Module 13 Data Structures & Algorithms

Module title	Data Structures and Algorithms
Module NFQ level (only if an NFQ level can be demonstrated)	6
Module number/reference	BSCH-DSA
Parent programme(s)	Bachelor of Science (Honours) in Computing Science
Stage of parent programme	Stage 2
Semester (semester1/semester2 if applicable)	Semester 2
Module credit units (FET/HET/ECTS)	ECTS
Module credit number of units	10
List the teaching and learning modes	Direct, Blended
Entry requirements (statement of knowledge, skill and competence)	Learners must have achieved programme entry requirements.
Pre-requisite module titles	BSCH-OOP
Co-requisite module titles	None
Is this a capstone module? (Yes or No)	No
Specification of the qualifications (academic, pedagogical and professional/occupational) and experience required of staff (staff includes workplace personnel who are responsible for learners such as apprentices, trainees and learners in clinical placements)	Qualified to as least a Bachelor of Science (Honours) level in Computer Science or equivalent and with a Certificate in Training and Education (30 ECTS at level 9 on the NFQ) or equivalent.
Maximum number of learners per centre (or instance of the module)	60
Duration of the module	One Academic Semester, 12 weeks teaching
Average (over the duration of the module) of the contact hours per week	5
Module-specific physical resources and support required per centre (or instance of the module)	One class room with capacity for 60 learners along with one computer lab with capacity for 25 learners for each group of 25 learners

Analysis of required learning effort		
	Minimum ratio teacher / learner	**Hours**
Effort while in contact with staff		
Classroom and demonstrations	1:60	24
Monitoring and small-group teaching	1:25	36
Other (specify)		
Independent Learning		
Directed e-learning		
Independent Learning		100
Other hours (worksheets and assignments)		90
Work-based learning – learning effort		
Total Effort		250

Allocation of marks (within the module)					
	Continuous assessment	**Supervised project**	**Proctored practical examination**	**Proctored written examination**	**Total**
Percentage contribution	60%			40%	100%

Module aims and objectives

This module builds on the work completed in the Object-Oriented Programming module and will apply the methods learned there to the design of classes that implement data structures. As in all programming modules, a key objective is the acquisition, on behalf of the learner, of good software engineering skills and the application of these skills to the design and implementation of software components. At the heart of all software design is the implementation of appropriate data structures that provide efficient data models for the problem at hand. Learners develop an in-depth knowledge of the standard generic data structures: stacks, queues, sets, bags and maps; and also learn to implement these using both linear (linked lists, arrays) and non-linear (binary search trees, avl trees, B-trees) data structures. Learners will also study Graph Theory and the fundamental graph searching algorithms. Unit testing will be used throughout to build test models for classes developed to implement data structures.

Minimum intended module learning outcomes

On successful completion of this module, the learner will be able to:

1. Implement and use data structures algorithms introduced on the course
2. Apply object-oriented methods and good practices when designing data structures
3. Implement both recursive and non-recursive solutions to classical data structure problems

4. Justify the requirement to design and implement fast sorting algorithms – Quicksort and Mergesort
5. Review simple algorithms using asymptotic analysis
6. Implement both linear and non-linear data structures
7. Explain how hash-tables and binary search trees minimize both the cost of insertion and retrieval
8. Explain how to optimise the performance of binary tree using well known methods such as avl and B-trees Use a map implementation (HashMap or TreeMap) to provide a data structure that models a given programming task

Rationale for inclusion of the module in the programme and its contribution to the overall MIPLOs

This module builds and extends the work completed by the learners in the stage 1 module on Computer Programming and the stage 2 module, Object-oriented Programming. It takes an object-oriented approach to the study of data structures and, hence, provides learners the experience of applying this methodology to a complex field of programming. The module itself is provides learners with the tools necessary to build models in many areas of computing. Data structures are fundamental to all advanced aspects of computer programming and are to be found in network algorithms, machine learning algorithms, artificial intelligence algorithms, encryption algorithms, graphics algorithms, compilers, and many other fields of algorithmic application. Appendix 1 of the programme document maps MIPLOs to the modules through which they are delivered.

Information provided to learners about the module

Learners receive a programme handbook to include module descriptor, module learning outcomes (MIMLO), class plan, assignment briefs, assessment strategy and reading materials.

Module content, organisation and structure

Recursion

- Concept of recursion, recursive functions
- Recursion over sequences
- Tail recursion
- Divide and conquer algorithms

Analysis of algorithms

- Counting and calculating the time complexity of an algorithm
- Basic ideas and definitions of asymptotic analysis

- Big O notation and its application to the evaluation of temporal cost of algorithms
- Comparing performance using big O notation
- Basic time and space analysis

Sorting

- Simple sorting algorithms: insertion, selection, bubble
- QuickSort. Merge Sort. Heapsort
- Analysis of sorting algorithms using big O notation
- Sorting in linear time

Typed Dynamic Data Structures

- Dynamic arrays; singly linked lists and doubly linked lists
- Writing encapsulated classes that provide implementations of these data structures
- Methods for: add, remove, search, size, toString, sort, replace

Generic Dynamic Data Structures

- Writing generic classes
- Using linear data structures to implement generic classes that encapsulate: stacks, queues, priority queues and sets
- Problem solving with these data structures

