Prepared by Linan Qiu <lq2137@columbia.edu>, adapted from Open Data Structures (opendatastructures.org)

Linked List

Now let's build another implementation of `AwsmList`. This time, we will adhere even more closely to the train analogy. Let's consider each car in a train – a single car does not know how many cars in total a train has. Instead, it only cares about the car in front of it (and possibly behind it). Then, at the front, we have a special car towing all of them along. Each of those cars can carry passengers, and if we want to extend a train, we simply add a car (train enthusiasts, hold your insults. I know I'm pushing the analogy a little). A linked list is exactly this.

A linked list (train) a sequence of nodes (cars). Each node (car) stores a data value (passenger) and a reference to the next node in the sequence (next car). For the last node (car), the next reference will be null (if you jump out of this car, good luck to you).

Let's see how we express this in code.

`AwsmNode`

First, let's create the "car" class.

```java
// AwsmNode.java

public class AwsmNode<T> {
  public T data;
  public AwsmNode<T> next;

  public AwsmNode(T data, AwsmNode<T> next) {
    this.data = data;
    this.next = next;
  }
}
```

This is basically a wrapper for a data and a reference to the next node. Really nothing much to this.

Let's go on to creating the train.

AwsmLinkedList

First let's make sure the linked list implements `AwsmList`. Then, our list uses the variable `head` to keep track of the first node. The first node is kind of special: it doesn't contain anything, and the current `next` reference is null (because we don't have any cars attached yet). We also add a `size` instance variable.

```java
// AwsmLinkedList.java

import java.util.Iterator;

public class AwsmLinkedList<T> implements AwsmList<T>, Iterable<T> {

  private AwsmNode<T> head;
  private int size;

  public AwsmLinkedList() {
    head = new AwsmNode<>(null, null);
    size = 0;
  }

  // ... other methods redacted
}
```

add

Now let's consider how we add something to the linked list. Let's say our train currently only has a head (and no other cars) like this:

```
[head]
```

Then, if we want to add another node containing say the string `A`, we create a new node to contain `A` and append it.

```
[head][A]
```

How exactly do we do this append? Well, in a train yard, we would hook the head's hook thingy onto the car that holds `A`. Same here: we set the `next` reference of `head` to the car containing `A`. In other words, we do this:

```java
public void addFirst (T item) {
  AwsmNode<T> newNode = new AwsmNode<>(item, null);
  // remember there's nothing behind the car holding A
```

```
  // so the car's next is null

  // then we hook head onto newNode
  head.next = newNode;
}
```

Now let's say we want to add another car holding `B` between `head` and the car holding `A`. Would this code stil work? Well, it won't. What would happen is this:

```
[head][B]   [A]
```

This is because we didn't set the car holding `B`'s `next` to the car holding `A`. We can do that by modifying our add method:

```
public void addFirst (T item) {
  AwsmNode<T> newNode = new AwsmNode<>(item, head.next);
  head.next = newNode;
}
```

In other words, we are saying that `newNode`'s `next` is going to be whatever `head.next` **was**, then we set `head.next` to `newNode`. This would ensure that the car holding `B`'s next gets set to the car holding `A`.

However, now we are only to the first position. What if we wanted to add **after** `[A]`? Say we wanted to add `[C]` like this:

```
[head][B][A][C]
```

Well an algorithm of the following sort would work:

```
public void addLast (T item) {
  AwsmNode<T> newNode = new AwsmNode<>(item, null);
  // remember that this is going to be last node, so newNode.next is null

  AwsmNode<T> current = head; // get a reference to head
  // let's try to find the last existing node
  while(current.next != null) {
    // advance one node
    current = current.next;
  }

  // now that we are at the last existing node (ie. [A])
  current.next = newNode;
}
```

Convince yourself that this works. Notice that this algorithm is $O(N)$ (as opposed to `addFirst` being $O(1)$). Unfortunately, this is not something that we can improve on for now.

What if we wanted to insert at a specific index? Say like this:

```
[head][B][A][C]

I want to insert [D] at index 2:
[head][B][A][D][C]
```

Well, I can use the same idea as `addLast` just that this time I count for a specific number of nodes.

```java
public void add (T item, int index) {
  AwsmNode<T> newNode = new AwsmNode<>(item, null);

  AwsmNode<T> current = head; // get a reference to head
  // let's try to find index-th node (excluding head)
  for (int i = 0; i < index; i++) {
    current = current.next;
  }
  // in our example, we'd now be at [A], which is perfect!
  // we want [A]'s next to be [D], and [D]'s next to be [A]'s original
  // next which is [C]
  newNode.next = current.next;
  current.next = newNode;
}
```

In fact, we can rewrite the code as such:

```java
public void add (T item, int index) {
  AwsmNode<T> current = head; // get a reference to head
  // let's try to find index-th node (excluding head)
  for (int i = 0; i < index; i++) {
    current = current.next;
  }
  // in our example, we'd now be at [A], which is perfect!
  // we want [A]'s next to be [D], and [D]'s next to be [A]'s original
  // next which is [C]
  AwsmNode<T> newNode = new AwsmNode<>(item, current.next);
  current.next = newNode;
}
```

to avoid writing an additional line.

