SWEN 601
Software Construction

Abstract Data Types, Queues, & Stacks

Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
   a. *The project **<u>may</u>** already contain some code!*
2. Create a `session` package. This is where you will write your solutions to today's activities.
3. Create a `homework` package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

<u>Do not</u> submit code that ***does not compile***. Comment it out if necessary.

Next Two Weeks

WEEK 05	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ					Quiz #12		
LECTURE			**No Class (Fall Break)**		Abstract Data Types, Queues, & Stacks		
HOMEWORK					*Hwk 13 Assigned*	Hwk 12 Due (**11:30pm**)	

WEEK 04	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ					Quiz #13		
LECTURE			Practicum 2		hashCode & Hashtables		
HOMEWORK	Hwk 13 Due (**11:30pm**)				*Hwk 14 Assigned*		

Data Structures

- A *data structure* stores data in a particular way.
- Most data structures provide some common operations.
  - **store** - add or insert data into the data structure
  - **retrieve** - get data from the data structure
  - **remove** - remove data from the data structure
  - **size** - get the number of elements in the structure
- Different data structures handle data in different ways.
  - The programmer chooses the data structure that handles data in the most efficient way for the program.
  - Choosing the right data structure for the program is essential to writing efficient code.

Many data structures have real world analogs that make them easy and intuitive to understand.

Array

- An **array** is the simplest data structure.
- Arrays are a fixed size.
  - Arrays do not dynamically shrink and grow based on the number of items added or removed.
- Items in an array are strictly ordered.
- Arrays provide only two basic operations.
  - **set** - set the value of an item at a specific index.
  - **get** - retrieve the item at a specified index.
- In Java, arrays are a basic data type.
- Arrays do not store *objects*, but instead a reference to the object that is stored elsewhere in memory.

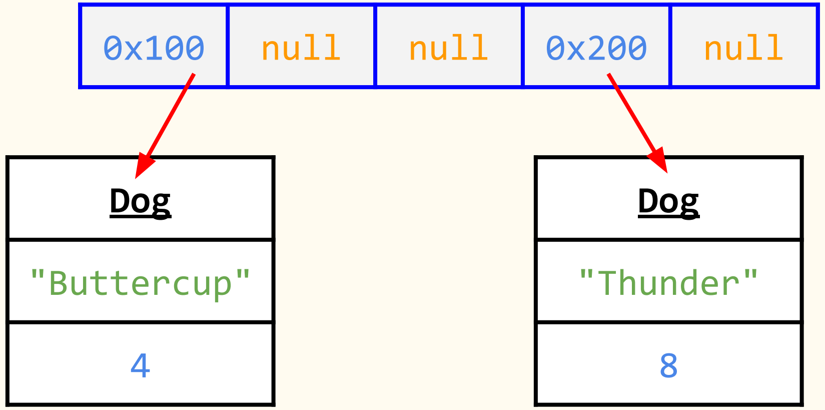
Array

```
public void arrayExample() {
    Dog[] dogs = new Dog[5];

    dogs[0] = new Dog( "Buttercup" );

    dogs[3] = new Dog( "Thunder" );

}
```

Arrays are a fixed length, so the values must be initialized to some default value when the array is created.

null	null	null	null	null

Items are added to specific indices.



Arrays

Q: Arrays are handy, but they have quite a few potential drawbacks. What are they?

(*hint, if you implemented the list homework correctly, you know all about them*)

A: Arrays are fixed size. If you don't know how many elements you need, you need to ***overallocate*** and then deal with the eventuality that your allocated space fills up.

A: If you run out of space, you need to make a new, bigger array and copy everything from the old small array into the new larger array.

A: Adding elements to or removing elements from the middle of the array means that you need to shift things around.

Q: So what are the alternatives?

Abstract Data Types

- An *abstract data type* (ADT) defines the behavior of a data structure from the point of view its user.
- The ADT defines the possible values and the operations available
- Even though an ADT defines specific, common operations for manipulating data, each ADT may be implemented in multiple ways.
- Choosing the right ADT isn't always enough. Choosing the correct *implementation* is also important.
    - Different implementations have different strengths and weaknesses.

Many data structures have real world analogs that make them easy and intuitive to understand.

Interfaces

- An Abstract Data Type (ADT) defines *what* the data type can do, but not *how* it does it.
    - It ***defines*** the behavior that the ADT must have.
    - It does not ***implement*** the behavior.

