COE 312 – Data Structures

Welcome to Exam II
Monday November 23, 2016

Instructor: Dr. Wissam F. Fawaz

Name: _____

Student ID: _____

Instructions:

1. This exam is **Closed Book**. Please do not forget to write your name and ID on the first page.
2. You have exactly **55 minutes** to complete the **4** required problems.
3. Read each problem carefully. If something appears ambiguous, please write your assumptions.
4. Points allocated to each problem are shown in square brackets.
5. Put your answers in the space provided only. No other spaces will be graded or even looked at.

Good Luck!!

Problem 1: Multiple Choice Questions (**10 minutes)** [20 points]

1) Assume `SNode<String> temp` is the tail node of a non-empty singly linked list. Which of the following conditions are true about `temp`?
   a. `(temp == tail)` evaluates to true
   b. `(temp.getNext()!=false)` evaluates to true
   c. **Both of the above**
   d. None of the above

2) Consider the following recursive method definition:

```
public String foo(String str) {
      if(str.length() == 0)
            return "";
      } else if (str.length() == 1) {
            return str.charAt(0) + "";
      }
      else {
            String str_mod = str.substring(1, str.length());
            return str.chartAt(0) + foo(str_mod);
      }
}
```

   What output does `foo` produce if a value of `"exam"` is passed as the parameter?
   a. `exa`
   b. `maxe`
   c. **`exam`**
   d. None of the above

3) When an exception occurs, it is said to have been
   a. caught
   b. **thrown**
   c. declared
   d. None of the above

4) Consider the following recursive method definition:

```
public int question4(int x, int y) {
       if (x == y) return 0;
       else return question4(x-1, y) + 1;}
```

   If the method is called as follows: `question4(8, 3)`, what is returned?
   a. 11
   b. 8
   c. **5**
   d. 3

5) Consider the recursive method from the previous question. The following method call leads to an infinite recursion: `question4(a, b)` if
   a. The value of `a` is equal to that of `b`
   b. The value of `a` is different from the value of `b`
   c. **The value of `a` is strictly less than that of `b`**
   d. None of the above

6) Which of the following is a correct Big-Oh characterization of the following summation: $\sum_{k=1}^{n} n^k$
   a. $O(n^n)$
   b. **$O(n^{n+1})$**
   c. Both of the above
   d. None of the above

7) Consider the tower of Hanoi problem. If there are 2 disks to move from the source tower to the destination tower, how many disk movements would it take to solve the problem in the recursive solution?
   a. 1
   b. 2
   c. **3**
   d. None of the above

8) Which of the following is a correct Big-Oh characterization of the running time of the following pseudo code?

```
sum ← 0
for i ← 1 to n do
     for j ← 1 to i do
          sum ← i*j
 return sum
```

   a. $O(n)$
   b. $O(\log_2 n)$
   c. **$O(n^2)$**
   d. $O(2^n)$

9) In an array-based implementation (non-circular and non-drifting) of a `queue` that always stores the rear element of the queue at index 0 in the array, the `enqueue` operation is
   a. **O(n)**
   b. O(1)
   c. Impossible to implement
   d. None of the above

10) Consider a circular drifting array-based implementation of the queue ADT with 10 items stored at `arr[2] (rear)` through `arr[11] (front)`. Assume that the capacity of the `arr` array is 12. What will be the new value of the `front` index after a `dequeue` operation is performed?
   a. 12
   b. **0**
   c. 10
   d. None of the above

Problem 2: Recursion (**15 minutes)** [20 points]

(1) Consider a method called `reverseStack` that prints the elements of a `Stack` parameter reversely. That is, the first element to be printed is the bottom of the stack and the last to be printed is the top of the stack. Implement this method in Java by using recursion. Use the following header for the method and write your code in the space provided.

```
void reverseStack(Stack stack) throws StackException{

      if(stack.size() == 0)
            return;
      else {
            Object toPrint = stack.pop();
            reverseStack(stack);
            System.out.print(toPrint + " ");
      }

 }
```

(2) Write a recursive function called `multiplyDigits` that returns the product of the digits that make up an integer parameter. For example, `sumDigits(234)` returns `24`. Note that modulo operator (%) can be used to extract the last digit while integer division can be employed to remove it as follows: 13%10 = 3 and 13/10=1.

```
int multiplyDigits(int value) {

      int lastDigit = value % 10;
      value = value /10;

      if(value == 0)
            return lastDigit;
      else
            return lastDigit * multiplyDigits(value);









}
```

Problem 3: Analysis of algorithms (**15 minutes)** [25 points]

1. Consider the following algorithm described in pseudo-code, which takes an array `A` of `n` integers as input and uses an initially-empty stack `S` as a local variable.

```
Let t <- 0;
Let S be an empty stack;
for i <- 0 to n – 1 do
   if A[i] < 0 then
        while S is not empty do
             t <- t + S.pop();
        end while
   else
        S.push(A[i]);
   end if
end for
while S is not empty do
   t <- t + S.pop();
end while
output t;
```

   a. What is the output of this algorithm for the array A = {1, -2, 3, 4, -3}?

8

   b. Describe in one sentence what this algorithm does.

This algorithm computes the sum of the positive values of array A.

   c. Characterize, using the Big-Oh notation, the running time of the above algorithm in terms of `n` under the assumption that `pop` and `push` operations require `O(n)` time each.

