DATA STRUCTURES AND ALGORITHMS

Hồ Sĩ Quốc-BH00938

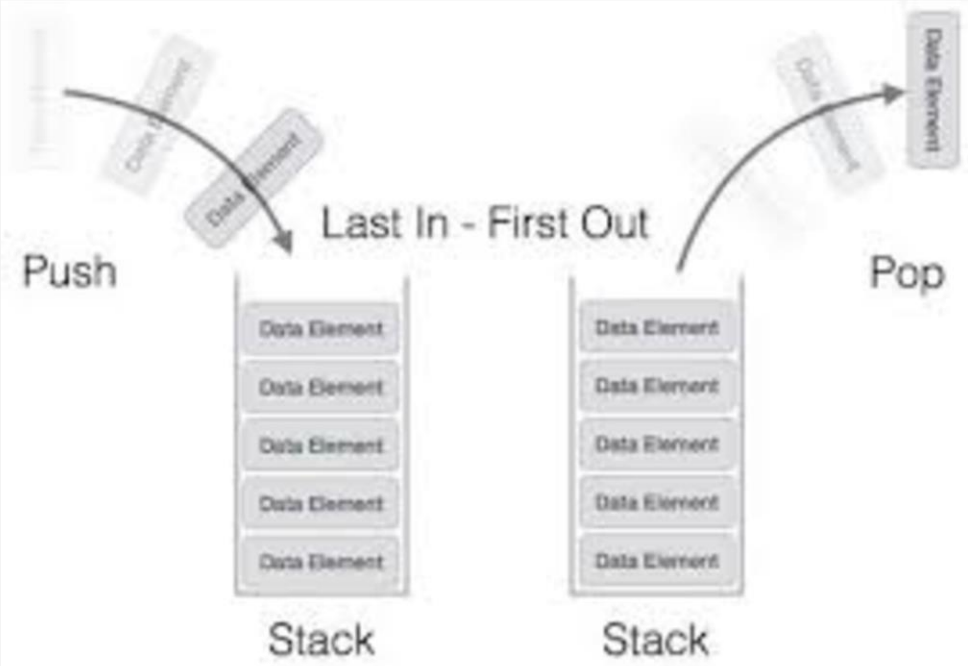
1. A STACK ADT, A CONCRETE DATA STRUCTURE FOR A FIRST IN FIRST OUT (FIFO) QUEUE.

1. Stack ADT
A stack is a data structure that follows the LIFO (Last In, First Out) principle, meaning that the last element inserted will be the first element taken out. Some main characteristics of a stack:
Push: Add an element to the top of the stack.
Pop: Get an element from the top of the stack.
Peek/Top: View the element at the top of the stack without removing it.
IsEmpty: Check if the stack is empty.

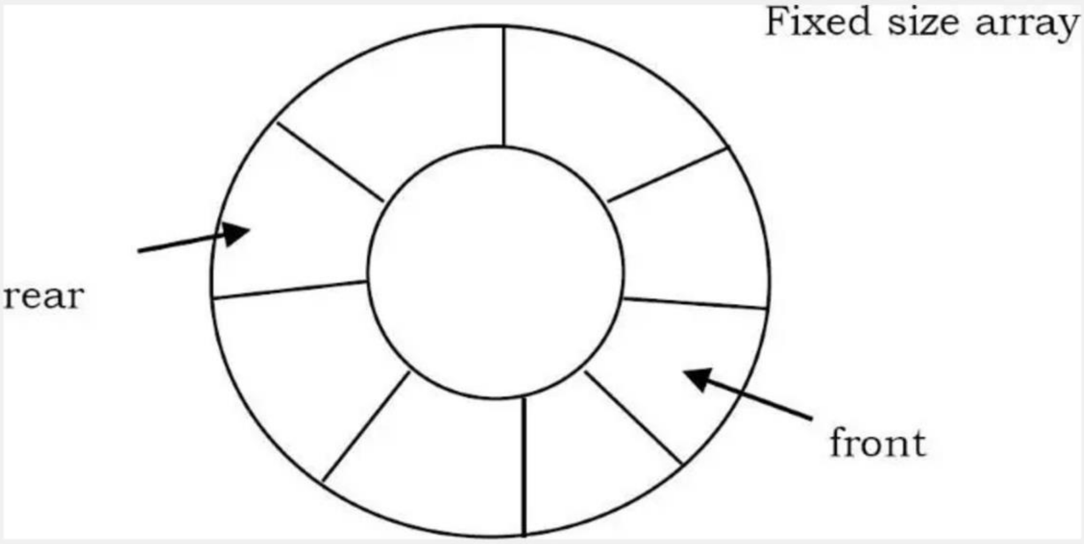
2. Queue ADT
A queue is a data structure that follows the FIFO (First In, First Out) principle, meaning that the first element inserted will be the first element taken out. Main operations in a queue:
Enqueue: Add an element to the end of the queue.
Dequeue: Get an element from the front of the queue.
Front: View the element at the front of the queue without removing it.
IsEmpty: Checks whether the queue is empty or not.

