CS 5004: Lecture 6

Northeastern University, Spring 2021

At the start of every lecture

1. Pull the latest code from the lecture-code repo
2. Open the Evening_lectures folder
3. Copy this week's folder somewhere else
   - So you can edit it without causing GitHub conflicts
4. Open the code:
   1. Find the build.gradle file in the folder called LectureX
   2. Double click it to open the project

Agenda

- Polymorphism recap
- Recursive data structures
- Recursive linked list
- Stack implementation using a recursive linked list
- Immutable stack using a recursive linked list

Subtype polymorphism recap

Polymorphism definition

The ability of one **object** to be viewed/used as different **types**.

- Object = an *instance* of a *class* (i.e. a variable)

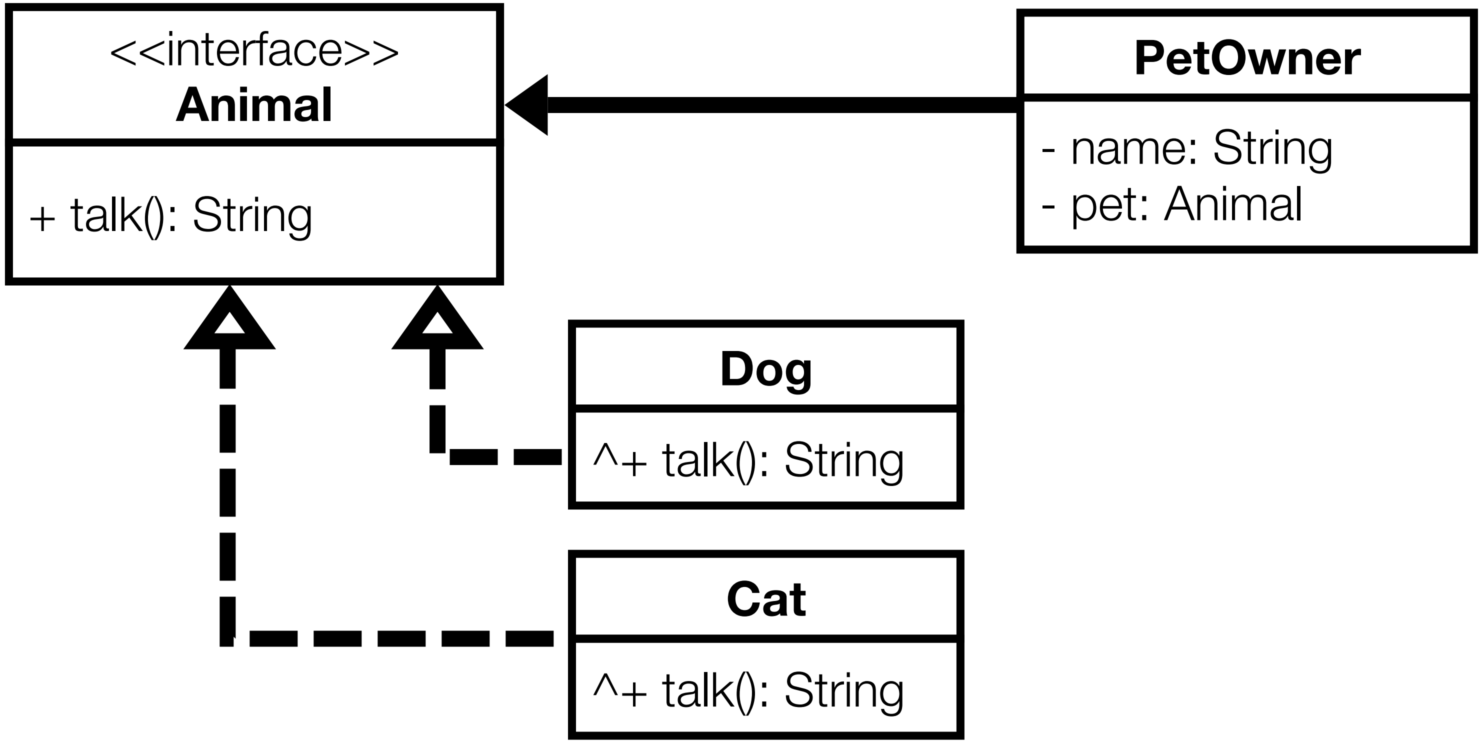
- Type = a *data type*
  - A **`class`** name
  - An **`abstract class`** name
  - An **`interface`** name

Subtype polymorphism

Made possible by **inheritance**.

- Every object will have multiple types
- An object is an **`instanceof`** its runtime type
- An object is an **`instanceof`** every type its runtime type inherits from

Subtype polymorphism




```
Cat cat; Dog dog;

dog instanceof Dog
cat instanceof Cat
dog instanceof Animal
cat instanceof Animal
```

* Lecture 4

Subtype polymorphism

```
public PetOwner(String name, Animal pet) {
  this.name = name;
  this.pet = pet;
}
```

We can do this because

`Cat` is a **subtype** of `Animal`

```
PetOwner owner = new PetOwner("Darth Vader", new Cat("Mittens"));

owner.getPet().talk();
```

Subtype polymorphism

```
public PetOwner(String name, Animal pet) {
  this.name = name;
  this.pet = pet;
}

PetOwner owner = new PetOwner(“Darth Vader”, new Cat(“Mittens”));

owner.getPet().talk();
```

An example of **dynamic dispatch**.

- Won’t know which implementation of talk() until runtime.

Subtype polymorphism

Equals method takes **any Object** as the parameter.

- All Java classes inherit Object therefore, all are `instanceof Object`

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Node node = (Node) o;
    return Objects.equals(getItem(), node.getItem()) &&
           Objects.equals(getNextNode(), node.getNextNode());
}
```

Subtype polymorphism

While an object is being viewed as a base/super class, **can't access subclass functionality**.

- **Cast** to get access to that functionality

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Node node = (Node) o;
    return Objects.equals(getItem(), node.getItem()) &&
            Objects.equals(getNextNode(), node.getNextNode());
}
```

Subtype polymorphism

Without the cast:

- compile time error
- class Object has no methods getItem or getNextNode.

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Node node = (Node) o;
    return Objects.equals(getItem(), node.getItem()) &&
           Objects.equals(getNextNode(), node.getNextNode());
}
```

Recursive data structures

Review: Recursion

- An operation defined in terms of itself.
- Solving a problem recursively means solving smaller occurrences of the same problem.
- **Recursive programming** – functions/methods/objects that call themselves to solve some problem.

Review: Recursive algorithms

Every recursive algorithm consists of:

- **Base case** – *at least one* occurrence of the problem that can be solved directly - the simplest case(s).
- **Recursive case** – more complex occurrences that can't be solved directly but can be *described in terms of smaller occurrences of the same problem*.

Recursion – (Python) example from 5001

```python
def gcd(num1, num2):
  min_num = min(num1, num2)
  max_num = max(num1, num2)
  if max_num % min_num == 0:
    return min_num
  else:
    return gcd(min_num, max_num % min_num)
```

Recursion – (Python) example from 5001

```
def gcd(num1, num2):
  min_num = min(num1, num2)
  max_num = max(num1, num2)
  if max_num % min_num == 0:
    return min_num
  else:
    return gcd(min_num, max_num % min_num)
```

Base case – smallest subproblem

Recursion – (Python) example from 5001

```
def gcd(num1, num2):
  min_num = min(num1, num2)
  max_num = max(num1, num2)
  if max_num % min_num == 0:
    return min_num
  else:
    return gcd(min_num, max_num % min_num)
```

Recursive case – call self with next smallest problem

Recursive data structures

A data structure partially composed of smaller or simpler instances of the same data structure.

Just like recursive functions, recursive structures have:

- Base case
- Recursive cases

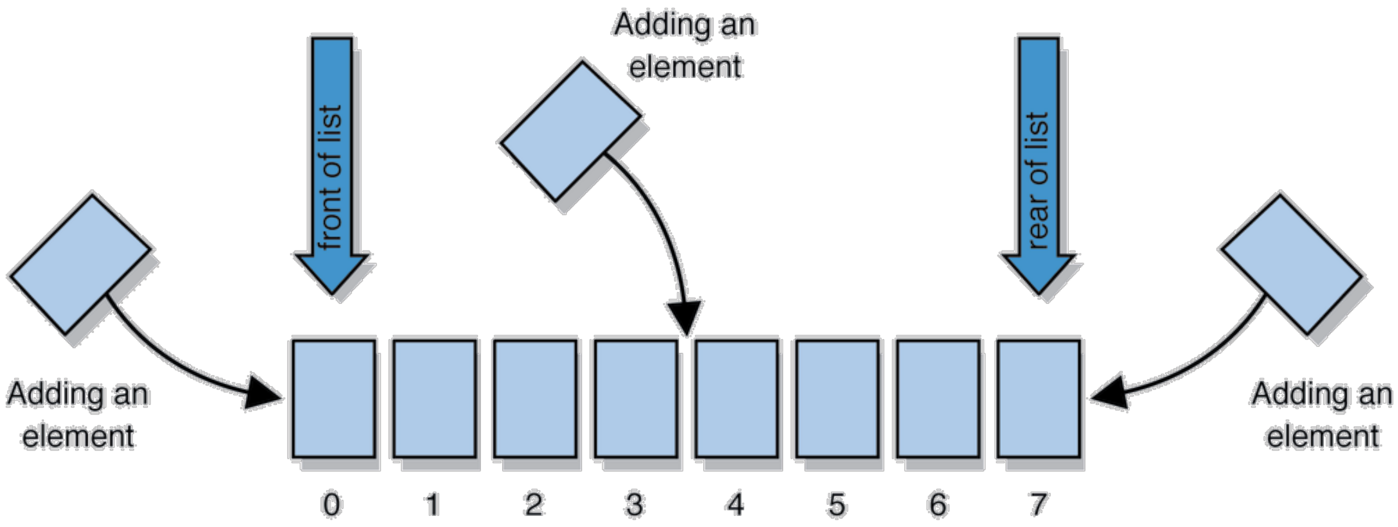
Recursive Linked List

List ADT and a linked list are not the same...

A List ADT may be implemented *using* a linked list

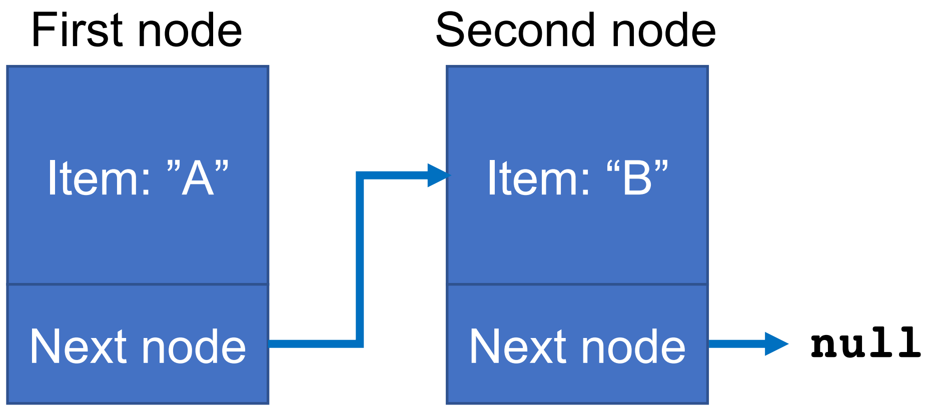
List ADT

Order



Linked List

Sequential* version



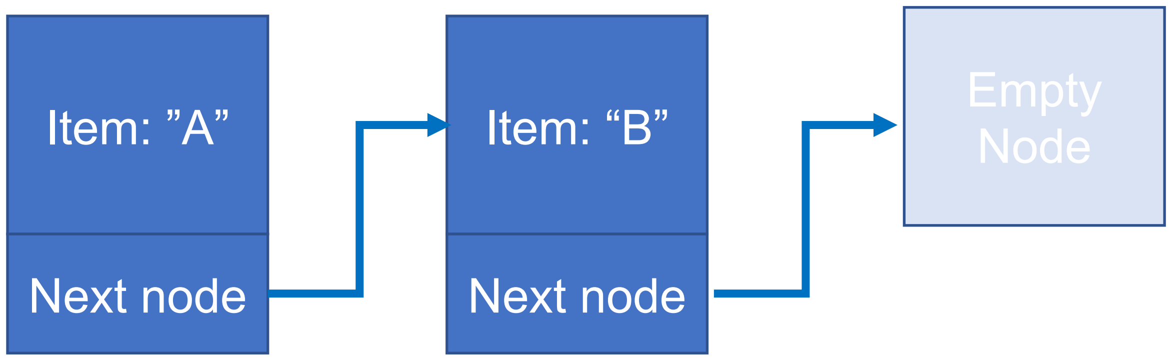

