Data Structures and Algorithms

Chapter 3 Linked List

Dr. Zhiqiang Liu

School of Software and Microelectronics, Northwest Polytechnical University

Outline

1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

Next Section

1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

3.1 Linked List

- A linked list stores a linear sequence of elements.
- Linked lists store elements in non-contiguous memory locations.

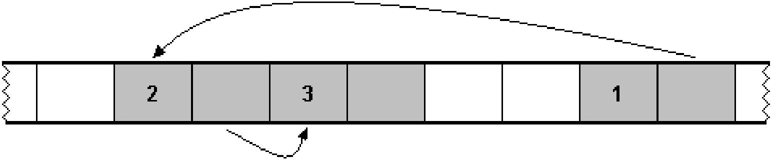
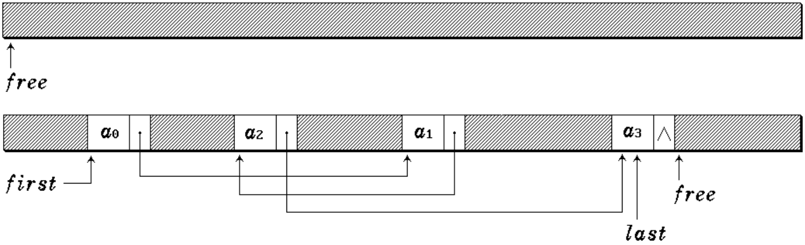


Figure 2 *A linked list in memory*

Next Subsection

1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

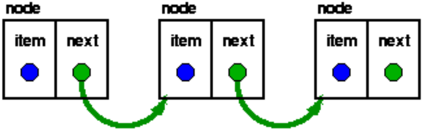
Single Linked list storage image



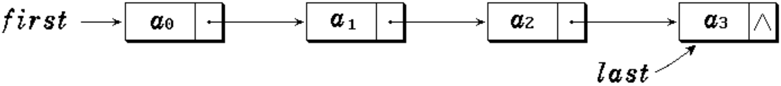
Linked lists store elements in non-contiguous memory locations

3.1.1 Singly Linked List

- Characteristics
  - Node representation



  - Linear structure



  - Extensible

Class definition of Single Linked List

- Class definition of Single Linked List: A linked list consists of:
  - ListNode Class
  - List Class
  - Iterator Class
- Definition
  - Compound definition
  - Nested definition

3.1.1 Singly Linked List

```
class List;              //Compound Class

class ListNode {      //ListNode Class
    //List class is its friend
    friend class List;
private:
    int data; //Node data
    ListNode *link;     //Pointer
};

class List { //List Class
public:
    ... ...
private:
    //head and tail pointer
    ListNode *first, *last;
};
```

3.1.1 Singly Linked List

```
class List { //List Class (Embedded definition)
public:
    //Operations
    ... ...
private:
    //Embedded ListNode Class
    class ListNode {
    public:
        int data;
        ListNode *link;
    };
    //Head and Tail pointer
    ListNode *first, *last;
};
```

Insert node into Linked List

- Three cases considered, First case:
  - Insert a node at beginning.

```
newnode->link = first;
first = newnode;
```

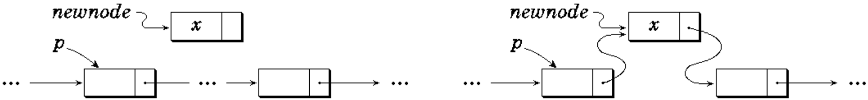
Insert node into Linked List

- Three cases considered, First case:
  - Insert a node at beginning.

```
newnode->link = first;
first = newnode;
```

- Three cases considered, Second case:
  Insert a node in middle of the list.

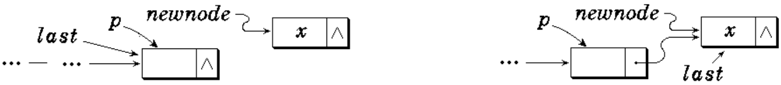
```
newnode->link = p->link;
p->link = newnode;
```



3.1.1 Singly Linked List

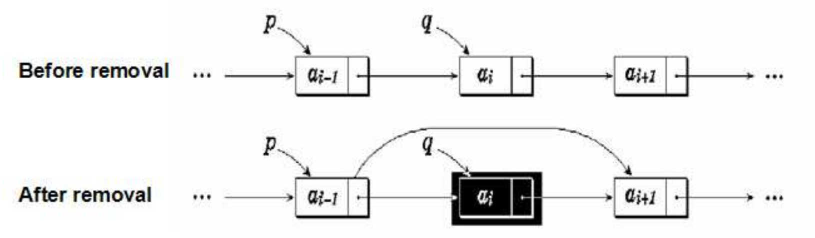
- Three cases considered, Third case:
  - Insert a node at end.

```
newnode->link = p->link;
p->link = last = newnode;
```



Remove node from Linked List

- Two cases considered
  - First case: Remove the first node of list;
  - Second case: Remove the node within the list or the last node.



