ESAT
INNOVATION SCHOOL
ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

Algoritmos y estructuras de datos

ESAT INNOVATION SCHOOL | ESCUELA SUPERIOR DE ARTE Y TECNOLOGÍA

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

Índice/Index

1.- Stack

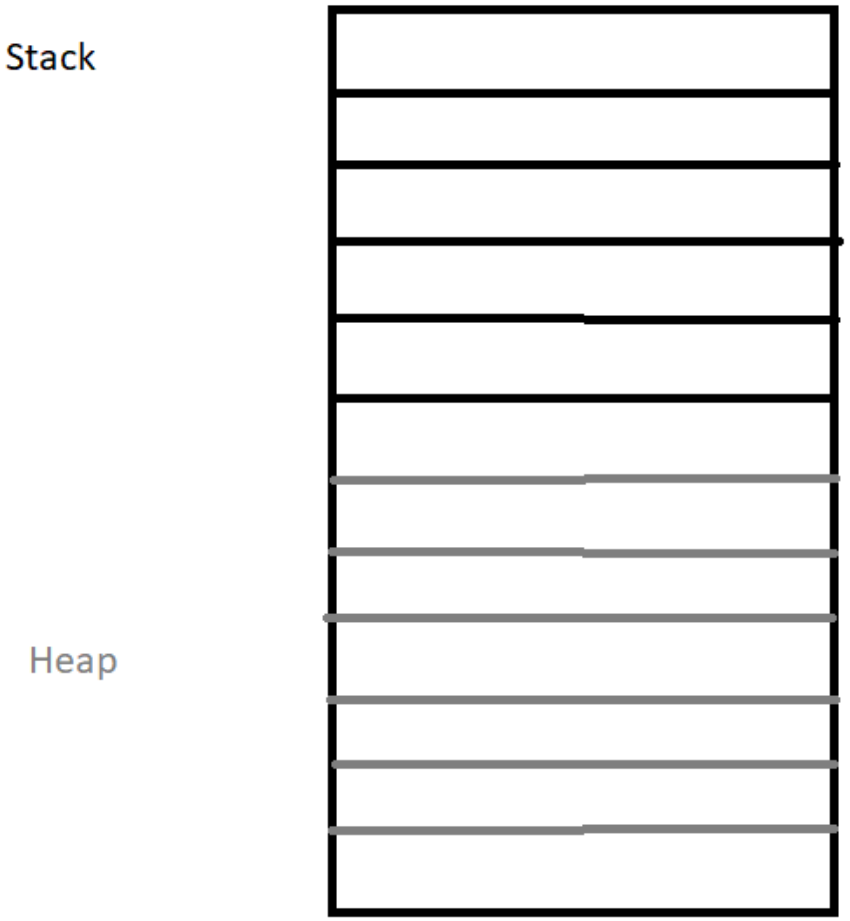
1.1.-What is it?

The stack is a memory area where automatic local variables are stored. This memory area is located at the highest addresses. It is a temporary memory, where the local variables of the called functions are stored. When you call a function, a new frame is created in the stack and the local variables of the function are pushed. After each function is called, a new frame pointer is created. When this function finishes, the variables are popped to free the memory. When this happens the memory of the variables is freed and the frame pointer of that function is removed. The stack also has a special pointer which is the stack pointer, this pointer is in charge of delimiting the end of the memory stack.

The memory stack acts as a lifo, that is to say that when a function is called, a push of its variables is made and it is executed, when it finishes executing, the function is unstacked by popping its variables and it returns to the previous jump. This means that when a function is called, the execution of the previous one cannot continue until the first one is finished. In the case that many functions are called one inside the other, the stack will grow a lot.

1.2.- Ejemplo gráfico

We have the Stack memory and then we have the heap.



When we create a variable within a function, it is stored in the stack.

