Stack and Queue

Lecture Task

- Enemy AI: Lecture Task 1 -

Functionality: In the game.enemyai.AIPlayer class, implement the following methods:

- A method named “locatePlayer” with:
  - Parameters of type String and LinkedListNode of PlayerLocation
    - The String represents the player id of the player to locate
    - The LinkedListNode is the head of a containing references to PlayerLocation objects
    - You can assume the input linked list is not empty/null
  - Returns the PlayerLocation of the player with the input id
    - The PlayerLocations objects have variables for x location, y location, and player id. The x/y coordinates are Doubles

- A method named “closestPlayer” with:
  - One parameter of type LinkedListNode of PlayerLocation
  - Returns the PlayerLocation of the closest player to this AI player
    - Do not return this player itself (eg. the distance would be 0 and it would always be returned
    - You have access to the id of the this player (The constructor parameter) which can be used to find the location of this player
    - Distance should be calculated as L2/Euclidean distance (Square root of the sum of the squares of the differences in x and y)

Testing: In the tests package, complete the test suite named LectureTask1 that tests this functionality.

Stack and Queue

- Data structures with specific purposes
  - Restricted features
- All operations are very efficient
  - Inefficient operations are not allowed
- We'll see a stack and queue using linked lists
- *Scala has builtin Stack and Queue classes

Stack

- LIFO
  - Last in First out
  - The last element pushed onto the stack is the first element to be popped off the stack
- Only the element on the top of the stack can be accessed

Stack Methods

- Push
  - Add an element to the top of the stack
- Pop
  - Remove the top element of the stack

Stack Implementation

- Implement a Stack class by wrapping a linked list
- Stack uses the linked list and adapts its methods to implement push and pop

```
class Stack[A] {

  var top: LinkedListNode[A] = null

  def push(a: A): Unit = {
    this.top = new LinkedListNode[A](a, this.top)
  }

  def pop(): A = {
    val toReturn = this.top.value
    this.top = this.top.next
    toReturn
  }

}
```

Stack Usage

- Create a new empty Stack
- Call push to add an element to the top
- Call pop to remove an element
- Same exact usage when using Scala's builtin Stack

```
val stack = new Stack[Int]()
stack.push(3)
stack.push(7)
stack.push(2)
stack.push(-5)

val element = stack.pop()
```

Stack Usage

- We can use Scala's list as a Stack
  - The preferred way to use the concept of a stack in practice

```
@deprecated("Stack is an inelegant and potentially poorly-performing wrapper around List.
             Use List instead: stack push x becomes x :: list; stack.pop is list.tail.", "2.11.0")
class Stack[+A] protected (protected val elems: List[A])
```

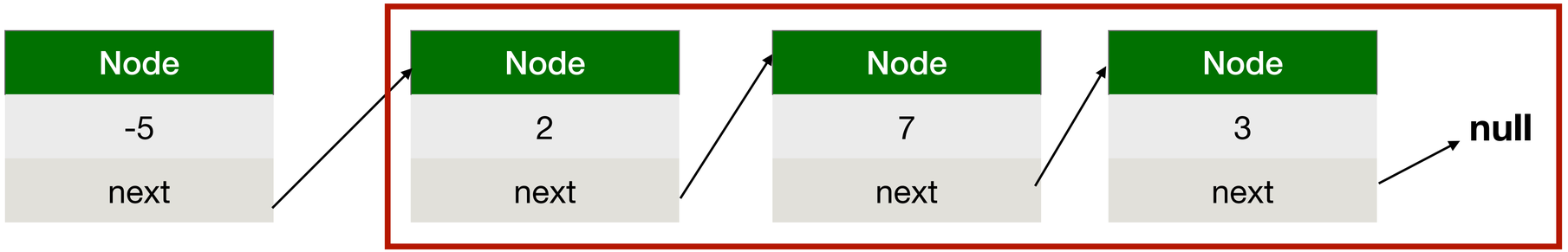
- This is very efficient!
- But wait.. doesn't this create a new list each time an element is pushed or popped since List is immutable?
  - No.. well, kind of

```
var stack = List[Int]()
stack = 3 :: stack
stack = 7 :: stack
stack = 2 :: stack
stack = -5 :: stack

val element = stack.head
stack = stack.tail
```

Stack Usage

- Before -5 is pushed, the stack is equal to nodes in the red box
- After pushing -5, the red box is unchanged
- A new List **is** returned, but it reuses the old List
  - No need to recreate the entire List

