CMPSC 24: Lecture 9
Abstract Data Type: Lists

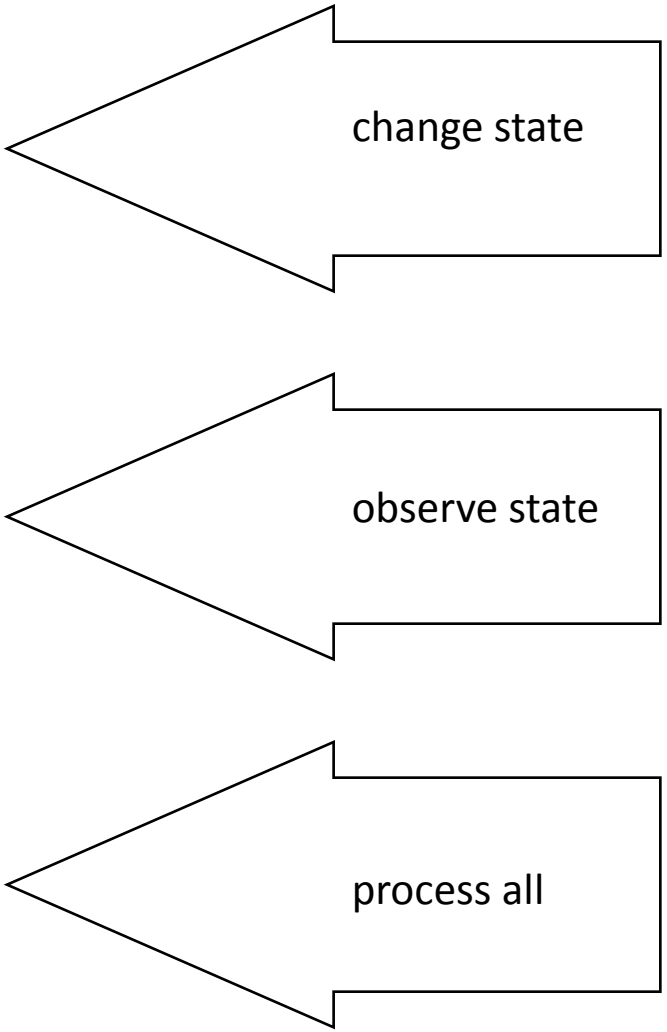
Divyakant Agrawal
Department of Computer Science
UC Santa Barbara

Lecture Plan

- Array Based implementation of sorted list
  - Time Complexity Analysis of Operations on Lists
  - Can we do better?
- Linked List implementation of Sorted List
  - Time Complexity of Operations
  - Can we use the optimization technique here?

ADT Sorted List

- **Transformers**
  - **MakeEmpty**
  - **InsertItem**
  - **DeleteItem**
- **Observers**
  - **IsFull**
  - **GetLength**
  - **RetrieveItem**
- **Iterators**
  - **ResetList**
  - **GetNextItem**



Member functions

Which member function specifications and implementations must change?

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

InsertItem for SortedList ADT (array)

- **Find proper location for the new element in the sorted list.**
- **Create space for the new element by moving down all the list elements that will follow it.**
- **Put the new element in the list.**
- **Increment length.**

Array Implementation

- **InsertItem**

```
Initialize location to position of first item
Set moreToSearch to (have not examined Info(last))
while moreToSearch
    switch (item.ComparedTo(Info(location)))
        case LESS    :  Set moreToSearch to false
        case EQUAL : // Cannot happen
        case GREATER     :
         Set location to Next(location)
         Set moreToSearch to (have not examined Info(last))
for index going from length DOWNTO location + 1
    Set Info(index ) to Info(index-1)
Set Info(location) to item
Increment length
```

Why can't `EQUAL` *happen?*

Array Implementation

```
void  SortedType :: InsertItem ( ItemType  item )
{  bool  moreToSearch ;
   int   location = 0 ;
                 // find proper location for new element
   moreToSearch = ( location < length ) ;
   while ( moreToSearch  )
   {  switch ( item.ComparedTo( info[location] ) )
        { case   LESS    : moreToSearch = false ;
                           break ;
       case  GREATER : location++ ;
                       moreToSearch = ( location < length ) ;
                  break ;
        }
  }              //  make room for new element in sorted list
   for  ( int  index = length ;  index  >  location ;  index-- )
        info [ index ] = info [ index - 1 ] ;
   info [ location ] = item ;
  length++ ;
}
```

DeleteItem for SortedList ADT (array)

- Find the location of the element to be deleted from the sorted list.
- Eliminate space occupied by the item being deleted by moving up all the list elements that follow it.
- Decrement length.

