Estructuras de datos

Listas para solución a problemas

Resultado de aprendizaje

Walter González Zúñiga

Aldo Barrera González

TID42M

Docente Milton Batres

UTCH | Universidad Tecnológica de Chihuahua

Contenido

Objetivo:

El objetivo de esta tarea es poner en práctica los conocimientos adquiridos a lo largo de la unidad 3, que comprende el uso de listas sencillas y doblemente enlazadas a través de diferentes problemas.

Este reporte busca documentar el proceso para resolver cada uno de los ejercicios.

Material:

- Formato de Reporte de Práctica
- Internet
- Computador
- NeoVim (desgraciadamente)

Desarrollo:

Problema 1

Planteamiento del problema

Se busca desarrollar un programa que implemente los ejercicios 1 y 2, permitiendo la manipulación de los elementos de las listas por parte del usuario. Para lograr este objetivo, se plantea la creación de un menú interactivo que permita la prueba de todos los métodos.

Ejercicio 1 – Lista de ternas

El problema consiste en escribir un método denominado **Generar_Lista_Resumen(L1, L2, R).** Este método debe recibir dos listas **L1** y **L2**, que contienen elementos del mismo tipo, y generar una lista R. Los elementos de **R** deben ser ternas de la forma: (elemento E, número de apariciones de E en L1, número de apariciones de E en L2).

Ejemplo de Problema:

Dadas las listas

L1 = <a, b, a, c, d, b> y L2 = <c, a, b, f, c>,

el método debe producir la lista

R = <(a, 2, 1), (b, 2, 1), (c, 1, 2), (d, 1, 0), (f, 0, 1)>.

Ejercicio 2 – Listas de dígitos y multiplicación

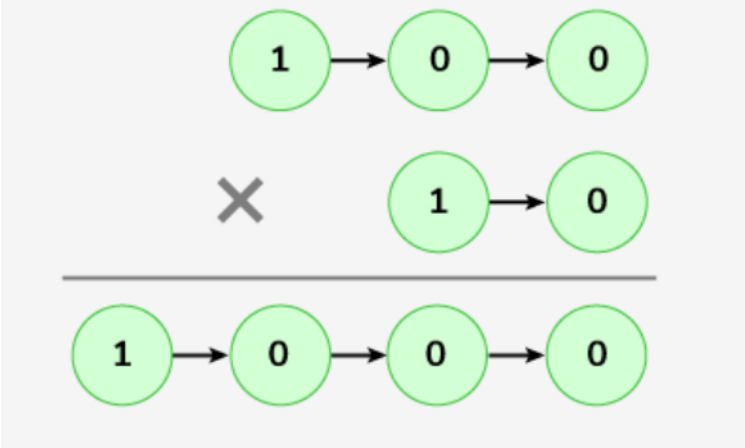
El problema consiste en escribir un método denominado **Generar_multiplicacion(L1, L2, R).** Este método debe recibir dos listas **L1** y **L2**, que contienen elementos del mismo tipo, y generar una lista **R**. Los elementos de **R** deben ser una nueva lista, donde cada número se almacena en un nodo individual.

Ejemplo de Problema:

El número 100 se almacena en L1 como **<1, 0, 0>**

y el número 10 se almacena en L2 como **<1, 0>**,

lo que resulta en la lista R como **<1, 0, 0, 0>.**



Solución de problemática

Ejercicio 1

Primero, se desarrolló la clase Lista Resumen, que se encargará de tratar las listas doblemente enlazadas del ejercicio 1 mediante su único método **generarListaResumen(DLinkedList<T> lista1, DLinkedList<T> lista2).**

Primero, comprueba si las listas recibidas son del mismo tipo de dato:

```
if (!lista1.mismoTipoDato(lista2)) {
    System.out.println("No guardan el mismo tipo de dato");
    return null;
}
```

Creamos una nueva lista dd (doblemente enlazada)

```
DLinkedList<Terna<T>> ternas = new DLinkedList<>();
```

Ahora, comenzamos un ciclo que se ejecuta si lista1 o lista2 tienen elementos dentro.

```
while (lista1.getLength() > 0 || lista2.getLength() > 0) {
    int counter1 = 0;
    int counter2 = 0;
    Nodo<T> elemento = null;
    if (!lista1.isEmpty()) {
        elemento = lista1.getLastElement();
    } else if (!lista2.isEmpty()) {
        elemento = lista2.getLastElement();
    }
```