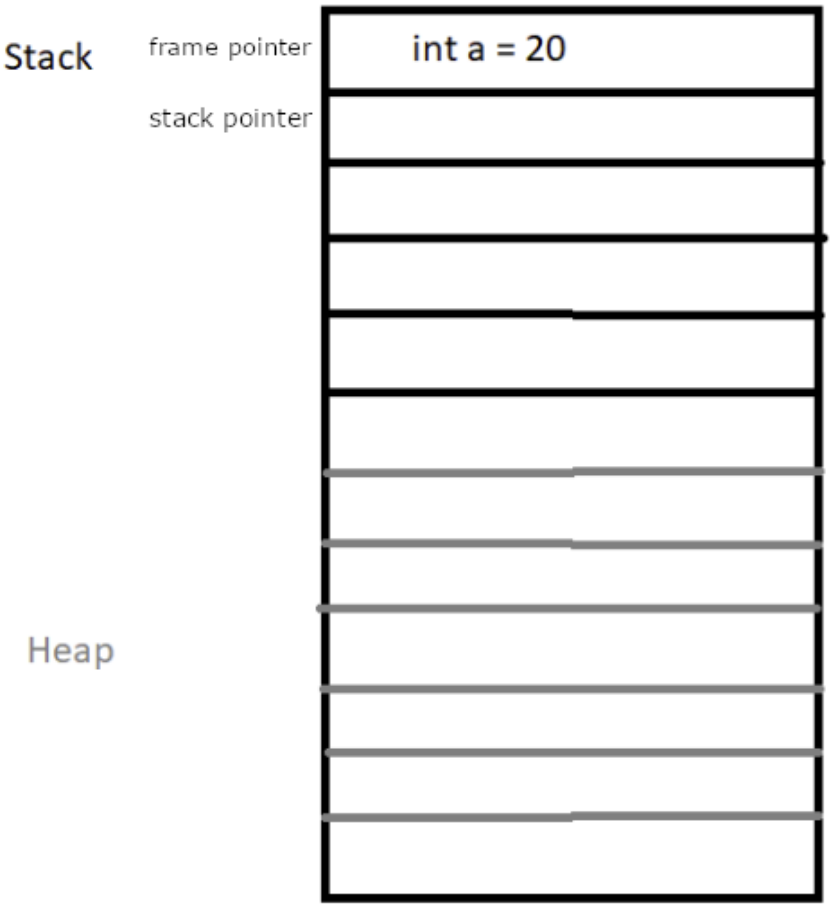
main.cc

```cpp
#include <stdio.h>


int main(){

  int a;
  a = 20;


  return 0;
}
```

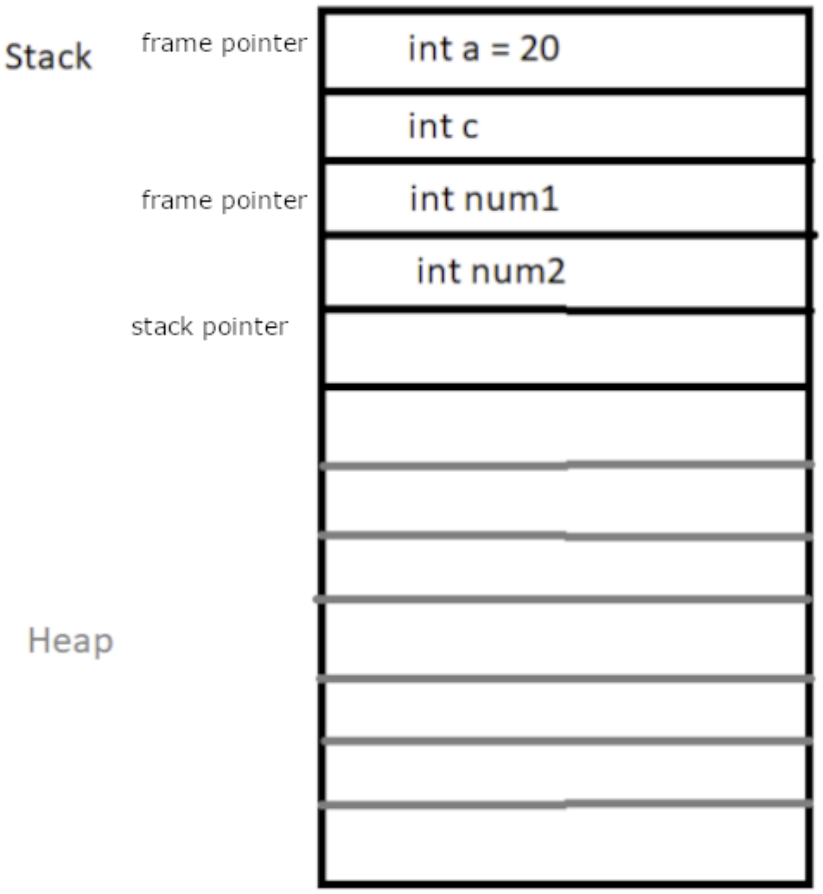


When we call a function, the variables used in it are stored in the stack.

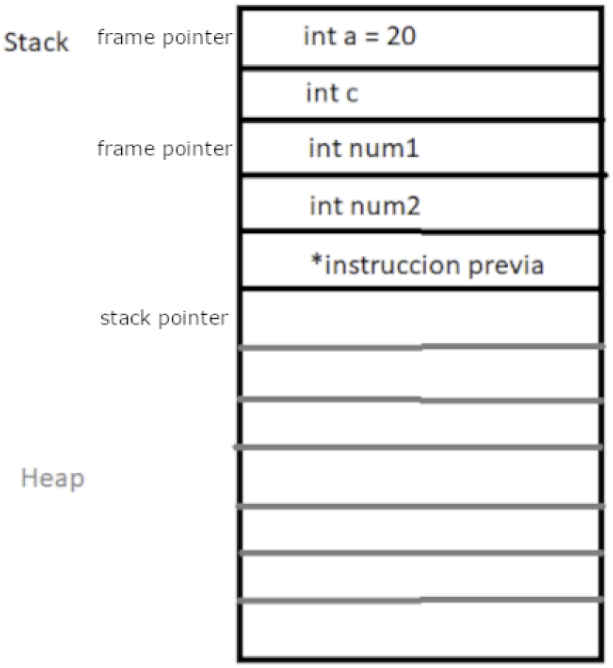
```
4   int Operation(int num1, int num2){
5
6     return num1+num2;
7   }
```

These variables are created at the time of the call.

