Estructuras de Datos y Algoritmos Prueba de Evaluación Continua Número 1

Ejercicio 1

La realización de dicho ejercicio, se hizo en base a las implementaciones correspondientes al libro Algorithms Fourth Edition, las cuales se adjuntan en el siguiente link: https://algs4.cs.princeton.edu/10fundamentals/

A) Dada la estructura Queue q = new Queue<Integer>():

q.dequeue() = El método dequeue, comprueba si la cola está o no vacía, antes de efectuar la operación de desencolado. En este caso, la cola está vacía, entonces la respuesta que obtendremos será una excepción del tipo NoSuchElementException. De esta manera, los punteros, first y last, se mantienen con valores nulos.

q.dequeue() = Como la cola sigue estando vacía, vuelve a suceder lo mismo explicado en el punto anterior.

q.size() = Dado que, la cola sigue estando vacía, el resultado devuelto por dicha operación es 0. Los punteros first y last, siguen siendo nulos.

q.enqueue(17) = En este caso, se agrega el objeto de tipo entero con valor 17 a la cola q. De esta forma, y como es el único elemento en la cola hasta el momento, los punteros first and last, apuntan los dos a la misma instancia del objeto Nodo que internamente posee el objeto Integer con el valor 17. El método es void, de manera que no nos devuelve nada.

q.enqueue(18) = En este caso, se agrega el objeto de tipo entero con valor 18 a la cola q. De esta forma, el puntero last queda apuntando a un Nodo con el valor 18, mientras que, el puntero first, se mantiene apuntando al objeto Nodo que almacena el valor 17. El método es void, de manera que no nos devuelve nada.

q.size() = Al invocar dicho método, la respuesta obtenida será el número 2. Los punteros first y last, se mantienen igual que en el punto anterior, ya que no se han quitado ni agregado elementos.

q.dequeue() = Como resultado de esta invocación, la cola nos devolverá un Integer, cuyo valor es 17. De esta forma, el valor 17 ha sido removido de la estructura, dejando nuevamente al puntero first y last apuntando al mismo nodo, que en este caso, es el que contiene el valor 18

q.dequeue() = Como resultado de esta invocación, la cola nos devolverá un Integer, cuyo valor es 18. De esta forma, el valor 18 ha sido removido de la estructura, dejando nuevamente al puntero first y last en null, ya que ahora, la cola ha quedado vacía.

q.dequeue() = Como resultado de esta invocación, la cola nos devolverá nuevamente una excepción del tipo NoSuchElementException, ya que la misma se encuentra vacía. Los punteros first y last, están ambos en null.

q.enqueue(21) = En este caso, se agrega el objeto de tipo entero con valor 21 a la cola q. Los punteros first and last, apuntan los dos a la misma instancia del objeto Nodo que internamente posee el objeto Integer con el valor 21. El método es void, de manera que no nos devuelve nada.

B) Dada la estructura Stack s = new Stack<Integer>():

s.push(18) = Al invocar dicho método, se agrega un Nodo cuyo valor es 18 a la pila s. De esta manera, el puntero first apunta ahora al Nodo que contiene el Integer con el valor 18. El método push es tipo void, con lo cual no se devuelve ningún valor.

s.pop() = La respuesta obtenida al ejecutarse dicho método es el objeto Integer con valor 18. El puntero first ahora tiene valor nulo, dado que, la pila ha quedado vacía.

s.pop() = Como la pila está vacía, al momento de ejecutarse dicho método, la respuesta obtenida va a ser una excepción del tipo NoSuchElementException. El puntero first, sigue teniendo valor null.

s.push(2) = Al invocar dicho método, se agrega un Nodo cuyo valor es 2 a la pila s. De esta manera, el puntero first apunta ahora al Nodo que contiene el Integer con el valor 2. El método push es tipo void, con lo cual no se devuelve ningún valor.

s.pop() = La respuesta obtenida al ejecutarse dicho método es el objeto Integer con valor 2. El puntero first ahora apunta a null, ya que la pila ha quedado nuevamente vacía.

s.push(3) = Al invocar dicho método, se agrega un Nodo cuyo valor es 3 a la pila s. De esta manera, el puntero first apunta ahora al Nodo que contiene el Integer con el valor 3. El método push es tipo void, con lo cual no se devuelve ningún valor.

s.push(4) = Al invocar dicho método, se agrega a la pila un Nodo cuyo valor es 4. De esta forma, el puntero first ahora hace referencia al Nodo que contiene el Integer con el valor 4

s.pop() = La respuesta obtenida al ejecutarse dicho método es el objeto Integer con valor 4. El puntero first ahora apunta al nodo que contiene el Integer con valor 3.

s.pop() = La respuesta obtenida al ejecutarse dicho método es el objeto Integer con valor 3. El puntero first ahora apunta a null, ya que la pila ha quedado nuevamente vacía.

