FACULTY OF COMPUTER SCIENCE
Programming II - Course 2021/22

Practical 1: Instructions

1. The Problem

This practical consists of implementing the main features of BIDFIC, an online auction website. To do this, it will be necessary to design a data structure capable of storing all the information associated with the products put on sale. In this first practical, students will develop the product management system, including operations for adding and removing products and for modifying their prices.

The aim of this work is to practice the concept of independence of implementation in the case of Abstract Data Types (ADTs). The student is asked to create two different implementations of an UNORDERED LIST, a STATIC implementation and a DYNAMIC implementation, which must work in a fully interchangeable way. So, the main program must not make any assumptions about the way an ADT is implemented.

2. Header File Types

Some data types will be defined in this header file (`types.h`) since they are necessary to solve the problem and they are used by both the ADT and the main program.

`NAME_LENGTH_LIMIT`	Maximum length of user and product identifiers (constant)
`tUserId`	User identifier (`string`)
`tProductId`	Product identifier (`string`)
`tProductCategory`	Category of a product (enumerated type: `{book, painting}`)
`tProductPrice`	Price of a product (`float`)
`tBidCounter`	Bid counter (`int`)
`tItemL`	Data for an element of the list (a product). It contains: • `seller`: type `tUserId` • `productId`: type `tProductId` • `productCategory`: type `tProductCategory` • `productPrice`: type `tProductPrice` • `bidCounter`: type `tBidCounter`

3. ADT List

The system will make use of an ADT List to hold the list of products and their associated data. Two different implementations of this ADT will be created:

1. A **STATIC** one using arrays (`static_list.c`) with a maximum size of 25 elements.
2. A singly-linked **DYNAMIC** one using pointers (`dynamic_list.c`).

3.1. Data types included in the ADT List

`tList`	Represents a list of products
`tPosL`	Position of an element in the list
`LNULL`	Constant used to represent null positions

3.2. Operations included in the ADT List

A common precondition for all these operations (except `CreateEmptyList`) is that the list must be previously initialised:

- `createEmptyList (tList) → tList`
  Creates an empty list.
  PostCD: The list is initialised and has no elements.

- `isEmptyList (tList) → bool`
  Determines whether the list is empty or not.

- `first (tList) → tPosL`
  Returns the position of the first element of the list.
  PreCD: The list is not empty.

- `last (tList) → tPosL`
  Returns the position of the last element of the list.
  PreCD: The list is not empty.

- `next (tPosL, tList) → tPosL`
  Returns the position in the list of the element following that one at the indicated position (or `LNULL` if the specified position has no next element).
  PreCD: The indicated position is a valid position in the list.

- `previous (tPosL, tList) → tPosL`
  Returns the position in the list of the element preceding that one at the indicated position (or `LNULL` if the specified position has no previous element).
  PreCD: The indicated position is a valid position in the list.

- `insertItem (tItemL, tPosL, tList) → tList, bool`
  Inserts an element containing the provided data item in the list. If the specified position is `LNULL`, then the element is added at the end of the list; otherwise, it will

be placed right before the element currently holding that position. **It the element could be inserted, the value `true` is returned, `false` otherwise.**
PreCD: The specified position is a valid position in the list or a `LNULL` position.
PostCD: The positions of the elements in the list following that of the inserted one may have varied.

- `deleteAtPosition (tPosL, tList) → tList`
  Deletes the element at the given position from the list.
  PreCD: The indicated position is a valid position in the list.
  PostCD: The positions of the elements in the list following that of the deleted one may have varied.

- `getItem (tPosL, tList) → tItemL`
  Returns the content of the element at the indicated position.
  PreCD: The indicated position is a valid position in the list.

- `updateItem (tItemL, tPosL, tList) → tList`
  Modifies the content of the element at the indicated position.
  PreCD: The indicated position is a valid position in the list.
  PostCD: The order of the elements in the list has not been modified.

- `findItem (tProductId, tList) → tPosL`
  Returns the position **of the first element in the list** whose product identifier matches the given one (or `LNULL` if there is no such element).

4. Description of the task

The task consists of implementing a single main program (`main.c`) to process the requests received from BIDFIC users, which follow this format:

`N productId userId productCategory productPrice`	***[N]ew:*** A new product is added
`D productId`	***[D]elete***: The product is removed
`B productId userId productPrice`	***[B]id***: Bid for a given product
`S`	***[S]tats:*** Current list of products and their data

The main program will contain a loop to process, one by one, the requests of the users. In order to simplify the development and testing of the system, the program must not prompt the user to input the data of each request. Instead, the program will take as input a file containing the sequence of requests to be executed (see document `RunScript.pdf`). For each loop iteration, the program will read a new request from the file and then process it. In order to make correction easier, all requests in the input file have been numbered consecutively.

