Homework 2, CS320, Spring 2014, McConnell

Due Monday, 9/22 before class

To check in use `~cs320/bin/checkin` HW2 filename

Make sure that it runs with the command `ipython -i Heap.py` on the department Linux machines before you turn it it.

Last modified 9/20/10:15

1. You have a ladder with n rungs and a bunch of identical jars. You want to test how high a rung a jar can be dropped from without breaking. (Assume there is a highest rung that you can drop a jar without breaking, and that if you drop one from the next higher rung, it always breaks.)

   (a) Give the best big-Θ bound on the number of drops you need to perform to find this rung. It should be expressed as a function of n.

   (b) Suppose you only have two jars. Give a big-Θ bound you can get on the number of drops to determine the rung. It should be $o(n)$.

   Notice that a dumb algorithm would be to drop the first jar from every rung until it breaks.

   Suppose we drop the first jar from every other rung, until it breaks. Now we have it narrowed down to two rungs, and we can figure out our answer by dropping the second jar once. However, the roughly $n/2$ drops for the first jar is still $\Theta(n)$. It's still $\Theta(n)$ if you use $n/3$ drops and two drops for the second jar.

   If we go to the other extreme, we could drop the first jar from the middle rung. This narrows the number of candidate rungs down to about $n/2$ with a single drop of the first jar. If the first jar has broken, we need to use about $n/2$ drops for the second jar, which is $\Theta(n)$.

   If you find something between these two extremes, one that roughly equalizes the worst-case number of drops for each of the two jars, you will see that you get a better big-Θ bound on the total number of drops.

   (c) Suppose you only have three jars. Give a big-Θ bound on the number of jars to determine the rung. *Hint: Once the first jar has broken, you've reduced the problem to the case of two jars, on some number n' of rungs. Try to find a solution that makes the worst-case number of drops for the first drop rougly equal to the number of drops for this smaller problem on n'.*

2. Go through the Python tutorial, experimenting with the examples it shows you. Investing some time in this will save you time on programming assignments in this course. An alternative is the following `Code Academy` tutorial that quizzes you as you go along:

   - http://www.codecademy.com/en/tracks/python

   Some of the initial exercises are quite easy, but you can skip ahead in the lessons.

3. Read about binary heaps in Chapter 2. You have seen them before in CS200. They support a *priority queue* abstract data type, which maintains a set of values, and supports the following operations:

   - Insert a value to the set;
   - Remove and return the minimum value from the set.

   Below is a partial implementation in Python. I've written some of the methods for you, and put in stubs for the methods you are supposed to complete. The comments on these stubs give hints about how to complete the methods.

4. Type in the partial implementation to a file called `Heap.py`; doing this will help you notice things about Python syntax that you would overlook if you only examined the code. Try to run it with the command:

   - ipython -i Heap.py

   Unless you typed it in perfectly, You will see some reports of mistakes, you will see how different errors are reported in Python. Sometimes they are reported when you run the `ipython` command, and sometimes when a method is invoked. You may also introduce bugs. Try to figure out how to fix them, but not too long; you have the answer key if you're stumped.

   Uncomment the __str__ method. This is similar to the Java reserved term `toString`, which gets called in a context where a string representation of an object is needed. An example of this is the last print statement in the __main__.

   From the interpreter, try creating a new heap Q, insert elements to it, and print it to make sure it is working correctly. This partial implementation supports `insert` operations but not `extractMin` operations.

5. To get it to support `extractMin` operations, fill out the stubs. You can test each one when you've finished it by calling it from the interpreter on P, which the `main` has created for you. The methods I've written for you can be used to remind yourself of Python syntax and various nice tricks the language supports.

6. Modify it so that it supports a k-ary heap, which is like a binary heap, except that each node has k children, rather than two. Verify that an insertion now takes $O(\log_k n)$ and an extract-min takes $O(k \log_k n)$ time. Now, k appears inside the big-O, because we've made it a variable. You can ignore constants in a big-O bound, but you cannot ignore variables.

   One thing you will have to modify is the constructor, since it will have to record an additional variable that keeps track of the degree.

   As k gets large, the first bound gets smaller and the second one gets larger. If the first operation is going to be used by a algorithm many more times than the second, then it pays to make k larger. It pays to choose k so that the total amount of time spent on the two operations is roughly equal. The strategy is similar to the one about dropping jars. We will see that we can get better bounds for some of the graph algorithms that we will study by using this trick.

Also, notice that for $k = 3$, $k \log_k n$ and $\log_k n$ are both smaller than they are for $k = 2$. A little-noticed fact is that choosing k to be 3 is always better than choosing it to be 2, though not in a big-O sense.