```java
public class Node {
  private DataType item;
  private Node next;

  public Node(DataType item, Node next)
{
    this.item = item;
    this.nextNode = nextNode;
  }
  // getters, setters, etc
}
```

*Linked list is always a recursive structure but methods may/may not use recursion

Linked List

Recursive version



Replace null at end of list with special type of Node

- Sometimes known as a *Cons* list

Recursive data structure

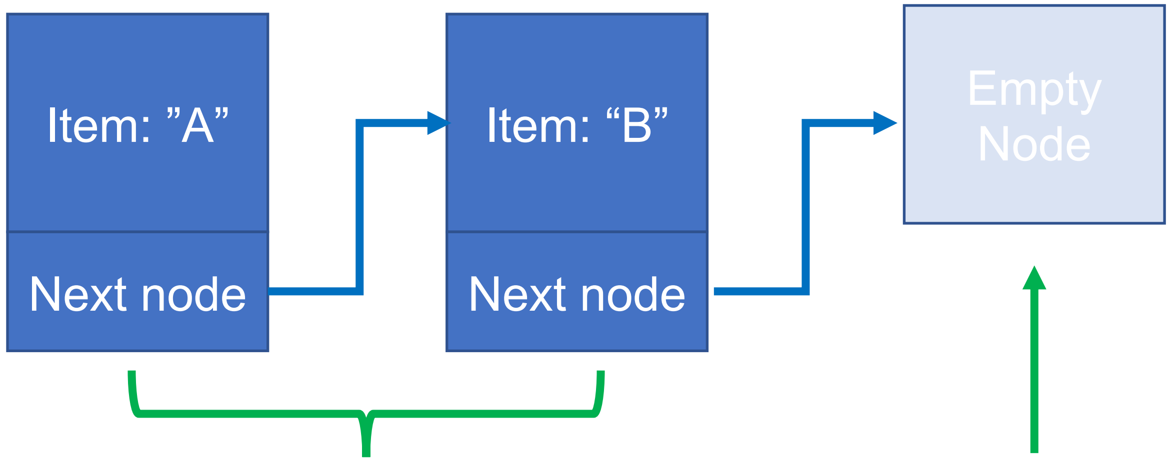
A data structure partially composed of smaller or simpler instances of the same data structure.

Just like recursive functions, recursive structures have:

- Base case
- Recursive cases

Linked List

Recursive version

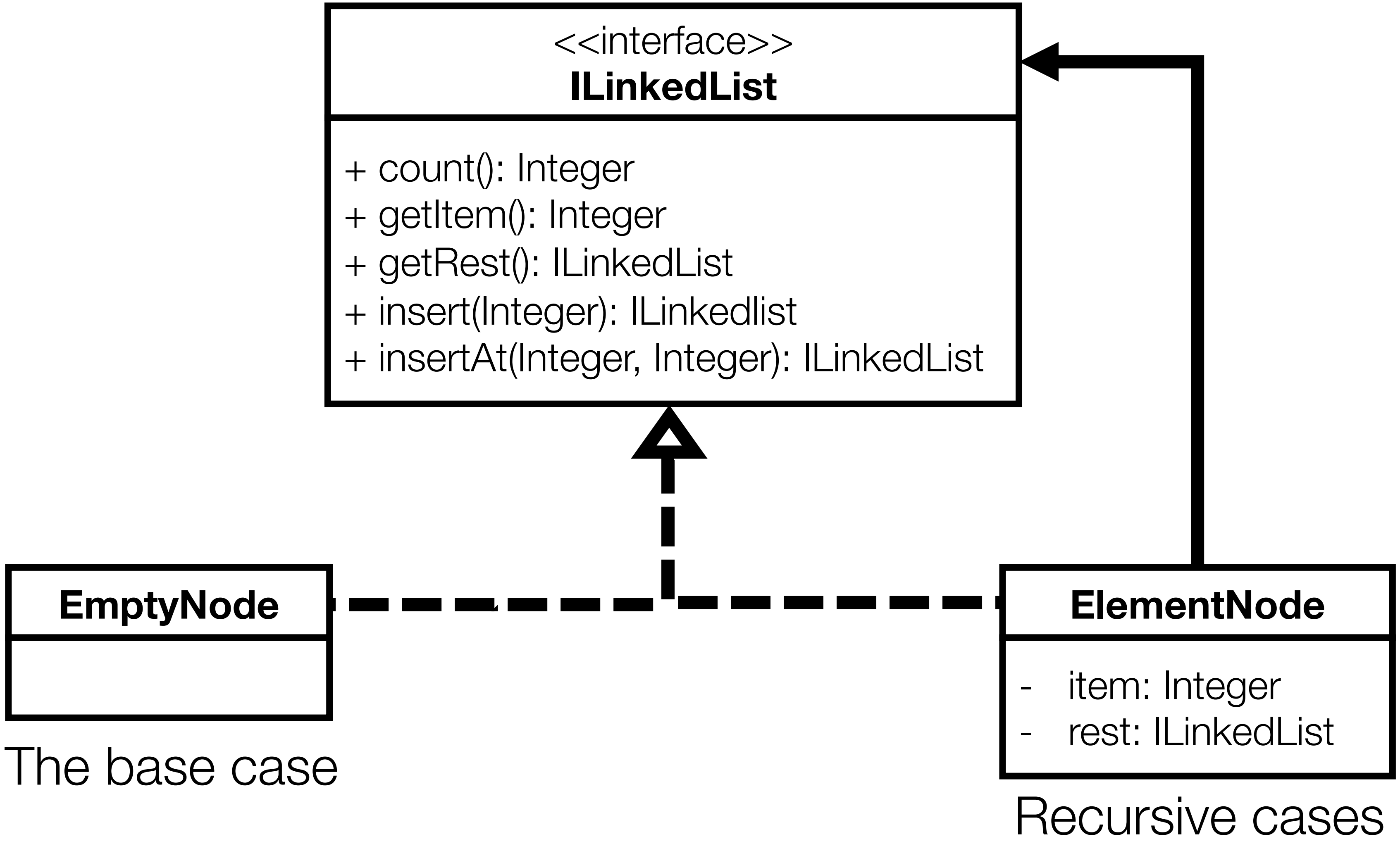


Replace null at end of list with special type of Node

Recursive case
Non-empty list

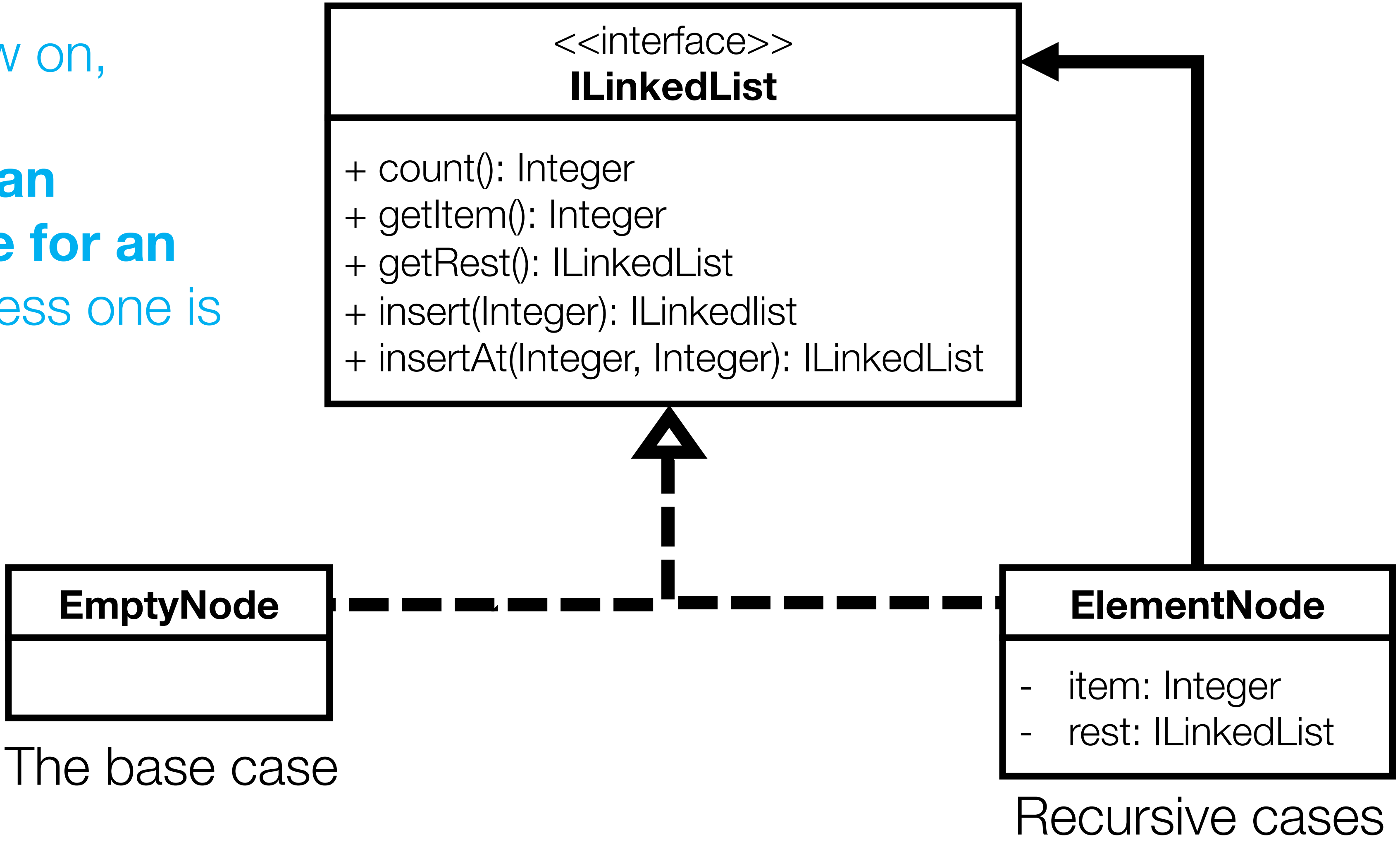
Base case
An empty list

Recursive linked list implementation



Recursive linked list implementation

From now on, **always provide an interface for an ADT** (unless one is provided)

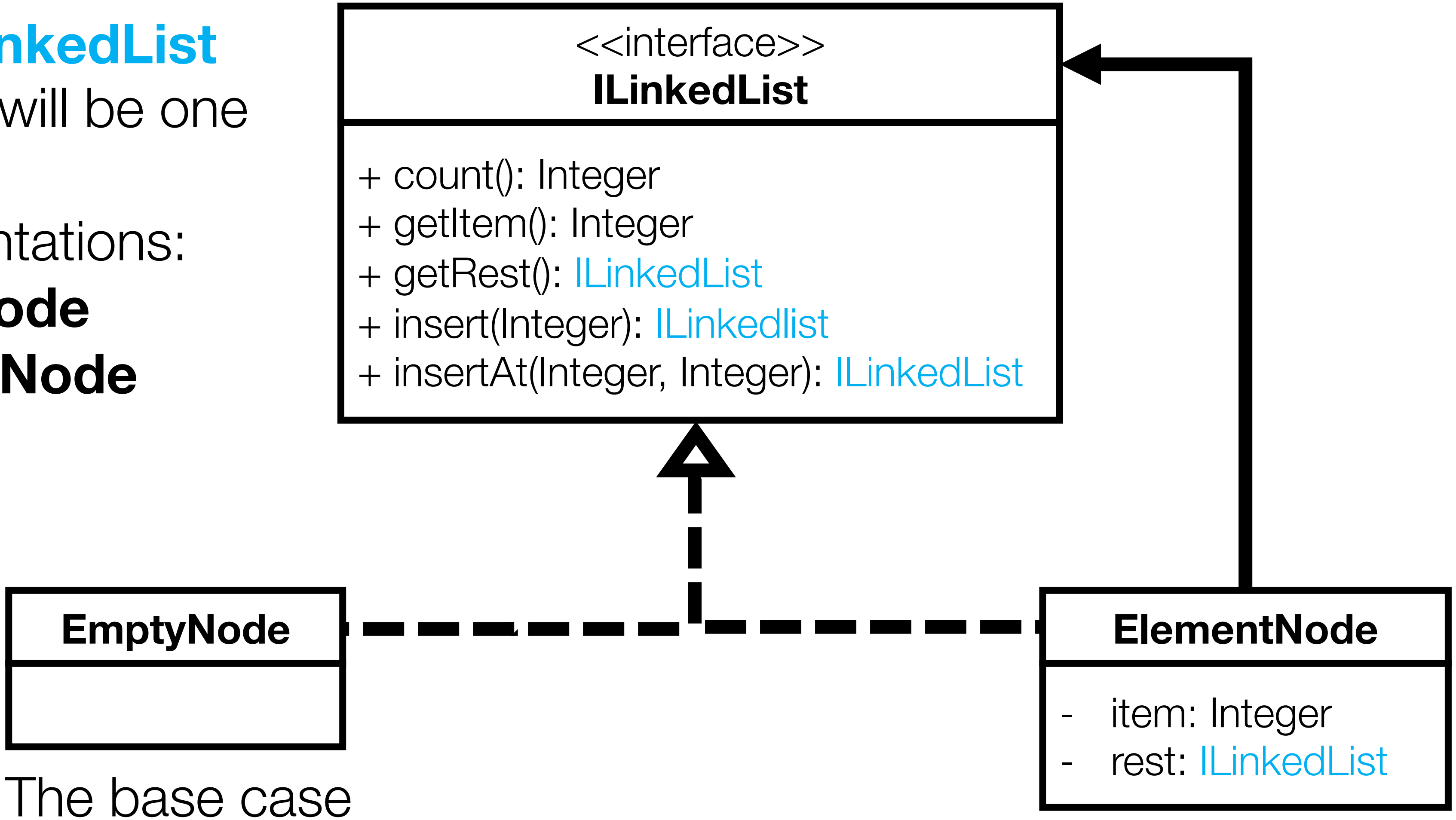


Note the subtype polymorphism

Every **ILinkedList** returned will be one of two implementations:

EmptyNode

ElementNode



The base case

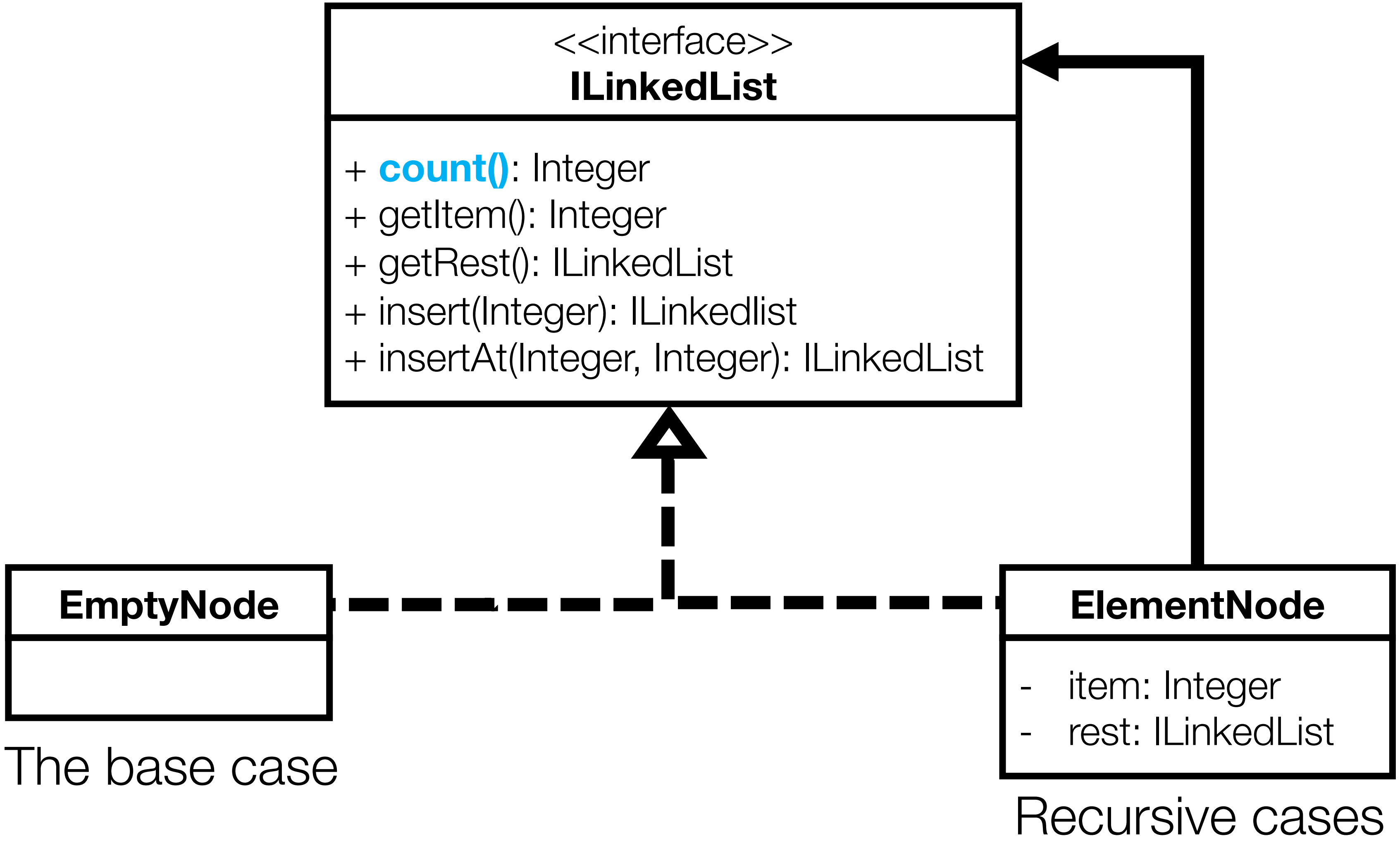
Recursive cases

Walkthrough

