Collection Classes – ArrayList and LinkedList

A *collection* is an object that serves as a repository for other objects. It is a generic term that can be applied to many situations, but we usually use it when discussing an object whose specific role is to provide services to add, remove, and otherwise manage the elements that are contained within. For example, the ArrayList class represents a collection. It provides methods to add elements to the end of a list or to a particular location in the list based on an index value. It provides methods to remove specific elements as needed.

Some collections maintain their elements in a specific order, while others do not. We will look at the second type later, but for now, we will focus on the former.

# ArrayList – A brief Overview

We should all have an idea of what an ArrayList is and how to use it (I hope)?

As a little recap, here’s a very simple example of usage:

import java.util.ArrayList;

class ArrayListExample1 {

public static void main(String[] args) {

## ArrayList<String> animals = new ArrayList<>();

*// add elements to the array list* animals.add("Dog"); animals.add("Cat"); animals.add("Horse");

System.*out*.println("ArrayList: " + animals);

*// remove element from index 2*

## String str = animals.remove(2);

System.*out*.println("Updated ArrayList: " + animals);

System.*out*.println("Removed Element: " + str);

}

}

I’m sure you can anticipate the output.

The ArrayList class is effectively a managed array. This means that the class maintains an array “under the hood”; we never access this directly (nor are we even aware of the fact, for the most part), but instead call methods which do the accessing for us.

But, you might say, arrays are fixed size and ArrayLists are dynamic, i.e. they grow as we add elements to the list.

Not quite true!

What actually happens is that the Arraylist starts off with a fixed-size array allocated. When it runs out of space, it will reallocate a bigger array, copy all the elements from the original to the new one, and continue on its merry way.

# A simplified ArrayList

Below is code for a simplified ArrayList that mimics some of the feature of java.util.ArrayList.

This version is not generic – it can only store Strings.

It doesn’t have any exception handling.

*/\*\**

* *A simplified ArrayList class that stores Strings*

*\*/*

public class MyStringArrayList {

*/\*\**

* *This will hold our data - remember an ArrayList is nothing more than a managed array*

*\*/*

private String[] buffer;

*/\*\**

* *Index of next free location - will also help us to determine if the buffer is full*

*\*/*

private int nextFreeLoc;

*/\*\* This will change as buffer fills up and we allocate more and more storage space \*/* private int currentCapacity;

private static final int *INITIAL\_CAPACITY* = 3;*//nice and small so that we test it quickly*

*/\*\* Default Constructor \*/* public MyStringArrayList(){ currentCapacity = *INITIAL\_CAPACITY*; nextFreeLoc = 0;

buffer = new String[currentCapacity];

}

*/\*\* Add to the end of the list.*

*\**

* *Each time you need to grow the array you should declare a temporary array*
* *which is double the currentCapacity of buffer.*

*\**

* *Copy everything in buffer to tempArray*

*\**

* *Then update the buffer reference to refer to tempArray*

*\**

* *@param elem The data to be added to the end of the managed array*

*\*/*

public void add(String elem)

{

growArrayIfNeeded(); *//I've farmed this out to a private "helper" method*

buffer[nextFreeLoc++] = elem;

}

* *Add an element to a specified index. Make sure to avoid adding beyond the end of the \* array (no gaps or bufferOverflows).*
* *Also, remember to "grow" the managed array, if required.*
* *@param index where to insert (ignore if greater than nextFreeLoc - otherwise you'll get gaps)*
* *@param elem the data to insert*

*\*/*

public void add(int index, String elem)

{

*//if it's valid* if (index <= nextFreeLoc)

{

*//Make sure that we "grow" the array if needed.* growArrayIfNeeded();

*//shuffle everything up from right to left*

*//Note that this is a much easier mechanism to implement than trying to insert the new*

*//element and then shuffle everything from left to right* for (int i = nextFreeLoc; i > index; i--)

{

buffer[i] = buffer[i-1];

}

*//Now everything has moved up we can simply insert the new element* buffer[index] = elem;