Reiniciamos los contadores después de cada iteración. Primero verificamos si lista1 contiene elementos. Si tiene elementos, extraemos el último valor. En caso de estar vacía, obtenemos el valor de lista2, ya que sabemos que esta última debe contener algún elemento.

```
    while (lista1.isThere(elemento) ||
lista2.isThere(elemento)) {
        if (lista1.isThere(elemento)) {
            counter1++;
        }
        if (lista2.isThere(elemento)) {
            counter2++;
            lista2.remove(elemento);
        }
        lista1.remove(lista1.getLastElement());
    }
```

Comprobamos si el elemento a consultar se encuentra en ambas listas, en caso de que si, aumentamos el contador y eliminamos el elemento encontrado de ambas listas, si este elemento está repetido, se vuelve a contar y se vuelve a borrar.

```
Terna<T> terna = new Terna<>(elemento.getValue(), counter1,
counter2);
    ternas.add(terna);}
```

Creamos una nueva terna con el elemento, cantidad de apariciones en cada lista y la agregamos a la lista de ternas.

Ejercicio 2

La clase **ListaNumbers** fue desarrollada para trabajar con listas doblemente enlazadas que representan números enteros. Esta clase extiende **DLinkedList<Integer>** e implementa varios métodos para manipular y operar con números de manera eficiente.

Al crear una instancia de ListaNumbers, se inicializa con un número entero:

```
public ListaNumbers(int number1) {
    super();
    this.number = number1;
    convertirNumeroAListaa();
}
```

El constructor llama al método convertirNumeroAListaa, que convierte el número entero en una lista de sus dígitos.

```
private void convertirNumeroAListaa() {
    int digit;
    int operation = number;
    while (operation > 0) {
        digit = operation % 10;
        this.addAt(0, digit);
        length++;
        operation /= 10;
    }
}
```

Se utilizó un algoritmo de extracción de digitos, obteniendo el digito menos significativo y agregandolo a la lista usando el metodo **addAt** con una posición fija de 0.

El metodo addAt está implementado de la siguiente forma:

Metodo de entrada:

```
@Override
public void addAt(long position, T value) {
    if (length == 0) {
        add(value);
    } else if (position >= length) {
        position = length - 1;
    } else {

        addAt(position, 0, value, root);
    }
}
```

Metodo recursivo sobrecargado:

```
public void addAt(long position, long index, T value, Nodo<T>
nodo) {
    if (index == position) {
        Nodo<T> inyected = new Nodo<>(value);
        if (nodo.getRight() == null) {
            tail.setLeft(inyected);
        }
        Nodo<T> nextNode = nodo.getRight();
        inyected.setRight(nodo.getRight());
        nextNode.setLeft(inyected);
        nodo.setRight(inyected);
    } else {
        addAt(position, ++index, value, nodo.getRight());
    }};
```

Asi agregamos cada nodo al **inicio de la lista,** para que conserver su valor decimal, cambiando el apuntador de root y de los nodos previamente añadidos.

El método convertirListaANumero reconstruye el número a partir de la lista de dígitos usando un metodo de inicio y uno recursivo sobrecargado.

Entrada:

```
public int convertirListaANumero() {
    return convertirListaANumero(this.length - 1, tail, 1);

}
```

Recursivo sobrecargado:

```
public int convertirListaANumero(long posicion, Nodo<Integer>
tail, int multiplicador) {
    if (tail == null) {
        return 0;
    }
    if (tail.getLeft() == null) {
        return 0;
    }
    int current = tail.getLeft().getValue() * multiplicador;
    return current + convertirListaANumero(--posicion,
tail.getLeft(), multiplicador * 10);
}
```

Se utiliza recursión para multiplicar cada dígito por su posición decimal correspondiente y suma los resultados para obtener el número original, se aprovecha el doble enlace ya que podemos recorrer la lista desde el valor menos significativo (que siempre será múltiplo de 1) hasta el valor más significativo que desconocemos que múltiplo de 10 puede ser sea.

Ahora que se tiene acceso a los numeros y listas de los numeros mediante metodos, podemos multiplicar dos listas de numeros diferentes.

