Ejercicio U7-B4_1:

```
import java.util.ArrayList;

public class App{
    public static void main(String[] args){
        ArrayList<Integer> ll=new ArrayList<>(); // Declaración y creación del ArrayList de enteros.
        ll.add(1); // Añade un elemento al final de la lista.
        ll.add(3); // Añade otro elemento al final de la lista.
        System.out.println("Despues de añadir 1 y 3:" +ll);
        ll.add(1,2); // Añade en la posición 1 el elemento 2.
        System.out.println("Despues de añadir en la posición 1 el elemento 2:" +ll);
        ll.add(ll.get(1)+ll.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.
        System.out.println("Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.:" +ll);
        ll.remove(0); // Elimina el primer elementos de la lista.
        System.out.println("Eliminado el primer elementos de la lista anterior:" +ll);
    }
}
```

- ¿LinkedList mejor o peor que ArrayList?

Observando el código anterior, lo único que hicimos fue edit->replace de LinkedList por ArrayList y todo funciona exactamente igual. Es lógico que todo siga funcionando igual ya que ambas clases hacen lo mismo: implantar el concepto de Lista y el concepto de lista se implanta técnicamente implementando el interface List. En ejemplos como el anterior no es relevante usar una clase u otra, ya que no hay un contexto de uso del que deducir cual es mejor debido al almacenamiento subyacente de cada clase.

- A la vista de lo fácil que es trabajar con ArrayList ¿Crees que merece la pena usar arrays tradicionales?. SÍ, MUCHAS VECES. Un par de razones (y hay más):
    - Los arrays tradicionales pueden almacenar tipos primitivos y cuando la eficiencia es importante, esto puede ser fundamental.
    - En el API java multitud de clases con métodos que trabajan en sus parámetros con Arrays. Si no entendemos los arrays no podemos disfrutar estos métodos

  Por tanto hay que tener claro que **¡es imposible librarse de los arrays!.**

Ejercicio U7-B4_2:

```
import java.util.ArrayList;
import java.util.List;

public class App{
    public static void main(String[] args){
        List<Integer> lista=new ArrayList<Integer>();
        lista.add(1); // Añade un elemento al final de la lista.
        lista.add(3); // Añade otro elemento al final de la lista.
        System.out.println("Despues de añadir 1 y 3:" +lista);
        lista.add(1,2); // Añade en la posición 1 el elemento 2.
        System.out.println("Despues de añadir en la posición 1 el elemento 2:" +lista);
        lista.add(lista.get(1)+lista.get(2)); // Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.
        System.out.println("Suma los valores contenidos en la posición 1 y 2, y lo agrega al final.:" +lista);
        lista.remove(0); // Elimina el primer elementos de la lista.
        System.out.println("Eliminado el primer elementos de la lista anterior:" +lista);
    }
}
```

Este ejemplo es tan pequeño que no es relevante, pero como siempre en caso duda cuanto más genérico sea el código mejor. Si mi código sólo requiere las operaciones generales de listas que están especificadas en el interface List, es mejor utilizar el interface List ya que si veo que me equivoqué en la elección del almacenamiento simplemente tengo que cambiar una instrucción:

```
 List<Integer> t=new ArrayList<Integer>();
```
Por
```
List<Integer> t=new LinkedList<Integer>();
```
Y el programa sigue funcionando exactamente igual. ¡Compruébalo!

Si precisara utilizar algún método específico(no especificado en la interface List) de LinkedList o ArrayList entonces sí que me vería obligado a utilizar una clase concreta.

