國立清華大學 電機工程學系

EE2410 Data Structure

Chapter 4

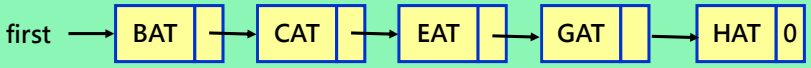
Linked List (Part I)

Outline

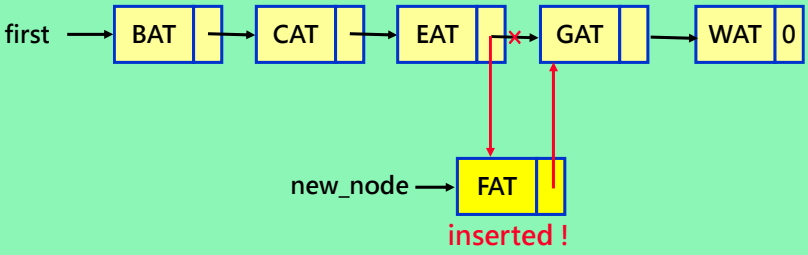
- **Singly Linked Lists**
- **A Reusable Linked List Class**
- **Circular Lists**
- **Linked Stacks and Queues**
- **Polynomials**

Why Linked List?

(Initial Linked List)

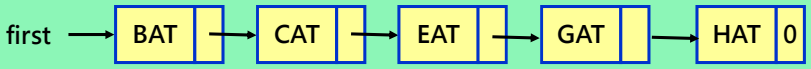


(Quick Insertion of a new node)

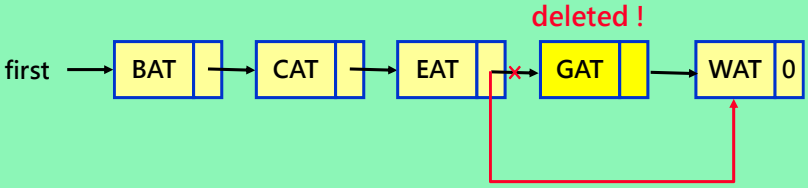


Deletion of A Node

(Initial Linked List)

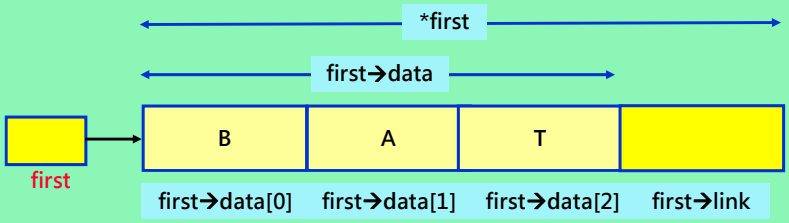


(Quick deletion of a new node)



Ex: Access The Data Of a Node

```
class ThreeLetterNode {
private:
      char  data[3];
      ThreeLetterNode *link;
};
main(){
   …
   ThreeLetterNode *first;  /* The data references are shown below
   …
}
```



Illegal access because data[] and link are private data !

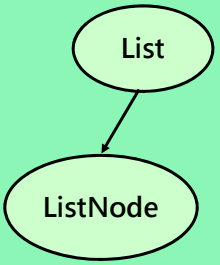
Dilemma of Node Data

- **Declaring the node data as public**
  - **will allow one to access the data through pointer**
  - **but the data encapsulation principle is violated**
- **Declaring the node data as private**
  - **will requires another member functions to access the data**
  - **E.g., Get_link(), Set_link(), Get_data(), Set_data()**
  - **The access is less efficient**

