Lists: An ADT for Sequential Data Organization

CMPT 115/117 lecture slides

Notes written by Mark Eramian, Ian McQuillan, Michael Horsch, Lingling Jin, and Dmytro Dyachuk

Objectives

After this topic, students are expected to

1. Describe the difference between lists and arrays.
2. Describe the typical list operations informally.
3. Define formal algorithm headers for list operations.
4. Draw the diagrams of the data structure of array-based lists as well as linked lists.
5. Describe the implementation of typical list operations in pseudocode.
6. Analyze the time complexities of typical list operations for array-based implementations and linked implementations.
7. Critically assess the decision to apply either of the two list implementations in different applications based on their properties of the data structures and the complexities of their operations.

Introduction: Lists

- A *list* is a sequentially organized collection of data.
- Examples: A list of dates, a list of student records, a list of webpages, a list of songs, any real data.
- The list contains data *elements*, e.g., student records, webpages, songs.
- A list allows elements to be inserted, removed, searched-for, modified.
- A list is an abstract design that can be implemented several ways.

Ordered vs. Unordered Lists

- A list is a sequence with a beginning (the head) and an end (the tail).
- Unordered lists: no sequence relationship between elements.
- Ordered lists: some attribute of the elements, called a *key*, determines the ordering.

Example (A unordered list of integers)

$\boxed{32}$ $\boxed{987}$ $\boxed{42}$ $\boxed{77}$ $\boxed{135}$

Example (An ordered list of characters)

$\boxed{A}$ $\boxed{B}$ $\boxed{D}$ $\boxed{K}$ $\boxed{Q}$ $\boxed{R}$ $\boxed{Y}$ $\boxed{Z}$

More about Keys for Ordered Lists

- In ordered lists, each element has a *key* that determines the proper order.
- For atomic data elements, the key of an element is simply the element itself, e.g. integers, characters.
- For compound data elements, one data field of the element is usually chosen as the key.

  E.g. if the elements of a list are the structure:

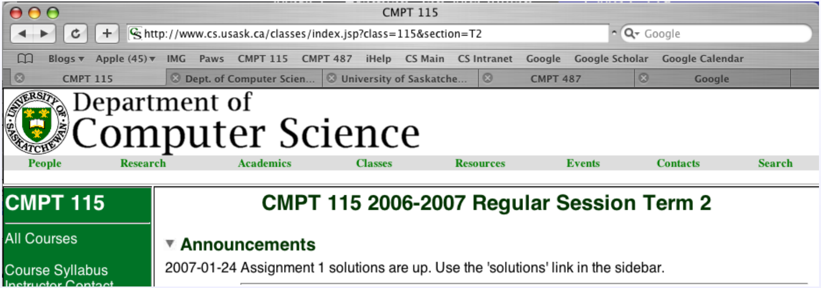
  ```
  Name
      refToChar firstName
      refToChar lastName
  end Name
  ```

  the `lastName` field might be used as the key; the list will be ordered by last name.
- We will sometimes call the rest of the data element, other than the key, the *satellite* data.

Why not just use arrays for sequential data?

1. Arrays have a fixed size; we may have unlimited amount of data.
2. Arrays can be used in unconstrained ways; an unneeded intellectual burden.

Tabbed Browsing



- Each tab stores a URL to be displayed (among other data).
- Need arbitrary number of tabs.
- User can delete any tab at any time.
- User can create a new tab at any time.
- When the browser window is closed, the list is destroyed (its memory recycled).

Simple Text Editor

- In early computer systems, we used *line editors* to edit text files.
- Let's see the Linux line editor, `ed` in action!
- A line editor can be implemented using a list of strings (character arrays) by storing one string per line of the text file in the list.
- Line editor commands:
  - insert line
  - delete line
  - edit line
  - etc
- Each line editor operation can be implemented using list operations.

Memory Management

- Some operating systems use lists to manage and organize the free memory blocks available on the heap.
- Memory managers use a *free list*, where each element consists of:
  - The starting address of a memory block.
  - The size of the memory block.
- Elements are ordered by starting address.
- When programmer calls `new`, the memory system traverses the free list, and finds the smallest block of available memory that can fulfill the request.
- Elements can be split into two nodes to make smaller blocks, or adjacent elements can be combined to make larger blocks, etc.
- Details of other exciting operating system mechanics such as this can be learned in CMPT 332 and 432.

Types of Operations: A first look

- Insert a new element into a list.
- Delete an element from a list.
- Retrieve (search for and return) an element in the list.
- Create a new empty list.
- Destroy the list, and any elements in it.
- Check if a list is empty or not.
- Find out how many elements are in the list.

Insertion

Example (Insertion into a unordered list of integers)

Insert 66

66 32 987 66 42 77 135 66

New element is permitted anywhere, but typically is inserted at one end or the other.

Example (Insertion in an ordered list of characters)

Insert 'H'

A B D H K Q R Y Z

The proper order must be maintained.

Deletion

- The list must be searched to locate the element to be deleted.
- Once found, it is removed from the list.

Example (Deleting a list element.)

Delete 42

$\boxed{32}\quad\boxed{987}\quad\overset{\downarrow}{\boxed{42}}\quad\boxed{77}\quad\boxed{135}\quad\Rightarrow\quad\boxed{32}\quad\boxed{987}\quad\boxed{77}\quad\boxed{135}$

- Note that there is no empty space left behind by the deleted element.
- How we search for the element will depend on how we implement the list.

Retrieval

- Retrieval requires that the desired element in a list be located, and given to the calling module *without modifying the contents of the list*.
- A search algorithm is used to find the desired element.

Variations for unordered lists

Depending on the application, it may be convenient to use specialized variations:

1. Insert at the *head* of the list
2. Insert at the *tail* of the list
3. Delete from the *head* (or *tail*) of the list
4. Retrieve from *head* (or *tail*) of the list

For these, the *position* is used, not any key.

List Operations - Summary

- For our list ADT, we will permit the following operations.
  - Create a new list
  - Retrieve an element (by key, or position)
  - Insert an element
  - Delete a given element (by key, or position)
  - Test if a list is empty
  - Get number of elements in the list
  - Destroy a list (and any contents)
- These operations can be understood by their intended effect, without knowing their implementation.

