Prepared by Linan Qiu <lq2137@columbia.edu>, adapted from Open Data Structures (opendatastructures.org) for CS3134 Spring 2016.

Array Based Lists

In the previous chapter, we talked about how arrays are insufficient for providing the full functionality of lists. (For those of you trying to scream `ArrayList` at me, yes I hear you. That's exactly what we're doing in this chapter – understanding what `ArrayList` does and creating our own `ArrayList`). Let's try and make it better.

`AwsmList` Interface

Now let's create a class for our list. After all, we want to wrap all our messy internals into one beautiful package so that when we (or someone else) uses it, we see not a whole bunch of weird private and public methods, but a beautiful `AwsmArrayList`.

You'll also find that a lot of the data structures we will cover in this class are already impelemnted by Java. This should tell you how important they are, but also how important it is to understand **how** exactly they work so that you can choose the right one to use for the right context. In this case, `ArrayList` is Java's implementation of an Array Based List. We shall call ours `AwsmArrayList`.

```
public class AwsmArrayList {

  private int[] data;

  public AwsmArrayList() {
    this.data = data;
  }
}
```

We can easily make this generic, and do:

```
public class AwsmArrayList<T> {

  private T[] data;

  public AwsmArrayList() {
    this.data = data;
  }
}
```

Ideally, if we want all the functionalities of a list, we should implement the `List<T>` interface for the full shebang. However, that'd take way too long. Instead, let's specify what we want from using our own interface. Remember that a list should:

- Store a, well, list of items
- Add and remove items from this list
- Retrieve a specific item from the list either by value (in a list of fruits, get me "Apple") or index (get me the 4th fruit in the list)
- Expand and shrink as needed
- Useres can run through this list of items one by one (**iterate** through the items)

This produces the following interface:

```java
// AwsmList.java

import java.util.Iterator;

public interface AwsmList<T> {

  public void addFirst(T item);

  public void addLast(T item);

  public void add(T item, int index);

  public void removeFirst();

  public void removeLast();

  public void remove(int index);

  public T getFirst();

  public T getLast();

  public T get(int index);

  public void setFirst(T item);

  public void setLast(T item);

  public void set(T item, int index);
```

```
  public Iterator<T> iterator();

  public int size();
}
```

This should fulfil all the conditions above. Expanding and shrinking is taken care of internally, hence there is no need for an user to call it manually. Hence, there's no specification in the interface for that functionality.

Intuition of `AwsmArrayList`

First, let's think intuitively how this will work. Let's say you have 10 slots in a carpark (or a parking lot. I'm from the part of the world that writes *colour*. Blame my colonial masters.)

```
[ ][ ][ ][ ][ ][ ][ ][ ][ ][ ]
```

Let's also say that you have a life long passion to be a... **valet**. You park cars for your customers. Being slightly OCD (like me), you always park cars as far to the left as possible, leaving no gaps. ie. your cars will be parked like this:

```
[a][b][c][ ][ ][ ][ ][ ][ ][ ]
```

Now let's say a customer `d` comes over and says that he wants to park his car after the current cars, you get an easy job. You simply add him to the 4th lot (or index 3).

```
[a][b][c][d][ ][ ][ ][ ][ ][ ]
```

If a customer `e`, however, wants to park at the 1st lot (index 0), you have to move everything down by one.

```
[e][a][b][c][d][ ][ ][ ][ ][ ]
```

Any further and our analogy kind of breaks down, so let's go ahead to the code. `:p`

`AwsmArrayList`

When we make `AwsmArrayList` implement `AwsmList`, and add the interface methods, we get this:

```java
// AwsmArrayList.java

import java.util.Iterator;

public class AwsmArrayList<T> implements AwsmList<T> {

  private T[] data;

  public static final int INITIAL_SIZE = 8;

  @SuppressWarnings("unchecked")
  public AwsmArrayList(int length) {
    data = (T[]) new Object[length];
  }

  public AwsmArrayList() {
    this(INITIAL_SIZE);
  }

  // ... rest of methods redacted
}
```

The `@Override` appears because Eclipse (which happens to be the IDE I am using) knows that I'm implementing (hence overriding the empty method signature) an interface's method. You'll also realize that you cannot declare an arary of generics like this: `T[] data = new T[8];` and you'll instead have to go through the `Object` class. This is an unfortunate consequence of generics, and the reasoning is a little convoluted and pedantic, so we're leaving it out here. This post explains it in good detail: Stackoverflow Answer

The rest of the code should be self explanatory. Let's speed up.