C) Dada la estructura Bag b = new Bag<Integer> ();

b.add(-88) = Al ejecutar dicho método, se agrega a la estructura bag un nodo con el Integer -88 en su interior. De esta manera, el puntero first apunta ahora al Nodo que contiene el Integer con valor -88. El método add, es tipo void, con lo cual, no devuelve ningún valor.

b.add(3) = Como resultado de esta ejecución, se agrega a la estructura bag un nuevo nodo con el valor 3 de tipo Integer. De esta manera, el puntero first apunta ahora al nuevo nodo recientemente agregado. Como el método add es void, no se devuelve ningún valor.

b.add(1) = Al ejecutar dicho método, se agrega a la estructura bag un nodo que contiene al valor Integer 1. Por lo tanto, el puntero first apunta ahora a este último. El método no devuelve nada, dado que el mismo es tipo void.

b.add(16) = Como resultado de esta ejecución, se agrega a la estructura bag un nodo que contiene el valor 16 internamente. El puntero first, ahora apunta a dicho nodo. Nuevamente, no obtenemos ningún valor de retorno, ya que, el método add es void.

b.add(9) = Al ejecutar dicho método, se agrega a la estructura bag un nodo con el Integer 9 en su interior. De forma que, el puntero first apunta ahora al nuevo nodo agregado. No obtenemos ningún valor de retorno, dado que, el método add es void.

Ejercicio 2

Para la realización de dicho ejercicio, se asume que la estructura Queue solo soporta Integers, de manera que la solución propuesta es la siguiente :

```
public void enqueueEven(Integer[] numbers){
      if(numbers == null){
         throw new IllegalArgumentException("The input array can't be null");
      }else if (numbers.length == 0){
         throw new IllegalArgumentException("The input array is empty");
      }
      for(int i = 0; i < numbers.length; i++){
         Integer current = numbers[i];
         if(current % 2 == 0) {
            this.enqueue(current);
         }
      }
 }

   public Integer[] dequeue(Integer n){
      Integer queueSize;
      Integer diffBetweenNAndSize;
      Integer[] elements;
      if(n == null || n <= 0){
         throw new IllegalArgumentException("The input number must be a positive integer");
      } else if (this.isEmpty()) {
         throw new NoSuchElementException("The queue is empty");
      }
      queueSize = this.size();
      diffBetweenNAndSize = queueSize - n;
      if(diffBetweenNAndSize <0 ) {
         elements = new Integer[queueSize];
      }else {
         elements = new Integer[n];
      }
      for(int i = 0; i < diffBetweenNAndSize; i++) {
         Integer element = this.dequeue();
         this.enqueue(element);
      }
      int index = 0;
      while(!this.isEmpty() && index < n) {
         elements[index] = this.dequeue();
         index++;
      }
      return elements;
   }
```

Ejercicio 3

A) Para almacenar el conjunto de objetos correspondientes a una sede determinada, es posible utilizar una Lista Enlazada o una Cola.
La Lista Enlazada es una buena opción, debido a que dicha estructura respeta el orden de inserción de los objetos. De esta manera, al iterar a través de los nodos de la misma, podemos asegurar que el orden original se mantiene.
Por otro lado, sería posible utilizar una Cola, ya que, la misma es una estructura tipo FIFO, motivo por el cual, los objetos serán almacenados en el orden en el que fueron insertados.
No obstante, cabe aclarar, que si bien el uso de la Cola cumple con el requerimiento pedido, debe tenerse en cuenta que si la iteración de la misma no se hace a través de su Iterador, cada vez que se llame al método dequeue(), se estará removiendo de la estructura el elemento en cuestión. Por este motivo, considero que utilizar una Lista es una mejor opción, ya que todos los métodos que la misma expone para obtener uno o N elementos, no alteran el estado interior de la estructura.

C) La estructura de datos que usaría es una Lista, puesto que la misma me permite almacenar todos los elementos en el orden que fueron insertados y al consultar por ellos, no se ve afectado el estado interno de la estructura de datos.
Con respecto a las interfaces correspondientes a la librería de Collections de Java, la interfaz adecuada para representar una lista es List. Si bien hay varias implementaciones de dicha interfaz, en este caso, se decidió utilizar ArrayList. Particularmente en este ejercicio, si pensamos en complejidad temporal, quizás no exista una diferencia tan marcada entre LinkedList y ArrayList, puesto que al momento de hacer la inserción, se podría decir que LinkedList es un poco mas optimo, puesto que la complejidad de la operación add(E element), siempre es constante, mientras que, en ArrayList es una complejidad constante amortizada, debido al trabajo de duplicación del array, que dicha implementación debe realizar cuando alcanza el máximo establecido internamente. Por otro lado, cuando se busca un elemento a través de su índice, ArrayList es más eficiente que LinkedList, dado que el acceso lo hace directamente, sin necesidad de iterar por todos los elementos previos al que se está buscando. Sin embargo, en el ejercicio presentado, la búsqueda se hace por el campo Id de la clase Site, motivo por el cual, más allá de la implementación de List que usemos, la búsqueda resultara con complejidad O(N).
No obstante, sí podemos establecer una diferencia clara en la complejidad espacial entre ArrayList y LinkedList. Los nodos de ArrayList ocupan menos espacio, por el hecho que no necesitan mantener una referencia al siguiente y al anterior, como si

necesita hacer LinkedList. De forma tal, que en lo que respecta a complejidad espacial, es más eficiente utilizar ArrayList.