ESAT
INNOVATION SCHOOL
ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

Algoritmos

ESAT INNOVATION SCHOOL | ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

Jose Maria Maestre Quiles & Hector Ochando
Algoritmos
Curso 2022/2023
Ivan Sancho

Índice/Index

ESAT INNOVATION SCHOOL | ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

1.- Stack

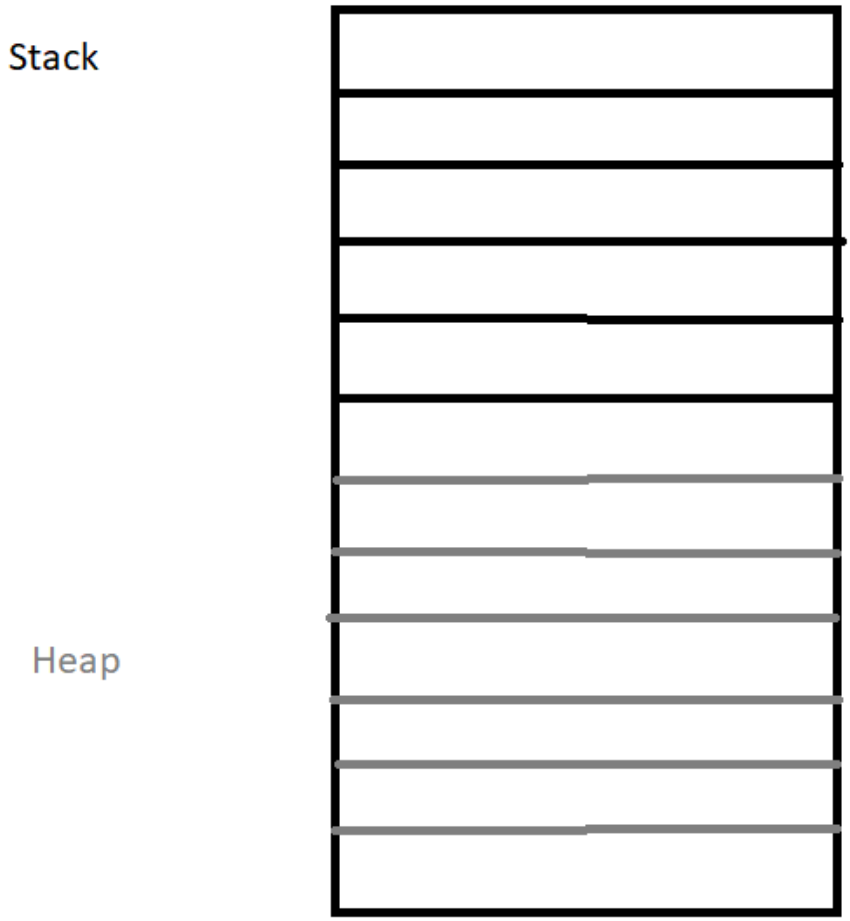
1.1.- ¿Qué es?

El stack es una zona de memoria donde se almacenan las variables locales automáticas. Esta zona de memoria se sitúa en las direcciones más altas. Es una memoria temporal, donde se almacenan las variables locales de las funciones llamadas. Para llamar a las funciones se hace “call” Cuando llamas a una función se crea un nuevo “frame” en el stack y se hace “push” de las variables locales de la función. Post cada función que se llama se crea un nuevo frame pointer. Cuando esta función acaba se hace un “pop” de las variables para liberar la memoria. Cuando esto ocurre se libera la memoria de de las variables y se elimina el frame pointer de esa función.
El stack también tiene un puntero especial que es el stack pointer, este puntero se encarga de delimitar el final de la memory stack.

La memory stack actúa como una lifo, es decir que cuando se llama(call) una función se hace un push de sus variables y se ejecuta, cuando acaba de ejecutarse se desapila la función haciendo pop de sus variables y se vuelve al anterior salto. Esto quiere decir que cuando se llama una función no se puede continuar la ejecución de la anterior hasta que se termine la primera. En el caso de que se llamen a muchas funciones una dentro de otra el stack crecerá mucho.

