INSTITUTO TECNOLÓGICO NACIONAL DE MEXICO CAMPUS ORIZABA

Materia:

Estructura De Datos

Maestra

Martínez Castillo María Jacinta

Integrantes:

Bravo Rosas Yamileth
Crescencio Rico José Armando
Gómez López Alberto Yeriel
Hernández Angón Diego
Morales Valdeolivar Gael Octavio

Grupo:

3PM – 4PM HRS

Clave:

3a3A

Especialidad:

Ing. informática

Fecha De Entrega

22/05/2023

REPORTE DE PRACTICAS

UNIDAD 6

Introducción

Una de las funciones que con mayor frecuencia se utiliza en los sistemas de información, es el de las consultas a los datos, se hace necesario utilizar algoritmos, que permitan realizar búsquedas de forma rápida y eficiente. La búsqueda, se puede decir que es la acción de recuperar datos o información, siendo una de las actividades que más aplicaciones tiene en los sistemas de información. Más formalmente se puede definir como “La operación de búsqueda sobre una estructura de datos es aquella que permite localizar un nodo en particular si es que éste existe”

Competencia especifica.

Específica**(s):**

Conoce, comprende y aplica los algoritmos de búsqueda para el uso adecuado en el desarrollo de aplicaciones que permita solucionar problemas del entorno.

Genéricas:

- La comprensión y manipulación de ideas y pensamientos.
- Metodologías para solución de problemas, organización del tiempo y para el aprendizaje
- Habilidad en el manejo de equipo de cómputo
- Capacidad para trabajar en equipo.
- Capacidad de aplicar los conocimientos en la práctica.

Marco teórico

Búsqueda Lineal

Consiste en recorrer y examinar cada uno de los elementos del array hasta encontrar el o los elementos buscados, o hasta que se han mirado todos los elementos del array. Este es el método de búsqueda más lento, pero si nuestra

información se encuentra completamente desordenada es el único que nos podrá ayudar a encontrar el dato que buscamos.

Para la búsqueda lineal, se consideran los siguientes pasos a seguir:

1. Toma el elemento que deseas buscar.
2. Comienza desde el primer elemento de la lista y compáralo con el elemento buscado.
3. Si el elemento coincide con el buscado, se ha encontrado el resultado y el proceso termina.
4. Si el elemento no coincide, pasa al siguiente elemento de la lista y repite el paso 3.
5. Continúa este proceso hasta que encuentres el elemento buscado o hayas recorrido toda la lista.
6. Si llegas al final de la lista sin encontrar el elemento, se considera que el elemento no está presente.

Ejemplos de búsqueda Lineal:

```
public class busquedas {

    public static boolean secuencialLineal(int a[], int valor){
        boolean existe=false;

        for (int i = 0;i<a.length;i++){
            if(valor==a[i])
                existe = true;
        }

        return existe;

    }
```

```
        sel=boton(menu);
        switch(sel){
            case "Secuencial Lineal":
                int SL[] = {1,2,3,4,5,6,7,8,9,10};
                toolsList.imprimePantalla("Busqueda de un numero del 1 al 10.");
                if(busquedas.secuencialLineal(SL, toolsList.leerByte("Numero a buscar:")))
                    toolsList.imprimePantalla("El elemento existe");
                else
                    toolsList.imprimePantalla("El elemento no existe");
                break;
```

La búsqueda lineal tiene una eficiencia de tiempo de O(n), donde "n" es el tamaño de la lista o arreglo en el que se está realizando la búsqueda. Esto significa que el tiempo de ejecución aumenta linealmente con el tamaño de la lista.

La búsqueda lineal es adecuada cuando la lista o arreglo no está ordenado o cuando el tamaño de la lista es pequeño. Sin embargo, puede volverse ineficiente en listas muy grandes, ya que se deben examinar todos los elementos de la lista en el peor caso.

En términos de eficiencia de espacio, la búsqueda lineal es muy eficiente, ya que solo requiere una cantidad constante de espacio adicional para almacenar las variables de control utilizadas en el proceso de búsqueda.

- **Peor caso:** El peor caso ocurre cuando el elemento buscado no está presente en el arreglo o se encuentra en la última posición. En este escenario, la búsqueda lineal debe recorrer todos los elementos del arreglo hasta llegar al final o encontrar el valor. Por lo tanto, en el peor caso, la búsqueda lineal tiene una complejidad temporal de O(n), donde n es la longitud del arreglo. Esto significa que el tiempo de ejecución crece linealmente con respecto al tamaño del arreglo.
- **Promedio caso:** En el promedio caso, asumimos que el elemento buscado tiene igual probabilidad de estar en cualquier posición del arreglo. Si consideramos n elementos en el arreglo, en promedio, se necesitará recorrer la mitad del arreglo para encontrar el valor buscado o determinar que no está presente. Por lo tanto, en el promedio caso, la complejidad temporal de la búsqueda lineal Sigue siendo O(n).
- **Mejor caso:** El mejor caso ocurre cuando el elemento buscado se encuentra en la primera posición del arreglo. En este escenario, la búsqueda lineal solo necesita realizar una comparación antes de encontrar el valor. Por lo tanto, en el mejor caso, la complejidad temporal de la búsqueda lineal es O(1), es decir, constante. Sin

embargo, es importante tener en cuenta que el mejor caso es poco común y no representa el rendimiento típico de la búsqueda lineal.

Complejidad en el tiempo

El algoritmo recorre secuencialmente los elementos del arreglo hasta encontrar el valor buscado o llegar al final del arreglo. En el peor caso, debe recorrer todos los elementos, lo cual implica una complejidad temporal de O(n), donde n es la longitud del arreglo. Esto significa que el tiempo de ejecución aumenta linealmente con el tamaño del arreglo.

Complejidad en el espacio

La búsqueda lineal no requiere memoria adicional más allá del arreglo en el que se realiza la búsqueda. Por lo tanto, la complejidad en el espacio de la búsqueda lineal es O(1), es decir, constante. No importa cuántos elementos haya en el arreglo, la cantidad de memoria utilizada por el algoritmo de búsqueda lineal no cambia.

Búsqueda Binaria

Es el algoritmo de búsqueda más popular y eficiente. De hecho, es el algoritmo de búsqueda más rápido. Al igual que la ordenación por salto, también necesita que se ordene el array. Se basa en el enfoque de dividir y conquistar en el que dividimos el array en dos mitades y luego comparamos el elemento que estamos buscando con el elemento del medio. Si el elemento del medio coincide, devolvemos el índice del elemento del medio; de lo contrario, nos movemos a la mitad izquierda y derecha según el valor del artículo.

Los pasos para seguir para una búsqueda binaria son los siguientes:

1. Toma el elemento que deseas buscar.
2. Establece dos índices: uno para el primer elemento de la lista (índice izquierdo) y otro para el último elemento (índice derecho).
3. Calcula el índice del elemento del medio entre el índice izquierdo y el índice derecho: índice_medio = (índice_izquierdo + índice_derecho) / 2.

4. Compara el elemento buscado con el elemento en el índice_medio.
    a. Si el elemento buscado es igual al elemento en el índice_medio, se ha encontrado el resultado y el proceso termina.
    b. Si el elemento buscado es menor que el elemento en el índice_medio, actualiza el índice_derecho a índice_medio - 1 y vuelve al paso 3.
    c. Si el elemento buscado es mayor que el elemento en el índice_medio, actualiza el índice_izquierdo a índice_medio + 1 y vuelve al paso 3.
5. Repite los pasos 3 y 4 hasta que encuentres el elemento buscado o hasta que el índice_izquierdo sea mayor que el índice_derecho.
6. Si el índice_izquierdo es mayor que el índice_derecho y no se ha encontrado el elemento, se considera que el elemento no está presente en la lista.