Array Implementation

- **DeleteItem**

```
Initialize location to position of first item
Set found to false
while NOT found
switch (item.ComparedTo(Info(location)))
    case GREATER : Set location to Next(location)
    case LESS    :    // Cannot happen
    case EQUAL   :    Set found to true
for index going from location +1 TO length -1
    Set Info(index - 1) to Info(index)
Decrement length
```

Why can't `LESS` happen?

Array Implementation

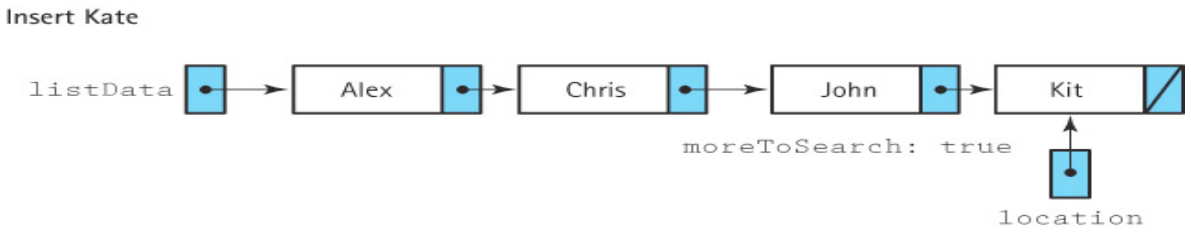
```
void  SortedType :: DeleteItem ( ItemType  item )
{
  int location = 0 ;
        //   find location of element to be deleted
   while ( item.ComparedTo ( info[location] )   !=  EQUAL )
     location++ ;
  // move up elements that follow deleted item in sorted list

   for (int  index = location + 1 ; index  <  length; index++)
     info [ index - 1 ] = info [ index ] ;
    length-- ;
}
```

Pointer-based Implementation of InsertItem

- *Let's see*

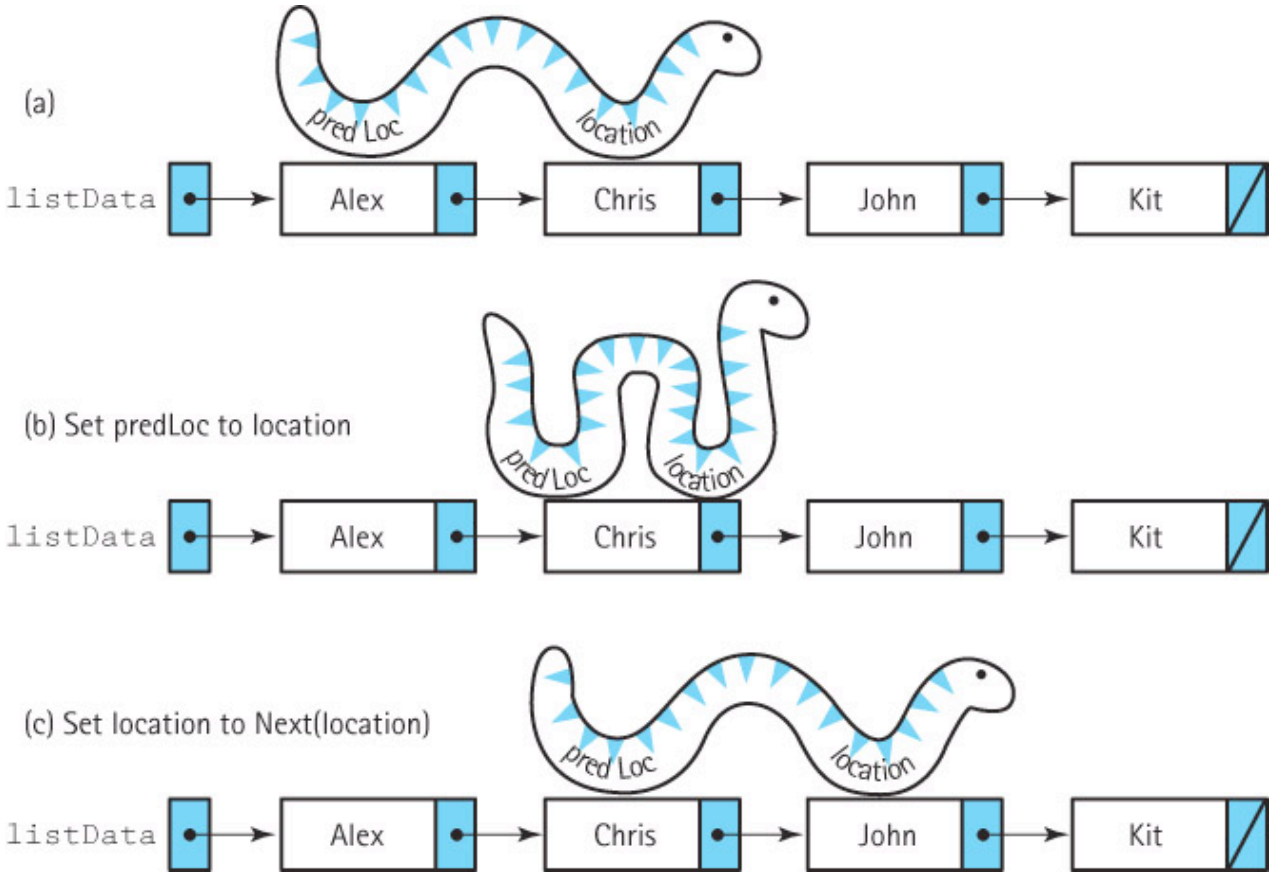
*Set location to **listData***
*Set moreToSearch to (**location != NULL**)*
while moreToSearch
*switch (item.ComparedTo(**location->info**))*
case GREATER :
*Set location to **location->next***
*Set moreToSearch to (**location != NULL**)*
case LESS : Set moreToSearch to false