1.2.- Ejemplo gráfico

Tenemos la memoria Stack y a continuación tenemos el heap

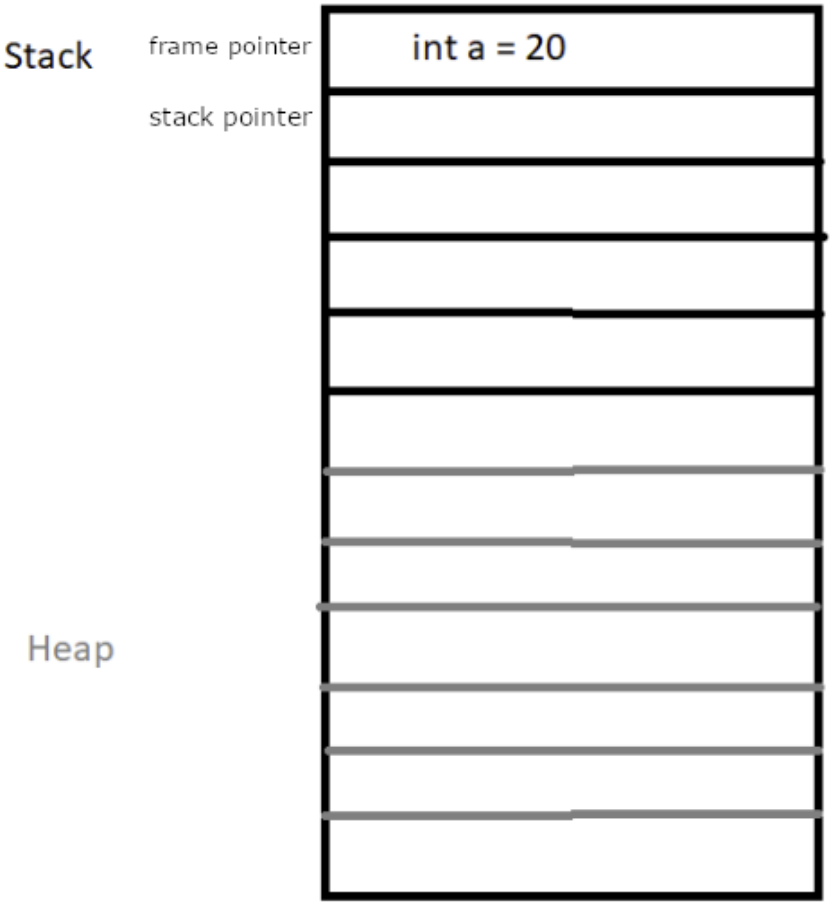


Cuando creamos una variable dentro de una función, esta se almacena en el stack

main.cc

main.cc > main()

```
1   #include <stdio.h>
2
3
4   int main(){
5
6     int a;
7     a = 20;
8
9
10    return 0;
11  }
```

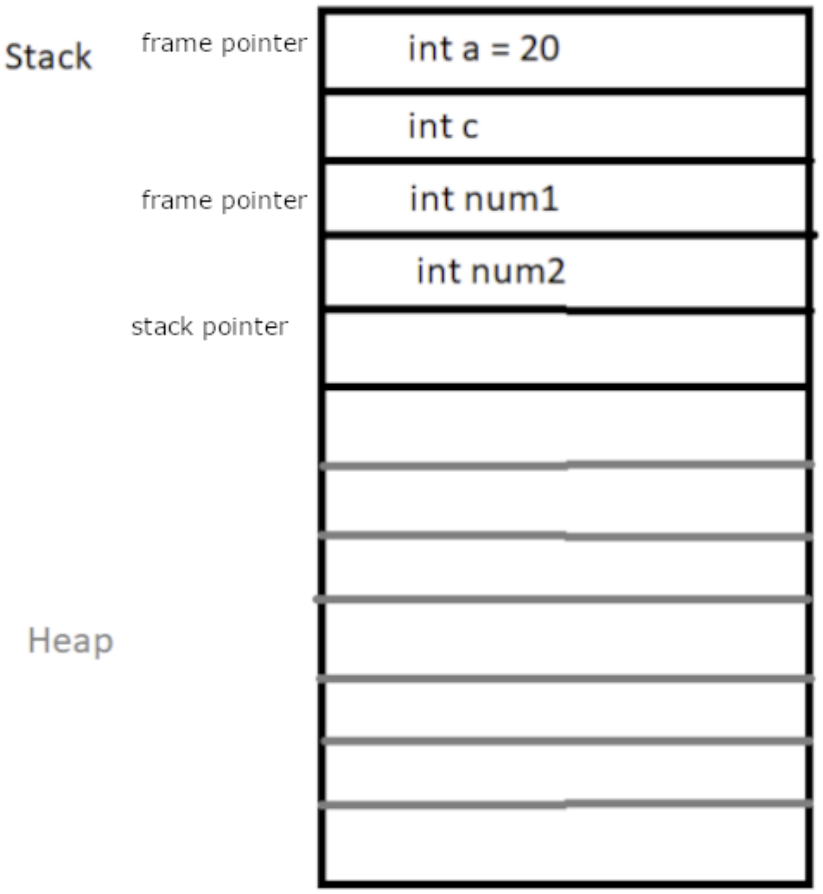


Cuando llamamos a una función, las variables utilizadas en ella, se almacenan en el stack.

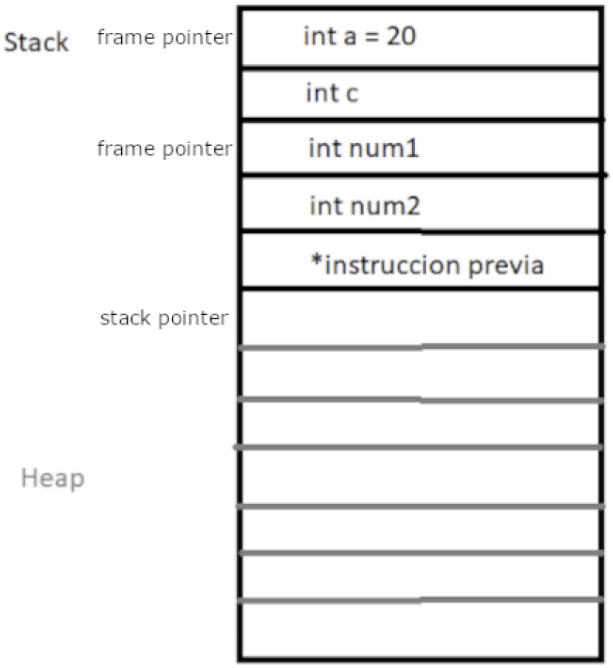
```
4   int Operation(int num1, int num2){
5
6     return num1+num2;
7   }
```