Q: What does this sound like?

A: An `interface`!

- A Java `interface` defines the (abstract) methods that an implementation must have, but does not provide an implementation.
- An ADT is an `interface`.

Why Interfaces?

- An `interface` defines (but does not implement) the behavior that all implementations must have.
- This allows programmers to *program to the interface* and write code without needing to know the messy details of the implementation.
- As we will soon see, many Abstract Data Types have more than one implementation.
- *Programming to the interface* means that you don't need to know about the implementation and can therefore work with *any* implementation.
    - Polymorphism!

Generics

- We would like our data structures to work with ***any*** type, not just ints and strings.
- There is a Java syntax using which an interface or a class may declare a ***type parameter***.
  - A type parameter is a ***fake type*** name that is a placeholder for a ***real type***.
  - It is specified after the class/interface name in angle brackets, e.g. `MyClass<T>`
- Once declared, the type parameter may be used in code as though it is a real type, e.g.
  - Return values: `public T getThing()`
  - Parameters: `void setThing(T thing)`
- When the class/interface is used as a variable type, a real type is ***substituted*** for the type parameter in the declaration.
  - `private MyClass<String> stringy;`

```
public class MyGeneric<T> {
  private T generic;

  public MyGeneric(T generic) {
    this.generic = generic;
  }

  public T getGeneric() {
    return generic;
  }
}
```

A real type is substituted for a type parameter when an instance of the generic is created. That instance will then only work with values compatible with the specified type.

```
MyGeneric<String> mg =
        new MyGeneric<>("Buttercup");
```

Generifying a Class

```
public class StringHolder {
  private String held;

  public StringHolder(String held) {
    this.held = held;
  }

  public String getHeld() {
    return held;
  }

  public void setHeld(String held) {
    this.held = held;
  }
}
```

```
public class AnythingHolder<T> {
  private T held;

  public AnythingHolder(T held) {
    this.held = held;
  }

  public T getHeld() {
    return held;
  }

  public void setHeld(T held) {
    this.held = held;
  }
}
```

Using Generics

- Java's generics are a ***compile time*** language.
  - The compiler insures that any types used with an instance of a generic class match the type specified when the class is created.
  - If you try to use a type that is not compatible with the declared type, your code will not compile.
- At run time, any generic type parameter is treated as an instance of `java.lang.Object`.
- But in your code, you don't need to cast back and forth between the type you want (e.g. `String`) and `Object`.

The type on the right side of an assignment can be ***inferred*** from the type declared on the left, and so the diamond operator (<>) can be used.

```
AnythingHolder<String> holder =
                  new AnythingHolder<>();
```

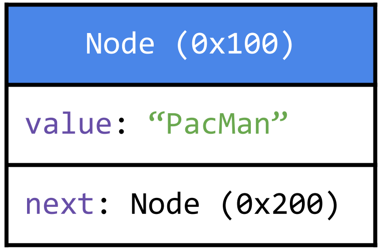
Trying to use a type that is not compatible with the type parameter causes a compiler error.

```
holder.setHeld(1234);
```

But different instances of the same class can be used with different types.

```
AnythingHolder<String> s =
                     new AnythingHolder<>();
AnythingHolder<Integer> i =
                     new AnythingHolder<>();
```

Activity: A Generic Node



We will be implementing our first data structures using ***linked nodes***. Write a ***generic*** Node class that meets the following requirements.

- It should declare a ***type parameter***.
- It should hold a generic **value** of some kind.
  - Write an ***accessor*** and a ***mutator***.
- It should hold a reference to the **next** Node (which is of the same generic type).
  - Write an ***accessor*** and a ***mutator***.
- It should provide at least one constructor.

Many data structures have real world analogs that make them easy and intuitive to understand.

You are washing dishes and stacking clean plates on the counter.

The **first** plate you clean is on the bottom of the stack. The **last** plate you clean is on the top.

The stack of plates is placed at a buffet table.

The last plate added is the first plate removed (from the top).

The first plate added is the last plate removed (from the bottom).

