TECNOLOGICO NACIONAL DE MEXICO

INSTITUTO TECNOLOGICO DE ORIZABA

Asignatura: Estructura de Datos

Carrera: Ingeniería en Informática

Estudiantes:

- Raúl García Jiménez - No. Control 19010405
- Dorantes Rodríguez Diego Yael - No. Control 21010184
- Carlos Eduardo Nicanor Flores- No. Control 21010205
- Axel Reyes Guevara-

Profesora: María Jacinta Martínez Castillo

Grupo: 3a3A

Fecha de entrega: 22/05/2023

Búsqueda Binaria

Investigación:

Introducción.

La búsqueda binaria es un algoritmo de búsqueda eficiente utilizado para encontrar un elemento específico en una lista ordenada. Consiste en dividir repetidamente la lista en dos mitades y descartar la mitad en la que no puede estar el elemento buscado, hasta que se encuentre el elemento deseado o se determine que no está presente.

El proceso comienza comparando el elemento medio de la lista con el valor buscado. Si el elemento medio coincide con el valor buscado, se devuelve su posición. Si el valor buscado es menor que el elemento medio, la búsqueda se realiza en la mitad inferior de la lista. Si el valor buscado es mayor, la búsqueda se realiza en la mitad superior de la lista. Este proceso de dividir y descartar se repite hasta que se encuentre el elemento deseado o se determine que no está presente.

La búsqueda binaria es muy eficiente porque reduce a la mitad el espacio de búsqueda en cada paso. Esto la convierte en una opción ideal cuando se trabaja con grandes conjuntos de datos ordenados. Su complejidad de tiempo es O(log n), donde n es el número de elementos en la lista.

En resumen, la búsqueda binaria consiste en buscar un elemento en una lista ordenada dividiendo repetidamente la lista en dos mitades y descartando la mitad en la que no puede estar el elemento, hasta encontrarlo o determinar que no está presente.

Ejemplo:

Aquí tienes un ejemplo de implementación de búsqueda binaria en Java utilizando un array ordenado:

BusquedaSecuencial1.java | BusquedaSecuencial2.java | BusquedaSecuencial3.java | *BinarySearch.java

```java
package Tarea;

public class BinarySearch {
    public static int binarySearch(int[] arr, int target) {
        int left = 0;
        int right = arr.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (arr[mid] == target) {
                return mid;
            }

            if (arr[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return -1; // El elemento no se encontró
    }

    public static void main(String[] args) {
        int[] array = {1, 3, 5, 7, 9, 11, 13};

        int target = 7;
        int index = binarySearch(array, target);

        if (index != -1) {
            System.out.println("El elemento " + target + " se encuentra en el índice " + index);
        } else {
            System.out.println("El elemento " + target + " no se encontró en el array");
        }
```

En este ejemplo, la función binarySearch realiza la búsqueda binaria en el array ordenado. Comienza definiendo un puntero izquierdo (left) que apunta al primer elemento del array y un puntero derecho (right) que apunta al último elemento del array. Luego, en cada iteración del bucle while, calculamos el índice medio (mid) y comparamos el valor en ese índice con el valor buscado. Si son iguales, se devuelve el índice. Si el valor medio es menor que el objetivo, se actualiza el puntero izquierdo a mid + 1 para buscar en la mitad derecha del array. Si el valor medio es mayor que el objetivo, se actualiza el puntero derecho a mid - 1 para buscar en la mitad izquierda del array. El bucle continúa hasta que se encuentra el elemento deseado o hasta que left es mayor que right, lo que indica que el elemento no se encuentra en el array. Finalmente, en el método main, se crea un array ordenado y se llama a la función binarySearch para buscar el elemento objetivo. Se imprime el resultado según el caso.

Análisis de eficiencia:

La búsqueda binaria es un algoritmo de búsqueda eficiente en estructuras de datos ordenadas. Su eficiencia se basa en la estrategia de dividir y conquistar, que reduce a la mitad el espacio de búsqueda en cada iteración.

La complejidad de tiempo de la búsqueda binaria es O(log n), donde n es el número de elementos en la estructura de datos. Esto significa que el tiempo de ejecución aumenta de forma logarítmica a medida que crece el tamaño de la estructura de datos. En comparación, otros algoritmos de búsqueda lineal tienen una complejidad de tiempo O(n), lo que los hace menos eficientes para grandes conjuntos de datos.

La búsqueda binaria se destaca particularmente cuando se trabaja con estructuras de datos ordenadas, como arrays o listas enlazadas ordenadas. Estas estructuras de datos permiten realizar divisiones eficientes para reducir el espacio de búsqueda.

En resumen, la búsqueda binaria ofrece una eficiencia superior en la búsqueda de elementos en estructuras de datos ordenadas. Su complejidad de tiempo logarítmica O(log n) y su estrategia de dividir y conquistar la convierten en una opción ideal para grandes conjuntos de datos ordenados.

Casos:

La búsqueda binaria tiene diferentes comportamientos según los casos: mejor de los casos, caso medio y peor de los casos. Estos casos se refieren a la ubicación del elemento buscado en la estructura de datos.

- En el mejor de los casos: El elemento buscado se encuentra en el centro de la estructura de datos en cada iteración. Esto permite que la búsqueda binaria encuentre el elemento deseado en el primer intento, lo que resulta en el rendimiento más eficiente.

Por ejemplo, si tenemos un array ordenado [1, 3, 5, 7, 9, 11, 13] y queremos buscar el número 7, la búsqueda binaria lo encontrará en la primera iteración, ya que está en el centro.

- En el caso medio: El elemento buscado no se encuentra en el centro ni en los extremos de la estructura de datos. La búsqueda binaria realizará un número promedio de comparaciones para encontrar el elemento deseado.

Por ejemplo, si tenemos un array ordenado [2, 4, 6, 8, 10, 12, 14] y buscamos el número 10, el elemento buscado no está en el centro pero tampoco en los extremos. La búsqueda binaria realizará un número moderado de comparaciones para encontrarlo.