3. FIFO Principle
FIFO (First In, First Out) is the main principle of a queue. This means that the first element to be put in will be the first element to be taken out, similar to queuing in a supermarket: first come first served.
In a queue, elements are added to the end of the queue (enqueue) and taken from the beginning of the queue (dequeue), ensuring that the earliest element is processed first. FIFO operations help to perform processes in order in a logical and efficient manner.

IMAGE OF ADT STACK



ADT QUEUE IMAGE

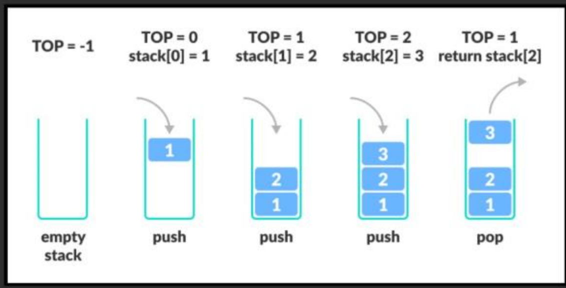


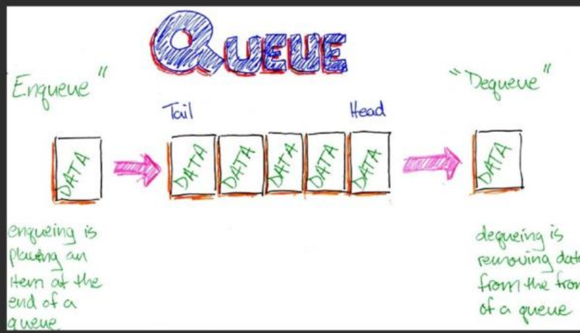
OMPARE DIFFERENT BETWEEN TACK AND QUEUE

Criterion	Stack	Queue
Operation Principle	LIFO (Last In, First Out): The last element added is the first one	FIFO (First In, First Out): The first e
Main Operations	- `Push` : - `Pop` : Removes and retrieves the top element. - `Peek` or `Top` : Views the top element without removing it. - `isEmpty` : Checks if the stack is empty.	- `Enqueue` : Adds an element to the end of the queue. - `Dequeue` : Removes and retrieves the front element. - `Front` or `Peek` : Views the front element without removing it. - `isEmpty` : Checks if the queue is empty.
Implementation	Can be implemented using arrays, linked lists, or software/system stacks.	Can be implemented using circular arrays, linked lists, or priority queues.
Applications in Programming	- Recursion: Used to store recursive function states. - Undo/Redo: Manages states for undoing actions. - Mathematical Expressions: Parses and evaluates expressions. - Bracket Matching: Checks if brackets in an expression are balanced.	- CPU Scheduling: Manages processes waiting to be executed in order. - Data Transmission: Used to send and receive data sequentially. - Print Queue Management: Holds documents waiting to be printed. - Graph Traversal (BFS): Stores the next vertices to visit in breadth-first traversal.
Advantages	- Simple and easy to understand. - Suitable for LIFO-based operations.	- Useful for handling tasks in FIFO order. - Manages data in situations requiring sequential processing.
Disadvantages	- Not suitable for FIFO scenarios. - Limited scalability in certain situations.	- Not suitable for LIFO operations. - Requires extra space for FIFO operations, especially with circular arrays.
Real-World Examples	- Undo (Undo) in text editors. - Call Management: Organizes recent calls at the top. - Depth-First Search (DFS) in graph traversal.	- Queue Management: Such as ticket queues or waiting for service. - Process Scheduling in CPU (Round Robin). - Breadth-First Search (BFS) in graph traversal.
Time Complexity	- Adding (`push`) and removing (`pop`) have O(1) time complexity.	- Adding (`enqueue`) and removing (`dequeue`) have O(1) time complexity for simple queues or circular arrays.