This method is $O(index)$, since we'd have to skip over `index` number of nodes. Unlike `AwsmArrayList` (or even `ArrayList`) we cannot directly access an element at an index. Instead, we'd have to quite literally jump the trains to get to where we want to.

We can again rewrite `addFirst` and `addLast` in terms of `add` with no loss to our Big-Oh (since addFirst will never run the `for` loop). We also add in some fancy stuff like updating `size` and checks for invalid `index`.

This results in:

```java
// AwsmLinkedList.java

import java.util.Iterator;

public class AwsmLinkedList<T> implements AwsmList<T>, Iterable<T> {

  private AwsmNode<T> head;
  private int size;

  public AwsmLinkedList() {
    head = new AwsmNode<>(null, null);
    size = 0;
  }

  @Override
  public void addFirst(T item) {
    add(item, 0);
  }

  @Override
  public void addLast(T item) {
    add(item, size);
  }

  @Override
  public void add(T item, int index) {
    if (index < 0 || index > size) {
      throw new IndexOutOfBoundsException();
    } else {
      AwsmNode<T> current = head;
      for (int i = 0; i < index; i++) {
        current = current.next;
      }
      AwsmNode<T> node = new AwsmNode<>(item, current.next);
      current.next = node;
      size++;
```

```
    }
  }
}
```

`remove`

Now how do we remove something from a train? We simply take away the entire car and relink the car originally in front of it with the car originally behind it.

Same thing here:

```
[head][B][A][D][C]
```

Let's say we call `removeFirst` (ie. we want to remove `[B]`). Here's what we can do:

```java
public void removeFirst() {
  head.next = head.next.next;
}
```

Good thing is in Java, we don't actually have to explicitly "take away" the `[B]` car. Instead, when Java sees that no object is pointing at (referencing) `[B]` any longer, it will sweep `[B]` up in a process that's aptly named garbage collecting.

Now what if we wanted to remove the last element? Same thing:

```java
public void removeLast() {
  AwsmNode<T> current = head;
  // we are checking for current.next.next because we want to be at the second last node
  while(current.next.next != null) {
    current = current.next;
  }
  // now we unlink the last car
  current.next = current.next.next;
  // or equivalently, current.next = null;
  // since current.next is the last element and hence
  // current.next.next will be null
}
```

We can utilize the same `for` loop construct to delete a node at a certain index:

```
[head][B][A][D][C]
```

Let's say we are trying to remove `[D]` which is index 2. We'd want to stop at index 1 `[A]` and link `[A]` to `[C]`.

```java
public void remove (int index) {
  AwsmNode<T> current = head; // get a reference to head
  // let's try to find (index - 1)-th node (excluding head)
  // in this case, we will stop at [A]
  for (int i = 0; i < index - 1; i++) {
    current = current.next;
  }
  // in our example, we'd now be at [A], which is perfect!
  // we want [A]'s next to be [C]
  current.next = current.next.next;
}
```

The code works even if we are removing the last node or the first node. Hence, we can again implement `removeFirst` and `removeLast` using `remove`. We sprinkle some fancy stuff on top and we get this:

```java
// AwsmLinkedList.java

import java.util.Iterator;

public class AwsmLinkedList<T> implements AwsmList<T>, Iterable<T> {

  private AwsmNode<T> head;
  private int size;

  public AwsmLinkedList() {
    head = new AwsmNode<>(null, null);
    size = 0;
  }

  @Override
  public void addFirst(T item) {
    add(item, 0);
  }

  @Override
  public void addLast(T item) {
    add(item, size);
  }

  @Override
  public void add(T item, int index) {
    if (index < 0 || index > size) {
      throw new IndexOutOfBoundsException();
    } else {
```

```java
      AwsmNode<T> current = head;
      for (int i = 0; i < index; i++) {
        current = current.next;
      }
      AwsmNode<T> node = new AwsmNode<>(item, current.next);
      current.next = node;
      size++;
    }
  }

  @Override
  public void removeFirst() {
    remove(0);
  }

  @Override
  public void removeLast() {
    remove(size - 1);
  }

  @Override
  public void remove(int index) {
    if (index < 0 || index > size - 1) {
      throw new IndexOutOfBoundsException();
    } else {
      AwsmNode<T> current = head;
      for (int i = 0; i < index - 1; i++) {
        current = current.next;
      }
      // remove current.next
      current.next = current.next.next;
      size--;
    }
  }
}
```

