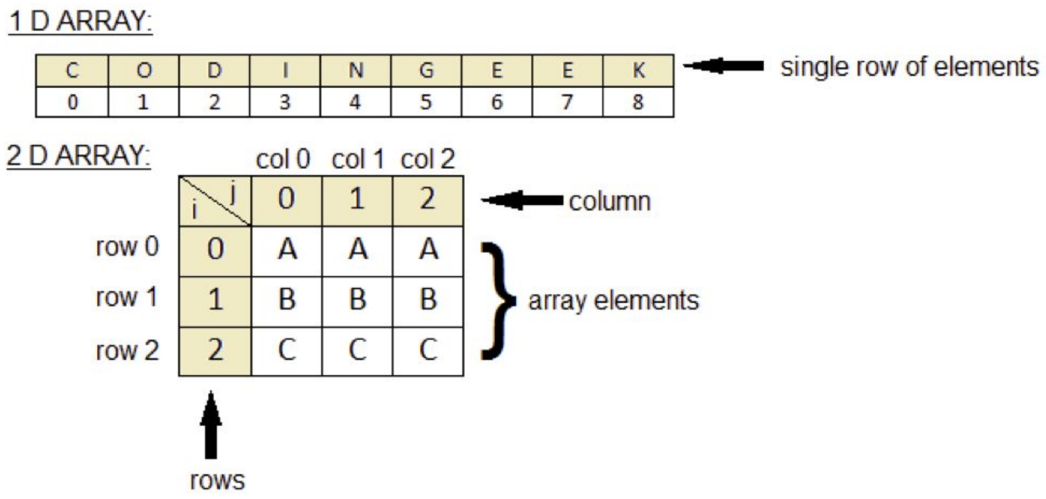
Unit 5: Abstract data structures

1. Two dimensional arrays

As the name suggests, a two dimensional array will allow us to model data that is two dimensional in nature. Programming uses for this include spreadsheets, databases and 2D games.



As with one dimensional arrays, we have a couple of methods of declaring static 2D arrays with Java.

Method 1

```
int numbers[][]= { {25,10, 5},  //row 0
                   { 4, 6,13},  //row 1
                   {45,90,78}   //row 2
                 };

for (int row=0; row<numbers.length; row++) {
    for (int col=0; col<numbers[row].length; col++) {
        int val = numbers[row][col];
        System.out.println("numbers["+row+"]["+col+"] = "+val);
    }
}
```

Method 2

```
int numbers[][] = new int[3][3];

numbers[0][0] = 25;
numbers[0][1] = 10;
numbers[0][2] = 5;
numbers[1][0] = 4;
numbers[1][1] = 6;
numbers[1][2] = 13;
numbers[2][0] = 45;
numbers[2][1] = 90;
```

```
numbers[2][2] = 78;

// using the other for-loop method
for (int[] row : numbers) {
    for (int cell : row) {
        System.out.println( cell );
    }
}
```

2D array questions

A teacher has decided to use a 2D array to store the marks for one of their classes. The grade book takes the following form:

Marksbook	Test 1	Test 2	Test 3	Test 4	Test 5
Student A	67%	50%	93%	83%	43%
Student B	70%	52%	96%	85%	48%
Student C	90%	81%	100%	93%	68%
Student D	55%	32%	71%	72%	58%
Student E	60%	47%	65%	74%	61%

Convert the above into a suitable 2D array then write code to determine the following

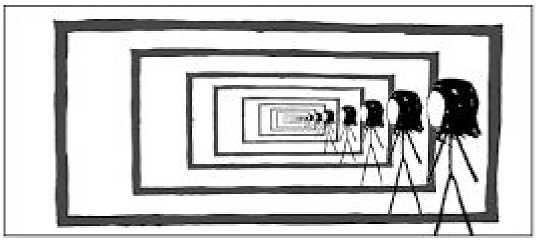
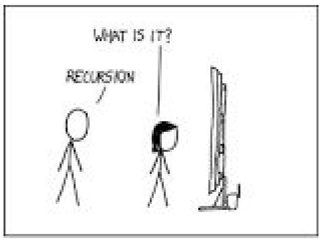
1. Determine the overall average mark
2. Determine the average mark for each individual student
3. Determine the average mark for each individual assessment
4. Given the following grade cutoffs, determine the grade for each student: A = 85%, B = 70%, C = 55%, D = 40%, F < 40%

TODO:

- Additional exercises needed. Something using spreadsheet data?, a 2D game grid?

