UNIVERSIDAD AUTÓNOMA GABRIEL RENE MORENO
FACULTAD DE INGENIERÍA Y CIENCIAS DE LA COMPUTACIÓN Y TELECOMUNICACIONES

ESTRUCTURAS DE DATOS 2

CONTENIDO: *// Breve Descripción del Contenido*

TAREA-6. INSERTAR, ELIMINAR ELEMENTOS EN LISTAS.

PORCENTAJE TERMINADO : 85%.

GRUPO: 14

Nombre	Registro
Cristian Gabriel Gedalge Cayhuara	219022062

Fecha de Presentación: Jueves ,11 de abril de 2024

COMENTARIO:
Me parecieron desafiantes e interesante los ejercicios propuestos en esta tarea pude entender más sobre la importancia de los punteros que fueron fundamentales para resolver los ejercicios, sé que mi código no es el mejor en tema de rapidez ya que lo hice a lo apurado,otros ejercicios me parecieron complicados por lo cual no acabe.

LISTA ARREGLO

```
public class Lista {

    private int max = 100;
    private int v[];
    private int cantElem;

    public Lista() {

        this.v = new int[max];
        this.cantElem = 0;
    }

//1.   L1.toString() : Método que devuelve una cadena, que representa la secuencia de
elementos de la lista L1.
    @Override
    public String toString() {
        String s = "[";
        int j = 0;
        while (this.cantElem > j) {
            s = s + this.v[j];
            if (j < this.cantElem) {
                s += ",";
            }
            j++;
        }
        return (s);
    }
//2.   L1.insertarPrim(x) : Método que inserta el elemento x, al final de la lista L1.

    public void insertarPrim(int x) {
        int cant = this.cantElem - 1;
        for (int i = cant; i > 0; i--) {
            this.v[i + 1] = this.v[i];
        }
        this.v[0] = x;
        this.cantElem++;

    }
//3.   L1.insertarUlt(x) : Método que inserta el elemento x, al final de la lista L1.

    public void insertarUlt(int x) {
        insertarIesimo(x, this.cantElem);
    }
```

//4. L1.insertarIesimo(x, i) : Método que inserta el elemento x, en la posición i, de la lista L1.

```
public void insertarIesimo(int x, int i) {
    int cant = this.cantElem - 1;
    for (int j = cant; j >= i; j--) {
        this.v[j + 1] = this.v[j];
    }
    this.v[i] = x;
    this.cantElem++;
}
```

//5. L1.insertarIesimo(L2, i) : Método que insertar los elementos de la lista L2 en la lista L1, desde la posición i.

```
public void insertarIesimo(Lista L2, int i) {
    for (int j = 0; j < L2.cantElem; j++) {
        insertarIesimo(L2.v[j], i);
        i++;
    }
}
```

//6. L1.insertarPrim(L2) : Método que insertar los elementos de la lista L2 al principio de la lista L1.

```
public void insertarPrim(Lista L2) {
    int i = 0;
    for (int j = 0; j < L2.cantElem; j++) {
        insertarIesimo(L2.v[j], i);
        i++;
    }
}
```

//7. L1.insertarUlt(L2) : Método que insertar los elementos de la lista L2 al final de la lista L1.

```
public void insertarUlt(Lista L2) {
    for (int i = 0; i < L2.cantElem; i++) {
        insertarIesimo(L2.v[i], this.cantElem);
    }
}
```

