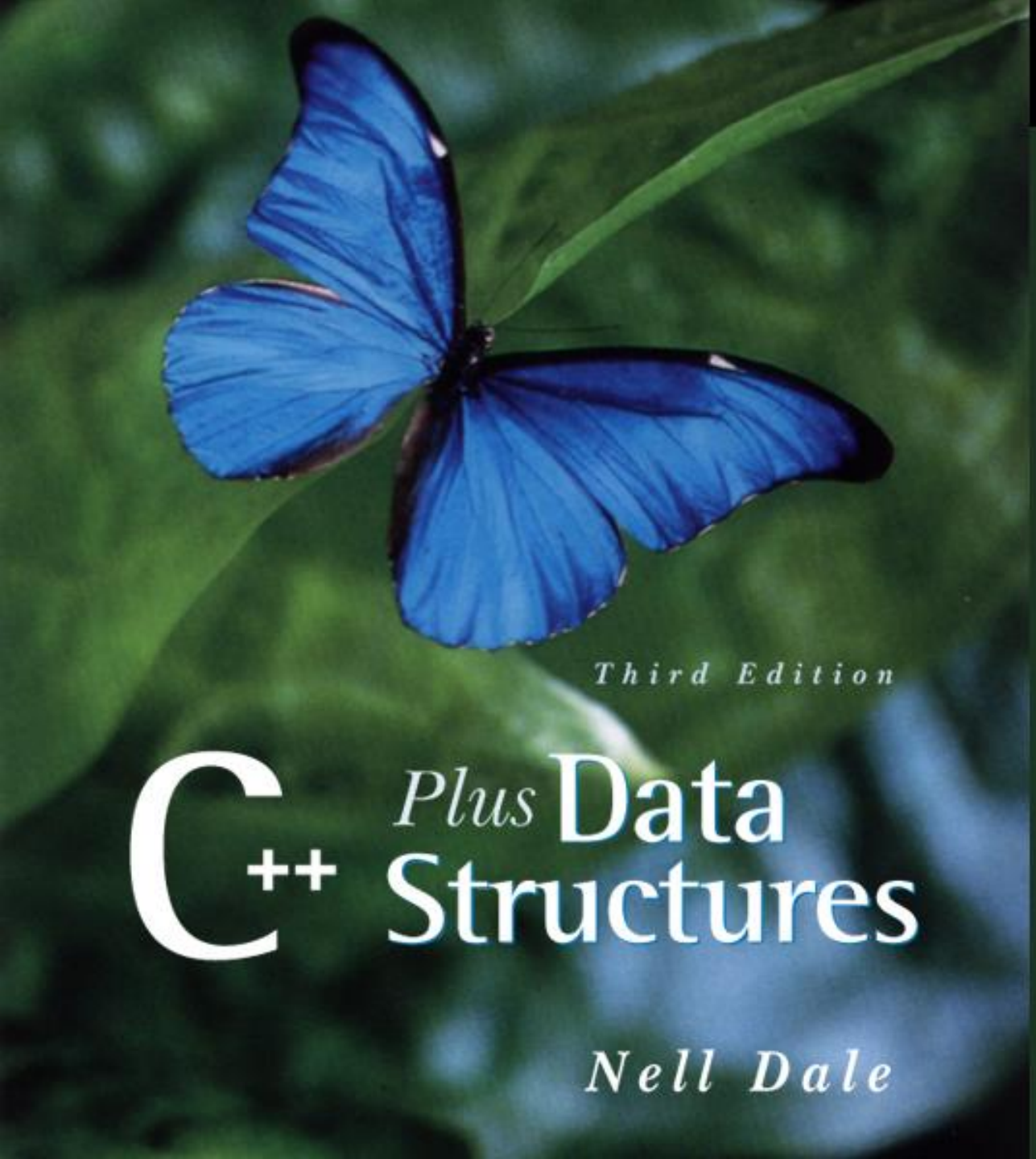


Chapter

5

Linked Structures

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.

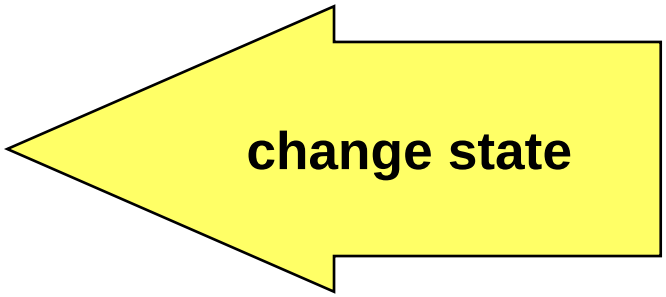
Stack ADT Operations

- **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.**
- **Top – Returns a copy of top items**

ADT Stack Operations

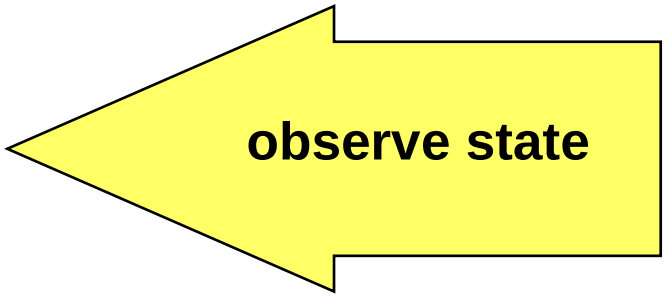
Transformers

- Push
- Pop



Observers

- IsEmpty
- IsFull
- Top

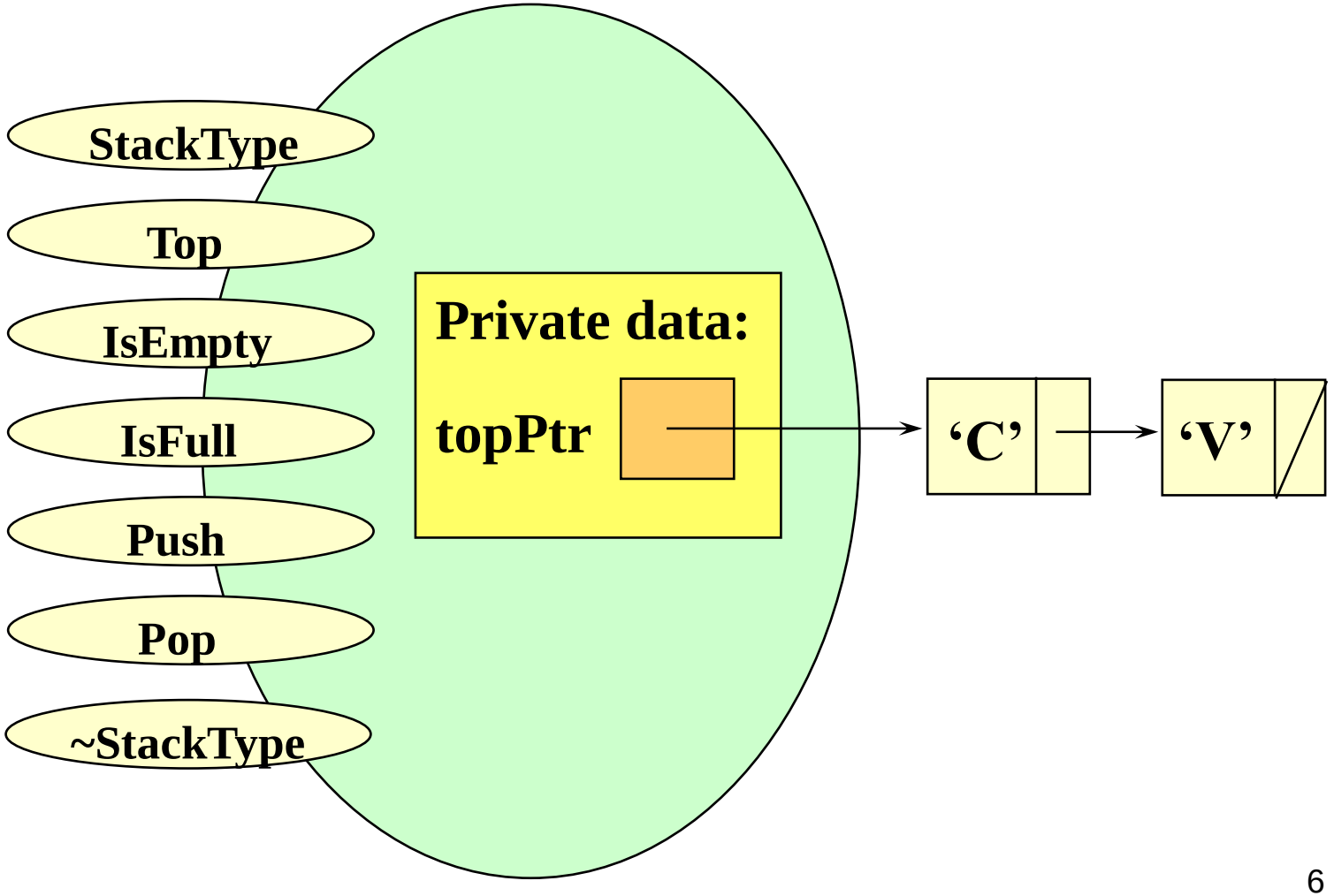


Another Stack Implementation

- **One advantage of an ADT is that the kind of implementation used can be changed.**
- **The dynamic array implementation of the stack has a weakness -- the maximum size of the stack is passed to the constructor as parameter.**
- **Instead we can dynamically allocate the space for each stack element as it is pushed onto the stack.**

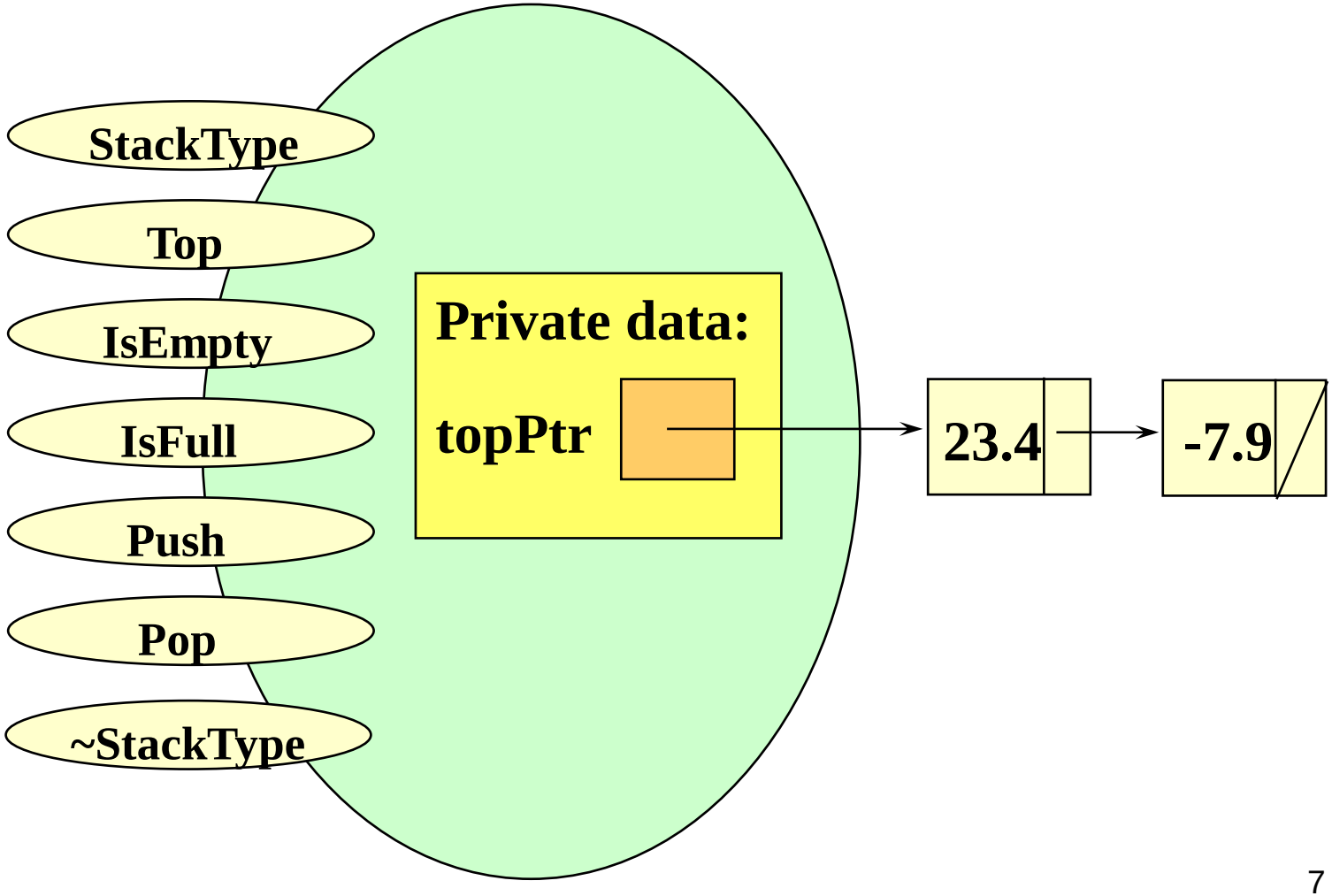
ItemType is char

class StackType



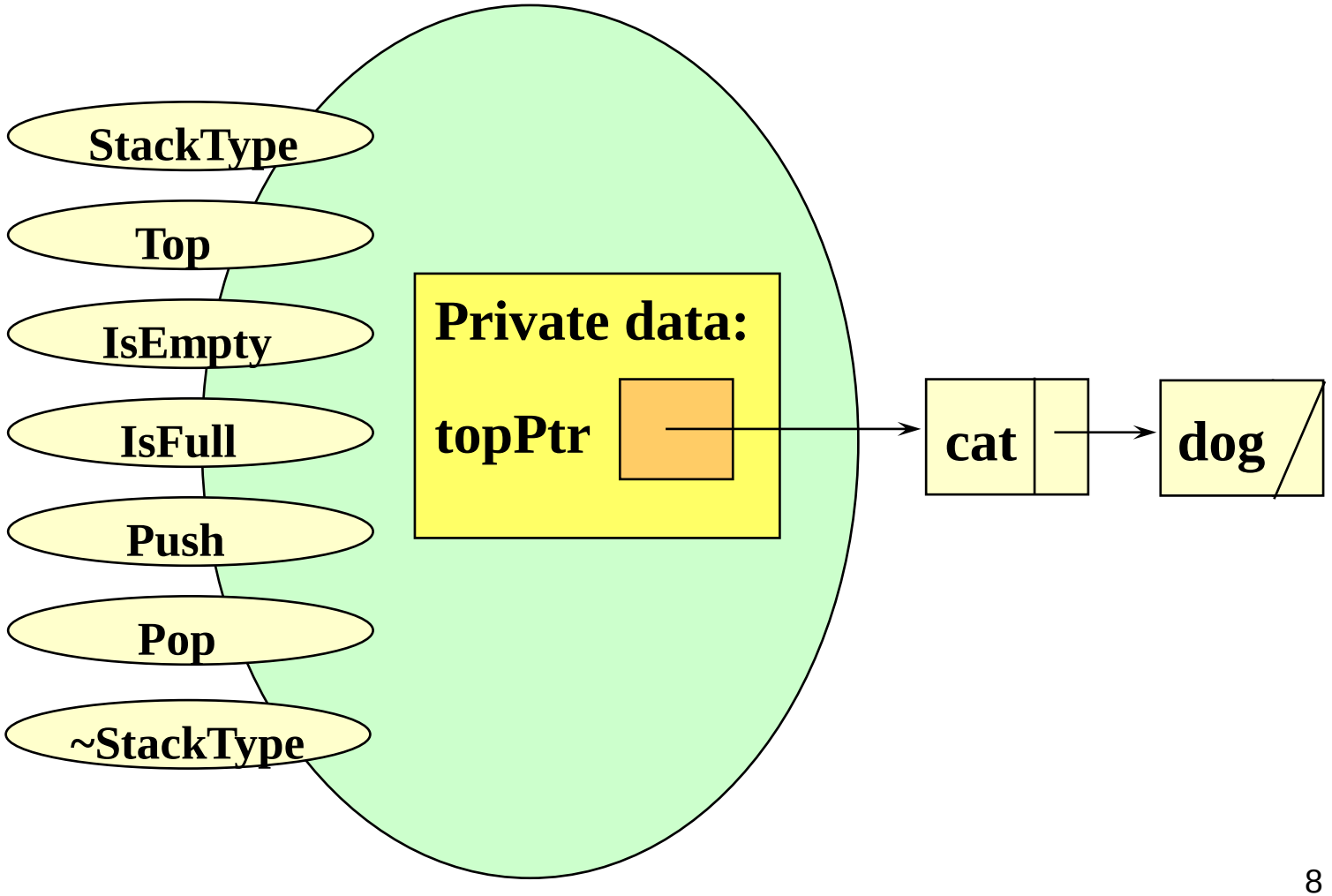
ItemType is float

class StackType



ItemType is StrType

class StackType



Tracing Client Code

letter ‘V’

```
char     letter = ‘V’;

StackType  myStack;

myStack.Push(letter);

myStack.Push(‘C’);

myStack.Push(‘S’);

If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}

myStack.Push(‘K’);
```

