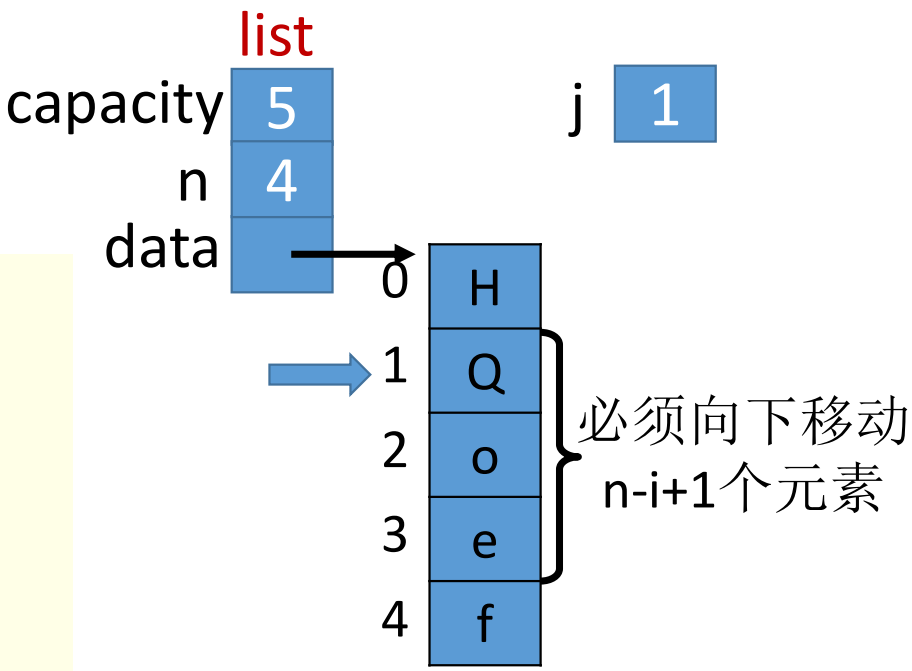
```
bool SqList<T>::insert(int i, T e){
    for(int j = n;  j>=i;  j--)
        data[j] = data[j-1];


}
```



insert

Insert(2,' Q')

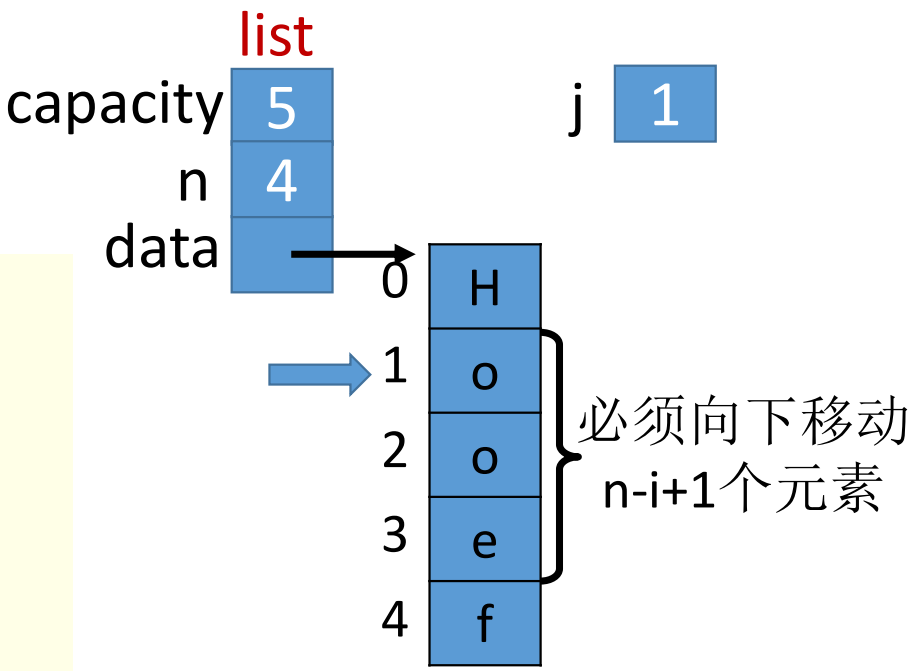
```
bool SqList<T>::insert(int i, T e){
    for(int j = n;  j>=i;  j--)
            data[j] = data[j-1];
    data[i-1] = e;

}
```



insert

Insert(2,' Q') Q

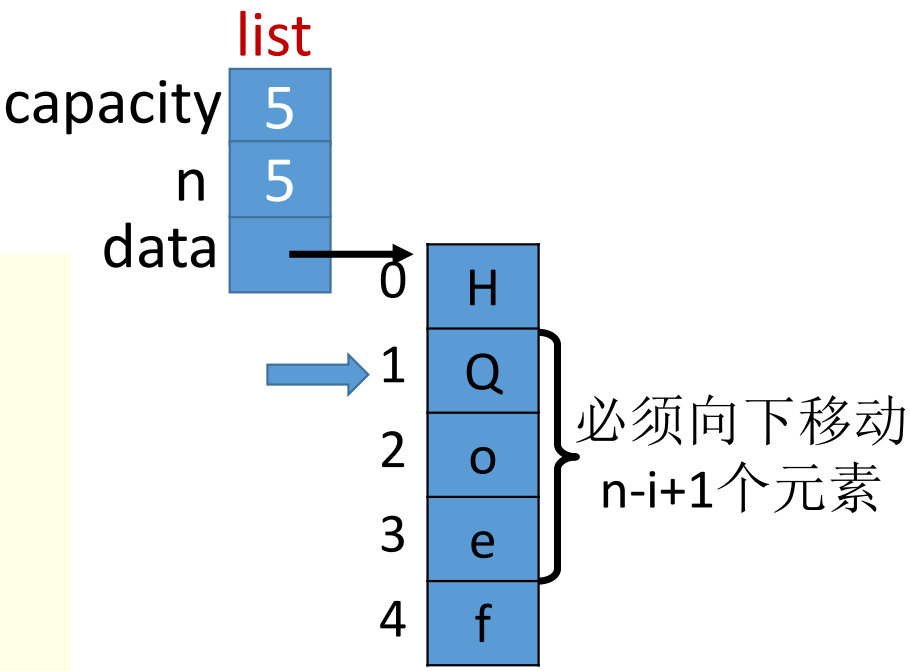
```
bool SqList<T>::insert(int i, T e){
    for(int j = n;  j>=i;  j--)
            data[j] = data[j-1];
    data[i-1] = e;

}
```



insert

Insert(2,' Q')

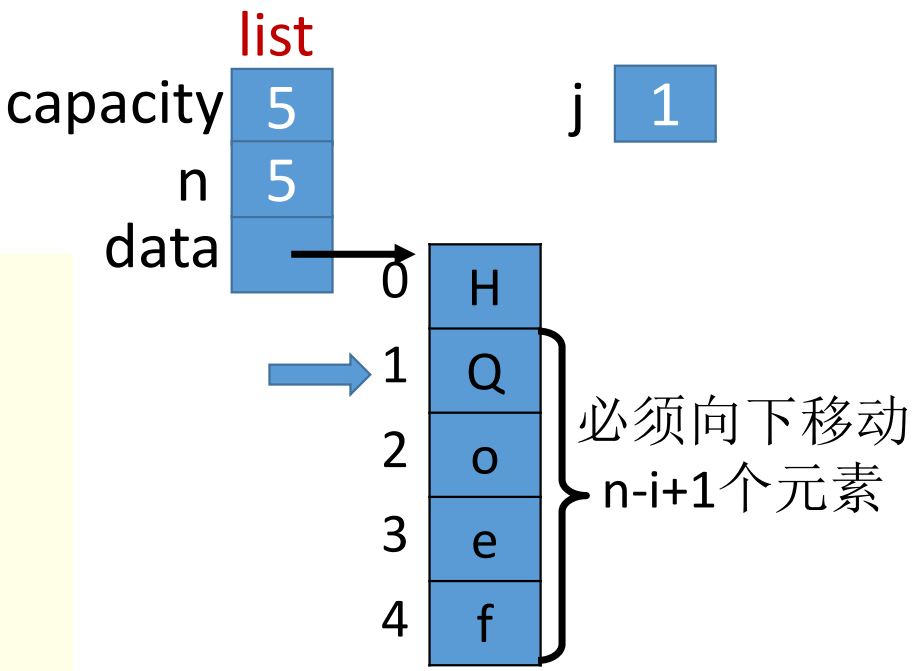
```
bool SqList<T>::insert(int i, T e){
    for(int j = n;  j>=i;  j--)
            data[j] = data[j-1];
    data[i-1] = e;
    n++;
}
```



insert

Insert(2,' P')

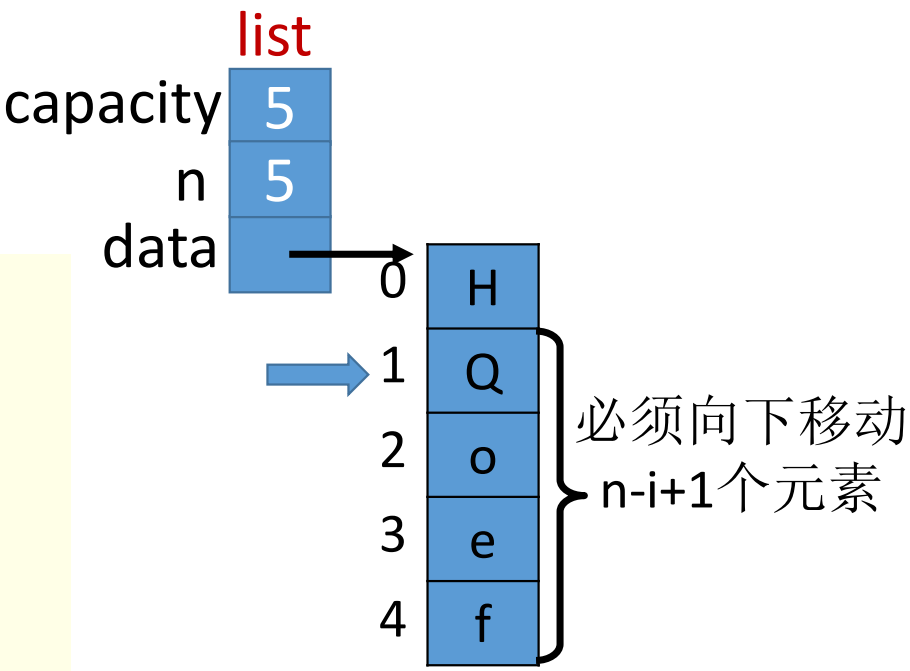
```
bool SqList<T>::insert(int i, T e){

    for(int j = n;  j>=i;  j--)
            data[j] = data[j-1];
    data[i-1] = e;
    n++;

}
```



insert

Insert(2,' P')

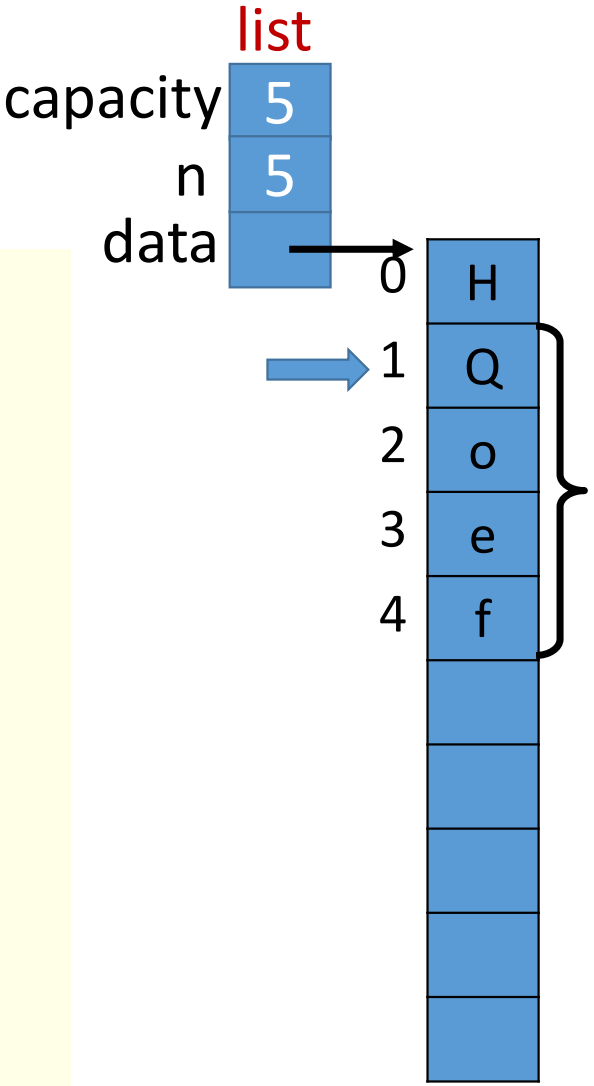
```
bool SqList<T>::insert(int i, T e){
    if(i < 1 || i > n + 1)      //i是否合法
        return false;
    if(n >= capacity ){ //分配更大内存块
        if(!realloc()) return false; }
    }
    for(int j = n;  j>=i;  j--)
        data[j] = data[j-1];
    data[i-1] = e;
    n++;
}
```



insert

Insert(2,' P')

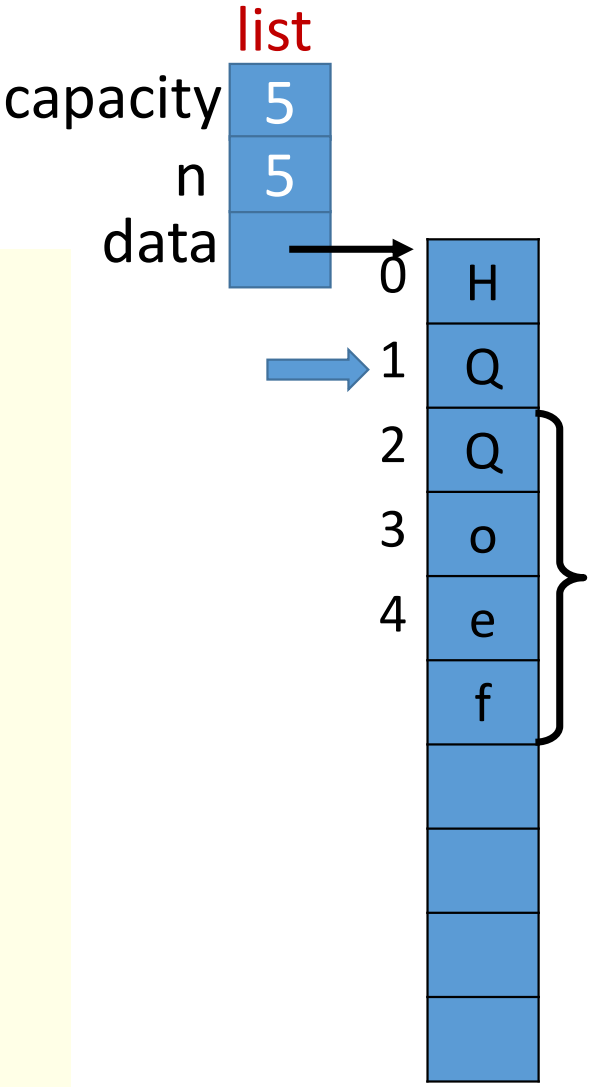
```
bool SqList<T>::insert(int i, T e){
    if(i < 1 || i > n + 1)      //i是否合法
        return false;
    if(n >= capacity ){ //分配更大内存块
        if(!realloc()) return false; }
    }
    for(int j = n;  j>=i;  j--)
        data[j] = data[j-1];
    data[i-1] = e;
    n++;
}
```



insert

Insert(2,' P')

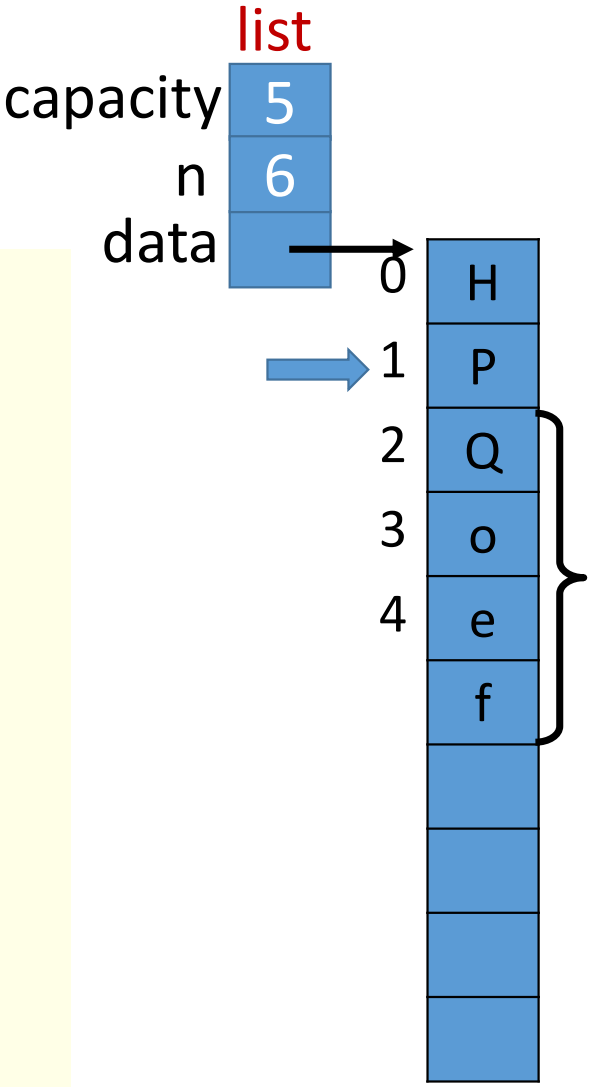
```
bool SqList<T>::insert(int i, T e){
    if(i < 1 || i > n + 1)      //i是否合法
        return false;
    if(n >= capacity ){ //分配更大内存块
        if(!realloc()) return false; }
    }
    for(int j = n;  j>=i;  j--)
        data[j] = data[j-1];
    data[i-1] = e;
    n++;
}
```



insert

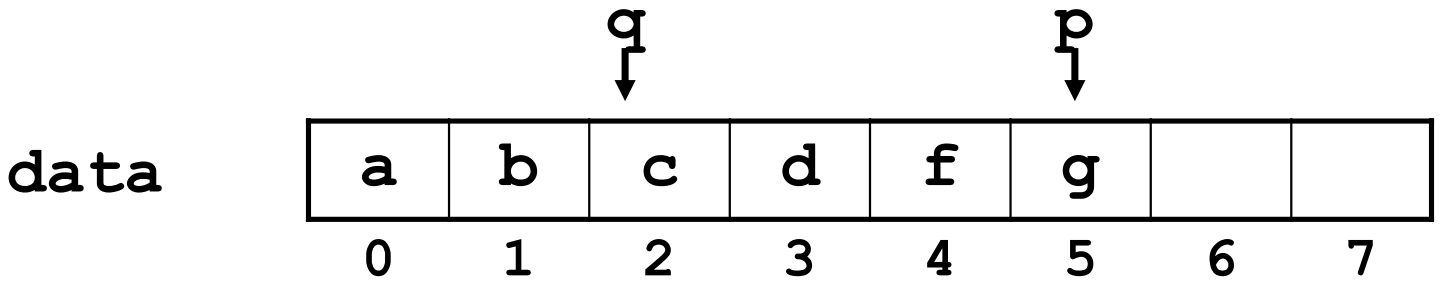
Insert(2,' P')

```
bool SqList<T>::insert(int i, T e){
    if(i < 1 || i > n + 1)      //i是否合法
        return false;
    if(n >= capacity ){ //分配更大内存块
        if(!realloc()) return false; }
    }
    for(int j = n;  j>=i;  j--)
        data[j] = data[j-1];
    data[i-1] = e;
    n++;
}
```