El método **generar_Multiplicacion** permite multiplicar dos números representados por instancias de **ListaNumbers** que recibe como parametro:

```
public ListaNumbers generar_multiplicacion(ListaNumbers l1, ListaNumbers
l2, ListaNumbers R) {
    int number1 = l1.convertirListaANumero();
    int number2 = l2.convertirListaANumero();

    int result = number1 * number2;

    while (!R.isEmpty()) {
        R.remove(R.getLastElement());
    }


    int digit;
    while (result > 0) {
        digit = result % 10;
        R.addAt(0, digit);
        result /= 10;
    }

    return R;
}
```

Usando las interfaces que creadas, accedemos al estado deseado que necesita la operación.

Este diseño permite manipular números de manera flexible usando listas, lo que es útil para entender cómo las estructuras de datos pueden representar y operar con datos numéricos en formas no convencionales.

Problema 2

Se busca utilizar una lista doblemente enlazada para resolver un problema de manipulación de conjuntos

Ejercicio 3

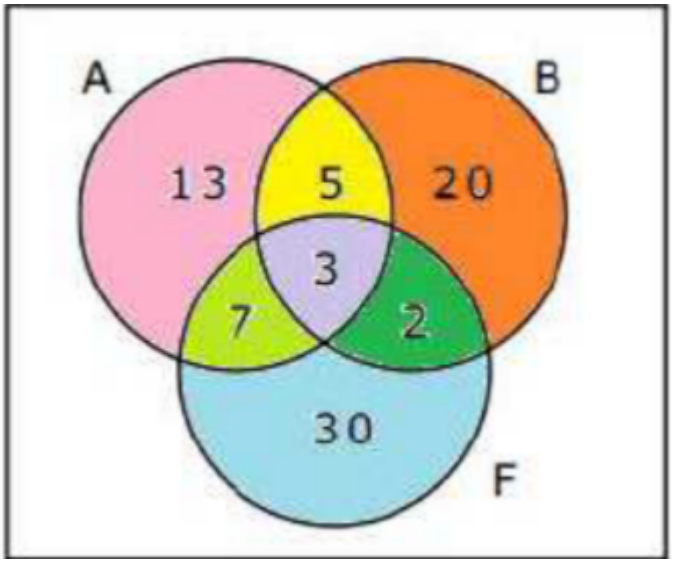
Planteamiento del problema

Se debe realizar un programa en java para resolver el siguiente ejercicio:

En una investigación realizada a un grupo de 100 personas, que estudiaban varios idiomas fueron los siguientes:

Español:28, alemán:30 y francés:42

Español y alemán:8, español y francés:10, alemán y francés:5 y los tres idiomas:3



El programa debe brindar los metodos para poder contestar lo siguiente

a) Cantidad de alumnos que no estudiaban idiomas
b) Cantidad de alumnos que tenían como francés el único idioma estudio.

Solución de problemática

Se codificó una clase Sets utilizando la que se realizó previamente en clase, extendiendo la clase DLinkedList para hacer uso del TDA y adaptar la implementación para hacer uso de los métodos heredados.

```
public class Sets<T> extends DLinkedList<T> implements Iterable<T> {
```

Llamamos al constructor super para crear la lista dd

```
    public Sets() {
        super();
    }
```

El metodo sumaConjunto se encarga de sumar cada elemento de la lista con un while, se hizo de esta manera para practicar la iteración de otra manera que no sea con recursión

```
    public int sumaConjunto() {
        int suma = 0;
        Nodo<T> current = root.getRight();
        while (current != null && current.getValue() != null) {
            suma += (Integer) current.getValue();
            current = current.getRight();
        }
        return suma;
    }

    public void clear() {
        while (!this.isEmpty()) {
            remove(this.getLastElement());
        }
    }
```

El meotodo de union revisa si el set a unir no es null y crea un nuevo set **result** para guardar la union. Se recorre la lista con un while y agrega todos los elementos de **this** al set **result.**

```
    public Sets<T> union(Sets<T> set) {
        if (set == null) {
            throw new IllegalArgumentException("El conjunto es null");
        }

        Sets<T> result = new Sets<>();

        if (root != null) {
            Nodo<T> current = root.getRight();
            while (current != null && current.getValue() != null) {
                result.add(current.getValue());
                current = current.getRight();
```

```
            }
        }
```