Tracing Client Code

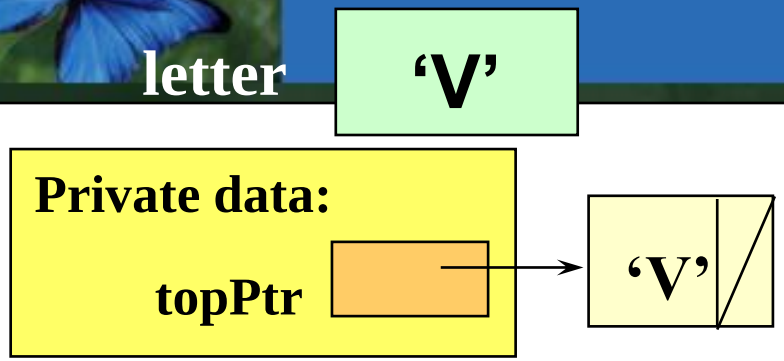
letter ‘V’

Private data:

topPtr NULL

```
char     letter = ‘V’;
StackType  myStack;
myStack.Push(letter);
myStack.Push(‘C’);
myStack.Push(‘S’);
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push(‘K’);
```

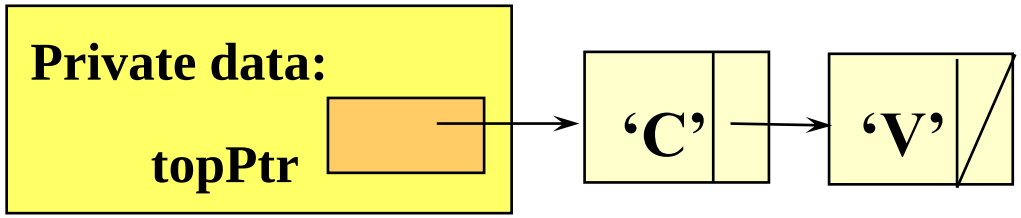
Tracing Client Code




```
char     letter = 'V';
StackType  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push('K');
```

Tracing Client Code

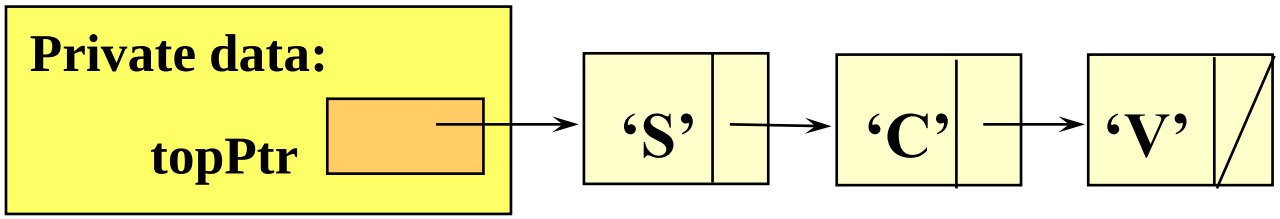
letter ‘V’




```
char     letter = 'V';
StackType  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push('K');
```

Tracing Client Code

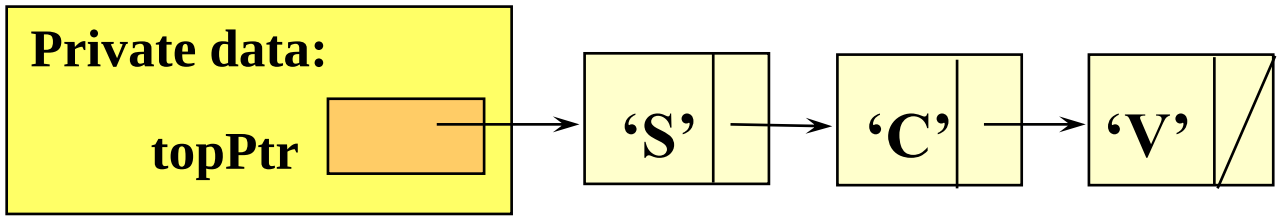
letter ‘V’




```
char     letter = 'V';
StackType  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push('K');
```

Tracing Client Code

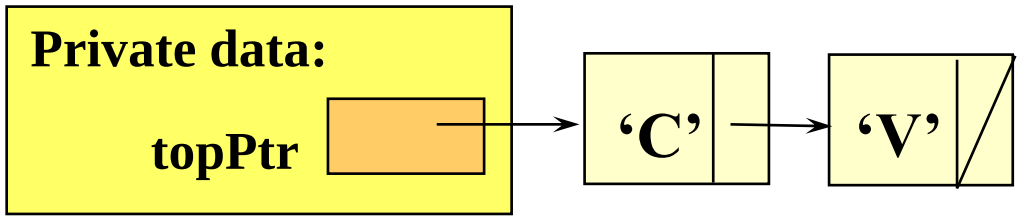
letter ‘V’




```
char     letter = 'V';
StackType  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push('K');
```

Tracing Client Code

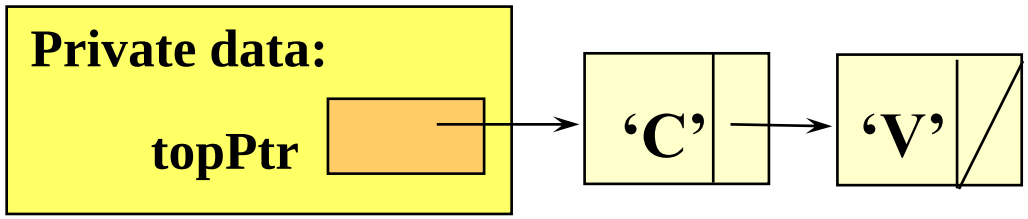
letter ‘S’




```
char     letter = 'V';
StackType  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push('K');
```

Tracing Client Code

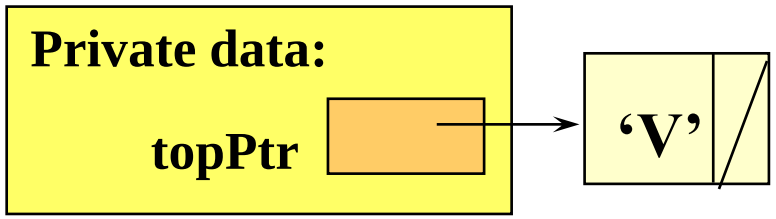
letter ‘S’




```
char     letter = 'V';
StackType  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push('K');
```

Tracing Client Code

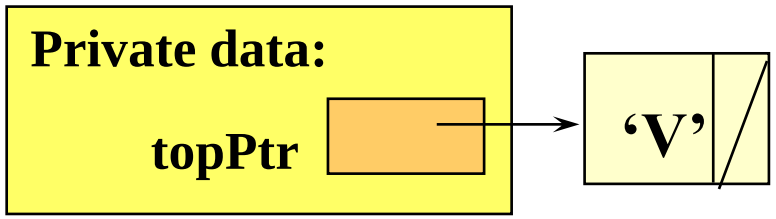
letter ‘C’




```
char     letter = ‘V’;
StackType  myStack;
myStack.Push(letter);
myStack.Push(‘C’);
myStack.Push(‘S’);
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push(‘K’);
```

Tracing Client Code

letter ‘C’




```
char     letter = ‘V’;
StackType  myStack;
myStack.Push(letter);
myStack.Push(‘C’);
myStack.Push(‘S’);
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push(‘K’);
```

Tracing Client Code

letter ‘V’

Private data:

topPtr 

```
char     letter = ‘V’;
StackType  myStack;
myStack.Push(letter);
myStack.Push(‘C’);
myStack.Push(‘S’);
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push(‘K’);
```

Tracing Client Code

letter ‘V’

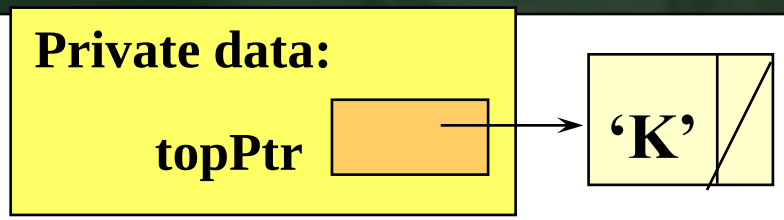
Private data:

topPtr 

```
char     letter = ‘V’;
StackType  myStack;
myStack.Push(letter);
myStack.Push(‘C’);
myStack.Push(‘S’);
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
myStack.Push(‘K’);
```

Tracing Client Code

letter ‘V’




```
char     letter = 'V';
StackType  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
If (!myStack.IsEmpty() )
{
   letter = myStack.Top( );
   myStack.Pop();
}
```

```
myStack.Push('K');
```

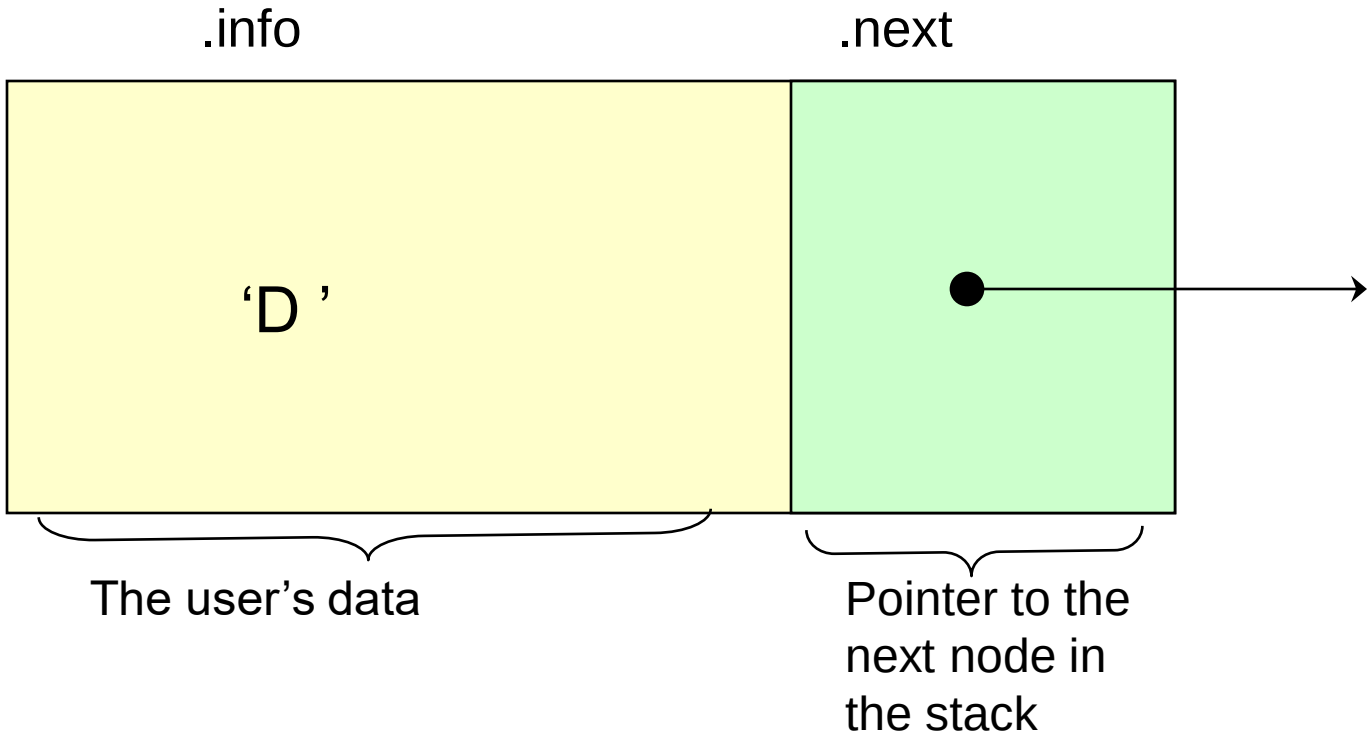
```
// DYNAMICALLY LINKED IMPLEMENTATION OF STACK
Struct NodeType;                    //Forward declaration

class StackType
{
public:
//Identical to previous implementation
private:
   NodeType* topPtr;
};
.
.
.
Struct NodeType
{
   ItemType  info;
   NodeType* next;
};
```

Using operator `new`

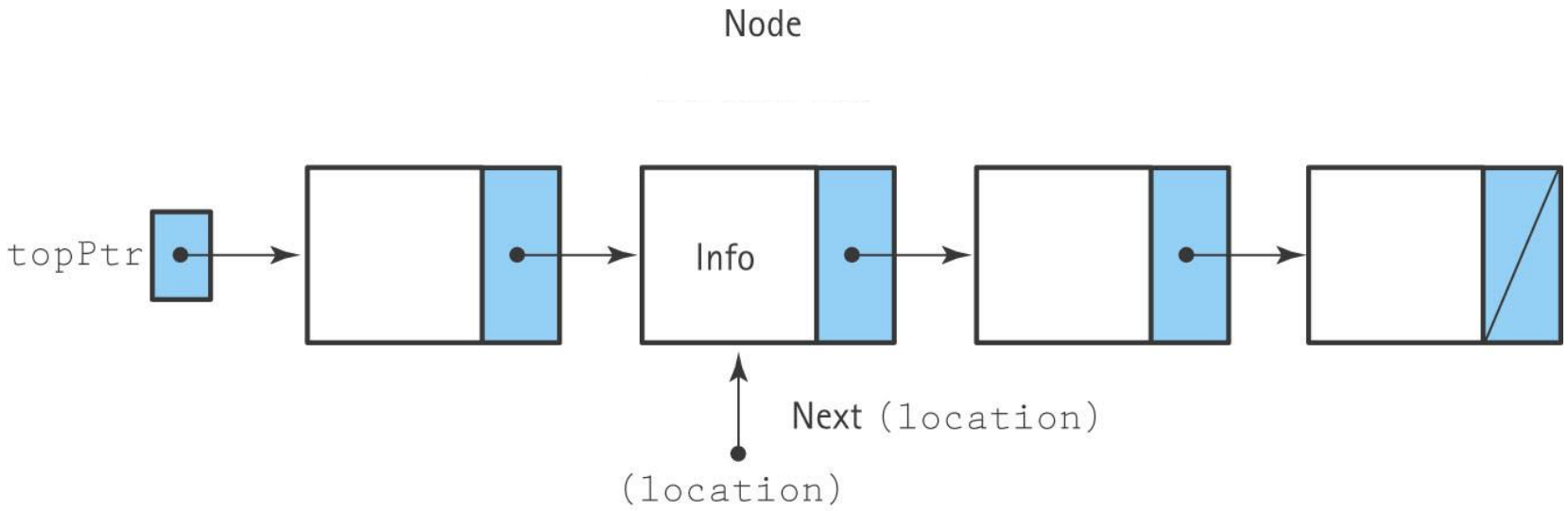
If memory is available in an area called the free store (or heap), operator new allocates the requested object, and returns a pointer to the memory allocated.

The dynamically allocated object exists until the delete operator destroys it.

A Single Node



Node terminology



Node(location) refers to all the data at location, including implementation-specific data

Info(location) refers to the user's data at location

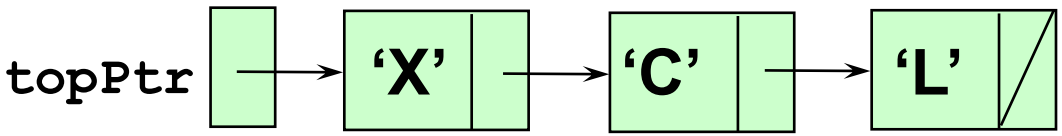
Info(last) refers to the user's data at the last location in the list

Next(location) gives the location of the node following Node(location)

Adding newItem to the stack

newItem ‘B’

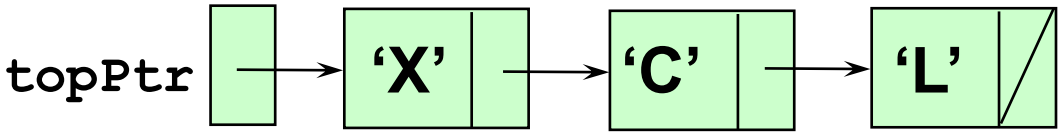
```
newItem = ‘B’;
NodeType*  location;
location = new  NodeType<char>;
location->info = newItem;
location->next = topPtr;
topPtr = location;
```



Adding newItem to the stack

newItem ‘B’

```
newItem = 'B';
NodeType*  location;
location = new  NodeType<char>;
location->info = newItem;
location->next = topPtr;
topPtr = location;
```

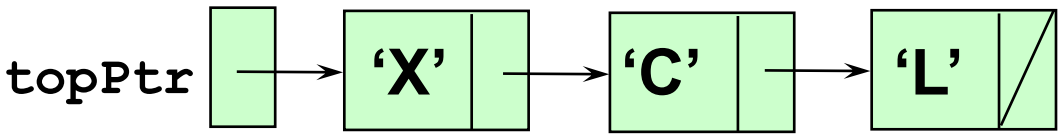


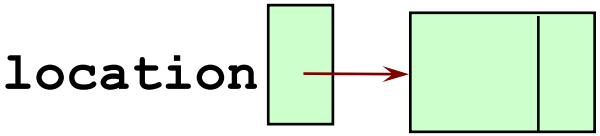
location

Adding newItem to the stack

newItem ‘B’

```
newItem = 'B';
NodeType*  location;
location = new  NodeType<char>;
location->info = newItem;
location->next = topPtr;
topPtr = location;
```

