CMPT 225

Lecture 4 – Data collection **List** ADT

Learning Outcomes

- At the end of this lecture, a student will be able to:
  - manage memory in C++, allocate and free arrays and individual elements in heap
  - differentiate between memory declared globally, on the stack, and on the heap
  - define one of the concrete data types, namely ***linked list***, and demonstrate, simulate, and trace its operations
  - manipulate pointers
  - write C++ code

Last Lecture

- ✓ Example: **Temperature ADT class** versus **non-ADT class** and class **test driver**
- ✓ Terminology
  - ✓ **Data Collection** (ADT) versus **Data Structure** (CDT)
  - ✓ Categories of **data organizations**
- ✓ Introduced our first **data collection: List**

As part of solving the **FriendsBook** problem

- ✓ Step 2 – Design **List** as an ADT class
  1. Design its ***public*** section (**public interface**) -> its operations
  2. Design its ***private*** section -> its data and the implementation of its operations

Hidden behind the wall

At least, a **CDT** + a variable to keep track of number of elements in **List**: **`elementCount`**

- ✓ Step 3 – Implementation of **List** ADT class
  1. Array-based implementation of **List** ADT

From last lecture: List Public Interface

1. Insert element into List
   - Position-oriented List:
     - Prepend (insert at the front – position `1`)
     - Append (insert at the back – position `elementCount + 1`)
     - Insert at a particular position (between the above two positions)
   - Value-oriented List:
     - Insert in **sort** order using the element's *search key*
2. Remove element from List
   - Position-oriented List: Remove element from either position `1`, position `elementCount`, or any position in between
   - Value-oriented List: Remove element that matches the *search key* (require Searching)
3. Get/retrieve a particular element from List …
   - Position-oriented List: … located at position X where range of X is [`1` .. `elementCount`]
     - Return a pointer or reference to *target* element
   - Value-oriented List: … that matches the given *search key* (require searching)
     - Return a pointer or reference to *target* element or its index

Today's menu

- Continue with Step 3 – Implementation of **List** ADT class
  1. Array-based implementation of **List** ADT
  - Design and implement our methods
  - Memory management:
    - Differentiate between ***stack-allocated*** (automatically allocated) and ***heap-allocated*** (dynamically allocated) **arrays**
- Introduce 2^{nd} data structure (CDT): **linked list**
- Build **linked list**: pointers and node objects
- Start looking at some **Linked list** operations

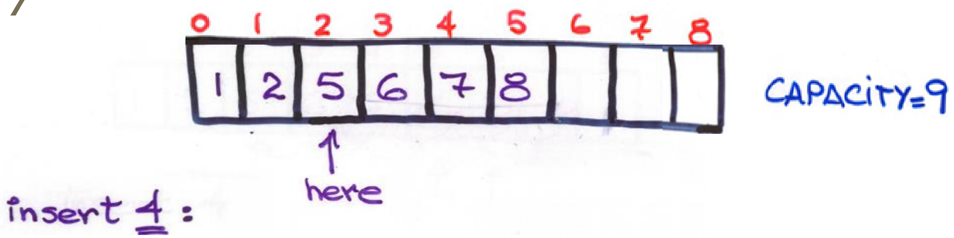
Step 3 – Array-based implementation of **List** ADT

- Considering a value-based **List**:
  - Insert element into **List** -> O(n)
    - Find where the element is to be inserted in the array (make sure element not already there): O(n) or O(log n)
    - Shift elements right from this location onwards to make room: O(n)
    - Insert element: O(1) and `elementCount++`: O(1)

Class invariant:
- **List** kept in sort order of ***search key*** and
- No duplications are allowed

Scenario 1

List sorted + no duplications allowed

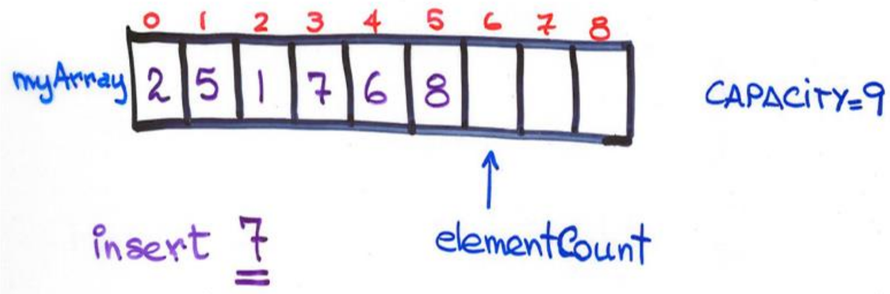


1. full? If so, expand first
2. find where `newElement` is to go
3. shift right to make space for `newElement`
4. `myArray[here] = newElement`
5. `elementCount++`

What happen when duplications are allowed?

