CS1010S Tutorial 5

Wang Zexin

National University of Singapore
Year 2 QF

wang.zexin@u.nus.edu

March 5, 2017

Today's Agenda

Recap - Tuple

Why do you use tuple?

- Allow for multiple storage
- Immutable
- Allow for nested tuple

Recap - List

Why do you use list?

- Mutable
- list1 + list2
- list.append(element) → also touch on **list comprehension**
- list.extend(element)
- del a[index]
- len, min, max, in, * scalar (sequence operations)
- list.copy() → **shallow copy**, often tested in finals
- list.insert(position, element)
- list.pop(position)
- list.remove(element)
- list.clear()
- list.reverse()

Recap - Box and pointer diagram

- Most important when you are using lists than tuples
- Below is a screenshot of your lecture notes
- Why is it so? Explain using box and pointer diagram

```
lst = [1, 2, 3]
lst2 = lst
lst == lst2          →True
lst is lst2          →True

lst += [4, 5, 6]
lst                  →[1, 2, 3, 4, 5, 6]
lst2                 →[1, 2, 3, 4, 5, 6]
lst == lst2          →True
lst is lst2          →True
```

Recap - Searching

- Linear search
  - Time: $O(n)$
- Binary search
  - only when array is sorted
  - Time: $O(\log n)$

Recap - Sorting

- Selection sort
  - Time: $O(n^2)$
  - Stable
- Merge sort
  - Time: $O(n \log n)$
  - Space: $O(n)$
  - Stable
- list.sort(key = lambda x: f(x), reverse = True)
  - Sort based on the value of f(x)
  - Sort to be in reverse order
- Other sorting methods and many more ...
  - Bubble sort
  - Quick sort
  - Shell sort
  - Bucket sort
  - Heap sort

Question 1

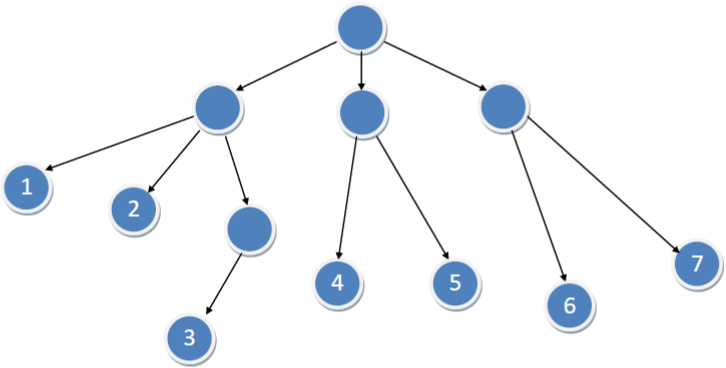
Draw box-and-pointer diagrams for the values of the following tuples.
((1, 2, (3,)), (4, 5), (6, 7))
(1, (2, 3, (4,)))
(1, (2, (3, (4, 5))))

Question 1 Discussion

- How important is your box-and-pointer diagram?
- You will never be asked to draw them.
- But, you will need to draw them for solving other problems.
- Especially some problems of Question 1 of Final Exam
- There is no fixed form that you need to follow
- As long as you understand what you draw

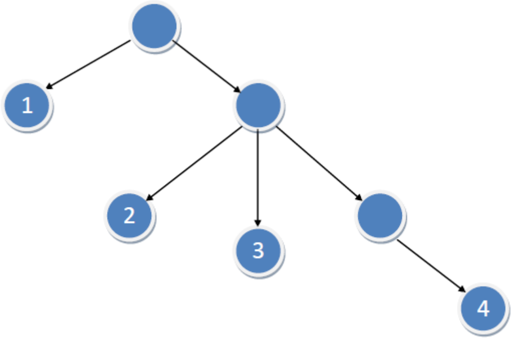
Question 1 Zexin's solution

((1, 2, (3,)), (4, 5), (6, 7))



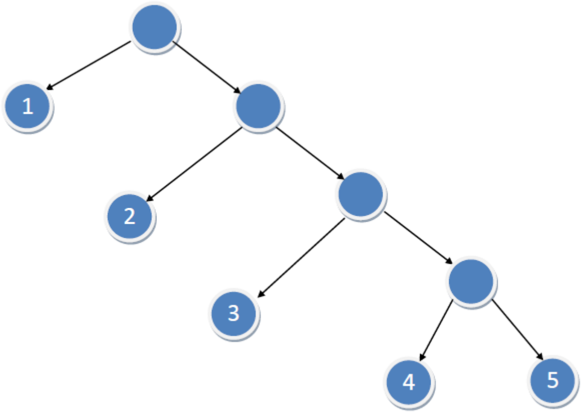
Question 1 Zexin's solution

(1, (2, 3, (4,)))