Ejercicio U7-B4_3:

```
import java.util.ArrayList;
import java.util.Arrays;

public class App{

   static boolean hayRaya(ArrayList<ArrayList<Character>> tabla, int fila, int columna, char jugador) {
       //se supone  que  fila y columna se corresponde a una celda libre

       boolean hay = false;
       tabla.get(fila).set(columna, jugador);
       //usamos equals para comparar 2 objetos Character
       //miramos si hay raya en fila
       if (tabla.get(fila).get(0).equals(tabla.get(fila).get(1)) && tabla.get(fila).get(1).equals(tabla.get(fila).get(2)) && tabla.get(fila).get(2) == jugador) {
           hay = true;
       } //miramos si raya en  columna
          else if (tabla.get(0).get(columna).equals(tabla.get(1).get(columna)) && tabla.get(1).get(columna).equals(tabla.get(2).get(columna)) && tabla.get(2).get(columna)
== jugador) {
           hay = true;
       } //miramos si hay raya en diagonal  2 casos.
       else if (tabla.get(0).get(0).equals(tabla.get(1).get(1)) && tabla.get(1).get(1).equals(tabla.get(2).get(2)) && tabla.get(2).get(2) == jugador) {
           hay = true;
       } else if (tabla.get(0).get(2).equals(tabla.get(1).get(1)) && tabla.get(1).get(1).equals(tabla.get(2).get(0)) && tabla.get(2).get(0) == jugador) {
           hay = true;
       } else {
           hay = false;
       }
       //volvemos a dejar como libre la celda, este método investiga pero no cambia nada
       tabla.get(fila).set(columna, '_');

       return hay;
   }

   public static void main(String args[]) {
       ArrayList<ArrayList<Character>> tabla= new ArrayList<>();
       //incializamos con asList de fila en fila
       tabla.add(0, new ArrayList<>(Arrays.asList('X','_', 'X')));
       tabla.add(1, new ArrayList<>(Arrays.asList('O','_', 'O')));
       tabla.add(2, new ArrayList<>(Arrays.asList('X','_', '_')));


      /* si iniciamos de celda en celda salen muchas líneas
       ArrayList<ArrayList<Character>> tabla = new ArrayList<>();
       tabla.add(new ArrayList<Character>());
       tabla.add(new ArrayList<Character>());
       tabla.add(new ArrayList<Character>());
       //para crear elementos y que el array crezca necesito add
       tabla.get(0).add('X');
       tabla.get(0).add('_');
       tabla.get(0).add('X');

       tabla.get(1).add('O');
       tabla.get(1).add('_');
       tabla.get(1).add('O');

       tabla.get(2).add('X');
       tabla.get(2).add('_');
       tabla.get(2).add('_');*/

       System.out.println(hayRaya(tabla, 0, 1, 'X') ? "SI" : "NO");
       System.out.println(hayRaya(tabla, 0, 1, 'O') ? "SI" : "NO");

   }
}
```

Parece que ... con arraylist es mucho más ilegible

Ejercicio U7-B4_4: Teclado estropeado id 144 (acepta el reto)

Ejercicio U7-B4_5:

```
import java.util.LinkedList;
class Pila<T>{
    LinkedList<T> listaDatos= new LinkedList<>();
    void push(T t){
        listaDatos.addFirst(t);
    }
    T pop(){
        return listaDatos.removeFirst();
    }
    boolean esVacia(){
        return listaDatos.isEmpty();
    }
}
public class App {

    public static void main(String args[]) {
        Pila<Character> miPila= new Pila<>();
        miPila.push('a');miPila.push('b');miPila.push('c');
        while(!miPila.esVacia()){
            System.out.print(miPila.pop()+" ");
        }
    }
}
```

Ejercicio U7-B4_6:

```
import java.util.Stack;

public class App {
    public static void main(String args[]) {
        Stack<Character>  miPila= new Stack<>();
        miPila.push('a');miPila.push('b');miPila.push('c');
        while(!miPila.empty()){
            System.out.print(miPila.pop()+" ");
        }
    }
}
```

Ejercicio U7-B4_7: Acepta el reto 141.

Ejercicio U7-B4_8:

Dependiendo de cómo quiera "ver" el principio y final de la cola puedo trabajar con addLast()/removeFirst() o bien con addFirst()/removeLast(). El efecto es el mismo. Lo importante es que se añade por un extremo y se elimina por el contrario que es lo que hace que la lista sea FIFO.

```
import java.util.LinkedList;
class App{
    public static void main(String[] args) {
        LinkedList<Integer> cola1 = new LinkedList<>();
        cola1.addLast(1);
        cola1.addLast(2);
        cola1.addLast(3);
        while (!cola1.isEmpty()) {
            System.out.print(cola1.removeFirst()+ " ");
        }
     }
}
```

```
import java.util.LinkedList;
class App{
    public static void main(String[] args) {
        LinkedList<Integer> mc1 = new LinkedList<>();
        mc1.addFirst(1);
        mc1.addFirst(2);
        mc1.addFirst(3);
        while (!mc1.isEmpty()) {
            System.out.print(mc1.removeLast()+ " ");
        }
        System.out.println();

    }
}
```

Observar en el api de linkedlist que:

- remove() y removeFirst() son equivalentes(hacen lo mismo)
- add() y addLast() también son equivalentes

Por lo tanto, si quiero pensar más facilmente en una cola lo mejor es evitar tanto lío de first y last y usar simplemente add() y remove() y sabemos que con estos cumplimos FIFO.

