DATA STRUCTURES
LECTURE NOTES
INFO1105

2015 Julian O.

TABLE OF CONTENTS

Week 1

Data Structures:

A data structure is a systematic method of organising and accessing data. The Java collections library has many data structures built-in including ArrayList, HashMap, LinkedList, and more.

An 'Abstract Data Type' (ADT) is a mathematical model of a data structure that specifies the type of data stored, the methods usable on them, and the parameters of these methods. ADTs are expressed in Java as interfaces. There can be several different data structures that use different implementations of the same ADT.

Analysis Tools:

Data structures are analysed and evaluated using analysis tools. There are seven functions that we use to analyse data structures which are ordered below by growth rate (where b is a positive constant).

Linear Function	$f(x) = c$
Logarithmic Function	$f(x) = \log_b x$
Linear Function	$f(x) = x$
n-log-n Function	$f(x) = x\log_2 x$
Quadratic Function	$f(x) = x^2$
Cubic Function	$f(x) = x^3$
Exponential Function	$f(x) = b^x$

Methods of Analysis:

Our aim is to design data structures with smallest running time. Generally, running time is directly proportional to the size of the input, and so we use the seven functions above to study the relationship between the running time and the size of the input for an data structure.

Experimental analysis is one method of analysing data structures that involves using the Java method 'System.currentTimeMillis()' to test the running time when the input size is at different values. This way we can create a graph of the running time in milliseconds against the input size. There are limitations to using this method though as not all input sizes can be tested, the data structure must be fully implemented, and we often cannot compare the results of two different data structures.

Another method of analysing data structures is to count the number of primitive operations the data structure performs. A primitive operation is a simple low-level instruction such as assigning a value to a variable, or calling a method. All primitive operations take approximately the same amount of time to perform, and so the amount performed is directly proportional to the running time for that input. Since data structures may run faster on some inputs than others of the same size, it is too difficult to define an accurate average running time, and so we tend to focus on the worst case.

Asymptotic Notation:

Since the running time of a data structure is dependent on the size of the input, we describe the running time using special notation. We use an 'O' to describe that the running time as a function of input size f(x) grows at a rate that is less than or equal to that of another function in an asymptotic sense like so: f(n) is O(g(n)). Another way to think of this is that we the maximum amount of operations a method would perform in the parentheses following the O. For example, a method that returns a single value from an array would have a running time of O(1) since it only performs a single operation regardless of the size of the array. However, a method that searches an array element by element may have to perform operations on every single element, so this method has a running time of O(n). Methods where the choice of the next element to perform an operation is one of several possibilities have a running time of O(log(n)), for example, a binary search.

Additionally, we can use an 'Ω' to denote that a function grows at a rate that is greater than or equal to that of another functions, and a 'Θ' to denote if two functions grow at the same rate.

We can use running times in asymptotic notation to analyse the running times of data structures in the long run. Consider the following table illustrating the growth rates of different functions.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Java Review:

Java is an object-oriented programming language. This means that a program is seen as a collection of objects (or instances) with different structures and behaviours that interact with each other. The structure and behaviour of an object is dependent on which class it belongs to.

Classes are part of a program that hold a set of instance variables and methods. Objects can be created using a special method called the 'constructor'. The class essentially defines the 'type' of an object.

Instance variables are a means of storing data of different types. These types include integers, floating-point numbers, characters, strings, boolean values and more. There are many operations we can perform on instance variables. Arrays can be used to store large amounts of instance variables of the same type.

Methods are a set of instructions that are stored in a class. Rather than writing out the same instructions over and over, we can write them once in a method and 'invoke' or 'call' the method whenever necessary to execute these instructions.

The main method is a special method written in a class that is invoked by the computer when the program is run. In this method, we specify instructions that program will perform at runtime. We manipulate the behaviour of a program depending on certain circumstances using the concept of 'control flow', which involves using if-else statements, switch statements, and loops.

One method of creating a set of classes involves defining a very general class, then creating subclasses of this class that inherit all the methods and variables of the general class (called a superclass). The subclasses are said to 'extend' the superclass. This concept is called 'inheritance'.
Another method of creating a set of classes involves creating an 'interface', which is a set of only constants and method headings and not their implementations. From here, we can create classes that 'implement' this interface, which means that they contain all the constants and implementations for all the methods.