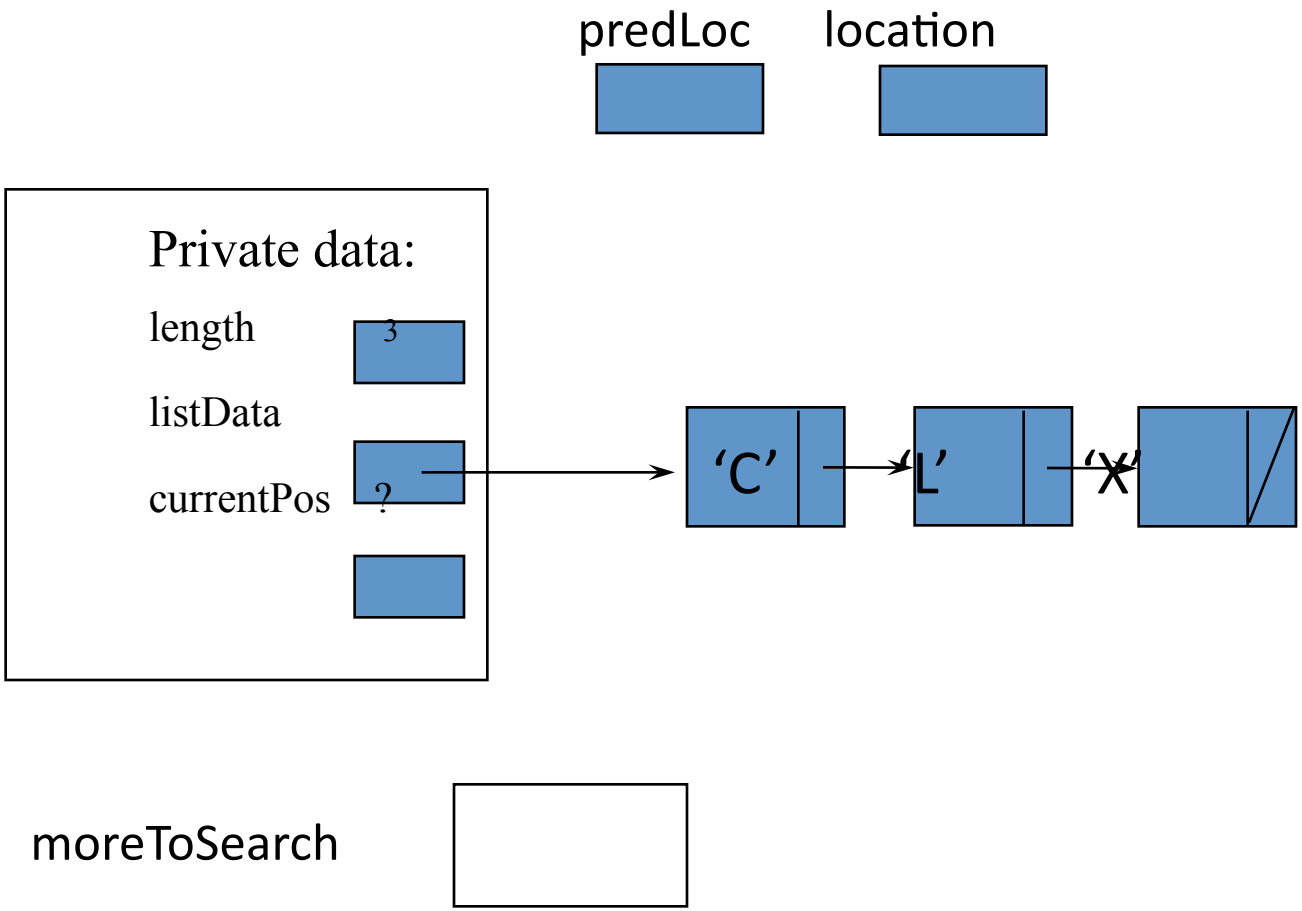
See the problem ?

