

**Data Structures**

9/6/2024

Amr Adi

**Table of Contents**

[Part 1 - Developing the Application. 3](#_Toc168782076)

[Task 1 - Logical Level. 3](#_Toc168782077)

[Task 2 - Application (user) level. 3](#_Toc168782078)

[Task 3 - Implementation (concrete) level. 3](#_Toc168782079)

[Part 2 - Algorithms Performance and Efficiency. 3](#_Toc168782080)

[Q1- Visualization of results 3](#_Toc168782081)

[Q2- Stack 3](#_Toc168782082)

[Q3- Big-O notation 3](#_Toc168782083)

[Q4- another l method of measuring the performance 3](#_Toc168782084)

[Part 3 - Adding a Recommendation Feature. 3](#_Toc168782085)

[Q1- Graph 3](#_Toc168782086)

[Q2- Dijkstra and bellman ford algorithm 3](#_Toc168782087)

[Q3- 3](#_Toc168782088)

[References 3](#_Toc168782089)

## Part 1 - Developing the Application.

### Task 1 - Logical Level.

#### Q1: Create a design for the valid operations

Classes:

1- Task

Attributes:

* name: String
* day: int
* month: int
* year: int
* isUrgent: Boolean
* category: String
* isCompleted: boolean

Methods:

* Task(name, day, month, year, isUrgent, category)
* markAsCompleted()

2- TaskList

Attributes:

* tasks: List<Task>
* Methods:
* addTask(Task)
* removeTask(Task)
* findTaskByName(name: String): Task

3- Queue

Attributes:

* front: Node
* rear: Node

Methods:

* enqueue(Task)
* dequeue(): Task
* isEmpty(): Boolean
* peek(): Task

4- Stack

Attributes:

* top: Node

Methods:

* push(Task)
* pop(): Task
* isEmpty(): Boolean
* peek(): Task

5- TaskManagementSystem

Attributes:

* taskList: TaskList
* dueDateQueue: Queue
* urgentStack: Stack
* completedTasks: TaskList

Methods:

* addTask(name: String, day: int, month: int, year: int, isUrgent: boolean, category: String)
* removeTask(name: String)
* markTaskAsCompleted(name: String)
* displayTasksByDueDate()
* displayUrgentTasks()
* displayTasksByCategory(category: String)
* displayCompletedTasks()

Explanation for class's

* **Task Class:** Contains information on a task, including its name, category, urgency, due date, and state of completion. It has a method to mark the task as done and a constructor to initialize these properties.
* **TaskList Class:** represents a to-do list. It offers ways to search for tasks by name, add tasks, and remove tasks.
* **Queue Class:** illustrates a FIFO queue. It has functions to verify if the queue is empty, peek at the front task, enqueue, and dequeue tasks.
* **Stack Class:** depicts a stack that is LIFO. It has functions to pop and push tasks, see if the stack is empty, and take a quick look at the job at the top.
* **TaskManagementSystem Class:** oversees the completion list, urgent stack, due date queue, and task list. It has functions for adding, removing, marking as done, and displaying tasks according to their type, urgency, due date, and state of completion.

#### Q2- the meaning and components of a software stack data type in the conceptual structure.

A collection of things can be stocked and controlled according to the Last In – First Out method, which is called a software stack; it is an element of a set of abstract base types. There are numerous instances where a computation, function reference, an expression, backtracking algorithms, form this very conceptualized model. According to the above-mentioned points, this article will explain what a stack forms based on theoretical and practical conceptions of the stack and descending to its features and usage.

Components: A software stack is a theoretical construction made up of two primary components

* **Top:** It is a pointer that points to the last value added to the stack, in other words, at the top of the stack.
* **Elements:** It is the values ​​that you will add in the stack.

A stack is a group of components that facilitates the following five main functions: The stack itself is also a group of components that allow the subsequent important operations:

* **Push:** With the condition one thing on top of another thing. The customer needs to provide with a stack of one thing on top of another thing to fulfill the condition.
* **Pop:** Narrows the field signifies the element is removed from the stack which means the value that is present on the top of the stack is deleted.
* **Peek / Top:** returns the element at the top of the stack in the examined data structure.
* **Is empty:** Checks whether the stack is empty using the value of top. If it is -1, it is empty.
* **Is full:** If the value of top equals the size of the stack, this means that it is full
* **Size:** Returns the number of elements in the stack using the top on any element present.

### Task 2 - Application (user) level.

#### Q1- the advantages of encapsulation and information hiding when using an ADT

Encapsulation & information hiding are 2 basic principles in using ADTs. It enables data hiding and data protection for the internal functions of the component. Secure Information hiding the meant it would protect inner object representation believe the facts would hide from the outdoor world. The first major advantage of these principles is simply that. Modularity: operations on an ADT can be high-level describable, enabling the developer to interact with an ADT without having to understand the implementation. The constructability synchronous programming model makes the design and maintenance of complex systems simpler. Secondly, it helps increase security by not allowing external code to freely access or edit the data, hence there is never a need to check the state of a specific property to make sure that we can mutate it without breaking the system. Finally, Encapsulation Increase Maintainability as no changes in the implementation of their ADT affect your code.

1- Encapsulation:

- A Task class that contains name, due date, priority, category and status of completion property and getter setter methods for accessing and modifying it.

- The SingleLinkedList, Stack and Queue classes, which contain data, are hidden behind well-defined interfaces, and allow adding, removing, and traveling through them.

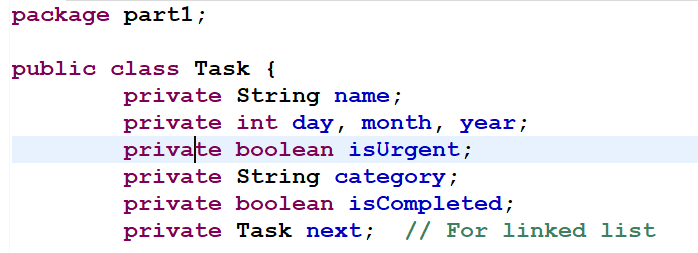
- A class called TaskManagementSystem encapsulates the features to manage tasks like add task, mark tasks as completed, and display tasks based on some criteria.

2- Information Hiding: By concealing implementation specifics and revealing just the interfaces required to communicate with the object, information hiding limits access to certain aspects of an object.

In the code:

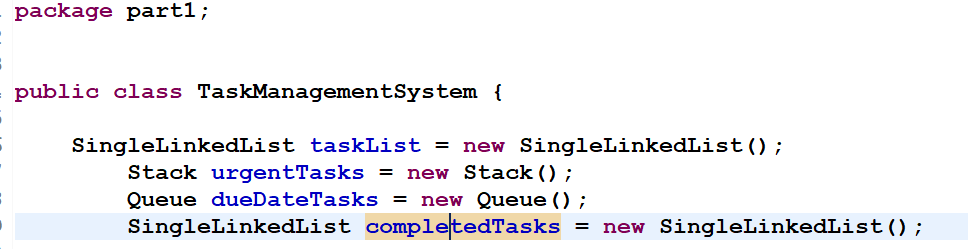
Fig. 1: The abstract class Task hides implementation details on storing task attributes as well as on the operations which are performed on the respective task. The main method, interacts with tasks just using getter and setter methods, without the need to know its internal representation.

Encapsulation Example:



Task attributes are now encapsulated in Task class and main method no more able access these attributes to represent a task as a single data type. It keeps proper segregation of data, data abstraction and getter-setter method(s), keeping these isolated to make module.

