# 5th Recitation

## Binary Search Trees (BST)

Instructions:

- Download the file `aed2324_p05.zip` from the Moodle page and unzip it (it contains the folder `lib`; the folder `Tests` folder with the files `dictionary.cpp`, `dictionary.h`, `funSetProblem.cpp`, `funSetProblem.h` and `tests.cpp`; and the `CMakeLists` and `main.cpp` files).
- On CLion, create a project by selecting the folder containing the extracted files above.

**1.** Electronic dictionaries are very useful tools. We want to implement a dictionary using a binary search tree (BST) where words are sorted alphabetically. Consider that the binary tree contains objects of the `WordMean` class, and the dictionary is represented by the `Dictionary`.

```
class WordMean {
   string word;
   string meaning;
public:
   WordMean(string w, string m);
   //…
};
```

```
class Dictionary {
     set<WordMean> words;
public:
     void addWord(WordMean wm);
     void readFile(ifstream& f);
     string consult(string word1, WordMean&
previous, WordMean& next) const;
     bool update(string word1, string
mean1);
};
```

**1.1.** Implement the member function:

```
void Dictionary::readFile(ifstream &fich)
```

This function reads the words and their meaning from a file (*) and stores this information in the `words` binary search tree. Note that the binary search tree must store the elements sorted alphabetically by the word in the dictionary. The file consists of an even number of lines, where the first line contains the word and the next line its meaning:

```
gato
mamifero felino
morango
fruto
…
```

* **Note:** You should adjust the line separators according to the operating system. In CLion, you open the `dic.txt` file, and a button will appear at the bottom right with the different line separators. Choose according to the operating system:
    a. Unix: LF
    b. Windows: CRLF
    c. Mac: CR

**1.2.** Implement the member function:

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

This function prints the contents of the dictionary on the monitor, sorted alphabetically by word, in the following format:

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

**Note:** unit tests never fail

**Expected time complexity:** $O(n)$

**1.3.** Implement the member function:

```
string Dictionary::consult(string w1, WordMean& previous, WordMean& next)
const
```

This function returns the meaning of the word *w1*. If the word *w1* does not exist in the dictionary, the function returns the string "*word not found*" and must place in *next* and *previous* the WordMean objects existing in the dictionary relating to the words immediately before and immediately after (alphabetical order) the word *w1*, respectively. If there is no previous word and/or no subsequent word, the respective objects contain *word=""* and *meaning=""*.

**1.4.** Implement the member function:

```
bool Dictionary::update(string w1, string m1)
```

This function changes the meaning of the word *w1* to a new meaning *m1*. If the word *w1* exists in the dictionary, the method returns *true*, otherwise, this new word with the meaning *m1* is added to the dictionary and the method returns *false*.

**Expected time complexity:** $O(n)$

**2.** Consider the `FunSetProblem` class which has only static methods. Implement the function:

```
pair<int,int> FunSetProblem::pairSum(const vector<int> &values, int sum)
```

Given a set of non-repeating integer *values* and a *sum* value, find out if there is a pair of values (non necessarily of adjacent elements) whose sum is *sum*. If such a pair, or pairs exist, the function should return one of the pairs, otherwise, it should return (0,0).

**Tip:** use a binary search tree to store the elements of the vector. While traversing the vector, also search the tree to see if there is a "pair" of the current element in the vector.

**Expected time complexity:** $O(n \times \log n)$

**Execution example:**
input: *values* = {7, 8, 12, 5, 2, 3, 5, 6} ; *sum* = 14
output: *result* = <8,6> ou <6,8>