- En el peor de los casos: El elemento buscado se encuentra en uno de los extremos de la estructura de datos. Esto obliga a la búsqueda binaria a realizar el máximo número de comparaciones posibles antes de encontrar el elemento deseado.

Por ejemplo, si tenemos un array ordenado [1, 3, 5, 7, 9, 11, 13] y buscamos el número 13, que está en el extremo derecho, la búsqueda binaria tendrá que realizar el máximo número de comparaciones para llegar al extremo derecho y encontrar el elemento deseado.

Complejidad en el tiempo y complejidad:

La búsqueda binaria tiene una complejidad en el tiempo de O(log n) en promedio y en el peor de los casos, lo que significa que su tiempo de ejecución aumenta de forma logarítmica a medida que aumenta el tamaño de la estructura de datos. En el mejor de los casos, su complejidad en el tiempo es O(1), lo que la hace muy eficiente.

En cuanto a la complejidad espacial, la búsqueda binaria tiene una complejidad de espacio de O(1), es decir, requiere una cantidad constante de espacio adicional independientemente del tamaño de la estructura de datos. No necesita almacenar información adicional proporcional al tamaño de los datos.

En resumen, la búsqueda binaria es un algoritmo eficiente en términos de tiempo, ya que reduce a la mitad el espacio de búsqueda en cada iteración, lo que la hace especialmente adecuada para estructuras de datos grandes y ordenadas. Además, su complejidad espacial es mínima, ya que no requiere espacio adicional significativo.

Búsqueda Secuencial

Investigación:

Introducción.

La búsqueda secuencial es un algoritmo utilizado en estructuras de datos para encontrar un elemento en una lista. Se realiza recorriendo cada elemento de la lista uno por uno hasta encontrar el valor buscado o llegar al final de la lista. Si se encuentra el elemento, se devuelve su posición; de lo contrario, se indica que el elemento no está presente. Es un método simple pero ineficiente, ya que su rendimiento depende del tamaño de la lista.

Ejemplos:

Implementaciones de búsqueda secuencial en diferentes estructuras de datos en Java: memoria estática, memoria dinámica (listas) y archivos.

1. Búsqueda secuencial en memoria estática (arreglo):

*BusquedaSecuencial1.java

```java
package Tarea;

public class BusquedaSecuencial1 {
    public static int busquedaSecuencial(int[] arreglo, int elemento) {
        for (int i = 0; i < arreglo.length; i++) {
            if (arreglo[i] == elemento) {
                return i; // Se encontró el elemento en la posición i
            }
        }
        return -1; // El elemento no se encuentra en el arreglo
    }

    public static void main(String[] args) {
        int[] arreglo = {2, 5, 8, 10, 15, 20};
        int elementoBuscado = 10;
        int resultado = busquedaSecuencial(arreglo, elementoBuscado);
        if (resultado == -1) {
            System.out.println("El elemento no se encuentra en el arreglo.");
        } else {
            System.out.println("El elemento se encuentra en la posición: " + resultado);
        }
    }
}
```

2. Búsqueda secuencial en memoria dinámica (lista enlazada):

```java
package Tarea;
import java.util.LinkedList;

public class BusquedaSecuencial2 {

    public static int busquedaSecuencial(LinkedList<Integer> lista, int elemento) {
        for (int i = 0; i < lista.size(); i++) {
            if (lista.get(i) == elemento) {
                return i; // Se encontró el elemento en la posición i
            }
        }
        return -1; // El elemento no se encuentra en la lista
    }

    public static void main(String[] args) {
        LinkedList<Integer> lista = new LinkedList<>();
        lista.add(2);
        lista.add(5);
        lista.add(8);
        lista.add(10);
        lista.add(15);
        lista.add(20);
        int elementoBuscado = 10;
        int resultado = busquedaSecuencial(lista, elementoBuscado);
        if (resultado == -1) {
            System.out.println("El elemento no se encuentra en la lista.");
        } else {
            System.out.println("El elemento se encuentra en la posición: " + resultado);
        }
    }
```

3. Búsqueda secuencial en archivos:

```java
package Tarea;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BusquedaSecuencial3 {

    public static int busquedaSecuencial(String archivo, String elemento) {
        try (BufferedReader br = new BufferedReader(new FileReader(archivo))) {
            String linea;
            int posicion = 0;
            while ((linea = br.readLine()) != null) {
                if (linea.equals(elemento)) {
                    return posicion; // Se encontró el elemento en la posición posicion
                }
                posicion++;
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return -1; // El elemento no se encuentra en el archivo
    }

    public static void main(String[] args) {
        String archivo = "datos.txt";
        String elementoBuscado = "Hola";
        int resultado = busquedaSecuencial(archivo, elementoBuscado);
        if (resultado == -1) {
            System.out.println("El elemento no se encuentra en el archivo.");
        } else {
            System.out.println("El elemento se encuentra en la posición: " + resultado);
        }
    }
}
```

En estos ejemplos, se muestra cómo implementar.

Análisis de eficiencia:

La búsqueda secuencial en estructuras de datos tiene una eficiencia lineal, es decir, su tiempo de ejecución aumenta proporcionalmente al tamaño de los datos. En el peor caso, cuando el elemento buscado se encuentra al final de la lista o no está presente, la búsqueda secuencial debe recorrer todos los elementos, lo que implica una complejidad de O(n), donde "n" es el número de elementos en la estructura de datos. Esto significa que el tiempo de ejecución aumenta de manera lineal a medida que la cantidad de datos aumenta.

En el mejor caso, cuando el elemento buscado se encuentra al principio de la lista, la búsqueda secuencial solo necesita realizar una comparación antes de encontrarlo. En este caso, la complejidad es O(1).
La búsqueda secuencial es una opción simple de implementar, pero puede volverse ineficiente para estructuras de datos grandes. Si se requiere una búsqueda más eficiente, especialmente en conjuntos de datos ordenados, se recomienda utilizar algoritmos de búsqueda más avanzados, como la búsqueda binaria, que tiene una complejidad logarítmica (O(log n)) y es más rápida en la mayoría de los casos.

