Homework 4

CS 173: Intermediate Computer Science — Spring 2014
Instructor: Thomas Bressoud — Due: 2013-02-25

- The point of this project is to practice linked lists by implementing your own doubly-linked list class.
- The earlier guidelines for documenting and commenting your code hold. You should also include unit tests for each class in a separate file.
- In addition to unit testing, you will change your CounterSim discrete-event simulator to use your new DLList class instead of the Python list in the implementation of your OrderedSet and we will also add randomness to our simulator.

Part 1: Design and implement a List ADT

In this project, you will write a doubly linked list implementation of a List ADT. Use the singly linked implementation in your book (and from class) as a starting point. But keep in mind that many of the methods will need to be changed to account for the double links. You should also think about ways in which the implementations of particular methods might be made more efficient with the addition of reverse links. Part of the grading will assess the efficiency of your implementation. Your class should contain at least the following methods:

- `__init__(self)`: constructs a `DLList` object
- `__len__(self)`: returns the length of the maintained list
- `__getitem__(self, index)`: allows access to list items via indexing
- `__setitem__(self, index, value)`: allows changing list items via indexing and assignment
- `insert(self, index, value)`: insert `value` at position `index`
- `__delitem__(self, position)`: delete the item in position index (via the del operator)
- `pop(self, index=-1)`: remove and return the item at position given by `index` with a default of the last item in the collection
- `index(self, value)`: return the position of the item with value value
- `remove(self, value)`: remove the item with value `value`
- `__str__(self)`: return a string representation of the list and should be identical to the string representation of a Python list with the same elements
- `__iter__(self)`: return an iterator for the list

If you include any private functions, their names should begin with a single underscore. Develop unit tests for each method **as you write it**. Test each method thoroughly before moving on to the next one. Also, remember docstrings and comments, of course. Files should be named `DLList.py` and `test_DLList.py` and the class name should be `DLList`.

Part 2: Integrate your `DLList` class into your OrderedSet/CounterEvent/CounterSim simulation

This should involve almost no code changes, and is simply giving us another means of driving the DLList class through its paces (beyond the unit test). The DLList will replace the Python list used as an instance variable in your OrderedSet. If the abstraction boundary was maintained in your original implementation, no changes should be required of the CounterEvent or CounterSim to accomplish this integration. All three of these classes and their unit tests should be resubmitted with this homework.

Part 3: From Deterministic to Stochastic Simulation

In Part 3, we will create two new classes: `ParticleSim` and `ParticleEvent`. Both of these classes share much in common with their counterparts from HW3, `CounterSim` and `CounterEvent`. In this description, I will focus on what is *different* about the two simulations.

The idea of a particle simulation is to model the random lifespan of radioactive particles. Say that we know the average lifespan of a radioactive particle is 4.0 time units. But each particle does not last the same amount of time, but their lifespan can be modeled by an exponential distribution that gives lifespan values about that mean – so some particles are less than 4 and some are more than 4, but the spread about 4 is not "uniform". We will talk in class more about distributions, but for now, we simply need to know that Python allows us to generate a random value from a particular distribution by using the `random` module, and knowing certain parameters that govern the specific shape of the distribution. The exponential is a nice distribution because, by just knowing the mean, you can determine the λ parameter and invoke Python's `expovariate` method to get a particular random value about that mean. I leave it to you to look up the Python documentation and figure out how to appropriately call the `expovariate` function from the `random` module.

ParticleEvent ADT

The `ParticleEvent` is even simpler than the `CounterEvent` from the last homework. In particular, the `execute()` method does not (re)schedule another event, but simply prints (and returns) a string of the form: `'Particle disintegrated at time t = '` concatenated with the string version of the time of the event.

The `__str__`, `time()`, and relational operator overloaded methods are all the same as that of `CounterEvent`.

ParticleSim ADT

The `now()`, `insert()`, and `doAllEvents()` methods of the `ParticleSim` class are identical to their counterparts in `CounterSim`. The method that needs to change in `ParticleSim` is the `setup` method.
The function header for `setup` looks as follows:

```
def setup(n = 5, mean = 4.0, seed = None):
```

The parameters are interpreted as follows:

- `n`: Number of `ParticleEvents` for setup to create at simulation startup, with a default of 5 if not specified.
- `mean`: Mean of the lifetime for each created `ParticleEvents`. The actual time to be passed at creation will be generated from the `random` module, and be based on this mean. Default mean lifetime is 4.0.
- `seed`: If not `None`, this parameter is used before any random values are generated by the random module. This is an integer to be passed to the `seed` function of the random module and is used to determine the starting point for pseudo-random-number generation. By specifying a specific seed, the random numbers, if generated in the same order, will give the same results, to allow for debugging and testing.

Testing

Assuming your homework 3 CounterEvent was working properly, the simplification in arriving at the above `ParticleEvent` should not result in any new problems, so unless you have difficulty, you are not required to submit a `test_ParticleEvent.py` unit test for that module.
Your `test_ParticleSim.py` should create and run a variety of particle simulations, varying the arguments to the simulator's `setup` method. The result will show the output from the prints of the underlying `ParticleEvent execute()` method, but will not be employing assertions of value correctness in the unit test methods.