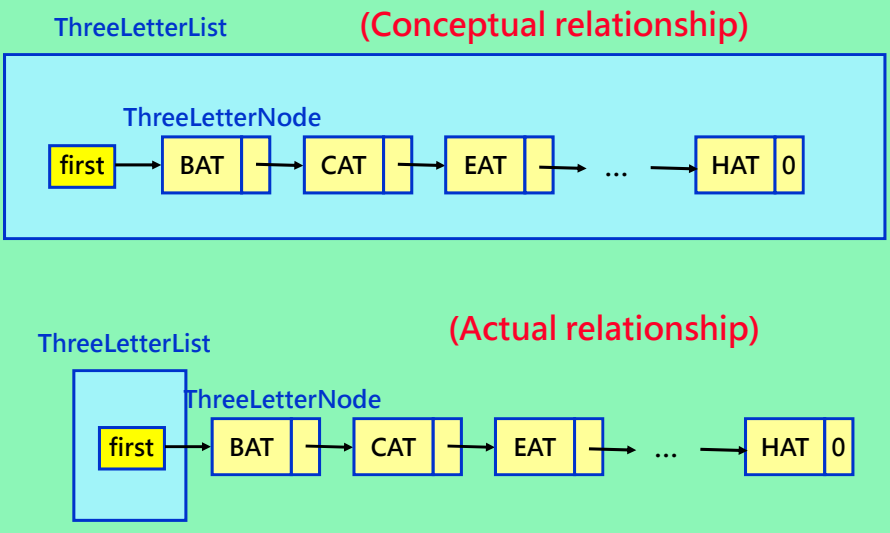
Composite Class

- **Has-A Relationship**
  - **We say that a data object of Type A HAS-A data object of Type B if A conceptually contains B or B is part of A**

```
class ThreeLetterList; // forward declaration
class ThreeLetterNode{
friend class ThreeLetterList;
private:
      char  data[3];
      ThreeLetterNode  *link;
};
class ThreeLetterList {
public:
      // List Manipulation operations
…
private:
     ThreeLetterNode  *first;
};
```



Relationship of List and Node



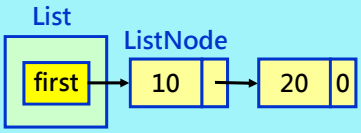
Nested Class

ThreeLetterNode is defined as an inner class of ThreeLetterList
The data of ThreeLetterNode are declared as public

```
class ThreeLetterList {
public:
      // List Manipulation operations
  …
private:
      class ThreeLetterNode {
      public:
            char  data[3];
            ThreeLetterNode  *link;
      };
      ThreeLetterNode  *first;
};
```

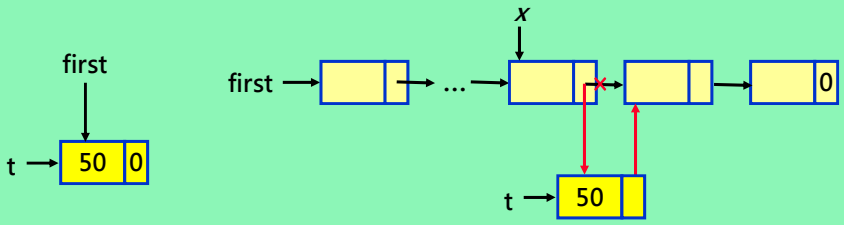

Example: Linked List Creation

```
class ListNode {
private:
   int   data;
   ListNode  *link;
}

void  List::Create2()
{
   first = new ListNode(10); // create and initialize first node
   // create and initialize second node and place its address in first→link
   first→link = new ListNode(20);
}

ListNode::ListNode(int element=0) // 0 is the default argument in constructor
{
   data = element;
   link = 0; // null pointer constant
}
```



Example: Linked List Insertion

```
void  List::Insert50(ListNode *x)
{
   ListNode  *t = new ListNode (50); // create and initialize new node
   if( ! first ) // insert into empty list
   {
      first = t;
      return; // exit List::Insert50
   }
   // insert after x
   t → link = x → link;
   x → link = t;
}
```



Outline

- **Singly Linked Lists**
- **A Reusable Linked List Class**
- **Circular Lists**
- **Linked Stacks and Queues**
- **Polynomials**

Template Definition of Linked List

```
template <class Type>  class List; // forward declaration

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

template  <class Type>
class   List {
public:
     List() { first = 0; };  // constructor initializing first to 0
     // list manipulation operations
     …
private:
    ListNode<Type>  *first;
};
```

A linked list of integers declared as:
List<int> intlist;

Direct Traversal of a List

pseudo-code

```
1.    // initialize a container C
2.    int  x = -MAXINT;
3.    for each item in Container C
4.    {
5.        current = current item of C;
6.        x = max(current, x);  // body
7.    }
8.    return (x);  // post-processing step
```

Direct Traversal

```
(Revision of statements 3 to 7 for integer container)
for ( ListNode<int> *ptr = first;    ptr !=0;    ptr = ptr → link )
{
   current = ptr → data;
   x = max(current, x);
}
```

in a member of List<Type>

Drawbacks of Direct Traversal of a Container Class

- **In a template class like List<Type>**
  - **Operations should be independent of the type, while direct traversal depends on the type of elements**
  - **For example, it does not make sense to compute the sum of a Rectangle container**
- **A new function requires traversal**
  - **of a container class needs the support of a new class member function**
  - **This is difficult because the class provider and class users might be different in a programming team**
- **Even if class user is allowed to add a new member function**
  - **He or she would need to know how the container is implemented**

