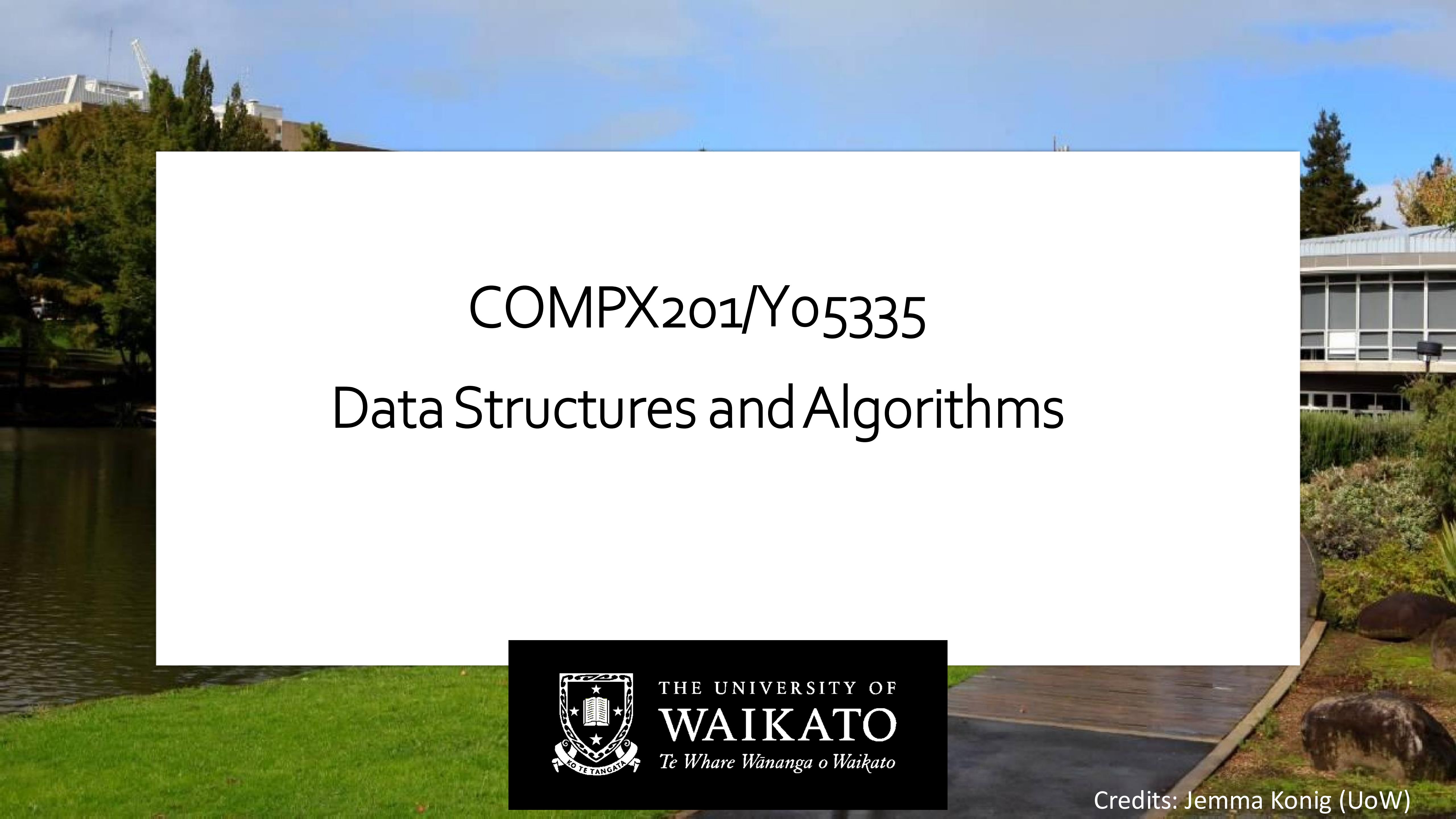


COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)

Linked Lists

COMPX201/Y05335

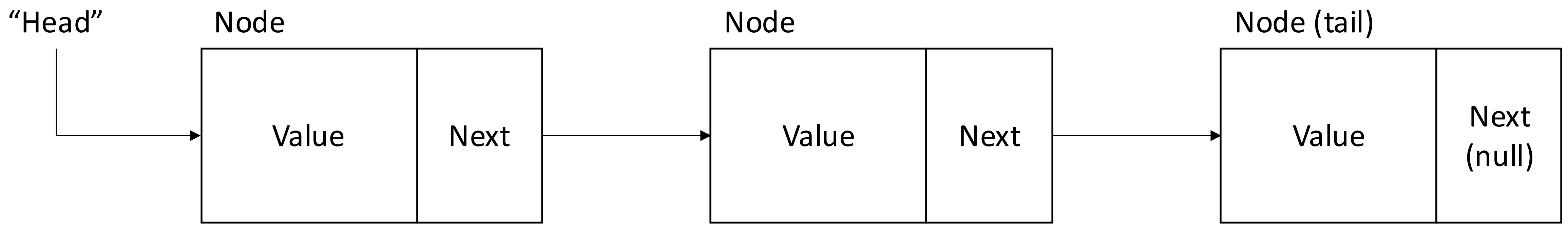
Overview

- What is a linked list?
- Operations
- Implementation
- Types

What is a linked list?

What is a linked list?

“An ordered [or unordered] set of data elements, each containing a link to its successor (and sometimes its predecessor).”



https://www.google.com/search?q=linked+list+definition&rlz=1C5CHFA_enNZ807NZ807&oq=linked+list+definition&aqs=chrome..69i57j35i39l2j69i61l2j69i60l3.2747j0j7&sourceid=chrome&ie=UTF-8

What is a linked list?

- A collection of items.
- Sets can’t have duplicates, lists can.
- Ordered in some way (even if order is random).
- First item, last item, i-th item etc.
- Self-referential data structure.

Linked list operations

Linked list operations

- Add an item.
- Remove an item.
- Has an item (“contains”).
- Is list empty?
- Get first item, last item, i-th item …
- Length
- Insert (if ordered to add in correct place).
- … etc …

Linked list implementations

Linked list implementations

- Array - could be computationally expensive.
- Better to use a custom list data structure …

Linked list implementations

(1) Iterative:

(2) Recursive:

Linked list implementations

(1) Iterative:

- Each item has the value of the item and a reference to the next item in the list.
- Use the pointer references to implement the operations.

(2) Recursive:

Linked list implementations

(1) Iterative:

- Each item has the value of the item and a reference to the next item in the list.
- Use the pointer references to implement the operations.

(2) Recursive:

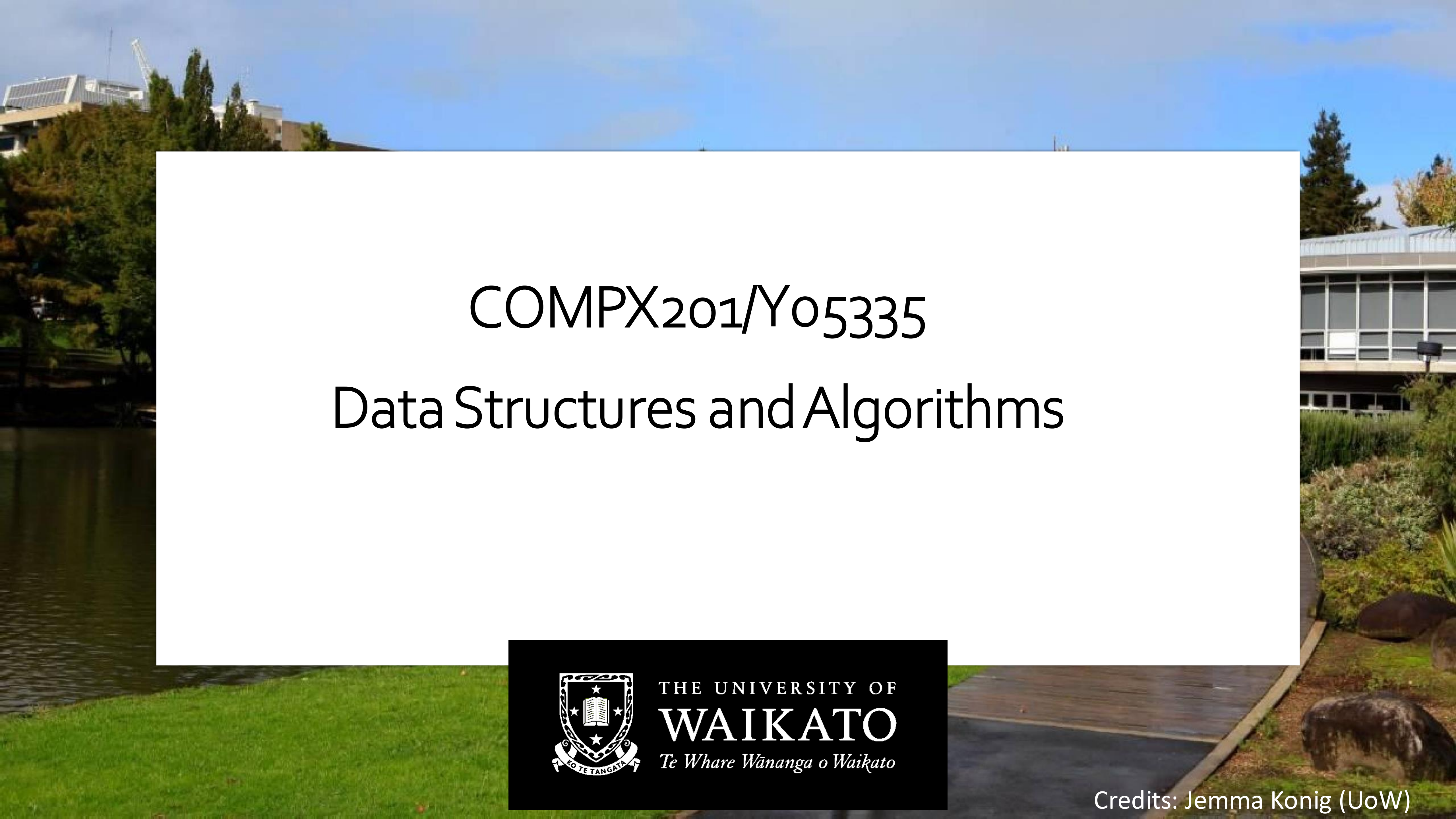
- List is either empty or has a head item and a tail.
- Define operations recursively.
- Less prone to bugs but uses more memory and slower.

Types of linked lists

Types of linked lists

- Access:
  - Restricted: access to items in the list is limited such as the first or last item (e.g. Queue).
  - Unrestricted: the list may be searched and items added or removed at any point.

