CSE225: Data Structure and Algorithms

Unsorted List

Lecture-07

List Definitions

Linear relationship

Each element except the first has a unique predecessor, and

Each element except the last has a unique successor.

Length

The number of items in a list;

The length can vary over time.

List Definitions

Unsorted list

A list in which data items are placed in no particular order;

the only relationship between data elements is the list predecessor and successor relationships.

Sorted list

A list that is sorted by the value in the key;

There is a semantic relationship among the keys of the items in the list.

Key

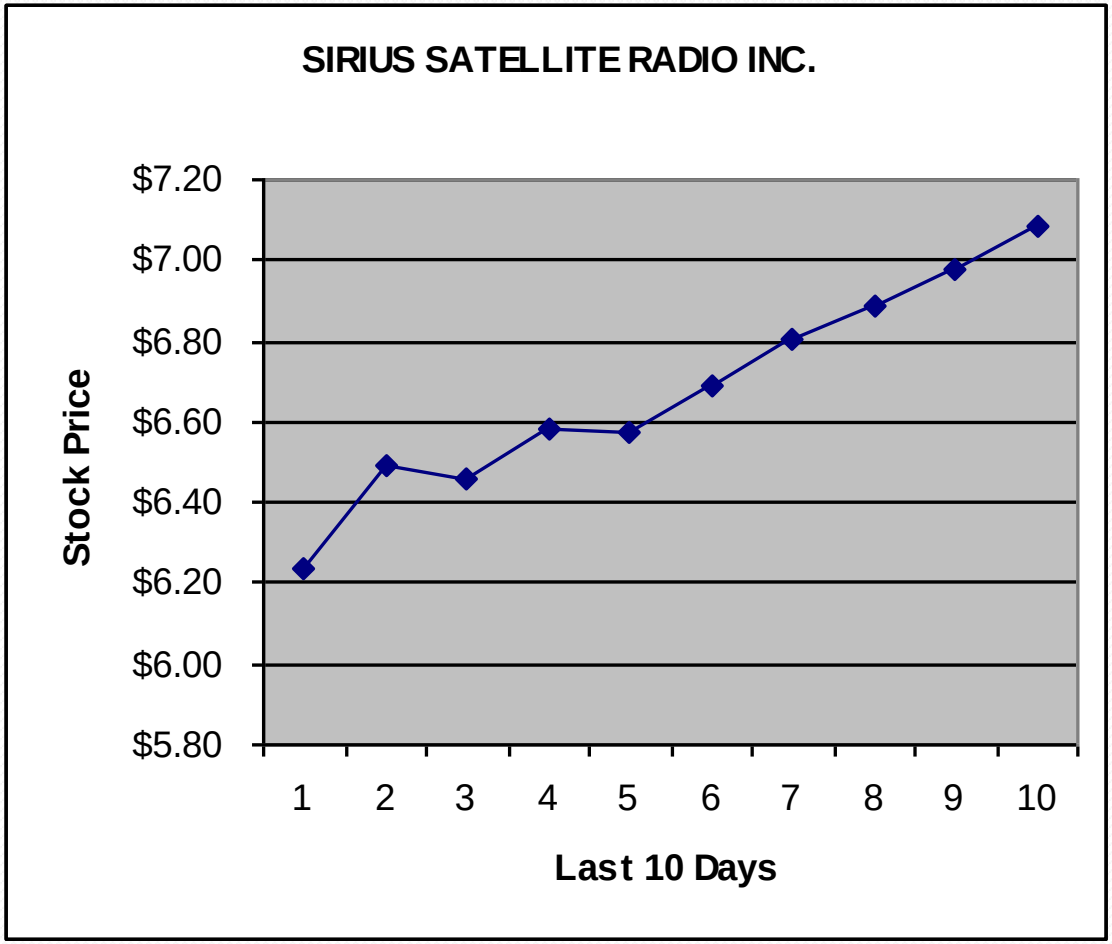
The attributes that are used to determine the logical order of the list.

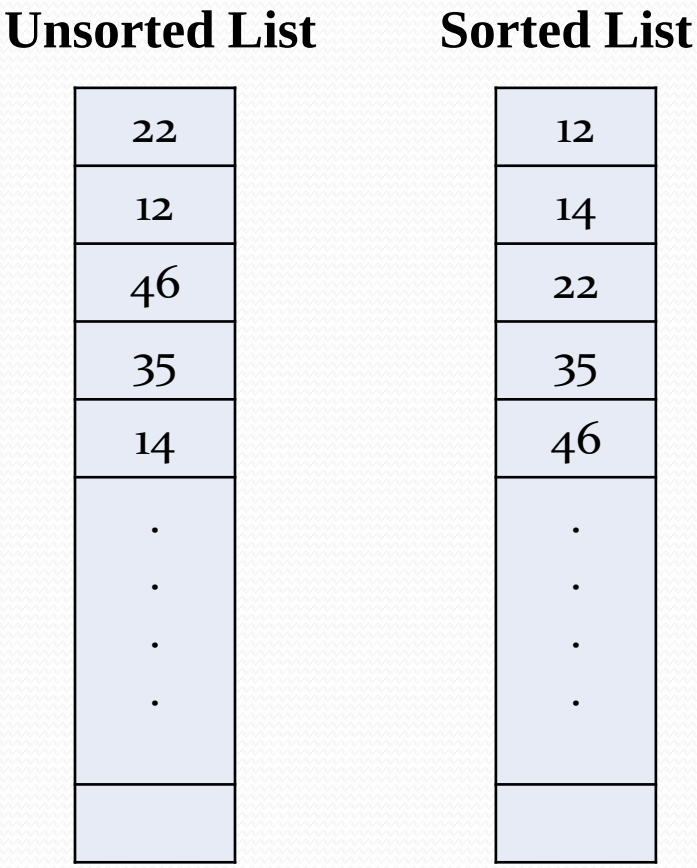
Data vs. Information

Data

- 6.34
- 6.45
- 6.39
- 6.62
- 6.57
- 6.64
- 6.71
- 6.82
- 7.12
- 7.06

Information




Unsorted List
Sorted List
22
12
46
35
14
12
14
22
35
46

Sorted List

ID	Name	Address
22	Jack Black	120 S. Virginia Street
45	Simon Graham	6762 St Petersburg
59	Susan O'Neal	1807 Glenwood, Palm Bay
66	David peterson	1207 E. Georgetown

Key

Abstract Data Type (ADT)

- **A data type whose properties (domain and operations) are specified independently of any particular implementation.**

Data from 3 different levels

- ***Application (or user) level:***

modeling real-life data in a specific context.

Why

- ***Logical (or ADT) level:***

abstract view of the domain and operations.

What

- ***Implementation level:***

specific representation of the structure to hold the data items, and the coding for operations.

How

ADT Operations

- **Constructor**
- **Transformer**
- **Observer**
- **Iterator**

Sorted and Unsorted Lists

UNSORTED LIST

Elements are placed into the list in no particular order.

SORTED LIST

List elements are in an order that is sorted in some way

- **either numerically,**
- **alphabetically by the elements themselves, or**
- **by a component of the element**
  - **called a KEY member**

ADT Unsorted List Operations

Transformers

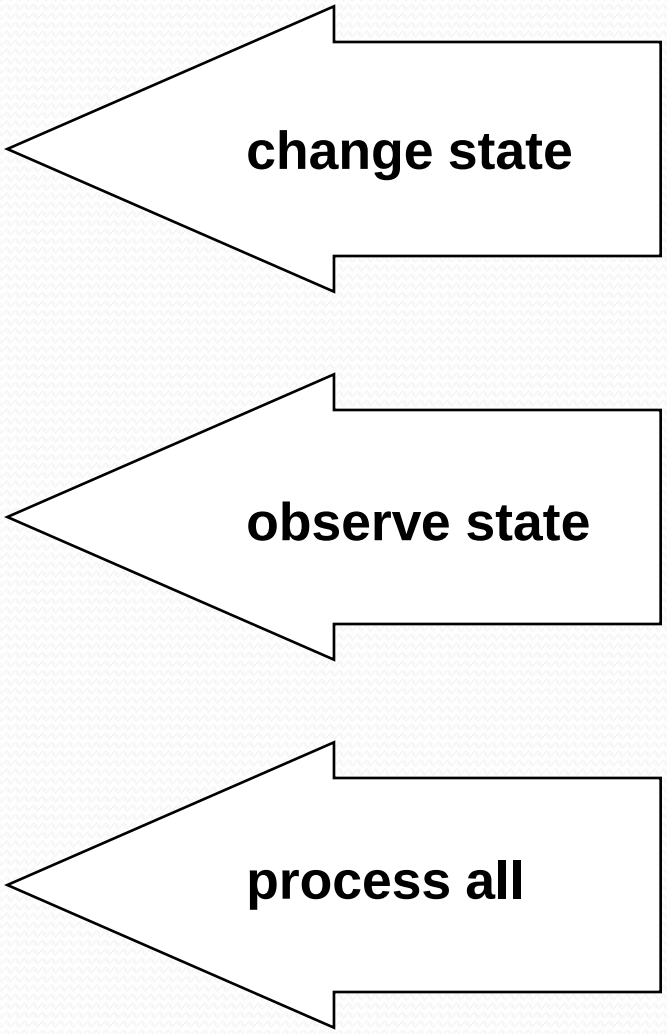
- MakeEmpty
- PutItem
- DeleteItem

Observers

- IsFull
- GetLength
- IsEmpty

Iterators

- ResetList
- GetNextItem



What is a Generic Data Type?

A generic data type (template class in C++) is a type for which the operations are defined but the types of the items being manipulated are not defined.

One way to simulate such a type for our UnsortedList ADT is via a user-defined class **`ItemType`** with

Specification of UnsortedType

Structure:	The list has a special property called the *current position* - the position of the last element accessed by **GetNextItem** during an iteration through the list. Only **ResetList** and **GetNextItem** affect the *current position*.
Operations (provided by Unsorted List ADT):	
MakeEmpty	
Function	Initializes list to empty state.
Precondition	
Postcondition	List is empty.
Boolean IsFull	
Function	Determines whether list is full.
Precondition	List has been initialized.
Postcondition	Returns true if list is full and false otherwise.

