UNIVERSITI TUNKU ABDUL RAHMAN

ACADEMIC YEAR 2019/2020

MID TERM TEST

UECS2083/UECS2413 PROBLEM SOLVING WITH DATA STRUCTURE AND ALGORITHMS

SATURDAY, 20TH JULY 2019 TIME :4.30 PM – 5.30 PM (1 HOURS)

BACHELOR OF ENGINEERING (HONOURS) ELECTRICAL AND ELECTRONIC ENGINEERING
BACHELOR OF SCIENCE (HONS) APPLIED MATHEMATICS WITH COMPUTING
BACHELOR OF SCIENCE (HONS) SOFTWARE ENGINEERING

Instruction to Candidates:

This question paper consists of 6 questions.

Answer all the questions in this question paper.

Student ID : ____

Name : ____

Course : ____

Lecture Group : ____

	MARKS
Question 1	
Question 2	
Question 3	
Question 4	
Question 5	
Question 6	
TOTAL:	

Question 1

(a) Write a **recursive method** for Ackermann function, $a(m,n)$ as defined below:

$$a(m,n) = \begin{cases} n+1 & if\ m = 0 \\ a(m-1,1) & if\ m > 0\ and\ n = 0 \\ a(m-1,\ a(m,n-1)\,) & if\ m > 0\ and\ n > 0 \end{cases}$$

(6 marks)

Answer:

Marking guideline:

- Recursive method header [1]
  ex:public static int RecursiveAckerman(int m, int n)
- Base case 1:
  - o if m==0 [1]
  - o return (n+1) [1]
- Base case 2:
  - o if (m > 0 && n == 0) [1]
  - o return *RecursiveAckerman*(m - 1, 1) [1]
- Recursive call:
  - o return *RecursiveAckerman*(m - 1, [1/2]
    RecursiveAckerman(m, n - 1)) [1/2]

```
public static int RecursiveAckerman(int m, int n) {
        if (m == 0) {
            return (n + 1);
        } else if (m > 0 && n == 0) {
            return RecursiveAckerman(m - 1, 1);
        } else {
            return RecursiveAckerman(m - 1,
                          RecursiveAckerman(m, n - 1));
        }
    }
```

(b) Describe the characteristics of recursive method. (2 marks)

Answer:

1. One or more base cases (the simplest case) are used to stop recursion.
2. Every recursive call reduces the original problem, bringing it increasingly close to a base case until it becomes that case.
3. The method is implemented using an if-else or a switch statement that leads to different cases.

In general, to solve a problem using recursion, you break it into subproblems. If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively. This subproblem is almost the same as the original problem in nature with a smaller size.

(c) Explain the cause of stack-overflow exception. (2 marks)

Answer:
When a method is invoked, its contents are placed into a stack. If a method is recursively invoked, it is possible that the stack space is exhausted. This causes stack overflow.

If recursion does not reduce the problem in a manner that allows it to eventually converge into the base case or a base case is not specified, infinite recursion can occur. The method runs infinitely and causes a StackOverflowError.

Question 2

(a) What is erasure ? (2 marks)

Answer:
Generic type information is used by the compiler to check whether the type is used safely. Afterwards the type information is erased. The type information is not available at runtime.

(b) If the compiler erases all type parameters at compile time, why should you use generics ? (2 marks)

Answer: (any 1 answer– 2m)
You should use generic because:
a) The java compiler enforces tighter type checks on generic code at compile time or potential errors can be detected by the compiler
b) Improve reliability and robustness
c) Generic support programming types as parameters
d) Generic enable you to implement generic algorithms
e) Elimination of cast

(c) Will the following class compile? If not, why? (2 marks)

```
public final class Algorithm {
     public static <T> T max(T x, T y) {
          return x > y ? x : y;
     }
}
```

Answer:
No. The greater than (>) operator applies only to primitive numeric types.

Question 3

What data structure would you use in the following problem?