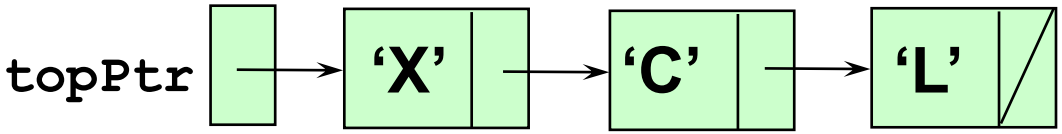


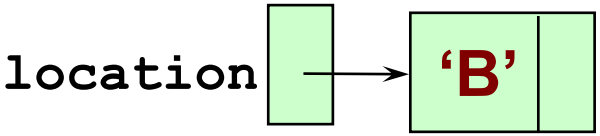


Adding newItem to the stack

newItem ‘B’

```
newItem = 'B';
NodeType*  location;
location = new  NodeType<char>;
location->info = newItem;
location->next = topPtr;
topPtr = location;
```

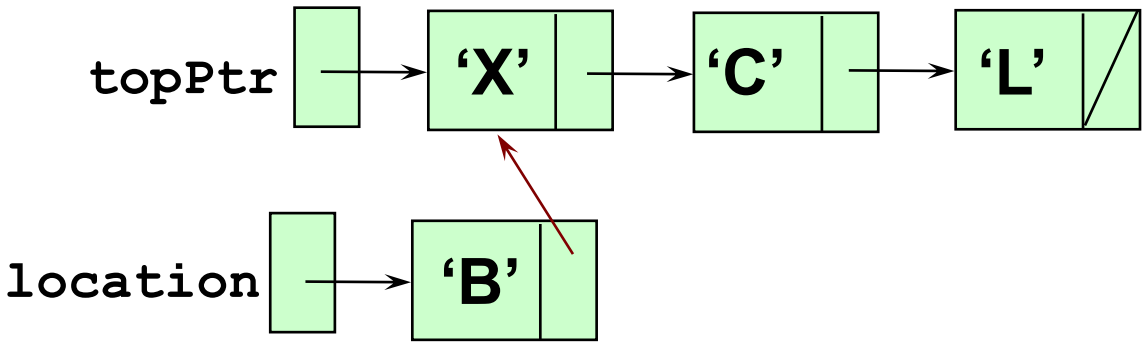




Adding newItem to the stack

newItem ‘B’

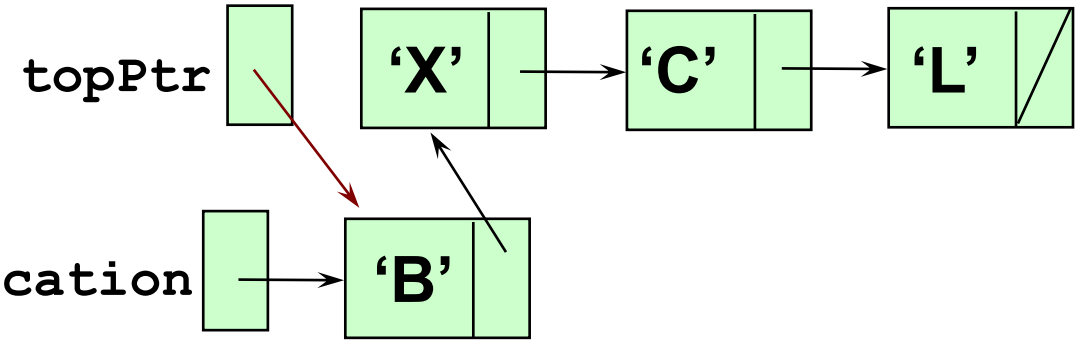
```
newItem = 'B';
NodeType*  location;
location = new  NodeType<char>;
location->info = newItem;
location->next = topPtr;
topPtr = location;
```



Adding newItem to the stack

newItem ‘B’

```
newItem = ‘B’;
NodeType*  location;
location = new  NodeType<char>;
location->info = newItem;
location->next = topPtr;
topPtr = location;
```



Implementing Push

```
void StackType::Push ( ItemType newItem )
   // Adds newItem to the top of the stack.
{
    if (IsFull())
      throw FullStack();
    NodeType*  location;
    location = new  NodeType<ItemType>;
    location->info = newItem;
    location->next = topPtr;
    topPtr = location;
}
```

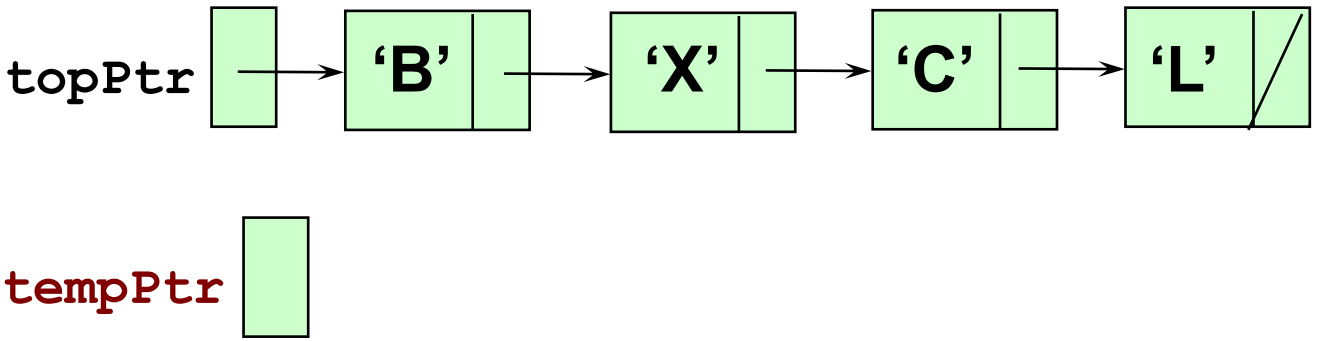
Using operator `delete`

The object currently pointed to by the pointer is deallocated, and the pointer is considered unassigned. The memory is returned to the free store.

Deleting item from the stack

item

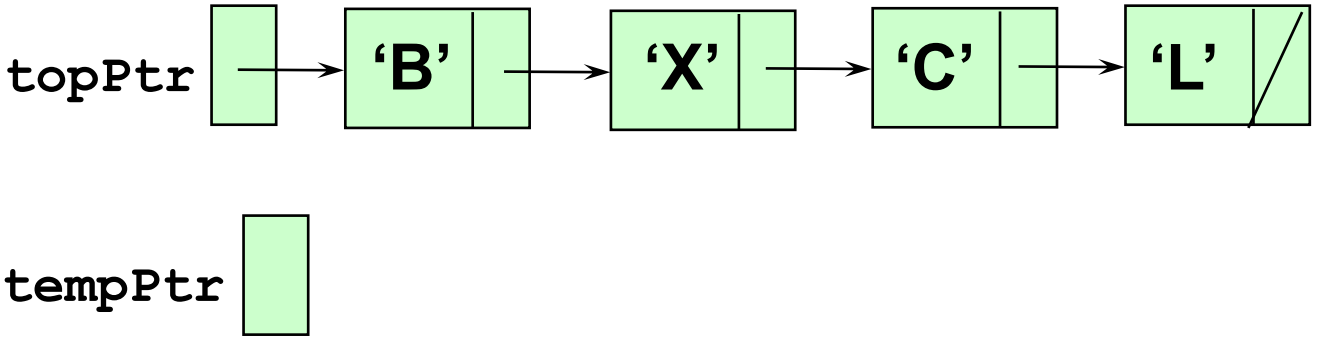
```
NodeType*   tempPtr;

item = topPtr->info;
tempPtr = topPtr;
topPtr = topPtr->next;
delete  tempPtr;
```



Deleting item from the stack

item ‘B’

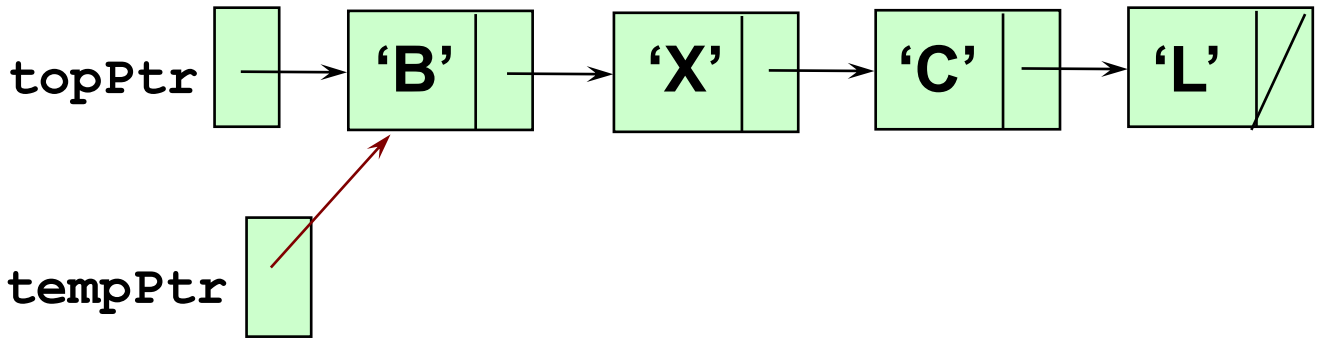
```
NodeType*  tempPtr;

item = topPtr->info;

tempPtr = topPtr;
topPtr = topPtr->next;
delete  tempPtr;
```



Deleting item from the stack

item ‘B’

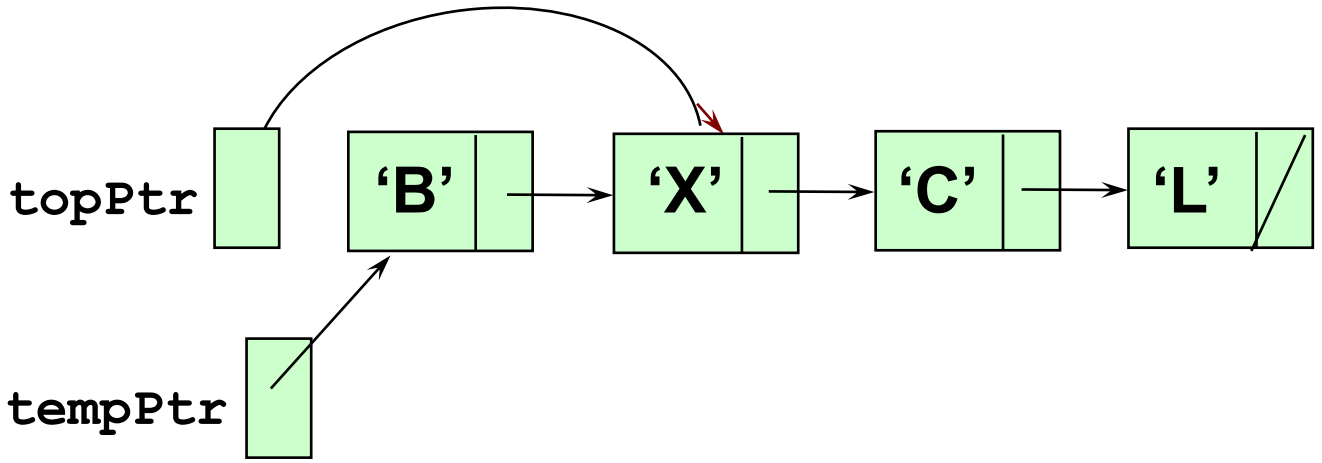
```
NodeType*  tempPtr;

item = topPtr->info;
tempPtr = topPtr;
topPtr = topPtr->next;
delete  tempPtr;
```



Deleting item from the stack

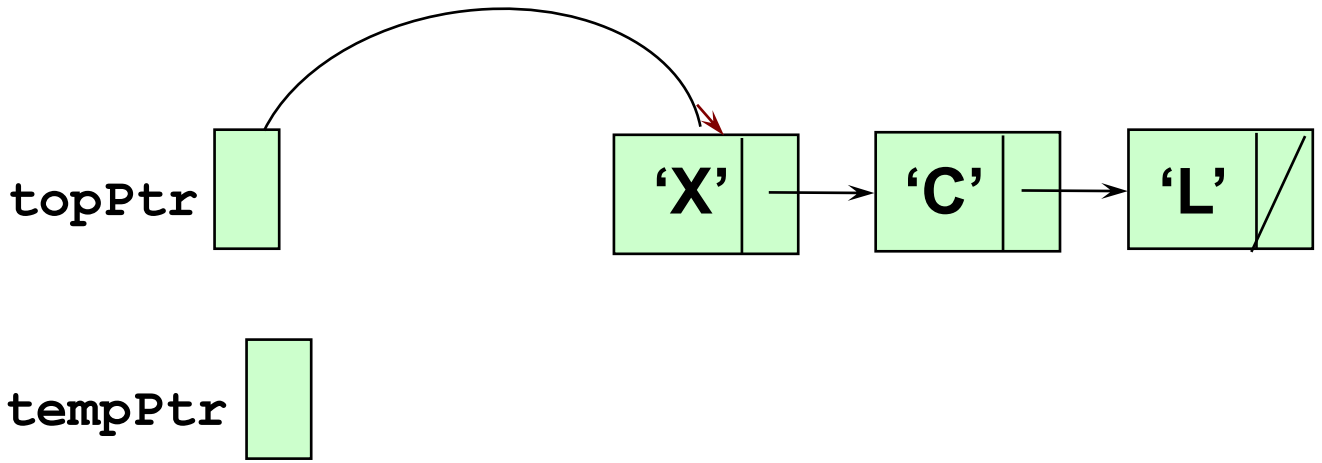
item ‘B’

```
NodeType*  tempPtr;

item = topPtr->info;
tempPtr = topPtr;
topPtr = topPtr->next;
delete  tempPtr;
```



item ‘B’

```
NodeType<ItemType>*   tempPtr;

item = topPtr->info;
tempPtr = topPtr;
topPtr = topPtr->next;
delete  tempPtr;
```



Implementing Pop / Top

```
void StackType::Pop()           // Remove top item from Stack.
{
   if (IsEmpty())
      throw EmptyStack();
   else
   {
      NodeType* tempPtr;
      tempPtr = topPtr;
      topPtr = topPtr ->next;
      delete tempPtr;
   }
}

ItemType StackType::Top()
// Returns a copy of the top item in the stack.
{
   if (IsEmpty())
      throw EmptyStack();
   else
      return topPtr->info;
}
```

Implementing IsFull

```
bool StackType::IsFull()  const
// Returns true if there is no room for another
// ItemType on the free store; false otherwise
{
   NodeType* location;
   try
   {
      location = new NodeType;
      delete location;
      return false;
    }
    catch(std::bad_alloc exception)
    {
      return true;
    }
}
```

Why is a destructor needed?

When a local stack variable goes out of scope, the memory space for data member topPtr is deallocated. But the nodes that topPtr points to are not automatically deallocated.

A class destructor is used to deallocate the dynamic memory pointed to by the data member.

Implementing the Destructor

```
stackType::~StackType()
// Post: stack is empty;
// All items have been deallocated.
{
  NodeType* tempPtr;

  while (topPtr != NULL)
  {
    tempPtr = topPtr;
    topPtr = topPtr-> next;
    delete tempPtr;
  }
}
```

Comparing stack implementations

Big-O Comparison of Stack Operations		
Operation	Array Implementation	Linked Implementation
Class constructor	O(1)	O(1)
MakeEmpty	O(1)	O(N)
IsFull	O(1)	O(1)
IsEmpty	O(1)	O(1)
Push	O(1)	O(1)
Pop	O(1)	O(1)
Destructor	O(1)	O(N)

What is a Queue?

- ***Logical (or ADT) level:*** A queue is an ordered group of homogeneous items (elements), in which new elements are added at one end (the rear), and elements are removed from the other end (the front).
- A queue is a FIFO "first in, first out" structure.

