Tecnológico Nacional de México
Campus Orizaba

Estructura de Datos

Ingeniería en Sistemas Computacionales

Tema 6
Búsquedas estructura de datos

Integrantes:

Castillo Solís Luis Ángel – 21010932
Flores Domínguez Ángel Gabriel – 21010951
Muñoz Hernández Vania Lizeth – 21011009
Romero Ovando Karyme Michelle – 21011037

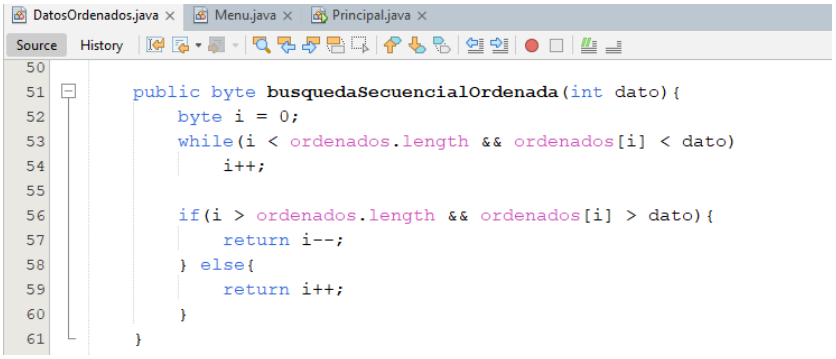
Grupo:
3g2B

Fecha de entrega: 22 / Mayo /2023

1. Algoritmos de búsqueda secuencial lineal.

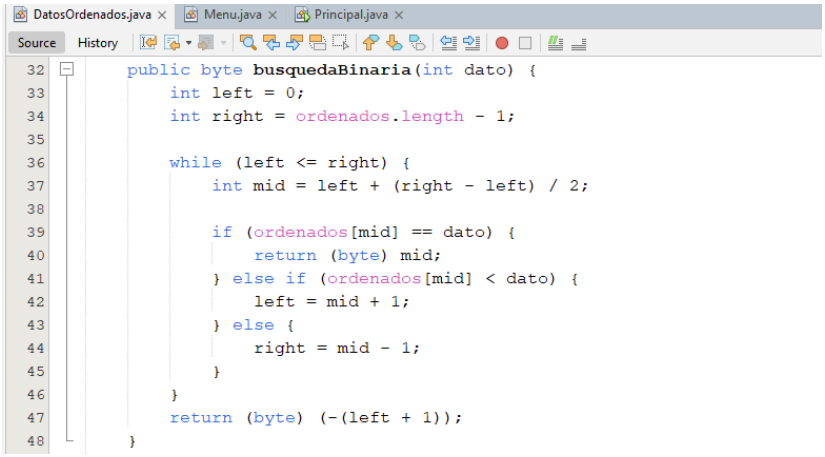
- Sustituir el algoritmo de búsqueda secuencial ordenada, por el algoritmo de búsqueda binaria en el proyecto de datos ordenados (Segundo proyecto del tema 1).

Este método es el que tenía el proyecto de Datos Ordenados con el algoritmo de Búsqueda Secuencial Lineal.

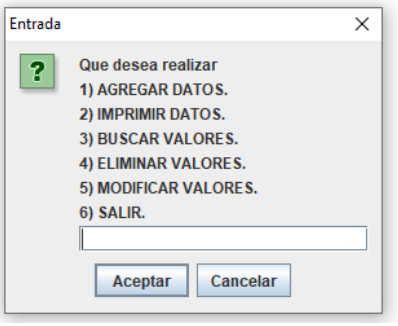


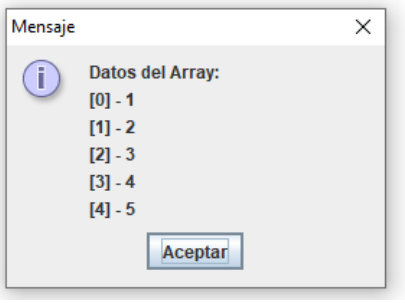

```
50
51      public byte busquedaSecuencialOrdenada(int dato){
52          byte i = 0;
53          while(i < ordenados.length && ordenados[i] < dato)
54              i++;
55
56          if(i > ordenados.length && ordenados[i] > dato){
57              return i--;
58          } else{
59              return i++;
60          }
61      }
```

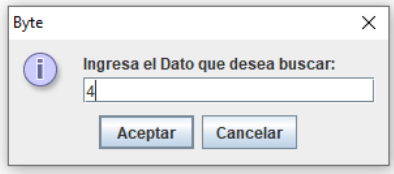
Una vez sustituido el método, queda de la siguiente manera con el algoritmo de Búsqueda Binaria y funciona de la misma manera.

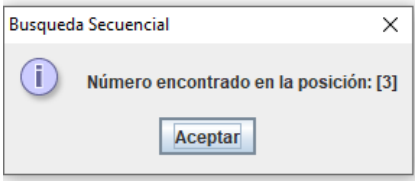


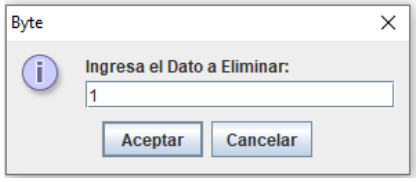

```
32      public byte busquedaBinaria(int dato) {
33          int left = 0;
34          int right = ordenados.length - 1;
35
36          while (left <= right) {
37              int mid = left + (right - left) / 2;
38
39              if (ordenados[mid] == dato) {
40                  return (byte) mid;
41              } else if (ordenados[mid] < dato) {
42                  left = mid + 1;
43              } else {
44                  right = mid - 1;
45              }
46          }
47          return (byte) (-(left + 1));
48      }
```

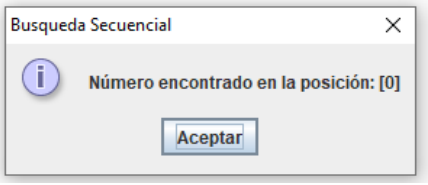


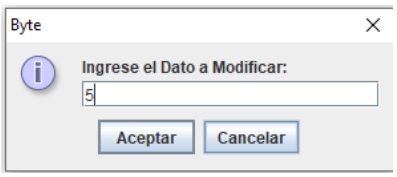


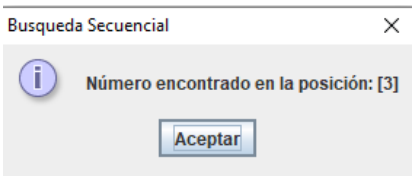


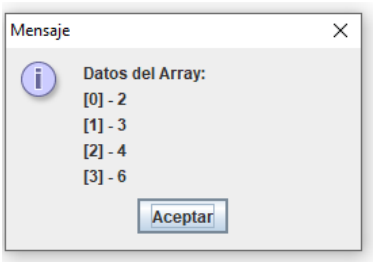


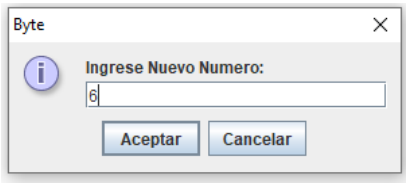












- En que consiste la búsqueda secuencial.

La búsqueda secuencial busca un elemento de una lista utilizando un valor destino llamado clave. En una búsqueda secuencial (a veces llamada búsqueda lineal), los elementos de una lista o vector se exploran (se examinan) en secuencia, uno después de otro.

El algoritmo de búsqueda secuencial compara cada elemento del array con la clave de búsqueda. Dado que el array no está en un orden prefijado, es probable que el elemento a buscar pueda ser el primer elemento, el último elemento o cualquier otro. De promedio, al menos, el programa tendrá que comparar la clave de búsqueda con la mitad de los elementos del array. El método de búsqueda lineal funcionará bien con arrays pequeños o no ordenados.

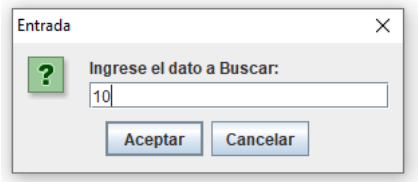
- Ejemplos de implementaciones del algoritmo en las estructuras de datos: memoria estática, memoria dinámica (listas) y archivos.

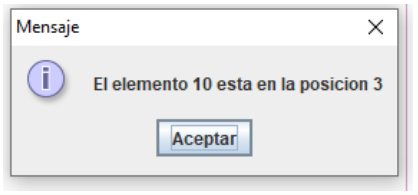
  - Ejemplo de memoria estática:

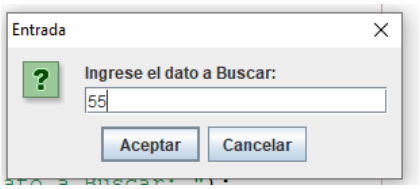
MemoriaDinamica.java | ArrayOperacion.java | MemoriaEstatica.java

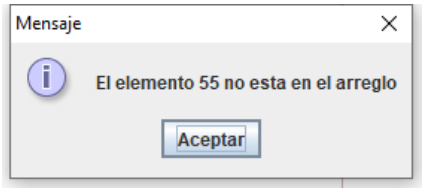
Source | History

```
1  package unidad_6;
2
3  public class MemoriaEstatica {
4      public static int busquedSec(int[] array, int dato) {
5          byte i = 0;
6          while(i < array.length && dato != array[i])
7              i++;
8          if(i < array.length){
9              return i;
10         }
11         else{
12             return -1;
13         }
14     }
16     public static void main(String[] args) {
17         int[] array = { 5, 8, 2, 10, 1, 6 };
18         int dato = Tools.leeEntero(msg: "Ingrese el dato a Buscar: ");
19
20         int pos = busquedSec(array, dato);
21         if (pos != -1) {
22             Tools.imprimeMSJ("El elemento " + dato + " esta en la posicion " + pos);
23         } else {
24             Tools.imprimeMSJ("El elemento " + dato + " no esta en el arreglo");
25         }
26     }
27 }
```









En este ejemplo, se declara un arreglo de nombre array con algunos datos, y buscamos el elemento que el usuario ingrese, utilizando la búsqueda secuencial.

El método busquedaSec recibe como parámetros el arreglo y el dato ingresado, y va a devolver la posición en la que se encuentra el elemento (si se encuentra) o -1 si no se encuentra. En el main se crea el arreglo, se pide el dato al usuario y luego se llama al método busquedaSec y se almacena el resultado en la variable pos.

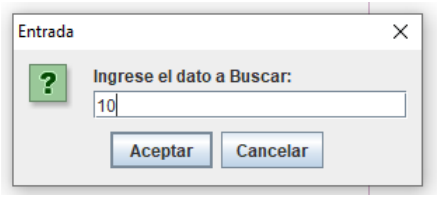
Si el resultado no es -1, significa que se encontró el elemento y se muestra un mensaje indicando la posición. De lo contrario, se muestra un mensaje indicando que el elemento no se encontró en el arreglo.