2. Recursion

To understand recursion, one must first understand recursion.

Recursion defines the solution to a problem in terms of itself. It is used to create looping behaviour without actually using a loop construct. To that end, any recursive algorithm can be solved iteratively, and vice-versa.

A simple example to illustrate recursion is calculating the factorial of a number. As a reminder, a factorial is the product of an integer and all the positive integers below it. The factorial of the number 4 is 4 x 3 x 2 x 1 which is 24. This may be calculated using a recursive or iterative approach as follows:

```
# Iterative solution
def factorial(n):
    value = 1
    while n > 0:
        value = value * n
        n = n - 1
    return value
```

```
# Recursive solution
def factorial(n):
    if n > 1:
        return n * factorial(n-1) # Notice the function is calling itself!
    else:
        return n
```

Every recursive algorithm has

- A **base case**: a point at which the recursion will stop because the most basic endpoint has been reached, so a simple answer can be given.
- A **test**: to determine if we have reached the base case. If we don't have a test, our recursive loop could go for infinity.

In a recursive algorithm, the computer "remembers" every previous state of the problem. This information is "held" by the computer on the "activation stack" (i.e., inside each function's memory workspace). Every function has its own workspace for every call of the function.

Recursive functionality particularly suit some types of problems. It can make algorithms a lot more elegant than the iterative equivalent. The trade off, however, is that stack space is limited and the computer will only be able to recurse so many times before it runs out of memory.

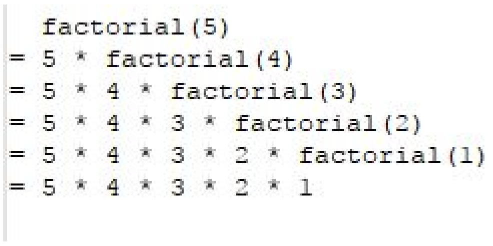
1. Identify the base case – and write the procedure for it

2. Identify the recurring case – and write the procedure for it
3. Identify the test for determining whether the present case is base or recurring
4. Code it
5. Trace test
6. Test on a small scale
7. Test on a larger scale

2.1 Factorials

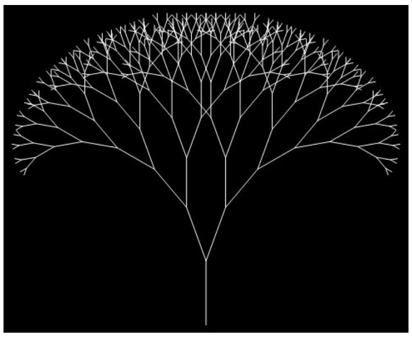
We saw the example code for factorial above.

What is the base case?
What is the test?
What is the recursive call?
How would we trace this algorithm?



```
  factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1
```

2.2 Fractal tree



The recursive steps for a fractal tree could be described as

- Draw the trunk
- At the end of the trunk, split by some angle and draw two branches
- Repeat at the end of each branch until a sufficient level of branching is reached

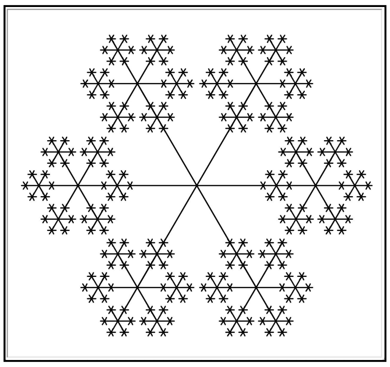
Without worrying about the programming, focusing just on the pseudo code logic, how would you create a fractal drawing algorithm?

Manually test it for depths 1, 2 and 3.

