Colecciones en Java

Autor : Jonathan Carrero

Contenido

1 Introducción 3

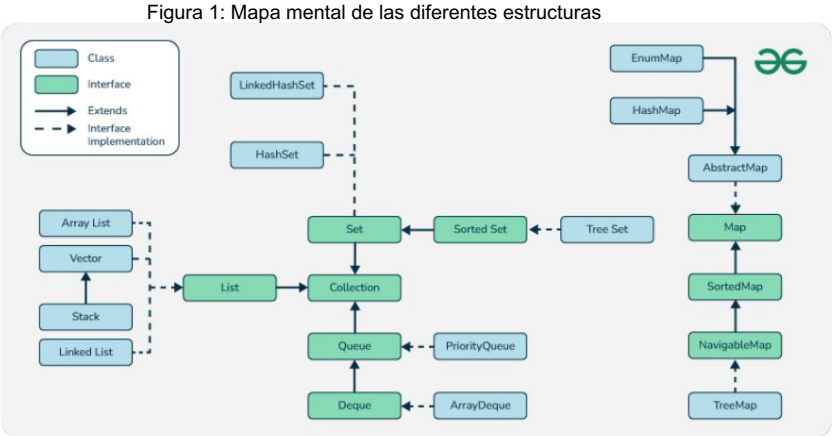
2 Estructuras de datos 5

2.1 Matrices 5
2.2 Listas 6
2.2.1 Lista de arreglos 6
2.2.2 Lista enlazada. 7
2.3 Pilas 8
2.4 Colas. 11
2.5 Árboles 13
2.6 Árboles binarios de búsqueda (BST). 13
2.7 Tablas hash. . . . . dieciséis

1 Introducción

Las colecciones permiten gestionar grupos de datos: almacenarlos, recuperarlos y manipularlos. Son un conjunto de interfaces, clases concretas y abstractas que definen tipos de datos abstractos en Java que implementan colecciones de elementos. Por ejemplo, lista, pila, cola, conjunto, mapa.

Figura 1: Mapa mental de las diferentes estructuras



Imagínese que es un bibliotecario responsable de gestionar una gran cantidad de libros.
Necesita un sistema para realizar un seguimiento eficiente de todos estos libros. En Java, el concepto de colecciones cumple una función similar. Las colecciones son como el equivalente digital de su sistema de organización en la biblioteca. Le permiten agrupar y administrar objetos de una manera estructurada y flexible.

Tarea: observando la figura, ¿existen instancias de objetos que no se crearían? Da algunos ejemplos.

Como puedes imaginar, las colecciones pueden tener un orden determinado. Este orden puede ser lineal o no lineal, por lo que existen diferentes estructuras para que podamos utilizar cada una de ellas. ellos cuando más los necesitamos.

Además, cuando se trabaja con aplicaciones Java del mundo real, a menudo se trabaja con grandes cantidades de datos, que es necesario organizar y acceder a ellos de forma eficiente.
Sin colecciones, tendría que escribir una gran cantidad de código personalizado para administrar estructuras de datos, lo cual sería difícil de mantener y de escalar.

Las colecciones de Java proporcionan una forma estandarizada de gestionar estos datos. Cuentan con métodos integrados que simplifican tareas complejas como la ordenación, la búsqueda y el filtrado. De hecho, las colecciones son dinámicas, por lo que pueden crecer o reducirse según sea necesario, lo que supone una ventaja significativa con respecto a las matrices, que tienen un tamaño fijo.

Figura 2: Estructuras de datos

Data Structures	Description
Array	Ordered collection of elements of the same data type.
Linked List	Dynamic data structure where elements are connected through nodes.
Stack	Follows Last In, First Out (LIFO) principle.
Queue	Follows First In, First Out (FIFO) principle.
Tree	Hierarchical data structures with a root node and branches.
Graph	Consists of nodes (vertices) and edges representing connections.