7. Test your methods thoroughly before you check them in.

```
#  Heap class:  The internal variables are:
#   self._H:  Heap array as defined in the book
#   self._allocation:  size of self._H; some elements at the end of this
#     array might be unused.  This is always a power of two.
#   self._size:  Number of elements currently stored in self._H;  this
#     is always at least self._allocation/4 and at most self._allocation
class Heap(object):

   # constructor:  create a heap with allocation 1 and size 0:
   # This is where the instance variables for the class are defined.
   # 'self' is the Python equivalent of 'this'.  Python requires you
   # to mention 'self' explicitly in a lot of contexts where 'this'
   # can be omitted in Java.
   #
   # To call it, execute P = Heap(); this assigns a reference to an
   # empty heap to variable P.  (It implements the "priority queue" ADT.)
   def __init__(self):
      self._H = [0]
      self._size = 0
      self._allocation = 1

   # get current size
   # If P contains a reference to it, then call it with P.getSize()
   # The P before the dot will assign a reference to the parameter 'self'
   def getSize(self):
      return self._size

   # get current allocation
   def getAllocation(self):
      return self._allocation

   # get a reference to the heap array
   def getArray(self):
      return self._H

   #  Find the parent of index pos, or return None if it's the root.
   #    There is no check to see if pos >= self._size
   #    Example of a call:  P.parent(i).  This assigns the reference in P
   #    to 'self' and the reference to the integer in pos to i.
   def parent(self, pos):
      if pos <= 0: return None
      else: return (pos - 1) / 2
```

```
#  Swap the elements at pos1 and pos2 of the heap array.  Leave the
#    structure unchanged if one or more of the indices are not in
#    {0, 1, 2, ... getSize()-1}.  Otherwise, swap the elements.
def swap (self, pos1, pos2):
    s = self.getSize()
    if 0 <= pos1 and 0 <= pos2 and pos1 < s and pos2 < s:
        self._H[pos1], self._H[pos2] = self._H[pos2], self._H[pos1]

# insert integer to heap.  Call:  P.insert(i)
def insert(self, item):
   # double allocation of array if it's full.  This takes time
   # proportional to the number n of items in the heap, but
   # the last time it happened, the heap had n/2 elements in it.
   # Averaging the cost of this over the n/2 insertions that have
   # taken place since this gives a constant average (amortized) cost
   # per insertion.
   if self._size == self._allocation:
      self._H = self._H[:] + [0] * self._allocation
      self._allocation = self._allocation * 2

   # insert the item at the end ...
   self._H[self._size] = item
   pos = self._size
   self._size += 1
   # restore the heap property by bubbling it up to a point where
   # it's at the root or its parent's key is at least as small ...
   while pos > 0 and self._H[self.parent(pos)] > self._H[pos]:
      self.swap(self.parent(pos), pos)
      pos = self.parent(pos)

# Return the index of child number i of the node at position pos.
#   Preconditions are that pos is a valid index.  It will return
#   the index of this child, whether or not it exists and whether
#   or not its position is beyond the end of the heap array
def childIndex (self, pos, i):
   return 0  # fill in code here

# Tell the key in child number i (i in {0,1}) of the node at position 'pos'
#  of heap array.  If there is no such child, return None.  Precondition
#  is that i is in {0,1}
def child (self, pos, i):
    return None  # fill in code here

"""
 Return a list of keys at children of the node at position pos.
  Return None if 'pos' is not in {0, 1, 2, ... size-1}
  If it is, it might have 2, 1 or 0 children; return a list of length 2, 1
  or 0.

  Here's a hint:  create a list of self.child(pos,i) for i in
  [leftChild, leftChild+1].  One or both of these could be None,
  indicating the child doesn't exist.  Create and return a new
  list that is equal to this list, but with the None values omitted.

"""
def children(self, pos):
   return None  # fill in code here

# Extract the minimum element of the heap, and restore the heap property.
# See our book for the algorithm.
def extractMin(self):
   return None # fill in code here


'''
   In a subtree rooted at pos in which the only place where the heap
    property does not apply is at the root, restore the heap property.
    A precondition is that 0 <= pos < self._size.

    Here's a hint:  call self.children(pos).  If the returned list C
    is empty, the element at 'pos' has no children, and the heap
    property applies.  Otherwise, if min(C) > self._H(pos),
    the heap property applies.  Otherwise, you can find which child
    number contains the minimum key with childNum = C.index(min(C)).  (Look
    up this 'index' method.)  You can tell the index of this child
    in self._H with a call to self.childIndex(pos, childNum).
'''

def heapify(self, pos):
   pass  # Fill in code here

'''
  Create a string that is suitable for displaying structure of the heap
  To print the heap array, print P._H.  This one displays which key
  is a child of which using indentation.  For example:

  3
    5
     10
     29
    4
     20
     15
```

```
      The least indented element, 3, is at the root.  Its children
      are the elements 5 and 4 at the next level of indentation.
      The two elements at the next level of indentation following 5,
      namely, 10 and 29, are the children of 5.  Similarly, 20 and 15
      are the children of 4.

    '''

    '''
    I've commented it out because it won't work until you've written
    some of your methods ...

    def __str__(self):
        if self.getSize() == 0:  return ""
        else:  return self.strAux(0, 0, "")

    def strAux(self, pos, depth, outString):
        outString = outString + "\n" + ' ' * depth + str(self._H[pos])
        children = self.children(pos)
        for i in range(len(children)):
            outString = self.strAux(self.childIndex(pos,i), depth+1, outString)
        return outString
    '''

if __name__ == "__main__":
     P = Heap()
     P.insert(10)
     P.insert(5)
     P.insert(15)
     P.insert(3)
     P.insert(29)

     print "Heap array:  " + str(P.getArray())
     print "Size: " + str(P.getSize())
     print "allocation " + str(P.getAllocation())

     #  This will print the structure of the heap once you've written
     #   your other methods, because of the __str__ method I've written
     print P
```