




# Chapter 3

## List, Stack, and Queue

- 
- Introduce the concept of Abstract Data Types(ADTS).
  - show how to efficiently perform operations on lists.
  - Introduce the stack ADT and its use in implementing  
recursion.
  - Introduce the queue ADT and its use in operating  
systems and algorithm design.

## 3.1 Abstract Data Types(ADTS)

1. ADT-----a set of objects together with a set of operations.  
Abstract data types are mathematical abstractions;  
nowhere in an ADT's definition is there any mention of  
how the set of operations is implemented.

## 3.1 Abstract Data Types(ADTS)

### 2. data object---

a set of instances or values

for example:

Boolean={false,true}

Digit={0,1,2,3,4,5,6,7,8,9}

Letter={A,B,.....Z,a,b,.....z}

NaturalNumber={0,1,2,.....}

Integer = {0, +1, -1, +2, -2, +3, -3, ...}

String={a,b,.....,aa, ab, ac,.....}

# 3.1 Abstract Data Types<sub>(ADTS)</sub>

## 3. data structure

is a data object together with the relationships among the instances and among the individual elements that compose an instance

- $\text{Data\_Structure} = \{D, R\}$
- $D$  --- data object,
- $R$  --- a set of relationships of all the data members in  $D$ .

## 3.2 The List ADT

$L = (e_1, e_2, e_3, \dots, e_n)$

list size is  $n$

if  $n=0$ : empty list

if  $n(\text{finite}) > 0$ :

$e_1$  is the first element

$e_n$  is the last element

$e_i$  precedes  $e_{i+1}$

## 3.2 The List ADT

Example:

Students =(Jack, Jill, Abe, Henry, Mary, ..., Judy)

Exams =(exam1, exam2, exam3)

Days of Week = (S, M, T, W, Th, F, Sa)

Months = (Jan, Feb, Mar, Apr, ..., Nov, Dec)

## 3.2 The List ADT

### Operations:

Create a linear list

determine whether the list is empty

determine the length of the list

find the kth of the element

search for a given element

delete the kth element

insert a new element just after the kth



## 3.2 The List ADT

### ADT specification of a linear list

AbstractDataType LinearList

{ instances

ordered finite collections of zero or more elements

operations

Create();            Destroy();

IsEmpty();          Length();

Find(k,x);          Search(x);

Delete(k,x);        Insert(k,x);

Output(out);

}

## 3.2.1. Simple Array Implementation of Lists

1. Use an array to represent the instance of an object

$(e_1, e_2, \dots, e_n)$

element    0    1    2                      n-1                      MaxSize-1



length=n

each position of the array is called a cell or a node  
mapping formula:  $\text{location}(i) = i - 1$      $O(1)$

### 3.2.1. Simple Array Implementation of Lists

- Search( *x* )       $O(\text{length})$

$L = (a, b, d, b, e)$

Search( $d$ ) = 3

Search( $a$ ) = 1

Search( $z$ ) = 0

$$\begin{aligned} \text{ACN} &= (1+2+\dots+n) / n \\ &= (1+n) * n / (2n) = (n+1) / 2 \end{aligned}$$

### 3.2.1. Simple Array Implementation of Lists

- `remove( k,x)`

delete the  $k$ 'th element and return it in  $x$

$L = (a,b,c,d,e)$

$delete(2,x) \Rightarrow$

$L=(a,c,d,e), \quad x=b,$

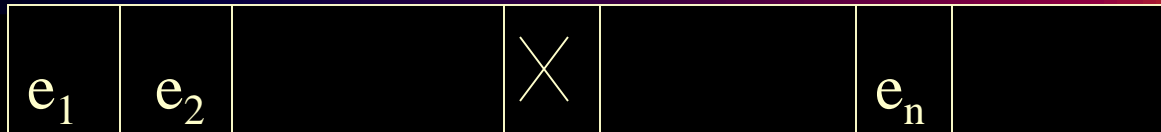
*and index of  $c,d,e$  decrease by 1*

$delete(0) \Rightarrow$  error

$delete(20) \Rightarrow$  error

$O(n)$

### 3.2.1. Simple Array Implementation of Lists



$$AMN = \sum_{i=0}^{n-1} (n-i-1)/n = (n-1+n-2+\dots+1+0)/n = (n-1)/2$$

### 3.2.1. Simple Array Implementation of Lists

- `insert (x , i)`

insert  $x$  after the  $i$ 'th element

$L = (a,b,c,d,e,f,g)$

*insert(0,h) =>*

$L = (h,a,b,c,d,e,f,g)$

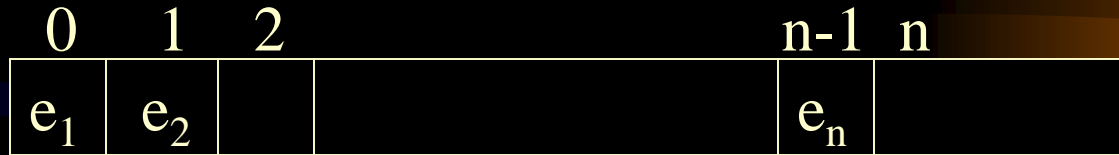
index of  $a,b,c,d,e,f$ , and  $g$  increase by 1

*insert(10,h) => error*

*insert(-6,h) => error*

$O(n)$

### 3.2.1. Simple Array Implementation of Lists



$$AMN = \sum_{i=0}^n (n-i) / (n+1) = (n+n-1+\dots+1+0) / (n+1) = n/2$$

## 3.2.1. Simple Array Implementation of Lists

### 2. Use array Implementation

merit : easy Search.

short coming : Insertion and Removing(Deletion) spend  
a lot of time.

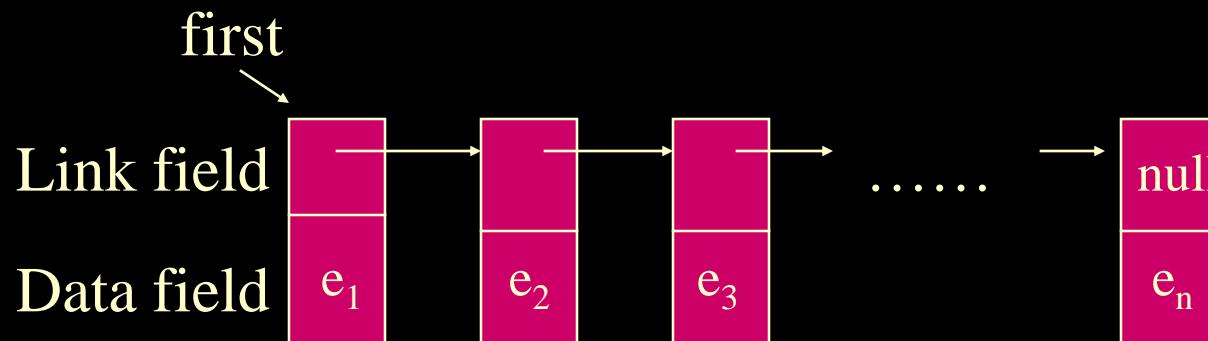


## 3.2.2. Linked Lists

In order to avoid the linear cost of insertion and deletion.

- 1) Each node of a data object keeps a link or a pointer about the location of other relevant nodes

$$L=(e_1,e_2,\dots\dots\dots e_n)$$



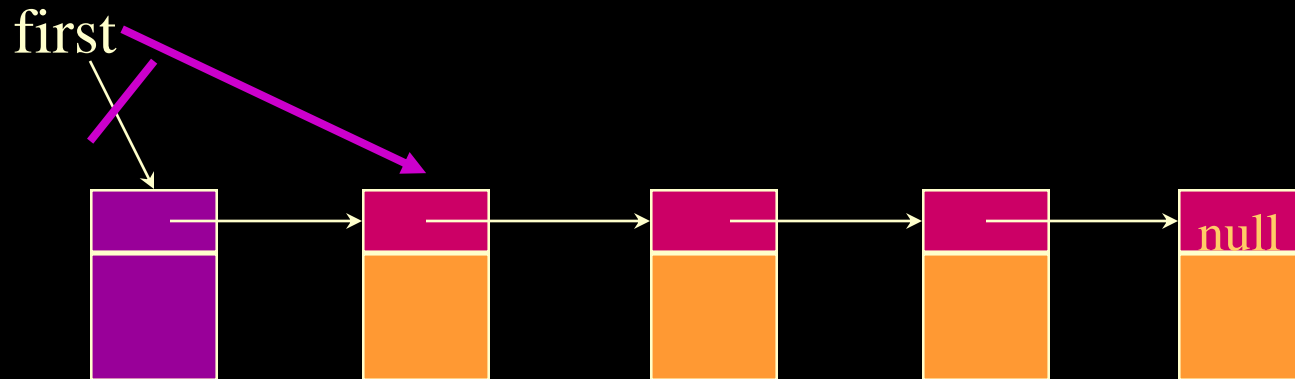
## 3.2.2. Linked Lists

- The figure above is called a single linked list, and the structure is also called a chain
- A chain is a linked list in which each node represents one element.
- There is a link or pointer from one element to the next.
- The last node has a **null** pointer.

## 3.2.2. Linked Lists

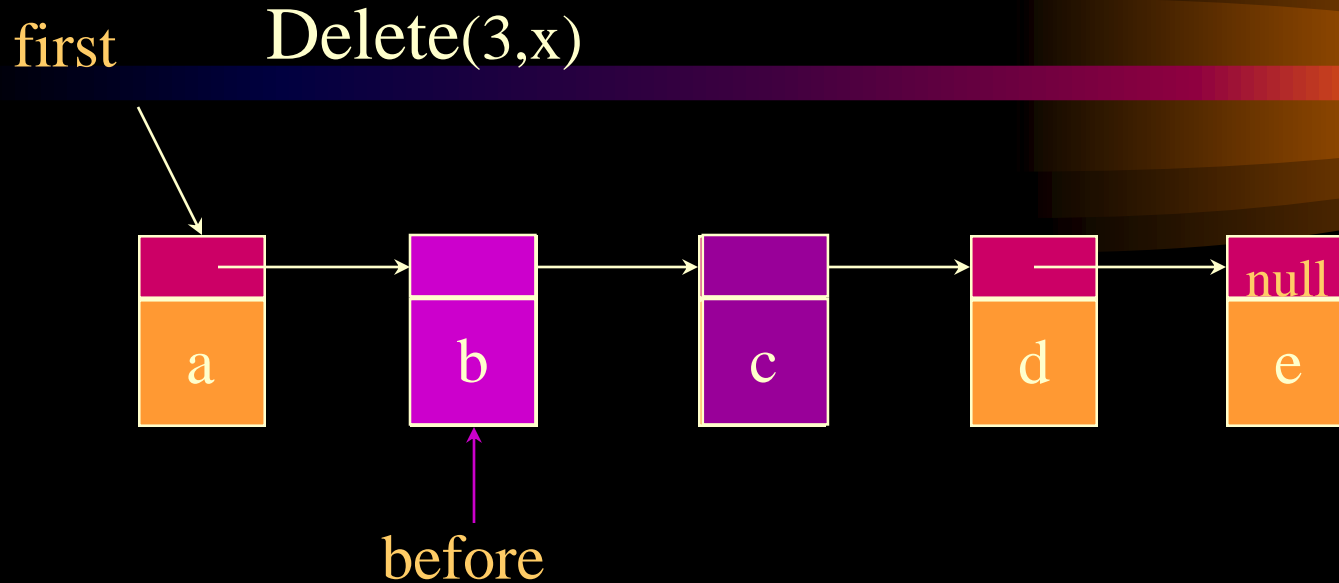
- Deletion a element of a chain

Delete(1,x)



