
# CSE 2017 Data Structures and Lab

# Lecture #3: Data Abstraction and List

Eun Man Choi

# Different Views of Data

1. Data abstraction and encapsulation

2. Data structure

3. Abstract data type operator categories

# 1. Data Abstraction

- **Separation of a data type's logical properties from its implementation.**

| LOGICAL PROPERTIES | IMPLEMENTATION |
|---|---|
| What are the possible values? | How can this be done in C++? |
| What operations will be needed? | How can data types be used? |

# Data Encapsulation

- is the separation of the representation of data from the applications that use the data at a logical level; a programming language feature that enforces information hiding.

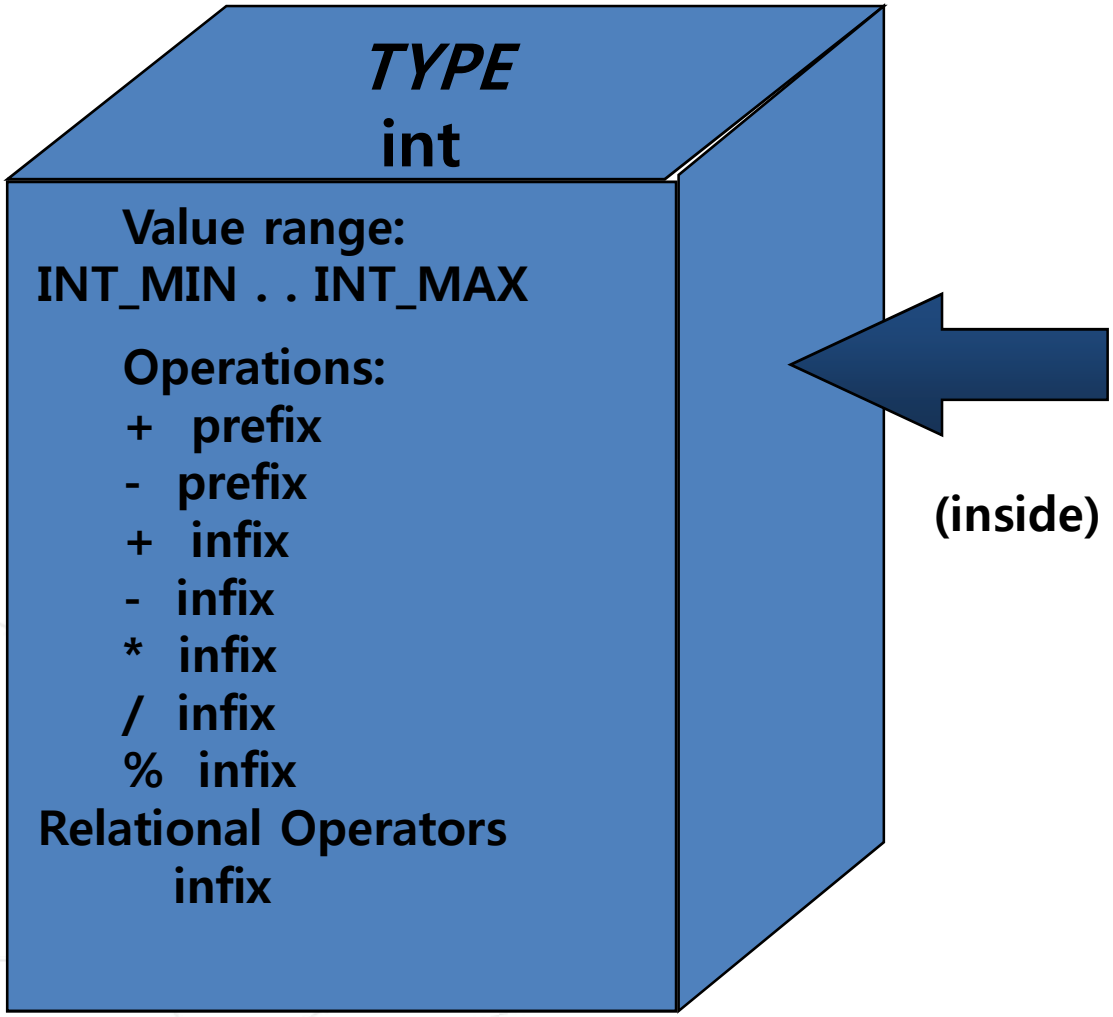
**APPLICATION**

```
int  y;

y = 25;
```

**REPRESENTATION**

```
0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1
```

# Encapsulated C++ Data Type int




(inside)

Representation of

int

as 16 bits two's complement

+

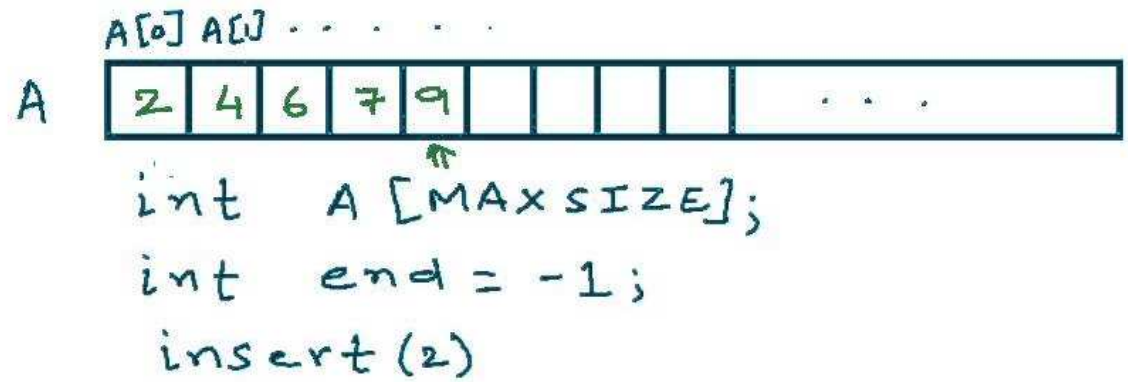
Implementation of Operations

# Abstract Data Type (ADT)

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

List as abstract data type

List
- empty list has size 0
- insert
- remove
- Count
- Read/modify element at a position
- Specify data-type



# 2. Data Structures

**Defined by**

- **the logical arrangement of data elements**
- **the set of operations we need to access the elements**

# Data from 3 different levels

- *Application (or user) level:* modeling real-life data in a specific context.

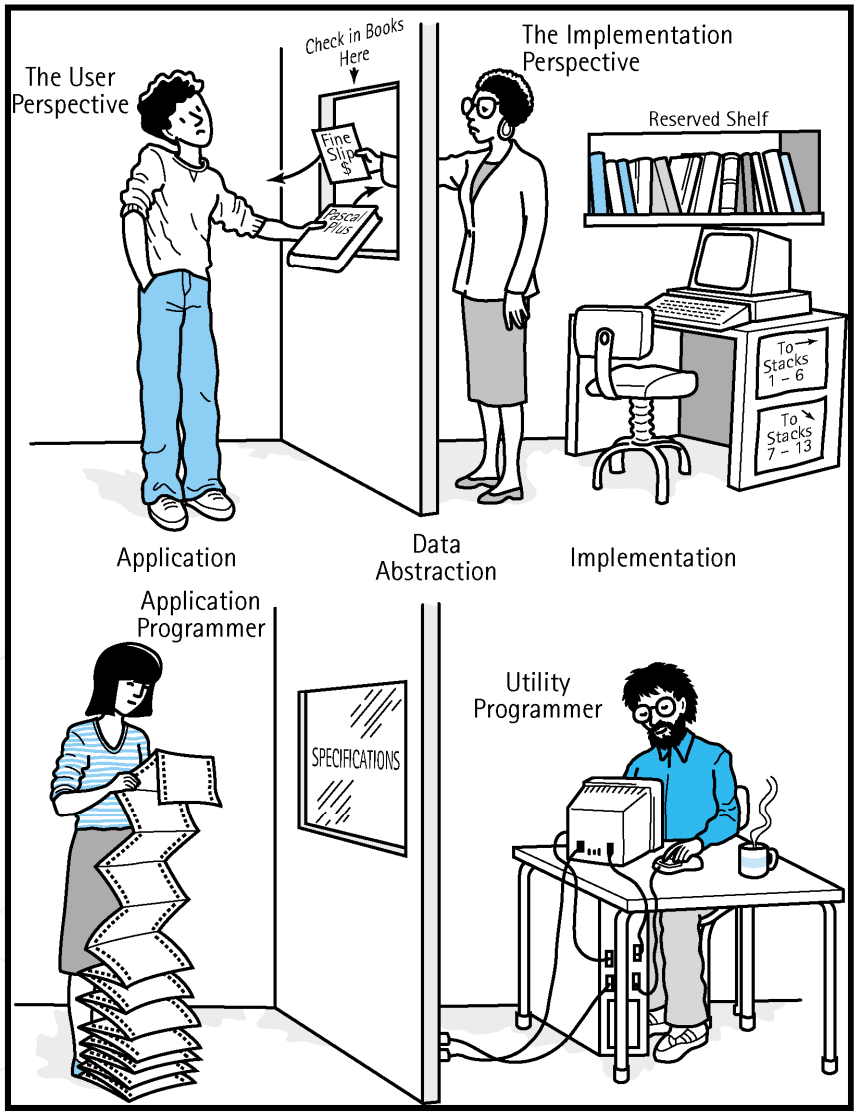
- *Logical (or ADT) level:* abstract view of the domain and operations.

WHAT

- *Implementation level:* specific representation of the structure to hold the data items, and the coding for operations.

HOW

# Logical Level



# Viewing a library from 3 different levels

- *Application (or user) level:* Library of Congress, or Baltimore County Public Library.

- *Logical (or ADT) level:* domain is a collection of books; operations include: check book out, check book in, pay fine, reserve a book.

- *Implementation level:* representation of the structure to hold the "books", and the coding for operations.

# 3. ADT Operator Categories

- **Constructor** -- creates a new instance (object) of an ADT.

- **Transformer** -- changes the state of one or more of the data values of an instance.

- **Observer** -- allows us to observe the state of one or more of the data values without changing them.

- **Iterator** -- allows us to process all the components in a data structure sequentially.

