UTM

CSC148H5
FINAL EXAM
STUDY GUIDE

Lecture Notes

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
WEDNESDAY, JANUARY 4, 2017

LECTURE 2: OBJECT ORIENTED PROGRAMMING AND INHERITANCE

Function-Based Approach

```
def sqrt(n):
      ''' (float) -> float
      '''
```

- Function make less sense when you have actors in your domain

Object-Oriented Programming

- We model actors, not functions

```
class eat_junk(....):

dan = Person()
dan.eat_junk()
```

Strange Notation for numbers:
7.sqrt()

Objects in Python:
String, list tuple, int, float, dictionary....

Sometimes we want to model stuff that isn't built-in already. To introduce a new type of object, we use **a new class.**

1. What can the new type of object do?
    - Actions (Methods)
    - Ex. If the class is person [`class person:`]
        - Methods: walk, sleep, cry, eat, pass_CSC148, drive
2. What does the object have?
    - Attributes, implemented using instance variables
    - Ex. If the class is person [`class person:`]
        - Attributes for Person: hair colour, eye colour, height
3. What are interactions with other objects?
    - Often someone gives you a spec for what your software should do. It's up to you to do an OO analysis of that spec.

Once you do the analysis (above), then you think about writing code.

- Find `classes` by finding nouns in the specification
- Find `methods` by searching for verbs
- Find `attributes` by searching for minor nouns

Example:
In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. Points are often written in parentheses with a comma separating the coordinates. For example, (0, 0) represents the origin, and (x, y) represents the point x units to the right and y units up from the origin. Some of the typical operations that one associates with points might be calculating the distance of a point from the origin or from another point, or finding a midpoint of two points, or asking if a point falls within a given rectangle or circle.

- Classes
    - point
- Methods
    - Calculate distance to origin, calculate distance between points, calculate midpoint, check if inside given rectangle, check if inside given circle, print points
- Attributes
    - x-coordinates
    - y-coordinates

```
from math import sqrt

class Point:
      def __init__ (self, x=0, y=0):
''' (Point, float, float)
Construct Point with given x and y coordinates
'''
self.x = x
self.y = y
```

Example:
A rational number consists of a numerator and denominator; the denominator cannot be 0. Rational numbers are written like 7/8. Typical operations include determining whether the rational is positive, adding two rationals, multiplying two rationals, comparing two rationals, and converting a rational to a string.

- Classes
    - Rational Number
- Methods
    - Determining whether the rational is positive, add, multiply, comparing two rationals, converting a rational to a string, print rational
- Attributes
    - Numerator
    - Denominator

The *cannot be 0* would not be considered a class, method or attribute. It is considered an exception. More information later in CSC148.

```
class Rational:
    def __init__(self, num, den):
        self.num = num
        self.den = den

    def is_positive(self):
        ''' (Rational) → bool
        Return whether self is positive '''
        return self.num >= 0 and self.den > 0 \
            or self.num < 0 and self.den < 0
        # The backslash continues the code on the next line

    # keep defining functions
    return self.num / self.den >= 0
```

Inheritance

Think about enemy ships in a game.

✓ Class: Ship
✓ Methods: move, shoot, take_damage, explode, dodge.....
✓ Attributes: health, location, number of bombs, speed.....

Now you want to add a new kind of ship to make the game more fun. You decide to add an indestructible ship. You're likely going to have all the same methods and attributes as before.

```
There's only one difference between the two classes....
    class Ship:
        def take_damage(self):
            self.hitpoints = self.hitpoints - 5
            ...

    class IndestributibleShip:
        def take_damange(self):
            pass
```

You have two choices for how to implement the indestructible ship:

1. Copy your original ship class, and change take_damage to be 'pass'.
   ISSUE: This leads to tons of code duplication which can also lead to replicating a bug many times
2. Inheritance - you only specify what changes....

```
class IndestructibleShip(Ship)
```

```
      def take_damange(self):
            pass

a = IndestructibleShip()
a.move()
a.shoot()
# all original methods are available
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2016)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
FRIDAY, JANUARY 6, 2017

LECTURE 3: INHERITANCE

Inheritance

- Many times, a new class will have much in common with an existing class
- Copying-and-pasting from the old class, and then making a few changes, is a bad idea
    - What if you then discover a bug in the original class?
    - What if you find that there are many classes you'd like to create from the original one?
- Allows a new class to specialize an existing class by specifying only what is different between the inherited class and original class
- Class that inherits is called a subclass
- Class that is inherited from is called a superclass

Is-a

- Inheritance models "is-a" relationships
- The subclass "is-a" subset of superclass
- Example
    - IndestructableShip is a Ship?
        - Yes; inheritance can be used

Let's think about points and line segments.

```
class Point:
class LineSegment:
```

Is a Point a Line Segment? No
Is a Line Segment a Point? No; No inheritance used.

This is a has-a relationship.

The line segment has two points.

You use instance variables to represent what the object has.

```
class LineSegment:
      def __init__(self, p1, p2):
            self.p1 = p1
            self.p2 = p2
```

Inheriting from a Class

- To declare a class, you would start off with

```
class Name:
```

- **To inherit from another class, add the superclass name in the class parameters**

```
class subclass(main_class):
class IndestructibleShip(Ship):
```

UTSC Class Example (from previous UTSC Exam)
Classes: building, room, house, business
Inheritance is useful for business and house. They're both kinds of buildings.

```
class Building:
class House(Building):
class Business(Building):
```

No inheritance for rooms. Buildings have rooms, a building is not a room. Rooms will be an attribute of your objects.

Exceptions: house has at most ten rooms, business cannot have a room named bedroom, business rooms are all >= 100 square feet.

Exercises

Question 1: Write one or more classes for the following specification. Begin by carrying out an object-oriented analysis to determine the classes, methods, attributes, and interactions.

Context: An airline reservation system

Each seat on a plane is classified as business class or economy, and has a unique name (like "17C"). Passengers have booking IDs (a mix of letters and numbers). When they book a seat, they request their preferred class (business or economy) and are given any seat in that class. If the chosen class is full, then their booking is unsuccessful. This airline gives passengers no choice about their specific seat. We want to be able to report on how full a flight is: the percentage of seats that are booked in economy, in business class and overall.

```
class Seat:
      def __init__(self, category):
            if category == 'e': # 'economy'
                  ...
if category == 'b': # 'business'
      ...
```

Main Class: Reservation System
Methods: assign seat, amount booked overall, amount booked in business, amount booked in economy

Attribute: List of business seats, list of economy seats

```
>>> s = ReservationSystem(['eco1','eco2'],['bus1'])
s.book_seat()
s.percentage_booked_overall()
```

What is a seat? It's basically a string
If something is just an int or a string or a float, think about whether it should really be a new class.

Question 2: Write one or more classes for the following specification. Begin by carrying out an object-oriented analysis to determine the classes, methods, attributes, and interactions.

Context: An inventory system

Items are for sale, each at its own price. Items are identified by their item number, and they also have a text description, such as "bath towel". There are categories that items belong to, such as "housewares" and "books". We need to be able to print a suitable price tag for an item (you can decide exactly the format). Sometimes an item is discounted by a certain percentage. We need to be able to compare two items to see which is cheaper.

```
class Item:
# Attribute: price, item number, description

class Category:
# Attribute: category_name

# Functions: print_price, discount_price, compare_items
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, JANUARY 9, 2017

LECTURE 4: Abstract Data Types (ADTs)

Abstract Data Types (ADTs)

Abstraction: Ignoring certain details to make problems easier to solve

- *Abstract* because there is no mention of implementation details
- *Data type* because it includes data and operations on those data
- Allows us to directly describe the data used in our problems, independent of implementation details

What are some examples of abstract data types (ADTs) that you've already used?

- Dictionary

```
d = {}
d[5] = 10
```

```
>>> d
{ 5: 10}
```

You can use these things without knowing how they work.

Stack ADT

- A stack is a sequence of objects
- Objects are removed in the opposite order that they are inserted
- Last in, first out (LIFO)
  - Similar to plates in a cupboard. You take out the top plate, you put them back in the stack. You don't take the bottom plate without taking out the top
- The object last inserted is at the **top** of the stack

Example of the Stack ADT Function

```
def second(value):
      result = abs(value)
      return result
def first(value):
      result = second(value)
      return "value = " + str(value)
```

```
def main():
      value = -50
      result = first(value)
print *result)
return None

main ()
```

Process

1. Main function gets called
2. FIrst function gets called
3. Second function gets called
4. Second returns
5. First returns
6. Main returns

We'll use this real-world description of a stack for our design:
A stack contains items of various sorts. New items are pushed on to the top of the stack, items may only be popped from the top of the stack. It's a mistake to try to remove an item from an item stack. We can tell how big a stack is, and what the top items is.

Stack Operations

```
push(o)  # Add a new item o to the top of the stack
pop()  # Remove and return top items
is_empty()  # Test if stack is empty
peek()  #return the top item (without removing it)
size()  # Return number of items in stack
```

Example

```
from stack import Stack
>>> s1 = Stack()
>>> s1.push(4)
>>> s1.push(6)
>>> s1.pop()
6
>>> s1.pop()
4
>>> s1.pop()
IndexError: pop from empty list
```

Implementation Possibilities

The public interface of our Stack ADT should be constant, but inside we could implement it in various ways

- Use a Python list, which already has a **`pop`** method and an **`append`** method
- Use a Python list, but push and pop from position 0
- Use a Python dictionary with integer keys 0,1,..., where the highest-numbered key is the most-recently pushed item

Example

- Start with an empty stack: []
- Push 5: [5]
- Push 8: [5,8]
- Pop: [5] (and returns 8)
- Pop: [] (and returns 5)
- Pop: **Error**

```
class Stack:
      ''' A last -in, first-out (LIFO) stack of items'''

      def __init__(self: 'Stack') -> None:
            '''A new empty Stack'''
            self._data = []

      def pop(self: 'Stack') -> object:
            '''Remove and return the top item.
            >>> s = Stack()
            >>> s.push(2)
            >>> s.push(3)
            >>> s.pop()
             3
            '''


      return self._data.pop()

      def is_empty(self: 'Stack') -> bool:
            ''' REturn whether the stack is empty.

            >>> s = Stack()
            >>> s.push(4)
            >>> s.pop()
            4
            >>> s.is_empty()
            True
            '''
```

```
        return self._data == []
        # or return len(self._data) == 0

    def push (self: 'Stack', o: 'object) -> None:
        ''' Place o on top of the stack.'''
        self._data.append(o)
```

Uses For A Stack

- Keep track of pages visited in a browser tab
- Keep track of function calls in a running program
- Check for balanced parentheses
- Keep undo/redo history in a text editor or word processor

and lots more

Queue ADT

- A sequence of objects again, but the operations are different than the stack ones
- Objects are removed in the same order they are inserted (first in, first out; FIFO)

```
enqueue(o)  # Add o to the end of queue
dequeue()  # Remove and return object at the front of queue
front()  # Return object at the front of the queue
is_empty()  # Test if queue is empty
size()  # Return numbers of items in queue
```

Example

- Queue: First in first out (FIFO)
- Lineup at a restaurant; restaurant is about to open
- First, Michael shows up
- Then, Nikki shows up
- Then, Dan shows up
- Then, Restaurant opens....who gets in first?
    - Michael
    - Nikki
    - Dan

One way to implement a queue:

```
def enqueue (self, item):
      .... append to the right end of a list

def dequeue(self):
      .... remove from the left of the list
```

What does this code do?

Imagine that we have stack s1 with some items in it. Let's say that the items are 3 4 5 6 7
7 is the top of the stack

I want to remove and print every item on the stack.

```
while not s1.is_empty():
      item = s1.pop()
      print(item)
      # print (s1.pop())
```

Output: 7 6 5 4 3
Let's say that you wanted to print them in the opposite order...how?

```
lst = []
while not s1.is_empty():
      lst.append(s1.pop())
lst.reverse()
for item in lst:
      print(item)
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
WEDNESDAY, JANUARY 11, 2017

LECTURE 5: STACK AND PRIORITY QUEUE ADT

Question
Write the body of the function below so that it satisfies its docstring. Assume that module stack.py defines a class stack that provides the usual methods: is_empty(), push(item), pop().

For full marks, your code must not depend on any details of the implementation of class Stack. in other words, the only thing you can do with a Stack object is to call some of its methods

```
for stack import Stack

def size(s):
      ''' (Stack) -> int
      Return the number of items on Stack s, *without* modifying s.
      (It's ok if the contents of s are modified during the execution
of this function, as long as
everything is restored before the function returns)
'''
# Hint: You can use more than one stack.
```

Solution 1

```
      return len(s._data)
      # Mark: 0/10
      # Cannot use underscore
```

Solution 2

```
      total = 0
      for items in s:
            total = total + 1
      return total
      # Mark: 0/10
      # You can't access the information like that. You can only check
the top. Cannot iterate.
```

Solution 3

```
      total = 0
      while not s.is_empty():
            s.pop()
            total = total + 1
```

```
        return total
        # Mark: 2.5/10
        # Does something. Returns the directed value but modifies the
stack
```

Solution 4

```
        s1 = Stack()
        s1.push(4)
        s1.push(5)
        #....
        result = size(s1)
        print(result)
        result = size(s1)
        print(result) #0
```

Good Solution 1:

```
new_s = Stack()
        total = 0
while no s.is_empty():
                total = total + 1
                item = s.pop()
                new_s.push(item)
                # new_s is the reverse of the original stack s
        while not new_s.is_empty():
                item = new_s.pop()
                s.push(item)
        return total
```

Another way to solve this is using a list instead of a second staff. THe risk of using list instead of stack is when putting the list back to together, the order could be messed up.

Good Solution 2