`first = first.link;`

## 3.2.2. Linked Lists

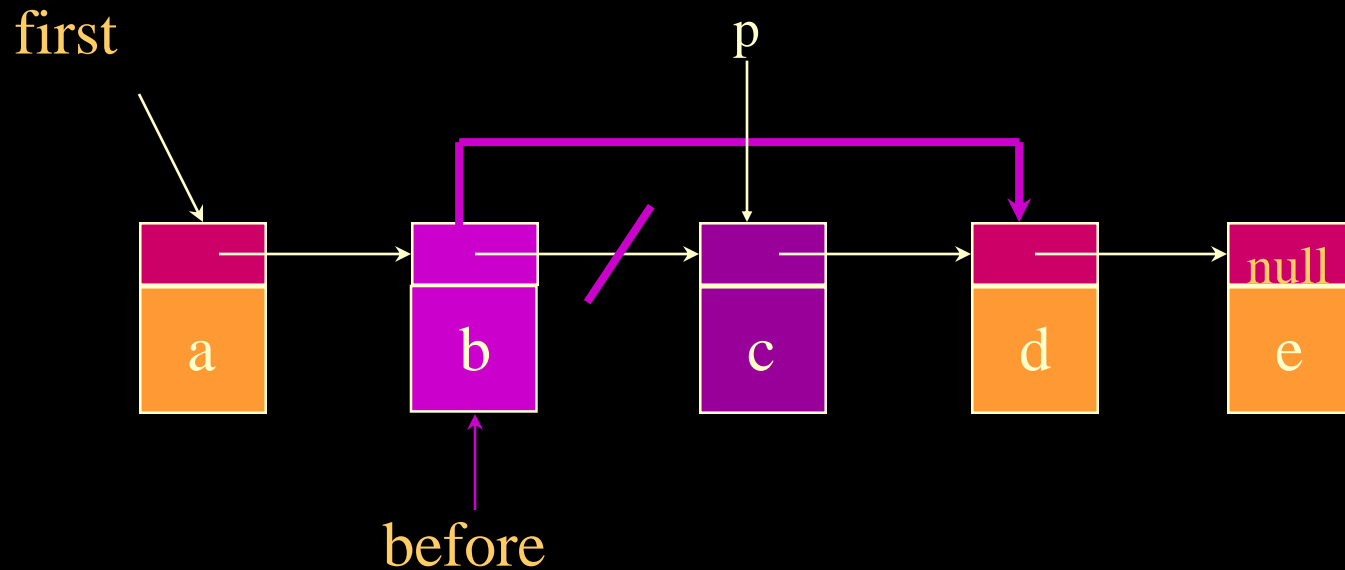


first get to node just before node to be removed

`before = first .link;`

## 3.2.2. Linked Lists

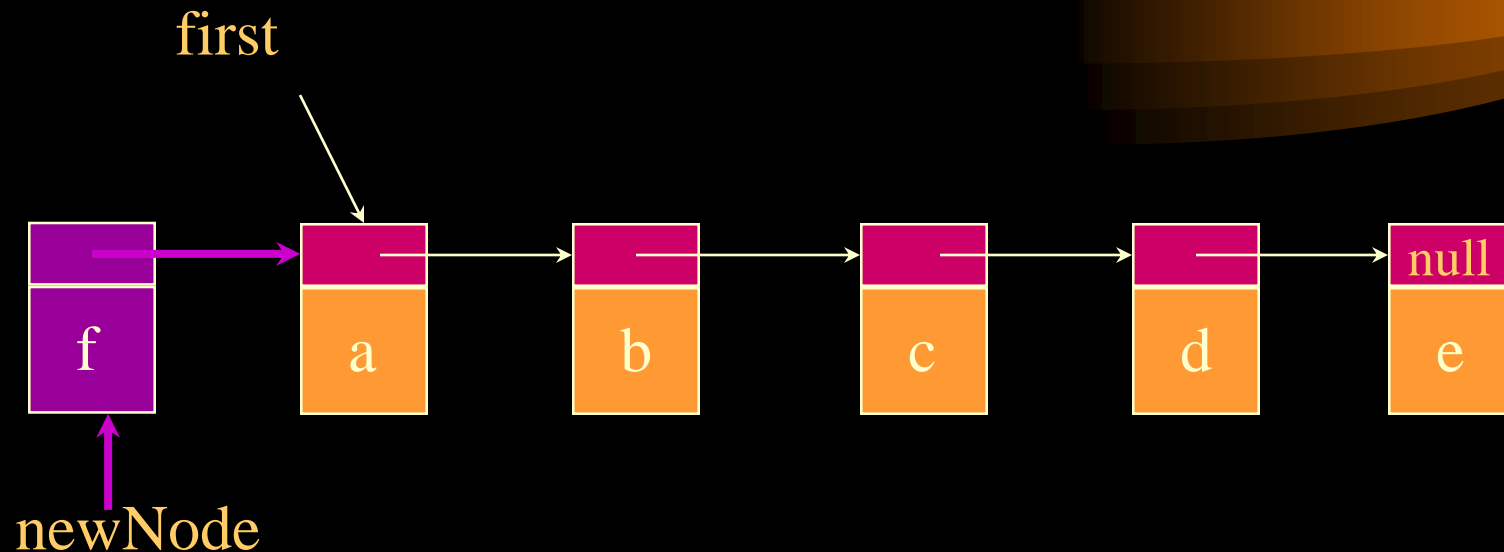
now change pointer in before



```
before .link = before .link .link;
```

## 3.2.2. Linked Lists

- insert operation ----insert(0,'f')

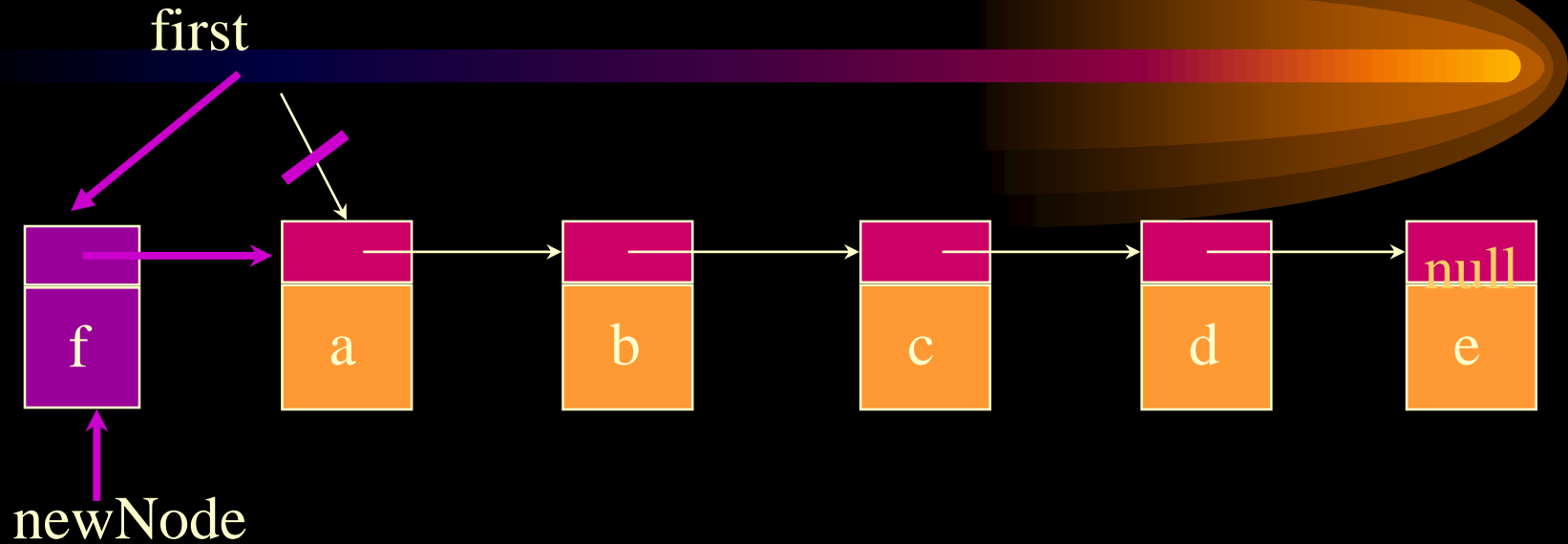


Step 1: get a node, set its data and link fields

```
ChainNode newNode =
```

```
new ChainNode('f', first);
```

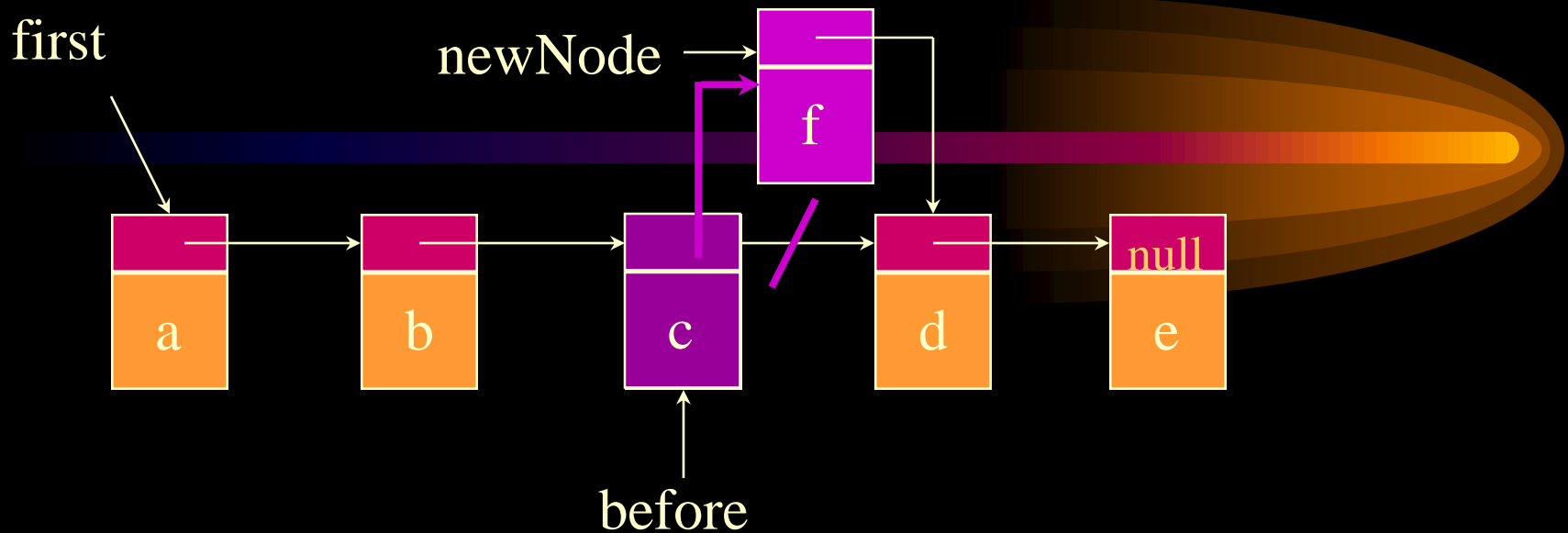
## 3.2.2. Linked Lists



Step 2: update first

```
first = newNode;
```

`insert(3, 'f')`



1. first find node whose index is 3
2. next create a node and set its data and link fields