(a) You need to write a program that stores elements in a list with frequent operation to add and insert elements at the end of the list. (2 marks)

Answer:

```
ArrayList
```

(b) You need to write a program that adds element in any order but removal of the elements in a sorted order. (2 marks)

Answer:

```
Priority Queue
```

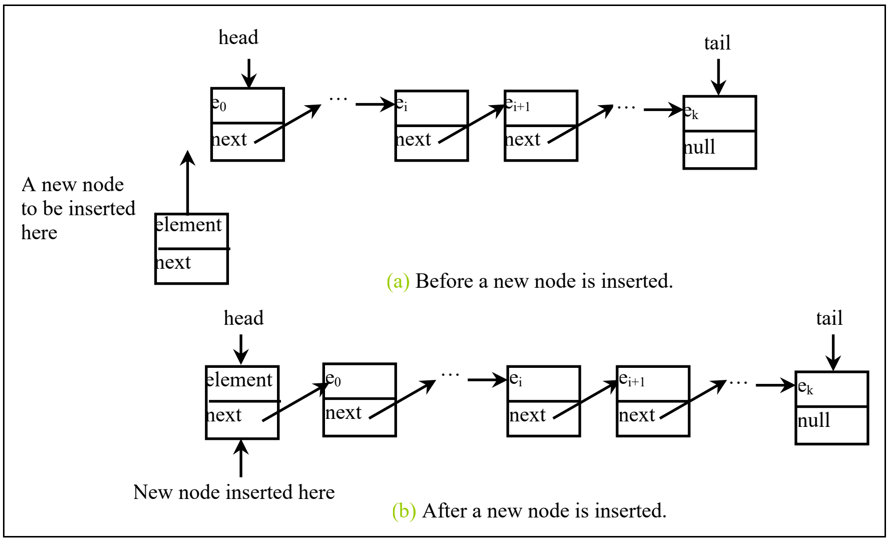
Question 4

(a) Explain the steps to insert a new node at the beginning of a linked list. (4 marks)

Answer:

```
1. Create a new node
   Node<E> newNode = new Node<E>(o);
2. making its link point to the current first node pointed
   to by head
   newNode.next = head;
3. changing head to point to this new node.
   head = newNode;
4. Increase the size
   size++;
```

```
public void addFirst(E o) {
  Node<E> newNode = new Node<E>(o);
  newNode.next = head;
  head = newNode;
  size++;
  if (tail == null)
    tail = head;
}
```



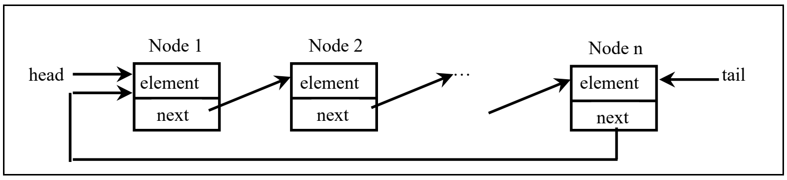
(a) Before a new node is inserted.

(b) After a new node is inserted.

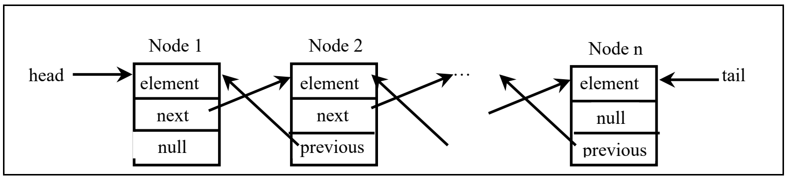
(b) Using examples, briefly explain the differences between circular linked list, doubly linked list and circular doubly linked list? (6 marks)

Answer: explanation [1mark each] examples (can be in a diagram format) [1mark each]

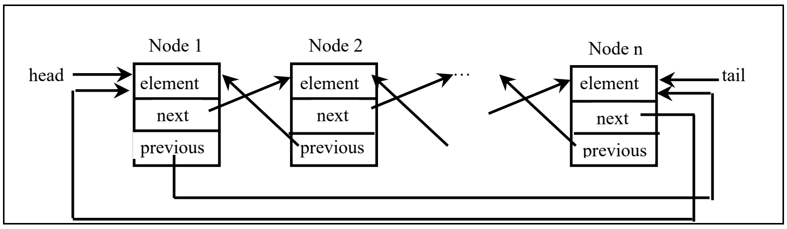
- ✦ A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node.



