LEARNING AI-DS-ML SKILLS

PYTHON PROGRAMMING

CHAPTER 6
LINKED LISTS + TREES

BY FRED ANNEXSTEIN

PYTHON PROGRAMMING

6.
LINKED LISTS AND TREES

BY FRED ANNEXSTEIN

CHAPTER 6 OVERVIEW

In this module we will be covering the abstract data types of linked lists and trees. We will examine the creation and processing of these objects using functional programming style. We will apply out tree implementations to the solution of several programming problems.

CHAPTER 6 OBJECTIVES

By the end of this module, you will be able to...

- Understand the notion of an abstract data type using the example abstractions of linked lists and trees.
- Understand a functional approach to the implementation of creating and processing linked lists and trees.
- Understand how to use recursion to effectively process linked lists and trees.
- Understand how to use linked lists and trees as a data structure underlying an immutable sequence.

LINKED LISTS

Python has many built-in types of sequences: lists, ranges, and strings, to name a few. We construct our own type of sequence called a linked list. A linked list is a simple type of sequence that is composed of multiple links that are connected.

The key idea for the link abstraction is that each link is a pair where the first element is an item-value in the linked list, and the second element is another link. Links can be also be empty, that is a special value marking the end of the linked list.

CONSTRUCTORS

We design a function to construct a linked link as follows.

```
empty = 'empty'

def link(first, rest=empty):
    return [first, rest]
```

This function constructs a linked list with a first element and next element, which is also a link list or defaults to the empty value for the empty linked list.

SELECTORS

We design two function to select elements from a linked link as follows.

```
def first(s):
    return s[0]

def rest(s):
    return s[1]
```

A linked list is a pair containing the first element of the sequence (in this case 1) and the rest of the sequence (in this case a representation of 2, 3, 4). The second element is also a linked list. The rest of the inner-most linked list containing only 4 is 'empty', a value that represents an empty linked list. Linked lists have recursive structure: the rest of a linked list is a linked list or 'empty'. We can define an abstract data representation to validate, construct, and select the components of linked lists.

```
>>> alpha = link(‘A’, link(‘B’,
link(‘C’, link(‘D’))))

>>> first(alpha)
'A'

>>> rest(alpha)
['B', ['C', ['D', 'empty']]]

>>> first(rest(alpha))
'B'
```

LINKED LIST SEQUENCE INTERFACE

The linked list can store a sequence of values in order, but we have not yet shown that it satisfies the sequence abstraction. Using the abstract data representation we have defined, we can implement the two behaviors that characterize a sequence: length and element selection.

```
def len_link(s):
    length = 0
    while s != empty:
       s, length = rest(s), length 1
    return length
```

```
def getitem_link(s, i):
    while i > 0:
        s, i = rest(s), i - 1
    return first(s)
```

```
>>> len_link(alpha)
4

>>> getitem_link(alpha, 3)
'D'
```

Now, we can manipulate a linked list just as we can with any sequence using these interface functions.

RECURSIVE IMPLEMENTATION

Both len_link and getitem_link are iterative. They move through each layer of nested pair until the end of the list (in len_link) or the desired element (in getitem_link) is reached. We can also implement length and element selection using recursion.

```
def len_recursive(s):
    if s == empty: return 0
    return 1 + len_recursive(rest(s))

def getitem_recursive(s, i):
    if i == 0: return first(s)
    return getitem_recursive(rest(s),
                             i - 1)
>>> len_recursive(alpha)
4
>>> getitem_recursive(alpha,3)
‘D'
```

SUMMING ELEMENTS

Let us write a function that takes in a linked list `lst` and a function `fn` which is applied to each number in `lst` and returns the sum.

```
# Recursive Solution
def sum_linked_list_r(lst, fn):
    if lst == empty:
        return 0
    return fn(first(lst)) +
sum_linked_list(rest(lst), fn)

# Iterative Solution
def sum_linked_list_i(lst, fn):
    sum = 0
    while lst != empty:
        sum += fn(first(lst))
        lst = rest(lst)
    return sum

>>> sum_linked_list_r(alpha, ord)
266

>>> sum_linked_list_r(alpha,
            lambda x: ord(x)-ord('A')
6
```

