CSCI 4041, Spring 2019, Quiz 3 (30 minutes, 20 points)

Name:

x500:

Discussion Start Time (**circle one**): 3:35 4:40 5:45 6:50 7:55 other:

---

1. (2 points each) True/False - Circle one. Note that when asking about the properties of an algorithm, we specifically mean the version of that algorithm discussed in lecture.

   True **False** All comparison sorts have a Θ(n lg n) worst case runtime.
   <All comparison sorts have a Ω(n lg n) worst case runtime, but that is not always a tight bound. For example, Insertion Sort has a $\Theta(n^2)$ worst case runtime>

   True **False** If Insertion Sort is used as the algorithm within Radix Sort to sort by each digit, Radix Sort may fail: the output array may not be in sorted order.
   <Insertion Sort is stable, so it will function correctly as the inner sort used for each digit within Radix Sort, even if it's not very efficient>

   True False When using a hash table implemented with a doubly-linked-list that resolves collisions by chaining, the worst case runtime of deletion is Θ(1).
   <Deletion is assumed to take as input a pointer to the object being deleted, so we simply need to find the elements on either side of the target element within the linked list, and attach them to each other. This has no dependence on the number of other elements within the table>

   True False The hash function $h(k, i) = (k^2 + i) \bmod m$ is an example of linear probing.
   <This is a linear function with respect to i, so it qualifies as linear probing>

   True False In a binary search tree where all nodes have distinct keys, the node with the minimum key can't have a left child.
   <If the minimum key had a left child, then it would need to be smaller than the minimum key to obey the binary search tree property and still be distinct. This is a contradiction.>

2. (6 points) Fill out this table. Assume that all elements in the array are distinct (except when specified otherwise) positive integers between 1 and k. Runtimes should be expressed in terms of n and k. Assume that Radix Sort uses Counting Sort for each digit.

	Runtime when array already sorted in increasing order	Runtime when array already sorted in decreasing order	Runtime when all elements in array are the same	Runtime for randomly distributed array	In place?	Stable?
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	Yes	Yes
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	No	Yes
Quicksort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \lg n)$	Yes	No
Heapsort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n)$	$\Theta(n \lg n)$	Yes	No
Counting Sort	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$	No	Yes
Radix Sort	$\Theta(n*\log(k))$	$\Theta(n*\log(k))$	$\Theta(n*\log(k))$	$\Theta(n*\log(k))$	No	Yes

3. (5 points) In the hash table below, show the result of inserting the keys 7, 1, and 8 using open addressing, with both the linear probing function h(k,i) = (k + 3i) mod 6, and the quadratic probing function h(k,i) = (k + i^2) mod 6.

   Then, suppose that the next value to be inserted (after 7, 1, and 8) will be a randomly chosen integer x in the range 100 to 699, inclusive. Compute the probability, as either a fraction or a percentage, that x will be placed into each of the slots in the table for each probing function. You must also compute the probability that x will not be able to be inserted into the table no matter what i value is chosen.

Slot	Key (linear) h(k,i)=(k+3i) mod 6	Key (quadratic) h(k,i)=(k+i^2) mod 6	Probability that x will be inserted here (linear)	Probability that x will be inserted here (quadratic)
0			1/6	2/6
1	7	7	0/6	0/6
2	8	1	0/6	0/6
3		8	1/6	0/6
4	1		0/6	2/6
5			2/6	2/6
Can't be inserted			2/6	0/6

<Linear insertion:
7: h(7,0) = (7+3(0)) mod 6 = 1
1: h(1,0) = (1+3(0)) mod 6 = 1 (collision)
h(1,1) = (1+3(1)) mod 6 = 4
8: h(8,0) = (8+3(0)) mod 6 = 2

Quadratic Insertion;
7: h(7,0) = (7+$(0)^2$) mod 6 = 1
1: h(1,0) = (1+$(0)^2$) mod 6 = 1 (collision)
h(1,1) = (1+$(1)^2$) mod 6 = 2
8: h(8,0) = (8+$(0)^2$) mod 6 = 2 (collision)
h(8,1) = (8+$(1)^2$) mod 6 = 3