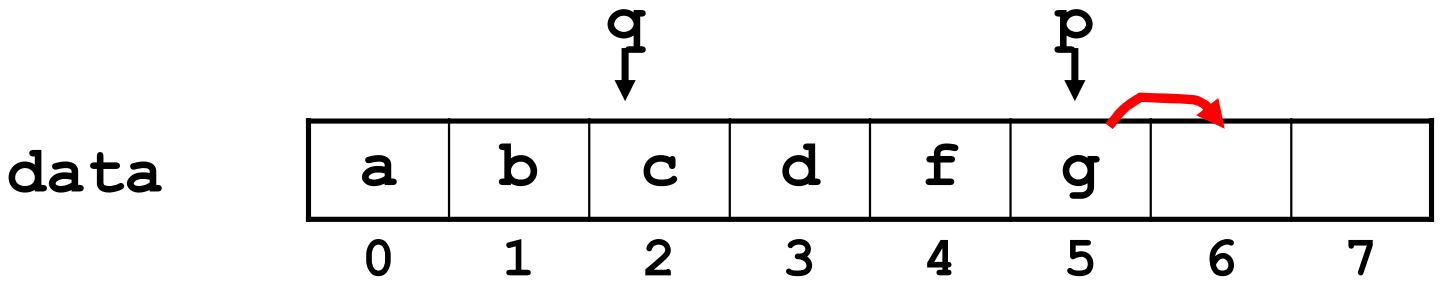
```
        T *q = &(data[i-1]);    // q指向插入的位置
        // p指向最后一个元素,
        // 从p到q的所有元素后移一个单元
        for(T* p=&(data[n-1]); p >= q; --p)
             *(p+1) = *p;



}
```

i=3，即插入在第三个元素之前



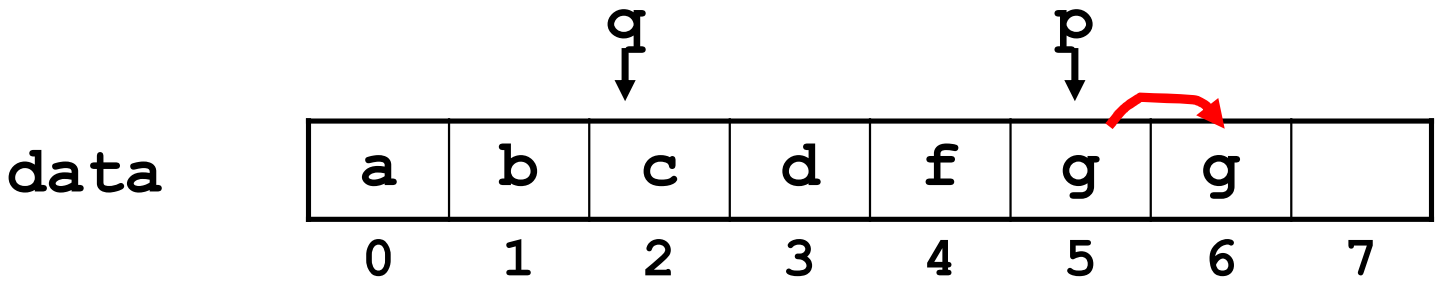
```
        T* q = &(data[i-1]);    // q指向插入的位置
        // p指向最后一个元素,
        // 从p到q的所有元素后移一个单元
        for(T* p=&(data[n-1]); p>=q; --p)
             *(p+1) = *p;



}
```

i=3，即插入在第三个元素之前



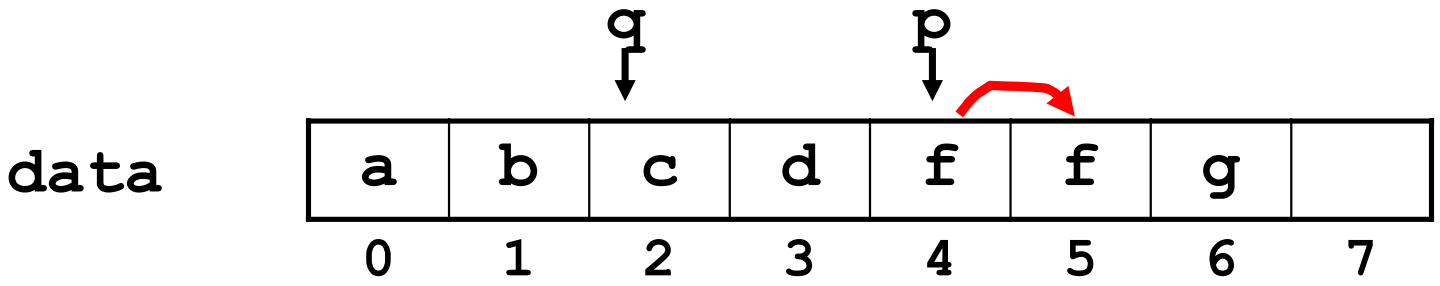
```
        T* q = &(data[i-1]);    // q指向插入的位置
        // p指向最后一个元素,
        // 从p到q的所有元素后移一个单元
        for(T* p=&(data[n-1]); p>=q; --p)
             *(p+1) = *p;



}
```

i=3，即插入在第三个元素之前



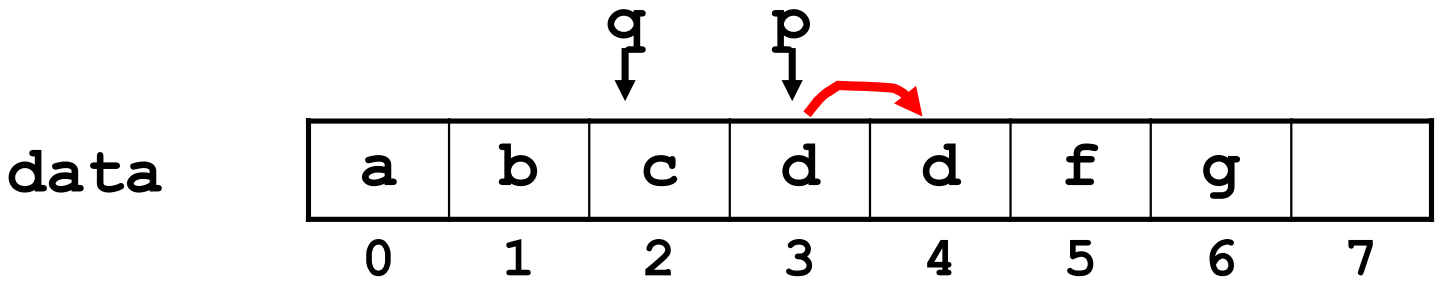
```
        T* q = &(data[i-1]);    // q指向插入的位置
        // p指向最后一个元素,
        // 从p到q的所有元素后移一个单元
        for(T* p=&(data[n-1]); p>=q; --p)
             *(p+1) = *p;



}
```

i=3，即插入在第三个元素之前



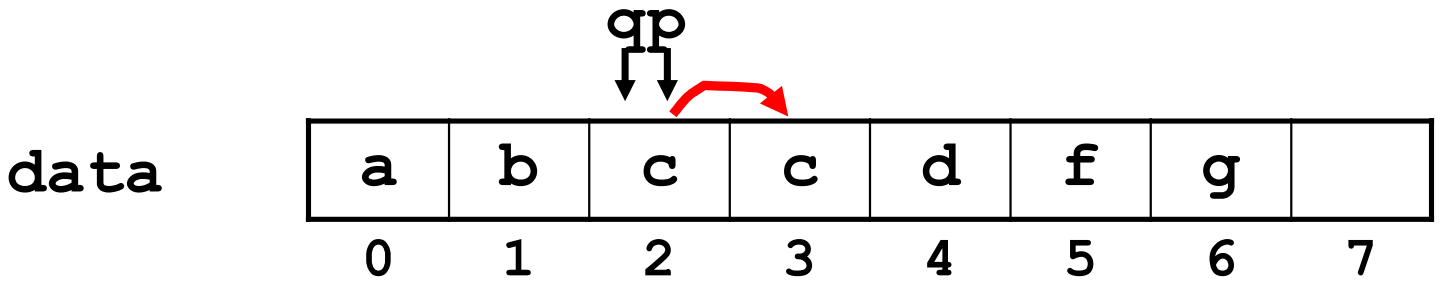
```
        T* q = &(data[i-1]);    // q指向插入的位置
        // p指向最后一个元素,
        // 从p到q的所有元素后移一个单元
        for(T* p=&(data[n-1]); p>=q; --p)
             *(p+1) = *p;



}
```

i=3，即插入在第三个元素之前



```
        T* q = &(data[i-1]);    // q指向插入的位置
        // p指向最后一个元素,
        // 从p到q的所有元素后移一个单元
        for(T* p=&(data[n-1]); p>=q; --p)
             *(p+1) = *p;



}
```

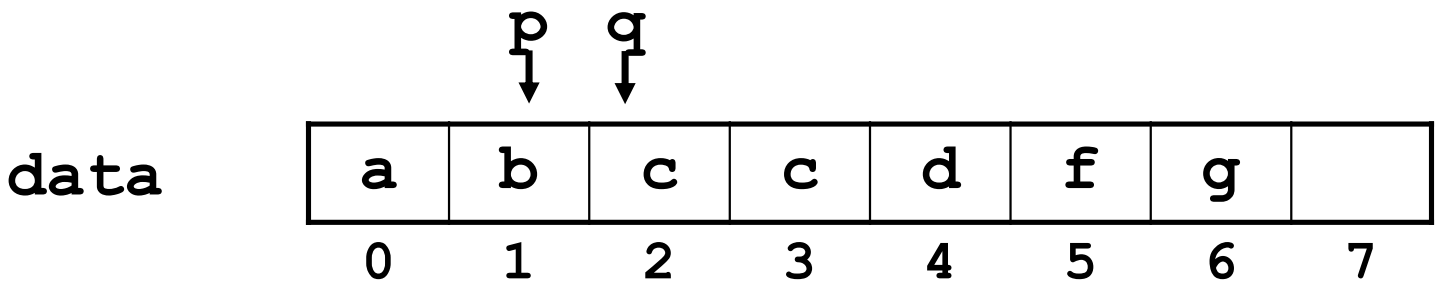
i=3，即插入在第三个元素之前



```
        T* q = &(data[i-1]);    // q指向插入的位置
        // p指向最后一个元素,
        // 从p到q的所有元素后移一个单元
        for(T* p=&(data[n-1]); p>=q; --p)
             *(p+1) = *p;
        *q = e;                 // 写进待插入的元素e


}
```

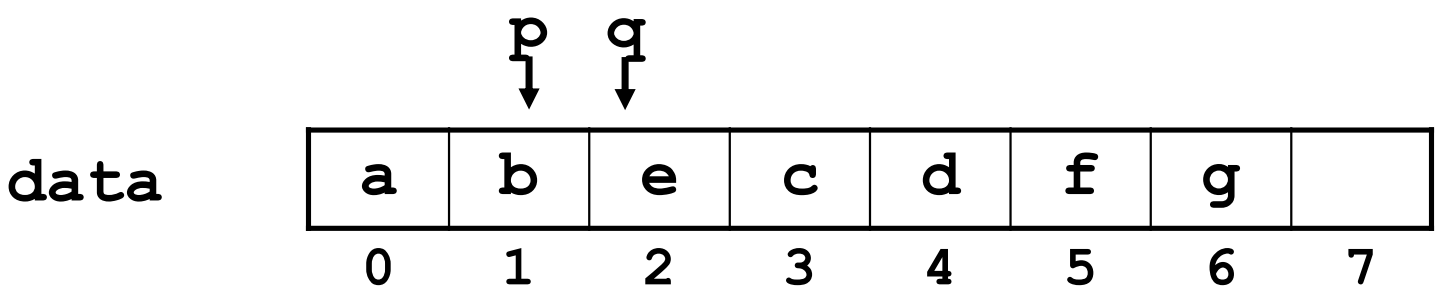
e

i=3，即插入在第三个元素之前



```
        T* q = &(data[i-1]);    // q指向插入的位置
        // p指向最后一个元素,
        // 从p到q的所有元素后移一个单元
        for(T* p=&(data[n-1]); p>=q; --p)
            *(p+1) = *p;
        *q = e;                 // 写进待插入的元素e
        ++n;      // 表长加1
        return true;
}
```

i=3，即插入在第三个元素之前



插入操作的时间复杂度

- 很显然，插入操作的复杂度由需要移动的元素的个数决定
- 而需要移动元素的个数由插入位置决定

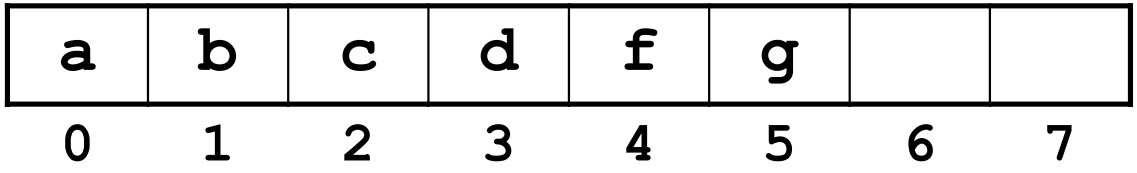
  i = n+1时，需要移动0个

  i = n时：1个

  …

  i = 1时：n个

即：需要移动的元素个数 = n+1-i



插入操作的时间复杂度

- 最差情况

  T(n) = O(n)

- 平均情况呢?

一共有1,2,...,n+1，n+1个可能的插入位置，在第i个位置上插入的概率是1/(n+1).

所以平均需要移动元素的个数

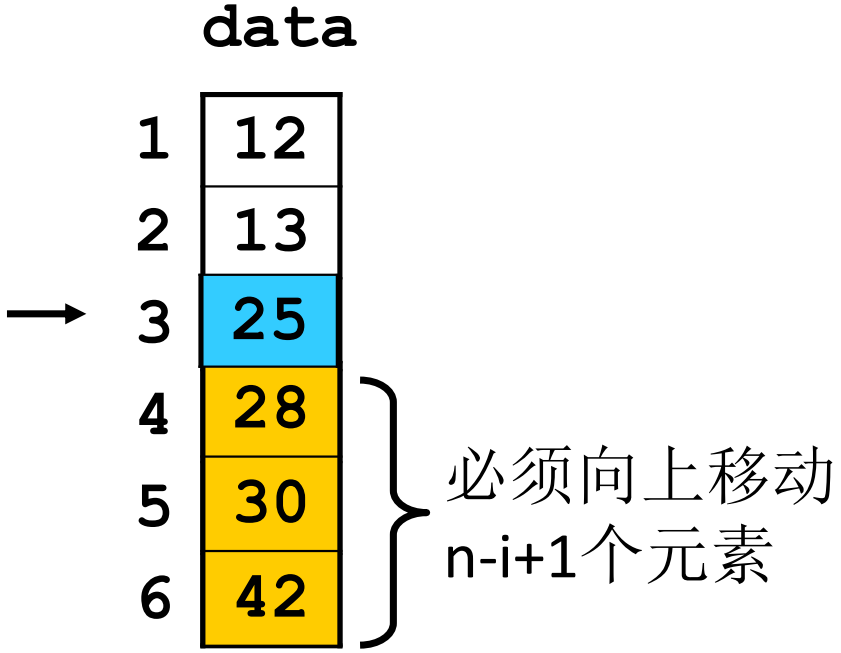
$$= \frac{1}{n+1}\sum_{i=1}^{n+1}(n+1-i) = \frac{1}{n+1}\times\frac{n(n+1)}{2} = \frac{n}{2}$$

所以平均时间复杂度 = O(n)

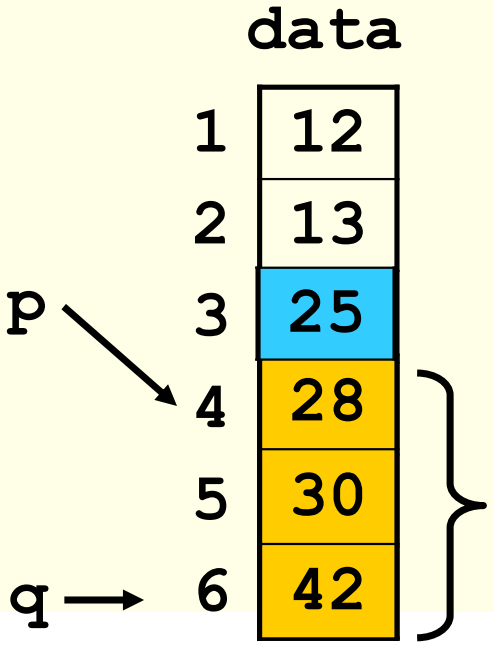
remove (int i)

- 删除第i个元素

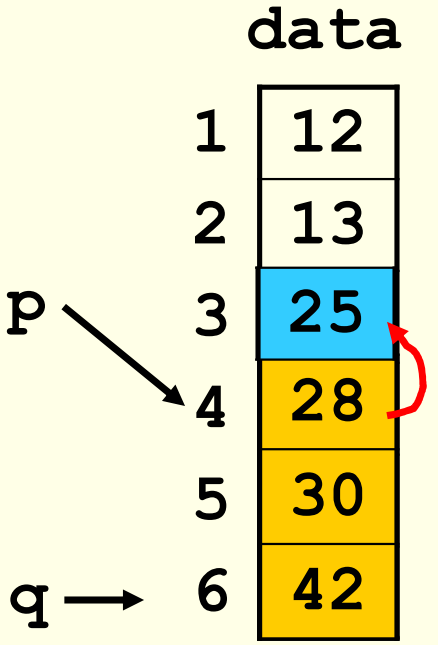
  i = 3



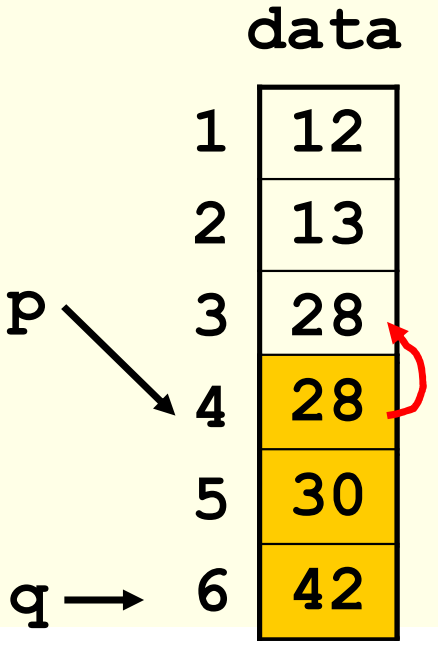
```
bool SqList<T>::remove( int i){
    if((i < 1) || (i > n))
        return false;
    T *p,*q;
    p = &(data[i]); //p指向被删除的节点

    // q指向最后一个节点
    q = data + n - 1;




}
```