//8. L1.insertarAsc(x) : Método que inserta el elemento x en su lugar correspondiente, en la lista L1, ordenada en forma ascendente.

```
public void insertarAsc(int x) {

    int i = 0;
    while (i < this.cantElem && this.v[i] < x) {
        i++;
    }
```

```
        insertarIesimo(x, i);
    }
//9. L1.insertarDes(x) : Método que inserta el elemento x en su lugar correspondiente, en la lista L1, ordenada en forma descendente.

    public void insertarDes(int x) {

        int i = 0;
        while (i < this.cantElem && this.v[i] > x) {
            i++;
        }
        insertarIesimo(x, i);
    }
//10. L1.concatenar(L2, L3): Método que concatena las listas L2 con L3 en L1.

    public void concatenar(Lista L2, Lista L3) {
        for (int i = 0; i < L2.cantElem; i++) {
            insertarIesimo(L2.v[i], this.cantElem);
        }
        for (int i = 0; i < 10; i++) {
            insertarIesimo(L3.v[i], this.cantElem);
        }
    }
//11. L1.intercalar(L2, L3): Método que intercala los elementos de las Listas L2 con L3 en L1.

    public void intercalar(Lista L2, Lista L3) {
        int cantTotal = L2.cantElem + L3.cantElem;
        int iL2 = 0, iL3 = 0;
        boolean flag = true;
        while (cantTotal > 0) {

            if (iL2 < L2.cantElem && flag) {
                insertarIesimo(L2.v[iL2], this.cantElem);
                iL2++;
                flag = false;
            } else {
                if (iL3 < L3.cantElem && flag) {
                    insertarIesimo(L3.v[iL3], this.cantElem);
                    iL3++;
                    flag = true;
                } else {
                    flag = true;
                }
            }
            cantTotal--;
        }
```

```
    }
//12. L1.merge(L2, L3): Método que realiza el merge en L1, de las listas ordenadas en
forma ascedente L2 y L3.

    public void merge(Lista L2, Lista L3) {
        int i = 0, j = 0;
        while (i < L2.cantElem && j < L3.cantElem) {
            if (L2.v[i] < L3.v[j]) {
                insertarIesimo(L2.v[i++], this.cantElem);
            } else {
                insertarIesimo(L3.v[j++], this.cantElem);
            }
        }
        while (i < L2.cantElem) {
            insertarIesimo(L2.v[i++], this.cantElem);
        }

        while (j < L3.cantElem) {
            insertarIesimo(L3.v[j++], this.cantElem);
        }
    }
//13.   L1.iguales() : Método Lógico que devuelve True, si todos los elementos de la lista
L1 son iguales.

    public boolean iguales() {
        if (this.cantElem == 0) {
            return false;
        }
        int i = 0;
        int ele = this.v[i];
        while (i < this.cantElem) {
            if (ele != this.v[i]) {
                return false;
            }
            i++;
        }
        return true;
    }
//14.   L1.diferentes() : Método Lógico que devuelve True, si todos los elementos de la lista
L1 son diferentes.

    public boolean diferentes() {
        if (this.cantElem == 0) {
            return false;
        }
        for (int i = 0; i < this.cantElem; i++) {
            int ele=this.v[i];
```

```
            int contador=0;
            for (int j = i; j < this.cantElem; j++) {
               if(ele==this.v[j]) contador++;
            }
            if(contador!=1)
            {
               return false;
            }
         }
         return true;
      }
//15.   L1.mayorElem() : Método que devuelve el mayor elemento de la lista L1.
      public int mayorElem(){
         int i = 0;
         int mayor = this.v[i];
         while (i < this.cantElem) {
            if (!(mayor > this.v[i])) {
               mayor=this.v[i];
            }
            i++;
         }
         return mayor;
}
//16.    L1.menorElem() : Método que devuelve el mayor elemento de la lista L1.
      public int menorElem(){
         int i = 0;
         int menor = this.v[i];
         while (i < this.cantElem) {
            if (!(menor < this.v[i])) {
               menor=this.v[i];
            }
            i++;
         }
         return menor;
      }
//17.    L1.ordenado()  : Método Lógico que devuelve True, si todos los elementos de la
lista L1 están ordenados en forma ascendente o descendente.
      public boolean ordenado()
      {
         return asc() ||desc();
      }
      public boolean asc()
      {
         int i = 0;
         int ele = this.v[i];
         i++;
         while(i < this.cantElem) {
```

```
            if (ele <= this.v[i]) {
                ele=this.v[i];
            }else{
                return false;
            }
            i++;
        }
        return true;
    }

    public boolean desc()
    {
        int i = 0;
        int ele = this.v[i];
        i++;
        while(i < this.cantElem) {
            if (ele >= this.v[i]) {
                ele=this.v[i];
            }else{
                return false;
            }
            i++;
        }
        return true;
    }
//18.   L1.indexOf(x) : Método que devuelve la posición de la primera ocurrencia del
elemento x. Si x no se encuentra en la lista L1, el método devuelve –1.
    public  int indexOf(int x)
    {
        for (int i = 0; i <this.cantElem; i++) {
            if(x==this.v[i])
            {
                return i;
            }
        }
        return -1;
    }
//19.   L1.indexOf(x, i) : Método que devuelve la posición de la primera ocurrencia del
elemento x, la búsqueda se realiza desde la posición i.
    public  int indexOf(int x,int i)
    {
        for (int j = i; j <this.cantElem; j++) {
            if(x==this.v[i])
            {
                return i;
            }
        }
```

```
        return -1;
    }
//20.   L1.lastIndexOf(x) : Método que devuelve la posición de la última ocurrencia del elemento x. Si x no se encuentra en la lista L1, el método devuelve –1.
    public  int lastIndexOf(int x)
    {
        for (int j = this.cantElem-1; j >=0; j--) {
            if(x==this.v[j])
            {
                return j;
            }
        }
        return -1;
    }
//21.   L1.lastIndexOf(x, i) : Método que devuelve la posición de la última ocurrencia del elemento x. Si x no se encuentra en la lista L1, el método devuelve –1. La búsqueda se realiza desde la posición i.
    public  int lastIndexOf(int x,int i)
    {
        for (int j = this.cantElem; j >=i; j--) {
            if(x==this.v[j])
            {
                return j;
            }
        }
        return -1;
    }
//22.   L1.reemplazar(x, y) : Método que reemplaza todas las ocurrencias del elemento x por el elemento y en la lista L1.
    public void reemplazar(int x, int y){
        for (int j =0; j<this.cantElem; j++) {
            if(x==this.v[j])
            {
                this.v[j]=y;
            }
        }
    }
//23.   L1.seEncuentra(x) : Método Lógico que devuelve True, si el elemento x, se encuentra en la lista L1.
    public boolean seEncuentra(int x ){
        for (int j =0; j<this.cantElem; j++) {
            if(x==this.v[j])
            {
                return true;
            }
        }
        return false;
```

```
    }
//24.   L1.frecuencia(x) : Método que devuelve la cantidad de veces que aparece el
elemento x en la lista L1.
    public int frecuencia(int x){
        int contador=0;
        for (int j =0; j<this.cantElem; j++) {
            if(x==this.v[j])
            {
                contador++;
            }
        }
        return contador;
    }
//25.   L1.existeFrec(k) : Método Lógico que devuelve True, si existe algún elemento que
se repite exactamente k-veces en la lista L1.
    public boolean existeFrec(int k){

        for (int j =0; j<this.cantElem; j++) {
            if(frecuencia(this.v[j])==k)
            {
                return true;
            }
        }
        return false;
    }
//26.   L1.mismasFrec() : Método Lógico que devuelve True, si los elementos de la lista L1
tienen la misma frecuencia.
     public boolean mismaFrec(){
        int frec=frecuencia(v[0]);
        for (int j =0; j<this.cantElem; j++) {
            if(!(frecuencia(this.v[j])==frec))
            {
                return false;
            }
        }
        return true;
    }
//27.   L1.poker() : Método Lógico que devuelve True, si los elementos de la lista L1
forman poker. (Todos los elementos son iguales excepto uno)
   public boolean poker() {
        int p=0;
        int ele1 = this.v[p];
        while (p < this.cantElem) {
            if (ele1 != this.v[p]) {
                int carta1 = frecuencia(ele1);
                int carta2 = frecuencia(this.v[p]);
                if (carta1 == 1 && carta2 > 1 && (carta1 + carta2) == this.cantElem) {
```

```
                return true;
            } else {
                if (carta1 > 1 && carta2 == 1 && (carta1 + carta2) == this.cantElem) {
                    return true;
                }
            }
        }
        p++;
    }
    return false;
}
```

//28. L1.existePar() : Método lógico que devuelve True, si la lista L1 contiene al menos un elemento par.

```
public boolean existePar()
{
    for (int i = 0; i <this.cantElem; i++) {
        if(this.v[i]%2==0)
        {
            return true;
        }
    }
    return false;
}
```

//29. L1.existeImpar() : Método lógico que devuelve True, si la lista L1 contiene al menos un elemento impar.

```
public boolean existeImPar()
{
    for (int i = 0; i <this.cantElem; i++) {
        if(this.v[i]%2!=0)
        {
            return true;
        }
    }
    return false;
}
```