Ejemplos de algoritmos de búsqueda binaria:

```java
public static int secuencialBinaria(int[] a, int valor){

    int inf = 0;
    int sup = a.length-1;
    int mitad = 0;

    while (inf <= sup){
        mitad = (inf+sup)/2
        if (valor == a[mitad]){
            return mitad;
        }
        else if (valor < a[mitad]){
            inf = mitad+1;
        }
        else if (valor > a[mitad]){
            sup = mitad-1;
        }
    }
    return -1;
}
```

```java
case "Secuencial Binaria":
    int SB[] = {10,20,1,430,30};
    toolsList.imprimePantalla(busquedas.imprimeOrdenados(SB));
    Arrays.sort(SB);
    toolsList.imprimePantalla("Arreglo ordenado:"+busquedas.imprimeOrdenados(SB));

    if(busquedas.secuencialBinaria(SB, toolsList.leerInt("Numero a buscar:"))>=0)
        toolsList.imprimePantalla("El dato existe");
    else
        toolsList.imprimePantalla("El dato no existe.");
    break;
```

La búsqueda binaria tiene una eficiencia de tiempo de O(log n), donde "n" es el tamaño de la lista o arreglo en el que se está realizando la búsqueda. Esto significa

que el tiempo de ejecución aumenta de forma logarítmica a medida que el tamaño de la lista crece.

La principal ventaja de la búsqueda binaria es que reduce a la mitad el espacio de búsqueda en cada iteración. En cada paso, el algoritmo compara el elemento buscado con el elemento en el medio del rango de búsqueda actual y descarta una mitad de la lista, concentrándose en la otra mitad. Esto reduce rápidamente el espacio de búsqueda a una pequeña fracción del tamaño original.

Es importante destacar que la búsqueda binaria requiere que la lista esté ordenada. Si la lista no está ordenada, se debe realizar un paso adicional para ordenarla, lo que lleva tiempo adicional y aumenta la complejidad total del algoritmo.

- Peor caso: El peor caso ocurre cuando el elemento buscado no está presente en la lista. En cada iteración, la búsqueda binaria divide a la mitad el espacio de búsqueda, descartando una porción significativa de la lista. Esto se repite hasta que el espacio de búsqueda se reduce a cero. En este caso, la búsqueda debe realizar logarítmicamente n comparaciones para determinar que el elemento no está presente.
- Caso promedio: El análisis del caso promedio es similar al del peor caso. En promedio, la búsqueda binaria realiza logarítmicamente n comparaciones para encontrar el elemento deseado o determinar que no está presente. Sin embargo, este análisis asume que los datos están distribuidos de manera uniforme y que no hay elementos que se busquen con mayor frecuencia que otros.
- Mejor caso: El mejor caso ocurre cuando el elemento buscado se encuentra exactamente en el centro de la lista. En este caso, la búsqueda binaria puede encontrar el elemento deseado en el primer paso, sin necesidad de realizar más comparaciones. El tiempo de ejecución es constante y no depende del tamaño de la lista.

Complejidad en el tiempo

En el peor caso y en el caso promedio, la búsqueda binaria tiene una complejidad de tiempo logarítmica, lo que significa que el número de comparaciones necesarias crece de manera muy lenta a medida que el tamaño de la lista aumenta. La búsqueda binaria divide repetidamente el espacio de búsqueda a la mitad en cada iteración, lo que reduce rápidamente el espacio de búsqueda a una fracción muy pequeña del tamaño original. Esto resulta en un tiempo de ejecución muy eficiente para la búsqueda binaria, especialmente en comparación con la búsqueda lineal.En el mejor caso, cuando el elemento buscado se encuentra en el centro de la lista, la búsqueda binaria puede encontrarlo en el primer paso sin necesidad de realizar más comparaciones. En este caso, la complejidad en el tiempo es constante O(1).

Complejidad en el espacio

En cuanto a la complejidad en el espacio, la búsqueda binaria tiene una complejidad en el espacio de O(1), lo que significa que requiere una cantidad constante de espacio adicional independientemente del tamaño de la lista. Esto se debe a que la búsqueda binaria no requiere estructuras de datos adicionales más allá de las variables utilizadas para realizar las comparaciones y controlar el rango de búsqueda. No se requiere espacio adicional proporcional al tamaño de la lista.

Búsqueda de Knuth Morris Pratt

La búsqueda de patrones de Knuth-Morris-Pratt (KMP) es un algoritmo eficiente para buscar la ocurrencia de un patrón dado dentro de un texto más largo. Fue desarrollado por Donald Knuth, Vaughan Pratt y James Morris en 1977.

El algoritmo KMP utiliza una estrategia de búsqueda inteligente que evita comparar los caracteres en el texto y el patrón más de una vez, lo que lo hace más eficiente que otros enfoques de búsqueda de patrones.

El proceso de búsqueda de patrones KMP se puede dividir en dos fases principales: la fase de preprocesamiento y la fase de búsqueda.

Fase de preprocesamiento:

Se construye una tabla de valores llamada tabla de fallos o tabla de prefijos que ayuda a determinar el desplazamiento adecuado cuando hay una falta de coincidencia.

La tabla de fallos se crea analizando el patrón y buscando prefijos que coincidan con sufijos.

Esta tabla se utiliza durante la fase de búsqueda para saltar posiciones innecesarias.

Fase de búsqueda:

Se compara el patrón con el texto carácter a carácter, empezando desde el inicio del texto.

Cuando se produce una falta de coincidencia entre el patrón y el texto, la tabla de fallos se utiliza para determinar el desplazamiento adecuado.

En lugar de volver a comenzar desde el principio del patrón, el algoritmo KMP utiliza la información almacenada en la tabla de fallos para saltar adelante en el texto y continuar la comparación desde allí.

Al evitar comparaciones redundantes, el algoritmo KMP logra una eficiencia en el tiempo de ejecución de O(n+m), donde n es la longitud del texto y m es la longitud del patrón. Esto lo hace particularmente útil cuando se busca un patrón dentro de un texto grande, ya que el tiempo de ejecución no depende linealmente del tamaño total de los datos

```java
public static int KMP(char[] texto, char[] pat) {
    int n = texto.length;
    int m = pat.length;
    int[] lps = new int[m];
    int tam = 0;
    int i = 1;
    lps[0] = 0;
    while (i < m) {
        if (pat[i] == pat[tam]) {
            tam++;
            lps[i] = tam;
            i++;
        } else {
            if (tam != 0) {
                tam = lps[tam - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
    int j = 0;
    for (i = 0; i < n; ) {
        if (pat[j] == texto[i]) {
            j++;
            i++;
        }
        if (j == m) {
            return i - j;
        } else if (i < n && pat[j] != texto[i]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
    return -1;
}
```

```java
case "Knuth Morris Pratt":
    char[] texto = {'A', 'B', 'C', 'D', 'A', 'B', 'C', 'D', 'A', 'B', 'D'};
    char[] patron = {'D', 'L', 'D'};
    toolsList.imprimePantalla(busquedas.imprimeOrdenados(texto));
    toolsList.imprimePantalla("Patron a buscar:" + busquedas.imprimeOrdenados(patron));
    if(busquedas.KMP(texto,patron)>=0)
        toolsList.imprimePantalla("El patron existe");
    else
        toolsList.imprimePantalla("El patron no existe.");
    break;
```

Análisis de eficiencia

El algoritmo de búsqueda de patrones de Knuth-Morris-Pratt (KMP) tiene una eficiencia en el tiempo de ejecución de O(n + m), donde n es la longitud del texto y m es la longitud del patrón.

