Instructions:

1. There are test questions on the front and the back of each sheet.

2. This is a closed book exam. Do not use any notes, books, or neighbors except your one page, two side, handwritten, “hack” sheet which MUST be turned in with your exam.

3. Show your work. Partial credit will be given. Grading will be based on correctness and clarity.

4. You have 75 minutes to complete the exam. Watch your time appropriately. You should take about 15 minutes per question section.

Integrity: The University of Richmond’s Honor Code is “We, the students of the University of Richmond, shall promote and uphold a community of integrity and trust.” Upon accepting admission to University of Richmond, a student immediately assumes a commitment to uphold the Honor Code, to accept responsibility for learning, and to follow the philosophy and rules of the Honor System. Students will be required to state their commitment on examinations, research papers, and other academic work. Ignorance of the rules does not exclude any member of the Richmond community from the requirements or the processes of the Honor System.

I agree to uphold this commitment and produce original work in this exam, i.e., I will not cheat nor will I consciously let anyone cheat.

Signature: ________________________________

DO NOT BEGIN THE EXAM UNTIL INSTRUCTED TO DO SO. GOOD LUCK!
1. (20 points) **True or False.**

**Circle** the correct answer for each question.

(a) **True** or **False**: Assuming equivalent theoretical complexities, array-based implementations of data structures and algorithms often outperform linked structure-based implementations in practice because of the memory hierarchy in modern computer architectures.

(b) **True** or **False**: The functions of the Queue ADT all run in $O(n)$ time.

(c) **True** or **False**: The functions of a singly-linked-list-based implementation of the Stack ADT can all be implemented to run in $O(1)$ time.

(d) **True** or **False**: The functions of a circular-array-based implementation of the queue ADT can all be implemented to run in $O(n^2)$ time.

(e) **True** or **False**: The functions of an singly-linked-list-based implementation of the deque ADT can all be implemented to run in $O(1)$ time.

(f) **True** or **False**: The List ADT provides access to elements using Positions, a mechanism for representing an element’s “position” in a sequence.

(g) **True** or **False**: The tree shown above has size 9 and height 4, and has 1 root, 5 internal nodes, and 4 leaves (external nodes).

(h) **True** or **False**: A pre-order traversal of the tree shown above could visit the nodes in the order: I, J, K, L, O, P, M, N, and Q.

(i) **True** or **False**: A pre-order traversal of the tree shown above could visit the nodes in the order: I, K, Q, O, J, L, P, M, and N.

(j) **True** or **False**: A post-order traversal of the tree shown above could visit the nodes in the order: M, Q, P, J, O, L, N, K and I.
2. (20 points) **Short Answer.**

Provide the best answer you can for all the questions below.

(a) One fundamental storage technique, a linked structure, encodes adjacency using pointers. If next and previous adjacency is stored, the structure is a(n) **Doubly-linked list**. A special case of this, **Circularly linked lists**, occurs when the first and last elements are marked adjacent.

(b) Express the function \( f(n) = 10,000\sqrt{n} + 0.0012 n^8 \) as concisely as possible in terms of Big-Oh \( (O()) \) notation \( O(n^8) \).

Express the function \( f(n) = 8 + \log_3 n + 3n^2 \sqrt{n} + 4n \log n \) as concisely as possible in terms of Big-Oh \( (O()) \) notation \( O(n^2 \sqrt{n}) \).

(c) If a Deque (double-ended queue) ADT is implemented using a doubly-linked list, then in the **worst case** a removeLast() operation takes time \( O(1) \). If the Deque is implemented using a growable circular-array, then in the **average case** a removeLast() operation takes time \( O(1) \) for a Deque of size \( n \).

(d) If a Positional List ADT is implemented using an expandable non-circular array, then an addFirst(\( e \)) operation takes in the best case \( O(n) \) time, in the average case \( O(n) \) time, and in the worst case \( O(n) \) time, assuming implementation uses an incremental-strategy for resizing an array.

(e) If the List ADT is implemented using a singly linked-list, then in the **best case** a get(\( i \)) operation takes time \( O(1) \) and in the **average case** an add(\( i, e \)) operation takes time \( O(n) \).
3. (10 points) **Stacks, Queues, Deques.**

(a) (6 points) Write pseudocode for and describe an algorithm that reverses the elements of a Queue.

**Algorithm** REVERSE(Q)

**Input:** Queue Q
1: Stack S ← ⌀
2: while ¬Q.isEmpty() do
3:   S.push(Q.dequeue())
4: while ¬S.isEmpty() do
5:   Q.enqueue(S.pop())

Essentially, the algorithm takes all elements of the queue and puts them in a stack. Then takes the elements of the stack and puts them back in the queue. This reverses the elements because of the FILO ordering of a stack.

(b) (2 points) Let any Stack, Queue, or Deque used in the algorithm be implemented efficiently, such that all methods of their ADTs run in \( O(1) \)-time. What is the time complexity of your algorithm on a Queue of size \( n \)? Do not provide justification.

\( O(n) \)

(c) (2 points) Let any Stack, Queue, or Deque used in the algorithm be implemented inefficiently, such that all method of their ADTs run in \( O(1) \)-time, except for size() and isEmpty() which run in \( O(n) \)-time. What is the time complexity of your algorithm on a Queue of size \( n \)? Do not provide justification.

\( O(n^2) \)
4. (20 points) Lists. Show how to perform a binary search with the List ADT. If you are unfamiliar, a binary search examines the middle element of a sorted set to quickly eliminate half of the set from consideration in the search. If the key is greater than the middle element, then the “lower” half of the set can be disregarded, otherwise the “top” half of the set is disregarded. Binary search examines $O(\log n)$ elements of the set (careful though, I did NOT say $O(\log n)$-time).