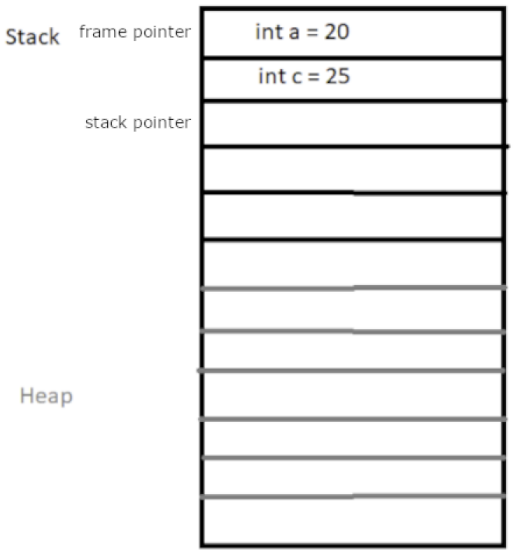
```
16    int c = Operation(a,5);
17
```



In addition, a pointer to the next instruction to be executed is stored, this pointer points to the last memory block used.

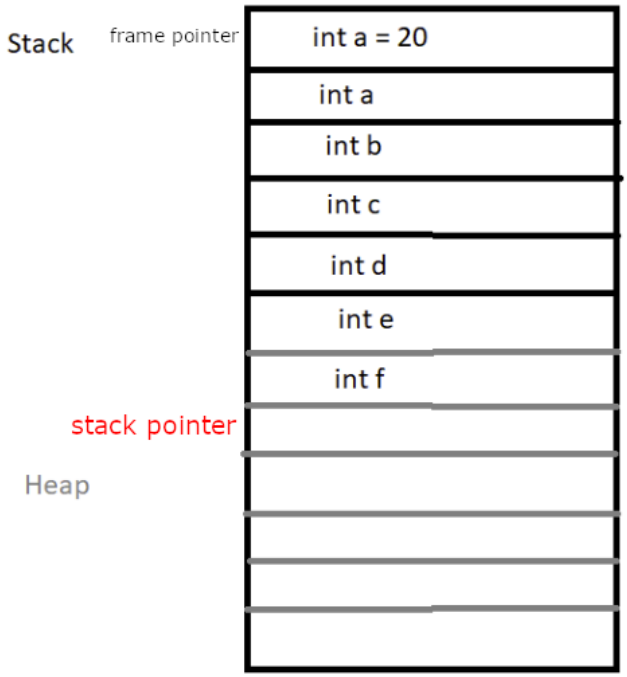


At the end of the function, everything in the function is destroyed and the previous instruction is called.