Estas variables se crean en el momento de la llamada

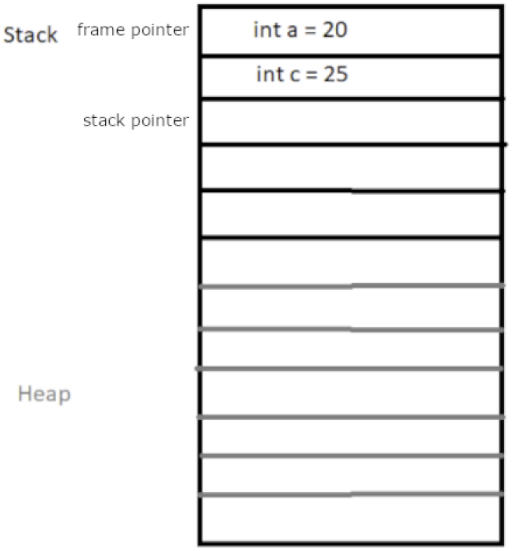
```
16    int c = Operation(a,5);
17
```



Además, se almacena un puntero a la siguiente instrucción que debe ejecutar, este puntero apunta al último bloque de memoria utilizado.

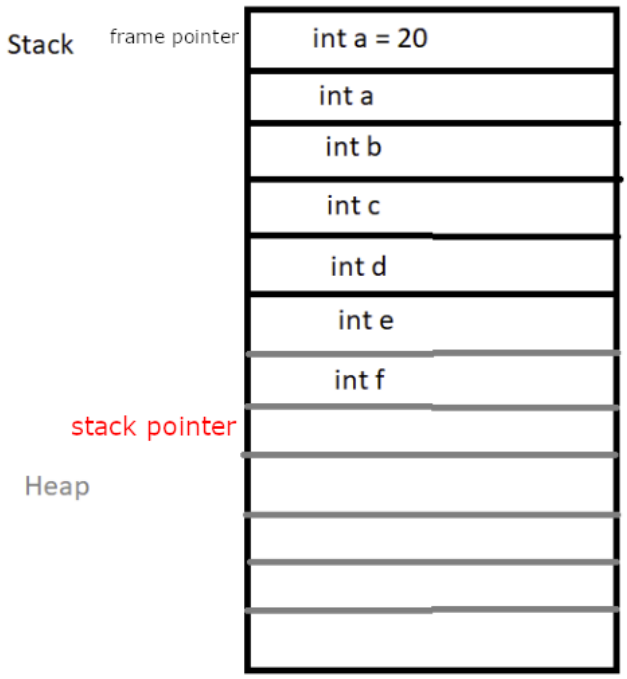


Cuando finaliza la función, todo lo que había en ella se destruye y llama a la instrucción previa



Si utilizamos excesiva memoria en el stack, sobrepasará al heap

```
4  int Operation(int num1, int num2){
5      int a;
6      int b;
7      int c;
8      int d;
9      int e;
10     int f;
11     return num1+num2;
12 }
13
```



Y obtendremos un error de overflow y se interrumpirá la ejecución del programa

2.- ADT Queue

El ADT Queue (cola) es una estructura de datos que sigue el principio FIFO (First In, First Out), lo que significa que el primer elemento en entrar es el primero en salir. Las funciones características propias del ADT Queue son:

- Enqueue (Encolar): Permite agregar un elemento al final de la cola. El elemento se inserta al final de la estructura, extendiendo así la cola.

- Dequeue (Desencolar): Elimina y devuelve el elemento que está en el frente de la cola. Es decir, el primer elemento que se insertó y que ha estado en la cola durante más tiempo.

- Front (Frente): Devuelve el elemento que se encuentra en el frente de la cola, sin eliminarlo. Permite observar cuál es el siguiente elemento que será desencolado.

- IsEmpty (Está vacía): Verifica si la cola está vacía, es decir, si no contiene elementos. Retorna un valor bool (verdadero o falso) que indica si la cola está vacía o no.

Estas funciones características son esenciales para trabajar con una cola y permiten realizar operaciones básicas de inserción, eliminación y acceso a los elementos de manera adecuada.

3.- ADT Vector Movable Head Double capacity

El ADT Vector Movable Head with double capacity, o vector con cabeza móvil y doble capacidad, es una variante del ADT Vector en el que se utiliza una cabeza móvil para realizar operaciones de inserción y eliminación eficientes. A continuación, se presentan las ventajas y desventajas de este ADT:

3.1.- Ventajas

- **Eficiencia en la inserción y eliminación**: La cabeza móvil permite insertar y eliminar elementos de manera eficiente, ya

que no es necesario desplazar todos los elementos restantes en el vector después de una operación.

- **Uso eficiente de la memoria**: Al utilizar una cabeza móvil y una capacidad doble, se puede optimizar el uso de la memoria. La capacidad se duplica cuando el vector está lleno, lo que reduce la frecuencia de redimensionamiento y minimiza la asignación y liberación de memoria.

- **Acceso directo a los elementos**: El acceso a los elementos en el vector se realiza mediante índices, lo que permite un acceso directo y rápido a cualquier posición del vector.

3.2.- Desventajas

- **Desperdicio de memoria**: Aunque el uso de una capacidad doble puede ser eficiente en términos de minimizar el número de redimensionamientos, también puede llevar al desperdicio de memoria. Si el vector no se llena, la capacidad adicional queda sin utilizar.

- **Complejidad en la implementación**: La gestión de la cabeza móvil y la redimensión del vector pueden agregar complejidad a la implementación del ADT. Esto puede dificultar el mantenimiento y la depuración del código.

- **Desempeño en operaciones de búsqueda**: Si bien el acceso a los elementos es eficiente en términos de tiempo constante (O(1)), las operaciones de búsqueda que requieren recorrer el vector pueden ser menos eficientes en comparación con otras estructuras de datos, como los árboles binarios de búsqueda.