What if … Scenario 2

List unsorted + duplications allowed



1. full? If so, expand first
2. `myArray[elemenCount] = newElement`
3. `elementCount++`

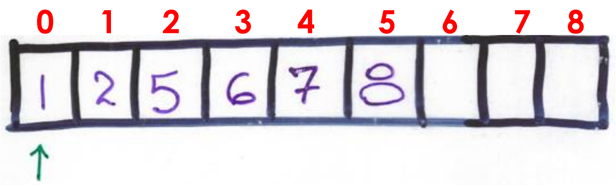
What happen when duplications are not allowed?

Step 3 – Array-based implementation of **List** ADT

- Considering a value-based **List**:
  - Remove element from **List** –> O(n)
    - Find element's location using its *search key*: O(n) or O(log n)
    - Shift elements left to overwrite it: O(n) and `elementCount--`: O(1)
      - No need to "erase" an element from an array cell, simply overwrite it

Scenario 1

List sorted + duplications not allowed

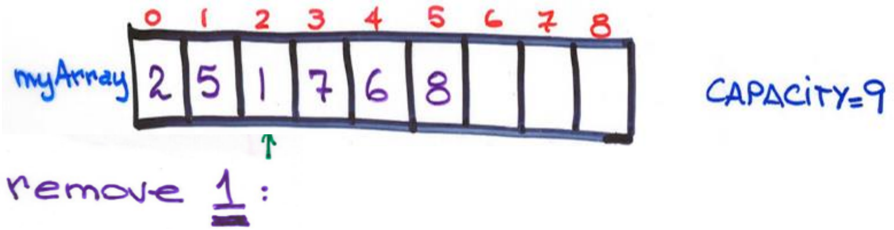


1. Empty?
2.
3.
4. `elementCount--`

What happen when duplications are allowed?

What if … Scenario 2

List unsorted + duplications not allowed



1. Empty?
2.
3.
4. `elementCount--`

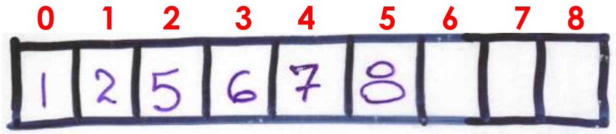
What happen when duplications are allowed?

Step 3 – Array-based implementation of List ADT

- Considering a value-based **List**:
  - Get/retrieve a particular element from **List** using its *search key* -> O(n) or O(log n)