# Lists

**Lists in every-day life:**

- **Grocery list**
- **Laundry list**
- **To-do list**
- **Invitation list**
- **…**

# CSE2017 Class List

| Name | SS | E-mail | Quiz1 | Lab1 | Lab2 | … | Midterm | … |
|---|---|---|---|---|---|---|---|---|
| XXX | 999-99-9999 | xx@xxx | 20 | 10 | 10 | | | |
| YYY | 000-00-0000 | yy@yyy | 25 | 9 | 10 | | | |
| ZZZ | 111-11-1111 | zz@zzz | 24 | 10 | 8 | | | |
| … | | | | | | | | |
| … | | | | | | | | |
| … | | | | | | | | |
| UUU | 222-22-222 | uu@uuu | 23 | 9 | 10 | | | |

# 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.

# Sorted and Unsorted Lists

## UNSORTED LIST

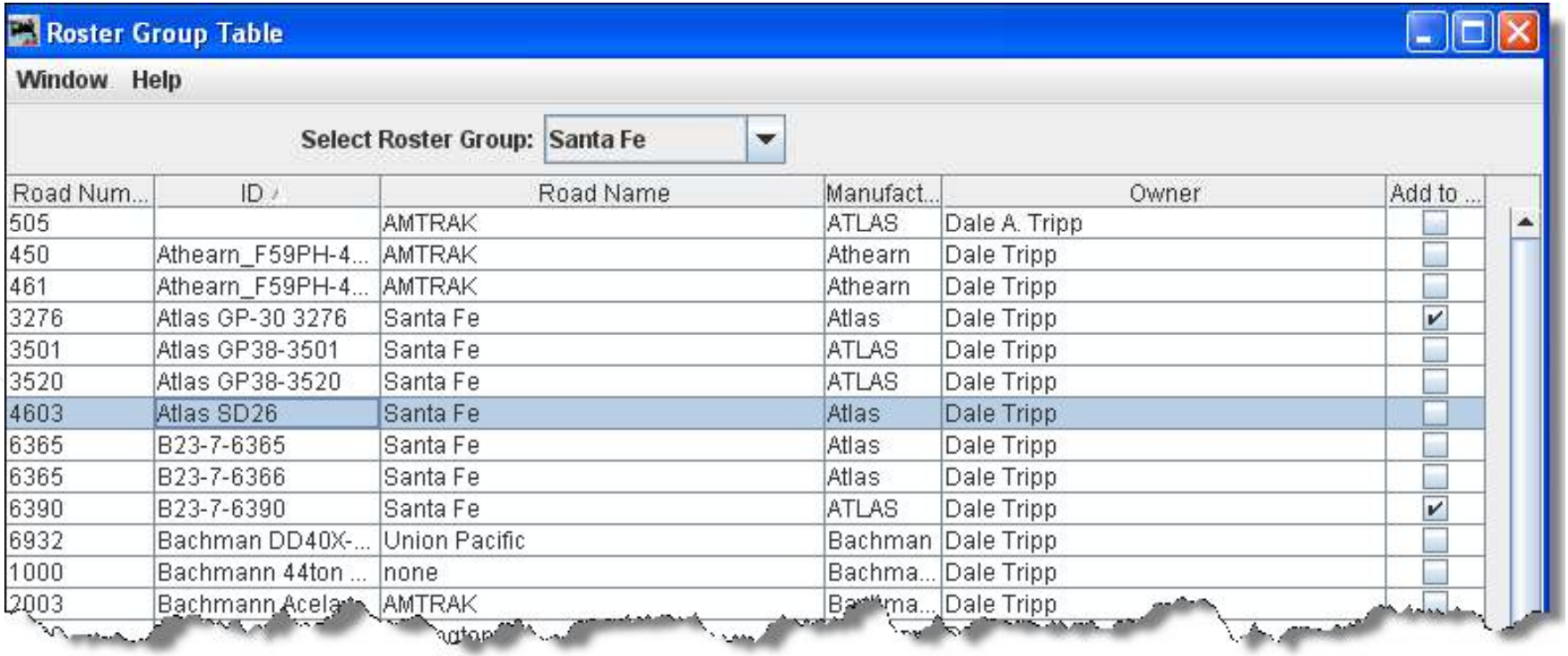
Elements are placed into the list in no particular order.

## SORTED LIST

List elements are in an order that is sorted in some way -- either numerically or alphabetically by the elements themselves, or by a component of the element (called a KEY member) .

# Key:

- **A member of a record (struct or class) whose value is used to determine the logical and/or physical order of the items in a list**



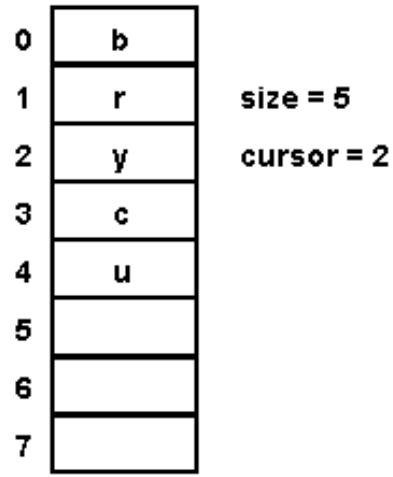
# ADT Unsorted List

- **Application Level**


- **Logical Level**

**Add a name on the list**
**Delete a name on the list**
**Print out all names**
**Retrieve a name**
**Sort names**

- **Implementation Level**



# ADT Unsorted List Operations

## Transformers

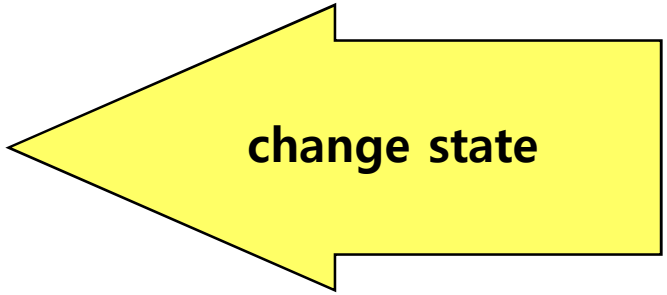
- MakeEmpty
- InsertItem
- DeleteItem

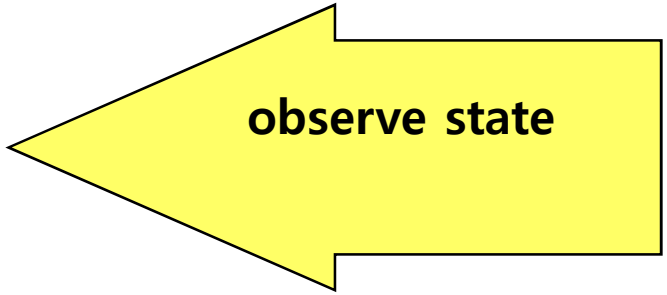
## Observers

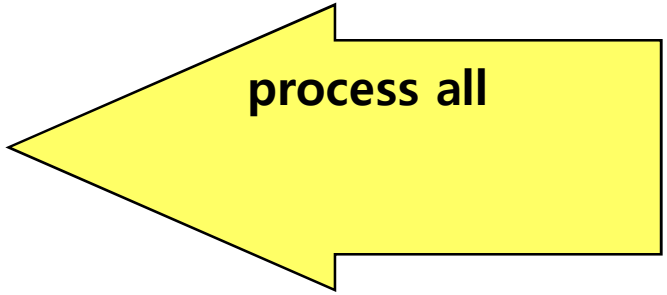
- IsFull
- LengthIs
- RetrieveItem

## Iterators

- ResetList
- GetNextItem



# What is a Generic Data Type?

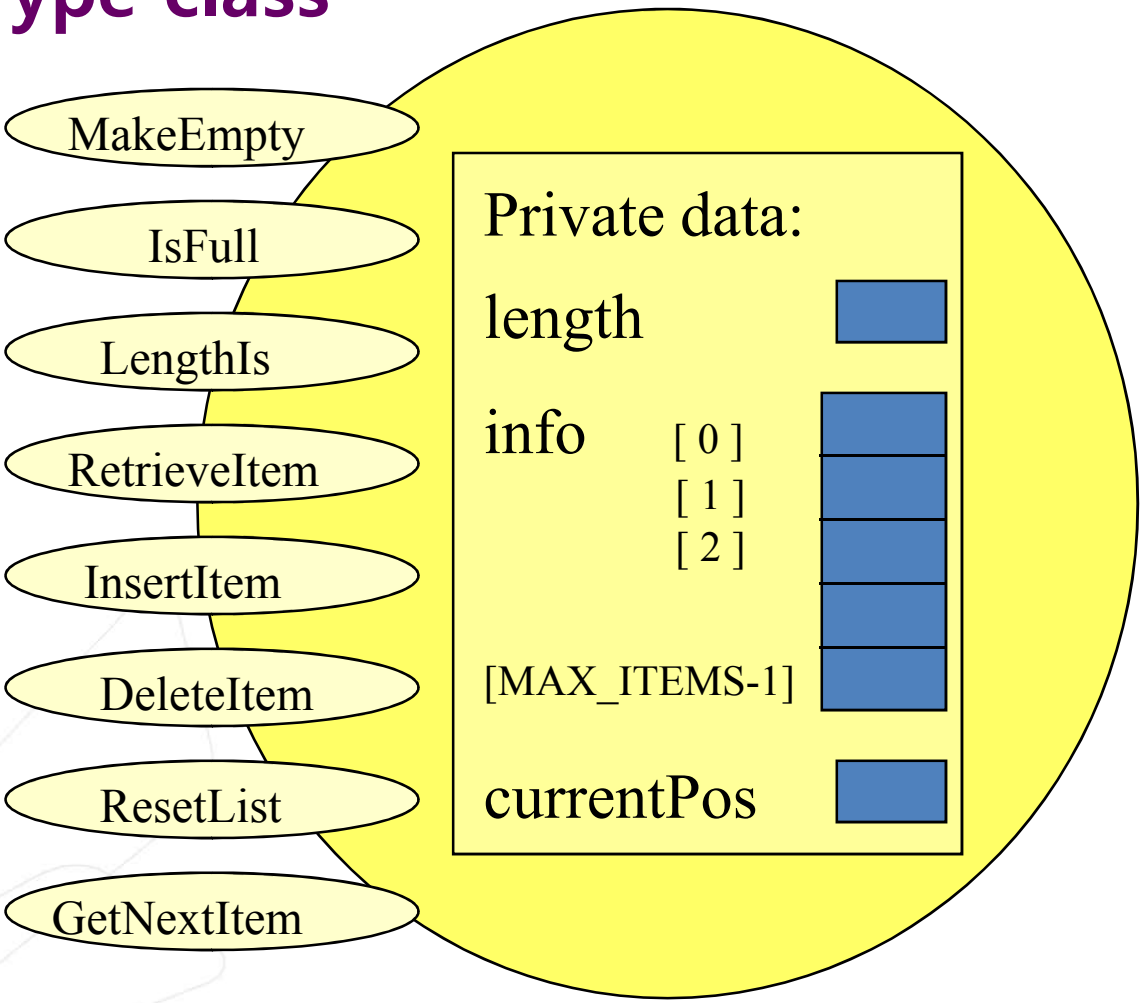
- A **generic data type** is a type for which the operations are defined but the types of the items being manipulated are not defined.