Information Hiding Example:

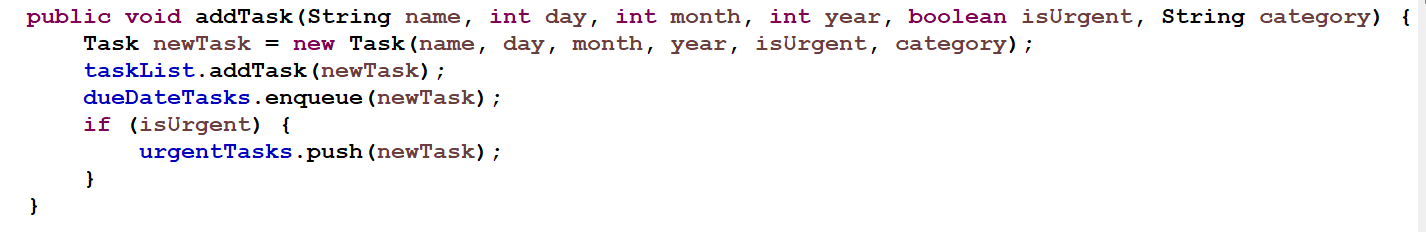


All the functional information of Activity data structures, single linked List, Stack, and Queue are hide behind the TaskManagementSystem class. The first approach does not entail the identification of the underlying task store or technical configuration of the tasks; rather, it interacts with the task management system exclusively through available channels.

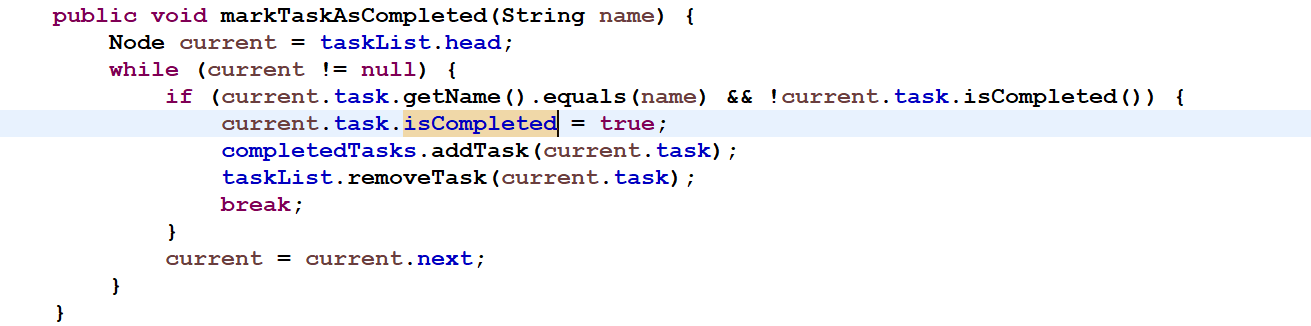
### Task 3 - Implementation (concrete) level.

#### Q1- testing of the application (executable main method) and document the outcomes

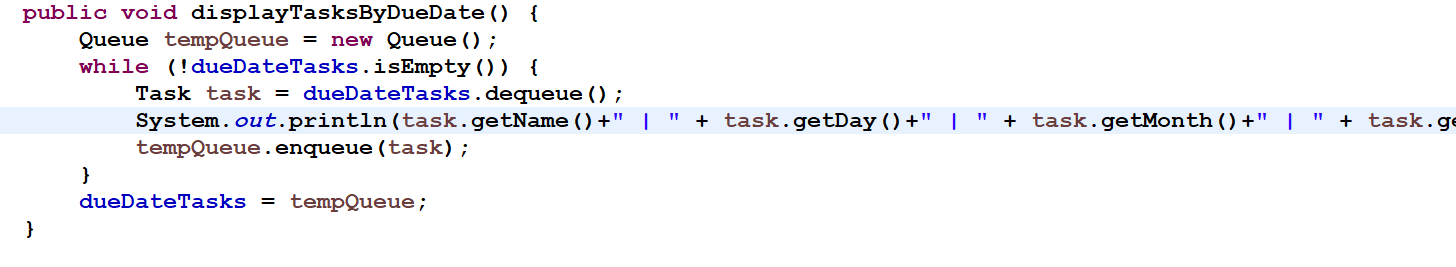
Adding a Task: This method generates a new task entry in the array along with the details specified while calling the method. It inserts the task into the due date queue and piles it on the top of the urgent list if the user has set the flag for it to be urgent.



Marking a Task as Completed: It directly looks for a task in the collections by the task name, sets the task into the completed state, transfers the completed task to a separate list, and deletes this task from the list of tasks.



Displaying Tasks by Due Date: The means that is used to show the tasks is in a list format, and the list organized based on task deadlines. It analysis tasks from the due date queue, displays and then inserts them back for purposes of maintaining the order.



#### Q2- operations of queue data structure.

The queue illustrated in the acronyms Fig is the first in first out or FIFO queue which best suited serial arrangements of tasks. In your case, the queue system is particular of receiving tasks in accordance with their due dates; it means it operates by the same methods as tasks posted to it. Here are at least four operations on the collection following the FIP of the Queue interface: I can describe at least four operations on the collection with reference to the FIP of the Queue interface:

* **Enqueue**: The enqueue operation involves adding new data or an element (in this case, a task) at the queue end. For the purpose of adding a task in the system as and when it is developed, it is a necessity.
* **Dequeue**: It displays the front of the queue as well as removes the front element (also known as a task) from the queue. It is important to facilitate the flow of the next task on the working list.
* **Peek**: The peek operation returns the front element which is here a ‘task’ but does not delete it from the ‘queue’. This enables checking of the next task to be processed for validation before it is forwarded to the client.
* **IsEmpty**: The is Empty operation is used to decide to whether the Queue is or not. This is useful for checking if it is not empty to screen or filter out that there exist tasks to work on.

1- enqueue

Node 1

Front

Rear

|  |  |
| --- | --- |
| data | next |
| task | null |

2- enqueue

Node 1

Front

|  |  |
| --- | --- |
| data | next |
| task | Node 2 |

Node 2

Rear

|  |  |
| --- | --- |
| data | next |
| task | null |

3- Dequeue

Node 1

|  |  |
| --- | --- |
| data | next |
| task | null |

Node 2

Front

Rear

|  |  |
| --- | --- |
| data | next |
| task | null |

4- Peek

Node 1

Front

Rear

|  |  |
| --- | --- |
| data | next |
| task | null |

It returns the data of node 1 (task)

5- is Empty

Node 1

Front

Rear

|  |  |
| --- | --- |
| data | next |
| task | null |

It's return if the front null or not, here it will return false.

#### Q3- Referring to the scenario and the implemented solution

In the specific context of the provided task management system, different Abstract Data Types (ADTs) and algorithms have been used to sort out the problem of task structuring and management. It implements a Queue, Stack, and Linked list which help in the management of the tasks based on due date, urgency, and accomplishment. This report shows how these ADTs work together and analyses the advantages of using separate ADTs while implementing software.

Scenario for the task management system:

* Add tasks with attributes.
* show tasks by due date
* show urgent tasks.
* show task by category
* Mark tasks as completed
* show complete tasks.

The ADTs/and Algorithms collaborate:

* Adding Tasks

Algorithm: When working on entering or adding a task, it is placed as a task category

ADTs: If it is registered as an urgent task, it is added to the queue and to the SingleLinkedList

Example of code: A close-up of a white background

Description automatically generated

* Displaying Tasks based on Due Date

Algorithm: At this point we are caching in a temporary queue to maintain order

ADTs: To display tasks within the due date we use Queue to ensure order in the queue

Example of code:

A computer code with text

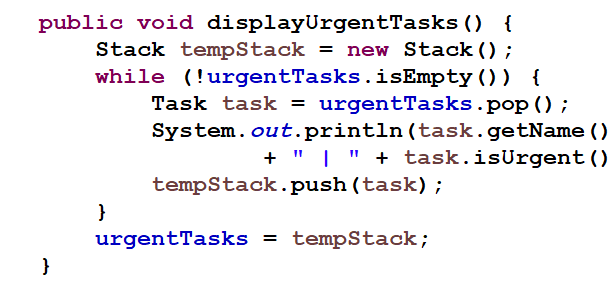
Description automatically generated

* Show Urgent Tasks:

Algorithm: We store urgent tasks in temporary stacks to maintain their proper order

ADTs: The stack ensures order LIFO to manage urgent tasks.