Specification of UnsortedType

int GetLength	
Function	Determines the number of elements in list.
Precondition	List has been initialized.
Postcondition	Returns the number of elements in list.
ItemType GetItem (ItemType item, Boolean &found)	
Function	Retrieves list element whose key matches item's key (if present).
Precondition	List has been initialized. Key member of item is initialized.
Postcondition	If there is an element some Item 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.**
PutItem (ItemType item)	
Function	Adds item to list.
Precondition	List has been initialized. List is not full. item is not in the list.
Postcondition	item is in the list**. List is changed.**

Specification of UnsortedType

DeleteItem (ItemType item)	
Function	Deletes the element whose key matches item's key.
Precondition	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-condition	No element in list has a key matching item's key.
ResetList	
Function	Initializes current position for an iteration through the list.
Precondition	List has been initialized.
Post-condition	Current position is prior to first element in list.
ItemType GetNextItem ()	
Function	Gets the next element in list.
Precondition	List has been initialized. Current position is defined. Element at current position is not last in list.
Post-condition	Current position is updated to next position. item is a copy of element at current position.

```
//  SPECIFICATION FILE    ( unsorted.h )
#include  "ItemType.h"
class  UnsortedType // declares a class data type
{
public :
  UnsortedType ( ); //  8 public member functions

  bool IsFull ( )     //const;
  bool IsEmpty ( )    const;
  int GetLength ( )   const ; // returns length of list
  ItemType GetItem ( ItemType  item, bool&  found);
  void PutItem ( ItemType  item );
  void DeleteItem ( ItemType  item );
  void ResetList ( );
  ItemType GetNextItem ();

private :
 int length;                  //  3 private data members
 ItemType info[MAX_ITEMS];
 int  currentPos;
};
```

Class Constructor

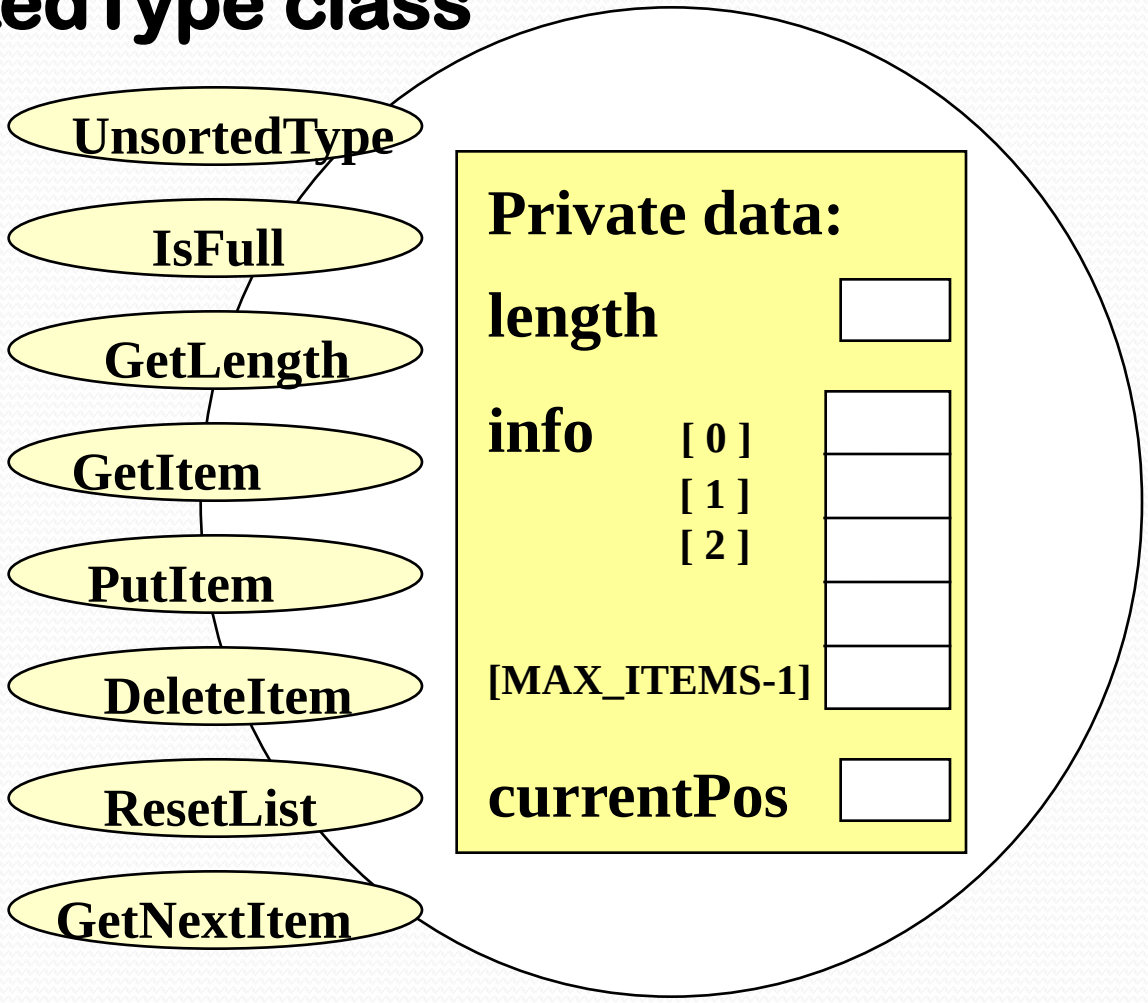
A special member function of a class that is implicitly invoked when a class object is defined.

Class Constructor Rules

1. A **constructor cannot return a function value**,
   and has no return value type.
2. A **class may have several constructors**.
   The compiler chooses the appropriate constructor by the number and types of parameters used.
3. Constructor parameters are placed in a parameter list in the declaration of the class object.
4. The **parameter less constructor** is the **default constructor**.
5. If a class has at least one constructor, and an array of class objects is declared, then one of the constructors must be the default constructor, which is invoked for each element in the array.

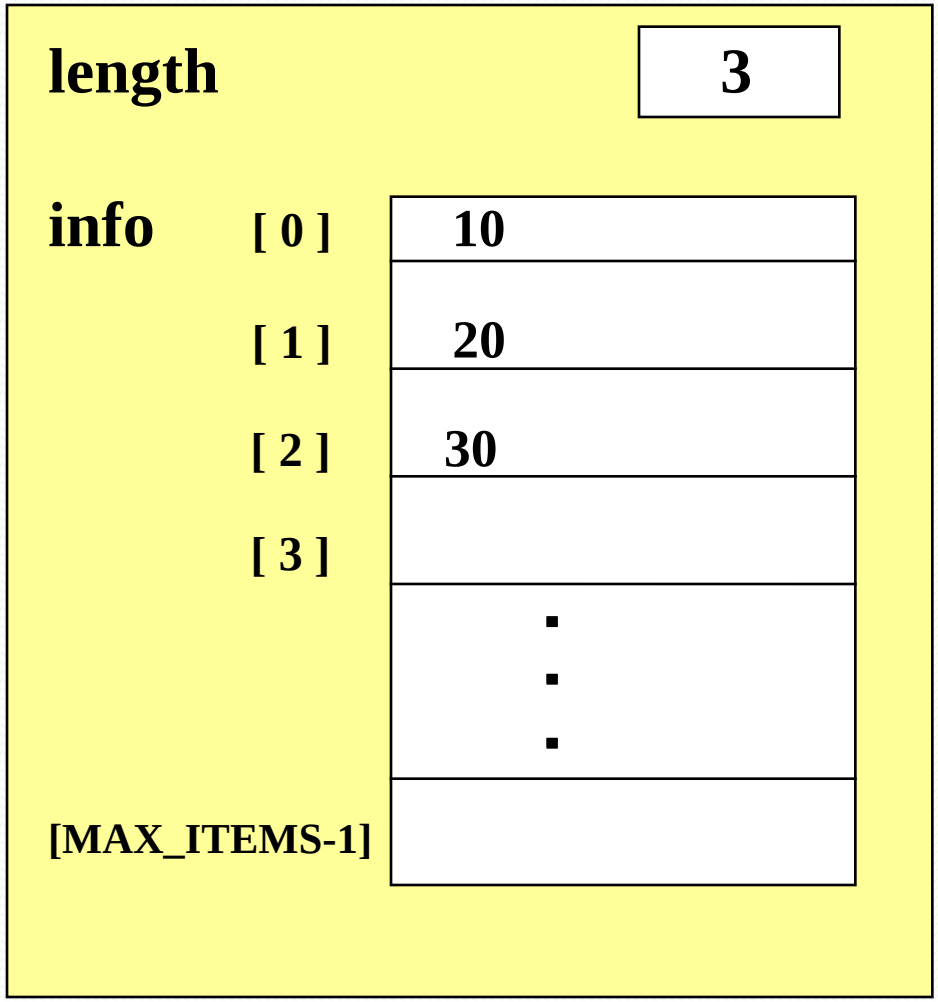
Class Interface Diagram

UnsortedType class



```
// IMPLEMENTATION FILE   ARRAY-BASED LIST   ( unsorted.cpp )
#include "itemtype.h"

void UnsortedType::UnsortedType ( )
// Pre: None.
// Post: List is empty.
{
  length  =  0;
  current_pos=-1
}

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++;


}
```

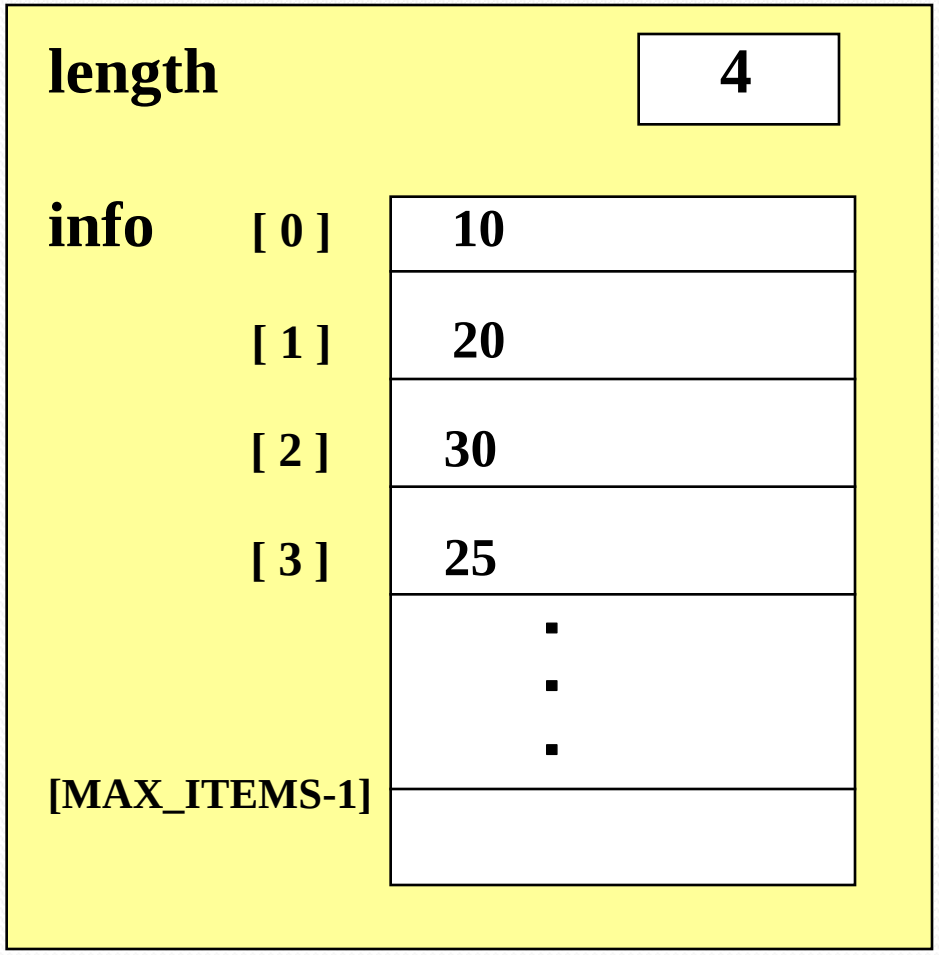
Before Inserting 25 into an Unsorted List



The item will be placed into the length location, and length will be incremented.

```
info[length] = item;
length++;
```

