Lab. Estructuras de Datos. 6 de marzo del 2022.

Dylan Antonio Elías Vásquez.

201931369.

Documentación Técnica

Centro Universitario De Occidente (CUNOC).

División Ciencias de la Ingeniería.

Ingeniería en Ciencias y Sistemas.

Calculo de la Complejidad de Métodos Implementados

Servicios Críticos:

1.

```java
public class BetsList {
    public void add(Apuesta data) {
        Node nodo = new Node(data); // ← pasos = 1

        if (root ≠ null) { // ← pasos = 2
            Node leaf = root.getLeft(); // ← pasos = 4
            root.setLeft(nodo); // ← pasos = 8
            nodo.setLeft(leaf); // ← pasos = 12
        } else {
            nodo.setLeft(nodo); // ← pasos = 6
        }
        root = nodo; // ← pasos: peor caso = 13, mejor caso = 7
        length++; // ← pasos: peor caso = 14, mejor caso = 8
    }
}

class Node {
    public void setLeft(Node left) { // pasos = 1
        this.left = left; // pasos = 2

        if (left ≠ null) // pasos = 3, mejor caso
            left.right = this; // pasos = 4, peor caso
    }

    public Node getLeft() {
        return left; // solo 1 paso
    }
}
```

En el peor de los casos, al *ingresar apuestas antes del inicio de la carrera*, vamos a realizar 14 pasos en total, este numero es invariable, por lo cual, su complejidad seria constante y después de quitar las constantes innecesarias tendríamos que es ‘O(1)’.

2.

```java
public class BetsList {
    public void filterBets() { // peor caso
        Node anchor = root; // ← pasos = 1

        int lastLength = length; // ← pasos = 2
        for (int item = 0; item < lastLength; item++) { // ← pasos = 4 + 2n
            DigitSet set = new DigitSet(); // ← pasos for = 1
            int[] caballos = anchor.getData().getApuestas(); // ← pasos for = 3

            try {
                if (caballos.length ≠ 10) // ← pasos for = 4
                    throw new BetsOutOfBounds(10 < caballos.length);

                for (int i = 0; i < 10; i++) { // ← pasos for = 2 + 2m ⇒ peor caso m = 10 ⇒ 22
                    set.add(caballos[i]); // ← pasos for for = 7
                }
            } catch (NotADigitException | RepeatedDigitExpection | BetsOutOfBounds e) {
                if (length > 1) {
                    anchor.getLeft().setRight(anchor.getRight());
                    if (anchor == root) {
                        root = anchor.getRight();
                    }
                } else if (length == 1) {
                    root = null;
                }
                length--;
            }
            anchor = anchor.getRight(); // pasos = 3
        }
    }
}

public class DigitSet {
    public void add(int digit) throws NotADigitException, RepeatedDigitExpection { // peor caso
        if (digit < 0 || 9 < digit) // pasos = 3, 1 de ↑
            throw new NotADigitException(digit);

        if (digits[digit] ≠ digit + 1) // pasos = 5 (1. suma, 2. asignacion)
            digits[digit] = digit + 1; // pasos = 7
        else
            throw new RepeatedDigitExpection(digit);
    }
}
```

Tomando todo en el peor caso, tenemos que el cuerpo del ciclo más anidado ejecuta 7 acciones, estas se ejecutaran 10 veces y se le sumaran 22 acciones propias del ciclo, dándonos 92 acciones. Ahora sumamos las 4 acciones que ejecutamos antes del ciclo, ahora tenemos 96, que se multiplicara por los ‘n’ elementos que hay en la lista y les sumamos 4 + 2n acciones propias

del ciclo que encierra a todas estas acciones, con el resultado de 98n + 4 acciones, para terminar, sumamos los 3 pasos fuera de ambos ciclos.