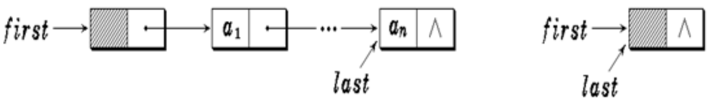
```
int List::Remove ( int i ) {
    //Remove i-th node of the list
    Node *p = first, *q;
    int k = 0;
    // Locate k and p (point to the node before the one
        will be removed)
    while ( p != NULL && k < i-1 )
    {
        p = p->link;
        k++;
    }    //Find out i-1-th node
    if ( p == NULL || p->link == NULL )
    {
        cout << "Invalid␣pos␣for␣removal!\n";
        return 0;
    }

    if ( i == 0 )
    {   //First case
        q = first;
        //Modify first  pointer
        p = first = first->link;
    }
    else
    {   //Second case
        q = p->link;
        p->link = q->link;
    }
    if ( q == last )
        last = p;   //Modify last pointer
    k = q->data;
    delete q;        //Release q
    return k;
}
```

Next Subsection

1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

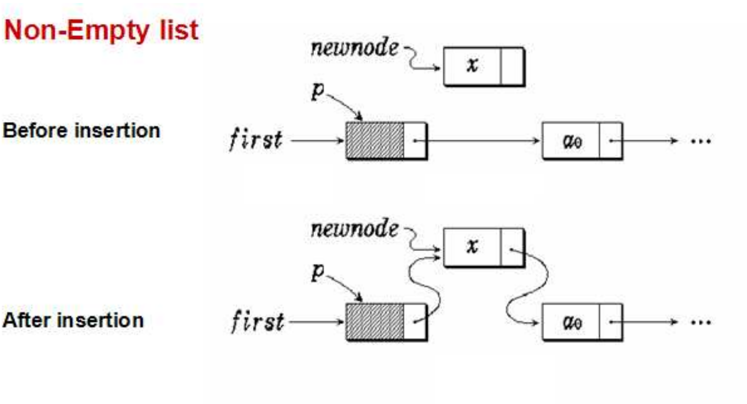
3.1.2 List with Dummy node

- Dummy node
  - at the beginning of the list
  - No data, just a indicator
- The aim of dummy node
  - Unify the operations of list, whether it is a Empty list or Non-empty list;
  - Simplify the operations of list