Only once you are satisfied with your pseudo code and trace table should you attempt the code implementation. This is the Java version from https://rosettacode.org/wiki/Fractal_tree#Java

```java
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JFrame;

public class FractalTree extends JFrame {
    public FractalTree() {
        super("Fractal Tree");
        setBounds(100, 100, 800, 600);
        setResizable(false);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    private void drawTree(Graphics g, int x1, int y1, double angle, int depth) {
        if (depth == 0) return;
        int x2 = x1 + (int) (Math.cos(Math.toRadians(angle)) * depth * 10.0);
        int y2 = y1 + (int) (Math.sin(Math.toRadians(angle)) * depth * 10.0);
        g.drawLine(x1, y1, x2, y2);
        drawTree(g, x2, y2, angle - 20, depth - 1);
        drawTree(g, x2, y2, angle + 20, depth - 1);
    }

    @Override
    public void paint(Graphics g) {
        g.setColor(Color.BLACK);
        drawTree(g, 400, 500, -90, 9);
    }

    public static void main(String[] args) {
        new FractalTree().setVisible(true);
    }
}
```

2.3 Snowflakes



Are you a snowflake?

The humble snowflake, is very similar to fractals and is recursive in nature.

Here is a Python implementation of a snowflake drawing algorithm. See the live demo at https://repl.it/@PaulBaumgarten/SnowflakeRecursion

```
from turtle import *

def drawFlake(length, depth):
    hideturtle()
    if depth > 0:
        for i in range(6):
            forward(length)
            drawFlake(length // 3, depth - 1)
            backward(length)
            left(60)

drawFlake(200,4)
```

2.4 Fibonacci

The fibonacci sequence is where each number is the sum of the previous two numbers.

The first few numbers in the sequence is: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144

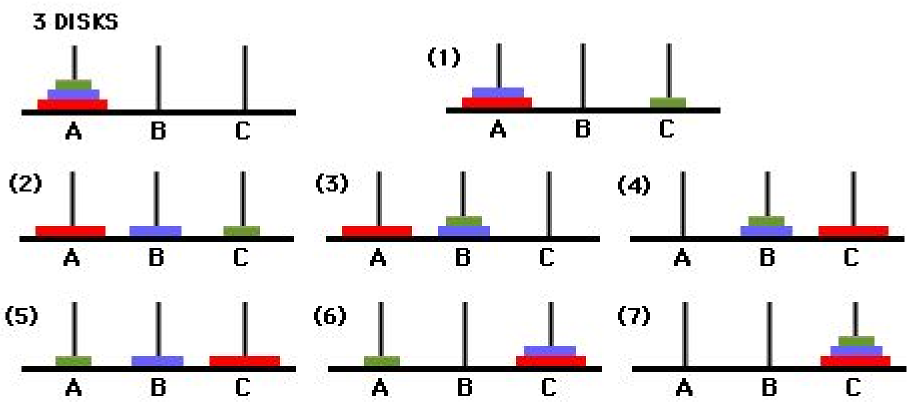
Determine the pseudo code, do a trace table test of your algorithm, then code it in Java.

2.5 Tower of hanoi

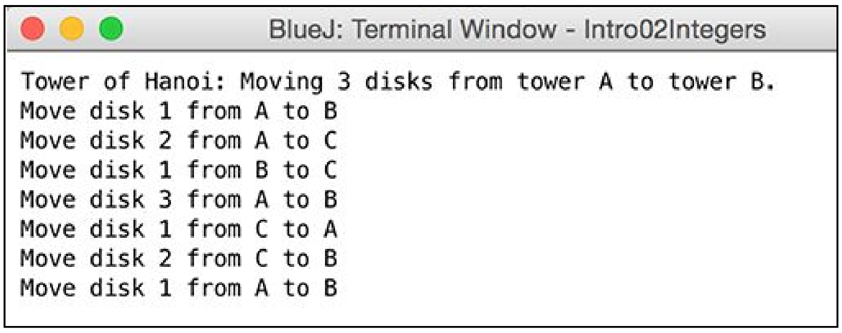
The Tower of Hanoi is a game that requires recursion to solve. In this puzzle, we have three pegs and several disks, initially stacked from largest to smallest on the left peg. The rules are:

- We are finished when the entire tower has been moved to another peg.
- We can only move one disk at a time.
- We can never place a larger disk on a smaller one.

