Systems Programming

Linked lists, stacks and queues

Julio Villena Román (LECTURER)
<jvillena@it.uc3m.es>

CONTENTS ARE MOSTLY BASED ON THE WORK BY:
Carlos Delgado Kloos and Jesús Arias Fisteus

Universidad
Carlos III de Madrid

Systems Programming

Linked lists

Julio Villena Román (LECTURER)
<jvillena@it.uc3m.es>

CONTENTS ARE MOSTLY BASED ON THE WORK BY:
Carlos Delgado Kloos and Jesús Arias Fisteus

Data Structures

- Abstraction that represents a collection of data in a program in order to ease its manipulation
- The suitability of a data structure depends on the nature of the data to be stored and how that data will be manipulated

Linear Data Structures

- Organize data as a sequence, where each piece of data has a preceding datum (except the first one) and a succeeding datum (except the last one)
- Examples of linear data structures:
  - Arrays
  - Linked lists
  - Stacks
  - Queues
  - Doubly ended queues

Arrays

- Arrays have two main advantages for storing linear data collections:
  - Random access: any position in the array can be accessed in constant time
  - Efficient use of memory when all the positions of the array are in use, because the array is stored in consecutive memory positions

Arrays

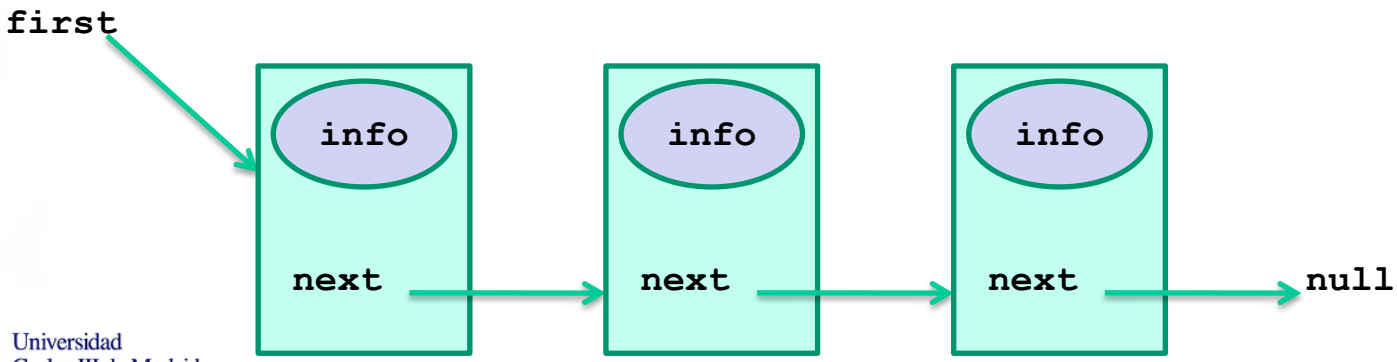
- Disadvantages (I):
  - Static size: a size must be established when the array is created, and cannot be changed later. The main problems it poses are:
    - Inefficient use of memory when more positions than needed are reserved, because of being the array sized for the worst case
    - It may happen at run-time that more positions than reserved are needed
  - Need of contiguous memory:
    - Even having the system enough free memory, it may happen that there is not enough contiguous space, due to memory fragmentation

Arrays

- Disadvantages (II):
  - Some operations on the array have a sub-optimum cost:
    - Insertions and removals of data in the first position or intermediate positions need data to be moved to consecutive memory positions
    - Concatenation of arrays: data has to be copied to a new array
    - Partition of an array in several pieces: data needs to be copied to new arrays

Linked Lists

- Ordered sequence of nodes in which each node stores:
  - A piece of data
  - A reference pointing to the next node
- Nodes do not need to be in consecutive memory positions



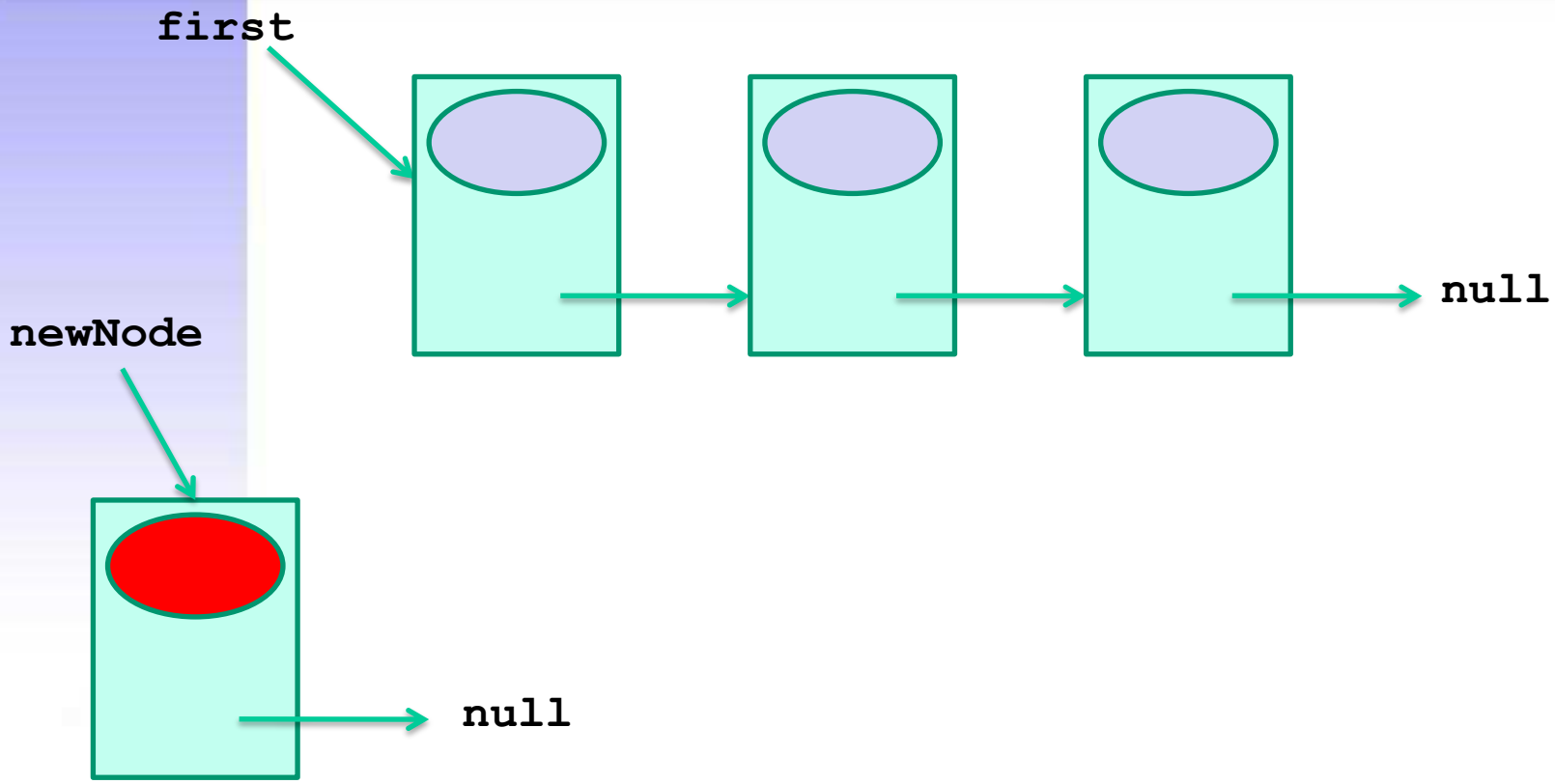
The Node Class

```
public class Node {
    private Object info;
    private Node next;

    public Node(Object info) {…}

    public Node getNext() {…}
    public void setNext(Node next) {…}
    public Object getInfo() {…}
    public void setInfo(Object info) {…}
}
```

The Node Class (Generic types)

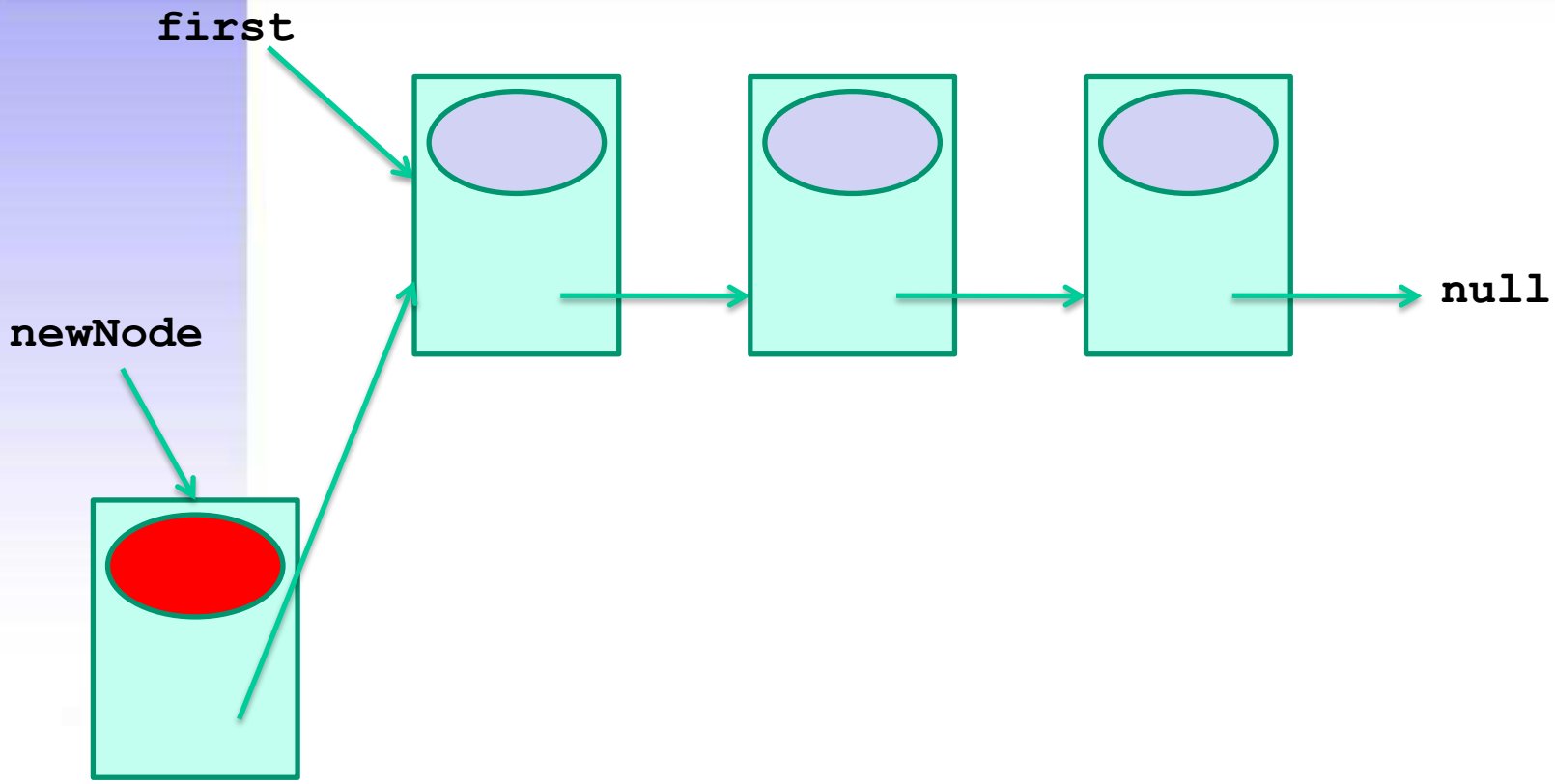
```
public class Node<T> {
    private T info;
    private Node<T> next;

    public Node(T info) {…}

    public Node<T> getNext() {…}
    public void setNext(Node<T> next) {…}
    public T getInfo() {…}
    public void setInfo(T info) {…}
}
```

