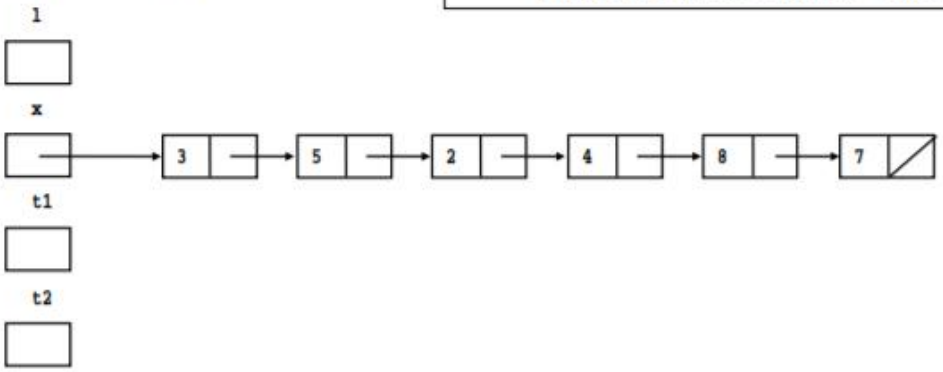
1. (4 pts) Examine the **mystery** method and hand simulate the call **mystery(x);** using the linked list below. **Lightly cross out** ALL references that are replaced and **Write in** new references: don't erase any references. It will look a bit messy, but be as neat as you can. Show references to **None** as **/**. Do your work on scratch paper first.

```
def mystery(l):
  while l.next != None and l.next.next != None:
    t1      = l.next
    t2      = t1.next
    t1.next = t2.next
    t2.next = t1
    l.next  = t2
    l       = t1
```

```
class LN:
    def __init__(self, value, next=None):
        self.value, self.next = value, next

class TN:
    def __init__(self, value, left=None, right=None):
        self.value, self.left, self.right = value, left, right
```



2. (2 pts) Draw the binary search tree that results from inserting the following values (in the following order) into an empty binary search tree: **13**, **10**, **6**, **5**, **8**, **3**, **4**, **16**, **11**, **17**, **1**, **2**, **15**, **19**, **18**, **14**, **9**, **20**, **7**, and **12**. Draw the number for each tree node, with lines down to its children nodes. Space-it-out to be easy to read. Answer the questions in the box.

Size =

Height =

3. (4 pts) Define an **iterative** function named **separate**; it is passed one linked list and a predicate; it returns a **2-tuple** of two linked lists: the first is a linked list of all the values in the parameter where the predicate returns **True**; the second is a linked list of all the values in the parameter where the predicate returns **False**; the values in each list must be the reverse of their order in the parameter linked list (this makes the code easier to write, not harder: adding a value at the front of a linked list is easier). For example if we defined

```
a = list_to_ll([1,2,3,4,5,6])
```

writing `even,odd = separate(a,lambda x : x%2 == 0)` results in **even** referring to a linked list containing the values **6**, **4**, and **2** (in that order) ; **odd** referring to a linked list containing the values **5**, **3**, and **1** (in that order). You **may not** use Python lists, tuples, sets, or dicts in your code: just use linked lists.

4. (4 pts) A min-heap is a binary tree whose order property is that every node is strictly smaller than any of its children (if a left/right child is **None**, you don't have to check it). Write the **recursive** function **is_min_heap** which checks this order property **for every node in the tree** and returns **True** if the binary tree is a min-heap and **False** if it is not. Think symmetry.

5. (4 pts) We learned that when we declare a **class** using inheritance, the **__bases__** attribute of the **class** is bound to a **tuple** of references to its **base** classes. Write a **recursive** function named **bases** that takes a reference to any **class** and returns a **set** containing that **class** and all its base classes (all the way back to the **object** class). You may not use the **__mro__** or **mro** attributes of the **class**, which would trivialize your function. You may use both iteration (over the **__bases__** list) and recursion to write this function. For example, given the following **class** definitions in a script (class **A** appears as **__main__.A**)

```
class F:pass
class C:pass
class G:pass
class B(F):pass
class D(G):pass
class A(B,C,D):pass
```

Calling `bases(A)` returns the **set**

```
{<class '__main__.A'>, <class '__main__.D'>, <class '__main__.B'>,
<class '__main__.G'>, <class 'object'>, <class '__main__.C'>,
<class '__main__.F'>}
```

Hint: the **union** method for a **set** takes any number of arguments (other **set**s: `a.union(b,c,d)`) and computes the union of all of them Remember how to use * to take a **tuple** of **set**s and convert them into that many **set** arguments in a function call.. You can also use **functools.reduce** with | (the **set** union operator: `a | b`).

6. (7 pts) Define a class named **popdict**, derived from the **dict** class; in addition to being a dictionary, it remembers how often each key is accessed (how **pop**ular it is), and iterates through the **popdict** in decreasing order of how frequently the key was accessed. Besides storing the standard dictionary of keys and their values, a **popdict** stores an auxiliary dictionary remembering how often each key has been accessed. The keys in the actual dictionary should always be the same as the keys in the auxiliary dictionary. See the notes for how **pdefaultdict** is derived from **dict**; this derivation is similar.

Define the derived class **popdict** with only the following methods:

- **__init__ (self,initial_dict=[],**kargs)**: initializes the dictionary and also creates an auxiliary popularity dictionary (I used a **defaultdict**) for remembering how often each key is accessed: initialize this popularity dictionary so that it shows each key in the newly initialized dictionary as being used once so far.
- **__getitem__ (self,key)**: for any **key** in the dictionary, return its associated value and increase its popularity (how often it has been used) by **1** in the popularity dictionary.
- **__setitem__ (self,key,value)**: set **key** to associate with **value** in the dictionary and increase its popularity (how often it has been used) by **1** in the popularity dictionary.
- **__delitem__ (self,key)**: for any **key** in the dictionary, remove it from both the dictionary and popularity dictionary.
- **__call__ (self,key)**: returns the popularity of **key** (the number of times it has been used); for a **key** not in the dictionaries, return **0**.
- **clear (self)**: remove all keys (and their associated values) from both dictionaries.
- **__iter__ (self)**: return a generator that will yield all the keys in order of decreasing popularity.