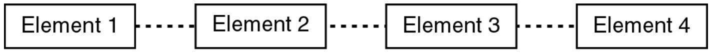
Lecture 7 - Data Structures

Linked Lists

What is a List?

- A list is a *homogeneous collection* of *elements*, with a *linear relationship* between elements.
- That is, each list element (except the first) has a *unique predecessor*, and each element (except the last) has a *unique successor*.



Properties of Lists

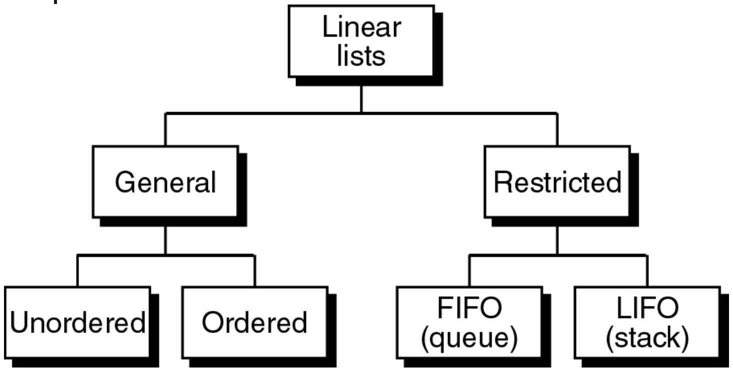
•Can have a *single element*

•Can have *no* elements

•There can be *lists of lists*

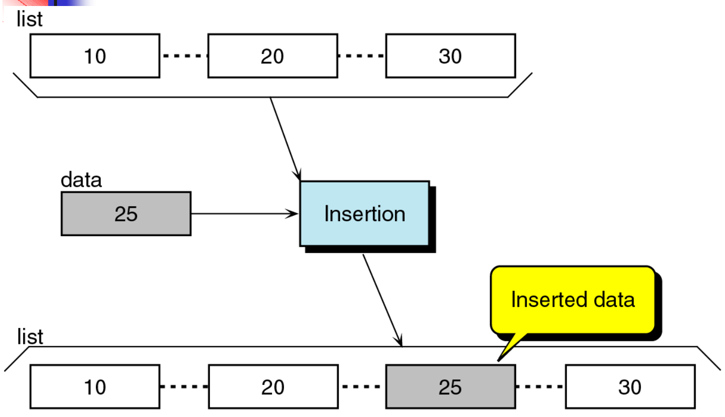
Consider Every Day Lists

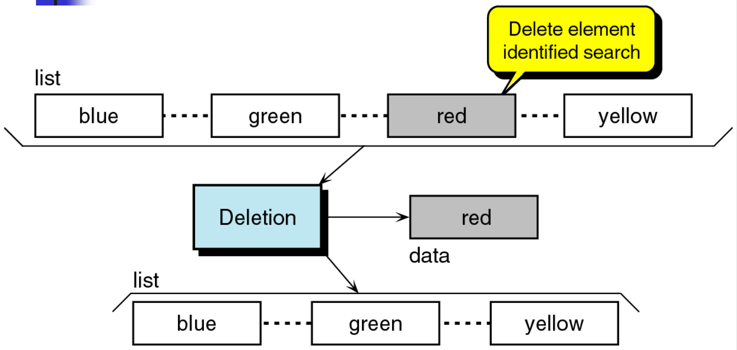
- Groceries to be purchased
- Job to-do list
- List of assignments for a course
- Dean's list
- Can you name some others??

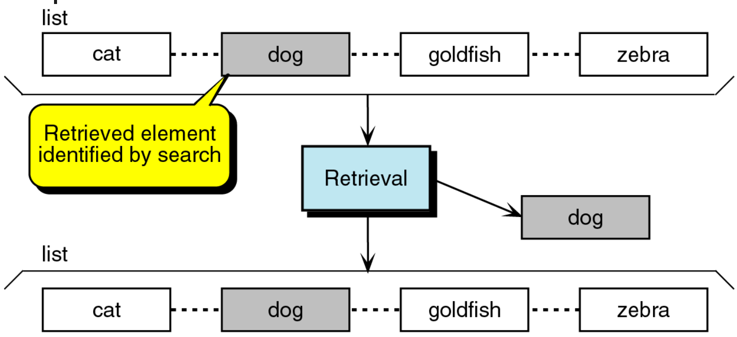


Basic Operations

- *Construct* an empty list
- Determine whether or not *empty*
- *Insert* an element into the list
- *Delete* an element from the list
- *Traverse* (iterate through) the list to
  - Modify
  - Output
  - Search for a specific value
  - Copy or save
  - Rearrange


list
10 20 30
data
25
Insertion
Inserted data
list
10 20 25 30


Delete element identified search
list
blue green red yellow
Deletion
red
data
list
blue green yellow

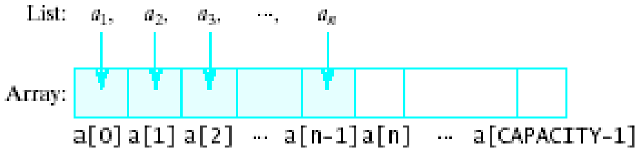

list
cat dog goldfish zebra
Retrieved element identified by search
Retrieval
dog
list
cat dog goldfish zebra

Designing a `List` Class

- Should contain at least the following function members
  - Constructor
  - `empty()`
  - `insert()`
  - `delete()`
  - `display()`
- Implementation involves
  - Defining *data members*
  - Defining *function members* from design phase

Array-Based Implementation of Lists

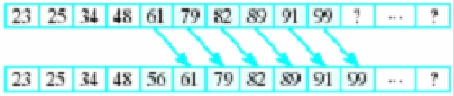
- An *array* is a viable choice for storing list elements
  - *Elements are sequential*
  - It is a commonly *available* data type
  - *Algorithm* development is *easy*
- Normally sequential orderings of list elements match with array elements


List: a_1, a_2, a_3, ⋯, a_n
Array:
a[0] a[1] a[2] ⋯ a[n-1] a[n] ⋯ a[CAPACITY-1]

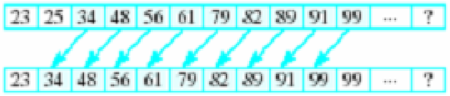
For an array, add a *mySize* member to store the *length (n)* of the list

Implementing Operations

- **Constructor**
  - Static array allocated at compile time
- **Empty**
  - Check if *size == 0*
- **Traverse**
  - Use a loop from **0th** element to `size – 1`
- **Insert**
  - *Shift* elements to *right* of insertion point



- **Delete**
  - *Shift* elements *back*



Basic Operations

Construction: For array: Set *mySize* to *0* ; if run-time array, allocate memory for it.

Empty: mySize == 0

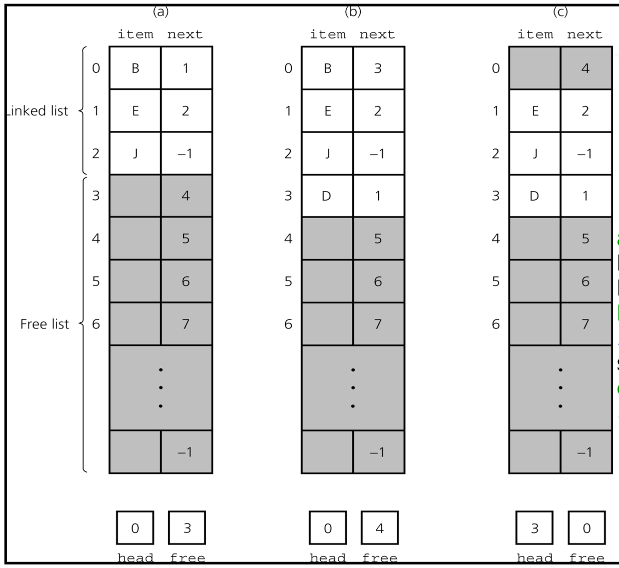
Traverse:

```
    for (int i = 0; i < size; i++)
        { Process(a[i]); }
or
    i = 0;
    while (i < size)
    {       Process(a[i]);
                    i++;
    }
```

Insert: Insert *6* after *5* in *3, 5, 8, 9, 10, 12, 13, 15*

3, 5, 6, 8, 9, 10, 12, 13, 15

```
// Shift array elements to make room.
for (int i = mySize ; i > pos; i-- )
    myArray[i] = myArray[i  - 1] ;
// Insert item at position pos and increase list size
myArray[pos] = item ;
mySize++ ;
```



	(a) item	(a) next	(b) item	(b) next	(c) item	(c) next
0	B	1	B	3		4
1	E	2	E	2	E	2
2	J	–1	J	–1	J	–1
3		4	D	1	D	1
4		5		5		5
5		6		6		6
6		7		7		7
	⋮		⋮		⋮	
		–1		–1		–1

	(a)	(b)	(c)
head	0	0	3
free	3	4	0

a) An array-based linked list
b) after *inserting D* in sorted order;
c) after *deleting B*

Basic properties

- The *efficiency* of an *insert* function depends on the *number of array elements* that must be *shifted* to make room for new element (number of times the *for* loop is executed).
- In the *worst case*, the new item must be inserted at the *beginning* of the list, which requires *shifting all of the array elements*.
- In the *average case*, *one-half* of the array elements must be shifted.
- Thus, for a list of *size n* the computing time for the worst-case and the average-case for an *insert* function is ***O(n).***
- The *best case* - *insert at the end* of the list. No elements need to be shifted, and the computing time does not depend on the size of the list. The best-case *complexity* for insertion is ***O(1).***
- If the *order* in a list is *not important*, new items can be inserted at any convenient location; in particular, at the *end of the list.*

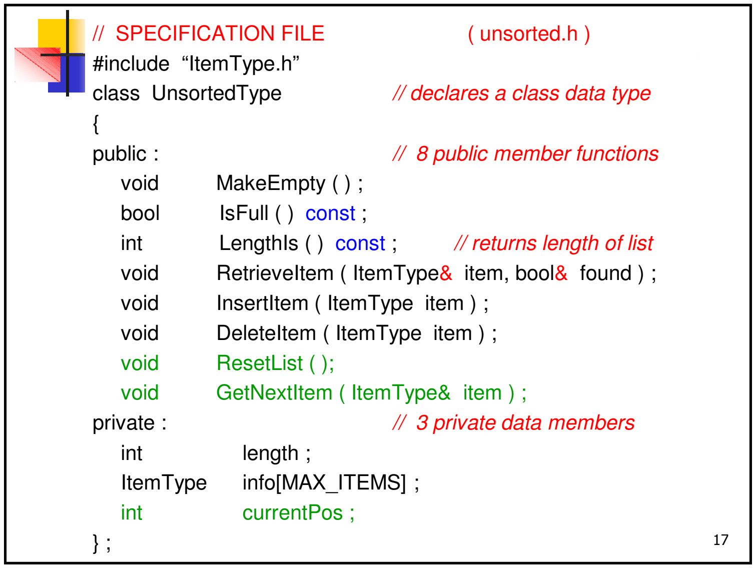
Delete: Delete *5* from preceding list:

3, 5, 6, 8, 9, 10, 12, 13, 15

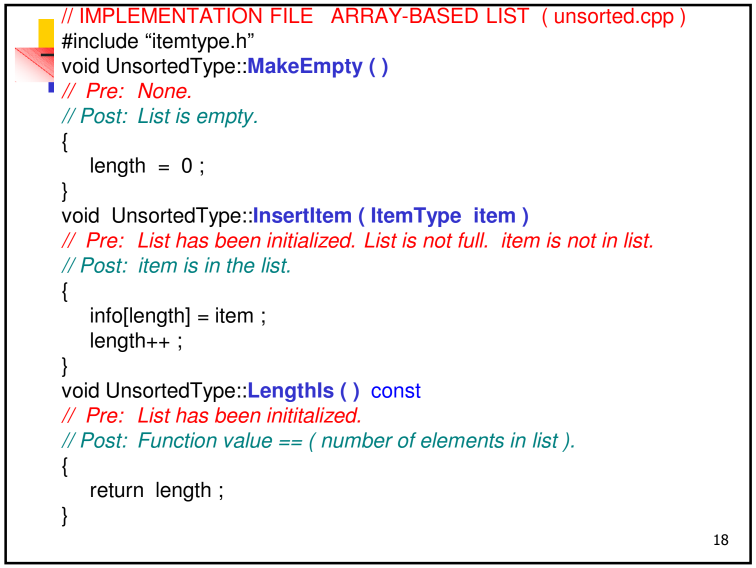
3, 6, 8, 9, 10, 12, 13, 15

```
// Shift array elements to close the gap.
for (int i = pos ; i < mySize – 1 ; i++ )
    myArray[i] = myArray[i + 1] ;
// Decrease list size
mySize-- ;
```

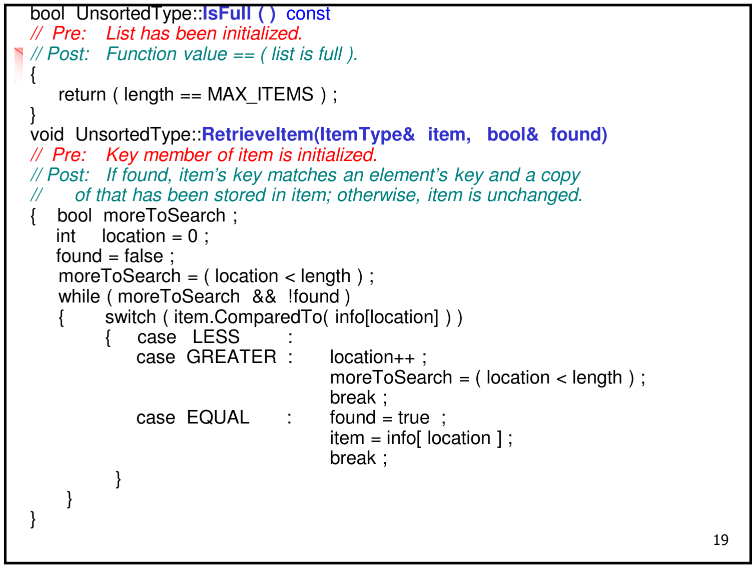
The computing time of a *delete* operation is the *same* as that of an *insert* function: ***O(n)*** in the *worst* and *average* cases and ***O(1)*** in the *best* case.



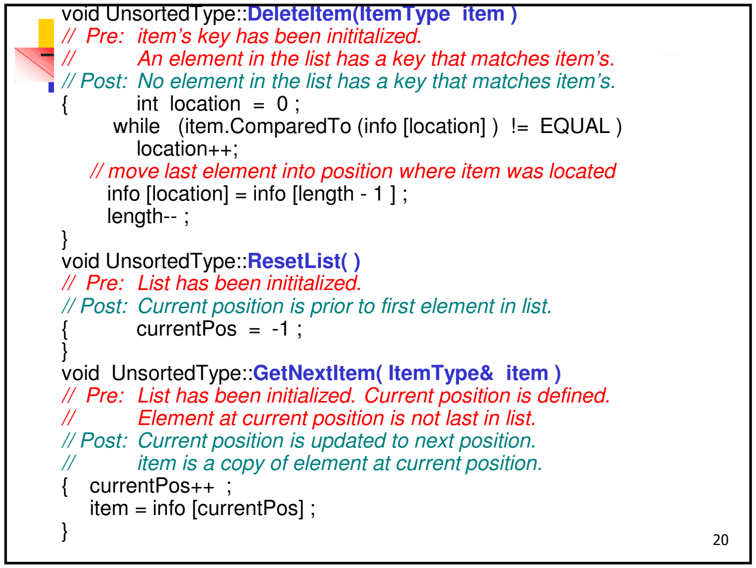
```
//  SPECIFICATION FILE                    ( unsorted.h )
#include  "ItemType.h"
class  UnsortedType                 // declares a class data type
{
public :                            //  8 public member functions
    void        MakeEmpty ( ) ;
    bool        IsFull ( )  const ;
    int         LengthIs ( )  const ;       // returns length of list
    void        RetrieveItem ( ItemType&  item, bool&  found ) ;
    void        InsertItem ( ItemType  item ) ;
    void        DeleteItem ( ItemType  item ) ;
    void        ResetList ( );
    void        GetNextItem ( ItemType&  item ) ;
private :                           //  3 private data members
    int             length ;
    ItemType        info[MAX_ITEMS] ;
    int             currentPos ;
} ;
```



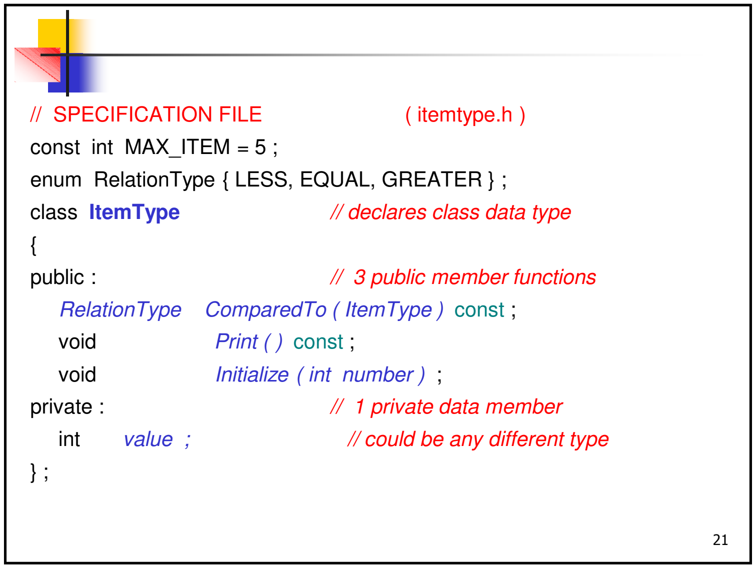
```
// IMPLEMENTATION FILE   ARRAY-BASED LIST  ( unsorted.cpp )
#include "itemtype.h"
void UnsortedType::MakeEmpty ( )
//  Pre:  None.
// Post:  List is empty.
{
    length  =  0 ;
}
void  UnsortedType::InsertItem ( ItemType  item )
//  Pre:  List has been initialized.  List is not full.  item is not in list.
// Post:  item is in the list.
{
    info[length] = item ;
    length++ ;
}
void UnsortedType::LengthIs ( )  const
//  Pre:  List has been inititalized.
// Post:  Function value == ( number of elements in list ).
{
    return  length ;
}
```