Resultado: $O(98n + 4) = \boldsymbol{O(n)}$

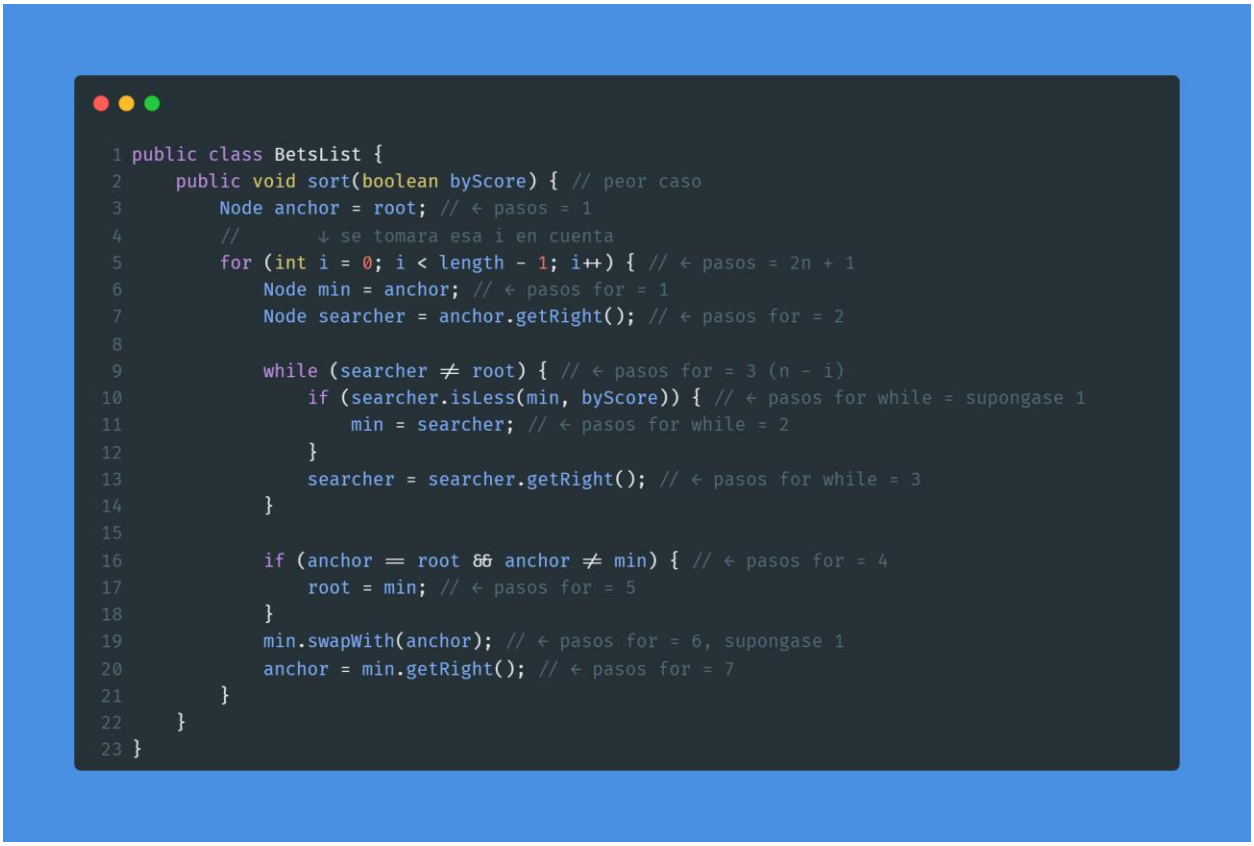
3.

```java
public class BetsList {
    public void setResults(int[] results) { // peor caso
        Node anchor = root; // ← pasos = 1

        for (int n = 0; n < length; n++) { // ← pasos = 2n + 2
            int[] caballos = anchor.getData().getApuestas(); // ← pasos for = 3
            int score = 0, puntos = 10; // ← pasos for = 5

            for (int i = 0; i < 10; i++) { // ← pasos for = 2(10) + 2
                if (caballos[i] == results[i]) { // ← pasos for for = 1
                    score += puntos; // ← pasos for for = 3 (1 mas por la suma)
                }
                puntos--; // ← pasos for for = 5 (1 mas por la resta)
            }

            anchor.getData().setScore(score); // ← pasos for = 7
            anchor = anchor.getRight(); // ← pasos for = 8
        }
    }
}
```