Pointer-based Implementation

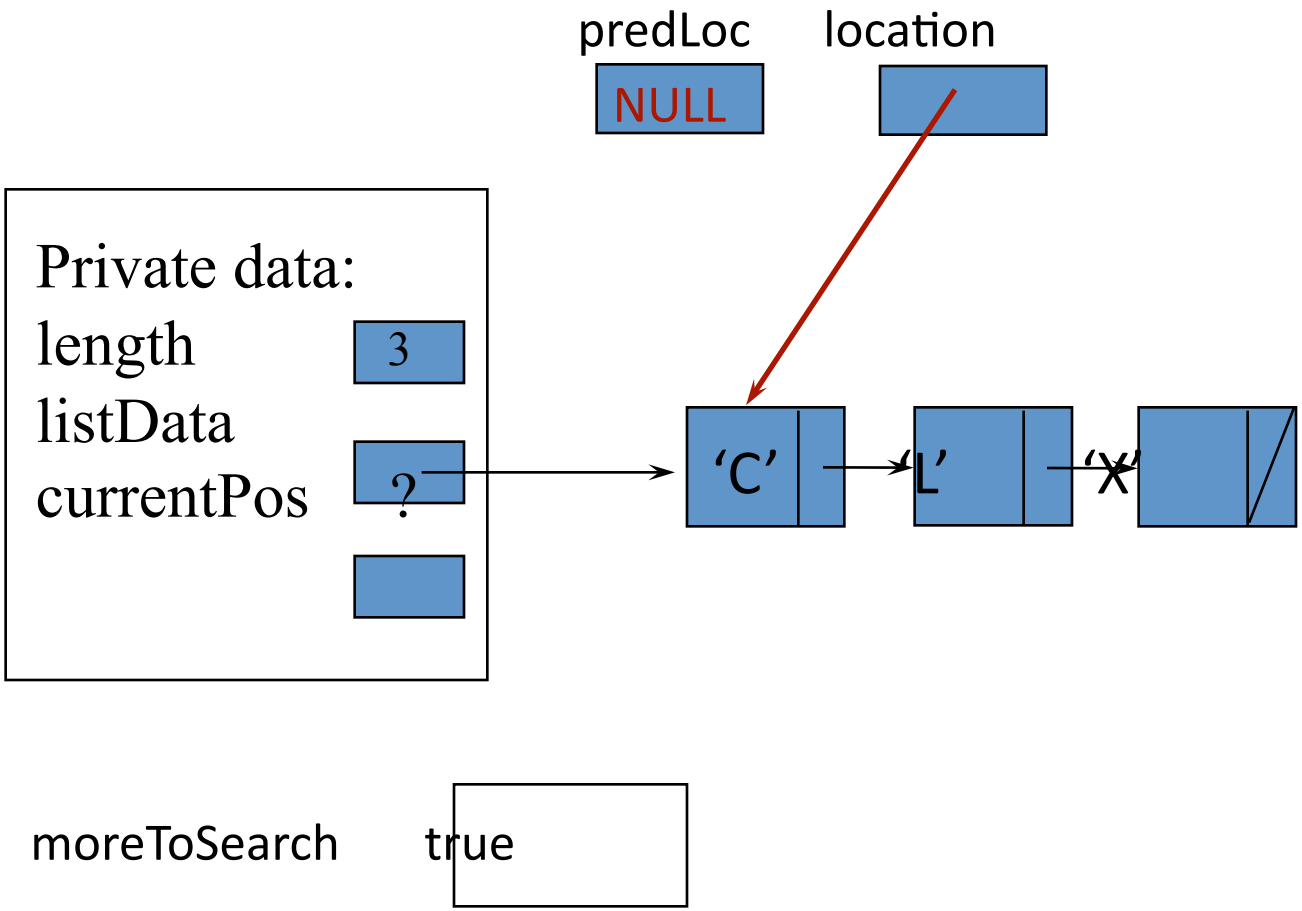
We need a