1. Fase de preprocesamiento:

- La construcción de la tabla de fallos se realiza en un bucle while que recorre el patrón una sola vez. Esto implica un tiempo de ejecución de O(m), donde m es la longitud del patrón.

2. Fase de búsqueda:

- El bucle principal de búsqueda recorre el texto una vez, con un índice que va de 0 a n-1, donde n es la longitud del texto. Por lo tanto, esta parte del algoritmo tiene una complejidad de O(n).

- En cada iteración del bucle de búsqueda, se realizan comparaciones entre los caracteres del patrón y el texto. Si hay una falta de coincidencia, se utiliza la información de la tabla de fallos para determinar el desplazamiento adecuado. Las comparaciones se realizan en tiempo constante, ya que solo se comparan caracteres individuales.

- En el peor caso, cuando no hay coincidencias entre el patrón y el texto, el desplazamiento máximo es m-1. Esto implica que cada posición del texto se compara a lo sumo m veces en total a lo largo del algoritmo.

- Dado que la construcción de la tabla de fallos tiene un tiempo de ejecución de O(m), y las comparaciones se realizan a lo sumo m veces en total, la fase de búsqueda tiene una complejidad de O(n + m).

En general, la complejidad de tiempo total del algoritmo KMP es O(n + m), lo que lo hace altamente eficiente en comparación con otros algoritmos de búsqueda de patrones como la fuerza bruta, que tiene una complejidad de O(n * m). El algoritmo KMP evita comparaciones redundantes utilizando la información almacenada en la tabla de fallos, lo que resulta en una mejora significativa en el rendimiento, especialmente cuando se trabaja con textos y patrones grandes.

La complejidad espacial del algoritmo KMP es O(m), ya que se necesita almacenar la tabla de fallos, que tiene una longitud igual a la del patrón.

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

El rendimiento del algoritmo de búsqueda de patrones de Knuth-Morris-Pratt (KMP) puede variar según diferentes casos, como el mejor de los casos, el caso medio y el peor de los casos. Analicemos cada uno de ellos con ejemplos:

1. Mejor de los casos:

- El mejor de los casos ocurre cuando el patrón y el texto tienen una coincidencia en la primera comparación. Esto significa que el primer carácter del patrón coincide con el primer carácter del texto.

- En este caso, la fase de búsqueda realizará una única comparación y encontrará una coincidencia en el primer índice del texto. Por lo tanto, la complejidad de tiempo será O(1).

Aquí un ejemplo:

Texto = “ABCDEF”

Patron = “A”

Indices = buscar_patron(texto, patron)

Print(“El patrón se encontró en los índices:”, indices)

2. Caso medio:

- El caso medio ocurre cuando hay múltiples ocurrencias del patrón en diferentes posiciones del texto, pero no se encuentran coincidencias inmediatamente.

- En este caso, el algoritmo KMP utilizará la información almacenada en la tabla de fallos para saltar posiciones innecesarias y realizará menos comparaciones en general. La complejidad de tiempo será O(n + m).

Aquí un ejemplo:

Texto = “ABCABCABC”

Patron = “ABC”

Indices = buscar_patron(texto, patron)

Print(“El patrón se encontró en los índices:”, indices)

3. Peor de los casos:

- El peor de los casos ocurre cuando no hay coincidencias entre el patrón y el texto, lo que significa que el patrón no aparece en absoluto o aparece solo en la última posición del texto.

- En este caso, el algoritmo KMP realizará comparaciones en todas las posiciones del texto y necesitará recorrer todo el patrón para cada posición. La complejidad de tiempo será O(n * m).

Aquí un ejemplo:

Texto = “AAAAAAA”

Patron = “BBB”

Indices = buscar_patron(texto, patron)

Print(“El patrón se encontró en los índices:”, indices)

En resumen, el algoritmo KMP tiene un mejor rendimiento en el caso medio, donde utiliza la información de la tabla de fallos para evitar comparaciones innecesarias. En el mejor de los casos, cuando hay una coincidencia inmediata, el rendimiento es aún mejor, mientras que en el peor de los casos, donde no hay coincidencias, el rendimiento se degrada y se asemeja a una búsqueda de fuerza bruta. Sin embargo, en general, el algoritmo KMP sigue siendo más eficiente que otros enfoques de búsqueda de patrones en la mayoría de los casos.

Complejidad en el tiempo y complejidad espacial

La complejidad en el tiempo y la complejidad espacial del algoritmo de búsqueda de patrones de Knuth-Morris-Pratt (KMP) son las siguientes:

Complejidad en el tiempo:

- Mejor caso y caso medio: O(n + m)

- Peor caso: O(n * m)

Donde:

- n es la longitud del texto.

- m es la longitud del patrón.

En el mejor caso y caso medio, el algoritmo KMP tiene una complejidad de tiempo de O(n + m). Esto se debe a que utiliza la información de la tabla de fallos para evitar comparaciones innecesarias y realizar saltos en el texto. La fase de preprocesamiento para construir la tabla de fallos tiene una complejidad de O(m), y la fase de búsqueda realiza una cantidad mínima de comparaciones en cada posición del texto.

En el peor caso, cuando no hay coincidencias entre el patrón y el texto, el algoritmo KMP realizará comparaciones en todas las posiciones del texto y necesitará recorrer todo el patrón para cada posición. Esto resulta en una complejidad de tiempo de O(n * m).

Complejidad espacial:

- O(m)

La complejidad espacial del algoritmo KMP es O(m), donde m es la longitud del patrón. Esto se debe a que se necesita almacenar la tabla de fallos, que tiene una longitud igual a la del patrón. La tabla de fallos ocupa un espacio adicional en la memoria para almacenar los valores de desplazamiento para cada posición del patrón.

Es importante destacar que la complejidad espacial de O(m) puede considerarse eficiente, ya que solo depende de la longitud del patrón y no del tamaño total del texto. Esto hace que el algoritmo KMP sea adecuado para buscar patrones en textos grandes, ya que no requiere un espacio adicional proporcional al tamaño total del texto.

Saltar búsqueda

Jump Search es un algoritmo de búsqueda utilizado para encontrar el valor de un elemento en una lista o matriz ordenada. También conocido como “búsqueda por salto”, este algoritmo combina las ventajas de la búsqueda lineal y la búsqueda binaria.

El proceso de búsqueda comienza dividiendo el arreglo en bloques de tamaño óptimo, generalmente de tamaño √n, donde “n” es el tamaño total del arreglo. Luego, se verifica si el valor que se busca se encuentra en el primer bloque. Si es así, la búsqueda termina y se devuelve la posición del elemento encontrado.

En caso contrario, se salta al siguiente bloque y se repite el proceso. Si el valor buscado es menor que el elemento actual del bloque, se realiza una búsqueda lineal dentro de ese bloque para encontrar el elemento deseado. Si el valor buscado es mayor, se continúa saltando a bloques sucesivos hasta encontrar un bloque que contenga un elemento mayor o igual al valor buscado.

