Assignment 1. Testing, Lists, and Trees

Data Structures & Algorithms 2022-2023, BKI, UvA

Deadline for this assignment: Friday February 18th, 5pm

Readability: In these assignments, we require that your code satisfy the Python style standards, as measured by the `flake8` tool used in CodeGrade. One point of your grade is awarded for passing the style check with zero warnings, please ensure this. You may install `black` and use `black -l 80 myfile.py` to automatically reformat your code and get rid of most warnings, but please do check your files before submitting to ensure they are readable, and check the CodeGrade output for style warnings. (5 pts)

1 Assessing correctness

Climate stations record hourly temperatures and store them in a linked list data structure. Three different contractors were tasked with implementing an algorithm to find the warmest temperature recorded. The temperatures are stored in a linked list, using the implementation provided in **weather.py**.

The contractors have been asked to implement a function to retrieve the highest temperature, subject to the following specification:

```
def get_highest_temperature(lst: TemperatureList)
    -> Optional[float]:
```

Given a `TemperatureList`*, return the highest observed temperature stored in the list. If the list is empty, return None.*

We are using the **pytest** library, which is currently a very popular and widely used testing library. Before you continue, make sure **pytest** is installed in your python environment, for instance by `python -m pip install pytest`

- Design and write a suite of at least five test functions. Describe what each test looks for. With these tests you should be able to make sure that given any implementation for the `get_highest_temperature` function, it should only pass for a correct implementation. It's ok to have some redundancy, but the textual descriptions for the tests should not be the same. (5 pts)

- The contractors have sent you the following three implementations, called `get_highest_temperature_{v1,v2,v3}`. Execute **weather.py** to run your tests on each of them. What can we conclude from the output? Can we pinpoint any issues? Answer these questions in a code comment. *(Hint: there is at least one bug!)* (4 pts)

- Try to trick the test suite: write your own implementation `get_highest_temperature_v4` that is incorrect but still passes at least one of the tests. Make sure at least one test is passed so the implementation makes some sense. (2 pts)

2 Navigating Complex Books

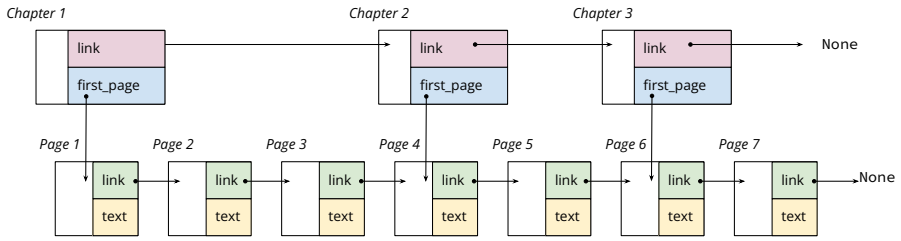
As part of a larger open-source project for building an efficient PDF document navigator and viewer, you have been tasked with building a data structure that is suitable for navigating through a book.

A book is, of course, a collection of pages. However, while some books are read page by page, many (e.g. textbooks) are too long for that, so we look things up by chapters instead. Therefore, we have a need for actions like:

- Get the first page of each chapter;
- Get all pages in the third chapter; etc.

We approach this problem using a **hierarchy of linked lists**. You will implement the data structure depicted below.

Figure 1: Book data structure: hierarchy of two linked lists, one for chapters, one for pages.



This structure contains two types of nodes:

1. `Chapter`, each containing a reference to the first page in the chapter, and a link to the next chapter node.
2. `Page`, each containing a string of text, and a link to the next page.

All pages are connected to each other even across chapters, so it's possible to just flip page by page and read the whole book. But chapters provide **skip-connections** or "fast lanes", allowing faster jumps through larger chunks of a document. Use the implementation provided in **booklist.py** as a starting point and don't forget to fill in your name and student number. Perform the following exercises:

1. create a `Book` class that stores a reference to the first chapter, and initializes as an empty book, with the first chapter being `None`. Then implement the `first_chapter` and `first_page` properties, returning the first chapter and the first page of the book, or `None` for an empty book. (In Python, properties are like methods that masquerade as attributes: you don't need parentheses to call them.) (2 pts)

2. implement `prepend_page` to add a new page at the beginning of the book (possibly creating a new chapter for it.) (2 pts)

3. implement `append_page` to add a new page at the end of the book (possibly creating a new chapter for it.) (4 pts)

4. implement `get_page` to retrieve a page by a given page number (one-indexed!). (3 pts)

5. implement `get_chapter_page` to retrieve a page from a given chapter (i.e., first flip to the chapter, then turn k pages; one-indexed.) (4 pts)

6. implement `length`: count the number of pages in the entire book. *(Hint: start at the first page and turn until reaching None.)* (1 pts)

7. implement `chapter_length`: count the number of pages in a given to retrieve a page from a given chapter. *(Hint: start at the first page in the chapter, and turn until reaching the first page in the following chapter, or the end of the book.)* (3 pts)

8. Add a `print_book_backward` function that displays the text of the **first page of each chapter**, starting with the last chapter and ending at the first. Here, you may choose one of the following approaches:

   (a) update the structure to also keep backward links

   (b) use an auxiliary stack storage

   (c) any other implementation

   Try matching the format of the original `print_book` function (You won't receive more points if you take more than one approach. For grading, you have to first say which approach you used.) (5 pts)

3 Learning to guess

We will build a computer program to play a guessing game. The player thinks of an animal and the computer must figure out what it is, by asking a series of yes/no questions. Here is an example interaction.

```
Welcome to the Animals game!

Is it a mammal?
>  yes
Is it bigger than a chair?
>  yes
Is it furry?
>  no
Is it a Dolphin?
>  yes
** I win! **
```

To have any shot at winning, the computer needs a **knowledge base**. In the first part of the assignment, we will provide a small knowledge base In the second part, you will enable the computer to **learn** from the player.

