Lista de Linguagens de Programação – 16

Nome: _____ Matrícula: _____

Os exercícios desta lista devem ser todos implementados em Python.

1. Este exercício faz referência às classes implementadas em Python disponíveis no arquivo `List.py` na página do curso.

   (a) Adicione um método `contains` à classe `List`, de modo tal que `x.contains(n)` retorne `true` se o valor inteiro `n` ocorrer na lista `x`, do tipo `List`. O método deverá retornar `false` caso contrário.

   (b) Adicione um método `equals` à classe `List`, de modo tal que `x.equals(y)` retorne `true` se `x` e `y` possuirem exatamente os mesmos inteiros, na mesma ordem. O método retornará falso caso contrário. A invariante que `x.equals(y)` e `y.equals(x)` são iguais deve ser mantida. Deve ser verdade também que que, caso `x == y`, entao `x.equals(y)` deve retornar `true`. O contrário não é sempre verdade.

   (c) Adicione um método `append` à classe `List`, de modo tal que `x.append(y)` retorne uma instância de `List` que seja igual a lista `x` seguida da lista `y`. Não podem haver efeitos colaterais nem em `x`, tampouco em `y`. Dica: faça uma cópia de `x`.

(d) Adicione um método `reverse` à classe `List`, de modo tal que `x.reverse()` retorne um objeto do tipo `List` que seja o reverso da lista original `x`. Não deve haver nenhum efeito colateral em `x`.

(e) Adicione um método `reverseMe` à classe `List`, de modo tal que `x.reverseMe()` não retorne nenhum valor, mas tenha o efeito colateral de inverter o conteúdo da lista. Implementar esta função, na verdade, é mais fácil se fizermos uma cópia de `x`. Claro, é muito mais interessante, e também mais difícil, fazer a inversão *in place*, sem criar novas instâncias da classe `ConsCell`.

(f) Adicione um método `sort` à classe `List`, de modo tal que `x.sort()` retorne um objeto `List` que seja uma versão da lista `x`, ordenada em ordem não decrescente. Você pode usar qualquer algoritmo de ordenação que você quiser. Não deve haver nenhum efeito colateral em `x`. Passe a operação de comparação como um parâmetro do método `sort`.

(g) Adicione um método `sortMe` à classe `List`, de modo tal que `x.sortMe()` não retorne qualquer valor, mas tenha o efeito colateral de ordenar o conteúdo de `x` em ordem crescente. Use o algoritmo de ordenação que quiser, mas não crie nenhum novo objeto do tipo `ConsCell`.

(h) Implemente o método `append` do exercício anterior, de modo recursivo, usando um estilo de programação mais funcional.

2. Para resolver este exercício você precisará da definição das classes `Node` e `Stack` disponíveis na página do curso (arquivo `Worklist.py`). Neste exercício você deverá escrever duas outras classes, conforme descrito a seguir:

   (a) Escreva uma classe `Queue` que implemente uma lista usando um esquema de colocação e remoção de dados do tipo *first-in/first-out*. A sua classe deve implemenar os mesmos métodos que a classe `Stack` abaixo:

```
class Stack:
  "Describes a simple list data type."

  def add(self, element):
    """Adds a new element into this list."""

  def remove(self):
    """Removes the next element from this list."""

  def hasMore(self):
    """Returns True if this list is not empty."""
```

   A classe deve ser implementada com uma lista encadeada. Não vale usar alguma classe já existente na biblioteca padrão de Python que implemente a lista para você.

(b) Escreva uma classe `PriorityQueue` que implementa os mesmos métodos que `Stack`. Independente da ordem em que *strings* são inseridos na fila, o método `remove` deve sempre retornar a menor *string* em termos lexicográficos. Para comparar duas *strings* lexicograficamente, use os operadores relacionais, isto é, $<, \leq, \geq, >, \neq, =$. Você não precisa produzir uma implementação muito eficiente. Você pode, por exemplo, usar uma implementação baseada em uma lista encadeada. Você pode modificar a classe `Node`, mas não de um jeito que quebre a classe `Stack` para a qual `Node` foi originalmente feita. Você vai também precisar escrever algum código para testar a sua classe, por exemplo:

```
>>> s = Stack()
>>> s.hasMore()
False
>>> s.add("a")
>>> s.add("c")
>>> s.add("b")
>>> while (s.hasMore()):
...   print s.remove()
...
a
b
c
```

(c) Que modificações seriam necessárias para que a classe `PriorityQueue` pudesse trabalhar com números inteiros, em vez de *strings*?

(d) Considere o método `removeAll` logo abaixo:

```
def removeAll(s):
  """Removes all the elements from the data structure."""
  while (s.hasMore()):
    print s.remove()
```

i. Qual o "contrato" que deve ser garantido pelos parâmetros reais deste método, Isto é, pelos elementos `s` passados para o método?

ii. Pesquise o significado da expressão *duck typing*. O que *duck typing* tem a ver com o nosso método `removeAll`?

3. Escreva uma classe `Int` com os seguintes componentes:

   (a) Um campo para armazenar um número inteiro.

   (b) Um construtor, de forma tal que `Int(x)` crie uma nova instância do tipo `Int`, com o valor do inteiro `x`.

   (c) Um método `plus`, de modo tal que `x.plus(y)` retorne um novo objeto do tipo `Int`.

   (d) Um método `toString`, de forma tal que `x.toString()` retorna uma *string* representando o objeto `x`.

   (e) Métodos `minus`, `times` e `div`, parecidos com o método `plus` do item anterior. O método `div` deve realizar divisão inteira, igual o operador barra (`/`) aplicado sobre valores inteiros.

   (f) Um método `isPrime`, de tal forma que `x.isPrime()` retorne `true` se o valor de `x` for um número primo.

   Em algumas linguagens orientadas por objetos, como *Smalltalk*, tudo são objetos – inteiros, booleans, caracteres, operadores aritméticos, índices de arranjos, tudo, tudo. Nestas linguagens, a sintaxe de expressões aritméticas realmente se parece com esta classe que você acabou de criar.