Chapter 3

ADT Unsorted List

Lists

- A **list** is a homogeneous collection of elements with a *linear relationship* between elements
- **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 the list, which can vary over time

List Definitions

- **Unsorted List:** A list in which data items are placed in no particular order
- **Sorted List:** A list that is sorted by the *keys* of the list items; there is a semantic relationship among the keys of the items in the list
- **Key:** The attributes used to determine the logical order of the list

Unsorted List: Operations

- Constructor: May make empty list or take some initial elements
- Transformers: PutItem, DeleteItem, MakeEmpty
- Observers: IsFull, GetLength
- Iterators: ResetList, GetNextItem

What is a generic data type?

- A type for which the operations are defined but the types of the items being manipulated are not defined
- Our Unsorted List ADT simulates this by using a user-defined class called ItemType that defines the member functions we need

Unsorted List: Application Level

- Unsorted lists seem to provide a few, limited operations
- But it provides all the tools we need to write more specialized functions
- Printing list items, reading items from a file, and so on can all be written using this limited set of operations

Unsorted List: Implementation

- Different ways of storing items in memory affect the performance of list operations
- **Array-based:** Elements are stored sequentially in a contiguous chunk of memory
- **Linked list-based:** Elements are stored in separate nodes, connected by pointers
- Using an array-based implementation for now

Array vs. Linked List

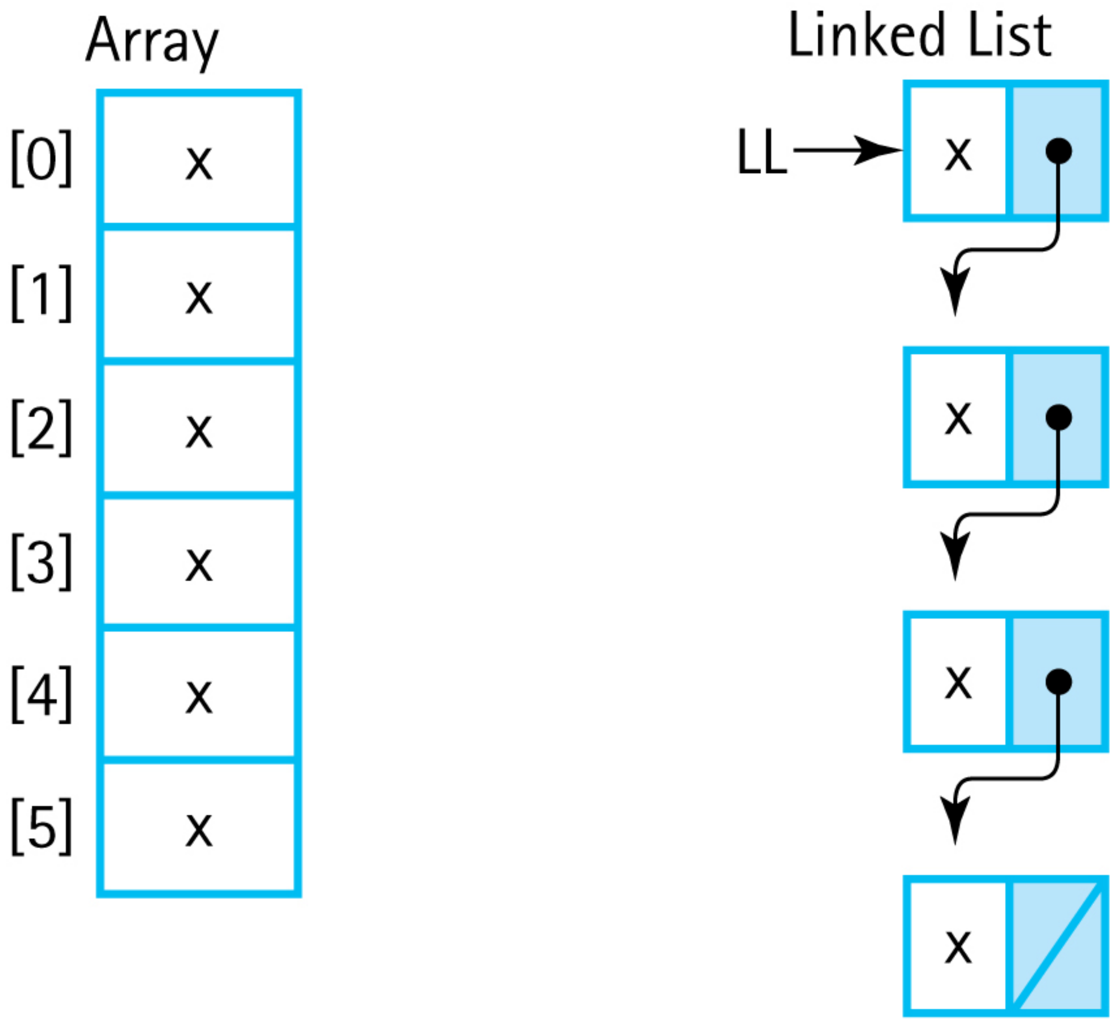


Figure 3.1 Comparison of Array and Linked Structure

Implementation: Constructors

- **Class Constructor:** A special member function that is implicitly invoked when a class object is created
  - Used to initialize objects and allocate resources
- For UnsortedType, the constructor sets the length to 0
- No resources need to be allocated because the size of the array is static

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: SomeClass anObject(param1, param2);
4) The parameterless constructor is the default constructor
5) A class must have a default (parameterless) constructor in order to create arrays of that class

Implementation: Observers

- IsFull and GetLength are straightforward
- GetItem requires a linear search through the list
  - Use ItemType.CompareTo to check equality
  - Use **`bool &found`** reference parameter to indicate success to the user
  - Return a *copy* of the found item

Implementation: Transformers

- PutItem: Insert item at the end of the list, increment length
- DeleteItem: Linear search to find item, but how to remove it?
  - If the item is at the end of the list, just reduce the length of the list by one
  - But what about if it's at the beginning?