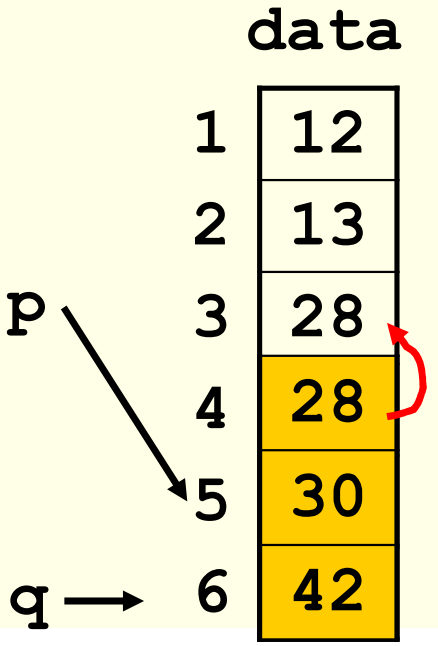
```
bool SqList<T>::remove( int i){
    if((i < 1) || (i > n))
        return false;
     T *p,*q;
     p = &(data[i]); //p指向被删除的节点

    // q指向最后一个节点
    q = data + n - 1;
    // 从p+1到q的所有节点前移一个单元
    for(    ; p <= q; ++ p)
        *(p-1) = *p;

    return true;
}
```



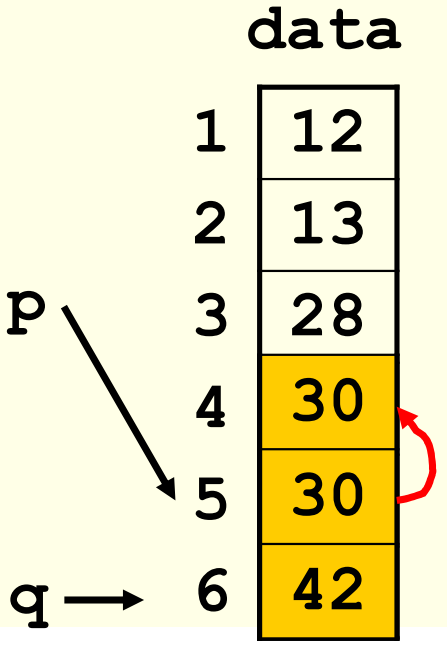
```
bool SqList<T>::remove( int i){
    if((i < 1) || (i > n))
        return false;
     T *p,*q;
     p = &(data[i]); //p指向被删除的节点

    // q指向最后一个节点
    q = data + n - 1;
    // 从p+1到q的所有节点前移一个单元
    for(    ; p <= q; ++ p)
        *(p-1) = *p;

    return true;
}
```



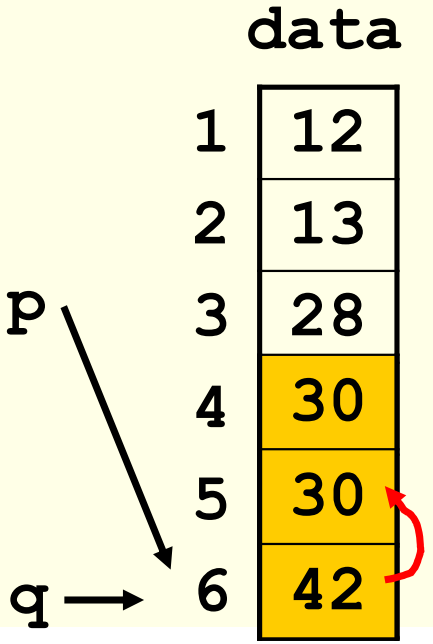
```
bool SqList<T>::remove( int i){
    if((i < 1) || (i > n))
        return false;
     T *p,*q;
     p = &(data[i]); //p指向被删除的节点

    // q指向最后一个节点
    q = data + n - 1;
    // 从p+1到q的所有节点前移一个单元
    for(     ; p <= q; ++ p)
        *(p-1) = *p;

    return true;
}
```



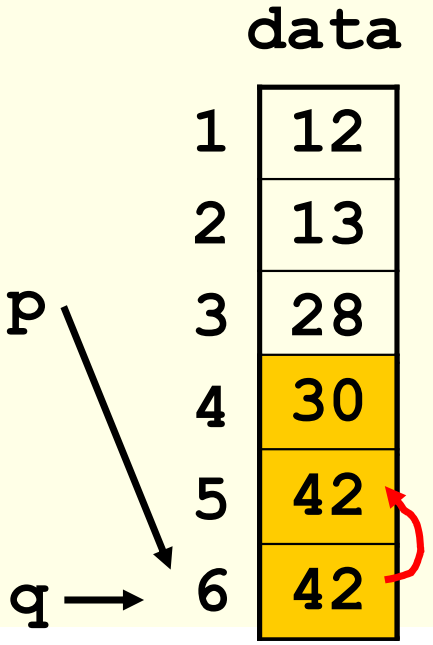
```
bool SqList<T>::remove( int i){
    if((i < 1) || (i > n))
        return false;
     T *p,*q;
     p = &(data[i]); //p指向被删除的节点

    // q指向最后一个节点
    q = data + n - 1;
    // 从p+1到q的所有节点前移一个单元
    for( ; p <= q; ++ p)
        *(p-1) = *p;

    return true;
}
```



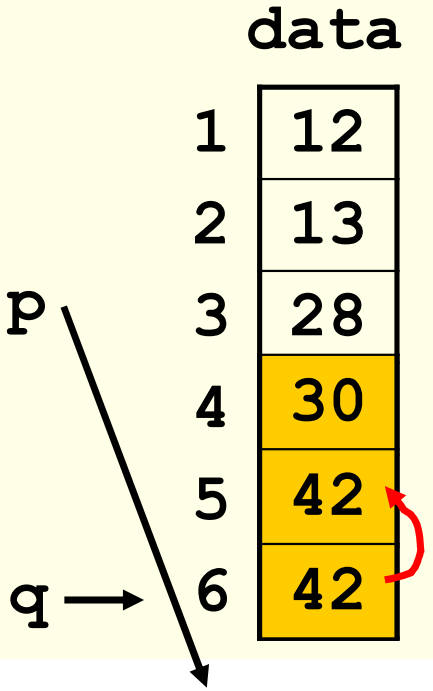
```
bool SqList<T>::remove( int i){
    if((i < 1) || (i > n))
        return false;
     T *p,*q;
     p = &(data[i]); //p指向被删除的节点

    // q指向最后一个节点
    q = data + n - 1;
    // 从p+1到q的所有节点前移一个单元
    for( ; p <= q; ++ p)
        *(p-1) = *p;

    return true;
}
```



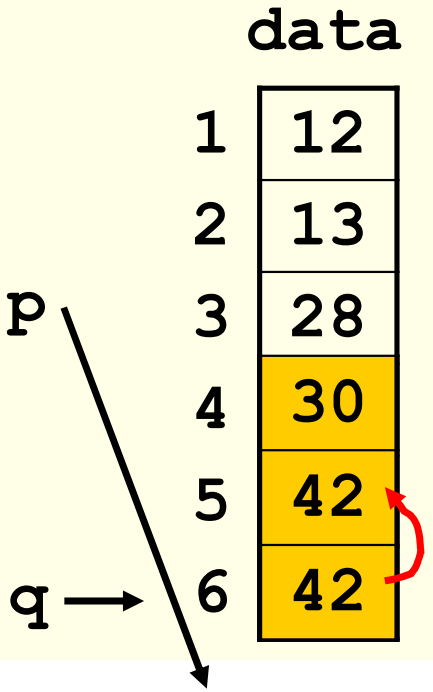
```
bool SqList<T>::remove( int i){
    if((i < 1) || (i > n))
        return false;
     T *p,*q;
     p = &(data[i]); //p指向被删除的节点

    // q指向最后一个节点
    q = data + n - 1;
    // 从p+1到q的所有节点前移一个单元
    for( ; p <= q; ++ p)
        *(p-1) = *p;

    return true;
}
```



```
bool SqList<T>::remove( int i){
    if((i < 1) || (i > n))
        return false;
     T *p,*q;
     p = &(data[i]); //p指向被删除的节点

    // q指向最后一个节点
    q = data + n - 1;
    // 从p+1到q的所有节点前移一个单元
    for( ; p <= q; ++ p)
        *(p-1) = *p;
    --n;              // 表长减1
    return true;
}
```



```
bool SqList<T>::remove( int i){
    if((i < 1) || (i > n))
        return false;
     T *p,*q;
     p = &(data[i]); //p指向被删除的节点

    // q指向最后一个节点
    q = data + n - 1;
    // 从p+1到q的所有节点前移一个单元
    for( ; p <= q; ++ p)
        *(p-1) = *p;
    --n;              // 表长减1
    return true;
}
```



删除操作的时间复杂度

- 最差情况

  T(n) = O(n)

- 平均情况呢?

  一共有（1,2,...,n）n个可能的删除位置，在第i个位置上删除的概率是1/n.

  所以平均需要移动元素的个数

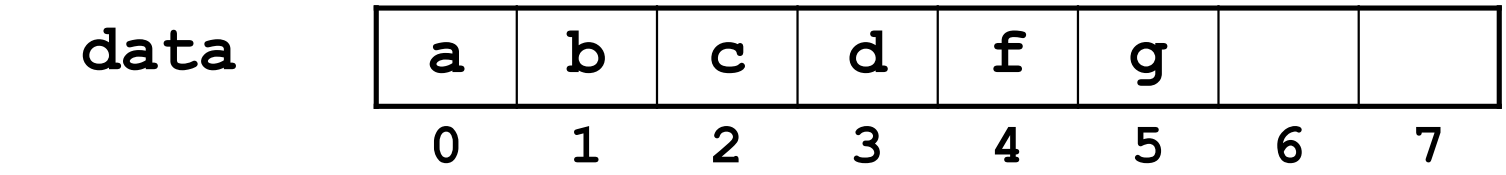
$$= \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{1}{n} \times \frac{n(n-1)}{2} = \frac{n-1}{2}$$

  所以平均时间复杂度 = O(n)

set(int i, T e)

- 序号为i的元素下标是i-1, 即 data[i-1]。
- 但要检查序号是否合法?

```
bool SqList<T>::set( int i, T e){
    if((i < 1) || (i > n))
        return false;
    data[i-1] = e;
    return true;
}
```



```
bool SqList<T>::get( int i, T &e){
    if((i < 1) || (i > n))
        return false;
    e = data[i-1];
    return true;
}
```

```
int SqList<T>::size( ){
    return n;
}
```

时间复杂度：O(1) 常数时间

int SqList<T>::find(T e)

- 条件可以很多，比如关键字相同

```
int SqList<T>::find(T e){
    for(int i=0 ; i < n ;i++)
       if(data[i]==e)
         return i+1;        //成功返回该元素序号
    return 0;               //失败
}
```

顺序表Array List

- 特点

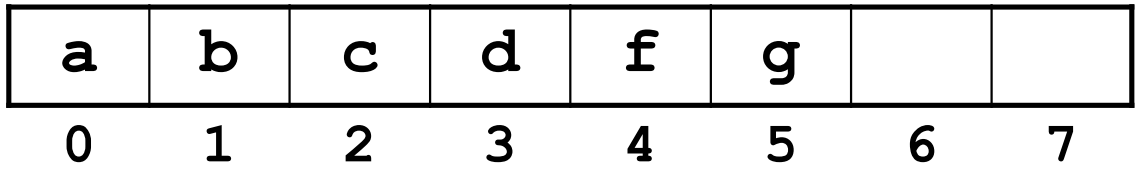
  各单元的内存地址连续

- 优点

  可**随机访问**任一元素, 即访问任何一个元素所用时间都相同

- 缺点

  插入、删除操作需移动大量元素，算法复杂度 = O(n)



实验一：顺序表的实现及测试

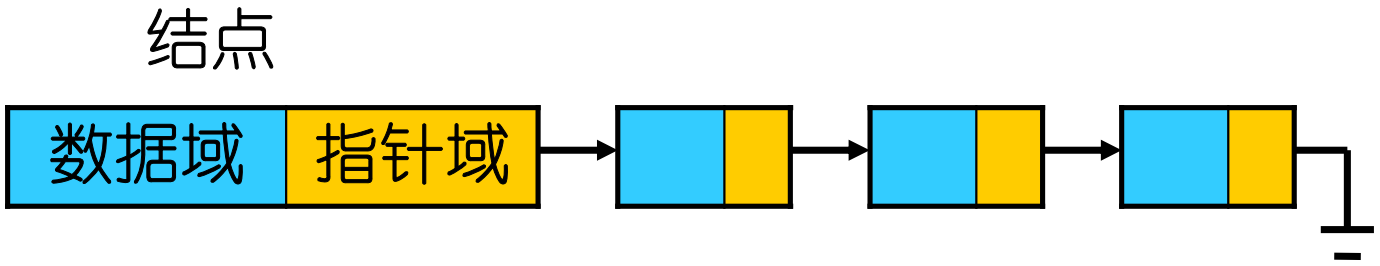
- 编写实现线性表的类模板SqList，并测试其功能

线性表的链式实现：链式表 LinkedList

- 特点：

  每个元素的存储地址任意,用一个结点存储一个数据元素。

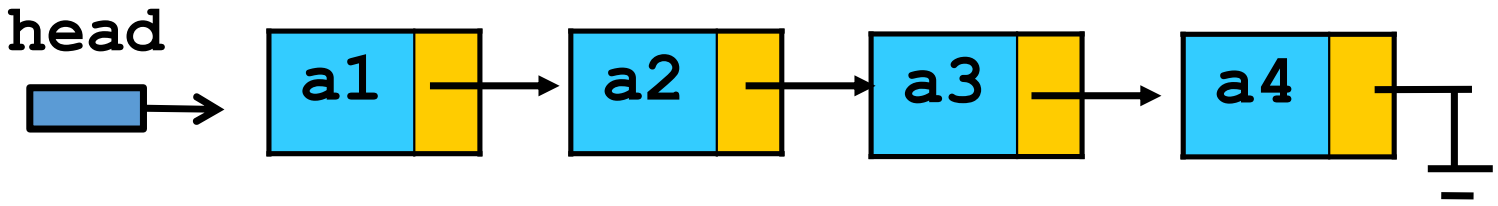
  每个元素的结点表示中还包括指向下一个元素结点位置的指针



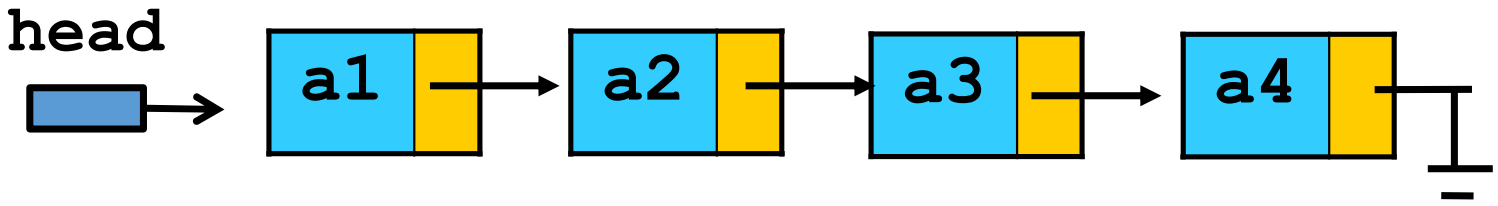
链式表LinkedList

- 结点包括：数据元素值　和 指向下一个元素结点的指针

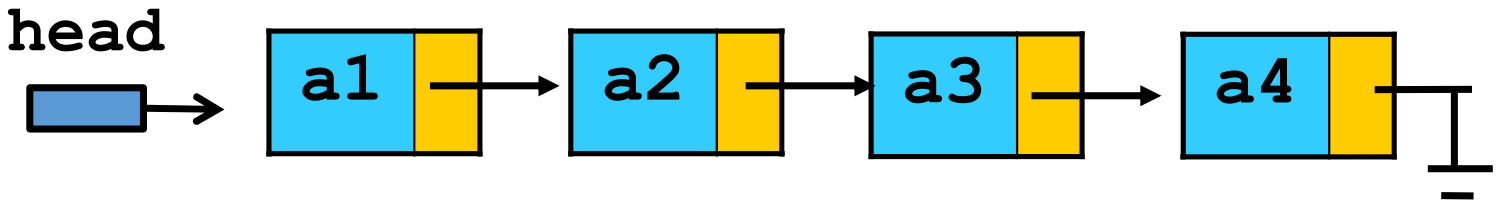
```
struct LNode{
    T data;
    LNode *next;
};
LNode* head;
```



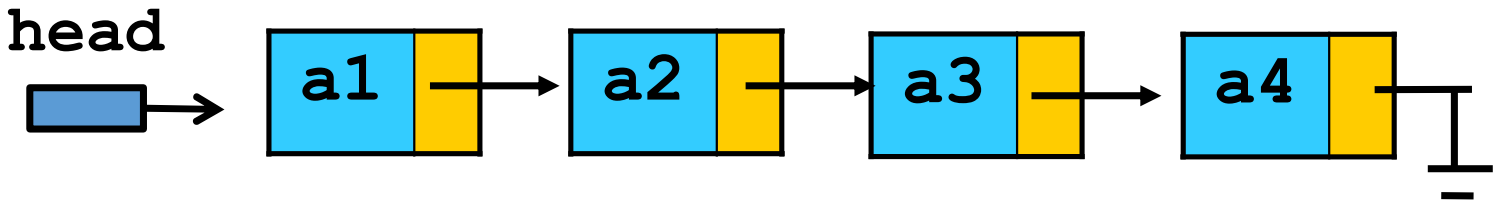
```
template<class T>
class LkList{
    struct LNode{
        T data;
        LNode *next;
    };
    LNode *head;
public:
    LkList() ;
    virtual  ~LkList() ;
```



```
bool insert (int i, T e) ;
bool remove (int i);
bool insert_front (T e);
bool push_back (T e);
bool remove_front ( );
bool pop_back ( );
```



```
    bool get (int i, T &e);
    bool set (int i, T e);
    int size () ;
    int find ( (*fun)());
    void traverse ( (*fun)());
    …
}
```