Es importante tener en cuenta que las ventajas y desventajas pueden variar según el contexto y las necesidades específicas de la aplicación. Además, la eficiencia y el rendimiento pueden depender de la implementación concreta del ADT Vector Movable Head with double capacity.

4.- Sorting Algorithms

4.1.- QuickSort

El algoritmo de ordenación Quicksort es un algoritmo de ordenamiento basado en la técnica de divide y vencerás. Fue desarrollado por Tony Hoare en 1959 y es uno de los algoritmos de ordenamiento más eficientes en la práctica. Los pasos son los siguientes:

1. **Elección del pivote**: Selecciona un elemento del arreglo como pivote. El pivote puede elegirse de diferentes formas, siendo la elección del elemento medio del arreglo una opción común.

2. **Partición**: Reorganiza el arreglo de manera que todos los elementos menores que el pivote estén a su izquierda y los elementos mayores estén a su derecha. Al final de esta etapa, el pivote estará en su posición final en el arreglo ordenado.

3. **Recursión**: Aplica recursivamente el algoritmo Quicksort a las dos subsecciones generadas por la partición. Es decir, se aplica Quicksort a la subsección de elementos menores que el pivote y a la subsección de elementos mayores.

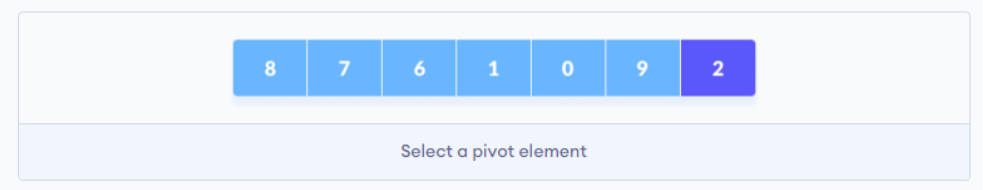
4. **Combinación**: Una vez que las subsecciones han sido ordenadas, se combinan para obtener el arreglo completamente ordenado.

El algoritmo se repite recursivamente hasta que el arreglo esté completamente ordenado, es decir, cuando no haya elementos que ordenar (cuando el tamaño de las subsecciones sea 0 o 1).

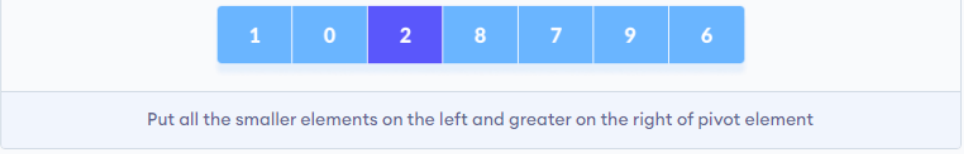
La elección del pivote es un aspecto crítico en el desempeño del algoritmo. Una elección subóptima del pivote puede llevar a un desempeño deficiente, especialmente en casos en los que los datos están preordenados o casi preordenados. Para mitigar esto, se pueden utilizar diferentes estrategias para seleccionar el pivote, como elegir el primer o último elemento del arreglo, o utilizar algoritmos más sofisticados como el método de la mediana de tres.

El tiempo de ejecución promedio del Quicksort es de O(n log n), donde n es el número de elementos a ordenar. Sin embargo, en el peor caso, cuando el pivote elegido siempre es el elemento más pequeño o más grande del arreglo, el tiempo de ejecución puede

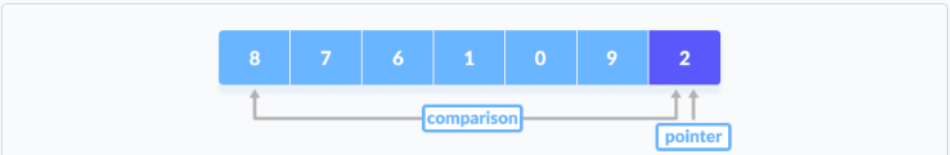
llegar a ser O(n^2). Para mitigar esto, se pueden utilizar variantes del algoritmo, como Quicksort con partición de tres vías o Quicksort aleatorizado.



Select a pivot element



Put all the smaller elements on the left and greater on the right of pivot element



Comparison of pivot element with element beginning from the first index

If the element is greater than the pivot element, a second pointer is set for that element.

Pivot is compared with other elements.

4.2.- Heap Sort

El algoritmo de ordenación HeapSort es un algoritmo eficiente que utiliza una estructura de datos llamada heap (montículo) para ordenar un arreglo. Los pasos son los siguientes:

1. **Creación del heap**: Comienza construyendo un heap a partir del arreglo de entrada. Un heap es una estructura de datos en forma de árbol binario completo en la que el valor de cada nodo es mayor o igual que los valores de sus hijos.

2. **Convertir a heap**: Para construir el heap, se recorre el arreglo de izquierda a derecha y se realiza un proceso llamado "flotación" (sift-up). En cada iteración, se compara el elemento actual con su padre y, si es mayor, se intercambian. Este proceso se repite hasta que el elemento llegue a la posición correcta en el heap.

3. **Ordenación**: Una vez que se ha creado el heap, el elemento máximo se encuentra en la raíz del árbol (posición 0 del arreglo). Se intercambia este elemento con el último elemento del arreglo y se reduce el tamaño del heap en 1. Luego, se realiza el proceso de "hundimiento" (sift-down) en la raíz para asegurar que el nuevo elemento en la raíz se coloque en la posición correcta dentro del heap.

