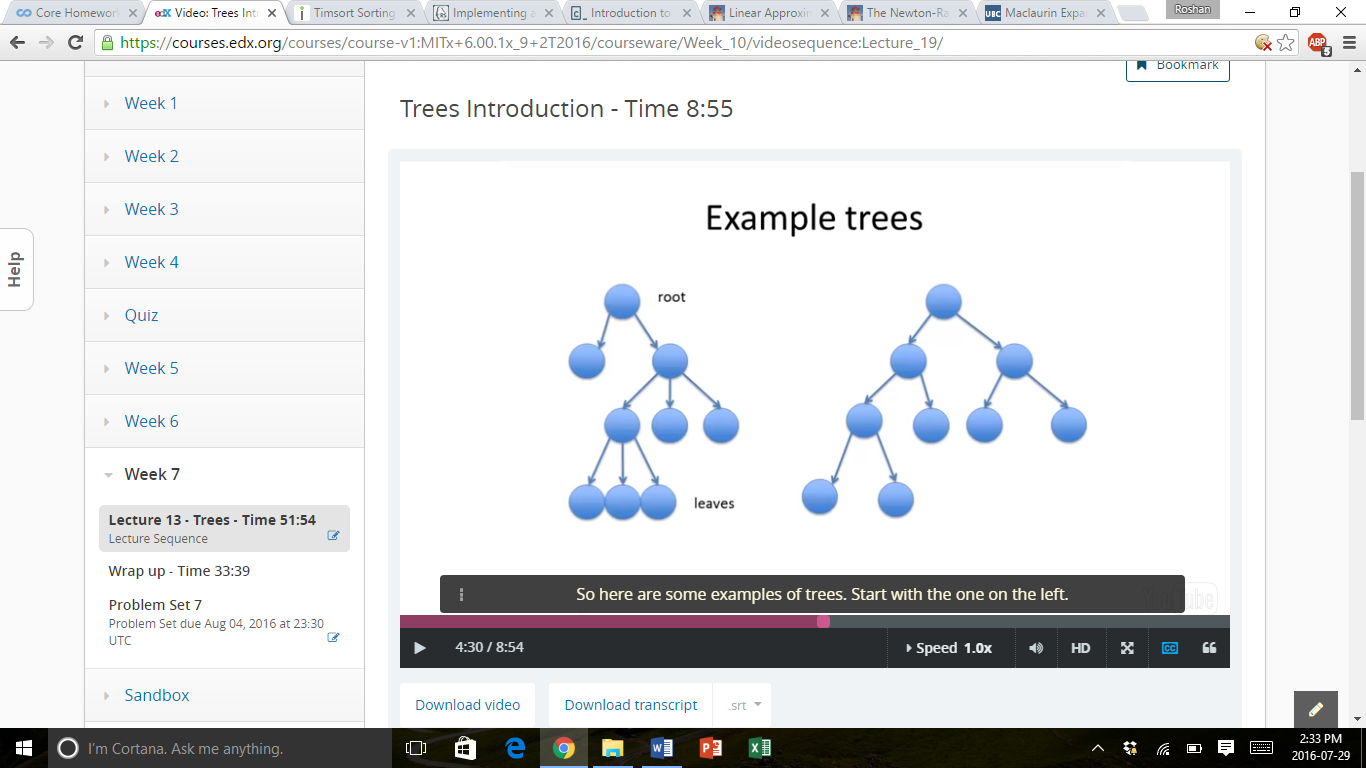
**MIT – 6.00.1x: Introduction to Computer Science and Programming**

**WEEK 7**

**Lecture 13: Trees**

Part 1: Trees Introduction

* Tree Definition
* A tree consists of one or more nodes.
  + - A node typically has a value associated with it.
* Nodes are connected by branches of the tree.
* A tree starts with a root node. Also, except for leaves, each node has one or more children.
  + - We refer to a node that has a child the parent node.
* In simple trees, no child has more than one parent, but the generalization (often called a graph) is also very useful.
* Example Trees