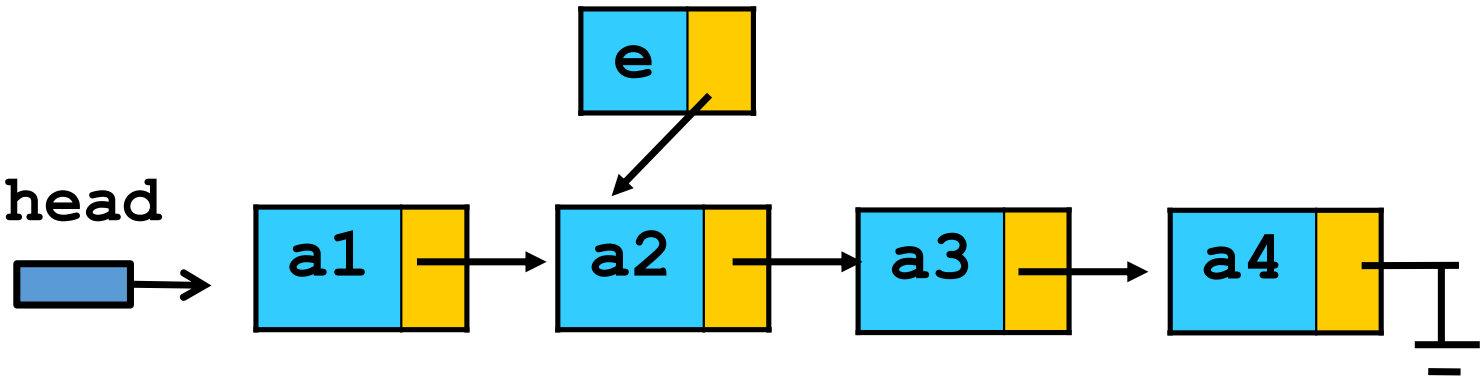


链式表LinkedList

- 第一个结点与其他结点不一样，前面没有“前驱结点”

  各种操作（如插入）需要区分对待是否是第一个结点还是其他结点。

insert(2,e)

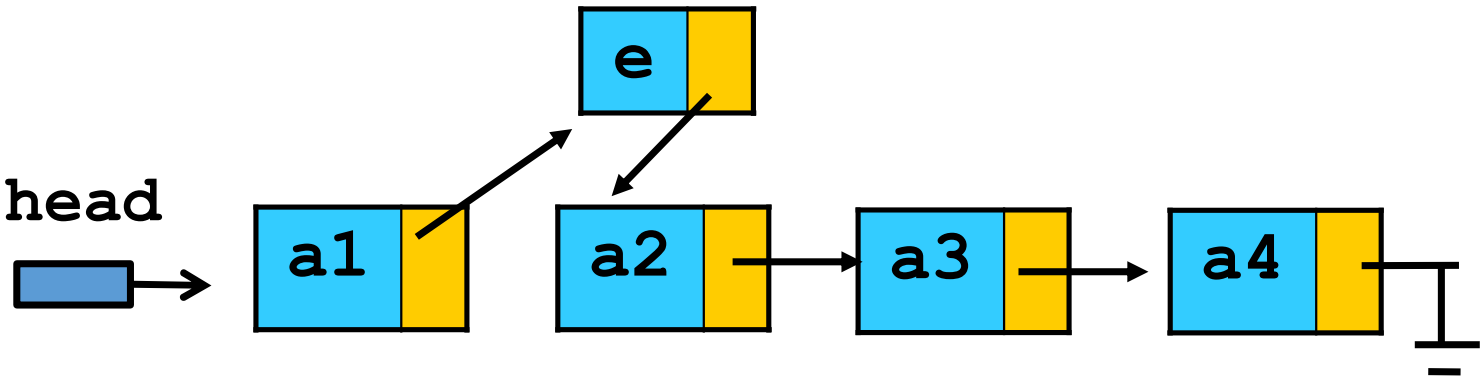


链式表LinkedList

- 第一个结点与其他结点不一样，前面没有“前驱结点”

  各种操作（如插入）需要区分对待是否是第一个结点还是其他结点。

insert(2,e)

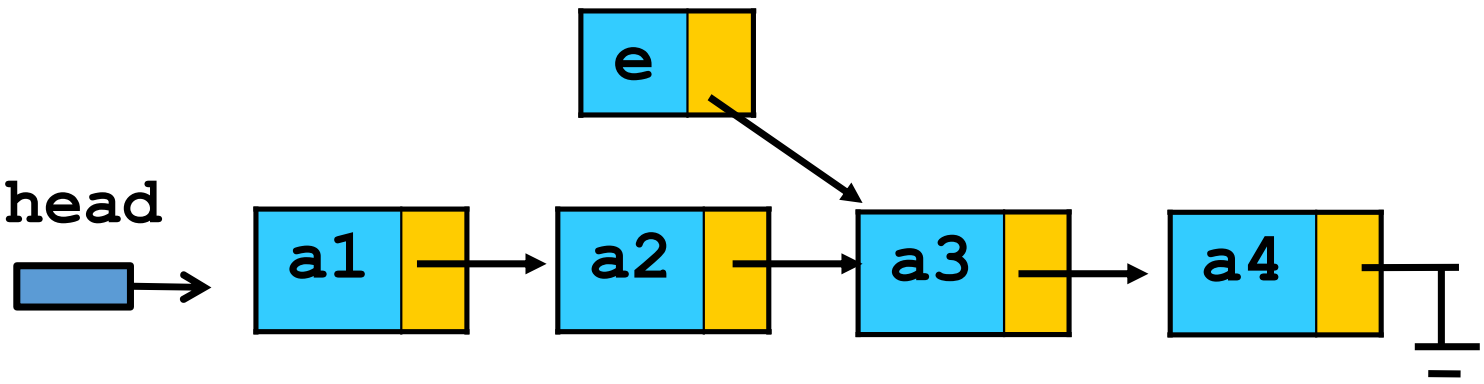


链式表LinkedList

- 第一个结点与其他结点不一样，前面没有“前驱结点”

  各种操作（如插入）需要区分对待是否是第一个结点还是其他结点。

insert(3,e)

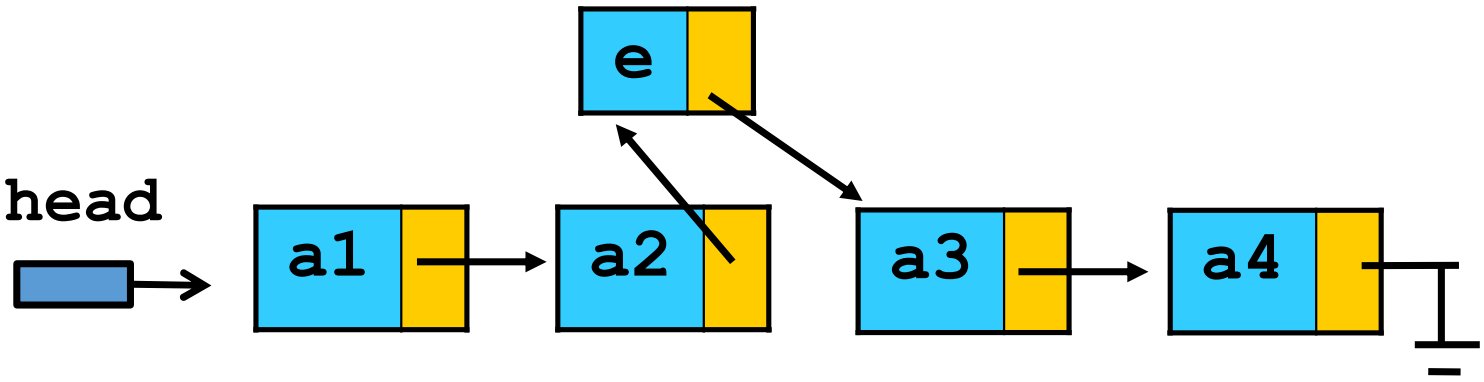


链式表LinkedList

- 第一个结点与其他结点不一样，前面没有“前驱结点”

  各种操作（如插入）需要区分对待是否是第一个结点还是其他结点。

insert(3,e)

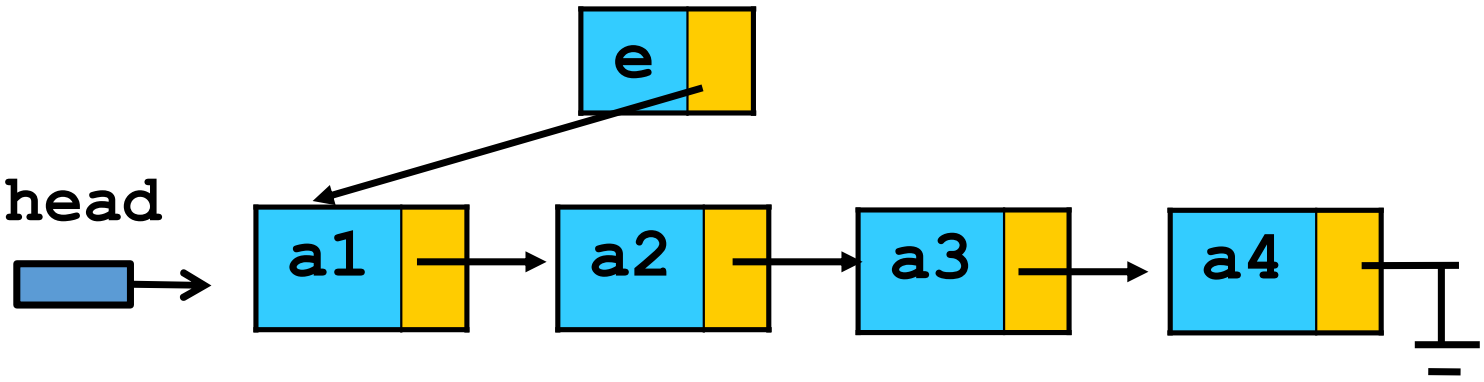


链式表LinkedList

- 第一个结点与其他结点不一样，前面没有“前驱结点”

  各种操作（如插入）需要区分对待是否是第一个结点还是其他结点。

insert(1,e)



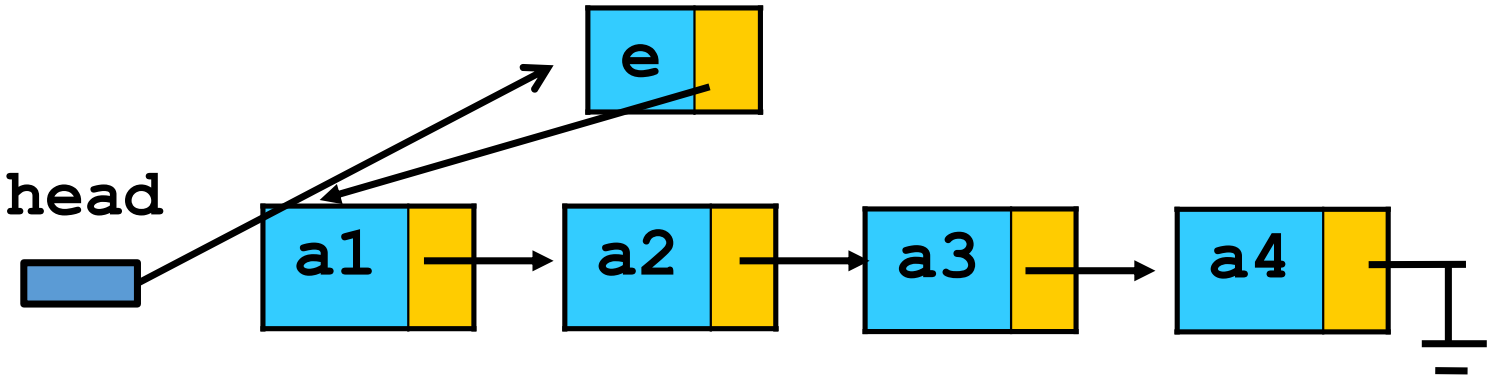
链式表LinkedList：结点

- 第一个结点与其他结点不一样，前面没有“前驱结点”

  各种操作（如插入）需要区分对待是否是第一个结点还是其他结点。

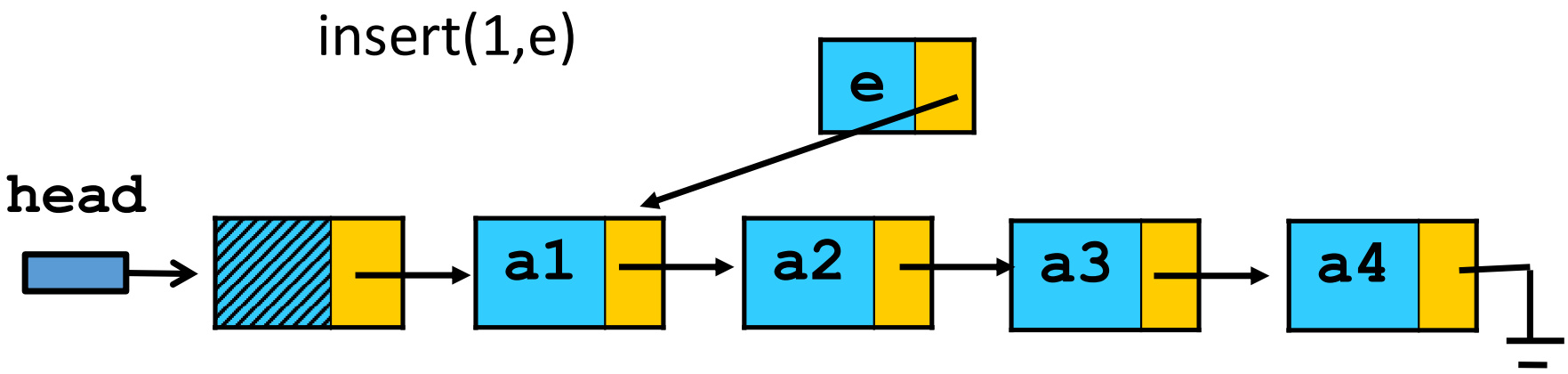
insert(1,e)

此时要修改L而不是前驱结点的next指针



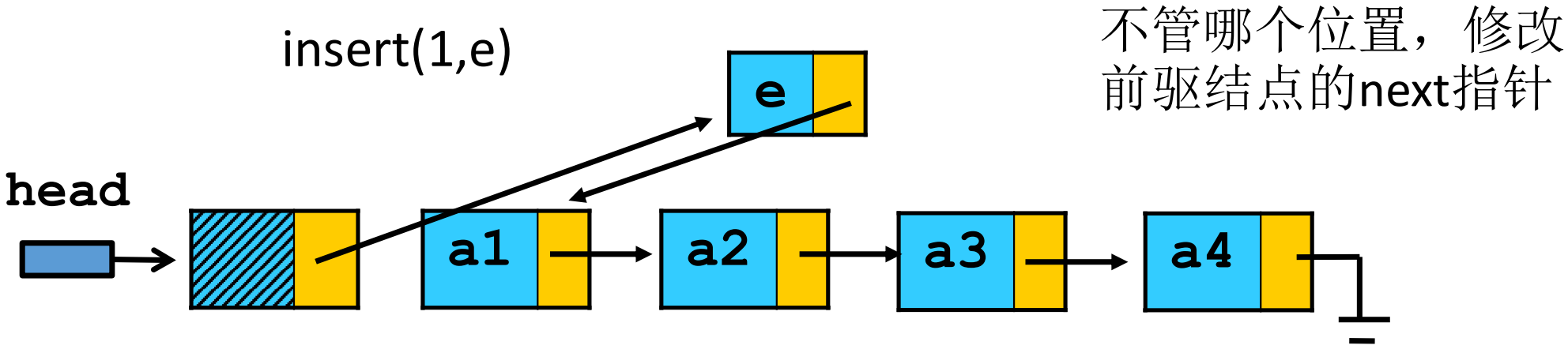
链式表LinkedList

- 第一个结点与其他结点不一样，前面没有“前驱结点”
- 算法（如插入）需要区分对待是否是第一个结点还是其他结点
- **解决方法**：在最前面始终保持一个无实际数据的“头结点”



链式表LinkedList

- 第一个结点与其他结点不一样，前面没有“前驱结点”
- 算法（如插入）需要区分对待是否是第一个结点还是其他结点
- **解决方法**：在最前面始终保持一个无实际数据的“头结点”

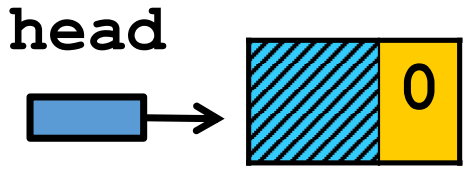


LkList<T>:: LkList( )

- L 指向一个新分配的“头结点“，头结点的next设置为0。表示是一个空表。

```
LkList<T>::LkList(){
   head = new LNode();
   if(!head) throw ”No Memory”;
   head->next = 0;

}
```

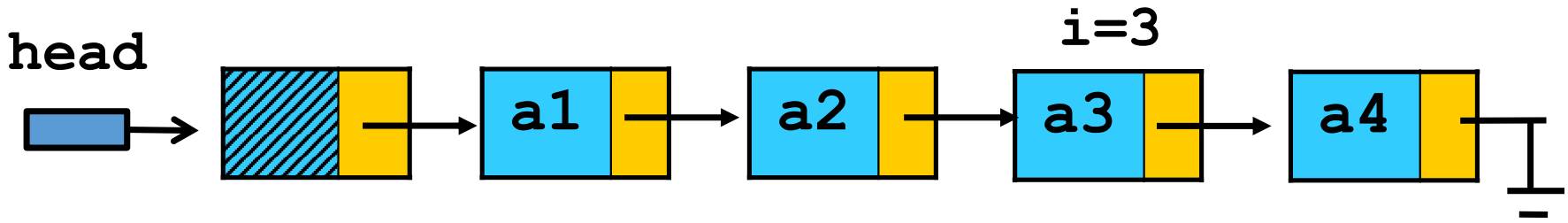
```
int main(){
    LkList <char> L;
    …
}
```



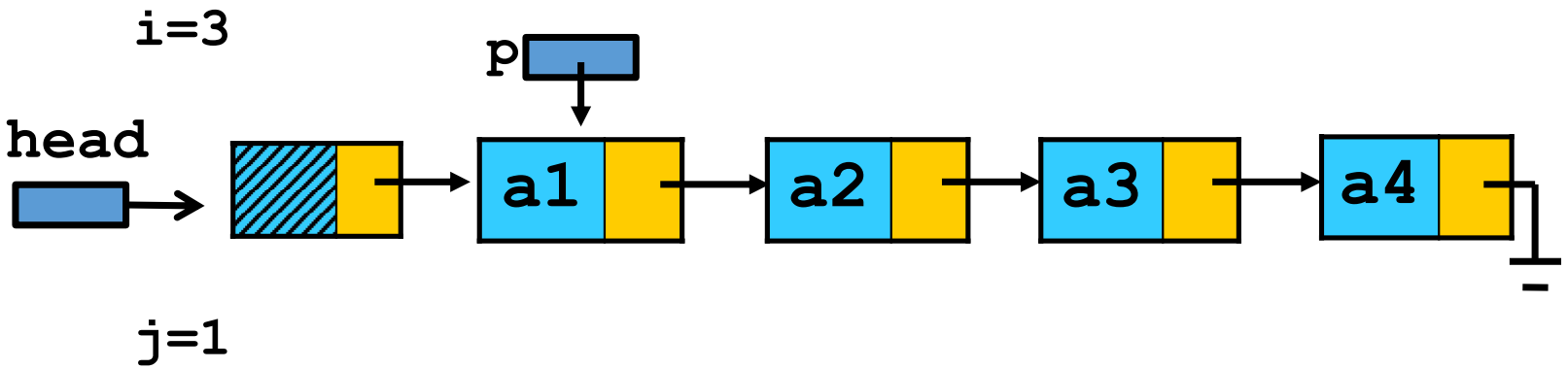
bool LkList<T>::get(int i, T &e)