Una vez que se encuentra un bloque que contiene un elemento mayor o igual al valor buscado, se realiza una búsqueda lineal dentro de ese bloque para encontrar el valor exacto. Si el valor buscado no se encuentra en la lista, se considera que el valor no existe en el arreglo.

```
        break;
    case "Busqueda Salto":
        int JS []= {10,13,15,26,28,50,56,88,94,127,159,356,480,567,689,699,780,850,956,995};
        toolsList.imprimePantalla(busquedas.imprimeOrdenados(JS));
        if(busquedas.jumpSearch(JS, toolsList.leerInt("Numero a buscar:"))>=0)
            toolsList.imprimePantalla("El dato existe");
        else
            toolsList.imprimePantalla("El dato no existe.");
        break;
```

```java
public static int jumpSearch(int a[], int valor){
    int inicio = 0;
    int fin = a.length;
    int tbloque = (int)Math.sqrt(a.length);

    while(a[fin-1]<=valor && fin<a.length) {
        inicio = fin;
        fin = fin+tbloque;
        if(fin>a.length-1)
            fin = a.length;
    }

    for (int i = inicio;i<fin;i++){
        if(a[i]==valor)
            return i;
    }
    return -1;
}
```

Análisis de eficiencia

El análisis de eficiencia del algoritmo de Jump Search se puede realizar considerando su complejidad de tiempo y espacio.

Complejidad de tiempo:

El algoritmo de Jump Search tiene una complejidad de tiempo de O(√n), donde “n” es el tamaño del arreglo o lista. Esto se debe a que el algoritmo realiza saltos de tamaño √n para encontrar un bloque que contenga un elemento mayor o igual al valor buscado. Luego, realiza una búsqueda lineal dentro de ese bloque, lo cual puede tener una complejidad de O(√n) en el peor de los casos si el elemento buscado se encuentra al final del bloque.

En comparación con la búsqueda lineal, que tiene una complejidad de tiempo de O(n), el algoritmo de Jump Search es más eficiente, especialmente en arreglos o listas ordenadas, ya que aprovecha la propiedad de ordenamiento para realizar

saltos más grandes. Sin embargo, la búsqueda binaria sigue siendo más eficiente, con una complejidad de tiempo de O(log n).

Complejidad de espacio:

El algoritmo de Jump Search utiliza una cantidad constante de espacio adicional para las variables necesarias, lo cual da una complejidad de espacio de O(1).

En resumen, el algoritmo de Jump Search es una mejora sobre la búsqueda lineal en términos de eficiencia, pero aún es menos eficiente que la búsqueda binaria. Es especialmente útil cuando el arreglo no cabe completamente en la memoria principal y se almacena en una memoria secundaria, como un disco duro, ya que minimiza el número de accesos a dicha memoria.

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

El análisis de los casos mejor, medio y peor de los casos en el algoritmo de Jump Search se puede realizar teniendo en cuenta la distribución de los elementos en el arreglo o lista y la posición del valor buscado.

1. **Mejor caso:**

El mejor caso ocurre cuando el elemento buscado se encuentra en la primera posición del arreglo o lista. En este caso, el algoritmo realiza un solo salto y encuentra el elemento deseado en el primer bloque. La complejidad de tiempo es O(1) en el mejor caso.

Ejemplo:

Arr = [1, 3, 5, 7, 9, 11, 13]

Valor buscado: 1

El algoritmo realiza un solo salto y encuentra el valor 1 en la posición 0.

2. **Caso medio:**

El caso medio ocurre cuando el elemento buscado está distribuido de manera uniforme en el arreglo o lista. En este caso, el algoritmo realiza varios saltos para encontrar el bloque que contiene un elemento mayor o igual al valor buscado y luego realiza una búsqueda lineal dentro de ese bloque. La complejidad de tiempo es O(√n) en el caso medio.

Ejemplo:

Arr = [2, 4, 6, 8, 10, 12, 14]

Valor buscado: 10

El algoritmo realiza dos saltos y encuentra el bloque [8, 10, 12, 14]. Luego realiza una búsqueda lineal y encuentra el valor 10 en la posición 4.

3. **Peor caso:**

El peor caso ocurre cuando el elemento buscado es el último elemento del arreglo o lista o no se encuentra en absoluto. En este caso, el algoritmo realiza saltos hasta llegar al último bloque y luego realiza una búsqueda lineal dentro de ese bloque. La complejidad de tiempo es O(√n) en el peor caso.

Ejemplo:

Arr = [2, 4, 6, 8, 10, 12, 14]

Valor buscado: 14 (no existe)

El algoritmo realiza dos saltos y llega al último bloque [12, 14]. Luego realiza una búsqueda lineal y no encuentra el valor buscado, devolviendo -1.

En resumen, el mejor caso ocurre cuando el elemento buscado se encuentra al principio del arreglo, el caso medio asume una distribución uniforme de los elementos y el peor caso ocurre cuando el elemento buscado es el último o no se encuentra en absoluto. La complejidad de tiempo del algoritmo de Jump Search es O(√n) en promedio, lo que lo hace eficiente en comparación con la búsqueda lineal, pero aún menos eficiente que la búsqueda binaria con complejidad O(log n).

En resumen:

- Complejidad en el tiempo: O(√n)

- Complejidad espacial: O(1)

La complejidad en el tiempo de O(√n) hace que el algoritmo de Jump Search sea más eficiente que la búsqueda lineal (O(n)) en arreglos o listas ordenadas, pero menos eficiente que la búsqueda binaria (O(log n)). La complejidad espacial de O(1) indica que el algoritmo no requiere asignar memoria adicional proporcional al tamaño del arreglo.

Búsqueda de interpolación

En que consiste la búsqueda de interpolación:

Es un algoritmo similar a búsqueda binaria para buscar un valor objetivo dado en un array ordenado, es un algoritmo utilizado para buscar un valor en una lista ordenada de elementos. A diferencia de otros algoritmos de búsqueda, la búsqueda de interpolación utiliza información sobre la distribución de los valores en la lista para realizar una estimación más precisa de la posición del valor buscado.

La búsqueda por interpolación calcula la posición de la sonda según esta fórmula:

$$probe = lowEnd + \frac{(highEnd - lowEnd) \times (item - data[lowEnd])}{data[highEnd] - data[lowEnd]}$$

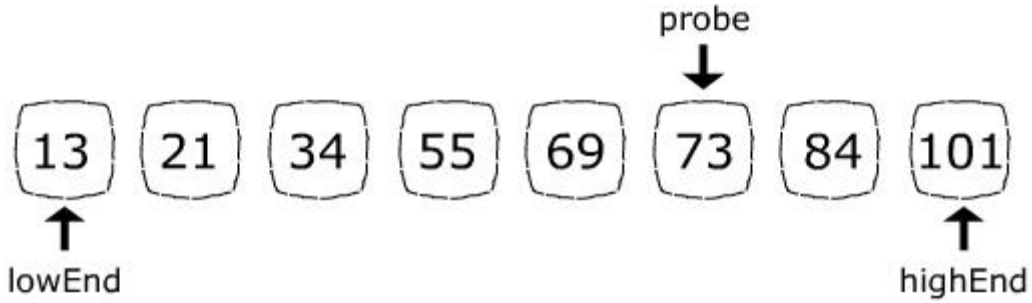
Veamos cada uno de los términos:

- probe: a este parámetro se le asignará la nueva posición de la probe.
- lowEnd: el índice del elemento más a la izquierda en el espacio de búsqueda actual.
- highEnd: el índice del elemento más a la derecha en el espacio de búsqueda actual.
- data []: la matriz que contiene el espacio de búsqueda original.
- ítem: el ítem que estamos buscando.

