Practical 7 – Binary Search Trees. Binary Trees.

Instructions

- Download the ***aeda2021_p07.zip*** file from the moodle page and unzip it (contains the ***lib*** folder, the ***Tests*** folder with files ***dictionary.h, dictionary.cpp, bst.h, dic.txt, binaryTree.h, game.h*** and ***game.cpp*** and ***tests.cpp*** and the ***CMakeLists*** and ***main.cpp*** files)

You must perform the exercise respecting the order of the questions

Exercise

1) Electronic dictionaries are very useful tools. But if they are not implemented in an appropriate structure, their handling can be quite time consuming. We intended to implement a dictionary using a binary search tree (BST) where the words are ordered alphabetically. Consider that the tree contains objects of the class **WordMeaning:**

```
class WordMeaning
{
    string word;       // the word
    string meaning;  // meaning of the word (explanation)
public:
    WordMeaning(string w, string m): word(w), meaning(m){}
    string getWord() const { return word; }
    string getMeaning() const { return meaning; }
    void setMeaning(string m) { meaning=m; }
};
```

We want to implement the class **Dictionary**, as follows:

```
class Dictionary {
    BST<WordMeaning> words;
public:
    BST<WordMeaning> getWords() const;
};
```

a) Complete the class **Dictionary**, properly declaring the data member `words`. Also implement the member function:

```
void Dictionary::readDictionary (ifstream &f)
```

that reads, from a file, the words and their meaning and stores that information in the tree. The file contains an even number of lines, where the first line contains the word and the next line its meaning

```
cat
feline mammal
banana
fruit
...
```

Note:

To access files in your program (you need to read the *dic.txt* file), you can:

a) specify the absolute path, or

b) change "Working directory" in CLion (Run -> Edit Configurations… -> Working directory) to the folder where the files are located, or

c) add the directive to *CMakeLists.txt*

set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_CURRENT_SOURCE_DIR}/Tests")

in this case, the compiled and read/write files are placed in the *project*/Tests

---

b) Implement the following member function in class **Dictionary**:

```
void Dictionary::print() const
```

that writes the contents of the dictionary alphabetically ordered by words, on the monitor, in the format:

```
word1
meaning of word1
word2
meaning of word2
....
```

c) Implement the following member function in class **Dictionary:**

```
string Dictionary::searchFor(string word) const
```

which returns the meaning of the word indicated as an argument. If the word does not exist, it throws the **WordInexistent** exception. This class has the following methods that you must implement:

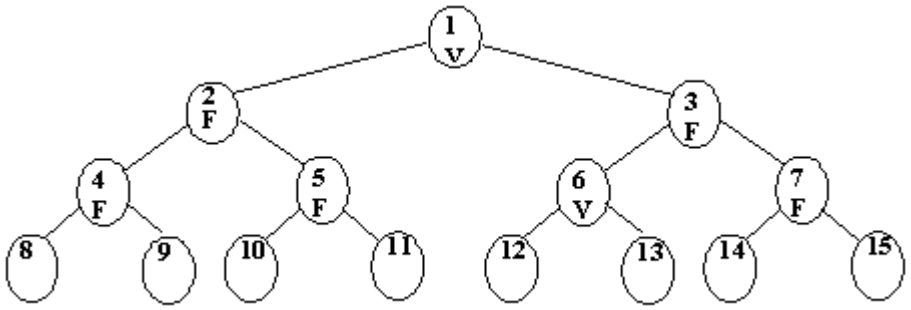
```
string WordInexistent::getWordBefore() const
// returns the word immediately before

string WordInexistent::getMeaningBefore() const
// returns the meaning of the word immediately before

string WordInexistent::getWordAfter() const
// returns the word immediately after

string WordInexistent::getMeaningAfter() const
// returns the meaning of the word immediately after
```

d) Implement the following member function in class **Dictionary**:

```
bool Dictionary::correct(string word, string newMeaning)
```

which changes the meaning of the word `word` to a new meaning indicated in the argument (`newMeaning`). If the word for which we want to change the meaning exists, the method returns true, otherwise this new word is added to the dictionary and the method returns false.

2) A ball is thrown over a set of circles arranged in the form of a complete binary tree (see the figure).



(*note*: a complete binary tree has all its levels completely filled)

Each circle has a score (points) and a state represented by a boolean value that indicates which path the ball takes when it reaches that circle: if the state is equal to false, the ball goes to the left; if it is equal to true, the ball goes to the right. When the ball passes through any circle, it changes its state: if it was true, it becomes false; if it was false, it becomes true. Whenever the ball passes in a circle, the number of visits to that circle is incremented (initially is equal to 0). When the ball hits a circle at the bottom (leaf of the tree), the player who launched the ball wins the number of points in that circle. The player who wins more points in a series of n rounds wins the game.

Use a binary tree (**BinaryTree**) to represent the set of circles that make up the game board (**Game** class). The information included in each node of the tree is represented in the **Circle** class:

```
class Circle {
    int points;
    bool state;
    int nVisits;
public:
    Circle(int p=0, bool s=false): points(p), state(s), nVisits(0) {}
    //…
};

 class Game {
   BinaryTree<Circle> game;
public:
   BinaryTree<Circle> &getGame () { return game; }
};
```

a) Implement the constructor of the **Game** class, which creates a game board:

```
Game::Game(int h, vector<int> &points, vector<bool> &states )
```

This function creates a complete binary tree, with height equal to *h*. The *points* and *states* vectors represent the score and the state of the circles (nodes of the tree), when making a visit by level.

Note: If you number the position of the nodes in a tree, traversed by level, from 0 to n-1 (n = total number of nodes in the tree), the node in position p has the left child and the right child in positions *2*p+1* e *2*p+2*, respectively.

b) Implement the function that describes the state of the game at a given time:

```
string Game::writeGame()
```

This function returns information about the entire game in the format *"points-state-nVisits\nl"* for each of the circles, where *points* is an integer, *state* is "true" or "false" and *nVisits* is the number of visits already made to that circle, when the tree is traversed by level.

c) Implement the function that performs a move:

```
int Game::move()
```

This function makes a move, according to the rules already described, and must change the state and increase the number of visits from all circles through which the ball passes. The function returns the score of the circle (tree leaf) where the thrown ball ends its course.

Tip: Use an iterator per level to traverse the tree.

d) Implement the function that determines what is the most visited circle:

```
int Game::mostVisited()
```

This function returns the number of visits made to the most visited circle so far, of all the moves already made (with the exception of the tree root, of course).