C++ Plus Data Structures

Nell Dale
David Teague

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.

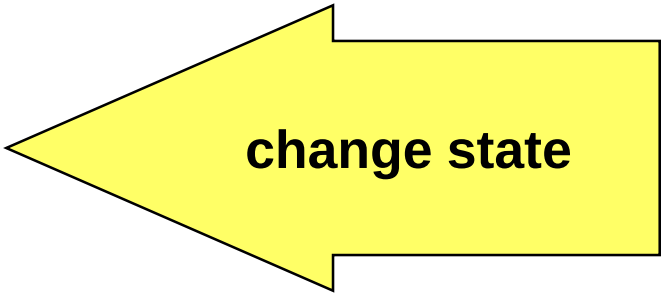
Stack ADT Operations

- **MakeEmpty -- Sets stack to an empty state.**
- **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 (ItemType& item) -- Removes the item at the top of the stack and returns it in item.**

ADT Stack Operations

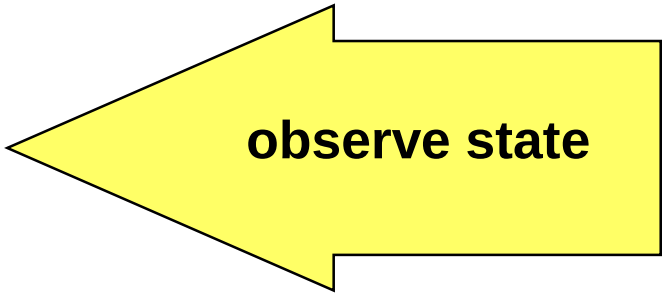
Transformers

- MakeEmpty
- Push
- Pop



Observers

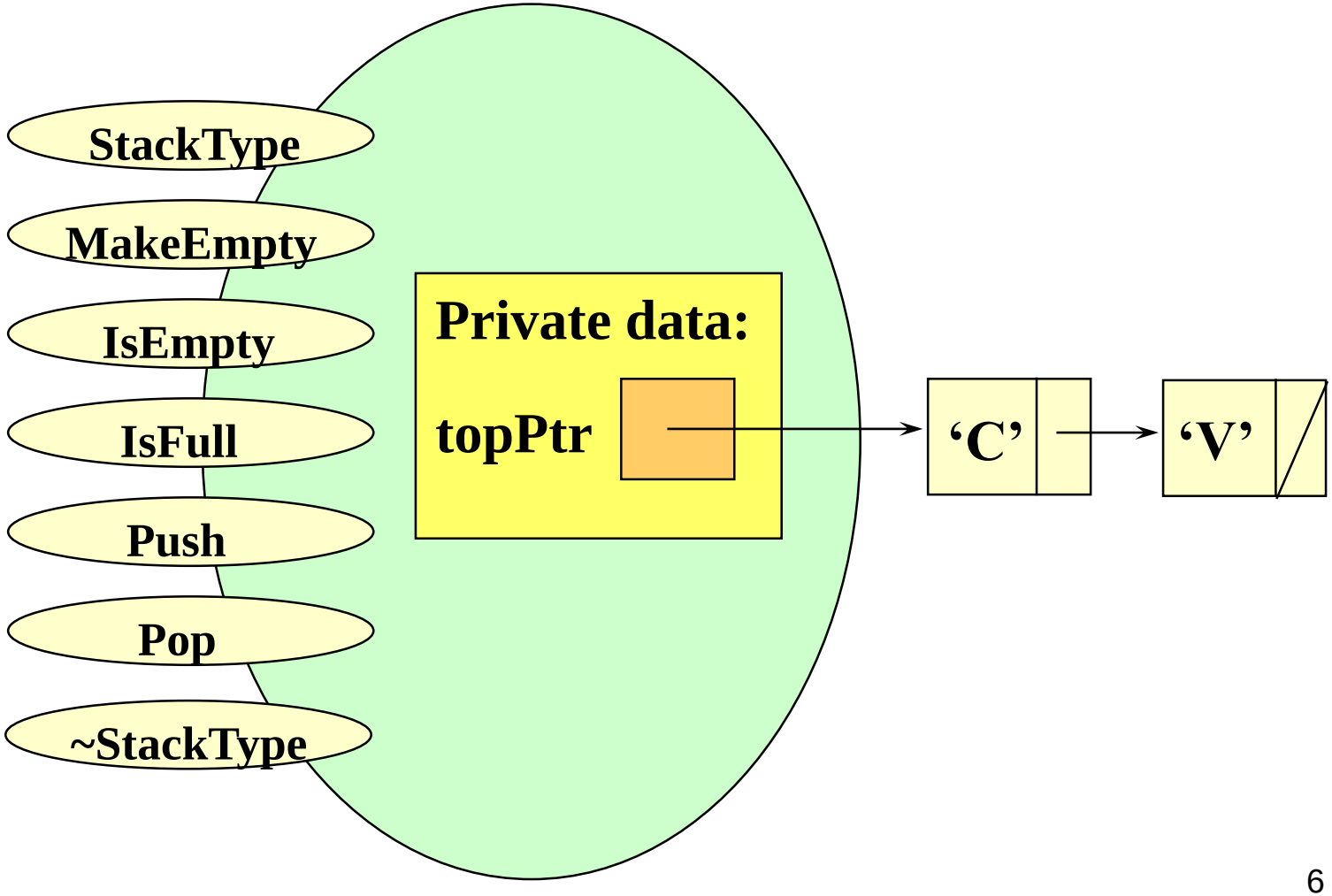
- IsEmpty
- IsFull



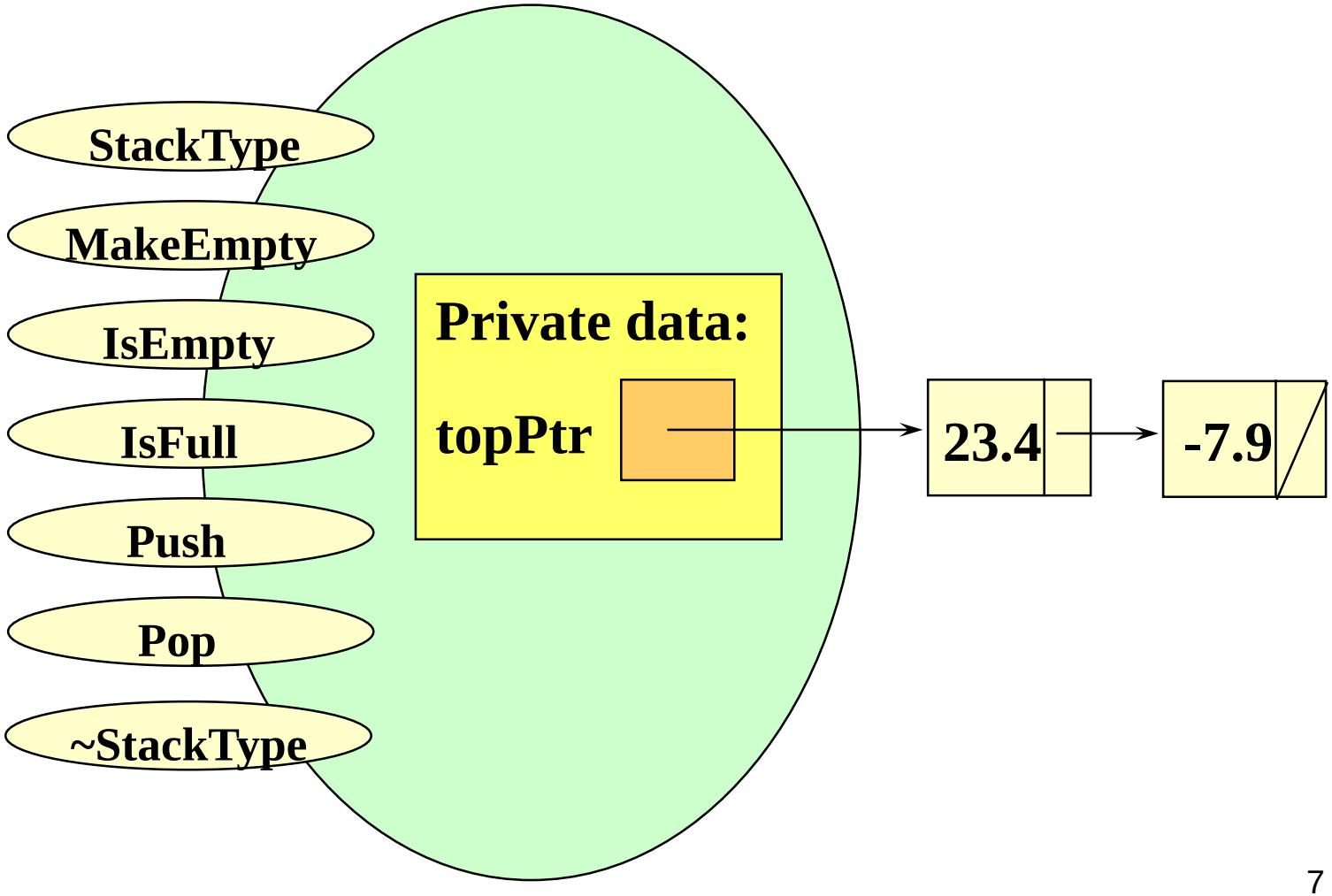
Another Stack Implementation

- 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.

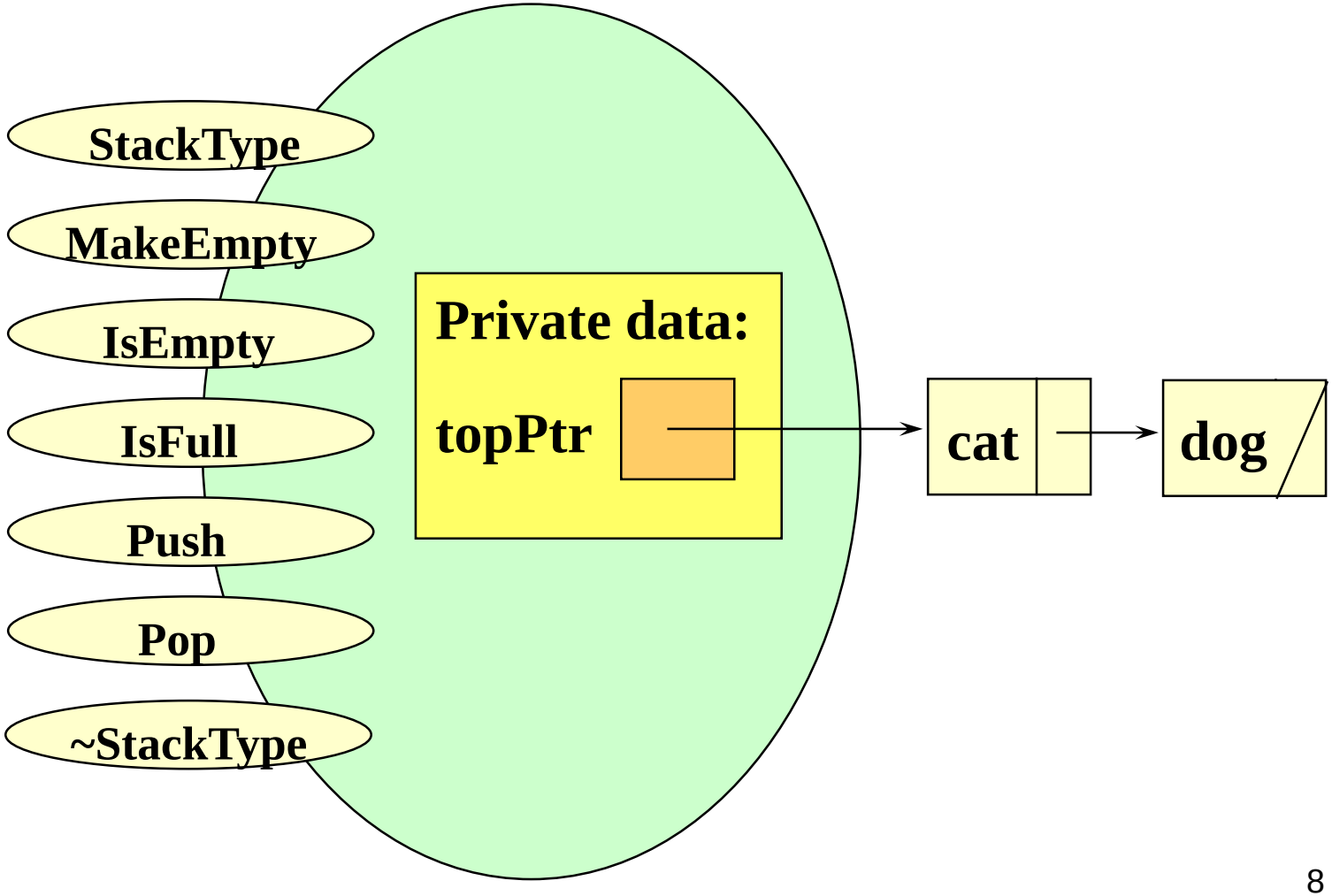
class StackType<char>



class StackType<float>



class StackType<StrType>



Tracing Client Code

letter ‘V’

```
char     letter = ‘V’;

StackType< char >  myStack;

myStack.Push(letter);

myStack.Push(‘C’);

myStack.Push(‘S’);

if ( !myStack.IsEmpty( ) )
     myStack.Pop( letter );

myStack.Push(‘K’);
```

letter ‘V’