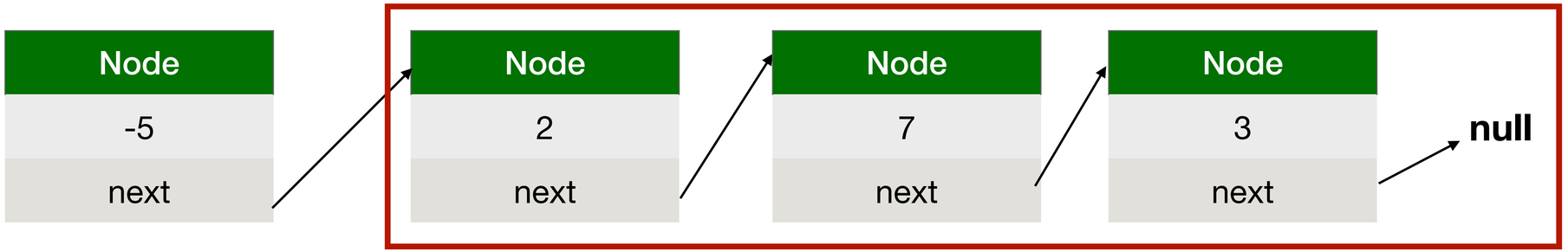
```
var stack = List[Int]()
stack = 3 :: stack
stack = 7 :: stack
stack = 2 :: stack
stack = -5 :: stack

val element = stack.head
stack = stack.drop(1)
```



Stack Usage

- Same efficiency when -5 is popped
- The red box never changed, but we update the reference stored in the stack variable
- Other parts of the program can share parts of a List without having their changes affect each other

```
var stack = List[Int]()
stack = 3 :: stack
stack = 7 :: stack
stack = 2 :: stack
stack = -5 :: stack

val element = stack.head
stack = stack.drop(1)
```



Queue

- FIFO
  - First in First out
  - The first element enqueued into the queue is the first element to be dequeued out of the queue
- Elements can only be added to the end of the queue
- Only the element at the front of the queue can be accessed

Queue Methods

- Enqueue
  - Add an element to the end of the queue
- Dequeue
  - Remove the front element in the queue

Queue Implementation

- Implement a Queue class by wrapping a linked list
- Queue needs a reference to the first and last element

```
class Queue[A] {

  var front: LinkedListNode[A] = null
  var back: LinkedListNode[A] = null

  def enqueue(a: A): Unit = {
    if (back == null) {
      this.back = new LinkedListNode[A](a, null)
      this.front = this.back
    } else {
      this.back.next = new LinkedListNode[A](a, null)
      this.back = this.back.next
    }
  }

  def dequeue(): A = {
    val toReturn = this.front.value
    this.front = this.front.next
    if(this.front == null){
      this.back = null
    }
    toReturn
  }

}
```

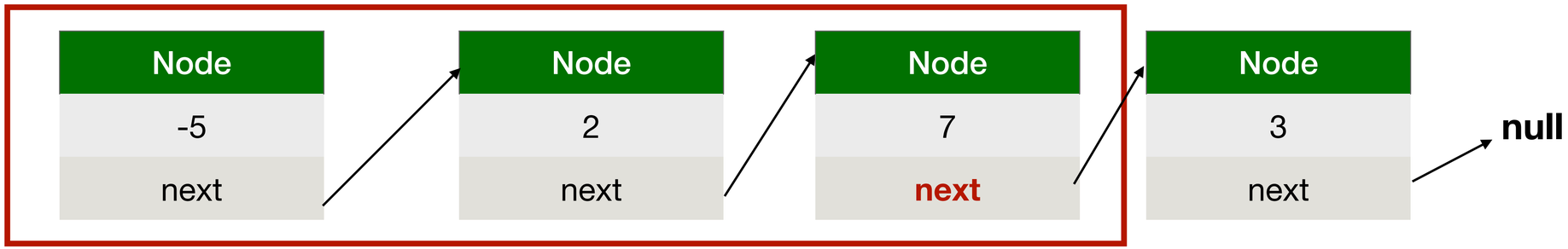
Queue Usage

- Create a new empty Queue
- Call enqueue to add an element to the back
- Call dequeue to remove the element at the front
- Same exact usage when using Scala's builtin Queue
  - [based on mutable List just like our implementation]

```
val queue = new Queue[Int]()
queue.enqueue(3)
queue.enqueue(7)
queue.enqueue(2)
queue.enqueue(-5)

val element = queue.dequeue()
```

Queue Usage

- No efficient way to use an immutable List as a queue
- To enqueue 3 the list in the red box must change
  - The next reference of the node containing 7 has to be updated
  - This List cannot be [should not be] used by other parts of the program since the List is changing



Stack Example

Infix Expressions

(12-4) - (8+9/3*2)

- The standard way to write an expression
- Operators placed **between** two operands
- Order of operations must be considered
- Parentheses used to override order of operations

Evaluating Infix Expressions

- PEMDAS
  - Parentheses -> Exponentiation -> Multiplication/ Division -> Addition/Subtraction

- (**12-4**) - (8+9/3*2)
- 8 - (8+**9/3***2)
- 8 - (8+**3*2**)
- 8 - (**8+6**)
- **8 - 14**
- **-6**

Postfix Expressions

- 12 4 - 8 9 3 / 2 * + -

- Advantages:
  - No parentheses needed
  - No order of operations to consider
  - Easy for computers to read
- Disadvantages
  - Hard for humans to read (Without practice)

Evaluating Postfix Expressions

- Find the first operator and evaluate it using the previous 2 operands
  - Repeat until there are no operators

