Why did the computer show up late to work?

It had a hard drive !

Source: https://www.rd.com/jokes/computer/

CMPT 225

Lecture 5 – **Linked list**-based implementation of **List** ADT class

Last Lecture

- ✓ Continued with Step 3 – Implementation of **List** ADT class
  - ✓ Array-based implementation of **List** ADT
  - ✓ Differentiated between ***stack-allocated*** (automatically allocated) and ***heap-allocated*** (dynamically allocated) **arrays**
- ✓ Introduced 2nd data structure (CDT): **linked list**
- ✓ Built **linked list**: pointers and node objects
- ✓ **Linked list** operations
  - ✓ insert @ front (prepend)
    - ✓ Generalization Principle

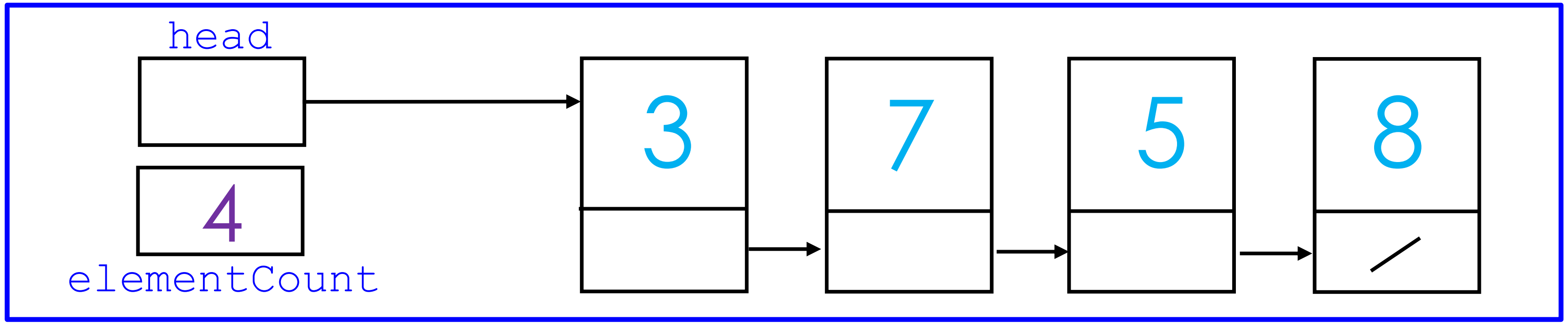
Today's Menu

- Finish looking at **Linked list** operations
- And various configurations of **linked lists**
  - Know when to use them (know their ***forte***)
- Step 3 – Implementation - **Linked list**-based implementation of **List** ADT class
  - Introduce a **Node** class
- Compare the two implementations of our **List** ADT class:
  - **Array**-based implementation
  - **Linked list**-based implementation

Time efficiency of this operation:
(expressed in Big O notation)

REVIEW

Traverse a linked list



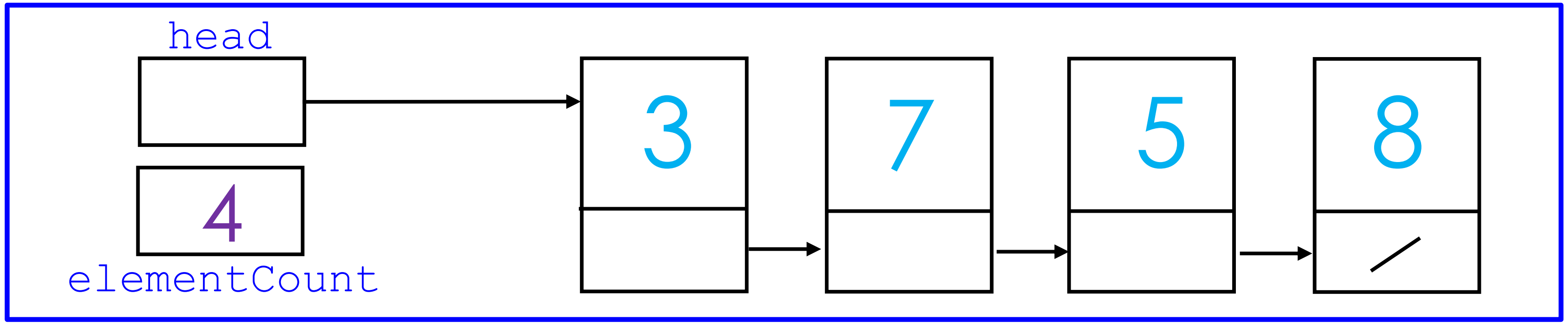
Local variable: current

pseudocode

```
// Anchor head of linked list
1. if ( head != nullptr )
     2. Node* current = head;
     3. while (current->next != nullptr)
          4. current = current->next;
```

REVIEW

Traverse - Do we need the anchor?