```
        list = []
        while not s.is_empty():
                lst.append(s.pop())
                total = len(lst)
        #... now put everything from lst back to s
        return total
```

Priority Queue ADT

- A sequence of objects
- Objects are removed in order of their priority

```
insert(o) # Add o to the priority queue
extractMin() #Remove and return object with minimum value
min()  #Return object with minimum value
is_empty  #size....same as previous
```

Example of priority queue
There's a hospital that takes patients.
Ritu shows up first...with a hangnail (not so severe)
Dan shows up second...with Dan is having a baby (severe)
Nikki shows up third...with serious case of vomiting (kind of severe)

In a stack, Nikki would be seen first. In a queue, whoever is the most severe will be looked at first. Hospital uses a priority queue...doesn't matter when you show up; what matters is severity of condition. Order will be Dan, Nikki, Ritu.

ADT Puzzle
You're given a list of integers; you're goal is to transform the list into a new list according to the following rule:
Find the leftmost pair of consecutive numbers in the list whose values are x and x+1, replace them by the single element whose value is 2x+1 and repeat the process using this new list. If no pair of integers satisfies this property, the process is complete.
Example: list [1,2,3,4] is transformed to [3,3,4] and then to [3,7]

Question 1: Could you scan from the right instead? Does it matter?
[1,2,3,4]
If I scan from the right, I get:
[1,2,7]
[3,7]. #that's the same result got by scanning from from the left

It matters which side I scan from....
[1,2,4,5]
Left:
[1,2,4,5]
[3,4,5]
[7,5]
Right
[1,2,4,5]
[1,2,9]
[3,9]

Question 2: which ADT is useful for helping us solve this question?
I think that a stack is useful here....
[1,5,10,1,2,4,8,3,4]
Stack: 1 5 10 15 8

Think about using a stack in this way to come up with an o(n) solution.

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
FRIDAY, JANUARY 6, 2017

LECTURE 6: STACK CONTINUED

Question
Why is lst.append(item) so much faster than lst.insert(0,item)?

Appending to a list is very easy:
[1, 2, 3, 4, 5, 6, 7]
Constant number of operations.

Now inserting at the beginning:
[0,   1, 2, 3, 4, 5, 6, 7]
Inserting at the beginning of a list is super slow.
Appending is faster than inserting.

Thinking about a list with 1,000,000 elements.
Append: 2 operations: add element and increase list size
Insert: 1,000,000 operations; every element has to be moved to the right.

Example 1
Consider the following code.

```
def change(s1: Stack) -> None:
     s2 = Stack()
     while not s1.is_empty():
        first = s1.pop()
        if s1.is_empty():
            s2.push(first)
        else:
            second = s1.pop()
            s2.push(second)
            s2.push(first)
    while not s2.is_empty():
      s1.push(s2.pop())
```

Given the below stack s, what are the contents of s after running change(s)?
For the 1 2 3 4 stack, we will get this from the first loop:
s1:
s2: 3 4 1 2

First loop is done. Now second loop...
s1: 2 1 4 3
s2:

Explain in plain English what 'change' does:

- can't refer to s2

Don't do this: "the function first checks whether s1 is empty. If not, it removes an item from s1. Then it checks if s1 is empty. If it is, it puts the item on s2; if not,it removes another item from s1. Then it pushes......."

- Every pair of numbers switches its position. [Mark: 1 of 5]
- Every pair of numbers is switched, but then the whole stack is reversed. [Mark: 0 of 5]

We have to refer to s1 in our explanation.....

- Swap each element in s1 with the element next to it.

Great answer: Each pair of neighboring elements in s1 swap places; if s1 is of odd length, the bottom element remains in-place.

Tip: don't "invent" special cases. Even if your code has special cases, it doesn't mean that your explanation has to.
In this case, the "odd-length" stack is a special case because otherwise it isn't clear what happens to the element that's left out of a pair.

Question 2
Write function `roll` *below.* `roll` *takes the element at position* `n` *on the stack and makes it the top element. (The top of stack is position 1, the next is position 2, and so on)*
For example, consider stack `s` *as A B C D, where the left end is the bottom of the stack and the right end is the top (so D is at the top of the stack).* `roll(s, 2)` *will yield A B D C and* `roll(s,3)` *will yield A C D B.*

Solution

```
def roll(s: Stack, n: int) -> None:
      s2 = Stack()
      for index in range (0, n):
            s2.push(s.pop())
      s3 = Stack()
      while not s2.is_empty():
            s3.push(s2.pop())
      while not s3.is_empty():
            s.push(s3.pop())
```

Question 3:
There are many ways to implement AFTs. Implement the queue ADT below using a Python dictionary.

```
class Queue:
''' A first-in, first-out (FIFO) queue of items'''

def __init__(self: 'Queue') -> None:
''' A new empty Queue,'''
      self._data = {}

# implement enqueue, dequeue, is_empty, front
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, JANUARY 16, 2017

LECTURE 7: EXCEPTIONS

Test 1 Details

- Friday Feb 3, starting at 5:30pm - 6:20pm
- Location: IB 110 or IB ____
- Coverage: Content from the first 4 weeks of lectures/classes

---

Stack/Roll Solution

Recall: Question tells you that "top of stack is position 1, next one down is position 2,...."

```
from stack import Stack
def roll (s: Stack, n: int) -> None:
      '''Perform the roll operation on s'''
# s: a b c d TOP
# roll(s,2)
# a b d c
```

Outline of how to solve this:

1. Move (pop) the top n-1 items from s to new stack s2
2. Pop item n from s; save that in a variable
3. Pop everything from s2 onto s.
   s is the same as it was at the beginning, except missing element n.
4. Push element n on s

Implementation

```
def roll(s: Stack, n: int) -> None:
      s2 = Stack()
      for index in range (0, n):
            s2.push(s.pop())
      s3 = Stack()
      while not s2.is_empty():
            s3.push(s2.pop())
      while not s3.is_empty():
            s.push(s3.pop())
```

Exceptions

- So far, our Python functions have returned values or modified objects
- But what if our function cannot complete successfully - then what should happen?
    - eg. Calling pop on an empty stack
    - Trying to create a fraction with 0 denominator

What is another example that we've had in this course of some constraint:

✓ Trying to modify a string
✓ Buildings example: business can't have bedroom, every room in a business must be less than or equal to 100 square feet

Imagine pop method for stacks....
This is how C, Unix, Linux etc. does it:

```
def pop(self):
      if s.is_empty():
            return -1 # -1 means bad case
      # return and remove the top item from stack

s =Stack()
# ..... operations on s
item = s.pop()
if item == -1:
      #.... handle error
else:
      #.... pop work successfully
```

In some languages (e.g. C), functions return “special values” to signify errors.

Problems with this approach:

1. You have to check every function/method call to see whether it failed
2. How do you know whether -1 means error or successfully removed -1 from stack

```
s = Stack()
s.push(-1)
item = s.pop()
if item == -1:
      print ('error')
```

There's no safe/reasonable special value to use in all cases.

What are exceptions?

- Exceptions allow you to structure code in a natural way so that error handling and recovery are isolated from the regular flow of your program
- An exception is an object that indicates an exceptional situation (not necessarily a problem)
- An exception gets raised during program execution & transfers control to an exception handler
- Exceptions must be "caught" by an exception handler somewhere along the line, or your program will crash

Examples of Exceptions

```
>>> 10 * (1 / 0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> 4 + junk
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name ’junk’ is not defined
```

```
>>> 1 / 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> a = [1,2,3]
>>> a[8]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> 'dan' + 4
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

```
>>> Junk = 'abc'
>>> 4 + Junk
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: ’int’ and ’str’
```

How to raise the error message

Two Ways to raise an error message:

```
# Raise ValueError with no message
>>> raise ValueError
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError

# Raise ValueError with message
>>> raise ValueError(’invalid time/date value’)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid time/date value
```

Example

```
>>> raise valueError('Andrew programming')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: Andrew Programming
```

User-Defined Exceptions; Raising exceptions

```
class AdnrewException(Exceptiion):
...     pass
...
>>> raise AndrewException
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
__main__.AndrewException
>>> raise AndrewException('Andrew typing')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
__main__.AndrewException: Andrew Typing
```

Handling Exceptions

- If a piece of code can raise an exception, you can put it in a `try` block
- If there is an associated `except` block that matches the kind of exception raised, then that block will handle the exception
- There may be more than one `except` block that matches; in this case, the first is used
- An except block "matches" a raised exception if the except block names the same class or a superclass of the exception

Ask user for filename:

```
filename = input('Enter filename: ')
try:
      f = open(filename)
except FileNotFOundError:
      … ask the user again
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
WEDNESDAY, JANUARY 18, 2017

LECTURE 8: EXCEPTIONS PART 2

Exceptions are used to handle runtime errors or unexpected cases.
Not to be used for actual bugs/errors in your code.
Exceptions used for cases that you can't control

Recap: Handling Exceptions

- If a piece of code can raise an exception, you can put it in a `try` block
- If there is an associated `except` block that matches the kind of exception raised, then that block will handle the exception
- There may be more than one `except` block that matches; in this case, the first is used
- An except block "matches" a raised exception if the except block names the same class or a superclass of the exception

Goal for this lecture: Learn how to handle/catch exceptions so they don't terminate our code

```
try:
      f = open('myfile.txt')
      s = f.readline()
      i = int(s)
      print ('success')
except IOError:
      print('Input/Output error.')
except ValueError:
      print ('Could not convert data to an integer.')
print ('continuing')
```

Execution with Exceptions

- If no exception occurs, no except block is executed
- If an exception occurs and an except block handles it, execution continues following the enclosing try block
- If an exception occurs and no except block handles it, the exception propagates up the function call stack until it is handled or terminates the program

Tracing Exception Code Example

```
def bad():
        return 2 / 0

def bad_caller():
        try:
                bad()
        except ValueError:
                print("Caught exception in bad_caller!")
def entry():
        try:
                bad_caller()
        except ZeroDivisionError:
                print("Caught exception in entry!")

entry()
```

When you run this code, you get

```
Caught exception in entry!
```

Process:

- entry gets called
- bad_caller gets called
- bad gets called
- bad generates an exception; bad terminates exceptionally because it doesn't have an exception handler for ZeroDivisionError
- We're back in bad_caller now, and going through its except blocks
- bad_caller has no handler for ZeroDivisionError; terminates exceptionally
- We're back in entry, checking its except blocks
- entry DOES have an exception handler for ZeroDivisionError; exception gets handled here

Another Tracing Exception Code Example

```
# ArithmeticError is a superclass of ZeroDivisionError
def bad():
        return 2 / 0

def bad_caller():
        try:
                bad()
        except ArithmeticError:
                print("Caught exception in bad_caller!")
def entry():
        try:
                bad_caller()
```

```
      except ZeroDivisionError:
            print("Caught exception in entry!")

entry()
```

Another Example using the exception blocks:

```
ok = False
try:
      f = open('myfile.txt')
      ok = True
except IOError:
      print('Input/Output Error')
if ok: #if True, file must be open
      try:
            s = f.readline()
            i = int(s)
      except ValueError:
            print('Could not convert data to an integer.')
      finally:
            f.close()
            print('File Closed')
```

Summary: finally block is for code that you always want to run, whether there was an exception or not. Usually used for cleanup activities: closing a file, closing network connection, printing exit/termination message, releasing resources that your program is using.

So far, you know these blocks for exceptions:

```
try
except
finally
```

The else block will be taught on Friday.

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
FRIDAY, JANUARY 6, 2017

LECTURE 9: EXCEPTIONS

Practice on Exception

Question 1
What is the output produced by the following code?

```
def f2(x):
      if x % 2 == 1:
            raise ArithmeticError
      return x // 2

def f1(x):
      while x > 0:
            try:
                  print(f2(x))
            except ArithmeticError:
                  print('caught')
            finally:
                  x = x - 1
f1(5)
```

Output of the code

- caught
- 2
- caught
- 1
- caught

Question 2
What is the output produced by parts a to e in the following code? Indicate the type of error followed by its corresponding print statement.

```
def division (a,b):
      try:
            answer = a / b
      except ZeroDivisionError:
            print('Cannot divide by zero')
        except TypeError:
            print('Wrong Type')
        except NameError:
            print('Name not defined inside')
        else:
            print('Answer available')
        finally:
            print ('End of program')

if __name__ == '__main__':
        # Part a
        division (6, 0)
# Cannot divide by 0
# End of program

        # Part b
        division(0, 42)
        #Answer Available
        #End of program

        # Part c
        division(100, 50)
        #Answer available
        #End of program

        # Part d
        division ('science',1)
        #wrong type
        #End of program

        # Part e
        try:
            division(x,1)
        except NameError:
            print('Name not defined outside')
        #name not defined outside
```

Python Idioms

- **`[<expression> for <variable> in <iterable>]`**
    - The `<variable>` steps through the `<iterable>` (like a for-loop)
    - For each value of `<variable>`, `<expression>` is evaluated and added to a running list

Example

```
>>> lst = [2, 4, 10, 20]
>>> for i in range(len(lst)):
...   lst[i] = lst[i] + 5
...
>>> lst
[7, 9, 15, 25]
>>> lst = [2, 4, 10, 20]
>>> lst
[2, 4, 10, 20]
>>> [x + 5 for x in lst]
[7, 9, 15, 25]
>>> lst
[2, 4, 10, 20]
>>> lst = [x + 5 for x in lst]
>>> lst
[7, 9, 15, 25]
```

Suppose we have a list of lines that each end with a newline.
Create a new list of the same elements but without the new lines

```
>>> lst = ['   abc   ', 'def\n', 'ghi\n\n']
>>> [s.strip() for s in lst]
['abc', 'def', 'ghi']
```