Linked List Iterator

- **An Iterator**
  - **is an object that is used to traverse all the elements of a container class C**
  - **useful in operations like**
    - **(1) print all integers in C**
    - **(2) Obtain the maximum, minimum, mean, or median of all integers in C**
    - **(3) Obtain the sum, product, or sum of squares of all integers in C**
    - **(4) Obtain all integers in C that satisfy some property P (e.g., integers that are positive, or the square of an integer, etc.)**
    - **(5) Obtain the integer *x* from C such that, for some function *f*, *f*(*x*) is the maximum**

Linked List With Iterator (I)

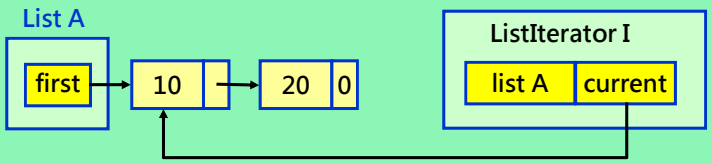
```
enum  Boolean  { FALSE, TRUE };
template <class Type> class List;
template <class Type> class ListIterator;

template <class Type> class ListNode{
friend class List<Type>;
friend class ListIterator<Type>;
private:
     Type  data;
     ListNode  *link;
};
template  <class Type> class List {
friend class ListIterator<Type>;
public:
     List() { first = 0; };  // constructor initializing first to 0
     // list manipulation operations  …
private:
    ListNode<Type>  *first;
};
TO BE CONTINUED
```

兩個朋友

一個朋友

Linked List With Iterator (II)

```
template <class Type> class ListIterator {
public:
     ListIterator( const List<Type>& l):  list(l), current (l.first) {};
     Boolean  NotNull();
     Boolean NextNotNull();
     Type  *First();
     Type  *Next();
private:
     const  List<Type>& list;      // refers to an existing list
     ListNode <Type>* current;  // points to a node in list
};
```

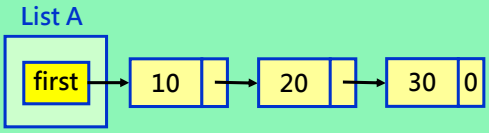


List Iterator Functions

```
template <class Type> // check that the current element in List is non-null
Boolean ListIterator<Type>::NotNull() {
   if ( current ) return TRUE;   else  return FALSE;
}
template <class Type> // check that the next element in List is non-null
Boolean ListIterator<Type>::NextNotNull() {
   if ( current && current → link)  return TRUE;   else  return FALSE;
}
template <class Type> // return a pointer to the first element of List
Type *ListIterator<Type>::First() {
   if ( list.first )  return (&list.first→data);   else  return 0;
}
template <class Type> // return a pointer to the next element of List
Type *ListIterator<Type>::Next()
   if ( current ) {
       current = current → link;
       if ( current )  return (&current→data);
  }
  else  return 0;
}
```


Example: Usage of Iterator

```
int  sum( const List<int>& input_list)
{
   ListIterator<int>  l (input_list); // l is associated with list input_list
   if( ! l.NotNull())  return 0;  // return 0 if the list is empty

   int  ret_value = *l.First();       // get the first element's pointer
   while ( l.NextNotNull() ) {      // iteratively sum up every element's value
           ret_value += *l.Next(); // get it, add it to the current total
   }
   return (ret_value);
}
```



A Stylish Foreach Macro

- **Assume List, ListNode, and ListIterator**
  - **have been defined as above**
- **Variables**
  - **list: an object of List**
  - **data_ptr: an object of ListNode's data pointer**
  - **gen: an object of ListIterator for list**

```
#define List_foreach(gen, node)   \
        for ( node = gen.First();   \
              gen.NotNull();        \
              node = gen.Next();    \
        )
```

Macro Definition

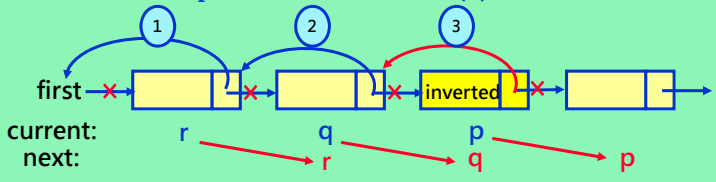
```
sum = 0;
ListIterator  gen(list);
List_foreach(gen, node){
    sum += *node;
}
```

