Introduction to Computer Science
Reichman University

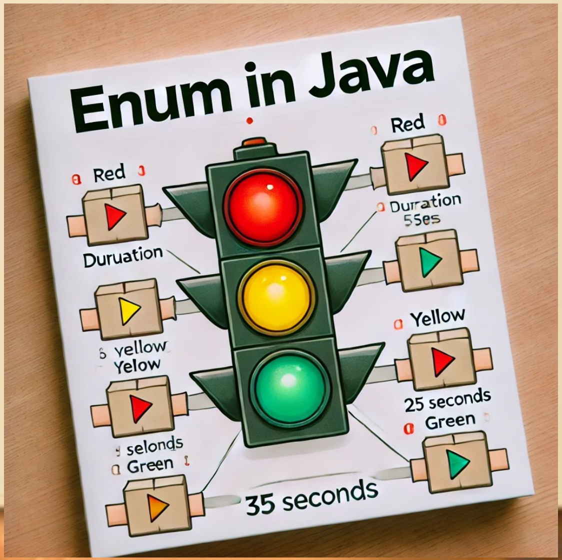
Recitation 10

Overview

- Enums
- Complexity
- Search
  - Binary Search
- Linked List
  - Node
  - Linked List

Recitation 10

Enums



What is Enum?

- Up to this point we have worked with primitive types, arrays, and Strings. However, the world is very complex, and not all thing can be defined into those types.
- Most of them can be represented by some combination of those types (this will be covered next week) however, there are some cases that when we want to define a type for a fixed number of values. For that we can use **enum.**
- Enum (short for enumeration) is a special data type in Java used to define a set of constants.
- These constants are often related and belong to the same category.
- Some examples:
  - Days of the week (Sunday, Monday etc.)
  - Months of the year (January, February etc.)
  - Suit of cards (Heart, Daimond etc.)
  - Value of cards (2, 3, …., 10, Jack, Queen, King, Ace)
  - Chess pieces (Pawn, Rook, etc.)
  - (Many more)

Why should we type them and not use existing types?

- **Type safety** - It will limit the usage of user to certain values, keeping to those values can prevent bugs and make the code more reliable.
- **Readability** – will make thing much more readable.
- **Prevent Magic values** – As you recall magic numbers is the concept of assigning fixed values of numbers/strings without context. Those values are considered bad practice. Those give much needed context
- **Reusable** - can be used in other places of code.
- **Iteration** – all enums come with the built-in method {enumName}.values() which returns an array of all enum values which in fact allows you to iterate over them.

Many more…

Enums Example

- Traffic lights usually have 3 colors
  - Red -> The cars should stop driving
  - Yellow -> The cars should slow down
  - Green -> The cars should go
- They move in cycle: Red -> Yellow -> Green -> Red

Non-Enum Example

```
public static int final RED = 0;
public static int final GREEN = 1;
public static int final YELLOW = 2;
public static String[] lightNames = [“RED”, “GREEN”, “YELLOW”];

public int getNextLight (int light) {
        int nextLight = RED;
        switch (light) {
            case RED:
                nextLight = GREEN;
                break;
            case YELLOW:
                nextLight = RED;
                break;
            case GREEN:
                nextLight = YELLOW;
                break;
            default:
                nextLight = RED;
                break;
        }
        System.out.println(“Current light: " + lightNames[light] + “Next light:" + lightNames[nextLight]);
        return nextLight;
}
```

Enums Example

```
public enum TrafficLightEnum {
    RED(“RED”),
    YELLOW(”YELLOW”),
    GREEN(”GREEN”);

    private final String light;

    TrafficLightEnum(String light) {
      this.light = light;
    }

    public String getLight() {
      return this.light;
    }

}
```

Enums Example

```
 public TrafficLightEnum getNextLight (TrafficLightEnum light) {
        TrafficLightEnum nextLight = TrafficLightEnum.RED;
        switch (light) {
            case RED:
                nextLight = TrafficLightEnum.YELLOW;
                break;
            case YELLOW:
                nextLight = TrafficLightEnum.GREEN;
                break;
            case GREEN:
                nextLight = TrafficLightEnum.RED;
                break;
            default:
                nextLight = TrafficLightEnum.RED;
                break;
        }
        System.out.println(“Current light: " + light + “ Next light: " + nextLight);
        return nextLight;
}
```

Enums Example

```
public CarActionEnum decide(TrafficLightEnum light) {
        CarActionEnum action = CarActionEnum.STOP;
        switch (light) {
            case RED:
                action = CarActionEnum.STOP;
                break;
            case YELLOW:
                action = CarActionEnum.SLOW_DOWN;
                break;
            case GREEN:
                action = CarActionEnum.GO;
                break;
            default:
          action = CarActionEnum.STOP;
          break;
        }
        return action;
}
public static void iterateEnum() {
        TrafficLightEnum[] lights = TrafficLightEnum.values();
        for (int i = 0; i < lights.length; i++) {
            System.out.println(lights[i]);
        }
}
```

```
public enum CarActionEnum {
    SLOW_DOWN("Slow Down"),
    STOP("Stop"),
    GO("Go"),

    private final String message;
    ...

}
```

Recitation 10

Complexity

Complexity

- There are 2 kinds of complexity.
  - Time
  - Space
- When it comes to this course when we refer to complexity we refer to Time Complexity.