Zip

`zip (iterable1, iterable2,...)`

- Combine the first elements of each iterable, second elements of each iterable etc.

```
# Zip Example 1
>>> list(zip([1, 2], [3, 4]))
[(1, 3), (2, 4)]
>>> for pair in zip([1, 2], [3, 4]):
...   print(pair)
...
(1, 3)
(2, 4)

# Zip Example two
>>> names = ['Dan', 'Samar', 'Ritu']
>>> ages = [22, 25, 28]
>>> names
['Dan', 'Samar', 'Ritu']
```

```
>>> ages
[22, 25, 28]
>>> zip(names, ages)
<zip object at 0x0060A1C0>
>>> list(zip(names, ages))
[('Dan', 22), ('Samar', 25), ('Ritu', 28)]
```

Using Comprehensions with Zip

Here zip creates a list of tuples, and the comprehension multiplies corresponding elements.

```
>>> x = [1, 2]
>>> y = [3, 4]
>>> [a * b for a, b in zip(x, y)]
[3, 8]
```

`[<expression> for <variable1> in <iterable1> for <variable2> in <iterable2> ...]`

A nested loop over the iterables.

```
>>> [(x, y) for x in range(2) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]
```

Filter

- `filter` is a functional tool
- Takes a `function` as a parameter
- Call `function` on each element of iterable and return the elements that cause the function to return True

`filter (function, iterable)`

```
>>> def is_even(x):
...   return x % 2 == 0
...
>>> is_even(4)
True
>>> is_even(7)
False
>>> lst = [1, 2, 3, 4, 5, 6, 10, 11, 12]
>>> filter(is_even, lst)
<filter object at 0x00600FD0>   #id may vary for your case
>>> list(filter(is_even, lst))
[2, 4, 6, 10, 12]
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, JANUARY 16, 2017

LECTURE 10: RECURSION

Threshold Concept in Computer Science
After you learn it, you don't think about computer science in the way as before.
Anything you can solve with recursion can be solved using a loop too

Recursion: Powerful problem-solving technique for helping us solve difficult problems

```
# Example without Recursion
def fact(n):
      total = 1
      for i in range(n+1):
            total = total * i
      return total
```

Example where recursion is useful; Binary Codes
Binary Code - a string of 0s and 1s
ex. '11100' , '0100011' , '' , '0' , '1' ...
also called binary string

We're going to be given an integer r, and we want to generate all binary code of length r.

```
def codes(r):
      ''' (int) -> list of str
      Return all binary codes of length r.

      >>> codes(2)    # What are all the binary codes of length 2?
      ['00','01','10','11']     # 4 binary codes of length 2
      >>> codes(1)    # All binary codes of length 1
      ['0','1']
      >>> codes(0)    #All binary codes of length 0
      # cannot be empty list [], there is ONE binary string of length
      0; empty string
      ['']
      >>> codes(3)  #All binary codes of length 3
      ['000','001','010','011','100','101','110','111']
      '''
```

Computer science experts rarely think about how recursion actually works inside.

Abstraction

Imagine that you have a solution to a smaller version of the subproblem that you want to solve.

Let's say that you're trying to solve codes(3).
Imagine that you have a solution to codes(2).
Your goal is to solve codes(3) using that solution to codes(2).
THink about how to extend codes(2) to codes(3).

Someone gives us this...
`['00','01','10','11']`

Look at the code '00'.

- If we add a '0' to the end: '000'
- If we add '1' to the end of '001'

Using '00', we managed to generate 2 of our length-3 codes.

Length-2 codes: 4. Length-3: 8
If every length-2 code gives us 2 length-3 codes, we're done.

Starting with '01' now....

- If we add a '0' to the end: '010'
- Add a '1' to the end: '011'

Let's start with '10':

- Add '0' to the end: '100'
- Add '1' to the end: '101'

Start with '11':

- Add '0' to the end: '110"
- Add '1' to the end '111'

We generated all 8 of length 3. Each length-2 code gave us 2 length-3 codes.

Algorithm
Given any size r-1, we know how to solve the problem for size r.
You take every code of r-1, and add '0' to the end.
Then, take every r-1 code again, and add a '1' to the end.
That gives you all binary codes of length r.

The way that we obtain the smaller solution is through recursion.
Recursion: calling the function that's currently being defined, but with a smaller value

Template for a recursive function has two parts:

1. One or more base cases
2. One or more recursive cases

```
def codes(r):
      if r == 0:
            return ['']
      # r > 0
smaller = codes(r-1)  #smaller is a list of str
      result = [] # this will be all length-r codes
      for code in smaller:  # code is a str
            result.append(code + '0')
            result.append(code + '1')
      return result

# smaller might be ['0', '1']
# It could also be ['00', '01', '10', '11']
# It's all of the codes of length r-1.
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
WEDNESDAY, JANUARY 25, 2017

LECTURE 11: RECURSION PART 2

Review: What is Recursion?

- Solving a problem by reducing it to subproblems, then combining the subproblem solutions to solve the original problem
- Subproblems must have the same structure as the original problem and be easier to solve
- Subproblems are so simple that they can be solve directly (without reducing them harm)

Example 1: Binary Recursion

```
# Binary example from the previous class
def codes(r):
      ''' (int) -> list of str
      Return all binary codes of length r.
      '''
      if r == 0:
            return ['']
      small = codes(r-1)
      lst = []
      for item in small:
            lst.append(item+'0')
            lst.append(item+'1')
      return lst
```

Intuition for why this function is correct.
Codes(0) is correct. Why?
[''] is returned; that's all of the length-0 codes

codes(1)....what does this do?
It does small = codes(1-1)
so, small = ['']
lst becomes ['0', '1']
codes(1) returns ['0','1']

What does codes(2) do?
small becomes ['0','1']
lst becomes ['00','01','10',11']
so codes(2) is correct? yes

Example 2: Finding the length of a string

Two reasons why this isn't good:

1. Python already has len(str) built-in
2. Even if you didn't use len, you could just use a loop to do this.

We're going to solve this recursively.
Think about the structure of a string of a certain length.

How can we split up a string?

1. Consider the string without the first character

'abcde' -> 'bcde'

2. Consider the string without its last character

'abcde' -> 'abcd'

3. Split string in middle; consider each half separately

'abcde' -> 'ab' and 'cde'

Let's imagine finding the length of some string: 'canada'
If you knew the length of 'anada' ... call that x.
What's the length of 'canada' then? x+1

If you knew the length of 'canad' ... call that y.
Then, length of 'canada' is y+1

What's the simplest case here? Base case
length of '' is 0

```
def string_length(s):
      ''' (str) -> int
      Return length of string s.

      >>> string_length('canada')
      6
      '''
      if s == '':
            return 0
      no_first = s[1:]
smaller = string_length(no_first) #smaller is an int
return smaller + 1

string_length ('') #definitely returns 0

string_lengt('c')
# no_first is ''
# smaller = string_length ('') = 0
# 0 + 1 = 1 ... so 1 is returned

string_length('ca')
# Smaller will end up being 1
```

Task: Solve this problem by removing the last character.
How do I remove the last character from a strong?

```
smaller = s[:-1]
```

Problem: Finding all permutations of string.

You're given a string as input: 'abc'
Your goal is to generate all of its permutations: How many are there? 3! = 6
They are: abc acb bac bca cab cba

As with all recursive functions, we have to:

1. Figure out the base cases
    Our base case here is the empty ''
    What are the permutations of ''?
    ['']. '' is the only permutation of the empty string

2. Understand the recursive structure of the problem. How do you use a solution to a smaller problem to solve a larger problem
    Let's think about string 'abc', finding all permutations of it.
    Imagine that you already had a solution to a smaller subproblem
    Imagine that we have a solution to the 'bc' subproblem
    All permutations of 'bc': ['bc', 'cb']

We have to go from bc cb **to** abc bac bca cab cba

bc has to give us some of the length-3 permutations.
Think about inserting 'a' back into 'bc'

- abc. inserted a at the beginning
- bac. inserted a in the middle
- bca. inserted a at end

These are the permutations of abc where b comes before c

Let's try to get three permutations out of cb as well.

- acb. inserted a at beginning
- cab. inserted a in the middle
- cba. inserted a at end

That's all of the length-3 permutations of abc.

```
def perms(s):
      ''' (str) -> list of str
      Return all permutations of s.
      '''

      >>> perms('abc')
      ['ab','ba']
      '''

      if s == '':
            return ['']
      smaller = perms(s[1:])
      #eg. smaller is ['ab', 'ba']

# Task For each of those smaller permutations: insert s[0] in every
position to generate the permutation of s

# return ....
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
FRIDAY, JANUARY 6, 2017

LECTURE 12: RECURSION 3

Recursion on a List

- Similar to how recursion works on strings
    - If you remove the first element from a list, then the rest of it is still a list, and a smaller list
    - You can also remove the last element
    - You can split list in middle which gives you two smaller lists

Let's write a recursive function to return the sum of a list of integers. No nesting.

```
>>> sum_list([2, 4, 5])
11
```

Base case: empty list []; answer is 0 here
Recursive Case: [1, 2, 3, 4, 5]
Answer is 1 + whatever the sum of the rest of the list is

```
def sum_list(lst):
''' (list of int) -> int '''
     if lst == []:
          return 0
     return lst[0] + sum_list(lst[1:])
```

Don't do this:

```
def sum_list(lst):
     if len(lst) == 0:
          return 0
     if len(lst) == 1:
          return lst[0]
     if len(lst) == 2:
          return lst[0] + lst[1]
     return lst[0] + ..... recursion
```

Arbitrary nesting

```
>>> sum_list([1, [2, 3], 4, [5, [6]]])
```

Base case this time: a list with no nesting
[1, 2, 3]
Recursive case: list that has some nesting
[1, [2, 3]]

```
def sum_list(lst):
'''(list of int possibly with nesting) -> int

Return sum of lst. lst could be nested.
>>> sum_list([1, [2, 3]])
6
'''
     total = 0
     for item in lst:
          if isinstance(item, int):
               total = total + item
          else:      # item is a list
               total = total + sum_list(item)
     return total
```

Let's show that this function works on [3, 4, [1, 2]]
Already you know that it works for [1, 2]. Part of our base case; no nesting

total starts off as 0
3 is an int. add 3 to total
4 is an int. so add 4 to total
sum_list([1, 2]) is 3, so we add 3 to total.

Note: the notion of problem 'size' here is in terms of maximum amount of nesting in a list. e.g. [1, 2] is "smaller" than [3, 4, [1, 2]] because its nesting depth is 1 less.

Here's a version that doesn't use any loop:

```
def sum_list(lst):
     if lst == []:
          return 0
     if isinstance(lst[0], int):
          return lst[0] + sum_list(lst[1:])
     else:      # lst[0] is a list
          return sum_list(lst[0]) + sum_list(lst[1:])
```

If you ignore the 'else', this function is very similar to the "no nesting" solution. So it shouldn't surprise you that this function works on a list with no nesting. But it also works on lists with arbitrary nesting!

Example: [[1, 2, 3], 4, 5]
First element is [1, 2, 3]. Function will return 6 on this. Why? No nesting.
6 + sum_list([4, 5])
sum_list([4, 5]) gives 9; works fine because there is no nesting.
6+9 = 15

Question 1

Here is a slight change to the recursive solution of the binary codes example that we studied in lecture. Is this new version still correct?

```
def codes(r):
''' (int) -> list of str

Return all binary codes of length r?
'''
    if r == 0:
        return ['']
    small = codes(r-1)
    lst = []
    for item in small:
         lst.append('0' + item) # was item + '0' in class
         lst.append('1' + item) # was item + '1' in class
    return lst
```

Base case is still correct: there is one binary string of length 0, and it is the empty string

Suppose we have a solution for r = 2:
00 01 10 11

Let's look at what the function does on a call with r = 3.

- From 00, we get 000 and 100
- From 01, we get 001 and 101
- From 10, we get 010 and 110
- From 11, we get 011 and 111

These are all of the binary strings of length 3. They're in a different order than the version from lecture, but that's OK.

This is not a proof, but hopefully this helps convince you that the code is still correct.

Question 2

Consider the following code

```
def mystery(lst, k, value):
      if len(lst) == 0:
            return k <= 0
      if len[0] == value:
            return mystery(lst[1:], k-1, value)
      else:
            return mystery(lst[1:], k, value)
```

We don't even know the types of the parameters. Figure those out first.

If-condition:
If true, then we found an occurrence of value at the beginning of lst. We recurse on the list without the first value, and k - 1 instead of k.
If false, then we did not find an occurrence of value at the beginning of lst. We recurse, but here we don't decrement k.

So k is decremented each time we find an occurrence of value.

Base case: returns true if k <= 0.
The function returns True iff lst contains at least k occurrences of value.

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, JANUARY 30, 2017

LECTURE 13: RECURSION 4

Towers of Hanoi Example

Overview

- You have three pegs, and a bunch of discs
- They're stacked so that the largest disc is on the bottom, second-largest is on top of that, third-largest on top of the second one….
- Goal: Move all discs from the first peg to the third peg
- You can only move one disc at a time
- You can never put a bigger disc on a smaller disc
- First peg is where discs start, third peg is where they have to end up, and second peg is for temporary storage

Suppose that we have three disks on peg 1.
Peg 1: 3 2 1 (It will take 7 moves to solve this puzzle.)

Recursive formulation of how to solve this problem:
Suppose that we have a single disc to move from peg 1 to peg 3. Also peg2 and peg3 are empty. We have to move this disc without putting it on top of something smaller. just move disc from peg 1 to peg 3
Peg 1:
Peg 2:
Peg 3: 1

How do we solve the problem for 0 disks?
Do nothing.

Recursive Case: We have more than 1 disc to move from peg 1 to peg 3
Let n be the number of disks to move from peg1 to peg3

We have to think of a smaller subproblem with exactly the same structure

1. Here's a subproblem: move n-1 disks from peg1 to peg2
   This moves everything off peg1 except the bottom disc
2. Move the biggest disc from peg1 to peg3.
3. Move the n-1 disks from peg2 to peg3

```
#peg1 is starting; peg2 is temp storage; peg3 is destination
# n disks to move

def hanoi (n, peg1, peg2, peg3):
      if n==1:
            print('Move from', peg1, 'to', peg3)
            return
      hanoi (n-1, peg1, peg3, peg2)
      print ('Move from', peg1, 'to', peg3)
      hanoi (n-1, peg2, peg1, peg3)
```

This is a building block for A1.
Here: 3 pegs
Assignment 1: 4 pegs
How do you solve this problem using 4 pegs?

