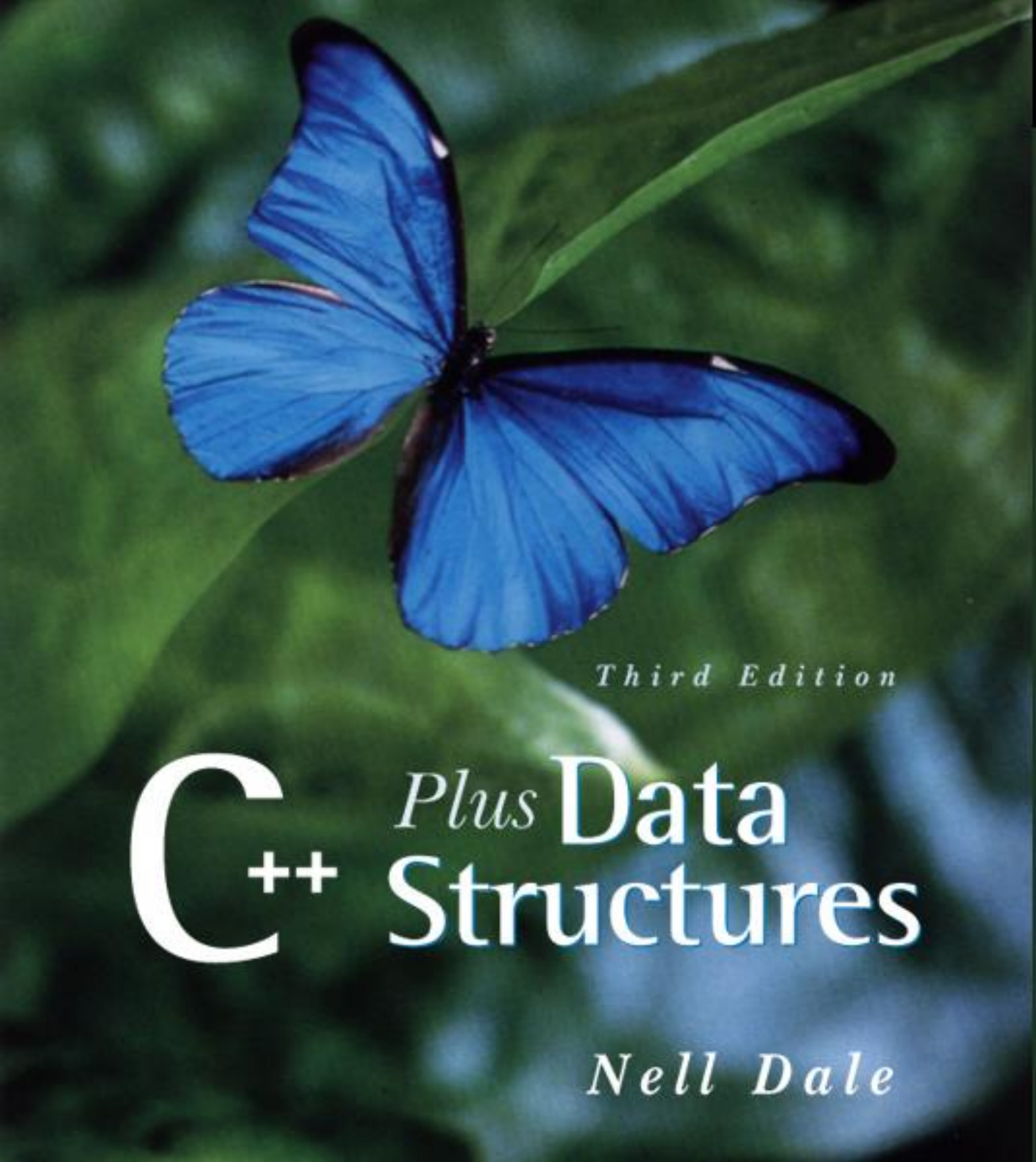


Chapter

6

Lists Plus

ADT Sorted List Operations

Transformers
- MakeEmpty
- InsertItem
- DeleteItem

Observers
- IsFull
- LengthIs
- RetrieveItem

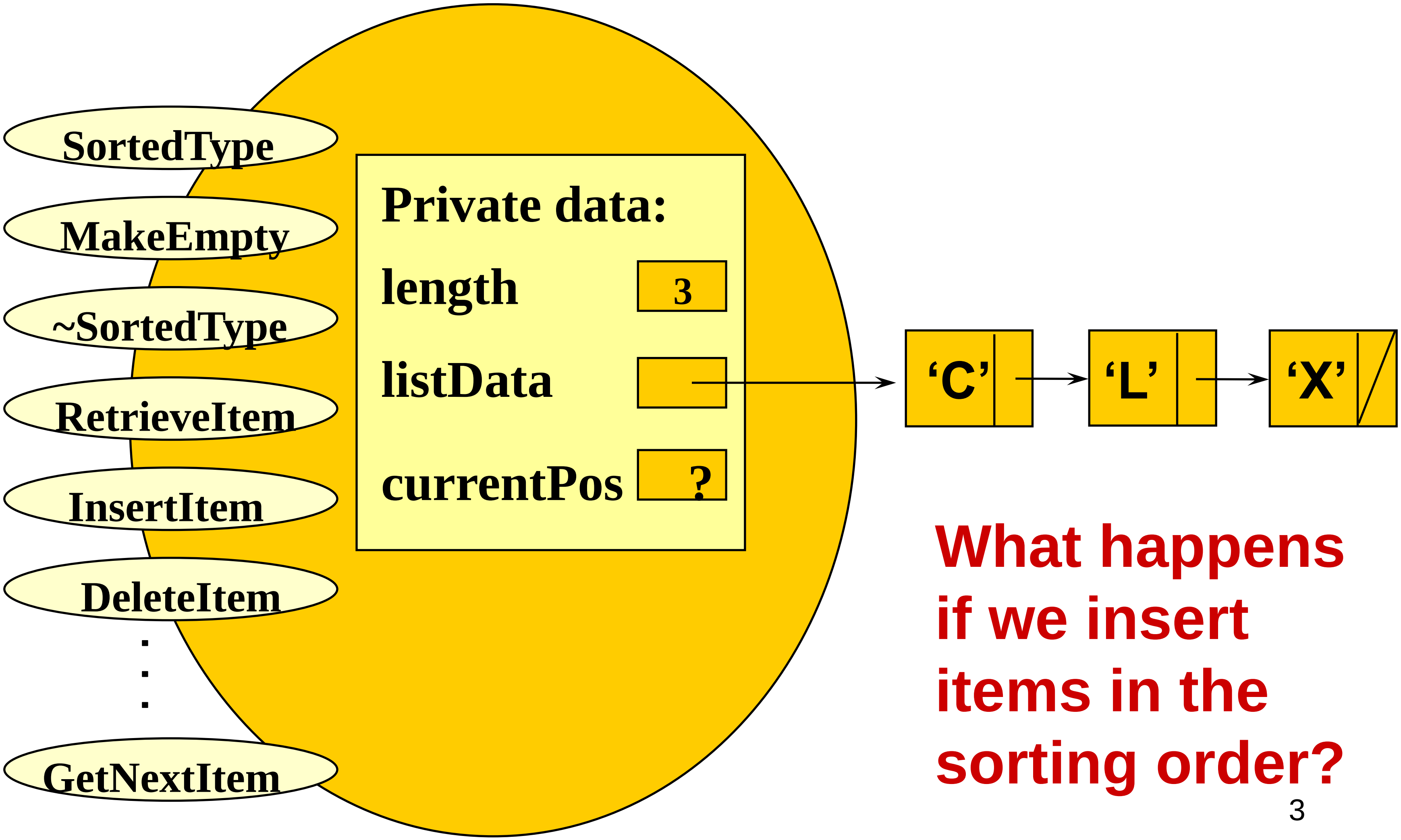
Iterators
- ResetList
- GetNextItem

change state

observe state

process all

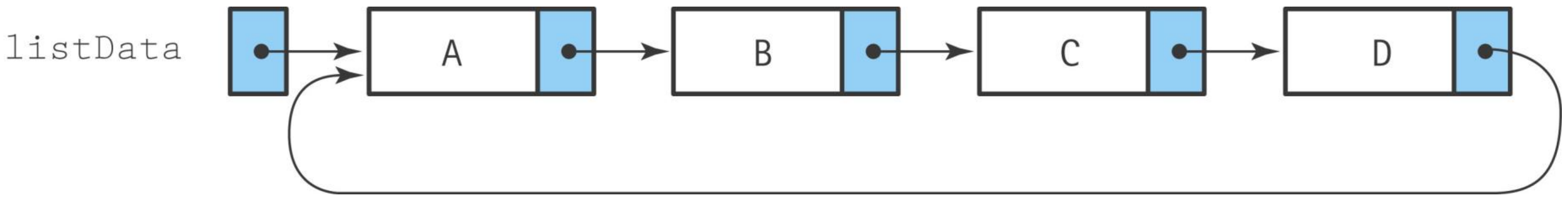
class SortedType<char>



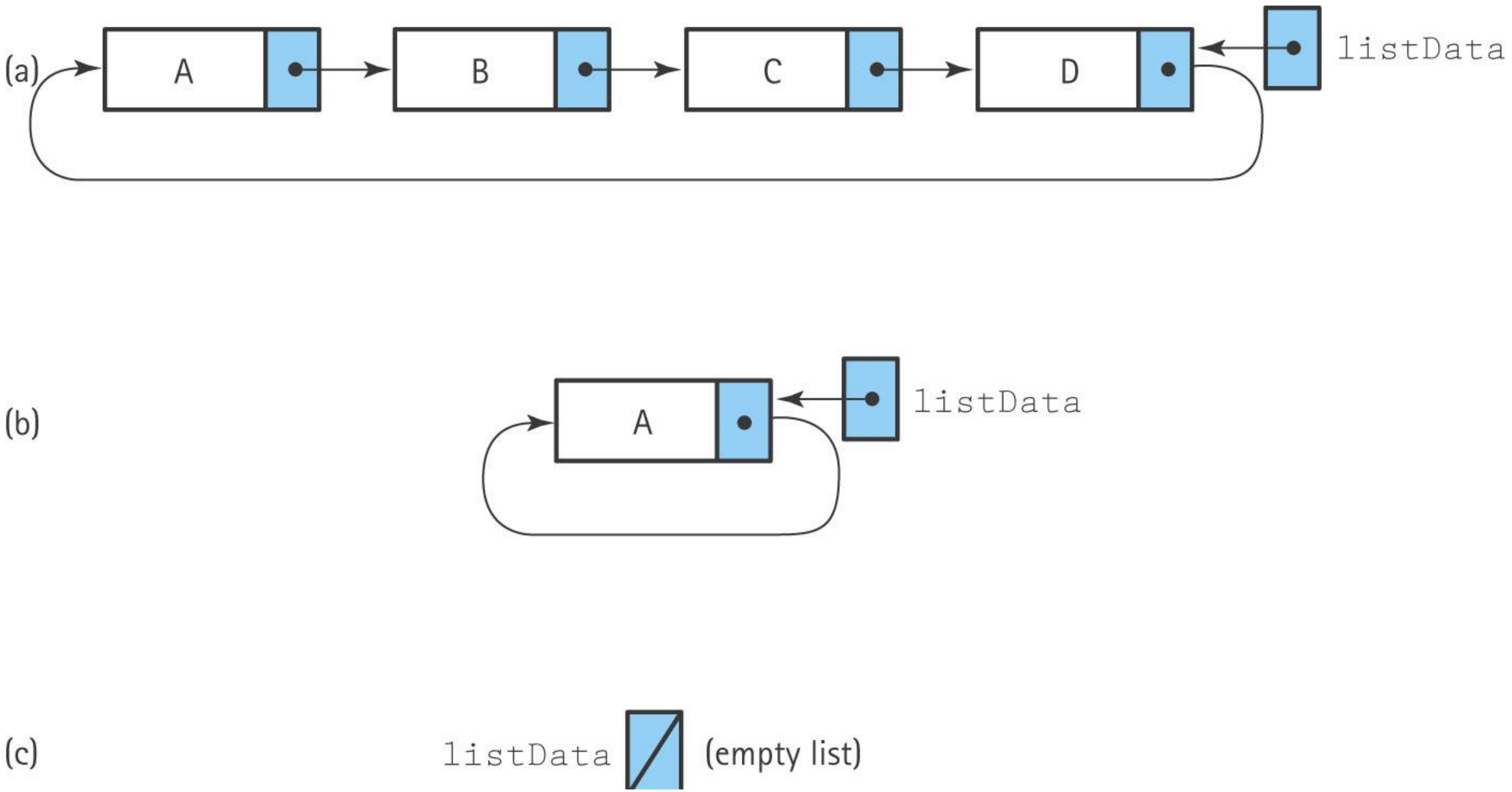
What happens if we insert items in the sorting order?

What is a Circular Linked List?