pseudocode

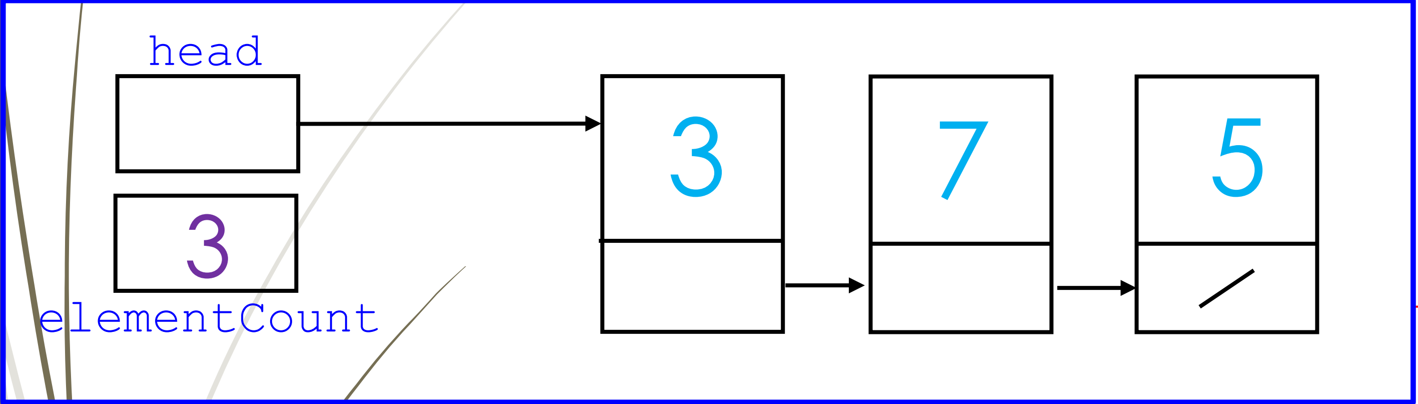
```
// Traverse linked list
1. if ( head != nullptr )
   2. while (head->next != nullptr)
      3. head = head->next;
```

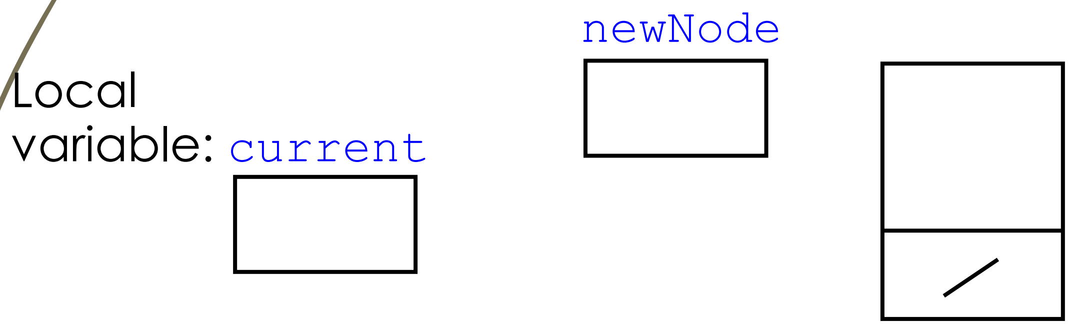
REVIEW

Time efficiency of this operation:
(expressed in Big O notation)

Insert an element into a linked list

- insert @ end (append)





pseudocode

```
... append(int newElement)
1. Node *newNode = new Node(newElement);
2. if (newNode != nullptr)
   3. if (head == nullptr)
      4. head = newNode;
     else
      // Move to the end of the list
      5. Node* current = head;    // Anchor
      6. while (current->next != nullptr)
          7. current = current->next;
      8. current->next = newNode;
   9. elementCount++;
```

A word about inserting an element into a linked list

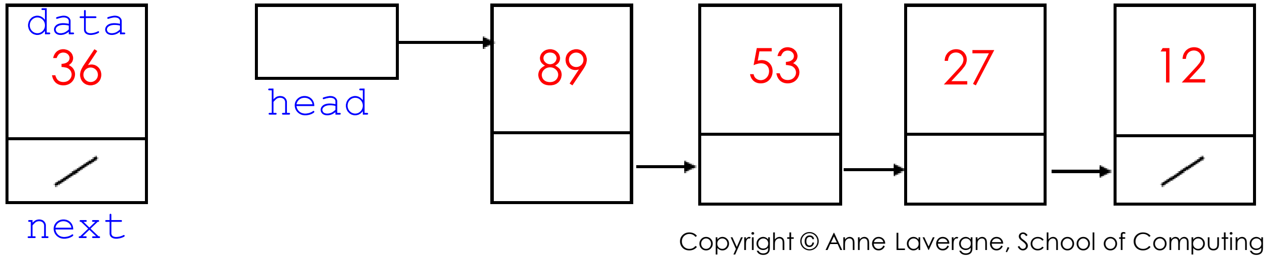
- @ specific location
  - When ***linked list*** is used as a data structure (CDT) for a position-oriented data collection ADT class like a List, we can indicate at which position we would like to insert an element
    - *position* is a parameter of the **`insert`** method

OR

  - When ***linked list*** is used as a data structure (CDT) for a value-oriented data collection ADT class like a List (which is kept sorted), in order to keep it sorted, ***we insert the element in sort order into the List*** using a ***search key*** (i.e., an element's attribute) — Algorithm 1
  - Alternatively, we could first prepend the element into the List (this is time efficient, i.e., O(1)), then we sort the List (sorting algorithms can be $O(n^2)$ or $O(n \log(n))$. As you can see, this 2nd way of "keeping the List sorted when we insert" is **not time efficient!** — Algorithm 2

Class invariant for this List class: the List is always sorted by …
e.g. ascending or descending alphabetical/numerical sort order of search key … depending on the problem we are solving.