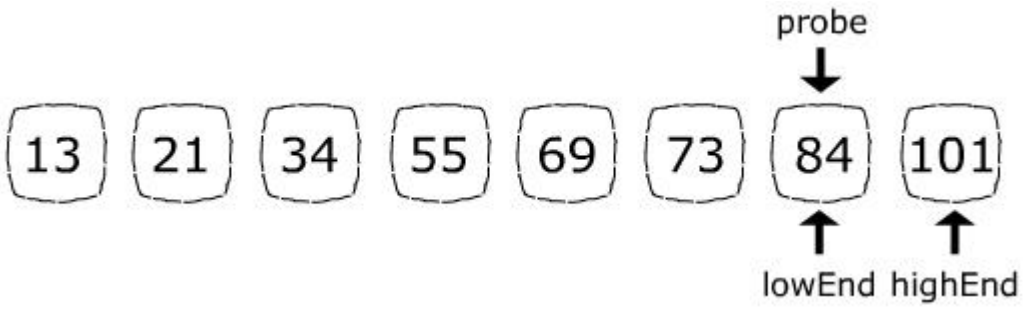
Digamos que queremos encontrar la posición de 84 en la siguiente matriz:

La longitud de la matriz es 8, por lo que inicialmente highEnd = 7 y lowEnd = 0 (porque el índice de la matriz comienza en 0, no en 1).

En el primer paso, la fórmula de posición de la sonda dará como resultado sonda = 5:



Debido a que 84 (el elemento que estamos buscando) es mayor que 73 (el elemento de posición actual de la sonda), el siguiente paso abandonará el lado izquierdo de la matriz asignando lowEnd = probe + 1. Ahora el espacio de búsqueda consta de solo 84 y 101. La fórmula de posición de la sonda establecerá la sonda = 6, que es exactamente el índice de 84:



Dado que se encuentra el elemento que buscábamos, se devolverá la posición 6.

Ejemplos de implementaciones del algoritmo en las estructuras de datos:

```
public class Main {

  public static void main(String[] args) {
    int[] array = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    int element = 12;

    int index = Search.interpolation(array, element);

    if (index ≠ -1) {
      toolsList.imprimePantalla("Elemento encontrado en la posicion: " + index);
    } else {
      toolsList.imprimePantalla("Elemento no encontrado");
    }
  }
}
```

```
public class Search {

  public static int interpolation(int[] array, int element) {
    int low = 0;
    int high = array.length - 1;

    while (low ≤ high && element ≥ array[low] && element ≤ array[high]) {
      if (low == high) {
        if (array[low] == element)
          return low;
        return -1;
      }

      int index = low + ((element - array[low]) * (high - low)) / (array[high] - array[low]);

      if (array[index] == element)
        return index;

      if (array[index] < element)
        low = index + 1;
      else
        high = index - 1;
    }
    return -1;
  }
}
```

Análisis de eficiencia:

La búsqueda de interpolación es eficiente en la mayoría de los casos, especialmente cuando los datos están uniformemente distribuidos. Sin embargo, en situaciones donde los datos son desiguales o no siguen un patrón predecible, la búsqueda de interpolación puede no ser tan eficiente como la búsqueda binaria, que garantiza una complejidad O (log n) en todos los casos.

Es importante tener en cuenta que la elección del algoritmo de búsqueda depende del contexto y las características específicas de los datos. En algunos casos, la búsqueda de interpolación puede ser más eficiente, mientras que en otros casos la búsqueda binaria u otros algoritmos de búsqueda pueden ser más apropiados.

Análisis de los casos.

La búsqueda de interpolación es un algoritmo de búsqueda que puede ser eficiente en ciertos escenarios, pero su eficiencia puede verse afectada por algunas condiciones particulares.

- **Mejor de los casos:**
  El mejor caso ocurre cuando el elemento buscado se encuentra en la posición estimada de forma precisa. En este caso, la complejidad de la búsqueda de interpolación puede ser tan baja como O(1), ya que se encuentra el elemento de inmediato.
  Ejemplo:

```
int[] arr = {1, 2, 3, 4, 5};
int target = 3;

int index = Search.interpolation(arr, target);
```

En este ejemplo, el arreglo está ordenado y el elemento buscado, 3, se encuentra exactamente en la posición estimada. Como resultado, la búsqueda de interpolación retorna el índice correspondiente al elemento de forma inmediata, sin necesidad de realizar más comparaciones

- **Caso medio:**

  En promedio, la búsqueda de interpolación tiene una complejidad O(log log n) cuando se aplica en una lista ordenada de tamaño n. Sin embargo, esto se basa en el supuesto de que los elementos en la lista están uniformemente distribuidos. En este caso, el algoritmo puede realizar estimaciones más precisas y reducir el rango de búsqueda rápidamente. En tales casos, la búsqueda de interpolación puede superar a la búsqueda binaria en términos de eficiencia promedio.

  Ejemplo:

```
int[] arr = {1, 3, 5, 7, 9};
int target = 5;

int index = Search.interpolation(arr, target);
```

  El arreglo está ordenado y los elementos están uniformemente distribuidos. El elemento buscado, 5, se encuentra aproximadamente en el centro del arreglo. La búsqueda de interpolación realiza estimaciones más precisas y reduce el rango de búsqueda rápidamente, lo que resulta en un tiempo de ejecución eficiente.

- **Peor de los casos:**

  El peor caso de la búsqueda de interpolación ocurre cuando los elementos no están uniformemente distribuidos y siguen una distribución desigual. En este escenario, la complejidad puede degradarse a O(n), donde n es el tamaño de la lista. Esto se debe a que la fórmula de interpolación puede

producir estimaciones inexactas y generar un comportamiento similar a una búsqueda lineal.

Ejemplo:

```
int[] arr = {1, 3, 5, 7, 9};
int target = 10;

int index = Search.interpolation(arr, target);
```

El arreglo está ordenado y los elementos están desiguales. El elemento buscado, 10, está fuera del rango de los elementos existentes en el arreglo. La fórmula de interpolación producirá estimaciones inexactas y la búsqueda de interpolación se comportará de manera similar a una búsqueda lineal, recorriendo todo el arreglo.

Complejidad en el tiempo:

Cuando el elemento buscado se encuentra en la posición estimada de forma precisa, la complejidad temporal de la búsqueda de interpolación puede ser O(1) ya que se encuentra el elemento de inmediato.

En el caso promedio, asumiendo una distribución uniforme de los elementos en la lista, la complejidad temporal de la búsqueda de interpolación es O(log log n), donde n es el tamaño de la lista. Esto se debe a que el algoritmo puede realizar estimaciones más precisas y reducir el rango de búsqueda rápidamente.

En el peor caso, cuando los elementos no están uniformemente distribuidos y siguen una distribución desigual, la complejidad temporal de la búsqueda de interpolación puede degradarse a O(n), donde n es el tamaño de la lista. En este escenario, la fórmula de interpolación puede producir estimaciones inexactas y generar un comportamiento similar a una búsqueda lineal.

Complejidad espacial:

La complejidad espacial de la búsqueda de interpolación es O(1), ya que no se requiere memoria adicional en relación con el tamaño de los datos de entrada. El algoritmo utiliza variables adicionales para mantener los límites del rango de búsqueda y realizar cálculos, pero no depende del tamaño de la lista para asignar memoria adicional.

En términos de complejidad espacial, la búsqueda de interpolación es eficiente, ya que solo requiere una cantidad constante de memoria adicional, independientemente del tamaño de la lista.

Búsqueda Exponencial

La búsqueda exponencial es un algoritmo de búsqueda utilizado en estructuras de datos ordenadas. Se basa en el principio de acelerar la búsqueda mediante la exploración de posiciones exponencialmente distantes en lugar de realizar pasos lineales.