Functional Interfaces

- Lambda expressions, functions, predicates and higher-order functions
- Writing functional interface methods for data structure classes

Generic HashTables

- Concept and definition of Hashtables
- Implementing classes that encapsulate a hash table
- Use of hash tables in implementing sets

Generic Binary Search Trees

- Concept and definition of Trees: representing rooted trees, binary search trees, query, insertion, deletion, traversal
- Optimising the performance of binary trees with avl trees and black-red trees and B-trees
- Implementing data structures with trees

Maps

- Definition of Map
- Implementations of Map interface – HashMap and TreeMap
- Problem solving with maps. Multi-sets (Bag) problem and implementation of Bag using map implementations

Graphs and graph algorithms

- Graphs: basic concepts and representation
- Breadth first search, depth first search
- Dijkstra shortest-path algorithm

Module teaching and learning (including formative assessment) strategy

The module is delivered through a combination of lectures and practical lab programming sessions. The learners complete a series of worksheets throughout the module that are directly related to the material covered in lectures. The emphasis is on developing sound software engineering skills in practical programming based on theoretical knowledge.

Assessment consists of a series of continuous assignments and a final examination. Each week learners are required to complete a series of programming tasks that relate to the material covered in lectures. The practical lab sessions are used to enforce concepts covered in the lectures and the worksheets are used to ensure that learners are keeping up with the material as it is delivered. Lab sessions are also used to deal with issues emerging from the worksheets. All work submitted by learners is assessed and comments are given to individual learners. Typically, there are 10 worksheets and the final mark is based on the seven best pieces of work submitted.

Timetabling, learner effort and credit

The module is timetabled as one 2-hour lecture and two 1.5-hour labs per week.
Continuous assessment spreads the learner effort to focus on the aspects of the course under discussion.

There are 60 contact hours made up of 12 lectures delivered over 12 weeks with classes taking place in a classroom. There are also 24 lab sessions delivered over 12 weeks taking place in a fully equipped computer lab. The learner will need 100 hours of independent effort to further develop the skills and knowledge gained through the contact hours. An additional 90 hours are set aside for learners to work on worksheets and assignments that must be completed for the module.

The team believes that 250 hours of learner effort are required by learners to achieve the MIMLOs and justify the award of 10 ECTS credits at this stage of the programme.

Work-based learning and practice-placement
There is no work based learning or practice placement involved in the module.

E-learning
The college VLE is used to disseminate notes, advice, and online resources to support the learners. The learners are also given access to Lynda.com as a resource for reference.

Module physical resource requirements
Requirements are for a classroom for 60 learners equipped with a projector, and a 25-seater computer lab for practical sessions with access to Java and a suitable development environment (for example Notepad++) (this may change should better techonologies arise).

Reading lists and other information resources
Recommended Text
Lectures on Data Structures and Algorithms in Java, Tony Mullins (2017), Griffith College

Bloch, J. (2018) *Effective Java*. Boston: Addison Wesley.

Secondary Reading
Goodrich, M. T., Goldwasser, M. H. and Tamassia, R. (2015) *Data Structures and Algorithms in Java:* Singapore: Wiley.

Dasgupta, S., Papadimitriou, C. H. and Vazirani, U. V. (2008) *Algorithms*. Boston; Montréal: McGraw-Hill Higher Education.

Weiss, M. A. (2012) *Data Structures and Algorithm Analysis in Java*. Harlow: Pearson Education.

Wirth, N. (2008) *Algorithms + Data Structures = Programs*. New York: Prentice-Hall.

Naftalin, M. and Wadler, P. (2007) *Java Generics and Collections: [speed up the Java development process*. New York: O'Reilly.

Specifications for module staffing requirements
For each instance of the module, one lecturer qualified to at least Bachelor of Science (Honours) in Computer Science or equivalent, andth a relevant third level teaching qualification (e.g. Certificate in Training and Education). Industry experience would be a benefit but is not a requirement.

Learners also benefit from the support of the programme director, programme administrator, learner representative and the Student Union and Counselling Service.

Module Assessment Strategy

The assignments constitute the overall grade achieved, and are based on each individual learner's work. The continuous assessments provide for ongoing feedback to the learner and relates to the module curriculum.

No.	Description	MIMLOs	Weighting
1	Series of weighted worksheets Worksheet 1: Learning outcomes 3 Worksheet 2: Learning outcomes 3,5 Worksheet 3: Learning outcomes 3,4,5 Worksheet 4: Learning outcomes 1,2,3,6 Worksheet 5: Learning outcomes 1,2,3,6 Worksheet 6: Learning outcomes 1,2,6 Worksheet 7: Learning outcomes 1,2,6,7,8 Worksheet 8: Learning outcomes 1,2,6,7,8,9	1-9	60%
9	Written exam that tests the theoretical aspects of the module	1-9	40%

All repeat work is capped at 40%.

Sample assessment materials

Note: All assignment briefs are subject to change in order to maintain current content.

Assignment 1

Question 1

Write a recursive function, called print, that takes two integer arguments a, b and prints the values in ascending order from a to b, inclusive. For example, print(3,6) outputs: 3 4 5 6.

