LINKED LISTS AND TREES 10

COMPUTER SCIENCE 88

April 14, 2021

1 Linked Lists

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__()`.

```
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 __repr__(self):
        if self.rest:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(repr(self.first), rest_str)

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

2 Questions

1. Write a function that takes in a a linked list and returns the sum of all its elements. You may assume all elements in `lnk` are integers.

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

Solution:

```
    if lnk == Link.empty:
        return 0
    return lnk.first + sum_nums(lnk.rest)
```

2. Write a iterative function `is_palindrome` that takes a LinkedList, `lnk`, and returns `True` if `lnk` is a palindrome and `False` otherwise. Hint: Use the `reverse` function from lab09.

```
def is_palindrome(lnk):
    """
    >>> one_link = Link(1)
    >>> is_palindrome(one_link)
    True
    >>> lnk = Link(1, Link(2, Link(3, Link(2, Link(1)))))
    >>> is_palindrome(lnk)
    True
    >>> is_palindrome(Link(1, Link(2, Link(3, Link(1)))))
    False
    """
```

Solution:

```
    reversed = reverse(lnk)
    while lnk is not Link.empty and reversed.first == lnk.
       first:
        reversed = reversed.rest
        lnk = lnk.rest
    return lnk is Link.empty
```

3. Write a function that takes a sorted linked list of integers and mutates it so that all duplicates are removed.

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

Solution: Recursive solution:

```
    if lnk is Link.empty or lnk.rest is Link.empty:
        return
    if lnk.first == lnk.rest.first:
        lnk.rest = lnk.rest.rest
        remove_duplicates(lnk)
    else:
        remove_duplicates(lnk.rest)
```

For a list of one or no items, there are no duplicates to remove.

Now consider two possible cases:

- If there is a duplicate of the first item, we will find that the first and second items in the list will have the same values (that is, `lnk.first == lnk.rest.first`). We can confidently state this because we were told that the input linked list is in sorted order, so duplicates are adjacent to each other. We'll remove the second item from the list.

  Finally, it's tempting to recurse on the remainder of the list (`lnk.rest`), but remember that there could still be more duplicates of the first item in the rest of the list! So we have to recurse on `lnk` instead. Remember that we have removed an item from the list, so the list is one element smaller than before. Normally, recursing on the same list wouldn't be a valid subproblem.

- Otherwise, there is no duplicate of the first item. We can safely recurse on the remainder of the list.

Iterative solution:

```
    while lnk is not Link.empty and lnk.rest is not Link.
       empty:
        if lnk.first == lnk.rest.first:
            lnk.rest = lnk.rest.rest
        else:
            lnk = lnk.rest
```

The loop condition guarantees that we have at least one item left to consider with `lnk`.

For each item in the linked list, we pause and remove all adjacent items that have the same value. Once we see that `lnk.first != lnk.rest.first`, we can safely advance to the next item. Once again, this takes advantage of the property that our input linked list is sorted.

3 Trees

In computer science, **trees** are recursive data structures that are widely used in various settings.

Contrary to our ideas of a tree, in computer science, a tree branches downward. The **root** of a tree starts at the top, and the **leaves** are at the bottom.

A tree is considered a recursive data structure because every branch from a node is also a tree.

Some terminology regarding trees:

- **Parent node**: A node that has branches. Parent nodes can have multiple branches.
- **Child node**: A node that has a parent. A child node can only belong to one parent.
- **Root**: The top node of the tree. In our example, the node that contains 7 is the root.
- **Label**: The value at a node. In our example, all of the integers are values.
- **Leaf**: A node that has no branches. In our example, the nodes that contain -4, 0, 6, 17, and 20 are leaves.
- **Branch**: A subtree of the root. Note that trees have branches, which are trees themselves: this is why trees are *recursive* data structures.
- **Depth**: How far away a node is from the root. In other words, the number of edges between the root of the tree to the node. In the diagram, the node containing 19 has depth 1; the node containing 3 has depth 2. Since there are no edges between the root of the tree and itself, the depth of the root is 0.
- **Height**: The depth of the lowest leaf. In the diagram, the nodes containing -4, 0, 6, and 17 are all the "lowest leaves," and they have depth 4. Thus, the entire tree has height 4.

In computer science, there are many different types of trees. Some vary in the number of branches each node has; others vary in the structure of the tree. Recall the tree abstract data type: a tree is defined as having a label and some branches. Trees can be implemented as an ADT, but we will only be focusing on the Tree class this semester. Below is the most basic implementation of a Tree class that we will be using.

```
class Tree:
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches

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

Notice that with this implementation we can mutate a tree using attribute assignment, which wouldn't be possible in the implementation using lists in an ADT.

```
>>> t = Tree(3, [Tree(4), Tree(5)])
>>> t.label = 5
>>> t.label
5
```

3.1 Questions

1. What would Python display? If you believe an expression evaluates to a *Tree* object, write *Tree*.

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

Solution: `0`

```
>>> t0.branches
```

Solution: `[]`

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

Solution: `AssertionError #As the branches must be Tree objects`

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

Solution: `Tree(1)`

```
>>> t2.branches[1].branches[0].label
```

Solution: `3`

2. Define a function `make_even` which takes in a tree `t` whose values 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.label
    2
    >>> t.branches[0].branches[0].label
    4
    """
```

Solution:

```
    if t.label % 2 != 0:
        t.label += 1
    for branch in t.branches:
        make_even(branch)
```

3. Write a function that combines the values 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.root
    4
    >>> combined.branches[0].root
    10
    """
```

Solution:

```
    combined = [combine_tree(b1, b2, combiner) for b1, b2
                in zip(t1.branches, t2.branches)]
    return Tree(combiner(t1.root, t2.root), combined)
```