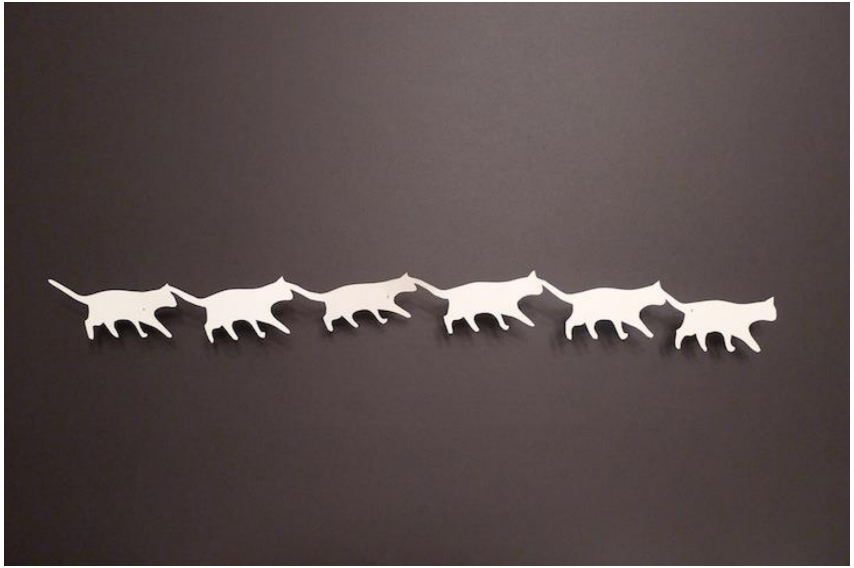
SEARCHING AND LINKED LIST



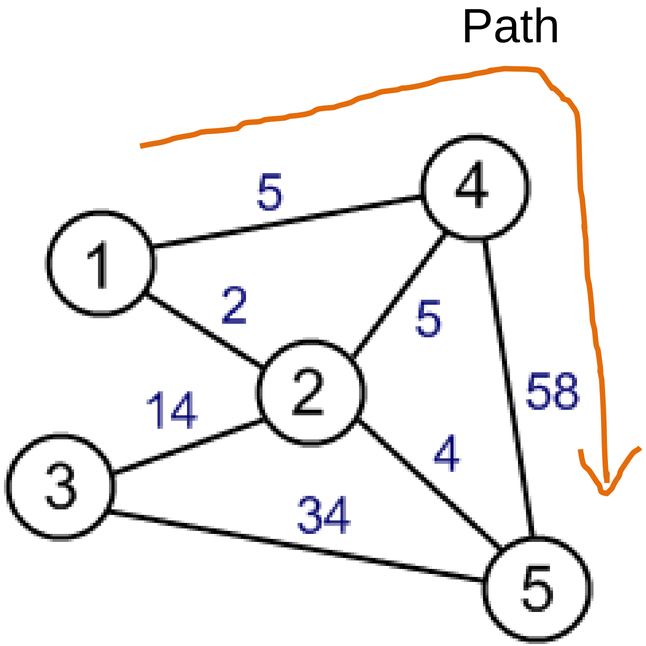
BME 121 2016
Jeff Luo

Topics

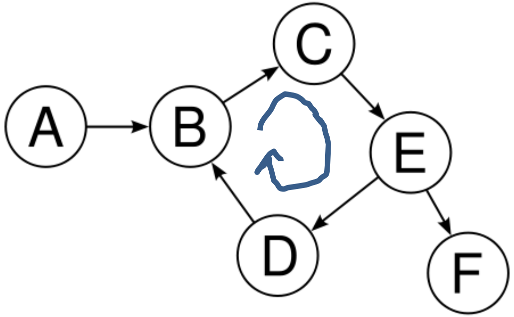
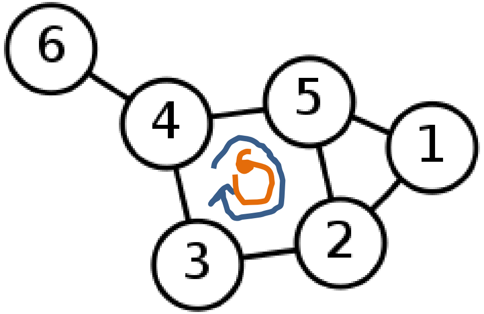
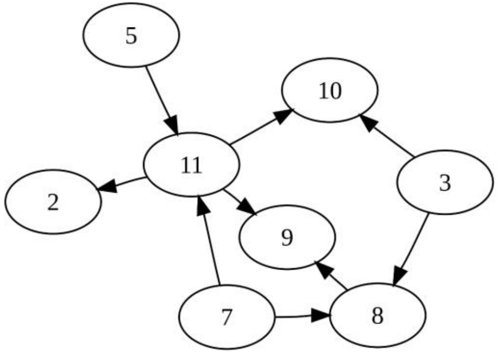
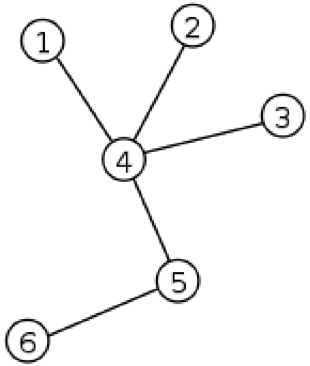
- The Graph Data Structure
- Searching
  - Binary Search of an Array
- WA7
- The Linked List Data Structure
  - Printing
  - ToArray
  - Find