El algoritmo de búsqueda exponencial funciona de la siguiente manera:

1. Comienza con un rango inicial que cubre toda la estructura de datos ordenada.
2. Compara el elemento buscado con el elemento en el medio del rango.

- Si son iguales, se ha encontrado el elemento y la búsqueda termina.
- Si el elemento buscado es menor, se reduce a la mitad el rango y se repite el paso 2 con la mitad inferior.
- Si el elemento buscado es mayor, se duplica el tamaño del rango y se repite el paso 2 con el nuevo rango.

Este proceso se repite hasta encontrar el elemento buscado o hasta que el rango se reduce a un solo elemento. Si el rango se reduce a un solo elemento y ese

elemento no coincide con el buscado, significa que el elemento no está presente en la estructura de datos.

A continuación, se muestra un ejemplo de implementación de búsqueda exponencial:

```java
public static int busquedaExponencial(int[] a, int valor) {
    int tamaño = a.length;
    if (a[0] == valor) {
        return 0;
    }

    int i = 1;
    while (i < tamaño && a[i] <= valor) {
        i *= 2;
    }

    return busquedaBinaria(a, valor, i / 2, Math.min(i, tamaño - 1));
}

public static int busquedaBinaria(int[] a, int clave, int inicio, int fin) {
    if (fin >= inicio) {
        int medio = inicio + (fin - inicio) / 2;

        if (a[medio] == clave) {
            return medio;
        }

        if (a[medio] > clave) {
            return busquedaBinaria(a, clave, inicio, medio - 1);
        }

        return busquedaBinaria(a, clave, medio + 1, fin);
    }

    return -1;
}
```

```java
case "Exponencial":
    int[] EXP = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20,22,24,26};
    toolsList.imprimePantalla(busquedas.imprimeOrdenados(EXP));
    if(busquedas.interpolacion(EXP, toolsList.leerInt("Numero a buscar:"))>=0)
        toolsList.imprimePantalla("El dato existe");
    else
        toolsList.imprimePantalla("El dato no existe.");
    break;
```

En términos de eficiencia, la búsqueda exponencial puede ser más rápida que la búsqueda binaria tradicional en ciertos casos, especialmente cuando el elemento buscado está más cerca del principio del arreglo. Sin embargo, en casos en los que

el elemento buscado está cerca del arreglo final, la búsqueda exponencial puede requerir más iteraciones que la búsqueda binaria para encontrar el elemento.

El análisis de los casos de la búsqueda exponencial es el siguiente:

- **Mejor de los casos:**

  El elemento buscado se encuentra en el primer intento, es decir, en el índice 0. La complejidad en tiempo es O(1), ya que solo se necesita una comparación.

- **Caso medio:**

  El elemento buscado está ubicado en alguna posición aleatoria dentro del arreglo. La complejidad en tiempo es O(log n), donde "n" es el tamaño del arreglo. Esto se debe a que la búsqueda se reduce a la búsqueda binaria en un rango específico del arreglo.

- **Peor de los casos:**

  El elemento buscado no está presente en el arreglo. La complejidad en tiempo es O(log n), ya que la búsqueda se reduce a la búsqueda binaria en un rango específico del arreglo, pero en última instancia no se encuentra el elemento.

En cuanto a la complejidad en el tiempo y la complejidad espacial:

- **Complejidad en el tiempo:**

  En el peor de los casos, la complejidad en el tiempo es O(log n), donde "n" es el tamaño del arreglo. Esto se debe a que la búsqueda se reduce a la

búsqueda binaria en un rango específico. Sin embargo, en el mejor de los casos, la complejidad en el tiempo es O(1), ya que el elemento buscado se encuentra en el primer intento.

- **Complejidad espacial:**

  La complejidad espacial es O(1) ya que no se requiere espacio adicional dependiente del tamaño del arreglo. Solo se utilizan variables adicionales para mantener el estado de la búsqueda.

Búsqueda de Fibonacci

La búsqueda de Fibonacci es un algoritmo de búsqueda de intervalo eficiente. Es similar a búsqueda binaria en el sentido de que también se basa en la estrategia de divide y vencerás y también necesita el array para ser ordenado. Además, la complejidad del tiempo para ambos algoritmos es logarítmica. Se llama búsqueda de Fibonacci porque utiliza la serie de Fibonacci (el número actual es la suma de dos predecesores F[i] = F[i-1] + F[i-2], F[0] = 0 y F[1]= 1 son los dos primeros números Series) y divide el array en dos partes con el tamaño dado por los números de Fibonacci. Es un método fácil de calcular que usa solo operaciones de suma y resta en comparación con la división, multiplicación y cambios de bits requeridos por la búsqueda binaria.

Ejemplo de implementación en Java de la búsqueda de Fibonacci:

```
case "Busqueda Fibonacci":
    int BF[] = {10,13,15,26,28,50,56,88,94,127,159,356,480,567,689,699,780};
    toolsList.imprimePantalla(busquedas.imprimeOrdenados(BF));
    if(busquedas.fibonacciSearch(BF, toolsList.leerInt("Numero a buscar:"))>=0)
        toolsList.imprimePantalla("El dato existe");
    else
        toolsList.imprimePantalla("El dato no existe.");
    break;
```

```java
public static int fibonacciSearch(int[] a, int target) {
    int n = a.length;
    int fib2 = 0;
    int fib1 = 1;
    int fib = fib2 + fib1;
    //calcular fibonacci
    while (fib < n) {
        fib2 = fib1;
        fib1 = fib;
        fib = fib2 + fib1;
    }
    int dif = -1;
    while (fib > 1) {
       int i = Math.min(dif + fib2, n - 1);
        if (a[i] < target) {
            fib = fib1;
            fib1 = fib2;
            fib2 = fib - fib1;
            dif = i;
        }
        else
            if (a[i] > target) {
                fib = fib2;
                fib1 = fib1 - fib2;
                fib2 = fib - fib1;
        }
        else {
            return i;
        }
    }
    if (fib1 == 1 && a[dif + 1] == target) {
        return dif + 1;
    }

   return -1;
}
```

Caso medio

Reducimos el espacio de búsqueda en un tercio / dos tercios en cada iteración y, por lo tanto, el algoritmo tiene una complejidad logarítmica. La complejidad de tiempo del algoritmo de búsqueda de Fibonacci es O(logn).

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 que comparamos.

Peor caso

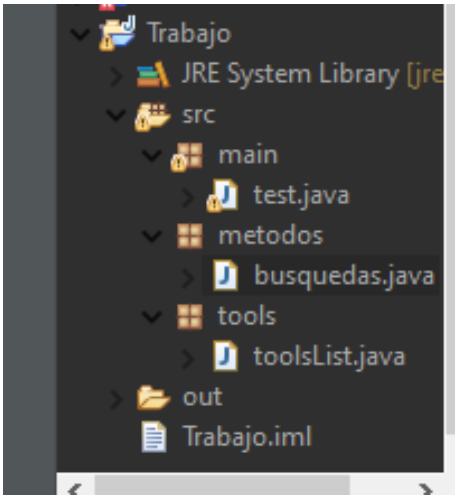
El peor de los casos ocurre cuando el elemento objetivo X siempre está presente en el subarreglo más grande. La complejidad de tiempo en el peor de los casos es O(logn). Es lo mismo que la complejidad del tiempo promedio de los casos.

Complejidad en el tiempo: La complejidad en el tiempo de la búsqueda de Fibonacci es O(log n). Esto se debe a que en cada iteración, el tamaño del rango de búsqueda se reduce aproximadamente a la mitad.