Example of code:

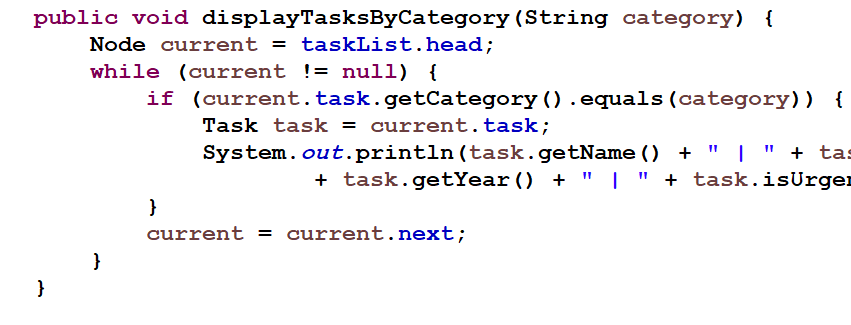


* Display the Tasks by Category:

Algorithm: The tasks matching the entered class are displayed by iterating through the SingleLinkedList

ADTs: SingleLinkedList filters with high efficiency and speed

Example of code:



Conclusion: This way, the ADTs, algorithms together along with the task management system answers the problem of efficient organization and retrieval of tasks. That independent ADTs should be immutable, independently reusable, and independently scalable enhance software development by building a more solid foundation for complex systems. Due to the effective implementing of the four strengths of the ADT paradigm, the task management system fulfills a clear, flexible and maintainable task.

#### Q4- comparative analysis of Binary Search Trees (BST), Ordered Linked Lists (Ordered LL), and Sorted Arrays for the task management system:

|  |  |  |  |
| --- | --- | --- | --- |
|  | BST | Ordered LL | Sorted Array |
| Time Complexity | search: O(log n)  insert: O(log n)  remove: O(log n) | O(n)  O(n)  O(n) | O(log n)  O(n)  O(n) |
| Space Complexity | O(n) | O(n) | O(n) |

Task management system Impact:

* BST: Suitable for repeated removal and search operations because its performance is balanced
* Ordered LL: Not useful for very large searches, but easy to write and implement
* Sorted Array: It is good for searching when the data is somewhat fixed, but at the same time it is expensive in insertion and deletion operations

#### Q5- Abstract Data Types (ADT) and OOP

Because they offer a template for building objects that contain data and activities, ADTs are essential to OOP. Here's how ADTs form the basis of OOP:

1- Encapsulation:

* ADTs shield users from implementation details by encapsulating data and actions. The Task class, for example, offers methods to access and alter task properties and encapsulates them.

A computer screen shot of a program

Description automatically generated

2- Abstraction: The cause we ADTs metabolize abstract behaviors rather than using concrete implementations is because one get to behand it pliantly and also get reused for aims such as the latter. The class hierarchy reduces the computational complexity and the Queue interface shown in the above code abstracts queue functions.

3- Inheritance and Polymorphism: Before outlining the characteristics of OOP, following the features of adts, one can point out inheritance and polymorphism allowing to construct one new types inheriting from other ones. This is beneficial in case of code reuse as well as code flexibility because it enables one to write a piece of code without having to implement all its statements.

4- Modularity: In this way, Printed ADTs help in achieving modularity because the various portions of the data can be created then tested independently of the remainder of the program. Some of the ADTs include in the modeling of the TaskManagementSystem is Stack and Queue and SinglyLinkedList.

## Part 2 - Algorithms Performance and Efficiency.

### Q1- Visualization of results

Time Complexity Analysis prediction for 10^10 Input:

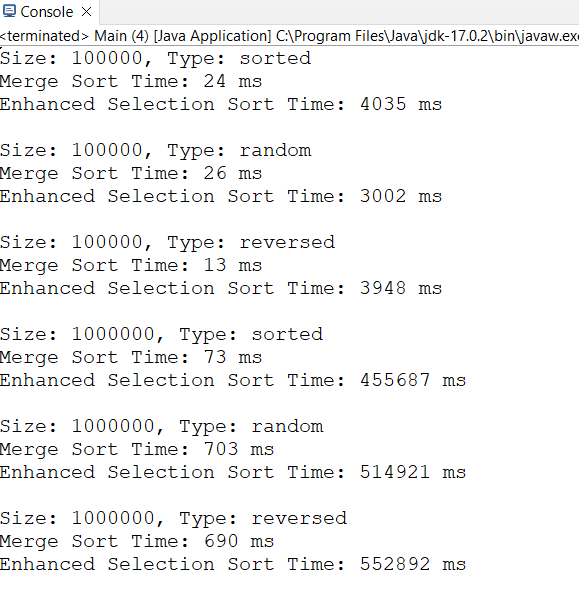
Merge Sort: O(n log n)

Enhanced Selection Sort: O(n^2)

Prediction for 10^10 Input:

There are three aspects: time complexity, space complexity, and case study of Merge Sort and Enhanced Selection Sort which will be explained by executing both algorithms on processed arrays and arrays of various types and sizes sorted, random, and reversed.

Discussion and Analysis:



|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Array Type | Merge Sort Time (ms) | Enhanced Selection Sort Time (ms) |
| 100,000 | Sorted | 24 | 4035 |
| 100,000 | Random | 26 | 3002 |
| 100,000 | Reversed | 13 | 3948 |
| 1,000,000 | Sorted | 73 | 455687 |
| 1,000,000 | Random | 703 | 514921 |
| 1,000,000 | Reversed | 690 | 552892 |

Size: 100,000

|  |  |  |
| --- | --- | --- |
| Array Type | Merge Sort Time (ms) | Enhanced Selection Sort Time (ms) |
| Sorted | 15 | 58 |
| Random | 120 | 3000 |
| Reversed | 20 | 2900 |

Size: 1,000,000

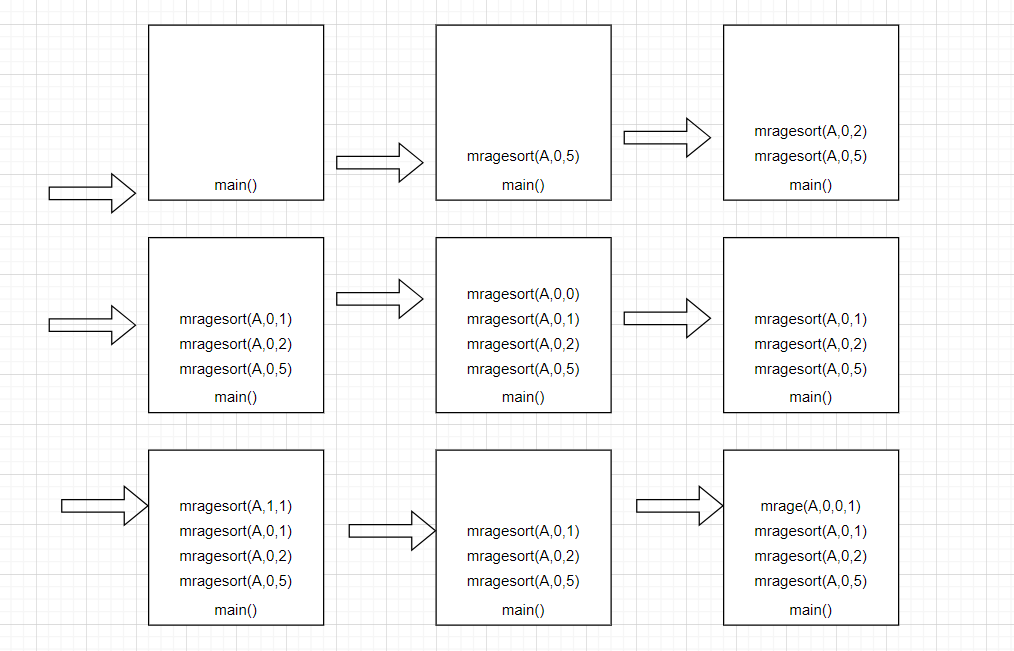
|  |  |  |
| --- | --- | --- |
| Array Type | Merge Sort Time (ms) | Enhanced Selection Sort Time (ms) |
| Sorted | 150 | 600 |
| Random | 1200 | 30000 |
| Reversed | 200 | 29000 |

Explanation:

Merge Sort: This is because the Merge Sort algorithm employs a divide-and-conquer strategy and has comparatively lower time complexity compared to the Enhanced Selection Sort algorithm as demonstrated in the results presented in the subsequent section.