After Inserting 25 into an Unsorted List



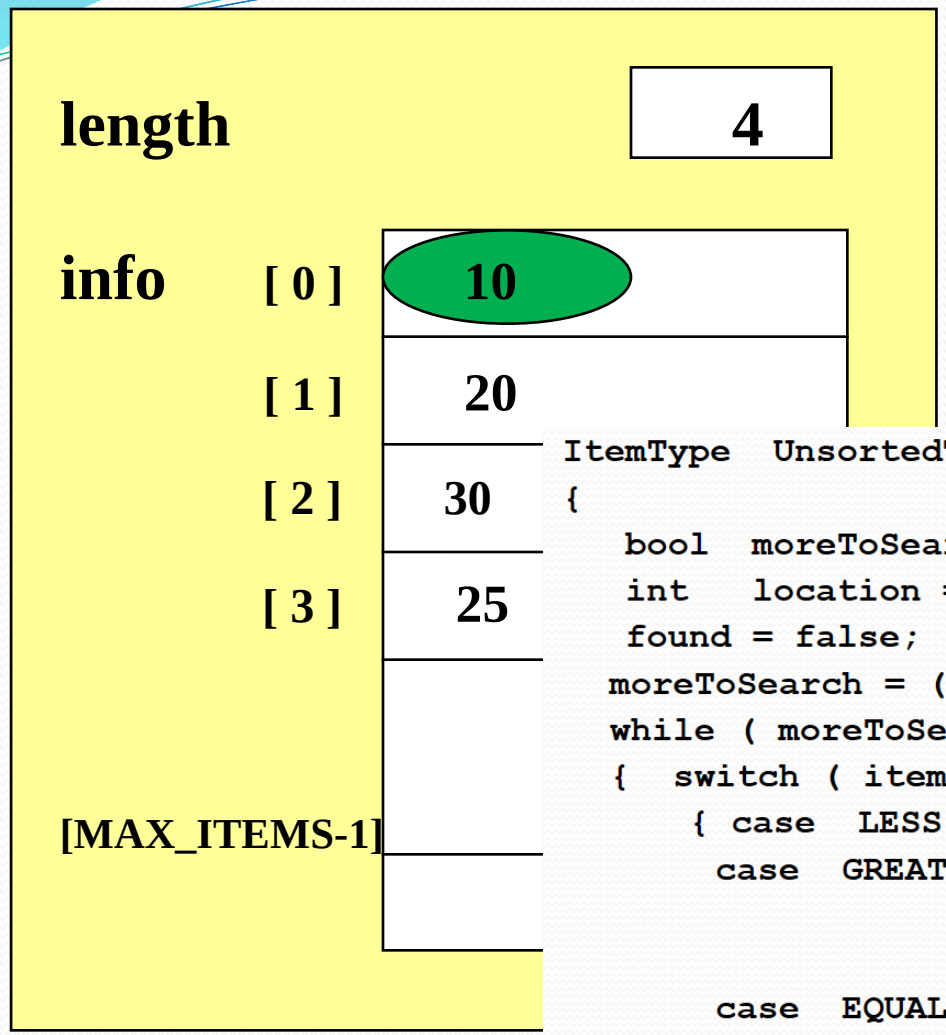
```
int UnsortedType::GetLength( )  const
// Pre: List has been inititalized.
// Post: Function value == ( number of elements in
//       list ).
{
  return  length;
}


bool  UnsortedType::IsFull ( )  const
//  Pre: List has been initialized.
// Post: Function value == ( list is full ).
{
  return ( length == MAX_ITEMS );
}
```

```
ItemType  UnsortedType::GetItem ( 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 is returned; otherwise, input item is returned.
{
   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;
     }
    }
  return item;
}
```


24

```
template <class ItemType>
void UnsortedType<ItemType>::RetrieveItem(ItemType& item,
bool &found)
{
    int location = 0;
    bool moreToSearch = (location < length);
    found = false;
    while (moreToSearch && !found)
    {
        if(item == info[location])
        {
            found = true;
            item = info[location];
        }
        else
        {
            location++;
            moreToSearch = (location < length);
        }
    }
}
```

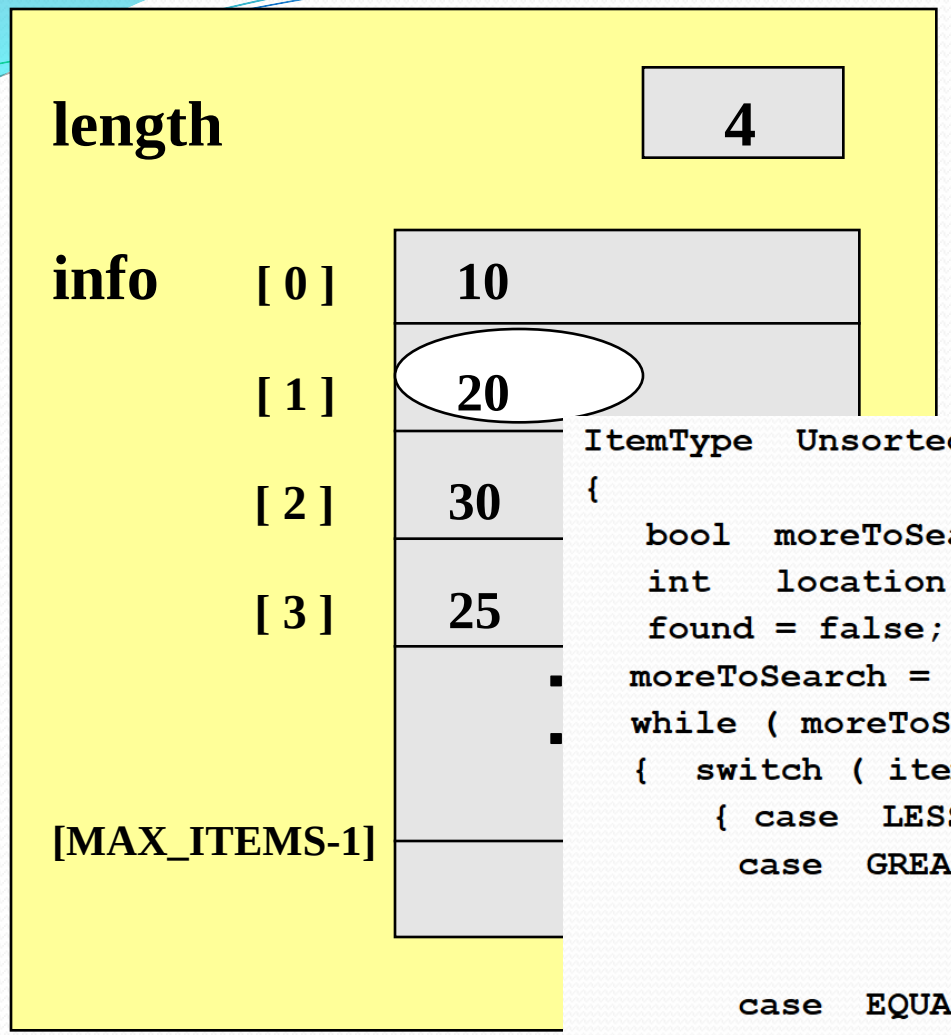
Getting 35 from an Unsorted List



moreToSearch:	true
found:	false
location:	0

```
ItemType  UnsortedType::GetItem (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;
      }
    }
  return item;
}
```

Getting 35 from an Unsorted List



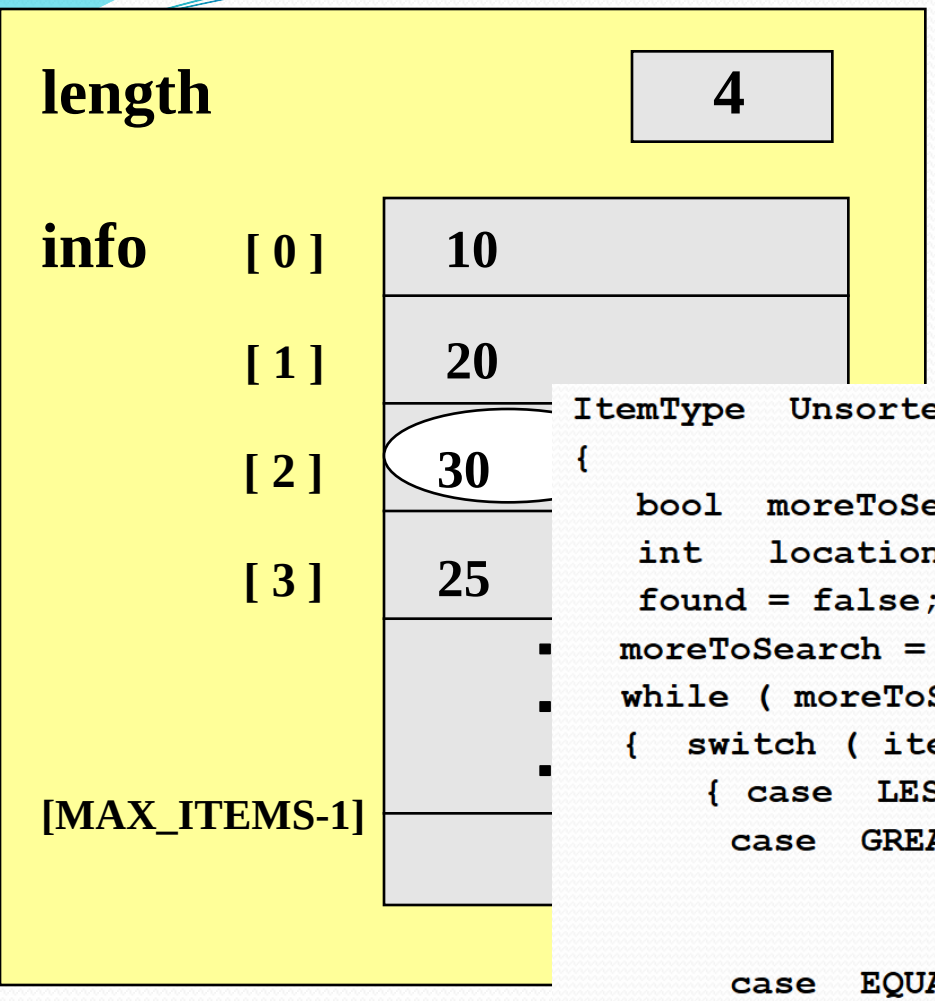
moreToSearch: true

found: false

location: 1

```
ItemType  UnsortedType::GetItem (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;
      }
    }
  return item;
}
```

