Name: __________ Key __________ Section: __________

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, cheat 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!

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>received</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>Bonus</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
</tr>
</tbody>
</table>
1. (20 points) **True or False.**

Circle the correct answer for each question.

(a) True or [False]: The functions of the stack ADT all run in $O(1)$ time.

(b) [True] or False: The functions of an array-based implementation of the stack ADT can all be implemented to run in amortized $O(1)$ time.

(c) [True] or False: The functions of a doubly-linked-list-based implementation of the queue ADT can all be implemented to run in $O(n)$ time.

(d) 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.

(e) [True] or False: The List ADT provides access to elements using indices, a mechanism for describing the location of an element in a sequence.

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

(g) True or [False]: A pre-order traversal of the tree shown above could visit the nodes in the order: A, C, F, G, B, E, H, I, and D.

(h) [True] or False: A pre-order traversal of the tree shown above could visit the nodes in the order: A, B, C, D, E, F, G, H, and I.

(i) True or [False]: A post-order traversal of the tree shown above could visit the nodes in the order: H, I, E, D, B, F, G, C and A.

(j) True or [False]: Sorting with a Priority Queue (PQ-sort) consists of performing an alternating sequence of a single insert(e)s and a single removeMin() operation.
2. (20 points) **Short Answer.**

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

(a) If a Deque (double-ended queue) ADT is implemented using a doubly-linked list, then in the worst case an `removeLast()` operation takes time \( O(1) \). If the Deque is implemented using a singly-linked list, then in the best case an `removeLast()` operation takes time \( O(n) \) for a Deque of size \( n \).

(b) If a Positional List ADT is implemented using a expandable circular array, then an `addFirst(e)` operation takes in the best case \( O(1) \) 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.

(c) If the List ADT is implemented using a linked-list, then in the worst case an `get(i)` operation takes time \( O(n) \) and in the average case an `add(i, e)` operation takes time \( O(n) \).

(d) When a PQ is implemented with a sorted sequence, the PQ-sort algorithm is referred to as **Insertion Sort** (a type of sorting algorithm) and runs in time \( O(n^2) \), and when a PQ is implemented with an unsorted sequence, the PQ-sort algorithm is referred to as **Selection Sort** (a type of sorting algorithm) and runs in time \( O(n^2) \).

(e) When a PQ is implemented with a heap, which is in turn realized by an array-based binary tree implementation, the `insert(e)` operation takes time \( O(\log n) \) and the `removeMin()` operation takes time \( O(\log n) \). In this case, the PQ-sort algorithm is referred to as **Heap Sort** (a type of sorting algorithm) and runs in time \( O(n \log n) \).
3. (20 points) **Tree Properties.**

(a) (5 points) Draw a general tree of size 5 with the minimum number of levels. Draw a general tree of size 4 with the maximum number of levels.

```
            1
           /|
          / |
         /  |
        /   |
```

(b) (5 points) What is the minimum number of levels $l$ of a proper binary tree in terms of its size $n$ and $l$, i.e., write $l$ as a function of $n$?

\[ l = \log(n + 1) \]

(c) (5 points) Prove your above equation using induction on the size of the proper binary tree $n$.

**Proof.** **Base:** In a tree of size 1, i.e., just a root node, then $l = \log(1 + 1) = 1$.

**Inductive Hypothesis:** Assume $l = \log(n + 1)$ for a proper binary tree of size $n$.

**Inductive Step:** Then, we must show that $l' = \log(n' + 1)$ for a proper binary tree of size $n' = n + 1$. To show this, there are two cases to consider:

(i) When we add a node, the level increases by 1, i.e., $l' = l + 1$. Then, by substitution, $l + 1 = \log(n + 2)$, which must be true if the level actually increased.

(ii) When we add a node, the level does not increase, i.e., $l' = l$. Then, by substitution, $l = \log(n + 2)$, which is true because of integer rounding in computation.

