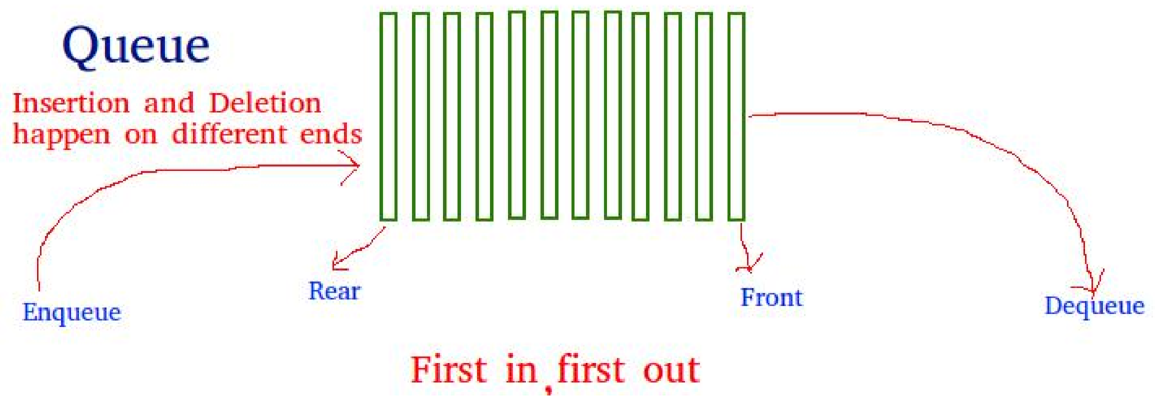
LINEAR DATA STRUCTURES AND ALGORITHMS. Part A: DATA STRUCTURES - ADT MyQueue

BACKGROUND.

A Queue is a linear structure which follows a particular order in which the operations are performed. It operates in a First In First Out (FIFO) fashion. A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. Elements are added to the rear of the queue and removed from the front.



Problem Specification:

In this assignment you will implement the ADT MyQueue with the following operations:

- **createEmpty():** It creates a new MyQueue with no elements and initialises each of the attributes.
- **isEmpty():** It returns whether the queue is empty or not.
- **front():** Returns the item at the top of the queue.
- **rear():** Returns the item at the back of the queue.
- **enqueue(int element):** It places an integer item into the rear of the queue (if there is space for it). If there is no space, it prints on the screen an error message informing that the queue is full.
- **dequeue():** It removes the integer that is in the front of the queue (if the queue is non-empty) and returns that item. If the queue is empty, it prints on the screen an error message and returns -1.
- **print():** It prints the items that are in the queue in a single line (the item in the front of the queue should appear in the left-most position). If the queue is empty, it prints an error message to the screen confirming that the stack is empty.

A.1 – STATIC IMPLEMENTATION (15%)

BACKGROUND.

The unit on Canvas contains the following files:

- **MyMain.java:** This class tests the functionality of the queue's static and dynamic implementations.
- **MyQueue.java:** This interface specifies the ADT MyQueue containing int elements.
- **MyStaticQueue.java:** This class implements all operations of MyQueue, using a static based implementation based on the following attributes:
    - private int items[];
    - private int numItems;
    - private int maxItems;

EXERCISE.

Implement the class MyStaticQueue.java. IMPORTANT: only modify this .java file. Look for the comments: //TO-COMPLETE

A.2 – DYNAMIC IMPLEMENTATION (15%)

Here there is no limit on the size of the queue.

BACKGROUND.

The unit on Canvas contains the following additional files:

- **MyNode.java:** This class models the concept of a single linked node containing an int *info* and a pointer to its next node *next*.
- **MyDynamicQueue.java:** This class implements all operations of MyQueue, using a dynamic based implementation based on the following attributes:
    - private MyNode head;

EXERCISE.

Implement the class MyDynamicQueue.java. IMPORTANT: only modify this .java file. Look for the comments: //TO-COMPLETE

A.3 – Priority Queuing (20%)

A dynamic implementation that prioritises items based on their integer value.

BACKGROUND.

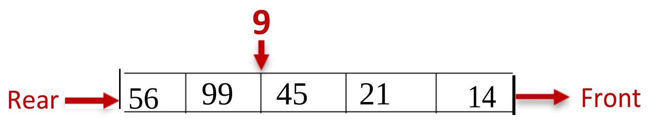
In this last section, we extend the ADT MyQueue implementations to include the capability of adding and removing elements based on a priority class. The priority classes are categorised as follows:

Class 1: Highest priority elements with integer values from 0-49

Class 2: Lowest priority elements with integer values 50 or greater

Highest priority elements will always reach the top of the queue ahead of low priority elements. Elements in the same priority class operate in the normal FIFO fashion.

Example: In the example shown below, we want to enqueue the number 9. The element is in the highest priority class so will get precedence over any low priority elements already in the queue. The new element is placed at the rear of the high priority elements.



- **MyPriorityQueue.java:** This class implements all operations of MyQueue, using a dynamic based implementation based on the following attributes:
    - private MyNode head;

EXERCISE.

Implement the class MyPriorityQueue.java. IMPORTANT: Look for the comments: //TO-COMPLETE

Part B: Divide and Conquer (20%)

Use the divide and conquer approach (recursion) to implement the functions outlined below.