THERE ARE 5 WAYS TO IMPLEMENT STACK AND QUEUE

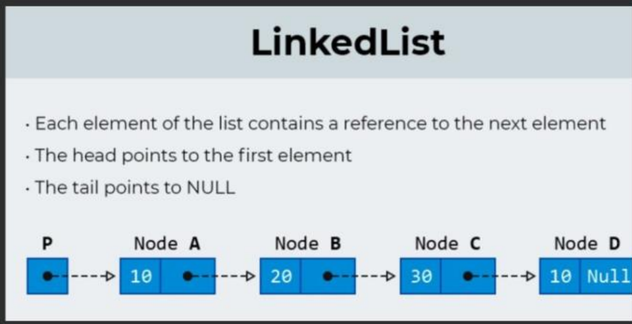
1. Array implementation
   - Stack: Use an array with a variable pointer or a high-level variable (top) to mark the top element position of the stack. When adding a new element (push), the top index will increase and when removing an element (pop), the top index will decrease.
   - Queue: Use an array with two pointers, front (front) and rear (rear), to mark the beginning and end of the queue. When enqueueing, the rear index will increase, and when dequeueing, the front index will increase.





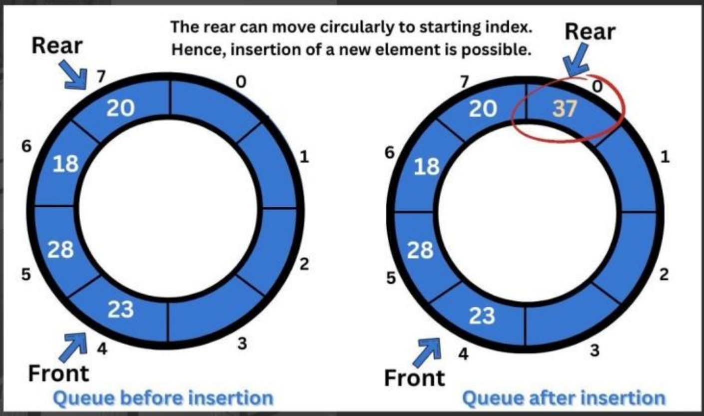
2. Implementation using Linked List

- Stack : Each element of the stack is a node in a linked list, usually with a variable top holding the address of the first node. When adding an element, the new node is linked to the top of the list (brought to the top), and when deleting, top moves to the next node.
- Queue : Uses two pointers front and rear to point to the first and last nodes of the queue. When adding an element (enqueue), the new node will be linked to rear. When deleting an element (dequeue), front points to the next node of the element.



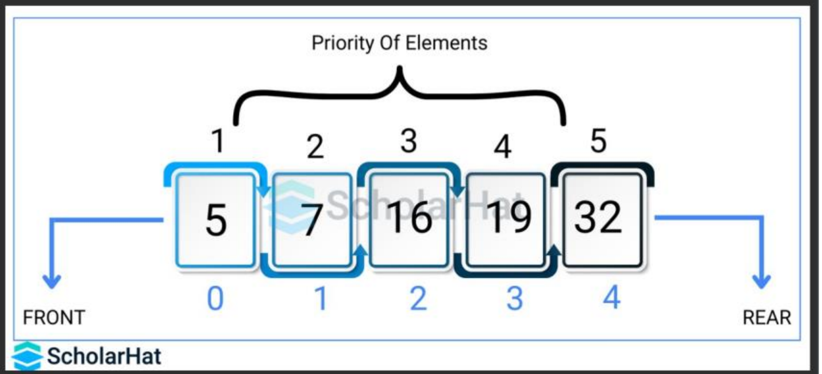
3. Loop Queue (Circular Queue)

- The queue solves the problem of memory waste in array growth. When the rear pointer reaches the end of the array, it can return to the first position if there is space. This helps to make good use of space, especially in the continuous loop of the system.