Stack

- A **Stack** is an ADT that stores items *in reverse order*.
- A Stack is usually meant to hold items temporarily for processing.
- A Stack provides several operations typical for an ADT:
    - **push** - add an item to the top of the stack.
    - **peek** - return but do not remove the newest item in the stack.
    - **pop** - remove and return the *newest* item in the stack.
    - **size** - return the number of elements in the stack.
- The order of the items is maintained.
- Let's define a stack interface!

A Stack stores items *in reverse order*.

Stacks are a LIFO data structure ("life-oh").

LIFO means "Last In, First Out." Popped items are returned in the opposite order that they were pushed.

Stacks are appropriate to use when items should be processed in reverse order, e.g. the call stack, backtracking.

Activity: A Stack Interface

Write an interface to represent a Stack data structure. Remember that an ***abstract data type*** defines the operations from the perspective of the user, but does not implement them.

- There should at least be one method for each operation:
  - **Push** an element.
  - **Pop** the newest element.
  - **Peek** at the newest element.
  - Get the **size** of the stack.
- Make sure that your interface is ***generic***.

Stack Usage

```
public void stackExample(Stack s) {

    stack.push( 123 );
    stack.push( 456 );
    stack.push( 789 );

    int num = stack.peek();

    while(s.size() > 0) {
      int next = s.pop();
      System.out.println(next);
    }
}
```

The method to the right accepts any Stack as a parameter, and then performs a few basic operations on it.

Q: What value is returned here?

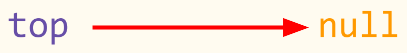
A: The most recently pushed item: `789`.

Q: In what order do we expect the numbers to be printed in the `while` loop?

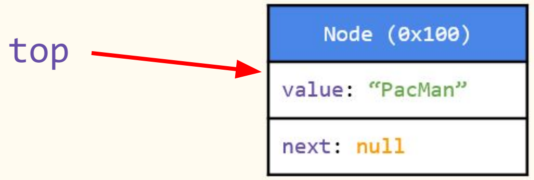
A: The reverse order in which they were pushed: `789`, `456`, `123`

A Node-Based Stack

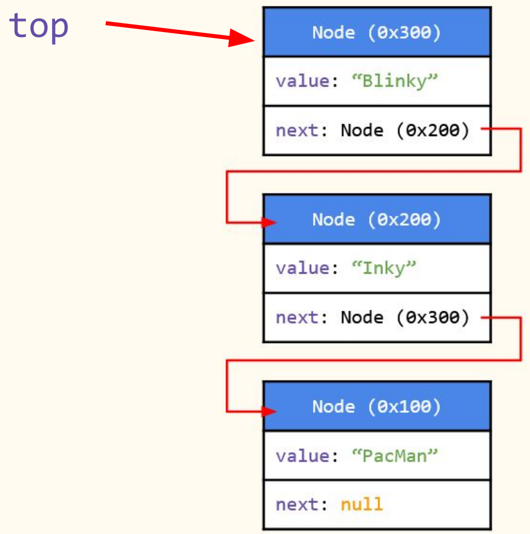
A stack must be able to add ***and*** remove elements at the ***top***. To start, it will be `null`.



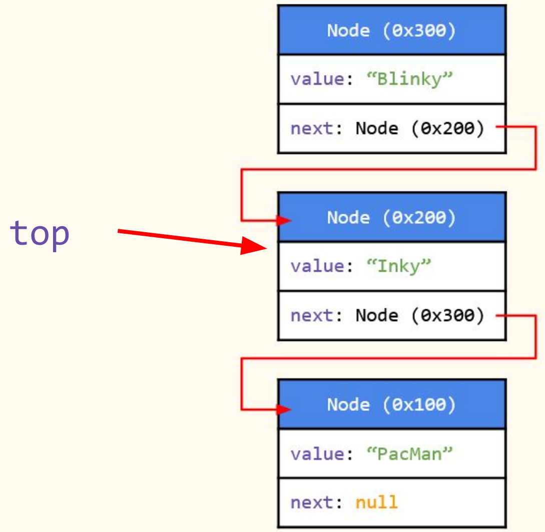
The first time a value is ***pushed***, a new node is created to be the **top** of the stack.



Each time a new value is ***pushed***, its node becomes the new ***top***.

