CSCE 221: Data Structures and Algorithms
Fall 2015
Exam 1

Name: Key UIN: Section: 

Instructions:

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

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

3. You have 50 minutes to complete the exam. Watch your time appropriately. You should take about 10 minutes per question section.

Integrity: The Aggie Honor Code is An Aggie does not lie, cheat, or steal or tolerate those who do. Upon accepting admission to Texas A&M University, 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 TAMU community from the requirements or the processes of the Honor System.

I agree to uphold this commitment and produce original work in this exam.

Signature: 

DO NOT BEGIN THE EXAM UNTIL INSTRUCTED TO DO SO. GOOD LUCK!

<table>
<thead>
<tr>
<th>Question</th>
<th>Points received</th>
<th>possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</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 a singly-linked-list-based implementation of the deque ADT can all be implemented to run in $O(1)$ time.

(e) **True** or **False**: The Sequence ADT generalizes the Vector ADT and the List ADT, and provides access to elements using both indices and positions.

(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 continually alternating `insert(e)` and `removeMin()` PQ operations.
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 `eraseBack()` operation takes time \(O(1)\). If the Deque is implemented using a singly-linked list, then in the **best case** an `eraseBack()` operation takes time \(O(n)\) for a Deque of size \(n\).

(b) If a List ADT is implemented using an expandable circular array, then an `insertFront(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 a Vector ADT is implemented using a linked-list, then in the worst case an `at(i)` operation takes time \(O(n)\) and in the average case an `insert(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 a vector-based structure of binary trees, 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 \(O(n \log n)\).

(f) What most interests you about computer science? and/or why are you majoring in a computing field?
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.

(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.

$$\square$$

(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 $n$?

$$l = n$$

5
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 \( L = \{e, b, a\} \).

![Tree Diagram]

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

**Algorithm** `PATH_TRAVERSE(T)`

**Input:** Tree \( T \)

`PATH_TRAVERSE(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
   1. \( L.\text{INSERTBACK}(m) \)
   2. \( m \leftarrow m.\text{PARENT}() \)
   3. \( n.\text{path} \leftarrow L \)
4. for \( c \in n.\text{CHILDREN}() \) do
   1. `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 are you simulating/using in your algorithm?

**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. How much extra space does your algorithm use for list storage? How much extra space does your algorithm use on the run-time stack? Please justify your complexities.

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

$O(h)$. 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 priority queue and 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 a vector-based heap?

\[
\text{Algorithm \texttt{push}(e)} \\
\text{Input: Element } e \\
PQ.\text{insert}(-PQ.\text{size}(), e) \\
O(\log n)
\]

(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?

\[
\text{Algorithm \texttt{pop}()} \\
PQ.\text{removeMin}() \\
O(1)
\]

(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?

\[
\text{Algorithm \texttt{top}()} \\
PQ.\text{min}() \\
O(n)
\]