The Graph Data Structure

- Organizes Nodes into a network
- Graphs have a special terminology:
  - A node is called a Vertex
  - A connection between a pair of vertices is called an Edge
  - Each edge can have a numerical weight, and directions



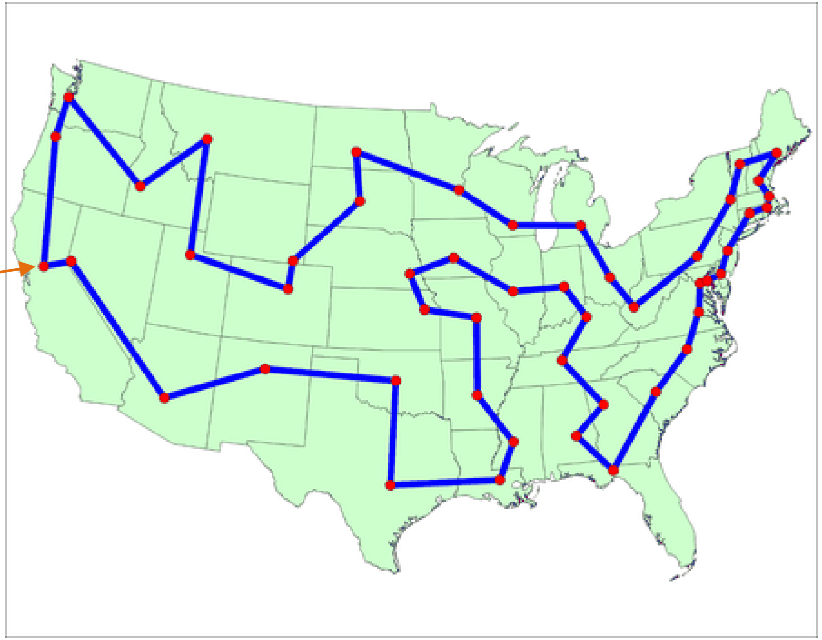
Four major styles of Graphs

	Directed	Undirected
Cyclic		
Acyclic (no cycles)		 * Basically a Tree, but no specific Root node