```
def rec_max(lst):
      '''(list of int) -> int
      Return max number in possible nested list of numbers.

      >>> rec_max([17,21,0])
      21
      >>> rec_max([17, [21,24],0])
      24
      '''

      nums = []
      for element in lst:
            if isinstance(element, int):
                  nums,append(element)
            else:
            nums.append(rec_max(element))
      return max(nums)
```

The recommended way to trace recursion...not how it happens inside of a computer.

```
rec_max([4, [5, 6], [7, [8, 9]]])
# What does rec-max returns on [5, 6]
>>> rec_max([5, 6])
6
>>> rec_max([8,9])
9
>>> rec_max([7, [8,9]])
```

```
nums = [7,9]
nums= [4, 6, 9]
max(nums) = 9
```

This is a bottom-up way of tracing recursive code.
Solve the smallest subproblems first
Whenever there's a recursive call on one of those subproblems, just fill-in the result

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
WEDNESDAY, FEBRUARY 8, 2017

LECTURE 14: RECURSION V

Recursion Function Example

```python
# Recall from last lecture
def rec_max(lst):
      ''' (list of int) -> int
      Return max number in possibly nested list of numbers.

      >>> rec_max([17, 21, 0])
      21
      >>> rec_max([17, [21, 24], 0])
      24
      '''

      nums = []
      for element in lst:
            if isinstance(element, int):
                  nums.append(element)
            else:
                  nums.append(rec_max(element))
      return max(nums)
```

# Process with Example
>>> rec_max([4, 3, [1, [5]]])
nums = [4, 3]
element = [1, [5]]

Function is on hold now...we have to figure out
rec_max([1, [5]])
nums = [1]
element = [5]
This call is now suspended... we have to figure out

rec_max([5])
nums = [5]

return max(nums) .... return 5
def rec(n):
    return 1 + rec(n-1)

rec(10)
1 + ... function is on hold, because we have to first figure out
rec (9)

rec(9) = 1 + ... function is on hold, waiting for rec(8)
rec(8) = 1 + ... on hold waiting for rec(7)
Eventually, you run out of memory. There is no termination and it keeps searching for a base case that it will never find.

Binary Search

Another example of recursion

How do you search a list?
CSC108

```
def search_list(lst,value):
      for element in lst:
            if element == value:
                  return True
      return False
```

Linear Search.
'4' in lst

[3, 5, 2, 1, 2, 3, 8], search for 8

Binary search is a lot faster than that

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, FEBRUARY 6, 2017

LECTURE 15: TREES

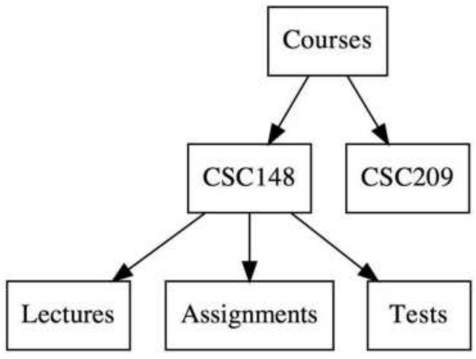
Motivating Trees

- A data structure is a way of organizing data
- Lists are linear structures
- They are linear in the sense that data is orders (ie. first piece of data is followed by second is followed by third.....)
- Underlying lists make sense for many applications:
  - Function calls in programs (Stacks)
  - A lineup in a bank (queue)
  - Event-handling based on timestamps (priority queue)
- Doesn't make sense to organize certain types of data into a linear structure
- Consider directories in a file system. They have a natural hierarchical structure that is difficult to represent linearly
- If we want to use a list, we might try storing the root directory at the first position and its subdirectories and files to its right

How would we know when the files of a subdirectory end and we are back up one level?
Ex. HTML structure or structure of a python program

Data Tree

- Set of nodes (often with values or labels), and directed edges that connects noes
- One specified node is a **root**
- Every node besides the root has exactly one parent



Example:

Root - Courses
Edges - The arrows
Node - Courses, CSC148, CSC209, Lectures, Assignments, Tests

Tree Terminology

Parent: a node is the parent of all nodes to which it has outgoing edges
- ex. Parent of Lectures is CSC148

Siblings: set of nodes that share a common parent
- ex. CSC148 and CSC209

Leaf: node that has no children (i.e. no outgoing edges)
- ex. Lectures, Assignments, Tests, CSC209

Internal node: non-leaf node; at least one child
- ex. Courses, CSC148

Path: sequence of nodes n1,n2,...,nk, where there is an edge from n1 to n2, n2 to n3, etc.
- ex. Courses → CSC148 → Lectures

Descendant: node n is a descendant of some other node p if there is a path from p to n
- ex. Everything is a descendant of courses in the tree

Subtree: a subtree of tree T is a tree whose root node r is a node in T, and which consists of all the descendants of r and the edges among them
- ex. The entire tree is a subtree, however lectures only can also be subtree

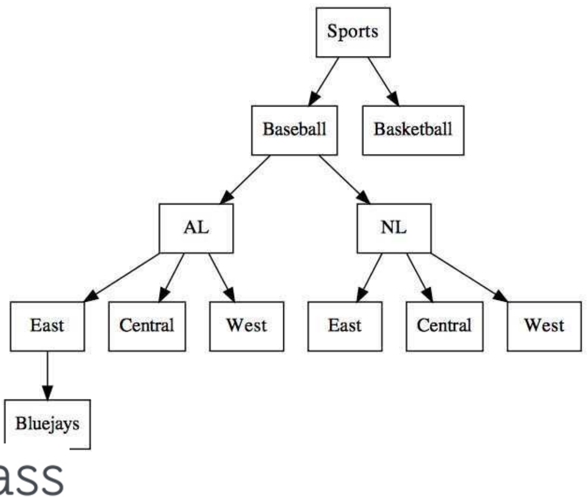
Branching Factor: maximum number of children of any node

Length of a path: number of edges on the path; Starting node and ending node and count the number of edges
- ex. Courses to Lectures: 2

Level (Depth): the level (or depth) of node n is the length of the path from the root node to n. The level of the root is 0
- ex. Depth of Courses is 0; depth of csc148 is 1; depth of lectures/assignments/tests is 2

Tree height: maximum of all node levels; The deepest depth will be height on the tree

Example



Height: 4
Branching Factor: 3
Depth of Baseball: 1
Length of path from Sports to AL: 2

Common Operations on Trees

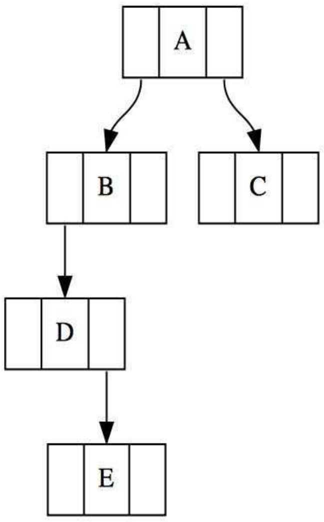
Here are operations we might want to do on trees:

- **Transverse -** Visit the nodes in some order and apply some operation to each node
- **Insert -** Add new nodes to your tree; inserting
- **Remove (Nodes)** - Delete nodes from your tree
- **Attach -** Attaching/Removing a subtree at a node
- **Remove (Subtree)** - Remove an entire subtree

Representing Binary Trees

- **Binary Tree -** Tree where each nodes has at most two children
- Represent binary trees in our programs in one of two ways:
  - List of lists
    - Uses list with nesting to assemble the tree
    - **`First`** element of the list contains the label of the root node
    - **`Second`** element is the list that represents the left subtree, or `None`
    - **`Third`** element is the list that represents the right subtree, or `None`
  - Nodes and references
    - More object oriented approach

List of Lists



```
['A',
    ['B', ['D', None, ['E', None, None]], None],
    ['C', None, None]]
```

Another Representation

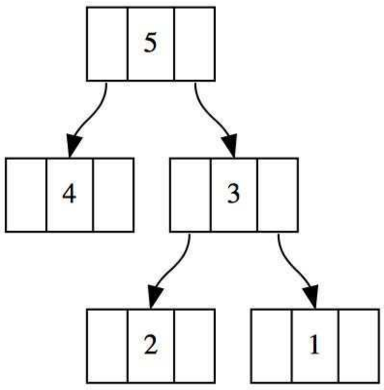
A → B
A → C

B → D
B → None (B's right side is empty)

D → None (D's left side is empty)
D → E

Another Example

```
[5, [4, None, None],
    [3, [2, None, None], [1, None, None]]]
```



Root of tree is 5

5's left subtree is just 4

5→ 4
5→ 3

3→ 2
3→ 1

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
WEDNESDAY, FEBRUARY 8, 2017

LECTURE 16: BINARY TREES

```
# A binary tree (BT) is None or a list of three elements

def binary_tree(value):
      ''' (value) ->  BT
      Return new BT with value as root and no children
      '''
      return [value, None, None]
def insert_left(bt, value):
      '''(BT, value) -> NoneType
      Insert value as the left node of the root of bt.
      '''
      if not bt:
            raise ValueError('cannot insert into empty tree')
      left_brach = bt.pop(1)
      bt.insert(1, [value, left_branch, None])
def insert_right(bt, value):
      ''' (BT, value) -> NoneType
      Insert value into the right subtree of t.
      '''
      if not bt:
            raise ValueError('cannot insert into empty tree')
      right_branch = pt.pop(2)
      bt.append([value, None, right_branch])
      # We are inserting into the right subtree
      # and making the old right subtree be the right subtree
      # of the new right subtree

      # [4, None, [10, None, [5, None, None]]]

def contains(bt, value):
      '''(BT, value) -> bool
      Return True iff value is in bt.

      >>> contains([4,[5, None, None], None], 5)
```

```
        True
        '''
        if not bt:
                return False
        # found at root, or in left, or in right
        return bt[0] == value or contains(bt[1], value) or
contains(bt[2], value)
```

```
>>> from ll import *
>>> t = binary_tree(10)
>>>t
[10, None, None]
>>> insert_left(t, 20)
>>> t
[10, [20, None, None], None]
>>> insert_left(t, 30)
>>> t
[10, [30,  [20, None, None], None], None]
```

Representation 2: Nodes and References

- Since the left and right children of a node are each roots of (sub)trees, we can model a tree as a recursive data structure
- Our tree objects will have attributes for the root value, left child and right child

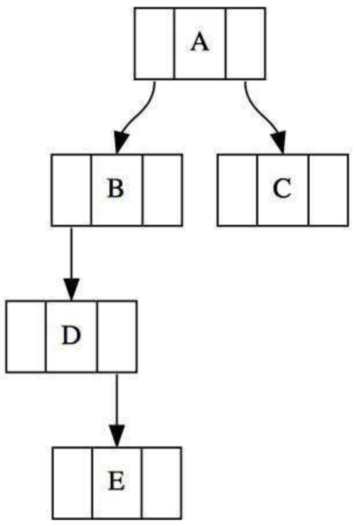
```
class BinaryTree:

        def __init__(self, value):
                self.key = value
                self.left = None
                self.right = None

        def insert_left():
                t = BinaryTree
                t.left = self.left
                self.left = t

        def contains(self, value):
        '''(BinaryTree, value) -> bool

        Return True iff value is in self
        '''
```

Tree Traversals

Traversal: Accessing each element of a structure

- We say we have visited an element when we have done something with it (eg. print, change, etc.)
- Lists have two obvious traversals: left to right and right to left
- Binary trees give us more ways to systematically visit each node
    - **Preorder**
        - Visit the root node, do a preorder traversal of the left subtree, and do a preorder traversal of the right subtree
        - 'pre': prefix, pre-med.... 'pre' means "before"
          Preface
          This tells you that the root goes before everything else
    - **Inorder**
        - Do an inorder traversal of the left subtree, visit the root node, and then do an inorder traversal of the right subtree
    - **Postorder**
        - Do a postorder traversal of the left subtree, do a postorder traversal of the right subtree, and visit the root node
        - 'post': postsecondary. post mortem, posthumous

Pre, in, and post tells you where the root shows up in the traversal order



Pre-Order: A, B, D, E, C
In-Order: D, E, B, A, C
Post-Order: E, D, B, C, A

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
FRIDAY, FEBRUARY 10, 2017

LECTURE 17: BINARY TREES 2

- **Preorder**
  - Visit the root node, do a preorder traversal of the left subtree, and do a preorder traversal of the right subtree
- **Inorder**
  - Do an inorder traversal of the left subtree, visit the root node, and then do an inorder traversal of the right subtree
- **Postorder**
  - Do a postorder traversal of the left subtree, do a postorder traversal of the right subtree, and visit the root node

```
class BinaryTree:
   def __init__(self, value):
      self.key = value
      self.left = None
      self.right = None

   def insert_left(self, value):
      if not self.left:
          self.left = BinaryTree(value)
      else:
          t = BinaryTree(value)
          t.left = self.left
          self.left = t

   def contains(self, value):
   '''Return True iff value appears as some key in tree.'''
      if self.key == value:
          return True
      if self.left:
          found = self.left.contains(value)
      if found:
          return True
      if self.right:
          found = self.right.contains(value)
```