A circular linked list is a list in which **every node has a successor**; the "last" element is succeeded by the "first" element.

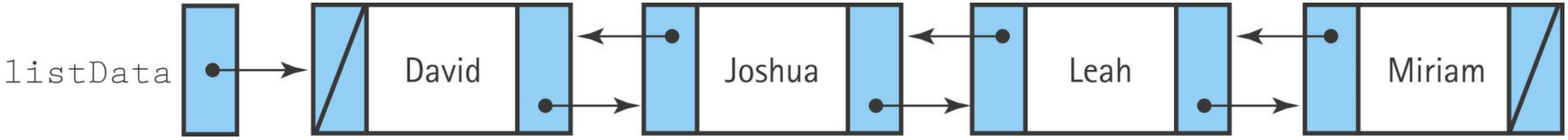


External Pointer to the Last Node



What is a Doubly Linked List?

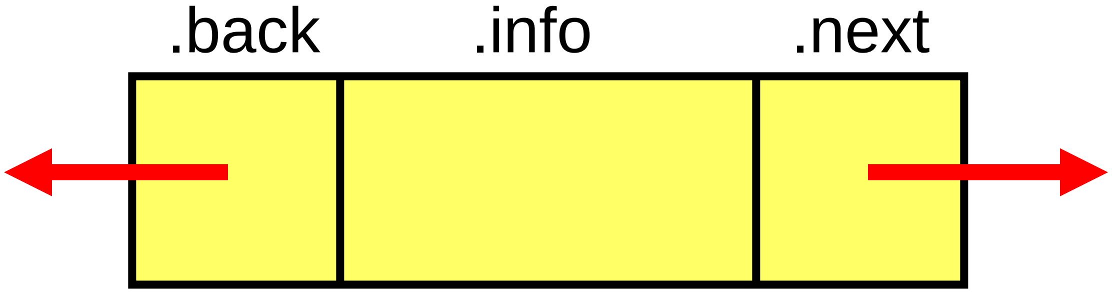
A doubly linked list is a list in which each node is linked to both its successor and its predecessor.



Node data

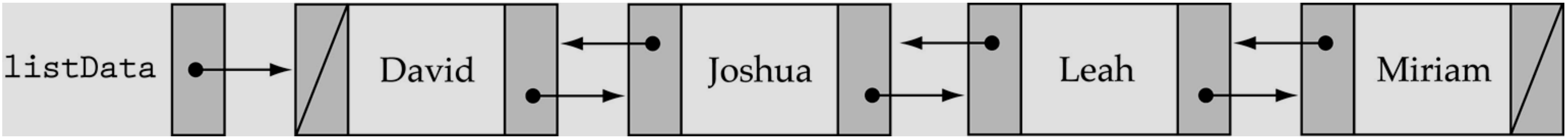
info: the user's data

next, back: the address of the next and previous node in the list



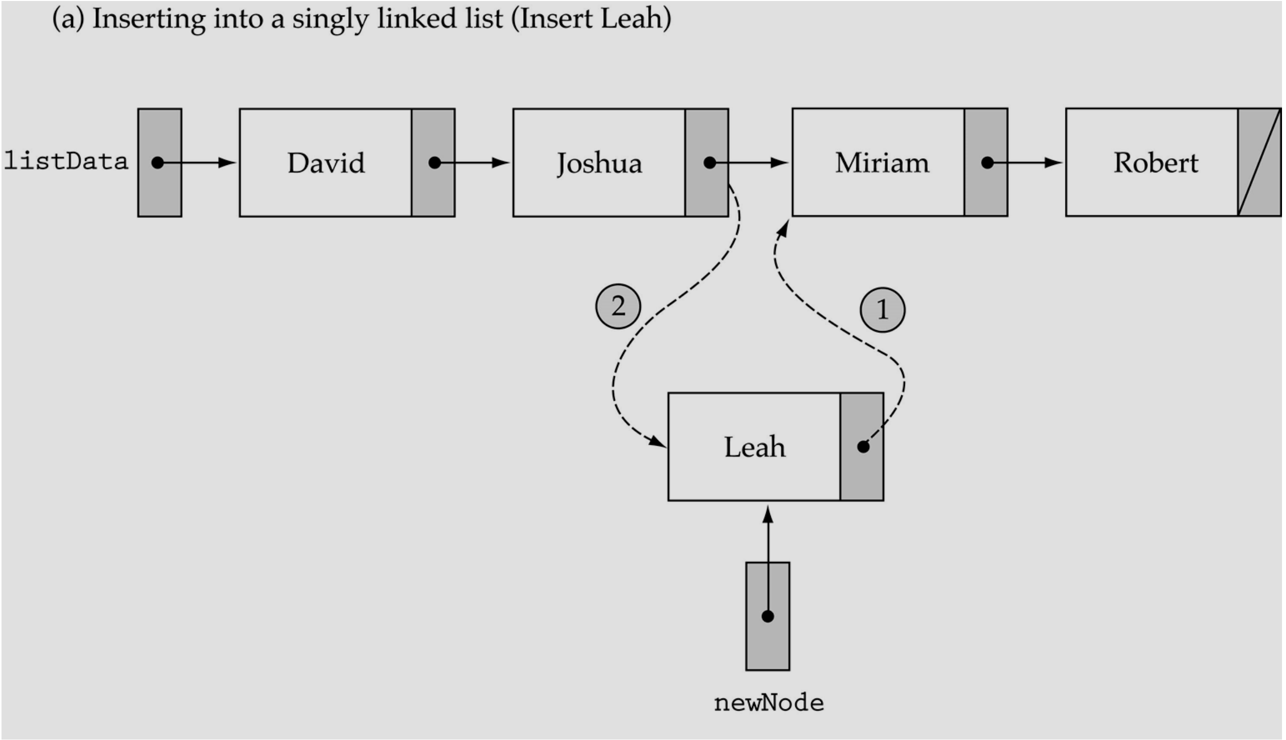
Node data (cont.)