* Binary Trees
  + A binary tree is a special version of a tree, where each node has at most 2 children.
  + Binary trees are very useful when storing and searching ordered data, or when determining the best decision to make in solving many classes of problems – decision trees.
* Binary Tree Class

class BinaryTree(object):

def \_\_init\_\_(self, value):

self.value = value

self.leftBranch = None

self.rightBranch = None

self.parent = None

def \_\_str\_\_(self):

return self.value

def setLeftBranch(self, node):

self.leftBranch = node

def setRightBranch(self, node):

self.rightBranch = node

def setParent(self, parent):

self.parent = parent

def getValue(self):

return self.value

def getLeftBranch(self):

return self.leftBranch

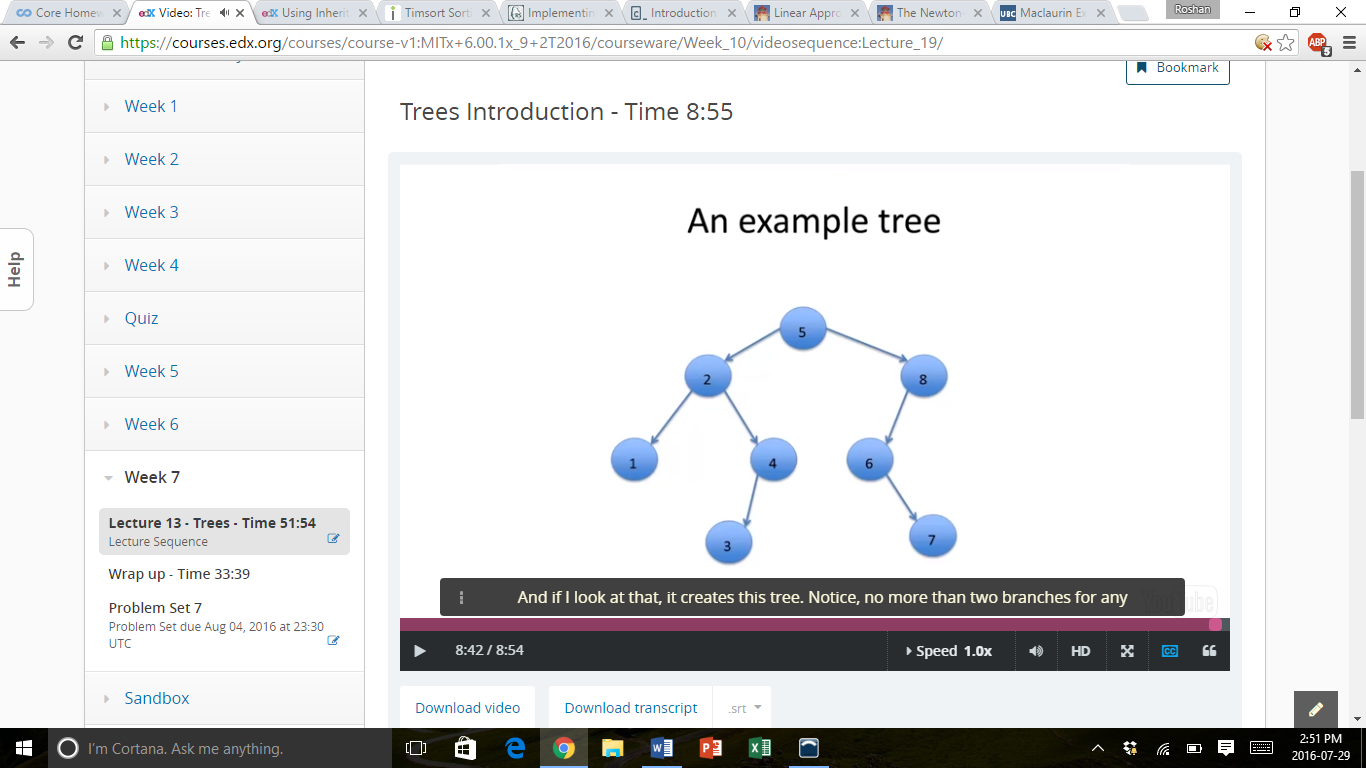
def getRightBranch(self):