An example of how it works with 3 disks.



You don't need to create a graphical output, a printed set of instructions is sufficient. Eg:



```
Tower of Hanoi: Moving 3 disks from tower A to tower B.
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
```

(note: you'd very quickly find solutions online... while I can't stop you, I emphasis this would deprive yourself of the learning experience the problem solving brings)

2.6 Binary search

By now we should all know and love the binary search algorithm, but did you realise there is a recursive version of the algorithm? Can you create the recursive version of the algorithm? (no cheating with google)

Here is an array of 50 sorted names you can use for your binary search.

```
String[] names =
{"Aaliyah","Abigail","Adalyn","Aiden","Alexander","Amelia","Aria","Aubrey","Ava","Avery","Benjamin","Caden","Caleb","Carter","Charlotte","Chloe","Daniel","Elijah","Emily","Emma","Ethan","Evelyn","Grayson","Harper","Isabella","Jack","Jackson","Jacob","James","Jayden","Kaylee","Layla","Liam","Lily","Logan","Lucas","Luke","Madelyn","Madison","Mason","Mia","Michael","Noah","Oliver","Olivia","Riley","Ryan","Sophia","William","Zoe"};
```

2.7 Sudoku

If you've completed a number of the above and are looking for a challenge, the popular numbers game of Sudoku is actually a recursive puzzle. Can you write an algorithm that will solve it? Check the Sudoku algorithm explainer on wikipedia.

3. Data structures overview

An abstract data structure:

- Uses the principle of abstraction. You create a model to represent data in the form you require, and hide the " behind the scenes" complexity of how it functions.
- Uses dynamic memory allocation to resize the data structure as required.

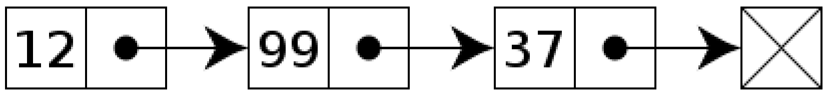
Dynamic	Static
Memory is allocated as the program runs.	Memory size is fixed, and is set at time of compilation.
Disadvantage: Possibility of overflow or underflow during runtime if allocations are exceeded.	Advantage: As memory size is fixed, there will be no problems with memory allocations during run time.
Advantage: Makes most efficient use of memory, uses only what is required.	Disadvantage: Potentially wasted memory space.
Disadvantage: Harder to program. Programmer must keep track of memory sizes and locations.	Advantage: Easier to program. Only need to check you don't exceed your preset limit.

4. Linked lists

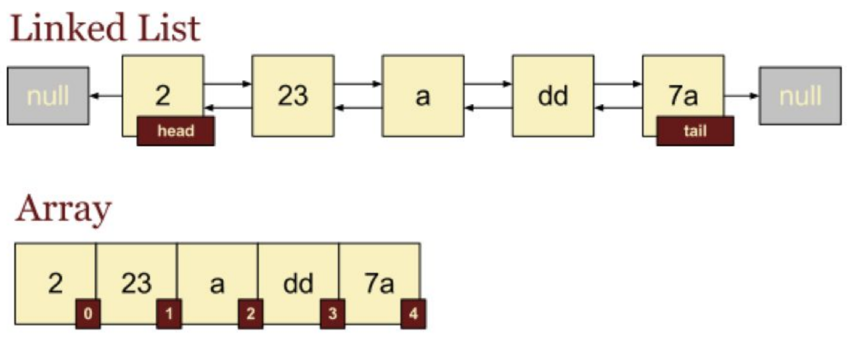
A linked list is a simple dynamic data structure that can grow and shrink according to your program's needs, giving it a significant advantage over static arrays. We will examine how a linked list works and create our own to get an understanding of the internal operations at work. The dynamic nature of linking nodes together forms the basis of all dynamic data structures.

4.1 The basic linked list

The basic linked list consists of a set of nodes. Each node has two elements: a data value, and a pointer to the next node. The following illustrates.



