Instituto tecnológico de Orizaba

Sistemas Computacionales

Estructura de Datos

Unidad 5

Profesora: María Jacinta Martínez Castillo

Integrantes del equipo:

Moran De la Cruz Aziel 21011006

Jiménez Jiménez Carlos Yael 21010975

Sebastián Brito García 21010929

Bandala Hernández Sebastián 21010921

Introducción

Una pila estática en Java es una estructura de datos que sigue el principio de LIFO (Last In, First Out), lo que significa que el último elemento insertado en la pila es el primero en ser eliminado. A diferencia de una pila dinámica, cuyo tamaño puede cambiar durante la ejecución del programa, una pila estática tiene un tamaño fijo que se determina en tiempo de compilación.

Es importante tener en cuenta que una pila estática tiene una capacidad limitada, determinada por el tamaño del arreglo. Si se intenta insertar un elemento cuando la pila está llena, se produce un desbordamiento de pila ("stack overflow"). Del mismo modo, si se intenta eliminar un elemento cuando la pila está vacía, se produce un subdesbordamiento de pila ("stack underflow").

Competencia especifica

La competencia específica de este programa radica en su eficiencia en el uso de memoria y velocidad de acceso. Su tamaño fijo y asignación de memoria predecible permiten un acceso rápido a los elementos y evitan los costos asociados con la asignación y liberación dinámica de memoria. Sin embargo, su incapacidad para crecer o encogerse dinámicamente puede limitar su utilidad en situaciones donde se requiere una capacidad variable.

Marco teórico

5.1 Algoritmos de ordenamiento interno

Los algoritmos de ordenamiento interno son aquellos que se utilizan para ordenar elementos dentro de una estructura de datos interna, como un arreglo o una lista, en la memoria principal de un programa. Estos algoritmos trabajan directamente con los elementos en la memoria y no requieren acceso a dispositivos externos.

Existen varios tipos de algoritmos de ordenamiento interno, algunos de los más comunes son:

public void burbujaSeñal();

Funcionamiento:

El método utiliza dos bucles anidados para recorrer el arreglo y comparar los elementos adyacentes para realizar los intercambios necesarios hasta que el arreglo esté completamente ordenado.

El bucle externo, controlado por la variable i, se ejecuta desde 0 hasta tope - 1 e indica el número de pasadas a través del arreglo.

El bucle interno, controlado por la variable j, se ejecuta desde 0 hasta tope - i - 1 y realiza las comparaciones y los intercambios entre los elementos adyacentes.

Dentro del bucle interno, se compara si el elemento en la posición j es mayor que el elemento en la posición j + 1. Si la condición se cumple, se realiza un intercambio de elementos utilizando una variable temporal temp.

El proceso de comparación e intercambio continúa en cada iteración hasta que se haya recorrido todo el arreglo en el bucle interno.

Este proceso se repite en cada pasada del bucle externo, pero en cada pasada, el elemento más grande se coloca en la última posición del arreglo.

Una vez que se completa el bucle externo, el arreglo estará ordenado de forma ascendente.

Resultado:

Después de ejecutar el método burbujaSeñal, el arreglo memoria contendrá los mismos elementos que antes, pero en orden ascendente.

public void dobleBurbuja();

Funcionamiento:

El método utiliza dos bucles anidados para recorrer el arreglo y comparar los elementos adyacentes para realizar los intercambios necesarios hasta que el arreglo esté completamente ordenado.

El bucle externo, controlado por la variable i, se ejecuta desde 0 hasta tope - 1 e indica el número de pasadas a través del arreglo.

En cada pasada, se inicializa una variable booleana intercambio como falsa.

El bucle interno, controlado por la variable j, se ejecuta desde 0 hasta tope - i - 1 y realiza las comparaciones y los intercambios entre los elementos adyacentes.

Dentro del bucle interno, se compara si el elemento en la posición j es mayor que el elemento en la posición j + 1. Si la condición se cumple, se realiza un intercambio de elementos utilizando una variable temporal temp, de manera similar al algoritmo de Bubble Sort.

Además de realizar los intercambios, se establece la variable intercambio como verdadera para indicar que se realizó al menos un intercambio en la pasada actual.

Después de completar el bucle interno en cada pasada, se verifica si se realizó algún intercambio durante esa pasada. Si no se realizó ningún intercambio, significa que el arreglo ya está ordenado y se puede finalizar el proceso antes de recorrer todas las pasadas.

Esto se logra mediante la instrucción if (!intercambio) { break; }, que finaliza el bucle externo si no se realizó ningún intercambio en la pasada actual.

Al detener el algoritmo antes de recorrer todas las pasadas cuando no se producen intercambios, se mejora la eficiencia en casos donde el arreglo ya está casi ordenado o cuando se han realizado la mayoría de los intercambios en las primeras pasadas.

Resultado:

Después de ejecutar el método dobleBurbuja, el arreglo memoria contendrá los mismos elementos que antes, pero en orden ascendente.

public void shellIncreDecre();

Funcionamiento:

El método utiliza una secuencia de brechas (gaps) para dividir el arreglo en subarreglos más pequeños y realizar comparaciones e intercambios entre los elementos de esos subarreglos.

Primero, se inicializa una variable n con el valor tope + 1, que representa la longitud del arreglo.

Luego, se calcula el valor inicial de la brecha dividiendo n entre 2.

A continuación, se inicia un bucle mientras la brecha sea mayor que 0.

Dentro del bucle, se ejecuta otro bucle que recorre los elementos del arreglo desde la posición brecha hasta la posición n - 1.

En cada iteración de este bucle interno, se selecciona un elemento temp en la posición i y se establece un índice j igual a i.

Se realiza otro bucle interno mientras j sea mayor o igual que la brecha y el elemento en la posición j - brecha sea mayor que temp.

Dentro de este segundo bucle interno, se mueve el elemento en la posición j - brecha hacia la posición j, y se decrementa el valor de j en brecha.

Una vez que se sale del segundo bucle interno, se coloca el elemento temp en la posición j del arreglo.

Después de completar el segundo bucle interno para todos los elementos en el bucle externo, se reduce la brecha a la mitad dividiéndola por 2.

Esto se logra mediante la instrucción brecha /= 2.

El proceso continúa hasta que la brecha sea menor o igual que 0, momento en el cual el algoritmo habrá ordenado el arreglo.

Resultado:

Después de ejecutar el método shellIncreDecre, el arreglo memoria contendrá los mismos elementos que antes, pero en orden ascendente.

public void seleDirecta();

Funcionamiento:

El método utiliza dos bucles anidados para recorrer el arreglo y seleccionar el elemento mínimo en cada iteración para colocarlo en su posición correcta.

El bucle externo, controlado por la variable i, se ejecuta desde 0 hasta tope - 1 e indica la posición actual desde donde se inicia la búsqueda del elemento mínimo.

En cada iteración del bucle externo, se inicializa la variable minIndex con el valor de i, que representa la posición actual del elemento mínimo.

Luego, se inicia un bucle interno, controlado por la variable j, que se ejecuta desde i + 1 hasta tope e itera sobre los elementos restantes del arreglo.

Dentro del bucle interno, se compara si el elemento en la posición j es menor que el elemento en la posición minIndex. Si la condición se cumple, se actualiza el valor de minIndex con el valor de j, indicando que se ha encontrado un nuevo elemento mínimo.

Después de completar el bucle interno, se intercambia el elemento mínimo encontrado en la posición minIndex con el elemento en la posición i.

Esto se realiza utilizando una variable temporal temp para almacenar temporalmente el valor del elemento mínimo, luego se asigna el valor del elemento en la posición i a la posición minIndex y finalmente se asigna el valor de temp al elemento en la posición i.

El proceso continúa hasta que se complete el bucle externo y se hayan ordenado todos los elementos.

Resultado:

Después de ejecutar el método seleDirecta, el arreglo memoria contendrá los mismos elementos que antes, pero en orden ascendente.

public void inserDirecta();

El bucle for itera desde i = 1 hasta i <= tope. tope probablemente es una variable que indica el índice máximo del arreglo memoria.

Dentro del bucle, se guarda el valor del elemento en la posición i del arreglo memoria en la variable key. Este elemento se considerará el elemento "actual" que se insertará en la posición correcta dentro de la porción ordenada del arreglo.

Se inicializa la variable j con i - 1. Esta variable se utilizará para comparar el elemento actual key con los elementos anteriores en el arreglo.

Se inicia un bucle while que se ejecuta mientras j es mayor o igual a 0 y el elemento en la posición j del arreglo memoria es mayor que key. Esto significa que aún hay elementos en la porción ordenada del arreglo que son mayores que el elemento actual key.

