Data Structures

© Martin Gonzalez-Rodriguez, PhD.

www.martin-gonzalez.es

Lecture 1

Algorithmics

“Geeks are people who love something so much that all the details matter”

Melissa Meyer

The Algorithms Class

The Algorithms class contains sample algorithms whose temporal complexity will be evaluated.

1. Design a O(n) method called ‘linear’, for a given workload called ‘n’.

```
public static void linear (long n)
 {
  for (long i=0; i<=n; i++)
       System.out.println (“executing job: “ + i);
 }
```

The TestBench Class

The TestBench class is used to evaluate the performance of generic algorithms

1. Add a method called 'test':

```
public static void test (long n)
{
  // executes an algorithm for a given workload 'n'...
  Algorithms.linear(n);
}
```

2. Invoke the test method from your *main* method (TestBench class)

```
TestBench.test(75000);
```

The TestBench Class

The ‘test’ method works as a clockwatch being used to record execution times.

1. Use the following method in ‘test’ to estimate the algorithm execution time:

```
System.currentTimeMillis();
```

The TestBench Class

The execution needs to be slowed down to make its execution time noticeable

1. Create a 'doNothing' method in the TestBench class.

```
public static final int SLEEP_TIME = 25;

public static void doNothing(long i)
 {
  System.out.println ("Doing nothing at iteration...
 ("+i+")");

  long endTime = System.currentTimeMillis() + SLEEP_TIME;
  while (System.currentTimeMillis() < endTime)
  {
      // do nothing
  }
 }
```

The TestBench Class

Slow down every algorithm in the Algorithms class.

1. Invoke the 'doNothing' method for each iteration of the algorithms.

```
public static void linear (long n)
 {
  for (long i=0; i<=n; i++)
      TestBench.doNothing(i);
 }
```

The TestBench Class

The ‘test’ method measures execution times.

1. Save the result of each test in a text file using a given range for the workload [startN, endN]:

```
public void test (String outputFileName, int startN, int
endN)
```

2. Output example:

```
TestBench.test(‘linear.txt’, 0, 4);

25
50
75
100
125
```

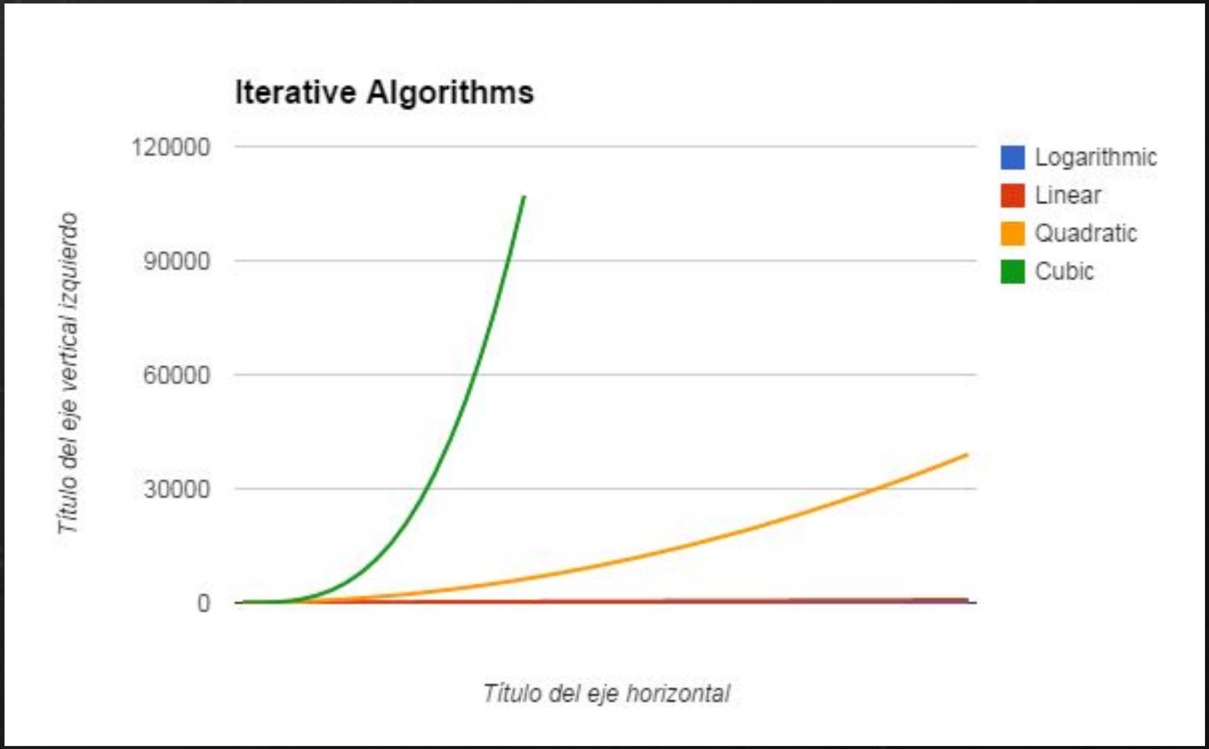
Working with Text Files (example)

```
{
  FileWriter file = null;
  PrintWriter pw = null;

  try  {
   file = new FileWriter(‘my file.txt’);
   pw = new PrintWriter(file);
   pw.println("250”);
   pw.println("1016”);
  } catch (Exception e) {
      e.printStackTrace();
  } finally {
     try {
       if (file != null)
           file.close();
     } catch (Exception e2) {
         e2.printStackTrace();
   }
}
```

The TestBench Class

Process the results in a spreadsheet.

1. Use Google Spreadsheets to plot a chart for the linear algorithm execution time.



HOMEWORK (1/3)

Improve the ‘test’ method to record several samples.

1. Add a new ‘samples’ parameter to the ‘test’ method.

```
public void test (String outputFileName, int samples, int
startN, int endN)
```

2. It indicates the number of times the experiment must be repeated.

   a. Saves the average execution time in the output text file.

HOMEWORK (2/3)

1. Develop quadratic, cubic and logarithmic algorithms in your Algorithms Class.

2. Test their performance using:

   a. Samples = 3;
   b. startN = 1;
   c. endN = 50; Use 20 for the cubic algorithm.
   d. SLEEP_TIME = 2;

HOMEWORK (3/3)

1. Research about computational Reflection in the Wikipedia. Get some code examples in the Internet.

2. [Optional] Implement the next method in the TestBench class:

```
public static void testAlgorithm (String className, String
methodName, long n) throws Exception;

// This method creates a new object of class ‘className’ at
// execution time, invoking its method ‘methodName’ providing
// ‘n’ as its only parameter.

// eg. TestBench.testAlgorithm (“Algorithms”, “linear”, 50);
```

Lecture 2

Recursion

“To iterate is human, to recurse divine”

L. Peter Deutsch

Recursion

Add a new method to the Algorithms class to calculate n! using an iterative algorithm

```
public static long factorial (long n)
{
  …
}
```

Implement the method and test it.

```
assertEquals (720, Algorithms.factorial(6));
```

Recursion

Add this recursive version of the same algorithm.

```
// Maths! Factorial definition
// 0! = 1
// n! = n(n-1)!

public static long factorialRec (long n)
{
  if (n == 0)
       return 1;
 else
       return n * factorialRec(n-1);
}
```

Test it.

```
assertEquals (720, Algorithms.factorialRec(6));
```

Temporal Complexity

Implement a new iterative method in the Algorithms class to calculate the power function of n (2^n).

```
public static long pow(long n)
{
  ...
}
```

Implement the method and test it.

```
assertEquals (1099511627776L, Algorithms.pow(40));
```

What is the temporal complexity of your algorithm?