To account for any unexpected errors, like accidentally dividing by zero or storing a data value in an instance variable of the wrong type, we use 'exceptions', which are a mechanism of handling an error.

Recursion:

Recursion occurs when a method calls itself. To avoid an infinite loop occurring, recursive methods have a 'base case', which are an instance where the method stops calling itself. Recursion can be very useful but is not always ideal, and as such it is often replaced with loops.

Linear recursion is the simplest form of recursion. Methods of linear recursion make one recursive call each time it is invoked until the base case. This type of recursion is useful when we are performing an algorithm on a set of elements. We say that 'tail recursion' occurs when a linearly recursive method's last operation is a recursive call. This can occur when we run out of elements in an array to perform recursive calls on.

Binary recursion is when a method makes two recursive calls. A method used to calculate the k^{th} Fibonacci number returns a sum of two recursive calls, the first for the $k\text{-}1^{th}$ Fibonacci number and the second for the $k\text{-}2^{th}$ Fibonacci number.

If a method makes n recursive calls, then it runs in O(n) time. Similarly, if a method makes log(n) recursive calls, then it runs in O(log(n)) time.

Week 2

Generics:

The generics framework in Java provides us with a means of creating classes, methods, and interfaces that can accept types as parameters. This means that instead of writing a bunch of different classes with the same code that only differ by the types of variables used, or having to typecast frequently, we can write one generic class that we can input different types into as parameters to create different objects with the same methods.

We declare a generic class or interface the same way we declare we usually do except that we put a parameter list in angle brackets right after the name. Consider the following example.

```
public class Pair<Key, Value>{
    Key key;
    Value value;

    public void set(Key k, Value v){
        this.key = k;
        this.value = v;
    }
}
```

Now we can create objects of this class that use different types by using the angle brackets. These objects can both use the set method.

```
Pair<String, Integer> pair1 = new Pair<String, Integer>();
Pair<Character, Double> pair2 = new Pair<Character, Double>();
```

We can also restrict what type of parameter is input into a generic class by using an extends clause. For example, using 'F extends Fruit' in the angle brackets will only allow you to input classes that extend the class Fruit into the generic class.

To declare a generic method, we put the parameter list in angle brackets before the return type in the method heading. This is among the method modifiers like public and static. For example, 'public static <F extends Fruit> int pickFruit(F)' is a valid declaration for the generic method 'pickFruit'.

Sequence ADTs:

Traditionally, if we have a collection of data objects in a sequence our first course of action is to store them in an array. However, there are many benefits of designing an abstract data type to handle the ordered collection. The main benefit being that many different systems can use different implementations of the same ADT with different performance tradeoffs. This way we can decide which implementation will best suit the data.

The Array List ADT:

A sequence that supports access to elements by index is called an 'array list'. An example of an ADT that handles ordered collections of data is the Array List ADT. This is expressed as a generic interface in Java that contains mainly the methods in the table below. Note that because this interface uses generics, we can use this single interface to create different lists that hold different types of values.

METHOD	OPERATION	RUNNING TIME
isEmpty()	Returns a true boolean value if the list is empty, otherwise it returns a false boolean value.	O(1) because this method only performs one operation regardless of the size of the list
size()	Returns an integer denoting the amount of elements in the list.	O(1) because this method only performs one operation regardless of the size of the list
get(i)	Returns the element in the list with the index i.	O(1) because this method only performs one operation regardless of the size of the list
set(i, e)	Set the element in the list with the index i as e. The element written over is returned.	O(1) because this method only performs one operation regardless of the size of the list
add(i, e)	Increments the index of all the elements with indices greater than equal to i by one to create a space, then puts the value e in that space.	O(n) because the running time of this method differs depending on the size of the list.
remove(i, e)	Returns then deletes the element at index i then decrements the index of all the elements with indices greater than i by one.	O(n) because the running time of this method differs depending on the size of the list.