Question 2

Given below is a recursive function sum(a,b) that calculates the summation of the numbers: a + (a+1) + (a+2) + .. + b. Write a tail recursive function that provides the same functionality.

```
static int sum(int a, int b){
        if(a == b) return a;
        else return (a + sum(a+1,b));
}
```

Question 3

Write a recursive function that prints a given positive integer value as its binary equivalent. For example, if the value is 11 the output should be 1011.

Assignment 2

Question 1 (5 marks)
Using the statement execution times defined for HAL, calculate the running time for the given function.

```
static int freq(int f[]){
      int k = 1; int j = 0;
      while(j < f.length){
            if(f[j] * 2 == j)
                  k = k * f[j];
            j++;
      }
      return k;
}
```

===
====

Question 2 (5 marks)
By calculating *big-O* for each of the given functions show that both functions fall into different categories.

```
static int intDiv(int a, int b){ //assume b > 0, a >=0
     int q = 0; int r = a;
     while(r >= b){
          q = q + 1;
          r = r - b;
     }
     return q;
}

static int intDiv1(int a, int b){
     int q = 0; int r = a;
     int c = 1;
     while(c*b <= a) c = c * 2;
     while(c > 1){
          c = c/2;
          if(a >= (q + c) * b){
               q = q + c; r = r - c * b;
          }
     }
     return q;
}
```

===

Question 3 (10 marks)
On Moodle you will find a file called `ComparisonSortFunctions.java`. This file contains two functions called `insertionSort` and `mergeSort`. It also contains a `main` method with code that initializes two integer arrays, f and g, with identical values. It invokes both functions on these data arrays to sort them. There is also a constant N that has no actual value. Your task is to complete the program using the benchmarking strategy described on page 50 of the textbook. Your program should print the time taken to perform each sort. To do this you must give N a value.
Your task is to use your program to benchmark each of the two sorting algorithms. By choosing different values of N you should record the performance of each test run. What

we want you to do is to write a report on what you find. You should also try to answer the following question: for what values of N does insertionSort outperforms mergeSort? An outline for your report is given below.

Report
Type of CPU:
Memory:
Test 1
Size of data: N =
Cost of InsertionSort
Cost of MergeSort
Analysis: (What do you conclude from the test)
…
Test 2
Size of data: N =
Cost of InsertionSort
Cost of MergeSort
Analysis:
...

Test 3
Size of data: N =
Cost of InsertionSort
Cost of MergeSort
Analysis:

For what values of N does insertionSort outperforms mergeSort?

Conclusion:

Assignment 3

Question 1
Implement a class called MyStack<E> that implements the Stack<E> interface using an ArrayList<E>. The stack, in this instance, is bounded. Write a test program for your stack class.
The interface for a generic stack is:

```
interface Stack<E>{
        public boolean push(E x);
        public boolean pop();
        public E top();
        public boolean empty();
        public boolean full();
public Iterator<E> iterator();
}
```

Question 2
Implement the Queue<E> interface with a class called MyQueue<E>. Your class should use the class ArrayDeque<E>() from the Java Collection classes to implement the interface. The relevant methods for this class are listed on page 151 of your textbook. Write a test program for your queue class.
The interface for a generic queue is:

```
interface Queue<T>{
        public boolean join(T x);
        public T top();
        public boolean leave();
        public boolean full();
        public boolean empty();
        public String toString();
        public boolean contains(T x);
}
```

Assignment 4

Question 1
A class Point that represents a point in the Cartesian plane is given. Re-write this class so that it meets the requirements for storage in our HashList<E> container. Test your class by creating a hash list of Point instances and running relevant queries on your list. The list should contain at least 10000 point instances. Part of your test should experiment with the number of lists in the table (the value of n passed to the constructor) to try to optimize the performance so that you don't get a large number of empty buffers and no buffer contains a large number of elements.

Question 2
Write the following methods for the MyHashList<E> class given in the assignment code. You should test these new methods by creating a hash list of integer values.

public LinkedList<E> getList(E x)	Returns a copy of the list of elements matching values whose hash code match that of x
public void remove(List<E> ls)	Remove elements in ls from table

List<E> get(Predicate<E> pr)	Returns the list of values that satisfy the predicate pr.

Assignment 5

Question 1

In the file Assignment5.java add code that creates a BinarySearchTree of Word values, where a Word is defined as a non-blank string. The tree should be constructed with a list of 20 words of your choice. Your code should provide a test platform for the methods in the class BinarySearchTree. In particular you must test add, remove, preOrder, inOrder, postOrder, height and contains.

The class Word is defined in the java file.

(Note: you may not amend the BinarySearchTree class when doing this.)

Question 2

Write the following methods for the BinarySearchTree<E> class discussed in the lecture notes.

public E maxElement()	Returns the largest element in the tree.
public ArrayList<E> leafNodes()	Returns an ArrayList containing the leaf nodes in the tree. A leaf node is one whose left and right children are null
Public List<E> get(Predicate<E> pr)	Retrieve a list that satisfies pr.