3.1 Using the knowledge base.

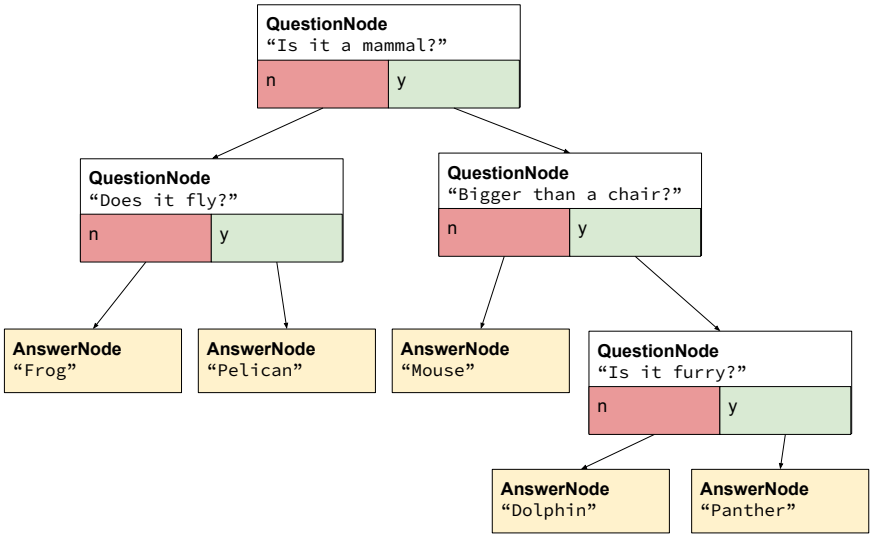
Our first computer player will have the following small knowledge base with only 4 questions and 5 possible answers.

```
[Question: Is it a mammal?]
 - no: [Question: Does it fly?]
   - no:  [Answer: Frog]
   - yes: [Answer: Pelican]
 - yes: [Question: Bigger than a chair?]
   - no:  [Answer: Mouse]
   - yes: [Question: Is it furry?]
     - no:   [Answer: Dolphin]
     - yes:  [Answer: Panther]
```

This knowledge base can be represented as a binary tree where the intermediate nodes are questions and the leaf nodes are animals. Implement this knowledge base in the provided python file **animal_game.py**. Note the implementation is very strict: Use attribute and function names exactly as given below to pass all Codegrade tests!

- Implement a `AnswerNode` class that has one attribute `answer`. Also implement a `QuestionNode` class that has three attributes: `question`, `if_yes`, and `if_no`. The `answer` and `question` attributes will hold a string. The other attributes will either contain another `QuestionNode` (a follow-up

Figure 2: Animal game knowledge base represented as a binary tree.



question) or an `AnswerNode` containing the answer. Also take care of empty nodes with attributes initialized as `None`. (1 pts)

- Implement a `KnowledgeBase` class containing one attribute `root` pointing to the root `QuestionNode` of the tree. (1 pts)
- Implement the function `def initialize_kb() -> None` to build the small initial knowledge base from above and save it to the `kb.pkl` file. (1 pts) (1 pts)
- Implement a function `def play(kb: KnowledgeBase) -> None` that plays one round of the game with the user. (3 pts)

  (Hint: Use the Python `isinstance` builtin function to decide whether the “current” node is a `QuestionNode` or a `AnswerNode`. You can also use pattern matching (python >= 3.11).)
- Win against the computer.

3.2 Growing the knowledge base.

When the computer loses, it is a sign that its knowledge base is lacking. For instance, there are other large furry mammals other than the panther. In such a situation, your program can **learn from the user**.

- When the program guesses incorrectly, prompt the user for the name of the animal they intended, and for a yes/no question that differentiates the two animals. Example interaction:

  ```
  (...)
  My guess is: Panther. Did I guess correctly?
  > no
  ** I lose :( **
  What was the animal you thought about?
  >  Tiger
  Enter a question with answer yes for Tiger, but no for Panther.
  > Does it have stripes?
  Thank you.
  Play again? >
  ```

  Update the knowledge base with the user-provided question and save it into `kb.pkl` using the `pickle` library. (3 pts)

- Add a method `KnowledgeBase.size() -> int` that returns the number of known animals in a knowledge base. (2 pts)

- Add a method `KnowledgeBase.depth() -> int` that returns the maximum number of questions (`QuestionNodes`) needed (in the worst case) to get to an answer. (Answers such as "is it a mouse?" don't count as questions.) (2 pts)

- Include the statistics about the size and maximum number of questions needed in the introduction of the game, i.e.

  ```
  Welcome to the Animals game!

  The current number of known animals by the knowledge base is 5.
  I can guess your animal in at most 4 questions.
  ```