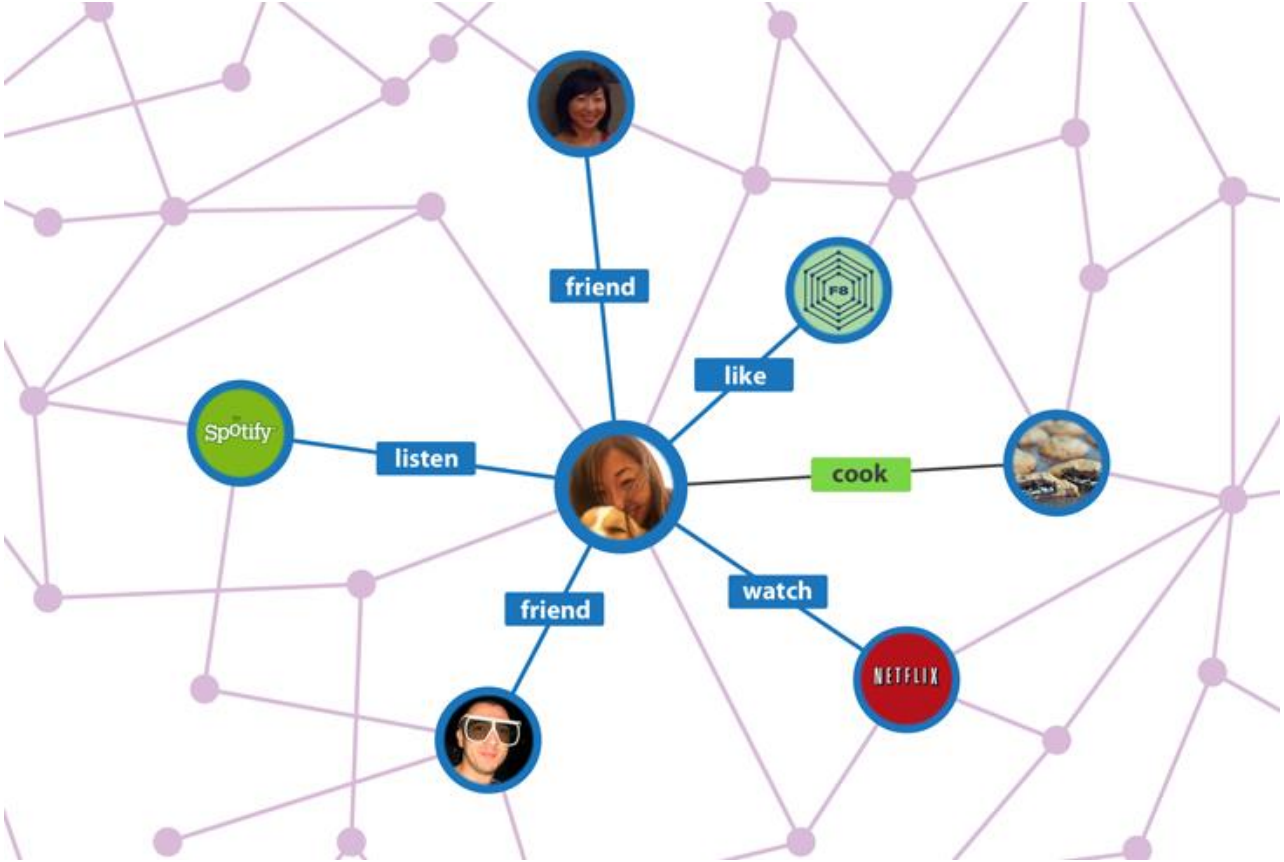
Graphs in use

- Traveling Salesman Problem: Calculate the shortest path to visit each (major) city of a country at most once.

(Ideas) Start Here!