Today's sample code > linkedlist

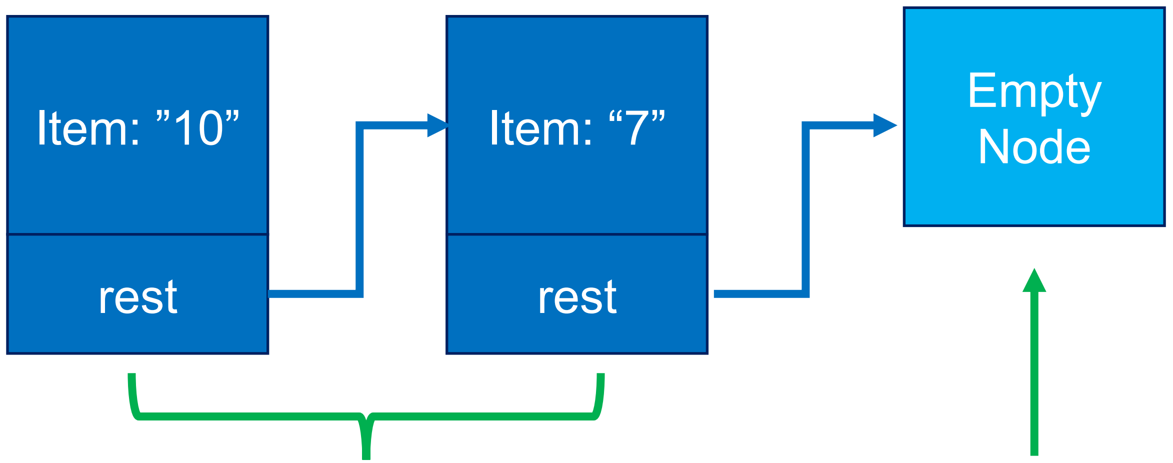
- The interface
- The test class
- The node classes
    - FYI: ElementNode can be "Cons", EmptyNode can be "Empty"

Recursive linked list implementation



Recursive linked list: `count()`

Where to start?



Recursive case

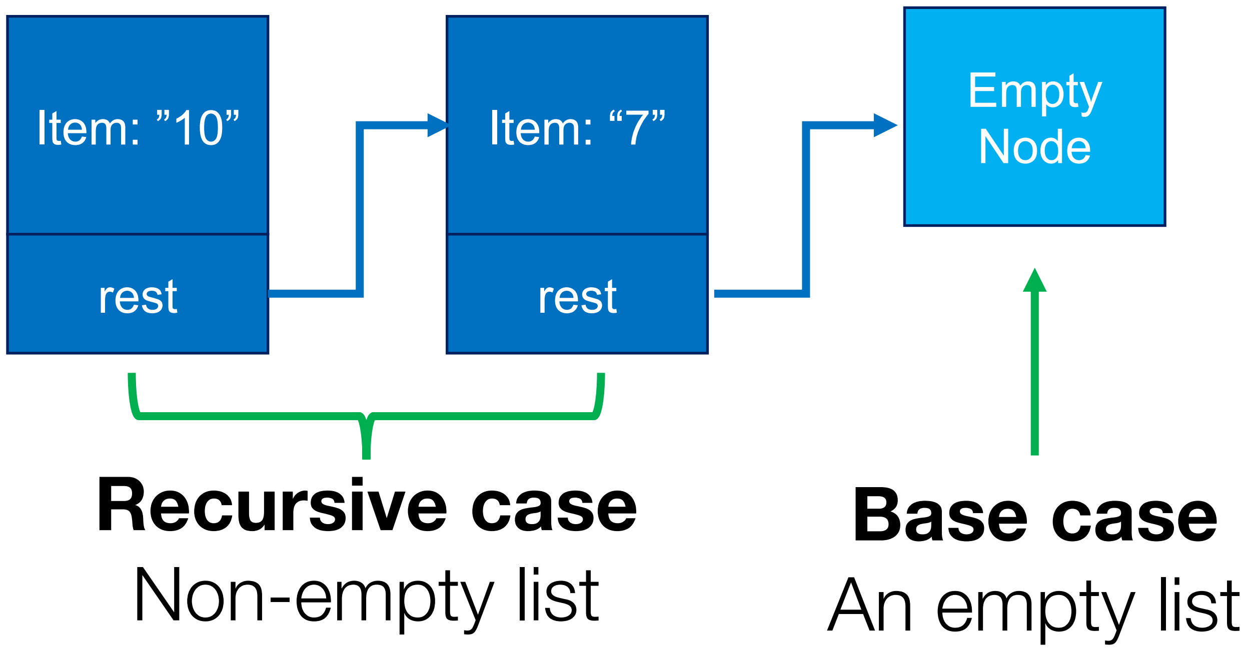
Non-empty list

Base case

An empty list

Recursive linked list: `count()`

Where to start?

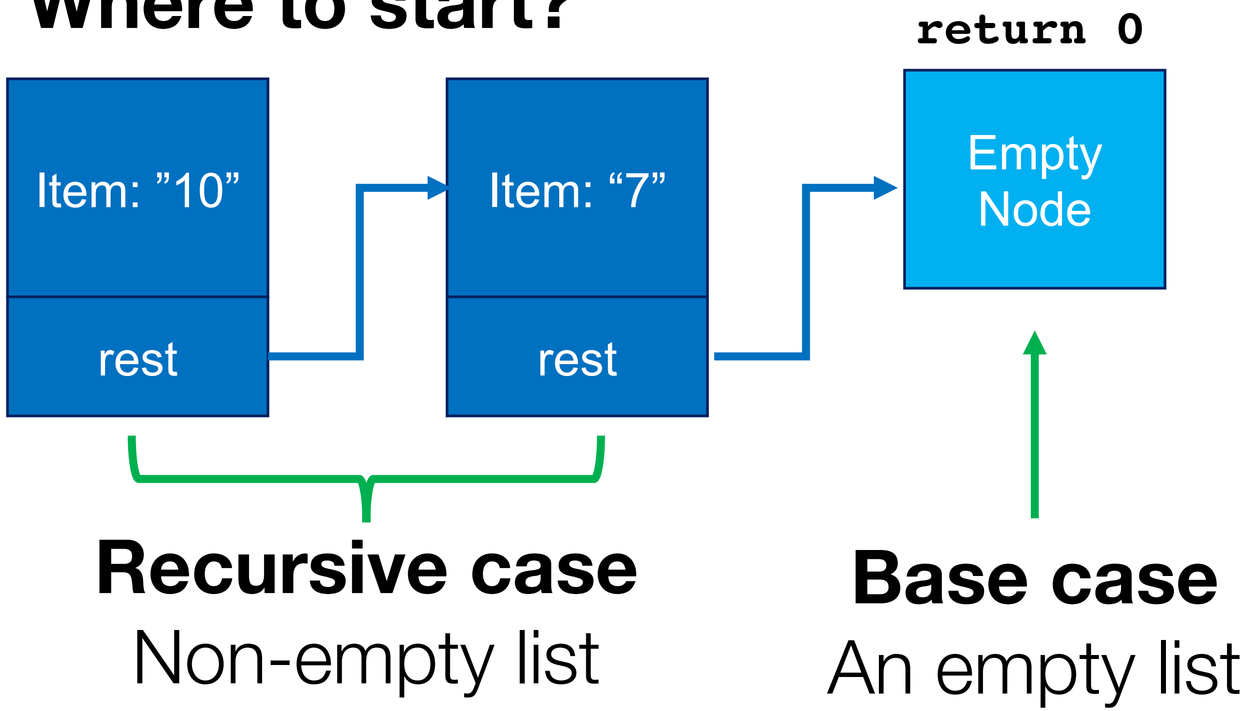


Just like a recursive function, **start with the base case**

- What should **`count()`** do if the list is empty?

Recursive linked list: `count()`

Where to start?

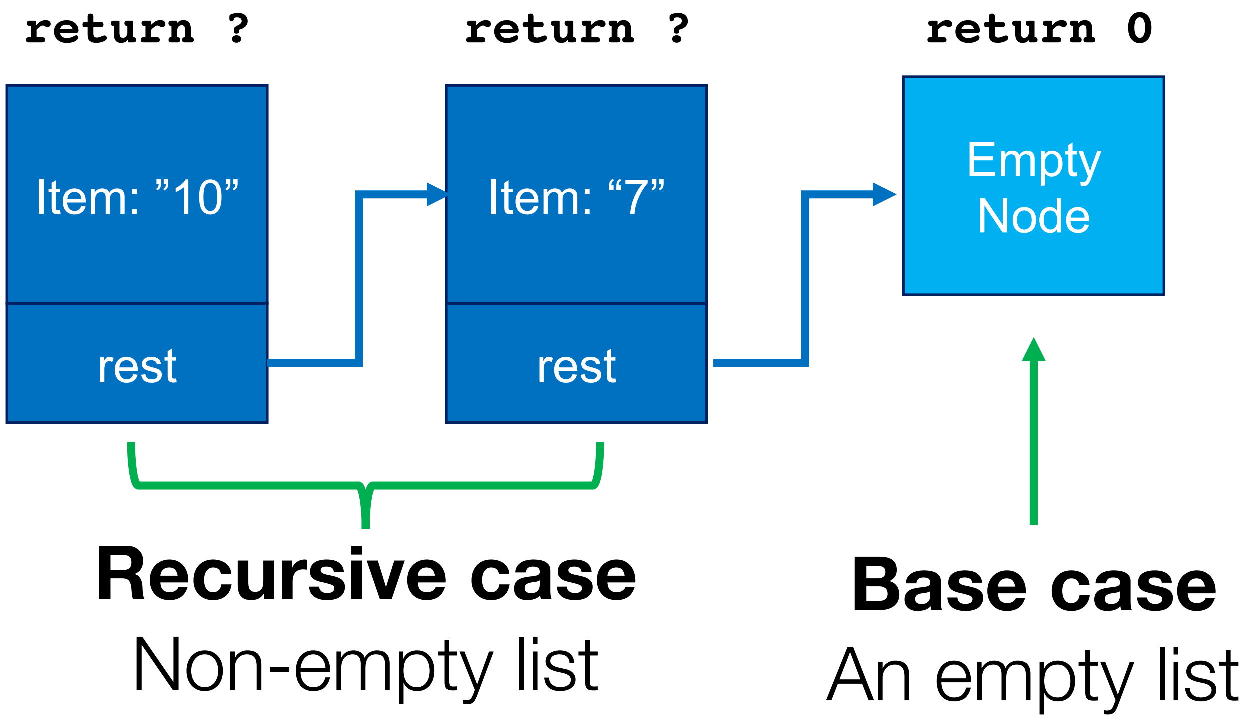


Just like a recursive function, **start with the base case**

- What should **`count()`** do if the list is empty?
  - An empty list has no items
  - → return 0

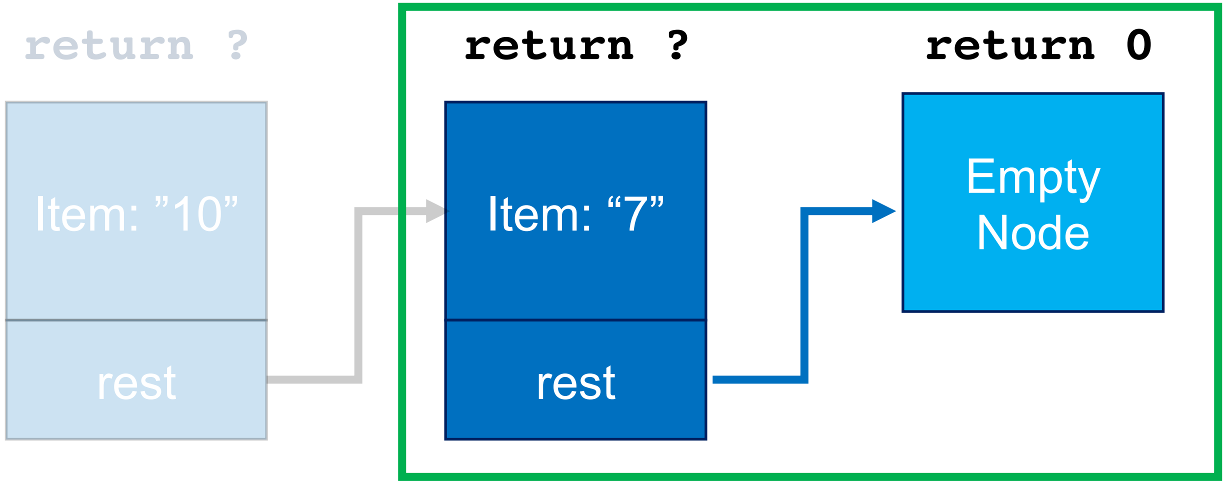
Recursive linked list: `count()`

What about the recursive case?



Recursive linked list: `count()`

What about the recursive case?



Think about the next simplest case, a list of 1.

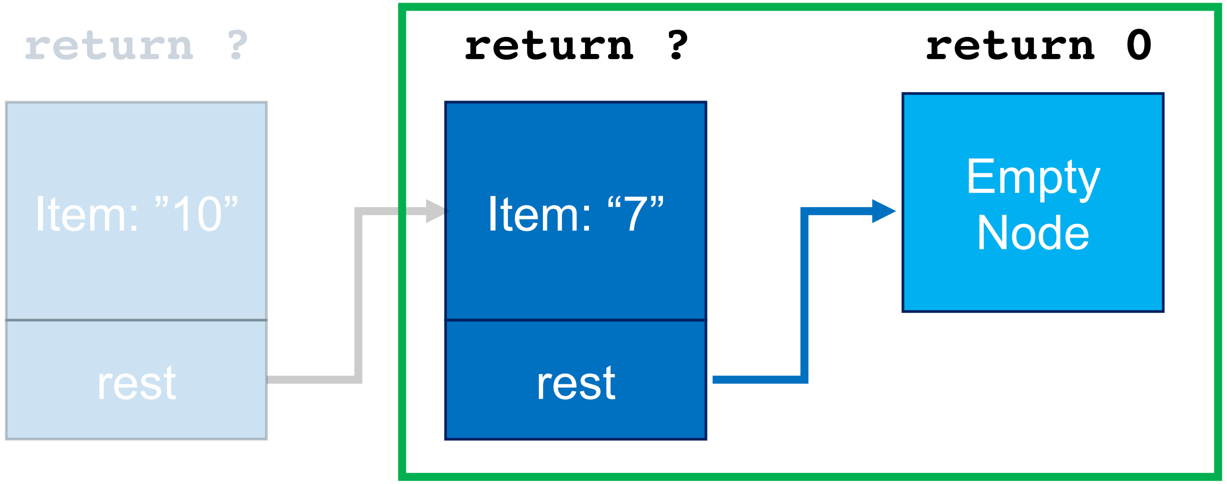
What we know:

- **`this.rest.count()`** is 0

The size of the list is 1 + the size of the rest of the list

Recursive linked list: `count()`

What about the recursive case?