In the last example we use Python ord() function which returns the Unicode code from a given character. This
function accepts a string of unit length as an argument and returns
the Unicode equivalence of the passed argument. In other words, given a string of length 1, the ord() function returns an

integer representing the Unicode code point of the character when an argument is a Unicode object, or the value of the byte when the argument is an 8-bit string. For example, ord(‘A’) returns the integer 65, ord(‘a’) returns the integer 97, ord(‘€’) (Euro sign) returns 8364. The ord function is the inverse of the chr() functions for 8-bit characters and of unichr() for Unicode objects. Character’s code point must be in the range [0..65535] inclusive. We can print math symbols since they reside starting at 8704 code point.

```
>>> for i in range(8704,8854):
    print(chr(i), end='  ‘)
∀   ∁   ∂   ∃   ∄   ∅   ∆   ∇   ∈   ∉   ∊   ∋   ∌
∍   ∎   ∏   ∐   ∑   −   ∓   ∔   ∕   ∖   ∗   ∘   ∙
√   ∛   ∜   ∝   ∞   ∟   ∠   ∡   ∢   ∣   ∤   ∥   ∦
∧   ∨   ∩   ∪   ∫   ∬   ∭   ∮   ∯   ∰   ∱   ∲
∳   ∴   ∵   ∶   ∷   ∸   ∹   ∺   ∻   ∼   ∽   ∾   ∿
≀   ≁   ≂   ≃   ≄   ≅   ≆   ≇   ≈   ≉   ≊   ≋   ≌
≍   ≎   ≏   ≐   ≑   ≒   ≓   ≔   ≕   ≖   ≗   ≘   ≙
≚   ≛   ≜   ≝   ≞   ≟   ≠   ≡   ≢   ≣   ≤   ≥   ≦
≧   ≨   ≩   ≪   ≫   ≬   ≭   ≮   ≯   ≰   ≱   ≲   ≳
≴   ≵   ≶   ≷   ≸   ≹   ≺   ≻   ≼   ≽   ≾   ≿   ⊀
⊁   ⊂   ⊃   ⊄   ⊅   ⊆   ⊇   ⊈   ⊉   ⊊   ⊋   ⊌   ⊍
⊎   ⊏   ⊐   ⊑   ⊒   ⊓   ⊔   ⊕
```

CREATING LINKED LISTS FROM LISTS

We now demonstrate a function that can construct a linked list from an ordinary list. We use a for loop to iterate through elements of the input list, and at each step we construct a new linked list by append the next element in turn as follows.

```
def list_to_link(lst):
    l_l = empty
    for x in lst:
        l_l = link(x,l_l)
    return l_l

>>> list_to_link(list(range(7)))
[[6, [5, [4, [3, [2, [1, [0,
‘empty']]]]]]]
```

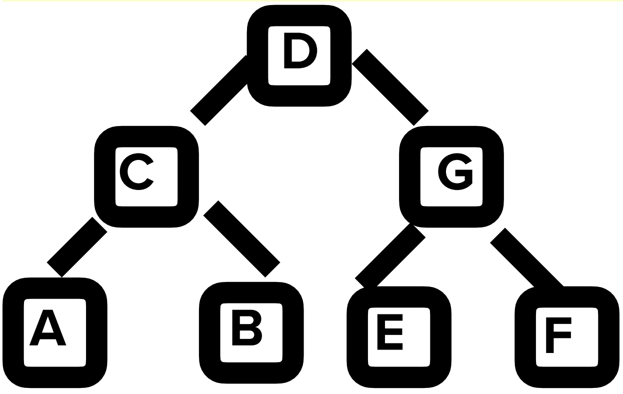
TREES

The tree is a generalization of linked list and is a fundamental data abstraction that imposes regularity on how hierarchical values are structured and processed. A tree has a root and a sequence of branches. Each branch of a tree is a tree. A tree with no branches is called a leaf. Any tree contained within a tree is called a sub-tree of that tree (such as a branch of a branch).

The root of each sub-tree of a tree is called a node in that tree.

The data abstraction for a tree consists of collection of functions including a constructor and selectors. We begin with a simplified version.

```
def tree(root_obj, branches=[]):
  return [root_obj] + list(branches)

def root(t):
  return t[0]

def branches(t):
  return t[1:]

def is_leaf(t):
  return not branches(t)


def add_branch(t,x):
  return tree(root(t),branches(t) +
[tree(x)])

alpha_tree =
  tree('D', [
   tree(‘C’, [tree('A'),tree('B')]),
   tree('G', [tree('E'),tree('F')])])
```



TRAVERSE TREE

To traverse a tree is to visit each node of the tree exactly once. Here is a function to do just that.

```
def traverse(t):
  print(root(t))
  if not is_leaf(t):
    for b in branches(t):
         traverse(b)

>>> traverse(alpha_tree)
D  C  A  B  G  E  F
```

When a node is first reached the function prints out the root value. It then checks if the node is a leaf node (this is the base case). If it is not a leaf, a recursive call is made on each tree in the list of branches. Traverse always visits nodes in a depth-first order.