Assignment 6

```
/**
 * Student name:
 *
 * Student number:

 For Assignment6 please complete the tasks listed for Question 1 and Question
2
*/

import java.util.*;
import java.util.function.*;
```

```
public class Assignment6{
  public static void main(String args[]){
    /*Question 1 ==========================================================
    Using the class NewNumbers listed below write a code sequence that tests
    the methods:
      forAll(Predicate<Integer> pr),
      List<Integer> getSubList(Predicate<Integer> pr)
      List<Integer> mapList(Function<Integer,Integer> f)
      Sample tests might be: all the values are positive, all negative,
      retrieve a list of the even numbers, a list of negative numbers,
      use mapList to return the square of all the numbers, etc
      You should write at least three tests for each method

==========================================================================*/
    // This code sets up a List
    NewNumbers lst = new NewNumbers();
    lst.add(Arrays.asList(1,2,3,4,6,-1,-5,7,8,12,4,-5,0,0,1,4,-2));
    lst.print(x->System.out.print(x+" "));




    /*Question 2
==============================================================
    Using the generic class MyList<E> listed below write a code sequence that
    tests all of its the methods. You are given a Book class that you can use
    to perform your tests. Note that creating a MyList<Integer> instance is not
    an acceptable data type for your tests.
    ============================================================= */

    MyList<Book> bls = new MyList<>();
  }
}

class NewNumbers{
  private List<Integer> data = new ArrayList<>();
  void add(int x){data.add(x);}
  void add(List<Integer> lst){data.addAll(lst);}
  boolean contains(Predicate<Integer> pr){
         for(Integer x : data)
          if(pr.test(x)) return true;
         return false;
  }
  boolean forAll(Predicate<Integer> pr){
         for(Integer x : data)
           if(!pr.test(x)) return false;
         return true;
  }
  List<Integer> getSubList(Predicate<Integer> pr){
         List<Integer> tmp = new ArrayList<>();
         for(Integer x : data)
          if(pr.test(x)) tmp.add(x);
         return tmp;
  }
  List<Integer> mapList(Function<Integer,Integer> f){
   List<Integer> tmp = new ArrayList<>();
   for(Integer x : data) tmp.add(f.apply(x));
   return tmp;
  }
  int count(Predicate<Integer> pr){
         int count = 0;
         for(Integer x : data) if(pr.test(x)) count++;
    return count;
  }
  int sum(Predicate<Integer> pr){
         int s = 0;
         for(Integer x : data) if(pr.test(x)) s += x;
         return s;
  }
  void print(Consumer<Integer> cn){
         data.forEach(cn);
         System.out.println();
  }
}

class MyList<E> implements Iterable<E>{
  private List<E> data = new LinkedList<>();
  public void add(E x){data.add(x);}
  public void add(List<E> ls){
    for(E x : ls) data.add(x);
  }
  public boolean contains(Predicate<E> pr){
    for(E x : data) if(pr.test(x)) return true;
    return false;
  }
  public List<E> filterList(Predicate<E> pr){
  List<E> tmp = new LinkedList<>();
  for(E x : data) if(pr.test(x)) tmp.add(x);
  return tmp;
  }
  public void remove(Predicate<E> pr){
  data.removeIf(pr);
  }
  public void print(Consumer<E> cn){
    data.forEach(cn);
    System.out.println();
  }
  public Iterator<E> iterator(){return data.iterator();}
}
class Book{
  private String title;
  private String author;
  public Book(String t, String a){title = t; author = a;}
  public String title(){return title;}
  public String author(){return author;}
  public boolean equals(Object ob){
  if(!(ob instanceof Book)) return false;
  Book b = (Book)ob;
  return title.equals(b.title) && author.equals(b.author);
  }
  public int hashCode(){return 31*title.hashCode()*author.hashCode();}
  public String toString(){return title+" "+author;}
}
```

Assignment 7

Please complete all parts of the question described below. This assignment forms part of the assessment for this module and you must upload your solution in the given file on or before the date given on Moodle.

Question 1

In the file Assignment7.java add code that creates a BinarySearchTree of Word values, where a Word is defined as a non-blank string. The tree should be constructed with a list of 20 words of your choice. Your code should provide a test platform for the methods in the class BinarySearchTree. In particular you must test add, remove, preOrder, inOrder, postOrder, height and contains.

The class Word is defined in the java file.

(Note: you may not amend the BinarySearchTree class when doing this.)

Question 2

Write the following methods for the BinarySearchTree<E> class discussed in the lecture notes.

public E maxElement()	Returns the largest element in the tree.
public ArrayList<E> leafNodes()	Returns an ArrayList containing the leaf nodes in the tree. A leaf node is one whose left and right children are null
Public List<E> get(Predicate<E> pr)	Retrieve a list that satisfies pr.