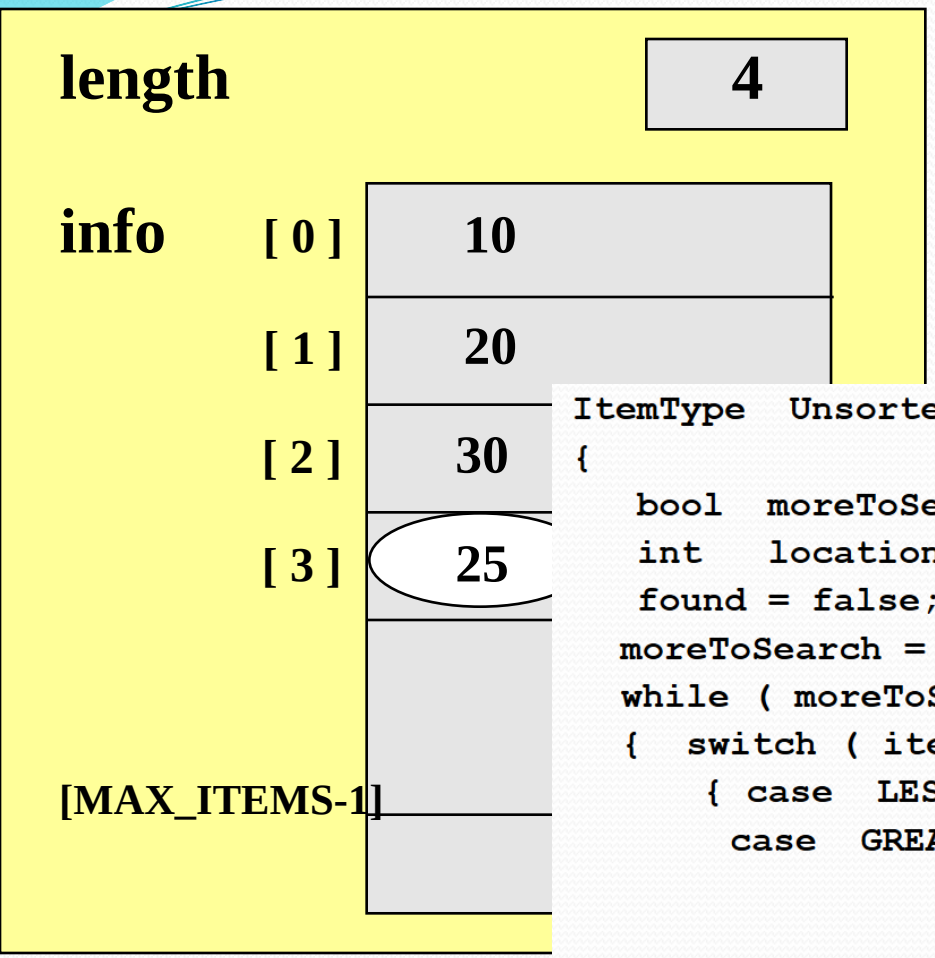
Getting 35 from an Unsorted List



moreToSearch:	true
found:	false
location:	2

```
ItemType  UnsortedType::GetItem (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;
      }
    }
  return item;
}
```

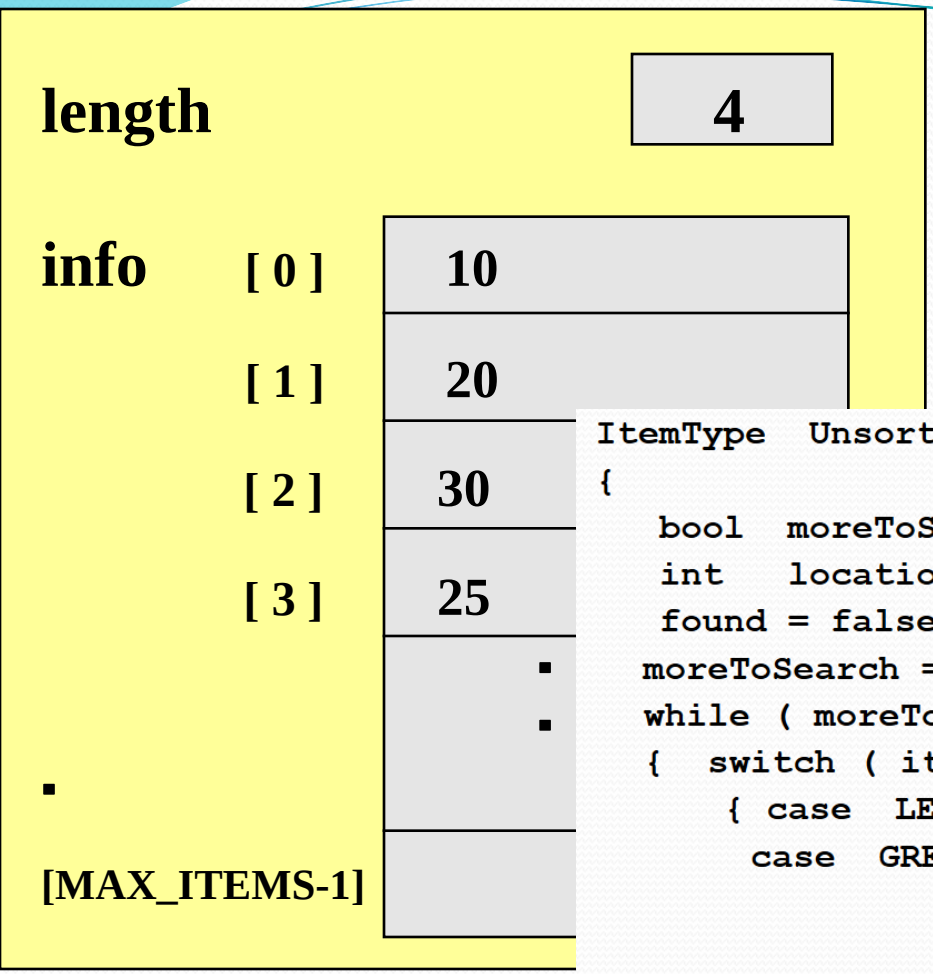
Getting 35 from an Unsorted List



moreToSearch:	true
found:	false
location:	3

```
ItemType  UnsortedType::GetItem (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;
      }
    }
  return item;
}
```

Getting 35 from an Unsorted List

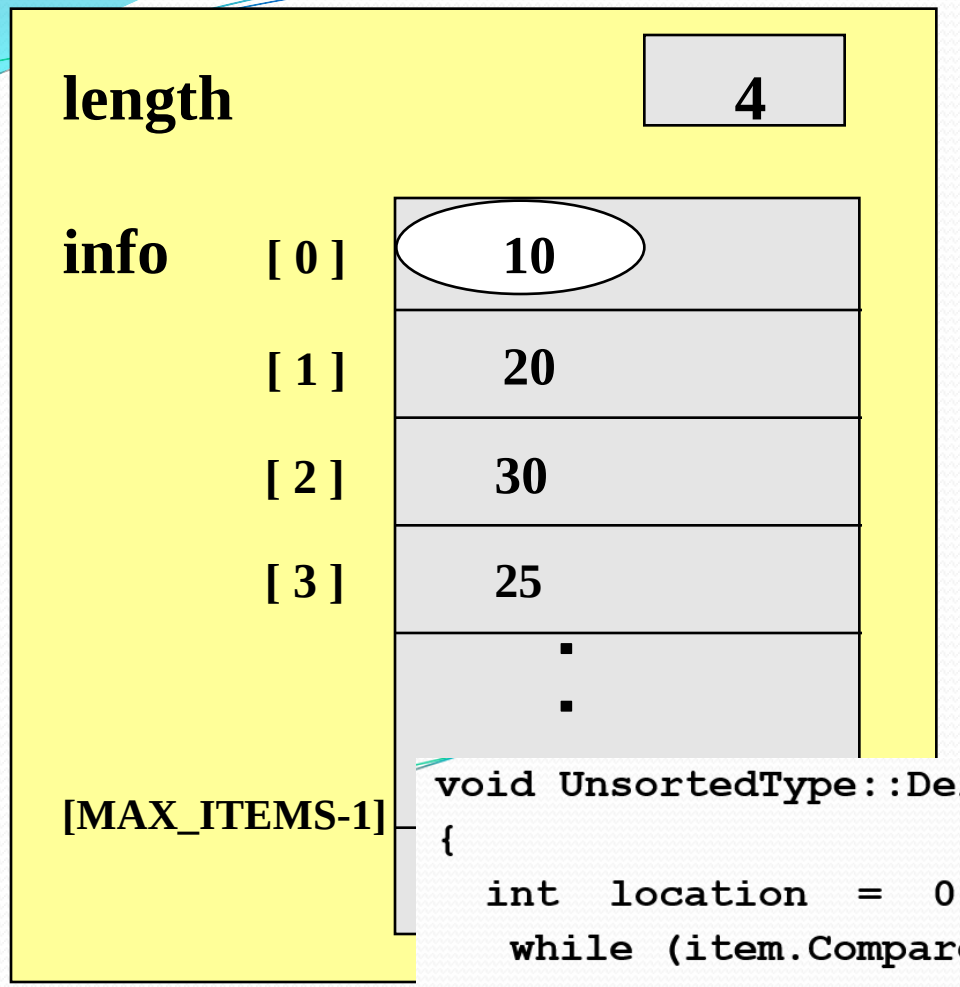


moreToSearch:	false
found:	false
location:	4

```
ItemType  UnsortedType::GetItem (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;
      }
    }
  return item;
}
```

```
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-- ;
}
```

Deleting 20 from an Unsorted List

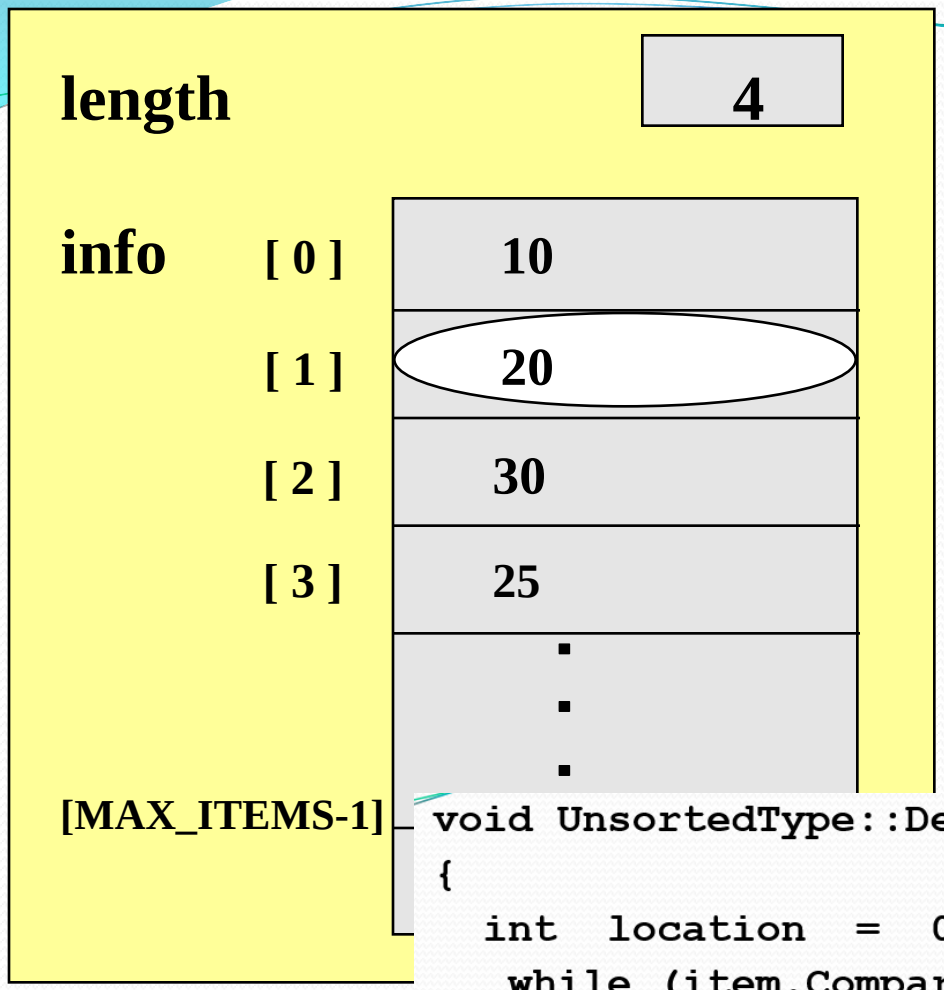


location: 0

Key 20 has
not been matched.

```
void UnsortedType::DeleteItem ( ItemType  item )
{
  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-- ;
}
```