Usage of List_foreach

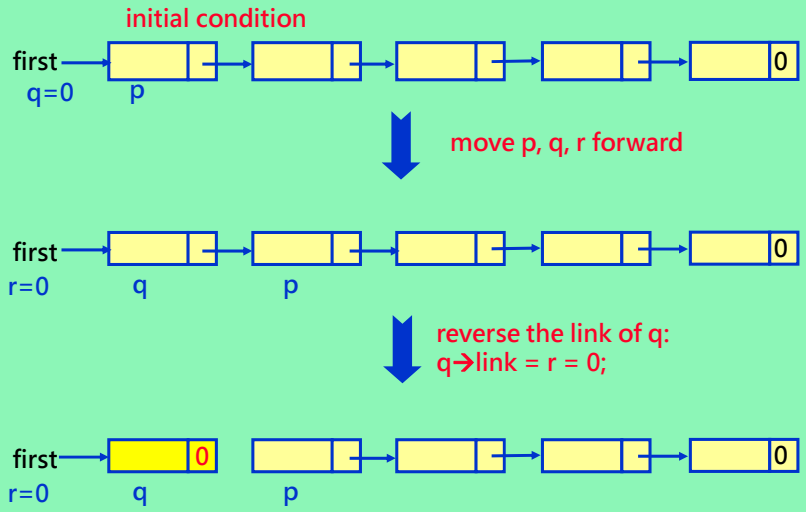
Inverting a Linked List (or Chain)

```
template <class Type> void List<Type>::Invert()
// A chain x is inverted so that if x = (a1, a2, …, an)
// then after execution, x = (an, an-1, …, a1)
{
   ListNode<Type>  *p = first;  *q = 0; // q trails p
   while(p) {
      ListNode<Type>  *r = q;   q = p; // r trails q
      p = p → link;  // remember the next node of the node being inverted q
      q → link = r;  // link q to the preceding node r
   }
   first = q;
}
```

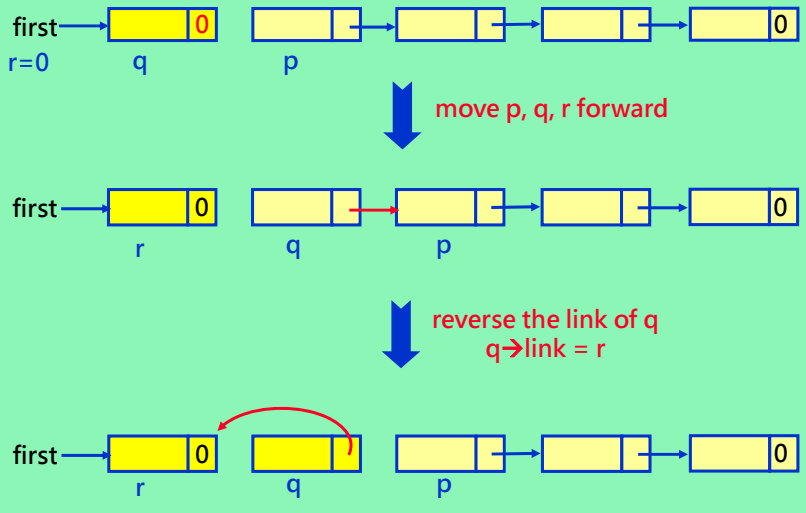
- **Inverting**
  - **can be done "in place" in linear time O(n)**