Assignment 8

```
/*
*       Student name:
*       Student number:
*/
/*
*  Assignment8
*
*                 County-Towns Problem
*
*  A data structure that encapsulates a list of counties and the names of towns
is
*  required
*
*  Two classes called County and Town are given. In each case they encapsulate a
String
*  that represents the name of the county or the name of the town. Both of
these
*  classes are immutable.
*  An outline of the class CountyTowns is also given.
*  The data structure TreeMap<County,TreeSet<Town>> is used to model the
*   relationship between counties and towns. We assume that a county does not
have
*   duplicate named towns but counties may have town names in common.
*  Your task is to complete the methods listed as part of the interface to this
class.
*  In each case the signature and semantics of the method are given.
*  You must also complete a simple test of these methods.
*/

import java.util.*;
public class Assignment8{
  public static void main(String args[]){
   //Data Setup section
===========================================================

        ArrayList<TreeSet<Town>> towns = new ArrayList<TreeSet<Town>>();
        TreeSet<Town> cork = new TreeSet<>(Arrays.asList(
          new Town("Bandon"),new Town("Blarney"),new Town("Fermoy"),new
Town("Kanturk")
        ));
        towns.add(cork);
        TreeSet<Town> limerick = new TreeSet<>(Arrays.asList(
          new Town("Croom"),new Town("Foynes"),new Town("Ballingarry")
        ));
        towns.add(limerick);
        TreeSet<Town> offaly = new TreeSet<>(Arrays.asList(
          new Town("Rhode"),new Town("Tullamore"),new Town("Barna")
        ));
        towns.add(offaly);
        TreeSet<Town> galway = new TreeSet<>(Arrays.asList(
          new Town("Athenry"),new Town("Barna"),new Town("Tuam")
        ));
        towns.add(galway);
        TreeSet<Town> dublin = new TreeSet<>(Arrays.asList(
          new Town("Howth"),new Town("Rush"),new Town("Skerries"),new
Town("Oldtown")
        ));
        towns.add(dublin);
        TreeSet<Town> mayo = new TreeSet<>(Arrays.asList(
          new Town("Ballina")
   ));
        towns.add(mayo);
        TreeSet<Town> tipperary = new TreeSet<>(Arrays.asList(
          new Town("Ballina"),new Town("Clonmel"),new Town("Ballingarry"),new
Town("Fethard")
        ));
        towns.add(tipperary);
        TreeSet<Town> kerry = new TreeSet<>(Arrays.asList(
          new Town("Barna"),new Town("Tralee"),new Town("Listowel"),new
Town("Oldtown")
        ));
        towns.add(kerry);
        List<County> cnts = new ArrayList<>(Arrays.asList(
         new County("Cork"),new County("Limerick"),new County("Offaly"),
         new County("Galway"),new County("Dublin"),new County("Mayo"),
         new County("Tipperary"),new County("Kerry")
```

```
          ));
  // End data setup ====================================================

  //Create Data Structure using data ========================================

    CountyTowns data = new CountyTowns();
    for(int j = 0; j < cnts.size();j++)
            data.add(cnts.get(j),towns.get(j));
          System.out.println(data);


  //=====================================================================
===
  //Test methods section based on given data set
=================================



  //=====================================================================
===
  }
}

//Code for classes County and Town
=====================================================
// Do not edit this section
============================================================

final class County implements Comparable<County>{
  private final String county;
  County(String name){
          assert name != null && name.length() > 0;
          county = name;
  }
  String county(){return county;}
  public String toString(){return county;}
  public boolean equals(Object ob){
          if(!(ob instanceof County)) return false;
          County cty = (County)ob;
          return county.equals(cty.county);
  }
  public int hashCode(){return county.hashCode();}
  public int compareTo(County cty){
          if(cty == null) return -1;
          return county.compareTo(cty.county);
```

```
 }
}
class Town implements Comparable<Town>{
 private  String town;
 Town(String name){
        assert name != null && name.length() > 0;
        town = name;
 }
 String town(){return town;}
 public String toString(){return town;}
 public boolean equals(Object ob){
        if(!(ob instanceof Town)) return false;
        Town tn = (Town)ob;
        return town.equals(tn.town);
 }
 public int hashCode(){return town.hashCode();}
 public int compareTo(Town tn){
        if(tn == null) return -1;
        return town.compareTo(tn.town);
 }
}
//=========================================================================
=
// Data Structure CountyTowns
============================================

class CountyTowns{
 private TreeMap<County,TreeSet<Town>> data;
 CountyTowns(){
        data = new TreeMap<>();
 }
 void add(County cty, Town town){
   //Add county and 1 town
        if(data.containsKey(cty))
          data.get(cty).add(town);
        else{
          TreeSet<Town> tmp = new TreeSet<>();
                tmp.add(town);
          data.put(cty,tmp);
        }
 }
 void add(County cty, Set<Town> towns){
        //Add county together with a list of towns
 }
 Set<Town> listTowns(County cty){
        //List towns in a given county
 }
 Set<County> counties(){
   //list all counties
 }
 Set<Town> listAllTowns(){
   //return list of all towns
 }
 public List<County> findCounty(Town tn){
   // find county or counties for a given town
 }
 public boolean containsTown(Town tn){
   //check if town recorded
 }
 public boolean containsCounty(County cty){
 //check if county recorded
 }
 public Map<Town,TreeSet<County>> mapTownToCounty(){
   //return a map that maps towns to counties
 }
 Collection<TreeSet<Town>> listAllTownsA(){
 return data.values();
 }
 public String toString(){return data.toString();}
}
```

GRIFFITH COLLEGE DUBLIN

QUALITY AND QUALIFICATIONS IRELAND
EXAMINATION

DATA STRUCTURES AND ALGORITHMS

Lecturer(s):

External Examiner(s): **Thanh Thoa Pham Thi**

Date: **XXXXXXXX** **Time:** **XXXXXXX**

THIS PAPER CONSISTS OF TWELVE QUESTIONS
TEN QUESTIONS TO BE ATTEMPTED
ALL QUESTIONS CARRY EQUAL MARKS
APPENDIX AT THE BACK OF THE EXAMINATION PAPER

QUESTION 1

(a) Write a recursive function that implements Euclid's algorithm to find the greatest common divisor of two positive numbers. The algorithm is defined as follows:

$$\gcd(x, y) = \begin{cases} y, & if x = 0 \\ \gcd(y\%x, x), & if x != 0 \end{cases}$$

(5 marks)

(b) Test your solution by showing that: gcd(45,60) = 15.