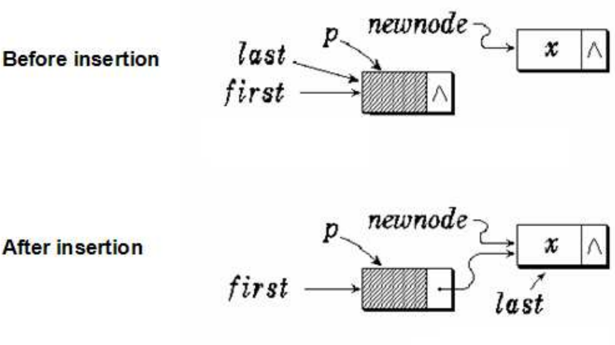


General Non-Empty List **Empty List**

Insertion for List with Dummy Node



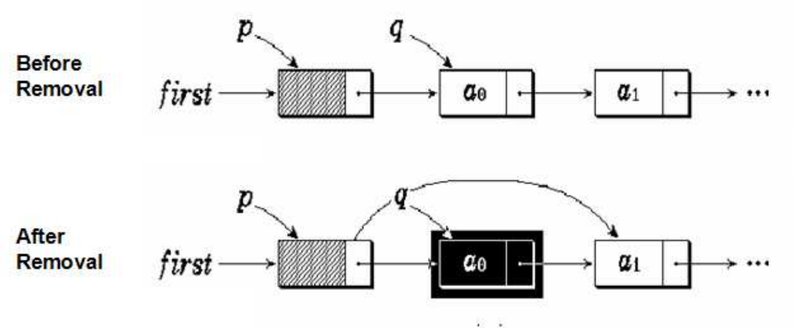
3.1.2 List with Dummy node



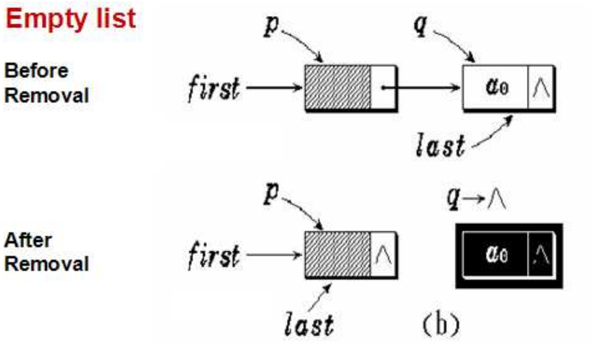
```
newnode->link = p->link;
if ( p->link == NULL ) last = newnode;
p->link = newnode;
```

3.1.2 List with Dummy node

Non-Empty list



3.1.2 List with Dummy node



Empty list

Before Removal

After Removal

```
q = p->link;
p->link = q->link;
delete q;
if(p->link == NULL)
    last = p;
```

Next Subsection

1 Linked List
- Singly Linked List
- List with Dummy node
- Class definition of Linked List Using Template
- Iterator Class of Linked List
- Static Linked List
- Circular list

2 Applications
- Joseph Problem
- Polynomial

3 Doubly Linked List (DblList)

4 Summary

5 QUIZ

```
template <class Type> class List;

template <class Type> class ListNode {
    friend class List<Type>;
    Type data;
    ListNode<Type> *link;
public:
    ListNode ( );
    ListNode ( const Type& item );
    //Get the address of next node(successor)
    ListNode<Type> *NextNode ( ) { return link; }
    //New a node with (item, next)
    ListNode<Type> *GetNode ( const Type&
        item, ListNode<Type> *next );
    //Insert p-node after the current node
    void InsertAfter ( ListNode<Type> *p );
    //Remove the node after current node
    ListNode<Type> *RemoveAfter ( );
};

```

```
template <class Type> class List {
    ListNode<Type> *first, *last;
public:
    List ( const Type & value ) {
        last =first = new ListNode<Type>( value );
    } //constructor
    ~List ( );
    void MakeEmpty ( );
    int Length ( ) const;
    ListNode<Type> *Find ( Type value );
    ListNode<Type> *Find ( int i );
    int Insert ( Type value, int i );
    Type *Remove ( int i );
    Type *Get ( int i );
};

//Constructor 1
template <class Type>
ListNode<Type> :: ListNode ( ) : link (NULL){ }

```

```
//Constructor 2
template <class Type>
ListNode<Type>::
ListNode( const Type & item ) :data (item), link ( NULL){ }

//New node
template <class Type>
ListNode<Type>  * ListNode<Type> ::
GetNode ( const Type & item,  ListNode<Type> *next = NULL )
{
    ListNode<Type> *newnode = new ListNode<Type> ( item );
    newnode->link = next;
    return newnode;
}
```

```
//Insert a node p after current node
template <class Type>
void ListNode <Type> :: InsertAfter ( ListNode<
    Type> *p )
{
    p->link = link;
    link = p;
}


//Remove the node after current node
template <class Type>
ListNode<Type>* ListNode<Type> :: RemoveAfter ( )
{
    ListNode<Type> *tempptr = link;
    if ( link == NULL ) return NULL;
    delete link;
    link = tempptr->link;
    return tempptr;
}
```

```
//Constructor (defined in the class declaration)
template <class Type>
List<Type> :: List ( const Type & value )
{
    last =first = new ListNode<Type>( value );
}

//Destructor
template <class Type>
List<Type> :: ~List ( )
{
    MakeEmpty ( );  delete first;
}

//Release the linked list
template <class Type>
void List<Type> :: MakeEmpty ( )
{
    ListNode<Type> *q;
    while ( first->link != NULL )
    {
        q = first->link;  first->link = q->link;
        delete q;
    }
    last = first;
}

//Get the number of the nodes
template <class Type>
int List<Type>::Length ( ) const {
    ListNode<Type> *p = first->link;
    int count = 0;
    while ( p != NULL ) {
        p = p->link;
        count++;
    }
    return count;
}