List kept in descending sort order of `data`:

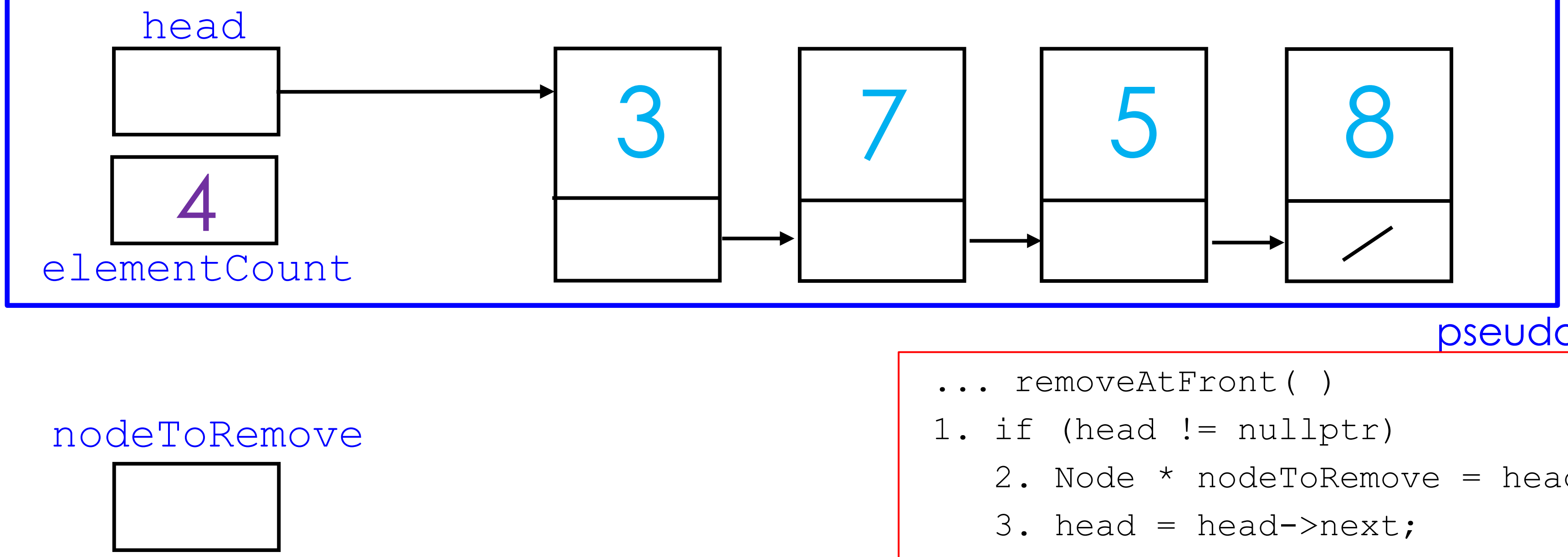


Time efficiency of this operation:
(expressed in Big O notation)

REVIEW

Remove an element from a linked list

- remove @ front



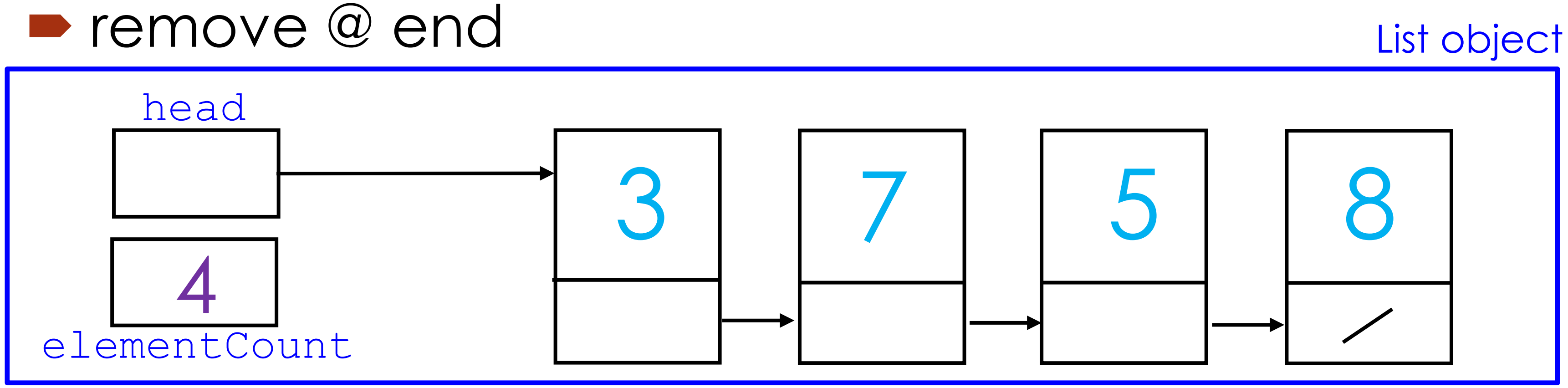
pseudocode

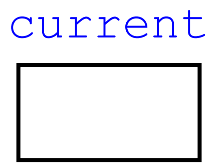
```
... removeAtFront( )
1. if (head != nullptr)
   2. Node * nodeToRemove = head;
   3. head = head->next;
   // Return node to the system
   4. nodeToRemove->next = nullptr;
   5. delete nodeToRemove;
   6. nodeToRemove = nullptr;
   7. elementCount--;
```

Time efficiency of this operation:
(expressed in Big O notation)

Remove an element from a linked list

- remove @ end





pseudocode

```
... removeAtEnd( )
1. if (head != nullptr)
     // Move to the end of the list
     2. Node * current = head;   // Anchor
     3. while (current->next != nullptr)
          4. current = current->next;
     // Then what???
```