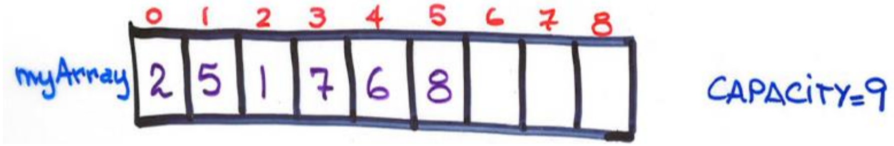
Scenario 1

List sorted + duplications not allowed



What if … Scenario 2

List unsorted + duplications not allowed

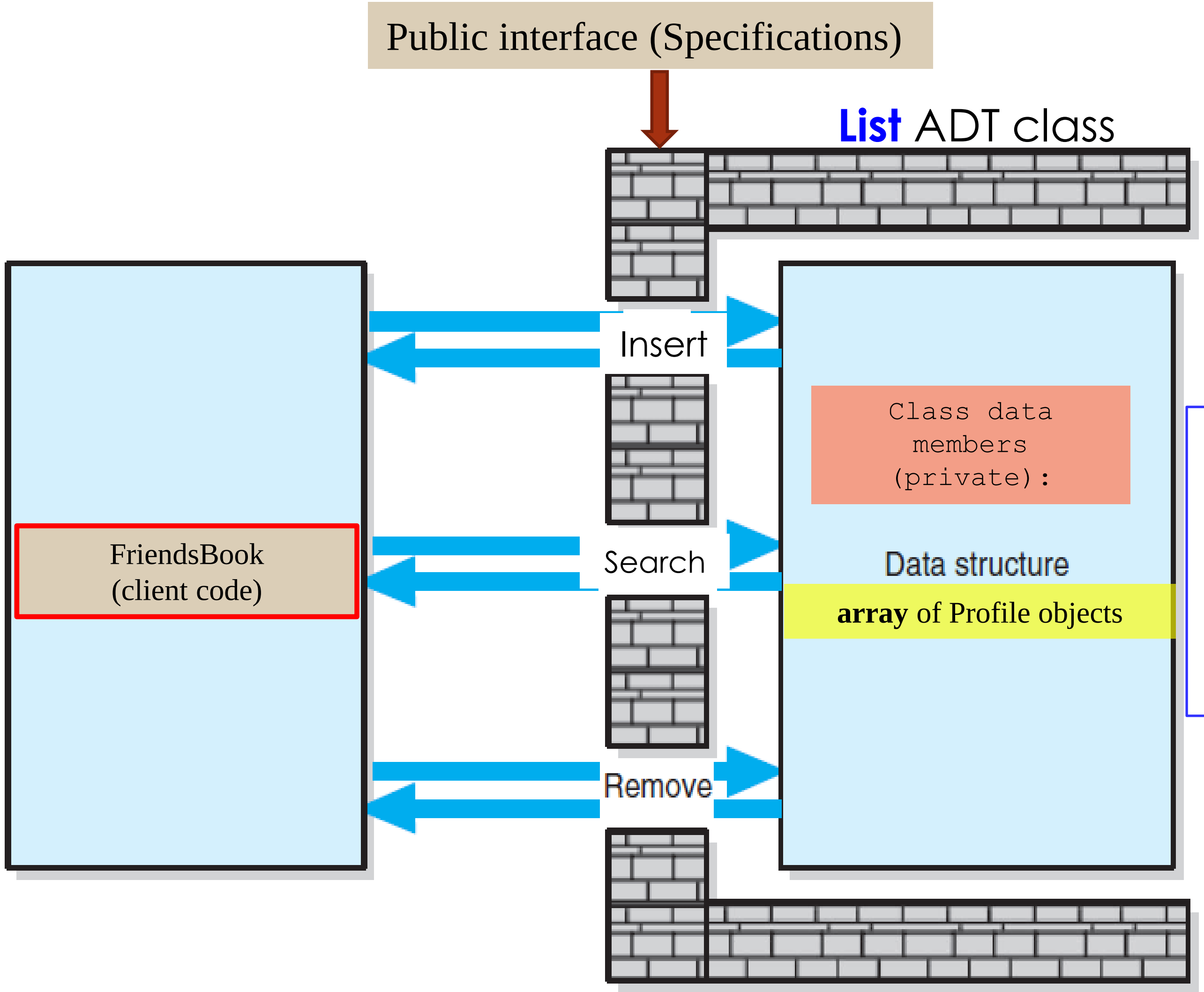


  - "Expand" (resize) the **List** (i.e., expand the underlying array): O(?)
    (see Expandable Array under **Lecture 3** for more information)
- **Note:** Easier to manage when there are no gaps in the array!

Step 3 – Array-based implementation of **List** ADT

- When designing and implementing our methods,
  - always choose the most time efficient algorithm keeping in mind the characteristics of the **List** -> sorted or not, duplications allowed or not