- One way to simulate such a type for our UnsortedList ADT is via a user-defined class `ItemType` with member function `ComparedTo` having enumerated type value `LESS`, `GREATER`, or `EQUAL`.

```
// SPECIFICATION FILE                    ( unsorted.h )
#include "ItemType.h"

class UnsortedType {          // declares a class data type
public :                      //  8 public member functions

  void      MakeEmpty ( ) ;
  bool      IsFull ( ) const ;
  int       LengthIs ( ) const ;      // returns length of list
  void      RetrieveItem ( ItemType& item, bool& found ) ;
  void      InsertItem ( ItemType item ) ;
  void      DeleteItem ( ItemType item ) ;
  void      ResetList ( );
  void      GetNextItem ( ItemType& item ) ;

private :                                  //  3 private data members

  int           length ;
  ItemType      info[MAX_ITEMS] ;
  int           currentPos ;
} ;
```

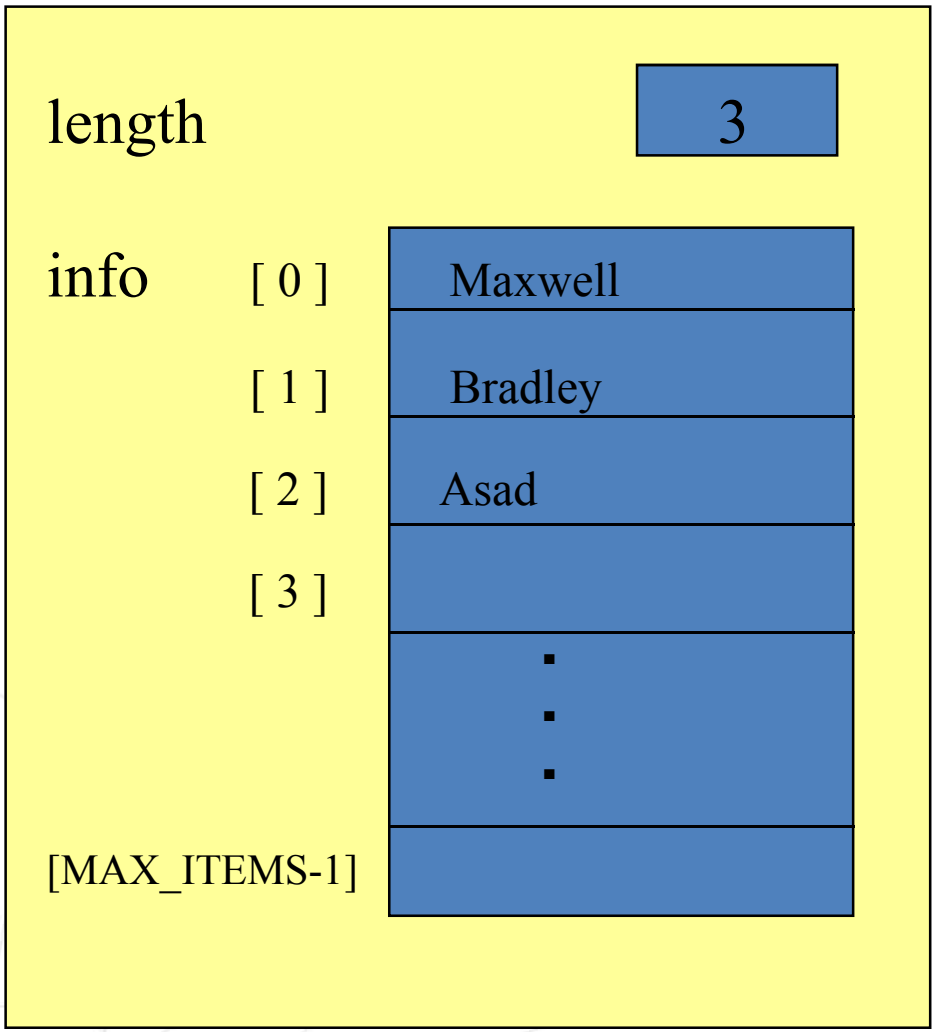
# Class Interface Diagram

## UnsortedType class



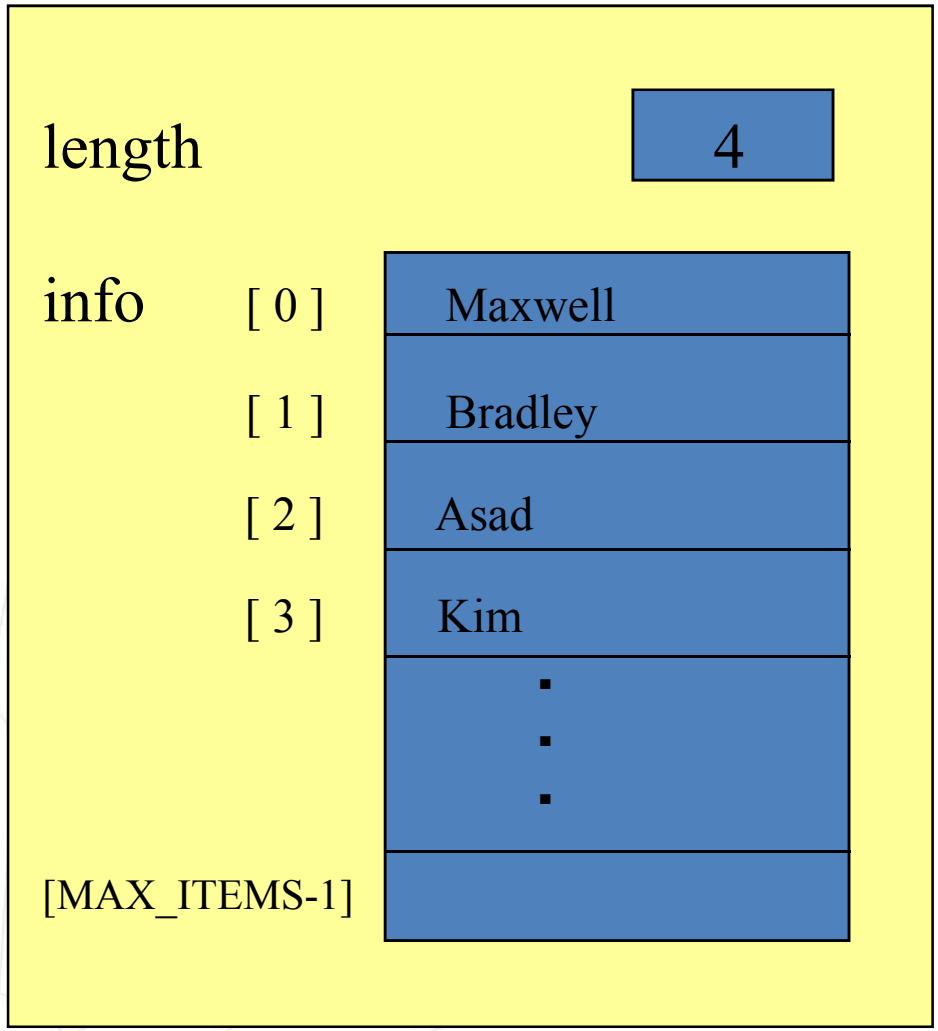
```
// IMPLEMENTATION FILE   ARRAY-BASED LIST  ( unsorted.cpp )
#include "itemtype.h"

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


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 Kim into an Unsorted List




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

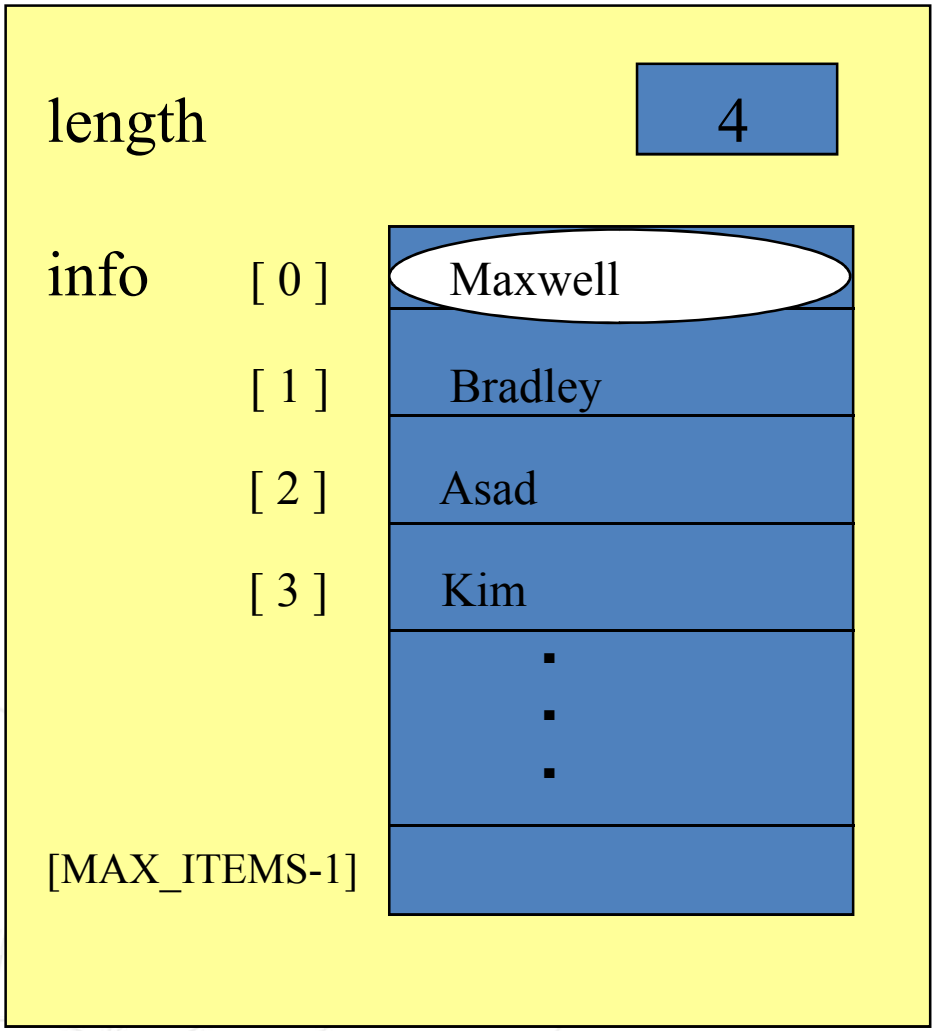
# After Inserting Kim into an Unsorted List