trailing pointer



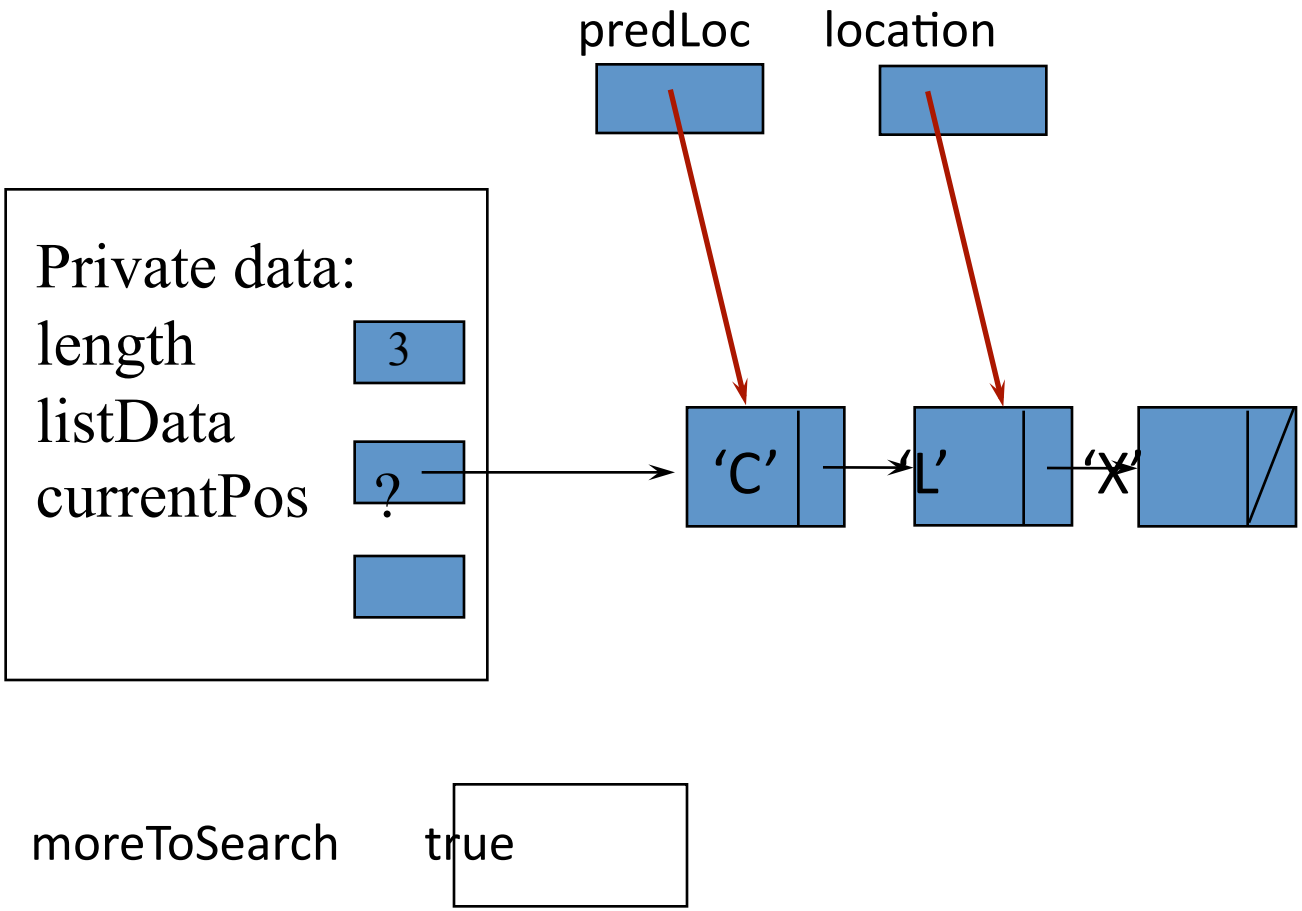
Inserting ‘S’ into a Sorted List