Inserting a node at the beginning



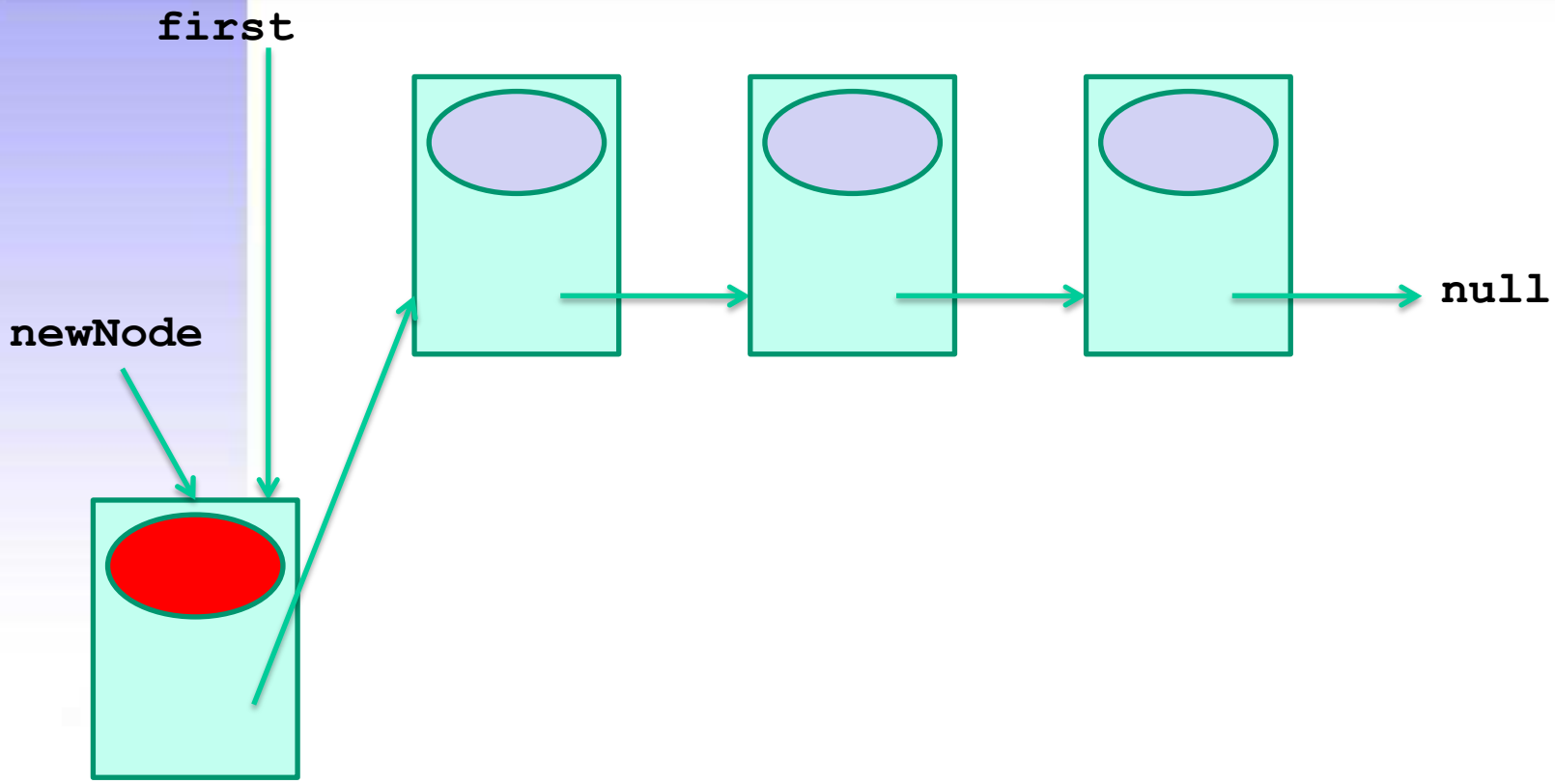

```
Node newNode = new Node(info);
```

Inserting a node at the beginning



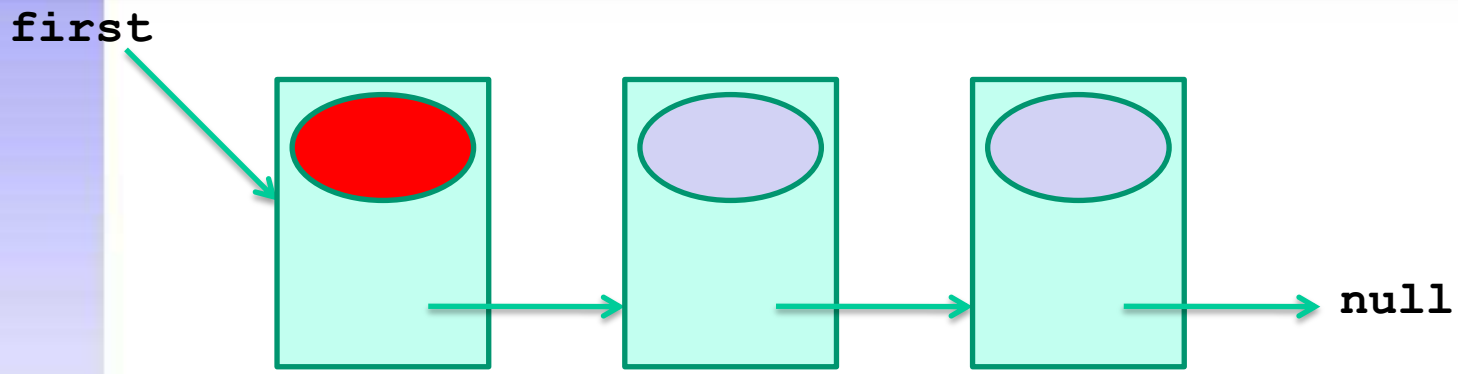

```
newNode.setNext(first);
```

Inserting a node at the beginning

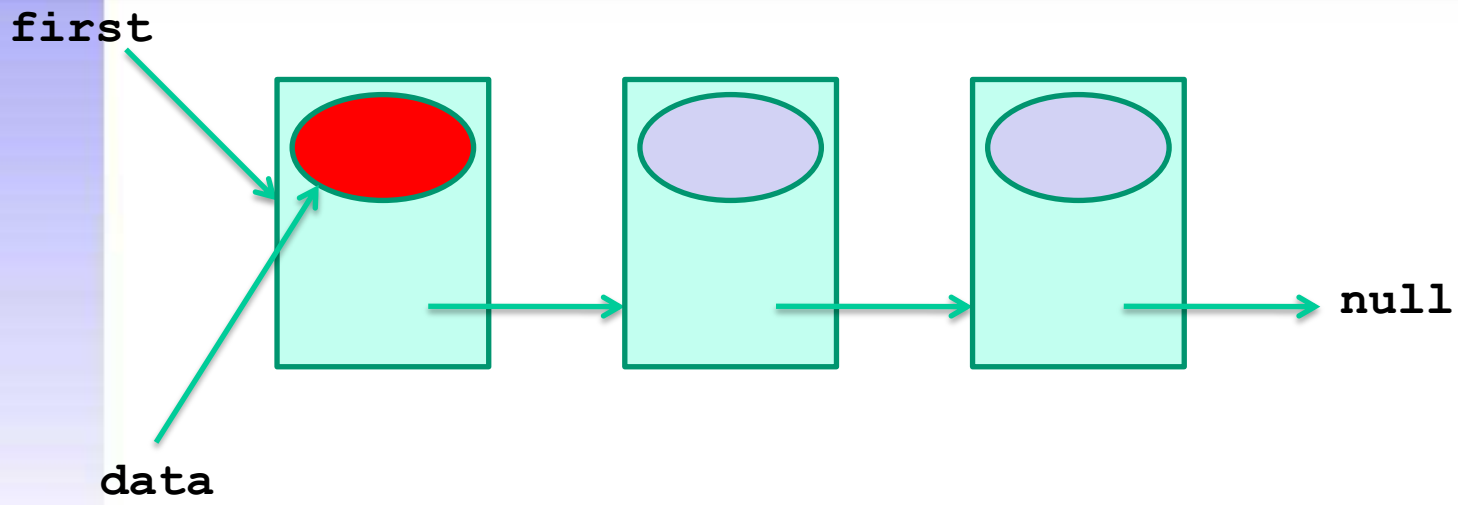



```
first = newNode;
```

Removing the first node

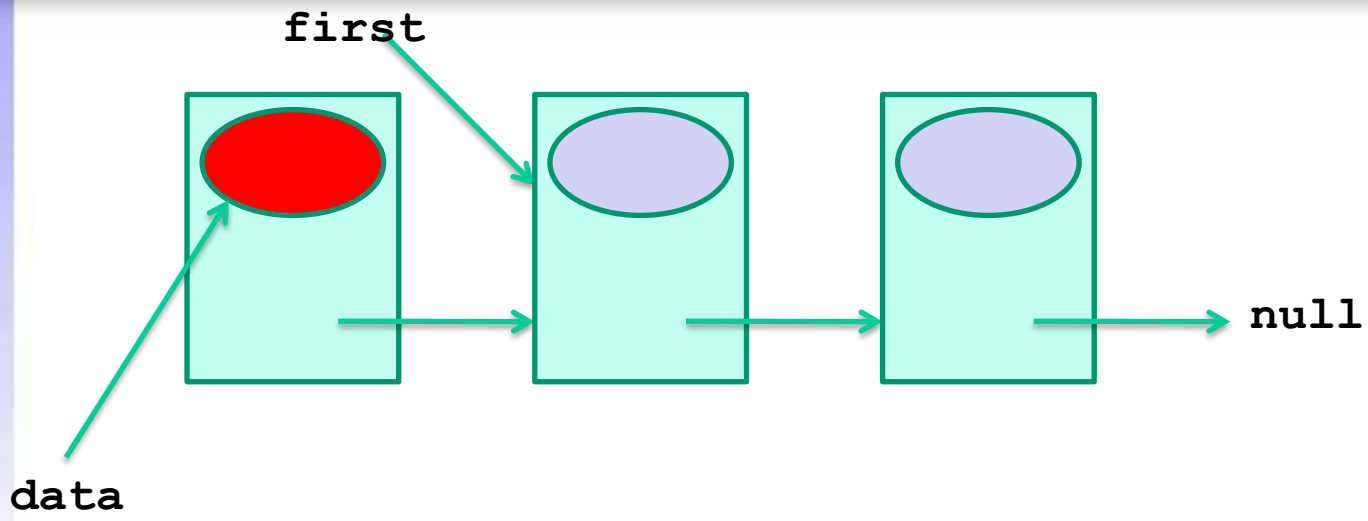


Removing the first node



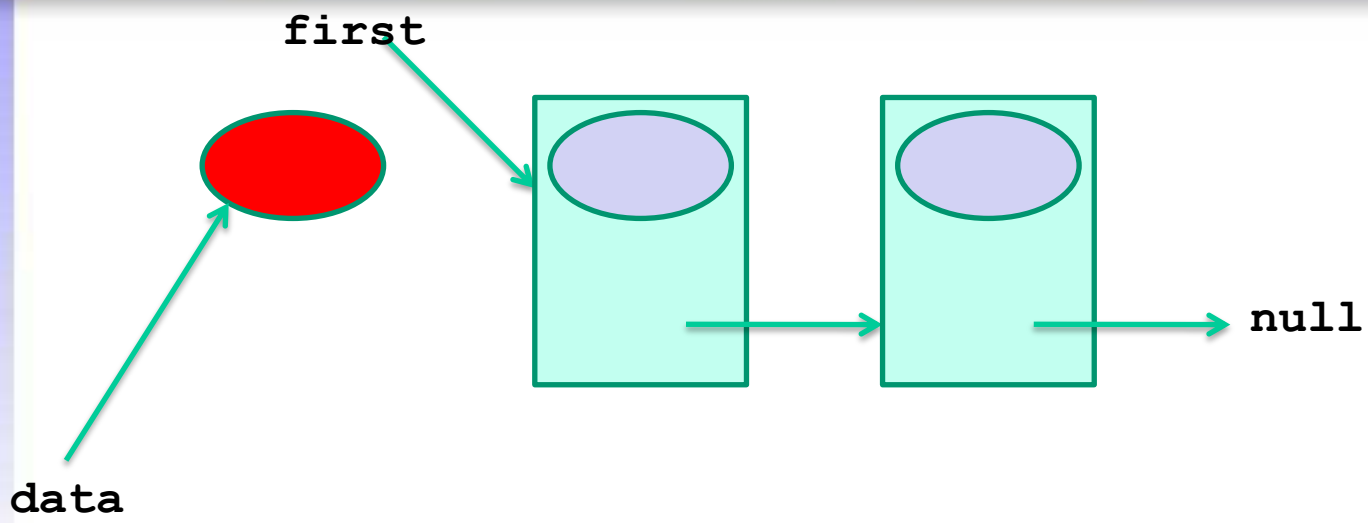

```
Object data = first.getInfo();
 (T)
```