Complexity

- Complexity is a measure used to analyze the efficiency of algorithms.
- It describes the amount of time an algorithm takes to run as a function of the size of its input.
- Complexity is typically expressed using Big O notation, which provides an upper bound on the growth rate of the algorithm's running time relative to the size of the input.
- In simpler terms, it helps us understand how the execution time of an algorithm increases with the size of the input data. This analysis helps in comparing different algorithms and predicting their performance on large inputs.

Complexity - Examples

- **<u>Constant complexity</u>** – O(1) - describes an algorithm whose execution time does not depend on the size of the input data. Regardless of the size of the input, the algorithm takes a constant amount of time to complete its execution.
  - Math operations (*,+,-,/,%)
  - Boolean operations (&&,||,!)
  - Logical operations (>,<,!=,== etc.)
  - Field access (In OOP)
  - Access any element in Array.

- **<u>Linear complexity</u>** – O(n) - describes an algorithm whose execution time increases linearly with the size of the input. In other words, as the size of the input data increases, the time taken by the algorithm to complete its execution also increases proportionally.
  - Iterating over 1D array/string
  - Find the middle element in LinkedList

- **<u>Quadratic complexity</u>** – O(n^2) - escribes an algorithm whose execution time grows quadratically with the size of the input data. In simpler terms, as the input size increases, the time taken by the algorithm to complete its execution increases quadratically, proportional to the square of the input size.
  - Iterating over a 2D array sized (nXn)

Recitation 10

Binary Search

Sequential Search

- Given an array and an element x, return the index of x in the array, returns -1 if x doesn't appear.

```
public static int sequentialSearch(int [] arr, int x) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == x) {
            return i;
        }
    }
    return -1;
}
```

~~**Run time complexity (in worse case)**~~: $O(n)$

Binary Search

- If the array is sorted, we can do better.
- The idea is that at each step we divide the search interval in half.
  - Begin with an interval covering the whole array.
  - If the value of x is less than the item in the middle, narrow the interval to the lower half.
  - Otherwise narrow it to the upper half.
  - Repeatedly check until the value is found or the interval is empty.

Binary Search

```
public static int binarySearch(int[] arr, int x){
         return binarySearch(arr, 0, arr.length - 1, x);
}
```

```
private static int binarySearch(int[] arr, int start, int end, int x){
     if (end < start){
         return -1;
     }
     int mid = (start + end)/2;
     // If the element is present at the middle itself
     if (arr[mid] == x) {
        return mid;
     }
     if (x < arr[mid]){
         return binarySearch(arr, start, mid - 1, x);
     }
     // else the element can only be present in right subarray
     return binarySearch(arr, mid + 1, end, x);
}
```

Binary Search

```
private static int binarySearch(int[] arr, int start, int end, int x){
     if (end < start){
         return -1;
     }
     int mid = (start + end)/2;
     // If the element is present at the middle itself
     if (arr[mid] == x) {
        return mid;
     }
     if (x < arr[mid]){
         return binarySearch(arr, start, mid - 1, x);
     }
     // else the element can only be present in right subarray
     return binarySearch(arr, mid + 1, end, x);
}
```

Running Time in worse case:

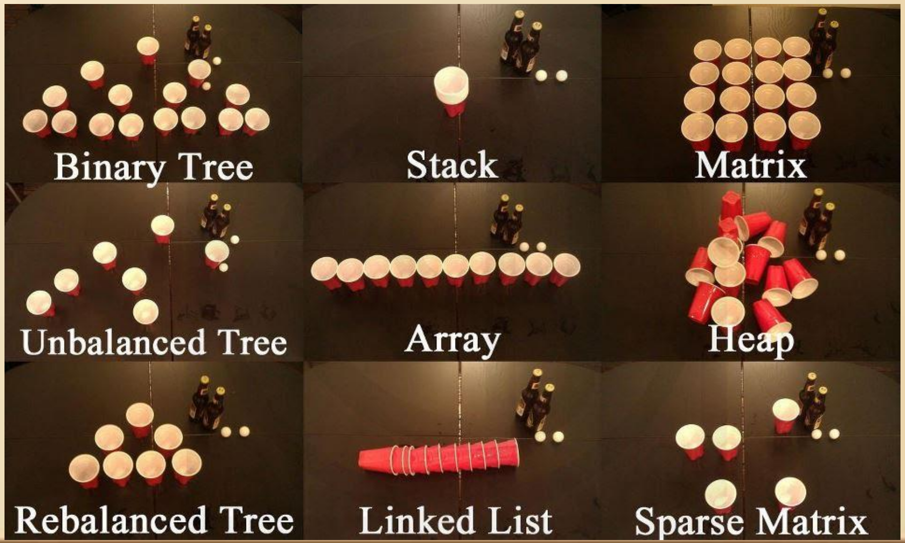
At each step we divide the interval into half, and stop when the interval size is 1.