Despues, se recorre el segundo **set** y se agrega el valor a **result** si no ese no se encuentra la lista.

```
        if (set.root != null) {
            Nodo<T> current = set.root.getRight();
            while (current != null && current.getValue() != null) {
                if (!result.isThere(current.getValue())) {
                    result.add(current.getValue());
                }
                current = current.getRight();
            }
        }

        return result;
    }
```

El metodo de diferencia agrega todos los elementos de **this** a **newSet** y luego los remueve si estos se encuentran en el **set** dado.

```
    public Sets<T> difference(Sets<T> set) {
        if (set == null) {
            throw new IllegalArgumentException("El conjunto es null");
        }

        Sets<T> newSet = new Sets<>();

        for (T element : this) {
            newSet.add(element);
        }

        for (T element : set) {
            newSet.remove(element);
        }

        return newSet;
    }
```

Posteriormente, en el metodo main, se implementó la creación de listas con los valores.

`Sets<Integer> frances = new Sets<>();`
 	
`frances.add(30);`
`frances.add(7);`
`frances.add(3);`
`frances.add(2);`
	`Sets<Integer> aleman = new Sets<>();`
	
	`aleman.add(20);`
	`aleman.add(5);`
	`aleman.add(3);`
	`aleman.add(2);`
		`Sets<Integer> espanol = new Sets<>();`
		
		`espanol.add(13);`
		`espanol.add(5);`
		`espanol.add(3);`
		`espanol.add(7);`

Se imprimen los elementos de cada conjunto

```
System.out.println("\nConjuntos iniciales:");
System.out.println("Set frances: " + frances);
System.out.println("Set aleman: " + aleman);
System.out.println("Set espanol: " + espanol);
```

Se crea una lista espanolAleman y se imprime

```
Sets<Integer> espanolAleman = espanol.union(aleman);
System.out.println("Unión de espanol y aleman: " +
espanolAleman);
```

Restamos al conjunto francés la union espanolAleman para obtener la diferencia y obtener la cantidad de alumnos que solo estudian francés

```
Sets<Integer> soloFrances =frances.difference(espanolAleman);
System.out.println("Alunmnos que solo están en francés: " +
soloFrances);
```

Se crea una lista con todos los valores, uniendo francés a la union de espanolAleman, despues se ejecuta el metodo sumaConjunto sobre la lista de todos los valores y se resta a 100 para obtener la cantidad total de alumnos que no estudian idiomas.

```
Sets <Integer> todos = espanolAleman.union(frances);
int cantAlumnos = todos.sumaConjunto();
System.out.println("Cantidad de alumnos que no estudian
idiomas: " + (100 - cantAlumnos));
```

Implementación en main y resultados

La clase Main sirve como interfaz de usuario para interactuar con las funcionalidades de las clases ListaNumbers, ListaResumen y Sets. A través de un menú en la consola, los usuarios pueden realizar operaciones con números y listas de manera interactiva.

Menú Principal

El programa inicia mostrando un menú principal que ofrece tres opciones:

1. Operaciones con ListaNumbers: Permite realizar diversas operaciones con números representados como listas.

2. Operaciones con ListaResumen: Facilita la comparación y el resumen de dos listas de caracteres.

3. Resultado de Sets: Ejecuta el resultado del ejercicio 3

4. Salir

```
public static void main(String[] args) {
    BufferedReader reader = new BufferedReader(new
InputStreamReader(System.in));
    boolean exit = false;

    while (!exit) {
        System.out.println("Seleccione una opción:");
        System.out.println("1. Operaciones con Lista de ternas");
        System.out.println("2. Operaciones con Listas de numeros");
        System.out.println("3. Operaciones con Sets");
        System.out.println("4. Salir");

        try {
            int choice = Integer.parseInt(reader.readLine());
            switch (choice) {
                case 1:
                    menuListaResumen(reader);
                    break;
                case 2:
                    menuListaNumbers(reader);
                    break;
                case 3:
                    resolverProblemasSets();
                    break;
                case 4:
                    exit = true;
                    System.out.println("Au revoir");
                    break;

                default:
                    System.out.println("Opción no válida. Intente de
nuevo.");
```