Dentro del bucle while, se desplaza cada elemento mayor que key una posición hacia la derecha, para abrir espacio para insertar key en la posición correcta.

Después de salir del bucle while, se encontró la posición correcta para insertar key. Se coloca key en la posición j + 1 del arreglo memoria.

El bucle for continúa iterando hacia el siguiente elemento en el arreglo memoria, repitiendo los pasos del 2 al 6 para cada elemento.

binaria();

Este método implementa el algoritmo de búsqueda binaria en un array ordenado. La búsqueda binaria divide el array en mitades repetidamente, comparando el elemento buscado con el elemento del medio de cada mitad. Si el elemento buscado es igual al elemento del medio, se retorna su posición. Si es menor, se busca en la mitad inferior del array, y si es mayor, se busca en la mitad superior. Este proceso se repite hasta encontrar el elemento o determinar que no está presente.

heapSort();

Este método implementa el algoritmo de ordenación heap sort. Heap sort utiliza una estructura de datos llamada heap para organizar los elementos. En la primera fase, se construye un heap a partir del array desordenado. Luego, se extrae el elemento máximo (raíz del heap) y se coloca al final del array ordenado. Se reconstruye el heap y se repite el proceso hasta que todos los elementos estén ordenados. Heap sort tiene una complejidad de tiempo de O(n log n) en todos los casos.

quicksortRecursivo();

Este método implementa el algoritmo de ordenación quicksort de manera recursiva. Quicksort elige un elemento llamado pivote y reorganiza los elementos del array de manera que los elementos menores al pivote estén a su izquierda y los elementos mayores estén a su derecha. Luego, se aplica el mismo proceso de forma recursiva a las sub-listas generadas a cada lado del pivote. Quicksort tiene una complejidad de tiempo promedio de O(n log n), pero puede degradarse a O(n^2) en el peor caso.

radix();

Este método implementa el algoritmo de ordenación radix sort. Radix sort ordena los elementos de manera incremental utilizando los dígitos de los números. Comienza por los dígitos menos significativos y va ordenando según cada dígito. En cada pasada, los elementos se agrupan en "contenedores" según el dígito actual. Luego, se combinan los contenedores en orden y se repite el proceso para el siguiente dígito. Radix sort tiene una complejidad de tiempo de O(d * (n + k)), donde d es el número de dígitos, n es el tamaño del array y k es el rango de los valores.

Materiales utilizados

Los materiales utilizados para esta práctica son:

Computadora

NetBeans 8.2

Desarrollo

Al ser un programa hecho con pilas estáticas necesitamos crear un interface que se llamara Memoria TDA la cual contendrá las implementaciones de los métodos que vallamos haciendo en otra clase que se llamara memoriaEstatica, en esta clase haremos un implements del interface Memoria TDA, crearemos nuestro array llamado memoria y crearemos un tope en byte ambos los pondremos en prívate.

Arreglo:

Creamos un arreglo de enteros con tamaño n para almacenar los elementos de la pila, inicializamos el tope de la pila en -1, indicando que la pila está vacía.

Métodos:

public void almacenaAleatorios()

El método utiliza la clase Math.random() para generar un número aleatorio entre 0.0 (inclusive) y 1.0 (exclusive). Luego, realiza algunos cálculos para ajustar ese número al rango especificado por min y max.

(max - min + 1): Calcula el rango de valores posibles.

* Math.random(): Multiplica el rango por un número aleatorio entre 0.0 y 1.0.

+ min: Ajusta el resultado para que esté dentro del rango correcto, sumando el valor mínimo min.

(int): Convierte el resultado final a un número entero truncando cualquier decimal.

public String impresionDatos()

imp: Es un objeto StringBuilder utilizado para construir la cadena de texto que contiene los datos de la pila.

El método recorre el arreglo memoria desde la posición 0 hasta la posición tope (que indica el índice del último elemento en la pila). En cada iteración, agrega el elemento en la posición i al StringBuilder utilizando el método append.

Luego, verifica si i es menor que tope. Si es así, agrega una coma y un espacio al StringBuilder para separar los elementos en la cadena de texto resultante.

Finalmente, se convierte el StringBuilder a una cadena de texto utilizando el método toString y se retorna.

public void burbujaSeñal()

El bucle externo for se encarga de realizar pasadas por la pila desde el primer elemento hasta el penúltimo elemento (i < tope). Cada pasada coloca el elemento más grande en la posición correcta al final de la pila.

El bucle interno for se ejecuta desde el primer elemento hasta el último elemento no ordenado (tope - i). El rango de iteración se reduce en cada pasada, ya que los elementos en las últimas posiciones ya están en su posición correcta.

Dentro del bucle interno, se compara cada elemento con su vecino siguiente. Si el elemento actual es mayor que su vecino, se realiza un intercambio de elementos.

Se utiliza una variable temporal temp para almacenar el valor del elemento actual antes del intercambio.

Después del intercambio, el elemento mayor se mueve hacia la derecha y el elemento menor se mueve hacia la izquierda.

Los bucles for continúan hasta que se hayan realizado todas las comparaciones necesarias para ordenar la pila.

public void dobleBurbuja()

Se declara una variable booleana intercambio que indica si se ha realizado algún intercambio durante una pasada por la lista. Inicialmente se establece en false.

El bucle externo for se encarga de realizar pasadas por la lista desde el primer elemento hasta el penúltimo elemento (i < tope). Cada pasada coloca el elemento más pequeño en la posición correcta al principio de la lista y el elemento más grande en la posición correcta al final de la lista.

Dentro del bucle externo, se reinicia la variable intercambio a false al comenzar una nueva pasada.

El bucle interno for se ejecuta desde el primer elemento hasta el último elemento no ordenado (tope - i). El rango de iteración se reduce en cada pasada, ya que los elementos en las últimas posiciones ya están en su posición correcta.

Dentro del bucle interno, se compara cada elemento con su vecino siguiente. Si el elemento actual es mayor que su vecino, se realiza un intercambio de elementos, al igual que en el algoritmo de burbuja estándar.

Se utiliza una variable temporal temp para almacenar el valor del elemento actual antes del intercambio.

Después del intercambio, el elemento mayor se mueve hacia la derecha y el elemento menor se mueve hacia la izquierda.

Si se realiza algún intercambio durante una pasada, se actualiza la variable intercambio a true.

Al final de cada pasada, se verifica si no se ha realizado ningún intercambio. Si no hay intercambios, significa que la lista está completamente ordenada y se puede finalizar el proceso de ordenamiento antes de completar todas las pasadas.

public void agregarDato(int dato)

El método recibe un parámetro dato que representa el nuevo valor que se desea agregar a la pila.

Primero, se verifica si hay espacio disponible en la pila utilizando el método espacioArray(). Este método debe estar implementado en la clase y debe retornar un valor booleano indicando si la pila tiene capacidad para almacenar más elementos.

Si hay espacio disponible, se ejecuta el bloque de código dentro del if.

Se incrementa el valor de tope en uno. Esto indica que se está agregando un nuevo elemento a la pila, por lo que el puntero tope se desplaza hacia la siguiente posición disponible en el arreglo memoria.

El valor dato se asigna al elemento en la posición indicada por tope en el arreglo memoria. Esto significa que el nuevo dato se agrega al final de la pila.

Si la pila está llena y no hay espacio disponible, se ejecuta el bloque de código dentro del else.

En este caso, se llama al método imprimirErrorMSJE de la clase Tools para mostrar un mensaje de error indicando que la memoria está llena. Este método debe estar implementado en la clase Tools y se encarga de imprimir un mensaje de error en el reporte.

public void shellIncreDecre()

El método utiliza la variable n para almacenar el tamaño de la pila (tope + 1). Esto indica la cantidad de elementos que se van a ordenar.

Se calcula la primera brecha como la mitad del tamaño de la pila (n / 2).

Se inicia un bucle while que se ejecuta mientras la brecha sea mayor que cero.

Dentro del bucle, se realiza una iteración a través de los elementos de la pila, comenzando desde la posición de la brecha y avanzando hasta el final (i = brecha; i < n).

Se guarda el valor del elemento actual en una variable temporal llamada temp.

Se establece una variable j con el valor de i.

Se inicia un bucle while que se ejecuta mientras j sea mayor o igual que la brecha y el elemento en la posición j - brecha sea mayor que temp.

Dentro del bucle while, se realiza un desplazamiento de los elementos hacia la derecha en la pila, asignando el valor del elemento en la posición j - brecha a la posición j. Esto se hace para crear espacio para insertar el valor temporal temp en la posición correcta.