Tarea: busca en Internet la representación gráfica de las estructuras de datos que hemos visto, para que puedas hacerte una idea de cómo se maneja la información y estructurado.

Figura 3: Algoritmos

Algorithms	Description	Example
Sorting Algorithms	Techniques to arrange elements in a specific order.	Quick Sort, Merge Sort
Searching Algorithms	Techniques to find an element's presence and locate its position.	Binary Search, Linear Search
Graph Algorithms	Algorithms used to traverse or search through graphs.	Breadth-First Search, Depth-First Search
Dynamic Programming	Optimizing problems with overlapping subproblems.	Fibonacci Series
Divide and Conquer	Breaks a problem into smaller subproblems and solves them.	Binary Search
Greedy Algorithms	Make locally optimal choices at each stage.	Dijkstra's Algorithm

Tarea: busca en internet una representación en vídeo de cómo funcionan los algoritmos que hemos visto. Observa las diferencias en cada uno de ellos y date cuenta de que todos tienen una finalidad y deben utilizarse para un caso concreto.

2 Estructuras de datos

Las estructuras de datos se utilizan para organizar y almacenar datos en una computadora. Nos ayudan a mantener la información en orden y facilitan su uso. Piense en las estructuras de datos como si fueran diferentes tipos de cajas o contenedores para sus cosas.

¿Qué son las estructuras de datos?

- Las estructuras de datos son formatos especiales para organizar y almacenar datos.
- Son como contenedores que contienen información de una manera específica.
- Los ejemplos incluyen listas, pilas y árboles.

¿Por qué son importantes?

- Ayuda a administrar y organizar grandes cantidades de datos.
- Facilitar y agilizar la búsqueda y el uso de la información.
- Permitir que las computadoras trabajen de manera más eficiente
- Esencial para escribir programas informáticos buenos y rápidos.

Comenzaremos con las más simples y pasaremos a estructuras más complejas. Para cada una estructura de datos, también veremos sus implementaciones y complejidad temporal.

2.1 Matrices

Los arrays son una de las estructuras de datos más simples y comunes en Java. Son como una fila de cajas donde puedes almacenar cosas. Un array es una colección de elementos del mismo tipo, almacena elementos en orden, uno tras otro, cada elemento tiene un número (llamado índice) para encontrarlo fácilmente y tienen un tamaño fijo que no cambia después de su creación.

¿Qué hace el siguiente código?

```
int [] números = nuevo int [5];

números [0] = 10;
números [1] = 20;
números [2] = 30;
números [3] = 40;
números [4] = 50;

System . out . println ("El tercer número es: " + números [2]) ;

para ( int i = 0; i < números . longitud ; i ++) {
        Sistema . out . println ( " Número números " + (yo + 1) + " es: " +
            [ i ]) ;
}
```

Las matrices son útiles cuando sabes cuántos elementos necesitas almacenar y cuándo
Quiere acceder a los elementos rápidamente por su posición.

Complejidad de tiempo para operaciones de matriz:

- Acceso: O(1)
- Buscar: O(n)
- Inserción: O(n)
- Eliminación: O(n)

2.2 Listas

Las listas son como matrices mejoradas. Pueden crecer o reducirse según sea necesario. Java tiene
dos tipos principales de listas: ArrayList y LinkedList.

2.2.1 Lista de matrices

Una ArrayList en Java es una estructura de datos redimensionable similar a una matriz que le permite
para almacenar una lista de elementos. A diferencia de las matrices normales, ArrayList puede almacenar automáticamente
ajusta su tamaño cuando agregas o eliminas elementos, haciéndolo más flexible.
Proporciona métodos para agregar, eliminar y acceder fácilmente a elementos, y mantiene
el orden en que se insertan los elementos.

Características de un ArrayList:

- Similar a una matriz, pero puede cambiar de tamaño.

- Bueno para almacenar y acceder a artículos rápidamente.
- No es bueno para agregar o quitar elementos en el medio.

