Q1(a) The Java application (outlined in appendix 1) is a variation on the Bounded Buffer

Producer Consumer Application. The Producer inserts items into an array

(size=4) and the Consumer deletes items from the array, ideally in the order

inserted.

For example if we insert: 1, 2, 3, 4

and then delete: 1, 2

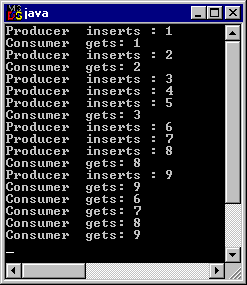
Then the situation will be as follows: **‘count’**  has a value of 2

nextout

1 2 3 4

Nextin

(i) Explain briefly why the following sequence might occur and what the

 problem is.

(4 marks)

1. Outline a new ‘synchronized’ version of the ‘LinkedStore’ class to ensure the

application is well-behaved

i.e.

- Items are deleted in the order inserted

* don’t allow items to be inserted if the array is full
* don’t allow deletion of items if array is empty

(9 marks)

(b) The following application is multithreaded and can give an inconsistent result

class Join3 extends Thread {

static int counter=0,delay=1000;

public void run(){

try {Thread.sleep(delay);}catch(Exception e){}

counter++;

}

public static void main( String[] args )

{ Join3 t1=new Join3();

Join3 t2=new Join3();

t1.start();

delay=0;

t2.start();

System.out.println("Value="+counter);

System.out.println("Completed");

}

}

1. Explain how each of the following results can occur

Value=2

Value=1

Value=0

(4 marks)

1. Show the modification required to the main method to promote a consistent result.

Value=2

(3 marks)

**[20 Marks]**

Q2(a) We want to use Hashing techniques to store a group of **N** Nodes into an array with **LEN**

Slots.

0 1 2 3 LEN-1

1. Complete the following Java code implementation for a simple Hash function that will

Map an integer key to an index value in the range 0..LEN-1

static int hash(in key){

}

(3 marks)

1. Explain what is meant by Collision Detection and demonstrate using diagrammed

examples how Linear Probing and Chaining can both be used to resolve collisions.

(4 marks)

1. For the Chaining Technique, with a size of **LEN** and a total of **N** items stored in the

HashTable, discuss with proofs, the performance characteristics for a

successful and unsuccessful search of the table.

(5 marks)

(b) Given the following incomplete Java application which uses the HashSet Collection

**public** **class** Q2b

{ **static** **boolean** search(HashSet set,**int** target){….

**static** **int** countEven(HashSet set){…

**static** **void** print(HashSet set){

Iterator i =set.iterator(); System.*out*.print("{");

**if** (i.hasNext()){ **int** element =(Integer)i.next();

System.*out*.print( " "+element );}

**while**(i.hasNext()){ **int** element =(Integer)i.next();

System.*out*.print( " ,"+element );

}

System.*out*.println("}");

}

**public** **static** **void** main( String[] args )

{ HashSet<Integer> set = **new** HashSet();

Collections.*addAll*(set , 4,5,3,2,3,1,3,8,9,11);

*print*(set);

**int** target=Console.*readInt*("Enter Target:");

System.*out*.println("Element Found="+ *search*(set,target));

System.*out*.println("No of Even Elements="+countEven(set));}

}

Outline a java implementation for the two functions

1. boolean search(HashSet set, int target) // returns true if ‘target’ is in the set
2. int countEven(HashSet set) // return the number of even elements

(8 marks)

**[20 Marks]**

Q3(a) Given the following Java class which defines a simple Container and has functions to

access the data as shown.

class Container{

private int counter;

private int max;

public Container(int c,int m){ counter=c;

max=m;

}

public int readCounter(){

return counter;

}

public int readMax(){

return max;

}

public void updateMax(int m){

max=m;

}

public void updateCounter(int c){

counter=c;

}

public int diffCounterMax(){

int res= max -counter;

return res;

}

public boolean counterGreaterThan(int v){

if(counter > v) return true;

else return false;

}

}

An object of this type can be defined and used as follows:

Container c = new Container(1,3);

System.out.println("Initial Value of Counter= "+c.readCounter());

System.out.println("Initial Value of Max= "+c.readMax());

System.out.println("Initial Difference= "+c.diffCounterMax());

System.out.println("Greater than 2: "+c.counterGreaterThan(2));

