CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computing@CIT

# ALGORITHMS AND DATA STRUCTURES

ASSIGNMENT 1: Data Structures for Database Management.
PART II: Java ADT, Generics and Reference-based Data Structures.

## 1. Background:

In the first part of the assignment you have become familiar with:

- The text file **database.txt**, which contains the information of a set of players.
- The Java class **myPlayer.java**, which is used to represent a football player.
- The Java interface **myList.java**, which so far contained:
    - The 5 basic operations of lists:
        1. my_create_empty
        2. my_get_length
        3. my_get_element
        4. my_add_element
        5. my_remove_element.
    - The 9 problem-specific operations for database management:
        1. load_file
        2. show_elements
        3. find_element
        4. show_info
        5. add
        6. update
        7. remove
        8. bubble_sort
        9. write_file
- The Java class **myListArrayList.java**, which implements the interface myList.java using a contiguous memory-based data structure: *ArrayList items*.
- The Java class **myMain_A01_Part1.java**, which creates the variable *myList m* to represent a database of players and triggers an interactive session with the user to perform different operations with the database.

**2. Redesigning our Application:**

The application design of part 1 is not fully satisfactory. Let's start from scratch and follow a top-down reasoning approach: How should we design our application?

1. *What are we doing in this application? What is the problem we are trying to solve?*
   Manage a football player database by supporting 9 concrete operations on it.

2. *Ok, what data do we need to manipulate?*
   A database of football players.

3. *Ok, so we need to represent in Java the concept "football player database".*
   We can create the Java Class "dbManagement". An object of this class would represent a concrete database of players. An object of this class is responsible of the management of the database it is representing, so it provides 9 methods to implement the required database operations.

4. *Ok, this sounds very good but…*
   *how does an object of dbManagement represent a concrete database of players?*

5. *So, first, we need to represent in Java the concept "football player".*
   We can create the Java Class "myPlayer". An object of this class would represent a concrete football player. And this is easy, a football player is represented by her/his name and her/his number of goals. A name can be represented by a String. A number of goals can be represented by an integer. So, myPlayer attributes are clear now!

6. *Coming back to question 4…*
   *how does an object of dbManagement represent a concrete database of players?*
   Now the answer to this question is clearer: As *a bunch* of myPlayer objects, one myPlayer object per player in the database!

7. *Can we be a little more specific? What do we mean with "a bunch"?*
   We want to:
   - ✓ Gather several myPlayer objects, placed in order one after another.
   - ✓ Know how many myPlayer objects we are gathering.
   - ✓ Access to each of them at any moment.
   - ✓ Add new myPlayers and remove existing ones.

8. *A list seems to be ideal for this case. Now, we know Java supports lists, but for this semester let's imagine it does not. So, we need to represent in Java the concept "list".*
   We create the new Abstract Data Type myList for it.
   - To define the public part we use a Java interface myList. This interface must provide the basic operations we described in the point *7* above:
     (1) my_create_empty, (2) my_get_length, (3) my_get_element,
     (4) my_add_element and (5) my_remove_element.

- To define the private part we use a Java class. An object of the class uses as its attribute the concrete data structure(s) to represent myList. An object of the class uses concrete algorithms to implement all the interface operations.

9. *Coming back to question 4…*
   *how does an object of dbManagement represent a concrete database of players?*
   Now it is completely clear: As a myList of myPlayer objects!
   So, dbManagement attribute is clear now!

10. *So, the application design is completely clear now.*
    *What do we need to make this application work?*
    *Or, in other words, what do we need to solve our original problem?*
    Create a main class myMain.java with a main method (main entry point).
    The main should create an object dbManagement ‘d’ and trigger an interactive session with the user (using ‘d’ to manage the database).

### 3. Java Files:

- **myMain_A01_Part2.java (Completed)**
  This class contains the main entry point of the application.
  - **static int select_db_manager(Scanner sc);**
    This method asks the user for the concrete myList implementation to be used: 1 for ArrayList, 2 for LinkedList, 3 for DoubleLinkedList.
  - **static int select_option(Scanner sc);**
    This method asks the user for a concrete database management operation to be performed.
  - **static void interactive_session();**
    This method creates the dbManagement d and triggers the user interactive session for managing the database.
  - **static void main();**
    This method is the main entry point of the application.
    It triggers interactive_session();

- **dbManagement.java (To be implemented)**
  This class represents a database of players.
  - **Attribute;**
    It uses a myList of myPlayers as its single attribute.
  - **dbManagement(int mode);**
    This method creates a new dbManagement object.
    If mode == 1 it uses the ArrayList-based implementation of myList.
    If mode == 2 it uses the LinkedList-based implementation of myList.
    If mode == 3 it uses the DoubleLinkedList-based implementation of myList.
  - **void load_file(String s);**
  - **void show_elements();**
  - **int find_element(String s);**
  - **void show_element(String s);**
  - **void add(String s, int i);**
  - **void update(String s, int g);**
  - **void remove(String s);**
  - **void bubble_sort();**
  - **void write_file(String s);**
    These are the methods you implemented in the first part of the assignment.
    Copy them to this file and tweak them so as to adapt them to the new class.