Tracing Client Code

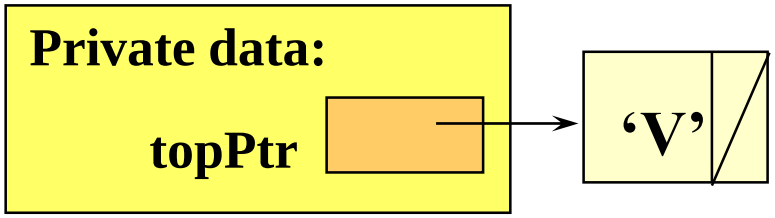
Private data:

topPtr NULL

```
char     letter = 'V';

StackType< char >  myStack;

myStack.Push(letter);

myStack.Push('C');

myStack.Push('S');

if ( !myStack.IsEmpty( ) )
     myStack.Pop( letter );

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

letter ‘V’

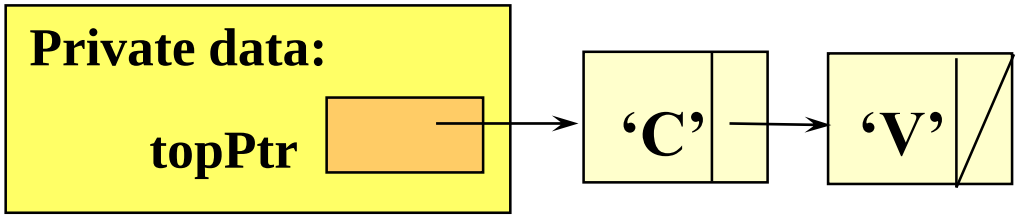
Tracing Client Code




```
char     letter = 'V';
StackType< char >  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
if ( !myStack.IsEmpty( ) )
     myStack.Pop( letter );
myStack.Push('K');
```

Tracing Client Code

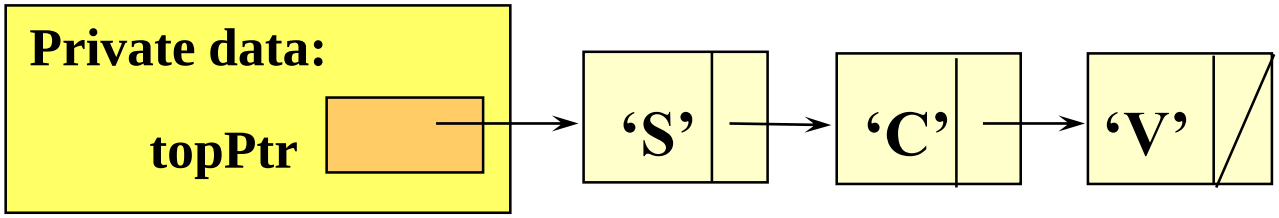
letter ‘V’




```
char     letter = 'V';
StackType< char >  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
if ( !myStack.IsEmpty( ) )
     myStack.Pop( letter );
myStack.Push('K');
```

Tracing Client Code

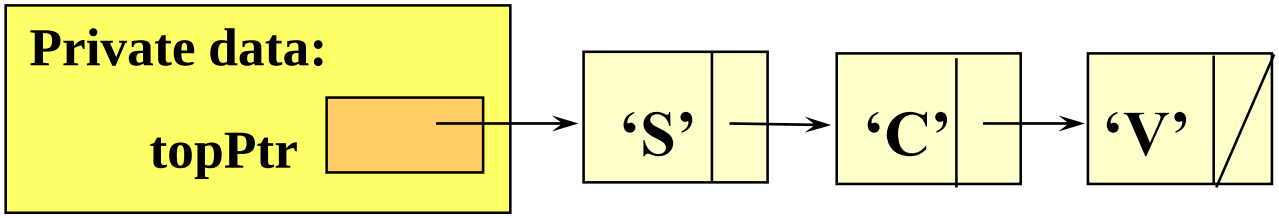
letter ‘V’




```
char     letter = 'V';
StackType< char >  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
if ( !myStack.IsEmpty( ) )
     myStack.Pop( letter );
myStack.Push('K');
```

Tracing Client Code

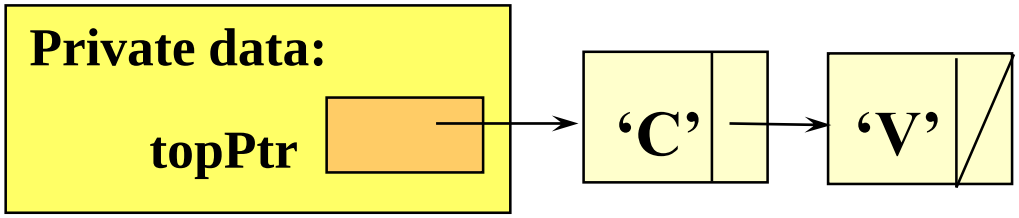
letter ‘V’




```
char     letter = 'V';
StackType< char >  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
if ( !myStack.IsEmpty( ) )
     myStack.Pop( letter );
myStack.Push('K');
```

letter ‘S’

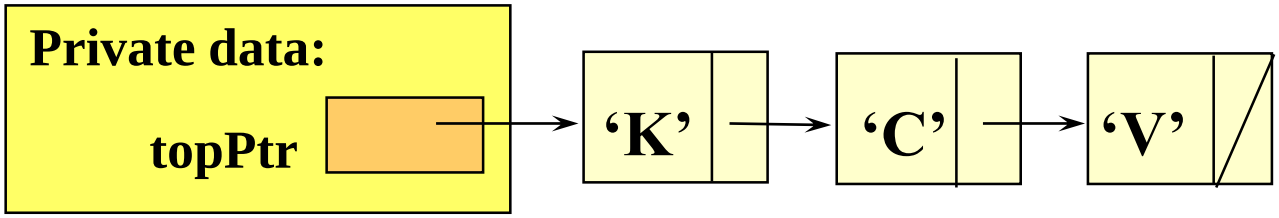
Tracing Client Code




```
char     letter = 'V';
StackType< char >  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
if ( !myStack.IsEmpty( ) )
    myStack.Pop( letter );
myStack.Push('K');
```

letter ‘S’

Tracing Client Code

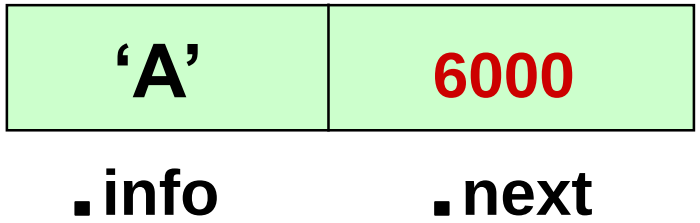



```
char     letter = 'V';
StackType< char >  myStack;
myStack.Push(letter);
myStack.Push('C');
myStack.Push('S');
if ( !myStack.IsEmpty( ) )
     myStack.Pop( letter );
```

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

Dynamically Linked Implementation of Stack

```
// DYNAMICALLY LINKED IMPLEMENTATION OF STACK
#include "ItemType.h"        // for ItemType
template<class ItemType>
struct NodeType
{
   ItemType  info;
   NodeType<ItemType>* next;
}
```



```
// DYNAMICALLY LINKED IMPLEMENTATION OF STACK  continued

template<class ItemType>
class StackType  {
public:
   StackType( );                        // constructor
        // Default constructor.
        // POST:  Stack is created and empty.
   void MakeEmpty( );
        // PRE:   None.
        // POST:  Stack is empty.
   bool IsEmpty( ) const;
        // PRE:   Stack has been initialized.
        // POST:  Function value = (stack is empty)
   bool IsFull( ) const;
        // PRE:   Stack has been initialized.
        // POST:  Function value = (stack is full)
```

```
// DYNAMICALLY LINKED IMPLEMENTATION OF STACK  continued

void Push( ItemType item );
       // PRE:  Stack has been initialized.
       //       Stack is not full.
       // POST: newItem is at the top of the stack.
void Pop( ItemType&  item );
       // PRE:  Stack has been initialized.
       //       Stack is not empty.
       // POST: Top element has been removed from stack.
       //       item is a copy of removed element.
~StackType( );                          // destructor
       // PRE:  Stack has been initialized.
       // POST: Memory allocated for nodes has been
       //       deallocated.
private:

   NodeType<ItemType>*  topPtr ;
};
```

```
// DYNAMICALLY LINKED IMPLEMENTATION OF STACK continued
// member function definitions for class StackType

template<class ItemType>
StackType<ItemType>::StackType( )          // constructor
{
   topPtr = NULL;
}

