1. A classic use of linked lists is for representing and manipulating polynomials. A polynomial is a mathematical formula of the form: $a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$. This formula can be represented by a linked list using the following node structure. Each node in the list stores the information about one non-zero term of the polynomial.

(a) Give the Java type declaration for the implementation of the ADT `Polynomial`. Assume you don’t have a pre-defined ADT `LinkedList`.

(b) How would you modify your answer to question (a) if you had already an ADT `LinkedList`?

(c) Consider the following interface `PolynomialList`. Write a Java class `Polynomial` that implements the interface `PolynomialList` using the type declaration given in (b). Assuming the existence of a class `ListBaseReference` that implements a Linked List with items of type `Object`.

   ```java
   public interface PolynomialList {
       public Boolean isEmpty();
       //post: returns true if the polynomial has no terms
       public int PolynomialSize();
       //post: return the number of terms in a polynomial
       public Term getTerm(int pos);
       //post: returns the Term at position pos in the polynomial
       public int getTermExponent(int pos);
       //post: return the exponent of the term at position pos
       public int getTermCoefficient (int pos);
       //post: return the exponent of the term at position pos
       public int getMaxExponent();
       //post: return the highest exponent in the polynomial.
       public void addTerm(int exp, int coeff);
       //post: add (i.e. summing) a new term with exponent exp and coefficient coeff
       // to the polynomial
   }
   ```

(d) Write the high access procedures specified below, which sums two polynomial expressions. For instance, the sum of the two polynomials $3x^2 + 2x + 5$ and $4x^5+5x^2$ would be the polynomial $4x^5 + 8x^2 + 2x + 5$. 

```java
public PolynomialList sumPolynomials(PolynomialList p1, PolynomialList p2) {
    // Implementation
}
```
Polynomial addPolynomials(Polynomial p1, Polynomial p2)
//pre: p1 and p2 are not empty
//post: returns a new polynomial that is the sum of the two given polynomials p1 and p2.

2. An hospital waiting list of people is represented by an ADT called **PriorityQ**. It consists of a list of queues, one queue corresponding to each of 4 priority levels (1, 2, 3, 4). Within the same priority level the Queue operates as a standard FIFO queue. Each data in a queue contains the patient’s name. The list of queues is ordered. The diagram below gives an example.

(a) Give the Java type declarations for this ADT **PriorityQ** in full, including the type declarations of all the data structures needed to implement it.

(b) Write the Java code for the following access procedures **Add** (specified below), assuming the availability of the access procedures **enqueue(String newpatient)**, which adds a new patient in a given queue, and **dequeue(String newpatient)**, which deletes an existing patient from a given queue. Give also the implementation of any auxiliary method or other access procedure of the underlying data structure that might be needed.

void Add(String newname, int priority)
//pre: newname is the name of the new patient, and priority is his/her priority level.
//post: adds the new patient taking account of the priority.

void Delete(String newname, int priority)
//pre: newname is the name of the new patient, and priority is his/her priority level.
//post: deletes the new patient taking account of the priority.

(c) Assume now the existence of a standard ADT Queue. Give the Java type declaration for the ADT **PriorityQ** using this standard ADT.
PROGRAMMING II – ABSTRACT DATA TYPES
Assessed Coursework: Model Answer

1. (a) Note that in this case since there is no existing implementation of LinkedList we have to declare Polynomial in its full data structure, i.e. the reference to the ADT Polynomial, which is essentially the reference to the first term in a linked list and the polynomial size. This is the same type declaration I have used in the lecture notes for linked lists. Note that in this case the class Term defines the type of the item stored in the linked list.

```java
class Polynomial{
    private Term head;
    private int numberOfTerms;
    //access procedures
}

class Term{
    private int coefficient;
    private int exponent;
    private Term next;
}
```

(b) Note that in this case we have a Linked List implementation that we can assumed to be for items of type Object. We can therefore either declare the Polynomial as a subclass of LinkedList, or even better use the LinkedList as the type of a private attribute of the class Polynomial. This second case is given here in the solution and it consists of the ADT Polynomial using the ADT LinkedList as its underlying data structure.