El análisis de los casos mejor, medio y peor de la búsqueda secuencial en estructuras de datos se resume de la siguiente manera:

- Mejor caso: Ocurre cuando el elemento buscado se encuentra al principio de la estructura de datos. En este caso, la búsqueda secuencial solo requiere realizar una comparación y encuentra el elemento de inmediato. La complejidad es O(1). Ejemplo: Buscar el número 2 en un arreglo [2, 5, 8, 10, 15, 20].

- Caso medio: En el caso medio, no se puede predecir la posición del elemento buscado en la estructura de datos. La búsqueda secuencial recorre los elementos uno por uno hasta encontrar una coincidencia o llegar al final de la estructura. La complejidad promedio es O(n/2), que se puede simplificar a O(n), donde "n" es el número de elementos en la estructura de datos. Ejemplo: Buscar el número 10 en un arreglo [2, 5, 8, 10, 15, 20].

- Peor caso: Ocurre cuando el elemento buscado se encuentra al final de la estructura de datos o no está presente en absoluto. En este caso, la búsqueda secuencial debe recorrer todos los elementos hasta llegar al final sin encontrar el elemento. La complejidad es O(n), donde "n" es el número de elementos en la estructura de datos. Ejemplo: Buscar el número 30 en un arreglo [2, 5, 8, 10, 15, 20].

Complejidad en el tiempo y complejidad espacial:
La búsqueda secuencial en estructuras de datos tiene una complejidad en el tiempo de O(n), lo que significa que su tiempo de ejecución aumenta de forma lineal a medida que aumenta el número de elementos en la estructura de datos. La complejidad espacial de la búsqueda secuencial es de O(1), lo que indica que utiliza una cantidad constante de memoria adicional, independientemente del tamaño de los datos. En resumen, la búsqueda secuencial es simple de implementar, pero puede volverse ineficiente en estructuras de datos grandes, ya que su rendimiento está directamente relacionado con la cantidad de elementos en la estructura.

Búsqueda de patrones de Knuth Morris Pratt

En que consiste la búsqueda de patrones de Knuth Morris Pratt

La búsqueda de patrones de Knuth-Morris-Pratt (KMP) es un algoritmo eficiente utilizado para buscar ocurrencias de un patrón específico en una cadena de texto. Fue desarrollado por Donald Knuth, Vaughan Pratt y James Morris en 1977.

El algoritmo KMP se basa en el concepto de evitar comparaciones innecesarias mientras busca el patrón en la cadena. En lugar de hacer coincidir el patrón desde el principio en cada intento, el algoritmo utiliza información previa sobre las coincidencias ya realizadas para evitar comparaciones redundantes.

La principal ventaja del algoritmo KMP es su eficiencia en términos de tiempo de ejecución. La construcción de la tabla de fallos se realiza en O(m), donde m es la longitud del patrón, y la búsqueda del patrón en el texto se realiza en O(n), donde n es la longitud del texto. Por lo tanto, el algoritmo KMP tiene una complejidad total de O(m + n), lo cual lo hace muy eficiente en comparación con otros algoritmos de búsqueda de patrones más ingenuos como la búsqueda exhaustiva.

Ejemplos de implementaciones del algoritmo en las estructuras de datos:

□□***Análisis de eficiencia***

El análisis de eficiencia del algoritmo de búsqueda de patrones de Knuth-Morris-Pratt (KMP) se puede realizar en función de la longitud del texto (n) y la longitud del patrón (m). Veamos el análisis de eficiencia de las implementaciones presentadas.

1. Implementación utilizando arreglos:

- Construcción de la tabla de fallos: La construcción de la tabla de fallos se realiza en un bucle que recorre el patrón una vez. Por lo tanto, tiene una complejidad de tiempo de O(m).
- Búsqueda del patrón en el texto: La búsqueda se realiza en un bucle que recorre el texto una vez y realiza comparaciones entre los caracteres del patrón y el texto. La complejidad de tiempo de este bucle es O(n).

En total, la complejidad de tiempo de esta implementación es O(m + n).

2. Implementación utilizando tablas:

- Construcción de la tabla de fallos: La construcción de la tabla de fallos se realiza en un bucle que recorre el patrón una vez. Tiene una complejidad de tiempo de O(m).
- Búsqueda del patrón en el texto: La búsqueda se realiza en un bucle que recorre el texto una vez y realiza comparaciones entre los caracteres del patrón y el texto. La complejidad de tiempo de este bucle es O(n).

En total, la complejidad de tiempo de esta implementación también es O(m + n).

Ambas implementaciones tienen la misma complejidad de tiempo y ofrecen un rendimiento eficiente. La construcción de la tabla de fallos se realiza en O(m), lo que

es beneficioso si se busca el mismo patrón en múltiples textos, ya que solo se necesita construir la tabla una vez. La búsqueda en sí se realiza en O(n), lo que garantiza una eficiencia lineal en función del tamaño del texto.
En términos de espacio, ambas implementaciones utilizan estructuras de datos adicionales para almacenar la información de la tabla de fallos. La cantidad de espacio adicional utilizado depende de la longitud del patrón (m). Ambas implementaciones utilizan O(m) de espacio adicional.

Análisis de los casos

-Casos peor y mejor para el patrón:

Respecto del patrón, no existe caso mejor ni peor (considerado por sí solo), puede demostrarse que en una tabla de fallos donde son todo ceros (o lo que es lo mismo, el primer carácter del patrón aparece una única vez (por ejemplo P ="ABBBBBBB")), tiene el mismo coste que cuando el patrón se compone de 2 únicos caracteres donde 1 se repite en todo el patrón y el otro aparece al final, (por ejemplo: P= "AAAAAAAB", (obsérvese la tabla F para dicho texto)) y tiene el mismo coste que en cualquier otra situación intermedia entre dichos extremos:

i	0	1	2	3	4	5	6	7
P[i]	A	A	A	A	A	A	A	B
F[i]	-1	0	1	2	3	4	5	6

En el caso de que el primer carácter aparezca una única vez (ver tabla aquí debajo), examina la A, si coincide, examina la B, en caso de fallo simplemente salta 1 carácter. Si el patrón existiere finalmente en el texto, será cuando haya oportunidad de examinar el resto de caracteres, antes de alcanzar el final del texto.

i	0	1	2	3	4	5	6	7
P[i]	A	B	B	B	B	B	B	B
F[i]	-1	0	0	0	0	0	0	0

Igualmente, cuando el primer carácter se repite excepto en el último carácter. Recorrería, hasta el último que genera el fallo, pero igualmente el puntero absoluto del texto, solo avanza 1, con cada fallo.