get and set

Similarly, we have to implement `get` and `set` using the for loop construct as well:

```java
// AwsmLinkedList.java

import java.util.Iterator;
```

```java
public class AwsmLinkedList<T> implements AwsmList<T>, Iterable<T> {
  // ... other methods redacted

  // I throw in a free toString() method too
  public String toString() {
    StringBuilder stringBuilder = new StringBuilder();
    AwsmNode<T> current = head.next;
    while (current != null) {
      stringBuilder.append(current.data);
      stringBuilder.append(" ");
      current = current.next;
    }
    return stringBuilder.toString().trim();
  }

  @Override
  public T getFirst() {
    return get(0);
  }

  @Override
  public T getLast() {
    return get(size - 1);
  }

  @Override
  public T get(int index) {
    if (index < 0 || index > size - 1) {
      throw new IndexOutOfBoundsException();
    } else {
      AwsmNode<T> current = head;
      for (int i = 0; i < index; i++) {
        current = current.next;
      }
      return current.data;
    }
  }

  @Override
  public void setFirst(T item) {
    set(item, 0);
  }

  @Override
  public void setLast(T item) {
    set(item, size - 1);
```

```
  }

  @Override
  public void set(T item, int index) {
    if (index < 0 || index > size - 1) {
      throw new IndexOutOfBoundsException();
    } else {
      AwsmNode<T> current = head;
      for (int i = 0; i < index; i++) {
        current = current.next;
      }
      current.data = item;
    }
  }

  @Override
  public int size() {
    return size;
  }
}
```

What is the runtime for `get()`? It is $O(N)$. Remember this. Now let's say I set up a linked list in this manner:

```
// in a main method far far away...
AwsmLinkedList<Integer> list = new AwsmLinkedList<>();
for (int i = 0; i < 100; i++) {
  list.add(i * 2);
}
```

Then I do this silly thing:

```
// continued from the above main method
for (int i = 0; i < 100; i++) {
  System.out.println(list.get(i));
}
```

What do you think is the overall runtime of this `for` loop? It is $O(N^2)$ because each `get()` is $O(N)$. **This is a huge efficiency killer** and you will be penalized heavily during your assignments if you do this. This is a very very bad way of iterating through a linked list. By doing this, you're not jumping through trains. Instead, you're making a guy jump to the 1st car, then jump to the 2nd car **by starting from the front again**, then jump to the 3rd car **by starting from the front again**. It'd be like middle school PE class where you do suicides at the basketball court. Bad memories huh?

So now we see the need for an iterator. Let's create one.

AwsmLinkedListIterator

Now let's try creating an iterator for the `AwsmLinkedList` so that we can iterate through the list easily and write lazy enhanced for loops.

This iterator is going to be different from `AwsmArrayListIterator` because we can't simply hold on to an entire `AwsmArrayList`. Instead, we'll have to hold on to a single `head`. This means that we **must** make `AwsmLinkedListIterator` a nested class of `AwsmLinkedList` because it'll need access to the head. Then, we can do what the train jumper does: jump down each car and report if there are people in each car.

```java
// AwsmLinkedList.java
import java.util.Iterator;

public class AwsmLinkedList<T> implements AwsmList<T>, Iterable<T> {

  private AwsmNode<T> head;
  private int size;

  public AwsmLinkedList() {
    head = new AwsmNode<>(null, null);
    size = 0;
  }

  @Override
  public Iterator<T> iterator() {
    return new AwsmLinkedListIterator<T>();
  }

  public class AwsmLinkedListIterator<AnotherT> implements Iterator<T> {
    public AwsmNode<T> current;

    public AwsmLinkedListIterator() {
      current = head.next;
    }

    @Override
    public boolean hasNext() {
      // ie. you're beyond the last car
      return current == null;
    }

    @Override
    public T next() {
      T data = current.data;
```

```
      // go to the next car
      current = current.next;
      return data;
    }
  }
}
```

Let's dive into this code in detail.

First, `AwsmLinkedListIterator` is a nested class, and it is generic. However, since it is another class, the generic parameter it takes in need not be `T`. After all, we are simply specifying another template. In fact, if you write `T` there, Java will still take it to mean a different `T` than the one you specified in the outside class. This seems like a bit of unnecessary flexibility that Java gives you, and indeed it is unnecessary for our use case. But don't blame the Oracle guys for being nice and considerate!