Think about the next simplest case, a list of 1.

What we know:

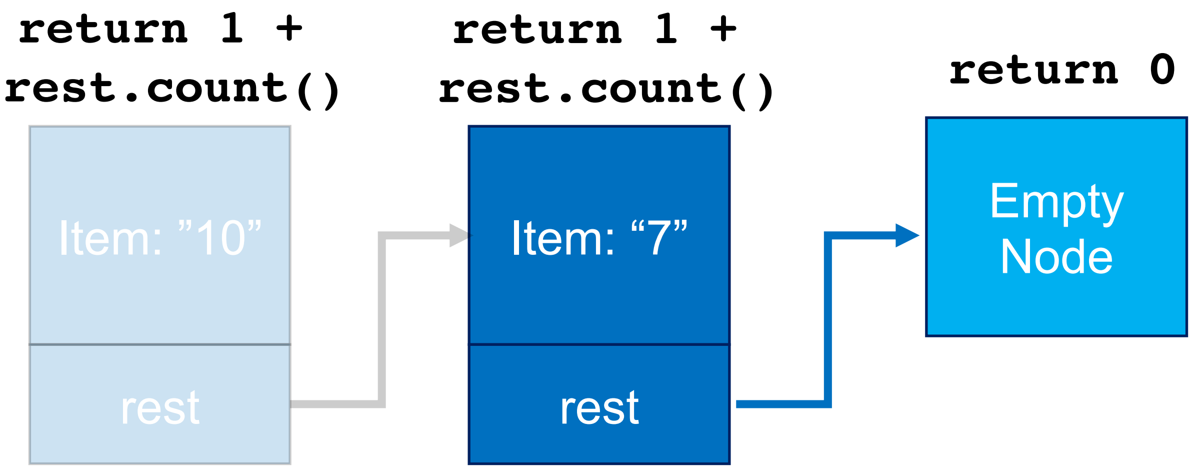
- **`this.rest.count()`** is 0

So, **`this.count()`** should return...

`1 + this.rest.count()`

Recursive linked list: `count()`

What about the recursive case?



Think about the next simplest case, a list of 1.

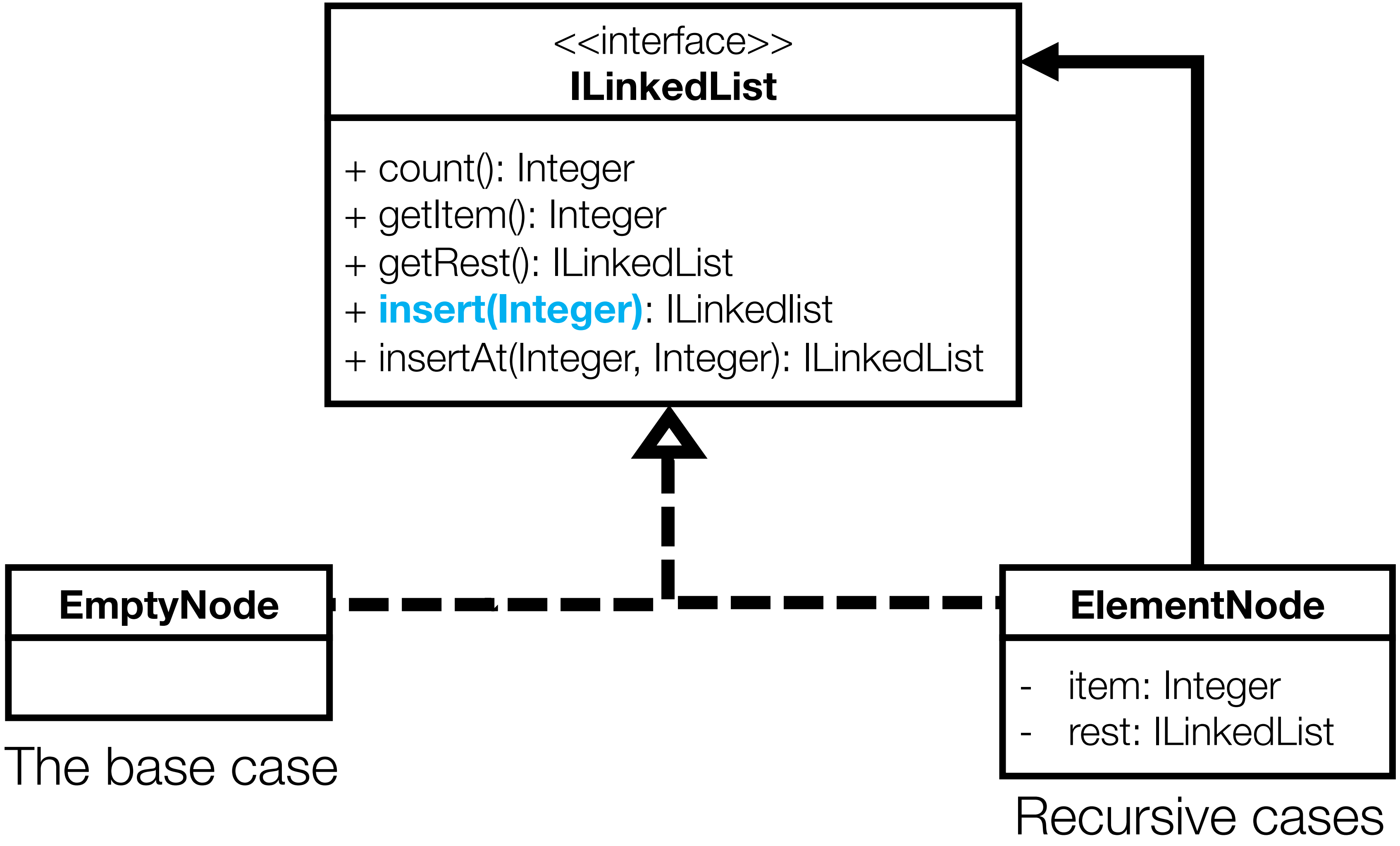
What we know:

- **`this.rest.count()`** is 0

So, **`this.count()`** should return...

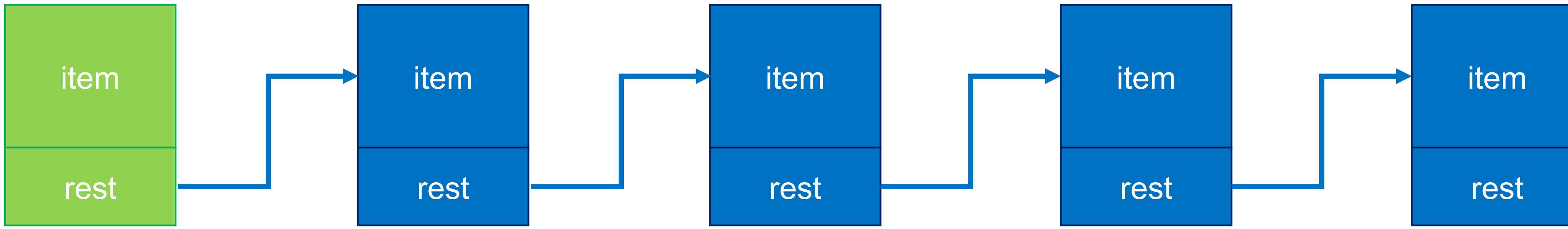
`1 + this.rest.count()`

Recursive linked list implementation



`insert(Integer): ILinkedList`

Create a new ElementNode containing the Integer, put it at the beginning



`insert(Integer): ILinkedList`

- Insert at the head of the list so doesn't need to be recursive
- BUT, still need to tackle insert for both node types
    - Head is list with contents > ElementNode
    - Head is empty list > Empty Node

Exercise 1: implementing insert

Consider how you could implement the insert method for the recursive linked list

- What would the EmptyNode implementation look like?
- What would the ElementNode implementation look like?

Discuss in breakout rooms (actually implementing is optional)

I will call on one or two groups to share their conclusions / questions

insert(Integer): ILinkedList

EmptyNode.java