Comparing the model of a linked list to a static array

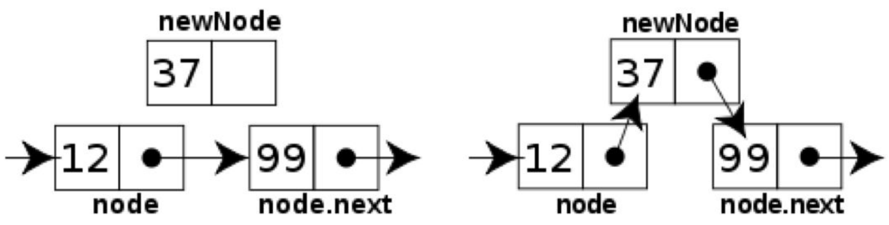


4.2 Linked list operations

Inserting a node

To insert a new node into an existing list involves:

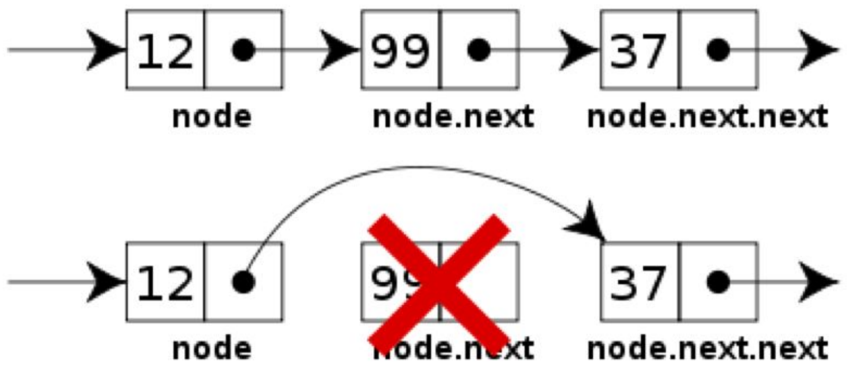
1. Create the new node with the intended value
2. Adjust the pointers of the existing nodes so it becomes part of the chain



Deleting a node

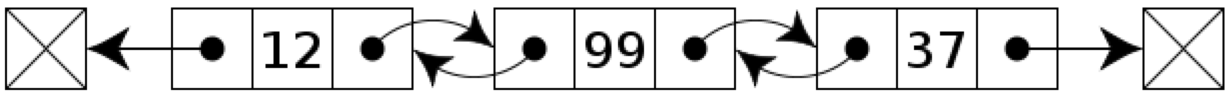
To delete a node from a list involves:

1. Save a reference to the target node into a temporary variable
2. Update the chain of node pointers so the target node is no longer in the list
3. USe the temporary reference to delete the target node from memory



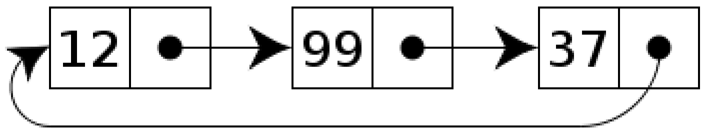
4.2 Doubly linked lists

A doubly linked list contains two pointers per node, one for *next* and one for *previous*.



4.3 Circular linked lists

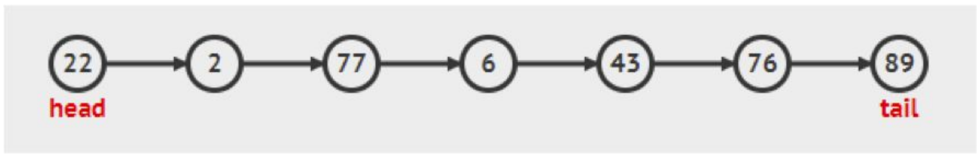
A circular linked list forms an infinite loop in it's chain by having the last node's "next" value point to the first item in the list.



4.4 Sketching and pseudocode questions

Students should be able to sketch diagrams illustrating:

- adding a data item to linked list,
- deleting specific data item,
- modifying the data held in the linked list,
- searching for a given data item.



1. Sketch adding the value 42 to the end of the list
2. Sketch deleting node with value 77
3. Sketch search for a node of value 6

TODO: More sketching exercises needed.