Removing the first node

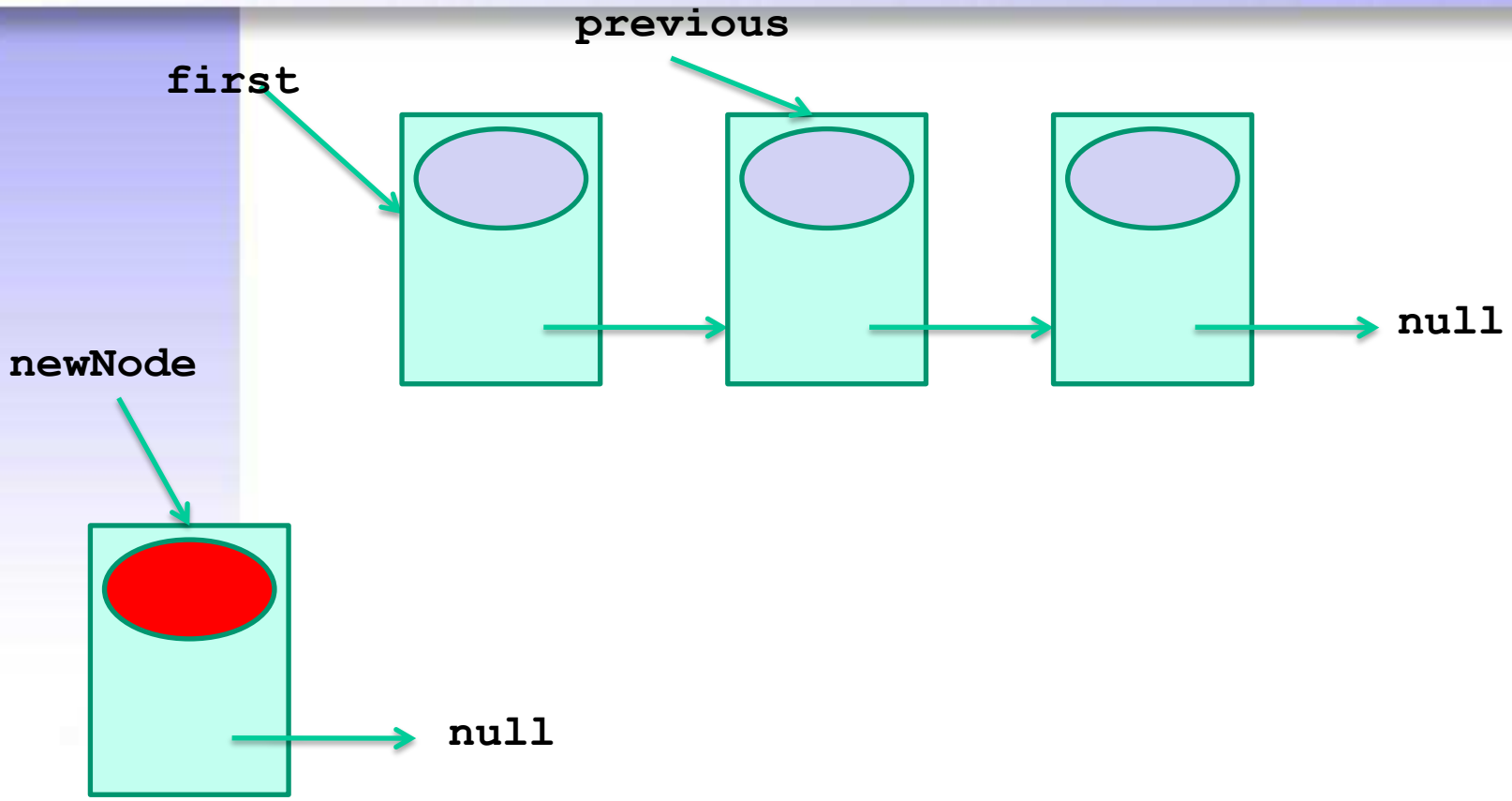



```
first = first.getNext();
```

Removing the first node

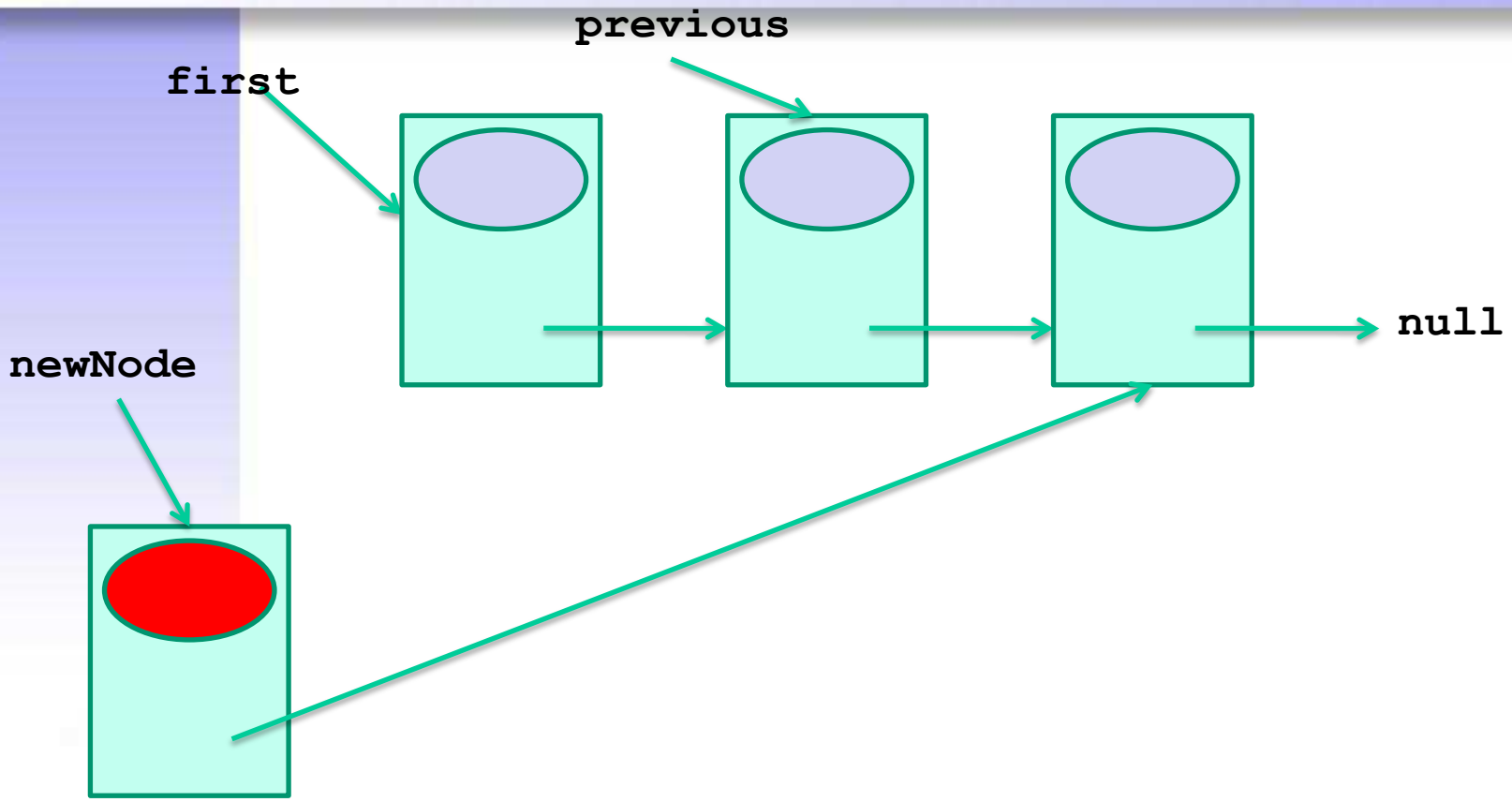


Inserting a node at an intermediate position



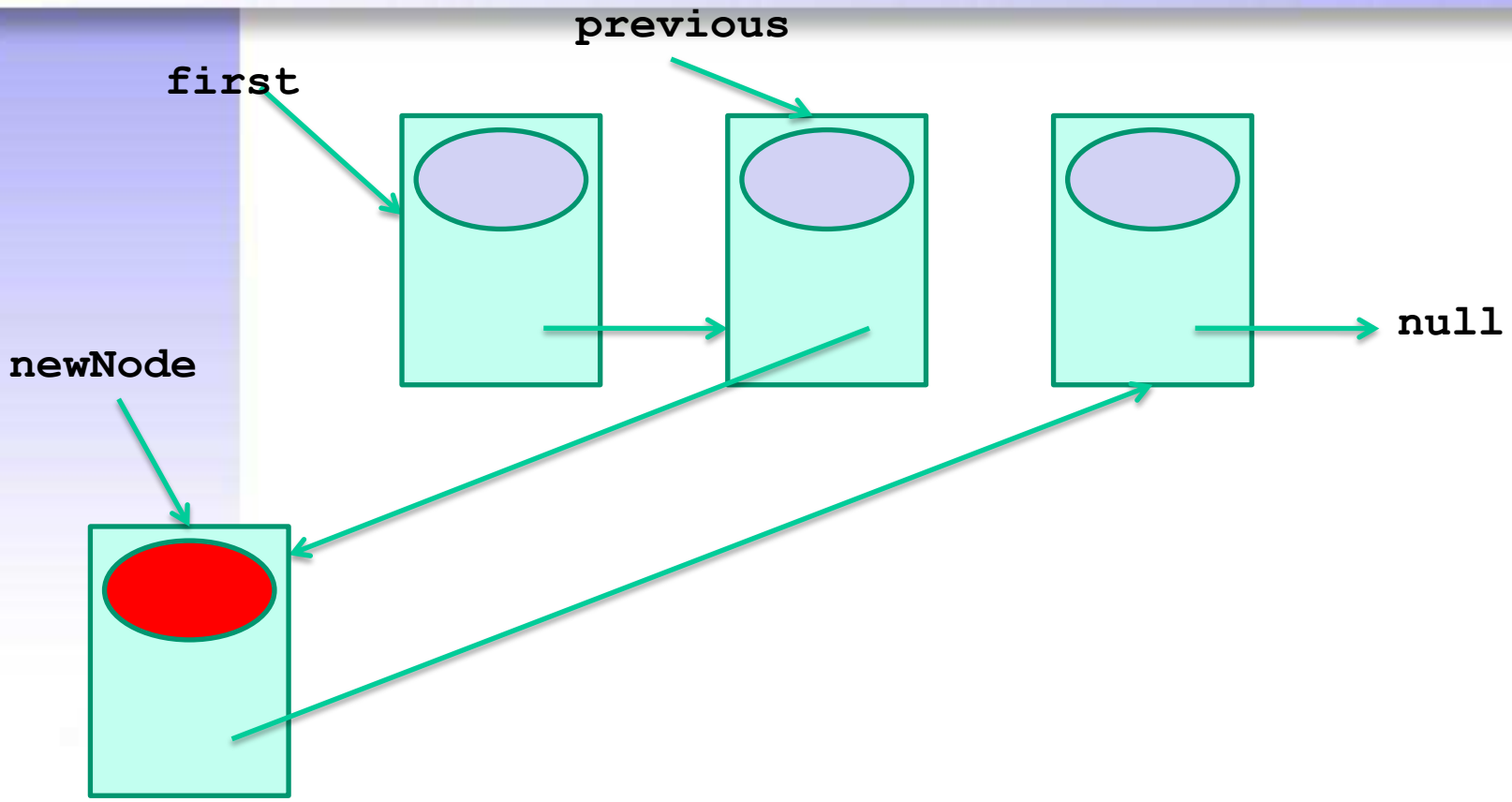

```
Node newNode = new Node(info);
```

Inserting a node at an intermediate position



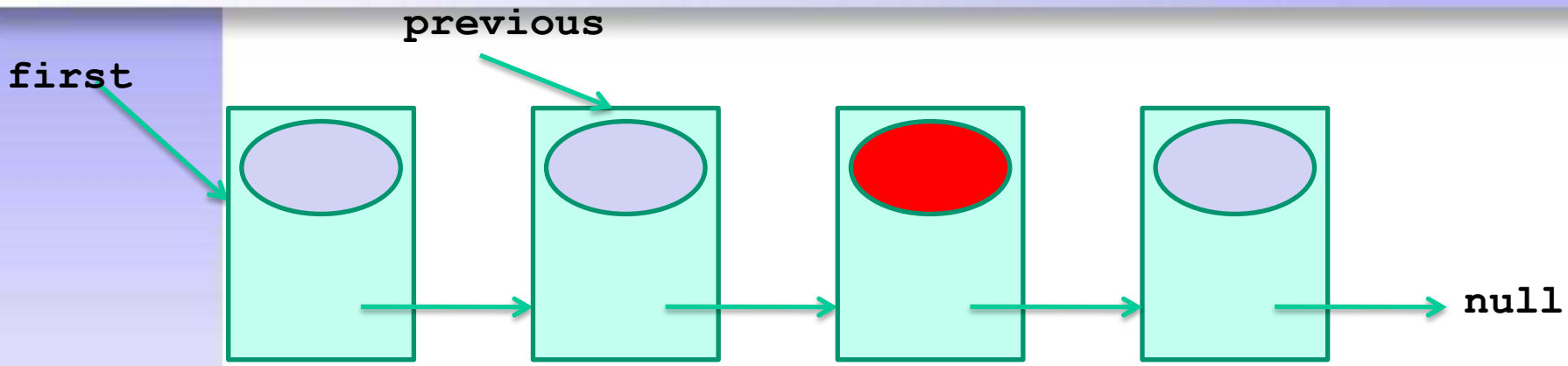

```
newNode.setNext(previous.getNext())
```

Inserting a node at an intermediate position

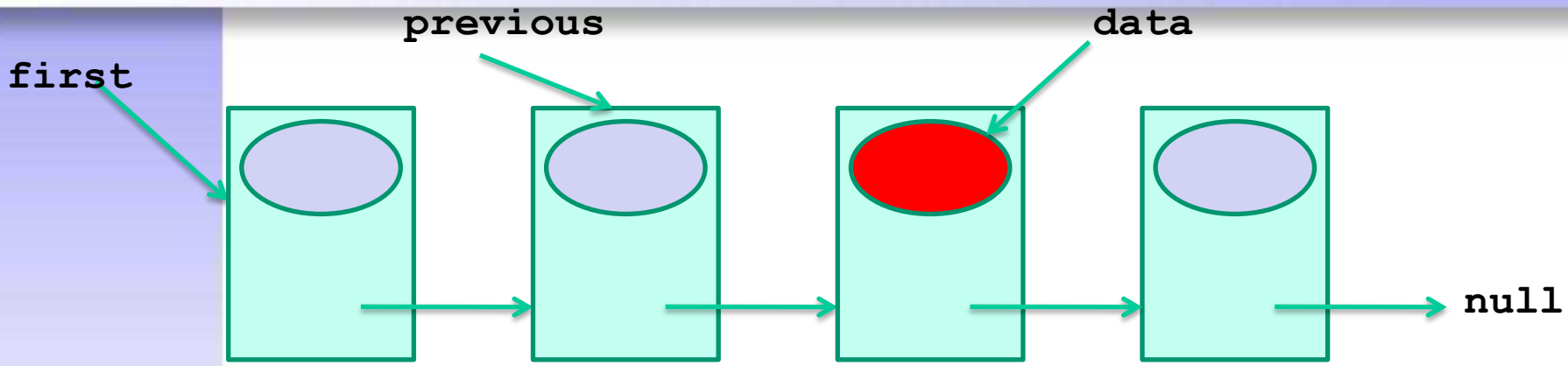



```
previous.setNext(newNode)
```

Removing an intermediate node

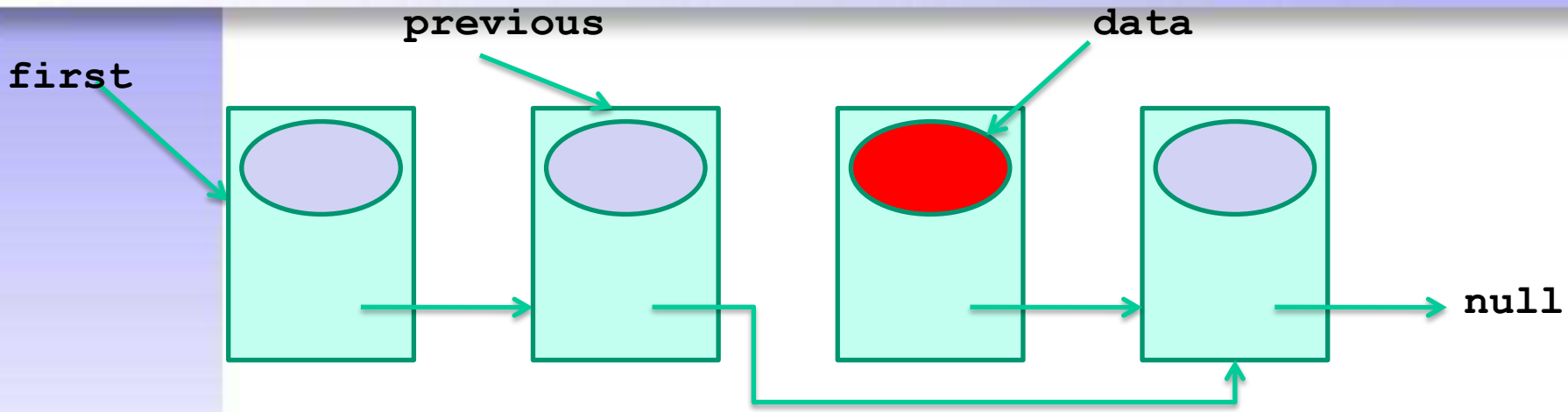


Removing an intermediate node



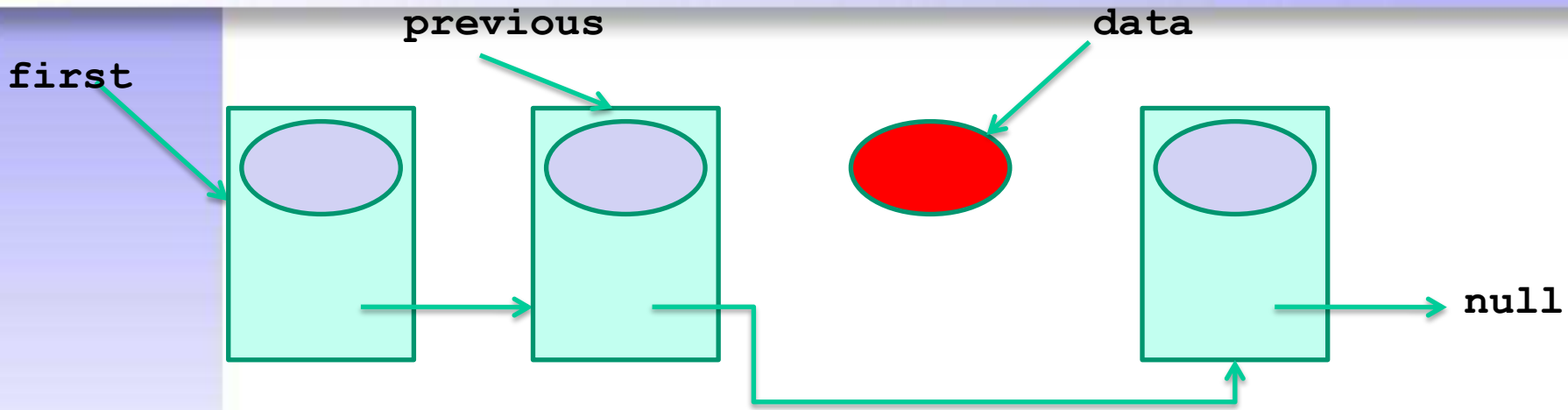

```
Object data = previous.getNext().getInfo();
   (T)
```

Removing an intermediate node

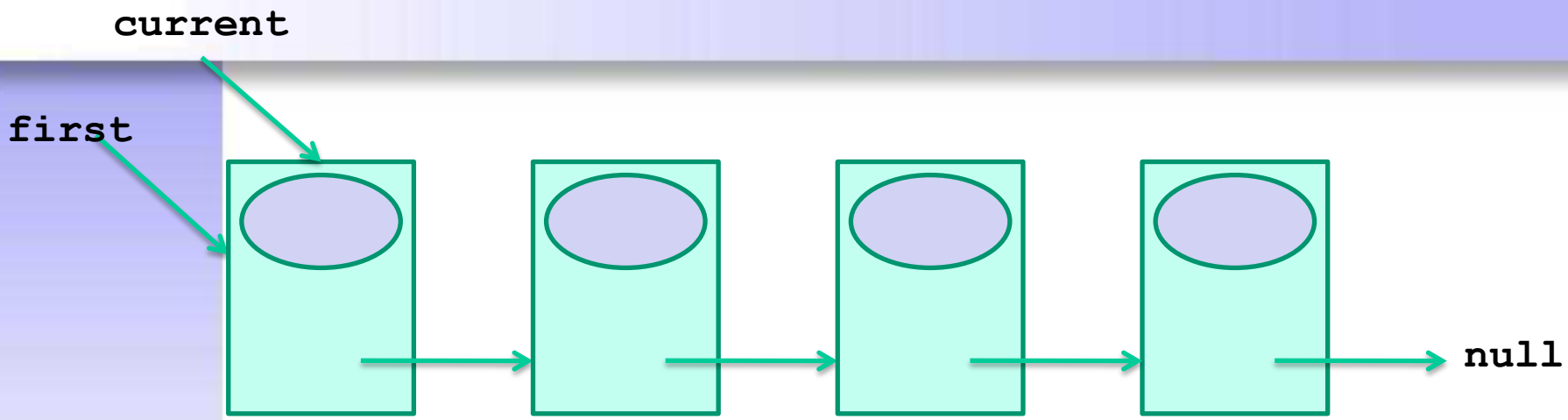



```
previous.setNext(previous.getNext().getNext())
```

Removing an intermediate node



Traversing the linked list




```
Node current = first;
while (current != null)
    current = current.getNext();
```

Traversing the list: looking for the last node

- A reference steps the list until a node is reached whose reference to the next node is null:

```
public Node searchLastNode() {
    Node last = null;
    Node current = first;
    if (current != null) {
        while (current.getNext() != null)
            current = current.getNext();
        last = current;
    }
    return last;
}
```

Traversing the list: looking for a piece of data

- A reference steps the list until the piece of information is reached. A counter is used in order to return its position in the list:

```
public int search(Object info) {
    int pos = 1;
    Node current = first;
    while (current != null
            && !current.getInfo().equals(info)) {
        pos += 1;
        current = current.getNext();
    }
    if (current != null)
        return pos;
    else
        return -1;
}
```

Advantages of Linked Lists

- Inserting and extracting nodes have a cost that does not depend on the size of the list
- Concatenation and partition of lists have a cost that does not depend on the size of the list
- There is no need for contiguous memory
- Memory actually in use at a given instant depends only on the number of data items stored in the list at that instant

Disadvantages of Linked Lists

- Accessing to arbitrary intermediate positions has a cost that depends on the size of the list
- Each node represents an overhead in memory usage

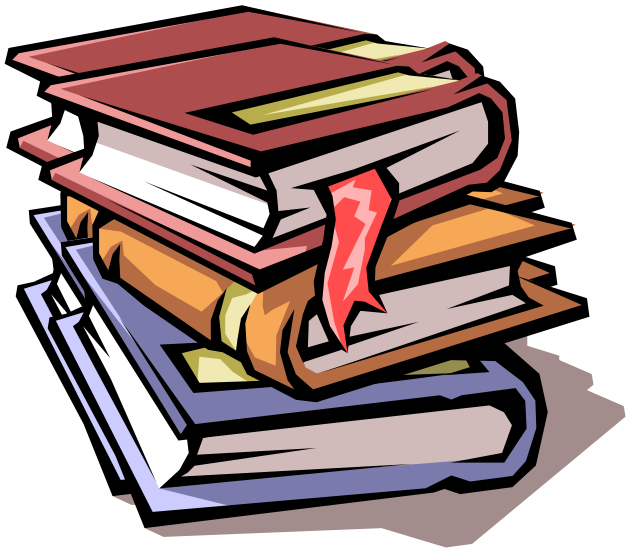
Systems Programming

Stacks

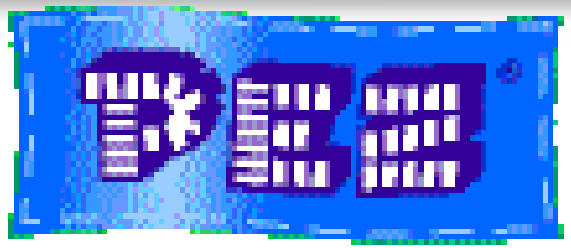
Julio Villena Román (LECTURER)
<jvillena@it.uc3m.es>

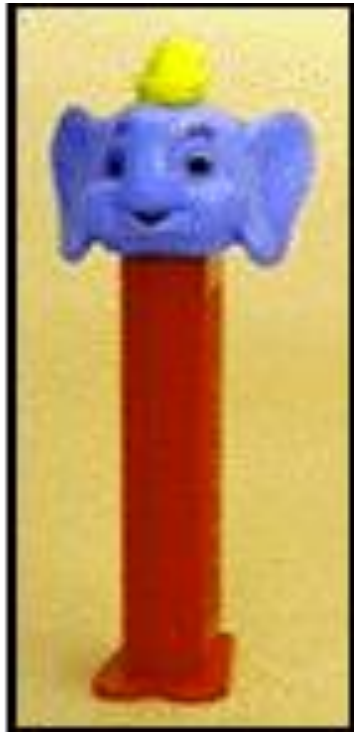
CONTENTS ARE MOSTLY BASED ON THE WORK BY:
Carlos Delgado Kloos