Deleting 20 from an Unsorted List

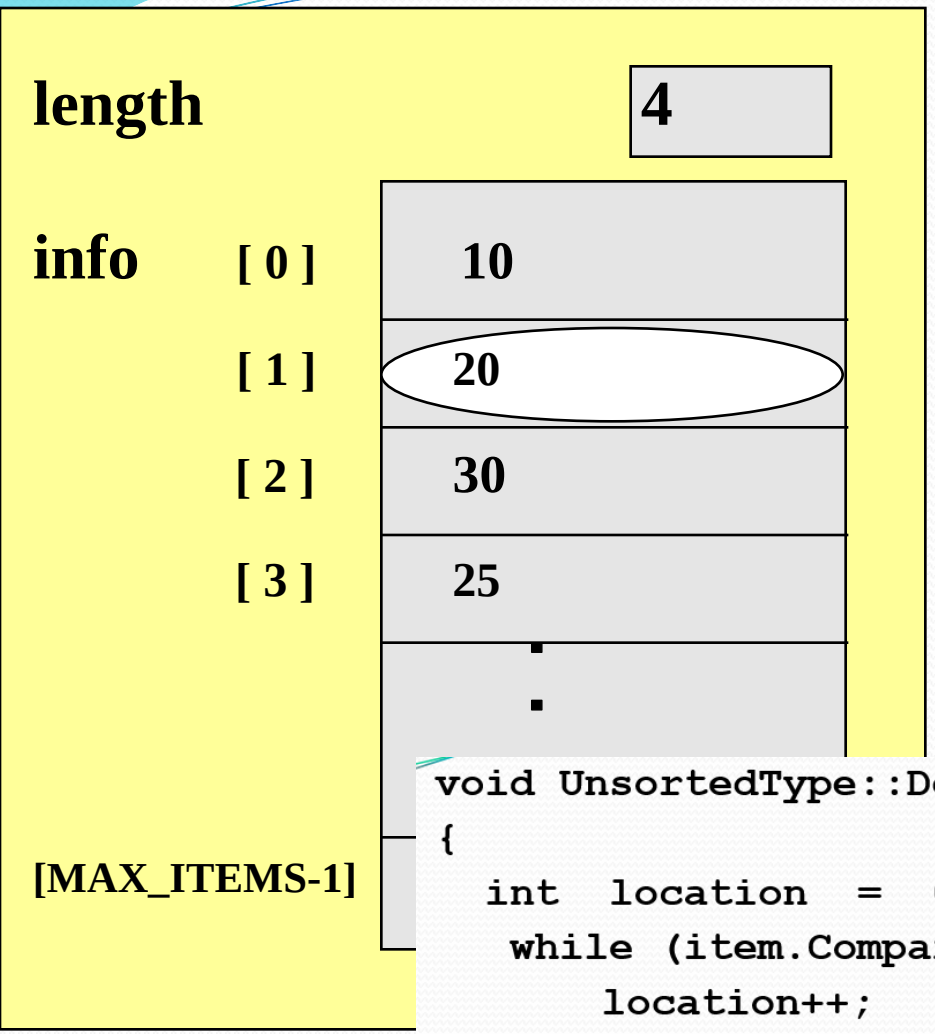


location: 1

Key 20 has been matched.

```
void UnsortedType::DeleteItem ( ItemType  item )
{
  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-- ;
}
```

Deleting 20 from an Unsorted List

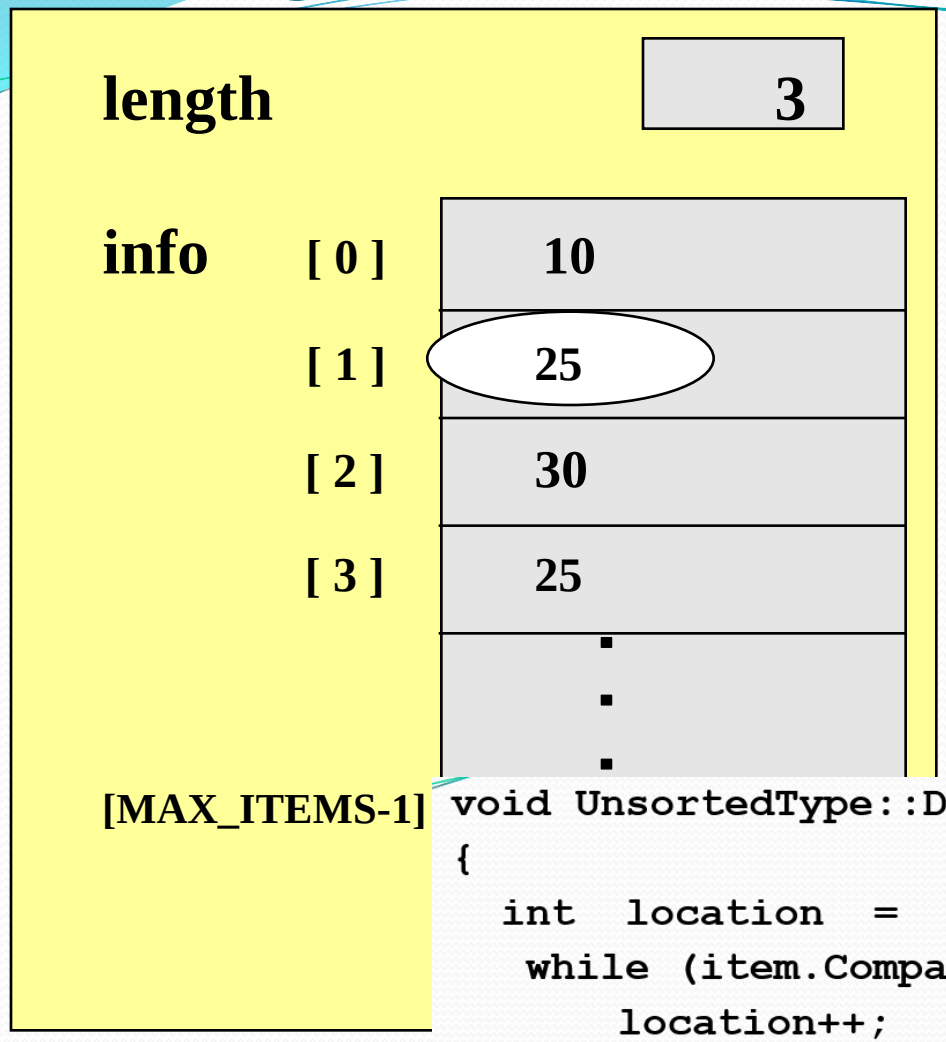


location: 1

Placed copy of last list element into the position where the key 20 was before.

```
void UnsortedType::DeleteItem ( ItemType  item )
{
  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-- ;
}
```

Deleting 20 from an Unsorted List



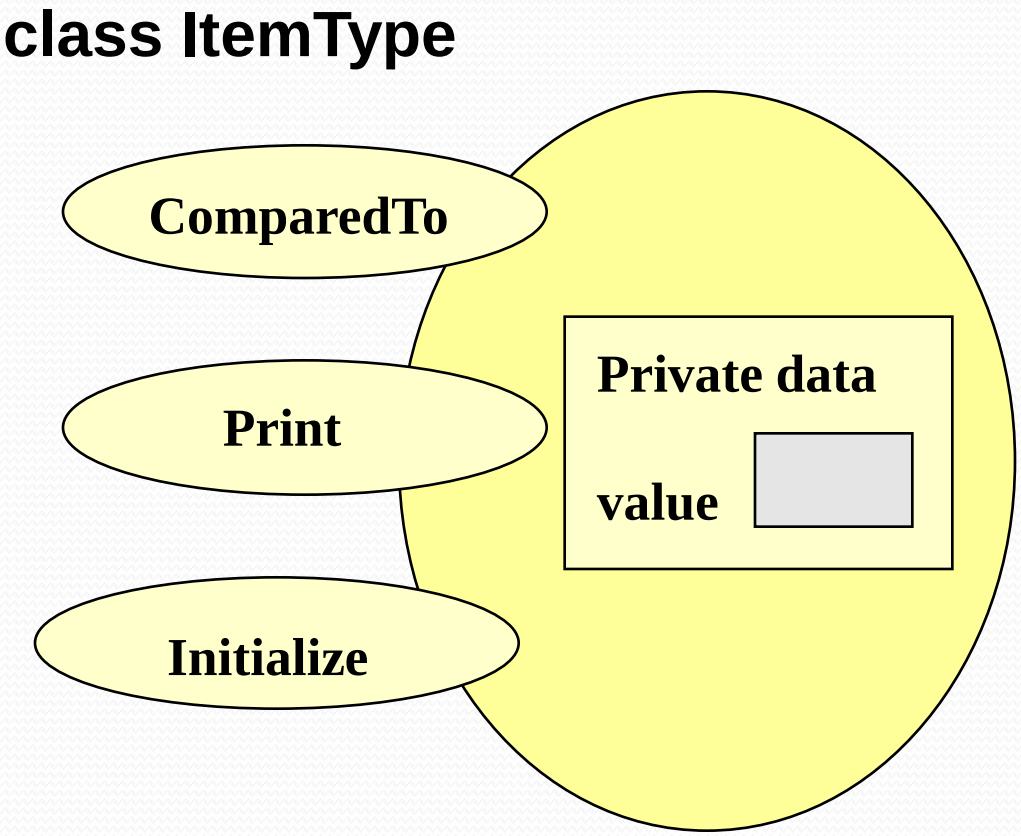
location: 1

Decremented length.

```
void UnsortedType::DeleteItem ( ItemType  item )
{
  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;
}

ItemType UnsortedType::GetNextItem ()
// 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++;
  return info [currentPos];
}
```