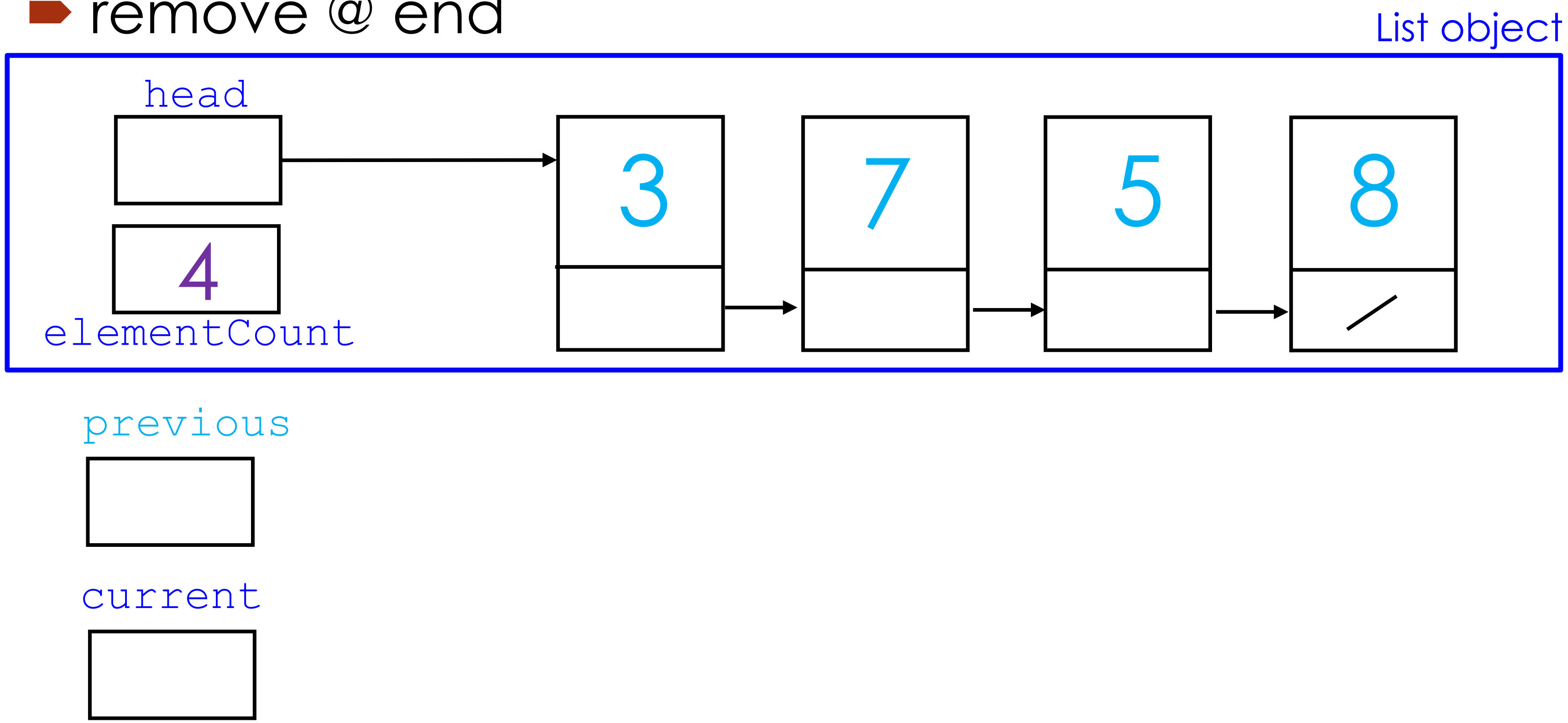
Issue with Traverse and possible solutions

1. Using a local variable `previous`
2. Adding another ***link*** into the **List**
3. Using a ***Look Ahead*** mechanism