Example

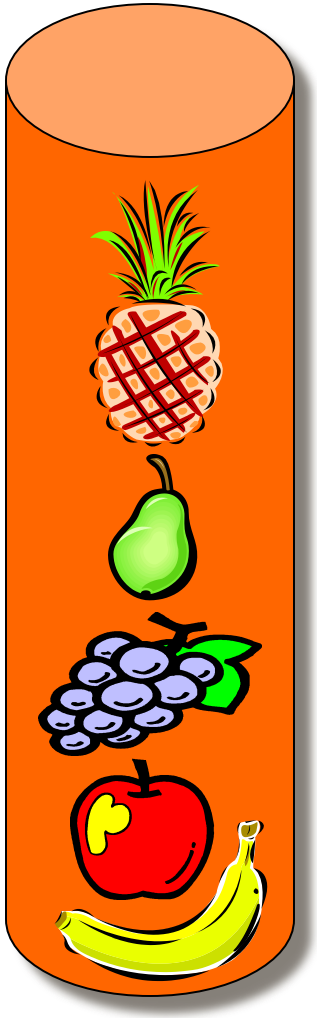


Example





Example



Features

- Linear structure
- Access on one end both for insertion and extraction

Main methods

- Insert on one end:

  `push(x)`

- Extract at the same end:

  `pop()`

Example: Check brackets

- Good:
  -
  - ()
  - (()(()))

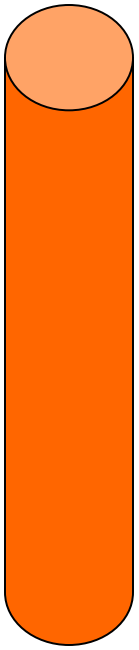
- Bad:
  -)(
  - (()
  - ())

- Rules:
  - Basic: open + close
  - Sequentiation: ()()
  - Nesting: (())

Example: Check brackets

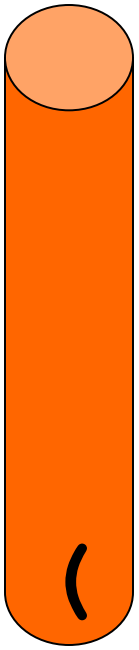
- Rules:
  - Each time we find a “(“ we will put it in the stack
  - Each time we find a “)” we will extract the upper “(“ of the stack
  - The string of parentheses is correct if the stack is empty after having gone through to complete string

Example: check (()(()())())



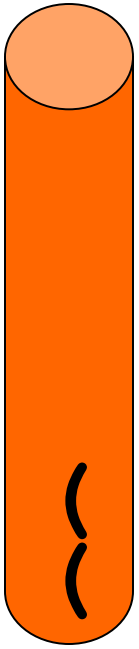
(( ) (( ) ( )) ( ))

Example: check (()(()())())



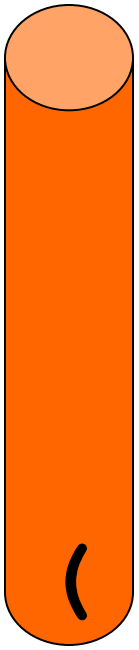
(( ) (( ) ( )) ( ))

Example: check (()(()())())



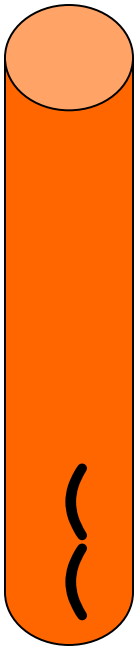
(( ) (( ) ( )) ( ))

Example: check (()(()())())



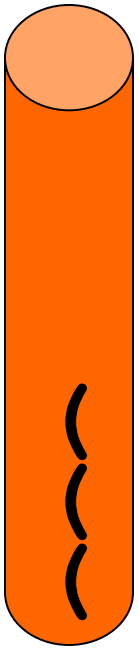
(( ) (( ) ( )) ( ))

Example: check (()(()())())



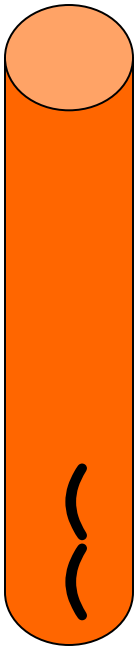
(( ) (( ) ( )) ( ))

Example: check (()(()())())



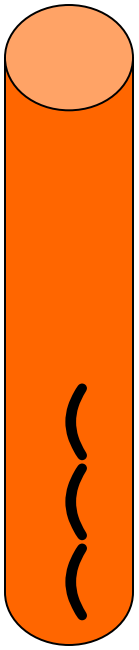
(((( ) ( )) ( ))

Example: check (()(()())())



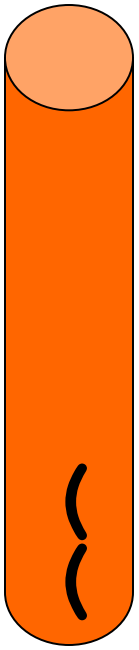
(( ) (( ) ( )) ( ))

Example: check (()(()())())



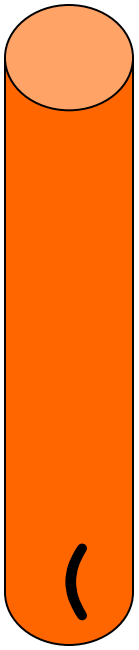
(( ) (( ) ( )) ( ))

Example: check (()(()())())



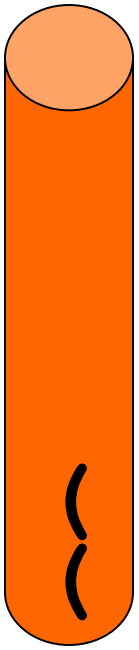
(( ) (( ) ( )) ( ))

Example: check (()(()())())



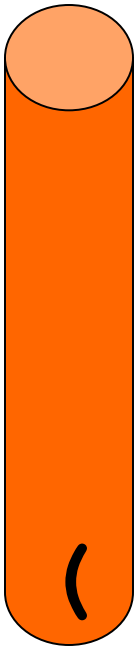
(( ) (( ) ( )) ( ))

Example: check (()(()())())



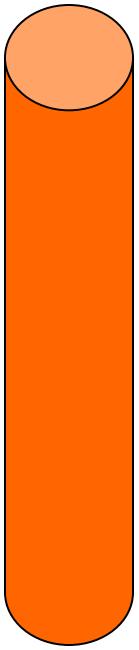
(( ) (( ) ( )) ( ))

Example: check (()(()())())



(( ) (( ) ( ) ( ) ( ))

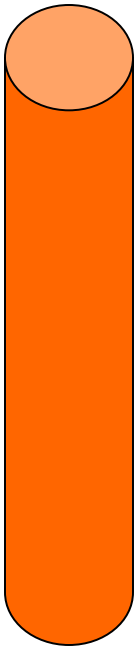
Example: check (()(()())())



Correct: We have completed the string and the stack is empty

(()(()(()())())

Example: check ([]{()<>}())



Correct: We have completed the string and the stack is empty

~~([ ] { ( ) < > } ( ))~~

Example: HTML

```
<b><i>hello</b></i>
```

- ([])
- Correct with HTML 1.0-4.0
- Incorrect with XHTML

```
<b><i>hello</i></b>
```

- ([])
- Correct with HTML 1.0-4.0
- Correct with XHTML

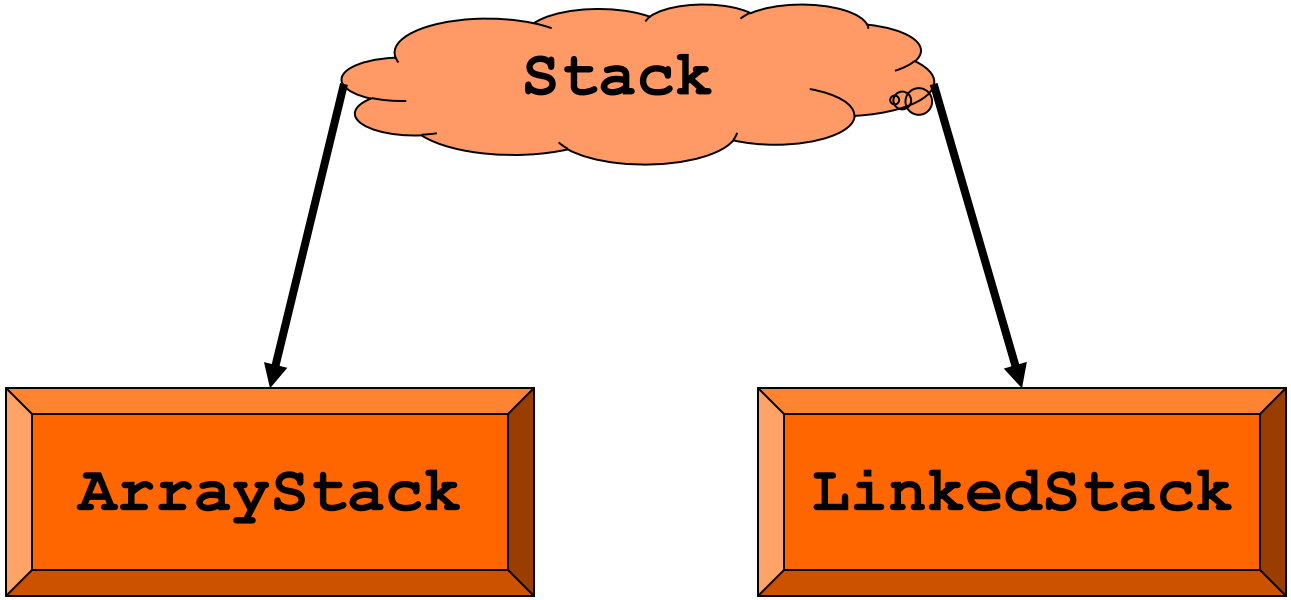
Stack interface

```
public interface Stack {
  public void push(Object o)
    throws StackOverflowException;
  public Object pop()
    throws EmptyStackException;
  public Object top()
    throws EmptyStackException;
  public int size();
  public boolean isEmpty();
}
```

Stack interface (Generic types)

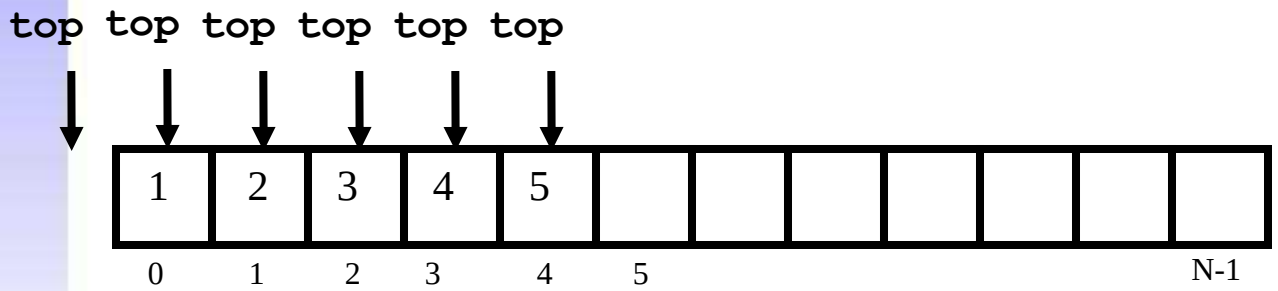
```
public interface Stack<T> {
  public void push(T o)
    throws StackOverflowException;
  public T pop()
    throws EmptyStackException;
  public T top()
    throws EmptyStackException;
  public int size();
  public boolean isEmpty();
}
```

One interface and several implementations



Array-based implementation

data



Array-based implementation

```
public class ArrayStack<T> implements Stack<T> {
  public static final int DEFAULT_CAPACITY = 1000;
  private int capacity;
  private T data[];
  private int top = -1;
  public ArrayStack() {
    this(DEFAULT_CAPACITY);
  }
  public ArrayStack(int capacity) {
    this.capacity = capacity;
    data = new T[capacity];
  }
  …
```

Array-based implementation

```
…
public int size() {
  return (top + 1);
}
public boolean isEmpty() {
  return (top < 0);
}
public T top() throws EmptyStackException {
  if (isEmpty())
    throw new EmptyStackException("empty");
  return data[top];
}
…
```

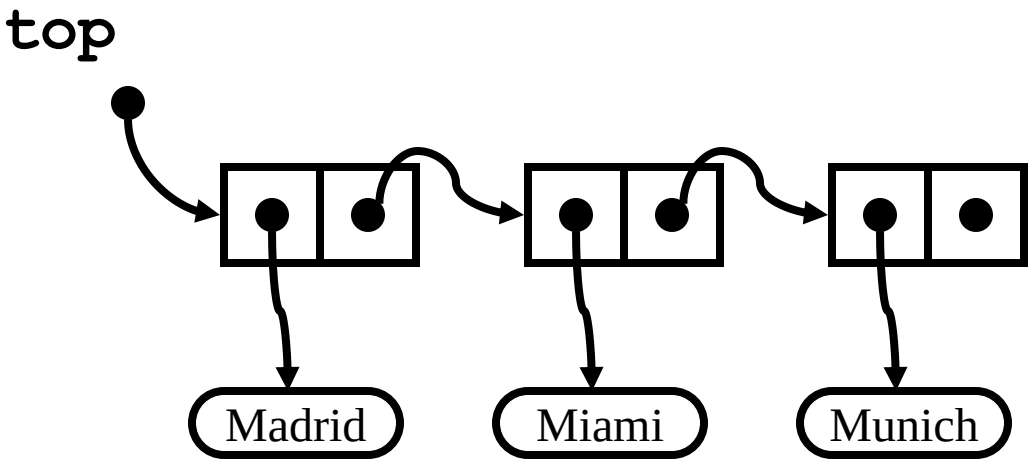
Array-based implementation

```
…
public void push(T o)
   throws StackOverflowException {
   if (size == capacity)
     throw new StackOverflowException();
   data[++top] = o;
}
…
```

Array-based implementation

```
  …
  public T pop() throws StackEmptyException {
    T o;
    if (top == -1)
      throw new EmptyStackException();
    o = data[top];
    data[top--] = null;
    return o;
  }
}
```

Implementation based on linked lists



Implementation based on linked lists

```
public class Node<T> {
  private T info;
  private Node<T> next;
  public Node(T info, Node next) {
    this.info = info;
    this.next = next;
  }
  void setInfo(T info) {this.info = info;}
  void setNext(Node<T> next) {this.next = next;}
  T getInfo() {return info;}
  Node<T> getNext() {return next;}
}
```

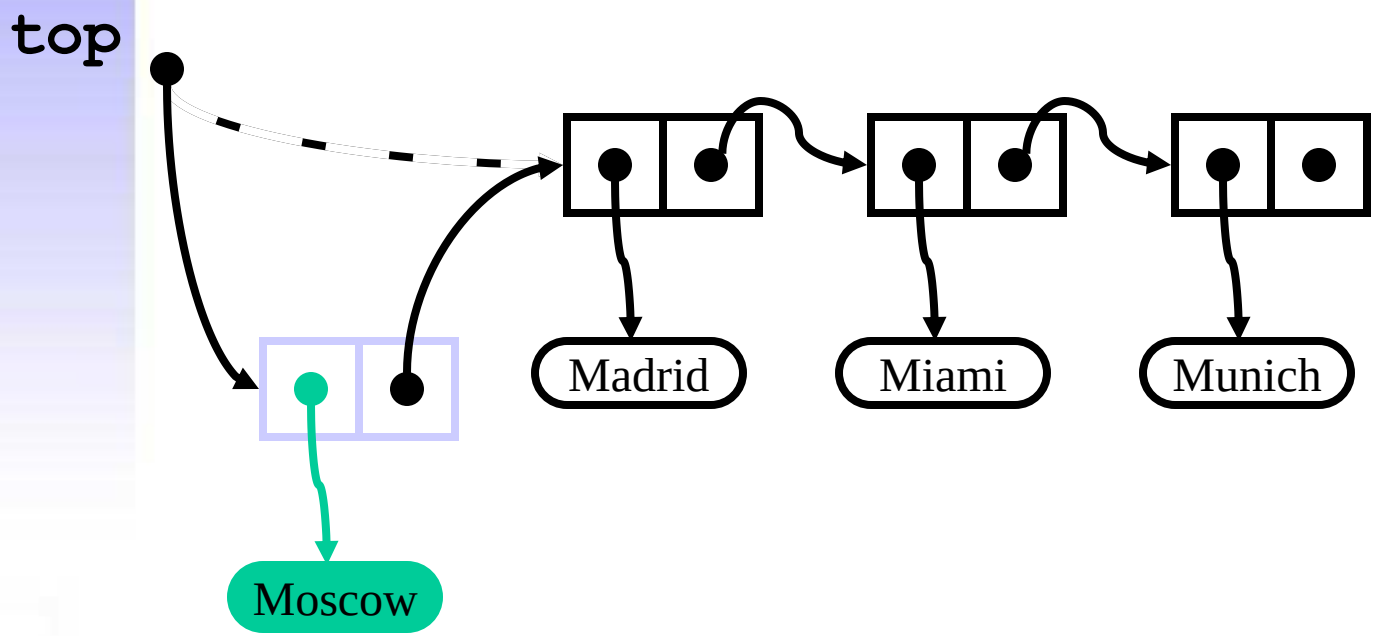
Implementation based on linked lists

```
public class LinkedStack<T> implements Stack<T> {
  private Node<T> top;
  private int size;

  public LinkedStack() {
    top = null;
    size = 0;
  }
  …
```