Temporal Complexity

Implement a recursive method in the Algorithms class to get the power function of n (2^n).

```
// Maths! Power function
// 2^0 = 1
// 2^n = 2^(n-1) + 2^(n-1)

public static long powRec1(long n)
{
  ...
}
```

Implement it and test it (if your dare!)

```
assertEquals (1099511627776L, Algorithms.powRec1(40));
```

Temporal Complexity

Optimize the previous version to execute only one recursive call per iteration.

```
public static long powRec2(long n)
{
  ...
}
```

You may test it again for n == 40.

```
assertEquals (1099511627776L, Algorithms.powRec2(40));
```

Temporal Complexity

Implement another recursive version based on this new definition of 2^n.

```
// Maths! Power function
// 2^0 = 1
// 2^n = 2^(n/2) * 2^(n/2)

public static long powRec3(long n)
{
  ...
}
```

Rounding issue!

```
if (n % 2 != 0) // n is an odd number
    multiply the result of 2^(n/2) * 2^(n/2) by 2 before returning its
    value.
```

Temporal Complexity

Optimize the previous version to execute only one recursive call per iteration.

```
public static long powRec4(long n)
{
  ...
}
```

The testAlgorithm Method

```
public static void testAlgorithm (String className, String
methodName, long n) throws Exception
    {
        Class<?> myClass = null;
        Object myObject = null;

        //Loads the class dynamically using reflection
        myClass = Class.forName(className);
        myObject = myClass.newInstance();

        //Gets a method instance
        Class<?>[] params=new Class[1];
        params[0]=Long.TYPE;

        Method m = myClass.getMethod(methodName, params);

        //Calls the method in java using reflection
        m.invoke(myObject, n);
    }
```

HOMEWORK (1/4)

1. Refactor the TestBench class to use computational reflection.

   a. The testAlgorithm() method should be now used inside the test() method in the TestBench class.

2. Repeat all the experiments of the previous homework using the new reflection-based version.

   a. Plot the corresponding charts using a spreadsheet.

HOMEWORK (2/4)

1. Use the TestBench class to evaluate the performance of the pow recursive methods with those parameters:

   a. Samples = 3;
   b. startN = 1;
   c. endN = 50; Use 12 for the the exponential version.
   d. SLEEP_TIME = 2;

2. Use a spreadsheet to plot a chart displaying the execution times for the different versions of pow.

HOMEWORK (3/4)

Create a generic GraphNode class with these properties.

```
private T element;
private boolean visited;
```

Implement getters and setters methods for each property as well as the next methods:

```
public String toString();
public void print();
```

HOMEWORK (4/4)

Format the output of the GraphNode toString() method to use this pattern:

```
//eg char element 'a' not already visited
GN(N:a/V:false)
```

Lecture 3

Generic Programming

“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live”

Martin Golding

Generic Programming

Create a 'Container' class to store a String 'element' as its unique property.

```
public class Container {

private String element; //
…
}
```

Implement and test the following methods:

```
public void setElement (String element);
public String getElement();
public String toString();
```

Generic Programming

Turn the ‘Container’ class into a generic T Class.

```
public class Container <T> {

private T element; //
…
}
```

Implement and test the following methods:

```
public void setElement (T element);
public T getElement();
public String toString();
```

Generic Programming

Create a 'GenericDataStructure' class to store Strings using a LinkedList.

```
public class GenericDataStructure
{
 private LinkedList<String> collection;
 …
}
```

Implement and test the following methods:

```
public void add(String element);
public String toString ();
public int compareTwoElements (int firstPos, int secondPos);
// returns -1 if the element in the 'firstPos' is lower than
 // the element in the 'secondPos'. Returns 0 if both are the
 // same and 1 if the second one is bigger than the first one.
```

Generic Programming

Turn the 'GenericDataStructure' class into a generic collection.

```
public class GenericDataStructure <T extends Comparable<T>>
{
 private LinkedList<T> collection;
 …
}
```

Implement and test the following methods:

```
public void add(T element);
public String toString ();
public int compareTwoElements (int firstPos, int secondPos);
// returns -1 if the element in the 'firstPos' is lower than
 // the element in the 'secondPos'. Returns 0 if both are the
 // same and 1 if the second is bigger than the first one.
```

HOMEWORK (1/3)

Create a generic GraphNode class with these properties.

```
    private T element;
    private boolean visited;
```

Implement getters and setters methods for each property as well as the next methods:

```
    public String toString();
    public void print();
```

HOMEWORK (2/3)

Format the output of the GraphNode toString() method to use this pattern:

```
//eg char element ‘a’ not already visited
GN(N:a/V:false)
```

HOMEWORK (3/3)

Create a 'SuperContainer' class to store a Container, a generic 'info' object and a int 'key' value.

```
public class SuperContainer <T, K> {

 private Container<T> container;
 private K info;
 private int key;
 …
}
```

Implement and test the setters and getters for each property and the toString() method.

Lecture 3 Evaluation

1. Make sure that the GraphNode's toString() method uses this output pattern:

```
//eg char element 'a' not already visited
GN(N:a/V:false)
```

2. Import and execute the GraphNodeTest file.

Lecture 4

Graphs

“I think computer science, by and large, is still stuck in the Modern age”

Larry Wall

Adjacency Matrix

Create a generic Graph class using the following data structures.

```
ArrayList<GraphNode<T>> nodes;
protected boolean[][] edges;
protected double[][] weight;
```

Implement protected getters for these properties (they will be used for testing purposes in JUnits).

Write the graph constructor:

```
public Graph(int n)
```

Essential Methods (1/2)

Implement the next methods

```
public int getNode (T element);
// Returns -1 (INDEX_NOT_FOUND) if the node is not found

public int getSize ();

public void addNode (T element) throws Exception

public boolean existsEdge (T origin, T dest) throws Exception

public void addEdge (T origin, T dest, double weight) throws
 Exception
```

Test every method using the corresponding JUnit. You may use the L4_Graph_Sample test file as an example.

Essential Methods (2/2)

Implement and test the following methods

```
public void removeEdge (T origin, T dest) throws Exception

public void removeNode (T element) throws Exception

public void print()
```

HOMEWORK

Implement the following methods

```
public boolean isDrainNode(T element)

public boolean isSourceNode(T element)

public int countDrainNodes ()

public int countSourceNodes()
```

Lecture 5

The Floyd Algorithm

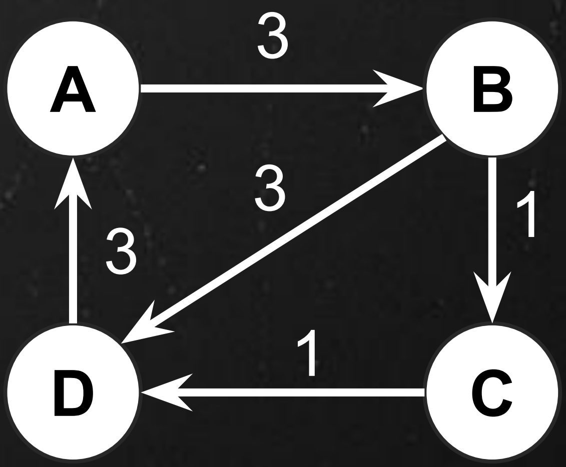
“Computer Science is a science of abstraction, creating the right model for a problem”

Alfred Aho

Traverse Methods

You may use this next JUnit as a guide for testing the methods to be developed during this lesson.