//30. L1.todoPares() : Método lógico que devuelve True, si todos los elementos de la lista L1 son pares.

```
public boolean todosPares()
{
    for (int i = 0; i <this.cantElem; i++) {
        if(this.v[i]%2!=0)
        {
            return false;
        }
    }
```

```
        return true;
    }
//31.   L1.todoImpares() : Método lógico que devuelve True, si todos los elementos de la
lista L1 son impares.
    public boolean todosImpares()
    {
        for (int i = 0; i <this.cantElem; i++) {
            if(this.v[i]%2==0)
            {
                return false;
            }
        }
        return true;
    }
//32.   L1.existeParImpar() : Método lógico que devuelve True, si en la lista L1 al menos
existe un elemento par y un elemento impar.
    public boolean existeParImpar()
    {
        return existePar()&& existeImPar();
    }
//33.   L1.alternos() : Método lógico que devuelve true, si la lista L1 contiene elementos en
la siguiente secuencia: par, impar, par, impar, . . . or  impar, par, impar, par, . . . .
    public boolean alternos()
    {
        return ParImpar() || ImparPar();
    }


    public boolean ParImpar()
    {
        for (int i = 0; i < this.cantElem; i++) {
            if(i%2==0)
            {
                if(this.v[i]%2!=0)
                    return false;
            }
            if(i%2!=0)
            {
                if(this.v[i]%2==0)
                    return false;
            }
        }
        return true;
    }
    public boolean ImparPar()
    {
        for (int i = 0; i < this.cantElem; i++) {
```

```
            if(i%2==0)
            {
               if(this.v[i]%2==0)
                  return false;
            }
            if(i%2!=0)
            {
               if(this.v[i]%2!=0)
                  return false;
            }
         }
         return true;
      }
//34.   L1.palindrome() : Método lógico que devuelve True, si la lista L1 contiene
elementos que forma un palíndrome. Ejemplo, caso anterior.
      public boolean palindrome() {
         int i = 0;
         int j = this.cantElem - 1;

         while (i < j) {
            if (this.v[i] != v[j]) {
               return false;
            }
            i++;
            j--;
         }

         return true;
      }
//35. L1.invertir() : Método que invierte los elementos de la lista L1.
      public void invertir()
      {
         int j=0;
         for (int i = this.cantElem-1; i >= (this.cantElem+1)/2; i--) {
            int ele=v[j];
            this.v[j]=this.v[i];

            v[i]=ele;
         }
      }
//ELIMINAR LOS ELEMENTOS DE UNA LISTA
//
//1. L1.eliminarPrim() : Método que elimina el primer elemento de la lista L1.
      public void eliminarPrim() {
         this.eliminarIesimo(0);
      }
//2. L1.eliminarUlt() : Método que elimina el último elemento de la lista L1.
```

```
    public void eliminarUlt() {
        this.eliminarIesimo(this.cantElem - 1);
    }
//3. L1.eliminarIesimo(i) : Método que elimina el i-ésimo elemento de la lista L1.
    public void eliminarIesimo(int i) {
        int k = i + 1;
        while (k < this.cantElem) {
            this.v[k - 1] = this.v[k];
            k = k + 1;
        }
        this.cantElem--;
    }
//4. L1.eliminarPrim(x) : Método que elimina el primer elemento x de la lista L1.
    public void eliminarPrim(int x) {
        boolean flag=true;
        int i=0;
       while(i<this.cantElem&& flag)
       {
          if(x==this.v[i])
          {
             eliminarIesimo(i);
             flag=false;
          }
          i++;
       }
    }
//5. L1.eliminarUlt(x) : Método que elimina el último elemento x de la lista L1.
    public void eliminarUlt(int x) {
        boolean flag=true;
        int i=this.cantElem-1;
       while(i>=0&& flag)
       {
          if(x==this.v[i])
          {
             eliminarIesimo(i);
             flag=false;
          }
          i--;
       }
    }
//6. L1.eliminarTodo( x ) : Método que elimina todos los elementos x de la lista L1.
    public void eliminarTodo(int x) {
        int i = 0;
        while (i < this.cantElem) {
            if (this.v[i] == x) {
                this.eliminarIesimo(i);
            }else{
```

```
            i++;
            }
        }
    }
//7. L1.eliminarPrim( n ) : Método que eliminar los primeros n-elementos de la lista L1.
    public void eliminarPrimm(int n) {
        if (n <= this.cantElem) {
            for (int i = n; i > 0; i--) {
                eliminarIesimo(0);
            }
        }

    }
//8. L1.eliminarUlt( n ) : Método que elimina los n-últimos elementos de la lista L1.
    public void eliminarUltimo(int n) {
        if (n <= this.cantElem) {
            for (int i = n; i > 0; i--) {
                eliminarUlt();
            }
        }
    }
//9. L1.eliminarIesimo(i, n) : Método que elimina los n-elementos de la lista L1, desde la
posición i.
    public void eliminarIesimo(int i,int n)
     {
        int contador=0;
        while(contador<n)
        {
            this.eliminarIesimo(i);
            i++;
            contador++;
        }
     }
//10. L1.eliminarExtremos( n ) : Método que eliminar la n-elementos de los extremos de la
lista L1.
    public void EliminarExtremos(int n) {
        boolean flag = true;
        while (n != 0 && cantElem != 0) {
            if (flag) {
                flag = false;
                eliminarPrim();
            } else {
                flag = true;
                eliminarUlt();
            }
            n--;
```

```
        }

    }
//11. L1.eliminarPares() : Método que elimina los elementos pares de la lista L1.
    public void eliminarPares() {
        int i = 0;
        while (i < this.cantElem) {
            if (this.v[i] % 2 == 0) {
                this.eliminarIesimo(i);
            }
            i++;
        }
    }
//12.L1.eliminarUnicos() : Método que elimina los elementos que aparecen solo una vez en
la lista L1.
     public void eliminarUnicos() {
        int i = 0;
        while (i < this.cantElem) {
            int j = 0;
            int contador = 0;
            while (j < this.cantElem) {
                if (this.v[i] == this.v[j]) {
                    contador++;
                }
                j++;
            }
            if (contador == 1) {
                eliminarIesimo(i);
            }
            i++;
        }
    }
//13 L1.eliminarTodo(L2) : Método que elimina todos los elementos de la lista L1, que
aparecen en la lista L2.
    public void eliminarTodo(Lista L2) {
        int j = 0;
        while (L2.cantElem > 0 && j < L2.cantElem) {
            int i = 0;
            int ele = L2.v[j];
            while (i < this.cantElem) {
                if (this.v[i] == ele) {
                    this.eliminarIesimo(i);
                } else {
                    i++;
                }
            }
            j++;
```

```
        }

    }
//14. L1.eliminarVeces(k) : Método que elimina los elementos que se repiten k-veces en la lista L1.
    public void eliminarVeces(int k)
    {
        int i=0;
        while(i<this.cantElem)
        {
            int contador=0;
            int ele=this.v[i];
            int j=0;
            while(j<this.cantElem)
            {
                if(ele==this.v[j])
                {
                    contador++;
                }
                j++;
            }
            if(contador==k)
            {
                this.eliminarTodo(ele);
            }
            i++;
        }
    }
//15. L1.eliminarAlternos() : Método que elimina los elementos de las posiciones alternas. (permanece, se elimina, permanece, se elimina, etc.)
    public void eliminarAlternos()
    {
        for(int i=1;i<this.cantElem;i++)
        {
            this.eliminarIesimo(i);
        }
    }
//16. L1.rotarIzqDer( n ) : Método que hace rotar los elementos de la lista L1 n-veces de izquierda a derecha.
    public void rotarIzqDer(int n) {
        for (int i = n; i > 0; i--) {
            int ele = this.v[this.cantElem - 1];
            eliminarUlt();
            insertarPrim(ele);
        }

    }
```

```
//17. L1.rotarDerIzq( n ) : Método que hace rotar los elementos de la lista L1 n-veces de derecha a izquierda.
    public void rotarDerIzq(int n) {

       for (int i = n; i > 0; i--) {
          int ele = this.v[0];
          eliminarPrim();
          insertarUlt(ele);
       }
    }
}
```

LISTA ENCADENADA SIMPLE

```
public class Nodo {
    public int elem;
    public Nodo prox;

    public Nodo(int elem, Nodo prox) {
        this.elem = elem;
        this.prox = prox;
    }

}
```

```
public class Lista {

    public Nodo prim;
    public int cantElem;
    public Nodo ult;

    public Lista() {
        this.prim = this.ult = null;
        this.cantElem = 0;
    }
```

//1. L1.toString() : Método que devuelve una cadena, que representa la secuencia de elementos de la lista L1.

```
    @Override
    public String toString() {
        String s = "[";
        Nodo p = this.prim;
        while (p != null) {
            s = s + p.elem;
            if (p.prox != null) {
                s = s + " ";
            }
            p = p.prox;
        }
        return s + "]";
    }
```