- Order:
  - Ordered: items are ordered by some criterion, such as key-value, priority, time of insertion (e.g. Most-recently-used (MRU)).
  - Unordered: items are randomly ordered.



COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)

Linked Lists in Java
Part 1

COMPX201/Y05335

Overview

- Defining our linked list
- Operations:
    - add()
    - print()

Defining our linked list

Defining our linked list

We are going to implement an ‘iterative’, ‘unrestricted’, ‘unordered’ linked list

Iterative:

- Each item has the value of the item and a reference to next item in the list.
- Use the pointer references to implement the operations.

Unrestricted:

- The list may be searched, and items may be added or removed at any point.

Unordered:

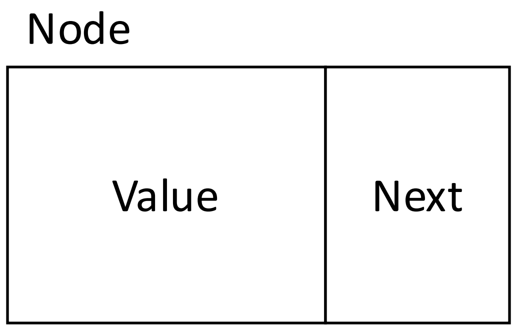
- Items are randomly ordered

Defining our linked list

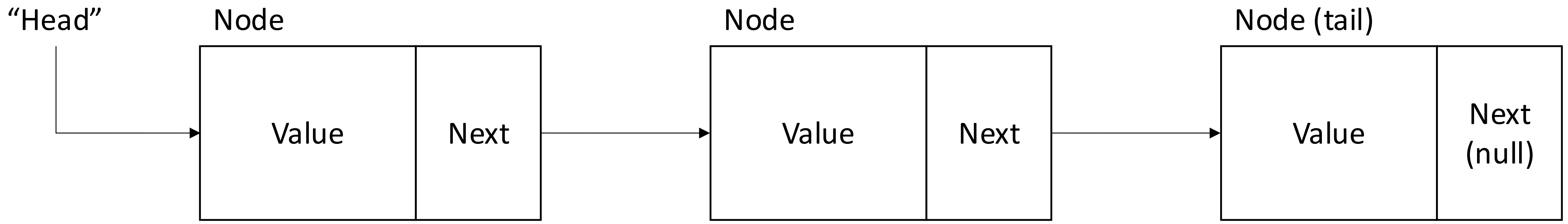
Iterative:

- Each item has the value of the item and a reference to next item in the list.
- Use the pointer references to implement the operations.

i.e., ‘Nodes’



Defining our linked list



Defining our linked list

Okay, let's try it ...

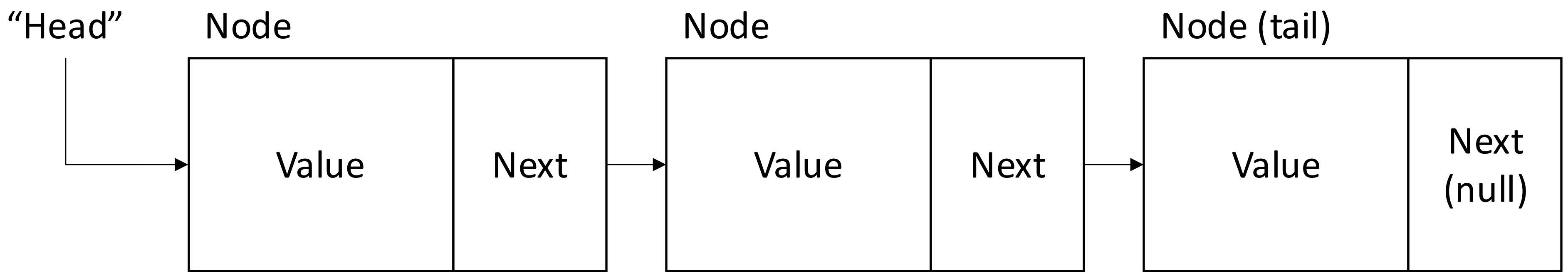
Add

- Add to start
- Add to end
- Insert (for ordered lists)
- Insert at position

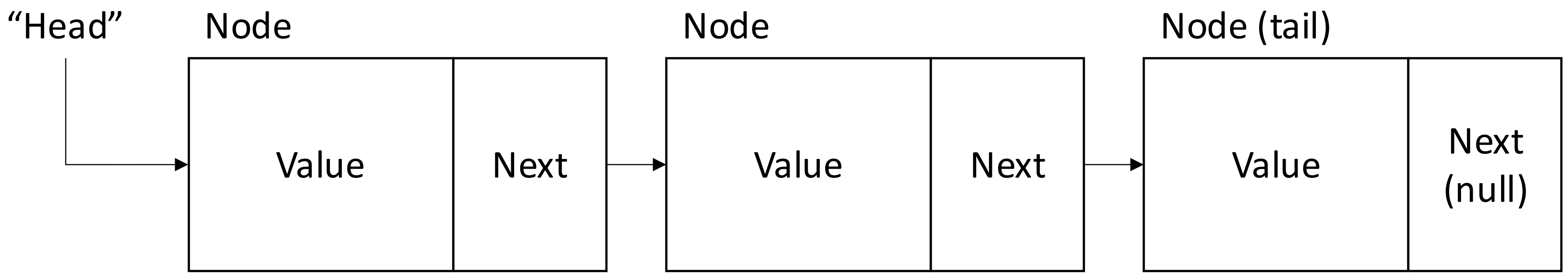
Add

- Add to start
- Add to end
- Insert (for ordered lists)
- Insert at position

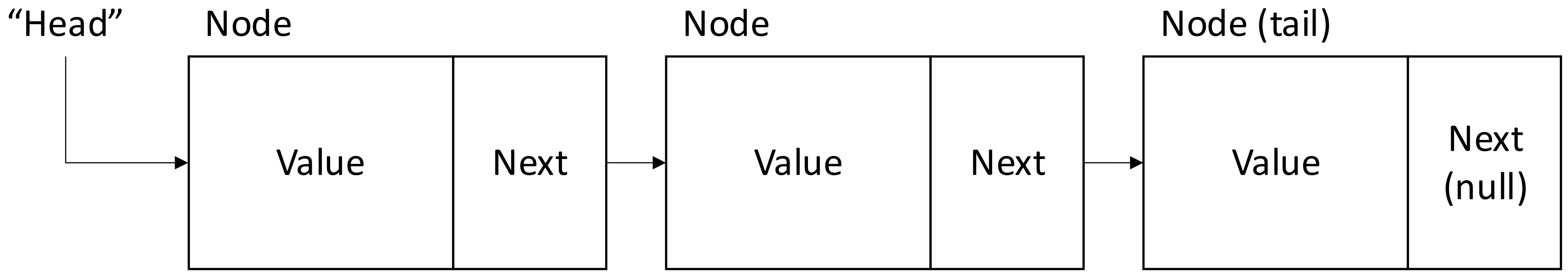
Add (add to start)

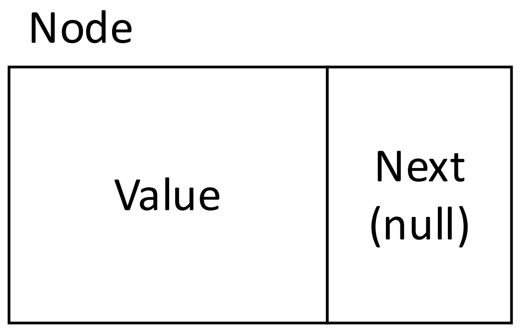


Add (add to start)



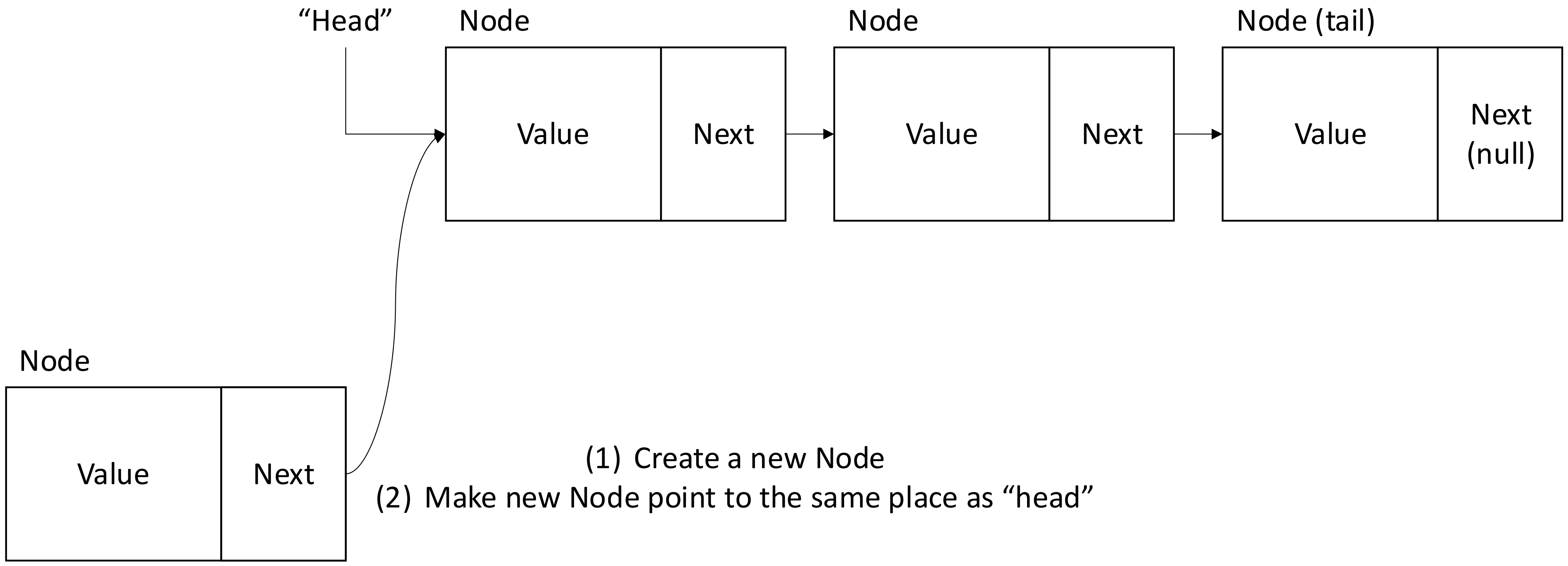
Add (add to start)