```java
public ILinkedList insert(Integer item)
{
  return new ElementNode(item, this);
}
```

insert(Integer): ILinkedList

EmptyNode.java

```java
public ILinkedList insert(Integer item)
{
  return new ElementNode(item, this);
}
```

ElementNode.java

```java
public ILinkedList insert(Integer item)
{
  return new ElementNode(item, this);
}
```

insert(Integer): ILinkedList

EmptyNode.java

```
public ILinkedList insert(Integer item)
{
  return new ElementNode(item, this);
}
```

ElementNode.java

```
public ILinkedList insert(Integer item)
{
  return new ElementNode(item, this);
}
```

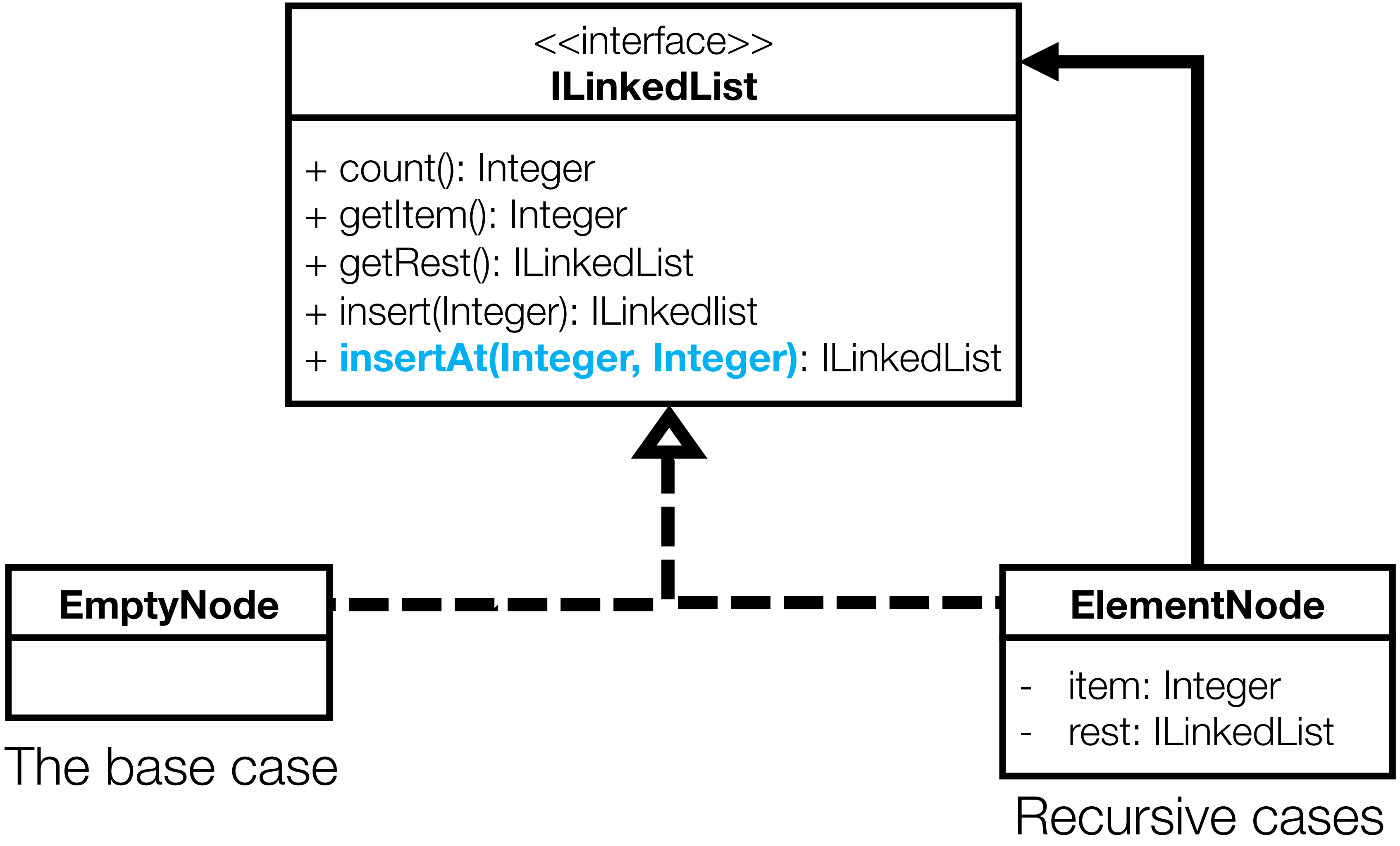
`this` represents the current "head" of the List

How does Java know which version to call?

Dynamic dispatch

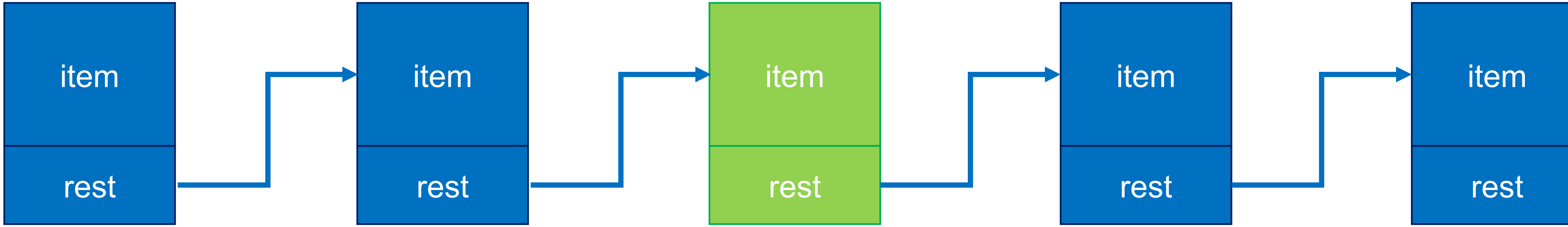
- If the list that calls insert is an **`ElementNode`**, Java will call the **`ElementNode insert`** implementation
- If the list that calls insert is an **`EmptyNode`**, Java will call the **`EmptyNode insert`** implementation

Recursive linked list implementation



`insertAt(Integer, Integer): ILinkedList`

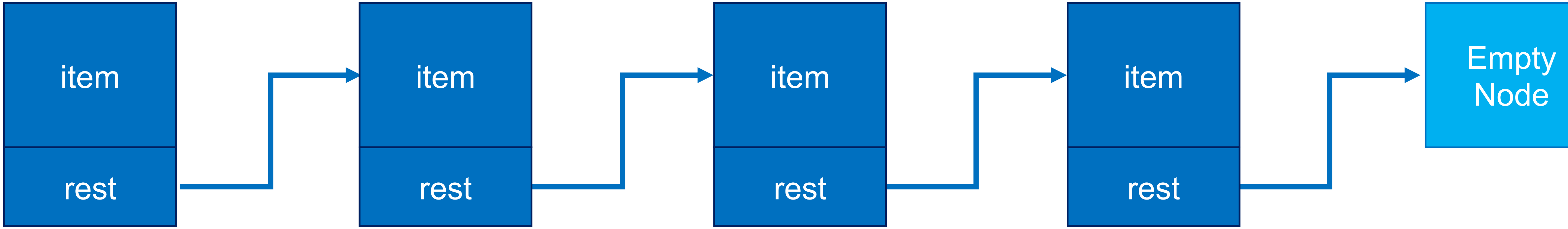
Create a new ElementNode containing the Integer, put it at index e.g. 2



`insertAt(Integer, Integer): ILinkedList`

- Insert at given index
- Will need to recursively check nodes to find the right index
- Also need to check index is in bounds

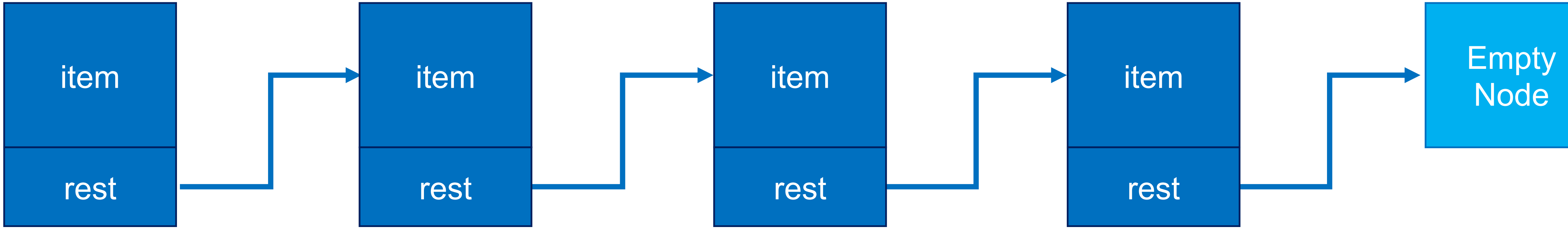
Example: insert at index 2



Start at node 0

index = 2

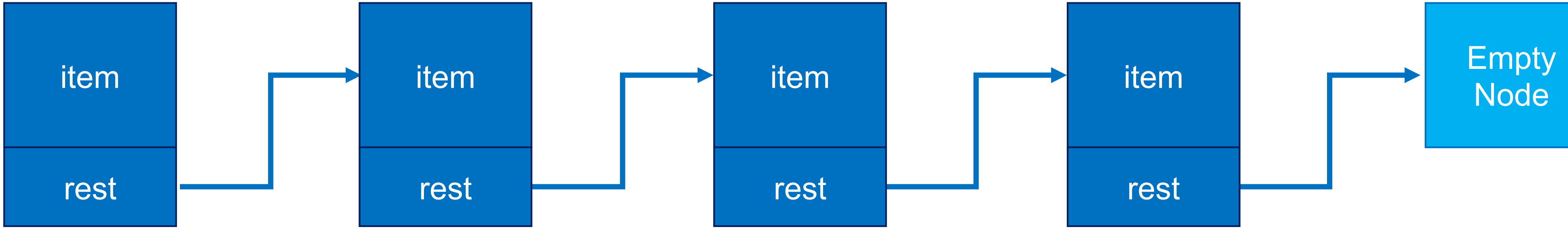
Example: insert at index 2



Go to node 1, subtract 1 from index

index = 1

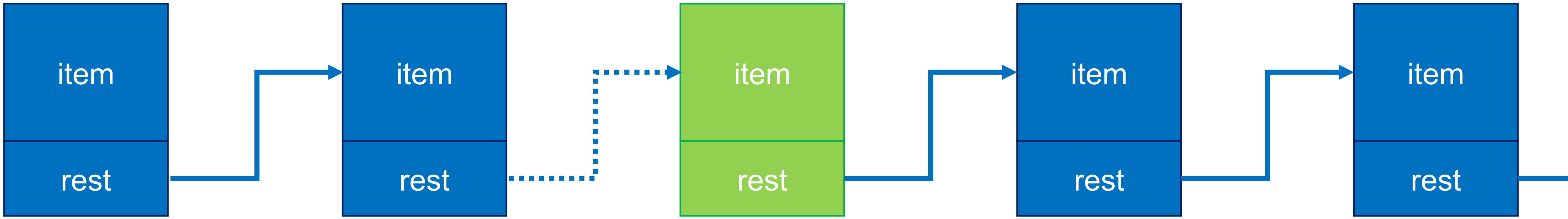
Example: insert at index 2



Go to node 2, subtract 1 from index

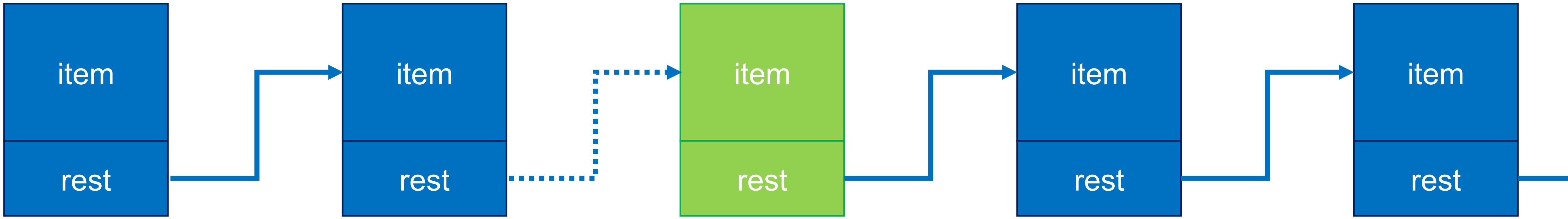
Index = 0

Example: insert at index 2



Create new node, set its next node to point to node already at node 2…

Example: insert at index 2



Create new node, set its next node to point to node already at node 2…

Connect to previous node (index 1)

insertAt implementation – base case

- Start with the base case – existing list is empty
- Two cases:
  - index = 0, same as insert()
  - index is out of range > throw exception

insertAt implementation – base case