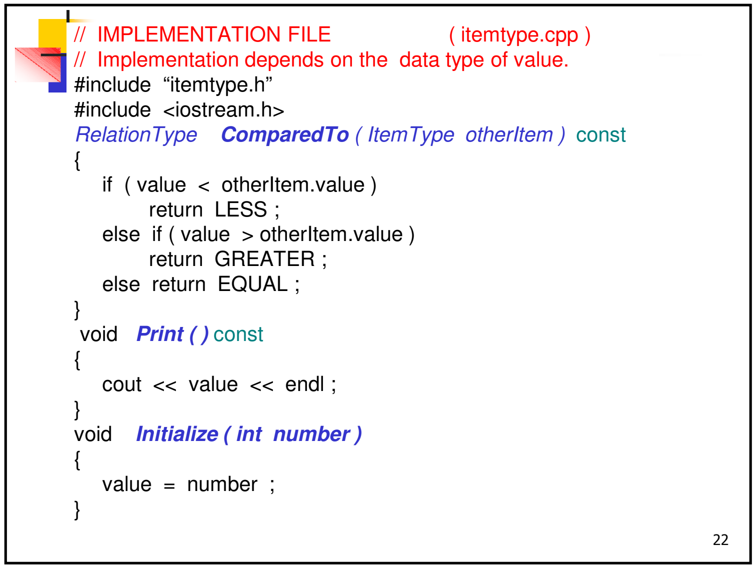
```
bool  UnsortedType::IsFull ( )  const
//  Pre:    List has been initialized.
// Post:    Function value == ( list is full ).
{
    return ( length == MAX_ITEMS ) ;
}
void  UnsortedType::RetrieveItem(ItemType&  item,   bool&  found)
//  Pre:    Key member of item is initialized.
// Post:    If found, item's key matches an element's key and a copy
//          of that has been stored in item; otherwise, item is unchanged.
{   bool  moreToSearch ;
    int     location = 0 ;
    found = false ;
    moreToSearch = ( location < length ) ;
    while ( moreToSearch  &&  !found )
    {       switch ( item.ComparedTo( info[location] ) )
            {   case   LESS        :
                case  GREATER  :        location++ ;
                                        moreToSearch = ( location < length ) ;
                                        break ;
                case  EQUAL       :     found = true   ;
                                        item = info[ location ] ;
                                        break ;
            }
    }
}
```



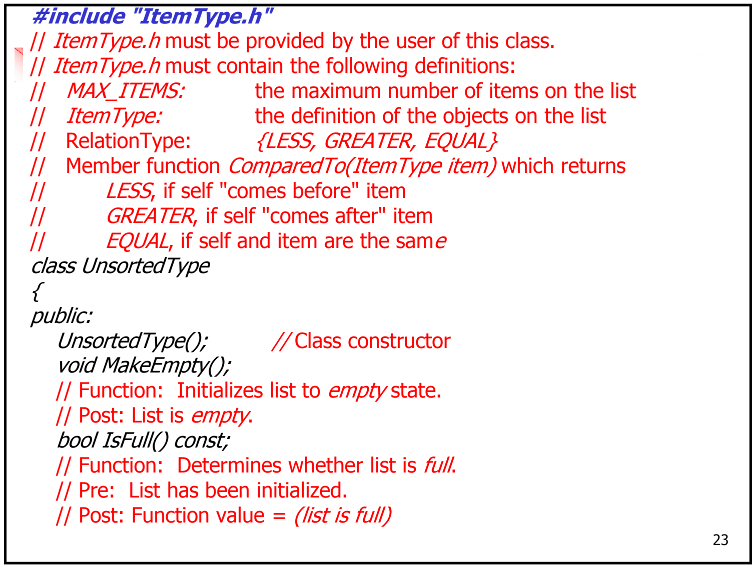
```
void UnsortedType::DeleteItem(ItemType  item )
//  Pre:   item's key has been inititalized.
//         An element in the list has a key that matches item's.
// Post:  No element in the list has a key that matches item's.
{         int  location  =  0 ;
       while   (item.ComparedTo (info [location] )  !=  EQUAL )
          location++;
    // move last element into position where item was located
       info [location] = info [length - 1 ] ;
       length-- ;
}
void UnsortedType::ResetList( )
//  Pre:  List has been inititalized.
// Post:  Current position is prior to first element in list.
{         currentPos  =  -1 ;
}
void  UnsortedType::GetNextItem( ItemType&  item )
//  Pre:  List has been initialized.  Current position is defined.
//        Element at current position is not last in list.
// Post:  Current position is updated to next position.
//        item is a copy of element at current position.
{   currentPos++  ;
    item = info [currentPos] ;
}
```



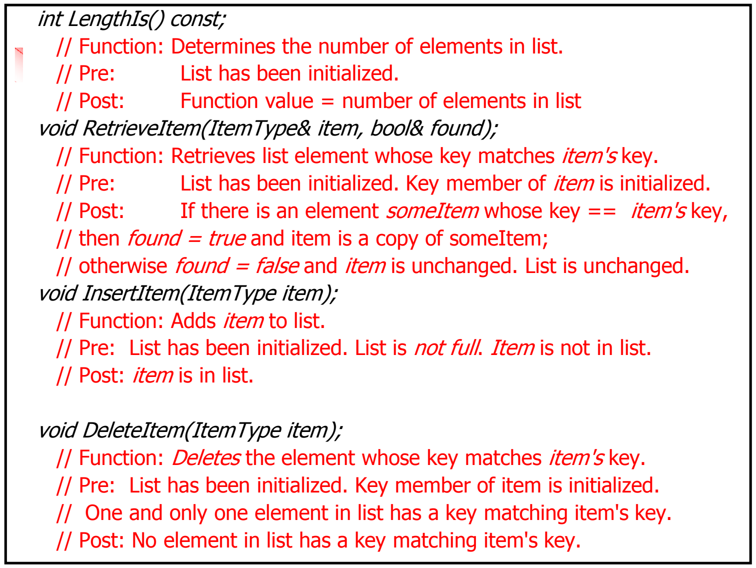
```
// SPECIFICATION FILE                    ( itemtype.h )
const  int  MAX_ITEM = 5 ;
enum  RelationType { LESS, EQUAL, GREATER } ;
class  ItemType                        // declares class data type
{
public :                               //  3 public member functions
    RelationType    CompareTo ( ItemType )  const ;
    void            Print ( )  const ;
    void            Initialize ( int  number )  ;
private :                              //  1 private data member
    int      value  ;                   // could be any different type
} ;
```



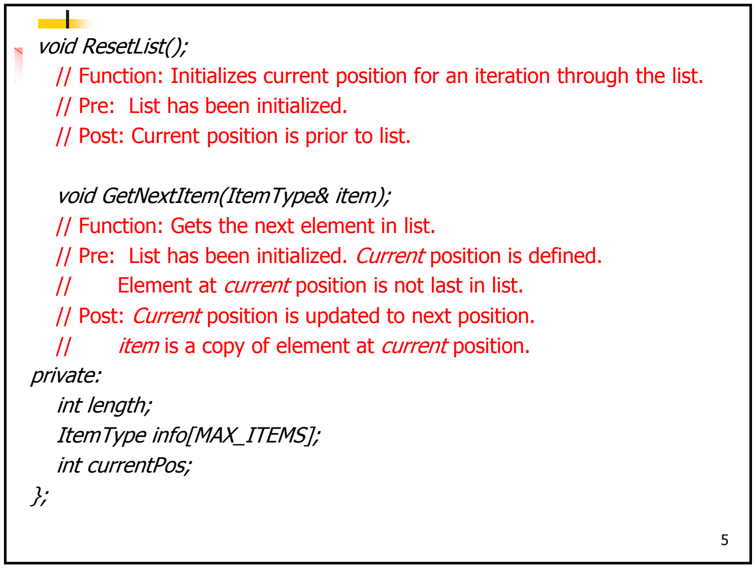
```
// IMPLEMENTATION FILE                  ( itemtype.cpp )
// Implementation depends on the  data type of value.
#include  "itemtype.h"
#include  <iostream.h>
RelationType    CompareTo ( ItemType  otherItem )  const
{
    if  ( value  <  otherItem.value )
          return  LESS ;
    else  if ( value  > otherItem.value )
          return  GREATER ;
    else  return  EQUAL ;
}
 void   Print ( ) const
{
    cout  <<  value  <<  endl ;
}
void    Initialize ( int  number )
{
    value  =  number  ;
}
```



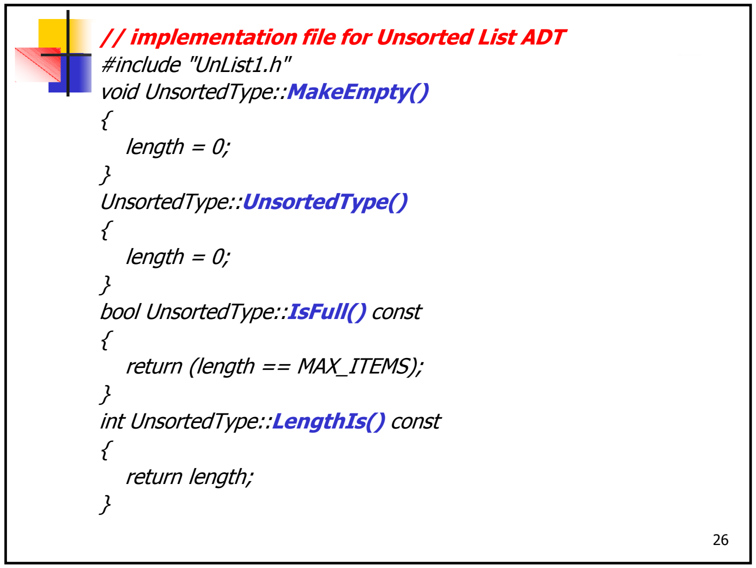
```
#include "ItemType.h"
// ItemType.h must be provided by the user of this class.
// ItemType.h must contain the following definitions:
//   MAX_ITEMS:          the maximum number of items on the list
//   ItemType:           the definition of the objects on the list
//   RelationType:       {LESS, GREATER, EQUAL}
//   Member function CompareTo(ItemType item) which returns
//        LESS, if self "comes before" item
//        GREATER, if self "comes after" item
//        EQUAL, if self and item are the same
class UnsortedType
{
public:
   UnsortedType();        // Class constructor
   void MakeEmpty();
   // Function:  Initializes list to empty state.
   // Post: List is empty.
   bool IsFull() const;
   // Function:  Determines whether list is full.
   // Pre:  List has been initialized.
   // Post: Function value = (list is full)
```



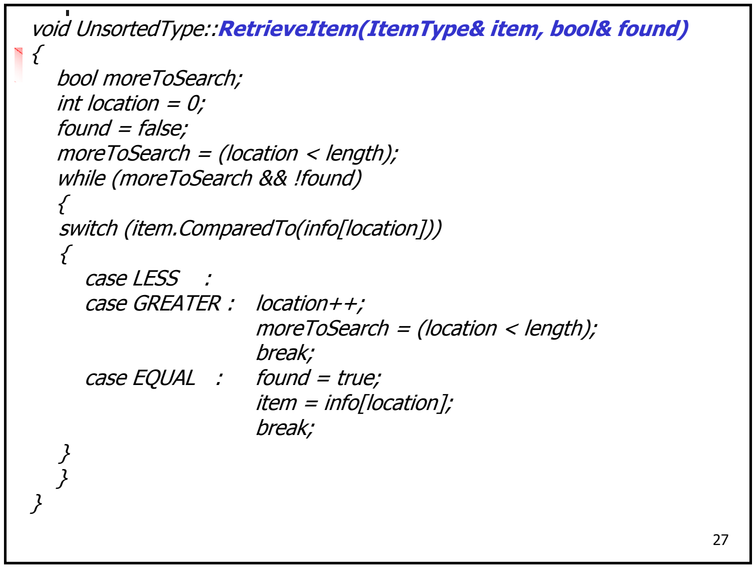
```
int LengthIs() const;
  // Function: Determines the number of elements in list.
  // Pre:          List has been initialized.
  // Post:         Function value = number of elements in list
void RetrieveItem(ItemType& item, bool& found);
  // Function: Retrieves list element whose key matches item's key.
  // Pre:          List has been initialized. Key member of item is initialized.
  // Post:         If there is an element someItem whose key ==  item's key,
  // then found = true and item is a copy of someItem;
  // otherwise found = false and item is unchanged. List is unchanged.
void InsertItem(ItemType item);
  // Function: Adds item to list.
  // Pre:  List has been initialized. List is not full. Item is not in list.
  // Post:  item is in list.

void DeleteItem(ItemType item);
  // Function: Deletes the element whose key matches item's key.
  // Pre:  List has been initialized. Key member of item is initialized.
  //  One and only one element in list has a key matching item's key.
  // Post: No element in list has a key matching item's key.
```



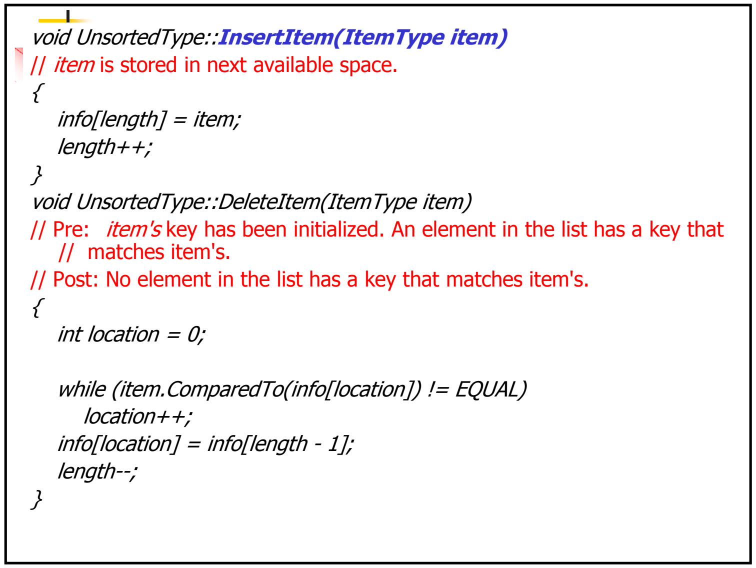
```
void ResetList();
  // Function: Initializes current position for an iteration through the list.
  // Pre:  List has been initialized.
  // Post: Current position is prior to list.

  void GetNextItem(ItemType& item);
  // Function: Gets the next element in list.
  // Pre:  List has been initialized. Current position is defined.
  //       Element at current position is not last in list.
  // Post: Current position is updated to next position.
  //       item is a copy of element at current position.
private:
  int length;
  ItemType info[MAX_ITEMS];
  int currentPos;
};
```



```
// implementation file for Unsorted List ADT
#include "UnList1.h"
void UnsortedType::MakeEmpty()
{
   length = 0;
}
UnsortedType::UnsortedType()
{
   length = 0;
}
bool UnsortedType::IsFull() const
{
   return (length == MAX_ITEMS);
}
int UnsortedType::LengthIs() const
{
   return length;
}
```



```
void UnsortedType::RetrieveItem(ItemType& item, bool& found)
{
   bool moreToSearch;
   int location = 0;
   found = false;
   moreToSearch = (location < length);
   while (moreToSearch && !found)
   {
   switch (item.ComparedTo(info[location]))
   {
      case LESS    :
      case GREATER :   location++;
                       moreToSearch = (location < length);
                       break;
      case EQUAL   :   found = true;
                       item = info[location];
                       break;
   }
   }
}
```



```
void UnsortedType::InsertItem(ItemType item)
// item is stored in next available space.
{
   info[length] = item;
   length++;
}
void UnsortedType::DeleteItem(ItemType item)
// Pre:   item's key has been initialized. An element in the list has a key that
   //  matches item's.
// Post: No element in the list has a key that matches item's.
{
   int location = 0;

   while (item.ComparedTo(info[location]) != EQUAL)
      location++;
   info[location] = info[length - 1];
   length--;
}
```

```
void UnsortedType::ResetList()
{
   currentPos = -1;
}

void UnsortedType::GetNextItem(ItemType& item)
{
   currentPos++;
   item = info[currentPos];
}
#include <iostream.h>
enum RelationType {LESS, GREATER, EQUAL};
class ItemType
{
public:
   void Print();
   void Insert(int newItem);
   RelationType ComparedTo(ItemType item2);
   int item;
};
```

```
void ItemType::Print()
{
   cout << item << endl;
}
void ItemType::Insert(int newItem)
{
   item = newItem;
}
RelationType ItemType::ComparedTo(ItemType item2)
{
   if (item < item2.item)
      return LESS;
   else if (item == item2.item)
      return EQUAL;
   else return GREATER;

}
const int MAX_ITEMS = 5;
```