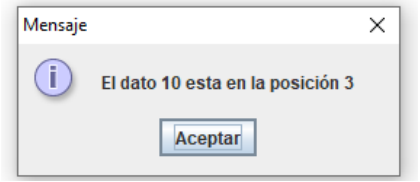
- Ejemplo memoria dinámica (listas):

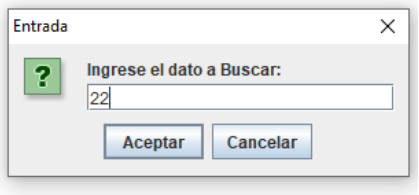
MemoriaEstatica.java | MemoriaDinamica.java | ArrayOperacion.java

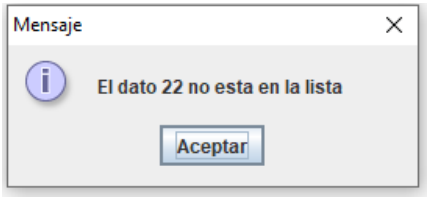
Source | History

```
1  package unidad_6;
2  import java.util.ArrayList;
3  import java.util.List;
4
5  public class MemoriaDinamica {
6      public static int busquedaSec(List<Integer> lista, int dato) {
7          byte i = 0;
8          while(i < lista.size() && dato!= lista.get(index:i))
9              i++;
10         if(i < lista.size()){
11             return i;
12         }
13         else{
14             return -1;
15         }
16     }
19     public static void main(String[] args) {
20         List<Integer> lista = new ArrayList<>();
21         lista.add(e: 2);
22         lista.add(e: 5);
23         lista.add(e: 8);
24         lista.add(e: 10);
25         lista.add(e: 1);
26         lista.add(e: 6);
27         int dato = Tools.leeEntero(msg: "Ingrese el dato a Buscar: ");
28         int pos = busquedaSec(lista, dato);
29
30         if (pos != -1) {
31             Tools.imprimeMSJ("El dato " + dato + " esta en la posición " + pos);
32         } else {
33             Tools.imprimeMSJ("El dato " + dato + " no esta en la lista");
34         }
35     }
36 }
```









En este ejemplo, utilizamos la clase ArrayList de Java para representar una lista dinámica. El método busquedaSec recibe como parámetros la lista y el elemento a buscar, y realiza una búsqueda secuencial en la lista para encontrar el elemento.

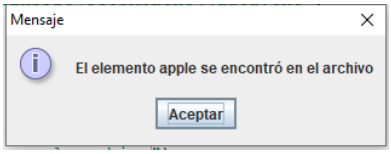
En el main se crea una instancia de ArrayList llamada lista y ahí agregamos varios datos. Se pide al usuario por medio de una ventana emergente ingresar el dato a buscar y se guarda en la variable dato, después de eso se manda a llamar al metodo de busquedaSec, almacenando al resultado en la variable pos.

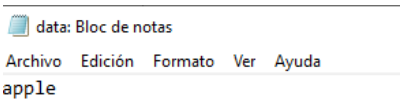
Si el resultado no es -1, se muestra un mensaje indicando la posición en la que se encontró el elemento. De lo contrario, se muestra un mensaje indicando que el elemento no se encontró en la lista.

En una lista, los elementos pueden estar en cualquier posición, por lo que la búsqueda secuencial recorrerá la lista de manera secuencial hasta encontrar el elemento o llegar al final de la lista.

- Ejemplo en archivos.

```java
package unidad_6;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Archivos {
    public static boolean busqueda(String url, String dato) {
        try (BufferedReader br = new BufferedReader(new FileReader(string: url))) {
            String line;
            while ((line = br.readLine()) != null) {
                if (line.equals(anObject:dato)) {
                    return true; // El elemento se encontró en el archivo
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false; // El elemento no se encontró en el archivo
    }
    public static void main(String[] args) {
        String url = "C:\\Users\\ovand\\OneDrive\\Documentos\\NetBeansProjectsApache\\data.txt";
        String dato = "apple";

        boolean result = busqueda(url, dato);

        if (result) {
            Tools.imprimeMSJ("El elemento " + dato + " se encontró en el archivo");
        } else {
            Tools.imprimeMSJ("El elemento " + dato + " no se encontró en el archivo");
        }
    }
}
```





En este ejemplo, se utiliza la clase BufferedReader para leer un archivo de texto línea por línea. El método búsqueda recibe como parámetros el nombre del archivo y el elemento que se quiere buscar en el archivo.

En el main, se establece el nombre del archivo y el objetivo a buscar que es “apple”. Luego se llama al método de búsqueda y almacena el resultado en la variable result. Si el resultado es true, significa que el elemento se encontró en el archivo y se muestra un mensaje indicando que se encontró. De lo contrario, se muestra un mensaje indicando que el elemento no se encontró en el archivo.

Análisis de eficiencia.

En términos de eficiencia, el algoritmo de búsqueda secuencial lineal es simple de implementar, pero no es eficiente para listas grandes o cuando se necesita realizar búsquedas frecuentes. Si la lista está ordenada, es más eficiente utilizar algoritmos de búsqueda binaria u otros métodos de búsqueda más sofisticados que aprovechen la información de ordenamiento. (Page, 2019)

Considerando la cantidad de comparaciones:

- Mejor caso: El elemento buscado está en la primera posición. Es decir, se hace una sola comparación.
- Peor caso: El elemento buscado está en la última posición. Necesitando igual cantidad de comparaciones que de elementos el arreglo.
- En promedio: El elemento buscado estará cerca de la mitad. Necesitando en promedio, la mitad de las comparaciones que de elementos.

Por lo tanto, la velocidad de ejecución depende linealmente del tamaño del arreglo. Es importante considerar estos factores al elegir el algoritmo de búsqueda adecuado según las características y tamaño de los datos a buscar.

En resumen, el análisis de eficiencia del algoritmo de búsqueda secuencial lineal es:

- Mejor caso: O(1) (constante).
- Peor caso: O(n) (lineal).

Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

La complejidad de la búsqueda secuencial diferencia entre el comportamiento en el peor y mejor caso.

- El mejor caso: se encuentra cuando aparece una coincidencia en el primer elemento de la lista, por lo que el tiempo de ejecución es O(1).
- El peor caso: se produce cuando el elemento no está en la lista 0 se encuentra al final de ella, Esto requiere buscar en todos los n términos, lo que implica una complejidad de O(n).
- El caso medio: requiere un poco de razonamiento probabilista. Para el supuesto de una lista aleatoria es probable que ocurra una coincidencia en cualquier posición, Después de la ejecución de un número grande de búsquedas, la posición media para una coincidencia es el elemento central n/2. El elemento central se obtiene después de n/2 comparaciones, que definen el coste esperado de la búsqueda Por esta razón, se dice que la prestación media de la búsqueda secuencial es O(n).

- Complejidad en el tiempo y complejidad espacial.

  - Complejidad del tiempo:

La complejidad en el tiempo se refiere a la cantidad de tiempo requerido por el algoritmo para ejecutarse en función del tamaño de los datos de entrada. En el caso del algoritmo de búsqueda secuencial lineal, en el peor caso, se debe recorrer toda la lista de elementos hasta encontrar el elemento buscado o determinar que no está presente.

Por lo tanto, la complejidad en el tiempo del algoritmo de búsqueda secuencial lineal es O(n), donde "n" es el tamaño de la lista. Esto significa que el tiempo de ejecución aumenta linealmente con el tamaño de la lista. (Rivera, 2021)

- Caso promedio: La complejidad temporal del algoritmo de búsqueda lineal es O(n).
- Mejor caso: La complejidad del tiempo en el mejor de los casos es O(1). Ocurre cuando el elemento a buscar es el primer elemento presente dentro del array.
- Peor caso: El peor de los casos ocurre cuando el elemento que estamos buscando no está presente dentro del array o está en el último índice del array. La complejidad de tiempo en el peor de los casos es O(n).

  - Complejidad espacial:

La complejidad espacial se refiere a la cantidad de memoria requerida por el algoritmo para almacenar y manipular los datos de entrada. En el caso del algoritmo de búsqueda secuencial lineal, no se requiere memoria adicional más allá de la lista o arreglo original donde se realiza la búsqueda.

Por lo tanto, la complejidad espacial del algoritmo de búsqueda secuencial lineal es O(1), es decir, constante. No importa cuántos elementos haya en la lista, la cantidad de memoria adicional necesaria para ejecutar el algoritmo no aumenta. (Jindal, 2021)

2. Algoritmos de búsqueda binaria.

- En que consiste la búsqueda binaria.

El algoritmo de búsqueda binaria lo que hace es repetidamente apuntar al centro de la estructura de búsqueda y dividir el espacio restante por la mitad, así hasta encontrar el valor buscado. Esto es una gran ventaja en cuanto a la búsqueda lineal que, para encontrar el valor buscado, debe ir descartando elemento por elemento. Si una lista está ordenada, la búsqueda binaria proporciona una técnica de

búsqueda mejorada. Una búsqueda binaria típica es la búsqueda de una palabra en un diccionario. (F, 2017)

Dada la palabra, se abre el libro cerca del principio, del centro o del final dependiendo de la primera letra de la palabra que busca. Se puede tener suerte y acertar con la página correcta pero, normalmente, no será así y el lector se mueve a la página anterior o posterior del libro. Por ejemplo, si la palabra comienza con “J” y se está en la “L” se mueve uno hacia atrás. El proceso continúa hasta que se encuentra la página buscada o hasta que se descubre que la palabra no está en la listar

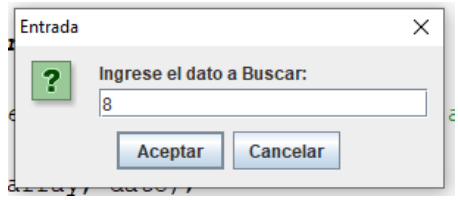
Una idea similar se aplica en la búsqueda en una lista ordenada. Se sitúa la lectura en el centro de la lista y se comprueba si nuestra clave coincide con el valor del elemento central Si no se encuentra el valor de la clave, se sitúa uno en la mitad inferior o superior del elemento central de la lista. En general, si los datos de la lista están ordenados, se puede utilizar esa información para acortar el tiempo de búsqueda.

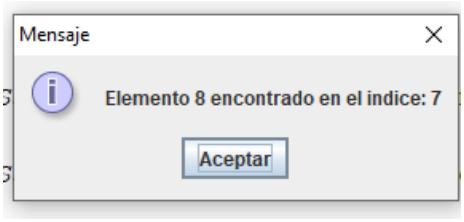
Ejemplos de implementaciones del algoritmo en las estructuras de datos.

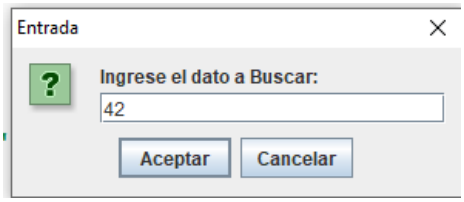
1. Implementación en un arreglo ordenado:

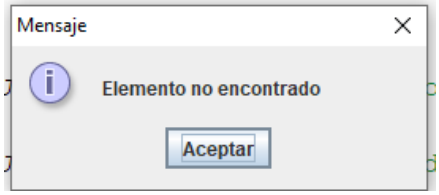
Ejemplo1.java

```java
package BusquedaBinaria;

public class Ejemplo1 {

    public static int binario(int[] array, int dato) {
        int izq = 0;
        int der = array.length - 1;

        while (izq <= der) {
            int mid = izq + (der - izq) / 2;

            if (array[mid] == dato) {
                return mid;
            } else if (array[mid] < dato) {
                izq = mid + 1;
            } else {
                der = mid - 1;
            }
        }
        return -1; // no esta en el arreglo
    }
    public static void main(String[] args) {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int dato = Tools.leeEntero(msg: "Ingrese el dato a Buscar: ");

        int pos = binario(array, dato);

        if (pos != -1) {
            Tools.imprimeMSJ("Elemento encontrado en el índice: " + pos);
        } else {
            Tools.imprimeMSJ(msj: "Elemento no encontrado");
        }
    }
}
```









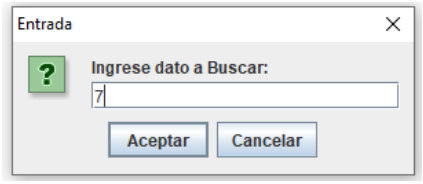
El método binario que realiza la búsqueda binaria en un arreglo recibe como parámetros un arreglo y el elemento que se desea encontrar. En esta búsqueda define dos variables, “izq” y “der”, el primero apunta al primer elemento del arreglo (índice 0) y el segundo punta al último elemento del arreglo (array.length - 1). En el while se calcula “mid”, esto para asegurar que el pos medio este en el centro del rango de búsqueda actual. Se verifica si el elemento en array[mid] es igual al dato que se busca, si es así se ha encontrado en elemento y se devuelve mid.

