Unsorted Lists

CSM 387 – Data Structures

Lecture 7

List Definitions

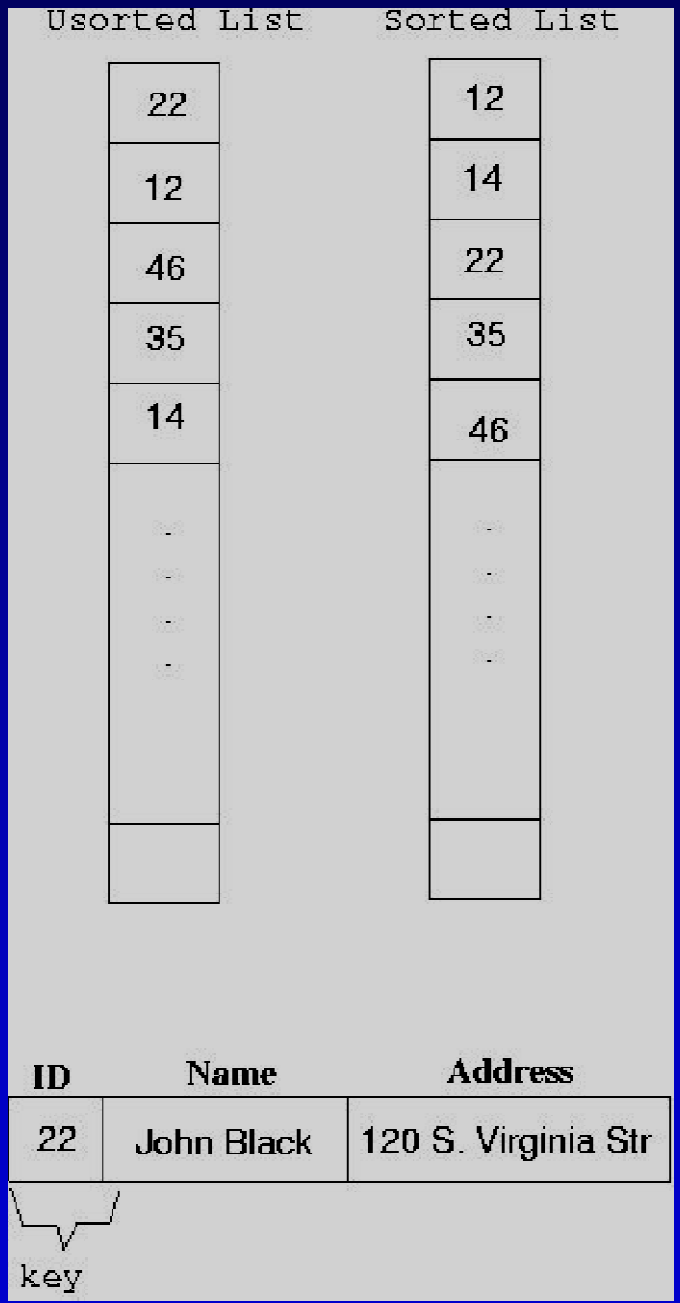
- Linear relationship: Each element except the first has a unique predecessor, and each element except the last has a unique successor.
- Length: The number of items in a list; the length can vary over time.

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.

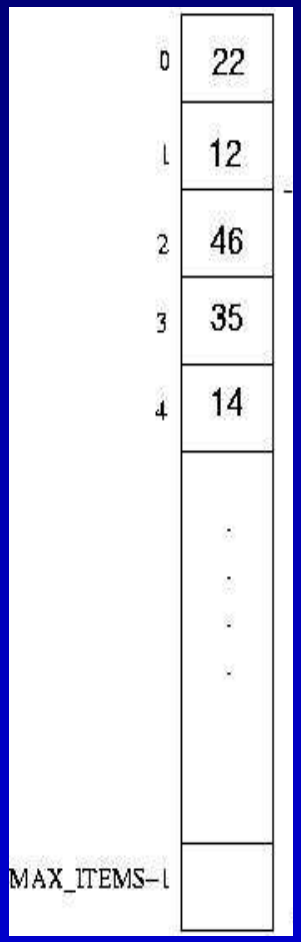
Sorted list

- A list in which data items are placed in a particular order.
- Key: The attributes that are used to determine the logical order of the list.

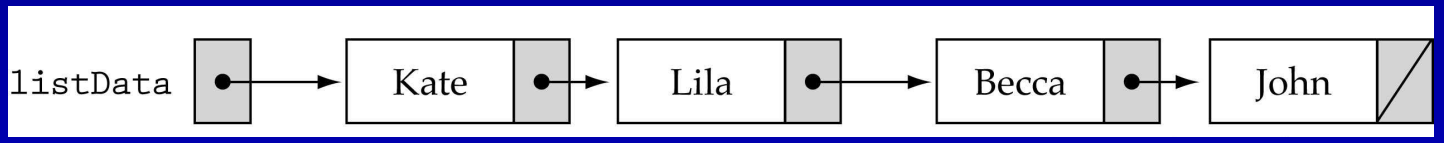

Usorted List
Sorted List
22
12
46
35
14
12
14
22
35
46
ID
Name
Address
22
John Black
120 S. Virginia Str
key

Unsorted List Implementations

Array-based



Linked-list-based



Remember

- Constructors
- Transformers
- Observers
- Iterators

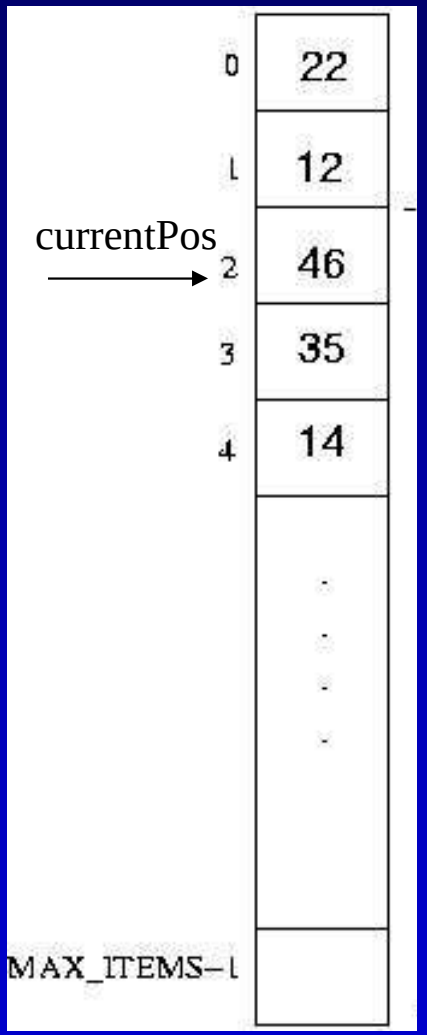
REMEMBER

- Transformers
  - MakeEmpty
  - InsertItem
  - DeleteItem
- Observers
  - IsFull
  - LengthIs
  - RetrieveItem
- Iterators
  - ResetList
  - GetNextItem

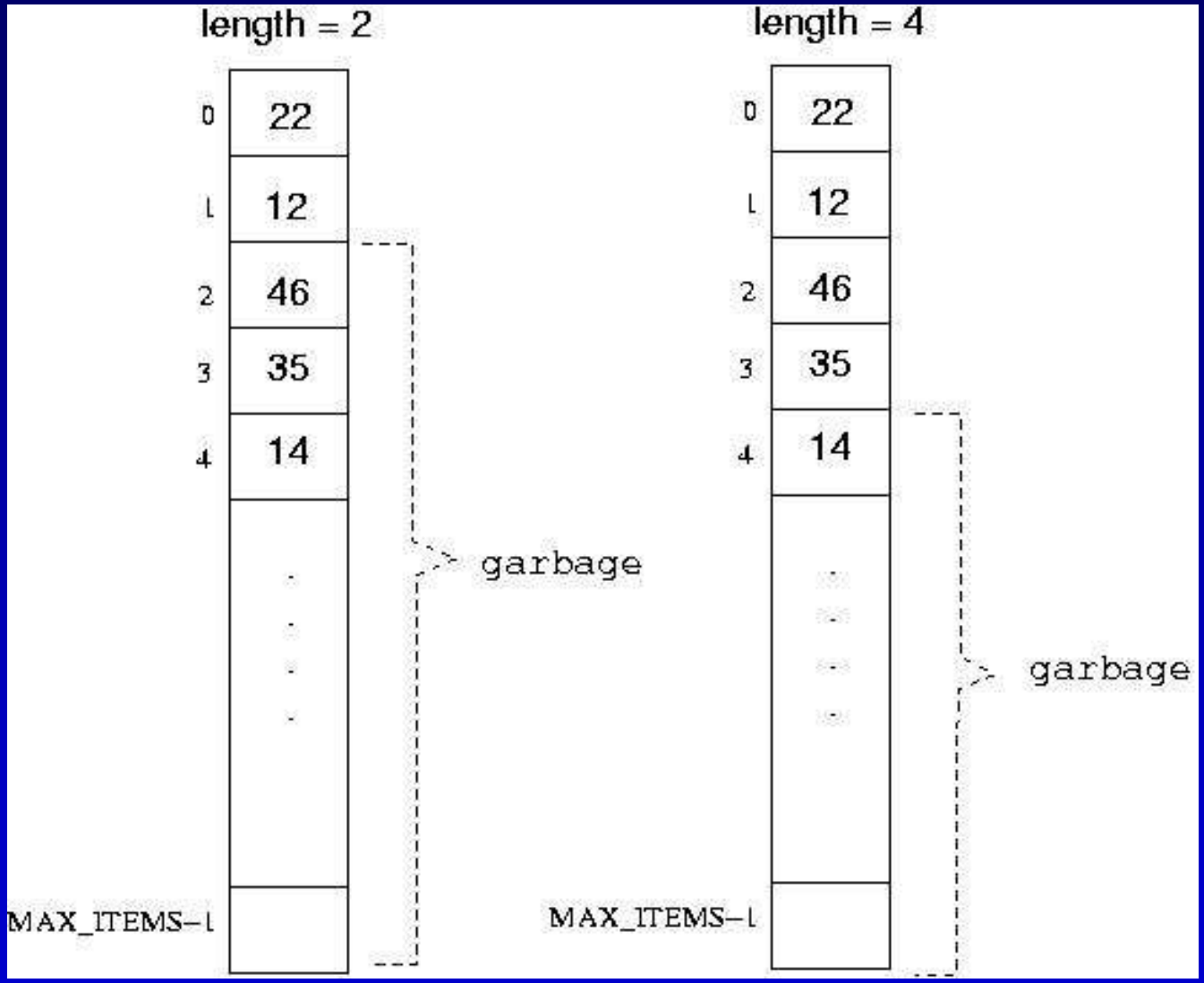
- CHANGE STATE
- OBSERVE STATE
- PROCESS ALL

