Graphs & Trees

Intro2CS – week 8-9
General Graphs

• Nodes (Objects) can have more than one reference.
• Example: think of implementing a Class “Person”
• Each Person has a list of friends, which are also of the same “Person” class.
• A “Social Network”

class Person:
    def __init__(self,name):
        self.__name = name
        self.__list_of_friends=[]

    def add_friend(self,other):
        self.__list_of_friends.append(other)

    ...
Graphs

• We generally think of the structure of these as a general (directed) Graph $G=(V,E)$
• $V$ is called the set of Nodes or Vertices
• $E \subseteq V \times V$ is called the set of Links, or Edges

$(v_1, v_2) \in E$ if and only if object $v_1 \in V$ has a link to $v_2 \in V$
Paths

- **A Path** in the graph (of length n) is a sequence of vertices $P = (v_1, v_2, \ldots, v_n)$ such that
  - $\forall i \in \{1, 2, \ldots, n\}$ $v_i \in V$
  - $\forall i \in \{1, 2, \ldots, n-1\}$ $(v_i, v_{i+1}) \in E$
Traversing general graphs

• Suppose we wish to visit all nodes in a graph and print their data (only once!)

• How do we do this?
```python
class Node:
    def __init__(self, data=None):
        self.data = data
        self.neighbors = set()

    def add_edge(self, other):
        self.neighbors.add(other)
```

---

```
First attempt

```python
def visit_all_from(node):
    print(node.data)
    for neighbor in node.neighbors:
        visit_all_from(neighbor)
```
What went wrong?
What if we had a loop in the graph?
Fixing the problem

```python
def visit_all_from(node):
    visited = set()
    _visit_helper(node, visited)

def _visit_helper(node, visited):
    if node in visited:
        return

    visited.add(node)
    print(node.data)
    for neighbor in node.neighbors:
        _visit_helper(neighbor, visited)
```
nodes = [Node(i) for i in range(5)]

nodes[0].add_edge(nodes[1])
nodes[0].add_edge(nodes[2])
nodes[1].add_edge(nodes[2])
nodes[2].add_edge(nodes[3])

print("*** From Node 0:")
visit_all_from(nodes[0])
print("\n*** From Node 4:")
visit_all_from(nodes[4])
Trees

- Directed Rooted Trees are one particularly useful class of Graphs.
  
- They have a special node called “the root”

- There is exactly one path from the root to every other node in the tree.
  
  – No Cycles!
Tree Terminology

• A node that is directly linked from \( v \) is often called a \textit{child} of \( v \) (and \( v \) is called the \textit{parent})
  – Example: 6,7 are children of 3

• A Node that has no outgoing links is called a \textit{Leaf}.
  – Example: 4,6,7,8 are all leaves
Tree Terminology

• The height of a node is the length of its path from the root (the root is of height 0).
• Example: Node 5 has height 2.

• The height of the tree is the length of the longest path.
• Example: the tree here has height 3 (due to the path to node 8)
Example: Trees that represent expressions

• One thing to naturally represent with trees is mathematical expressions:

\[(5 + 8) \times (6 + (2 / 7))\]

• Leaves are all literals, internal nodes are operators
• Notice that the order of the children matters
In Python

```python
class TreeNode:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

expr1 = TreeNode("+",
                 TreeNode("*", TreeNode("3"), TreeNode("4")),
                 TreeNode("7"))
```
Computing the value of an expression

```python
def compute(root):
    if root.data == "+":
        return compute(root.left)+compute(root.right)
    elif root.data == "*":
        return compute(root.left)*compute(root.right)
    elif root.data == "/":
        return compute(root.left)/compute(root.right)
    elif root.data == "-":
        return compute(root.left)-compute(root.right)
    else:
        return float(root.data)
```