$$\approx \log(n)$$

Introduction to Computer Science
Reichman University

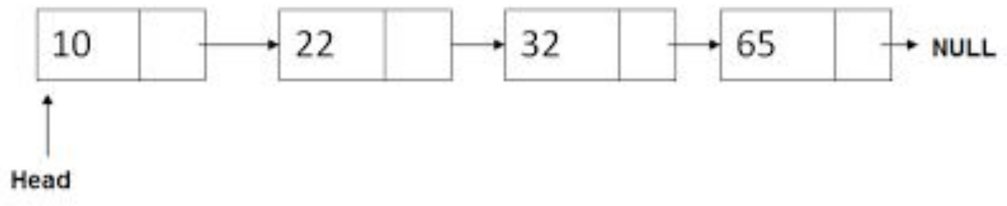
Recitation 10

Linked Lists



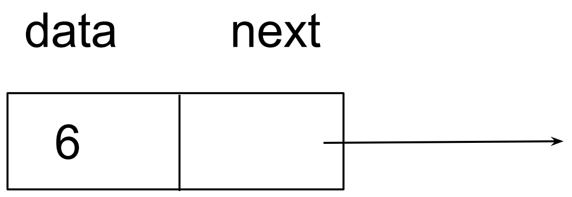
Linked Lists - overview

- Linked lists are among the simplest and most common data structures.
- Unlike arrays, linked lists allow us to dynamically allocate memory as needed.
- This means we do not have to know in advance the size of the list.
- This data structure is actually composed of 3 classes:
  - Node – holds one element of the list, this is where the magic happens.
  - LinkedList – wraps the entire list, contains functions to manipulate the list.
  - ListIterator- helper class for iterating over the list. Much in the same way a *for* loop is used for iterating over an array.



Node

- The nodes are the atoms of the list.
- Each node represents one link of the linked list.
- Nodes usually have at lease two fields:
  - **data** – the element contained in the node. (other names: value etc.)
  - **next** – a pointer to the next node in the list. (other field Node can have: prev, left, right etc.)
- This means every node in the list has 'access' to all succeeding nodes.
- So, it is enough to know the head node, in order to know the entire list.



Node

```java
/** Represents a node in a linked list.  A node has a value and a pointer
 * to another node. */
public class Node {

   int data;
   Node next;

   /** Constructs a node with the given data. The new node will point to
    *  the given node (next). */
   Node(int data, Node next) {
      this.data = data;
      this.next = next;
   }

   /** Constructs a node with the given data. The new node will point
    *  to null */
   Node(int data) {
      this(data, null);
   }


}
```

Linked List

- A list is composed of a sequence of nodes.
- As explained before, the only field really necessary for the list is:
  - first/head – a pointer to the first element of the list.
- In practice, we use some more 'service' fields to increase efficiency, such as:
  - size – the size of the list.
  - last/tail – a pointer to the last element of the list.
- Note: you will not always have those fields. read the documentation.

Linked List

```
public class LinkedList {
    private Node first;
    private int size;

    /** Constructs an empty list. */
    public LinkedList();

    /** Inserts the given element at the beginning of this list */
    public void addFirst (int elem);

    /** Inserts the given element at the end of this list */
    public void addLast (int elem);

    /** Returns the  first
        location of the element in this list, or -1 is no such element. */
    public int indexOf (int elem);

    /** Removes the first occurrence of the given element from this list. */
    public void remove (int elem);

    /** Returns the data at the specified position in this list. */
    public int getData(int index)

// More LinkedList methods follow.
```

Question 1 – Linked List Constructor

- Design a program in Java where you write a constructor method for a LinkedList class. The program doesn't any other elements. And create an empty LinkedList
- Here is the client code for help