Se actualiza el valor de j disminuyendo la brecha (j -= brecha).

Finalmente, se asigna el valor de temp a la posición j en la pila, colocando el valor en la posición correcta.

Después de finalizar la iteración interna, se reduce la brecha a la mitad (brecha /= 2).

El bucle while se repite hasta que la brecha sea igual a cero, lo que indica que se ha realizado la última pasada y la pila está completamente ordenada.

public void seleDirecta()

El método utiliza un bucle for para iterar sobre los elementos de la pila.

Dentro del bucle externo, se declara una variable minIndex que representa el índice del elemento mínimo en cada iteración. Se inicializa con el valor de i, que es el índice actual.

Se inicia un bucle for interno que comienza desde i + 1 y recorre hasta el valor de tope.

Dentro del bucle interno, se compara el elemento en la posición j con el elemento en la posición minIndex. Si el elemento en la posición j es menor que el elemento en la posición minIndex, se actualiza el valor de minIndex para apuntar al nuevo índice del elemento mínimo.

Después de finalizar el bucle interno, se intercambian los valores entre el elemento en la posición minIndex y el elemento en la posición i. Esto asegura que el elemento mínimo encontrado en cada iteración se coloque en la posición correcta.

El proceso se repite hasta que se hayan recorrido todos los elementos de la pila.

public void inserDirecta()

El método utiliza un bucle for para iterar sobre los elementos de la pila, comenzando desde el índice 1 hasta el valor de tope.

Dentro del bucle, se guarda el valor del elemento actual en la variable key. Esto nos permite trabajar con ese valor y mantenerlo a salvo mientras se desplazan los elementos.

Se establece una variable j que se inicializa con el valor de i - 1. Esta variable se utiliza para comparar el elemento key con los elementos anteriores en la pila.

Se inicia un bucle while que se ejecuta mientras j es mayor o igual a cero y el elemento en la posición j es mayor que key.

Dentro del bucle while, se desplaza los elementos hacia la derecha, asignando el valor del elemento en la posición j a la posición j + 1. Esto crea espacio para insertar el valor key en la posición correcta.

Se decrementa el valor de j para continuar comparando con los elementos anteriores.

Después de finalizar el bucle while, se asigna el valor de key en la posición j + 1. Esto coloca el valor key en la posición correcta dentro de la porción ya ordenada de la pila.

El proceso se repite hasta que se hayan recorrido todos los elementos de la pila.

public void binaria()

El método utiliza un bucle for para iterar sobre los elementos de la pila, comenzando desde el índice 1 hasta el valor de tope.

Dentro del bucle, se guarda el valor del elemento actual en la variable key. Esto nos permite trabajar con ese valor y mantenerlo a salvo mientras se realiza la inserción.

Se establecen tres variables: left, right y mid. left se inicializa en 0 y right se inicializa en i - 1. Estas variables se utilizan para delimitar la porción de la pila en la que se realizará la búsqueda binaria.

Se inicia un bucle while que se ejecuta mientras left sea menor o igual que right.

Dentro del bucle while, se calcula el valor de mid como el promedio de left y right. Esto divide la porción en dos mitades.

Se compara el valor de key con el elemento en la posición mid de la pila. Si key es menor que el elemento en mid, se actualiza right a mid - 1. Si key es mayor o igual que el elemento en mid, se actualiza left a mid + 1.

Después de finalizar el bucle while, se realiza un bucle for que comienza desde i - 1 y se mueve hacia la izquierda hasta left. Esto desplaza los elementos hacia la derecha para crear espacio para insertar key en la posición correcta.

Se asigna el valor de key en la posición left. Esto coloca el valor key en la posición correcta dentro de la porción ya ordenada de la pila.

El proceso se repite hasta que se hayan recorrido todos los elementos de la pila.

public void heapSort()

El método utiliza dos bucles for para realizar el proceso de ordenamiento.

El primer bucle for se utiliza para construir un montículo a partir de los elementos de la pila. Comienza desde tope / 2 y se mueve hacia atrás hasta 0. En cada iteración, se llama al método heapify para ajustar el montículo y asegurar que se cumplan las propiedades de un montículo.

El método heapify se encarga de ajustar el montículo. Recibe dos parámetros: n que representa el tamaño del montículo y i que indica la posición del elemento actual en el montículo. Durante la ejecución de heapify, se realiza un proceso de intercambio entre el elemento en la posición i y sus hijos, si es necesario, para asegurar que el elemento más grande se encuentre en la posición i y se preserven las propiedades del montículo.

Después de construir el montículo, se inicia el segundo bucle for que realiza el proceso de extracción de elementos del montículo y los coloca en la posición correcta en la pila ordenada. El bucle comienza desde tope y se mueve hacia atrás hasta 1.

Dentro del bucle, se intercambia el primer elemento del montículo (que se encuentra en la posición 0) con el último elemento no ordenado de la pila (que se encuentra en la posición i). Luego, se llama a heapify para ajustar el montículo en la porción no ordenada de la pila.

El proceso de intercambio y ajuste del montículo se repite hasta que todos los elementos hayan sido extraídos y colocados en la pila ordenada.

```
private void heapify(int n, int i)
```

El método heapify recibe dos parámetros: n que indica el tamaño del subárbol (o montículo) y i que indica la posición del elemento actual en el subárbol.

Se inicializa la variable largest con el valor de i, que representa la posición del elemento más grande dentro del subárbol actual.

Se calculan las posiciones de los hijos izquierdo y derecho del elemento actual. El hijo izquierdo se encuentra en la posición 2 * i + 1 y el hijo derecho se encuentra en la posición 2 * i + 2.

Se realizan dos comparaciones para determinar si alguno de los hijos es mayor que el elemento en la posición largest. Si el hijo izquierdo (left) es mayor que el elemento en largest, se actualiza largest con la posición del hijo izquierdo. Si el hijo derecho (right) es mayor que el elemento en largest, se actualiza largest con la posición del hijo derecho.

Después de determinar la posición del elemento más grande en el subárbol, se realiza una comparación final entre largest y i. Si son diferentes, significa que se encontró un elemento mayor en algún hijo, por lo que se realiza un intercambio entre el elemento en i y el elemento en largest. Esto asegura que el elemento más grande se encuentre en la posición i.

Luego, se llama recursivamente a heapify en el subárbol correspondiente a la posición largest. Esto garantiza que cualquier cambio realizado mantenga la propiedad del montículo en todo el subárbol.

public void quicksortRecursivo()

El método quicksortRecursivo invoca al método quicksort pasando como argumentos los índices de inicio y fin de la sección que se desea ordenar. En este caso, los índices son 0 y tope, que representan toda la pila.

El método quicksort implementa el algoritmo de ordenamiento rápido. Este algoritmo divide la sección en subsecciones más pequeñas y luego las ordena recursivamente.

En cada llamada recursiva del método quicksort, se selecciona un elemento llamado "pivote" de la sección. El pivote puede ser cualquier elemento de la sección, pero en este caso se utiliza el primer elemento (índice 0) como pivote.

A continuación, se realiza una partición en la sección de modo que todos los elementos menores que el pivote se coloquen a la izquierda del pivote y todos los elementos mayores se coloquen a la derecha.

Después de la partición, el pivote se encuentra en su posición final y todos los elementos a su izquierda son menores o iguales, mientras que todos los elementos a su derecha son mayores.

Luego, se llama recursivamente al método quicksort para ordenar las subsecciones izquierda y derecha del pivote por separado. Esto se realiza hasta que las subsecciones tengan un solo elemento, lo que significa que están ordenadas.

private void quicksort(int low, int high)

El método quicksort recibe dos parámetros: low y high, que representan los índices de inicio y fin de la sección que se desea ordenar en la pila estática.

Se verifica si low es menor que high. Si esta condición se cumple, significa que la sección contiene más de un elemento y se puede proceder con el algoritmo de ordenamiento rápido.

En cada llamada recursiva de quicksort, se selecciona un elemento llamado "pivote" de la sección. La elección del pivote puede variar, pero en este caso no se especifica. Por lo general, se elige el último elemento de la sección.

A continuación, se llama al método partition para realizar una partición en la sección. El método partition reorganiza los elementos de la sección de modo que todos los elementos menores que el pivote se coloquen a la izquierda del pivote y todos los elementos mayores se coloquen a la derecha. Además, devuelve la posición final del pivote después de la partición.

Después de la partición, el pivote se encuentra en su posición final y todos los elementos a su izquierda son menores o iguales, mientras que todos los elementos a su derecha son mayores.