Formal algorithm headers for list operations

- An application programmer uses ADTs knowing only the interface, not the implementation.
- An ADT programmer knows only the ADT implementation, not the application.
- This division of labour is crucial to CERAR principles, even if you are on both sides of the interface.
- The following slides give typical algorithm headers for list operations. The same headers can often be used for many different implementations.
- Note: the header for some operations (e.g., CreateList) may change slightly depending on the implementation.

Creating and Destroying a List

Algorithm *CreateList*(*size*)
Create a new list with a given capacity.

Pre: *size* :: the capacity of the array
Post: memory is allocated from the heap to store a List.
Return: a reference to a newly allocated empty list.

Algorithm *DestroyList*(*rList*)
All data stored in the list is destroyed; $O(size)$

Pre: *rList* :: reference to a list to deallocate.
Post: *rList* is deallocated; all elements deallocated.

Retrieving a List Element

Algorithm *RetrieveElement*(*rList*, *target*, *el*)
Retrieve an element with a given target key from the list, copying it into **el*.

Pre: *rList* :: a reference to the list from which to retrieve.
target :: the key of the element to retrieve.
el :: a reference to a variable of type Element
in which to store the retrieved element
Post: copy of the element with key *target* placed in **el*;
or **el* undefined if *target* not found.
Return: **true** if successful, **false** if *target* not found

Inserting an element

Algorithm *InsertTail*(*rList*, *el*)
Put given element into the given list at the tail.

Pre: *rList* :: a reference to a list into which to insert
el :: an Element
Post: *el* is inserted into the list at the tail.
Return: **true** if successful, **false** otherwise

Algorithm InsertHead(rList, el)
Put given element into the given list at the head.

Pre: *rList* :: a reference to a list into which to insert
el :: an Element
Post: *el* is inserted into the list at the head
Return: **true** if successful, **false** otherwise

Deleting from a List

Algorithm *DeleteElement*(*rList*, *target*, *el*)
Removes element matching given target, from the list, copying it to **el* first.

Pre: *rList* :: reference to a list to delete from
target :: key of the element to be deleted
el :: reference to an allocated element
Post: Contents of element stored in **el*
first node containing el is removed from the list if such a node exists.
Return: **true** if a node was deleted, otherwise **false**.

Algorithm *DeleteTail*(*rList*, *el*)
Removes the element at the tail, copying it into **el* first.

Pre: *rList* :: a List
el :: a reference to an Element
Post: Last element is deleted from *rList*. A copy of the element is stored in **el*.
Return: **true** if success, otherwise **false**.

Checking the size of a list

Algorithm *ListIsEmpty*(*rList*)
Checks if the list is empty or not.

Pre: *rList* :: a reference to a list
Return: **true** if *rList* has no elements in it; otherwise **false**.

Algorithm *ListCount*(*rList*, *el*)
Returns the number of elements currently stored in the list.

Pre: *rList* is a list
Return: an **Integer** value representing the number of elements in *rList*.

Using the List Interface

Using the interface is easy: just normal function calls.

```
refToList myList ← CreateList(1000)  //Create the List

Insert(myList, 3)                    //Insert into the list
Insert(myList, 9)
Insert(myList,27)
Insert(myList,81)

Integer num                          //retrieve from the list
if (RetrieveElement(myList, 27, &num))
then print "in the list"
else print "not in the list"
end

DeleteElement(myList, 9, &num)       //delete from the list

DestroyList(myList)                  //destroy the list
```

Using the List ADT

- When using an ADT, only use the operations provided.
- Never work directly with the data, even if you know how.
- Using the ADT operations guarantees that the data stored is always consistent.
- Assuming that the ADT is correct and robust, of course!
- Clever tricks and stop-gap measures that evade the ADT operations lead to failure, and are indicators of poor ADT design.

Looking forward: Two implementations

In the following, we will see two implementations of the List ADT:

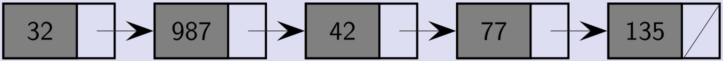
1. Array Representation

Example

0	1	2	3	4
32	987	42	77	135

2. Linked Nodes Representation

Example



Lists using a simple array

- An array has a known capacity. But a list can vary in size. It can even be empty!
- So we need to keep track of how many elements are in the list.
- We will assume an unordered list. We will add elements at the far end of the list.
- The size of the list will index the *next* available location for new elements.
- When we delete an element, we have to shift all the elements after it.
- Retrieval will use linear search.
- This example is not really very good. But it lets us build on what we know.

Data Structure for Array-based List

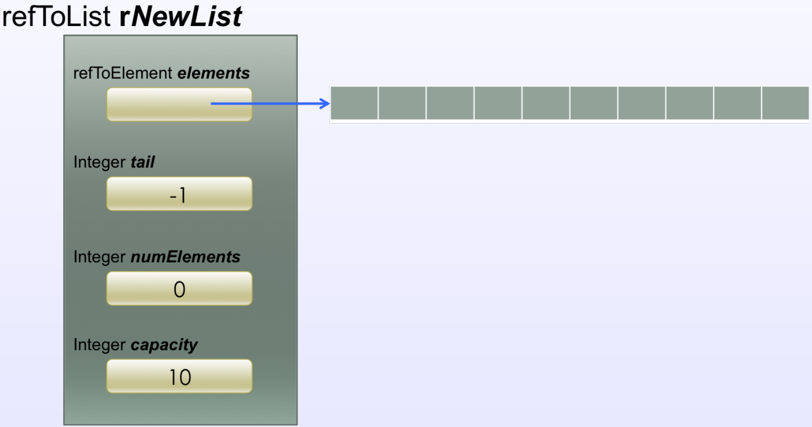
While the array can store the elements, we need other information too.

1. Index of the tail
2. Number of elements currently in the array
3. Total capacity of the array

```
List
    refToElement elements    // Array of elements
    Integer tail             // Index of tail
    Integer numElements      // size of the list
    Integer capacity         // Max number of elements
end List
```

CreateList

`CreateList` operation would create an empty list:



CreateList

Algorithm CreateList(size)
Create a new list.
Pre: *size* :: the capacity of the array
Returns: a reference to a newly allocated list that is initialized to be empty.

refToList rNewList $\leftarrow$ **allocate new** *List*;
rList $\Rrightarrow$ *capacity* $\leftarrow$ size
rList $\Rrightarrow$ *tail* $\leftarrow$ -1
rList $\Rrightarrow$ *numElements* $\leftarrow$ 0
rList $\Rrightarrow$ *elements* = **allocate new** Element [size]
return *rNewList*

Retrieving a List Element

- Retrieval of an element returns a copy of an element in the list. The element is identified by its Key.

```
Algorithm RetrieveElement(rList, target, el)
Pre: rList :: a reference to the list from which to retrieve.
     target :: the key of the element to retrieve.
     el :: a reference to a variable of type Element
         in which to store the retrieved element
Post: copy of the element with key target placed in *el
      or *el undefined if target not found.
Returns: true if successful, false if target not found

for current from 0 to rList ↠ numElements
    if (target == key of rList ↠ elements[current])
        *el ← rList ↠ elements[current]
        return true
    end if
end
return false
```

Using `RetrieveElement`

Suppose `Element` is the type `Integer`.