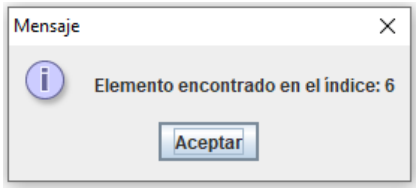
Si array[mid] es menor que el dato, se actualiza el índice izq a mid + 1 para restringir la búsqueda a la mitad derecha del rango actual. Si array[mid] es mayor que el dato se actualiza el índice der a mid – 1 para restringir la búsqueda a la mitad izquierda del rango actual. El while se va a repetir mientras izq sea menor o igual que der. Si izq se vuelve mayor que der, significa que el elemento no está presente en el arreglo y se devuelve -1.

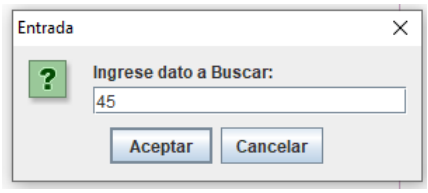
En el main se crea un arreglo ordenado y se pide el elemento a buscar que se guarda en la variable dato. Se llama al método binario pasando el arreglo y el dato a buscar y se almacena en pos. Se verifica si pos es diferente de -1, si es así se imprime un mensaje indicando que el elemento fue encontrado en el índice pos. De lo contrario, se imprime un mensaje indicando que el elemento no fue encontrado.

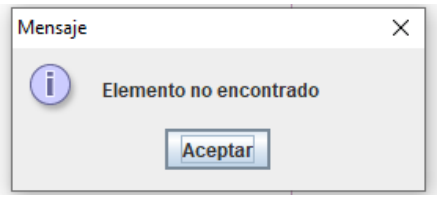
2. ArrayList ordenado:

```java
package BusquedaBinaria;
import java.util.ArrayList;

public class Ejemplo2 {

    public static int binario(ArrayList<Integer> lista, int dato) {
        int izq = 0;
        int der = lista.size() - 1;

        while (izq <= der) {
            int mid = izq + (der - izq) / 2;

            if (lista.get(index:mid) == dato) {
                return mid;
            } else if (lista.get(index:mid) < dato) {
                izq = mid + 1;
            } else {
                der = mid - 1;
            }
        }
        return -1; // El elemento no se encuentra en el ArrayList
    }
    public static void main(String[] args) {
        ArrayList<Integer> lista = new ArrayList<>();
        lista.add(e: 1);
        lista.add(e: 2);
        lista.add(e: 3);
        lista.add(e: 4);
        lista.add(e: 5);
        lista.add(e: 6);
        lista.add(e: 7);
        lista.add(e: 8);
        lista.add(e: 9);

        int dato = Tools.leeEntero(msg: "Ingrese dato a Buscar: ");
        int pos = binario(lista, dato);

        if (pos != -1) {
            Tools.imprimeMSJ("Elemento encontrado en el índice: " + pos);
        } else {
            Tools.imprimeMSJ(msj: "Elemento no encontrado");
        }
    }
}
```









En este ejemplo se utiliza un ArrayList ordenado para realizar la búsqueda binaria. El método binario recibe como parámetros el arraylist y el elemento a buscar, es básicamente el mismo método que en el ejemplo anterior. El while se va a repetir hasta que izq sea meyor que der, momento en el que se devuelve -1 para indicar que el elemento no se encontró en el arraylist.

En la función main, se crea un ArrayList ordenado y se realiza una búsqueda del elemento objetivo utilizando el método binario. Luego, se imprime un mensaje indicando si el elemento se encontró o no, junto con el índice si se encontró.

- Análisis de eficiencia.

La búsqueda binaria es un algoritmo eficiente para buscar un elemento específico en una lista ordenada. Su eficiencia se basa en la división sucesiva del espacio de búsqueda en dos partes, descartando la mitad de los elementos en cada iteración.

Es especialmente útil cuando se trabaja con listas ordenadas, ya que aprovecha esta propiedad para reducir rápidamente el espacio de búsqueda y encontrar el elemento deseado en menos pasos en comparación con la búsqueda lineal u otros algoritmos de búsqueda. (Gil, 2020)

Contado comparaciones:

- Mejor caso: El elemento buscado está en el centro. Por lo tanto, se hace una sola comparación.
- Peor caso: El elemento buscado está en una esquina. Necesitando log2(n) cantidad de comparaciones.
- En promedio: Serán algo como log2(n/2).

Por lo tanto, la velocidad de ejecución depende logarítmicamente del tamaño del arreglo.

- Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos:

La eficiencia de la búsqueda binaria puede variar según el caso en el que se encuentre.

- *Mejor de los casos:*

  Ocurre cuando el elemento buscado se encuentra exactamente en el medio de la lista. En este caso, la búsqueda binaria encuentra el elemento deseado en el primer paso, lo que significa que solo se requiere una comparación para encontrar el elemento.

- Por ejemplo:

Se tiene una lista ordenada de números del 1 al 10, y se quiere buscar el número 5. La lista sería: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. En este caso, la búsqueda binaria identifica rápidamente que el número 5 se encuentra en la posición central de la lista y lo encuentra en el primer paso.

- *Caso medio:*

Se refiere a situaciones en las que el elemento buscado no está en el centro de la lista, pero tampoco en el extremo. En este caso, se necesitan más comparaciones, pero aun así, la búsqueda binaria es eficiente debido a su naturaleza de dividir el espacio de búsqueda a la mitad en cada iteración.

- Por ejemplo:

Se tiene una lista ordenada de números del 1 al 10, y se quiere buscar el número 8. La lista sería: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. En este caso, la búsqueda binaria divide la lista en dos mitades y se compara el número buscado con el elemento central. Al darse cuenta de que el número buscado es mayor que el elemento central, descarta la primera mitad de la lista y continúa la búsqueda en la segunda mitad. Realiza una o dos iteraciones adicionales y encuentra el número.

- *Peor de los casos:*

Ocurre cuando el elemento buscado no está presente en la lista. En este caso, la búsqueda binaria debe realizar todas las iteraciones posibles hasta llegar al punto en el que el espacio de búsqueda se reduce a un solo elemento.

- Por ejemplo:

Se tiene una lista ordenada de números del 1 al 10, y queremos buscar el número 11. La lista sería: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. En este caso, la búsqueda binaria divide la lista en dos mitades, compara el número buscado con el elemento central y se da cuenta de que es mayor. Descarta la primera mitad de la lista y continúa la búsqueda en la segunda mitad. Repite este proceso varias veces hasta que el espacio de búsqueda se reduce a un solo elemento, pero ese elemento no es igual al número buscado. En este caso, se requieren log2(n) iteraciones para descubrir que el elemento no está presente.

- Complejidad en el tiempo y complejidad espacial.

  - Complejidad en el tiempo:

La búsqueda binaria tiene una complejidad en el tiempo de O(log n), donde "n" representa el tamaño del conjunto de elementos ordenados en el que se realiza la búsqueda. Esto significa que el tiempo de ejecución del algoritmo crece de forma logarítmica con respecto al tamaño del conjunto de elementos.

En cada paso de la búsqueda binaria, el algoritmo divide el conjunto de elementos restantes a la mitad, reduciendo así el espacio de búsqueda. Esto se realiza comparando el elemento a buscar con el elemento central del conjunto actual y descartando la mitad en la que no puede estar el objetivo. Dado que el conjunto se divide en dos en cada paso, el algoritmo puede encontrar el elemento en un tiempo relativamente pequeño, incluso para conjuntos de elementos muy grandes.

  - Complejidad espacial:

La complejidad espacial de la búsqueda binaria es de O(1), lo que significa que el algoritmo utiliza una cantidad constante de espacio adicional, independientemente del tamaño del conjunto de elementos. En la búsqueda binaria, solo se necesita almacenar el valor del elemento central, el valor del objetivo y algunas variables adicionales para realizar las comparaciones y realizar un seguimiento de los índices de inicio y fin del conjunto de elementos actual. No se requiere almacenar ningún otro elemento del conjunto, lo que resulta en una complejidad espacial constante.

Dado que la cantidad de espacio utilizado por el algoritmo no depende del tamaño del conjunto de elementos, la complejidad espacial se considera constante, O(1). Esto hace que la búsqueda binaria sea eficiente en términos de uso de memoria.

3. Búsqueda de patrones de Knuth Morris Pratt.

- En que consiste la búsqueda de patrones de Knuth Morris Pratt.

  Lo que hace este algoritmo es utilizar los datos de comparación anteriores.
  Se calcula una función de prefijos que brinda información sobre el patrón a la hora de hacer las comparaciones. Permite saber el corrimiento sobre el patrón hasta la próxima comparación con algún carácter en el texto, esta información es registrada en una tabla de fallos o tabla de siguientes que dice como realizar las comparaciones en la cadena donde se busca un patrón.
  Puede buscar un patrón en O(n) tiempo, ya que nunca vuelve a comparar un símbolo de texto que coincida con un símbolo de patrón. Sin embargo, utiliza la tabla de fallos o siguientes para analizar la estructura del patrón. La construcción

de una tabla de siguientes toma O(m) tiempo. Por lo tanto, la complejidad temporal general del algoritmo KMP es O(m + n).

- Ejemplos de implementaciones del algoritmo en las estructuras de datos

```java
package ListasSimples;
import java.util.ArrayList;
import java.util.List;
import EntradaSalida.Tools;
public class KMP {

    public static List<Integer> busquedaKMP(String texto, String patron) {
        List<Integer> coincidencias = new ArrayList<>();
        int n = texto.length();
        int m = patron.length();
        int[] tabla = tablaPrefijos(patron);

        int i = 0; // Índice para el texto
        int j = 0; // Índice para el patrón

        while (i < n) {
            if (patron.charAt(j) == texto.charAt(i)) {
                j++;
                i++;
            }

            if (j == m) {
                coincidencias.add(i - j);
                j = tabla[j - 1];
            } else if (i < n && patron.charAt(j) != texto.charAt(i)) {
                if (j != 0) {
                    j = tabla[j - 1];
                } else {
                    i++;
                }
            }
        }

        return coincidencias;
    }

```

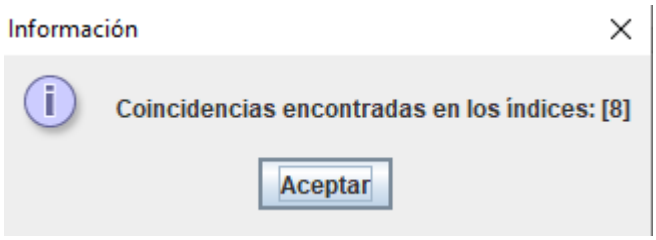
Este primer método, “busquedaKMP” se utiliza para buscar un patrón en un texto utilizando el algoritmo KMP. Su resultado es una lista de enteros que indican los índices donde se encontraron coincidencias del patrón en el texto.

```java
    private static int[] tablaPrefijos(String patron) {
        int m = patron.length();
        int[] lps = new int[m];
        int len = 0; // Longitud del sufijo más largo previo

        int i = 1;
        while (i < m) {
            if (patron.charAt(i) == patron.charAt(len)) {
                len++;
                lps[i] = len;
                i++;
            } else {
                if (len != 0) {
                    len = lps[len - 1];
                } else {
                    lps[i] = 0;
                    i++;
                }
            }
        }

        return lps;
    }

```

Este método se encarga de construir la tabla de prefijos (también conocida como el arreglo LPS) utilizando el patrón proporcionado.

```java
    public static void main(String[] args) {
        String texto = "CTCACTGCCTGCCTAG";
        String patron = "CTGCCTAG";
        List<Integer> res = busquedaKMP(texto, patron);

        if (res.isEmpty()) {
            Tools.errorMsj("No se encontraron coincidencias del patrón en el texto.");
        } else {
            Tools.imprime("Coincidencias encontradas en los índices: " + res);
        }
    }
}
```

En el método main, se muestra un ejemplo de uso donde se busca el patrón que nosotros deseemos en el texto. Si se encuentran coincidencias, se imprime la lista de índices donde se encontraron. En caso contrario, manda un mensaje de error, informándonos de que no se encontraron coincidencias.