Question 1 Zexin's solution

(1, (2, (3, (4, 5))))



Question 2

Write expressions using index notation that will return the value 1 when the identifier tup is bound to the following values: (7, (6, 5, 4), 3, (2, 1)) ((7), (6, 5, 4), (3, 2), 1) (7, (6,), (5, (4,)), (3, (2, (1,)))) (7, ((6, 5), (4,), 3, 2), ((1,),))

Question 2 Discussion

Your task now is to:

- Draw the box-and-pointer diagram in your head
- And identify which element you are going to visit
- Count the index of the element and put it down

Question 2 Zexin's solution

Denote the element which you are going to visit by bolding it

- (7, (6, 5, 4), 3, (2, **1**))
- (7, (6, 5, 4), 3, **(2, 1)**)
- **(7, (6, 5, 4), 3, (2, 1))**

If we go backwards, the indices we are going to record down are: 3 and 1
Hence the answer is tup[3][1]

Question 2 Zexin's solution

Denote the element which you are going to visit by bolding it

- ((7), (6, 5, 4), (3, 2), **1**)
- **((7), (6, 5, 4), (3, 2), 1)**
- Problem now is: what is (7)?
- Is this really a problem?

If we go backwards, the indices we are going to record down is: 3
Hence the answer is tup[3]

Question 2 Zexin's solution

Denote the element which you are going to visit by bolding it

- (7, (6,), (5, (4,)), (3, (2, (**1**,))))
- (7, (6,), (5, (4,)), (3, (2, **(1,)**)))
- (7, (6,), (5, (4,)), (3, **(2, (1,))**))
- (7, (6,), (5, (4,)), **(3, (2, (1,)))**)
- **(7, (6,), (5, (4,)), (3, (2, (1,))))**

If we go backwards, the indices we are going to record down are: 3, 1, 1, 0
Hence the answer is tup[3][1][1][0]

Question 2 Zexin's solution

Denote the element which you are going to visit by bolding it

- (7, ((6, 5), (4,), 3, 2), ((**1**,),))
- (7, ((6, 5), (4,), 3, 2), (**(1,)**,))
- (7, ((6, 5), (4,), 3, 2), **((1,),)**)
- **(7, ((6, 5), (4,), 3, 2), ((1,),))**

If we go backwards, the indices we are going to record down are: 2, 0, 0
Hence the answer is tup[2][0][0]

Question 2 Zexin's solution

```
tup = (7, (6, 5, 4), 3, (2, 1))
print(tup[3][1])
tup = ((7), (6, 5, 4), (3, 2), 1)
print(tup[3])
tup = (7, (6 ,), (5, (4 ,)) , (3, (2, (1 ,))))
print(tup[3][1][1][0])
tup = (7, ((6, 5), (4 ,), 3, 2), ((1 ,) ,))
print(tup[2][0][0])
```

- Order of growth for time complexity is?
- Order of growth for space complexity is?

Question 3

Write a Python function called *even_rank* that takes in a tuple as its only argument and returns a tuple containing all the elements of even rank (i.e. every second element from the left) from the input tuple.
Test input: even_rank(('a', 'x', 'b', 'y', 'c', 'x', 'd', 'p', 'q'))
Test output: ('x', 'y', 'x', 'p')

Question 3 Discussion

- Iteration
  - Relatively more intuitive to do
  - Do you need to deal with the last element?
- Iteration
  - In fact easier to do
  - Transform the corner case that tup has len smaller than 2 into the base case!

Question 3 Zexin's solutions

```python
def even_rank(tup):
    l = len(tup)
    newTup = tuple()
    for i in range(l // 2):
        newTup = newTup + (tup[i*2+1],)
    return newTup

def even_rank(tup):
    if len(tup) < 2:
        return ()
    else:
        return (tup[1],) + even_rank(tup[2:])
```

- Order of growth for time complexity is ?
- Order of growth for space complexity is ?

Question 4

Write a function called *odd_even_sums* that takes in a tuple of numbers as its only argument and returns a tuple of two elements: the first is the sum of all oddranked numbers in the input tuple, whereas the second element is the sum of all even-ranked elements in the input.