   => `current->next->next`

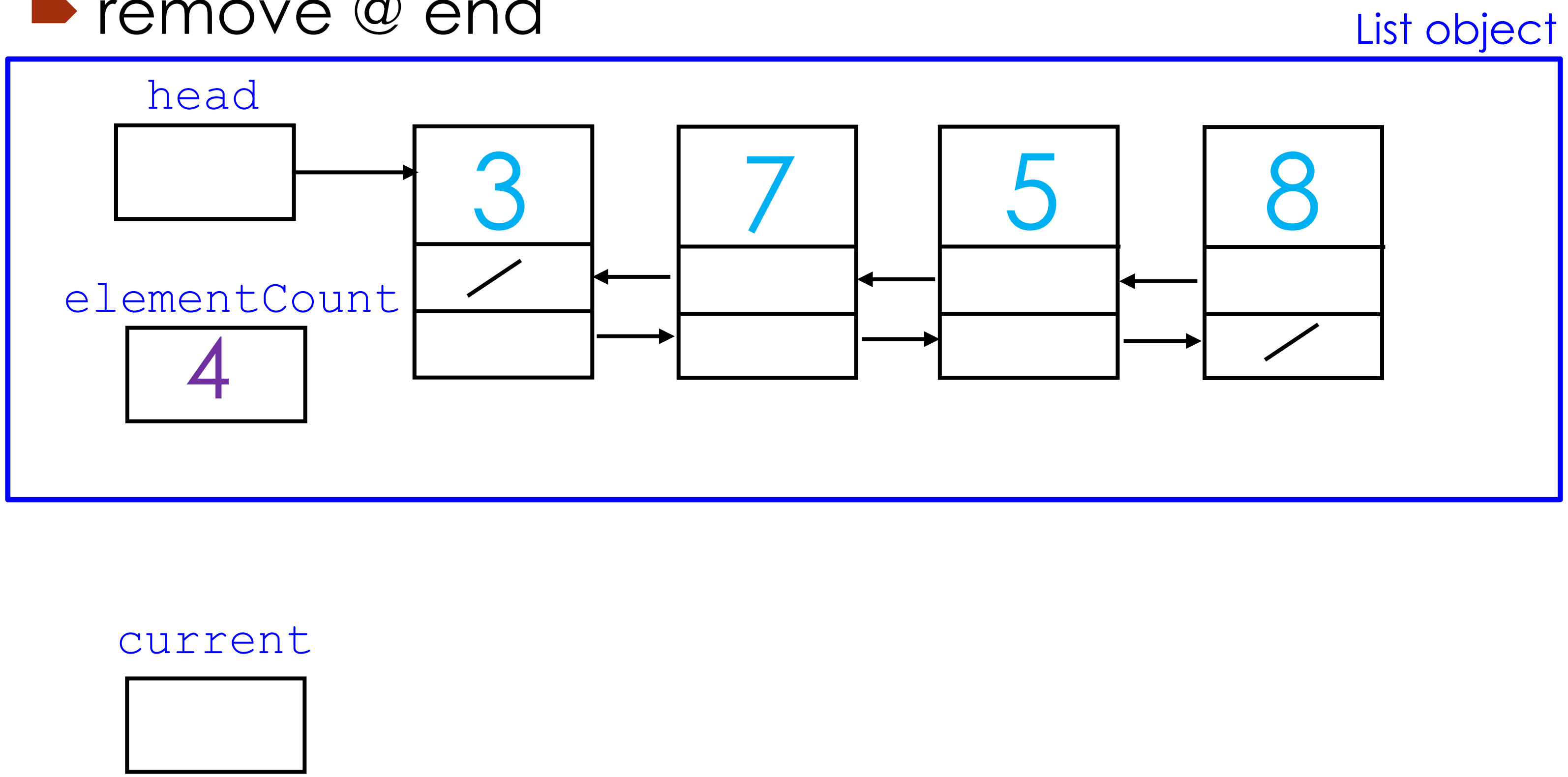
1. Removal – with `previous`

- remove @ end



2. Removal – adding another ***link*** into the List

- remove @ end



Removing an element from a **linked list**

- @ specific location
  - When ***linked list*** is used as a data structure (CDT) for a position-oriented data collection ADT class like a List, we can indicate the position of the element we wish to remove
    - *position* is a parameter of the **`remove`** method

OR

  - When ***linked list*** is used as a data structure (CDT) for a value-oriented data collection ADT class like a List (which is kept sorted), we can indicate which element to remove by supplying a ***search key*** (i.e., an element's attribute)
    - *search key* is a parameter of the **`remove`** method

Class invariant for this List class: the List is always kept in sorted order

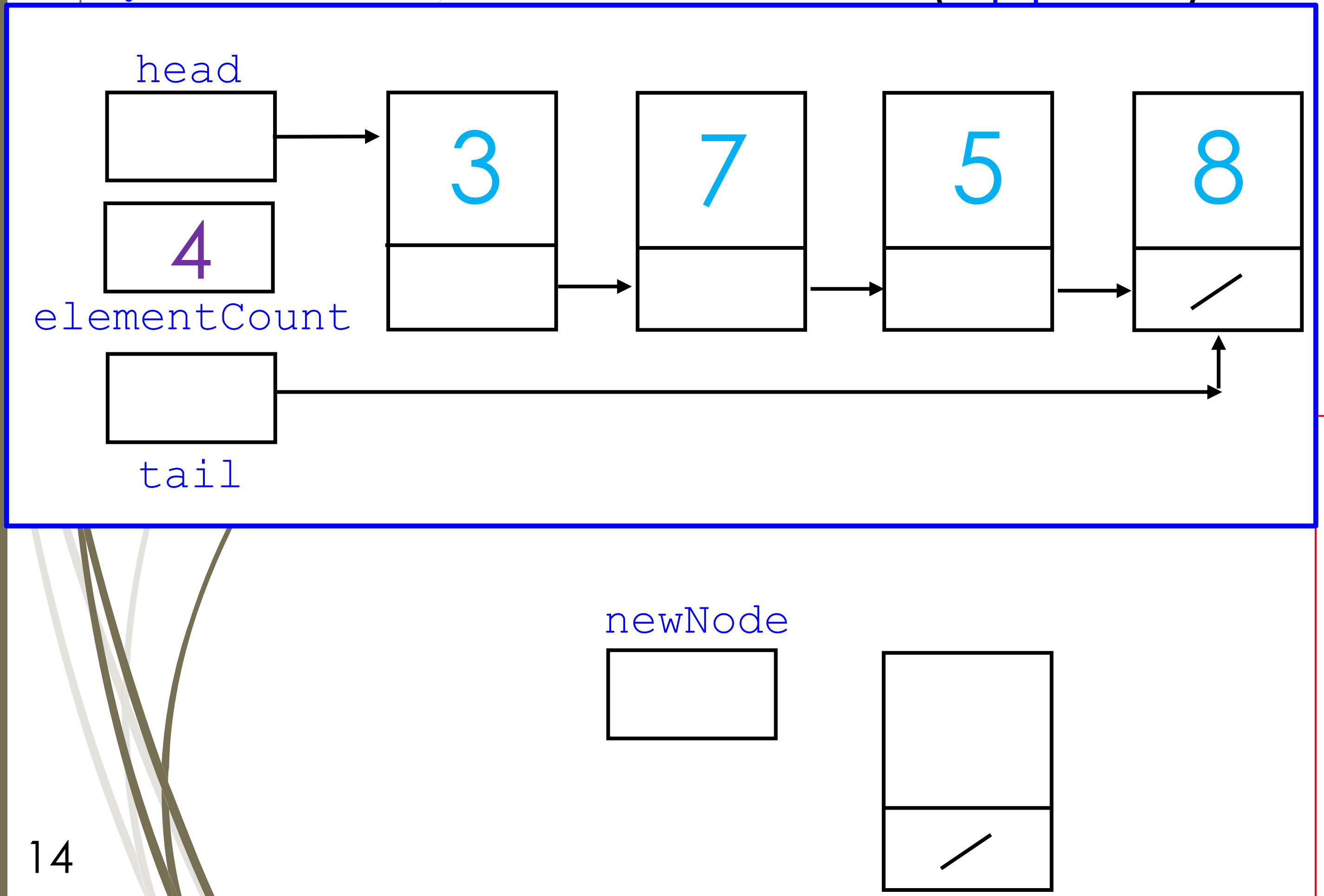
Improving the insertion of an element @ end

