Web Resource: Trie Trees

We eventually want to retrieve all of the data using the prefix5 command. To get there, it is advantageous that when you are retrieving data, you build a trie tree. A trie tree is a rooted tree allowing for an unbounded number of child, unlike a binary tree that can have at most 2. The figure below (taken from wikipedia) gives an example. Note that the tree is built so that each unique prefix is kept as a separate node. Since all words start with ‘r’ but split after that, ‘r’ is a node value. After that, all words continue with ‘om’ or ‘ub’, so that is split as well.

Our Trie trees will be a bit different from the one in the picture: nodes will have the string (rather than the arcs). The trie tree should always have a root whose string is “”. Siblings will be in sorted order (e.g., node ‘an’ will be before node ‘ulus’).

Below, I have included code that you must use (it is also on the course website as hw7.py). You will use a class for the nodes. I have given you the initializer. A node has a parent, first child pointer, and a sibling pointer. So, all children of a parent node are kept as a singly-linked list through the sibling link. The parent accesses the children through the child pointer, which points to the first child. All children point to their parent.

The code also includes some testing code, which your code should work with. The test code also helps demonstrate how the new routines should work.

In the next 3 questions you will add some routines that give basic support for trie-trees: addchild, find and get_word. These routines will be used in a future homework to build more user-friendly routines, which will be part of a future homework.

class Node:
    def __init__(self,value):
        self.parent = None
        self.child = None
        self.sibling = None
        self.value = value

    def add_child(self,other):
        raise NotImplementedError
def find(self,string):
    raise NotImplementedError

def walk(self,prefix=""):  
    prefix += self.value
    if self.child is None:
        return [prefix]

    results = []
    ch = self.child
    while ch is not None:
        results.extend(ch.walk(prefix))
        ch = ch.sibling
    return results

def test():
    top = Node(""
    top.add_child(Node("r"))
    top.find("r").add_child(Node("ub"))
    top.find("r").add_child(Node("om"))
    top.find("rom").add_child(Node("ulus"))
    top.find("roman").add_child(Node("e"))
    print(top.walk())
    print(top.find("roman").get_word())

if __name__ == "__main__":
    test() 

Question 1: Add Child

addchild is used to add a new child to a parent node already in the tree for a given character string. For example, for the figure above, if there is already a node for ‘r’ and ‘om’ (but ‘om’ does not yet point to ‘r’), addchild can be used to add the ‘om’ node as a child of ‘r’. If there is a node for ‘ulus’, addchild can be used to add that node as a child of ‘om’. Note: this function does not need to determine how to split up a word, such as splitting ‘romans’ into ‘r’, ‘om’, and ‘ans’. We will address that in a later homework assignment.

addchild should keep the children of a node in sorted order. So when adding a new child to a parent that already has children, addchild needs to insert the child into the right spot in the single-linked list. So if ‘ub’ has been added to parent ‘r’, and then ‘om’ is added, ‘om’ needs to be inserted before ‘uh’. This is different than in the figure: in our version, ‘an’ should come before ‘ulus’ under the r-om path.

Hand in your code.

Question 2: Find

find is used to search the tree to find the node that the string ends with. For the tree in the figure above, and if ‘top’ is a pointer to the top node, top.find(‘rube’) will walk down from the ‘r’ node, to the ‘ub’ node, and then to the ‘e’ node, and return that ‘e’ node. If the string does not exactly correspond to a path in the tree, find should return None.

Note that this find method find is specific to objects in our Node class. It is different than the find method that works for strings. Python knows which method to use based on the type of the object.
Hand in your code.

**Question 3: Get Word**

The method `get_word` is used to create the word that ends at the node. Thus, it should trace its way to the root of the Trie Tree in creating its output.

Hand your code.

**Graphs**

**Question 4: Space for Storing a Graph**

Say you have a unweighted undirected graph with 1024 vertices, numbered 0 through 1023. Say the graph is pretty sparse and has 2048 edges altogether.

How much space (in bits/bytes) would that graph take to represent as an adjacency matrix? Assume we can store edges on a bit-level within the matrix.

How much space (in bits/bytes) would an adjacency-list representation of the graph take. Assume we are working on a 64-bit system, where each reference/pointer would take 8-bytes. You should follow the implementation of adjacency-lists exactly as suggested by the textbook.

**All about Eve**

The following questions use data from Eve Online, a MMORPG (massively multiplayer online role-playing game). We will be using this data for some of the graph theory problems coming up. The purpose of the first set of questions is to get basic support in place: reading in the data and setting up a class structure for it.