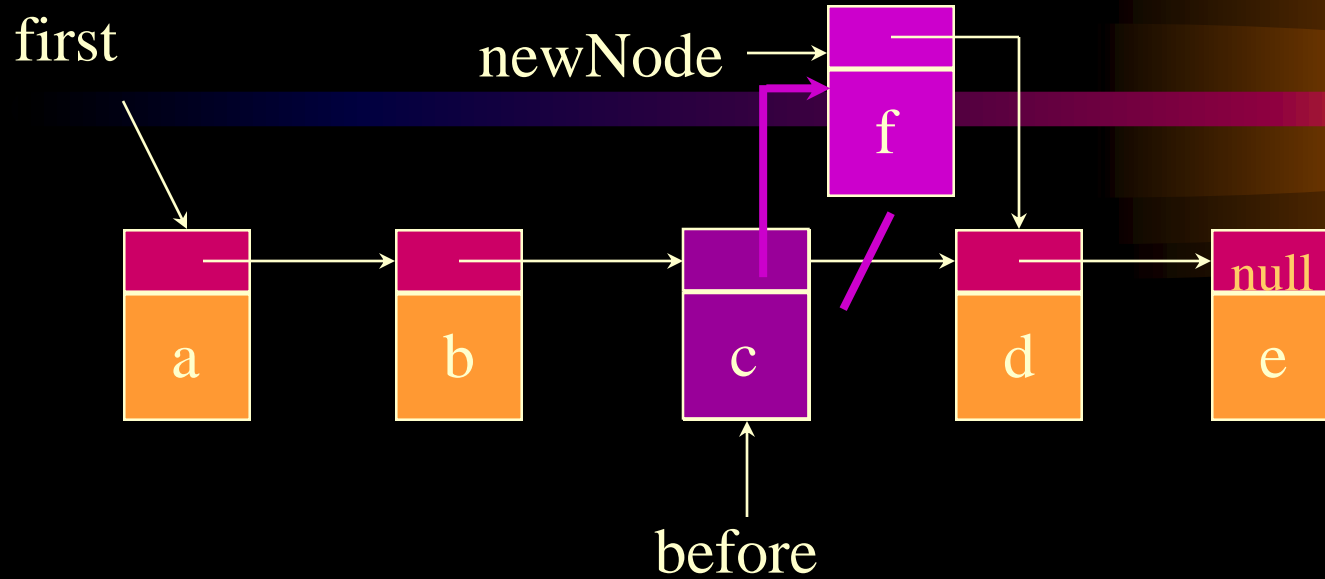
```
ChainNode newNode = new ChainNode('f',before .link);
```

3. finally link **before** to **newNode**

```
before .link = newNode;
```



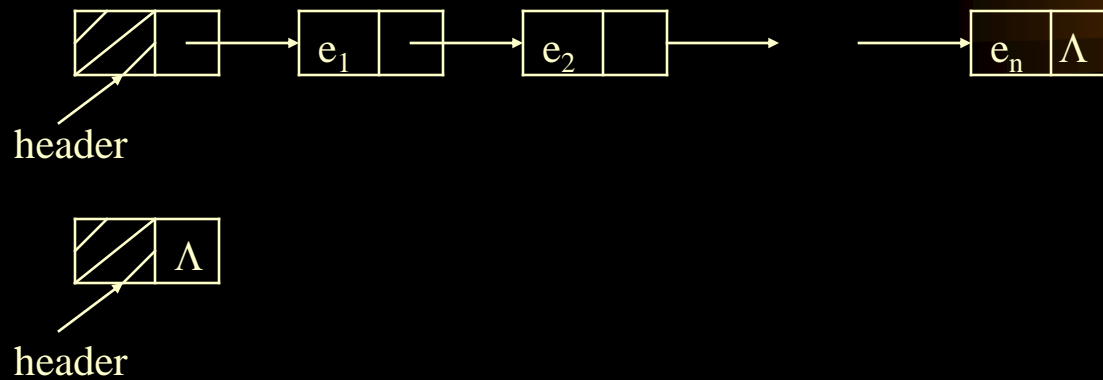
## Two-Step insert(3, 'f')



```
before = first .link .link;  
newNode .link = before .link;  
before .link = newNode;
```

## 3.2.3 Programming Details

### 1. Header (dummy node)



## 3.2.2. Linked Lists

### 2. Class definition

ListNode	——	代表结点的类	} 都是包DataStructures的一部分
LinkedList	——	代表表本身的类	
LinkedListItr	——	代表位置的类	

## 3.2.2. Linked Lists

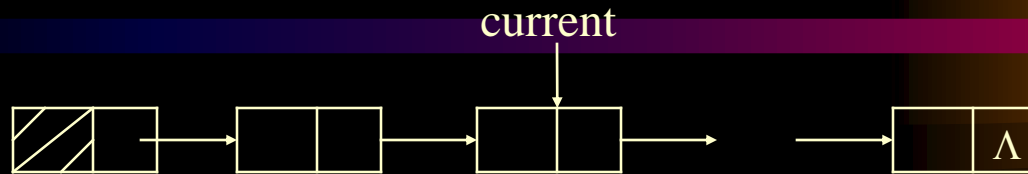
### 1) ListNode class



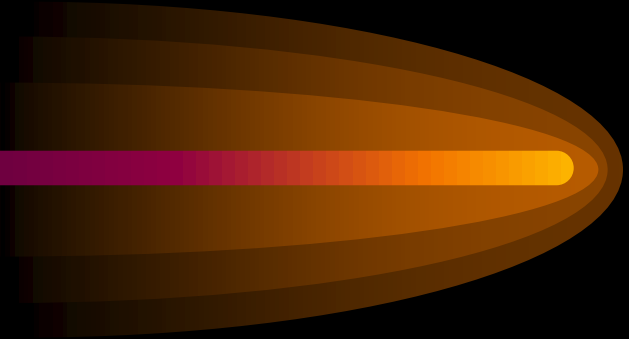
```
package DataStructures;
class ListNode
{
    ListNode( object theElement)
    {
        this( theElement, null);
    }
    ListNode( object theElement, ListNode n)
    {
        element = theElement;
        next = n;
    }
    object element;
    ListNode next;
}
```

## 3.2.2. Linked Lists

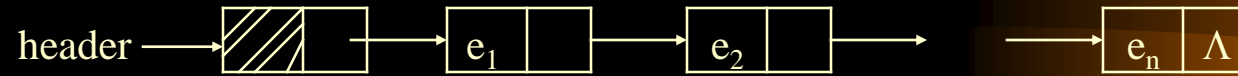
2) Iterator class for linked lists



```
package DataStructures
public class LinkedListItr
{  LinkedListItr( ListNode theNode)
    {  current = theNode;
    }
    public boolean isPastEnd( )
    {  return current == null;
    }
    public object retrieve( )
    {  return isPastEnd( ) ? Null : current.element;
    }
    public void advance( )
    {  if( ! isPastEnd( ) )
        current = current.next;
    }
    ListNode current;
}
```



### 3) LinkedList class



```
package DataStructures;
public class LinkedList
{
    public LinkedList( )
    {
        header = new ListNode( null );
    }
    public boolean isEmpty( )
    {
        return header.next == null ;
    }
    public void makeEmpty( )
    {
        header.next = null;
    }
    public LinkedListItr zeroth( )
    {
        return new LinkedListItr( header );
    }
    public LinkedListItr first( )
    {
        return new LinkedListItr( header.next );
    }
    public LinkedListItr find( object x )
    public void remove( object x )
    public LinkedListItr findPrevious( object x )
    public void insert( object x, LinkedListItr p )

    private ListNode header;
}
```



Method to print a list

```
public static void printList( LinkedList theList )
```

```
{  if ( theList.isEmpty( ) )
```

```
    System.out.print( "Empty list" ) ;
```

```
    else
```

```
    {  LinkedListItr itr = theList.first( );
```

```
        for( ; ! Itr.isPastEnd( ); itr. Advance( ) )
```

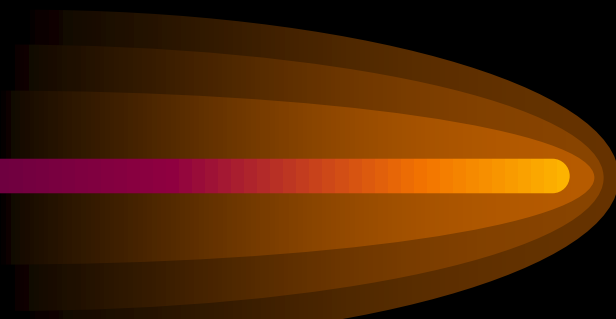
```
            System.out.print( itr.retrieve( ) + " " ) ;
```

```
    }
```

```
    System.out.println( );
```

```
}
```

# Operation:

- Constructors
  - isEmpty
  - makeEmpty
  - Zeroth and first return iterators corresponding to the header and first element.
  - Find(x)
- 

```
public LinkedListItr find (object x)
{
    ListNode itr = header.next;
    while ( itr != null && !itr.element.equals( x ) )
        itr = itr.next;
    return new LinkedListItr( itr );
}
```

$O(N)$

- Remove( x )

```
public void remove( object x )
```

```
{   LinkedListItr p = findprevious( x );  
    if( p.current.next != null )  
        p.current.next = p.current.next.next;  
}
```

O(1)

- Findprevious ( x )

```
public LinkedListItr findPrevious( object x )  
{  ListNode itr = header;  
    while( itr.next !=null && !itr.next.element.equals( x ))  
        itr = itr.next;  
    return new LinkedListItr( itr );  
}
```