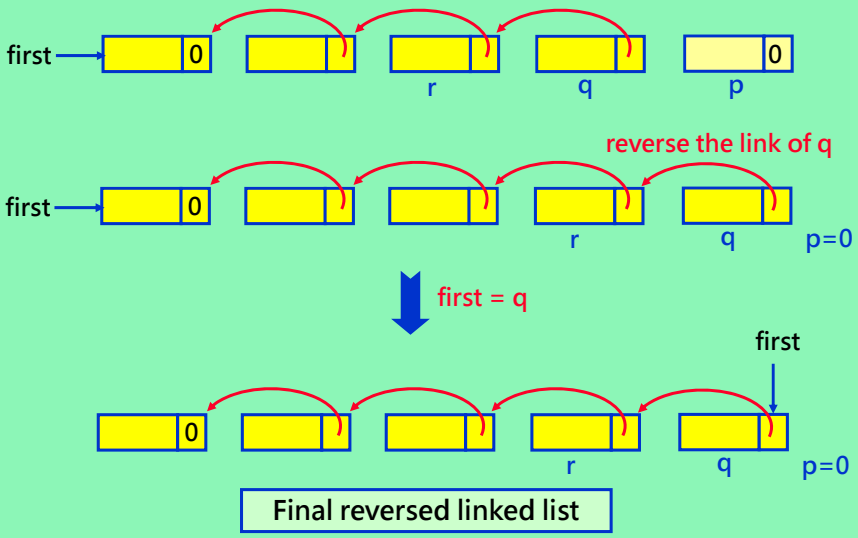
Process of Inverting a List



Process of Inverting a List



Process of Inverting a List

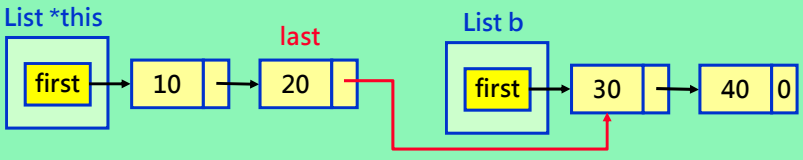


Concatenating Two Chains

- **The complexity is also linear**

```
template <class Type>
void  List<Type>::Concatenate ( List<Type> b)
// this = (a1, a2, …, an) and b = (b1, b2, …, bm) m, n ≧ 0
// produces the new chain z = (a1, a2, …, an, b1, b2, …, bm) in this
{
  if ( ! first ) {  first = b.first;  return; }
  if ( b.first ) {  // finding the last node of *this
      for (ListNode<Type> *p = first;  p→link;   p = p → link);  // no body
      p → link = b.first;
  }
}
```

false when p is the last in *this



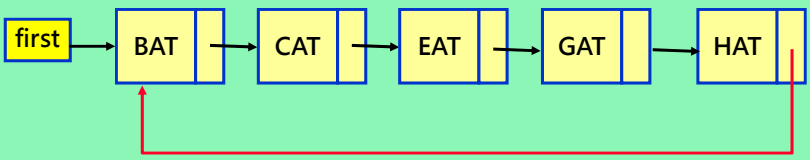
Outline

- Singly Linked Lists
- A Reusable Linked List Class
- Circular Lists
- Linked Stacks and Queues
- Polynomials

Basics of Circular List

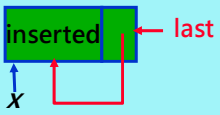
- **Major Features**
  - The link field of the last element points to the first element
  - Check if last element:

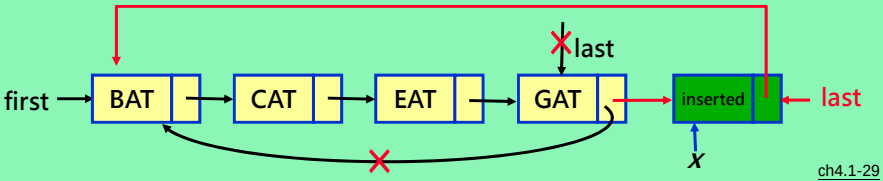
    (current → link == first) instead of (current → link = 0)



Insertion-To-Rear Function

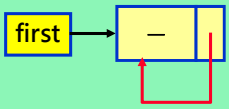
```
template <class Type>
void  CircularList::Insert_to_Rear( ListNode<Type> *x)
// insert the node pointed at by x at the rear of the circular
// list *this, where last points to the last node in the list
{
  if ( ! last ) {   // empty list
      last = x;  x → link = x;
  }
  else {
      x → link = last → link;
      last → link = x;
      last = x;
  }
}
```

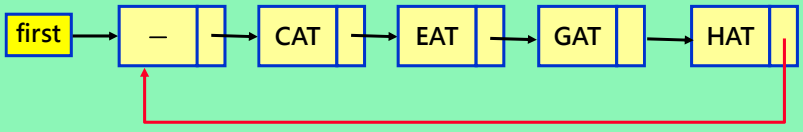




Dummy Head Node

- **In some applications**
  - **using simple circular list structure cause problems as the empty list has to be handled as a special case**
- **To avoid such as special cases**
  - **a dummy head node is introduced**

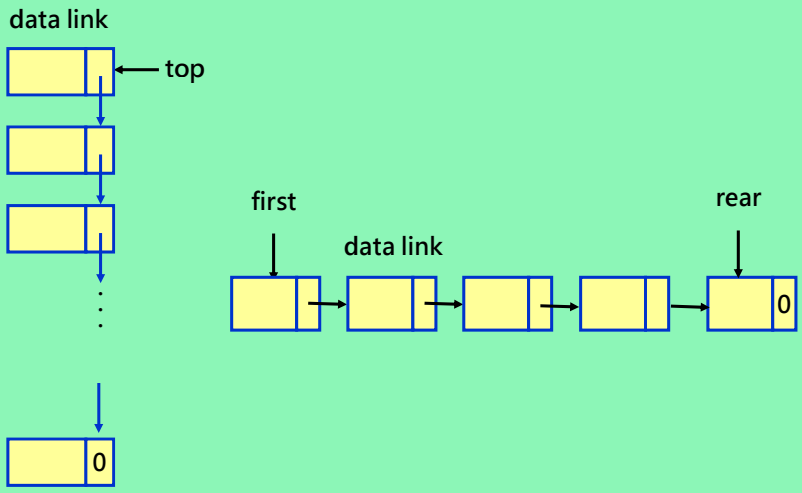




Outline

- **Singly Linked Lists**
- **A Reusable Linked List Class**
- **Circular Lists**
- **Linked Stacks and Queues**
- **Polynomials**