In the constructor, notice that we did not take `head` in as an argument. However, we still get to access the `head` in the outer class. This is why we declared the inner class iterator as **non-static** – so that we can directly access variables like this. We keep a reference to the first element (ie. `head.next`) as current. Hence, when someone asks our iterator for the first element via `next()`, we'd return the first element in the linked list (before moving on to the next one via `current = current.next`). We check if there's a next element by seeing if `current == null`. This would happen if we called `next()` on the last element (where `current.next` is `null` and we set `current = current.next`, hence making `current null`).

In the `iterator()` method in the outer class, we return a newly minted instance of `AwsmLinkedListIterator` that takes in, as input to the constructor, the `head` of this particular instance of `AwsmLinkedList`.

This is why `AwsmLinkedListIterator` **has to be a nested class**: it needs access to `head`. Short of making `head` public (which we really don't want to because we want to hide implementation details from users who may do silly things to themselves), there's really nothing we can do to pass `head` to `AwsmLinkedListIterator` if `AwsmLinkedListIterator` wasn't a nested class.

Making `AwsmNode` Nested

We can also make `AwsmNode` nested, since again it will only be used in the context of `AwsmLinkedList`. Again, for illustrative purposes, I gave a ridiculous name to the generic parameter of `AwsmNode` to show that within the context of `AwsmNode`, the generic parameter is different and belongs to `AwsmNode` only. This is the full source code for `AwsmLinkedList`

```java
// AwsmLinkedList.java

import java.util.Iterator;

public class AwsmLinkedList<T> implements AwsmList<T>, Iterable<T> {

  private AwsmNode<T> head;
  private int size;

  public AwsmLinkedList() {
    head = new AwsmNode<>(null, null);
    size = 0;
  }

  @Override
  public void addFirst(T item) {
    add(item, 0);
  }

  @Override
  public void addLast(T item) {
    add(item, size);
  }

  @Override
  public void add(T item, int index) {
    if (index < 0 || index > size) {
      throw new IndexOutOfBoundsException();
    } else {
      AwsmNode<T> current = head;
      for (int i = 0; i < index; i++) {
        current = current.next;
      }
      AwsmNode<T> node = new AwsmNode<>(item, current.next);
      current.next = node;
      size++;
    }
  }

  @Override
  public void removeFirst() {
    remove(0);
  }

  @Override
  public void removeLast() {
```

```
  remove(size - 1);
}

@Override
public void remove(int index) {
  if (index < 0 || index > size - 1) {
    throw new IndexOutOfBoundsException();
  } else {
    AwsmNode<T> current = head;
    for (int i = 0; i < index - 1; i++) {
      current = current.next;
    }
    // remove current.next
    current.next = current.next.next;
    size--;
  }
}

public String toString() {
  StringBuilder stringBuilder = new StringBuilder();
  AwsmNode<T> current = head.next;
  while (current != null) {
    stringBuilder.append(current.data);
    stringBuilder.append(" ");
    current = current.next;
  }
  return stringBuilder.toString().trim();
}

@Override
public T getFirst() {
  return get(0);
}

@Override
public T getLast() {
  return get(size - 1);
}

@Override
public T get(int index) {
  if (index < 0 || index > size - 1) {
    throw new IndexOutOfBoundsException();
  } else {
    AwsmNode<T> current = head;
    for (int i = 0; i < index; i++) {
```

```java
      current = current.next;
    }
    return current.data;
  }
}

@Override
public void setFirst(T item) {
  set(item, 0);
}

@Override
public void setLast(T item) {
  set(item, size - 1);
}

@Override
public void set(T item, int index) {
  if (index < 0 || index > size - 1) {
    throw new IndexOutOfBoundsException();
  } else {
    AwsmNode<T> current = head;
    for (int i = 0; i < index; i++) {
      current = current.next;
    }
    current.data = item;
  }
}

@Override
public Iterator<T> iterator() {
  return new AwsmLinkedListIterator<T>();
}

@Override
public int size() {
  return size;
}

public class AwsmLinkedListIterator<AnotherT> implements Iterator<T> {
  public AwsmNode<T> current;

  public AwsmLinkedListIterator() {
    current = head;
  }
```

```java
    @Override
    public boolean hasNext() {
      // ie. you're beyond the last car
      return current == null;
    }

    @Override
    public T next() {
      T data = current.data;
      // go to the next car
      current = current.next;
      return data;
    }
  }

  public class AwsmNode<YetAnotherT> {
    public YetAnotherT data;
    public AwsmNode<YetAnotherT> next;

    public AwsmNode(YetAnotherT data, AwsmNode<YetAnotherT> next) {
      this.data = data;
      this.next = next;
    }
  }

  public static void main(String[] args) {
    AwsmLinkedList<Integer> list = new AwsmLinkedList<>();
    list.add(1, 0);
    list.add(2, 1);
    list.add(3, 1);
    System.out.println(list);
  }
}
```

Aaaaaaand we're done!