¿Qué hace el siguiente código?

```
importar java .util .ArrayList ;

clase pública ArrayListExample {
        public static void principal (cadena [] argumentos) {
                ArrayList < String > frutas = new ArrayList < String >() ;
                frutas . añadir ("Manzana ") ;
                frutas. añadir ("Plátano ") ;
                frutas. añadir ("Cereza ") ;

                Sistema . out . println ( " Frutas : " + frutas );
                Sistema . out . println ( " Segunda fruta : " + frutas . get (1) ) ;
                frutas . conjunto (0    ,    " Albaricoque ") ;
                frutas. quitar ("Cereza ");
                Sistema . out . println ( " Frutas actualizadas :                    " + frutas );
        }
}
```

Complejidad de tiempo para operaciones de ArrayList:

- Acceso: O(1)
- Buscar: O(n)
- Inserción (al final): O(1)
- Inserción (en índice específico): O(n)
- Eliminación (al final): O(1)
- Eliminación (en índice específico): O(n)

2.2.2 Lista enlazada

Una lista enlazada en Java es una estructura de datos que consta de una secuencia de elementos, donde cada elemento está vinculado al siguiente y al anterior, formando una cadena.
A diferencia de una ArrayList, que almacena elementos en un bloque contiguo de memoria,
Una lista enlazada almacena cada elemento en un nodo separado que contiene el elemento en sí y las referencias a los nodos anterior y siguiente. Esto hace que LinkedList
Eficiente para inserciones y eliminaciones en cualquier posición, pero más lento para acceder. elementos por índice.

Características de una lista enlazada:

- Compuesto por elementos separados unidos entre sí
- Es útil para agregar o eliminar elementos en cualquier parte de la lista.
- No es tan bueno para acceder a elementos por posición.

¿Qué hace el siguiente código?

```
30 importar java . util . LinkedList ;
31
32 clase pública LinkedListExample {
33         public static void principal (cadena [] argumentos) {
34                 ListaEnlazada < Entero > números = nueva ListaEnlazada < Entero
                        >() ;
35                 números . suma (10) ;
36                 números . suma (20) ;
37                 números . suma (30) ;
38
39                 Sistema . out . println ( " Números : números .          " + números );
40                 addFirst (5) ;
41                 números .addLast(40);
42                 Sistema . out . println ( " Números actualizados :          " + números );
43                 numeros . removeFirst();
44                 Sistema . out . println ( " Números finales :          " + números );
45         }
46 }
```

Complejidad de tiempo para operaciones de LinkedList:

- Acceso: O(1)
- Buscar: O(n)
- Inserción (al principio/final): O(1)
- Inserción (en índice específico): O(n)
- Eliminación (al principio/final): O(1)
- Eliminación (en índice específico): O(n)

2.3 Pilas

Una pila en Java es una estructura de datos que sigue el principio de último en entrar, primero en salir (LIFO).
principio, es decir, el último elemento añadido es el primero que se elimina.

Puedes pensar en ello como una pila de platos: agregas nuevos platos encima y cuando
Si necesitas una, toma la placa superior. En Java, una pila permite operaciones como
Push (para agregar elementos), pop (para eliminar el elemento superior) y peek (para mirar).
en el elemento superior sin quitarlo). Se utiliza comúnmente para escenarios como
retroceso, mecanismos de deshacer y análisis de expresiones. Por lo tanto, son útiles
para tareas que necesitan revertir el orden o realizar un seguimiento de lo que sucedió por última vez.

Tarea: sin mirar el código, crear una clase Stack<T> que simule una
apilar utilizando un ArrayList<T>.