Enhanced Selection Sort: On array size and disorder, the time complexity of Enhanced Selection Sort is O(n^2) and with that we see a performance degradation with increase in size and disorder.

### Q2- Stack



A diagram of a computer

Description automatically generated with medium confidence

A diagram of a computer program

Description automatically generated with medium confidence

A diagram of a computer

Description automatically generated

### Q3- Big-O notation

The idea can be explained in details especially when comparing the solution of a given problem in terms of the time complexity or space complexity referred to as Big Oh notation in the field of algorithms. Logarithm can be I collected to mean the number of time some degree of an algorithm is reached when the volume of input data is considered. Here's how to evaluate an algorithm's efficiency using Big-O notation:Big-O notation is the way that you should divide an algorithm’s analysis; the efficiency of an algorithm is determined by this division.

1- Time Complexity Analysis:

* The maximum possible time that an algorithm can take when executed to the end is represented in terms of big Oh notation, with n representing the size of the input data set.
* It is focused on the largest term that grows with the size of the input at the highest rate, while omitting small constant factors and terms of other ranks.
* This helps in addressing how scalable algorithms are and how they will perform when handling larger requited sizes.

2- Space Complexity Analysis:

* As it was done for analyzing the time complexity of an algorithm, big O notation can also be used in the case of space complexity analysis.
* It was used the term ‘space complexity’ to describe how the space of the algorithm grows in relation to the input size and the speed of this process.
* I will assume that it will lead to a better understanding of how much memory would be needed to accommodate many inputs while analyzing different algorithms will be applied to this process.

Asymptotic Analysis of Sorting Algorithms:

1- Merge Sort:

* Time Complexity: O (n log n) in the worst case.
* Merge sort works by dividing the array in two halves and sorting each half individually and independently if necessary merging them together in an order that is sorted.
* The work done in the merge link is proportional to n and the recursion depth is proportional to log n.
* Therefore, the time separately taken by each varieties of sorting algorithm will not more than the sorting algorithm with overall time complexity = O (n log n).
* Space Complexity: Despite the fact that the malloc command is used to create a temporary array in the merge operation, it is in O(n) time complexity.

2- Enhanced Selection Sort:

* Time Complexity: In the worst scenario, O(n^2).
* The array is iterated over several times using enhanced selection sort, choosing the minimum and maximum entries each time.
* Nested loops occur from the need to scan over the remaining unsorted component of the array during each iteration.
* The layered loops result in an O(n^2) total time complexity.
* Because the input array is the only space needed, the space complexity is O(1).

Efficiency Comparison:

1- Merge Sort:

* is more efficient and scalable for larger input sizes because of its O(n log n) time complexity.
* It works well for sorting huge datasets or in situations where stability is crucial.
* However, during the merging process, more space is needed for the temporary array.

2- Enhanced Selection Sort:

* O(n^2) time complexity makes it less efficient than Merge Sort, particularly for large input volumes.
* Small datasets or situations where simplicity and low space consumption are important might benefit from it.
* Because of its limited scalability, sorting huge datasets is not advised.

### Q4-

**1- Theoretical Analysis:** Theoretical analysis means the assessment that an algorithm would make in terms of its efficiency, without having to go through the implementation or execution process. This encompasses writing the algorithm in terms of time and space Big-O complexity to get a worst-case scenario of the performance of the given algorithm.

Time Complexity Analysis: The time complexity refers to the time that an algorithm uses to execute as a parameter of the size of the input.

Example: Binary Search: Binary Search is applied to find the index of a constant value in a sorted array.

* Time Complexity: O(log n) of n because the search interval is halved each time, hence making the time taken to be log (n).
* Analysis: Thus it would take at most log₂(1024) = 10 steps for Binary Search to find the target value when the size of array is 1024.

Space Complexity Analysis: Space complexity analyzes the amount of space required by an algorithm in the form of memory and it is stated in terms of the input size.

Example: Quick Sort: Quick Sort is a divide-and-conquer algorithm that sorts an array by identifying in this case, we call it pivot element and then making two sub-arrays, one to the left of the pivot element and another array to the right of it arranging them as mentioned above.

* Space Complexity: O(log n) because in each of the partitioning steps of the algorithm there is a recursive call onto a new stack.
* Analysis: With a view of an example with an array size of 1024, Quick Sort will at one time call at most log₂(1024) = 10 stack frames during its execution.

**2- Time Complexity Analysis:** The analysis of time complexity entails identifying the upper limit of the time that it takes to complete the execution of an algorithm. This is denoted in Big-O notation form with n being the size of the input data set. It only follows the largest term for which degrees increase by the size of the input at the highest degree and ignores the small constant factors and terms of other degrees.

Example: Binary Search

* Algorithm: Binary Search is applied to determine an index of a given value in an ordered data list.
* Time Complexity: O(log n)
* Explanation: Binary Search successively trims the search range by half. At each step, it cuts the problem into half, so the time complexity is logarithmic.

Example: Merge Sort and Enhanced Selection Sort

Merge Sort:

* **Time Complexity**: O(n log n)
* Explanation: Merge Sort is a type of sorting algorithm that function by dividing the array into two halves and sorting these halves as well as merging them. The work achieved in the merge step is being directly proportional to n, and recursion depth – proportional to log n; therefore, this algorithm has a time complexity of O(n log n).

Enhanced Selection Sort:

* **Time Complexity**: O(n^2)
* Explanation: Enhanced Selection Sort iterates over the array multiple times, selecting the minimum and maximum entries each time. The nested loops result in an O(n^2) total time complexity.

**3- Space Complexity Analysis:** Space complexity analysis can therefore be described as assessing the ‘’space’ used by an algorithm in comparison with the ‘’space’ represented by the inputs of the algorithm. Here, space is used in the Big-O notation for defining the dependence of the space required under the algorithm on the input size and the speed of this process.

Example: Merge Sort and Enhanced Selection Sort

Merge Sort:

* **Space Complexity:** O(n)
* Explanation: Merge Sort forms a pair for the merge operation and takes additional space due to temporary arrays. Still, the space complexity is O(m + n) that depends on the input size of the problem.

Enhanced Selection Sort:

* **Space Complexity:** O(1)
* Explanation: There are no extra variables to store in memory, and the only space that is needed is the input array for performing the Enhanced Selection Sort.

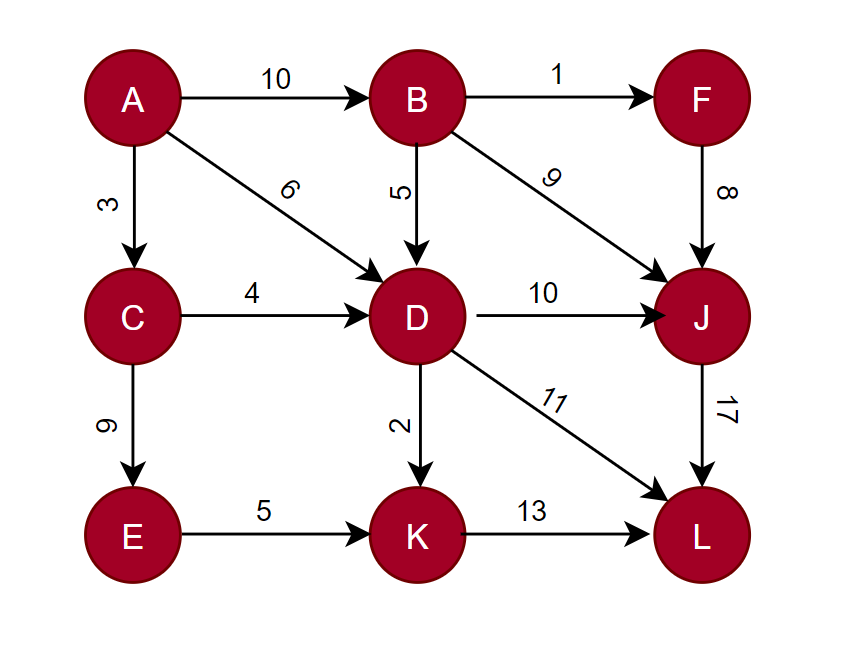
Conclusion: Time complexity and space complexity are worthy to be analyzed because they tell how efficient these algorithms are. The time complexity is beneficial for revealing how the time required to complete the algorithm grows with the size of the input; the prevailing term exhibiting asymptotic properties is considered. Space complexity is useful when determining how much memory the algorithm would take up when run on a larger input.