Luego, se realiza una llamada recursiva a quicksort para ordenar las subsecciones izquierda y derecha del pivote por separado. La subsección izquierda tiene un rango de low a pi - 1, y la subsección derecha tiene un rango de pi + 1 a high.

Este proceso recursivo continúa hasta que las subsecciones tengan un solo elemento, lo que significa que están ordenadas.

private int partition(int low, int high)

El método partition recibe dos parámetros: low y high, que representan los índices de inicio y fin de la sección que se desea particionar en la pila estática.

El método selecciona un elemento como pivote, que en este caso es el elemento en la posición high.

Se inicializa una variable i con el valor de low - 1. Esta variable se utiliza para rastrear la posición del último elemento menor que el pivote durante el proceso de partición.

Se inicia un bucle for que recorre los elementos de low a high - 1. Para cada elemento en esta sección, se compara con el pivote.

Si un elemento es menor que el pivote, se incrementa i en 1 y se realiza un intercambio entre el elemento en la posición i y el elemento en la posición j. Esto garantiza que todos los elementos menores que el pivote queden a la izquierda de i.

Después de que el bucle for ha recorrido todos los elementos de la sección, se realiza un último intercambio entre el elemento en la posición i + 1 (la posición siguiente a la última posición de un elemento menor) y el pivote. Esto coloca el pivote en su posición final después de la partición.

Finalmente, se devuelve el valor de i + 1, que representa la posición final del pivote después de la partición.

public void radix()

El método radix comienza encontrando el valor máximo (max) y el valor mínimo (min) dentro de la pila estática. Estos valores se utilizan para determinar el número de dígitos necesarios para el ordenamiento.

A continuación, se inicia un bucle externo que se ejecuta mientras el cociente entre (max - min) y exp sea mayor a 0. La variable exp se inicializa en 1 y se multiplica por 10 en cada iteración. Esta variable representa el exponente utilizado para extraer los dígitos de los números en la pila.

Dentro del bucle externo, se llama al método countSort pasando como argumentos exp y min. El método countSort se encarga de ordenar los elementos de la pila en función del dígito correspondiente a la posición exp de cada número.

El proceso de ordenamiento se repite en cada iteración del bucle externo, aumentando el valor de exp y utilizando el siguiente dígito de los números para el ordenamiento.

Después de que se completan todas las iteraciones del bucle externo, la pila estará ordenada de forma ascendente.

private void countSort(int exp, int min)

El método countSort recibe dos parámetros: exp y min. exp representa el exponente utilizado para extraer el dígito de cada número, y min es el valor mínimo en la pila.

Se crea un arreglo output del mismo tamaño que la pila para almacenar los elementos ordenados temporalmente.

Se crea un arreglo count de tamaño 10 para realizar el conteo de la frecuencia de los dígitos.

Se inicializa el arreglo count con ceros utilizando el método Arrays.fill(count, 0).

Se recorre la pila y se incrementa la posición correspondiente en el arreglo count para cada dígito. La fórmula ((memoria[i] - min) / exp) % 10 calcula el dígito en la posición exp de cada número y se utiliza como índice en el arreglo count.

A continuación, se realiza una modificación en el arreglo count. Cada elemento en el arreglo se suma con el elemento anterior para obtener las posiciones finales de los dígitos en el arreglo output. Esto se realiza para garantizar que los elementos con el mismo dígito se coloquen en orden relativo en el arreglo output.

Se recorre la pila en sentido inverso y se coloca cada elemento en su posición correspondiente en el arreglo output. La fórmula ((memoria[i] - min) / exp) % 10 se utiliza nuevamente para determinar la posición en output basada en el dígito en la posición exp del número.

Después de colocar todos los elementos en el arreglo output, se copian nuevamente en la pila memoria en el mismo orden.

private int getMax()

Se inicializa una variable max con el primer elemento de la pila (memoria[0]).

Se recorre la pila desde el segundo elemento hasta el último (i = 1 hasta tope).

En cada iteración, se compara el elemento actual (memoria[i]) con el valor almacenado en max.

Si el elemento actual es mayor que max, se actualiza el valor de max con el elemento actual.

Al finalizar el recorrido de la pila, el valor almacenado en max será el máximo valor encontrado.

El método retorna el valor máximo encontrado.

private int getMin()

Se inicializa una variable min con el primer elemento de la pila (memoria[0]).

Se recorre la pila desde el segundo elemento hasta el último (i = 1 hasta tope).

En cada iteración, se compara el elemento actual (memoria[i]) con el valor almacenado en min.

Si el elemento actual es menor que min, se actualiza el valor de min con el elemento actual.

Al finalizar el recorrido de la pila, el valor almacenado en min será el mínimo valor encontrado.

El método retorna el valor mínimo encontrado.

public void intercalacion()

Se obtiene el tamaño de la pila más 1 y se almacena en la variable n.

Se crea un arreglo auxiliar del mismo tamaño que la pila, llamado auxiliar, que servirá para almacenar los elementos durante el proceso de intercalación.

Se llama al método intercalacionRecursiva() pasando como argumentos el índice inicial (0), el índice final (n - 1) y el arreglo auxiliar.

El método intercalacionRecursiva() se encargará de realizar la intercalación recursiva de los elementos de la pila.

private void intercalacionRecursiva(int izq, int der, int[] auxiliar)

Se verifica si el índice izquierdo (izq) es menor que el índice derecho (der). Esta condición asegura que hay elementos en el rango a ordenar.

Si se cumple la condición anterior, se calcula el índice medio (medio) como el promedio de izq y der.

Se llama recursivamente al método intercalacionRecursiva() con los siguientes argumentos:

Para el primer llamado recursivo: izq como índice inicial, medio como índice final y el mismo arreglo auxiliar (auxiliar).

Para el segundo llamado recursivo: medio + 1 como índice inicial, der como índice final y el mismo arreglo auxiliar (auxiliar).

Después de los llamados recursivos, se llama al método merge() para intercalar y combinar los elementos en el rango especificado (izq a der), utilizando el arreglo auxiliar (auxiliar).

private void merge(int izq, int medio, int der, int[] auxiliar)

Se crea una copia de los elementos en el rango especificado (izq a der) en el arreglo auxiliar (auxiliar). Esto se realiza para preservar los valores originales durante el proceso de intercalación.

Se inicializan tres índices:

i representa el índice actual del primer subrango (desde izq hasta medio).

j representa el índice actual del segundo subrango (desde medio + 1 hasta der).

k representa el índice actual en el rango original (izq a der) donde se insertarán los elementos combinados.

Se realiza un bucle mientras i sea menor o igual que medio y j sea menor o igual que der:

Se compara el valor en auxiliar[i] con el valor en auxiliar[j].

Si auxiliar[i] es menor o igual que auxiliar[j], se copia auxiliar[i] en la posición k de la memoria original y se incrementa i.

Si auxiliar[i] es mayor que auxiliar[j], se copia auxiliar[j] en la posición k de la memoria original y se incrementa j.

Se incrementa k después de copiar el elemento.

Después de salir del bucle anterior, puede haber elementos restantes en uno de los subrangos. Se realiza otro bucle para copiar los elementos restantes del subrango que no se haya completado en la memoria original. Esto se hace para garantizar que todos los elementos se copien correctamente.

public void mezclaDirecta()

Se obtiene el tamaño del arreglo a ordenar, que es igual a tope + 1, y se crea un arreglo auxiliar llamado auxiliar del mismo tamaño.

Se llama al método mezclaDirectaRecursiva() para iniciar el proceso de ordenamiento pasando los siguientes parámetros:

0 como el límite izquierdo del rango a ordenar.

n - 1 como el límite derecho del rango a ordenar.

auxiliar como el arreglo auxiliar que se utilizará durante el proceso de mezcla.

El método mezclaDirectaRecursiva() se encarga de realizar el ordenamiento mediante la técnica de mezcla directa.

Dentro del método mezclaDirectaRecursiva(), se verifica si el límite izquierdo (izq) es menor que el límite derecho (der). Si esta condición es verdadera, significa que hay al menos dos elementos en el rango y se puede realizar el ordenamiento.

Se calcula el punto medio (medio) del rango dividiendo la suma de izq y der entre 2.

Se llama recursivamente al método mezclaDirectaRecursiva() para ordenar la mitad izquierda del rango, pasando como parámetros:

izq como límite izquierdo.

medio como límite derecho.

auxiliar como arreglo auxiliar.

Se llama recursivamente al método mezclaDirectaRecursiva() para ordenar la mitad derecha del rango, pasando como parámetros:

medio + 1 como límite izquierdo.

der como límite derecho.

auxiliar como arreglo auxiliar.