(3 marks)

(c) What is the difference between a tail recursive function and non-tail recursive function?

(2 marks)

Total (10 marks)

QUESTION 2

(a) Using the statement execution times defined for HAL (See **Appendix** at the end of the exam paper), calculate the running time of the given code fragment.

```
int f[] = new int[1000];
for(int j = 0; j < f.length; j++){
  if(j % 2 != 0) f[j] = 1;
  else f[j] = 0;
}
```

(5 marks)

(b) Show that function sumN is *O(1)* and sumN1 is *O(n)*. What conclusion can be drawn from this analysis?

```
static long sumN(long n){
   long s = n*(n+1)/2;
   return s;
}

static long sumN1(long n){
   long s = 0;
   for(int j=0; j < n; j++) s=s+(j+1);
   return s;
}
```

(5 marks)

Total (10 marks)

QUESTION 3

(a) What do we mean by stating that a program whose cost function is $O(log_2 n)$ *performs better* that one that has a cost function of *O(n)*? Explain why binary searching *performs better* than linear searching.

(3 marks)

(b) Would you say that divide and conquer algorithms are *O(n)* or $O(log_2 n)$?

(1 mark)

(c) Draw a diagram to illustrate a linked list of integer values. The list should be constructed by entering the following list of numbers in the given order: 1, 4, 5, 6, 7, 2, 10, 12. Numbers should be inserted at the tail of the list.

(3 marks)

(d) In relation to the design of data structures explain what the term *genericity* means. Why is it important to make data structures *generic*?

(3 marks)

Total (10 marks)

QUESTION 4

(a) Write a function that sorts an array of integer values. You may use any sorting algorithm you have studied.

(7 marks)

(b) Analyse the performance of your chosen sorting function and contrast it with any other sorting function you have studied in your course.

(3 marks)

Total (10 marks)

QUESTION 5

Given below is the class StringList that uses a singly linked list to manage a collection of strings. New elements are inserted at the head of the list and the method add(String x) is given. The private class Node is used to implement nodes in the list and encapsulates both the data element x and a pointer to the next node in the list, if any. Its methods should be familiar to you from the work covered in lectures and labs. Your task is to complete the three methods whose signatures are given. Method add(String f[]) that inserts an array of strings in the list **(3 marks)**; method numChars() that counts the number of characters in all the strings in the list **(4 marks)** and method size() should return the number of strings in the list **(3 marks)**.

```
class StringList{
  Node head = null;
  public void add(String x){
          Node nw = new Node(x);
          if(head == null) head = nw;
```

```
            else{
              nw.setNext(head);
              head = nw;
            }
  }
  public void add(String f[]){..}
  public int numChars(){..}
  public int size(){..}
  private class Node{
   String data;
   Node next;
   public Node(String x){data = x; next = null;}
   public Node next(){return next;}
   public void setNext(Node p){next = p;}
   public String data(){return data;}
}
```

Total (10 marks)

QUESTION 6

(a) A stack is a *last in, first out* linear data structure. It is characterized by two main operations: push and pop. The push operation adds a new item to the top of the stack, or initializes the stack if it is empty. The pop operation removes the element at the top of the stack, if not empty. This means that elements are removed in inverse order to their insertion. Those last in get to leave first. To inspect the current element at the head of the stack a method top is provided. Given below is the generic class StackArray<E> that uses an ArrayList to implement stack behaviour. The methods size() and toString() are provided. Your task is to implement methods pop, push and top. (See **Appendix** at the end of the exam paper for relevant methods.)

```
class StackArray<E>{
 private ArrayList<E> stack = new ArrayList<>();
 …
 public int size(){return stack.size();}
 public String toString(){
      return stack.toString();
 }
}
```

(6 marks)

(b) Using your class StackArray write a code fragment that creates a stack of 10 randomly generated integer values such that the value at the top of the stack is always the largest value.

(4 marks)

Total (10 marks)

QUESTION 7

(a) Using class Function<T,R> write a function called square that takes an integer as argument and returns the square of its value. Write an assert statement to test your function.

(3 marks)

(b) Write a Predicate function called allEven that takes a list of integers as argument and returns true if the list contains only even numbers; false otherwise.

(2 marks)

(c) What are higher order functions?

(3 marks)

(d) Using the higher order method replaceAll from class ArrayList write a lambda expression as argument that multiplies all values in lst, given below, by 2.