Implementation: DeleteItem

Two strategies for removing an item in the middle of an array-based list:

- **Move Up:** Move all items after the removed item up one position; this is inefficient for long lists
- **Swap:** The list is unordered, so copy the item at the end of the list into the deleted item’s position

DeleteItem: Move or Swap?

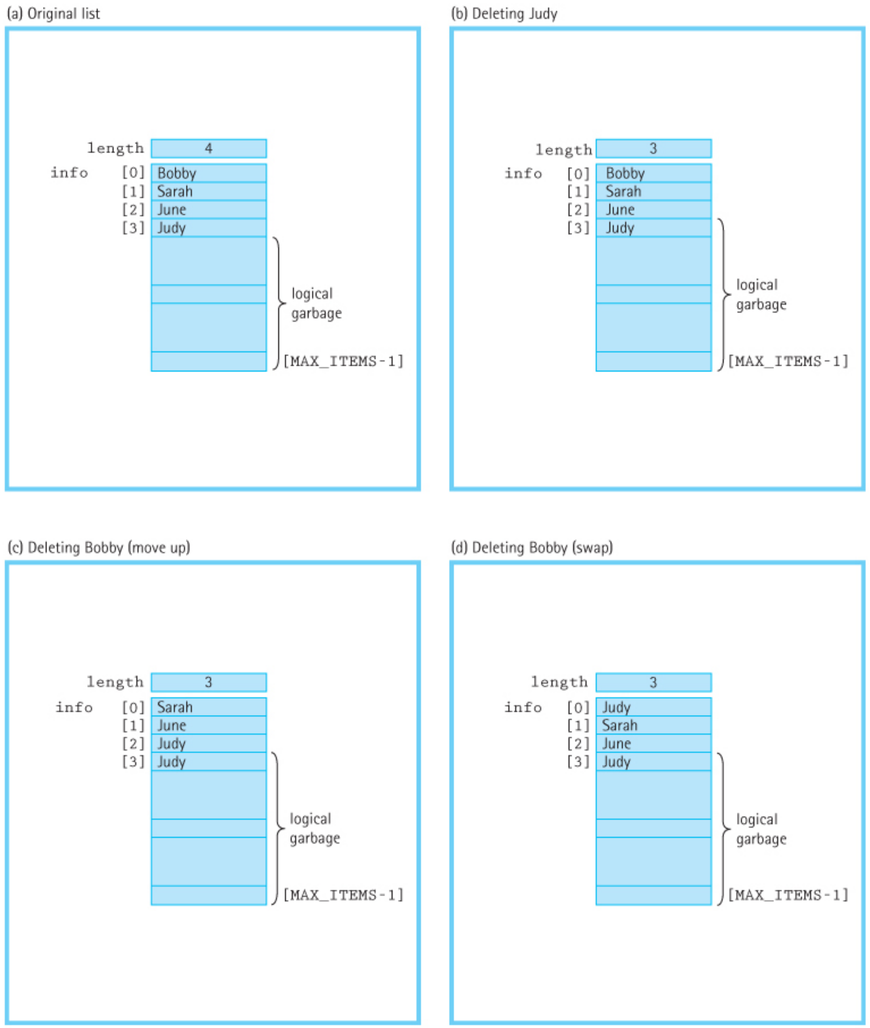


Figure 3.4 Deleting an item in an unsorted list (a) Original list (b) Deleting Judy (c) Deleting Bobby (move up) (d) Deleting Bobby (swap)

Implementation: Iterators

- The field **`currentPos`** indicates the current position of iteration in the list
- GetNextItem increments **`currentPos`** and returns the item at that position
- ResetList must set **`currentPos`** to the first item's predecessor
  - For arrays, that's -1

Unsorted List: Test Plan

- Preconditions and postconditions determine the black-box tests
- Code of functions determines clear-box tests
- We make sure the tests touch upon edge cases, such as the first and last items in the list, or a list with only one item

Pointer Types: Logical Level

- A **pointer variable** contains the *memory address* of another variable
- They are used for *indirect addressing* of data and for *dynamic allocation* of memory
- Pointers are declared using an asterisk:

```
int* intPointer;
```

Pointer Operators

- The prefix ampersand (&) or “address of” operator returns the address of a variable:
  - `int alpha = 10;`
  - `intPointer = α`
  - intPointer now contains alpha’s memory address
- The asterisk *dereference* operator (*) denotes the variable to which a pointer points:
  - `*intPointer = 25; // alpha is now 25`

Dynamic Allocation

- **Dynamic Allocation:** Allocation of memory space for a variable at run time, as opposed to *static allocation* at compile time
- Dynamic allocation creates variables on the **heap** (or **free store**), a section of memory reserved for dynamic allocation

Dynamic Allocation (cont.)