Implementation based on linked lists

```
…
public int size() {
  return size;
}
public boolean isEmpty() {
  return (top == null);
}
public T top() throws EmptyStackException {
  if (top == null)
    throw new EmptyStackException();
  return top.getInfo();
}
…
```

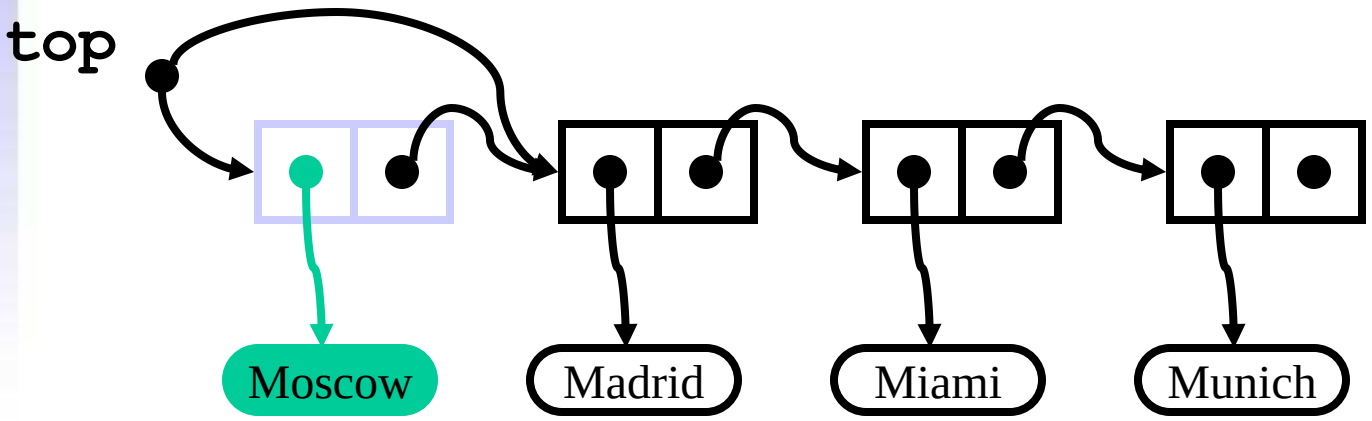
Insertion (`push`)



Implementation based on linked lists

```
…
public void push(T info) {
   Node<T> n = new Node<T>(info, top);
   top = n;
   size++;
}
…
```

Extraction (pop)



Implementation based on linked lists

```
  …
  public T pop() throws EmptyStackException {
    T o;
    if (isEmpty())
      throw new EmptyStackException();
    o = top.getInfo();
    top = top.getNext();
    size--;
    return o;
  }
}
```

Systems Programming

Queues

Julio Villena Román (LECTURER)
<jvillena@it.uc3m.es>

CONTENTS ARE MOSTLY BASED ON THE WORK BY:
Carlos Delgado Kloos

Example

- The queue at the bus stop
- The printer queue

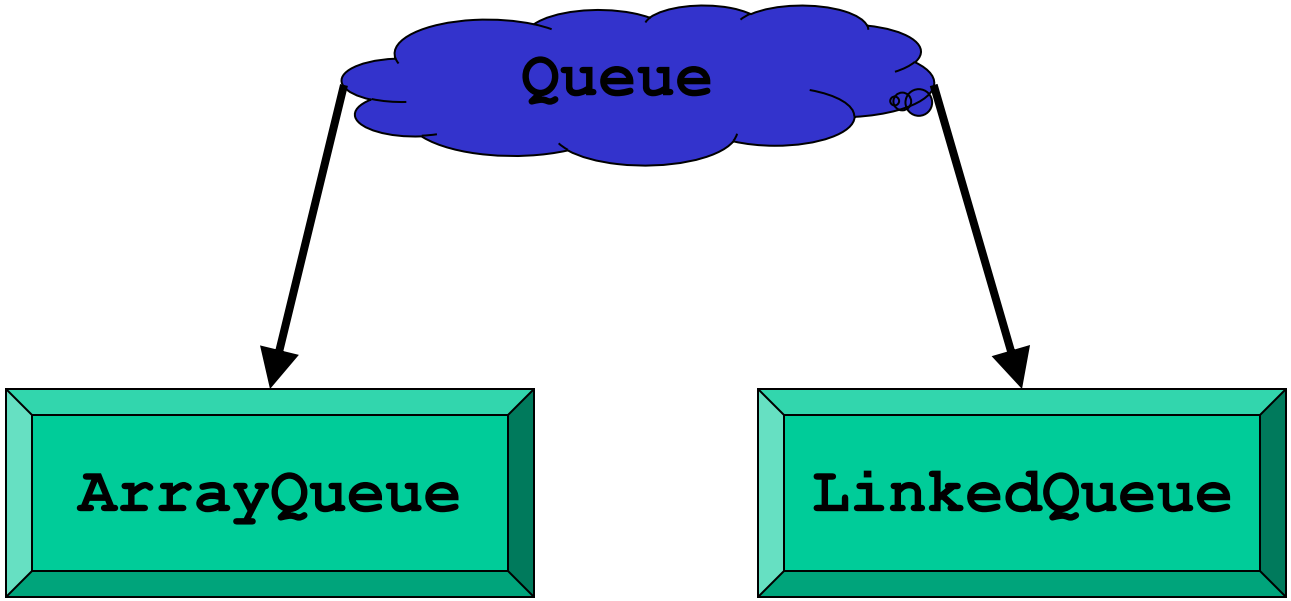
Characteristics

- Linear structure
- Access on one end for insertion and on the other for extraction

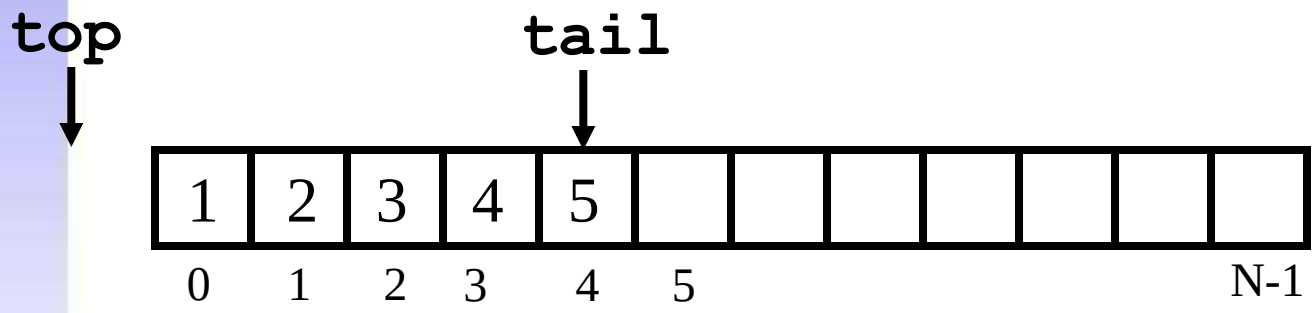
Queue interface

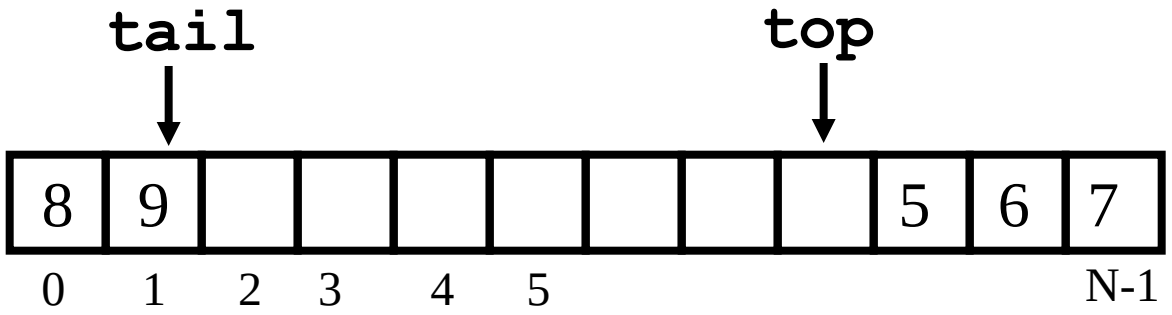
```
public interface Queue<T> {
  public int size();
  public boolean isEmpty();
  public void enqueue(T o)
    throws QueueOverflowException;
  public T dequeue()
    throws EmptyQueueException;
  public T front()
    throws EmptyQueueException;
}
```

One interface and several implementations

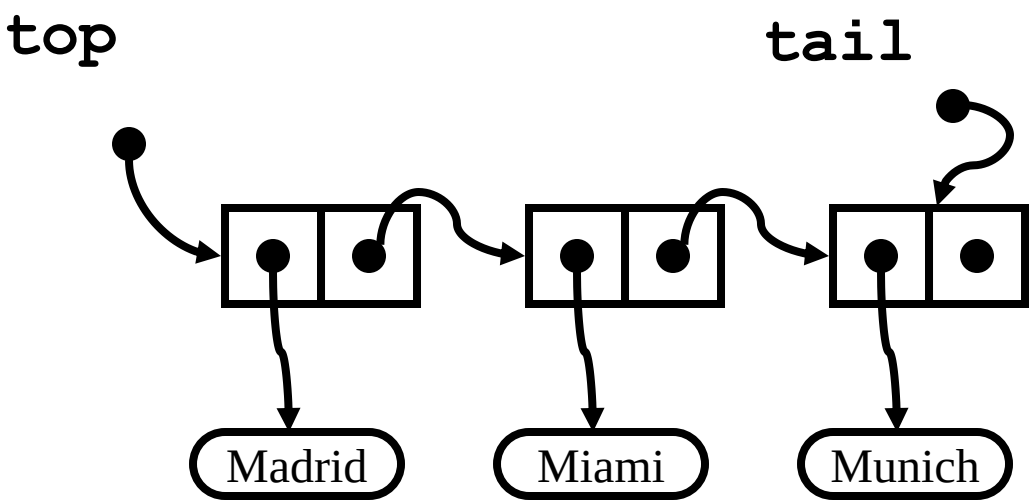


Array-based implementation





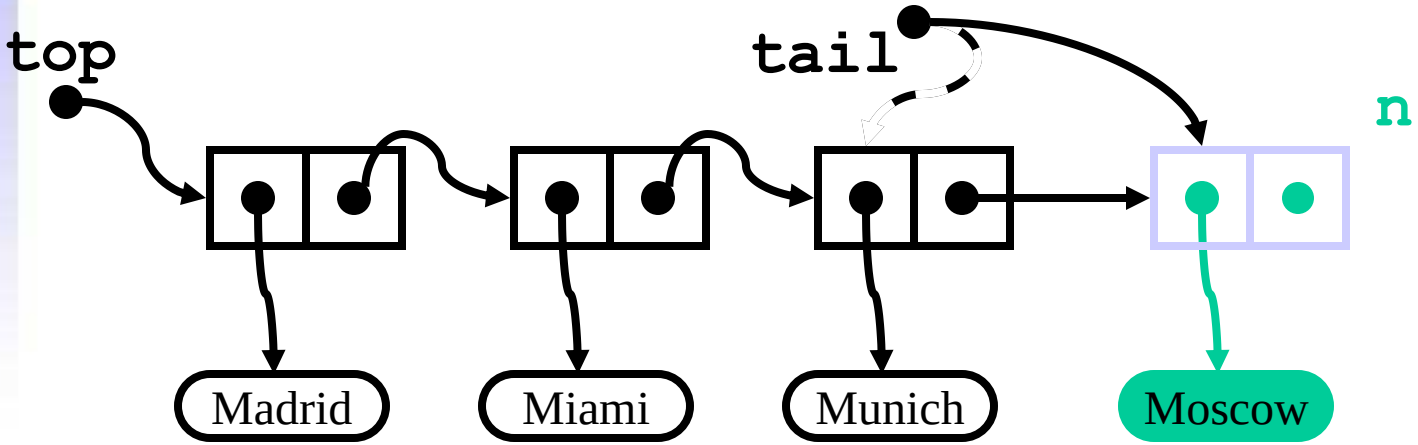
Implementation based on linked lists



Implementation based on linked lists

```
public class LinkedQueue<T> implements Queue<T> {
  private Node<T> top = null;
  private Node<T> tail = null;
  private int size = 0;
  …
}
```

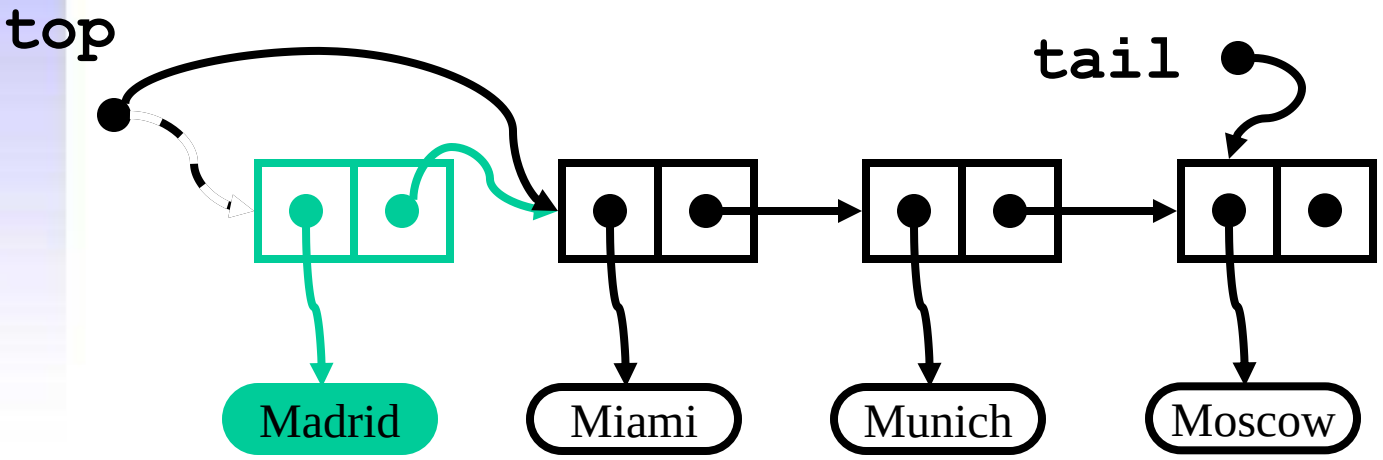
Insertion (enqueue)



Implementation based on linked lists

```
public void enqueue(T info) {
  Node<T> n = new Node<T>(info, null);
  if (top == null)
    top = n;
  else
    tail.setNext(n);
  tail = n;
  size++;
}
```

Extraction (dequeue)



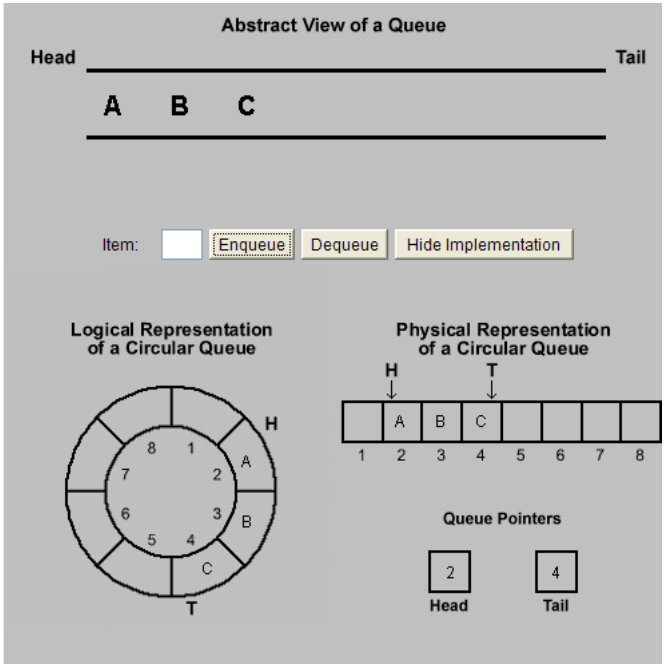
Implementation based on linked lists

```
public T dequeue()
   throws EmptyQueueException {
   T o;
   if (top == null)
      throw new EmptyQueueException();
   o = top.getInfo();
   top = top.getNext();
   if (top == null)
      tail = null;
   size--;
   return o;
}
```