```
public ILinkedList insertAt(Integer item, Integer index)
                              throws IndexOutOfBoundsException {
  if (!index.equals(0)) {
    throw new IndexOutOfBoundsException();
  } else {
    return new ElementNode(item, this);
  }
}
```

insertAt implementation – base case

```
public ILinkedList insertAt(Integer item, Integer index)
                            throws IndexOutOfBoundsException {
  if (!index.equals(0)) {
    throw new IndexOutOfBoundsException();
  } else {
    return new ElementNode(item, this);
  }
}
```

Index out of range

insertAt implementation – base case

```
public ILinkedList insertAt(Integer item, Integer index)
                          throws IndexOutOfBoundsException {
  if (!index.equals(0)) {
    throw new IndexOutOfBoundsException();
  } else {
    return new ElementNode(item, this);
  }
}
```

Insert here…

insertAt implementation – recursive case

- Three cases:
  - Index is out of range → throw exception
  - This is the index we want to insert at → insert here
  - This is NOT the index we want to insert at → check next node

insertAt implementation – recursive case

```
public ILinkedList insertAt(Integer item, Integer index)
                                  throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                                 this.rest.insertAt(item, index-1));
  }
}
```

insertAt implementation – recursive case

```
public ILinkedList insertAt(Integer item, Integer index)
                                throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                           this.rest.insertAt(item, index-1));
  }
}
```

Index out of range

insertAt implementation – recursive case

```
public ILinkedList insertAt(Integer item, Integer index)
                                    throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                                    this.rest.insertAt(item, index-1));
  }
}
```

This is NOT the index we want to insert at

insertAt implementation – recursive case

```
public ILinkedList insertAt(Integer item, Integer index)
                                    throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                this.rest.insertAt(item, index-1));
  }
}
```

Recursive call

Reduce index

insertAt implementation – recursive case

```
public ILinkedList insertAt(Integer item, Integer index)
                                      throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                                      this.rest.insertAt(item, index-1));
  }
}
```

This is the index we want to insert at

insertAt implementation – recursive case

```
public ILinkedList insertAt(Integer item, Integer index)
                                    throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                           this.rest.insertAt(item, index-1));
  }
}
```

Copy the contents of this node
to maintain link from previous node

insertAt implementation – recursive case

```
public ILinkedList insertAt(Integer item, Integer index)
                                  throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                                    this.rest.insertAt(item, index-1));
  }
}
```

Return new node with new item to the previous recursive call

insertAt implementation – recursive case

```
public ILinkedList insertAt(Integer item, Integer index)
                                throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                           this.rest.insertAt(item, index-1));
  }
}
```

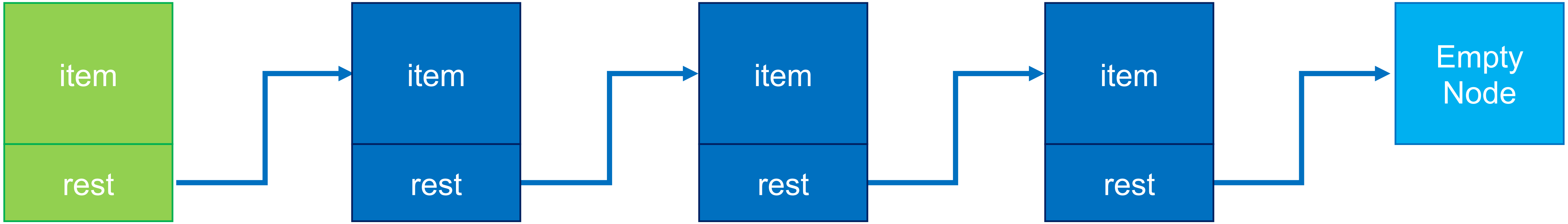
Point “rest” to the copy node

Break (10 mins)

10 8 6 4 2 0

Efficiency

Linked List is very efficient when you're only adding/removing from the front



Accessing/inserting/removing anywhere else is less efficient

- Have to traverse the list each time

insertAt efficiency – recursive case

```
public ILinkedList insertAt(Integer item, Integer index)
                                      throws IndexOutOfBoundsException {
  if (index > this.count() || index < 0) {
    throw new IndexOutOfBoundsException();
  } else if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                                      this.rest.insertAt(item, index-1));
  }
}
```

Do we really need to do this?

insertAt efficiency – recursive case

Pro

- Prevents list traversal if the index is invalid
- `list.insertAt(100, -1);`

Con

- If the index is valid, the check is repeated every node
- `list.insertAt(100, list.count());`

```
if (index > this.count() || index < 0) {
  throw new IndexOutOfBoundsException();
}
```

insertAt efficiency – recursive case

Pro

- Prevents list traversal if the index is invalid
- `list.insertAt(100, -1);`
- Design choice: if you think invalid inserts will be more common, keep the check

Con

- If the index is valid, the check is repeated every node
- `list.insertAt(100, list.count());`

```
if (index > this.count() || index < 0) {
  throw new IndexOutOfBoundsException();
}
```

insertAt efficiency – recursive case

Pro

- Prevents list traversal if the index is invalid
- `list.insertAt(100, -1);`
- Design choice: if you think invalid inserts will be more common, keep the check

Con

- If the index is valid, the check is repeated every node
- `list.insertAt(100, list.count());`
- Design choice: if you think valid inserts will be more common, can we remove the check?

```
if (index > this.count() || index < 0) {
  throw new IndexOutOfBoundsException();
}
```

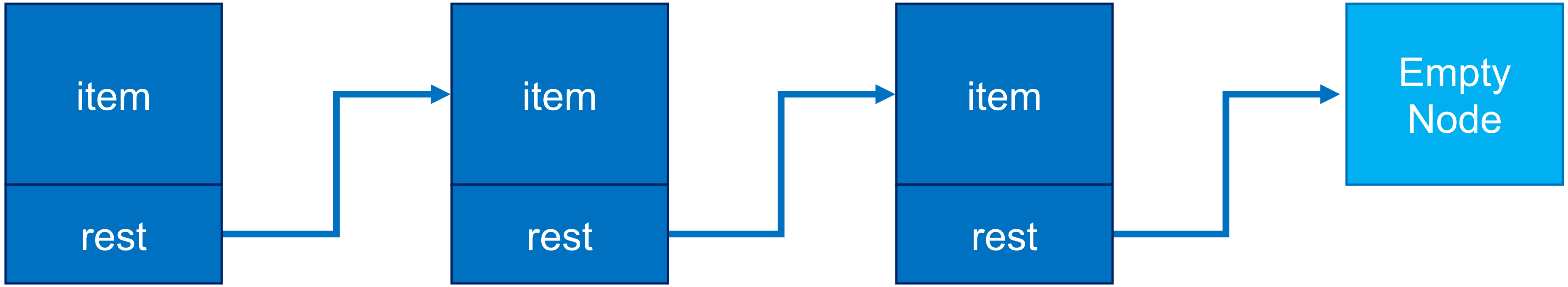
insertAt recursive case – another approach

```
public ILinkedList insertAt(Integer item, Integer index)
                                throws IndexOutOfBoundsException {
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item,
                               this.rest.insertAt(item, index –
1));
  }
}
```

We can remove the index check

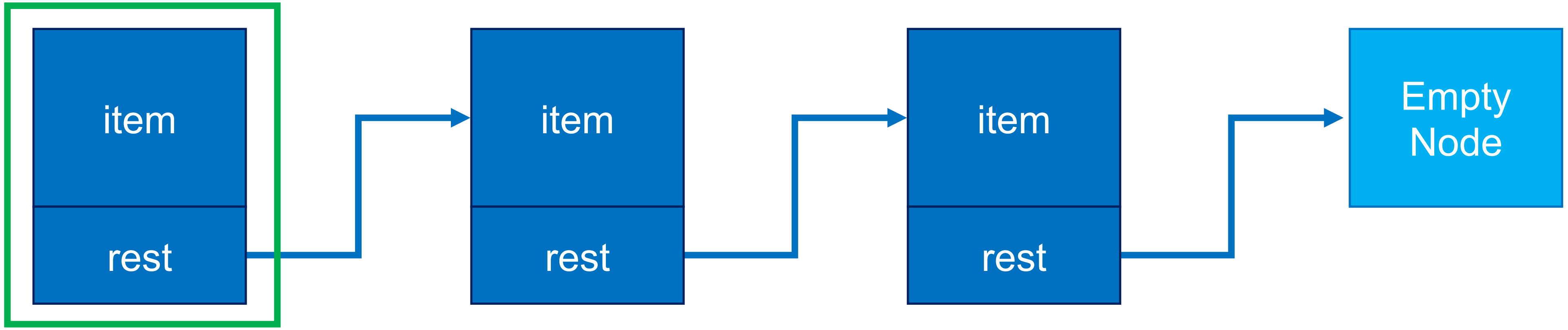
EmptyNode will catch an invalid index

Example: `insertAt(100, -1);`