> *Integer num*
> *RetrieveElement*(*rList*, 42, &*num*)

- A copy of the element 42 is placed into `num` if it is in the list.

Suppose `Element` is the type `Person`, and the key is the `lastName` field of `Person`.

> *Person aPersonRecord*
> *RetrieveElement*(*rList*, "Smith", &*aPersonRecord*)

- If there is a record in the list with last name Smith, a copy of the *entire* `Person` structure is placed into `aPersonRecord`.

Inserting at the End

Algorithm *InsertTail*(*rList*, *el*)

Pre: *rList* :: a reference to a list into which to insert
el :: an Element
Post: *el* is inserted into the list
Return: **true** if successful, **false** otherwise

if ($rList \Rrightarrow numElements == rList \Rrightarrow capacity$)
  return false // Special case when list is full
else
  // put the new element in the position indexed by numElements
  $rList \Rrightarrow elements[numElements] \leftarrow el$
  $rList \Rrightarrow numElements \leftarrow rList \Rrightarrow numElements + 1$
  $rList \Rrightarrow tail \leftarrow rList \Rrightarrow tail + 1$
end if

return true

InsertHead

Algorithm InsertHead(rList, el)

Pre: *rList* is a reference to a list into which to insert
el is an Element
Post: *el* is inserted as the first element of the list
Return: **true** if successful, **false** otherwise

if ($rList \twoheadrightarrow numElements = rList \twoheadrightarrow capacity$)
  return false
end if

// Shift each element of the elements array one index to the right.
$i \leftarrow rList \twoheadrightarrow numElements$
while($i > 0$)
  $rList \twoheadrightarrow elements[i] \leftarrow rList \twoheadrightarrow elements[i-1]$
  $i \leftarrow i - 1$
end while

$rList \twoheadrightarrow elements[0] \leftarrow el$

$rList \twoheadrightarrow tail \leftarrow rList \twoheadrightarrow tail + 1$
$rList \twoheadrightarrow numElements \leftarrow rList \twoheadrightarrow numElements + 1$

return true

Deleting from a List

Algorithm *DeleteElement*(*rList*, *target*, *el*)
Removes data for given target, from the list, storing it in **el*

Pre: *rList* :: reference to a list to delete from
target :: key of the element to be deleted
el :: reference to an element
Post: Contents of node stored in **el*
first node containing el is removed from the list if such a node exists.
Returns: **true** if a node was deleted, otherwise **false**.

```
current ← 0
while ( current < rList⤃numElements AND key of rList⤃elements[current] != target )
   current ← current + 1
end
if ( current == (rList⤃numElements )
    return false
else
   *el ← rList⤃elements[current]
    rList⤃numElements ← rList⤃numElements - 1
    while ( current < rList⤃numElements)
          rList[current] ← rList[current+1]
          current ← current + 1
    end
    return true
end if
```

DeleteTail

Algorithm *DeleteTail*(*rList*, *el*)
Pre: *rList* is a list
el is a reference to an Element
Post: Last element is deleted from *rList*. A copy of the element is stored in **el*.
Return: **true** if success, otherwise **false**.

if ($rList \Rrightarrow tail$ = -1)
  return false
end if

$*el \leftarrow rList \Rrightarrow elements[rList \Rrightarrow tail]$
$rList \Rrightarrow tail \leftarrow rList \Rrightarrow tail - 1$

$rList \Rrightarrow numElements \leftarrow rList \Rrightarrow numElements - 1$
return true

Destroying a List

Algorithm *DestroyList*(*rList*)
All data stored in the list is destroyed

Pre: *rList* :: reference to a list to destroy
Post: *rList* is deallocated

deallocate $rList \Leftrightarrow elements$
deallocate *rList*

Time Complexity of List Operations

Operation	Worst Case Time	Best Case Time
CreateList	O(1)	O(1)
RetrieveElement	O(n)	O(1)
InsertHead		
InsertTail		
InsertAfter		
DeleteHead		
DeleteTail		
DeleteElement		
ListIsEmpty	O(1)	O(1)
ListCount	O(1)	O(1)
DestroyList		

n is the number of elements in the list.

Time Complexity of List Operations

Operation	Worst Case Time	Best Case Time
`CreateList`	O(1)	O(1)
`RetrieveElement`	O(n)	O(1)
`InsertHead`	O(n)	O(n)
`InsertTail`	O(1)	O(1)
`InsertAfter`	O(n)	O(n)
`DeleteHead`	O(n)	O(n)
`DeleteTail`	O(1)	O(1)
`DeleteElement`	O(n)	O(n)
`ListIsEmpty`	O(1)	O(1)
`ListCount`	O(1)	O(1)
`DestroyList`	O(1)	O(1)

n is the number of elements in the list.

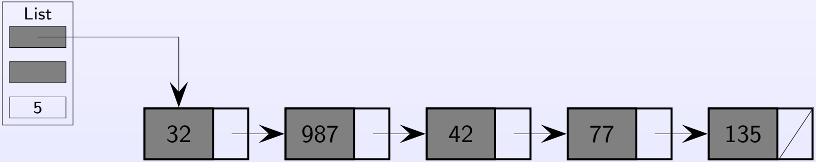
Summary

- Moving things around (DeleteElement) is expensive, even at $O(n)$.
- The array has a fixed capacity. It might be way too big, or too small. We may not know exactly how big we need a list to be.
- We can do better if we use the heap to allocate storage for each element that we need. This requires a bit more sophistication. See next section!