Add

Let's implement the `addFirst`, `addLast` and `add` functionality. It would be easiest to implement the `add` functionality first. Here, we are adding an element at a specific index. So for example, if I have a list of Pokemons `["pikachu", "bulbasaur", "mew"]` and I want to add `"squirtle"` in between `"pikachuk"` and `"bulbasaur"`, I'd do `add("squirtle", 1)` ie. add `"squirtle"` at index 1. Then, I'd need to shift `"bulbasaur"` and `"mew"` down by one position. However, if I'm adding `"squirtle"` at position 4, which is an invalid position since the highest index right now is only position 2 (remember this is zero indexed), then I'd just append `"squirtle"` to the back of the entire list.

To do this, we need to add a variable `size` that keeps track of the number of elements we have. Remember, `size` may not be the same as `data.length` since

`data.length` is the max capacity we have, while `size` is the number of elements that we have **filled up**. We initialize `size` to 0 in the constructor.

```java
// AwsmArrayList.java

import java.util.Iterator;

public class AwsmArrayList<T> implements AwsmList<T> {

  private T[] data;
  private int size;

  public static final int INITIAL_SIZE = 8;

  @SuppressWarnings("unchecked")
  public AwsmArrayList(int length) {
    data = (T[]) new Object[length];
    size = 0;
  }

  public AwsmArrayList() {
    this(INITIAL_SIZE);
  }

  @Override
  public void addFirst(T item) {
    // TODO Auto-generated method stub

  }

  @Override
  public void addLast(T item) {
    // TODO Auto-generated method stub

  }

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

      if (size == data.length) {
```

```
        // list is currently full. will need to expand.
        // expand(); (not yet implemented)
      }

      // move everything after and at index up by 1
      for (int i = size ; i > index; i--) {
        data[i] = data[i - 1];
      }

      data[index] = item;
      size++;
    }
  }

  // ... rest of methods redacted
}
```

We also make a placeholder for the `expand()` function that we haven't implemented yet. Be patient.

Again, in this method, we:

- Check if `index` is smaller than 0 and throw an exception if it is.
- Check if `index` is greater than `size`. ie. if we are trying to insert an item at position 6 into an array containing 3 elements (current `size` is 2, and the max possible index for adding is 3 (one past the last element)).
- If `size` is equal to `data.length`, we expand the array. This happens when we have say 5 elements in a length 5 array. (eg. `["pikachu", "bulbasaur", "mew", "squirtle", "charmander"]`). We obviously can't insert anything into anywhere. So we need to expand the array into an array of length say 10 and copy over the old items. (eg. `["pikachu", "bulbasaur", "mew", "squirtle", "charmander", null, null, null, null, null]`) We haven't implemented this yet.
- We move everything at the index and after it up by 1 slot.
- We insert the new item at the requested `index` and increment `size`

The reason why we implement `add()` first which seems very complex is so that we can "cheat" and be lazy (aha!) for `addFirst` and `addLast`. After all, those are just inserting items at specific locations. Hence, we easily implement `addFirst` and `addLast`:

```
// AwsmArrayList.java

import java.util.Iterator;
```

```java
public class AwsmArrayList<T> implements AwsmList<T> {

  private T[] data;
  private int size;

  public static final int INITIAL_SIZE = 8;

  @SuppressWarnings("unchecked")
  public AwsmArrayList(int length) {
    data = (T[]) new Object[length];
    size = 0;
  }

  public AwsmArrayList() {
    this(INITIAL_SIZE);
  }

  @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 {
      if (size == data.length) {
        // list is currently full. will need to expand.
        // expand(); (not yet implemented)
      }

      // move everything after and at index up by 1
      for (int i = size; i > index; i--) {
        data[i] = data[i - 1];
      }

      data[index] = item;
      size++;
    }
  }
```

```
  // ... rest of methods redacted
}
```

You'll find this pattern of implementing something that seems harder and more general then saving time later on by reusing it very very common in programming. This is one major way to save time.