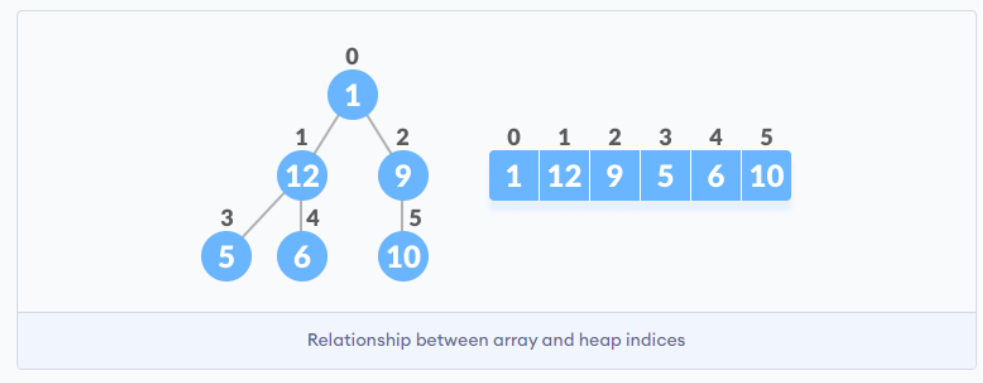
4. **Repetir**: Se repite el paso 3 hasta que el tamaño del heap sea 1. En cada iteración, se intercambia el elemento máximo con el último elemento no ordenado del arreglo y se coloca en su posición correcta. Esto se realiza hasta que todos los elementos estén en su posición final.

Al finalizar este proceso, el arreglo estará ordenado de menor a mayor (o de mayor a menor, dependiendo de cómo se haya construido el heap inicialmente).

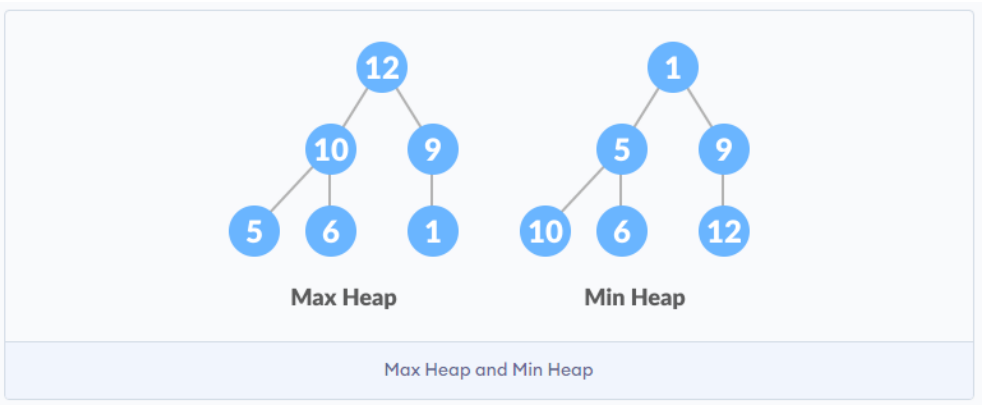
La principal ventaja del HeapSort es que garantiza un tiempo de ejecución de O(n log n) en todos los casos, tanto en el mejor como en el peor caso. Sin embargo, el HeapSort puede ser menos eficiente en comparación con otros algoritmos de ordenación en situaciones en las que la memoria caché es relevante, ya que su acceso a memoria no es tan contiguo como en otros algoritmos.

En resumen, HeapSort es un algoritmo de ordenación eficiente que utiliza una estructura de datos de heap para ordenar un arreglo.

Garantiza un tiempo de ejecución de O(n log n) y se utiliza en situaciones en las que se necesita una ordenación estable en todos los casos.



Relationship between array and heap indices



Max Heap and Min Heap

4.3.- Bubble Sort

El algoritmo Bubble Sort es un algoritmo simple de ordenación que compara e intercambia repetidamente pares de elementos adyacentes hasta que el arreglo esté completamente ordenado. Los pasos son los siguientes:

1. **Comparación de elementos adyacentes**: Comienza comparando el primer elemento con el segundo elemento, luego el segundo elemento con el tercero, y así sucesivamente, hasta llegar al penúltimo elemento del arreglo.

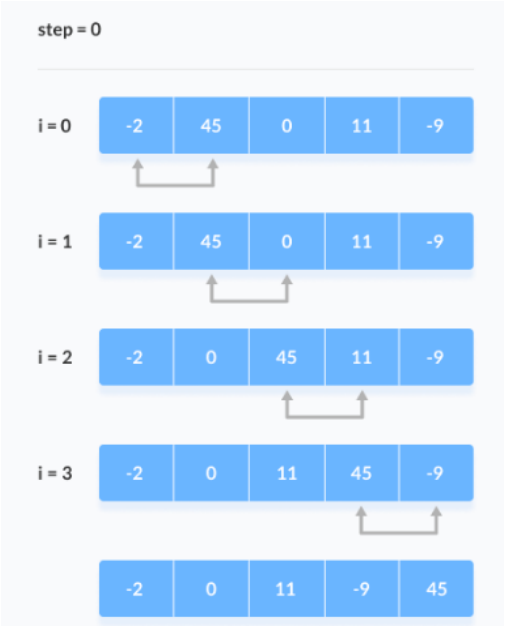
2. **Intercambio de elementos**: Si el par de elementos adyacentes está en el orden incorrecto (es decir, el elemento de la izquierda es mayor que el de la derecha en una ordenación ascendente), se intercambian. Esto significa que el elemento mayor "burbujea" hacia la derecha.