Array-based Implementation

```
template<class ItemType>
class UnsortedType {
 public:
    void MakeEmpty();
    bool IsFull() const;
    int LengthIs() const;
    void RetrieveItem(ItemType&, bool&);
    void InsertItem(ItemType);
    void DeleteItem(ItemType);
    void ResetList();
    bool IsLastItem()
    void GetNextItem(ItemType&);
 private:
    int length;
    ItemType info[MAX_ITEMS];
    int currentPos;
};
```



Length: length of the list (different from array size!)



Array-based Implementation (cont.)

```
template<class ItemType>
void UnsortedType<ItemType>::MakeEmpty()
{
 length = 0;
}
```

O(1)

```
template<class ItemType>
bool UnsortedType<ItemType>::IsFull() const
{
 return (length == MAX_ITEMS);
}
```

O(1)

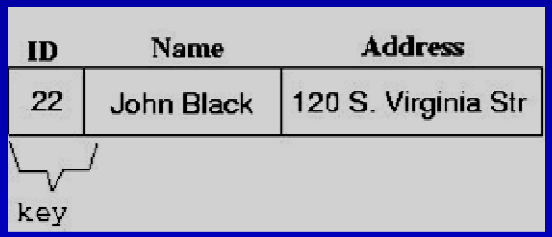
Array-based Implementation (cont.)

```
template<class ItemType>
int UnsortedType<ItemType>::LengthIs() const
{
 return length;
}
```

O(1)

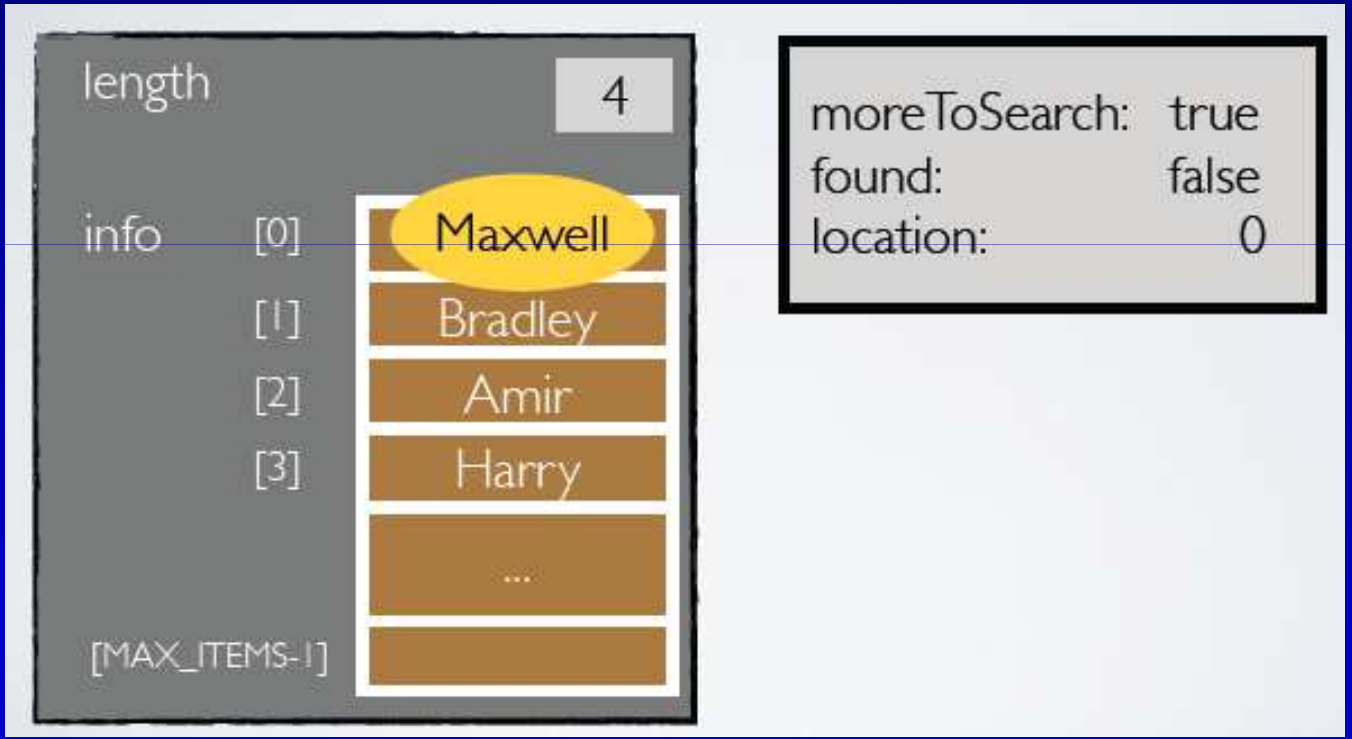
RetrieveItem (ItemType& item, boolean& found)

- ***Function***: Retrieves list element whose key matches item's key (if present).
- ***Preconditions***: (1) List has been initialized,
  (2) Key member of item has been initialized.

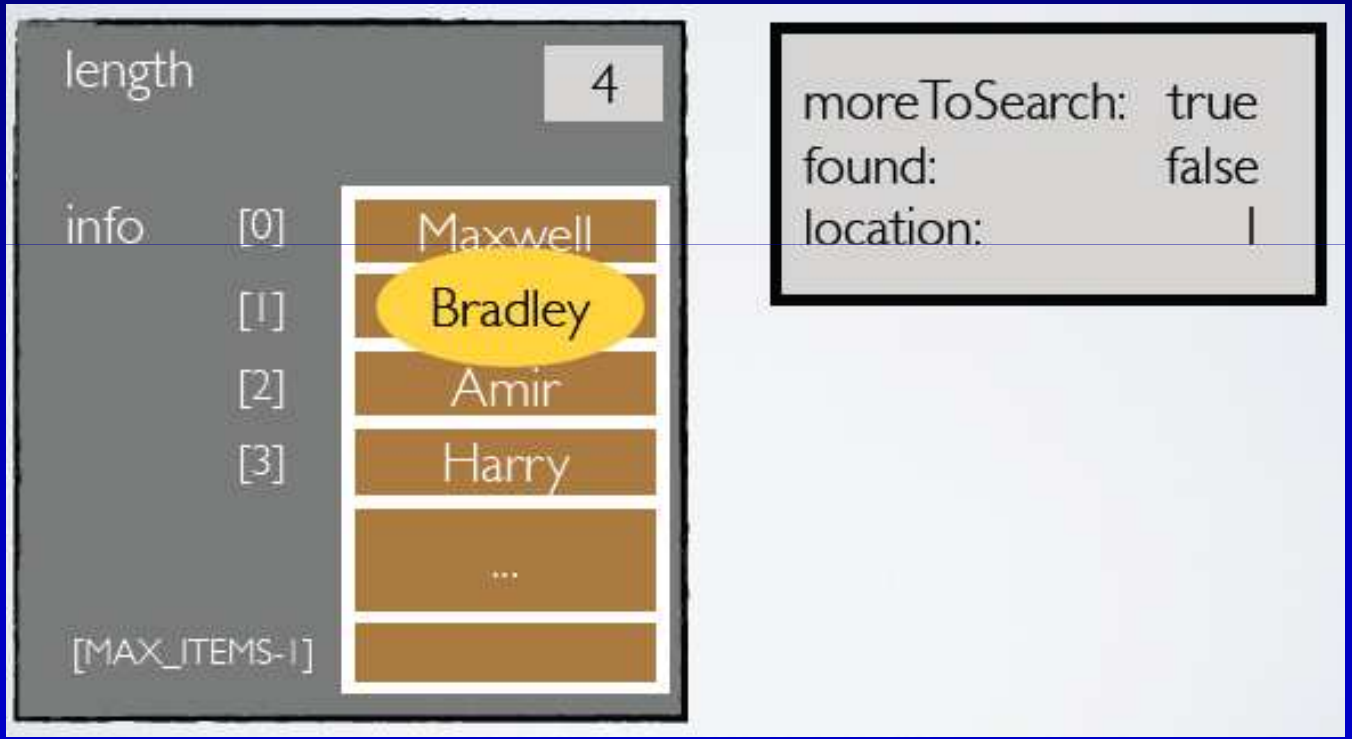


- ***Postconditions***: (1) If there is an element *someItem* whose key matches item's key, then *found=true* and *item* is a copy of *someItem*; otherwise, *found=false* and *item* is unchanged, (2) List is unchanged.