template<class ItemType>
void StackType<ItemType>::IsEmpty( )  const
       // Returns true if there are no elements
       // on the stack; false otherwise
{
   return  ( topPtr == NULL );
}
```

Using operator `new`

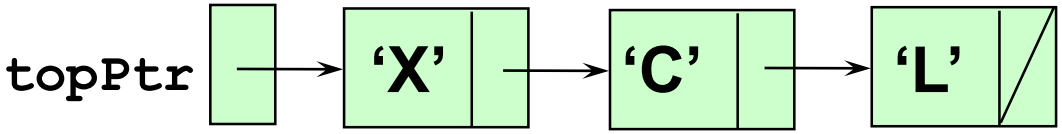
If memory is available in an area called the free store (or heap), operator new allocates the requested object, and returns a pointer to the memory allocated.

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

Adding newItem to the stack

newItem 'B'

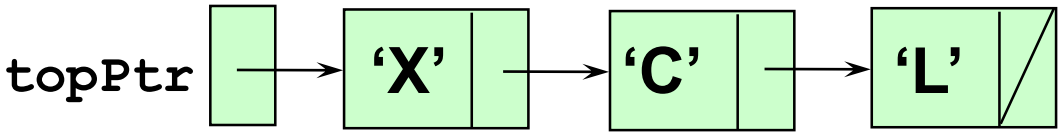
```
newItem = 'B';
NodeType<char>*  location;
location = new  NodeType<char>;
location->info = newItem;
location->next = topPtr;
topPtr = location;
```



newItem ‘B’

Adding newItem to the stack

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

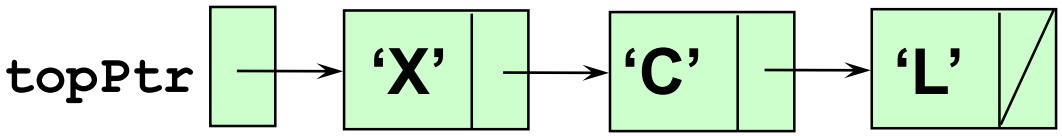


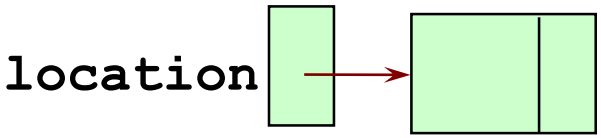
location

newItem ‘B’

Adding newItem to the stack

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

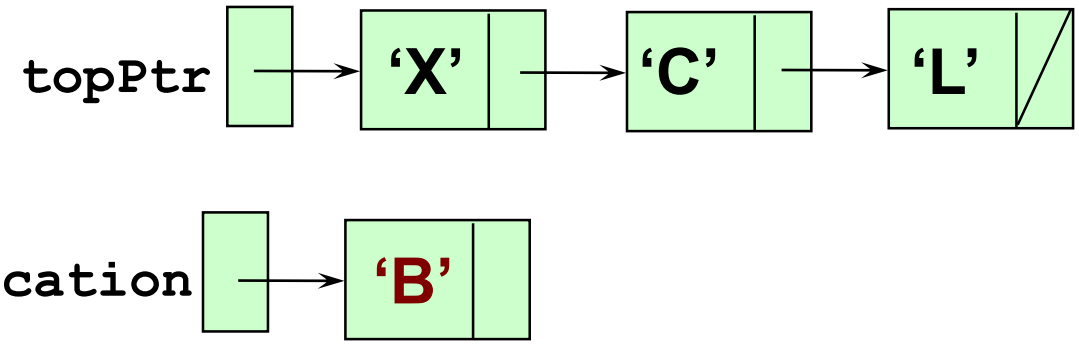




newItem ‘B’

Adding newItem to the stack

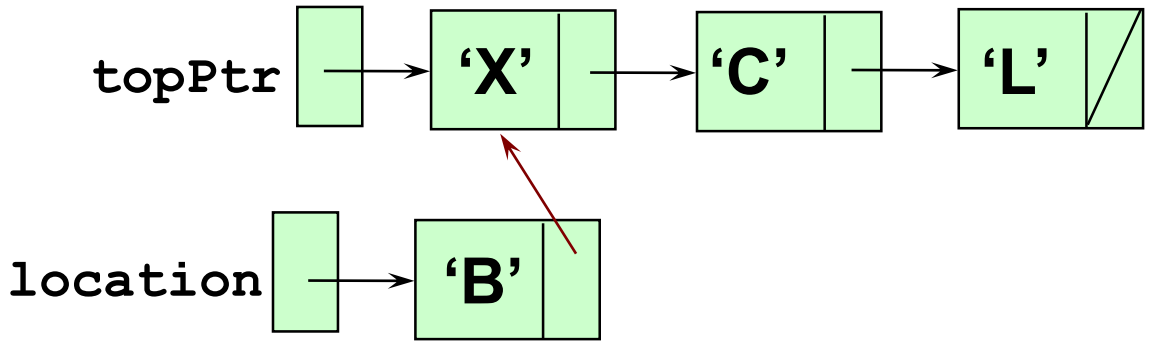
```
newItem = 'B';
NodeType<char>*  location;
location = new  NodeType<char>;
location->info = newItem;
location->next = topPtr;
topPtr = location;
```



newItem ‘B’

Adding newItem to the stack

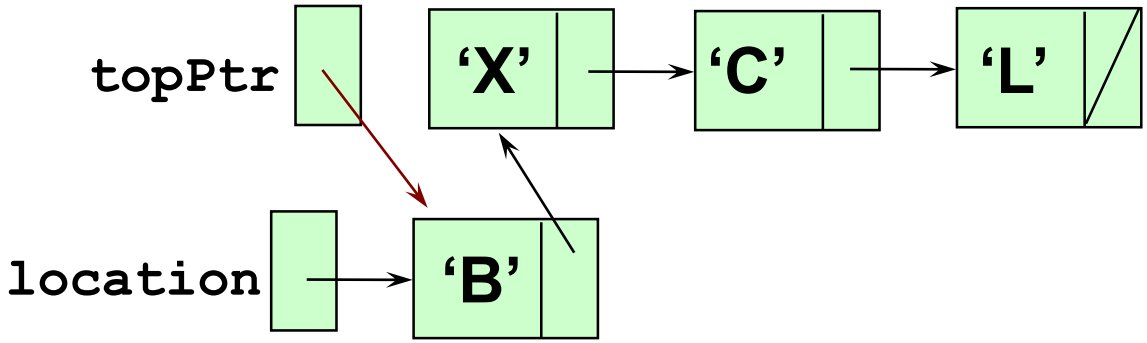
```
newItem = 'B';
NodeType<char>*  location;
location = new  NodeType<char>;
location->info = newItem;
location->next = topPtr;
topPtr = location;
```



newItem 'B'

Adding newItem to the stack

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

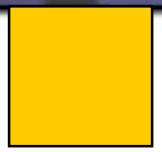


Implementing Push

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

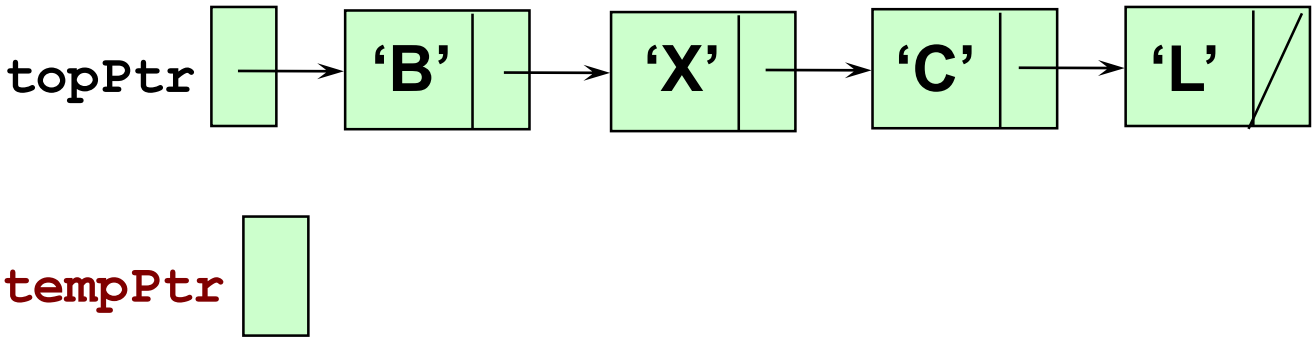
Using operator `delete`

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

item 

Deleting item from the stack

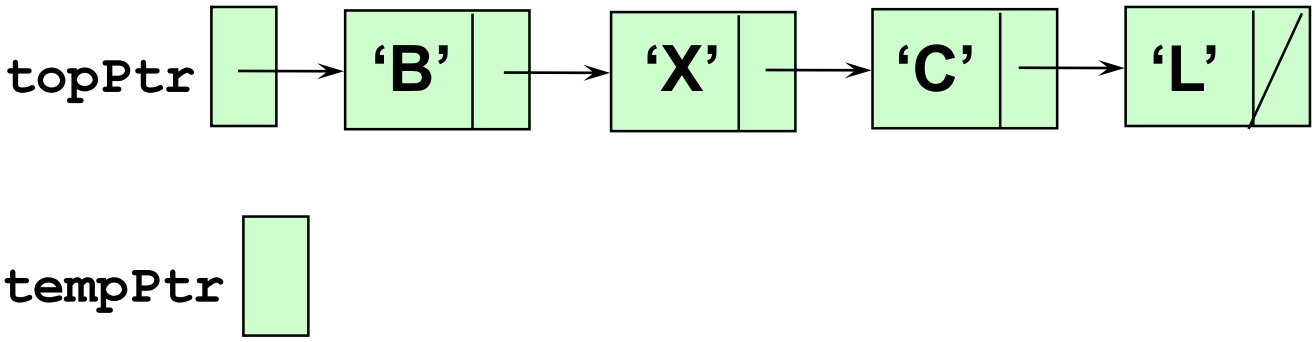
```
NodeType<ItemType>*   tempPtr;

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



item 'B'

Deleting item from the stack

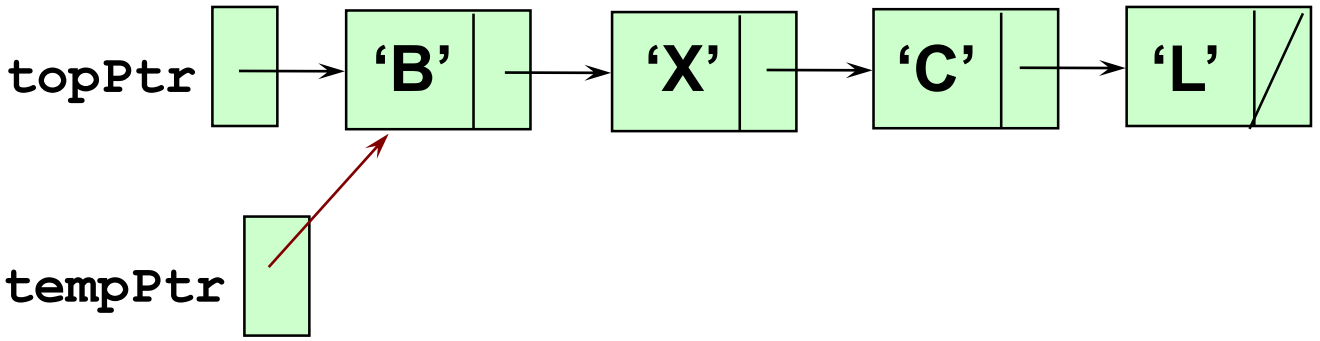
```
NodeType<ItemType>*  tempPtr;

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



item ‘B’

Deleting item from the stack

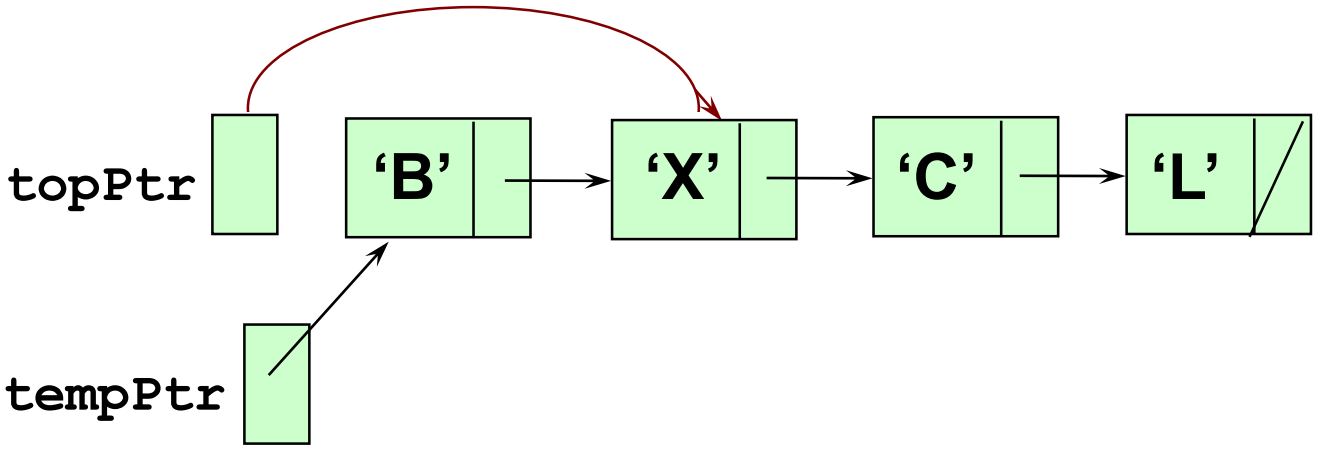
```
NodeType<ItemType>*   tempPtr;

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



item 'B'