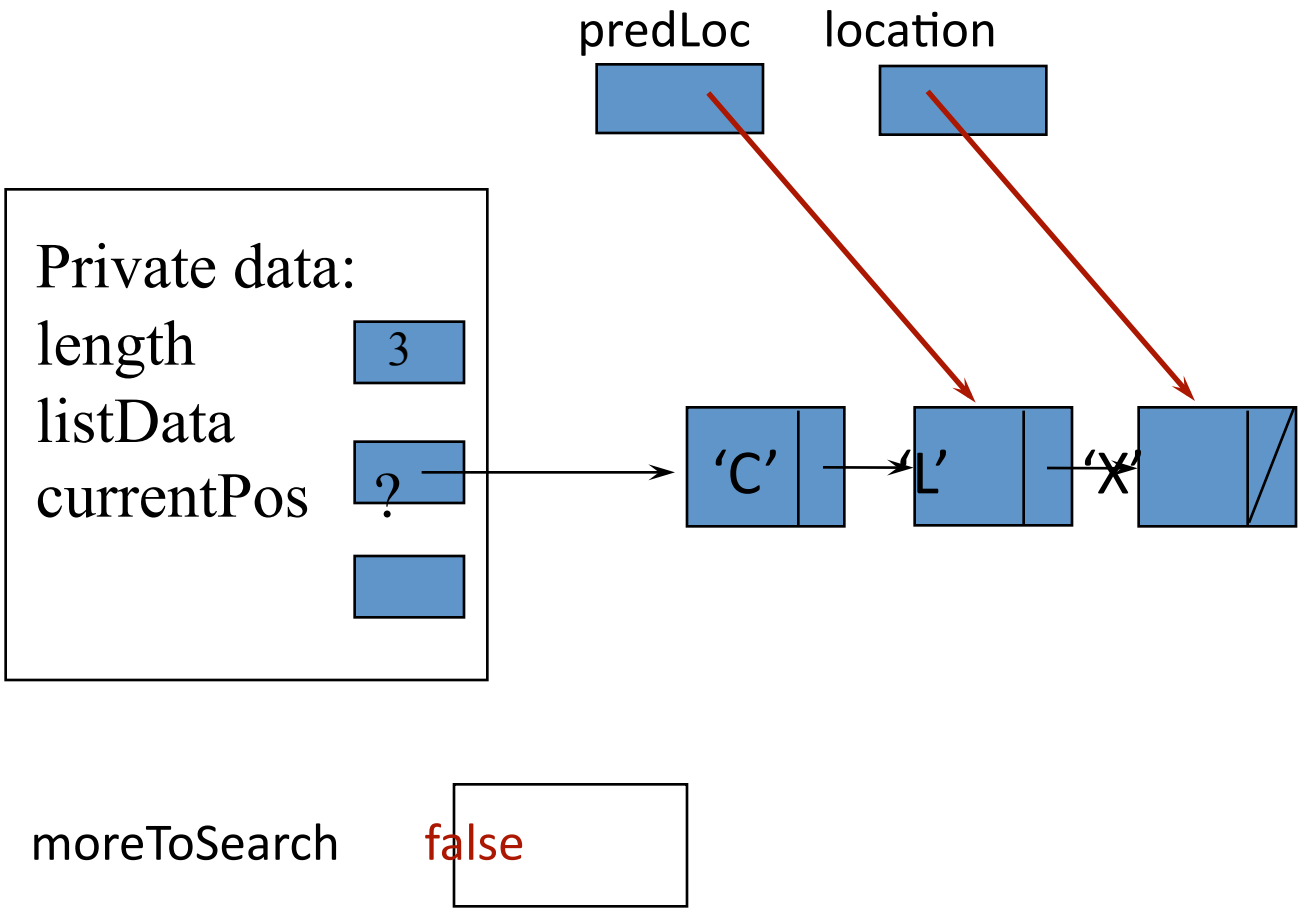
Finding proper position for ‘S’