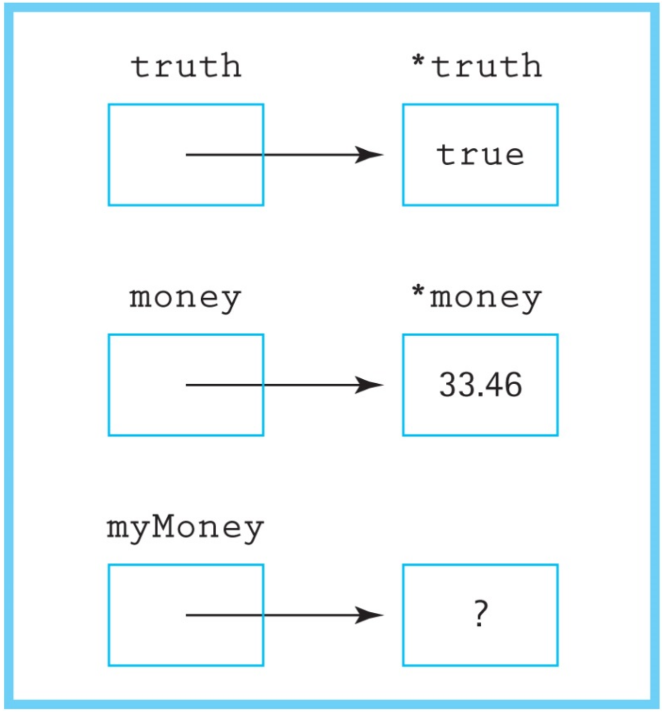
- Dynamic allocation uses the keyword **`new`**:
  - **`int* ptr = new int;`**
- **`new`** returns a pointer to the newly allocated int on the heap, and the value can only be accessed via this pointer
- Pointers can point to nothing using NULL
- If no memory is available on the heap, **`new`** returns NULL

Memory Leak

- A **memory leak** is the loss of available memory space that occurs when some dynamically allocated memory is never deallocated
- Dynamically allocated memory that can’t be accessed is called **garbage**

Memory Leak Example

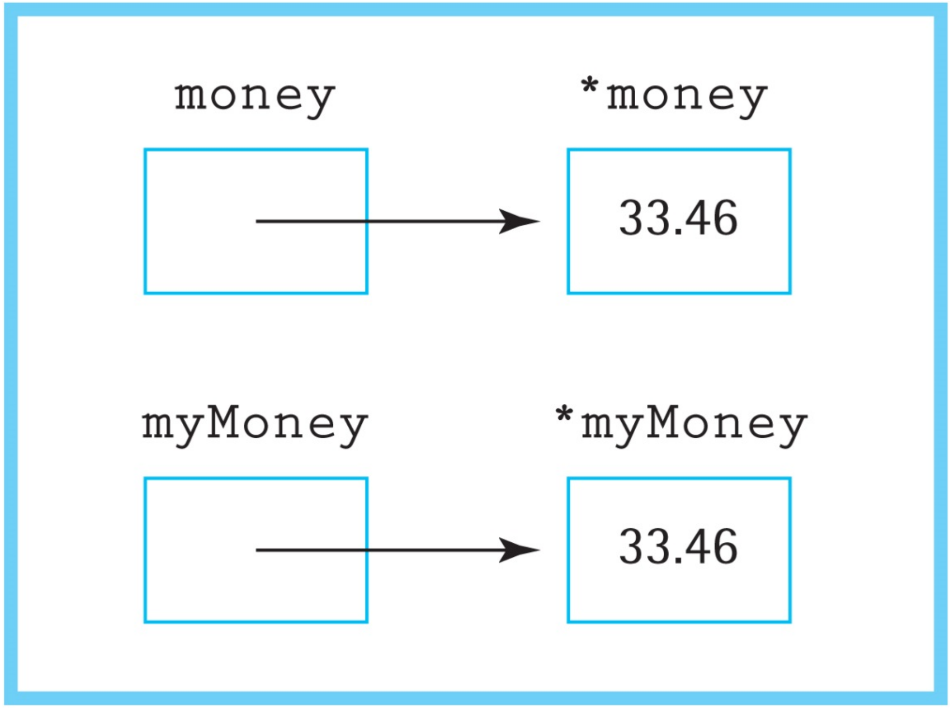
```
float* money = new float;
*money = 33.46;
float* myMoney = new float;
```



Memory Leak Example (cont.)

This copies the value pointed to by money into the memory pointed to by myMoney:

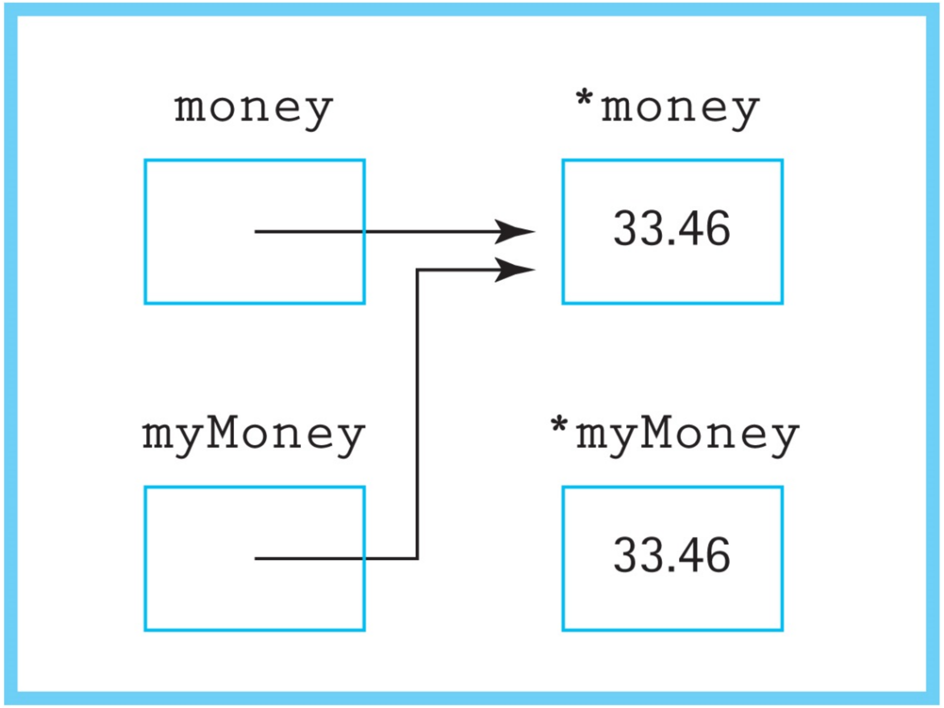
```
*myMoney = *money;
```



Memory Leak Example (cont.)

