Linked List
Lecture Question

Task: Write reduce for our linked list

- Write a method in the week8.linkedlist.LinkedListNode class (from the repo) named reduce that:
  - Takes a function of type \((A, A) \Rightarrow A\)
  - Returns \(A\)
  - Combines all the elements of the list into a single value by applying the provided function to all elements
  - You may assume the function is commutative
  - If the list has size 1, return that element without calling the provided function

**Example:**
If head stores a reference to the List(4, 6, 2)

```
head.reduce((a: Int, b: Int) => a + b) == 12
```
Recall - Array

- Sequential
- One continuous block of memory
- Random access based on memory address
  - \( \text{address} = \text{first\_address} + (\text{element\_size} \times \text{index}) \)
- Fixed Size
- Since memory adjacent to the block may be used
- Efficient when you know how many elements you'll need to store
Arrays are stored on the heap
Pointer to index 0 goes on the stack
add index * sizeOfElement to 1503 to find each element
This is called random access
Recall - Linked List

- Sequential
- Spread across memory
- Each element knows the memory address of the next element
  - Follow the addresses to find each element
- Variable Size
- Store new element anywhere in memory
- New element stores address of the first element
- myList stores a list containing: [5,3,1]
- Last link stores null
  - We say the list is "null terminated"
  - When we read a value of null we know we reached the end of the list
### Linked List

#### Program Stack

<table>
<thead>
<tr>
<th>Main Frame</th>
<th>name:myList, value:506</th>
</tr>
</thead>
</table>

#### Program Heap

<table>
<thead>
<tr>
<th></th>
<th>name: value, value:5</th>
</tr>
</thead>
<tbody>
<tr>
<td>506</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>name: next, value:795</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>name: value, value:3</th>
</tr>
</thead>
<tbody>
<tr>
<td>795</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>name: next, value:416</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>name: value, value:1</th>
</tr>
</thead>
<tbody>
<tr>
<td>416</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>name: next, value:null</td>
</tr>
</tbody>
</table>

```class LinkedListNode[A](var value: A, var next: LinkedListNode[A]) {
}
```

```var myList: LinkedListNode[Int] = new LinkedListNode[Int](1, null)
myList = new LinkedListNode[Int](3, myList)
myList = new LinkedListNode[Int](5, myList)```

- We create our own linked list class by defining a node
- A node represents one "link" in the list
- The list itself is a reference to the first/head node
- Note: This is a **mutable** list
Linked List Algorithms

- We know the structure of a linked list
- How do we operate on these lists?
- We would like to:
  - Find the size of a list
  - Print all the elements of a list
  - Access elements by location
  - Add/remove elements
  - Find a specific value
Size

- Navigate through the entire list until the next reference is null
- Count the number of nodes visited
- Could use a loop. Recursive example shown

```python
def size(): Int = {
    if(this.next == null){
        1
    }else{
        this.next.size() + 1
    }
}
```
To String

- Same as size, but accumulate the values as strings instead of counting the number of nodes.
- Recursion makes it easier to manage our commas.
- "", " is only appended if it's not the last element.

```java
override def toString: String = {
  if (this.next == null) {
    this.value.toString
  } else {
    this.value.toString + "", " + this.next.toString
  }
}
```
Access Element by Location

• Simulates array access
• Take an "index" and advance through the list that many times
• MUCH slower than array access
  - Calls next n times - O(n) runtime
  - ex. apply(4) is the same as this.next.next.next.next

```scala
def apply(i: Int): LinkedListNode[A] = {
  if (i == 0) {
    this
  } else {
    this.next.apply(i - 1)
  }
}
```
Add an Element

• To add an element we first need a reference to the node before the location of the new element

• Update the next reference of this node

• Want to add 2 in this list after 3
Add an Element

- Need reference to the node containing 3
Add an Element

- Need reference to the node containing 3
- Create the new node with next equal to this node's next
- This node's next is set to the new node
Add an Element

- Need reference to the node containing 3
- Create the new node with next equal to this node's next
- This node's next is set to the new node

```scala
def insert(element: A): Unit = {
  this.next = new LinkedListNode[A](element, this.next)
}
```
Delete a Node

- Want to delete the node containing 2
- Need a reference to the previous node
Delete a Node

- Update that node's next to bypass the deleted node
- Don't have to update deleted node
- The list no longer refers to this node

```
Node 4
    next

Node 3
    next

Node 1
    next
```
Delete a Node

- Update that node's next to bypass the deleted node
- Don't have to update deleted node
- The list no longer refers to this node

```scala
def deleteAfter(): Unit = {
  this.next = this.next.next
}
```
Find a Value

• Navigate through the list one node at a time
  • Check if the node contains the value
  • If it doesn't, move to the next node
  • If the end of the list is reached, the list does not
    contain the element

```python
def find(toFind: A): LinkedListNode[A] = {
    if (this.value == toFind) {
        this
    } else if (this.next == null) {
        null
    } else {
        this.next.find(toFind)
    }
}
Find - Recursion v. Iteration

```python
def findIterative(toFind: A): LinkedListNode[A] = {
    var node = this
    while (node != null) {
        if (node.value == toFind) {
            return node
        }
        node = node.next
    }
    null
}

def find(toFind: A): LinkedListNode[A] = {
    if (this.value == toFind) {
        this
    } else if (this.next == null) {
        null
    } else {
        this.next.find(toFind)
    }
}
```
ForEach

- Call a function on each node of the list

```scala
def foreach(f: A => Unit): Unit = {
  f(this.value)
  if(this.next != null) {
    this.next.foreach(f)
  }
}
```
Map Usage

• Recall the map method for builtin List
• Used to transform every element in a list

```scala
val numbers: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val numbersSquared = numbers.map((n: Int) => n * n)
println(numbersSquared)
```

List(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
Map

- Apply a function to each element of the list
- Return a new list containing the return values of the function

```scala
def map(f: A => A): LinkedListNode[A] = {
  val newValue = f(this.value)
  if (this.next == null) {
    new LinkedListNode[A](newValue, null)
  } else {
    new LinkedListNode[A](newValue, this.next.map(f))
  }
}
```
Map - Change Type

- Can change the type of the returned list with a second type parameter
- A could be equal to B if you don't want to change the type
- Example: You want to divide a list of Ints by 2 and have to return a list of Doubles to avoid truncation

```scala
def map[B](f: A => B): LinkedListNode[B] = {
  val newValue = f(this.value)
  if (this.next == null) {
    new LinkedListNode[B](newValue, null)
  } else {
    new LinkedListNode[B](newValue, this.next.map(f))
  }
}
```
Lecture Question

Task: Write reduce for our linked list

- Write a method in the week8.linkedlist.LinkedListNode class (from the repo) named reduce that:
  - Takes a function of type \((A, A) \Rightarrow A\)
  - Returns \(A\)
  - Combines all the elements of the list into a single value by applying the provided function to all elements
  - You may assume the function is commutative
  - If the list has size 1, return that element without calling the provided function

Example:
If head stores a reference to the List(4, 6, 2)

```
head.reduce((a: Int, b: Int) => a + b) == 12
```