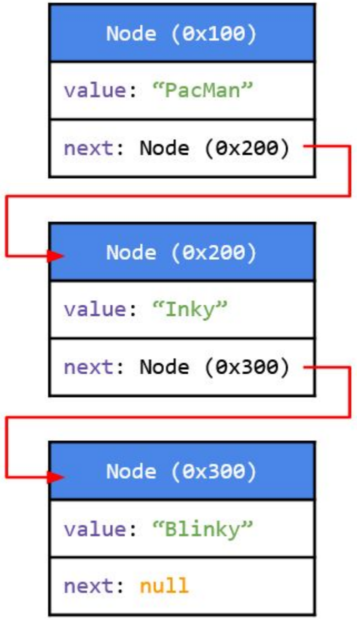


When a value is ***popped*** the ***top*** moves to the ***current*** top's next node and the value is returned.



Activity: A Stack Implementation

Now it's time to create an implementation of your generic `Stack` interface.



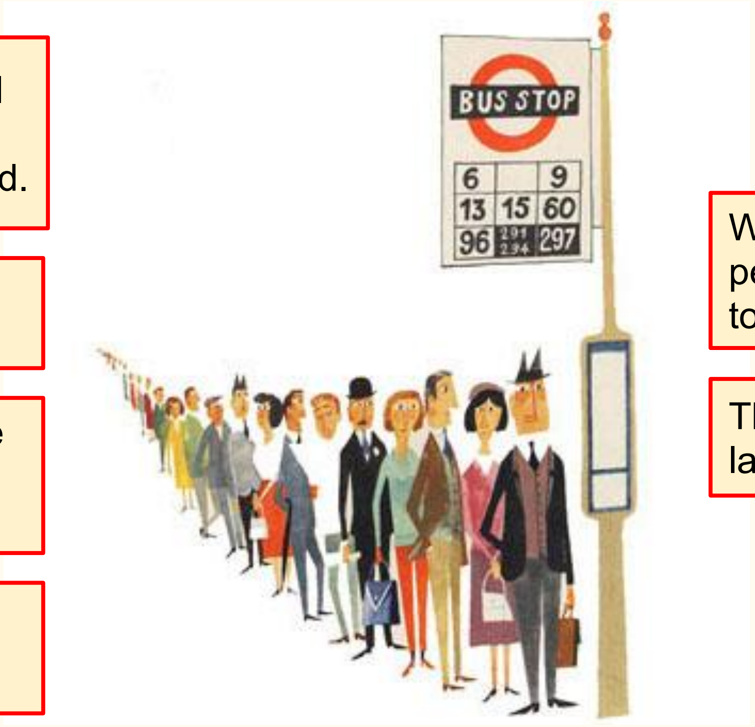
- Your class should be called `NodeStack` and it should be ***generic***.
  - The ***type parameter*** should be declared with the class, e.g. `NodeStack<T>`.
  - The same parameter should be used when implementing the `Stack` interface.
- It will need a reference to the **top** of the stack.
- To start, implement the **size** method.
- Next implement **peek**.
- Then implement **push**.
- Finally, implement **pop**.

Many data structures have real world analogs that make them easy and intuitive to understand.

There is a line of passengers waiting for the bus.

The first person to arrive at the bus top is the first person in line.

The last person to arrive is the last in line.



When the bus arrives, the first person in line is the first person to get on.

The last person in line is the last person to get on.

Queue

- A **Queue** is an ADT that, like a list, stores data *in the order that the items are added.*
- Unlike some other data structures, a Queue is usually meant to hold items temporarily for processing.
- A Queue provides at least the following operations:
    - **enqueue** - add an item to the end of the queue.
    - **peek** - return but do not remove the oldest item in the queue.
    - **dequeue** - remove and return the oldest item in the queue.
    - **size** - the number of elements currently in the queue.
- The order of the items is maintained.
- Let's define a Queue interface...

A Queue stores items *in a specific, predictable order*.

Queues are a FIFO data structure ("figh-foe").

FIFO means "First In, First Out." Popped items are returned in the same order that they were pushed.

Queues are appropriate to use when items should be processed in the order they are created, e.g. logs.

Activity: A Queue Interface

Write an interface to represent a Queue data structure. Remember that an ***abstract data type*** defines the operations from the perspective of the user, but does not implement them.

- There should at least be one method for each operation:
  - **Enqueue** an element.
  - **Dequeue** the oldest element.
  - **Peek** at the oldest element.
  - Get the **size** of the queue.
- Make sure that your interface is ***generic***.