L5_Graph_Floyd_DFP_sampleTest



Traverse Methods

Test the navigation methods using the corresponding JUnit.

```
// Example
assertEquals("a-b-c-d-", g1.traverseGraphDF('a'));
assertEquals("b-c-d-a-", g1.traverseGraphDF('b'));
assertEquals("c-d-a-b-", g1.traverseGraphDF('c'));
assertEquals("d-a-b-c-", g1.traverseGraphDF('d'));
```

Graph Navigation

1. Implement the next PUBLIC navigation method...

```
public String traverseGraphDF(T element) {
  sets the 'visited' flag to false for each node.
  int v = getNode (element); //gets the node's coordinate
  if v exists return (DFPrint(v));
  else return null;}
```

2. ...and its related PRIVATE method.

```
public String DFPrint(int v) {
  sets the 'visited' flag of v to true.
 String aux = GraphNode's v element.toString()
  for each reachable node from v that has not been
  previously visited
     aux += DFPrint (that node)
  return aux;}
```

Floyd Algorithm

Implement and test the following Floyd service methods:

```
public final static double INFINITE = Double.MAX_VALUE;
public final static int EMPTY = -1;

protected double[][] getA(); //Cost matrix
protected int[][] getP(); //Pathway matrix, P[i][j]

protected void initsFloyd();
// Auxiliary method invoked whenever the actual
// Floyd method is invoked

// Reserves memory for A and P.
// Copies weight over A (placing INFINITE whenever the
// related slot in Edges is false
// Fills P with EMPTY…
// Set the cost of going from one node to itself to 0
```

Floyd Algorithm

Implement and Floyd methods:

```
public void floyd(int An)
//computes the Floyd’s cost matrix A up to the An iteration
{
    initsFloyd();

   for (int k=0; k<An; k++)
     for (int i=0; i<getSize(); i++)
         for (int j=0; j<getSize(); j++)
             if (A[i][k] + A[k][j] < A[i][j])
             {
                 A[i][j] = A[i][k] + A[k][j];
                 P[i][j] = k;
             }
}
```

HOMEWORK

Implement and test the following Floyd methods

```
public String printFloydPath (T departure, T arrival) throws
 Exception

// output example: "V1V3V5V6",

// for the call "V1" + g.printFloydPath ("V1", "V6") + "V6"
```

Lecture 6

The Dijkstra Algorithm

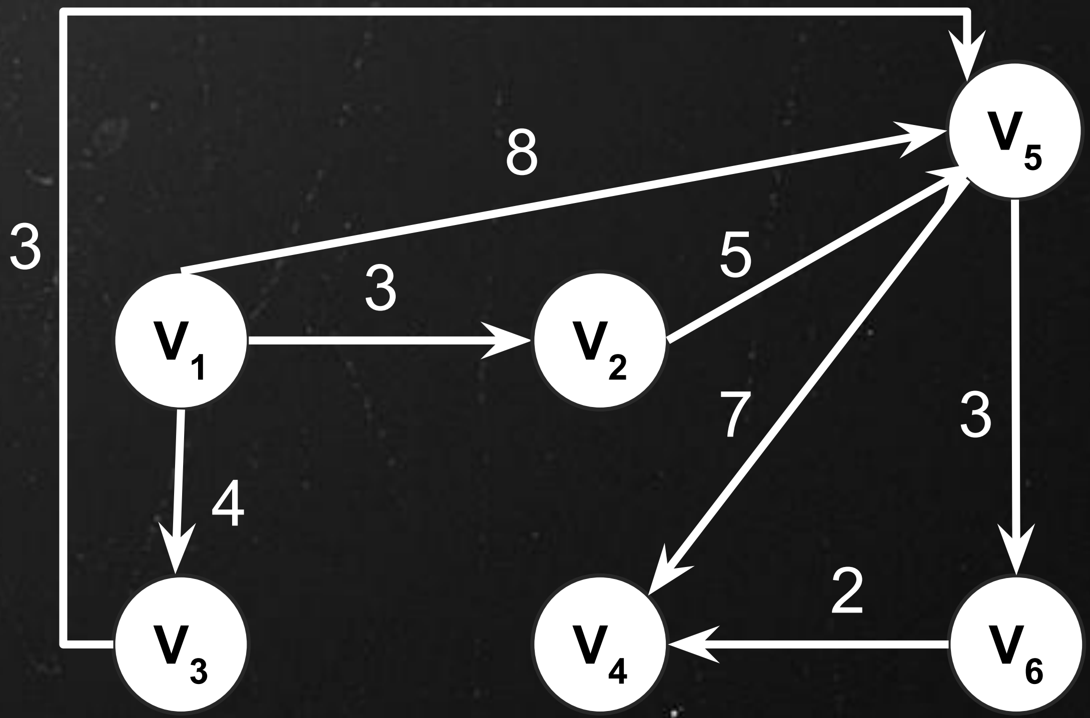
“The question of whether computers can think is like the question of whether submarines can swim”

Edsger W. Dijkstra

Test

You may use this next JUnit as a guide for testing the methods to be developed during this lesson.

L6_Graph_Dijkstra_sampleTest JUnit.



The Dijkstra Algorithm

Create and test the following methods.

```
public double[][] getD()
// creates a double [1][D.length] array and copies D
// over row 0.
// Surpasses bug in assertArrayEquals!

public int[] getPD () // Dijkstra’s pathways

private void initDijkstra (T departure)
// Verifies that the departure element exists
// Initializes the D and PD structures
// Initializes the S set (invokes setVisible(false) for each
// node in the graph.
// Sets the visible flag of the departure node to true.
```

The Dijkstra Algorithm

Implement the $O(n^3)$ version of the Dijkstra algorithm.

```
public void Dijkstra(T departure){
 initDijstra(departure);

    for (int p=1; p<getSize(); p++)   {
        Evaluate the cost of every edge {k, w} where k is
        member of the S set and w is member of V-S.

        Select the edge of minimum cost, adding w to the S
        set. w is the node with the lowest cost in D!

        For each node m in V-S update costs:
            if (D[w] + weight[w][m] < D[m]) {
              D[m] = D[w] + weight[w][m];
              P[m] = w;
            }
      }
}
```

(Home Work) The Dijkstra Algorithm

Implement the $O(n^2)$ version of the Dijkstra algorithm selecting pivot W on D.

HOMEWORK

Fully test every single class in the project (TestBench, Graph).

Its proper operation may be evaluated during the incoming tests and exams.

Lecture 6/B

Exercises

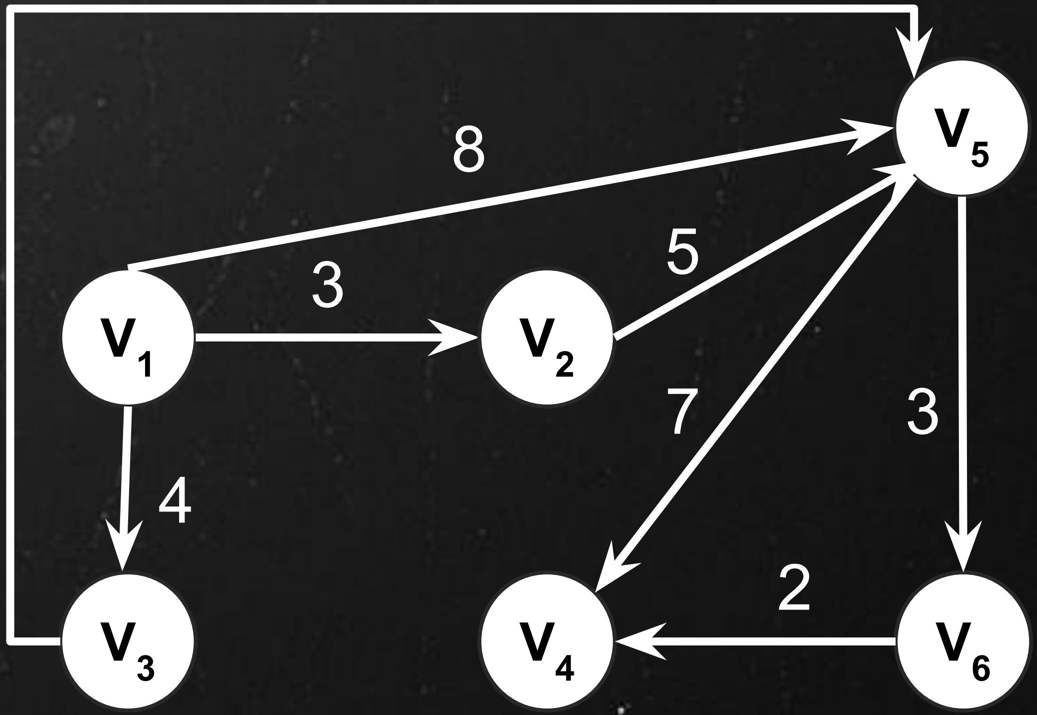
“I do not fear computers. I fear lack of them”

Isaac Asimov

Test

You may use the next JUnit as a guide for testing the methods developed during this lesson.