```
void UnsortedType::LengthIs ( )  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 ) ;
}
```

```
void  UnsortedType::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 ;
    int     location = 0 ;

    found = false ;
    moreToSearch = ( location < length ) ;
    while ( moreToSearch  &&  !found ) {
        switch ( item.ComparedTo( info[location] ) ) {
            case   LESS        :
            case  GREATER    :  location++ ;
                                  moreToSearch = ( location < length ) ;
            case  EQUAL      :  found = true  ;
                                  item = info[ location ] ;
                                  break ;
        }
    }
}
```

# Retrieving Park from an Unsorted List

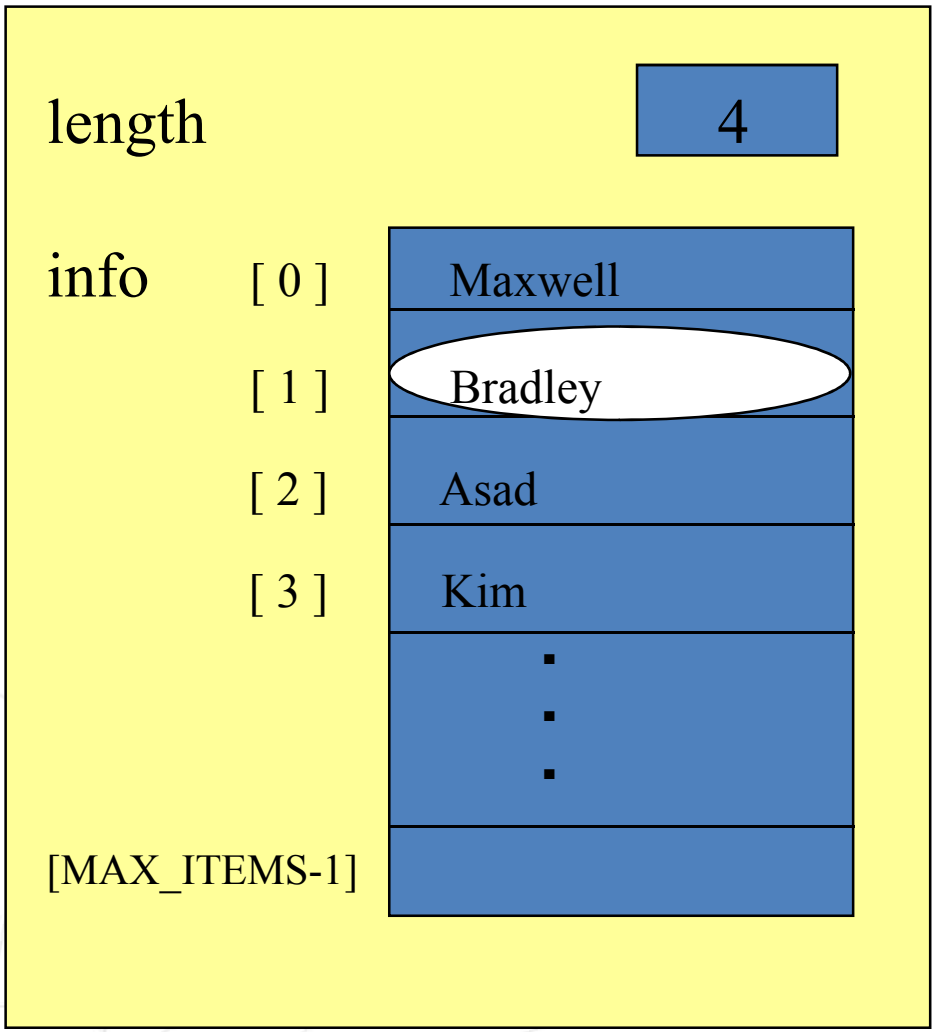



**moreToSearch:** **true**

**found:** **false**

**location:** **0**

# Retrieving Park from an Unsorted List

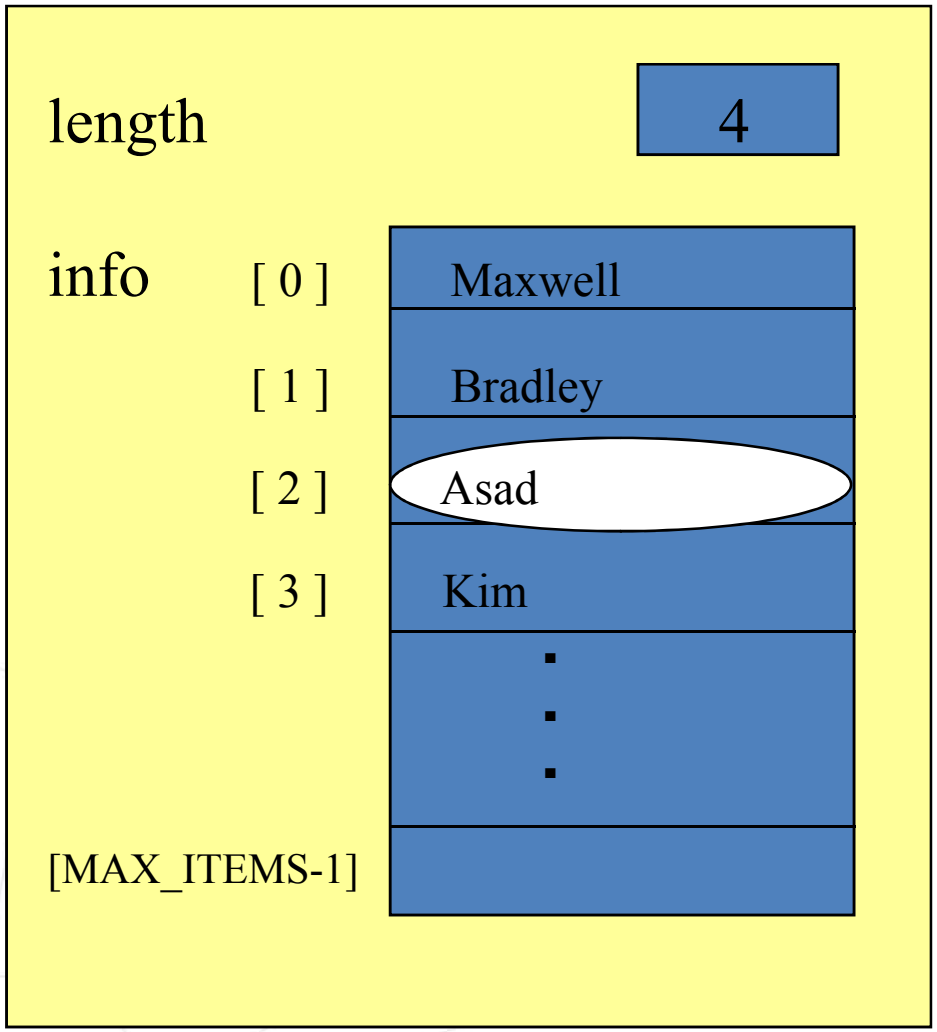



**moreToSearch:** true

**found:** false

**location:** 1

# Retrieving Park from an Unsorted List

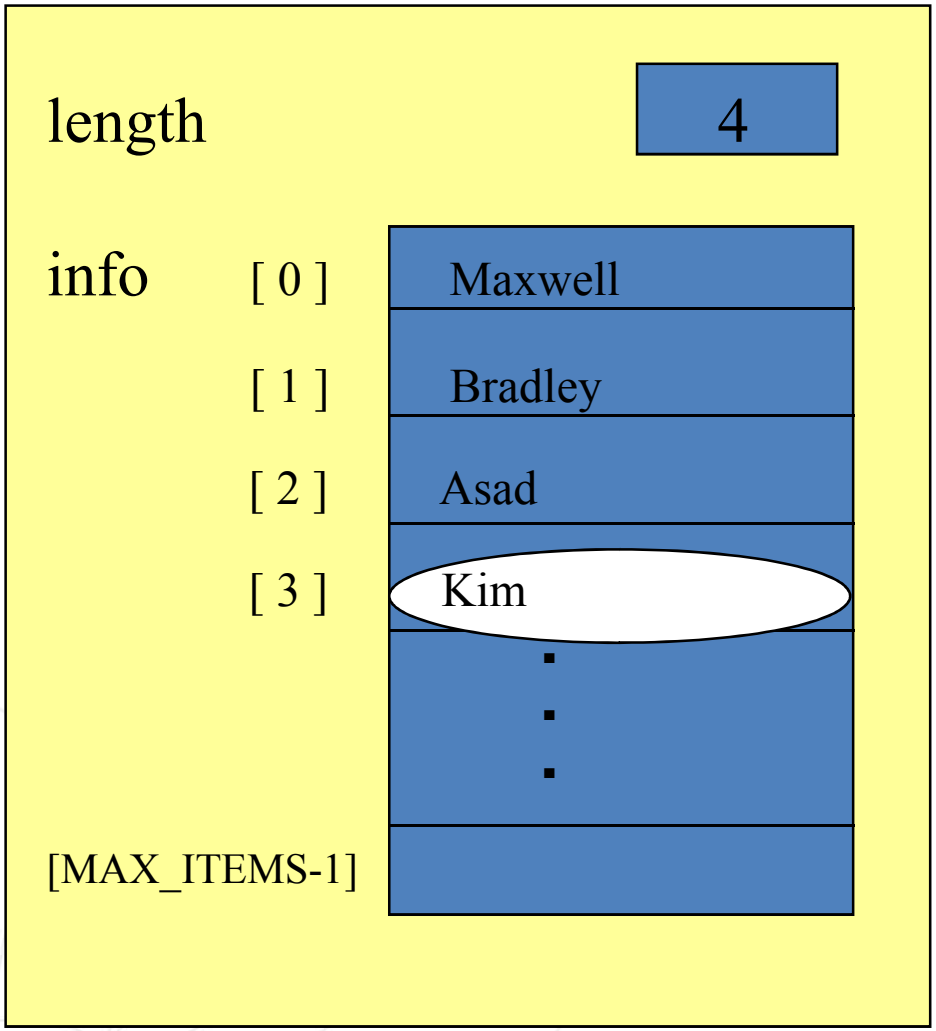



**moreToSearch:** true

**found:** false

**location:** 2

# Retrieving Park from an Unsorted List

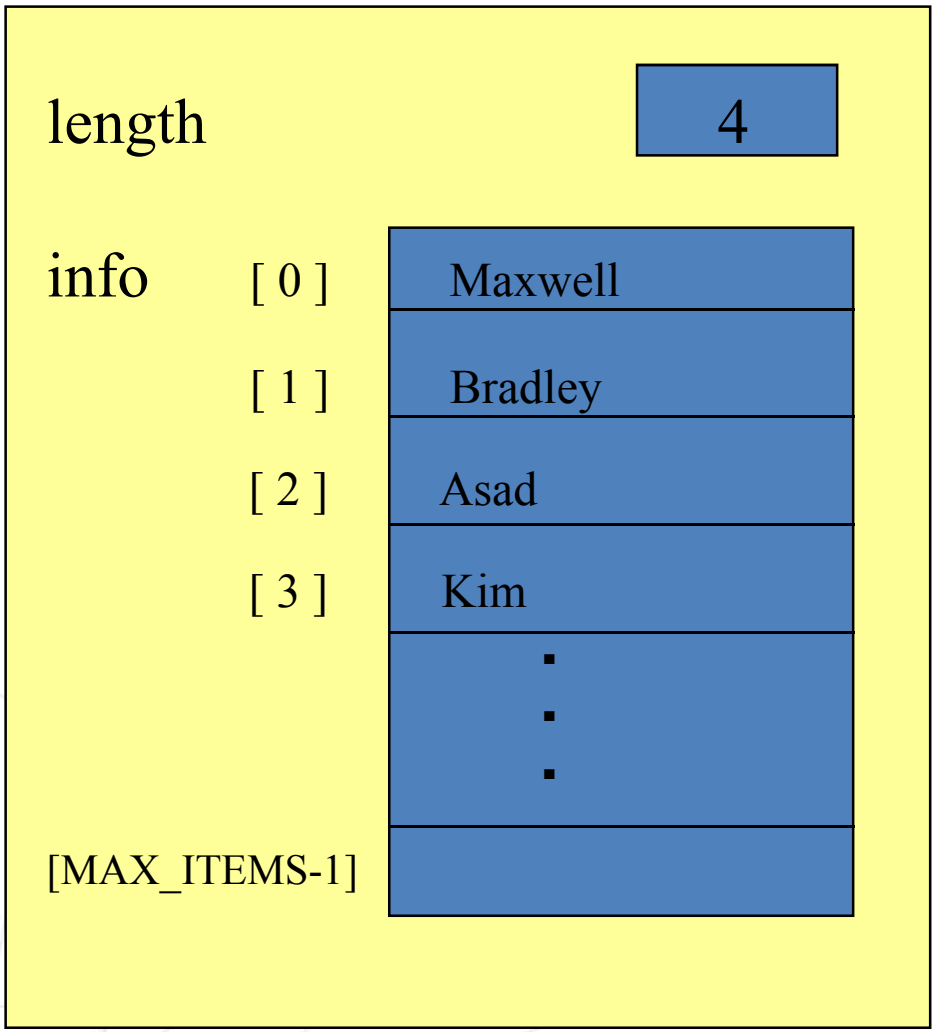



**moreToSearch:** true

**found:** false

**location:** 3

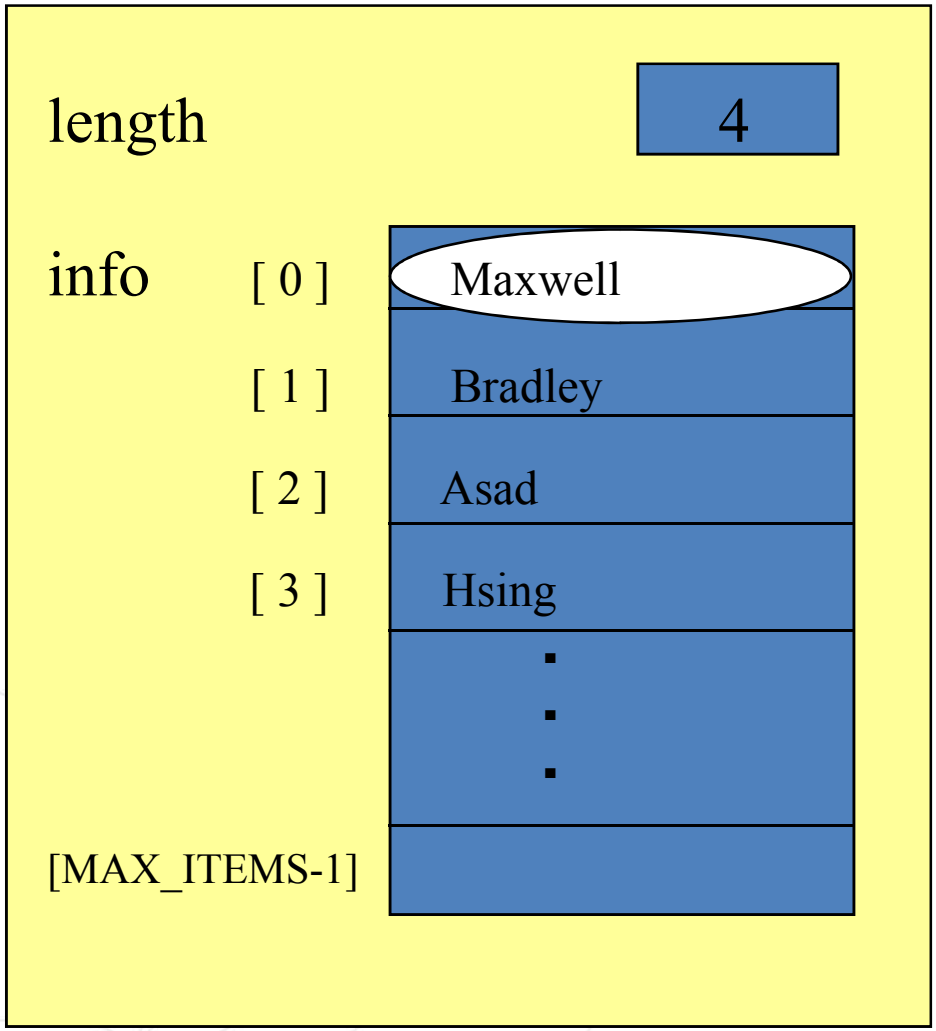
# Retrieving Park from an Unsorted List




| | |
|---|---|
| **moreToSearch:** | **false** |
| **found:** | **false** |
| **location:** | **4** |