Una vez que se han ordenado las mitades izquierda y derecha del rango, se llama al método merge() para combinar ambas mitades en un solo rango ordenado.

Al finalizar el proceso de mezcla, la memoria contendrá los elementos ordenados en el rango completo desde izq hasta der

private void mezclaDirectaRecursiva(int izq, int der, int[] auxiliar)

El método recibe tres parámetros: izq que representa el límite izquierdo del rango a ordenar, der que representa el límite derecho del rango a ordenar, y auxiliar que es un arreglo auxiliar utilizado durante el proceso de mezcla.

El método verifica si el límite izquierdo izq es menor que el límite derecho der. Si esta condición es verdadera, significa que hay al menos dos elementos en el rango y se puede realizar el ordenamiento.

Se calcula el punto medio medio del rango dividiendo la suma de izq y der entre 2.

Se realiza una llamada recursiva al método mezclaDirectaRecursiva() para ordenar la mitad izquierda del rango, pasando como parámetros:

izq como límite izquierdo.

medio como límite derecho.

auxiliar como arreglo auxiliar.

Se realiza otra llamada recursiva al método mezclaDirectaRecursiva() para ordenar la mitad derecha del rango, pasando como parámetros:

medio + 1 como límite izquierdo.

der como límite derecho.

auxiliar como arreglo auxiliar.

Después de que ambas mitades hayan sido ordenadas recursivamente, se llama al método mergeSort() para combinar las dos mitades ordenadas en un solo rango ordenado.

private void mergeSort(int izq, int medio, int der, int[] auxiliar)

El método recibe cuatro parámetros: izq que representa el límite izquierdo del primer subrango, medio que indica el límite derecho del primer subrango y el límite izquierdo del segundo subrango, der que representa el límite derecho del segundo subrango, y auxiliar que es un arreglo auxiliar utilizado durante el proceso de mezcla.

Se calcula el tamaño de cada subrango:

n1 representa el tamaño del primer subrango y se obtiene restando izq a medio y sumando 1.

n2 representa el tamaño del segundo subrango y se obtiene restando medio a der.

Se crean dos arreglos auxiliares: izquierda y derecha, con tamaños n1 y n2 respectivamente, para almacenar los elementos de cada subrango.

Se copian los elementos correspondientes del arreglo memoria a los arreglos izquierda y derecha.

En el caso de izquierda, se copian los elementos desde izq hasta medio.

En el caso de derecha, se copian los elementos desde medio + 1 hasta der.

Se inicializan tres índices: i para recorrer izquierda, j para recorrer derecha, y k para recorrer el rango original en memoria.

Se realiza la mezcla ordenada de los dos subrangos:

Mientras ambos subrangos tengan elementos por comparar (es decir, i es menor que n1 y j es menor que n2), se compara el elemento actual de izquierda con el elemento actual de derecha.

Si el elemento de izquierda es menor o igual al elemento de derecha, se coloca el elemento de izquierda en la posición k del rango original en memoria, se incrementa i y k.

Si el elemento de izquierda es mayor al elemento de derecha, se coloca el elemento de derecha en la posición k de memoria, se incrementa j y k.

Este proceso se repite hasta que se hayan recorrido todos los elementos de uno de los subrangos.

Si quedan elementos en el subrango izquierda que no se han copiado a memoria, se copian en las posiciones restantes de memoria.

Si quedan elementos en el subrango derecha que no se han copiado a memoria, se copian en las posiciones restantes de memoria.

Ahora tenemos que crear una clase donde se ejecutara todo con sus respectivos tools, vamos a utilizar una interfaz grafica como menú, En la clase interfaz grafica creamos un JFrame un botón de aceptar y una lista desplegable de las opciones a elegir a la hora de elegir esta opción pasa por un if y else que comparan la opción que elegimos para compararlo con el resto y que método vamos a ocupar

Creamos un objeto de la clase memoria estática con un límite de 10 valores e iniciamos con los 15 diferentes métodos de funcionalidades y dependiendo la opción elegida se ejecutara en su momento (básicamente es un menú con case pero con métodos)

Análisis de Ejecución

public void burbujaSeñal();

Análisis de eficiencia:

Complejidad temporal en el peor de los casos: O(n^2) El algoritmo de ordenación de burbuja tiene una complejidad cuadrática en el peor de los casos. Esto se debe a que hay dos bucles anidados: el bucle externo ejecuta "tope" veces, y el bucle interno ejecuta "tope - i" veces en cada iteración del bucle externo. En cada iteración, el algoritmo compara elementos adyacentes y realiza un intercambio si es necesario. Por lo tanto, en el peor de los casos, el número total de comparaciones y asignaciones realizadas será aproximadamente (tope * (tope - 1)) / 2, lo que da como resultado una complejidad de O(n^2).

Complejidad temporal en el mejor de los casos: O(n) En el mejor de los casos, cuando el arreglo "memoria" ya está ordenado, el algoritmo de burbuja aún recorrerá todos los elementos en los bucles anidados, pero no realizará ningún intercambio, ya que la condición "memoria[j] > memoria[j + 1]" nunca se cumplirá. Esto significa que el número de comparaciones y asignaciones será igual al número de elementos en el arreglo, es decir, "tope". Por lo tanto, en el mejor de los casos, la complejidad temporal del algoritmo es O(n).

Complejidad espacial: O(1) El algoritmo de burbuja no utiliza memoria adicional que crezca con el tamaño de los datos de entrada. Solo se utiliza una cantidad constante de memoria para mantener variables temporales y contadores. Por lo tanto, la complejidad espacial del algoritmo es O(1), lo que significa que el uso de memoria es constante.

Análisis de los casos:

Mejor caso:

Descripción: El mejor caso ocurre cuando el arreglo ya está completamente ordenado.

Complejidad temporal: O(n)

Explicación: En el mejor caso, el bucle externo se ejecutará "tope" veces, pero no se realizarán intercambios en el bucle interno. Esto se debe a que los elementos ya están en el orden correcto. Por lo tanto, la complejidad temporal será lineal, ya que se recorren todos los elementos una vez, pero no se realizan operaciones de intercambio.

Peor caso:

Descripción: El peor caso ocurre cuando el arreglo está en orden descendente o completamente desordenado.

Complejidad temporal: O(n^2)

Explicación: En el peor caso, el bucle externo se ejecutará "tope" veces, y en cada iteración, el bucle interno se ejecutará "tope - i" veces. Esto significa que se realizarán un número total de comparaciones y asignaciones aproximadamente igual a (tope * (tope - 1)) / 2, lo que da como resultado una complejidad cuadrática.

Caso promedio:

Descripción: El caso promedio se refiere a una distribución aleatoria de los elementos en el arreglo.

Complejidad temporal: O(n^2)

Explicación: En promedio, el algoritmo de burbuja realizará un número cuadrático de comparaciones y asignaciones. Esto se debe a que, en cada iteración, los elementos se comparan y se intercambian si es necesario. Aunque algunos pares de elementos pueden estar en el orden correcto, otros requerirán intercambios, lo que da como resultado una complejidad cuadrática en promedio.

public void dobleBurbuja();

Análisis de eficiencia:

Complejidad temporal en el peor de los casos: O(n^2) Al igual que el algoritmo de burbuja tradicional, el algoritmo de doble burbuja tiene una complejidad cuadrática en el peor de los casos. Esto se debe a que hay dos bucles anidados: el bucle externo ejecuta "tope" veces, y el bucle interno ejecuta "tope - i" veces en cada iteración del bucle externo. En cada iteración, el algoritmo compara elementos adyacentes y realiza un intercambio si es necesario. En el peor de los casos, el número total de comparaciones y asignaciones será aproximadamente (tope * (tope - 1)) / 2, lo que da como resultado una complejidad de O(n^2).

Complejidad temporal en el mejor de los casos: O(n) Al igual que el algoritmo de burbuja tradicional, en el mejor de los casos, cuando el arreglo "memoria" ya está ordenado, el algoritmo de doble burbuja realizará solo una pasada a través del arreglo sin realizar ningún intercambio. Esto significa que el número total de

comparaciones y asignaciones será igual al número de elementos en el arreglo, es decir, "tope". Por lo tanto, en el mejor de los casos, la complejidad temporal del algoritmo es O(n).

Complejidad espacial: O(1) Al igual que el algoritmo de burbuja tradicional, el algoritmo de doble burbuja no utiliza memoria adicional que crezca con el tamaño de los datos de entrada. Solo se utiliza una cantidad constante de memoria para mantener variables temporales y contadores. Por lo tanto, la complejidad espacial del algoritmo es O(1), lo que significa que el uso de memoria es constante.