```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

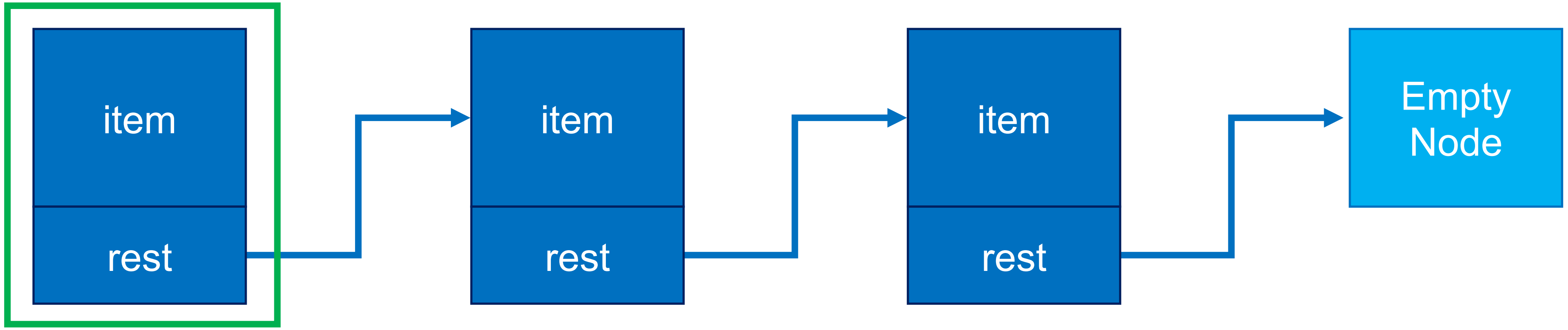
Example: `insertAt(100, -1);`



index: -1

```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

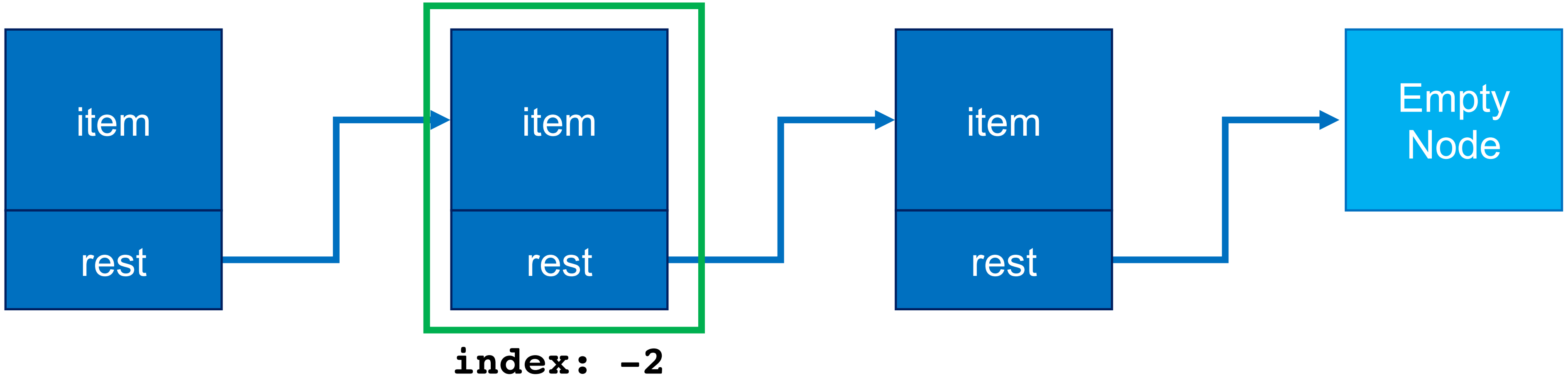
Example: `insertAt(100, -1);`



`index: -1`

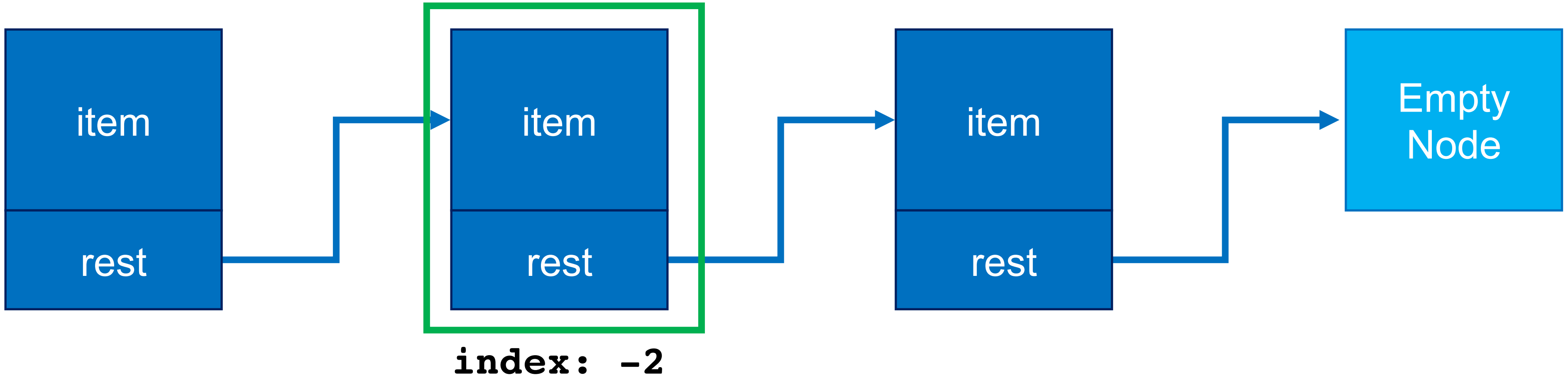
```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

Example: `insertAt(100, -1);`



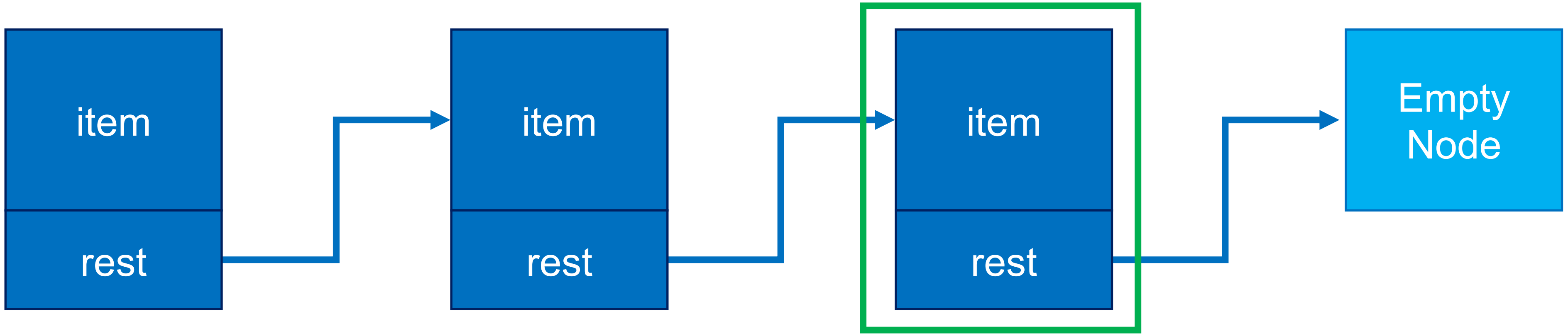

```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

Example: `insertAt(100, -1);`




```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

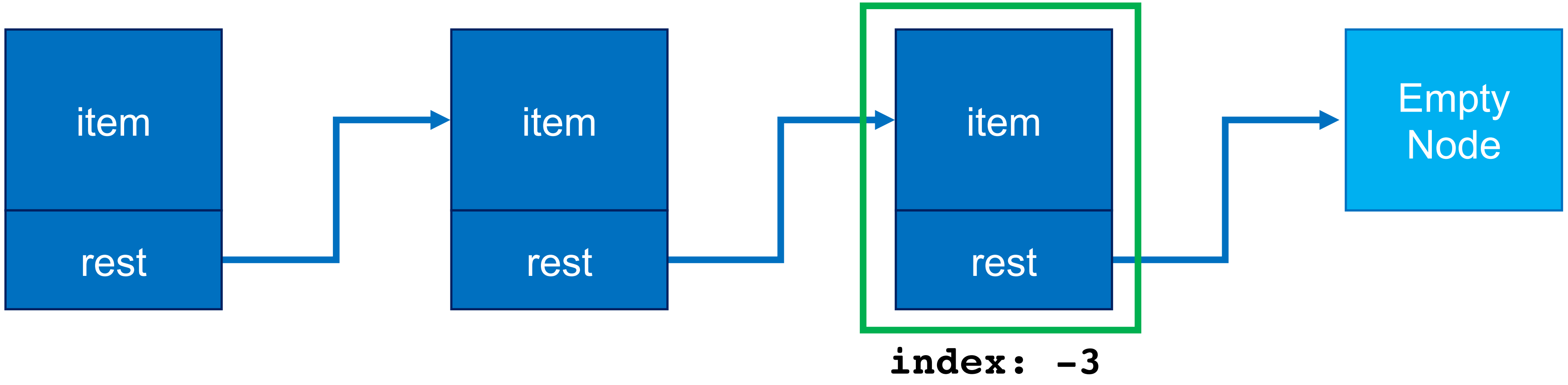
Example: `insertAt(100, -1);`



`index: -3`

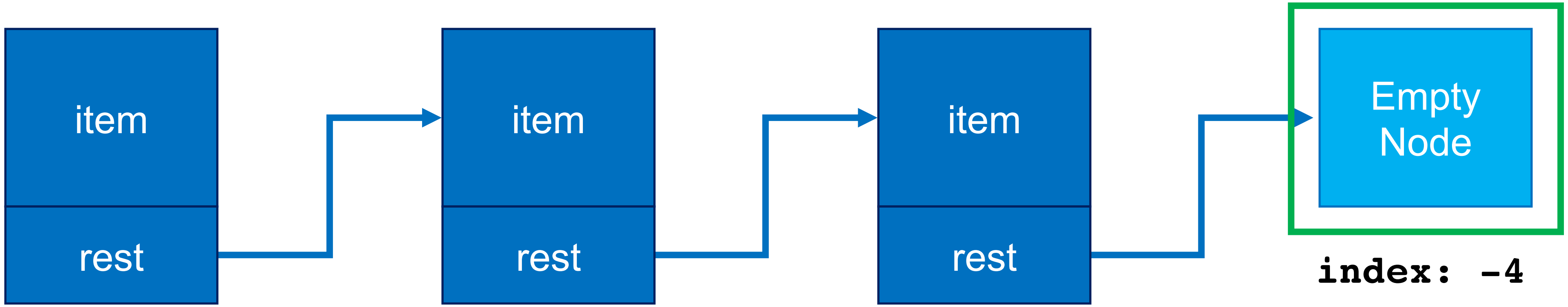
```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

Example: `insertAt(100, -1);`



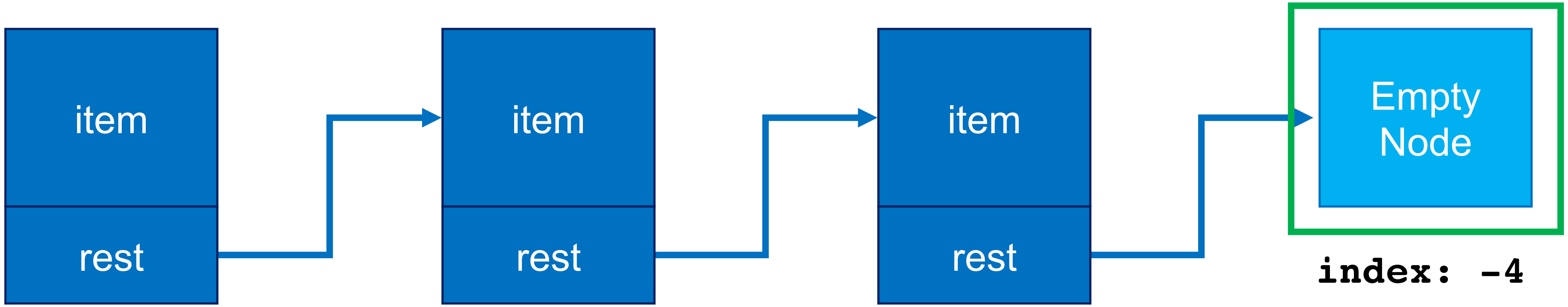

```
public ILinkedList insertAt(Integer item, Integer index) { // in ElementNode
  if (index.equals(0)) {
    ILinkedList thisCopy = new ElementNode(this.item, this.rest);
    return new ElementNode(item, thisCopy);
  } else {
    return new ElementNode(this.item, this.rest.insertAt(item, index-1));
  }
}
```

Example: `insertAt(100, -1);`




```
public ILinkedList insertAt(Integer item, Integer index) { // in EmptyNode
  if (!index.equals(0)) {
    throw new IndexOutOfBoundsException();
  } else {
    return new ElementNode(item, this);
  }
}
```

Example: `insertAt(100, -1);`




```
public ILinkedList insertAt(Integer item, Integer index) { // in EmptyNode
  if (!index.equals(0)) {
    throw new IndexOutOfBoundsException();
  } else {
    return new ElementNode(item, this);
  }
}
```

Walkthrough

Completed Linked List implementation

Stack implementation

Using a recursive linked list

A mutable Stack ADT

- **`void push(Integer item)`** - push an Integer on to the Stack
- **`Integer pop() throws EmptyStackException`** – returns and removes the most recently-added item.
- **`Integer top() throws EmptyStackException`** – returns the most recently-added item
- **`boolean isEmpty()`** – checks if the Stack is empty.

Fields and constructor(s)

```java
public class Stack implements IStack {
  private ILinkedList top;

  private Stack() {
    this.top = new EmptyNode();
  }

  public static Stack createEmpty() {
    return new Stack();
  }
}
```

Fields and constructor(s)

```
public class Stack implements IStack {
  private ILinkedList top;
```

Underlying data structure

```
  private Stack() {
    this.top = new EmptyNode();
  }

  public static Stack createEmpty() {
    return new Stack();
  }
}
```

Fields and constructor(s)

```
public class Stack implements IStack {
  private ILinkedList top;

  private Stack() {
    this.top = new EmptyNode();
  }

  public static Stack createEmpty() {
    return new Stack();
  }
}
```

Why private?

- Sometimes want to prevent direct access to constructors
- Most useful for immutable
- Not necessary here (but fine)

Fields and constructor(s)

```
public class Stack implements IStack {
  private ILinkedList top;

  private Stack() {
    this.top = new EmptyNode();
  }

  public static Stack createEmpty() {
    return new Stack();
  }
}
```

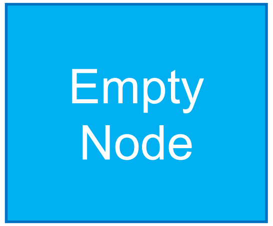
Convenience method creates a Stack without "new"

Creating a Stack

```
Stack aStack = Stack.createEmpty();
```

Inside **`aStack`**:

```
this.top =
```



A mutable Stack ADT

- `void push(Integer item)` - push an Integer on to the Stack
- **`Integer pop() throws EmptyStackException`** – returns and removes the most recently-added item.
- **`Integer top() throws EmptyStackException`** – returns the most recently-added item
- **`boolean isEmpty()`** – checks if the Stack is empty.

push(Integer): void

```
public void push(Integer item) {
  this.top = this.top.insert(item);
}
```

push(Integer): void