We can do better with a Linked List

Aim: No size limit
Faster deletes

Complication: Requires an extra record type, and more complicated code



Linked Lists

- In a linked list, each element is stored in a *node*.
- A node consists of the element in that node, and a reference to the next node (or NULL if there is no next element).
- This design allows us to create storage for each individual thing we want to insert, exactly when we need it.
- The reference to the next node allows us to create a sequence of nodes, which is the defining characteristic of a list.

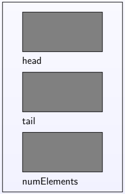
Data Structures: Node record type to link things together

```
Node
    Element data;         // placeholder type
                         // that can be anything we want
    refToNode next;   // points to another node
end Node
```

- A `node` record can link one node to another node.
- We'll need a node record for each element in a list.
- `Element` is a placeholder type which can be anything – but all elements in the list are of the the same type.

Data Structures: List record type to store key nodes

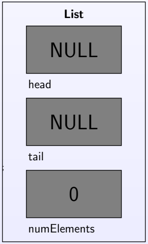
List
  refToNode head; // A reference to the first node.
  refToNode tail; // A reference to the last node.
  Integer numElements; // Number of elements in the list.
end *List*



- A list of elements has one list record, but many node records.
- The list record stores key information about a list:
    - references to the first and last node
    - The number of elements
    - sometimes other information as well
- The elements in the list are accessible through the head.
- A list record also helps separate the data from an application program.

Creating a new List record

A new list has no data in it.



Algorithm *CreateList*()
Create a new list.
Returns: a reference to a newly allocated list record
The record represents an empty list.

refToList rNewList ← **allocate new** *List*
rNewList ⤃ *head* ← **NULL**
rNewList ⤃ *tail* ← **NULL**
rNewList ⤃ *numElements* ← 0
return *rNewList*

Returning the size of the list

Algorithm *ListCount*(*rList*)
Returns number of elements in the list.

Pre: *rList* :: reference to list
Post: list is unchanged
Returns: number of elements in list

return *rList* $\Rrightarrow$ *numElements*

If we did not have the number of elements defined as part of the list record type, how could you determine the size?

Checking if List is Empty

Algorithm *ListIsEmpty*(*rList*)
Determines if a list is empty.

Pre: *rList* is a reference to the list to test for emptiness
Returns: **true** if list is empty, otherwise **false**

if ($rList \Rrightarrow numElements == 0$)
  return true
else
  return false
end if

Can you think of another way to implement it?

Destroying a List

Algorithm *DestroyList(rList)*
Deletes a list.

Pre: *rList* :: reference to a list.
Post: All data is deleted and memory returned to the heap.

// Delete the list data and free memory
refToNode rWalker, rTemp
rWalker $\leftarrow$ *rList* $\Rrightarrow$ *head*
while(*rWalker* != **NULL**)
    rTemp $\leftarrow$ *rWalker*
    rWalker $\leftarrow$ *rWalker* $\Rrightarrow$ *next*
    deallocate *rTemp*
end while

// Now deallocate the list structure itself
deallocate *rList*

- Why do we need the `rTemp` variable?
- What if we just wanted to empty a list instead of destroy it?

Search for a List Element

- Simplified: Assume the an element is an integer

Algorithm *SearchList*(rList, target)
Pre: *rList* :: a reference to the list from which to retrieve.
target :: the value to search for
Returns: **true** if *target* found; **false** if *target* not found

refToNode current
$current \leftarrow rList \Rrightarrow head$
while(*current* != **NULL**)
  if (*target* == $current \Rrightarrow data$)
    return true
  end if
  $current \leftarrow current \Rrightarrow next$
end
return false

Search for a List Element

- Advanced: An element could be anything (a record, say)
- Assume that each record has a key value we can look for

Algorithm *SearchList*(rList, target)
Pre: *rList* :: a reference to the list from which to retrieve.
target :: the value to search for
Returns: **true** if *target* found; **false** if *target* not found

refToNode current
current $\leftarrow$ *rList* $\Rrightarrow$ *head*
while(*current* != **NULL**)
    if (*target* == *key* of *current*$\Rrightarrow$*data*)
        return true
    end if
    current $\leftarrow$ *current*$\Rrightarrow$*next*
end
return false

Retrieving a List Element

- Retrieval of an element returns a copy of an element in the list. The element is identified by its Key.

Algorithm *RetrieveElement*(rList, target, el)
Pre: *rList* :: a reference to the list from which to retrieve.
    target :: the key of the element to retrieve.
    el :: a reference to a variable of type Element
        in which to store the retrieved element
Post: copy of the element with key *target* placed in **el*
    or **el* undefined if *target* not found.
Returns: **true** if successful, **false** if *target* not found

```
refToNode current
current ← rList ⤃ head
while(current != NULL)
    if (target == key of current⤃data)
        *el ← current⤃data
        return true
    end if
    current ← current⤃next
end
return false
```

Inserting at the Beginning

Algorithm *InsertHead*(*rList*, *el*)

Pre: *rList* :: reference to a list into which to insert
el :: an Element
Post: *el* is inserted as the first element of the list
Return: **true** if successful, **false** otherwise

refToNode rNew $\leftarrow$ **allocate new** *Node*
if (*rNew* == **NULL**)
  return false
end if

rNew $\nRightarrow$ *data* $\leftarrow$ *el*
rNew $\nRightarrow$ *next* $\leftarrow$ *rList* $\nRightarrow$ *head*
rList $\nRightarrow$ *head* $\leftarrow$ *rNew*

if (*rList* $\nRightarrow$ *tail* == **NULL**)
  rList $\nRightarrow$ *tail* $\leftarrow$ *rNew*
end if

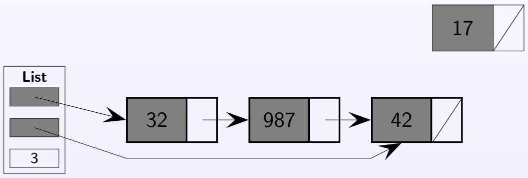
rList $\nRightarrow$ *numElements* $\leftarrow$ *rList* $\nRightarrow$ *numElements* + 1

return true

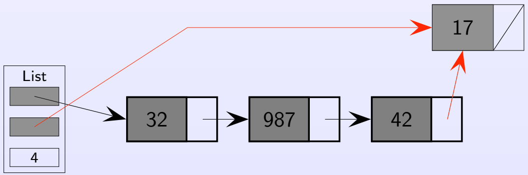
Inserting into a List

Insert at the end.