(a) (10 points) Write a recursive approach in pseudocode for binary search of a List $L$ for a Key $k$ (the target element). The binary search should determine if $k$ exists in $L$. Assume the elements are comparable, and to simplify this concept in pseudocode use any of: $<$, $>$, $\leq$, $\geq$, $=$, and/or $\neq$. Do not provide an explanation for the pseudocode.

**Algorithm binarySearch** $(L, k)$

**Input:** List $L$, Key $k$

**Output:** true if $k \in L$, false otherwise

1: return binarySearch $(L, k, 0, L.size() - 1)$

**Algorithm binarySearch** $(L, k, l, h)$

**Input:** List $L$, Key $k$, Indices $l, h$

**Output:** true if $k \in L$, false otherwise

1: if $l > h$ then
2: return false
3: $m \leftarrow \frac{l + h}{2}$
4: Element $mid = L.get(m)$
5: if $mid = k$ then
6: return true
7: else if $mid < k$ then
8: return binarySearch $(L, k, l, m - 1)$
9: else
10: return binarySearch $(L, k, m + 1, h)$
(b) (4 points) Assume the List ADT is implemented with an array. State and justify the time complexity of your approach for a List of size $n$.

$O(\log n)$. Essentially, because we have access to any element with GET($i$) in constant time, and the fact that in the worst case there will be $O(\log n)$ examinations, the total is $O(\log n)$.

(c) (4 points) Assume the List ADT is implemented with a doubly-linked list. State and justify the time complexity of your approach for a List of size $n$.

$O(n \log n)$. Because each access with GET($i$) takes in the worst case $O(n)$, and the fact that in the worst case there will be $O(\log n)$ examinations, the total is $O(n \log n)$. If we have access to the internals of the doubly-linked list, we can improve this to $O(n)$-time.

(d) (2 points) State and justify the memory complexity of your approach for a List of size $n$.

$O(\log n)$. The recursive depth is proportional to the number of examinations.
5. (30 points) **Tree Traversal.**

A level-order tree traversal is one that visits all positions at level $l$ before visiting any position at level $l + 1$.

(a) (6 points) **Properties.** What is the upper and lower bound for the number of levels $l$ in the following types of trees? Define the bounds in terms of the tree’s size $n$.

- A general tree: $2 \leq l \leq n$
- A proper binary tree: $\log(n + 1) \leq l \leq \frac{n + 1}{2}$
- An improper binary tree: $\log(n + 1) \leq l \leq n$

(b) (10 points) **Pseudocode.** Write a level-order traversal for a general tree $T$. Let $\text{visit}(p)$ be the generic visit action for the traversal.

**Algorithm** `LEVELORDER(T)`

**Input:** Tree $T$

1. Queue $Q \leftarrow \{T.\text{ROOT}()\}$
2. while $\neg Q.\text{isEmpty}()$ do
3.   Position $p \leftarrow Q.\text{DEQUEUE}()$
4.   visit($p$)
5.   for all Positions $c \in T.\text{CHILDREN}(p)$ do
6.     $Q.\text{ENQUEUE}(c)$

(c) (4 points) **Correctness.** Explain your pseudocode above and describe why it performs a level-order traversal.

Essentially, we enqueue each new level at the end of the queue. So the current level is at the front of the queue. Its children by definition are on the next level.
(d) (2 points) **Time Complexity.** State but do not justify the time complexity of your algorithm for a tree of size \( n \), assuming \( \text{visit}(p) \) occurs in \( O(1) \)-time.

\[ O(n) \]

(e) (4 points) **Memory Complexity: Best Case.** State and justify the best case memory complexity of your approach for a tree of size \( n \), assuming \( \text{visit}(p) \) occurs in \( O(1) \)-memory.

\( O(1) \). In the best case, each level of the tree contains a constant number of nodes, specifically 1. So the queue never contains more than a single node in it.

(f) (4 points) **Memory Complexity: Worst Case.** State and justify the worst case memory complexity of your approach for a tree of size \( n \), assuming \( \text{visit}(p) \) occurs in \( O(1) \)-memory.

\( O(n) \). In the worst case, \( n - 1 \) nodes are located on the second level of the tree. In this case, the queue will contain all positions of the tree. Thus, in total it required \( O(n) \) memory.
6. (10 points) **Bonus.** Let $T$ be a binary tree with $n$ positions, and, for any position $p$ in $T$, let $d_p$ denote the depth of $p$ in $T$. The **distance** between two positions $p$ and $q$ in $T$ is $d_p + d_q - 2 \times d_a$, where $a$ is the lowest common ancestor (LCA) of $p$ and $q$. The **diameter** of $T$ is the maximum distance between two positions in $T$. Describe an efficient algorithm for finding the diameter of $T$. What is the running time of your algorithm? Justify your complexity.

**Algorithm `diameter(T)`**  
**Input:** Binary tree $T$  
**Output:** Diameter of the tree  
1: `return diameter(T, T.root())`

**Algorithm `diameter(T, n)`**  
**Input:** Binary tree $T$ and position $n$  
**Output:** Diameter of the subtree rooted at $n$  
1: if `T.isExternal(n)` then  
2: $n.height \leftarrow 0$  
3: `return 0`  
4: else  
5: Position $l \leftarrow T.left(n)$  
6: Position $r \leftarrow T.right(n)$  
7: $d_l \leftarrow diameter(T, l)$  
8: $d_r \leftarrow diameter(T, r)$  
9: $n.height \leftarrow 1 + \max(l.height + r.height)$  
10: `return \max(d_l, d_r, l.height + r.height + 1)`

This algorithm runs in $O(n)$ time, as it is a post-order traversal that does $O(1)$ work for each visit.