Ejercicio U7-B4_9:

Al usar referencias de tipo Queue, aunque el objeto sea de tipo LinkedList no puedo usar todos los métodos de LinkedList sólo los especificados en Queue. En el ejemplo offer es similar a addLast() y poll() es equivalente a removeFirst()

```
import java.util.LinkedList;
import java.util.Queue;

class App {

    public static void main(String[] args) {
        Queue<Integer> cola1 = new LinkedList<>();
        cola1.offer(1);
        cola1.offer(2);
        cola1.offer(3);
        while (!cola1.isEmpty()) {
            System.out.print(cola1.poll() + " ");
        }
    }
}
```

O equivalente

```
import java.util.LinkedList;
import java.util.Queue;

class App {

    public static void main(String[] args) {
        Queue<Integer> cola1 = new LinkedList<>();
        cola1.add(1);
        cola1.add(2);
        cola1.add(3);
        while (!cola1.isEmpty()) {
            System.out.print(cola1.remove() + " ");
        }
    }
}
```

Ejercicio U7-B4_10:

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Random;

class NodoArbol {

    NodoArbol nodoIzq;
    int datos;
    NodoArbol nodoDer;

    public NodoArbol(int datosNodo) {
        datos = datosNodo;
        nodoIzq = nodoDer = null; //recien creado un nodo, no tiene hijos
    }

}

class Arbol {

    private NodoArbol raiz;

    public Arbol() {
        raiz = null;
    }

    // si el valor ya existe en el arbol, no inserta nada
    public void insertar(int valorInsertar) {
        if (raiz == null) {
            raiz = new NodoArbol(valorInsertar);
```

```
        } else {
            ayudanteInsertarNodo(raiz, valorInsertar);
        }
    }

    private void ayudanteInsertarNodo(NodoArbol a, int valorInsertar) {
        // inserta en el subárbol izquierdo
        if (valorInsertar < a.datos) {
            // inserta nuevo NodoArbol
            if (a.nodoIzq == null) {
                a.nodoIzq = new NodoArbol(valorInsertar);
            } else {
                ayudanteInsertarNodo(a.nodoIzq, valorInsertar);
            }
        } else if (valorInsertar > a.datos) { // inserta en el subárbol derecho
            if (a.nodoDer == null) {
                a.nodoDer = new NodoArbol(valorInsertar);
            } else {
                ayudanteInsertarNodo(a.nodoDer, valorInsertar);
            }
        }
    }

    public void recorridoPreordenConTAB() {
        ayudantePreordenConTAB(raiz, "");
    }

    private void ayudantePreordenConTAB(NodoArbol nodo, String tab) {
        if (nodo == null) {
            System.out.println(tab + "null");
            return;
        }

        System.out.println(tab + nodo.datos);
        tab = tab + "\t";
        ayudantePreordenConTAB(nodo.nodoIzq, tab);
        ayudantePreordenConTAB(nodo.nodoDer, tab);
    }

    void recorridoAnchura() {
        Queue<NodoArbol> cola = new LinkedList<>();
        if (raiz == null) {
            return;
        }
        cola.add(raiz);
        while (!cola.isEmpty()) {
            NodoArbol n = cola.remove();
            System.out.print(n.datos + " ");
            if (n.nodoIzq != null) {
                cola.add(n.nodoIzq);
            }
            if (n.nodoDer != null) {
                cola.add(n.nodoDer);
            }

        }
    }
}

public class App {

    public static void main(String args[]) {
        Arbol arbol = new Arbol();
        int valor;
        Random numeroAleatorio = new Random();

        System.out.println("Insertando los siguientes valores: ");
        // inserta 10 enteros aleatorios de 0 a 99 en arbol
        //puede ser menos de 10 si se generan duplicados
        for (int i = 1; i <= 10; i++) {
            valor = numeroAleatorio.nextInt(100);
            System.out.print(valor + " ");
            arbol.insertar(valor);
```

```
        }
        System.out.println("\nRecorrido preorden con indentaciones.....");
        arbol.recorridoPreordenConTAB();

        System.out.println("\nrecorrido en anchura");
        arbol.recorridoAnchura();
    }
}
```

Ejercicio U7-B4_11: Acepta el reto, al mundial con transatlántico 473

Obligatorio con interface Queue