return self.rightBranch

def getParent(self):

return self.parent

* Constructing an Example Tree

n1 = BinaryTree(1)

n2 = BinaryTree(2)

n4 = BinaryTree(4)

n5 = BinaryTree(5)

n6 = BinaryTree(6)

n7 = BinaryTree(7)

n8 = BinaryTree(8)

n5.setLeftBranch(n2)

n2.setParent(n5)

n5.setRightBranch(n8)

n8.setParent(n5)

n2.setLeftBranch(n1)

n1.setParent(n2)

n2.setRightBranch(n4)

n4.setParent(n2)

n8.setLeftBranch(n6)

n6.setParent(n8)

n6.setightBranch(n7)

n7.setParent(n6)

Part 2: Searching a Tree

* Searching a Tree
  + Imagine that we want to examine a tree.
    - To determine if an element is present.
    - To find a path to a particular solution.
    - To make a series of decisions to reach some objective.
  + Depth First Search
    - Start with the root.
    - At any node, if we haven’t reached our objective, take the left branch first.
    - When we get to a leaf, backtrack to the first decision point and take the right branch.
  + Breadth First Search
    - Start with the root.
    - Then proceed to each child at the next level, in order.
    - Continue until the objective is reached.
* Depth First Search for Containment
  + The idea is to keep a data structure (called a stack) that holds nodes that still need to be explored.
  + Use an evaluation function to determine when the objective has been reached (i.e. for containment, whether the value of the node is equal to the desired value).
  + Start with the root node. Then add children, if any, to the front of the data structure, with the left branch first. Continue in this manner.

Part 3: Decision Trees

* Decision Trees
  + A decision tree is a special type of binary tree (though it could be a more general tree with multiple children).
  + At each node, a decision is made, with a positive decision taking the left branch, and a negative decision taking the right branch.
  + When we reach a leaf that satisfies some goal, the path back to the root node defines the solution to the problem captured by the tree.
* Building a Decision Tree
  + One way to approach decision trees is to construct an actual tree and then search it.
  + An alternative is to implicitly build the tree and use it only as we need it.
  + As an example, we will build a decision tree for a knapsack problem.
* The Knapsack Problem
  + Suppose we are given a set of objects, each with a value and a weight.
  + We have a finite sized knapsack, into which we want to store some of the items.
  + We want to store the items that have the most value, subject to the constraint that there is a limit to the cumulative weight that will be allowed.
* Building a Decision Tree
  + For the knapsack problem, we can build a decision tree as follows.
    - At the root level, we decide whether to include the first element (left branch), or not (right branch).
    - At the *n*th level, we make the same decision for the *n*th element in the tree.
    - By keeping track of what we have included so far, and what we have left to consider, we can generate a binary tree of decisions.

Part 4: Implicit Search

* Decision Trees
  + Depth first search (DFS) and breadth first search (BFS) still search the same number of nodes, it is simply that the order they search the nodes in is different.
  + If we are willing to settle for “good enough”, then there is a difference in the work done by the two search methods.
* Searching an Implicit Tree
  + Our approach is inefficient, as it constructs the entire decision tree and then searches through it.
  + An alternative is only to generate the nodes of the tree as they are needed.
  + Here is an example for the case of a knapsack problem, but the same idea could also be captured in other search problems.

Part 5: Overgrown Trees

* Searching These “Overgrown Trees”
  + What happens if we run depth first search on this kind of tree?
    - An infinite loop may occur when the item is present, and an infinite loop will always occur when the item is not present.
  + What happens if we run breadth first search on this kind of tree?
    - Inefficient as it has to repeatedly go through the same nodes. However, it works when the item is present, but an infinite loop will always occur if the item is not present.