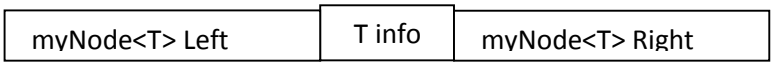
- **myPlayer.java (Completed)**
  This class represents a player. It is the same as in the first part of the assignment.
  - **Attributes;**
    private String name;
    private int goals;
  - **myPlayer(String s, int i);**
  - **String get_name();**
  - **int get_goals();**
  - **void set_name(String s);**
  - **void set_goals(int i);**
  - **void print_info();**
  - **boolean smaller(myPlayer player);**

- **myList.java (Completed)**
  This is myList ADT we are seeing in the lectures. It uses generics + exceptions.
  - //**myList<T> my_create_empty();** ← Commented.
  - **int my_get_length();**
  - **T my_get_element(String s) throws myException;**
  - **void my_add_element(int index, T element) throws myException;**
  - **void my_remove_element(int index) throws myException;**

- **myListArrrayList.java (Completed)**
  This is the ArrayList-based implementation of myList we are seeing in the lectures. It uses generics + exceptions.
  - **Attribute;**
    private ArrayList<T> items;
  - **myListArrayList();**
  - **int my_get_length();**
  - **T my_get_element(String s) throws myException;**
  - **void my_add_element(int index, T element) throws myException;**
  - **void my_remove_element(int index) throws myException;**

- **myException.java (Completed)**
  This is the class for defining our own exceptions we are seeing in the lectures.
  - **myException();**
    For throwing an exception with a fixed message.
  - **myException(String s);**
    For throwing an exception with a specific message.

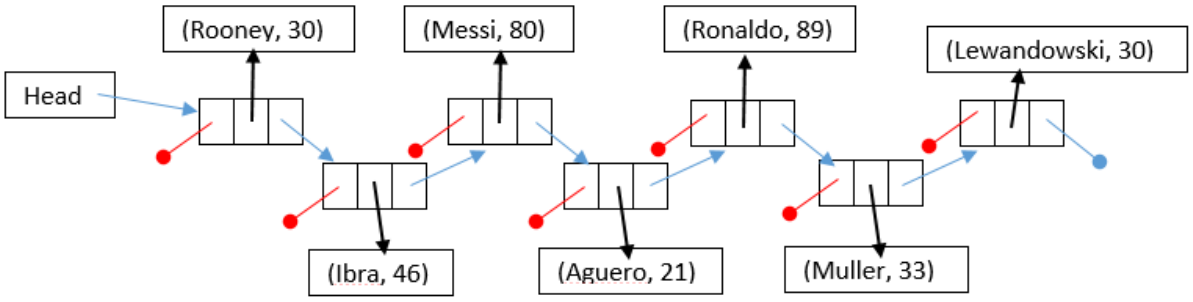
- **myNode.java (Completed)**
  This is the node representation for reference-based implementations of myList.
  It uses generics. Its graphic representation is as follows:




  - **Attribute;**
    private T info;
    private myNode<T> left;
    private myNode<T> right;
  - **myNode(T n);**
    The constructor just requires the information gathered by the node to set info to it. Both left and right are initialised to null.
  - **T getInfo();**
  - **myNode<T> getLeft();**
  - **myNode<T> getRight();**
  - **void setInfo(T n);**
  - **void setLeft(myNode<T> n);**
  - **void setRight(myNode<T> n);**
    The get and set methods allow to retrieve and update the attributes, resp.