ArrayList<Integer> lst = new ArrayList<>(Arrays.asList(2,3,4,5,6,7,8,9));

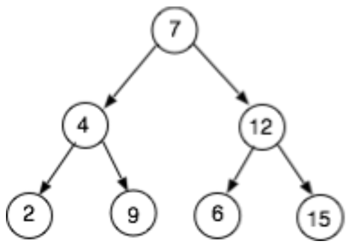
(2 marks)

(See **Appendix** at the end of the exam paper for relevant methods for this question.)

Total (10 marks)

QUESTION 8

(a) Explain why the tree given in the diagram below is a binary tree but not a binary search tree.



(2 marks)

(b) Using a diagram insert the following list of elements in a binary search tree:

6,3,8,7,2,0,10,1.

(3 marks)

(c) In the binary search tree, created for part b, list the order in which the nodes are visited under *preorder* and *postorder* traversals.

(4 marks)

(d) Name the traversal required that retrieves an ordered list.

(1 mark)

Total (10 marks)

QUESTION 9

Given below is a class called PersonHobbies that uses a map to model the relationship between persons and their hobbies. The constructor creates a default map with some sample persons and their hobbies.

Your tasks are:

(a) List the values of the set returned by the method persons();

(2 marks)

(b) List the values returned by the method hobbies();

(2 marks)

(c) Complete the method listPerson(String h) that takes a hobby as argument and returns those persons that participate in h;

(3 marks)

(d) Complete the method numHobbies(String p) that takes a person p as argument and returns the number of hobbies for p.

(3 marks)

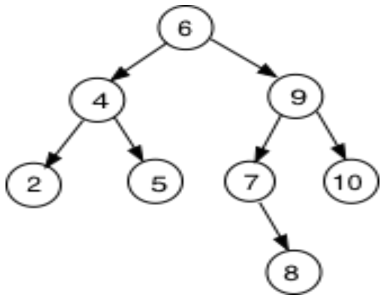
```
class PersonHobbies{
    private Map<String, List<String>> map = new TreeMap<>();
    public PersonHobbies(){
      map.put("John", new ArrayList<>(Arrays.asList("Football","Cinema","Golf")));
      map.put("Mary",new ArrayList<>(Arrays.asList("Cinema","Walking")));
      map.put("Sheila",new ArrayList<>(Arrays.asList("Golf")));
    }
    public Set<String> persons(){return map.keySet();}
    public Set<String> hobbies(){
     Set<String> tmp = new TreeSet<>();
     for(String n : map.keySet()) tmp.addAll(map.get(n));
         return tmp;
    }
    public Set<String> listPerson(String h){..}
    public int numHobbies(String p){..}
}
```

(See **Appendix** at the end of the exam paper for relevant methods for this question.)

Total (10 marks)

QUESTION 10

(a) Show that the *avl* tree given to the right is balanced.



(3 marks)

(b) What is the advantage of using *avl* trees over binary search trees?

(2 marks)

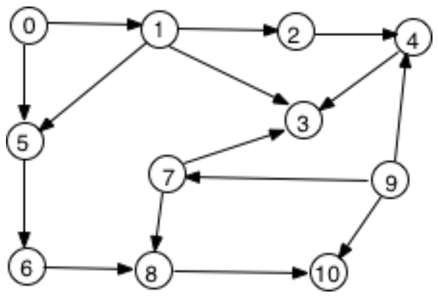
(c) Insert the given list of values in an *avl* tree. The list is: 7, 4, 10, 3, 5, 6, 2, 1. For full marks you must show the construction of the tree.

(5 marks)

Total (10 marks)

QUESTION 11

(a) Using the given graph below list the order of nodes visited using both a breadth first traversal and a depth first traversal.



(4 marks)

(b) Construct a B-tree with a maximum of 4 items per node for the list of numbers: *9, 20, 1, 12, 25, 7, 14, 21, 6, 5, 10, 18, 15, 11, 4*.

(6 marks)

Total (10 marks)

QUESTION 12

(a) Explain, with the aid of diagrams, how a hash table can be used to optimise, provide an *O(1)* solution, the cost of insertion and retrieval for data collections.

(5 marks)

(b) When you are planning to use the data structure HashSet to manage a set of objects what methods must your class implement? Why must you implement these methods? Why should the attributes used by these methods be immutable?

(See **Appendix** at the end of the exam paper for relevant methods for this question.)

(5 marks)

Total (10 marks)

Appendix

Calculating Running Times on HAL	
Statement	**Unit cost (ns)**
-, *, /, %, ^, <, >, ==, >=, <=, !=, =	*10ns*
Function invocation	*50ns*
Argument passing	*10ns* per argument
Return	*50ns*
if(b) s1; else s2	the cost of b plus the max cost of s1, s2
for, while loops	*totalCost = cost of initialization of variables + (n+1) * cost of evaluating guard on loop + n * cost of executing loop body, where n equals the number of iterations of the loop.*
new	*100ns*
Calculating array indices	*50ns*
Math.random()	*100ns*

Laws of *big-O*

The laws of *big-O are*:

1. **Summation**

   *O(1)+O(1)+..+O(1) = k * O(1) = O(1)*, where *k* is a constant.
   *O(n) + O(n)+..+O(n) = k * O(n) = O(n)*, where *k* is a constant
   O(n) + O(m) = max(O(n), O(m))
   e.g. $O(n^3) + O(n^5) = O(n^5)$