The add and remove methods are especially helpful because once an array is declared, its length cannot be changed, however the objects of classes implementing the list ADT are capable of calling the add and remove methods which overcome this problem.

Array Lists:

ArrayList is a collection class in the java.util library that implements the Array List ADT. This class also contains some additional methods like 'clear' which removes all the elements from the list, and 'toArray' which returns an array that contains the ordered elements of the list. The ArrayList class still stores elements in an array, however, when the array is full and we wish to add another element, a new array of twice the size is created and the elements of the old array are copied to it before we can add more elements. This is known as an 'extendable array'.

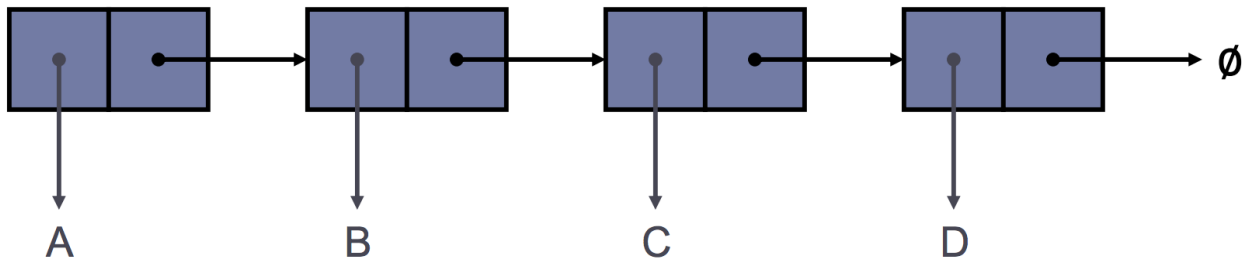
The Positional List ADT:

Another method of organising a sequence of data is to use a position-based list. Instead of accessing elements based on their index, a position-based list will access elements based on their position relative to other elements in the list. The position of an element is defined by its neighbours: the element before and the element after. We use a cursor or a pointer to single out an element. If we know the position of the first element, we can find all the other elements. In Java, the Positional List ADT is a generic interface that mainly contains the methods outlined in the following table. Note that the running times in this table refer to those of a doubly linked list (the main type of positional list).

METHOD	OPERATION	RUNNING TIME
first()	Returns the first element in the list.	O(1)
last()	Returns the last element in the list.	O(1)
prev(p)	Returns the element before the position p in the list.	O(1)
next(p)	Returns the element after the position p in the list.	O(1)
set(p, e)	Replaces the element at position p in the list with e.	O(1)
addFirst(e)	Inserts a new element e at the beginning of the list.	O(1)
addLast(e)	Inserts a new element e at the end of the list.	O(1)
addBefore(p, e)	Inserts a new element e before the element with position p.	O(1)
addAfter(p, e)	Inserts a new element e after the element with position p.	O(1)
remove(p)	Removes and returns the element with position p in the list.	O(1)

Linked Lists:

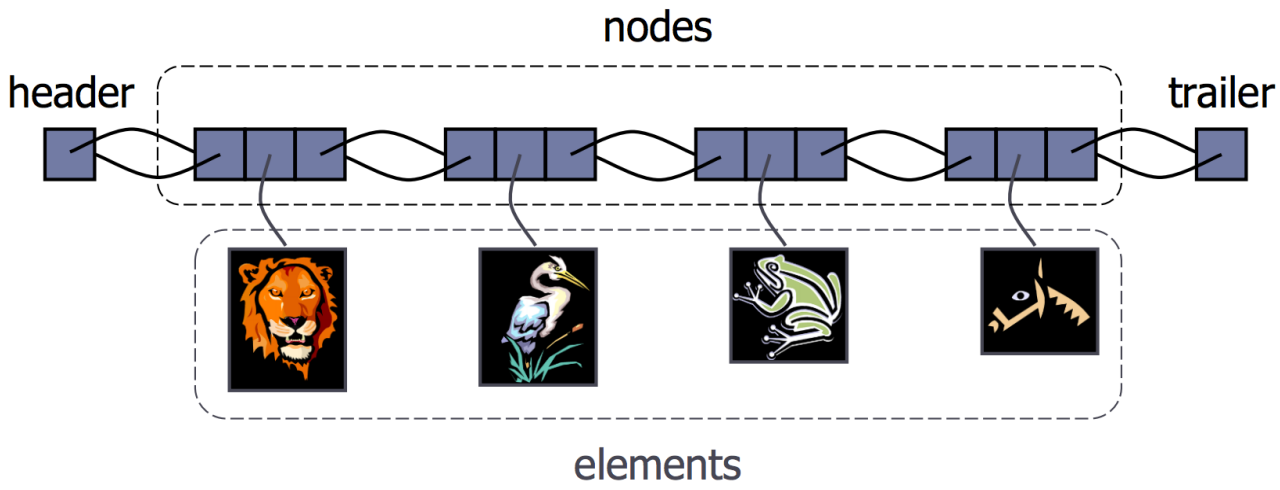
LinkedList is a collection class that implements the Positional List ADT. A linked list is a data structure consisting of a set of nodes that are linked to one another. A node consists of an element, and a link to the next element. When arranged as follows, we create a 'singly linked list'.