Activity

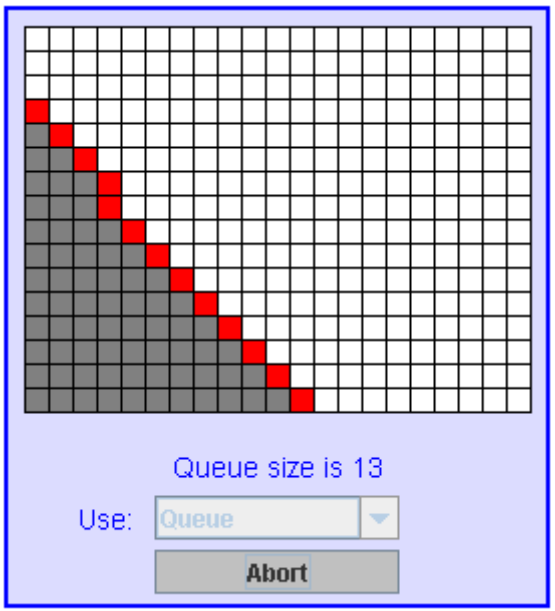
- View queue animations:

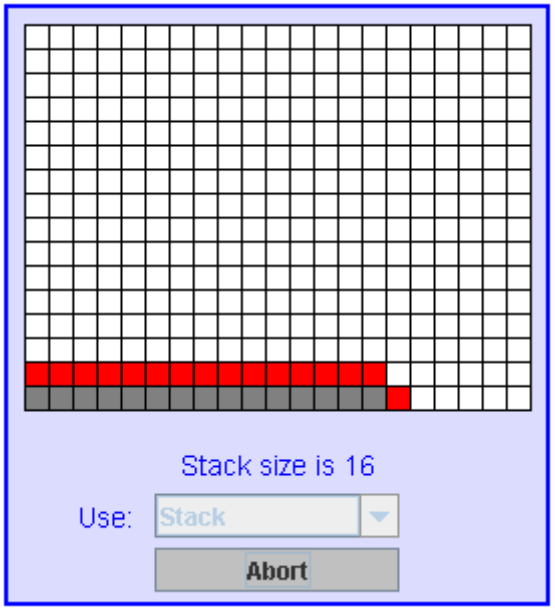
  http://courses.cs.vt.edu/csonline/DataStructures/Lessons/QueuesImplementationView/applet.html



Activity

- Try the applet **`DepthBreadth.java`** that can be found here:
  `http://www.faqs.org/docs/javap/c11/s3.html`

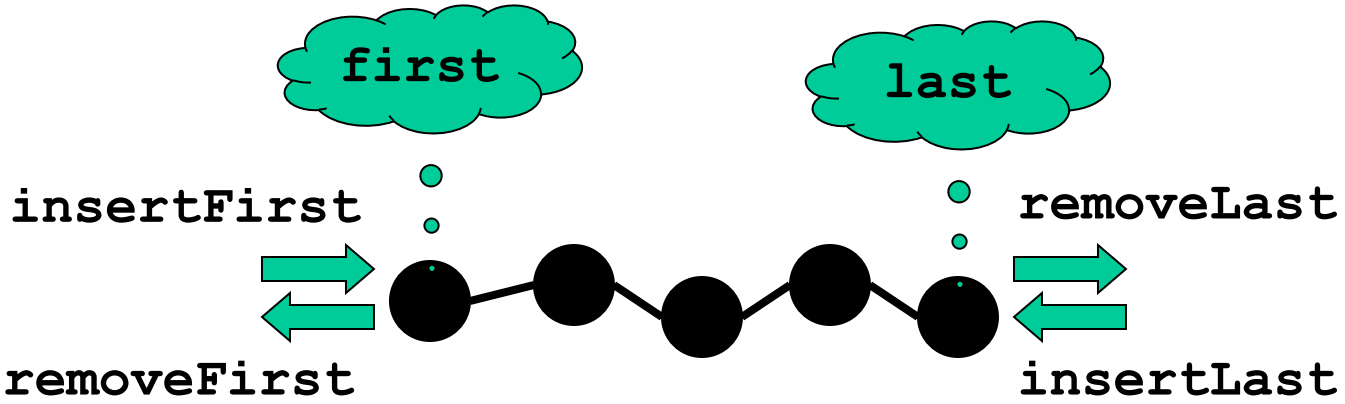




Other kinds of queues (not queues any more)

- Double-ended queues
- Priority queues

Deques
(Double-ended queues)



Interface for deques

```
public interface Deque<T> {
   public int size();
   public boolean isEmpty();

   public void insertFirst(T info);
   public void insertLast(T info);
   public T removeFirst() throws EmptyDequeException;
   public T removeLast() throws EmptyDequeException;

   public T first() throws EmptyDequeException;
   public T last() throws EmptyDequeException;
}
```

Stacks and queues as deques

Stack	Deque
`size()`	`size()`
`isEmpty()`	`isEmpty()`
`top()`	`last()`
`push(o)`	`insertLast(o)`
`pop()`	`removeLast()`

Queue	Deque
`size()`	`size()`
`isEmpty()`	`isEmpty()`
`front()`	`first()`
`enqueue(o)`	`insertLast(o)`
`dequeue()`	`removeFirst()`

Definition of stacks from deques

```
public class DequeStack<T> implements Stack<T> {
  private Deque<T> deque;

  public DequeStack() {
    deque = new Deque<T>();
  }
  public int size() {
    return deque.size();
  }
  public boolean isEmpty() {
    return deque.isEmpty();
  }
```

Definition of stacks from deques

```
public void push(T info) {
   deque.insertLast(info);
}
public T pop()
   throws EmptyStackException {
   try {
      return deque.removeLast();
   } catch (EmptyDequeException e) {
      throw new EmptyStackException();
   }
}
```

Definition of stacks from deques

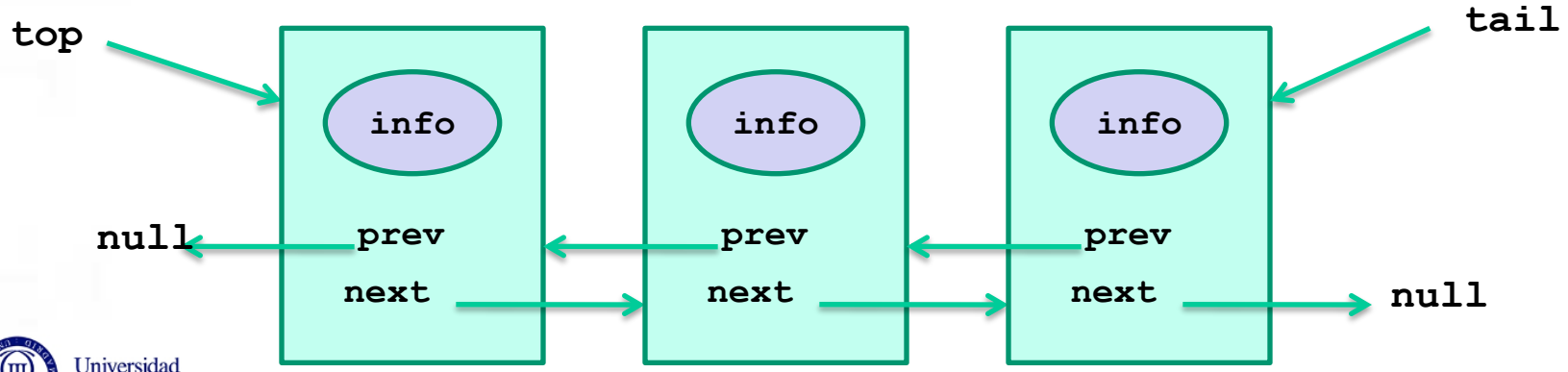
```
public T top()
   throws EmptyStackException {
   try {
      return deque.last();
   } catch (EmptyDequeException e) {
      throw new EmptyStackException();
   }
}
```

Implementation of deques based on lists

- Singly-linked lists are not appropriate because `removeLast` requires the whole list to be traversed, in order to get the reference of the last-but-one node
- Solution: doubly-linked lists

Doubly Linked Lists

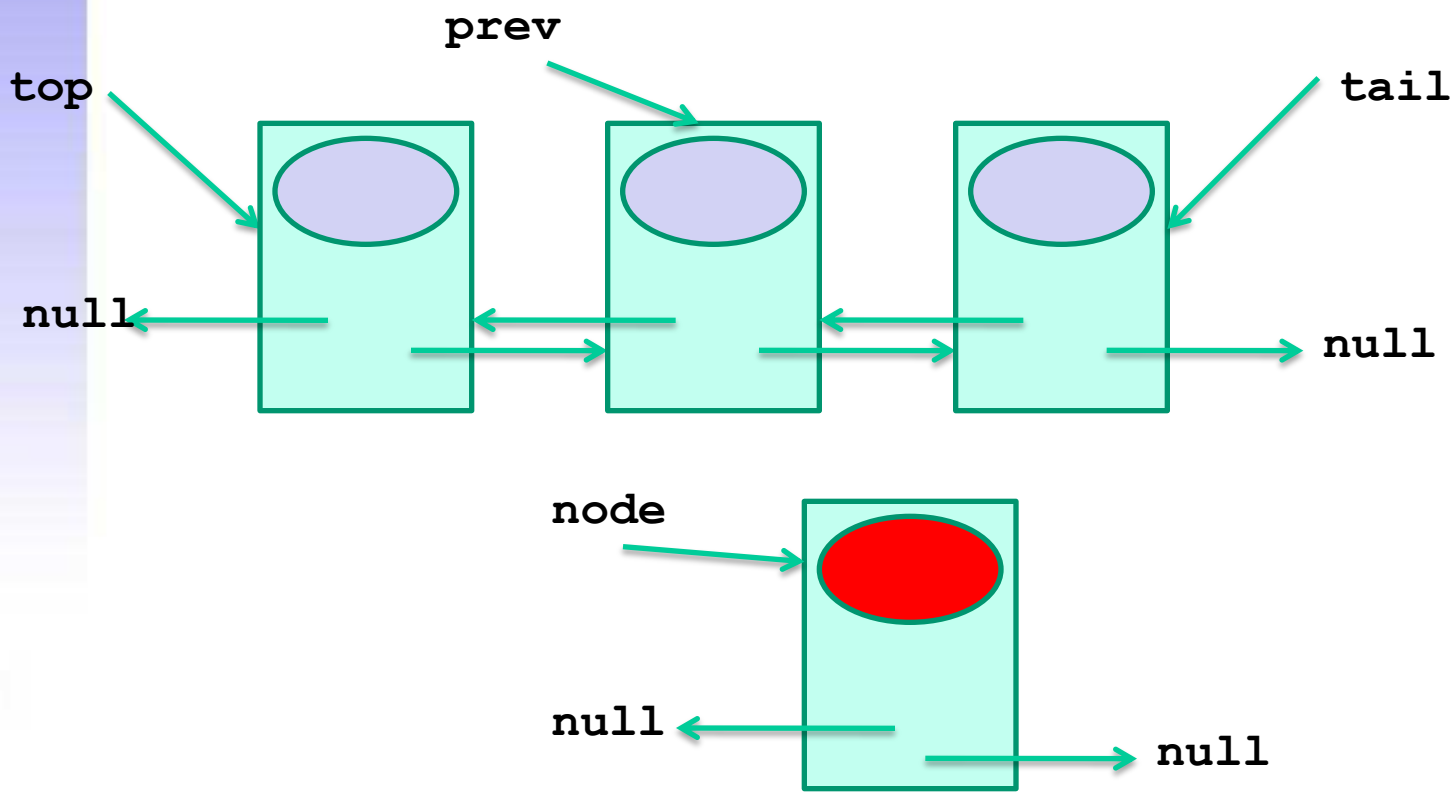
- Linked lists in which each node has an additional reference pointing to the previous node in the list
  - Can be traversed both from the beginning to the end and vice-versa
  - `removeLast` does not need the whole list to be traversed



The DLNode class

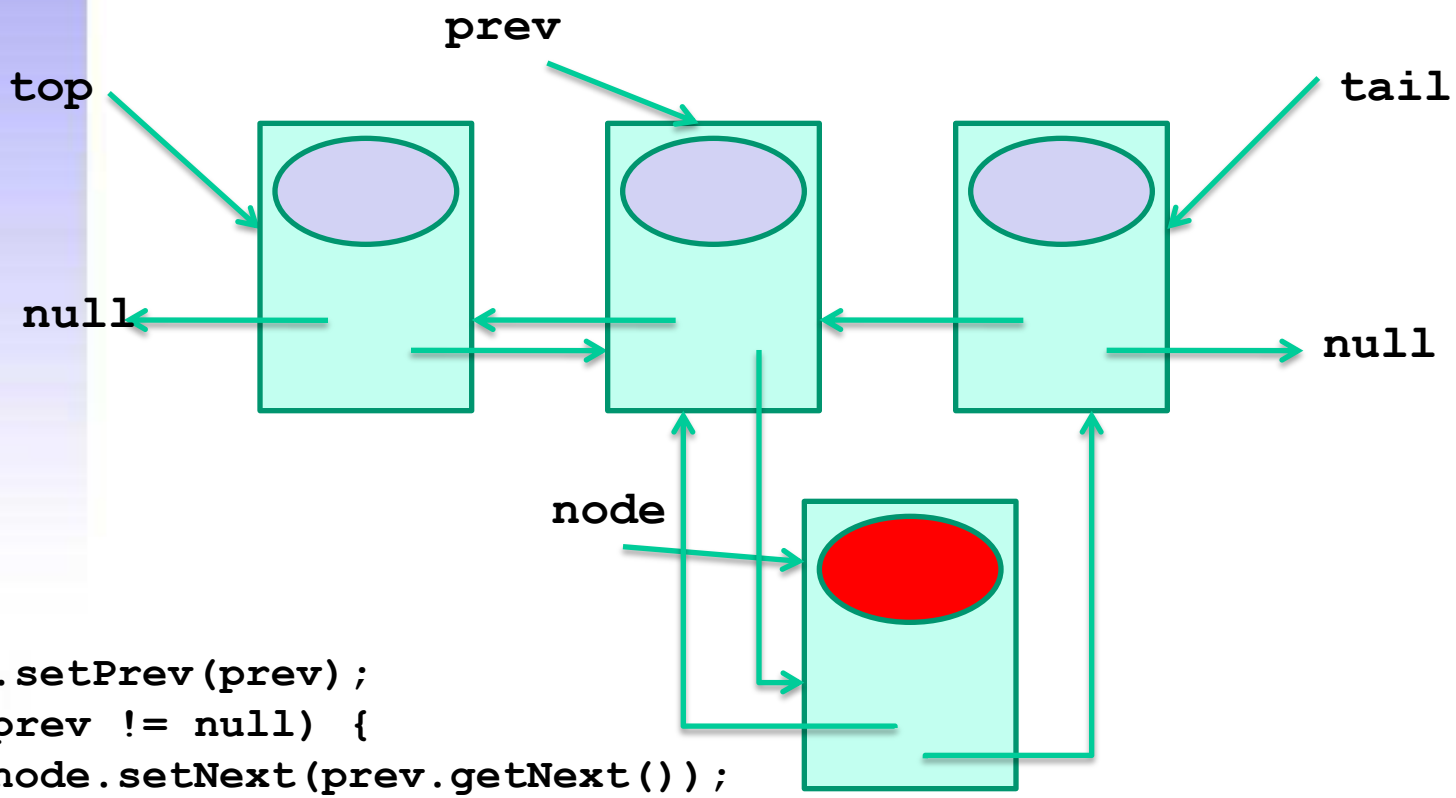
```
Public class DLNode<T> {
    private T info;
    private DLNode<T> next;
    private DLNode<T> prev;

    public DLNode(T info) {…}
    public DLNode(T info, DLNode prev, DLNode next) {…}

    public DLNode<T> getNext() {…}
    public void setNext(DLNode<T> next) {…}
    public DLNode<T> getPrev() {…}
    public void setPrev(DLNode<T> prev) {…}
    public T getInfo() {…}
    public void setInfo(T info) {…}
}
```