Análisis de los casos:

Mejor caso:

Descripción: El mejor caso ocurre cuando el arreglo ya está completamente ordenado.

Complejidad temporal: O(n)

Explicación: En el mejor caso, el algoritmo de doble burbuja realizará una pasada a través del arreglo sin realizar ningún intercambio. Esto se debe a que los elementos ya están en el orden correcto. Por lo tanto, la complejidad temporal será lineal, ya que se recorren todos los elementos una vez, pero no se realizan operaciones de intercambio.

Peor caso:

Descripción: El peor caso ocurre cuando el arreglo está en orden descendente o completamente desordenado.

Complejidad temporal: O(n^2)

Explicación: En el peor caso, el bucle externo se ejecutará "tope" veces, y en cada iteración, el bucle interno se ejecutará "tope - i" veces. Esto significa que se realizarán un número total de comparaciones y asignaciones aproximadamente igual a (tope * (tope - 1)) / 2, lo que da como resultado una complejidad cuadrática.

Caso promedio:

Descripción: El caso promedio se refiere a una distribución aleatoria de los elementos en el arreglo.

Complejidad temporal: O(n^2)

Explicación: En promedio, el algoritmo de doble burbuja realizará un número cuadrático de comparaciones y asignaciones. Esto se debe a que, en cada iteración, los elementos se comparan y se intercambian si es necesario. Aunque algunos pares de elementos pueden estar en el orden correcto, otros requerirán intercambios, lo que da como resultado una complejidad cuadrática en promedio.

public void shellIncreDecre();

Análisis de eficiencia:

Complejidad temporal en el peor de los casos: O(n^2) El algoritmo de Shell Sort tiene una complejidad cuadrática en el peor de los casos. El rendimiento exacto depende del patrón de incremento y decremento utilizado. En este caso, se utiliza un patrón de brecha decreciente. El algoritmo realiza múltiples pasadas a través del arreglo, donde en cada pasada se comparan y se realizan intercambios en subarreglos separados por la brecha. A medida que la brecha se va reduciendo, el tamaño de los subarreglos también se reduce, lo que permite realizar comparaciones y asignaciones más eficientes. Sin embargo, en el peor de los casos, el algoritmo puede requerir un número cuadrático de comparaciones y asignaciones.

Complejidad temporal en el mejor de los casos: O(n log n) El algoritmo de Shell Sort tiene una complejidad promedio de O(n log n) en el mejor de los casos. Aunque el rendimiento exacto depende del patrón de incremento y decremento utilizado, en general, Shell Sort tiene una mejor eficiencia que los algoritmos de ordenación cuadráticos, como Bubble Sort o Selection Sort. El uso de subarreglos y la disminución gradual de la brecha permiten reducir la cantidad de comparaciones y asignaciones necesarias en cada pasada.

Complejidad espacial: O(1) El algoritmo de Shell Sort no utiliza memoria adicional que crezca con el tamaño de los datos de entrada. Solo se utiliza una cantidad constante de memoria para mantener variables temporales y contadores. Por lo tanto, la complejidad espacial del algoritmo es O(1), lo que significa que el uso de memoria es constante.

Análisis de los casos:

Mejor caso:

Descripción: El mejor caso ocurre cuando el arreglo ya está completamente ordenado.

Complejidad temporal: Dependiente del patrón de incremento y decremento utilizado.

Explicación: En el mejor caso, el algoritmo de Shell Sort puede tener un rendimiento cercano a lineal (O(n)), dependiendo del patrón de incremento y decremento utilizado. Si el patrón es adecuado y las brechas permiten una reducción eficiente de los subarreglos, el algoritmo puede tener un mejor rendimiento en comparación con otros algoritmos de ordenación cuadráticos. Sin embargo, el rendimiento exacto dependerá del patrón específico utilizado.

Peor caso:

Descripción: El peor caso ocurre cuando el arreglo está en orden descendente o cuando el patrón de incremento y decremento no es eficiente.

Complejidad temporal: O(n^2)

Explicación: En el peor caso, el algoritmo de Shell Sort puede requerir un número cuadrático de comparaciones y asignaciones. Si el patrón de incremento y decremento no permite una reducción efectiva de los subarreglos o el arreglo está en un orden desfavorable, el algoritmo puede comportarse de manera similar a un algoritmo de ordenación cuadrático, como Bubble Sort o Insertion Sort.

Caso promedio:

Descripción: El caso promedio se refiere a una distribución aleatoria de los elementos en el arreglo y un patrón de incremento y decremento eficiente.

Complejidad temporal: Dependiente del patrón de incremento y decremento utilizado.

Explicación: En promedio, el algoritmo de Shell Sort puede tener una complejidad mejor que cuadrática (mejor que O(n^2)), pero aún puede ser menos eficiente en comparación con otros algoritmos de ordenación más avanzados. La eficiencia promedio dependerá del patrón de incremento y decremento utilizado, así como de la distribución de los elementos en el arreglo.

public void seleDirecta()

Análisis de eficiencia:

El bucle externo "for" se ejecuta "tope" veces, donde "tope" representa la longitud de la matriz.

Dentro del bucle externo, hay otro bucle "for" que también se ejecuta "tope" veces.

En cada iteración del bucle interno, se realiza una comparación y una asignación en el peor de los casos.

Por lo tanto, el número total de comparaciones y asignaciones realizadas por el algoritmo es aproximadamente (tope^2) / 2.

La eficiencia temporal (complejidad temporal) del algoritmo de selección directa es O(n^2), donde "n" es el tamaño de la matriz. Esto significa que el tiempo de ejecución del algoritmo aumenta cuadráticamente con el tamaño de la matriz. Para una matriz grande, el algoritmo puede ser relativamente lento.

Análisis de casos:

Mejor caso: En el mejor caso, la matriz ya está ordenada. Sin embargo, incluso en el mejor caso, el algoritmo realiza comparaciones en todas las iteraciones del bucle interno. Aunque no hay necesidad de realizar intercambios, el algoritmo sigue siendo ineficiente en términos de comparaciones.

Peor caso: En el peor caso, la matriz está ordenada en orden descendente. Esto significa que se deben realizar comparaciones y asignaciones en todas las iteraciones del bucle interno, y se deben realizar intercambios en todas las iteraciones del bucle externo. El peor caso ocurre cuando la matriz está completamente desordenada.

Caso promedio: En promedio, el algoritmo de selección directa requiere (tope^2) / 2 comparaciones y asignaciones, independientemente de la distribución inicial de los elementos en la matriz.

public void inserDirecta()

Análisis de eficiencia:

El bucle externo "for" se ejecuta "tope" veces, donde "tope" representa la longitud de la matriz.

Dentro del bucle externo, hay un bucle "while" que se ejecuta en el peor de los casos hasta que "j" llegue a cero.

En cada iteración del bucle interno, se realiza una asignación en el peor de los casos.

Por lo tanto, el número total de asignaciones realizadas por el algoritmo es aproximadamente (tope^2) / 2.

La eficiencia temporal (complejidad temporal) del algoritmo de inserción directa es O(n^2), donde "n" es el tamaño de la matriz. Esto significa que el tiempo de ejecución del algoritmo aumenta cuadráticamente con el tamaño de la matriz. Para una matriz grande, el algoritmo puede ser relativamente lento.

Análisis de casos:

Mejor caso: En el mejor caso, la matriz ya está ordenada. En este caso, el bucle interno "while" no se ejecuta y el algoritmo realiza una sola comparación en cada iteración del bucle externo. Por lo tanto, en el mejor caso, el algoritmo puede tener una complejidad temporal de O(n) debido al bucle externo. Sin embargo, todavía se realizan asignaciones en todas las iteraciones del bucle externo, lo que da como resultado una complejidad promedio de O(n^2).

Peor caso: En el peor caso, la matriz está ordenada en orden descendente. Esto significa que se deben realizar comparaciones y asignaciones en todas las iteraciones del bucle interno, y se deben realizar asignaciones en todas las iteraciones del bucle externo. El peor caso ocurre cuando la matriz está completamente desordenada.

Caso promedio: En promedio, el algoritmo de inserción directa requiere (tope^2) / 2 asignaciones y un número variable de comparaciones dependiendo de la distribución inicial de los elementos en la matriz.

binaria();

Análisis de eficiencia:

En general, la búsqueda binaria es un algoritmo eficiente para buscar elementos en un array ordenado, ofreciendo tiempos de ejecución rápidos en la mayoría de los casos y una complejidad espacial mínima. Sin embargo, es importante tener en cuenta que el array debe estar previamente ordenado para que la búsqueda binaria funcione correctamente.