Client code

```
LinkedList q = new LinkedList();
// Creates an empty list
```

Question 1 – Solution

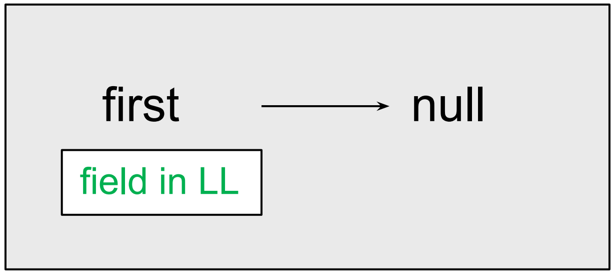
```
public LinkedList () {
    this.first = null;
    this.size = 0;
}
```

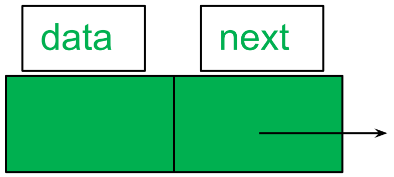
Question 1 – solution visualization.

```java
public LinkedList () {
    this.first = null;
    this.size = 0;
}
```

Client code

```java
LinkedList q = new LinkedList();
// Creates an empty list
```





Question 2 – addFirst

- Design a program that implements the 'addFirst(int elem)' method. This method inserts an element at the beginning of a LinkedList.
- Think: what are the steps required?
- Here is the client code for help

Client code

```
LinkedList q = new LinkedList();
q.addFirst(7);
q.addFirst(3);
// Creates the list (3 7)
```

Question 2 – addFirst – Plan

Steps

1. Creating a new Node which contains the value passed through the variable 'elem' to the function. and store it in a variable.

Why? My object my rules. I need to wrap the data in way which can be worked with my object.

2. Assigning the new node's pointer next to point towards the first element.

Why? If we don't do that we will lose the data we already stored

3. Assigning the linked list first field to point toward the new node.

Why? We must connect the first field to the new first element of the list otherwise we won't have access it

4. Increase the size by 1.

Why? We maintain the correctness of the size at all time

Question 2 - Solution

```
public void addFirst(int elem) {
    Node newNode = new Node(elem);
    newNode.next = this.first;
    this.first = newNode;
    this.size++;
}
```

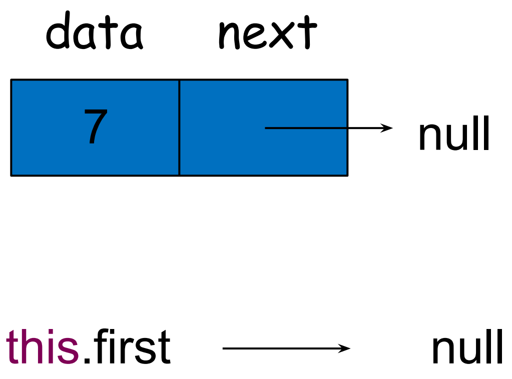
Question 2 – Solution visualization

```
public void addFirst(int elem) {
    Node newNode = new Node(elem);
    newNode.next = this.first;
    this.first = newNode;
    this.size++;
}
```

Client code

```
LinkedList q = new LinkedList();
q.addFirst(7);
q.addFirst(3);
// Creates the list (3 7)
```

line 2 client,
line 1 function



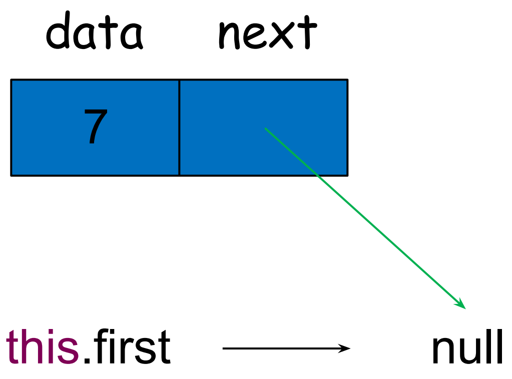
Question 2 – Solution visualization (Continued)

```
public void addFirst(int elem) {
    Node newNode = new Node(elem);
    newNode.next = this.first;
    this.first = newNode;
    this.size++;
}
```

Client code

```
LinkedList q = new LinkedList();
q.addFirst(7);
q.addFirst(3);
// Creates the list (3 7)
```

line 2 client,
line 2 function



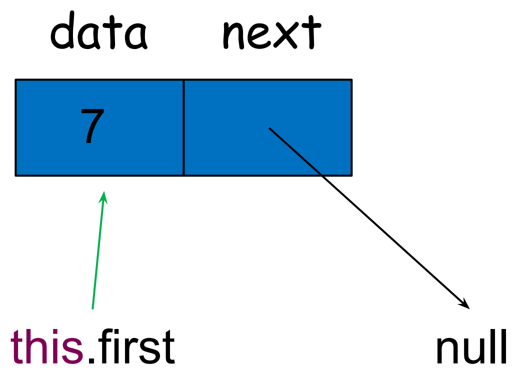
Question 2 – Solution visualization (Continued)

```
public void addFirst(int elem) {
    Node newNode = new Node(elem);
    newNode.next = this.first;
    this.first = newNode;
    this.size++;
}
```