Podemos crear una pila simple usando una lista de arreglos. A continuación, se muestra un ejemplo:

```
47 importar java . util . ArrayList ;
48
49 clase pública Pila <T > {
50         ArrayList privado <T > pila;
51
52         pila pública () {
53                 pila = nueva ArrayList <T >() ;
54         }
55
56         //Agrega un elemento a la parte superior de la pila
57         public void push (elemento T) {
58                 pila . agregar ( elemento );
59         }
60
61         //Elimina y devuelve el elemento superior de la pila
62         público T pop() {
63                 si ( estáVacío () ) {                                                    "
64                         lanzar nueva IllegalStateException ("La pila está vacía")
                                ) ;
65                 }
66                 retorna pila . eliminar ( pila . tamaño () - 1) ;
67         }
68
69         //Mira el elemento superior sin quitarlo
70         público T peek() {
71                 si ( estáVacío () ) {                                                    "
72                         lanzar nueva IllegalStateException ("La pila está vacía")
                                ) ;
73                 }
74                 retorna pila . get ( pila . tamaño () - 1) ;
75         }
```

```
76
77          // Comprueba si la pila está vacía
78          booleano público está vacío () {
79                  devuelve pila .isEmpty();
80          }
81
82          // Obtener el tamaño de la pila
83          público int tamaño () {
84                  retorna pila . tamaño();
85          }
86 }
```

Aquí te explicamos cómo utilizar esta pila:

```
87 clase pública StackExample {
88          public static void principal (cadena [] argumentos) {
89                  Pila < String > bookStack = nueva Pila < String >() ;
90
91                  //Añadir libros a la pila
92                  bookStack . push (" Libro 1" ) ;
93                  bookStack . push ("Libro 2" ) ;
94                  bookStack . push ("Libro 3" ) ;
95
96                  Sistema . out . println ( " Tamaño de pila : " + bookStack . tamaño ()
                            ) ;
97                  Sistema . out . println ("Libro superior: " + bookStack .peek () );
98
99                  //Quitar el libro superior
100                 Cadena removedBook = bookStack . pop ();
101                 Sistema . out . println ( " Libro eliminado : " + libroeliminado );
102
103                 Sistema . out . println ("Nuevo tamaño de pila: tamaño () ) ; " + pila de libros .
104                 Sistema . out . println ("Nuevo libro superior: " + bookStack . vistazo
                            () ) ;
105         }
106 }
```

Complejidad temporal de las operaciones de pila:

- Empujar: O(1)
- Pop: O(1)

- Vistazo: O(1)
- Buscar: O(n)

2.4 Colas

Una cola es otra estructura de datos importante. Funciona como una fila de personas.
Esperando algo: la primera persona que se une a la fila es la primera en salir.
Sigue el principio FIFO (First In First Out), es decir, el primer elemento
El primero que se agrega es el que se elimina. Es como una fila de personas esperando ser atendidas:
La primera persona en la fila es atendida primero. En Java, una cola proporciona operaciones como
oferta (para agregar elementos), encuesta (para eliminar y devolver el elemento frontal) y vistazo
(para ver el elemento frontal sin quitarlo). También es útil para administrar
tareas en orden, como trabajos de impresión o solicitudes de servicio al cliente.

Tarea: sin mirar el código, crear una clase Queue<T> que simule una
cola que utiliza una LinkedList<T>.

Podemos crear una cola sencilla utilizando una LinkedList. A continuación, se muestra un ejemplo:

```
107 importar java . util . LinkedList ;
108
109 clase pública Cola <T > {
110         cola privada LinkedList <T >;
111
112         cola pública () {
113                 cola = nueva LinkedList <T >() ;
114         }
115
116         //Añadir un elemento al final de la cola
117         public void enqueue(T elemento) {
118                 cola .addLast(elemento);
119         }
120
121         //Eliminar y devolver el elemento frontal de la cola
122         público T desencolar() {
123                 si ( estáVacío () ) {
124                         lanzar nueva IllegalStateException ("La cola está vacía")
                                ) ;
125                 }
126                 devolver cola .removeFirst();
127         }
128
```

```
129         //Mira el elemento frontal sin quitarlo
130         público T peek() {
131             si ( estáVacío () ) {
132                 lanzar nueva IllegalStateException ("La cola está vacía")
                        ) ;
133             }
134             devolver cola .getFirst();
135         }
136
137         //Comprueba si la cola está vacía
138         booleano público está vacío () {
139             devolver cola .isEmpty();
140         }
141
142         // Obtener el tamaño de la cola
143         público int tamaño () {
144             devolver cola .size();
145         }
146 }
```