$O(n^2)$

2. Consider the following algorithm described in pseudo-code, which takes an array A of n positive integers as input and uses an initially-empty queue Q as a local variable:

```
let Q be an empty queue;
for i = 0 to n-1 do
     if A[i] is an odd number then
          Q.enqueue(A[i]);
     else
          while Q is not empty do
               print(Q.dequeue() + " ");
          end while
     end if
end for
while Q is not empty do
     print(Q.dequeue() + " ");
end while
```

a. Describe in one sentence what this algorithm does.

This algorithm prints all the odd values of array A out on the same line.

b. Characterize, using the Big-Oh notation, the running time of the above algorithm in terms of `n` under the assumption that each `dequeue` operation requires `O(1)` time.

O(n)

Problem 4: Queues (**15 minutes)** [35 points]

1) Write a class called `SLLBasedQueue<T>` that represents a singly linked list based queue data structure. In particular, you are required to:
    a. Create the `SinglyLinkedList<T>` data structure as well as all of its associated components (the `SNode<T>` class, `SList<T>` interface, and `EmptyListException` class).
    b. Create the queue class using the singly linked list developed in (a) such that the methods of the `Queue` ADT are implemented to run in `O(1)` time.

```
public class EmptyListException extends Exception {

	public EmptyListException(String msg) {
		super(msg);
	}
}
public interface Position<T> {
	public T getElement();

}
public class SNode<T> implements Position<T> {
	private T element;
	private SNode<T> next;
	public SNode(T element, SNode<T> next) {
		super();
		this.element = element;
		this.next = next;
	}
	public T getElement() {
		return element;
	}
	public SNode<T> getNext() {
		return next;
	}
	public void setNext(SNode<T> next) {
		this.next = next;
	}
	public void setElement(T element) {
		this.element = element;
	}
}
public interface SList<T> {
	public int size();
	public boolean isEmpty();
	public T removeFromHead() throws EmptyListException;
	public T removeFromTail() throws EmptyListException;
	public void insertAtHead(T e);
	public void insertAtTail(T e);

}
public class SinglyLinkedList<T> implements SList<T> {
	private int size;
	private SNode<T> head, tail;
	private StringBuilder sb;
	public SinglyLinkedList() {
```

```
		size = 0;
		head = tail = null;
		sb = new StringBuilder();

	}
	public int size() {
		return size;
	}

	public boolean isEmpty() {
		return (size==0);
	}
	public T removeFromHead() throws EmptyListException {
		if(isEmpty())
			throw new EmptyListException("List is empty!");

		T toReturn = head.getElement();

		if(head == tail) {
			head = tail = null;
		} else {
			head = head.getNext();
		}

		size--;
		sb.delete(0, sb.indexOf(toReturn.toString())+
				toReturn.toString().length()+1);
		return toReturn;
	}

	public T removeFromTail() throws EmptyListException {
		if(isEmpty())
			throw new EmptyListException("List is empty!!");

		T toReturn = tail.getElement();

		if(head == tail) {
			head = tail = null;
		} else {
			SNode<T> temp = head;
			while(temp.getNext() != tail)
				temp = temp.getNext();
			temp.setNext(null);
			tail = temp;
		}


		size--;
		sb.delete(sb.indexOf(toReturn.toString()),
				sb.indexOf(toReturn.toString())+
				toReturn.toString().length()+1);
		return toReturn;
	}
	public void insertAtHead(T e) {
		SNode<T> newSNode = new SNode<>(e, head);

		if(isEmpty())
```

```
                tail = newSNode;
            head = newSNode;

            size++;
            sb.insert(0, e.toString() + " ");

      }
      public void insertAtTail(T e) {
            SNode<T> newSNode = new SNode<>(e, null);

            if(isEmpty()) {
                  head = newSNode;
            } else {
                  tail.setNext(newSNode);
            }

            tail = newSNode;
            size++;
            sb.append(e.toString() + " ");

      }

      public T getHead() throws EmptyListException {
            if(isEmpty())
                  throw new EmptyListException("List is empty!!");
            return head.getElement();
      }

      public T getTail() throws EmptyListException {
            if(isEmpty())
                  throw new EmptyListException("List is empty!!");
            return tail.getElement();
      }

      public String toString() {
            return sb.toString();
      }

}
public interface Queue<T> {
      public int size();
      public boolean isEmpty();
      public T dequeue() throws EmptyListException;
      public void enqueuer(T e);
      public T front() throws EmptyListException;
}
public class SLLBasedQueue<T> implements Queue<T> {
      private SinglyLinkedList<T> sll;

      public SLLBasedQueue() {
            sll = new SinglyLinkedList<>();
      }

      @Override
      public int size() {
            return sll.size();
      }
```

```
	@Override
	public boolean isEmpty() {
		return sll.isEmpty();
	}

	@Override
	public T dequeue() throws EmptyListException {
		return sll.removeFromHead();
	}

	@Override
	public void enqueue(T e) {
		sll.insertAtTail(e);
	}

	@Override
	public T front() throws EmptyListException {
		return sll.getHead();
	}

	public String toString() {
		return sll.toString();
	}
}
```

Appendix: Classes and Methods

1. **Methods related to SinglyLinkedList<T> and SNode<T>:**

SinglyLinkedList<T>
int size() boolean isEmpty() void insertAtHead(T e) void insertAtTail(T e) T removeFromHead() throws EmptyListException T removeFromTail() throws EmptyListException T getHead() throws EmptyListException T getTail() throws EmptyListException
SNode<T>
T getElement() SNode<T> getNext() void setElement(T e) void setNext(SNode<T> n)