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

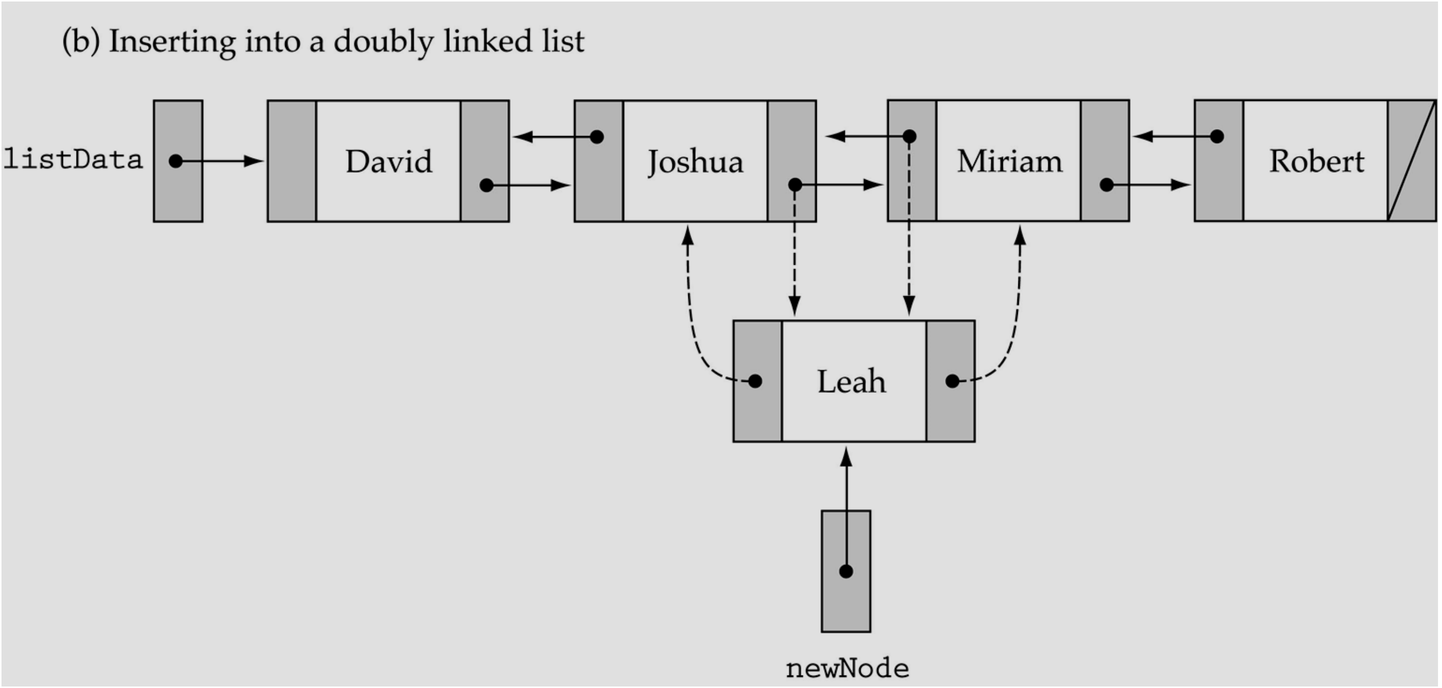


Finding a List Item

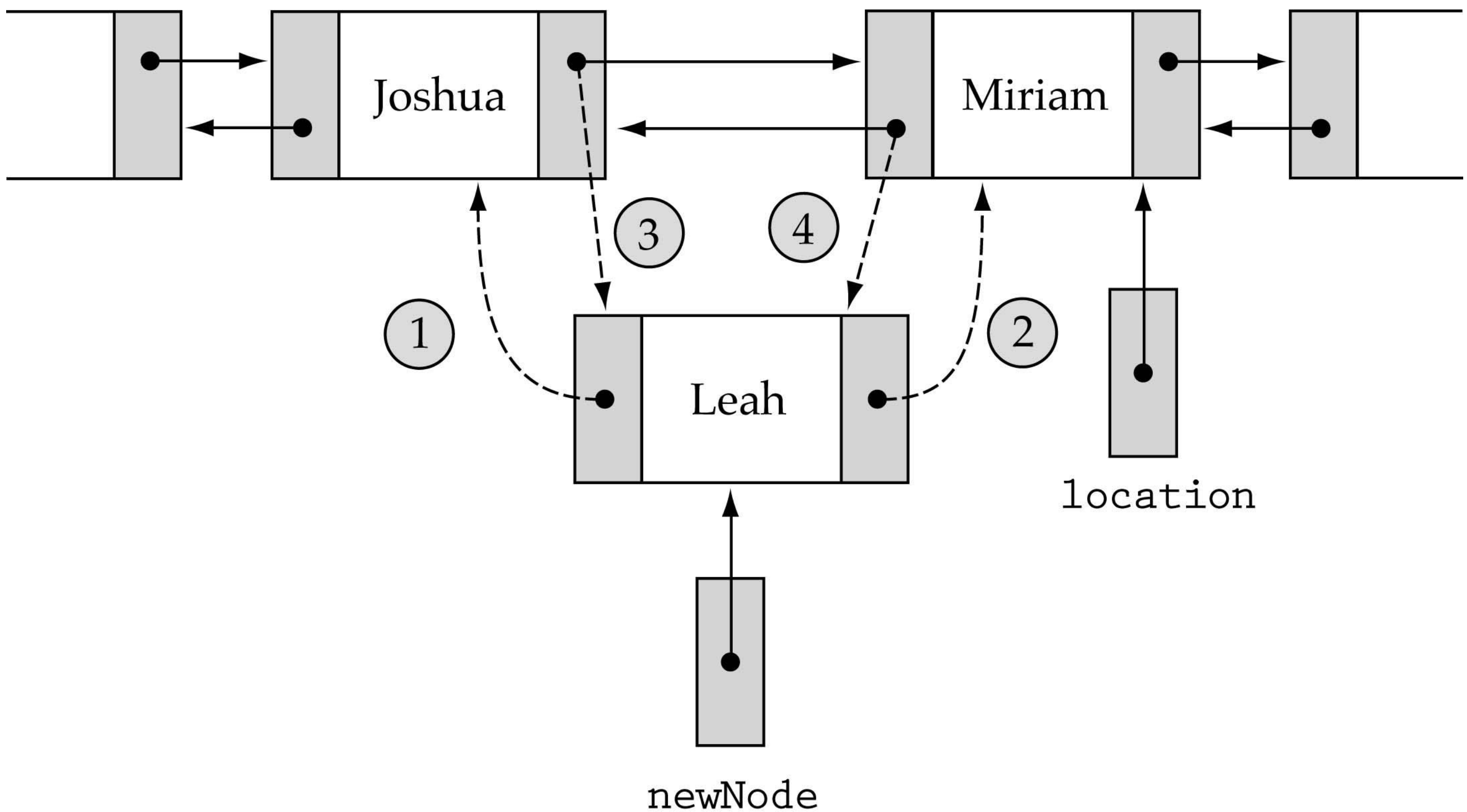
We no longer need to use *prevLocation* (we can get the predecessor of a node using its *back* member)



Finding a List Item (cont.)



Inserting into a Doubly Linked List



1. newNode->back = location->back;
2. newNode->next = location
3. location->back->next=newNode;
4. location->back = newNode;

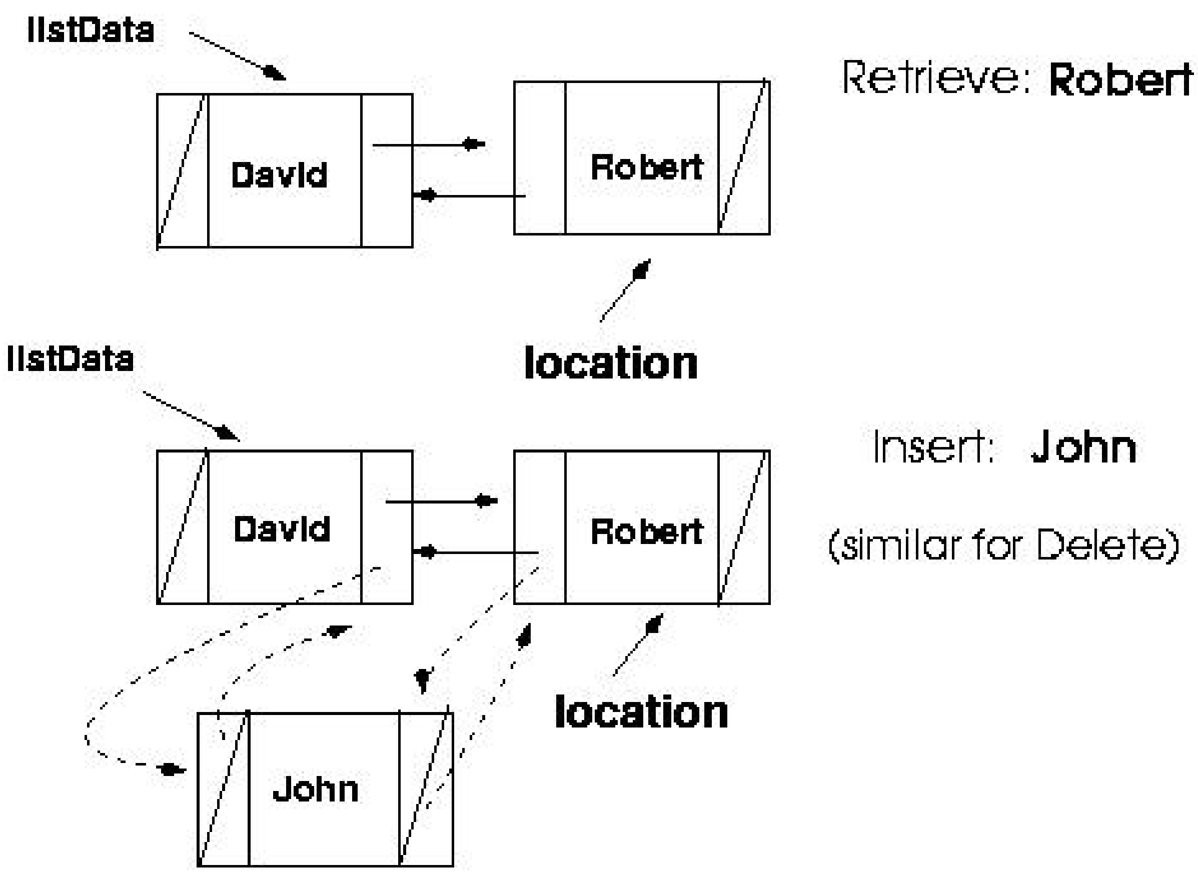
FindItem(listData, item, location, found)

RetrieveItem, **InsertItem**, and **DeleteItem** all require a search !