If we use too much memory in the stack, it will overrun the 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
```



And we will get an overflow error and the execution of the program will be interrupted.

2.- ADT Queue

The ADT Queue is a data structure that follows the FIFO (First In, First Out) principle, which means that the first item in is the first item out. The characteristic functions of the ADT Queue are:

- Enqueue: Allows an element to be added to the end of the queue. The element is inserted at the end of the structure, thus extending the queue.

- Dequeue: Removes and returns the element that is at the front of the queue. That is, the first item that was inserted and has been in the queue the longest.

- Front: Returns the element at the front of the queue, without removing it. It allows to observe which is the next element to be unqueued.

- IsEmpty: Checks if the queue is empty, i.e. if it contains no elements. Returns a bool value (true or false) indicating whether the queue is empty or not.

These characteristic functions are essential for working with a queue and allow basic insert, delete and access operations to be performed properly.

3.- ADT Stack

The ADT Stack is a data structure that follows the LIFO (Last In, First Out) principle, which means that the last element in is the first out. The characteristic features of the ADT Stack are:

- Push: When you push, the new item is placed on top of the stack. The most recently added item becomes the top item in the stack.
- Pop: Pop is used to describe the operation of removing the top element from the stack. When popping, the element on top of the stack is removed and returned. After popping, the next item on the stack becomes the new top item.
- Top: Used to view the top element without being removed.

4.- ADT Vector Movable Head Double capacity

The ADT Vector Movable Head with double capacity is a variant of the ADT Vector in which a movable head is used to perform efficient insertion and removal operations. The advantages and disadvantages of this ADT are presented below:

4.1.- Advantages

- **Efficient insertion and removal:** The moving head allows for efficient insertion and removal of elements, as it is not necessary to move all remaining elements in the vector after an operation.

- **Efficient memory usage:** By using a moving head and double capacity, memory usage can be optimized. Capacity is doubled when the vector is full, which reduces the frequency of resizing and minimizes memory allocation and release.

- **Direct access to elements:** Access to the elements in the vector is via indexes, allowing direct and quick access to any position in the vector.

4.2.- Disadvantages

- **Memory wastage:** Although the use of a double capacity can be efficient in terms of minimizing the number of resizes, it can also lead to memory wastage. If the vector is not filled, the additional capacity remains unused.

- **Implementation complexity:** Moving head management and vector resizing can add complexity to the ADT implementation. This can make code maintenance and debugging difficult.

- **Performance in search operations:** While accessing elements is efficient in terms of constant time (O(1)), search operations that require traversing the vector may be less efficient compared to other data structures, such as binary search trees.

It is important to note that the advantages and disadvantages may vary depending on the context and specific needs of the application. In addition, efficiency and performance may depend on the particular implementation of the ADT Vector Movable Head with double capacity.

5.- Sorting Algorithms

5.1.- QuickSort

The Quicksort sorting algorithm is a sorting algorithm based on the divide-and-conquer technique. It was developed by Tony Hoare in 1959 and is one of the most efficient sorting algorithms in practice. The steps are as follows:

1. **Choice of pivot:** Select one element of the array as the pivot. The pivot can be chosen in different ways, with the choice of the middle element of the array being a common option.

2. **Partitioning:** Rearrange the array so that all elements smaller than the pivot are to its left and all elements larger than the pivot are to its right. At the end of this stage, the pivot will be in its final position in the ordered array.

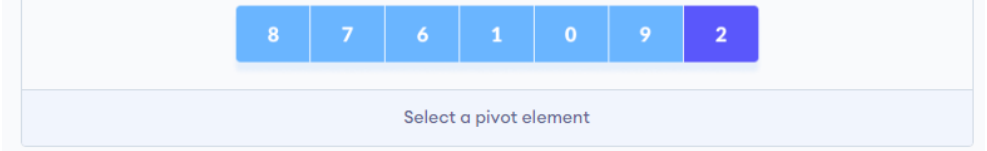
3. **Recursion:** Recursively applies the Quicksort algorithm to the two subsections generated by the partition. That is, Quicksort is applied to the subsection of elements smaller than the pivot and to the subsection of elements larger than the pivot.

4. **Combining:** Once the subsections have been sorted, they are combined to obtain the fully sorted array.

The algorithm is repeated recursively until the array is completely sorted, i.e. when there are no elements to sort (when the size of the subsections is 0 or 1).

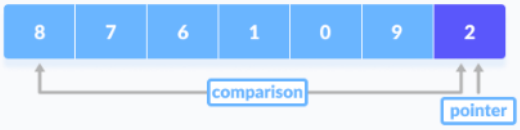