Lo mismo que el anterior:

- Cuerpo ciclo interno: 5
- Ciclo interno con todo: $2\ (10) +\ 2\ +\ 10\ (5) = 72$
- Ciclo externo con el cuerpo: $2n + 2 + n\ (8 + 72) = 82n + 2$
- **Resultado**: $82n + 3$ = **O(n)** <=> lineal

4.



```java
public class BetsList {
    public void sort(boolean byScore) { // peor caso
        Node anchor = root; // ← pasos = 1
        //       ↓ se tomara esa i en cuenta
        for (int i = 0; i < length - 1; i++) { // ← pasos = 2n + 1
            Node min = anchor; // ← pasos for = 1
            Node searcher = anchor.getRight(); // ← pasos for = 2

            while (searcher ≠ root) { // ← pasos for = 3 (n - i)
                if (searcher.isLess(min, byScore)) { // ← pasos for while = supongase 1
                    min = searcher; // ← pasos for while = 2
                }
                searcher = searcher.getRight(); // ← pasos for while = 3
            }

            if (anchor == root && anchor ≠ min) { // ← pasos for = 4
                root = min; // ← pasos for = 5
            }
            min.swapWith(anchor); // ← pasos for = 6, supongase 1
            anchor = min.getRight(); // ← pasos for = 7
        }
    }
}
```

En este varia la cantidad de veces que se ejecuta, para simplificar el procedimiento se tomara solo las partes que tendrán formaran parte de la anotación BigO, el ciclo interno, así que se hará por medio de un ejemplo:

Con una lista de 10 elementos, la primera iteración del ciclo externo va a tener 9 iteraciones del ciclo interno, la siguiente iteración tendrá 8, el siguiente 7, y así sucesivamente.

Si lo ampliamos a n elementos la primera iteración tendrá 9 iteraciones por parte del segundo ciclo, y la ultima terminará con 1 iteración. En una ecuación esto sería:

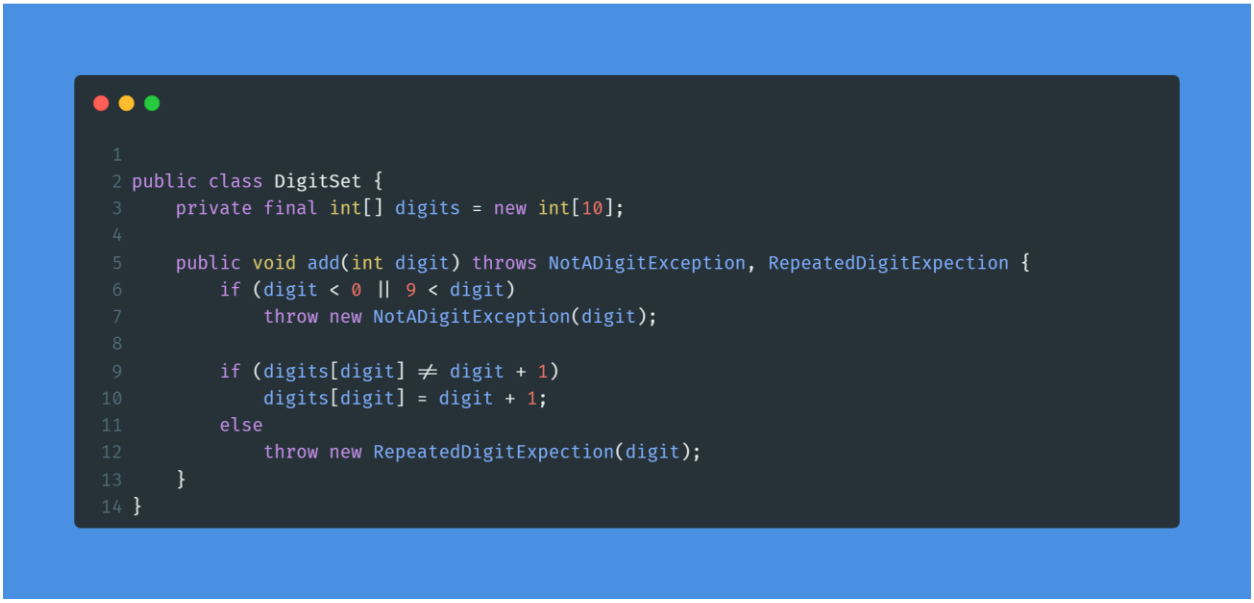
$$(n-1)+(n-2)+(n-3)+\cdots+1=\frac{n(n-1)}{2}$$