L6B_Exercises_sampleTest JUnit.


$$\begin{tikzcd}
 & & & V_5 \arrow[d, "3"] \arrow[dl, "7"'] \\
V_1 \arrow[r, "3"] \arrow[rrru, "8"] \arrow[d, "4"] & V_2 \arrow[rru, "5"] & & \\
V_3 \arrow[rrruu, bend left, "3"] & & V_4 & V_6 \arrow[l, "2"']
\end{tikzcd}$$

Exercises

Implement a method to explore the graph using the Breadth-first search approach.

```
public String BFPrint (T element)
// return null if the element does not exist

// Example
assertEquals("V1-V2-V3-V5-V4-V6-", g.BFPrint("V1"));
 assertEquals("V2-V5-V4-V6-"       , g.BFPrint("V2"));
 assertEquals("V3-V5-V4-V6-"       , g.BFPrint("V3"));
 assertEquals("V4-"                , g.BFPrint("V4"));
 assertEquals("V5-V4-V6-"          , g.BFPrint("V5"));
 assertEquals("V6-V4-"             , g.BFPrint("V6"));
```

Exercises

Hint: use a FIFO to store the nodes to be processed.

```
public String BFPrint(int v) {
  sets the 'visited' flag of v to true.
  put v into the FIFO
  {
    process first element in the FIFO (and remove it!)
    for each node reachable from that element (and
    not already visited)
     mark it as visited and place it at the end of FIFO
  }
 while (!FIFO.isEmpty())
```

Exercises

Implement a method to obtain the center of a graph.

```
public T getCenter ()

// Example
assertEquals("V4", g.getCenter());
```

Exercises

Remember: execute Floyd first

```
public T getCenter()
{
  floyd (getSize());

  get the eccentricity for each destination node)
       (maximum cost for each column in A)

      select the node with the minimum eccentricity.
```

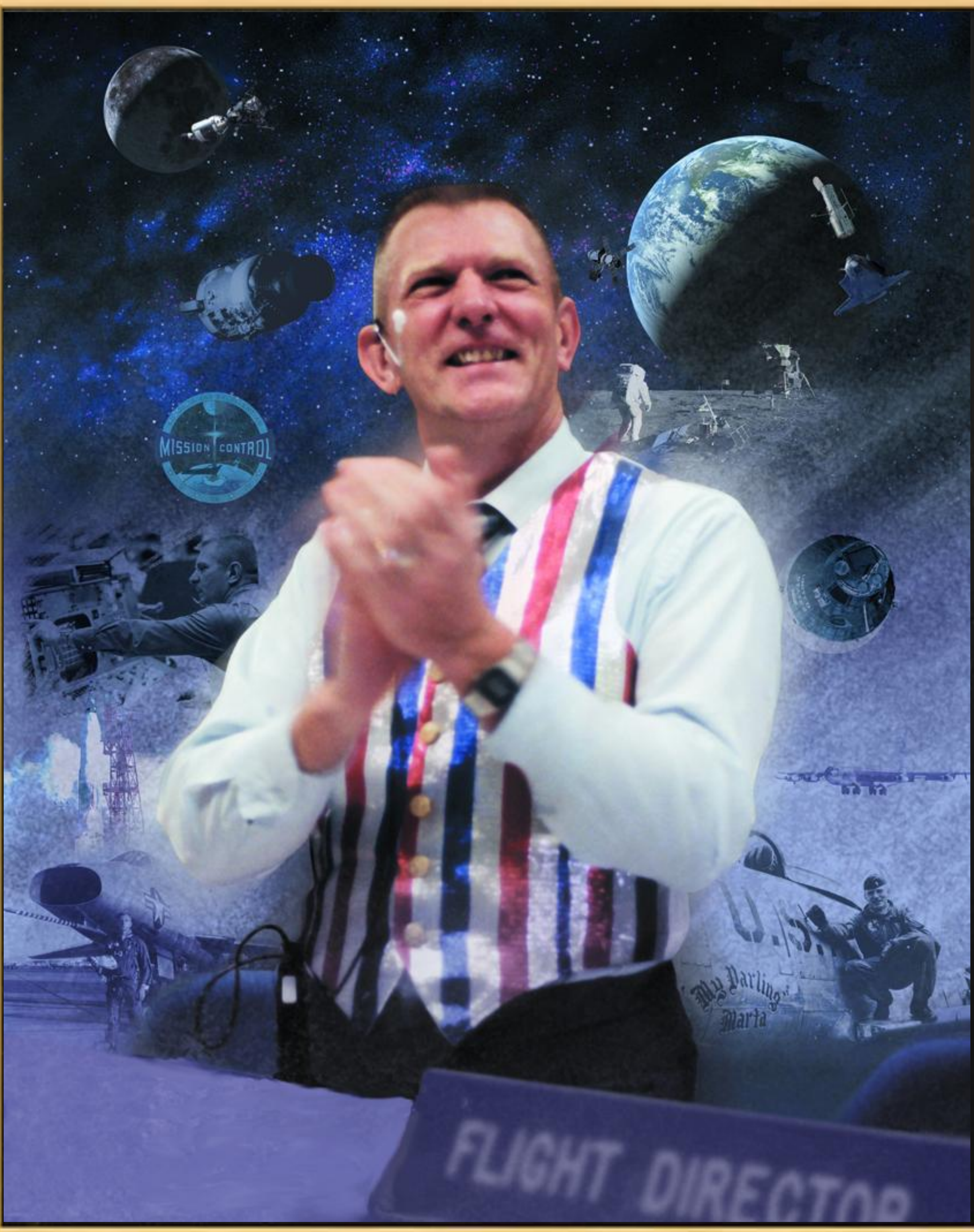
Exercises

Implement a method to obtain the shortest length path between two nodes.

```
public int shortestPathLength (T origin, T destination)

// Example
assertEquals (2, g.shortestPathLength("V1", "V6"));
```

Exercises

Hint: create a version of Floyd where the A matrix represents length instead of cost.



FAILURE IS NOT AN OPTION

Gene Kranz

REMEMBER!

- DEVELOP!
- DOCUMENT!
- TEST!

Martin :)

Lecture 7

Search Trees

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

Martin Fowler

AVLNode

Create a generic AVLNode class with the following fields:

```
private T element;
private AVLNode<T> left;
private AVLNode<T> right;
```

Implement getters and setters methods for each field.