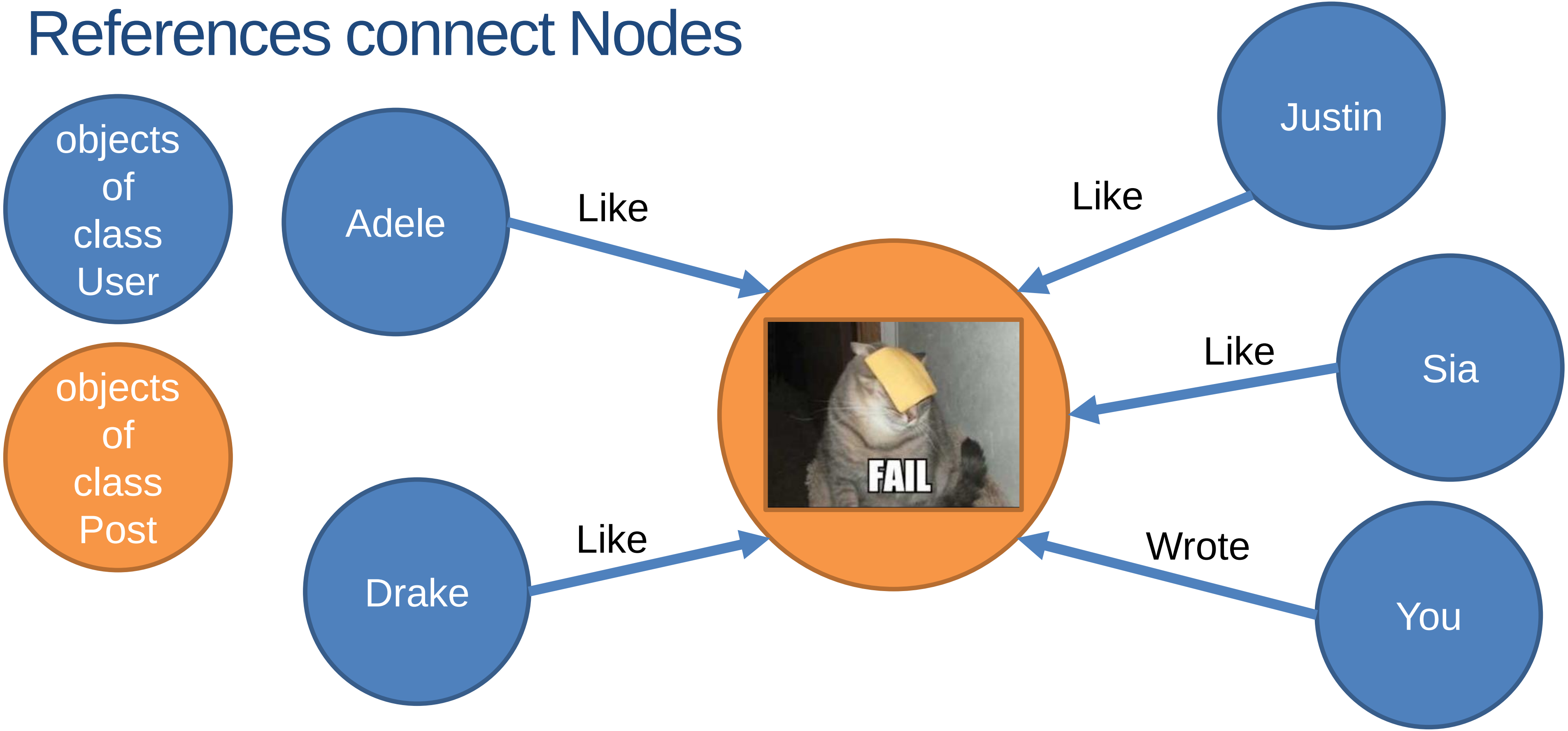
Graphs in use



Nodes/Vertices represent people or things

Edges represent associations or relationships

References connect Nodes



To be efficient, we don’t create more than one object for each post, or user

Basic Code for a Node

- Linked List / Stack / Queue:

```
class Node {
    Type data;
    Node next;
    // methods…
}
```

- Tree:

```
class Node {
    Type data;
    Node[] children;
    // methods…
}
```

- Graph:

```
class Node {
    Type data;
    Node[] neighbours;
    // methods…
}
```

- The difference is in how many other nodes each node references (points to), and how the overall structure looks like

The Searching Problem

- At a high level, searching is actually about finding a solution to a problem that satisfies a set of constraints, in a search space that contains many candidate solutions.
- Ex 1:
  - Problem: Find missing sock
  - Constraint: Has a pattern matching the one I have in my hand
  - Search Space: Closet
- Ex 2:
  - Problem: Find min value of an array
  - Constraint: Smallest value
  - Search Space: A bunch of values organized in an array
    - Each number in the array is a candidate for being the min value

The Searching Problem

- It is common to use the size of the search space as a way to define how difficult the search problem will be:
  - Finding a sock in a room vs a whole house (multiple rooms)
  - Finding min value in a 1D array vs a 100D array

- The main steps to a search problem is:
  - Model and bound the search space
  - Come up with a way to find the optimal solution in the search space
  - Improve the speed at which we find this optimal solution

Brute-Force Search

- Idea: explore the entire search space and examine every possible solution to see if it fits our constraint
  - Socks: look inside every compartment of the closet
  - Min value: look at every single value in the array
- Advantages:
  - Straight forward
  - Does not need any kind of organization of the data in the search space (always adaptable to new problems)
- Disadvantages:
  - Slow: the solution is usually just 1 data point out of the entire space, and the space is really big in real world data problems. Examining every data point is going to take forever.