```

```
//Search a value in the list
template <class Type>
ListNode<Type>*List <Type>:: Find ( Type value ) {
    ListNode<Type> *p = first->link;

    while ( p != NULL && p->data != value )
        p = p->link;
    return p;
}

//Find out the i-th node, return ’its address
template <class Type>
ListNode<Type> *List<Type> :: Find ( int i ) {
    if ( i < -1 ) return NULL;
    if ( i == -1 ) return first;
    ListNode<Type> *p = first->link;
    int j = 0;
    while ( p != NULL && j < i )
      {
        p = p->link;
        j = j++;
    }
    return p;
}

//Insert a new node (value) before the i-th node in the list
template <class Type>
int List<Type> :: Insert ( Type value, int i ) {
    ListNode<Type> *p = Find ( i-1 );
    if ( p == NULL ) return 0;
    ListNode<Type> *newnode =
        GetNode ( value, p->link );
    if ( p->link == NULL )
        last = newnode;
    p->link = newnode;
    return 1;
}


```

```
//Remove the i-th node in the list
template <class Type>
Type *List<Type>::Remove ( int i ) {
    ListNode<Type> *p = Find (i-1), *q;
    if ( p == NULL || p->link == NULL )
        return NULL;
    q = p->link;
    p->link = q->link;
    Type value = new Type ( q->data );
    if ( q == last )
        last = p;
    delete q;
    return &value;
}

//Find out the i-th node, return ’its value
template <class Type>
Type *List<Type>::Get ( int i ) {
    ListNode<Type> *p = Find ( i );

    if ( p == NULL || p == first )
        return NULL;
    else
        return & p->data;
}
```

Next Subsection

1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

3.1.4 Iterator Class of Linked List

- The objective of iterator class
  - To search in the linked list
- Principal of iterator class
  - Friend class of ListNode and List classes
  - Iterator object can refer existing CList object
  - Point to the position of the current node in the list
  - Provide several methods for testing and searching

Template description of three Classes of Linked list

```
enum Boolean { False, True };
template <class Type> class List;
template <class Type> class ListIterator;

template <class Type> class ListNode {
friend class List <Type>;
friend class ListIterator <Type>;
public:
    ... ...
private:
    Type data;
    ListNode<Type> *link;
};
```

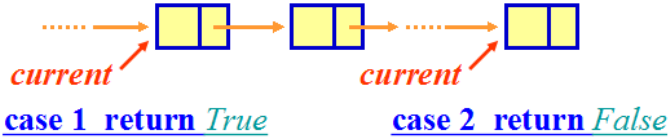
3.1.4 Iterator Class of Linked List

```
template <class Type> class  ListIterator {
public:
    ListIterator ( const List<Type> & l )
: list ( l ), current ( l.first ) { }
    Boolean NotNull ( );

    Boolean NextNotNull ( );

    ListNode <Type> *First ( );
    ListNode <Type> *Next ( );
private:
    const List<Type> & list;
    ListNode<Type> *current;
}
```

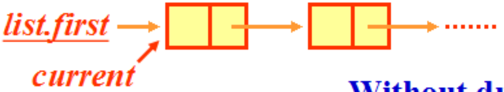
Implementation of methods of Iterator Class

```
//Check whether the current node is NULL or not template <class Type>
Boolean ListIterator<Type> :: NotNull ( ) {
     if ( current != NULL ) return True;
     else return False;
}
```

```
//Check whether the next node of the current
//node is NULL or not
template <class Type>
Boolean ListIterator<Type>::NextNotNull ( ) {
    if ( current != NULL &&
        current->link != NULL ) return True;
    else return False;
}
```

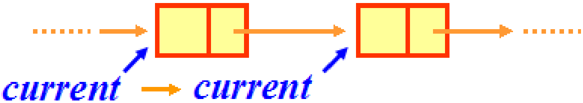


```
//return the address of the first node
template <class Type>
ListNode<Type>* ListIterator<Type> :: First ( ) {
    if ( list.first != NULL ){
        current = list.first;
        return current;
    }
    else { current = NULL;  return NULL; }
}
```



Without dummy node

```
template <class Type>
ListNode<Type>* ListIterator<Type> :: Next ( ) {
     if ( current != NULL
             && current->link != NULL ) {
         current = current->link;
         return current;
     }
     else { current = NULL;  return NULL; }
}
```



Example: compute the sum of elements of the list using iterator class

```
int sum ( const List<int> &l )
{
    ListIterator<int> li ( l );
    if ( ! li.NotNull () )
        return 0;
    int retval =  li.First()->getData();
    while ( li.nextNotNull () )
        retval +=  li.Next()->getData();

    return retval;
}
```