Step 3 – Array-based implementation of **List** ADT



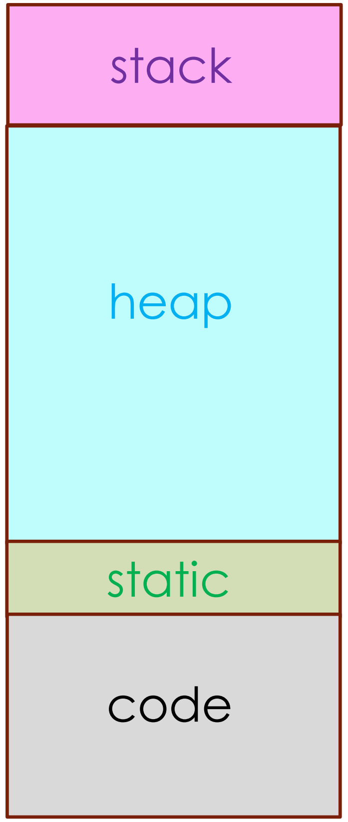
The ***stack*** segment of memory may not be the best place to hold (potentially) large amount of data (Profile objects) … Hum!

Activity: ***Stack*** versus ***heap*** memory allocation

ListTestDriver.cpp:

```
int main() {
  // Object instantiation
  List profiles = List();
```

Memory layout



List.h #1

```
/* Header Comment Block */
...
class List  {
private:
  constexpr static int SIZE = 5;
  Profile elements[SIZE];
  unsigned int elementCount;

public:
...
  insert(...)
};
```

List.h #2

```
/* Header Comment Block */
...
class List  {
private:
  constexpr static int SIZE = 5;
  Profile * elements;
  unsigned int elementCount;

public:
  // Destructor
  ~List();
  ...
  insert(...)
};
```

```
delete [] elements;
```

in List.cpp

```
if (elementCount == 0)
  elements = new Profile[SIZE];
if (elements == nullptr) // error!
else // insert new element
```

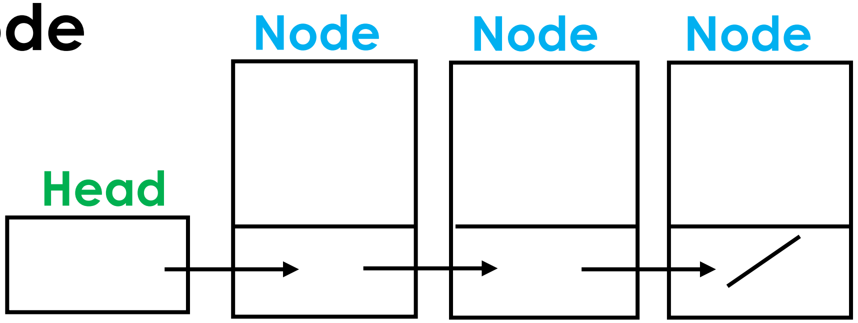
Step 3 – Array-based implementation of **List** ADT

- When designing and implementing our methods, consider ***memory management***
  - Differentiate between ***stack-allocated*** and ***heap-allocated*** arrays
  - When to use ***stack-allocated*** memory?
  - When to use ***heap-allocated*** memory?

- When using ***heap-allocated*** array, we need to consider:
  - Copy constructor
  - Overload *assignment* operator
  - `new` operator
  - Destructor
  - `delete [ ]` operator

Introducing 2nd concrete data structure: *linked list*

- Made of **pointers** and **node**



- Characteristics:
  - Adv:
    - Flexible/unbounded size: it grows or shrinks as needed
    - Operations on linked list do not require the shifting of elements
  - Disadv: Sequential access
    - Elements must be accessed in some specific order dictated by the links