2. **Product**

   $O(n) * O(n) = O(n^2)$
   $n * O(n) = O(n^2)$
   *O(n) * O(m) = O(n * m)*
   *O(k * f(n)) = k * O(f(n)) = O(f(n))*, where *k* is a constant
   $O(n^a) * O(n^b) = O(n^{a+b})$

The *big-O* sets of order functions form a chain of sub-sets as follows:
$O(1) \ll O(\log_2 n) \ll O(n) \ll O(n * \log_2 n) \ll O(n^2) \ll O(n^k, k > 2) \ll O(a^n) \ll (n!)$

Constructor	ArrayList<E>() ArrayList<E>(Collection) LinkedList<E>() LinkedList<E>(Collection)
Insert item	add(E elem)
Insert list	addAll(Collection<? extends E> lst)
Remove item	remove(Object ob)
Contains item	Boolean contains(Object ob)
Number of elements	int size()
Convert to string	toString()
Empty set	Boolean isEmpty()
Remove elements	clear()
Retrieve element given index value	E get(int index);
Insert element at index	add(int index, E elem);
Change element at index	E set(int index, E elem);
Remove element at index	E remove(int index)
Get index of object	int indexOf(E elem);
Additional Methods for LinkedList class	
Add new element at head of list	addFirst(E elem)
Return element at head of list	E getFirst()
Remove element at head of list	E removeFirst()

Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array. If the list fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this list.	<T> T[] toArray(T[] a)
	
	An example is:
	ArrayList<Integer> lst = new ArrayList<>(Arrays.asList(3,2,6,9,1));
	Integer f[] = new Integer[lst.size()];
	f = lst.toArray(f);

Applies the given action function to all the elements in the list in order.	forEach(Consumer<? super E> action)
Removes all values that satisfy the given predicate filter	removeIf(Predicate<? super E> filter)

Replaces each element of this list with the result of applying the operator function op to that element.	replaceAll(UnaryOperator<E> op)
Sorts this list according to the order specified by the given Comparator cmp.	sort(Compaparator<? super E> cmp)

Constructor	ArrayDeque<E>() ArrayDeque<E>(Collection) ArrayDeque(int numElements)

Insert item	addFirst(E elem)
	addLast(E elem)

Get element without removing it – throws exception if queue empty	E getFirst() E getLast()
Get element without removing it – returns null is queue empty	E peekFirst() E peekLast()
Contains item	Boolean contains(Object ob)
Number of elements	int size()
Returns true if queue empty	Boolean isEmpty()
Convert to string	toString()
Empty set	Boolean isEmpty()
Remove elements	clear()
Retrieve head or tail element, returning null if queue empty	E pollfirst() E pollLast()

Returns an array containing all of the elements in this list in proper sequence; the runtime type of the returned array is that of the specified array. If the list fits in the specified array, it is returned therein. Otherwise, a new array is allocated with the runtime type of the specified array and the size of this list.	<T> T[] toArray(T[] a)
	
	An example is:
	ArrayDeque<Integer> dlst = new ArrayDeque<>(Arrays.asList(3,2,6,9,1));
	Integer f[] = new Integer[dlst.size()];
	f = dlst.toArray(f);

Constructor	HashMap<K,V>() HashMap <K,V>(Map<? extends K, ? extends V> mp) TreeMap<K,V>() TreeMap <K,V>(Map<? extends K, ? extends V> mp) EnumMap(Class<K> keyType)
Add or replace a key-value pair	put(K key, V value) putAll(Map<? extends K, ? extends V> mp)
If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.	V putIfAbsent(K key, V value)
Remove key-value pair and returns value associated with key, or null	V remove(Object key)
Replaces the entry for the specified key only if it is currently mapped to some value.	V replace(K key, V value)
Replaces the entry for the specified key only if currently mapped to the specified value.	boolean replace(K key, V oldValue, V newValue)
Contains key	boolean containsKey(Object key)
Contains value	boolean containsValue(Object value);
Number of elements	int size()
Convert to string	toString()
Empty set	boolean isEmpty()
Remove elements	clear()
Retrieve value	V get(Object key);
Retrieve the key set	Set <K> keySet();
Retrieve values	Collection<V> values();

Table of Specialized Functions

Function Name	**Argument Type**	**Return Type**	**Abstract Method Name**	**Purpose**
Function<T,R>	T	R	apply	Takes one argument and return a value of type R
BiFunction<T,U,R>	T,U	R	apply	Takes two arguments and return a value of type R
Supplier<T>	None	T	get	Takes no argument and return a value of type T
Consumer<T>	T	void	accept	Consumes a value of type T
BiConsumer<T,U>	T, U	void	accept	Consumes values of type T and U
UnaryOperator<T>	T	T	apply	A function that takes a value of type T as argument and returns a value of type T
BinaryOperator<T>	T, T	T	apply	A function that takes two values of type T as argument and returns a value of type T
Predicate<T>	T	boolea n	test	A function that takes a value of type T and returns a boolean value.
BiPredicate<T, U>	T, U	boolea n	test	A function that takes two arguments of type T and U and returns a boolean value.