Client code

```
LinkedList q = new LinkedList();
q.addFirst(7);
q.addFirst(3);
// Creates the list (3 7)
```

line 2 client,
line 3 function



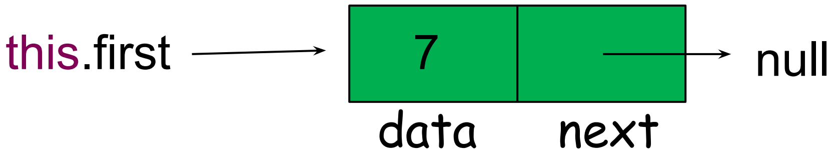
Question 2 – Solution visualization (Continued)

```java
public void addFirst(int elem) {
    Node newNode = new Node(elem);
    newNode.next = this.first;
    this.first = newNode;
    this.size++;
}
```

Client code

```java
LinkedList q = new LinkedList();
q.addFirst(7);
q.addFirst(3);
// Creates the list (3 7)
```

line 2 client,
line 3 function



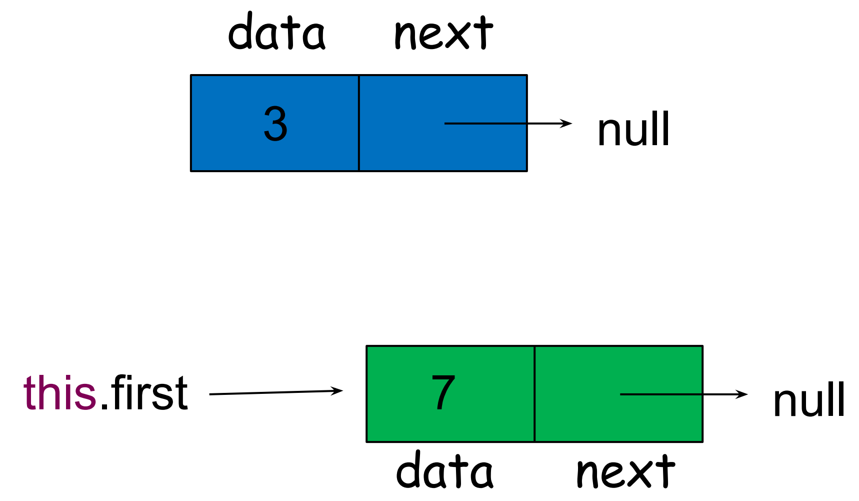
Question 2 – Solution visualization (Continued)

```
public void addFirst(int elem) {
    Node newNode = new Node(elem);
    newNode.next = this.first;
    this.first = newNode;
    this.size++;
}
```

Client code

```
LinkedList q = new LinkedList();
q.addFirst(7);
q.addFirst(3);
// Creates the list (3 7)
```

line 3 client,
line 1 function



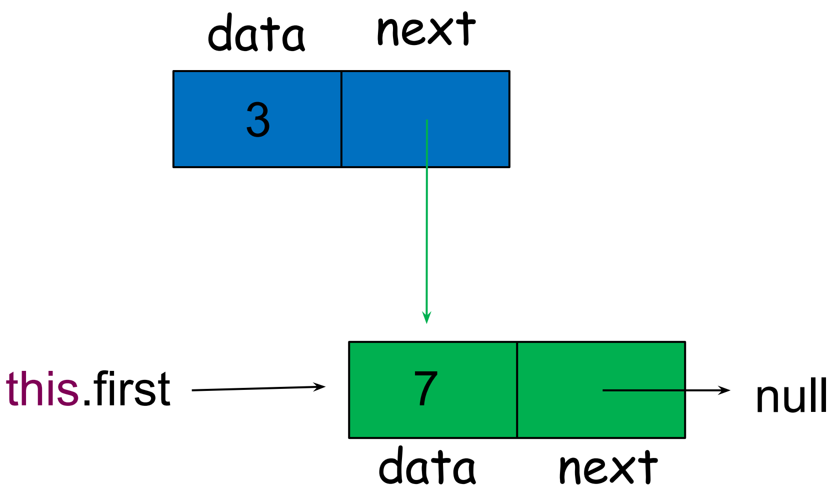
Question 2 – Solution visualization (Continued)

```
public void addFirst(int elem) {
    Node newNode = new Node(elem);
    newNode.next = this.first;
    this.first = newNode;
    this.size++;
}
```