O(n) -> O(1)

Doubly Headed Singly Linked list
=> DHSL list

- insert @ end (append)

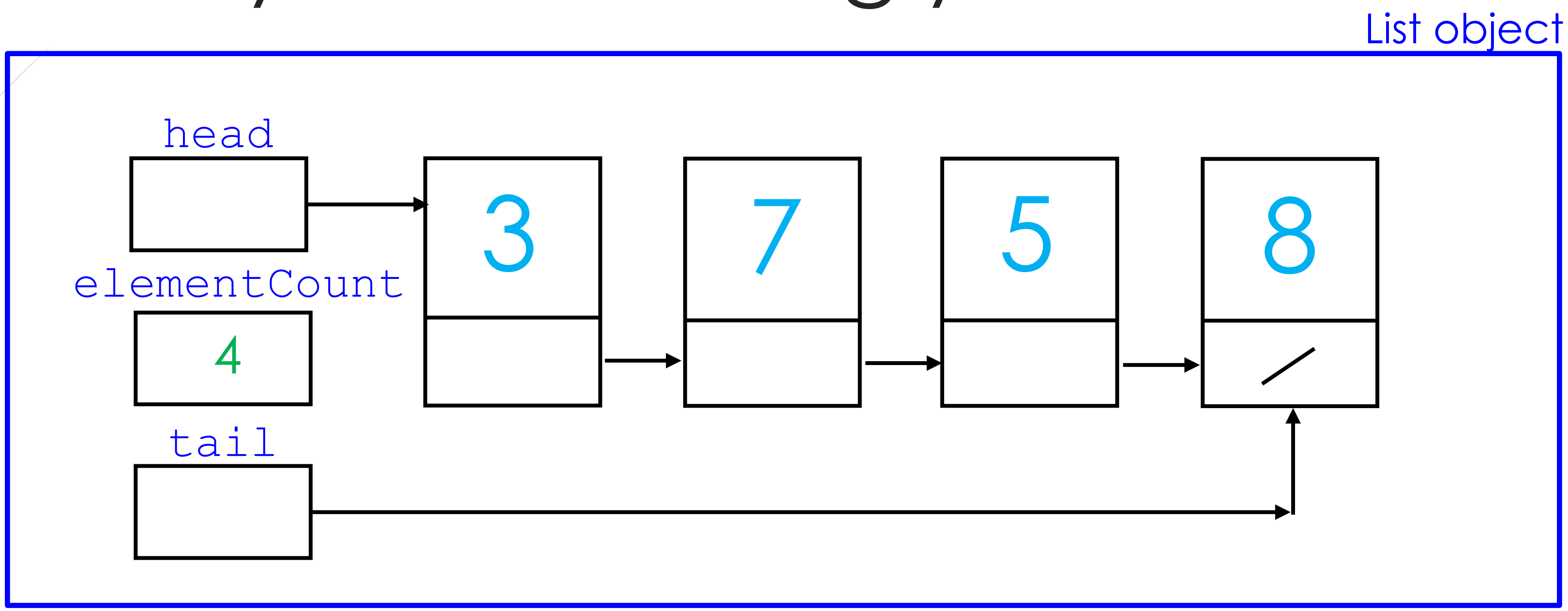
List object



pseudocode

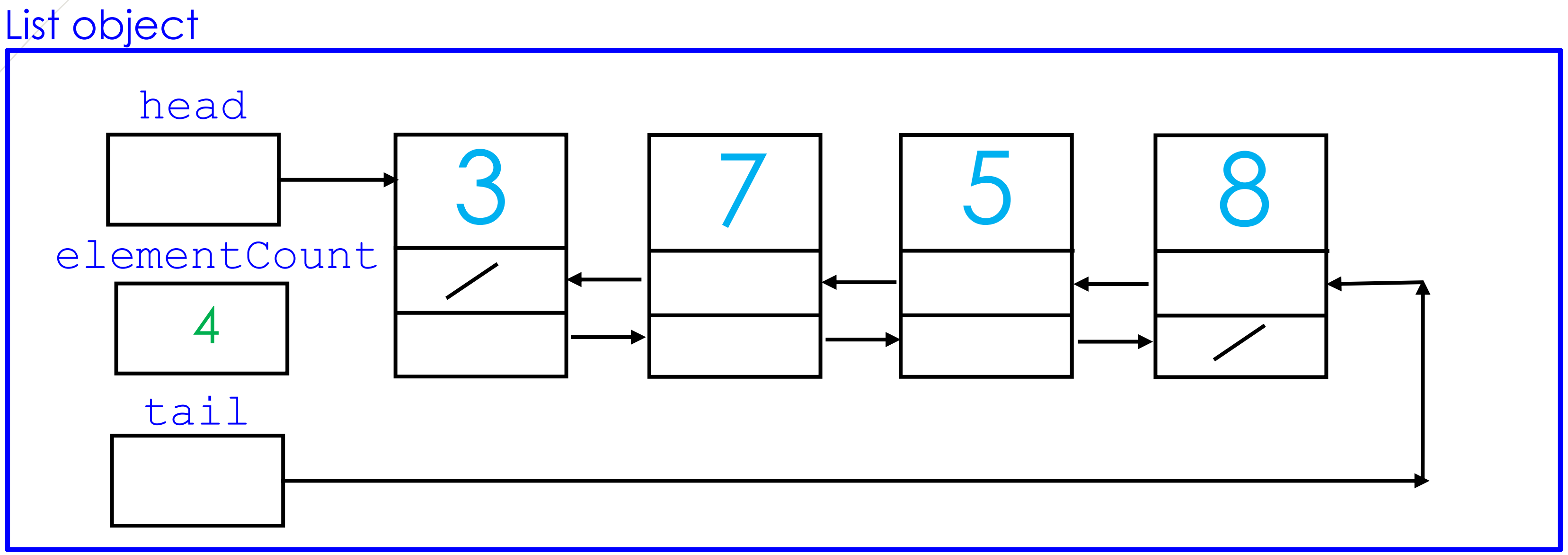
```
... append(int newElement)
1. Node *newNode = new Node(newElement);
2. if (newNode != nullptr)
     3. if (tail == nullptr) // or head == nullptr
        4. head = newNode;      when List empty
        5. tail = newNode;
     else
        6. tail->next = newNode;
        7. tail = newNode;
   10. elementCount++;
```