Test input: odd_even_sums((1, 3, 2, 4, 5))
Test output: (8, 7)
Test input: odd_even_sums((1,))
Test output: (1, 0)
Test input: odd_even_sums(())
Test output: (0, 0)

Question 4 Discussion

- Iteration
  - Relatively more intuitive to implement?
  - This time we will have to deal with the last element
- Recursion
  - Easier to implement?
  - The base case is never so simple
  - Adding numbers to elements of the resultant tuple can be troublesome!

Question 4 Zexin's iterative solution

```python
def odd_even_sums(tup):
    l = len(tup)
    oddSum = 0
    evenSum = 0
    for i in range(l // 2):
        oddSum += tup[i*2]
        evenSum += tup[i*2+1]
    if l % 2:
        oddSum += tup[-1]
    return (oddSum, evenSum)
```

- Order of growth for time complexity is ?
- Order of growth for space complexity is ?

Question 4 Zexin's recursive solution

```python
def odd_even_sums(tup):
    if len(tup) == 0:
        return (0, 0)
    elif len(tup) == 1:
        return (tup[0], 0)
    else:
        result = odd_even_sums(tup[2:])
        return (result[0]+tup[0], result[1]+tup[1])
```

- Order of growth for time complexity is ?
- Order of growth for space complexity is ?

Question 5

Write a function called **hanoi** that takes in 4 parameters:

- number of disks
- source pole
- destination pole
- auxiliary pole

Test input: hanoi(1, 1, 2, 3)
Test output: ((1, 2),)
Test input: hanoi(1, 1, 3, 2)
Test output: ((1, 3),)
Test input: hanoi(3, 1, 2, 3)
Test output: ((1, 2), (1, 3), (2, 3), (1, 2), (3, 1), (3, 2), (1, 2))

Question 5 Discussion

- Iteration
  - Please do not use iteration to implement hanoi.
  - There is a way called backtracking if you are **really** bored.
- Recursion
  - Easier to implement? Of course
  - The base case is very simple
  - If there is only one disk, just move it from source to destination
  - The recurrence relation is as follows:
  - If there are more than one disks,
  - move the most upper (n-1) disks from source to auxiliary
  - move the most lower one from source to destination
  - move the most upper (n-1) disks from auxiliary to destination

Question 5 Zexin's solution

```
def hanoi(numDisks, source, destination, auxiliary):
    if numDisks == 1:
        return ((source, destination), )
    else:
        return hanoi(numDisks - 1, source, auxiliary, destination) \
            + ((source, destination),) \
            + hanoi(numDisks - 1, auxiliary, destination, source)
```

- Order of growth for time complexity is ?
- Order of growth for space complexity is ?

Extra stuff: your midterm question

If time permits, we will go through this.

```
def grill_on_flame(kebab, flame, at):
    <PRE>
    return tail(<T1>,
                <T2>,
                <T3>)


def tail(f, a, n):
    if n == 0:
        return a
    else:
        return tail(f, f(n, a), n-1)
```

- This is the first problem of your midterm question 3.
- We think this is quite hard.
- Going through this is more for the benefits of those going for remidterm.

Extra stuff: your midterm question

More details about the grill_on_flame here

```python
def grill_on_flame(kebab, flame, at):
    if flame == "":
        return kebab
    else:
        kebab = grill_piece(kebab, at, int(flame[0]))
        return grill_on_flame(kebab, flame[1:], at + 1)
```

If you are wondering where is the definition for grill_piece,
do we really need the definition of that?

- Difficulty lies in the fact that we have to start from the middle.
- The pieces to grill and the times for grilling all depend on *flame* only.
- Now we need to define something to extract those information out.

Extra stuff: your midterm question

- *tail* gives you the ability to repeat certain function n times on the same object.
- So the function we define should be grilling the piece at index $at + n$.
- And the function should be grilling that piece with time $flame[n-1]$.

So here is what we should put in the <PRE> part.

```
def grill_piece_at(n, kebab):
    return grill_piece(kebab, at + n - 1, int(flame[n-1]))
```

Extra stuff: your midterm question

So here are the answers:

The <PRE> part is

```
def grill_piece_at(n, kebab):
    return grill_piece(kebab, at + n - 1, int(flame[n-1]))
```

- The <T1> part is: *grill_piece_at*
- The <T2> part is: *kebab*
- The <T3> part is: *len*(*flame*)

Feedback & more

- Slides + relevant material available at:

  `https://github.com/wangzexin/Teaching`

- After the tutorial, if you have further questions:

  `wang.zexin@u.nus.edu`

Thank You

wang.zexin@u.nus.edu