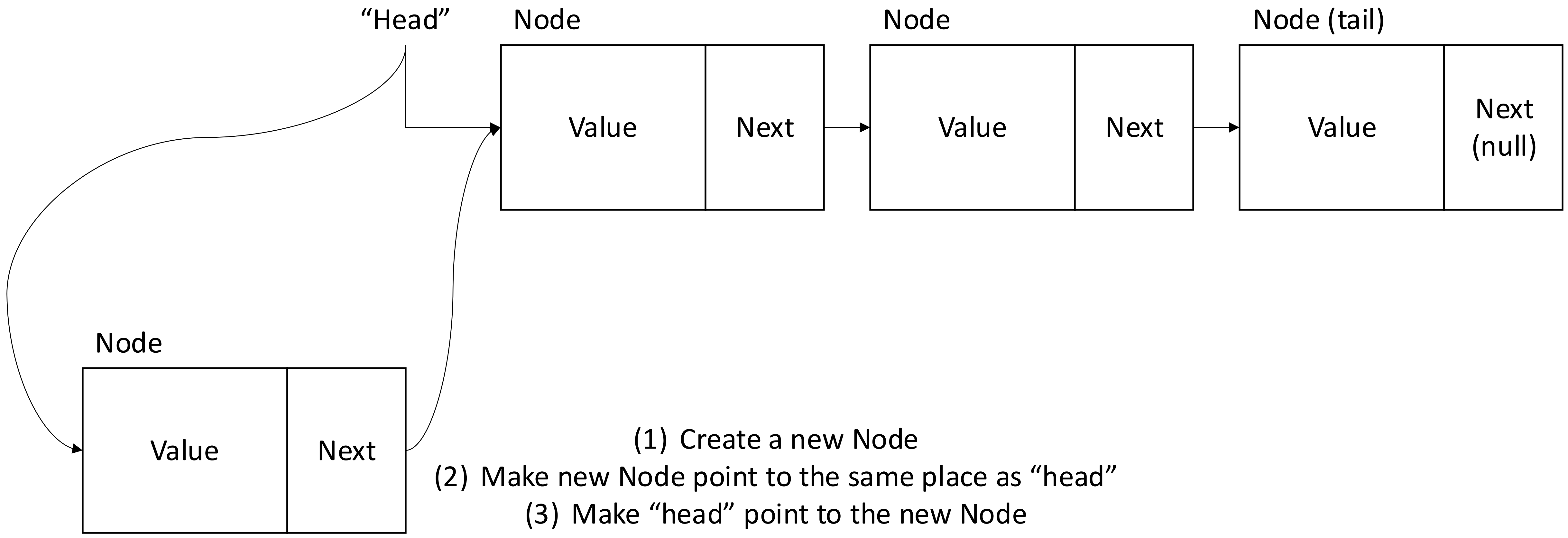
(1) Create a new Node

Add (add to start)



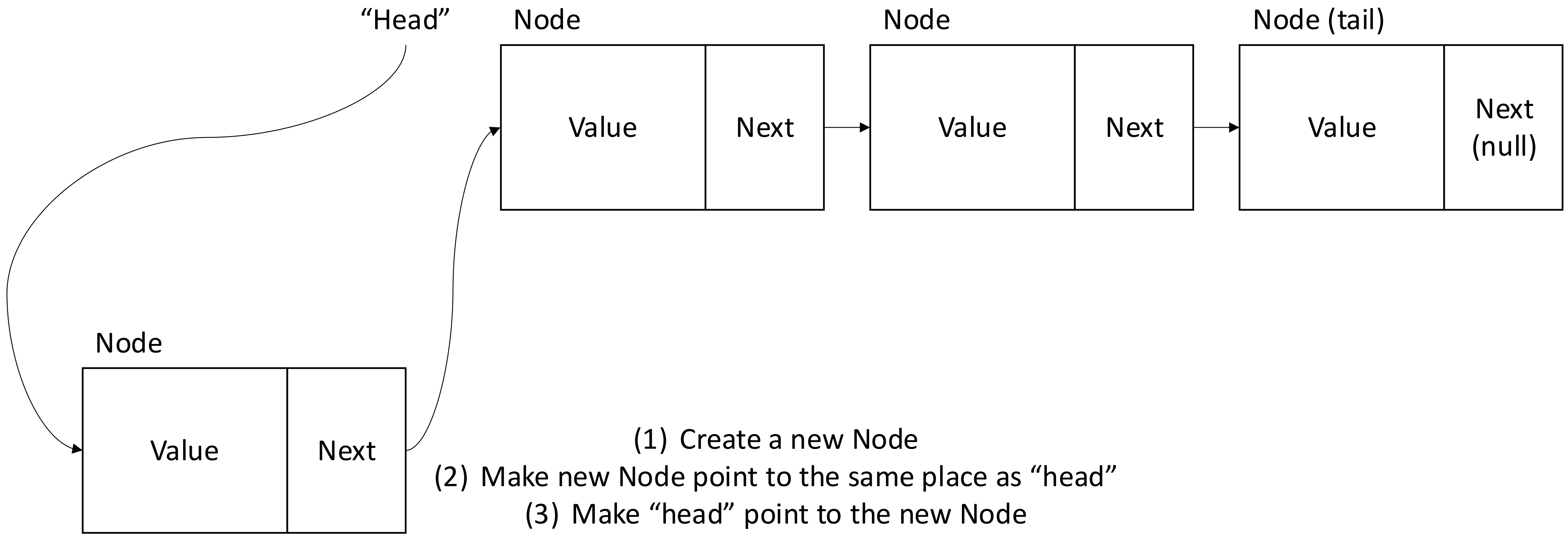
(1) Create a new Node
(2) Make new Node point to the same place as "head"

Add (add to start)



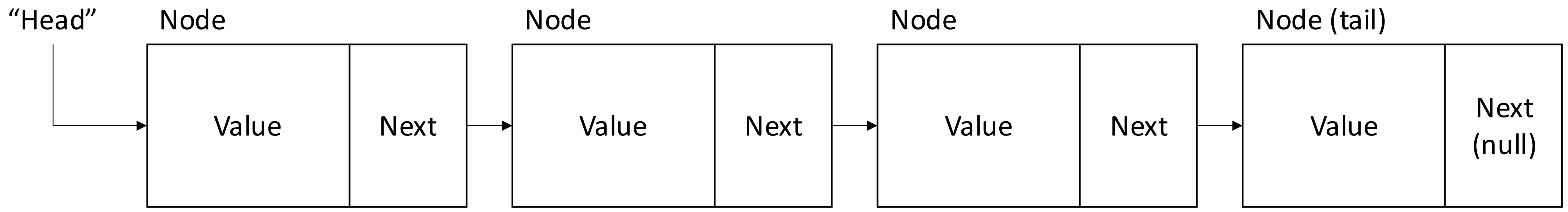
(1) Create a new Node
(2) Make new Node point to the same place as "head"
(3) Make "head" point to the new Node

Add (add to start)



(1) Create a new Node
(2) Make new Node point to the same place as "head"
(3) Make "head" point to the new Node

Add (add to start)



Add (add to start)

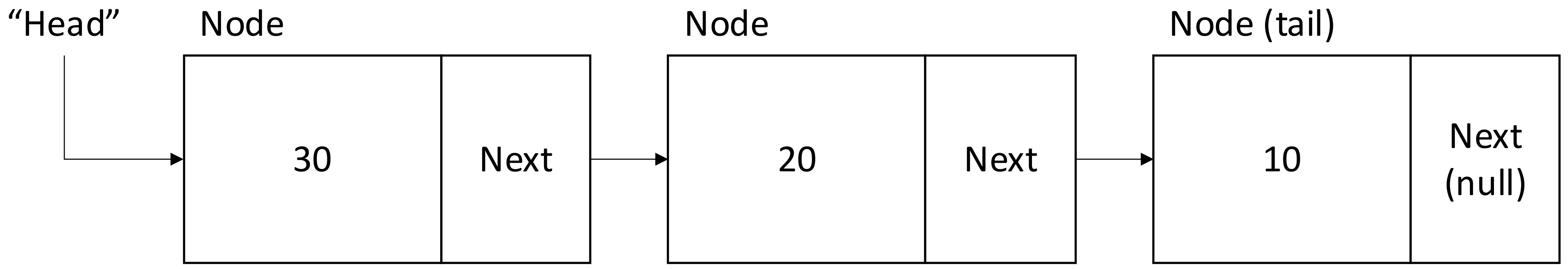
Okay, let's try it ...

Print

```
IntegerLinkedList integerLinkedList = new IntegerLinkedList();
integerLinkedList.add(10);
integerLinkedList.add(20);
integerLinkedList.add(30);
```

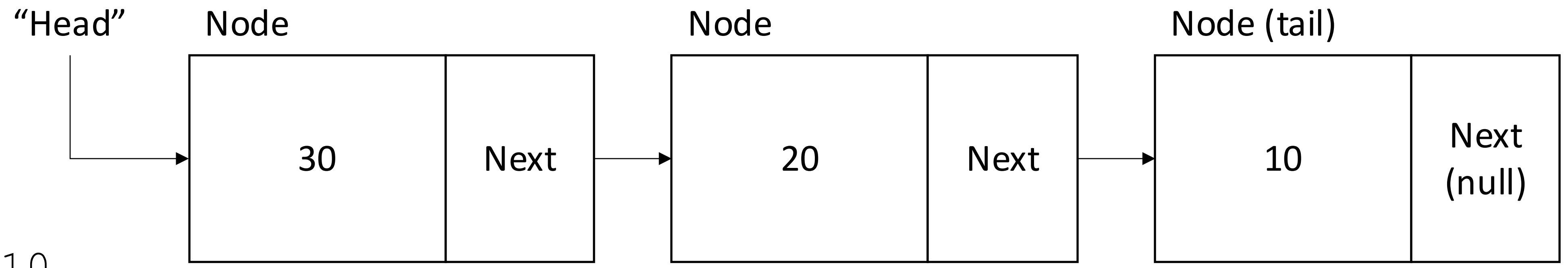
Print

```
IntegerLinkedList integerLinkedList = new IntegerLinkedList();
integerLinkedList.add(10);
integerLinkedList.add(20);
integerLinkedList.add(30);
```