La diferencia entre uno y otro caso consiste básicamente en donde se localiza el puntero en el patrón y en qué momento se recorre el resto del patrón si al principio o al final del texto.

-Casos mejor y peor para el texto

Para el texto, el mejor caso se da cuando el patrón se localiza en la posición 0. El peor caso se da cuando el patrón se localiza en la última posición. Si el texto no se encuentra, es igual o ligeramente mejor que el peor caso, y es en tal caso dependiente del patrón (ver sección anterior y siguiente).

-Casos peor y mejor en la búsqueda

Considerando conjuntamente el patrón y el texto, el peor caso se puede dar en función de donde y cuantas veces surja la ruptura del patrón. Cada carácter fallido necesita ser comprobado otra vez con aquel con el que se alinea. Esto implica que el peor caso puede llegar a tener un costo de O(n) + O(k*2), si bien en la práctica estos casos son raros, pues no es frecuente que deban realizarse búsqueda con patrones o textos cuyos caracteres tengan una alta frecuencia de repetición (como los que se muestran de ejemplo a continuación) ...

Ejemplos: Patrón (tabla F): Texto = nº de comparaciones precisas (en todos los ejemplos el tamaño del patrón y del texto es igual: 6:30)

AAAAAZ (-1 0 1 2 3 4):AAAAACAAAAACAAAAACAAAAACAAAAAZ = 51

AAAAAZ (-1 0 1 2 3 4):AAAAAAAAAAAAAAAAAAAAAAAAAAAAAZ = 54

ABBBBB (-1 0 0 0 0 0):ABBBBZABBBBZABBBBZABBBBZABBBBB = 35 (coste típico m+n).

Complejidad en el tiempo y complejidad espacial

la complejidad de tiempo del algoritmo KMP es O(m + n), donde m es la longitud del patrón y n es la longitud del texto. Es una complejidad lineal en función del tamaño total de los datos de entrada y ofrece un rendimiento eficiente en la mayoría de los casos prácticos.

la complejidad espacial del algoritmo KMP es O(m), donde m es la longitud del patrón. El espacio adicional requerido por el algoritmo es proporcional a la longitud del patrón utilizado en la búsqueda.

Es importante tener en cuenta que la complejidad en el tiempo y la complejidad espacial mencionadas son para el algoritmo KMP básico. Pueden existir variaciones y optimizaciones del algoritmo que podrían afectar las complejidades en el tiempo y el espacio.

Método Saltar búsqueda

En que consiste la búsqueda en el método Saltar búsqueda

El método "Saltar búsqueda" es una estrategia utilizada en algoritmos de búsqueda para acelerar el proceso de búsqueda al evitar explorar ciertas partes del espacio de soluciones. En lugar de seguir una búsqueda exhaustiva y explorar cada posible solución, el método "Saltar búsqueda" salta ciertos estados o ramas en el árbol de búsqueda.

La idea principal detrás de este método es que, en algunos casos, es posible determinar de antemano que ciertas partes del espacio de soluciones no contienen la solución óptima o no son prometedoras en términos de encontrar una solución aceptable. En lugar de gastar tiempo y recursos explorando esas partes, se toma la decisión de saltarlas y dirigirse directamente a áreas más prometedoras del espacio de búsqueda.

Es importante destacar que el método "Saltar búsqueda" puede ahorrar tiempo y recursos en determinados casos, pero existe el riesgo de que se salte la solución óptima si se toman decisiones incorrectas sobre qué partes omitir. Por lo tanto, es necesario tener cuidado al aplicar este método y considerar las características del problema y la calidad de las soluciones que se desean encontrar.

Ejemplos de implementaciones del algoritmo en las estructuras de datos:

□□*Análisis de eficiencia*

La eficiencia del método "Saltar búsqueda" depende en gran medida de la calidad de las heurísticas o criterios utilizados para determinar qué partes del espacio de soluciones se deben omitir. Si las heurísticas son precisas y pueden identificar correctamente las áreas no prometedoras, el método puede ahorrar una cantidad significativa de tiempo y esfuerzo.

Sin embargo, la eficiencia del método también está sujeta a riesgos. Si las heurísticas son incorrectas o insuficientes, existe la posibilidad de que se omita la solución óptima y se encuentre una solución subóptima o incluso ninguna solución en absoluto.

□□*Análisis de los casos*

Supongamos que tienes que encontrar el número más grande en una lista desordenada de números.

Mejor de los casos:

En este caso, tienes una heurística muy precisa que te permite identificar rápidamente la mitad de la lista que contiene el número más grande. Saltas la otra

mitad de la lista y te enfocas solo en la mitad más grande. Repites el proceso de salto, dividiendo la lista restante en mitades más pequeñas hasta que encuentras el número más grande. Debido a la precisión de la heurística, logras encontrar el número más grande en un tiempo muy corto y evitas explorar la mayoría de los números de la lista.

Caso medio:

En este caso, la heurística que utilizas es generalmente efectiva para identificar la mitad de la lista que contiene el número más grande. Sin embargo, en algunas ocasiones, la heurística puede ser menos precisa y no saltar la mitad correcta de la lista. Aunque se realizan algunos saltos incorrectos y se exploran algunas partes innecesarias de la lista, aún se logra encontrar el número más grande de manera más eficiente que una búsqueda exhaustiva.

Peor de los casos:

En el peor de los casos, la heurística que utilizas es completamente ineficiente y no te proporciona ninguna información útil sobre qué partes de la lista saltar. Como resultado, el método "Saltar búsqueda" no ofrece ninguna ventaja y se vuelve similar a una búsqueda exhaustiva. Tendrías que explorar todos los números de la lista uno por uno hasta encontrar el número más grande, lo que requeriría el mismo tiempo y recursos que una búsqueda lineal tradicional.