```
        if found:
            return True
        return False

    def preorder(t):
    '''Return a list of t's keys in preorder.'''
        if not t:
            return []
            # non-empty tree
        return [t.key] + preorder(t.left) + preorder(t.right)
```

On the list of lists representation, write:

- preorder: return a list of node values in preorder
- inorder: return a list of node values in inorder
- postorder: return a list of node values in postorder

On the nodes and references representation, write:

- insert_right: add a value as the right child
- contains: return True iff the binary tree contains a given value

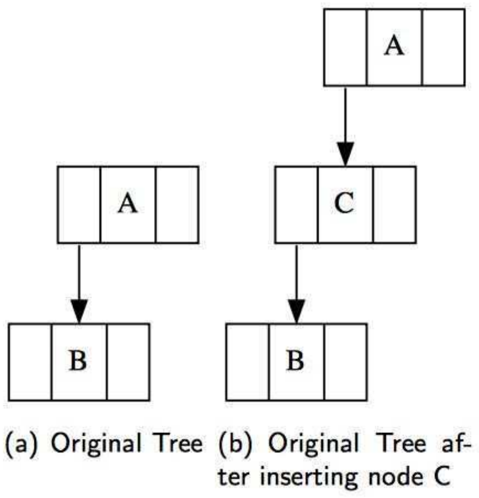


Figure: Inserting a Node Using Code on Previous Slide

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, FEBRUARY 13, 2017

LECTURE 18: BINARY TREES 3 + N-ARY TREES

Given a binary tree of height h, what is the maximum number of nodes in the tree?
Challenge: given a ternary ("branching factor at most 3") tree of height h, what is the maximum number of nodes in the tree?

Maximum number of nodes in a binary tree of height h.
height 0, max nodes is 1

height 1, max nodes is 3
Root 4, right child of root 5, left child root 6

height 2, max nodes is 7
height 3, max nodes is 15

N-ary Trees

- Recall: Binary Trees
  - List of Lists: Each list had an item, left list and right list
  - Nodes and References: each node had an item, left subtree, right subtree
- But remember that not all trees are binary trees!
  - Branching factor can be 3,4 etc.
- How do we handle arbitrary ('n-ary') trees in our code

Overview

- We will use a nodes and references representation
- left and right references are not going to work anymore
- Instead, we will use a list of subtrees

```
def __init__(self: 'Tree', value: object =None, children: list =None):
    """Create node with value and any number of children"""
    self.value = value
    if not children:
        self.children = []
    else:
        self.children = children[:]
```

Location of Children

- Our n-ary trees keep only the order of children, not their location
    - eg. below, you don't know if 2 is the first or second or some other child of t1

```
>>> t1 = Tree(4, [Tree(2)])
>>> t2 = Tree(5, [Tree(6), Tree(7)])

>>> from tree import *
>>> t1 = Tree(8)
>>> 51
Tree(8, [])
>>> t2 = Tree(10)
>>> t2
Tree(10, [])
>>> t3 = Tree(12)
>>> t3
Tree(12,[])
>>> t4 = Tree(20, [t1,t2,t3])
>>> t4
Tree(20, [Tree(8, []), Tree(10, []), Tree(12, [])]
```

N-ary Tree Functions

```
class Tree:
  """Tree ADT; nodes may have any number of children"""

  def __init__(self: 'Tree',
               value: object =None, children: list =None):
    """Create node with value and any number of children"""

    self.value = value
    if not children:
      self.children = []
    else:
      self.children = children[:]

  def __repr__(self: 'Tree') -> str:
    """Return representation of Tree as a string"""

    return ('Tree(' + repr(self.value) + ', ' +
            repr(self.children) + ')')

def arity(t: Tree) -> int:
  """Return maximum branching factor of t
  >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
  >>> tn3 = Tree(3, [Tree(6), Tree(7)])
  >>> tn1 = Tree(1, [tn2, tn3])
  >>> arity(tn1)
  4
  """
  if t.children == []:
    return 0
  most = len(t.children)
  for child in t.children:
      branching = arity(child)
      if branching > most:
          most = arity(child)
  return most

def count(t: Tree) -> int:
  """Return number of nodes in t

  >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
  >>> tn3 = Tree(3, [Tree(6), Tree(7)])
  >>> tn1 = Tree(1, [tn2, tn3])
  >>> count(tn1)
  9
  """
  if t.children == []:
      return 1
  total = 1
  for child in t.children:
      total = total + count(child)
  return total

def height(t: Tree) -> int:
  """Return length of longest path of t

  >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
  >>> tn3 = Tree(3, [Tree(6), Tree(7)])
  >>> tn1 = Tree(1, [tn2, tn3])
  >>> height(tn1)
  2
  """
  if t.children == []:
    return 0
  tallest = 0
  for child in t.children:
    tallest = max(tallest, height(child))
  return tallest + 1

def leaf_count(t: Tree) -> int:
  """Return number of leaves in t

  >>> tn2 = Tree(2, [Tree(4), Tree(4.5), Tree(5), Tree(5.75)])
  >>> tn3 = Tree(3, [Tree(6), Tree(7)])
  >>> tn1 = Tree(1, [tn2, tn3])
  >>> leaf_count(tn1)
  6
  """
  if not t.children:
    return 1
  total = 0
  for child in t.children:
    total = total + leaf_count(child)
  return total

if __name__ == '__main__':
  import doctest
  doctest.testmod()
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
WEDNESDAY, FEBRUARY 15, 2017

LECTURE 19: BINARY SEARCH TREES

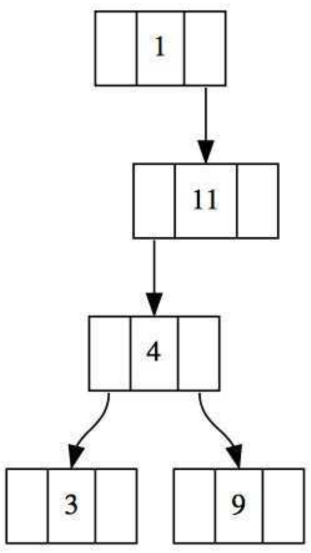
Motivating Binary Search Trees

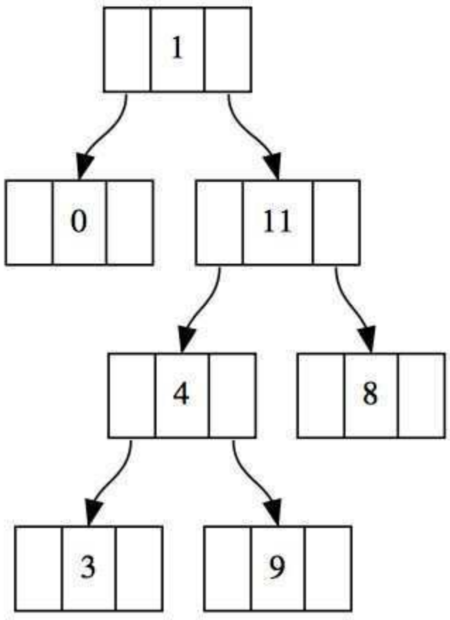
- We've seen examples of where a tree structure is more appropriate than a linear structure
    - e.g. directory hierarchy, representing relationships between items
- We will use binary search trees to allow for efficient searching of a collection of data
    - Don't confuse binary trees and binary search trees!
    - Binary tree: branching factor at most 2
    - Binary search trees: binary tree with extra constraint

What is a Binary Search Tree

- A Binary Search Tree (BST) is a binary tree in which
    - Every node has a value
    - Every node value is
        - Greater than the values of all nodes in its left subtree
        - Less than the values of all nodes in its right subtree
- This is called the **BST property**

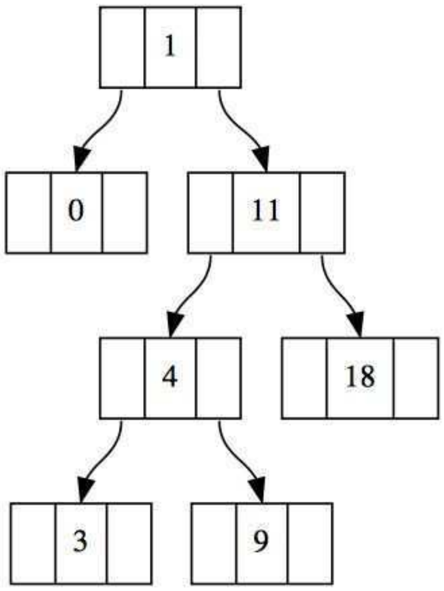
Example Binary Search Tree





Is this a BST?

- No
- The right subtree must be bigger than its parent node



Is this a subtree?

- Yes
- All right subtrees is greater than its parent

Searching a BST

- Suppose we want to know whether value v exists in a BST
- We compare v to the value r at the root
    - If v = r, then the value is found and we are done
    - If v < r, we proceed down the left subtree and repeat the process
    - If v > r, we proceed down the right subtree and repeat the process
- If we go off the tree in this process, then the value is bf not in the BST

Let's try this:
e.g. height h = 1, result is 3

Prove that a binary search tree of height h has at most 2^(h+1)-1 nodes.

Base case: h = 0.

- We have to prove that a BST of height 0 has at most 1 node.
- Proof: if you have more than one node, then the height must be at least 1.

Inductive case:

- Suppose h > 0.
- IH: for any height j < h, maximum number of nodes is 2^(j+1)-1

In a BST of height h, what's the maximum number of nodes?

- Root: 1
- Plus maximum number of nodes in left subtree
- Plus maximum number of nodes in right subtree

What is the maximum height of the left subtree? h-1
What is the maximum height of the right subtree? h-1

We can use IH on these two subtrees

- Left subtree has at most 2^(h)-1 nodes
- Right subtree also has at most 2^(h)-1 nodes

1+2^h-1+2^h-1
…
= 2*2^h-1
= 2^(h+1)-1
Exercise: prove the minimum number of nodes for a BST of height h.

Properties of BST Insertion

- Claim 1: time taken to search depends on the height of the BST.
  - True. Just like search
- Claim 2: when you insert some value v, it becomes a leaf of the BST.
  - True: insertion only happens when you fall off the tree
- Claim 3: different insertion orders determine the shape and height of the tree.
  - True
  -

Suppose that we insert in this order: 1, 2, 3, 4, 5, 6, 7, …
What will this BST look like?

- Root 1
- 2 is right child of 1
- 3 is right child of 2
- 4 is right child of 3
- 5 is right child of 4

- 6 is right child of 5
- ...

This will be a chain. Terrible performance on this tree.
Here's a better insertion order: 3, 6, 2, 5, 4, ... (This will be more balanced than before)

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
FRIDAY, FEBRUARY 17, 2017

LECTURE 20: BINARY SEARCH TREES 2

BTS Representation

- We have seen two ways to represent trees
  - List of lists
  - Nodes and references
- We'll use a form of nodes and references to represent a BST

We'll use a `BTNode` class to represent a node in the BST
We'll use a `BST` class to represent the tree itself

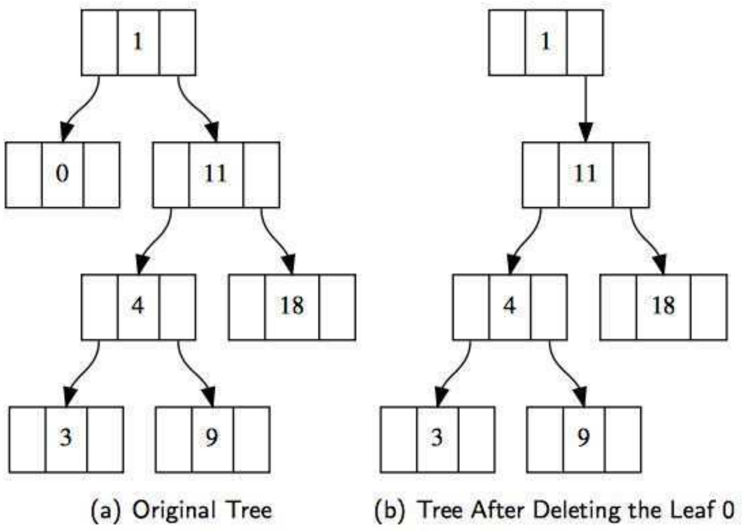
- The `BST` class has a `root` attribute that is
  - None when `BST` is empty
  - A reference to the root `BTNode` of the tree

Deleting a Node

- Deleting has more cases than insertion
- Depends on the where the node exists in the tree
- We must be able to delete the node without violating the BST property
- We will discuss how to delete
  - A leaf node
  - A node with one child
  - A node with two children