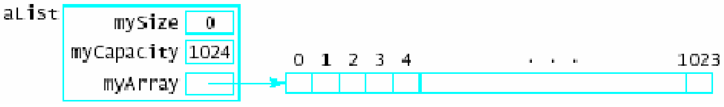
Notes on Class Design

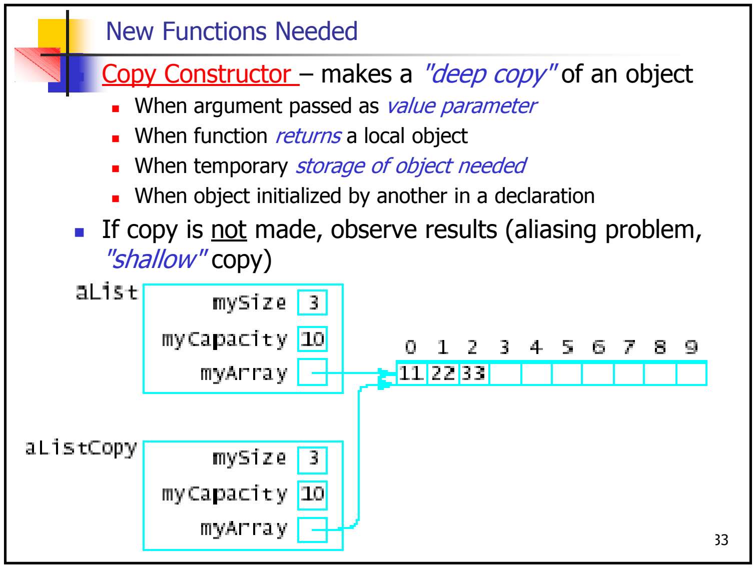
If a class allocates memory at *run time* using the `new`, then it should provide …

- A *destructor*
- A *copy constructor*
- An *assignment operator*

New Functions Needed

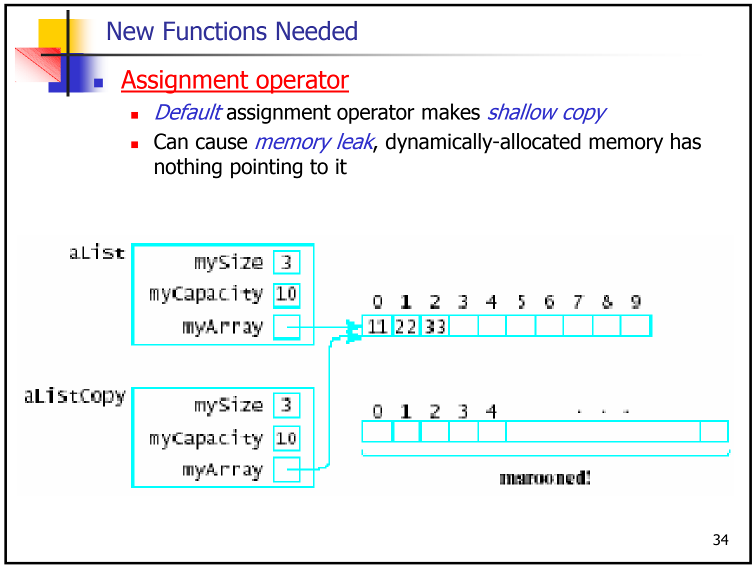
- Destructor
  - When class object goes out of scope the *pointer* to the dynamically allocated memory is *reclaimed* automatically
  - The *dynamically* allocated memory is *not*
  - The *destructor reclaims dynamically allocated memory*





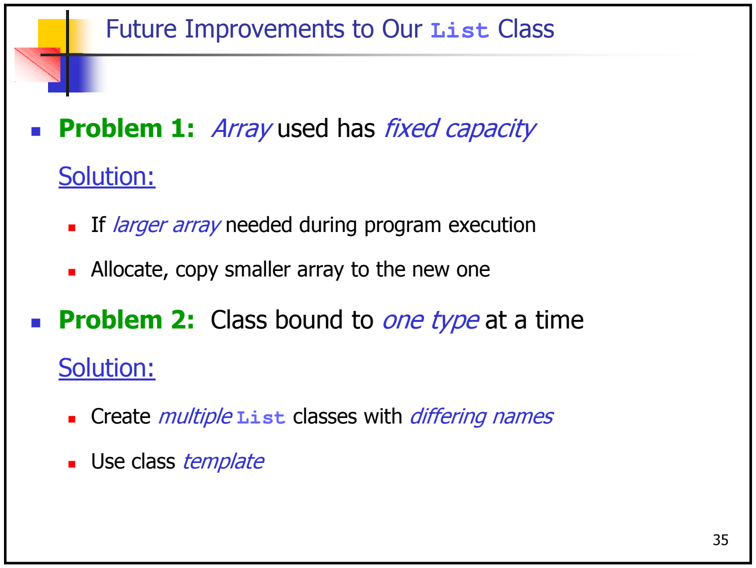
New Functions Needed

- Copy Constructor – makes a *"deep copy"* of an object
  - When argument passed as *value parameter*
  - When function *returns* a local object
  - When temporary *storage of object needed*
  - When object initialized by another in a declaration
- If copy is not made, observe results (aliasing problem, *"shallow"* copy)



New Functions Needed

- Assignment operator
  - *Default* assignment operator makes *shallow copy*
  - Can cause *memory leak*, dynamically-allocated memory has nothing pointing to it

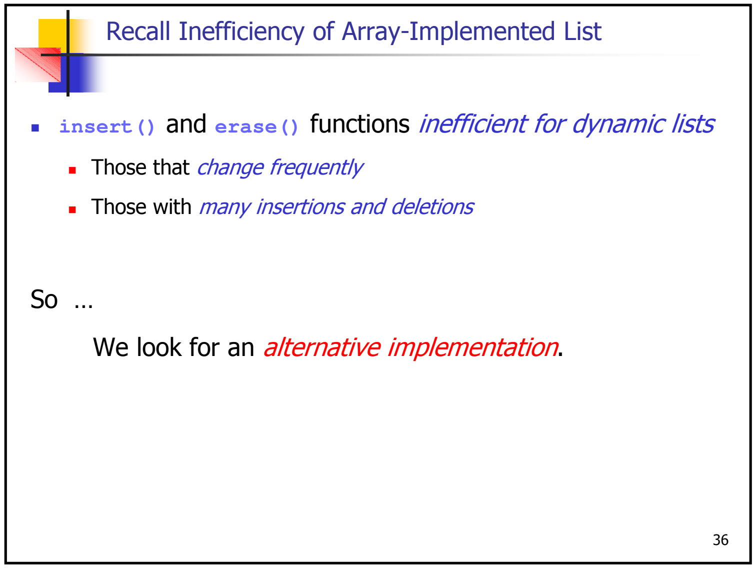


Future Improvements to Our `List` Class

- **Problem 1:** *Array* used has *fixed capacity*

  Solution:
  - If *larger array* needed during program execution
  - Allocate, copy smaller array to the new one
- **Problem 2:** Class bound to *one type* at a time

  Solution:
  - Create *multiple* `List` classes with *differing names*
  - Use class *template*

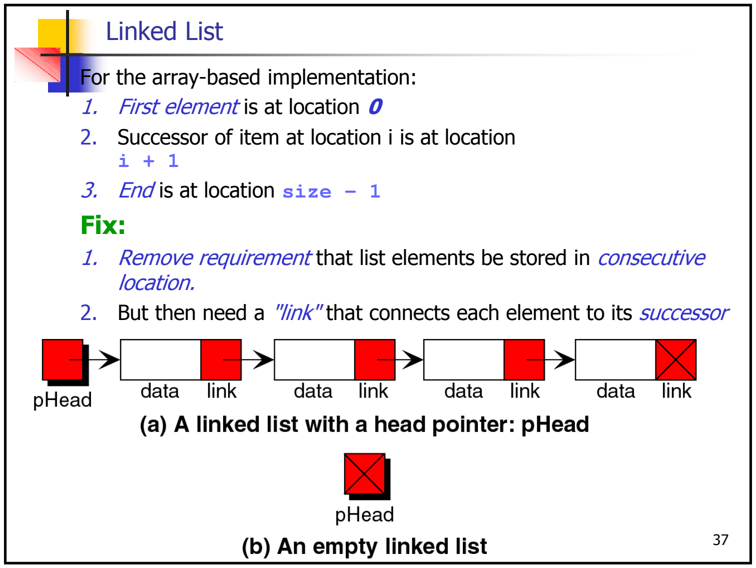


Recall Inefficiency of Array-Implemented List

- `insert()` and `erase()` functions *inefficient for dynamic lists*
  - Those that *change frequently*
  - Those with *many insertions and deletions*

So …

We look for an *alternative implementation*.



Linked List

For the array-based implementation:

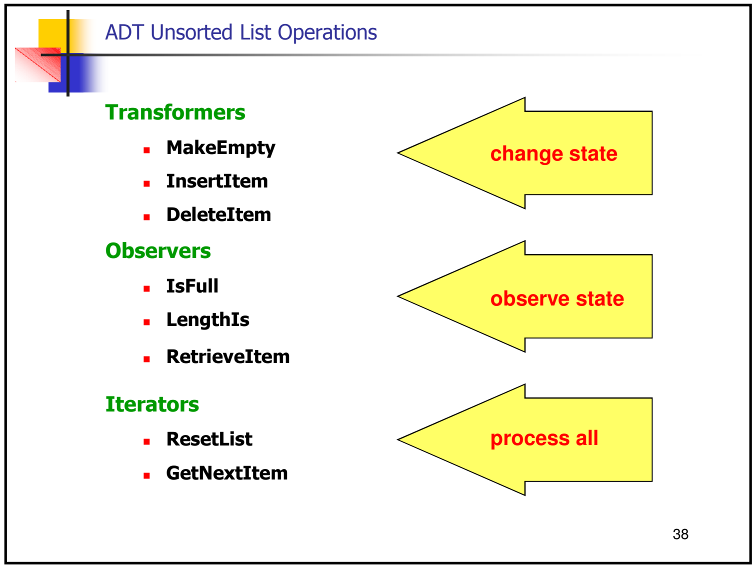
1. *First element* is at location ***0***
2. Successor of item at location i is at location `i + 1`
3. *End* is at location `size – 1`

Fix:

1. *Remove requirement* that list elements be stored in *consecutive location.*
2. But then need a *"link"* that connects each element to its *successor*

(a) A linked list with a head pointer: pHead

(b) An empty linked list



ADT Unsorted List Operations

Transformers

- **MakeEmpty**
- **InsertItem**
- **DeleteItem**

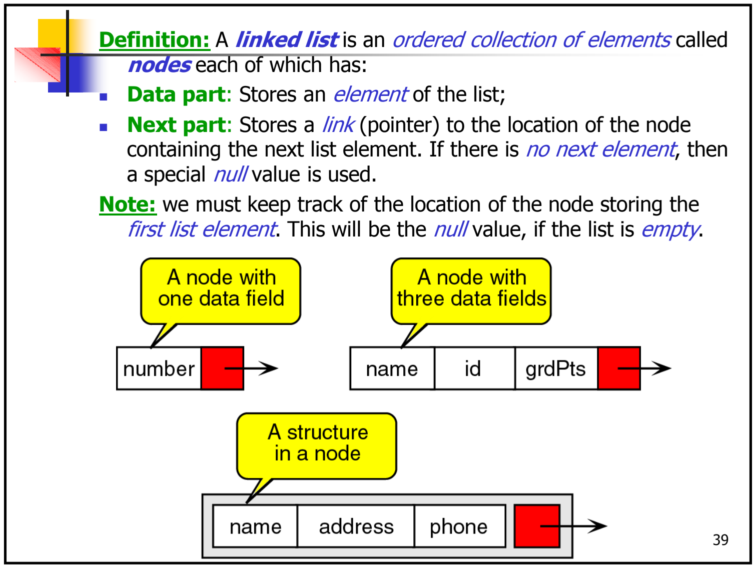
Observers

- **IsFull**
- **LengthIs**
- **RetrieveItem**

Iterators

- **ResetList**
- **GetNextItem**

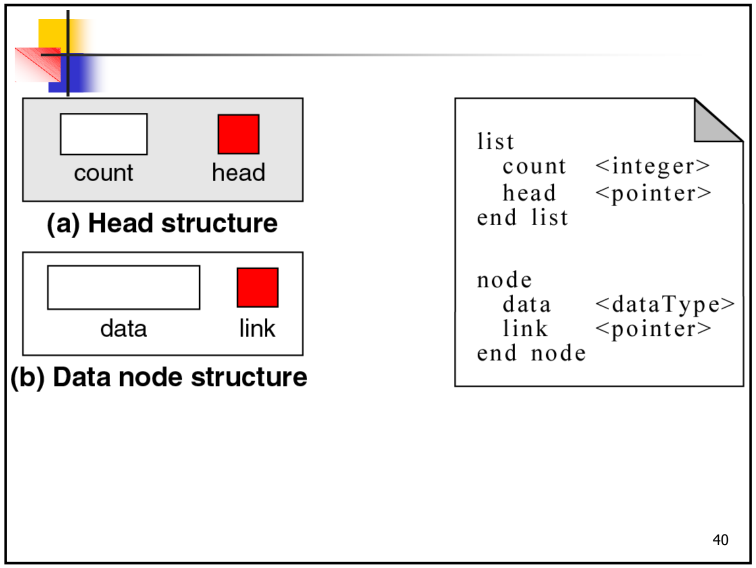
change state
observe state
process all



Definition: A ***linked list*** is an *ordered collection of elements* called ***nodes*** each of which has:

- **Data part**: Stores an *element* of the list;
- **Next part**: Stores a *link* (pointer) to the location of the node containing the next list element. If there is *no next element*, then a special *null* value is used.

Note: we must keep track of the location of the node storing the *first list element*. This will be the *null* value, if the list is *empty*.



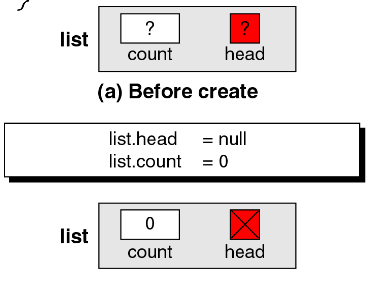
(a) Head structure

(b) Data node structure

```
list
   count    <integer>
   head     <pointer>
end list


node
   data     <dataType>
   link     <pointer>
end node
```

Basic Operations:

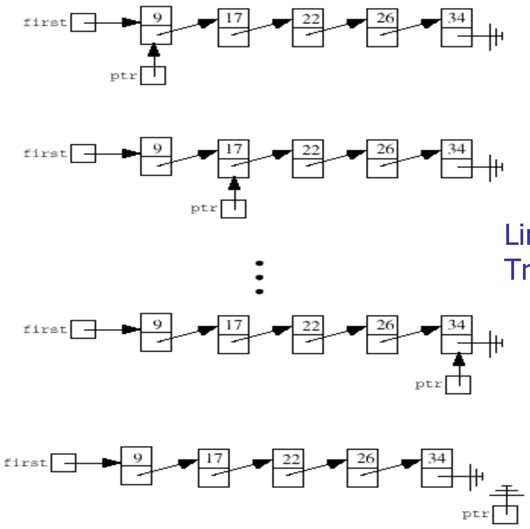
```
Construction:   first = null_value ;
Empty:          first = = null_value ;
Traverse:       ptr = first ;
                while (ptr != null_value)
                {
                        Process data part of node pointed to by ptr ;
                        Ptr = next part of node pointed to by ptr
                }
```