Before:



After:



Inserting at the End

Algorithm *InsertTail*(*rList*, *el*)

refToNode rNew ← **allocate new** *Node*
if (*rNew* == **NULL**)
    return false
end if

rNew ⤖ *data* ← *el*
rNew ⤖ *next* ← **NULL**

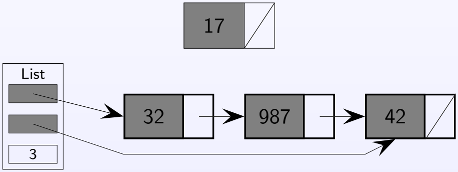
// Special case when list is empty
if (*rList* ⤖ *tail* == **NULL**)
    rList ⤖ *head* ← *rNew*;
    rList ⤖ *tail* ← *rNew*;
else
    // Previous last node must point to new last node
    rList ⤖ *tail* ⤖ *next* ← *rNew*;
    rList ⤖ *tail* ← *rNew*
end if

rList ⤖ *numElements* ← *rList* ⤖ *numElements* + 1
return true

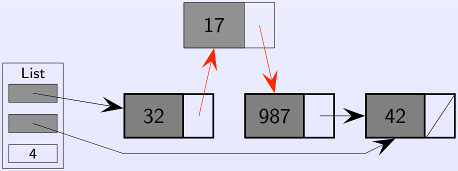
Inserting into a List

Arbitrary Insertion

Before:



After:



Inserting into a List

Insert After an Element

Algorithm *InsertAfter*(*rList*, *target*, *el*)
Inserts element into a list after a given target element.

Pre: *rList* :: a reference to the list into which to insert
el :: element to insert
target :: the key of the element after which el is to be inserted
Post: *el* is a member of the list and its predecessor is the element with key *target*. If no element with key *target* exists, *el* is placed at the end of the list.
Returns: **true if** successful, otherwise **false**.

```
// Find the element with key 'el'
refToNode rPre ← NULL
current ← rList ⤖ head
while( current != NULL )
    if (target == key of current ⤖ data)
        rPre ← current
        break
    end if
    current ← current ⤖ next
end while

// continued next slide...
```

Inserting into a List
Insert After an Element

```
// ... continued from previous slide

// If element with key 'target' was not found...
if ( rPre == NULL )
    InsertTail(rList, el)
else
    // Otherwise, insert after the found element
    rNew ← allocate new Node
    if (rNew == NULL)
        return false
    end if

    rNew ⤃ data ← el
    rNew ⤃ next ← rPre ⤃ next
    rPre ⤃ next ← rNew

    // If inserting after last element, adjust tail
    if ( rPre == rList ⤃ tail )
        rList ⤃ tail ← rNew
    end if
    rList ⤃ numElements ← rList ⤃ numElements + 1
end if

return true
```

Deleting from a List

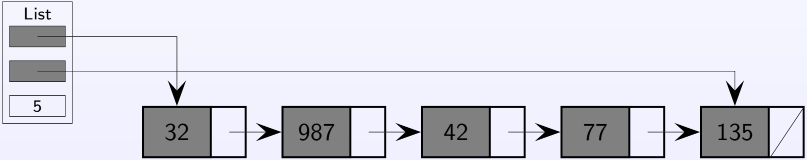
- If we delete a node n, we must link n's predecessor to n's successor.
- A couple of special cases arise when deleting the first, or last element.

Deleting from a List

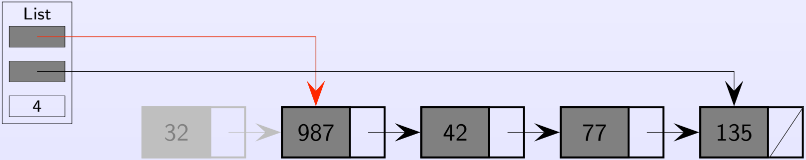
Deleting the First Element

Deleting first element.

Before:



After:



Deleting the First Element

Algorithm *DeleteHead*(*rList*, *el*)
Pre: *rList* :: a list
el :: a reference to an Element
Post: First element is deleted from *rList*. A copy of the element is stored in **el*.
Returns: **true** if success, otherwise **false**.

if ($rList \Rrightarrow head$ == **NULL**)
    return false
end if

$*el \leftarrow rList \Rrightarrow head \Rrightarrow data$
$temp \leftarrow rList \Rrightarrow head$
$rList \Rrightarrow head \leftarrow rList \Rrightarrow head \Rrightarrow next$

// Deallocate the deleted node.
deallocate *temp*

// If we deleted the only element, adjust tail.
if ($rList \Rrightarrow head$ == **NULL**)
    $rList \Rrightarrow tail \leftarrow$ **NULL**
end if

$rList \Rrightarrow numElements \leftarrow rList \Rrightarrow numElements$ - 1
return true

Deleting the Last Element

Algorithm *DeleteTail*(*rList*, *el*)
Pre: *rList* :: a list
el :: a reference to an Element
Post: Last element is deleted from *rList*. A copy of the element is stored in **el*.
Return: **true** if success, otherwise **false**.

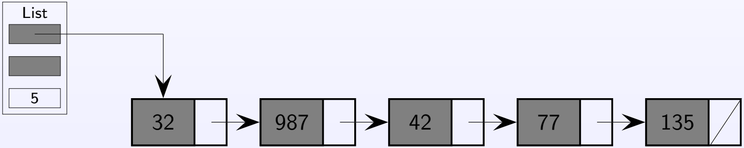
...

Deleting from a List

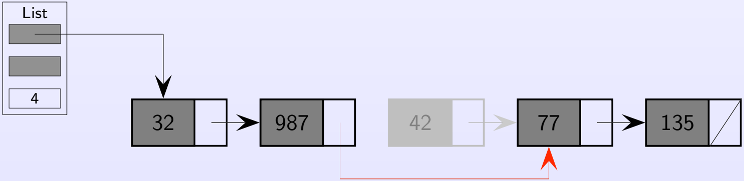
Deleting other Elements

General deletion.

Before:



After:



Deleting from a List

Algorithm *DeleteElement*(*rList*, *target*, *el*)
Deletes a node.