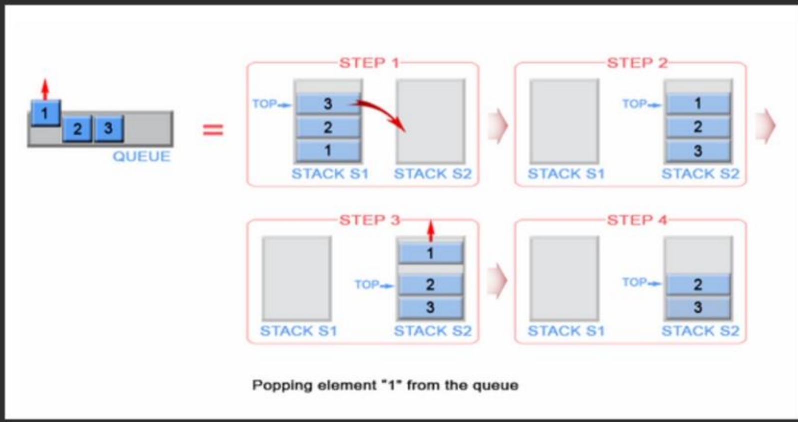
4. Priority Queue (Priority Queue)

- Priority queues arrange elements in priority order, commonly used in Dijkstra's algorithm like algorithm. Implementations can be a heap (level detection) or use array sorting, where elements with higher priority will be accessed first



5. Implementation with two stacks (for Queue)

- A special way to implement Queue is to use two stacks. The first stack holds the elements when enqueueing, and when dequeuing, the elements from the top of the stack will be transferred to the second stack, reversing the order to add the FIFO principle.

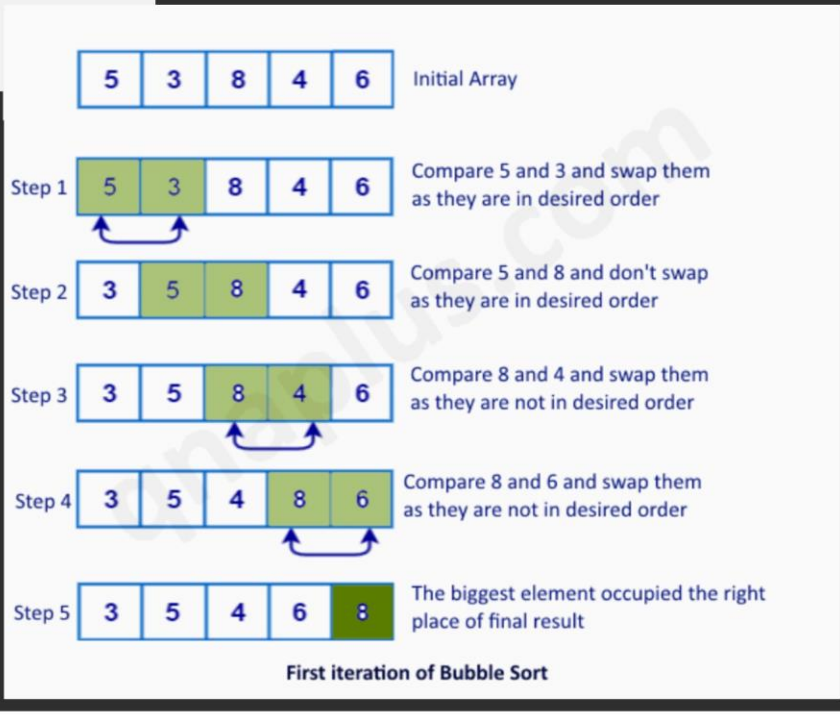



TWO
SORTING
ALGORITHMS.