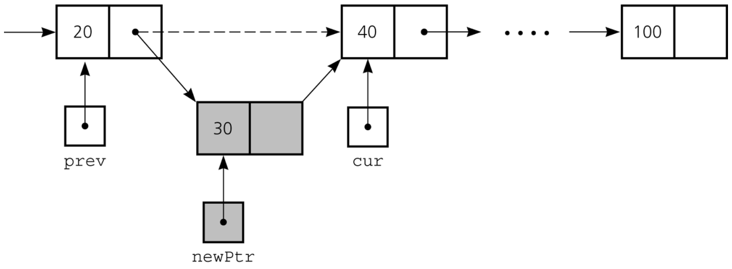
(a) Before create

(b) After create

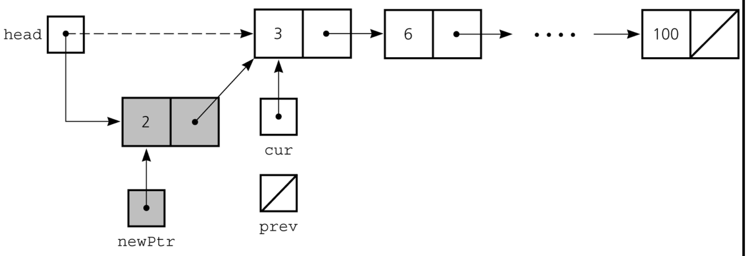
Linked List Traversal

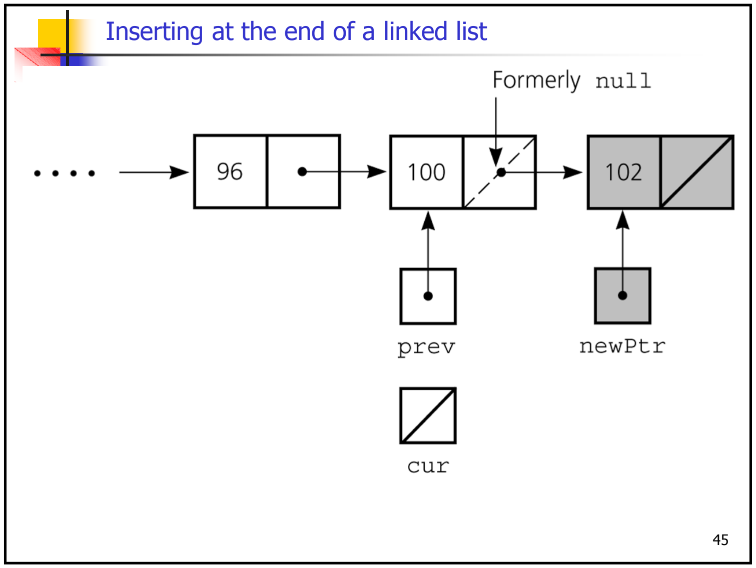


Inserting a new node into a linked list

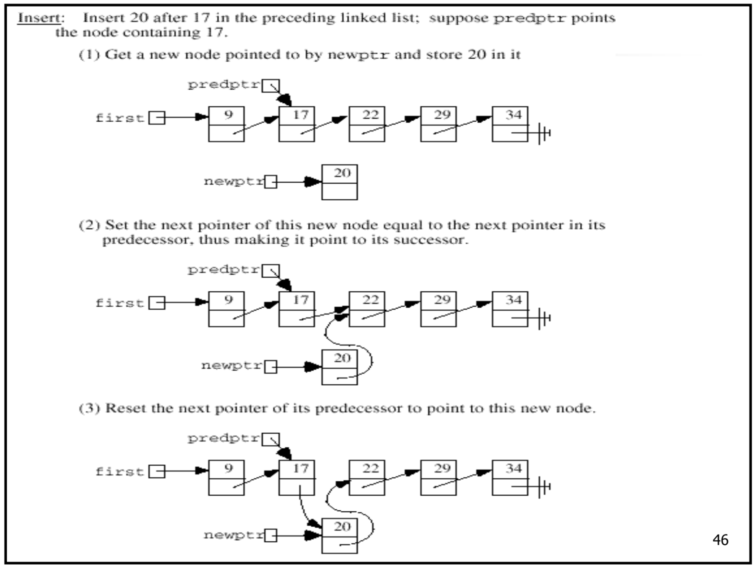


Inserting at the beginning of a linked list





Inserting at the end of a linked list

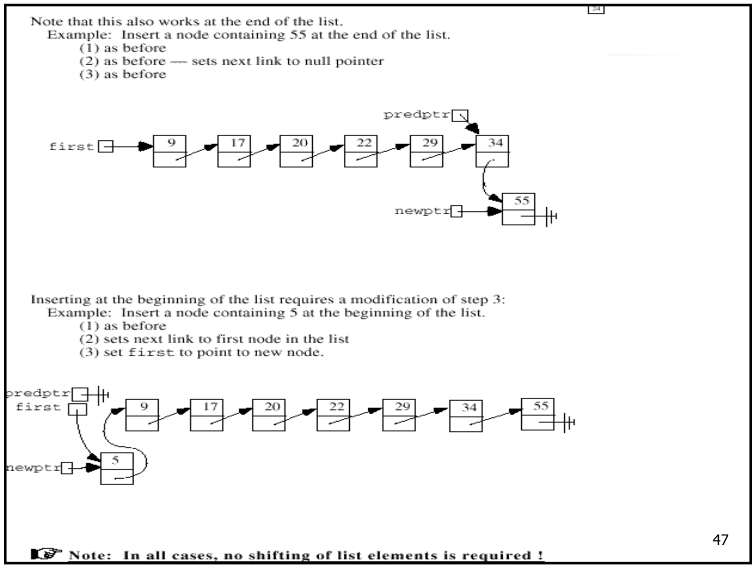


Insert: Insert 20 after 17 in the preceding linked list; suppose `predptr` points the node containing 17.

(1) Get a new node pointed to by `newptr` and store 20 in it

(2) Set the next pointer of this new node equal to the next pointer in its predecessor, thus making it point to its successor.

(3) Reset the next pointer of its predecessor to point to this new node.



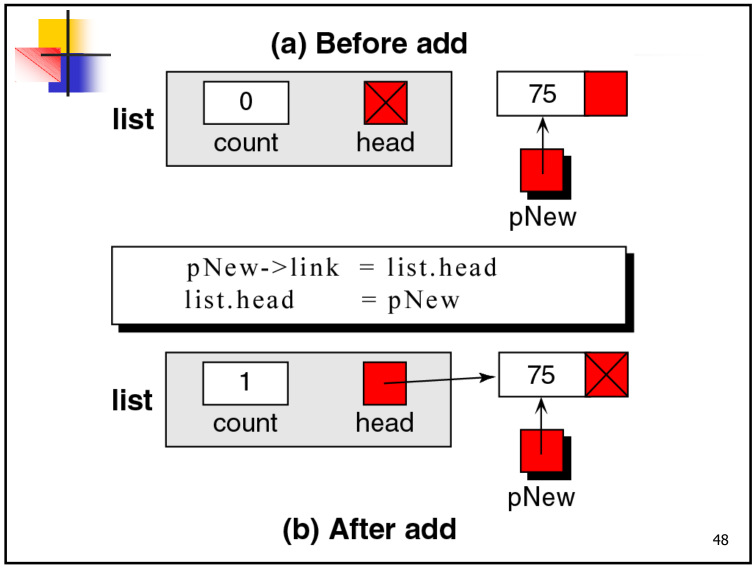
Note that this also works at the end of the list.

Example: Insert a node containing 55 at the end of the list.
(1) as before
(2) as before — sets next link to null pointer
(3) as before

Inserting at the beginning of the list requires a modification of step 3:

Example: Insert a node containing 5 at the beginning of the list.
(1) as before
(2) sets next link to first node in the list
(3) set `first` to point to new node.

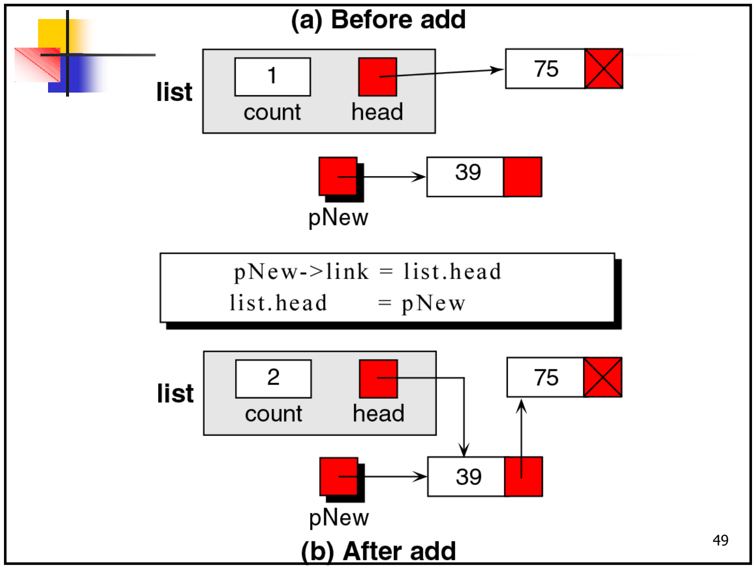
☞ **Note: In all cases, no shifting of list elements is required !**



(a) Before add

```
pNew->link  = list.head
list.head   = pNew
```

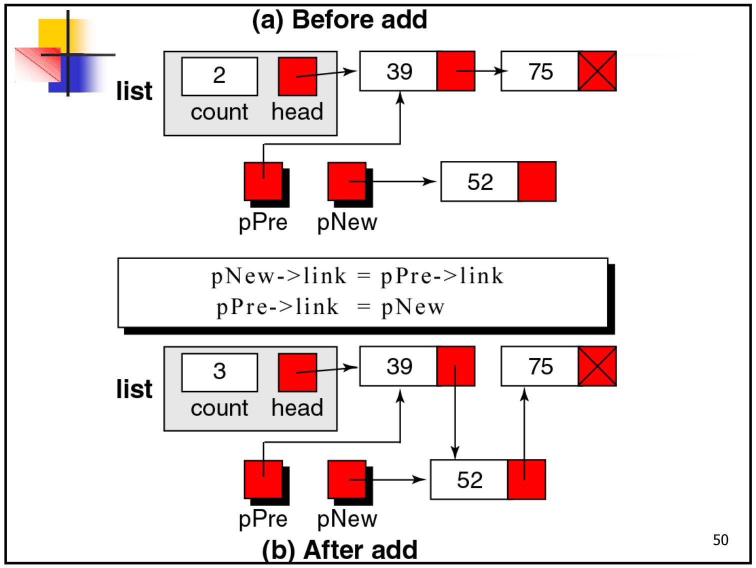
(b) After add



(a) Before add

```
pNew->link = list.head
list.head    = pNew
```

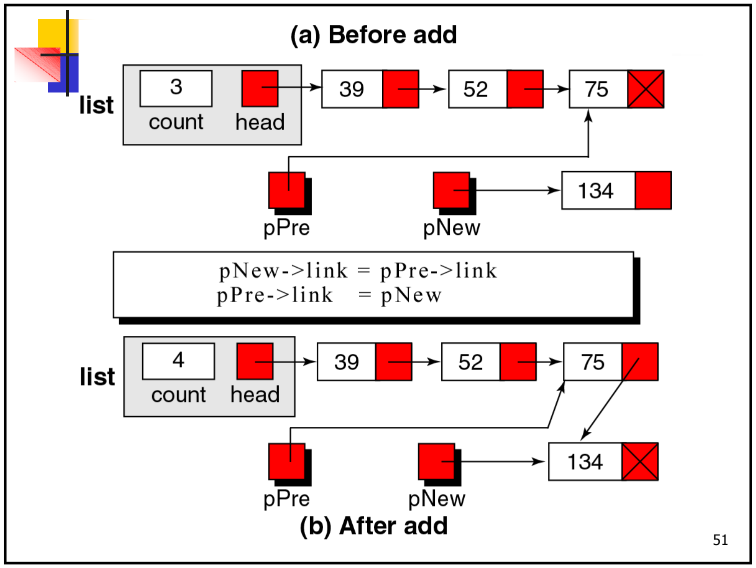
(b) After add



(a) Before add

```
pNew->link = pPre->link
pPre->link  = pNew
```

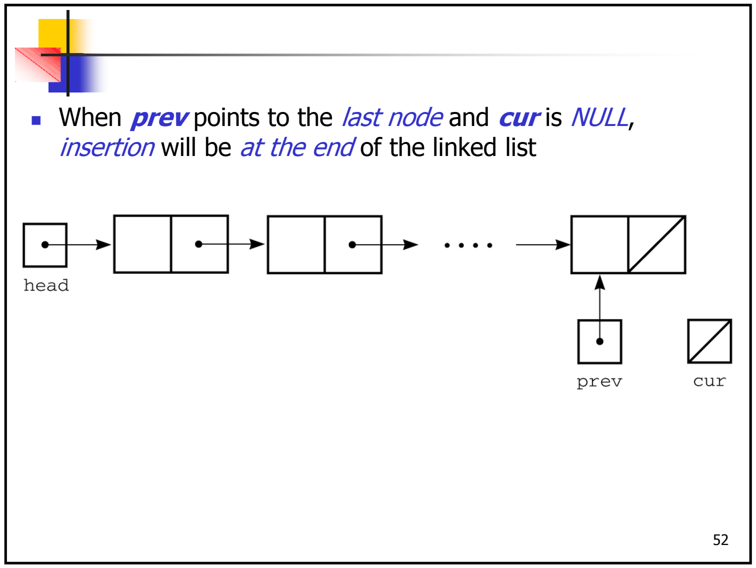
(b) After add



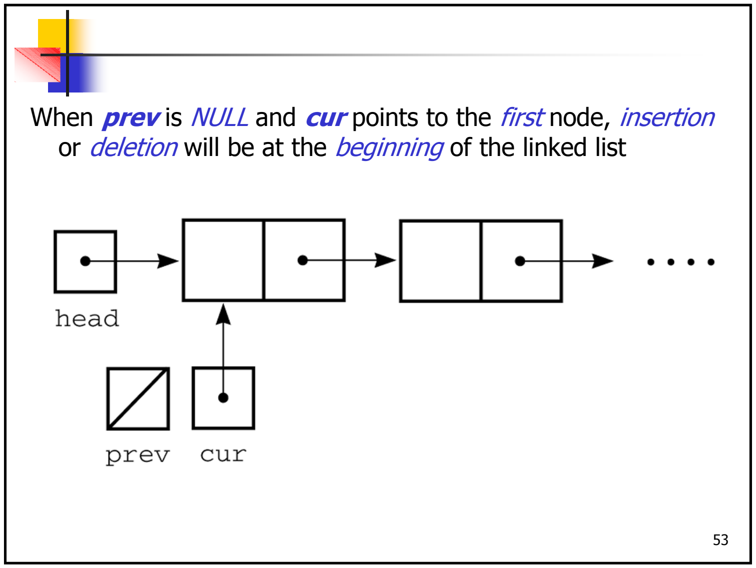
(a) Before add

```
pNew->link = pPre->link
pPre->link   = pNew
```

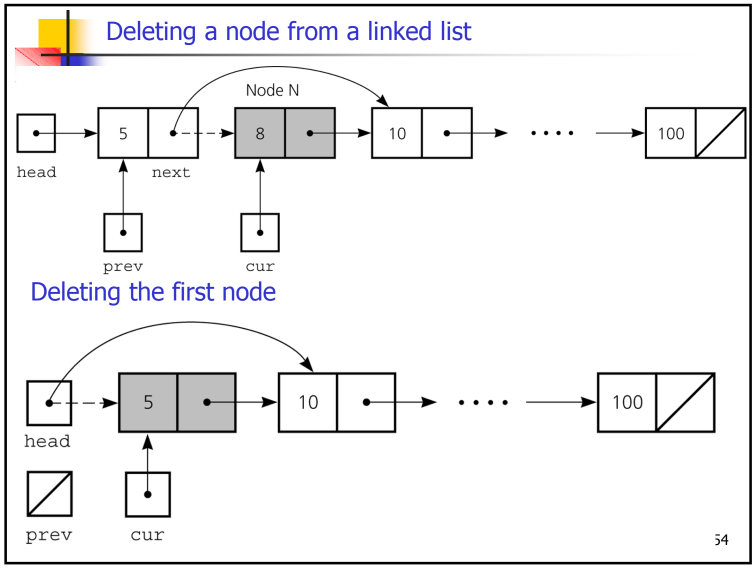
(b) After add



- When ***prev*** points to the *last node* and ***cur*** is *NULL*, *insertion* will be *at the end* of the linked list

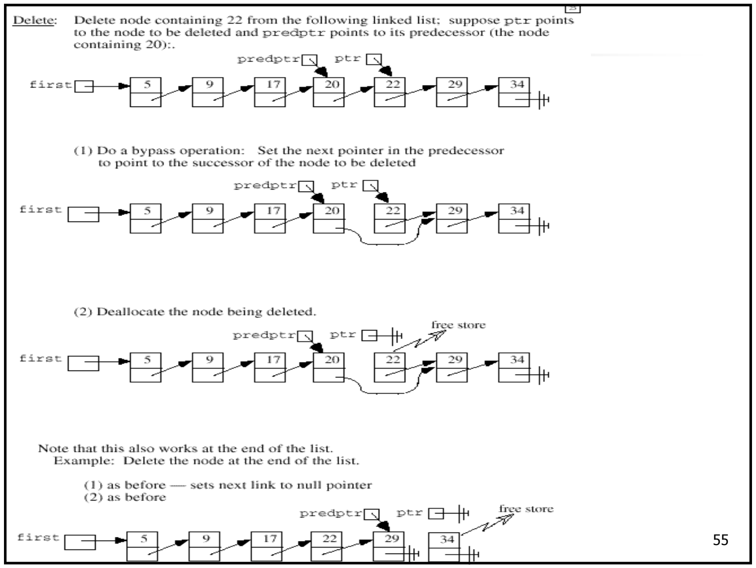


When ***prev*** is *NULL* and ***cur*** points to the *first* node, *insertion* or *deletion* will be at the *beginning* of the linked list



Deleting a node from a linked list

Deleting the first node



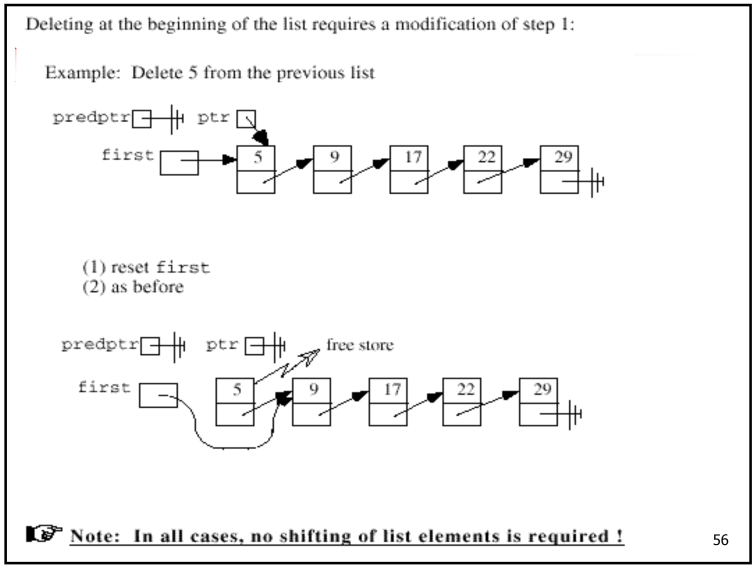
Delete: Delete node containing 22 from the following linked list; suppose ptr points to the node to be deleted and predptr points to its predecessor (the node containing 20):.