- 访问线性表的第i个元素，要从表头起沿着指针一个一个元素的查找

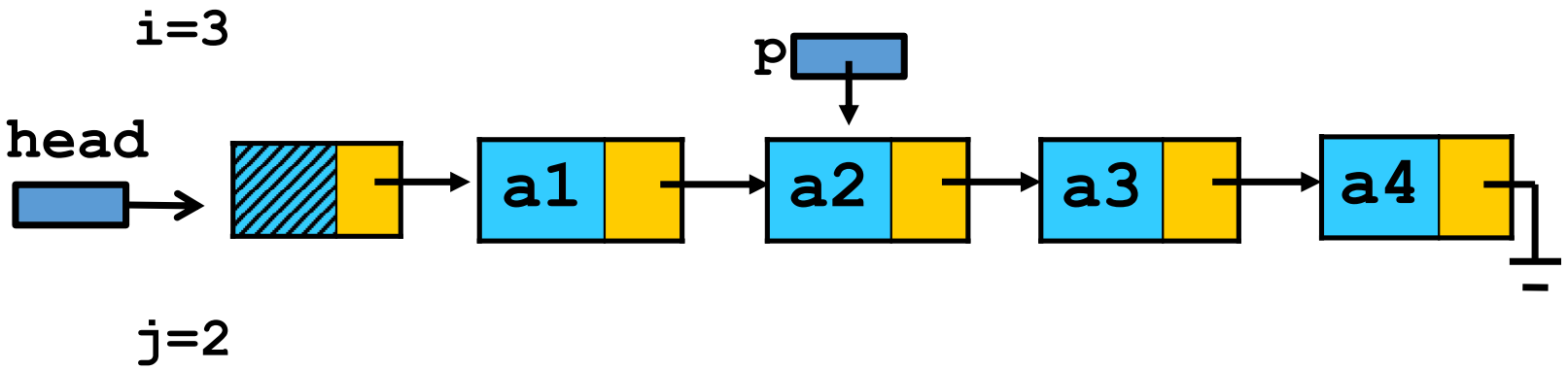
- 显然，访问第i个节点所需时间由i决定, 所以存取操作复杂度 = O(n)



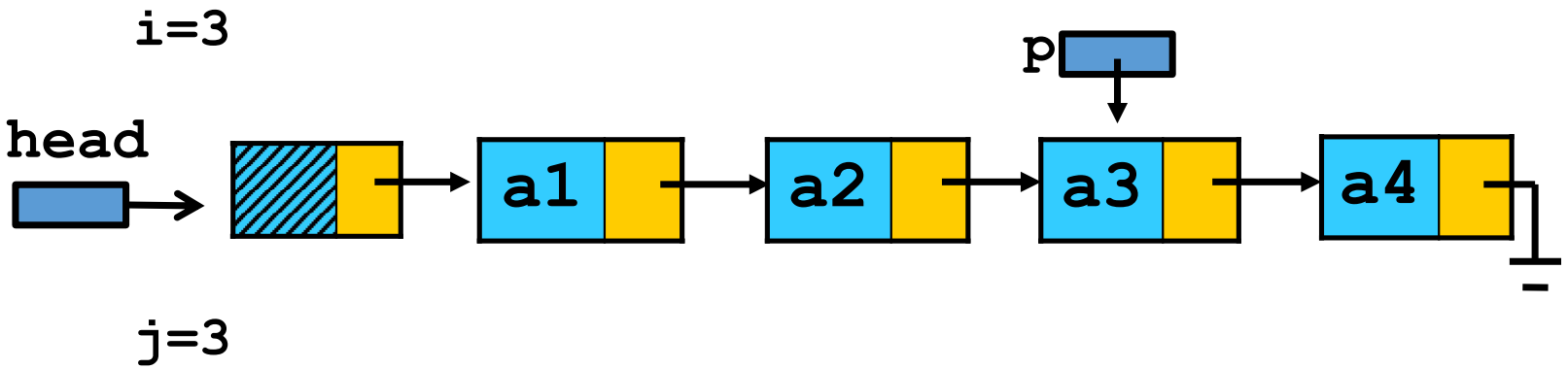
```
LNode *p = head->next;  int j = 1;
// 循环直到p为空或到了第i个节点
while(p && j < i) {
    p = p->next;
    ++ j;
}
if(!p || j > i)    // 第i个节点不存在
    return false;
```



```
LNode *p = head->next;  int j = 1;
// 循环直到p为空或到了第i个节点
while(p && j < i) {
    p = p->next;
    ++ j;
}
if(!p || j > i)    // 第i个节点不存在
    return false;
```

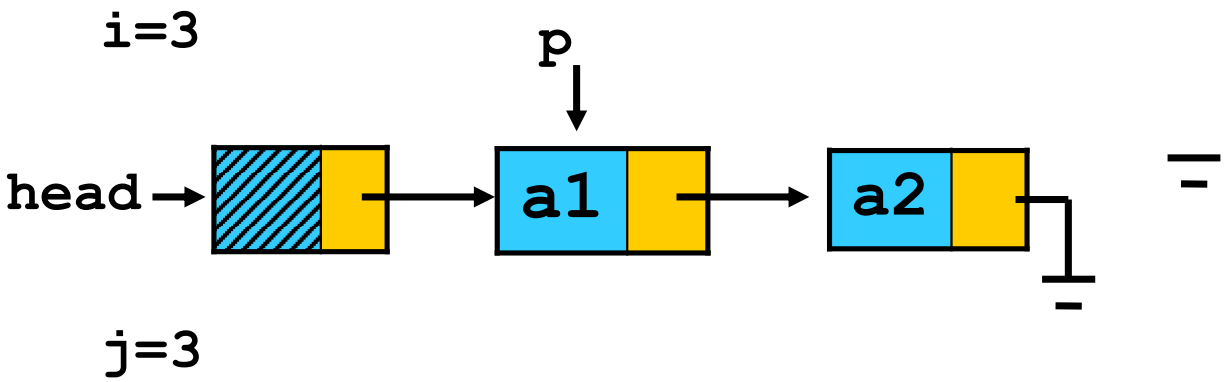


```
LNode *p = head->next;  int j = 1;
// 循环直到p为空或到了第i个节点
while(p && j < i) {
    p = p->next;
    ++ j;
}
if(!p || j > i)    // 第i个节点不存在
    return false;
```



```
LNode *p = head->next; int j = 1;
// 循环直到p为空或到了第i个节点
while(p && j < i) {
    p = p->next;
    ++ j;
}
if(!p || j > i)    // 第i个节点不存在
    return false;
```

p已经走到了尽头，
却还没找到第i个节点，
说明第i个节点不存在



bool LkList<T>::get(int i, T &e)

```
bool LkList<T>::get(int i, T &e){
   LNode *p = head->next; int j = 1;
   // 循环直到p为空或到了第i个节点
   while(p && j < i) {
      p = p->next;
      ++ j;
   }
   if(!p || j > i)    // 第i个节点不存在
      return false;
   e = p->data;       // copy数据到e中
   return true;
}
```

get的时间复杂度

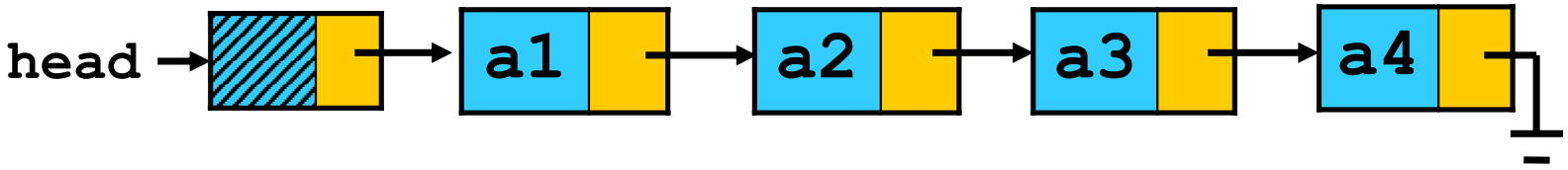
第1个元素，1次“指针赋值”；

第2个元素，2次“指针赋值”；

…

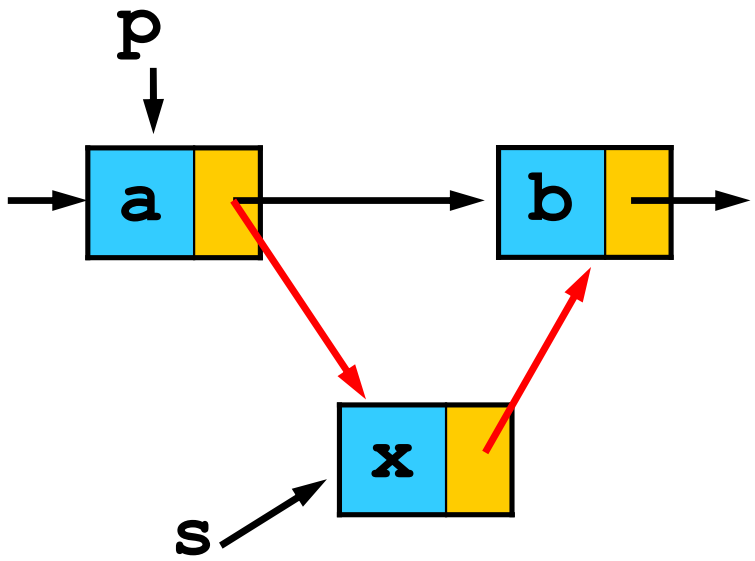
第n个元素，n次“指针赋值”；

- 时间复杂度为O(n)



bool LkList<T>::insert(int i, T e)

- 定位到序号为i的结点的“前驱结点”。即i-1号结点。

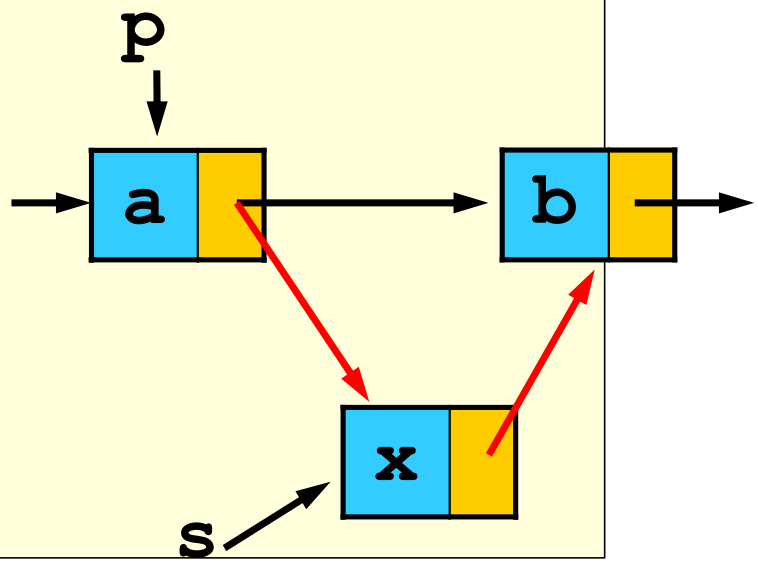


bool LkList<T>::insert(int i, T e)

- 定位到序号为i的结点的“前驱结点”。即i-1号结点。

```
bool LkList<T>::insert( int i, T e) {
    LNode* p = head; int j = 0;

    // 寻找第i-1个节点
    while(p && j < i-1){
        p = p->next;
        ++ j;
    }

    // 若第i-1个节点不存在
    if(!p || j > i-1)
        return false;
```

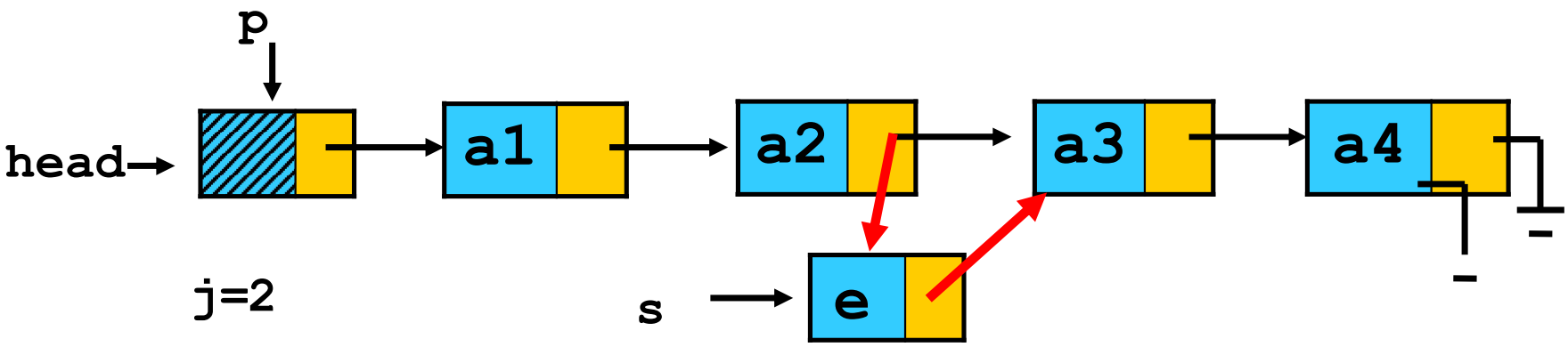
不就是get()中的代码么？



```
// 生成一个新节点，并链接到L中
s =  new LNode();
s->data = e;
s->next = p->next;
p->next = s;
return true;
}
```

注意：这两条语句的顺序不能颠倒

i=3，即在第3个节点前面插入一个新的节点

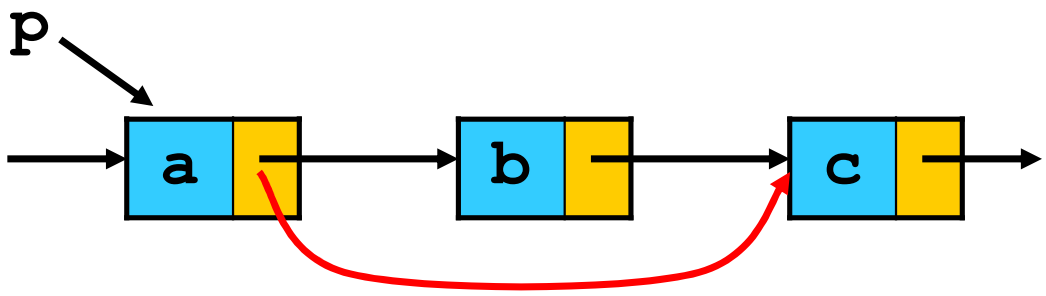


LkList<T>:: insert的时间复杂度

- 寻找插入位置的时间复杂度为O(n).
- 修改指针的时间复杂度为O(1).
- 因此，总的时间复杂度为O(n).

bool LkList<T>::remove (int i)

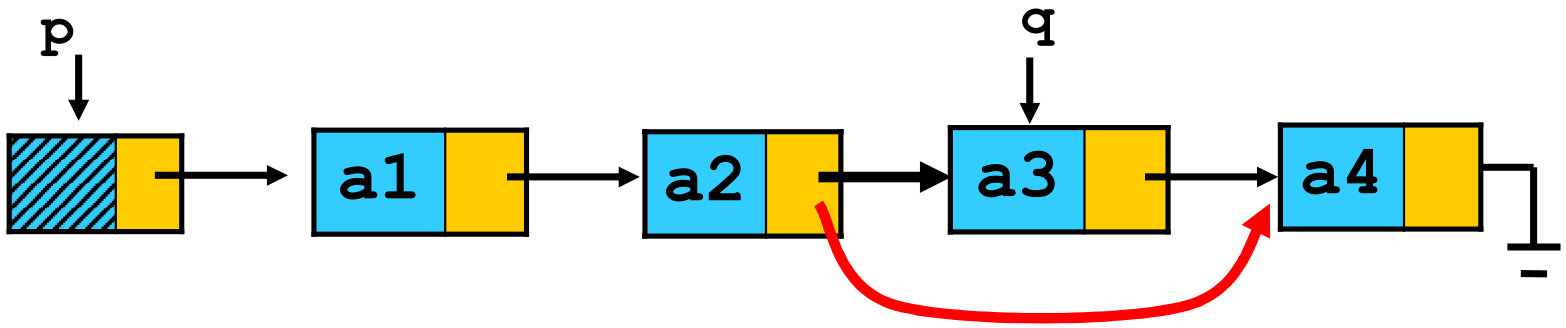
- 过程和insert类似：寻找删除结点的前驱结点，修改相应指针。需要释放掉被删除结点占用的内存块。



```
bool LkList<T>::remove( int i){
    LNode* p = head;  int j = 0;

    // 让p指向第i-1个节点
    while(p && j < i-1){
        p = p->next;
        ++ j;
    }

    // 若第i个节点不存在
    if(!p ||!(p->next) || j > i-1)
        return false;
```

```
    LNode* q = p->next; // q指向待删除节点
    p->next = q->next; // 使q脱离链表

    delete q;                  // 释放q的空间
    return true;
}
```

i=3，即删除第3个节点

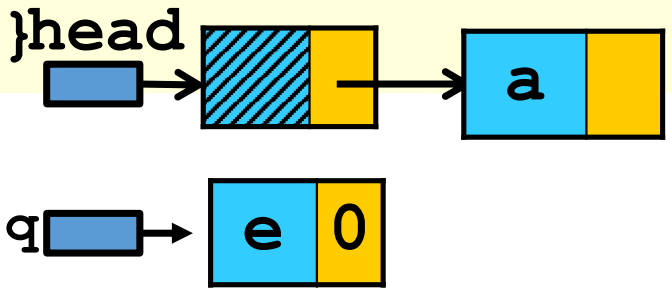


j=2

bool LkList<T>::insert_front (T e)

- 前插法(insert_front): 头结点后插入新结点

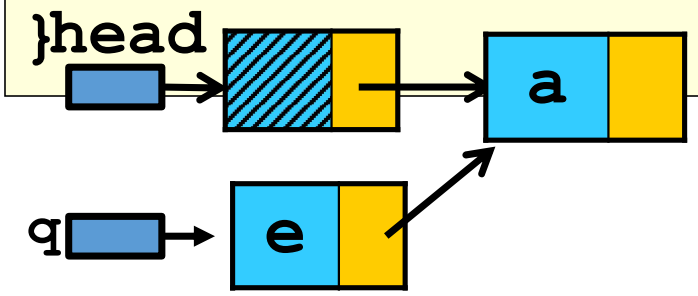
```
bool LkList<T>::insert_front( T e){
    LNode *q = new LNode();
    if(!q) return false;
    q->data = e;  q->next = 0;
```



bool LkList<T>::insert_front (T e)