What can we do with an linked list

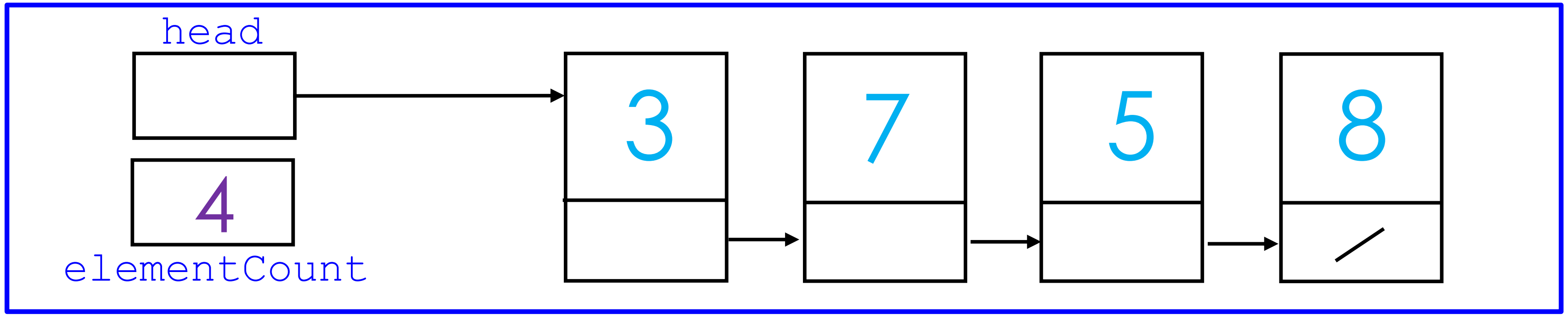
- Insert element into it
  - Prepend (insert at the front – position `1`)
  - Append (insert at the back – position `elementCount + 1`)
  - Insert at a particular position (between the above two positions)
  - Insert in **sort** order using the element's *search key*
- Remove element from it
  - Remove element from either position `1`, position `elementCount`, or any position in between
  - Remove element that matches the *search key* (require Searching)
- Traverse (iterate through)
  - Search for (find/get) a particular element – may not visit every element
- ~~Expand (resize) a linked list~~
- No gaps in a linked list to manage!

REVIEW

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

Insert an element into a linked list

- insert @ front (prepend)

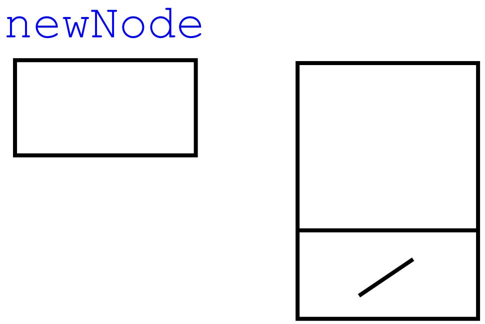


6

pseudocode

```
... prepend(int newElement) // prepend into a List
1. Node *newNode = new Node(newElement);
2. if (newNode != nullptr)
   3. newNode->next = head;   // Whether Head NULL or not
   4. head = newNode;
   5. elementCount++;
```