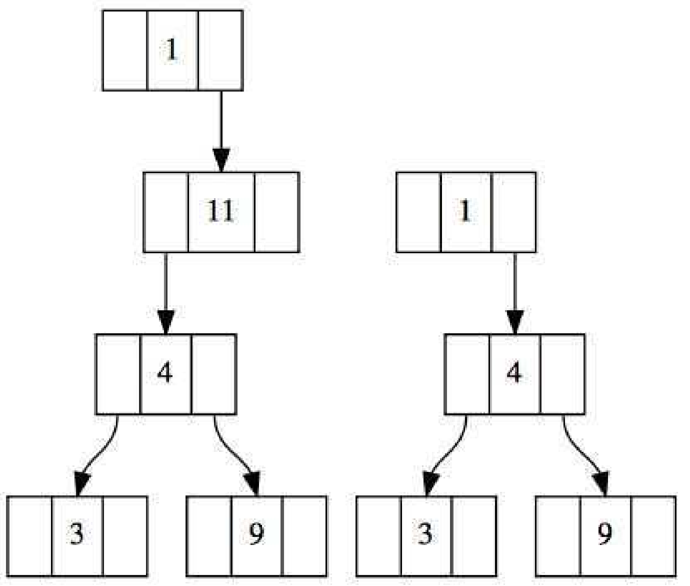
Deleting a Leaf

To delete a leaf, just remove it



Deleting a Node: One Child

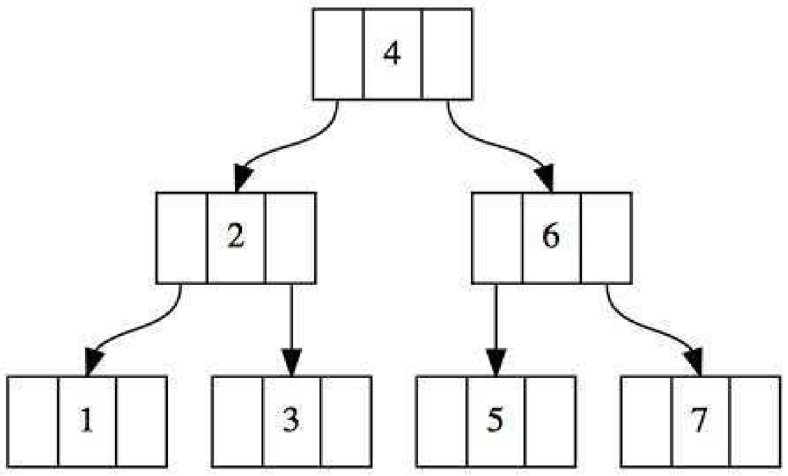
To delete a node with a single child, cut out that node



(a) Original Tree (b) Tree After Deleting the Node 11

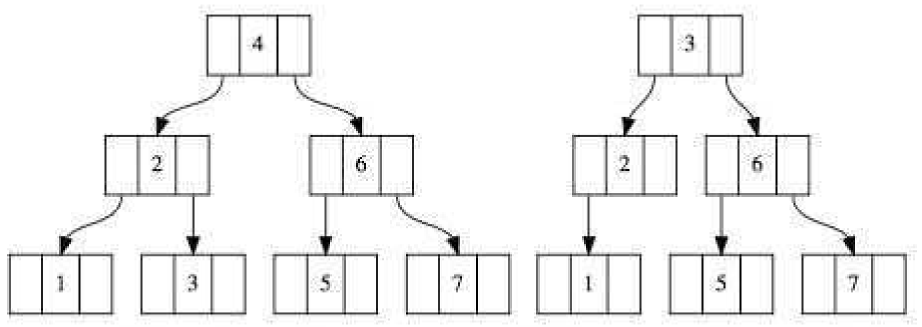
Deleting a Node: Two Children

When a node has two children, it may not be correct to move one of the children up eg. removing 4 in the example below



- To delete a node with two children, replace it by its predecessor
- This yields a new BST that cannot violate the BST property
- The predecessor of a node n with two children is the maximum node found in the left subtree of n. Why?
    - It cannot be in the right subtree (those are larger than n)
    - The tree rooted at n contains n and our proposed predecessor p
    - If n is the left child of its parent, its parent (and everything in its right subtree) is bigger than n

- If n is the right child of its parents, its parent (and everything in its left subtree) is smaller than p
- Continue this reasoning all the way up to the root



(a) Original Tree (b) Tree After Deleting the Node 4

Finding Maximum of Subtree

- To find the maximum of a subtree t, we keep traversing right children until we get to a node with no right child
- Proof: at each step, we reduce the portion of the tree that contains the maximum until we have one node remaining
- Since this node has no right child, we know how to remove it (i.e. we are in the case of deleting a leaf or a node with a single child)

BST with delete function Code

```
"""binary search tree ADT"""

from bt_node import BTNode


class BST:
    """Binary search tree."""

    def __init__(self: 'BST', root: BTNode=None) -> None:
        """Create BST with BTNode root."""
        self._root = root

    def __repr__(self: 'BST') -> str:
        """Represent this binary search tree."""
        return 'BST(' + repr(self._root) + ')'

    def insert(self: 'BST', data: object) -> None:
        """Insert data, if necessary, into this tree.
        >>> b = BST()
        >>> b.insert(8)
        >>> b.insert(4)
        >>> b.insert(2)
        >>> b.insert(6)
        >>> b.insert(12)
        >>> b.insert(14)
        >>> b.insert(10)
        >>> b
        BST(BTNode(8, BTNode(4, BTNode(2, None, None), BTNode(6, None,
None)),\
 BTNode(12, BTNode(10, None, None), BTNode(14, None, None))))
    """
        self._root = _insert(self._root, data)

    def find(self: 'BST', data: object) -> BTNode:
        """Return node containing data, otherwise return None."""
        return _find(self._root, data)

    def delete(self: 'BST', data: object) -> None:
        """Remove, if present, node containing data.

        >>> b = BST()
        >>> b.insert(8)
        >>> b.insert(4)
        >>> b.insert(2)
        >>> b.insert(6)
        >>> b.insert(12)
        >>> b.insert(14)
        >>> b.insert(10)
        >>> b.delete(12)
        >>> b
        BST(BTNode(8, BTNode(4, BTNode(2, None, None), BTNode(6, None,
None)),BTNode(10, None, BTNode(14, None, None))))
        >>> b.delete(14)
        >>> b
        BST(BTNode(8, BTNode(4, BTNode(2, None, None), BTNode(6, None,
None)),\
 BTNode(10, None, None)))
        """
        self._root = _delete(self._root, data)

    def height(self: 'BST') -> int:
        """Return height of this tree."""
        return _height(self._root)


def _insert(node: BTNode, data: object) -> BTNode:
    """Insert data in BST rooted at node, if necessary, and return
root."""
    return_node = node
    if not node:
        return_node = BTNode(data)
    elif data < node.data:
        node.left = _insert(node.left, data)
    elif data > node.data:
        node.right = _insert(node.right, data)
    else:  # nothing to do
        pass
    return return_node

def _find(node: BTNode, data: object):
    """Return the node containing data, or else None."""
    if not node or node.data == data:
        return node
    else:
        return (_find(node.left, data) if data < node.data
                else _find(node.right, data))

def _delete(node: BTNode, data: object) -> BTNode:
    """Delete, if exists, node with data and return resulting tree."""
    # Algorithm for _delete:
    # 1. If this node is None, return that
    # 2. If data is less than node.data, delete it from left child and
    #     return this node
    # 3. If data is more than node.data, delete it from right child
    #     and return this node
    # 4. If node with data has fewer than two children,
    #     and one is None, return the other one
    # 5. If node with data has two non-None children,
    #     replace data with that of its largest child in the left
subtree,
    #     delete that child, and return this node
    return_node = node
    if not node:
        pass
    elif data < node.data:
        node.left = _delete(node.left, data)
    elif data > node.data:
        node.right = _delete(node.right, data)
    elif not node.left:
        return_node = node.right
    elif not node.right:
        return_node = node.left
    else:
        node.data = _find_max(node.left).data
        node.left = _delete(node.left, node.data)
    return return_node


def _find_max(node: BTNode) -> BTNode:
    """Find and return maximal node, assume node is not None"""
    return _find_max(node.right) if node.right else node


def _height(node: BTNode) -> int:
    """Return height of tree rooted at node."""
    return 1 + max(_height(node.left), _height(node.right)) if node
else -1

if __name__ == '__main__':
    import doctest
    doctest.testmod()
    b = BST()
    b.insert(8)
    b.insert(4)
    b.insert(2)
    b.insert(6)
    b.insert(12)
    b.insert(14)
    print(b.height())
    print(b.find(4))
    print(b.find(7))
    print(str(b))
    b.delete(12)
    print(b)
    b.delete(14)
    print(b)
    print(b.height())
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, FEBRUARY 27, 2017

LECTURE 21: BINARY SEARCH TREES 3

Test 2

- Friday March 10, 2017 at 5:00PM
- Use pen; not pencil
- If your utorID starts with a to s, go to IB110
- If your utorID starts with t to z, go to IB245

Question 1

a) Consider each of the following orders of insertion into an empty BST. Which one results in the tree where searches are fastest?

A. 1, 2, 3, 4, 5, 6, 7
B. 7, 6, 5, 4, 3, 2, 1
C. 3, 2, 1, 7, 6, 5, 4

Solution:

Properties of BST Insertion

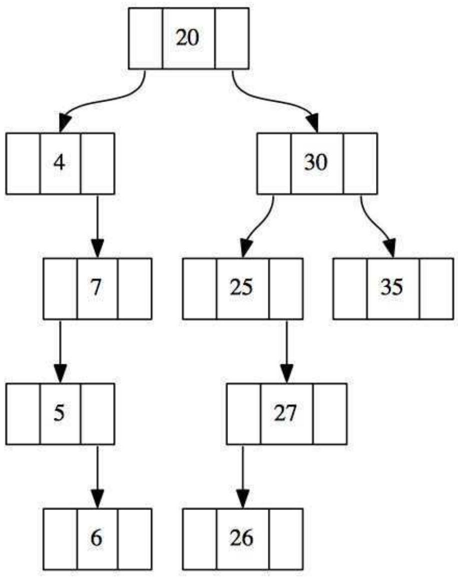
- Claim 1: time taken to search depends on the height of the BST.
  - True. Just like search
- Claim 2: when you insert some value v, it becomes a leaf of the BST.
  - True: insertion only happens when you fall off the tree
- Claim 3: different insertion orders determine the shape and height of the tree.
  - True...

Suppose that we insert in this order: 1, 2, 3, 4, 5, 6, 7, ... What will this BST look like?

Root 1
2 is right child of 1
3 is right child of 2
4 is right child of 3
5 is right child of 4
6 is right child of 5
...
This will be a chain. Terrible performance on this tree.

Here's a better insertion order: 3, 6, 2, 5, 4, ...
This will be more balanced than before.

b) What is the maximum height of a BST that contains 32 elements? What is the minimum height of a BST that contains 32 elements?



Question 2
Show the BST after deleting node 7. Show the BST after deleting node 20.

Question 3
The algorithm for deleting a node with two children that we discussed during lecture involved replacing the unwanted value with the maximum value from the left subtree.

a) What is the algorithm to find the maximum value of a BST?

```
def max_value(t):
      while t.right:
            t = t.right
      return t.value
```

b) Your friend proposes an alternative delete algorithm as follows: replace the unwanted value with the maximum value from the right subtree.
Does this still work? If yes, explain why. If not, explain why not and fix the algorithm.

Root is 10
left child of the root is 5
right child of root is 15
right child of 15 is 30
- algorithm would work but not a good example of showing why node won't work

Delete 10
- If I try to use this algorithm, what do I get?

Root is 30
Left child or root is 5
Right child of root is 15

Not a BST. Algorithm of using the max is right subtree doesn't work
What could be used from right subtree?

Min Value

Two algorithms for deleting a node with two children:

1. Replace it by maximum in the left subtree
2. Replace by minimum in right subtree

Question 4

Here is a Binary Tree Node class for this question. The empty tree will be represented as None.

```
class BTNode:

       """Binary Tree node."""
       def __init__(self: ’BTNode’, data: object,
                    left: ’BTNode’=None, right: ’BTNode’=None) ->
None:
         """Create BT node with data and children left and right."""
          self.data, self.left, self.right = data, left, right

def is_bst(t: BTNode) -> bool:
      """ Return True iff binary tree t follows the binary search tree
property. """
      if not t:  # empty binary tree
            return True
      left_ok = t.left is None or  t.data > t.left.data
# Property is True if left subtree is None or the data is greater than
the value on the left
      right_ok = t.right is None or t.data < t.right.data
#Check to see if the root in the right child is ok

# Root 10
# Left child of root is 5
# Right child of root is 20
# Left child of 5 is 6
# Can't just check the root, you need to check every node in the BST
(right and left properties have to be true)
# For the whole thing to be a binary tree
return left_ok and right_ok and is_bst(t.left) and is_bst(t.right)
# This solution is WRONG
```

To show that this algorithm is wrong:

1. Find a BST where it returns FAlse
    a. No counterexamples here.
2. Find a non-BST that returns True
   Root 10
   Left child of 10 is 5
   Right child of 5 is 50

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
WEDNESDAY, MARCH 1, 2017

LECTURE 22: LINKED LIST

Linked Lists

- We'll implement our own list ADT using linked lists
- We will use two classes for our implementation
    - Node: Allows us to string together elements of a list
    - LinkedList: stores a reference to the start of the list, plus other attributes to make some list operations faster

Node

Our node class allows us to make a chain of values.
We will notice that:

- We can quickly add to the beginning
- We cannot quickly add to the end

```
>>> a = Node(4)
>>> b = Node(5)
>>> c = Node(6)
>>> a
Node(4)
>>> b
Node(5)
>>> c
Node(6)
>>> a.next
>>> a
Node(4)
>>> a.next = b
>>> a
Node(4, Node(5))
>>> b
Node(5)
>>> c
Node(6)
>>> a.next = c
>>> a
```

```
Node(4, Node(6))
>>> a = Node(4)
>>> b = Node(5)
>>> c = Node(6)
>>> a.next = b
>>> a
Node(4, Node(5))
>>> a.next
Node(5)
>>> a.next.next(c)
>>> a
Node(4, Node(5, Node(6)))
>>> a.value
4
>>> a.next
Node(5, Node(6))
>>> a.next.value
5
>>> a.next.next
Node(6)
>>> a.next.next.value
6
>>> a.next.next.next
>>> a.next.next.value = 99
>>> a.next.next.value
99
```

```
class Node:
""" Node in a linked list"""
        def __init__(self: 'Node', value:object, next: 'Node' = None) -> None:
        """ Create Node self with data value and successor next."""
                self.value, self.next = value, next

        def __repr__ (self: 'Node') -> str:
        """ Return str representation of Node"""
                if not self.next:
                        return 'Node({})'.format(repr(self.value))
                else
                return 'Node({},{})'.format(repr(self.value), repr(self.next))

        def prepend(n, value):
        ''' (Node, int) -> Nonde
        Prepend value to n, and return the beginning of the new list.
        >>> a = Node(4)
        >>> b = Node(5)
        >>> a.next = b
        >>> prepend(a, 6)
        >>> print(a)
        Node(6, Node(4, Node(5))
        '''
        first_element = Node(value)
        first_element.
        return first_element
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
FRIDAY, MARCH 3, 2017

LECTURE 23: LINKED LISTS 2

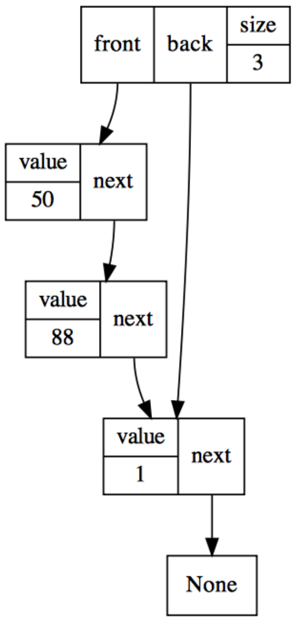
Test Content
Everything up to monday

How to make append fast:
Let's just keep a reference to the last node; then we don't have to find it… no loop

We're also going to add the number of elements in our list.
This makes it fast to find the length of the list.