The class Term needs still to be given to define the type of the items stored in the linked list. This class is as given in (a) above. The class Polynomial needs instead to be modified. In this case it uses the following data type declaration:

```java
class Polynomial{
    private ListReferenceBased ListTerms;
    // access procedures
}
```

(c) Using the type declaration given in (b) the implementation of the ADT Polynomial includes the implementation of the class Term and that of the class Polynomial. The first is given below and follows the example implementation used in the lecture notes for a class Node.

```java
public class Term {
    private int exponent;
    private int coefficient;
    private Term next;
    public Term(int exp, int coef){
        exponent = exp;
        coefficient = coef;
        next = null;
    }
    public int getCoefficient() { return coefficient;}
    public Term getNext() { return next;}
```
public void setExponent(int exp) { exponent = exp; }
public void setCoefficient(int coeff) { coefficient = coeff; }
public void setNext(Term newTerm) { next = newTerm; }
}

// end of Term class

Note that the implementation of the access procedures getMaxExponent() and addTerm(int exp, int coeff) varies depending on how terms are assumed to be stored in the Linked List. The question has left you free to decide. Terms can be stored in a decreasing ordering with respect to their non-null exponents, or arbitrarily, with no particular ordering. The implementation below uses the first assumption: decreasing ordering on the exponents. The implementation for the case when terms have no ordering within the Linked list is given in Appendix at the end of this file.

ASSUMPTION: Terms are stored in the Polynomial ADT structure in a decreasing ordering with respect to their exponent. The term with highest exponent will be the first element in the linked list and so on. This assumption is preserved by the operation “addTerm(int exp, int coeff)” and the higher-level procedure “addPolynomials(Polynomial p1, Polynomial p2)”.

public class Polynomial implements PolynomialList {
    private ListReferenceBased ListTerms;
    public Polynomial() { ListTerms = new ListReferenceBased(); }
    public boolean isEmpty() { return ListTerms.isEmpty(); }
    public int PolynomialSize() { return ListTerms.size(); }
    public Term getTerm(int pos) { return (Term) ListTerms.get(pos); }
    public int getTermExponent(int pos) {
        return ((Term) ListTerms.get(pos)).getExponent();
    }
    public int getTermCoefficient(int pos) {
        return ((Term) ListTerms.get(pos)).getCoefficient();
    }
    public int getMaxExponent() {
        return ((Term) ListTerms.get(1)).getExponent();
    }
    public void addTerm(int exp, int coeff) {
        Term newTerm = new Term(exp, coeff);
        if (ListTerms.size() == 0) { ListTerms.add(1, newTerm); }
        else {
            int pos = 1;
            while (pos <= ListTerms.size() &&
                (((Term) ListTerms.get(pos)).getExponent() > exp)) {
                pos++;
            }
            if (pos > ListTerms.size()) {
                ListTerms.add(ListTerms.size() + 1, newTerm);
            } else {
                if (((Term) ListTerms.get(pos)).getExponent() == exp) {
                    int coefficient = ((Term) ListTerms.get(pos)).getCoefficient();
                    int newcoefficient = coeff + coefficient;
                    if (newcoefficient != 0) {
                        Term temp = new Term(exp, coeff + coefficient);
                        ListTerms.remove(pos);
                        ListTerms.add(pos, temp);
                    } else { ListTerms.remove(pos); }
                } else { ListTerms.add(pos, newTerm); }
            }
        }
    }
}
(d) The implementation of the higher-level access procedure `addPolynomials` is the same independently of the assumption on the ordering of the Terms. If we use the access procedure `addTerm` implemented above, the result of `addPolynomials` will be a new polynomial where terms satisfy the assumption on the ordering. If instead the implementation given in appendix is used for `addTerm` the new polynomial will have terms stored without the decreasing ordering over the exponents.

```java
Polynomial addPolynomials(Polynomial p1, Polynomial p2)
//pre: p1 and p2 are not empty
//post: returns a new polynomial that is the sum of the two given polynomials p1 and p2
for (int pos=1; pos <= p2.PolynomialSize(); pos++) {
    int tempexp = p2.getTermExponent(pos);
    int tempcoef = p2.getTermCoefficient(pos);
    p1.addTerm(tempexp, tempcoef);
}
return p1;
```