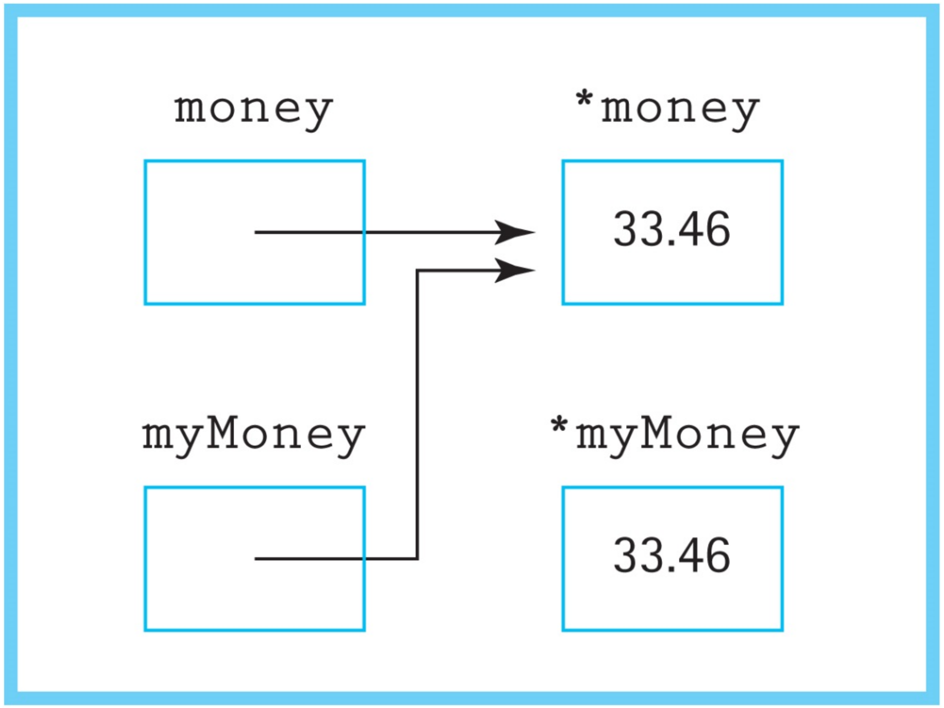
This makes myMoney point to the same address that money points to:

`myMoney = money;`



Memory Leak Example (cont.)

The memory cell originally used by myMoney is now inaccessible. Since there's no way to collect the garbage, it is a small memory leak.



The Delete Operator

- The delete operator deallocates the memory pointed to by the argument
- Using **`delete myMoney`** would safely clean up the memory allocated to myMoney
- C++ requires *manual memory management*

Pointers: Application Level

- The name of an array variable when used without brackets is a *pointer constant*
- Pointers can be used with constant types, allowing objects to link to each other
- If myPtr is a pointer to an object, the fields of the object are accessed using the arrow operator: myPtr->field

Pointers: Implementation Level

- Pointers and dynamic memory are thankfully handled entirely by the operating system, compiler, and run-time system
- Except you **must** remember to delete dynamically allocated memory when you are finished with it!

UnsortedType as a Linked List

- **Linked List:** A collection of nodes that are linked together in a chain using pointers
- **Node:** The basic component of a linked list; stores data and a pointer to the next node
- Nodes are created when needed using dynamically allocated memory
- The last node in the list has a NULL pointer