Client code

```
LinkedList q = new LinkedList();
q.addFirst(7);
q.addFirst(3);
// Creates the list (3 7)
```

line 3 client,
line 2 function



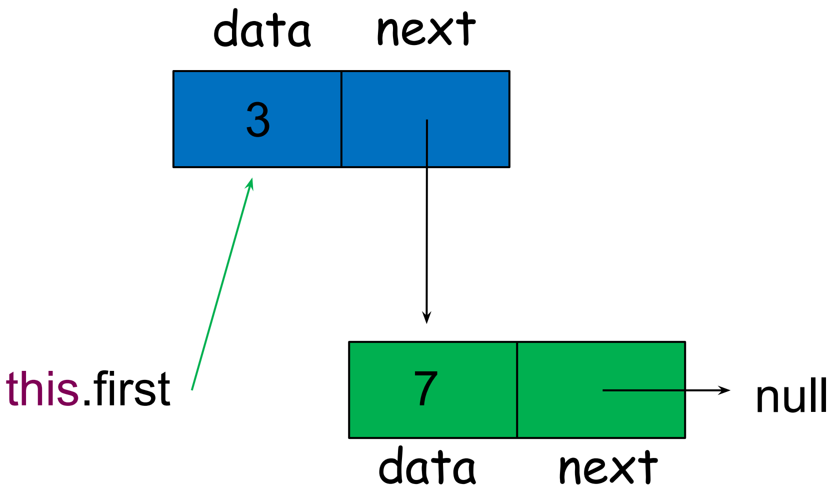
Question 2 – Solution visualization (Continued)

```
public void addFirst(int elem) {
    Node newNode = new Node(elem);
    newNode.next = this.first;
    this.first = newNode;
    this.size++;
}
```

Client code

```
LinkedList q = new LinkedList();
q.addFirst(7);
q.addFirst(3);
// Creates the list (3 7)
```

line 3 client,
line 3 function

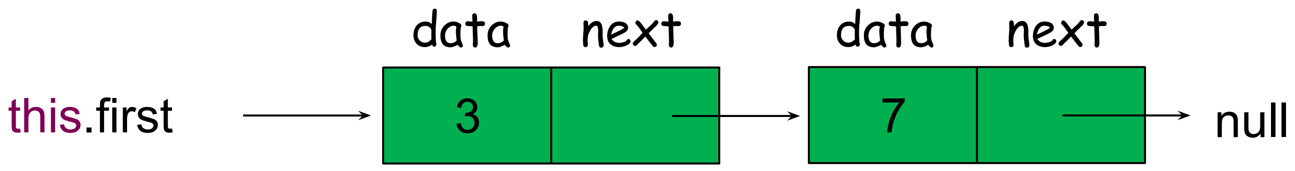


Question 2 – Solution visualization (Continued)

```
public void addFirst(int elem) {
    Node newNode = new Node(elem);
    newNode.next = this.first;
    this.first = newNode;
    this.size++;
}
```

Client code

```
LinkedList q = new LinkedList();
q.addFirst(7);
q.addFirst(3);
// Creates the list (3 7)
```



Question 2, Expansion 1 - AddLast

```
public void addLast(int elem) {
     Node cur = this.first;
     while (cur.next != null) {
          cur = cur.next;
     }
          cur.next = new Node(elem);
          this.size++;
 }
```

What is the time complexity of this functions in terms of the list size n?

Answer: O(n) since we go through the entire array, each time doing a constant number of operations

Question 3 – Remove at Index

- Design a program that implements the 'removeAtIndex (int index)' method. This method removes the element in the index.
- If the index is an illegal index (negative or bigger/equal than the size) return false, else return true to denote successful node removal.

Question 3 – Solution (two pointers)

```
public boolean removeAtIndex(int index) {
        if (index < 0 || index >= this.size || this.first == null) {
            // Index out of bounds or list is empty
            return false;
        }

        if (index == 0) {
        this.first = this.first.next; // Remove first
        this.size--;
        return true;
        }
        // Traverse the list to find the node at the given index
        Node prev = null;
        Node current = this.first;
        for (int i = 0; i < index; i++) {
        prev = current;
            current = current.next;
        }


        // Remove the node at the given index

        prev.next = current.next;
        this.size--; // Decrease the size of the list
        return true;
}
```

Question 3 – Solution (single pointer)

```
public boolean removeAtIndex(int index) {
        if (index < 0 || index >= this.size || this.first == null) {
            // Index out of bounds or list is empty
            return false;
        }

        if (index == 0) {
        this.first = this.first.next; // Remove first
        this.size--;
        return true;
        }
        // Traverse the list to find the node at the given index
        Node current = this.first;
        for (int i = 0; i < index - 1; i++) {
            current = current.next;
        }

        // Remove the node at the given index

        prev.next = current.next;
        this.size--; // Decrease the size of the list
        return true;
}
```

Question 4 - Contains

- The function *contains(int elem)* returns true if this list contains the specified element.
- How can we make this recursive?
- What extra parameters will the recursion need?

