RECURSIVE OBJECTS 9

COMPUTER SCIENCE 61A

July 21, 2016

1 Linked Lists in OOP

1.1 A New Implementation

Linked lists are data abstractions that can have multiple implementations. Previously, we saw linked lists implemented using Python lists. Today, we will look at linked lists implemented using Object-Oriented Programming. Here it is:

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
    def __getitem__(self, i):
        if i == 0:
            return self.first
        return self.rest[i-1]
    def __len__(self):
        return 1 + len(self.rest)
```

When we implemented linked lists using Python lists, we called `first(lnk)` and `rest(lnk)` to access the `first` and `rest` elements. This time, we can write `lnk.first` and `lnk.rest` instead. In the former, we could access the elements, but we could not modify them. In the latter, we can access and also modify the elements. In other words, linked lists implemented using OOP is mutable.

In addition to the constructor `__init__`, we have the special Python methods `__getitem__` and `__len__`. Note that any method that begins and ends with two underscores is a special Python method. Special Python methods may be invoked using built-in functions and special notation. The built-in Python element selection operator, as in `lst[i]`, invokes `lst.__getitem__(i)`. Likewise, the built-in Python function `len`, as in `len(lst)`, invokes `lst.__len__()`.

1.2 Questions

1. Write `sum_nums`, which takes in a linked list `lnk` and sums up all elements in `lnk`. You may assume all elements in `lnk` are integers. `sum_nums` should return the sum, an integer.

```
def sum_nums(lnk):
    """
    >>> a = Link(1, Link(6, Link(7)))
    >>> sum_nums(a)
    14
    """
```

2. Write `multiply_lnks`, which takes in a Python list of `Link` objects and multiplies them element-wise. It should return a new linked list. If not all of the `Link` objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the `Link` objects are shallow linked lists, and that `lst_of_lnks` contains at least one linked list.

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest
    ()
    """
```

1.3 Extra Questions

1. Define `reverse`, which takes in a linked list and reverses the order of the links. The function may *not* return a new list; it must mutate the original list. Return a pointer to the head of the reversed list.

```
def reverse(lnk):
    """
    >>> a = Link(1, Link(2, Link(3)))
    >>> r = reverse(a)
    >>> r.first
    3
    >>> r.rest.first
    2
    """
```

2. Write a function `remove_duplicates` that takes as input a sorted LinkedList of integers, `lnk`, and mutates `lnk` so that all duplicates are removed.

```
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> unique = remove_duplicates(lnk)
    >>> len(unique)
    2
    """
```

2 Trees in OOP

2.1 Another New Implementation

Trees are also data abstractions that can have multiple implementations. Previously, we implemented the tree abstraction using Python lists. Let's look at another implementation using objects instead. With this implementation, we can easily specify specialized tree types, such as binary trees , using inheritance.

```
class Tree:
    def __init__(self, entry, children=[]):
        for c in children:
            assert isinstance(c, Tree)
        self.entry = entry
        self.children = children

    def is_leaf(self):
        return not self.children
```

Notice that with this implementation we can mutate the entry of a tree by reassigning `tree.entry`. In the previous implementation using lists, this was not possible, because the abstraction barrier prevented us from seeing how the tree was implemented.

2.2 Questions

1. Consider the following definitions and assignments and determine what Python would output for each of the calls below *if they were evaluated in order*.

```
>>> t0 = Tree(0)
>>> t0.entry


>>> t0.children


>>> t1 = Tree(0, [1, 2])#Is this a valid tree?


>>> t2 = Tree(0, [Tree(1), Tree(2, [Tree(3)])])
>>> t2.children[0]


>>> t2.children[1].children[0].entry
```

2. Define a function make_even which takes in a tree t whose entries are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers remain the same.

```
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.entry
    2
    >>> t.children[0].children[0].entry
    4
    """
```

2.3 Extra Question

1. Write a function that combines the entries of two trees `t1` and `t2` together with the `combiner` function. Assume that `t1` and `t2` have identical structure. This function should return a new tree.

```
def combine_tree(t1, t2, combiner):
    """
    >>> a = Tree(1, [Tree(2, [Tree(3)])])
    >>> b = Tree(4, [Tree(5, [Tree(6)])])
    >>> combined = combine_tree(a, b, mul)
    >>> combined.entry
    4
    >>> combined.children[0].entry
    10
    """
```

3 Binary Search Trees

3.1 Intro to Binary Search Trees

A Binary Search Tree (BST) is a special kind of tree that satisfies the following properties:

- Every node of a BST has at most two children called `left` and `right`. The children are also BST's.
- For every node, the left child's entry is less than or equal to the parent's entry.
- For every node, the right child's entry is greater than the parent's entry.

```
# Binary Search Tree (BST) Class
class BST:
    empty = ()
    def __init__(self, entry, left=empty, right=empty):
        assert left is BST.empty or isinstance(left, BST)
        assert right is BST.empty or isinstance(right, BST)

        self.entry = entry
        self.left = left
        self.right = right

        if left is not BST.empty:
            assert left.max <= entry
        if right is not BST.empty:
            assert entry < right.min

    @property
    def max(self):
        if self.right is BST.empty:
            return self.entry
        return self.right.max

    @property
    def min(self):
        if self.left is BST.empty:
            return self.entry
        return self.right.min
```

CS 61A Summer 2016: Brian Hou and Marvin Zhang, with
Daniel Haas, Eric Bo, Jack Thakar, Jerome Baek, Katya Stukalova, Marsalis Gibson, Mitas Ray, Neil Agarwal, Samantha Wong, Tammy Nguyen, Vivian Fang, and Zhen Qin

3.2 Questions

1. Define a function `insert` that takes in a `BST`, `bst`, and a number, `n`, and mutates `bst` by inserting a new node. `insert` should place the new node as a leaf in the correct position. If `t` is the `BST` on the left, then calling `insert(t, 3)` will change `t` to the `BST` on the right. Do not return a new `BST` unless `bst` is empty, in which case return a `BST` containing only `n`.

```
          4                                   4
        /   \                               /   \
       2     5           ->                2     5
     /                                   /   \
    1                                   1     3
```

```
def insert(bst, n):
    """
    >>> bst = BST(4, BST(2, BST(1)), BST(5))
    >>> insert(bst, 3)
    >>> bst.left.right.entry
    3
    """
```