Análisis de casos:

1. Mejor de los casos: En el mejor de los casos, el elemento buscado se encuentra exactamente en el medio del array ordenado. En este escenario, la búsqueda binaria tiene un rendimiento óptimo. La búsqueda se realiza dividiendo el array por la mitad en cada paso, lo que resulta en un tiempo de ejecución eficiente. La complejidad de tiempo en el mejor caso es O(1), ya que se puede encontrar el elemento en el primer intento.

   Ejemplo: Supongamos que tenemos el siguiente array ordenado: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]. Si queremos buscar el número 50, la búsqueda binaria encontrará el número en el primer intento y devolverá su posición.

2. Caso medio: En el caso medio, el elemento buscado no se encuentra necesariamente en el medio del array, pero tampoco es el peor caso. La búsqueda binaria sigue dividiendo el array en mitades sucesivas hasta encontrar el elemento buscado. En promedio, la búsqueda binaria realiza log2(n) comparaciones, donde n es el tamaño del array. Por lo tanto, la complejidad de tiempo en el caso medio es O(log n).

   Ejemplo: Consideremos el mismo array ordenado: [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]. Si buscamos el número 70, la búsqueda binaria realizará aproximadamente log2(10) ≈ 3 comparaciones antes de encontrar el número.

3. Peor de los casos: En el peor de los casos, el elemento buscado no está presente en el array o se encuentra en la posición más baja o más alta. En este escenario, la búsqueda binaria realizará log2(n) comparaciones hasta llegar al final del array sin encontrar el elemento. La complejidad de tiempo en el peor caso también es O(log n).

Complejidad en el tiempo y complejidad espacial:

La complejidad espacial, la búsqueda binaria no requiere memoria adicional más allá del array original. Solo se necesitan variables adicionales para realizar las comparaciones y realizar el seguimiento de los índices de inicio y fin del espacio de búsqueda. Por lo tanto, la complejidad espacial es constante O(1).

Su complejidad en el tiempo varía dependiendo del caso, pero sigue siendo muy eficiente en comparación con otros algoritmos de búsqueda. Además, su complejidad espacial es constante, lo que lo hace adecuado para manejar grandes conjuntos de datos sin consumir mucha memoria adicional.

public void heapSort()

Análisis de eficiencia:

Heap sort es un algoritmo de ordenación eficiente que ofrece un buen rendimiento en la mayoría de los casos. A diferencia de otros algoritmos de ordenación, como quicksort o mergesort, heap sort no tiene casos particulares que afecten su rendimiento. La complejidad en el tiempo es consistente en todos los casos, lo que garantiza que el tiempo de ejecución sea predecible.

Análisis de casos:

Mejor de los casos, caso medio y peor de los casos:

Para el algoritmo de ordenación heap sort, no hay diferencia en el rendimiento entre los casos mejor, medio y peor. La complejidad de tiempo y la complejidad espacial son las mismas en todos los casos.

Complejidad en el tiempo y complejidad espacial:

Complejidad en el tiempo: Heap sort tiene una complejidad de tiempo de O(n log n) en todos los casos, donde n es el tamaño del array a ordenar. Esto significa que el tiempo de ejecución aumenta de manera logarítmica en función del tamaño del array.

Complejidad espacial: Heap sort tiene una complejidad espacial de O(1), lo que significa que no se necesita memoria adicional más allá del array original. Las operaciones de ordenación se realizan directamente en el array existente, sin requerir estructuras de datos adicionales.

quicksortRecursivo();

Análisis de eficiencia:

Quicksort es un algoritmo de ordenación eficiente en la mayoría de los casos, pero puede tener un rendimiento deficiente en el peor caso. Es ampliamente utilizado debido a su eficiencia promedio y su simplicidad de implementación. Sin embargo, en situaciones en las que el peor caso es una preocupación, se pueden considerar otros algoritmos como mergesort o heapsort, que tienen un rendimiento más predecible.

Análisis de casos:

Mejor de los casos: En el mejor de los casos, el pivote seleccionado divide el array en dos subarrays de igual tamaño (o aproximadamente igual). Esto significa que en cada llamada recursiva, el tamaño del array se reduce a la mitad. En este caso, el quicksort tiene un rendimiento óptimo. La complejidad de tiempo en el mejor caso es O(n log n), donde n es el tamaño del array.

Ejemplo: Supongamos que tenemos el siguiente array desordenado: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Si aplicamos el método quicksortRecursivo() a este array, el pivote puede ser el número 5. El array se dividirá en dos subarrays de igual tamaño: [1, 2, 3, 4] y [6, 7, 8, 9, 10]. Luego, se aplicará quicksort recursivamente en cada subarray.

Caso medio: En el caso medio, el pivote seleccionado divide el array de manera que los subarrays resultantes tengan tamaños aproximadamente iguales. Sin embargo, no se garantiza que sean exactamente de la mitad del tamaño del array. En promedio, quicksort recursivo tiene una complejidad de tiempo de O(n log n).

Ejemplo: Consideremos el mismo array desordenado: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Si aplicamos el método quicksortRecursivo() a este array, el pivote puede ser el número 7. El array se dividirá en dos subarrays: [1, 2, 3, 4, 5, 6] y [8, 9, 10]. Luego, se aplicará quicksort recursivamente en cada subarray.

Peor de los casos: En el peor de los casos, el pivote seleccionado siempre es el elemento más pequeño o el más grande del array. Esto puede ocurrir cuando el array está ordenado en orden ascendente o descendente. En este escenario, el quicksort tiene un rendimiento deficiente y puede degenerar en una complejidad de tiempo cuadrática O(n^2).

Complejidad en el tiempo y complejidad espacial:

La complejidad en el tiempo indica cuántas operaciones se necesitan para completar el algoritmo en función del tamaño del array a ordenar. En el mejor y caso medio, quicksort tiene una complejidad de tiempo de O(n log n). Esto significa que el tiempo de ejecución aumenta de manera logarítmica en función del tamaño del array.

La complejidad espacial se refiere a la cantidad de memoria adicional que se requiere para ejecutar el algoritmo. En el caso de quicksort recursivo, la complejidad espacial es determinada por la profundidad de la pila de llamadas recursivas. En promedio, la profundidad máxima de la pila de llamadas recursivas es O(log n), lo que implica que la complejidad espacial también es O(log n).

radix();

Análisis de eficiencia:

Radix sort es un algoritmo de ordenación estable que se basa en la ordenación de los elementos por dígitos. Es eficiente en términos de complejidad en el tiempo, ya que su rendimiento está linealmente relacionado con el tamaño del array y el número de dígitos. Sin embargo, la complejidad espacial puede aumentar si la base utilizada o los valores posibles de los dígitos son grandes. En general, radix sort es una opción adecuada cuando se trabaja con números enteros y se conoce el rango de valores posibles.

Análisis de casos:

Mejor de los casos, caso medio y peor de los casos: En el caso de radix sort, no hay una diferencia significativa en el rendimiento entre los casos mejor, medio y peor. La complejidad de tiempo y la complejidad espacial son las mismas en todos los casos.

Complejidad en el tiempo y complejidad espacial:

Complejidad en el tiempo: La complejidad de tiempo de radix sort es O(d * (n + b)), donde d es el número de dígitos en los números a ordenar, n es el tamaño del array y b es la base utilizada para representar los números. En general, la complejidad de tiempo de radix sort es lineal en función del tamaño del array y el número de dígitos.

Complejidad espacial: La complejidad espacial de radix sort es O(n + b), donde n es el tamaño del array y b es la base utilizada. La complejidad espacial es

determinada por el tamaño del array y los valores posibles que pueden tener los dígitos en la base utilizada.

intercalación();

Análisis de eficiencia:

La mezcla simple (intercalación) es un algoritmo de ordenación simple pero eficiente, especialmente cuando se trabaja con arrays pequeños o cuando se requiere una implementación sencilla. La complejidad en el tiempo es lineal en función del tamaño del array, lo que garantiza un rendimiento aceptable en la mayoría de los casos. La complejidad espacial también es lineal, ya que se necesita un array adicional del mismo tamaño que el array original para almacenar los elementos intercalados en el orden correcto.

Análisis de casos:

Mejor de los casos, caso medio y peor de los casos: En el caso de la mezcla simple (intercalación), no hay una diferencia significativa en el rendimiento entre los casos mejor, medio y peor. La complejidad de tiempo y la complejidad espacial son las mismas en todos los casos.