Complejidad en el tiempo y complejidad espacial

Complejidad en el tiempo:

La complejidad en el tiempo se refiere a la cantidad de tiempo que tarda el algoritmo en ejecutarse, en función del tamaño del problema. En el caso del método "Saltar búsqueda", la complejidad en el tiempo puede variar según la eficiencia de las heurísticas utilizadas y la cantidad de exploración requerida.

Es difícil proporcionar una complejidad en el tiempo específica para el método "Saltar búsqueda" sin conocer el problema y las heurísticas utilizadas. La complejidad en el tiempo dependerá de cómo se diseñe y aplique el algoritmo en el contexto específico.

Complejidad espacial:

La complejidad espacial del método "Saltar búsqueda" suele ser similar a la de una búsqueda exhaustiva, ya que generalmente se necesita almacenar la misma cantidad de información o estructuras de datos para representar el espacio de soluciones y llevar a cabo la exploración.

Es importante considerar que la complejidad espacial también puede depender de cómo se implemente el método "Saltar búsqueda" y de los requisitos específicos del problema. Por ejemplo, si se utilizan estructuras de datos adicionales para

almacenar información sobre las áreas a omitir, la complejidad espacial puede aumentar en comparación con una búsqueda exhaustiva básica.

En resumen, la complejidad en el tiempo y la complejidad espacial del método "Saltar búsqueda" dependen del problema y de la implementación específica del algoritmo. Se debe tener en cuenta la calidad de las heurísticas utilizadas y cómo se diseñe el algoritmo para determinar con precisión la complejidad en el tiempo y espacial en un contexto particular.

Búsqueda de interpolación

***En que cosiste*:**

La búsqueda de interpolación es un método utilizado para estimar o encontrar un valor desconocido en un conjunto de datos conocidos. Se basa en la idea de que los datos existentes tienen cierta continuidad o patrón, por lo que es posible aproximarse al valor desconocido utilizando información de los datos conocidos que están antes y después del valor buscado.

El objetivo es construir una función o curva suave que se ajuste a los datos conocidos y que pase lo más cercano posible al valor desconocido. Para lograr esto, se utilizan diferentes técnicas de interpolación, como la interpolación lineal, polinómica o mediante splines. La elección del método depende de la naturaleza de los datos y de la precisión deseada.

La interpolación lineal, por ejemplo, establece una línea recta entre los dos puntos conocidos más cercanos al valor buscado. La interpolación polinómica utiliza un polinomio para ajustar los datos conocidos y estimar el valor intermedio. La interpolación spline utiliza funciones suaves que se ajustan a los datos conocidos mediante segmentos polinómicos.

Ejemplos de implementaciones el algoritmo en las estructuras de datos

Ananalisis de eficiencia:

En términos generales, la búsqueda de interpolación tiene una complejidad promedio de O(log n), donde "n" representa la cantidad de datos conocidos. Esto significa que la búsqueda de interpolación tiende a ser más eficiente que la

búsqueda lineal (O(n)), pero menos eficiente que la búsqueda binaria estándar (O(log n)).

el análisis de eficiencia de la búsqueda de interpolación implica considerar factores como la distribución de los datos, el tamaño del rango de búsqueda y la elección de la estructura de datos adecuada. Además, es importante tener en cuenta que el rendimiento real puede variar según el contexto de aplicación y la implementación específica del algoritmo.

Analisis de casos

Mejor de los casos:

En el mejor de los casos, cuando los datos conocidos están uniformemente distribuidos y el valor buscado se encuentra exactamente en el punto medio del rango, la búsqueda de interpolación puede ser altamente eficiente.

En este escenario ideal, la búsqueda de interpolación converge rápidamente al valor buscado con una complejidad de tiempo cercana a O(1). La estimación inicial basada en la interpolación estará muy cerca del valor real debido a la distribución uniforme de los datos. Esto significa que se necesitarán pocas iteraciones para llegar a una aproximación precisa del valor buscado, lo que resulta en un tiempo de ejecución muy rápido.

En términos de eficiencia espacial, la búsqueda de interpolación no requiere ningún espacio adicional más allá del almacenamiento de los datos conocidos. No se necesita crear estructuras de datos adicionales ni realizar un uso adicional de memoria.

Aquí tienes un ejemplo para ilustrar el mejor caso en la búsqueda de interpolación:

Supongamos que tenemos los siguientes datos conocidos:

Datos: [2, 4, 6, 8, 10] Valores: [20, 40, 60, 80, 100]

Queremos encontrar el valor correspondiente a x = 6. Como el valor buscado se encuentra exactamente en uno de los puntos conocidos (x = 6), la búsqueda de interpolación devuelve directamente el valor correspondiente sin necesidad de realizar cálculos adicionales.

En este ejemplo, el valor buscado es 60, ya que coincide con el valor en el punto conocido x = 6. En el mejor caso, la búsqueda de interpolación es extremadamente eficiente, ya que encuentra el resultado en tiempo constante, sin importar la cantidad de datos conocidos.

Caso medio:

En un caso medio, la búsqueda de interpolación puede tener un desempeño eficiente pero no garantiza la máxima eficiencia. La complejidad de tiempo promedio es logarítmica, lo que significa que a medida que aumenta la cantidad de datos conocidos, el tiempo de ejecución aumenta de forma gradual.

En general, se espera que la búsqueda de interpolación realice un número moderado de iteraciones para converger al valor buscado. Sin embargo, la eficiencia puede verse afectada por la distribución de los datos y la posición del valor buscado dentro del rango.

Si los datos están distribuidos de manera uniforme y el valor buscado se encuentra relativamente cerca de los datos conocidos, la búsqueda de interpolación puede converger rápidamente, ya que la estimación inicial estará cercana al valor real. Esto implica que se necesitarán menos iteraciones para obtener una aproximación precisa.

Supongamos que tenemos los siguientes datos conocidos:

Datos: [1, 3, 5, 7, 9] Valores: [10, 20, 30, 40, 50]

Queremos encontrar el valor correspondiente a x = 6. En este caso, el valor buscado no coincide exactamente con ningún punto conocido, por lo que se realiza la interpolación para obtener una estimación precisa.