The first and last node of a linked list are called the 'head' and 'tail', respectively, and are referenced to variables with the same name. Note that the next reference of the tail of a list points to a null object. Linked lists are especially useful because the running time for most methods is O(1) regardless of the size of the list. For example, we can easily insert a node at the head of the list by creating a new node and setting its next reference to the current head, then updating the head to the new node. This method has a running time of O(1) since it takes the same amount of time regardless of the size of the list. Similar methods with a running time of O(1) can be used to return the first element, remove the first element, add an element to the end of the list, or return the last element. However, in order to remove the last element, we would have to change the tail variable to reference the second-last node. Doing this would require us to find the second-last node by running through the list from the beginning since the nodes in this list only reference the next node down the list and not the one before, therefore a method of removing the last node would have a running time of O(n).

To implement a singly linked list we have to define a Node class with a constructor that defines a reference to an element and a reference to the next node in the list. It would be wise to create the Node class using generics so we can create many different lists that store different classes of objects without having to write new classes to suit each list.

A 'doubly linked list' is created from nodes that consist of an element, a link to the next element, and a link to a previous element (called 'prev'). The LinkedList class in Java creates this type of list, not singly linked lists. Doubly linked lists also have two extra 'sentinel' nodes at the front and back of the list called the 'header' and 'trailer', respectively, that store no element. The sentinel nodes are not counted when determining the size of a list, meaning that an empty list contains only the header and trailer. Also, the prev reference of the header and the next reference of the trailer are linked to null objects and are often omitted from diagrams.



Doubly linked lists enable more operations to be performed efficiently in comparison to a singly linked list, however, doubly linked lists take up more space. If we do not need the efficient operations exclusive to a doubly linked list, we should use a singly linked list which is simpler and smaller. Adding and removing nodes anywhere in the list is simply a matter of updating the next and prev references of the nodes around the insertion or removal. These methods have a running time of O(1).