Probabilities: Note that there are exactly 100 integers between 100 and 699 inclusive that are 0 mod 6, 100 that are 1 mod 6, 100 that are 2 mod 6, and so on. This means that out of the 600 options, there is a 1/6 chance that the randomly chosen number will be x mod 6 for x = {0, 1, 2, 3, 4, 5}.
Linear Insertion:

- 0 mod 6: ((0 mod 6) + 3(0)) mod 6 = 0
- 1 mod 6: ((1 mod 6) + 3(0)) mod 6 = 1 (collision)
  - ((1 mod 6) + 3(1)) mod 6 = 4 (collision)
  - ((1 mod 6) + 3(2)) mod 6 = 1 (collision, cycle repeats, this can't be inserted)
- 2 mod 6: ((2 mod 6) + 3(0)) mod 6 = 2 (collision)
  - ((2 mod 6) + 3(1)) mod 6 = 5
- 3 mod 6: ((3 mod 6) + 3(0)) mod 6 = 3
- 4 mod 6: ((4 mod 6) + 3(0)) mod 6 = 4 (collision)
  - ((1 mod 6) + 3(1)) mod 6 = 1 (collision)
  - ((4 mod 6) + 3(2)) mod 6 = 4 (collision, cycle repeats, this can't be inserted)
- 5 mod 6: ((5 mod 6) + 3(0)) mod 6 = 5

Probabilities: 1/6 land on 0, 1/6 land on 3, 2/6 land on 5, 2/6 can't be inserted
Quadratic Insertion:

- 0 mod 6: ((0 mod 6) + $(0)^2$) mod 6 = 0
- 1 mod 6: ((1 mod 6) + $(0)^2$) mod 6 = 1 (collision)
  - ((1 mod 6) + $(1)^2$) mod 6 = 2 (collision)
  - ((1 mod 6) + $(2)^2$) mod 6 = 5
- 2 mod 6: ((2 mod 6) + $(0)^2$) mod 6 = 2 (collision)
  - ((2 mod 6) + $(1)^2$) mod 6 = 3 (collision)
  - ((2 mod 6) + $(2)^2$) mod 6 = 0
- 3 mod 6: ((3 mod 6) + $(0)^2$) mod 6 = 3 (collision)
  - ((3 mod 6) + $(1)^2$) mod 6 = 4
- 4 mod 6: ((4 mod 6) + $(0)^2$) mod 6 = 4
- 5 mod 6: ((5 mod 6) + $(0)^2$) mod 6 = 5

Probabilities: 2/6 land on 0, 2/6 land on 4, 2/6 land on 5>

4. (4 points) Why do we need a Build-Max-Heap function to initialize the max heap at the beginning of Heapsort, when we could accomplish the same goal by just calling Max-Heap-Insert n times?

(This is substantially more in-depth than necessary, you really only need the last sentence).

Build-Max-Heap is $\Theta(n)$ time in the worst case, since it is running max-heapify, which takes $\Theta(h)$ time in the worst case where h is the height of the node it's run on, at most $2^{(\lg n)-h} = n/2^h$ times at a given height h. This means that the total runtime can be computed as the sum of $(nh*2^{-h})$ from h=1 to h=lg(n), and that summation converges to <= 2n. Running Max-Heap-Insert n times, on the other hand, is in the worst case going to be $\Theta(\lg(n) - h)$ for a node at height h, because the worst case is inserting something at a leaf node that rises all the way to the top of the heap, and this can happen for every leaf node if we insert things in ascending order. This means that we are instead taking the sum of $(n(\lg(n)- h)*2^{-h})$ from h=1 to h=lg(n), which is dominated by the h = 0 term which yields n lg n. Essentially, the case of having to traverse the entire height of the tree can happen for every leaf node with this strategy instead of just at the root, which is (n/2) nodes, so that's n lg n.

So we use Build-Max-Heap instead of Max-Heap-Insert n times because it's $\Theta(n)$ time total instead of $\Theta(n \lg n)$ in the worst case.