//2. L1.insertarPrim(x) : Método que inserta el elemento x, al final de la lista L1.

```
    public void insertarPrim(int x) {
        if (vacio()) {
            prim = ult = new Nodo(x, null);
```

```
        } else {
            prim = new Nodo(x, prim);
        }
        this.cantElem++;
    }
//3.   L1.insertarUlt(x) : Método que inserta el elemento x, al final de la lista L1.

    public void insertarUlt(int x) {
        if (vacio()) {
            prim = ult = new Nodo(x, null);
        } else {
            ult = ult.prox = new Nodo(x, null);
        }
        this.cantElem++;
    }
//4. L1.insertarIesimo(x, i) : Método que inserta el elemento x, en la posición i, de la lista
L1.

    public void insertarIesimo(int x, int i) {
        int k = 0;
        Nodo p = prim, ap = null;
        while (p != null && k < i) {
            ap = p;
            p = p.prox;
            k = k + 1;
        }
        insertarNodo(x, ap, p);

    }

    public void insertarNodo(int x, Nodo ap, Nodo p) {
        if (ap == null) {
            insertarPrim(x);
        } else if (p == null) {
            insertarUlt(x);
        } else {
            ap.prox = new Nodo(x, p);
            this.cantElem++;
        }
    }
//5. L1.insertarIesimo(L2, i) : Método que insertar los elementos de la lista L2 en la lista
L1, desde la posición i.

    public void insertarIesimo(Lista L2, int i) {
        Nodo p = this.prim;
        while (p != null) {
            insertarIesimo(p.elem, i);
```

```
            p = p.prox;
        }
    }
```
//6. L1.insertarPrim(L2) : Método que insertar los elementos de la lista L2 al principio de la lista L1.

```
    public void insertarPrim(Lista L2) {
        Nodo p = this.prim;
        while (p != null) {
            insertarPrim(p.elem);
            p = p.prox;
        }

    }
```
//7. L1.insertarUlt(L2) : Método que insertar los elementos de la lista L2 al final de la lista L1.

```
    public void insertarUlt(Lista L2) {
        while (L2 != null) {
            insertarUlt(L2.prim.elem);
            L2.prim = L2.prim.prox;

        }
    }
```
//8. L1.insertarAsc(x) : Método que inserta el elemento x en su lugar correspondiente, en la lista L1, ordenada en forma ascendente.

```
    public void insertarAsc(int x) {

        Nodo p = this.prim;
        Nodo a = null;
        while (p != null && p.elem < x) {
            a = p;
            p = p.prox;
        }
        insertarNodo(x, a, p);
    }
```

//9. L1.insertarDes(x) : Método que inserta el elemento x en su lugar correspondiente, en la lista L1, ordenada en forma descendente.

```
    public void insertarDes(int x) {

        Nodo p = this.prim;
        Nodo a = null;
        while (p != null && p.elem > x) {
            a = p;
            p = p.prox;
```

```
        }
        insertarNodo(x, a, p);
    }
//10. L1.concatenar(L2, L3): Método que concatena las listas L2 con L3 en L1.

    public void concatenar(Lista L2, Lista L3) {
        Nodo p2 = L2.prim;
        Nodo p3 = L3.prim;
        while (p2 != null) {
            insertarUlt(p2.elem);
            p2 = p2.prox;
        }
        while (p3 != null) {
            insertarUlt(p3.elem);
            p3 = p3.prox;
        }
    }
//11. L1.intercalar(L2, L3): Método que intercala los elementos de las Listas L2 con L3 en
L1.

    public void intercalar(Lista L2, Lista L3) {
        Nodo p2 = L2.prim;
        Nodo p3 = L3.prim;
        boolean flag = true;
        while (p2 != null && p3 != null) {
            if (flag) {
                insertarUlt(p2.elem);
                p2 = p2.prox;
                flag = false;
            } else {
                insertarUlt(p3.elem);
                p3 = p3.prox;
                flag = true;
            }
        }

        while (p2 != null) {
            insertarUlt(p2.elem);
            p2 = p2.prox;
        }
        while (p3 != null) {
            insertarUlt(p3.elem);
            p3 = p3.prox;
        }
    }
//12. L1.merge(L2, L3): Método que realiza el merge en L1, de las listas ordenadas en
forma ascedente L2 y L3.
```

```
    public void merge(Lista L2, Lista L3) {
        Nodo p2 = L2.prim;
        Nodo p3 = L3.prim;
        while (p2 != null && p3 != null) {
            if (p2.elem < p3.elem) {
                insertarUlt(p2.elem);
                p2 = p2.prox;

            } else {
                insertarUlt(p3.elem);
                p3 = p3.prox;

            }
        }
        while (p2 != null) {
            insertarUlt(p2.elem);
            p2 = p2.prox;
        }
        while (p3 != null) {
            insertarUlt(p3.elem);
            p3 = p3.prox;
        }
    }
```

//13. L1.iguales() : Método Lógico que devuelve True, si todos los elementos de la lista L1 son iguales.

```
    public boolean iguales() {
        int elemento = prim.elem;
        Nodo p = prim;
        while (p != null) {
            if (elemento != p.elem) {
                return false;
            }
            p = p.prox;
        }
        return true;
    }
```

//14. L1.diferentes() : Método Lógico que devuelve True, si todos los elementos de la lista L1 son diferentes.

```
    public boolean diferentes() {
        Nodo cabeza = prim;
        while (cabeza != null) {
            int elemento = cabeza.elem;
            Nodo copia = cabeza.prox;
            while (copia != null) {
```

```
            if (elemento == copia.elem) {
                return false;
            }
        }
    }
    return true;
}
```

//15. L1.mayorElem() : Método que devuelve el mayor elemento de la lista L1.

```
public int mayorElem() {
    int ele = this.prim.elem;
    while (this.prim != null) {
        if (!(ele > this.prim.elem)) {
            ele = this.prim.elem;
        }
        prim = prim.prox;
    }
    return ele;

}
```

//16. L1.menorElem() : Método que devuelve el mayor elemento de la lista L1.

```
public int menorElem() {
    Nodo P = prim;
    int ele = P.elem;
    while (P != null) {
        if (ele > P.elem) {
            ele = P.elem;
        }
        P = P.prox;
    }
    return ele;

}
```

//17. L1.ordenado() : Método Lógico que devuelve True, si todos los elementos de la lista L1 están ordenados en forma ascendente o descendente.

```
public boolean ordenado() {
    return Ascendente() || Descendente();
}

public boolean Ascendente() {
    Nodo P = prim;
    int elemento = P.elem;
    while (P != null) {
        if (elemento <= P.elem) {
            elemento = P.elem;
```

```
            } else {
                return false;
            }

            P = P.prox;
        }
        return true;
    }

    public boolean Descendente() {
        Nodo P = prim;
        int elemento = P.elem;
        while (P != null) {
            if (elemento >= P.elem) {
                elemento = P.elem;
            } else {
                return false;
            }

            P = P.prox;
        }
        return true;
    }
```

//18. L1.indexOf(x) : Método que devuelve la posición de la primera ocurrencia del elemento x. Si x no se encuentra en la lista L1, el método devuelve –1.

```
    public int indexOf(int x) {
        Nodo p = this.prim;
        int i = 0;
        while (p != null) {
            if (x == this.prim.elem) {
                return i;
            }
            p = p.prox;
            i++;
        }

        return -1;
    }
```

//19. L1.indexOf(x, i) : Método que devuelve la posición de la primera ocurrencia del elemento x, la búsqueda se realiza desde la posición i.

```
    public int indexOf(int x, int i) {
        Nodo p = this.prim;
        int j = 0;
        while (p != null) {
            if (i >= j) {
```

```
            if (p.elem == x) {
                return i;
            }
            i++;
        }
        p = p.prox;
        j++;
    }

    return -1;
}
```

//20. L1.lastIndexOf(x) : Método que devuelve la posición de la última ocurrencia del elemento x. Si x no se encuentra en la lista L1, el método devuelve –1.

```
public int lastIndexOF(int x) {
    Nodo p = this.ult;
    int i = 0;
    int indice = -1;
    boolean flag = false;
    while (p != null) {
        if (x == this.prim.elem) {
            flag = true;
            indice = i;

        }
        p = p.prox;
        i++;
    }
    if (flag) {
        return indice;
    } else {
        return -1;
    }
}
```