AVLNode

Add the following constructors

```
public AVLNode (T element)

public AVLNode (T element, AVLNode<T> left, AVLNode<T> right)
```

AVLNode

Also add the following output methods

```
public String toString()
{
  return getElement().toString();
}

public void print()
{
    System.out.println (toString());
}
```

Binary Search Tree (BST)

Create a generic AVLTree class (initially behaving as a BST)

```
public class AVLTree <T extends Comparable<T>>
```

Implement the next methods:

```
public void add (T element);

public String toString();
// Preorder traversing.
// print a dash ('-') to represent null pointers.
```

Binary Search Tree (BST)

Test the AVLTree class using the toString() method

```
// Example.
AVLTree<Character> a = new AVLTree<Character>
a.add('b');
assertEquals ("b--", a.toString());
a.add('a');
assertEquals ("ba---", a.toString());
a.add('d');
assertEquals ("ba--d--", a.toString());
a.add('c');
assertEquals ("ba--dc---", a.toString());
a.add('g');
assertEquals ("ba--dc--g--", a.toString());
a.add('i');
assertEquals ("ba--dc--g-i--", a.toString());
a.add('h');
assertEquals ("ba--dc--g-ih---", a.toString());
```

Binary Search Tree (BST)

Implement the next method in the AVLTree class

```
public boolean search (T element);
```

a. Implement basic tests

```
// Example
AVLTree<Character> a = new AVLTree<Character>();
a.add('b');
a.add('a');
a.add('d');
a.add('c');
a.add('g');
a.add('i');
a.add('h');
assertEquals (true, a.search('i'));
assertEquals (false, a.search('f'));
```

Binary Search Tree (BST)

Implement the following method in the AVLTree class

```
protected T getMax(AVLNode<T> theRoot);
```

a. Implement basic tests

```
// Example.
  AVLTree<Character> a = new AVLTree<Character>();

  a.add('b');
  a.add('a');
  a.add('d');
  a.add('c');
  a.add('g');
  a.add('i');
  a.add('h');
  assertEquals ('i', (char) a.getMax(a.getRoot()));
```

HOMEWORK (1/2)

Create a generic GraphPerformanceTest class with the following methods.

```
public static Graph<Integer> initGraph (int n)
// returns a graph of Integer elements containing n nodes
// every node is connected with each other by an edge of
// weight calculated as a random value.

public static void runDijkstra(long n)
// calls to the initGraph(n) method and runs the Dijkstra
// algorithm on the resulting graph.

public static void runFloyd(long n)
 // calls to the initGraph(n) method and runs the Floyd
// algorithm on the resulting graph.
```

HOMEWORK (2/2)

Use the GraphPerformanceTest to measure the performance of the Dijkstra and Floyd Algorithms

```
TestBench.test("01_Graph_Floyd.txt", 3, 100, 300,
 "GraphPerformanceTest", "runFloyd");

TestBench.test("02_Graph_Dijkstra.txt", 3, 100, 300,
 "GraphPerformanceTest", "runDijkstra");

TestBench.test("03_Graph_Build.txt", 3, 100, 300,
 "GraphPerformanceTest", "initGraph");
```

a. Create performance charts for Dijkstra and Floyd (subtract the time required to create the graph).

Lecture 8

Binary Search Trees

“One of my most productive days was throwing away 1000 lines of code”

Ken Thompson

Binary Search Tree (BST)

Implement the next method in the AVLTree class

```
public void remove (T element);



// PSEUDO CODE

1. Find the element to be deleted.
    a. If any of its subtrees (left/right) is null, return the
       opposite subtree (to be assigned to its father’s link).
    b. else // The element has two children!
        i. Replace the element with the max value obtained from
           its left subtree (use the getMax method for this).
       ii. Delete the element from its left subtree (invoke to
           the recursive remove method again but this time
           provide its left subtree as the root’s parameter).
```

Binary Search Tree (BST)

Test the remove method

```
//Example
AVLTree<Character> a = new AVLTree<Character>();
a.add('b');
a.add('a');
a.add('d');
a.add('c');
a.add('g');
a.add('i');
a.add('h');
assertEquals ("ba--dc--g-ih---", a.toString());
a.remove('b');
assertEquals ("a-dc--g-ih---", a.toString());
a.remove('g');
assertEquals ("a-dc--ih---", a.toString());
```

BST conversion into AVL trees

The Balance Factor (BF) for a given node can be obtained from its relative height.

1. Add this field to the AVLNode class.

```
private int height; // node's height
```

2. Provide setters and getters for this field.

3. Modify the toString() method to display the height of the node.

```
// USE THIS FORMAT PLEASE...
getElement().toString() + "(" + getHeight() + ")";
```

BST conversion into AVL trees

Use backtracking recursion in the edit methods of the AVL class (add and remove) to update the node's height.

1. Create a method called updateHeight() in the AVLNode class to update the node's height.

   a. Update the node's height. Use the relative height from its children to obtain it.

2. Invoke the updateHeight() as the last statement of the add and remove methods in the AVLTree class.

```
theRoot.updateHeight();
return (theRoot);
```

BST conversion into AVL trees

Implement tests to verify the proper working of the updateHeight method.

```
// Example
AVLTree<Character> a = new AVLTree<Character>();
a.add(‘b’);
a.add(‘a’);
a.add(‘d’);
a.add(‘c’);
a.add(‘g’);
a.add(‘i’);
a.add(‘h’);
assertEquals ("b(4)a(0)--d(3)c(0)--g(2)-i(1)h(0)---",
 a.toString());
```

HOMEWORK

Implement a method to merge the contents of the BST/AVL tree with the elements of a second one.

```
public AVLTree<T> joins (AVLTree<T> tree)
```

a. Fully test this method.

```
    AVLTree<Character> a = new AVLTree<Character>();
    a.add('b');
    a.add('a');
    a.add('d');
    AVLTree<Character> b = new AVLTree<Character>();
    b.add('c');
    b.add('g');
    b.add('i');
    b.add('d');
    assertEquals ("b(3)a(0)--d(2)c(0)--g(1)-i(0)--",
     a.joins(b).toString());
```

HOMEWORK (2)

Implement a method to intersect the contents of the BST/AVL tree with the elements of a second one.

```
public AVLTree<T> intersection (AVLTree<T> tree)
```

a. Fully test this method.

```
AVLTree<Character> a = new AVLTree<Character>();
a.add(‘b’);
a.add(‘a’);
a.add(‘d’);
AVLTree<Character> b = new AVLTree<Character>();
b.add(‘c’);
b.add(‘g’);
b.add(‘i’);
b.add(‘d’);
assertEquals ("d(0)--", a.intersection(b).toString());
```

Lecture 9

AVL Trees

“Controlling complexity is the essence of computer programming”

Brian Kernigan

BST conversion into AVL trees

It is time to get the Balance Factor (BF) for each node. It may be obtained using node's height.

1. Add the next method to the AVLNode class.

```
public int getBF(); // AVL node's BF must be in {-1, 0, 1}
```

2. Use the height of the left and right subtrees of the node to obtain its balance factor.

3. Modify the toString() method to display the BF of the node (instead of the relative height)

```
// USE THIS FORMAT PLEASE...
getElement().toString() + "(" + getBF() + ")";
```

BST conversion into AVL trees

Test the getBF method of the AVLNode using the toString method in the AVLTree class.

```
// Example
AVLTree<Character> a = new AVLTree<Character>();
a.add('b');
a.add('a');
a.add('d');
a.add('c');
a.add('g');
a.add('i');
a.add('h');
assertEquals ("b(3)a(0)--d(2)c(0)--g(2)-i(-1)h(0)---",
 a.toString());
```

AVL Trees

Use backtracking recursion to update the BF of each node in the search path in order to balance the tree.

