TrieRex

Koalab koala@epitech.eu

Abstract: In this subject, you will learn how to properly use data structures. As an example, we will ask you to implement and use a Trie.

Contents

Chapter I

Instructions

Test duration: 4 hours

I.1 Instructions

- Create an `auteur` file in your `rendu/` directory. It must contain your login, followed by a “\n”.

  Exemple :

  ```
  $> cat -e auteur
  login_x$
  $>
  ```

- Forgetting the auteur file, or having an error within, will cost you a null grade at your exam, no matter how good your work was.

- Any function implemented in a header file, and any unprotected header file, will cost you a null grade to the concerned exercise.

- Any text output is made on the standard output and is ended by a “\n”, unless anything else is explicitly specified.

- When an example output contains a newline, you mustn’t consider it if it follows an antislash ("\"). The same thing applies for a string contained within "", of course. For example, the string "For \
  pony !" should be considered as "For pony !"

- Files to turn in for each exercise must be added to the /rendu/exN directory, N being the exercise number. For example, /rendu/ex1 for exercise 1.

- Each exercise reuses the code from the previous one. You’ll have to copy over the required files.

- File names that are specified must be PRECISELY respected, as well as functions and classes names.

- Remember that you are now coding in C++, not in C. Thus, the following functions are FORBIDDEN, and using them will be rewarded by a -42 : *alloc, *printf, free
- You should FULLY read each exercise before beginning to work on it.
- All your classes must be in canonical form.
- Remember to protect your header files.
- You cannot use any of these for the whole exam:
  - std::map
  - std::unordered_map
  - std::multimap

Switches and forests of "if"s (and/or else if) are FORBIDDEN in the entirety of this examination. These cases will be verified during your exam's correction, and any cheat attempt will be punished with a -42, without any kind of mercy. Only branchings lower or equal to IF THEN (ELSE IF THEN ELSE) are allowed.

I.2 Naming convention

- Usually, files associated with a class will always be named NameOfTheClass.hpp and NameOfTheClass.cpp (if need be).
- Identifiers you'll have to use will always be formatted as follow:

```
thisIsSomeFunction();
int anIntegerNumber;
std::list<int> anIntegerList;
getSomeClassAttribute();
```

I.3 Compiling the exercises

- The moulinette will compile your code with the flags: `-W -Wall -Werror`.
- To avoid any compilation problem with the moulinette, include any required header file in your include files (`*.hpp`).
- Please carefully note that none of your files must contain a `main` function. We will use our own `main` function to compile and test your code.
- This test is composed of 4 exercises, worth 5 points each. Be really careful and focused, no mistake is allowed and the moulinette will stop at the first error.

I.4 Graduation

Number	Title	Points
0	Trie ?	5 points
1	Equiping the Trie-Rex	5 points
2	Going through the trie	5 points
3	The Harvesting	3 points
4	You should Trie !	2 points

Good luck.

Chapter II

Introduction

You are here to help Koala harvesters in their search for Koaleaves. Koaleaves are very rare, growing only on the highest Trie leaves. The first step in gathering those leaves is finding a way to climb up there.

Hopefully, Koalas have learnt how to ride the mighty Trie-Rex. But that is not enough; you will need to teach these creatures how to climb without ruining the Trie to avoid losing some precious Koaleaves.

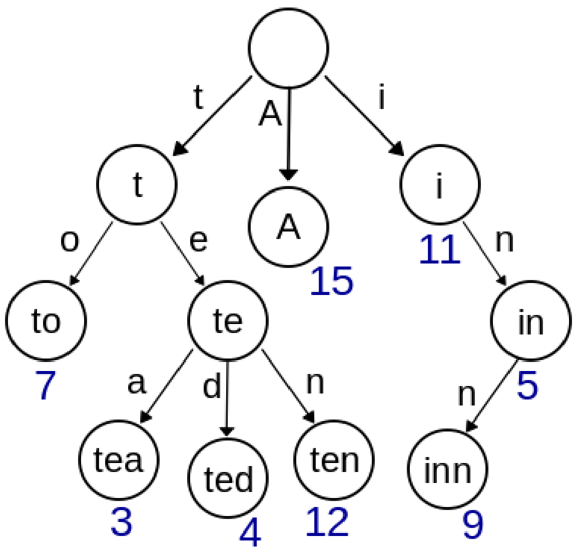
Chapter III

Exersice 0 : Trie ?