- Análisis de eficiencia

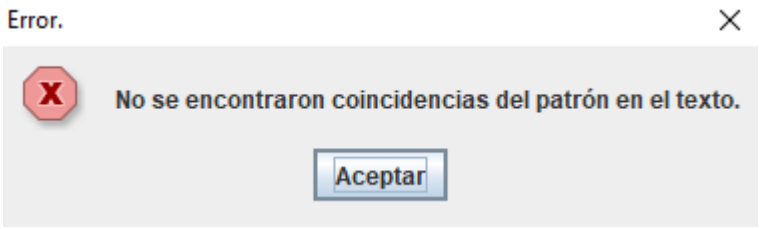
Eficiente en la búsqueda de patrones en una cadena de texto, especialmente cuando se trata de patrones largos o cuando el patrón contiene repeticiones. Su utilización es amplia en aplicaciones que involucran búsqueda de texto, procesamiento de lenguaje natural, análisis de secuencias de ADN y muchas otras áreas donde la eficiencia en la búsqueda de patrones es crucial. Su eficiencia radica en la construcción de la tabla de prefijos, que evita comparaciones innecesarias al buscar el patrón en el texto.

- Análisis de los casos: mejor de los casos, caso medio y peor de los casos con ejemplos

Para poder analizar los casos de rendimiento del KMP hay que basarnos en el número de comparaciones realizadas entre el texto y el patrón durante la búsqueda. Los casos de rendimiento se dividen en tres categorías: el mejor de los casos, el caso medio y el peor de los casos. Veamos cada uno de ellos con ejemplos:

1. Mejor de los casos: El mejor de los casos ocurre cuando el patrón no se repite en absoluto en el texto. En este caso, el algoritmo KMP realiza el mínimo número de comparaciones posible, lo que lo convierte en su mejor rendimiento.

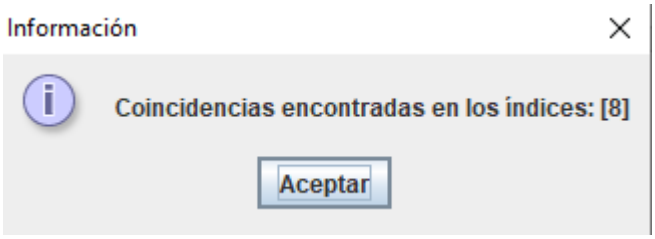
   Ejemplo:
   Texto: "abcdefgh"
   Patrón: " xyz "



En este caso, el algoritmo KMP recorrerá todo el texto sin encontrar ninguna coincidencia del patrón. Realizará n comparaciones, donde n es la longitud

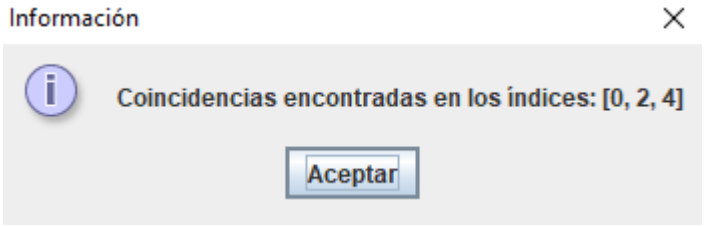
del texto. Por lo tanto, la complejidad de tiempo en el mejor de los casos es O(n).

2. Caso medio: El caso medio ocurre cuando hay algunas coincidencias parciales del patrón en el texto, pero el patrón no se repite completamente. Aquí, el patrón se encuentra parcialmente en diferentes partes del texto. El algoritmo KMP utilizará la información de la tabla de prefijos para evitar comparaciones innecesarias y encontrar todas las coincidencias. Este es el caso que hemos mostrado en el ejemplo de aplicación.



3. Peor de los casos: El peor de los casos ocurre cuando el patrón se repite completamente en el texto. Esto significa que habrá múltiples coincidencias completas del patrón en diferentes posiciones del texto.

   Ejemplo:
   Texto: "abababab"
   Patrón: "abab"



   Aquí, el patrón "abab" se repite completamente en el texto. El algoritmo KMP encontrará todas las coincidencias, pero realizará más comparaciones que en los casos anteriores.

- Complejidad en el tiempo y complejidad espacial

  - Complejidad de tiempo:
    La construcción de la tabla de prefijos tiene una complejidad de tiempo de O(m), donde m es la longitud del patrón. Esto se debe a que se recorre el patrón una vez para calcular los valores de la tabla de prefijos.

- Complejidad de espacio:
  El KMP utiliza espacio adicional para almacenar la tabla de prefijos, que tiene una longitud igual a la del patrón. Por lo tanto, la complejidad de espacio es O(m), donde m es la longitud del patrón.

4. METODO SALTAR BUSQUEDA.

En que consiste el método saltar búsqueda.

La Skip List es una estructura de datos probabilística que proporciona una alternativa eficiente a las listas enlazadas y a los árboles balanceados en ciertos escenarios.

La Skip List está compuesta por una serie de listas enlazadas paralelas, donde cada lista representa un nivel diferente de la estructura. La lista más baja es similar a una lista enlazada simple y contiene todos los elementos de la estructura. A medida que subes de nivel, los elementos se "saltan" con una cierta probabilidad determinada por una distribución aleatoria.

El objetivo principal de utilizar el salto en la Skip List es acelerar las operaciones de búsqueda. Al saltar niveles, se puede evitar recorrer toda la estructura de datos para encontrar un elemento en particular, lo que resulta en una complejidad de búsqueda promedio de O(log n), donde "n" es el número de elementos en la lista.

Los pasos involucrados en la búsqueda en una Skip List:

1. Comenzando desde la lista superior (la más alta), comparas el valor actual con el valor que estás buscando.
2. Si el valor actual es mayor, te mueves a la lista inferior y repites el paso 1.
3. Si el valor actual es menor o igual, te mueves al siguiente elemento en la misma lista y repites el paso 1.
4. Si el valor actual es igual al valor buscado, has encontrado el elemento y la búsqueda termina.

Si mientras realizas los pasos anteriores llegas al final de la lista actual sin encontrar el elemento buscado, te desplazas hacia abajo a la lista inferior y continúas desde el primer paso. Este proceso se repite hasta que se encuentre el elemento o hasta que llegues a la lista más baja.

La inserción y eliminación de elementos en una Skip List también involucran el uso del salto para mantener la estructura equilibrada y cumplir con las probabilidades de salto adecuadas.

En resumen, la técnica de salto en la búsqueda de una Skip List te permite reducir el tiempo necesario para encontrar elementos en la estructura, evitando recorrer todas las listas enlazadas en cada operación de búsqueda.

- Ejemplos de implementaciones del algoritmo en las estructuras de datos

```java
package Ejemplos;

import java.util.Random;

class SkipListNode {
    int value;
    SkipListNode next;
    SkipListNode down;

    public SkipListNode(int value) {
        this.value = value;
        this.next = null;
        this.down = null;
    }
}

public class MetodoSkip {
    private static final double PROBABILITY = 0.5; // Probabilidad de salto

    private SkipListNode head;
    private Random random;

    public MetodoSkip() {
        head = new SkipListNode( value: Integer.MIN_VALUE);
        random = new Random();
    }

    public boolean search(int target) {
        SkipListNode current = head;

```

```
31          while (current != null) {
32              if (current.value == target) {
33                  return true; // Elemento encontrado
34              } else if (current.next == null || current.next.value > target) {
35                  current = current.down; // Moverse hacia abajo
36              } else {
37                  current = current.next; // Moverse hacia adelante
38              }
39          }
40
41          return false; // Elemento no encontrado
42      }
43
44      public void insert(int value) {
45          SkipListNode current = head;
46
47          while (current != null) {
48              if (current.next == null || current.next.value > value) {
49                  if (current.down == null) {
50                      // Insertar nuevo nodo en el nivel más bajo
51                      SkipListNode newNode = new SkipListNode(value);
52                      newNode.next = current.next;
53                      current.next = newNode;
54                      break;
55                  }
56                  current = current.down; // Moverse hacia abajo
57              } else {
58                  current = current.next; // Moverse hacia adelante
59              }
60          }
```

```
61
62          maybePromote(); // Probabilidad de promoción a un nivel superior
63      }
64
65      public void delete(int value) {
66          SkipListNode current = head;
67
68          while (current != null) {
69              if (current.next == null || current.next.value > value) {
70                  current = current.down; // Moverse hacia abajo
71              } else if (current.next.value == value) {
72                  current.next = current.next.next; // Eliminar el nodo
73                  current = current.down; // Moverse hacia abajo
74              } else {
75                  current = current.next; // Moverse hacia adelante
76              }
77          }
78      }
79
80      private void maybePromote() {
81          if (random.nextDouble() < PROBABILITY) {
82              SkipListNode newHead = new SkipListNode( value: Integer.MIN_VALUE);
83              newHead.down = head;
84              head = newHead;
85          }
86      }
87
88      // Método auxiliar para imprimir la Skip List
89      public void printList() {
90          SkipListNode current = head;
```

```
91
92          while (current != null) {
93              SkipListNode levelNode = current;
94              while (levelNode != null) {
95                  System.out.print(levelNode.value + " ");
96                  levelNode = levelNode.next;
97              }
98              System.out.println();
99              current = current.down;
100         }
101     }
102
103     // Ejemplo de uso
104     public static void main(String[] args) {
105         MetodoSkip skipList = new MetodoSkip();
106
107         skipList.insert( value: 3);
108         skipList.insert( value: 6);
109         skipList.insert( value: 2);
110         skipList.insert( value: 8);
111         skipList.insert( value: 1);
112
113         skipList.printList(); // Imprime la Skip List
114
115         System.out.println("¿Existe el valor 6? " + skipList.search( target: 6));
116         System.out.println("¿Existe el valor 5? " + skipList.search( target: 5));
117
118
119     }
120 }
```