Inserting a node



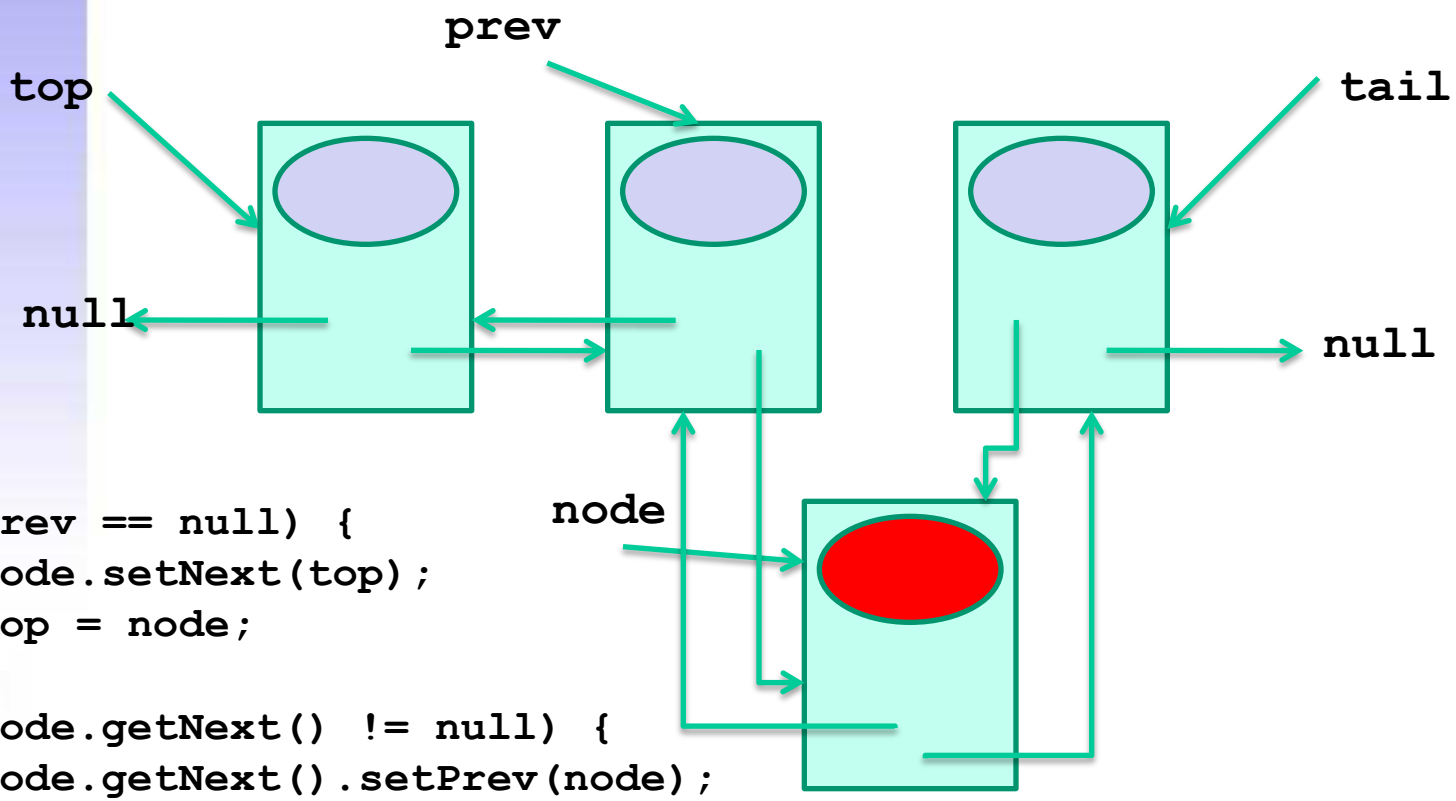

```
DLNode<T> node = new DLNode<T>(data);
```

Inserting a node




```
node.setPrev(prev);
if (prev != null) {
    node.setNext(prev.getNext());
    prev.setNext(node);
}
```

Inserting a node

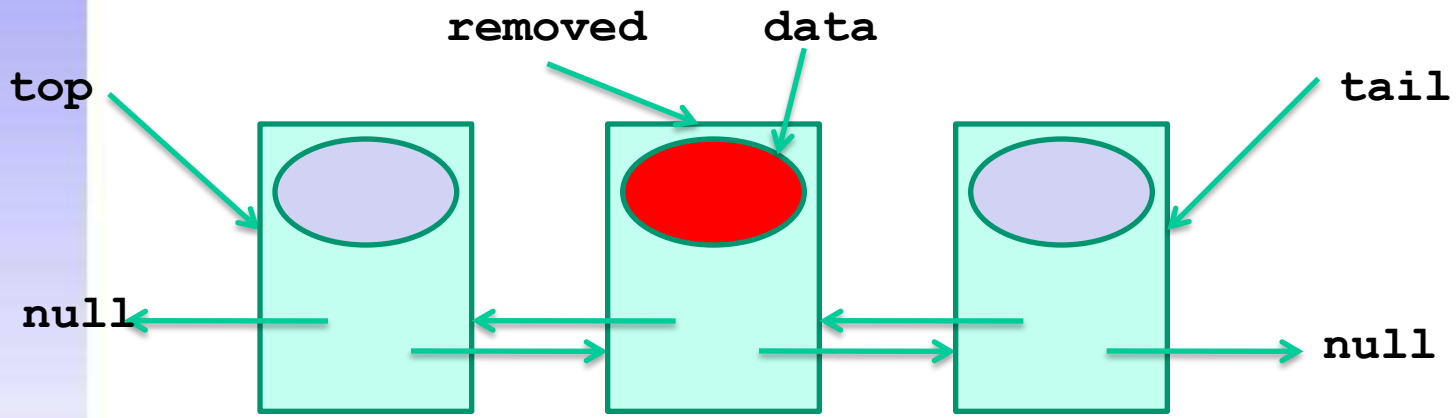



```
if (prev == null) {
    node.setNext(top);
    top = node;
}
if (node.getNext() != null) {
    node.getNext().setPrev(node);
} else {
    tail = node;
}
```

Inserting a node

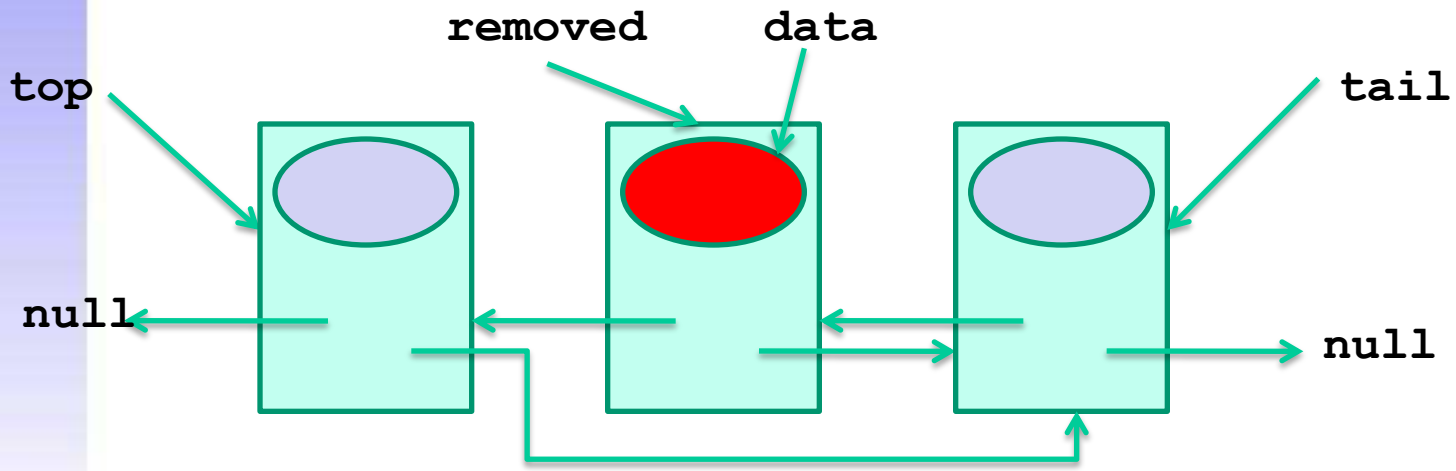
```
/**
  * Inserts 'data' after the 'prev' node. If 'prev'
  * is null, 'data' is inserted at the first position
  */
public void insert(DLNode prev, T data) {
    DLNode<T> node = new DLNode<T>(data);
    node.setPrev(prev);
    if (prev != null) {
        node.setNext(prev.getNext());
        prev.setNext(node);
    } else {
        node.setNext(top);
        top = node;
    }
    if (node.getNext() != null) {
        node.getNext().setPrev(node);
    } else {
        tail = node;
    }
}
```

Removing a node



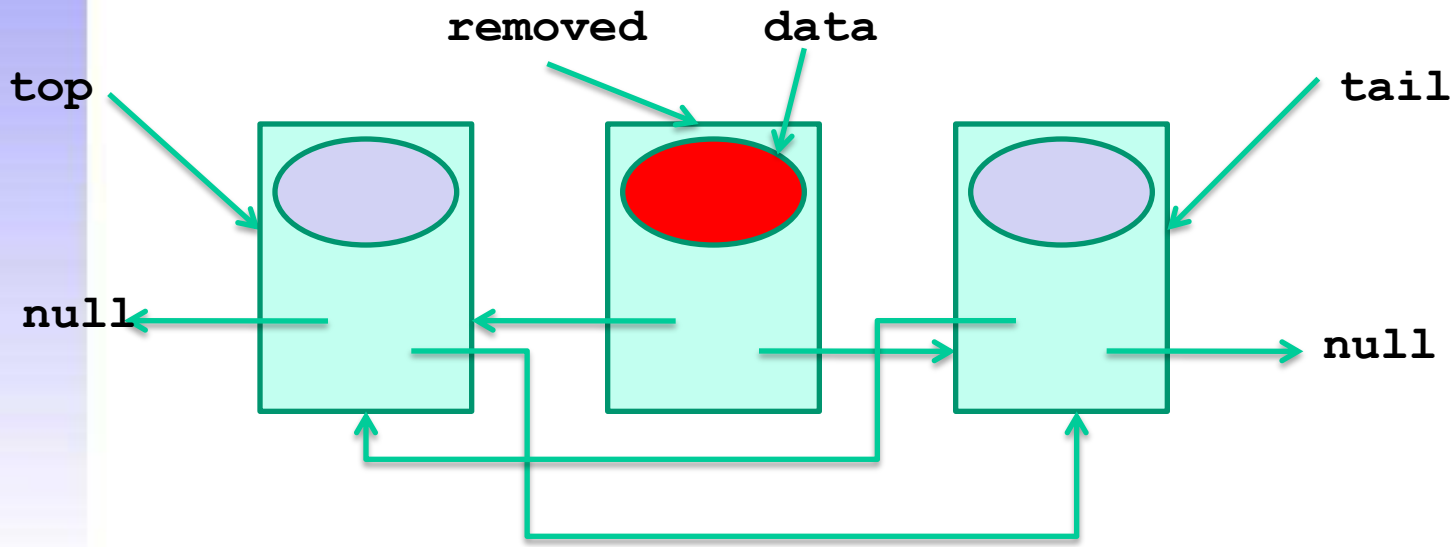

```
T data = removed.getInfo();
```

Removing a node



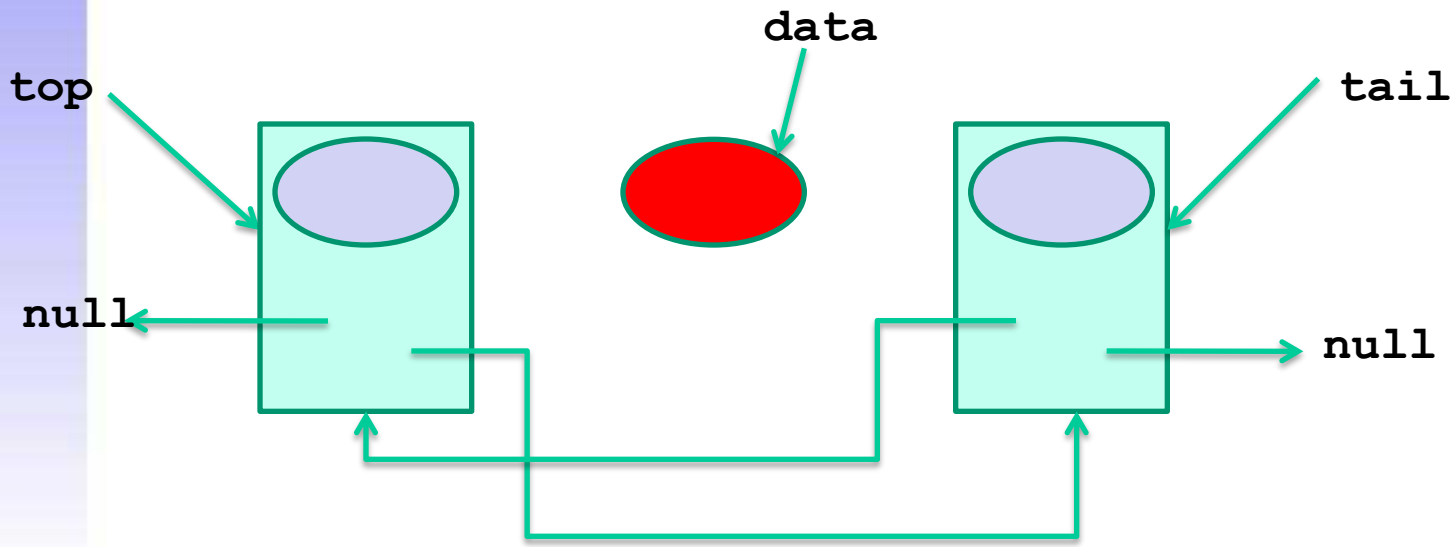

```
if (removed.getPrev() != null) {
    removed.getPrev().setNext(removed.getNext());
} else {
    top = removed.getNext();
}
```

Removing a node




```
if (removed.getNext() != null) {
    removed.getNext().setPrev(removed.getPrev());
} else {
    tail = removed.getPrev();
}
```

Removing a node



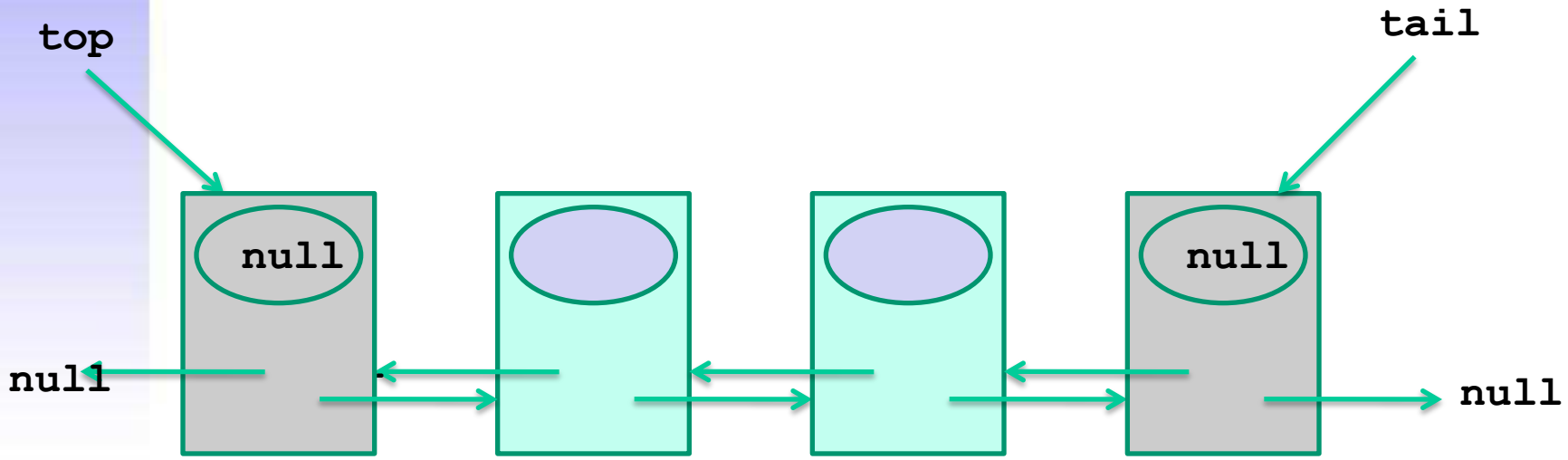
Removing a node

```
/**
  * Removes a node from the list and returns
  * the information it holds.
  */
public T remove(DLNode<T> removed) {
    T data = removed.getInfo();
    if (removed.getPrev() != null) {
        removed.getPrev().setNext(removed.getNext());
    } else {
        top = removed.getNext();
    }
    if (removed.getNext() != null) {
        removed.getNext().setPrev(removed.getPrev());
    } else {
        tail = removed.getPrev();
    }
    return data;
}
```

Alternate implementation

- Checking that the previous and next nodes are not null makes the previous implementation complex and error-prone
- Possible simplification:
  - Create two special (dummy) nodes, without associated info, so that one is always at the beginning of the list and the other one is always at the end:
    - An empty list contains only those two empty nodes
    - For insertions and removals, it is guaranteed that the previous and next nodes exists, so there is no need to check them
    - References `top` and `tail` do not need to be updated