	Exercise : 0	points : 5
Learning how to recognize a Trie		
Turn-in directory: `/rendu/ex0`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `Trie.hpp, Trie.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

First, the Koalas will have to make sure their Trie-Rex are able to recognize Tries. It is your job to teach them how.

A Trie (or radix tree) is an ordered tree data structure. For this first assignment, we ask you to implement such a structure.



Create a TrieNode structure that will hold informations about one node of the Trie:

- A char **data** that will hold the character.
- An unsigned int **weight** which indicates the importance of a node.
- A TrieNode** **children** used to store the node's children.

Your structure must provide an operator overload to allow printing the content of a node.

```
std::cout << node << std::endl;
```

Should output "('character', weight)":

```
$> ('a', 5)
```

Now that we have the basic structure of a Trie, let's start to encapsulate it. Create a Trie class, that will be default construtible. Upon construction, it will initialize the root of the Trie and hold a pointer to it.
Remember, your class will have to be in cannonical form.

Chapter IV

Exercise 1 : Equiping the TrieRex

	Exercise : 1	points : 5
Equiping the TrieRex		
Turn-in directory: `/rendu/ex1`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `Trie.hpp, Trie.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

The Koalas now have a proper way to recognize a Trie. But the TrieRex doesn't have anything to correctly climb the Trie ! You'll have to teach them how to properly manipulate it.

Add the following functions to your Trie class:

- **Add a word to the Trie**:

```
void  addWord(std::string const& word);
```

  This function will add **word** characters one by one to the Trie. Starting from the root, check if a character already exists in the children. If it does, increment it's **weigth** by one, if it doesn't, add it as a child. Then, change the current node to this new child, and repeat until the end of **word**.

- **Remove a word from the Trie**

```
void  removeWord(std::string const& word);
```

  This function does exactly the opposite of **addWord**.

Once again, provide operator overloads to allow insertion and deletion in the Trie:

```
trie << "Koala";
```

Adds “Koala” to the Trie.

```
trie >> "Koala";
```

Removes “Koala” from the Trie.

If a word cannot be found in the Trie, you must throw an exception of your own making, leaving the Trie unchanged.

Chapter V

Exercise 2 : Going through the Trie

	Exercise : 2	points : 5
Going through the Trie		
Turn-in directory: `/rendu/ex2`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `Trie.hpp, Trie.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

Now that you can correctly manipulate a Trie, you will have to teach your Trie-Rex how to go through the Trie.

Implement the following function:

- **Get completion at point**

  ```
  std::vector<std::string> const& getCompletion(std::string const& input);
  ```

  This function returns a vector containing all the possible completions of **input**. If **input** is empty, the function returns the whole Trie.

We also ask that you overload the access operator:

```
std::vector<std::string> result = trie["koa"];
```

This call should work exactly as **getCompletion** does.

Chapter VI

Exercise 3: The Harvesting

	Exercise : 3	points : 3
The Harvesting		
Turn-in directory: `/rendu/ex3`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `Trie.hpp, Trie.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

You now have all the tools required for Koaleaves harvesting. Hence, your last assignment will be to actually determine which branch holds the rarest leaves.

Add a function **getLeaves** to your Trie class:

```
std::string const& getLeaves(uint8_t rarity);
```

This function returns all the nodes below **rarity percent** of the average weight of a node in the Trie, ordering them by weight.

Chapter VII

Exercise 4: You should Trie !

Exercise : 4		points : 2
You should Trie !		
Turn-in directory: `/rendu/ex4`		
Compiler: `g++`	Compilation flags: `-W -Wall -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in : `Trie.hpp, Trie.cpp`		
Remarks : `n/a`		
Forbidden functions : `None`		

Now that you have some basic functionalities, let's see how to properly use a Trie.

- Add a template parameter to your Trie class. This template will be used to be able to store a vector of T values. Then modify your **addWord** function so that it also takes a T reference.

```
void addWord(std::string const& word, T const& value);
```

- Modify your **operator**[] overload so that it returns a T reference if you can find the whole string in your trie. If you can't find the exact string but can still give possible corrections, you have to throw an exception of your own making, with a message following this syntax:

```
Could not find [input str].
Possible corrections:
koala
koalab
...
```

All exception thrown must inherit from std::exception. std::map or equivalents are forbidden for the whole exam.