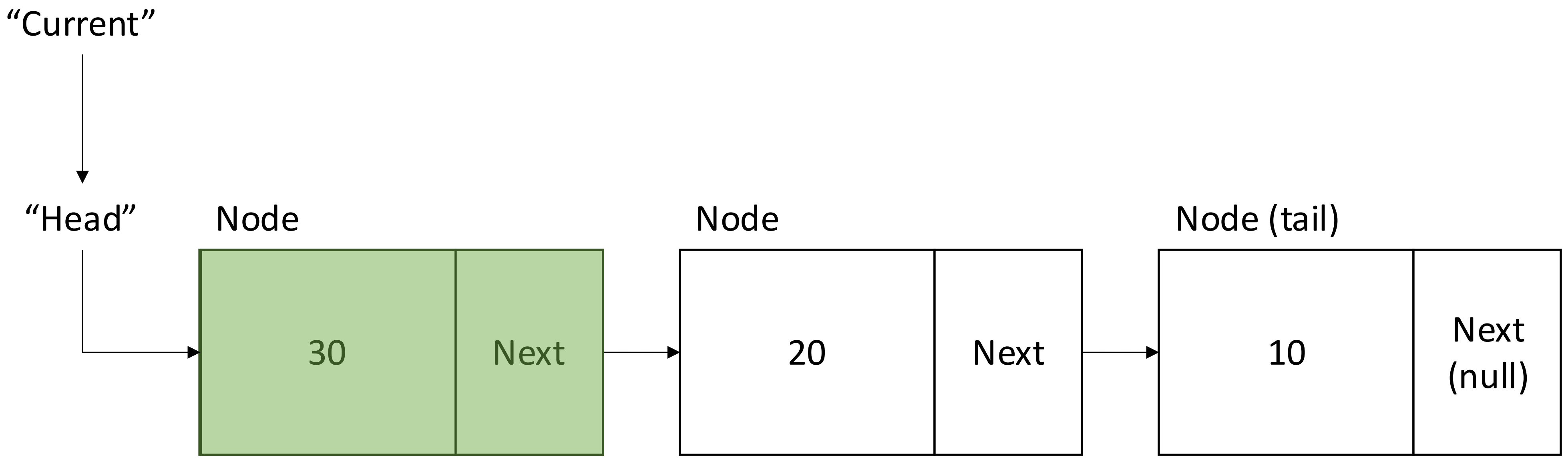
Print

```
IntegerLinkedList integerLinkedList = new IntegerLinkedList();
integerLinkedList.add(10);
integerLinkedList.add(20);
integerLinkedList.add(30);
```



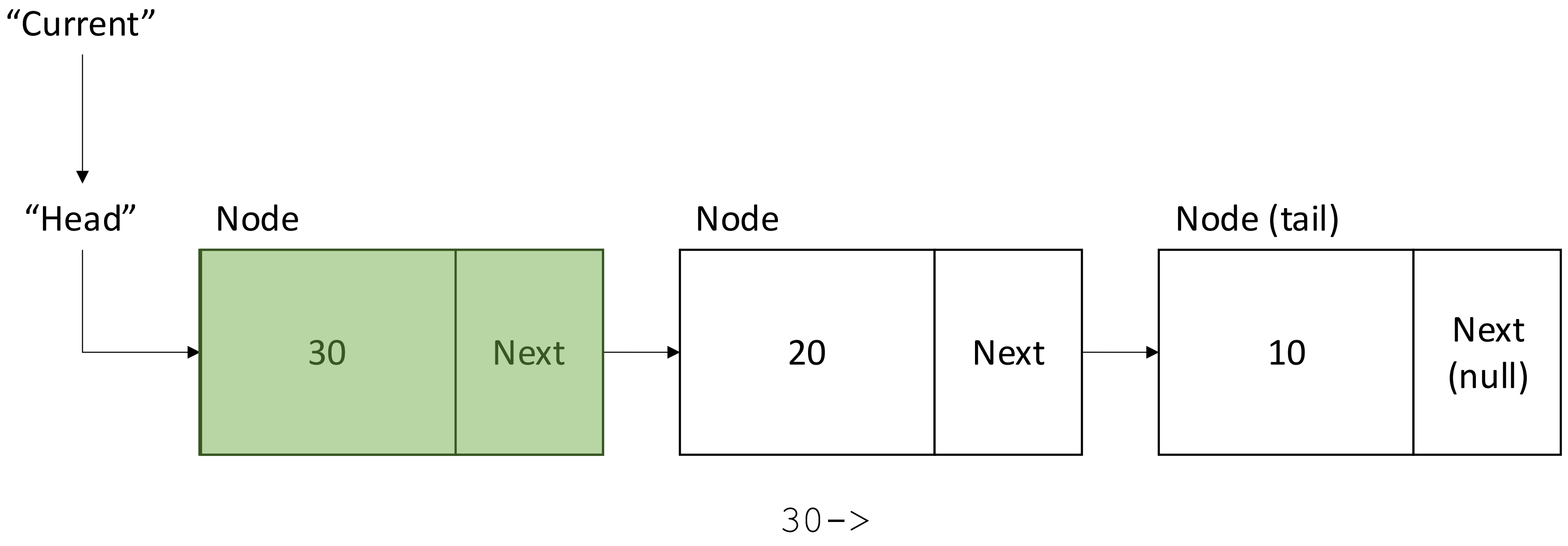
```
10
20->10
30->20->10
```

Print



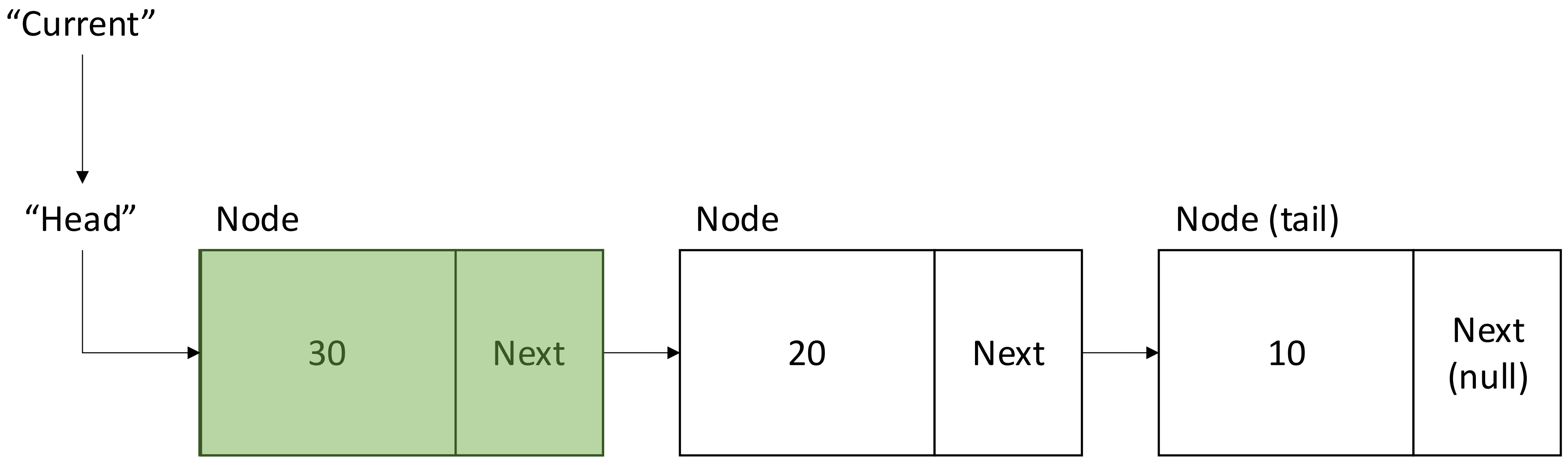
(1) Create a pointer to the 'current' Node

Print



(1) Create a pointer to the 'current' Node
(2) If current isn't null, print its value

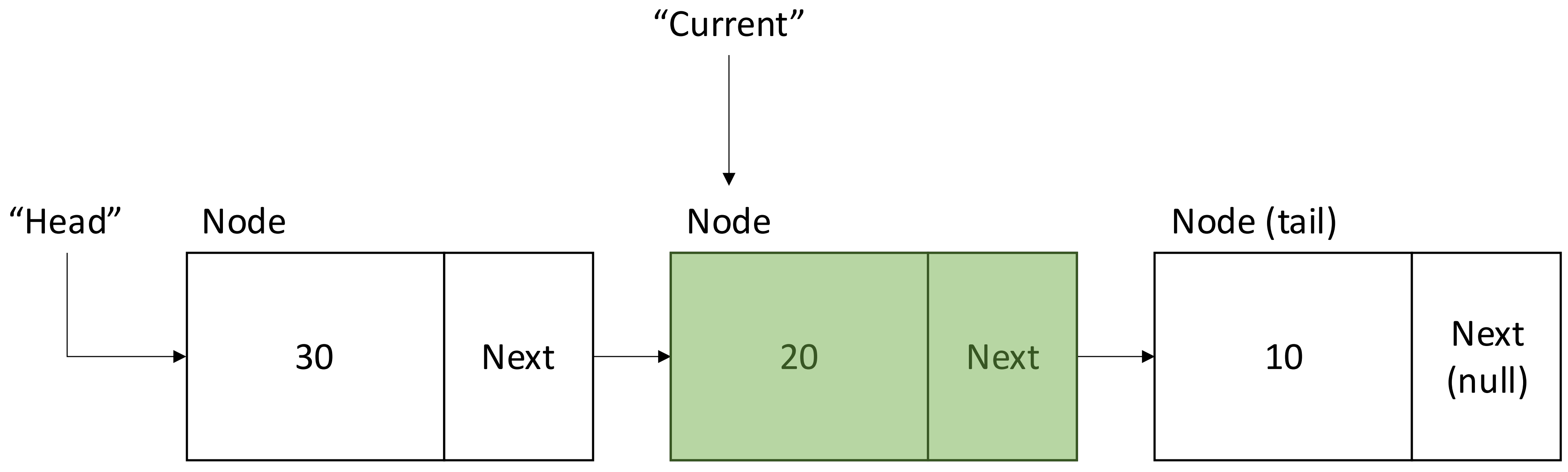
Print



```
30->
```

(1) Create a pointer to the ‘current’ Node
(2) If current isn’t null, print its value
(3) Move current along one