//21. L1.lastIndexOf(x, i) : Método que devuelve la posición de la última ocurrencia del elemento x. Si x no se encuentra en la lista L1, el método devuelve –1. La búsqueda se realiza desde la posición i.

```
public int lastIndexOF(int x, int i) {
    Nodo p = this.ult;
    int j = 0, indice = -1;
    boolean flag = false;
    while (p != null) {
        if (i >= j) {
            if (x == this.prim.elem) {
                flag = true;
                indice = j;
```

```
            }
        }
        i++;
        p = p.prox;
        j++;
    }
    if (flag) {
        return indice;
    } else {
        return -1;
    }
}
```
//22. L1.reemplazar(x, y) : Método que reemplaza todas las ocurrencias del elemento x por el elemento y en la lista L1.

```
public void reemplazar(int x, int y) {
    Nodo p = this.prim;
    while (p != null) {
        if (p.elem == x) {
            p.elem = y;
        }
        p = p.prox;
    }
}
```
//23. L1.seEncuentra(x) : Método Lógico que devuelve True, si el elemento x, se encuentra en la lista L1.

```
public boolean seEncuentra(int x) {
    Nodo p = this.prim;
    while (p != null) {
        if (p.elem == x) {
            return true;
        }
        p = p.prox;
    }
    return false;
}
```
//24. L1.frecuencia(x) : Método que devuelve la cantidad de veces que aparece el elemento x en la lista L1.

```
public int frecuencia(int x)
{
    Nodo p=prim;
    int contador=0;
    while(p!=null)
    {
        if(x==p.elem)
```

```
            {
                contador++;
            }
            p=p.prox;
        }
        return contador;
    }
```
//25. L1.existeFrec(k) : Método Lógico que devuelve True, si existe algún elemento que se repite exactamente k-veces en la lista L1.
```
    public boolean existeFrec(int x)
    {
        Nodo p=prim;
        while(p!=null)
        {
            if(x==frecuencia(p.elem))
            {
                return true;
            }
            p=p.prox;
        }
        return false;
    }
```
//26. L1.mismasFrec() : Método Lógico que devuelve True, si los elementos de la lista L1 tienen la misma frecuencia.
```
     public boolean mismaFrec()
    {
        Nodo p=prim;
        int frec=frecuencia(p.elem);
        while(p!=null)
        {
            if(frec!=frecuencia(p.elem))
            {
                return false;
            }
            p=p.prox;
        }
        return true;
    }
```
//27. L1.poker() : Método Lógico que devuelve True, si los elementos de la lista L1 forman poker. (Todos los elementos son iguales excepto uno)

```
    public boolean poker() {
        Nodo P = this.prim;
        int ele1 = P.elem;
        while (P != null) {
            if (ele1 != P.elem) {
                int carta1 = frecuencia(ele1);
```

```
            int carta2 = frecuencia(P.elem);
            if (carta1 == 1 && carta2 > 1 && (carta1 + carta2) == this.cantElem) {
               return true;
            } else {
               if (carta1 > 1 && carta2 == 1 && (carta1 + carta2) == this.cantElem) {
                  return true;
               }
            }
         }
         P = P.prox;
      }
      return false;
   }
```
//28. L1.existePar() : Método lógico que devuelve True, si la lista L1 contiene al menos un elemento par.

```
   public boolean existePar() {
      Nodo p = prim;
      while (p != null) {
         if (p.elem % 2 == 0) {
            return true;
         }
         p = p.prox;
      }
      return false;
   }
```
//29. L1.existeImpar() : Método lógico que devuelve True, si la lista L1 contiene al menos un elemento impar.

```
   public boolean existeImpar() {
      Nodo p = prim;
      while (p != null) {
         if (p.elem % 2 != 0) {
            return true;
         }
         p = p.prox;
      }
      return false;
   }
```
//30. L1.todoPares() : Método lógico que devuelve True, si todos los elementos de la lista L1 son pares.

```
   public boolean todosPares() {
      Nodo p = prim;
      while (p != null) {
         if (!(p.elem % 2 == 0)) {
            return false;
```

```
            }
            p = p.prox;
        }
        return true;
    }
```
//31. L1.todoImpares() : Método lógico que devuelve True, si todos los elementos de la lista L1 son impares.
```
     public boolean todosImpares() {
        Nodo p = prim;
        while (p != null) {
            if ((p.elem % 2 == 0)) {
                return false;
            }
            p = p.prox;
        }
        return true;
    }
```
//32. L1.existeParImpar() : Método lógico que devuelve True, si en la lista L1 al menos existe un elemento par y un elemento impar.

```
    public boolean parImpar() {
        Nodo p = prim;
        boolean par = false, impar = false;
        while (p != null) {
            if (p.elem % 2 == 0) {
                par = true;
            } else {
                impar = true;
            }
            p = p.prox;
        }
        return par && impar;
    }
```
//33. L1.alternos() : Método lógico que devuelve true, si la lista L1 contiene elementos en la siguiente secuencia: par, impar, par, impar, . . . or impar, par, impar, par, ...
```
    public boolean alternos()
    {
        return ParImpar() || ImparPar();
    }


    public boolean ParImpar()
    {
        Nodo p=prim;
        boolean flag=true;
        while(p!=null)
        {
```

```
            if(flag)
            {
               if(p.elem%2!=0)
                  return false;
               flag=false;

            }else{
               if(p.elem%2==0)
               {
                  return false;
               }else{
                  flag=true;
               }
            }
            p=p.prox;
         }
         return true;

      }
      public boolean ImparPar()
      {
         Nodo p=prim;
         boolean flag=true;
         while(p!=null)
         {
            if(flag)
            {
               if(p.elem%2==0)
                  return false;
               flag=false;

            }else{
               if(p.elem%2!=0)
               {
                  return false;
               }else{
                  flag=true;
               }
            }
            p=p.prox;
         }
         return true;
      }
//34.   L1.palindrome() : Método lógico que devuelve True, si la lista L1 contiene
elementos que forma un palíndrome. Ejemplo, caso anterior. 7
      public boolean palindrome()
      {
```

```
//        Nodo p = invertir();


        while(prim!=null)
        {

        }
        return false;
    }
//35. L1.invertir() : Método que invierte los elementos de la lista L1.
    public void invertir()
    {

        invertirR(prim);

    }
    public void invertirR(Nodo p)
    {
        if(p.prox!=null)
        {
            prim=ult=p;
        }else{
            invertirR(p.prox);
            ult=ult.prox=new Nodo(p.elem,null);

        }
    }
//
//
//ELIMINAR LOS ELEMENTOS DE UNA LISTA
//
//1. L1.eliminarPrim() : Método que elimina el primer elemento de la lista L1.
    public void eliminaPrim()
    {
        eliminarIesimo(0);
    }
//2. L1.eliminarUlt() : Método que elimina el último elemento de la lista L1.
    public void eliminarUlt()
    {
        eliminarIesimo(this.cantElem-1);
    }
//3. L1.eliminarIesimo(i) : Método que elimina el i-ésimo elemento de la lista L1.
   public void eliminarIesimo(int i)
   {
       int k=0;
       Nodo p=prim;
       Nodo a=null;
```

```
        while(p!=null )
        {
            if(k==i)
            {
                unirNodos(a,p.prox);
            }
            a=p;
            p=p.prox;
            k++;
        }
    }

    public void unirNodos(Nodo a, Nodo p)
    {
        if(a==null)
        {
            this.prim=p;
        }else if(p==null){
            a.prox=p;
        }else{
            a.prox=p;
        }

    }

//4. L1.eliminarPrim(x) : Método que elimina el primer elemento x de la lista L1.
    public void eliminarPrim()
    {
        eliminarIesimo(0);
    }
//5. L1.eliminarUlt(x) : Método que elimina el último elemento x de la lista L1.
    public void eliminarUlt(int x)
    {
        eliminarIesimo(this.cantElem-1);
    }
//6. L1.eliminarTodo( x ) : Método que elimina todos los elementos x de la lista L1.

    public void eliminarTodo(int x)
    {
        Nodo p=this.prim;
        int i=0;
        while(p!=null)
        {
            if(x==p.elem)
            {
                eliminarIesimo(i);
            }else{
```

```
            p=p.prox;
            i++;
          }
        }
    }
//7. L1.eliminarPrim( n ) : Método que eliminar los primeros n-elementos de la lista L1.
    public void eliminarPrim(int n)
    {
        Nodo p=prim;
        while(p!=null&& n>0){
            eliminarPrim();
            n--;

        }
    }
//8. L1.eliminarUlt( n ) : Método que elimina los n-últimos elementos de la lista L1.

//9. L1.eliminarIesimo(i, n) : Método que elimina los n-elementos de la lista L1, desde la
posición i.
//
//10. L1.eliminarExtremos( n ) : Método que eliminar la n-elementos de los extremos de la
lista L1.
//
//11. L1.eliminarPares() : Método que elimina los elementos pares de la lista L1.
//
//12.L1.eliminarUnicos() : Método que elimina los elementos que aparecen solo una vez en
la lista L1.
//
//13 L1.eliminarTodo(L2) : Método que elimina todos los elementos de la lista L1, que
aparecen en la lista L2.
//
//14. L1.eliminarVeces(k) : Método que elimina los elementos que se repiten k-veces en la
lista L1.
//
//15. L1.eliminarAlternos() : Método que elimina los elementos de las posiciones alternas.
(permanece, se elimina, permanece, se elimina, etc.)
//
//16. L1.rotarIzqDer( n ) : Método que hace rotar los elementos de la lista L1 n-veces de
izquierda a derecha.
//
//17. L1.rotarDerIzq( n ) : Método que hace rotar los elementos de la lista L1 n-veces de
derecha a izquierda.

    public boolean vacio() {
        return this.cantElem == 0;
    }
```

