DISCUSSION 08

Linked Lists, Mutable Trees, Efficiency

Mingxiao Wei
mingxiaowei@berkeley.edu

Oct 20, 2022

LOGISTICS 

- Homework 06 due today 10/20
- ANTS  
  - The whole project tomorrow 10/21
  - Submit by today for one extra point!
- Reminder - Homework 05 Recovery (Ed post #2128)

ABOUT THE MDITERM 😮

- Logistics - Ed post #2141
  - If you need ANY alterations (left-handed desk, mask-required rooms, remote, other accommodations due to DSP or otherwise), please fill out this form by Sun 10/23!!
- Familiarize yourself with the study guide - this is a good starting point to go over the topics!
- Preparations
  - Familiarize yourself with the topics in scope
  - Attend review session (or watch recordings/slides) for more topical review - see Ed for more info
  - Do practice exams!
    - Quality > quantity
    - Post on exam threads on Ed for help
    - Walkthrough videos/guide are your friend!

FROM LAST TIME... 👀

What's the best fruit?

grape!	Lychee
Peach!	Black Bear
Orange 🍑	Chinese Wampi
MANGOSTEEN	mango :)
tomato:)	Grapes
watermelon and mango	potato
watermelon	pineapple
Banana	Watermelon
Watermelon	grape
strawberries	mango
apple	Apple
apple	apple
pear	apple

LINKED LIST 

LINKED LIST

- A linked list is either:
  - An empty linked list - `Link.empty`
  - An instance from the `Link` class, containing a `first` value and the `rest` of the linked list, which is another linked list - recursive object
- When we say a "node" of the linked list, we usually refer to a `Link` object
- To check whether or not a linked list is empty, use `is` to compare it against `Link.empty`

LINKED LIST - IMPLEMENTATION

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

- `Link.empty` can be implemented as literally anything as long as we use `is` to compare it against other linked lists

LINKED LIST - REPR

```
class Link:
    empty = ()
    ...
    def __repr__(self):
        if self.rest:
            rest_repr = ',' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'
```

- `__repr__` returns a string, that, when evaluated, returns a `Link` object containing the same values

LINKED LIST - STR

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

- `__str__` returns a string where:
  - the whole linked list is wrapped by angle brackets
  - Each item is represented by their own `str()` method
  - Every two adjacent items are separated by a whitespace

CONSTRUCTING A LINKED LIST

```
lst = [1, 2, 3, 4]

# -------- Iterative Approach ---------
def iterative_constructor (lst):
    iterative_link = Link.empty
    for elem in lst:
        iterative_link = Link(elem, iterative_link)
    return iterative_link
# -------- Recursive Approach ---------
def recursive_constructor (lst):
    if lst == []:
        return Link.empty
    return Link(lst[0], recursive_constructor(lst[1:]))
```

- recursion - construct from the front
- iteration - construct from the back

LINKED LIST - PROBLEM SOLVING STRATEGIES

- Pay attention to whether it's **mutation** or **constructing a new linked list**
  - For mutation problems, the return value is often `None`
  - For problems that return a new linked list, in what order should we constrcut the list?
- To mutate a linked list, reassign its instance attributes (`link.first = ...` or `link.rest = ...`)
- Before accessing any instance attributes from a `Link` object, make sure that it's not `Link.empty`!

WORKSHEET Q2-5

TREES 

TREES

- OOP trees - mutable, use `Tree(...)` to construct

```
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
```

TREES - PROBLEM SOLVING STRATEGIES

- Pay attention to whether it's **mutation** or **constructing a new linked trees**
- To mutate a tree object:
  - reassign its instance attributes (`t.label = ...` or `t.label = ...`)
  - use list mutation method on its branches - `t.branches` is a **list** of trees!
- For mutation problems:
  - Think about which should be mutated first - the root node or its branches?
  - The return value is often `None`
  - Sometimes the case case is implicit - if we have for loop that iterates through all the branches, the body of the for loop will not be executed if `t.branches` is an empty list (i.e., when `t` is a leaf

WORKSHEET Q6, 7

ORDER OF GROWTH

ORDER OF GROWTH

- Order of growth (efficiency) - how the runtime of the function changes as the input size increases
- Input size (not the definition, but as a rule of thumb)
  - numeric input - magnitude of the number
  - Python lists - length of the list
  - linked list/trees/other recursive objects - number of nodes
- Runtime (not the definition, but as a rule of thumb)
  - often measured as the number of operations
- For 61A, we use the theta notation - for input of size n, the runtime of the function is denoted by $\Theta(f(n))$

ORDER OF GROWTH - CONSTANT

- Constant ↔ Θ(1)
- The runtime of the function does not change as the input size changes
- For example:

```
def square(x):
    return x * x
```

input	function call	return value	operations
1	`square(1)`	1*1	1
2	`square(2)`	2*2	1
...	...	...	...
100	`square(100)`	100*100	1
...	...	...	...
n	`square(n)`	n*n	1

ORDER OF GROWTH - LOGARITHMIC

- Logarithmic $\leftrightarrow \Theta(\log n)$
- Often when we keep dividing the input by a constant
- For example:

```
def foo(x):
    while x > 0:
        print('hey')
        x //= 2
```

ORDER OF GROWTH - LINEAR

- Linear $\leftrightarrow \Theta(n)$
- For example:

```
def factorial(x):
    prod = 1
    for i in range(1, x + 1):
        prod *= i
    return prod
```

input	function call	return value	operations
1	`factorial(1)`	1*1	1
2	`factorial(2)`	2*1*1	2
...	...	...	...
100	`factorial(100)`	100*99*...*1*1	100
...	...	...	...
n	`factorial(n)`	n*(n-1)*...*1*1	n

ORDER OF GROWTH - QUADRATIC

- Quadratic $\leftrightarrow \Theta(n^2)$
- For example:

```
def bar(n):
    for a in range(n):
        for b in range(n):
            print(a,b)
```

input	function call	operations (prints)
1	`bar(1)`	1
2	`bar(2)`	4
...	...	...
100	`bar(100)`	10000
...	...	...
n	`bar(n)`	n^2

ORDER OF GROWTH - EXPONENTIAL

- Exponential $\leftrightarrow \Theta(c^n)$, where c is a constant
- For example:

```
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n - 1) + rec(n - 1)
```

input	function call	return value	operations
1	`rec(1)`	2	1
2	`rec(2)`	4	3
...	...	...	...
10	`rec(10)`	1024	1023
...	...	...	...
n	`rec(n)`	`2^n`	`2^n`

ORDER OF GROWTH - OTHER NOTES

- constant < logarithmic < linear < quadratic < exponential
- Constants are ignored
- We only consider the term that grows fastest

WORKSHEET Q8

ATTENDANCE! 🤠

go.cs61a.org/mingxiao-att

- The attendance form and slides are both linked on our section website!
- Once again, please do remember to fill out the form by midnight today!!