1. BUBBLE SORT

- WITH THE NON-DECREASING ARRANGEMENT FROM LEFT TO RIGHT, OUR GOAL IS TO GRADUALLY MOVE THE LARGEST NUMBERS TO THE END OF THE ARRAY (FAR RIGHT).
- STARTING FROM POSITION 1, CONSIDER EACH PAIR OF 2 ELEMENTS IN TURN. IF THE ELEMENT ON THE RIGHT IS SMALLER THAN THE ELEMENT ON THE LEFT, WE WILL SWAP THESE 2 ELEMENTS. IF NOT, CONSIDER THE NEXT PAIR. BY DOING SO, THE SMALLER ELEMENT WILL "FLOAT" UP, WHILE THE LARGER ELEMENT WILL "SINK" AND MOVE TO THE RIGHT.
- AT THE END OF THE FIRST ROUND, WE WILL HAVE MOVED THE LARGEST ELEMENT TO THE END OF THE ARRAY. IN THE SECOND ROUND, WE CONTINUE TO START AT THE FIRST POSITION LIKE THAT AND MOVE THE SECOND LARGEST ELEMENT TO THE SECOND POSITION AT THE END OF THE ARRAY

ALGORITHM OPERATIONS



First iteration of Bubble Sort

ILLUSTRATION CODE

```java
public class BubbleSort {
    public static void bubbleSort(int[] arr) {  1 usage
        int n = arr.length;
        boolean swapped;
        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Hoán đổi arr[j] và arr[j + 1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            // Nếu không có hoán đổi, mảng đã được sắp xếp
            if (!swapped) {
                break;
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};
        bubbleSort(arr);
        System.out.println("Mảng sau khi sắp xếp: ");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

ILLUSTRATION CODE

ARRAY BEFORE SORTING

```
int[] arr = {64, 34, 25, 12, 22, 11, 90};
```

ARRAY AFTER SORTING

```
Mảng sau khi sắp xếp:
11 12 22 25 34 64 90
```

2. QUICK SORT

- THE QUICKSORT ALGORITHM IS ONE OF THE MOST WIDELY USED SORTING ALGORITHMS, ESPECIALLY FOR SORTING LISTS/ARRAYS WITH MANY ELEMENTS.
- THE QUICK SORT ALGORITHM IS A DIVIDE AND CONQUER ALGORITHM. THAT IS, THE ORIGINAL ARRAY IS DIVIDED INTO 2 ARRAYS, EACH OF WHICH IS SORTED SEPARATELY. THEN, THE SORTED OUTPUT IS MERGED TO FORM THE FINAL SORTED ARRAY.
- THE COMPLEXITY OF THE ALGORITHM IS O(N LOG N). THEREFORE, QUICK SORT IS SUITABLE FOR SORTING LARGE ARRAYS.
- TO BE MORE PRECISE, THE QUICKSORT ALGORITHM PERFORMS MULTIPLE ARRAY SPLITS BY COMPARING WITH THE MARKED ELEMENT (CALLED THE PIVOT). WHEN THE RECURSION ENDS, THE ARRAY IS SORTED. WHAT MAKES THE ALGORITHM FAST IS THE “IN-PLACE SORTING”. THAT IS, THE SORTING IS DONE RIGHT IN THE ARRAY WITHOUT CREATING A NEW ARRAY.

ALGORITHM
OPERATIONS

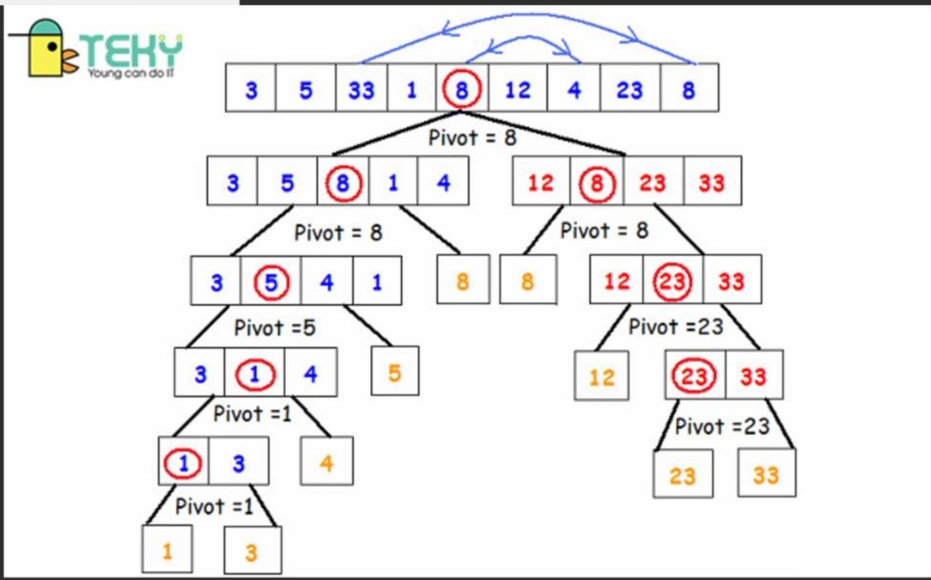


ILLUSTRATION CODE