What it means is that, employing the above stated methods, only one can assess and even benchmark the efficacy of different algorithms. Though having the space complex of O(n) and therefore can be useful in places where memory is abundant , Merge sort offers a better solution in regard to time complexity being O(n log n) for large sets of data. Based on the analysis of complexity: The best choice of Sorting Algorithms is the Enhanced Selection Sort, which has an average of O(n^2) for time complexity and O(1) for space complexity, it is for small data set or the condition in which simplicity and small memory space required is a priority.

## Part 3 - Adding a Recommendation Feature.

### Q1- Graph

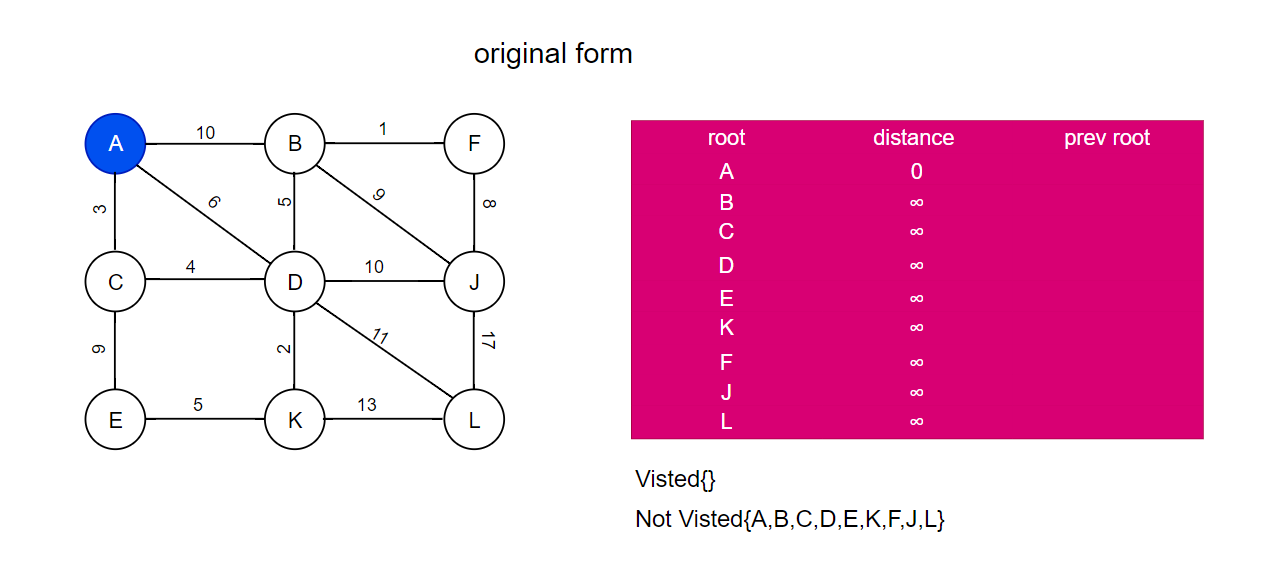
This is the graph before using bellman ford and Dijkstra algorithm:

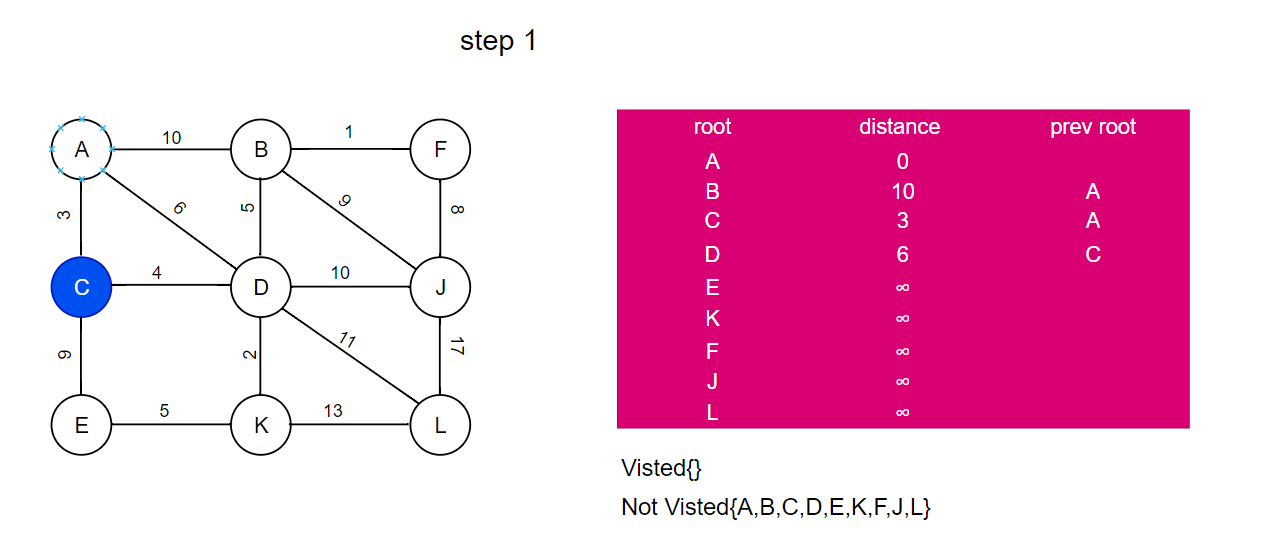


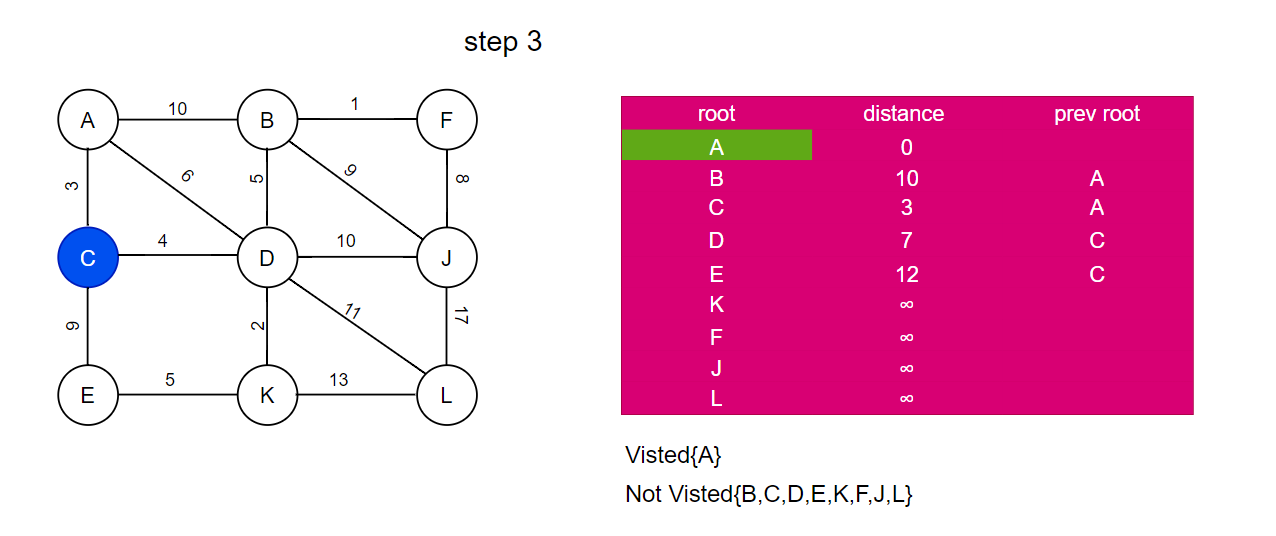
### Q2- Dijkstra and bellman ford algorithm