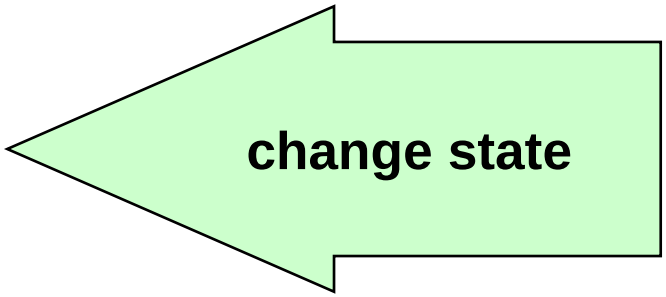
Queue ADT Operations

- **MakeEmpty -- Sets queue to an empty state.**
- **IsEmpty -- Determines whether the queue is currently empty.**
- **IsFull -- Determines whether the queue is currently full.**
- **Enqueue (ItemType newItem) -- Adds newItem to the rear of the queue.**
- **Dequeue (ItemType& item) -- Removes the item at the front of the queue and returns it in item.**

ADT Queue Operations

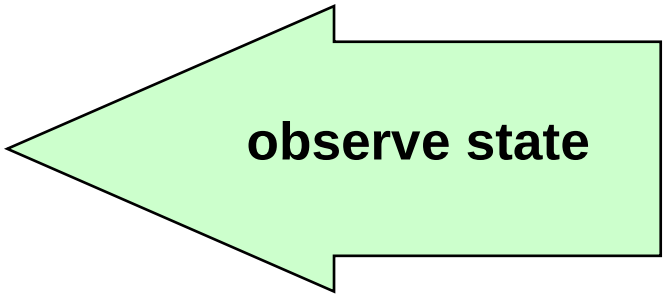
Transformers

- MakeEmpty
- Enqueue
- Dequeue



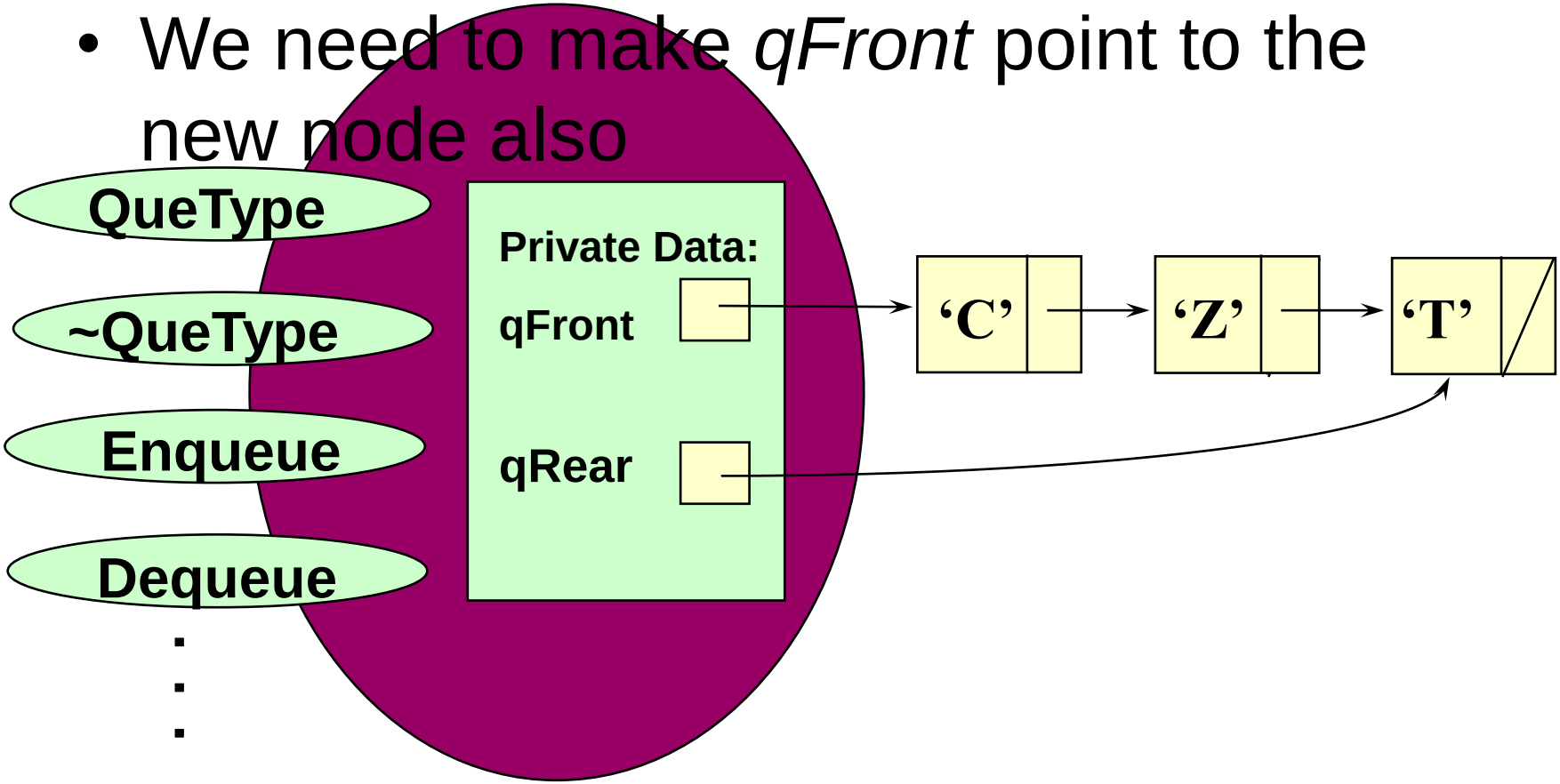
Observers

- IsEmpty
- IsFull



class QueType<char>

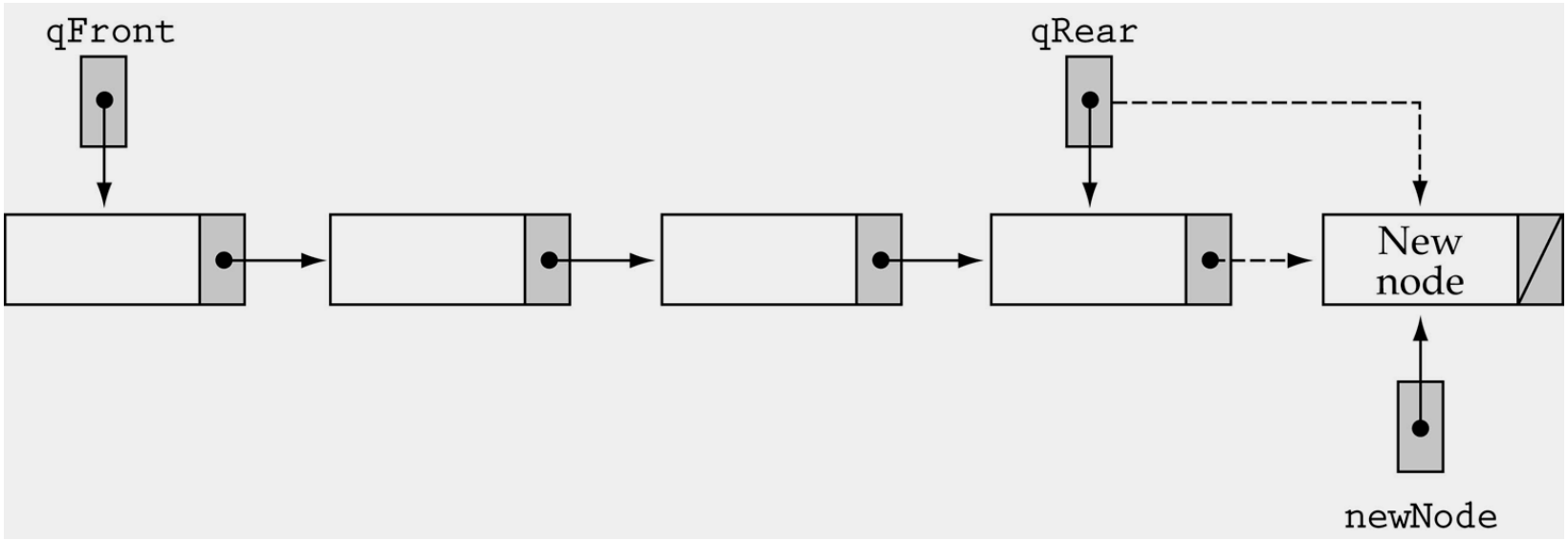
- We need to make *qFront* point to the new node also



```
// DYNAMICALLY LINKED IMPLEMENTATION OF QUEUE

#include "ItemType.h"        // for ItemType

template<class ItemType>
class QueType  {
public:
   QueType( );                    // CONSTRUCTOR
   ~QueType( ) ;                  // DESTRUCTOR
   bool IsEmpty( ) const;
   bool IsFull( ) const;
   void Enqueue( ItemType item );
   void Dequeue( ItemType&  item );
   void MakeEmpty( );
private:
   NodeType<ItemType>*  qFront;
   NodeType<ItemType>*  qRear;
};
```

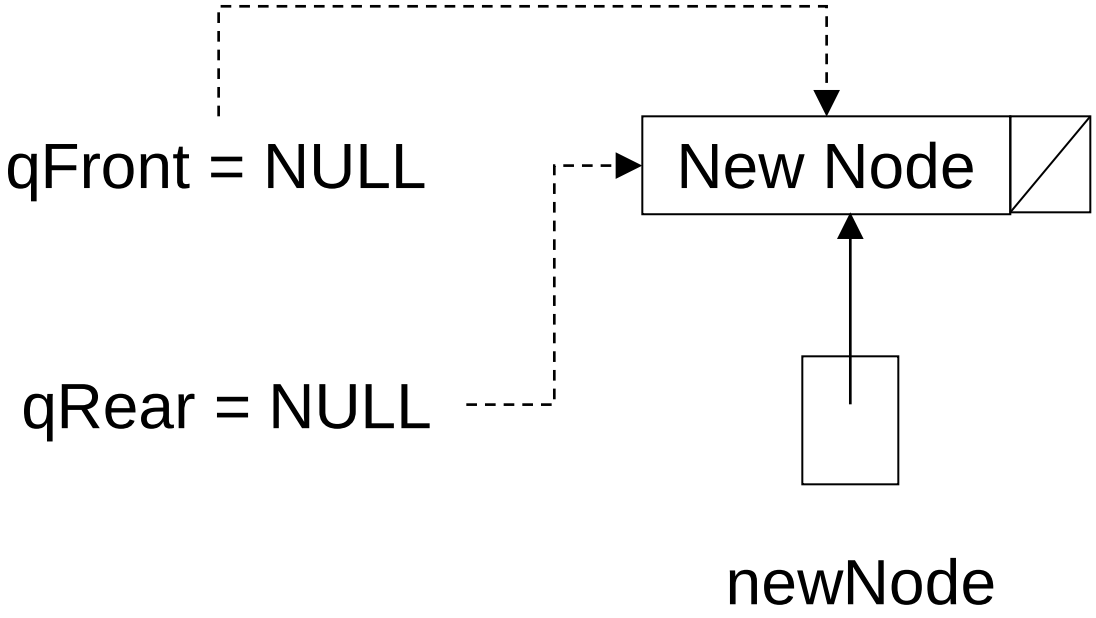
```
// DYNAMICALLY LINKED IMPLEMENTATION OF QUEUE   continued
// member function definitions for class QueType

template<class ItemType>
QueType<ItemType>::QueType( )      // CONSTRUCTOR
{
   qFront = NULL;
   qRear = NULL;
}

template<class ItemType>
bool QueType<ItemType>::IsEmpty( ) const
{
   return ( qFront == NULL )
}
```

Enqueuing (non-empty queue)



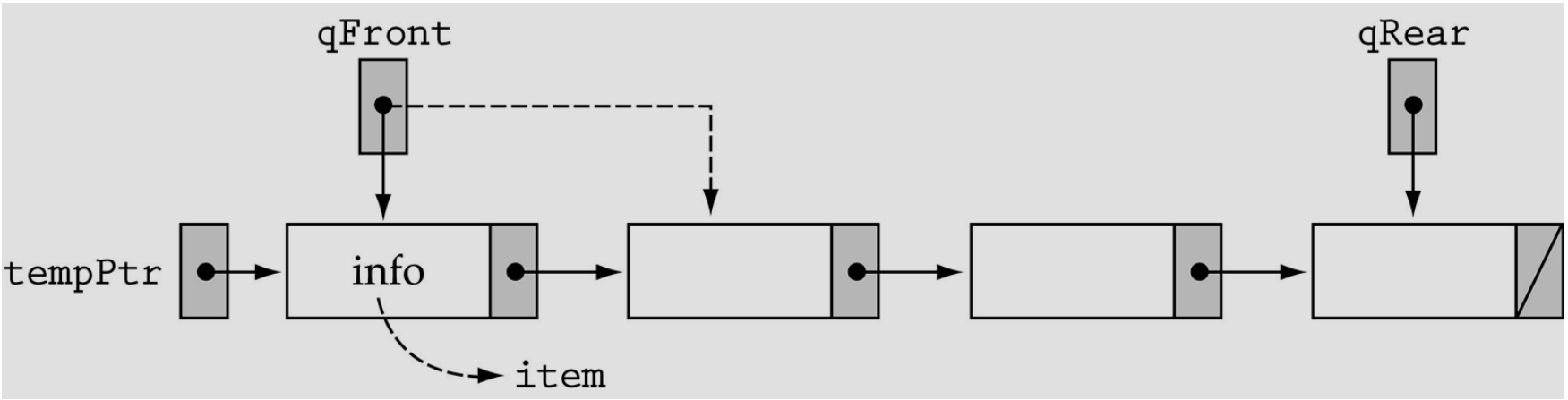
Enqueuing (empty queue)

- We need to make *qFront* point to the new node also



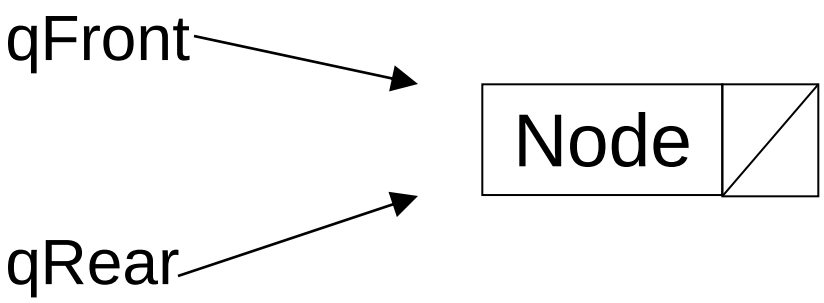
```
template<class ItemType>
void QueType<ItemType>::Enqueue( ItemType newItem )
        // Adds newItem to the rear of the queue.
        // Pre:  Queue has been initialized.
        //       Queue is not full.
        // Post: newItem is at rear of queue.
{
   NodeType<ItemType>*  ptr;

   ptr = new  NodeType<ItemType>;
   ptr->info = newItem;
   ptr->next = NULL;
   if   ( qRear == NULL )
        qFront = ptr;
   else
        qRear->next = ptr;
   qRear = ptr;
}
```

Dequeueing (the queue contains more than one element)



Dequeueing (the queue contains only one element)

- We need to reset *qRear* to NULL also



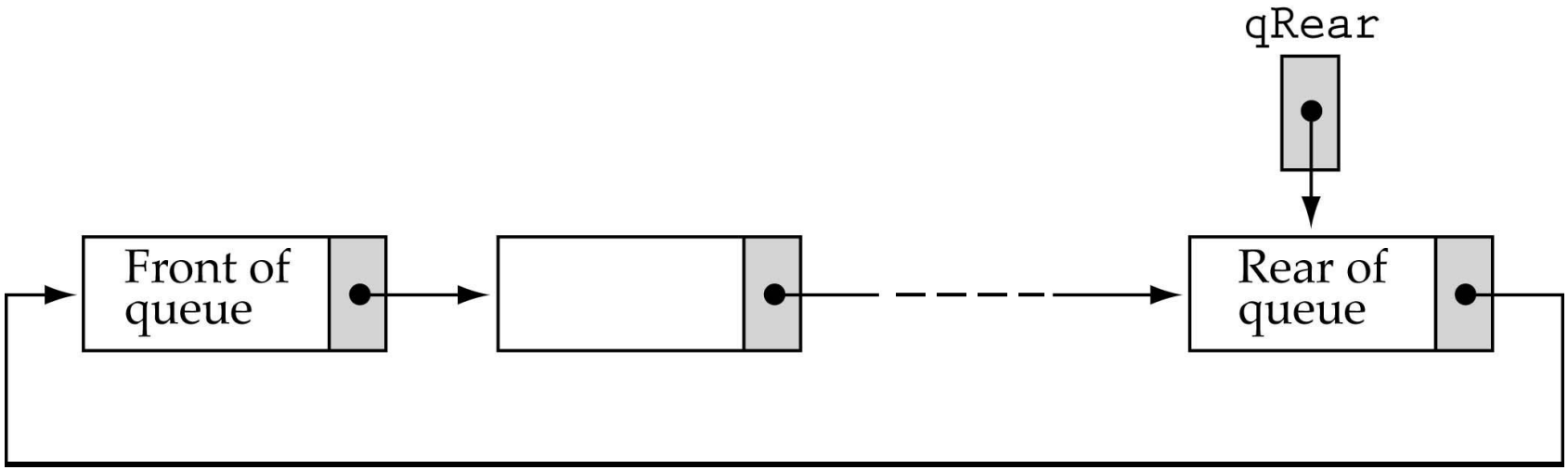
After dequeue:

qFront = NULL

qRear = NULL

```
template<class ItemType>
void QueType<ItemType>::Dequeue( ItemType& item )
       // Removes element from from front of queue
       // and returns it in item.
       // Pre:  Queue has been initialized.
       //       Queue is not empty.
       // Post: Front element has been removed from queue.
       //        item is a copy of removed element.
{
   NodeType<ItemType>*  tempPtr;

   tempPtr = qFront;
   item = qFront->info;
   qFront = qFornt->next;
   if   ( qFront == NULL )
        qRear = NULL;
   delete  tempPtr;
}
```

