# Assignment 1

Value **40%** of Coursework

**Individual work**

# Learning outcomes

*Students will be able to understand*

1.1 Data structures

1.2 The applications of data structures

1.3 Object-oriented programming concepts

1.4 Methods for program testing

*Students will have acquired skills in:*

2.1 Data abstraction

2.2 The use of data structures

2.3 Programming at a more advanced level in a high-level object-oriented language

2.4 Program testing and documentation

*Students will have acquired skills in:*

3.1 Self-management

3.2 Learning

3.3 Communication

3.4 Problem solving

3.5 Information technology

# Process and what submit to Student Website

The assignment submitted should be compressed into a .zip or .rar file, the following files should be contained in the compressed file：

* a *report* as a *Microsoft Word* document containing text of all your classes.

filename format: *12345678\_CHC5223\_CW1\_Report.docx*

* a .*zip or .rar file* containing the project: the runnable *jar* file (if available) and all the program’s source texts (*.java*), including those provided

filename format: *12345678\_CHC5223\_ CW1\_Files.zip/rar*

# Requirement types

* Functional requirements (FRs) specify how the software must behave.
* Operational requirements (ORs) specify aspects such as efficiency.
* Developmental requirements (DRs) specify how your software must be developed.

# General requirements

**DR1**: All your programming must conform to “Java Conventions and Programming Guidelines”

**DR2:** You must paste the source code of all your classes into your report, as *text*, not images.

# Introduction

The topic of this assignment is **hash tables**.

This assignment requires you to implement **two** provided Java interfaces (see Appendixes). The interface *IMember* describes methods for retrieving details about a member of an association.

The interface *IMemberDB* describes methods for an abstract data type (ADT) which holds a database of Member objects.

You must *implement* *IMemberDB* as a hash table for Assignment 1.

You will implement *IMemberDB* as a binary search tree for Assignment 2.

You **must not** make any changes to these interfaces.

# Requirements

## Task 1

**FR 1**: You must create a Java class *Member* that implements the interface *IMember*.

**FR2:** You must also implement a constructor or constructors for this class.

4 marks

## Task 2

**FR3**: You must create a Java class called *MemberHash* that implements the interface *IMemberDB*.

**DR3:** You must implement *MemberHash* as a hash table

**DR4:** The constructor for *MemberHash* must print the string “Hash Table” to *System.out*.

**DR5:** You **must** use a Java *array* for implementing the hash table (not an array list or other collection class). You **must not** encapsulate existing implementations of collections in your submission. For example, you **must not** create a *HashMap* object and call methods on that object from your class. Failure to comply with this will result in **zero marks** for this part.

You may take a ‘lazy’ approach to deletion, such as marking the item as no longer present.

Take care that you have not used a linear search *O(n)* in the hash table where you should have used hashing, aiming towards *O(1).*

6 marks

**DR6**: You must devise your own *hash function* that will work well for surnames (family names)

You may assume that the names are expressed solely in uppercase ISO basic Latin alphabet (ASCII). You may also assume that enough additional parts are appended to names so that no two members have exactly the same name. You need not regard uppercase and lowercase as the same.

You will need to perform some experiments to find a good hash function. Include an explanation of your hash function and your experiments in your report.

Take care to avoid *numeric overflow* when calculating your hash function. This can be done by applying a mod (%) operation at early stages of the calculation, rather than just at the end.

4 marks

**DR7:** You must not use the Java built-in *hashCode* method, though you can experiment with it.

**FR4**: You must ensure that *collisions* are catered for well and do not lead to excessive clustering. You will need to perform some experiments to find a good collision-resolution strategy. Include an explanation of your collision-resolution strategy and your experiments in your report.

**DR8**: You must *chaining* with a linked list implemented devised by you (not using the *LinkedList* or *ArrayList* or any other class from the Java library). You will need to make a linear search within the chain.

4 marks

## Task 3

**DR9**: You must make appropriate use of assertions (*assert* statements) to protect preconditions of the operations. Remember to enable assertion checking for your project.

4 marks

## Task 4

**DR10**: Your class must keep track of the ‘load factor’ of the hash table and display this after each insertion or deletion. Note that with chaining the load factor can exceed 100%.

2 marks

## Task 5

**DR11**: You must make your class *log* monitoring information, either to a text file or by calls of *System.out.println*.

It must log (at least):

* the *size* and *load factor* of the hash table after each addition;
* for every *addition* of a Member (*put*), attempt to *get* the Member or attempt to *remove* the Member:
  + the *Member name;*
  + the *hash value* calculated for it ;
  + the sequence of buckets (array locations and list nodes) visited.
* Paste your log into your report.