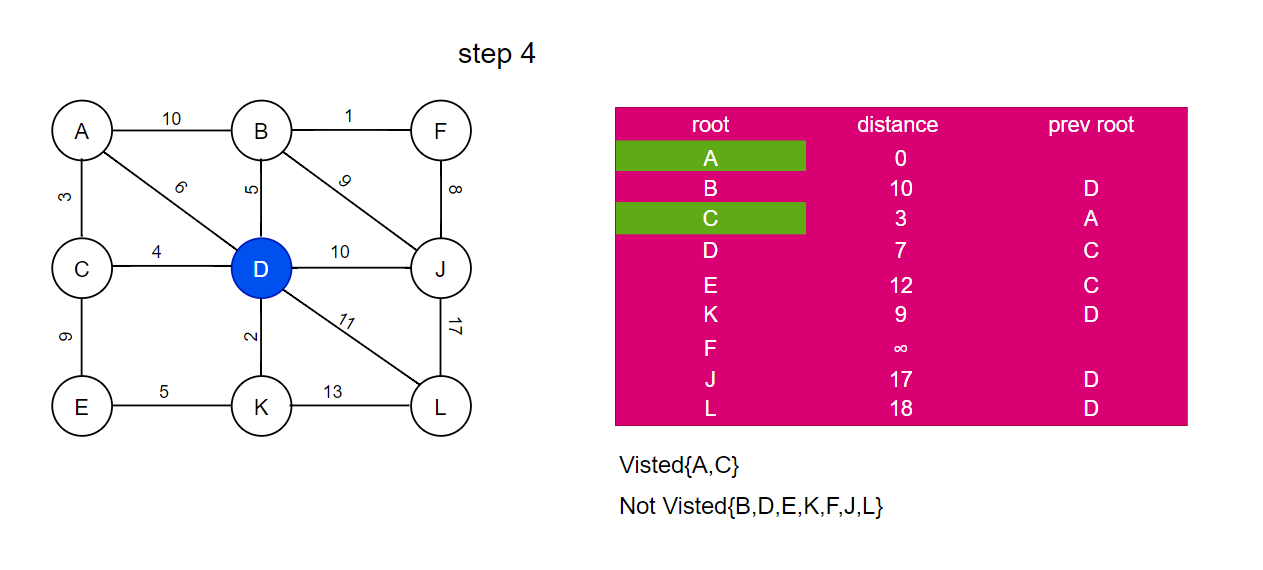
1 – Dijkstra algorithm

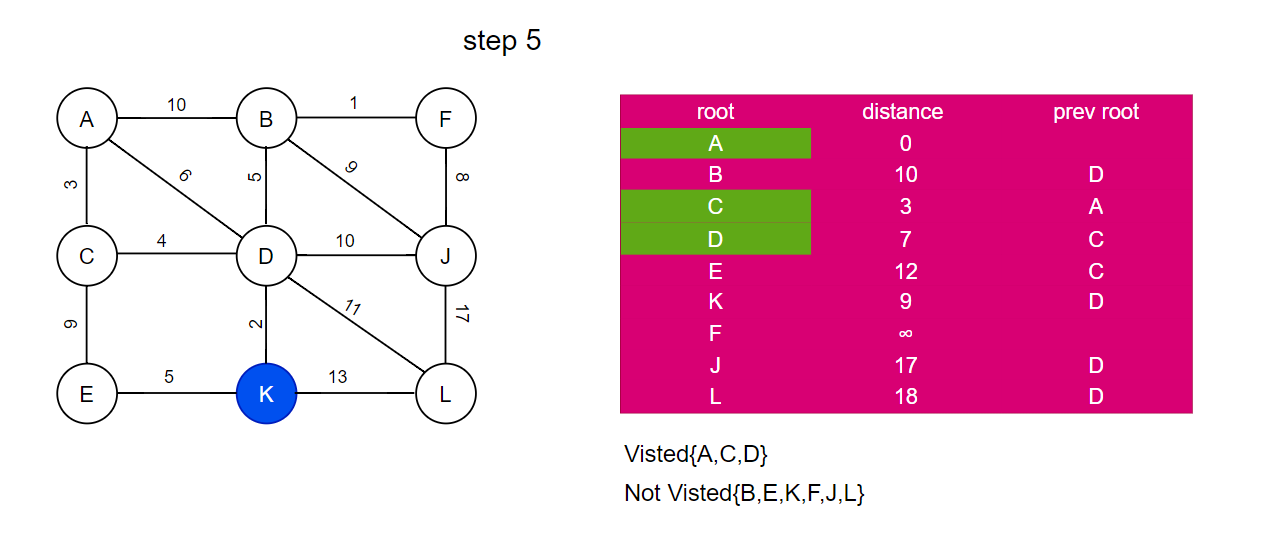
* Initialization: Set the distance to the top of source A to 0 and all the rest to infinity.
* Create a priority table and set the source vertex equal to 0.
* Processing: Pop the root with the same distance from the priority queue (initially A)
* Should a shorter path be discovered, update the distances to its neighbors. Until the priority queue is empty, keep doing this.
* Repeat when the priority queue is empty.