```
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-- ;
}
```

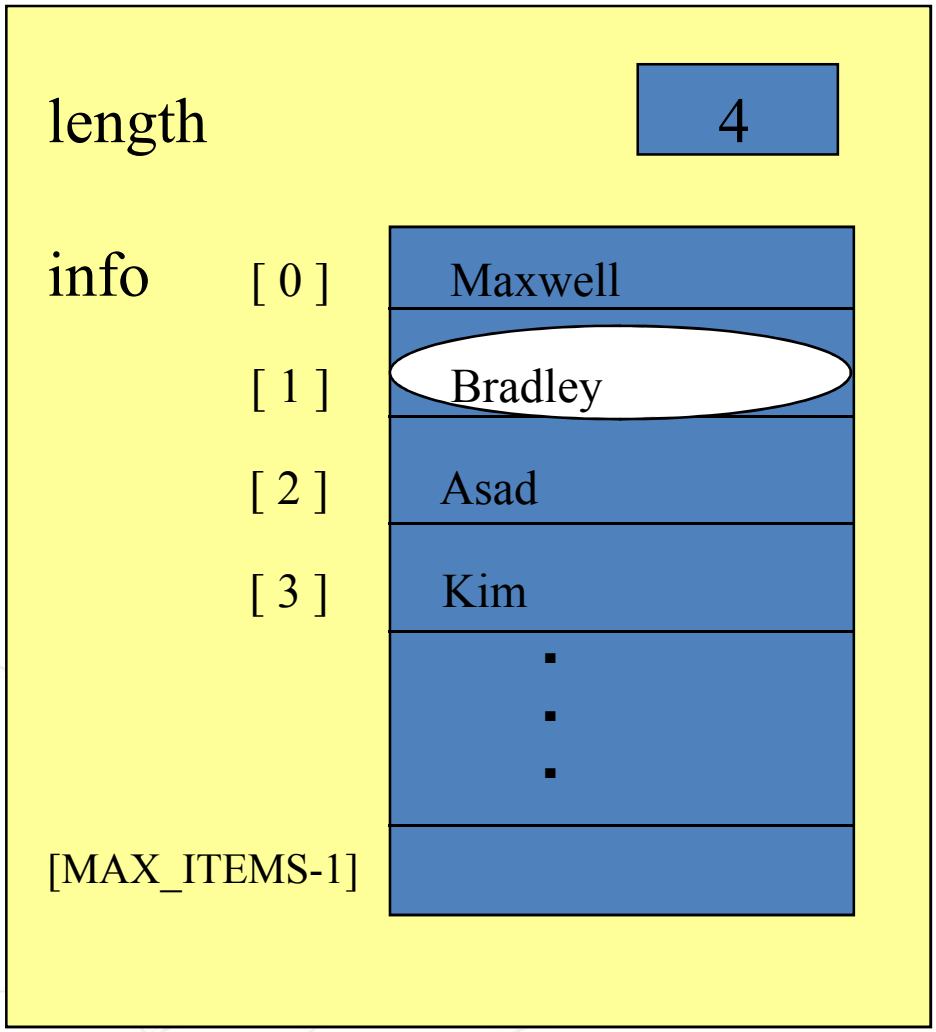
# Deleting Bradley from an Unsorted List




**location:** **0**

**Key Bradley has not been matched.**

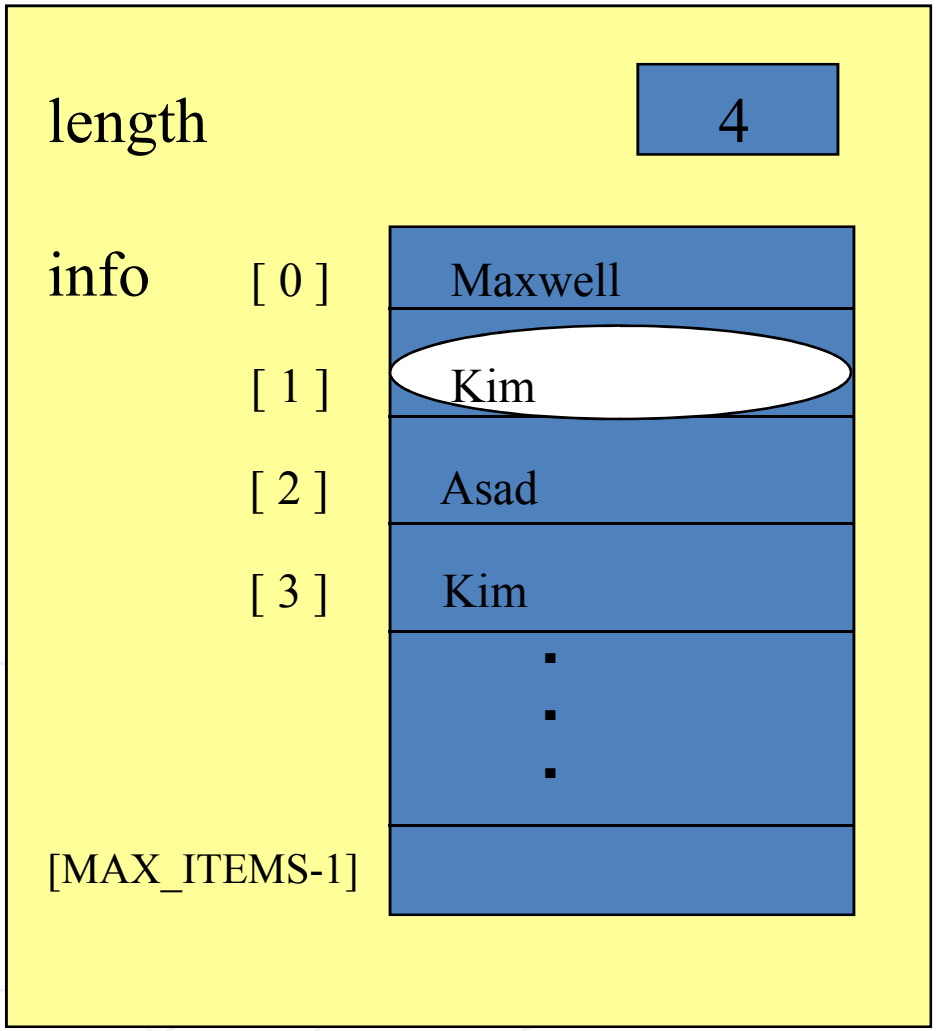
# Deleting Bradley from an Unsorted List




**location:** **1**

**Key Bradley has been matched.**

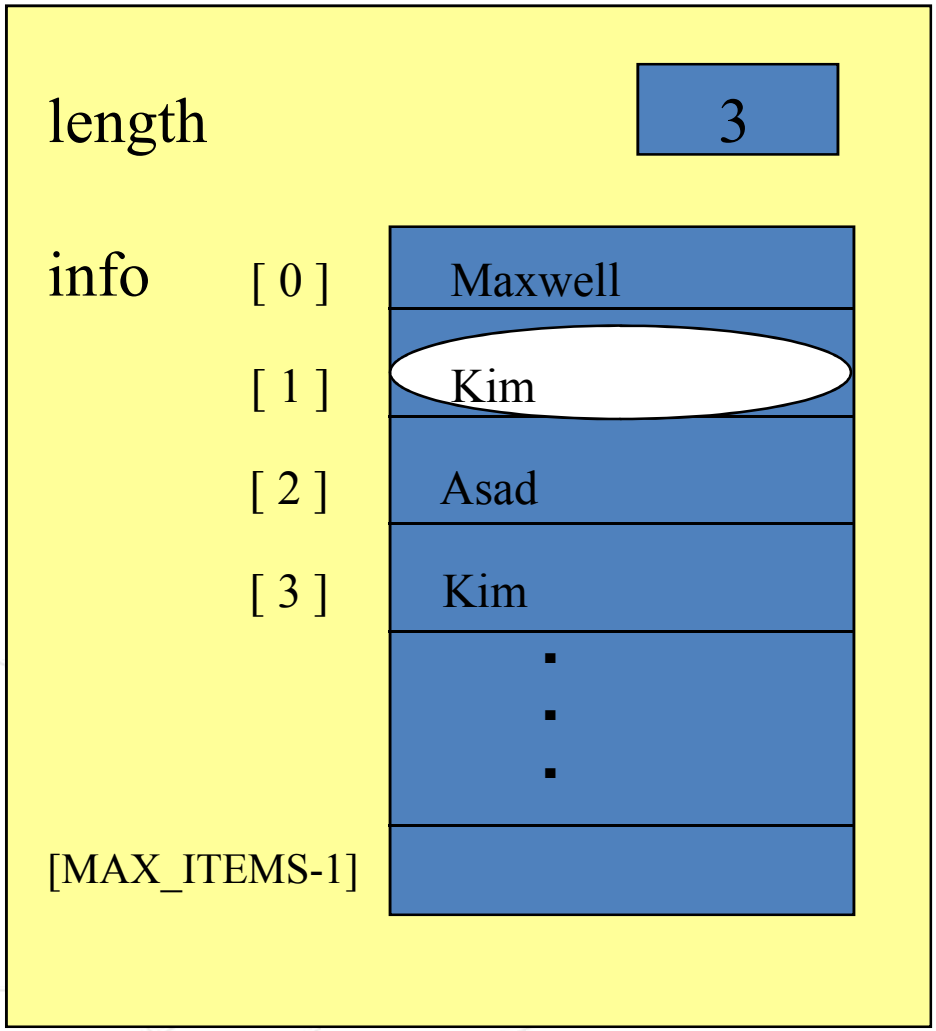
# Deleting Bradley from an Unsorted List




**location:** **1**

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

# Deleting Bradley from an Unsorted List




**location:** **1**

**Decremented length.**