Complejidad en el tiempo y complejidad espacial:

Complejidad en el tiempo: La complejidad de tiempo de la mezcla simple es O(n), donde n es el tamaño del array. En cada paso de la intercalación, se comparan los elementos de los dos subarrays y se colocan en orden en un nuevo array resultante. Esto implica recorrer todos los elementos una vez, lo que tiene una complejidad lineal en función del tamaño del array.

Complejidad espacial: La complejidad espacial de la mezcla simple es O(n), donde n es el tamaño del array. Se requiere un array adicional para almacenar los elementos intercalados en el orden correcto.

mezclaDirecta();

Análisis de eficiencia:

La mezcla directa es un algoritmo de ordenación eficiente que utiliza el enfoque de "divide y conquista". La complejidad en el tiempo es logarítmica en función del tamaño del array, lo que garantiza un rendimiento aceptable incluso en casos grandes. La complejidad espacial es lineal, ya que se necesita un array adicional del mismo tamaño que el array original para almacenar los elementos mezclados. Sin embargo, es importante tener en cuenta que la recursión utilizada en el algoritmo puede ocupar espacio adicional en la pila de llamadas recursivas.

Análisis de casos:

Mejor de los casos, caso medio y peor de los casos: En el caso de la mezcla directa, no hay una diferencia significativa en el rendimiento entre los casos mejor, medio y peor. La complejidad de tiempo y la complejidad espacial son las mismas en todos los casos.

Complejidad en el tiempo y complejidad espacial:

Complejidad en el tiempo: La complejidad de tiempo de la mezcla directa es O(n log n), donde n es el tamaño del array. En cada paso de la mezcla, el array se divide en mitades y se mezclan en orden. Luego, se repite el proceso de manera recursiva hasta que los subarrays tengan solo un elemento. La operación de mezcla tiene una complejidad de tiempo logarítmica, ya que se realizan comparaciones y asignaciones en función del tamaño de los subarrays.

Complejidad espacial: La complejidad espacial de la mezcla directa es O(n), donde n es el tamaño del array. Durante la mezcla, se necesita un array adicional del mismo tamaño que el array original para almacenar los elementos mezclados en el orden correcto. Además, se puede utilizar recursión para dividir y mezclar los subarrays, lo que puede requerir espacio adicional en la pila de llamadas recursivas.

generaRandom(int min, int max);

Análisis de eficiencia:

El método generaRandom(int min, int max) tiene una eficiencia alta, con una complejidad en el tiempo y una complejidad espacial constantes (O(1)). Genera un número aleatorio dentro de un rango dado de forma rápida y sin dependencia del tamaño de los datos o de otros factores.

Análisis de casos:

El método generaRandom(int min, int max) no es un algoritmo de ordenación, sino una función que genera un número aleatorio dentro de un rango específico. Por lo tanto, no tiene casos de mejor, medio y peor caso, ni una complejidad en el tiempo y complejidad espacial en el sentido tradicional de los algoritmos de ordenación.

Complejidad en el tiempo y complejidad espacial:

La complejidad de tiempo y complejidad espacial del método generaRandom() dependen en gran medida de la implementación específica de la función de generación de números aleatorios utilizada. En general, la complejidad de tiempo y espacial de una función de generación de números aleatorios es bastante baja y se considera constante, independientemente del rango de números a generar.

La complejidad en el tiempo y espacial puede estar influenciada por el algoritmo o la fuente de generación de números aleatorios utilizada, como el generador lineal congruencial (GLC) o el algoritmo de Mersenne Twister. Estos algoritmos suelen tener una complejidad de tiempo y espacial constante, es decir, O(1).

5.2 Algoritmos de ordenamiento externos:

Llamamos ordenamiento externo cuando debemos ordenar archivos que son (mucho) más grandes de lo que nuestra memoria puede llegar a abarcar. Esto es así porque, en caso contrario, podríamos simplemente cargar el archivo en memoria y ordenarlo usando cualquier algoritmo de ordenamiento convencional.
Supongamos entonces que contamos con un caso así. Estamos queriendo ordenar un archivo completamente desordenado, bajo algún criterio que no es importante. El algoritmo que podríamos utilizar podría hacer algo como:

1. Generar k particiones ordenadas. Una partición de un archivo es un sub-archivo, donde todos juntos tienen todas las líneas ó registros del archivo original, sin repeticiones. Cada archivo tendrá n_i registros (con i entre 0 y $k-1$), donde la suma de todos los n_i será n, la cantidad total de registros del archivo original.
2. Juntar las k particiones ordenadas en un nuevo archivo ordenado. Esto sería una generalización del *intercalar ordenado* de mergesort.

5.2.1 Intercalación:

En este método de ordenamiento existen dos archivos con llaves ordenadas, los cuales se mezclan para formar un solo archivo.
La longitud de los archivos puede ser diferente.
El proceso consiste en leer un registro de cada archivo y compararlos, el menor es almacenando en el archivo de resultado y el otro se compara con el siguiente elemento del archivo si existe. El proceso se repite hasta que alguno de los archivos quede vacío y los elementos del otro archivo se almacenan directamente en el archivo resultado.

Mezcla Natural: El método de ordenamiento por mezcla natural, también conocido como ordenamiento por fusión natural, es un algoritmo utilizado para ordenar listas o arreglos de elementos. A diferencia de otros métodos de ordenamiento, la mezcla natural no se basa en comparaciones directas entre los elementos. En cambio, aprovecha la estructura de las listas o arreglos para realizar el ordenamiento de manera eficiente.

El proceso de ordenamiento por mezcla natural se basa en el concepto de "mezcla" de dos o más sublistas ordenadas para formar una lista más grande y ordenada. El algoritmo funciona dividiendo la lista original en sublistas ordenadas de manera ascendente y luego fusionándolas hasta obtener una lista completamente ordenada.

El ordenamiento por mezcla natural se puede describir en los siguientes pasos:

- División: La lista original se divide en sublistas ordenadas. Cada sublista consiste en elementos contiguos que están en orden ascendente.
- Fusión: Se seleccionan dos sublistas adyacentes y se fusionan en una lista más grande y ordenada. Este proceso se repite hasta que todas las sublistas estén fusionadas en una sola lista ordenada.
- Repetición: Los pasos de división y fusión se repiten hasta que la lista completa esté ordenada.

Una característica importante de la mezcla natural es que se aprovecha la secuencia ordenada de elementos en la lista original. Si la lista ya está parcialmente ordenada, el algoritmo puede realizar el ordenamiento de manera eficiente, evitando comparaciones innecesarias entre elementos.

Mezcla Directa: El método de ordenamiento por mezcla directa, también conocido como ordenamiento por mezcla top-down, es otro algoritmo que se basa en la técnica de mezcla para ordenar elementos. A diferencia de la mezcla natural, la mezcla directa no requiere que la lista original tenga una estructura parcialmente ordenada.

El proceso de ordenamiento por mezcla directa se puede describir en los siguientes pasos:

- División: La lista original se divide en sublistas más pequeñas de manera recursiva hasta que cada sublista contenga un solo elemento.
- Fusión: Las sublistas individuales se fusionan en sublistas más grandes y ordenadas. Durante la fusión, se comparan los elementos de las sublistas y se colocan en orden ascendente.
- Repetición: Los pasos de división y fusión se repiten hasta que todas las sublistas estén fusionadas en una sola lista ordenada.

El ordenamiento por mezcla directa utiliza una estrategia "divide y conquista" para ordenar la lista. Al dividir la lista original en sublistas más pequeñas, se facilita la tarea de ordenamiento, ya que se manejan conjuntos más pequeños de elementos en cada iteración. Luego, las sublistas ordenadas se fusionan para obtener una lista ordenada final.

La mezcla directa es un método estable y eficiente para ordenar elementos, aunque puede requerir un espacio adicional en memoria para almacenar las sublistas durante el proceso de fusión. Sin embargo, debido a su naturaleza recursiva, puede requerir más tiempo de ejecución en comparación con otros algoritmos de ordenamiento más eficientes en algunos casos.

Conclusión

En conclusión, los métodos de ordenamiento en Java son herramientas fundamentales para organizar eficientemente elementos en estructuras de datos. Estos métodos ofrecen diferentes algoritmos que permiten ordenar elementos de manera ascendente o descendente según ciertos criterios establecidos.

Los métodos de ordenamiento en Java, como el algoritmo de ordenamiento de burbuja, de inserción, de selección, de mezcla, de quicksort, entre otros, ofrecen diferentes enfoques y estrategias para lograr el ordenamiento de los elementos. Cada algoritmo tiene su propio rendimiento en términos de tiempo de ejecución y complejidad, por lo que es importante seleccionar el método más adecuado según el tamaño de la lista y los requisitos de rendimiento del programa.