Linked Stack and Queue



Stack Class Definition

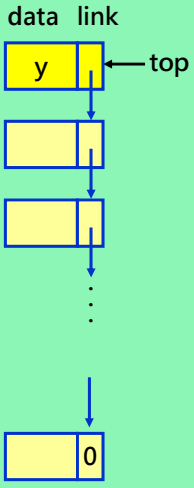
```
class  Stack;  // forward declaration

class StackNode {
friend  class  Stack;
private:
   int   data;
   StackNode  *link;
   StackNode ( int d = 0, StackNode *l = 0) : data (d), link (l) {} ; // constructor
};

class Stack {
public:
  Stack() { top = 0; } ; // constructor
  void  Add(const int);
  int* Delete (int&);
private:
  StackNode  *top;
  void  StackEmpty();
};
```


Stack Add and Delete

```
void  Stack::Add ( const int y) {
   top = new StackNode (y, top);
}
```

old top

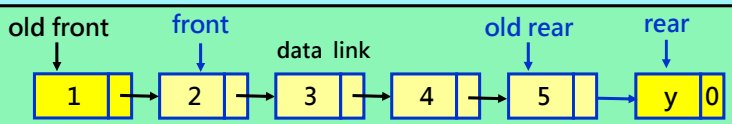
```
int  *Stack::Delete ( int& retvalue)
// Delete top node from stack and return a pointer to its data
{
   if (top == 0) {  StackEmpty(); return 0; }
    // return null pointer constant
   StackNode  *delnode = top;
   retvalue = top → data; // get data field of top node
   top = top→link;          // update the top pointer
   delete  delnode;         // free the node
   return   &retvalue;      // return pointer to data
}
```



Queue Add and Delete

```
void  Queue::Add ( const int y)
{
   if ( front == 0 )  front = rear = new  QueueNode(y, 0); // empty queue
   else   rear = rear → link = new  QueueNode(y, 0);
   // attach node and update rear
}
```

```
int  *Queue::Delete ( int& retvalue)
// Delete the first node in queue and return a pointer to its data
{
   if (front == 0) {  QueueEmpty(); return 0; } // return null pointer constant
   QueueNode  *x = front;
   retvalue = front → data; // get data
   front = x → link;          // update front node
   delete  x;                 // free the node
   return   &retvalue;  // return pointer to data
}
```



Outline

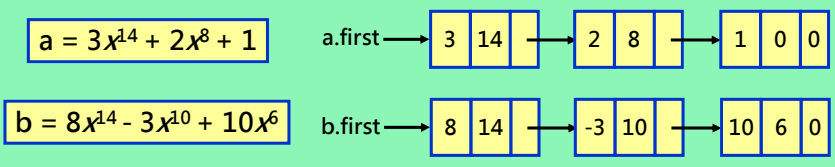
- **Singly Linked Lists**
- **A Reusable Linked List Class**
- **Circular Lists**
- **Linked Stacks and Queues**
- **Polynomials**

Is-Implemented-By Relationship

- **Definition**
  - **A data object of Type A IS-IMPLEMENTED-IN-TERMS-OF a data object of Type B if the Type B object is central to the implementation of Type A object.**
  - **This relationship is usually expressed by declaring the Type B object as a data member of the Type A object**
- **Next: Polynomial implemented by linked list**