```
void UnsortedType::ResetList ( ) {
//  Pre: List has been inititalized.
// Post:Current position is prior to first element in list.
  currentPos  =  -1 ;
}

void  UnsortedType::GetNextItem ( ItemType&  item )  {
//  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++  ;
  item = info [currentPos] ;
}
```

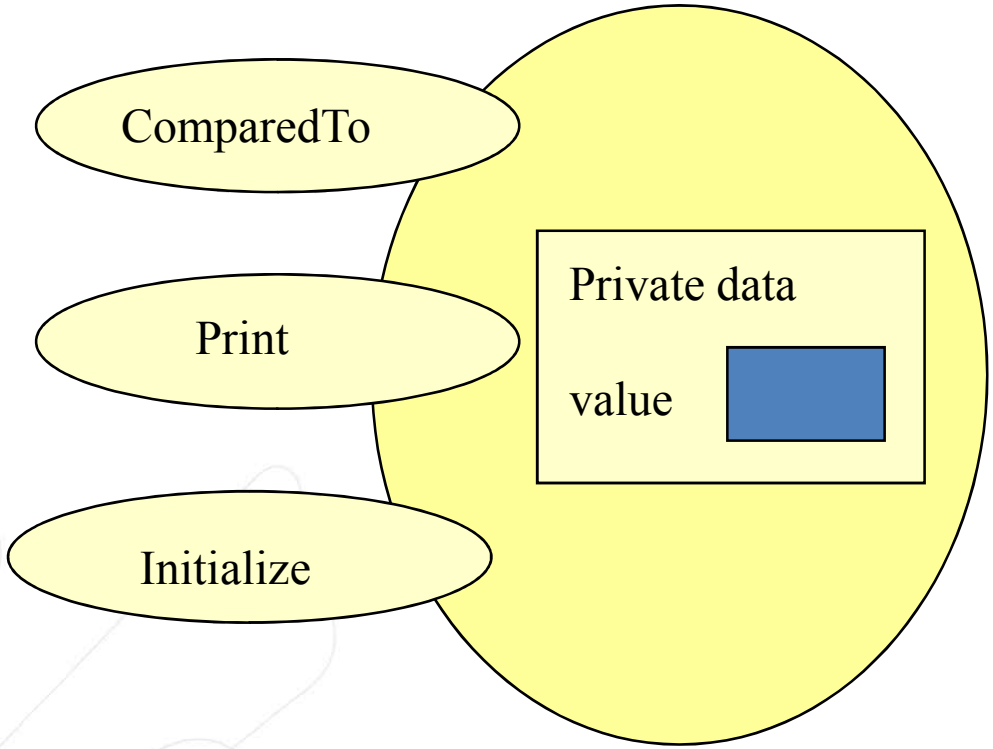
# Specifying class ItemType

```
//  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
} ;
```

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

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

RelationType ComparedTo ( ItemType otherItem ) const {
  if ( value < otherItem.value )
        return LESS ;
  else if ( value > otherItem.value )
        return GREATER ;
  else return EQUAL ;
}

void Print ( ) const {
  cout << value << endl ;
}

void Initialize ( int number ) {
  value = number ;
}
```