Queue Usage

```
public void queueExample(Queue q) {

    q.enqueue( 123 );
    q.enqueue( 456 );
    q.enqueue( 789 );

    int num = q.peek();

    while(q.size() > 0) {
      int next = q.dequeue();
      System.out.println(next);
    }
}
```

The method to the right accepts any Queue as a parameter, and then performs a few basic operations on it.

Q: What value is returned here?

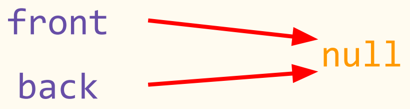
A: The oldest item still in the queue: `123`.

Q: In what order do we expect the numbers to be printed in the `while` loop?

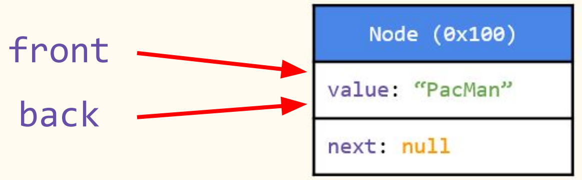
A: The same order in which they were enqueued: `123`, `456`, `789`

A Node-Based Queue

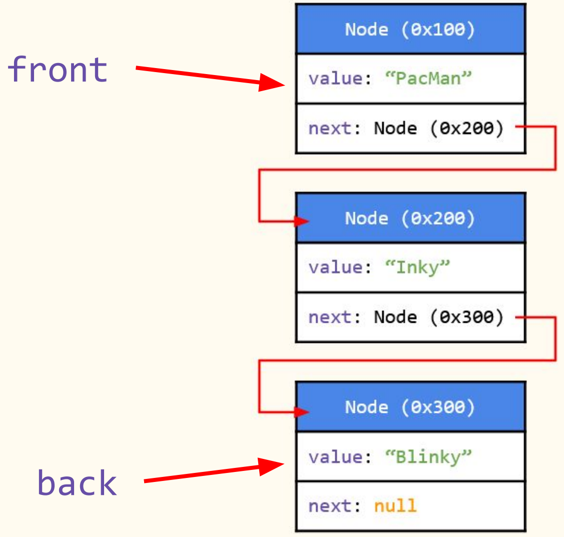
A queue must be able to add elements to the ***back*** and remove elements from the ***front***, and so will need fields for both. To start, they will be `null`.



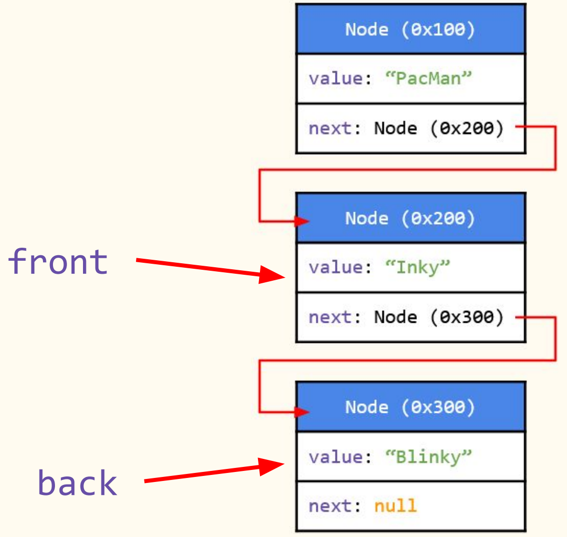
The first time a value is ***enqueued***, a new Node is created. ***Both*** the front and the back will refer to this new node.



Each time a new value is ***enqueued***, its node becomes the new ***back***.

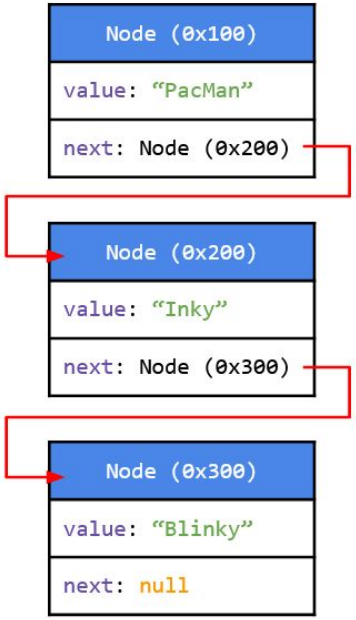


When a value is ***dequeued*** the ***front*** moves to the ***current*** front's next node and the value is returned.