2 (a) class PriorityQ{
private ListNode head;
private int numberofPriorities;
//access procedures
}
class ListNode{
private Queue PatientsQ;
private int priorityLevel;
private ListNode next;
//access procedures
}
class Queue{
private QueueNode first;
private QueueNode last;
//access procedures
}
class QueueNode{
private String name;
private QueueNode next;
//access procedures
}

(b) The priority value passed in input is assumed to be a correct value. No need to check. The priorities are also sorted within the linked list. So the following implementation is sufficient. Note that a standard enqueue access procedure for queues would have been sufficient here, instead of the specified `enqueue((String) newname)`; but this is not relevant to answer the question correctly. Similarly for the case of `Delete(String newname, int priority)`.

```java
public void Add(String newname, int priority){
    ListNode temp = head;
```
for (int pos = 1; pos < priority; pos++) {
    temp = temp.getNext();
    Queue patientqueue = temp.getQueue();
    patientqueue.enqueue((String) newname);
}

public ListNode getNext() {  // this is an access procedure of the class ListNode
    return next;
}

public Queue getQueue() {  // this is also an access procedure of the class ListNode
    return PatientsQ;
}

public void Delete(String newname, int priority) {
    ListNode temp = head;
    for (int pos = 1; pos < priority; pos++) {
        temp = temp.getNext();
        Queue patientqueue = temp.getQueue();
        patientqueue.dequeue((String) newname);
    }
}

(c) the difference between this answer and that given in part (a) is that since in this case we can assume the existence of an ADT Queue, we don’t need to declare the class Queue, and in the class ListNode the type Queue is such an existing implementation of queues.

class PriorityQ {
    private ListNode head;
    private int numberPriorities;
    //access procedures
}
class ListNode {
    private Queue PatientsQ;
    private int pLevel;
    private ListNode next;
    //access procedures
}
class QueueNode {
    private String name;
    private QueueNode next;
    //access procedures
}
APPENDIX: Alternative implementation for the ADT Polynomial

(c2) Assuming instead that terms are stored in any arbitrary order, the class Polynomial is implemented as above except for the two methods getMaxExponent() and addTerm(int exp, int coeff). These are implemented as follows:

```java
public int getMaxExponent(){
    //post: returns -1 if the Polynomial is empty.
    //      Otherwise returns the highest exponent in the Polynomial
    int maxExp = -1;
    int pos = 1;
    while (pos <= ListTerms.size()){
        if (((Term) ListTerms.get(pos)).getExponent() > maxExp) {
            maxExp = ((Term) ListTerms.get(pos)).getExponent();
            pos++;
        }
    }
    return maxExp;
}

public void addTerm(int exp, int coeff){
    Term newTerm = new Term(exp,coeff);
    if (ListTerms.size() == 0) { ListTerms.add(1,newTerm); }
    else{
        int position = findTerm(exp);
        if (position != 0){
            int coefficient = ((Term) ListTerms.get(position)).getCoefficient();
            int newcoefficient = coeff+coefficient;
            if (newcoefficient != 0){
                Term temp = new Term(exp, coeff+coefficient);
                ListTerms.remove(position);
                ListTerms.add(position, temp);
            }else {ListTerms.remove(position);}
        }else { ListTerms.add(1,newTerm); }
    }
}
```

where int findTerm(int exp) is an auxiliary procedure implemented as follows:

```java
private int findTerm(int exp){
    //returns 0 if Term is not in the polynomial;
    //otherwise returns the position of the term in the linked list;
    int pos = 1;
    while (pos <= ListTerms.size() &&
        ((Term) ListTerms.get(pos)).getExponent() != exp){pos++;}
    if (pos <= ListTerms.size()) { return pos; }
    else { return 0; }
}
```