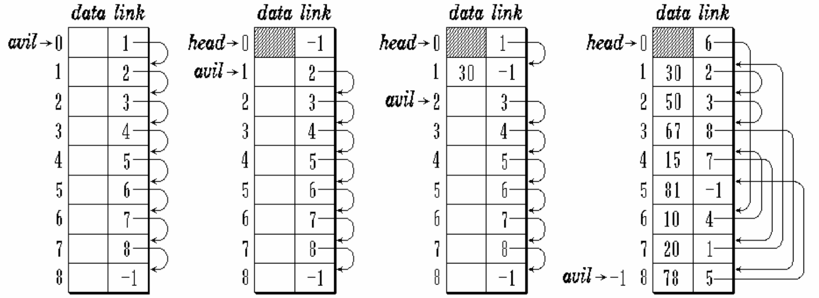
Next Subsection

1. Linked List
    - Singly Linked List
    - List with Dummy node
    - Class definition of Linked List Using Template
    - Iterator Class of Linked List
    - Static Linked List
    - Circular list
2. Applications
    - Joseph Problem
    - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

3.1.5 Static Linked List

Defined by array

Store space is invariable



		data	link
avil →	0		1
	1		2
	2		3
	3		4
	4		5
	5		6
	6		7
	7		8
	8		−1

		data	link
head →	0	▨	−1
avil →	1		2
	2		3
	3		4
	4		5
	5		6
	6		7
	7		8
	8		−1

		data	link
head →	0	▨	1
	1	30	−1
avil →	2		3
	3		4
	4		5
	5		6
	6		7
	7		8
	8		−1

		data	link
head →	0	▨	6
	1	30	2
	2	50	3
	3	67	8
	4	15	7
	5	81	−1
	6	10	4
	7	20	1
avil → −1	8	78	5

New node j = avil; avil = A[avil].link;

Release A[i].link = avil; avil = i;

Next Subsection

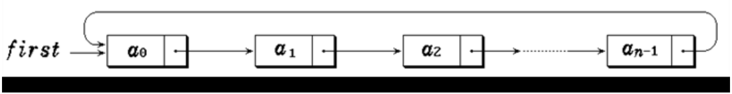
1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

3.1.6 Circular list

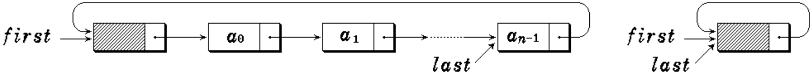
- Circular list is an advanced singly linked list
  - The link field of the last node of the list is not 0 yet, pointing to the head of list;
  - If we know the position of arbitrary node of the circular list, we could access all the nodes one by one.
- Circular list has two kinds of representation
  - Without Dummy node
  - With dummy node

Example

- An example of circular list



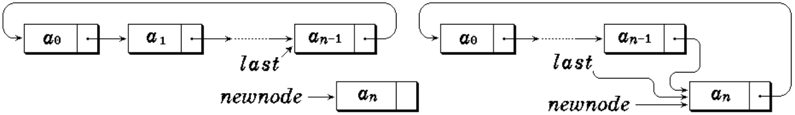
- Circular list with DN



Class of circular list

```
template <class Type> class CircList;

template <class Type> class CircListNode {
    friend class CircList;
public:
    CircListNode ( Type d = 0,
        CircListNode<Type> *next = NULL ) :
    data ( d ), link ( next ) { }
private:
    Type data;
    CircListNode<Type> *link;
}

template <class Type> class CircList {
public:
    CircList ( Type value );
    ~CircList ( );
    int Length ( ) const;
    Boolean IsEmpty ( )
    { return first->link == first; }
    Boolean Find ( const Type & value );
    Type getData ( ) const;
    void Firster ( ) { current = first; }
    Boolean First ( );
    Boolean Next ( );
    Boolean Prior ( );
    void Insert ( const Type & value );
    void Remove ( );
private:
    CircListNode<Type> *first, *current, *last;
};
```

3.1.6 Circular list



Next Section

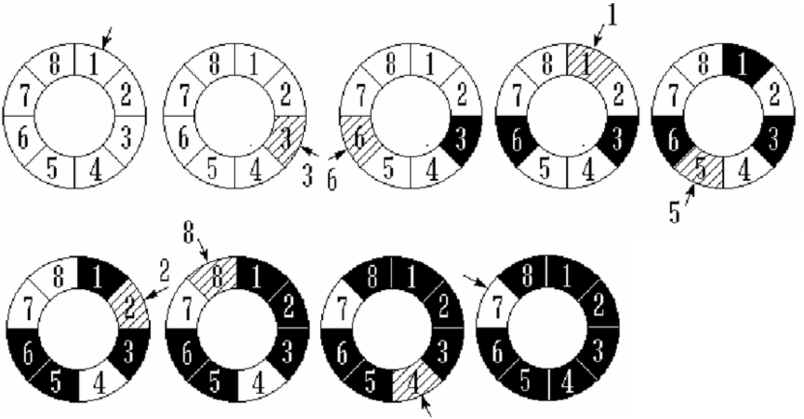
1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