Brute-Force Search

- In sequentially ordered data structures (arrays, linked lists), it simply involves sequentially accessing <u>every element</u> of the data structure
  - Forward Search: from 0 to Length – 1 (or head to tail)
  - Backward Search: from Length – 1 to 0 (or tail to head)
  - 2D arrays: look at every row and column in some order

- Examples:
  - 1D array: Min(), Max(), Median(), …
  - Strings: IndexOf(), LastIndexOf(), Contains(), …

Brute-Force Search

Heuristics Based Search

- Heuristic: a math function that ranks alternatives in each step of a search algorithm based on available information to decide what area of the search space to explore in the next step
  - I.e., look at a restricted subset of the entire search space

- Example: what if an array was sorted from smallest to largest?
  - Min() -> only look at the first value in the array
  - Max() -> only look at the last value in the array
  - Median() -> only look at the value at index [0 + Length – 1] / 2

- When the array has millions of elements, this will speed it up a lot!

Heuristics Based Search

- Advantages:
  - Faster, as it takes advantage of the organization of data in the search space
- Disadvantages:
  - Complex analysis
  - Organizing the data in computer memory may be more expensive than the search task
  - Solutions are not always adaptable to new problems
- To solve a problem using heuristics based search, it takes 2 steps:
  - Organize the information in the search space
  - Come up with an algorithm based on this organization

Binary Search

- Binary Search cuts the search space in half in each step of the algorithm.
- Requires a sorted array as its input.
- Depending on the search value, it (repeatedly) looks at the left or the right half of the (sub)array.

- Problem: Given a sorted array, find and return the index of a value in the array, or -1 if it doesn't exist in the array

- Ex, given `int[] x = new int[] { 1, 2, 3, 5, 8, 9, 11 }`
  write `Find(int searchValue)`

Binary Search Procedure

- Start minIndex at 0, maxIndex at Length-1
- Repeat:
  - Compute middleIndex as (maxIndex + minIndex) / 2
    - x[middleIndex] is the current median value
  - Compare searchValue to the median value:
    - If it is equalled to the median then return that index
    - If it is smaller than the median then look at the left half of the array (set maxIndex = medianIndex - 1)
    - If it's greater than the median then look at the right half of the array (set minIndex = medianIndex + 1)
  - If we can't find it (if the minIndex and maxIndex cross over), return -1

Binary Search, Visualized

x.Find(8)

index	0	1	2	3	4	5	6
value	1	2	3	5	8	9	11

minIndex

maxIndex

middleIndex

8 > 5

8 < 9

8 == x[middleIndex],
so we return middleIndex

median = x[middleIndex]
middleIndex = (minIndex + maxIndex) / 2

Binary Search, Visualized

x.Find(7)

index	0	1	2	3	4	5	6
value	1	2	3	5	8	9	11

minIndex 0

4

3 maxIndex 4 6

middleIndex

7 > 5

7 < 9

7 < 8

maxIndex < minIndex,
so we return -1

median = x[middleIndex]
middleIndex = (minIndex + maxIndex) / 2

Binary Search Code

- Open up BinarySearch.cs on OneDrive to see the code!

WA7

- Write a loop to count the number of “Drug” objects where the drug name contains the string “VITAMIN”
  - Hint: “hello world”.Contains(“hello”) == true

- Create an array of “Drug” objects of that size

- Write a loop to fill the array with those “Drug” objects where the drug name contains the string “VITAMIN”

- Write a loop to display each “Drug” object in the array on the console.