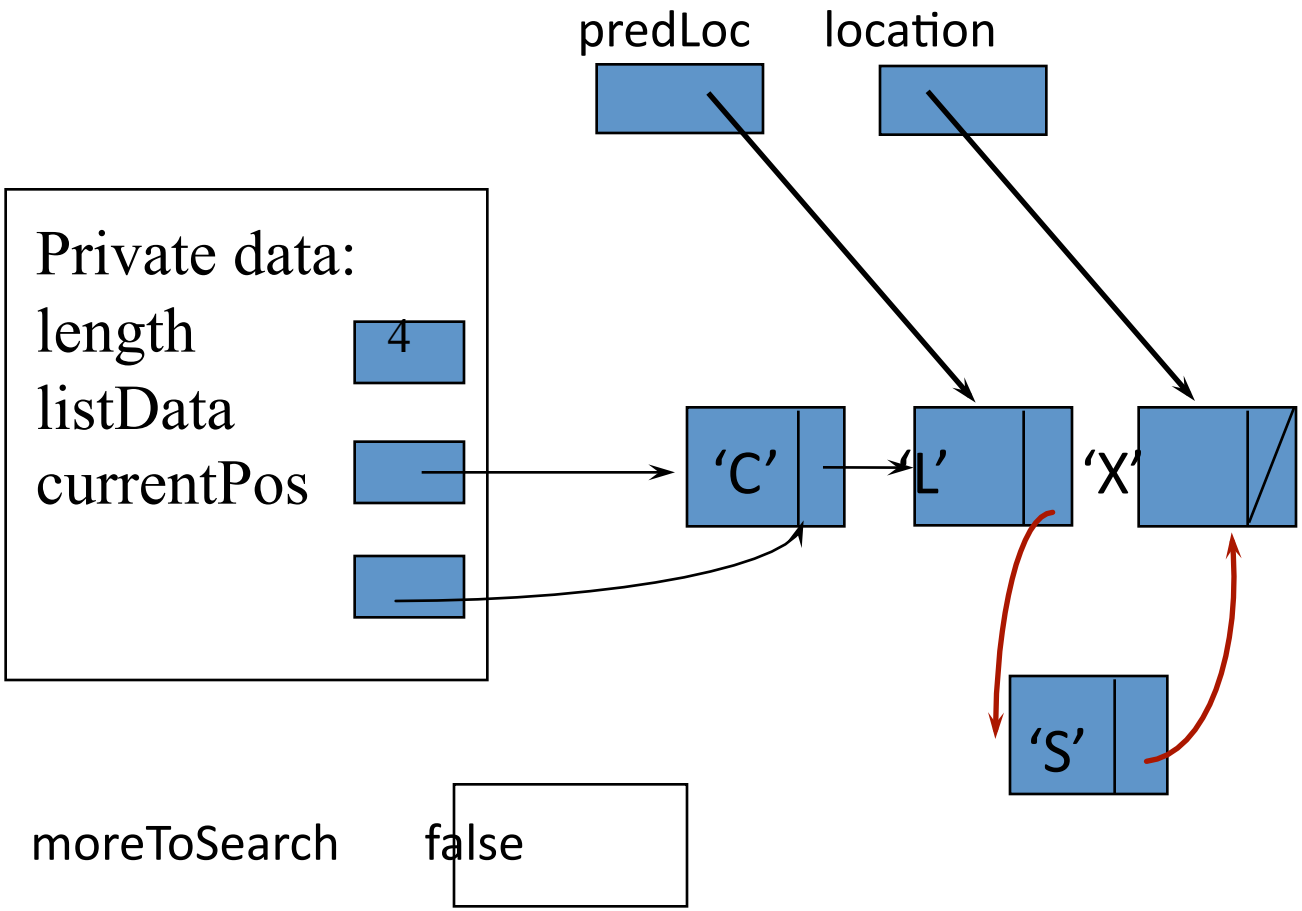
Finding proper position for ‘S’



Finding Proper Position for 'S'



Inserting ‘S’ into Proper Position



DeleteItem for SortedList ADT (pointers)

- Any changes?

How to improve searching in sorted lists?

- Binary search

Binary Seach in a Sorted List

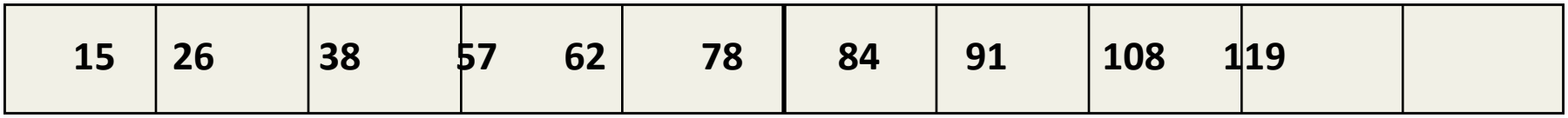
- **Examine the element in the middle of the array. Is it the sought item? If so, stop searching. Is the middle element too small? Then start looking in second half of array. Is the middle element too large? Then begin looking in first half of the array.**
- **Repeat the process in the half of the list that should be examined next.**
- **Stop when item is found, or when there is nowhere else to look.**

```
void  SortedType::RetrieveItem ( ItemType&  item,   bool&  found )
{  int midPoint ;
   int   first  =  0;
   int  last  = length - 1 ;

   found = false ;
   while (( first  <=  last )  &&  !found )
   {  midPoint  =  ( first + last ) / 2 ;   // INDEX OF MIDDLE ELEMENT
     switch ( item.ComparedTo( info [ midPoint ] ) )
        {
        case   LESS     :     . . .  // LOOK IN FIRST HALF NEXT
        case  GREATER   :     . . .  // LOOK IN SECOND HALF NEXT
        case  EQUAL     :     . . .  // ITEM HAS BEEN FOUND
        }
   }
}
```