ItemType Class Interface Diagram



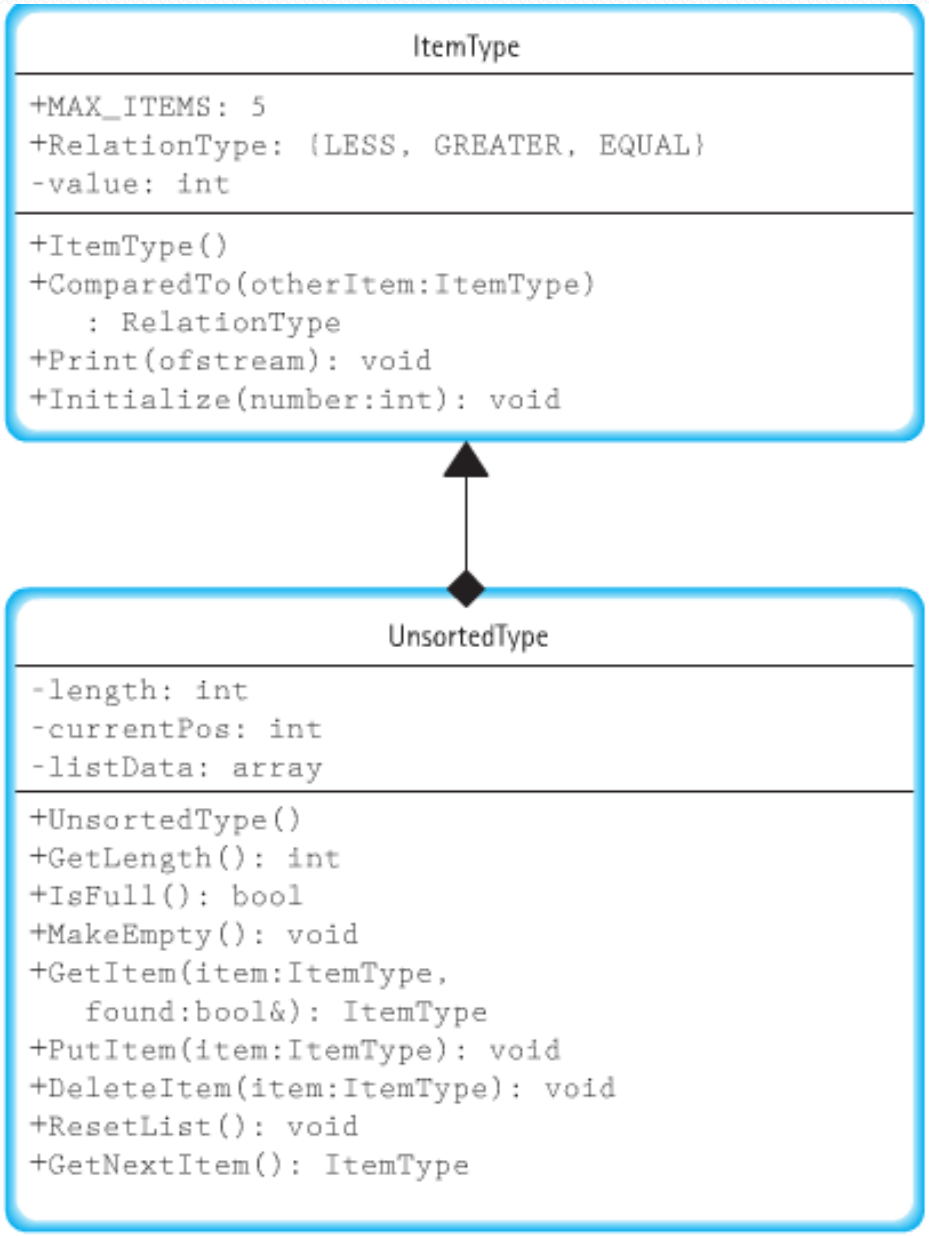
```
void UnsortedType::MakeEmpty ( )
// Post: list is empty.
{
  length =  0;
}
```

Specifying class ItemType

```
//  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  ComparedTo ( ItemType )  const;
  void     Print ( )  const;
  void     Initialize ( int  number ) ;

private :                    // 1 private data member
  int value ;                // could be any different
                             // type, including a class
} ;
```

```
// IMPLEMENTATION FILE          ( itemtype.cpp )
// Implementation depends on the  data type of value.

#include  "itemtype.h"
#include  <iostream>

RelationType Itemtype::ComparedTo(ItemType otherItem)
  const
{
  if  ( value  <  otherItem.value )
        return  LESS;
  else  if ( value  > otherItem.value )
        return  GREATER;
  else  return  EQUAL;
}
 void ItemType::Print ( ) const
{
  using namespace std;
  cout  <<  value  <<  endl;
}
void ItemType::Initialize ( int  number )
{
  value  =  number;
}
```

UML diagrams



Remember?

- A list is a homogeneous collection of elements, with a linear relationship between elements.
- 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

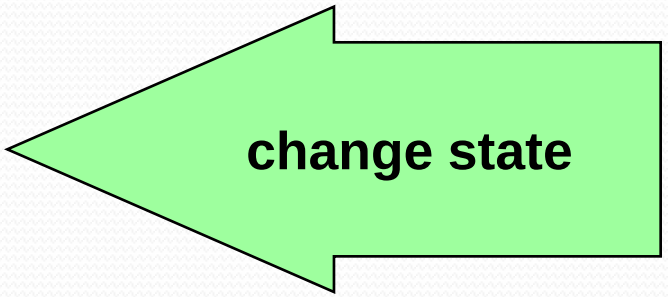
- MakeEmpty
- PutItem
- DeleteItem

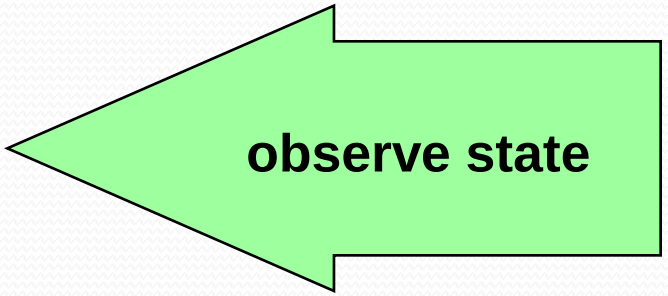
Observers

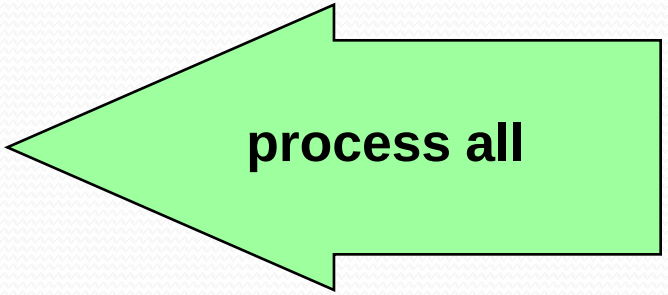
- IsFull
- GetLength
- GetItem

Iterators

- ResetList
- GetNextItem





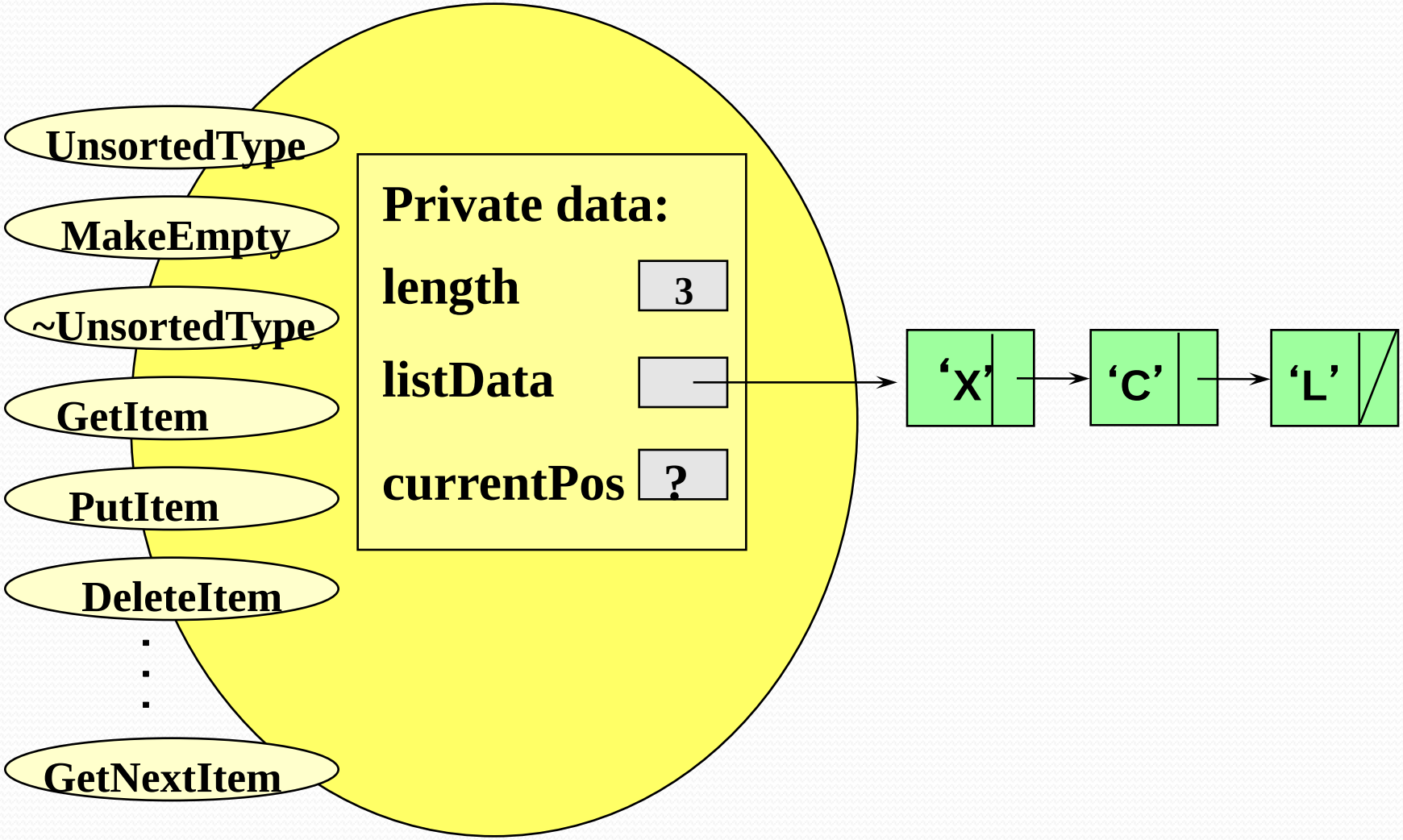


```
#include  "ItemType.h"          //  unsorted.h

    .   .   .
 struct NodeType;

class  UnsortedType
{
public :                        //  LINKED LIST IMPLEMENTATION
  // The public interface is the same

private :
  // The private part is different
  NodeType<ItemType>*  listData;
  int length;
  NodeType<ItemType>*  currentPos;
};
```

```
struct NodeType {
   ItemType info;
   NodeType*  next;
} ;
```

Do we have to keep a length field?
Do we need an IsFull?

class UnsortedType<char>

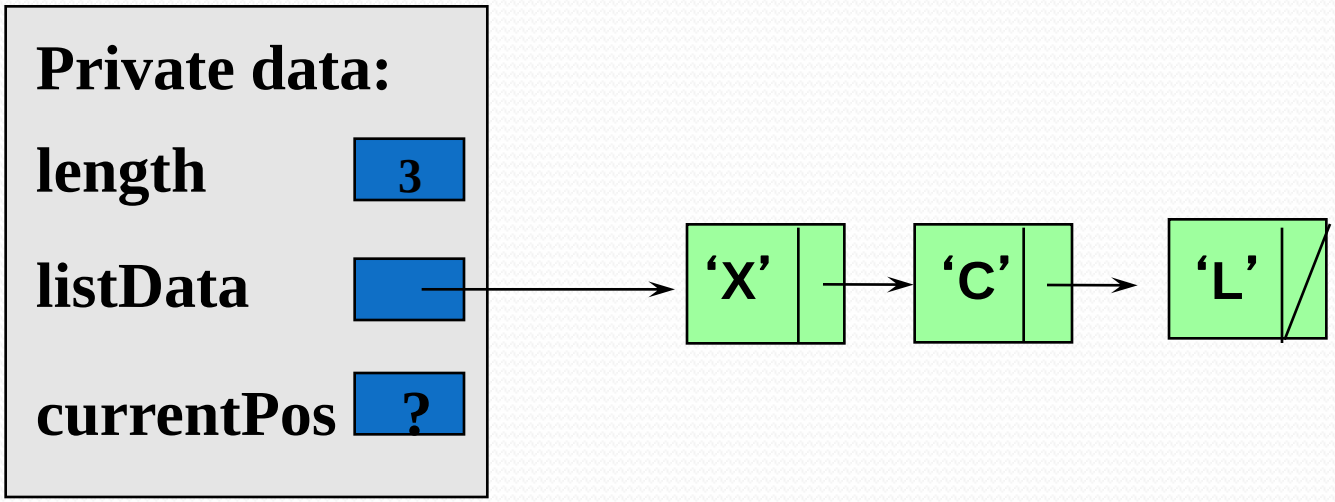


```
// LINKED LIST IMPLEMENTATION  ( unsorted.cpp )
#include "itemtype.h"

UnsortedType::UnsortedType ( ) // constructor
// Pre: None.
// Post: List is empty.
{
  length  =  0;
  listData = NULL;
}

int  UnsortedType::GetLength(  )  const
// Post:  Function value = number of items in the list.
{
  return  length;
}
```