The Eve data is a bit messy, just as real world data often is.

The CSV (comma separated values) file `hw7-universe.csv` contains information about the Eve Online universe. The universe is made up of approximately 8,000 solar systems which have one or more star-gates, allowing travel to adjacent solar systems. The fields you are interested in are:

- `system_id` a unique integer identifier for each system (refer to this as `id`)
- `solarsystem_name` a string containing the name of the system (refer to this as `name`)
- `stargates` a list of `system_ids` that you can reach from the current system in 1 hop.
- `x, y, z` the system’s coordinates
- `security_status` a rating of how lawful the system is

**Question 5: Create a System Class**

Create a class called `System` so that you can have an object for each system. Each object should have 7 fields above in, and the class initializer should initialize them, based on input arguments.

Add in two class variables `id_to_obj` and `name_to_obj`. These are variables associated with the class rather than an object in the class. These will be python dictionaries to map `system_id` and `solarsystem_name` to
the object’s pointer. So the class initializer should update these two class variables each time a new system is added.

The purpose of the class variables `id_to_obj` and `name_to_obj` is to provide a way to get a system’s object pointer if you just know its system id, or name.

Hand in your code.

**Question 6: Reading in the file with ‘with’ and csv module**

In this homework, you will be reading a file and processing it. Python has two ways of opening and closing files. The first method involves using the open keyword:

```python
f = open('some_file.txt')
# doing stuff
f.close()
```

For this assignment, we will be using the preferred python way by using a ‘context manager’, using the ‘with’ keyword, just as you did in homework 2.

```python
with open("some_file.txt") as f:
    # do stuff with the file handler f
    # notice that we’re indented
    # when I’m done with the file, go back to my original indention
```

Although we rarely use modules in this class, we will be using the csv module to read in the csv file. This will allow for easy access to the 7 fields of interest. Ignore any system that has a `system_id` greater than 31,000,000.

You will need to parse the string of stargates yourself to turn it into a python list, which is what you will pass to the System class initializer. For example, for the first system (30000588) has a value in its stargates field of “[30000589, 30000716, 30000558, 30000587]”. You need to turn this string into the corresponding python list. Strip out any stargates with a `system_id` greater than 31,000,000.

Put your code for reading in the csv file into a static method of the System class called `load_eve`.

Turn in your code.

**Question 7: Numbering systems from 0**

The ids start at 3-million. As we will be using the systems as vertices in a graph, add another field to your class instances called `index`. Number all of the systems in order that they are read in starting at 0. This should be done by the class initializer, rather than by the class method that reads in the csv.

You will need a class variable `next_index` to keep track of the next index to use (this will be your 3rd class variable). Add a field `index` to each object so that the object also knows what index it is.

Also, add a 4th class variable `index_to_obj` to keep a mapping from indices to objects.

Turn in your code.

**Question 8: Converting Stargates to Objects**

The list of stargates for each object is a list of their `system_id`'s. It would be more useful to have this as a list of their objects. Write a class method `convert_stargates` to convert each object’s list of stargates into
a list of their object pointers. Have `load_eve` call `convert_stargates` after it has finished reading in all of the data.

Turn in your code for `convert_stargates`.

**Question 9: Creating an Adjacency List**

In the next question, you will be doing a breadth first search. The systems will be the nodes in the graph, and the stargates will tell you what other systems each system is adjacent to.

To do a breadth first search, we need an adjacency list. Since all systems have an index starting 0, we will use this to refer to the nodes, and what nodes they are adjacent to.

Create a class method called `create_adjacency_list` that will create a list of adjacency lists. The outer list has an entry for each system and the inner list is a list of all of the other systems it is connect to via a stargate.

Turn in your code.

**Question 10: Breadth First Search**

Implement the bread-first-search algorithm of the textbook. Stay as true to it as possible! For a queue, just use a list, and where an enqueue is append and dequeue is pop(0). For color, d, and π, we could create a class for vertices, but seems a bit much. Let’s just use a list for each of them (can use the vertex indexes). Call the function BFS, and it should take two arguments: a graph in an adjacency list format and a start node. It should output the array π.

Note BFS will not be part of your System class, as it is a generic BFS algorithm, and not specific to System at all.

Turn in your code.

**Question 11: Finding Shortest Routes**

Make a function that takes two inputs, a source and a destination, in terms of their solarsystem name. It should use BFS and it should return the path as a list of solarsystem names.

The shortest path from “Jita” to “Dodixie” should be be 13 systems, including the start and destination.

Turn in your code.