Trace of Binary Search

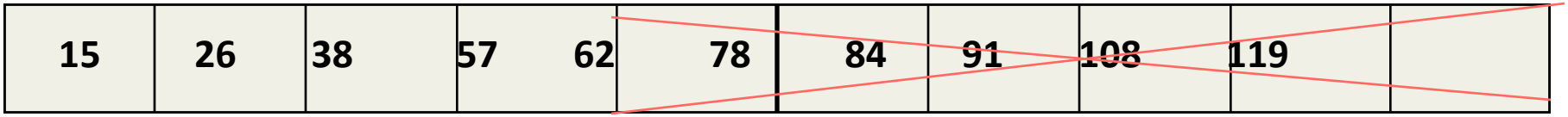
item = 45



info[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

first midPoint last

LESS → last = midPoint - 1



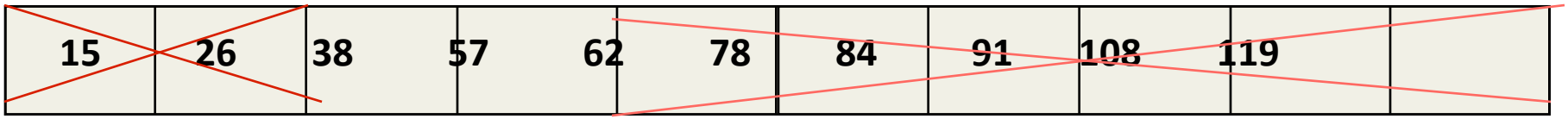
info[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

first midPoint last

GREATER → midPoint + 1

Trace continued

item = 45

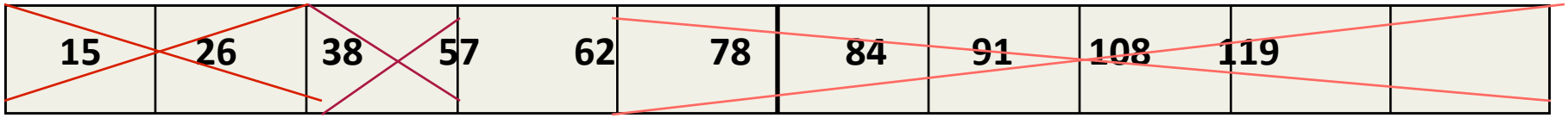


info[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

first, midPoint

last

GREATER → **= midPoint + 1**



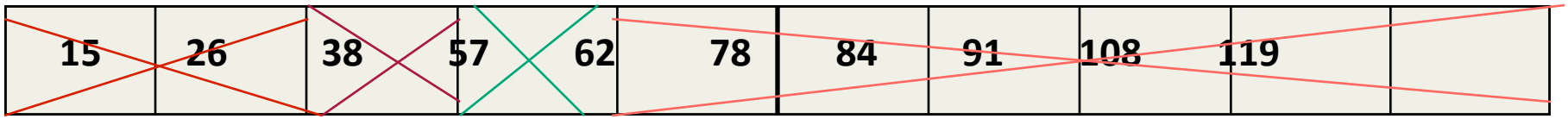
info[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

first, midPoint, last

LESS → **= midPoint - 1**

Trace concludes

item = 45



last **first**

first > last → **p terminates**

```
void  SortedType::RetrieveItem ( ItemType&  item,   bool&  found )
//  ASSUMES info ARRAY SORTED IN ASCENDING ORDER
{  int midPoint ;
   int  first  =  0;
   int  last  = length - 1 ;
   found = false ;
   //1
   while (( first  <=  last )  &&  !found )
   {  midPoint  =  ( first + last ) / 2 ;   //2
     switch ( item.ComparedTo( info [ midPoint ] ) )
        {  case   LESS       :       last = midPoint - 1 ;
                    break ;
           case  GREATER  :   first = midPoint + 1 ;
                    break ;
           case  EQUAL    :   found = true  ;
                    item = info[ midPoint ] ;
                    break ;
         }
   }
}
```

Loop Invariant

- 0 <= first <= last+1 <= length ^

found → (length > 0 ^ item = info[(first+last)/2])
^

J

¬found → (item **not in** info[0 .. first-1] ^
item **not in** info[last+1 .. length-1])

K

- At 1:: 0 <= first <= last <= length-1 ^ J ^ K
- At 2:: 0 <= first <= midpoint <= last <= length-1 ^ J ^ K
- LESS:: 0 <= first <= (midpoint-1) + 1 <= length ^ item **not in** info[midpoint .. length-1])
- GREATER:: 0 <= midpoint +1 <= last + 1 <= length ^ item **not in** info[0 .. midpoint]