(1) Do a bypass operation: Set the next pointer in the predecessor to point to the successor of the node to be deleted

(2) Deallocate the node being deleted.

Note that this also works at the end of the list.
Example: Delete the node at the end of the list.

(1) as before — sets next link to null pointer
(2) as before

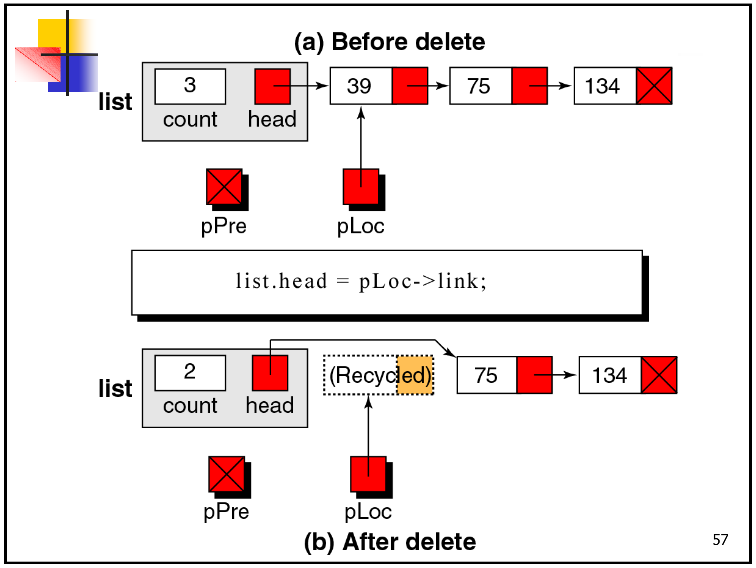


Deleting at the beginning of the list requires a modification of step 1:

Example: Delete 5 from the previous list

(1) reset first
(2) as before

☞ **Note: In all cases, no shifting of list elements is required !**



(a) Before delete

list

3 count

head

39

75

134

pPre

pLoc

```
list.head = pLoc->link;
```

list

2 count

head

(Recycled)

75

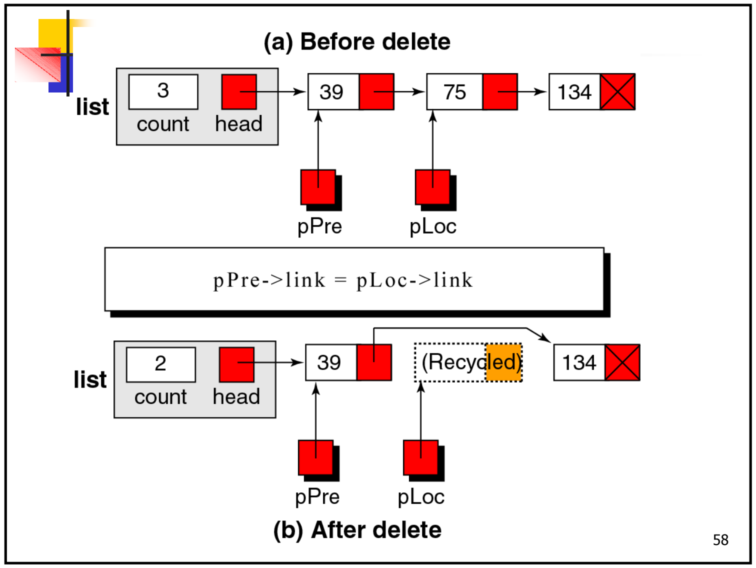
134

pPre

pLoc

(b) After delete

57



(a) Before delete

list

3 count

head

39

75

134

pPre

pLoc

```
pPre->link = pLoc->link
```

list

2 count

head

39

(Recycled)

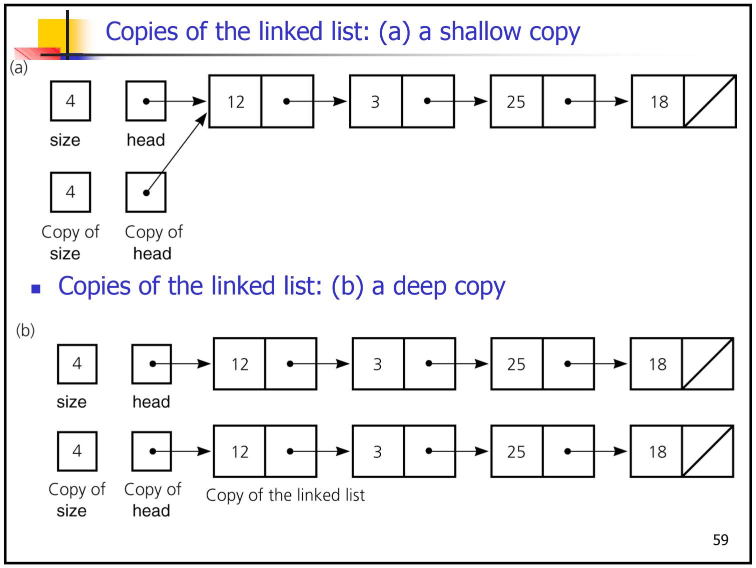
134

pPre

pLoc

(b) After delete

58



Copies of the linked list: (a) a shallow copy

(a)

4 size

head

12

3

25

18

4 Copy of size

Copy of head

- Copies of the linked list: (b) a deep copy

(b)

4 size

head

12

3

25

18

4 Copy of size

Copy of head

12

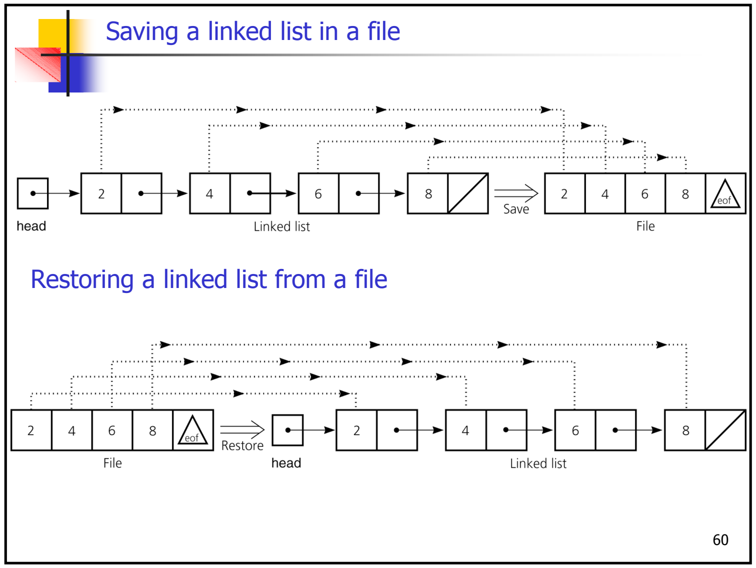
3

25

18

Copy of the linked list

59



Saving a linked list in a file

head

2

4

6

8

Linked list

Save

2 4 6 8 eof

File

Restoring a linked list from a file

2 4 6 8 eof

File

Restore

head

2

4

6

8

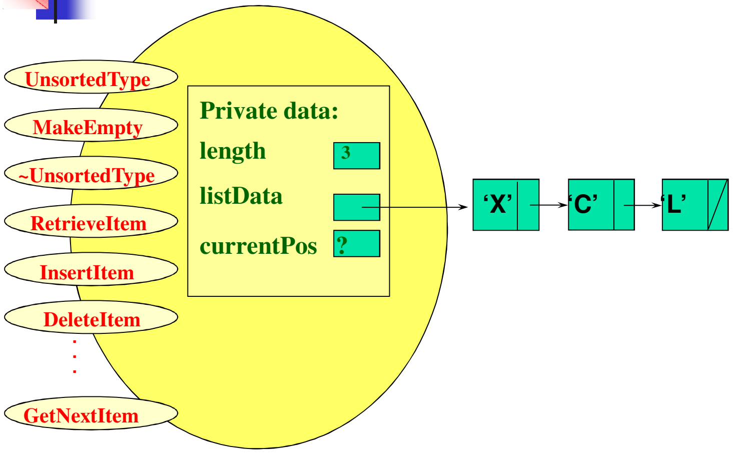
Linked list

60

```
template <class ItemType>
class  UnsortedType
{
public :                    //  LINKED LIST IMPLEMENTATION
   UnsortedType ( ) ;
   ~UnsortedType ( ) ;
   void    MakeEmpty ( ) ;
   bool    IsFull ( )   const ;
   int     LengthIs ( )   const ;
   void    RetrieveItem  ( ItemType&   item, bool&   found )
   ;
   void    InsertItem  ( ItemType   item ) ;
   void    DeleteItem  ( ItemType   item ) ;
   void    ResetList  ( );
   void    GetNextItem  ( ItemType&   item ) ;
private :
   NodeType<ItemType>*  listData;
   int                   length;
   NodeType<ItemType>*  currentPos;
} ;
```

```
template<class ItemType>
struct NodeType;


template <class ItemType>
struct NodeType
{
   ItemType info;
   NodeType* next;
};
```

class UnsortedType<char>



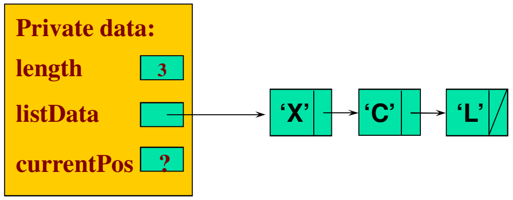
```
// LINKED LIST IMPLEMENTATION  ( unsorted.cpp )
template <class ItemType>
UnsortedType<ItemType>::UnsortedType( )  // constructor
//  Pre: None.
// Post: List is empty.
{
  length  =  0 ;
  listData = NULL;
}

template <class ItemType>
int  UnsortedType<ItemType>::LengthIs(  )  const
// Post:  Function value = number of items in the list.
{
  return  length;
}
```

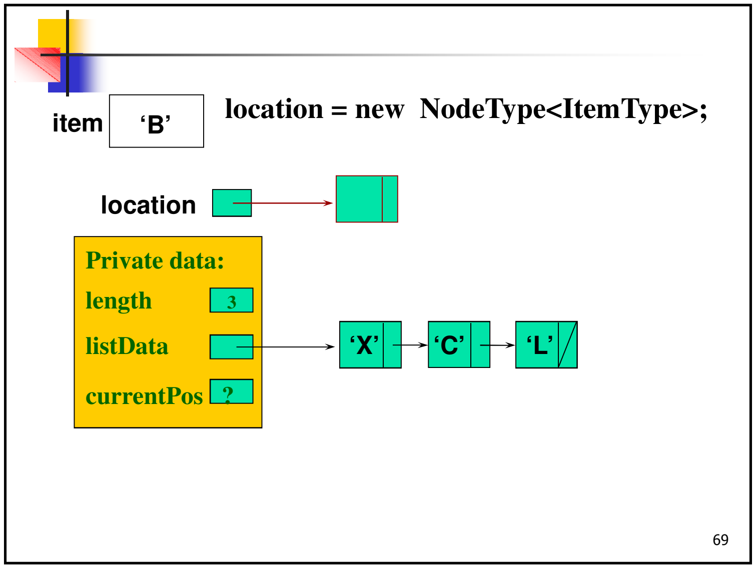
```
template <class ItemType>
bool ListType<ItemType>::IsFull() const
// Returns true if there is no room for another ItemType on
// the free store; false otherwise.
{
    NodeType<ItemType>* ptr;
    ptr = new NodeType<ItemType>;
    if (ptr == NULL)
          return true;
    else
    {
          delete ptr;
          return false;
    }
}
template <class ItemType>
void ListType<ItemType>::MakeEmpty()
// Post: List is empty; all items have been deallocated.
{
    NodeType<ItemType>* tempPtr;
    while (listData != NULL)
    {
          tempPtr = listData;
          listData = listData->next;
          delete tempPtr;
    }
    length = 0;
}
```

```
template <class ItemType>
void  UnsortedType<ItemType>::RetrieveItem( ItemType&  item, bool&
    found )
//  Pre: Key member of item is initialized.
// Post: If found, item's key matches an element's key in the list
//  and a copy of that element has been stored in item; otherwise,
// item is unchanged.
{  bool  moreToSearch ;
    NodeType<ItemType>*  location ;
   location = listData ;
    found = false ;
    moreToSearch = ( location  !=  NULL ) ;
    while ( moreToSearch  &&  !found )
    {  if ( item  == location->info )           // match here
      {  found = true ;
           item  = location->info ;
       }
       else                                      // advance pointer
       {  location = location->next ;
           moreToSearch = ( location  !=  NULL ) ;
        }
      }
}
```

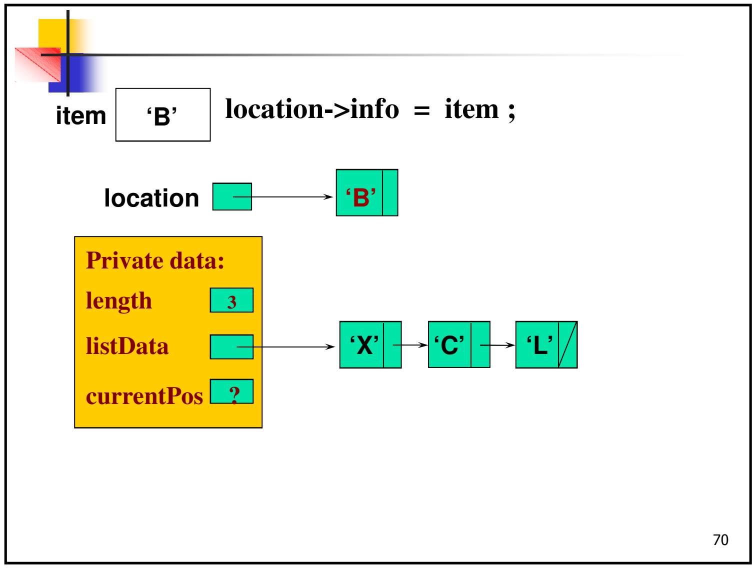
```
template <class ItemType>
void UnsortedType<ItemType>::InsertItem ( ItemType  item )
//  Pre: list is not full and item is not in list.
// Post: item is in the list; length has been incremented.
{
    NodeType<ItemType>*  location ;
    // obtain and fill a node
    location = new  NodeType<ItemType> ;
    location->info = item ;
    location->next = listData ;
    listData = location ;
    length++ ;
}
```

Inserting 'B' into an Unsorted List

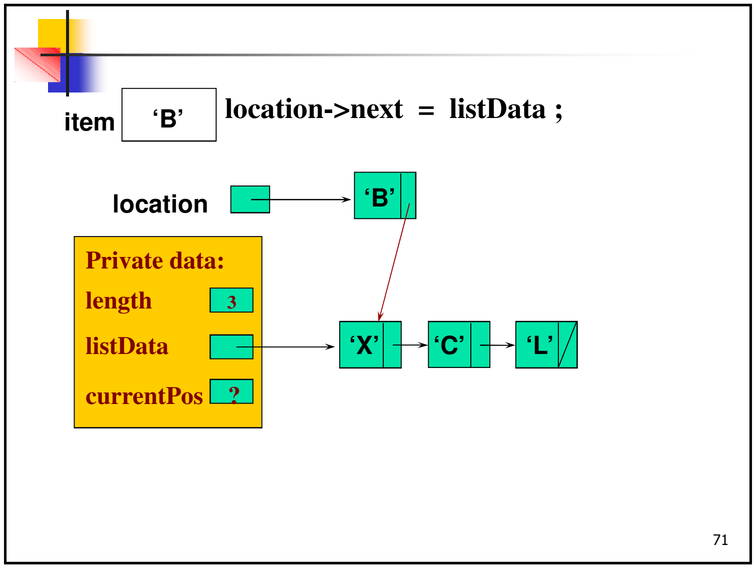




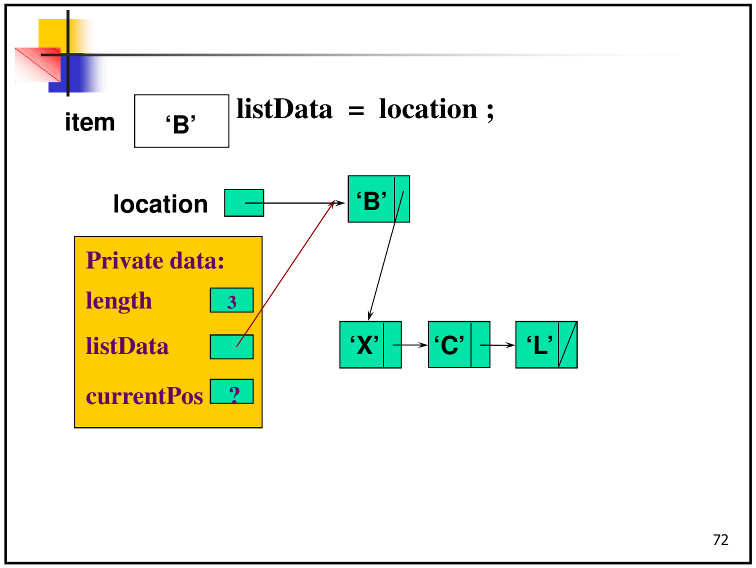
location = new NodeType<ItemType>;