La búsqueda de interpolación se realiza de la siguiente manera:

1. Se calcula la posición estimada del valor buscado utilizando una fórmula que considera los valores mínimos y máximos del rango y los índices correspondientes. En este caso, el rango es [5, 7] y el valor estimado es:

posición_estimada = (x - x_izq) / (x_der - x_izq) = (6 - 5) / (7 - 5) = 0.5

2. Se busca el intervalo en el que se espera que se encuentre el valor buscado. En este caso, el valor estimado de 0.5 está entre el tercer y el cuarto punto conocido (5 y 7).

3. Se aplica la interpolación lineal entre los valores correspondientes a los puntos conocidos en el intervalo. En este caso, se realiza una interpolación lineal entre los valores 30 y 40, ya que corresponden a los puntos conocidos 5 y 7, respectivamente.

valor_estimado = valor_izq + (valor_der - valor_izq) * posición_estimada
valor_estimado = 30 + (40 - 30) * 0.5 valor_estimado = 30 + 10 * 0.5
valor_estimado = 35

En este ejemplo, el valor buscado estimado para x = 6 es 35.

Peor de los casos:

En el peor de los casos en la búsqueda de interpolación, los datos están distribuidos de tal manera que la interpolación no puede proporcionar estimaciones precisas del valor buscado.

Supongamos que tenemos los siguientes datos conocidos:

Datos: [1, 3, 5, 7, 9] Valores: [10, 20, 30, 40, 50]

Queremos encontrar el valor correspondiente a x = 8. En este caso, el valor buscado se encuentra más allá del último punto conocido, lo que hace que la interpolación sea menos precisa.

La búsqueda de interpolación se realiza de la siguiente manera:

1. Se calcula la posición estimada del valor buscado utilizando una fórmula que considera los valores mínimos y máximos del rango y los índices correspondientes. En este caso, el rango es [7, 9] y el valor estimado es:

posición_estimada = (x - x_izq) / (x_der - x_izq) = (8 - 7) / (9 - 7) = 0.5

2. Se busca el intervalo en el que se espera que se encuentre el valor buscado. En este caso, el valor estimado de 0.5 está más allá del último punto conocido (7).

3. Se intenta aplicar la interpolación lineal entre los valores correspondientes a los puntos conocidos en el intervalo. Sin embargo, no hay más puntos conocidos después del último punto (7), lo que hace que la interpolación sea imposible.

En este ejemplo, en el peor caso, la búsqueda de interpolación no puede proporcionar una estimación precisa del valor buscado, ya que está más allá del rango de los datos conocidos.

Complejidad en el tiempo y complejidad espacial:

La complejidad en el tiempo y la complejidad espacial de la búsqueda de interpolación pueden variar dependiendo de la implementación específica y de las características de los datos y la estructura de datos utilizada.

Complejidad en el tiempo:

- Mejor caso: En el mejor caso, cuando el valor buscado coincide exactamente con uno de los puntos conocidos, la búsqueda de interpolación puede tener una complejidad de tiempo constante, O(1). Esto se debe a que no es necesario realizar iteraciones adicionales ni

aproximaciones, ya que el valor buscado se encuentra directamente en los datos conocidos.

- Caso medio y peor caso: En casos medios y peores, donde se requiere realizar interpolación para obtener una estimación precisa, la complejidad en el tiempo de la búsqueda de interpolación suele ser O(log n), donde n es la cantidad de datos conocidos. Esto se debe a que la búsqueda de interpolación implica comparaciones y cálculos iterativos para estimar el valor buscado utilizando los datos conocidos más cercanos.

Complejidad espacial: La complejidad espacial de la búsqueda de interpolación depende de cómo se almacenen y gestionen los datos. En términos generales, la complejidad espacial puede ser:

- Enfoque basado en arreglos o listas: Si los datos se almacenan en un arreglo o lista, la complejidad espacial es lineal, O(n), donde n es la cantidad de datos conocidos. Esto se debe a que se requiere espacio para almacenar todos los elementos de los datos conocidos.
- Enfoque basado en estructuras de datos avanzadas: Si se utilizan estructuras de datos más complejas, como árboles de búsqueda binaria, tablas de hash o estructuras de datos especializadas para la interpolación, la complejidad espacial puede variar dependiendo de la implementación específica de la estructura de datos utilizada.

Busqueda exponencial

En que consiste:

La búsqueda exponencial, también conocida como búsqueda por salto exponencial o búsqueda binaria exponencial, es un algoritmo de búsqueda utilizado para encontrar un elemento específico en una lista ordenada. A diferencia de la búsqueda binaria estándar, que divide el rango de búsqueda por la mitad en cada iteración, la búsqueda exponencial realiza saltos exponenciales en el rango de búsqueda para encontrar una estimación inicial de la posición del elemento buscado.

La idea clave de la búsqueda exponencial es realizar saltos exponenciales en el rango de búsqueda hasta que se encuentre un límite superior en el que el elemento buscado podría estar presente. Luego, se realiza una búsqueda binaria estándar dentro de ese rango más pequeño para encontrar el elemento de manera eficiente.

Analisis de casos

Mejor de los casos:

En el mejor de los casos, la búsqueda exponencial puede encontrar el elemento buscado en el primer salto exponencial. Esto ocurre cuando el elemento buscado se encuentra en la posición inicial del rango de búsqueda.

Aquí tienes un ejemplo para ilustrar el mejor caso en la búsqueda exponencial:

Supongamos que tenemos la siguiente lista ordenada:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Queremos buscar el elemento 1 en la lista. En este caso, el elemento buscado se encuentra en la posición inicial del rango de búsqueda. La búsqueda exponencial procede de la siguiente manera:

1. Comienza con un rango de búsqueda que abarca toda la lista: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].
2. Compara el elemento buscado (1) con el elemento en la posición inicial del rango de búsqueda (1). Como son iguales, se ha encontrado el elemento y se devuelve su posición.