Operaciones con ListaNumbers

Dentro de menuListaNumbers, el usuario ingresa los valores de los numeros a tratar y se crean dos listas con los valores

```
ListaNumbers lnum1 = new ListaNumbers(num1);
ListaNumbers lnum2 = new ListaNumbers(num2);
```

```
Ingrese el primer número:
456
Ingrese el segundo número:
22234
```

- Convertir números a listas: Muestra la representación en lista de los números ingresados utilizando el metodo toString de ListaNumbers

```
Seleccione una operación:
1. Convertir número a lista
2. Convertir lista a número
3. Multiplicar números
4. Volver al menú principal
1
Lista del primer número: < 4 , 5 , 6 >
Lista del segundo número: < 2 , 2 , 2 , 3 , 4 >
```

- Convertir listas a números: Reconstruye y muestra el número original a partir de su lista de dígitos utilizando el metodo de convertirListaaNumero sobre cada objeto ListaNumbers

```
Seleccione una operación:
1. Convertir número a lista
2. Convertir lista a número
3. Multiplicar números
4. Volver al menú principal
2
Número del primer lista: 456
Número del segundo lista: 22234
```

- Multiplicar números: Realiza la multiplicación de dos números representados como listas y muestra el resultado; se crea una nueva lista utilizando el método multiplicar sobre uno de las listas y recibiendo otra como argumento.

```
ListaNumbers R = new ListaNumbers(0);
R = lnum1.generar_multiplicacion(lnum1, lnum2, R);
System.out.println("El producto de " + num1 + " y " + num2 + " es: " +
R.convertirListaANumero());
System.out.println("Lista del producto: " + R);
```

```
Seleccione una operación:
1. Convertir número a lista
2. Convertir lista a número
3. Multiplicar números
4. Volver al menú principal
3
El producto de 456 y 22234 es: 10138704
Lista del producto: < 1 , 0 , 1 , 3 , 8 , 7 , 0 , 4 >
```

Operaciones con ListaResumen

En menuListaResumen, el programa crea dos listas vacias y el usuario ingresa la longitud de los arreglos y los llena con letras realizando lo siguiente:

- **Ingresar listas de caracteres:** Se solicita al usuario que ingrese elementos para dos listas.

```
Ingrese el número de elementos para la primera lista:
5
Ingrese los elementos de la primera lista (un solo carácter por línea):
a
s
d
f
g
Ingrese el número de elementos para la segunda lista:
3
Ingrese los elementos de la segunda lista (un solo carácter por línea):
f
f
r
```

- **Generar y mostrar resumen:** Utiliza un nuevo objeto ListaResumen para generar un resumen de las ocurrencias de cada carácter en ambas listas utilizando el metodo generarListaResumen y recibiendo las dos listas previas como argumentos.

El resultado de ListaResumen genera una DlinkedList de ternas, que son recibidas por un nuevo DlinkedList<Terna <Character>>

```
ListaResumen<Character> listaResumen = new ListaResumen<>();
DLinkedList<Terna<Character>> resumen =
listaResumen.generarListaResumen(lista1, lista2);
```

El resultado se muestra utilizando un iterador para recorrer la lista de ternas.

```
Resumen de las listas:
<g,1,0>
<f,1,2>
<s,1,0>
<a,1,0>
<r,0,1>
```

Resultado de ejercicio de conjuntos

Al seleccionar la opción 3, se ejecuta el método resolverProblemaSet().

```
Conjuntos iniciales:
Set frances: { 30, 7, 3, 2 }
Set aleman: { 20, 5, 3, 2 }
Set espanol: { 13, 5, 3, 7 }
Unión de espanol y aleman: { 13, 5, 3, 7, 20, 2 }
Alunmnos que solo están en francés: { 30 }
Cantidad de alumnos que no estudian idiomas: 20
```

Obteniendo que hay **30** alumnos que solo estudian francés y **20** alumnos que no estudian idiomas

Orden te tiempos de ejecución

Ejercicio 1 – Ternas

El bucle while exterior se ejecuta hasta que ambas listas estén vacías resultando en **O(n+m)** iteraciones donde n y m son las longitudes de lista1 y lista2.