Retrieve MIKE from an Unsorted List

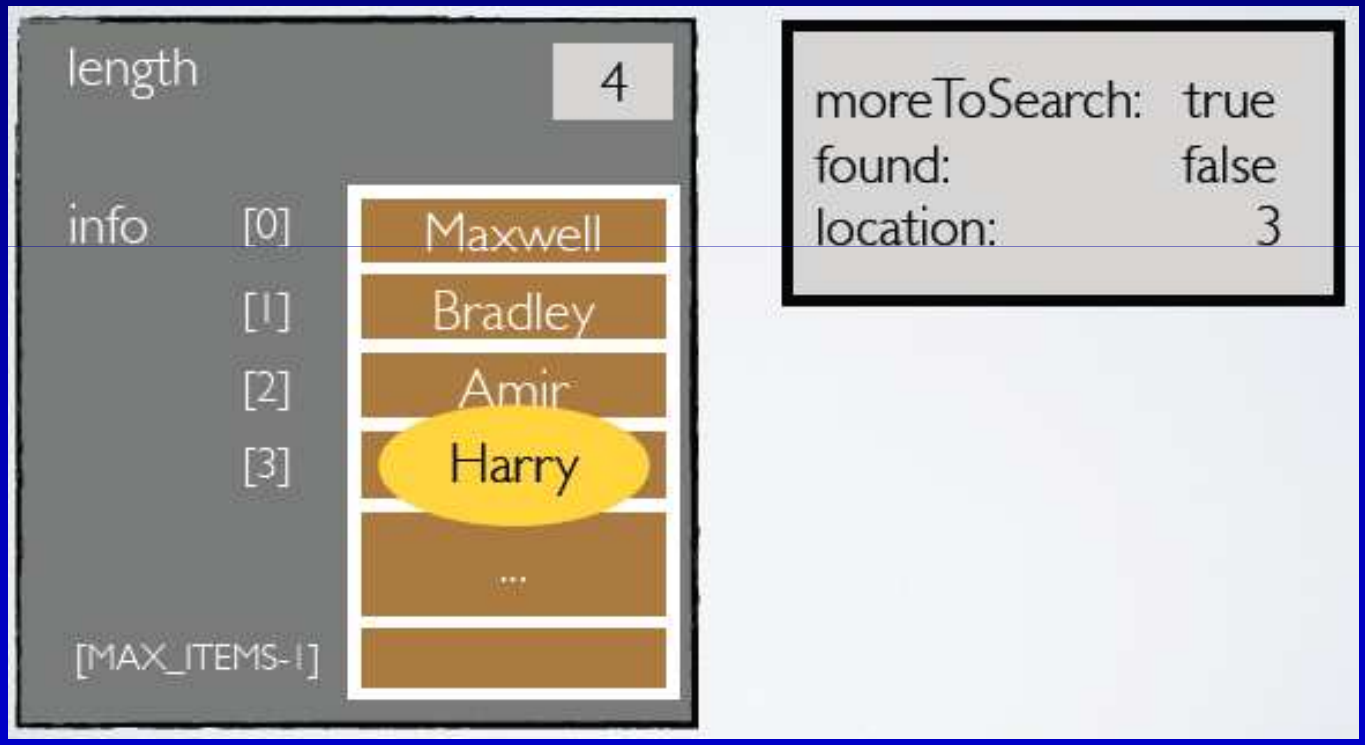


Retrieve MIKE from an Unsorted List

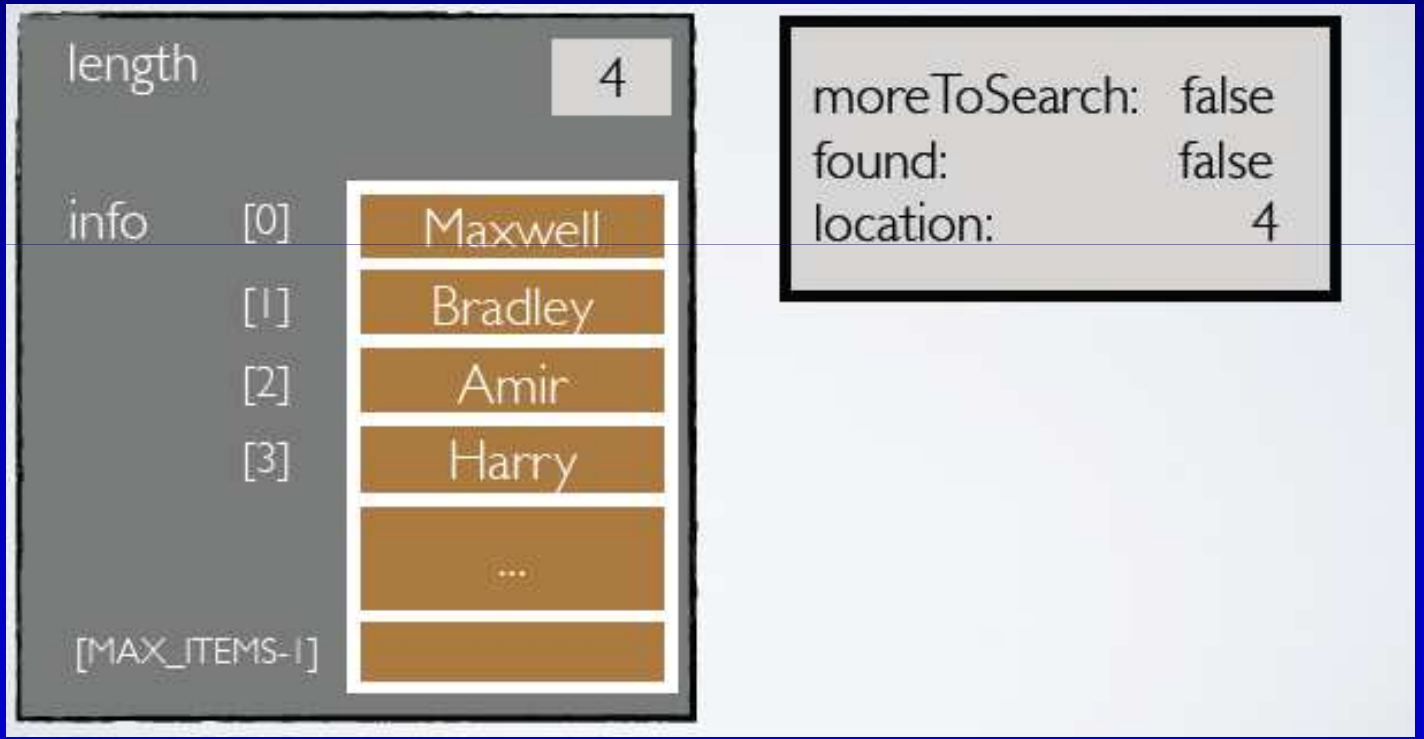


Retrieve MIKE from an Unsorted List

Retrieve MIKE from an Unsorted List

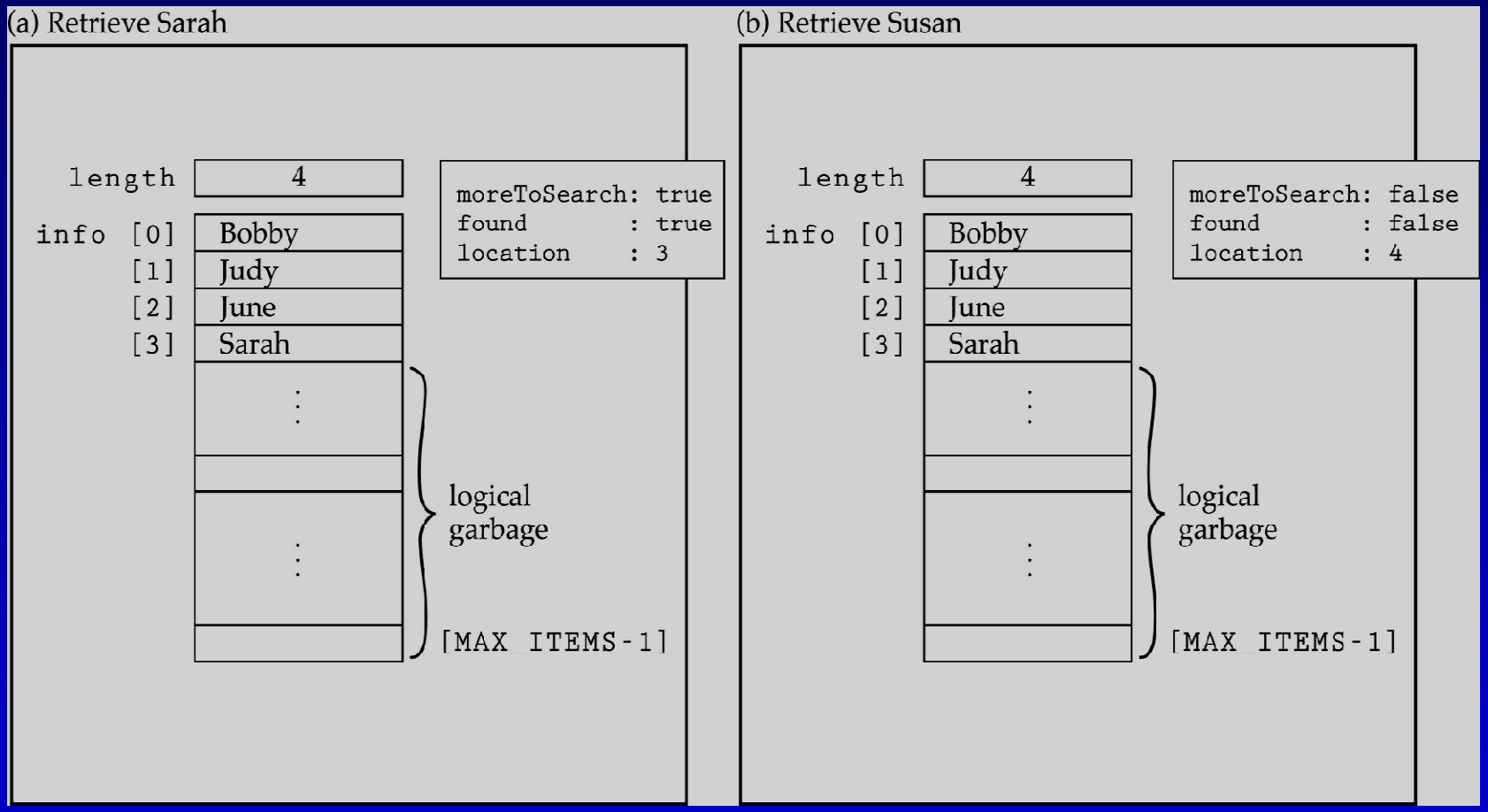


Retrieve MIKE from an Unsorted List

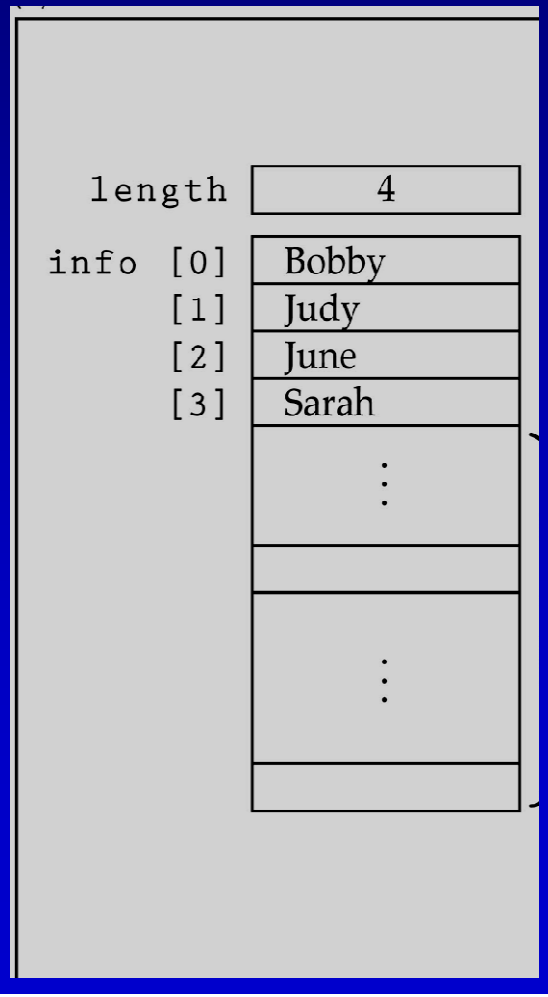


Item is in the list

Item is NOT in the list



Exercise: Retrieve **Judy** from the List below and show the status of the 3 control parameters; moreToSearch, found & location



Array-based Implementation (cont.)

```
template<class ItemType>
void UnsortedType<ItemType>::RetrieveItem (ItemType& item,
                                           bool& found)
{
 int location = 0;
 found = false;

 while( (location < length) && !found)
    if (item == info[location]) {
       found = true;