1. Create a method called updateBF() in the AVLTree class to update the node's height.

```
private AVLNode<T> updateBF (AVLNode<T> theRoot)
{
    theRoot.updateHeight();
    return (theRoot);
}
```

2. Replace the call to updateHeight() in the last statement of the add and remove methods to...

```
return(updateBF (theRoot));
```

AVL Trees

Test it again, it should work in the same way as it did in the last version.

```
// Example
AVLTree<Character> a = new AVLTree<Character>();
a.add('b');
a.add('a');
a.add('d');
a.add('c');
a.add('g');
a.add('i');
a.add('h');
assertEquals ("b(3)a(0)--d(2)c(0)--g(2)-i(-1)h(0)---",
 a.toString());
```

AVL Trees

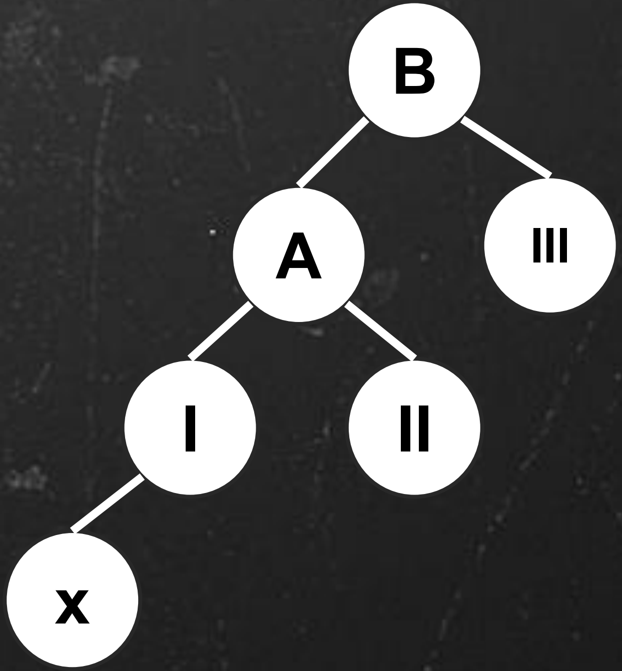
Use the *updateBF* to detect nodes unbalanced as well as to balance the tree using backtracking recursion.

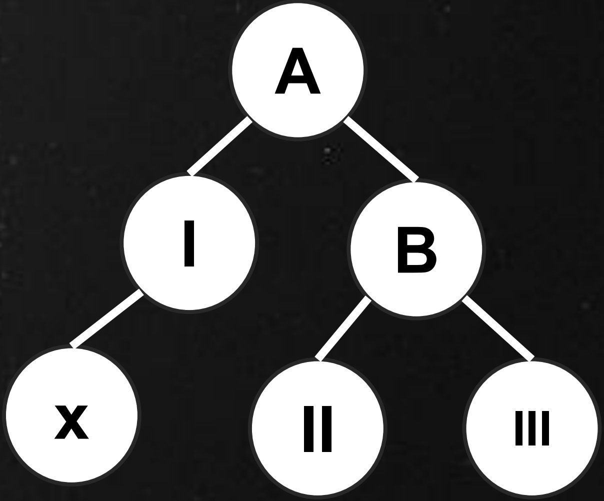
```
private AVLNode<T> updateBF (AVLNode<T> theRoot)
{
    if (theRoot.getBF() == -2) // Left Rotation!
    {
        //Does the AVL needs single or double rotation?
    }
    else
    if (theRoot.getBF() == 2)// Right Rotation!
    {
        //Does the AVL needs single or double rotation?
    }

    theRoot.updateHeight();
     return (theRoot);
}
```

AVL Trees

Detect and implement single (left & right) rotation.

```
private AVLNode<T> singleLeftRotation (AVLNode<T> b)
```





```
private AVLNode<T> singleRightRotation (AVLNode<T> b)
```

AVL Trees

Test the single rotation methods.

```
// Example
AVLTree<Character> a = new AVLTree<Character>();
a.add('a');
a.add('b');
a.add('c');
a.add('d');
a.add('e');
assertEquals ("b(1)a(0)--d(0)c(0)--e(0)--", a.toString());

a.add('f');
assertEquals ("d(0)b(0)a(0)--c(0)--e(1)-f(0)--",
 a.toString());
```

AVL Trees

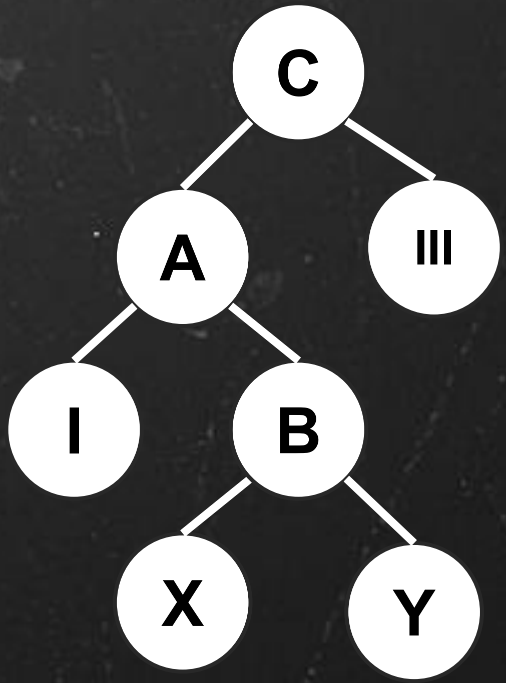
Implement double left & right rotations.

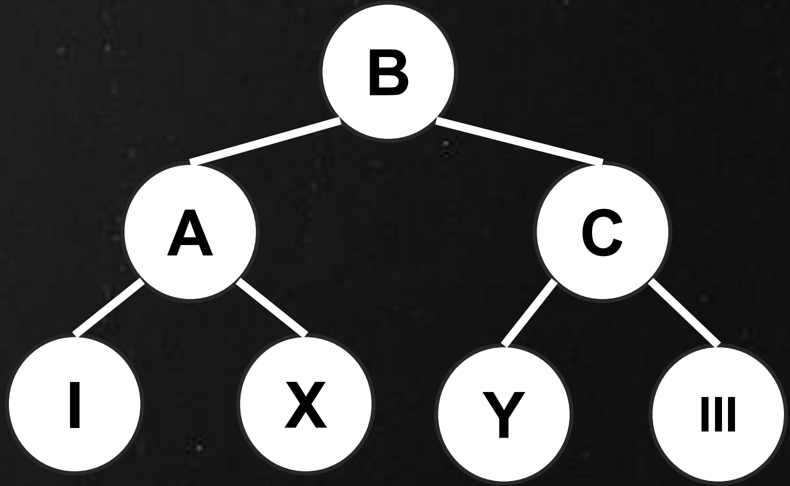
```
…
if (theRoot.getBF() == -2)
        {
            if (theRoot.getLeft().getBF() <= 0)
                theRoot = singleLeftRotation (theRoot);
            else
                theRoot = doubleLeftRotation (theRoot);
        }
else
        if (theRoot.getBF() == 2)
        {
                if (theRoot.getRight().getBF() >= 0)
                    theRoot = (singleRightRotation (theRoot));
                else
                    theRoot = (doubleRightRotation (theRoot));
        }
...
```

AVL Trees

Detect and implement double (left & Right) rotation.

```
private AVLNode<T> doubleLeftRotation (AVLNode<T> c)
```





```
private AVLNode<T> doubleRightRotation (AVLNode<T> c)
```

AVL Trees

Test the double rotation methods.

```
// Example
AVLTree<Character> a = new AVLTree<Character>();
a.add('e');
a.add('g');
a.add('b');
a.add('d');
a.add('c');

assertEquals ("e(-1)c(0)b(0)--d(0)--g(0)--", a.toString());
```