A circular linked queue design



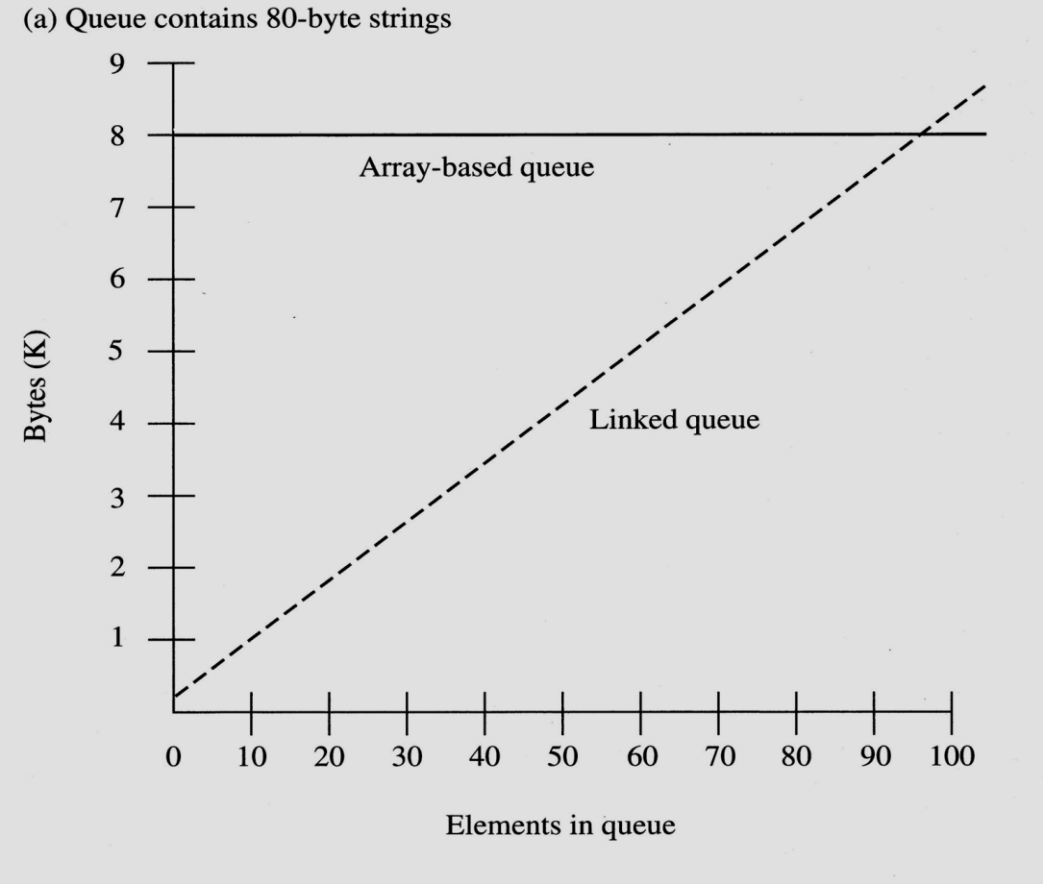
We can access both ends of the queue from a single pointer

Comparing queue implementations: Example 1

Memory requirements

- Array-based implementation
  - Assume a queue (size: 100) of strings (80 bytes each)
  - Assume indices take 2 bytes
  - Total memory: (80 bytes x 101 slots) + (2 bytes x 2 indexes) = 8084 bytes
- Linked-list-based implementation
  - Assume pointers take 4 bytes
  - Total memory per node: 80 bytes + 4 bytes = 84 bytes

Comparing queue implementations: Example 1 (cont'd)

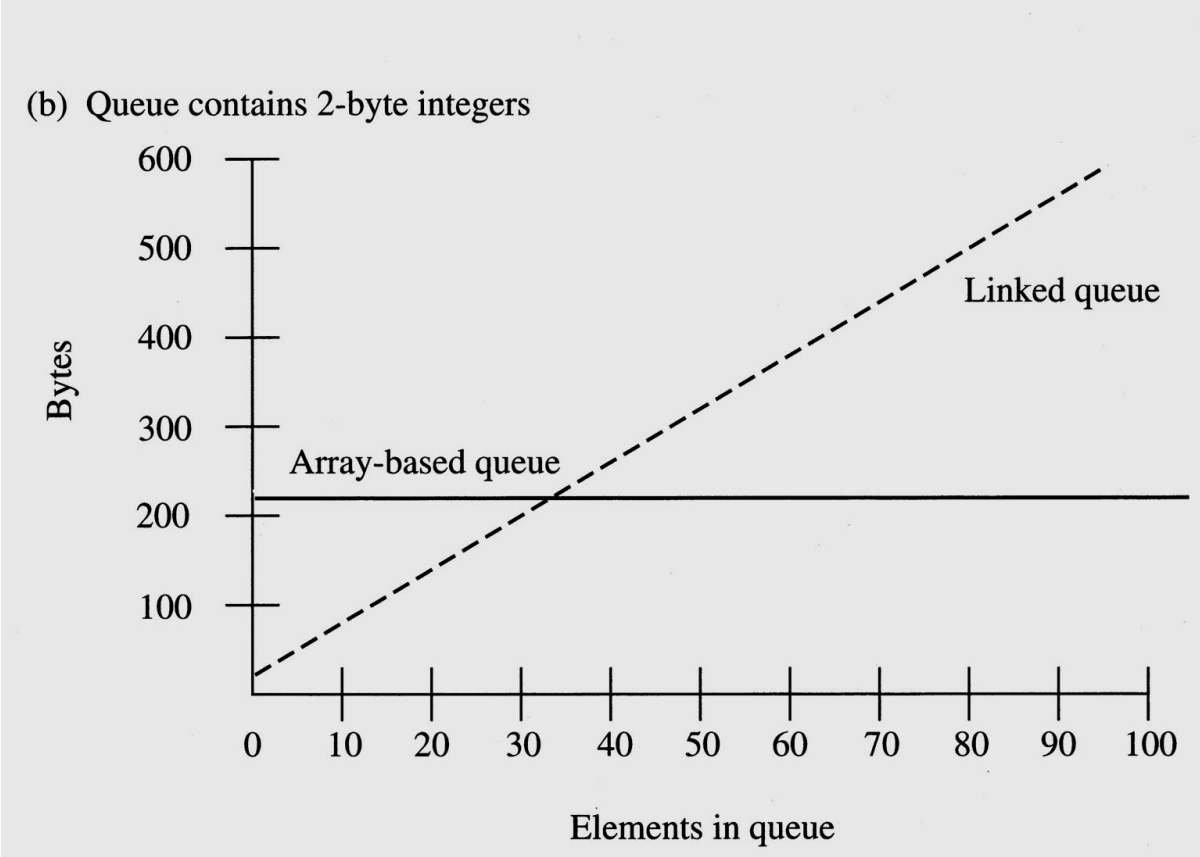


Comparing queue implementations: Example 2

Memory requirements

- Array-based implementation
  - Assume a queue (size: 100) of short integers (2 bytes each)
  - Assume indices take 2 bytes
  - Total memory: (2 bytes x 101 slots) + (2 bytes x 2 indexes) = 206 bytes
- Linked-list-based implementation
  - Assume pointers take 4 bytes
  - Total memory per node: 2 bytes + 4 bytes = 6 bytes

Comparing queue implementations: Example 2 (cont'd)



Comparing queue implementations

Big-O Comparison of Queue Operations		
Operation	Array Implementation	Linked Implementation
Class constructor	O(1)	O(1)
MakeEmpty	O(1)	O(N)
IsFull	O(1)	O(1)
IsEmpty	O(1)	O(1)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Destructor	O(1)	O(N)

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.

ADT Unsorted List Operations

Transformers

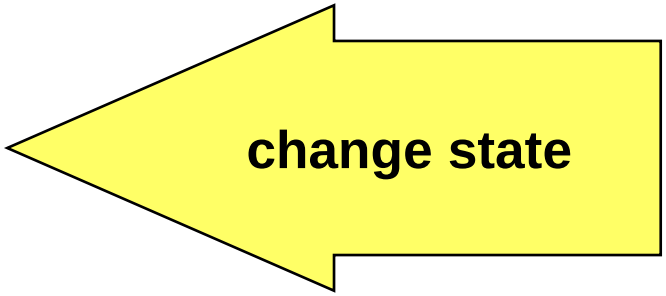
- MakeEmpty
- InsertItem
- DeleteItem

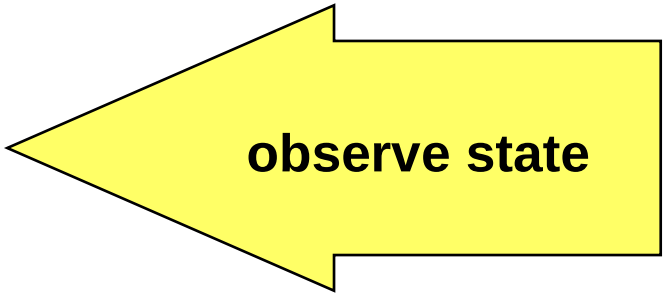
Observers

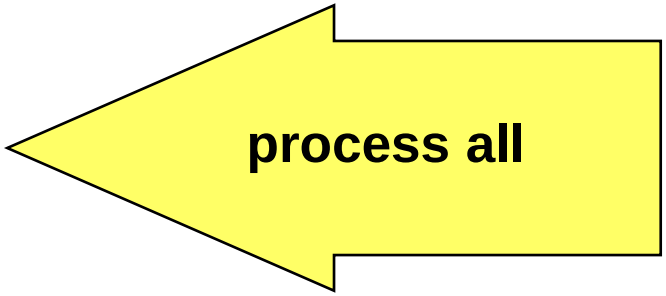
- IsFull
- LengthIs
- RetrieveItem

Iterators

- ResetList
- GetNextItem

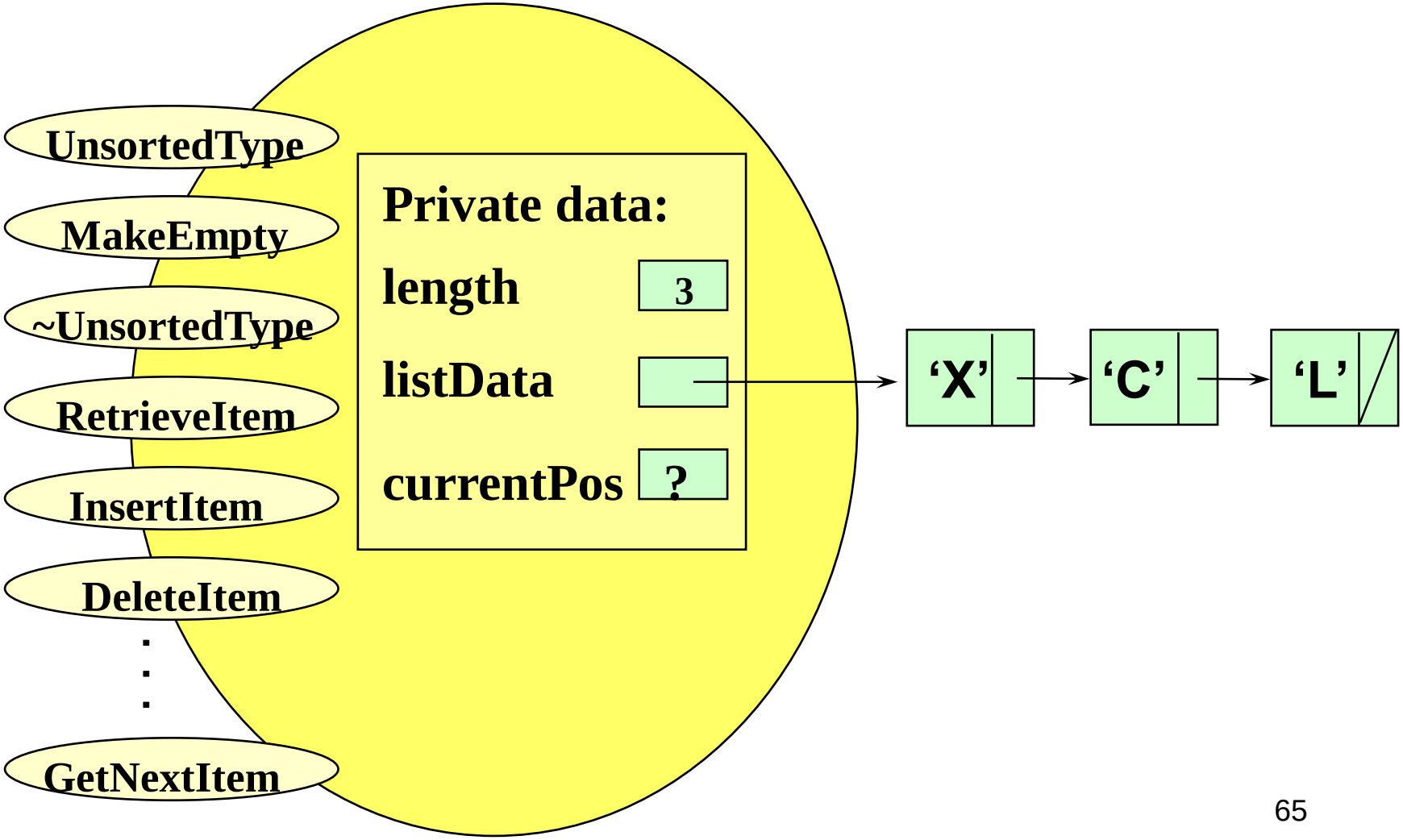






```
#include  "ItemType.h"          //  unsorted.h
    .   .   .
template <class ItemType>
class  UnsortedType
{
public :                        //  LINKED LIST IMPLEMENTATION
  UnsortedType ( ) ;
  ~UnsortedType ( ) ;
  void   MakeEmpty ( ) ;
  bool   IsFull ( )  const ;
  int    LengthIs ( )  const ;
  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;
} ;
```

class UnsortedType<char>

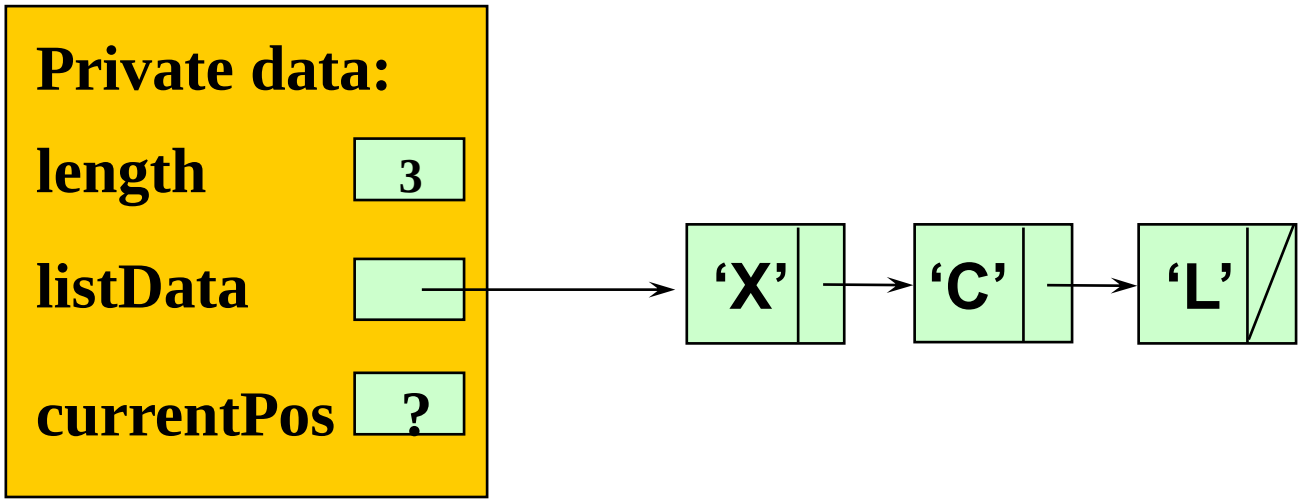


```
// LINKED LIST IMPLEMENTATION  ( unsorted.cpp )
#include "itemtype.h"

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 ) ;
      }

    }
}
```