```python
expr1 = TreeNode("+",
                TreeNode("*", TreeNode("3"),TreeNode("4")),
                TreeNode("7"))
print(compute(expr1))
```
def tree_to_expr(tree_root):
    if tree_root.left:
        return "(" + tree_to_expr(tree_root.left) + \
            tree_root.data + \\n            tree_to_expr(tree_root.right) + ")"
    else:
        return tree_root.data

expr1 = TreeNode("+",
            TreeNode("*", TreeNode("3"), TreeNode("4")),
            TreeNode("7"))
print(tree_to_expr(expr1))
Polish Notation

- Polish notation (also called prefix notation) is just a different way to write mathematical expressions.
- The operator is always written to the left.
- Instead of $(2+3)$ write: $(+ 2 3)$
- We never need parentheses when writing this way. Order of operations is always well defined:

\[
+ \times 3 4 - 2 5 \\
(+ (\times 3 4) (- 2 5))
\]

In "regular" notation: $(3 \times 4) + (2-5)$
```python
def tree_to_polish(tree_root):
    result = tree_root.data
    if tree_root.left:
        result += " " + tree_to_polish(tree_root.left) + \
        " " + tree_to_polish(tree_root.right)
    return result

expr1 = TreeNode("+",
    TreeNode("*", TreeNode("3"), TreeNode("4")),
    TreeNode("7"))
print(tree_to_expr(expr1))
print(tree_to_polish(expr1))
```
Traversing a tree

The order of visiting a tree can be defined (recursively)

- **Pre order**: print the root, then print the subtrees
  - Example: when we were printing an expr in polish notation

- **In order**: left subtree, root, right subtree
  - Example: when we were printing a “regular expression”

- **Post order**: print the subtrees, then the root
  - Example: reverse polish notation
def _p_polish_helper(polish_expr):
    if polish_expr[0] in ("+", "-", "+", "/"):
        arg1, remainder = _p_polish_helper(polish_expr[1:])
        arg2, remainder = _p_polish_helper(remainder)
        return TreeNode(polish_expr[0], arg1, arg2), remainder
    else:
        return TreeNode(polish_expr[0]), polish_expr[1:]

def parse_polish(text):
    tree, leftovers = _p_polish_helper(text.split(" "))
    if leftovers:
        print("There were extra symbols left over.")
    return tree
Trees – Twenty Questions

Does it have four legs?  y
Is it really large?  y
Is it an elephant?  n
I guessed wrong.

What did you have in mind?  a rhino
Please enter a question to differentiate between an elephant and a rhino: does it have a horn? an elephant. does it have a horn?  n

• Do you want to play again?  y
Four legs?

Really large?

Has wings?

- y: elephant
- n: dog
- y: bird
- n: snake

- n: dog
- n: snake
YES = 'y'
NO = 'n'

def get_yes_no_answer(question):
    while True:
        answer = input(question + " ")
        if answer == YES:
            return True
        elif answer == NO:
            return False
        else:
            print("I did not understand.")
class Question:
    def __init__(self, question_text, yes_answer=None, no_answer=None):
        self.__question_text = question_text
        self.__on_yes_answer = yes_answer
        self.__on_no_answer = no_answer
def ask_question(self):
    if self.__on_yes_answer is not None:
        if get_yes_no_answer(self.__question_text):
            self.__on_yes_answer.ask_question()
        else:
            self.__on_no_answer.ask_question()
    else:
        if get_yes_no_answer("is it " + self.__question_text + "?"):
            print("I knew it!")
        else:
            print("I guessed wrong.")
            self.__add_new_question()}
def __add_new_question(self):
    other_answer = input("What did you have in mind? ")
    new_question = input("Please enter a question to differentiate between " +\
                         self.__question_text + " and " + other_answer + " : ")
    if get_yes_no_answer(self.__question_text+"."+new_question):
        self.__on_yes_answer = Question(self.__question_text)
        self.__on_no_answer = Question(other_answer)
    else:
        self.__on_yes_answer = Question(other_answer)
        self.__on_no_answer = Question(self.__question_text)
    self.__question_text = new_question
def play_twenty_questions():
    root_question = Question("a sparrow")

    print("Let's play twenty questions. Think of something...")
    print("I'll guess it!")
    root_question.ask_question()
    while get_yes_no_answer("\n\nDo you want to play again?"):  
        root_question.ask_question()

    print("\n\nHere are all the possible answers entered into the game:")
    root_question.print_all_answers()

if __name__ == "__main__":
    play_twenty_questions()
def print_all_answers(self):
    if self.__on_yes_answer is None:
        print(self.__question_text)
    else:
        self.__on_no_answer.print_all_answers()
        self.__on_yes_answer.print_all_answers()