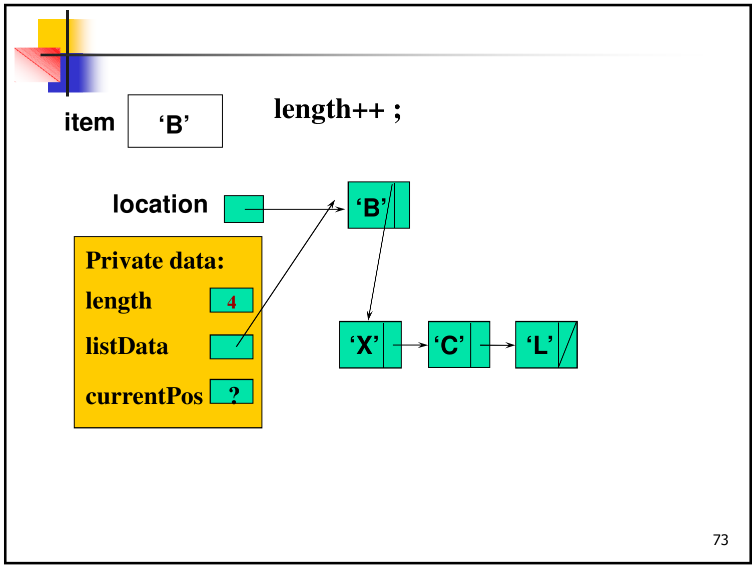
location->info = item ;



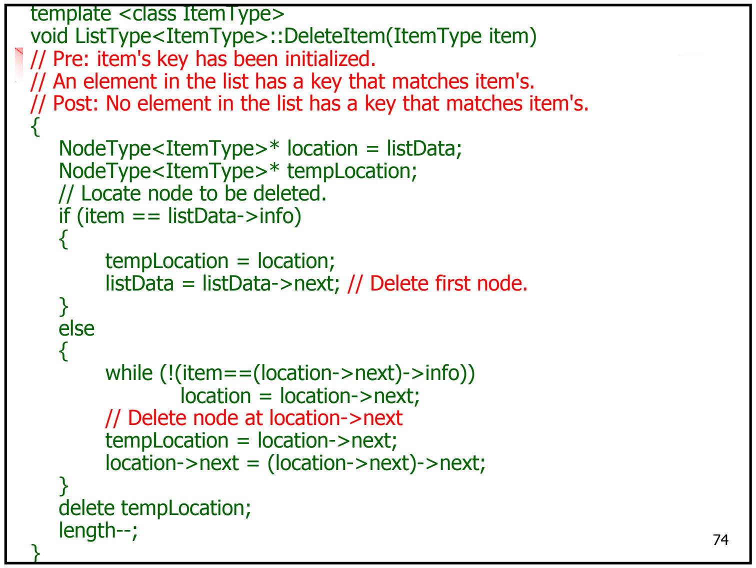
location->next = listData ;



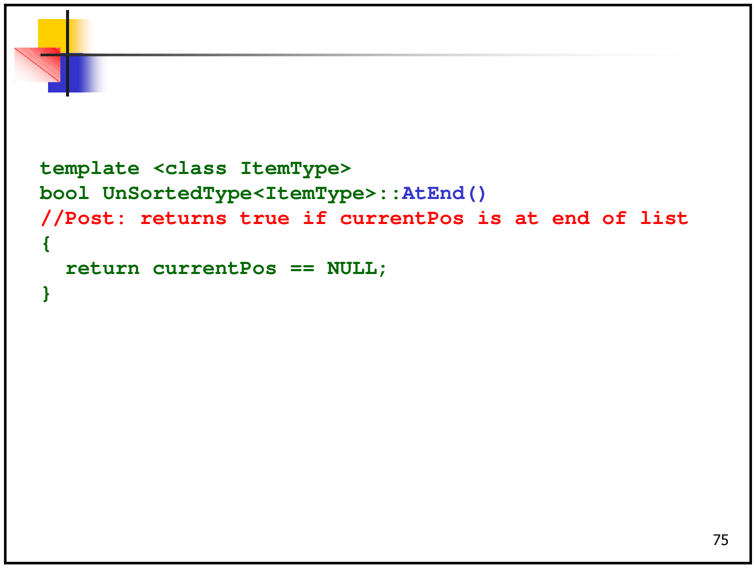
listData = location ;



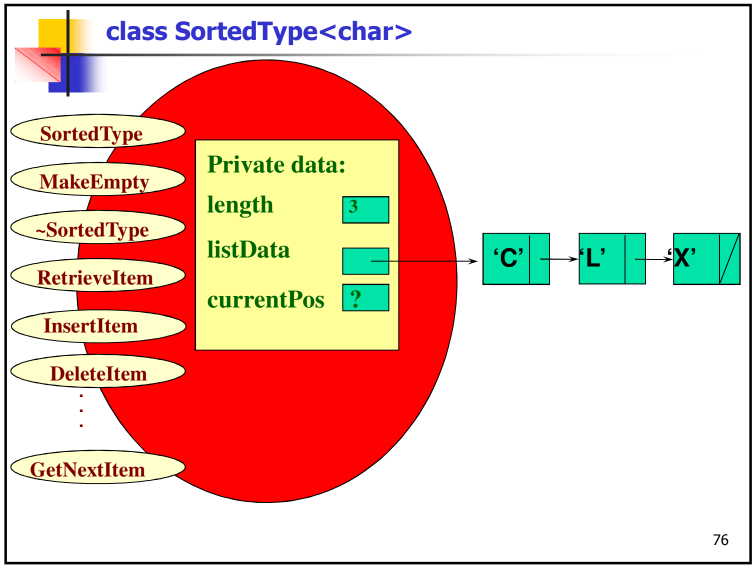
length++ ;



```
template <class ItemType>
void ListType<ItemType>::DeleteItem(ItemType item)
// Pre: item's key has been initialized.
// An element in the list has a key that matches item's.
// Post: No element in the list has a key that matches item's.
{
    NodeType<ItemType>* location = listData;
    NodeType<ItemType>* tempLocation;
    // Locate node to be deleted.
    if (item == listData->info)
    {
        tempLocation = location;
        listData = listData->next; // Delete first node.
    }
    else
    {
        while (!(item==(location->next)->info))
            location = location->next;
        // Delete node at location->next
        tempLocation = location->next;
        location->next = (location->next)->next;
    }
    delete tempLocation;
    length--;
}
```



```
template <class ItemType>
bool UnSortedType<ItemType>::AtEnd()
//Post: returns true if currentPos is at end of list
{
  return currentPos == NULL;
}
```



class SortedType<char>

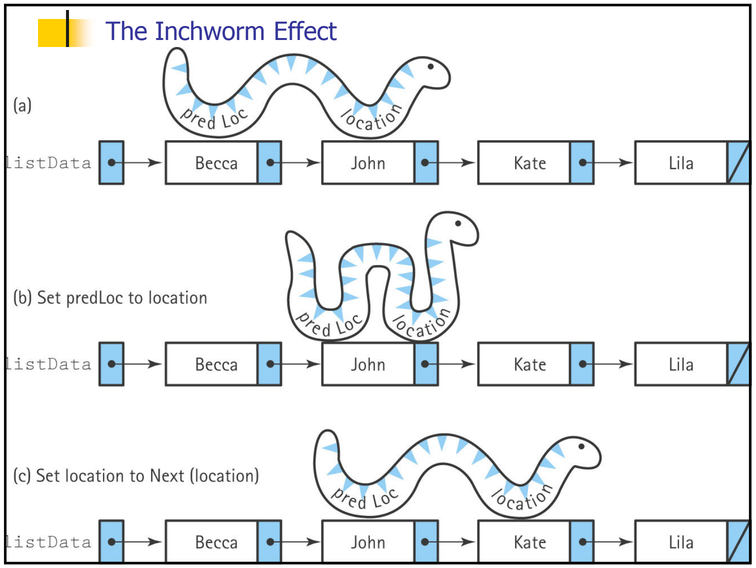
Ordered Link List

- In an ordered linked list the *elements are sorted*
- Because the list is ordered, we need to *modify* the algorithms (from how they were implemented for the regular linked list) for the *search, insert*, and *delete* operations

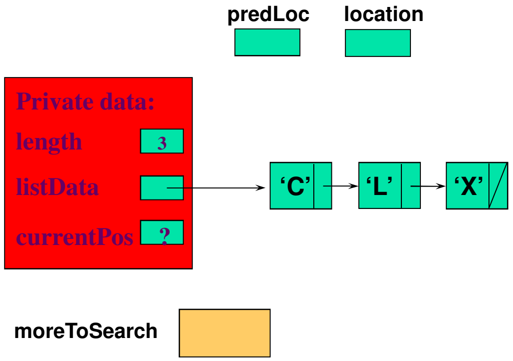
Implementing *SortedType* member function *InsertItem*

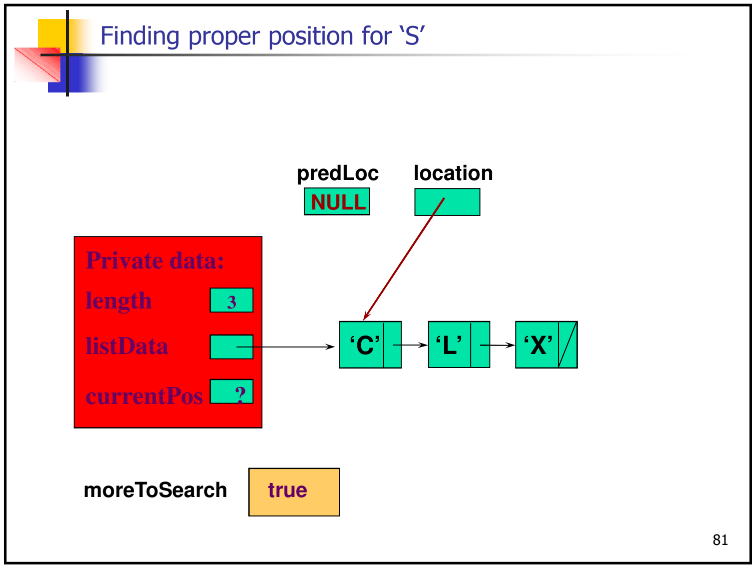
```
// LINKED LIST IMPLEMENTATION            (sorted.cpp)

template <class ItemType>
void  SortedType<ItemType> :: InsertItem ( ItemType  item )
//  Pre: List has been initialized. List is not full.
//  item is not in list.
//  List is sorted by key member.
// Post:  item is in the list.  List is still sorted.
{
          .
          .
          .

}
```

The Inchworm Effect



Inserting 'S' into a Sorted List





Finding proper position for ‘S’

predLoc

NULL

location

Private data:

length 3

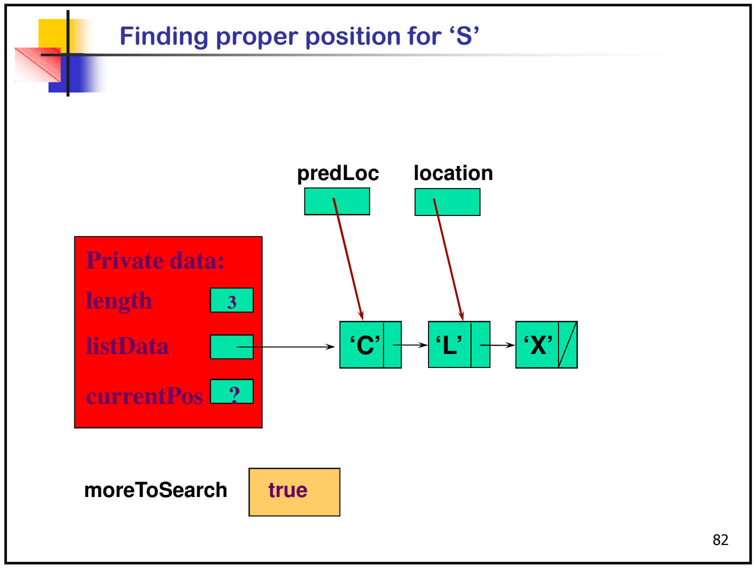
listData

currentPos ?

‘C’ ‘L’ ‘X’

moreToSearch true

81



Finding proper position for ‘S’

predLoc

location

Private data:

length 3

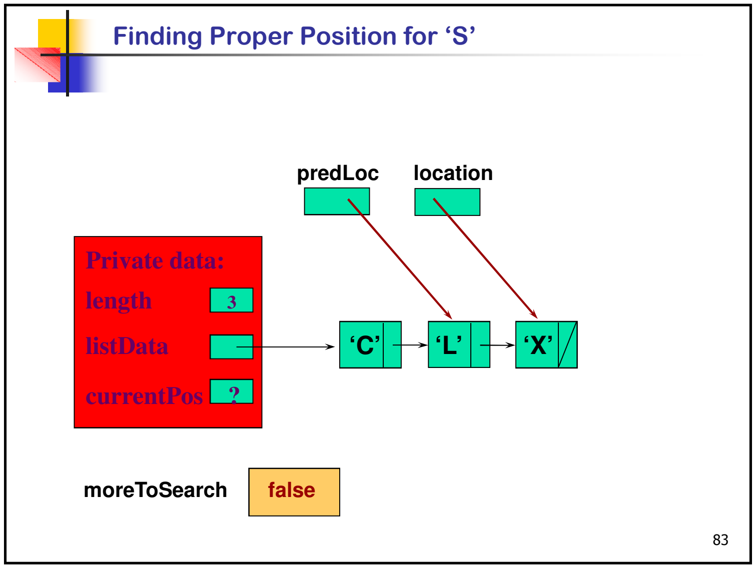
listData

currentPos ?

‘C’ ‘L’ ‘X’

moreToSearch true

82



Finding Proper Position for ‘S’

predLoc

location

Private data:

length 3

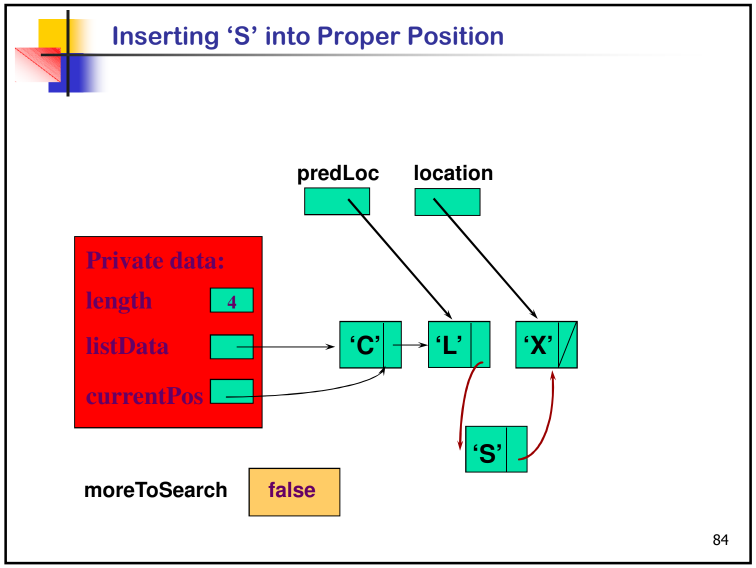
listData

currentPos ?

‘C’ ‘L’ ‘X’

moreToSearch false

83



Inserting ‘S’ into Proper Position

predLoc

location

Private data:

length 4

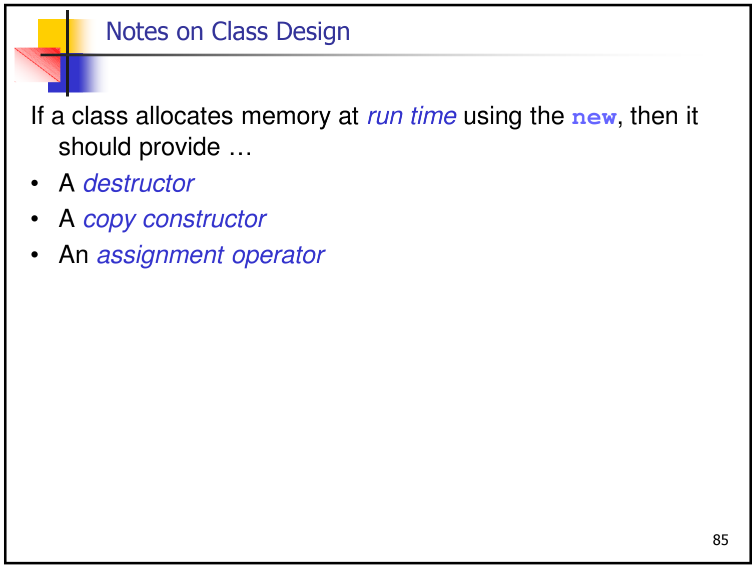
listData

currentPos

‘C’ ‘L’ ‘X’ ‘S’

moreToSearch false

84

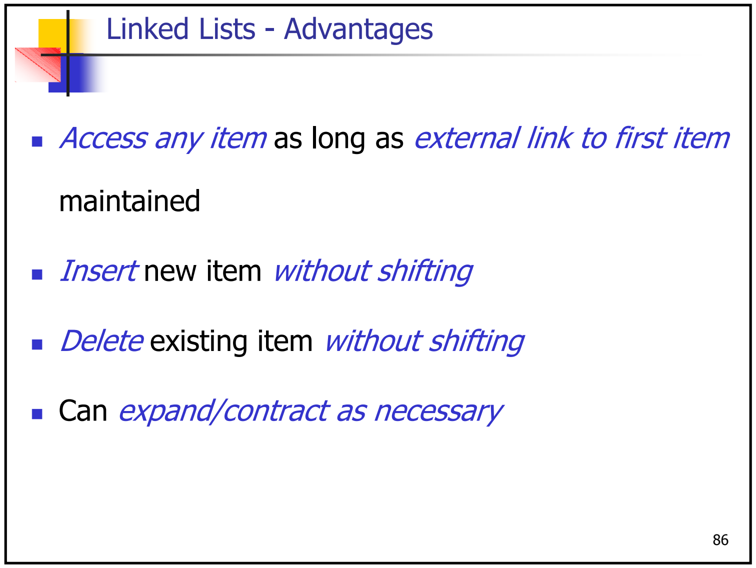


Notes on Class Design

If a class allocates memory at *run time* using the **`new`**, then it should provide …

