Data Structures - Test 2

Question 1. (10 points) What is printed by the following program?

```
def recFn(a, b):
    if a == b:
        return b
    elif a > b:
        return a
    else:
        return a + recFn(a + 1, b - 2) - b

print("Result = ", recFn(1, 10))
```

Question 2. a) (12 points) Write a recursive Python function to compute the following mathematical function, G(n):

\[ G(n) = \begin{cases} 
  n & \text{for all values of } n \leq 3 \text{ (e.g., } G(2) \text{ value is 2)} \\
  G(n-4) + G(n-2) & \text{for all values of } n > 3.
\]

```
def G(n):
```

b) (8 points) For the above recursive function G(n), complete the calling-tree for G(8).

```
G(8)
   /   \
  /     \
G(4)   G(6)
```

c) (3 points) What is the value of G(8)?

d) (2 points) What is the maximum number of call-frames of G on the run-time stack when calculating G(8) recursively?
Question 3. (15 points) Consider the following simple sorts discussed in class -- all of which sort in ascending order.

```python
def bubbleSort(myList):
    for lastUnsortedIndex in range(len(myList)-1,0,-1):
        alreadySorted = True
        for testIndex in range(lastUnsortedIndex):
            if myList[testIndex] > myList[testIndex+1]:
                temp = myList[testIndex]
                myList[testIndex] = myList[testIndex+1]
                myList[testIndex+1] = temp
                alreadySorted = False
        if alreadySorted:
            return
```

```python
def insertionSort(myList):
    for firstUnsortedIndex in range(1,len(myList)):
        itemToInsert = myList[firstUnsortedIndex]
        testIndex = firstUnsortedIndex - 1
        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1
        myList[testIndex + 1] = itemToInsert
```

```python
def selectionSort(aList):
    for lastUnsortedIndex in range(len(aList)-1, 0, -1):
        maxIndex = 0
        for testIndex in range(1, lastUnsortedIndex+1):
            if aList[testIndex] > aList[maxIndex]:
                maxIndex = testIndex
        # exchange the items at maxIndex and lastUnsortedIndex
        temp = aList[lastUnsortedIndex]
        aList[lastUnsortedIndex] = aList[maxIndex]
        aList[maxIndex] = temp
```

<table>
<thead>
<tr>
<th>Type of sorting algorithm</th>
<th>Initial Ordering of Items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Descending</td>
</tr>
<tr>
<td>bubbleSort.py</td>
<td>24.5</td>
</tr>
<tr>
<td>insertionSort.py</td>
<td>14.2</td>
</tr>
<tr>
<td>selectionSort.py</td>
<td>7.3</td>
</tr>
</tbody>
</table>

a) Explain why insertionSort on a descending list (14.2 s) takes longer than insertionSort on a random list (7.3 s).

b) Explain why insertionSort on a descending list (14.2 s) takes longer than selectionSort on a descending list (7.3 s).

c) Explain why bubble sort is $O(n^2)$ in the worst-case.
Question 4. Two common rehashing strategies for open-address hashing are linear probing and quadratic probing:

| quadratic probing | Check the square of the attempt-number away for an available slot, i.e.,  
| home address + ( (rehash attempt #)^2 + (rehash attempt #) ) / 2 | % (hash table size), where the hash table size is a power of 2. Integer division is used above |

a) (8 points) Insert “Paul Gray” and then “Sarah Diesburg” using Linear (on left) and Quadratic (on right) probing.

<table>
<thead>
<tr>
<th>Hash Table with Linear Probing</th>
<th>Hash function</th>
<th>Hash Table with Quad. Probing</th>
</tr>
</thead>
<tbody>
<tr>
<td>0  Ben Schafer</td>
<td>hash(John Doe) = 7</td>
<td>0  Ben Schafer</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3  Philip East</td>
<td>hash(Philip East) = 3</td>
<td>3  Philip East</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>5  Mark Fienup</td>
<td>hash(Mark Fienup) = 6</td>
<td>5</td>
</tr>
<tr>
<td>6  John Doe</td>
<td>hash(Ben Schafer) = 0</td>
<td>6  Mark Fienup</td>
</tr>
<tr>
<td>7</td>
<td>hash(Paul Gray) = 6</td>
<td>7  John Doe</td>
</tr>
<tr>
<td></td>
<td>hash(Sarah Diesburg) = 0</td>
<td></td>
</tr>
</tbody>
</table>

b) (4 points) In open-address hashing (like the pictures above), how do we handle deleting items in the hash table?

c) (3 points) In open-address hashing (like the pictures above), how do deleted items effect performance of searching?

d) (5 points) In closed-address hashing (e.g., ChainingDict like picture below), if the load factor (# items / hash table size) is close to 1, say 0.99, would you expect average-case searches of O(1)? (Justify your answer)
Question 5. (20 points) In class we discussed the insertionSort code shown in question 3 on page 2 which sorts in ascending order (smallest to largest) and builds the sorted part on the left-hand side of the list.

For this question write a variation of insertion sort that:
- sorts in **descending order** (largest to smallest), and
- builds the **sorted part on the right-hand side** of the list, i.e.,

```
Unsorted Part
myList: 0 1 2 3 4 5 6 7 8
```

```
Sorted Part
```

```
10 20 35 40 45 60 25 50 90
```

def insertionSortVariation(myList):

```
```

Question 6. Recall the general idea of Heap sort which uses a min-heap (class BinHeap with methods: `BinHeap()`, `insert(item)`, `delMin()`, `isEmpty()`, `size()`) to sort a list.

**General idea of Heap sort:**

1. Create an empty heap
2. Insert all n list items into heap
3. `delMin` heap items back to list in sorted order

```
myList                      sorted list with n items
```

```
myList                    unsorted list with n items
```

```
heap with n items
```

```
myList                      sorted list with n items
```

b) (5 points) Determine the overall $O(\, )$ for your heap sort and briefly justify your answer. Let $n = \text{len}(\text{myList})$.