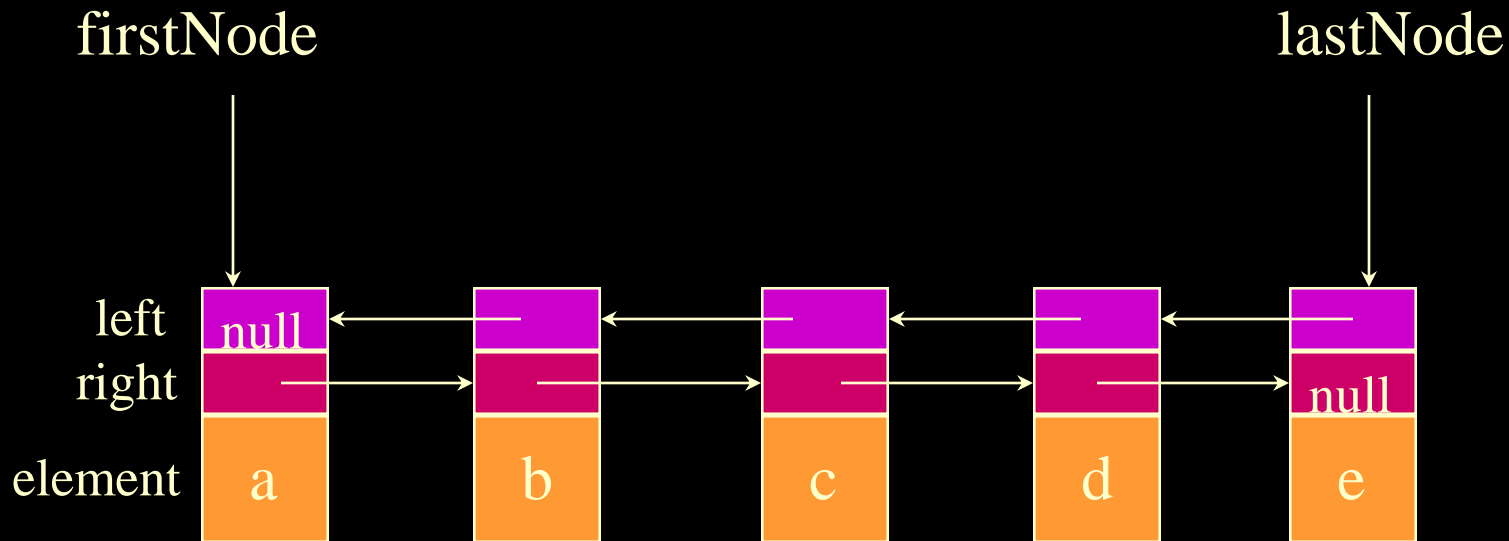
$O(N)$

- Insert(x, p)

```
public void insert( object x, LinkedListItr p)
{   if( p!=null && p.current != null )
        p.current.next = new ListNode( x, p.current.next );
}
```

O(1)

## 3.2.4. Doubly Linked Lists



### 3.2.4. Doubly Linked Lists

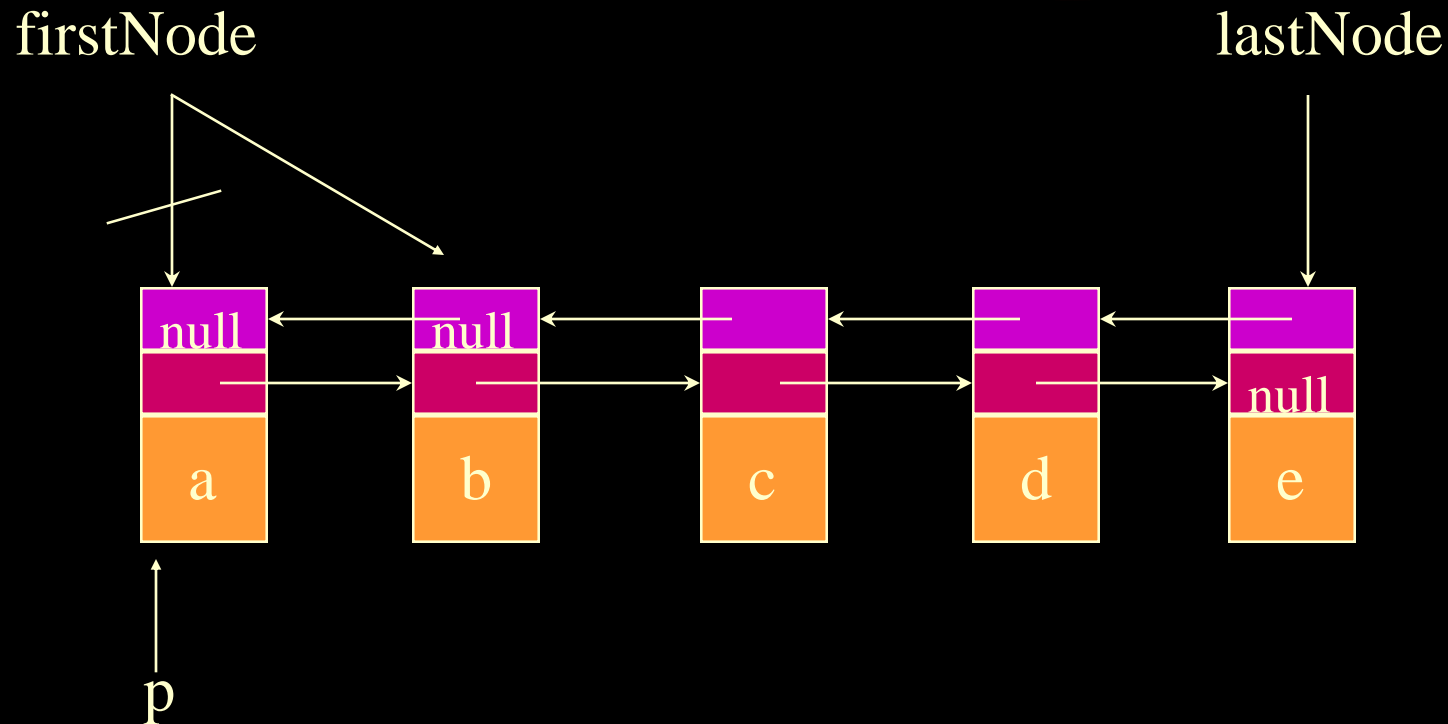
operations: insert delete

A decorative graphic consisting of several overlapping, elongated, teardrop-shaped elements in shades of brown and orange, pointing towards the right. A horizontal bar with a gradient from purple to yellow is positioned in front of the graphic, passing behind the text 'insert delete'.



## 3.2.4. Doubly Linked Lists

Delete(1)



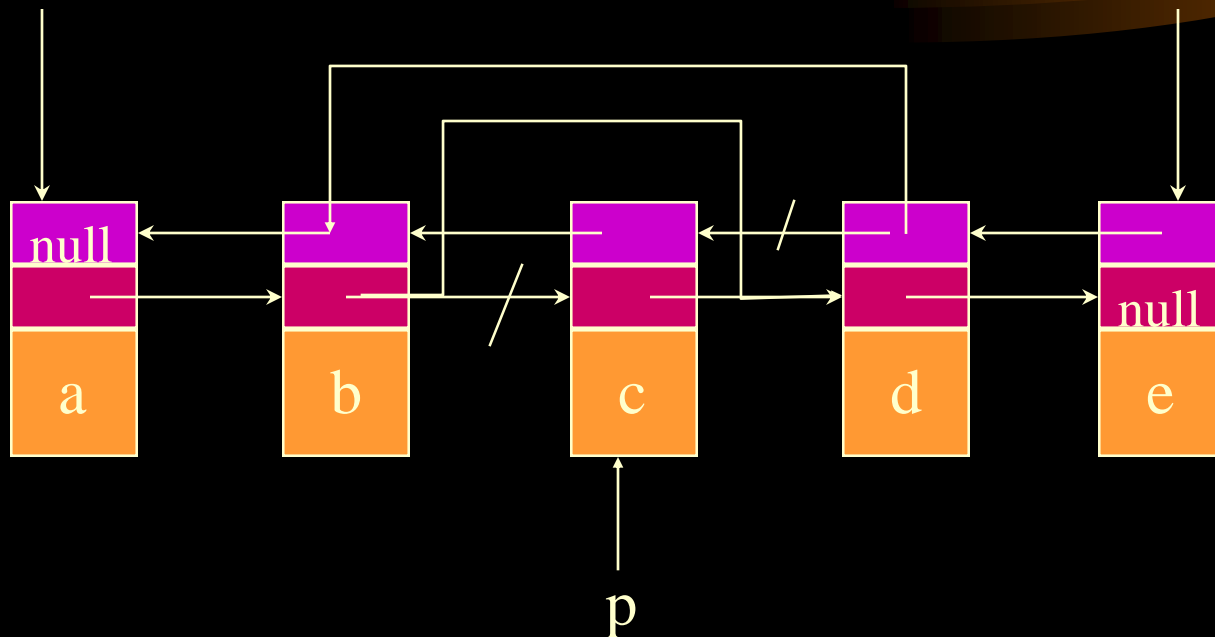
```
P=firstNode; firstNode=p .right; firstNode .left=null;
```