Deleting item from the stack

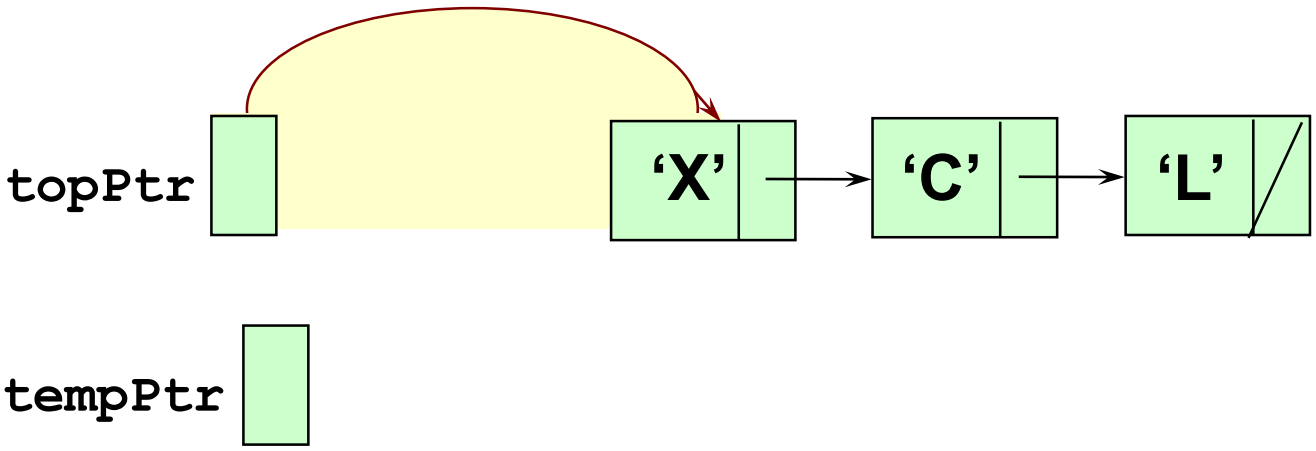
```
NodeType<ItemType>*   tempPtr;

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



item 'B'

Deleting item from the stack

```
NodeType<ItemType>*  tempPtr;

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



Implementing Pop

```
template<class ItemType>
void StackType<ItemType>::Pop ( ItemType& item )
   // Removes element at the top of the stack and
   // returns it in item.
{ if (IsEmpty())
    throw PopOnEmptyStack();
   NodeType<ItemType>*  tempPtr;
   item = topPtr->info;
   tempPtr = topPtr;
   topPtr = topPtr->next;
   delete  tempPtr;
}
```

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

```
// Alternate form that works with older compilers
template<class ItemType>
bool StackType<ItemType>::IsFull( )  const
// Returns true if there is no room for another NodeType
// node on the free store; false otherwise.
{
   NodeType<ItemType>* location;
   location = new NodeType<ItemType>;
   if  ( location == NULL )
      return true;
   else
   {
     delete location;
     return false;
   }
}
```

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.

```
template<class ItemType>
void StackType<ItemType>::MakeEmpty( )
   // Post:  Stack is empty; all elements deallocated.
{
   NodeType<ItemType>*  tempPtr;;
   while  ( topPtr != NULL )
   {
        tempPtr = topPtr;
        topPtr = topPtr->next;
        delete  tempPtr;
   }
}

template<class ItemType>
StackType<ItemType>::~StackType( )        // destructor
{
   MakeEmpty( );
}
```

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.

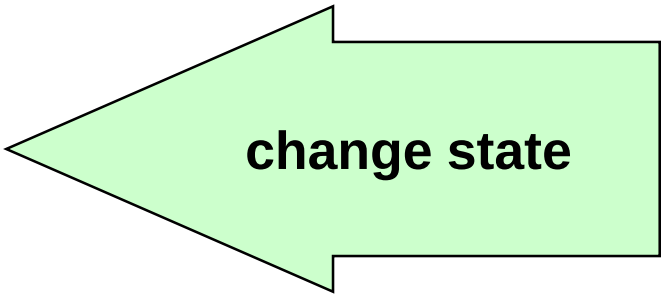
Queue ADT Operations

- **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

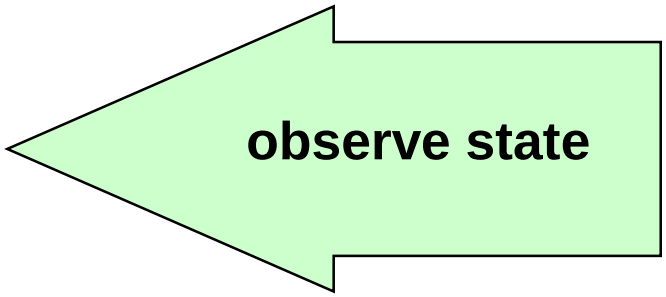
Transformers

- MakeEmpty
- Enqueue
- Dequeue

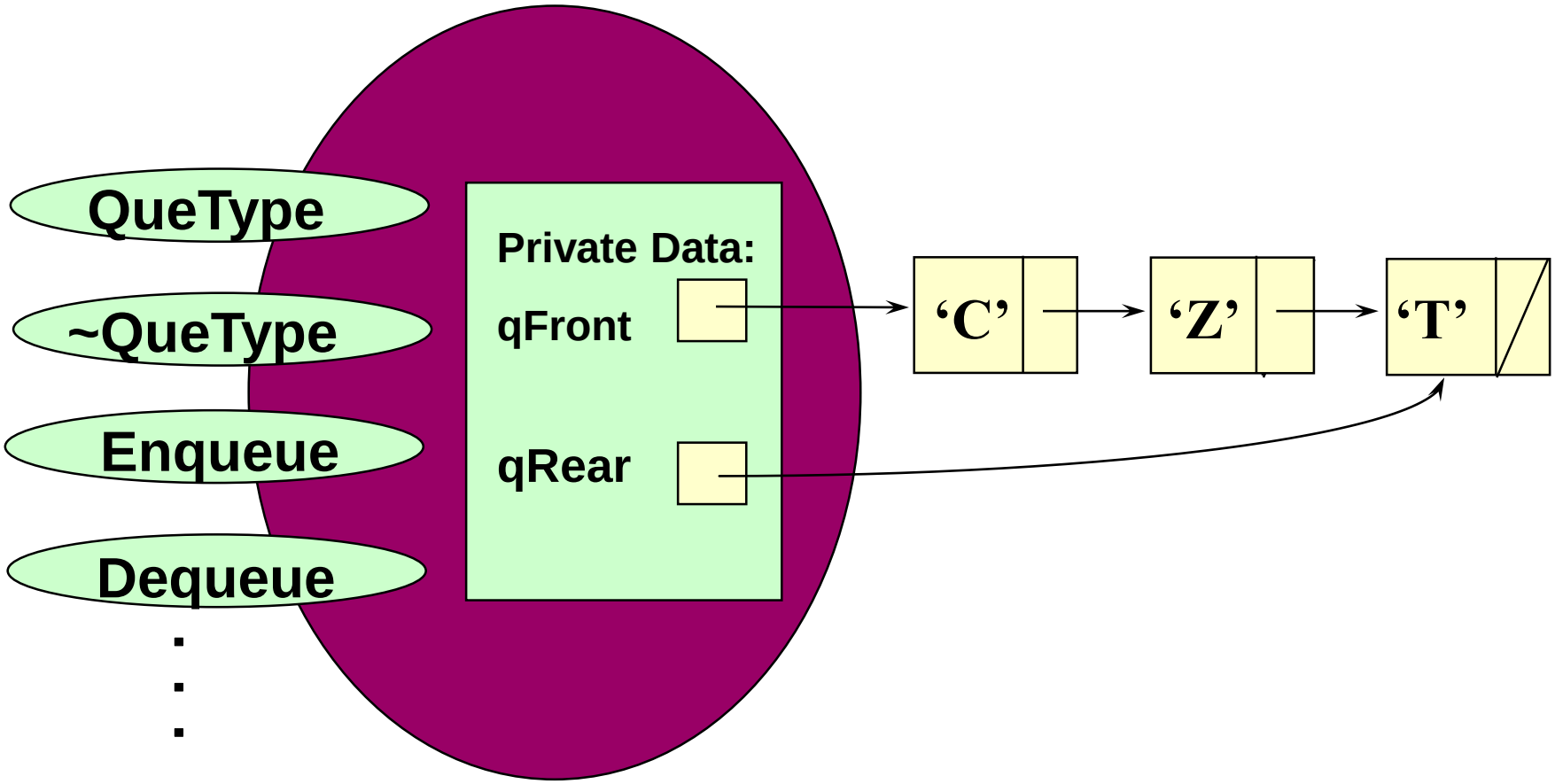


Observers

- IsEmpty
- IsFull



class QueType<char>

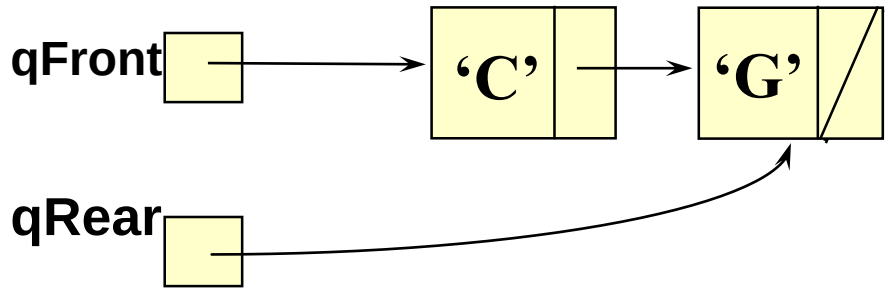


```
// DYNAMICALLY LINKED IMPLEMENTATION OF QUEUE

#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;
};
```

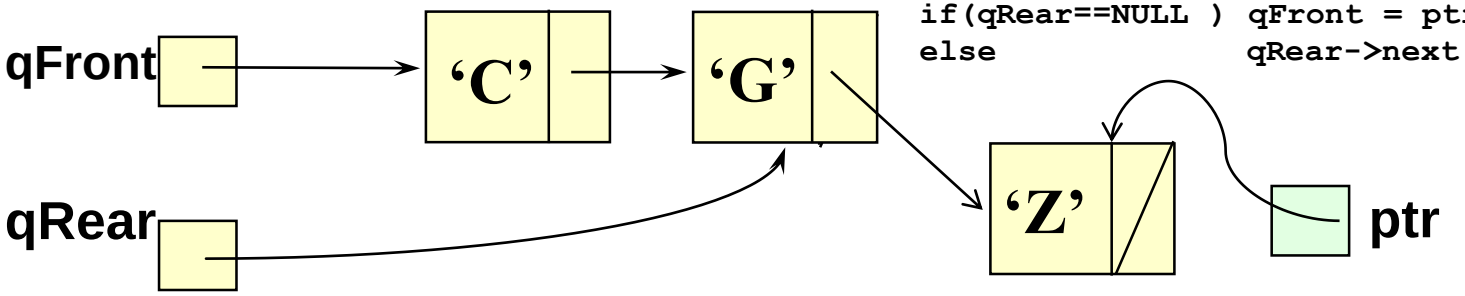
```
// DYNAMICALLY LINKED IMPLEMENTATION OF QUEUE   continued
// member function definitions for class QueType

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

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

Add 'Z' to Queue

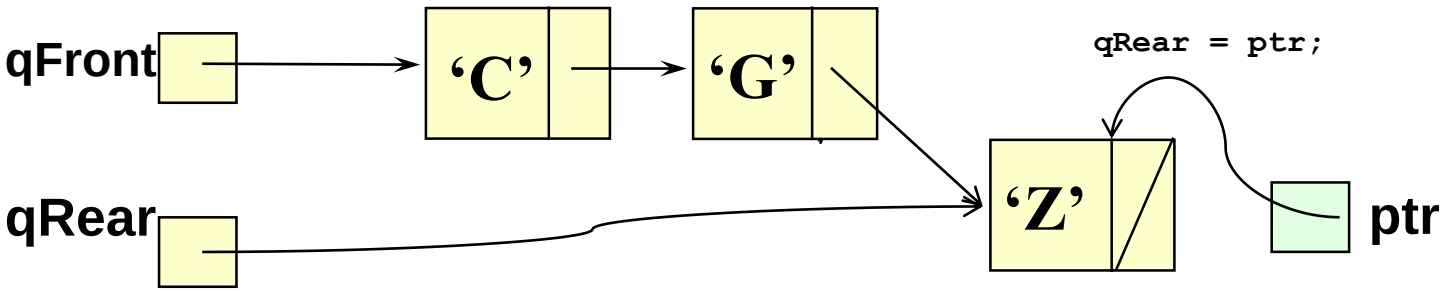
```
ptr = new  NodeType<ItemType>;
ptr->info = newItem;
ptr->next = NULL;
```