Complejidad espacial: La complejidad espacial de la búsqueda de Fibonacci es O(1), ya que no se requiere almacenamiento adicional en función del tamaño de los datos de entrada.

Desarrollo de la práctica.

Para el presente reporte se creo la siguiente estructura de paquetes y clases para el correcto orden y ejecución del programa



Se creo un menú donde tenga todas las opciones de búsqueda

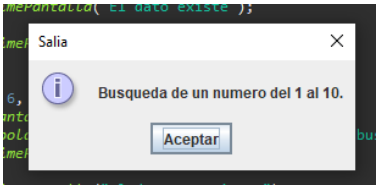
Nodito.java | listaDobleDesordenada.java | Operaciones.java | DatosDesordenadosDobleLiga.java | busquedas.java | toolsList.java | test.java

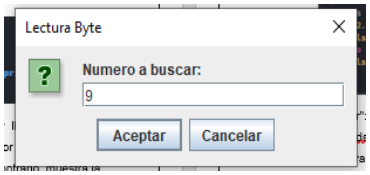
```
1  package main;
2  import tools.*;
7
8
9  public class test {
10     public static void main(String[] args) {
11
12         String menu = "Secuencial Lineal,Secuencial Binaria,Knuth Morris Pratt,Busqueda Salto,Interpolacion,Exponencial,Busqueda Fibonacci,Salir";
13         menu3(menu);
14
15     }
16     public static String boton(String menu) {
17         String valores[]=menu.split(",");
18         int n;
19         n = JOptionPane.showOptionDialog(null," SELECCIONA DANDO CLICK ", " M E N U",
20                 JOptionPane.NO_OPTION,
21                 JOptionPane.QUESTION_MESSAGE,null,
22                 valores,valores[0]);
23         return ( valores[n]);
24     }
25
26     public static void menu3(String menu)
27     {
28         String sel="";
29         do {
30             sel=boton(menu);
31             switch(sel){
32                 case "Secuencial Lineal":
33                     int SL[] = {1,2,3,4,5,6,7,8,9,10};
34                     toolsList.imprimePantalla("Busqueda de un numero del 1 al 10.");
35                     if(busquedas.secuencialLineal(SL, toolsList.leerByte("Numero a buscar:")))
36                         toolsList.imprimePantalla("El elemento existe");
37                     else
38                         toolsList.imprimePantalla("El elemento no existe");
39                     break;
```

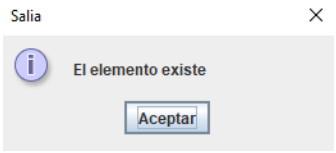
```
                case "Secuencial Binaria":
                    int SB[] = {10,20,1,430,30};
                    toolsList.imprimePantalla(busquedas.imprimeOrdenados(SB));
                    Arrays.sort(SB);
                    toolsList.imprimePantalla("Arreglo ordenado:"+busquedas.imprimeOrdenados(SB));

                    if(busquedas.secuencialBinaria(SB, toolsList.leerInt("Numero a buscar:"))>=0)
                        toolsList.imprimePantalla("El dato existe");
                    else
                        toolsList.imprimePantalla("El dato no existe.");
                    break;
                case "Knuth Morris Pratt":
                    char[] texto = {'A', 'B', 'C', 'D', 'A', 'B', 'C', 'D', 'A', 'B', 'D'};
                    char[] patron = {'D', 'L', 'D'};
                    toolsList.imprimePantalla(busquedas.imprimeOrdenados(texto));
                    toolsList.imprimePantalla("Patron a buscar:" + busquedas.imprimeOrdenados(patron));
                    if(busquedas.KMP(texto,patron)>=0)
                        toolsList.imprimePantalla("El patron existe");
                    else
                        toolsList.imprimePantalla("El patron no existe.");
                    break;
                case "Busqueda Salto":
                    int JS []= {10,13,15,26,28,50,56,88,94,127,159,356,480,567,689,699,780,850,956,995};
                    toolsList.imprimePantalla(busquedas.imprimeOrdenados(JS));
                    if(busquedas.jumpSearch(JS, toolsList.leerInt("Numero a buscar:"))>=0)
                        toolsList.imprimePantalla("El dato existe");
                    else
                        toolsList.imprimePantalla("El dato no existe.");
                    break;
                case "Interpolacion":
                    int[] ITP = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
                    toolsList.imprimePantalla(busquedas.imprimeOrdenados(ITP));
                    if(busquedas.interpolacion(ITP, toolsList.leerInt("Numero a buscar:"))>=0)
                        toolsList.imprimePantalla("El dato existe");
                    else
                        toolsList.imprimePantalla("El dato no existe.");
                    break;
```

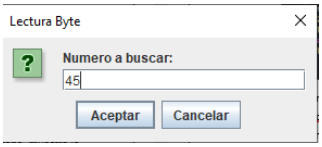
```
77                  case "Exponencial":
78                      int[] EXP = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20,22,24,26};
79                      toolsList.imprimePantalla(busquedas.imprimeOrdenados(EXP));
80                      if(busquedas.interpolacion(EXP, toolsList.leerInt("Numero a buscar:"))>=0)
81                          toolsList.imprimePantalla("El dato existe");
82                      else
83                          toolsList.imprimePantalla("El dato no existe.");
84                      break;
85                  case "Busqueda Fibonacci":
86                      int BF[] = {10,13,15,26,28,50,56,88,94,127,159,356,480,567,689,699,780};
87                      toolsList.imprimePantalla(busquedas.imprimeOrdenados(BF));
88                      if(busquedas.fibonacciSearch(BF, toolsList.leerInt("Numero a buscar:"))>=0)
89                          toolsList.imprimePantalla("El dato existe");
90                      else
91                          toolsList.imprimePantalla("El dato no existe.");
92                      break;
93
94                  case "Salir":;  break;
95              }
96          }while(!sel.equalsIgnoreCase("Salir"));
97      }
98  }
```