## 3.2.4. Doubly Linked Lists

Delete(3)

firstNode

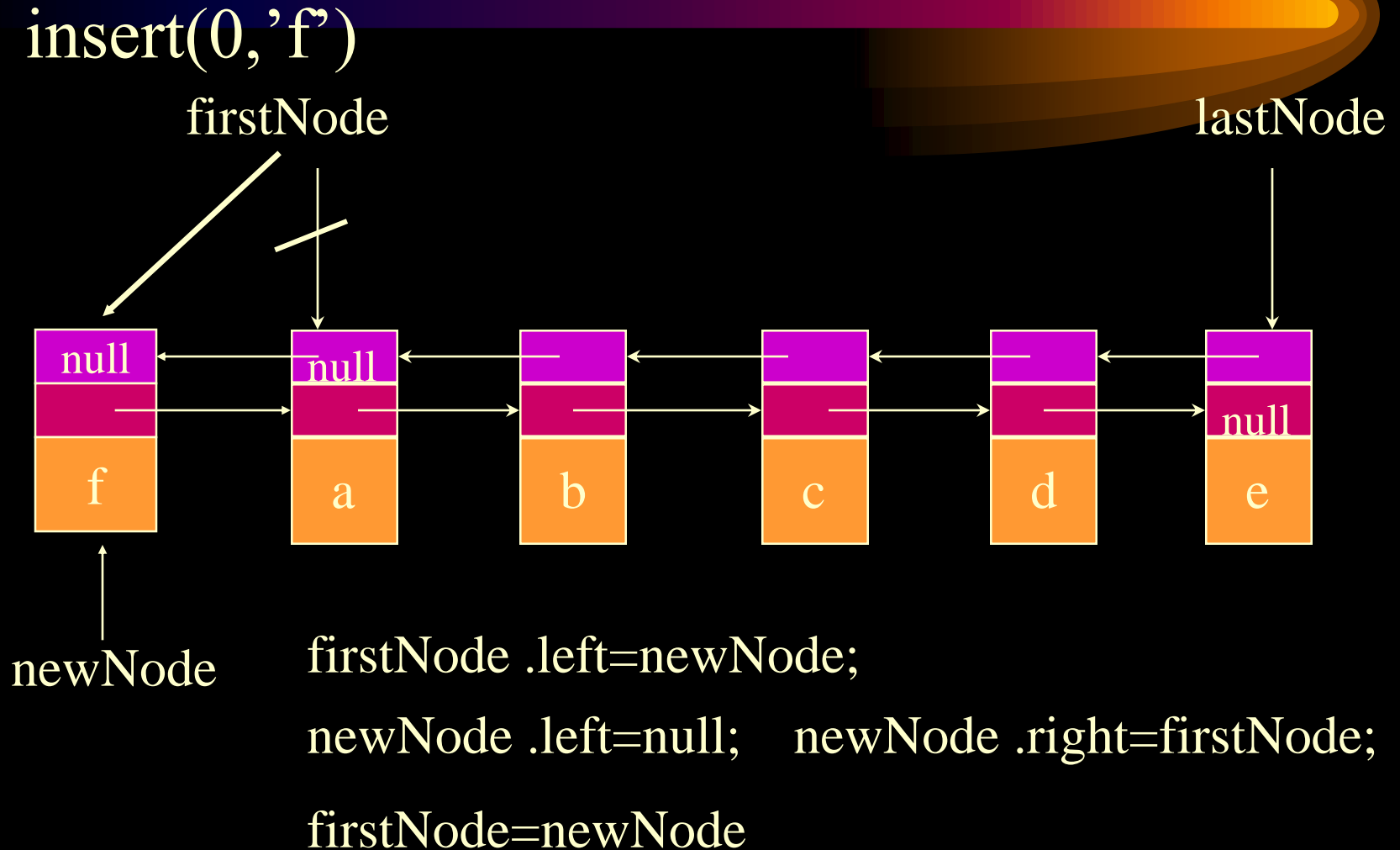
lastNode



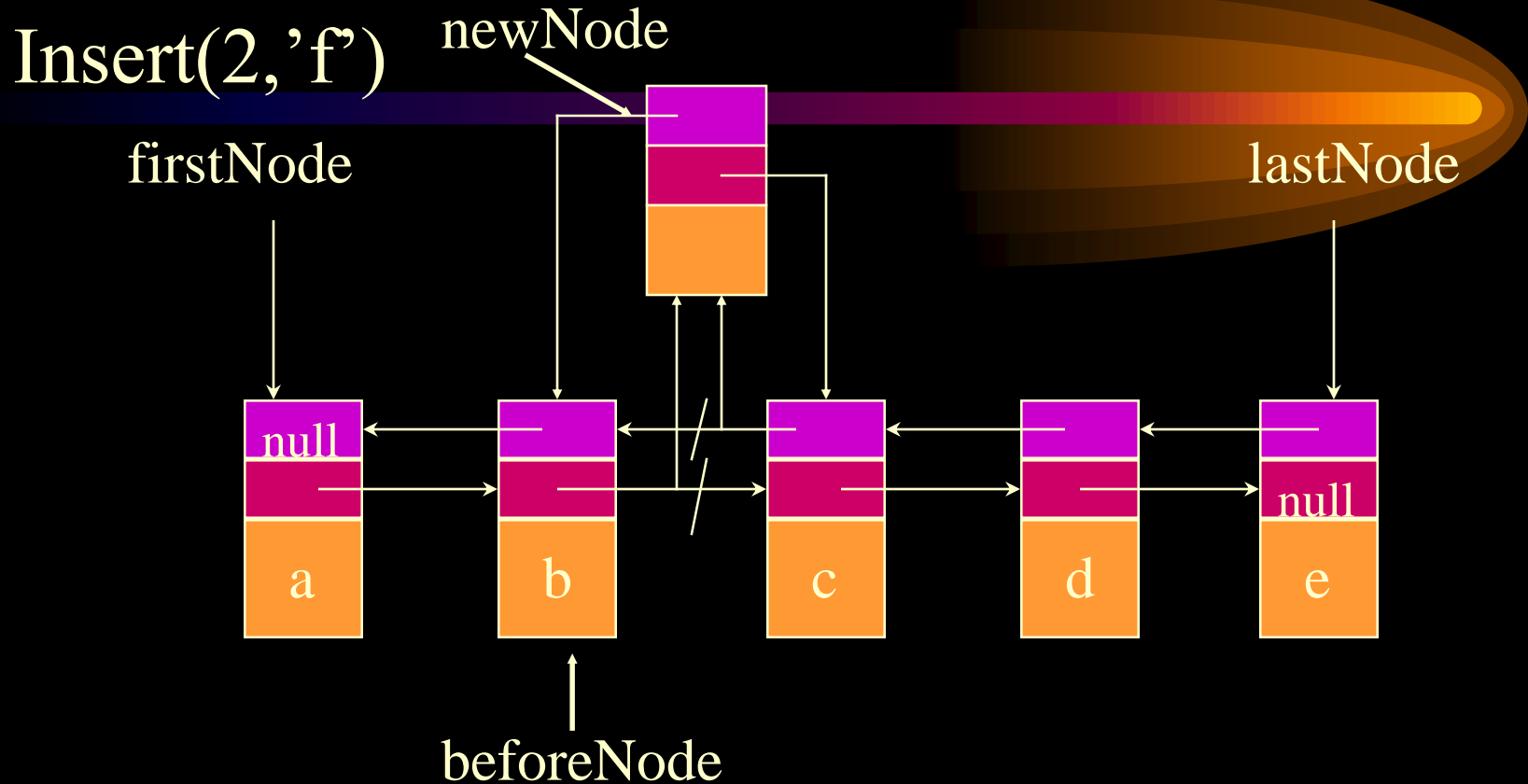
`P .left .right = p .right;`

`P .right .left = p .left;`

## 3.2.4. Doubly Linked Lists



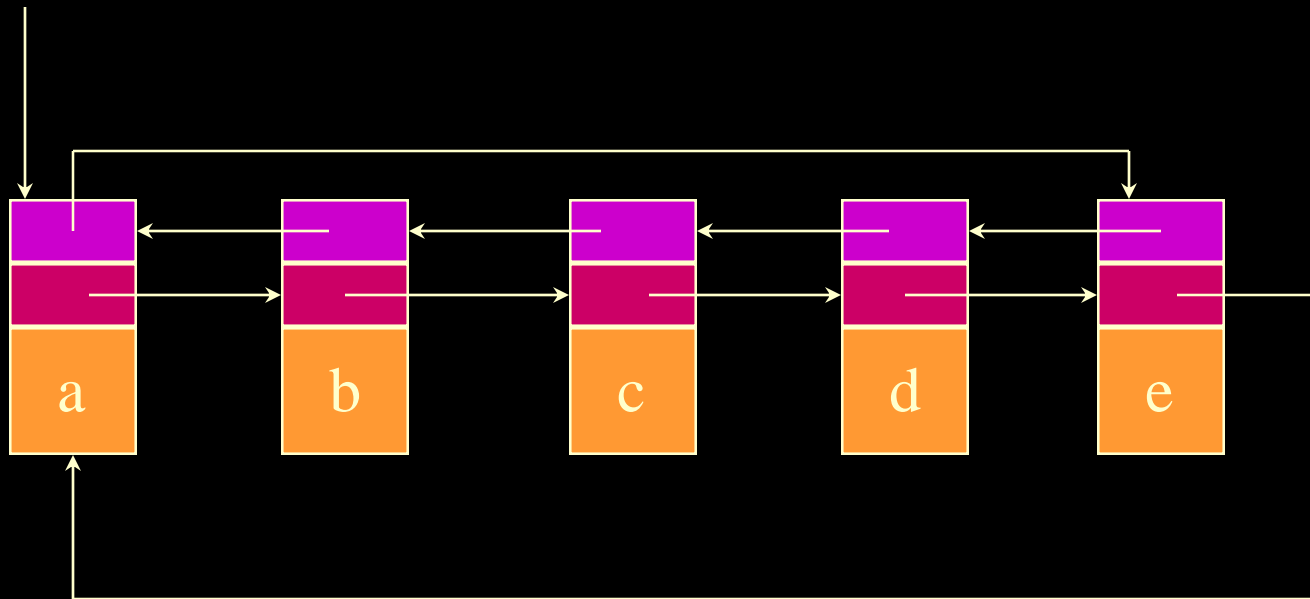
### 3.2.4. Doubly Linked Lists



```
newNode .left=beforeNode;   newNode .right=beforeNode .right;  
beforeNode .right .left=newNode;   beforeNode .right=newNode;
```