The choice of the pivot is a critical aspect in the performance of the algorithm. A suboptimal choice of pivot can lead to poor performance, especially in cases where the data is pre-sorted or nearly pre-sorted. To mitigate this, different strategies can be used to select the pivot, such as choosing the first or last element of the array, or using more sophisticated algorithms such as the median-of-three method.

The average execution time of Quicksort is O(n log n), where n is the number of elements to sort. However, in the worst case, when the chosen pivot is always the smallest or largest element in the array, the runtime can become O(n^2). To mitigate this, variants of the algorithm can be used, such as Quicksort with three-way partitioning or randomized Quicksort.

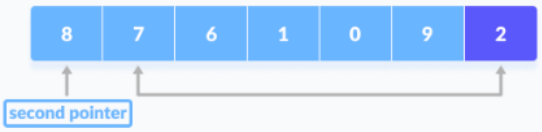


Select a pivot element

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.

5.2.- Heap Sort

The HeapSort algorithm is an efficient sorting algorithm that uses a data structure called a heap to sort an array. The steps are as follows:

1. **Creating the heap**: Start by building a heap from the input array. A heap is a data structure in the form of a complete binary tree in which the value of each node is greater than or equal to the values of its children.

2. **Convert to heap**: To build the heap, the array is traversed from left to right and a process called "sift-up" is performed. At each iteration, the current element is compared to its parent and, if it is larger, they are swapped. This process is repeated until the element reaches the correct position in the heap.

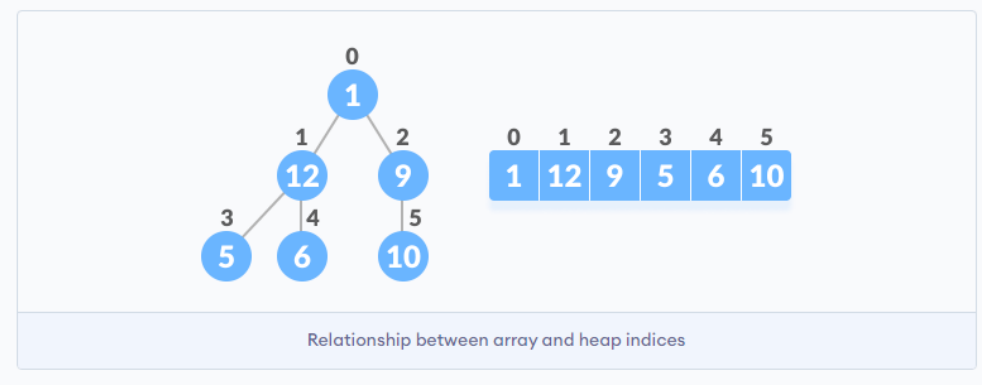
3. **Sorting**: Once the heap has been created, the maximum element is at the root of the tree (position 0 of the array). This element is exchanged with the last element in the array and the size of the heap is reduced by 1. Then, the sift-down process is performed at the root to ensure that the new element at the root is placed in the correct position within the heap.

4. **Repeat**: Step 3 is repeated until the heap size is 1. At each iteration, the maximum element is swapped with the last unordered element in the array and placed in its correct position. This is done until all elements are in their final position.

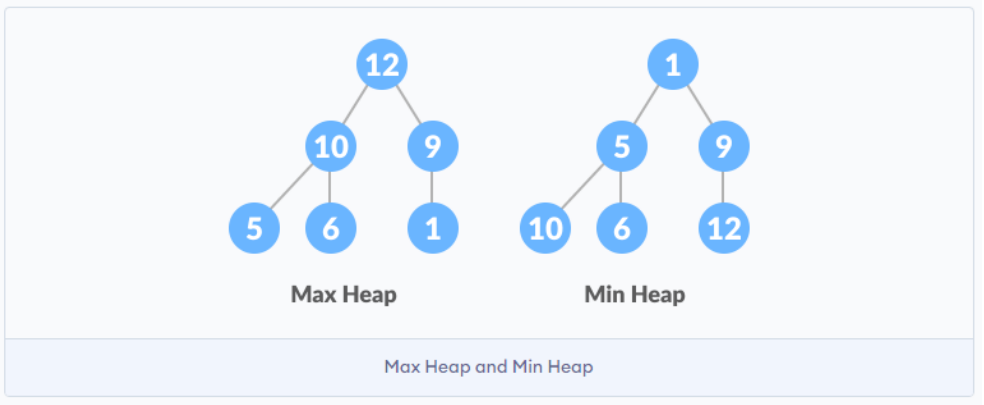
At the end of this process, the array will be sorted from smallest to largest (or from largest to smallest, depending on how the heap was initially constructed).

The main advantage of HeapSort is that it guarantees a runtime of O(n log n) in all cases, both in the best and worst case. However, HeapSort may be less efficient compared to other sorting algorithms in situations where cache memory is relevant, as its memory access is not as contiguous as in other algorithms.

In summary, HeapSort is an efficient sorting algorithm that uses a heap data structure to sort an array. It guarantees a runtime of O(n log n) and is used in situations where stable sorting is needed in all cases.



Relationship between array and heap indices



Max Heap and Min Heap

5.3.- Bubble Sort

The Bubble Sort algorithm is a simple sorting algorithm that repeatedly compares and swaps pairs of adjacent elements until the array is completely sorted. The steps are as follows:

1. **Comparing adjacent elements**: It starts by comparing the first element with the second element, then the second element with the third element, and so on, until the penultimate element of the array is reached.

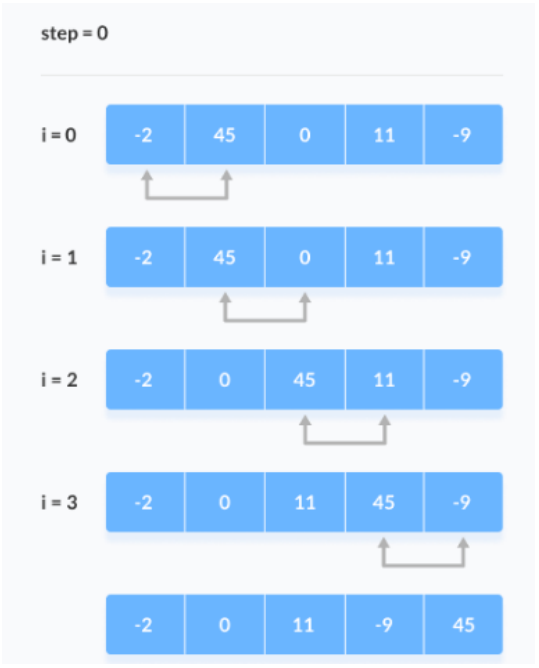
2. **Element swapping**: If the pair of adjacent elements is in the wrong order (i.e. the element on the left is greater than the element on the right in an ascending order), they are swapped. This means that the larger element "bubbles" to the right.

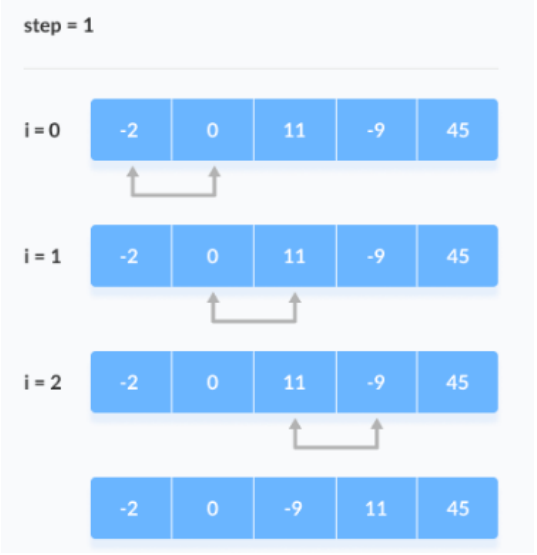
3. **Repetition**: Step 1 and 2 are repeated for each remaining element in the array, moving one position to the right in each iteration. On each pass, the largest of the unordered elements "bubbles" to the right.

4. **Completion**: Once a full pass of the array is completed without any swaps, the array is considered to be sorted and the algorithm terminates.

The Bubble Sort is a simple to implement sorting algorithm, but its performance is less efficient compared to more advanced algorithms such as Quicksort or mergesort. Its average running time is O(n^2), where n is the number of elements in the array. In the worst case, if the array is in descending order, the running time can be O(n^2). However, in the best case, when the array is already sorted, the runtime can be O(n), since no swaps are performed.

The Bubble Sort can be useful in situations where the array is small or nearly sorted, as its implementation is simple and does not require additional data structures. However, in general, other more efficient sorting algorithms are preferred for larger arrays or in applications where optimal performance is required.





5.4.- Cocktail Sort