```
if(qRear==NULL ) qFront = ptr;
else             qRear->next = ptr;
```

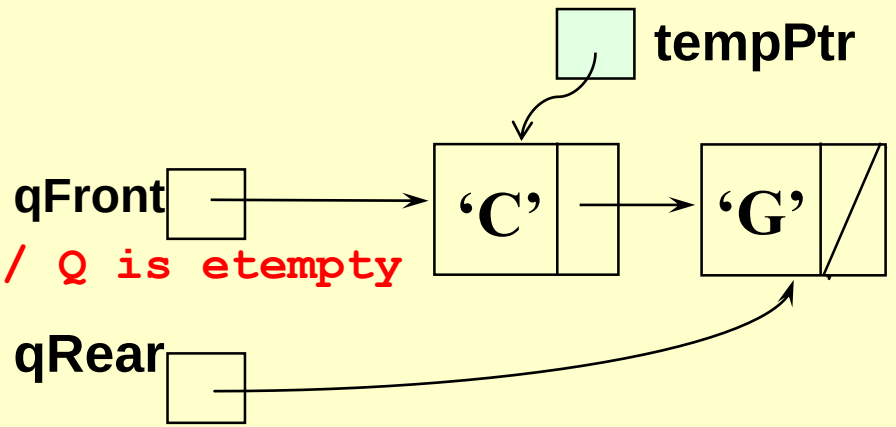



```
qRear = ptr;
```



```
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 ) // Q is empty. one element
        qFront = ptr;
   else // Q is notempty. Add to the end of Q
        qRear->next = ptr;
   qRear = ptr; // update qReal
}
```

```
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 ) // Q is etempty
      qRear = NULL;
   delete  tempPtr;
}
```



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

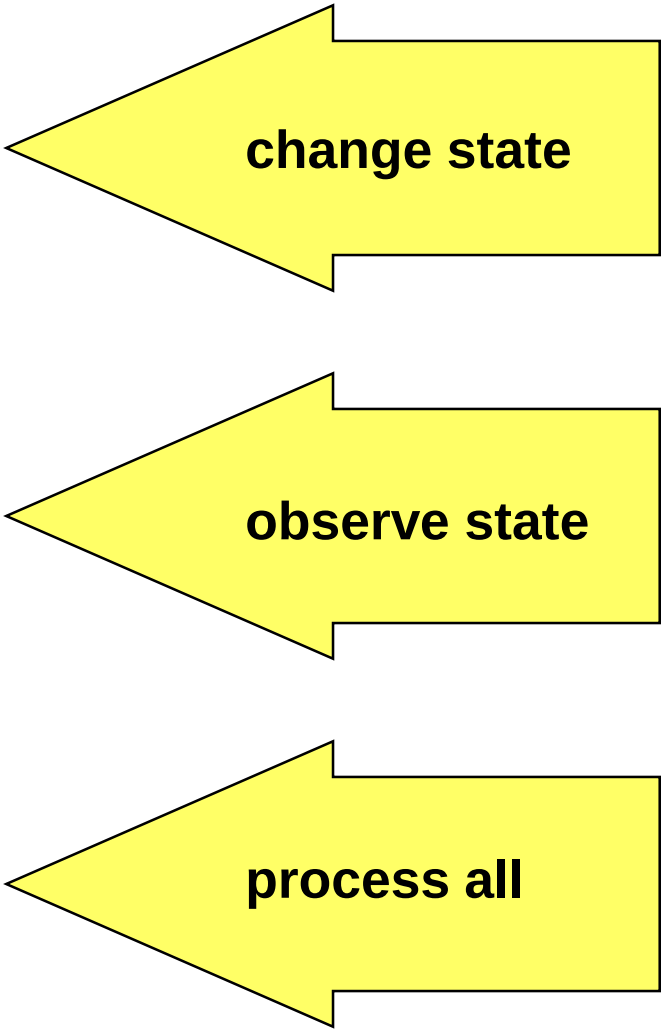
- MakeEmpty
- InsertItem
- DeleteItem

Observers

- IsFull
- LengthIs
- RetrieveItem

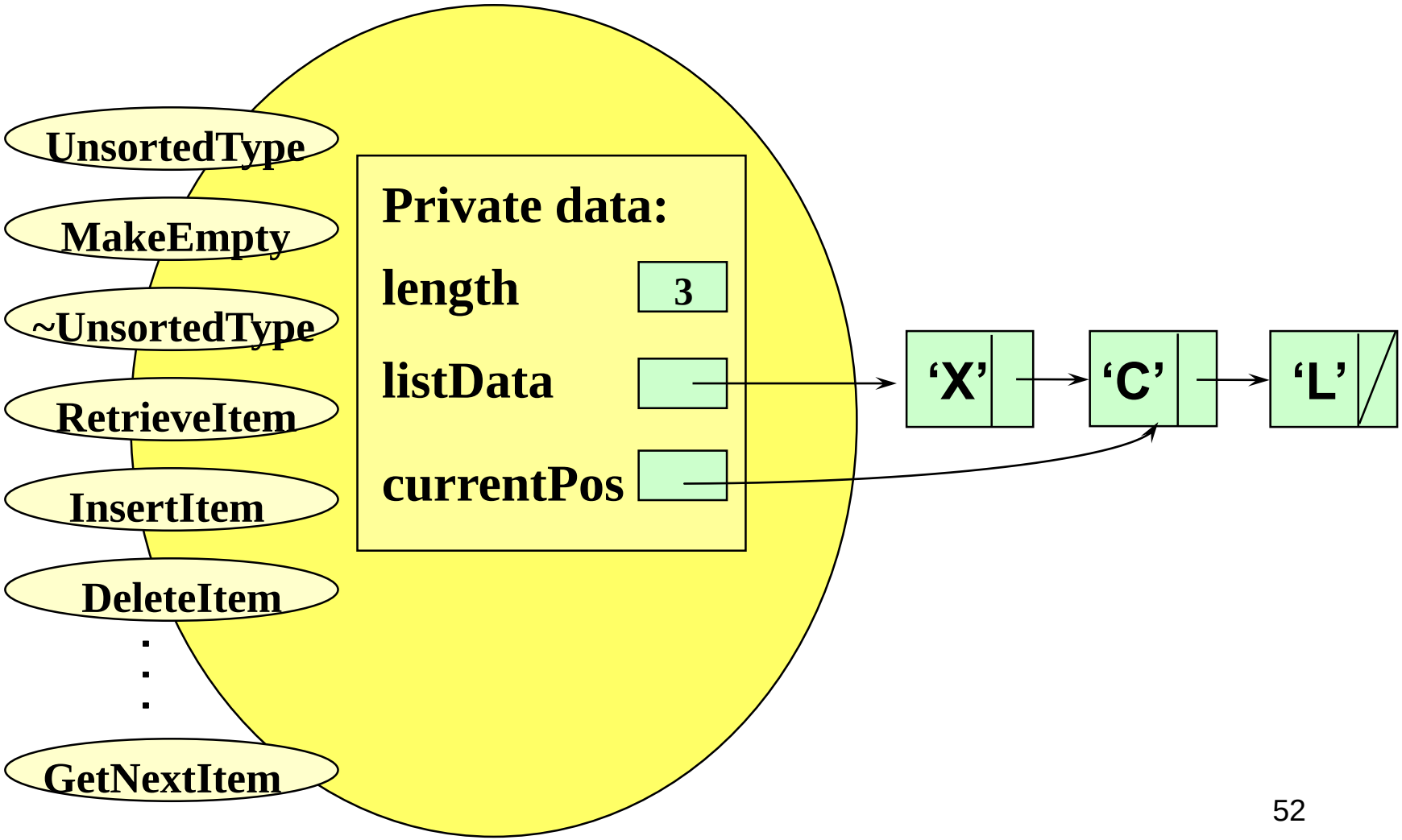
Iterators

- 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>

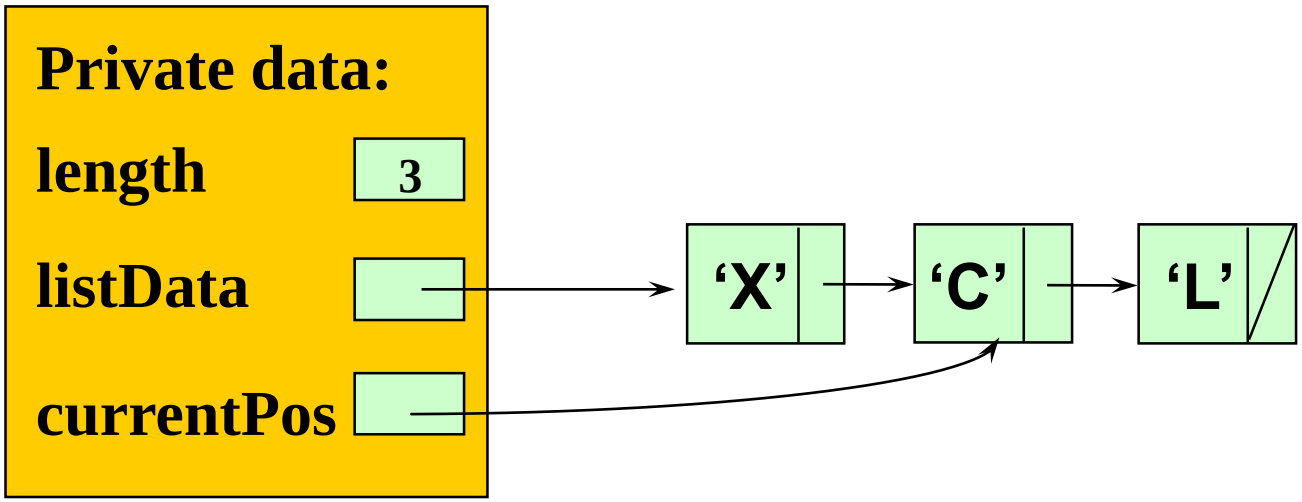


```
// LINKED LIST IMPLEMENTATION  ( unsorted.cpp )
#include "itemtype.h"

