CSE 4256 Class 5: Default Dictionaries and Deques - Due Saturday, June 12

Default Dictionaries

The following code returns a dictionary with the word count for each word in a list of words.

```
def count_words(l):
    d = {}
    for word in l:
    if word not in d:
        d[word] = 1
    else:
        d[word] += 1
    return d
```

Notice that we need to have two cases: one for if the word is not in the dictionary, and the other for if it is.

We can shorten the code by using a default dictionary.

```
from collections import defaultdict
def count_words2(l):
    d = defaultdict(int)
    for word in l:
        d[word] += 1
    return d
```

When creating a default dictionary, we must pass it the type of the values that will be stored in the dictionary. We can use any type that we want, including lists.

The following code takes a list of words as an argument and returns a dictionary containing lists of the words that have the same first letter.

```
def list_words(dictionary_of_words):
    d = defaultdict(list)
    for word in dictionary_of_words:
        d[word[0]].append(word)
    return d
result = list_words(["antelope", "bear", "alligator", "aardvark", "cat", "bee"])
print(result)
```

Deques

So far in this course, we have been working with lists. A list is an object that is built on top of an array. Arrays store elements in contiguous computer memory. This allows us constant time access to any element in the array through indexing. Arrays do have a drawback, however. If we remove the first element in an array, every other element in the array needs to be shifted to the left. In other words, removing an element from a list can take linear time.

A Deque is a double-ended queue which is implemented using a linked-list. Unlike with lists, you can't access elements in constant time. However, you can remove the first element of a deque in constant time. Under the hood, an element is removed from the queue by removing the first element, then removing the links off of the first element and having the headpointer point to the second element. This can be done in constant time regardless of the length of the linked list. The computer though takes care of these low level details for you. To actually use a deque as a queue, you only need to know two methods: **append()**, which pushes an element onto the end of the queue, and **popleft()**, which removes the element at the front.

Deques can be used as stacks. To do this, use the methods **append()** and **pop()**.

The deque class must be imported before you can use it.

```
from collections import deque
d = deque()
d.append(42)
d.append(12)
print(d.popleft())
```

Homework

1. Write a method called is **balanced(s)** that take a string of open and closed parentheses and returns **True** if the string is balanced, **False** otherwise. The method should use a deque as a stack.

For example, the following parentheses are balanced:
()()
(())()

While the following are not:
)(
))()

2. Write a method that implements breadth-first search on an undirected graph. Your method should be called **bfs(graph, start)**, where graph is a dictionary representation of a graph, and start is the starting vertex. The vertices of the graph are number 0, 1, 2, 3, . . . , n. The method should return a list containing the distance from the starting vertex for each vertex. in the graph.

Below is the psuedo-code for the method you will implement:

```
1) Create a list that holds the distance of each vertex from the source
vertex.

2) Set the distance of the starting vertex to 0, the rest to infinity
in the list.

2) Create a queue and put the starting vertex on the queue.

3) while the queue has elements in it:
    3a) Remove the element at the front of the queue.
    3b) If the element you removed has its distance set to infinity,
        4) Set its distance to 1 + (the distance of the vertex it was
        was visited from), and add it to the queue.
5) After the loop breaks, return the list of distances.
```

3. Write a method called **is_connected(graph)** that takes a dictionary representation of an undirected graph as an argument and returns **True** if the graph is connected, **False** otherwise. The method should call **bfs(graph, start)** as a sub-procedure.

4. Write a method called **connected_components(graph)** that takes a dictionary representation of an undirected graph as an argument and returns an integer representing the number of connected components in the graph. The method should repeatedly call bfs(graph, start) as a sub-procedure.