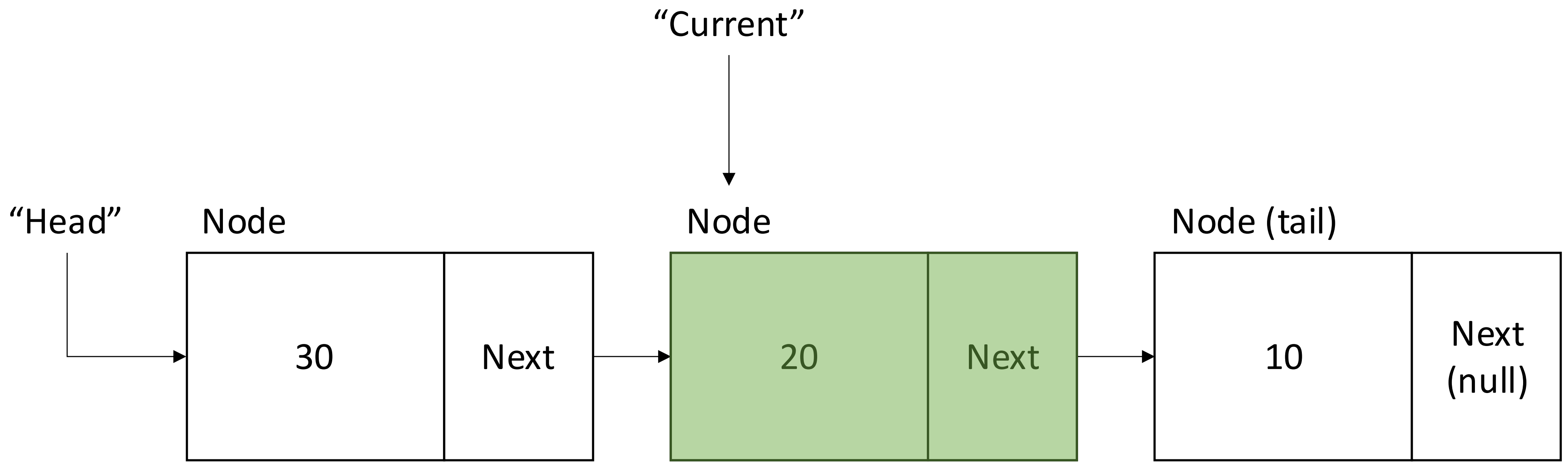
Print



```
30->
```

(1) Create a pointer to the 'current' Node
(2) If current isn't null, print its value
(3) Move current along one

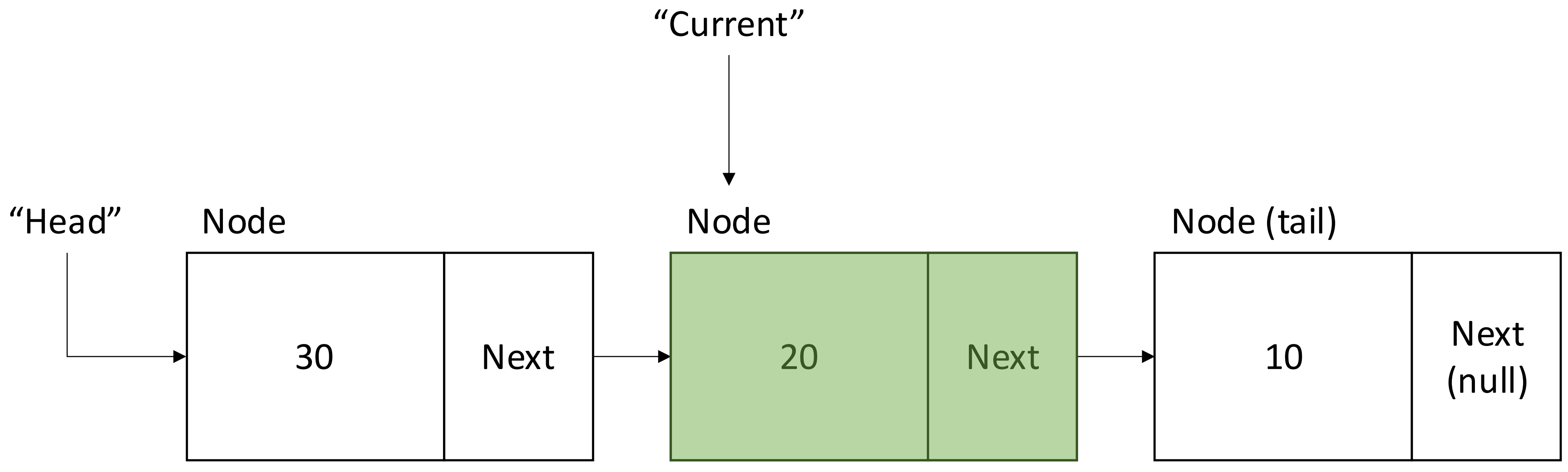
Print



```
30->
```

(1) Create a pointer to the 'current' Node
(2) If current isn't null, print its value
(3) Move current along one

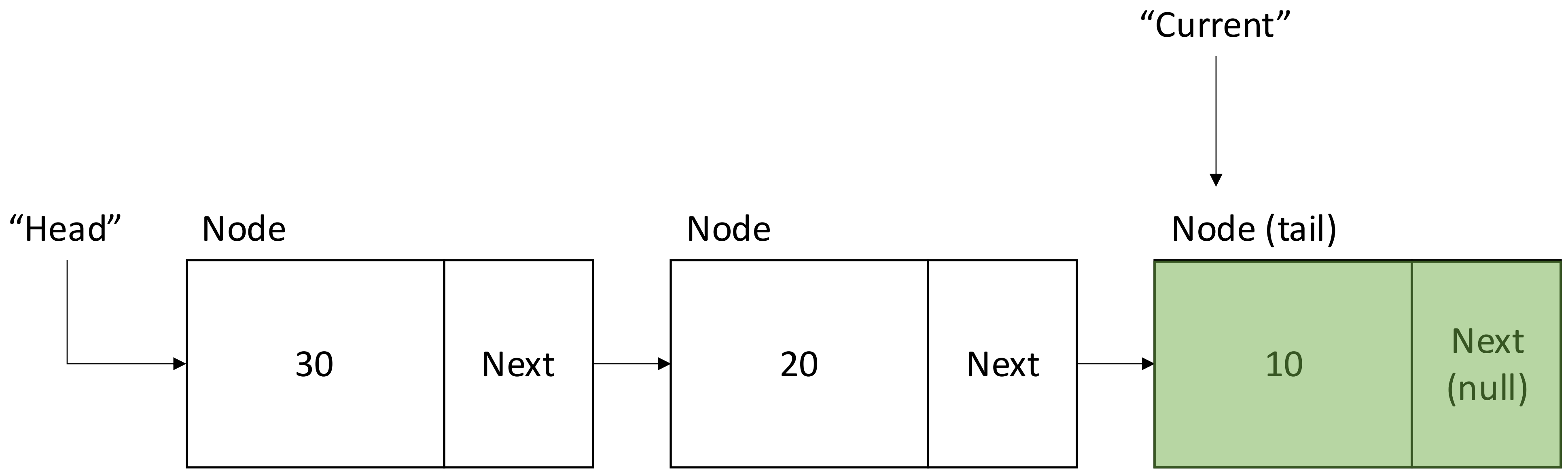
Print



```
30->20->
```

(1) Create a pointer to the 'current' Node
(2) If current isn't null, print its value
(3) Move current along one
(4) Repeat Step 2 and 3. Stop when current is null

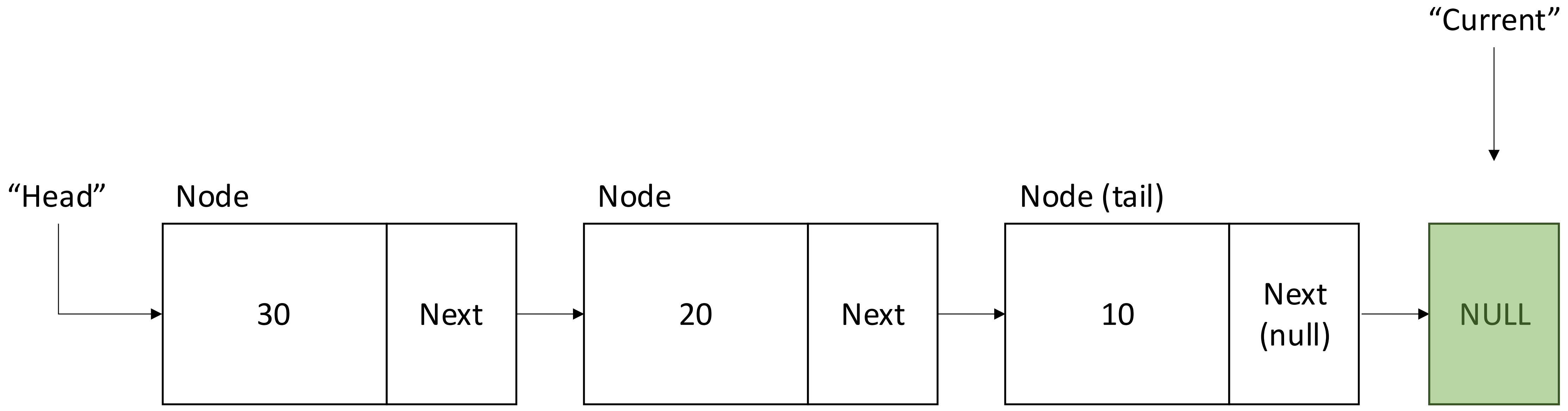
Print



```
30->20->10->
```

1. Create a pointer to the 'current' Node
2. If current isn't null, print its value
3. Move current along one
4. Repeat Step 2 and 3. Stop when current is null

Print



```
30->20->10->
```

1. Create a pointer to the 'current' Node
2. If current isn't null, print its value
3. Move current along one
4. Repeat Step 2 and 3. Stop when current is null

