Data Structures - Test 2

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

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

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

```
  1  10  
  2  8  
  3  6  
  4  4  
  Result = -14
```

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

\[
G(n) = \begin{cases} 
  n & \text{if } n \leq 3 \\
  G(n-4) + G(n-2) & \text{for all } n > 3
\end{cases}
\]

```python
def G(n):
    if n <= 3:
        return n
    else:
        return G(n-4) + G(n-2)
```

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

![Calling-tree for G(8)](image)

```
G(0) + G(2)   \( G(6) \) + \( G(4) \) = 2
```

\[
G(0) + G(2)   \( G(6) \) + \( G(4) \) = 2
```

c) (3 points) What is the value of G(8)? \( \boxed{6} \)
d) (2 points) What is the maximum number of call-frames of G on the run-time stack when calculating G(8) recursively? \( \boxed{4} \)
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

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

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).

For descending order the whole sorted part is compared and shifted before inserting at index 0. On random order, we expect only needing to compare and shift half way down sorted part.

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

Insertion sort must compare and shift whole sorted part. Selection sort compares whole unsorted part to find max. item’s index, but only needs 3 moves to swap max item. Insertion does more moves.

c) Explain why bubble sort is $O(n^2)$ in the worst-case.

Descending order is the worst-case for bubble sort since item at index 0 must compare and swap down whole unsorted part. Since the unsorted part shrinks by one on each iteration of the order, the number of compares and swaps is \( \frac{n \times (n-1)}{2} = \frac{n^2 - n}{2} \) which is \( O(n^2) \).
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., 
|                   | \[\text{home address} + \left(\frac{(\text{rehash attempt} \#)^2 + (\text{rehash attempt} \#)}{2}\right) \mod (\text{hash table size})\], 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.

Hash Table with Linear Probing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben Schafer</td>
<td>Phillip East</td>
<td>Sarah D.</td>
<td>John Doe</td>
<td>Mark Fienup</td>
<td>Paul Gray</td>
<td>Philip East</td>
<td>John Doe</td>
</tr>
</tbody>
</table>

Hash function
- hash(John Doe) = 7
- hash(Philip East) = 3
- hash(Mark Fienup) = 6
- hash(Ben Schafer) = 0
- hash(Paul Gray) = 6
- hash(Sarah Diesburg) = 0

Hash Table with Quad. Probing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ben Schafer</td>
<td>Philip East</td>
<td>Sarah D.</td>
<td>John Doe</td>
<td>Mark Fienup</td>
<td>Paul Gray</td>
<td>(0 + \left(\frac{2^2+2}{2}\right)) \mod 8 = 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(0 + \left(\frac{3^2+1}{2}\right)) \mod 8 = 6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(0 + \left(\frac{4^2+4}{2}\right)) \mod 8 = 2</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? We overwrite deleted item with special "deleted" value (e.g., True in OpenAddrHashTable class) which is different than None for empty slot.

c) (3 points) In open-address hashing (like the pictures above), how do deleted items affect performance of searching? Slots with "deleted" values are treated as non-matches for target key, i.e., need to keep search until target key or None/empty slot.

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)

Yes, if hash function is doing a "good" job, then each slot has a list of about 1 in size. Search a list of 1 (or a few items) is constant.
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 | Sorted Part
myList:       |                    | Sorted
0 1 2 3 4 5 6 7 8  |                     | Sorted
        | 25 40 90 60 50 45 35 20 10 |
```

```
def insertionSortVariation(myList):
    for lastUnsorted in range(len(myList)-2, -1, -1):
        itemToInsert = myList[lastUnsorted]
        testIndex = lastUnsorted + 1
        while testIndex <= len(myList) - 1 and myList[testIndex] > itemToInsert:
            myList[testIndex] = myList[testIndex - 1]
            testIndex += 1
        myList[testIndex - 1] = itemToInsert
```

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

```
from bin_heap import BinHeap

def heapSort(myList):
    myHeap = BinHeap()  # Create an empty heap
    for item in myList:
        myHeap.insert(item)
    for index in range(len(myList) - 1, -1, -1):
        myList[index] = myHeap.delMin()
```

**a) (5 points)** Complete the code for heapSort so that it sorts in descending order

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

$O(n \log n)$: The two for-loops are not nested so we add their contributions. Each for-loop loops $n$ times and calls insert or delMin which are $O(\log n)$ because the heap has height $\log n$ (see above diagram).