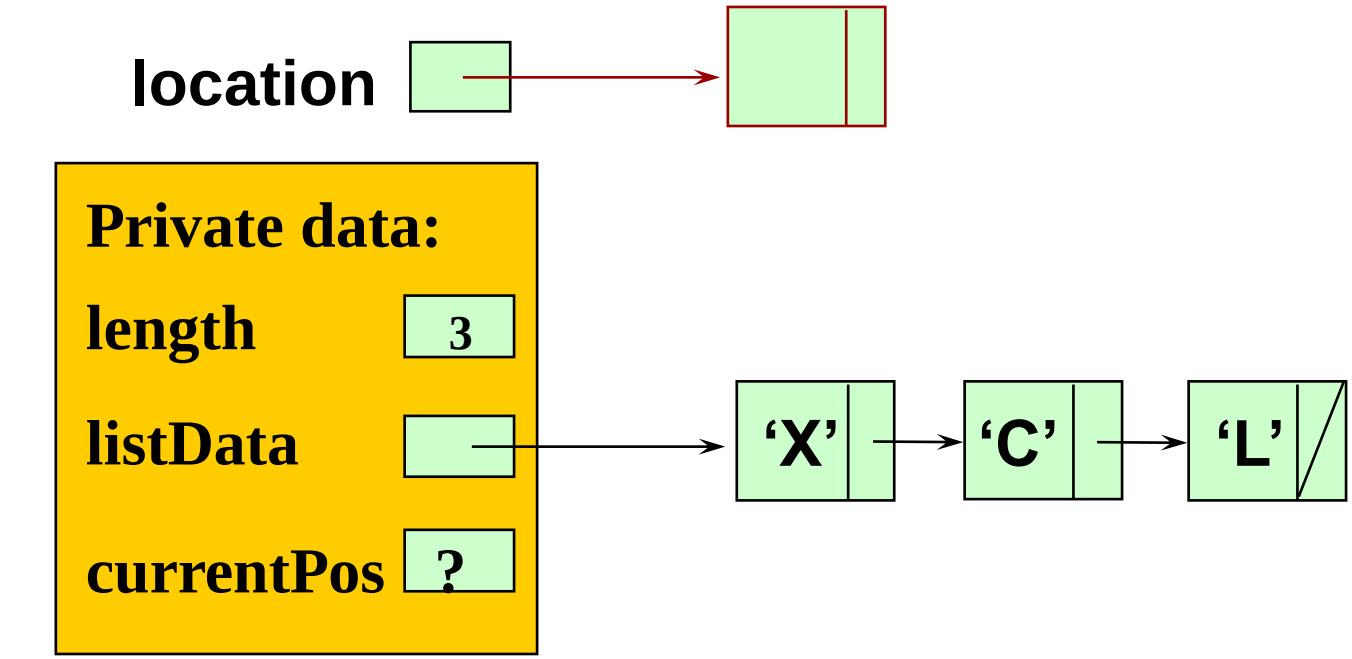
```
template <class ItemType>
void UnsortedType<ItemType>::InsertItem ( ItemType  item )
//  Pre: list is not full and item is not in list.
// Post: item is in the list; length has been incremented.
{
   NodeType<ItemType>*  location ;
   // obtain and fill a node
   location = new  NodeType<ItemType>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

Inserting ‘B’ into an Unsorted List

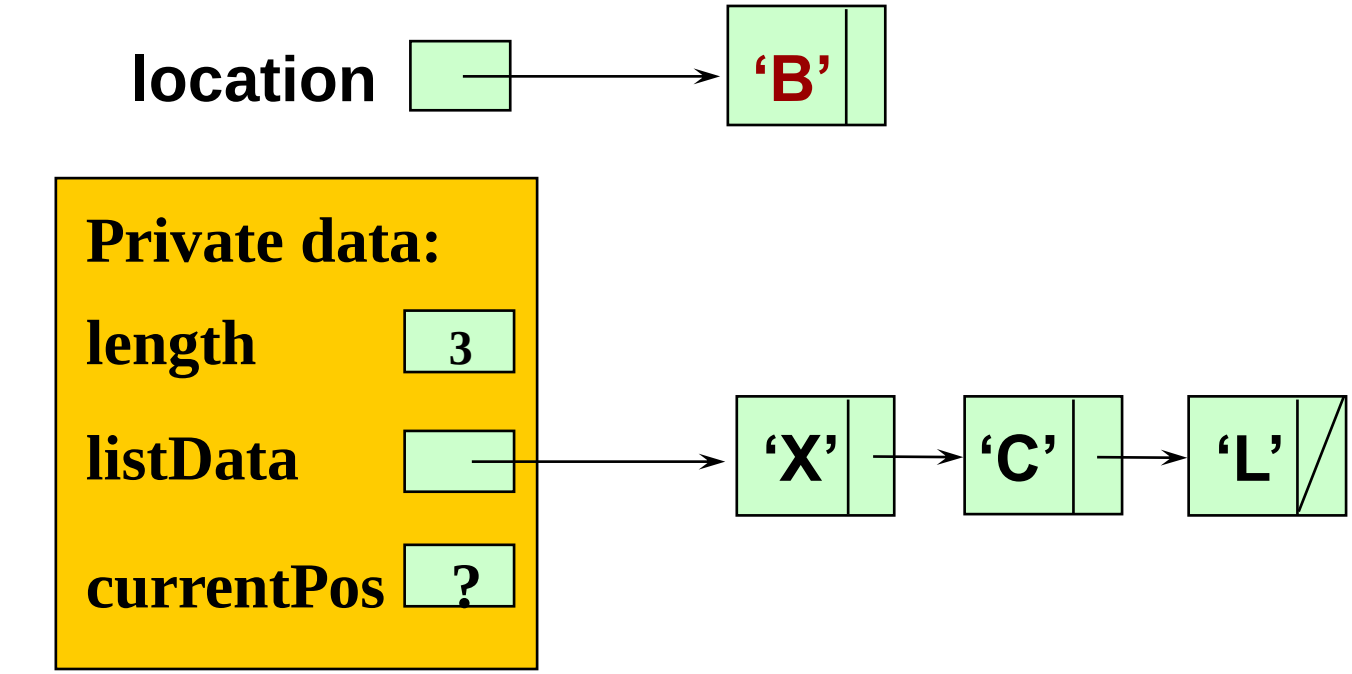


item ‘B’

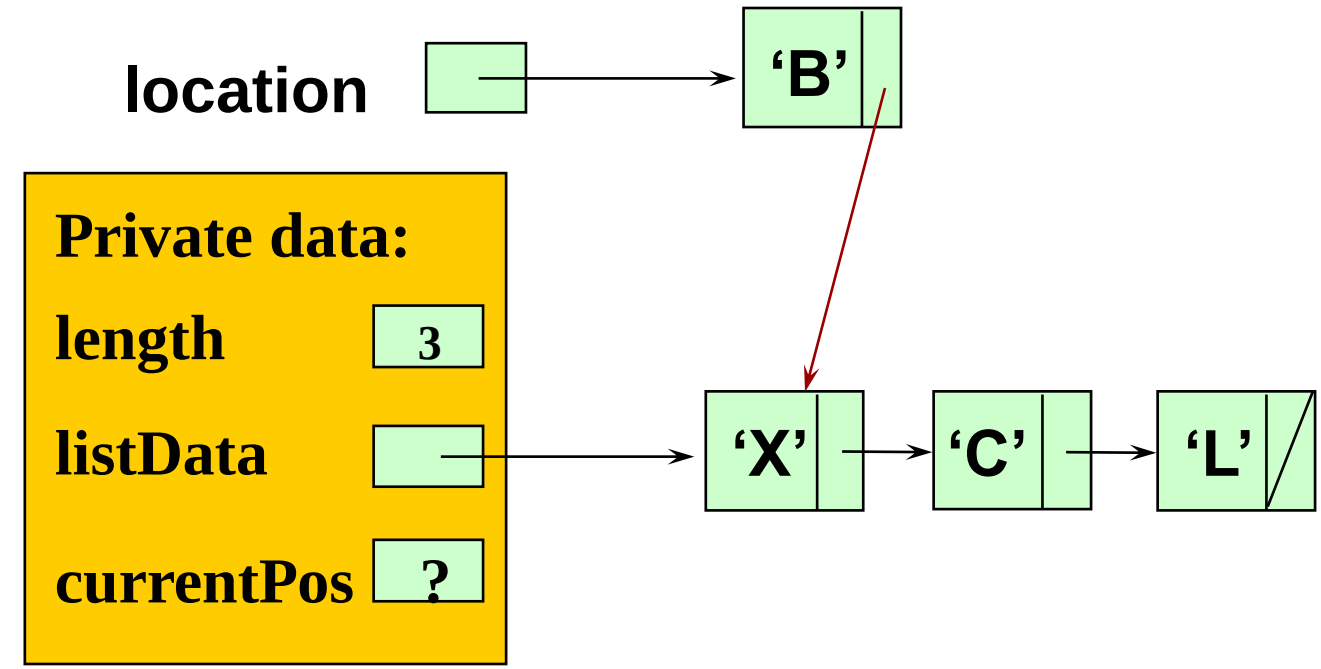
location = new NodeType<ItemType>;



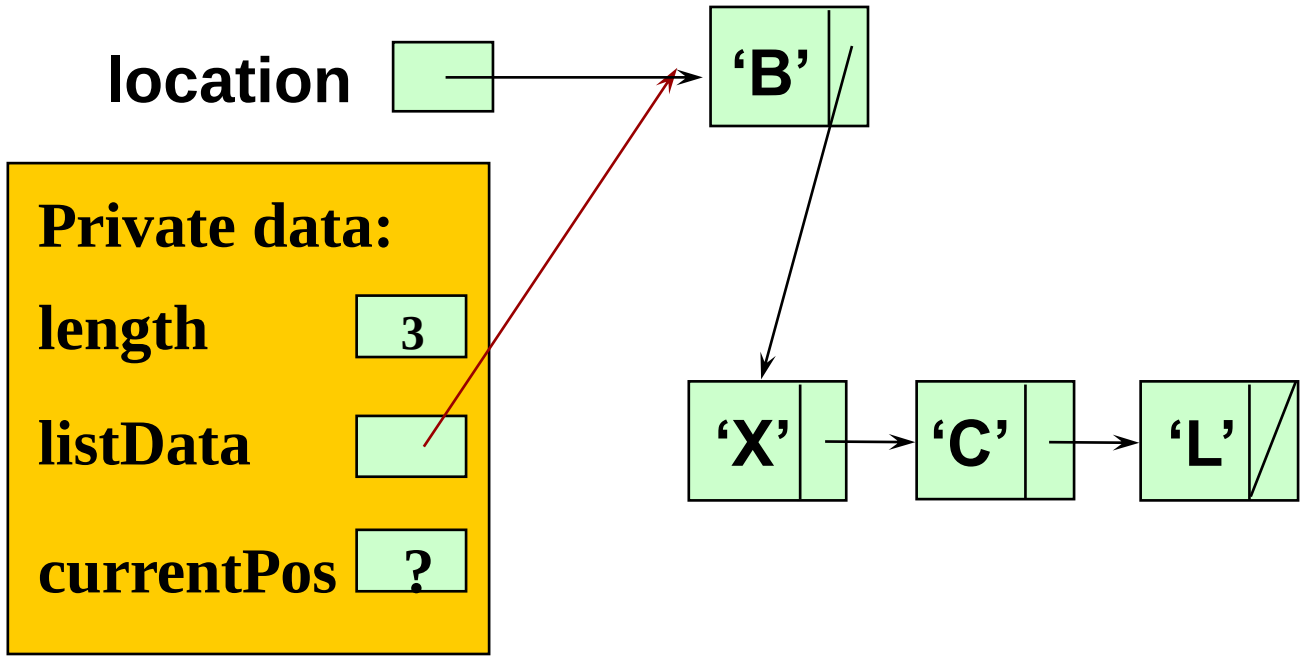
item ‘B’ location->info = item ;

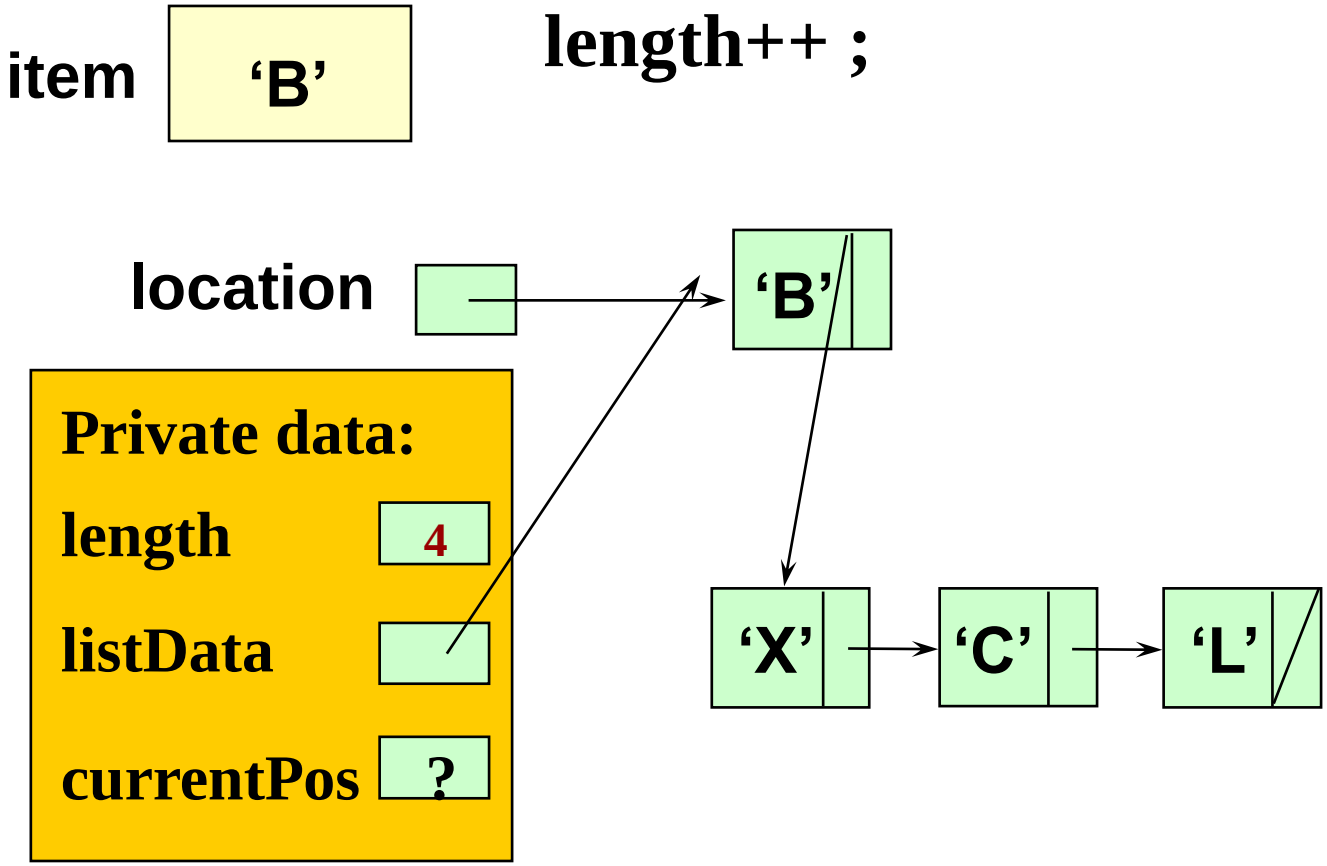


item ‘B’ location->next = listData ;



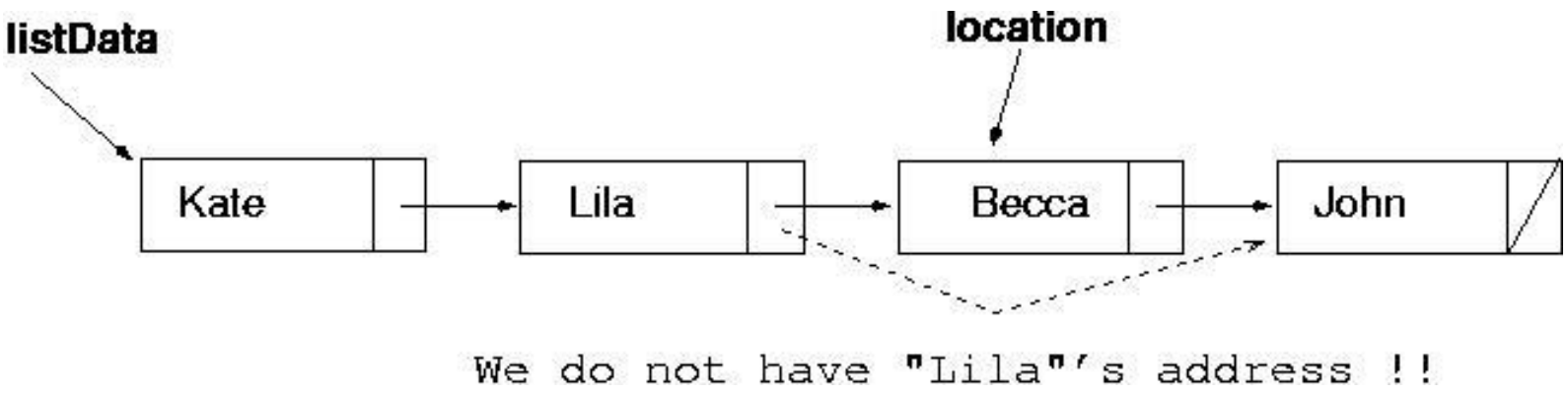
item ‘B’ listData = location ;




item
'B'
length++ ;
location
'B'
Private data:
length
4
listData
currentPos
?
'X'
'C'
'L'

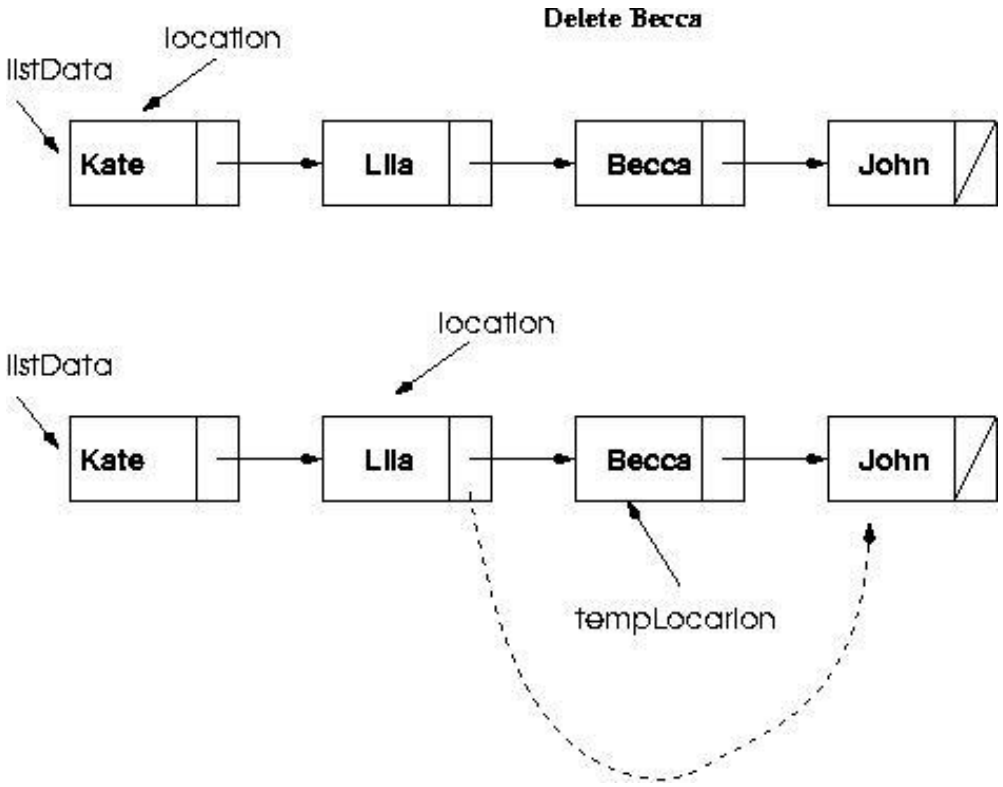
Function DeleteItem

- Find the item first
- In order to delete it, we must change the pointer in the *previous* node!!

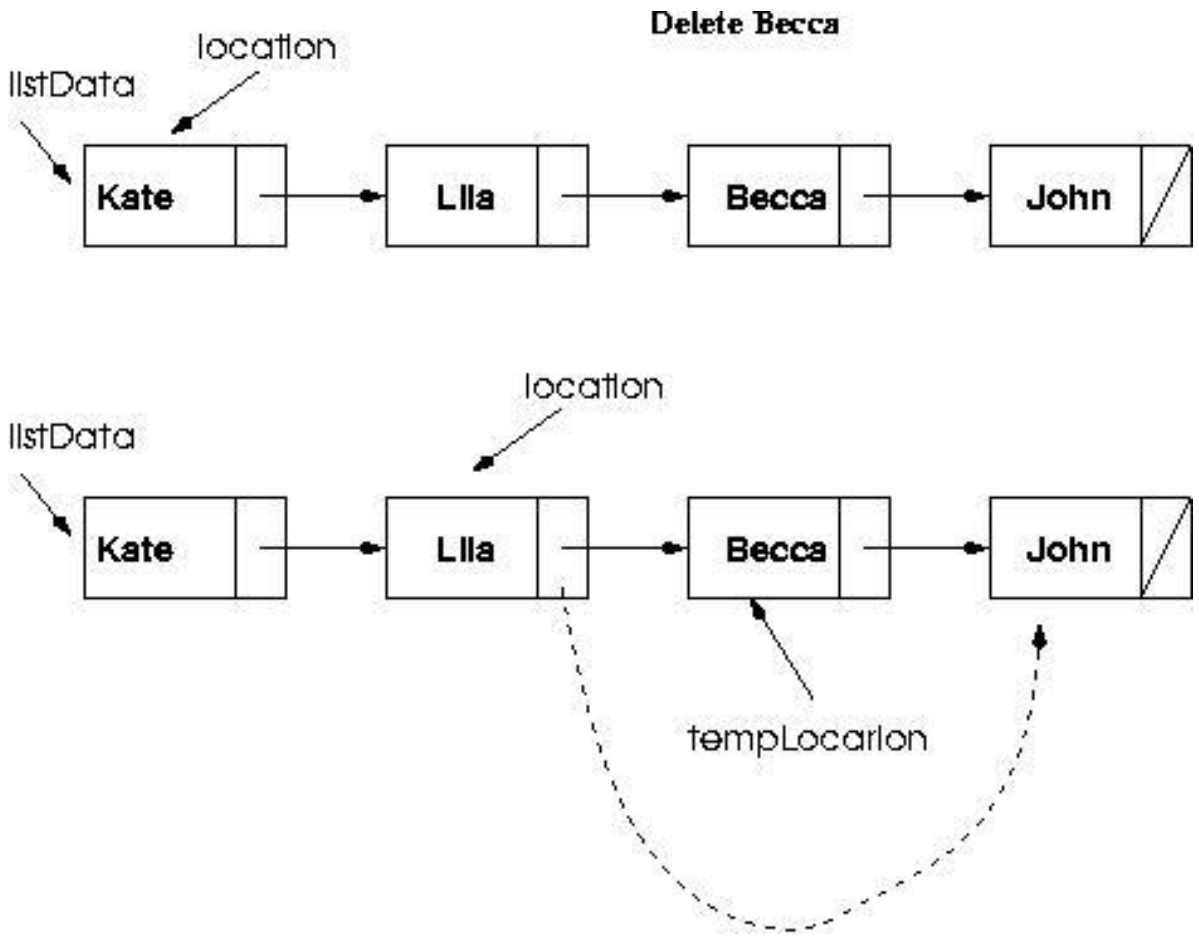


Function DeleteItem (cont.)

- Solution: compare **one item ahead** (*(location->next)->info)*) !!
- Deleting the first node is a special case ...



Function DeleteItem (cont.)



Important: This implementation will work without problems ONLY if the item to be deleted IS in the list ! (precondition)

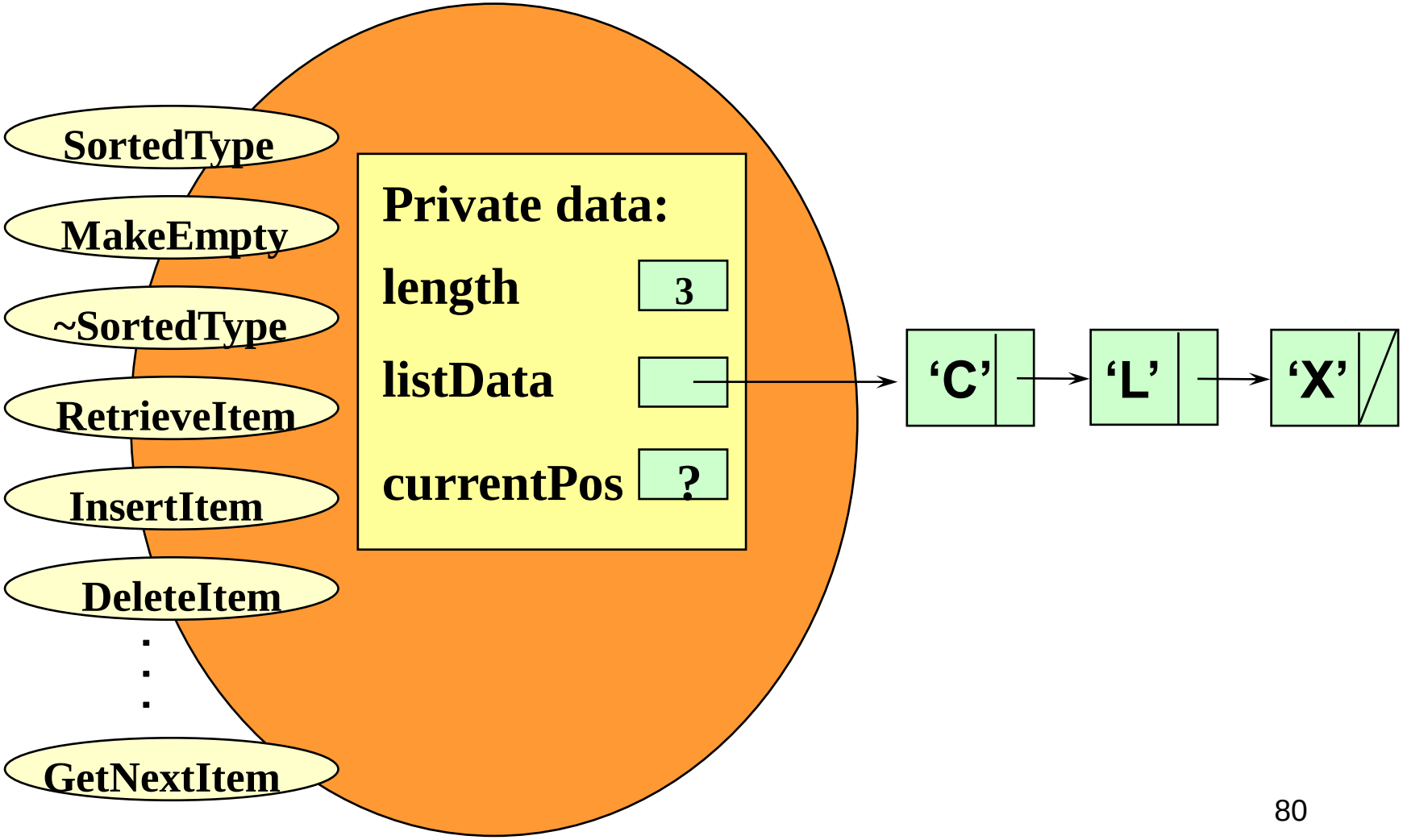
Function DeleteItem (cont.)