Output - Unidad6 (run)

```
run:
-2147483648
-2147483648
-2147483648
-2147483648
-2147483648 1 2 3 6 8
¿Existe el valor 6? true
¿Existe el valor 5? false
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Explicación del código.

La clase SkipListNode representa un nodo de la Skip List. Cada nodo tiene un valor, una referencia al siguiente nodo en el mismo nivel (next), y una referencia al nodo inferior en el nivel inferior (down).

La clase SkipList es la implementación de la Skip List. Tiene un nodo head que representa el nivel más alto de la Skip List y un generador de números aleatorios random para la probabilidad de promoción.

El método search recibe un valor target y busca ese valor en la Skip List. Comienza desde el nodo head y recorre los niveles de la lista, moviéndose hacia abajo o hacia adelante según corresponda. Si encuentra el valor, retorna true; de lo contrario, retorna false.

El método insert recibe un valor y lo inserta en la Skip List. Comienza desde el nodo head y recorre los niveles de la lista. Si encuentra el lugar correcto para insertar el valor (entre dos nodos o en el nivel más bajo), crea un nuevo nodo y lo inserta en la posición adecuada.

El método delete recibe un valor y lo elimina de la Skip List. Recorre los niveles de la lista y busca el nodo con el valor deseado. Si lo encuentra, simplemente lo elimina ajustando las referencias next apropiadas.

El método maybePromote se llama después de cada inserción y tiene una probabilidad de promover el nivel más alto de la Skip List. Esto se logra creando un nuevo nodo newHead que se convierte en el nuevo nodo head, con una referencia down al antiguo nodo head. Esto ayuda a mantener la estructura equilibrada.

El método printList es un método auxiliar para imprimir la Skip List. Recorre los niveles de la lista y muestra los valores de cada nivel en una línea separada.

En el ejemplo de uso en el método main, se crea una instancia de SkipList. Luego se insertan varios valores en la lista y se llama al método printList para mostrar la estructura de la Skip List resultante.

Finalmente, se realiza una búsqueda en la Skip List para verificar si un valor específico existe en ella.

Eso es básicamente cómo funciona la implementación de la Skip List en el código proporcionado.

- Análisis de eficiencia.

Búsqueda (Search): En el peor caso, la búsqueda en una Skip List tiene una complejidad de tiempo promedio de O(log n), donde "n" es el número de elementos en la lista. Esto se debe a que en cada nivel de la lista, el número de nodos a recorrer

se reduce a la mitad en promedio. Dado que la lista puede tener múltiples niveles, la búsqueda se realiza a través de múltiples capas, y la cantidad total de nodos recorridos se mantiene en un rango logarítmico.

Inserción (Insert): La inserción en una Skip List también tiene una complejidad de tiempo promedio de O(log n). Al igual que en la búsqueda, se recorren múltiples capas de la lista para encontrar el lugar adecuado para insertar el nuevo nodo. La probabilidad de promoción de nuevos niveles ayuda a mantener un equilibrio en la estructura, evitando que la lista crezca en exceso y manteniendo la complejidad logarítmica.

Eliminación (Delete): La eliminación en una Skip List tiene una complejidad de tiempo promedio de O(log n). Al igual que en la búsqueda y la inserción, se recorren múltiples capas de la lista para encontrar el nodo que se desea eliminar. Una vez que se encuentra, la eliminación implica ajustar las referencias next apropiadas para eliminar el nodo de la lista.

Es importante tener en cuenta que estos análisis de eficiencia son en promedio, ya que las operaciones en una Skip List dependen de la probabilidad de promoción de nuevos niveles y la distribución aleatoria de los elementos. En el peor caso, si todos los elementos se promocionan a niveles superiores, la complejidad podría ser O(n), similar a una lista enlazada simple.

En cuanto al espacio, la Skip List requiere un espacio adicional para almacenar las referencias y nodos adicionales en los niveles superiores. En general, el espacio requerido por una Skip List es proporcional al número de elementos almacenados en ella.

La Skip List ofrece un equilibrio entre las estructuras de datos basadas en listas enlazadas y los árboles balanceados. Proporciona una complejidad logarítmica promedio para las operaciones de búsqueda, inserción y eliminación, lo que la convierte en una opción eficiente en muchos escenarios. Sin embargo, es importante tener en cuenta que la elección de una estructura de datos depende de las características específicas del problema y de los requisitos de eficiencia y espacio de cada caso de uso.

Análisis de casos: mejor de los casos, caso medio, y peor de los casos

1. Búsqueda (Search):

Mejor caso: El mejor caso ocurre cuando el elemento buscado se encuentra en el primer nivel de la lista, es decir, en la lista más alta. En este caso, la complejidad de tiempo sería O(1), ya que el elemento se encuentra directamente en el primer nivel sin necesidad de descender a niveles inferiores.

Caso medio: En promedio, la búsqueda en una Skip List tiene una complejidad de tiempo de O(log n). Esto se debe a que en cada nivel, el número de nodos a recorrer se reduce a la mitad en promedio, lo que lleva a un número de pasos logarítmico en relación con el tamaño de la lista.

Peor caso: El peor caso ocurre cuando el elemento buscado se encuentra en el nivel más bajo de la lista, lo que implica recorrer todos los niveles de la lista para llegar a él. En este caso, la complejidad de tiempo sería O(log n), ya que se deben recorrer todos los niveles para encontrar el elemento.

2. Inserción (Insert):

Mejor caso: El mejor caso ocurre cuando se inserta un nuevo elemento en el primer nivel de la lista, sin necesidad de promover o crear niveles adicionales. En este caso, la complejidad de tiempo sería O(1), ya que la inserción se realiza directamente en el primer nivel sin necesidad de ajustar niveles adicionales.

Caso medio: En promedio, la inserción en una Skip List tiene una complejidad de tiempo de O(log n). Esto se debe a que, en cada nivel, se recorren una cantidad logarítmica de nodos para encontrar el lugar correcto para insertar el nuevo elemento, y la probabilidad de promoción ayuda a mantener el equilibrio en la estructura.

Peor caso: El peor caso ocurre cuando el nuevo elemento se promociona a todos los niveles de la Skip List, lo que implica recorrer todos los niveles para insertar el elemento en cada nivel. En este caso, la complejidad de tiempo sería O(log n), similar al caso medio.

3. Eliminación (Delete):

Mejor caso: El mejor caso ocurre cuando el elemento a eliminar se encuentra en el primer nivel de la lista y no hay necesidad de ajustar los niveles superiores. En este caso, la complejidad de tiempo sería O(1), ya que la eliminación se realiza directamente en el primer nivel sin necesidad de ajustes adicionales.

Caso medio: En promedio, la eliminación en una Skip List tiene una complejidad de tiempo de O(log n). Esto se debe a que, en cada nivel, se recorren una cantidad logarítmica de nodos para encontrar el elemento a eliminar y realizar los ajustes necesarios en las referencias next.

Peor caso: El peor caso ocurre cuando el elemento a eliminar se encuentra en el nivel más bajo de la lista, lo que implica recorrer todos los niveles para encontrar el elemento y realizar los ajustes correspondientes en cada nivel. En este caso, la complejidad de tiempo sería O(log n), similar al caso medio.

En el mejor caso, las operaciones de búsqueda, inserción y eliminación pueden tener una complejidad de tiempo constante O(1) si los elementos se encuentran directamente en los niveles superiores. En el caso medio y peor, la complejidad de

tiempo es logarítmica O(log n) debido al recorrido de múltiples niveles. La probabilidad de promoción de niveles en la inserción ayuda a mantener el equilibrio y evitar casos extremadamente desfavorables.

Complejidad en el tiempo y complejidad especial

Complejidad temporal:

- Búsqueda (Search): En el caso promedio, la búsqueda tiene una complejidad temporal de O(log n), donde "n" es el número de elementos en la lista. Esto se debe a que en cada nivel se reduce a la mitad la cantidad de nodos a recorrer en promedio. En el peor caso, la complejidad también es O(log n), cuando el elemento buscado se encuentra en el nivel más bajo y se deben recorrer todos los niveles para llegar a él.
- Inserción (Insert): En promedio, la inserción tiene una complejidad temporal de O(log n), ya que se deben recorrer múltiples niveles para encontrar el lugar adecuado para insertar el nuevo elemento. En el peor caso, la complejidad también es O(log n) si el nuevo elemento se promociona a todos los niveles de la lista.
- Eliminación (Delete): Al igual que la búsqueda y la inserción, la eliminación tiene una complejidad temporal promedio de O(log n), ya que se deben recorrer múltiples niveles para encontrar el elemento a eliminar y realizar los ajustes necesarios en las referencias next. En el peor caso, la complejidad también es O(log n).

Complejidad espacial:

- La complejidad espacial de una Skip List es proporcional al número de elementos almacenados en ella. Requiere espacio adicional para almacenar los nodos y referencias adicionales en los niveles superiores de la lista. En general, la complejidad espacial es O(n), donde "n" es el número de elementos en la lista.

Es importante tener en cuenta que estos análisis de complejidad son en promedio y en el peor caso, ya que las operaciones en una Skip List dependen de la probabilidad de promoción de niveles y la distribución de los elementos. La elección de una estructura de datos debe considerar tanto la eficiencia temporal como espacial, así como las características y requisitos específicos del problema a resolver.

5. BUSQUEDA DE INTERPOLACION.

En que consiste la búsqueda de interpolación

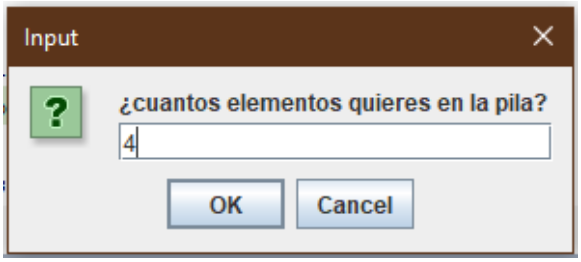
La búsqueda por interpolación es un algoritmo de búsqueda utilizado para encontrar un elemento en una colección de datos ordenada. A diferencia de otros algoritmos de búsqueda, como la búsqueda binaria, la búsqueda por interpolación utiliza una estimación lineal basada en la distribución de los valores en la colección para aproximar la posición del elemento buscado.

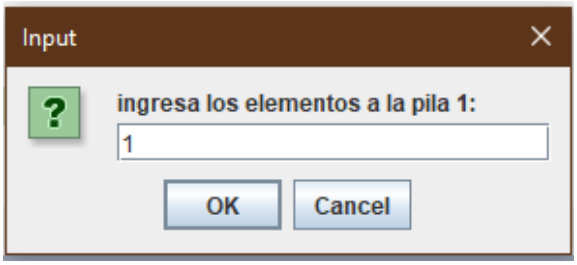
El algoritmo de búsqueda por interpolación se basa en la interpolación lineal, que es una técnica para estimar un valor desconocido entre dos valores conocidos. La idea principal de la búsqueda por interpolación es estimar la posición del elemento buscado utilizando una fórmula que toma en cuenta la posición relativa de los valores extremos y el valor buscado.

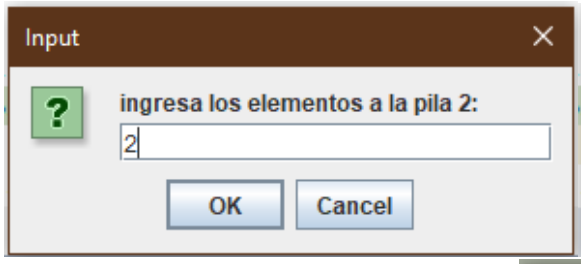
Ejemplos de implementaciones del algoritmo en las estructuras de datos

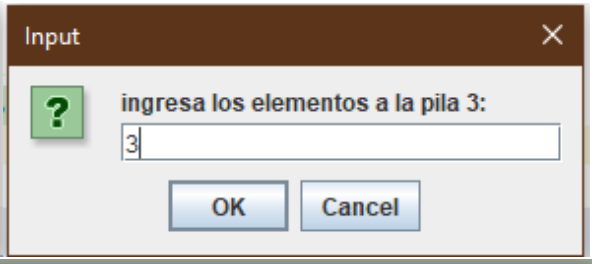
```java
import java.util.Stack;
import EntradaSalida.Tools;
public class busquedaDeInterpolacion {

    public static int  BusquedaDeInterpolacion(Stack<Integer> pila, int target) {
        if (pila == null || pila.isEmpty()) {
            return -1;
        }

        int left = 0;
        int right = pila.size() - 1;
        int[] A = new int[pila.size()];

        for (int i = pila.size() - 1; i >= 0; i--) {
            A[i] = pila.pop();
        }

        while (A[right] != A[left] && target >= A[left] && target <= A[right]) {
            int mid = left + ((target - A[left]) * (right - left) / (A[right] - A[left]));
            if (target == A[mid]) {
                return mid;
            } else if (target < A[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

```

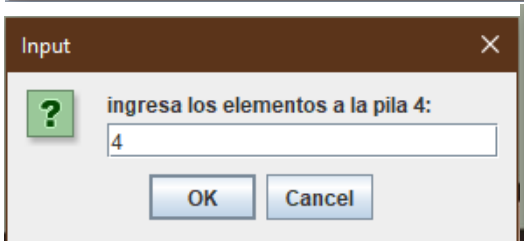
```java
            if (target == A[left]) {
                return left;
            }

            return -1;
        }

        public static void main(String[] args) {
            Stack<Integer> pila = new Stack<>();

            int n = Tools.leeEntero("¿cuantos elementos quieres en la pila? ");

            for (int i = 0; i < n; i++) {
                int element = Tools.leeEntero("ingresa los elementos a la pila " + (i + 1) + ":");
                pila.push(element);
            }

            int llave = Tools.leeEntero("igresa que numero quieres buscar dentro de la pila:");
            int index = BusquedaDeInterpolacion(pila, llave);