AVL Trees

Fully test the AVL tree.

```
AVLTree<Character> a = new AVLTree<Character>();
a.add('a');
a.add('b');
a.add('d');
assertEquals ("b(0)a(0)--d(0)--", a.toString());

AVLTree<Character> b = new AVLTree<Character>();
b.add('c');
b.add('g');
b.add('i');
b.add('d');
assertEquals ("g(-1)c(1)-d(0)--i(0)--", b.toString());

assertEquals ("d(0)b(0)a(0)--c(0)--g(1)-i(0)--",
 a.joins(b).toString());
```

HOMEWORK

Implement a getHeight method in the AVLTree class to explore the tree from its root, returning its height.

a. The method can not make use of the height property included in the AVL nodes.

```
// Example
AVLTree<Character> a = new AVLTree<Character>();
a.add('b');
a.add('a');
a.add('d');
a.add('c');
a.add('g');
a.add('i');
a.add('h');
assertEquals (3, a.getHeight());
```

Lecture 10

Priority Queues

“The function of good software is to make the complex appear to be simple”

Grady Booch

Binary Heaps

Create a generic BinaryHeap class with the following field.

```
    private ArrayList<T> heap;
```

a. Initialise the heap field in the constructor.

```
    heap = new ArrayList<T>();
```

b. Implement basic methods like:

```
    public boolean isEmpty() // return heap.isEmpty();
    public void print() // System.out.println (toString());
    public String toString () // return heap.toString();
```

Binary Heaps

Design a filterUp method.

```
// Executes ascending filtering on the node referred by
// a given position in the ArrayList.

private void filterUp(int pos)
```

Pseudocode.

```
Repeat until pos reaches the root (slot 0 in the ArrayList)
or until its value is greater than that of its father.

If the item is lower than its father (placed in the slot
E[(i-1)/2]) swap their positions.

// Collections.swap(heap, pos, father);
```

Binary Heaps

Use the filterUp method to implement the add method.

```
public void add(T element)
```

Pseudocode.

```
1.  Place the element at the end of the heap (ArrayList).
2.  Apply the filterUp method.
```

Binary Heaps

Test the add method.

```
BinaryHeap<Integer> a = new BinaryHeap<Integer>();
a.add(10);
a.add(9);
a.add(8);
assertTrue(a.toString().equals("[8, 10, 9]"));
a.add(7);
assertTrue(a.toString().equals("[7, 8, 9, 10]"));
a.add(6);
assertTrue(a.toString().equals("[6, 7, 9, 10, 8]"));
a.add(5);
assertTrue(a.toString().equals("[5, 7, 6, 10, 8, 9]"));
a.add(4);
assertTrue(a.toString().equals("[4, 7, 5, 10, 8, 9, 6]"));
```

Binary Heaps

Design a filterDown method.

```
// Applies descending filtering to the node referred by
// a given position in the ArrayList.

private void filterDown(int pos)
```

Pseudocode.

```
while (pos is not a leaf)
  1.- Select the children of 'pos' owning the smallest value.
  2.- if the value of pos > value of the child {
          Collections.swap(heap, pos, child);
          pos = child;
        }
       else
        stop!
```

Binary Heaps

Use the filterDown method to implement the getMin method.

```
public T getMin() // AKA remove
                  // returns the maximum priority element
```

Pseudocode.

```
1.  Return the first element in the ArrayList (the heap's root)
2.  Place the last element of the ArrayList in the heap's root.
3.  Filter it down!
```

Binary Heaps

Test the getMin() method.

```
BinaryHeap<Integer> heap = new BinaryHeap<Integer>();
heap.add(9);
heap.add(8);
heap.add(7);
heap.add(6);
heap.add(5);
heap.add(1);
heap.add(2);
heap.add(3);
heap.add(4);

assertEquals(heap.toString(), "[1, 3, 2, 4, 7, 8, 5, 9, 6]");
assertEquals (1, (int) heap.getMin());
assertEquals(heap.toString(), "[2, 3, 5, 4, 7, 8, 6, 9]");
```

Binary Heaps

Test the BinaryHeap class invoking the add and getMin methods.

```
// EXAMPLE
BinaryHeap<Character> b = new BinaryHeap<Character>();

 b.add('f');
 b.add('g');
 b.add('a');
 b.add('z');
 b.add('d');

 System.out.println (b.toString());
 assertEquals(b.toString(), "[a, d, f, z, g]");
 assertEquals('a', (char) b.getMin());
 assertEquals(b.toString(), "[d, g, f, z]");
```

HOMEWORK

Add a second constructor to build a binary heap from an array of elements.

```
public BinaryHeap(T[] elements)

// EXAMPLE
BinaryHeap<Integer> a = new BinaryHeap<Integer>({10, 9, 8, 7,
 6, 5, 4, 3, 2, 1});
 assertEquals(a.toString(), "[1, 2, 4, 3, 6, 5, 8, 10, 7,
 9]");
```

Tip:

Load the elements in the ArrayList and then call to the filterDown method for every item that is not a leaf.

Lecture 11

Hash Tables

“Be a yardstick of quality. Some people aren’t used to an environment where excellence is expected”

Steve Jobs

HashNode

Create a generic HashNode class and include the following constant fields.

```
public final static int EMPTY = 0;
public final static int VALID = 1;
public final static int DELETED = 2;
```

a. Define the next fields.

```
private T element;
private int status;
```

b. Implement getters and setters methods for each field.

c. Default constructor set the status to EMPTY.

Hash Table

Design a generic HashTable based on Open Addressing. Include the next constant fields.

```
protected final static int LINEAR_PROBING    = 0;
protected final static int QUADRATIC_PROBING = 1;
```

a. Define the next fields.

```
private int B = 7;
private int redispersionType = LINEAR_PROBING ;
private double minLF = 0.5;
```

b. Implement the following constructor.

```
public HashTable(int B, int redispersionType, double minLF)
```

Hash Table

Add the following hashing method.

```
/**
* Hashing function
*
* @param element to be stored.
* @param i Attempt number.
* @return slot in the array where the element should be
* placed
*/
protected int f (T element, int i)
{
    switch (redispersionType) {
        case LINEAR_PROBING:  return ...
}
```

Hash Table

Test the hashing function f with integer elements.

```
// Example
 HashTable<Integer> a = new HashTable<Integer>(5,
HashTable.LINEAR_PROBING, 0.5);
 assertEquals(2, a.f(7, 0));
 assertEquals(3, a.f(7, 1));
 assertEquals(4, a.f(7, 2));
 assertEquals(0, a.f(7, 3));

 // Example
 HashTable<Integer> b = new HashTable<Integer>(5,
HashTable.QUADRATIC_PROBING, 0.5);
 assertEquals(2, b.f(7, 0));
 assertEquals(3, b.f(7, 1));
 assertEquals(1, b.f(7, 2));
 assertEquals(1, b.f(7, 3));
```

Hash Table

Test the hashing function f with char elements.

```
// Example
 HashTable<Character> a = new HashTable<Character>(5,
HashTable.LINEAR_PROBING, 0.5);
 assertEquals(0, a.f('A', 0));
 assertEquals(1, a.f('A', 1));
 assertEquals(2, a.f('A', 2));
 assertEquals(3, a.f('A', 3));

 // Example
 HashTable<Character> b = new HashTable<Character>(5,
HashTable.QUADRATIC_PROBING, 0.5);
 assertEquals(0, b.f('A', 0));
 assertEquals(1, b.f('A', 1));
 assertEquals(4, b.f('A', 2));
 assertEquals(4, b.f('A', 3));
```

Hash Table

Define the associativeArray to store the hash nodes as an ArrayList.

```
private ArrayList<HashNode<T>> associativeArray;
```

Reserve memory for the ArrayList and initialise it inserting B empty HashNodes.