Write a general non-member function **FindItem** that takes *item* as a parameter and returns *location* and *found.*

InsertItem and DeleteItem need *location* (ignore *found*)

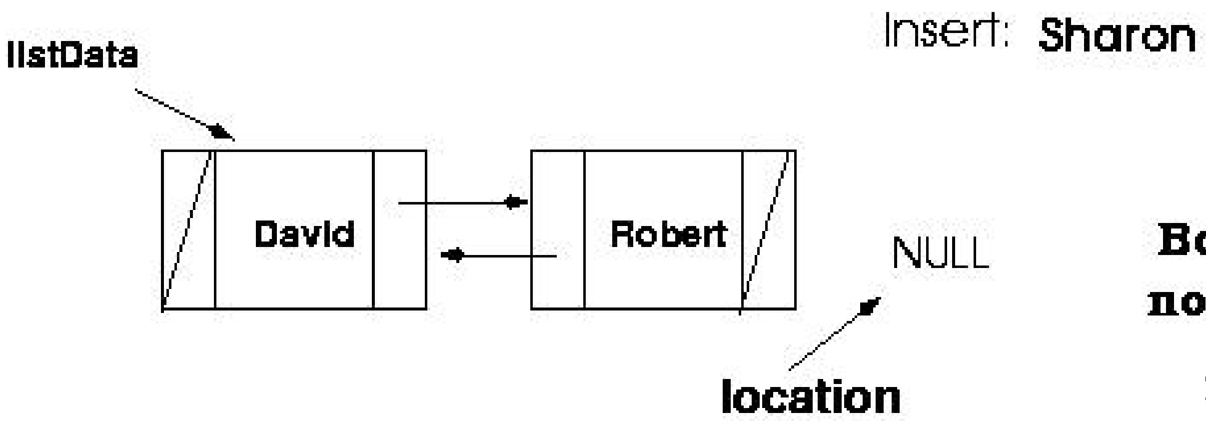
RetrieveItem needs *found (*ignores *location)*


listData
David
Robert
Retrieve: Robert
location
listData
David
Robert
Insert: John
(similar for Delete)
location
John

Finding a List Item (cont.)

```
template<class ItemType>
void FindItem(NodeType<ItemType>* listData, ItemType item,
    NodeType<ItemType>* &location, bool &found)
{
// precondition: list is not empty

 bool moreToSearch = true;

 location = listData;
 found = false;

 while( moreToSearch && !found) {

   if(item < location->info)
     moreToSearch = false;
   else if(item == location->info)
     found = true;
   else {
     location = location->next;
     moreToSearch =
          (location != NULL);
   }
 }
}
```

Problem: Finding a List Item



Book's solution will not work in this case !

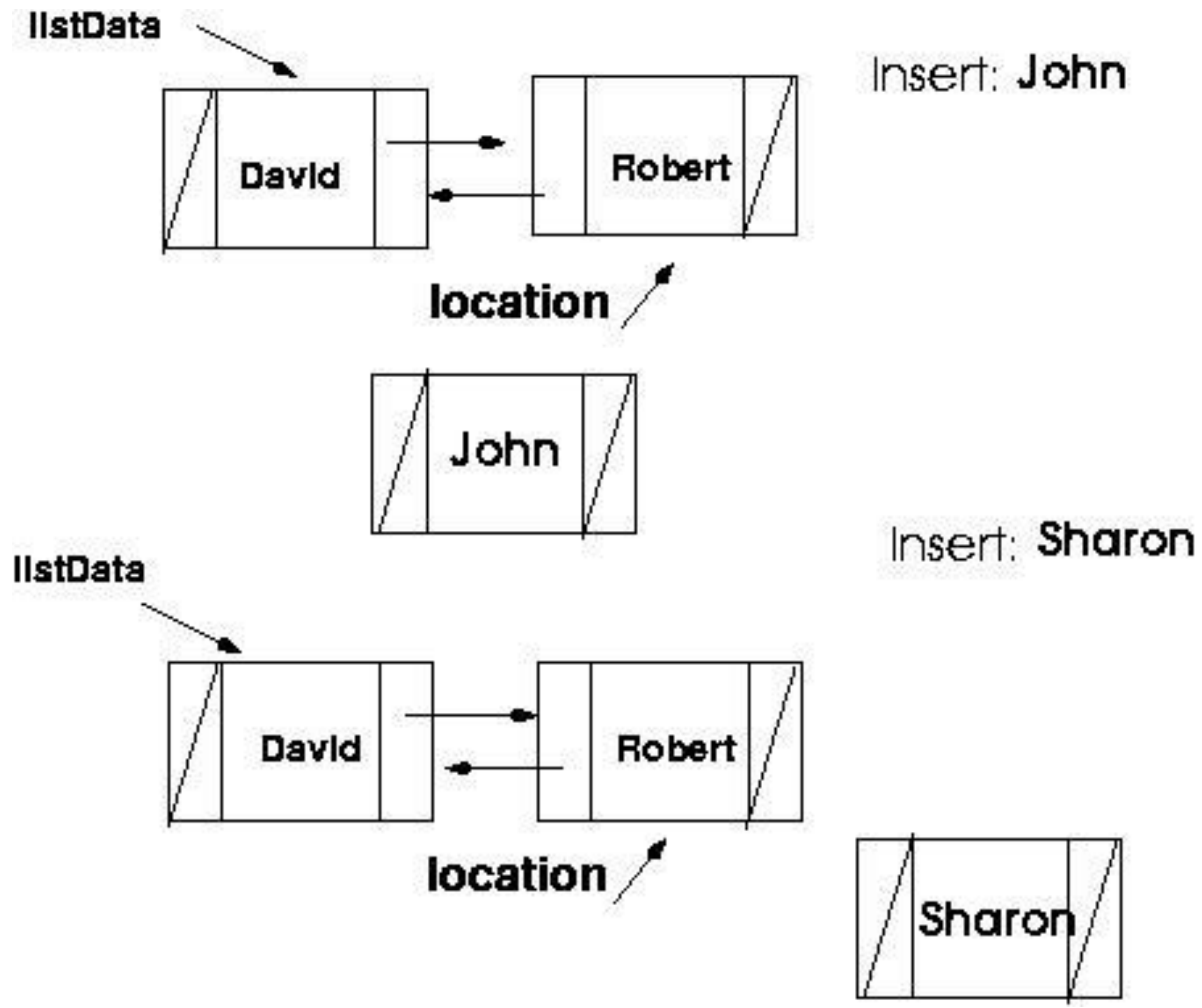
fix the problem:

```
if(location->next != NULL)
  location = location->next;
```

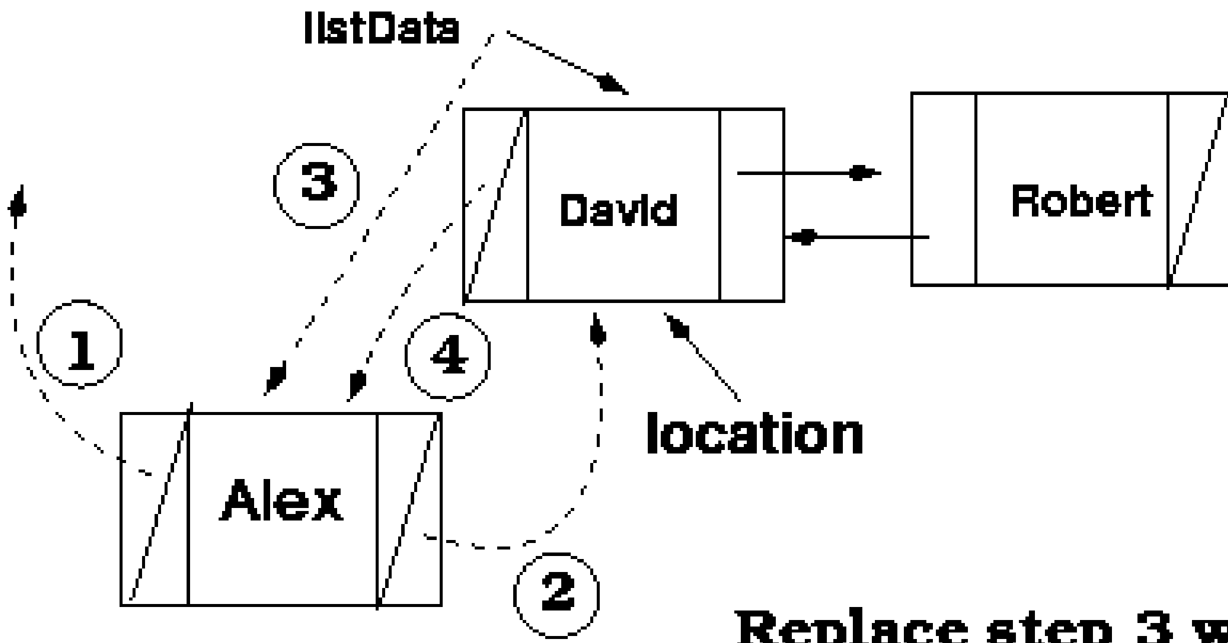
Correction: Finding a List Item

```
template<class ItemType>
void FindItem(NodeType<ItemType>* listData, ItemType item,
    NodeType<ItemType>* &location, bool &found)
{
// precondition: list is not empty

 bool moreToSearch = true;

 location = listData;
 found = false;

 while( moreToSearch && !found) {

   if(item < location->info)
     moreToSearch = false;
   else if(item == location->info)
     found = true;
   else {
     if(location->next == NULL)
       moreToSearch = false;
     else
       location = location->next;
   }
  }
}
```

How can we distinguish between the following two cases?