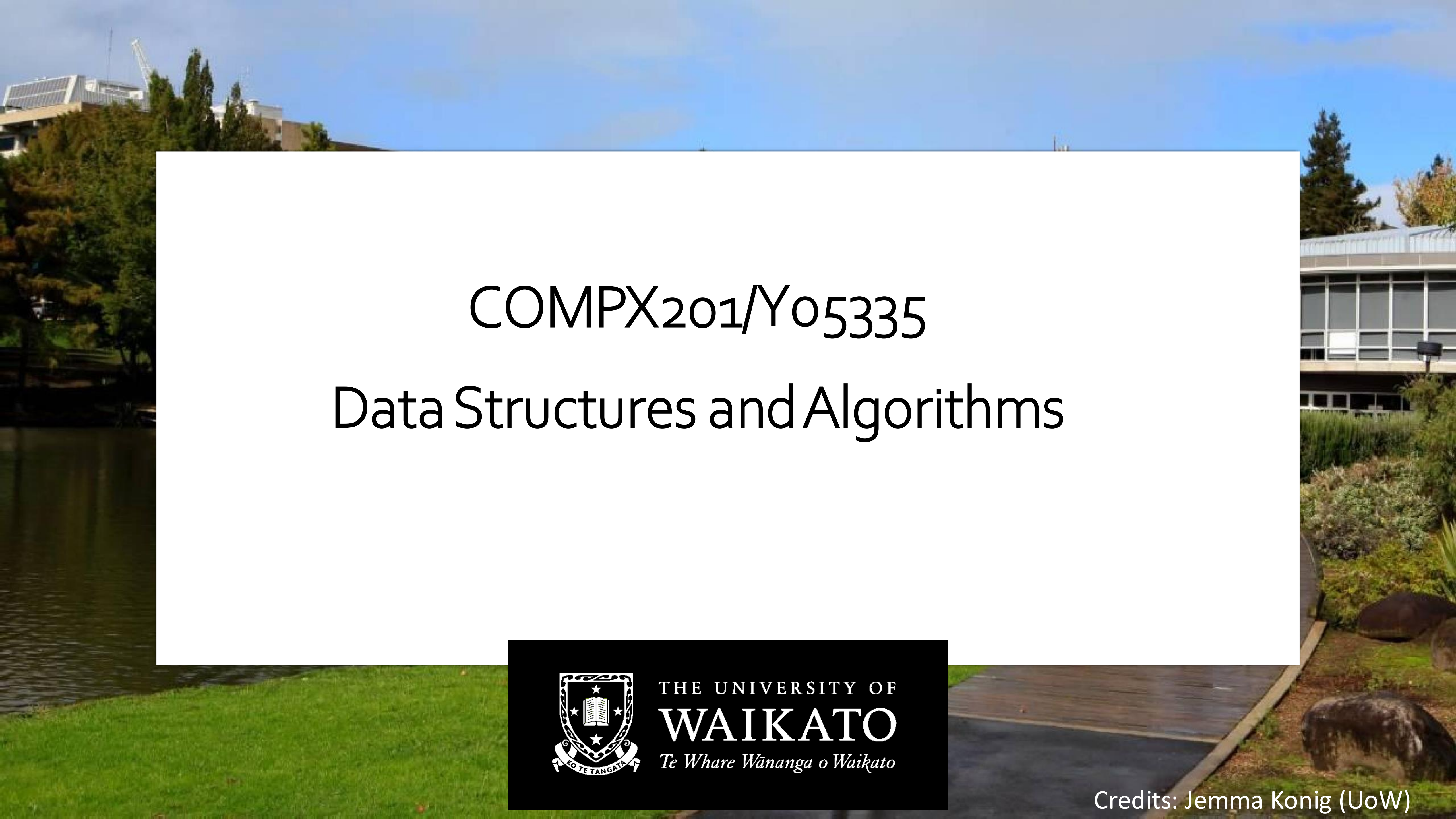
Print

(1) Create a pointer to the 'current' Node
(2) If current isn't null, print its value
(3) Move current along one
(4) Repeat Step 2 and 3. Stop when current is null

```
CREATE a pointer to the 'current' Node
WHILE current isn't null
    PRINT value
    MOVE current along one
END WHILE
```

Print

Okay, let's try it ...



COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)

Linked Lists in Java
Part 2

COMPX201/Y05335

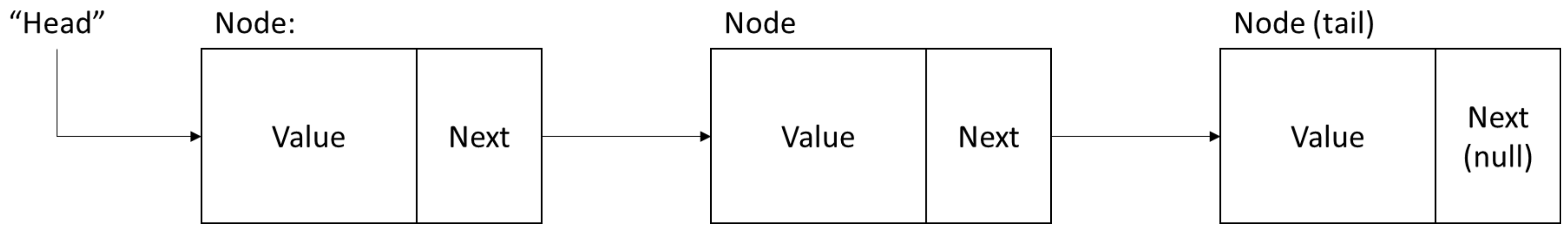
Overview

- Operations:
  - isEmpty
  - hasValue
  - getLength
  - remove
  - insert

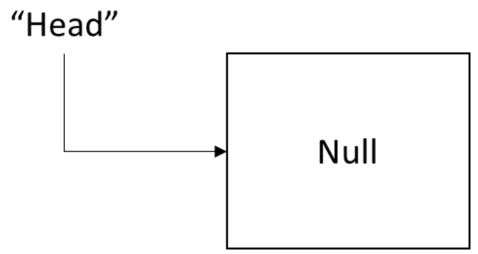
isEmpty

How do we tell if our Linked List is empty?

A non-empty list will look something like this …



An empty list will look like this …



isEmpty

Okay, let's try it ...

hasValue

How do we tell if our Linked List contains a specific value?

Iterate through each node in the list (like we did with print)

```
WHILE not at end of list (node is not null),
    IF the node has value X
        RETURN true
    END IF
    MOVE to next node
END WHILE
RETURN false
```

hasValue

Okay, lets try it …

getLength

How do we get the length of our Linked List?

Note, this is our own custom data structure, so there are no built-in methods that we can use. We must build our own method.

Iterate over our list, counting the nodes as we go

```
WHILE not at end of list (node is not null),
        INCREMENT counter
        MOVE to next node
END WHILE
```

getLength

Okay, lets try it …

Implementing a Linked List in Java – Node.java

```
public class Node{
    int value;
    Node next;

    public Node(int x){
        value = x;
        next = null;
    }
}
```

Implementing a Linked List in Java – IntLinkedList.java

```
public class IntLinkedList{
    Node head;

    public IntLinkedList(){
        head = null;
    }

    public void addNode(int n){
        Node newNode = new Node(n);

        // If the list is empty
        if(head == null){
            head = newNode;
        } else{
            newNode.next = head;
            head = newNode;
        }
    }

    public boolean isEmpty(){
        if(head == null){
            return true;
        }
        return false;
    }
```

Implementing a Linked List in Java – IntLinkedList.java

```
    public boolean hasNode(int x){
        Node current = head;

        while(current != null){
            if(current.value == x){
                return true;
            }
            current = current.next;
        }
        return false;
    }

    public void print(){
        Node current = head;

        while(current != null){
            System.out.print(current.value + "->");
            current = current.next;
        }
        System.out.println();
    }

    public int getLength(){
        Node current = head;
        int length = 0;

        while(current != null){
            length++;
            current = current.next;
        }
        return length;
    }
}
```

Implementing a Linked List in Java – Main.java

```
public class Main{
    public static void main(String [] args){
        // System.out.println("Testing...");
        IntLinkedList intLinkedList = new IntLinkedList();

        System.out.println("Is the list empty?: " + intLinkedList.isEmpty());
        System.out.println("Printing an empty list...");
        intLinkedList.print();
        System.out.println();
        System.out.println("The list contains " + intLinkedList.getLength() + " nodes");

        intLinkedList.addNode(10);
        intLinkedList.addNode(20);
        intLinkedList.addNode(30);

        System.out.println("Is the list empty?: " + intLinkedList.isEmpty());
        System.out.println("Printing a non-empty list...");
        intLinkedList.print();
        System.out.println("The list contains " + intLinkedList.getLength() + " nodes");
    }
}
```

Remove

- Remove from start
- Remove from end
- Remove by value
- Remove at position

Remove

- Remove from start
- Remove from end
- Remove by value
- Remove at position

```
IntegerLinkedList integerLinkedList = new IntegerLinkedList();
integerLinkedList.add(10);
integerLinkedList.add(20);
integerLinkedList.add(30);

integerLinkedList.remove(20);
```

Remove(x)

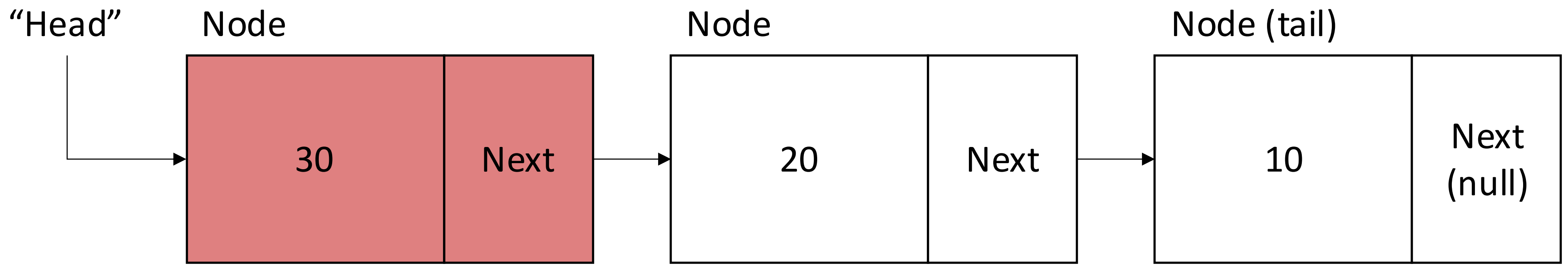
Conditions:

1. List must not be empty
2. Value must exist in list

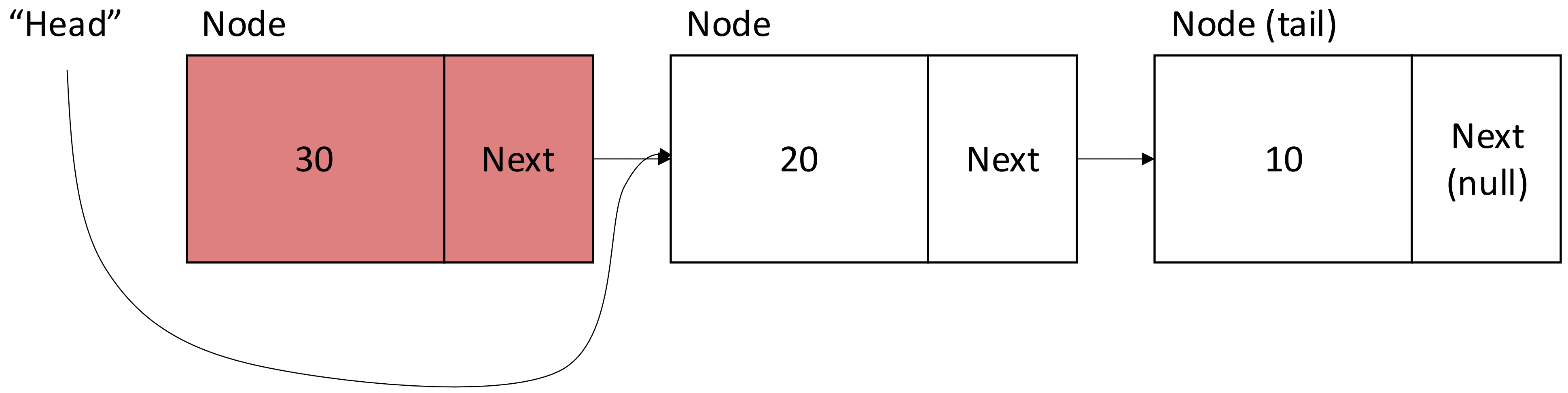
Two cases:

1. Remove the head
2. Remove any other node (including the tail)

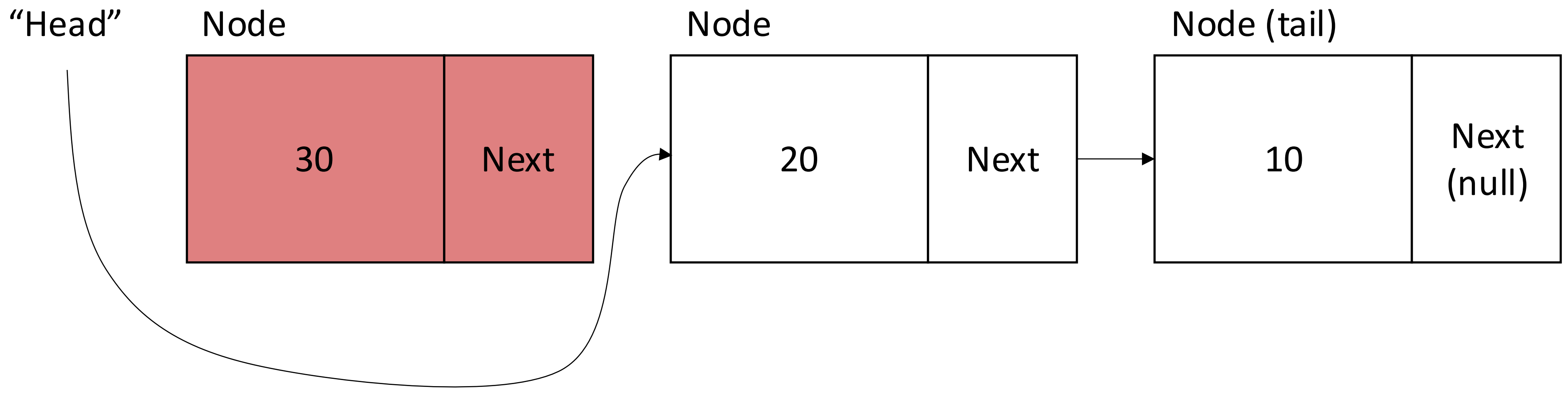
Remove (x) from head



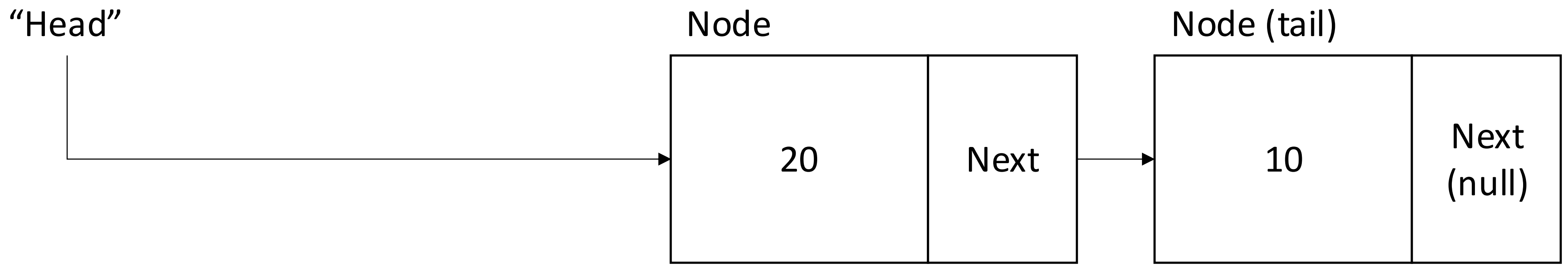
Remove (x) from head



Remove (x) from head



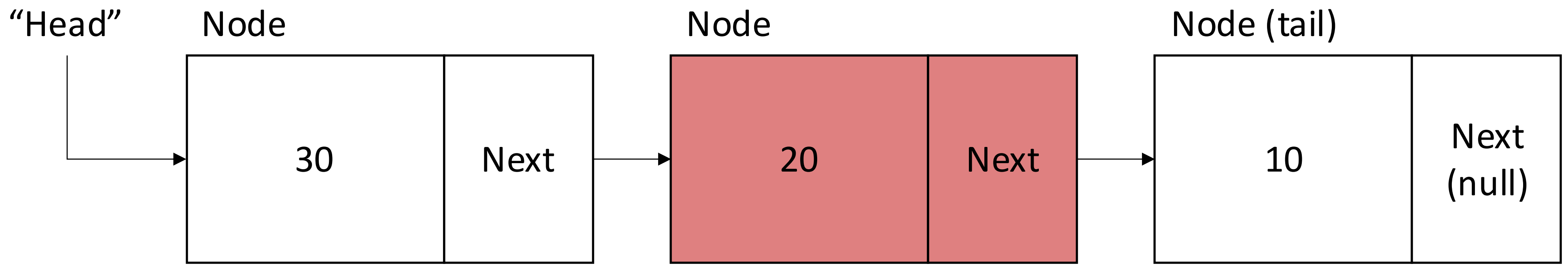
Remove (x) from head



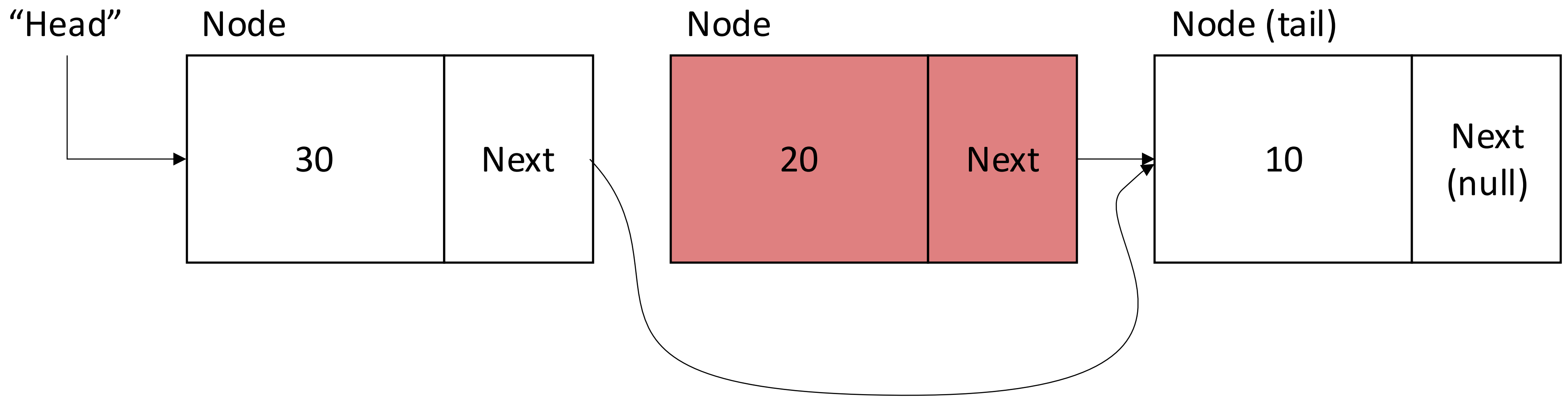
Remove (x) from head

```
IF head has value X
        SET head to be head.next
END IF
```

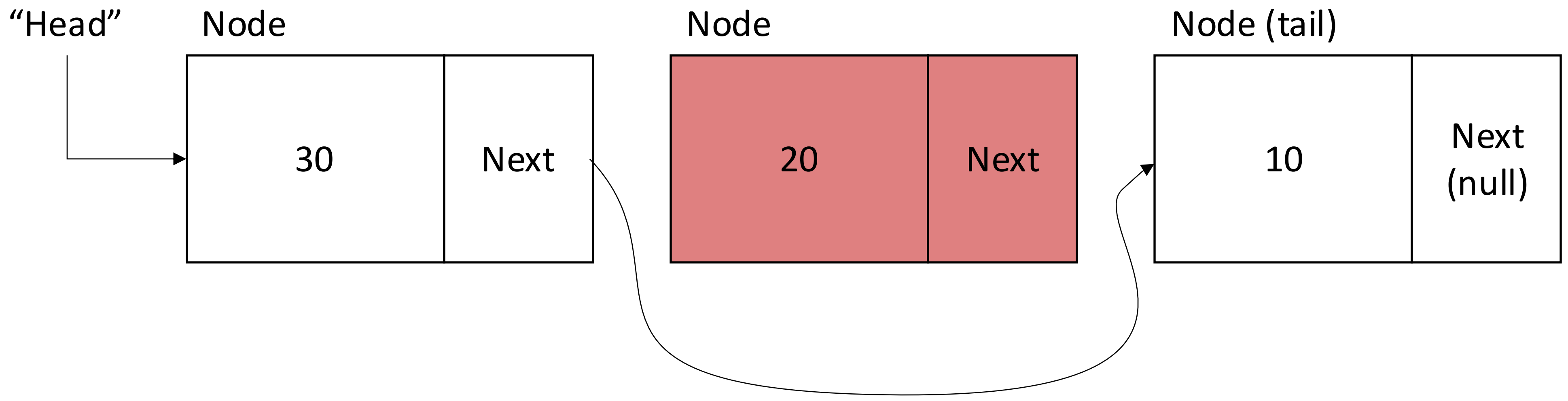
Remove (x) from any other nodes



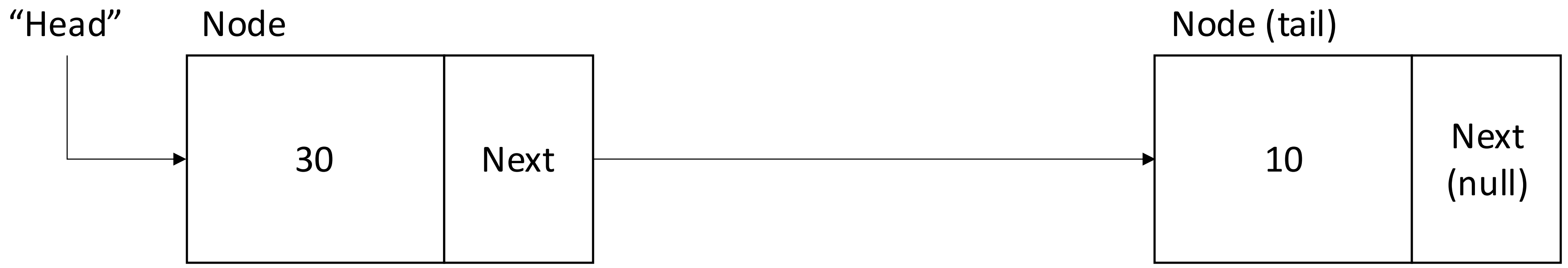
Remove (x) from any other nodes



Remove (x) from any other nodes



Remove (x) from any other nodes



Remove (x) from head

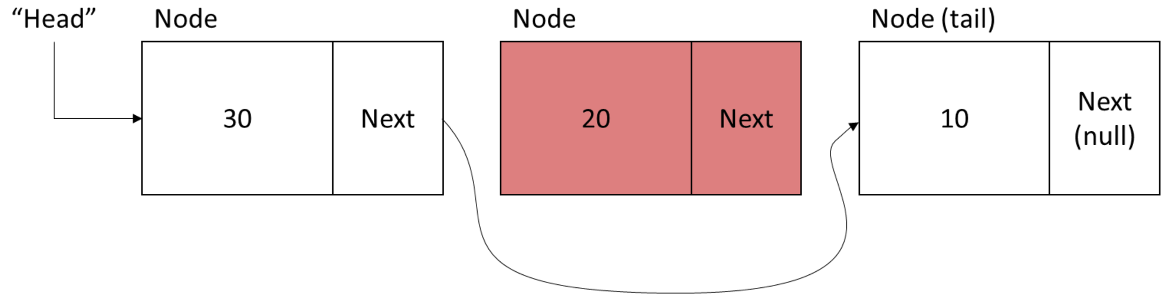
```
IF current node has value X
        SET previous.next to current.next
END IF
```