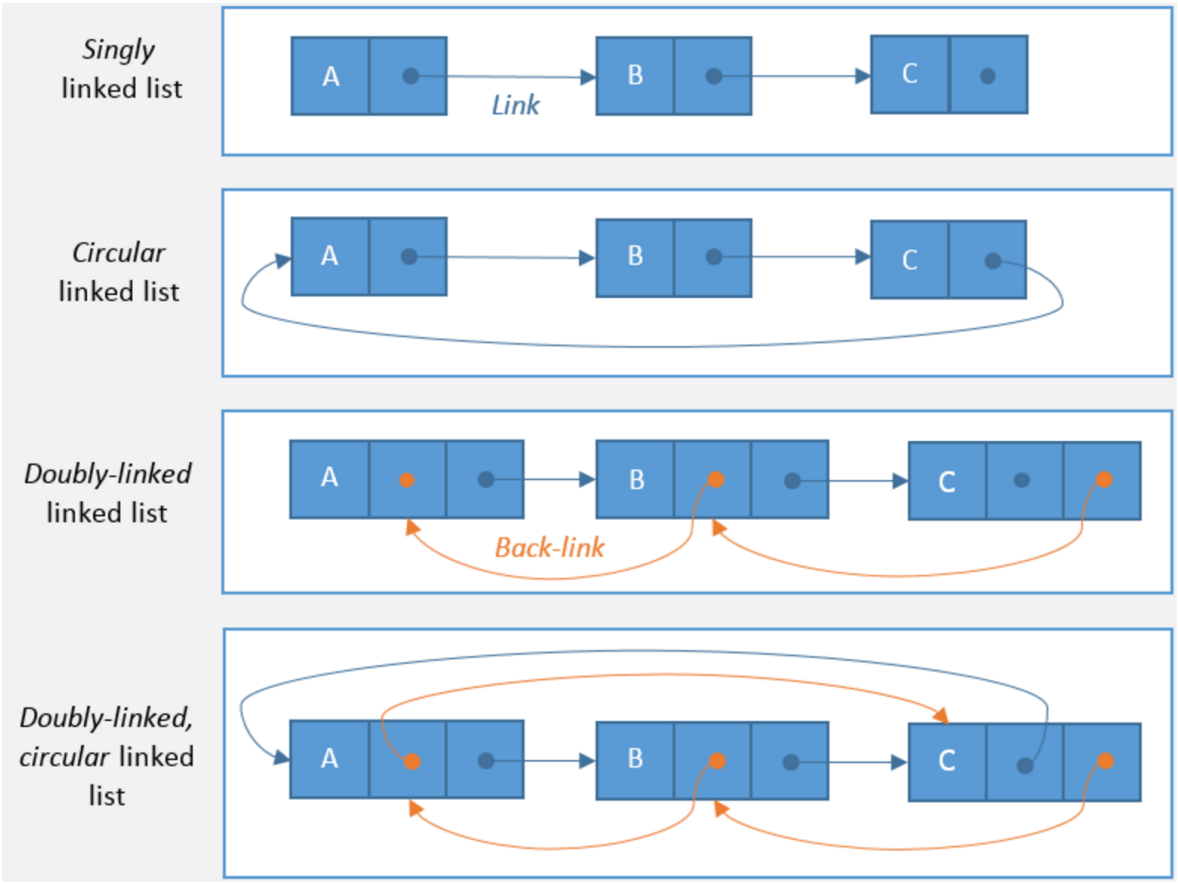
The way we implement a doubly linked list is similar to how we implement a singly linked list except that the Node class contains an extra instance variable to reference the previous node.

We can also create singly-linked circular lists and doubly-linked circular lists. In these lists, the references of the head and tail of the list refer to each other, so there is no null pointer.

The Iterator ADT:

Iterators are similar to position-based lists except that iterators contain a type of cursor or pointer that select one element as the current element. We use iterators to cycle through the elements in a list. This is useful for displaying all the elements in a collection, etc. The Iterator ADT contains two methods: 'hasNext' which tests if there are any elements after the current element, and 'next' which returns the element after the current element. In order to use iterators, a sequence ADT like the Positional List ADT has to implement the Iterable ADT. The Iterable ADT is an interface with only one method: 'iterator' which returns an iterator for all the elements in a collection.

To use iterators, Java provides us with a special loop called the 'for-each loop' that can perform statements for each element in a list returned by an iterator. The syntax is as follows.




```
for (list_type variable_name : list){
    statements
}
```

Here the statements will be repeated for each element in the list (where the list implements the Iterable ADT). This syntax is a shorthand Java provides us with for an actual for-loop that uses iterators.

The 'listIterator' method that is implemented in many lists Java provides us with produces an iterator that can traverse forward and backward. This iterator is can be thought of as a 'list cursor'. We can use methods like add, previous, next, set, and more to use and manipulate lists based on the position of the list cursor.

Lists in the Java Collections Framework:

The Java Collections Framework is a set of classes and interfaces that Java provides us with that allow us to define and implement various data structures for storing collections of data. The root interface of these data structures is java.util.Collection. The Collection interface extends the Iterable interface, which means that it contains the method 'iterator'. Also, the Collection interface is extended by an interface called 'List' which is implemented by the classes ArrayList and LinkedList.