*//Obviously, we've added an extra element so we must update to reflect this* nextFreeLoc++;

}

}

*/\*\* Retrieve an element from the list*

*\**

* *@param index the location to be return*
* *@return the data at buffer[index]*

*\*/*

public String get(int index)

{ if(index >= nextFreeLoc)

{

return null;

}

return buffer[index];

}

* *Removes the first occurrence of an element from the array*
* *It does this by "closing the gap" after/if it finds a matching element in the array.*
* *@param elem the element to remove*

*\*/*

public void remove(String elem)

{

boolean matchFound = false;

for (int index = 0; index < nextFreeLoc && !matchFound; index++)

{

if(buffer[index].equals(elem))

{

matchFound = true;

*//Close the gap - move elements 1 position to the left* for( int i = index; i<nextFreeLoc; i++)

{

buffer[i] = buffer[i+1];

}

nextFreeLoc--;

}

}

}

*/\*\**

* *Remove the element at the specified index.*

*\**

* *@param index the index of the element that should be removed*

*\*/*

public void remove(int index)

{

*//if it's valid* if (index <= nextFreeLoc)

{

*//Close the gap - move elements 1 position to the left* for( int i = index; i<nextFreeLoc; i++)

{

buffer[i] = buffer[i+1];

}

nextFreeLoc--;

}

}

* *Searches through the array to see if a matching element is present \* Note: We already use this mechanism for one of the remove() methods.*
* *@param elem element to search the array for*
* *@return whether the element was present in the list or not*

*\*/*

public boolean contains(String elem)

{

boolean matchFound = false;

for (int index = 0; index < nextFreeLoc && !matchFound; index++){ if(buffer[index].equals(elem)){ matchFound = true;

} }

return matchFound;

}

*/\*\**

*\**

* *@return whether the list is empty or not*

*\*/*

public boolean isEmpty()

{

return (nextFreeLoc == 0);

}

*/\*\**

*\**

* *@return the size (i.e. the number of elements stored) in the list*

*\*/*

public int size()

{

*//System.out.println(buffer.length);* return nextFreeLoc;

}

*/\*\**

* *Private helper method to check if the currently allocated space is full.*
* *If it is then it will allocate a bigger array, copy the contents, and set our \* instance field (buffer) to refer to the newly allocated space.*

*\*/*

private void growArrayIfNeeded()

{

if(nextFreeLoc == currentCapacity){

*//Allocate double the space - that will keep us going for a while*

String[] tempArr = new String[buffer.length \* 2]; currentCapacity \*= 2;

*//copy from the old space into the new* for(int i = 0; i < buffer.length; i++){ tempArr[i] = buffer[i];

}

*//Now, update so that our managed array points at the newly created array* buffer = tempArr;

}

}

@Override

public String toString()

{

String data = "";

for(int i = 0; i < nextFreeLoc; i++)

{

data += " " +buffer[i] + ",";

}

return "MyStringArrayList[" + data + " ]";

}

*/\* //This is IntelliJ's version of toString()*

*@Override*

*public String toString() { return "MyStringArrayList{" +*

*"buffer=" + Arrays.toString(buffer) +*

*'}';*

*}\*/*

}

The following points should be noted about the code above:

1. Make sure you appreciate how the internal array is managed.
2. The get() method will execute really quickly. **Random accessing of any element in an array takes constant time**, i.e. it doesn’t matter if the element is the first, last, or somewhere in the middle, it will take the exact same time to execute (later, we can contrast this with another type of list, a socalled LinkedList).
3. Removing from the middle of the list and adding to the middle of the list can be time consuming because they both require a shuffling of elements (either to close a gap or to create a gap.)

**In fact, the time taken for this is proportional to the number of elements in the list** (compare this with get() or even adding to the end of the list.)

Another related point is that adding to the list is constant time ***unless*** there is a need to grow the array. So, in many cases it will be very fast, but periodically it will take a relatively large amount of time; again, this operation is proportional to the number of elements in the list.