Cocktail Sort, also known as bidirectional Bubble Sort, is a variant of the Bubble Sort algorithm. Like the Bubble Sort, the Cocktail Sort repeatedly compares and swaps pairs of adjacent elements to sort the array. However, unlike the Bubble Sort, the Cocktail Sort makes passes in both forward and backward directions, allowing it to perform both forward and backward swaps.

The steps are as follows:

1. **Start**: It begins by making a forward pass through the array, comparing and swapping pairs of adjacent elements, just as in the standard Bubble Sort.

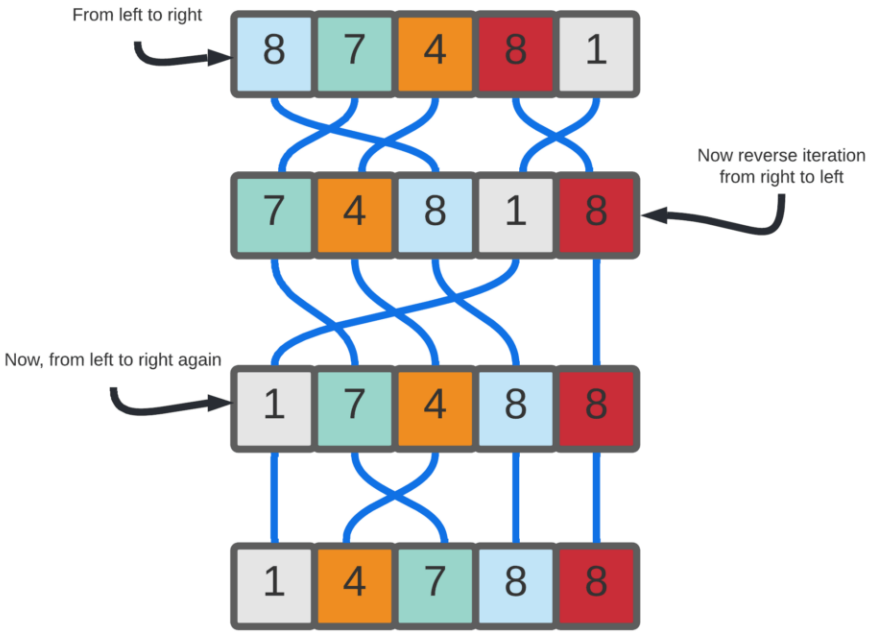
2. **Backward pass**: After completing a forward pass, it performs a backward pass through the array, from the last element to the second. Again, compare and swap pairs of adjacent elements in a downward direction.

3. **Repeat**: Repeat steps 1 and 2 until no swaps are made in any of the passes, indicating that the array is completely sorted.

By making passes in both directions, the Cocktail Sort can carry large elements to the right and small elements to the left on each iteration, speeding up the sorting process compared to the standard Bubble Sort.

Like the Bubble Sort, the Cocktail Sort has an average runtime of O(n^2), where n is the number of elements in the array. In the best case, when the array is already sorted, the runtime can be O(n), while in the worst case, when the array is in descending order, it can be O(n^2).

The Cocktail Sort can be useful in situations where you have a partially sorted array, as it can take advantage of the sorted sections and reduce the number of comparisons and moves required. However, in general, more efficient sorting algorithms, such as Quicksort or mergesort, are preferred for larger arrays or in applications where optimal performance is sought.



6.- Search Algorithms Comparison

- **Breadth-first search (BFS):**

  The BFS explores all neighboring nodes of a node before moving on to the neighboring nodes of neighboring nodes. It starts at the initial node and moves level by level in the network.
  It uses a data structure called a queue (FIFO - First In, First Out) to store the nodes to be explored.
  It finds the optimal solution closest to the initial node in terms of number of edges or levels.
  It guarantees to find the shortest solution in terms of number of edges or levels if the network has edges with equal weight.

- **Depth Search (DFS):**

  DFS explores a path until it reaches a node with no unexplored neighbors, and then backtracks and explores other paths. It starts at the initial node and moves as deep as possible in the network before backtracking.
  It uses a data structure called a stack (LIFO - Last In, First Out) to store the nodes to be explored.
  It does not guarantee to find the optimal solution in terms of number of edges or levels, as it may continue to explore deeper before backtracking.
  It can be more efficient in terms of memory space, as it does not need to store all the nodes of a complete level before moving on to the next one.

- **Comparison:**

Time complexity: In general, both algorithms have similar time complexity, as they visit each node and edge exactly once. However, in the worst case, DFS may explore the entire network in depth before finding a solution, which could take longer than BFS if the solution is at a level close to the initial node.

Memory space: BFS tends to use more memory space than DFS, as it stores all nodes of a complete level before

moving on to the next level. In contrast, DFS only needs to store the nodes of the current path on the stack.

Optimal solutions: If all edges have equal weight, the BFS guarantees to find the shortest solution in terms of number of edges or levels. DFS does not guarantee an optimal solution, as it may continue to explore in depth before backtracking.

Data structures used: BFS uses a queue to store the nodes to be scanned, while DFS uses a stack. This has implications for the order of scanning and the behaviour of the algorithm.

In summary, BFS is suitable for finding optimal solutions closer to the initial node and for graphs with edges of equal weight, while DFS is useful when you want to explore in depth and do not need to find the optimal solution in terms of number of edges or levels.

7.- ADT as base of OOP

Abstract Data Types (ADTs) are fundamental in object-oriented programming for several reasons. Some of these are:

- **Data abstraction**: ADTs allow the abstraction of the internal implementation details of a data structure and focus on specifying the operations that can be performed on it. This aligns with one of the key principles of OOP, which is information hiding or encapsulation. ADTs provide a clear and defined interface for interacting with data, which promotes encapsulation and the hiding of internal details.

- **Modularity and code reuse**: TDAs encourage the creation of independent, reusable modules that encapsulate logic and related data. These modules can be treated as objects in the OOP, which facilitates their reuse in different parts of the code or in different projects. ADTs provide a way to modularise and organise code, which is one of the pillars of OOP.