Nodes

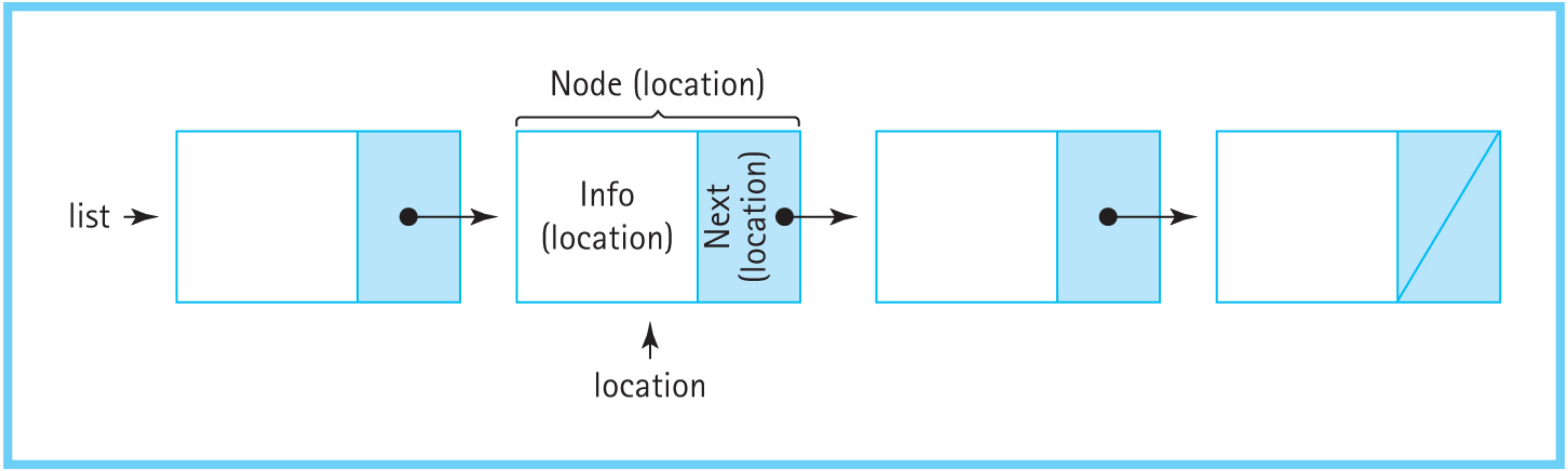


Figure 3.11 Node Terminology

PutItem

- When the list is empty, PutItem creates a node and sets the *external pointer* listData to it