(d) (5 points) What is the maximum number of levels $l$ of an improper binary tree in terms of its size $n$, i.e., write $l$ as a function of $n$?

\[ l = n \]
4. (20 points) **Trees.**

Describe an algorithm for computing and storing a path at each node to the root of the tree. That is, describe a special traversal technique which stores a List of nodes being an ordered path back to the root for each node. As an example, the path from node e in the following tree would be a List \( L = \{e, b, a\} \). You may assume each node of the tree contains a member variable called `path` for storage.

```
(8 points) **Pseudocode.** Describe your pseudocode for your traversal.

**Algorithm path_traverse(T)**

**Input:** Tree \( T \)
1: **path_traverse(T.T[root])**

**Algorithm path_traverse(n)**

**Input:** Node \( n \)
1: List \( L \) \( \leftarrow \) \( \emptyset \)
2: Node \( m \) \( \leftarrow n \)
3: while \( m \neq \text{nullptr} \) do
4: \( L.I N S E R T B A C K (m) \)
5: \( m \leftarrow m.P A R E N T () \)
6: \( n.path \leftarrow L \)
7: for \( c \in n.C H I L D R E N () \) do
8: \( \text{path_traverse}(c) \)
```

(b) (2 points) **Correctness.** Explain your pseudocode above and describe why it computes the paths correctly.

The above algorithm begins by calling the recursive function, which is a preorder traversal of the general tree structure. This ensures that it visits every node. For each node I compute the path back to the list storing it in a List. Based on this, the algorithm computes what it should.
(c) (2 points) Questions. What type of traversal technique is your algorithm (or is it a combination of multiple types)? What other data structures involved in your algorithm besides the obvious Tree and List?

Pre-order traversal. I am also using a Stack through the run-time system.

(d) (4 points) Time Complexity. Quantify the time complexity for your algorithm. Be sure to justify your complexity.

\(O(nh)\). The traversal visits each node, i.e., \(O(n)\), and performs a path computation at each visit. The path computation takes time \(O(h)\) because it must traverse to the height of the tree. Thus the total is \(O(nh)\).

(e) (4 points) Space Complexity. Clearly, the tree itself has storage \(O(n)\) for each node of the tree. (a) How much extra space does your algorithm use for list storage? (b) How much extra space does your algorithm use on the run-time stack? Please justify your complexities.

(a) \(O(nh)\). Each list will have length \(O(h)\) because it is a path to the root, and there are \(n\) vertices.

(b) \(O(h)\). The run-time stack will only use \(O(h)\) because the recursive depth of the function is maximum the height of the tree.
5. (20 points) **Priority Queues.** Show how to implement the Stack ADT using only a minimum priority queue and at most one additional data member.

(a) (5 points) **General approach.** Describe the general approach, assumptions, and extra data.

My approach to ensure the FILO ordering that a stack produces is to use the negation of the size of the priority queue as the element’s key. (specifically, the size at item insertion). This ensures that the first element has the greatest key, and is thus the last one removed, and so forth. I will use no extra data members. Let \( PQ \) be the underlying priority queue.

(b) (5 points) **Push.** Describe your algorithm for push in pseudocode. What is the time complexity of this operation if the priority queue is implemented using an array-based heap?

**Algorithm** \textsc{push}(e) \hspace{1cm} \( O(\log n) \)

\begin{align*}
\textbf{Input:} & \text{ Element } e \\
1: & PQ.\text{insert}(-PQ.\text{size}(), e)
\end{align*}

(c) (5 points) **Pop.** Describe your algorithm for pop in pseudocode. What is the time complexity of this operation if the priority queue is implemented with a sorted list?

**Algorithm** \textsc{pop}() \hspace{1cm} \( O(1) \)

\begin{align*}
1: & PQ.\text{removeMin}()
\end{align*}

(d) (5 points) **Top.** Describe your algorithm for top in pseudocode. What is the time complexity of this operation if the priority queue is implemented with an unsorted list?

**Algorithm** \textsc{top}() \hspace{1cm} \( O(n) \)

\begin{align*}
1: & PQ.\text{min}()
\end{align*}
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** $\text{Diameter}(T)$

**Input:** Tree $T$

**Output:** Diameter of the tree

1: return $\text{Diameter}(T.\text{root})$

**Algorithm** $\text{Diameter}(n)$

**Input:** Tree node $n$

**Output:** Diameter of the subtree rooted at $n$

1: if $n = \emptyset$ then
2: return $-1$
3: $l \leftarrow n.\text{LEFT}()$
4: $r \leftarrow n.\text{RIGHT}()$
5: return $\max(\text{Diameter}(l), \text{Diameter}(r), d_l + d_r + 2)$

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