# Linked Lists – An Alternative to Array Lists

So, we’ve seen that an ArrayList is not really a dynamic data structure. There is an alternative type of list that is truly dynamic – a linked list.

One of the problems that arrays impose upon the ArrayList implementation is that arrays are contiguous blocks, i.e. the whole array occupies a ‘block’ of memory.

This means that if the start of the array is at memory location 0x00FF0100[[1]](#footnote-1), for example, then the second element (assuming that each element takes up 4 bytes) will be at location 0x00FF0104, and the fourth element will be at location 0x00FF010C etc.

***This is why random accessing an element in an array is so fast*** – if we want to access the nth element of the array, then it’s a simple arithmetic operation for the CPU to calculate the offset from the start of the array.

**It’s also why we must perform shuffling operations** to create space (for new elements being added to the middle of the array) and to close gaps (for elements being removed from the middle of the array.)

# Using a Linked List

The LinkedList class is also part of the java.util package.

It’s almost identical in terms of usage to an ArrayList – what we need to establish is how it differs internally (and why it is sometimes a preferred alternative).

In fact, if we take the first example in this document and replace our ArrayList with a LinkedList we get an identical use-case:

class LinkedListExample1 {

public static void main(String[] args) {

## LinkedList<String> animals = new LinkedList<>();

*// add elements to the array list* animals.add("Dog"); animals.add("Cat"); animals.add("Horse");

System.*out*.println("ArrayList: " + animals);

*// remove element from index 2*

## String str = animals.remove(2);

System.*out*.println("Updated ArrayList: " + animals);

System.*out*.println("Removed Element: " + str);

}

}

What exactly is a Linked List?

The basic idea of a linked list is that each element (of the list) contains not only data (as in, an object reference) but a ***link***, i.e. a reference, to the next element in the list.

Elements of a linked list are often called **nodes**. We need a special reference to the start of the list and this is often called **head** in code.

Below is a simple memory-map representation of a linked list. One of the things to note is that, because it’s not contiguous in memory, the list can grow indefinitely: all we have to do is use the **new** operator to allocate our object’s memory and then store the reference (to the newly allocated memory) in our list.

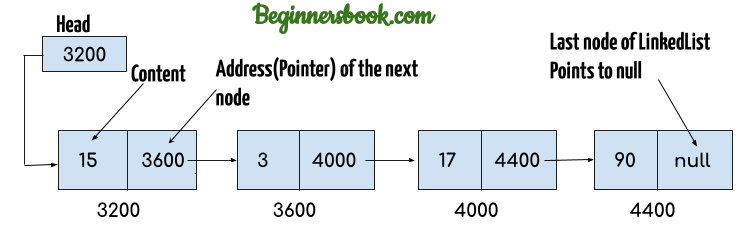
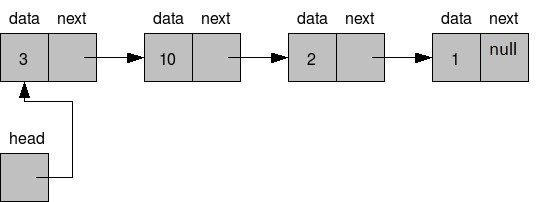


Figure: example of a **singly-linked** list.

Most illustrations don’t explicitly show the values/addresses stored in the Node – they simply use arrows, such as this one:



How do we create this capability?

# Creating a basic singly linked list

Well, here’s an example (our list is simply capable of storing integer data but it could be anything really…. and if we were to use generics it really could be anything really.)

class LinkedList {

Node head; // head of list

/\* Linked list Node\*/ private class Node { int data;

Node next;

// Constructor to create a new node Node(int d) { data = d; next = null;

}

}

}

## Adding to the start of the list

Now that we have that in place, we need some way to put stuff into the list. Here’s the simplest way: adding to the beginning of the list:

class LinkedList {

Node head; // head of list

public void addToStart(int val)

{

Node newNode = new Node(val); newNode.next = head; head = newNode;

}

/\* Linked list Node\*/ private class Node { int data;

Node next;

// Constructor to create a new node

Node(int d) { data = d; next = null;

}

}

}

Exercise: imagine you call addToStart(5) followed by addToStart(3). Try to draw the operations

(of the code in addToStart()) as they would occur.