4.5 Programming a linked list

Once we have an understanding of the internal operations of the linked list, let's consider the external operations. The point of a linked list is to give us a dynamic alternative to a static array. Accordingly, it would be usual to want a variety of functions that allow for easy insertion and removal of items at specific points within a list.

Some common operations when using linked lists include:

- addHead(value) - add a node with this value to the front of the list
- addTail(value) - add a node with this value to the end of the list
- addAt(value, index) - add a node with this value to this specific place in the list
- add(value) - same as addTail()
- insert(value) - in order insertion
- delete(value) - find this value in the list and delete it
- list()
- getNext() - advance pointer to next item in the list
- resetNext() - returns to the beginning of the list
- hasNext() - returns true/false if there is a next item
- isEmpty()
- isFull()

The Node class forms the basis of the internal data structure for the linked list.

```java
public class Node {
    private int value;
    private Node right;
    private Node left;

    Node(int value) {
        this.value = value;
        this.next = null;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

    public Node getNext() {
        return next;
    }

    public void setNext(Node next) {
        this.next = next;
    }

    public boolean hasNext() {
        return (next != null);
    }
}
```

The LinkedList class is a container for Node objects, and contains methods for the programmer to use to access and mutate the data in a more organised way, similar to an ArrayList.

```java
public class LinkedList {
    private Node head;
    private int listCount;

    public LinkedList() {
        head = new Node(-1); // Dummy value that we will just ignore
        listCount = 0;
    }

    public void addTail(int val) {
        // TODO: Code this yourself
    }
    public void addHead(int val) {
        // TODO: Code this yourself
    }

    public void addSorted(int val) {
        Node current = head;
        while (current.hasNext() && (current.getNext().getValue() < val)) {
            current = current.getNext();
        }
        Node tmp = current.getNext();
        current.setNext( new Node(val) );
        current.getNext().setNext(tmp);
    }

    public int get(int index) {
        // TODO: Code this yourself
        return 0;
    }

    public boolean remove(int index) {
        // TODO: Code this yourself
        return false;
    }

    public int size() {
        return listCount;
    }

    public String toString() {
        Node current = head;
        String output = "LinkedList[ ";
        while (current.hasNext()) {
            current = current.getNext();
            output += Integer.toString(current.getValue())+" ";
        }
        output += "]";
        return output;
    }
}
```

Once the LinkedList class is complete, it's functionality becomes available for use as a general data structure as per this example.

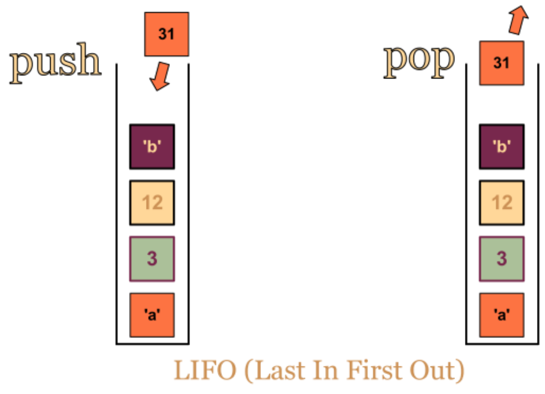
```java
class Main {
  public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.addSorted(5);
    list.addSorted(10);
    list.addSorted(15);
    list.addSorted(30);
    list.addSorted(40);
    list.addSorted(45);
    list.addSorted(20);
    System.out.println(list);
  }
}
```

4.6 Programming questions

TODO

5. Stacks & queues

Building on the idea of the linked list, stacks and queues are semi-specialised forms of dynamic data structures.