Next Subsection

1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

3.2.1 Joseph Problem

- Joseph Problem: m=3, n=8



```
#include <iostream.h>
#include “CircList.”h
Template<Type> void CircList<Type>
:: Josephus ( int n, int m ) {
    Firster ( );
    for ( int i = 0; i < n-1; i++ ) {
        for ( int j = 0; j < m-1; j++ )
            Next ( );
        cout << "The␣out␣one␣is"
            << getData ( )  << endl;
        Remove ( );
    }
}

void main ( ) {
    CircList<int> clist;
    int n, m;
    cout << "Enter␣the␣Number␣of␣Contestants?";
    cin >> n >> m;
    //Construct Joseph circle
    for ( int i=1; i<=n; i++ )
```

```
        clist.insert (i);
    //Call Joseph function
    clist.Josephus (n, m);
}
```

Next Subsection

1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

3.2.2 Polynomial

- Polynomial

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$
$$= \sum_{i=0}^{n} a_i x^i$$

- Node:

$data = Term$

$coef$	exp	$link$

Class description of polynomial using Linked List

```
struct Term {
    int coef;
    int exp;
    Term ( int c, int e ) { coef = c;  exp = e; }
};

class Polynomial    {
    List<Term> poly;
    friend Polynomial & operator +
          ( Polynomial &, Polynomial &);
};
```

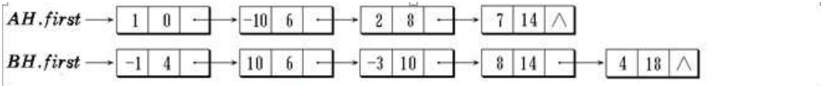
Polynomial Addition

$$AH = 1 - 10x^{6} + 2x^{8} + 7x^{14}$$

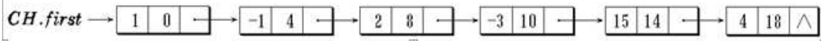
$$BH = -x^{4} + 10x^{6} - 3x^{10} + 8x^{14} + 4x^{18}$$

$$CH = AH + BH$$

$$= 1 - x^{4} + 2x^{6} - 3x^{10} + 15x^{14} + 4x^{18}$$



Before addition



After addition

```
 Polynomial & operator + ( Polynomial & ah,
               Polynomial & bh )
{
    ListNode<Term> *pa, *pb, *pc, *p;
    // Aiter, Biter
    ListIterator<Term> Aiter ( ah.poly );
    ListIterator<Term> Biter ( bh.poly );
    // pa, pb
    pa = pc = Aiter.First ( );   // ah
    pb = p = Biter.First ( );     // bh
    pa = Aiter.Next ( );
    pb = Biter.Next ( );
    delete p;
```

```
while ( Aiter.NotNull ( ) && Biter.NotNull ( ) )
    switch ( compare ( pa$\to$exp, pb$\to$exp ) ) {
    case ‘ = ’ :
        pa->coef = pa->coef + pb->coef;
        p = pb;   pb = Biter.Next ( );  delete p;
        if ( !pa->coef ) {
            p = pa;  pa = Aiter.Next ( );
            delete p;
        }
        else {
            pc->link = pa;   pc = pa;
            pa = Aiter.Next ( );
        }
         break;
    case ‘ > ’ :      // pa->exp> pb->exp
         pc->link = pb;   pc = pb;
         pb = Biter.Next ( );   break;
   case ‘ < ’ :       // pa->exp< pb->exp
      pc->link = pa;   pc = pa;
      pa = Aiter.Next ( );
   }
```

```
  if ( Aiter.NotNull ( ) )
      pc->link = pa;
  else
      pc->link = pb;
}
```

Next Section

1 Linked List
- Singly Linked List
- List with Dummy node
- Class definition of Linked List Using Template
- Iterator Class of Linked List
- Static Linked List
- Circular list

2 Applications
- Joseph Problem
- Polynomial

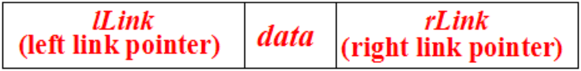
3 Doubly Linked List (DblList)