En este ejemplo, el elemento 1 se encuentra en la posición inicial del rango de búsqueda, por lo que se encuentra en la primera iteración del algoritmo de búsqueda exponencial. El mejor caso en la búsqueda exponencial tiene una complejidad de tiempo constante, O(1), ya que se encuentra el elemento buscado sin necesidad de realizar más iteraciones.

En un caso medio:

En un caso medio de búsqueda exponencial, el elemento buscado se encuentra en una posición dentro del rango de búsqueda, pero no en la posición inicial. Esto significa que se requerirán varios saltos exponenciales y una búsqueda binaria dentro del rango actualizado para encontrar el elemento.

Aquí tienes un ejemplo para ilustrar un caso medio en la búsqueda exponencial:

Supongamos que tenemos la siguiente lista ordenada:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Queremos buscar el elemento 7 en la lista. En este caso, el elemento buscado se encuentra en una posición dentro del rango de búsqueda, pero no en la posición inicial. La búsqueda exponencial procede de la siguiente manera:

1. Comienza con un rango de búsqueda que abarca toda la lista: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].
2. Compara el elemento buscado (7) con el elemento en la posición inicial del rango de búsqueda (1). Como es menor, se actualiza el rango de búsqueda al intervalo [1, 2].
3. Realiza una búsqueda binaria dentro del rango actualizado [1, 2]. El elemento buscado (7) es mayor que el elemento en la posición inicial (1), por lo que se actualiza el rango de búsqueda al intervalo [2].
4. Compara el elemento buscado (7) con el elemento en la posición inicial del rango de búsqueda (2). Como es mayor, se actualiza el rango de búsqueda al intervalo [2, 3].
5. Realiza una búsqueda binaria dentro del rango actualizado [2, 3]. El elemento buscado (7) se encuentra en la posición 3 del rango.

En este ejemplo, el elemento 7 se encuentra en una posición dentro del rango de búsqueda después de realizar varios saltos exponenciales y una búsqueda binaria dentro del rango actualizado.

En el peor de los casos:

En el peor de los casos, la búsqueda exponencial puede requerir un número significativo de saltos exponenciales y búsquedas binarias para encontrar el elemento buscado. Esto ocurre cuando el elemento buscado está cerca del final de la lista ordenada.

Aquí tienes un ejemplo para ilustrar el peor caso en la búsqueda exponencial:

Supongamos que tenemos la siguiente lista ordenada:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Queremos buscar el elemento 10 en la lista. En este caso, el elemento buscado está cerca del final de la lista. La búsqueda exponencial procede de la siguiente manera:

1. Comienza con un rango de búsqueda que abarca toda la lista: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

2. Compara el elemento buscado (10) con el elemento en la posición inicial del rango de búsqueda (1). Como es mayor, se duplica el tamaño del rango de búsqueda y se repite el paso 2.
3. Se actualiza el rango de búsqueda al intervalo [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] y se compara el elemento buscado (10) con el elemento en la posición inicial (1). Como es mayor, se duplica el tamaño del rango de búsqueda nuevamente y se repite el paso 2.
4. El proceso de duplicación del rango de búsqueda se repite varias veces hasta que el rango de búsqueda incluye el elemento buscado 10.
5. Luego, se realiza una búsqueda binaria dentro del rango actualizado para encontrar el elemento buscado.

En este ejemplo, el elemento 10 está cerca del final de la lista, lo que significa que se necesitarán múltiples saltos exponenciales y búsquedas binarias para encontrarlo.

Complejidad en el tiempo y complejidad espacial

Complejidad en el tiempo:

- Mejor caso: En el mejor caso, cuando el elemento buscado se encuentra en la posición inicial del rango de búsqueda, la búsqueda exponencial puede tener una complejidad de tiempo constante, O(1). Esto se debe a que se encuentra el elemento de manera inmediata sin necesidad de realizar iteraciones adicionales.
- Caso medio y peor caso: En casos medios y peores, la complejidad en el tiempo de la búsqueda exponencial es O(log n), donde n es la longitud de la lista. Esto se debe a que se realizan saltos exponenciales y búsquedas binarias dentro del rango actualizado en cada iteración para acercarse al elemento buscado. La búsqueda exponencial sigue dividiendo el rango de búsqueda por la mitad en cada iteración, lo que resulta en una complejidad logarítmica.

Complejidad espacial: La complejidad espacial de la búsqueda exponencial depende de cómo se almacenen los datos y de la implementación específica. En general, la complejidad espacial de la búsqueda exponencial es O(1), es decir, constante. Esto se debe a que no se requiere espacio adicional en función del tamaño de los datos o la longitud de la lista. El algoritmo utiliza solo unas pocas variables para realizar comparaciones y realizar las iteraciones necesarias, sin necesidad de almacenar elementos adicionales de la lista.

Busqueda de fibonacci

En que consiste la búsqueda de Fibonacci

La búsqueda de Fibonacci es un algoritmo utilizado para encontrar un número específico en la secuencia de Fibonacci. La secuencia de Fibonacci es una serie de números en la que cada número es la suma de los dos números anteriores. Comienza con 0 y 1, y luego cada número subsiguiente es la suma de los dos números anteriores.

La búsqueda de Fibonacci se basa en la propiedad de que los números de Fibonacci están ordenados de manera creciente. Esto significa que se puede aplicar una búsqueda binaria para encontrar un número en la secuencia de Fibonacci de manera eficiente.

• Ejemplos de implementaciones del algoritmo en las estructuras de datos:

• Análisis de eficiencia

Para analizar la eficiencia de la búsqueda de Fibonacci, podemos utilizar la notación de O grande (Big O notation). La notación O grande nos permite expresar la complejidad de un algoritmo en términos de su crecimiento relativo en función del tamaño de la entrada.

En el caso de la búsqueda de Fibonacci, la complejidad puede ser analizada de la siguiente manera:

Encontrar el número de Fibonacci más cercano o igual a la longitud del arreglo:

• En el peor caso, el bucle while recorrerá la secuencia de Fibonacci hasta encontrar el número más cercano a la longitud del arreglo.

• La secuencia de Fibonacci crece exponencialmente, pero la relación entre los números consecutivos se acerca a la proporción áurea (aproximadamente 1.618).