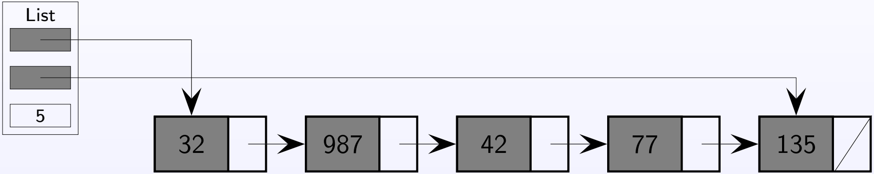
Pre: *rList* :: reference to a list to delete from
target :: key of the element to be deleted
el :: reference to an element
Post: Contents of node stored in **el*
first node containing el is removed from the list if such a node exists.
Returns: **true** if a node was deleted, otherwise **false**.

```
refToNode rPre, rTmp

if ( rList ⤃ head == NULL )
    return false
end if

// Search for element with key 'target'
rPre ← NULL
rTmp ← rList ⤃ head

while( rTmp != NULL )
    if ( target == key of rTmp ⤃ data )
        *el ← rTmp ⤃ data;
        break
    end if

    rPre ← rTmp
    rTmp ← rTmp ⤃ next
end while

// continued next slide...
```

Deleting from a List

```
// ... continued from previous slide

if (rTmp == NULL)
    return false
else if (rPre == NULL and rTmp != rList ⤀ tail)
    // deleting first node which is not the only node
    rList ⤀ head ← rList ⤀ head ⤀ next
else if (rPre == NULL)
    // deleting first node which is the only node
    rList ⤀ head ← NULL
    rList ⤀ tail ← NULL
else if ( rTmp == rList ⤀ tail )
    //deleting last node
     rList ⤀ tail ← rPre
     rPre ⤀ next ← NULL
else
    // Deleting other nodes
    rPre ⤀ next ← rTmp ⤀ next
end if

end if

deallocate rTmp

rList ⤀ numElements ← rList ⤀ numElements - 1
return true
```

Deleting from a List



Element DataOut
// Delete the first element
DeleteHead(*rList*, &*DataOut*)

// Delete 42
DeleteFromList(*rList*, 42, &*DataOut*)

What does DataOut contain after the first delete?
After the second?
What does the list look like after both calls?

Time Complexity of List Operations

Operation	Worst Case Time	Best Case Time
`CreateList`		
`RetrieveElement`		
`InsertHead`		
`InsertTail`		
`InsertAfter`		
`DeleteHead`		
`DeleteTail`		
`DeleteElement`		
`ListIsEmpty`		
`ListCount`		
`DestroyList`		

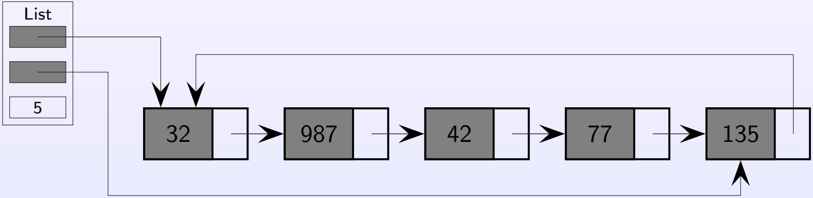
Time Complexity of List Operations

Operation	Worst Case Time	Best Case Time
`CreateList`	O(1)	O(1)
`RetrieveElement`	O(n)	O(1)
`InsertHead`	O(1)	O(1)
`InsertTail`	O(1)	O(1)
`InsertAfter`	O(n)	O(1)
`DeleteHead`	O(1)	O(1)
`DeleteTail`	O(n)	O(n)
`DeleteElement`	O(n)	O(1)
`ListIsEmpty`	O(1)	O(1)
`ListCount`	O(1)	O(1)
`DestroyList`	O(n)	O(n)

n is the number of elements in the list.

Circular Linked Lists

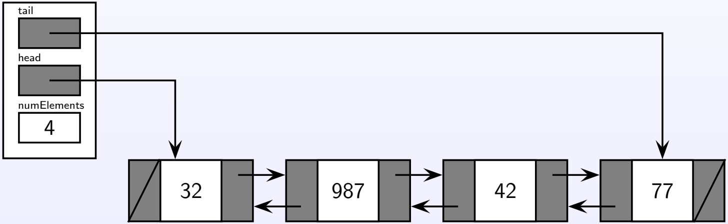
- A variant of a linked list is the *circular linked list*.
- In such a list the last node points back to the first node instead of referencing NULL.



Circular Linked Lists

- The algorithms for the operations are more or less the same except:
  - Slightly different (and simplified) updating of pointers for insertion and deletion.
  - Operations that have to search the list need to check for the end of the list differently.

Doubly Linked Lists



- In a doubly linked list, each node has a reference to its predecessor as well as its successor.
- Aside: How might we make a circular doubly linked list?

Data Structures

- Add a `prev` reference to nodes:

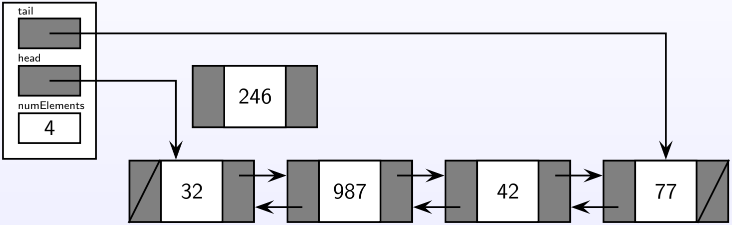
```
Node
    Element data        // An 'Element' can be anything we want.
    refToNode next      // A reference to the next node in the list
    refToNode prev      // A reference to the previous node
end Node
```

- The List structure stays the same:

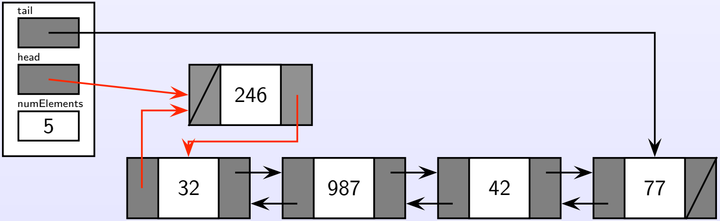
```
List
    refToNode head          // Reference to the first node in the list.
    reftoNode tail          // Reference to the last node in the list.
    Integer numElements     // Number of elements in the list.
end List
```

Insert at Beginning

Before:



After:



Algorithm *InsertHead*(*rList*, *el*)
Insert data at beginning of list.