//11. L1.reemplazar(x, y) : Método que reemplaza todas las ocurrencias del elemento x por el elemento y en la lista L1.

```
    public void insertarLugar(int x) {
        Nodo p = prim, ap = null;
        while (p != null && x > p.elem) {
            ap = p;
            p = p.prox;
        }
        insertarNodo(x, ap, p);
    }

    public static void main(String[] args) {
        Lista L1 = new Lista();
        L1.insertarIesimo(4, 0);
        L1.insertarIesimo(4, 1);
        L1.insertarIesimo(5, 2);
        L1.insertarUlt(4);
//        L1.insertarUlt(5);
        System.out.println(L1.toString());

        Lista L2 = new Lista();
        L2.insertarIesimo(6, 0);
        L2.insertarIesimo(4, 1);
        L2.insertarIesimo(2, 2);
//        System.out.println(L2.toString());
//        L1.reemplazar(4,44);
        //System.out.println(L1.ordenado());
        System.out.println(L1.poker());
    }
}
```

LISTA DOBLEMENTE ENCADENADA

```
public class Nodo {
   public Nodo ant;
   public int elem;
   public Nodo prox;
   public Nodo(Nodo ant,int x,Nodo prox)
   {
      this.ant=ant;
      this.elem=x;
      this.prox=prox;

   }


}
```

```
public class Lista {

   public Nodo prim;
   public Nodo ult;
   public int cantElem;

   public Lista() {
      prim = ult = null;
      this.cantElem = 0;
   }
```

//1. L1.toString() : Método que devuelve una cadena, que representa la secuencia de elementos de la lista L1.

```
   @Override
   public String toString() {
      String s1 = "[";
      Nodo p = prim;
      while (p != null) {
         s1 = s1 + p.elem;
         if (p.prox != null) {
            s1 = s1 + "  ";
         }
         p = p.prox;
      }
      return s1 + "]";
```

```
  }
//2.  L1.insertarPrim(x) : Método que inserta el elemento x, al final de la lista L1.

  public void insertarPrim(int x) {
    if (vacia()) {
      prim = ult = new Nodo(null, x, null);

    } else {
      prim = prim.ant = new Nodo(null, x, prim);

    }
    cantElem = cantElem + 1;
  }
//3.  L1.insertarUlt(x) : Método que inserta el elemento x, al final de la lista L1.

  public void insertarUlt(int x) {
    if (vacia()) {
      prim = ult = new Nodo(null, x, null);
    } else {
      ult = ult.prox = new Nodo(ult, x, null);

    }
    cantElem = cantElem + 1;
  }
//4. L1.insertarIesimo(x, i) : Método que inserta el elemento x, en la posición i, de la lista
L1.

  public void insertarIesimo(int x, int i) {
    int k = 0;
    Nodo p = prim, ap = null;
    while (k < i && p != null) {
      ap = p;
      p = p.prox;
      k = k + 1;

    }
    insertarNodo(ap, p, x);
  }
//5. L1.insertarIesimo(L2, i) : Método que insertar los elementos de la lista L2 en la lista
L1, desde la posición i.

  public void insertarIesimo(Lista L2, int i) {

    Nodo p = L2.prim;
```

```
      while (p != null) {

         insertarIesimo(p.elem, i);
         p = p.prox;
         i++;

      }
   }
```
//6. L1.insertarPrim(L2) : Método que insertar los elementos de la lista L2 al principio de la lista L1.

```
   public void insertarPrim(Lista L2) {
      Nodo p = L2.prim;
      int i = 0;
      while (p != null) {
         insertarIesimo(p.elem, i);
         p = p.prox;
         i++;
      }
   }
```
//7. L1.insertarUlt(L2) : Método que insertar los elementos de la lista L2 al final de la lista L1.

```
   public void insertarUlt(Lista L2) {
      while (L2 != null) {
         insertarUlt(L2.prim.elem);
         L2.prim = L2.prim.prox;

      }
   }
```
//8. L1.insertarAsc(x) : Método que inserta el elemento x en su lugar correspondiente, en la lista L1, ordenada en forma ascendente.

```
   public void insertarLugarAsc(int x) {
      Nodo p = prim, ap = null;
      while (p != null && p.elem < x) {
         ap = p;
         p = p.prox;

      }
      insertarNodo(ap, p, x);
   }
```
//9. L1.insertarDes(x) : Método que inserta el elemento x en su lugar correspondiente, en la lista L1, ordenada en forma descendente.

```
  public void insertarLugarDes(int x) {
    Nodo p = prim, ap = null;
    while (p != null && p.elem > x) {
      ap = p;
      p = p.prox;

    }
    insertarNodo(ap, p, x);
  }
//10. L1.concatenar(L2, L3): Método que concatena las listas L2 con L3 en L1.

  public void concatenar(Lista L2, Lista L3) {
    Nodo p2 = L2.prim;
    Nodo p3 = L3.prim;
    while (p2 != null) {
      insertarUlt(p2.elem);
      p2 = p2.prox;
    }
    while (p3 != null) {
      insertarUlt(p3.elem);
      p3 = p3.prox;
    }
  }
//11. L1.intercalar(L2, L3): Método que intercala los elementos de las Listas L2 con L3 en
L1.

  public void intercalar(Lista L2, Lista L3) {
    Nodo p2 = L2.prim;
    Nodo p3 = L3.prim;
    boolean flag = true;
    while (p2 != null && p3 != null) {
      if (flag) {
        insertarUlt(p2.elem);
        p2 = p2.prox;
        flag = false;
      } else {
        insertarUlt(p3.elem);
        p3 = p3.prox;
        flag = true;
      }
    }

    while (p2 != null) {
```

```
            insertarUlt(p2.elem);
            p2 = p2.prox;
        }
        while (p3 != null) {
            insertarUlt(p3.elem);
            p3 = p3.prox;
        }
    }
```