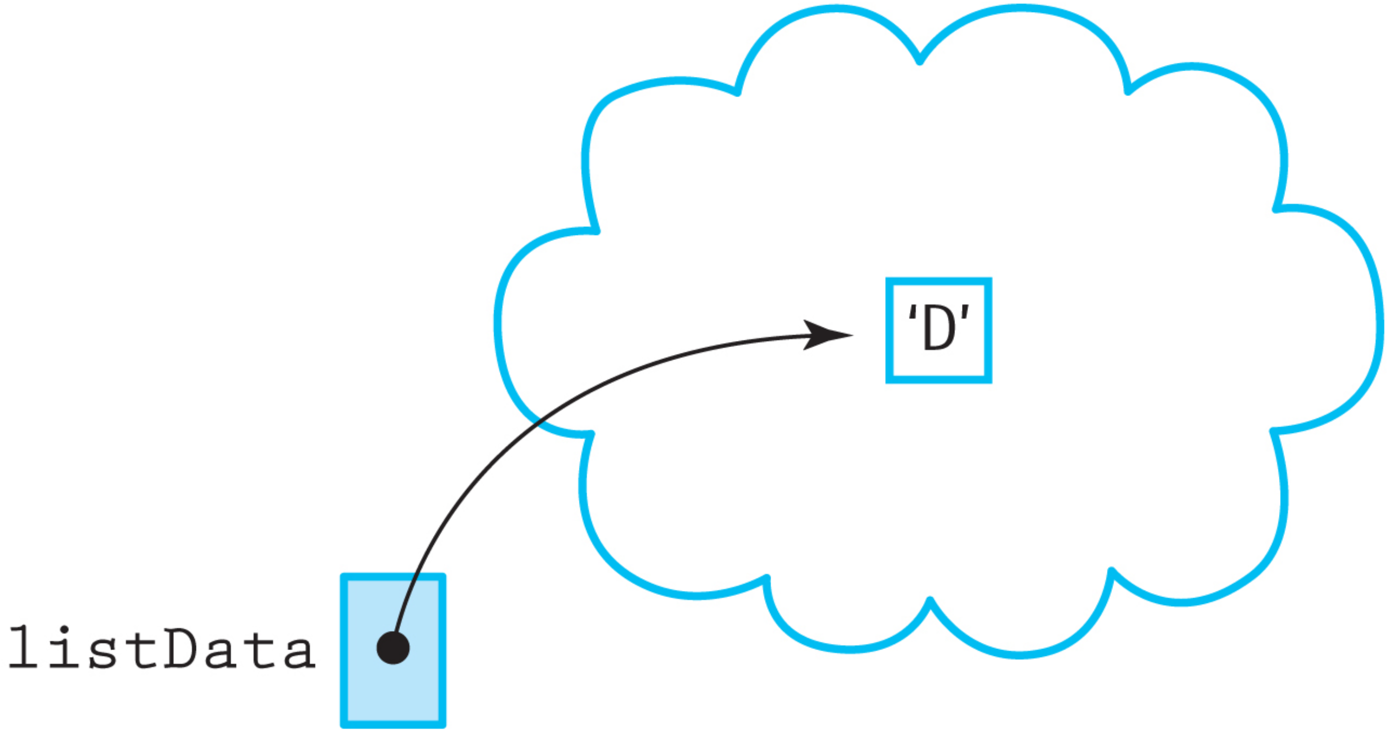


Figure 3.9 Putting in the First Element

Building the Chain

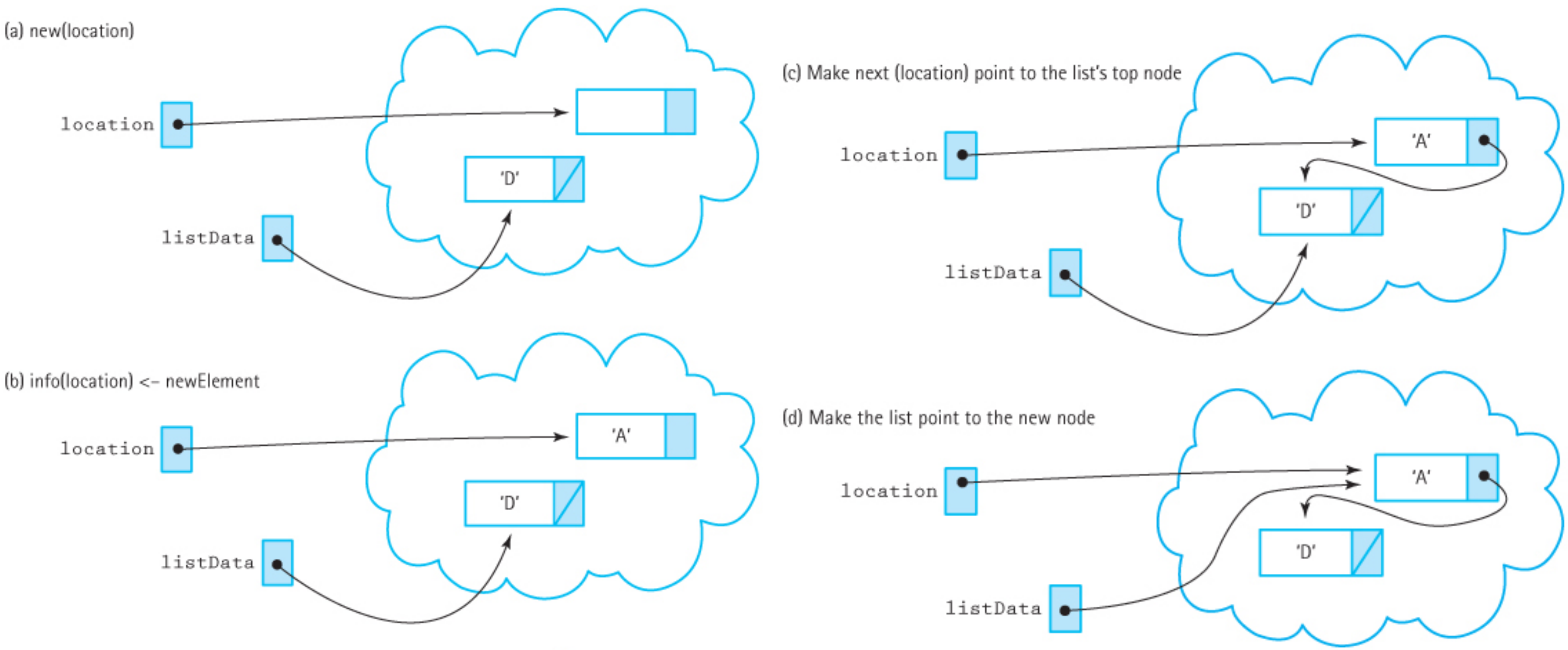


Figure 3.12 The second PutItem operation (a) new(location) (b) info(location) <- newElement (c) Make next (location) point to List's tope node (d) Make List point to the new node

Length and Linked Lists

- For array-based lists, the length field must be present in order to define the extent of the list within the array
- Linked lists don’t have this restriction
- Instead of a field, GetLength could walk the list and count the number of nodes

Iterators

- The array-based implementation used currentPos, an array index
- Linked lists use a pointer to a node instead
- GetNextItem advances it: currentPos->next
- ResetList sets it to NULL
  - GetNextItem checks if it’s NULL and sets it to the first item of the list

PutItem

1) Create a new node
2) Set the node’s info to the input data
3) Set the node’s next pointer to the listData, the first item in the list
4) Set listData to point to the new node

Order matters! Doing step 4 before step 3 would lead to a memory leak of the rest of the list.

PutItem (cont.)

```
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;
      // create a new node and fill it
  location = new  NodeType<ItemType>;
  location->info = item;
  location->next = listData;
  // the new node becomes head of the list
  listData = location;
  length++;
}
```