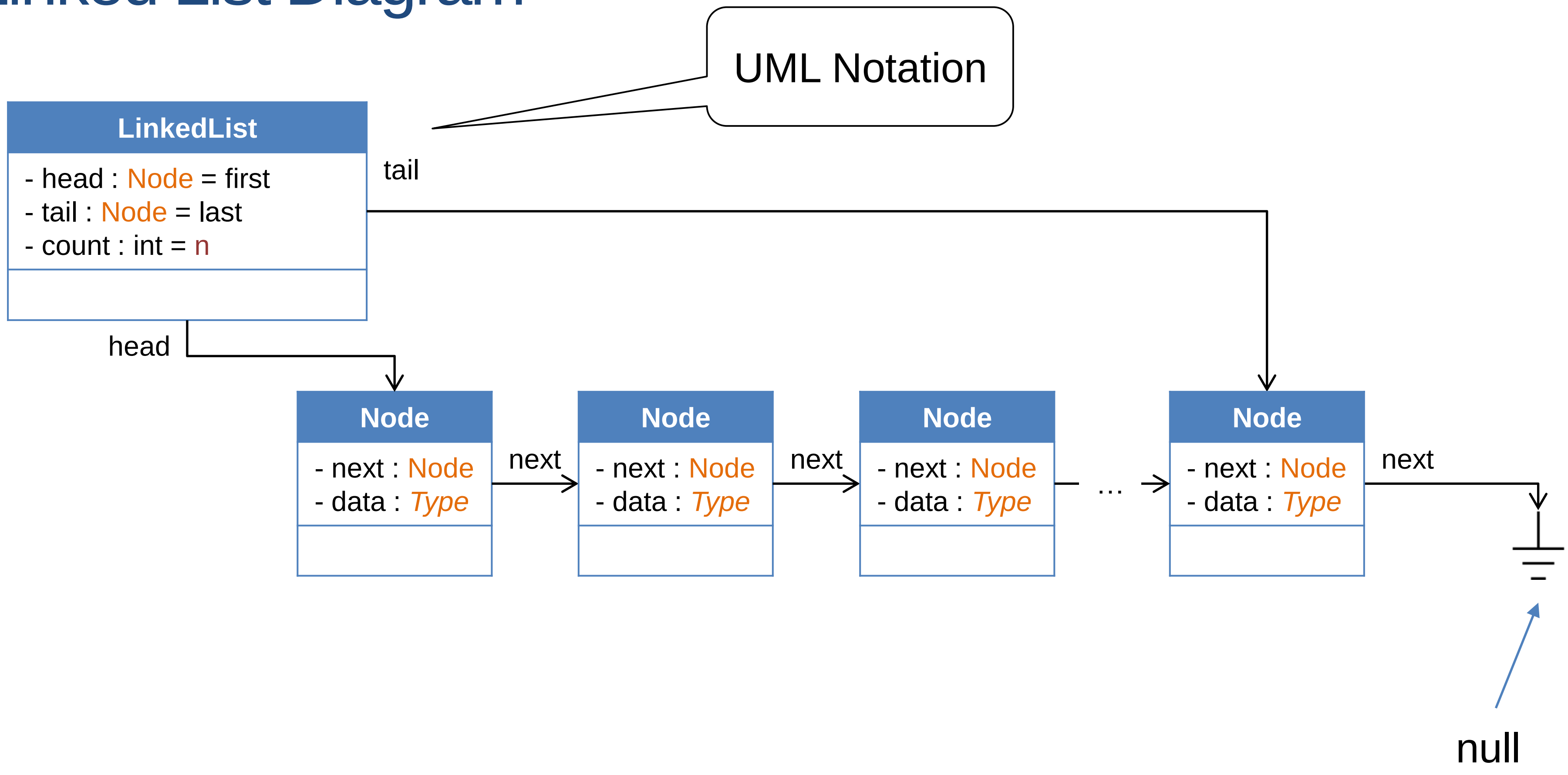
The Linked List Data Structure

- A Linked List is a sequential data structure which grows in size depending on the number of data elements we want to have in it.
  - Array is also a sequential data structure, but Arrays are fixed in size once it is created.
- Linked List adds nodes (normally to the end) for each element we put into the Linked List.

- Note: C# has an implementation of Linked List, specifically class LinkedList<T> from System.Collections.Generic

- We are going to write our own version of LinkedList to study this data structure

Linked List Diagram



Linked List Basic Code Structure

```
class Node {
    Type data;
    Node next;
    // methods…
}
```

- Each node stores one element:
- Field `data` stores some value
- Reference `next` points to the next node in the chain

```
class LinkedList {
    Node head;
    Node tail;
    int count;
    // methods…
}
```

- Reference `head` always points to the first node
- Reference `tail` always points to the last node
- `count` keeps track of how many nodes we have in the linked list

Linked List Operations

- Printing a Linked List:

- Create a temporary reference and start it at head node
- Loop until we've reached pass the tail node
  - Display current node's contents
  - Advance to next node

- Converting to Array:

- Create an array with size = count
- Create a temporary reference and start it at head node
- Create an array index variable and start it at 0
- Loop until we've reached pass the tail node
  - Copy the current node's content into the array at the current index
  - Advance to next node
  - Index++
- (similar to WA7)

Linked List Operations

- Forward Brute Force Searching:

- Create a temporary reference and start it at head node
- Loop until we’ve reached pass the tail node
  - If current node’s contents match what we are searching for, return a reference to that node
  - Otherwise advance to next node
- After the loop (ie it cannot be found), return null

Linked List Operations

- Adding and removing nodes to a linked list requires Case Analysis:

- Appending a node to the tail of a list (2 cases):

  Case 1: no nodes exist in the list

  → add the node at head

  → set tail to also point to this node

  → count++

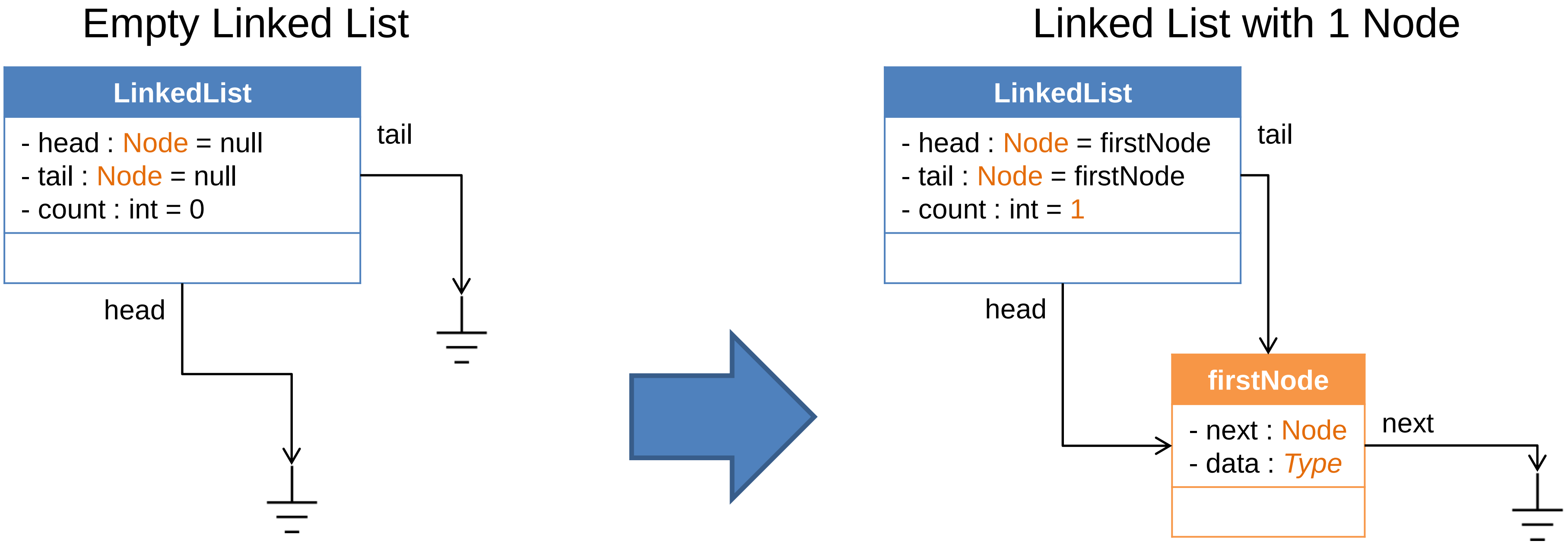
  Case 2: 1 or more nodes exist in the list

  → add the node after tail

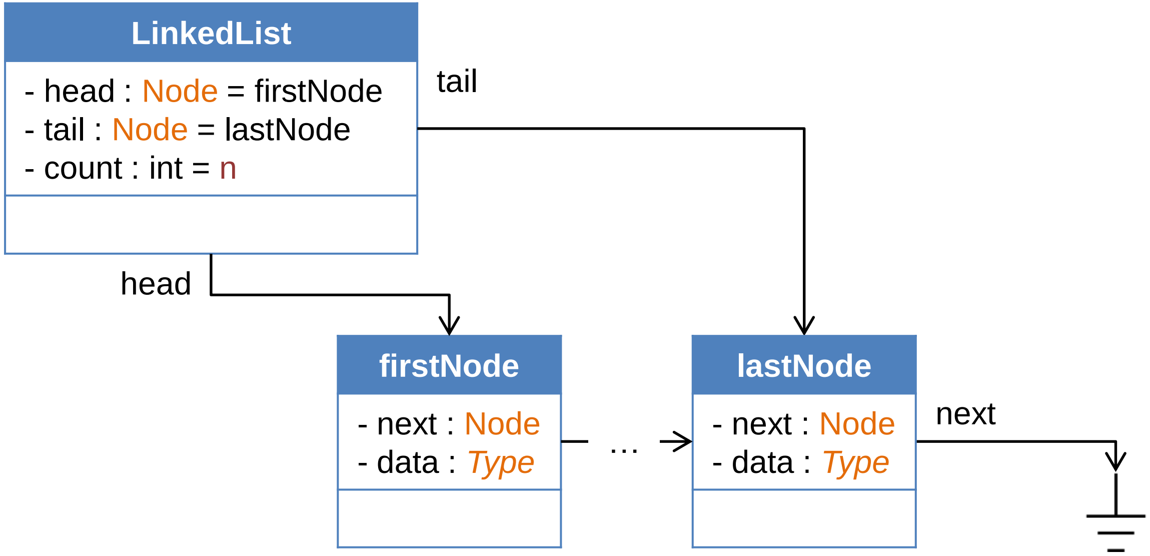
  → update tail to point to this new node