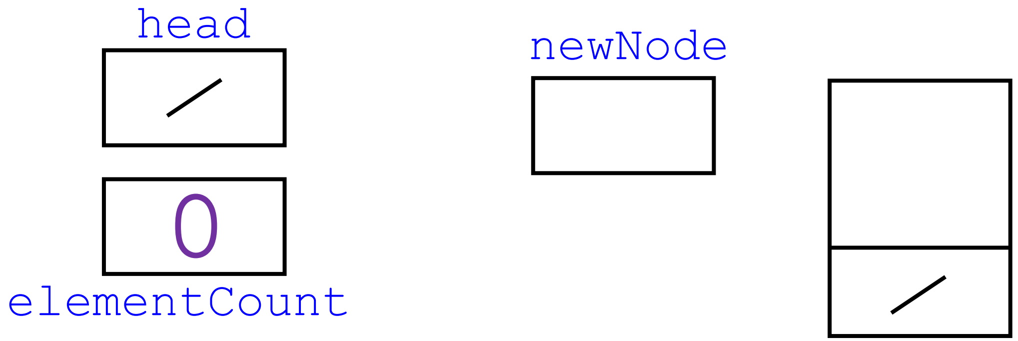
Order is important



REVIEW

Generalisation Principle

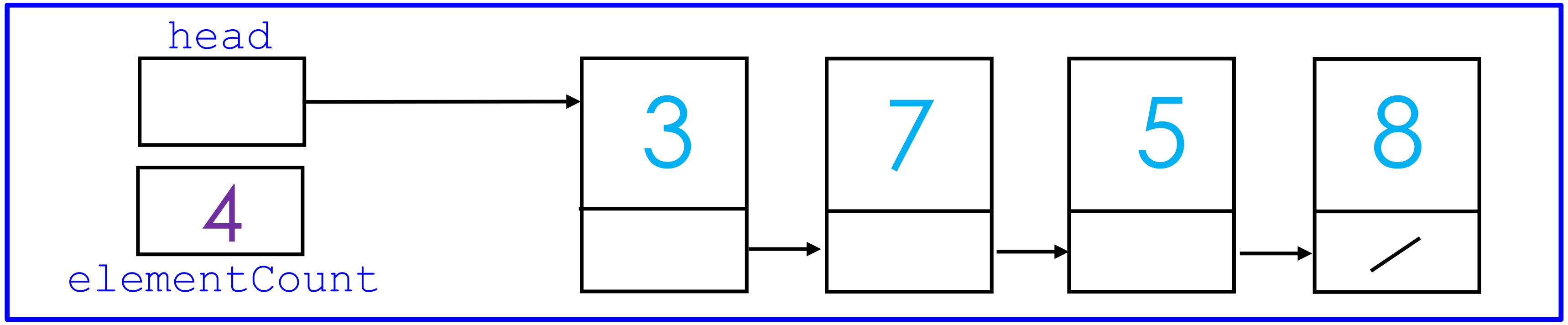
- Ensuring that our code works in all situations – all the states a linked list can have – all test cases:
  1. When linked list is empty
  2. When it has 1 element,
  3. When it has many elements,
  4. etc.
- Here, will our code work when the linked list is empty?



