FACULTY OF COMPUTER SCIENCE

Programming II - Course 2019/20

Practical Exercise #1: Instructions

1. The problem

This first practical exercise consists of implementing the counting system for the electronic voting machines that will be used in the polling stations during the next elections. The system will take account of the different political parties that have received votes in a certain polling station, the number of votes given to each party, the amount of null votes and the turnout. The system must also allow for making parties illegal.

The aim of this work is to practise 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 a name (constant)
`tPartyName`	Name (acronym) of the political party (`string`).
`tNumVotes`	Number of votes received (`int`)
`tItemL`	Data for a party. It contains: • `partyName`: of type `tPartyName` • `numVotes`: of type `tNumVotes`

3. ADT List

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

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

3.1. Data types included in the ADT List

`tList`	Represents a list of parties
`tPosL`	Position of an element of 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 following that one we indicate (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 preceding the one we indicate (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 subsequent to the inserted one may have changed their values.

- `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 deleted position and the positions of the elements of the list subsequent to it may have changed their values.

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

- `updateVotes (tNumVotes, tPosL, tList) → tList`
  Modifies the number of votes of the element at the position we indicate.
  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 (tPartyName, tList) → tPosL`
  Returns the position **of the first element in the list** whose party name is the one indicated (or `LNULL` if there is no such item).

4. Description of the task

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

`N partyName`	***[N]ew***: A new party is added with its number of votes set to 0.
`V partyName`	***[V]ote***: Adds one vote to the given party.
`I partyName`	***[I]llegal:*** Illegalises a party. The party is removed from the list and all its votes become null votes.
`S totalVoters`	***[S]tats:*** Calculates the statistics of turnout and votes received by every party. `totalVoters` stands for the total number of voters registered in the polling station.

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:

```
********************
CC_T:_party/totalvoters_XX
```

where CC is the number of the request; T is the type of operation (N, V, I or S), XX is the acronym of the party (partyName) or the number of voters in the electoral list (totalVoters), as appropriate; and ␣ represents a blank. Notice that only the necessary parameters are printed; for example, for a [S]tats request we would show "01 S: totalvoters 145", while for a [N]ew request we would show "02 N: party PBF".

2. The corresponding request is processed:

- If the operation is **[N]ew**, that party must be added **at the end** of the party list with its number of votes initialized to 0. In addition, a message like this will be displayed:

  ```
  *␣New:␣party␣XX
  ```

  where, again, XX is the partyName and ␣ represents a blank. The rest of messages follow the same format.

  If a party with that *partyName* already exists, the following message will be printed:

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

  It is not possible to incorporate new parties (request N) once the voting process has started (request V). All the test files provided, as well as those that will be used for the assessment, fulfill these two premises; thus, it is not necessary to check it.

- If the operation is **[V]ote**, the given party will be located, its vote counter will be incremented by 1 and the following message will be displayed:

  ```
  *␣Vote:␣party␣XX␣numvotes␣YY
  ```

  In the event that there is no party with that *partyName*, the following message must be printed:

  ```
  +␣Error:␣Vote␣not␣possible.␣Party␣XX␣not␣found.␣NULLVOTE
  ```

  and this vote will be counted as a NULL vote. Note than although there is no party representing such NULL vote, it is necessary to account for these votes.

- If the operation is **[I]llegalize**, the system will locate and remove that party from the list, and its votes will become NULL votes. A message like this will be displayed:

  ```
  *␣Illegalize:␣party␣XX
  ```

  In the event that there is no party with that *partyName*, the following message must be printed:

  ```
  +␣Error:␣Illegalize␣not␣possible
  ```

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

```
Party_XX1_numvotes_YY1_(ZZ1%)
Party_XX2_numvotes_YY2_(ZZ2%)
        …
Party_XXn_numvotes_YYn_(ZZn%)
Null votes NN
Participation:_KK_votes_from_VV_voters_(PP%)
```

  where `XX` is the `partyName`, `YY` is the absolute number of votes received, `ZZ` is the percentage of votes received by the party, `NN` is the number of null votes, `KK` is the number of voters that have participated, `VV` is the total number of voters (`totalVoters`) and `PP` is the turnout.

  The percentages of votes received by each party are calculated on the total of valid votes (i.e. without including null votes). No percentage is given for the null vote (only the absolute number) since it does not count as a valid vote.

  The turnout (i.e. the percentage of voters that have participated) is calculated as the proportion of people who have exercised their right to vote (votes to parties + null votes) on the total of possible voters of the electoral register. This later number is provided within the `[S]tats` request itself.

5. Running the program

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

Submission deadline: **Friday March 20, 2020 at 22:00.**
Date of the **checkpoint test**:

- Week of March 2-6: Using the list implementation provided, implementation and testing of part of the operations of `main.c`: **[N]ew**, **[S]tats** and **[V]ote**.