PutItem: Empty List

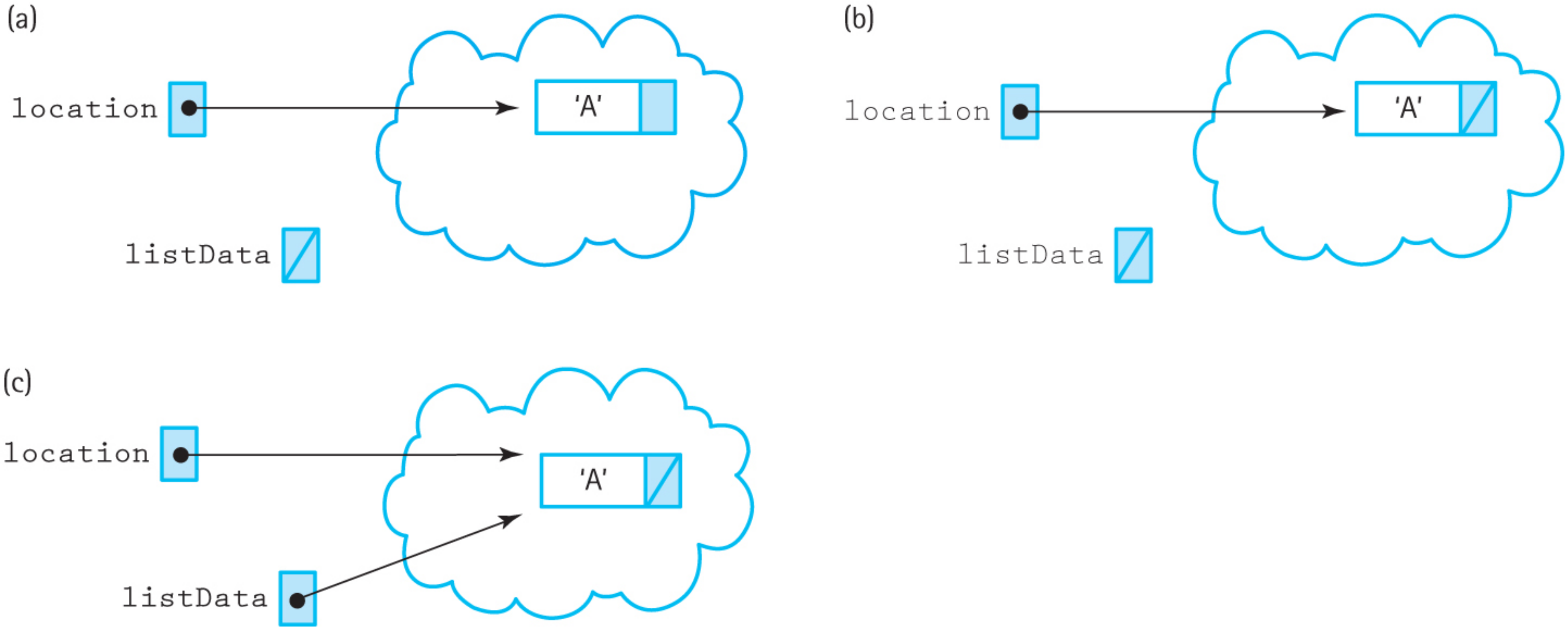


Figure 3.16 Putting an Item into an Empty List

Constructor

- Largely unchanged
- Set length to 0
- Set the external pointer to NULL

IsFull

- Linked lists don’t have an explicit size limit
- New nodes can be allocated until there is no more memory to use
- When this occurs, new will throw a bad_alloc exception
- IsFull uses a try-catch block to allocate a node and returns true if a bad_alloc is thrown

IsFull (cont.)

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

MakeEmpty

- MakeEmpty must deallocate each node individually in order to empty the list
- This is accomplished using a while loop
- Iteration starts at listData, the head of the list, and continues using listData->next
- Iteration stops when listData is NULL

GetItem

- The algorithm is unchanged: Linear search through the list to find the desired item
- In fact, the implementation is largely unchanged; it just needs to use pointers instead of array indices

GetItem (cont.)

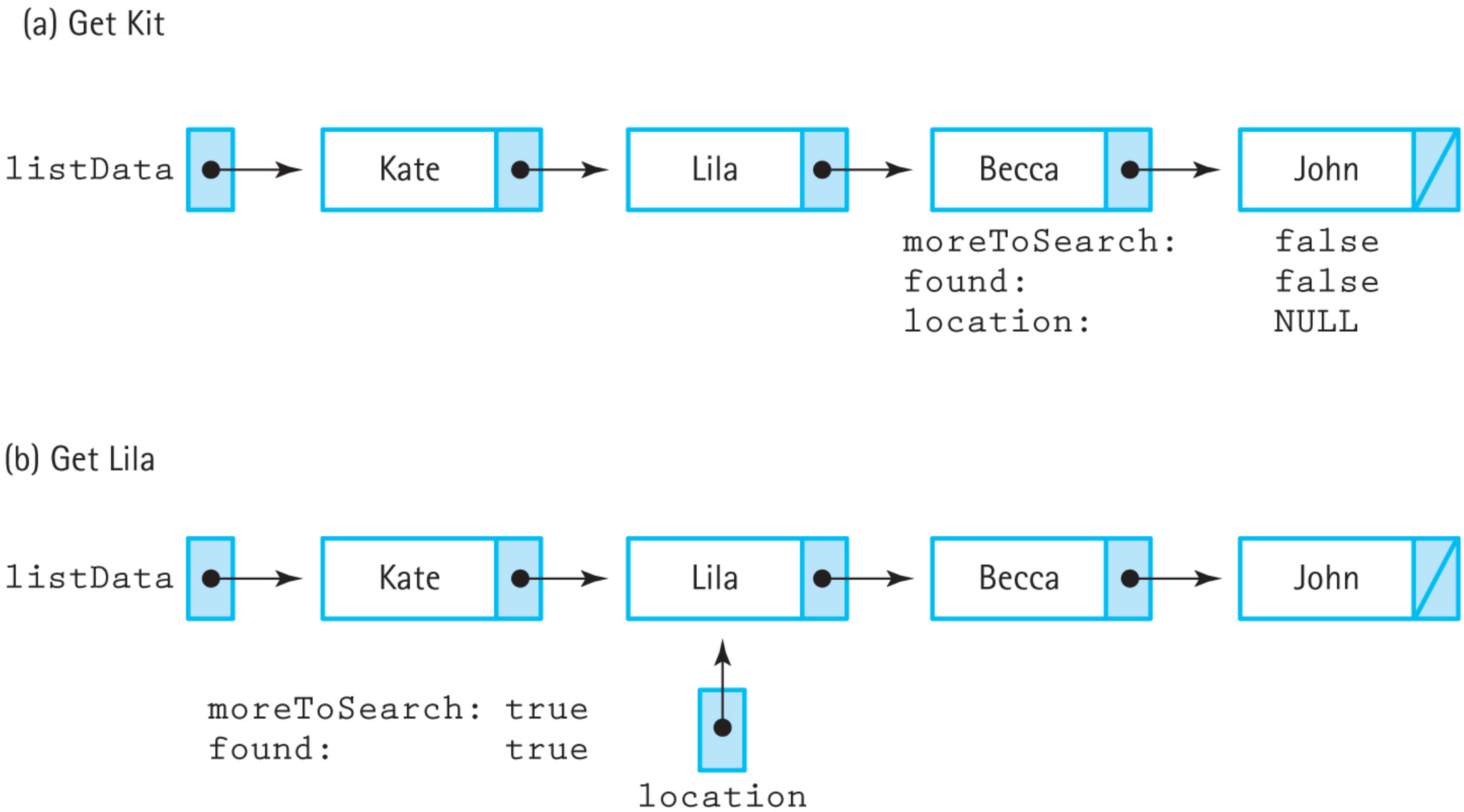


Figure 3.17 Retrieving an item in an unsorted linked list (a) Get Kit (b) Get Lila

DeleteItem

- Deleting an item requires updating the pointer of its predecessor
- Algorithm is the same, but search looks at the item of location->next in order to have access to the predecessor (location)

DeleteItem (cont.)