- 前插法(insert_front): 头结点后插入新结点

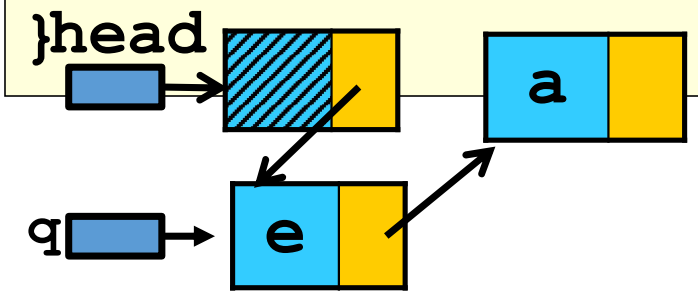
```
bool LkList<T>::insert_front( T e){
    LNode *q = new LNode();
    if(!q) return false;
    q->data = e;  q->next = 0;
    q->next = head->next; //新结点和头结点的next同指向
}
```



bool LkList<T>::insert_front (T e)

- 前插法(insert_front): 头结点后插入新结点

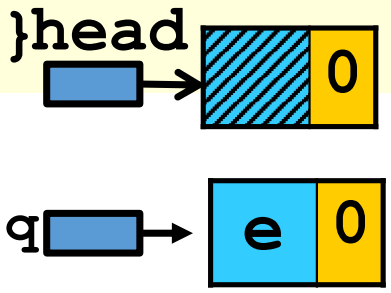
```
bool LkList<T>::insert_front( T e){
    LNode *q = new LNode();
    if(!q) return false;
    q->data = e;  q->next = 0;
    q->next = head->next; //新结点和头结点的next同指向
    head->next = q;  //新结点挂在头结点后面，成为首结点
    return true;
}
```



bool LkList<T>::insert_front (T e)

- 前插法(insert_front): 头结点后插入新结点

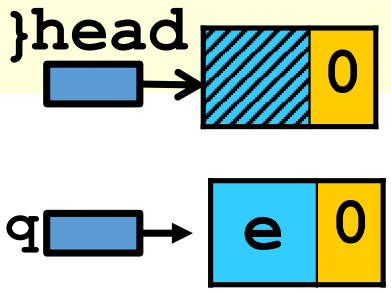
```
bool LkList<T>::insert_front( T e){
    LNode *q = new LNode();
    if(!q) return false;
    q->data = e;  q->next = 0;
```



bool LkList<T>::insert_front (T e)

- 前插法(insert_front): 头结点后插入新结点

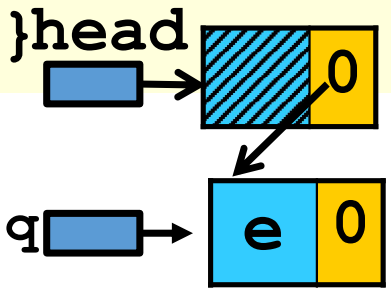
```
bool LkList<T>::insert_front( T e){
    LNode *q = new LNode();
    if(!q) return false;
    q->data = e;  q->next = 0;
    q->next = head->next; //新结点和头结点的next同指向
```



bool LkList<T>::insert_front (T e)

- 前插法(insert_front): 头结点后插入新结点

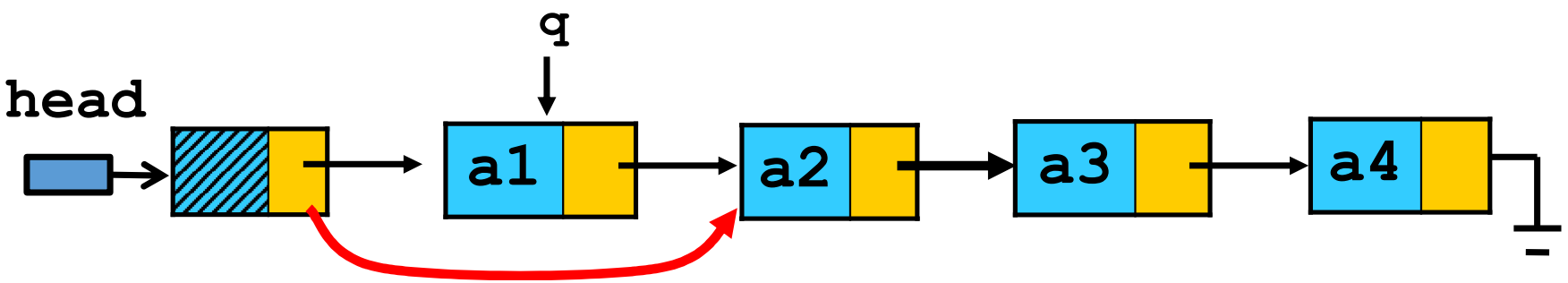
```
bool LkList<T>::insert_front( T e){
    LNode *q = new LNode();
    if(!q) return false;
    q->data = e;  q->next = 0;
    q->next = head->next; //新结点和头结点的next同指向
    head->next = q;  //新结点挂在头结点后面，成为首结点
    return true;
}
```



练习：

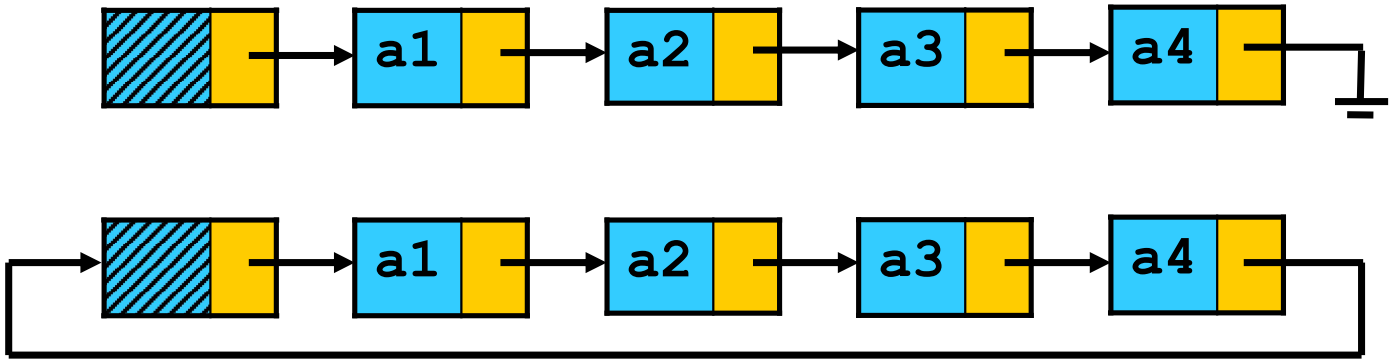
- 实现：remove_front, push_back, pop_back, ~LkList()等其他操作

```
q = head->next;
head->next = q->next;
delete q;
```



循环链表

- 非循环链表
  - 尾指针为空，浪费
- 循环链表
  - 尾指针指向表头

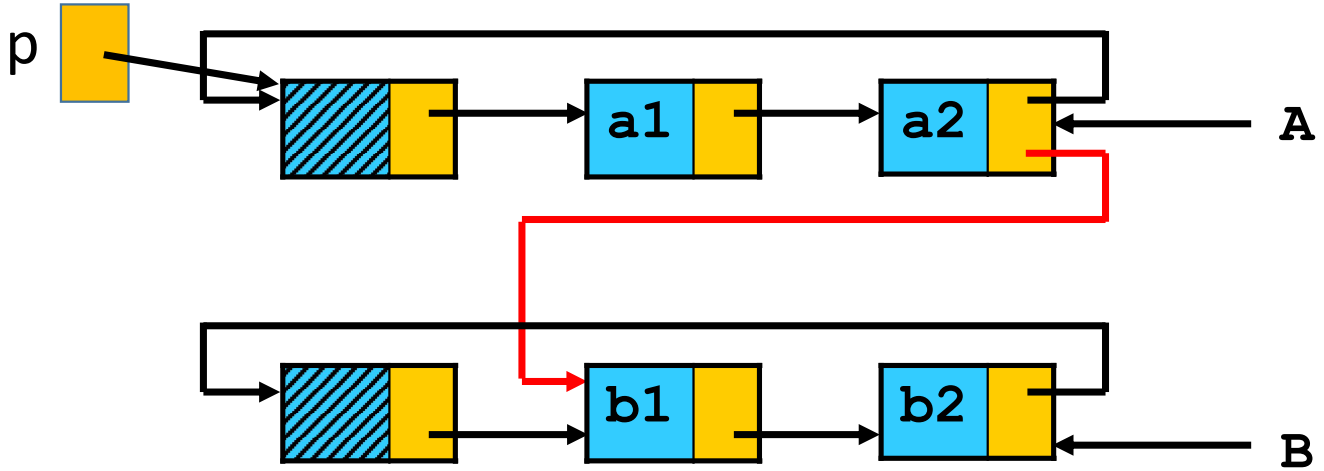


循环链表

p = A->next;

A->next = B->next->next

- 循环链表通常设尾指针
  - 要找头指针？尾指针指向的就是
  - 便于链表合并

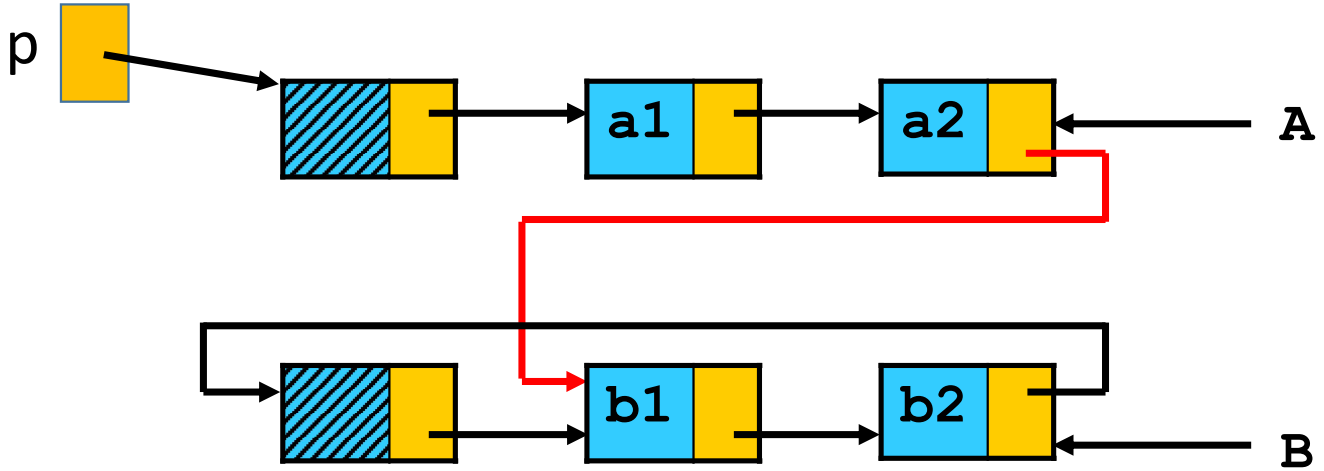


循环链表

p = A->next;

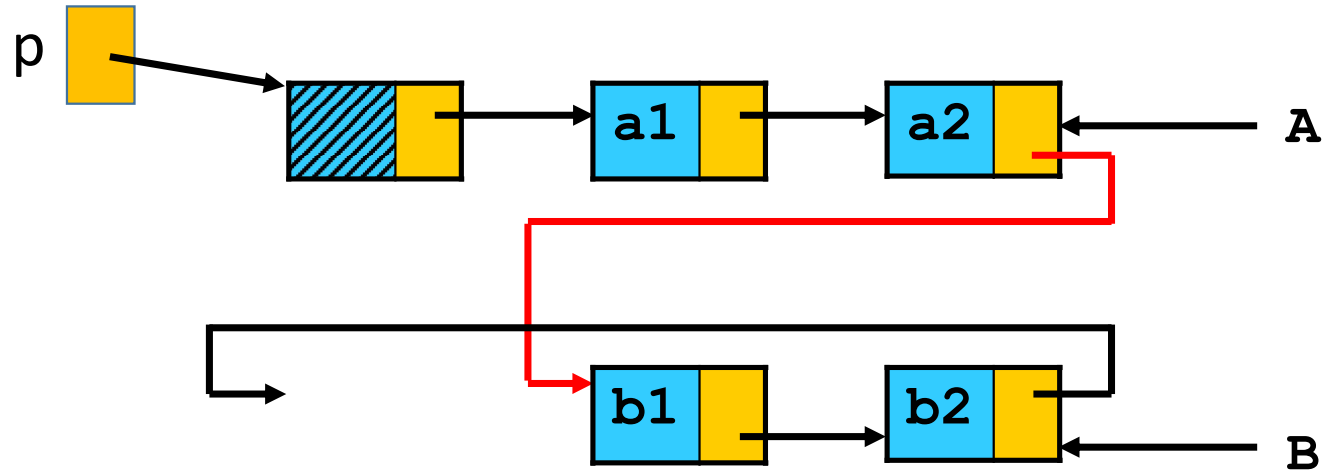
A->next = B->next->next

- 循环链表通常设尾指针
  - 要找头指针？尾指针指向的就是
  - 便于链表合并



循环链表

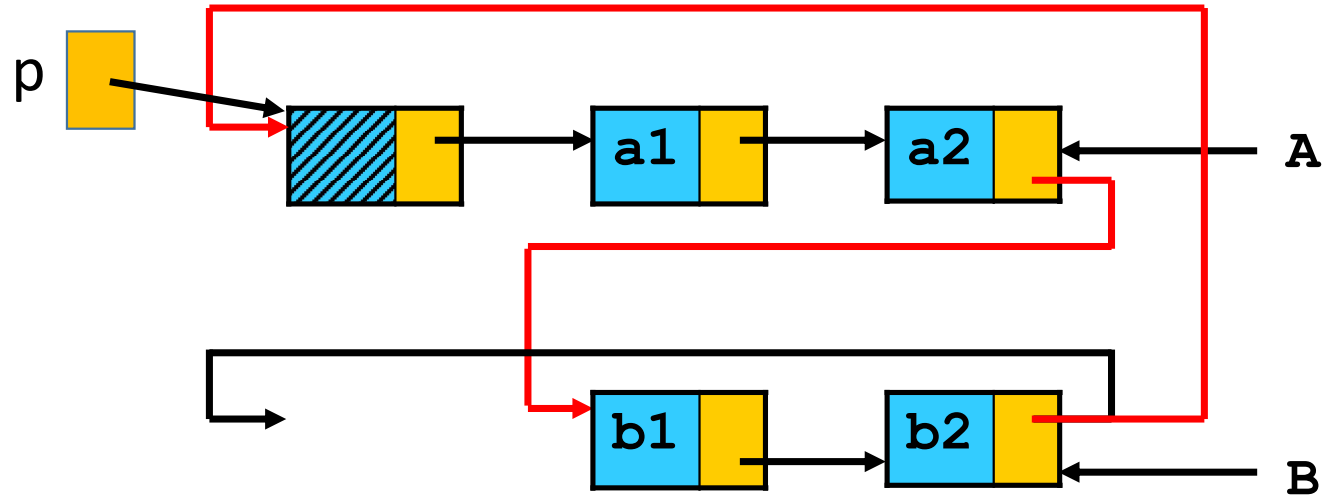
- 循环链表通常设尾指针
  - 要找头指针？尾指针指向的就是
  - 便于链表合并




```
p = A->next;
A->next = B->next->next
delete B->next;
```

循环链表

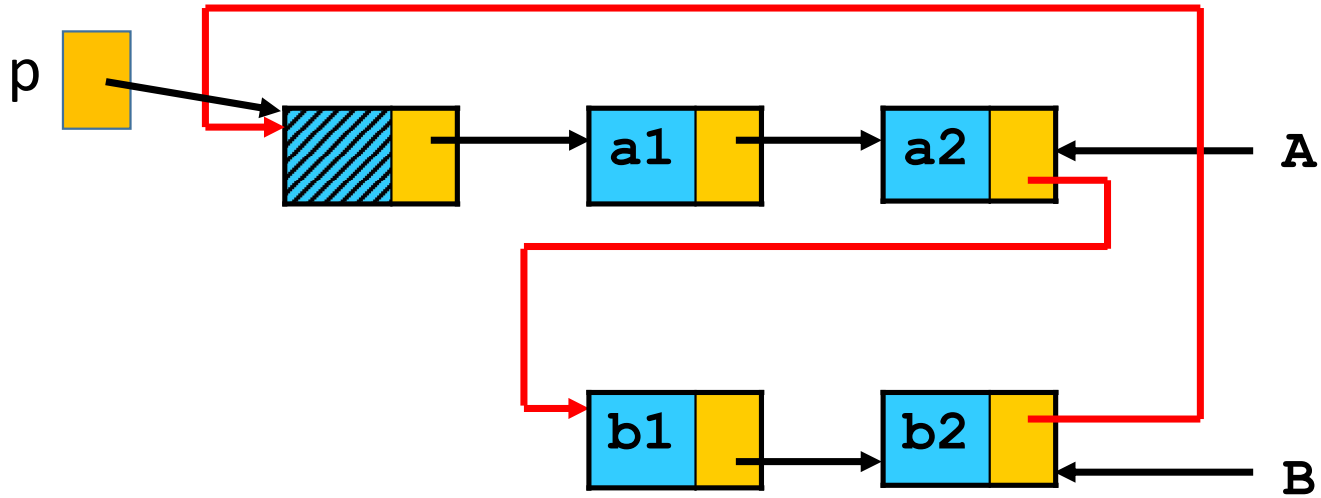
- 循环链表通常设尾指针
  - 要找头指针？尾指针指向的就是
  - 便于链表合并




```
p = A->next;
A->next = B->next->next
delete B->next;
B->next = p;
```

循环链表

- 循环链表通常设尾指针
  - 要找头指针？尾指针指向的就是
  - 便于链表合并



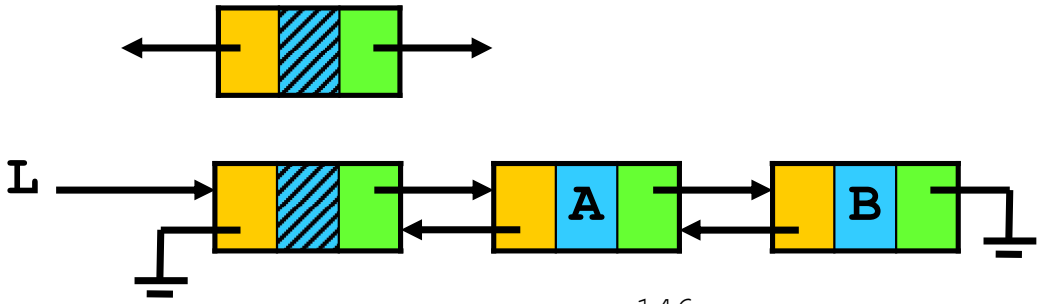

```
p = A->next;
A->next = B->next->next
delete B->next;
B->next = p;
A = B;
```