        item = info[location];
    }
     else
        location++;
}
```

O(N)

Who should overload “==“ ?

InsertItem (ItemType item)

- *Function*: Adds item to list

- *Preconditions*:

  (1) List has been initialized,

  (2) List is not full,

  (3) *item* is not in list (i.e., no duplicates)

- *Postconditions*: *item* is in list.

Array-based Implementation (cont.)

```
template<class ItemType>
void UnsortedType<ItemType>::InsertItem (ItemType
                                         item)
{
 info[length] = item;
 length++;
}
```

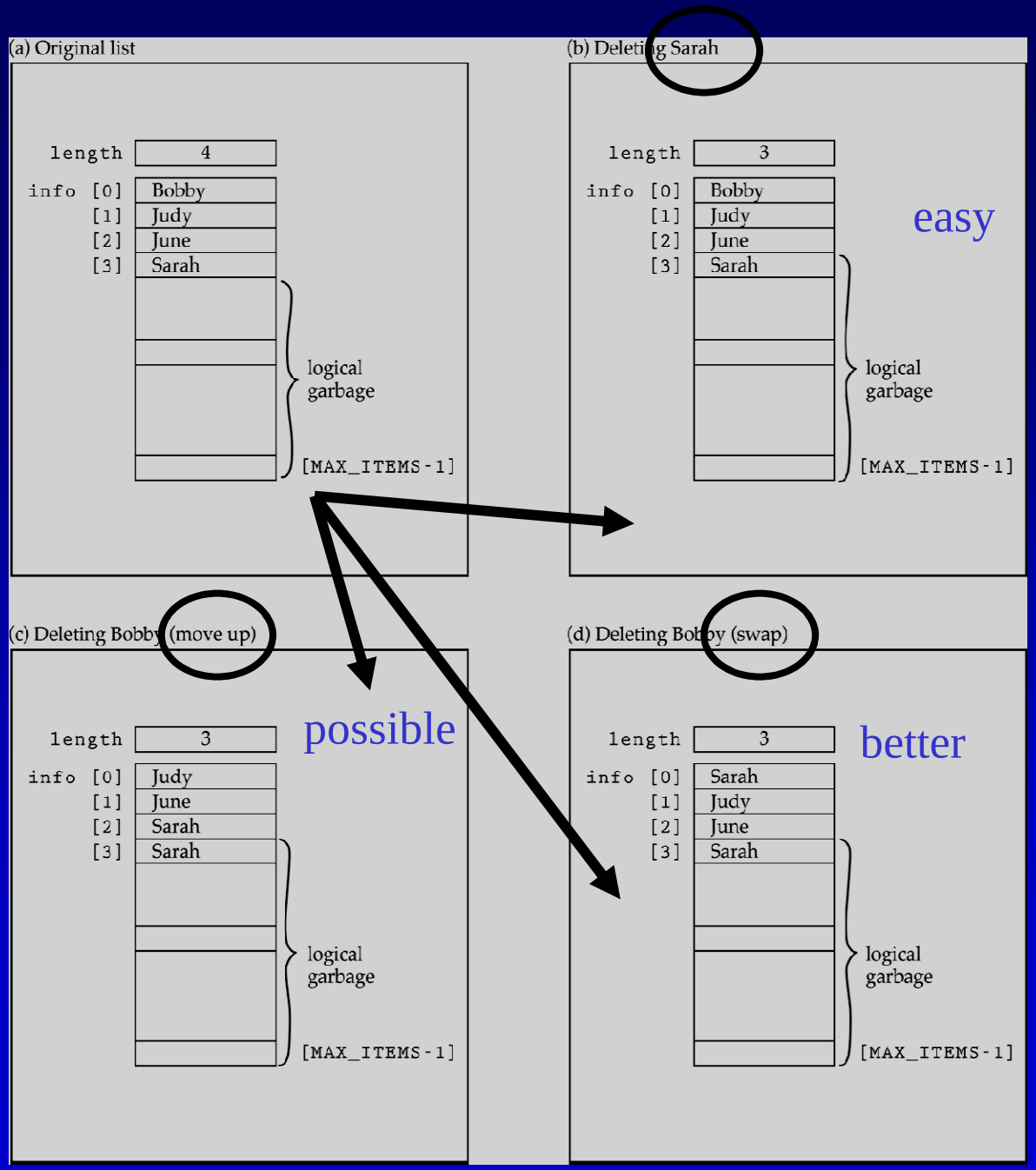
O(1)

DeleteItem (ItemType item)

- *Function*: Deletes the element whose key matches *item's* key

- *Preconditions*: (1) List has been initialized, (2) Key member of *item* has been initialized, (3) There is only one element in list which has a key matching *item's* key.