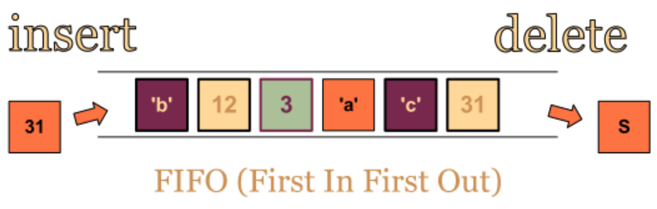
A stack is a LIFO (last in, first out) data structure with the following methods:

- push() to add to the stack
- pop() to remove from the stack
- isEmpty() to query stack status

A real world example of the stack is your undo buffer in an editing program.

```
import java.util.LinkedList;

public class MyStack {
    private LinkedList list;

    public MyStack() {
        list = new LinkedList();
    }

    public boolean isEmpty() {
        return (list.size() == 0);
    }

    public void push(Object item) {
        list.add(item);
    }

    public Object pop() {
        Object item = list.get(list.size());
        list.remove(list.size());
        return item;
    }
}
```

A queue is a FIFO (first in, first out) data structure:



The standard functions for interfacing with a queue are:

- enqueue() to add an item to the back of the queue.
- dequeue() to remove an item from the front of the queue.
- isEmpty() to query the queue status.

Real world examples of the queue are your networking buffer, keyboard buffer or printing queue.

```java
import java.util.LinkedList;

public class MyQueue {
    // Use the Java built in LinkedList to manage our queue
    private LinkedList list;

    public MyQueue() {
        list = new LinkedList();
    }

    public boolean isEmpty() {
        return (list.size() == 0);
    }

    public void enqueue(Object item) {
        list.add(item);
    }

    public Object dequeue() {
        Object item = list.get(0);
        list.remove(0);
        return item;
    }

    public Object peek() {
        return list.get(0);
    }

    public static void main(String[] args) {
        MyQueue q = new MyQueue();
        System.out.println( "Queue:" );
        q.enqueue("person 1");
        q.enqueue("person 2");
        q.enqueue("person 3");
        q.enqueue("person 4");
        q.dequeue();
        q.enqueue("person 5");
        q.enqueue("person 6");
        while (! q.isEmpty() ){
            int i = 1;
            System.out.println( "Now calling customer " + q.dequeue() );
        }
    }
}
```

If we wanted to implement our own stack or queue, in addition to using a dynamic structure such as the LinkedList like we used above, we can make one quite easily using an object that used a static array as an instance variable.

What would be required of our object to be able to implement this?

Static stack & queue task

Build the classes StaticStack and StaticQueue that use a statically declared array as the data store. Use an array size of 100 items. Your class should only publicly expose the methods of a queue or stack, with the addition of an isFull() method.

Stacks & queues programming questions

1. For any given string, reverse its contents by way of using a stack.

2. For any given string, use a stack to determine if every opening parenthesis is matched with a closing parenthesis.

3. What does the following code fragment print when n is 50? Give a high-level description of what the code fragment does when presented with a positive integer n.

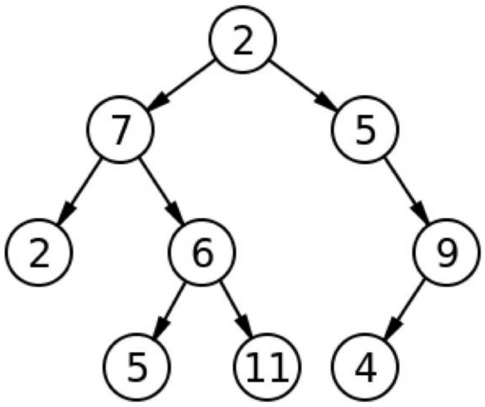
```
Stack stack = new Stack();
while (n > 0) {
    stack.push(n % 2);
    n /= 2;
}
while (!stack.isEmpty()) {
     print(stack.pop());
}
println();
```

4. Consider the following pseudo code. Assume that MyQueue is an integer queue. What does the function `func()` do?

```
void func(int n) {
    MyQueue q = new MyQueue();
    q.enqueue(0);
    q.enqueue(1);
    for (int i = 0; i < n; i++) {
        int a = q.dequeue();
        int b = q.dequeue();
        q.enqueue(b);
        q.enqueue(a + b);
        print(a);
    }
}
```

5. (challenging) For any given string, use a stack to create a PEMDAS compliant calculator. For example, `calculate("( 2 + ( ( 3 + 4 ) * ( 5 * 6 ) ) )");`

6. Binary trees



Trees are a commonly used data structure in computing. One place you will have used them all the time without even a moment's thought is when navigating the folder/file structure of your computer.