Linked List Class

- Adding to the end of a chain of nodes requires us to find the last node
- This search is why adding to the end is slow! So, let's just keep a reference to that last node
- We will store in our LinkedList objects
    - Reference to front (first) node
    - Reference to back (last) node
    - Number of elements in linked list



Example
Three nodes, with references to first node, last node and size

How to write Linked List Methods
- It's often useful to think in terms of three specific cases:
    - What to do if the linked list is empty
    - What to do if it has one node
    - What to do if it has more than one node
- You might not have to write three separate cases for each method, but always test that these three cases are covered!

Linked List Methods

We'll work through writing some linked list methods.

- prepend: add a new value to the start of the linked list
- delete_front: remove the first value
- __contains__: return True iff the linked list contains the specified value
- __getitem__: return value at specified index

Code

```
from node import Node

class LinkedListError(Exception):
      pass

class LinkedList:
""" Collection of Nodes to form a linked list"""

      def __init__(self: 'LinkedList') -> None:
      """ Create empty LinkedList"""
            self.front, self.back, self.size = None, None, 0
            # Our linked list has these attributes
            # front: beginning node of list
            # back: last node in list
            # size: number of elements in list
            # Invariant: something that's always true; consistency
requirements
            # These have to be true at all times or list is corrupt
            # Scenarios
            # 1. If front is None, then back is None
            # 2. If front is None, size must be 0
            # 3. If front is None, size must be > 0
            # 4.  Size must be set to 0
            # 5. What do we know about back.next? Must be None
            # 6. If front is back, size is 0 or 1

def __repr__(self: 'LinkedList') -> str:
""" Return str representation of LinkedList"""
      return 'LinkedList({}).format(self.front)'

def append(self: 'LinkedList', value:object) -> None:
""" Append Node with value to the end of self."""
      if self.size == 0:  # list is empty
            self.size = 1
            self.front Node(Value)
            # self.back = Node(Value) is wrong because it creates a
second node
            # We only need one in our list
            self.back = self.front
      else: # non empty list
            # 3 -> 1 -> 2 -> 5 ->
            # append 10 to the list above
            # 1. Does self.front change? no
            # 2. Does self.back changee? Yes. Changes to the 10 node
            # 3. self.size change? 1
            self.size = self.size + 1
            new_node = Node(value)
            self.back.next = new_node
            self.back = new_node
            # self.back = self.back.next


      def prepend(self: 'LinkedList', value:object) -> None:
      """ Prepend Node with value to the beginning of self."""
            if self.size == 0:
                  self.front = Node(value)
                  self.back = self.front
                  self.size = 1
            else:
                  self.size = self.size + 1
                  new_node = Node(value)
                  new_node.next = self.front
                  self.front = new_node

      def length(self):
            return self.size

      #Not every operation is going to be this fast.

      def __getitem__ (self, index):
      ''' Return Node at given index in linked list.'''
      # loop required here

      # Tradeoffs between linked list and built-in Python list
      # 1. Linked list has fast prepend, fast append, but slow access to
anywhere else in the list
      # If you want element 50….slow
      # 2. Build-in list has slow prepend, fast append, fast access to
any element in the list
      # If you want element 50, you just do lst[50]....very fast
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, MARCH 6, 2017

LECTURE 24: LINKED LISTS 3

Here is a linked list node for the following questions. The empty linked list will be represented as None.

```
class Node:
      def __init__(self,value):
            self.value = value
            self.text = None
```

Question 1: Let's practice with this linked list node. Show the linked list that is created by the following code.

```
a = Node(4)
a.next = Node(5)
b = Node(6)
c = a.next

c.next = b
```

a: 4->5->6->None
b: 6->None
c: 5->6->None

Final linked list is 4->5->6->None

Code

```
from node import *
>>> a = Node(4)
>>> a
Node(4)
>>> a.next = Node(5)
>>> a
Node (4, Node(5))
```

```
>>> b = Node(6)
>>> b
Node(6)
>>> c = a.next
>>> c
Node(5)
>>> a.next
Node(5)
>>> c.next = b
>>> c
Node(5, Node(6))
>>> a
Node(4, Node(5, Node(6)))
```

Question 2: Explain in plain English the purpose of each of the following functions.

```
def mystery1 (lnk: Noe):   #purpose?
      while lnk and lnk.next:
            lnk.next = lnk.next.next
            lnk = lnk.next
```

# 1. Mystery1 deletes every node except for the third.
# 2. Deletes every other node in the linked lists
# 3. Delete every node except the last node
# 4. If odd number of nodes, delete all but last; if even number, delete all nodes
# 5. Makes first three nodes be the same as the third node

Function gets called with
1->2->3->4->5
1-> lnk 3->4->5
1->3->lnk 5

Good answer: Remove the second, fourth, sixth, etc. nodes from lnk

```
def f(s):
      s = 'q'
s = 'a'
f(s)
print(s) #a
```

```
def mystery2 (lnk: Node) -> Node:  #purpose?
      if not lnk:
            return
```

```
lnk = lnk.next
start = lnk
while lnk and lnk.next:
      lnk.next = lnk.next.next
      lnk = lnk.next
return start
```

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
WEDNESDAY, MARCH 8, 2017

LECTURE 25: ALGORITHM EFFICIENCY

For the same problem there can be multiple different algorithms that solve it

Anagrams

- Two words are anagrams if the letters of one word can be rearranged to be the letters of the other word.
- **horse** and **shore** are anagrams
- Are these?
    - bill, lib [NO]
    - deposit, topside [YES]
    - march, charm [YES]
    - sport, torts [NO]
    - traffic, traffic

Generating All Anagrams

- Given a word, suppose that we want to generate all anagrams of that word
- We have an English wordlist available
- x is an anagram of w if x is a rearrangement of w and is in the wordlist
- This eliminates "anagrams" that are not real words

Anagrams, Permutations Approach

What happens with larger words, generating all permutations? What are all permutations of march?

There are 5! = 120 potential anagrams.
marhc
armch
…
How many real anagrams are there?
march
charm

Out of 120 potential anagrams, 2 are real.
Looks like there's a lot of room for improvement here.

e.g. Find all anagrams of
mississauga
11!... huge number!

Anagrams, Signature Approach
We want to find all anagrams of 'march'
Signature of march is achmr

Look at all signatures of words in the dictionary…
adapt? aadpt. Not an anagram
…
child. cdhil. Not an anagram
…
charm. achmr. ANAGRAM
You're not going to generate millions or billions of useless permutations this way.

Important Lesson
If your problem size is small, who cares how you solve it.
Bad algorithms break down only when the problem gets large.

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, MARCH 13, 2017

LECTURE 27: ALGORITHM EFFICIENCY
(Lecture 26 was a test date)

Runtime Analysis

- Measuring elapsed runtime is one way to get a sense of an algorithm's efficiency
- However, runtime depends on the computer on which the program is run, the programming language being used, and lots of other factors
- Instead, we will characterize time efficiency in a way that
    - Is independent of the particular computer
    - Ignores small differences between algorithms
- A **step** is a basic unit of computation that can be carried out in a fixed amount of time
- We want to determine the number of steps that an algorithm takes as a function of its input size
- How we define input size depends a lot on the problem
    - e.g. for the anagram problem, the input size involves word length and number of words in the dictionary
- Typically the input size is the number of elements in the input
- For a given algorithm, we'll use the function T(n) to denote the number of steps that the algorithm takes on input size n

```
def max_segment_sum(L):
      '''(list of int) -> int
      Return maximum segment sum of L.
      '''
      max_so_far = 0
      for lower in range(len(L)):
            for upper in range(lower, len(L)):
                  sum = 0
                  for i in range (lower, upper + 1):
                        sum = sum + L[i]
                  max_so_far = max(max_so_far, sum)
      return max_so_far
```

Is this algorithm fast? NO
I want to characterize the runtime of this algorithm in terms of n, the length of the list.

We're going to add up the number of steps that it takes on a list of length n
We want an upper-bound for the number of steps
How many times will sum = sum + L[i] run?
lower loop executes n times.
For each iteration of lower, upper runs at most n times.
For each iteration of upper, i runs at most n times
sum = sum + L[i] will run at most n*n*n = n^3 times

Total: n^3 + 2 + n^2 + n^2
= n^3+2n^2+2

Does the 2 steps really matter?
Support n is 20
20^3+2*20^2+2 = 8402
Compare 8400 to 8402….2 doesn't matter at all

A segment of L is just a slice of L
Segment is same as slice
A segment is maximum if it has the largest sum of any segment

[5, 1, 2, 3]
maximum segment sum here is 11
5+1+2+3 = 11
If numbers are all positive, just take all of them as the maximum segment

[4, -3, 9, -8, 3]
If you take all elements, you get a sum of 5
Which segment is better than this?
[4, -3, 9] has a sum of 10

What is the maximum segment sum in [-1, -2, -3]
It's 0. Why? Which segment gives me sum of 0?
Empty segment []

Official Solution
Question: how many times is sum = sum + L[i] executed? (This is the same as asking for the number of iterations of the inner loop)

- The outer loop executes n times
- The middle loop is executed at most n times for each iteration of the outer loop
- So, the middle loop executes at most n_2 times
- The inner loop is executed at most n times for each iteration of the middle loop
- So, the inner loop executes at most n_3 times
- Similarly, sum = 0 and max_so_far = ... are executed at most n_2 times

Conclusion: an upper bound on the number of steps we execute is $n_3 + 2n_2$

Analyzing Segment-Sum Algorithm

Observation: as n increases, the n^3 term in n^3+2n^2 comes to dominate, and 2n^2 doesn't contribute much to T(n).

Big Oh

Even if we had $T(n)=n_3+4n_2$, or $T(n)=n_3+50n_2$, when n becomes large, the n_2 term will be much smaller than the n_3 term

- To measure the efficiency of an algorithm, we focus only on the approximate number of steps it takes
- We are not concerned with deriving an exact value for T(n), so we ignore its constant factors and nondominant terms
- Big Oh notation makes this idea precise

n^3+500n^2 = o(n^3)
500n^3+1000000n = o(n^3)

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, MARCH 13, 2017

LECTURE 28: ALGORITHM EFFICIENCY 2

Properties of Big Oh

- Constant Factors disappear
  - eg. 6n and n/2 depend on O(n)
- Lower order terms disappear
  - eg. n^5 + n^3 + 6n^2 is O(n^5)
- We can often just look at the loop structure of a program to determine its growth rate

Big Oh Approximations

- Big Oh gives us an upper bound on the time that our algorithm takes to execute
- It gives no guarantee that the bound is "close" to what actually happens
- It's equally valid to say that the segment-sum code is O(n3), O(n6), O(2n), etc.
- However, saying that it is O(n3) gives us the most useful information
- O(n3) is a "tight bound" (i.e. most accurate bound): there is no smaller function q for which it is still O(q)

We'll go from fastest to slowest types of algorithms:

O(1) is the fastest. constant-time algorithm

- Runtime doesn't depend at all on the problem size

Examples:

- Prepending to a linked list; does not depend on length of list
- Check if two integers are equal
- For loop with a constant

```
for i in range(1000):
      print ('hi')
```

O(log n) is a bit slower
Examples:

- binary search of a list
- guessing the number game. 50, 75, 87 .....

Slower than log is O(n). Linear Algorithm

Examples:

- Regular search through a Python list

```
for item in lst:
      if item == value:
            return True
```

- If you only have Node, not LinkedList….finding the last node is O(n)
- Find maximum value in a Python List

Slower than linear O(n log n)

Examples:

- Quick Sort
- Merge Sort

Slower than n log n is O(n^2). quadratic

Examples:

- Bubble sort
- insertion sort
- selection sort

Slower than n^2 is O(n^3)

Examples:

- Maximum segment-sum example from Monday

O(n^4)

O(n^5)

O(n^6)

Exponential Algorithms are slower than all of the above O(2^n)

These are not useful in practice unless problem size is very small… like 10 or 20 or 30

Even slower than 2^n is n^n

There is a strong dividing line between O(n^k) and O(2^n)

Examples