Polynomial Class

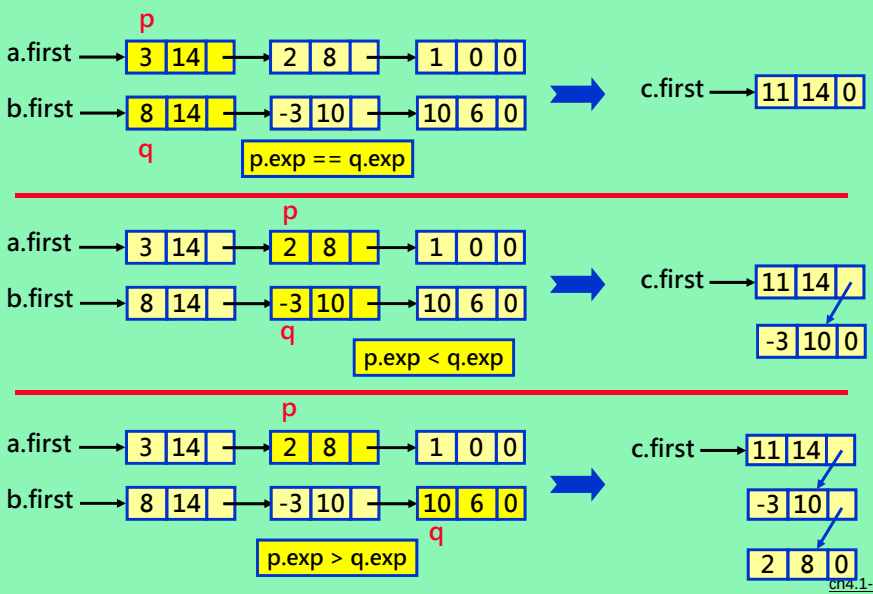
```
struct  Term
// all members of Term are public by default
{
   int   coef; // coefficient
   int   exp;  // exponent
   void  Init ( int c, int e ) {  coef = c;  exp = e; } ;
};
class Polynomial
{
friend  Polynomial  operator+( const Polynomial&, const Polynomial&);
private:
     List<Term> poly;
};
```



```
1 Polynomial operator+(const Polynomial& a , const Polynomial& b)  {
2 // Polynomials a and b are added and the sum returned
3       Term *p, *q, temp ;
4       ListIterator <Element > Aiter (a.poly) ; ListIterator <Element > Biter (b.poly) ;
5       Polynomial c ;
6       p = Aiter. First () ; q = Biter. First () ; // get first node in a and b
7       while (Aiter . NotNull () && Biter . NotNull ()) { // current node is not null
8             switch (compare(p →exp,q →exp)) {
9                   case '=':
10                        int x = p →coef + q →coef ; temp . Init (x,q →exp) ;
11                        if (x) c . poly . Attach (temp) ;
12                        p = Aiter.Next () ; q = Biter.Next () ;  // advance to next term
13                        break;
14                  case '<':
15                        temp.Init(q →coef, q →exp) ; c.poly.Attach (temp) ;
16                        q = Biter.Next () ; // next term of b
17                        break;
18                  case '>':
19                        temp . Init(p →coef, p →exp) ; c.poly.Attach (temp) ;
20                        p = Aiter . Next () ; // next term of a
21            }
22      }
23      while (Aiter.NotNull ()) {  // copy rest of a
24            temp.Init(p →coef, p →exp) ; c.poly.Attach (temp) ;
25            p = Aiter.Next () ;
26      }
27      while (Biter.NotNull ()) {  // copy rest of b
28            temp.Init(q →coef, q →exp) ; c.poly.Attach (temp) ;
29            q=Biter.Next () ;
30      }
31      return c ;
32 }
```

adding two polynomials

Generating The First Three Terms



Analysis of Operator+

- **Computing Time**
  - (1) coefficient additions
  - (2) exponent comparisons
  - (3) addition/deletions to available space
  - (4) creation of new nodes
- **Assume that**
  - polynomial *a* has *m* terms, while *b* has *n* terms
- **Coefficient additions: [0, min{m, n}] times**
  - **Lower-bound**: when **none** of the exponents **are equal**
  - **Upper-bound**: when the exponents of one polynomial are a **subset** of the exponents of the other polynomial
- **Overall Complexity: O(m+n)**

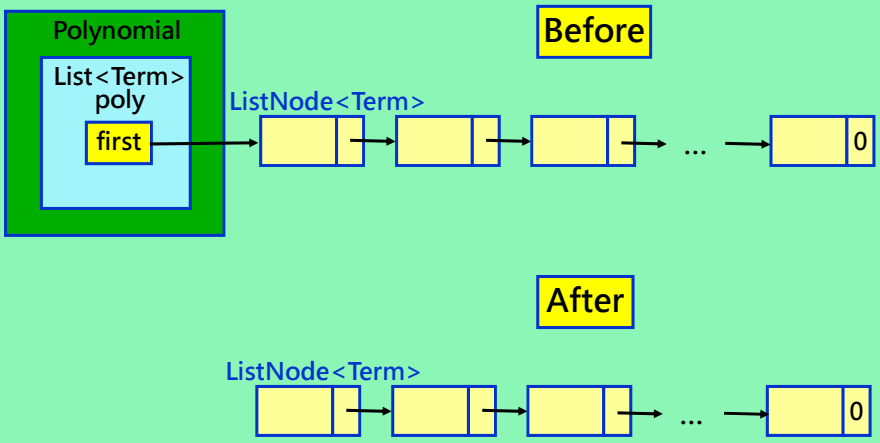
Example: Polynomial Computation

- **d(x) = a(x) * b(x) + c(x)**

```
void  func()
{
    Polynomial  a, b, c, d, t;
    cin >> a; // read and create polynomial
    cin >> b;
    cin >> c;
    t = a * b;
    d = t + c;
    cout << d;
}
```

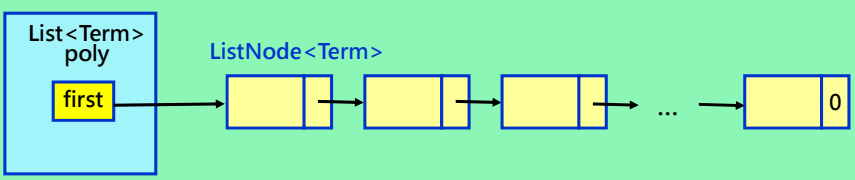
Problem:
When the function terminates, the memory occupied by the polynomials a, b, c, d, t may not be freed automatically
→ because ListNode<Term> objects are not physically contained in List<Term> objects.