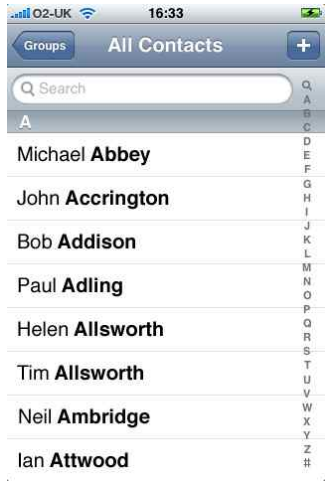
# ItemType Class Interface Diagram

- **class ItemType**



# ADT Sorted List

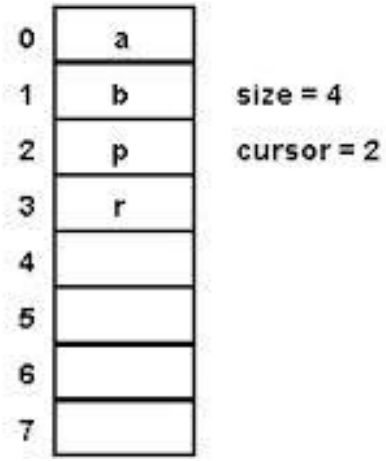
- **Application Level**




- **Logical Level**

**Add a name on the list**
**Delete a name on the list**
**Print out all names**
**Retrieve a name**

- **Implementation Level**

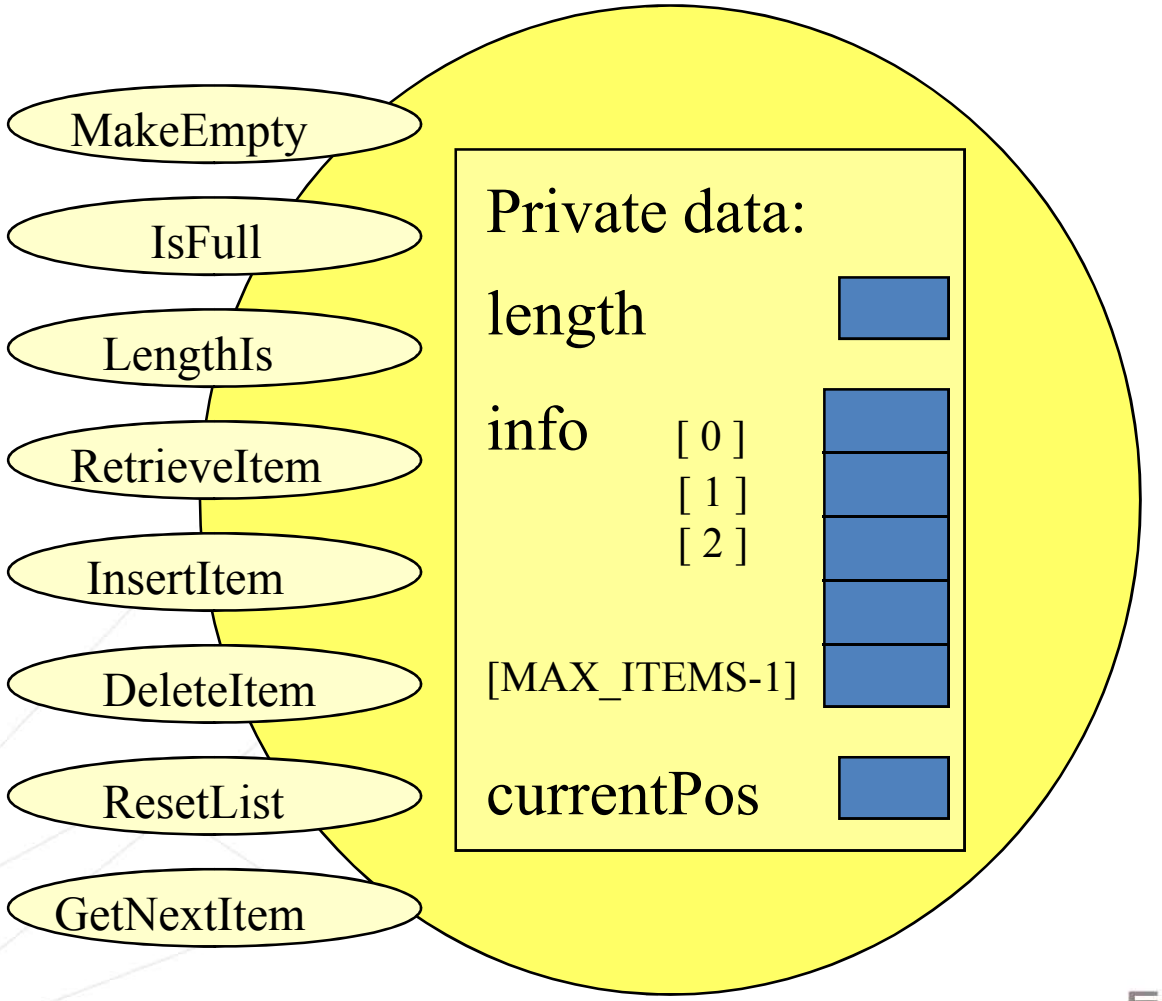





동국대학교
dongguk university

# SortedType Class Interface Diagram

- **SortedType class**



# Member functions

- Which member function specifications and implementations must change to ensure that any instance of the Sorted List ADT remains sorted at all times?
  - InsertItem
  - DeleteItem

# InsertItem algorithm for SortedList ADT

- 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.

# Implementing SortedType member function InsertItem

```
// IMPLEMENTATION FILE                    (sorted.cpp)

#include "itemtype.h"        // also must appear in client code

void  SortedType :: InsertItem ( ItemType  item ) {
//  Pre:  List has been initialized. List is not full.  item is not in list.
//        List is sorted by key member using function ComparedTo.
// Post:  item is in the list.  List is still sorted.
                        .
                        .
                        .

}
```

```
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 algorithm for SortedList ADT

- 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.

# Implementing SortedType member function DeleteItem

```
// IMPLEMENTATION FILE     continued                (sorted.cpp)

void  SortedType :: DeleteItem ( ItemType  item )
//  Pre:  List has been initialized. Key member of item is initialized.
//        Exactly one element in list has a key matching item's key.
//        List is sorted by key member using function ComparedTo.
// Post:  No item in list has key matching item's key.
//         List is still sorted.
{
                    .
                    .
                    .

}
```

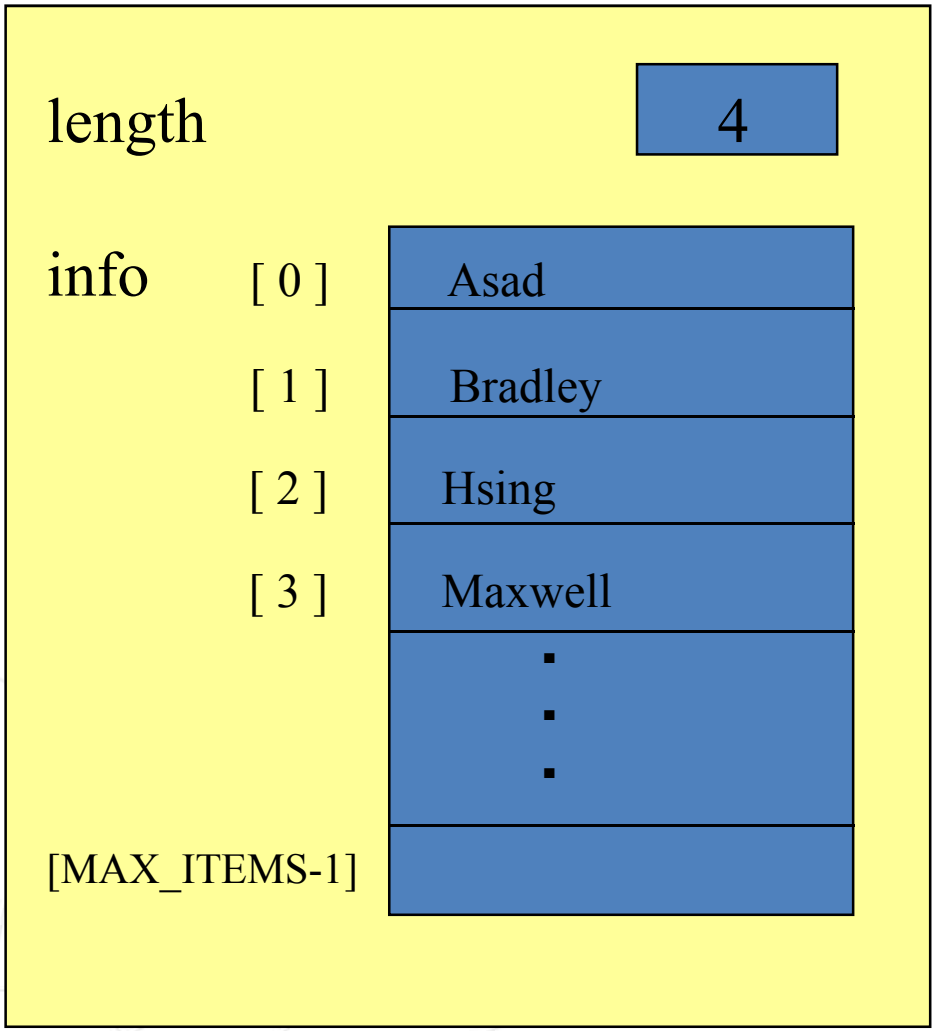
```
void  SortedType :: DeleteItem ( ItemType  item ) {
    int     location = 0 ;
  //   find location of element to be deleted

  while ( item.ComparedTo ( info[location] )  !=  EQUAL )
        location++ ;