Question 4 – Solution

```
public boolean contains (int e) {
    return this.contains(e, this.first);
}

private boolean contains (int e, Node current) {
    if (current == null){
         return false;
    }
    if (current.data == e){
         return true;
    }
    return this.contains(e, current.next);
}
```

Question 4, Expansion 1 - IndexOf

- The function *indexOf(int e)* returns the index of the first appearance the given element appears in.
- If the element doesn't appear in the list, returns -1.
- Again, how can we make this recursive?

Question 4, Expansion 1 - Solution

```java
public int indexOf (int e){
    if (!this.contains(e)) {
        return -1;
    }
    return this.indexOf(e, this.first);
}


private int indexOf (int e, Node current){
    if (current.data == e){
        return 0;
    }
    return 1 + this.indexOf(e, current.next);
}
```

Question 5 - Get

- The function *get(int index)* returns the element at the given index.
- The function throws an exception for illegal indices.

Question 5 – Solution

```
public int get(int index){
    if (index >= this.size || index < 0){
     throw new IndexOutOfBoundsException("Illegal index " + index);
    }
    Node current = this.first;
    for (int i = 0; i < index; i++) {
         current = current.next;
    }
    return current.data;
}
```

Question 6 - ToArray

- The function *toArray()* returns the elements of the list in an array. In the same order the elements appear in the list.

Question 6 - Solution

```java
public int[] toArray() {
    int[] arr = new int[this.size];
    Node cur = this.first;
    for (int i = 0; i < arr.length; i++) {
        arr[i] = cur.data;
        cur = cur.next;
    }
    return arr;
}
```

Question 7 - Add

- The function *add(int index, int e)* adds a given element at a given index.
- Note that adding an element requires direct access to the nodes.

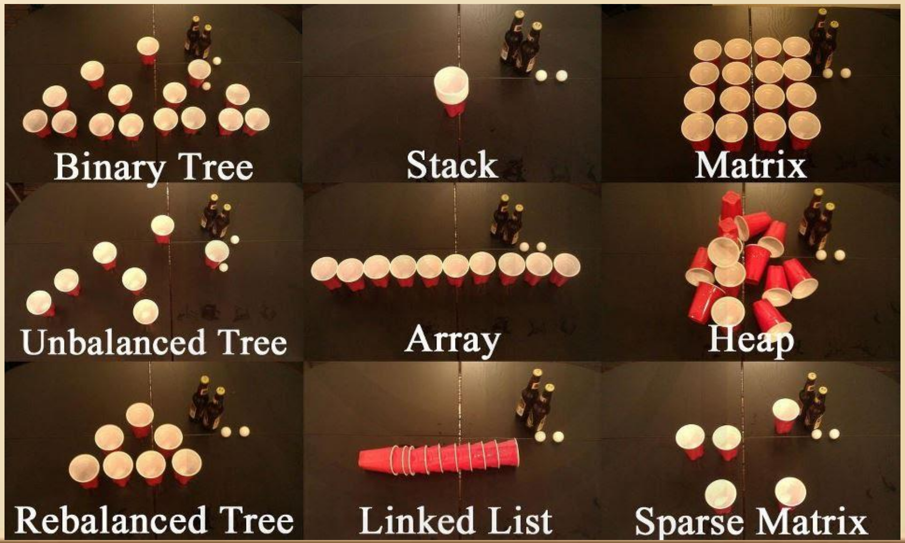
Question 7 - Solution

```
public void add(int index, int e) {
    if (index < 0 || index > this.size) {
     throw new IndexOutOfBoundsException("illegal index " + index);
    }

    if (index == 0) {
     this.addFirst(e); // adds to first
     return;
    }

    if (index == this.size) {
     this.addLast(e); // adds to last
     return;
    }

    Node insert = new Node(e);
    Node prev = null;
    Node current = this.first;
    for (int i = 0; i < index; i++) {
     prev = current;
     current = current.next;
    }
    prev.next = insert;
    insert.next = current;
    this.size++;
 }
```