- A *destructor*
- A *copy constructor*
- An *assignment operator*

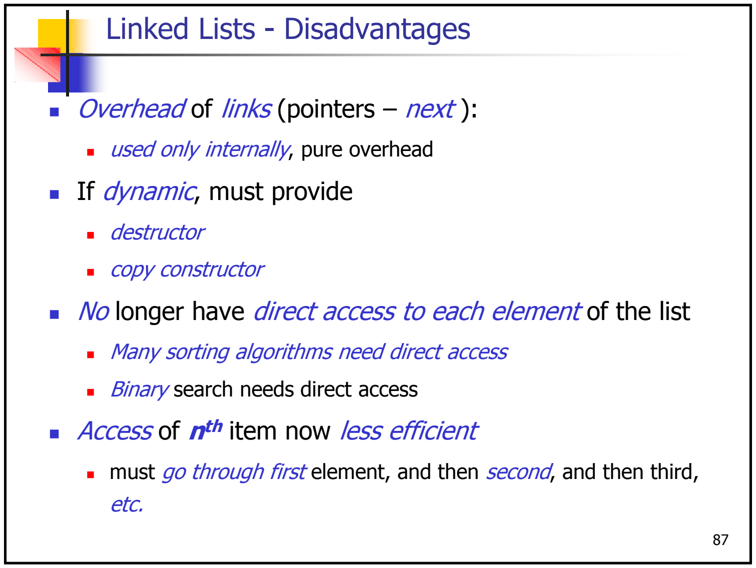
85



Linked Lists - Advantages

- *Access any item* as long as *external link to first item* maintained
- *Insert* new item *without shifting*
- *Delete* existing item *without shifting*
- Can *expand/contract as necessary*

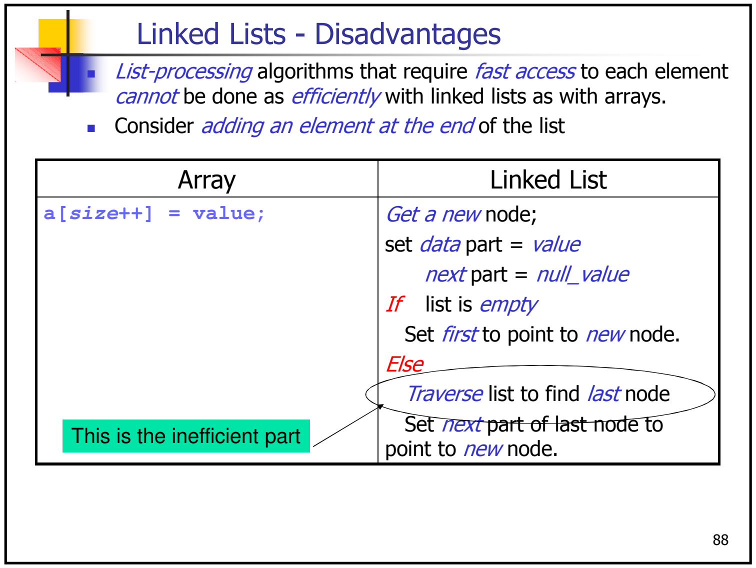
86



Linked Lists - Disadvantages

- *Overhead* of *links* (pointers – *next*):
  - *used only internally*, pure overhead
- If *dynamic*, must provide
  - *destructor*
  - *copy constructor*
- *No* longer have *direct access to each element* of the list
  - *Many sorting algorithms need direct access*
  - *Binary* search needs direct access
- *Access* of ***n^{th}*** item now *less efficient*
  - must *go through first* element, and then *second*, and then third, *etc.*

87



Linked Lists - Disadvantages

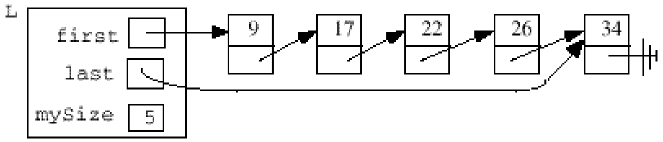
- *List-processing* algorithms that require *fast access* to each element *cannot* be done as *efficiently* with linked lists as with arrays.
- Consider *adding an element at the end* of the list

Array	Linked List
`a[size++] = value;` This is the inefficient part	*Get a new* node; set *data* part = *value* *next* part = *null_value* *If* list is *empty* Set *first* to point to *new* node. *Else* *Traverse* list to find *last* node Set *next* part of last node to point to *new* node.

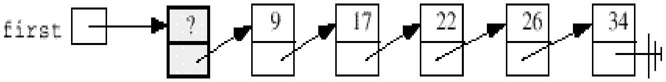
88

Other Kinds of Linked Lists

In some applications, it is convenient to *keep access* to both the *first* node and the *last* node in the list

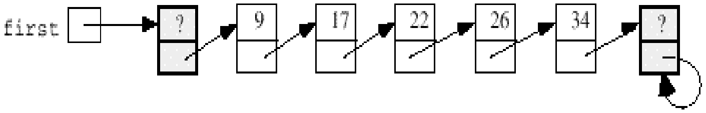


•Sometimes a ***head node*** is used so that ***every node has a predecessor***, which thus eliminates special cases for inserting and deleting.



The *data* part of the head node might be used to store some *information* about the list, as *number of values in the list*.

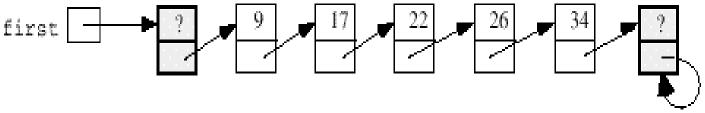
- Sometimes a **trailer node** is also used so that **every node has a successor**.



Two or more lists can *share the same trailer node*.

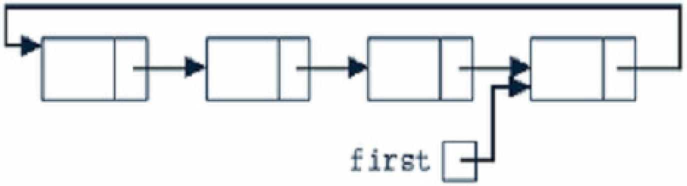
Linked Lists With Header and Trailer Nodes

- One way to *simplify insertion and deletion* is *never to insert* an item *before the first* or *after the last* item and *never to delete* the *first* node
- You can set a *header node* at the beginning of the list containing a *value smaller than the smallest* value in the *data set*
- You can set a *trailer node* at the end of the list containing a *value larger than the largest* value in the *data set*
- These two nodes, *header* and *trailer*, serve merely to *simplify the insertion and deletion* algorithms and are *not part of the actual list*.
- The actual list is between these two nodes.



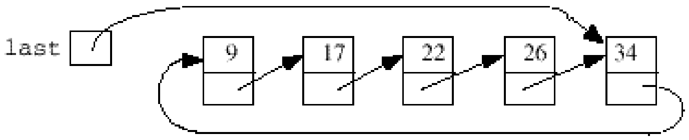
Circular Linked List

- A linked list in which the *last node points to the first node* is called a circular linked list
- In a circular linked list with more than one node, it is *convenient* to make the *pointer first point to the last node* of the list



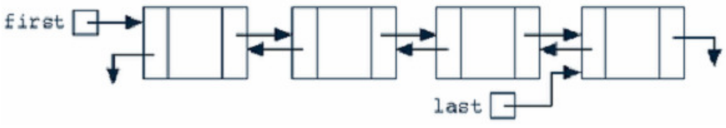
❖In other applications (e.g., *linked queues*), a ***circular linked list*** is used; *instead* of the *last node* containing a *NULL* pointer, it contains a *pointer to the first node* in the list.

❖For such lists, one can use a *single pointer to the last node* in the list, because then one has *direct access* to it and *"almost-direct"* access to the *first* node.



Doubly Linked List

- A doubly linked list is a linked list in which every *node* has a ***next*** pointer and a ***back*** pointer
- Every node *(except the last node)* contains the address of the *next* node, and every node *(except the first node)* contains the address of the *previous* node.
- A doubly linked list *can be traversed in either direction*

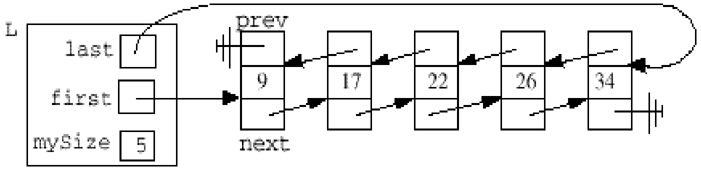


❖More lists, however, are *uni-directional;* we can only move *from one node to the next*.

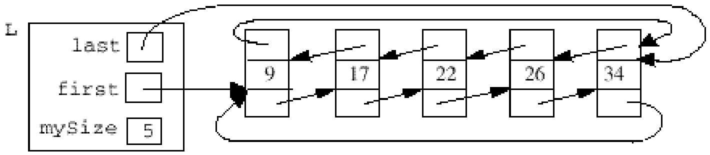
❖In many applications, *bidirectional movement* is necessary.

❖In this case, each *node* has *two pointers* — one to its *successor* (*null* if there is *none*) and one to its *predecessor* (*null* if there is *none.*)

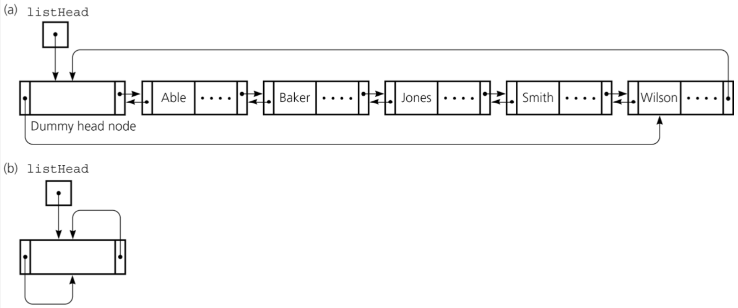
❖Such a list is commonly called a **doubly- linked** (or **symmetrically-linked**) **list**.



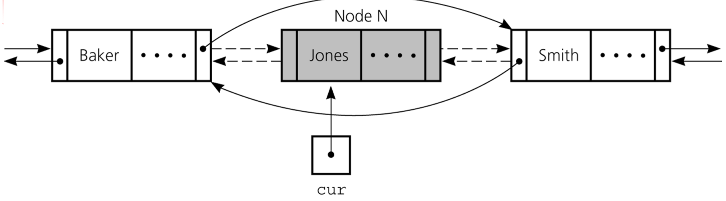
❖We could modify this *doubly-linked list* so that both lists are *circular* forming a **doubly-linked ring.**



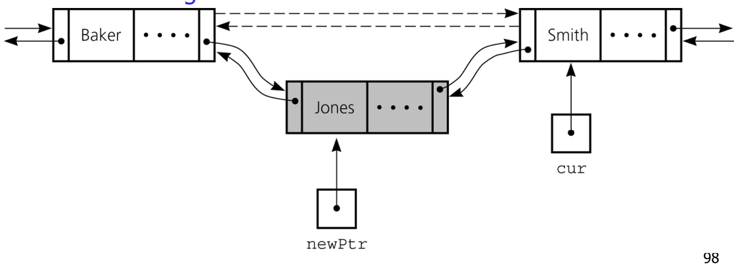
(a) A *circular doubly linked list* with a *dummy head* node;
(b) An *empty list* with a *dummy head* node



Pointer changes for deletion



- Pointer changes for insertion

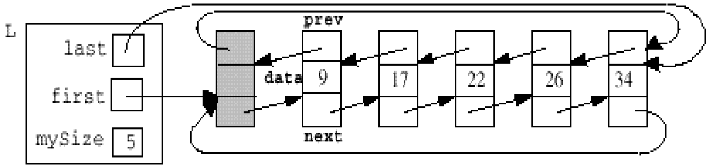


The STL list Class Template ***list*** is a *sequential container* that is *optimized for insertion and erasure at arbitrary points* in the sequence.

1. Implementation

As a ***circular doubly-linked list with head node.***
Its node structure is:

struct list_node
{
pointer next, prev;
T data;
}



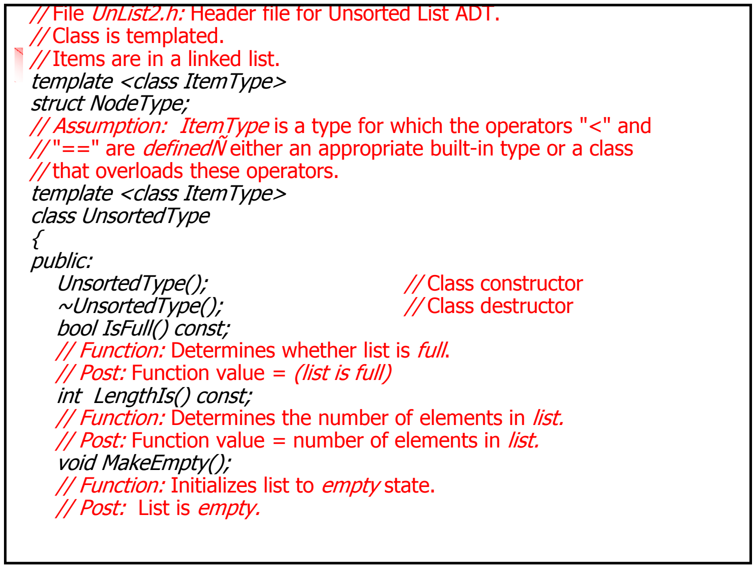
References

```
#include  "ItemType.h"              //  unsorted.h
   . . .
template <class ItemType>
class  UnsortedType
{
public :                   //  LINKED LIST IMPLEMENTATION
    UnsortedType ( ) ;
    ~UnsortedType ( ) ;
    void        MakeEmpty ( ) ;
    bool        IsFull ( )  const ;
    int         LengthIs ( )  const ;
    void        RetrieveItem ( ItemType&  item, bool&  found ) ;
    void        InsertItem ( ItemType  item ) ;
    void        DeleteItem ( ItemType  item ) ;
    void        ResetList ( );
    void        GetNextItem ( ItemType&  item ) ;
private :
    NodeType<ItemType>*  listData;
    int     length;
    NodeType<ItemType>*  currentPos;
} ;
```

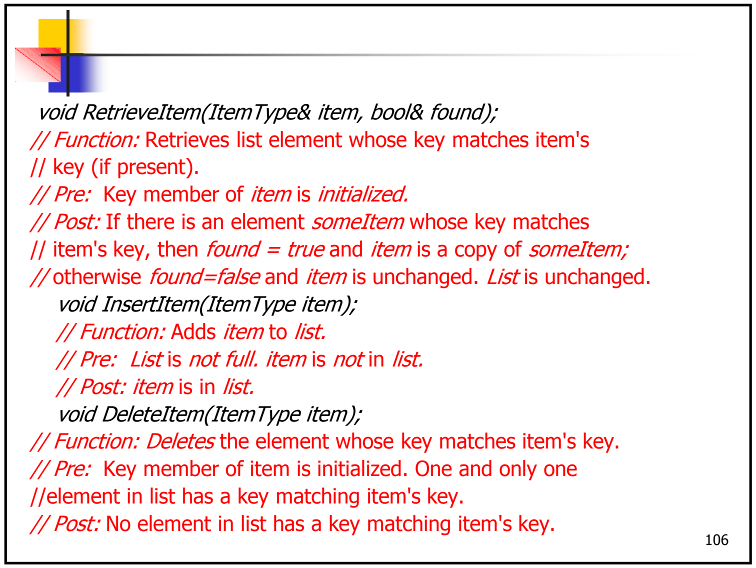
```
// LINKED LIST IMPLEMENTATION  ( unsorted.cpp )
#include "itemtype.h"
template <class ItemType>
UnsortedType<ItemType>::UnsortedType ( )      // constructor
//  Pre:  None.
// Post: List is empty.
{
    length  =  0 ;
    listData = NULL;
}
template <class ItemType>
int  UnsortedType<ItemType>::LengthIs (  )  const
// Post:  Function value = number of items in the list.
{
    return  length;
}
```

```
template <class ItemType>
void  UnsortedType<ItemType>::RetrieveItem( ItemType&  item, bool&  found )
//  Pre:   Key member of item is initialized.
// Post:  If found, item's key matches an element's key in the list and a copy
//           of that element has been stored in item; otherwise, item is
    unchanged.
{   bool  moreToSearch ;
    NodeType<ItemType>*  location ;
    location = listData ;
    found = false ;
    moreToSearch = ( location  !=  NULL ) ;
    while ( moreToSearch  &&  !found )
    {      if ( item  == location->info )           // match here
           {   found = true ;
               item  = location->info ;
           }
           else                                     // advance pointer
           {   location = location->next ;
               moreToSearch = ( location  !=  NULL ) ;
           }
    }
}
```