4 Summary

5 QUIZ

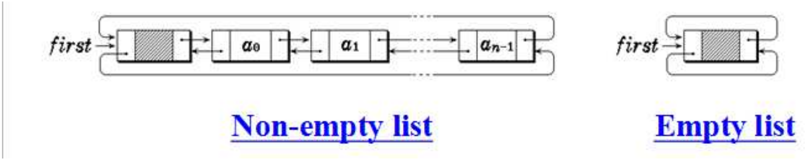
3.3 Doubly Linked List (DblList)

- Drawback of singly linked list
  - Can't visit the predecessor of the current node
- Improved criteria
  - Besides the existed successor linked pointer, add another pointer which used to point to the predecessor node



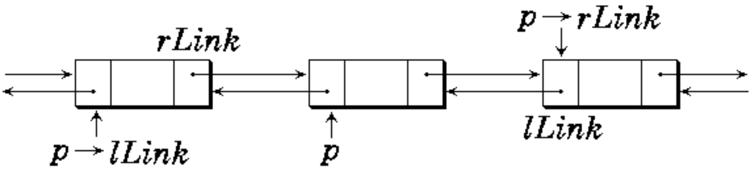
lLink (left link pointer)	*data*	*rLink* (right link pointer)

3.3 Doubly Linked List (DblList)



- Relations between the predecessor, current node and successor

```
p==p->lLink->rLink==p->rLink->lLink
```

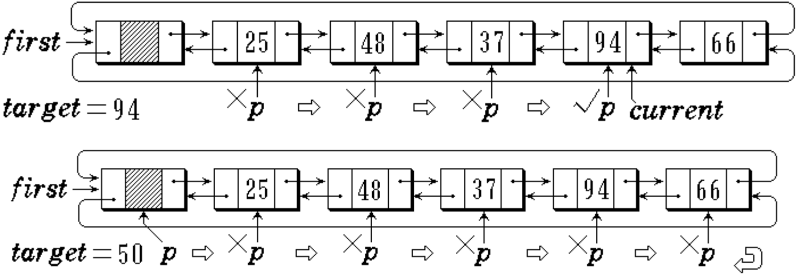


Class of DblList

```
template <class Type> class DblList;
template <class Type> class DblNode {
friend class DblList<Type>;
private:
    Type data;                      //data
    DblNode<Type> *lLink, *rLink;   //pointer

DblNode(Type value, DblNode<Type> *left,
    DblNode<Type> *right):data (value), lLink (left),
        rLink (right)
   { }
DblNode ( Type value ) : data (value),
          lLink (NULL), rLink (NULL){ }
};
```

```
template <class Type> class DblList {
public:
  DblLIst ( Type uniqueVal );
  ~DblList ( );
  int Length ( ) const;
  int IsEmpty ( ) { return first->rlink == first; }
  int Find ( const Type & target );
  Type getData ( ) const;
  void Firster ( ) { current = first; }
  int First ( );
  int Next ( );
  int Prior ( );
  int operator!(){ return current != NULL; }
  void Insert ( const Type & value );
  void Remove ( );
private:
  DblNode<Type> *first, *current;
};
```

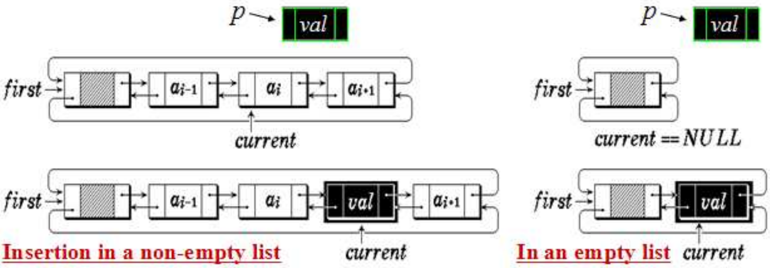
Searching in DblList



```
//Find out the target in the DbL-List

template <class Type>
int DblList<Type>::Find ( const Type & target ) {
    //Return 1 if success, otherwise return 0
    DblNode<Type> *p = first->rLink;
    while ( p != first && p->data != target )
      p = p->rLink;
    if ( p != first ) { current = p;  return 1; }
    return 0;
}
```

Insertion after current node in DblList

```
p->rLink =current->rLink;
current->rLink = p;
current->rLink->Link = current;
current = current->rLink;
current->rLink->Link = current;
```