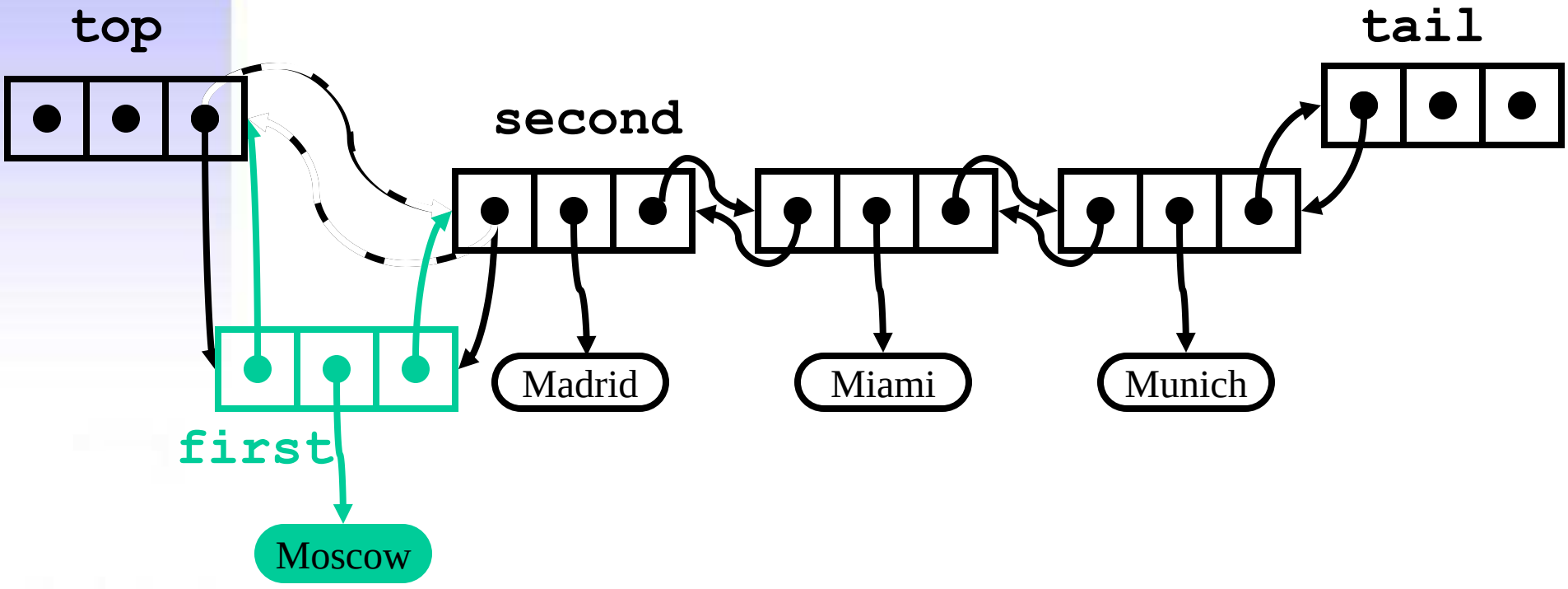
Alternate implementation



Implementation based on lists

```
public class DLDeque<T> implements Deque<T> {
  private DLNode<T>top, tail;
  private int size;
  public DLDeque() {
    top = new DLNode<T>();
    tail = new DLNode<T>();
    tail.setPrev(top);
    top.setNext(tail);
    size = 0;
  }
  …
```

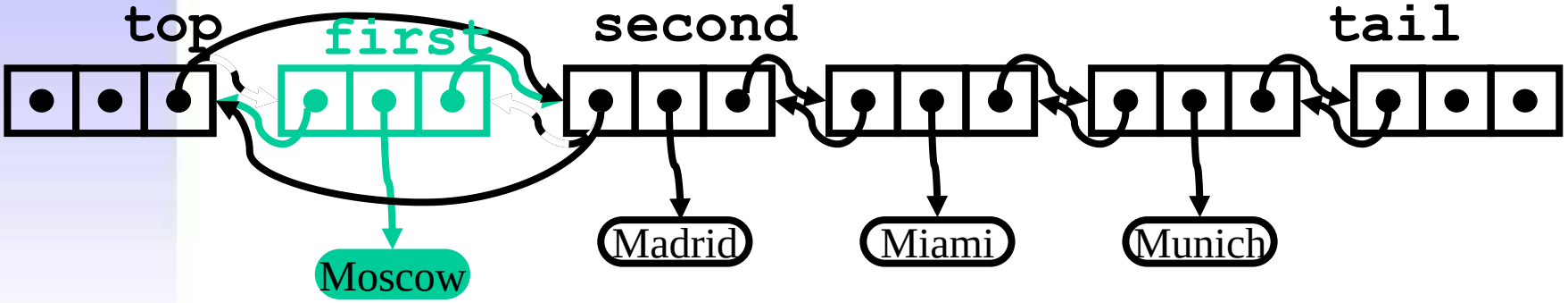
Insertion



Implementation based on lists

```
…
public void insertFirst(T info) {
   DLNode<T> second = top.getNext();
   DLNode<T> first = new DLNode<T>(info, top, second);
   second.setPrev(first);
   top.setNext(first);
   size++;
}
…
```

Extraction



Implementation based on lists

```
public T removeFirst()
   throws EmptyDequeException {
   if (top.getNext() == tail)
      throw new EmptyDequeException();
   DLNode<T> first = top.getNext();
   T info = first.getInfo();
   DLNode<T> second = first.getNext();
   top.setNext(second);
   second.setPrev(top);
   size--;
   return info;
}
```

Activity

- Review how “queues” are implemented in
  - **http://docs.oracle.com/javase/tutorial/collections/interfaces/queue.html**
  - **http://docs.oracle.com/javase/6/docs/api/java/util/Queue.html**

Method Summary	
boolean	add(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
E	element() Retrieves, but does not remove, the head of this queue.
boolean	offer(E e) Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
E	peek() Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
E	poll() Retrieves and removes the head of this queue, or returns null if this queue is empty.
E	remove() Retrieves and removes the head of this queue.

Methods inherited from interface java.util.Collection
addAll, clear, contains, containsAll, equals, hashCode, isEmpty, iterator, remove, removeAll, retainAll, size, toArray, toArray

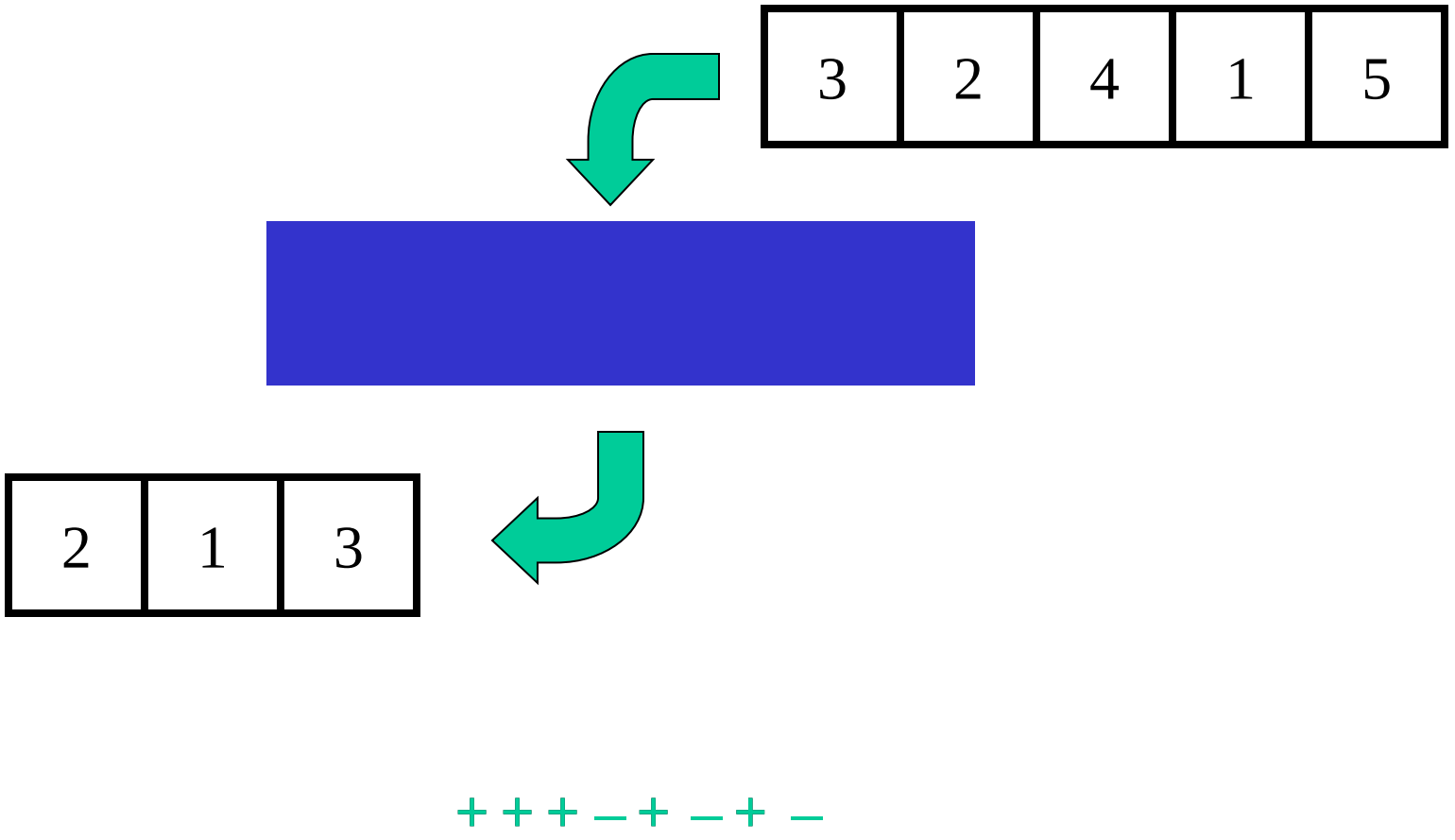
Priority queue

- A priority queue is a linear data structure where elements are retuned according to a value associated to them (priority) (and not necessarily to the order of insertion)
- The priority might be the value of the element itself, but it might also differ from it

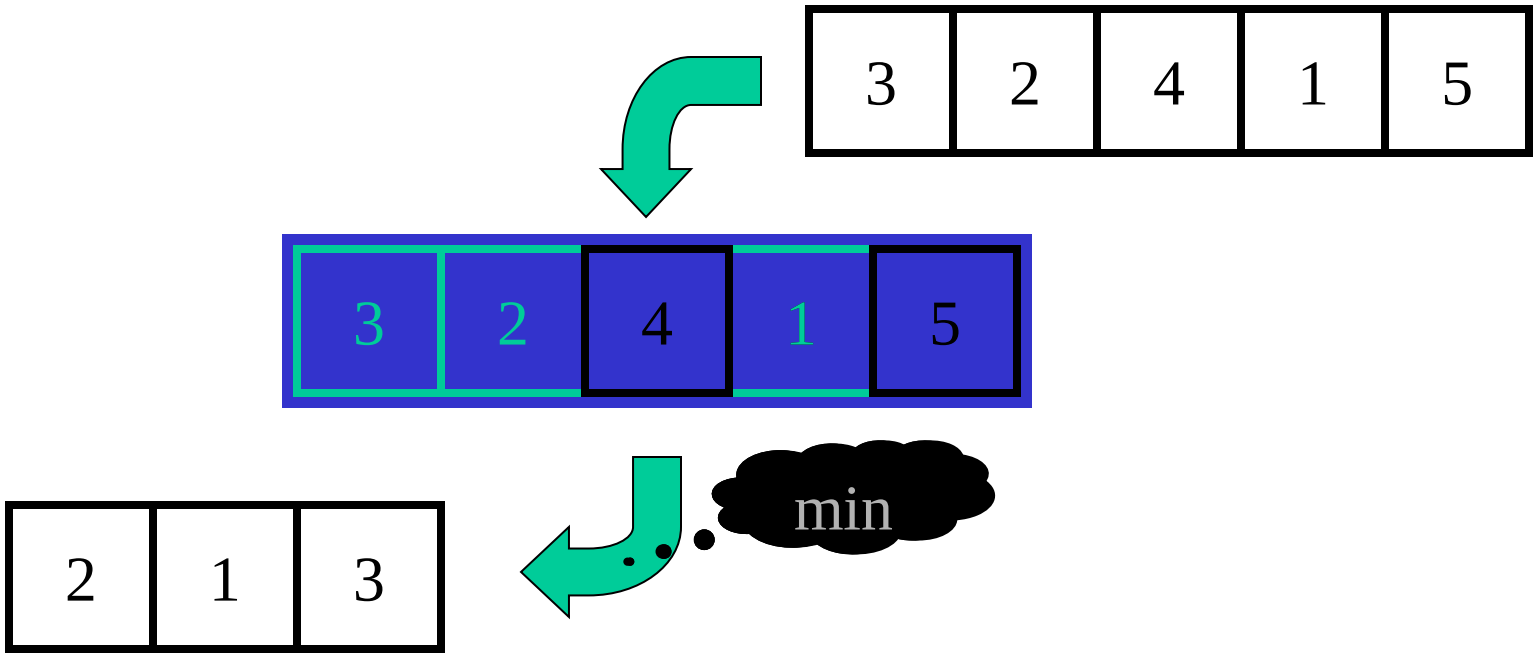
Interface for priority queues

```
public interface PriorityQueue<T> {
  public int size();
  public boolean isEmpty();
  public void insertItem(Comparable priority,
                         T info);
  public T minElem()
    throws EmptyPriorityQueueException;
  public T removeMinElem()
    throws EmptyPriorityQueueException;
  public T minKey()
    throws EmptyPriorityQueueException;
}
```

Example

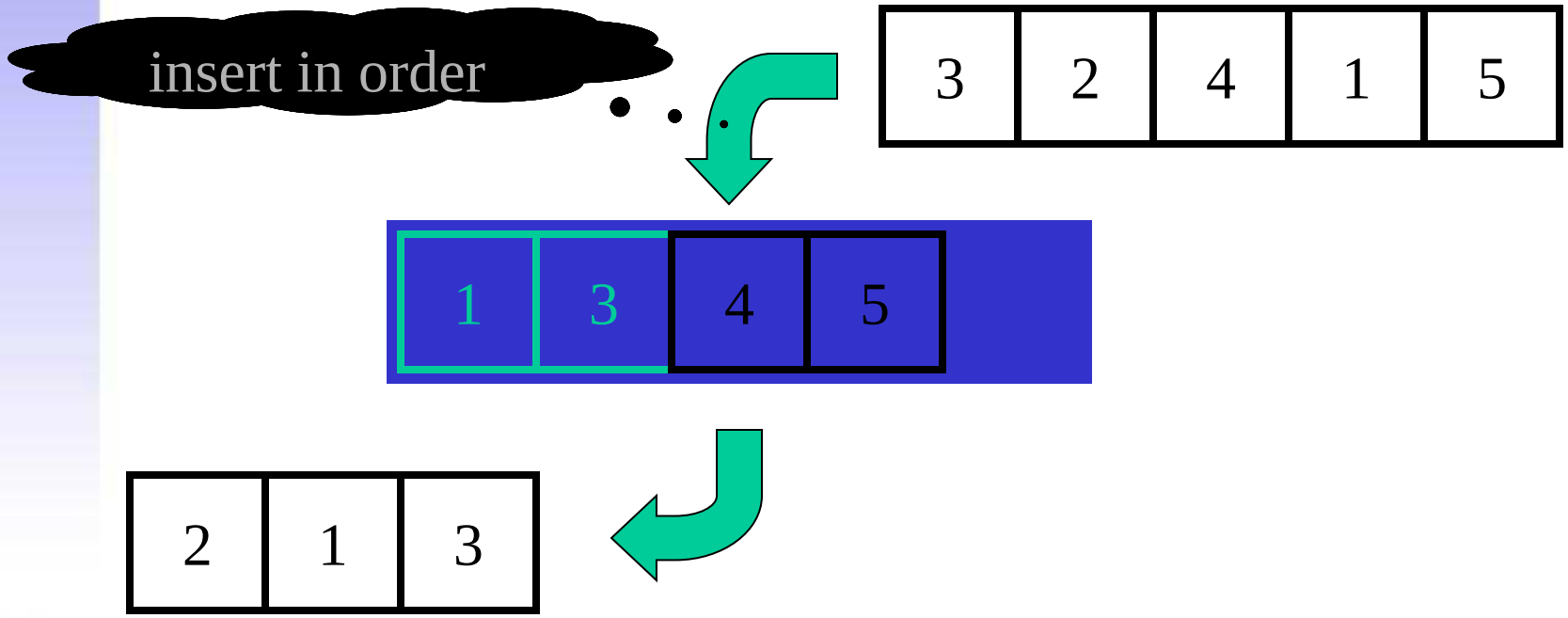


Example



+ + + – + – + –

Example



+ + + – + – + –

Implementations

- With an unsorted sequence
  - Easy insertion 
  - Comparison needed for extraction 
- With a sorted sequence
  - Comparison needed for insertion 
  - Easy extraction 

Activity

- Try
  http://www.akira.ruc.dk/~keld/algoritmik_e99/Applets/Chap11/PriorityQ/PriorityQ.html