```
template <class ItemType>
void UnsortedType<ItemType>::DeleteItem(ItemType item)
{
 NodeType<ItemType>* location = listData;
 NodeType<ItemType>* tempLocation;

 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--;
}
```

Comparing unsorted list implementations

Big-O Comparison of Unsorted List Operations		
Operation	Array Implementation	Linked Implementation
Class constructor	O(1)	O(1)
Destructor	O(1)	O(N)
MakeEmpty	O(1)	O(N)
IsFull	O(1)	O(1)
LengthIs	O(1)	O(1)
ResetList	O(1)	O(1)
GetNextItem	O(1)	O(1)
RetrieveNextItem	O(N)	O(N)
InsertItem	O(1)	O(1)
DeleteItem	O(N)	O(N)

class SortedType<char>



Function RetrieveItem

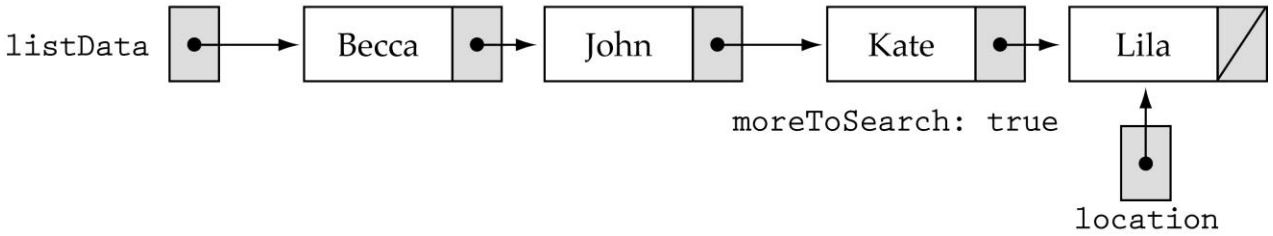
```
template<class ItemType>
void SortedType<ItemType>::RetrieveItem(ItemType&
   item, bool& found)
{
 NodeType<ItemType>* location;

 location = listData;
 found = false;

 while( (location != NULL) && !found) {

   if (location->info < item)
     location = location->next;
   else if (location->info == item) {
     found = true;
      item = location->info;
  }
   else
     location = NULL;   // no reason to continue
 }
}
```

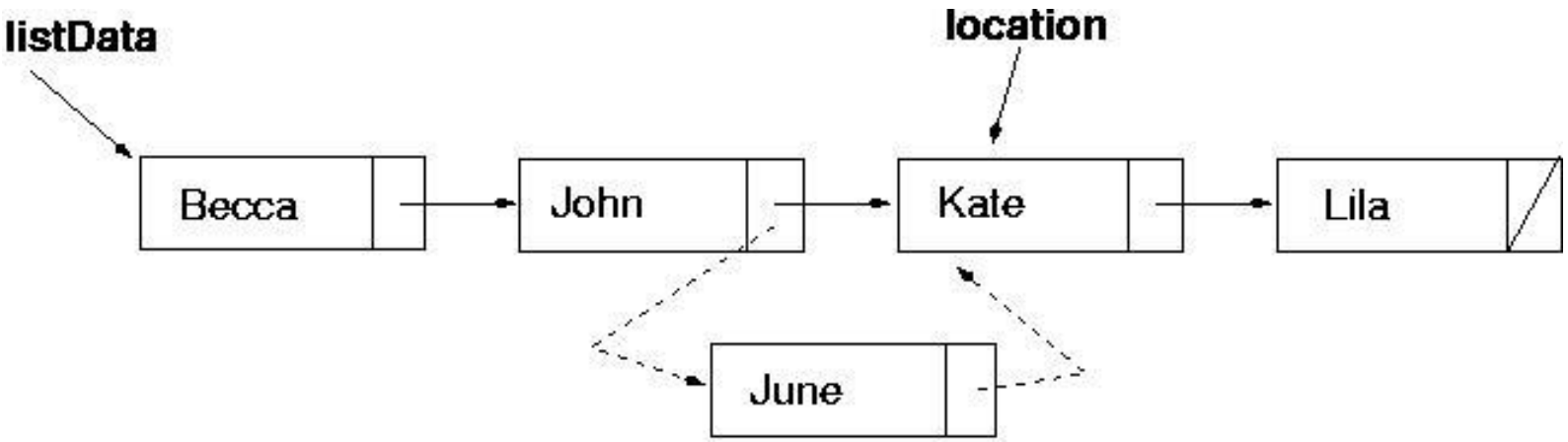
Can we use binary search with linked list?

Implementing SortedType member function InsertItem

```
// LINKED LIST IMPLEMENTATION                    (sorted.cpp)
#include "ItemType.h"

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.
{
                .
                .
                .