BACKGROUND.

The unit on Canvas contains the following files:

- **MyMain.java:** This class tests the functionality of the divide&conquer implementation.
- **MyList.java, MyStaticList.java, MyNode.java, MyDynamicList.java:** These classes define for the package MyList<T> which we have studied previously in the lectures.
- **DivideAndConquerAlgorithms.java:** This class contains the proposed divide&Conquer functions you have to implement.

EXERCISE.

1. public MyList<Integer> getEven(MyList<Integer> m);
   Given a MyList of Integers m, uses recursion to return a new list containing of all the even numbered elements m.

2. public int getProduct(MyList<Integer> m);
   This function returns the product of all elements in m (-1 if m is empty).

3. public boolean isEqual(MyList<Integer> m1, MyList<Integer> m2);
   Given two MyList of Integers, uses recursion to determine if both lists are identical (contain the same elements in the same order).

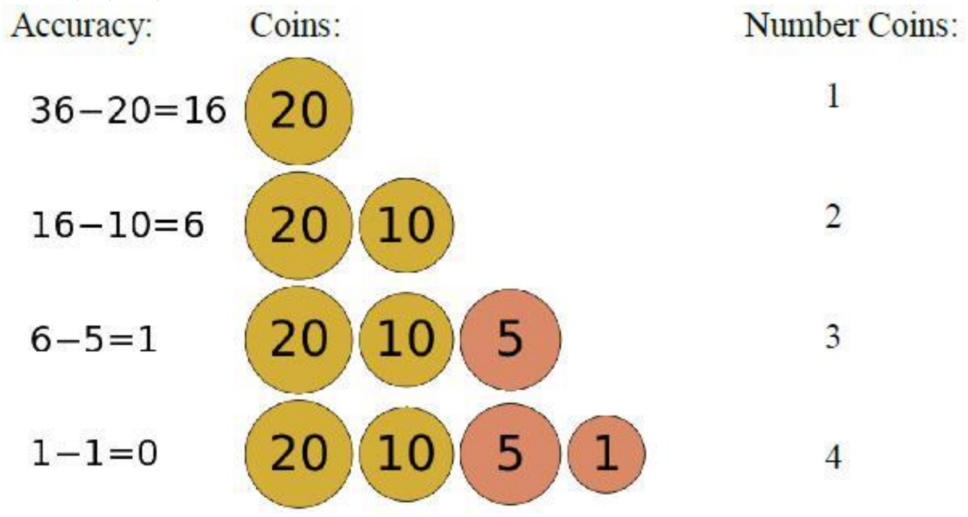
4. public int fermat(int n);
   Mathematically, a Fermat number is defined as:
   $= 2^2 + 1 \text{ for } \qquad \geq 0$

   Thus, the Fermat series is as follows:
   3, 5, 17, 257, 65537……..

Part C: Greedy Algorithms (30%)

BACKGROUND.

The **change-making problem** addresses the question of finding the minimum number of coins that add up to a given amount of money. It is a knapsack type problem and has applications wider than just currency.
Greedy algorithms determine minimum number of coins to give while making change. Note that there is no limit on how many coins can be returned from each type of coin. These are the steps a human would take to emulate a greedy algorithm to represent 36 cents using only coins with values {1, 5, 10, 20}:



The unit on Canvas contains the following files:

- **MyMain.java:** This class tests the functionality of the greedy changemaking implementation.
- **MyList.java, MyStaticList.java, MyNode.java, MyDynamicList.java:** These classes define for the package MyList<T> which we have seen previously in the lectures of the Block II: Data Structures.
- **ChangeMaking.java:** This class contains the proposed Greedy algorithm that you have to implement. The algorithm solves the problem of change making described above.

EXERCISE.
Implement the functions of the class ChangeMakingJava.java according to the schema of greedy algorithms studied in the lectures.

1. selectionFunctionFirstCandidate:
   Choose the first available coin from all candidates.

2. selectionFunctionBestCandidate:
   Select the candidate to ensure the minimal amount of coins are used.

3. feasibilityTest
   Given a current solution and a selected candidate, this function states whether the candidate must be added to the solution or discarded
4. solutionTest
   Given a current solution, this function states whether it is a final solution or it can still be improved
5. objectiveFunction
   This function computes how many coins are used in the solution
6. solve: This function calls to the above functions in order to solve the problem. The selection function executed is selectionFunctionFirstCandidate if *typeSelectFunc=1*. Instead, if *typeSelectFunc=2*, selectionFunctionBestCandidate is the selected function. This function is responsible for printing on the screen the output of the problem (see Figure above as example of output) and must include the:

   Accuracy, which is the difference of amount of change provided minus the target amount of money (input of the function).

   Coins that are provided as change.

   Number of coins provided as change.

SUBMISSION DETAILS.

Submission Details.

Please submit the following:

- Part A.1 File “MyStaticQueue.java”.
- Part A.2 File “MyDynamicQueue.java”.
- Part A.3 File “MyPriorityQueue.java”.
- Part B File “DivideAndConquerAlgorithms.java”
- Part C File “ChangeMaking.java”