            if (index != -1) {
                Tools.imprimeMSJ("NUMERO ENCONTRADO EN EL ESPACIO: " + index);
            } else {
               Tools.imprimeMSJ("NUMERO NO EN CONTRADO EN LA PILA!!!!!!!!");
            }
        }
    }
```

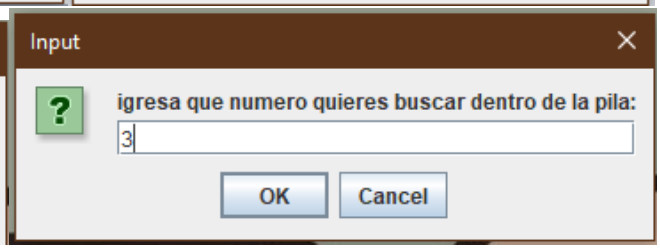


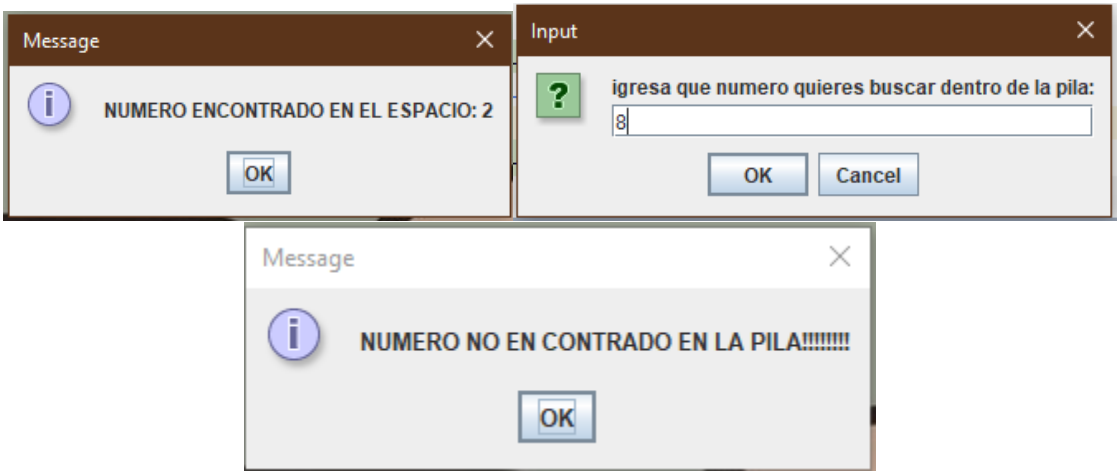












- Explicación del Código:

La función "BusquedaDeInterpolacion" recibe una pila de números enteros (llamada "pila") y un número objetivo (llamado "target") como entrada, y devuelve la posición en la que se encuentra ese número objetivo dentro de la pila. Si el número objetivo no se encuentra en la pila, se devuelve -1.

Primero, la función realiza algunas verificaciones iniciales. Si la pila no existe o está vacía, se devuelve -1 porque no hay elementos para buscar.

A continuación, se crean dos punteros, llamados "left" y "right", que representan los índices izquierdo y derecho de la pila, respectivamente. También se crea un arreglo llamado "A" con el mismo tamaño que la pila para almacenar los elementos de la pila.

Luego, se recorre la pila en orden inverso utilizando un bucle "for" y el método "pop()" de la clase Stack. En cada iteración, se extrae un elemento de la pila y se guarda en el arreglo A. Esto se hace para que los elementos en A estén en el mismo orden que en la pila original.

Una vez que se han extraído todos los elementos de la pila y se han almacenado en A, comienza la búsqueda por interpolación. Se utiliza un bucle "while" que se ejecuta mientras el valor más a la derecha de A sea diferente al valor más a la izquierda, y el valor objetivo (target) se encuentre dentro del rango de valores en A.

En cada iteración del bucle, se calcula el índice medio (llamado "mid") utilizando una fórmula de interpolación. Esta fórmula tiene en cuenta la posición relativa del valor objetivo y los valores extremos de A para estimar su posición aproximada.

Se comparan el valor objetivo y el valor en el índice medio. Si son iguales, se devuelve el índice medio, indicando que se ha encontrado el valor objetivo.

Si el valor objetivo es menor que el valor en el índice medio, se actualiza el puntero derecho (right) para restringir la búsqueda a la mitad izquierda de A.

Si el valor objetivo es mayor que el valor en el índice medio, se actualiza el puntero izquierdo (left) para restringir la búsqueda a la mitad derecha de A.

Si el bucle "while" termina sin encontrar una coincidencia exacta del valor objetivo en A, se realiza una verificación final. Si el valor objetivo es igual al valor en la posición left de A, se devuelve el índice left. De lo contrario, se devuelve -1 para indicar que el valor objetivo no se encontró en la pila.

Y el main lo que hace es que solicita al usuario ingresar el número de elementos que desea tener en la pila, seguido de los elementos individuales. Luego, se le pide al usuario que ingrese un número para buscar dentro de la pila. El código utiliza el algoritmo de búsqueda por interpolación (BusquedaDeInterpolacion) para buscar la llave en la pila y muestra un mensaje indicando si se encontró o no.

Análisis de eficiencia

El análisis de eficiencia de la búsqueda de interpolación se realiza en términos de tiempo de ejecución y complejidad temporal. Aquí está el análisis de eficiencia de la búsqueda de interpolación:

- Tiempo de ejecución promedio: O(log log n) en promedio y O(n) en el peor caso.
    - En promedio, la búsqueda de interpolación tiene un tiempo de ejecución de O(log log n), donde n es el tamaño del arreglo.
    - Esto se debe a que la búsqueda de interpolación utiliza una interpolación lineal para estimar la posición del elemento objetivo en el arreglo.
    - La interpolación lineal es más precisa cuando los elementos del arreglo están uniformemente distribuidos.
    - En promedio, la búsqueda de interpolación reduce el rango de búsqueda a aproximadamente la mitad en cada iteración.
    - Sin embargo, en el peor caso, cuando los elementos del arreglo no están uniformemente distribuidos, la búsqueda de interpolación puede requerir O(n) comparaciones, lo que se asemeja a una búsqueda lineal.
- Peor caso: O(n)
    - El peor caso ocurre cuando los elementos del arreglo no están uniformemente distribuidos y la interpolación lineal no es efectiva para reducir el rango de búsqueda.

- En este caso, la búsqueda de interpolación se degrada a una búsqueda lineal y requiere realizar comparaciones con cada elemento del arreglo.
- Como resultado, el tiempo de ejecución en el peor caso es lineal, O(n), donde n es el tamaño del arreglo.

➢ Espacio auxiliar: O (1)

- La búsqueda de interpolación no requiere espacio adicional en la memoria, aparte de las variables locales utilizadas en la función.
- Por lo tanto, el espacio auxiliar requerido es constante, O(1).

Es importante tener en cuenta que estos análisis de eficiencia asumen que el arreglo está ordenado de manera ascendente y que los elementos están distribuidos de manera uniforme. Si estas condiciones no se cumplen, el rendimiento de la búsqueda de interpolación puede verse afectado negativamente.

Análisis de casos: mejor de los casos, caso medio, y peor de los casos

El análisis de casos de la búsqueda de interpolación se realiza considerando el mejor de los casos, el caso medio y el peor de los casos:

➢ Mejor de los casos:

- El mejor caso ocurre cuando el elemento buscado está presente en la posición media del arreglo o cerca de ella.
- En este caso, la interpolación lineal es altamente efectiva para estimar la posición del elemento objetivo en el arreglo.
- La búsqueda de interpolación reduce rápidamente el rango de búsqueda en cada iteración y encuentra el elemento objetivo en pocas comparaciones.
- En términos de tiempo de ejecución, el mejor caso se aproxima a O(1), ya que se requiere un número constante de comparaciones.

➢ Caso medio:

- El caso medio ocurre cuando el elemento buscado no está en la posición media del arreglo, pero los elementos están uniformemente distribuidos.
- La búsqueda de interpolación utiliza la interpolación lineal para estimar la posición del elemento objetivo.

- En promedio, la búsqueda de interpolación reduce el rango de búsqueda a aproximadamente la mitad en cada iteración.
- Por lo tanto, en el caso medio, el número de comparaciones requeridas es logarítmico en el tamaño del arreglo.
- En términos de tiempo de ejecución y complejidad temporal, el caso medio se aproxima a O(log log n), donde n es el tamaño del arreglo.

➢ Peor de los casos:

- El peor caso ocurre cuando los elementos del arreglo no están uniformemente distribuidos y la interpolación lineal no es efectiva para reducir el rango de búsqueda.
- Esto puede ocurrir, por ejemplo, cuando los elementos están agrupados en un extremo del arreglo o están dispuestos en orden descendente.
- En el peor caso, la búsqueda de interpolación puede requerir un número lineal de comparaciones con todos los elementos del arreglo.
- Como resultado, en el peor caso, el tiempo de ejecución y la complejidad temporal son O(n), donde n es el tamaño del arreglo.

Es importante tener en cuenta que el rendimiento de la búsqueda de interpolación puede variar según la distribución de los elementos en el arreglo. Si los elementos no están uniformemente distribuidos, la búsqueda de interpolación puede no ser eficiente y puede degradarse a una búsqueda lineal. Por lo tanto, es recomendable utilizar la búsqueda de interpolación en conjunción con arreglos ordenados y uniformemente distribuidos para obtener mejores resultados.

Complejidad en el tiempo y complejidad especial

Es importante tener en cuenta que la búsqueda por interpolación puede tener una complejidad en el tiempo superior a otros algoritmos de búsqueda, como la búsqueda binaria, que tiene una complejidad O(log n) en todos los casos. La eficiencia de la búsqueda por interpolación depende en gran medida de la distribución de los elementos en la pila y la posición del valor objetivo.

En cuanto a la complejidad espacial, la búsqueda por interpolación no requiere espacio adicional más allá del arreglo o estructura de datos utilizada para almacenar los elementos. Por lo tanto, la complejidad espacial es O(1), es decir, constante, ya que no se requiere un espacio adicional proporcional al tamaño de los datos.

6. BUSQUEDA EXPONENCIAL

- En que consiste la búsqueda exponencial

La búsqueda exponencial, también conocida como búsqueda duplicada o búsqueda con los dedos, es un algoritmo de búsqueda utilizado para encontrar un elemento en una colección de datos ordenada. A diferencia de otros algoritmos de búsqueda, como la búsqueda lineal o la búsqueda a binaria, la búsqueda exponencial no examina los elementos en orden secuencial o divide el espacio de búsqueda a la mitad. En cambio, realiza saltos exponenciales para acercarse al elemento buscado de manera más rápida.

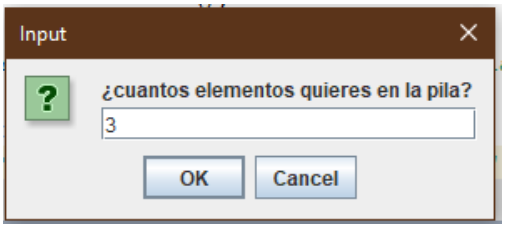
La idea principal detrás de la búsqueda exponencial es realizar saltos exponenciales para acercarse rápidamente al rango donde se espera encontrar el elemento buscado. Luego, se realiza una búsqueda binaria en ese rango más pequeño para encontrar el elemento de manera más precisa.

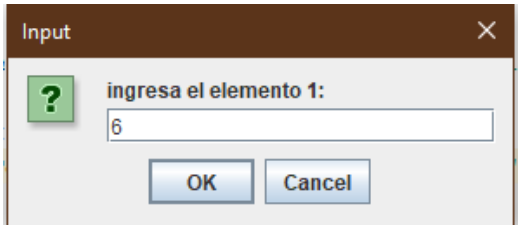
La búsqueda exponencial puede ser eficiente cuando el tamaño de la colección de datos es desconocido y no se dispone de información sobre su tamaño. Sin embargo, su eficiencia puede verse afectada si el elemento buscado se encuentra cerca del inicio de la colección, ya que se realizarán saltos exponenciales significativos antes de realizar la búsqueda binaria.

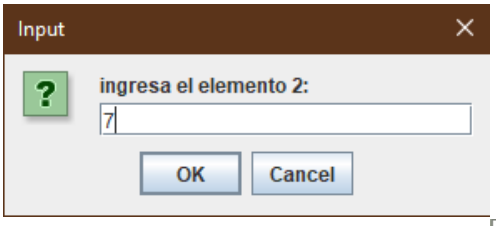
- Ejemplos de implementaciones de algoritmos en las estructuras de datos

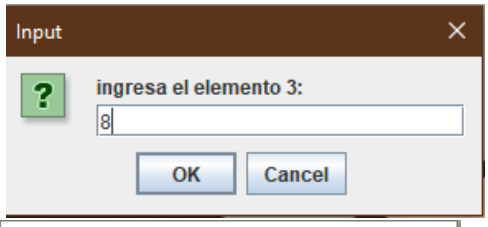
```java
import EntradaSalida.Tools;
import java.util.Stack;
public class BusquedaExponencial {