For each line of the input file, the program will do the following:

1. A header with the operation to be performed is shown. This header consists of a first line with 20 asterisks and a second line indicating the operation as shown below:

```
********************
NN␣T:␣product␣PP␣seller/bidder␣UU␣category␣CC␣price␣ZZ
```

where `NN` is the number of the request; `T` is the type of operation (`N`, `D`, `B` or `S`); `PP` is the identifier of the product; `UU` is the identifier of the user who is selling the product (*seller*) in the case of a `[N]ew` request, or bidding for it (*bidder*) in the case of `[B]id`; `CC` is the category of the product; `ZZ` is the price of the product (with 2 decimal places); and ␣ represents a blank. Only the necessary parameters are printed; i.e., for a `[S]tats` request we will only show "`01 S`", while for a `[N]ew` request we will show "`01 N: product Product1 seller User2 category book price 15.00`".

2. The corresponding request is processed:

- If the operation is **`[N]ew`**, that product must be added **at the end** of the product list, with its corresponding product identifier, the user identifier of its seller, its category and price. Its bid counter will be initialized to 0. In addition, a message like this will be displayed:

  ```
  * New: product PP seller UU category CC price ZZ
  ```

  The rest of messages follow the same format.

  In the event that a product with such product identifier already exists, or the insertion cannot be completed, the following message will be printed:

  ```
  + Error: New not possible
  ```

- If the operation is **`[D]elete`**, the system will locate and remove that product from the list. In addition, a message like this will be displayed:

  ```
  * Delete: product PP seller UU category CC price ZZ bids II
  ```

  where `II` is the number of bids received by that product.

  In the event that there is no product with the given identifier, the following message must be printed:

  ```
  + Error: Delete not possible
  ```

- If the operation is **`[B]id`**, the product is located, its price is updated to the bid value, and its bid counter is incremented by 1. In addition, a message like this, containing the updated values of its bid counter and price, will be displayed:

  ```
  * Bid: product PP seller UU category CC price ZZ bids II
  ```

If there is no product with such identifier, the bidder and the seller are the same person, or the bid value is not higher than the current price, the following message must be printed:

```
+ Error: Bid not possible
```

- If the operation is **`[S]tats`**, the whole list of current products will be displayed as follows:

```
Product PP1 seller UU1 category CC1 price ZZ1 bids II1
Product PP2 seller UU2 category CC2 price ZZ2 bids II2
...
Product PPn seller UUn category CCn price ZZn bids IIn
```

Below this list we will also print a table showing, for each product category, the number of products in that category, the sum of their prices, and their average price. Such table must follow the following format:

```
Category  Products    Price  Average
Book      %8d %8.2f %8.2f
Painting  %8d %8.2f %8.2f
```

Where `%8d` indicates that the corresponding integer is right justified (with 8 digits) and `%8.2f` indicates a total size of 8 digits, with 2 decimal places. In the event that the user list is empty, the following message must be printed:

```
+ Error: Stats not possible
```

5. Reading the input files

To facilitate the development of this practical, the following materials are provided: (1) a folder `CLion` that includes a template project (`P1.zip`) along with a file that explains how to use it (`Howto_use_IDE.pdf`); and (2) a folder `script` which contains a file (`script.sh`) that allows you to test the system with all the test files supplied at once. A document explaining how to run it is also provided (`RunScript.pdf`). Finally, to avoid problems when running the `script`, it is recommended **NOT to directly copy-paste the text of this document into the source code**, since the PDF format may include invisible characters that may result in (apparently) valid outputs to be considered incorrect.

6. Important Information

The document `DeliveryGuidelines.pdf`, available on the course website, clearly outlines the delivery guidelines to be followed. For an adequate **follow-up of this practical**, two **mandatory partial deliveries** will be made before the deadlines and with the contents indicated below:

- Checkpoint #1: Friday **March 4th**, at 22:00. Implementation and testing of the dynamic version of the list ADT (submission of files `types.h`, `dynamic_list.c` and `dynamic_list.h`).

- Checkpoint #2: Friday **March 11th**, at 22:00. Implementation and testing of the static version of the list ADT (submission of files `types.h`, `static_list.c` and `static_list.h`).

To check the correct operation of the ADT implementations, the test file `test_list.c` is provided. The implementations delivered by the students will be assessed automatically. For this purpose, the script provided will be used to check whether the corresponding checkpoint is passed or not (see document `AssessmentCriteria.pdf`).

Final submission deadline: Friday, **March 25th**, at 22:00.