Remove

Now let's implement `removeFirst`, `removeLast` and `remove`. Again, we adopt the same strategy: implement the hardest and most general case first. `remove` is kind of like the opposite of `add`. We don't bother with `shrink` first, since it is slightly less of a concern practically (lists very often grow more than they shrink).

```
// AwsmArrayList.java

import java.util.Iterator;

public class AwsmArrayList<T> implements AwsmList<T> {

  private T[] data;
  private int size;

  public static final int INITIAL_SIZE = 8;

  @SuppressWarnings("unchecked")
  public AwsmArrayList(int length) {
    data = (T[]) new Object[length];
    size = 0;
  }

  public AwsmArrayList() {
    this(INITIAL_SIZE);
  }

  @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 {
    if (size == data.length) {
      // list is currently full. will need to expand.
      // expand(); (not yet implemented)
    }

    // move everything after and at index up by 1
    for (int i = size; i > index; i--) {
      data[i] = data[i - 1];
    }

    data[index] = item;
    size++;
  }
}

@Override
public void removeFirst() {
  // TODO Auto-generated method stub

}

@Override
public void removeLast() {
  // TODO Auto-generated method stub

}

@Override
public void remove(int index) {
  if (index < 0 || index > size - 1) {
    throw new IndexOutOfBoundsException();
  } else {
    // move everything after and at index down by 1
    for (int i = index; i < size; i++) {
      data[i] = data[i + 1];
    }

    data[size - 1] = null;
```

```
      size--;
    }
  }

  // ... rest of methods redacted
}
```

Let's walk ourselves through this method.

- First we check if the `index` is less than 0 (ie. if the user is trying to remove at an invalid position) or if it is greater than the index of the last element (ie. `size - 1`)
- Then, we check if the `index` is greater than `size - 1`. This is slightly different from `add`, since we're not checking one past the last element (as in adding) but just the last element itself. For example, in an array of pokemons `["pikachu", "bulbasaur", "mew", null, null]`, the max index for removal is 2 ie. `size - 1`.
- We move everything at and after the index down by 1. So let's say we're removing the item at index 1 in the array `["pikachu", "bulbasaur", "mew", null, null]`, then we remove `"bulbasaur"` and move `"mew"` down by 1.
- This would result in `["pikachu", "mew", "mew", null, null]` if you follow the `for` loop above. That results in one additional `"mew"` which we can easily set to `null` in the line `data[size - 1] = null`.
- Finally we decrement `size`

We can now easily implement `removeFirst` and `removeLast`

```
// AwsmArrayList.java

import java.util.Iterator;

public class AwsmArrayList<T> implements AwsmList<T> {

  private T[] data;
  private int size;

  public static final int INITIAL_SIZE = 8;

  @SuppressWarnings("unchecked")
  public AwsmArrayList(int length) {
    data = (T[]) new Object[length];
    size = 0;
  }
```

```java
public AwsmArrayList() {
  this(INITIAL_SIZE);
}

@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 {
    if (size == data.length) {
      // list is currently full. will need to expand.
      // expand(); (not yet implemented)
    }

    // move everything after and at index up by 1
    for (int i = size; i > index; i--) {
      data[i] = data[i - 1];
    }

    data[index] = item;
    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 {
      // cannot remove beyond the last element
      if (index > size - 1) {
        index = size - 1;
      }

      // move everything after and at index down by 1
      for (int i = index; i < size; i++) {
        data[i] = data[i + 1];
      }

      data[size - 1] = null;
      size--;
    }
  }

  // ... rest of methods redacted
}
```

Get and Set

Implementing `get` `getFirst` and `getLast` is trivial since we are using an array. The same applies for `set`, `setFirst` and `setLast`.

```
// AwsmArrayList.java

import java.util.Iterator;

public class AwsmArrayList<T> implements AwsmList<T> {

  private T[] data;
  private int size;

  public static final int INITIAL_SIZE = 8;

  @SuppressWarnings("unchecked")
  public AwsmArrayList(int length) {
    data = (T[]) new Object[length];
    size = 0;
  }

  public AwsmArrayList() {
    this(INITIAL_SIZE);
```

```
}

@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 {
    if (size == data.length) {
      // list is currently full. will need to expand.
      // expand(); (not yet implemented)
    }

    // move everything after and at index up by 1
    for (int i = size; i > index; i--) {
      data[i] = data[i - 1];
    }

    data[index] = item;
    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 {
    // cannot remove beyond the last element
    if (index > size - 1) {
      index = size - 1;
    }

    // move everything after and at index down by 1
    for (int i = index; i < size; i++) {
      data[i] = data[i + 1];
    }

    data[size - 1] = null;
    size--;
  }
}

@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 {
    return data[index];
  }
}

@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 {
      data[index] = item;
    }
  }

  @Override
  public Iterator<T> iterator() {
    // TODO Auto-generated method stub
    return null;
  }
}
```