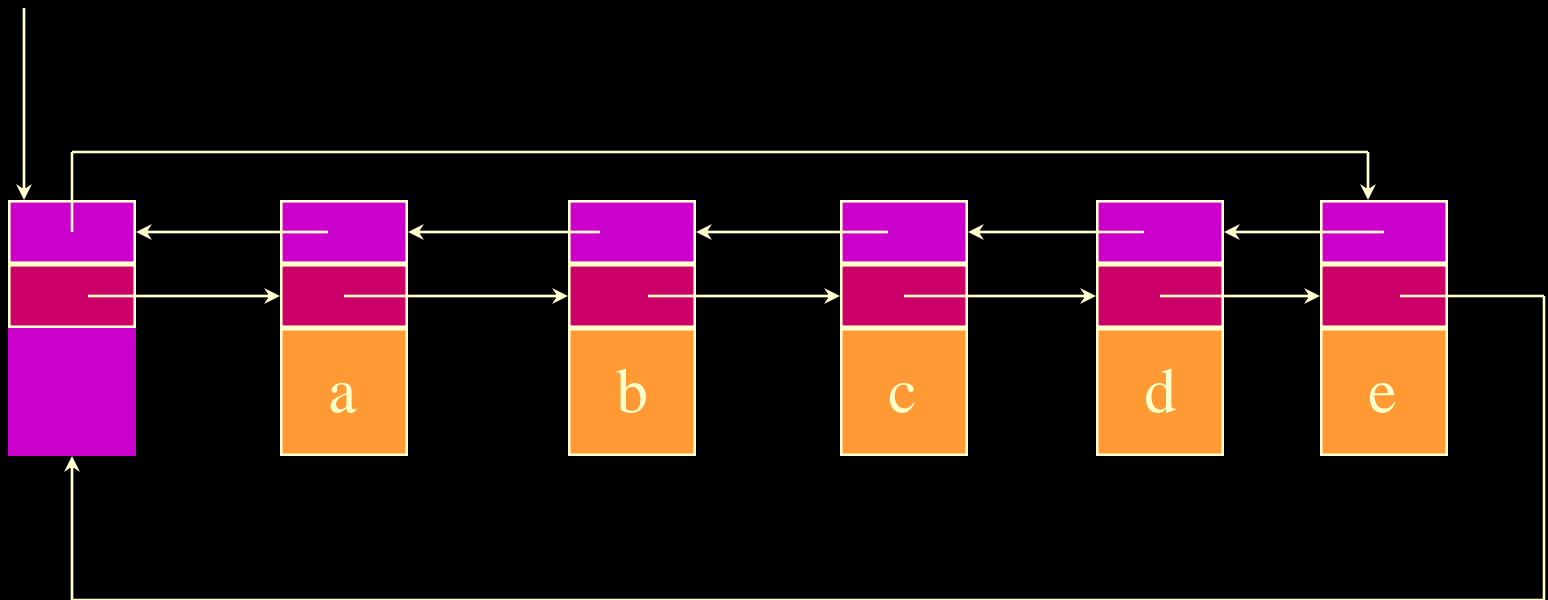
### 3.2.4. Doubly Linked Circular Lists

firstNode



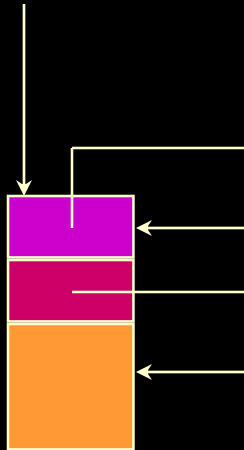
# Doubly Linked Circular List With Header Node

headerNode

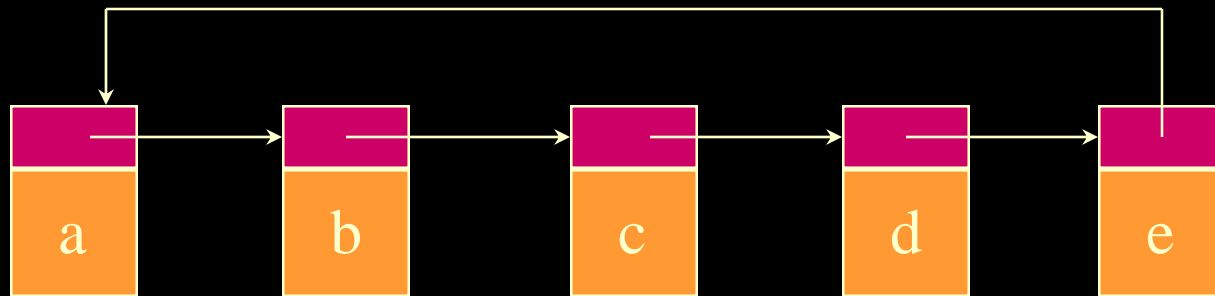


# Empty Doubly Linked Circular List With Header Node

headerNode



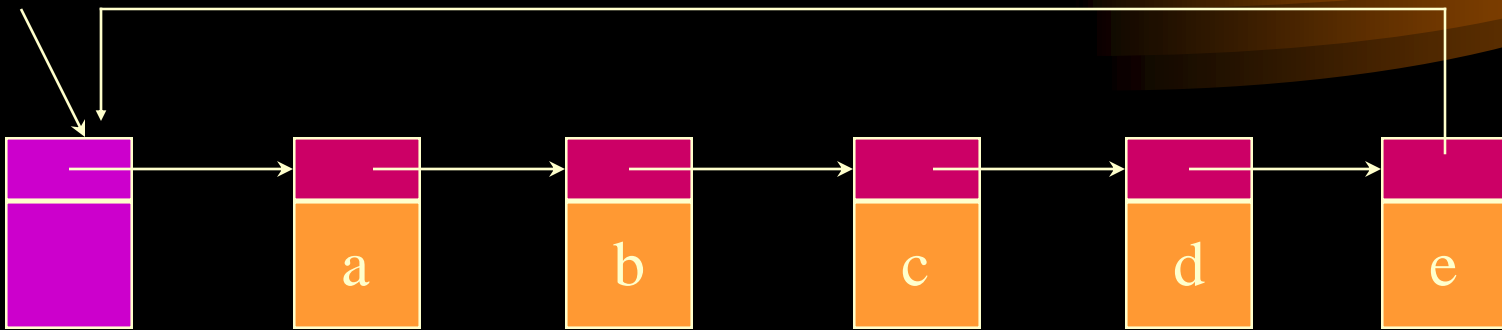
## 3.2.5. Circular Linked Lists



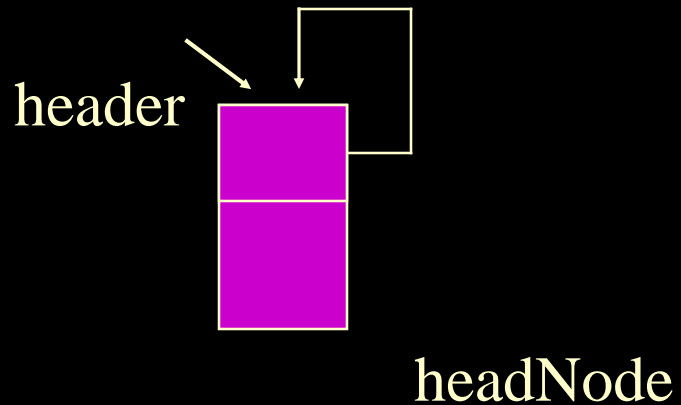


## 3.2.5. Circular Linked Lists

header



headNode

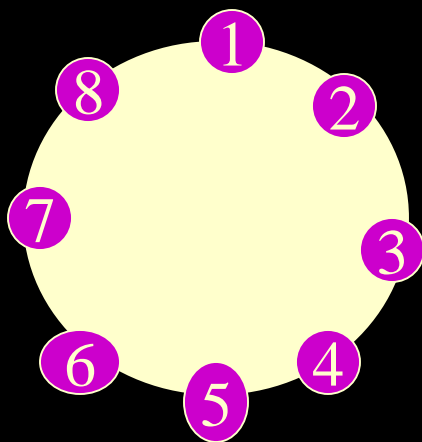


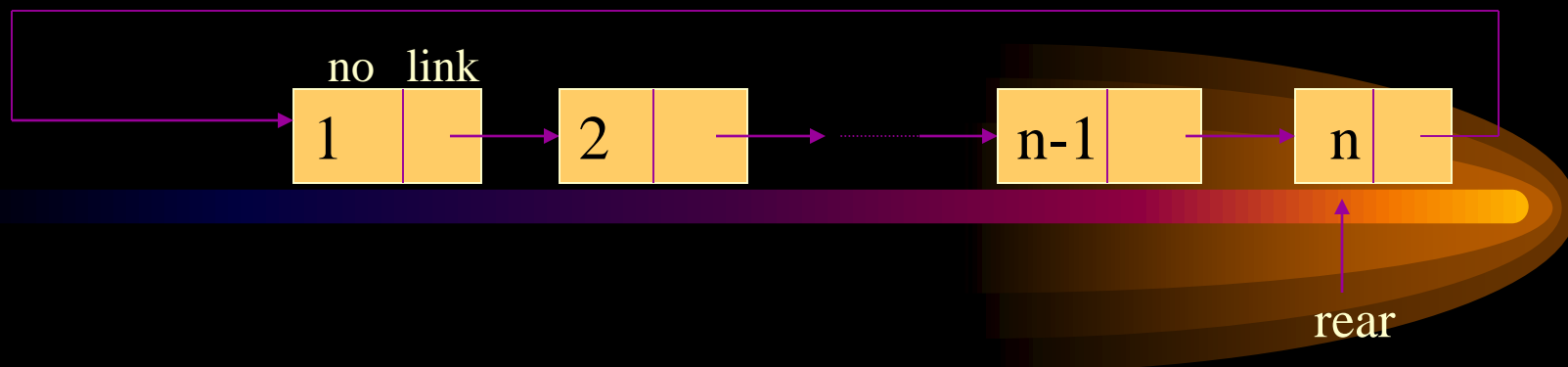
例子:

用循环链表求解约瑟夫(josephus)问题

约瑟夫问题：实际上是一个游戏。书中以旅行社从 $n$ 个旅客中  
选出一名旅客，为他提供免费环球旅行服务。

例， $n=8$ ,  $m=3$ （报数）从1号开始报数出列顺序为：3, 6, 1, 5, 2,  
8, 4。最后一个编号7的旅客将赢得环球旅游。





rear: 每次指向要出队列的前一个结点

出队列的人也用链表来表示:

head: 指向出队列结点链表的开头结点

p: 指向出队列结点链表的尾结点

以上rear, head, p都是ListNode的一个对象引用。

```
1.  w = m;
2.  for( int i = 1; i<= n-1; i++)
    {  1) for (int j =1; j<=w-1; j++) rear = rear.link;
        2) if (i == 1)
            { head = rear.link ; p = head; }
            else
                { p.link = rear.link; p = rear.link; }
        3) rear.link = p.link;
    }
3.  P.link = rear;
    rear.link = null;
```