双向链表

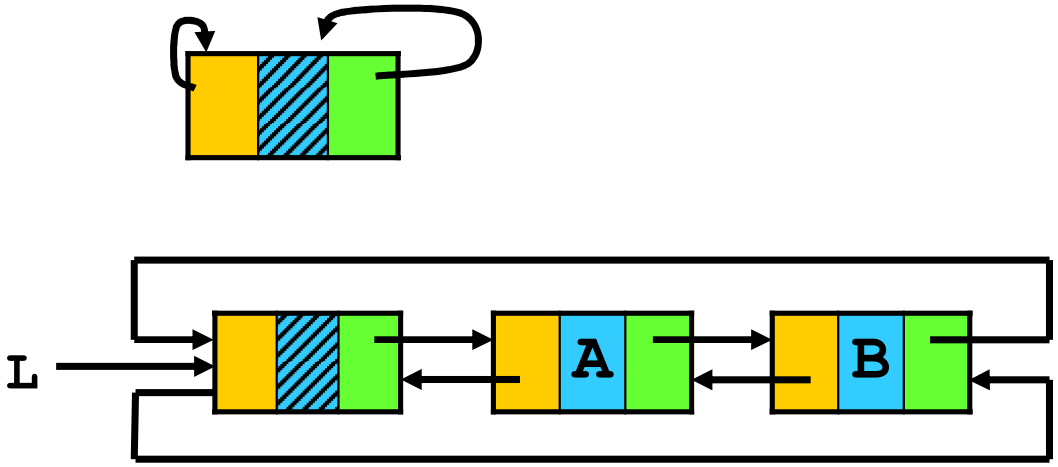
- 单向链表
  - 只知道后继节点，不知前趋节点
  - NextElem操作复杂度为O(1)
  - PriorElem操作复杂度为O(n)
    - 必须从头开始查找
- 双向链表
  - 增加一个前趋指针

存储结构

```
template<class T>
struct DulNode{
    T   data;
    DulNode *prior;
    DulNode *next;
};
```

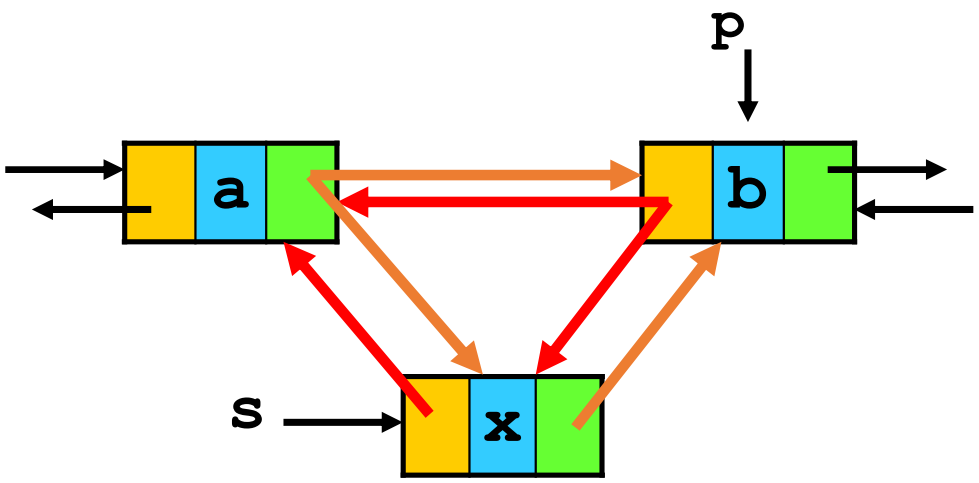


循环双向链表



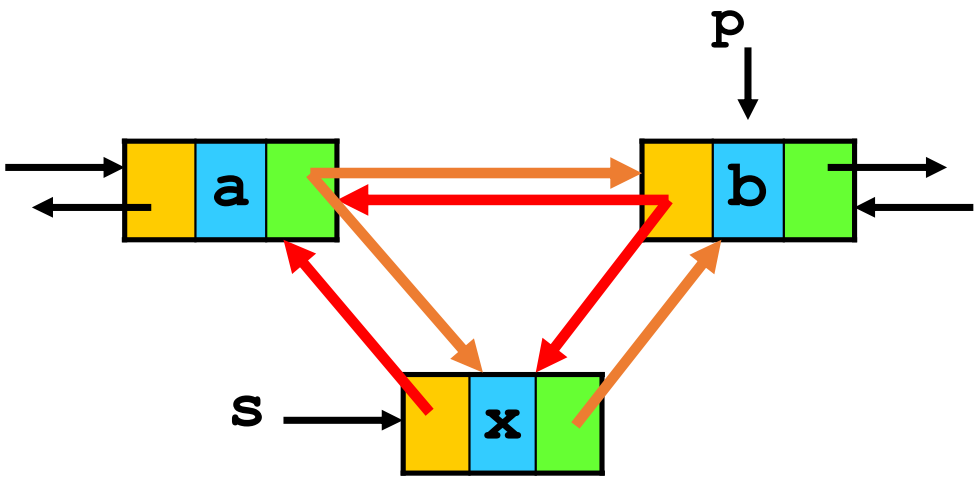
双向链表: 插入操作

- 在第i个节点p之前，插入节点s



双向链表: 插入操作

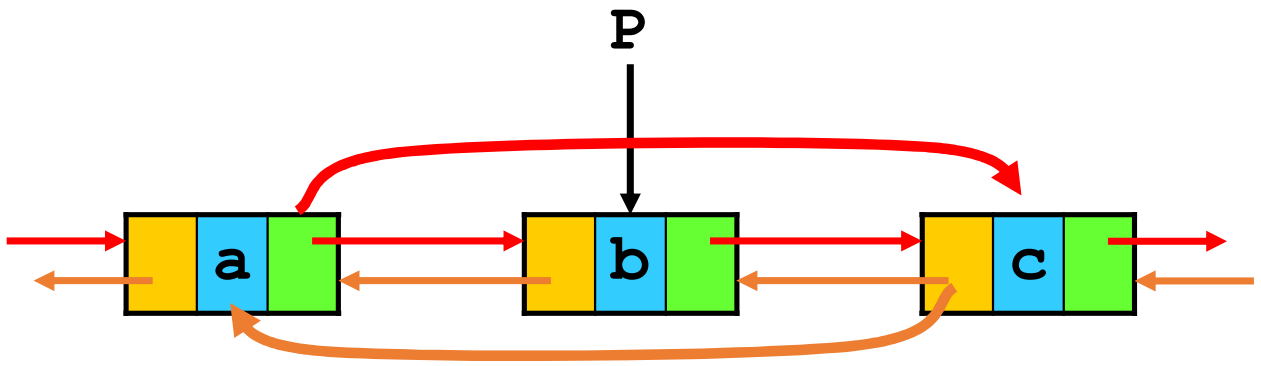
```
s->prior = p->prior;
p->prior->next = s;
s->next = p;
p->prior = s;
```



双向链表: 删除操作

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

绕过p

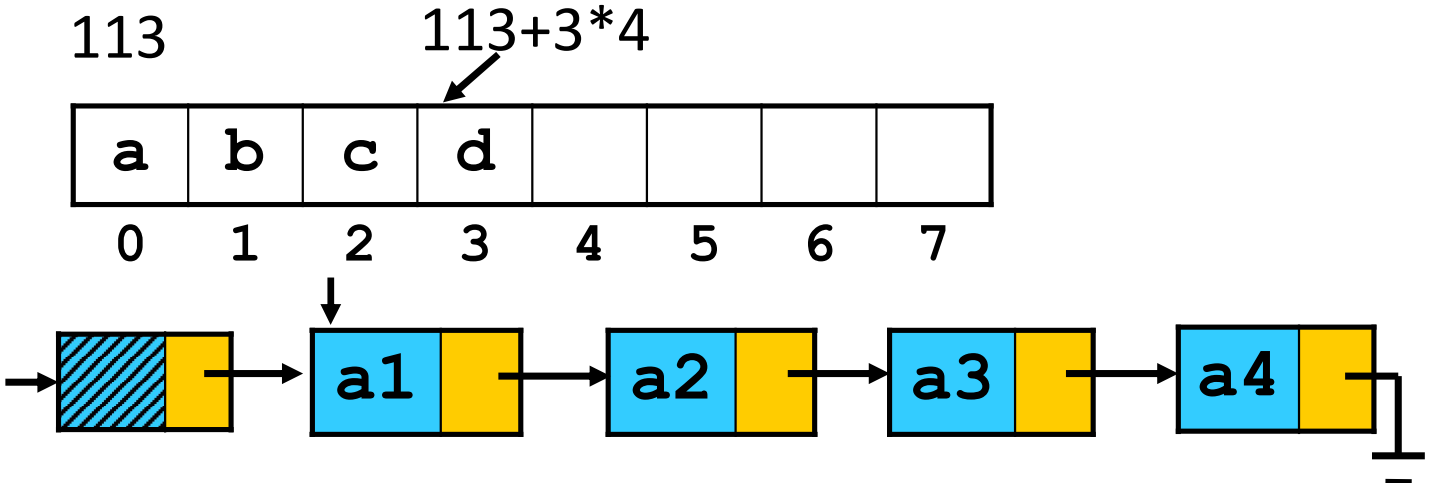


Array VS Linked List

- 序号读取元素

顺序表	链表
O(1)	O(n)

A[i]= *(A+i)



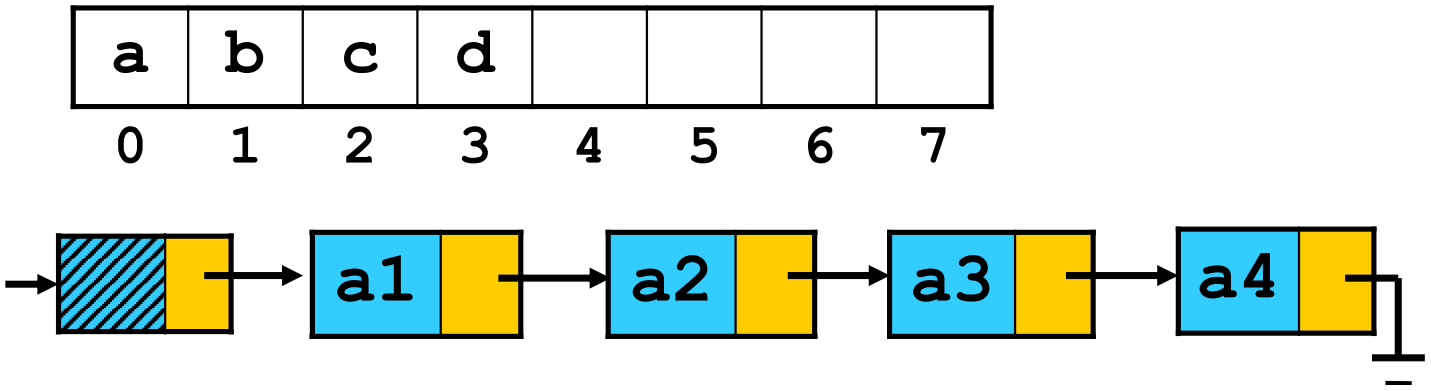
Array VS Linked List

- 内存使用

顺序表	链表
分配预留空间	按需分配
实际/容量	元素大小/结点大小
可能没有足够大空间	可利用很小的碎片空间

$$\frac{\text{元素个数}}{\text{空间容量}} = \frac{4}{8}$$

$$\frac{\text{元素大小}}{\text{元素大小+指针大小}}$$

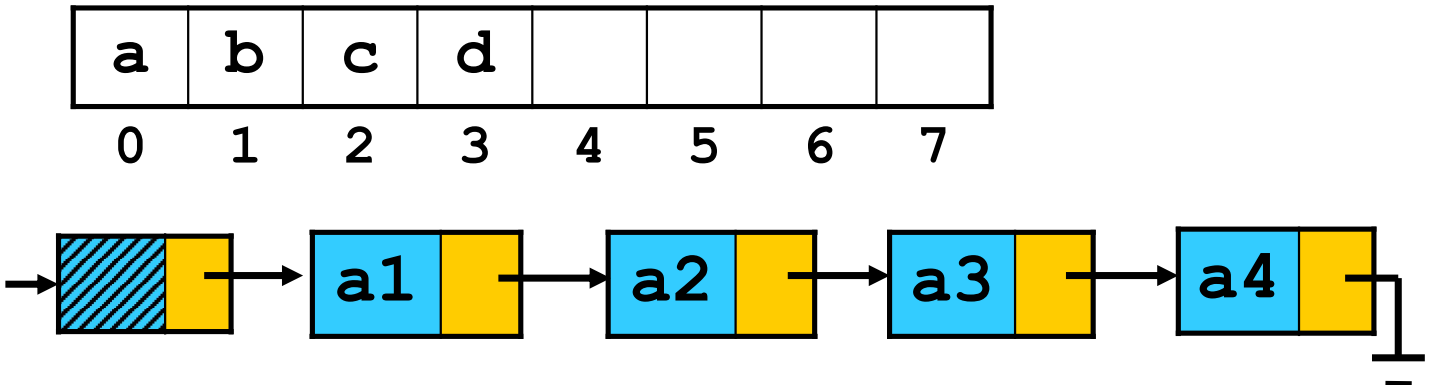


Array VS Linked List

- 插入元素

  insert_front(e)

顺序表	链表
O(n)	O(1)



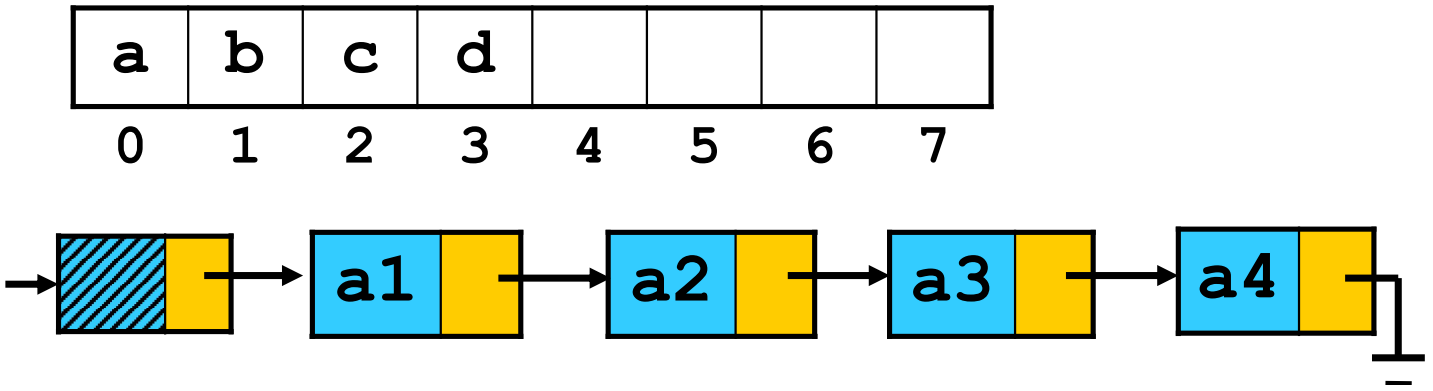
Array VS Linked List

- 插入元素

  push_back(e)

顺序表	链表
O(1)	O(n)

如果重新分配
内存，也是O(n)



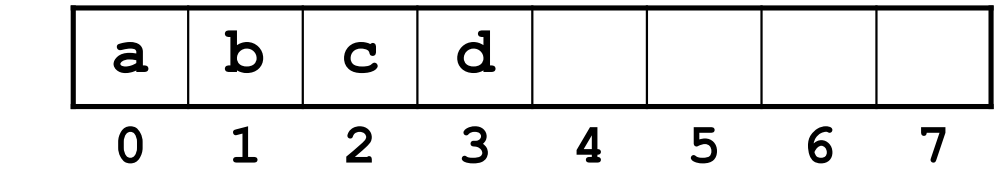
Array VS Linked List

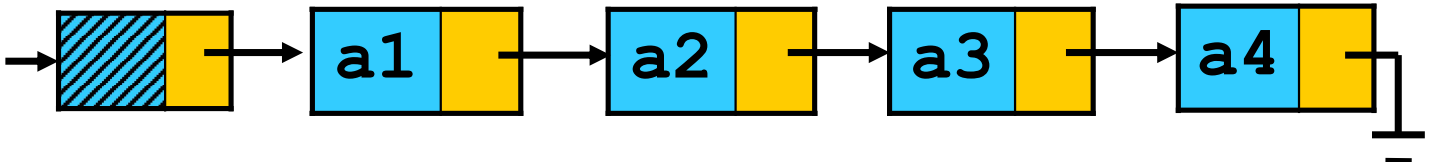
- 插入元素

  insert(pos,e)

顺序表	链表
O(n)	O(n)

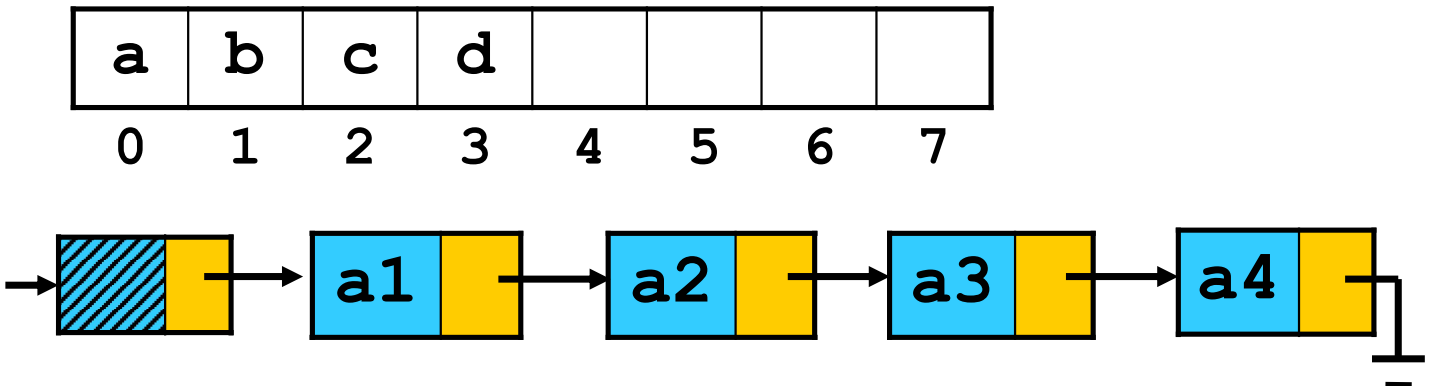
但修改指针比移动大的数据元素要快得多！





Array VS Linked List

- 删除delete/remove情况类似



练习：

- 实现线性表（顺序表或链式表）的逆置

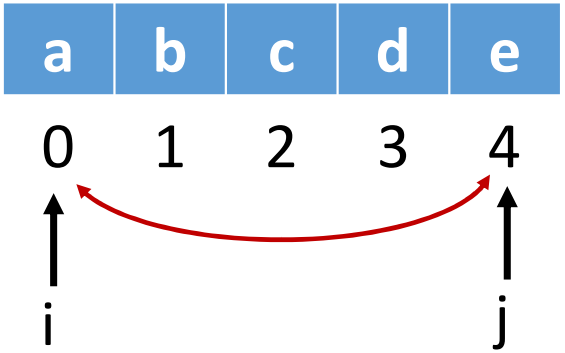
  顺序表： 夹逼法

  链式表： 链表的前插法

- 删除排序的线性表（顺序表或链式表）的重复元素，要求不分配额外数据元素存储空间

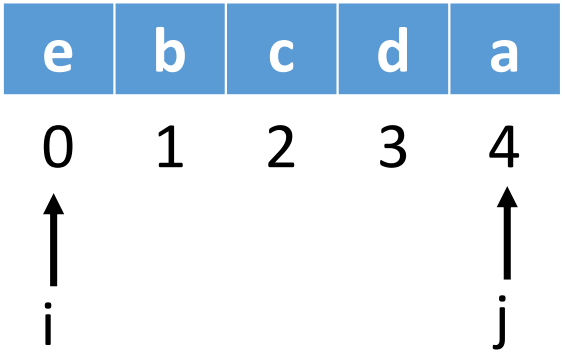
  输入： 1->2->3->3->4->4->5，输出： 1->2->4->5.
  输入： 1->1->1->2->3，输出： return 1-> 2->3.