```java
public class QuickSort {
    public static void quickSort(int[] arr, int low, int high) {  3 usages
        if (low < high) {
            // Chia mảng và lấy chỉ số của pivot
            int pi = partition(arr, low, high);

            // Đệ quy sắp xếp các phần tử trước và sau pivot
            quickSort(arr, low, high: pi - 1);
            quickSort(arr, low: pi + 1, high);
        }
    }

    private static int partition(int[] arr, int low, int high) {  1 usage
        int pivot = arr[high]; // Chọn phần tử cuối làm pivot
        int i = (low - 1); // Chỉ số của phần tử nhỏ hơn

        for (int j = low; j < high; j++) {
            // Nếu phần tử hiện tại nhỏ hơn hoặc bằng pivot
            if (arr[j] <= pivot) {
                i++;

                // Hoán đổi arr[i] và arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
```

```java
        // Hoán đổi arr[i + 1] và arr[high] (pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }

    public static void main(String[] args) {
        int[] arr = {10, 7, 8, 9, 1, 5};
        int n = arr.length;

        quickSort(arr, low: 0, high: n - 1);
        System.out.println("Mảng sau khi sắp xếp: ");
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

ILLUSTRATION CODE

ARRAY BEFORE SORTING

```
int[] arr = {10, 7, 8, 9, 1, 5};
```

ARRAY AFTER SORTING

```
1 5 7 8 9 10
```

COMPARE BUBBLE SORT AND QUICK SORT.

1. DEFINITION AND HOW IT WORKS

BUBBLE SORT:
DEFINITION: A SIMPLE SORTING ALGORITHM THAT WORKS BY CONTINUOUSLY CHANGING ADJACENT ELEMENTS IF THEY ARE OUT OF ORDER.
HOW IT WORKS: ITERATE THROUGH THE LIST MANY TIMES; IN EACH ITERATION, THE LARGEST (OR SMALLEST) ELEMENT WILL EVENTUALLY "FLOAT".
NOTABLE FEATURES: EASY TO UNDERSTAND, EASY TO IMPLEMENT BUT VERY SLOW WHEN VIEWING LARGE LISTS.

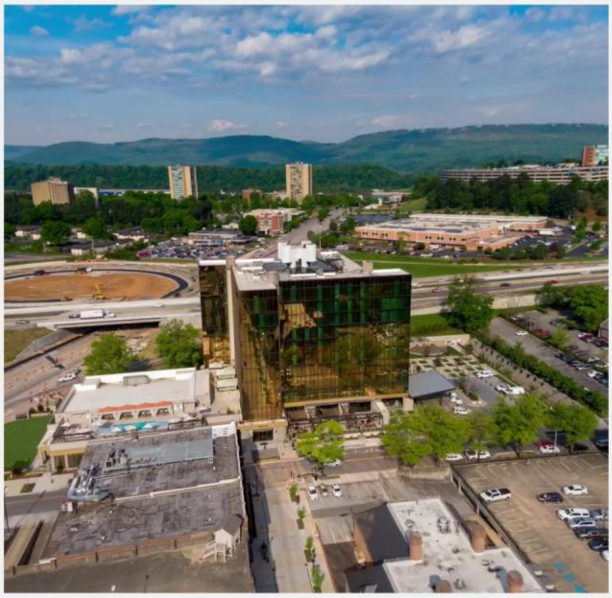
QUICK SORT:
DEFINITION: A QUICK SORTING ALGORITHM THAT WORKS BASED ON THE DIVIDE-AND-CONQUER MECHANISM (DIVIDE AND CONQUER), DIVIDING THE ARRAY INTO PARTS AND SORTING THEM INDEPENDENTLY.
HOW IT WORKS: CHOOSE AN ELEMENT AS THE "PIVOT", DIVIDE THE ELEMENTS INTO TWO GROUPS (SMALLER AND LARGER THAN THE PIVOT), THEN SORT EACH OF THESE GROUPS.
NOTABLE FEATURES: FAST AND EFFICIENT, ESPECIALLY WITH LARGE DATA.