- *Postconditions*: No element in list has a key matching *item's* key.


(a) Original list
length
4
info [0]
[1]
[2]
[3]
Bobby
Judy
June
Sarah
logical garbage
[MAX_ITEMS-1]
(b) Deleting Sarah
length
3
info [0]
[1]
[2]
[3]
Bobby
Judy
June
Sarah
logical garbage
[MAX_ITEMS-1]
easy
(c) Deleting Bobby (move up)
length
3
info [0]
[1]
[2]
[3]
Judy
June
Sarah
Sarah
logical garbage
[MAX_ITEMS-1]
possible
(d) Deleting Bobby (swap)
length
3
info [0]
[1]
[2]
[3]
Sarah
Judy
June
Sarah
logical garbage
[MAX_ITEMS-1]
better

Array-based Implementation (cont.)

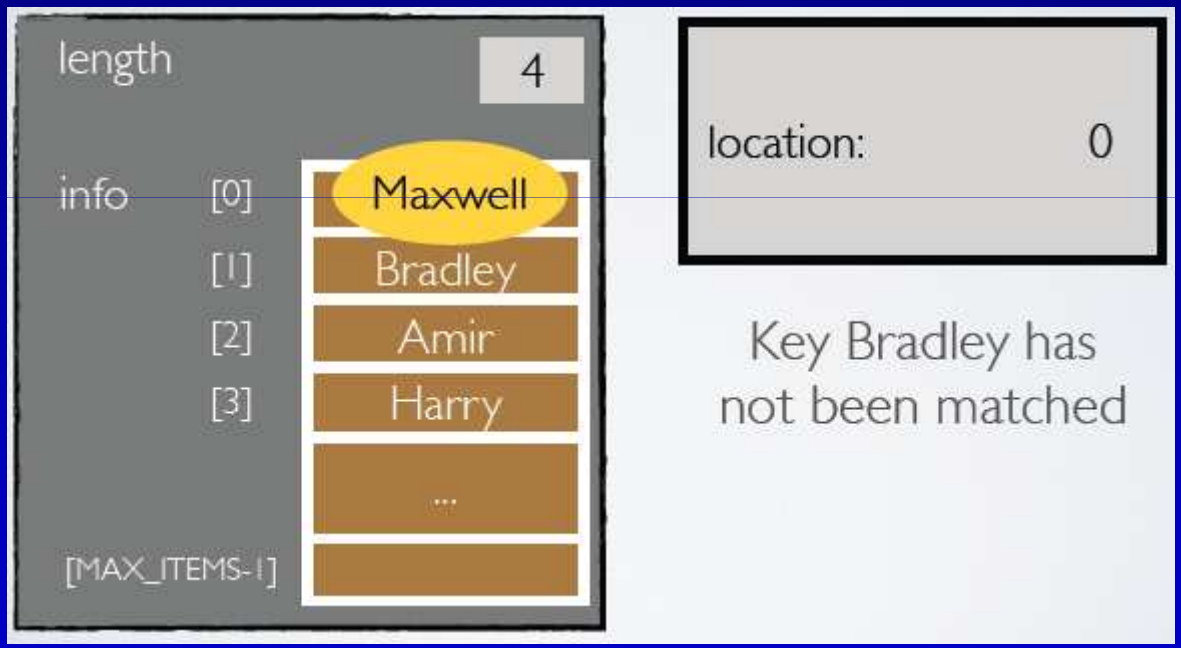
```
template<class ItemType>
void UnsortedType<ItemType>::DeleteItem(ItemType item)
{
 int location = 0;

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

 info[location] = info[length - 1];
 length--;
}
```

O(N)

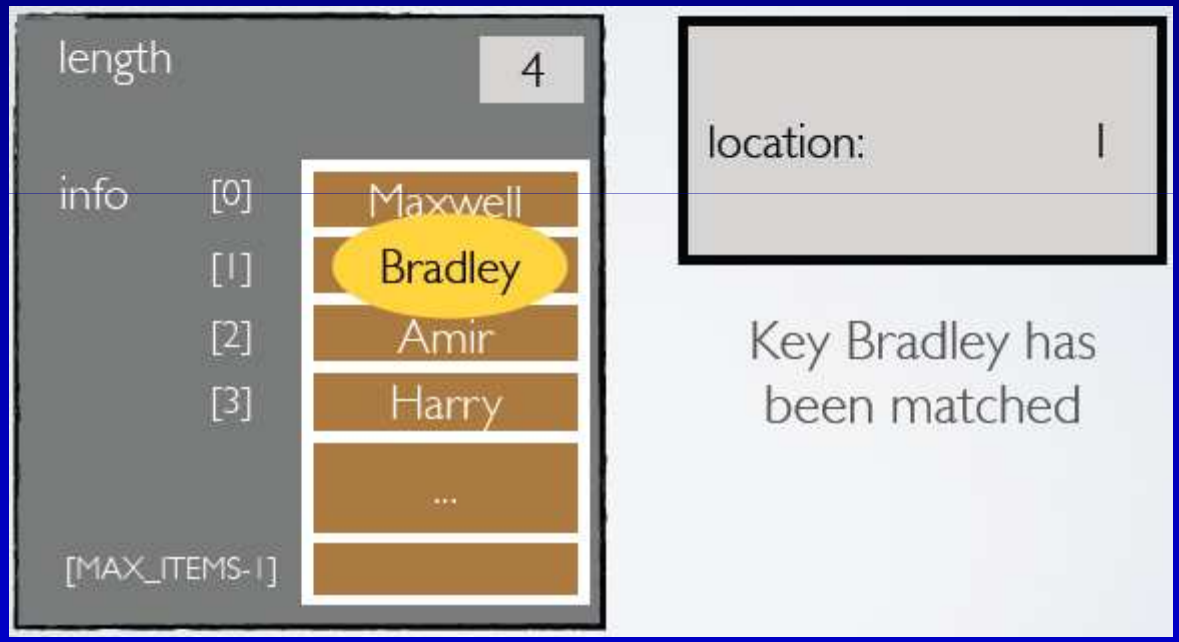
O(1)

O(N)