}
```

Problem with Implementing Function InsertItem





We do not have "John"'s address !!

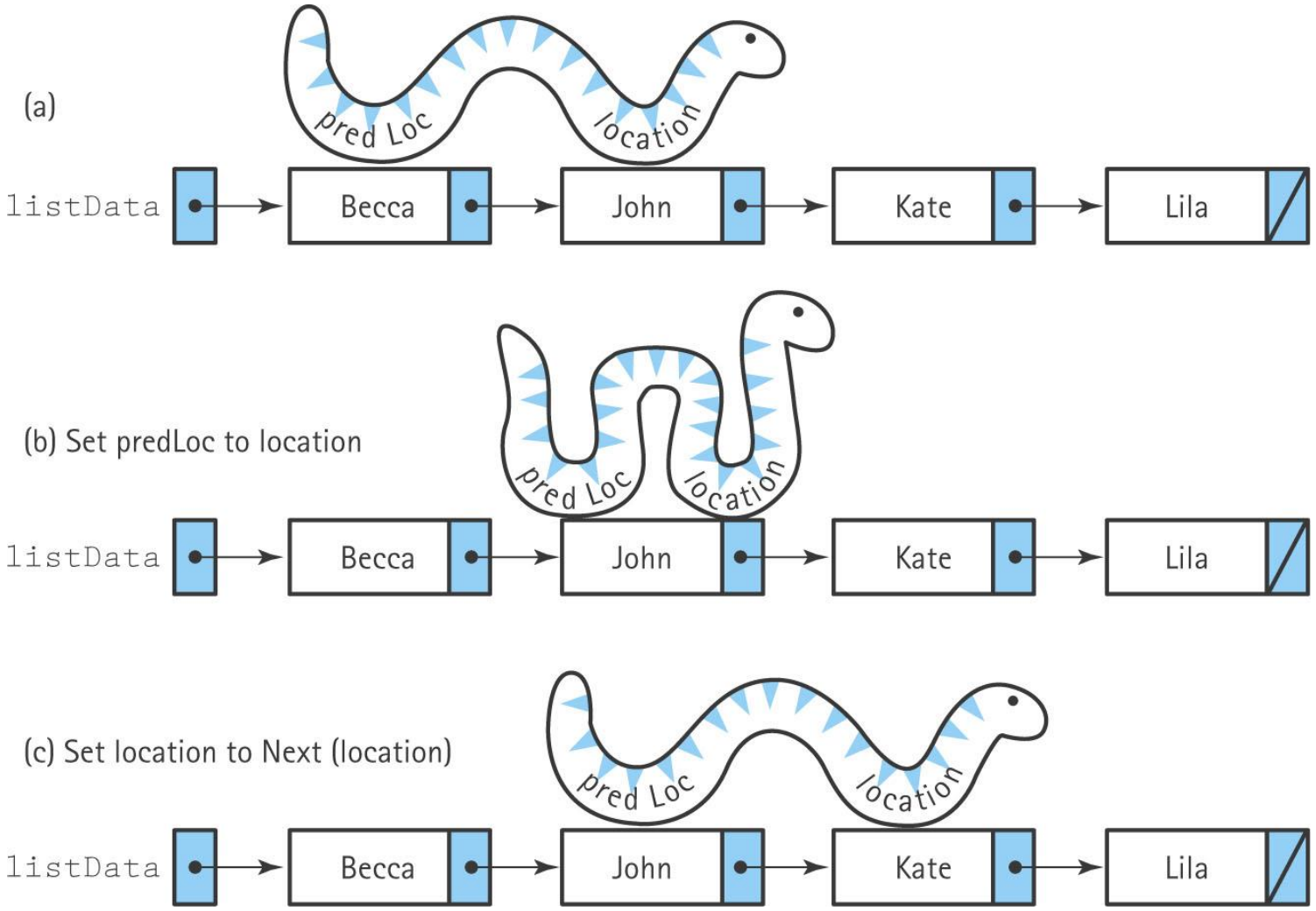
Solution

- Remind the technique of comparing one item ahead in the unsorted list case
  - (item == (location->next)->info)
- Can we use that technique??
  - NO!!! That implementation does not work
  - What if the new item was supposed to go at the end of the list?
- In general, we must keep track of the previous pointer, as well as the current pointer.

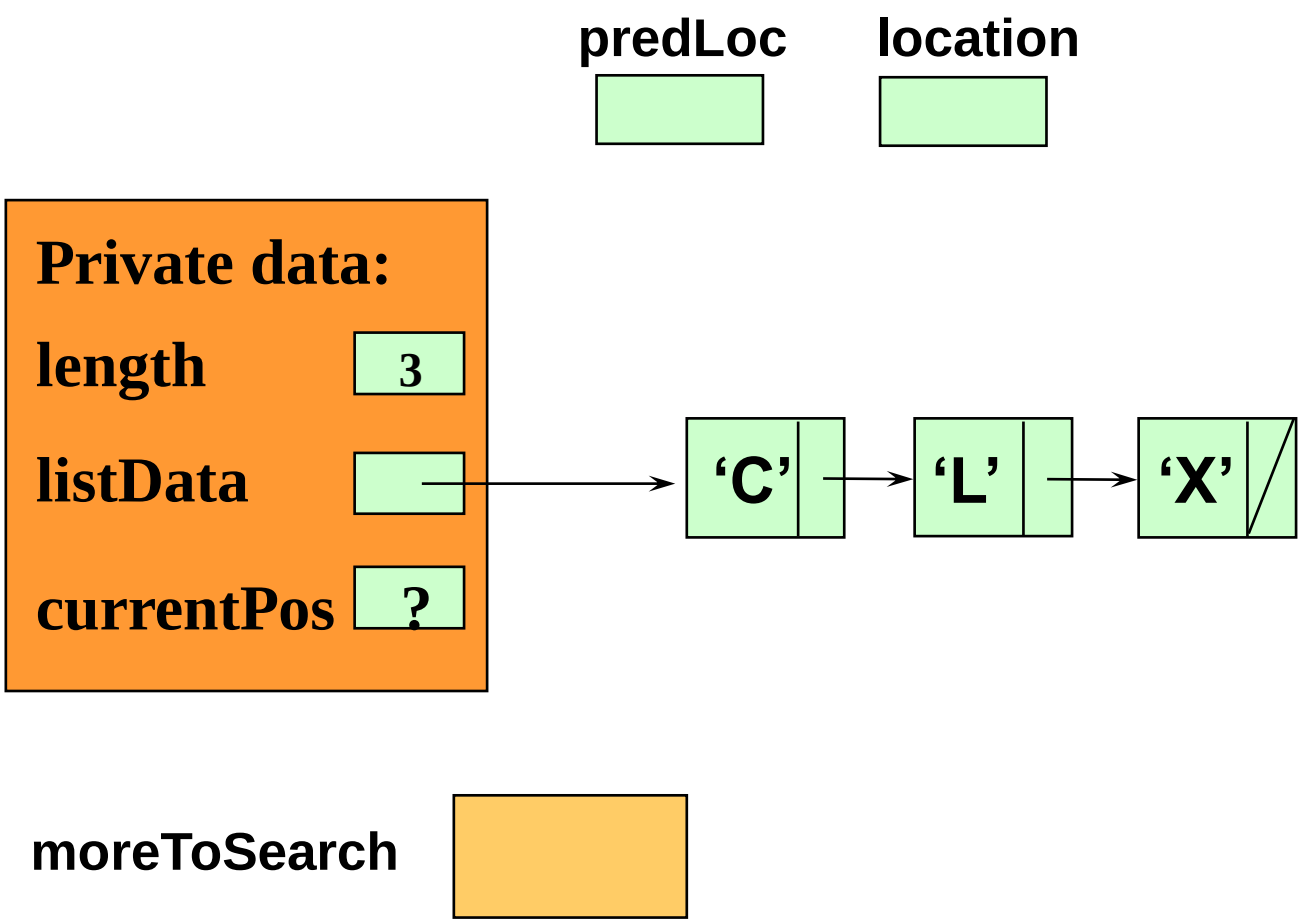
InsertItem algorithm for Sorted Linked List

- Find proper position for the new element in the sorted list using two pointers predLoc and location, where predLoc trails behind location.
- Obtain a node for insertion and place item in it.
- Insert the node by adjusting pointers.
- Increment length.

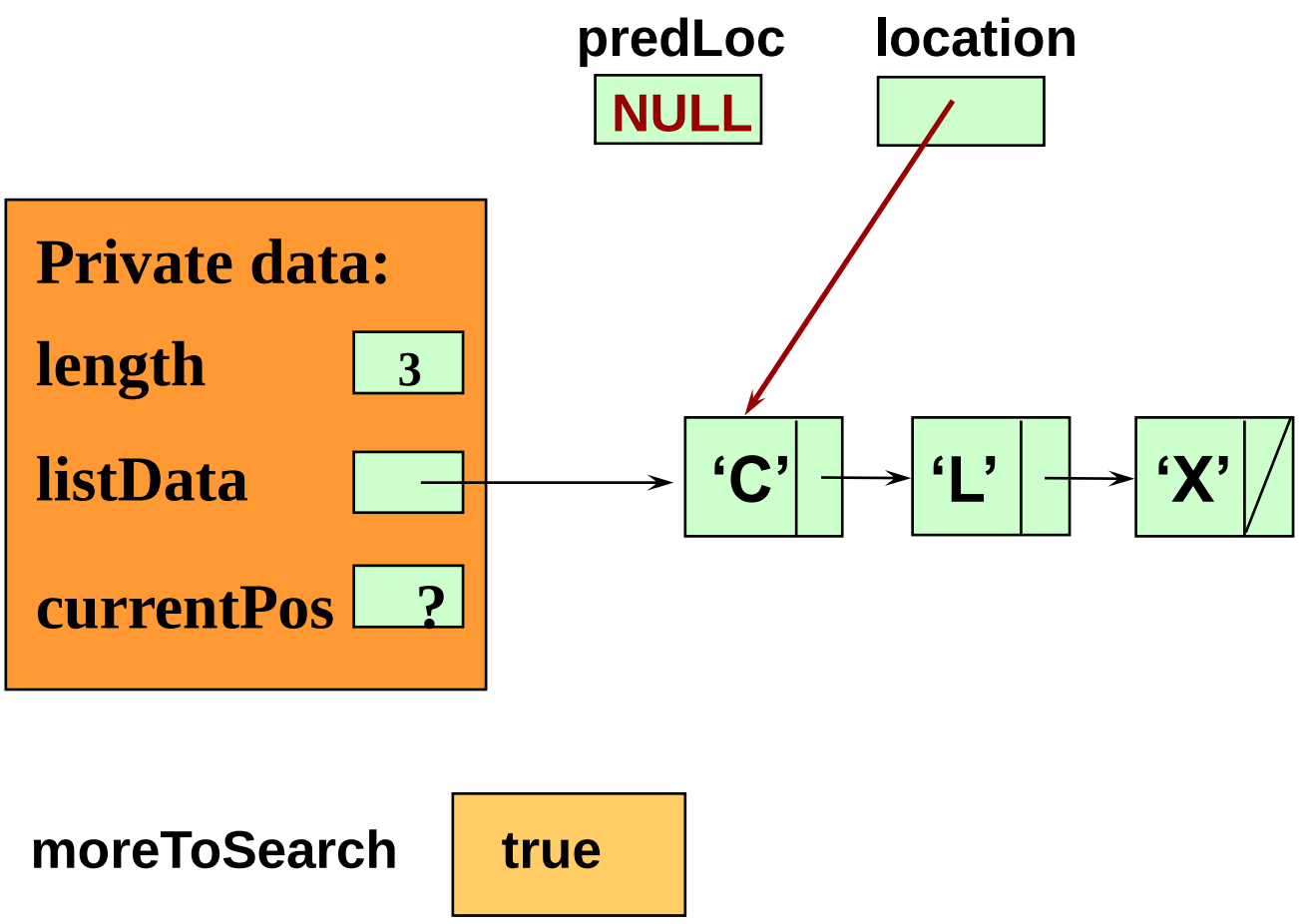
The Inchworm Effect



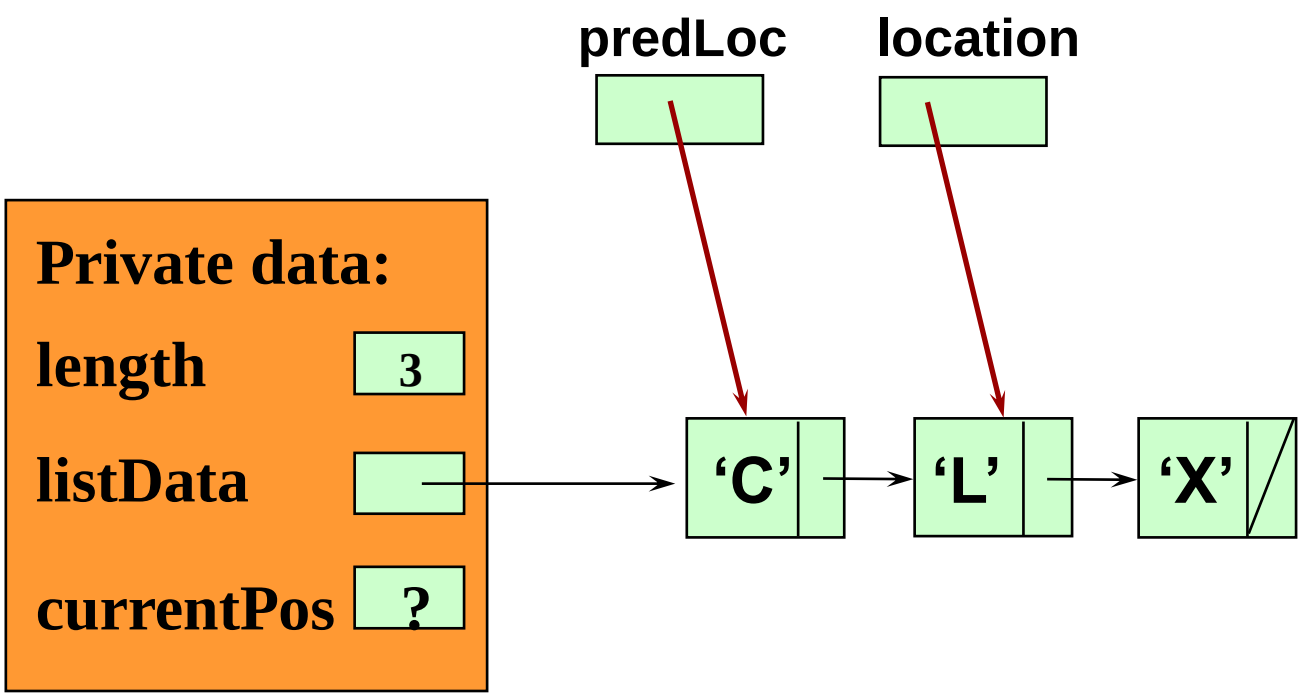
Inserting ‘S’ into a Sorted List



Finding proper position for ‘S’

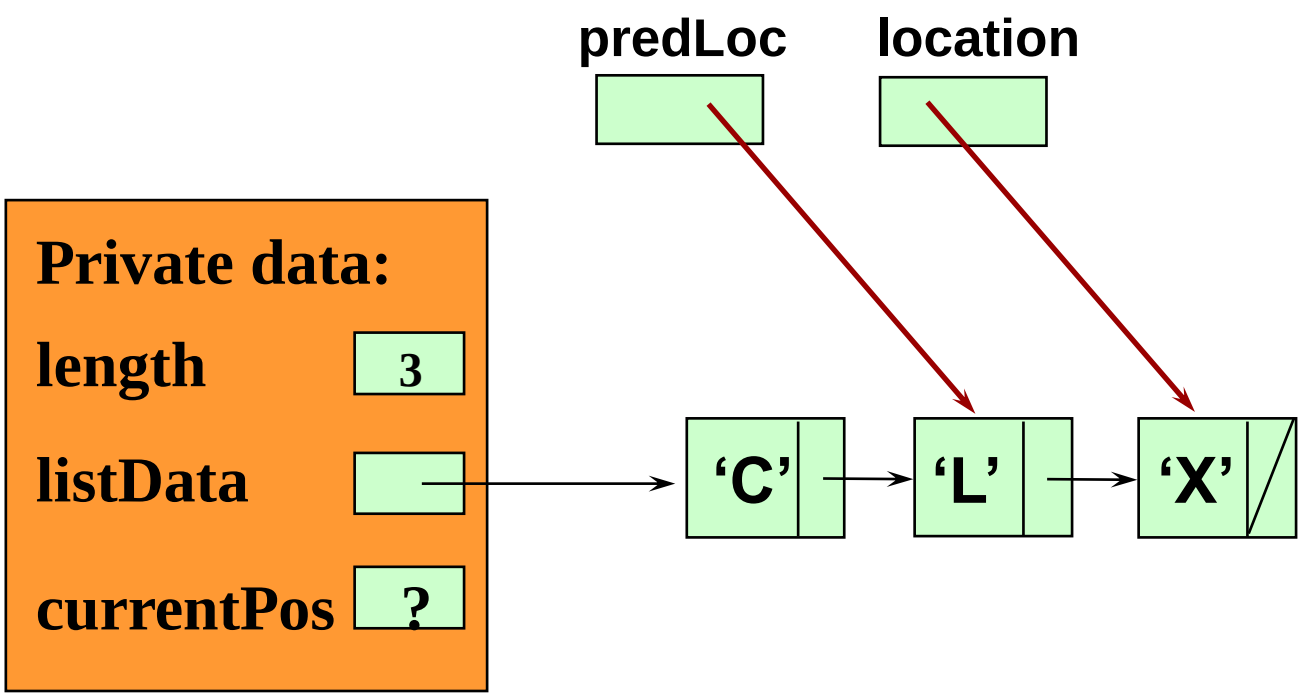


Finding proper position for ‘S’



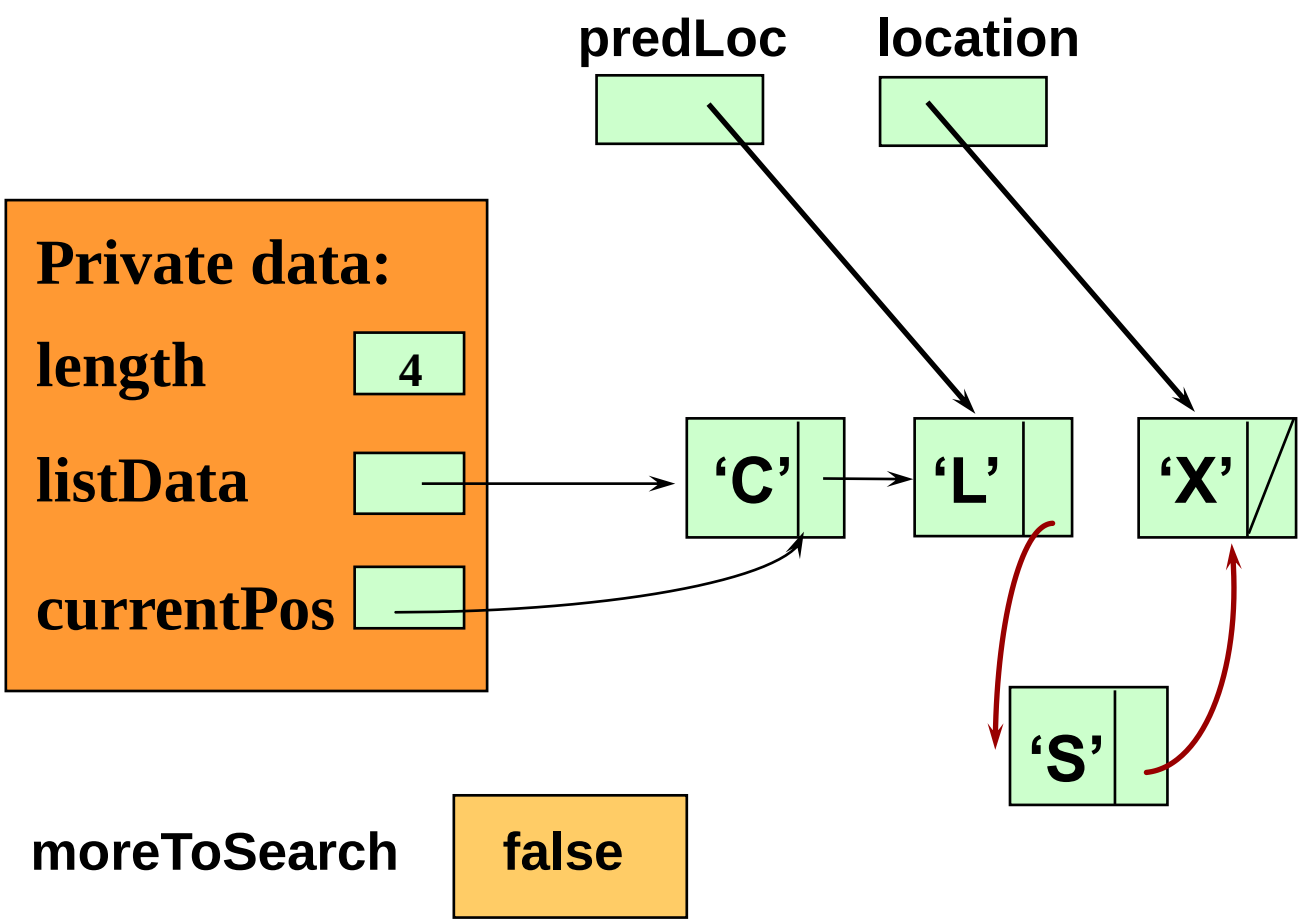
moreToSearch true

Finding Proper Position for 'S'



moreToSearch **false**

Inserting ‘S’ into Proper Position



Comparing sorted list implementations

Big-O Comparison of Sorted List Operations		
Operation	Array Implementation	Linked Implementation
Class constructor	O(1)	O(1)
Destructor	O(1)	O(N)
MakeEmpty	O(1)	O(N)
IsFull	O(1)	O(1)
LengthIs	O(1)	O(1)
ResetList	O(1)	O(1)
GetNextItem	O(1)	O(1)
RetrieveNextItem	O(N) or O(logN)	O(N)
InsertItem	O(N)	O(N)
DeleteItem	O(N)	O(N)