Special case: inserting in the beginning



Insert: **Alex**

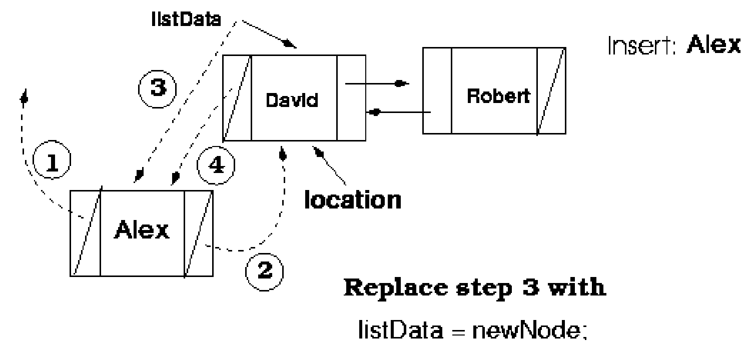
Replace step 3 with

listData = newNode;

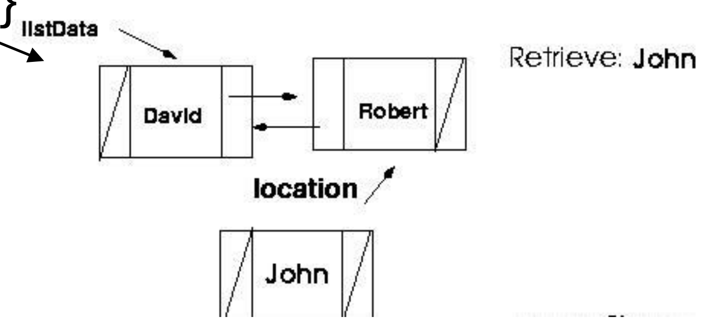
(See Page 11)

Inserting into a Doubly Linked List

```
template<class ItemType>
void SortedType<ItemType>::InsertItem(ItemType item)
{

NodeType<ItemType>* newNode;
 NodeType<ItemType>* location;
 bool found;

 newNode = new NodeType<ItemType>;
 newNode->info = item;
 if (listData != NULL) {

   FindItem(listData, item, location, found);

   if (location->info > item) {
     newNode->back = location->back; (1)
     newNode->next = location; (2)
     if (location != listData) // special case
       (location->back)->next = newNode; (3)
```

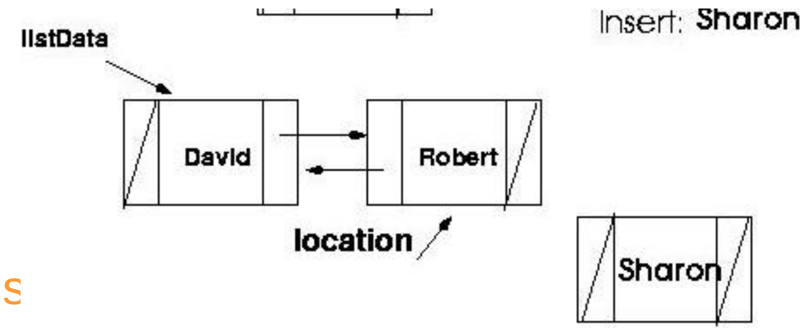



```
else
   listData = newNode; (3)
location->back = newNode; (4)
}
```

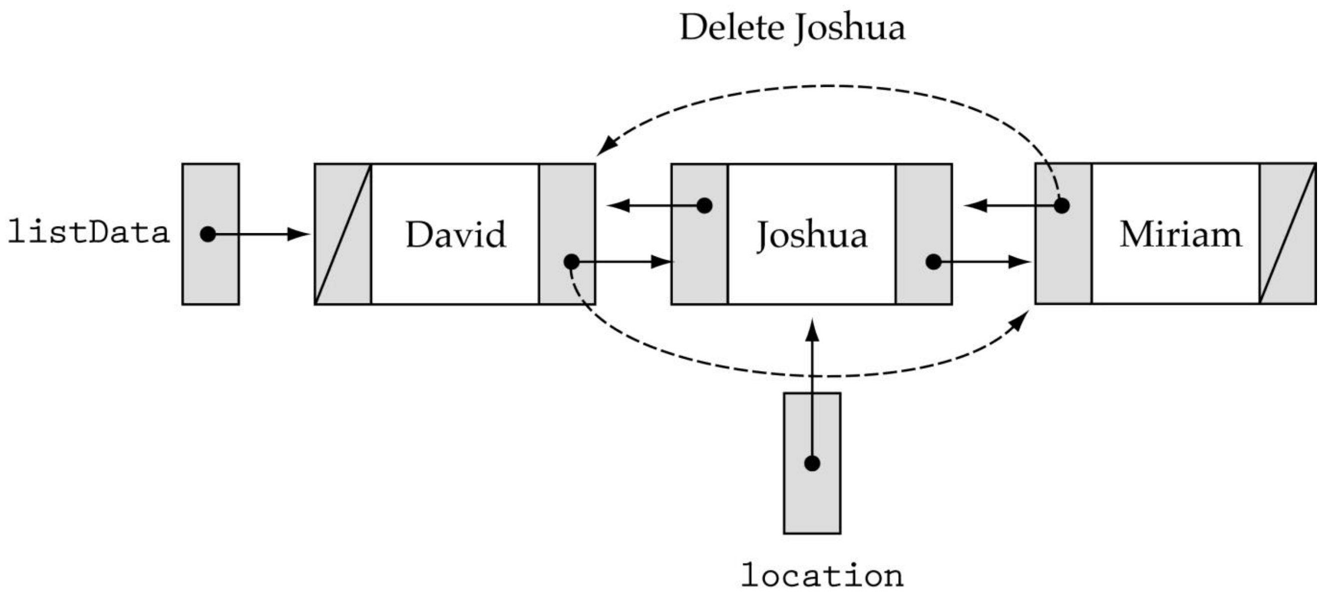


Inserting into a Doubly Linked List (cont.)

```
else {                    // insert at the end
    newNode->back = location;
    location->next = newNode;
    newNode->next = NULL;
  }
}
else {                    // insert into an empty lis
  listData = newNode;
  newNode->next = NULL;
  newNode->back = NULL;
}
length++;
}
```



Deleting from a Doubly Linked List



Be careful about the end cases!!

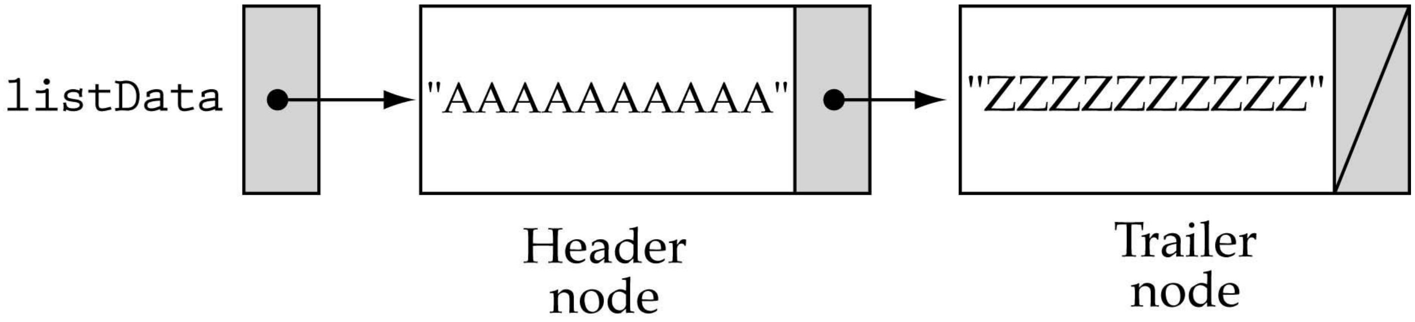
Headers and Trailers

- Special cases arise when we are dealing with the first or last nodes
- How can we simplify the implementation?
  - Idea: make sure that we never insert or delete the ends of the list
  - How? Set up dummy nodes with values outside of the range of possible values

Headers and Trailers (cont.)

Header Node: contains a value smaller than any possible list element

Trailer Node: contains a value larger than any possible list element



A linked list as an array of records

What are the advantages of using linked lists?