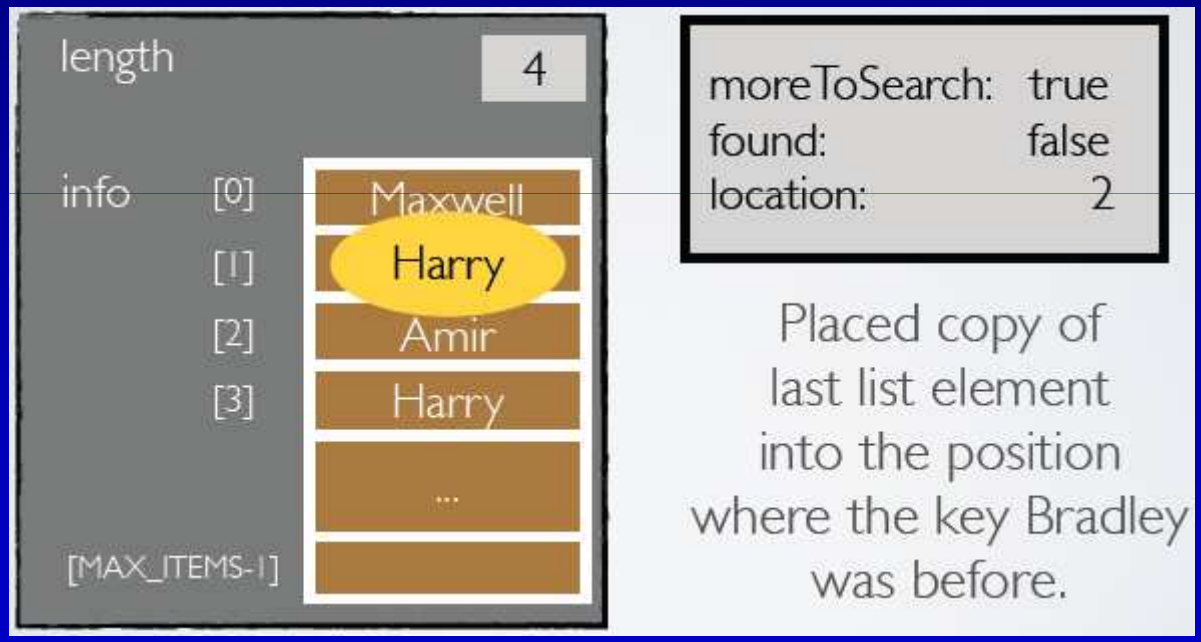
Deleting BRADLEY from an unsorted list



Deleting BRADLEY from an unsorted list



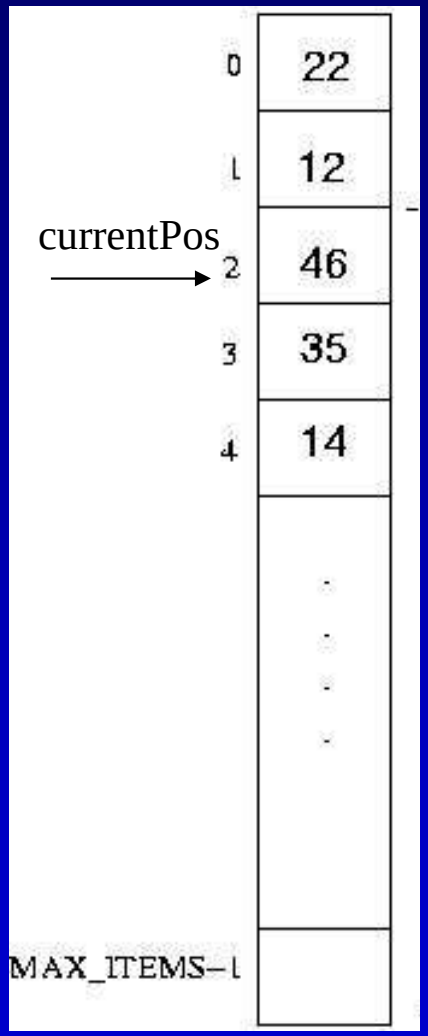
Deleting BRADLEY from an unsorted list



Deleting BRADLEY from an unsorted list

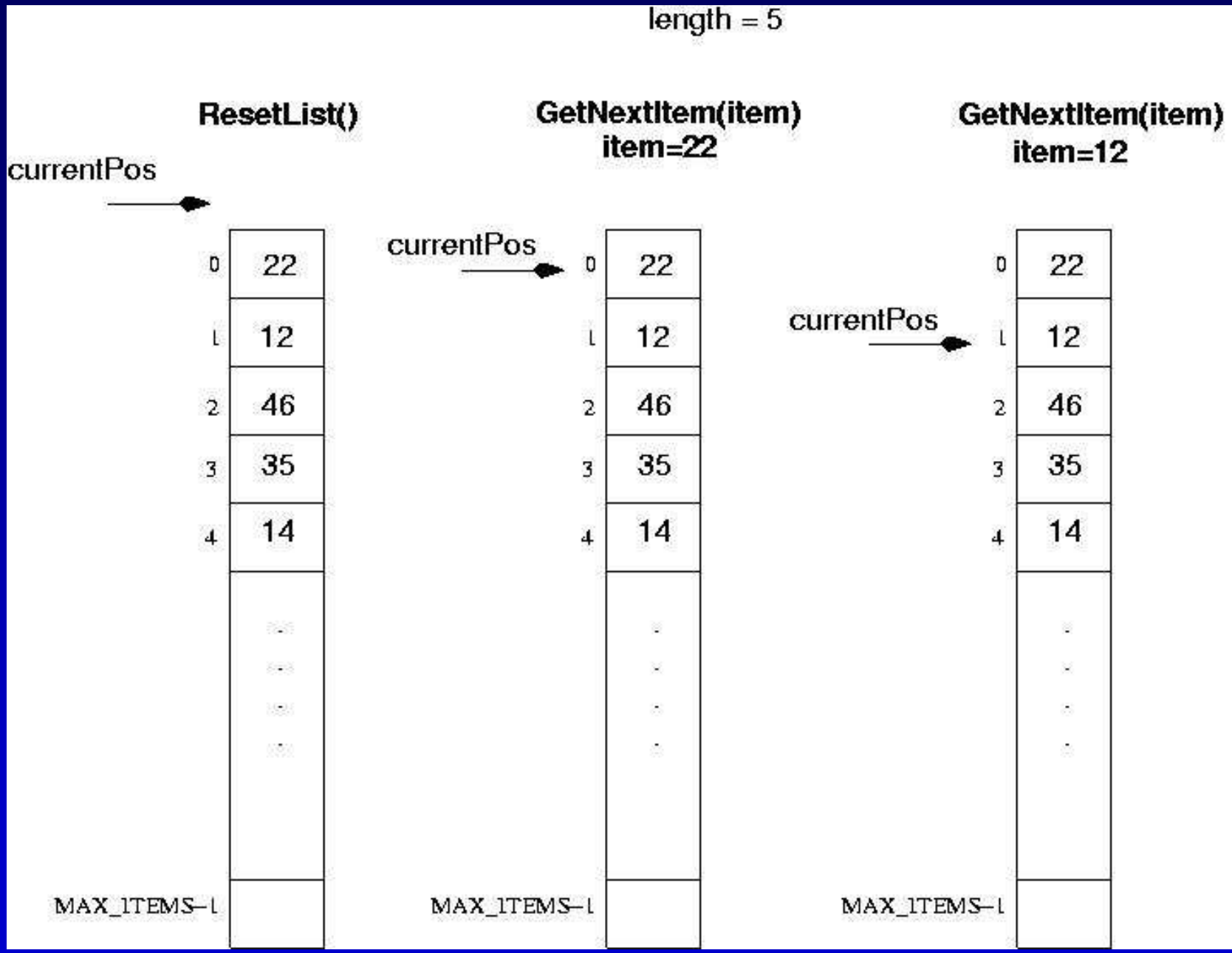
Traversing the List

- *ResetList*
- *GetNextItem*
- *IsLastItem*



ResetList

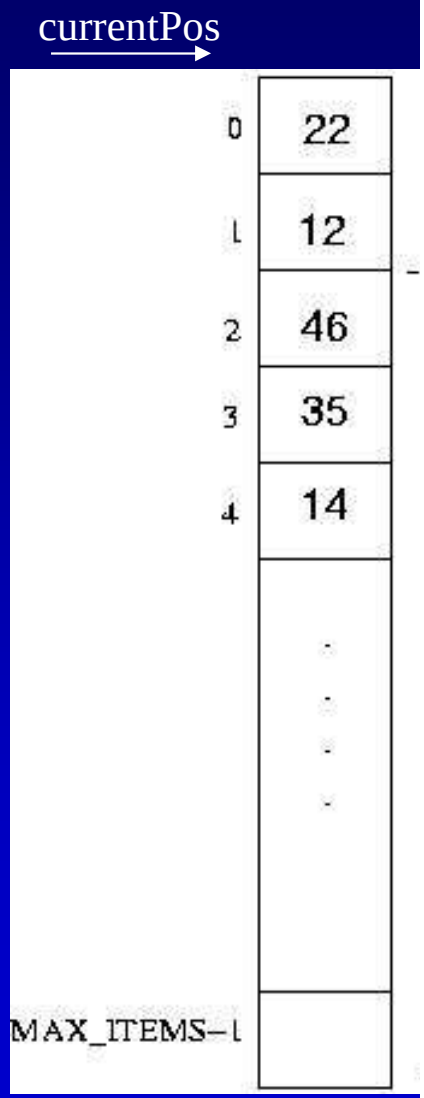
- *Function*: Initializes *currentPos* for an iteration through the list.
- *Preconditions*: List has been initialized
- *Postconditions*: Current position is prior to first element in list.


length = 5
ResetList()
currentPos
0
1
2
3
4
22
12
46
35
14
MAX_ITEMS–1
GetNextItem(item)
item=22
currentPos
0
1
2
3
4
22
12
46
35
14
MAX_ITEMS–1
GetNextItem(item)
item=12
currentPos
0
1
2
3
4
22
12
46
35
14
MAX_ITEMS–1

Array-based Implementation (cont.)

```
template<class ItemType>
void UnsortedType<ItemType>::ResetList()
{
 currentPos = -1;
}
```

O(1)



GetNextItem (ItemType& item)

- *Function*: Gets the next element in list.
- *Preconditions*: (1) List has been initialized, (2) Current position is not the last item.
- *Postconditions*: (1) Current position is updated to next position, (2) *item* is a copy of element at current position.