- **12 4 -** 8 9 3 / 2 * + -
- 8 8 **9 3 /** 2 * + -
- 8 8 **3 2 *** + -
- 8 **8 6 +** -
- **8 14 -**
- **-6**

Infix -> Postfix

- Shunting Yard
  - Convert infix to postfix
- Read expression left to right
- Copy operands to the output
- Push operators and parentheses onto a **stack**
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent (Should be evaluated later) operator is on top or the stack is empty
- After reading the entire input, copy the rest of the stack to the output

(12-4) - (8+9/3*2) ➡ 12 4 - 8 9 3 / 2 * + -

Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output

Input

(12-4) - (8+9/3*2)

Stack

Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output

Input

12-4) - (8+9/3*2)

Stack

Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output

12

Input

-4) - (8+9/3*2)

Stack

Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

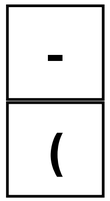
Output

12

Input

4) - (8+9/3*2)

Stack



Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

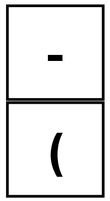
Output

12 4

Input

) - (8+9/3*2)

Stack



Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output

12 4 -

Input

) - (8+9/3*2)

Stack

Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output

12 4 -

Input

- (8+9/3*2)

Stack

Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

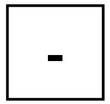
Output

12 4 -

Input

(8+9/3*2)

Stack



Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

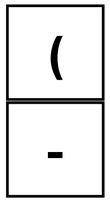
Output

12 4 -

Input

8+9/3*2)

Stack



Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

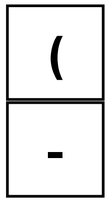
Output

12 4 - 8

Input

+9/3*2)

Stack



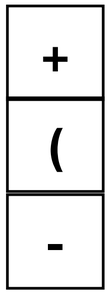
Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output **Input**

12 4 - 8 **9**/3*2)

Stack



Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

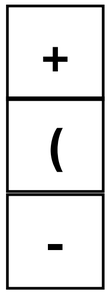
Output

12 4 - 8 9

Input

/3*2)

Stack



Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

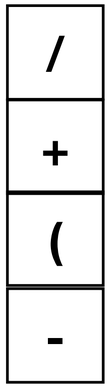
Output

12 4 - 8 9

Input

3*2)

Stack



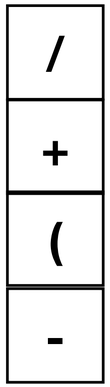
Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output **Input**

12 4 - 8 9 3 ***2**)

Stack



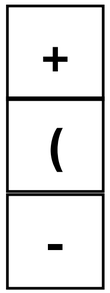
Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output **Input**

12 4 - 8 9 3 / ***2**)

Stack



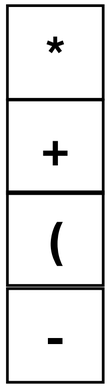
Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output **Input**

12 4 - 8 9 3 / **2**)

Stack



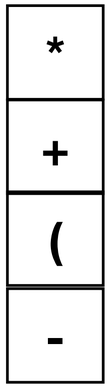
Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output **Input**

12 4 - 8 9 3 / 2)

Stack



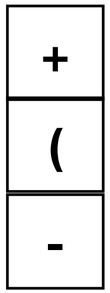
Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output **Input**

12 4 - 8 9 3 / 2 *)

Stack



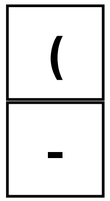
Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output **Input**

12 4 - 8 9 3 / 2 * +)

Stack



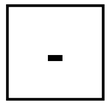
Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output **Input**

12 4 - 8 9 3 / 2 * +

Stack



Infix -> Postfix

- Read expression left to right
- Copy operands to the output
- Push operators and parentheses on a stack
  - If reading), move top of stack to output until (is popped
  - If reading an operator, first move top of stack to output until a lower precedent operator is on top or the stack is empty

Output **Input**

12 4 - 8 9 3 / 2 * + -

Stack

Lecture Task

- Enemy AI: Lecture Task 1 -

Functionality: In the game.enemyai.AIPlayer class, implement the following methods:

- A method named “locatePlayer” with:
  - Parameters of type String and LinkedListNode of PlayerLocation
    - The String represents the player id of the player to locate
    - The LinkedListNode is the head of a containing references to PlayerLocation objects
    - You can assume the input linked list is not empty/null
  - Returns the PlayerLocation of the player with the input id
    - The PlayerLocations objects have variables for x location, y location, and player id. The x/y coordinates are Doubles

- A method named “closestPlayer” with:
  - One parameter of type LinkedListNode of PlayerLocation
  - Returns the PlayerLocation of the closest player to this AI player
    - Do not return this player itself (eg. the distance would be 0 and it would always be returned
    - You have access to the id of the this player (The constructor parameter) which can be used to find the location of this player
    - Distance should be calculated as L2/Euclidean distance (Square root of the sum of the squares of the differences in x and y)

Testing: In the tests package, complete the test suite named LectureTask1 that tests this functionality.