(1) Dynamic memory allocation

(2) Efficient insertion-deletion (for sorted lists)

Can we implement a linked list without dynamic memory allocation ?

A linked list as an array of records (cont.)

nodes	.info	.next
[0]	David	4
[1]		
[2]	Miriam	6
[3]		
[4]	Joshua	7
[5]		
[6]	Robert	-1
[7]	Leah	2
[8]		
[9]		

list: 0

A Sorted list Stored in an Array of Nodes

nodes	.info	.next
[0]	David	4
[1]		
[2]	Miriam	6
[3]		
[4]	Joshua	7
[5]		
[6]	Robert	−1
[7]	Leah	2
[8]		
[9]		

list 0

An Array with Linked List of Values and Free Space

nodes	.info	.next
[0]	David	4
[1]		5
[2]	Miriam	6
[3]		8
[4]	Joshua	7
[5]		3
[6]	Robert	NUL
[7]	Leah	2
[8]		9
[9]		NUL

list	0
free	1

An Array with Three Lists (Including the Free List)

free: 7

nodes	.info	.next
[0]	John	4
[1]	Mark	5
[2]		3
[3]		NUL
[4]	Nell	8
[5]	Naomi	6
[6]	Robert	NUL
[7]		2
[8]	Susan	9
[9]	Susanne	NUL

list1: 0

list2: 1

Recall Definition of Stack

Logical (or ADT) level: A stack is an ordered group of homogeneous items (elements), in which the removal and addition of stack items can take place only at the top of the stack.

A stack is a LIFO "last in, first out" structure.

Stack ADT Operations

IsEmpty -- Determines whether the stack is currently empty.

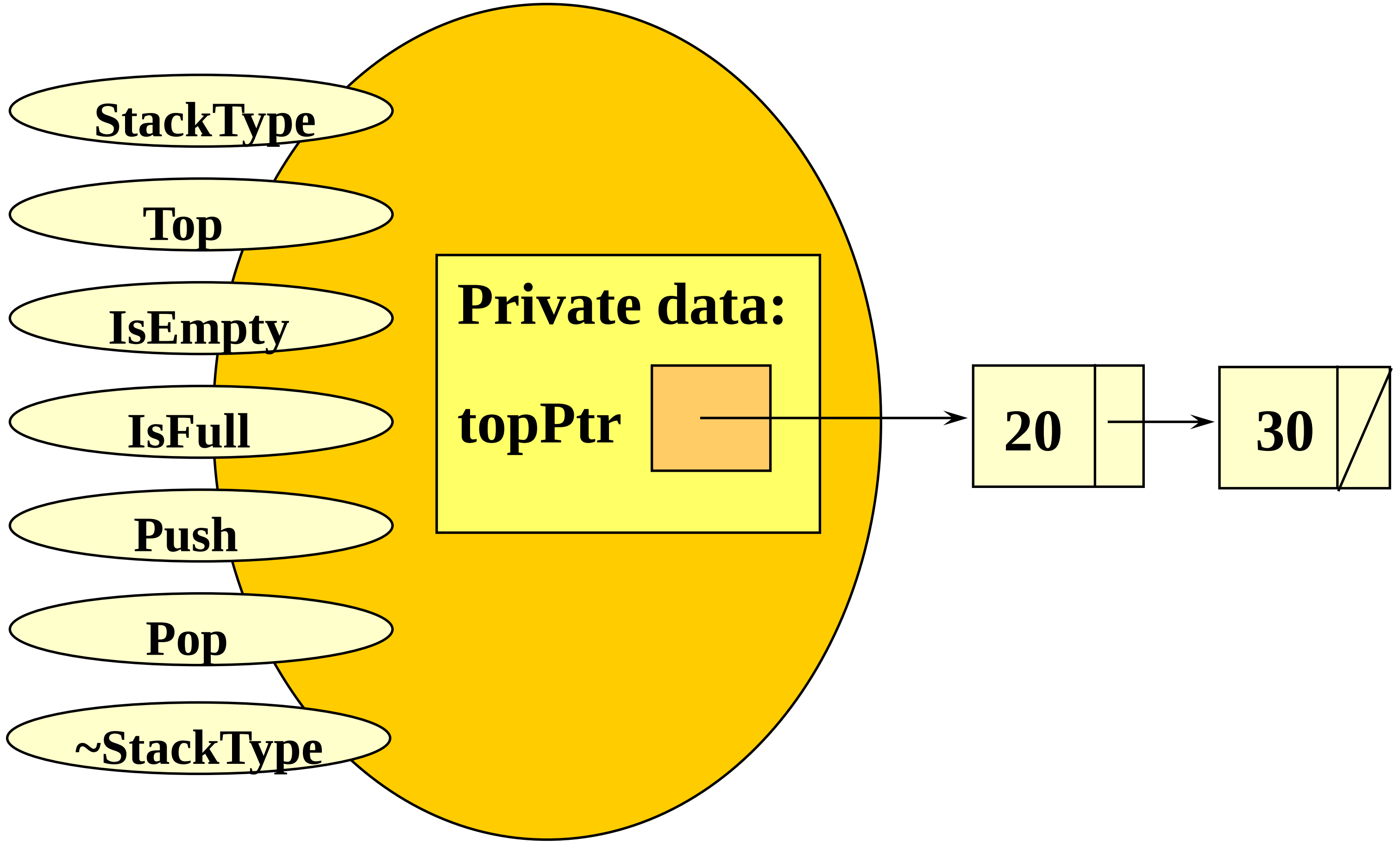
IsFull -- Determines whether the stack is currently full.

Push (ItemType newItem) -- Adds newItem to the top of the stack.

Pop -- Removes the item at the top of the stack and returns it in item.

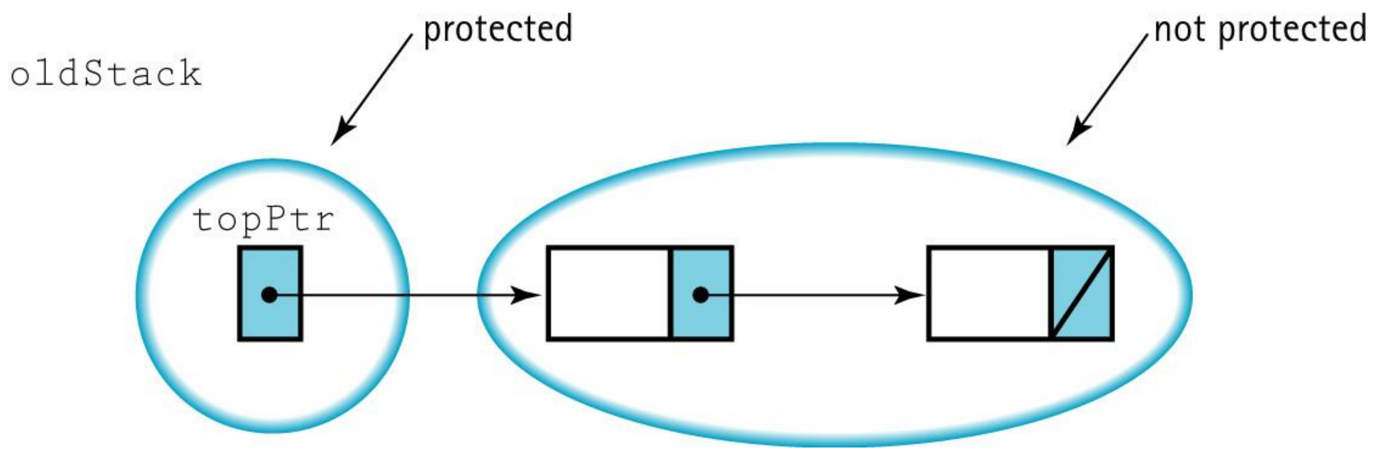
Top -- Returns a copy of the top item

class StackType<int>



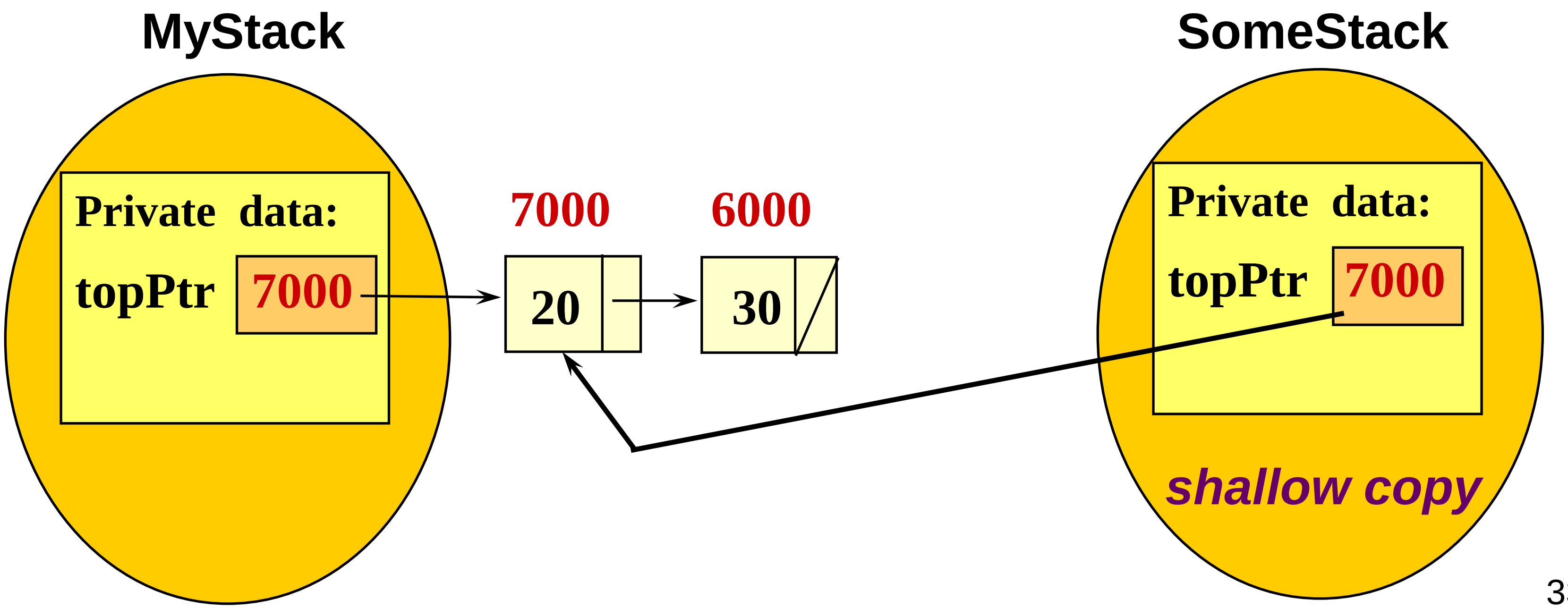
What happens . . .