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

REVIEW

Traverse a linked list



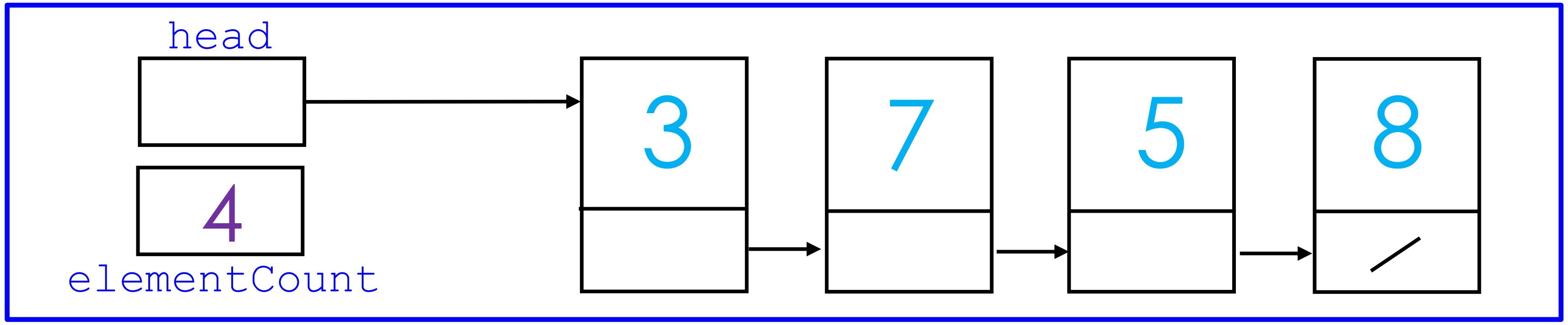
Local variable: current

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

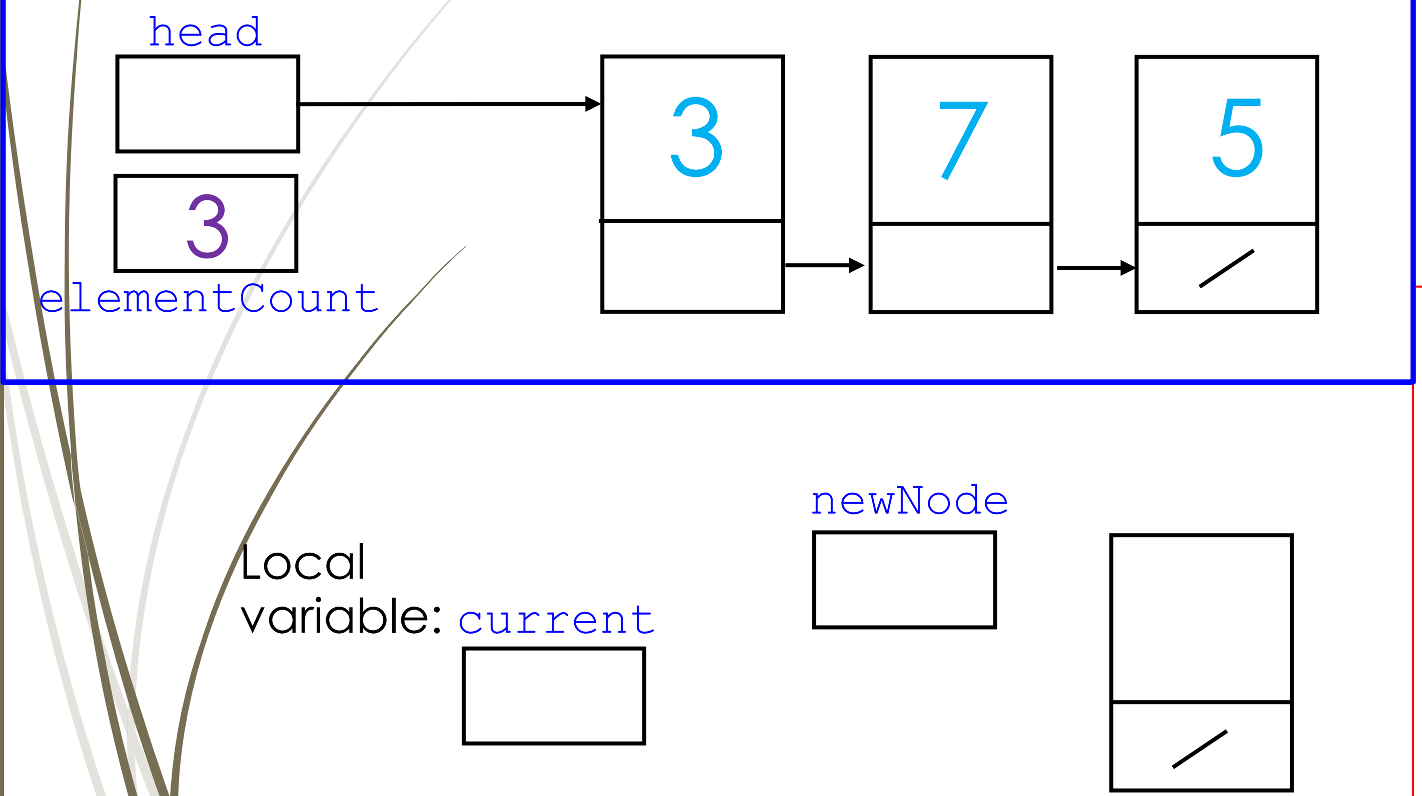
```
// Traverse linked list
1. if ( head != nullptr )
   2. while (head->next != nullptr)
      3. head = head->next;
```

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

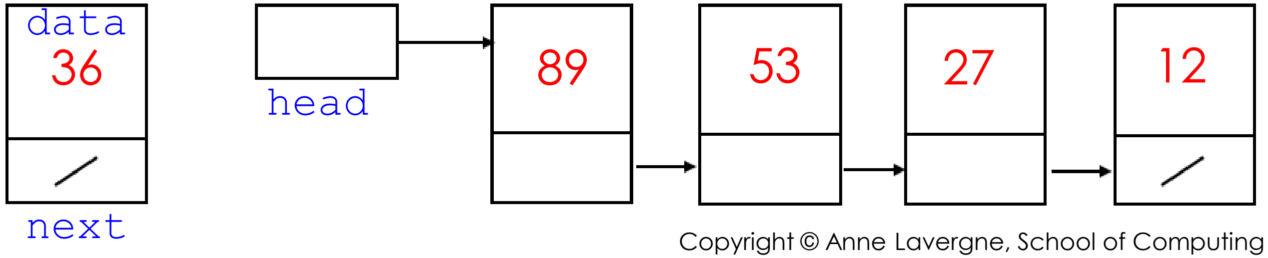
- When ***linked list*** is used as a data structure (CDT) for a value-oriented data collection ADT class like a List (which is kept sorted), in order to keep it sorted, ***we insert the element in sort order into the List*** using a ***search key*** (i.e., an element's attribute)
- Alternatively, we 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 1

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`:



✓Learning Check

- ✓ 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

Next Lecture

- 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