## 3.2.6. Examples

### 1. Polynomial ADT

$$p_n(x) = a_0x^{e0} + a_1x^{e1} + a_2x^{e2} + \dots + a_nx^{en}$$

- Array implementation

example:  $p(x) = 3x^4 - 5x^3 + 8x^2 + 2x - 1$

coeffArray	3	-5	8	2	-1
------------	---	----	---	---	----

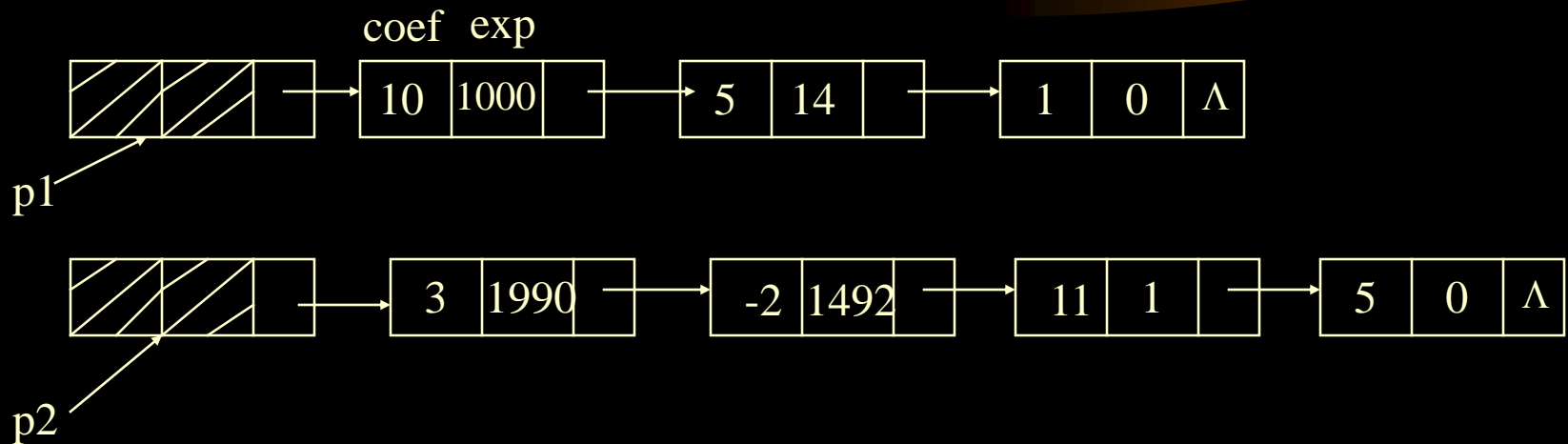
$$p(x) = 2x^{1000} + 8x^{50} - 2$$

## 3.2.6. Examples

- Linked list representations

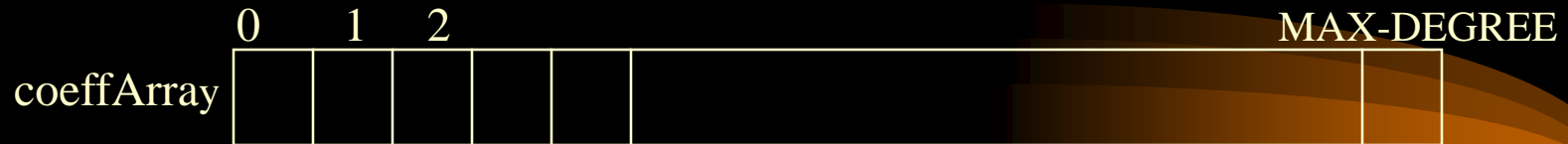
$$p_1(x) = 10x^{1000} + 5x^{14} + 1$$

$$p_2(x) = 3x^{1990} - 2x^{1492} + 11x + 5$$



polynomial operations: addition , multiplication, and so on.

# 1) Array implementation of the polynomial ADT



`highPower`

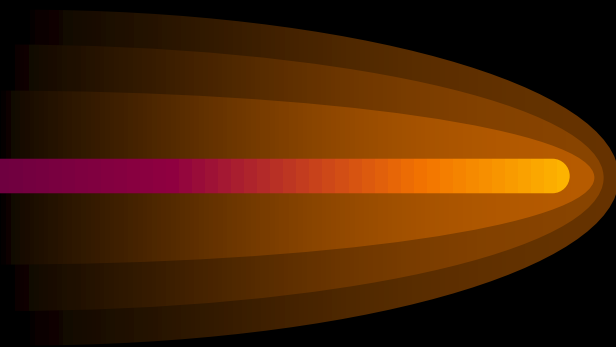
```
public class Polynomial
{
    public Polynomial() { zeroPolynomial(); }
    public void insertTerm( int coef, int exp )
    public void zeroPolynomial( )
    public Polynomial add( Polynomial rhs )
    public Polynomial multiply( Polynomial rhs ) throws Overflow
    public void print( )

    public static final int MAX-DEGREE = 100;
    private int coeffArray[ ] = new int [MAX-DEGREE + 1];
    private int highPower = 0;
}
```

```

public void zeroPolynomial( )
{
    for( int i = 0; i<=MAX-DEGREE; i++)
        coeffArray[i] = 0;
    highPower = 0;
}

```



```

public Polynomial add( Polynomial rhs )
{
    Polynomial sum = new Polynomial( );
    sum.highPower = max( highPower, rhs.highPower );
    for( int i = sum.highPower; i>=0; i--)
        sum.coeffArray[i] = coeffArray[i] + rhs.coeffArray[i] ;
    return sum;
}

```

Example:

$$p_1(x) = 3x^8 - 5x^3 + 3x - 1$$

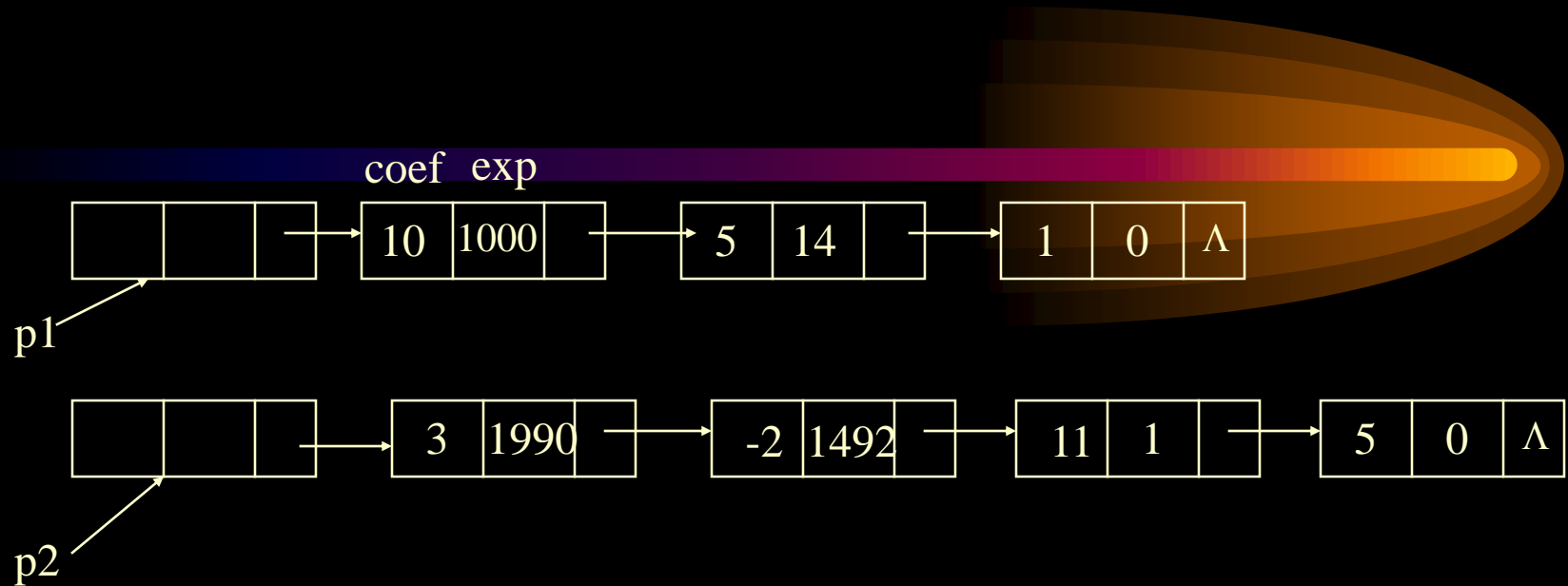
$$p_2(x) = 4x^6 + 2x^2 + 2$$

0	1	2	3	4	5	6	7	8
-1	3	0	-5	0	0	0	0	3
2	0	2	0	0	0	4	0	0



```
Public Polynomial multiply( Polynomial rhs ) throws overflow
{
    Polynomial product = new Polynomial( );
    product.highPower = highPower + rhs.highPower;
    if( product.highPower > MAX-DEGREE )
        throw new overflow( );
    for( int i = 0; i<= highPower; i++ )
        for( int j = 0; j<= rhs.highPower; j++ )
            product.coeffArray[ i + j ] += coeffArray[i] * rhs.coeffArray[j];
    return product;
}
```

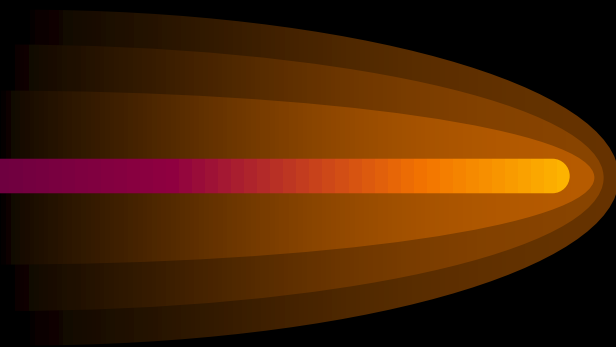
## 2) Class skeletons for linked list implementation of the Polynomial ADT



```
public class Literal
{ //Various constructors(not shown)
  int coefficient;
  int exponent;
}

public class Polynomial
{ public Polynomial( ) { /* Exercise */ }
  public void insertTerm( int coef, int exp ) { /* Exercise */ }
  public void zeroPolynomial( ) { /* Exercise */ }
  public Polynomial add( Polynomial rhs ) { /* Exercise */ }
  public Polynomial multiply( Polynomial rhs ) { /* Exercise */ }
  public void print( ) { /* Exercise */ }

  private List terms; /* A List of Literals, sorted by exponent */
}
```



多项式相加：

\*问题：  $A(X) = 2X^{100} + 3X^{14} + 2X^8 + 1$

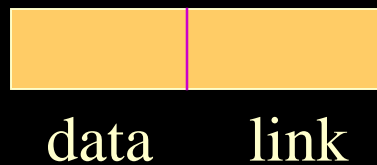
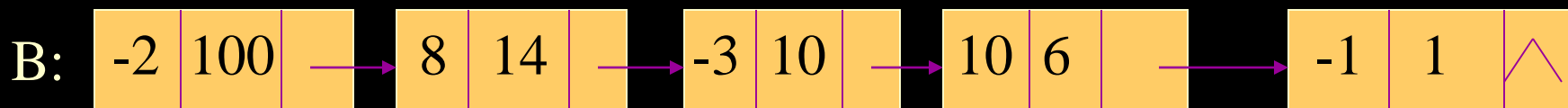
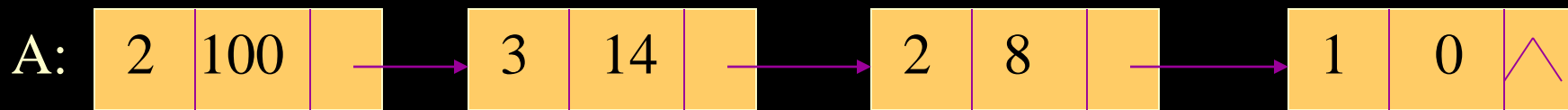
$$B(X) = -2X^{100} + 8X^{14} - 3X^{10} + 10X^6 - X$$

$$A(X) + B(X) = 11X^{14} - 3X^{10} + 2X^8 + 10X^6 - X + 1$$

存放非零指数的系数与指数，因此每个结点有三个域组成。



item



$$A(X)+B(X)=A(X)$$

具体实现时，并不要再重新申请结点，完全利用原来两个链表的结点。

方法：设4个引用变量：

pa, pb, pc, p(c++需要)