When a function is called that uses pass by value for a class object like our dynamically linked stack?



Pass by value makes a shallow copy

```
StackType<int>  MyStack;          //  CLIENT CODE
     .
     .
     .
MyFunction( MyStack );            //  function call
```



Shallow Copy vs. Deep Copy

A shallow copy copies only the class data members, and does not copy any pointed-to data.

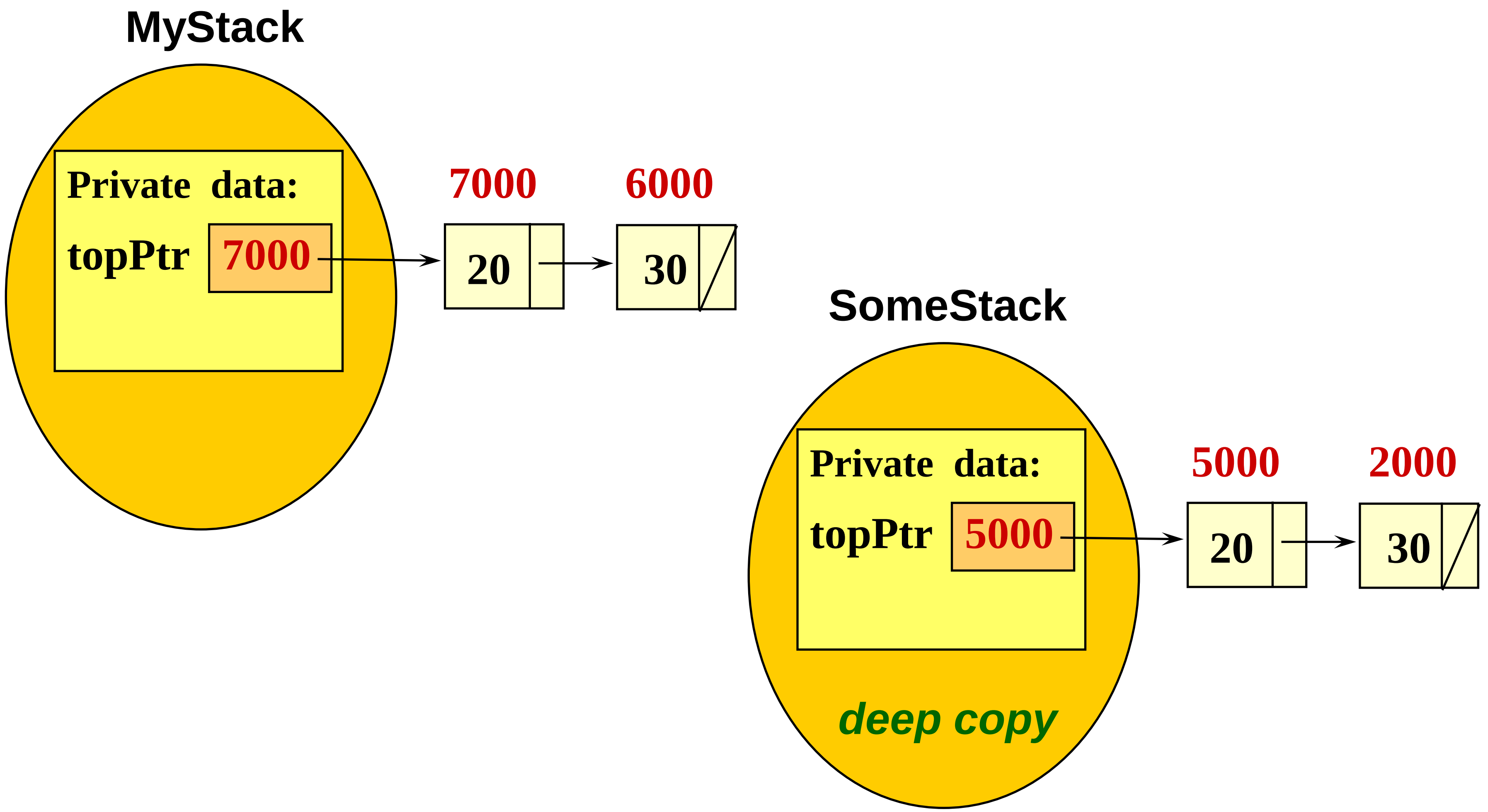
A deep copy copies not only the class data members, but also makes separately stored copies of any pointed-to data.

What's the difference?

A shallow copy shares the pointed to data with the original class object.

A deep copy stores its own copy of the pointed to data at different locations than the data in the original class object.

Making a deep copy



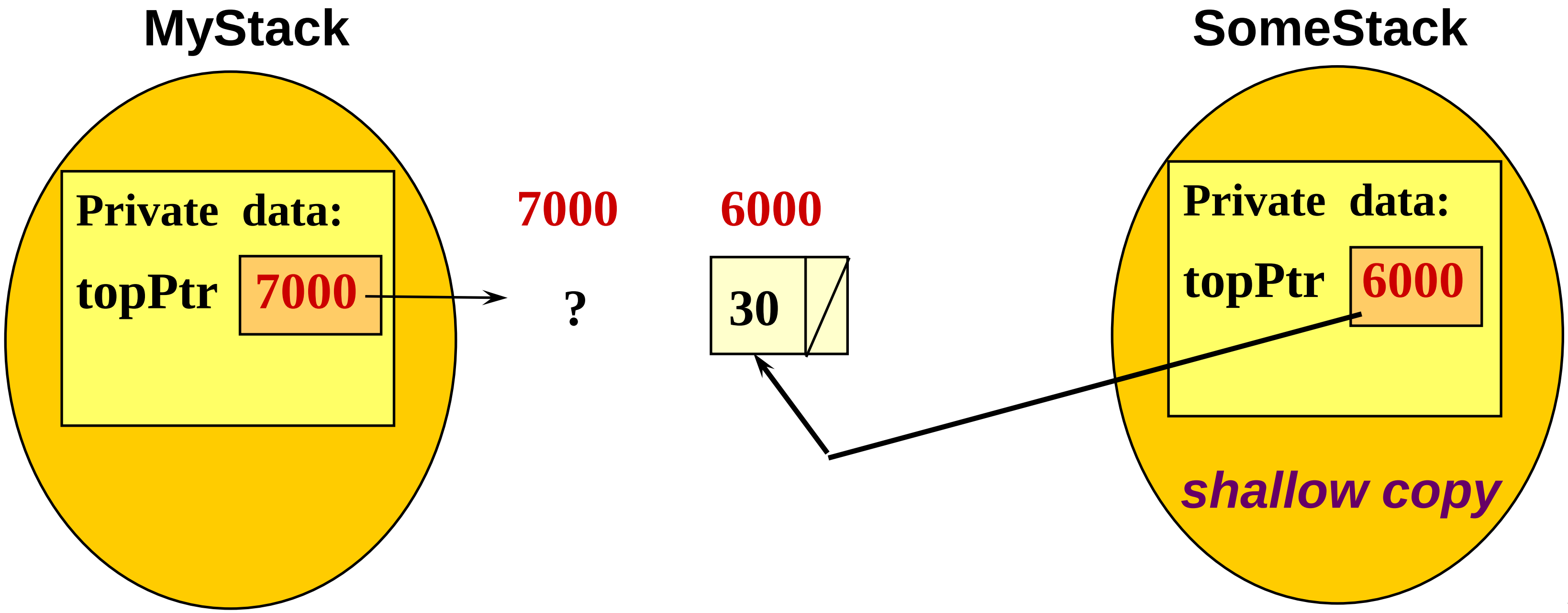
Suppose `MyFunction` Uses Pop

```
                                          // FUNCTION CODE
template<class ItemType>
void  MyFunction( StackType<ItemType> SomeStack )
   // Uses pass by value
{
      ItemType  item;
      SomeStack.Pop(item);
      .
      .
      .
}
```

WHAT HAPPENS IN THE SHALLOW COPY SCENARIO?