Aquí te explicamos cómo utilizar esta cola:

```
147 clase pública QueueExample {
148         public static void principal (cadena [] argumentos) {
149             Cola < String > customerQueue = nueva Cola < String >() ;
150
151             //Añadir clientes a la cola
152             customerQueue . enqueue (" Cliente 1" ) ;
153             customerQueue . enqueue (" Cliente 2" ) ;
154             customerQueue . enqueue (" Cliente 3" ) ;
155
156             Sistema . out . println ( " Tamaño de la cola : " + cola de clientes .
                    tamaño () ) ;
157             Sistema . out . println ( " Cliente frontal : customerQueue . " +
                    peek () ) ;
158
159             // Eliminar el cliente frontal
160             CadenaservedCustomer = customerQueue .dequeue ();
161             Sistema . out . println ( " Cliente atendido : " +
                    servidoCliente );
162
163             Sistema . out . println ("Nuevo tamaño de cola: " +
```

```
                customerQueue . size () ) ; System .
164             out . println ("Nuevo cliente frontal : customerQueue . peek () ) ;      "   +

165         }
166 }
```

Complejidad temporal de las operaciones de cola:

- Poner en cola: O(1)
- Quitar de la cola: O(1)
- Vistazo: O(1)
- Buscar: O(n)

2.5 Árboles

Un árbol en Java es una estructura de datos jerárquica en la que cada elemento, llamado nodo, está conectado a otros en una relación padre-hijo. El nodo superior se llama raíz y cada nodo puede tener cero o más nodos secundarios. Los árboles son útiles para representar datos jerárquicos, como sistemas de archivos, estructuras organizativas o procesos de toma de decisiones. En Java, existen tipos específicos de árboles, como los árboles binarios, en los que cada nodo tiene como máximo dos hijos, y árboles equilibrados como TreeMap o TreeSet, que mantienen los elementos en orden ordenado para realizar búsquedas e inserciones rápidas.
Los árboles se utilizan comúnmente en búsquedas, ordenamientos y representaciones jerárquicas.

2.6 Árboles binarios de búsqueda (BST)

Es un tipo de árbol binario en el que cada nodo tiene como máximo dos hijos y el árbol mantiene un orden específico. Para cualquier nodo dado, el hijo izquierdo contiene valores menores que el valor del nodo y el hijo derecho contiene valores mayores que el valor del nodo. Esta propiedad hace que las operaciones de búsqueda, inserción y eliminación sean eficientes, ya que permite delimitar rápidamente dónde se debe colocar o encontrar un valor. Los árboles binarios de búsqueda se utilizan comúnmente para buscar, ordenar y mantener un conjunto dinámico de datos ordenados.

Por cierto, utilizaremos la clase Comparable. Esta interfaz impone un ordenamiento total a los objetos de cada clase que la implementa. Este ordenamiento se denomina ordenamiento natural de la clase y el método compareTo de la clase se denomina ordenamiento natural de la clase.
como su método de comparación natural.

Ahora, implementemos nuestro primer árbol. Para simplificarlo, crearemos un árbol donde

Cada nodo almacena un valor int. Tenga en cuenta que, aunque es un árbol BST, este árbol
No se equilibra por sí mismo. Los árboles que tienen esta propiedad son los árboles AVL, que son
fuera del alcance de este tema debido a su complejidad.

Tarea: sigue las indicaciones del profesor para desarrollar tu primera implementación del árbol BST. Es muy importante
entender cada uno de los pasos, luego habrá
una actividad de implementación del BST.