Array-based Implementation (cont.)

```
template<class ItemType>
void UnsortedType<ItemType>::GetNextItem (ItemType& item)
{
 currentPos++;
 item = info[currentPos];
}
```

O(1)

Array-based Implementation (cont.)

```
template<class ItemType>
bool UnsortedType<ItemType>::IsLastItem ()
{
 return (currentPos == length-1);
}
```

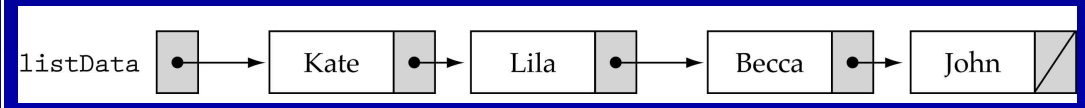
O(1)

Warning

- When traversing the list using GetNextItem, the list should not be modified (i.e., add or delete elements)
- This might modify the value of *currentPos*.

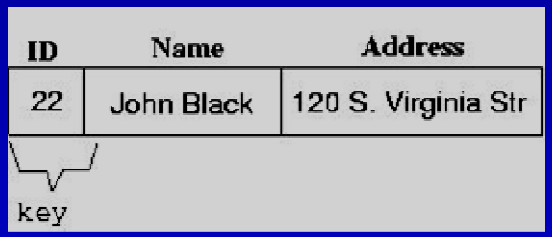
Linked-list-based Implementation

```
template <class ItemType>
struct NodeType;

template<class ItemType>
class UnsortedType {
  public:
    UnsortedType();
    ~UnsortedType();
    void MakeEmpty();
    bool IsFull() const;
    int LengthIs() const;
    void RetrieveItem(ItemType&, bool&);
    void InsertItem(ItemType);
    void DeleteItem(ItemType);
    void ResetList();
    bool IsLastItem() const;
    void GetNextItem(ItemType&);
  private:
    int length;
    NodeType<ItemType>* listData;
    NodeType<ItemType>* currentPos;
};
```



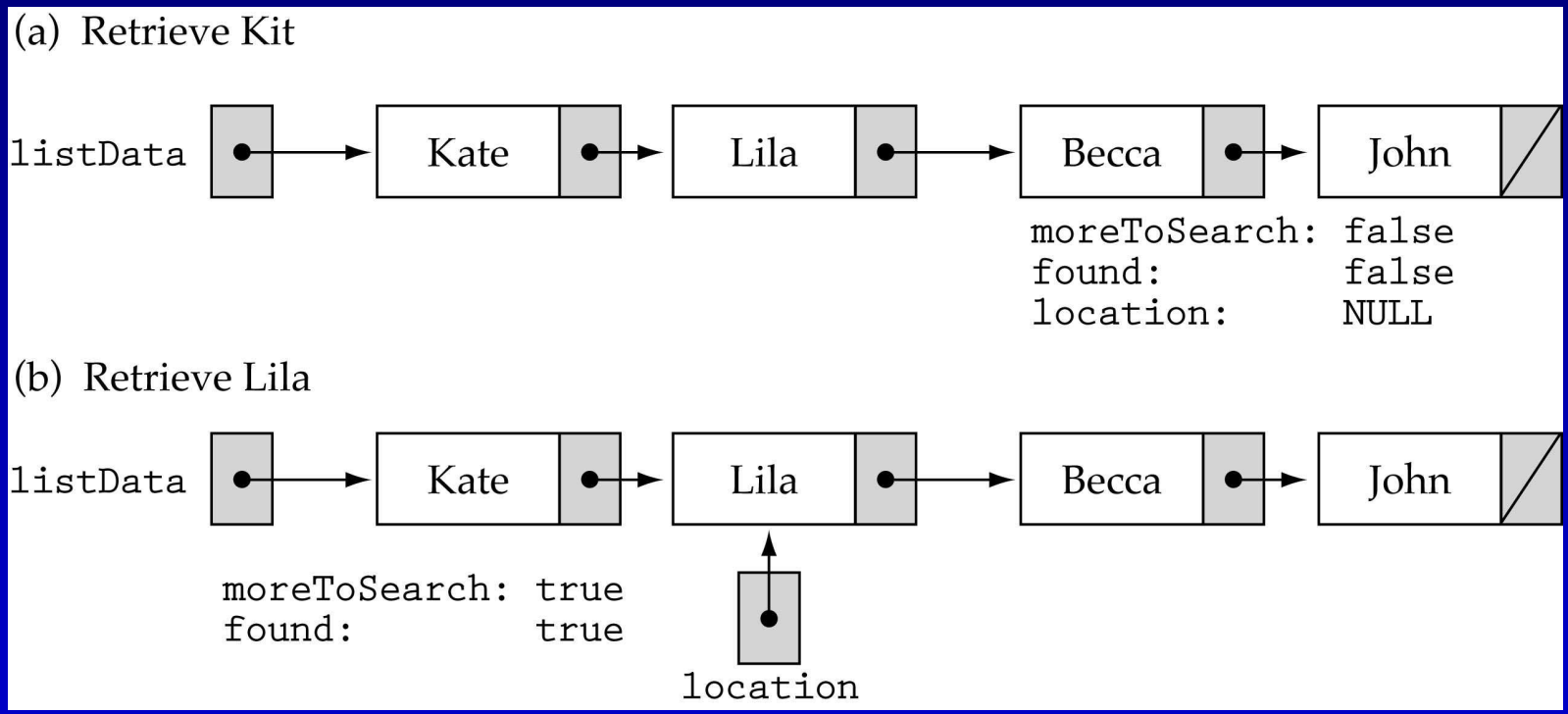
RetrieveItem (ItemType& item, boolean& found)

- ***Function***: Retrieves list element whose key matches item's key (if present).
- ***Preconditions***: (1) List has been initialized,
  (2) Key member of item has been initialized.



- ***Postconditions***: (1) If there is an element *someItem* whose key matches item's key, then *found=true* and *item* is a copy of *someItem*; otherwise, *found=false* and *item* is unchanged, (2) List is unchanged.

RetrieveItem



RetrieveItem (cont.)

```
template<class ItemType>
void UnsortedType<ItemType>::RetrieveItem
                                    (ItemType& item, bool& found)
{
 NodeType<ItemType>* location;

 location = listData;
 found = false;

 while( (location != NULL) && !found) {

   if(item == location->info) {
     found = true;
     item = location->info;
   }
    else
     location = location->next;
 }
}
```

O(N)

InsertItem (ItemType item)

- *Function*: Adds item to list

- *Preconditions*:

  (1) List has been initialized,

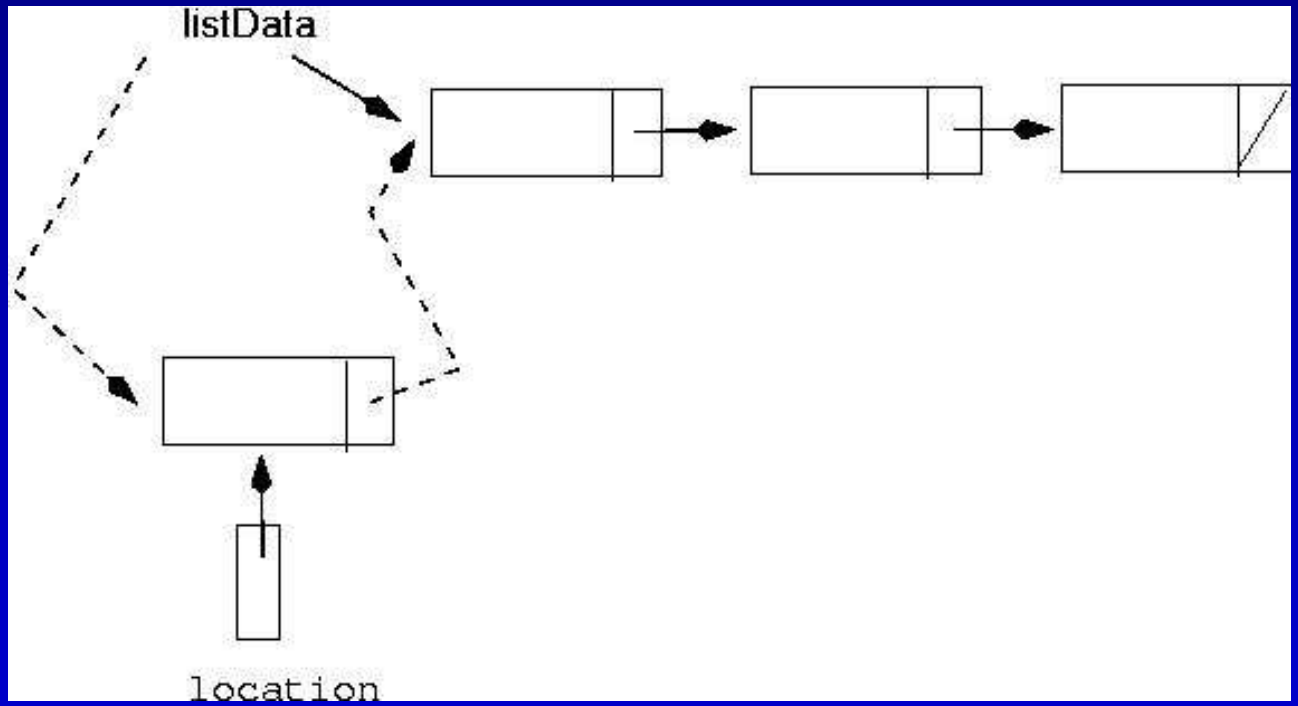
  (2) List is not full,

  (3) *item* is not in list (i.e., no duplicates)

- *Postconditions*: *item* is in list.