Resultado: $\boldsymbol{O(n^2)}$ *dejando el termino dominante*

Otros métodos:

Solo algunos.

Comprobación de caballo repetido:



```java
public class DigitSet {
    private final int[] digits = new int[10];

    public void add(int digit) throws NotADigitException, RepeatedDigitExpection {
        if (digit < 0 || 9 < digit)
            throw new NotADigitException(digit);

        if (digits[digit] != digit + 1)
            digits[digit] = digit + 1;
        else
            throw new RepeatedDigitExpection(digit);
    }
}
```

Este método como se puede ver a simple vista ejecuta unas pocas instrucciones, sin ciclos ni recursión, teniendo una **complejidad lineal o O(n).**

No se incluyeron más métodos por no calcular la complejidad de librerías de java.io o swing, o métodos de los objetos ya implementados como String, Integer, etc.

Argumentación del método de ordenamiento.

Solo se hizo uso de un solo método de ordenamiento, este es el *Selection Sort* u ordenamiento por selección (tal vez), este método como se mostro con anterioridad tiene una complejidad cuadrática, como se solicitó para la práctica, este es su peor caso, mejor caso, caso promedio, etc., por lo cual, puede tener una complejidad mayor que, por ejemplo, el *Insertion Sort*, pero este tiene dos beneficios, por los cuales fue escogido.

El primer beneficio que nos aporta, es que, es simple de programar, si bien el *Insertion Sort* también lo puede ser, el *Selection Sort* se busca un mínimo y se intercambia por el primer elemento de una ‘subcadena’ en lugar de ejecutar una gran cantidad de comparaciones en ambas direcciones de la lista para encontrar el lugar del mínimo encontrado. Pero donde brilla el

Selection Sort, y principal razón por la cual se escogió, es por lo que se acaba de mencionar, las comparaciones.

El segundo beneficio del *Selection Sort* es que realiza menos comparaciones, al mismo tiempo que más copias. Pero nuestras comparaciones se hacen de la siguiente manera:

```
1
2  class Node {
3      public boolean isLess(Node node, boolean byScore) {
4          if (byScore) {
5              return this.getData().getScore() < node.getData().getScore();
6          } else {
7              return 0 < node.getData().getApostador().compareToIgnoreCase(this.getData().getApostador());
8          }
9      }
10 }
```

Si se compara por la puntuación puede ser rápida la comparación, pues llamamos dos valores primitivos y los comparamos, pero si se compara alfabéticamente, tenemos que comparar carácter por carácter, un buen caso solo comparamos 1 carácter, pero, que hay del peor caso, el nombre del apostador no tiene un límite de caracteres y tampoco es único, si dos personas tienen un nombre igual (o una persona apuesta más de una vez) y tiene un nombre largo, esta comparación tendrá una comparación aun mayor (no más que una lineal, pero se puede considerar).

Por lo contrario, el método para intercambiar dos nodos (si sucede como creo que es, y que sería más notorio en C++) se intercambian la dirección en memoria de los objetos nodo que tienen a sus lados, sin más información que esta dirección. Así que, en esta situación **puede** ser buena opción considerar utilizar *Selection Sort* por sobre *Insertion Sort* a pesar de su mayor complejidad en la mayoría de los casos.