1)初始化： pc , pa , pb ;

2)当pa和pb都有项时

pc永远指向相加时结果链表的最后一个结点。

a)指数相等(  $pa.exp == pb.exp$  )

对应系数相加:  $pa.coef = pa.coef + pb.coef$  ;

$p = pb(c++ \text{需要})$  ;  $pb$ 前进 ;

if (系数相加结果为0){  $p = pa$ ;  $pa$ 前进; }

else {  $pc.link = pa$ ;  $pc = pa$ ;  $pa$ 前进 }

b)指数不等  $pa.exp < pb.exp$  //pb要插入结果链表

{  $pc.link = pb$  ;  $pc = pb$  ;  $pb$ 前进 }

c)指数不等  $pa.exp > pb.exp$  //pa要插入结果链表

{  $pc.link = pa$  ;  $pc = pa$  ;  $pa$ 前进 }

3)当两链表中有一链表为空, 则将另一链表链入结果链表就可以

if ( $pb$ 空了){  $pc.link = pa$  ; }

else  $pc.link = pb$  ;

算法分析：设两个多项式的项数分别是 $m$ 和 $n$ ，则总的比较次数为 $O(m+n)$

最坏情况下：两个多项式的指数项都不等且交叉递增，如

$$A(x): \quad a_5x^5 + a_3x^3 + a_1x + a_0 \quad m=4 \quad \text{比较 } m+n-1 \text{ 次}$$

$$B(x): \quad b_4x^4 + b_2x^2 + b_0 \quad n=3$$



## 考研统考题

(15分)已知一个带有表头结点的单链表,结点结构为 

data	link
------	------

, 假设该链表只给出了头指针 list. 在不改变链表的前提下, 请设计一个尽可能高效的算法, 查找链表中倒数第k个位置上的结点(k为正整数). 若查找成功, 算法输出该结点的data域的值, 并返回1; 否则返回0. 要求:

- 1) 描述算法的基本设计思想;
- 2) 描述算法的详细实现步骤;
- 3) 根据设计思想和实现步骤, 采用程序设计语言描述算法( 使用C或C++或JAVA语言实现), 关键之处请给出简要注释.

## 3.2.6. Cursor implementation of Linked Lists

use array to implement linked list:

cursorSpace		element	next
0			1
1			2
2			3
3			4

header	→	0		6
		1	30	2
p	→	2	50	3
		3	67	8
		4	15	7
		5	81	0
		6	10	4
		7	20	1
		8	78	5

P = p.next

p = cursorSpace[p].next

p.data

cursorSpace[p].data

## 3.2.6. Cursor implementation of Linked Lists

1) Node and iterator for cursor implementation of linked lists

```
class CursorNode
{  CursorNode( object theElement )
    {  this( theElement, 0 );  }
    CursorNode( object theElement, int n )
    {  element = theElement;
        next = n;
    }

    object element;
    int next;
}
```

## 3.2.6. Cursor implementation of Linked Lists

```
public class CursorListItr
{
    CursorListItr( int theNode ) { current = theNode; }
    public boolean isPastEnd( ) { return current == 0; }
    public object retrieve( )
    {
        return isPastEnd( ) ? null:
            CursorList.cursorSpace[ current ].element;
    }
    public void advance( )
    {
        if( !isPastEnd( ))
            current = CursorList.cursorSpace[ current ].next;
    }

    int current;
}
```

## 3.2.6. Cursor implementation of Linked Lists

2) Class skeleton for CursorList

```
public class CursorList
```

```
{ private static int alloc( )
```

```
    private static void free( int p)
```

```
    public CursorList( )
```

```
        { header = alloc( ); cursorSpace[ header ].next = 0; }
```

```
    public boolean isEmpty( )
```

```
        { return cursorSpace[ header ].next == 0; }
```

```
    public void makeEmpty( )
```

```
    public CursorListItr zeroth( )
```

```
        { return new CursorListItr( header ); }
```

```
    public CursorListItr first( )
```

```
        { return new CursorListItr( cursorSpace[ header ].next ); }
```

```
public CursorListItr find( object x )  
public void insert( object x, CursorListItr p)  
public void remove( object x )  
public CursorListItr findPrevious( object x )
```

```
private int header;  
static CursorNode [ ] cursorSpace;
```

```
private static final int SPACE-SIZE = 100;
```

```
static
```

```
{  cursorSpace = new CursorNode[ SPACE-SIZE ];  
    for( int i = 0; i<SPACE-SIZE; i++)  
        cursorSpace[ i ] = new CursorNode( null, i + 1 );  
    cursorSpace[ SPACE-SIZE-1].next = 0;  
}
```

```
}
```

## 3.2.6. Cursor implementation of Linked Lists

Some Routines:

- Alloc and free

```
private static int alloc( )
```

```
{  int p = cursorSpace[ 0 ].next;  
    cursorSpace[0].next = cursorSpace[p].next;  
    if( p == 0 )  
        throw new OutOfMemoryError( );  
    return p;  
}
```

```
private static void free( int p )
```

```
{  cursorSpace[p].element = null;  
    cursorSpace[p].next = cursorSpace[0].next;  
    cursorSpace[0].next = p;  
}
```

### 3.2.6. Cursor implementation of Linked Lists

- Find routine----cursor implementation

```
public CursorListItr find( object x )  
{   int itr = cursorSpace[ header ].next;  
    while( itr != 0 && !cursorSpace[ itr ].element.equals( x ) )  
        itr = cursorSpace[ itr ].next;  
    return new CursorListItr( itr );  
}
```



## 3.2.6. Cursor implementation of Linked Lists

- Insertion routine for linked lists---cursor implementation

```
public void insert( object x, CursorListItr p )
```

```
{  if( p != null && p.current != 0)
```

```
    {  int pos = p.current;
```

```
        int tmp = alloc( );
```

```
        cursorSpace[ tmp ].element = x;
```

```
        cursorSpace[ tmp ].next = cursorSpace[ pos ].next;
```

```
        cursorSpace[ pos ].next = tmp;
```

```
    }
```

```
}
```

## 3.2.6. Cursor implementation of Linked Lists

- Deletion routine for linked lists----cursor implementation

```
public void remove( object x )
{   CursorListItr p = findPrevious( x );
    int pos = p.current;

    if( cursorSpace[ pos ].next != 0 )
    {   int tmp = cursorSpace[ pos ].next;
        cursorSpace[ pos ].next = cursorSpace[ tmp ].next;
        free( tmp );
    }
}
```

- makeEmpty for cursor implementation

```
public void makeEmpty( )
{   while( !isEmpty( ) )
        remove( first( ).retrieve( ) );
}
```

## 3.2.7. Examples

Radix Sort

288 371 260 531 287 235 56 299 18 23

260	531 371		23		235	56	287	18 288	299
-----	------------	--	----	--	-----	----	-----	-----------	-----

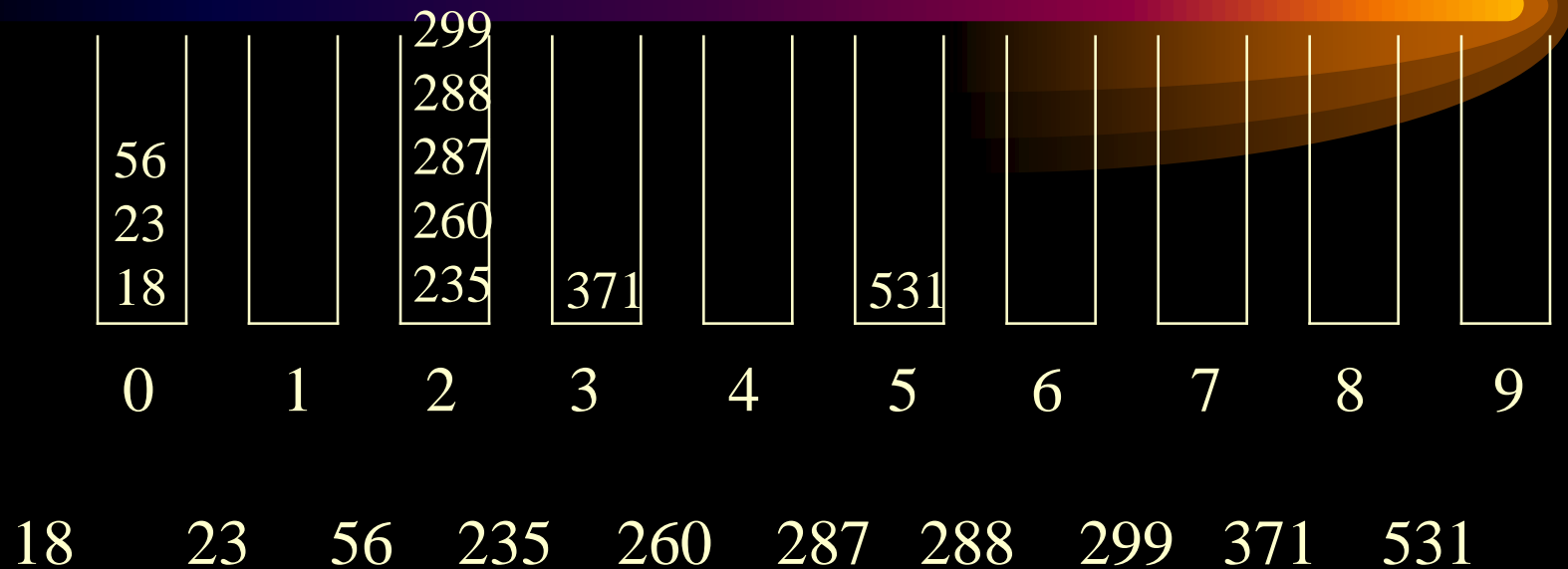
260 371 531 23 235 56 287 288 18 299

	18	23	235 531		56	260	371	288 287	299
--	----	----	------------	--	----	-----	-----	------------	-----

0 1 2 3 4 5 6 7 8 9

18 23 531 235 56 260 371 287 288 299

### 3.2.7. Examples



实现：原始要排序的数据、桶中的数据都可以用链表来实现。

## Chapter 3

### exercises:

1. Swap two adjacent elements by adjusting only the links(and not the data) using:
  - a. Singly linked lists.
  - b. Doubly linked lists.
2. Given two sorted lists  $L_1$  and  $L_2$ , write a procedure to compute  $L_1 \cap L_2$  using only the basic list operations.
3. Given two sorted lists,  $L_1$  and  $L_2$ , write a procedure to compute  $L_1 \cup L_2$  using only the basic list operations.
4. Write a nonrecursive method to reverse a singly linked List in  $O(N)$  time.

# Chapter 3

上机实习题：

3. 多项式相加， 用链表实现。
4. Josephus(  $n, m$ ), 用数组、链表实现。