```
def bigoh2(n):
      sum = 0
      for i in range(1, n // 2):
            sum = sum + 1
      for j in range(1, n*n)
            sum = sum + 1
      print(sum)
```

```
# First loop does n/2 steps

def bigoh3(n):
        sum = 0
        if n%2 == 0:
                for j in range(1, n*n):
                        sum = sum+1
        else:
                for k in range(5, n+1):
                        sum = sum +k
        print(sum)

# If n is even, we do n^2 work
If n is odd, we do n work

def bigoh4(m,n):
        sum = 0
        for i in range(1, n+1):
                for j in range(1, m+1):
                sum = sum + 1
        print(sum)

# The i loop iterates n times
# For each iteration of the i loop, the j loop iterates m times
# Therefore, sum = sum + 1 runs n*m times
```

O(nm) or O(mn)

Suppose that you answered incorrectly:
O(m^2)

Why is this wrong?

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
MONDAY, MARCH 20, 2017

LECTURE 30: BIG OH 3 AND SORTING

```
def f3(t):
      if not t:
            return []
      return [t.key] + f3(t.left) + f3(t.right)
```

Big Oh worst-case time complexity for this function : o(...)
Unlike our previous examples last week, this current one is recursive.

Two questions:

1. What is the purpose of this function?
   f3 returns a list of keys from binary tree t in preorder
2. The code if o(...)

Selection Sort

```
def find_min(L, i):
      '''(list, int) -> int
      Return the index of the smallest item in L[i:]
      '''
      smallest_index = i
      for j in range(i+1, len(L)):
            if L[j] < L[smallest_index]:
                  smallest_index = j
      return smallest_index

def selection_sort(L):
      '''(list) -> NoneType)
      Sort the elements of L in non-descending order.
      '''
      for i in range(len(L) - 1):
            smallest_index = find_min(L, i)
            L[smallest_index], L[i] = L[i], L[smallest_index]
```

Selection sort: keep choosing the smallest element and swapping it into place

[1, 2, 3, 4, 7, 7, 8, 9]
Separates sorted from non-sorted

```
def insert(L, i):
      '''(list, int) -> NoneType
      Move L[i] to where it belongs in L[:i}.
      '''
      v = L[i]
      while i > 0 and L[i-1] > v:
            L[i] = L[i - 1]
            i -= 1
            L[i] = v

def insertion_sort(L):
      ''' (list) -> NoneType
      Sort the elements of L in non-descending order.
      '''
      for i in range(1, len(L)):
            insert(L, i)
```

Insertion Sort

```
def insert(L, i):
      '''(list, int) -> NoneType
      Move L[i] to where it belongs in L[:i}.
      '''
      v = L[i]
      while i > 0 and L[i-1] > v:
            L[i] = L[i - 1]
            i -= 1
            L[i] = v

def insertion_sort(L):
      ''' (list) -> NoneType
      Sort the elements of L in non-descending order.
      '''
      for i in range(1, len(L)):
            insert(L, i)
```

How it works: take next element and insert it into the proper place in the sorted part of the list.

[4, 10, _ 1, 2, 5, 0]
→ [1, 4, 10,_ 2, 5, 0]
→ [1, 2, 4, 10, _ 5, 0]
→ [1, 2, 4, 5, 10, _ 0]
→ [0, 1, 2, 4, 5, 10]

Insertion sort is o(n^2)

The Slow Sorts

- Selection sort and insertion sort are both O(n2)
  - They have an outer loop that runs n times
  - On each iteration, a function is called that takes at most n steps
- We'll discuss a much faster recursive sorting method called quicksort
- Interestingly, the worst-case running time of quicksort is still O(n2), though on average it is O(nlogn)

Properties of Quicksort

- Unlike the iterative sorts, quicksort is not an in-place sorting method
  - We use at least lg n additional stack space to carry out the recursion
- The algorithm works by choosing a pivot element, and partitioning the list so that elements smaller than the pivot are to its left and elements bigger than the pivot are to its right
- If we could then sort these two sublists, the original list would be entirely sorted
- We sort these sublists recursively

Partition Procedure

- We will look at a function to partition list lst[left:right+1] around pivot pivot
- As we proceed through the list, we will maintain three consecutive slices
  - Elements < pivot
  - Elements ≥ pivot
  - Unprocessed elements

```
def partition (lst, left, right, pivot):
      ''' (list, int, int, int) -> int
      Rearrange lst[left:right+1] so that elements >= pivot
      come after elements < pivot;
      return index of first element >= pivot
      '''

      i = left
      j = right
      while j <= right:
            if lst[j] < pivot:
                  lst[i], lst[j] = lst[j], lst[i]
```

```
        i += 1
    return i
```

Here's a sample list:
[1, 5, 2, 7, 4, 8, 3]
Let's use a pivot of 3

[1, 5, 2, 7, 4, 8, 3]
i = 0
j = 0
Swap lst[0] and lst[[0]
i = 1, j = 1
5 < pivot? No. No swap, just increment j
i = 1, j = 2
Perform swap; increment i and j
i = 2, j = 3
7 < pivot? No. increment j
i = 2, j = 4
4 < pivot? No. increment j
i = 2, j = 5
8 < pivot? No. increment j
i = 2, j = 6
3 < 3? No. increment j
i = 2, j = 7

[1, 2,   5, 7, 4, 8, 3]

What do you notice about the stuff on the left compared to the stuff on the right?

Compare
1, 2< all less than 3
to
5, 7, 4, 8, 3: all >= 3

We've split the list into two parts; stuff < pivot, stuff >= pivot
We have not yet sorted this list.

What have we done, then?
We've reduced the sorting problem to two smaller subproblems

Original problem: sort the whole list

Now we have two smaller subproblems:

1. Sort the stuff that's smaller than 3

2. Sort the stuff >= pivot

How do we solve these two subproblems?
Recursively call quicksort on left, recursively call quicksort on right

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
WEDNESDAY, MARCH 22, 2017

LECTURE 31: SORTING

Quick Reminders:

- Lab Today/Tomorrow - Reading Comprehension Test in first hour; optional 2nd hour

RECAP: Partition Procedure

- We look at a function to partition list lst[left:right+1] around pivot pivot
- As we proceed through the list, we will maintain three consecutive slices
  - Elements < pivot
  - Elements ≥ pivot
  - Unprocessed elements
- We're going to keep indices i and j
- Stuff to the left of i is less than the pivot
- Stuff from i up to but not including j is greater or equal to the pivot
- Stuff from j to the right is unprocessed

Partition Code

```
def partition(lst, left, right, pivot):
    '''(list, int, int, int) -> int
    Rearrange lst[left:right+1] so that elements >= pivot
    come after elements < pivot;
    return index of first element >= pivot
    '''
    i = left
    j = left
    while j <= right:
        if lst[j] < pivot:
            lst[i], lst[j] = lst[j], lst[i]
            i += 1
        j += 1
    return i
```

Running the code

```
>>> from partition import *
>>>  lst = [5, 1, 2, 7, 6, 8, 20]
>>> partition(lst, 0, len(lst)-1, 5)
2
>>> lst
[1, 2, 5, 7, 8, 8, 20]
>>> lst = [3, 2, 1, 4, 5, 6, 7, 20, 30]
8
>>> lst
[3, 2, 1, 4, 5, 6, 7, 20, 30]
>>>
```

Incorrect Code (but very close to being correct)

```
from partition import partition

def quicksort(lst, left, right):
      '''(list, int, int) -> NoneType
      Sort lst[left:right+1] in nondecreasing order.
      '''
      if left < right:
            pivot = lst[right]
            i = partition(lst, left, right-1, pivot)
            quicksort(lst, left, i-1)
            quicksort(lst, i, right)
```

Case that breaks this version of quicksort: [5, 4, 3, 2, 1]
pivot is 1
When we partition this list using pivot 1, what do we get?
Values smaller than 1: []
Values >= 1: [5, 4, 3, 2, 1]

Good Code

```
from partition import partition

def quicksort(lst, left, right):
      '''(list, int, int) -> NoneType
      Sort lst[left:right+1] in nondecreasing order.
      '''
      if left < right:
            pivot = lst[right]
            i = partition(lst, left, right-1, pivot)
            lst[i], lst[right] = lst[right], lst[i]
           quicksort(lst, left, i-1)
            quicksort(lst, i, right)
```

Quicksort is fast when the list is randomly-ordered.
However, if the list is sorted, quicksort might:

1. Take a huge amount of time
2. Crash

Why?

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
What's our pivot? 10
We recurse on [1, 2, 3, 4, 5, 6, 7, 8, 9]....everything except pivot 10

Next pivot? 9
We recurse on [1, 2, 3, 4, 5, 6, 7, 8]........everything except pivot 9

How many recursive calls will we make? len(lst)

If len(list) == 10; no problem
But if len(list) == 2000… maximum recursion depth error

How do we fix this?

- Choose a smarter pivot
- Don't just use the rightmost element

What should be my rule for choosing the pivot? Choose leftmost element?

- No
- Exactly the same problem as the rightmost

How about choosing middle element as pivot?
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] is definitely fixed now.

Unfortunately, this pivot-choice can be broken too:
[1, 2, 3, 4, 10, 6, 7, 8, 9, 5]
You'll take the pivot 10….that's the worse case.

How about the average of all elements?
[1, 2, 3, 4, 10, 6, 7, 8, 9, 5]
The average is ~5. That's a good pivot
[1, 2, 3, 4, 5, 6, 7, 8, 9, 9999999999]
Now the average is a huge number….terrible pivot again

Take three elements and take the middle one

CSC148H5S - INTRODUCTION TO COMPUTER SCIENCE (WINTER 2017)
UNIVERSITY OF TORONTO MISSISSAUGA
PROFESSOR DANIEL ZINGARO

LECTURE NOTES
JONATHAN HO
FRIDAY, FEBRUARY 10, 2017

LECTURE 32: QUICKSORT AND MERGESORT

1. **Recall our quicksort partition function:**

```
def partition(lst, left, right, pivot):
''' (list, int, int, int) -> int

Rearrange lst[left:right+1] so that elements >= pivot
come after elements < pivot;
return index of first element >= pivot
'''
      i = left
      j= left
      while j <= right
            if lst[j] < pivot:
                  lst[i], lst[j] = lst[j], lst[i]
                  i +=1
            j += 1
      return i
```

a) What is printed by the following code?

```
lst = [4, 8, 1, 3, 5, 9, 1]
i = partition(lst, 0, 6, 3)
print(i, lst)
```

lst = [1, 8, 4, 3, 5, 9, 1]
i= 1
j = 3

b) Consider the following code

```
lst = [4, 8, 1, 3, 5, 9, 1]
i = partition(lst, 0, 6, ______)
print(i)
```

Fill in the blank with a pivot that will cause 0 to be printed.

What should I choose for pivot so that 0 is printed? 1? 9? 10? 0?
To return a value of 0, i is never allowed to increase.
i can never change.
Every value has to be >= pivot for i not to change.
1 or 0

Now, fill in the blank with a pivot that will cause 7 to be printed?

2. Recall our quicksort function:
from partition import partition

```
def quicksort(lst, left, right):
      pivot = lst[right]
      i = partition(lst, left, right - 1, pivot(
      lst[i], lst[right] = lst[right], lst[i]
      quicksort(st, left, i-1)
      quicksort(lst, i+1, right)
```

a) What kind of performance do we get when sorting a list that is reverse-sorted?

b) Consider this alternative implementation of quicksort. Does it make the reverse-sorted case better or worse?

```
from partition import partition

def quicksort(lst, left, right):
      '''(list, int, int) -> NoneType
      Sort lst[left:right+1] in nondereasing order
      '''
      if left < right:
            mid = (left + right) //2
            lst[mid], lst[right] = lst[right], lst[mid]
            pivot = lst[right]
            i = partition(lst, left, right - 1, pivot)
            lst[i], lst[right] = lst[right], lst[i]
            quicksort(lst, left, i - 1)
            quicksort(lst, i+1, right)
```

Merge Sort

- Merge sort is always O(n log n), even in the worst case
- It works by
  - Recursively sorting the first half of the list
  - Recursively sorting the second half of the list

- Merging the two halves into a newly sorted list
- At each level of recursion, the merging takes a total of n steps, and there are log n levels before we get to the base case
- Ther merging requires an auxiliary list (not required in merge sort)

```
def merge(list1: list, list2: list) -> list:
      '''Return merge of sorted list1 and list2'''

      lst = []
      i = 0
      j = 0
      while i < len(list1) and j < len(list2):
            if lst1[i] < list2[j]:
                  lst.append(list1[i])
                  i = i+1
            else:
                  lst.append(lst2[j])
                  j= j+1

      lst.extend(lst1[i:])
      lst.extend(lst2[j:])
      return lst
```

lst1 = [2, 4, 6, 10, 20, 30]
lst2 = [1, 3, 5, 7]

Merge these two into a combined list:
[1, 2, 3, 4, 5, 6, 7, 10, 20, 30]

This merge helper function better be fast. This function is O(n).
Single while-loop, goes through every element of lst1 and lst2 once.

It's O(n), just like partition.

```
from merge import merge

def mergesort(lst: list) -> list:
      '''Return a sorted copy of lst'''
      if len(lst) <= 1:
            return lst[:]
      mid = len(lst) // 2
      left = lst[:mid]
      right = lst [mid:]
      left_s = mergesort(left)
      right_s = mergesort(right)
```

```
    # left_s and right_s are
    return merge(left_s, right_s)
```

Compare and Contrast quicksort and mergesort

1. Mergesort is always O(n log n); quicksort is usually O(n log n) but could be O(n^2) is worst-case.
2. Mergesort uses more memory than quicksort
   We're making copies of the original list.
   The merge helper function wastes memory by copying the smaller lists into a bigger list.