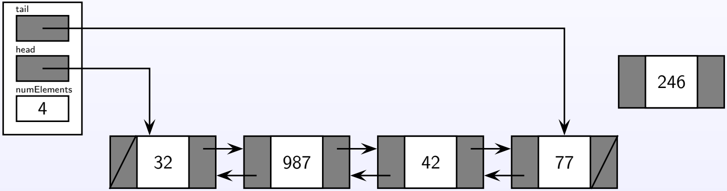
Pre: *rList* is reference to the list into which to insert
el is the element to insert
Post: Element has been inserted as the first node.
Returns: **true** if successful, **false** otherwise

refToNode rNew ← **allocate new** *Node*
if (*rNew* = **NULL**) **return false**
rNew ⤃ *next* ← **NULL**
rNew ⤃ *prev* ← **NULL**
rNew ⤃ *data* ← *el*

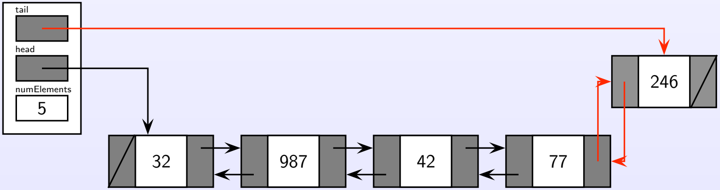
if (*ListIsEmpty*(*rlist*))
rList ⤃ *head* ← *rNew*
rList ⤃ *tail* ← *rNew*
else
rList ⤃ *head* ⤃ *prev* ← *rNew*
rNew ⤃ *next* ← *rList* ⤃ *head*
rList ⤃ *head* ← *rNew*
end if
rList ⤃ *numElements* ← *rList* ⤃ *numElements* + 1
return true

Insert at End

Before:



After:



Algorithm *InsertTail*(*rList*, *el*)
Insert data at end of list.

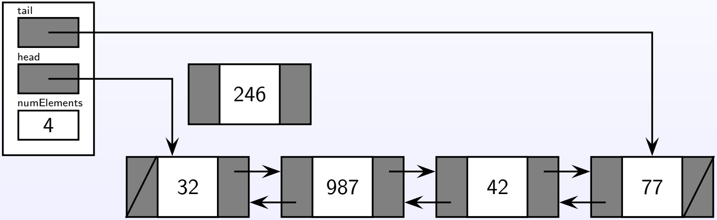
Pre: *rList* is reference to the list into which to insert
el is the element to insert
Post: Element has been inserted as the last node.
Returns: **true** if successful, **false** otherwise (e.g. if out of memory)

refToNode rNew $\leftarrow$ **allocate new** *Node*
if (*rNew* = **NULL**) **return false**
rNew $\Rrightarrow$ *next* $\leftarrow$ **NULL**
rNew $\Rrightarrow$ *prev* $\leftarrow$ **NULL**
rNew $\Rrightarrow$ *data* $\leftarrow$ *el*

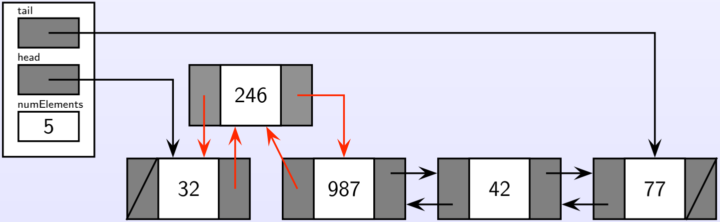
if (*ListIsEmpty*(*rlist*))
rList $\Rrightarrow$ *head* $\leftarrow$ *rNew*
rList $\Rrightarrow$ *tail* $\leftarrow$ *rNew*
else
rList $\Rrightarrow$ *tail* $\Rrightarrow$ *next* $\leftarrow$ *rNew*
rNew $\Rrightarrow$ *prev* $\leftarrow$ *rList* $\Rrightarrow$ *tail*
rList $\Rrightarrow$ *tail* $\leftarrow$ *rNew*
end if
rList $\Rrightarrow$ *numElements* $\leftarrow$ *rList* $\Rrightarrow$ *numElements* + 1
return true

Insert After Element

Before:



After:



Arbitrary Insertion

Algorithm *InsertAfter*(*rList*, *target*, *el*)
Insert data after a specified element.

Pre: *rList* is reference to the list into which to insert
target is the key of the element after which
new element should be inserted
el is the element to insert
Post: Element has been inserted after element with given key
Returns: **true** if successful, **false** otherwise

refToNode rNew ← **allocate new** *Node*
if (*rnew* = **NULL**) **return false**
rNew ⤃ *next* ← **NULL**
rNew ⤃ *prev* ← **NULL**
rNew ⤃ *data* ← *el*

// Search for 'target'
rLoc ← *rList* ⤃ *head*
while(*rLoc* != **NULL**)
  if (*target* == key of *rLoc* ⤃ *data*)
    break
  end if
  rLoc ← *rLoc* ⤃ *next*
end while

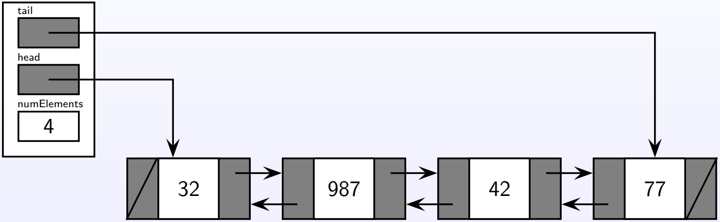
// continued next slide...

Arbitrary Insertion

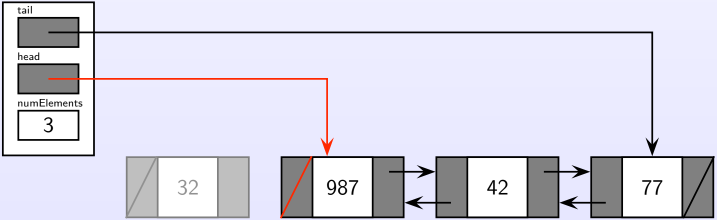
```
// ... continued from previous slide

if ( rLoc != NULL )
    rNew ⇻ prev ← rLoc
    rNew ⇻ next ← rLoc ⇻ next
    // in case we are inserting after the last element...
    if ( rLoc == rList ⇻ tail )
        rList ⇻ tail ← rNew
    else
        rLoc ⇻ next ⇻ prev ← rNew
    end if
    rLoc ⇻ next ← rNew
    rList ⇻ numElements ← rList ⇻ numElements + 1
    return true
else
    return false
end if
```

Delete Head

Before:



After:



Delete From Head

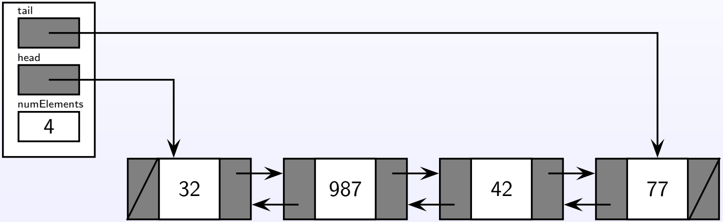
Algorithm DeleteHead(rList, el)
Delete data at beginning of list.