//12. L1.merge(L2, L3): Método que realiza el merge en L1, de las listas ordenadas en forma ascedente L2 y L3.

```
    public void merge(Lista L2, Lista L3) {
        Nodo p2 = L2.prim;
        Nodo p3 = L3.prim;
        while (p2 != null && p3 != null) {
            if (p2.elem < p3.elem) {
                insertarUlt(p2.elem);
                p2 = p2.prox;

            } else {
                insertarUlt(p3.elem);
                p3 = p3.prox;

            }
        }
        while (p2 != null) {
            insertarUlt(p2.elem);
            p2 = p2.prox;
        }
        while (p3 != null) {
            insertarUlt(p3.elem);
            p3 = p3.prox;
        }
    }
```

//13. L1.iguales() : Método Lógico que devuelve True, si todos los elementos de la lista L1 son iguales.

```
    public boolean iguales() {
        int elemento = prim.elem;
        Nodo p = prim;
        while (p != null) {
            if (elemento != p.elem) {
                return false;
            }
```

```
            p = p.prox;
        }
        return true;
    }
//14.   L1.diferentes() : Método Lógico que devuelve True, si todos los elementos de la
lista L1 son diferentes.

    public boolean diferentes() {
        Nodo cabeza = prim;
        while (cabeza != null) {
            int elemento = cabeza.elem;
            Nodo copia = cabeza.prox;
            while (copia != null) {
                if (elemento == copia.elem) {
                    return false;
                }
            }
        }
        return true;
    }
//15.   L1.mayorElem() : Método que devuelve el mayor elemento de la lista L1.

    public int mayorElem() {
        int ele = this.prim.elem;
        while (this.prim != null) {
            if (!(ele > this.prim.elem)) {
                ele = this.prim.elem;
            }
            prim = prim.prox;
        }
        return ele;

    }
//16.    L1.menorElem() : Método que devuelve el mayor elemento de la lista L1.

    public int menorElem() {
        Nodo P = prim;
        int ele = P.elem;
        while (P != null) {
            if (ele > P.elem) {
                ele = P.elem;
            }
            P = P.prox;
        }
```

```
        return ele;

    }
```
//17. L1.ordenado() : Método Lógico que devuelve True, si todos los elementos de la lista L1 están ordenados en forma ascendente o descendente.

```
    public boolean ordenado() {
        return Ascendente() || Descendente();
    }

    public boolean Ascendente() {
        Nodo P = prim;
        int elemento = P.elem;
        while (P != null) {
            if (elemento <= P.elem) {
                elemento = P.elem;
            } else {
                return false;
            }

            P = P.prox;
        }
        return true;
    }

    public boolean Descendente() {
        Nodo P = prim;
        int elemento = P.elem;
        while (P != null) {
            if (elemento >= P.elem) {
                elemento = P.elem;
            } else {
                return false;
            }

            P = P.prox;
        }
        return true;
    }
```
//18. L1.indexOf(x) : Método que devuelve la posición de la primera ocurrencia del elemento x. Si x no se encuentra en la lista L1, el método devuelve –1.

```
    public int indexOf(int x) {
        Nodo p = this.prim;
```

```
    int i = 0;
    while (p != null) {
      if (x == this.prim.elem) {
        return i;
      }
      p = p.prox;
      i++;
    }

    return -1;
  }
```
//19. L1.indexOf(x, i) : Método que devuelve la posición de la primera ocurrencia del elemento x, la búsqueda se realiza desde la posición i.

```
  public int indexOf(int x, int i) {
    Nodo p = this.prim;
    int j = 0;
    while (p != null) {
      if (i >= j) {
        if (p.elem == x) {
          return i;
        }
        i++;
      }
      p = p.prox;
      j++;
    }

    return -1;
  }
```
//20. L1.lastIndexOf(x) : Método que devuelve la posición de la última ocurrencia del elemento x. Si x no se encuentra en la lista L1, el método devuelve –1.

```
  public int lastIndexOF(int x) {
    Nodo p = this.ult;
    int i = 0;
    int indice = -1;
    boolean flag = false;
    while (p != null) {
      if (x == this.prim.elem) {
        flag = true;
        indice = i;

      }
```

```
            p = p.prox;
            i++;
        }
        if (flag) {
            return indice;
        } else {
            return -1;
        }
    }
```

//21. L1.lastIndexOf(x, i) : Método que devuelve la posición de la última ocurrencia del elemento x. Si x no se encuentra en la lista L1, el método devuelve –1. La búsqueda se realiza desde la posición i.

```
    public int lastIndexOF(int x, int i) {
        Nodo p = this.ult;
        int j = 0, indice = -1;
        boolean flag = false;
        while (p != null) {
            if (i >= j) {
                if (x == this.prim.elem) {
                    flag = true;
                    indice = j;

                }
            }
            i++;
            p = p.prox;
            j++;
        }
        if (flag) {
            return indice;
        } else {
            return -1;
        }
    }
```

//22. L1.reemplazar(x, y) : Método que reemplaza todas las ocurrencias del elemento x por el elemento y en la lista L1.

```
    public void reemplazar(int x, int y) {
        Nodo p = this.prim;
        while (p != null) {
            if (p.elem == x) {
                p.elem = y;
            }
```

```
        p = p.prox;
      }
    }
```
//23. L1.seEncuentra(x) : Método Lógico que devuelve True, si el elemento x, se encuentra en la lista L1.

```
    public boolean seEncuentra(int x) {
      Nodo p = this.prim;
      while (p != null) {
        if (p.elem == x) {
          return true;
        }
        p = p.prox;
      }
      return false;
    }
```
//24. L1.frecuencia(x) : Método que devuelve la cantidad de veces que aparece el elemento x en la lista L1.

```
    public int frecuencia(int x) {
      Nodo p = prim;
      int contador = 0;
      while (p != null) {
        if (x == p.elem) {
          contador++;
        }
        p = p.prox;
      }
      return contador;
    }
```
//25. L1.existeFrec(k) : Método Lógico que devuelve True, si existe algún elemento que se repite exactamente k-veces en la lista L1.

```
    public boolean existeFrec(int x) {
      Nodo p = prim;
      while (p != null) {
        if (x == frecuencia(p.elem)) {
          return true;
        }
        p = p.prox;
      }
      return false;
    }
```

//26. L1.mismasFrec() : Método Lógico que devuelve True, si los elementos de la lista L1 tienen la misma frecuencia.

```
public boolean mismaFrec() {
    Nodo p = prim;
    int frec = frecuencia(p.elem);
    while (p != null) {
        if (frec != frecuencia(p.elem)) {
            return false;
        }
        p = p.prox;
    }
    return true;
}
```

//27. L1.poker() : Método Lógico que devuelve True, si los elementos de la lista L1 forman poker. (Todos los elementos son iguales excepto uno)

```
public boolean poker() {
    Nodo P = this.prim;
    int ele1 = P.elem;
    while (P != null) {
        if (ele1 != P.elem) {
            int carta1 = frecuencia(ele1);
            int carta2 = frecuencia(P.elem);
            if (carta1 == 1 && carta2 > 1 && (carta1 + carta2) == this.cantElem) {
                return true;
            } else {
                if (carta1 > 1 && carta2 == 1 && (carta1 + carta2) == this.cantElem) {
                    return true;
                }
            }
        }
        P = P.prox;
    }
    return false;
}
```

