0) This test is out of 30 points. Each question is worth 10.

0.5) DO NOT WRITE ON OR EAT THIS SHEET.

0.9) You MUST write legibly, and the indentation on python code lines <u>must be obvious.</u> Can't read, no points.

1) Create the contents of the program file *committee_sort.py* which will read an input file containing the members of many committees, and will sort the names of the members of each committee, and write the results to an output file. Each line in the input file will contain the name of a committee and list its members. The committee name and member names will be separated by commas. You'll put the names in alphabetical order (assume that the names are all in lower-case) and write the lines to the output file. For instance, if the program is called from the command-line as…

```
python  committee_sort.py  fred.csv  george.csv
```

… and *fred.csv* contains:

```
Ultimate Frisbee,peter brooks,isaac newton,harry truman
Fractions,
Model U.N.,lady gaga,frank sinatra,fred the baker,lillian hellman
```

For instance, in the first line above, "Ultimate Frisbee" is the committee name, and the rest of the 3 names are committee members. There will be one or more lines in the input file, and each committee will have zero or more members – there's no limit on the number of lines or members/committee on each line. Ignore any committee with no members. Here's what the output file *george.csv* should look like:

```
Ultimate Frisbee,harry truman,isaac newton,peter brooks
Model U.N.,frank sinatra,fred the baker,lady gaga,lillian hellman
```

Of course, you'll have to obtain the input and output filenames from the command-line. Also, you should indicate, in the usual way on the first line, that you want the Python3 interpreter.

2) You'll be creating 3 methods of a class called *Dlist*, which is a doubly-linked ordered list. It stores instances of the *Node* class (below), and maintains the nodes in increasing value order. Create the methods: *__init__(), delete()* and *tolist()*. Assume that the *insert()* method has been created already by Guido Van Rossum.

```
class Node:  # these contain only data, so
             # only __init__() is necessary
             # self.next and self.previous will be used as
             # pointers in the linked list
   def __init__(self,value):
      self.value = value
      self.next = None
      self.previous = None

class Dlist:
   def __init__(self):
      # your glorious code here
   def insert(self, value):
      # secretly written by Guido Van Rossum
   def delete(self, value):
      # you code to remove the first occurrence of
      # value in the list and return True, or return False if not found
   def tolist(self):
      # your code to return a list of the values in order,
      # and remove all nodes in Dlist (return empty list if no nodes)
```

3) You are going to create two views of a priority queue as a min-heap, containing upper-case letters from your name. Write your first and last name, in upper-case letters with no spaces. (e.g. PETERBROOKS) Take the first 7 letters, and ignore the rest (e.g. PETERBR) – (Bill Ni: you are a problem – use BILLNIB). Now push each letter, one by one, onto a priority-queue, and display the resultant min-heap -- the letters are naturally compared by alphabetic position, and therefore the earliest letter (in my case: "B") will be at the root. My tree will look like the first one below. Then pop two letters off the queue, and display the new resultant heap (the second one below). This is where your scratch paper comes in handy. So, your answer will be two pictures, like the ones below, showing where the letters are in your heaps.

I want to see all 9 versions of your heap, one after each of 7 pushes, and after each of the 2 pops. Make sure the diagrams don't interfere visually with each other (write small).

I'd like to address an ambiguity in the "bubbling-down" of data that I may not have covered when I showed the "bubbling-up" and "bubbling-down" procedures on the board. When a piece of data is "bubbling-down", at any one point it will be the parent of zero, one or two children. If at least one of the children is smaller than the parent, we'll swap the parent and the smaller child. That much we've covered. But what if there are two children, and they are equal to each other, and both are smaller than the parent? Which child should we swap the parent with? To resolve that ambiguity, we should swap the parent with the left child.