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>::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>  curNode;
    found = false ;
    ResetList();
    while ( !found && GetCurrentNode(curNode) )
    {       if ( item  == curNode )  // need operator overloading
            {    found = true ;
                 item  = curNode;
            }
    }
}
int  UnsortedType<ItemType>::GetCurrentNode(ItemType&  item)
{
    if (currentPos != NULL)
    { item=currentPos->info; currentPos=currentPos->next; return 1;}
    else return 0;
}
```

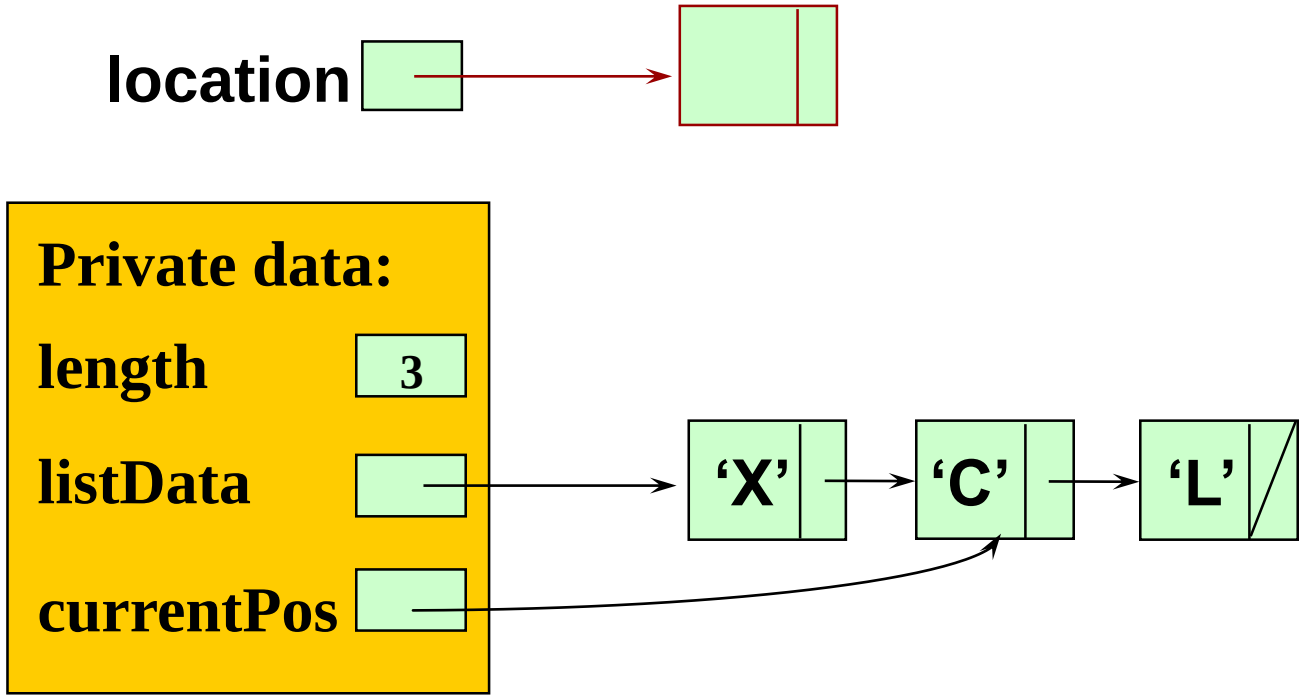
Inserting 'B' into an Unsorted List

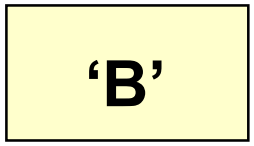


‘B’

item

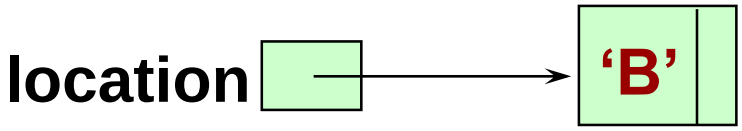
location = new NodeType<ItemType>;

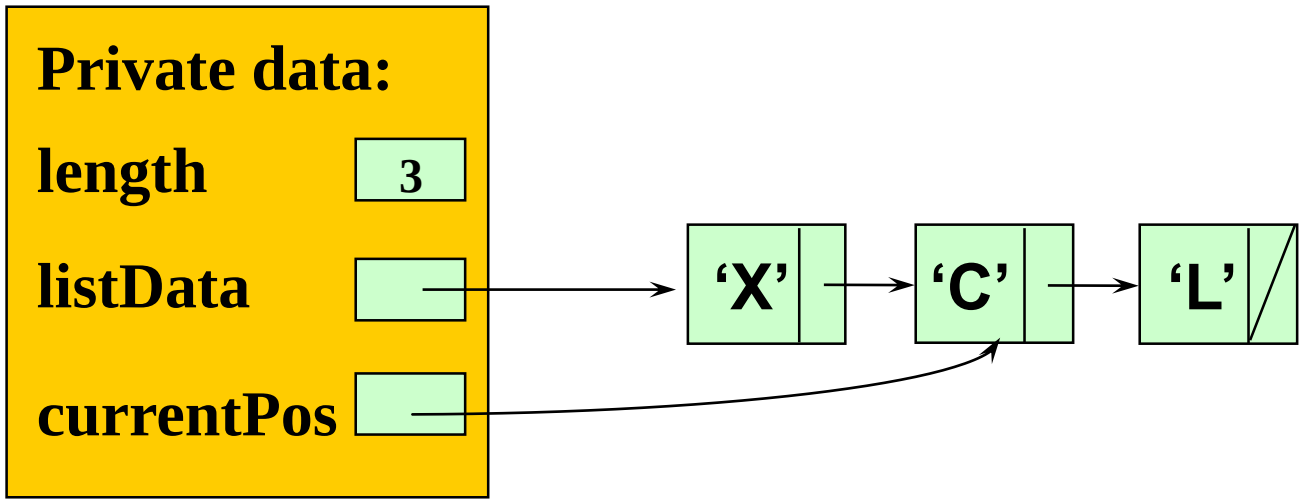




item

location->info = item ;

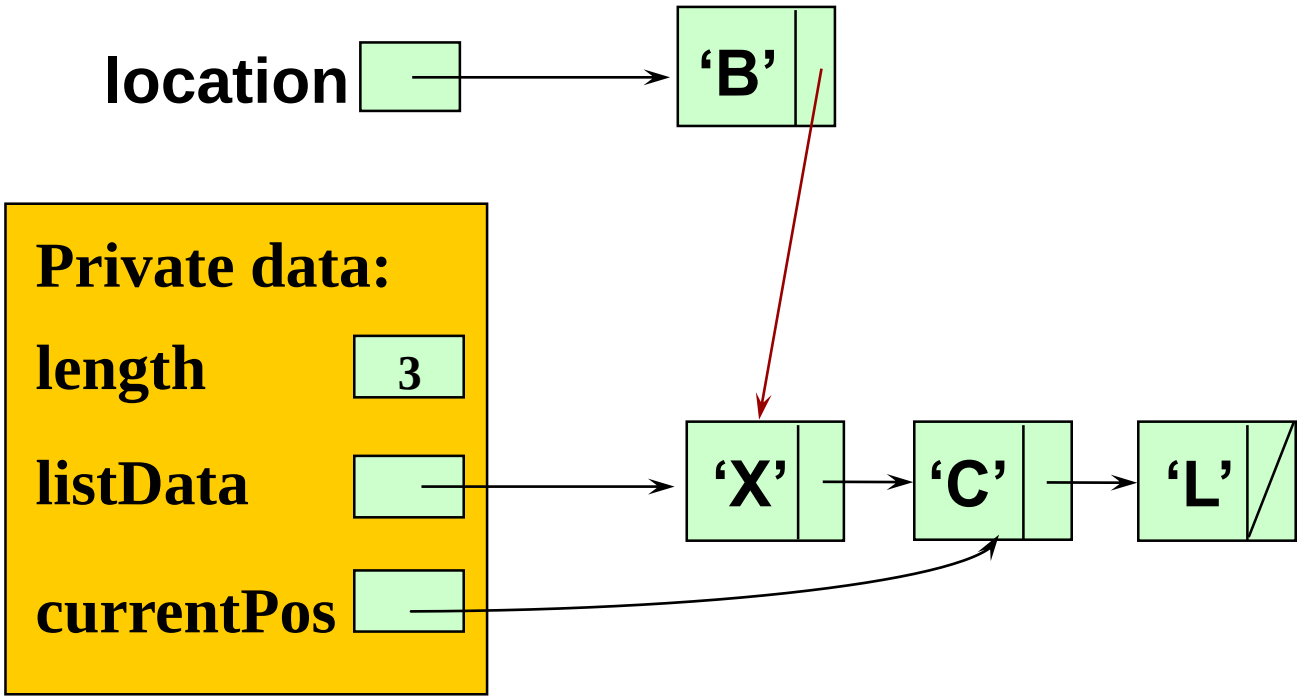




‘B’

item

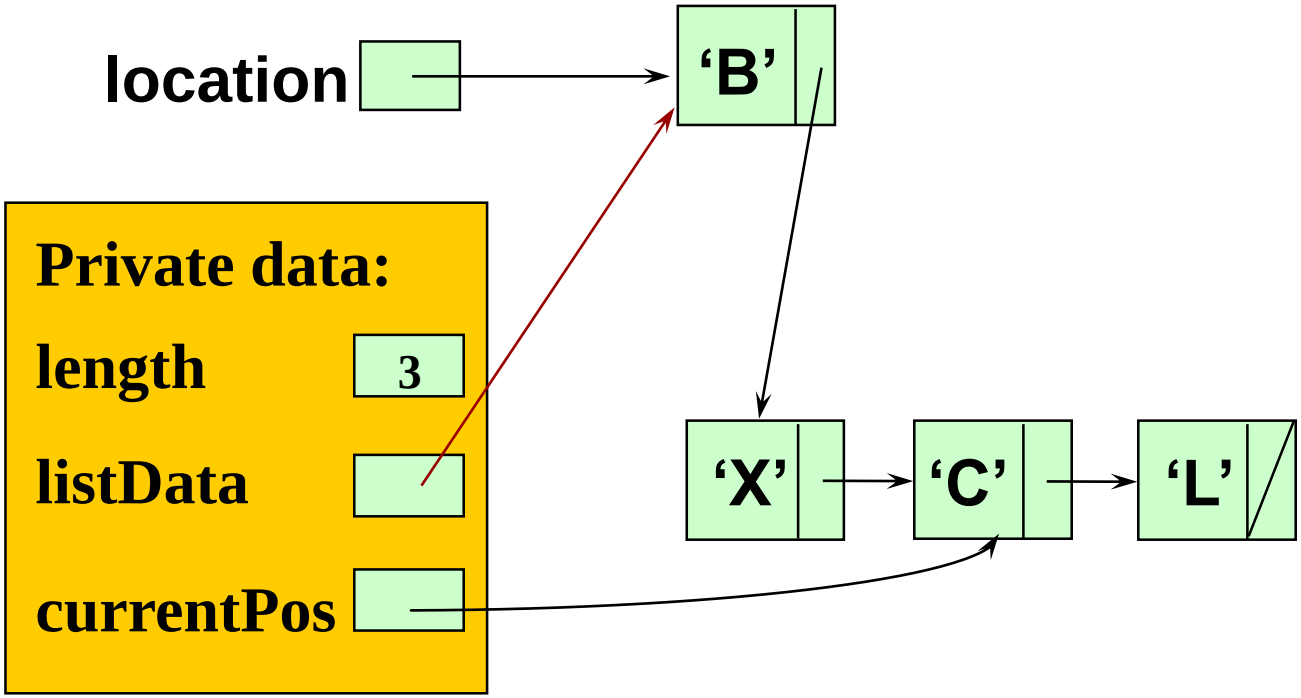
location->next = listData ;



‘B’

item

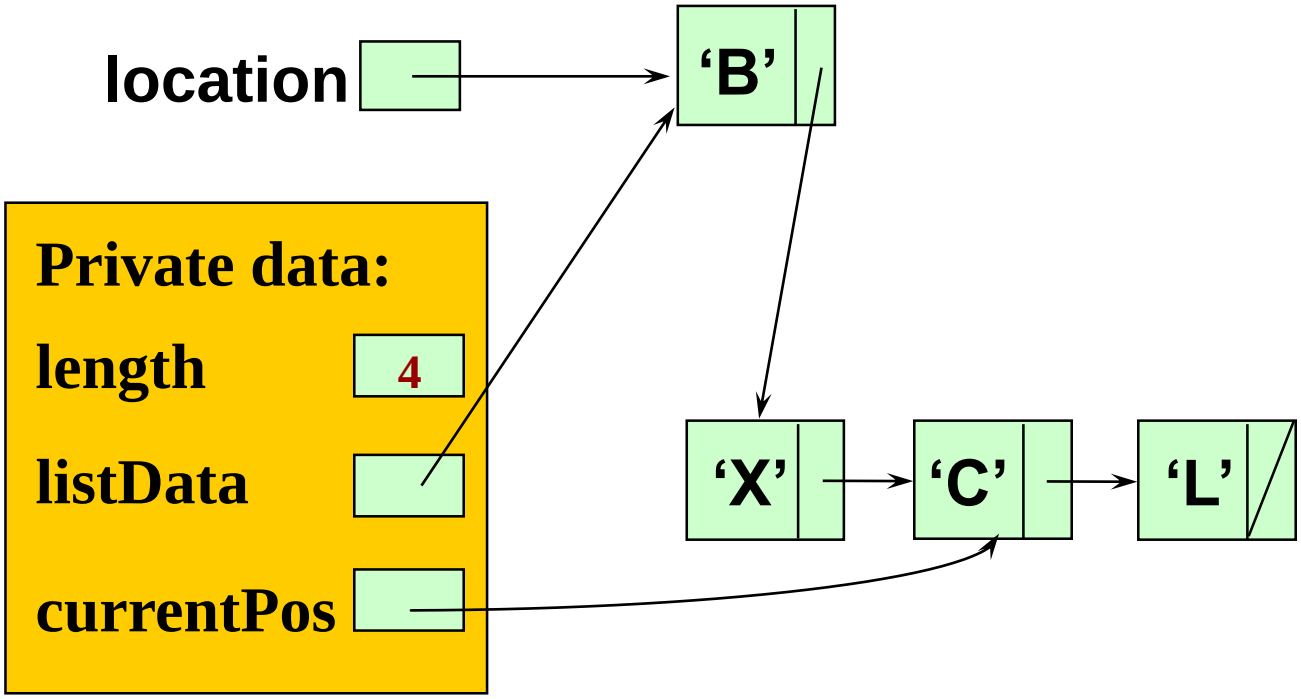
listData = location ;



‘B’

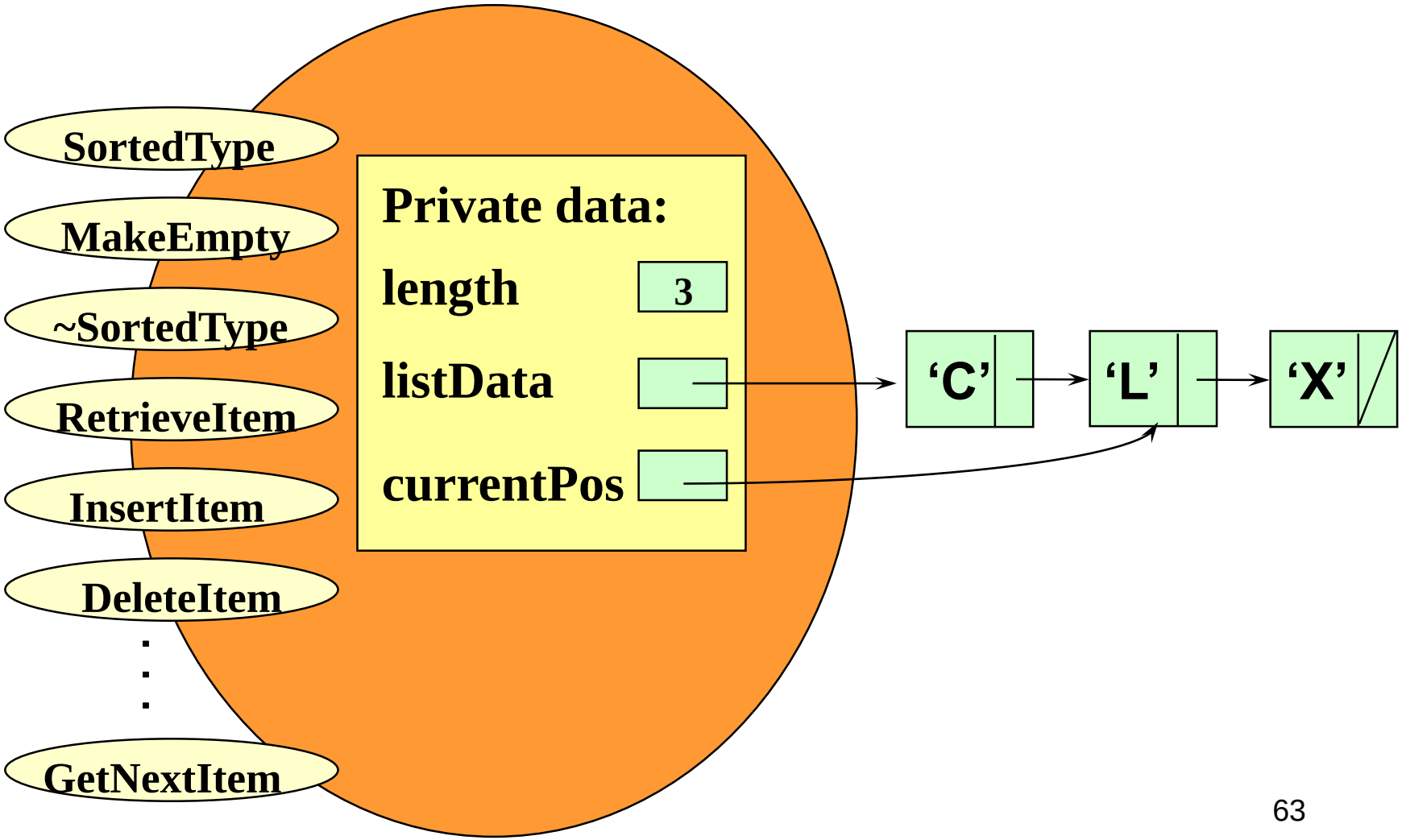
item

length++ ;