This course only requires you to be familiar with the binary tree, a tree that has no more than two branches coming off each node.

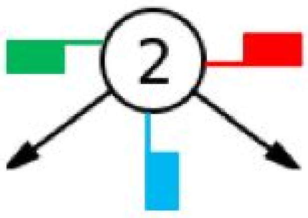
Terminology pertaining to binary trees you need to know

- Left child
- Right child
- Parent
- Sub tree
- Root
- Leaf
- Height of a node: The length of the longest downward path to a leaf from that node.
- Depth of a node: The length of the path to its root (i.e., its root path)

There are three ways of traversing a tree, starting from the root:

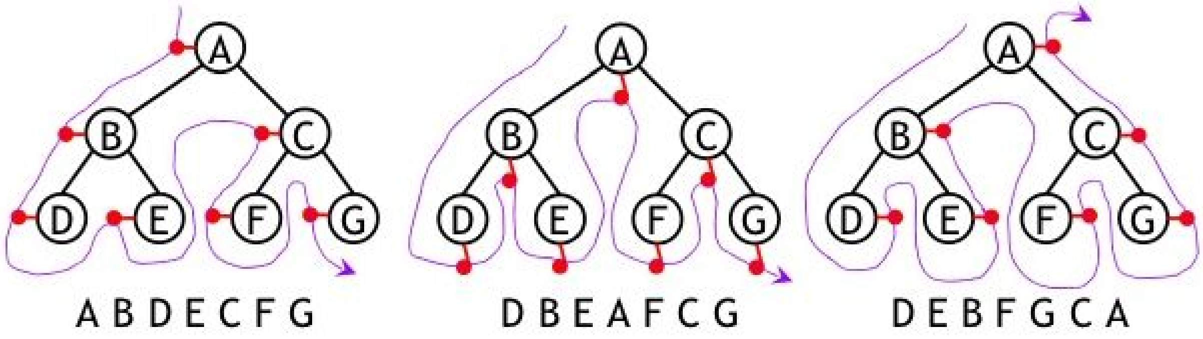
- Pre order: Print the node, visit the left node, visit the right node
- In order: Visit the left node, print the node, visit the right node
- Post order: Visit the left node, visit the right node, print the node.

A simple way of visually remembering these traversal methods is to imagine flags as follows



- Pre order: The order in which you visit the green flags
- In order: The order in which you visit the blue flags
- Post order: The order in which you visit the red flags

Example



So, with the original tree shown at the start (root node = 2), what would the pre-order, in-order and postorder traversal be?

BInary tree programming

Create the MyBinaryTree class such that the following would work...

```
public static void main(String[] args) {
    MyBinaryTree bt = new MyBinaryTree();
    bt.insert( 13 );
    bt.insert( 4 );
    bt.insert( 2 );
    bt.insert( 15 );
    bt.insert( 100 );
    bt.insert( 1000 );
    bt.insert( 222 );
    bt.insert( 23 );
    bt.insert( 7 );
    bt.insert( 8 );
    System.out.println("Node count");
    System.out.println( bt.countNodes() );
    System.out.println("Pre order traversal");
    bt.preorder();
    System.out.println("In order traversal");
    bt.inorder();
    System.out.println("Post order traversal");
    bt.postorder();
}
```

Binary tree programming questions

1. Program the function inorder() to output the data contained in a binary tree using inorder tree traversal
2. Program the function preorder() to output the data contained in a binary tree using preorder tree traversal
3. Program the function postorder() to output the data contained in a binary tree using postorder tree traversal
4. Program the function lookup(), which given a value, will search the binary tree to determine if the value is present in the tree, and returns true or false accordingly (you may assume the contents of the tree are sorted in order)
5. Program the function insert(), which given a value, will insert a new node with the given value at the correct location within the tree, and shuffles the rest of the tree accordingly as required.
6. Program the function maxdepth() which returns the longest path from the root node to the furthest leaf node.

7. Program the function isInOrder() which searches the contents of the tree and returns true if the contents are sorted for in order traversal.

(when you are ready, ask me for the solutions file, "cs-unit-5-binarytree-problems-with-solutions.pdf")

Past paper questions for review

(refer to separate document)