Last Lecture

- Class documentation
- Linked lists and its operations
Learning Outcomes

At the end of this lecture, a student will be able to:
- define one of the concrete data types, namely linked list, and demonstrate, simulate, and trace its operations
- write C++ code
- manipulate pointers
- manage memory in C++
Today’s menu

- Finish introducing linked list
- A second implementation of our List ADT class:
  - Link-based implementation of List ADT class
Insert an element into a linked list

- @ front

```c
... insert(int newElement) // insert of List ADT class
    Node *newNode = new Node(newElement);
    newNode->next = head; // Head NULL or not
    head = newNode;
    elementCount++;
```
Insert an element into a linked list

- Will our code work in all situations?
  - Here, will our code work when linked list is empty:

```c
... insert(int newElement) // insert of List ADT class
    Node *newNode = new Node(newElement);
    newNode->next = head;  // Head NULL or not
    head = newNode;
    elementCount++;
```
/ Anchor head of linked list
Node* current = head;
while (current -> next != NULL)
    current = current -> next;
@ end

... insert(int newElement)
Node *newNode = new Node(newElement);
if (head == NULL)
    head = newNode;
else
    // Move to the end of the list
    Node* current = head;  // Anchor
    while (current -> next != NULL)
        current = current -> next;
    current -> next = newNode;
    elementCount++;
Insert an element into a linked list

- @ specific location
  - When linked list is used as a data structure for a position-oriented data collection ADT like a List ADT
  
  OR

- When linked list is used as a data structure for a value-oriented data collection ADT like a List ADT, kept in sorted order
Remove an element from a linked list

@ front

```c
... remove( )
if (head != NULL)
    Node* nodeToRemove = head;
    head = head -> next;
    // Return node to the system
    nodeToRemove -> next = NULL;
    delete nodeToRemove;
    nodeToRemove = NULL;
    elementCount--;
```
Remove an element from a linked list

```c
if (head != NULL)
    // Move to the end of the list
    Node* current = head;  // Anchor
    while (current -> next != NULL)
        current = current -> next;
    // Then what???
```
Issue with Traverse

- Using “current -> next”?

- Using “current -> next -> next”?
  -> called “Look Ahead”
  - Advantage:

  - Disadvantage:
Removal – with previous

@ end
Insertion - Improvement

@ end

Diagram of sequential steps.
doubly headed singly linked list

- Advantages:
- Disadvantages:
Removal – Improvement

@ end

[Diagram of process flow]
singly headed doubly linked list

- Advantages:

- Disadvantages:
doubly headed doubly linked list

- Advantages:

- Disadvantages:
Removal

@ specific location
- When linked list is used as a data structure for a position-oriented data collection ADT like our List ADT

OR
- When linked list is used as a data structure for a value-oriented data collection ADT like our List ADT (kept in sorted or unsorted order)
linked lists are very flexible

- singly headed singly linked circular list

- singly headed doubly linked circular list
Be Careful

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

with *local variables* of various List ADT methods manipulating the linked list
  - Such as *current* and *previous*

and with *next* and *back* *data members* of the Node class
Activity - Problem Statement

- We are asked to develop a software application to manage customer accounts.
- These accounts are to be:
  - Printed in ascending alphabetical sort order of customer last name.
  - Printed in ascending numerical sort order of customer SIN.
  - A lot!

- Let’s design the underlying data structure of our ADT class:
  - Let’s create a linked list that would allow us to perform these operations in $O(n)$.
Implementation - Node Class

```
/*
 * Node.h
 *
 * Class Definition: Node of a singly linked list
 * in which the data is of "int" data type.
 *
 * Created on:
 * Author:
 */

// #pragma once is shorthand for the 2 #include guards we've seen before:
// #ifndef _NODE and #define _NODE
#pragma once

class Node
{
    public:
        // Public attributes - Why are the attributes public?
        int data;       // The data in the node
        Node* next;    // Pointer to next node

        // Constructors and destructor
        Node();
        Node(int theData);
        Node(int theData, Node* theNextNode);
}; // end Node
```
/*
 * Node.cpp
 *
 * Class Definition: Node of a singly linked list
 * in which the data is of "int" data type.
 *
 * Created on:
 * Author:
 */

Node::Node()
{
    data = 0;
    next = NULL;
}

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

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

// end Node.cpp
/*  * List.h  *
 *  *
 * Class Description: A list ADT - inspired from List from our textbook.
 * Class Invariant: List position of elements start at 1 (not 0).
 *  *
 * Note 1: This is a link-based implementation of a data collection List ADT class
 * as its underlying data structure is a singly-headed singly-linked list (SHSL).
 *  *
 * Note 2: The public methods have the same declaration as the methods of the array-based implementation of List class.
 *  *
 * Inspired on: January 2019
 * Author: AL
 */

#pragma once
#include "Node.h"

using namespace std;

class List {

private:
    int elementCount; // Number of elements currently stored in the list.
    Node *head; // Pointer to the first node in the list
Construct a List object (containing a linked list)

CODE:

```c
elementCount = 0;
head = NULL;
```
Learning Check

- We can now …
  - Create linked list
  - Perform operations on a linked list
  - Create a Node class
  - Implement a List ADT class:
    - Using an array
    - Using a linked-list
Next Lecture

- Review of the Big-O notation
- Compare the two implementations of our List ADT class:
  - Array-based implementation
  - Link-based implementation