COMPARE BUBBLE SORT AND QUICK SORT.

5. ADVANTAGES AND DISADVANTAGES

	Bubble Sort	Quick Sort
Advantages	- Simple, easy to understand and implement	- Fast and efficient for large datasets
	- Suitable for small or nearly sorted lists	- Has a good average-case complexity, effective on large arrays
Disadvantages	- Slow, unsuitable for large datasets	- Unstable, may change the order of identical elements
	- High time complexity	- Requires careful pivot selection to avoid the worst case

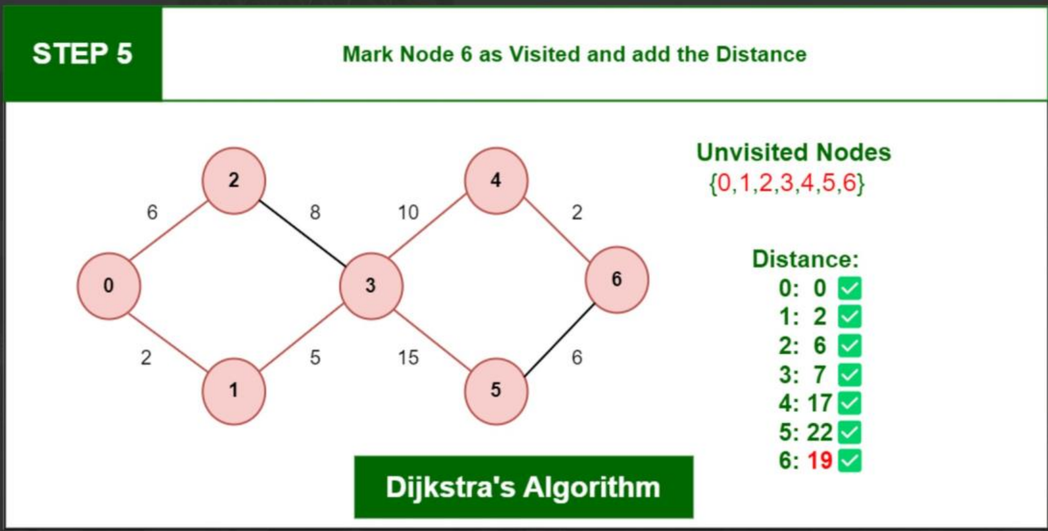
TWO NETWORK SHORTEST PATH ALGORITHMS.



1. DIJKSTRA'S ALGORITHM

- Purpose: Find the shortest path from a source node to all other nodes in a non-negative weighted graph.
- Procedure: It works by initializing the distance from the source to all nodes as infinity, except the source itself, which is set to 0. The algorithm then iteratively selects the unvisited node with the smallest known distance, updates its neighbors with the smallest possible distance, and repeats until all nodes are visited.