    private static int BusquedaBinaria(int[] A, int left, int right, int x) {
        if (left > right) {
            return -1;
        }
        int mid = (left + right) / 2;
        if (x == A[mid]) {
            return mid;
        } else if (x < A[mid]) {
            return BusquedaBinaria(A, left, mid - 1, x);
        } else {
            return BusquedaBinaria(A, mid + 1, right, x);
        }
    }

    public static int BusquedaExponencial(Stack<Integer> pila, int x) {
        if (pila == null || pila.isEmpty()) {
            return -1;
        }

        int[] A = new int[pila.size()];
        int index = pila.size() - 1;
        while (!pila.isEmpty()) {
            A[index] = pila.pop();
            index--;
```

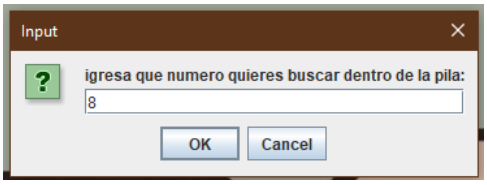
```
30          }
31
32          int bound = 1;
33
34          while (bound < A.length && A[bound] < x) {
35              bound *= 2;
36          }
37          return BusquedaBinaria(A, bound / 2, Math.min(bound, A.length - 1), x);
38      }
39
40      public static void main(String[] args) {
41          Stack<Integer> pila = new Stack<>();
42
43          int n = Tools.leeEntero("¿cuantos elementos quieres en la pila? ");
44
45          for (int i = 0; i < n; i++) {
46              int element = Tools.leeEntero("ingresa los elementos a la pila " + (i + 1) + ":");
47              pila.push(element);
48          }
49
50          int llave = Tools.leeEntero("igresa que numero quieres buscar dentro de la pila:");
51          int index = BusquedaExponencial(pila, llave);
52
53          if (index != -1) {
54              Tools.imprimeMSJ("NUMERO ENCONTRADO EN EL ESPACIO: " + index);
55          } else {
56             Tools.imprimeMSJ("NUMERO NO EN CONTRADO EN LA PILA!!!!!!!!");
57          }
58      }
59  }
```

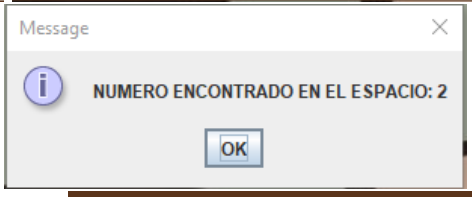


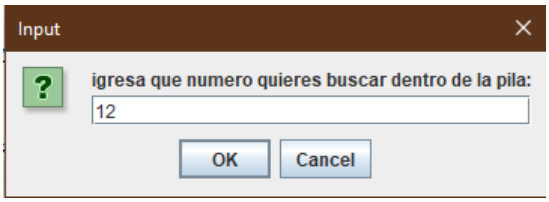


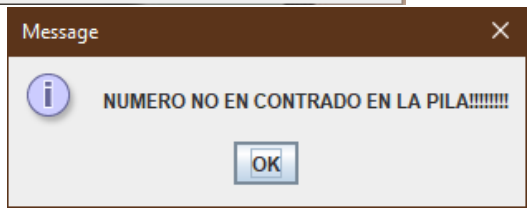












- Explicación del Código:

La función BusquedaBinaria es una implementación recursiva de la búsqueda binaria. Recibe como parámetros el arreglo A, los índices izquierdo (left) y derecho (right) que definen el rango de búsqueda, y el elemento objetivo x. La función divide el rango a la mitad y compara el elemento en el medio con el objetivo. Si son iguales, devuelve el índice medio. Si el objetivo es menor, realiza una llamada recursiva a BusquedaBinaria en el subarreglo izquierdo. Si el objetivo es mayor, realiza una llamada recursiva en el subarreglo derecho. Este proceso se repite hasta que se encuentra el objetivo o se determina que no está presente en el arreglo.

La función BusquedaExponencial implementa la búsqueda exponencial utilizando una pila. Recibe como parámetros la pila pila y el elemento objetivo x. Primero, crea un arreglo A del tamaño de la pila y guarda los elementos de la pila en orden inverso en el arreglo. Luego, utiliza un bucle while para incrementar exponencialmente el límite bound hasta que el límite sea mayor que el tamaño del arreglo o el elemento en bound sea mayor que x. Después, llama a la función BusquedaBinaria pasando el arreglo A, los índices de inicio y final del subarreglo y el elemento objetivo x. El resultado de la búsqueda se devuelve como el índice del elemento encontrado o -1 si no se encuentra.

En general, este código realiza una búsqueda exponencial en un arreglo utilizando recursión y una pila para almacenar los elementos. Primero, invierte el orden de los elementos de la pila y luego realiza una búsqueda binaria en el subarreglo correspondiente utilizando los límites determinados por la búsqueda exponencial.

Y el main lo que hace es que solicita al usuario ingresar el número de elementos que desea tener en la pila, seguido de los elementos individuales. Luego, se le pide al usuario que ingrese un número para buscar dentro de la pila. El código utiliza el algoritmo de búsqueda por interpolación (BusquedaExponencial) para buscar la llave en la pila y muestra un mensaje indicando si se encontró o no.

Análisis de eficiencia

El análisis de eficiencia de la búsqueda exponencial se realiza en términos de tiempo de ejecución y complejidad temporal. Aquí está el análisis de eficiencia de la búsqueda exponencial:

- Tiempo de ejecución promedio: O(log n)
    - El tiempo de ejecución promedio de la búsqueda exponencial es logarítmico en el tamaño del arreglo.
    - Esto se debe a que la búsqueda exponencial divide repetidamente el tamaño del problema en la mitad hasta encontrar un rango en el que el elemento buscado pueda estar presente.

- La búsqueda se realiza incrementando exponencialmente los límites del rango, lo que reduce la cantidad de iteraciones necesarias para encontrar el rango adecuado.

➢ Peor caso: O(log n)

- En el peor caso, la búsqueda exponencial también tiene una complejidad de O(log n).
- Esto ocurre cuando el elemento buscado se encuentra en la última posición del arreglo o no está presente en el arreglo.
- En cada iteración, el tamaño del rango de búsqueda se duplica, lo que permite una búsqueda más rápida en comparación con la búsqueda binaria estándar.
- Sin embargo, en el peor caso, la búsqueda exponencial termina realizando una búsqueda binaria en el rango determinado, lo que resulta en una complejidad logarítmica.

➢ Espacio auxiliar: O(n)

- En el caso de la implementación utilizando una pila, se requiere espacio adicional para almacenar el arreglo **A** que contiene los elementos extraídos de la pila en orden inverso.
- El tamaño del arreglo **A** es igual al tamaño de la pila.
- Por lo tanto, el espacio auxiliar requerido es proporcional al tamaño de la entrada, es decir, O(n).

Es importante tener en cuenta que estos análisis de eficiencia asumen que el arreglo está ordenado. Si el arreglo no está ordenado, se requeriría un paso adicional para ordenar el arreglo, lo que aumentaría la complejidad del algoritmo.

Análisis de casos: mejor de los casos, caso medio, y peor de los casos

El análisis de casos de la búsqueda exponencial se realiza considerando el mejor de los casos, el caso medio y el peor de los casos:

➢ Mejor de los casos:

- El mejor caso ocurre cuando el elemento buscado está presente en la primera posición del arreglo.
- En este caso, la búsqueda exponencial encuentra el elemento de inmediato en la primera iteración y devuelve su índice.

- El tiempo de ejecución es constante, O(1), ya que solo se realiza una comparación para verificar si el elemento en la posición inicial es igual al objetivo.
- En términos de complejidad temporal, el mejor caso es O(1).

➢ Caso medio:

- El caso medio ocurre cuando el elemento buscado se encuentra en una posición aleatoria dentro del arreglo, pero no está en la primera ni en la última posición.
- La búsqueda exponencial divide repetidamente el tamaño del problema en la mitad hasta encontrar un rango en el que el elemento buscado pueda estar presente.
- En promedio, la búsqueda exponencial requiere un número de iteraciones proporcional a logaritmo base 2 del tamaño del arreglo.
- Por lo tanto, en términos de tiempo de ejecución y complejidad temporal, el caso medio es O(log n), donde n es el tamaño del arreglo.

➢ Peor de los casos:

- El peor caso ocurre cuando el elemento buscado está en la última posición del arreglo o no está presente en el arreglo.
- En la búsqueda exponencial, el límite del rango se incrementa exponencialmente hasta encontrar un rango en el que el elemento buscado pueda estar presente.
- En el peor caso, el límite del rango superará el tamaño del arreglo antes de encontrar el rango adecuado.
- Una vez que se encuentra el rango adecuado, se realiza una búsqueda binaria en ese rango.
- En total, se requieren logaritmo base 2 del tamaño del arreglo más el número de comparaciones realizadas en la búsqueda binaria.
- Por lo tanto, en términos de tiempo de ejecución y complejidad temporal, el peor caso también es O(log n), donde n es el tamaño del arreglo.

Es importante tener en cuenta que estos análisis de casos asumen que el arreglo está ordenado de manera ascendente. Además, si el arreglo no está ordenado, se requeriría un paso adicional para ordenar el arreglo, lo que aumentaría la complejidad del algoritmo.

Complejidad en el tiempo y complejidad especial

- Complejidad en el tiempo:
  - Mejor caso: O(1) En el mejor caso, el elemento buscado se encuentra en la primera posición del arreglo, por lo que la búsqueda exponencial devuelve el resultado en un solo paso.
  - Caso promedio: O(log n) En promedio, la búsqueda exponencial tiene una complejidad de tiempo logarítmica. Esto se debe a que en cada iteración, el rango de búsqueda se duplica, lo que resulta en una reducción exponencial del espacio de búsqueda.
  - Peor caso: O(log n) En el peor caso, la búsqueda exponencial tiene una complejidad de tiempo logarítmica. Esto ocurre cuando el elemento buscado se encuentra cerca del final del arreglo, lo que requiere más iteraciones para encontrarlo.
- Complejidad espacial:
  - La complejidad espacial de la búsqueda exponencial en Java es O(1) porque no requiere espacio adicional que crezca con el tamaño del arreglo. Solo se necesitan algunas variables locales para realizar las operaciones de búsqueda.

La búsqueda exponencial en Java requiere que el arreglo esté ordenado de manera ascendente para que funcione correctamente. Además, su rendimiento depende de la distribución de los datos en el arreglo. Si los elementos están uniformemente distribuidos, la búsqueda exponencial es más eficiente. Sin embargo, si los elementos están agrupados o desordenados, puede requerir más iteraciones y tener un rendimiento inferior.

7. BUSQUEDA DE FIBONACCI.

- En que consiste la búsqueda exponencial

El método de Fibonacci es una secuencia matemática en la que cada número es la suma de los dos números anteriores. La secuencia comienza con 0 y 1, por lo que los primeros números de Fibonacci son: 0, 1, 1, 2, 3, 5, 8, 13, 21, y así sucesivamente.

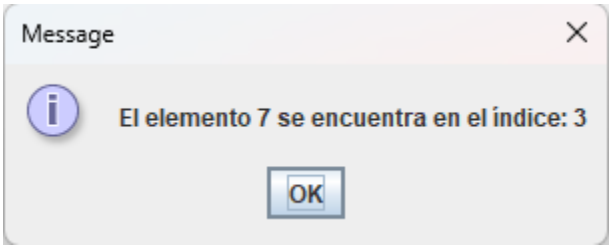
Este método se puede utilizar en la búsqueda de estructuras de datos, como por ejemplo en la búsqueda binaria. La búsqueda binaria es un algoritmo eficiente para buscar un elemento en una lista ordenada. Consiste en dividir repetidamente la lista en dos mitades y determinar en cuál de las dos mitades se encuentra el elemento buscado.

La idea de utilizar la secuencia de Fibonacci en la búsqueda binaria es determinar los índices de división de la lista utilizando números de Fibonacci. En lugar de dividir la lista en dos mitades exactamente en el centro, se utiliza un índice de división que está más cerca de la proporción áurea, que es aproximadamente 0.618.

- Ejemplos de implementaciones del algoritmo en las estructuras de datos