InsertItem

- Insert the item at the beginning of the list



InsertItem (cont.)

```
template <class ItemType>
void UnsortedType<ItemType>::InsertItem (ItemType newItem)
{
 NodeType<ItemType>* location;

 location = new NodeType<ItemType>;
 location->info = newItem;
 location->next = listData;
 listData = location;
 length++;
}
```

O(1)

DeleteItem (ItemType item)

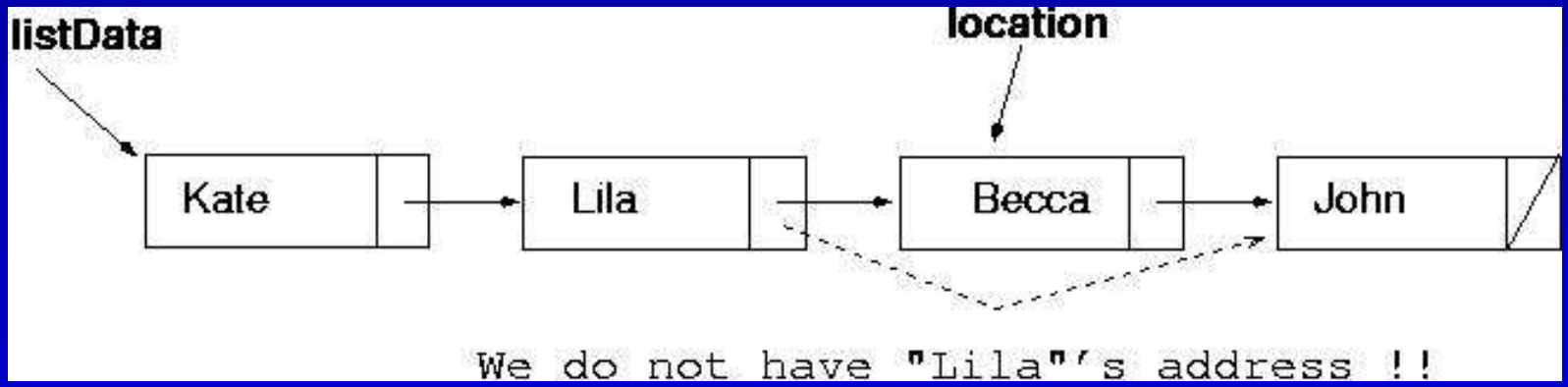
- *Function*: Deletes the element whose key matches *item's* key

- *Preconditions*: (1) List has been initialized, (2) Key member of *item* has been initialized, (3) There is only one element in list which has a key matching *item's* key.

- *Postconditions*: No element in list has a key matching *item's* key.

DeleteItem

- Find the item to be deleted.
- In order to delete it, we must change the "next" pointer in the *previous* node!

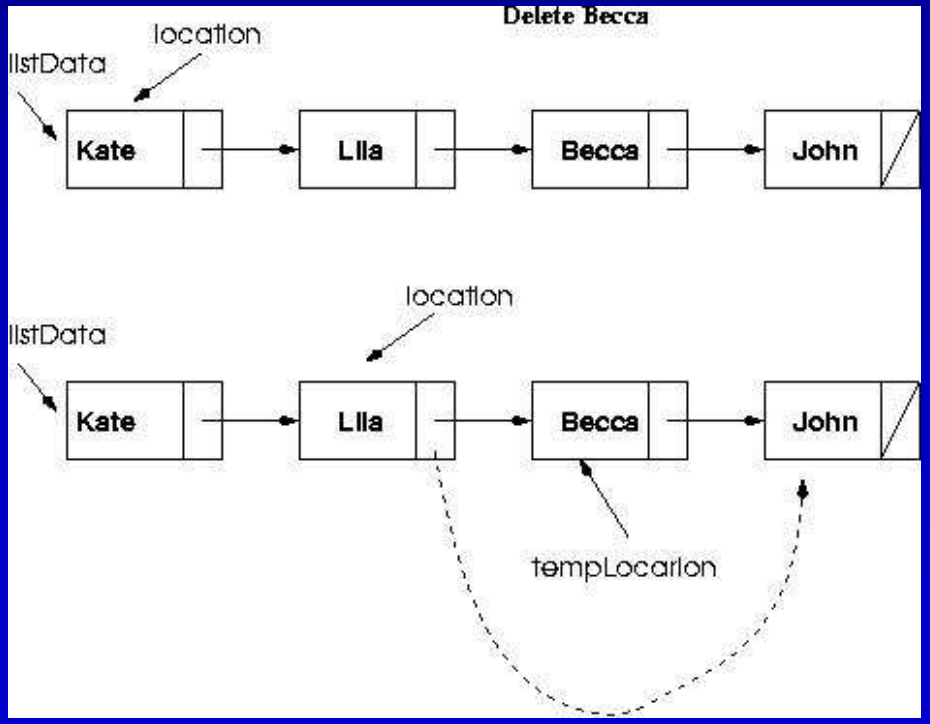


DeleteItem (cont.)

- Idea: compare **one item ahead**:

  i.e., *(location->next)->info)*

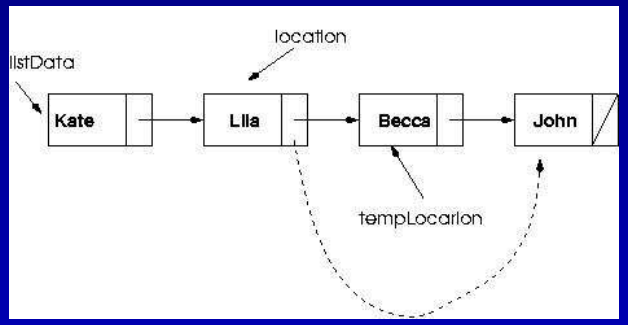
- Works efficiently since the item to be deleted is on the list (i.e., precondition).



Special case: deleting the first node!

Function DeleteItem (cont.)

```
template <class ItemType>
void UnsortedType<ItemType>::DeleteItem(ItemType item)
{
 NodeType<ItemType>* location = listData;
 NodeType<ItemType>* tempLocation;

 if(item == listData->info) {
   tempLocation = listData; // special case
   listData = listData->next;
 }                                          O(1)
 else {

   while(!(item == (location->next)->info))
     location = location->next;             O(N)
   // delete node at location->next

   tempLocation=location->next;
   location->next = tempLocation->next;
 }                                          O(1)
```




```
 delete tempLocation;
 length--;
}
```

O(1)

Overall: O(N)

Other UnsortedList functions

```
template<class ItemType>
UnsortedType<ItemType>::UnsortedType()
{
 length = 0;
 listData = NULL;
}
```

O(1)

```
template<class ItemType>
void UnsortedType<ItemType>::MakeEmpty()
{
 NodeType<ItemType>* tempPtr;

 while(listData != NULL) {
   tempPtr = listData;
   listData = listData->next;
   delete tempPtr;
 }
 length=0;
}
```

O(N)

Other UnsortedList functions (cont.)

```
template<class ItemType>
UnsortedType<ItemType>::~UnsortedType()
{
 MakeEmpty();
}
```

O(N)

```
template<class ItemType>
bool UnsortedType<ItemType>::IsFull() const
{
 NodeType<ItemType>* ptr;
 ptr = new NodeType<ItemType>;
 if(ptr == NULL)
   return true;
 else {
   delete ptr;
   return false;
 }
}
```

O(1)

Other UnsortedList functions (cont.)

```
template<class ItemType>
int UnsortedType<ItemType>::LengthIs() const
{
 return length;
}
```

O(1)

```
template<class ItemType>
int UnsortedType<ItemType>::ResetList()
{
 currentPos = listData;
}
```

O(1)

Other UnsortedList functions (cont.)

```
template<class ItemType>
void UnsortedType<ItemType>::GetNextItem(ItemType& item)
{
 item = currentPos->info;
 currentPos = currentPos->next;
}
```

O(1)

```
template<class ItemType>
bool UnsortedType<ItemType>::IsLastItem() const
{
 return(currentPos == NULL);
}
```

O(1)

Comparing unsorted list implementations

Big-O Comparison of Unsorted List Operations		
Operation	Array Implementation	Linked Implementation
Constructor	O(1)	O(1)
Destructor	O(1)	O(N)
MakeEmpty	O(1)	O(N)
IsFull	O(1)	O(1)
LengthIs	O(1)	O(1)
ResetList	O(1)	O(1)
GetNextItem	O(1)	O(1)
RetrieveItem	O(N)	O(N)
InsertItem	O(1)	O(1)
DeleteItem	O(N)	O(N)