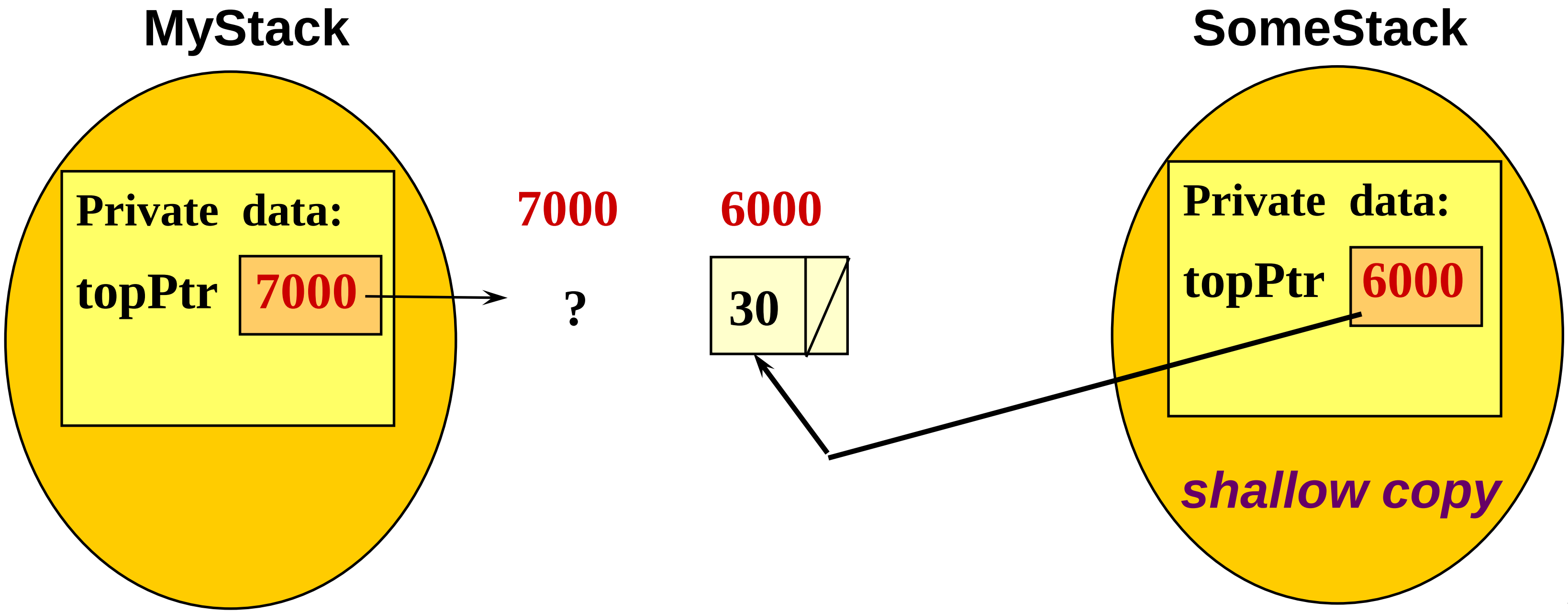
`MyStack.topPtr` is left dangling

```
StackType<int>  MyStack;        // CLIENT CODE
     .
     .
     .
MyFunction( MyStack );
```



`MyStack.topPtr` is left dangling

NOTICE THAT NOT JUST FOR THE SHALLOW COPY, BUT ALSO FOR ACTUAL PARAMETER MyStack, THE DYNAMIC DATA HAS CHANGED!



As a result . . .

This default method used for pass by value is not the best way when a data member pointer points to dynamic data.

Instead, you should write what is called a copy constructor, which makes a deep copy of the dynamic data in a different memory location.

Use call by reference in the case where you do not need a copy of the passing object.

More about copy constructors

When there is a copy constructor provided for a class, the copy constructor is used to make copies for pass by value.

You do not call the copy constructor.

Like other constructors, it has no return type.

Because the copy constructor properly defines pass by value for your class, it must use pass by reference in its definition.

Copy Constructor

Copy constructor is a special member function of a class that is implicitly called in these three situations:

- **passing object parameters by value,**
- **initializing an object variable in a declaration,**
- **returning an object as the return value of a function.**

```
// DYNAMICALLY LINKED IMPLEMENTATION OF STACK
template<class ItemType>
class StackType  {
public:
   StackType( );
       // Default constructor.
       // POST:  Stack is created and empty.
   StackType( const StackType<ItemType>& anotherStack );
       // Copy constructor.
       // Implicitly called for pass by value.
   .
   .
   .
   ~StackType( );
       // Destructor.
       // POST: Memory for nodes has been deallocated.
private:
   NodeType<ItemType>*  topPtr ;
};
```

Classes with Data Member Pointers Need

CLASS CONSTRUCTOR

CLASS COPY CONSTRUCTOR

CLASS DESTRUCTOR

```
template<class ItemType>          // COPY CONSTRUCTOR
StackType<ItemType>::
StackType( const StackType<ItemType>& anotherStack )
{  NodeType<ItemType>* ptr1 ;
   NodeType<ItemType>* ptr2 ;
   if ( anotherStack.topPtr == NULL )
        topPtr = NULL ;
   else                        // allocate memory for first node
   {    topPtr = new NodeType<ItemType> ;
        topPtr->info = anotherStack.topPtr->info ;
        ptr1 = anotherStack.topPtr->next ;
        ptr2 = topPtr ;
        while ( ptr1 != NULL )      // deep copy other nodes
        {      ptr2->next = new NodeType<ItemType> ;
               ptr2 = ptr2->next ;
               ptr2->info = ptr1->info ;
               ptr1 = ptr1->next ;
        }
        ptr2->next = NULL ;
   }
}
```

What About the Assignment Operator?

The default method used for assignment of class objects makes a shallow copy.

If your class has a data member pointer to dynamic data, you should write a member function to overload the assignment operator to make a deep copy of the dynamic data.

```
// DYNAMICALLY LINKED IMPLEMENTATION OF STACK
template<class ItemType>
class StackType  {
public:
   StackType( );
       // Default constructor.
   StackType( const StackType<ItemType>& anotherStack );
       // Copy constructor.
   void  operator= ( StackType<ItemType> );
       // Overloads assignment operator.
   .
   .
   .

   ~StackType( );
       // Destructor.
private:
   NodeType<ItemType>*  topPtr ;
};
```

C++ Operator Overloading Guides

1. All operators except these :: . sizeof ?: may be overloaded.
2. At least one operand must be a class instance.
3. You cannot change precedence, operator symbols, or number of operands.
4. Overloading ++ and -- requires prefix form use by default, unless special mechanism is used.
5. To overload these operators = ( ) [ ] member functions (not friend functions) must be used.
6. An operator can be given multiple meanings if the data types of operands differ.

Using Overloaded Binary operator+

When a Member Function was defined

myStack + yourStack

myStack.operator+(yourStack)

When a Friend Function was defined

myStack + yourStack

operator+(myStack, yourStack)

Case Study: Implementing a large integer ADT

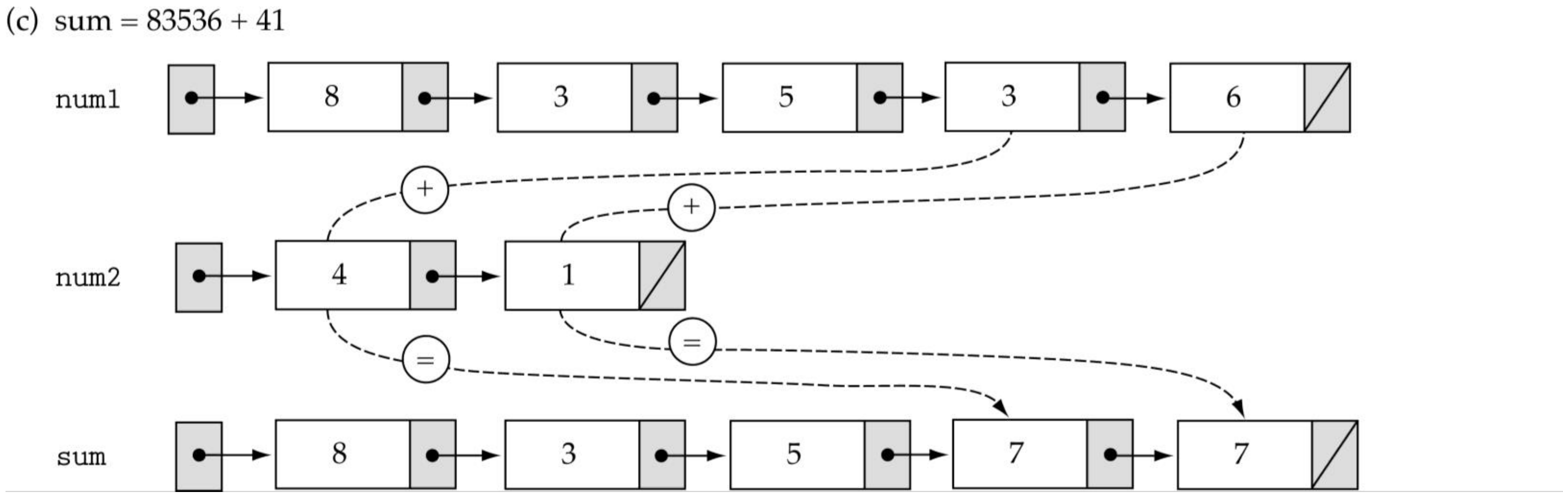
The range of integer values varies from one computer to another

For *long* integers, the range is

[-2,147,483,648 to 2,147,483,647]

How can we manipulate larger integers?

Case Study: Implementing a large integer ADT (cont.)



- A special list ADT