• Por lo tanto, la búsqueda del número de Fibonacci más cercano se puede realizar en O(log n), donde n es la longitud del arreglo.

Realizar la búsqueda binaria utilizando los números de Fibonacci:

• En cada iteración del bucle while, se compara el elemento clave con un elemento en el índice de desplazamiento.

• Dado que los números de Fibonacci están ordenados de manera creciente, la búsqueda binaria se realiza en una porción más pequeña del arreglo en cada iteración.

• El bucle while se ejecutará hasta que fibM sea menor o igual a 1, lo que indica que se ha reducido a una porción constante del arreglo.

• En el peor caso, la búsqueda binaria se realiza en O(log n), donde n es la longitud del arreglo.

En general, la eficiencia de la búsqueda de Fibonacci se puede expresar como O(log n), donde n es la longitud del arreglo. Esto significa que la complejidad del algoritmo crece de forma logarítmica en relación al tamaño del arreglo.

En comparación con la búsqueda binaria tradicional, la búsqueda de Fibonacci puede ser ligeramente más lenta debido a los cálculos adicionales para encontrar el número de Fibonacci más cercano. Sin embargo, en casos de arreglos grandes, la diferencia en eficiencia puede no ser significativa.

Es importante tener en cuenta que el análisis de eficiencia es una estimación teórica y puede variar en la práctica según diversos factores, como el rendimiento del hardware y la implementación específica del algoritmo.

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

El análisis de casos para la búsqueda de Fibonacci implica considerar diferentes escenarios en términos de la ubicación del elemento buscado en la secuencia de Fibonacci y el tamaño del arreglo. Aquí están los casos clave a tener en cuenta:

Mejor caso: El elemento buscado está en la primera posición de la secuencia de Fibonacci.

En este caso, la búsqueda se resuelve de inmediato en la primera iteración, ya que el elemento buscado coincide con el elemento en el índice de desplazamiento.

La complejidad de tiempo sería O(1).

Caso medio: El elemento buscado está ubicado en una posición aleatoria dentro del arreglo.

En promedio, la búsqueda de Fibonacci realizará menos comparaciones en comparación con la búsqueda binaria tradicional debido al crecimiento exponencial más lento de la secuencia de Fibonacci.

La complejidad de tiempo sería O(log n), donde n es la longitud del arreglo.

Peor caso: El elemento buscado no está presente en el arreglo o está ubicado en una posición cercana al final del arreglo.

En el peor caso, la búsqueda de Fibonacci realizará comparaciones adicionales en la parte final del arreglo después de haber encontrado el número de Fibonacci más cercano a la longitud del arreglo.

Sin embargo, dado que la búsqueda binaria se realiza en una porción más pequeña del arreglo en cada iteración, la complejidad de tiempo seguirá siendo O(log n).

En términos de eficiencia, la búsqueda de Fibonacci se considera una mejora en comparación con la búsqueda binaria tradicional cuando el tamaño del arreglo es grande. A medida que el tamaño del arreglo aumenta, la diferencia en el número de comparaciones entre ambos métodos disminuye.

Es importante destacar que, aunque la búsqueda de Fibonacci puede ser más eficiente que la búsqueda binaria en algunos casos, existen otras estructuras de datos y algoritmos, como los árboles de búsqueda binaria balanceados (por ejemplo, AVL o árbol rojo-negro), que pueden ofrecer una mejor eficiencia en la búsqueda en general, especialmente para conjuntos de datos más grandes o dinámicos.

• Complejidad en el tiempo

La complejidad temporal de la búsqueda de Fibonacci se puede analizar en función del tamaño del arreglo en el que se realiza la búsqueda. A continuación, se muestra la complejidad en el tiempo en diferentes casos:

Mejor caso: O(1)

En el mejor caso, el elemento buscado coincide con el elemento en el índice de desplazamiento en la primera iteración. Esto significa que se encuentra de inmediato y la búsqueda se resuelve en tiempo constante.

Caso medio: O(log n)

En promedio, la búsqueda de Fibonacci realizará menos comparaciones que la búsqueda binaria tradicional debido al crecimiento exponencial más lento de la secuencia de Fibonacci. A medida que el tamaño del arreglo aumenta, la diferencia en el número de comparaciones entre ambos métodos disminuye.

Peor caso: O(log n)

En el peor caso, la búsqueda de Fibonacci realizará comparaciones adicionales en la parte final del arreglo después de haber encontrado el número de Fibonacci más cercano a la longitud del arreglo. Sin embargo, la búsqueda binaria se sigue realizando en una porción más pequeña del arreglo en cada iteración, lo que mantiene la complejidad en O(log n).

En resumen, la complejidad en el tiempo de la búsqueda de Fibonacci es O(log n), donde n es el tamaño del arreglo en el que se realiza la búsqueda. Esto implica que el tiempo de ejecución del algoritmo aumenta de manera logarítmica a medida que crece el tamaño del arreglo.

• Complejidad espacial

La complejidad espacial de la búsqueda de Fibonacci se refiere a la cantidad de memoria adicional que se requiere para ejecutar el algoritmo. En el caso de la búsqueda de Fibonacci, la complejidad espacial es O(1), es decir, constante.

La razón principal es que no se requiere almacenar ninguna estructura de datos adicional aparte del arreglo en el que se realiza la búsqueda. Las variables utilizadas para realizar los cálculos de los números de Fibonacci y el índice de desplazamiento son de tamaño constante y no dependen del tamaño del arreglo de entrada.

Esto significa que la búsqueda de Fibonacci no requiere asignar memoria adicional a medida que el tamaño del arreglo aumenta. La cantidad de memoria utilizada por el algoritmo se mantiene constante, lo que resulta en una complejidad espacial de O(1).

Es importante tener en cuenta que esta complejidad se refiere específicamente al espacio adicional utilizado por el algoritmo de búsqueda de Fibonacci. No tiene en cuenta el espacio requerido para almacenar el arreglo en sí, ya que ese espacio generalmente se considera parte de la entrada del algoritmo y no es específico de la búsqueda de Fibonacci en sí misma.