- ✦ A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. So, a doubly linked list can be traversed forward and backward.



- ✦ A *circular, doubly linked list* is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node



Question 5

(a) List two (2) differences between Comparable and Comparator. (4 marks)

Answer: Any 2 differences – 1m each

No.	**Comparable**	**Comparator**
1)	Comparable provides only one sort of sequence.	The Comparator provides multiple sorts of sequences.
2)	It provides one method named compareTo().	It provides one method named compare().
3)	It is found in java.lang package.	It is located in java.util package.
4)	If we implement the Comparable interface, The actual class is modified.	The actual class is not changed.

(b) Show the output from the following sequence of stack operation. (6 marks)

```
    Stack<Integer> stackA = new Stack<>();
          int x = 5;
          int y = 3;
          stackA.push(8);
          stackA.push(9);
          System.out.println(stackA.peek());
          stackA.push(y);
          System.out.println(stackA.peek());
          stackA.push(x+y);
          System.out.println(stackA.pop());
          x = stackA.peek();
          System.out.println(x);
          stackA.pop();
          stackA.push(22+x);
          System.out.println(stackA.pop());
          System.out.println(stackA.pop());
```

Answer:

Output:

```
9
3
8
3
25
9
```

Question 6

Define a class with methods described as follows. Refer to Appendix A for the UML diagram for Collection interface, LinkedList, Queue and Iterator interface.

(a) A main method that:

- Creates a queue as follows: (2 marks)
  {“RED”, “GREEN”, “BLUE”, “BLACK”, “WHITE”}
- Finds the size of the queue. (2 marks)
- Gets the first element of the queue without removing it. (2 marks)
- Tests the client methods that you will write for part (b). (2 marks)

(b) A client method that prints all the elements of the specified queue in lower case using iterator. The method header is as follows: (2 marks)

```
public static void print (Queue<String> queue)
```

Answer:

```
Marking guideline:
```

- Creates a queue as follows: {“RED”, “GREEN”, “BLUE”, “BLACK”, “WHITE”} (2 marks)

```
Queue<String> queue = new LinkedList<>();     [1]
queue.offer("RED");                           [1]
queue.offer("GREEN");
queue.offer("BLUE");
queue.offer("BLACK");
queue.offer("WHITE");
```

- Finds the size of the queue. (2 marks)

```
System.out.println(queue.size());             [2]
```

- Gets the first element of the queue without removing it. (2 marks)

```
System.out.println(queue.element());          [2]
or
System.out.println(queue.peek());             [2]
```

- Tests the client methods that you will write for part (b). (2 marks)