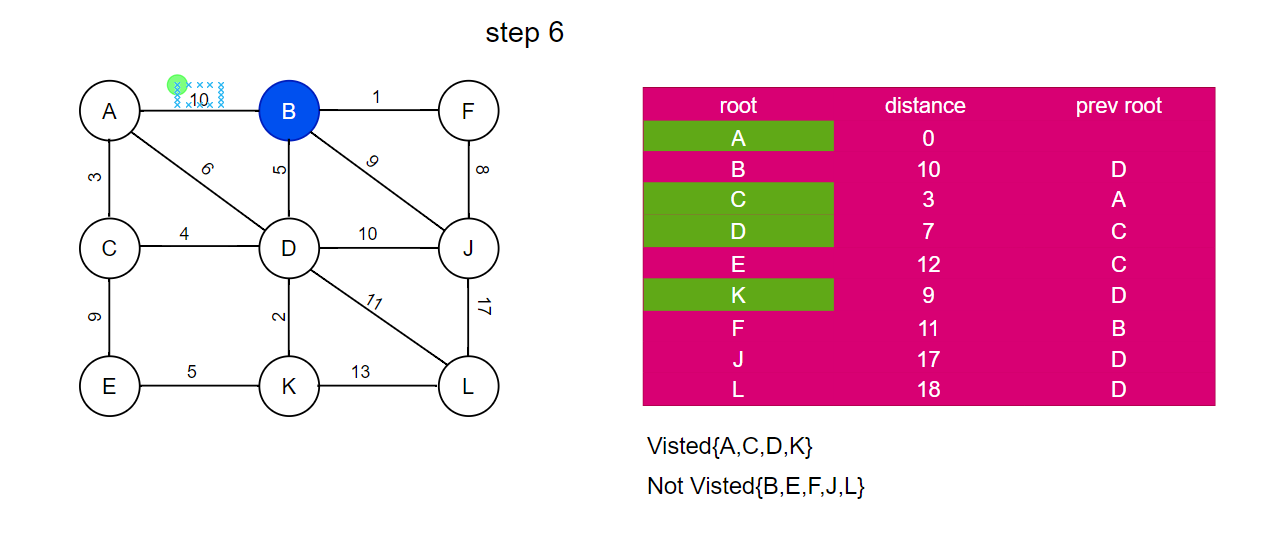


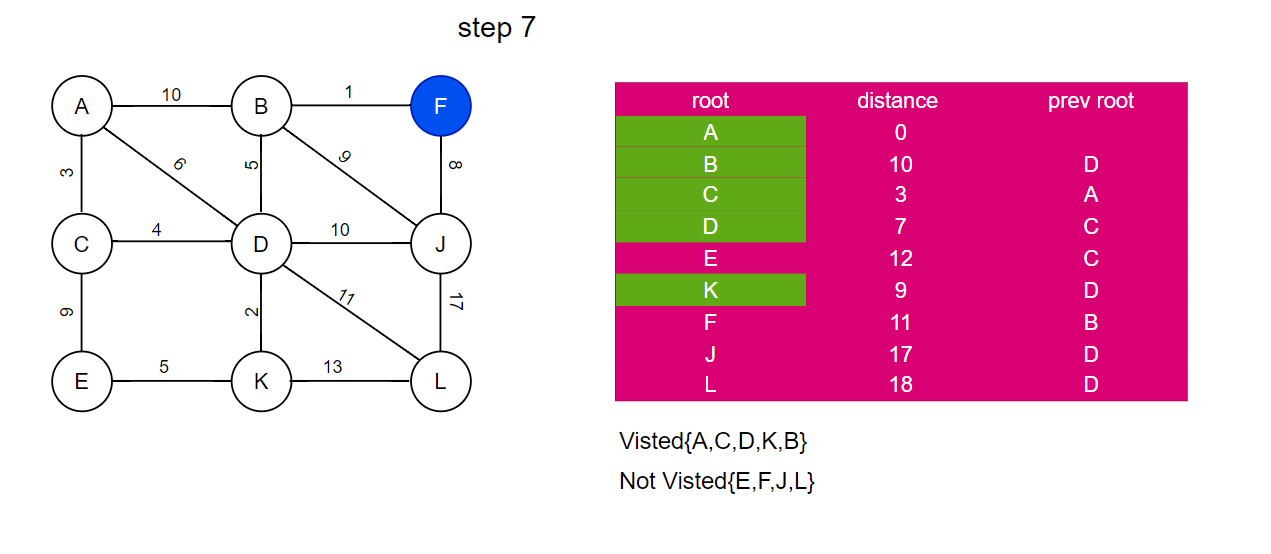


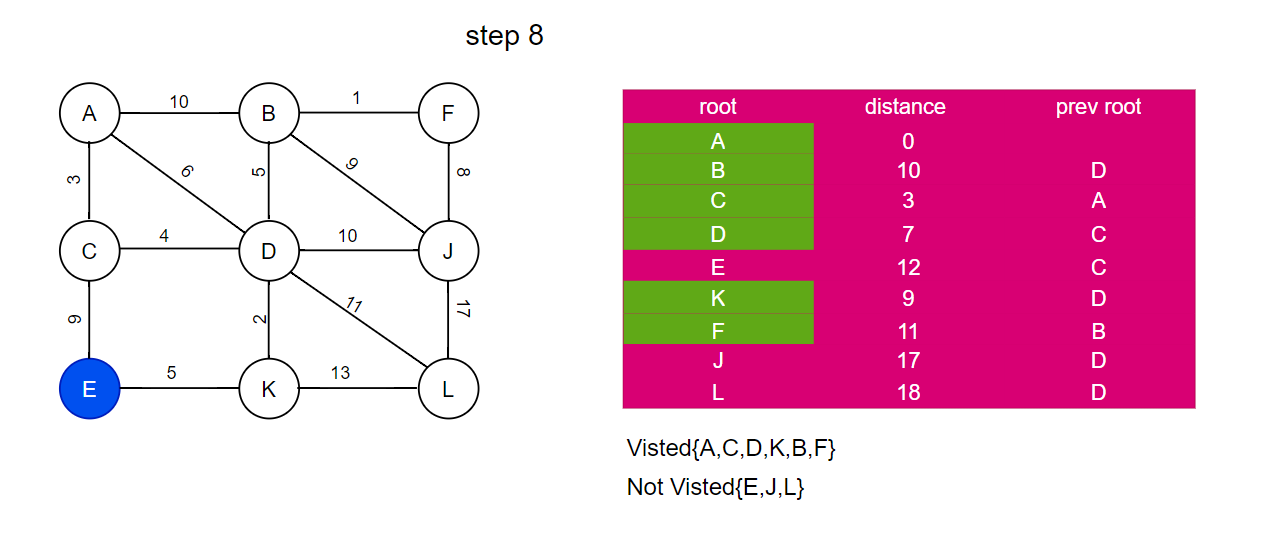


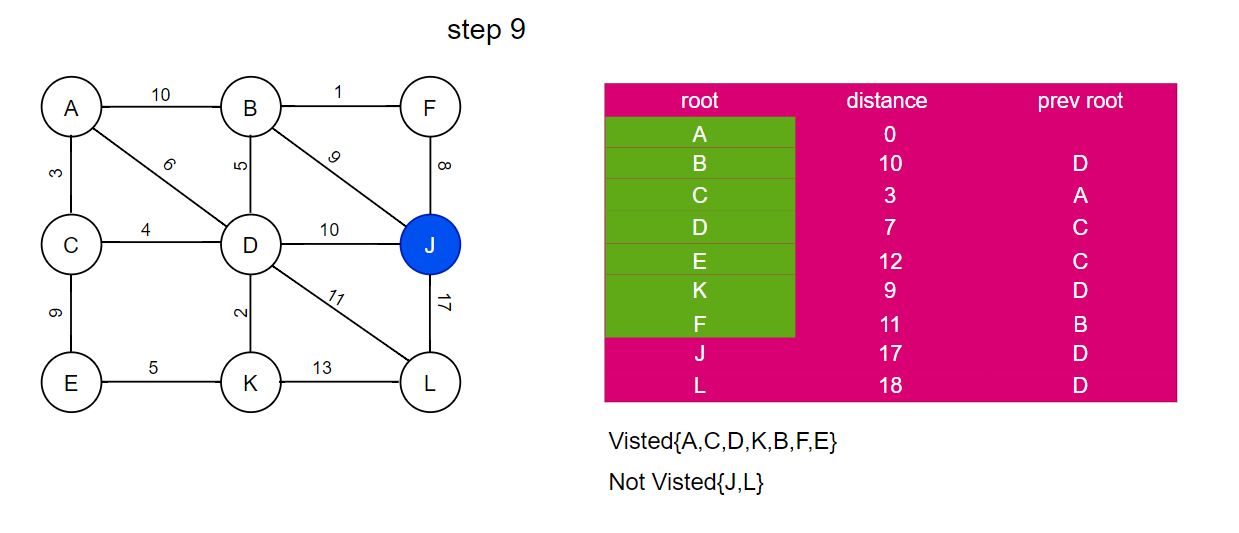


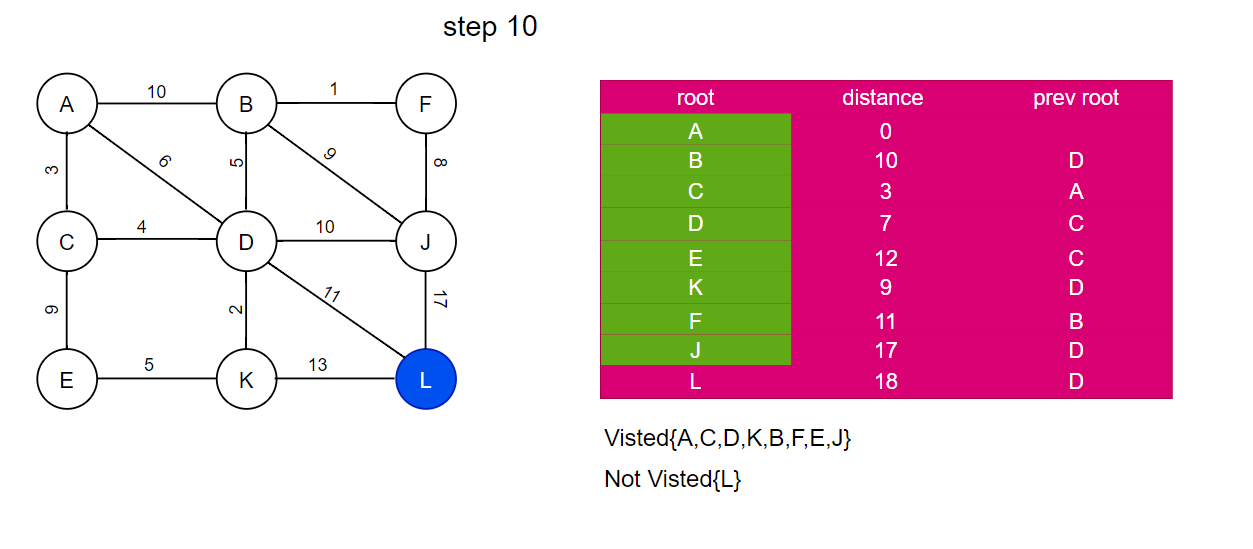


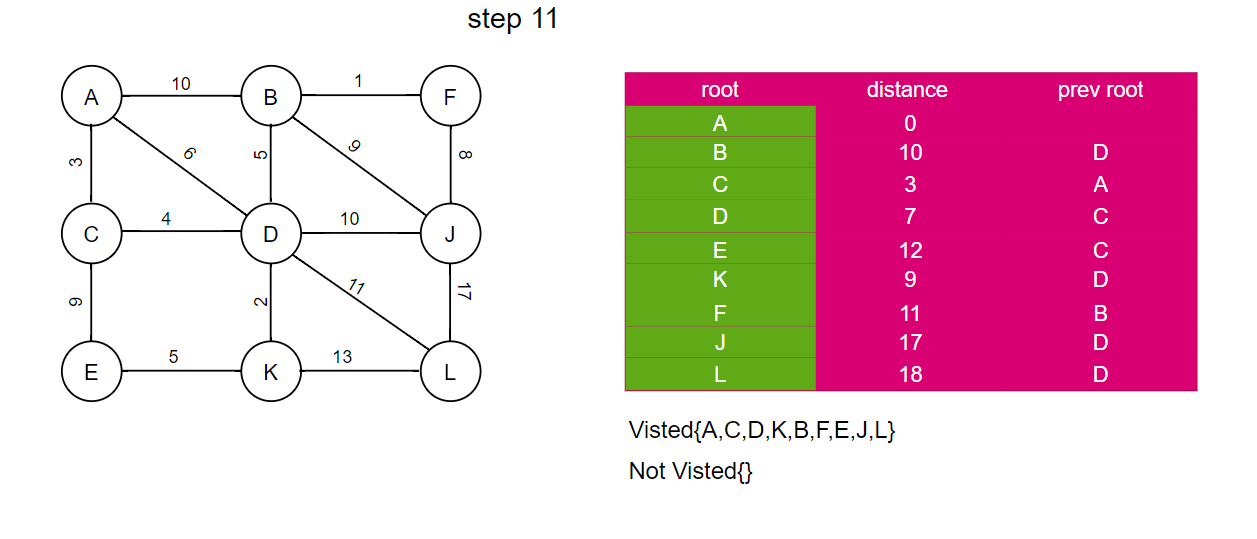


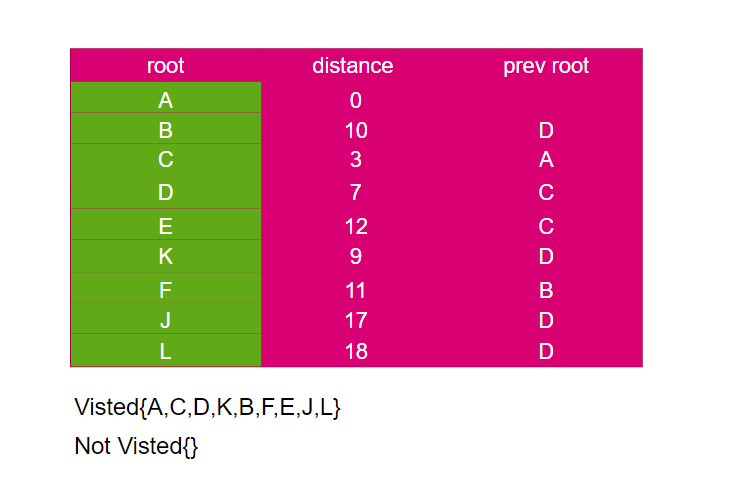






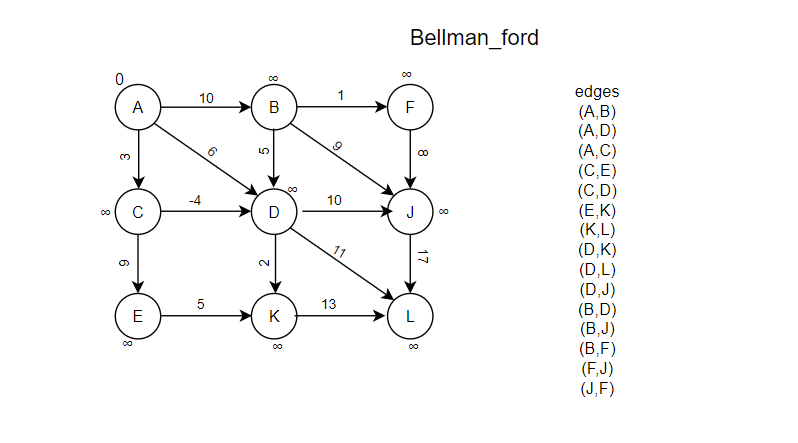






2- Bellman ford algorithm

* Initialization: Set the distance to the source root A to 0 and all others to infinity.
* Relaxation: (Where |V| is the number of vertices) Repeat |V| - 1 times. If a shorter path is discovered for any edge, update the distance to the destination vertex.
* Check for Negative Cycles: Take one more deep breath. There is a negative weight cycle if any distance is changed.



A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a diagram

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

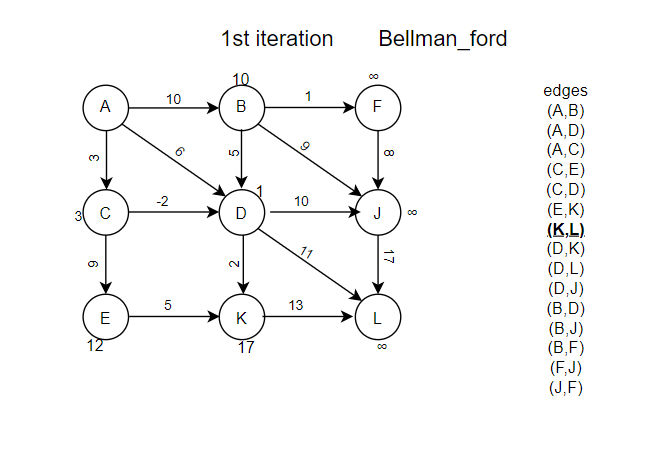
Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated



A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

A diagram of a network

Description automatically generated

### Q3-

# References

SearchAppArchitecture. (n.d.). *What is a Software Stack?* [online] Available at: <https://www.techtarget.com/searchapparchitecture/definition/software-stack>.

‌Sumo Logic (n.d.). *What is Encapsulation in OOP?* [online] Sumo Logic. Available at: <https://www.sumologic.com/glossary/encapsulation/>.

‌Quora. (n.d.). *What is a FIFO queue in Java?* [online] Available at: https://www.quora.com/What-is-a-FIFO-queue-in-Java [Accessed 8 Jun. 2024].

‌Stack Overflow. (n.d.). *Difference between binary search and binary search*

*tree?* [online] Available at: https://stackoverflow.com/questions/21586085/difference-between-binary-search-and-binary-search-tree [Accessed 8 Jun. 2024].

‌www.javatpoint.com. (n.d.). *Merge Sort - javatpoint*. [online] Available at: <https://www.javatpoint.com/merge-sort>.

‌Patrao, B. (2023). *Dijkstra’s vs Bellman-Ford Algorithm*. [online] Medium. Available at: https://medium.com/@brianpatrao1996/dijkstras-vs-bellman-ford-algorithm-383e4771c2cb.

‌