A continuación se muestra una implementación simple de un árbol de búsqueda binario utilizando algunas clases Java.
Tenga en cuenta que los datos pueden ser genéricos como <T>:

```
167 clase pública BinarySearchTree < T extiende Comparable <T > > {
168         TreeNode privado <T > raíz;
169
170         public void insert(T datos) {
171                 raíz = insertRec ( raíz , datos ) ;
172         }
173
174         privado TreeNode <T > insertRec ( TreeNode <T > raíz , T datos )
                {
175                 si ( raíz == nulo ) {
176                         raíz = nuevo TreeNode <T >(datos);
177                         devolver raíz;
178                 }
179
180                 si (datos.compareTo(raíz.datos) < 0)
181                         raíz . izquierda = insertRec ( raíz . izquierda , datos ) ;
182                 de lo contrario si (datos.compareTo(raíz.datos) > 0)
183                         raíz . derecha = insertRec ( raíz . derecha , datos ) ;
184
185                 devolver raíz;
186         }
187
188         búsqueda booleana pública (datos T) {
189                 devolver búsquedaRec (raíz , datos ) ;
190         }
191
192         booleano privado searchRec ( TreeNode <T > raíz , T datos ) {
193                 si ( raíz == null || raíz . datos . es igual a ( datos ) )
194                         devuelve raíz != null ;
195
196                 si (datos.compareTo(raíz.datos) < 0)
197                         devolver búsquedaRec ( raíz . izquierda , datos ) ;
```

```
198
199                 devolver búsquedaRec ( raíz . derecha ,          datos ) ;
200         }
201 }
```

A continuación se explica cómo utilizar este árbol de búsqueda binaria:

```
202 clase pública BSTExample {
203         public static void principal (cadena [] argumentos) {
204                 ÁrbolBinarioDeBúsqueda<Entero>bst = nuevo ÁrbolBinarioDeBúsqueda
                        < >() ;
205
206                 // Insertar algunos números
207                 bst . insertar (50) ;
208                 bst . insertar (30) ;
209                 bst . insertar (70) ;
210                 bst . insertar (20) ;
211                 bst . insertar (40) ;
212
213                 // Buscar números
214                 Sistema . out . println ("¿Hay 20 en el árbol? buscar (20) ) ;          "   +bst.

215                 Sistema . out . println ("¿Está 60 en el árbol?                          "   +bst.
                        buscar (60) ) ;
216         }
217 }
```

Complejidad temporal de las operaciones BST:

Figura 4: Operaciones de complejidad en BST

Operations	Best case time complexity	Average case time comp.	Worst case time comp.
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Tarea: esta tarea podría simular un ejercicio de examen. Sin consultar el código proporcionado por el profesor, implementa un BST que almacena un int en cada nodo con todos los métodos vistos (excepto eliminar cuando el nodo tiene dos hijos).

Ya conocemos Java, por lo que la dificultad en este ejercicio no es implementar, sino
Pensar en lo que quieres implementar.

2.7 Tablas hash

Una tabla hash en Java es una estructura de datos que almacena pares clave-valor, lo que permite recuperación rápida de valores en función de sus claves. Se utilizan ampliamente en muchas aplicaciones donde el acceso rápido a los datos es crucial. Sin embargo, si varias claves se asignan a la mismo índice (una colisión), técnicas adicionales como encadenamiento o direccionamiento abierto Se utilizan para resolver el conflicto. Las tablas hash se implementan comúnmente en Java. a través de clases como HashMap o Hashtable.