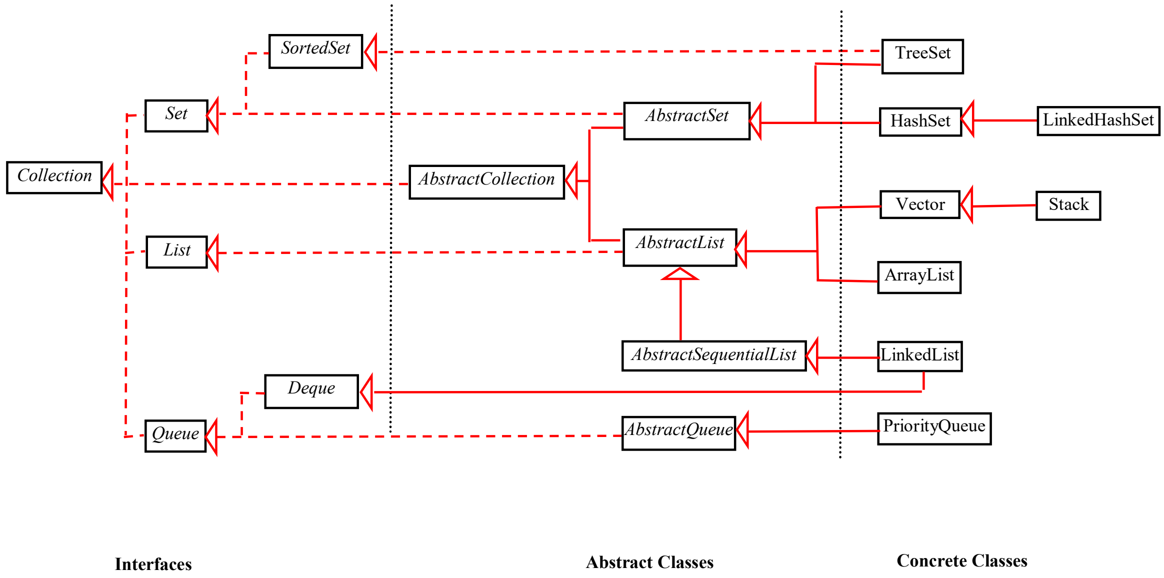
```
print(queue);                                 [2]
```

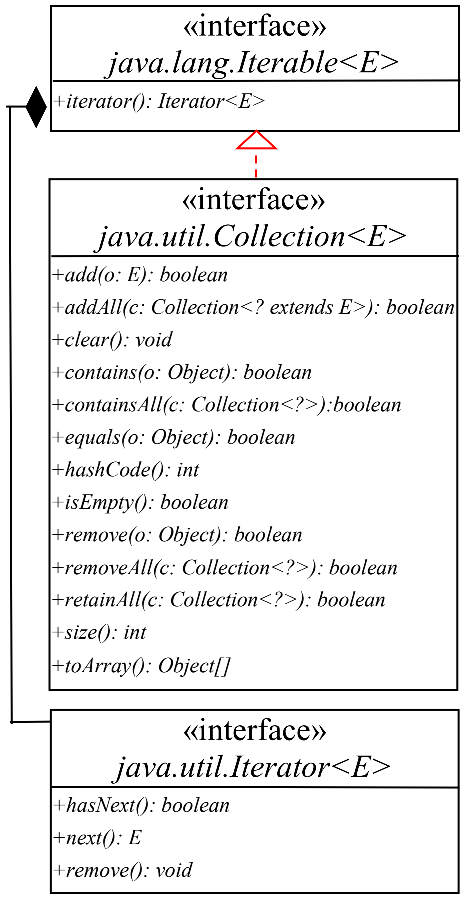
- A client method that prints all the elements of the specified queue in lower case using iterator. The method header is as follows: (2 marks)

```
public static void print(Queue<String> queue) {     [1/2]
     for(String e:queue)                            [1/2]
     System.out.print(e.toLowerCase() + "\t");      [1]
     System.out.println("");
     }
or

public static void print(Queue<String> queue) {     [1/2]
  Iterator<String> iterator = queue.iterator();     [1/2]
  While(iterator.hasNext()){                        [1/2]
  System.out.print(iterator.next().toLowerCase()); [1/2]
     }
```

```
import java.util.*;

public class testQueue {

   public static void main(String[] args) {
    Queue<String> queue = new LinkedList<>();
    queue.offer("RED");
    queue.offer("GREEN");
    queue.offer("BLUE");
    queue.offer("BLACK");
    queue.offer("WHITE");

    System.out.println("Size of queue is : " + queue.size());
    System.out.println("First element is : " + queue.element());
    print(queue);
       }

   public static void print(Queue<String> queue) {
          for(String e:queue)
          System.out.print(e.toLowerCase() + "\t");
          System.out.println("");
          }
}
```