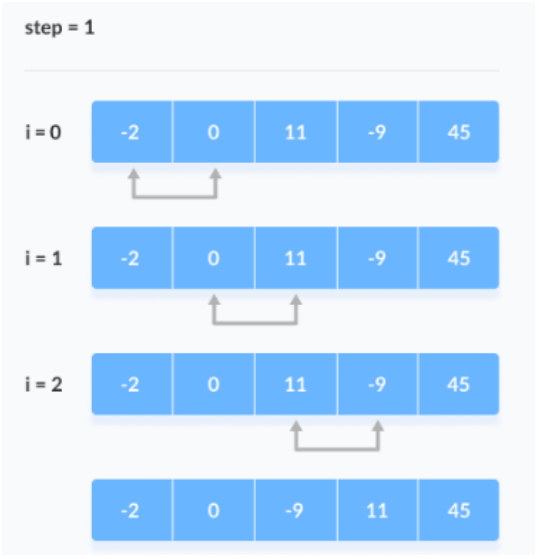
3. **Repetición**: Se repite el paso 1 y 2 para cada elemento restante en el arreglo, avanzando una posición hacia la derecha en cada iteración. En cada pasada, el elemento más grande de los elementos no ordenados "burbujea" hacia la derecha.

4. **Finalización**: Una vez que se completa una pasada completa del arreglo sin realizar intercambios, se considera que el arreglo está ordenado y el algoritmo termina.

El Bubble Sort es un algoritmo de ordenación sencillo de implementar, pero su rendimiento es menos eficiente en comparación con otros algoritmos más avanzados como Quicksort o mergesort. Su tiempo de ejecución promedio es de O(n^2), donde n es el número de elementos en el arreglo. En el peor caso, si el arreglo está en orden descendente, el tiempo de ejecución puede ser O(n^2). Sin embargo, en el mejor caso, cuando el arreglo ya está ordenado, el tiempo de ejecución puede ser O(n), ya que no se realizan intercambios.

El Bubble Sort puede ser útil en situaciones en las que el arreglo es pequeño o casi está ordenado, ya que su implementación es simple y no requiere estructuras de datos adicionales. Sin embargo, en general, se prefieren otros algoritmos de ordenación más eficientes para arreglos más grandes o en aplicaciones donde se requiere un rendimiento óptimo.


step = 0
i = 0
-2 45 0 11 -9
i = 1
-2 45 0 11 -9
i = 2
-2 0 45 11 -9
i = 3
-2 0 11 45 -9
-2 0 11 -9 45


step = 1
i = 0
-2 0 11 -9 45
i = 1
-2 0 11 -9 45
i = 2
-2 0 11 -9 45
-2 0 -9 11 45

4.4.- Cocktail Sort

El Cocktail Sort, también conocido como Bubble Sort bidireccional, es una variante del algoritmo Bubble Sort. Al igual que el Bubble Sort, el Cocktail Sort compara y intercambia repetidamente pares de elementos adyacentes para ordenar el arreglo. Sin embargo, a diferencia del Bubble Sort, el Cocktail Sort realiza pasadas en ambas direcciones, hacia adelante y hacia atrás, lo que le permite realizar intercambios tanto en sentido ascendente como descendente.

Los pasos son los siguientes:

1. **Inicio**: Comienza realizando una pasada hacia adelante a través del arreglo, comparando y intercambiando pares de elementos adyacentes, al igual que en el Bubble Sort estándar.

2. **Pasada hacia atrás**: Después de completar una pasada hacia adelante, realiza una pasada hacia atrás a través del arreglo, desde el último elemento hasta el segundo. Nuevamente, compara y intercambia pares de elementos adyacentes en sentido descendente.

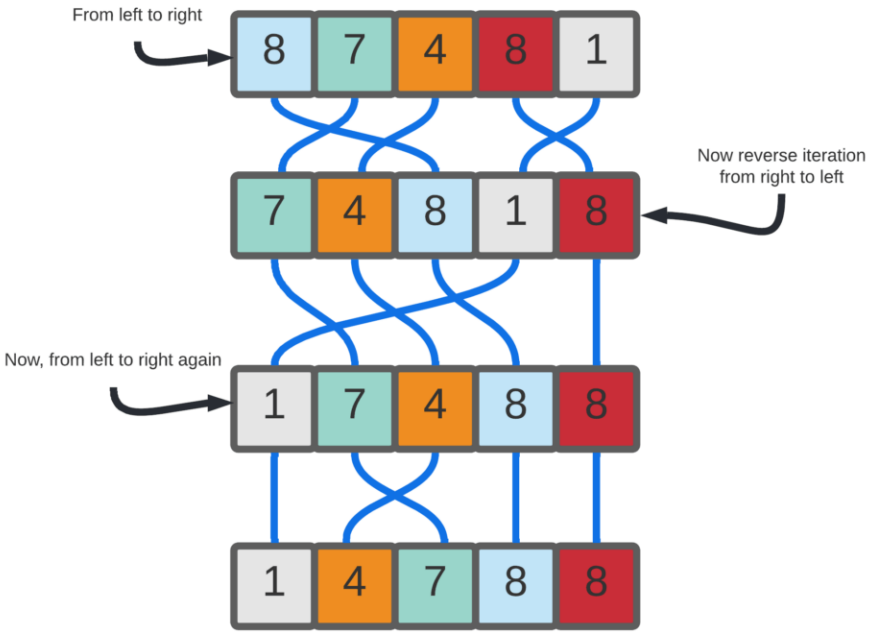
3. **Repetición**: Repite los pasos 1 y 2 hasta que no se realicen intercambios en ninguna de las pasadas, lo que indica que el arreglo está completamente ordenado.