- **Polymorphism**: ADTs can be defined abstractly, which means that the specification of their operations can be independent of the concrete implementation. This allows for polymorphism, which is one of the key concepts of OOP. Polymorphism allows different implementations of an ADT to be treated uniformly through a common interface, which facilitates code flexibility and extensibility.

- **Inheritance**: ADTs can be used as the basis for the class hierarchy in OOP. Inheritance allows the creation of more specific subtypes of a base ADT, which inherit the operations and features of the parent ADT. This facilitates the creation of specialisation and generalisation relationships between ADTs and promotes code reuse and design organisation.

I agree that ADTs are the foundation of object-oriented programming. ADTs provide a way to abstract and encapsulate data and operations, which is fundamental to the OOP paradigm. The encapsulation promoted by ADTs allows internal implementation details to be hidden and provides a clear and defined interface for interacting with data.

In addition, TDAs encourage modularity and code reuse, which is a key principle in OOP. The ability to define independent and reusable modules facilitates code organisation and long-term maintenance. It also allows building class hierarchies through inheritance, which promotes extensibility and the creation of specialisation and generalisation relationships.

Polymorphism, made possible by ADTs, is another important aspect of OOP. Polymorphism allows different implementations of an ADT to be treated uniformly through a common interface, which increases the flexibility and adaptability of the code.

In summary, ADTs are fundamental to OOP-based programming because of their abstraction, encapsulation, modularity and code reuse capabilities, as well as their support for polymorphism and inheritance. These concepts are fundamental in designing efficient, maintainable and scalable software.

8.- Advantages ADT MemoryNode

The ADT Memory Node allows us to obtain data independence from the algorithms, which implies a clear separation between the data and the algorithms used to process it. Some advantages are:

- **Reuse of algorithms**: By separating the data from the algorithms, the algorithms become independent of the underlying data structure. This allows the same algorithm to be reused with different data structures implementing the ADT Memory Node. For example, if we have a generic search algorithm that uses memory nodes, we can use it with different data structures that implement that ADT, such as a linked list, a binary tree or a matrix. This separation promotes algorithm reuse and avoids the need to rewrite or adapt algorithms every time the data structure is changed.

- **Flexibility in data handling**: By using the ADT Memory Node, algorithms do not need to know the internal details of the data structure. They only need to know how to interact with the memory nodes, i.e. how to access, modify or delete the data contained in them. This provides flexibility in data handling, as the underlying data structures can be changed or updated without affecting the logic of the algorithms. For example, if we decide to change from a linked list to an array, there is no need to modify the existing algorithms, as they will continue to interact with the memory nodes in the same way.

- **Code maintainability and readability**: The separation between data and algorithms provided by the ADT Memory Node improves code maintainability and readability. By having an abstraction layer that encapsulates the data and its manipulation, the code becomes more modular and easier to understand. Algorithms can focus on the logic of processing the data, while the data structure deals with storing and managing the data itself. This makes the code easier to understand, maintain and debug, as each component has a clearly defined responsibility.

The use of the ADT Memory Node provides independence of the data from the algorithms, which brings a number of advantages, such as reusability of algorithms, flexibility in data handling and improved maintainability and readability of the code. This separation encourages a modular and flexible design, allowing changes to the data structure without affecting the logic of the algorithms and facilitating collaboration and long-term maintenance of the code.

9.- Advantages of C++ port

Encapsulation and data privatisation in C++ offers several advantages compared to other programming approaches. The following are some of the most relevant advantages:

- **Information hiding**: Encapsulation allows the internal details of a class or data structure to be hidden by limiting direct access to its private members. This prevents users of the class from manipulating the internal data in undesirable ways and helps maintain data integrity and consistency.