```java
package Ejemplos;
import EntradaSalida.Tools;

public class   MetodoFibonacci {
   public static int search(int[] arr, int target) {
      int fib2 = 0; // Valor anterior a fib1
      int fib1 = 1; // Valor actual de la secuencia de Fibonacci
      int fib = fib1 + fib2; // Siguiente valor de la secuencia de Fibonacci

      while (fib < arr.length) {
         fib2 = fib1;
         fib1 = fib;
         fib = fib1 + fib2;
      }

      int offset = -1;

      while (fib > 1) {
         int i = Math.min(offset + fib2, arr.length - 1);

         if (arr[i] < target) {
            fib = fib1;
            fib1 = fib2;
            fib2 = fib - fib1;
            offset = i;
         } else if (arr[i] > target) {
            fib = fib2;
            fib1 = fib1 - fib2;
            fib2 = fib - fib1;
         } else {
```

```
31              return i;
32            }
33          }
34
35          if (fib1 == 1 && arr[offset + 1] == target) {
36            return offset + 1;
37          }
38
39          return -1;
40      }
41
42      public static void main(String[] args) {
43          int[] arr = {1, 3, 5, 7, 9, 11, 13};
44          int target = 7;
45          int index = search(arr, target);
46          Tools.ImprimeMensaje("El elemento " + target + " se encuentra en el índice: " + index );
47      }
48  }
49
```



- Explicación del código.

1. Primero, definimos una clase llamada FibonacciSearch.
2. Dentro de la clase, tenemos el método search que realiza la búsqueda utilizando el método de Fibonacci. Recibe dos parámetros: arr, que es el arreglo en el que se realizará la búsqueda, y target, que es el elemento que buscamos.
3. Declaramos tres variables enteras: fib2, fib1 y fib. Estas variables representan los números de Fibonacci utilizados en el algoritmo. Inicialmente, fib2 es 0, fib1 es 1 y fib es la suma de fib2 y fib1.
4. Luego, entramos en un bucle while que se ejecuta mientras fib sea menor que la longitud del arreglo arr. El propósito de este bucle es encontrar el número de Fibonacci más cercano o igual a la longitud del arreglo. En cada iteración, actualizamos los valores de fib2, fib1 y fib para avanzar en la secuencia de Fibonacci.

5. Después del bucle anterior, declaramos la variable offset e inicializamos su valor en -1. Esta variable representa el índice de desplazamiento en la búsqueda.
6. A continuación, ingresamos en otro bucle while que se ejecuta mientras fib sea mayor que 1. Este bucle realiza la búsqueda propiamente dicha utilizando la secuencia de Fibonacci.
7. En cada iteración, calculamos el índice i como la suma de offset y fib2, o el índice más cercano al valor de búsqueda.
8. Luego, comparamos el elemento en el índice i del arreglo arr con el elemento buscado (target). Si el elemento es menor que el objetivo, actualizamos los valores de fib, fib1, fib2 y offset para continuar buscando en la mitad superior del arreglo. Si el elemento es mayor que el objetivo, actualizamos los valores de fib, fib1 y fib2 para continuar buscando en la mitad inferior del arreglo. Si el elemento es igual al objetivo, encontramos el elemento y devolvemos el índice i.
9. Finalmente, después del bucle de búsqueda, verificamos si fib1 es igual a 1 y si el elemento en offset + 1 es igual al objetivo. Si se cumple esta condición, significa que el elemento buscado se encuentra en el siguiente índice después de offset, y lo devolvemos como resultado.
10. En el método main, creamos un arreglo arr de ejemplo con valores ordenados y un objetivo target que queremos buscar. Luego, llamamos al método search pasando el arreglo y el objetivo, y almacenamos el resultado en la variable index.
11. Finalmente, imprimimos el resultado en la consola, indicando el elemento buscado y el índice donde se encuentra en el arreglo.

Este código implementa la búsqueda utilizando el método de Fibonacci en una estructura de datos en Java, específicamente en el algoritmo de búsqueda binaria. Espero que esta explicación te haya sido útil. Si tienes alguna otra pregunta, no dudes en preguntar.

- Análisis de eficiencia.

El algoritmo de búsqueda de Fibonacci se basa en la búsqueda binaria, pero en lugar de dividir el arreglo en dos mitades exactamente en el centro, utiliza una proporción áurea aproximada (0.618) para determinar el índice de división. Esto puede ayudar a reducir el número de comparaciones y mejorar ligeramente el rendimiento en comparación con la búsqueda binaria tradicional.

Análisis de casos: mejor de los casos, caso medio, y peor de los casos

Mejor de los casos:

- El mejor de los casos ocurre cuando el elemento buscado se encuentra en la primera comparación del bucle de búsqueda, es decir, en el primer intento.
- En este caso, el algoritmo encontrará el elemento de manera muy eficiente y se ejecutará en el mínimo tiempo posible.
- La complejidad temporal en el mejor de los casos es O(1), ya que se encuentra el elemento de inmediato sin realizar iteraciones adicionales.

Caso medio:

- En el caso medio, el elemento buscado se encuentra en una posición aleatoria dentro del arreglo.
- El algoritmo de búsqueda de Fibonacci divide el arreglo en rangos más grandes al principio y luego reduce el rango en cada iteración.
- Aunque puede haber una mejora en comparación con la búsqueda binaria tradicional, el número de comparaciones y el tiempo de ejecución en el caso medio seguirán siendo bastante eficientes.
- La complejidad temporal en el caso medio es O(log N), donde N es la longitud del arreglo.

Peor de los casos:

- El peor de los casos ocurre cuando el elemento buscado no está presente en el arreglo y se debe realizar una búsqueda exhaustiva hasta el final del rango de búsqueda.
- En este caso, el algoritmo de búsqueda de Fibonacci se comporta similar a la búsqueda binaria tradicional, ya que debe realizar una cantidad significativa de comparaciones antes de determinar que el elemento no está presente.
- La complejidad temporal en el peor de los casos es O(log N), donde N es la longitud del arreglo.

En términos de eficiencia espacial, el algoritmo de búsqueda de Fibonacci tiene una complejidad espacial de O(1), es decir, constante, ya que no requiere estructuras de datos adicionales para almacenar información.

Complejidad en el tiempo y complejidad especial

La eficiencia del algoritmo de búsqueda de Fibonacci se puede analizar en términos de su complejidad temporal y espacial.

Complejidad temporal:

- El bucle inicial que encuentra el número de Fibonacci más cercano a la longitud del arreglo tiene una complejidad de O(log N), donde N es la longitud

del arreglo. Esto se debe a que la secuencia de Fibonacci crece exponencialmente y se acerca a la longitud del arreglo de manera logarítmica.

- El segundo bucle de búsqueda tiene una complejidad de O(log N). En cada iteración, se reduce a aproximadamente la mitad el rango de búsqueda, siguiendo la secuencia de Fibonacci. El número de iteraciones en este bucle es proporcional a la longitud del número de Fibonacci encontrado previamente.
- En general, el algoritmo de búsqueda de Fibonacci tiene una complejidad temporal de O(log N).

Complejidad espacial:

- El algoritmo de búsqueda de Fibonacci no requiere espacio adicional significativo más allá del arreglo de entrada y algunas variables enteras para realizar los cálculos.
- La complejidad espacial es de O(1), es decir, constante, ya que no se requiere una estructura de datos adicional para almacenar información.

En resumen, el algoritmo de búsqueda de Fibonacci ofrece una mejora leve en la eficiencia en comparación con la búsqueda binaria tradicional, reduciendo el número de comparaciones. Sin embargo, su complejidad temporal sigue siendo O(log N), lo cual es muy eficiente para la búsqueda en arreglos ordenados.

Es importante tener en cuenta que la eficiencia del algoritmo puede variar en función de varios factores, como el tamaño del arreglo, la distribución de los elementos y el valor buscado. En algunos casos, la búsqueda binaria tradicional puede ser igualmente eficiente o incluso más rápida que la búsqueda de Fibonacci.

En general, el algoritmo de búsqueda de Fibonacci es una alternativa interesante para la búsqueda en arreglos ordenados, pero se recomienda realizar pruebas y comparaciones con otros algoritmos para determinar cuál es el más adecuado para un caso específico.

Bibliografía

Charras, C. (1997). *IGM*. Obtenido de Knuth-Morris-Pratt algorithm: http://www-igm.univ-mlv.fr/~lecroq/string/node8.html

F, E. (18 de Diciembre de 2017). *Medium*. Obtenido de medium.com: https://medium.com/@Emmitta/b%C3%BAsqueda-binaria-c6187323cd72

Franco, E. (s.f.). *Análisis de algoritmos.* Escuela Superior de Cómputo: Instituto Politécnico Nacional.

Gil, D. (26 de April de 2020). *Medium*. Obtenido de medium.com: https://medium.com/@daniel.patrick.gil/b%C3%BAsqueda-binaria-dec386ad8525

Jindal, H. (11 de Marzo de 2021). *DelftStack*. Obtenido de www.delftstack.com: https://www.delftstack.com/es/tutorial/algorithm/linear-search/

Knuth, D., Morris, J., & Pratt, V. (1977). Fast pattern matching in strings. *Journal on Computing*, 323-350.

Page, B. (11 de Julio de 2019). *EDteam*. Obtenido de ed.team: https://ed.team/comunidad/ventajas-y-desventajas-de-la-busqueda-lineal

Rivera, J. (31 de Marzo de 2021). *freeCodeCamp*. Obtenido de www.freecodecamp.org: https://www.freecodecamp.org/espanol/news/introduccion-a-la-complejidad-temporal-de-los-algoritmos/

InstintoProgramador. (14 de 06 de 2019). *InstintoProgramador*. Obtenido de Algoritmos de búsqueda en Java: https://www.instintoprogramador.com.mx/2019/06/algoritmos-de-busqueda-en-java_14.html

Jindal, H. (30 de 01 de 2023). *delftstack*. Obtenido de Búsqueda de interpolación: https://www.delftstack.com/es/tutorial/algorithm/interpolation-search/

P, R. (05 de 01 de 2023). *Búsqueda de interpolación*. Obtenido de techie delight: https://www.techiedelight.com/es/interpolation-search/#:~:text=La%20b%C3%BAsqueda%20por%20interpolaci%C3%B3n%20es,ordenan%20las%20entradas%20del%20libro..

Zohonero Martínez, I. y Joyanes Aguilar, L. (2008). Estructuras de datos en Java. Madrid etc, Spain: McGraw-Hill España. Recuperado de https://elibro.net/es/ereader/itorizababiblio/50117?page=207.

PuntoComNoEsUnLenguaje. (2012, noviembre). Fibonacci en Java [Blog post]. Recuperado de http://puntocomnoesunlenguaje.blogspot.com/2012/11/fibonacci-en-java.html

Barcelona Geeks. (s.f.). Método skip() del Scanner en Java con ejemplos [Página web]. Recuperado de https://barcelonageeks.com/metodo-skip-del-escaner-en-java-con-ejemplos/

Lin, A., et al. (2 de julio de 2019). Algoritmo de búsqueda interna. WikiJournal of Science, 2(1), 5. doi:10.15347/WJS/2019.005.