If we want to reach a random leaf of the tree we can do a random walk as follows. At each node, check if it is a leaf. If it is not we can choose a random branch to continue the walk. We use the randrange function from the standard random library to choose a random branch as we go down the tree.

```
from random import randrange

def random_walk(t):
  print(root(t), end= ' ')
  if not is_leaf(t):
    random_walk(branches(t)
        [randrange(len(branches(t)))])

>>> random_walk(alpha_tree)
D C B

>>> random_walk(alpha_tree)
D G E

>>>random_walk(alpha_tree)
D G F
```

FIBONACCI TREE

The following function builds and returns a binary tree that is constructed out of the fibonacci sequence. A binary tree has exactly two branches from each non-leaf node. The function works recursively. The base case if when the value n is 1 or 0, and the function returns the single leaf with value 1. Otherwise, the function constructs a tree from the two sub-trees fibtree(n-1) and fibtree(n-2), setting them as two branches from root. The value stored in the root is simply the sum of the roots of the two branches.

```
def fibtree(n):
  if n ==1 or n==0:
    return tree(1)
  else:
    t1= fibtree(n-1)
    t2= fibtree(n-2)
    return tree(root(t1) + root(t2),
[t1] + [t2])

>>> f= fibtree(5)

>>> root(f)
8

>>> traverse(f)
8  5  3  2  1  1  1  2  1  1  3  2  1
1  1

>>> random_walk(f)
8 5 2 1

>>> random_walk(f)
8 5 3 1
```

```
>>> random_walk(f)
8 3 1

>>> random_walk(f)
8 3 2 1
```

CHAPTER 6 EXERCISES

Question 1: is_sorted
Implement the is_sorted(lst) function, which returns True if the linked list lst is sorted in increasing from left to right. If two adjacent elements are equal, the linked list is still considered sorted.

Question 2: Interleave
Write interleave(s0, s1), which takes two linked lists and produces a new linked list with elements of s0 and s1 interleaved. In other words, the resulting list should have the first element of the s0, the first element of s1, the second element of s0, the second element of s1, and so on.

If the two lists are not the same length, then the leftover elements of the longer list should still appear at the end.

Question 3: Height

Define the function height, which takes in a tree as an argument and returns the number of branches it takes start at the root node and reach the leaf that is farthest away from the root.

Note: given the definition of the height of a tree, if height is given a leaf, what should it return?

Question 4: Sprout leaves

Define a function sprout_leaves that, given a tree, t, and a list of values, vals, and produces a tree with every leaf having new children that contain each of the items in vals. Do not add new children to non-leaf nodes.

Question 5: DNA Sequence Matching

The mad scientist John Harvey Hilfinger has discovered a gene that compels people to enroll in CS2023. You may be afflicted!

A DNA sequence is represented as a linked list of elements `A`, `G`, `C` or `T`. This discovered gene has sequence `C A T C A T`. Write a function `has_2023_gene` that takes a DNA sequence and returns whether it contains the 2023 gene as a sub-sequence.

First, write a function `has_prefix` that takes two linked lists, `s` and `prefix`, and returns whether `s` starts with the elements of `prefix`.

Note that `prefix` may be larger than `s`, in which case the function should return `False`.

Question 6: Count Change (with Linked Lists!)

A set of coins makes change for `n` if the sum of the values of the coins is `n`. For example, if you have 1-cent, 2-cent and 4-cent coins, the following sets make change for `7`:

- 7 1-cent coins
- 5 1-cent, 1 2-cent coins
- 3 1-cent, 2 2-cent coins
- 3 1-cent, 1 4-cent coins
- 1 1-cent, 3 2-cent coins
- 1 1-cent, 1 2-cent, 1 4-cent coins

Thus, there are 6 ways to make change for `7`. Write a function `count_change` that takes a positive integer `n` and a linked list of the coin denominations and returns the number of ways to make change for `n` using these coins.

Question 7: Assume there is a function cons(a, b) that constructs a pair, and car(pair) and cdr(pair) returns the first and last element of that pair. For example,

```
>>> car(cons(3, 4))
3
>>> cdr(cons(3, 4))
4
```

Here is an implementation of cons which uses a function as a return value for a pair:

```
def cons(a, b):
    def pair(f):
        return f(a, b)
    return pair
```

Provide an implementation for the functions car and cdr.