System.out.println("Greater than 0: "+c.counterGreaterThan(0));

System.out.println();

c.updateCounter(3);

c.updateMax(7);

System.out.println("New Value of Counter= "+c.readCounter());

System.out.println("New Value of Max= "+c.readMax());

System.out.println("New Difference= "+c.diffCounterMax());

System.out.println("Greater than 2: "+c.counterGreaterThan(2));

1. Use Generics to rewrite the ‘Container’ class so it can be used with both Numeric and Comparable type ‘int’, ‘float’, ‘double’ (i.e. counter and max variables should have the same generic type)
2. Outline code that defines and uses an Object of this new class.

(11 Marks)

(b) Given the following incomplete Java application which uses the ArrayList Collection

public class ArrayList1

{

static boolean allSingleDigit(ArrayList list){ **// to be completed**

}

public static void main( String[] args )

{

ArrayList<Integer> list = new ArrayList();

Collections.addAll(list , 4,5,3,2,23,1,3);

boolean res1=allSingleDigit(new ArrayList(list));

System.out.println("Only Single Digit values in list="+res1); }

}

Outine a java **recursive** implementation for a function which returns true if all the elements

in the list are single digit (i.e.<10). Can assume the list only contains positive integers

**static boolean allSingleDigit(ArrayList list)**

i.e. allSingleDigit([4,5,3,2,23,1,3]) returns false

allSingleDigit([4,5,3,2,3,1,3]) returns true

(9 Marks)

**[20 Marks]**

Q4 The following Java class includes a collection of 8 integer values records as 8 individual

attributes

**class Octo**

**{**

**private int v1,v2,v3,v4,v5,v6,v7,v8;**

**int length=6;**

**int index=0;**

**public Octo(int[] li) { v1 = li[0]; v2 = li[1]; v3 = li[2];**

**v4 = li[3]; v5 = li[4]; v6 = li[5];**

**v7=li[6]; v8=li[7];}**

**public String printList()**

**{**

**return "["+v1+"," +v2 + "," + v3 + "," + v4 +","+ v5+","+ v6+","+v7+","+v8+ "]";**

**}**

**public void gotoFirst() { index = 0; }**

**public int[] toArray(){**

**return new int[]{v1,v2,v3,v4,v5,v6,v7,v8};**

**}**

**}**

Outline the modifications necessary to use the Iterator pattern enabling clients to iterated

over this collection using the methods: iterator(), next(), hadNext()

An example of the Client using the complete iterator is shown below:

**public static int sum(Octo o){**

**int res=0;**

**Iterator value = o.iterator();**

**while(value.hasNext()){**

**res+=(Integer)value.next();}**

**return res;**

**}**

**[20 marks]**

Appendix 1

import java.io.\*;

class LinkedStore{

private int arr[]=new int[4];;

private int count;

private int nextin;

private int nextout;

public LinkedStore(){ for(int i=0;i<4;i++)

arr[i]=0;

count=0;

nextin=0;

nextout=0;}

public void insert(int val) throws Exception{

arr[nextin]=val;

nextin=nextin+1;

count++;

if (nextin>3) nextin=0;

}

public int delete() throws Exception

{int res=arr[nextout];

nextout++;

if (nextout>3) nextout=0;

count--;

return res;

}

}

class Producer extends Thread {

private LinkedStore store;

public Producer(LinkedStore ns){store=ns;}

public void run() {

for (int i = 1; i < 10; i++) {

try{ store.insert(i);}

catch(Exception e){}

System.out.println("Producer "+ " inserts : " + i);

try {

sleep((int)( Math.random()\*10)); }

catch (InterruptedException e) {} }

}

}

class Consumer extends Thread {

private LinkedStore store;

public Consumer(LinkedStore ns){store=ns;}

public void run() {

int value = 0;

for (int i = 1; i < 10; i++) {

try{ value = store.delete();}

catch(Exception e){}

System.out.println("Consumer "+ " gets: " + value);

try { sleep((int)( Math.random()\*20));}

catch (InterruptedException e) {} }

}

}

public class SlicedStack2 {

public static void main(String[] args)

{

LinkedStore ls=new LinkedStore();

Producer p=new Producer(ls);

Consumer c=new Consumer(ls);

p.start();

c.start();

try{

System.in.read();}

catch(Exception e){}

}

}