If we add a facility to print the list, then we can test the Linked List:

class LinkedList {

Node head; // head of list

public void addToStart(int val)

{

Node newNode = new Node(val); newNode.next = head; head = newNode;

}

public void printList()

{

//us this to "walk" or traverse the list Node current = head; while(current != null){

System.out.print(current.data + ", "); current = current.next;

}

}

/\* Linked list Node\*/ private class Node { int data;

Node next;

// Constructor to create a new node

Node(int d) { data = d; next = null;

}

}

}

public class LinkListTester

{

public static void main(String[] args)

{

LinkedList aList = new LinkedList(); aList.addToStart(20); aList.addToStart(5); aList.addToStart(15); aList.addToStart(10);

aList.printList();

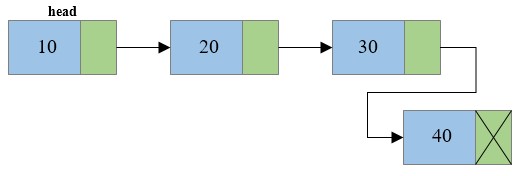
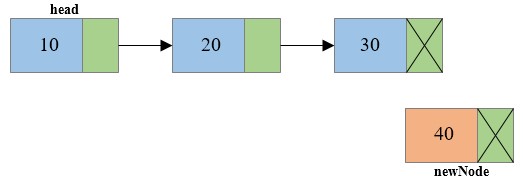
} }

## Adding to the end of the list

Of course, addToStart() possibly isn’t what you want for normal operation - they are stored in the reverse order to the insertion order

**We should have an add() method that behaves like the ArrayList’s basic add() method**, i.e. it will add the new element to the end of the list (let’s still keep the method addToStart() in our class though – it might come in handy for certain things).

Here are some diagrams to illustrate (we’re adding a node with data element ‘40’ to the end):



So, what do we need to do?

We need to *traverse/’walk’* to the last element in the list (similar to the idea for printList()) and then *update* that node’s reference.

Exercise: Implement the **add()**method.

Notes:

1. When you’re testing it initially, build up your list initially using a couple of calls to addToStart(). Then make some calls to add()and verify that it behaves as expected.
2. Now (as a further test), just use the add()method to build a list. Does it work in this scenario? If not, try to ascertain what the issue is and fix it.

So, adding to the end of the list is slow (especially if the list is big) because we have to traverse to the end of the entire list to do our insertion – it’s directly proportional to the number of elements in the list. Contrast this to inserting at the end of an Arraylist which takes constant time.

Could we improve this?

Yes. In addition to the **head** reference, we could also store a **tail** reference in our list, which we would use to **keep track of the last element in the list**. We would no longer have to traverse from the **head** to get to the end before doing the actual insertion.

Exercise: Rewrite your add() method to implement this more efficient approach.

## Get an element from the list

Say that we want to retrieve an element from the list. We could write a method with the following signature, public int get(index i),which would return the data field from the node at that index.

To implement this, our code definitely has to traverse to *index i* in order to access the node and its data. Compare again with ArrayLists – they can access the index in constant time (due to the contiguous nature of arrays, as explained earlier), so ArrayLists are more efficient at random access of the data.

Does that mean that ArrayLists are better?

Well, it depends on the requirements of your application and data structure. Shortly, we will look at other data structures that can be built from either ArrayLists or LinkedLists, and we will hopefully see that some shortcomings (of either data structure) may not be of consequence depending on the requirements.

Exercise: Write (and test) the get() method mentioned above.

1. 0x (Zero -x) is commonly used to denote hexadecimal numbers. [↑](#footnote-ref-1)