```
public void push(Integer item) {
   this.top = this.top.insert(item);
}
```

Dynamic dispatch. One of:

`EmptyNode's insert(Integer)`

`ElementNode insert(Integer)`

The linked list is immutable so reassign `this.top`

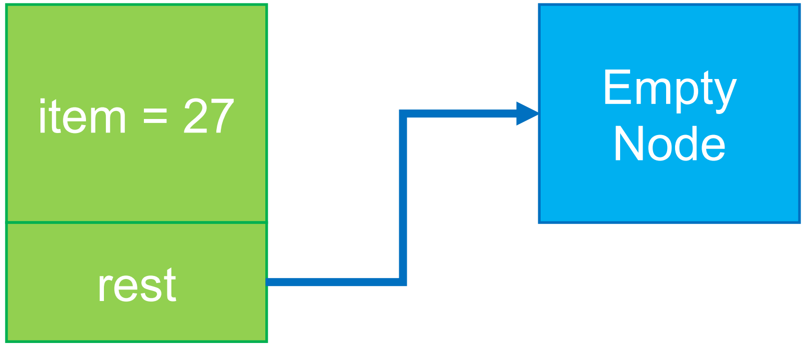
- `Stack` is mutable

Pushing items on to the Stack

```
aStack.push(27);
```

Inside **`aStack`**:

`this.top =`

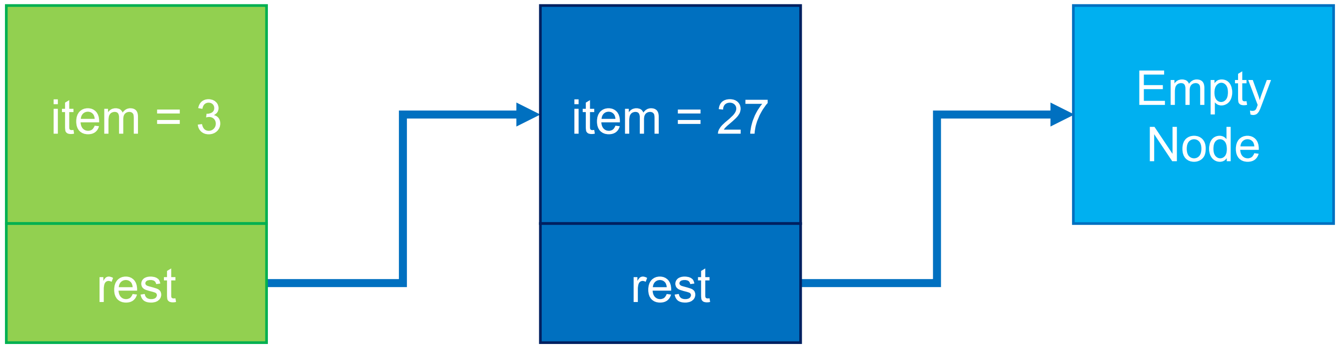


Pushing items on to the Stack

```
aStack.push(27);
aStack.push(3);
```

Inside **aStack**:

`this.top =`



A mutable Stack ADT

- **`void push(Integer item)`** - push an Integer on to the Stack
- **`Integer pop() throws EmptyStackException`** – returns and removes the most recently-added item.
- **`Integer top() throws EmptyStackException`** – returns the most recently-added item
- **`boolean isEmpty()`** – checks if the Stack is empty.

isEmpty(): boolean

```
public boolean isEmpty() {
  return this.top.count().equals(0);
}
```

isEmpty(): boolean

```
public boolean isEmpty() {
   return this.top.count().equals(0);
}
```

Dynamic dispatch. One of:

`EmptyNode's count()`

`ElementNode count()`

A mutable Stack ADT

- **`void push(Integer item)`** - push an Integer on to the Stack
- **`Integer pop() throws EmptyStackException`** – returns and removes the most recently-added item.
- **`Integer top() throws EmptyStackException`** – returns the most recently-added item
- **`boolean isEmpty()`** – checks if the Stack is empty.

top(): Integer

```
public Integer top() throws EmptyStackException
{
  if (this.isEmpty())
    throw new EmptyStackException();
  return this.top.getItem();
}
```

top(): Integer

```
public Integer top() throws EmptyStackException
{
  if (this.isEmpty())
    throw new EmptyStackException();
  return this.top.getItem();
}
```

Ensures the specification is met

top(): Integer

```
public Integer top() throws EmptyStackException
{
   if (this.isEmpty())
      throw new EmptyStackException();
   return this.top.getItem();
}
```

Not necessary

A “checked” exception

Built-in, inherits

`RunTimeException`

top(): Integer

```
public Integer top() throws EmptyStackException
{
  if (this.isEmpty())
    throw new EmptyStackException();
  return this.top.getItem();
}
```

Dynamic dispatch. One of:

`EmptyNode's getItem()`

`ElementNode getItem()`

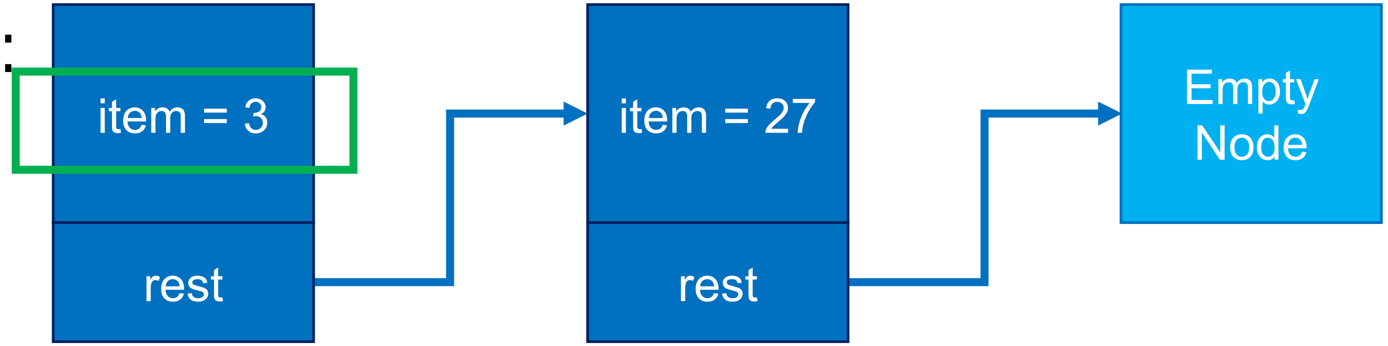
(*We* know it can't be an EmptyNode but the compiler does not)

Getting the top item

`aStack.top();` → 3

Inside **`aStack`**:

`this.top =`



A mutable Stack ADT

- `void push(Integer item)` - push an Integer on to the Stack
- `Integer pop() throws EmptyStackException` – returns and removes the most recently-added item.
- `Integer top() throws EmptyStackException` – returns the most recently-added item
- `boolean isEmpty()` – checks if the Stack is empty.

pop(): Integer

```java
public Integer pop() throws EmptyStackException
{
   Integer poppedItem = this.top();
   this.top = this.top.getRest();
   return poppedItem;
}
```

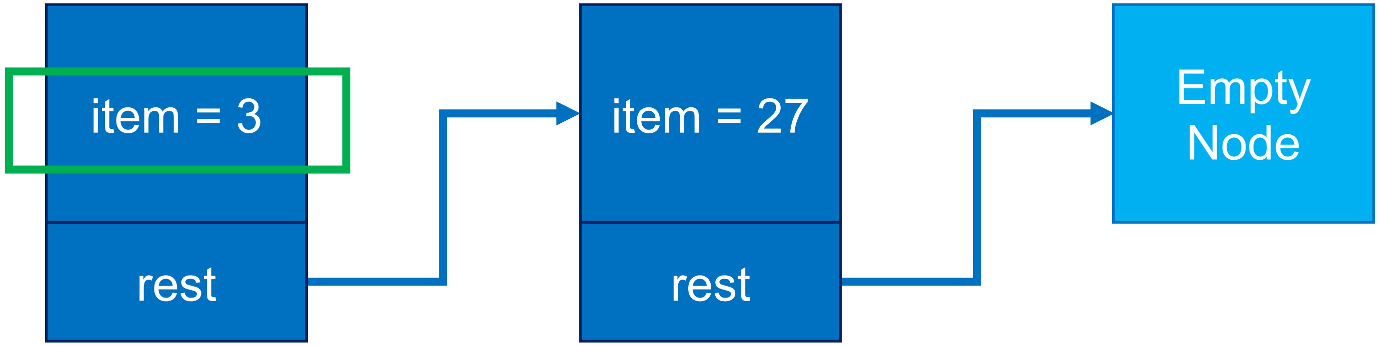
pop(): Integer

```
public Integer pop() throws EmptyStackException
{
  Integer poppedItem = this.top();
  this.top = this.top.getRest();
  return poppedItem;
}
```

Will throw exception if empty

- Stores the return value if not

this.top =

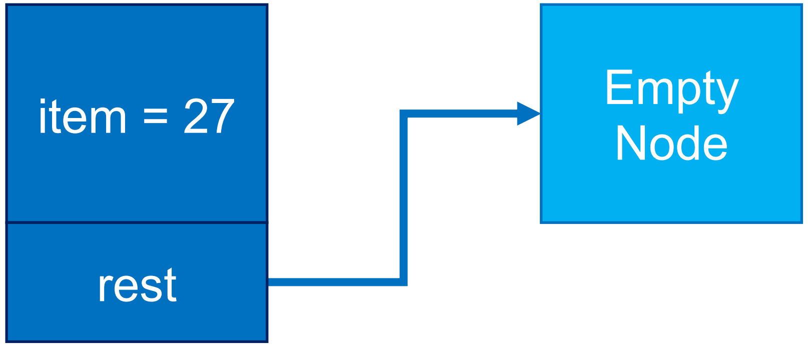


pop(): Integer

```
public Integer pop() throws EmptyStackException
{
   Integer poppedItem = this.top();
   this.top = this.top.getRest();
   return poppedItem;
}
```

Removes the top element

this.top =



Walkthrough

- Completed Stack
- Tests

Exercise 2

- Implement **`Queue`** with the recursive linked list as the underlying data structure
- Use the **`IQueue`** interface & tests from this week's sample code

Immutable Stack

Using a recursive linked list

An immutable Stack ADT

- **`IImmutableStack createEmpty()`** – creates a new empty stack.
- **`IImmutableStack push(Integer item)`** - returns a new stack with item at the top.
- **`IImmutableStack pop() throws EmptyStackException`** – ~~returns and~~ removes the most recently-added item.
- **`Integer top() throws EmptyStackException`** – returns the most recently-added item
- **`boolean isEmpty()`** – checks if the Stack is empty.

An immutable Stack ADT

- `IImmutableStack createEmpty()` – creates a new empty stack.
- `IImmutableStack push(Integer item)` - returns a new stack with item at the top.
- `IImmutableStack pop() throws EmptyStackException` – removes the most recently-added item.
- `Integer top() throws EmptyStackException` – returns the most recently-added item
- `boolean isEmpty()` – checks if the Stack is empty.

Fields and constructors

```
public class ImmutableStack implements IImmutableStack {
  private final ILinkedList top;

  private ImmutableStack() {
    this.top = new EmptyNode();
  }

  private ImmutableStack(ILinkedList elements) {
    this.top = elements;
  }

}
```

Fields and constructors

```
public class ImmutableStack implements IImmutableStack {
  private final ILinkedList top;

  private ImmutableStack() {
    this.top = new EmptyNode();
  }

  private ImmutableStack(ILinkedList elements) {
    this.top = elements;
  }

}
```

Underlying data structure

- `final` ensures immutability

Fields and constructors

```
public class ImmutableStack implements IImmutableStack {
  private final ILinkedList top;

  private ImmutableStack() {
    this.top = new EmptyNode();
  }

  private ImmutableStack(ILinkedList elements) {
    this.top = elements;
  }

}
```

Doesn't *need* to be private

- emptyStack will serve as constructor

Fields and constructors

```
public class ImmutableStack implements IImmutableStack {
  private final ILinkedList top;

  private ImmutableStack() {
    this.top = new EmptyNode();
  }

  private ImmutableStack(ILinkedList elements) {
    this.top = elements;
  }
}
```

Definitely private → Don't want clients to know about the underlying s

- Need for immutable methods

createEmpty(): IImmutableStack

Typically static, calls private/public constructor

```
public static ImmutableStack createEmpty() {
    return new ImmutableStack();
}
```

Aside: choosing array vs linked list for underlying data structure

Rules of thumb

- Use an array when random access is important
  - i.e. access by index
  - will be faster for insert at index as well
- Use a linked list when random access/order is not important
  - faster for add/remove (doesn't involve resizing)

Assignment 6

Recursive ADTs

- No choice of underlying data structure
  - i.e. no arrays
- Don’t assume today’s Linked List is the best ADT for the job!
  - Use two nodes BUT
  - Linked list ADT methods may support different operations
  - Nodes may need different fields

Exercise 3: implement the remaining ImmutableStack methods

- **`IImmutableStack createEmpty()`** – creates a new empty stack.
- **`IImmutableStack push(Integer item)`** - returns a new stack with item at the top.
- **`IImmutableStack pop() throws EmptyStackException`** – removes the most recently-added item.
- **`Integer top() throws EmptyStackException`** – returns the most recently-added item
- **`boolean isEmpty()`** – checks if the Stack is empty.