- **myListLinkedList.java (Completed)**
  This is the LinkedList-based implementation of myList we are seeing in the lectures. It uses generics + exceptions.
  - **Attribute;**
    private myNode<T> head;
  - **myListLinkedList();**
  - **int my_get_length();**
  - **T my_get_element(String s) throws myException;**
  - **void my_add_element(int index, T element) throws myException;**
  - **void my_remove_element(int index) throws myException;**

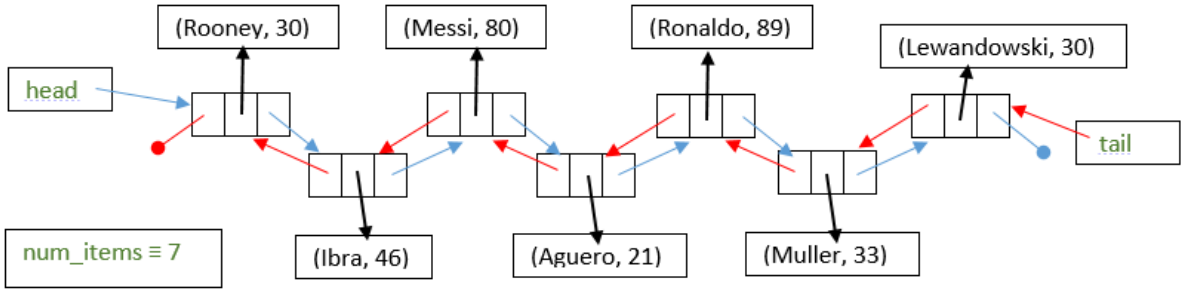
  The following example shows the myListLinkedList representation of the database contained in database.txt:




**Interesting features:**
- This class is provided completed, so as to serve you as an example for implementing the myListDoubleLinkedList.java class.
- As you can see, the attribute Left of each node is unused.
- head points to the first myNode<myPlayer> of myList.
- Each myNode of the list points to:
  - An object info. In this case, an object of type myPlayer. For example, the first node of the list points to the object myPlayer with name Rooney and 30 goals. In the example these pointers are marked with black colour.
  - The right myNode of the list, i.e., the one placed at its right hand side. In the example these pointers are marked with blue colour.
  - The left myNode of the list, i.e., the one placed at its left hand side. In the example these pointers are marked with red colour.
- Null pointers are represented with circles

- **myListDoubleLinkedList.java (To be Implemented)**
  This class implements a DoubleLinkedList-based implementation of myList.
  It uses generics + exceptions.
  - **Attribute;**
    private myNode<T> head;
    private myNode<T> tail;
    private int num_items;
  - **myListDoubleArrayList();**
  - **int my_get_length();**
  - **T my_get_element(String s) throws myException;**
  - **void my_add_element(int index, T element) throws myException;**
  - **void my_remove_element(int index) throws myException;**

  The following example shows the myListDoubleLinkedList representation of the database contained in database.txt:




**Interesting features:**
- head points to the first myNode<myPlayer> of myList.
- tail points to the last myNode<myPlayer> of myList.
- First myNode.left points to null and last myNode.right points to null.
- Each myNode of the list points to:
  - An object info. In this case, an object of type myPlayer. For example, the first node of the list points to the object myPlayer with name Rooney and 30 goals. In the example these pointers are marked with black colour.
  - The right myNode of the list, i.e., the one placed at its right hand side. In the example these pointers are marked with blue colour.
  - The left myNode of the list, i.e., the one placed at its left hand side. In the example these pointers are marked with red colour.
- Null pointers are represented with circles

**Extra Requirement:**
The goal of a double linked list is to provide an optimised version of the operations my_get_element, my_add_element and my_remove_element. Thus, if the index we want to access to belongs to the first half of myList, the we start traversing the nodes from head; otherwise, we start traversing the nodes from tail. Note now we have an extra attribute num_items that tell us the amount of myNode of myList.

## 4. Marking Scheme and Submission Date:

The goal of the assignment is to complete the following 2 Java classes:

- dbManagement.java (10 marks).
    1. Attribute + Constructor → 1 mark.
    2. void load_file(String s); → 1 mark.
    3. void show_elements();→ 1 mark
    4. int find_element(String s); → 1 mark
    5. void show_element(String s); → 1 mark
    6. void add(String s, int i); → 1 mark
    7. void update(String s, int g); → 1 mark
    8. void remove(String s); → 1 mark
    9. void bubble_sort();→ 1 mark
    10. void write_file(String s); → 1 mark

- myListDoubleLinkedList.java (10 marks)
    1. Attribute + Constructor + int my_get_length() → 1 mark.
    2. T my_get_element(int index) throws myException; → 3 marks
    3. void my_add_element(int index, T element) throws myException; → 3 marks
    4. void my_remove_element(int index) throws myException; → 3 marks

The remaining files of the application are completed, so you do not have to edit them.

**Submission instructions:** Submit the application via Blackboard with a single zip file containing dbManagement.java and myListDoubleLinkedList.java completed. Please, do not include the remaining files, just these two!

**Submission deadline:** Sunday 20th March, 11:59pm.