  //  move up elements that follow deleted item in sorted list

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

        length-- ;
}
```

# Improving member function RetrieveItem

- Recall that with the Unsorted List ADT we examined each list element beginning with info[ 0 ], until we either found a matching key, or we had examined all the elements in the Unsorted List.

- How can the searching algorithm be improved for Sorted List ADT?

# Retrieving Eric from a Sorted List




**The sequential search for Eric can stop when Hsing has been examined.**

# Binary Seach in a Sorted List

- **Examines 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.

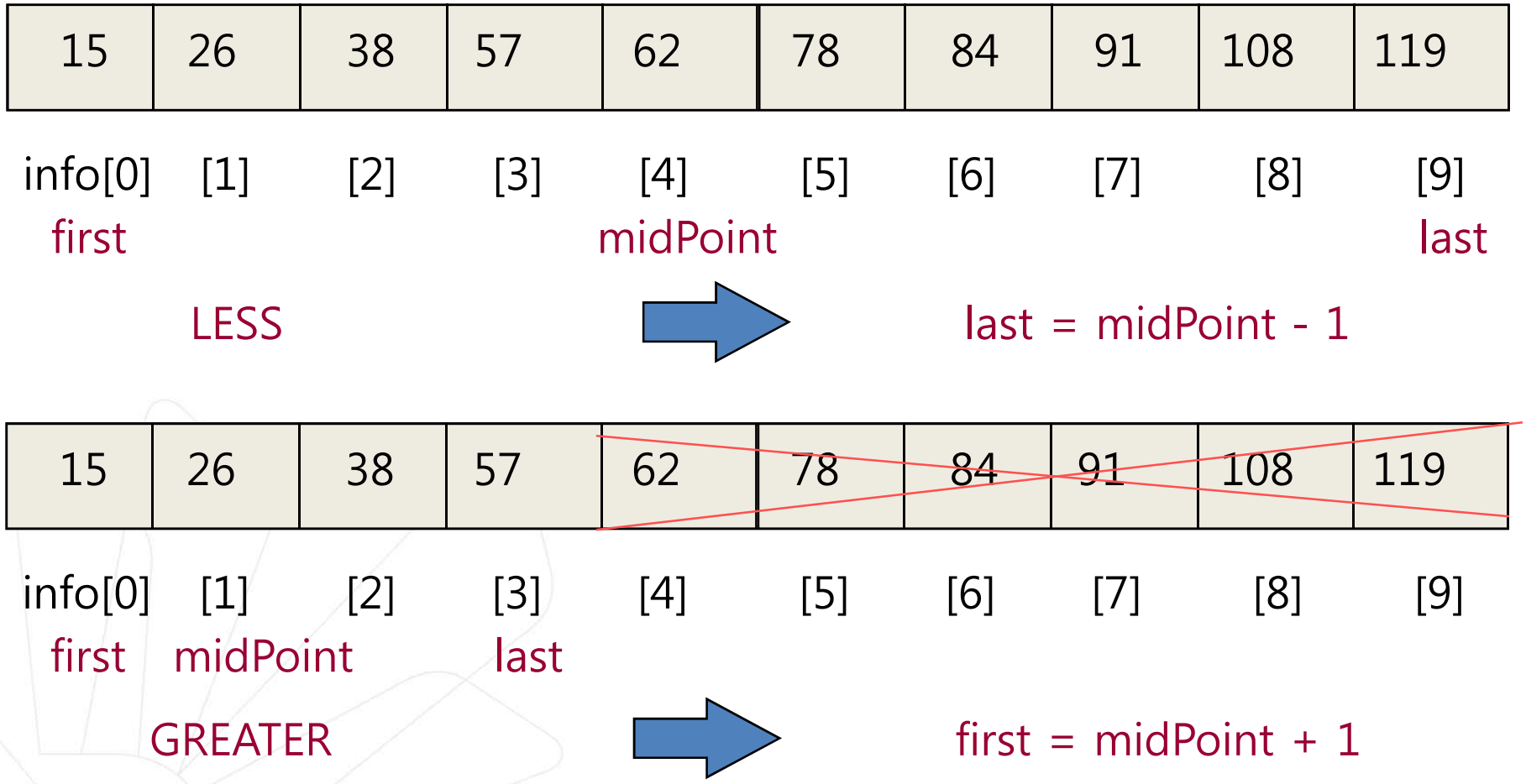
- **Repeat the process in the half of the list** that should be examined next.

- Stop when item is found, or when there is nowhere else to look and item has not been found.

```
void SortedType::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.
   int     midPoint ;
   int     first  =  0;
           int       last  = length - 1 ;
   bool  moreToSearch  =  ( first  <=  last ) ;

  found = false ;
   while ( moreToSearch  &&  !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
           }
53 }
}
```

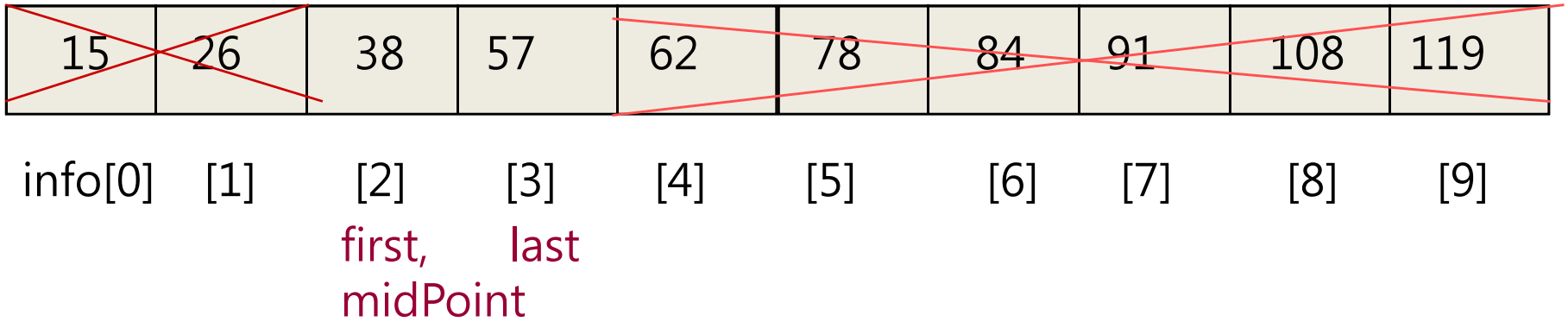
# Trace of Binary Search

item = 45

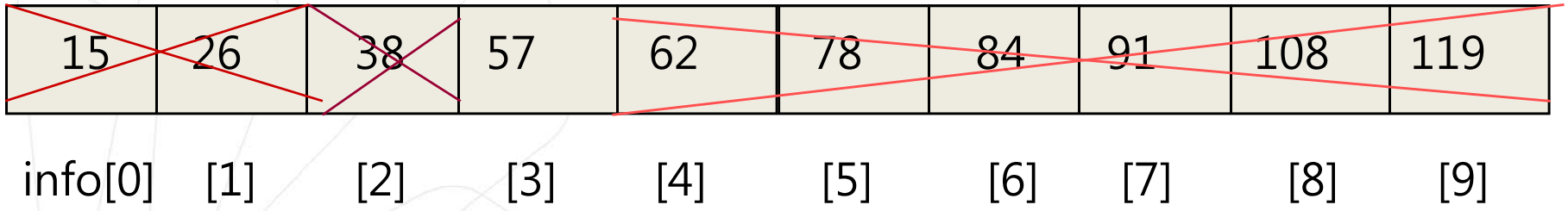


# Trace continued

item = 45



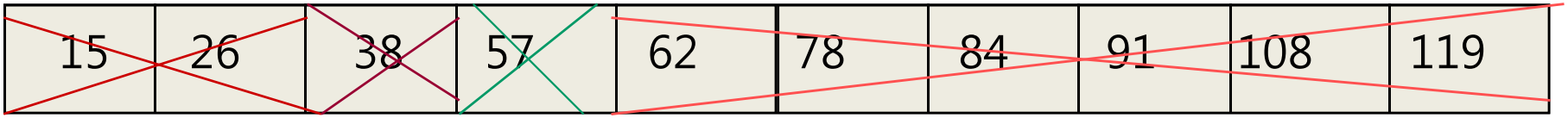

GREATER → first = midPoint + 1




LESS → last = midPoint - 1

# Trace concludes

item = 45




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

last first

first > last

found = false

```
void SortedType::RetrieveItem ( ItemType& item, bool& found ) {
// ASSUMES info ARRAY SORTED IN ASCENDING ORDER
  int      midPoint ;
  int      first  =  0;
           int        last  = length - 1 ;
  bool  moreToSearch  =  ( first  <=  last ) ;
  found = false ;

  while ( moreToSearch  &&  !found ) {
           midPoint  =  ( first + last ) / 2 ;
           switch ( item.ComparedTo( info [ midPoint ] ) )  {
               case  LESS        :       last = midPoint - 1 ;
                                         moreToSearch = ( first <= last ) ;
                                         break ;
                case  GREATER  :        first = midPoint + 1 ;
                                         moreToSearch = ( first <= last ) ;
                                         break ;
                case  EQUAL       :      found = true  ;
                                         item = info[ midPoint ] ;
                                         break ;
           }
57   }
}
```

# Comparison of Algorithms

- **Order of Magnitude of a Function**

- **The order of magnitude, or Big-O notation, of a function expresses the computing time of a problem as the term in a function that increases most rapidly relative to the size of a problem.**

# Names of Orders of Magnitude

**N:** **size of the problem**

| | |
|---|---|
| $O(1)$ | bounded (by a constant) time |
| $O(\log_2 N)$ | logarithmic time |
| $O(N)$ | linear time |
| $O(N*\log_2 N)$ | $N*\log_2 N$ time |
| $O(N^2)$ | quadratic time |
| $O(2^N)$ | exponential time |

| N | $\log_2 N$ | $N*\log_2 N$ | $N^2$ | $2^N$ |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 2 |
| 2 | 1 | 2 | 4 | 4 |
| 4 | 2 | 8 | 16 | 16 |
| 8 | 3 | 24 | 64 | 256 |
| 16 | 4 | 64 | 256 | 65,536 |
| 32 | 5 | 160 | 1024 | 4,294,967,296 |
| 64 | 6 | 384 | 4096 | |
| 128 | 7 | 896 | 16,384 | |

# Big-O Comparison of List Operations

| OPERATION | UnsortedList | SortedList |
|---|---|---|
| RetrieveItem | O(N) | O(N) linear search<br>$O(\log_2 N)$ binary search |
| InsertItem | | |
| Find | O(1) | O(N) search |
| Put | O(1) | O(N) moving down |
| Combined | O(1) | O(N) |
| DeleteItem | | |
| Find | O(N) | O(N) search |
| Put | O(1) swap | O(N) moving up |
| Combined | O(N) | O(N) |