Remove(x)

Note, we need to know

a) the previous node

b) the current node

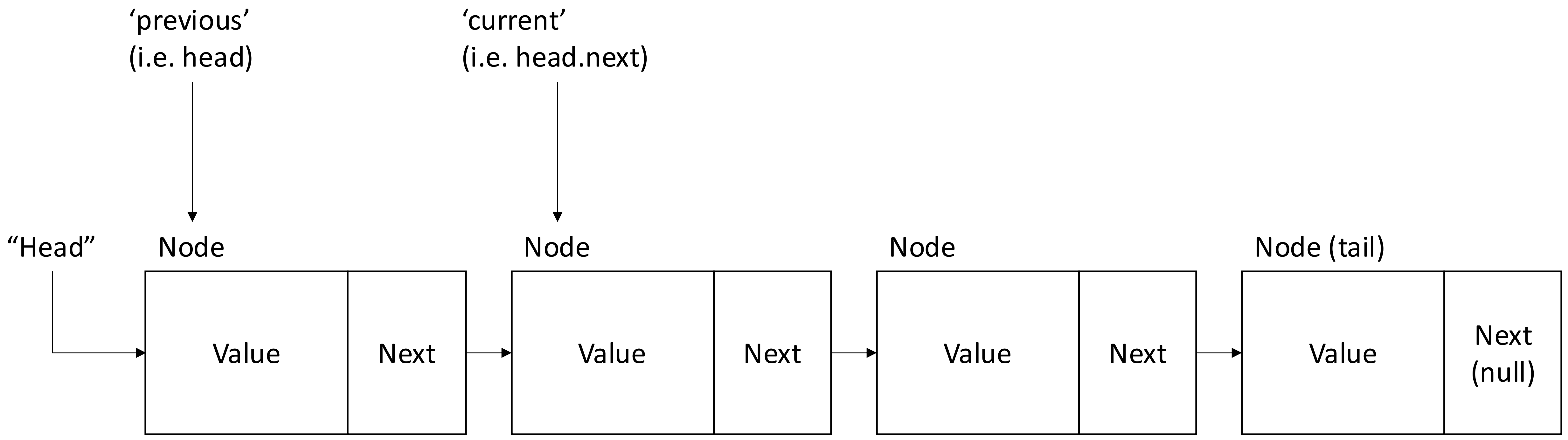


BUT we don't have access to the previous node i.e. we can say "node.next" but not "node.previous". Instead, we 'look ahead'.

For example:

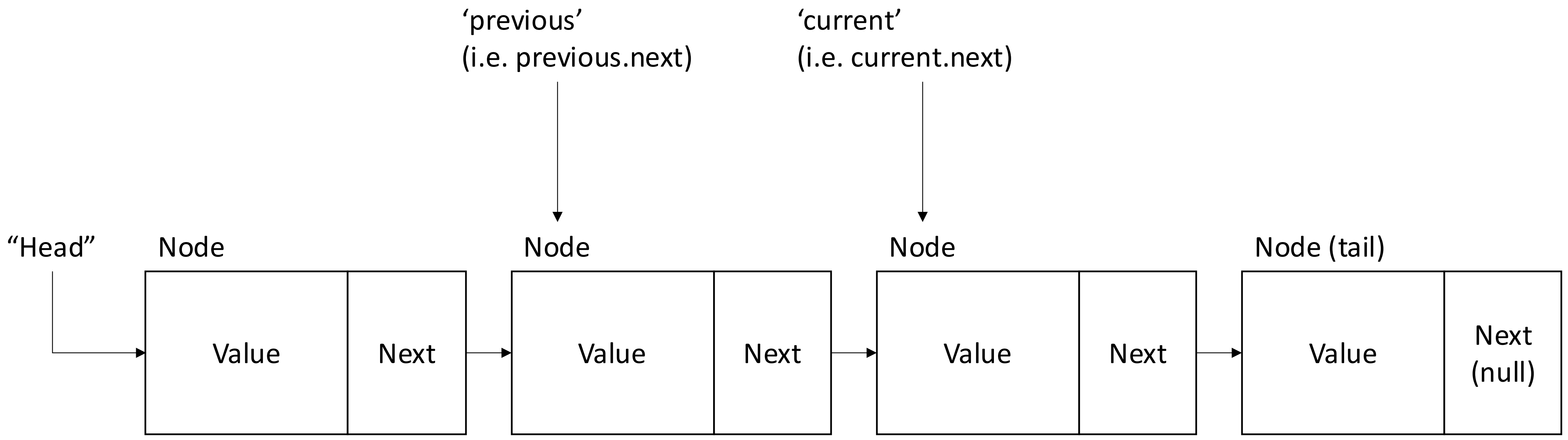
- The 'previous' node = node
- The 'current' node = node.next

Remove (x)



‘previous’
(i.e. head)
‘current’
(i.e. head.next)
“Head”
Node
Value
Next
Node
Value
Next
Node
Value
Next
Node (tail)
Value
Next
(null)

Remove (x)


‘previous’
(i.e. previous.next)
‘current’
(i.e. current.next)
“Head”
Node
Value
Next
Node
Value
Next
Node
Value
Next
Node (tail)
Value
Next
(null)

Remove(x)

Okay, let's try it ...

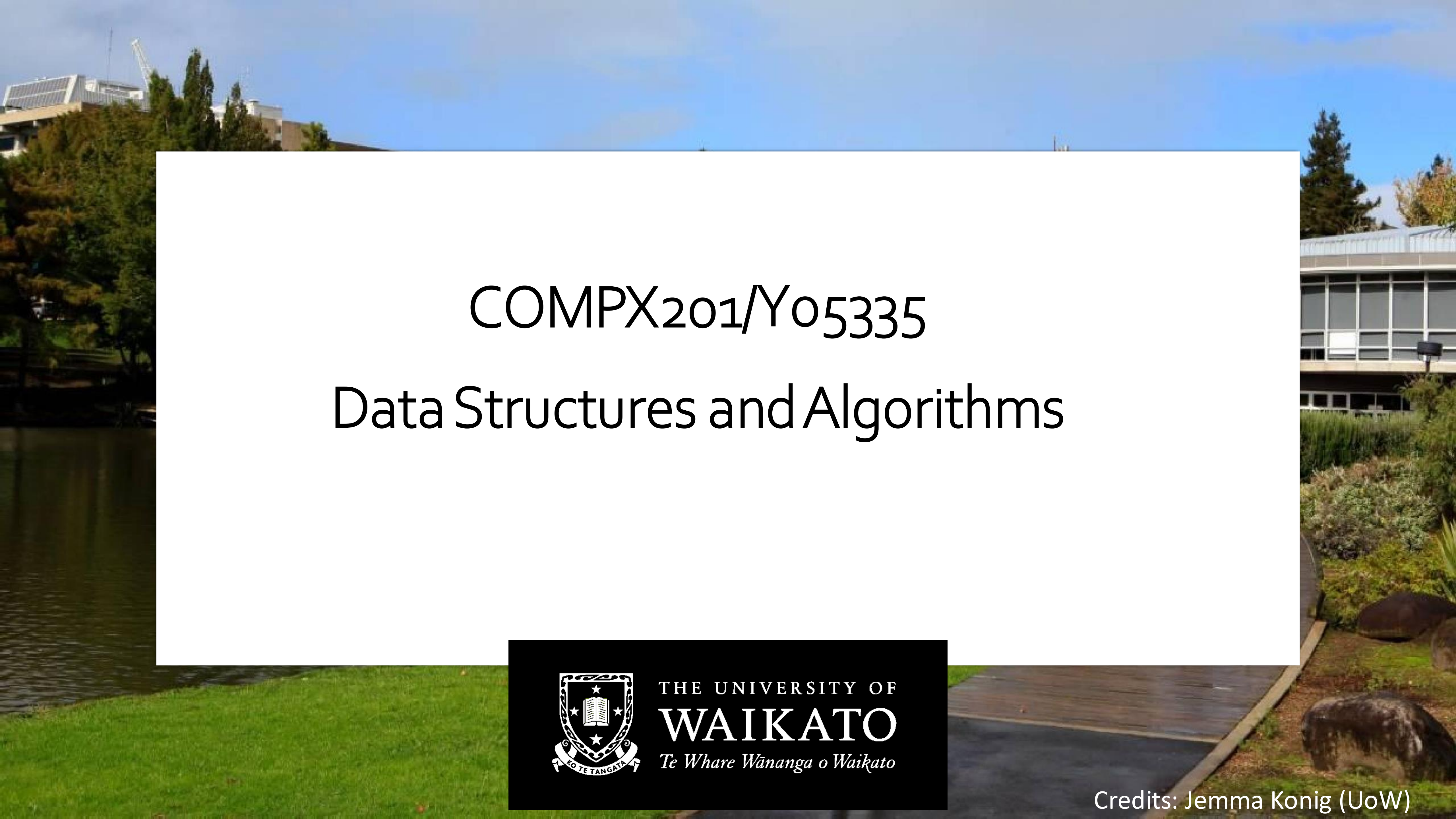
Insert(x)

Used in an ordered list. For example:

10->20->30->50->60

Insert(40)

10->20->30->**40**->50->60



COMPX201/Y05335

Data Structures and Algorithms

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Credits: Jemma Konig (UoW)