búsqueda lineal

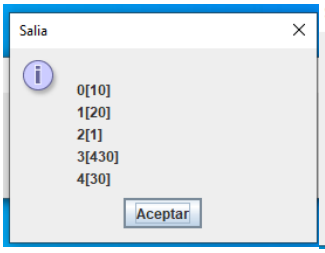


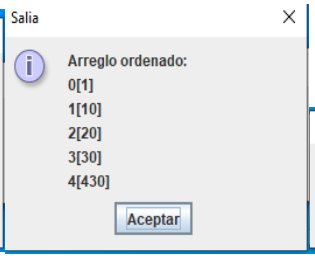


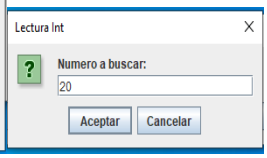


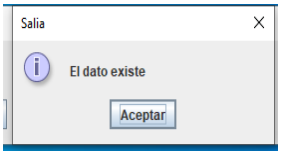


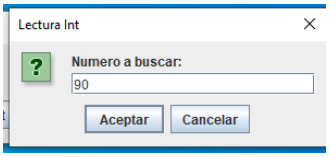
Busqueda binaria

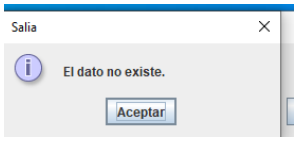




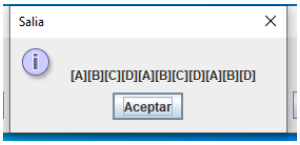


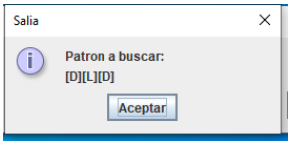


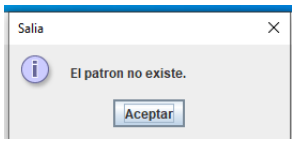




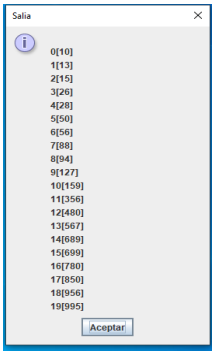
Búsqueda de Knuth Morris Pratt

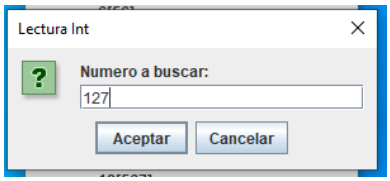


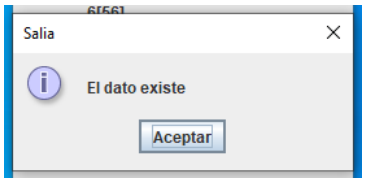


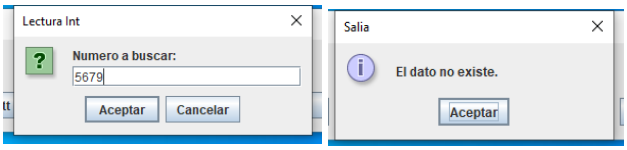


Busqueda Salto

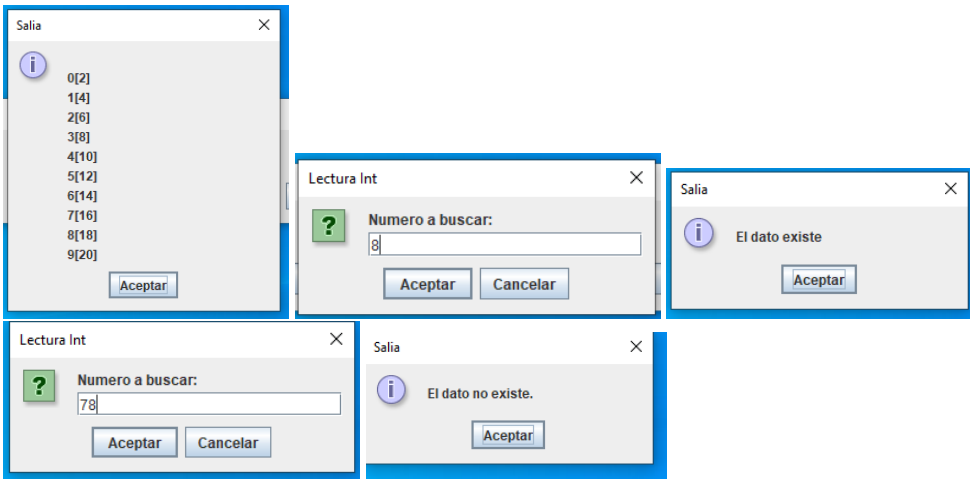




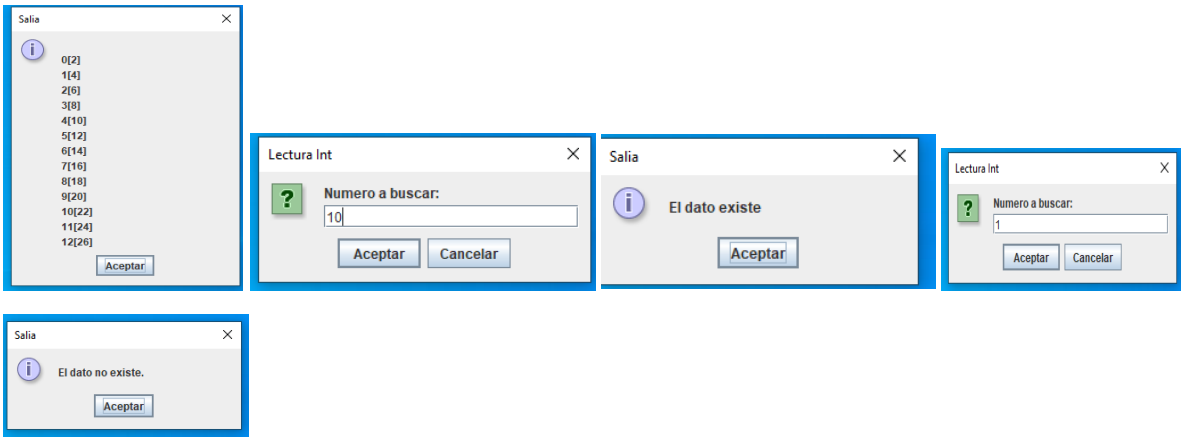




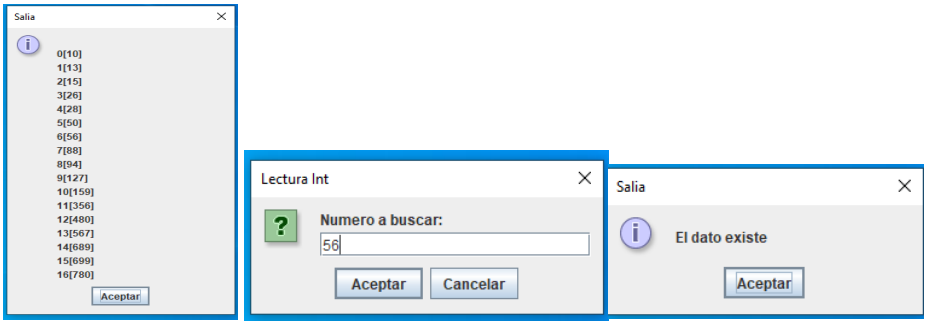
Busqueda interpolacion

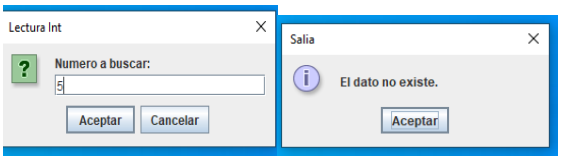


Busqueda exponencial



Busqueda Finonacci





Recursos materiales y equipo.

• Computadora.

• Software IDE Eclipse For Java Developers,

• Lectura de los materiales de apoyo

Conclusión

En el campo de las estructuras de datos, los métodos de búsqueda realizan un papel fundamental. Estos métodos nos permiten buscar y acceder eficientemente a los elementos almacenados en diferentes estructuras, mejorando así el rendimiento y la eficiencia de nuestros programas.

A lo largo de este análisis, hemos explorado varios métodos de búsqueda comunes en estructuras de datos. Estos incluyen la búsqueda lineal, la búsqueda binaria y la búsqueda de fibonacci, entre otros.

Al comprender y aplicar los métodos de búsqueda adecuados en nuestras estructuras de datos, podemos mejorar significativamente el rendimiento y la eficiencia de nuestros programas, lo que se traduce en una mejor experiencia para el usuario final y un uso más eficiente de los del sistema.

Referencias

- Joyanes, Zahonero. Estructura de Datos en C++. McGraw Hill. Madrid, España. 2007. ISBN: 978-84-481-5645-9. • EUÁN AVILA JORGE IVAN Y CORDERO BORBOA LUIS GONZAGA., Estructuras de datos, (1ª reimpresión.), MÉXICO, LIMUSA, tomada de la primera edición de la UNAM (FACULTAD DE INGENIERÍA), 1989.
- (S/f). Dyndns.org. Recuperado el 22 de mayo de 2023, de http://ual.dyndns.org/biblioteca/Estructura%20de%20Datos/Pdf/09%20Metodos%20de%20busqueda.pdf