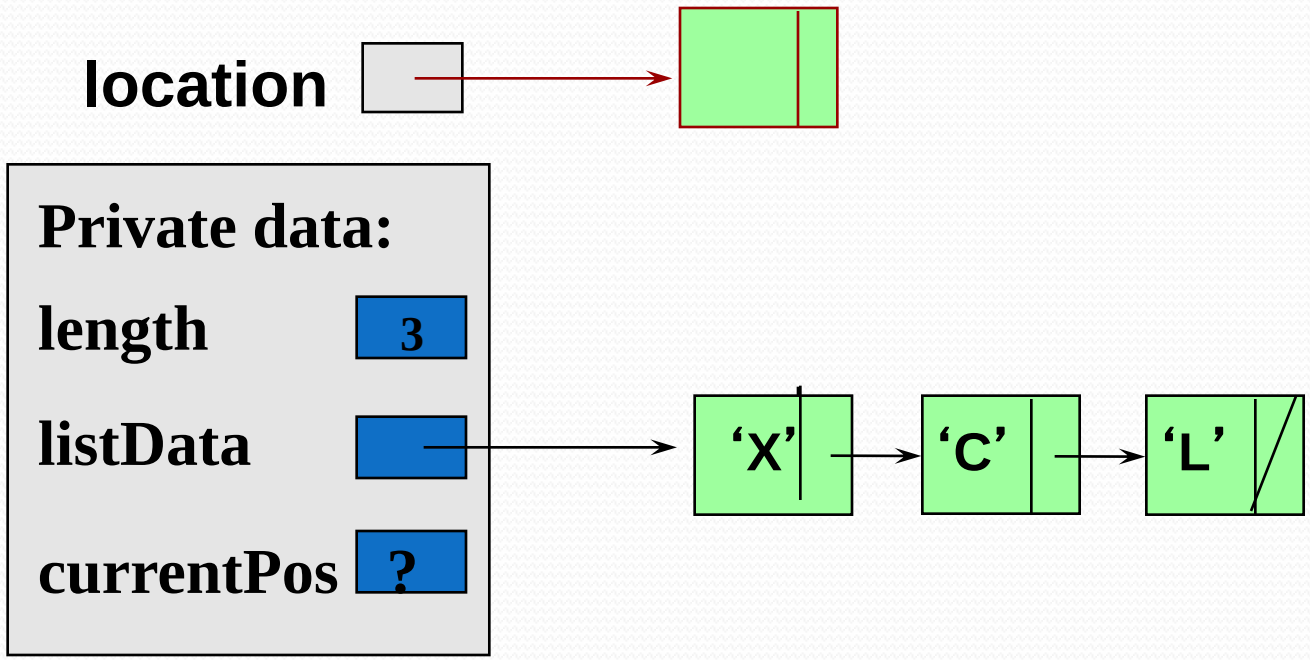
```
ItemType  UnsortedType::GetItem( ItemType  item, bool&  found )
// Pre: Key member of item is initialized.
// Post: If found, item's key matches an element's key in the list
// a copy of that element is returned; otherwise,
// original item is returned.
{
  bool  moreToSearch;
   NodeType<ItemType>*  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 );
       }
   }
    return item;
}
```

```
void UnsortedType::PutItem ( 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++;
}
```

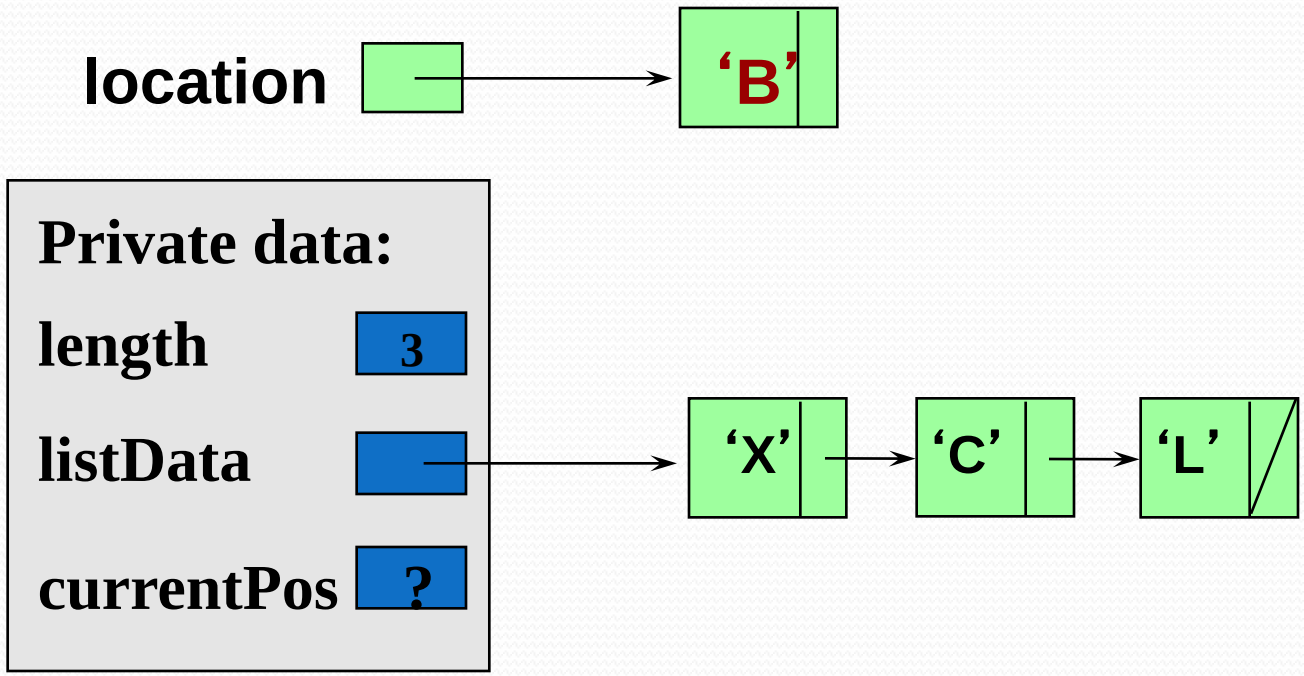
Inserting ‘B’ into an Unsorted List



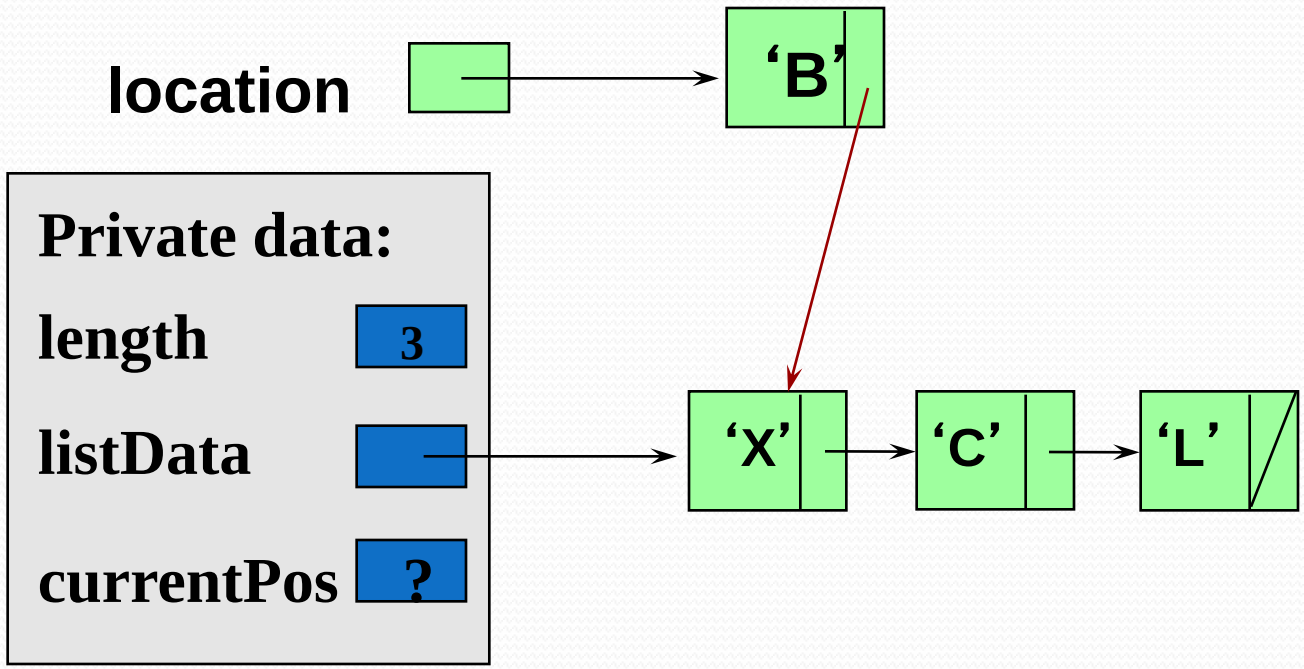
item ‘B’ location = new NodeType;