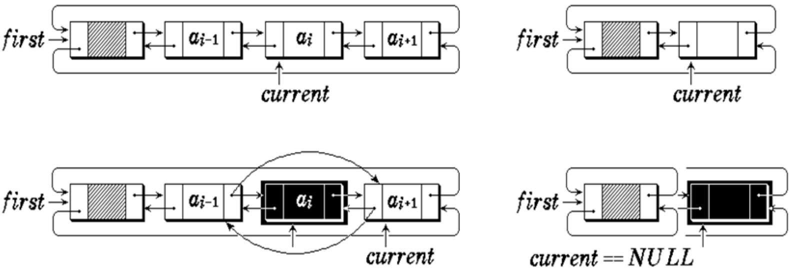
Insertion in a non-empty list

In an empty list

```
//Insert a node (value) after the current node
template <class Type>
void DblList<Type>::Insert ( const Type & value ) {
     if ( current == NULL )
         current = first->rLink =
         new DblNode ( value, first, first );
     else {
         current->rLink =new DblNode
        ( value, current, current->rLink );
     current = current->rLink;
     }
     current->rLink->lLink = current;
}
```

Node removal in DblList

```
current->rLink->lLink=current->lLink;
current->lLink->rLink=current->rLink;
current=current->rLink;
```



```
//Remove the current node
template <class Type>
void DblList<Type>::Remove ( ) {
    if ( current != NULL ) {
        DblNode *temp = current;
        current = current->rLink;
        current->lLink = temp->lLink;
        temp->lLink->rLink = current;
        delete temp;
        if ( current == first )
           if ( IsEmpty ( ) )
                current = NULL;
           else
                current = current->rLink;
    }
}
```

```
//Constructor

template <class Type>
DblList<Type>::DblLIst ( Type uniqueVal ) {
     first = new DblNode<Type> ( uniqueVal );
     first->rLink = first->lLink = first;
     current = NULL;
}
```

```
//Get the number of the nodes in the DbL List

template <class Type>
int DblList<Type>::Length ( ) const {
    DblNode<Type> * p = first->rLink;
    int count = 0;
    while ( p != first )
        { p = p->rLink;  count++; }
    return count;
}
```

```
//Move the current pointer to the first node

template <class Type>
int DblList<Type>::First ( ) {
    if ( !IsEmpty ( ) )
         { current = first->rLink;  return 1; }
    current = NULL;
    return 0;
}
```

```
//Move the current pointer to the next node

template <class Type>
int DblList<Type>::Next ( ) {
    if ( current->rLink == first )
     { current = NULL;  return 0; }
    current = current->rLink;
    return 1;
}
```

```
//Move the current pointer to the prior node

template <class Type>
int DblList<Type>::Prior ( ) {
     if ( current->lLink == first )
     { current = NULL;  return 0; }
     current = current->lLink;
     return 1;
}
```

Next Section

1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

3.4 Summary

1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

Next Section

1. Linked List
   - Singly Linked List
   - List with Dummy node
   - Class definition of Linked List Using Template
   - Iterator Class of Linked List
   - Static Linked List
   - Circular list
2. Applications
   - Joseph Problem
   - Polynomial
3. Doubly Linked List (DblList)
4. Summary
5. QUIZ

3.5 QUIZ and homework

1、设计一个算法，从顺序表中删除其值在s和t（包含相等）之间的所有元素，若顺序表为空，则显示出错信息并退出运行。

```
template <class DataType> int deleteNo_stot(
    SeqList &L, DataType s, DataType t)
{}
```

2、设有一个表头指针为h的单链表。设计一个算法，通过遍历一趟链表，将连表中所有节点的链接方向逆转。

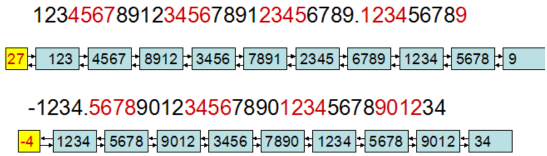
```
void Reverse(LinkNode *h)
{}
```

3.5 QUIZ and homework

3、试设计一个算法，改造一个带表头结点的双向循环链表，所有结点的原有次序保持在各个结点的右链域rLink中，并利用左链域lLink把所有结点按照其值从小到大的顺序连接起来。

```
typedef struct DblNode{
    int data;  struct DblNode *lLink, rLink;
} DblNode;
typdef DblNode* DblList;
void SortedList(DblList dblist){}
```

- 4、 For a large number, how to represent it and implement the $+, -, *, /$ operations? For example:



123456789123456789123456789.123456789

27 | 123 | 4567 | 8912 | 3456 | 7891 | 2345 | 6789 | 1234 | 5678 | 9

-1234.567890123456789012345678901234

-4 | 1234 | 5678 | 9012 | 3456 | 7890 | 1234 | 5678 | 9012 | 34