Introduction to Computer Science
Reichman University

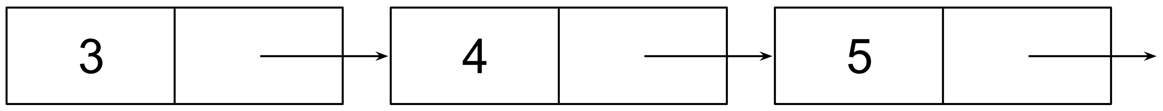
Recitation 10

Linked Lists – List Operations

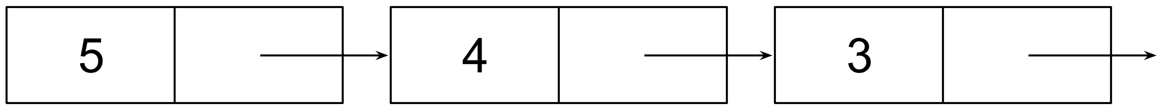


Question 8 - Reverse list

- Design a program which reverses the order of the elements in the list. Without changing the original list.
- Note: You may assume that you are in LinkedList class
- example :
- Before:



- After:



Question 8 – Solution

```
public LinkedList reverse() {
       Node cur = this.first;
       LinkedList rev = new LinkedList();
       while (cur != null){
      rev.addFirst(cur.data);
           cur = cur.next;
       }
       return rev;
   }
```

Question 9 - Linked List - SubList

- In the next question we will build a variation of the function str.substring(int start, int end) but instead of working on a String we will work on a list.
- Given a Linked List and 2 indices (i,j) and i<=j we will return the sublist which starts at the i'th index and ends the j'th index (not included)

Question 9 – Solution

```
public LinkedList subList(int start, int end) {
       if (start < 0 || end >= this.size || start > end) {
           throw new InvalidParameterException();
       }
       Node cur = this.first;
       LinkedList sub = new LinkedList();
       for (int i = 0; i < this.size; i++) {
           if (start <= i && i < end) {
               sub.addLast(cur.data);
           }
           cur = cur.next;
       }
       return sub;
}
```

- How many steps did this take.
  - assuming the only two fields of LinkedList are first and size?

Question 9 – Solution

```
public LinkedList subList(int start, int end) {
       if (start < 0 || end >= this.size || start > end) {
           throw new InvalidParameterException();
       }
       Node cur = this.first;
       LinkedList sub = new LinkedList();
       for (int i = 0; i < this.size; i++) {
           if (start <= i && i < end) {
               sub.addLast(cur.data);
           }
           cur = cur.next;
       }
       return sub;
}
```

- How many steps did this take. O(n^2)
  - every addLast method takes O(n). done n times therefore
  - n * O(n) = O(n^2)

Question 9 – Solution

```
public LinkedList subList(int start, int end) {
       if (start < 0 || end >= this.size || start > end) {
           throw new InvalidParameterException();
       }
       Node cur = this.first;
       LinkedList sub = new LinkedList();
       for (int i = 0; i < this.size; i++) {
           if (start <= i && i < end) {
               sub.addLast(cur.data);
           }
           cur = cur.next;
       }
       return sub;
}
```

- Can we improve that? How?
- Yes, there 2 options:
  - Shortening the complexity of addLast by adding and maintaining a last field.
  - Using different functions which cost less!

Question 9, Expansion 1 – Solution

```
public LinkedList subList(int start, int end) {
       if (start < 0 || end > this.size || start > end) {
           throw new InvalidParameterException();
       }
       Node cur = this.first;
       LinkedList sub = new LinkedList();
       for (int i = 0; i < this.size; i++) {
           if (start <= i && i < end) {
               sub.addFirst(cur.data);
           }
           cur = cur.next;
       }
       return sub.reverse();
}
```

```
public LinkedList reverse() {
       Node cur = this.first;
       LinkedList rev = new LinkedList();
       while (cur != null){
       rev.addFirst(cur.data);
               cur = cur.next;
       }
       return rev;
}
```

Question 9, Expansion 1 – Solution

```
public LinkedList subList(int start, int end) {
        if (start < 0 || end > this.size || start > end) {
            throw new InvalidParameterException();
        }
        Node cur = this.first;
        LinkedList sub = new LinkedList();
        for (int i = 0; i < this.size; i++) {
            if (start <= i && i < end) {
                sub.addFirst(cur.data);
            }
            cur = cur.next;
        }
        return sub.reverse();
}
```

```
public LinkedList reverse() {
        Node cur = this.first;
        LinkedList rev = new LinkedList();
        while (cur != null){
        rev.addFirst(cur.data);
            cur = cur.next;
        }
        return rev;
}
```

- Constructor call + field call + boolean checks (first 4 lines) -> O(1)
- addFirst -> O(1)
  - But we use it for every element in the list (in the worst case scenario), the operation was done n times therefore-> O(1) * n = O(n)
- reverse is done in the end and not for every element. If we look in the implementation we did there and we also used addFirst for every element of the list thefore -> O(1) * n = O(n)
- Total: O(1) + O(n) + O(n) = O(n)

we may have done more ops but each one was 'cheaper' in running time.

Question 10 - Linked List - Equals

- Given 2 String linked lists, build a function which return true if the elements of the lists are appearing in the same order.