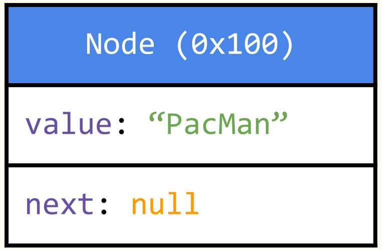


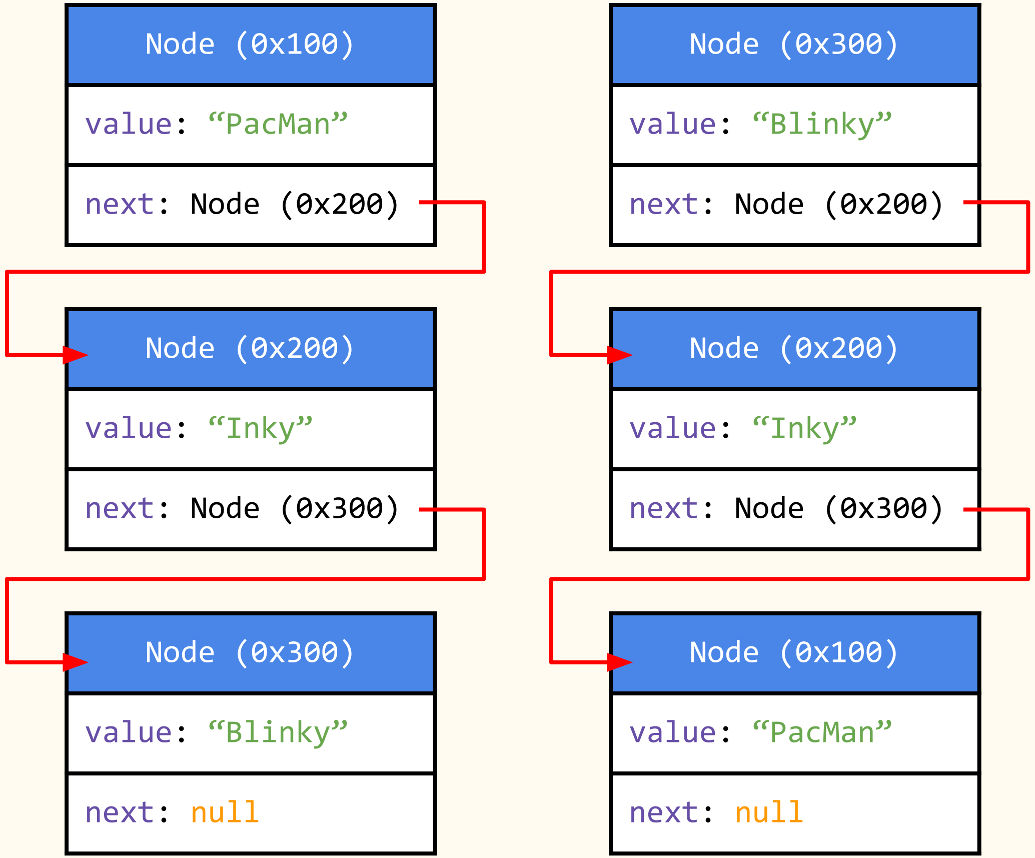
Activity: A Queue Implementation

Now it's time to create an implementation of your generic Queue interface.



- Your class should be called `NodeQueue` and it should be ***generic***.
  - The ***type parameter*** should be declared with the class, e.g. `NodeQueue<T>`.
  - The same parameter should be used when implementing the `Queue` interface.
- It will need a reference to both the **front** and the **back** of the queue.
- To start, implement the **size** method.
- Next implement **peek**.
- Then implement **enqueue**.
- Finally, implement **dequeue**.


Node (0x100)
value: “PacMan”
next: null


Node (0x100)
value: “PacMan”
next: Node (0x200)
Node (0x200)
value: “Inky”
next: Node (0x300)
Node (0x300)
value: “Blinky”
next: null
Node (0x300)
value: “Blinky”
next: Node (0x200)
Node (0x200)
value: “Inky”
next: Node (0x300)
Node (0x100)
value: “PacMan”
next: null