Al realizar pasadas en ambas direcciones, el Cocktail Sort puede llevar elementos grandes hacia la derecha y elementos pequeños hacia la izquierda en cada iteración, acelerando el proceso de ordenación en comparación con el Bubble Sort estándar.

Al igual que el Bubble Sort, el Cocktail Sort tiene un tiempo de ejecución promedio de O(n^2), donde n es el número de elementos en el arreglo. En el mejor caso, cuando el arreglo ya está ordenado, el tiempo de ejecución puede ser O(n), mientras que en el peor caso, cuando el arreglo está en orden descendente, puede ser O(n^2).

El Cocktail Sort puede ser útil en situaciones en las que se tiene un arreglo parcialmente ordenado, ya que puede aprovechar las secciones ordenadas y reducir el número de comparaciones y movimientos necesarios. Sin embargo, en general, se prefieren

algoritmos de ordenación más eficientes, como Quicksort o mergesort, para arreglos más grandes o en aplicaciones donde se busca un rendimiento óptimo.



4.- Comparación Algoritmos

4.1 Comparación Bubble Sory y Cocktail Sort

El algoritmo de ordenación Bubble Sort y el algoritmo de ordenación Cocktail Sort son variaciones del mismo enfoque básico, conocido como ordenación por intercambio. Ambos algoritmos funcionan intercambiando pares adyacentes de elementos hasta que la lista esté ordenada. Sin embargo, hay una diferencia clave entre ellos: la dirección en la que se realiza el intercambio.

Bubble Sort: El algoritmo Bubble Sort recorre la lista de elementos de izquierda a derecha y compara pares adyacentes de elementos. Si un par está en el orden incorrecto, se intercambian. Este proceso se repite hasta que no se realicen más intercambios, lo que indica que la lista está ordenada. En cada iteración, el elemento más grande "burbujea" hacia la derecha.

Cocktail Sort: El algoritmo Cocktail Sort, también conocido como Bubble Sort bidireccional o Shaker Sort, es una variante del Bubble Sort. A diferencia del Bubble Sort, el Cocktail Sort recorre la lista en ambas direcciones, hacia adelante y hacia atrás. En cada pasada, se realizan dos fases: una hacia adelante y otra hacia atrás. Durante la fase hacia adelante, se realizan los intercambios necesarios para mover el elemento más grande hacia la posición correcta al final de la lista. Luego, durante la fase hacia atrás, se realiza lo mismo pero moviendo el elemento más pequeño hacia la posición correcta al inicio de la lista. Este proceso se repite hasta que la lista esté completamente ordenada.

La principal ventaja del Cocktail Sort sobre el Bubble Sort es que puede mejorar la eficiencia en ciertos casos. Al recorrer la lista en ambas direcciones, puede reducir el número de pasadas necesarias para ordenar la lista cuando hay elementos grandes desplazados hacia el final o elementos pequeños desplazados hacia el inicio. Sin embargo, en promedio, ambos algoritmos tienen una complejidad de tiempo similar de O(n^2), donde n es el número de elementos en la lista. Por lo tanto, para

conjuntos de datos grandes, se recomendarían algoritmos más eficientes, como Quick Sort o Merge Sort.

4.1 Comparación QuickSort y Heap Sort

Diferencias principales:

Enfoque: QuickSort utiliza la estrategia "divide y vencerás" y se basa en la partición de la lista, mientras que Heap Sort se basa en la estructura de datos heap para ordenar la lista.

Complejidad: En el caso promedio, QuickSort tiene una complejidad de tiempo de O(n log n), lo que lo hace eficiente para conjuntos de datos grandes. Por otro lado, Heap Sort tiene una complejidad de tiempo garantizada de O(n log n), lo que lo hace más consistente en términos de rendimiento, pero puede ser menos eficiente que QuickSort en el caso promedio.

Espacio adicional: QuickSort suele ser más eficiente en términos de uso de memoria, ya que puede ordenar la lista in situ, es decir, sin requerir memoria adicional significativa., por otro lado, generalmente requiere una estructura de datos heap adicional para ordenar la lista, lo que puede ocupar más espacio en memoria.

5.- ADT Como base de la POO

Los ADT (Abstract Data Types) son fundamentales en la programación basada en la orientación a objetos por varias razones. Algunas de estas son:

- Abstracción de datos: Los ADT permiten abstraer los detalles internos de la implementación de una estructura de datos y se centran en la especificación de las operaciones que se pueden realizar sobre ellos. Esto se alinea con uno de los principios clave de la OOP, que es la ocultación de la información o encapsulación. Los ADT proporcionan una interfaz clara y definida para interactuar con los datos, lo que promueve la encapsulación y el ocultamiento de detalles internos.

- Modularidad y reutilización de código: Los ADT fomentan la creación de módulos independientes y reutilizables que encapsulan la lógica y los datos relacionados. Estos módulos pueden ser tratados como objetos en la OOP, lo que facilita su reutilización en diferentes partes del código o en diferentes proyectos. Los ADT proporcionan una forma de modularizar y organizar el código, lo cual es uno de los pilares de la OOP.

- Polimorfismo: Los ADT pueden ser definidos de manera abstracta, lo que significa que la especificación de sus operaciones puede ser independiente de la implementación concreta. Esto permite el polimorfismo, que es uno de los conceptos clave de la OOP. El polimorfismo permite tratar diferentes implementaciones de un ADT de manera uniforme a través de una interfaz común, lo que facilita la flexibilidad y extensibilidad del código.