Now let's ignore the `iterator()` method for a while (until the next chapter) since that fella deserves a chapter to himself. Now we have a somewhat functional ArrayList (that cannot expand or shrink).

Preliminary Testing

Let's take our `AwsmArrayList` for a spin.

```
// AwsmArrayList.java

public static void main(String[] args) {
  AwsmArrayList<String> list = new AwsmArrayList<>();
  list.addFirst("pikachu");
  list.addLast("bulbasaur");
  list.addLast("mew");

  System.out.println(list.getFirst()); // pikachu
  System.out.println(list.get(1)); // bulbasaur
  list.remove(1);
  System.out.println(list.get(1)); // mew
  System.out.println(list.getLast()); // mew
  list.removeLast();
  System.out.println(list.getFirst()); // pikachu
  // so far so good!

  // let's break it
  AwsmArrayList<Integer> oops = new AwsmArrayList<>();
  for (int i = 0; i < 8; i++) {
    oops.addLast(i);
```

```
  }
  oops.addLast(8); // this line breaks the code
  // the INITIAL_SIZE is 8
  // we are adding the 9th element and this busts the limit
}
```

Turns out our `AwsmArrayList` cannot expand, and this is causing us quite a bit of inconvenience.

Expansion

However, adding an expansion function is pretty easy. We simply create a larger array, copy everything over from the old array, then set the old array's reference to the new array.

```
// AwsmArrayList.java

import java.util.Iterator;

public class AwsmArrayList<T> implements AwsmList<T> {

  private T[] data;
  private int size;

  public static final int INITIAL_SIZE = 8;
  public static final int GROWTH_FACTOR = 2;

  @SuppressWarnings("unchecked")
  public AwsmArrayList(int length) {
    data = (T[]) new Object[length];
    size = 0;
  }

  public AwsmArrayList() {
    this(INITIAL_SIZE);
  }

  @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 {
    if (size == data.length) {
      // list is currently full. will need to expand.
      expand();
    }

    // move everything after and at index up by 1
    for (int i = size; i > index; i--) {
      data[i] = data[i - 1];
    }

    data[index] = item;
    size++;
  }
}

// ... other methods redacted

@SuppressWarnings("unchecked")
private void expand() {
  T[] newData = (T[]) new Object[data.length * GROWTH_FACTOR];
  for (int i = 0; i < size; i++) {
    newData[i] = data[i];
  }
  data = newData;
  System.out.println("Array has expanded to length " + data.length);
}

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

@Override
public Iterator<T> iterator() {
  // TODO Auto-generated method stub
  return null;
}
```

```
  public static void main(String[] args) {
    AwsmArrayList<String> list = new AwsmArrayList<>();
    list.addFirst("pikachu");
    list.addLast("bulbasaur");
    list.addLast("mew");

    System.out.println(list.getFirst()); // pikachu
    System.out.println(list.get(1)); // bulbasaur
    list.remove(1);
    System.out.println(list.get(1)); // mew
    System.out.println(list.getLast()); // mew
    list.removeLast();
    System.out.println(list.getFirst()); // pikachu

    // let's break it
    AwsmArrayList<Integer> oops = new AwsmArrayList<>();
    for (int i = 0; i < 8; i++) {
      oops.addLast(i);
    }
    oops.addLast(8);
    System.out.println(oops.get(8));
  }
}
```

We add in the `size()` method trivially.

We also uncomment the `expand()` line in `add()` to ensure that the array expands whenever necessary.

And voila! Our `AwsmArrayList` produces the following output from the main method:

```
pikachu
bulbasaur
mew
mew
pikachu
Array has expanded to length 16
8
```

This is exactly what we would expect.

Notice that the `iterator()` method is still kind of blank. Indeed, there's no good way for us to go through each item other than using `get`. We'll cover that in the next chapter.