- **Protection and security**: By making data private, access to it can be controlled through public methods of the class. This allows you to apply validations and restrictions on data manipulation, which helps ensure data consistency and security. In addition, encapsulation helps prevent unauthorised access to the data from other parts of the program.

- **Modularity and maintainability**: Encapsulation allows the definition of clear and well-defined interfaces to interact with a class or data structure. This encourages modularity and the development of independent components, which makes the code easier to understand and maintain. In addition, if the internal details of a class change in the future, only the public methods of the class need to be updated, while the rest of the code used by the class remains unchanged.

- **Abstraction and reusability**: Encapsulation allows the internal complexity of a class to be hidden and a simplified interface to be presented for use. This facilitates abstraction and the development of more readable and understandable code. In addition, encapsulating data and related functionality in a class promotes code reuse by allowing other components to use the class in a simple and consistent way.

Other relevant advantages of the port to C++ include:

- **Support for object-oriented programming**: C++ is an object-oriented programming language, which means that it offers features such as inheritance, polymorphism and encapsulation, which make it easier to structure and organise code in larger projects. This allows for modular and extensible design, resulting in more maintainable and scalable code.

- **Access to C++ libraries and ecosystem**: By porting to C++, a large set of libraries and tools developed for the language can be accessed. This provides advantages in terms of code reuse, efficiency and compatibility with specific systems and platforms.

In summary, encapsulation and data privatisation in C++ provide advantages such as information hiding, data protection, modularity, abstraction and code reuse. Furthermore, the port to C++ offers additional benefits such as efficiency, support for object-oriented programming and access to a broad ecosystem of libraries and tools. These advantages make encapsulation and porting to C++ best practices in software development.

10.- Logger

A logger is a tool for recording events or occurrences that happen during the execution of a program or system. Its main purpose is to record relevant information about the operation of the program, such as error messages, warnings, important events and any other type of log that may be useful for monitoring, analysis and diagnosis of problems.

The logger is a fundamental part of log management (logging) in software development. It provides a structured way to capture and store information about the behaviour and state of an application at runtime. Some common characteristics of loggers include:

- **Logging levels**: Loggers allow different severity levels to be defined for the messages being logged, such as DEBUG, INFO, WARNING, ERROR and CRITICAL. This allows you to filter the logs according to their importance and to adjust the desired level of detail in the log.

- **Log destinations**: Loggers can have different destinations for storing logs, such as text files, databases, remote servers, consoles or any other desired media. This provides flexibility to tailor log output to the specific needs of the execution environment.

- **Log format**: Loggers typically provide the ability to customise the format of logged messages, including information such as the date and time of the log, the severity level, the name of the component logging the message, and the specific content of the message.

- **Flexible configuration**: Loggers typically allow for flexible configuration, either through configuration files, environment variables or programmatically. This makes it easy to adjust the configuration of the logger without having to modify the source code of the application.

The use of a logger in an application has several advantages:

- It facilitates the tracking and debugging of problems by providing a detailed record of what has happened to the application during execution.
- It helps in monitoring and analysing performance, allowing the identification of bottlenecks, response times and unexpected behaviour.
- Enables logging of events relevant to the business or to meet audit and compliance requirements.
- It facilitates collaboration and teamwork, as multiple developers can share and analyse logs to understand application behaviour.

In summary, a logger is an essential tool in software development that records events and occurrences during the execution of an application. It provides valuable information for tracking, analysing and diagnosing problems, and facilitates monitoring, debugging and collaboration in software development.

11.- 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/