- Herencia: Los ADT pueden ser utilizados como base para la jerarquía de clases en la OOP. La herencia permite crear subtipos más específicos de un ADT base, que heredan las operaciones y características del ADT padre. Esto facilita la creación de relaciones de especialización y generalización entre los ADT y promueve la reutilización del código y la organización del diseño.

Estoy de acuerdo en que los ADT son la base de la programación basada en la orientación a objetos. Los ADT proporcionan una forma de abstraer y encapsular datos y operaciones, lo cual es fundamental en el paradigma de la OOP. La encapsulación promovida por los ADT permite ocultar los detalles internos de implementación y ofrece una interfaz clara y definida para interactuar con los datos.

Además, los ADT fomentan la modularidad y la reutilización de código, lo cual es un principio clave en la OOP. La capacidad de definir módulos independientes y reutilizables facilita la organización del código y su mantenimiento a largo plazo. También permite construir jerarquías de clases a través de la herencia, lo que promueve la extensibilidad y la creación de relaciones de especialización y generalización.

El polimorfismo, posible gracias a los ADT, es otro aspecto importante de la OOP. El polimorfismo permite tratar diferentes implementaciones de un ADT de manera uniforme a través de una interfaz común, lo que aumenta la flexibilidad y la capacidad de adaptación del código.

En resumen, los ADT son fundamentales en la programación basada en la OOP debido a su capacidad de abstracción, encapsulación, modularidad y reutilización de código, así como su apoyo al polimorfismo y la herencia. Estos conceptos son fundamentales en el diseño de software eficiente, mantenible y escalable.

6.- Ventajas ADT MemoryNode

El ADT Memory Node nos permite obtener independencia de los datos respecto a los algoritmos, lo que implica una separación clara entre los datos y los algoritmos utilizados para procesarlos. Algunas ventajas son:

- **Reutilización de algoritmos**: Al separar los datos de los algoritmos, los algoritmos se vuelven independientes de la estructura de datos subyacente. Esto permite que un mismo algoritmo pueda ser reutilizado con diferentes estructuras de datos que implementen el ADT Memory Node. Por ejemplo, si tenemos un algoritmo de búsqueda genérico que utiliza nodos de memoria, podemos utilizarlo con diferentes estructuras de datos que implementen ese ADT, como una lista enlazada, un árbol binario o una matriz. Esta separación promueve la reutilización de algoritmos y evita la necesidad de reescribir o adaptar los algoritmos cada vez que se cambia la estructura de datos.

- **Flexibilidad en el manejo de datos**: Al utilizar el ADT Memory Node, los algoritmos no necesitan conocer los detalles internos de la estructura de datos. Solo necesitan saber cómo interactuar con los nodos de memoria, es decir, cómo acceder, modificar o eliminar los datos contenidos en ellos. Esto brinda flexibilidad en el manejo de datos, ya que se pueden cambiar o actualizar las estructuras de datos subyacentes sin afectar la lógica de los algoritmos. Por ejemplo, si decidimos cambiar de una lista enlazada a una matriz, no es necesario modificar los algoritmos existentes, ya que seguirán interactuando con los nodos de memoria de la misma manera.

- **Mantenibilidad y legibilidad del código**: La separación entre datos y algoritmos proporcionada por el ADT Memory Node mejora la mantenibilidad y legibilidad del código. Al tener una capa de abstracción que encapsula los datos y su manipulación, el código se vuelve más modular y fácil de entender. Los algoritmos pueden enfocarse en la lógica de procesamiento de los datos, mientras que la estructura de datos se ocupa de almacenar y gestionar los datos en sí. Esto facilita la comprensión, el mantenimiento y la depuración del código, ya que cada componente tiene una responsabilidad claramente definida.

El uso del ADT Memory Node proporciona independencia de los datos respecto a los algoritmos, lo que conlleva una serie de ventajas, como la reutilización de algoritmos, la flexibilidad en el manejo de datos y la mejora en la mantenibilidad y legibilidad del código. Esta separación fomenta un diseño modular y flexible, permitiendo cambios en la estructura de datos sin afectar la lógica de los algoritmos y facilitando la colaboración y el mantenimiento a largo plazo del código.

2.- Bibliografía

Stack based memory [digital information] wikipedia.org [consulted 14/12/2022]. Available on: https://en.wikipedia.org/wiki/Stack-based_memory_allocation

Memory stack [digital information] tutorialspoint.com [consulted 14/12/2022]. Available on: https://www.tutorialspoint.com/what-is-memory-stack-in-computer-architecture

Memory Stack [digital information] sciencedirect.com [consulted 14/12/2022]. Available on: https://www.sciencedirect.com/topics/engineering/stack-memory

Memory Stack [digital information] sciencedirect.com [consulted 14/12/2022]. Available on: https://www.philadelphia.edu.jo/academics/qhamarsheh/uploads/Lecture%2015%20Stack%20Operations.pdf

Bubble sort [digital information] programiz.com [consulted 29/05/2023]. Available on: https://www.programiz.com/dsa/bubble-sort

Quick sort [digital information] programiz.com [consulted 29/05/2023]. Available on: https://www.programiz.com/dsa/quick-sort

Heap sort [digital information] programiz.com [consulted 29/05/2023]. Available on: https://www.programiz.com/dsa/heap-sort

Cocktail sort [digital information] geeksforgeeks.org [consulted 29/05/2023]. Available on: https://www.geeksforgeeks.org/cocktail-sort/

Chat gpt [digital information] openai.com/blog/chatgpt [consulted 29/05/2023]. Available on: https://chat.openai.com/