```
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++ ;
}
```

class SortedType<char>



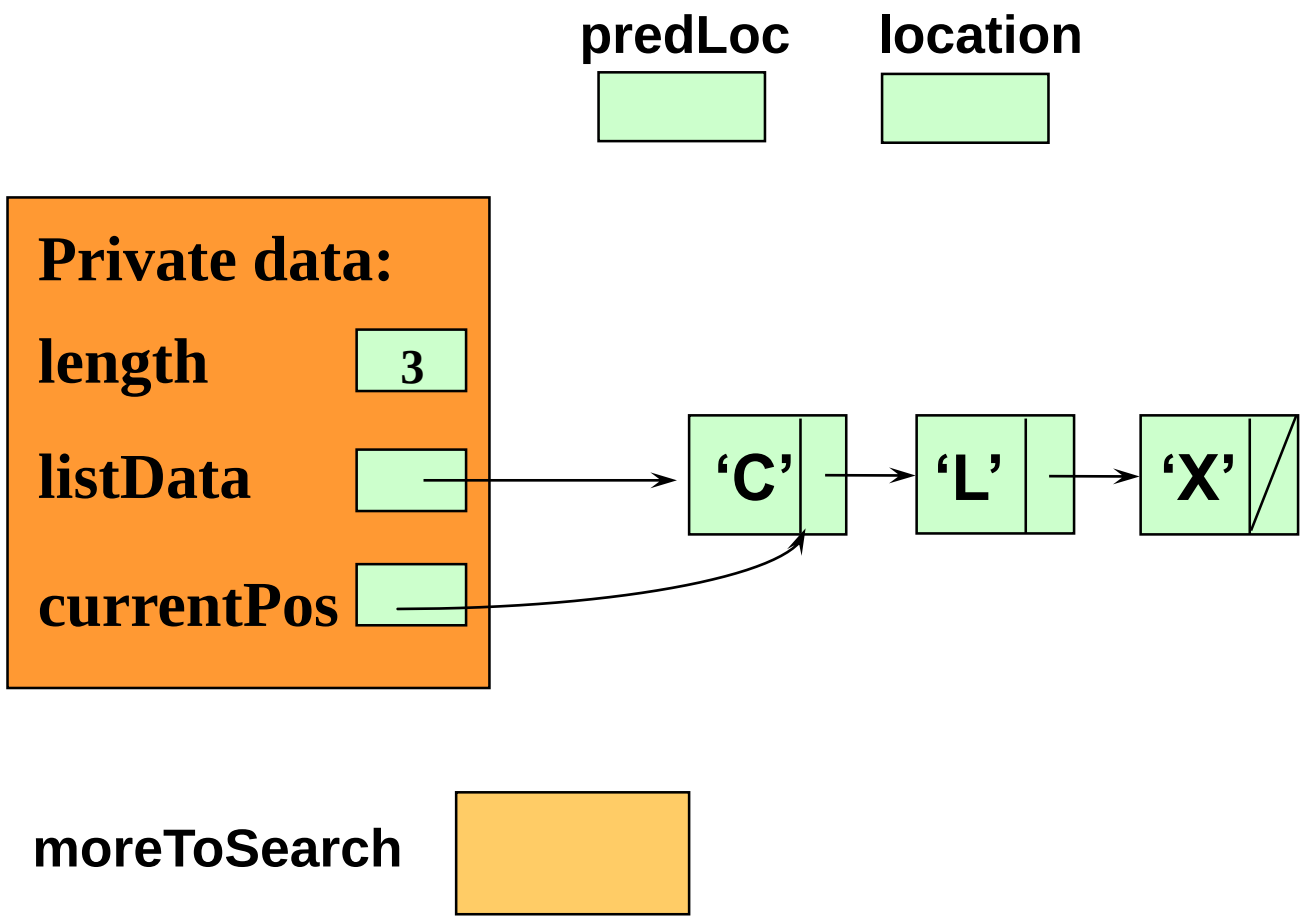
InsertItem algorithm for Sorted Linked List

- 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.

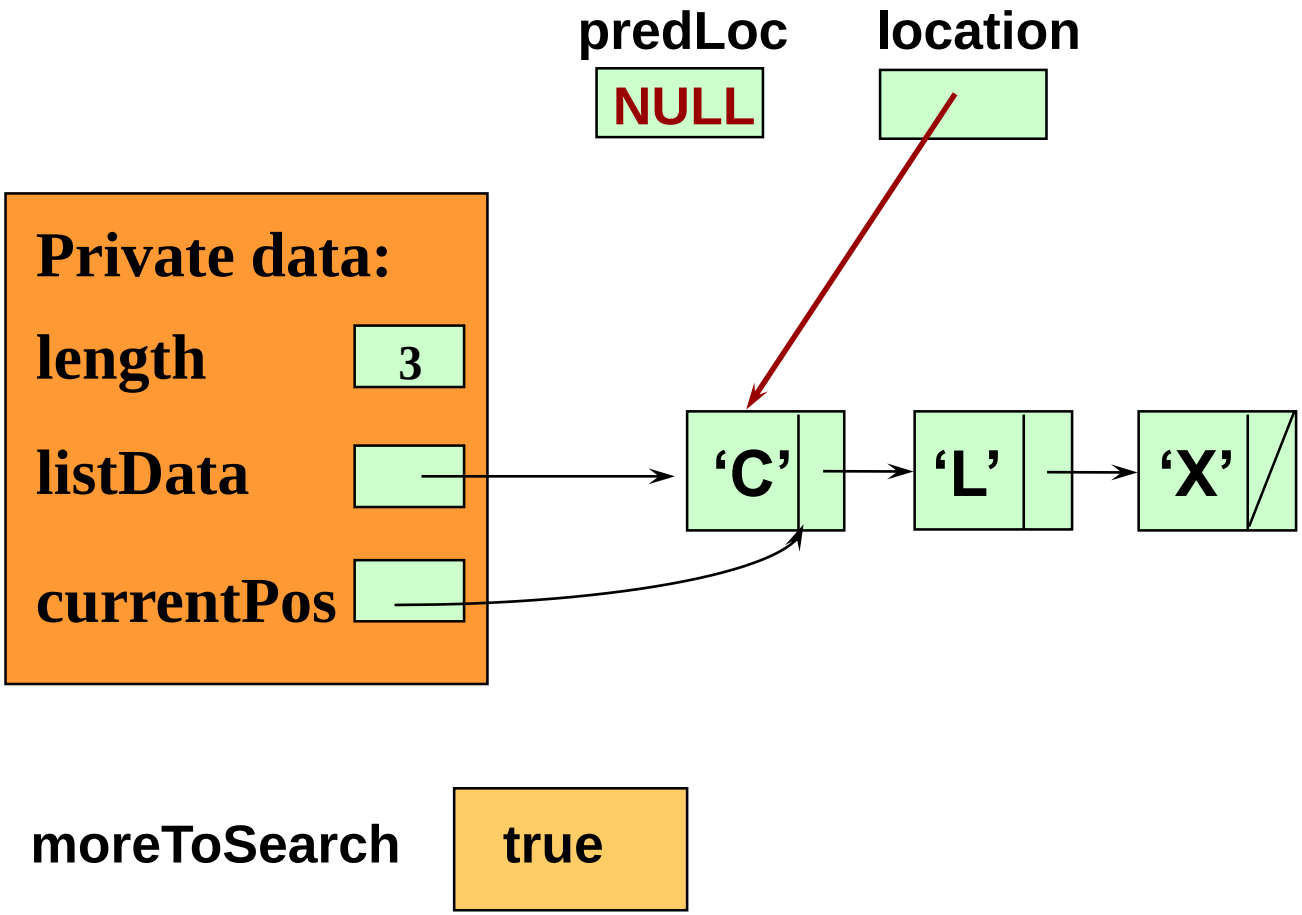
Implementing SortedType member function InsertItem

```
// 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.
{
                    .
                    .
                    .

