Exercises

Set 12

DM562 Scientific Programming
DM857 Introduction to Programming
DS830 Introduction to Programming

1 Implementing Recursive Data Structures

1. A *(singly) linked list* is a list implementation where each element is stored together with a pointer to the following one (if any). Develop a class `LinkedList` implementing linked lists. Use an inner class Your implementation should provide the following methods.

   (a) A constructor that takes no arguments for creating empty lists.

   (b) A method `__repr__(self)` that returns a textual representation of this list.

   (c) A method `add(self,value)` for adding `value` at the beginning of this list.

   (d) A method `is_empty(self) -> bool` to check whether this list is empty or not.

   (e) A method `clear(self)` that removes all elements stored in this list.

   (f) A method `copy(self)` that creates a copy of this list. (Hint: leave this method for after you implemented `__reversed__`.)

   Your implementation should provide the methods expected by python from container types[1] where applicable, in particular, the following.

   (g) Method `__len__(self)` that returns the length of this list. This method is called to by the built-in function `len()`.

   (h) Method `__iter__(self)` that returns an iterator[2] object for traversing the list. This method is called to by the built-in function `iter()`. (Hint: write a generator iterator using `yield`. You can start by writing a method that calls `print()` on each element of the list and then rewrite it replacing `print` with `yield`.)

   (i) Method `__contains__(self,item)` that checks if `item` is in the list. This method is called to implement membership test operators e.g., `item in self` (default implementation uses linear search and `__iter__()`).

   (j) Method `__reversed__(self)` that returns a new LinkedList with the same element of this one but in the opposite order. This method is called to by the built-in function `reversed()`.

   (k) Methods `__getitem__(self,key)`, `__setitem__(self,key)`, `__delitem__(self,key)` for retrieving, setting, and deleting an element of this list, respectively. For simplicity, you can ignore sequences and support only the case where `key` is an integer. These methods are invoked to implement evaluation of `self[key]` in expressions, assignments and `del self[key]`.

[1] https://docs.python.org/3/reference/datamodel.html#emulating-container-types
[2] https://docs.python.org/3/glossary.html#term-iterator

2. A *stack* is data structure that maintains a linear collection of elements that must be removed in the opposite order in which they were added (like a stack of books). Develop a class `Stack` implementing stacks. Your implementation should provide the methods expected from a container types where applicable and the following ones.

   (a) A method `add(self,item)` that adds `item` on top of the stack.

   (b) A method `peek(self)` that returns the top elements of the stack.

   (c) A method `pop(self)` that removes the top element from the stack and returns it.

3. A *queue* is data structure that maintains a linear collection of elements that must be removed in the same order in which they were added. Develop a class `Queue` implementing queues. Your implementation should provide the methods expected from a container types where applicable and the following.

   (a) A method `enqueue(self,item)` that adds `item` on at the end of this queue.

   (b) A method `dequeue(self)` that returns the next element in this queue and removes it.

4. A *doubly linked list* is a list implementation where elements are stored in nodes together with pointers to both the previous and next item, and the list itself has pointer to the first and last element of the list. Develop a class `DoublyLinkedList` implementing doubli linked lists. Your implementation should provide the methods expected from a container types where applicable and the following.

   (a) `insert(self,index,value)` that inserts `value` in position `index` shifting all subsequent elements.

5. A *binary tree* is a data structure for collections where elements are stored in nodes each with reference to (at most) two other nodes typically called *left* and *right child*. There are three ways to iterate through the elements of a bianary tree:

   - *pre-order traversal* where we process the element in the node, then left subtree, and finally the right subtree.
   - *in-order traversal* where we process the left subtree, then element in the node, and finally the right subtree.
   - *post-order traversal* where we process the left subtree, then the right subtree, and finally the element in the node.

   Develop a class `BinaryTree` implementing binary trees. Your implementation should provide the methods expected from a container types where applicable and the following ones.

   (a) A method `root(self) -> Any` that returns the element stored in the root node.

   (b) Methods `left(self) -> BinaryTree` and `right(self) -> BinaryTree` that return the subtrees with the left and right child of the current root as their root, respectively.

   (c) A method `height(self) -> int` that returns the longest path from the root of this tree to a leaf node.

   (d) A method `mirror(self) -> BinaryTree` that returns a mirror copy of this tree (every left child is a right child in the mirror copy).

   (e) Methods `preorder(self)`, `inorder(self)`, `postorder(self) -> Iterator` for iterating over the elements stored in this tree following a pre-order, in-order, and post=order traversal, respectively. You can write three generators (using `yield`) or iterators (as inner classes that implement `__next__`).