Para cada iteración:

La operación isThere() es O(n) u O(m)

La operación remove() es O(n) u O(m)

Estas operaciones se realizan múltiples veces en el bucle while interno

Por lo tanto, la complejidad total es **$O((n+m)^2)$** porque:

Para cada elemento (n+m), podríamos necesitar recorrer ambas listas y en el peor caso, necesitamos comparar cada elemento contra todos los demás elementos.

Esta complejidad es debido a la implementación, la cual asume recibir una lista no ordenada y detiene la valoración de elementos cuando se encuentra con un valor nulo (indicando que llegó al final de la lista).

Si se tuviera un método de ordenamiento podríamos implementar un método que valora caracteres y deja de buscar en la lista cuando el valor evaluado ya no es el mismo al que se desea contar.

Así que este método, en lugar de tener una complejidad de **$O((n+m)^2)$** podría reducirse hasta **O(n+m).**

Ejercicio 2 – Dígitos

Se tienen dos tiempos de ejecución diferentes:

Conversión de numero a lista y lista a número.

Ambas conversiones tienen un tiempo de ejecución de **O(n),** donde n es la cantidad de elementos a convertir, ya sea dígitos o elementos de una lista.

Multiplicación de listas

La complejidad es O(n + m + d) debido a:

1. Primera conversión: O(n) al recorrer la primera lista una vez.

2. Segunda conversión: O(m) al recorrer la segunda lista una vez.

3. Multiplicación: O(1) por ser una operación básica de enteros.

4. Conversión del resultado: O(d) al crear nuevos ListaNumbers con hasta log(nm) dígitos.

¿El tiempo de ejecución cambiaría al tener una lista simple enlazada?

Si y no, actualmente nuestra lista se lee en tiempo O(N) debido a que accedemos desde **tail**.

Si se tuviera una lista simple enlazada tendríamos que cambiar el acercamiento y la implementación ya que no se tendría acceso a los nodos previos y solo se puede leer hacia delante.

Tendríamos que hacer una de dos cosas:

Usar un stack para almacenar la lista, que por la naturaleza del stack podríamos acceder a nuestra lista desde el ultimo digito hasta el primero. Este acercamiento leería dos veces los valores de la lista al guardarlos en el stack y al sacarlos, obteniendo

O atravesar la lista desde el inicio y calcular las potencias de 10 de forma diferente, elevando 10 a la potencia length – 1.

Conclusiones:

Este resultado de aprendizaje puso a prueba todo lo aprendido en la unidad con problemas sumamente interesantes.

Al extender la clase DLinkedList para cada clase fue un gran apoyo para utilizar esta estructura de datos e implementando los métodos necesarios para cada ejercicio sin tener que recrear la estructura para todos o crear un monolito con todos los métodos necesarios.

El ejercicio de multiplicación de listas fue el más interesante, y queremos volver a realizar el problema ahora sin utilizar la conversión a entero y realizar la implementación como si fuera a mano, que nos hubiera encantado incluir, pero la complejidad era demandante. Por lo que se buscó cumplir con los requerimientos explícitamente pedidos y cumplirlos.

Además, recreación de la clase conjuntos heredando de DLinkedList fue el ejemplo más claro de como los TDA deben de crearse con la finalidad de ser interfaces para realizar operaciones más complejas.

Finalmente, el análisis para determinar el time complexity de los diferentes algoritmos nos llevó a una investigación interesante para entender cómo funcionan los algoritmos y la forma en la complejidad aumenta acorde a los inputs y operaciones.

Sin duda el trabajo más demandante hasta ahora, (como lo pongo en cada reporte), pero nos gusta saber que cada reto es más complicado que el anterior pero siempre son realizables.

Referencias

Mentor, P. (2023, October 26). Mastering Time Complexity - perspective mentor - medium. Medium. https://medium.com/@perspectivementor/mastering-time-complexities-bb09e40b11e9

Categorizar eficiencia de tiempo de ejecución (artículo) | Khan Academy. (n.d.). Khan Academy. https://es.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/evaluating-algorithms/a/comparing-run-time-efficiency

Admin. (2023, January 6). Set Theory (Basics, Definitions, Types of sets, Symbols & Examples). BYJUS. https://byjus.com/maths/basics-set-theory/