  → count++

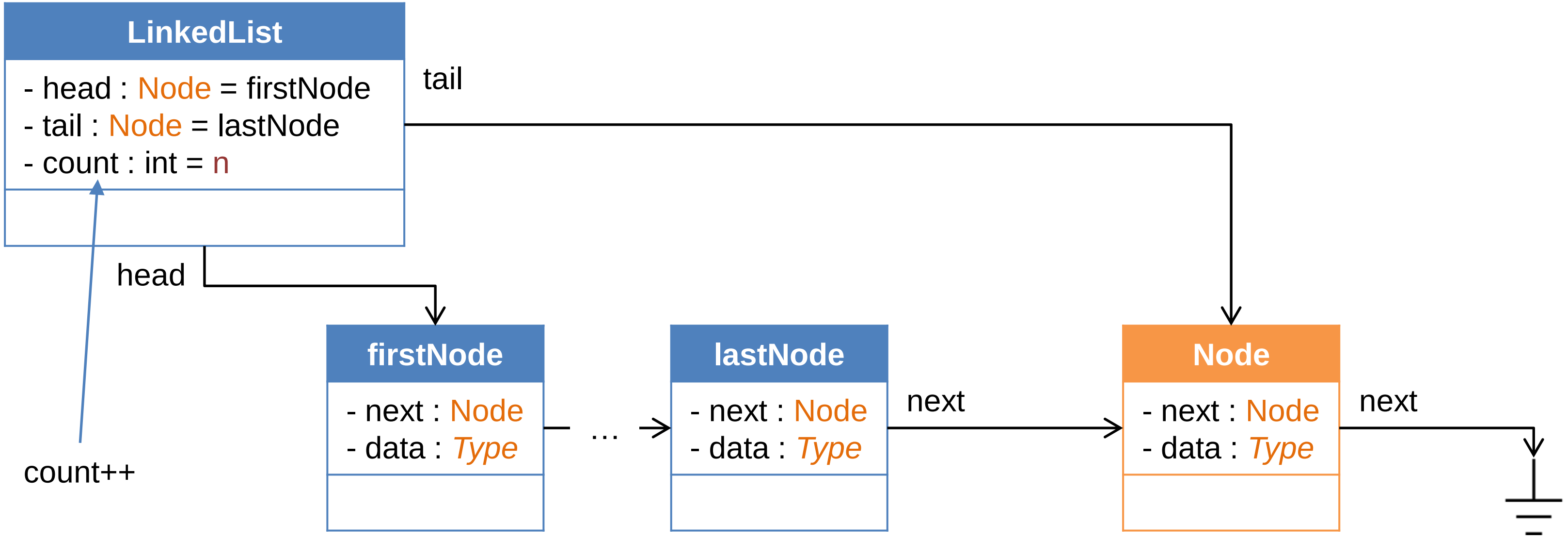
Appending to a Linked List – Case 1



Appending to a Linked List – Case 2 (Before)



Appending to a Linked List – Case 2 (After)



Linked List Case Analysis

- Generally, being with the following cases:
  - Case 1: 0 nodes
  - Case 2: 1 node
  - Case 3: 2 nodes
  - Case 4: many nodes
- Depending on the operation, some of the cases overlap. We will go through the other operations on Friday!

- Adding: Append(), Prepend(), InsertAt(index), InsertInOrder()
- Removing: RemoveHead(), RemoveTail(), RemoveAt(index)
  RemoveMin(), RemoveMax()

Linked List Code

- Open up LinkedList.cs on OneDrive to see the code!