4 marks

We have supplied a main program *CHC5223.java* for your use with this assignment. It has a very simple textual user interface to perform operations on the database. The user types the first letter of the menu option:

*D)isplay P)ut G)et C)ontains S)ize R)emove Q)uit?*

The program can load records for the database from a *csv* file, where column A is the *family-name,* column B is *the fore-name(s)* and column C is the *affiliation*. The program treats family-name + “,” fore-names as the member’s name.

The name of the file is set in the static variable *filename*.

Sample files *sampleMembersUK.csv* and *sampleMembersUS.csv* each contain 500 members in this format.

## Task 6

**DR12:** You must devise a test plan for your implementation. Be sure to check (among many other cases):

* each of the implemented methods is working correctly
* that values added can subsequently be found
* that existing values are not lost when other values are added
* that deleting an item does not disturb the other items

The test plan must include, for each test case:

* Description of case to be tested
* Value(s) to be used
* Expected results
* Actual results (placeholder for Task 6)

4 marks

## Task 7

**DR13:** By using the supplied main program, or by other means, you must test your *MemberHash*. Set the capacity of the hash table to a small value so that collisions are sure to occur.

(You will use the same main program for Assignment 2 simply by replacing the implementation of *IMemberDB* that uses a hash table by the one that uses a binary search tree.)

Include your test plan, test data used, expected results and actual results in your report. You must show your actual results and the logging information copied from your log file or the output pane of your IDE. Do not simply state “test passed”, or similar – show evidence!

6 marks

## Task 8

**DR14**: You must state honestly which of the requirements of Assignment 1 you have successfully fulfilled, citing evidence. Also comment on the *time efficiency* and *space efficiency* of your implementation of the hash table.

2 marks

total 40 marks

# Relevant quotation

*“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”*

Professor Sir Tony Hoare   
*1980 Turing Award Lecture; Communications of the ACM 24 (2), (February 1981): pp. 75-83*

Please try to do this the first way.

# Obtaining help

It is always permissible to request advice on what is required for this assignment. Please try to do this during normal contact time and avoid asking for such help in the last week before the deadline.

You can discuss the requirements and the material covered in the assignment with others but what you create must be all your own work. Be careful to avoid collision.

Declare in your report any help you have received other than that from the module teaching team.

# Feedback

In addition to the written feedback that we aim to provide within the normal interval, you will be able to obtain fast, brief, verbal formative feedback and help on correcting your work at your practical classes.

# Appendix 1: *IMember* interface

package CHC5223;

/\*\*

\*

\* DO NOT CHANGE THIS INTERFACE

\* You must create a class that implements this interface

\*

\* objects of a class implementing this interface holds information

\* about a Member

\*/

public interface IMember {

/\*\*

\* Retrieves the name of the member

\* @pre true

\* @return the name of the member

\*/

public String getName();

/\*\*

\* Retrieves the member's affiliation

\* @pre true

\* @return the member's affiliation

\*/

public String getAffiliation();

@Override

/\*\*

\* @return the member's name and affiliation as string

\*/

public String toString();

}

# Appendix 2: *IMemberDB* interface

package CHC5223;

/\*\*

\*

\* an objects of a class implementing this interface holds a

\* database of member information

\* DO NOT CHANGE THIS INTERFACE

\* You must create a class that implements this interface

\*

\*/

public interface IMemberDB {

/\*\*

\* Empties the database.

\* @pre true

\*/

public void clearDB();

/\*\*

\* Determines whether a member's name exists as a key inside the database

\* @pre name is not null and not empty string

\* @param name the member name (key) to locate

\* @return true iff the name exists as a key in the database

\*/

public boolean containsName(String name);

/\*\*

\* Returns a Member object mapped to the supplied name.

\* @pre name not null and not empty string

\* @param name The Member name (key) to locate

\* @return the Member object mapped to the key name if the name

exists as key in the database, otherwise null

\*/

public Member get(String name);

/\*\*

\* Returns the number of members in the database

\* @pre true

\* @return number of members in the database.

\*/

public int size();

/\*\*

\* Determines if the database is empty or not.

\* @pre true

\* @return true iff the database is empty

\*/

public boolean isEmpty();

/\*\*

\* Inserts a Member object into the database, with the key of the supplied

\* member's name.

\* Note: If the name already exists as a key, then then the original entry

\* is overwritten.

\* This method must return the previous associated value

\* if one exists, otherwise null

\*

\* @pre member not null and member name not empty string

\*/

public Member put(Member member);

/\*\*

\* Removes and returns a member from the database, with the key

\* the supplied name.

\* @param name The name (key) to remove.

\* @pre name not null and name not empty string

\* @return the removed member object mapped to the name, or null if

\* the name does not exist.

\*/

public Member remove(String name);

/\*\*

\* Prints the names and affiliations of all the members in the database in

\* alphabetic order.

\* @pre true

\*/

public void displayDB();

}