He aquí un ejemplo:

```
importar java . util . HashMap ;

clase pública HashMapExample {
        public static void principal (cadena [] argumentos) {
                // Crear un nuevo HashMap
                HashMap < String , Integer > edades = new HashMap < >() ;

                // Agregar pares clave-valor al mapa
                edades . poner ("Alicia ", 25);
                edades . poner ("Bob", 30);
                edades . poner (" Carol ", 35);

                // Obtener un valor del mapa
                int bobAge = edades . get (" Bob" ) ;
                Sistema . out . println ("La edad de Bob:                    " + bobEdad);

                // Verificar si existe una clave
                si ( edades . contieneClave (" David ") ) {
                        Sistema . out . println ( " Se conoce la edad de David ") ;
                } demás {
                        Sistema . out . println ( " La edad de David es desconocida ");
                }

                // Actualizar un valor
                edades . poner ("Alicia ", 26);

                // Eliminar un par clave-valor
                edades . eliminar (" Carol ");

```

```
247                 // Imprimir todas las entradas
248                 Sistema . out . println ("\ nTodas las entradas :") ; for ( String
249                 nombre : edades . keySet () ) { Sistema . out . println
250                         ( nombre + ": " + edades . get ( nombre ) )
                                ;
251                 }
252         }
253 }
```

HashMap es muy eficiente para almacenar y recuperar datos cuando se tiene una clave única para cada dato. Es particularmente útil cuando se necesita buscar valores con frecuencia en función de sus claves.

Complejidad temporal de HashMap en Java:

- Insertar: O(1) caso promedio, O(1) peor caso
- Eliminar: O(1) caso promedio, O(1) peor caso
- Búsqueda: O(1) caso promedio, O(1) peor caso

Tarea: consultar la documentación para conocer los métodos de tabla hash disponibles.

Tarea: vamos a hacer una pequeña guía telefónica utilizando tablas hash. Escuche los detalles de implementación que ofrece el docente para tener una idea de qué hacer con cada método.

A continuación se muestra un ejemplo de lo que debería hacer su implementación.

```
254 --- Menú de la guía telefónica ---
255 1. Agregar/Actualizar contacto
256 2. Eliminar contacto
257 3. Búsqueda de contacto
258 4. Listar todos los contactos
259 5. Salida
260 Ingrese su elección: 1
261 Introduzca el nombre del contacto: Jonathan
262 Introducir número de teléfono : 643887102
263 Contacto añadido/actualizado exitosamente.
264
265 --- Menú de la guía telefónica ---
266 1. Agregar/Actualizar contacto
267 2. Eliminar contacto
268 3. Búsqueda de contacto
269 4. Listar todos los contactos
270 5. Salida
271 Ingrese su elección: 1
```

```
Introduzca el nombre del contacto: Luis
Introduzca el número de teléfono: 644398705 Contacto
añadido / actualizado correctamente.

--- Menú de la guía telefónica ---
1. Agregar/Actualizar contacto
2. Eliminar contacto
3. Búsqueda de contacto
4. Listar todos los contactos
5. Salida
Ingrese su elección: 4
Contactos en la agenda telefónica :
Nombre: Jonathan , Número de teléfono: 643887102
Nombre: Luis , Número de teléfono: 644398705

--- Menú de la guía telefónica ---
1. Agregar/Actualizar contacto
2. Eliminar contacto
3. Búsqueda de contacto
4. Listar todos los contactos
5. Salida
Ingrese su elección : 3
Introduzca el nombre del contacto a buscar: Luis
El número de teléfono de Luis es: 644398705

--- Menú de la guía telefónica ---
1. Agregar/Actualizar contacto
2. Eliminar contacto
3. Búsqueda de contacto
4. Listar todos los contactos
5. Salida
Ingrese su elección: 2
Ingrese el nombre del contacto a eliminar: Jonathan
Contacto eliminado exitosamente.

--- Menú de la guía telefónica ---
1. Agregar/Actualizar contacto
2. Eliminar contacto
3. Búsqueda de contacto
4. Listar todos los contactos
5. Salida
Ingrese su elección : 4
```

```
314 contactos en la agenda telefónica:
315 Nombre: Luis , Número de teléfono: 644398705
316
317 --- Menú de la guía telefónica ---
318 1. Agregar/Actualizar contacto
319 2. Eliminar contacto
320 3. Búsqueda de contacto
321 4. Listar todos los contactos
322 5. Salida
323 Ingrese su elección: 5 324 Salir de la agenda
telefónica.
```