Hash Table

Implement the print and toString methods for the hash table… defining

```
public void print ();
public String toString();

//USE THIS FORMAT PLEASE (space as separator)
[0] (0) = null - [1] (0) = null - [2] (0) = null - [3] (0) =
null - [4] (0) = null -

// [Slot] (Node’s status) = element.toString() -
```

Hash Table

Implement the following methods.

```
public double getLF() // an ‘n’ double field is required!

public void add (T element);

public boolean search (T element);
```

Hash Table

Test both add and search methods

```
// Example
HashTable<Integer> a = new HashTable<Integer>(5,
HashTable.LINEAR_PROBING, 1.0);
 a.add(4);
 a.add(13);
 a.add(24);
 a.add(3);

 assertEquals("[0] (1) = 24 - [1] (1) = 3 - [2] (0) = null -
[3] (1) = 13 - [4] (1) = 4 - ", a.toString());
 assertEquals(true, a.search(3));
 assertEquals(false, a.search(12));
```

Hash Table

Test the add and search methods using quadratic probing too.

```
// Example
 HashTable<Integer> b = new HashTable<Integer>(5,
HashTable.QUADRATIC_PROBING, 1.0);
 b.add(4);
 b.add(13);
 b.add(24);
 b.add(3);

 assertEquals("[0] (1) = 24 - [1] (0) = null - [2] (1) = 3 -
[3] (1) = 13 - [4] (1) = 4 - ", b.toString());
 assertEquals(true, b.search(3));
 assertEquals(false, b.search(12));
```

Hash Table

Implement the remove method using the Lazy Deletion approach.

```
public void remove (T element);
```

Hash Table

Test the remove method.

```
        // Example
        HashTable<Integer> a = new HashTable<Integer>(5,
HashTable.LINEAR_PROBING, 1.0);
        a.add(4);
        a.add(13);
        a.add(24);
        a.add(3);
        a.remove(24);
        assertEquals(true, a.search(3));
        assertEquals("[0] (2) = 24 - [1] (1) = 3 - [2] (0) = null -
[3] (1) = 13 - [4] (1) = 4 - ", a.toString());

        a.add(15);
        assertEquals(true, a.search(3));
        assertEquals("[0] (1) = 15 - [1] (1) = 3 - [2] (0) = null -
[3] (1) = 13 - [4] (1) = 4 - ", a.toString());
```

Hash Table

Test for quadratic probing too.

```
        // Example
        HashTable<Integer> b = new HashTable<Integer>(5,
    HashTable.QUADRATIC_PROBING, 1.0);
        b.add(4);
        b.add(13);
        b.add(24);
        b.add(3);

        b.remove(24);
        assertEquals(true, b.search(3));
        assertEquals("[0] (2) = 24 - [1] (0) = null - [2] (1) = 3 -
[3] (1) = 13 - [4] (1) = 4 - ", b.toString());


        b.add(15);
        assertEquals(true, b.search(3));
        assertEquals("[0] (1) = 15 - [1] (0) = null - [2] (1) = 3 -
[3] (1) = 13 - [4] (1) = 4 - ", b.toString());
```

HOMEWORK

Develop the next methods in the HashTable class based on the management of prime numbers.

```
// Verifies whether a given number is a prime number
private boolean isPrime(int number)


// Returns the prime number predecessor of a given int number
private int getPrevPrimeNumber(int number)


// Returns the prime number successor of a given int number
private int getNextPrimeNumber(int number)
```

Lecture 11

Dynamic Resizing

“Everything is theoretically impossible, until it is done”

Robert A. Heinlein

Hash Table

Add the following field to the Hash table

```
private int R = 5;
```

a. Update it in the constructor

```
this.R = getPrevPrimeNumber(B);
```

b. Implement double hashing in the f function

```
protected int f (T element, int i)
{
    switch (redispersionType) {
            ...
        case DOUBLE_HASHING:  return ...

}
```

Hash Table

Test the double hashing function f with integer elements.

```
// Example
HashTable<Integer> c = new HashTable<Integer>(5,
 HashTable.DOUBLE_HASHING, 0.5);
assertEquals(2, c.f(7, 0));
assertEquals(4, c.f(7, 1));
assertEquals(1, c.f(7, 2));
assertEquals(3, c.f(7, 3));
assertEquals(0, c.f(7, 4));

assertEquals(0, c.f(0, 0));
assertEquals(1, c.f(2, 4));
assertEquals(2, c.f(3, 3));
assertEquals(3, c.f(32, 1));
assertEquals(4, c.f(1045, 2));
```

Hash Table

Test the double hashing function f with char elements.

```
/// Example
HashTable<Character> c = new HashTable<Character>(5,
 HashTable.DOUBLE_HASHING, 0.5);
assertEquals(0, c.f('A', 0));
assertEquals(1, c.f('A', 1));
assertEquals(2, c.f('A', 2));
assertEquals(3, c.f('A', 3));
assertEquals(4, c.f('A', 4));

assertEquals(2, c.f('a', 0));
assertEquals(4, c.f('a', 1));
assertEquals(1, c.f('a', 2));
assertEquals(3, c.f('a', 3));
assertEquals(0, c.f('a', 4));
```

Hash Table

Test the add, search and remove methods with double hashing.

```
// Example
HashTable<Integer> c = new HashTable<Integer>(5,
 HashTable.DOUBLE_HASHING, 1.0);
c.add(4);
c.add(13);
c.add(24);
c.add(3);
assertEquals("[0] (0) = null - [1] (1) = 3 - [2] (1) = 24 -
 [3] (1) = 13 - [4] (1) = 4 - ", c.toString());
c.remove(24);
assertEquals("[0] (0) = null - [1] (1) = 3 - [2] (2) = 24 -
 [3] (1) = 13 - [4] (1) = 4 - ", c.toString());
assertEquals(true, c.search(3));
c.add(15);
assertEquals(true, c.search(3));
assertEquals("[0] (1) = 15 - [1] (1) = 3 - [2] (2) = 24 - [3]
 (1) = 13 - [4] (1) = 4 - ", c.toString());
```

Hash Table

Add the following method to the HashTable class.

```
public ArrayList<HashNode<T>> getAssociativeArray()
```

a. It will be used to obtain the new array after the completion of the dynamic resizing.

Hash Table

Add a couple of private/protected methods to perform dynamic resizing.

```
// Resizes the table dynamically to the a given size.
private void dynamicResize (int newSize)

// Resizes the table dynamically to next prime to
// the B*2.
private void dynamicResize ()
```

Invoke it just after the execution of the add method if...

```
if (getLF() > minLF)
  dynamicResize();
```

Hash Table

Test the dynamic resizing feature in different scenarios

```
// Example
HashTable<Integer> a = new HashTable<Integer>(5,
 HashTable.LINEAR_PROBING, 0.5);
a.add(4);
assertEquals (0.2, a.getLF(), 0.1);
a.add(13);
assertEquals (0.4, a.getLF(), 0.1);
assertEquals ("[0] (0) = null - [1] (0) = null - [2] (0) =
 null - [3] (1) = 13 - [4] (1) = 4 - ", a.toString());

a.add(24); // DYNAMIC RESIZING!
assertEquals (0.27, a.getLF(), 0.1);
assertEquals("[0] (0) = null - [1] (0) = null - [2] (1) = 24
 - [3] (1) = 13 - [4] (1) = 4 - [5] (0) = null - [6] (0) =
 null - [7] (0) = null - [8] (0) = null - [9] (0) = null -
 [10] (0) = null - ", a.toString());
```

HOMEWORK

Fully test the Hash table for the three different probing methods.

1. Test it with different Load Factors.
2. Verify that the dynamic resizing procedures work as expected.