}
```

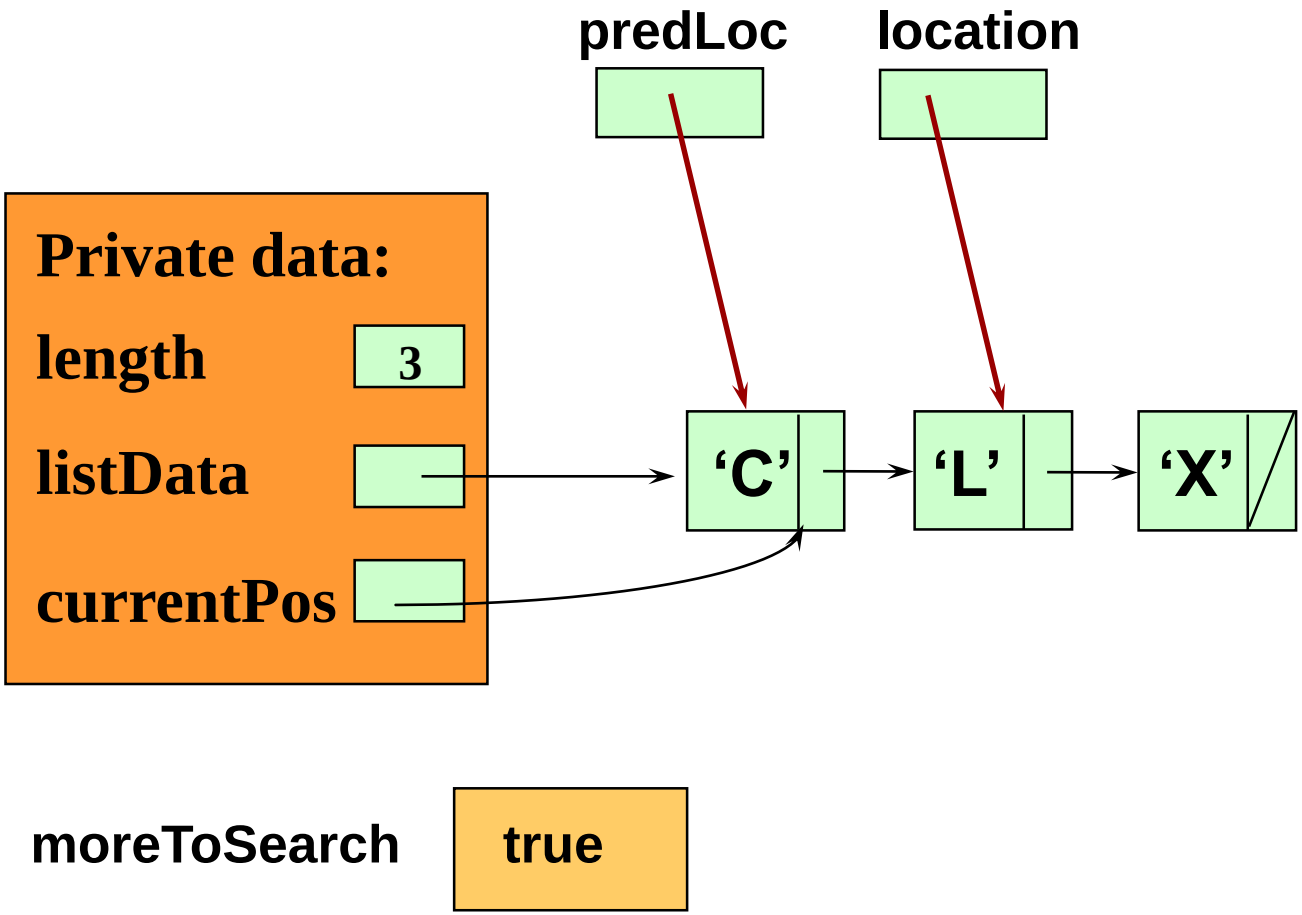
Inserting 'S' into a Sorted List



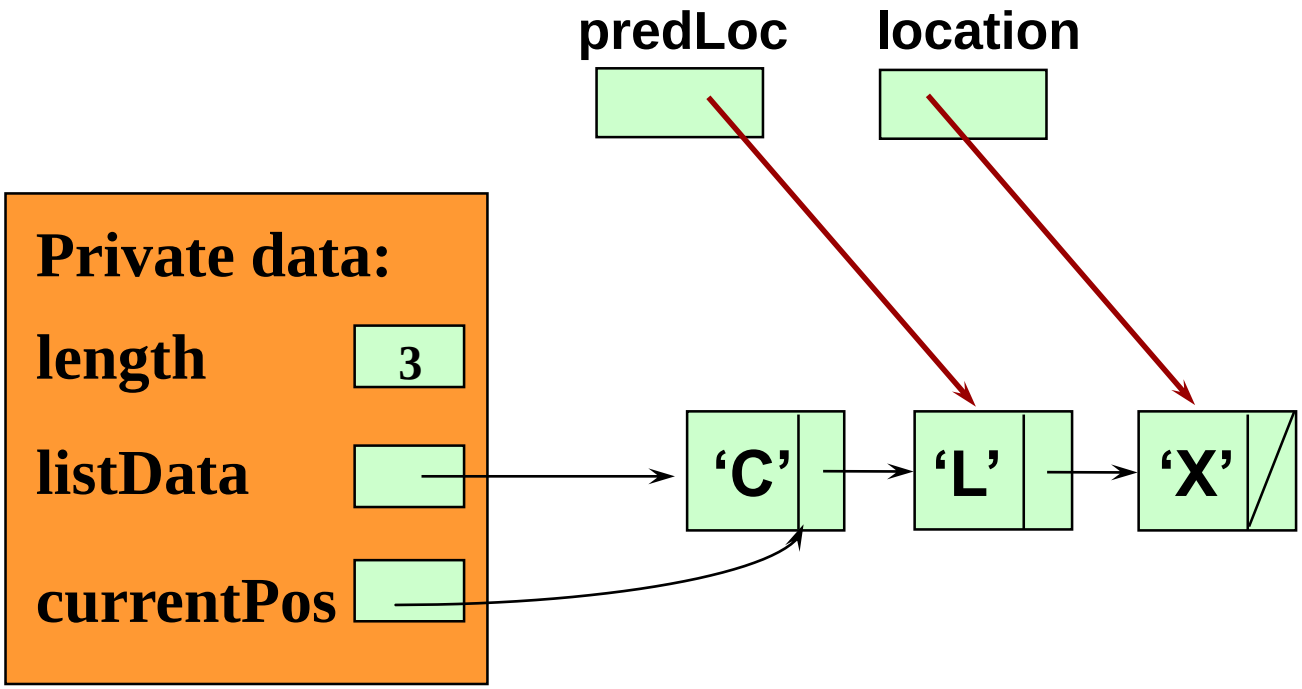
Finding proper position for 'S'



Finding proper position for 'S'



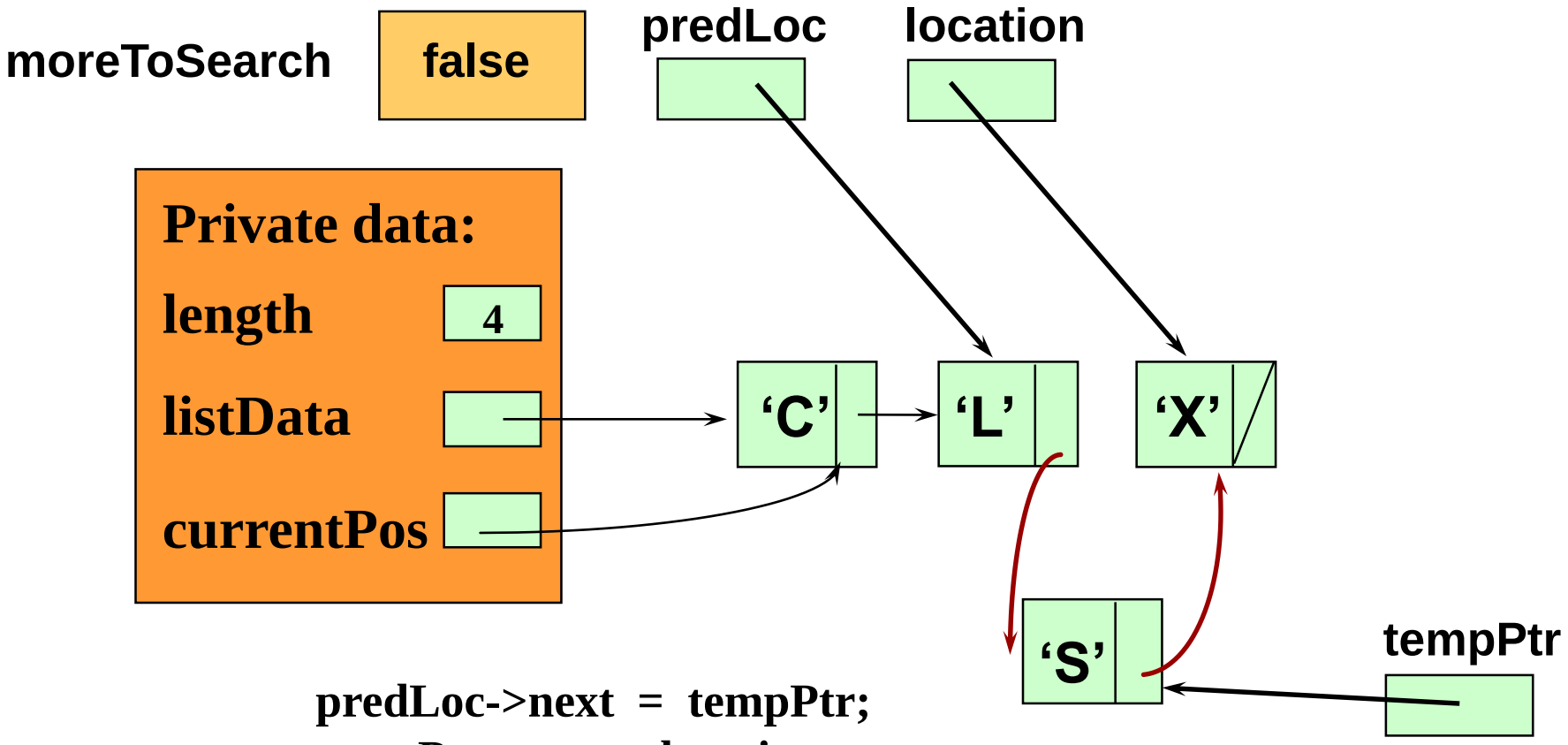
Finding proper position for 'S'



moreToSearch
But S<X

true

Inserting 'S' into proper position



```
void  SortedType :: InsertItem ( ItemType  item )
{   bool  moreToSearch ;
      NodeType<ItemType>*  location, predLoc, tempPtr ;
      tempPtr = new  NodeType<ItemType> ;
      tempPtr->info = item ;
      if (listData==NULL)  {listData=tempPtr;   tempPtr->next=NULL;    length++;          }
      else     {
            location = predLoc=listData ;
            moreToSearch = ( location !=NULL ) ;
            while ( moreToSearch  )
            {  switch ( item.ComparedTo( location->info ) )
               {    case  LESS       :  moreToSearch = false ;
                                        break ;
                    case  GREATER  :   predLoc=location;
                                       location=location->next;
                                       moreToSearch = ( location !=NULL ) ;
                                       break ;
                }     }
            if (predLoc==location) { // add to front
                tempPtr->next = listData;    listData=tempPtr;      }
            else {       // add between  two nodes or to the end(location==NULL)
                  predLoc->next = tempPtr;
                  tempPtr->next=location;        }
            length++ ;
      }
}
```