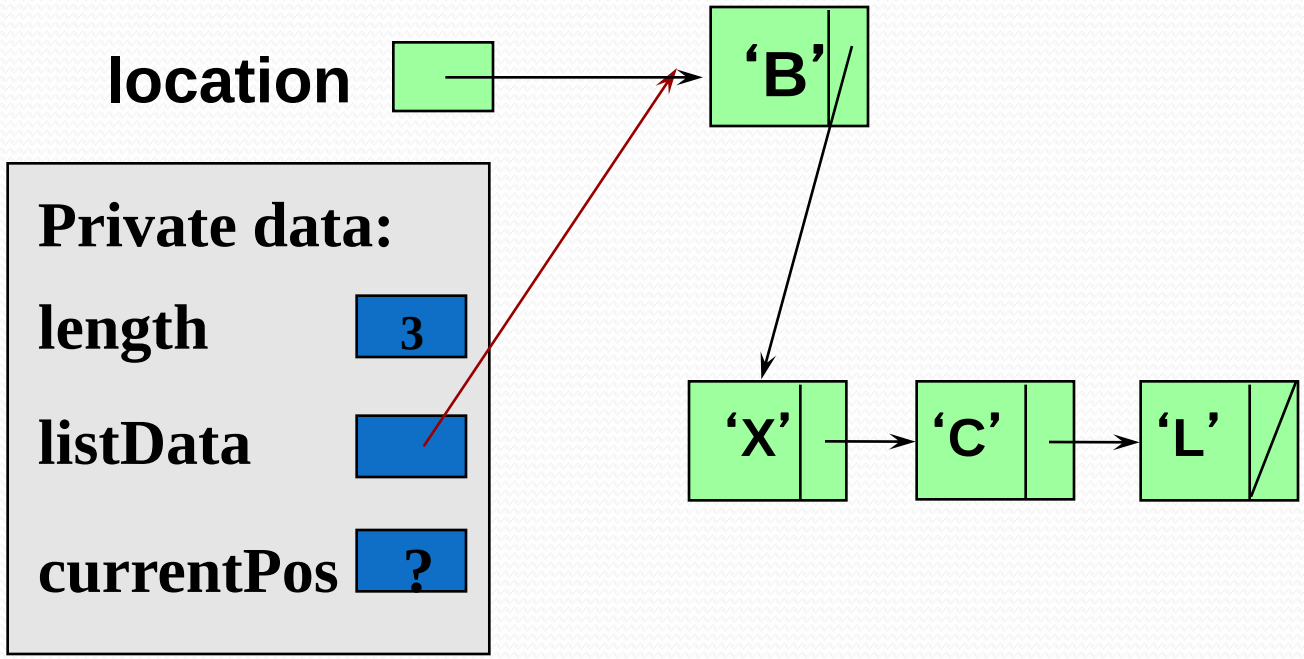
item ‘B’ location->info = item ;



item ‘B’ location->next = listData ;

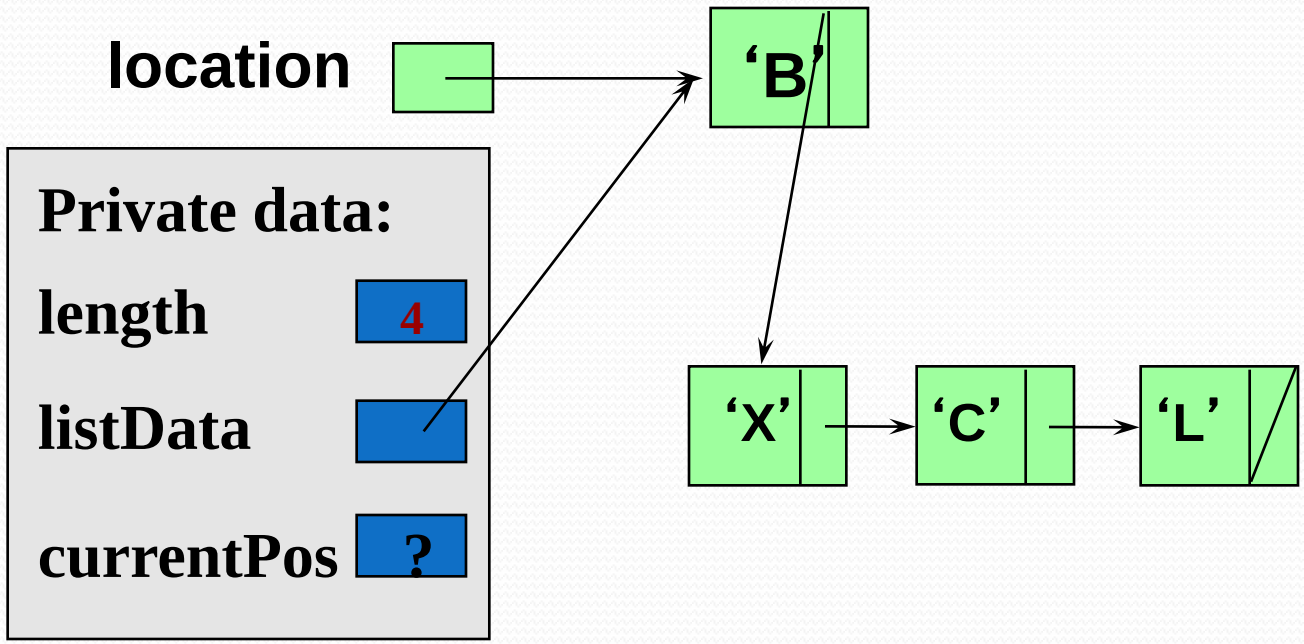


item ‘B’ listData = location ;

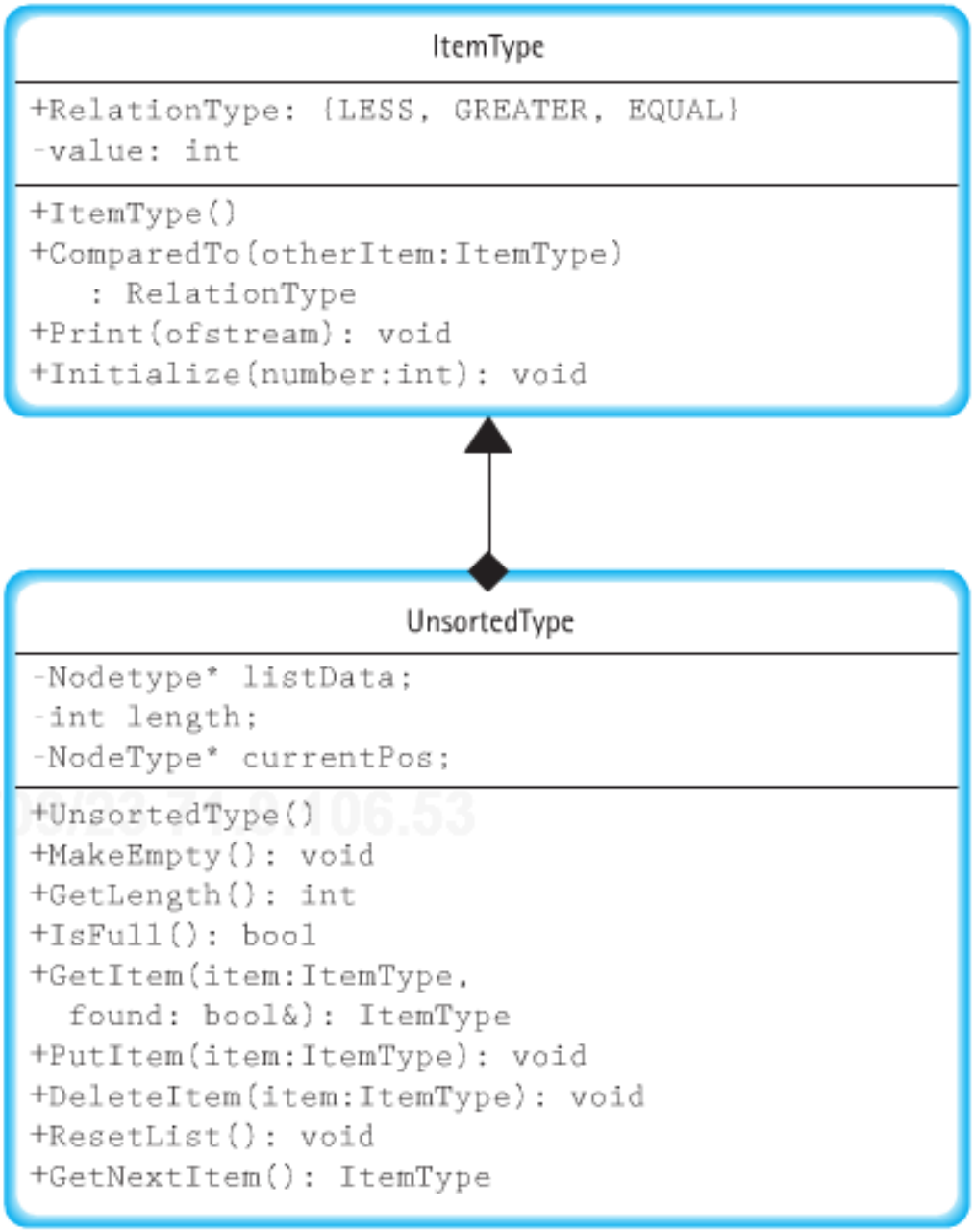


item ‘B’

length++ ;



UML diagrams



Big-O Comparison of Unsorted List Operations

Array Implementation		Linked Implementation
O(1)	Class constructor	O(1)
O(1)	MakeEmpty	O(*N*)
O(1)	IsFull	O(1)
O(1)	GetLength	O(1)
O(1)	ResetList	O(1)
O(1)	GetNextItem	O(1)
O(*N*)	GetItem	O(*N*)
	PutItem	
O(1)	Find	O(1)
O(1)	Insert	O(1)
O(1)	Combined	O(1)
	DeleteItem	
O(*N*)	Find	O(*N*)
O(1)	Delete	O(1)
O(*N*)	Combined	O(*N*)

Overview

- The order of adding 1 to each element in a one dimensional array of *N* integers.

A. O(1)

B. O(log*N*)

C. O(*N*)

D. O(*N* log*N*)

E. O(*N***N*)

- The order of adding 1 to each element in a square two dimensional array of integers where the number of rows is *N*.

A. O(1)

B. O(log*N*)

C. O(*N*)

D. O(*N* log*N*)

E. O(*N***N*)

Overview

- What is special about the last node in a dynamic linked list?

A. Its component (data) member is empty.

B. Its component (data) member contains the value 0.

C. Its link member is empty.

D. Its link member contains the value NULL.

E. It has no link member.

- A fixed-sized structure; array
  - the mechanism for accessing the structure is built into C++.

- A variable-sized, user-defined structure; list
  - the mechanism for accessing the structure must be provided through functions.

Overview

- To prevent a compile-time error, how should the following code be changed?

```
struct ListNode                    // Line 1
{                                  // Line 2
        int      dataVal;          // Line 3
        NodeType* next;            // Line 4
};                                 // Line 5
```

A. Insert the following before line 1:

```
typedef ListNode* NodeType*;
```

B. Insert the following before line 1:

```
struct ListNode;
typedef ListNode* NodeType*;
```

C. Replace line 4 with the following:

```
ListNode* next;
```

D. Do either b or c above.

E. Do any of a, b, or c above.

Overview

- What symbol does C++ use to terminate the internal representation of strings?

A. 'n'
B. '\n'
C. '\0'
D. '\#'
E. C++ doesn't use a symbol to terminate a string.

- A generic data type is one in which the types of the items being manipulated are defined, but the operations are not.
- It is not possible to use a list without knowing how it is implemented.
- A constructor cannot be explicitly called by the client program.
- O(1) is called constant time.
- O(N) is called linear time.
- A destructor is a special operation that is implicitly called when a class object goes out of scope.

Overview

- Deleting from an unsorted list requires that the elements below the one being deleted be moved up one slot.
- The algorithms for finding an element in an unsorted list is O(n).
- The next item in a linked list always can be found by accessing the next physical location in memory.
- Given only the external pointer to a linked list, it is faster to insert a node at the front of the list than at the back.
- The external pointer is considered to be one of the nodes of a linked list
- If **`currPtr`** points to a node in a dynamic linked list, the operation **`currPtr++`** advances to the next node in the list.
- Reading components into an initially empty list is faster if the list is represented directly as an array rather than a linked list.
- With a list ADT, insertions and deletions *at the front of the list* are faster with a linked list representation than with a direct array representation.
- With a list ADT, insertions and deletions at the back of the list are faster with a linked list representation than with a direct array representation.