Doubly Headed Singly Linked list – DHSL list



- Advantage: Allows us to insert @ end (append) in O(1). ☺
- Disadvantage: More code to develop, maintain and test. ☹
  => Need to keep **`tail`** properly updated.

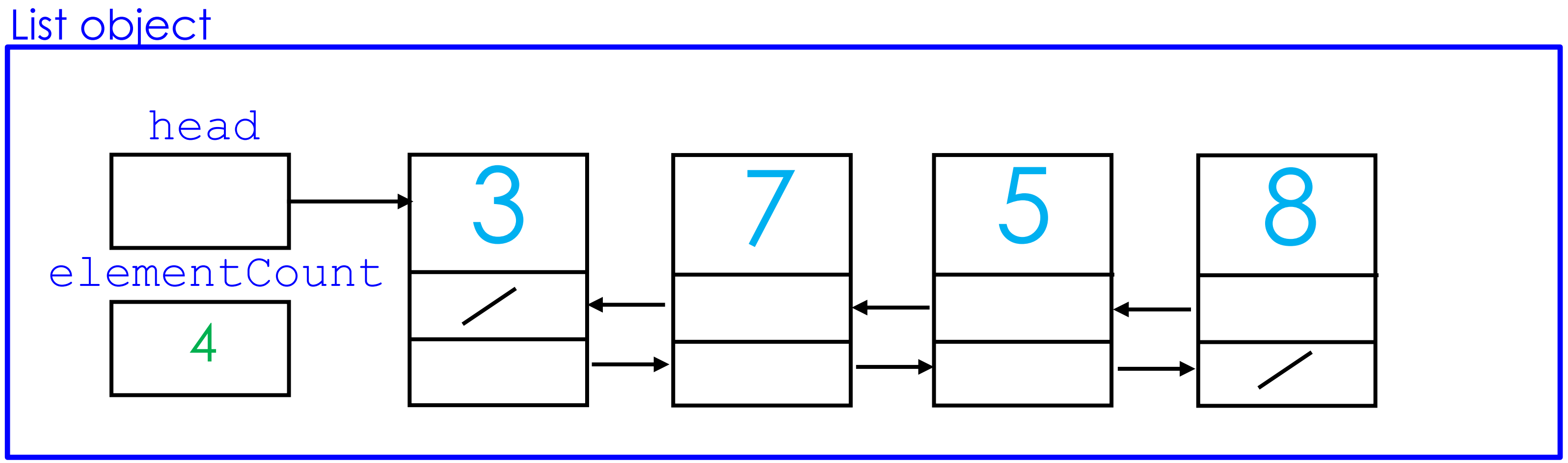
Doubly Headed Doubly Linked list – DHDL list



- Advantage: Allows us to remove the element at the end in O(1). ☺
- Disadvantage: More code to develop, maintain and test. ☹
  => Need to keep **`tail`** and the second link **`back`** properly updated.

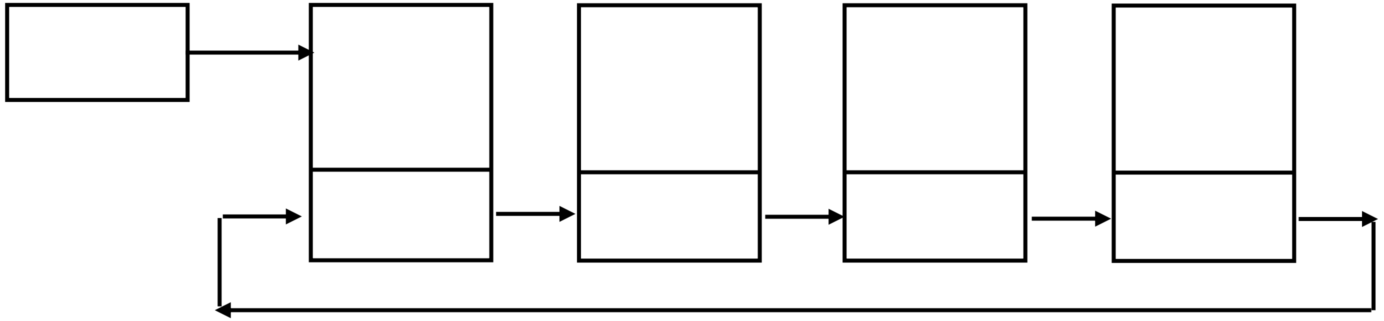
Various configurations of **linked lists**

- **Linked lists** are very flexible
- Singly Headed Doubly Linked list – SHDL list:

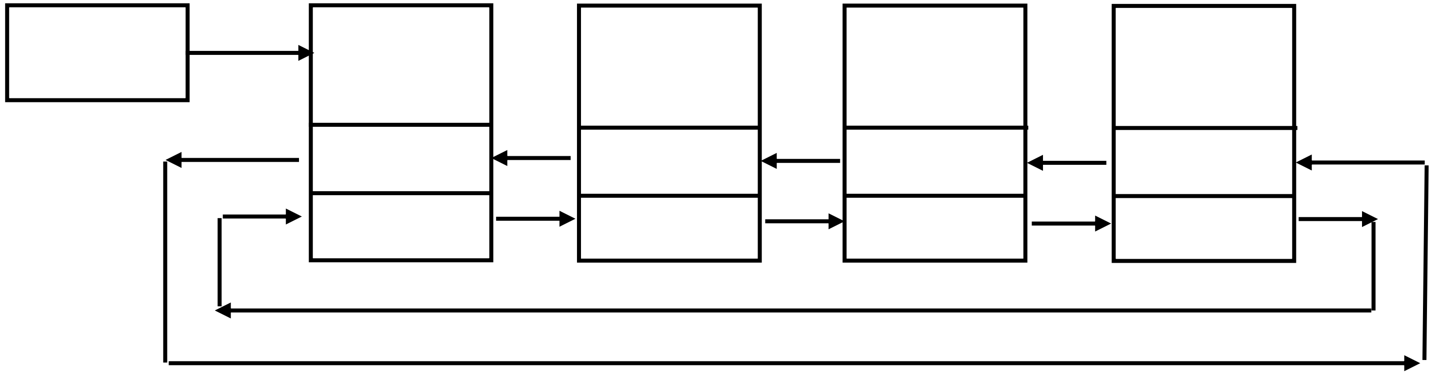


Various configurations of **linked lists** (cont'd)

- Singly Headed Singly Linked circular list:



- Singly Headed Doubly Linked circular list:

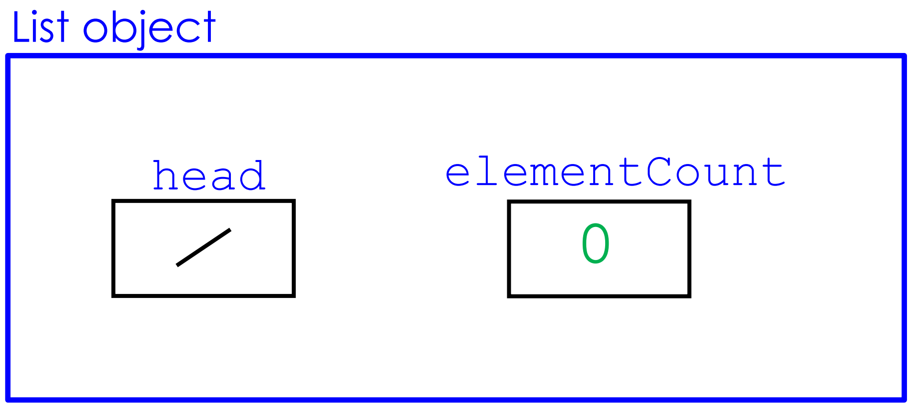


Step 3 - Implementation - Node Class

```
/*
 * Node.h
 *
 * Class Definition: Node of a singly linked (SL) list
 *                   in which the data is of "int" data type.
 *                   Designed and implemented as a non-ADT.
 *
 * Created on:
 * Author:
 */


class Node {

public:

    // Public data members - Why are the data members public?
    int data = 0;               // The data in the node
    Node * next = nullptr;      // Pointer to next node

    // Constructors
    Node();
    Node(int theData);
    Node(int theData, Node * theNextNode);

}; // end Node
```

```
/*
 * Node.cpp
 *
 * Class Definition: Node of a singly linked (SL) list
 *                   in which the data is of "int" data type.
 *                   Designed and implemented as a non-ADT.
 *
 * Created on:
 * Author:
 */

#include <cstdio>  // Needed for NULL
#include "Node.h"

Node::Node() {}

Node::Node(int theData) {
    data = theData;
}

Node::Node(int theData, Node * theNextNode) {
    data = theData;
    next = theNextNode;
}

// end Node.cpp
```

Construct a **List** object - `constructor` (made of an empty linked list)



Be Careful

- Do not confuse ***data members*** of the **List** ADT class (components of a **linked list**)
  - Such as `head` and `tail`

  and ***data members*** of the Node class
  - Such as `next` and `back`

  with ***local variables*** of the methods (of **List** ADT class) manipulating the **linked list**
  - Such as `current` and `previous`

List ADT class invariant?
- Element duplication allowed?
- …?

Comparing various implementations of the position-oriented List ADT class using Big O notation

- Time efficiency of their operations (worst case scenario):

Operations	**array-based** (array allocated on the heap)	**link-based**
getElementCount		
insert		
remove		
removeAll/clear		
retrieve/get		

List ADT class invariant?
- Sorted
- Element duplication allowed?
- …?

Comparing various implementations of the value-oriented List ADT class using Big O notation

- Time efficiency of their operations (worst case scenario):

Operations	array-based (array allocated on the heap)	link-based
getElementCount		
insert		
remove		
removeAll/clear		
retrieve/get (search)		

√ Learning Check

- ✓ We can now …
  - ✓ Perform operations on a **linked list**
  - ✓ Create **linked list** of various configurations (SHSL list, DHSL list, DHDL list, …)
    - ✓ Know when to use them (know their ***forte***)
  - ✓ Step 3 – Implement a data collection **List** ADT class:
    - ✓ Using an **array** (heap-allocated)
    - ✓ Using a **linked list**
      - ✓ Create a **Node** class
  - ✓ Compare the efficiency of the methods for both implementations of our **List** ADT class

Next Lecture

- Step 4 – Compilation and Testing
- Documentation
- Introduce our **next linear data collection**