Pre: *rList* is reference to the list from which to delete
el is reference to variable to store a copy of the deleted element
Post: First element has been deleted and node memory freed.
el points to a copy of the deleted element.
Returns: **true** if successful, **false** otherwise

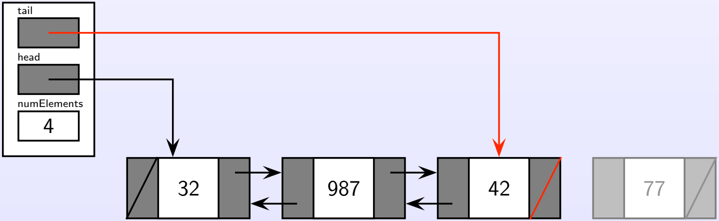
```
if ( ListIsEmpty() )
    return false
end if

*el ← rList ⤃ head ⤃ data

refToNode temp ← rList ⤃ head
rList ⤃ head ← rList ⤃ head ⤃ next
if ( rList ⤃ head != NULL )
    rList ⤃ head ⤃ prev ← NULL
else
    rList ⤃ tail ← NULL
end if

deallocate temp
rList ⤃ numElements ← rList ⤃ numElements - 1
return true
```

Delete Tail

Before:



After:



Delete From Tail

Algorithm DeleteTail(rList, el)
Delete data from the end of list.

Pre: *rList* is reference to the list from which to delete
el is reference to variable to store deleted element
Post: Last element has been deleted and node memory freed.
a copy of the deleted element is stored in **el*
Returns: **true** if successful, **false** otherwise

if (*ListIsEmpty*())
  return false
end if

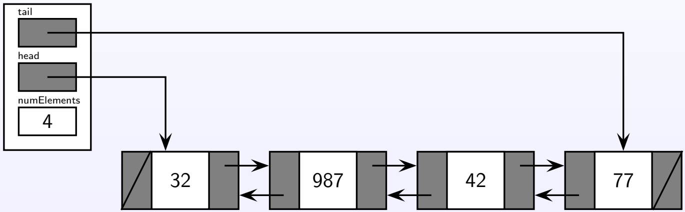
$*el \leftarrow rList \twoheadrightarrow tail \twoheadrightarrow data$

$refToNode\ temp \leftarrow rList \twoheadrightarrow tail$
$rList \twoheadrightarrow tail \leftarrow rList \twoheadrightarrow tail \twoheadrightarrow prev$
if ($rList \twoheadrightarrow tail$!= **NULL**)
  $rList \twoheadrightarrow tail \twoheadrightarrow next \leftarrow$ **NULL**
else
  $rList \twoheadrightarrow head =$ **NULL**
end if

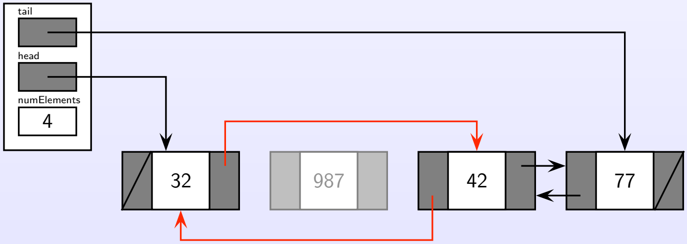
deallocate temp
$rList \twoheadrightarrow numElements \leftarrow rList \twoheadrightarrow numElements - 1$
return true

Delete Arbitrary

Before:



After:



Delete Arbitrary

Algorithm DeleteElement(rList, target, el)
Delete data from arbitrary spot in list.

Pre: *rList* is reference to the list from which to delete
target is the key of the element to delete
el is reference to variable to store deleted element
Post: Last element has been deleted and node memory freed.
A copy of the deleted element is stored in **el*
Returns: **true** if successful, **false** otherwise

```
// Search for 'target'
refToNode rLoc ← rList ⤔ head
while( rLoc != NULL )
    if ( target = key of rLoc ⤔ data )
        break
    end if
    rLoc ← rLoc ⤔ next
end while

// continued next slide ...
```

Delete Arbitrary

```
// ... continued from previous slide

if ( rLoc != NULL )
    *el ← rLoc ⤃ data
    if (rLoc ⤃ prev != NULL)
        rLoc ⤃ prev ⤃ next ← rLoc ⤃ next
    else rList ⤃ head ← rLoc ⤃ next
    if (rLoc ⤃ next != NULL)
        rLoc ⤃ next ⤃ prev ← rLoc ⤃ prev
    else rList ⤃ tail ← rLoc ⤃ next
    end if
    deallocate rLoc
    rList ⤃ numElements ← rList ⤃ numElements - 1
    return true
else
    return false
end if
```

Time Complexity of Doubly Linked List Operations

Operation	Worst Case Time	Best Case Time
`CreateList`	O(1)	O(1)
`RetrieveElement`	O(n)	O(1)
`InsertHead`		
`InsertTail`		
`InsertAfter`		
`DeleteHead`		
`DeleteTail`		
`DeleteElement`		
`ListIsEmpty`	O(1)	O(1)
`ListCount`	O(1)	O(1)
`DestroyList`	O(n)	O(n)

n is the number of elements in the list

Time Complexity of List Operations

Operation	Worst Case Time	Best Case Time
`CreateList`	O(1)	O(1)
`RetrieveElement`	O(n)	O(1)
`InsertHead`	O(1)	O(1)
`InsertTail`	O(1)	O(1)
`InsertAfter`	O(n)	O(1)
`DeleteHead`	O(1)	O(1)
`DeleteTail`	O(1)	O(1)
`DeleteElement`	O(n)	O(1)
`ListIsEmpty`	O(1)	O(1)
`ListCount`	O(1)	O(1)
`DestroyList`	O(n)	O(n)

n is the number of elements in the list