**Lecture 14: Wrap Up**

Part 1: Computational Thinking

* What do Computer Scientists Do?
  + They think computationally
    - Abstractions, decompositions, algorithms, automated execution
  + Computational thinking will be a fundamental skill used by everyone eventually.
* Computational Thinking: the Process
  + Identify or invent useful abstractions.
    - Suppressing details, formulating interfaces
  + Formulate a solution to a problem as a computational experiment using abstractions.
  + Design and construct a sufficiently efficient implementation of this experiment.
  + Validate the experimental setup (i.e. debug it).
  + Run experiment.
  + Evaluate results of experiment.
  + Repeat as needed.
* The Three A’s of Computational Thinking
  + Abstraction
    - Choosing the right abstractions.
    - Operating in terms of multiple layers of abstraction simultaneously.
    - Defining the relationship between the layers.
  + Automation
    - Think in terms of mechanizing our abstractions.
    - Mechanization is possible.
* Because we have precise and exacting notations and models.
  + Algorithms
    - Language for describing automated processes.
    - Also allows for abstraction of details.
* Example of Computational Thinking
  + How difficult is this problem and how can I best solve it?
    - Theoretical computer science gives precise meaning to these and related questions.
  + Thinking recursively
    - Reformulating a difficult problem into one we know how to solve.
    - Reduction, embedding, transformation, simulation.

Part 2: Where Have You Been?

* Where Have You Been?
  + Four major topics (and a language)
    - Learning a language for expressing computations – Python
    - Learning about the process of writing and debugging a program – Be systematic
    - Learning to estimate the computational complexity
    - Learning about the process of moving from a problem statement to a computational formulation of a method for solving the problem – Use abstraction
    - Learning a basic set of recipes – Algorithms
* Why Python?
  + Relatively easy to learn and use
    - Simple syntax
    - Interpretive, which makes debugging easier
    - Don’t have to worry about memory management
  + Modern
    - Supports currently stylish mode of programming, object-oriented programming
  + Increasingly Popular
    - Used in an increasing number of subjects at MIT and elsewhere
    - Increasing use in industry
    - Large and ever growing set of libraries
* Writing, Testing, and Debugging Programs
  + Take it one step at a time
    - Understand problem
    - Think about overall structure and algorithms independently of expression in programming language
    - Break into smaller parts
    - Identify useful abstractions (data and functional)
    - Code and unit test a part at a time
    - First functionality, then efficiency
    - Start with pseudo code
  + Be Systematic
    - When debugging, think of the scientific method
    - Ask yourself why the program did what it did, not why it didn’t do what you wanted it to do
* Estimating Complexity
  + Big O notation – orders of growth
    - Exponential, Polynomial, Log-Linear, Linear, Logarithmic, Constant
  + Recognizing common patterns of computation
  + Learning to map problems into templates of solutions
    - Bisection search, tree search, etc.
  + Some problems are inherently expensive to solve
* From Problem Statement to Computation
  + Break the problem into a series of smaller problems
  + Try and relate problem to a problem you or somebody else has already solved
    - E.g. can it be viewed as a knapsack problem or an optimization problem, more generally?
  + Think about what kind of output you might like to see
  + Think about how to approximate solutions
    - Solve a simpler problem
    - Find a series of solutions that approaches (but may never reach) a perfect answer
* Algorithms
  + Kinds of Algorithms
    - Exhaustive enumeration, Guess and check, Successive approximation, Greedy algorithms, Divide and conquer, Decision trees
  + Specific Algorithms
    - E.g. Binary search, Selection sort, Merge sort, DFS, BFS

Part 3: Where are You Headed?

* Where are You Headed?
  + There are other CS courses for which you are now prepared.
    - 6.00.2x – optimization, modelling, simulation
    - Introductory algorithms and data structures
    - Introduction to artificial intelligence
    - Software engineering
    - Computer architecture