顺序表的逆置(反转)



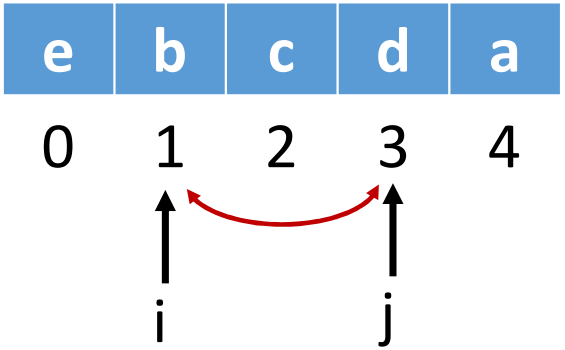

```
void converse(T arr[], int n ){
    for(int i=0,j = n-1 ;  ){
        T t =arr[i];
        arr[[i] = arr[j];
        arr[j] = t;
    }
}
```

顺序表的逆置(反转)



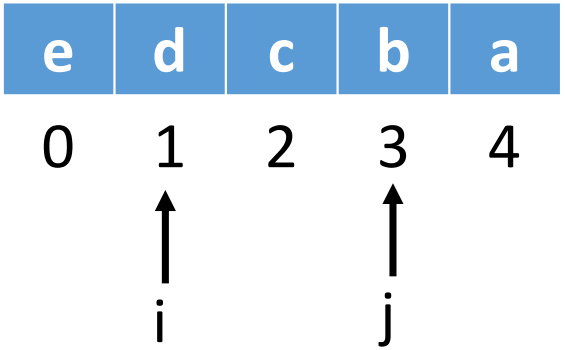

```
void converse(T arr[], int n ){
    for(int i=0,j = n-1 ;i<j; i++,j-- ){
        T t =arr[i];
        arr[[i] = arr[j];
        arr[j] = t;
    }
}
```

顺序表的逆置(反转)



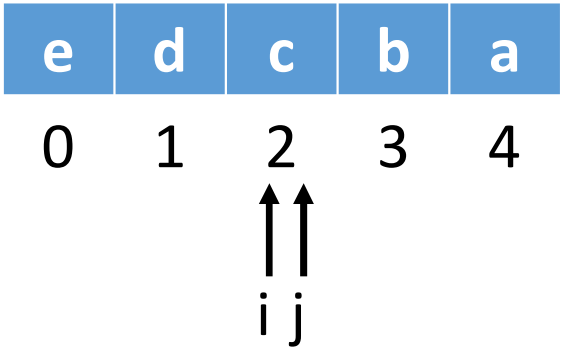

```
void converse(T arr[], int n ){
    for(int i=0,j = n-1 ;i<j; i++,j-- ){
        T t =arr[i];
        arr[[i] = arr[j];
        arr[j] = t;
    }
}
```

顺序表的逆置(反转)



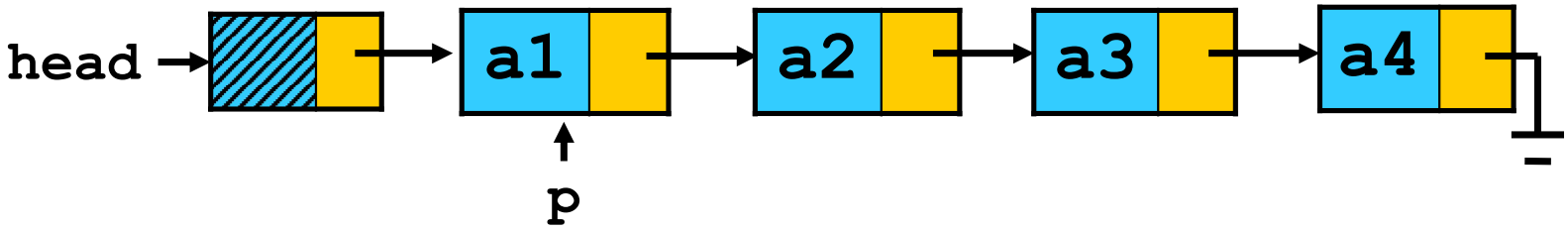

```
void converse(T arr[], int n ){
    for(int i=0,j = n-1 ;i<j; i++,j-- ){
        T t =arr[i];
        arr[[i] = arr[j];
        arr[j] = t;
    }
}
```

顺序表的逆置(反转)



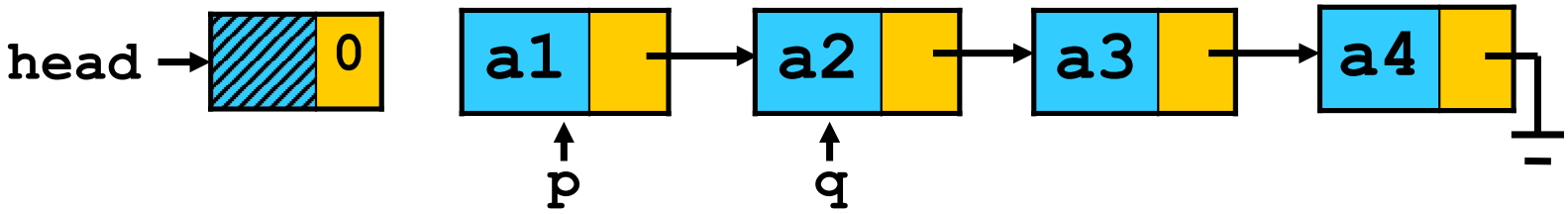

```
void converse(T arr[], int n ){
    for(int i=0,j = n-1 ;i<j; i++,j-- ){
        T t =arr[i];
        arr[[i] = arr[j];
        arr[j] = t;
    }
}
```

链表的逆置(反转)



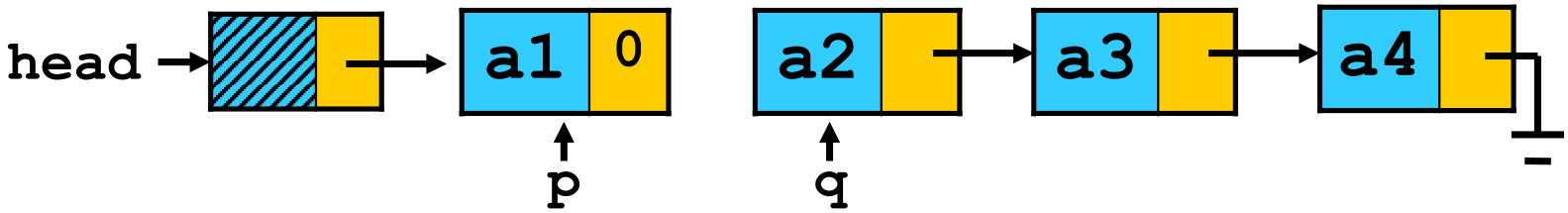

```
p = head->next;
head->next = 0;
```

链表的逆置(反转)



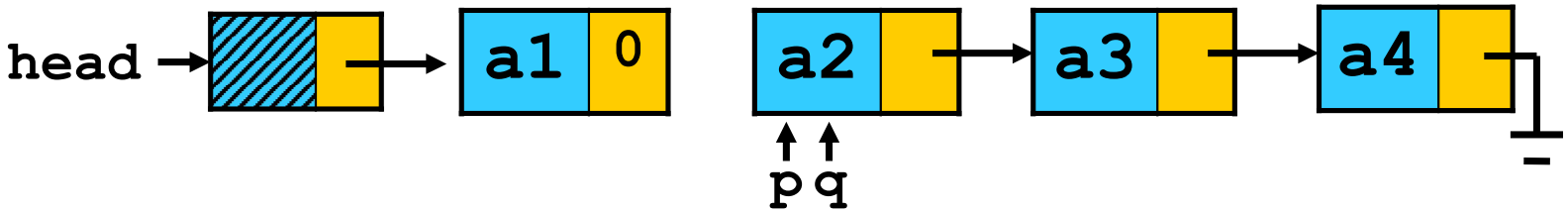

```
p = head->next;
head->next = 0;
while(p){ //前插法到头结点后面
    q = p->next; //保存下一个结点地址
    p->next = head->next; //指向新链表首结点
    head->next = p;

}
```

链表的逆置(反转)




```
p = head->next;
head->next = 0;
while(p){ //前插法到头结点后面
    q = p->next; //保存下一个结点地址
    p->next = head->next; //指向新链表首结点
    head->next = p;

}
```

链表的逆置(反转)




```
p = head->next;
head->next = 0;
while(p){ //前插法到头结点后面
    q = p->next; //保存下一个结点地址
    p->next = head->next; //指向新链表首结点
    head->next = p;
    p = q;
}
```

删除有序表中的重复元素

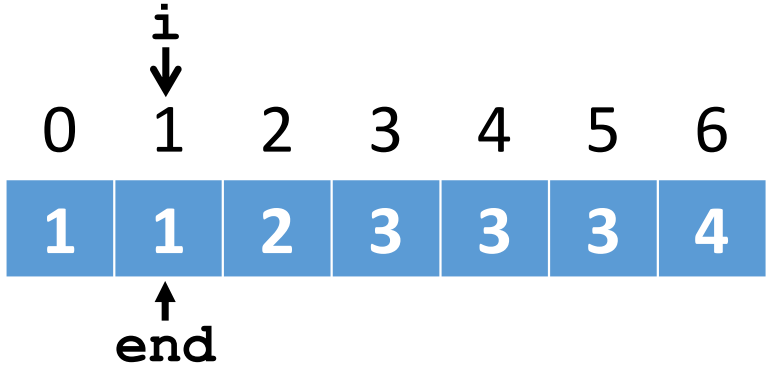
输入

```
[1,1,2,3,3,3,4]
或
[2,5,5,7,7,9,9]
```

输出

```
[1,2,3,4]
或
[2,5,7,9]
```

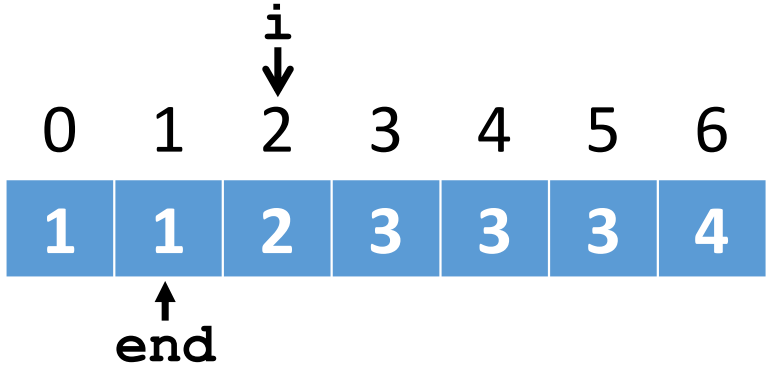
删除有序表中的重复元素



因nums[i]==a，所以i继续后移

a = nums[0]; *//初始化第一个不重复数字*
end =1; *//指向新表的最后位置*

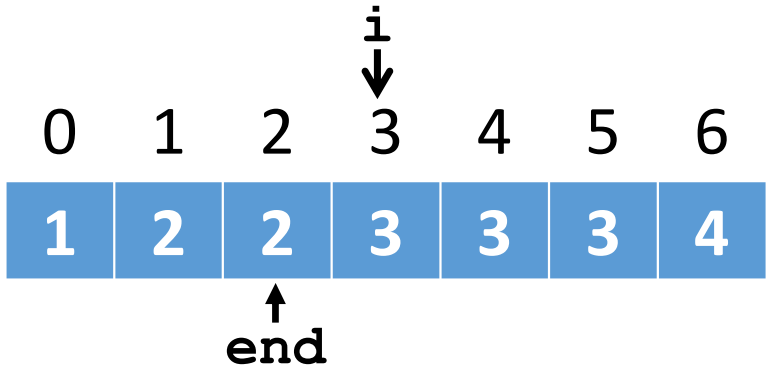
删除有序表中的重复元素



发现nums[i]!=a，插入新元素

```
nums[end] = nums[i];
a = nums[i];
end++; i++;
```

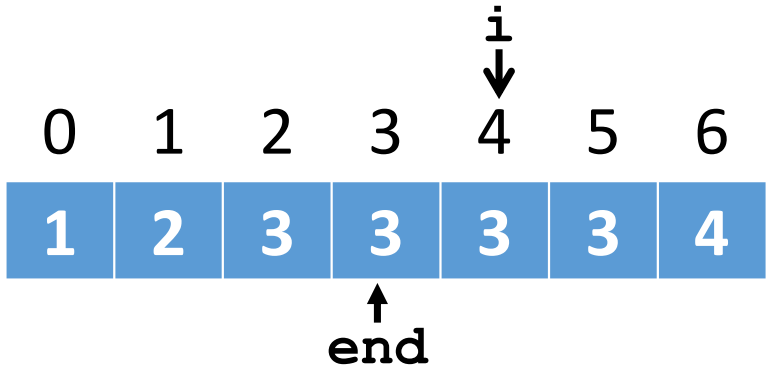
删除有序表中的重复元素



发现nums[i]!=a，插入新元素

```
nums[end] = nums[i];
a = nums[i];
end++; i++;
```

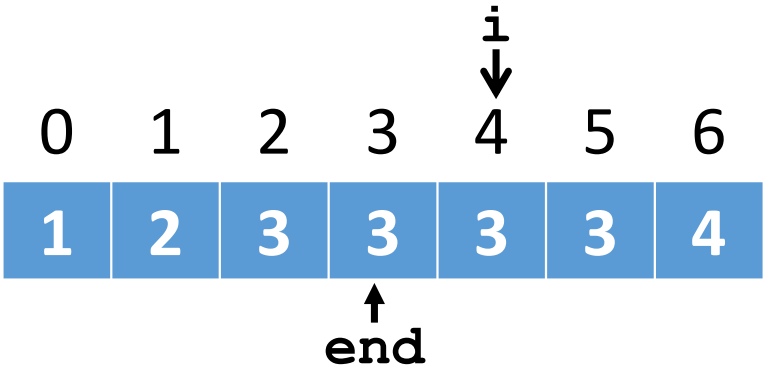
删除有序表中的重复元素



发现nums[i]!=a，插入新元素

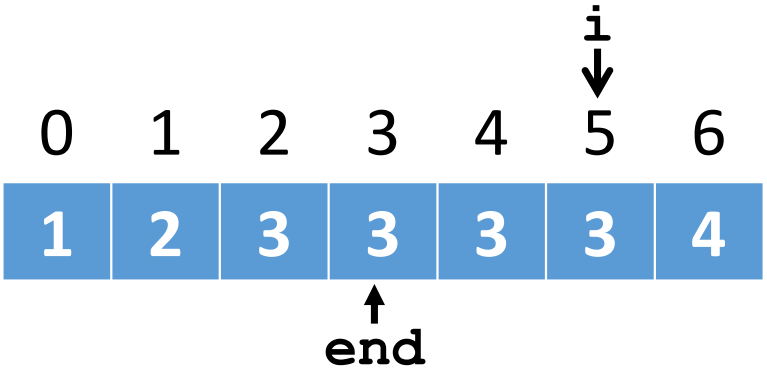
```
nums[end] = nums[i];
a = nums[i];
end++; i++;
```

删除有序表中的重复元素



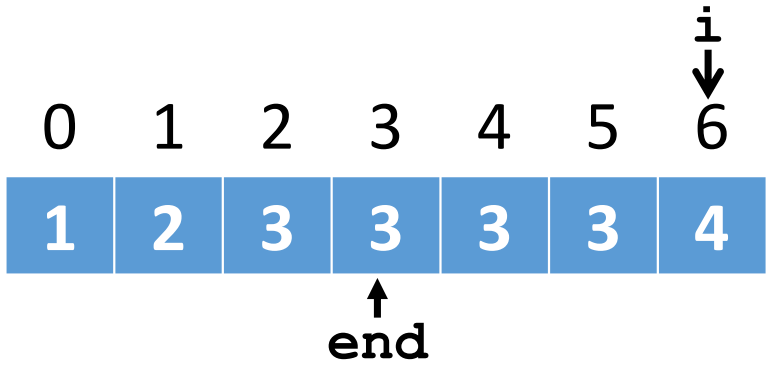
因nums[i]==a，所以i继续后移

删除有序表中的重复元素



因nums[i]==a，所以i继续后移

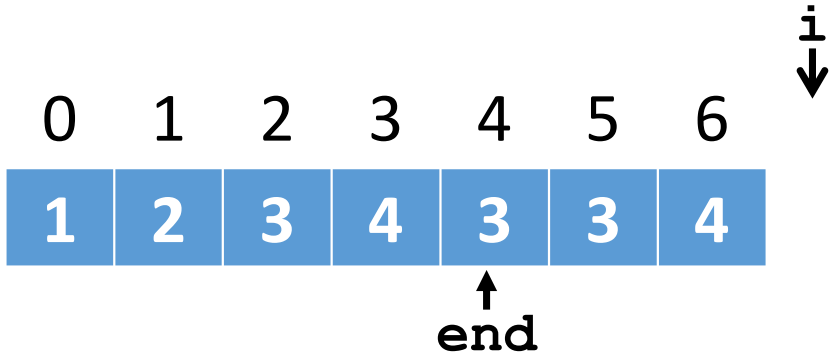
删除有序表中的重复元素



发现nums[i]!=a，插入新元素

```
nums[end] = nums[i];
a = nums[i];
end++; i++;
```

删除有序表中的重复元素



发现nums[i]!=a，插入新元素

```
nums[end] = nums[i];
a = nums[i];
end++; i++;
```

实验

- 实现顺序表并测试
- 实现单链表并测试
- 实现双向链表并测试

关注

https://a.hwdong.com

B站或微博：hw-dong

网易云课堂：hwdong

腾讯课堂：hwdong.ke.qq.com

QQ群： 101132160