Polynomial Object Before And After It Goes Out Of Scope



Erase A Polynomial

```
template <class Type>
List<Type>::~List()
// Free all nodes in the chain
{
    ListNode<Type> *next;
    for (; first; first = next) {
        next = first → link
        delete first;
    }
}
```



Space Management Of ListNodes

- **By incorporating Circular List**
  - **Freeing all the nodes in a list is more efficient**
- **Reuse strategy**
  - **Deletion: nodes that have been "deleted" are actually maintained in a pool → available (av) space**
  - **Request for a new node:**

    (1) If available space is not empty, recycle one of them

    (2) If available space is empty → by "new" command

Getting and Returning A Node

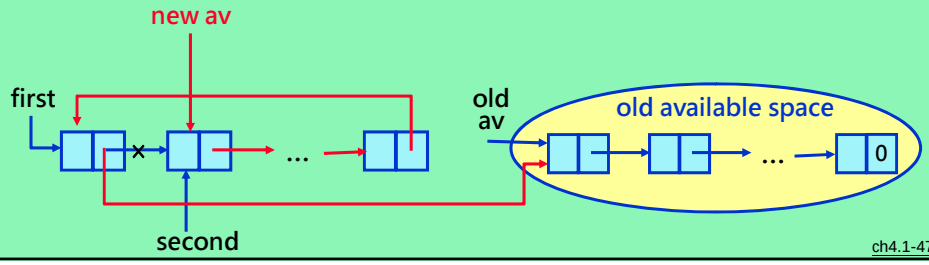
```
template <class Type>
ListNode<Type>* CircularList::GetNode()
// Provide a node for use
{
    ListNode<Type>* x;
    if ( ! av )  x = new ListNode<Type>;   // request for a new one
   else  {  x = av;   av = av → link; }        // recycle one from AV-pool
   return x;
}
```

```
template <class Type>
void  CircularList<Type>::RetNode(ListNode<Type>* x)
// Free the node pointed by x
{
    x → link = av;
   av = x;
}
```

Erasing A Circular List

```
template <class Type>
void CircularList<Type>::~CircularList()
// Erase the entire circular list pointed by first
{
   if ( first ) {
      ListNode*  second = first → link; // second node
      first → link = av;    // first node linked to av
      av = second;   // second node of list becomes front of av list
      first = 0;
   }
}
```

A Circular List can be erased in a fixed amount of time
→ Independent of the number of nodes in the list




```
Polynomial  operator+ (const Polynomial& a, const Polynomial& b)
{
   Term  *p, *q, temp;
  CircularListIterator<Term>  Aiter (a.poly);
  CircularListIterator<Term>  Biter (b.poly);
  Polynomial  c;  // assume the constructor creates a head node with exp = -1
  p = Aiter.first();   q = Biter.first();
  while(1) {
     switch ( compare ( p→exp, q→exp ) )  {
        case '=':
            if ( p → exp == -1 ) return c;
            else {
                 int  sum = p → coef + q → coef;
                 if (sum) {  temp.Init (sum, q → exp);  c.poly.Attach ( temp);  }
                 p = Aiter.Next();   q = Biter.Next();
            }
            break;
        case '<':
                 temp.Init(q→coef, q→exp); c.poly.Attach(temp); q=Biter.Next(); break;
        case '>':
                 temp.Init(p→coef, p→exp); c.poly.Attach(temp); p=Aiter.Next(); break;
     } // end of switch and while
  }
}
```

Adding circularly represented polynomials

Using List Iterator in STL

C++ reference 網頁: http://www.cppreference.com/wiki/start

STL 網頁: http://www.cppreference.com/wiki/stl/start

STL List 網頁: http://www.cppreference.com/wiki/stl/list/start

```
// Create a list of characters
list<char> my_list;
for( int i = 0; i < 10; i++ ) {
  my_list.push_front( i + 'a' );
}
// Display the list
list<char>::iterator it;
for( it = my_list.begin(); it != my_list.end(); ++it) {
    cout << *it;
}
```


The End of Part I

Next Topic:

Linked Lists Part II