//28. L1.existePar() : Método lógico que devuelve True, si la lista L1 contiene al menos un elemento par.

```
public boolean existePar() {
    Nodo p = prim;
    while (p != null) {
        if (p.elem % 2 == 0) {
            return true;
```

```
        }
        p = p.prox;
      }
      return false;
   }
```
//29. L1.existeImpar() : Método lógico que devuelve True, si la lista L1 contiene al menos un elemento impar.

```
   public boolean existeImpar() {
      Nodo p = prim;
      while (p != null) {
         if (p.elem % 2 != 0) {
            return true;
         }
         p = p.prox;
      }
      return false;
   }
```
//30. L1.todoPares() : Método lógico que devuelve True, si todos los elementos de la lista L1 son pares.

```
   public boolean todosPares() {
      Nodo p = prim;
      while (p != null) {
         if (!(p.elem % 2 == 0)) {
            return false;
         }
         p = p.prox;
      }
      return true;
   }
```
//31. L1.todoImpares() : Método lógico que devuelve True, si todos los elementos de la lista L1 son impares.

```
   public boolean todosImpares() {
      Nodo p = prim;
      while (p != null) {
         if ((p.elem % 2 == 0)) {
            return false;
         }
         p = p.prox;
      }
      return true;
   }
```

//32. L1.existeParImpar() : Método lógico que devuelve True, si en la lista L1 al menos existe un elemento par y un elemento impar.

```
public boolean parImpar() {
   Nodo p = prim;
   boolean par = false, impar = false;
   while (p != null) {
      if (p.elem % 2 == 0) {
         par = true;
      } else {
         impar = true;
      }
      p = p.prox;
   }
   return par && impar;
}
```

//33. L1.alternos() : Método lógico que devuelve true, si la lista L1 contiene elementos en la siguiente secuencia: par, impar, par, impar, . . . or impar, par, impar, par, ...

```
public boolean alternos() {
   return ParImpar() || ImparPar();
}

public boolean ParImpar() {
   Nodo p = prim;
   boolean flag = true;
   while (p != null) {
      if (flag) {
         if (p.elem % 2 != 0) {
            return false;
         }
         flag = false;

      } else {
         if (p.elem % 2 == 0) {
            return false;
         } else {
            flag = true;
         }
      }
      p = p.prox;
   }
   return true;
```

```
    }

    public boolean ImparPar() {
        Nodo p = prim;
        boolean flag = true;
        while (p != null) {
            if (flag) {
                if (p.elem % 2 == 0) {
                    return false;
                }
                flag = false;

            } else {
                if (p.elem % 2 != 0) {
                    return false;
                } else {
                    flag = true;
                }
            }
            p = p.prox;
        }
        return true;
    }
//34.   L1.palindrome() : Método lógico que devuelve True, si la lista L1 contiene
elementos que forma un palíndrome. Ejemplo, caso anterior. 7

    public boolean palindrome() {
//        Nodo p = invertir();

        while (prim != null) {

        }
        return false;
    }
//35. L1.invertir() : Método que invierte los elementos de la lista L1.

    public void invertir() {

        invertirR(prim);

    }

    public void invertirR(Nodo p) {
        if (p.prox != null) {
```

```
            prim = ult = p;
        } else {
            invertirR(p.prox);
//          ult=ult.prox=new Nodo(p.elem,null);

        }
    }

//
//
//
//ELIMINAR LOS ELEMENTOS DE UNA LISTA
//
//1. L1.eliminarPrim() : Método que elimina el primer elemento de la lista L1.
    public void eliminarPrim() {
        if (vacia()) {
            return;
        }
        if (prim == ult) {
            prim = ult = null;
        } else {
            prim.prox.ant = null;
            prim = prim.prox;

        }
        cantElem--;

    }
//2. L1.eliminarUlt() : Método que elimina el último elemento de la lista L1.

    public void eliminarUlt() {
        if (vacia()) {
            return;
        }
        if (prim == ult) {
            prim = ult = null;

        } else {
            ult.ant.prox = null;
            ult = ult.ant;

        }
        cantElem--;
```

```
  }
//3. L1.eliminarIesimo(i) : Método que elimina el i-ésimo elemento de la lista L1.

  public void eliminarIesimo(int i) {
    int k = 0;
    Nodo p = prim, ap = null;
    while (k < i && p != null) {
      ap = p;
      p = p.prox;
      k = k + 1;
    }
    eliminarNodo(ap, p);
  }

  public Nodo eliminarNodo(Nodo ap, Nodo p) {
    if (p == null) {
      return null;
    }
    if (ap == null) {
      eliminarPrim();
      return prim;
    }
    if (p.prox == null) {
      eliminarUlt();
      return null;
    }
    ap.prox = p.prox;
    p.prox.ant = ap;
    cantElem--;
    return ap.prox;
  }
//4. L1.eliminarPrim(x) : Método que elimina el primer elemento x de la lista L1.

//5. L1.eliminarUlt(x) : Método que elimina el último elemento x de la lista L1.
//
//6. L1.eliminarTodo( x ) : Método que elimina todos los elementos x de la lista L1.
//
//7. L1.eliminarPrim( n ) : Método que eliminar los primeros n-elementos de la lista L1.
  public void eliminarPrim(int n) {
    if (vacia()) {
      return;
    }
    while (prim != null && n > 0) {
      eliminarPrim();
```

```
        n--;
      }
    }
//8. L1.eliminarUlt( n ) : Método que elimina los n-últimos elementos de la lista L1.

    public void eliminarUlt(int n) {
      if (vacia()) {
        return;
      }
      while (prim != null && n > 0) {
        eliminarUlt();
        n--;
      }
    }
//9. L1.eliminarIesimo(i, n) : Método que elimina los n-elementos de la lista L1, desde la
posición i.
//
//10. L1.eliminarExtremos( n ) : Método que eliminar la n-elementos de los extremos de la
lista L1.
//
//11. L1.eliminarPares() : Método que elimina los elementos pares de la lista L1.
//
//12.L1.eliminarUnicos() : Método que elimina los elementos que aparecen solo una vez
en la lista L1.
//
//13 L1.eliminarTodo(L2) : Método que elimina todos los elementos de la lista L1, que
aparecen en la lista L2.
//
//14. L1.eliminarVeces(k) : Método que elimina los elementos que se repiten k-veces en la
lista L1.
//
//15. L1.eliminarAlternos() : Método que elimina los elementos de las posiciones alternas.
(permanece, se elimina, permanece, se elimina, etc.)
//
//16. L1.rotarIzqDer( n ) : Método que hace rotar los elementos de la lista L1 n-veces de
izquierda a derecha.
//
//17. L1.rotarDerIzq( n ) : Método que hace rotar los elementos de la lista L1 n-veces de
derecha a izquierda.

    public void insertarNodo(Nodo ap, Nodo p, int x) {
      if (ap == null) {
        insertarPrim(x);
      } else if (p == null) {
```

```
            insertarUlt(x);
        } else {
            ap.prox = p.ant = new Nodo(ap, x, p);
            this.cantElem++;
        }
    }

    public boolean vacia() {
        return this.cantElem == 0;
    }

//    public void insertarLugar(int x) {
//        Nodo p = prim, ap = null;
//        while (p != null && x > p.elem) {
//            ap = p;
//            p = p.prox;
//
//        }
//        insertarNodo(ap, p, x);
//
//    }

    public static void main(String[] args) {
        Lista L1 = new Lista();
        L1.insertarIesimo(4, 0);

        L1.insertarUlt(1);
//        L1.insertarUltm(22);
        //   L1.insertarLugarAsc(6);
        System.out.println(L1.palindrome());
        Lista L2 = new Lista();
        L2.insertarIesimo(6, 0);
        L2.insertarUlt(5);
        System.out.println(L1);

        System.out.println(L2);
        L1.insertarIesimo(L2, 2);
        System.out.println(L1);

    }
}
```