1. DIJKSTRA'S ALGORITHM



ILLUSTRATIVE EXAMPLE

Giả sử ta có 4 điểm A, B, C, và D với các cạnh và số như sau:

- A → B = 1
- A → C = 4
- B → C = 2
- B → D = 5
- C → D = 1

Bắt đầu từ điểm A:

1. **Bước đầu tiên :**
   - Khoảng cách: A = 0, B = ∞, C = ∞, D = ∞
   - Đặt A là "đang chuẩn bị", khoảng cách từ A đến B là 1, và từ A đến C là 4.
   - Cập nhật: A = 0, B = 1, C = 4, D = ∞.
2. **Bước thứ hai :**
   - Chọn B (có khoảng cách nhỏ nhất 1).
   - Từ B, khoảng cách đến C là$1 + 2 = 3$(nhỏ hơn 4, nên cập nhật) và khoảng cách đến D là $1 + 5 = 6$.
   - Cập nhật: A = 0, B = 1, C = 3, D = 6.
3. **Bước thứ ba :**
   - Chọn C (có khoảng cách nhỏ nhất 3).
   - Từ C, khoảng cách đến D là$3 + 1 = 4$(nhỏ hơn 6, nên cập nhật).
   - Cập nhật: A = 0, B = 1, C = 3, D = 4.
4. **Bước cuối cùng :**
   - D là điểm cuối cùng và đã đạt được khoảng cách ngắn nhất.

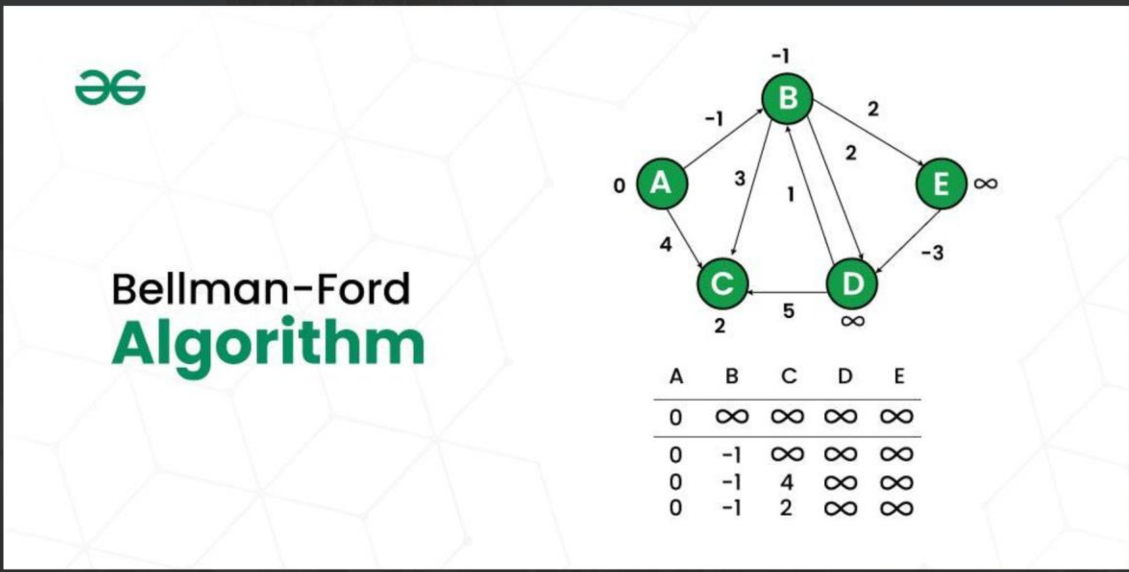
Kết quả cuối cùng là:

- Khoảng cách từ A đến B là 1
- Khoảng cách từ A đến C là 3
- Khoảng cách từ A đến D là 4

2. BELLMAN-FORD ALGORITHM

- Objective: Find the shortest path from a source node to all other nodes in the graph that may have negative edge weights.
- Procedure: The distance initialization algorithm is similar to Dijkstra's algorithm but works by relaxing each edge in the graph $V-1$ V−1 times (where V V is the number of vertices). Each iteration updates the distance to the neighboring nodes and if a shorter path is found, it replaces the current distance.

2. BELLMAN-FORD ALGORITHM



THE IDEA OF IMPLEMENTING DIJKSTRA'S ALGORITHM

Dijkstra's algorithm is based on the principle of reduction. In which more accurate values will gradually replace an approximation to the exact distance until the shortest distance is achieved. The approximation to each vertex is estimated to be much larger than the actual distance and will gradually be replaced by the smallest value of the old value equal to the length of a newly found path.

The algorithm uses a priority queue combined with a greedy algorithm that selects the nearest unprocessed vertex and performs this reduction on all edges it traverses.

THANK YOU