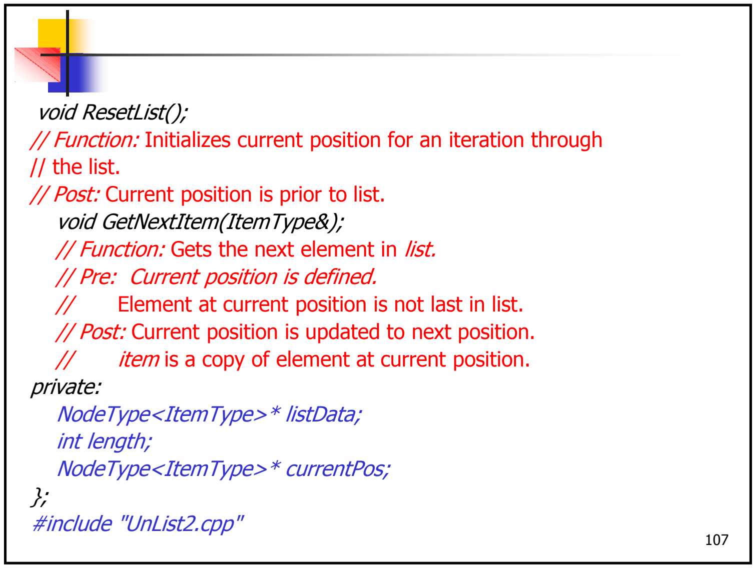
```
template <class ItemType>
void UnsortedType<ItemType>::InsertItem ( ItemType  item )
//  Pre:    list is not full and item is not in list.
// Post:   item is in the list;  length has been incremented.
{
    NodeType<ItemType>*  location ;
                                     // obtain and fill a node
    location = new  NodeType<ItemType> ;
    location->info = item ;
    location->next = listData ;
    listData = location ;
    length++ ;
}
```



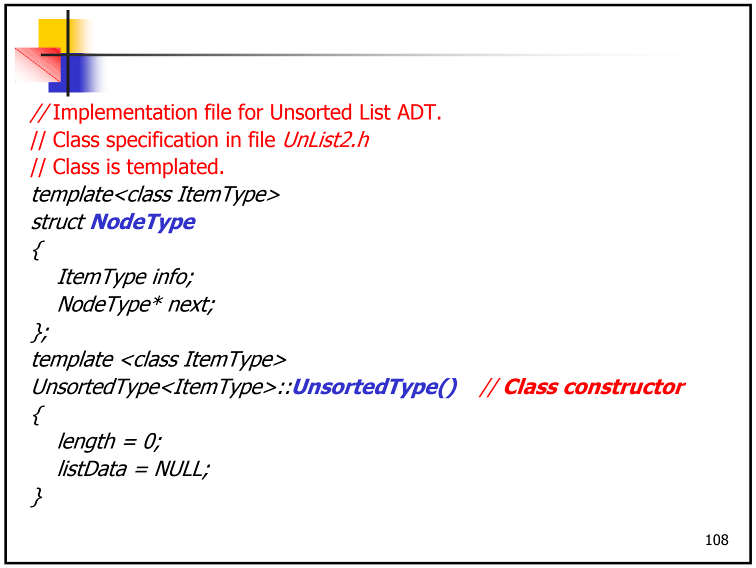
```
// File UnList2.h: Header file for Unsorted List ADT.
// Class is templated.
// Items are in a linked list.
template <class ItemType>
struct NodeType;
// Assumption:  ItemType is a type for which the operators "<" and
// "==" are definedÑ either an appropriate built-in type or a class
// that overloads these operators.
template <class ItemType>
class UnsortedType
{
public:
    UnsortedType();                       // Class constructor
    ~UnsortedType();                      // Class destructor
    bool IsFull() const;
    // Function: Determines whether list is full.
    // Post: Function value = (list is full)
    int  LengthIs() const;
    // Function: Determines the number of elements in list.
    // Post: Function value = number of elements in list.
    void MakeEmpty();
    // Function: Initializes list to empty state.
    // Post:  List is empty.
```



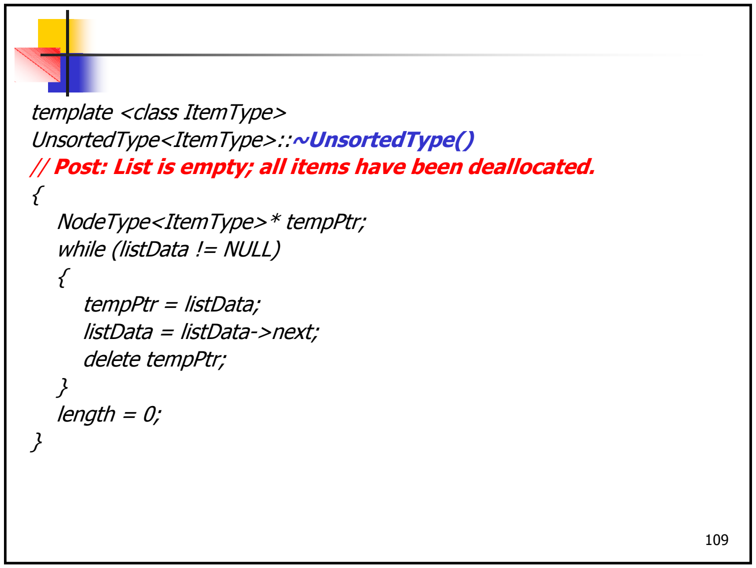
```
 void RetrieveItem(ItemType& item, bool& found);
// Function: Retrieves list element whose key matches item's
// key (if present).
// Pre:  Key member of item is initialized.
// Post: If there is an element someItem whose key matches
// item's key, then found = true and item is a copy of someItem;
// otherwise found=false and item is unchanged. List is unchanged.
    void InsertItem(ItemType item);
    // Function: Adds item to list.
    // Pre:  List is not full. item is not in list.
    // Post: item is in list.
    void DeleteItem(ItemType item);
// Function: Deletes the element whose key matches item's key.
// Pre:  Key member of item is initialized. One and only one
//element in list has a key matching item's key.
// Post: No element in list has a key matching item's key.
```



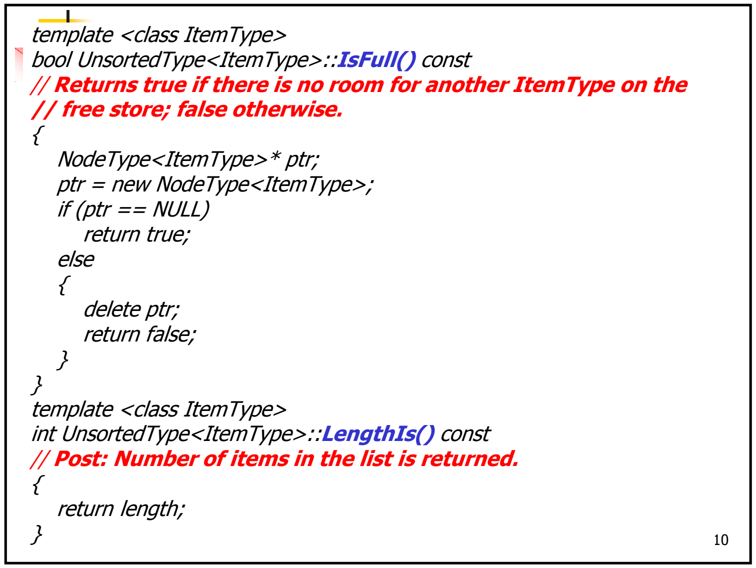
```
 void ResetList();
// Function: Initializes current position for an iteration through
// the list.
// Post: Current position is prior to list.
    void GetNextItem(ItemType&);
    // Function: Gets the next element in list.
    // Pre:  Current position is defined.
    //       Element at current position is not last in list.
    // Post: Current position is updated to next position.
    //       item is a copy of element at current position.
private:
    NodeType<ItemType>* listData;
    int length;
    NodeType<ItemType>* currentPos;
};
#include "UnList2.cpp"
```



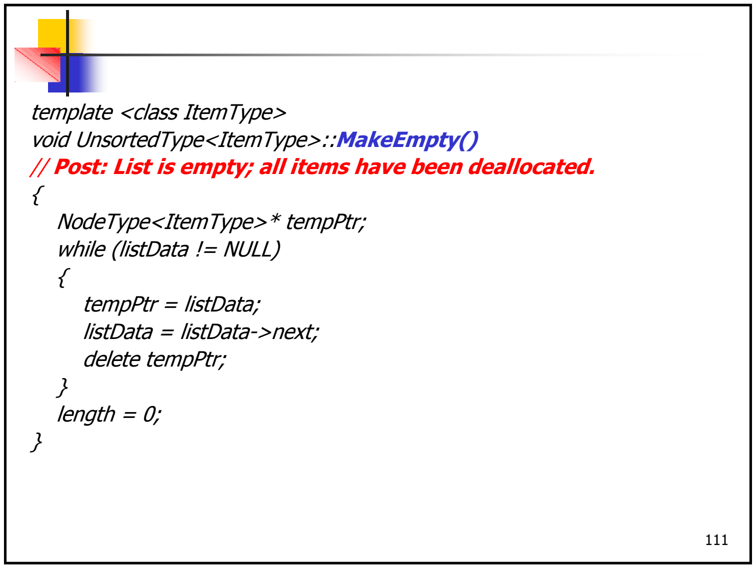
```
//Implementation file for Unsorted List ADT.
// Class specification in file UnList2.h
// Class is templated.
template<class ItemType>
struct NodeType
{
    ItemType info;
    NodeType* next;
};
template <class ItemType>
UnsortedType<ItemType>::UnsortedType()    // Class constructor
{
    length = 0;
    listData = NULL;
}
```



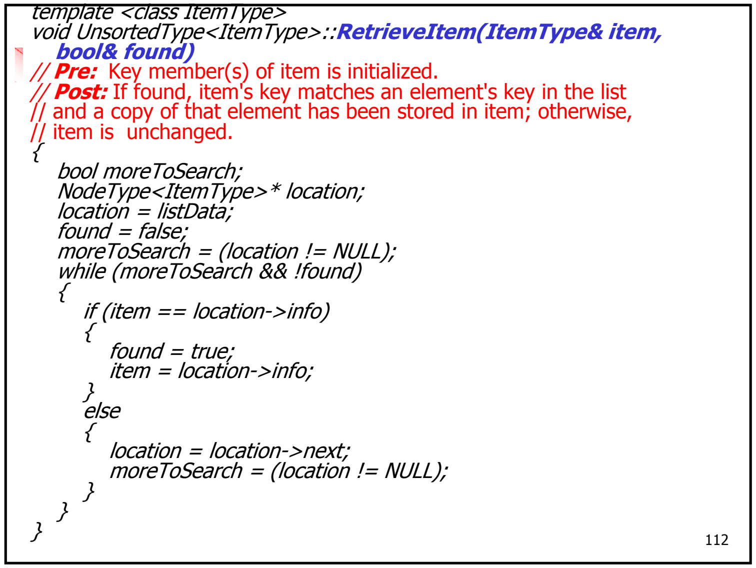
```
template <class ItemType>
UnsortedType<ItemType>::~UnsortedType()
// Post: List is empty; all items have been deallocated.
{
    NodeType<ItemType>* tempPtr;
    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}
```



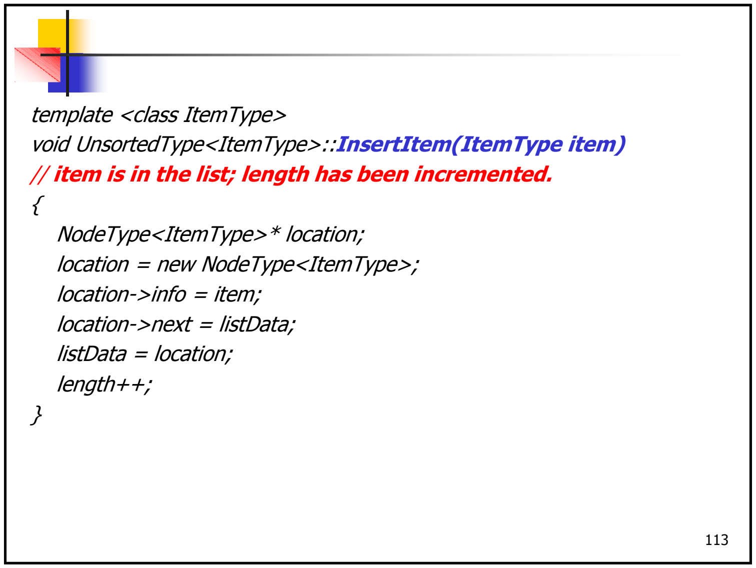
```
template <class ItemType>
bool UnsortedType<ItemType>::IsFull() const
// Returns true if there is no room for another ItemType on the
// free store; false otherwise.
{
    NodeType<ItemType>* ptr;
    ptr = new NodeType<ItemType>;
    if (ptr == NULL)
        return true;
    else
    {
        delete ptr;
        return false;
    }
}
template <class ItemType>
int UnsortedType<ItemType>::LengthIs() const
// Post: Number of items in the list is returned.
{
    return length;
}
```



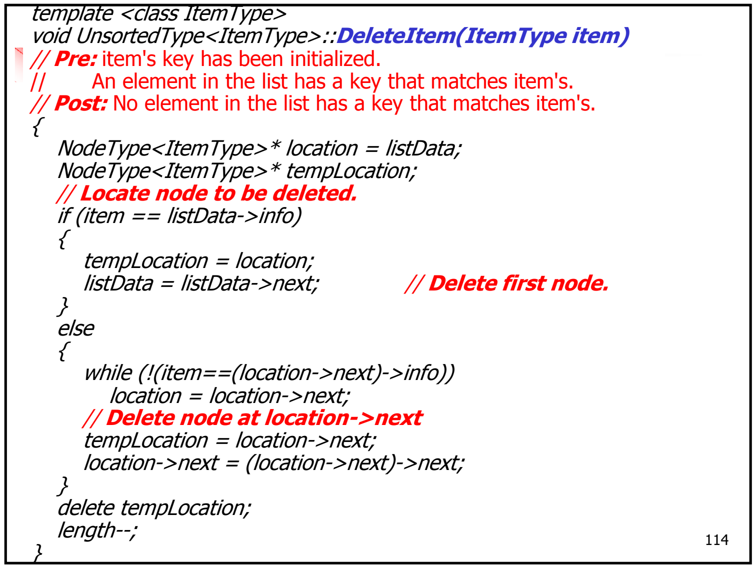
```
template <class ItemType>
void UnsortedType<ItemType>::MakeEmpty()
// Post: List is empty; all items have been deallocated.
{
    NodeType<ItemType>* tempPtr;
    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}
```



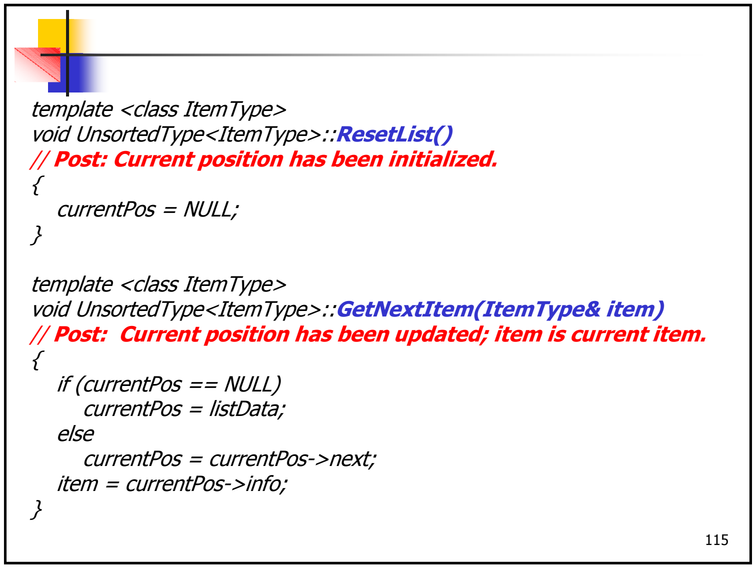
```
template <class ItemType>
void UnsortedType<ItemType>::RetrieveItem(ItemType& item,
    bool& found)
// Pre:  Key member(s) of item is initialized.
// Post: If found, item's key matches an element's key in the list
// and a copy of that element has been stored in item; otherwise,
// item is  unchanged.
{
    bool moreToSearch;
    NodeType<ItemType>* location;
    location = listData;
    found = false;
    moreToSearch = (location != NULL);
    while (moreToSearch && !found)
    {
        if (item == location->info)
        {
            found = true;
            item = location->info;
        }
        else
        {
            location = location->next;
            moreToSearch = (location != NULL);
        }
    }
}
```



```
template <class ItemType>
void UnsortedType<ItemType>::InsertItem(ItemType item)
// item is in the list; length has been incremented.
{
   NodeType<ItemType>* location;
   location = new NodeType<ItemType>;
   location->info = item;
   location->next = listData;
   listData = location;
   length++;
}
```



```
template <class ItemType>
void UnsortedType<ItemType>::DeleteItem(ItemType item)
// Pre: item's key has been initialized.
//       An element in the list has a key that matches item's.
// Post: No element in the list has a key that matches item's.
{
   NodeType<ItemType>* location = listData;
   NodeType<ItemType>* tempLocation;
   // Locate node to be deleted.
   if (item == listData->info)
   {
      tempLocation = location;
      listData = listData->next;            // Delete first node.
   }
   else
   {
      while (!(item==(location->next)->info))
         location = location->next;
      // Delete node at location->next
      tempLocation = location->next;
      location->next = (location->next)->next;
   }
   delete tempLocation;
   length--;
}
```



```
template <class ItemType>
void UnsortedType<ItemType>::ResetList()
// Post: Current position has been initialized.
{
   currentPos = NULL;
}

template <class ItemType>
void UnsortedType<ItemType>::GetNextItem(ItemType& item)
// Post:  Current position has been updated; item is current item.
{
   if (currentPos == NULL)
      currentPos = listData;
   else
      currentPos = currentPos->next;
   item = currentPos->info;
}
```