APPENDIX A: Java Collection Framework Hierarchy, The Collection interface, LinkedList class, Queue interface and Stack Class





«interface»
java.lang.Iterable<E>

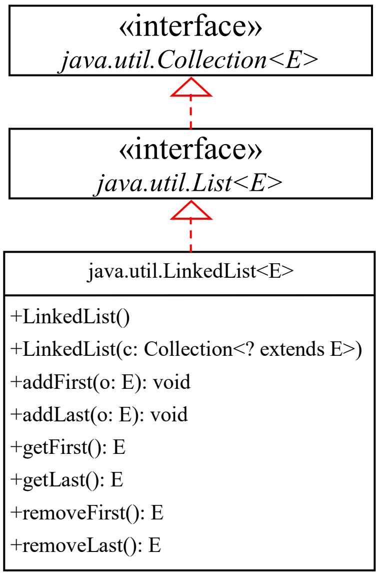
+*iterator(): Iterator<E>*	Returns an iterator for the elements in this collection.

«interface»
java.util.Collection<E>

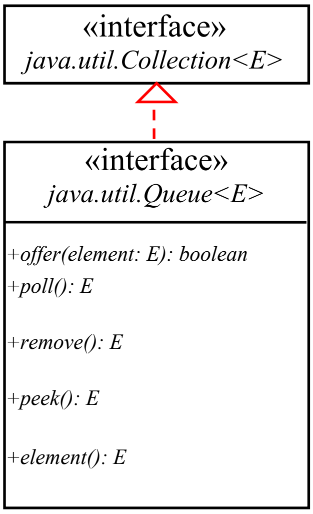
+*add(o: E): boolean*	Adds a new element o to this collection.
+*addAll(c: Collection<? extends E>): boolean*	Adds all the elements in the collection c to this collection.
+*clear(): void*	Removes all the elements from this collection.
+*contains(o: Object): boolean*	Returns true if this collection contains the element o.
+*containsAll(c: Collection<?>):boolean*	Returns true if this collection contains all the elements in c.
+*equals(o: Object): boolean*	Returns true if this collection is equal to another collection o.
+*hashCode(): int*	Returns the hash code for this collection.
+*isEmpty(): boolean*	Returns true if this collection contains no elements.
+*remove(o: Object): boolean*	Removes the element o from this collection.
+*removeAll(c: Collection<?>): boolean*	Removes all the elements in c from this collection.
+*retainAll(c: Collection<?>): boolean*	Retains the elements that are both in c and in this collection.
+*size(): int*	Returns the number of elements in this collection.
+*toArray(): Object[]*	Returns an array of Object for the elements in this collection.

«interface»
java.util.Iterator<E>

+*hasNext(): boolean*	Returns true if this iterator has more elements to traverse.
+*next(): E*	Returns the next element from this iterator.
+*remove(): void*	Removes the last element obtained using the next method.



Creates a default empty linked list.
Creates a linked list from an existing collection.
Adds the object to the head of this list.
Adds the object to the tail of this list.
Returns the first element from this list.
Returns the last element from this list.
Returns and removes the first element from this list.
Returns and removes the last element from this list.



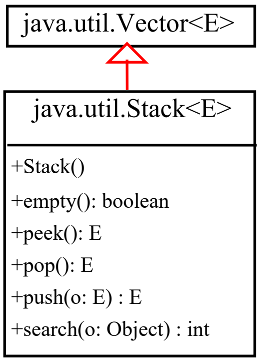
Inserts an element to the queue.

Retrieves and removes the head of this queue, or null if this queue is empty.

Retrieves and removes the head of this queue and throws an exception if this queue is empty.

Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.

Retrieves, but does not remove, the head of this queue, throwing an exception if this queue is empty.



Creates an empty stack.
Returns true if this stack is empty.
Returns the top element in this stack.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.
Returns the position of the specified element in this stack.