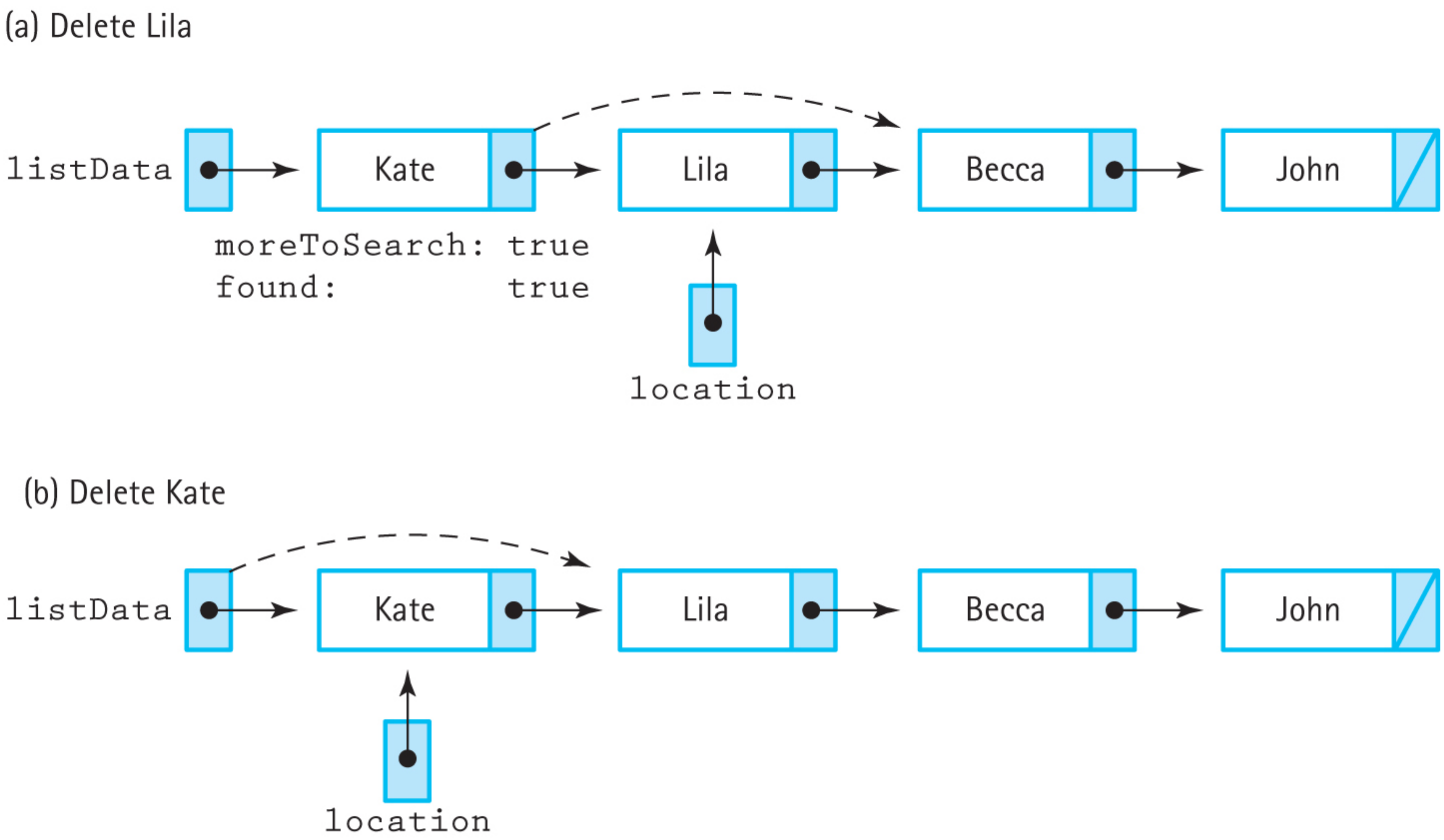


Figure 3.18 Deleting an interior node and deleting the first node (a) Delete Lila (b) Delete Kate

Lifetime of a Variable

- **Lifetime:** The time during execution that a variable has memory assigned to it
- Global variable: The entire execution of a program
- Local Variable: The execution of the block it is in
- Dynamically allocated variable: From when it is allocated to when it is deallocated
- The *static* keyword allows local variables to live outside the time of their blocks

Class Destructors

- An object is deallocated when it leaves scope, but any data it points to is not – memory leak!
- A **class destructor** is a method that is implicitly invoked when an object leave scope
- The linked list destructor (**`~UnsortedList`**) must clean up the object’s memory by deallocating all the nodes in the list

Comparing Implementations

- The array-based list allocates enough memory for the max list size, no matter how many items are actually in the list
- The linked list-based list only uses enough memory for the items in the list
- Both have operations that are O(1)
- Linked list's MakeEmpty is O(*N*)

Comparing Implementations (cont.)

Table 3.2 Big-O Comparison of Sorted List Operations

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

Table 3.2 Big-O Comparison of Sorted List Operations